



IPR MEDIA
ИЗДАТЕЛЬСТВО

EDP HUB
ИЗДАТЕЛЬСТВО

П.И. Меликов



ИЗУЧАЕМ ОСНОВЫ PYTHON

ПРАКТИЧЕСКИЙ КУРС ДЛЯ ДАТА-АНАЛИТИКОВ

П.И. Меликов

**ИЗУЧАЕМ ОСНОВЫ PYTHON.
ПРАКТИЧЕСКИЙ КУРС
ДЛЯ ДАТА-АНАЛИТИКОВ**

**Москва
Ай Пи Ар Медиа
2023**

**Алматы
EDP Hub (Идипи Хаб)
2023**

УДК 004.432
ББК 32.973.2
М47

Автор:

Меликов П.И. — руководитель отдела портативного оборудования
неразрушающего контроля ООО «НПК «ТЕХНОВОТУМ»»,
магистр по направлению 15.04.04 «Автоматизация технологических
процессов и производств»

Рецензенты:

Снежко В.Л. — доктор технических наук, профессор,
заведующий кафедрой систем автоматизированного проектирования
и инженерных расчетов Российского государственного
аграрного университета — МСХА имени К.А. Тимирязева;
Зотов А.А. — доктор технических наук, профессор, профессор кафедры
проектирования и прочности авиационно-ракетных и космических изделий
Московского авиационного института (МАИ)

Меликов, Павел Ильич.

М4 Изучаем основы Python. Практический курс для дата-аналитиков / П.И. Меликов. — Москва : Ай Пи Ар Медиа ; Алматы : EDP Hub (Идипи Хаб), 2023. — 480 с.
ISBN 978-5-4497-2162-4 (Ай Пи Ар Медиа)
ISBN 978-601-81002-1-5 (EDP Hub (Идипи Хаб))

Издание представляет собой практический курс, состоящий из 13 разделов (12 из которых интерактивные — для среды JupyterLab). Курс содержит введение в основы языка Python с дальнейшим упором на аналитику данных (работа с наборами данных, статистика, язык SQL, метрики и машинное обучение). Формат материала прост для понимания, а каждый новый раздел включает в себя набор необходимых файлов, ссылок и практических заданий.

Предназначено для широкого круга читателей, интересующихся вопросами программирования и аналитики данных на Python.

Благодаря рассмотрению основ языка Python потенциальной аудиторией курса могут являться обучающиеся, ранее не изучавшие программирование.

ISBN 978-5-4497-2162-4
ISBN 978-601-81002-1-5

© Меликов П.И., 2023
© ООО Компания «Ай Пи Ар Медиа», 2023
© TOO «EDP Hub (Идипи Хаб)», 2023

СОДЕРЖАНИЕ

https://t.me/it_boooks/2

ВВЕДЕНИЕ	8
-----------------------	----------

1. PYTHON ДЛЯ АНАЛИЗА ДАННЫХ.

ОСНОВЫ И ИНСТРУМЕНТЫ.....	11
----------------------------------	-----------

Базовые инструменты аналитики.

Настройка окружения.....	11
---------------------------------	-----------

Работа с Jupyter и Anaconda Navigator	14
---	----

Взаимодействие с Python в Jupyter-notebook.....	17
---	----

Интерактивный режим интерпретатора Python.....	18
--	----

Менеджеры пакетов pip и conda.....	23
------------------------------------	----

Работа в PyCharm	25
------------------------	----

Среда для выполнения занятий по данному курсу (JupyterLab)	32
---	----

Синтаксис, типы, функции, цикл while

и условные выражения.....	35
----------------------------------	-----------

Математика в Python	37
---------------------------	----

Порядок операций	39
------------------------	----

Комментарии.....	39
------------------	----

Переменные.....	42
-----------------	----

Строки	43
--------------	----

Цикл while	49
------------------	----

Логические (или булевы) выражения	52
---	----

Условные инструкции	53
---------------------------	----

else и elif-операторы «когда это не так»	54
--	----

Отступы.....	56
--------------	----

Функции.....	58
--------------	----

Параметры и возвращаемые значения — взаимодействие с функциями.....	59
--	----

Лямбда-функции	60
----------------------	----

Программа «Калькулятор».....	61
------------------------------	----

Пользовательские функции (свои собственные функции).....	65
---	----

Передача параметров функциям	67
------------------------------------	----

Модернизация программы «Калькулятор»	68
Хитрые способы передачи параметров	70
Наборы данных, цикл for, объектно-ориентированное программирование, работа с файлами и исключения.....	72
Кортежи, списки, словари и массивы.....	73
Цикл for	86
Функция range().....	88
Пример. Создание функции «меню».....	92
Наша первая «игра».....	93
Улучшение игры	98
Классы	99
Файловый ввод-вывод	111
Обработка исключений (ошибок)	118
Стандартные библиотеки	128
Интерфейс операционной системы.....	129
Поиск файлов на основе подстановочных данных (Wildcards).....	132
Аргументы командной строки	133
Соответствие строковому шаблону (регулярные выражения)	136
Математика, случайные числа и статистика	141
Доступ в Интернет.....	144
Даты и время	145
Сжатие данных	146
Измерение производительности	147
Контроль качества кода	148
Многопоточность и многопроцессорность	150
2. РАБОТА С МАССИВАМИ ДАННЫХ	157
Библиотека NumPy	157
Создание массивов.....	158
Структурированные массивы.....	163
Печать массивов	165
Базовые операции.....	167

Индексы, срезы, итерации.....	170
Манипуляции с формой.....	174
Объединение массивов.....	178
Разбиение массива.....	179
Копии и представления.....	179
Маскированные массивы.....	182
Библиотека pandas	184
Знакомство с библиотекой pandas.....	185
Структура данных Series.....	186
Структура данных DataFrame.....	191
Доступ к данным в структурах pandas.....	194
Pandas и отсутствующие данные.....	205
3. СТАТИСТИКА В PYTHON	213
Статистика в NumPy и в Statistics.	
Введение в Scipy.Stats	213
Статистические функции в NumPy.....	213
Некоторые статистические функции из Statistics.....	221
Статистика при помощи модуля Scipy.Stats.....	226
Scipy. Примеры статистики, z-статистика и p-value	243
Генерация случайных чисел.....	243
Проблема отсутствия явно указанных параметров формы.....	258
Анализ принадлежности ряда значений к конкретному закону распределения.....	260
Куртозис и сдвиг (моменты и прочие понятия статистики).....	263
Диаграмма размаха.....	265
Примеры статистики из жизни.....	268
4. SQL И МЕТРИКИ ДАННЫХ	296
SQL — язык для работы с базами данных (на примере SQLite)	296
Немного о базах данных.....	296
SQLite.....	297

Выполнение запросов к базе данных	300
Курсор в базе данных	301
Оператор CREATE TABLE и таблицы в базе данных.....	302
Оператор INSERT.....	302
Создание именованного датафрейма из таблицы SQLite.....	304
Операторы и функции SQL (для аналитики данных и вычисления метрик).....	307
Типы данных (полей) в SQLite.....	320
Метрики (на примере ключевых продуктовых метрик)	323
Бизнес-метрики для продаж	323
Подключение библиотек, загрузка и предобработка данных	324
Метрика «ежемесячный доход».....	330
Метрика «ежемесячный темп роста дохода»	333
Метрика «ежемесячный доход по странам».....	335
Метрика «ежемесячная активность клиентов».....	339
Метрика «ежемесячное количество заказов».....	341
Метрика «ежемесячный средний доход за заказ»	343
Метрика «коэффициент новых клиентов»	344
Метрика «ежемесячная ставка удержания»	352
Метрика «скорость оттока клиентов».....	356
Метрика «коэффициент удержания когорт»	357
 5. МАШИННОЕ ОБУЧЕНИЕ (АНАЛИТИКА БОЛЬШИХ ДАННЫХ).....	 367
TensorFlow Keras. Решение математических задач	367
Рассмотрим библиотеку TensorFlow и ее надстройку Keras.....	367
Пример 1. $Y = 3X + 1$ (основы работы с TensorFlow Keras: нейронные сети, их функции активации, веса, смещения и типы ошибок)	371
Пример 2. $Y = 2X$	385
Пример 3. $Y = \sin(X)$	392
Пример 4. $Y = (K + X) \times X$	396
Настройки оптимизатора	400

TensorFlow Keras. Классификация изображений	403
Создание нейросети и использование датасета MNIST	403
Подготовка и классификация пользовательской картинки	422
Сделаем выводы	426
Библиотека Scikit-learn.	
Обучение с учителем и без учителя	429
Знакомство с библиотекой Scikit-learn	429
Обучение с учителем	430
Обучение без учителя	431
Функциональность Scikit-learn	432
Кластерный анализ (общий обзор, К-средних, линейная регрессия и деревья решений)	433
БУДЕТ ПОЛЕЗНО	463

ВВЕДЕНИЕ

Данное издание представляет собой практико-ориентированный курс, состоящий из 13 разделов, каждый из которых содержит общую часть и самостоятельную часть (проверочные задания и вопросы).

Курс рекомендуется для широкого круга читателей, интересующихся вопросами программирования и аналитики на Python, включая такие вопросы как: работа с наборами данных, статистика, язык SQL, метрики и машинное обучение.

Благодаря рассмотрению основ языка Python, потенциальной аудиторией могут являться читатели, ранее не изучавшие основы программирования.

В курсе используется форматирование специальных слов, к примеру как у слова Python, которое упрощает визуальное восприятие ключевых слов.

Первый раздел вводный — в нем рассмотрен процесс подготовки окружения. Остальные являются интерактивными и представляют собой `ipynb`-ноутбуки, выполняемые локально в среде JupyterLab, в которой можно быстро изменять и выполнять программный код, а также проводить с ним любые эксперименты.

К курсу прилагается весь необходимый для обучения набор файлов (скачать его с Яндекс.Диска можно по ссылке [1] в списке полезных источников на стр. 463).

На момент разработки курса в него были включены только наиболее актуальные и востребованные материалы, такие как основы работы со средами программирования Jupyter и PyCharm, менеджером пакетов и окружения conda, базовые стандартные модули языка Python (например, `os`, `re`, `multiprocessing`), работа с массивами при помощи модулей NumPy и Pandas, статистика при помощи Statistics и SciPy, работа с базами данных на примере SQLite, вычисление ключевых метрик, а также машинное обучение при помощи TensorFlow и Scikit-learn.

Материал апробирован более двух раз на группах численностью не менее 10 человек каждая, вследствие чего внесены дополнения и исправления.

Что ждет вас в конце изучения курса? После знакомства с вводной частью вы будете знать:

- основные инструменты для работы с языком Python и аналитики данных: PyCharm; Anaconda Navigator, Jupyter, conda и иные;

- основы языка программирования Python: базовый синтаксис, типы данных, функции, циклы, условные выражения, кортежи, списки, словари и массивы, классы, наследование, указатели и словари классов, модули, файловый ввод-вывод, обработку исключений;

- функциональность стандартных модулей/библиотек, поставляемых совместно с интерпретатором Python, а также основные принципы практической работы с ними;

- функциональность, доступные методы и объекты библиотек NumPy и pandas, а также основные принципы практической работы с ними;

- функциональность, доступные методы и объекты статистики библиотек/модулей NumPy, Statistics и Scipy.Stats, а также основные принципы практической работы с ними;

- функциональность, доступные методы и объекты статистики библиотеки Scipy: генерация случайных чисел и параметры формы распределения;

- принадлежность ряда значений к конкретному закону распределения (на основе параметров формы, куртозиса и др.);

- методику построения различных типов диаграмм при помощи библиотек/модулей Matplotlib и Plotly, в том числе диаграмм размаха;

- примеры описательной статистики из реального мира;

- язык SQL на примере работы с системой управления базой данных (СУБД) SQLite, а также основные принципы практической работы с базами данных как с одним из возможных типов источников получения информации (в том числе для расчета бизнес-метрик);

- практические примеры бизнес-метрик;

- north star метрики (т.е. метрики «полярной звезды»);

- когортный анализ (аналитика на основе характеристик действий пользователей/клиентов);

- основы работы с библиотекой для машинного обучения TensorFlow и надстройкой над ней — Keras;

- основы нейронных сетей и практические аспекты работы с ними;
 - методы решения простейших математических задач при помощи нейронных сетей;
 - методику написания программ для распознавания цифр на базе TensorFlow Keras;
 - принципы создания и тестирования нейросетей и их обучения по базе данных MNIST;
 - функциональность, доступные методы и объекты библиотеки Scikit-learn, а также основные принципы практической работы с ними для реализации математических алгоритмов;
 - основные положения машинного обучения с учителем и без учителя: метод k-средних, линейная регрессия и деревья решений.
- А после выполнения всех практических заданий научитесь:
- устанавливать и настраивать инструменты для работы с языком Python и аналитики данных;
 - разрабатывать и запускать программный код, включающий в себя использование полученных знаний;
 - разрабатывать новые программные решения (алгоритмы, методы и иные);
 - строить новые гипотезы и проверять их на практике.

Автор выражает глубокую признательность за поддержку при написании данной книги генеральному директору ООО «НПК «ТЕХНОВОТУМ»» А.М. Слядневу, а также следующим сотрудникам ФГБОУ ВО «МГТУ «СТАНКИН»»: д-ру техн. наук, доц. Р.А. Нежметдинову (также директор Единого Деканата); канд. техн. наук, доц. А.П. Никищечкину; канд. техн. наук, доц. С.В. Соколову.

Отдельная благодарность выражается Семену Лукашевскому и другим авторам, публикующим образовательную информацию в сети Интернет в свободном доступе.

PYTHON ДЛЯ АНАЛИЗА ДАННЫХ. ОСНОВЫ И ИНСТРУМЕНТЫ

БАЗОВЫЕ ИНСТРУМЕНТЫ АНАЛИТИКИ. НАСТРОЙКА ОКРУЖЕНИЯ

https://t.me/it_boooks/2

Цели выполнения работы

Получение представления о существующих инструментах для работы с языком Python и аналитики данных. Изучение основных принципов практической работы с ними.

Порядок выполнения работы

Данный раздел является вводным и в нем будут рассмотрены основные программные инструменты аналитики данных, которые используются при изучении рассматриваемого в рамках курса материала.

Перед исследованием основных принципов работы с ними нам нужно их установить. Для этого нам потребуется скачать две программы: PyCharm и Anaconda, скачать которые можно по ссылкам, указанным в источниках [2]¹ и [4] соответственно. Обе программы

¹ Вся работа по данной книге была выполнена в версии Anaconda 2021.11 (Nov. 17, 2021). По вашему усмотрению вы можете скачать эту же версию из архива (см. источник [3]) во избежание непредвиденных трудностей.

доступны бесплатно для индивидуального использования как для операционных систем семейства Windows, так и для Linux. Произведите их скачивание и установку (после установки не запускайте их, пока далее не будет написано запустить).

Ниже представлено краткое описание этих программ и их целевое назначение в рамках данного курса:

1. PyCharm — интегрированная среда разработки (или на англ. IDE — *Integrated Development Environment*) для разработки на языке Python. Позволяет эффективно разрабатывать и отлаживать (исправлять ошибки в коде или анализировать его эффективность) сложные программы.

2. Anaconda Navigator — программа, представляющая собой дистрибутив (набор) популярных свободных библиотек для аналитики данных на языках Python и R². Основная цель — поставка единым комплектом наиболее тематических модулей (таких как NumPy, SciPy и других модулей, в том числе и рассматриваемых в рамках данного учебного пособия) с разрешением возникающих зависимостей и конфликтов, которые неизбежны при одиночной установке модулей.

Так как каждый ваш проект может требовать разные версии одних и тех же модулей (библиотек функциональности), то необходимо иметь возможность установки в системе нескольких версий одних и тех же пакетов.

Основная особенность Anaconda Navigator заключается в том, что она является графической надстройкой над консольным/терминальным менеджером разрешения зависимостей conda, и так как она (Anaconda Navigator) ориентирована именно на аналитику данных, то уже имеет в своем составе предустановленные наиболее часто используемые библиотеки при аналитике данных.

Стоит отметить, что для установки библиотек существуют и другие стандартные менеджеры пакетов, например менеджер пакетов pip для Python. Далее мы еще рассмотрим принципы работы как conda, так и pip, но на текущий момент важно отметить, что conda включает в себя кроме менеджера пакетов еще и менеджер виртуального окружения [5], в то время как pip — это только ме-

² R — язык программирования для статистической обработки данных и их отображения.

неджер пакетов [6] (но существует возможность управления виртуальными окружениями отдельно, например при помощи модуля виртуальных окружений `venv`).

Виртуальное окружение используется для управления набором модулей, которые подключаются к вашей программе. Одна программа использует одни пакеты, вторая — другие, а третья может использовать все сразу, но других версий. В реальном (системном) окружении вашей операционной системы может существовать только одна версия библиотек, вы не можете установить несколько сразу. Удалять и устанавливать пакеты каждый раз при работе с разными задачами — неэффективный подход. Для решения этой задачи и существуют менеджеры виртуальных окружений.

Если провести аналогию, то виртуальное окружение — это как отдельные Word-документы, которые в целом являются одной и той же сущностью, но имеют разный текст с разными параметрами форматирования (цвет и т.п.). То есть вы создаете под каждый свой проект отдельное виртуальное окружение, не «засоряя» системное окружение, и используете в проекте только нужные версии библиотек и интерпретатора Python (интерпретатор — это программа, выполняющая ваши программы) [7].

При помощи менеджера пакетов производится установка/обновление/удаление пакетов нужных версий в созданном и управляемом менеджером виртуальных окружений виртуальном окружении.

PyCharm является средой для программирования, в ней нет ориентированности на анализ данных, т.е. она не позволяет аналитику данных удобно изучать данные и сразу документировать результат анализа. Но PyCharm отлично выступает второй ступенью разработки аналитики данных, когда вы уже разработали каркас приложения для аналитики и вам нужно его масштабировать или повысить его эффективность.

В роли первой ступени аналитики данных обычно выступает работа в Jupyter-ноутбуках. Это интерактивный программный инструмент, работа с которым похожа на работу с программой Word, т.е. он позволяет разрабатывать форматированную текстовую документацию, но внутри которой можно создавать и выполнять программный код [8].

Средства Jupyter ориентированы непосредственно на аналитику данных, поэтому они встроены в Anaconda Navigator и устанавливать их отдельно не потребуется.

Работа с Jupyter и Anaconda Navigator

Необходимо запустить Anaconda Navigator (рис. 1). Слева доступен выбор вкладок, среди них две основные:

- Home — открыта по умолчанию (рис. 1), на ней можно запускать некоторые утилиты в выбранном окружении, например Jupyter-notebook;
- Environments — вкладка для управления окружениями и установки пакетов (рис. 2).

На вкладке Home запустите утилиту Jupyter-notebook. Откроется страница в браузере (рис. 3), при помощи которой вы сможете работать локально с Jupyter-ноутбуками. Далее мы рассмотрим, как это делать.

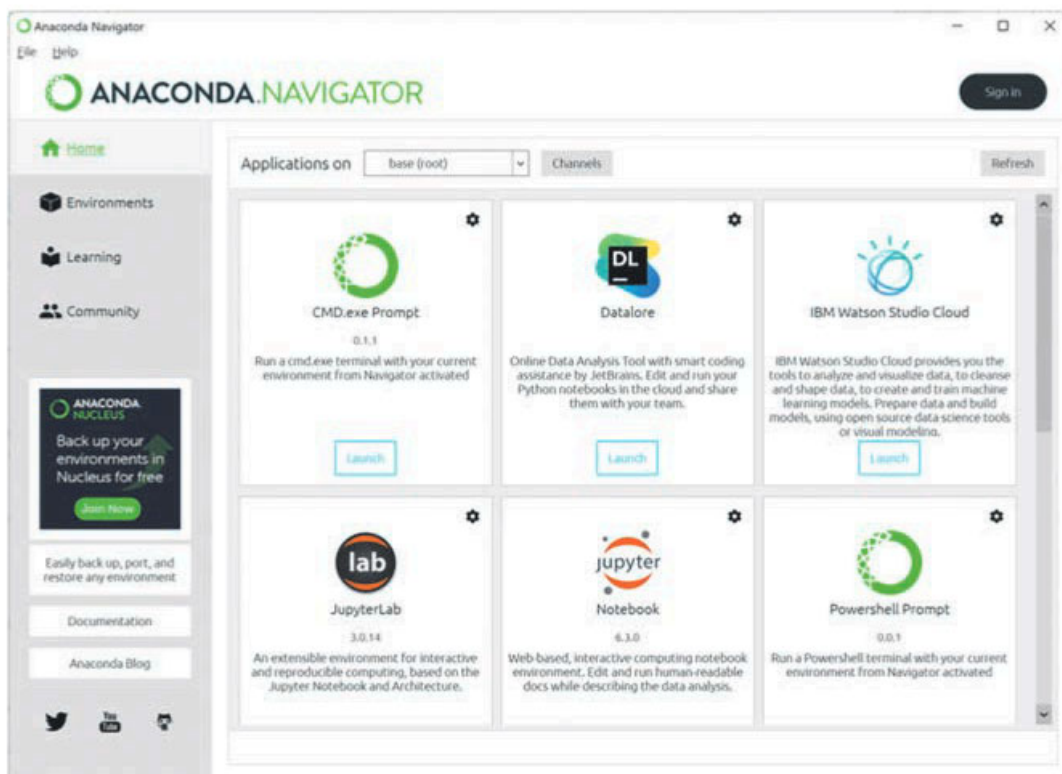
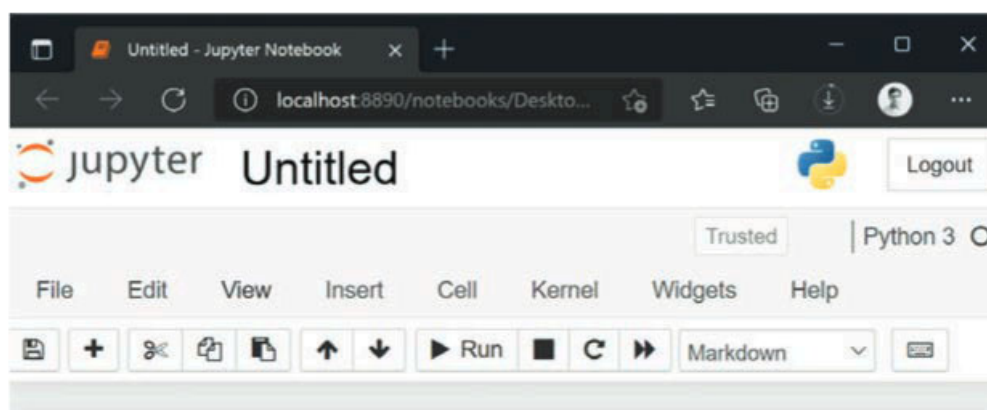


Рис. 1. Anaconda Navigator (вкладка Home)



Рис. 2. Anaconda Navigator (вкладка Environments)



Сложный анализ данных

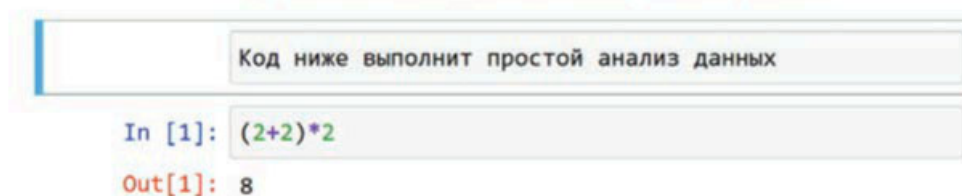


Рис. 3. Интерфейс Jupyter - notebook

Также существуют онлайн-сервисы, такие как, например, Google Colab, Azure Notebooks или Kaggle. Их интерфейс очень похож на интерфейс Jupyter-ноутбуков, и они позволяют их импортировать и выполнять. То есть доступно бесплатное использование некоторой части мощностей серверов Google или Microsoft, но работу в рамках данного курса мы будем выполнять именно в локально установленном Jupyter, потому что бесплатные онлайн-сервисы предоставляют скромные мощности, и работа в них происходит медленно.

Существует и утилита JupyterLab (рис. 4), которая является более современной реализацией Jupyter-ноутбуков. В основном отличия только в том, что добавлен многооконный режим и улучшен файловый проводник. Эта утилита тоже предустановлена в Anaconda Navigator.

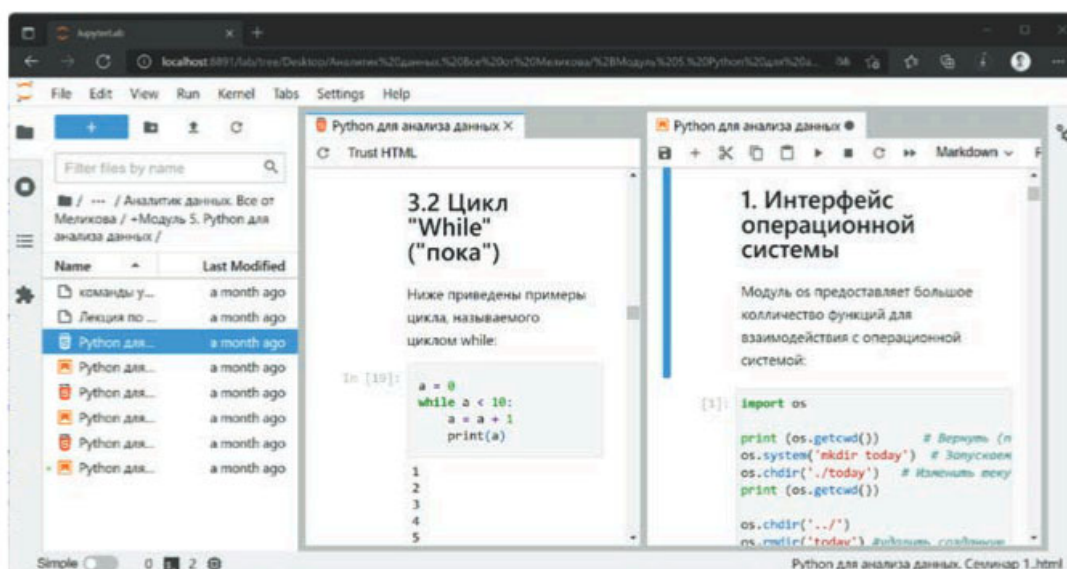


Рис. 4. Интерфейс JupyterLab

Если вам нужен Jupyter-notebook или JupyterLab для целой группы людей, например для отдела аналитики, то вы можете на отдельном сервере запустить JupyterHub, внешне он ничем не отличается, но позволяет работать нескольким пользователям одновременно на одном сервере, а также ограничивать доступные им ресурсы и предоставлять средства управления пользователями. Также абсолютно бесплатен.

Взаимодействие с Python в Jupyter-notebook

Jupyter-notebook — это среда разработки, где сразу можно видеть результат выполнения кода и его отдельных фрагментов. Отличие от традиционной среды разработки (например, PyCharm) в том, что код можно разбить на куски, выполнять их в произвольном порядке и оставлять отформатированные комментарии к коду.

В такой среде разработки можно, например, создать блок, содержащий «отрывок» исходного кода, и сразу проверить результат его работы (который выводится для каждого такого «куска» кода отдельно, сразу после него) без запуска программы целиком; можно очень просто и быстро изменять порядки блоков с кодом или добавлять форматированные блоки с текстовым описанием. Например, на рис. 3 приведены блок типа Markdown (это язык текстовой разметки) и один блок типа Code, содержащий Python-код, который будет выполняться.

На рис. 3 также можно увидеть, что слева от блока с кодом написано «In[1]:» (англ. *in* — «ввод»). Это означает «ввод кода», т.е. этот код будет передан интерпретатору Python, он его выполнит, и если у кода есть какой-то вывод, например вывод текста в результате своей работы, то тогда результат работы этого кода выведется под блоком кода в разделе «Out[1]:». В данном случае результат выполнения Python — это число 8.

Число 1 в квадратный скобках в «In[1]:» и «Out[1]:» — это порядковый номер выполнения. Например, если выполнить этот же блок второй раз, то номер изменится на 2. Данный номер ни на что не влияет. В рамках данного курса мы будем часто работать с блокнотами Jupyter (блокнот — *notebook* (англ.)), и в рассматриваемых примерах порядковый номер будет абсолютно случаен или вовсе вырезан, чтобы не отвлекать читателя.

Результат работы кода не обязательно может сопровождаться выводом текста, код может давать графики, таблицы, html-документы и др. И все перечисленное может быть выведено в секции «In[]» или «Out[]:».

Проработайте представленную ниже последовательность и изучите самостоятельно основные доступные функции в интерфейсе блокнота (рис. 3):

- создание нового блокнота;
- добавление Markdown и Code -ячеек (блоков) в блокнот;
- запуск ячеек при помощи кнопки Run;
- попробуйте ввести в Markdown-ячейку текст «# Сложный анализ данных» и выполнить. Вы увидите, что она отформатируется так же, как на рис. 3, и станет заголовком.

В рамках курса мы не будем отдельно рассматривать язык разметки Markdown, но он очень простой, и вы сможете изучить его пассивно при исследовании основ аналитики на Python, так как весь представленный в данной книге материал сопровождается готовыми блокнотами, которые вы можете выполнять при ее прочтении (и просматривать содержимое их ячеек).

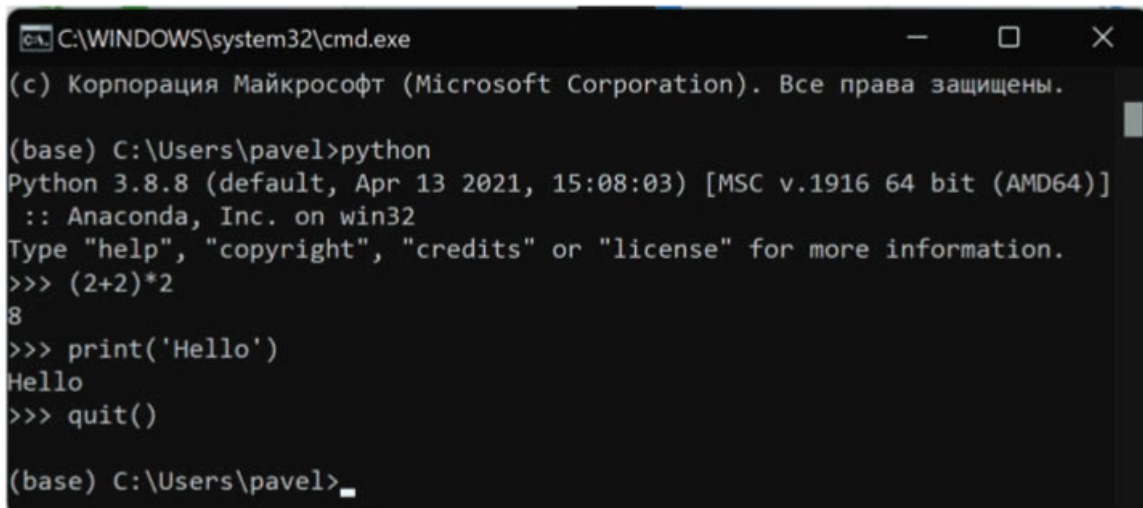
Интерактивный режим интерпретатора Python

Давайте подробнее закрепим понимание работы блокнотов Jupyter. В большей степени их алгоритм работы очень схож с интерактивным режимом работы интерпретатора языка Python.

При таком режиме вы запускаете интерпретатор, а далее он ожидает ввода строк Python-кода. Когда вы ввели код, например « $(2+2) * 2$ », нажали клавишу ввода (*enter* (англ.)), интерпретатор выполнит введенный код и выведет вам результат его работы. То есть для данного примера — число 8. Это продемонстрировано на рисунке ниже (рис. 5).

Для запуска и изучения интерактивного режима интерпретатора, как на рисунке выше (рис. 5), вы должны проделать следующие действия:

- в Anaconda Navigator во вкладке Home запустить `cmd.exe`;
- как на рис. 5 написать команду «python», запустится интерпретатор в интерактивном режиме;
- далее вы можете вводить любой Python-код, нажимать Enter, и интерпретатор будет его выполнять и выводить результат;
- для выхода из интерпретатора можете ввести команду «quit()» — это функция выхода, или просто закройте окно командной строки/терминала.



```
C:\WINDOWS\system32\cmd.exe
(с) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

(base) C:\Users\pavel>python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]
:: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> (2+2)*2
8
>>> print('Hello')
Hello
>>> quit()

(base) C:\Users\pavel>
```

Рис. 5. Интерактивный режим интерпретатора

Кроме интерактивного режима вы можете запускать при помощи интерпретатора файлы с Python-кодом, например, написав «python my_programm.py», если файл исходного кода my_programm.py находится в той же папке, которая сейчас активна в командной строке, например в «C:\Users\pavel». А создать файл программы my_programm.py, содержащий исходный Python-код, вы можете любым способом — просто создать его в обычном текстовом редакторе (блокноте) или в среде разработки PyCharm, получить его путем экспорта из Jupyter, скачать из Интернета и т.д.

Создавать каждый раз вначале своей аналитической деятельности большую программу в виде исходного кода, т.е. с расширением «.py», вам будет неудобно, так как в аналитике данных нужно оставлять много текстового описания к коду, потому что суть вашей работы концентрируется не в прикладном программировании, а в аналитике при помощи программирования.

Поэтому Jupyter-блокноты являются наиболее полезным инструментом для аналитики данных. Весь код из ноутбуков можно экспортировать через меню **File**, если вы хотите использовать его как готовую программу. А если вам необходима отчетность, то вы можете распечатать ваш ноутбук через это же меню, потому что блокнот представляет собой почти готовый отчет по аналитике.

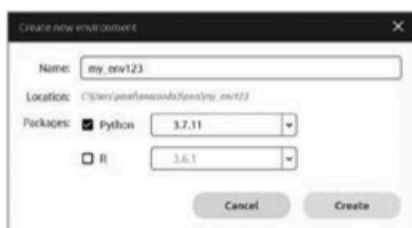
Посмотрим еще раз на рис. 5, на строчку «(base) C:\Users\pavel>python»:

- это командная строка/терминал вашей операционной системы;

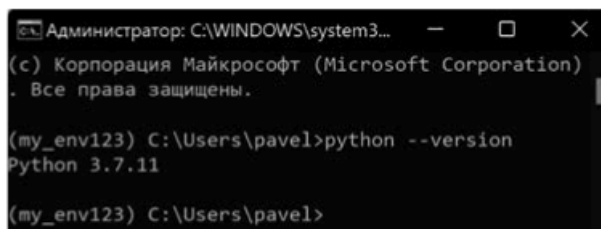
- когда командная оболочка вашей операционной системы ожидает ввода команд, она выводит текущее расположение, в котором вы находитесь. То есть в данном случае мы находимся по пути «C:\Users\pavel». Это сейчас не играет никакой роли, но при помощи командой оболочки операционной системы мы можем выполнять любые другие команды, отличные от запуска интерпретатора при помощи команды «python», например можем копировать файлы или получить их список;

- как мы видим, левее от пути выводится «(base)». Это уже и есть некоторая из удобных функций программы Anaconda Navigator. Префикс «(base)» означает, что в данной командной строке команды выполняются от окружения base. Это базовое системное окружение, создающееся в Anaconda Navigator. Ранее уже было сказано, для чего нужны окружения, а в частности — виртуальные окружения. Они необходимы для использования библиотек нужных вам версий в каждом из проектов.

Попробуйте на вкладке Environments программы Anaconda Navigator создать новое виртуальное окружение (заодно там же попробуйте установить любой Python-пакет), выбрав версию интерпретатора чуть более старую, чем предлагает Anaconda Navigator (для примера возьмем версию 3.7.11, рис. 6, а). Затем на вкладке Home выберите это окружение и запустите для него утилиту cmd.exe (предварительно потребуется ее установить). Вы увидите, что префикс окружения изменится (рис. 6, б).



а



б

Рис. 6. Создание окружения my_env123:

а — создание окружения; б — cmd.exe окружения my_env123

Также вы увидите, что на вкладке Home по-прежнему есть Jupyter, но теперь он существует просто с кнопкой «установить». То есть он не установлен. Дело в том, что Jupyter — это Python-пакет. Звучит сложно, но вы можете на вкладке Environments в списке установленных пакетов базового окружения увидеть, что Jupyter установлен, а в новом, созданном вами окружении, нет. Попробуйте установить его с вкладки Home (либо вручную с вкладки Environments) и увидите, что он появится в списке установленных пакетов на вкладке Environments, и вы сможете его запустить, уже используя новое окружение.

На самом деле глупо в каждое новое виртуальное окружение устанавливать свой собственный Jupyter, потому что Jupyter поддерживает работу с виртуальными окружениями (имеется в виду не работа внутри окружения, а то, что вы можете к Jupyter, например, установленному в базовом окружении, подключить новые виртуальные окружения и в каждом блокноте можно будет выбирать, в каком его окружении выполнять).

Чтобы ваше виртуальное окружение отображалось в Jupyter, запущенном в base (базовом системном окружении), необходимо:

- установить модуль `ipykernel` в нужное вам виртуальное окружение, например в созданное нами окружение `my_env123` на вкладке Environments. Это добавит в наше окружение поддержку ядра Jupyter;

- в `cmd.exe` окружения `my_env123` написать команду `python -m ipykernel install --user --name my_env123 --display-name «my_env123»`. Эта команда добавит наше окружение в список окружений для Jupyter. На рисунке ниже (рис. 7) показано, как выбирать окружение, в котором будет выполняться блокнот. Разница между окружениями в том, что в них разные версии интерпретатора, и это демонстрируется в ячейках. Первая ячейка запущена в базовом окружении (как и сам Jupyter), а вторая — в виртуальном окружении `my_env123`;

- для удаления виртуального окружения вы можете воспользоваться командой (выполненной из base-окружения или того окружения, из которого запускается Jupyter) `jupyter kernelspec uninstall my_env123`.

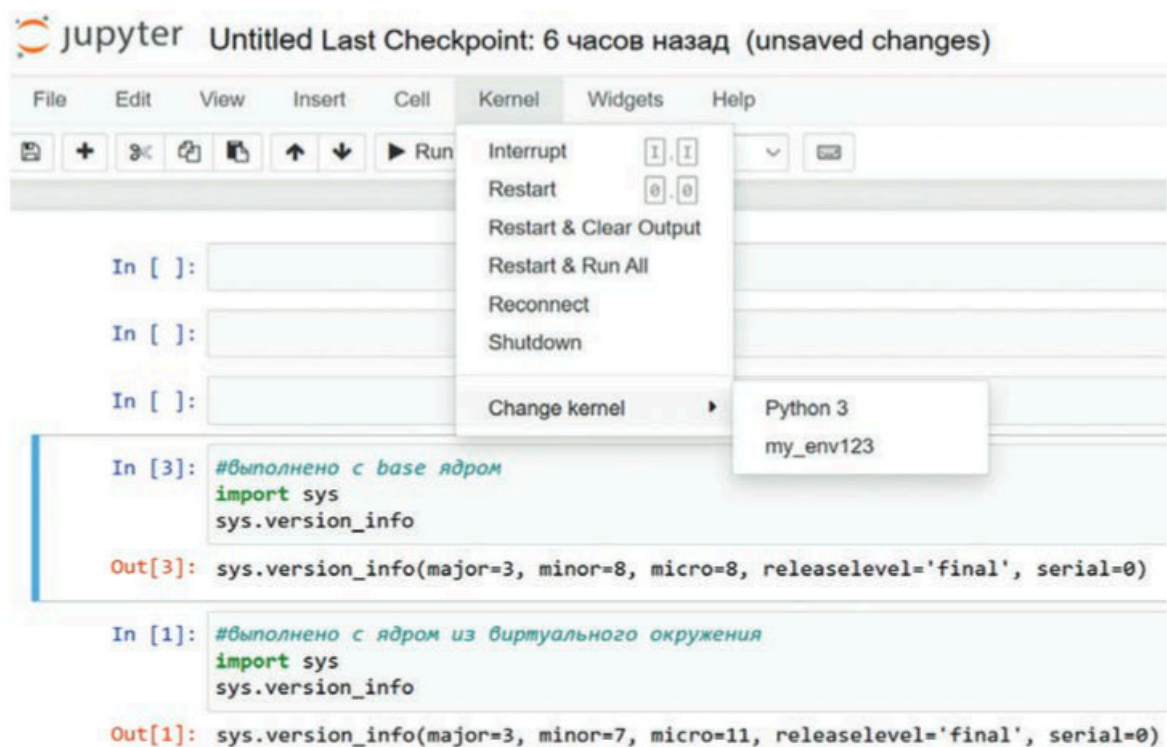


Рис. 7. Использование окружения my_env123 в Jupyter

В каждое виртуальное окружение вы можете устанавливать не только библиотеки разных версий, но и Python-интерпретатор нужной вам версии. Оба этих пункта являются важными, так как когда вы используете какие-либо сложные и очень функциональные библиотеки (например, библиотеку для машинного обучения TensorFlow), то они могут применять конкретные версии библиотек и интерпретатора (зачастую устаревшие), и если эти сложные библиотеки используют уже какую-либо определенную версию других библиотек, то будет сложно отдельно подключить в этом же окружении эти же уже вложенные библиотеки (они называются зависимостями), но других версий. Обычно об этом пишется в документации.

В целом, в рамках представленного в данной книге практического курса рассматриваются простые примеры, поэтому вы можете выполнять представленные примеры в базовом окружении и не создавать отдельных виртуальных окружений. Большая часть используемых в данной книге библиотек уже предустановлена в базовом окружении в Anaconda Navigator.

Стоит отметить, что на самом деле кроме базового системного окружения существует еще системное окружение пользователя. То есть вы можете установить в базовое окружение одни библиотеки, и они будут доступны всем пользователям, а в окружение пользователей — другие или эти же, но других версий. В этом случае, если в окружении пользователя не будет найдена нужная библиотека, тогда произойдет обращение к библиотекам системного окружения. Но для аналитики данных проще использовать виртуальные окружения Anaconda Navigator и не разбираться с созданием пользователей, ручной установкой пакетов именно в окружения пользователей и т.п. (потому что окружение пользователей слабо решает все задачи и в нем тоже лучше создавать виртуальные окружения).

Менеджеры пакетов `pip` и `conda`

Кроме менеджера пакетов `conda`, который мы использовали через графическую надстройку над ним Anaconda Navigator, очень часто применяется менеджер пакетов `pip`. У обоих менеджеров разные репозитории (хранилища, источники) пакетов. Скорость появления пакетов новых стабильных версий в разных репозиториях не является в рамках данного курса значимой, поэтому можно утверждать, что и там, и там доступны одни и те же пакеты.

Разница заключается в следующих **аспектах**:

- `conda` включает в себя менеджер виртуальных окружений, а `pip` — нет (это уже было сказано ранее);
- различные способы разрешения зависимостей;
- `pip` устанавливает только пакеты Python (и сам `pip` тоже является Python-пакетом), тогда как `conda` устанавливает пакеты, которые могут содержать программное обеспечение, написанное на любом языке.

Разрешение зависимостей необходимо в ситуации, когда библиотека 1 использует другую зависимую библиотеку 2, например, версии больше или равной 2.0 (в библиотеках обычно прописывается именно условие «больше», а не условие «строго равно версии...»), а библиотека 3 также применяет библиотеку 2, но версии не новее 2.5. При наличии средства разрешения зависимости бу-

дет установлена библиотека 2 любой доступной версии из диапазона [2.0; 2.5].

При установке пакетов `pip` устанавливает зависимости последовательно. Не прилагается никаких усилий для обеспечения того, чтобы зависимости всех пакетов выполнялись одновременно. Это может привести к тому, что работоспособность пакетов в виртуальных средах будет снижена, если пакеты, установленные ранее, имеют несовместимые версии зависимостей по сравнению с пакетами, установленными позже.

В менеджере `conda` используется средство разрешения зависимостей для проверки, что все требования всех пакетов, установленных в среде, выполнены. Эта проверка может занять дополнительное время, но помогает предотвратить создание сломанной виртуальной среды.

Перед использованием `pip` интерпретатор Python должен быть установлен через менеджер системных пакетов или путем загрузки и запуска установщика. `Conda`, с другой стороны, может устанавливать пакеты Python, а также интерпретатор Python напрямую.

`Anaconda Navigator` использует по умолчанию пакеты из данного репозитория, доступного в сети «Интернет» (см. источник [9]), а `pip` — из репозитория (см. источник [10]). Вы можете зайти на данные веб-ресурсы и изучить их содержимое. Некоторые из часто необходимых возможностей: изучить список доступных версий, следить за появлением новых версий или перейти на сайт разработчика пакета. Также на обоих веб-ресурсах в качестве подсказки отображается, как установить просматриваемый пакет через командную оболочку вашей операционной системы, например, пакет `tensorflow` (используется в рамках данного издания) можно установить следующим образом:

```
pip install tensorflow
или
conda install -c anaconda tensorflow
```

Как пользоваться этими менеджерами пакетов, рекомендуется изучить в сети Интернет. Это не должно вызвать большие сложности ввиду того, что синтаксис их команд очень простой. В приме-

ре выше установится пакет `tensorflow` самой последней версии (так происходит по умолчанию), потому что мы не указали, какая версия нам нужна. Далее мы можем удалять или обновлять пакеты.

Вы также можете устанавливать пакеты прямо через Jupyter-блокнот, как это делается, можно изучить, например, по источнику [11].

В рамках данного курса будет достаточно устанавливать пакеты через вкладку `Environments` программы `Anaconda Navigator` и не прибегать к использованию `conda`, `pip` или каким-либо другим способам³.

Работа в PyCharm

Запустите PyCharm. При первом запуске вам выведется окно с названием `Import PyCharm Settings` (с англ. — «импортировать PyCharm-настройки»), откажитесь от импорта настроек. Откроется окно «Welcome to PyCharm», нажмите «New Project» (с англ. — «создание нового проекта»).

Перед вами появится окно настроек нового проекта, как показано на рисунке ниже (рис. 8).

В данном окне вы можете:

- установить или оставить по умолчанию местоположение вашего проекта (`Location`);
- выбрать виртуальное окружение и его интерпретатор. При первом запуске PyCharm (рис. 8) предлагает вам во вкладке выбора интерпретатора (`Base interpreter`) скачать интерпретатор с официального сайта Python;
- выбрать, нужно ли создать файл `main.py` (в любом проекте логично наличие хотя бы одного файла исходного кода, поэтому рекомендуется не убирать галочку, чтобы этот файл создался и вам не пришлось делать это вручную и указывать в нем точку входа).

³ При установке через Jupyter вы можете столкнуться с тем, что окружение виртуальное и пакеты в нем установлены извне или что окружение не настроено и установка пакетов через Jupyter внешне работает, но по факту пакеты невозможно подключать. Об этих неприятных вещах указано в источнике [11] в разделе “The Details: Why is Installation from Jupyter so Messy?” (с англ. — «Подробности: Почему установка через Jupyter такая беспорядочная?»).

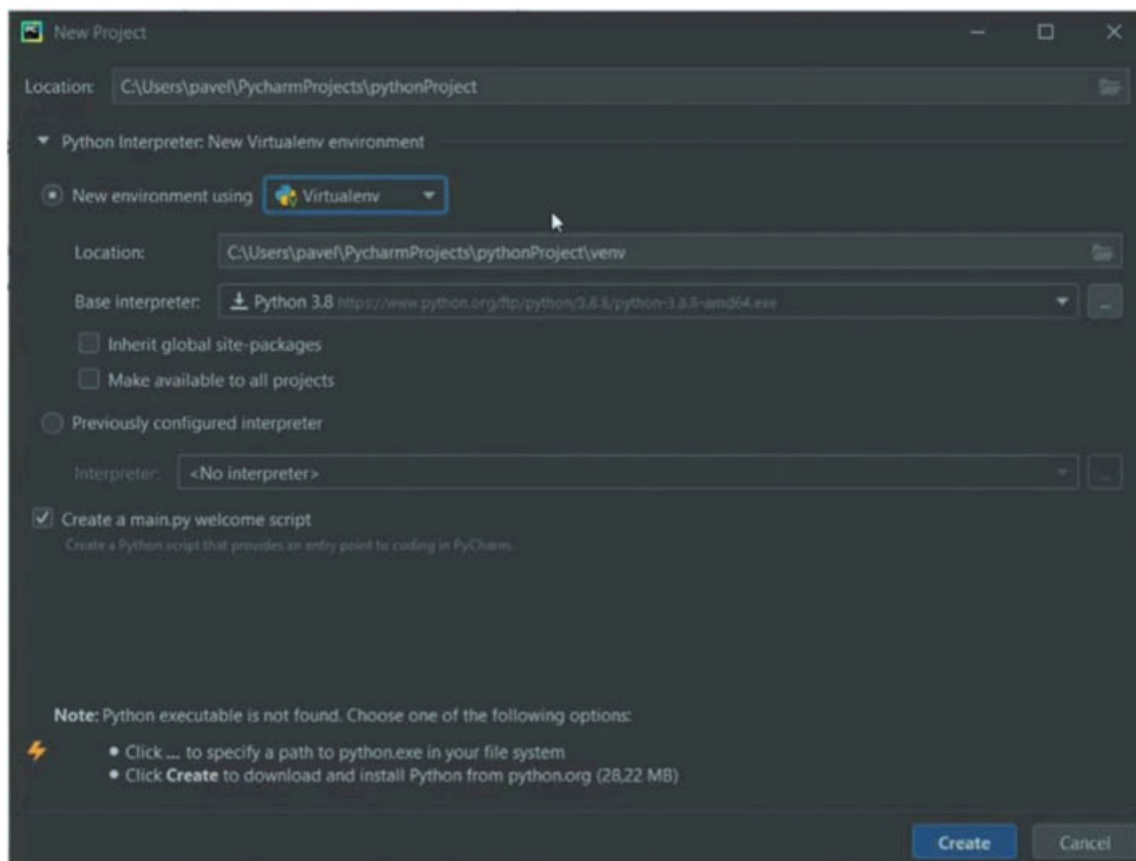


Рис. 8. Создание нового проекта в PyCharm

Ничего не меняйте в настройках создаваемого проекта и создайте его как есть. Вместо менеджера виртуального окружения conda в проекте по умолчанию будет использован менеджер виртуальных окружений venv, в этом нет ничего критичного. Давайте сначала изучим основы PyCharm, а уже далее рассмотрим, как установить уже известный нам менеджер conda.

После создания проекта он будет открыт и перед вами будет представлено окно, изображенное на рисунке ниже (рис. 9). Справа вверху есть зеленая кнопка запуска, нажмите ее, и выполнится по умолчанию созданная программа `main.py`.

Результат выполнения данной программы выводится в окно вывода внизу экрана, и в данном случае единственное, что делает программа, это выводит строчку «Hi, PyCharm».

По центру экрана выводится исходный код программы `main.py`, который вы можете изменять. Попробуйте в коде найти слово

«PyCharm», заменить его на что-то другое и запустить программу. Вы должны увидеть отображение введенного вами текста в окне вывода.

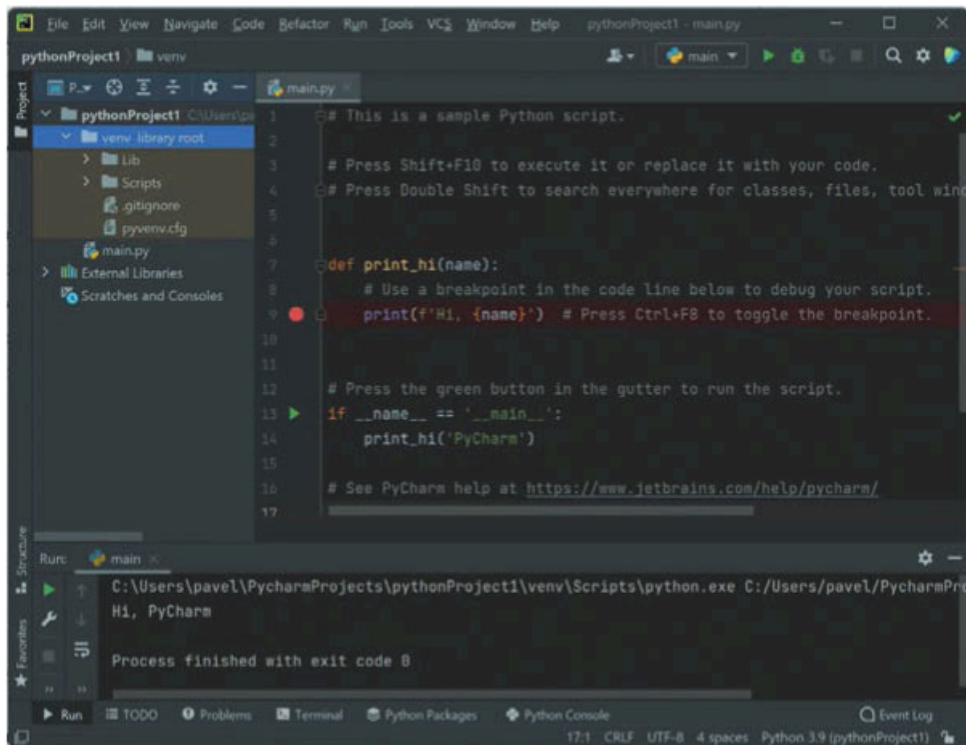


Рис. 9. Первый проект в PyCharm

Вы можете заметить в исходном коде, что некоторые строки начинаются с отступа, длиной в четыре пробела. Так в языке Python разделяются элементы кода (их вложенность). Далее мы это еще изучим.

Попробуйте заменить полностью все содержимое исходного кода программы на следующее и посмотреть на результат. Результат должен быть очевиден: теперь вместо «Hi, *ваш текст*» выведется «Привет».

```
if __name__ == '__main__':
    print('Привет')
```

Строка «`if __name__ == '__main__':`» определяет точку входа в вашу программу. Дело в том, что интерпретатор, когда откры-

вает на выполнение программу, ищет в ней точку входа, т.е. тот код, с которого начинается выполнение кода. Вероятно, это единственный момент, который никак не был отражен при рассмотрении Jupyter. Но если вы захотите выгрузить исходный код из Jupyter (при помощи: «File → Download as → Python (.py)», рис. 10) и запустить его в PyCharm, то вам просто будет необходимо вставить все его содержимое в тело точки входа (самый простой вариант). То есть под строчкой «if __name__ == '__main__':» с отступом в четыре пробела у каждой строчки вашего кода.

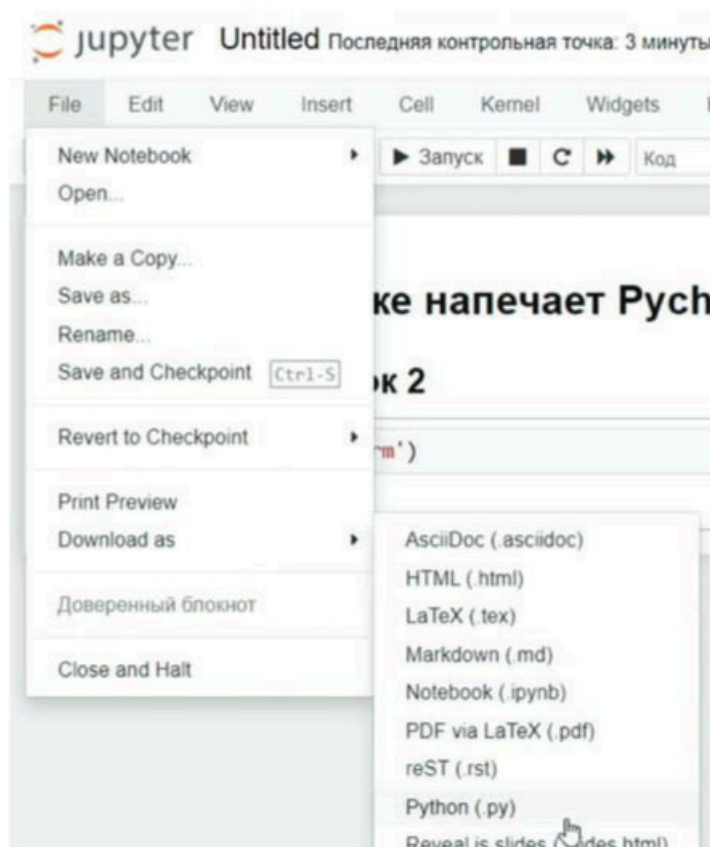


Рис. 10. Экспорт кода в Jupyter

Но обратите внимание, в коде блокнота Jupyter могут встречаться некоторые функции, называемые «магическими» и содержащие знак процента в начале строки, которые нужны в только для управления работой самого блокнота (часто — для вывода графиков внутри блокнота), поэтому может быть необходимо их удалять при вставке кода в PyCharm. К счастью, PyCharm во вкладке

«Problems» под полем вывода результата работы программы сообщит об ошибках и даже подскажет, как их можно исправить.

Теперь рассмотрим установку пакетов. Для этого под окном вывода результата работы вашей программы есть раздел «Python Packages», в котором вы можете устанавливать необходимые пакеты через менеджер пакетов `pip` (рис. 11), введя только название нужного пакета. Но есть более удобный способ: в самом верху окна, правее от кнопки запуска программы, есть кнопка настроек («шестеренка»), нажав ее, перейдите в настройки (Settings), далее найдите в настройках раздел настроек вашего проекта (рис. 12), выберите раздел настроек интерпретатора (Python interpreter), и в нем вам будет доступно управление пакетами через графический интерфейс.

Ранее мы с вами подробно рассматривали Anaconda Navigator — удобный менеджер виртуальных окружений и менеджер пакетов, содержащий множество предустановленных библиотек для аналитики данных. Давайте вернемся к нему и создадим в PyCharm проект на основе виртуального окружения Anaconda Navigator.

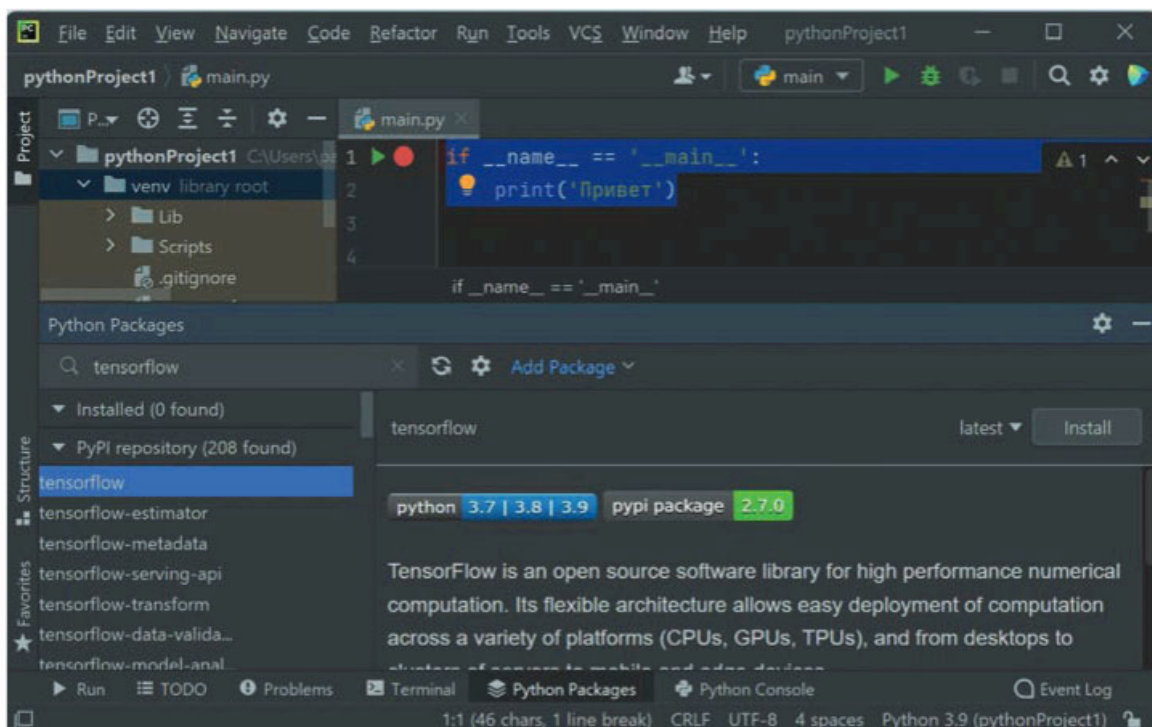


Рис. 11. Установка библиотек в PyCharm через `pip`

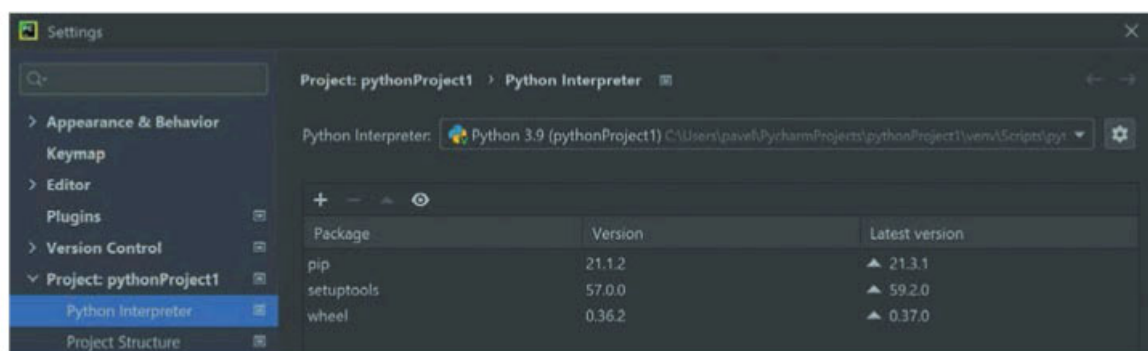


Рис. 12. Установка библиотек в PyCharm через настройки

В PyCharm создайте новый проект. В настройках проекта в разделе выбора виртуального окружения выберите «Previous configured interpreter» (с англ. — «ранее настроенный интерпретатор») и нажмите на три точки (кнопка, расположенная правее от выпадающего списка). Выберите «Conda Environment» (рис. 13).

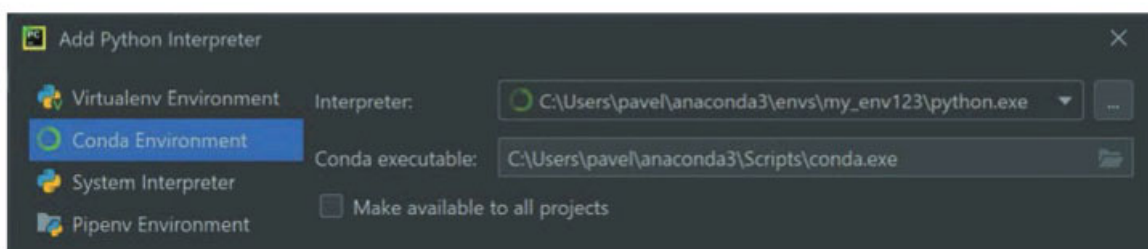


Рис. 13. Выбор conda-окружения в PyCharm

Здесь (представлено на рис. 13) необходимо указать путь к интерпретатору (Interpreter). На рисунке показано, что выбран следующий путь:

`C:\Users\pavel\anaconda3\envs\my_env123\python.exe`

Это потому, что мы создавали в Anaconda Navigator виртуальное окружение с именем «my_env123», PyCharm нашел его и выбрал интерпретатор «python.exe», находящийся именно в этом виртуальном окружении. У каждого виртуального окружения есть своя папка, которая и хранит все необходимые вещи для работы

программы, например библиотеки или интерпретатор. Это вы можете увидеть даже на более ранних рисунках (например, на рис. 9): в дереве файлов проекта создалась папка виртуального окружения «venv», хранящая все библиотеки, подключенные к проекту.

Если вы не создавали своего собственного виртуального окружения в Anaconda Navigator, то просто укажите интерпретатор базового «(base)» окружения Anaconda Navigator, который хранится по схожему пути:

`C:\Users\pavel\anaconda3\python.exe`

Теперь создайте проект, и в нем будет использоваться окружение Anaconda Navigator вместо окружения «venv». Папка «venv» в дереве файлов проекта будет отсутствовать (рис. 14), а пакеты вы можете устанавливать через Anaconda Navigator, и они будут доступны программе, которую вы разрабатываете в PyCharm. Попробуйте запустить ту же `main.py`-программу, она будет работать точно так же.

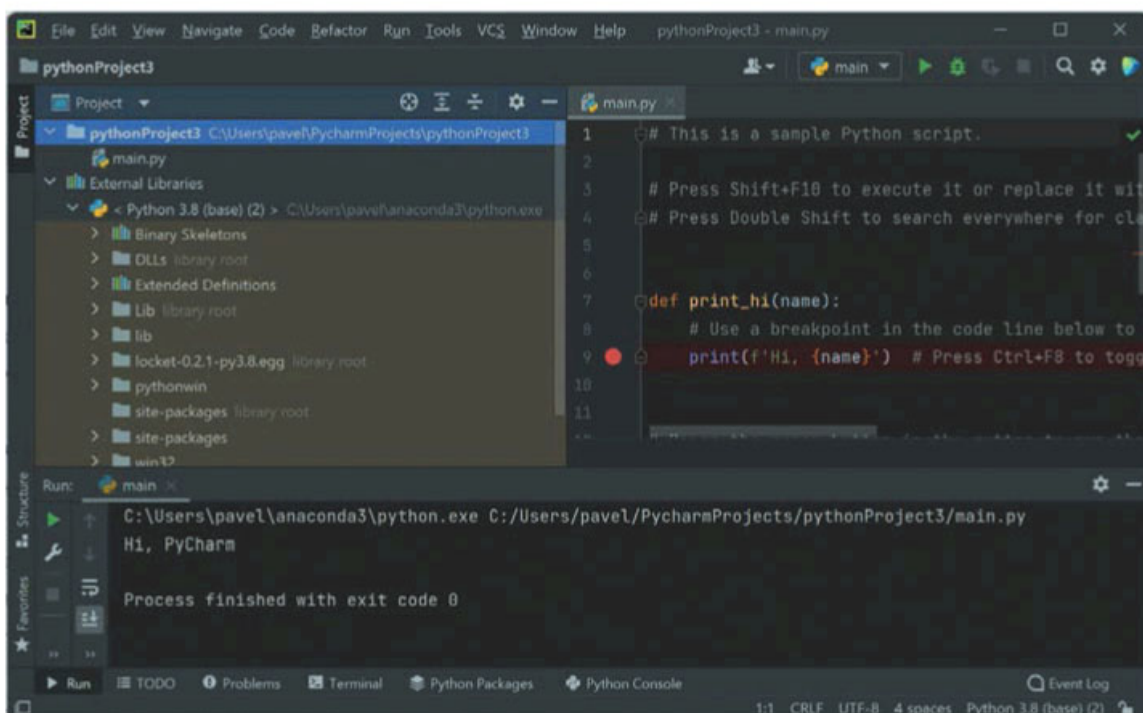


Рис. 14. PyCharm-проект с окружением conda

Среда для выполнения занятий по данному курсу (JupyterLab)

Предполагается выполнение всех дальнейших занятий в блокнотах в утилите JupyterLab. Вместе с данной книгой распространяется набор блокнотов и некоторых необходимых для их выполнения файлов (картинки, наборы данных и прочие файлы). Ниже представлена краткая инструкция по запуску данных блокнотов на компьютере:

1. Скачайте и установите Anaconda Navigator (см. сноску 1 на стр. 11). Данный пункт уже был выполнен в ходе текущего занятия.
2. На вкладке Environments программы Anaconda Navigator импортируйте конфигурацию виртуального окружения, т.е. файл «data_analyst_v1.yml».

Данный файл также распространяется вместе с книгой и содержит список всех необходимых в рамках данного курса модулей Python и прочих пакетов. Он сгенерирован при помощи менеджера пакетов conda (или его можно также получить из графического интерфейса при помощи Anaconda Navigator), поэтому вы можете легко создавать подобные файлы описания окружения и обмениваться ими со своими коллегами. Аналогичные файлы доступны и в других менеджерах пакетов, например pip.

Ниже представлено содержание файла «data_analyst_v1.yml» на случай, если вы не получили совместно с книгой никаких файлов. Указания версий, например «=3.8.2», можно удалить или заменить на ближайшие, если они окажутся ненайденными.

```
name: data_analyst_v1
channels:
  - defaults
dependencies:
  - python=3.8.2
  - ipython=7.29.0
  - ipywidgets=7.6.5
  - plotly=5.1.0
  - jupyterlab=3.2.5|3.2.1
  - matplotlib=3.4.3
  - nbformat=5.1.3
```

```
- numpy=1.21.2
- pandas=1.3.4
- pydot=1.4.1
- scikit-learn=1.0.1
- seaborn=0.11.2
- tensorflow=2.4.1|2.3.0
- scipy=1.7.1
prefix: data_analyst_v1
```

3. Вам может потребоваться установить утилиту `graphviz` (доступна для скачивания в источнике [12]).

При тесте на Windows 11 устанавливать ее отдельно не потребовалось, так как она сама это сделала при импорте конфигурации виртуального окружения (из п. 2) и установке пакета `pydot`, в зависимости (т.е. автоматическую установку) которого входит наличие `graphviz`.

Однако на Ubuntu 20.10 `graphviz`, установленный через `conda`, не доступен⁴ в ноутбуке одного из занятий (занятие по библиотеке `TensorFlow`), поэтому его нужно установить вручную, например командой «`apt-get install graphviz`».

Вы можете игнорировать данный пункт до тех пор, пока в указанном блокноте не встретите описанную в сноске 4 ошибку.

Если же вам все-таки потребуется вручную в Windows установить `graphviz`, то в установщике выбирайте пункт «`add to PATH for all user`». Это означает, что эта утилита будет доступна отовсюду и всем в системе.

4. На вкладке `Home` программы `Anaconda Navigator` запустите `JupyterLab` от виртуального окружения `data_analyst_v1` (оно выберется автоматически, если вы только что его импортировали).

⁴ Выводится ошибка:

```
«('Failed to import pydot. You must `pip install pydot` and install
graphviz (https://graphviz.gitlab.io/download/), ', 'for
`pydotprint` to work.')
```

Для понимания, почему это происходит: `pydot` является Python-интерфейсом для утилиты `graphviz`. Видимо, в Windows-версии пакет `pydot` корректно использует `graphviz`, установленный именно через `conda`, а Linux-версия `pydot` «пытается» использовать системно установленный `graphviz`, а не `graphviz` виртуального окружения. Актуально по состоянию на момент написания данной книги.

При последующих запусках Anaconda Navigator просто выберите на вкладке Home именно это окружение и из него запускайте JupyterLab.

5. После запуска JupyterLab откроется браузер. Импортируйте блокноты и все необходимые для их работы файлы. Теперь можно запускать блокноты.

В каждом блокноте можно выбрать, с каким kernel (ядром), он будет выполняться («вкладка kernel → Change Kernel» (англ. *change* — «сменить»)).

Так как JupyterLab запущен именно из только что созданного окружения, то ядро с названием «Python 3 (ipykernel)» — это то ядро, которое и было создано посредством импортирования виртуального окружения.

То есть используя для запуска блокнотов ядро «Python 3 (ipykernel)», вы будете применять ядро, которое содержит в себе все необходимые для изучения книги модули и пакеты⁵.

Задания для проверки

1. Установите программу Anaconda Navigator, откуда запустите Jupyter Notebook и попробуйте выполнить любой код в ячейке, например код, представленный ниже. Затем попробуйте сделать то же самое через JupyterLab.

```
print('Hello world')
```

2. Опишите кратко, что такое Jupyter Notebook и JupyterLab, а также их базовый интерфейс и функционал.

3. Попробуйте проверить, установлены ли Python-пакеты например, Pandas и NumPy. Если не установлены, то попробуйте сде-

⁵ К сожалению, ядро по умолчанию в любом окружении называется именно «Python 3 (ipykernel)». Если вы запустите JupyterLab не из окружения, созданного при помощи импорта файла «data_analyst_v1.yml», даже если вы выберете ядро «Python 3 (ipykernel)», то, несмотря на то что названия ядер совпадает, это не означает, что подразумевается именно то целевое ядро, где все необходимые компоненты установлены.

лать это при помощи Anaconda Navigator либо любым другим способом.

4. Установите IDE PyCharm и попробуйте выполнить в ней любой Python-код возможным любым способом, желательно — используя виртуальное окружение Anaconda Navigator.

5. Используя окружение «data_analyst_v1.yml», попробуйте в JupyterLab выполнить ноутбук по библиотеке TensorFlow и проверить, что нет никаких ошибок отсутствия модулей.

СИНТАКСИС, ТИПЫ, ФУНКЦИИ, ЦИКЛ WHILE И УСЛОВНЫЕ ВЫРАЖЕНИЯ⁶

Цели выполнения работы

Получение базовых знаний по основам языка программирования Python: базовый синтаксис, типы данных, функции, циклы, условные выражения. Изучение основных принципов практической работы с ними.

Порядок выполнения работы

Выполнение текущего и всех дальнейших занятий необходимо производить в блокнотах в JupyterLab.

На предыдущем занятии упоминалось использование функции `print()` для вывода произвольного текста. Понятие «функция» будет рассмотрено позже в ходе этого занятия. Сейчас давайте посмотрим на полный синтаксис этой функции, т.е. на ее набор параметров:

```
print(*objects, sep=' ', end='n', file=sys.stdout, flush=False)
```

где:

- `objects` — объект, который нужно вывести. Символ звезды «*» обозначает, что объектов может быть несколько;

⁶ Разработано на основе блокнотов из англоязычного источника [13].

- `sep` разделяет объекты. Значение по умолчанию: «' '», т.е. выводимые объекты будут разделены пробелом;

- `end` ставится после всех объектов;

- `file` — ожидается объект с методом `write (string)`. Если значение не задано, для вывода объектов используется файл `sys.stdout`, т.е. системный вывод (терминал, консоль или ячейка Jupyter Notebook);

- `flush` — если задано значение `True`, поток принудительно сбрасывается в файл. Значение по умолчанию: `False`. Примечание: `sep`, `end`, `file` и `flush` — это аргументы — ключевые слова. Если хотите воспользоваться аргументом `sep`, используйте «`print(*objects, sep = <separator>)`», а не «`print(*objects, <separator>)`».

Описанное выше (синтаксис функции) является отрывком из документации на нее в сети «Интернет». Из этого отрывка мы можем сделать вывод, что возможно печатать несколько значений сразу, если передать их функции `print()` с разделением через запятые, например, выполнив представленный ниже код, интерпретатор Python выведет три числа (рис. 15), разделенные разделителем по умолчанию, которым является знак пробел.

```
---in↓---  
print(1,2,3)  
---↑in_out↓---  
1 2 3  
---↑out---
```



```
In [1]: print(1,2,3)  
1 2 3
```

Рис. 15. Результат работы `print(1, 2, 3)`

«In» с англ. — «ВВОД», а «out» — «ВЫВОД», т.е. введя и выполняя команду «`print(1, 2, 3)`», мы увидим результат ее выполнения: «1 2 3». Далее большинство выполняемых кодов будет представлено именно в таком формате. Код из секции «In» вам необходимо

выполнять у себя на компьютере и сравнивать полученный результат с представленным в данной книге.

Число в квадратных скобках (см. «In [1]:») — это порядковый номер водимой команды. То есть номер 1 означает, что это первая команда по счету, которую уже выполняли в данном Jupyter-блокноте. Этот номер ни на что не влияет. Если номер не указан, значит ячейка не выполнялась, а если указано «In [*]:», значит команда еще выполняется. Чтобы сбросить результат выполнения, можно сбросить ядро «Kernel → Restart Kernel and Clear All Outputs» из меню сверху.

Попробуйте изменить разделитель «sep» и набор завершающих символов «end»:

```
---in↓---
print(1,2,3, sep='_', end='\nконец') #\n — спец символ
перевода строки
---↑in_out↓---
1_2_3
конец
---↑out---
```

Математика в Python

Самый простой способ использовать Python для вычислений — это выполнить команду, представленную ниже. Попробуйте, применяя полученную информацию, сложить два других числа, например $20 + 80$.

```
---in↓---
1 + 1
---↑in_out↓---
2
---↑out---
```

Только что была рассмотрена операция «сложение» — одна из доступных арифметических операций в Python. В таблицах ниже представлены некоторые математические операторы и функции (табл. 1) и операторы присваивания (табл. 2).

Таблица 1

Математические операторы и функции

Оператор или функция	Описание	Пример
<code>+, -, *, /</code>	Сложение, вычитание, умножение, деление	$10 + 21 = 31$
<code>%</code>	Деление с остатком. Делит левый операнд на правый операнд и возвращает остаток	$21 \% 10 = 1$
<code>**</code>	Возведение в степень	$a ** b$, т.е. возвести a в степень b
<code>//</code>	Деление с округлением вниз (до ближайшего меньшего)	$9 // 2 = 4$, $9,0 // 2,0 = 4,0$, $-11,0 // 3 = -4,0$
<code>divmod(X, Y)</code>	Возвращает пару «частное — остаток от деления аргументов»	<code>divmod(9, 4) = (2, 1)</code>
<code>abs(X)</code>	Возвращает модуль числа	<code>abs(-5)=5</code>
<code>x.is_integer()</code>	Проверяет целочисленного ли типа число	$x=2.59$ <code>x.is_integer() = ложь</code>
<code>bin(x)</code>	Преобразование целого числа в двоичное	<code>bin(8)= '0b1000'</code>

Попробуйте выполнить некоторые представленные выше операции. Например, нахождение остатка от деления при помощи оператора «%»:

```

---in↓---
23 % 3
---↑in_out↓---
2
---↑out---
```

Таблица 2

Операторы присваивания

Оператор	Описание	Примеры
=	Присваивает значение правого операнда левому	c = 23 присвоит переменной c значение 23
+= и так же: -=, *=, /=, %=, **=, //=	Прибавит значение правого операнда к левому и присвоит эту сумму левому операнду. Аналогичное доступно для всех других математических операторов	c = 5 a = 2 c += a равносильно: c = c + a. c будет равно 7

Количество выполняемых математических операций может быть расширено при помощи использования функций из большого числа доступных Python-библиотек, например можно пользоваться и другими, обычно более сложными операциями, такими как перемножение матриц и т.д.

Порядок операций

Порядок операций в Python можно задавать при помощи круглых скобок. Ниже представлен пример их использования. Попробуйте убрать скобки и посмотреть на результат.

```

---in↓---
(1 + 2) * 3
---↑in_out↓---
9
---↑out---
```

Комментарии

Комментарии — это пояснения к исходному тексту программы, находящиеся непосредственно внутри комментируемого кода. Синтаксис комментариев определяется языком программирования.

ния. С точки зрения компилятора или интерпретатора комментарии — часть текста программы, не влияющая на ее семантику. Комментарии не оказывают никакого влияния на результат компиляции программы или ее интерпретацию.

Выполните следующие три строчки:

```
---in↓---  
# Строка ниже закомментирована и не выполнится  
# print("Привет")  
print("Мир")  
---↑in_out↓---  
Мир  
---↑out---
```

и вы увидите, что выполнится только «`print("Мир")`», а «`# print("Привет")`» — нет, так как является комментарием. Все, что идет после знака решетки до конца строки, — это комментарий. Не обязательно комментировать всю строку, можно оставить комментарий правее от кода.

Для многострочных комментариев, которые обычно используются как строки документации в коде, в Python можно применять следующую конструкцию:

```
---in↓---  
"""  
Это строковый литерал. Он часто используется как  
многострочный комментарий  
"""  
print("Мир")  
---↑in_out↓---  
Мир  
---↑out---
```

Посмотрим на исходный код, которой представлен ниже. Вы бы могли его вынести (или экспортировать) в отдельный файл с названием, например, «Есенин.py» и запускать эту программу без Jupyter или PyCharm, а используя только Python интерпретатор.


```

---in↓---
#Простая программа
print("Пой же, пой. На проклятой гитаре")
print("Пальцы пляшут твои в полукруг.")
print("Захлебнуться бы в этом угаре,", end = "\n")
print("Мой последний, единственный друг.")
---↑in_out↓---
Пой же, пой. На проклятой гитаре
Пальцы пляшут твои в полукруг.
Захлебнуться бы в этом угаре,
Мой последний, единственный друг.
---↑out---

```

Вставьте ее в ячейку Jupyter. Если вы запустите программу (нажатием сочетания клавиш `shift + Return` или нажатием кнопки запуска ячейки), интерпретатор выполнит строки с 1 по 5 друг за другом и выведет результат.

Стоит отметить, что, например, в среде программирования PyCharm автоматически появляются варианты автодополнения кода, который вы пишете (рис. 16, также обратите внимание на многострочные комментарии, они выделяются зеленым цветом), но в Jupyter для показа вариантов автоматического дополнения нужно нажать клавишу `tab`, и тогда либо вам будут предложены варианты дополнения, либо произойдет автоматическая вставка. Например, если вы напишете «`pri`» и нажмете клавишу `tab`, то произойдет автоматическая вставка слова «`print`».

Вернемся к программе «Есенин.py». В строке 4 параметр `end =»\n`», передающийся функции `print()`, настраивает ее таким образом, чтобы она вставляла специальный символ перевода строки `\n`, который отвечает за действие в конце выполнения этой функции. Хотя она и так по умолчанию переводит строку на новую, но для наглядности это сообщено вам. Вы можете, например, вставить пробел вместо перевода строки.

```

---in↓---
print("Захлебнуться бы в этом угаре,", end = " ")
print("Мой последний, единственный друг.")
---↑in_out↓---

```

Захлебнуться бы в этом угаре, Мой последний, единственный друг.

---↑out---

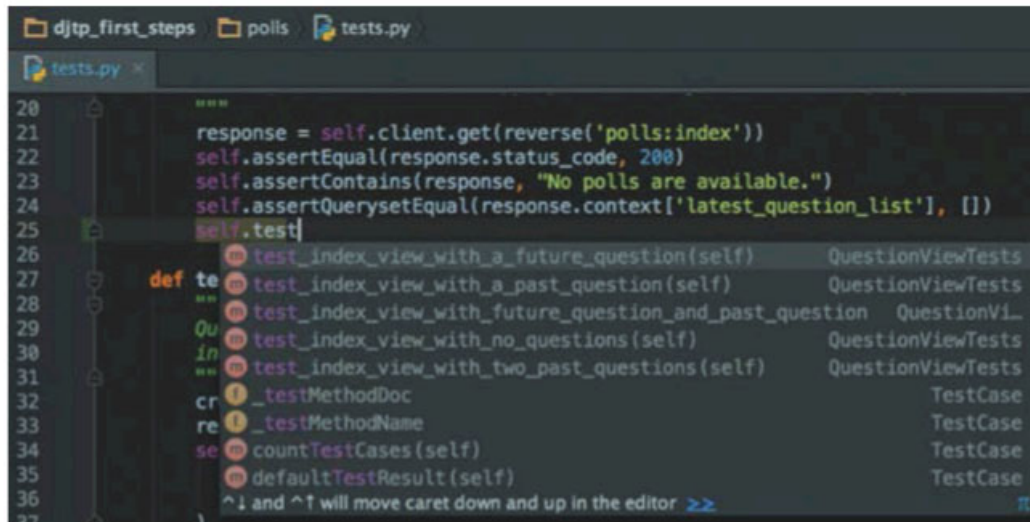


Рис. 16. Подсветка синтаксиса в PyCharm

Переменные

Теперь давайте начнем вводить переменные. Переменные хранят значение, которое можно просмотреть или изменить позже. Создадим программу, использующую переменные. Запустите представленный ниже код и попробуйте разобраться в результатах его работы:

```
---in↓---
#демонстрация переменных
print("Эта программа представляет собой демонстрацию
переменных")
v = 1
print("Сейчас значение v: ", v)
v = v + 1
print("v теперь равно самому себе плюс один, что делает
его равным ", v)
v = 51
print("v может хранить любое числовое значение для
использования в другом месте.")
```

```

print("например, в предложении. v теперь равно ", v)
print("v умножить на 5 равно", v * 5)
print("но само v все еще остается равно", v)
print("чтобы увеличить v в пять раз, нужно набрать v = v * 5")
v = v * 5
print("вот так, теперь v равно", v, "а не ", v / 5)
---↑in_out↓---

```

Эта программа представляет собой демонстрацию переменных
 Сейчас значение v: 1
 v теперь равно самому себе плюс один, что делает его
 равным 2
 v может хранить любое числовое значение для использования
 в другом месте.
 например, в предложении. v теперь равно 51
 v умножить на 5 равно 255
 но само v все еще остается равно 51
 чтобы увеличить v в пять раз, нужно набрать v = v * 5
 вот так, теперь v равно 255 а не 51.0
 ---↑out---

Обратите внимание, что мы также можем записать $v=v+1$ как $v+=1$. Это можно использовать для всех операторов (например, $-=$, $*=$, $/=$). Попробуйте сделать это в приведенном выше коде.

Для названий переменных рекомендуется использовать строчные буквы или «верблюжий регистр». CamelCase (с англ. — «верблюжий регистр», также «горбатый регистр», «стиль верблюда») — стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово внутри фразы начинается с прописной буквы.

В названии переменных нельзя использовать специальные символы или начинать названия с цифр.

Строки

Переменные хранят значения для использования их при дальнейшей работе программы. Вы можете изменить их в любой момент. Однако вы можете хранить в переменных не только цифры. Переменные могут содержать, например, текст.

Переменная, содержащая текст, называется строкой. Попробуйте выполнить эту программу:


```
---in↓---
#объявление и определение строковых переменных, сложение
строк
word1 = "Добрый"
word2 = "вечер"
word3 = "вам!"
print(word1, word2)
sentence = word1 + " " + word2 + " " + word3
print(sentence)
---↑in_out↓---
Добрый вечер
Добрый вечер вам!
---↑out---
```

Указанные выше переменные содержали текст. Имена переменных также могут быть длиннее одной буквы, здесь у нас были слова `word1`, `word2` и `word3`.

В примере выше продемонстрировано, что строки можно складывать вместе, чтобы получились более длинные слова или предложения. Тем не менее оператор сложения не добавляет пробелы между словами, поэтому можно вставить строку, содержащую только пробел «« «» (показано в примере).

Попробуйте использовать следующий код и объясните, что он делает:

```
---in↓---
text = "abcdefghij"
len(text)
---↑in_out↓---
10
---↑out---
```

Представленный выше код выводит количество символов в строке. А теперь попробуйте выполнить следующее:

```
---in↓---
print(text[4])
---↑in_out↓---
e
---↑out---
```

Здесь мы хотим напечатать символ, стоящий в позиции 4. Обратите внимание, что первый символ «a» находится в позиции 0, потому что индексация в языке программирования Python начинается с нуля. Итак, позиция 4 возвращает нам букву «e». А теперь выполните:

```
---in↓---
print(text[:4])
print(text[4:])
---↑in_out↓---
abcd
efghij
---↑out---
```

Конструкция `[:4]` выбирает символы с индексами 0, 1, 2, 3, т.е. «abcd». Используя оператор обрезания «двоеточие» в конструкции `[4:]`, мы осуществляем выборку элементов, начинающихся с позиции с индексом 4, до конца строки, в результате получаем вывод куска исходной строки «efghij». На английском «обрезка» — *slice*, поэтому часто результат обрезки называют «слайс».

Мы также можем указать диапазон. Попробуйте:

```
---in↓---
print(text[4:8])
---↑in_out↓---
efgh
---↑out---
```

Для диапазонов максимальное значение (индекс 8 из примера выше) не включается в выборку.

Отрицательные индексы означают индексацию справа налево, а положительные — классическую, слева направо.

Попробуем вывести второй символ справа, а затем выбрать четвертый символ и все символы правее (т.е. сделаем вырезку «начать с 4-го символа справа и вывести все символы от этой позиции до конца строки»).

```
---in↓---
#text = "abcdefghij"
print(text[-2])
print(text[-4:])
---↑in_out↓---
i
ghij
---↑out---
```

Попробуйте выполнить и понять, что делает этот код:

```
---in↓---
print(text[::2])
---↑in_out↓---
Acegi
---↑out---
```

Данный код вырезает каждый второй элемент. Условно «второй», потому что по факту он вырезает каждый первый по индексу элемент, так как индексация начинается с нуля, а указывать в данной конструкции необходимо тот индекс, до которого нужно выбирать символы, не включая его.

Мы также можем добавить результаты нарезки списка к переменным и вставить их в строки, используя метод `format()` (что такое «метод» будет рассмотрено на следующем занятии):

```
---in↓---
nrOfCharacters = len(sentence)
lastWord = sentence[-4:-1]
print(«В предложении есть {} символов, а последнее слово: {}».  
format(nrOfCharacters, lastWord))
---↑in_out↓---
В предложении есть 17 символов, а последнее слово: вам
---↑out---
```

Таким образом, вы можете легко вставлять переменные в места в строках, отмеченные оператором ассоциации «`{}`». Они будут подставлены в том же порядке, что и переменные в методе «`.format()`». Это также можно записать как:


```
---in↓---
print(f"В предложении {nrOfCharacters} букв и последнее
слово: {lastWord}")
---↑in_out↓---
В предложении 17 букв и последнее слово: вам
---↑out---
```

У нас все еще в «памяти» интерпретатора языка Python хранится переменная «предложение» (sentence). Попробуйте вывести второе слово предложения, используя полученные навыки:

```
---in↓---
print(sentence)
print(sentence[6:12])
---↑in_out↓---
Добрый вечер вам!
вечер
---↑out---
```

Помимо нарезки списка есть также другие операции, которые мы можем применить к строкам. Мы можем подсчитать количество вхождений определенного символа в строку:

```
---in↓---
print(sentence.count('o'))
---↑in_out↓---
0
---↑out---
```

Мы также можем найти положение символа. Ниже представлено два примера с использованием двух разных методов.

```
---in↓---
print(text.find('e'))
sometext = "Привет, как дела как?"
print(sometext.rfind("как"))
---↑in_out↓---
4
17
---↑out---
```

Метод `rfind()` возвращает последнее появление строки. Итак, в текстовой переменной `sometext` мы дважды употребляем слово «как». Метод `rfind()` возвращает 17, это означает, что последний раз, когда он нашел слово «как», оно начиналось с позиции 17 (считая с 0).

Есть еще несколько полезных методов для строк, некоторые примеры которых представлены ниже. Запустите приведенный ниже код и проанализируйте результат:

```
---in↓---
# Переводит строку в верхний регистр
print(sometext.upper())

# Разбивает строку и возвращает ее как элементы списка.
Вы узнаете о списках позже
print(sometext.split(","))

# Заменяет строки
print(sometext.replace("?", "!"))
---↑in_out↓---
ПРИВЕТ, КАК ДЕЛА КАК?
['Привет', ' ' как дела как?']
Привет, как дела как!
---↑out---
```

Также существуют некоторые специальные символы:

- «`\n`» — переходит на новую строку;
- «`\`» — `escape`-символ (англ. *escape* — «освобождающий», «экранирующий»). Вы можете поместить его перед другим символом, который имеет значение в коде и не считается строкой. Как один из примеров применения, экранирование часто используется, чтобы создать строки, представляющие собой пути к файлу (например, «`C:\\папка\\имя_файла.txt`»). Поскольку «`\`» уже является `escape`-символом, нам нужно использовать его дважды, чтобы экранировать косую черту в строке («экранировать» означает, что в нашей строке действительно используется символ «слэш», а не специальный символ).

Ниже представлено несколько примеров использования специальных символов «\n» и «\». Обратите внимание, что специальных символов существует большее количество, чем два указанных. Они аналогичны тем, что применяются в других языках программирования.

```

---in↓---
print("Это очень длинное предложение, и я хочу разбить
его на две строки.")
print("Это очень длинное предложение \n, и я хочу разбить
его на две строки.")
print("Это предложение содержит цитату, и я не хочу,
чтобы строка заканчивалась (пока) \"")
---↑in_out↓---
Это очень длинное предложение, и я хочу разбить его на две
строки.
Это очень длинное предложение
, и я хочу разбить его на две строки.
Это предложение содержит цитату, и я не хочу, чтобы
строка заканчивалась (пока) "
---↑out---

```

Цикл while

Представьте, что вам нужна программа, которая сделает что-то 20 раз. Вы можете скопировать и вставить код 20 раз и получить потенциально более медленную и практически нечитаемую программу. Для решения этой проблемы вы можете «приказать» компьютеру повторять фрагмент кода между точками А и В, пока не выполнится условие, при котором нужно будет остановиться. Это называется циклом.

Ниже приведены примеры цикла, называемого циклом *while* (англ. *while* — «пока»). Пока выполняется некоторое условие, то код, располагающийся внутри цикла *while*, будет выполняться.

```

---in↓---
a = 0
while a < 10:

```



```
    a = a + 1
    print(a)
---↑in_out↓---
1
2
---часть вывода вырезана---
8
9
10
---↑out---
```

Код, который расположен внутри цикла, называется телом цикла. Тело бывает не только у циклов, но и у множества других конструкций языка программирования. Тело и вложенность кода задаются при помощи четырех пробелов (или нажатия клавиши `tab`).

Ниже представлено описание кода на алгоритмическом языке. Алгоритмический язык описывает последовательность работы программы на естественном русском языке:

```
'a' теперь равно 0
Пока "a" меньше 10, сделайте следующее:
    Сделайте единицу больше, чем она есть.
    Напечатайте на экране, сколько сейчас стоит "a".
```

Давайте рассмотрим, какую последовательность действий компьютер будет производить при выполнении цикла `while`:

ИТЕРАЦИЯ 0

```
"A" меньше 10? ДА (его 0)
Сделать 'a' на один больше (теперь 1)
напечатать на экране значение 'a' (1)
```

ИТЕРАЦИЯ 1

```
"A" меньше 10? ДА (его 1)
Сделайте "a" на один больше (теперь 2)
напечатать на экране, что такое 'a' (2)
-----
```

ИТЕРАЦИЯ 9

```
"A" меньше 10? НЕТ (значение "A" = 10, следовательно,
не оно меньше 10)
```

Не делать цикл
 Кода больше не осталось, поэтому программа завершается

Цикл `while` имеет следующий синтаксис:

```
while {условие продолжения цикла}:
    {код тела цикла (что делать)}
    {тело всегда отделяется tab-ом или четырьмя
пробелами}
код здесь не зациклен
потому что он без отступа
```

Теперь проанализируйте результат выполнения представленного ниже кода:

```
---in↓---
x = 10
while x != 0:
    print(x)
    x = x - 1
    print("мы посчитали x, и теперь он равен", x)
print("И вот цикл закончен.")
---↑in_out↓---
10
мы посчитали x, и теперь он равен 9
9
мы посчитали x, и теперь он равен 8
8
---вывод обрезан---
мы посчитали x, и теперь он равен 1
1
мы посчитали x, и теперь он равен 0
И вот цикл закончен.
---↑out---
```

В примере было показано, что «`print("И вот цикл закончен.")`», не являющийся элементом тела цикла, выполнился только один раз. Цикл выполняет то, что внутри его тела, пока условие выполняется «`x != 0`», т.е. истинно.

Логические (или булевы) выражения

В области, отмеченной {условия продолжения цикла}, которую мы с вами рассматривали выше, указывается выражение, называемое логическим или булевым. Логическое выражение означает вопрос, на который можно ответить истина или ложь (`true` и `false` на англ. соответственно). Например, если вы хотите сравнить ваш возраст с возрастом человека рядом, вы должны ввести:

Мой возраст == возраст человека рядом со мной

Для создания логических выражений вы можете использовать любые операторы сравнения, которые представлены в таблице ниже (табл. 3).

Таблица 3

Операторы сравнения

Оператор	Описание	Пример для <code>a=10</code> , <code>b=20</code>
<code>==</code> , <code>!=</code> или <code><></code>	Оператор «равно» и два варианта оператора «не равно» соответственно	<code>(a == b)</code> не верно <code>(a != b)</code> истинно
<code>></code> и <code><</code>	Оператор «больше» и оператор «меньше» соответственно	<code>(a > b)</code> не верно <code>(a < b)</code> истинно
<code>>=</code> и <code><=</code>	Оператор «больше или равно» и оператор «меньше или равно» соответственно	<code>(a >= b)</code> не верно <code>(a <= b)</code> истинно

Не стоит путать «`=`» и «`==`». Оператор присваивания «`=`» делает то, что находится слева, равным тому, что находится справа. оператор сравнения «`==`» определяет, совпадает ли элемент слева с тем, что находится справа, и возвращает `True` или `False`.

Условные инструкции

Условные инструкции (выражения) — это когда часть кода запускается только при соблюдении определенных условий. Это похоже на работу цикла `while`, рассмотренную ранее, однако тело условных инструкций выполняется только один раз. Самым распространенным условным выражением в любом языке программирования является оператор `if`. Ниже представлен его синтаксис:

```
if {условия, которые должны быть выполнены}:
    {сделай это}
    {и это}
    {и это}
{но это происходит независимо}
{потому что это без отступа}
```

Рассмотрим несколько примеров использования `if`:

```
---in↓---
#Пример 1
y = 1
if y == 1:
    print("y по-прежнему равно 1, я просто проверял")
---↑in_out↓---
y по-прежнему равно 1, я просто проверял
---↑out---
```

```
---in↓---
#Пример 2
print("Мы покажем четные числа до 20")
n = 1
while n <= 20:
    if n % 2 == 0:
        print(n)
    n = n + 1
print("готово")
---↑in_out↓---
Мы покажем четные числа до 20
2
```

```
4
6
---часть вывода вырезана---
18
20
готово
---↑out---
```

Пример 2 выглядит непростым. Но все, что мы в нем выполняем, — это запуск оператора `if` каждый раз, когда выполняется тело цикла `while` (каждый новый такт выполнения тела называется итерацией). Математическая операция «%» означает остаток от деления, т.е. осуществляется проверка того, что ничего не осталось (остаток равен нулю), если число делится на два — показывая, что оно четное. Если значение переменной `n` четное, то программа выводит его.

else и elif-операторы «когда это не так»

Есть несколько способов использования оператора `if` для ситуаций, когда ваше логическое выражение оказывается ложным. Это добавление после оператора `if` операторов `else` и `elif`.

Оператор `else`, указанный после `if`, определяет, что делать, если условие в `if` не выполняется. Например, в следующем коде переменная `a` не больше пяти, поэтому то, что указано в `else`, выполняется:

```
---in↓---
a = 1
if a > 5:
    print("Это не должно выполниться.")
else:
    print("Это должно выполниться.")
---↑in_out↓---
Это должно выполниться.
---↑out---
```

Оператор `elif` — это сокращенный способ указания конструкции `else if`. Когда условие оператора `if` ложно, `elif` будет делать то, что содержится в его теле, если выполняются условия. Например:

```

---in↓---
z = 4
if z > 70:
    print("Что-то не так")
elif z < 7:
    print("Все ок")
---↑in_out↓---
Все ок
---↑out---
```

Использование оператора `if` вместе с `else` и `elif` выглядит следующим образом:

```

if {#условия}:
    {#выполнить этот код}
elif {#условия}:
    {run this code}
elif {#условия}:
    {#выполнить этот код}
else:
    {#выполнить этот код}
# У вас может быть столько или меньше операторов elif,
# сколько вам нужно
# Но у вас может быть не более одного оператора else
# и только после всех остальных if и elifs.
```

Один из наиболее важных моментов, который следует помнить, — это то, что необходимо ставить двоеточие в конце каждой строки, содержащей `if`, `elif`, `else` или `while`. В условиях вы также можете указывать составные условия, созданные при помощи логических операторов «&» или «|» («и» и «или» соответственно):

```

---in↓---
if (2>0) & (3>0):
    print("...")
---↑in_out↓---
...
---↑out---
```


Отступы

Вложенность строк кода в Python, как уже было сказано, определяется при помощи четырех пробелов перед каждой вложенной строкой кода. При этом строки кода разделяются между собой переносом строки на новую (т.е., например, вы не можете напрямую написать несколько `print()` в одной строчке и разделить их запятыми).

Уровней вложенности кода может быть много. Ниже представлен пример, представляющий собой объединение рассмотренных ранее примеров.

```
---in↓---
a = 10
while a > 0:
    print(a)
    if a > 5:
        print("Большое число!")
    elif a % 2 != 0:
        print("Это нечетное число")
        print("И не больше пяти.")
    else:
        print("это число не больше 5")
        print("и не является нечетным")
    a = a - 1
    print("мы просто сделали на единицу меньше, чем было!")
    print ("и если a не больше 0, мы выполним цикл снова.")
print ("ну, похоже, что 'a' теперь не больше 0!")
print ("цикл окончен, и без дальнейших действий, и эта
программа тоже!")
---↑in_out↓---
10
Большое число!
мы просто сделали на единицу меньше, чем было!
и если a не больше 0, мы выполним цикл снова.
--часть вывода вырезана, так как она аналогична выводу числа
10–5
Это нечетное число
И не больше пяти.
```

```
мы просто сделали на единицу меньше, чем было!  
и если a не больше 0, мы выполним цикл снова.  
4  
это число не больше 5  
и не является нечетным  
мы просто сделали на единицу меньше, чем было!  
и если a не больше 0, мы выполним цикл снова.  
3  
Это нечетное число  
И не больше пяти.  
мы просто сделали на единицу меньше, чем было!  
и если a не больше 0, мы выполним цикл снова.  
2  
это число не больше 5  
и не является нечетным  
мы просто сделали на единицу меньше, чем было!  
и если a не больше 0, мы выполним цикл снова.  
1  
Это нечетное число  
И не больше пяти.  
мы просто сделали на единицу меньше, чем было!  
и если a не больше 0, мы выполним цикл снова.  
ну, похоже, что 'a' теперь не больше 0!  
цикл окончен, и без дальнейших действий, и эта программа  
тоже!  
---↑out---
```

Обратите внимание на **три уровня отступов**:

- каждая строка первого уровня начинается без пробелов. Это основная программа, и она всегда будет выполняться;

- каждая строка на втором уровне начинается с четырех пробелов. Когда есть `if` или цикл на первом уровне, все на втором уровне после этого будет заиклено, пока новая строка снова не начнется на первом уровне;

- каждая строка третьего уровня начинается с восьми пробелов. Когда есть `if` или цикл на втором уровне, все на третьем уровне после этого будет заиклено, пока новая строка снова не начнется на втором уровне.

Это продолжается бесконечно, пока у человека, пишущего программу, сохраняется понимание того, что он пишет.

В других языках программирования вложенность определяется иначе, например в C++ для определения вложенности используются фигурные скобки и точки с запятой вместо четырех пробелов и переноса строк соответственно.

Функции

Функции — это небольшие структуры кода, которые выполняют определенную задачу, когда вы запускаете их.

После того как вы создали функцию, вы можете использовать ее в любое время и в любом месте. Это экономит ваше время и усилия, связанные с необходимостью указывать один и тот же набор кода в разных местах. При этом если вам потребуется поменять код функции, то сделать это нужно будет только в одном месте — там, где мы его создали, а не во всех местах, в которых просто использовали.

Python имеет множество готовых функций, которые вы можете применять, просто «вызывая» их. Вызов функции (именно так называется запуск функции на выполнение) подразумевает, что вы вводите имя функции, выполняется вложенный в нее код, а затем она на месте своего имени возвращает значение (как переменная). Ниже представлен пример синтаксиса вызова функции: имя_функции (параметры) .

Имя функции определяет, какую функцию вы хотите применить. Например, функция `raw_input()` (с англ. — «сырой ввод») может использоваться для решения задачи, имя которой и несет. Например, обрабатывать «сырой ввод» значений с клавиатуры.

Параметры — это значения, которые вы передаете функции, чтобы сообщить ей, что она должна делать и как. Например, если функция создается для умножения любого требуемого числа на пять, то в параметрах указывается, какое число ее следует умножить на пять. Введите число 70 в параметры, тогда функция выполнит 70×5 и вернет результат данного действия.

Параметры и возвращаемые значения — взаимодействие с функциями

Теперь мы знаем, что придуманная и потенциально написанная нами пользовательская функция может умножать число на пять, но что она должна «показать» пользователю после своего выполнения? Результат умножения, который является (и называется) возвращаемым значением функции.

Говоря упрощенно, когда интерпретатор выполняет функцию, он «видит» не имя функции, а результат того, что функция «сделала», т.е. ее возвращаемое значение. Переменные делают то же самое: «компьютер не видит имени переменной»; при использовании переменной он оперирует только значением, которое содержит эта переменная.

Назовем функцию, которая умножает любое число на пять, именем «`multiply()`». При вызове функции (так, как уже было сказано, называется ее запуск на выполнение) вы помещаете в скобки число (это называется параметром), которое хотите умножить. Итак, если вы бы набрали следующее⁷:

```
a = multiply (70)
```

Интерпретатор в действительности «увидит»:

```
a = 350
```

Функция запустилась, а затем вернула число в основную программу в зависимости от того, какие параметры ей были заданы (в данном случае у нас был только один параметр).

Теперь давайте попробуем выполнить это с реальной функцией и посмотрим, что она делает. Для примера, в Python существует стандартная функция с названием «`input`», которая «просит» пользователя что-то ввести (на языке программистов — ожидает пользовательского ввода). Затем она превращает введенное значение в строку текста. Посмотрите на следующий код и результат его выполнения:

⁷ Не вводите этот код, `multiply()` не является стандартной функцией, встроенной в Python. Это пользовательская функция, которую мы создадим позже.


```
---in↓---
# эта строка делает 'a' равным тому, что вы вводите
a = input("Введите что-нибудь, и это будет
повторяться на экране:")
# эта строка печатает, чему равно значение 'a'
print(a)
---↑in_out↓---
Введите что-нибудь, и это будет повторяться на экране: 1
1
---↑out---
```

Допустим, в приведенной выше программе вы набрали текст «hello», когда она попросила вас ввести что-то. На компьютере эта программа будет выглядеть так:

```
---in↓---
a = "hello"
print("hello")
---↑in_out↓---
hello
---↑out---
```

Помните, что переменная — это сохраненное значение. Для компьютера переменная `a` не выглядит как `a`, она выглядит как значение, которое хранится внутри нее. Функции аналогичны основной программе (т.е. программе, в которой выполняется функция), они выглядят как значение того, что они дают в ответ на выполнение.

Функция `input()` сначала выводит текстовое сообщение, которое ей передали в качестве параметра, а затем считывает то, что введет пользователь, и после того, как он нажмет клавишу `enter`, она вернет ту строку, которую и вводил пользователь.

Лямбда-функции

Лямбда-выражение позволяет создавать анонимные функции, или, говоря упрощенно, создавать одной маленькой строчкой кода полноценную функцию, у которой не будет имени, и использоваться она будет сразу и только там, где вы ее написали. Ниже

представлено и прокомментировано несколько примеров использования лямбда-функций. В целом, никаких существенных отличий от обычных функций нет, и лямбда-функции бывают удобными в силу своей краткости.

Лямбда-функции ограничены одним выражением. Это означает, что лямбда-функция не может использовать никакие операторы, даже оператор `return`.

```

---in↓---
#1-й вариант. Именованная лямбда-функция
check_price = lambda price: price >= 2
print (check_price(2))
#2-й вариант. Неименованная лямбда-функция
print ((lambda price: price >= 2) (2))
#несколько параметров. Проверка, что цена*количество не
больше 9
print ((lambda price, quantity: price*quantity <= 9) (2, 10))
---↑in_out↓---
True
True
False
---↑out---
```

Программа «Калькулятор»

Напишем другую программу, которая будет выступать в роли калькулятора. На этот раз она будет выполнять что-то более масштабное, чем то, что мы делали раньше. Появится меню, в котором компьютер «спросит» пользователя, какую операцию он хочет выполнить: сложить, вычесть, разделить или перемножить два числа. Единственная проблема — функция `input()` возвращает то, что вы вводите в виде строки: нам нужна цифра 1, а не буква l (в Python есть разница, потому что переменные обладают типом).

К счастью, в Python есть встроенная функция `eval()`, которая выполняет любой Python-код, который вы ей передадите. Например, если вы передадите текст `'1+1'`, то она вернет результат, равный 2. Это сработает, потому что данный текст тоже является кодом, несмотря на то что в нем нет переменных, а есть только два числа и математическая операция сложения.

Используя `eval()`, мы можем избежать сложностей с ручным разбором строки на отдельные значения и математические операторы. Если вы введете число 1, то результат будет следующий:

```
---in↓---
# в этой строке "a" становится равным введенному вами
# значению. Она не принимает строки
a = eval(input("Введите что-нибудь, и это будет
повторено на экране: "))
# эта строка печатает, чему равно 'a'
print(a)
---↑in_out↓---
Введите что-нибудь, и это будет повторено на экране: 1
1
---↑out---
```

Теперь давайте спроектируем более сложный калькулятор. Например, мы хотим, чтобы нам возвращалось меню каждый раз, когда заканчивается сложение, вычитание и т.д. Другими словами, используем цикл, определяющий, должна ли программа работать дальше или калькулятор нам больше не нужен. Мы хотим, чтобы в меню была опция если «вы введете это число», то «осуществить выход из калькулятора».

Давайте сначала напомним модернизированную версию калькулятора на алгоритмическом языке:

НАЧАТЬ ПРОГРАММУ

вывести приветственное сообщение

```
# Распечатайте, какие у вас есть варианты
    печать Вариант 1 — сложить
    печать Вариант 2 — вычесть
    печать Вариант 3 — умножить
    печать Вариант 4 — разделить
    печать Вариант 5 — выйти из программы

спросить, какой вариант ты хочешь
если это вариант 1:
    спроси первое число
    спроси второе число
```

```
    сложить их вместе
    распечатать результат на экране
если это вариант 2:
    спроси первое число
    спроси второе число
    вычесть одно из другого
    распечатать результат на экране
если это вариант 3:
    спроси первое число
    спроси второе число
    умножить!
    распечатать результат на экране
если это вариант 4:
    спроси первое число
    спроси второе число
    разделить одно на другое
    распечатать результат на экране
если это вариант 5:
    скажите циклу прекратить цикл
Распечатать на экране прощальное сообщение
КОНЕЦ ПРОГРАММЫ
```

Реализуем данный псевдокод при помощи Python:

```
---in↓---
# программа-калькулятор

# эта переменная сообщает циклу, должен он заикливаться
или нет.
#1 означает цикл. Все остальное означает "не заикливаться".
loop = 1

# эта переменная содержит выбор пользователя в меню:
choice = 0

while loop == 1:
    #print what options you have
    print("Добро пожаловать в программу calculator.py")
    print("Доступны следующие функции:")
    print(" ")
```



```
print("1. Сложение")
print("2. Вычитание")
print("3. Умножение")
print("4. Деление")
print("5. Выход")
print(" ")

choice = eval(input("Осуществите выбор функции: "))
if choice == 1:
    add1 = eval(input("Прибавить: "))
    add2 = eval(input("к: "))
    print(add1, "+", add2, "=", add1 + add2)
elif choice == 2:
    sub2 = eval(input("Вычесть: "))
    sub1 = eval(input("из: "))
    print(sub1, "-", sub2, "=", sub1 - sub2)
elif choice == 3:
    mul1 = eval(input("Умножить: "))
    mul2 = eval(input("на: "))
    print(mul1, "*", mul2, "=", mul1 * mul2)
elif choice == 4:
    div1 = eval(input("Поделить это: "))
    div2 = eval(input("на: "))
    print(div1, "/", div2, "=", div1 / div2)
elif choice == 5:
    loop = 0
print("Спасибо вам за использование программы calculator.
py!")
---↑in_out↓---
```

Добро пожаловать в программу calculator.py
Доступны следующие функции:

1. Сложение
2. Вычитание
3. Умножение
4. Деление
5. Выход

1 + 1 = 2

Добро пожаловать в программу calculator.py
Доступны следующие функции:

1. Сложение
2. Вычитание

3. Умножение

4. Деление

5. Выход

Спасибо вам за использование программы `calculator.py`!

---↑out---

Поэкспериментируйте: попробуйте все варианты, вводя целые числа (числа без десятичной точки) и числа с добавлением после десятичной точки (известные в программировании как числа с плавающей запятой). Попробуйте ввести текст и посмотрите, как программа перестает работать (с этим можно справиться, используя обработку ошибок, о которой мы поговорим позже).

Пользовательские функции (свои собственные функции)

Ранее мы рассмотрели использование некоторых встроенных в Python функций. Если вы хотите написать свои собственные функции, то необходимо использовать оператор⁸ `def`.

Синтаксис использования оператора `def` следующий:

```
def function_name(параметр_1, параметр_2):
    {код функции}
    {больше кода}
    {больше кода}
    return {возвращаемое значение функции}
{этого кода нет в функции}
{потому что это без отступа}
# не забудьте поставить двоеточие ":" в конце
# строки, начинающейся с def
```

где `function_name` — это имя функции. Вы пишете код, который находится в функции под этой строкой, с отступом в четыре пробела. Мы еще поговорим о передаваемых (и принимаемых) функции (и функцией) параметрах `параметр_1` и `параметр_2` позже, а пока представьте, что между скобками нет ничего.

⁸ **Оператор** — это то, что говорит интерпретатору Python, что делать, например оператор «+» говорит Python добавить что-то, оператор `if` — что-то делать, если условия выполнены.

Функции выполняются полностью независимо от основной программы. Ранее было сказано, что, когда интерпретатор переходит к функции, он «видит» не функцию, а значение, которое функция возвращает. Для компьютера переменная выглядит как значение, которое хранится внутри нее. Для функции аналогично: они выглядят как значение того, что они дают в ответ на свое выполнение.

Функция похожа на миниатюрную программу, которой передаются некоторые параметры, затем она запускается сама и далее возвращает значение. Ваша основная программа «видит» только возвращаемое значение. Если бы эта функция в конце своего тела имела бы инструкцию возвращаемого значения `return` (англ. *return* — «вернуть»):

```
return "hello"
```

то тогда интерпретатор в вашей программе «увидит» строку «hello», на том месте, где было имя функции.

Поскольку это отдельная программа, функция не видит⁹ никаких переменных, которые есть основной программе, а ваша основная программа не видит никаких переменных, которые находятся в функции. Например, вот функция, которая выводит на экран слово «Привет», а затем возвращает число «1234» в основную программу:

```
---in↓---
# Ниже приведено определение пользовательской функции
def hello():
    print("Привет")
    return 1234
# А вот вызов (использование) данной пользовательской функции
print(hello())
---↑in_out↓---
```

⁹ Ранее слово «видит» использовалось в кавычках, так как не применялось в качестве термина. Здесь же слово «видит» является термином, потому что в программировании есть такое понятие, как «зона видимости». В данном контексте функция действительно «не видит переменных основной программы» (цитата), потому что они находятся вне ее зоны видимости. Она видит только переданные ей в параметрах переменные, либо свои внутренние переменные, либо переменные, объявленные внутри нее, но с флагом `global`.

```
Привет  
1234  
---↑out---
```

При выполнении «`def hello()`» была создана (объявлена) функция с именем `hello`. Когда была запущена строка «`print (hello())`», была выполнена функция `hello()` (и код внутри нее был запущен).

Функция `hello()` напечатала на экране `hello`, а затем вернула число «1234» обратно в основную программу. Основная программа теперь «видит» строку как `print («1234»)` и в результате выводит «1234».

Этим объясняется все, что произошло. Помните, что у основной программы не было представления о том, что на экране было напечатано слово `hello`. Все, что она «увидела», было «1234», и вывела это на экран.

Передача параметров функциям

Вспомните, как мы определяли функции:

```
def имя_функции (параметр_1, параметр_2):  
    {это код функции}  
    {дополнительный код}  
    {дополнительный код}  
    return {значение (например, текст или число), чтобы  
    вернуться в основную программу}
```

где вместо `параметр_1` и `параметр_2` вы помещаете имена переменных. Ввести можно столько параметров, сколько вам потребуется, достаточно просто разделять их запятыми.

Когда вы запускаете функцию, первое значение, которое вы помещаете в круглые скобки, переходит в переменную, где находится `параметр_1`. Второй (после первой запятой) параметр перейдет к переменной, в которой находится `параметр_2`. Это происходит для любого количества параметров функции (от нуля и больше). Например:


```
---in↓---
def funnyfunction(first_word, second_word, third_word):
    print("Слово: " + first_word + second_word + third_
word)
    return first_word + second_word + third_word
---↑in_out↓---
---↑out---
```

Когда вы запускаете (вызываете) указанную выше функцию, вы должны ввести что-то вроде: `funnyfunction («мясо», «людоед», «человек»)`. Первое значение (т.е. «мясо») будет помещено в переменную с именем `first_word`. Второе значение в скобках (то есть людоед) будет помещено в переменную с именем `second_word` и т.д. Вот как значения передаются из основной программы в функции, т.е. в скобках после имени функции.

Добавьте в приведенный выше код вызов функции.

Модернизация программы «Калькулятор»

Вернемся к программе «Калькулятор». Давайте перепишем ее с использованием функций, чтобы она была более читаемой. Сначала мы определим все функции, которые мы собираемся применить, с помощью оператора `def`. Затем у нас будет основная программа, в которой весь этот беспорядочный код будет заменен функциями.

```
---in↓---
# программа-калькулятор
# Здесь мы определим наши функции
# это печатает главное меню и предлагает выбор
def menu():
    #печать списка доступных функций
    print("Добро пожаловать в программу calculator.py")
    print("Доступны следующие функции:")
    print(" ")
    print("1. Сложение")
    print("2. Вычитание")
    print("3. Умножение")
```

```
    print("4. Деление")
    print("5. Выход")
    print(" ")
    return eval(input("Осуществите выбор функции: "))

# функция сложения
def add(a,b):
    print(a, "+", b, "=", a + b)

#функция вычитания
def sub(a,b):
    print(b, "-", a, "=", b - a)

#функция умножения
def mul(a,b):
    print(a, "*", b, "=", a * b)

# функция деления
def div(a,b):
    print(a, "/", b, "=", a / b)

loop = 1
choice = 0
while loop == 1:
    choice = menu()
    if choice == 1:
        add(eval(input("Прибавить: ")),eval(input("к: ")))
    elif choice == 2:
        sub(eval(input("Вычесть: ")),eval(input("из: ")))
    elif choice == 3:
        mul(eval(input("Умножить: ")),eval(input("на: ")))
    elif choice == 4:
        div(eval(input("Разделить: ")),eval(input("на: ")))
    elif choice == 5:
        loop = 0
    print("Спасибо за использование программы calculator.py!")
---↑in_out↓---
```

Добро пожаловать в программу calculator.py

Доступны следующие функции:

1. Сложение
2. Вычитание

```
3. Умножение
4. Деление
5. Выход
2 + 2 = 4
Добро пожаловать в программу calculator.py
Доступны следующие функции:
1. Сложение
2. Вычитание
3. Умножение
4. Деление
5. Выход
Спасибо за использование программы calculator.py!
---↑out---
```

В исходной программе было 34 строки кода. В новом коде фактически 35 строк. Он немного длиннее, но, если смотреть на него с точки зрения опытного программиста, на самом деле он проще и более читаем.

Вы определили все свои функции вверху файла программы. На самом деле это не часть вашей основной программы, это просто множество маленьких программ, которые вы вызовете позже¹⁰. Вы можете повторно использовать их в другой программе, не вынося из этого файла. Это будет рассмотрено на следующем занятии.

Если вы посмотрите на основную часть программы (между строками «loop=1» и «print("Спасибо за ...)»), вы увидите, что это всего 15 строк кода. Это означает, что, если бы вы захотели написать эту программу другим способом, вам нужно было бы использовать всего около 15 строк, в отличие от 34 строк, которые вам обычно приходилось бы писать без функций.

Хитрые способы передачи параметров

Рассмотрим, что означают следующие строки программы «Калькулятор»:

```
add(eval(input("Прибавить: ")),eval(input("к: ")))
```

¹⁰ Наступило время задуматься еще раз о том, что такое «точка входа», которая отражена на рис. 9 на стр. 27 и описана после него.

В строке выше происходит вызов пользовательской функции `add()` и передача ей значений двух параметров, каждое из которых было получено как возвращенное от `eval(input())`. В конечном счете, когда вы уже ввели два числа, например 2 и 20, то произойдет вызов `add()`-функции с этими параметрами:

```
add (2, 30)
```

Как более простой пример, в качестве параметров для функции сложения вы можете использовать обычные переменные вместо функций:

```
value1= 2  
value2 = 30  
add (value1, value2)
```

После вызова функции запустится ее код, т.е. он сложит полученные параметры 2 и 30, а затем распечатает результат. В функции сложения нет инструкции `return`, т.е. эта функция ничего не возвращает в основную программу. Она просто складывает два числа и сразу самостоятельно выводит их на экран, а основная программа ничего из этого не видит.

Функция, в которую вы передаете параметры, не может изменять исходные параметры. Единственное, что видит функция, — это значения, которые ей передаются в качестве параметров. Эти значения помещаются в переменные, которые упоминаются при определении `add()` (строка «`def add (a, b)`»). Затем функция использует эти параметры для выполнения своей работы.

Это определяется зоной видимости, про которую упомянуто в сноске 9 на стр. 66. Чтобы функции были доступны некоторые переменные из основной программы, нужно внутри функции разрешить к ним доступ через ключевое слово `global`. Иначе будут созданы новые локальные переменные с аналогичным именем внутри функции. Подробнее про область видимости в Python читайте самостоятельно.

Подытожим вышесказанное про функции:

- единственное, что функции основной программы видят, — это параметры, которые им передаются;
- единственное, что основная программа видит из функций, — это их возвращаемое значение, которое функция возвращает.

Задания для проверки

1. Что делает функция `print()` и как с ее помощью выводить значение переменной? Опишите значение параметра `end`. Как вывести две строки одной строчкой (сложить строки)?

2. Опишите существующие типы переменных в Python (строки, числа и др.).

3. Как обрезать строки?

4. Что такое цикл `while`? Объясните на примере.

5. Сделайте собственную функцию, которая бы вычисляла сумму двух чисел с плавающей и проверяла, больше ли она числа «пи». Если больше, то сделайте вывод текста «больше...», и наоборот.

НАБОРЫ ДАННЫХ, ЦИКЛ FOR, ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ, РАБОТА С ФАЙЛАМИ И ИСКЛЮЧЕНИЯ¹¹

Цели выполнения работы

Получение базовых знаний и основных принципов практической работы с ними по следующим основам языка программирования Python:

- кортежи, списки, словари и массивы;
- цикл `for`;

¹¹ Разработано на основе блокнотов из англоязычного источника [13].

- классы, наследование, указатели и словари классов, модули;
- файловый ввод-вывод;
- обработка исключений.



Порядок выполнения работы

Кортежи, списки, словари и массивы

Продолжая материал, рассмотренный на предыдущем занятии, вернемся к переменным. Переменные хранят одну единицу информации.

Что, если вам нужно хранить длинный список информации? Например, названия месяцев в году. Или, может быть, длинный набор больших данных? Для решения подобных задач существуют различные типы **наборов данных**, которые могут хранить не только одно значение, но и несколько сразу, — это массивы, кортежи, списки и словари:

- массивы хранят набор данных одного и того же типа. Каждое из них нумеруется начиная с нуля. Первый элемент имеет нулевой индекс, второй — 1, третий — 2 и т.д. Пример: строковые имена ваших контактов;

- списки — то же самое, что и массивы, но могут хранить значения не одного типа, а разных: строковых, числовых и др. — и при этом одновременно. Вы можете удалять значения из списка и добавлять новые в конец. Пример: имена ваших контактов, где имя задано не только строкой, но и числом или любым типом вперемешку;

- кортежи похожи на списки, но вы не можете изменять их значения. Значения, которые вы задаете его элементам, устанавливаются только один раз — при создании (определении) кортежа, и вы придерживаетесь их весь последующий период работы программы. Опять же, каждое значение пронумеровано начиная с нуля, для удобства. Пример: названия месяцев в году;

- словари. В словаре есть «указатели» на слова и значение каждого соответствующего своему указателю слова. Указатель называется ключом, и ключи используются вместо индексов, т.е. значения

в словаре не нумеруются. Вы можете добавлять, удалять и изменять значения в словарях. Пример: телефонная книга (имя контакта — это ключ, а значение — это телефонный номер).

Кортежи

Кортежи создаются следующим образом: вы даете кортежу имя, а затем список значений, которые он будет хранить. Например, месяцы года:

```
months = ('Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', \
          'Июль', 'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь')
```

Обратите внимание, что знак «\» (обратный слэш) в конце первой строки кода переносит ее на следующую строку. Это полезный способ сделать большие строки более читабельными, и он никак не влияет на работу вашей программы.

Технически язык Python позволяет не писать круглые скобки при объявлении кортежа, но лучше их ставить (в целях обеспечения читабельности кода). У вас могут быть пробелы после запятых, если вы считаете это необходимым, но на самом деле это также не имеет значения.

Большинство IDE имеют функцию, которая позволяет улучшать читаемость кода автоматически. Либо можно воспользоваться средствами форматирования кода (англ. *code formatter* или *code beautifier* — «украшатель кода»), доступными даже как онлайн-сервисы в сети Интернет, которые приведут ваш код к рекомендуемым стандартам языка (стандартам внешнего вида).

Подробные стандарты по написанию Python кода представлены в стандарте PEP-8 (от англ. *Python Enhancement Proposals* — «предложения по усовершенствованию Python», где номер 8 — руководство по стилю кода Python), доступном в источнике [14].

Вернемся к рассмотрению примера кортежа с именами месяцев года. Python организует эти значения (имена месяцев) и присваивает им нумерацию, т.е. индексы, начиная с нуля и далее в положительную сторону, в том порядке, в котором вы их вводили. Например, созданный ранее кортеж месяцев будет организован следующим образом (табл. 4).

Таблица 4

Кортеж имен месяцев года

Индекс	Значение
0	Январь
1	Февраль
---часть таблицы вырезана---	
11	Декабрь

Это и есть кортеж. Он прост для понимания. Как с этим работать, будет рассмотрено далее.

Списки

В отличие от кортежей, списки изменяемы (англ. *mutable* — «изменяемы», но в программистской среде часто принято говорить «мутабельны»), т.е. их значения могут быть изменены в ходе работы программы. В большинстве случаев мы используем списки, а не кортежи, именно потому, что мы имеем возможность изменять значения элементов, если нам это потребуется.

Списки определяются похожим на определение кортежей образом. Допустим, у вас есть пять кошек, которых зовут Том, Снэппи, Китти, Джесси и Честер. Чтобы поместить их в список, нужно выполнить следующий код:

```

---in↓---
cats = ['Tom', 'Snappy', 'Kitty', 'Jessie', 'Chester']
---↑in_out↓---
---↑out---
```

Как показано выше, код создания (объявления с определением) в точности такой же, как при создании кортежа, за исключением того, что все значения заключены в квадратные, а не круглые, скобки. Опять же, вам не обязательно ставить пробелы после запятой.

Вы можете использовать значения из списков точно так же, как и с кортежами. Например, чтобы напечатать имя третьего кота, вы можете воспользоваться следующим способом:

```
---in↓---
print(cats[2])
---↑in_out↓---
Kitty
---↑out---
```

Мы используем индекс, равный 2, для третьего кота, потому что индексация начинается с нуля.

Вы также можете вспомнить ряд операторов работы со строками, например оператор обрезки «:» (двоеточие) в коде `cats[0:2]` выведет имена котов из диапазона (0:2], т.е. имена первого и второго котов, как показано ниже:

```
---in↓---
print(cats[0:2])
---↑in_out↓---
['Tom', 'Snappy']
---↑out---
```

Списки сами по себе могут быть изменены. Чтобы добавить значение в список, примените функцию `append()` (англ. *append* — «добавить»). Допустим, у вас появилась новая кошка по имени Кэтрин. Чтобы добавить ее в список, сделайте следующее:

```
---in↓---
cats.append('Catherine')
---↑in_out↓---
---↑out---
```

В Python есть два способа обращения к последнему элементу в списке:

- `cats[-1]`, в Python отрицательная индексация начинается с конца (т.е. справа налево), а значит, первый с конца индекс и есть индекс последнего элемента;

- `cats[len(cats)-1]`, где функция `len()` возвращает длину объекта, т.е. в данном случае размер списка. Так как индекс перво-

го элемента равен нулю, а не единице (как привычно человеку), то, чтобы получить индекс последнего элемента, необходимо из размера списка вычесть единицу.

Использование этих двух способов показано ниже:

```

---in↓---
print (cats[-1]) # вывод последнего элемента
print (cats[ len(cats) - 1 ]) # вывод последнего элемента
print (cats) # вывод всех элементов
---↑in_out↓---
Catherine
Catherine
['Tom', 'Snappy', 'Kitty', 'Jessie', 'Chester',
'Catherine']
---↑out---
```

Синтаксис конструкции добавления нового элемента `cats.append('Catherine')` может показаться странным. Эта функция находится в конце названия списка, после точки. В программировании это называется методом. Метод — это функция, принадлежащая какому-то объекту (точка и означает эту принадлежность). Больше об этом будет рассказано позже. В данном случае объектом является список, и у него есть встроенный в Python метод `append()`. Рассмотрим синтаксис добавления нового значения в список:

```

# добавить новое значение в конец списка:
list_name.append(значение, которое вы хотите добавить)

#например, добавить число 5038 в список 'числа'
numbers.append(5038)
```

Для добавления сразу множества элементов, а не только одного элемента, можно использовать метод `extend()`.

Теперь рассмотрим ситуацию удаления кота Снэппи из списка (например, потому что он пропал). Удаление элементов из списка производится при помощи метода `del()`:

```
---in↓---
#Удаление кота с индексом 1, т.е. второго кота.
del cats[1]
print (cats)
---↑in_out↓---
['Tom', 'Kitty', 'Jessie', 'Chester', 'Catherine']
---↑out---
```

Кот Снэппи действительно удален.

Словари

Рассмотрим другой пример. Вам нужно позвонить своей сестре, маме, сыну и всем, кому нужно знать, что их любимый кот пропал. Для этого вам понадобится телефонная книга.

Итак, списки, которые мы использовали выше, не совсем подходят для телефонной книги. Вам нужно знать число (представляющее собой телефонный номер), основанное на чьем-либо имени, а не наоборот, как мы делали со списком кошек (т.е. обращались к конкретным именам кошек по индексу).

В примерах месяцев и кошек мы указали интерпретатору номер элемента (т.е. индекс, который даже нельзя изменить, так как он проставляется автоматически), а интерпретатор вернул нам имя кота, соответствующее этому индексу. В примере с телефонной книгой нам необходимо обращаться к элементам не по индексу, а по именам контактов. Для этого нам понадобятся словари.

У словарей есть пары «ключ — значение». В телефонной книге находятся имена людей, а затем соответствующие им телефонные номера, каждая такая пара и является парой «ключ — значение» (имя контакта и его телефонный номер).

Создание словаря очень похоже на создание кортежа или списка. Кортежи объявляются при помощи круглых скобок «()», списки — при помощи квадратных «[]». При определении или объявлении словарей используются фигурные скобки «{ }». Ниже представлен пример словаря, элементами которого являются четыре телефонных номера:

```

---in↓---
#Создать телефонную книгу, представляющую собой словарь Python
phonebook = {'Andrew Parson':8806336, \
'Emily Everett':6784346, 'Peter Power':7658344, \
'Lewis Lane':1122345}
print('Телефон контакта \"Lewis Lane\":
',phonebook['Lewis Lane'])
---↑in_out↓---
Телефон контакта "Lewis Lane":  1122345
---↑out---
```

Номер Льюиса Ламе напечатан на экране. Обратите внимание, как вместо определения значения числовым индексом, как в примерах с кошками и месяцами, мы определяем значение, используя другое в качестве индекса, или ключа, в данном случае имя человека. То есть при определении словаря вы указываете не только значения его элементов, а вместо значений определяете пару «ключ — значение», в которой сначала идет ключ, который отделяется от соответствующего ему значения при помощи двоеточия, а затем само значение.

В примере значения являются целочисленными, но можно также использовать строковые или любые другие объекты, например в значениях могут храниться вложенные массивы и т.п.

Теперь добавим в книгу новые телефонные номера:

```

---in↓---
# Добавьте человека 'Gingerbread Man' (по-русски -
"Колобок") в телефонную книгу:
phonebook['Gingerbread Man'] = 1234567
---↑in_out↓---
---↑out---
```

Представленная выше строка означает добавление нового элемента в словарь, у которого (у элемента) ключом будет «Gingerbread Man», а значением — номер «1234567».

Проверьте, добавлен ли новый контакт в словарь, используя код ниже:


```
---in↓---
if phonebook.get('Gingerbread Man') != None:
    print ('Да, Gingerbread Man добавлен')
---↑in_out↓---
Да, Gingerbread Man добавлен
---↑out---
```

Удаление записей в словаре происходит аналогично, как и в списке. Допустим, Эндрю Парсон — ваш сосед и он украл вашу кошку. Вы больше никогда не захотите с ним разговаривать, а значит, вам не нужен его номер. Для его удаления можно использовать представленный ниже код:

```
---in↓---
del phonebook['Andrew Parson']
---↑in_out↓---
---↑out---
```

Оператор `del` удаляет любой элемент в списке или словаре. В словаре элементы называются записью. Запись — это пара «ключ — значение».

Проверим, пропал ли номер, используя представленный ниже код. Если `phonebook.get()` возвращает `None` (можно перевести как «ничего/никто/нет/пустой», эквивалент нуля), значит, такой записи нет.

```
---in↓---
print (phonebook.get('Andrew Parson'))
---↑in_out↓---
None
---↑out---
```

Следующий же код вызовет ошибку, поскольку мы пытаемся обратиться к несуществующему ключу:

```
---in↓(raises-exception)---
print (phonebook['Andrew Parson'])
---↑in_out↓---
```

```
-----
-----
KeyError                                Traceback (most
recent call last)
<ipython-input-13-5fab7f37904b> in <module>
      1 #а это вызовет ошибку, ведь мы пытаемся
обратиться к несуществующему ключу
----> 2 print (phonebook['Andrew Parson'])
KeyError: 'Andrew Parson'
---↑out---
```

Функций для работы с наборами данных (списки, словари и др.), таких как, например, уже рассмотренные методы добавления `append()` и удаления элементов `del()`, довольно много.

Ниже представлена программа, в которой отражены некоторые из наиболее интересных доступных методов для работы со словарями (многие аналогичные методы доступны и для списков, массивов и других наборов данных). В коде есть комментарии, объясняющие последовательность его работы.

```
---in↓---
# Несколько примеров работы со словарем

# Сначала определяем словарь
# на этот раз в нем ничего не будет
возраст = {}
ages = {}

#Добавляем несколько имен в словарь
ages['Sue'] = 23
ages['Peter'] = 19
ages['Andrew'] = 78
ages['Karren'] = 45

# Используем оператор if, чтобы найти ключ в списке.
# Помните, как работают операторы if -
# они выполняют код внутри своего тела, если что-то ИСТИНО
# и не выполняют, когда что-то ЛОЖНО.
if 'Sue' in ages:
    print("Sue есть в словаре. Ей", \
ages['Sue'], " года")
```

```
else:
    print("Sue нет в словаре.")

#Используйте функцию keys() -
#Эта функция возвращает список
# всех названий ключей. Например:
print("В словаре есть следующие люди: ")
print(ages.keys())

# Вы можете использовать эту функцию, чтобы
# поместить все имена ключей в список:
keys = ages.keys()

# Вы также можете получить список
# всех значений в словаре при помощи метода values ():
print("Возраст людей следующий: ", \
ages.values())

## Поместите возраст в список:
values = ages.values()

# Вы можете сортировать списки с помощью функции sorted ()
# Отсортирует все значения в списке
# по алфавиту, численно и т.д.
# Словари нельзя сортировать -
# они в произвольном порядке
print(keys)
sortedkeys = sorted(keys)
print(sortedkeys)

print(values)
sortedvalues = sorted(values)
print(sortedvalues)

# Вы можете узнать количество записей
# при помощи функции len ():
print("В словаре ", \
len(ages), " записи")
---↑in_out↓---
```

```
Sue есть в словаре. Ей 23 года
В словаре есть следующие люди:
dict_keys(['Sue', 'Peter', 'Andrew', 'Karren'])
Возраст людей следующий: dict_values([23, 19, 78, 45])
dict_keys(['Sue', 'Peter', 'Andrew', 'Karren'])
['Andrew', 'Karren', 'Peter', 'Sue']
dict_values([23, 19, 78, 45])
[19, 23, 45, 78]
В словаре 4 записи
---↑out---
```

Проведите несколько экспериментов с представленным выше кодом.

Массивы

Отличия массивов от словарей представлены ниже (табл. 5). Основное отличие заключается в том, что массивы хранят данные только в одном конкретном типе.

Таблица 5

Сравнение списков и массивов

Список	Массив
Может состоять из элементов, принадлежащих к разным типам данных	Состоит только из элементов, принадлежащих к одному типу данных
Нет необходимости явно импортировать модуль для объявления	Необходимо явно импортировать модуль <code>array</code> при помощи « <code>import array</code> »
Невозможно напрямую обрабатывать арифметические операции	Может напрямую обрабатывать арифметические операции
Могут быть вложенными для размещения элементов различного типа	Может содержать вложенные элементы, но они должны быть одинакового размера
Предпочтителен для более малых наборов данных	Предпочтителен для больших наборов однородных данных

Окончание табл. 5

Список	Массив
Большая гибкость позволяет легко изменять (добавлять, удалять) данные	Меньшая гибкость после добавления, удаление должно выполняться поэлементно
Весь список можно распечатать без явного зацикливания	Цикл должен быть сформирован для печати или доступа к компонентам массива
Использует повышенный объем памяти для удобного добавления элементов	Сравнительно компактнее по объему памяти

Обычно массивы используются только при системном программировании на Python, поскольку там могут повыситься скорость выполнения кода.

Списки более удобны и работают с достаточной в большинстве случаев скоростью, поэтому лучше пользоваться ими.

В табл. 6 представлено сравнение доступных в Python типов переменных, которые могут принимать массивы, с аналогичными типами в языке программирования C.

Таблица 6

Типы переменных в C и Python

Код типа	C тип	Python тип	Минимальный размер (байт)
b	signed char	int	1
B	unsigned char	int	1
u	Py_UNICODE	Unicode символ; устарело с Python 3.3	2
h	signed short	int	2
H	unsigned short	int	2
i	signed int	int	2

Окончание табл. 6

Код типа	C тип	Python тип	Минимальный размер (байт)
I	unsigned int	int	2
l	signed long	int	4
L	unsigned long	int	4
q	signed long long	int	8
Q	unsigned long long	int	8
f	float	float	4
d	double	float	8

Сравните два примера ниже. В первом — работа с массивом, во втором — работа со списком.

```

---in↓---
#Пример — массив
import array

arr=array.array('i',) #объявление массива
arr=[1,3,4] #определение массива
print (arr)
---↑in_out↓---
[1, 3, 4]
---↑out---

---in↓---
#Пример — список
myList=list() #объявление списка
myList = [1,3,4] #определение списка
print (myList)
---↑in_out↓---
[1, 3, 4]
---↑out---
```

В примере выше показано, что сначала можно объявлять (создавать) массив «arr=array.array('i',)», а потом определять

(т.е. заполнять значениями). Это полезно, когда код большой, очень гибкий и легко масштабируемый.

Так же и с любыми другими объектами. Например, вы можете сначала объявить список, т.е. создать пустой список, а потом определить, заполнив его значениями:

```
---in↓---
phonebook1 = {} #или так phonebook1=list()
phonebook1={'Andrew Parson':8806336, 'Emily
Everett':6784346}
print (phonebook1)
---↑in_out↓---
{'Andrew Parson': 8806336, 'Emily Everett': 6784346}
---↑out---
```

Цикл for

Ранее уже был рассмотрен цикл `while`. На текущий момент вы получили базовые понятия о списках, поэтому теперь можно рассмотреть и цикл `for`.

Цикл `for` часто используется, чтобы что-то сделать с каждым значением в списке или повторить какой-либо участок кода несколько раз. Рассмотрим пример:

```
---in↓---
# Пример цикла for
# Сначала создадим список:
newList = [45, 'eat me', 90210, «The day has come, the
walrus said, \
to speak of many things", -67]

# создаем цикл:
# цикл проходит по каждому элементу списка newList
и печатает его
for value in newList:
    print(value)
---↑in_out↓---
45
eat me
```

```
90210
```

```
The day has come, the walrus said, to speak of many things
-67
---↑out---
```

Когда цикл выполняется, происходит последовательный перебор всех значений в списке, указанном после «in». На каждом новом шаге (итерации) цикла значение «активного» элемента списка (т.е. элемента с индексом, соответствующим номеру текущей итерации выполнения данного цикла) записывается в переменную `value` и печатается.

Рассмотрим еще один аналогичный пример:

```
---in↓---
word = 'Привет'
index = 0
for letter in word:
    print("итерация", index, letter)
    index += 1 #прибавить к индексу 1
---↑in_out↓---
итерация 0 П
итерация 1 р
итерация 2 и
итерация 3 в
итерация 4 е
итерация 5 т
---↑out---
```

Обратим внимание:

- как показано в примере выше, строки — это списки с большим количеством символов;

- программа последовательно перебрала каждую букву (или значение) в слове и распечатала их на экране.

Однако пройти весь список целиком можно было и при помощи цикла `while`, например так:

```
---in↓---
word = 'Привет'
index = 0
```



```
while index < len(word): #не определенные ранее переменные
    по умолчанию получают значение, равное нулю. То есть i=0
    print("итерация",index, word[index])
    index += 1 #прибавить к индексу 1
---↑in_out↓---
итерация 0 П
итерация 1 р
итерация 2 и
итерация 3 в
итерация 4 е
итерация 5 т
---↑out---
```

Чем можно воспользоваться, если необходимо производить итерации цикла по конкретному диапазону индексов? Для решения подобной задачи доступна функция `range()`.

Функция `range()`

Функция `range()` (англ. *range* — «диапазон») генерирует последовательность целых чисел в указанном диапазоне. **Данная функция может принимать один, два или три аргумента:**

- если задан только один аргумент, то генерируются числа от 0 до указанного числа, не включая его;
- если заданы два аргумента, то числа генерируются от первого числа до второго, не включая его;
- если заданы три аргумента, то третье число — это шаг.

Например, `range(5, 11)` сгенерирует последовательность «5, 6, 7, 8, 9, 10», при этом она не будет являться списком, словарем или чем-либо иным. Функция `range()` производит объекты своего собственного класса¹² — диапазоны:

```
---in↓---
a = range(-10, 10)
print (a)
```

¹² Классы будут рассмотрены позже. Класс — это еще один тип в языке Python.

```

print (type(a))
range(-10, 10)
---↑in_out↓---
<class 'range'>
---↑out---
```

Несмотря на то что мы не видим последовательности чисел, она есть, и мы можем обращаться к ее элементам:

```

---in↓---
print ('первый элемент: ',a[0])
print ('последний элемент:',a[-1])
---↑in_out↓---
первый элемент:  -10
последний элемент: 9
---↑out---
```

Обратите внимание, что последний элемент последовательности равен 9, а не 10. Это объясняется тем, что вызовом `range(-10, 10)` числа генерируются от первого включительно и до второго не включительно чисел, указанных в параметрах соответственно.

Итак, зачем нам понадобилась функция `range()` в теме про цикл `for`? Дело в том, что вместе они образуют неплохой тандем. `For` как цикл перебора элементов, в отличие от `while`, позволяет не следить за тем, достигнут ли конец условия. Не нужно вводить счетчик для этого, изменять его в теле и проверять условие в заголовке. Функция `range()` возвращает последовательность целых чисел, которые можно использовать как индексы для элементов того же списка:

```

---in↓---
#получим диапазон индексов букв в слове
#функция len() возвращает длину объекта, т.е. в данном
случае длину слова
range(len(word))
---↑in_out↓---
range(0, 6)
---↑out---
```

Здесь с помощью функции `len()` измеряется длина списка (список в данном случае — это набор букв «Привет»). Длина списка равна 6.

После этого число 6 передается в функцию `range()`, и она генерирует последовательность чисел от 0 до 5 включительно. Это как раз индексы элементов нашего списка. Теперь «соединим» `for` и `range()`:

```
---in↓---
index = 0
for index in range(len(word)):
    print("итерация",index, word[index])
---↑in_out↓---
итерация 0 П
итерация 1 р
итерация 2 и
итерация 3 в
итерация 4 е
итерация 5 т
---↑out---
```

В заголовке цикла `for` берутся элементы объекта `range`, а не исходного не списка. Список, элементы которого планируется перезаписывать, тут, по сути, не фигурирует. Если заранее знать длину списка, то заголовок может выглядеть так: `for index in range(6)`. То, как используется `index`¹³ в теле цикла, — другой вопрос.

Еще доступна весьма полезная функция `zip()`, которая принимает на вход несколько списков и создает из них один список (в Python 3 создается не `list` (список, т.е. не список списков) объект, а специальный `zip`-объект) кортежей, такой, что первый элемент полученного списка содержит кортеж из первых элементов всех списков-аргументов. Таким образом, если ей передать два списка, то она отработает следующим образом:

¹³ Вместо идентификатора «`index`» может быть любой другой идентификатор или даже несколько, которые будут последовательно итерировать соответствующие им уровни массивов. Это удобно использовать при итерациях по элементам многомерных массивов (наборов данных).

```

---in↓---
list1 = 'abc'
list2 = [10, 20, 30]
print (list(zip(list1,list2)))
---↑in_out↓---
[('a', 10), ('b', 20), ('c', 30)]
---↑out---
```

При помощи функции `zip()` мы можем сделать цикл `for` более непонятным на первый взгляд (попробуйте понять, как это работает):

```

---in↓---
my_list = [['Петя', 'Коля', '1'], ['Дурак', 'Молодец',
'2345']]

new_list=list(zip(my_list[0],my_list[1]))
print (new_list)

print ('----')

for i, j in new_list:
    print(i,j)

print ('----\nТо же самое, обратите внимание на индексы
В КОДЕ:')
print (new_list[0][0],new_list[0][1])
print (new_list[1][0],new_list[1][1])
print (new_list[2][0],new_list[2][1])
---↑in_out↓---
[('Петя', 'Дурак'), ('Коля', 'Молодец'), ('1', '2345')]
----

Петя Дурак
Коля Молодец
1 2345
----
То же самое, обратите внимание на индексы В КОДЕ:
Петя Дурак
```


Коля Молодец

1 2345

---↑out---

То есть, написав после `for` две переменных, первая будет итератором верхнего уровня списка, а вторая — итератором более низкого уровня списка: `new_list[i][j]`.

Пример. Создание функции «меню»

Приступим к написанию более сложных программ. До сих пор мы изучили переменные, списки, циклы и функции. Это почти все, что нам нужно для базового программирования.

Итак, давайте поставим перед собой задачу:

- программа запрашивает строку со всеми пунктами меню в ней и текстовую строку с вопросом;
- необходимо наличие проверки, что каждый пункт меню уникален.

Ниже представлен пример создания пользовательской функции «menu»:

```
---in↓---
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print (" " + entry)
    return eval(input(question)) - 1
---↑in_out↓---
---↑out---
```

Из кода «`def menu (list, question)`» видно, что наша функция требует два параметра для работы:

- список всех пунктов меню;
- вопрос, который она задаст, когда все параметры будут напечатаны.

Для каждой записи в списке выполняются следующие действия:

- код «`print(1 + list.index(entry),end=»»)`» использует метод `index()` для поиска, где в списке находится запись. Функция

`print()` затем распечатает ее номер, увеличенный на 1, чтобы нумерация начиналась не с 0, а с 1 и была более понятной;

- код «`print «)» + entry`» печатает скобку, а затем имя записи;

- после завершения цикла `for «eval(input(question) - 1)»` задает вопрос и возвращает значение индекса пункта меню в основную программу.

Фактически программа заняла всего пять строк.

Было бы лучше, если бы все описанные комментарии были оставлены внутри кода с помощью изученной на предыдущем занятии техники оставления комментариев. Помните, что, если вы собираетесь публиковать свой код в общий доступ, многие люди будут его «читать», и чтобы его можно было быстрее понять, лучше оставлять комментарии внутри кода.

Попробуйте описанное здесь вставить в исходный код как комментарии, используя однострочные комментарии при помощи символа решетки «`#`» и многострочные при помощи «`"""`» (см. стр. 39, раздел «Комментарии»).

Грамотно расставляя комментарии в коде, будет проще понимать как суть самого кода, так и суть комментариев.

Наша первая «игра»

В качестве примера использования цикла `for` рассмотрим текстовую приключенческую игру.

Ее сюжет будет происходить в одной комнате дома, в которой есть пять вещей и дверь. В одной из пяти вещей спрятан ключ от двери. Игроку нужно найти ключ, затем с его помощью открыть дверь.

Сначала опишем программу на русском алгоритмическом языке, а затем реализуем ее на Python:

Вывести доступные функции меню программы

Вывести приветственное сообщение с описанием комнаты.

Выведем список из шести доступных вещей: горшечные

растения, живопись, \

ваза, абжур, тудля и дверь

Компьютер считает, что дверь заперта, и знает, где ключ

Ввести меню, показывающее, чем вы можете "управлять":
вывести 6 предметов, один из которых пользователь может
выбрать, чтобы посмотреть

если пользователь захотел посмотреть на предмет:

Горшечное растение:

Если ключ здесь, отдайте ключ игроку

в противном случае скажите, что его здесь нет

картина:

то же, что и выше

и т.п.

дверь:

Если у игрока есть ключ, пусть откроет дверь

В противном случае попросите его присмотреться

Сообщить игроку о завершении игры.

Основываясь на этих исходных данных, мы можем написать программу:

```
---in↓---
```

```
# Текстовая приключенческая игра
```

```
#функция, реализующая функциональность меню
```

```
def menu(list, question):
```

```
    for entry in list:
```

```
        print(1 + list.index(entry),end="")
```

```
        print (" " + entry)
```

```
    return eval(input(question)) - 1
```

```
# Сообщим компьютеру основную информацию о комнате:
```

```
items = ["Цветок в горшке","Картина","Ваза","Абажур",  
"Обувь","Дверь"]
```

```
# Ключ находится в вазе (или запись номер 2 в списке выше):
```

```
keylocation = 2
```

```
# Переменная, в значении которой хранится, нашли вы ключ
или нет. 0 - не нашли
keyfound = 0

loop = 1 #переменная для зацикливания

# Дадим вводный текст:
print("Прошлой ночью вы легли спать, не выходя из
собственного дома.")

print("Теперь вы заперты в комнате. Вы не знаете, как")
print("вы туда попали или сколько сейчас времени.
В комнате вы видите")
print(len(items), "вещей:")
for x in items:
    print(x)
print("")
print("Дверь заперта. Может быть, где-нибудь можно найти
ключ?")

#Пусть ваше меню выводится циклически, пока вы не найдете
ключ:
while loop == 1:
    choice = menu(items, "Что вы хотите проверить? ")
    if choice == 0:
        if choice == keylocation:
            print("Вы нашли ключик в горшке.")

            print("")
            keyfound = 1
        else:
            print("Вы ничего не нашли в горшке.")
            print("")
    elif choice == 1:
        if choice == keylocation:
            print("Вы нашли маленький ключик за картиной.")
            print("")

            keyfound = 1
        else:
            print("Вы ничего не нашли за картиной.")
            print("")
```



```
elif choice == 2:
    if choice == keylocation:
        print("Вы нашли в вазе маленький ключик.")
        print("")
        keyfound = 1
    else:
        print("Вы ничего не нашли в вазе.")

        print("")
elif choice == 3:
    if choice == keylocation:
        print("Вы нашли маленький ключик в абажуре.")
        print("")
        keyfound = 1
    else:
        print("Вы ничего не нашли в абажуре.")
        print("")
elif choice == 4:
    if choice == keylocation:
        print("Вы нашли ключик в туфле.")
        print("")
        keyfound = 1
    else:
        print("Вы ничего не нашли в туфле.")
        print("")
elif choice == 5:
    if keyfound == 1:
        loop = 0
        print("Вы вставляете ключ, поворачиваете его
и слышите щелчок.")

        print("")
    else:
        print("Дверь заперта, нужно найти ключ.")
        print("")

# Помните, что обратная косая черта продолжает
# код в следующей строке
print("Свет заливает комнату так, как будто \
вы открываете дверь к своей свободе.")
```

---↑in_out↓---

Прошлой ночью вы легли спать, не выходя из собственного дома.
Теперь вы заперты в комнате. Вы не знаете, как
вы туда попали или сколько сейчас времени. В комнате вы видите
6 вещей
Цветок в горшке
Картина
Ваза
Абажур
Обувь
Дверь

Дверь заперта. Может быть, где-нибудь можно найти ключ?

- 1) Цветок в горшке
- 2) Картина
- 3) Ваза
- 4) Абажур
- 5) Обувь
- 6) Дверь

Вы нашли в вазе маленький ключик.

- 1) Цветок в горшке
- 2) Картина
- 3) Ваза
- 4) Абажур
- 5) Обувь
- 6) Дверь

Вы вставляете ключ, поворачиваете его и слышите щелчок.

Свет заливает комнату так, как будто вы открываете дверь
к своей свободе.

---↑out---

Очень простая программа, демонстрирующая многие из уже рассмотренных основ программирования на Python. Пусть вас не пугает объем кода в 53 строки, потому что в коде очень много операторов `if`, которые программисту очень легко «читать» (как только вы поймете все отступы).

Улучшение игры

Функция `menu()` сокращает объем кода. Цикл `while`, который у нас есть, плохо читаем — четыре уровня отступов для простой программы. Давайте немного перепишем код (это называется «рефакторинг»¹⁴), чтобы он стал более читаемым.

Код сильно улучшится, когда мы введем классы. Но это будет рассмотрено позже. А пока давайте создадим функцию, которая улучшит читабельность кода. Мы передадим ей два параметра — сделанный нами выбор меню и расположение клавиши. Она вернет одно значение — был ли ключ найден.

```
---in↓---
def inspect(choice, location):
    if choice == location:
        print("\nВы нашли ключ!\n")
        return 1
    else:
        print("\nНичего интересного тут...\n")
        return 0
---↑in_out↓---
---↑out---
```

В ранее представленном коде игры произведите следующие **изменения**:

1. Вставьте функцию `inspect()`.
2. Замените цикл `while` следующим кодом:

```
---in↓---
while loop == 1:
    keyfound = inspect(menu(items, "Что вы хотите
    проверить? "), keylocation)
    if keyfound == 1:
```

¹⁴ В программировании термин «рефакторинг» означает изменение исходного кода программы без изменения его внешнего поведения. Выполняется для улучшения понятности кода или изменения его структуры, для удаления «мертвого кода», все это нужно для того, чтобы в будущем код было легче поддерживать и развивать.

```
print("Вы вставляете ключ в замок двери,  
поворачиваете его и слышите щелчок!")  
loop = 0  
---↑in_out↓---  
---↑out---
```

Теперь программа стала намного короче и читабельнее — ее размер уменьшился с 83 строк до 50.

Конечно, программа потеряла незначительную часть универсальности — все предметы в комнате теперь реагируют одинаково. Вы автоматически открываете дверь, когда находите ключ. Игра становится чуть менее интересной.

Классы

Использование функций сильно сокращает объем кода, повышает его читаемость и масштабируемость (т.е. код становится более гибким). Например, вы можете много раз в разных частях программы выполнять одну и ту же функцию, а потом легко ее изменить, если потребуются доработки, и вам при этом не придется переделывать весь код, необходимо будет изменить только определение функции в одном месте.

У функций есть свои ограничения. Они не хранят никакой информации, как это делают переменные: каждый раз, когда функция запускается, она делает это заново. Однако некоторые функции и переменные очень тесно связаны друг с другом. Например, представьте, что у вас есть класс (объект типа) «ключка для гольфа». Он содержит информацию о нем (т.е. о переменных), например о длине вала, материале рукоятки и материале головки. У него также есть функции, связанные с ним, такие как функция раскачивания клюшки и др. Для этих функций вам необходимо знать все эти переменные клюшки.

Когда клюшка выполняет какую-либо свою функцию, она меняет свои переменные, например переменная «усталость клюшки» увеличивается при каждом вызове функции «воспользоваться клюшкой». Необходим способ сгруппировать тесно связанные функции и переменные в одном месте, чтобы они могли взаимодействовать друг с другом.

Если у вас есть несколько клюшек для гольфа, вы можете сделать базовый класс «клюшка», от которого будут использоваться стандартные вещи для клюшек, а более тонкие особенности будут реализовываться в отдельном классе, созданном вами для каждой конкретной клюшки. Таким образом создание программного гольф-клуба сильно упростится.

Для решения подобных задач применяется такой подход, как объектно-ориентированное программирование. Он объединяет функции и переменные таким образом, чтобы они могли видеть друг друга и работать вместе, тиражироваться и изменяться по мере необходимости. И для этого мы используем так называемые классы.

Создание класса

Что такое класс? Думайте о классе как о проекте. Это не что-то само по себе отдельное, это некоторый шаблон, описывающий набор функциональности. Например, вы можете создать множество объектов из какого-либо конкретного шаблона/прототипа, и это множество объектов или вовсе один объект называется в объектно-ориентированном программировании «экземпляры класса».

Классы создаются при помощи оператора `class`:

```
# Объявление класса
class class_name:
    [выражение 1]
    [выражение 2]
    [и т.д.]
```

Добавим конкретики. Посмотрим на пример класса `Shape` (с англ. — «фигура»):

```
---in↓---
#Пример класса
class Shape:
    def __init__(self, x, y):
        self.x = x #размеры по оси x и y
        self.y = y
    description = "Эта форма еще не описана"
```

```

author = "У этой формы еще нет автора"
def area(self): #функция вычисления площади
    return self.x * self.y
def perimeter(self): #функция вычисления периметра
    return 2 * self.x + 2 * self.y
def describe(self, text): #функция установки описания
    self.description = text
def authorName(self, text): #функция установки авторства
    self.author = text
def scaleSize(self, scale): #функция масштабирования
    self.x = self.x * scale
    self.y = self.y * scale
---↑in_out↓---
---↑out---

```

Вы создали описание формы (т.е. ее переменных) и то, какие операции вы можете делать с этой формой (т.е. функции). Это очень важно, вы создали не настоящую форму, а просто описание того, что такое форма. Форма имеет ширину *x*, высоту *y*, а также площадь и периметр `area(self)` и `perimeter(self)`. Когда вы определяете класс, код не запускается — вы просто создаете вложенные в него функции и переменные.

Функция под названием `__init__` запускается, когда мы создаем экземпляр класса `Shape`, т.е. когда мы создаем фактическую форму на основе шаблона/прототипа. Как это работает, вы поймете позже¹⁵.

Атрибут `self` (с англ. — «сам/себя») связывает элементы с текущим классом и другими элементами этого класса. `Self` — это первый параметр в любой функции, определенной внутри класса. Любая функция или переменная, созданная на первом уровне отступа (т.е. строки кода, начинающиеся на одну вкладку справа от того места, где мы помещаем класс `Shape`, автоматически помещаются в `self`). Чтобы получить доступ к этим функциям и перемен-

¹⁵ В рамках данной книги многие вещи рассматриваются поверхностно. Рекомендуется после полного ознакомления с данным занятием самостоятельно изучить связанные с объектно-ориентированным программированием понятия, такие как атрибуты, свойства, конструктор/деструктор и др. Например, подробную информацию можно изучить на русском языке в источнике [15].

ным в другом месте внутри класса, перед их именем нужно написать `self` и точку (например, `self.variable_name`). Без `self` вы можете использовать переменные только внутри функции, в которой они определены, а не в других функциях того же класса.

Использование классов

Ниже представлен пример того, что называется созданием экземпляра класса. Учтите, что приведенный выше код класса `Shape` должен был бы быть запущен.

```
---in↓---  
rectangle = Shape(100, 45)  
---↑in_out↓---  
---↑out---
```

В первую очередь действует функция инициализации `__init__`. Мы создаем экземпляр класса, сначала определяя его имя (в данном случае `Shape`), а затем в скобках значения, передаваемые в функцию `__init__`. Функция `__init__`¹⁶ запускается (используя параметры, которые вы указали в скобках), а затем выдает экземпляр этого класса, которому в данном случае присваивается имя «rectangle».

Экземпляр класса `rectangle` можно считать некоторой замкнутой коллекцией переменных и функций. Поскольку мы присвоили экземпляру класса имя `rectangle` (с англ. — «прямоугольник»), для доступа к функциям и переменным данного экземпляра класса извне мы теперь можем писать, например, `rectangle.perimeter()`.

Посмотрим, как работает показанный ранее код:

```
---in↓---  
#вычисление площади прямоугольника  
print(rectangle.area())
```

¹⁶ Кроме инициализатора «`__init__`», также можно указывать деструктор «`__del__`», код в котором будет выполнять необходимые действия, когда вы удаляете экземпляр класса при помощи «`del имя_экземпляра`».

```
#вычисление периметра прямоугольника
print(rectangle.perimeter())

#установка описания прямоугольника
rectangle.describe("Ширина прямоугольника более чем в два
раза больше его высоты")

#уменьшение прямоугольника на 50 %
rectangle.scaleSize(0.5)

#вывод новой площади прямоугольника
print(rectangle.area())
---↑in_out↓---
4500
290
1125.0
---↑out---
```

Благодаря использованию атрибута `self` мы делаем возможным просмотр и изменение переменных экземпляра класса, а также получаем доступ к функциям, которые создавали при определении прототипа класса.

Мы не ограничены одним экземпляром класса — у нас может быть столько экземпляров одного и того же прототипа класса, сколько захотим. Например:

```
---in↓---
longrectangle = Shape(120,10) #длинный прямоугольник
fatrectangle = Shape(130,120) #"жирный" прямоугольник
---↑in_out↓---
---↑out---
```

и как `longrectangle`, так и `fatrectangle` имеют свои собственные функции и переменные, содержащиеся внутри них, они полностью независимы друг от друга. Нет ограничений на количество экземпляров, которые мы можем создать.

Поэкспериментируйте с несколькими разными экземплярами класса `Shape`.

Базовая терминология объектно-ориентированного программирования

Рассмотрим основные **положения объектно-ориентированного программирования**:

- когда мы впервые описываем класс, мы определяем его (так же, как и с функциями);
- возможность группировать похожие функции и переменные вместе называется инкапсуляцией¹⁷;
- переменная внутри класса называется атрибутом класса;
- функция внутри класса называется методом класса;
- класс находится в той же категории вещей, что и переменные, списки, словари и т.д. То есть все они — объекты;
- класс известен как «структура данных», он содержит данные и методы их обработки.

Наследование

Мы знаем, как классы группируют вместе переменные и функции, известные как атрибуты и методы, так что и данные, и код для их обработки находятся в одном месте. Мы можем создать любое количество экземпляров этого класса, поэтому нам не нужно писать новый код для каждого нового объекта, который мы создаем. Но как насчет добавления дополнительных атрибутов и методов в классы? Здесь в игру вступает такое понятие, как «наследование».

Мы определяем новый класс на основе другого, родительского класса. Новый класс получает все методы и атрибуты от родителя, после чего мы также имеем возможность добавлять к унаследованному классу другие новые элементы или переопределять уже унаследованные: если какие-либо новые атрибуты или методы имеют то же имя, что и атрибут или метод в нашем родительском классе, они используются вместо родительского.

Рассмотрим вновь класс Shape:

```
class Shape:
    def __init__(self, x, y):
        self.x = x
```

¹⁷ **Инкапсуляция** (англ. *encapsulation*, от лат. *in capsula*) — в информатике размещение в одном компоненте данных и методов, которые с ними работают.

```

        self.y = y
    description = "Эта форма еще не описана"
    author = "У этой формы еще нет автора"
    def area(self):
        return self.x * self.y
    #---код обрезан---

```

Если бы мы хотели определить новый класс, скажем, квадрат, на основе нашего предыдущего класса Shape (форма), мы бы использовали следующий код (обратите внимание, что у квадрата стороны равны):

```

---in↓---
class Square(Shape):
    def __init__(self, x):
        self.x = x
        self.y = x
---↑in_out↓---
---↑out---

```

Это похоже на обычное определение класса, но на этот раз мы указали в скобках после имени родительский класс, от которого мы унаследовали все атрибуты, методы и т.п. Как видите, из-за этого мы очень быстро смогли описать квадрат. Это потому, что мы унаследовали все от класса shape (называемого базовым) и изменили только то, что нужно было изменить. В данном случае мы переопределили только функцию `__init__` в Shape так, чтобы значения X и Y были одинаковыми.

Создадим еще один новый класс, на этот раз унаследованный от Square. Это будут два квадрата, один сразу слева от другого:

```

---in↓---
# По форме это будет выглядеть примерно так:
#
# |   |   |
# |   |   |
# |___|___|

```

```
class DoubleSquare(Square): #класс "двойной квадрат"
    def __init__(self,y):
        self.x = 2 * y
        self.y = y
    def perimeter(self):
        return 2 * self.x + 3 * self.y
---↑in_out↓---
---↑out---
```

В данном случае нам также пришлось переопределить метод `perimeter()`, так как в нашей фигуре есть линия, идущая вниз, посередине фигуры.

Попробуйте создать экземпляр этого класса и поэкспериментируйте с разными значениями. Поскольку класс `Shape` уже был запущен, вы можете добавить только новые классы и определение экземпляров.

Ниже показано создание экземпляров, унаследованных от первичного базового класса `Shape`, классов `Square` и `DoubleSquare`, полученного двойным наследованием (т.е. наследуется не от `Shape`, а от `Square`, унаследованного от `Shape`):

```
---in↓---
testsquare = Square(5)
testdouble = DoubleSquare(6)
---↑in_out↓---
---↑out---
```

Указатели и словари классов

Когда вы указываете, что одна переменная равна другой, например `variable2 = variable1`, переменная слева от знака равенства принимает значение переменной справа от него. С экземплярами класса это происходит немного иначе: имя слева становится экземпляром класса справа. Таким образом, в `instance2 = instance1` (англ. *instance* — «зависимость»), `instance2` указывает на `instance1` — одному экземпляру класса дано два имени, и вы можете получить доступ к экземпляру класса через любое имя.

В других языках подобные действия выполняются с помощью указателей, однако в Python все это происходит «за кулисами».

Рассмотрим словари классов. Мы можем назначить экземпляр класса записи в списке или словаре. Это позволяет существовать практически любому количеству экземпляров класса при запуске нашей программы. Давайте посмотрим на пример ниже:

```
---in↓---
# Сначала создайте словарь:
dictionary = {}

# Затем создайте несколько экземпляров классов в словаре:
dictionary["DoubleSquare 1"] = DoubleSquare(5)
dictionary["long rectangle"] = Shape(600,45)

# Теперь вы можете использовать их как обычный класс:
print(dictionary["long rectangle"].area())

dictionary["DoubleSquare 1"].authorName("Есенин")
print(dictionary["DoubleSquare 1"].author)
---↑in_out↓---
27000
Есенин
---↑out---
```

Используя словари, работать с экземплярами классов удобнее, особенно если вам потребуется обрабатывать их в циклах.

Модули

Вы можете задаться вопросом: «Как мне использовать мои классы в различных программах, просто постоянно копировать их туда?» Ответ: нет, необходимо поместить классы в модуль, чтобы его можно было применять для импорта в другие программы.

Часто модули называются обобщенно — библиотеки. Модули также позволяют скрывать свое внутреннее устройство (исходный код) и предоставлять только то, что доступно через свой интерфейс. Например, вы можете откомпилировать свой модуль и распространять/продавать его не как файл с исходным кодом с расширением «.py», а как кроссплатформенный байтовый

код с расширением «.рус», который также может быть «запутан» (англ. *obfuscate*¹⁸, т.е. усиленно защищен).

Модуль — это файл, который (как правило) содержит только определения переменных, функций и классов. Например, так может выглядеть модуль, который может храниться в файле с именем `moduletest.py`:

```
---in↓---
### ПРИМЕР МОДУЛЯ PYTHON
# Определите некоторые переменные:
numberone = 1
ageofqueen = 78

# объявление некоторых функций
def printhello():
    print("привет")

def timesfour(input):
    print(eval(input) * 4)

# объявление класса
class Piano:
    def __init__(self):
        self.type = input("Какое пианино? ")
        self.height = input("Какая высота в метрах? ")
        self.price = input("Сколько оно стоит ")
        self.age = input("Сколько ему лет? ")

    def printdetails(self):
        print("Это пианино имеет высоту " + self.height +
" метров", end=" ")
        print(self.type, "ему, " + self.age, "лет и его
стоимость\
" + self.price + " долларов.")
---↑in_out↓---
---↑out---
```

¹⁸ **Обфускация** (от лат. *obfuscare* — затенять, затемнять; и англ. *obfuscate* — делать неочевидным, запутанным, сбивать с толку) или запутывание кода — приведение исходного текста или исполняемого кода программы к виду, сохраняющему ее функциональность, но затрудняющему анализ, понимание алгоритмов работы и модификацию при декомпиляции.

Как видите, модуль очень похож на обычную программу в Python.

Как использовать модуль (т.е. вынесенную в отдельный файл функциональность)? Для этого применяется оператор `import` (с англ. — «импорт», «подключение») его частей или модуля полностью в другие программы.

Чтобы импортировать все переменные, функции и классы из `moduletest.py` в другую программу, которую вы пишете, например в основную программу `mainprogram.py`, в основной программе будет следующее:

```
### mainprogram.py
import moduletest # импорт другого модуля
```

Это предполагает, что (одно из трех):

- модуль находится в том же каталоге, что и `mainprogram.py`;
- модуль является модулем по умолчанию, который поставляется с Python;
- модуль автоматически или вручную установлен при помощи менеджера пакетов в системный каталог, содержащий все модули.

Указывать расширение «.py» при импорте не нужно.

Обычно все операторы `import` помещают в начало файла исходного кода для удобства, но технически они могут быть где угодно.

Чтобы использовать элементы модуля в основной программе, используется следующий синтаксис: пишется название модуля, потом точка, а далее имя элемента из этого модуля. Например, ниже показано создание экземпляра пианино `cfcpiano` и вызов метода `printdetails()`:

```
### ИСПОЛЬЗОВАНИЕ ИМПОРТИРОВАННОГО МОДУЛЯ
# Используйте код вида modulename.itemname
print(moduletest.ageofqueen)
cfcpiano = moduletest.Piano()
cfcpiano.printdetails()
```

Как видите, модули, которые вы импортируете, очень похожи на классы: все, что находится внутри них, должно начинаться с имени этого модуля, чтобы оно работало.

Существуют и другие варианты подключения модуля, при которых у модуля меняется имя (устанавливается его псевдоним) или производится «бесшовный» импорт функциональности, когда мы можем использовать внутренности модуля без указания его имени или псевдонима. Рассмотрим эти варианты далее.

Пример еще одного модуля

Чтобы каждый раз при использовании элементов модуля не указывать его имя, можно применять оператор `from`. Синтаксис его использования следующий:

```
from modulename import itemname
```

Ниже представлен пример импорта переменных `ageofqueen` и `printhello` из ранее рассмотренного модуля `moduletest`. Мы можем применять эти переменные без указания имени/псевдонима модуля, из которого мы их использовали:

```
###импорт элементов прямо в вашу программу
```

```
#их импорт
from moduletest import ageofqueen
from moduletest import printhello
```

```
#теперь используем их
print(ageofqueen)
printhello()
```

Это уберет множество вложенностей в вашем коде и сделает его более читабельным.

Если необходимо импортировать сразу все компоненты модуля, вы можете использовать знак звездочки «`*`» вместо указания имени объекта:

```
from modulename import *
```

Полный импорт всех объектов может вызвать проблемы, если в вашей программе есть объекты с такими же именами, как у эле-

ментов в модуле. С большими модулями это может легко произойти и вызовет много проблем.

Лучший способ избежать этого — импортировать модуль обычным способом (без оператора `from`), а затем назначить элементам локальные имена:

```
# Присвоение элементам модуля локального имени
timesfour = moduletest.timesfour

# Использование локального имени
print(timesfour(565))
```

Последний удобный способ импорта модулей — использование псевдонима. Для этого вы можете использовать оператор `as` (с англ. — «как»). Это выглядит так:

```
import moduletest as mt # импорт модуля с псевдонимом

# использование модуля
print(mt.age)
cfcpiano = mt.Piano()
cfcpiano.printdetails()
```

ФАЙЛОВЫЙ ВВОД-ВЫВОД

Открытие файла

Чтобы открыть текстовый файл, можно использовать функцию `open()`. Вы передаете функции `open()` определенные параметры, чтобы указать, каким образом следует открывать файл. Полный список вариантов открытия файла вы можете увидеть в таблице ниже (табл. 7).

Давайте откроем файл «`readme.txt`» для чтения и распечатаем его содержимое:

```
---in↓---
filename = 'readme.txt' # путь к файлу
fl = open(filename, 'r') # имя файла
```



```
for line in fl:
    print(line, end='')
fl.close() # Закрыть файл
---↑in_out↓---
Line 1

Line 3

Line 4

Line 6
---↑out---
```

Таблица 7

Режимы открытия файла при помощи open()

Режим	Обозначение
'r'	открытие на чтение (является значением по умолчанию)
'w'	открытие на запись, содержимое файла удаляется; если файла не существует, создается новый
'x'	открытие на запись, если файла не существует, иначе исключение
'a'	открытие на дозапись, информация добавляется в конец файла
'b'	открытие в двоичном режиме
't'	открытие в текстовом режиме (является значением по умолчанию)
'+'	открытие на чтение и запись

Лучше написать код следующим образом:

```
---in↓---
filename = 'readme.txt'
with open(filename, 'r') as fl:
    for line in fl:
        print(line, end='')
---↑in_out↓---
```

```
# вывод аналогичен предыдущему
---↑out---
```

При втором способе вам не нужно добавлять закрытие файла при помощи «`fl.close`», он автоматически закрывается.

Конструкция «`with ... as`» используется для оборачивания выполнения блока инструкций менеджером контекста. Иногда это более удобная конструкция, чем «`try...except...finally`», для обработки ошибок (будет рассмотрено на данном занятии позже).

Основная причина использования менеджера контекста во время открытия файла заключается в том, что этот файл будет открыт в любом случае, независимо от того, все ли в порядке или возникает ли какое-либо исключение, и гарантированно будет закрыт, а место в памяти, которое он занимал при открытии файла, будет очищено.

Курсор при вводе-выводе

Если вы хотите распечатать весь файл сразу вместо того, чтобы перебирать строки, вы можете использовать следующее:

```
---in↓---
filename = './readme.txt'
fl = open(filename, 'r')
print(fl.read())
---↑in_out↓---
Line 1

Line 3
Line 4

Line 6
---↑out---
```

Если вы попытаетесь выполнить «`print (fl.read ())`» второй раз, то произойдет ошибка, заключающаяся в том, что курсор сменил свое место.

Невидимый курсор сообщает функции чтения (и многим другим функциям ввода-вывода), с чего начать¹⁹. Чтобы установить, где находится курсор, можно применить функцию `seek()`. Она используется следующим образом:

```
seek(offset, whence)
```

Параметр «`whence`» (с англ. — «откуда») не является обязательным параметром и определяет, откуда искать. Если `whence` равен 0, байты/буквы считаются сначала. Если он равен 1, байты отсчитываются от текущей позиции курсора. Если 2, то байты отсчитываются с конца файла. Если туда ничего не помещено, предполагается, что 0, т.е. чтение будет производиться с начала файла.

Параметр «`offset`» (с англ. — «смещение») определяет, как далеко от «откуда» перемещается курсор. Например:

- вызов «`fl.seek(45, 0)`» переместит курсор на 45 байт/букв после начала файла;

- вызов «`fl.seek(10, 1)`» переместит курсор на 10 байт/букв после текущей позиции курсора;

- вызов «`fl.seek(-77, 2)`» переместит курсор на 77 байт/букв перед концом файла (обратите внимание на знак минус перед числом 77).

Мы можем использовать `fl.seek()`, чтобы перейти к любому месту в файле, а затем попробовать ввести `print(fl.read())`. Он будет печататься с того места, откуда вы искали. Но помните, что `fl.read()` перемещает курсор в конец файла, вам придется выполнить поиск снова.

Другие функции ввода-вывода

Есть много других функций, которые помогают работать с файлами. Посмотрим на некоторые наиболее интересные из них: `tell()`,

¹⁹ Также стоит отметить, что курсор часто используется при работе с базами данных, это будет рассмотрено далее в рамках данной книги. В базах данных при помощи курсора можно получать, например, позицию последнего запроса, т.е., к примеру, какой уникальный номер получил последний вставленный в таблицу базы данных элемент.

`readline()`, `readlines()`, `write()` и `close()`. Ниже приведено их описание:

- `tell()` возвращает курсор в файле. У него нет параметров, просто введите его (как в примере ниже). Это полезно для понимания того, где находится курсор. Чтобы использовать эту функцию, введите «`fileobjectname.tell()`», где `fileobjectname` — это имя файлового объекта, созданного при открытии файла (в «`openfile = open(<pathToFile>, <r>)`» имя файлового объекта `openfile`);

- `readline()` читает с того места, где находится курсор, до конца строки. Помните, что конец строки — это не край экрана, строка заканчивается, когда вы нажимаете клавишу Enter, чтобы создать новую строку. Эта функция полезна, например, при чтении журнала событий. Единственный параметр, который вы можете передать в «`readline()`» при необходимости, поместив соответствующее число в скобки, — это максимальное количество байтов/букв для чтения;

- `readlines()` считывает все строки, начиная с курсора и до конца, и возвращает список, в котором каждый элемент содержит строку кода. Используйте его с `fileobjectname.readlines()`. Работа `readlines()` демонстрируется при помощи кода ниже:

```
---in↓---
filename = './readme.txt'
fl = open(filename, 'r')
print (fl.readlines ())
fl.close() #Закрыть файл
---↑in_out↓---
['Line 1\n', '\n', 'Line 3\n', 'Line 4\n', '\n', 'Line
6']
---↑out---
```

или этот же список, представленный в табл. 8;

- `write()` — это функция записи в файл. Она пишет с того места, где находится курсор, и перезаписывает текст перед ним, как в MS Word, где вы нажимаете «вставить», и он записывает поверх старого текста. Синтаксис следующий: `fileobjectname.write('Строка, которую вы хотите записать');`

- `close()` закрывает файл.

Таблица 8

Содержимое файла `readme.txt`

Индекс	Значение
0	'Line 1'
1	
2	'Line 3'
3	'Line 4'
4	
5	'Line 6'

Попробуйте поэкспериментировать с созданием и чтением файла через Python. Вы можете создать файл в стороннем редакторе (в том числе Jupyter имеет возможность создавать и редактировать текстовые файлы) или сразу начать работать с функцией записи.

```
---in↓---
#просто создадим файл и запишем в него пару строк
filename = './readme.txt'
fl = open(filename, 'w')
fl.write('Привет\n')
fl.write('123')
fl.close() #Закреть файл

#прочитаем содержимое файла
with open(filename, 'r') as fl:
    for line in fl:
        print(line)
---↑in_out↓---
Привет

123
---↑out---
```

Объекты, сохраняемые в файл

Вы можете сохранять объекты напрямую в файл. Объект в этом случае может быть переменной, экземпляром класса или списком, словарем или кортежем. Можно выполнять `dump` (с англ. — «сброс», т.е. сброс данных) и другие вещи. Другими словами, вы «сохраняете» свои объекты.

Для сохранения объектов вы можете воспользоваться методом `dump()`, который находится внутри модуля `pickle` (с англ. — «мариновать»), поэтому в начале программы вам нужно будет написать «`import pickle`» и убедиться, что данный модуль (пакет) установлен.

После импорта откроем пустой файл и используем «`pickle.dump()`», чтобы поместить в него объект:

```
---in↓---
### pickletest.py
import pickle

# давайте создадим список для "маринования"
picklelist = ['один', 2, 'три', 'четыре', 5, 'Можете
сосчитать?']

# теперь создаем файл
# вы можете заменить имя файла на любое
# wb означает, что файл записан в двоичном виде (т.е. не
.txt и не .docx)
file = open('filename', 'wb')

pickle.dump(picklelist, file) #"замаринуем рассол"

#закроем файл
file.close()
---↑in_out↓---
---↑out---
```

где:

- `picklelist` — это объект, который вы хотите сохранить в файл;
- `file` — это файловый объект, в который вы хотите осуществить запись (в данном случае файловым объектом является `file`).

Обновите список файлов и убедитесь, что файл с именем `filename` был создан.

Теперь, чтобы повторно открыть или разобрать ваш файл, мы будем использовать `pickle.load()`:

```
---in↓---
import pickle

unpicklefile = open('filename', 'rb') # открываем файл для
чтения
unpickledlist = pickle.load(unpicklefile) # загружаем
список из файла
unpicklefile.close() #закроем файл

# Попробуем напечатать список и проверить, что все в порядке
for item in unpickledlist:
    print(item)
---↑in_out↓---
один
2
три
четыре
5
Можете сосчитать?
---↑out---
```

Есть важное ограничение: таким образом мы можем поместить в файл только один объект. Можно обойти его, поместив множество выбираемых объектов в список или словарь, а затем при чтении обработав этот список или словарь. Это самый быстрый и простой способ, более сложные способы вы можете изучить самостоятельно.

Обработка исключений (ошибок)

При помощи программы ниже продемонстрируем возникновение исключения в программе:

```
---in↓(raises-exception)---
def menu(list, question):
    for entry in list:
```

```

        print(1 + list.index(entry),end="")
        print(" " + entry)

    return input(question) - 1

# вызов функции меню
answer = menu(['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I'], \
'Какая буква ваша любимая ')

print('Вы выбрали ответ ' + str(answer + 1))
---↑in_out↓---
1) A
2) B
3) C
4) D
5) E
6) F
7) H
8) I
Какая буква ваша любимая  1
-----
-----
TypeError                                Traceback (most
recent call last)
<ipython-input-36-82d612b55785> in <module>
      7
      8 # вызов функции меню
----> 9 answer = menu(['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I'], \
    10 'Какая буква ваша любимая ')
    11
<ipython-input-36-82d612b55785> in menu(list, question)
      4         print(" " + entry)
      5
----> 6         return input(question) - 1
      7
      8 # вызов функции меню
TypeError: unsupported operand type(s) for -: 'str' and 'int'
---↑out---
```

Наиболее частые проблемы с кодом возникают по собственной вине программиста. Печально, но факт.

Что мы видим в выводе примера, продемонстрированного выше? Python пытается сказать, что вы не можете объединить список букв и число в один список (строку текста). Давайте рассмотрим сообщение об ошибке и определим его содержимое:

- «- - ->» показывает строки, в которых обнаружена ошибка. Сначала он указывает на строку 10 (строка), а затем на строку 6 (вычисление, при котором мы вычитаем целое число из строки). Обратите внимание, что строка 6 была в функции;

- «TypeError: неподдерживаемые типы операндов для - : 'str' и 'int' » — ошибка типов, когда вы попытались вычесть друг из друга несовместимые переменные.

Существует несколько списков файлов (в данном случае функций) и кодов для одной ошибки, потому что ошибка возникла при взаимодействии двух строк кода (например, при использовании функции ошибка возникла в строке, где функция была вызвана, и в строке функции, где все пошло не так).

Теперь мы знаем, в чем проблема и как ее исправить. В сообщении об ошибке указано, где находится проблема, поэтому мы сосредоточимся только на этом фрагменте кода:

```
answer = menu (['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I'], \
'Какая буква ваша любимая? ')
```

Это вызов функции. Ошибка произошла в функции в следующей строке:

```
return input(question) - 1
```

Функция `input()` всегда возвращает строку, отсюда и наша проблема. Давайте заменим `input()` на `eval(input())`.

Функция `eval()`, как было рассмотрено на предыдущем занятии, позволяет программе запускать код Python, который мы ей передаем, внутри себя.

Функция «меню» по завершении своего выполнения возвращает выбранное пользователем число с вычитанием из него единицы, чтобы индексы начинались с нуля (т.е. индексация элементов в списках всегда начинается с нуля). Заменив возвращаемое значение на следующий код, мы исправим ошибку:

```
return eval(input(question)) - 1
```

Теперь данный код полностью работоспособен:

```
---in↓---
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print(" " + entry)
    return eval(input(question)) - 1

# вызов функции меню
answer = menu(['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I'], \
'Какая буква ваша любимая ')

print('Вы выбрали ответ ' + str(answer + 1))
---↑in_out↓---
1) A
2) B
3) C
4) D
5) E
6) F
7) H
8) I
Какая буква ваша любимая 1
Вы выбрали ответ 1
---↑out---
```

Все работает правильно, если вводить число. В ситуации, когда пользователь случайно введет букву вместо числа, снова возникнет ошибка.

Если вы не знаете, как предотвратить возникновение ошибки, то один из лучших и простых способов — использовать операторы `try` и `except` для обработки исключений.

Пример использования `try` в программе:

```
try:
    function(world, parameters)
```

```
except:
    print(world.errormsg)
```

Сначала запускается код из тела `try`. Если возникает ошибка, интерпретатор переходит в раздел `except` и печатает код ошибки `world.errormsg`. Программа не останавливается и не дает сбой, т.е. при возникновении ошибки она просто запускает код, содержащийся в теле `except` (тело `except` называется обработчиком ошибки), а затем продолжает работу.

Давайте попробуем обработать ошибки ввода букв вместо числа. Воспользуйтесь кодом ниже и попробуйте ввести букву, когда программа попросит ввести число, и посмотрите, что произойдет. Мы исправили одну проблему, но теперь она вызвала другую:

```
---in↓(raises-exception)---
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print(" " + entry)
    try:
        return eval(input(question)) - 1
    except NameError:
        print("Введите правильное число")

# running the function
# remember what the backslash does
answer = menu(['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I'], \
    'Какая буква ваша любимая ')

print('Вы выбрали ответ ' + str(answer + 1))
---↑in_out↓---
```

- 1) A
- 2) B
- 3) C
- 4) D
- 5) E
- 6) F
- 7) H
- 8) I

Какая буква ваша любимая Ф

Введите правильное число

TypeError Traceback (most recent call last)

<ipython-input-38-9f17d20db70c> in <module>

13 'Какая буква ваша любимая ')

14

---> 15 print('Вы выбрали ответ ' + str(answer + 1))

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

---↑out---

На этот раз произошло то, что функция меню не вернула никакого значения, она только напечатала сообщение об ошибке. Когда в конце программы мы пытаемся напечатать возвращаемое значение, увеличенное на единицу, какое значение будет возвращено? Нет возвращаемого значения. Что такое «1+...»? Это некорректное выражение.

Мы бы могли просто вернуть любой старый номер в программе, но это было бы неправильно. Корректно исправить программу и предотвратить возникновение этого исключения можно следующим образом:

---in↓---

def menu(list, question):

for entry in list:

print(1 + list.index(entry), end="")

print(" " + entry)

try:

return eval(input(question)) - 1

except NameError:

print(«Введите правильное число»)

answer = menu(['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I'], \n 'Какая буква ваша любимая? ')

try:

print(«Вы выбрали ответ », (answer + 1))


```
# вы можете ставить что-либо после запятой в операторе
'print',
# и это будет продолжаться, как если бы вы снова набрали
'print'
except:
    print(«\nНеправильный ответ»)
    # '\ n' используется для форматирования, а именно – это
    символ переноса строки. Попробуйте без него
    ---↑in_out↓---
```

1) A
2) B
3) C
4) D
5) E
6) F
7) H
8) I

Какая буква ваша любимая? Ф
Введите правильное число
Неправильный ответ
---↑out---

Подход, который мы использовали выше, не рекомендуется, потому что кроме ошибки, которая, как мы знаем, может произойти, `except` также перехватывает и все остальные. Это означает, что мы никогда не увидим ошибку, которая могла бы вызвать проблемы в будущем?

Если `except` отлавливает каждую скрытую ошибку, у нас нет никакого способа контролировать, с какими конкретно ошибками мы имеем дело. Мы можем не увидеть любые другие ошибки, с которыми мы еще не столкнулись в данной книге. У нас также мало шансов на устранение более чем одного типа ошибок в одном и том же блоке кода.

Что делать, когда все безнадежно? Вот пример кода с такой ситуацией:

```
---in↓(raises-exception)---
print("Программа вычитания, v0.0.1 (бета)")
a = eval(input('Введите 1-е число для вычитания > '))
```

```

b = eval(input('Введите 2-е число > '))
print(a - b)
---↑in_out↓---
Программа вычитания, v0.0.1 (бета)
Введите 1-е число для вычитания >  М
-----
NameError                                Traceback (most
recent call last)
<ipython-input-41-95bfdd9a742a> in <module>
      1 print("Программа вычитания, v0.0.1 (бета)")
----> 2 a = eval(input('Введите 1-е число для вычитания > '))
      3 b = eval(input('Введите 2-е число > '))
      4 print(a - b)
<string> in <module>
NameError: name 'М' is not defined
---↑out---
```

Если ввести два числа, все работает корректно. Ввод буквы приведет к возникновению ошибки `NameError` (с англ. — «ошибка имени»).

Давайте перепишем код, чтобы иметь дело только с ошибкой `NameError`. Мы поместим программу в цикл, чтобы она перезапускалась в случае возникновения ошибки (с помощью оператора `continue` (с англ. — «продолжить»), который снова запускает цикл сверху, и оператора `break`, который приводит к выходу из цикла):

```

---in↓(raises-exception)---
print("Программа вычитания, v0.0.2 (бета)")
loop = 1
while loop == 1:
    try:
        a = eval(input('Введите 1-е число для вычитания > '))
        b = eval(input('Введите 2-е число > '))
    except NameError:
        print("\nВы не можете вычитать буквы")
        continue
    print(a - b)
    try:
        loop = eval(input('Нажмите 1, чтобы попробовать
снова> '))
```

```
except NameError:
    loop = 0
---↑in_out↓---
Программа вычитания, v0.0.2 (бета)
Введите 1-е число для вычитания > !
Traceback (most recent call last):

  File "/home/pmelikov/.local/lib/python3.8/site-
packages/IPython/core/interactiveshell.py", line 3441, in
run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-42-207f142451a1>", line 12, in
<module>
    loop = eval(input('Нажмите 1, чтобы попробовать
снова> '))

  File "<string>", line 1
    !
    ^
SyntaxError: unexpected EOF while parsing
---↑out---
```

В представленном выше коде происходит перезапуск цикла, если пользователь ввел что-то не так. В строке 12 находится предположение, что пользователь хочет выйти из программы, если он не нажал 1 (1 приводит к продолжению цикла), поэтому выходим из программы.

Но проблемы не исчезли. Если пользователь оставит один из запрашиваемых вводов значений пустым или введет специальный символ, например «!» или «;», программа выдаст нам «SyntaxError» (с англ. — «синтаксическая ошибка»).

Решим данную проблему. Когда пользователь вводит специальные символы вместо числа (например, восклицательный знак), программа будет выдавать другое сообщение об ошибке:

```
---in↓---
print("Subtraction program, v0.0.2 (beta)")
loop = 1
```

```
while loop == 1:
    try:
        a = eval(input('Введите 1-е число для вычитания >
    '))
        b = eval(input('Введите 2-е число > '))
    except NameError:
        print("\nВы не можете вычитать буквы")
        continue
    except SyntaxError:
        print("\nПожалуйста, введите число")
        continue
    print(a - b)
    try:
        loop = eval(input('Нажмите 1, чтобы попробовать
снова> '))
    except (NameError, SyntaxError):
        loop = 0
---↑in_out↓---
```

Subtraction program, v0.0.2 (beta)

Введите 1-е число для вычитания > ф

Вы не можете вычитать буквы

Введите 1-е число для вычитания > !

Пожалуйста, введите число

Введите 1-е число для вычитания > 1

Введите 2-е число > 2

-1

Нажмите 1, чтобы попробовать снова> 0

---↑out---

Тем самым мы сделали достаточно работоспособную программу для вычитания двух чисел. Стоит отметить, что было бы гораздо более правильно обрабатывать вводимые значения (проверять, что введено число), чем просто отлавливать возникающие ошибки (отлавливание ошибок — это борьба с последствиями их возникновения, а не с причиной).

В завершение занятия удалим все созданные файлы, чтобы не засорять память компьютера:


```
---in↓---
import os

files = ['filename',]
for f in files:
    if os.path.isfile(f): # если файл существует
        os.remove(f)
---↑in_out↓---
---↑out---
```

Задания для проверки

1. Опишите отличия массивов, кортежей, списков и словарей.
2. Приведите пример кода, реализующего проверку наличия в словаре записи с каким-то определенным ключом. Например, наличия контакта в телефонной книге.
3. При помощи цикла `for` выведите таблицу умножения для числа 5. То есть число 5 должно умножаться на каждое из `[0; 9]` чисел, и результат выводится пользователю.
4. Приведите пример кода, который записывает/создает текстовый файл, записывает в него две строчки «Hello» и «123», а затем считывает его и выводит его содержимое. Код прокомментируйте.
5. Что такое исключения (ошибки) и как их можно обработать? Что лучше выбрать: написать программу так, чтобы не возникало ошибок или чтобы ошибки были обработанными?
6. Если происходит ошибка, интерпретатор выводит информацию об этой ошибке. Приведите пример такого вывода интерпретатора и опишите значение выводимой информации.

СТАНДАРТНЫЕ БИБЛИОТЕКИ²⁰

Цели выполнения работы

Получение представления о функциональности стандартных модулей/библиотек, поставляемых совместно с интерпретатором Python. Изучение основных принципов практической работы с ними.

²⁰ Разработано на основе официальной документации Python (источник [16]).

Порядок выполнения работы

Интерфейс операционной системы

Модуль «os» предоставляет большое количество функций для взаимодействия с операционной системой, т.е. позволяет осуществлять вызов системных команд, например, таких как получение полного пути к текущему каталогу, из которого выполняется программа, смена каталога, его удаление и многое другое. Ниже приведено три примера:

```

---in↓---
import os

print (os.getcwd())      # Вернуть (получить) текущий
рабочий каталог
os.system('mkdir today') # Запускаем команду mkdir
в системной оболочке
os.chdir('./today')      # Изменить текущий рабочий каталог
(перейти в созданную папку)\
print (os.getcwd())

os.chdir('../')
os.rmdir('today') #удалить созданную папку
print (os.getcwd())
---↑in_out↓---
/home/pmelikov/Python для анализа данных. Семинар 3.
/home/pmelikov/Python для анализа данных. Семинар 3./today
/home/pmelikov/Python для анализа данных. Семинар 3.
---↑out---
```

Обязательно используйте «import os» вместо «from os import *». Это не позволит функции `os.open()` заменить встроенную функцию `open()`, которая работает по-другому (мы с ней ознакомились на предыдущем занятии, стр. 111).

Вы можете воспользоваться функциями `dir()` и `help()`, передав «os» в качестве параметра, чтобы получить список доступных методов этой библиотеки и справку по ним. Например:

```
---in↓---
import os

print('---первые 5 строк из вывода dir:')
display (dir(os)[:5])

print('---help для os.cpu_count:')
help(os.cpu_count)
print(os.cpu_count())
---↑in_out↓---
---первые 5 строк из вывода dir:
['DirEntry', 'F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY']
---help для os.cpu_count:
Help on built-in function cpu_count in module nt:

cpu_count()
    Return the number of CPUs in the system; return None
    if indeterminable.

    This number is not equivalent to the number of CPUs
    the current process can
    use. The number of usable CPUs can be obtained with
    ``len(os.sched_getaffinity(0))``

12
---↑out---
```

Для повседневных задач управления файлами и каталогами модуль `shutil` предоставляет более простой в использовании интерфейс более высокого уровня. Например, ниже показано копирование и перемещение файлов:

```
---in↓---
import shutil

shutil.copyfile('ЗАНЯТИЕ 4.ipynb', 'copy_of_lesson.
ipynb') # копирование файла
os.mkdir('my_dir') # создание папки my_dir, если ее
не существует
```

```
shutil.move('copy_of_lesson.ipynb', 'my_dir');
# перемещение файла copy_of_lesson.ipynb в папку my_dir
---↑in_out↓---
---↑out---
```

Для использования переменных окружения можно применять:

```
os.path.expanduser() и os.path.expandvars():
---in↓---
print (
os.path.expandvars('%USERNAME%'),
os.path.expanduser('~'),
sep='\n'
)
---↑in_out↓---
pavel
C:\Users\pavel
---↑out---
```

Переменные окружения — это переменные, которые используются в командной оболочке вашей операционной системы, например в них может содержаться имя операционной системы либо абсолютно любая переменная, которую вы можете самостоятельно установить в системной оболочке заранее (под системной оболочкой имеется в виду терминал/консоль).

Например «~/Downloads» является каталогом с загрузками текущего активного пользователя в Linux (т.е. того, от имени которого вы выполняете свой код). Знак тильды «~» — это переменная окружения, означающая домашний каталог пользователя. То есть при помощи тильды вы можете указывать пути относительно домашнего каталога.

В Windows обратиться к аналогичному каталогу можно, например, так: «C:\\Users\\%USERNAME%\\Downloads», где «%USERNAME%» — имя пользователя (ваша программа сама его подставит, если вы скажете ей сначала взять его из системных переменных).

Пример выполнен в Windows. Из-за особенностей «os.path.expanduser('~)», вне зависимости от операционной системы,

знак тильды означает домашний каталог пользователя. Обычно системная переменная окружения «~» в Windows не используется, вместо него принято применять «%USERNAME%».

Поиск файлов на основе подстановочных данных (Wildcards)

Модуль `glob` находит все пути, совпадающие с заданным шаблоном в соответствии с правилами, используемыми системной оболочкой операционной системы.

Обрабатываются символы:

- «*» — произвольное количество символов;
- «?» — один символ и диапазоны символов с помощью «[]».

Для поиска специальных символов заключайте их в квадратные скобки. Например, «[?]» соответствует символу «?»;

- `glob.glob(pathname)` — возвращение списка путей, соответствующих шаблону `pathname`. Список может оказаться пустым, если нет соответствий. Путь может быть абсолютным (например, «/usr/src/Python-1.5/Makefile») или относительным (например, «../../Tools//.gif» или «~/Downloads»);

- `glob.escape(pathname)` экранирует все специальные символы для `glob` (например, «?», «*» и «[»). Специальные символы в имени диска не экранируются (так как они там не учитываются), т.е. в Windows «`escape(«//?/c:/Quo vadis?.txt»)`» возвращает «`//?/c:/Quo vadis[?].txt`».

Указанный ниже код выведет имена всех файлов в текущей папке, содержащие «*.ipynb» (т.е. отфильтруются все файлы с расширением блокнотов Jupyter). Звездочка означает подстановку любого количества разных знаков.

```
---in↓---
import glob

print (glob.glob('*.ipynb'))
#или так, если нам нужно именно занятие от 0 до 9.
# ? соответствует одному любому символу, в данном
случае — точке
glob.glob('*[0-9]?ipynb')
```

```
---↑in_out↓---  
['ЗАНЯТИЕ 4.ipynb']  
['ЗАНЯТИЕ 4.ipynb']  
---↑out---
```

Аргументы командной строки

Когда при запуске кода вам нужно передать ему какие-либо параметры, то возникает необходимость в обработке аргументов командной строки. Эти аргументы хранятся в атрибуте `argv` модуля `sys` в виде списка.

Рассмотрим пример. Если бы наш код хранился в каком-то файле, то мы бы написали, например, «`python demo.py one two three`». Тогда бы запустилась программа `demo.py` и ей передались три параметра.

То есть имя программы передается первым аргументом интерпретатору (интерпретатор Python — это тоже программа, и ей можно передавать параметры, наподобие того, как мы делали с функциями), а далее мы можем добавлять сколько угодно параметров, в данном случае это три текстовых параметра «one», «two» и «three», которые интерпретатор передаст в нашу программу, и она может их как-либо дальше использовать.

Вы могли, например, использовать одну и ту же программу, но в разных целях. Например, если передан параметр «one» — делать одно, а если другой — то другое действие или все сразу. Или просто передавать какое-либо значение, например путь к файлу, который программа обрабатывает по заложенному в нее алгоритму.

Мы используем Jupyter и он позволяет выполнять не только Python-код, но и системные команды, если перед командой написать восклицательный знак. Запустим Python-код через вызов системной команды запуска интерпретатора, о чем говорит восклицательный знак. То есть здесь (при использовании «`!python`») не интерактивный интерпретатор `ipython` среды Jupyter выполнит Python-код, а системный интерпретатор. При этом код передается как аргумент (т.е. даже не обязательно, чтобы ваша программа хранилась в файле).

```
---in↓---
!python -c "import sys; print(sys.argv)" one two three
---↑in_out↓---
['-c', 'one', 'two', 'three']
---↑out---
```

Эта системная команда выполнила Python-код, который вывел полный список полученных аргументов командной строки.

Модуль `argparse` предоставляет более сложный механизм для обработки аргументов командной строки. Следующий код извлекает одно или несколько имен файлов и необязательное количество отображаемых строк:

```
---in↓---
command="""import argparse
parser = argparse.ArgumentParser(prog = "top_lines",
description = "Показать верхние строки из каждого файла")
#указание имени программы и ее описания
parser.add_argument("filenames", nargs="+") #добавление
аргументов к программе, чтобы она их обрабатывала, когда
вызывается вся программа интерпретатором
parser.add_argument("-l", "--lines", type=int,
default=10) #добавление аргумента lines
args = parser.parse_args() #запись прочитанных аргументов
в массив
print('----->Аргументы:\n', args)

for file_name in args.filenames:
    print ('----->',file_name)
    fl = open(file_name, 'r', encoding="utf-8")
    print (*fl.readlines()[0:args.lines], end="")
    fl.close() #Закрыть файл
"""

with open("top.py", "w", encoding="utf-8") as text_file:
    text_file.write(command)

#запуск программы через системный интерпретатор,
а не блокнотовский
!python "top.py" "ЗАНЯТИЕ 4.ipynb" "top.py" -l=3
```

```

print ("----->HELP:")
!python "top.py" -h
---↑in_out↓---
----->Аргументы:
  Namespace(filenamees=['ЗАНЯТИЕ 4.ipynb', 'top.py'],
lines=3)
-----> ЗАНЯТИЕ 4.ipynb
{
  "cells": [
    {
-----> top.py
import argparse
  parser = argparse.ArgumentParser(prog = "top_lines",
description = "Показать верхние строки из каждого файла")
#указание имени программы и ее описания
  parser.add_argument("filenamees", nargs="+") #добавление
аргументов к программе, чтобы она их обрабатывала, когда
вызывается вся программа интерпретатором
----->HELP:
usage: top_lines [-h] [-l LINES] filenamees [filenamees ...]

Показать верхние строки из каждого файла

positional arguments:
  filenamees

optional arguments:
  -h, --help            show this help message and exit
  -l LINES, --lines LINES
---↑out---
```

Ниже представлены три основных аргумента для функции `argparse.ArgumentParser()`:

- «prog» — название программы (по умолчанию «sys.argv [0]»);
- «description» — текст, отображаемый перед справкой по аргументу (по умолчанию равен None);
- «add_help» — добавить «-h/- -help» параметр парсера, отвечающий за вывод справки по программе (по умолчанию равен True).

При помощи кода выше был создан файл программы `top.py`. Вы можете запускать данную программу через системную обо-

лочку (командную строку или терминал), например командой «python top.py --lines = 5 alpha.txt beta.txt» программа top.py принимает args.lines, равным 5, а args filenames — равным ['alpha.txt', 'beta.txt']. Аналогичное продемонстрировано в самом примере.

Соответствие строковому шаблону (регулярные выражения)

Модуль re предоставляет инструменты регулярных выражений для расширенной обработки строк. Для сложных сопоставлений и манипуляций регулярные выражения предлагают краткие, оптимизированные решения:

```
---in↓---
import re
print (re.findall(r'\bf[a-z]*', 'which foot or hand fell
fastest')) #найдет любые слова, начинающиеся на f[любая_
буква]любой_символ
---↑in_out↓---
['foot', 'fell', 'fastest']
---↑out---
```

где:

- \b — обозначение границы слова;
- [a-z] — любая буква;
- * — любой символ в произвольном количестве.

Когда вам нужно найти символ в строке, в большей части случаев вы можете использовать непосредственно этот символ или строку. Например, когда нам нужно проверить наличие слова «dog», то мы будем применять набор букв «dog» для поиска.

Конечно, существуют определенные символы, которые заняты регулярными выражениями. Они также известны как метасимволы. Полный список метасимволов, которые поддерживают регулярные выражения Python, представлен ниже:

. ^ \$ * + ? { } [] | ()

Давайте взглянем, как они работают. Основная связка метасимволов, с которой вы будете сталкиваться, — это квадратные скобки: «[» и «]». Они используются для создания набора символов, которые вы можете сопоставить. Вы можете отсортировать символы самостоятельно, например, так: «[xyz]», если вам нужно в строке найти слово «xyz». Вы также можете применить тире для выражения ряда символов: [a-g] (это означает, что будет осуществлен поиск по всем символам от a до g).

Фактически для выполнения поиска нам нужно добавить начальный искомый символ и конечный. Чтобы упростить это, мы можем использовать звездочку. Вместо сопоставления символ звездочки указывает регулярному выражению, что предыдущий символ может быть сопоставлен сколько угодно раз (включая ни разу, т.е. ноль раз). Давайте посмотрим на пример, чтобы лучше понять, о чем речь:

```
a[b-f]*f
```

Этот шаблон регулярного выражения показывает, что мы ищем букву a, ноль или несколько букв из нашего класса, [b-f], и поиск должен закончиться буквой f. Давайте используем это выражение в Python:

```
---in↓---
import re
text = 'abcdfghijk'

parser = re.search('a[b-f]*f', text)
print(parser.group()) # 'abcdf'
---↑in_out↓---
abcdf
---↑out---
```

Это выражение «просмотрит» всю переданную ей строку, в данном случае это строка «abcdfghijk». Выражение найдет букву a в начале поиска. После нее должна следовать любая буква из диапазона [b-f], потом любое количество разных символов (за них отвечает звездочка), а заканчиваться все должно буквой f.

Если совпадение не будет найдено, тогда мы получим «None». Чтобы увидеть, как выглядит совпадение, вам нужно вызывать метод `group()`. Список большинства доступных методов доступен в таблице ниже (табл. 9), где RE (англ. *Regular Expression* — «регулярное выражение»).

Таблица 9

**Некоторые доступные методы
модуля регулярных выражений**

Метод / Атрибут	Назначение
<code>group()</code>	Вернуть строку, совпадающую с RE
<code>start()</code>	Вернуть начальную позицию совпадения
<code>end()</code>	Вернуть конечную позицию совпадения
<code>span()</code>	Вернуть кортеж, содержащий «(начало, конец)» позиции совпадения (англ. <i>span</i> — «промежуток»)
<code>match()</code>	Поиск первого совпадения с RE в первой строке
<code>search()</code>	Поиск первого совпадения с RE во всех строках
<code>findall()</code>	Поиск всех совпадений с RE во всех строках. Вернет список (из отфильтрованных строк)
<code>finditer()</code>	Поиск всех совпадений с RE во всех строках. Вернет итератор. Удобно использовать для составных регулярных выражений, например, когда в одной строке содержится адрес абонента в первом совпадении, а в другой строке в третьем совпадении — его ФИО
<code>compile()</code>	«Компиляция» шаблона RE для удобного дальнейшего использования
<code>sub()</code>	Возвращает исходную строку, но с замененными словами, которые совпали с шаблоном RE

Существует еще один метасимвол повторений, аналогичный символу «*». Это символ «+», который будет сопоставлять один или более раз. Разница со звездочкой, которая сопоставляет от ну-

ля и выше раз, незначительна: символу «+» необходимо как минимум одно вхождение искомого символа.

Рассмотрим метасимвол «?», применение которого выглядит так: «со-?ор». Он будет сопоставлять и «соор», и «со-ор», т.е. это означает, что может присутствовать знак, стоящий перед ним.

Еще одна конструкция из метасимволов повторений — это «{a,b}», где a и b являются десятичными целыми числами. Это значит, что должно быть не менее a повторений, но и не более b.

Ниже представлен пример выборки двух комбинаций из четырех доступных: «xz», «xbbz», «xbbbz» и «xbbbbz»:

```
---in↓---
import re
text = 'xbbbbz xz xz xbbz'

parser = re.findall('xb{1,4}z', text)
print(parser) # 'abcdf'
---↑in_out↓---
['xbbbbz', 'xbbz']
---↑out---
```

Выберутся (или отфильтруются) комбинации «xbbbbz» и «xbbz», но не «xz», так как она не содержит b.

Следующий метасимвол — это «^» ((знак «карет», подробнее в источнике [17]), который некоторые люди называют «крышечка», «домик» или «знак возведения в степень»). Этот символ позволяет сопоставить символы, которые не находятся в списке нашего класса. Другими словами, он будет дополнять наш класс, если мы разместим «^» внутри него. Если этот символ находится вне класса, тогда мы попытаемся найти совпадения с данным символом.

Наглядным примером будет следующий: «[^aа]». Это выражение будет искать совпадения с любой буквой, кроме а. Ниже представлен код данного примера.

```
---in↓---
import re
text = 'anton oleg aanton'
```



```
parser = re.findall('[^a]* ', text)
print(parser) # 'abcdf'
---↑in_out↓---
['nton oleg ']
---↑out---
```

Подробнее регулярные выражения и другие метасимволы можно изучить в источнике [18].

Когда вам необходимы только простые операции, предпочтительны встроенные строковые методы, потому что их легче читать и отлаживать, чем регулярные выражения. Например, замена слов при помощи строкового метода `replace()`:

```
---in↓---
'tea for too'.replace('too', 'two')
---↑in_out↓---
'tea for two'
---↑out---
```

Ниже показан пример использования «откомпилированных» регулярных выражений. Также ниже приведено два примера замены определенных слов, соответствующих «откомпилированному» шаблону при помощи метода `sub()`.

```
---in↓---
import re
p = re.compile('(blue|white|red)')

#вывод слов, соответствующих шаблону
print (p.findall('blue socks and red shoes'))

#замена слов, соответствующих шаблону (замена только
нулевого индекса)
my_str=p.sub('colour', 'blue socks and red shoes')
print(my_str)

#замена слов, соответствующих шаблону
#(count=1, значит замена нулевого и первого вхождения)
my_str=p.sub('colour', 'blue socks and red shoes', count=1)
print(my_str)
```

```

---↑in_out↓---
['blue', 'red']
colour socks and colour shoes
colour socks and red shoes
---↑out---

```

Суть «компиляции» регулярного выражения заключается в удобстве записи и использования, т.е. вы отдельно заранее создаете шаблон регулярного выражения, а потом можете удобно применять с ним любые доступные методы.

Математика, случайные числа и статистика

Математический модуль предоставляет доступ к базовым функциям математики, реализованным в откомпилированном модуле Python. Откомпилированные модули Python обладают высокой производительностью. Использование откомпилированных модулей выглядит так же, как и использование неоткомпилированных, но вы не можете посмотреть их исходный код.

Некоторый список математических функций и констант представлен в таблице ниже (табл. 10).

Таблица 10

Некоторые доступные функции и константы библиотеки `math`

Функция/константа	Описание
<code>ceil (x)</code>	Возвращает наименьшее целое число, большее или равное <code>x</code>
<code>pi</code> , <code>e</code>	3.14..., 2.71...
<code>copysign (x, y)</code>	Возвращает <code>x</code> со знаком числа <code>y</code>
<code>fabs (x)</code>	Возвращает абсолютное значение <code>x</code>
<code>factorial (x)</code>	Возвращает факториал <code>x</code>
<code>floor (x)</code>	Возвращает наибольшее целое число, меньшее или равное <code>x</code>

Функция/константа	Описание
<code>frexp (x)</code>	Возвращает мантиссу и показатель степени x как пару «(m, e)»
<code>fsum</code> (итерируемый объект)	Возвращает точную сумму значений с плавающей запятой
<code>isfinite (x)</code>	Возвращает <code>True</code> , если x не является ни бесконечностью, ни NaN (англ. <i>Not-a-Number</i> — «не число»)
<code>isinf (x)</code>	Возвращает <code>True</code> , если x — положительная или отрицательная бесконечность
<code>isnan (x)</code>	Возвращает <code>True</code> , если x — NaN
<code>ldexp (x, i)</code>	Возвращает $x \cdot (2^{**i})$
<code>modf (x)</code>	Возвращает дробную и целую части x
<code>trunc (x)</code>	Возвращает усеченное целочисленное значение x
<code>exp (x)</code>	Возвращает e^{**x}
<code>log (x [, b])</code>	Возвращает логарифм x по основанию b (по умолчанию, без указания b , берется экспонента)
<code>pow (x, y)</code>	Возвращает x в степени y (всегда возвращает число с плавающей точкой, в отличие от встроенного оператора возведения в степень « $**$ »)
<code>atan (x)</code>	Возвращает арктангенс x
<code>atan2 (y, x)</code>	Возвращает арктангенс (y / x)
<code>hypot (x, y)</code>	Возвращает евклидову норму $\sqrt{x^2 + y^2}$
<code>degrees (x)</code>	Преобразует угол x из радиан в градусы
<code>acosh (x)</code>	Возвращает обратный гиперболический косинус x
<code>erf (x)</code>	Возвращает функцию ошибок в x (также упоминается как интеграл вероятности)
<code>gamma (x)</code>	Возвращает гамма-функцию в точке x

Ниже представлен приведен использования модуля `math`:

```
---in↓---
import math
print (math.cos(math.pi / 4))
print(math.log(1024, 2))
---↑in_out↓---
0.7071067811865476
10.0
---↑out---
```

Модуль `random` предоставляет инструменты для случайного выбора и генерации случайных чисел:

```
---in↓---
import random

print(random.choice(['apple', 'pear', 'banana']))
#случайный выбор из списка
print (random.sample(range(100), 10)) # генерация
случайного набора значений
print (random.random())      # случайное с плавающей точкой
float
print (random.randrange(6))   # случайное значение
integer, выбранное из диапазона (6), т.е. от [0 до 6)
---↑in_out↓---
apple
[65, 79, 15, 97, 3, 5, 49, 98, 46, 25]
0.46503625848606844
0
---↑out---
```

Стандартный модуль статистики `statistics` вычисляет основные статистические свойства (среднее значение, медиана, дисперсия и т.д.) числовых данных. Ниже представлен простой пример использования стандартного модуля статистики для вычисления среднего значения, медианы и дисперсии:

```
---in↓---
import statistics
```



```
data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
print (statistics.mean(data)) #среднее
print (statistics.median(data)) #медиана
print (statistics.variance(data)) #дисперсия
---↑in_out↓---
1.6071428571428572
1.25
1.3720238095238095
---↑out---
```

Стандартный модуль статистики `Statistics` будет также рассмотрен позже (стр. 213).

Доступ в Интернет

Существует ряд модулей для доступа в Интернет и обработки интернет-протоколов. Два из самых простых — `urllib.request` для получения данных из URL-адресов и `smtplib` для отправки почты.

Например, в примере ниже скачивается код интернет-страницы, доступной по url: `https://stankin.ru/sys/img/logo_ru.svg`, и отображается в ячейке Jupyter-блокнота (полный вывод данного кода показан на рисунке ниже (рис. 17)).

```
---in↓---
from urllib.request import urlopen
with urlopen('https://stankin.ru/sys/img/logo_ru.svg') as response:
    html = response.read()
    #for line in response:
    #    print(line)

from IPython.display import HTML, display
#отрисовка html-кода
display (HTML(html.decode("utf-8")))
#показать кусочек скачанного html-кода
display (html[:60])
---↑in_out↓---
---вывод показан на рисунке ниже (рис. 17)---
---↑out---
```



```
b'<svg width="567" height="318" viewBox="0 0 567 318" fill="no'
```

Рис. 17. Вывод примера использования `urllib.request`

Даты и время

Модуль `datetime` предоставляет классы для управления датами и временем как простыми, так и сложными способами. Хотя арифметика даты и времени поддерживается, но основное внимание уделяется эффективному извлечению элементов для форматирования и обработки вывода. Модуль также поддерживает объекты с учетом часового пояса.

```
---in↓---
# даты легко конструируются и форматируются
from datetime import date

now = date.today()
print (now)
print (date(2003, 12, 2))

print (now.strftime("%m-%d-%y. %d %b %Y это %A %d день %B."
'12-02-03. 02 Dec 2003 это Вторник или 02 день декабря.'))

# даты поддерживают календарную арифметику
birthday = date(1964, 7, 31)
age = now - birthday
print ("Вам ", age.days, " дней или", int(age.days /
365.25), " лет")
---↑in_out↓---
```

```
2022-01-26
2003-12-02
01-26-22. 26 Jan 2022 это Wednesday 26 день
January.12-02-03. 02 Dec 2003 это Вторник или 02 день
декабря.
Вам 20998 дней или 57 лет
---↑out---
```

Сжатие данных

Стандартные форматы архивирования и сжатия данных напрямую поддерживаются встроенными модулями, включая следующие форматы: «.zlib», «.gzip», «.bz2», «.lzma», «.zip» и «.tar».

Ниже представлен пример сжатия строки «Пой же пой на проклятой гитаре»:

```
---in↓---
import zlib

s = bytes("Пой же пой на проклятой гитаре", 'utf-8')
print("Размер до сжатия: ", len(s))
crc1= zlib.crc32(s)
print ("Контрольная сумма до сжатия: ", crc1)

t = zlib.compress(s)
print ("Размер после сжатия: ", len(t))

t=zlib.decompress(t)
crc2= zlib.crc32(t)
print ("Контрольная сумма после распаковки: ", crc2)
if crc1==crc2:
    print ('\033[92m'"Контрольные суммы
равны!"+'\033[0m')
---↑in_out↓---
Размер до сжатия: 55
Контрольная сумма до сжатия: 1530206883
Размер после сжатия: 53
Контрольная сумма после распаковки: 1530206883
Контрольные суммы равны!
---↑out---
```

Проверка контрольных сумм — важная часть при работе с файлами, она позволяет проверить, не испорчен ли файл при таких манипуляциях, как сохранение, сжатие, перемещение и др.

Измерение производительности

Некоторые программисты Python проявляют глубокий интерес к знанию относительной производительности различных подходов к решению одной и той же проблемы. Python предоставляет инструмент измерения, который сразу же отвечает на эти вопросы.

Первый подход измерения производительности — это использование модуля `timeit`, который позволяет измерить ее для вашей функции, и для этого ему требуется **три параметра**:

- первый параметр — функция, которую вы хотите протестировать;

- второй параметр — начальная инициализация значений для теста функции;

- третий параметр — сколько раз тестировать.

Ниже представлен код, реализующий измерение производительности при помощи модуля `timeit`:

```

---in↓---
#1 вариант измерения производительности
import timeit
iterations=5 # сколько раз тестировать быстродействие
usec=timeit.timeit('a=a**b; print ("Я посчитал:",a)',
setup='a=5; b=2', number= iterations)
print("суммарно за", usec, "микросекунд")
---↑in_out↓---
Я посчитал: 25
Я посчитал: 625
Я посчитал: 390625
Я посчитал: 152587890625
Я посчитал: 23283064365386962890625
суммарно за 0.00030600797617807984 микросекунд
---↑out---
```

Второй подход — измерение времени через таймер (увеличьте число итераций, чтобы увидеть время, так как тут оно в секундах):


```
---in↓---
#2 вариант измерения производительности
import time

start_time = time.time()

for i in range(5):
    a=5; b=2
    a=a**b; print ("Я посчитал:",a)

print("суммарно за %s сек ---" % (time.time() - start_
time))
---↑in_out↓---
Я посчитал: 25
Я посчитал: 25
Я посчитал: 25
Я посчитал: 25
Я посчитал: 25
суммарно за 0.0009772777557373047 сек ---
---↑out---
```

Контроль качества кода

Один из подходов к разработке высококачественного программного обеспечения — это писать тесты для каждой функции по мере ее разработки и часто запускать их в процессе разработки.

Модуль `doctest` предоставляет инструмент для сканирования модуля и проверки тестов, встроенных в строки документации программы. Создание теста производится очень просто: необходимо в строку документации (она же — многострочный комментарий в начале тела функции, см. стр. 40) вставить код, который представляет собой вызов данной функции и передачу ей каких-либо параметров.

Это как улучшает саму документацию к функции, предоставляя пользователю пример ее вызова, так и позволяет модулю `doctest` убедиться, что код проходит данный тест.

В примере ниже продемонстрировано тестирование функции `average()` благодаря встроенному тесту «`print(average([20, 30, 70]))`». Код теста отделяется при помощи символов ввода

«>>>», а после кода теста должен следовать его образцовый результат (валидное значение тестирования, в данном случае — число 40.0):

```
---in↓---
def average(values):
    """Вычисляет среднее арифметическое списка чисел.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # автоматически проверять встроенные
тесты
---↑in_out↓---
TestResults(failed=0, attempted=1)
---↑out---
```

Все содержимое строчек, обособленное при помощи «""", — это комментарий, который будет полностью отображаться, когда вы наводите мышкой на функцию `average()` в среде разработки, например, PyCharm, что представлено на рисунке ниже (рис. 18).

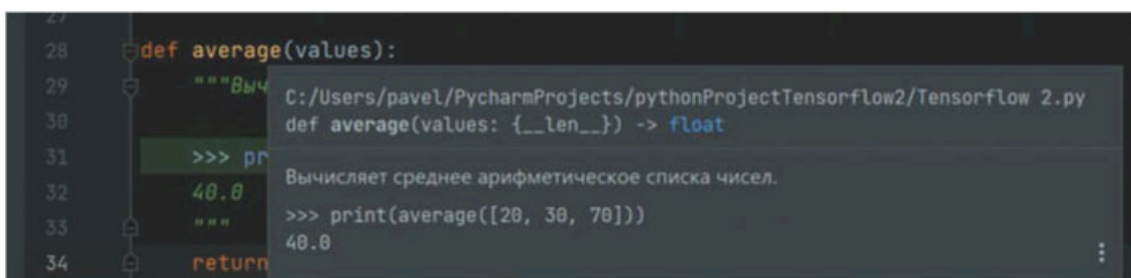


Рис. 18. Вывод комментариев в PyCharm

Модуль `unittest` не так прост, как модуль `doctest`, но он позволяет хранить более полный набор тестов в отдельном файле:

```
---in↓---
import unittest
```

```
class TestStringMethods(unittest.TestCase):

    def test_upper(self): #тест функции upper()
(преобразует регистр в верхний)
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper()) #проверка, что
все в верхнем регистре
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # проверяем, что s.split не работает, если
разделитель не является строкой
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'],
exit=False)
---↑in_out↓---
...
-----
-----
Ran 3 tests in 0.002s
OK
---↑out---
```

Многопоточность и многопроцессорность

Многопоточность — это метод разделения и распределения задач, которые не зависят друг от друга. Потоки можно использовать, например, для распараллеливания выполняемых задач, когда они не зависят друг от друга или связаны при помощи методов синхронизации, как пример — запуск файлового ввода-вывода с вычислениями в другом потоке (т.е. один поток работает с файлом, а другой производит обработку данных этого файла, тем самым обработка данных может начаться раньше, не дожидаясь полного прочтения файла и т.п.).

В следующем коде показано, как модуль может осуществлять некоторые задачи в фоновом режиме, пока основная программа продолжает выполняться. В ней мы производим запаковку файла в отдельном потоке в фоновом режиме.

```

---in↓---
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_
DEFLATED)
        f.write(self.infile)
        f.close()
        print('Готова фоновая zip запаковка файла:', self.
infile)
f = open("mydata.txt", "a")
f.write(«Теперь в файле больше содержимого!»)
f.close()

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('Основная программа продолжает работать на переднем
плане.')
```

background.join() # Дождитесь завершения фоновой задачи
print('Основная программа дождалась завершения фоновой
обработки.')

```

---↑in_out↓---
```

Основная программа продолжает работать на переднем плане.
Готова фоновая zip запаковка файла: mydata.txt

Основная программа дождалась завершения фоновой обработки.
---↑out---

Основная проблема многопоточных приложений — это координация потоков, которые совместно используют данные или другие общие ресурсы. С этой целью модуль потоковой передачи предоставляет ряд примитивов синхронизации: блокировки, события, переменные состояния и семафоры.

Несмотря на то что эти инструменты являются мощными, незначительные ошибки проектирования могут привести к проблемам, которые трудно воспроизвести. Таким образом, предпочтительный подход к координации задач состоит в том, чтобы сконцентрировать весь доступ к ресурсу в одном потоке, а затем использовать модуль очереди для подачи в этот поток запросов из других потоков. Приложения, применяющие объекты `Queue` (с англ. — «очередь», т.е. в контексте — «использующие очереди») для межпоточного взаимодействия и координации, проще разрабатывать, они более читабельны и надежны.

Модуль `threading` использует потоки, а модуль `multiprocessing` — процессы. Разница в том, что потоки выполняются в одном и том же пространстве памяти (позволяет интерпретатору использовать только одно логическое ядро процессора из всех доступных), в то время как процессы имеют отдельную память. При этом каждый процесс может состоять из множества потоков.

Многопоточность при помощи `threading` применяется для задач, связанных с вводом-выводом / памятью (например, работы с файлами).

Многопроцессорность при помощи `multiprocessing` достигает полного (истинного) параллелизма и используется для задач, связанных с процессором (для сложных параллельных вычислений).

Пример `multiprocessing`: умножение чисел в массиве на 2:

```
---in↓---
command="""
import os
from multiprocessing import Process

def doubler(number):
    \"""
    Функция умножитель на два
```

```

\"""
result = number * 2
proc = os.getpid()
print('{0} doubled to {1} by process id: {2}\\n'.
format(number, result, proc), end="")

if __name__ == '__main__':
    import time

    numbers = [5, 10, 15, 20, 25]
    procs = []
    start_time = time.time()
    print ('0сек-> Создание и запуск процессов...',
flush=True)
    for index, number in enumerate(numbers):
        proc = Process(target=doubler, args=(number,))
        procs.append(proc)
        proc.start()

    print ('%.2фсек-> Все процессы запущены. Ожидание
завершения процессов...' % (time.time() - start_time),
flush=True)
    for proc in procs:
        proc.join()

    print("%.2фсек-> Все процессы завершены, умножение
окончено." % (time.time() - start_time))
    print ("Так долго, потому что накладные расходы
большие. Умножение работает быстро, а вот создание
потокa и процессов – нет... ")
    print("По сути, на создание и ожидание одного
процесса уходит около %.2f миллисекунд" % ((time.time()
- start_time)/len(procs)*1000))
"""
with open("my_multiprocessing.py", "w", encoding="utf-8")
as text_file:
    text_file.write(command)

#запуск программы через системный интерпретатор,
а не блокнотовский

```

```
!python "my_multiprocessing.py"
---↑in_out↓---
0сек-> Создание и запуск процессов...
0.02сек-> Все процессы запущены. Ожидание завершения
процессов...
5 doubled to 10 by process id: 10960
10 doubled to 20 by process id: 11448
15 doubled to 30 by process id: 2360
20 doubled to 40 by process id: 8556
25 doubled to 50 by process id: 7944
0.12сек-> Все процессы завершены, умножение окончено.
Так долго, потому что накладные расходы большие.
Умножение работает быстро, а вот создание потоков
и процессов — нет...
По сути, на создание и ожидание одного процесса уходит
около 23.26 миллисекунд
---↑out---
```

В данном примере каждый экземпляр функции умножения выполняется отдельным процессом с уникальным номером (или идентификатором, или *id* — часть англ. слова *identifier* — «идентификатор»). То есть за счет многопоточности мы в итоге могли бы выполнить данную программу ровно в 5 раз быстрее, потому что элементов в массиве 5, и было бы выполнено 5 истинно²¹ параллельных умножений, если на компьютере доступно 5 ядер процессора.

По факту это оказалось не так, потому что само умножение работает быстро, а создание потоков и процессов — медленно, и возникают слишком большие накладные расходы. Но если бы наша функция делала что-то более сложное, чем одну операцию умножения, то мы бы увидели почти кратный прирост производительности.

Стоит отметить, что в данном коде сам код программы сохраняется в виде файла и запускается не при помощи `ipython`

²¹ Современные процессоры и операционные системы имеют адаптивный кэш и гипервизор задач, и на самом деле могут выполнять множественные процессы и потоки возможно только лишь на одном ядре процессора, потому что так может оказаться быстрее, чем при распределении по ядрам. Нужно грамотно подходить к распределению задач при проектировании программы.

Jupyter-интерпретатора, а при помощи системного Python-интерпретатора. Так сделано, потому что на Windows модуль `multiprocessing` имеет трудности при работе в интерактивном интерпретаторе. Этот же код без проблем работает в Linux прямо в ячейке ноутбука и без сохранения в отдельный файл. Это вызвано особенностями процессов/потоков конкретных операционных систем (см. официальную документацию — [19]). В целом работать с `multiprocessing` в блокноте особого смысла не имеет, так что вряд ли это проблема. Особенно учитывая, что это ограничение есть только с Windows.

В завершение занятия удалим все созданные файлы и папки, чтобы не засорять память компьютера:

```
---in↓---
import os, shutil

files = ['top.py', 'my_multiprocessing.py', 'myarchive.
zip', 'mydata.txt']
for f in files:
    if os.path.isfile(f): # если файл существует
        os.remove(f)

if os.path.isdir('my_dir'): # если папка существует
    shutil.rmtree('my_dir')
---↑in_out↓---
---↑out---
```

А также закроем базу данных и удалим ее:

```
---in↓---
connection.close()

import os
if os.path.isfile('test.db'): # если файл существует
    os.remove('test.db')
---↑in_out↓---
---↑out---
```


? Задания для проверки

1. Что такое командная оболочка операционной системы, например терминал или командная строка? Как и для чего с ней можно взаимодействовать через Python? Приведите пример.

2. Работа с `glob` и регулярные выражения. Приведите прокомментированные примеры работы с этими двумя модулями и сделайте вывод, в каких случаях их применение может быть полезным при аналитике данных.

3. Поэкспериментируйте с модулем `datetime` и приведите какой-либо собственный пример кода работы с этой библиотекой.

4. Многопоточность. Приведите примеры, когда она может быть полезна для аналитики данных.

РАБОТА С МАССИВАМИ ДАННЫХ

БИБЛИОТЕКА NUMPY²²

Цели выполнения работы

Получение представления о функциональности, доступных методах и объектах библиотеки NumPy. Изучение основных принципов практической работы с ними.

Порядок выполнения работы

NumPy — это библиотека языка Python, добавляющая поддержку больших многомерных массивов и матриц вместе с большой библиотекой высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.

Основным объектом NumPy является однородный многомерный массив (в numpy называется `numpy.ndarray`). Это многомерный массив элементов (обычно чисел) одного типа.

Ниже представлены наиболее важные атрибуты объектов `ndarray`:

- `ndarray.ndim` — число измерений (чаще их называют «оси») массива;

- `ndarray.shape` — размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси.

²² Разработано на основе русскоязычного источника [20] и официального руководства NumPy [21].

Для матрицы из n строк и m столбцов `shape` будет (n, m) . Число элементов кортежа `shape` равно `ndim`;

- `ndarray.size` — количество элементов массива. Очевидно, равно произведению всех элементов атрибута `shape`;

- `ndarray.dtype` — объект, описывающий тип элементов массива. Можно определить `dtype`, используя стандартные типы данных Python. NumPy здесь предоставляет целый букет возможностей, как встроенных, например: `bool`, `character`, `int8`, `int16`, `int32`, `int64`, `float8`, `float16`, `float32`, `float64`, `complex64`, `object`, так и возможность определить собственные типы данных, в том числе и составные;

- `ndarray.itemsize` — размер каждого элемента массива в байтах;

- `ndarray.data` — буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

Создание массивов

В NumPy существует много способов создать массив. Один из наиболее простых — создание из обычных списков или кортежей Python, используя функцию `numpy.array()` (обратите внимание: `array()` — это метод, создающий объект типа `ndarray`):

```
---in↓---
import numpy as np

a = np.array([1, 2, 3])
print(a)
print(type(a)) #вывод типа переменной a
---↑in_out↓---
[1 2 3]
<class 'numpy.ndarray'>
---↑out---
```

Приведем пример создания двухмерного массива, где элемент `b[0][0] = 1.5`, а элемент `b[1][0] = 4`:

```

---in↓---
b = np.array([[1.5, 2, 3], [4, 5, 6]])
---↑in_out↓---
---↑out---
```

Функция `array()` трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности (но можно и переопределить его в момент создания). Например, в примере ниже, так как один из элементов, т.е. число 1.5, принадлежит к типу `float`, то и остальные элементы будут восприняты как `float` (о чем говорит точка после числа, и при этом без указания значения дробной части):

```

---in↓---
b = np.array([[1.5, 2, 3], [4, 5, 6]])
print (b)
---↑in_out↓---
[[1.5 2.  3. ]
 [4.  5.  6. ]]
---↑out---
```

Можно также переопределить тип в момент создания:

```

---in↓---
b = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=complex)
print (b)
---↑in_out↓---
[[1.5+0.j 2. +0.j 3. +0.j]
 [4. +0.j 5. +0.j 6. +0.j]]
---↑out---
```

Функция `array()` — не единственная функция для создания массивов. Обычно элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип создаваемого массива — `float64`).

Функция `zeros()` создает массив из нулей, а функция `ones()` — из единиц. Обе функции принимают кортеж с размерами и аргумент `dtype`, ее синтаксис следующий:

```
numpy.ones(shape, dtype=None, order='C', *, like=None)
```

Она возвращает новый массив заданной формы, типа и порядка, заполненный единицами, для этого ей необходимо передать следующие **параметры**:

- `shape` — последовательность целых чисел, определяющая форму нового массива, например `(2, 3)` или `2` (одномерный массив с двумя элементами);

- `dtype` — необязательный параметр Желаемый тип данных для массива, например, `numpy.int8`. По умолчанию `numpy.float64`;

- `order{'C', 'F'}` — необязательный параметр, по умолчанию равный `C`, определяющий порядок сохранения многомерных данных в памяти в порядке строк (стиль `C`, т.е. стиль языка `C`) или столбцов (стиль языка `Fortran`);

- `likearray_like` — ссылочный объект, позволяющий создавать массивы, которые не являются массивами NumPy (полезно, если вы хотите сделать из обычного массива массив NumPy).

Ниже представлен пример создания массивов двух разных форм, заполненных нулями:

```
---in↓---
print (np.zeros((3, 5)))
print('\n')
print (np.ones((2, 2, 2)))
---↑in_out↓---
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

[[[1. 1.]
  [1. 1.]]
```

```
[[1. 1.]
 [1. 1.]]
---↑out---
```

Функция `eye()` создает единичную матрицу (двухмерный массив, где по диагонали расположены единицы):

```
---in↓---
np.eye(5)
---↑in_out↓---
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
---↑out---
```

Функция `empty()` создает массив без его заполнения. Исходное содержимое случайно и зависит от состояния памяти на момент создания массива (т.е. от того «мусора», что в ней хранится):

```
---in↓---
print (np.empty((3, 3)))
print('Если вам повезет, то "мусор" должен быть разный')
print (np.empty((3, 3)))
---↑in_out↓---
[[1.52207159e-316 0.00000000e+000 4.94065646e-324]
 [4.94065646e-324 6.90276400e-310 6.90264011e-310]
 [6.90276292e-310 1.22814776e-314 0.00000000e+000]]
Если вам повезет, то "мусор" должен быть разный
[[1.52207159e-316 0.00000000e+000 4.94065646e-324]
 [4.94065646e-324 6.90276400e-310 6.90264011e-310]
 [6.90276292e-310 1.22814776e-314 0.00000000e+000]]
---↑out---
```

Для создания последовательностей чисел в NumPy имеется функция `arange()`, аналогичная встроенной в Python функции `range()` (ее мы рассматривали при изучении цикла `for`), только вместо списков она возвращает массивы и принимает не только целые значения. Ее синтаксис следующий:

```
numpy.arange ([start,] stop, [step,], dtype = None)
```

Она возвращает объект типа «`numpy.ndarray`». Первые три параметра определяют диапазон значений, а четвертый — указывает тип элементов:

- `start` — это число (целое или десятичное), которое определяет первое значение в массиве;

- `stop` — это число, определяющее конец массива и не включенное в массив;

- `step` — это число, которое определяет интервал (разность) между каждыми двумя последовательными значениями в массиве и по умолчанию равно 1;

- `dtype` — это тип элементов выходного массива; по умолчанию используется значение `None`.

Ниже представлен пример использования функции `arange()`:

```
---in↓---
print (np.arange(10, 30, 5))
print (np.arange(0, 1, 0.1))
---↑in_out↓---
[10 15 20 25]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
---↑out---
```

Если функция `arange()` используется с аргументами типа `float`, то предсказать количество элементов в возвращаемом массиве непросто. Гораздо чаще возникает необходимость указания не шага изменения чисел в заданном диапазоне, а количества чисел в нем. Функция `linspace()`, так же как и `arange()`, принимает три аргумента, но третий аргумент как раз и указывает количество чисел в диапазоне. Ниже представлен пример использования функции `linspace()`:

```
---in↓---
np.linspace(0, 2, 9) # 9 чисел от 0 до 2 включительно
---↑in_out↓---
array([0. , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
---↑out---
```

Функция `fromfunction()` создает массив NumPy посредством применения пользовательской функции ко всем комбинациям индексов. Для этого она требует передать ей следующие **параметры**:

- `function` — подлежащая выполнению функция. Она может содержать N параметров, при этом количество параметров определяет размерность выходного массива. Каждый из N параметров перебирает элементы вдоль определенной оси. Например, если мы используем два параметра и указываем размеры массива (3, 3), то один из параметров «пробежал» бы значения массива `array([[0, 0, 0], [1, 1, 1], [2, 2, 2]])`, а другой — значения массива `array([[0, 1, 2], [0, 1, 2], [0, 1, 2]])`;

- `shape` — целое число, список или кортеж целых чисел. Задает размеры необходимого массива;

- `dtype` — тип данных NumPy (необязательный параметр). Определяет тип данных выходного массива.

Ниже представлен пример использования функции `fromfunction()`, в котором на основе индексов элементов был определен массив, где в значения записался результат вычисления индекса для конкретного элемента:

```
---in↓---
def f1(i, j):
    return 4 * i + j

print (np.fromfunction(f1, (3, 4))) # (3, 4)- размер массива
print (np.fromfunction(f1, (3, 3)))
---↑in_out↓---
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
[[ 0.  1.  2.]
 [ 4.  5.  6.]
 [ 8.  9. 10.]]
---↑out---
```

Структурированные массивы

Существует возможность создавать массивы с разными типами данных в элементах. То есть в одном массиве могут быть как текстовые, целочисленные, так и другие элементы. Для этого нуж-

но произвести разметку типов при создании массива. Ниже представлен пример, как это можно сделать и работать с этим.

```
---in↓---
#разметка имен и типов полей
student_def = [("name", "S10"), ("marks", "f8")]

#создание структурированного массива, заполненного нулями
#в качестве типа данных передаем нашу разметку student_def
#2 — количество элементов
student_array = np.zeros((3), dtype=student_def)

#добавление двух оценок в наш структурированный массив
student_array[0] = ("Notmelikov", 95)
student_array[1] = ("Melikov", 90)
student_array[2] = ("Melikov1234567", 91)

print ('1)', student_array)
print ('2)', student_array["name"])
print ('3)', student_array["marks"])

# Получение имени людей, у которых оценка больше 93
print ('4)', student_array[student_array['marks'] > 93]
['name'])

# Получение оценки для тех, у кого имя равно "Melikov"
#буква b перед именем означает, что его нужно взять
в байтовом представлении
print ('5)', student_array[student_array['name'] ==
b"Melikov"]['marks'])

#Получение тех, у кого имя начинается на 'Melikov'
who=np.char.startswith(student_array['name'],b'Melikov')
#получение их индексов
indexes=np.flatnonzero(who)
#печать тех оценок для людей с этими индексами
print ('6)', student_array[indexes]['marks'])

#сортировка по возрастанию оценок, а затем по имени
sorted_by_marks = np.sort(student_array, order=['marks',
'name'])
print ('7)', sorted_by_marks)
```

```

---↑in_out↓---
1) [(b'Notmelikov', 95.) (b'Melikov', 90.) (b'Melikov123',
91.)]
2) [b'Notmelikov' b'Melikov' b'Melikov123']
3) [95. 90. 91.]
4) [b'Notmelikov']
5) [90.]
6) [90. 91.]
7) [(b'Melikov', 90.) (b'Melikov123', 91.) (b'Notmelikov',
95.)]
---↑out---
```

`S10` — это тип `string` с длиной 10 букв (в примере также показано, что если попытаться задать больше букв, то часть из них просто пропадет), а тип `f8` — это `float8`, подробнее о доступных типах в NumPy можно почитать в источнике [22].

Структурированные массивы похожи на структуры в языке C, они могут быть полезны, потому что просты и работают очень быстро.

Печать массивов

Если в массиве слишком много элементов, чтобы их все отобразить на экране, NumPy автоматически скрывает центральную часть массива и выводит только его крайние элементы.

```

---in↓---
print(np.arange(0, 3000, 1))
---↑in_out↓---
[  0    1    2 ... 2997 2998 2999]
---↑out---
```

Если же вам действительно нужно увидеть весь массив, используйте функцию `numpy.set_printoptions()` для установки числа выводимых элементов. В примере ниже мы указываем библиотеке NumPy вывести столько элементов, сколько вообще может содержать структура данных на данной платформе (на основе системной переменной `sys.maxsize`), но вы можете установить любое целое число:

```
---in↓---
import sys

#sys.maxsize – максимальное значение числа типа Py_ssize_t,
#т.е., например, максимальная размерность массива
np.set_printoptions(threshold=sys.maxsize) #threshold –
#максимальное допустимое для вывода число элементов.
print(np.arange(0, 200, 1))
print('Вы бы могли работать с массивами размером ', sys.
maxsize)
---↑in_out↓---
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
---вывод обрезан, тут были числа от 18 до 179---
180 181 182 183 184 185 186 187 188 189 190 191 192 193
194 195 196 197 198 199]
Вы бы могли работать с массивами размером
9223372036854775807
---↑out---
```

В данном примере можно вывести на экран сразу все элементы массива, размер которого может быть вплоть до более чем 9 квинтиллионов элементов, т.е. максимальное количество элементов для 64-битной системы (соответствует типу `int64`).

Также с помощью этой функции можно настроить печать массивов «под свои нужды». Функция `numpy.set_printoptions()` принимает несколько аргументов:

- `precision` — количество отображаемых цифр после запятой (по умолчанию 8);
- `threshold` — количество элементов в массиве, вызывающее обрезание элементов (по умолчанию 1 000);
- `edgeitems` — количество элементов в начале и в конце каждой размерности массива (по умолчанию 3);
- `linewidth` — количество символов в строке, после которых осуществляется перенос (по умолчанию 75);
- `suppress` — если `True`, не печатает маленькие значения в `scientific notation` (с англ. — «в научной нотации», по умолчанию `False`);
- `nanstr` — строковое представление неопределенных значений `NaN`;

и еще несколько очень полезных параметров, которые можно изучить в официальной документации (см. источник [23]).

Базовые операции

Математические операции над массивами выполняются поэлементно. Ниже представлены примеры математических операций между двумя массивами. Обратите внимание, что при делении на 0 интерпретатор выдает предупреждение:

```

---in↓---
import numpy as np

a = np.array([20, 30, 40, 50])
b = np.arange(4)

print ('a=',a)
print ('b=',b)

print ("\n1)", a + b)
print ("2)", a - b)
print ("3)", a * b)
print ("4)", a / b) # При делении на 0 возвращается inf
(бесконечность)
print ("5)", a ** b) # Возведение в степень
print ("6)", a % b) # При взятии остатка от деления на
0 возвращается
---↑in_out↓---
0
a= [20 30 40 50]
b= [0 1 2 3]

1) [20 31 42 53]
2) [20 29 38 47]
3) [ 0 30 80 150]
4) [          inf 30.          20.          16.66666667]
5) [          1          30      1600 125000]
6) [0 0 0 2]
<ipython-input-12-b1849c29d9fa>:12: RuntimeWarning:
divide by zero encountered in true_divide

```



```
print ("4)", a / b) # При делении на 0 возвращается
inf (бесконечность)
<ipython-input-12-b1849c29d9fa>:14: RuntimeWarning:
divide by zero encountered in remainder
print ("6)", a % b) # При взятии остатка от деления
на 0 возвращается 0
---↑out---
```

Для подобных действий, естественно, массивы должны быть одинаковых размеров, иначе произойдет ошибка:

```
---in↓(raises-exception)---
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[1, 2], [3, 4], [5, 6]])
c + d
---↑in_out↓---
-----
-----
ValueError                                Traceback (most
recent call last)
<ipython-input-13-76dd06fe5117> in <module>
      2 d = np.array([[1, 2], [3, 4], [5, 6]])
      3
----> 4 c + d
ValueError: operands could not be broadcast together with
shapes (2,3) (3,2)
---↑out---
```

Также можно производить математические операции между массивом и числом. В этом случае к каждому элементу применяется результат вычисления между этим числом и элементом массива.

```
---in↓---
print ('a=', a)
print ("1)", a + 1)
print ("2)", a ** 3)
print ("3)", a < 35) # Можно производить фильтрацию
---↑in_out↓---
a= [20 30 40 50]
1) [21 31 41 51]
```

```

2) [ 8000 27000 64000 125000]
3) [ True  True False False]
---↑out---
```

NumPy также предоставляет множество математических операций для обработки массивов. Полный список можно посмотреть в источнике [24]. Ниже представлен пример получения косинуса:

```

---in↓---
print ('a=',a)
np.cos(a)
---↑in_out↓---
a= [20 30 40 50]
array([ 0.40808206,  0.15425145, -0.66693806,
 0.96496603])
---↑out---
```

Многие унарные операции, такие как, например, вычисление суммы всех элементов массива, представлены также и в виде методов класса ndarray.

```

---in↓---
a = np.array([[1, 2, 3], [4, 5, 6]])

print ("1)", np.sum(a))
print ("2)", a.sum()) # у "a" есть все эти методы, ведь
оно тоже член библиотеки numpy
print ("3)", a.min())
print ("4)", a.max())
---↑in_out↓---
1) 21
2) 21
3) 1
4) 6
---↑out---
```

По умолчанию эти операции применяются к массиву, как если бы он был списком чисел, независимо от его формы. Однако, указав параметр `axis`, можно применить операцию для указанной оси массива:

```
---in↓---
print (a.min(axis=0)) # Наименьшее число в каждом столбце
print (a.min(axis=1)) # Наименьшее число в каждой строке
---↑in_out↓---
[1 2 3]
[1 4]
---↑out---
```

Индексы, срезы, итерации

Одномерные массивы осуществляют операции индексирования, срезов и итераций очень схожим образом с обычными списками и другими последовательностями Python.

```
---in↓---
a = np.arange(10) ** 3
print ("1)", a)
print ("2)", a[1])
print("3)", a[3:7])

a[3:7] = 8
print ("4)", a)

print("5)", a[::-1])

for i in a:
    print(i ** (1/3))
---↑in_out↓---
1) [ 0  1  8 27 64 125 216 343 512 729]
2) 1
3) [ 27  64 125 216]
4) [ 0  1  8  8  8  8  8 343 512 729]
5) [729 512 343  8  8  8  8  8  1  0]
0.0
1.0
2.0
2.0
2.0
2.0
2.0
```

```
6.9999999999999999
7.9999999999999999
8.9999999999999998
---↑out---
```

Удалять при помощи срезов, т.е. представленным ниже способом, не получится:

```
---in↓(raises-exception)---
del a[4:6] # возникнет ошибка. Удалять нельзя
---↑in_out↓---
-----
-----
ValueError                                Traceback (most
recent call last)
<ipython-input-20-b962281a0530> in <module>
----> 1 del a[4:6] #возникнет ошибка. Удалять нельзя
ValueError: cannot delete array elements
---↑out---
```

У многомерных массивов на каждую ось приходится один индекс. Индексы передаются в виде последовательности чисел, разделенных запятыми (т.е. кортежами):

```
---in↓---
b = np.array([[ 0, 1, 2, 3],
               [10, 11, 12, 13],
               [20, 21, 22, 23],
               [30, 31, 32, 33],
               [40, 41, 42, 43]])
print ('b=', b)
print ("1) Вторая строка, третий столбец: ", b[2,3])
print ("2) То же самое: ", b[(2,3)])
print ("3) То же самое: ", b[2][3])
print ("4) Третий столбец: ", b[:,2])
print ("5) Первые две строки: ", b[:2])
print ("5) Вторая и третья строки: ", b[1:3, : : ] )
---↑in_out↓---
b= [[ 0  1  2  3]
     [10 11 12 13]
```



```
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]]
1) Вторая строка, третий столбец: 23
2) То же самое: 23
3) То же самое: 23
4) Третий столбец: [ 2 12 22 32 42]
5) Первые две строки: [[ 0  1  2  3]
[10 11 12 13]]
5) Вторая и третья строки: [[10 11 12 13]
[20 21 22 23]]
---↑out---
```

Когда индексов меньше, чем осей, отсутствующие индексы предполагаются дополненными с помощью срезов:

```
---in↓---
b[-1] # Последняя строка. Эквивалентно b[-1, :]
---↑in_out↓---
array([40, 41, 42, 43])
---↑out---
```

`b[i]` можно читать как:

"`b[i, <столько символов ':' , сколько нужно>]`"

В NumPy это также может быть записано с помощью точек как «`b[i, ...]`».

Например, если `x` имеет ранг 5 (т.е. у него 5 осей), тогда:

- `x[1, 2, ...]` эквивалентно `x[1, 2, :, :, :]`;
- `x[... , 3]` то же самое, что: `x[:, :, :, :, 3]`;
- `x[4, ... , 5, :]` эквивалентно `x[4, :, :, 5, :]`.

Далее представлен пример, демонстрирующий описанное выше:

```
---in↓---
a = np.array([[0, 1, 2], [10, 12, 13]], [[100, 101,
102], [110, 112, 113]])
print ('a=', a)
```

```

print ("1"),a.shape) #вывод "формы", т.е. размерности
print ("2"),a[1, ...]) # то же, что a[1, :, :] или a[1]
print ("3"),a[... ,2]) # то же, что a[:, :, 2]
---↑in_out↓---
a= [[[ 0  1  2]
      [10 12 13]]

      [[100 101 102]
       [110 112 113]]]
1) (2, 2, 3)
2) [[100 101 102]
     [110 112 113]]
3) [[ 2 13]
     [102 113]]
---↑out---
```

Итерирование многомерных массивов начинается с первой оси:

```

---in↓---
for row in a:
    print(row)
---↑in_out↓---
[[ 0  1  2]
 [10 12 13]]
[[100 101 102]
 [110 112 113]]
---↑out---
```

Однако если мы столкнемся с ситуацией, когда необходимо перебрать поэлементно весь массив, как если бы он был одномерным, то для этого будет полезен атрибут `flat`:

```

---in↓---
print ('a=',a)
for el in a.flat:
    print(el)
---↑in_out↓---
a= [[[ 0  1  2]
      [10 12 13]]
```

```
[[100 101 102]
 [110 112 113]]
0
1
2
10
...
110
112
113
---↑out---
```

Ранее мы уже рассматривали, что, написав после `for` две переменных, первая будет итератором верхнего уровня списка, а вторая — итератором более низкого уровня списка: `a[i][j]` (сноска 13 на стр. 90).

```
---in↓---
for i, j in a:
    print(i, j)
---↑in_out↓---
[0 1 2] [10 10 10]
[100 101 102] [10 10 10]
---↑out---
```

Манипуляции с формой

Как уже говорилось, у массива есть форма `shape`, определяемая числом элементов вдоль каждой оси:

```
---in↓---
print ('a=', a)
print (a.shape) #вывод "формы", т.е. размерности
---↑in_out↓---
a= [[[ 0  1  2]
      [ 10 10 10]]

      [[100 101 102]
       [ 10 10 10]]]
(2, 2, 3)
---↑out---
```

Форма массива может быть изменена с помощью различных методов, примеры использования которых представлены ниже:

- метод `ravel()` — сделать массив плоским:

```
---in↓---
a.ravel() # Делает массив плоским
---↑in_out↓---
array([ 0,  1,  2, 10, 10, 10, 100, 101, 102, 10, 10, 10])
---↑out---
```

- метод `shape()` — смена формы массива:

```
---in↓---
a.shape = (6, 2) # Изменение формы
a
---↑in_out↓---
array([[ 0,  1],
       [ 2, 10],
       [10, 10],
       [100, 101],
       [102, 10],
       [ 10, 10]])
---↑out---
```

- метод `transpose()` — транспонирование:

```
---in↓---
a.transpose() # Транспонирование
---↑in_out↓---
array([[ 0,  2, 10, 100, 102, 10],
       [ 1, 10, 10, 101, 10, 10]])
---↑out---
```

- метод `reshape()` — изменение формы массива. Отличие от `shape()` заключается в том, что `reshape()` возвращает новое представление массива, т.е. исходный массив не изменяется. Представление будет рассмотрено позже.

```
---in↓---
a.reshape((3, 4)) # Изменение формы
```

```
---↑in_out↓---
array([[ 0,  1,  2, 10],
       [10, 10, 100, 101],
       [102, 10, 10, 10]])
---↑out---
```

Порядок элементов в массиве в результате функции `ravel()` соответствует следующему стилю: чем правее индекс, тем он «быстрее изменяется» — за элементом `a[0, 0]` следует `a[0, 1]`. Если одна форма массива была изменена на другую, массив переформируется также в этом же стиле.

Функции `ravel()` и `reshape()` также могут работать (при использовании дополнительного аргумента) и в другом стиле, в котором быстрее изменяется более левый индекс. Эти два стиля называются C (имеется в виду стиль массива в языке C) и Fortran стили соответственно.

```
---in↓---
a = np.array([[0, 1, 2], [10, 12, 13]], [[100, 101,
102], [110, 112, 113]])
print (a)
print ("Форма исходного массива:", a.shape)
print (a.reshape((1, 12), order='C'))
print (a.reshape((1, 12), order='F'))
---↑in_out↓---
[[[ 0  1  2]
  [10 12 13]]

 [[100 101 102]
  [110 112 113]]]
Форма исходного массива: (2, 2, 3)
[[ 0  1  2 10 12 13 100 101 102 110 112 113]]
[[ 0 100 10 110  1 101 12 112  2 102 13 113]]
---↑out---
```

Метод `reshape()` возвращает ее аргумент с измененной формой (новое представление), в то время как метод `resize()` изменяет сам массив:


```

---in↓---
a = np.array([[0, 1, 2], [10, 12, 13]], [[100, 101,
102], [110, 112, 113]])

print (a)
print (a.reshape((1, 12)))
print("1)\n", a)

print (a.resize((1, 12)))
print("2)\n", a)
---↑in_out↓---
[[[ 0  1  2]
  [ 10 12 13]]

 [[100 101 102]
  [110 112 113]]]
[[ 0  1  2 10 12 13 100 101 102 110 112 113]]
1)
[[[ 0  1  2]
  [ 10 12 13]]

 [[100 101 102]
  [110 112 113]]]
None
2)
[[ 0  1  2 10 12 13 100 101 102 110 112 113]]
---↑out---

```

Если при операции такой перестройки один из аргументов задается как -1, то он автоматически рассчитывается в соответствии с остальными заданными (т.е. в массиве было 12 элементов, значит, 12 элементов делят на 3 строки, что равно 4 элементам в каждой строке):

```

---in↓---
a.reshape((3, -1))
---↑in_out↓---
array([[ 0,  1,  2, 10],
       [12, 13, 100, 101],
       [102, 110, 112, 113]])
---↑out---

```

Объединение массивов

Несколько массивов могут быть объединены вместе вдоль разных осей с помощью методов `hstack()` и `vstack()`. `hstack()` объединяет массивы по первым осям, `vstack()` — по последним:

```
---in↓---
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print ( np.vstack((a, b)) )
print("\n")
print ( np.hstack((a, b)) )
---↑in_out↓---
[[1 2]
 [3 4]
 [5 6]
 [7 8]]

[[1 2 5 6]
 [3 4 7 8]]
---↑out---
```

Метод `column_stack()` объединяет одномерные массивы в качестве столбцов двумерного массива:

```
---in↓---
np.column_stack((a, b))
---↑in_out↓---
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
---↑out---
```

Аналогично для строк имеется функция `row_stack()`:

```
---in↓---
np.row_stack((a, b))
---↑in_out↓---
array([[1, 2],
       [3, 4],
```

```

    [5, 6],
    [7, 8]])
---↑out---
```

Разбиение массива

Используя `hsplit()`, вы можете разбить массив вдоль горизонтальной оси, указав либо число возвращаемых массивов одинаковой формы, либо номера столбцов, после которых массив разрезается «ножницами». Функция `vsplit()` разбивает массив вдоль вертикальной оси, а `array_split()` позволяет указать оси, вдоль которых произойдет разбиение.

```

---in↓---
a = np.arange(12).reshape((2, 6))
print (a)
print (np.hsplit(a, 3)) # Разбить на 3 части
print (np.hsplit(a, (3, 4))) # Разрезать a после третьего
и четвертого столбца
---↑in_out↓---
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
[array([[0, 1],
        [6, 7]]), array([[2, 3],
        [8, 9]]), array([[4, 5],
        [10, 11]])]
[array([[0, 1, 2],
        [6, 7, 8]]), array([[3],
        [9]]), array([[4, 5],
        [10, 11]])]
---↑out---
```

Копии и представления

При работе с массивами их данные иногда необходимо копировать в другой массив, а иногда нет. Это часто является источником путаницы. Возможно **три случая**:

- вообще никаких копий. Простое присваивание не создает ни копии массива, ни копии его данных. Python передает изменение

мые объекты как ссылки, поэтому вызовы функций также не создают копий.

```
---in↓---
a = np.arange(12)
b = a # Нового объекта создано не было
print(a)
print(b)
```

`print (b is a)` # `a` и `b` – это два имени для одного и того же объекта `ndarray`. Мы задали интерпретатору вопрос: является ли объект `b` объектом `a`?

```
b.shape = (3,4) # изменит форму a
print (a.shape)
---↑in_out↓---
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[ 0  1  2  3  4  5  6  7  8  9 10 11]
True
(3, 4)
---↑out---
```

- представление или поверхностная копия. Разные объекты массивов могут использовать одни и те же данные. Метод `view()` создает новый объект массива, являющийся представлением тех же данных.

```
---in↓---
c = a.view()
print (c is a)
print (c.base is a) # c – это представление данных,
                     принадлежащих a
print ( c.flags.owndata)
c.shape = (2,6) # форма a не поменяется
print (a.shape)
c[0,4] = 1234 # данные a изменятся
print (a)
---↑in_out↓---
False
True
```

```
False
(3, 4)
[[ 0  1  2  3]
 [1234 5 6 7]
 [ 8  9 10 11]]
---↑out---
```

Стоит отметить, что уже рассмотренная операция среза (слайсинга) массива тоже является представлением и не меняет исходный массив:

```
---in↓---
print (a, "\n")
s = a[:,1:3] #тут мы строчки не срезали, а вырезали
столбцы с индексом от 1 включительно до 3 не включительно
s[:] = 10
print (a, "\n")
print (s)
---↑in_out↓---
[[ 0  1  2  3]
 [1234 5 6 7]
 [ 8  9 10 11]]

[[ 0 10 10  3]
 [1234 10 10 7]
 [ 8 10 10 11]]

[[10 10]
 [10 10]
 [10 10]]
---↑out---
```

- глубокая копия. Метод `copy()` создаст настоящую копию массива и его данных:

```
---in↓---
d = a.copy() # создается новый объект массива с новыми
данными
print (d is a)
print (d.base is a) # d не имеет ничего общего с a
d[0, 0] = 9999
```



```
print (a, "\n")
print (d, "\n")
---↑in_out↓---
False
False
[[ 0  10  10  3]
 [1234 10  10  7]
 [ 8  10  10 11]]

[[9999 10  10  3]
 [1234 10  10  7]
 [ 8  10  10 11]]
---↑out---
```

Маскированные массивы

Во многих случаях наборы данных могут быть неполными или испорченными наличием недопустимых данных. Например, датчику не удалось записать данные или он записал недопустимое значение. Модуль `numpy.ma` обеспечивает удобный способ решения этой проблемы, вводя такое понятие, как «маскированные массивы».

Маскированный массив представляет собой комбинацию стандартного массива `numpy.ndarray` и маски. Маской является массив логических значений, определяющий для каждого элемента связанного массива, является ли значение допустимым или нет. Если элемент маски является валидным (допустимым, правильным), то соответствующий элемент связанного массива является допустимым и считается размаскированным. Пакет гарантирует, что маскированные записи не будут использоваться в вычислениях. Ниже приведено несколько примеров работы с этим модулем.

```
---in↓---
import numpy as np
import numpy.ma as ma

#В качестве иллюстрации рассмотрим следующий набор данных:
x = np.array([1, 2, 3, -1, 5])
#выведем среднее арифметическое значений
print ('0)исходный массива', x)
print ('1)среднее исходного массива', x.mean())
```

```

#Допустим, мы хотим отметить четвертую запись как
недействительную.
# Для этого проще всего создать маскированный массив:
mx = ma.masked_array(x, mask=[0, 0, 0, 1, 0])
print ('2)маскированный массив',mx)

#Теперь мы можем вычислить среднее значение набора данных,
#не принимая во внимание недопустимые данные:
print ('3)среднее маскированного массива',mx.mean())

#маскировка элемента по индексу (-1 – это индекс первого
с конца элемента)
mx[-1] = ma.masked
print ('4)замаскировали еще и последний элемент',mx)

#маскировка элемента по значению с условием (значение >= 2)
mx = ma.masked_greater_equal(mx, 2)
print ('5)замаскировали все числа больше или равные 2',mx)
---↑in_out↓---
0)исходный массива [ 1  2  3 -1  5]
1)среднее исходного массива 2.0
2)маскированный массив [1 2 3 -- 5]
3)среднее маскированного массива 2.75
4)замаскировали еще и последний элемент [1 2 3 -- --]
5)замаскировали все числа больше или равные 2 [1 -- -- -- --]
---↑out---

```

Не обязательно использовать именно маскированные массивы при работе с реальными данными, содержащими ошибки. Это всего лишь один из удобных способов обработки некорректных значений.

Вы можете любым другим способом бороться с ошибками, например удалять некорректные значения из массива или заменять их на среднее значение, если у вас многомерные массивы и вы не хотите терять целую запись параметров из-за одного некорректного поля в ней (например, если у вас данные хранятся в парах «имя студента + оценка» и «имя студента» некорректно, то для расчета средней оценки по всем студентам вовсе не обязательно удалять всю пару, вы можете просто не использовать имя или заменить его на какой-нибудь текст, если вам все же нужно куда-либо его потом вывести для наличия как такового).

В целях проверки данных либо в любых других целях в NumPy существует множество неописанных здесь функций и методов, которые предлагается изучить самостоятельно на официальном сайте библиотеки (см. источник [21]).

Задания для проверки

1. Базовая функциональность NumPy. Опишите основные доступные функции.

2. Создание массива, заполненного нулями. Приведите пример кода.

3. Математические операции между массивами. Представьте код, где массив со значениями от 1 до 9 умножается на константу 2. Должна получиться таблица умножения для числа 2.

4. Слайсы (обрезка массива). Представьте код для четырех случаев:

- вывести все элементы, кроме первых трех;
- вывести все элементы, кроме последних трех;
- вывести все элементы, кроме первых и последних трех;
- слайсы с двумерными массивами.

5. Приведите пример объединения массивов из NumPy.

6. Приведите пример изменения формы массива из задания 3 (т.е. одномерный массив «таблица умножения двойки») на двумерный массив, где половина элементов в первом массиве, а вторая половина — во втором.

БИБЛИОТЕКА PANDAS²³

Цели выполнения работы

Получение представления о функциональности, доступных методах и объектах библиотеки pandas. Изучение основных принципов практической работы с ними.

²³ Разработано на основе русскоязычного источника [25] и официального руководства pandas [26].



Порядок выполнения работы

Библиотека pandas — это удобный и быстрый инструмент для работы с данными, обладающий большим функционалом (подробнее — на официальном сайте [26]). Pandas прекрасно подходит для работы с одномерными и двумерными таблицами данных, хорошо интегрирован с внешним миром — есть возможность работать с файлами CSV, таблицами Excel, может стыковаться с языком R.

Знакомство с библиотекой pandas

Библиотека pandas предоставляет две структуры: `Series` и `DataFrame` для быстрой и удобной работы с данными (на самом деле их три, есть еще одна структура — `Panel`, но в данный момент она находится в статусе устаревшей и в будущем будет исключена из состава библиотеки pandas).

`Series` — это маркированная одномерная структура данных, ее можно представить как таблицу с одной строкой. С `Series` можно работать двумя способами: как с обычным массивом (обращаться по номеру индекса), так и как с ассоциированным массивом, когда можно использовать ключ для доступа к элементам данных. `Series` способен хранить один любой тип данных (целые числа, строки, числа с плавающей запятой, объекты Python и т.д.).

`DataFrame` (с англ. — «фрейм данных» или «датафрейм») — это двумерная маркированная (размеченная) структура. Идейно она очень похожа на обычную таблицу, что выражается в способе ее создания и работе с ее элементами (очень удобно использовать, например, для размещения результата SQL-запроса, потому что он обычно тоже является электронной таблицей). Обычно это наиболее часто используемый объект pandas. Как и `Series`, `DataFrame` может хранить внутри себя много типов переменных: словари одномерных массивов, списков или серий (`Series`); NumPy-массивы; другие `DataFrame`. При создании датафрейма дополнительно вы можете установить метки (имена) строк и столбцов.

`Panel` представляет собой трехмерную структуру данных. О `Panel` мы больше говорить не будем, так как уже было упомянуто, что этот объект является устаревшим.

Структура данных Series

Для того чтобы начать работать со структурами данных из `pandas`, требуется предварительно импортировать необходимые модули. Убедитесь, что нужные модули установлены на вашем компьютере.

Помимо самого `pandas` нам понадобится библиотека `numpy`, которую мы изучали ранее. Импортируем нужные нам библиотеки:

```
---in↓---  
import numpy as np  
import pandas as pd  
---↑in_out↓---  
---↑out---
```

Конструктор класса `Series` выглядит следующим образом:

```
pandas.Series(data=None, index=None, dtype=None,  
name=None, copy=False, fastpath=False)
```

где представлены следующие принимаемые параметры:

- `data` — массив, словарь или скалярное значение, на базе которого будет построен `Series`;
- `index` — список меток, который будет использоваться для доступа к элементам `Series`. Длина списка должна быть равна длине `data`;
- `dtype` — объект `numpy.dtype`, определяющий тип данных;
- `copy`. Создает копию массива данных, если параметр равен `True`, в ином случае ничего не делает.

В большинстве случаев при создании `Series` используют только первые два параметра. Рассмотрим различные варианты, как это можно сделать.

Создание Series из списка Python

Самый простой способ создать `Series` — это передать только список Python в качестве единственного параметра:


```

---in↓---
s1 = pd.Series([1, 2, 3, 4, 5])
print(s1)
---↑in_out↓---
0      1
1      2
2      3
3      4
4      5
dtype: int64
---↑out---
```

В данном примере (представленном выше) была создана структура `Series` на базе списка. Для доступа к элементам `Series` в данном случае можно использовать только положительные целые числа — левый столбец чисел, начинающийся с нуля, это как раз и есть индексы элементов структуры, которые представлены в правом столбце.

Передадим в качестве второго элемента список строк (в нашем случае это отдельные символы), такой шаг позволит нам обращаться к элементам структуры `Series` не только по численному индексу, но и по метке, что сделает работу с таким объектом похожей на работу со словарем.

```

---in↓---
s2 = pd.Series([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd', 'e'])
print(s2)
---↑in_out↓---
a      1
b      2
c      3
d      4
e      5
dtype: int64
---↑out---
```

Обратите внимание на левый столбец, в нем содержатся метки, которые мы передали в качестве `index`-параметра при создании структуры. Правый столбец — это по-прежнему элементы нашей структуры.

Создание Series из ndarray массива из numpy

Объект `ndarray` представляет собой многомерный, однородный массив элементов фиксированного размера (можно назвать его «самый обычный NumPy-массив, с которым мы уже работали много раз»).

Создадим `Series` из пяти чисел на основе NumPy-массива, который аналогичен списку из предыдущего раздела:

```
---in↓---
#сначала создадим numpy-массив и выведем его тип
ndarr = np.array([1, 2, 3, 4, 5])
print ('тип исходного массива', type(ndarr))

#Теперь создадим Series с буквенными метками.
s3 = pd.Series(ndarr, ['a', 'b', 'c', 'd', 'e'])
print(s3)
print ('тип pandas массива', type(s3))
---↑in_out↓---
тип исходного массива <class 'numpy.ndarray'>
a      1
b      2
c      3
d      4
e      5
dtype: int32
тип pandas массива <class 'pandas.core.series.Series'>
---↑out---
```

Создание Series из словаря (dict)

Еще один способ создать структуру `Series` — это использовать словарь для одновременного задания меток и значений:

```
---in↓---
d = {'a':1, 'b':2, 'c':3}
s4 = pd.Series(d)
print(s4)
---↑in_out↓---
```

```
a      1
b      2
c      3
dtype: int64
---↑out---
```

Ключи (keys) из словаря `d` станут метками структуры `s4`, а значения (values) словаря — значениями в структуре. Пожалуй, это наиболее удобный способ создания `Series`.

Создание `Series` с использованием константы

Рассмотрим еще один способ создания структуры. На этот раз значения в ячейках структуры будут одинаковыми, т.е. равны константе `a`, значение которой 7.

```
---in↓---
a = 7
s5 = pd.Series(a, ['a', 'b', 'c'])
print(s5)
---↑in_out↓---
a      7
b      7
c      7
dtype: int64
---↑out---
```

Работа с элементами `Series`

При работе с элементами типа `Series` доступно:

- обращение по численному индексу, при таком подходе работа со структурой не отличается от работы со списками в Python;
- использование меток, тогда работа с `Series` будет похожа на работу со словарем в Python;
- получение слайсов (обрезки);
- работа как с векторами: сложение, умножение вектора на число и т.п.

Описанное выше представлено в примере:

```
---in↓---
s6 = pd.Series([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd', 'e'])

print ('0)Исходный Series s6\n', s6)
print ('1)Обращение по индексу 2(как работа со списком)
s6[2]\n', s6[2])
print ('2)Обращение по метке d (как работа со словарем)
s6['d']\n', s6['d'])
print ('3)Получение слайса \"только первые два элемента\"
s6[:2]\n', s6[:2])
print ('4)Фильтрация по условному выражению \"<=3\"
s6[s6 <= 3]\n', s6[s6 <= 3])
s7 = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
print ('5)Сложение s6 с вектором s7 [10, 20, 30, 40, 50]
s6 + s7\n', s6 + s7)
print ('6)Умножение s6 на константу 3 s6 * 3\n', s6 * 3)
---↑in_out↓---
0)Исходный Series s6
  a    1
  b    2
  c    3
  d    4
  e    5
dtype: int64
1)Обращение по индексу 2(как работа со списком) s6[2]
  3
2)Обращение по метке d (как работа со словарем) s6['d']
  4
3)Получение слайса \"только первые два элемента\" s6[:2]
  a    1
  b    2
dtype: int64
4)Фильтрация по условному выражению \"<=3\" s6[s6 <= 3]
  a    1
  b    2
  c    3
dtype: int64
5)Сложение s6 с вектором s7 [10, 20, 30, 40, 50] s6 + s7
  a    11
  b    22
```

```

c      33
d      44
e      55
dtype: int64
6) Умножение s6 на константу 3 s6 * 3
a       3
b       6
c       9
d      12
e      15
dtype: int64
---↑out---
```

где «\n» — символ перевода строки на новую, а «\» — экранирующий символ для использованных специальных символов (кавычек).

Структура данных DataFrame

Если `Series` представляет собой одномерную структуру, которую можно воспринимать как таблицу с одной строкой, то `DataFrame` — это уже двухмерная структура — полноценная таблица с множеством строк и столбцов.

Конструктор класса `DataFrame` выглядит так:

```
pandas.DataFrame(data=None, index=None, columns=None,
dtype=None, copy=False)
```

Он полностью аналогичен конструктору `Series`, но принимает на один параметр больше — `columns`. Этот параметр представляет собой список меток для полей (имена столбцов таблицы).

Структуру `DataFrame` можно создать на базе тех же объектов, из которых можно построить `Series`. Рассмотрим на практике различные подходы к созданию `DataFrame`.

Создание DataFrame из словаря

Создание `DataFrame` из словаря объектов типа `Series` показано в примере ниже (дополнительно выведены метки столбцов и строк):


```
---in↓---
d = {"price":pd.Series([1, 2, 3], index=['v1', 'v2', 'v3']),
     "count": pd.Series([10, 12, 7], index=['v1', 'v2', 'v3'])}
df1 = pd.DataFrame(d)

print(df1)
print(df1.index)
print(df1.columns)
---↑in_out↓---
   price  count
v1      1     10
v2      2     12
v3      3      7
Index(['v1', 'v2', 'v3'], dtype='object')
Index(['price', 'count'], dtype='object')
---↑out---
```

Теперь построим DataFrame на базе аналогичного словаря, но созданного из элементов ndarray:

```
---in↓---
d2 = {"price":np.array([1, 2, 3]), "count": np.array([10,
12, 7])}
df2 = pd.DataFrame(d2, index=['v1', 'v2', 'v3'])
print(df2)
print(df2.index)
print(df2.columns)
---↑in_out↓---
   price  count
v1      1     10
v2      2     12
v3      3      7
Index(['v1', 'v2', 'v3'], dtype='object')
Index(['price', 'count'], dtype='object')
---↑out---
```

Как видно, результат аналогичен предыдущему. Вместо ndarray можно использовать обычный список из Python.

Создание DataFrame из списка словарей

До этого мы создавали DataFrame из словаря, элементами которого были структуры Series, списки и массивы, сейчас мы создадим DataFrame из списка, элементами которого являются словари. В примере ниже также обратите внимание на использование метода `info()` из библиотеки pandas, который выводит информацию о датафрейме.

```

---in↓---
d3 = [{"price": 3, "count":8}, {"price": 4, "count": 11}]
df3 = pd.DataFrame(d3)
print(df3)
print(df3.info())
---↑in_out↓---
   price  count
0      3      8
1      4     11
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  -
 0  price    2 non-null      int64
 1  count    2 non-null      int64
dtypes: int64(2)
memory usage: 160.0 bytes
None
---↑out---
```

Создание DataFrame из многомерного массива

Создать DataFrame можно также и из многомерных массивов. Для примера пусть это будет двухмерный `ndarray` из библиотеки `numpy`:

```

---in↓---
nda1 = np.array([[1, 2, 3], [10, 20, 30]])
df4 = pd.DataFrame(nda1)
print(df4)
```

```
---↑in_out↓---
      0    1    2
0     1    2    3
1    10   20   30
---↑out---
```

Доступ к данным в структурах pandas

Основные методы для работы с объектами `DataFrame` представлены в таблице ниже (табл. 11).

Таблица 11

Основные методы работы с `DataFrame`

Операция	Синтаксис	Возвращаемый результат
Выбор столбца	<code>df[col]</code>	Series
Выбор строки по метке	<code>df.loc[label]</code>	Series
Выбор строки по индексу	<code>df.iloc[loc]</code>	Series
Слайс по строкам	<code>df[0:4]</code>	DataFrame
Выбор строк, отвечающих условию	<code>df[bool_vec]</code>	DataFrame

При работе со структурами `Series` и `DataFrame` библиотеки `pandas`, как правило, используют два основных способа получения значений элементов.

Первый способ основан на использовании меток, в этом случае работа ведется через метод `loc()`. Если вы обращаетесь к отсутствующей метке, то будет сгенерировано исключение `KeyError`.

Второй способ основан на применении целых чисел для доступа к данным, он предоставляется через метод `iloc()`. При использовании `iloc()`, если вы обращаетесь к несуществующему элементу, будет сгенерировано исключение `IndexError`.

Оба подхода позволяют использовать:

- метки в виде отдельных символов `['a']` или чисел `[5]`;
- список меток `['a', 'b', 'c']` или массивы целых чисел `[0, 1, 2]`;

- слайс меток ['a': 'c'] или целых чисел [1:4];
- условие выражения для фильтрации значений, например «a['x']==1» или «(a['x']==1) & (a['y']==10)»;
- вызываемую функцию с одним аргументом (будет рассмотрена позже).

В зависимости от типа используемой структуры будет вполне логичным образом меняться набор параметров для вызываемых методов, а именно, в случае с DataFrame нужно передавать два индекса, а в случае с Series — только один. Например для метода выборки строки по метке loc():

- для Series вызов выглядит так: «s.loc[индекс_строки]»;
- для DataFrame так: «df.loc[индекс_строки, индекс_колонки]».

Ниже представлен пример программной реализации описанных в таблице методов для обращения к данным библиотеки pandas:

```

---in↓---
#Для начала создадим DataFrame.
d = {"price":np.array([1, 2, 3]), "count": np.array([10,
20, 30])}
df = pd.DataFrame(d, index=['a', 'b', 'c'])

print('1)Исходный датафрейм df\n', df)
print('2)Операция df[\count\']: выбор столбца по метке\n',
df['count'])
print('3)Операция df.loc[\a\']: выбор строки по метке\n',
df.loc['a'])
print('4)Операция df.iloc[1]: выбор строки по индексу\n',
df.iloc[1])
print('5)Операция df[0:2]: slice по строкам\n', df[0:2])
print('6)Операция df[df[\count\'] >= 20]: выбор строк,
отвечающих условию\n', df[df['count'] >= 20])
print('7)Операция: выбор count из первой найденной строки,\
отвечающей составному условию. df[ (df[\count\'] >= 20)
& (df[\price\']>2.5)][\count\'][0]\n', df[ (df['count']
>= 20) & (df['price']>2.5)][count'][0])
---↑in_out↓---

```

1) Исходный датафрейм df

	price	count
a	1	10
b	2	20
c	3	30

2) Операция df['count']: выбор столбца по метке

a	10
b	20
c	30

Name: count, dtype: int32

3) Операция df.loc['a']: выбор строки по метке

price	1
count	10

Name: a, dtype: int32

4) Операция df.iloc[1]: выбор строки по индексу

price	2
count	20

Name: b, dtype: int32

5) Операция df[0:2]: slice по строкам

	price	count
a	1	10
b	2	20

6) Операция df[df['count'] >= 20]: выбор строк, отвечающих условию

	price	count
b	2	20
c	3	30

7) Операция: выбор count из первой найденной строки, отвечающей составному условию. df[(df['count'] >= 20) & (df['price'] > 2.5)]['count'][0]

30

---↑out---

Что касается примеров 6 и 7 из кода выше, отметим следующее: на практике очень часто приходится получать определенную отфильтрованную подвыборку из существующего набора данных. Например, нужно получить все товары, скидка на которые больше пяти процентов, или выбрать из базы информацию о сотрудниках мужского пола старше 30 лет. Это очень похоже на процесс фильтрации при работе с базами данных. В данном издании еще будут рассмотрены аналогичные действия при помощи языка SQL.

Обращение через вызываемую функцию

Существует подход для обращения к элементам pandas-объекта через callable (с англ. — «вызываемая») функцию. Внешне это очень похоже на работу с условными выражениями и используется с той же целью — для фильтрации значений по какому-либо условию.

При таком подходе в квадратных скобках указывается не индекс или метка, а функция (очень часто указывается короткая лямбда-функция (понятие «лямбда-функция» рассматривалось на стр. 60), которая и используется для выборки элементов.

Пример получения всех строк, где цена больше или равна 2, а количество больше 25, представлен ниже:

```

---in↓---
#map - применяет к каждому элементу функцию
#в данном случае мы используем лямбда-функцию
print (
df [
    df["price"].map(lambda price: price >= 2)
    &
    df["count"].map(lambda count: count > 25)
]
)
---↑in_out↓---
   price  count
c      3     30
---↑out---
```

Использование атрибутов для доступа к данным

Для доступа к данным можно использовать атрибуты структур, в качестве которых выступают метки. Начнем со структуры `Series`. Для доступа к элементу через атрибут необходимо указать его через точку после имени структуры:

```

---in↓---
s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
print (s, '\n')
print (s.a)
print (s.b)
```

```
---↑in_out↓---
```

```
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

```
10
```

```
20
```

```
---↑out---
```

Так как структура `s` имеет метки `'a'`, `'b'`, `'c'`, `'d'`, `'e'`, то для доступа к элементу с меткой `'a'` мы можем использовать синтаксис `s.a`. Этот же подход можно применить для объекта типа `DataFrame`. Например, для доступа к столбцу `'price'`:

```
---in↓---
```

```
d = {"price": [1, 2, 3], "count": [10, 20, 30], "percent":
[24, 51, 71]}
df = pd.DataFrame(d, index=['a', 'b', 'c'])
print (df, '\n')
print (df.price)
```

```
---↑in_out↓---
```

```
price  count  percent
a         1     10      24
b         2     20     51
c         3     30     71
```

```
a     1
```

```
b     2
```

```
c     3
```

```
Name: price, dtype: int64
```

```
---↑out---
```

Получение случайного набора из структур pandas

Библиотека `pandas` предоставляет возможность получить случайный набор данных из существующей структуры. Такой функционал доступен как для `Series`, так и для `DataFrame`, и реали-

зуется при помощи метода `sample()`, который и предоставляет случайную выборку.

Для того чтобы выбрать случайным образом один, `n` или некоторую долю элементов, можно воспользоваться следующим синтаксисом (на примере `Series`):

```

---in↓---
s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
print('0)Исходный series s\n', s)
print('1)выбрать 1 случайное значение s.sample()\n',
s.sample())
print('2)выбрать 3 случайных значений s.sample(n=3)\n',
s.sample(n=3))
print('3)выбрать долю от общего числа объектов
в структуре s.sample(frac=0.3)\n', s.sample(frac=0.3))
---↑in_out↓---
0)Исходный series s
  a    10
  b    20
  c    30
  d    40
  e    50
dtype: int64
1)выбрать 1 случайное значение s.sample()
  b    20
dtype: int64
2)выбрать 3 случайных значений s.sample(n=3)
  e    50
  c    30
  a    10
dtype: int64
3)выбрать долю от общего числа объектов в структуре
s.sample(frac=0.3)
  b    20
  e    50
dtype: int64
---↑out---
```

Очень интересной особенностью является то, что мы можем передать вектор весов вероятностей выборки, длина которого

должна быть равна количеству элементов в структуре. Сумма весов должна быть равна единице, а вес в данном случае — это вероятность появления элемента в выборке.

Давайте сформируем вектор вероятностей для строк DataFrame и сделаем выборку из трех элементов:

```
---in↓---
d = {"price": [1, 2, 3, 5, 6], "count": [10, 20, 30, 40, 50], "percent": [24, 51, 71, 25, 42]}
df = pd.DataFrame(d)
w = [0.1, 0.2, 0.5, 0.1, 0.1]

print ('0) Исходный датафрейм df\n', df)
print ('1) Выборка трех строк на основе весов \n\
df.sample(n = 3, weights=[0.1, 0.2, 0.5, 0.1, 0.1])\n',
df.sample(n = 3, weights=w))
---↑in_out↓---
0) Исходный датафрейм df
   price  count  percent
0      1     10      24
1      2     20      51
2      3     30      71
3      5     40      25
4      6     50      42
1) Выборка трех строк на основе весов
df.sample(n = 3, weights=[0.1, 0.2, 0.5, 0.1, 0.1])
   price  count  percent
1      2     20      51
2      3     30      71
3      5     40      25
---↑out---
```

При работе с DataFrame можно указывать ось (ось 0 — строки, ось 1 — столбцы):

```
---in↓---
print ('0) df.sample(axis=1)\n', df.sample(axis=1))
print ('1) df.sample(n=2, axis=0)\n', df.sample(n=2,
axis=0))
---↑in_out↓---
```

```

0)df.sample(axis=1)
   percent
0         24
1         51
2         71
3         25
4         42
1) df.sample(n=2, axis=0)
   price  count  percent
3        5     40       25
4        6     50       42
---↑out---
```

Добавление элементов в структуры

Увеличение размера структуры, т.е. добавление новых, дополнительных, элементов, — это довольно распространенная задача. Добавление нового элемента в `Series` показано ниже:

```

---in↓---
s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
print ('0)Исходный Series s\n', s)
s['f'] = 60
print ('1) Добавили s[\f] = 60\n', s)
---↑in_out↓---
```

0)Исходный Series s

a	10
b	20
c	30
d	40
e	50

dtype: int64

1) Добавили s['f'] = 60

a	10
b	20
c	30
d	40
e	50
f	60

dtype: int64

```

---↑out---
```


Добавлять столбцы и строки в структуру DataFrame можно по новой метке/индексу (ниже продемонстрированы различные варианты). Для удаления строк или столбцов существует метод `drop()`:

```
---in↓---
d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent":
[24, 51, 71]}
df = pd.DataFrame(d)

print ('0)Исходный DataFrame df\n', df)

#добавление новой колонки
df['value'] = [3, 14, 7]
print ('1) Добавили столбец value\n', df)

#добавление новой колонки
my_column = pd.DataFrame({'new_column': [111, 222, 333]})
df=df.assign(new_column =my_column)
print ('2) Добавили столбец new_column\n', df)

#добавление новой строки
df.loc[3] = ['11', '22', '33', '44', np.NaN]
print ('3) Добавили строку 11 22 33 44 NaN\n', df)

#добавление новой строки
my_row = {'price': '1', 'count' : '2', 'percent' : '3'}
#ignore_index - чтобы продолжилась индексация т.е.
присвоится 5-й индекс
df=df.append(my_row, ignore_index=True)
print ('4) Добавили строку 1 2 3 NaN NaN\n', df)

#удаление по осям
df=df.drop(['value', 'new_column'], axis=1)
print ('5) Удалили колонки value и new_column\n', df)
---↑in_out↓---
```

0)Исходный DataFrame df

	price	count	percent
0	1	10	24
1	2	20	51
2	3	30	71

```

1) Добавили столбец value
   price  count  percent  value
0       1     10       24      3
1       2     20       51     14
2       3     30       71      7
2) Добавили столбец new_column
   price  count  percent  value  new_column
0       1     10       24      3         111
1       2     20       51     14         222
2       3     30       71      7         333
3) Добавили строку 11 22 33 44 NaN
   price  count  percent  value  new_column
0       1     10       24      3         111.0
1       2     20       51     14         222.0
2       3     30       71      7         333.0
3      11     22       33     44           NaN
4) Добавили строку 1 2 3 NaN NaN
   price  count  percent  value  new_column
0       1     10       24      3         111.0
1       2     20       51     14         222.0
2       3     30       71      7         333.0
3      11     22       33     44           NaN
4       1      2        3     NaN           NaN
5) Удалили колонки value и new_column
   price  count  percent
0       1     10       24
1       2     20       51
2       3     30       71
3      11     22       33
4       1      2        3
---↑out---
```

Вывод значений на основе таблицы True/False

Если вы построите структуру той же размерности, что и ваша структура с данными, но вместо данных будут значения `True/False`, то вы можете использовать эту структуру для маскировки значений в исходной структуре (почти так, как мы делали с NumPy, только тут мы не прячем значения, а заменяем на NaN).

Конечно, это всего лишь подобие маскировки в сравнении с полноценной маскировкой в NumPy, потому что помеченные как NaN значения все равно будут существовать и использоваться при вычислениях.

Рассмотрим пример. В pandas есть метод `isin()`, который проверяет, содержится ли определенное значение или значения в элементах нашей структуры. В примере ниже, если элемент имеет значение, равное любому значению из 1, 2, 3, 20, то в представлении на месте этого элемента записывается True, а если значение другое, то False.

```
---in↓---
df = pd.DataFrame({"price": [1, 2, 3], "count": [10, 20, 30],
                  "percent": [24, 51, 71]})
print ('0)Исходный DataFrame df\n', df)
print ('1)df.isin([1, 2, 3, 20])\n', df.isin([1, 2, 3, 20]))
print ('2)df [df.isin([1, 2, 3, 20])]\n', df
      [df.isin([1, 2, 3, 20])])
---↑in_out↓---
0)Исходный DataFrame df
   price  count  percent
0      1     10      24
1      2     20      51
2      3     30      71
1)df.isin([1, 2, 3, 20])
   price  count  percent
0   True   False   False
1   True   True   False
2   True   False   False
2)df [df.isin([1, 2, 3, 20])]
   price  count  percent
0      1    NaN     NaN
1      2   20.0     NaN
2      3    NaN     NaN
---↑out---
```

Тем самым в п. 2 примера мы получили новую pandas-структуру, точно такого же размера, как и исходная. Используя ее (см.

п. 3 в примере), мы можем вывести отфильтрованные значения, а те значения, которые не прошли фильтрацию, будут выведены как NaN (т.е. пустые).

Для решения этой же задачи более простым путем можно использовать готовый pandas-метод `mask()`. Он может не только не прошедшие фильтрацию значения устанавливать в NaN, но и присвоить им какое-либо конкретное значение.

```

---in↓---
print ('0)Исходный DataFrame df\n', df)
print ('1)df.mask(df['count'] > 21, 'Не подходит')\n',
df.mask( df['count'] > 21, 'Не подходит'))
---↑in_out↓---
0)Исходный DataFrame df
   price  count  percent
0      1    10      24
1      2    20      51
2      3    30      71
1)df.mask(df['count'] > 21, 'Не подходит')
   price  count  percent
0      1    10      24
1      2    20      51
2  Не подходит  Не подходит  Не подходит
---↑out---
```

Pandas и отсутствующие данные

Обратим внимание, что в документации по библиотеке pandas есть целый раздел, посвященный данной тематике. Для наших экспериментов создадим структуру DataFrame, которая будет содержать пропуски.

Сначала импортируем необходимые нам библиотеки:

```

---in↓---
import pandas as pd
from io import StringIO
---↑in_out↓---
---↑out---
```

Модуль `StringIO` позволяет работать со строкой как с файловым объектом. Все операции с файловым объектом производятся в оперативной памяти (т.е. значительно быстрее, чем при работе с реальным файлом, расположенном на накопителе). Этот модуль удобно использовать, когда наш файл виртуальный, т.е. хранится только в оперативной памяти. Не всегда удобно работать с записанными на накопитель файлами, так как, например, это часто вызывает проблемы при многопоточной обработке файла.

CSV (англ. *Comma-Separated Values* — «значения, разделенные запятыми») — это один из наиболее простых и распространенных форматов хранения данных, в котором элементы отделяются друг от друга запятыми. Очень часто различные наборы данных мы получаем именно в CSV-формате (в него можно экспортировать данные из Excel, баз данных и почти откуда угодно, потому что это один из самых простых форматов). В том числе многие наборы данных, которые можно скачать в сети «Интернет» в целях обучения, представлены именно в этом формате.

Pandas «умеет» считывать содержимое из реального csv-файла и записывать его в `DataFrame`. Мы бы могли просто создать файл на накопителе через обычный блокнот, а затем открыть его при помощи `pandas`, но в целях разнообразия и повышения кругозора читателя данного издания это сделано именно с использованием виртуального файла. Если бы мы использовали реальный файл, то он бы выглядел представленным на рисунке ниже способом (рис. 19).

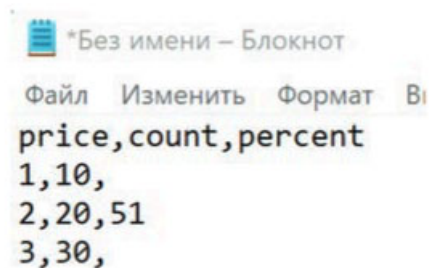


Рис. 19. Создание файла в блокноте Windows

Сначала мы создадим строку, передадим ее в функцию `StringIO()`, которая, в свою очередь, вернет виртуальный файл, который далее сможет открыть `pandas`. В итоге в переменной `df`

мы имеем все значения, которые были прочитаны из нашего виртуального файла:

```

---in↓---
data = 'price,count,percent\n1,10,\n2,20,51\n3,30, '
df = pd.read_csv(StringIO(data))
print (df)
---↑in_out↓---
   price  count  percent
0      1     10      NaN
1      2     20     51.0
2      3     30      NaN
---↑out---
```

Полученный объект `df` — это `DataFrame` с пропусками. В нашем примере у объектов с индексами 0 и 2 отсутствуют данные в поле `percent`. Отсутствующие данные помечаются как `NaN`. Добавим к существующей структуре еще один объект (запись), у которого будет отсутствовать значение в поле `count` (англ. *none* — «ничто», т.е. пустое значение).

`NaN` — это тип для отсутствующих числовых данных из библиотеки `NumPy` (т.е. не является стандартным, как `float` или `int`), на основе которой и работает `pandas`. В отличие от `None` (является стандартным для `Python`), `NaN` — это число, хоть и без установленного значения. `NaN` — это математический тип пустоты. Думайте о нем так же, как о понятии «бесконечность»: она, как и `NaN`, «теоретическое понятие» в математике. А `None` — это любое пустое значение, не обязательно число.

```

---in↓---
df.loc[3] = {'price':4, 'count':None, 'percent':26.3}
print (df)
---↑in_out↓---
   price  count  percent
0      1     10      NaN
1      2     20     51.0
2      3     30      NaN
3      4    None     26.3
---↑out---
```

Мы могли вместо «'count':None» написать «'count':np.NaN» для красоты и однородности, и это было бы более правильно, и все работало бы точно так же, но тогда бы вы, возможно, никогда не узнали про существование типа None.

Для начала обратимся к методу `isnull()` из библиотеки `pandas`, который позволяет быстро определить наличие элементов NaN в структурах:

```
---in↓---
pd.isnull(df)
---↑in_out↓---
price count percent
0      False False True
1      False False False
2      False False True
3      False True      False
---↑out---
```

Таким образом мы получаем таблицу того же размера, но на месте реальных данных в ней находятся логические переменные, которые принимают значение `False`, если значение поля у объекта есть, или `True`, если значение в данном поле отсутствует. Как видим, метод `isnull()` нам пометил все пустоты как `True`.

Дополнительно можно посмотреть подробную информацию об объекте (датафрейме), для этого можно воспользоваться методом `info()`.

```
---in↓---
df.info()
---↑in_out↓---
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4 entries, 0 to 3
Data columns (total 3 columns):
#   Column      Non-Null Count  Dtype
---  -
0   price       4 non-null      int64
1   count       3 non-null      object
2   percent     2 non-null      float64
```

```
dtypes: float64(1), int64(1), object(1)
memory usage: 128.0+ bytes
---↑out---
```

В нашем примере видно, что объект `df` имеет три столбца (`count`, `percent` и `price`), при этом в столбце `price` все объекты значимы, т.е. не `NaN`, в столбце `count` — один пустой объект, в поле `percent` — два пустых объекта.

Обратите внимание, что тип (`Dtype`) столбца `count` показывается как `object`. Это потому, что мы при добавлении одной новой строки использовали `None` вместо `pr.NaN`, и из-за этой одной строки весь столбец принял «обобщенный» тип «объект» («`object`»). Если бы мы написали именно «математическую пустоту» `pr.NaN`, то тип отображался бы корректно. Было бы более правильно написать `pr.NaN`.

Можно воспользоваться следующим подходом для получения количества `NaN`-элементов в записях:

```
---in↓---
df.isnull().sum()
---↑in_out↓---
price      0
count      1
percent    2
dtype: int64
---↑out---
```

Замена отсутствующих данных

Отсутствующие данные объектов можно заменить на конкретные числовые значения, для этого можно использовать метод `fillna()`. Для экспериментов будем использовать структуру `df`, созданную ранее.

```
---in↓---
print (df)
print (df.fillna(0))
---↑in_out↓---
```

```
      price count percent
0         1    10      NaN
1         2    20     51.0
2         3    30      NaN
3         4   None     26.3
      price count percent
0         1    10       0.0
1         2    20     51.0
2         3    30       0.0
3         4     0     26.3
---↑out---
```

Этот метод по умолчанию не изменяет исходную структуру, он возвращает структуру `DataFrame`, созданную на базе существующей, с заменой пустых значений на те, что переданы в метод в качестве аргумента.

Не всегда целесообразно заменять отсутствующие данные нулями, как показано выше. Иногда недостающие данные можно заполнить средним значением по столбцу:

```
---in↓---
df=df.fillna(df.mean())
print (df)
---↑in_out↓---
      price count percent
0         1   10.0     38.65
1         2   20.0     51.00
2         3   30.0     38.65
3         4   20.0     26.30
---↑out---
```

В зависимости от задачи применяется тот или иной метод заполнения отсутствующих элементов, это может быть нулевое значение, математическое ожидание, медиана и т.п. Для замены `NaN`-элементов на конкретные значения можно использовать интерполяцию, которая реализована в методе `interpolate()`, и другие доступные методы.

Удаление объектов/столбцов с отсутствующими данными

Довольно часто используемый подход при работе с отсутствующими данными — это удаление записей (строк) или полей (столбцов), в которых встречаются пропуски. Для того чтобы удалить все объекты, которые содержат значения NaN, воспользуйтесь методом `dropna()` без аргументов:

```
---in↓---
df.dropna()
---↑in_out↓---
price      count  percent
1         2      20      51.0
---↑out---
```

Вместо записей можно удалить поля, для этого нужно вызвать метод `dropna()` с аргументом `axis=1`:

```
---in↓---
df.dropna(axis=1)
---↑in_out↓---
price
0      1
1      2
2      3
3      4
---↑out---
```

Pandas позволяет задать порог на количество не-NaN-элементов. В приведенном ниже примере будут удалены все столбцы, в которых количество не-NaN-элементов меньше трех:

```
---in↓---
df.dropna(axis = 1, thresh=3)
---↑in_out↓---
price      count
0         1      10
1         2      20
```



```
2      3      30
3      4      None
---↑out---
```

? Задания для проверки

1. Pandas. Основные сведения, сравнение с функциональностью NumPy.
2. Приведите пример `Series` и попробуйте: выводить на экран, добавлять элементы, изменять значения.
3. Повторите п. 2 для объекта `DataFrame`.
4. Попробуйте открыть при помощи `pandas` реальный CSV-файл с пропусками или некорректными значениями и исправить их.

СТАТИСТИКА В PYTHON

СТАТИСТИКА В NUMPY И В STATISTICS. ВВЕДЕНИЕ В SCIPY.STATS²⁴

Цели выполнения работы

Получение представления о функциональности, доступных методах и объектов статистики библиотек/модулей NumPy, Statistics и Scipy.Stats. Изучение основных принципов практической работы с ними.

Порядок выполнения работы

Статистические функции в NumPy

NumPy имеет набор из следующих статистических функций:

- 1) `np.amin()` — определяет минимальное значение элемента вдоль указанной оси;
- 2) `np.amax()` — определяет максимальное значение элемента вдоль оси;
- 3) `np.mean()` — определяет среднее значение набора данных;
- 4) `np.median()` — определяет медианное значение набора данных;

²⁴ Разработано на основе англоязычного источника [27], официальной документации Python [16], официального руководства Scipy [28], matplotlib [29] и plotly [30].

5) `np.std()` — определяет стандартное отклонение;
6) `np.var()` — определяет дисперсию;
7) `np.ptp()` — возвращает диапазон значений по оси;
8) `np.average()` — определяет средневзвешенное значение;
9) `np.percentile()` — определяет n-й процентиль (или персентиль, в разных источниках данный термин называется по-разному) данных по указанной оси.

Далее рассмотрим примеры работы с этими функциями.

Максимум и минимум массива

Пример определения минимального и максимального значения элементов массива вдоль указанной оси при помощи методов `np.amin()` и `np.amax()` показан ниже:

```
---in↓---
import numpy as np

arr= np.array([[1,23,78],[98,60,75],[79,25,48]])
print(arr)
print(np.amin(arr)) # Minimum Function
print(np.amax(arr)) # Maximum Function
---↑in_out↓---
[[ 1 23 78]
 [98 60 75]
 [79 25 48]]
1
98
---↑out---
```

Среднее арифметическое

Среднее арифметическое (или обычно говорят кратко — «среднее») — это сумма элементов, деленная на их сумму и вычисляемая по следующей формуле (формула (1)):

$$\bar{x} = \frac{1}{n} (x_1 + x_2 + \dots + x_n). \quad (1)$$

Метод `np.mean()` вычисляет среднее значение, складывая все элементы массивов, а затем делит его на количество элементов. Мы также можем указать ось, по которой может быть вычислено среднее значение. Пример вычисления среднего арифметического значения представлен ниже.

```

---in↓---
a = np.array([5, 6, 7])
print(a)
print(np.mean(a))
---↑in_out↓---
[5 6 7]
6.0
---↑out---
```

Медиана

Медиана (с лат. «середина») — середина множества чисел. Формула отличается для множеств с четным и нечетным числом элементов. Формула (2) — формула медианы для массивов с четным числом элементов, формула (3) — для нечетных.

$$M_e = \frac{n + 1}{2}; \quad (2)$$

$$M_e = \frac{n}{2}, \quad (3)$$

где M_e — индекс медианного значения в ранжированном (сортированном) по возрастанию массиве; n — количество элементов в этом массиве.

Рассмотрим пример с заранее отсортированным в порядке возрастания (проранжированном) массивом (вам не нужно его вручную сортировать, `np.median()` сама все отсортирует и найдет медиану, просто сейчас мы взяли уже отсортированный массив, чтобы продемонстрировать, как вручную считается медианное значение). Итак, рассмотрим массив (с четным числом индивидуальных значений), представленный в таблице ниже (табл. 12).

Таблица 12

Пример массива для определения медианы

x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12
27.90	27.95	28.00	28.00	28.05	28.05	28.10	28.10	28.10	28.15	28.20	28.25

Индекс медианы $M_e = (12 + 1)/2 = 6.5$. То есть нужно найти значение числа с индексом 6.5, но индексы бывают только четные, поэтому определяется среднее арифметическое чисел с индексом 6 и 7: $(28.05 + 28.10)/2 = 28.075$. Итого: медианное значение (медиана) равно значению 28.075.

Для массива с нечетным числом (например, этого же массива, но без значения x12) элементов мы бы взяли значение с индексом 6 как значение медианы. То есть медианой было бы значение 28.05.

Рассмотрим программную реализацию вычисления медианы при помощи `np.median()`. Данная функция вычисляет медиану как для одномерных, так и для многомерных массивов. При помощи медианы можно разделять верхний и нижний диапазон значений данных (относительно срединного значения), если взять ее индекс.

```

---in↓---
print(a)
print(np.median(a))
---↑in_out↓---
[5 6 7]
6.0
---↑out---
```

Среднеквадратичное (стандартное) отклонение

Среднеквадратическое отклонение (формула (4), также называется стандартным отклонением) определяется как квадратный корень из дисперсии случайной величины (формула (5)):

$$\sigma = \sqrt{D[x]}; \quad (4)$$

$$D[x] = \frac{\sum |x - \bar{x}|^2}{n}, \quad (5)$$

где: $D[x]$ — дисперсия случайной величины, т.е. мера разброса значений случайной величины (разброс); σ — стандартное отклонение.

Ниже представлена программная реализация, в которой при помощи метода `np.std()` вычисляется стандартное отклонение (*std* — от англ. *standart* — «стандартное»).

```
---in↓---
print(a)
print(np.std(a))
---↑in_out↓---
[5 6 7]
0.816496580927726
---↑out---
```

В вышестоящем коде мы получили стандартное отклонение, равное 0.82, теперь мы рассмотрим, как оно вычисляется. Для этого вы можете выполнить представленный ниже код в ячейке Jupyter или посчитать представленную в нем математику самостоятельно.

```
---in↓---
(((5-6)**2+(6-6)**2+(7-6)**2 #Разброс от 5 до 7
)/a.size # делить на кол-во элементов 3
)**0.5 # корень квадратный
---↑in_out↓---
0.816496580927726
---↑out---
```

Как видно, результат совпадает с посчитанными при помощи метода `np.std()`. Также существует метод `np.var()`, вычисляющий разброс (т.е. дисперсию). Вычисление дисперсии представлено ниже.

```
---in↓---
print(a)
print(np.var(a))
```

```
---↑in_out↓---  
[5 6 7]  
0.6666666666666666  
---↑out---
```

Дисперсия получилась равной 0.66, что соответствует формуле (4), в которой говорится, что стандартное отклонение — это квадратный корень из дисперсии, т.е. стандартное отклонение 0.81, возведенное в квадрат, равно дисперсии, т.е. 0.66.

Среднее взвешенное

Функция `np.average()` определяет средневзвешенное значение и может использоваться для многомерных массивов. Средневзвешенное значение рассчитывается путем умножения компонента на его вес, веса указываются отдельно. Если веса не указаны, результат будет таким же, как и среднее значение (среднее арифметическое). Ниже представлен пример вычисления среднего взвешенного при помощи данной функции.

```
---in↓---  
print(a)  
print(np.average(a)) # без веса то же, что и среднее  
  
wt = np.array([8,2,3]) # массив весов  
print(np.average(a, weights=wt)) # средневзвешенное значение  
---↑in_out↓---  
[5 6 7]  
6.0  
5.615384615384615  
---↑out---
```

Процентили

Медиана представляет собой среднюю точку распределения: половина наблюдений расположена над ней, а другая половина — под ней (например, медиана чисел [3, 4, 5, 6, 102] составляет 5). Термин «медиана» был рассмотрен ранее, и его суть пригодится для понимания схожего понятия «процентиль».

Процентиль (персентиль) — это такое значение x_p из массива (или в терминологии статистики это называется совокупностью значений), что значения p -й части совокупности меньше или равны этому значению x_p . Мы рассмотрим, что это означает, на примере программного кода ниже, но сначала изучим синтаксис функции вычисления процентилей:

```
numpy.percentile (данные, q, ось),
```

где принимаются следующие параметры: данные — входной массив; q — процентиль (от 0 до 100), который нужно вычислить для элементов массива; ось — указание оси, по которой выполняется расчет.

Ниже представлен пример вычисления 67-го процентиля для набора данных. В данном случае 67-й процентиль определяет позицию некоторого «порогового» значения в ранжированном (отсортированном) по возрастанию ряде значений, которое нас интересует. Посмотрим на код, представленный ниже.

```
---in↓---
a = np.array([2, 5, 20, 20])
print(a)
print(np.percentile(a, 67, 0))
---↑in_out↓---
[ 2  5 20 20]
20.0
---↑out---
```

Мы получили значение 67-го процентиля, равное 20. Это означает, что 33 % (одна треть) всех значений имеют значение выше 20.

Мы можем представить это графически (рис. 20) при помощи кода ниже:

```
---in↓---
import matplotlib.pyplot as plt

percentel_arr= np.percentile(a, list(range(101)))

plt.plot(percentel_arr, color="r")
plt.show()
print (percentel_arr)
```

```
---↑in_out↓---  
---в выводе выводится график, представленный на рис. 20---  
[ 2.    2.09  2.18  2.27  2.36  2.45  2.54  2.63  2.72  2.99  
#часть процентилей вырезана  
20.    20.    20.  ]  
---↑out---
```

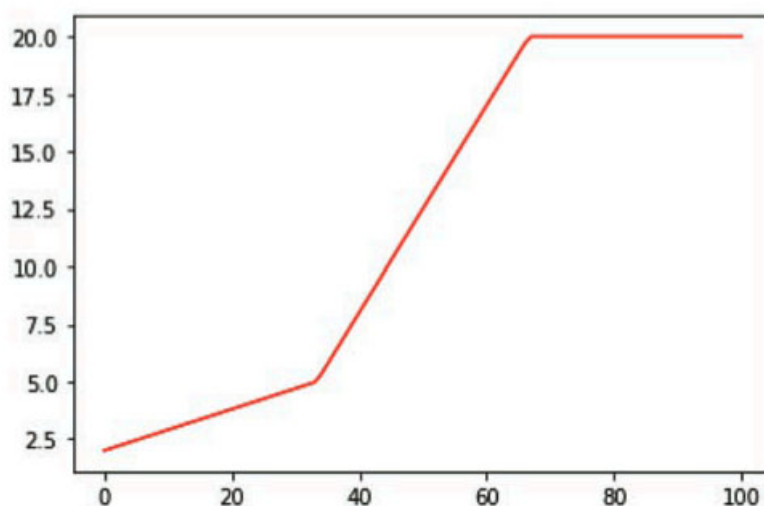


Рис. 20. График процентилей и значений

Из графика процентилей и значений (рис. 20) видно, что нулевой процентиль равен 2, это значит, что 0 % от количества значений имеют значение меньше двух; а 100-й процентиль равен 20, это означает, что все 100 % значений ниже или равны значению 20.

Стоит отметить, что значение 50-го процентиля равно медианному значению (медианы). Также существует такое понятие, как «квартиль». Квартиль — это процентиль, который делит ряд значений на четыре равные части. Например: первый квартиль (25 %) — это 25-й процентиль, а второй квартиль — это медиана или 50-й процентиль. Квартили часто используются в бизнес-аналитике и всего лишь отражают значения четырех процентилей.

Полный размах

Функция `np.ptp()` полезна для определения размаха значений вдоль оси. Ниже представлен пример.

```

---in↓---
a = np.array([[2,10,20],[6,10,60]])
print(np.ptp(a,0)) # пример 1

b = np.array([2,10,20,5])
print(np.ptp(b,0)) # пример 2

b = np.array([[2,10,20,5]])
print(np.ptp(b,1)) # пример 3
---↑in_out↓---
[ 4  0 40]
18
[18]
---↑out---

```

Как показано в примере, можно находить либо размах между элементами одномерных массивов многомерного массива (пример 1), одномерного массива (пример 2, у одномерных массивов всего одна ось, и она нулевая), либо размах между максимальным и минимальным значениями в конкретном одномерном массиве многомерного массива (пример 3).

Некоторые статистические функции из Statistics

Гармоническое среднее

Гармоническое среднее является обратным средним значением обратных величин всех элементов в наборе данных и определяется по следующей формуле (формула (6)):

$$\bar{x}_{\text{гармонич}} = \sum \frac{1}{x_i}, \quad (6)$$

где $\bar{x}_{\text{гармонич}}$ — гармоническое среднее; $i = [1, 2, \dots, n]$ и n — количество элементов в наборе данных x .

Один из вариантов реализации гармонического среднего на «чистом» Python (т.е. без использования библиотек):


```
---in↓---
import numpy as np
a = np.array([5,6,7])

print(a)
print("Среднее:", np.average(a))
print("Медиана:", np.median(a))

my_sum=0
for item in a:
    my_sum = my_sum + (1 / item)
    print (1 / item)
hmean = len(a)/my_sum
print("Гармоническое Среднее:", hmean)
---↑in_out↓---
[5 6 7]
Среднее: 6.0
Медиана: 6.0
0.2
0.16666666666666666
0.14285714285714285
Гармоническое Среднее: 5.88785046728972
---↑out---
```

Вы также можете рассчитать этот показатель с помощью стандартного модуля Python statistics:

```
---in↓---
import statistics
statistics.harmonic_mean(a)
---↑in_out↓---
5.88785046728972
---↑out---
```

Давайте рассмотрим пример, демонстрирующий применение гармонического среднего.

Первые 100 км автомобиль проехал со скоростью 50 км/ч, а следующие 100 км — со скоростью 80 км/ч. С какой средней скоростью двигался автомобиль на всем пути?

Сначала, наверное, может показаться, что правильное значение 65 км/ч, потому что $(50 + 80) / 2 = 65$. Однако быстро становится понятно, что если бы другой автомобиль двигался со средней скоростью, то он провел бы в пути столько же времени, сколько и первый. Именно в этом и заключается смысл усреднения в данном случае. В этом примере нам на помощь приходит среднее гармоническое.

Для нашей задачи искомое среднее равно (формула (7)):

$$\frac{2}{\frac{1}{50} + \frac{1}{80}} = 61.54 \text{ км/ч.} \quad (7)$$

И действительно, в первом случае автомобиль затратил 2 ч на преодоление 100 км со скоростью 50 км/ч, и еще 1.25 ч ему потребовалось на следующие 100 км, потому что скорость возросла до 80 км/ч. Таким образом, всего ушло 3.25 ч. Если бы автомобиль все 200 км двигался со скоростью 61.54 км/ч, то у него также ушло бы на дорогу 3.25 ч.

Близость значений 65 (арифметического среднего) и 61.54 (гармонического среднего) не должна вас обманывать. Среднее гармоническое в данном случае не просто дает более точный результат. Это единственно правильный способ усреднения, потому что он соответствует физическому смыслу измеряемых явлений. При других исходных данных разница между средним гармоническим и средним арифметическим могла бы быть больше. Но среднее арифметическое здесь не имеет никакого смысла. Для усреднения в подобных задачах допустимо использовать только среднее гармоническое, и в примере ниже демонстрируется, что полученная путем арифметического среднего скорость соответствует суммарному времени в пути 3.07 ч, хотя по условию задачи время в пути составило 3.25 ч.

```
---in↓---
print ('часов в пути по условию:', 100/50 + 100/80)
print ('часов в пути при арифметическом:', (100*2)/65)
print ('часов в пути при гармоническом:', (100*2)/61.54)
```

```

---↑in_out↓---
часов в пути по условию: 3.25
часов в пути при арифметическом: 3.076923076923077
часов в пути при гармоническом: 3.249918752031199
---↑out---

```

Среднее геометрическое

Геометрическое среднее является n -м корнем произведения всех элементов x_i в наборе данных x , что определяется следующей формулой (8):

$$\bar{x}_{\text{геом}} = \sqrt[n]{\prod x_i}, \quad (8)$$

где $\bar{x}_{\text{геом}}$ — среднее геометрическое; \prod — знак, означающий произведение (перемножение всех элементов); $i = 1, 2, \dots, n$. Например, для массива из двух чисел, $\bar{x}_{\text{геом}}$ — это корень квадратный из их произведения и т.д.

На следующем рисунке (рис. 21) показаны средние арифметические, гармонические и геометрические значения набора данных (Mean, Harmonic mean и Geometric mean соответственно):

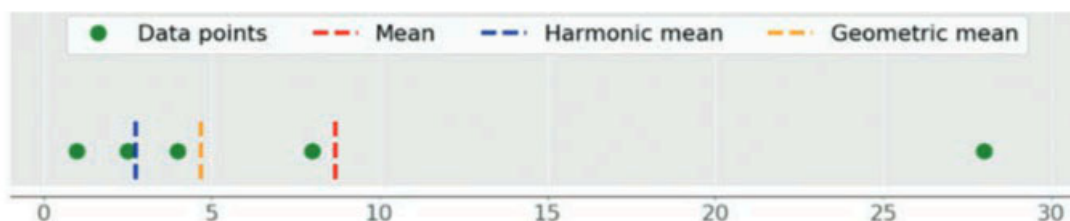


Рис. 21. Различные виды среднего значения, где Data points — значения элементов исходного массива

Зеленые точки представляют точки данных 1, 2.5, 4, 8 и 28. Красная пунктирная линия — это среднее (арифметическое) значение. Синяя пунктирная линия — среднее гармоническое, а желтая — среднее геометрическое. Наглядно сравните существенную разницу в значениях средних величин.

Вы можете реализовать среднее геометрическое без использования библиотек следующим образом:

```

---in↓---
x = [8.0, 1, 2.5, 4, 28.0]

gmean = 1
for item in x:
    gmean *= item
gmean **= 1 / len(x)
gmean
---↑in_out↓---
4.677885674856041
---↑out---
```

Значение среднего геометрического (4.67) в этом случае значительно отличается от значений арифметических (8.7, для справки отображено на рисунке (рис. 21)) и гармонических (2.76) средних для того же набора данных x .

Среднее геометрическое применяется в тех случаях, когда индивидуальные значения признака представляют собой относительные величины динамики, построенные в виде связанных величин, как отношение к предыдущему уровню каждого уровня в ряду динамики, т.е. характеризует средний коэффициент роста.

Ниже представлен пример использования доступного метода `geometric_mean()` в модуле `statistics` для вычисления геометрического среднего, который выполняет за программиста уже описанную последовательность действий и возвращает среднее геометрическое:

```

---in↓---
gmean = statistics.geometric_mean(x)
gmean
---↑in_out↓---
4.67788567485604
---↑out---
```

У вас будет тот же результат, что и в предыдущем примере, но с минимальной ошибкой округления. Дело в том, что компьютер считает числа с плавающей точкой (дробные числа) с некоторой погрешностью (например, `print (0.1 + 0.1 + 0.1)` выведет

0.30000000000000004). Кратно усугублять ситуацию и уменьшать точность может большое количество таких округлений.

Почему вообще возникает неточность, рассматривать не будем. Главное отметить, что необходимо использовать те средства, которые предназначены для решения соответствующих им задач.

В данном случае лучше использовать готовый метод `statistics.geometric_mean()`, чем писать свой собственный код. Также бывают ситуации, когда нам нужно использовать не общепринятое округление (округление в большую сторону), а банковское округление, чтобы избежать накопления ошибок. Но это не касается темы текущего материала и сообщено вам для расширения кругозора и самостоятельного изучения.

Если вы передаете данные с NaN-значениями, тогда `statistics.geometric_mean()` будет вести себя как большинство аналогичных функций и вернет NaN. Если есть хотя бы один 0, вернется 0.0 и будет показано предупреждение. Если вы укажете хотя бы одно отрицательное число, то получите NaN и предупреждение. Ниже показан пример, в котором в массиве есть одно NaN-значение.

```
---in↓---
import math

x_with_nan = [8.0, 1, 2.5, math.nan, 4, 28.0]
gmean = statistics.geometric_mean(x_with_nan)
gmean
---↑in_out↓---
nan
---↑out---
```

Статистика при помощи модуля `Scipy.Stats`

Этот модуль содержит большое количество распределений вероятностей, сводную и частотную статистику, корреляционные функции и статистические тесты, маскированную статистику, ядерную оценку плотности, функциональность квази-Монте-Карло и многое другое.

Статистика — это очень обширная составляющая анализа данных, многие направления которых охватывает библиотека SciPy.

Основные ее функции представлены ниже:

- непрерывные распределения;
- многомерные распределения;
- дискретные распределения;
- сводные статистические данные;
- частотная статистика;
- корреляционные функции;
- статистические тесты;
- трансформации;
- статистические расстояния;
- случайная вариативная генерация;
- круговые статистические функции;
- функции таблицы непредвиденных обстоятельств;
- сюжет-тесты;
- функции скрытой статистики;
- одномерная и многомерная ядерная оценка плотности.

Полный список функциональности `Scipy.Stats` доступен на официальном сайте (см. источник [31]). В разделе «Summary statistics» (с англ. — «Сводная статистика») данного источника, вы можете ознакомиться с реализациями некоторых уже изученных функций (например геометрическое среднее) и иных.

`Scipy.Stats`, как и большинство Python-модулей, имеет обширное учебное пособие на английском языке, которое вы можете попробовать изучить самостоятельно по источнику [32].

Давайте кратко рассмотрим выдержки из данного учебного пособия, адаптированные на русский язык. Изучим работу со следующими **функциями библиотеки `scipy`**:

- дискретные статистические распределения;
- непрерывные статистические распределения.

Случайные переменные

Существует два общих класса распределения, которые были реализованы для непрерывных случайных величин и дискретных случайных величин. С помощью этих классов было реализовано

более 80 типов непрерывных случайных величин и 10 дискретных случайных величин. Кроме того, пользователь может легко добавить новые типы распределения.

Все статистические функции и доступные случайные величины находятся в субпакете `scipy.stats`.

В приведенных ниже примерах мы в основном будем сосредотачиваться на непрерывных случайных величинах (на англ. обозначаются как RV, т.е. *Random Value* — «случайная величина»).

Практически все рассматриваемое также может быть применимо и к дискретным переменным, но с некоторыми отличиями, о которых вы сможете самостоятельно прочитать в разделе «Specific points for discrete distributions» (с англ. — «Конкретные точки для дискретных распределений») источника [32].

Для импорта базовой функциональности модуля `scipy.stats` воспользуемся следующим кодом:

```
---in↓---
from scipy import stats
---↑in_out↓---
---↑out---
```

Многие объекты и функции вынесены в отдельные submodule субмодуля `scipy.stats`, например, если нам нужно работать с нормальным распределением, мы можем импортировать его функциональность представленным ниже способом (т.е. некоторые отдельные объекты, такие как функциональность распределений, нужно дополнительно импортировать отдельно):

```
---in↓---
#для работы с нормальным распределением
from scipy.stats import norm
---↑in_out↓---
---↑out---
```

Получение справки

Во-первых, все классы распределений сопровождаются вспомогательными функциями.

Для того чтобы получить основную информацию о доступной в классе функциональности, мы можем использовать вызов «`print(stats.norm.doc)`», который выведет документацию субмодуля, содержащего функциональность нормального распределения.

Например, используя встроенную документацию, мы можем узнать, что, чтобы найти область допустимых значений функции распределения (т.е. верхнюю и нижнюю границы распределения), мы можем использовать метод `norm.support()`, который вернет два значения — значения нижней и верхней границы:

```
---in↓---
print('границы распределения: \nнижняя: %s, \nверхняя:
%s' % norm.support())
---↑in_out↓---
границы распределения:
нижняя: -inf,
верхняя: inf
---↑out---
```

Вывести все доступные методы и свойства распределения можно с помощью вызова `dir(norm)`. Некоторые из методов являются внутренними и не доступны для вызова извне, т.е. используются для вычислений внутри класса (модуля распределения).

Ниже показано, что модуль, реализующий «нормальное распределение», содержит, например, метод вычисления медианы «`median()`», аналог которого мы уже использовали при работе с другими библиотеками.

```
---in↓---
rv = norm()
dir(rv) # вывод доступных методов
---↑in_out↓---
[---часть методов вырезана---
'__dir__',
'__doc__',
'dist',
'mean',
```

```
'median',  
'moment']  
---↑out---
```

Давайте получим список доступных распределений:

```
---in↓---  
dist_continu = [d for d in dir(stats) if  
isinstance(getattr(stats, d), stats.rv_continuous)]  
dist_discrete = [d for d in dir(stats) if  
isinstance(getattr(stats, d), stats.rv_discrete)]  
  
print('количество непрерывных распределений: %d' %  
len(dist_continu))  
print('количество дискретных распределений: %d' %  
len(dist_discrete))  
---↑in_out↓---  
количество непрерывных распределений: 104  
количество дискретных распределений: 19  
---↑out---
```

Мы видим, что доступна работа с очень большим количеством функций распределения.

Общие методы

Основными общедоступными методами класса непрерывных случайных величин являются следующие методы, при помощи которых можно получить, например, кумулятивную функцию распределения случайных величин (CDF), или любую другую функцию, или набор параметров из нижеперечисленных:

- `rvs` — получение случайных значений в указываемом в параметрах количестве;
- `PDF` — вычисление функции плотности вероятности для ряда значений;
- `SF` — функция выживания (1-CDF);
- `ppf` — функция для вычисления точек плотности (поиск процентиля, обратная CDF);
- `isf` — обратная функция выживания (обратная SF);

где буквы в аббревиатурах: *d* — *distribution* (с англ. — «распределение»); *f* — *function* (с англ. — «функция»).

В рамках данного издания не рассматривается теория статистики, поэтому ее предлагается изучить самостоятельно. Однако далее мы изучим некоторые простые аспекты работы с данными функциями.

Типы данных в статистическом анализе

Разделение данных на качественные и количественные — основополагающий принцип статистического анализа данных. Типы данных представлены на рисунке ниже (рис. 22) и определяют, какие статистические методы могут использоваться с ними. Например, вы должны анализировать непрерывные данные иначе, чем категориальные, так как в противном случае вы получите неправильный результат анализа. Поэтому, зная типы данных, с которыми вы имеете дело, вы сможете выбрать правильный метод анализа.



Рис. 22. Типы данных в статистическом анализе

Рассмотрим каждый тип данных в отношении того, какие статистические методы и инструменты визуализации можно к ним применять. Чтобы правильно понять, что мы сейчас будем изучать,

вы должны знать основы описательной статистики, т.е. иметь представление о следующих понятиях: нормальное распределение, центральная тенденция (средняя, мода, медиана), меры изменчивости (диапазон, квартили), дисперсия и стандартное отклонение, модальность, перекош, эксцесс.

Некоторую часть этих понятий мы уже рассмотрели в рамках данного занятия, закрепить их понимание и изучить нерассмотренные материалы вы можете в источнике [33]. Рекомендуется ознакомиться с ее содержанием для лучшего понимания изучаемой темы.

Для визуализации статистики наиболее часто используется три типа диаграмм, представленных на рисунке ниже (рис. 23).

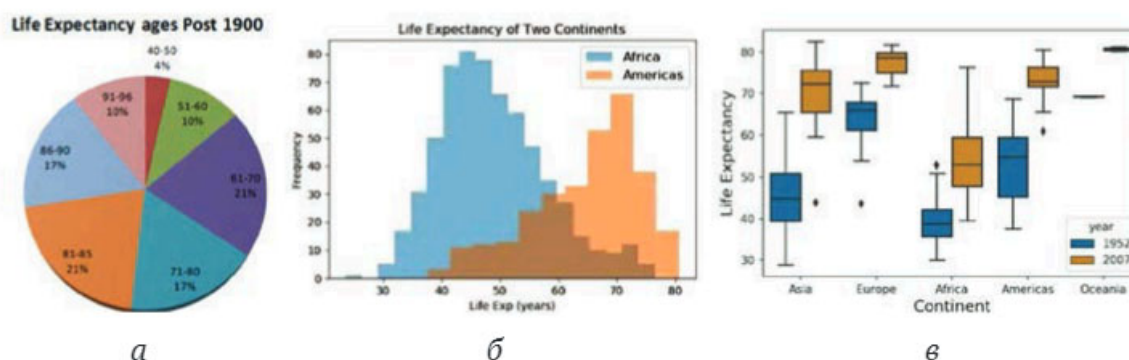


Рис. 23. Диаграммы в статистическом анализе:
 а — круговая диаграмма; б — столбиковая диаграмма;
 в — диаграмма размаха

Номинальные данные

Когда вы имеете дело с номинальными данными, вы собираете информацию посредством определения частоты каких-либо вещей (например, насколько часто группа крови — четвертая отрицательная) либо строите частоту в контексте доли от общего количества значений (можно выразить долю в процентном соотношении, например 0 % людей имеют пятую группу крови).

Номинальные данные можно посчитать (суммарное количество значений), можно определить процент от целого, однако нельзя вычислить среднее значение. Например, можно говорить о том,

сколько продуктов в вашей корзине было взято в молочном отделе или сколько процентов от покупок занимают овощи, но посчитать среднее количество конкретных типов товаров корзине невозможно: нельзя утверждать, что «в среднем в моей корзине 20 % товаров занимает молоко», потому что у нас одна корзина.

Для визуализации номинальных данных часто используют круговую диаграмму или гистограмму (т.е. столбиковые диаграммы).

В случае если доступны только две категории, данные относят к типу дихотомических (или бинарных, т.е. 0 или 1). Ответы на вопросы, требующие ответа «да/нет», — это и есть дихотомические данные. Если, делая покупки, вы собрали данные о том, продавался товар со скидкой или нет, это и будут дихотомические данные.

Порядковые данные

Когда вы имеете дело с порядковыми данными, вы можете использовать те же методы, что и для номинальных данных, но в случаях, когда также есть доступ к некоторым дополнительным инструментам, вы можете связать элементы данных с частотами, пропорциями, процентами, а также применять процентиля, медианы, моды и межквартильный диапазон.

Наиболее часто используемые средства визуализации те же, что и у номинальных данных: круговые диаграммы и гистограммы (т.е. столбиковые диаграммы).

Например, вы можете в статистике настроения людей или в статистике оценок учащихся вычислять среднее настроение или среднюю оценку каждого человека, потому что эти примеры имеют как бы несколько порядков «измеримости», в случае с настроением — оно каждый месяц у одного и того же человека разное, как и в случае с оценкой — она разная по каждому предмету.

Но возможность вычисления среднего, медианы и подобных показателей доступна не всегда. Например, если мы имеем данные анкет обратной связи, в которых производился опрос клиентов по поводу того, довольны ли они качеством обслуживания и вкусом блюд в ресторане, и при этом на выбор были предложены варианты: «очень доволен, нейтрально, очень недоволен», то логично, что

будет невозможно посчитать среднее количество клиентов, которые ответили «очень доволен». Мы можем лишь делать вывод типа «в среднем клиенты довольны, количество довольных клиентов 60 %», но не «в среднем довольно 60 % клиентов».

Ситуацию также усугубляет логическая неоднозначность веса ответов. Формулировка ответов схожа, и кажется, что линейно отражает степень удовлетворенности. Но по факту мы имеем три разных ответа, расценивание которых может накладывать ограничения на методы статистики. Часто встречаются и другие подобные спорные моменты.

Непрерывные данные

Когда вы имеете дело с непрерывными данными, вы можете использовать большинство из всех доступных методов для описания данных.

Для визуализации непрерывных данных наиболее часто применяют столбиковые диаграммы (гистограммы) или диаграммы размаха (или «ящик с усами», англ. *box plot*). С помощью гистограммы вы можете проверить центральную тенденцию, изменчивость, модальность (мода — это значение, которое случайная величина на заданном множестве наблюдений принимает наиболее часто) и эксцесс распределения (англ. *kurtosis*, мера остроты пика в распределении случайной величины, характеризует удаленность пика значений от средней величины).

Обратите внимание, что гистограмма в отличие от диаграммы размаха не может показать вам наличие каких-либо выбросов в данных.

Кумулятивная функция распределения (CDF)

Начнем изучение некоторых конкретных примеров работы с модулем статистики библиотеки SciPy. Рассмотрим кумулятивную функцию распределения (CDF, англ. *Cumulative Distribution Function*). Она показывает вероятность того, что целевое значение меньше указанного значения либо равно ему. Она доступна только для количественных целевых значений.

В коде ниже получим график кумулятивной функции (рис. 24) для нормального закона распределения для диапазона значений $[-3; 3]$ с шагом 0.5:

```

---in↓---
import numpy as np
import matplotlib.pyplot as plt #библиотека для графиков
#выводить график внутривстрочно
%matplotlib inline

x = np.arange(-3, 3, 0.5)
my_cdf = norm.cdf (x)

plt.plot(x, my_cdf)
plt.grid(linestyle='--') #добавить пунктирную сетку
plt.show() #показать график

print ('значения', x)
print ('cdf', my_cdf.round(2)) #округлим до 2 знаков после
запятой
---↑in_out↓---
#часть вывода показана на рисунке (рис. 24)
значения [-3.  -2.5 -2.  -1.5 -1.  -0.5 0.  0.5 1.  1.5 2.  2.5]
cdf [0.  0.01 0.02 0.07 0.16 0.31 0.5  0.69 0.84 0.93 0.98 0.99]
---↑out---

```

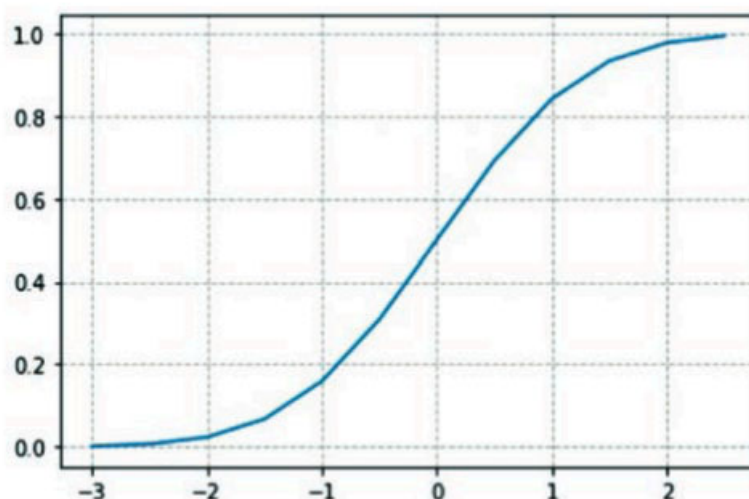


Рис. 24. Диаграмма CDF

Как видим на рисунке (рис. 24), центр данной функции соответствует вероятности 0.5. Это значит, что с вероятностью 50 % наше значение меньше или равно нулю (по определению кумулятивной функции) или, например, с вероятностью 0.99 значение числа в ряду значений меньше или равно 2.5, т.е. почти 100 % («почти», потому что у нас шаг 0.5 в ряду, и происходит небольшая ошибка округления; т.е. если установить шаг, например, 0.1, то значение CDF округлится в сторону 100 %, что на самом деле более точно, чем 99 %).

Обращаем внимание, что данные `np.arange(-3, 3, 0.5)` используются только для построения горизонтальной оси и не являются случайными величинами. Здесь мы использовали метод `cdf()` из класса нормального распределения, поэтому и получили «идеальный» график CDF для нормального распределения с областью значений от -3 до 2.5.

Вывод графиков при помощи библиотеки `matplotlib` (и `plotly`)

Отметим, что в примере выше мы используем библиотеку `matplotlib` с целью создания графика. Это наиболее распространенная библиотека, решающая задачи визуализации данных на графиках и диаграммах.

Мы не будем отдельно изучать ее или другие подобные библиотеки, но они будут часто необходимы в дальнейшей работе. Поэтому рекомендуется самостоятельно изучить данную библиотеку путем экспериментов с передаваемыми ей параметрами.

Строка `«%matplotlib inline»` называется магической командой²⁵, встроенной в интерактивную оболочку IPython для языка программирования Python. При помощи данной строки мы указываем интерактивной оболочке, чтобы библиотека `matplotlib` выводила графики внутрь блокнота Jupyter.

Если написать, например, `«%matplotlib qt»`, то графики будут выводиться при помощи фреймворка графических интерфейсов qt, если он доступен в вашей операционной системе. Вы сможете взаимодействовать с графиком в отдельном окне (как показано на рис. 25), а не в браузере.

²⁵ Ознакомиться со всеми встроенными магическими командами можно в источнике [34]. Их довольно много.

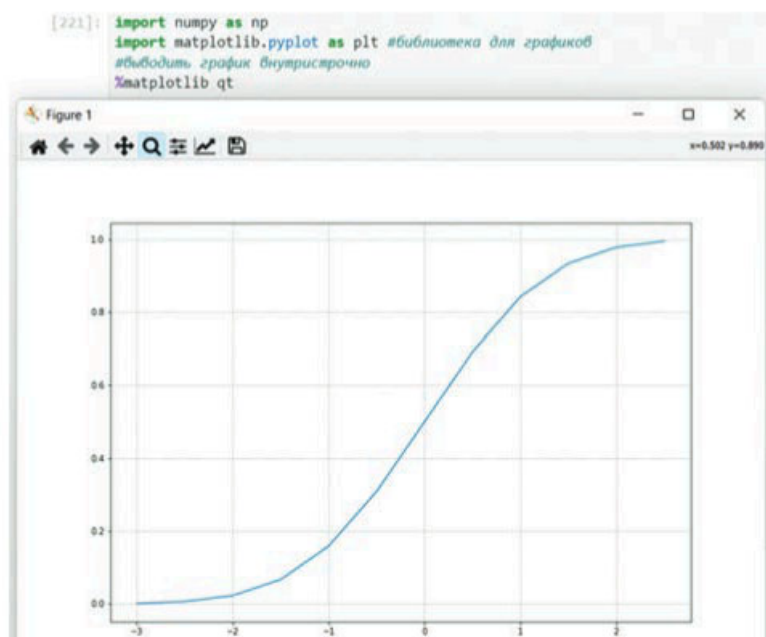


Рис. 25. Вывод matplotlib в qt-окне

Стоит отметить, что, например, при помощи библиотеки `plotly` можно встраивать интерактивные графики внутрь вывода Jupyter-блокнота, но это требует наличия расширения в Jupyter. В занятии по библиотеке `TensorFlow` для разнообразия будет использована именно библиотека `plotly` (для построения графиков).

В целом, далее будет представлено много различных применений библиотеки `matplotlib` и `plotly`, поэтому вы сможете изучить аспекты вывода диаграмм в непринужденно-пассивном стиле обучения на практике.

NumPy в контексте Scipy.

Кумулятивная функция распределения (CDF, продолжение)

Вы можете передавать как NumPy-массивы в кумулятивную функцию (как было показано в примере выше), так и обычные Python-массивы. Ниже показано, что результат генерации вероятностей одинаковый:

```
---in↓---
print (norm.cdf([-3, 0, 3]))
```

```
import numpy as np
print (norm.cdf(np.array([-3, 0, 3])))
---↑in_out↓---
[0.0013499 0.5 0.9986501]
[0.0013499 0.5 0.9986501]
---↑out---
```

Так как многие методы возвращают NumPy-массивы, то вы можете применять к ним уже рассмотренные (и встроенные в NumPy) методы, например нахождение средних значений, отклонений, минимумов и максимумов и других показателей.

Несколько примеров таких методов продемонстрировано ниже.

```
---in↓---
x = np.arange(-3, 3.5, 0.5) #"случайные" значения
c = norm.cdf(x) #значения процентилей
print ('▶x:', x)
print ('▶c:', c.round(2))
print('▶x.mean(), x.std(), x.var(), x.ptp():')
print (x.mean().round(2), x.std().round(2), x.var().
round(2), x.ptp().round(2), sep=", ")
print('▶c.mean(), c.std(), c.var(), c.ptp():')
print (c.mean().round(2), c.std().round(2), c.var().
round(2), c.ptp().round(2), sep=", ")
---↑in_out↓---
▶x: [-3. -2.5 -2. -1.5 -1. -0.5 0. 0.5 1. 1.5 2. 2.5 3. ]
▶c: [0. 0.01 0.02 0.07 0.16 0.31 0.5 0.69 0.84 0.93 0.98
0.99 1. ]
▶x.mean(), x.std(), x.var(), x.ptp():
0.0, 1.87, 3.5, 6.0
▶c.mean(), c.std(), c.var(), c.ptp():
0.5, 0.4, 0.16, 1.0
---↑out---
```

где:

- x — «случайные» значения. По факту они не случайные. Используются здесь просто как область значений для построения cdf нормального распределения;

- c — значения процентилей.

Встроенные в NumPy методы:

- `.mean()` — среднее арифметическое;
- `.std()` — стандартное отклонение;
- `.var()` — дисперсия, или разброс;
- `.ptp()` — размах.

Один из примеров — вывод среднего по рассчитанным значениям CDF (кумулятивной функции распределения) при помощи `c.mean()`.

По сути, вычисление значений кумулятивной функции распределения дает нам значения процентилей (по определению кумулятивной функции распределения), только в виде десятичной дроби, т.е., например, 0.50 вместо 50 %.

В примере видно, что среднее по «процентилям» `c.mean()` равно 0.5. Это логично, потому что взятые «случайные» значения в ряде `x` на самом деле не случайны и распределены равномерно (так как получены простой генерацией «`np.arange(-3, 3.5, 0.5)`», т.е. дискретно от -3 включительно до 3.5 не включительно с шагом 0.5).

Если бы «случайные» значения были сконцентрированы правее от значения медианы, то значение `c.mean()` было бы больше 0.5, как показано в примере ниже.

```

---in↓---
x = np.arange(-3, 3.5, 0.5) #"случайные" значения
x = np.append(x, np.arange(1, 1.5, 0.1)) #увеличим
количество значений.
#т.е. "уплотним" распределение правее медианы
x = np.sort(x) #отсортируем значения по возрастанию (это
обязательно)
c = norm.cdf(x) #значения процентилей
print('►x:', x)
print('►c.mean(), c.std(), c.var(), c.ptp():')
print(c.mean().round(2), c.std().round(2), c.var().
round(2), c.ptp().round(2), sep=", ")
---↑in_out↓---
►x: [-3.  -2.5 -2.   -1.5 -1.   -0.5  0.    0.5  1.    1.
 1.1  1.2  1.3  1.4
 1.5  2.   2.5  3. ]

```

```
►c.mean(), c.std(), c.var(), c.ptp():  
0.61, 0.38, 0.15, 1.0  
---↑out---
```

Чтобы найти медиану распределения, мы можем воспользоваться, например, уже известной нам NumPy-функцией `np.median()`, передав в нее ряд значений.

Ниже будет продемонстрирован пример использования `np.median()`, а также применение функции поиска значения процентиля (т.е. `ppf`-функции, которая является обратной по отношению к CDF).

Функция получения значения процентиля (PPF)

PPF (англ. *Percent Point Function*) — функция, используемая для получения значения, соответствующему определенному процентилю.

Например, как уже было сказано (последний абзац раздела «Процентили» на стр. 220), значение 50-го процентиля соответствует значению медианы. То есть, передав методу `norm.ppf()` 50-й процентиль, вы получите значение медианного среднего значения. Однако эта функция принимает числа в диапазоне от 0 до 1, а не от 0 до 100, поэтому в случае с 50-м процентилем указывайте «`norm.ppf(0.5)`», как в примере ниже:

```
---in↓---  
import numpy as np  
  
x = np.arange(-3, 3.5, 0.5)  
x = np.append(x, np.arange(1, 1.5, 0.1)) #увеличим  
количество значений.  
#т.е. "уплотним" распределение правее медианы  
x = np.sort(x) #отсортируем значения по возрастанию (это  
обязательно)  
print("►x: ", x)  
  
print ("►np.median(x): ", np.median(x))
```



```

my_ppf = norm.ppf(0.5, loc= np.median(x), scale=x.std())
print ("►norm.ppf(0.5), т.е. 50-й процентиль или медиана:
", my_ppf)

#np.percentile() и norm.ppf()
print (np.percentile(x,30,0), "не равно", norm.ppf(0.3,
loc=np.median(x), scale=x.std()))
print ("т.к. norm.ppf() работает только с нормальным
распределением")
---↑in_out↓---
►x:  [-3.  -2.5 -2.  -1.5 -1.  -0.5  0.   0.5  1.   1.
1.1  1.2  1.3  1.4
     1.5  2.   2.5  3. ]
►np.median(x):  1.0
►norm.ppf(0.5), т.е. 50-й процентиль или медиана:      1.0
-0.450000000000000002 не равно 0.11903488514629679
т.к. norm.ppf() работает только с нормальным
распределением
---↑out---

```

Обратите внимание, что ряд дискретных значений, сгенерированный от -3 до 3 с шагом 0.5 , распределен по равномерному закону, а не по нормальному. Поэтому получение любых других значений процентилей (кроме медианного) будет давать неправильные значения процентилей. Про вызов «`norm.ppf(0.5, loc=np.median(x), scale=x.std())`», а точнее про параметры `loc` и `scale` (сдвиг и масштаб соответственно), будет рассказано на следующем занятии.

Согласитесь, что странно при помощи конструкции «`norm.ppf(0.5, loc=np.median(x), scale=x.std())`», уже содержащей медианное значение `np.median(x)`, получать это же медианное значение.

Дело в том, что если бы использованный ряд значений соответствовал нормальному закону распределения, то как параметр сдвига мы бы указали среднее арифметическое значение, т.е. «`loc=x.mean()`».

Указывать медиану как параметр сдвига некорректно, но в примере выше так было сделано только для того, чтобы продемонстрировать работоспособность функции получения процентилей `ppf` из модуля для работы с нормальными распределениями на при-

мере работы с рядом, распределенным не по нормальному закону распределения. Этот нюанс и работа со сдвигом и масштабом будут подробно рассмотрены на следующем занятии.

Также на следующем занятии мы изучим, как генерировать ряды случайных величин, распределенных по нормальному закону распределения, и закрепим полученные навыки на примере из реальной жизни.

Аналогичным образом работают и другие базовые методы, такие как PDF (функция плотности вероятностей), обратная CDF (т.е. функция выживания, может обозначаться как S_f) и другие, не упомянутые в разделе «Общие методы» на стр. 230.

Задания для проверки

1. Базовая функциональность NumPy. Опишите доступные статистические функции. Повторите то же для встроенного модуля `statistics`.

2. Примеры применения среднеарифметического, медианы, среднегеометрического и гармонического среднего, процентилей.

3. Опишите типы данных в статистическом анализе. Приведите примеры.

4. Модуль `Scipy.Stats`. Опишите доступные возможности для статистики.

5. Представьте код, который строит графики кумулятивной функции распределения (CDF), функции получения процентилей (`ppf`) и функции плотности вероятности (PDF) для сгенерированного при помощи NumPy набора дискретных значений от -3 до 3 с дискретностью 0.1 (как в примерах в занятии).

6. Обратите внимание, что данные $[-3; 3)$ распределены по равномерному закону распределения, так как генерируются при помощи «`np.array([-3, 0, 3])`», поэтому влияют только на область значений по горизонтальной оси (на следующем занятии это будет рассмотрено подробнее).

7. Проведите эксперименты с библиотекой для построения графиков `matplotlib`. Попробуйте добавить `label` для графиков, `xlabel/ylabel` для осей, и `title` для всего графика целиком. На следующем занятии будет представлено решение этих задач.

SCIPY. ПРИМЕРЫ СТАТИСТИКИ, Z-СТАТИСТИКА И P-VALUE²⁶

Цели выполнения работы

Продолжение изучения функциональности, доступных методов и объектов статистики библиотеки Scipy: генерация случайных чисел и параметры формы распределения. Исследование принадлежности ряда значений к конкретному закону распределения (на основе параметров формы, куртозиса и др.). Построение диаграмм размаха. Рассмотрение примеров описательной статистики из реального мира (Z-статистика и p-значение).

Порядок выполнения работы

Генерация случайных чисел

Генерация случайных чисел (рядов величин случайного распределения) зависит от генераторов из модуля `numpy.random`.

Часто бывает необходимо сгенерировать ряд чисел, который при каждой новой регенерации одинаков, например мы хотим его использовать как опорный при сравнении с другим рядом чисел, чтобы сравнить, насколько схожа форма исследуемого распределения с тем или иным законом распределения, чтобы далее определить, какие статистические методы и подходы возможно применить к данному ряду случайных величин.

Чтобы случайно сгенерированный опорный ряд чисел не менялся при каждом новом запуске кода и не отвлекал тем самым концентрацию аналитика данных (или ввиду любой другой причины), мы можем сделать сгенерированный ряд чисел воспроизводимым.

Чтобы добиться воспроизводимости, необходимо явно указать генератор случайных чисел. В NumPy генератор является экземпляром класса `numpy.random.Generator`. Вот канонический (т.е. наиболее часто используемый) способ создания генератора:

²⁶ Разработано на основе официального руководства SciPy [28] и русскоязычного источника [35].

```
---in↓---
from numpy.random import default_rng
rng = default_rng()
---↑in_out↓---
---↑out---
```

Таким образом, мы записали в переменную `rng` экземпляр генератора, который далее сможем использовать в `scipy`. Однако необходимо изначально «закрепить» сид (англ. *seed* — «семя»²⁷), и сделать это можно представленным ниже способом. Сид используется как начальное условие для генерации случайного числа, т.е. некоторый стартовый шум. Чем он длиннее, тем лучше.

```
---in↓---
#попробуйте заменить 1234 на что-то свое
rng = default_rng(1234)
---↑in_out↓---
---↑out---
```

Генерация по нормальному закону распределения

В примере ниже мы сгенерировали 5 случайных величин, распределенных по нормальному закону распределения, используя вызов «`norm.rvs(size=5, random_state=rng)`»:

```
---in↓---
#для работы с нормальным распределением
from scipy.stats import norm

norm.rvs(size=5, random_state=rng)
---↑in_out↓---
array([-1.60383681,  0.06409991,  0.7408913 ,
  0.15261919,  0.86374389])
---↑out---
```

²⁷ То есть сид — это, условно говоря, «что-то с определенным набором исходных признаков, которые будут воспроизведены». В данном случае речь про опорную комбинацию чисел, на базе которой генерируется случайная последовательность.

где:

- `size` — количество значений для генерации (сколько сгенерировать);

- `random_state` — целочисленное значение сида, или экземпляр класса генератора. В данном случае мы передали экземпляр генератора с указанным сидом.

Теперь данные генерируются на основе экземпляра генератора `rng`, в котором ранее был указан сид «1234», и форма их распределения всегда будет одинакова.

По умолчанию, т.е. если не указывать параметр `random_state`, используется `numpy.random.RandomState`-генератор со случайным сидом (но `numpy.random.Generator` — более совершенный генератор). Если вас устраивает генератор по умолчанию, то вы можете установить только сид, а не менять генератор целиком:

```
---in↓---
norm.rvs(size=5, random_state=1234)
---↑in_out↓---
array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 ,
        -0.72058873])
---↑out---
```

Если вам нужно сгенерировать сид, то хорошим способом для этого является представленный ниже:

```
---in↓---
from numpy.random import SeedSequence

print(SeedSequence().entropy)
---↑in_out↓---
266228675869187944950978558454195848660
---↑out---
```

В рамках данного занятия конкретное значение сида не является принципиальным. Вы можете установить сид на свое усмотрение или вовсе его не указывать.

Кроме генерации рядов с нормальным распределением, можно генерировать наборы случайных величин с гамма-распреде-

нием или с любым другим типом распределения, полный список которых можно найти в официальной документации `scipy` или `numpy`. Принцип такой же, как и при рассмотренном ранее вызове `norm.rvs()`, только может отличаться набор передаваемых параметров. Например, для гамма-распределения требуется дополнительно указывать параметр формы. В коде ниже показано три примера генерации рядов с гамма-распределением.

```
---in↓---
from scipy.stats import gamma

a = 1.99 # форма
b = 1   # масштаб

rng = default_rng(seed=1234)

print(
    "1)Дважды вызовем gamma.rvs() из scipy.stats:\n",
    gamma.rvs(a=a, scale=b, size=3, random_state=rng),
    "не равно",
    gamma.rvs(a=a, scale=b, size=3, random_state=rng),
)
print("Так как сид при каждом использовании меняется")
print("2)Вызов rng.gamma() из numpy.random:")
rng = default_rng(seed=1234)
print(rng.gamma(a, scale=b, size=3))
print(
    "Он равен первому запуску gamma.rvs из пункта 1, так
как у него сид тоже 1234",
)

rng = default_rng(seed=1234)
print(
    "3)gamma.rvs со сдвигом loc=+5:\n",
    gamma.rvs(a=a, loc=5, scale=b, size=3, random_state=rng),
)
print("Значения равны первому запуску из п. 1, но каждое
увеличено на 5.")
---↑in_out↓---
```



```

1) Дважды вызовем gamma.rvs() из scipy.stats:
[0.33106082 2.80495534 3.03563217] не равно [0.38916187
0.30440218 1.08075301]
Так как сид при каждом использовании меняется
2) Вызов rng.gamma() из numpy.random:
[0.33106082 2.80495534 3.03563217]
Он равен первому запуску gamma.rvs из пункта 1, так как
у него сид тоже 1234
3) gamma.rvs со сдвигом loc=+5:
[5.33106082 7.80495534 8.03563217]
Значения равны первому запуску из п. 1, но каждое
увеличено на 5.
---↑out---
```

Представим комментарии для каждого из трех пунктов приведенного выше примера (нумерация соответствует нумерации в коде):

1. Даже если вы указали сид, то он используется только как начальное условие.

На втором и следующих запусках генератор на основе, казалось бы, этого же сида будет генерировать уже другие значения. Поэтому необходимо указывать сид каждый раз при необходимости при одном и том же действии получить одинаковые ряды случайных чисел.

Но такое бывает редко. Обычно вы указываете сид один раз, а далее получаете разные наборы данных на его основе. Когда нужно скопировать ряд чисел, то проще его действительно скопировать, чем «сбрасывать» генератор и устанавливать сид заново.

На основе однократно установленного сида вы можете получить, например, 3 ряда значений, которые константны при каждом перезапуске кода и не равны друг другу.

2. Так как функционал генерации случайных значений из библиотеки `scipy` основан на использовании библиотеки `numpy`, вы можете сгенерировать этот же ряд чисел, но сразу при помощи `numpy`²⁸.

Установив одинаковый сид и генератор, используя `scipy` и `numpy` методы генерации, мы получим одинаковые значения.

²⁸ Сравните, например, описания гамма-генератора из `scipy` и из `numpy`, доступных в источниках [36] и [37] соответственно. Они почти идентичны.

3. У `scipy` по сравнению с `numpy` доступны некоторые дополнительные возможности. Например, в данном случае в п. 3 примера представлено, что у метода генерации `scipy` нам доступен дополнительный параметр `loc`, который сдвигает все значения на требуемую величину.

При любой генерации последовательности случайных значений необходимое количество значений указывается именно как именованный параметр, т.е., например, «`size=3`». Что в случае генерации с нормальным распределением выглядит следующим образом:

```
---in↓---
norm.rvs(size=3)
---↑in_out↓---
array([ 0.77889638,  0.36312105, -2.07973357])
---↑out---
```

Не подумайте, что, написав «`norm.rvs(5)`», сгенерируется 5 чисел:

```
---in↓---
norm.rvs(5)
---↑in_out↓---
2.682483179604766
---↑out---
```

В примере выше аргумент 5 без ключевого слова интерпретируется как первый возможный аргумент, чем является ключевое слово `loc` (сдвиг значений), принимаемое всеми непрерывными распределениями. Это подводит нас к теме следующего подраздела, где мы более детально рассмотрим передаваемые параметры в методы генерации рядов случайных чисел.

Сдвиг и масштабирование

Все непрерывные распределения принимают `loc` и `scale` в качестве ключевых параметров для настройки положения и масштаба распределения (дословно с англ.: `loc` — *location* — «положение» или «сдвиг», а `scale` — «масштаб»).

Значения по умолчанию `loc=0` и `scale=1`.

Для стандартного нормального распределения положением должно являться среднее арифметическое значение, а масштаб — стандартное отклонение. Ниже показано, как получить правильный результат вычисления CDF-функции для всего ряда нормально распределенных случайных величин (п. 2 в коде ниже).

```

---in↓---
import numpy as np
import scipy.stats as stats

x = norm.rvs(size=5)
x=np.sort(x) #сортируем по возрастанию
print('1)x:', x)

my_cdf = norm.pdf(x, loc=x.mean(), scale=x.std())
print('2)cdf для x:', my_cdf)

print('3)stats.describe(x):', stats.describe(x))
---↑in_out↓---
1)x: [-0.625651    -0.26728092 -0.10486824  0.17924833
 0.62711208]
2)cdf для x: [0.35914866 0.81544587 0.93288882 0.82722257
 0.27306869]
3)stats.describe(x): DescribeResult(nobs=5,
minmax=(-0.6256510048396975, 0.6271120837434032),
mean=-0.03828795136620948, variance=0.22298632785041095,
skewness=0.23880761812403134,
kurtosis=-0.9884490395821279)
---↑out---

```

где:

- `x.mean()` — вычисление среднего арифметического для выборки `x`;

- `x.std()` — вычисление стандартного отклонения.

В п. 3 кода выше продемонстрировано использование метода `stats.describe()` библиотеки `scipy`, который выводит статистику по нашему распределению. Три последних значения — это разброс (дисперсия), асимметрия и коэффициент эксцесса (`variance`, `skewness` и `kurtosis` соответственно).

Генерация по показательному (экспоненциальному) закону распределения

Чтобы дополнительно проиллюстрировать масштабирование, рассмотрим подробнее кумулятивную функцию CDF и функцию плотности вероятности PDF для экспоненциального (или показательного) распределения. Функция плотности вероятности²⁹ для экспоненциального распределения вычисляется по формуле (9):

$$F(x) = \begin{cases} e^{-\lambda x}, & x \geq 0; \\ 0, & x < 0, \end{cases} \quad (9)$$

где:

- $F(x)$ — значение вероятности для значения x ;
- вместо параметра λ (`lambda`) указывается масштаб «`scale=1/lambda`». Так как масштаб по умолчанию равен 1, то, если не указывать масштаб³⁰, и λ равна единице.

Кумулятивная функция распределения (CDF) показывает вероятность того, что целевое значение меньше указанного значения либо равно ему. Она доступна только для количественных целевых значений.

Функция плотности вероятности (PDF) показывает распределение целевых значений. Для количественных целевых значений она позволяет определять вероятность того, что они находятся в данной области.

Поэкспериментируйте со сменой положения (`loc`) и масштаба (`scale`) в коде ниже.

```
---in↓---
import matplotlib.pyplot as plt # библиотека для графиков
from scipy.stats import expon

rng = default_rng(seed=1234)
```

²⁹ На русской версии ресурса «Википедия» подробно описаны функции и параметры для многих типов распределений. Например, в источнике [38] вы можете ознакомиться с функцией нормального распределения и его функцией плотности вероятности PDF.

³⁰ Взято из документации `scipy`, которую вы можете изучить в источнике [39].


```

scale = 1 # масштаб
loc = 0 # положение

xvalues = expon.rvs(size=100, loc=loc, scale=scale,
random_state=rng)
xvalues = np.sort(xvalues) # сортируем по возрастанию

pdf = stats.expon.pdf(xvalues, loc, scale) # функция
плотности вероятности
cdf = stats.expon.cdf(xvalues, loc, scale) #
кумулятивная функция

plt.xlabel("Значение X")
plt.ylabel("Значение вероятности")
plt.plot(xvalues, pdf, label="pdf: Плотность вероятности")
plt.plot(xvalues, cdf, label="cdf: Кумулятивная функция")
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.15))
# показать названия
plt.show() # показать график
---↑in_out↓---
вывод представлен на рисунке ниже (рис. 26)
---↑out---
```

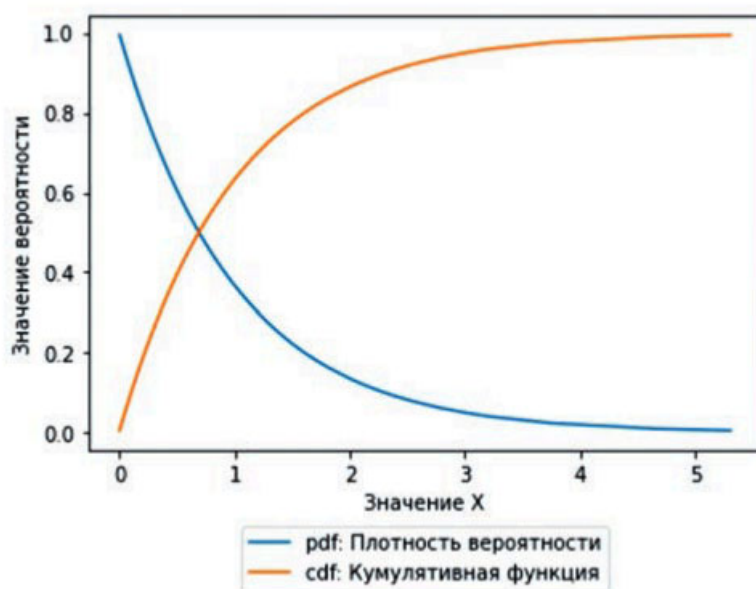


Рис. 26. Вывод графика CDF
для экспоненциального распределения

Генерация по равномерному закону распределения

Рассмотрим и равномерное распределение. Установите в коде выше `scale=5` и `loc=0`, чтобы увидеть классическое равномерное распределение:

```
---in↓---
from scipy.stats import uniform

scale = 4
loc = 0

xvalues = [0, 1, 2, 3, 4, 5]
cdf = uniform.cdf(xvalues, loc, scale)

plt.title("CDF для равномерного распределения")
plt.xlabel("Значение X")
plt.ylabel("Значение cdf вероятности")
plt.plot(xvalues, cdf)
---↑in_out↓---
вывод представлен на рисунке ниже (Рис. 27)
---↑out---
```

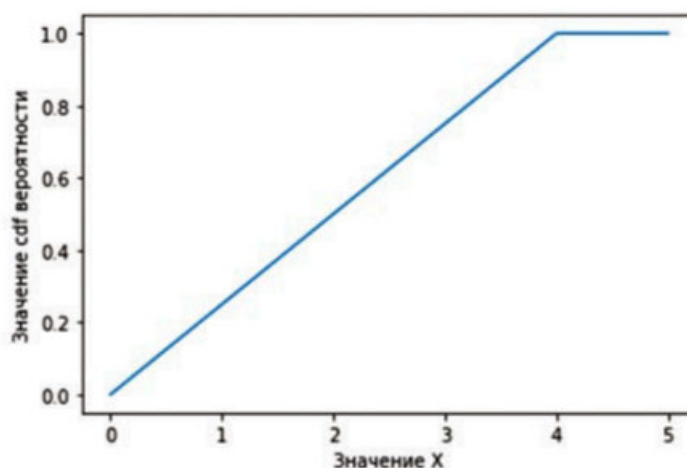


Рис. 27. Вывод графика CDF для равномерного распределения

Рекомендуется устанавливать `loc` и `scale`-параметры в явном виде, путем передачи ключевых слов и их значений, а не просто перечислять значение аргументов по порядку («имя_параметра=значение», а не «значение»).

Также вы можете создать экземпляр класса распределения, а потом его использовать. Например, представленным ниже способом:

```

---in↓---
xvalues = range(0, 11)
xvalues = np.sort(xvalues) # сортировка по возрастанию
print(xvalues)

rv = uniform(loc=xvalues.min(), scale=xvalues.max())
my_pdf = rv.pdf(xvalues)
print(my_pdf)
print("У равномерного распределения pdf равно для всех
значений, так как они равновероятны.")

plt.title("uniform.pdf: Плотность вероятности
равномерного распределения")
plt.xlabel("Значение X")
plt.ylabel("Значение вероятности")
plt.plot(xvalues, my_pdf)
---↑in_out↓---
[ 0  1  2  3  4  5  6  7  8  9 10]
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
У равномерного распределения pdf равно для всех значений,
так как они равновероятны.
#часть вывода представлена на рисунке ниже (Рис. 28)
---↑out---

```

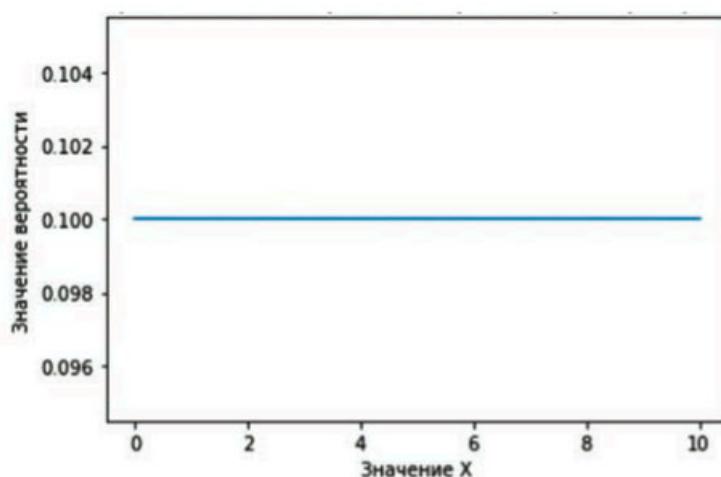


Рис. 28. Вывод плотности вероятности (PDF) для равномерного распределения

В примере выше в переменную `rv` мы записали сконфигурированное распределение. Такое действие в `scipy` называется «заморозкой» распределения. Используя «замороженное» распределение, мы можем передавать ему только данные, а все остальные параметры останутся такими, как было указано при «заморозке». В данном примере вызов «`rv.pdf(xvalues)`» по факту выполнит «`uniform.pdf(xvalues, loc=xvalues.min(), scale=xvalues.max())`».

Генерация по гамма-закону распределения. Дополнительные параметры формы

В то время как общая непрерывная случайная величина может быть сдвинута и масштабируется при помощи `loc` и `scale`-параметров, для некоторых других типов распределения могут существовать другие параметры формы. Например, гамма-распределение имеет следующую функцию плотности (формула (10) — полный вариант³¹, формула (11) — упрощенный):

$$F(x, \alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1}}{\Gamma(\alpha)} e^{-\beta x}; \quad (10)$$

$$F(x, a) = \frac{x^{a-1}}{\Gamma(a)} e^{-x}. \quad (11)$$

Формулы (10) и (11) справедливы для $x \geq 0$. При $x < 0$ плотность вероятности равна нулю. $\Gamma(a)$ — это гамма-функция.

Как видно, в формулах (10) и (11) требуется указывать как минимум еще один параметр, кроме уже рассмотренных `loc` и `scale`, а именно параметр формы α (альфа). Обратите внимание, что вы дополнительно также можете указать и параметр β , задав для ключевого слова `scale` значение $1/\beta$.

Реализуем сказанное выше, а именно: зададим параметры формы α и β гамма-распределения (код ниже). Поэкспериментируйте с параметрами формы и сравните их с готовыми графиками плотности вероятности гамма-распределения, доступными в сети Ин-

³¹ Подробнее можно изучить по источнику [36].

тернет, например, по источнику [40] (рис. 30, где k — это a , а θ — это $scale$ ³²).

```

---in↓---
from scipy.stats import gamma

a = 9.0
beta = 2.0
scale = 1 / beta

xvalues = gamma.rvs(size=300, a=a, scale=scale)
xvalues = np.sort(xvalues) # сортируем по возрастанию

pdf = gamma.pdf(xvalues, a, loc=0, scale=scale)

plt.title("gamma.pdf: Функция плотности для гамма-
распределения")
plt.xlabel("Значение X")
plt.ylabel("Значение вероятности")

plt.plot(xvalues, pdf)
---↑in_out↓---
#часть вывода представлена на рисунке ниже (Рис. 29)
---↑out---
```

Когда α — целое число, то такое гамма-распределение также называется распределением Эрланга порядка альфа, а когда $\alpha = 1$, то такое гамма-распределение сводится (преобразуется) к экспоненциальному (показательному) распределению с параметром β (напоминаем, в интерпретации рисунка из сети «Интернет» (рис. 30) вместо α и β понимается k и θ соответственно, т.е. разные обозначения одного и того же, и при этом в `scipy` вы задаете β через «`scale=1/β`» и θ лишь отражает β , но не является им).

³² В разных источниках отличаются как обозначения, так и варианты представления формул. Рисунок намеренно не адаптирован, чтобы читатель данного материала сам учился адаптироваться к разнородности материала. Это важный навык для аналитика данных. Вернитесь также к уже рассмотренным графикам других распределений и сравните их с материалами из сети Интернет. Например, на ресурсе «Википедия» много материалов с полезными графиками.

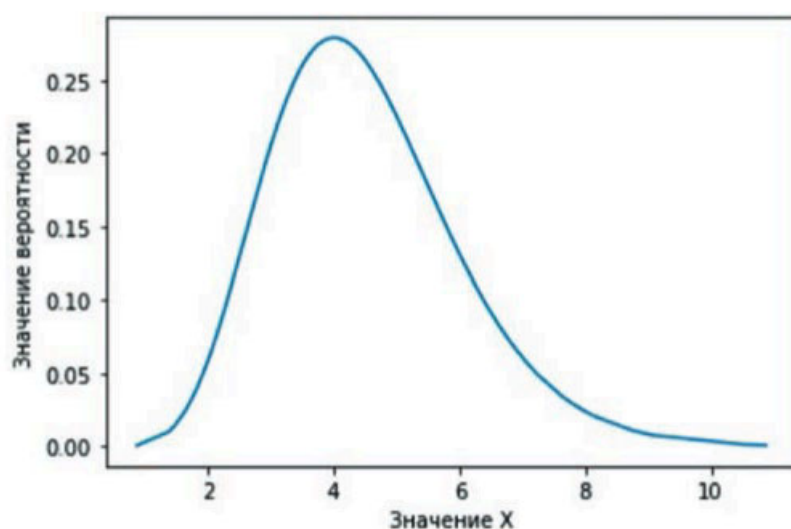
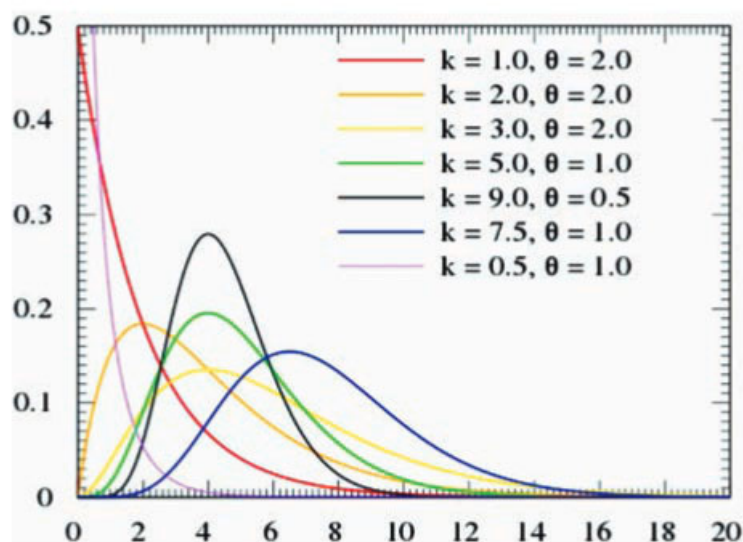


Рис. 29. Вывод плотности вероятности (PDF) для гамма-распределения

Рис. 30. PDF для гамма-распределения
(с разными значениями параметров)

При этом во втором случае (когда гамма-распределение сводится к показательному распределению), наш β -параметр является λ -параметром формулы (9) (формула плотности вероятности показательного распределения)³⁵.

Говоря более конкретно и более простым языком, приведем пример, на рис. 30 график с $k = 1$ — это распределение Эрланга пер-

³⁵ То есть, к сожалению, снова некоторая путаница обозначений.

вого порядка и показательное распределение с $\lambda = 0.5$ одновременно ($a=1$, $\text{beta}=0.5$, $\text{scale}=1/\text{beta}$).

Сравните «`stats.expon.pdf(xvalues, loc=0, scale=1/0.5)`» с «`gamma.pdf(xvalues, a=1, loc=0, scale=(1/0.5))`». Сделать это можно при помощи кода, который будет представлен далее.

Графики совпадут и будут как на рисунке после кода (рис. 31). Обратите внимание, что в коде специально сдвинут один график относительно другого на 0.1 по горизонтальной оси и на 0.01 — по вертикальной. Сдвиг добавлен только с той целью, чтобы можно было увидеть оба графика, иначе один бы перекрыл полностью другой, так как они полностью совпадают, потому что гамма-распределение полностью «свелось» к экспоненциальному.

```
---in↓---
a = 1
beta = 0.5
scale = 1 / beta

rng = default_rng(seed=1234)
xvalues_gamma = gamma.rvs(size=300, a=a, scale=scale,
random_state=rng)
xvalues_gamma = np.sort(xvalues_gamma) # сортируем
по возрастанию

pdf_gamma = gamma.pdf(xvalues_gamma, a, loc=0, scale=scale)

plt.title("Функция плотности для гамма и показательного
распределений")

scale = 2 # масштаб
loc = 0 # положение

rng = default_rng(seed=1234)
xvalues_expon = expon.rvs(size=100, loc=loc, scale=scale,
random_state=rng)
xvalues_expon = np.sort(xvalues_expon) # сортируем
по возрастанию
```

```
pdf_expon = stats.expon.pdf(xvalues_expon, loc, scale) #
функция плотности вероятности

plt.xlabel("Значение X")
plt.ylabel("Значение вероятности")
plt.plot(xvalues_expon+0.1, pdf_expon+0.01, label="pdf_
expon")
plt.plot(xvalues_gamma, pdf_gamma, label="pdf_gamma")
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.15))
# показать названия
plt.show() # показать график
---↑in_out↓---
#часть вывода представлена на рисунке ниже (Рис. 31)
---↑out---
```

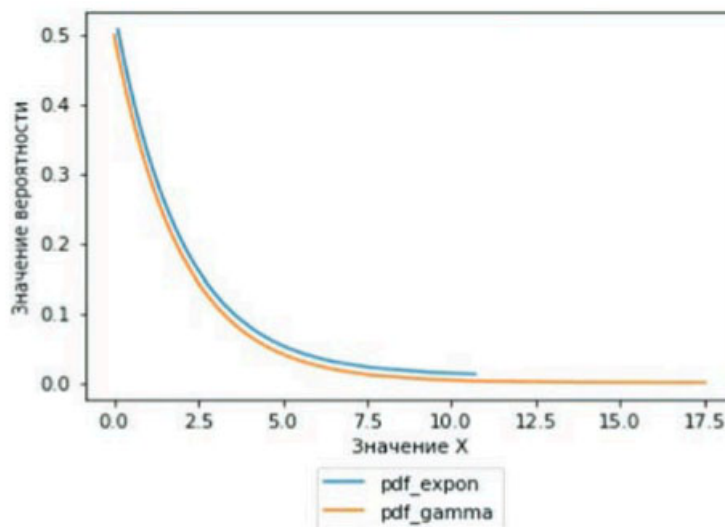


Рис. 31. PDF гамма-распределения,
сведенного к показательному распределению

Проблема отсутствия явно указанных параметров формы

В примере ниже продемонстрировано наглядное влияние отсутствия указания сдвига и масштаба в явном виде (также актуально и для иных параметров формы).

```
---in↓---
x_rvs = norm.rvs(size=30)
x_rvs = np.sort(x_rvs)
```

```

print('стандартное отклонение)>, x_rvs.std())
print('разброс (дисперсия)',x_rvs.var())
print('среднее',x_rvs.mean())
print('медиана',np.median(x_rvs))

#График PDF
my_cdf=norm.pdf(x_rvs, loc=x_rvs.mean(), scale=x_rvs.std())
plt.plot(x_rvs, my_cdf, label='Сдвиг и масштаб указан
корректно')
my_cdf = norm.pdf(x_rvs, scale=x_rvs.std())
plt.plot(x_rvs, my_cdf, label='Сдвиг не указан
(по умолчанию сдвиг=0)')
plt.legend()
plt.grid(linestyle='--') #добавить пунктирную сетку
plt.show() #показать график
---↑in_out↓---
стандартное отклонение) 1.1932269699295412
разброс (дисперсия) 1.4237906017672344
среднее 0.11498667296324895
медиана 0.402552621453243
#часть вывода представлена на рисунке ниже (Рис. 32)
---↑out---
```

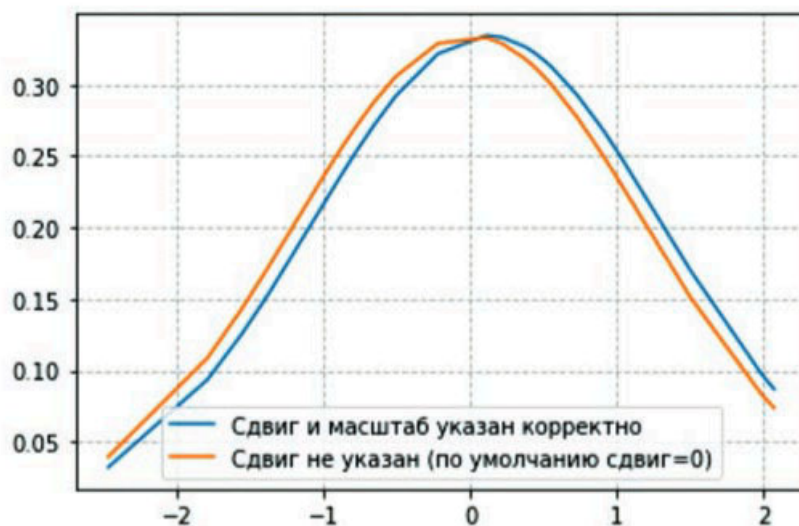


Рис. 32. Отсутствие явного указания параметров формы и положения

Как видно на рисунке ниже (рис. 32), графики функций плотности различаются, хотя построены на одном и том же ряде значе-

ний. Это происходит, потому что без указания параметров формы и сдвига используются значения по умолчанию и строится график для них (т.е. со стандартным сдвигом и масштабом).

Обратите внимание, что ваши данные применяются лишь для задания диапазона значений по осям, а форма графика будет идеальной, потому что рассчитывается по формуле той функции распределения, которую вы используете.

Как обойти данное ограничение, если заранее неизвестно, по какому закону распределен ряд значений? Рассмотрим далее.

Анализ принадлежности ряда значений к конкретному закону распределения

В примере ниже для набора случайно распределенных величин продемонстрирована некоторая последовательность действий как один из возможных подходов определения закона распределения.

```
---in↓---
def my_calc_cdf(x):
    x = np.sort(x)
    my_cdf = (
        1.0 * np.arange(len(x)) / (len(x) - 1)
    ) # calculate the proportional values of samples
    # print (x)
    # print (my_cdf)
    return my_cdf

x = np.arange(-3, 3.5, 0.5)
x_rvs = norm.rvs(size=300, loc=x.mean(), scale=x.std())
x_rvs = np.sort(x_rvs)

# График CDF для np.arange(-3, 3.5, 0.5) и np.arange(-3,
3.5, 0.5)+np.arange(1, 1.5, 0.01)
my_cdf = my_calc_cdf(x)
plt.plot(x, my_cdf, label='Для данных np.arange(-3, 3.5, 0.5)
("вручную" my_calc_cdf)')

x = np.append(x, np.arange(1, 1.5, 0.01)) # увеличим
количество значений.
```



```

my_cdf = my_calc_cdf(x)
plt.plot(x, my_cdf, label='Для данных np.arange(-3, 3.5,
0.5)+np.arange(1, 1.5, 0.01) ("вручную" my_calc_cdf)',)

plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.05))
plt.grid(linestyle="--") # добавить пунктирную сетку
plt.title("График CDF для np.arange(-3, 3.5, 0.5)
и np.arange(-3, 3.5, 0.5)+np.arange(1, 1.5, 0.01)")
plt.show() # показать график

# График CDF для нормального распределения
my_cdf = my_calc_cdf(x_rvs)
plt.plot(x_rvs, my_cdf, label='Для действительно случайно
распределенных величин x_rvs ("вручную" my_calc_cdf)',)

my_cdf = norm.cdf(x_rvs, loc=x_rvs.mean(), scale=x_rvs.std())
plt.plot(x_rvs, my_cdf, label="Для действительно случайно
распределенных величин x_rvs (при помощи scipy norm.cdf)",)

plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.05))
plt.grid(linestyle="--") # добавить пунктирную сетку
plt.title("График CDF для нормального распределения")
plt.show() # показать график

# График PPF (ОБРАТНЫЙ К CDF)
percentel_arr = np.linspace(0, 100, len(x_rvs))
my_ppf = norm.ppf(x_rvs, loc=x_rvs.mean(), scale=x_rvs.std())
plt.plot(x_rvs, my_ppf, label="Для действительно случайно
распределенных величин x_rvs (при помощи scipy norm.ppf)",)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.05))
plt.grid(linestyle="--") # добавить пунктирную сетку
plt.title("PPF-функция для получения значения процентиля
(обратная к CDF)\наналогична графику выше, но оси
перевернуты")
plt.show() # показать график
---↑in_out↓---
#вывод представлен на рисунках ниже (Рис. 33, Рис. 34, Рис. 35)
---↑out---

```

Имея некоторые данные, вы всегда можете «вручную» построить, например, график CDF (или по сути график процентилей, пер-

вый график из вывода кода, или рис. 33) при помощи своей собственной функции `my_calc_cdf()`, которая просто сгенерирует проценти́ли с некоторым константным шагом в количестве, равном количеству ваших значений.

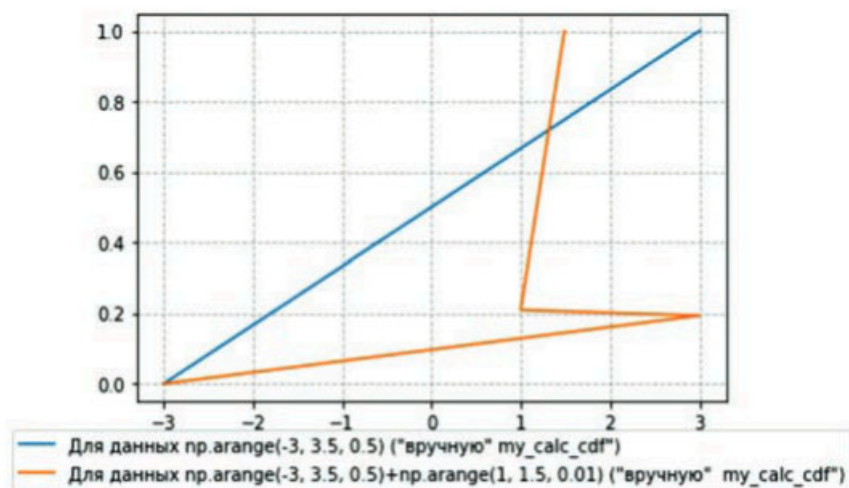


Рис. 33. CDF для рядов, для которых неизвестен (условно) закон распределения

Это просто и правильно, однако данные необходимо предварительно отсортировать, например, при помощи `np.sort()`.

После получения CDF для наших данных необходимо сравнить его с CDF того закона распределения, с которым мы предполагаем установку сходства. Например, в данном случае выведем CDF нормального закона (второй выведенный примером рисунок, или рис. 34) и сравним его с CDF того ряда, анализ которого нам интересен (первый выведенный кодом рисунок, или рис. 33).

Как видим, графики CDF для обоих рядов из первого, выведенного примером рис. 33, не похожи на графики из второго (рис. 34). Делаем вывод, что изучаемые ряды не имеют CDF, схожую с CDF нормального закона, а значит, возможно, в целом не имеют с нормальным законом распределения ничего общего. Возможно, было проще сравнить графики распределения с целью получения этого же вывода, но мы намеренно пошли сложным путем.

Чтобы вспомнить и закрепить, что такое `ppf` (функция получения проценти́лей) и как она работает, в примере выше также выводится третий график (рис. 35), представляющий полный аналог

второго графика из вывода данного примера (рис. 34), но оси инвертированы. Это происходит, потому что `ppf` является обратной по отношению к CDF функцией.

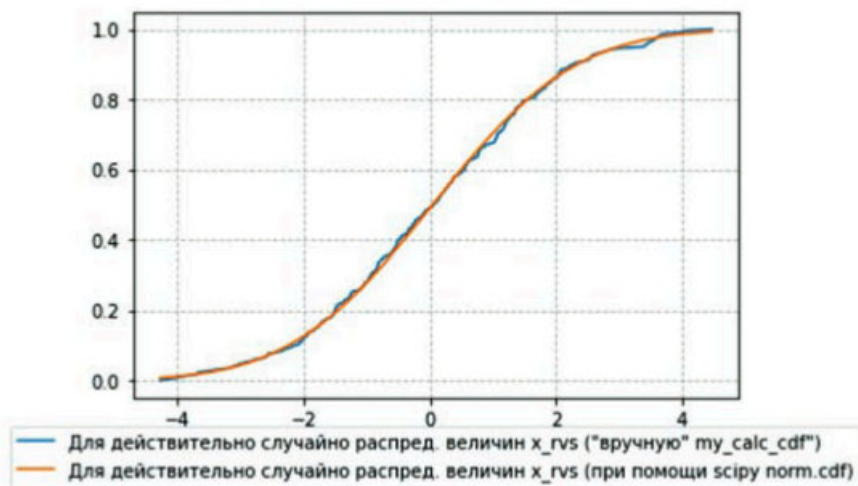


Рис. 34. CDF для нормального закона распределения

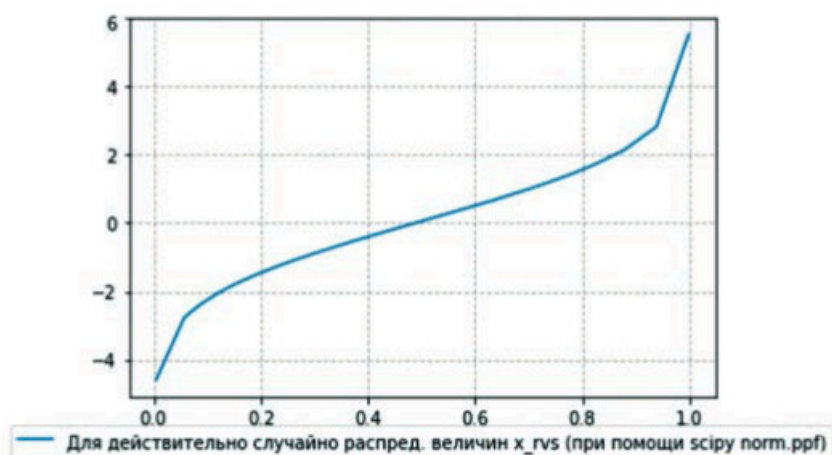


Рис. 35. PPF для нормального закона распределения

Куртозис и сдвиг (моменты и прочие понятия статистики)

В `scipy` доступны функции и методы для получения многих других, кроме уже рассмотренных, параметров статистики. В примере ниже представлено вычисление куртозиса (острота пика в середине) и перекоса (значений относительно медианы).

```
---in↓---
from scipy.stats import kurtosis, skew

x = norm.rvs(size=10)
x = np.sort(x)

my_cdf = norm.pdf(x, loc=x.mean(), scale=x.std())
kurt=kurtosis(x) #куртозис, или коэффициент эксцесса
(острота пика)
skew=skew(x) # перекоc
mean, var= norm.stats(loc=x.mean(), scale=x.std(),
moments='mv')
print('среднее ', mean)
print('разброс (дисперсия) ', var, 'или', x.var())
print('перекоc ', skew)
print('куртозис (острота пика)', kurt)
plt.plot(x, my_cdf)
plt.grid(linestyle='--') #добавить пунктирную сетку
plt.show() #показать график
---↑in_out↓---
среднее -0.04117791641498889
разброс (дисперсия) 0.6581657734136146 или
0.6581657734136146
перекоc 0.38184612969060433
куртозис (острота пика) -1.0715797870783699
#часть вывода представлена на рисунке ниже (рис. 36)
---↑out---
```

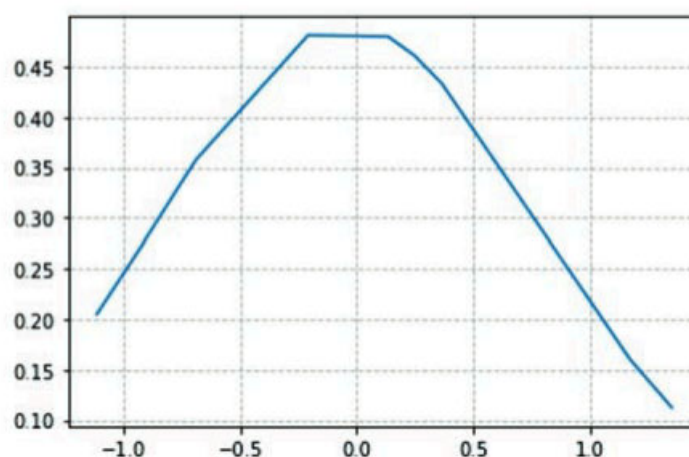


Рис. 36. CDF для нормального закона распределения

Диаграмма размаха

Графики CDF, PDF и др. просты (строятся по всего по двум рядам значений — значениям случайных величин и соответствующим им значениям функции, например PDF), но не компактны, поэтому в описательной статистике очень часто используется построение диаграмм размаха (англ. *box-and-whiskers diagram* или просто *box plot* — «ящик с усами»).

Диаграммы размаха в удобной форме показывают медиану (или, если нужно, среднее), нижний и верхний квартили, минимальное и максимальное значения выборки и выбросы. Расстояния между различными частями ящика также позволяют определить степень разброса (дисперсии) и асимметрии данных.

Несколько таких «ящиков» можно нарисовать бок о бок, чтобы визуально сравнивать одно распределение с другим; их можно располагать как горизонтально, так и вертикально.

Подробно изучить компоненты диаграммы размаха вы можете в источнике [41] (но основные положения представлены графически на рис. 37), а все ее параметры — в документации той библиотеки, при помощи которой вы планируете их строить.

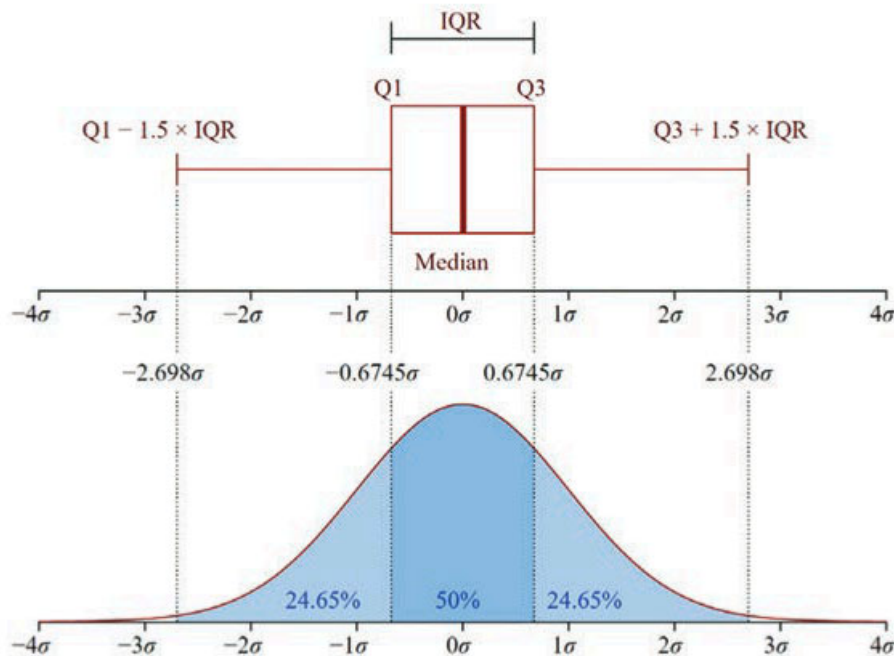


Рис. 37. Графический смысл диаграммы размаха на примере PDF

На рис. 37 σ (сигма) — это среднеквадратичное отклонение, а Q1 и Q3 — первый и третий квартили.

Далее будет представлен пример вывода диаграммы размаха поверх функции плотности вероятности для ряда с нормальным распределением. Попробуйте, используя данный пример, закрепить знания, уже полученные на занятиях по статистике. В этом вам поможет текстовый вывод данного примера (значение медианы, процентилей и др.). Единственное, что не было рассмотрено, — это IQR.

IQR (англ. *Interquartile Range*) — это межквартильный диапазон. Он определяется как разница между 75-м и 25-м процентилями данных. IQR является примером усеченного оценщика³⁴, определяемого как 25-процентильный усеченный диапазон, который повышает точность статистики набора данных за счет снижения вклада отдаленных точек. Величина IQR определяет размер прямоугольника диаграммы размаха (в примере ниже — ширину).

```
---in↓---
from scipy.stats import iqr

x_rvs = norm.rvs(size=200)
x_rvs = np.sort(x_rvs)

my_pdf=norm.pdf(x_rvs, loc=x_rvs.mean(), scale=x_rvs.var())

print("стандартное отклонение", x_rvs.std())
print("размах (дисперсия)", x_rvs.var())
print("среднее арифметическое", x_rvs.mean())
print("медиана", np.median(x_rvs))
iqr = iqr(x_rvs)
print("iqr", iqr)
print("25-й процентиль", np.percentile(x_rvs, 25))
print("75-й процентиль", np.percentile(x_rvs, 75))
print('25-й процентиль + 1.5*IQR', np.percentile(x_rvs, 25)
- 1.5 * iqr)
```

³⁴ В статистике **усеченный оценщик** — это оценщик, полученный из другого оценщика путем исключения некоторых экстремальных значений (выбросов) с целью повышения степени истинности статистики.


```

print('75-й процентиль - 1.5*IQR', np.percentile(x_rvs, 75)
      - 1.5 * iqr)
print('"Интеграл pdf (должен стремиться к 1, если все
в порядке)"', np.trapz(my_pdf, x_rvs))

# График PDF
plt.boxplot(x_rvs, vert=False, positions=[my_pdf.max().
round(3), ],)
plt.plot(x_rvs, my_pdf, label="Плотность вероятности (pdf)")
plt.legend()
plt.grid(linestyle="--") # добавить пунктирную сетку

plt.show() # показать график
---↑in_out↓---
стандартное отклонение 0.9913010241771213
размах (дисперсия) 0.9826777205346096
среднее арифметическое 0.21235706928827278
медиана 0.263603213186545
iqr 1.3747224311496087
25-й процентиль -0.41303097300964947
75-й процентиль 0.9616914581399593
25-й процентиль + 1.5*IQR -2.4751146197340623
75-й процентиль - 1.5*IQR -1.1003921885844536
"Интеграл pdf (должен стремиться к 1, если все
в порядке)" 1.0033805670962272
#часть вывода представлена на рисунке ниже (рис. 38)
---↑out---

```

«Усы ящика» (длины горизонтальных линий слева и справа от прямоугольника) по умолчанию равны значению $IQR \times 1,5$. То есть сам IQR занимает 25 % всего диапазона вправо и влево от середины (медиана отмечена чертой в центре прямоугольника, суммарно IQR занимает 50 %), а «усы» — почти всю оставшуюся часть значений.

Любые точки данных, которые попадают внутрь «усов», называются **мягкими выбросами**. Точки, которые выходят за пределы «усов», называются **экстремальными выбросами** (и от них обычно избавляются).

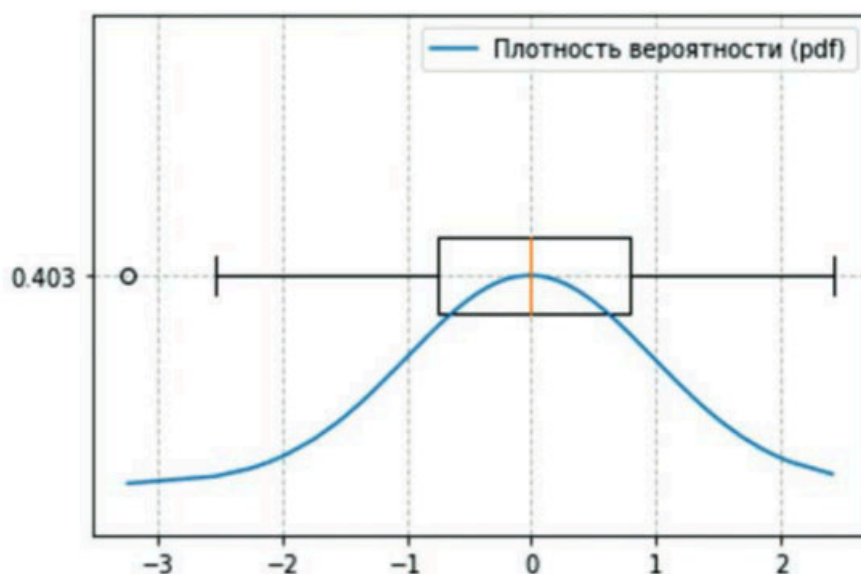


Рис. 38. Диаграмма размаха на графике плотности вероятности

Примеры статистики из жизни³⁵

Пример с временем доставки

Рассмотрим реальный пример из жизни: статистика времени доставки пиццы. Сначала произведем все импорты и настройки:

```
---in↓---
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns # statistical data visualization
from scipy import stats

sns.set()
from pylab import rcParams

rcParams["figure.figsize"] = 10, 6 # установка размеров
графиков
%config InlineBackend.figure_format = 'svg'
```

³⁵ Представленный в текущем разделе материал взят из русскоязычного источника [35]. Выражаем его автору (Семену Лукашевскому) большую благодарность как за материал, так и за согласие на его использование в данной книге. Материал представлен практически в оригинальном виде, с минимальными изменениями.

```
np.random.seed(42) # начальные условия для генератора
случайных чисел
---↑in_out↓---
---↑out---
```

Теперь давайте представим, что в течение года мы заказываем пиццу на дом, при этом мы каждый раз смотрим на настенные стрелочные часы, отмечая время, которое проходит между заказом и доставкой, целым количеством минут. Тогда накопленные за год данные о доставке пиццы могли бы выглядеть так:

```
---in↓---
norm_rv = stats.norm(
    loc=30, scale=5
) # генерируем значения с размахом "30±5",
распределенные по нормальному закону распределения
samples = np.trunc(norm_rv.rvs(365))
samples[:30]
---↑in_out↓---
array([32., 29., 33., 37., 28., 28., 37., 33., 27., 32.,
       27., 27., 31.,
        20., 21., 27., 24., 31., 25., 22., 37., 28., 30.,
       22., 27., 30.,
        24., 31., 26., 28.])
---↑out---
```

Теперь выясним среднее время доставки пиццы и его средне-квадратическое отклонение:

```
---in↓---
samples.mean(), samples.std()
---↑in_out↓---
(29.52054794520548, 4.77410133275075)
---↑out---
```

Можно сказать, что время доставки пиццы занимает около 30 ± 5 мин. Посмотрим на то, как распределены данные:

```
---in↓---
sns.histplot(x=samples, discrete=True);
```

```
---↑in_out↓---  
#вывод представлен на рисунке ниже (рис. 39)  
---↑out---
```

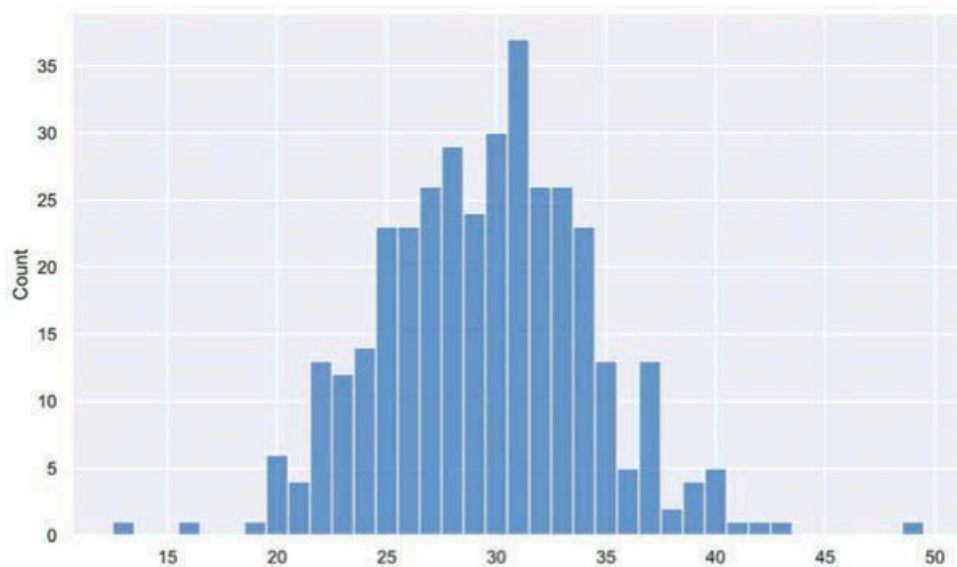


Рис. 39. Распределение данных

Глядя на такой график (рис. 39), мы вполне можем допустить, что время доставки пиццы имеет нормальный закон распределения с параметрами $\mu=30$ (среднее) и $\sigma=5$ (среднеквадратичное отклонение).

Посмотрим схожесть полученного распределения также с гамма, бета и треугольным законом распределения:

```
---in↓---  
norm_rv = stats.norm(loc=30, scale=5)  
beta_rv = stats.beta(a=5, b=5, loc=14, scale=32)  
gamma_rv = stats.gamma(a=20, loc=7, scale=1.2)  
tri_rv = stats.triang(c=0.5, loc=17, scale=26)  
  
x = np.linspace(10, 50, 300)  
  
sns.lineplot(x=x, y=norm_rv.pdf(x), color="r", label="norm")  
sns.lineplot(x=x, y=beta_rv.pdf(x), color="g", label="beta")  
sns.lineplot(x=x, y=gamma_rv.pdf(x), color="k", label="gamma")  
sns.lineplot(x=x, y=tri_rv.pdf(x), color="b", label="triang")
```



```
sns.histplot(x=samples, discrete=True, stat="probability",
alpha=0.2);
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 40)
---↑out---
```

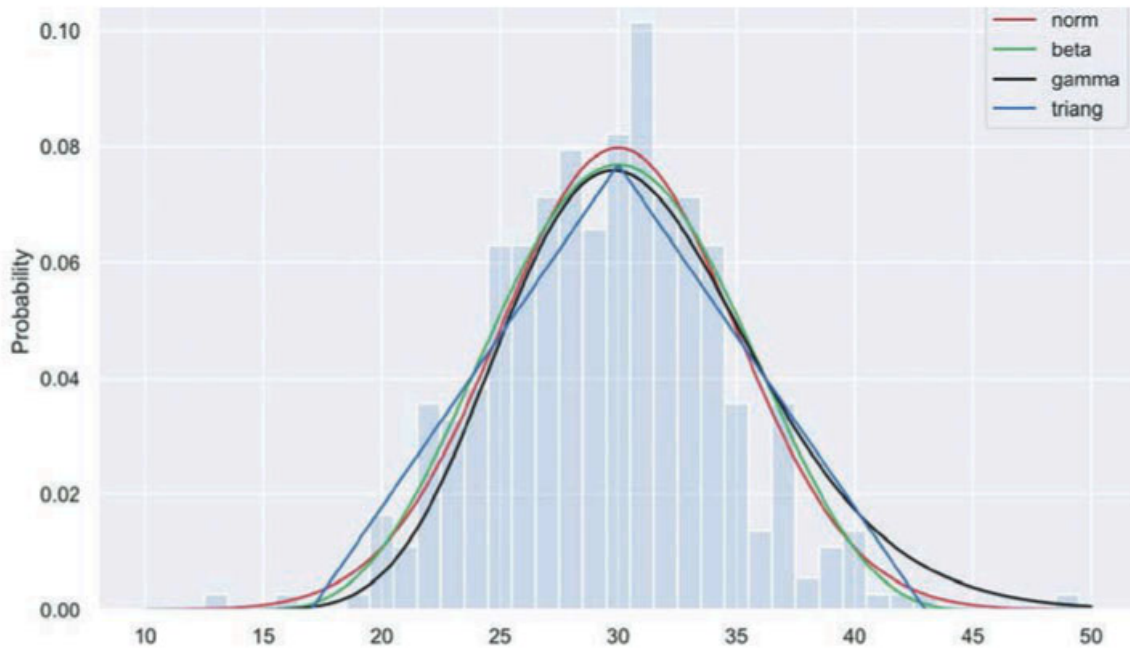


Рис. 40. Распределение данных в сравнении с разными типами распределения

К какому конкретно типу распределения относится время доставки пиццы из множества всевозможных?

Доставкой пиццы занимается человек, а сам процесс доставки сопровождается множеством случайных событий, которые могут произойти на его пути.

Конечно, мы можем придумать очень много таких событий, вплоть до самых невероятных. Тем не менее для нас важно, чтобы они описывались такими переменными, значения которых равновероятны, т.е. распределены по равномерному закону.

В качестве примера можно придумать переменную X_1 , которая будет описывать время ожидания зеленого света светофора на перекрестке. Если это время заключено в промежутке от нуля до четырех минут, то сегодня оно может составлять:


```
---in↓---
unif_rv = stats.uniform(loc=0, scale=4)
unif_rv.rvs()
---↑in_out↓---
0.8943833540778106
---↑out---
```

А завтра, послезавтра и послепослезавтра это время может быть равно:

```
---in↓---
unif_rv.rvs(size=3)
---↑in_out↓---
array([3.85289016, 0.0486179 , 3.87951531])
---↑out---
```

Теперь представим, что таких переменных 15, и значение каждой из них вносит свой вклад в общее время доставки, потому что эти события могут складываться. Если мы придумали всего 15 случайных переменных: X_1, X_2, \dots, X_{15} , то можно сказать, что общее время доставки является их суммой и тоже является случайной величиной, которую можно обозначить буквой Y (формула (12)):

$$Y = X_1 + X_2 + \dots + X_{15} = \sum_{i=1}^{15} X_i. \quad (12)$$

В коде ниже мы получим 10 тыс. сумм, каждая из которых является суммой всех 15 равномерно распределенных (`unif_rv`) случайных величин:

```
---in↓---
Y_samples = [unif_rv.rvs(size=15).sum() for i in range(10000)]
sns.histplot(x=Y_samples);
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 41)
---↑out---
```

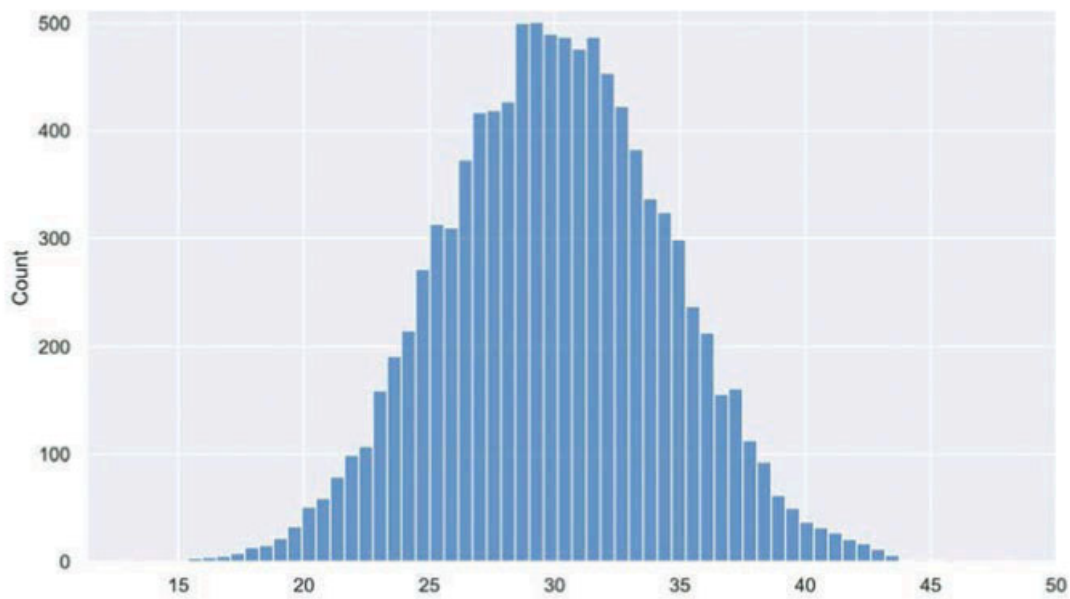


Рис. 41. Распределение 10 тыс. сумм 15 случайных величин

Распределение суммы случайных переменных стремится к нормальному распределению при увеличении количества слагаемых в этой сумме.

Конечно, пример с доставкой пиццы не совсем корректен для демонстрации предельной теоремы, потому что все события, которые мы придумали, носят условный характер: это означает, что они могут входить в сумму в самых разных комбинациях. Например, сегодня время доставки задавалось как в формуле (13), а завтра это время может задаваться как в формуле (14):

$$Y = X_3 + X_5 + X_7 + X_{11}; \quad (13)$$

$$Y = X_1 + X_3 + X_9 + X_{10} + X_{13} + X_{15}. \quad (14)$$

Будет ли теперь Y распределена по нормальному закону? Учитывая, что сумма нормально распределенных величин тоже имеет нормальное распределение, можно дать утвердительный ответ. Именно поэтому, когда мы взглянули на распределение 365 значений времени доставки, мы практически сразу решили, что перед нами нормальное распределение, даже несмотря на то, что оно во все не похоже на идеальный «колокол».

Z-значения

Допустим, по прошествии года у нас появился новый сосед, и он так же, как и мы, решил ежедневно заказывать пиццу. И вот после трех дней мы наблюдаем, как этот сосед обвиняет доставщика в слишком долгом ожидании заказа. Мы решаем поддерживать доставщика и говорим, что в среднем время доставки занимает 30 ± 5 мин, на что наш сосед отвечает, что все три раза он ждал больше 40 мин, а это определенно больше 35 мин.

Почему наш сосед так уверен в долгой доставке? И вообще, оправдана ли его уверенность? Очевидно он, как и некоторые люди, думает, что 30 ± 5 мин означает, что доставка может длиться 27, 31, даже 35 мин, но никак не 23 или 38 мин. Однако мы заказывали пиццу 365 раз и знаем, что доставка может длиться и 20, и даже 45 мин. А фраза 30 ± 5 мин означает лишь то, что какая-то значительная часть доставок будет занимать от 25 до 35 мин.

Зная параметры распределения, мы даже можем смоделировать несколько тысяч доставок и предположить величину этой части:

```
---in↓---
N = 5000
t_data = norm_rv.rvs(N)
t_data[(25 < t_data) & (t_data < 35)].size / N
---↑in_out↓---
0.7008
---↑out---
```

Где-то две трети значений укладываются в интервал от 25 до 35 мин. А сколько значений будет превосходить 40 мин? Отфильтруем значения и выведем дробную часть, сколько они занимают по отношению к общему числу значений:

```
---in↓---
t_data[t_data > 40].size / N
---↑in_out↓---
0.0248
---↑out---
```

Оказывается, только чуть более двух процентов значений превышают 40 мин. Но ведь сосед заказывал пиццу три раза подряд, и все три раза доставка длилась больше 40 мин. Может, сосед оказался просто очень везучим, ведь вероятность трех таких долгих доставок чрезвычайно мала и равна 0.02 в третьей степени:

```
---in↓---
0.02 ** 3
---↑in_out↓---
8.0000000000000001e-06
---↑out---
```

Для решения подобных задач обычно пользуются Z-значениями.

Z-оценка, или стандартизированная оценка (на англ. *z-score* или *standard score* соответственно) — это мера относительного разброса наблюдаемого или измеренного значения, которая показывает, сколько стандартных отклонений составляет его разброс относительно среднего значения. Это безразмерный статистический показатель, используемый для сравнения значений разной размерности или с разными шкалами измерений.

Вычислить Z-значение можно по следующей формуле (15):

$$Z = \frac{y - \mu}{\sigma}, \quad (15)$$

где y — это время доставки, т.е. какое-то конкретное значение случайной переменной Y , которая имеет нормальное распределение, а μ и σ ³⁶ — это математическое ожидание и среднеквадратическое отклонение соответственно, т.е. параметры распределения, и в нашем случае они равны 30 и 5 мин соответственно.

Давайте рассчитаем Z-значение для 40 мин (формула (16)):

$$Z = \frac{40 - 30}{5} = 2. \quad (16)$$

В числителе мы определили, на какую величину время доставки отличается от среднего времени доставки, далее мы поделили

³⁶ Ранее в данной книге для расчета μ и σ мы использовали методы `std()` и `var()` соответственно из библиотеки NumPy.

это значение на стандартное отклонение времени доставки. Но как интерпретировать данный результат и зачем вообще использовать Z-значение? Чтобы объяснить это, мы воспользуемся графиком:

```

---in↓---
fig, ax = plt.subplots()
x = np.linspace(norm_rv.ppf(0.001), norm_rv.ppf(0.999), 200)
ax.vlines(40.5, 0, 0.1, color="k", lw=2)
sns.lineplot(x=x, y=norm_rv.pdf(x), color="r", lw=3)
sns.histplot(x=samples, stat="probability", discrete=True);
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 42)
---↑out---
```

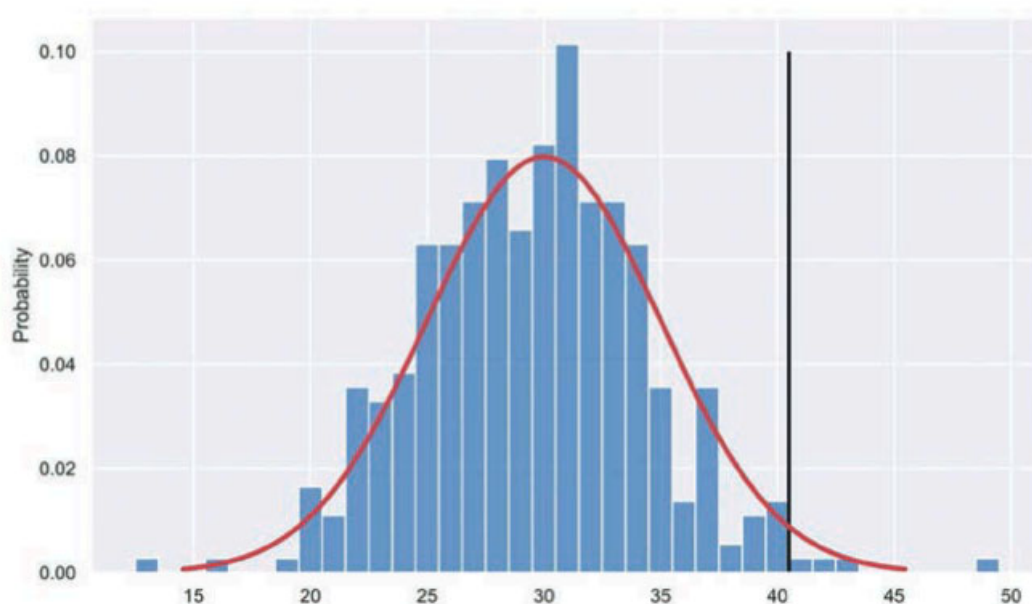


Рис. 42. Распределение 10 тыс. сумм 15 случайных величин

На данном графике (вывод примера выше, или рис. 42) мы изобразили гистограмму наших данных `samples`, но теперь высота прямоугольников показывает не количество вхождений каждого значения в выборку, а вероятность их появления в выборке. Или, говоря конкретнее, данный график отражает функцию распределения плотности вероятности, т.е. PDF.

Красной линией мы отрисовали «образцовую» функцию распределения плотности вероятности для значений времени доставки.

Выше мы пытались экспериментальным путем определить, какую долю составляют значения времени больше 40 мин. И данный график должен натолкнуть нас на мысль о том, что есть два способа сделать это.

Первый — экспериментальный, т.е. мы моделируем, скажем, 5 000 доставок, строим гистограмму и находим сумму высот прямоугольников, расположенных правее черной линии:

```

---in↓---
np.random.seed(42)
N = 10000
values = np.trunc(norm_rv.rvs(N))

fig, ax = plt.subplots()
v_le_41 = np.histogram(values, np.arange(9.5, 41.5))
v_ge_40 = np.histogram(values, np.arange(40.5, 51.5))
ax.bar(np.arange(10, 41), v_le_41[0] / N, color="0.8")
ax.bar(np.arange(41, 51), v_ge_40[0] / N)
p = np.sum(v_ge_40[0] / N)
ax.set_title("P(Y>40min) = {:.3}".format(p))
ax.vlines(40.5, 0, 0.08, color="k", lw=1);
---↑in_out↓---
#вывод представлен на рисунке ниже (Рис. 43)
---↑out---

```

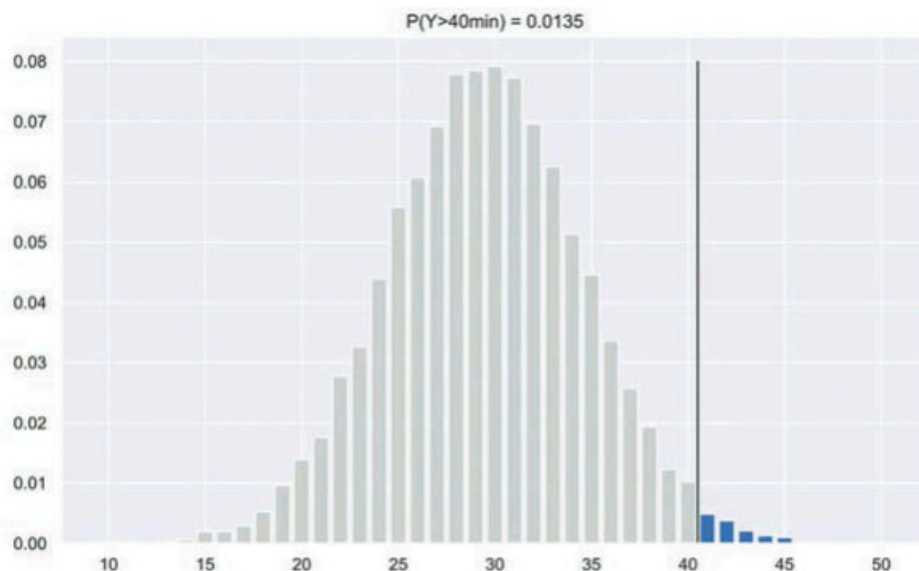


Рис. 43. Плотность вероятности для доставок, которые больше 40 мин

Второй путь — аналитический. Поскольку мы знаем, что значения доставки пиццы берутся из нормального распределения, то мы можем воспользоваться функцией распределения плотности вероятности нормального распределения.

Функция распределения плотности хороша тем, что площадь под ней всегда равна единице, и если нас интересует вероятность того, что величина примет значение больше или меньше указанного, то достаточно найти площадь под функцией, которая находится правее или левее этого значения. Например, вычислить и изобразить вероятность того, что время доставки будет длиться более 40 мин, можно так:

```
---in↓---
fig, ax = plt.subplots()

x = np.linspace(norm_rv.ppf(0.0001), norm_rv.ppf(0.9999), 300)
ax.plot(x, norm_rv.pdf(x))

ax.fill_between(x[x > 41], norm_rv.pdf(x[x > 41]),
np.zeros(len(x[x > 41])))
p = 1 - norm_rv.cdf(41)
ax.set_title("P(Y>40min) = {:.3}".format(p))
ax.hlines(0, 10, 50, lw=1, color="k")
ax.vlines(41, 0, 0.08, color="k", lw=1);
---↑in_out↓---
#вывод представлен на рисунке ниже (Рис. 44)
---↑out---
```

Значения вероятности в первом и втором случаях получились довольно близкими. Но необходимо отметить, что в первом случае мы использовали дискретную случайную величину, а во втором — непрерывную.

В первом случае мы считаем, сколько раз доставка составляла более 40 мин, т.е. сколько доставок длились 41, 42, 43 и т.д. мин.

Во втором случае мы считаем, сколько непрерывных величин оказалось правее значения 41.0.

Модуль SciPy позволяет создавать собственные распределения случайных величин, в руководстве к этой библиотеке есть

пример, в котором показано создание дискретно-нормального распределения. Но если задуматься, то наличие дискретных значений и использование непрерывных функций распределения — это своего рода неизбежность, ведь подавляющее большинство измерений носит дискретный характер: рост, вес, доход и т.д. Так что далее по этому поводу больше не будет приводиться замечаний.

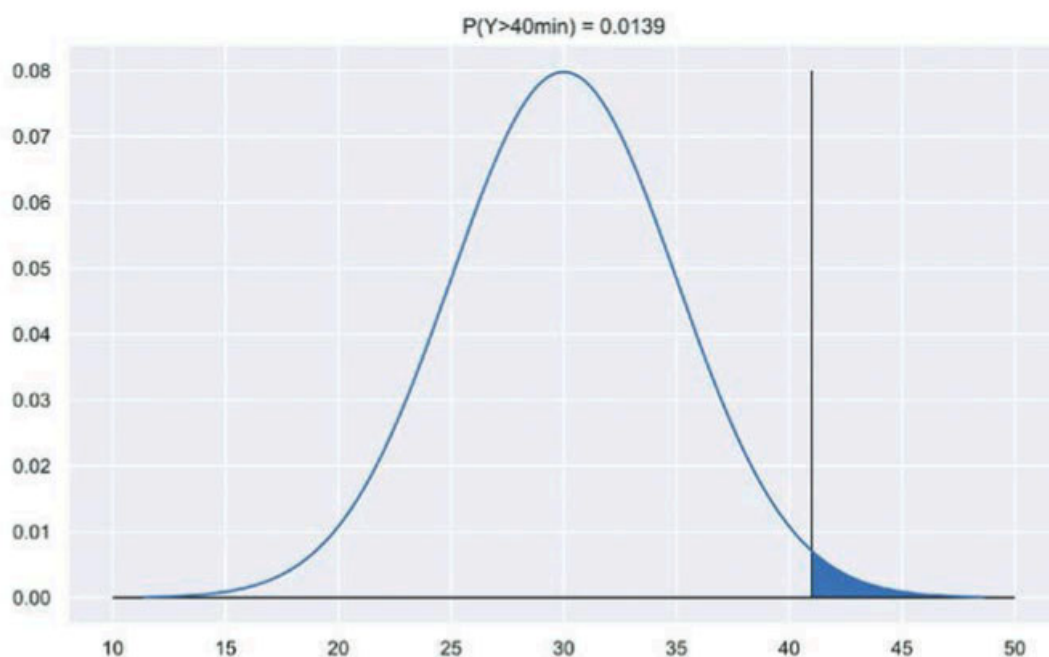


Рис. 44. Плотность вероятности для доставок, которые больше 40 мин

Пример с ростом народов

Вернемся к Z-значениям, но пример с доставкой пока отложим на второй план и временно перейдем к новому примеру.

Итак, допустим мы оказались в Средиземье и каким-либо образом выяснили, что рост хоббитов и гномов в сантиметрах распределен как $N(91; 8^2)$ и $N(134; 6^2)$. Если рост Фродо равен 99 см, а рост Гимли — 143 см, то как понять, чей рост более типичен среди представителей каждого народа? Чтобы выяснить это, мы можем изобразить функцию распределения плотности вероятности для каждого народа с отмеченными значениями, а заодно определить долю тех, кто превышает эти значения:


```

---in↓---
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

nrv_hobbit = stats.norm(91, 8)
nrv_gnome = stats.norm(134, 6)

for i, (func, h) in enumerate(zip((nrv_hobbit, nrv_gnome),
(99, 143))):
    x = np.linspace(func.ppf(0.0001), func.ppf(0.9999), 300)
    ax[i].plot(x, func.pdf(x))
    ax[i].fill_between(x[x > h], func.pdf(x[x > h]),
np.zeros(len(x[x > h])))
    p = 1 - func.cdf(h)
    ax[i].set_title("P(H > {} см) = {:.3}".format(h, p))
    ax[i].hlines(0, func.ppf(0.0001), func.ppf(0.9999),
lw=1, color="k")
    ax[i].vlines(h, 0, func.pdf(h), color="k", lw=2)
    ax[i].set_ylim(0, 0.07)
fig.suptitle("Хоббиты слева, гномы справа", fontsize=20);
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 45)
---↑out---

```

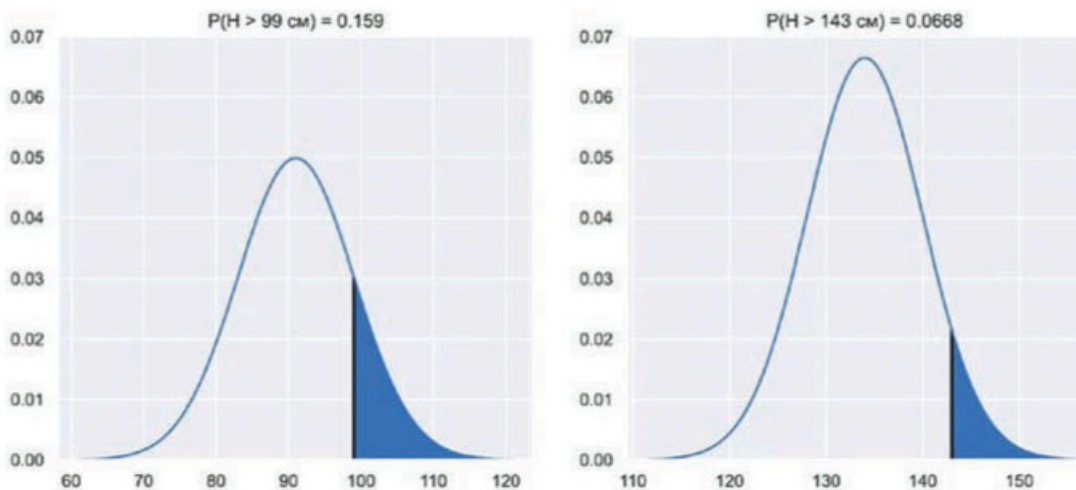


Рис. 45. Рост двух народов (хоббиты слева, гномы справа)

По этим графикам (вывод примера выше, рис. 45) можно предположить, что рост Фродо несколько ближе к вершине распределения, чем рост Гимли. А это значит, что вероятность встретить

хоббита с таким же ростом, как у Фродо, несколько больше вероятности встретить гнома с ростом, как у Гимли. Именно это и понимается под словом «типичность».

Выполнить сравнение типичности можно гораздо проще и нагляднее, если воспользоваться Z -значениями (формулы (17)–(18) и код ниже):

$$Z = \frac{99 - 91}{8} = 1; \quad (17)$$

$$Z = \frac{143 - 134}{6} = 1.5. \quad (18)$$

```

---in↓---
fig, ax = plt.subplots()
N_rv = stats.norm()
x = np.linspace(N_rv.ppf(0.0001), N_rv.ppf(0.9999), 300)
ax.plot(x, N_rv.pdf(x))
ax.hlines(0, -4, 4, lw=1, color="k")
ax.vlines(1, 0, 0.4, color="r", lw=2, label="Фродо")
ax.vlines(1.5, 0, 0.4, color="g", lw=2, label="Гимли")
ax.legend();
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 46)
---↑out---
```

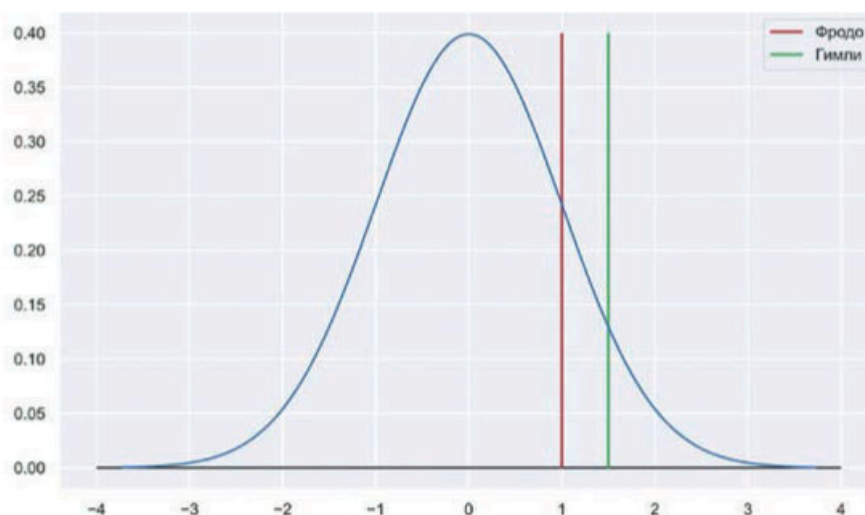


Рис. 46. Рост Фродо и Гимли

Огромным преимуществом Z -значений является то, что они стандартизированы (нормализованы)³⁷, т.е. преобразованы так, словно они взяты из стандартного нормального распределения $N(0; 1)$, именно поэтому два Z -значения нарисованы на фоне единственной кривой.

Однако в общем случае даже построение графиков вовсе не обязательно, потому что меньшие по модулю Z -значения обладают большей частотой появления. Одного предположения (формула (19)) уже достаточно для того, чтобы понять, какое из значений находится ближе к вершине распределения, и сделать соответствующие выводы:

$$|z_{\text{Фродо}}| < |z_{\text{Гимли}}| \quad (19)$$

Сравнение Z -значений нескольких величин из разных «нормальных» выборок с разными параметрами распределения возможно потому, что сами Z -значения измеряются в «сигмах», т.е. имеют размерность стандартного отклонения³⁸.

Имеет смысл отметить, что с математически строгой точки зрения вероятность получения какого-либо конкретного значения из непрерывного распределения стремится к нулю. В таких случаях всегда говорят о вероятности попадания этого значения в заданный интервал. Поэтому когда мы говорим, что Z -значение какой-либо величины позволяет оценить вероятность (частоту) ее появления (этой величины), то мы вовсе не выходим за границы здравого смысла.

В реальном мире мы никогда не работаем с действительно непрерывными величинами. Да, мы можем измерять рост в нанометрах, а время в микросекундах, но это все равно будут дискретные

³⁷ **Стандартизованная случайная величина** (или нормализованная) — это случайная величина, математическое ожидание которой равно нулю, а среднеквадратическое отклонение — единице. Или если случайная величина имеет математическое ожидание μ , а среднеквадратическое отклонение — σ , то соответствующая стандартизованная случайная величина имеет вид $X_{\text{стандартиз.}} = (X - \mu) / \sigma$. Распределение стандартизованной случайной величины называют «стандартным распределением».

³⁸ Это станет более очевидным, если еще раз взглянуть на знаменатель формулы Z -значения (формула (15)).

значения. Распределение дискретных величин может хорошо приближаться распределением величин непрерывных, и нам этого достаточно.

Пример с временем доставки (продолжение)

Теперь давайте снова вернемся к нашему соседу, который возмущен слишком долгой доставкой пиццы. Ранее мы вычислили Z -значение для 40 мин, и оно составило 2 (т.е. $Z_{40\text{мин}} = 2$, формула (16) на стр. 275). Пока у нас нет опыта, достаточного для того, чтобы сразу понять, много это или мало, то лучше изображать эти значения графически:

```
---in↓---
fig, ax = plt.subplots()
N_rv = stats.norm()
x = np.linspace(N_rv.ppf(0.0001), N_rv.ppf(0.9999), 300)
ax.plot(x, N_rv.pdf(x))
ax.hlines(0, -4, 4, lw=1, color="k")
ax.vlines(2, 0, 0.4, color="g", lw=2);
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 47)
---↑out---
```

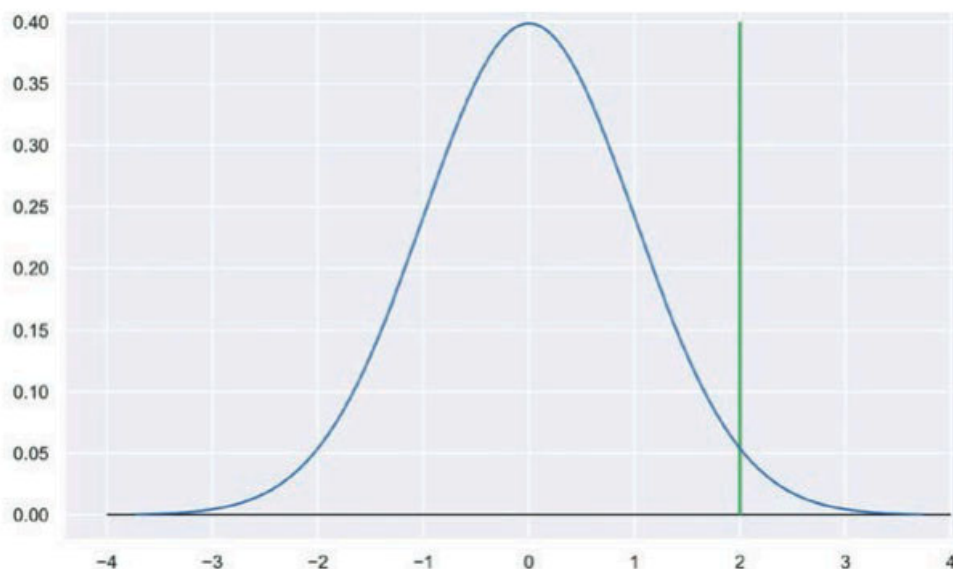


Рис. 47. $Z_{40\text{мин}} = 2$

Это значение находится справа от вершины на расстоянии 2σ (2 сигма, т.е. на расстоянии двух стандартных отклонений), что довольно много. Однако сосед утверждает, что ждал пиццу больше 40 мин три дня подряд. Возможно, он и не вычислял среднее время ожидания своей пиццы, но три значения — это уже статистика.

Характеристики генеральной совокупности называют параметрами, а характеристики выборки — статистиками.

Замеряя время доставки на протяжении 365 дней, мы сделали вывод о параметрах генеральной совокупности, т.е. времени всех возможных доставок пиццы, решив, что эти значения берутся из $N(30; 5^2)$. Зная распределение, мы можем поэкспериментировать. Например, наш сосед сделал всего три заказа, и по его ощущениям среднее время доставки было больше 40 мин. Попробуем повторить этот эксперимент 5 000 раз и посмотрим на то, как распределено среднее трех заказов:

```
---in↓---
sns.histplot(np.trunc(norm_rv.rvs(size=(N, 3))).
mean(axis=1), bins=np.arange(19, 42));
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 48)
---↑out---
```

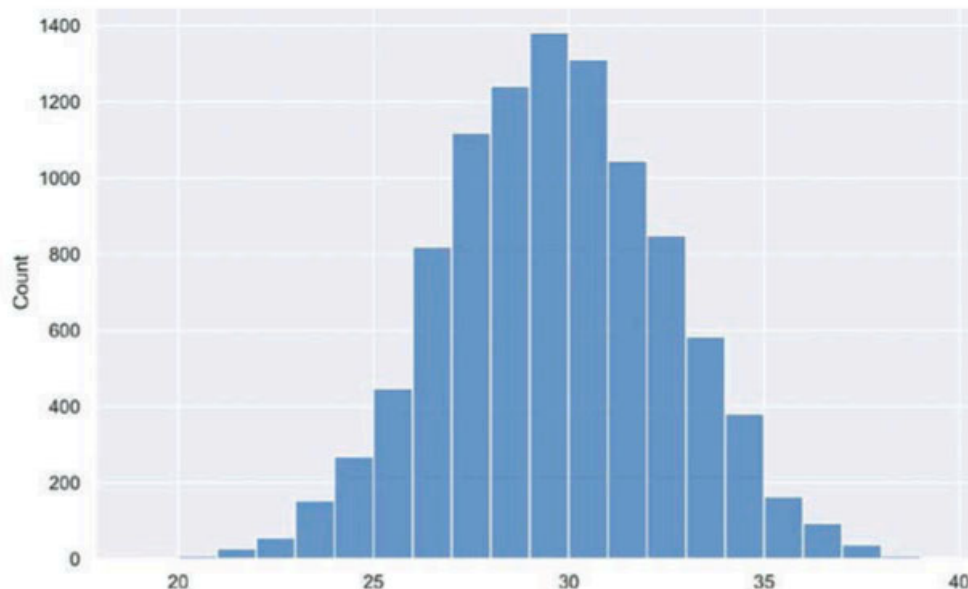


Рис. 48. Количество тех или иных сроков доставки

Судя по графику из вывода примера выше (рис. 48), получить среднее значение трех заказов большее 40 мин практически нереально (точнее, маловероятно на фоне общего числа других сроков доставки). Поэтому тут возможны два предположения:

- либо сосед врет;
- либо это обусловлено генеральной совокупностью, т.е. по какой-либо причине доставка и правда длится дольше обычного.

Вычисленное выше Z -значение для 40 мин ($Z_{40\text{мин}} = 2$) позволяет оценить долю (вероятность появления) значений, больших 40:

```
---in↓---
1 - N_rv.cdf(2)
---↑in_out↓---
0.02275013194817921
---↑out---
```

Поэтому не удивительно, что вероятность получить среднее время трех доставок большее 40 мин очень мала:

```
---in↓---
(1 - N_rv.cdf(2)) ** 3
---↑in_out↓---
1.1774751750476762e-05
---↑out---
```

Если каждое отдельное значение времени доставки мы можем оценить с помощью Z -значения, то для того, чтобы оценить вероятность среднего арифметического этих значений, нам нужно воспользоваться Z -статистикой (формула (20)):

$$Z = \frac{\bar{x} - \mu}{\frac{\sigma}{\sqrt{n}}}, \quad (20)$$

где \bar{x} — это среднее арифметическое значение нашей выборки; μ и σ — математическое ожидание и стандартное отклонение соответственно для генеральной совокупности; n — размер выборки.

Предположим, что мы сделали три заказа и среднее (арифметическое) значение оказалось равным 35 мин, тогда Z -статистика будет вычисляться так (формула (21)):

$$Z = \frac{35 - 30}{\frac{5}{\sqrt{3}}}. \quad (21)$$

Z -статистика, как и Z -значение, является стандартизированной величиной и также измеряется в сигмах, что позволяет использовать стандартное нормальное распределение для подсчета вероятностей (см. сноску 37 на стр. 282). Фактически мы задаемся вопросом: а какова вероятность того, что среднее значение времени трех доставок попадет в промежуток $[\mu - |\mu - \bar{x}|; \mu + |\mu - \bar{x}|]$?

В нашем случае этот промежуток выглядит как [25; 35] мин. Как и ранее, мы можем найти данную вероятность с помощью моделирования:

```
---in↓---
N = 10000
means = np.trunc(norm_rv.rvs(size=(N, 3))).mean(axis=1)
means[(means >= 25) & (means <= 35)].size / N
---↑in_out↓---
0.9241
---↑out---
```

Или, если представить графически:

```
---in↓---
N = 10000
fig, ax = plt.subplots()
means = np.trunc(norm_rv.rvs(size=(N, 3))).mean(axis=1)
h = np.histogram(means, np.arange(19, 41))
ax.bar(np.arange(20, 25), h[0][0:5] / N, color="0.5")
ax.bar(np.arange(25, 36), h[0][5:16] / N)
ax.bar(np.arange(36, 41), h[0][16:22] / N, color="0.5")
p = np.sum(h[0][6:16] / N)
ax.set_title("P(25min < Y < 35min) = {:.3}".format(p))
ax.vlines([24.5, 35.5], 0, 0.08, color="k", lw=1);
```



```

---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 49)
---↑out---
```

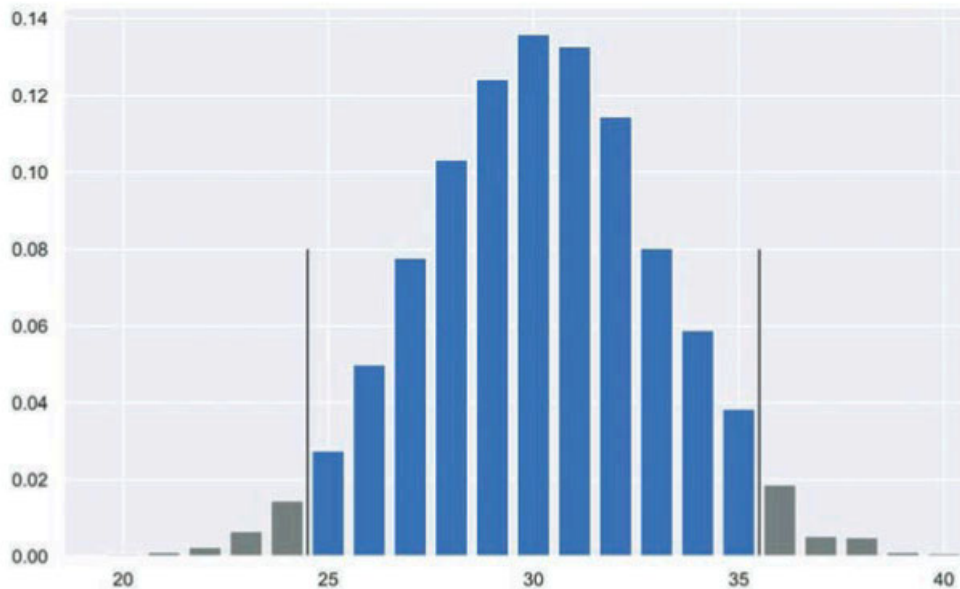


Рис. 49. Вероятность $P(25 \text{ мин} < Y < 35 \text{ мин}) = 0.916$

С другой стороны, мы можем вычислить ту же вероятность аналитическим способом на основе Z-значений:

```

---in↓---
x, n, mu, sigma = 35, 3, 30, 5
z = abs((x - mu) / (sigma / n ** 0.5))

N_rv = stats.norm()
fig, ax = plt.subplots()
x = np.linspace(N_rv.ppf(0.0001), N_rv.ppf(0.9999), 300)
ax.plot(x, N_rv.pdf(x))
ax.hlines(0, x.min(), x.max(), lw=1, color="k")
ax.vlines([-z, z], 0, 0.4, color="g", lw=2)
x_z = x[(x > -z) & (x < z)] # & (x<z)
ax.fill_between(x_z, N_rv.pdf(x_z), np.zeros(len(x_z)),
alpha=0.3)

p = N_rv.cdf(z) - N_rv.cdf(-z)
ax.set_title("P({:.3} < z < {:.3}) = {:.3}".format(-z, z, p));
```

```

---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 50)
---↑out---

```

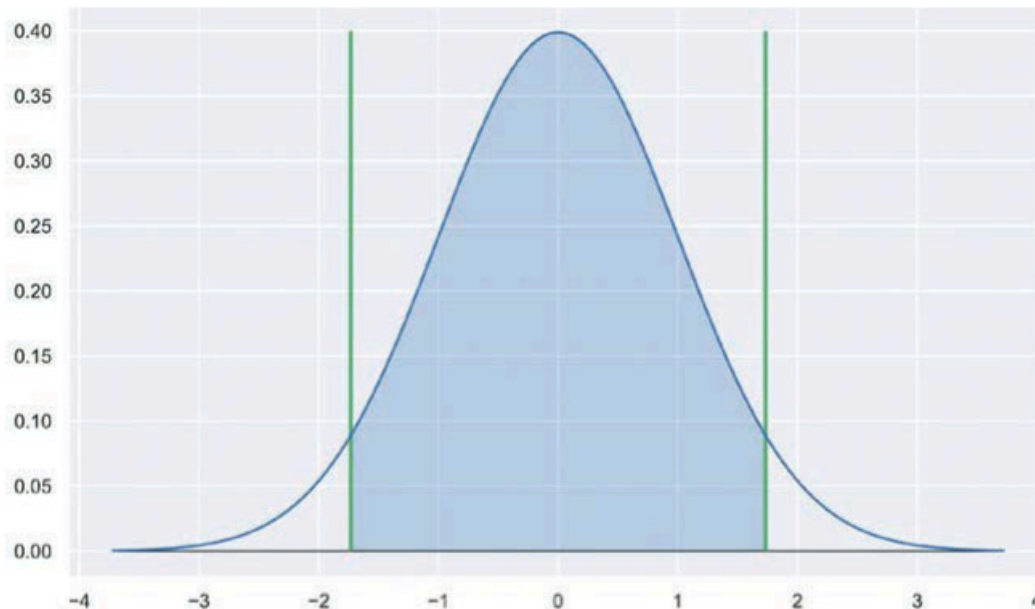


Рис. 50. Вероятность $P(1.73 < z < 1.73) = 0.917$

Обратите внимание на то, что Z -статистика зависит как от среднего выборки, так и от размера выборки n . Если мы закажем пиццу 5, 30 или 100 раз, то какова вероятность того, что среднее значение времени доставок окажется в интервале $[29; 31]$? Рассмотрим это аналитически:

```

---in↓---
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(12, 4))

for i, n in enumerate([5, 30, 100]):
    x, mu, sigma = 31, 30, 5
    z = abs((x - mu) / (sigma / n ** 0.5))

    N_rv = stats.norm()
    x = np.linspace(N_rv.ppf(0.0001), N_rv.ppf(0.9999), 300)
    ax[i].plot(x, N_rv.pdf(x))
    ax[i].hlines(0, x.min(), x.max(), lw=1, color="k")
    ax[i].vlines([-z, z], 0, 0.4, color="g", lw=2)

```

```

x_z = x[(x > -z) & (x < z)] # & (x<z)
ax[i].fill_between(x_z, N_rv.pdf(x_z),
np.zeros(len(x_z)), alpha=0.3)

p = N_rv.cdf(z) - N_rv.cdf(-z)
ax[i].set_title("n = {}, z = {:.3}, p = {:.3}".
format(n, z, p))
fig.suptitle(r"Z-статистики для $\bar{x} = 31$",
fontsize=20, y=1.02);
---↑in_out↓---
#Вывод представлен на рисунке ниже (рис. 51)
---↑out---

```

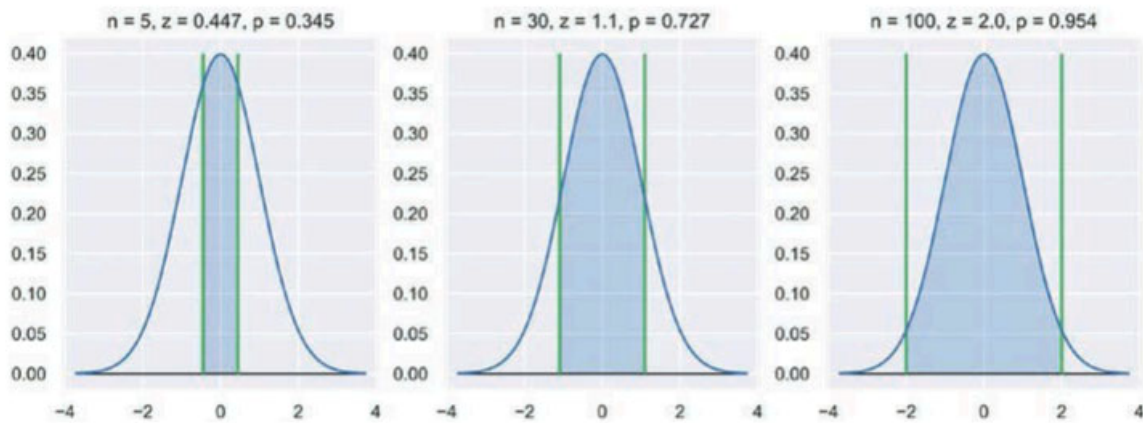


Рис. 51. Вероятности для из интервала [29; 31] для 5, 30 или 100 доставок

При 5 заказах среднее выборки попадает в интервал [29; 31] скорее случайно (см. вывод примера выше или рис. 51), чем систематически. При 30 заказах около четверти средних значений так и не войдут в заданный интервал. И только при 100 заказах мы можем быть более-менее обоснованно уверены в том, что отклонение среднего выборки от среднего генеральной совокупности не будет больше 1 мин.

С другой стороны, мы можем рассуждать и по-другому: если среднее генеральной совокупности равно 30 мин, то какова вероятность получить среднее выборки, равное 31 мин, если мы сделаем 5, 30 или 100 заказов?

Очевидно, что при $n = 5$ среднее выборки может отклоняться очень сильно, следовательно, вероятность получить $\bar{x} = 31$ мин очень высока. Но при $n = 100$ среднее выборки практически не от-

клоняется от среднего генеральной совокупности, поэтому получить случайным образом $\bar{x} = 31$ при $n = 100$ практически невозможно. Что это значит? А это значит, что если мы сделали 100 заказов и получили среднее время доставки, равное 31 мин, то скорее всего мы ошибаемся насчет того, что среднее генеральной совокупности равно 30 мин.

Но как быть с нашим соседом, ведь он сделал всего 3 заказа? Неужели он действительно прав? Судя по всему, да. Даже если среднее время ожидания составило 40 мин, то Z -статистика будет равна 3.81, а площадь под кривой будет практически равна 1:

```
---in↓---
x, n, mu, sigma = 41, 3, 30, 5
z = abs((x - mu) / (sigma / 3 ** 0.5))

N_rv = stats.norm()
fig, ax = plt.subplots()
x = np.linspace(N_rv.ppf(1e-5), N_rv.ppf(1 - 1e-5), 300)
ax.plot(x, N_rv.pdf(x))
ax.hlines(0, x.min(), x.max(), lw=1, color="k")
ax.vlines([-z, z], 0, 0.4, color="g", lw=2)
x_z = x[(x > -z) & (x < z)] # & (x<z)
ax.fill_between(x_z, N_rv.pdf(x_z), np.zeros(len(x_z)),
alpha=0.3)

p = N_rv.cdf(z) - N_rv.cdf(-z)
ax.set_title("P({:.3} < z < {:.3}) = {:.3}".format(-z, z, p));
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 52)
---↑out---
```

В свою очередь, это значит, что вероятность случайного отклонения среднего выборки из трех значений, взятых из генеральной совокупности с $\mu = 30$ (математическое ожидание) и $\sigma = 5$ (стандартное отклонение), более чем на 10 мин исчезающе мала. В этом случае мы можем сделать уже обоснованные предположения:

- либо наш сосед просто чрезвычайно «везучий» человек;
- либо доставка пиццы и вправду теперь выполняется дольше обычного.

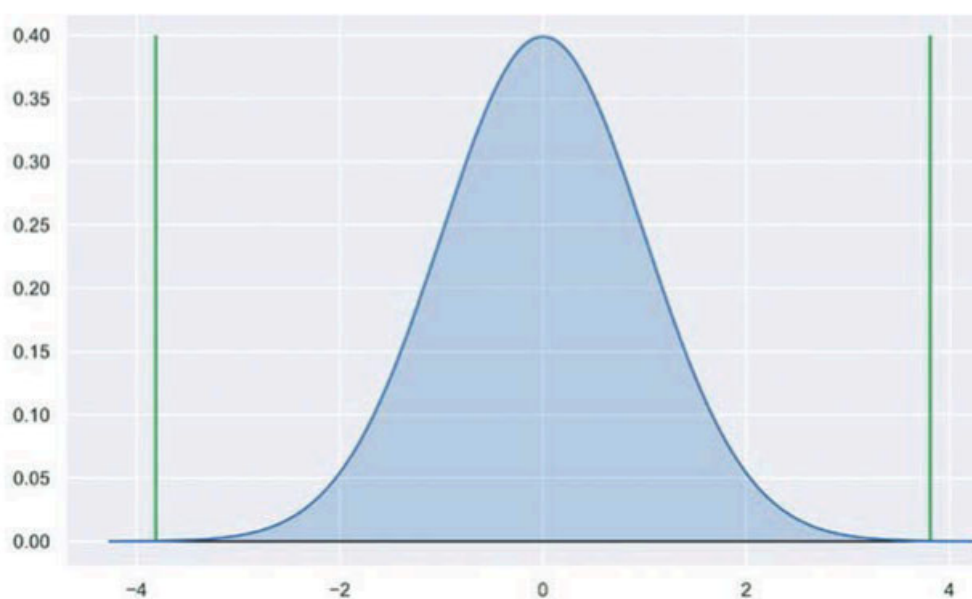


Рис. 52. Z-статистика при доставках, где средним является 40 мин

Что из этих пунктов является наиболее вероятным? Скорее всего, сосед прав насчет долгой доставки.

Р-значения

Мы смогли убедиться в том, что Z-статистика позволяет оценить вероятность того, что среднее выборки размером n , взятой из генеральной совокупности, попадет в заданный интервал значений. Это удобно тем, что позволяет сделать вывод о случайности полученного \bar{x} .

Чем меньше модуль значения Z-статистики, тем меньше достоверность среднего. Например, выше мы видели, что вероятность попадания \bar{x} при $n = 5$ в интервал $[29; 31]$ составляет всего около 0.35. В то время как вероятность непадания в заданный интервал равна $1 - 0.35 = 0.65$. Поэтому мы и сделали вывод о том, что значение $\bar{x} = 31$ при $n = 5$ скорее обусловлено случайностью, чем какими-либо объективными причинами.

Фактически значение вероятности 0.65, полученное выше, — это и есть то самое p-значение, которое как раз и показывает вероятность случайного выхода за границы интервала, задаваемого значением Z-статистики.

P-значение (англ. *P-value* — *p*-уровень значимости, *p*-критерий) — вероятность получить для данной вероятностной модели распределения значений случайной величины такое же или более экстремальное значение статистики (среднего арифметического, медианы и др.) по сравнению с ранее наблюдаемым при условии, что нулевая гипотеза верна.

P-значение для данного случая можно изобразить следующим образом:

```
---in↓---
x, n, mu, sigma = 31, 5, 30, 5
z = abs((x - mu) / (sigma / n ** 0.5))
N_rv = stats.norm()
fig, ax = plt.subplots()
x = np.linspace(N_rv.ppf(0.0001), N_rv.ppf(0.9999), 300)
ax.plot(x, N_rv.pdf(x))
ax.hlines(0, x.min(), x.max(), lw=1, color="k")
ax.vlines([-z, z], 0, 0.4, color="g", lw=2)
x_le_z, x_ge_z = x[x < -z], x[x > z]
ax.fill_between(x_le_z, N_rv.pdf(x_le_z), np.zeros(len
(x_le_z)), alpha=0.3, color="b")
ax.fill_between(x_ge_z, N_rv.pdf(x_ge_z), np.zeros(len
(x_ge_z)), alpha=0.3, color="b")

p = N_rv.cdf(z) - N_rv.cdf(-z)
ax.set_title("P({:.3} < z < {:.3}) = {:.3}".format(-z, z, p));
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 53)
---↑out---
```

Чем меньше *p*-значение тем меньше вероятность того, что среднее выборки получено случайно. При этом *p*-критерий напрямую связан с двусторонними гипотезами, т.е. гипотезами о попадании величины в заданный интервал.

Если мы получили какие-либо результаты, но *p*-значение оказалось довольно большим, то вряд ли эти результаты могут считаться значимыми. Причем традиционно уровень статистической

значимости, обозначаемый как $\alpha = 0.05$ (альфа)³⁹, говорит о том, что для подтверждения значимости результатов р-критерий должен быть меньше этого уровня.

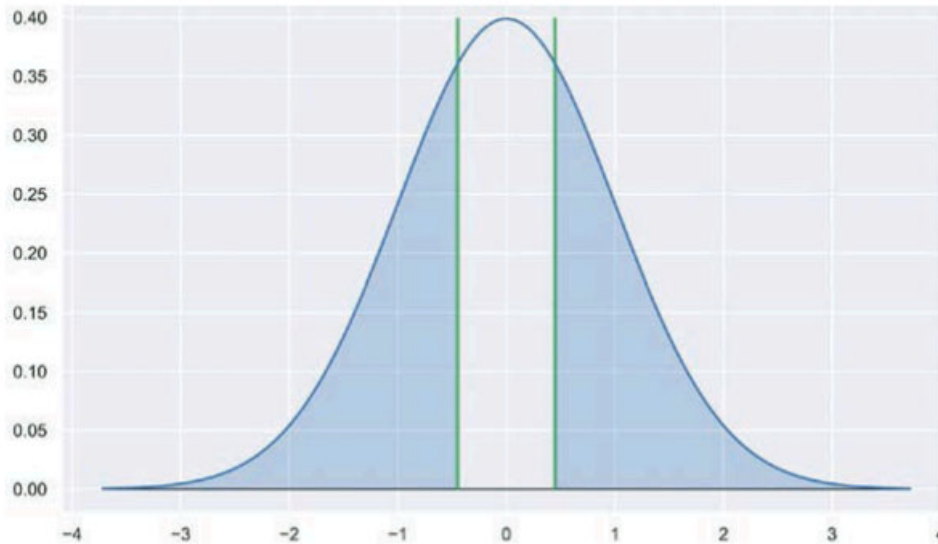


Рис. 53. P -значение = $P(-0.447 < Z < 0.447) = 0.345$

Однако стоит обязательно отметить, что уровень статистической значимости, традиционно принимаемый равным 0.05, может быть непригоден в некоторых областях исследований. Например, в сфере образования возможно можно обойтись $\alpha = 0.1$, а вот в квантовой физике, вероятно, придется снизить этот уровень до 5 сигм, т.е. α будет равна:

```
---in↓---
1 - (N_rv.cdf(5) - N_rv.cdf(-5))
---↑in_out↓---
5.733031438470704e-07
---↑out---
```

Также не следует забывать о том, что значимость может зависеть как от среднего выборки, так и от ее размера. Если вы полу-

³⁹ В статистике величину (значение) переменной называют статистически значимой, если мала вероятность случайного возникновения этой или еще более крайних величин. Здесь под крайностью понимается степень отклонения тестовой статистики от нулевой гипотезы. **Нулевая гипотеза** — принимаемое по умолчанию предположение о том, что не существует связи между двумя наблюдаемыми событиями, феноменами.

чили несколько значений, крайне нехарактерных для генеральной совокупности, как в случае с нашим соседом, то это уже повод насторожиться. Например, если наш сосед так же, как и мы, измерял время с помощью часов, то получить большие значения времени он мог из-за «севшей» батарейки.

Ошибки, связанные с извлечением выборки (сбором данных), весьма распространены. Если никаких ошибок нет, то для большей уверенности достаточно еще немного увеличить выборку. Например, наш сосед мог бы сделать еще два заказа и только после этого начать жаловаться на длительное время доставки.

С другой стороны, чтобы подтвердить небольшие отклонения от среднего генеральной совокупности, придется очень сильно увеличивать размер выборки. Так, например, если мы хотим заявить с уровнем значимости $\alpha = 0.05$, что среднее время доставки пиццы равно 31, а не 30 мин, как считалось ранее, то придется сделать не менее 100 заказов.

? Задания для проверки

1. Приведите примеры кода с генерацией случайно распределенных величин по различным законам распределения. Приведите пример установки сида генератора, опишите, для чего это может потребоваться.

2. Приведите примеры графика разных функций (CDF, ppf или др.) для гамма-распределения с разными параметрами `scale`, `loc` и `a`. Попробуйте описать полученные результаты.

3. При каком `a` гамма-распределение сводится к экспоненциальному? Приведите полный код, который выводит график какой-либо функции для гамма-распределения и аналогичный график для нормального распределения. Графики должны совпасть.

4. Опишите, зачем необходимо построение диаграммы размаха и в чем ее преимущество по сравнению с представлением в виде графиков каких-либо других функций, например PDF.

Постройте диаграмму размаха для данных, сгенерированных при помощи представленного ниже кода. Вычислять и выводить график PDF для этого не нужно.

```

---in↓---
x = np.arange(-3, 3.5, 0.5)
x = np.append(x, np.arange(1, 1.5, 0.1)) #увеличим
количество значений.
#т.е. "уплотним" распределение правее медианы
x = np.sort(x) #отсортируем значения по возрастанию (это
обязательно)
---↑in_out↓---
---↑out---

```

Убедитесь, что центр прямоугольника смещен относительно медианы (она отмечается чертой). Это нужно потому, что в наших данных есть перекося. Вычислите данный перекося (он называется куртозисом) при помощи кода ниже и проанализируйте результат:

```

---in↓---
from scipy.stats import kurtosis, skew
print(kurtosis(x))
---↑in_out↓---
---↑out---

```

5. Что такое Z-статистика? Как по ней можно оценить типичность какого-либо значения (например, насколько типичен рост мышленного существа для своего народа)? Что такое p -значение?

SQL И МЕТРИКИ ДАННЫХ

SQL — ЯЗЫК ДЛЯ РАБОТЫ С БАЗАМИ ДАННЫХ (НА ПРИМЕРЕ SQLITE)⁴⁰

Цели выполнения работы

Получение представления о языке SQL на примере работы с системой управления базой данных (СУБД⁴¹) SQLite. Изучение основных принципов практической работы с базами данных как с одним из возможных типов источников получения информации (в том числе для расчета бизнес-метрик).

Порядок выполнения работы

Немного о базах данных

Исходные данные, которые предоставляются аналитику данных для анализа, часто имеют специфический формат, который не только влияет на процесс чтения этих данных, но и определяет доступный функционал для их обработки и устанавливает ограничения как на этот функционал, так и на сами данные.

Например, мы уже рассматривали работу с CSV-файлами (данные, разделенные запятыми), этот формат не дает нам никакого удобного функционала для работы с такими данными, но зато он очень простой.

⁴⁰ Разработано на основе документации SQLite Python [42].

⁴¹ Также термин «база данных» сокращенно обозначается как БД.

База данных — совокупность данных, хранимых в соответствии со схемой данных, манипулирование которыми выполняют при помощи готового инструмента — СУБД.

Для взаимодействия с базой данных очень часто используется язык SQL (англ. *structured query language* — «язык структурированных запросов») — декларативный язык программирования, применяемый для создания, модификации и управления данными в реляционной базе данных. Реляционная база данных — это набор данных с predetermined связями между ними.

Используя средства СУБД, вы можете обрабатывать данные без отдельных языков программирования. Например, в данном модуле будут рассмотрено использование СУБД для вычисления различных метрик данных.

Метрики данных — это количественные показатели чего-либо, например:

- в бизнес-аналитике метриками могут быть число активных покупателей или пользователей, отток клиентов;
- в машинном обучении метриками может быть точность, с которой модель обрабатывает данные, или метрика «скорость» — как быстро происходит процесс обучения модели;
- многие другие метрики, которые интересны заказчику или исполнителю.

Если метрика наиболее точно описывает качество протекающих процессов, то ее называют ключевой.

На данном занятии на примере взаимодействия с базой данных будут рассмотрены некоторые простые метрики с использованием набора данных по продажам автомобилей.

Довольно часто аналитика данных ведется либо только с использованием языка SQL, либо совместно с другим языком программирования. Например, как в данном случае, с языком Python. Обычно при помощи одного лишь SQL сложные алгоритмы не реализуют: это либо громоздко, либо работает медленно.

SQLite

SQLite — компактная встраиваемая СУБД, где база данных представляет собой простой файл. В документации это называется

автономной, бессерверной, не требующей конфигурации и транзакционной СУБД SQL.

SQLite довольно проста и легковесна, поэтому даже поставляется как стандартная библиотека совместно с языком Python3.

Подключим библиотеку `sqlite3` к нашему коду и посмотрим, какую версию данной библиотеки мы имеем:

```
---in↓---
import sqlite3
print('Версия библиотеки sqlite3:', sqlite3.version)
---↑in_out↓---
Версия библиотеки sqlite3: 2.6.0
---↑out---
```

Теперь мы можем создать базу данных и подсоединиться к ней. Эти два действия выполняются одной функцией `sqlite3.connect()`. То есть если база данных с указанным именем уже существует, то мы просто откроем ее, а если не существует, то она автоматически будет создана и открыта.

```
---in↓---
connection = None
connection = sqlite3.connect('./test.db')
---↑in_out↓---
---↑out---
```

Как нам понять, что база данных успешно создалась? Просто посмотрим, какие файлы у нас есть. Для этого существует встроенная в Linux системная утилита `ls` или в Windows — `dir`. В списке файлов вы должны увидеть появившийся `test.db`, т.е. файл базы данных:

```
---in↓---
import platform

if platform.system()=='Windows':
    !dir ".\\"
if platform.system()=='Linux':
    !ls "./"
```

```
---↑in_out↓---
```

Том в устройстве C не имеет метки.

Серийный номер тома: 1111-1111

Содержимое папки C:\Users\pavel\Desktop\Аналитик данных\
Модуль 3.

Метрики данных

```
01.01.2022  20:54    <DIR>          .
01.01.2022  18:35    <DIR>          ..
10.10.2021  17:44                53 386 Chrysanthemum50.jpg
10.10.2021  17:44            45 580 638 data.csv
01.01.2022  20:39                0 test.db
01.01.2022  20:52            25 626 Метрики данных.
```

Семинар 1.ipynb

```
01.01.2022  20:01            4 046 095 Метрики данных.
```

Семинар 2..ipynb

```
      5 файлов      49 705 745 байт
      2 папок    107 770 667 008 байт свободно
```

```
---↑out---
```

Знак «!» перед командой означает, что это не Python-код, а код, который выполнится в системной оболочке. Это относится к понятию «магия» интерактивного интерпретатора, которое уже упоминалось ранее (см. сноску 25 на стр. 236).

При помощи восклицательного знака будет осуществлен запуск системной утилиты `ls` или `dir` (в зависимости от типа платформы, которую вы используете). При этом утилите будет передан параметр, в данном случае это путь к папке, содержимое которой мы хотим вывести. В нашем примере этот путь — это текущий каталог («`./`» для Linux или «`.\`» для Windows, где у Windows из-за использования в структуре файловой системы обратных слэшей обратный слэш экранирован при помощи второго обратного слэша). Текущий путь можно было и не указывать в виде параметра: эти утилиты и так по умолчанию выполняются для текущего каталога.

Если бы мы хотели указать иной каталог, например, возврат на 2 уровня вверх, то написали бы «`../..`» или «`..\..\`» или просто «`../`» и «`..\`». Также можно указывать не относительные, а абсолютные пути.

В данном случае вы могли посмотреть, создан ли файл `test.db` и через обычный графический проводник или при помощи проводника JupyterLab (но в нем файл может появиться с задержкой, возможно, потребуется вручную обновить список файлов). Вы можете проверить успешное создание файла базы данных любым удобным вам способом.

Выполнение запросов к базе данных

Вернемся к основной теме занятия. Если файл `test.db` появился в файловой системе, значит, файл базы данных был успешно создан. Наш Python-интерпретатор все еще работает. База данных не просто была создана, мы к ней подключены и можем пользоваться ей. Давайте выполним запрос к СУБД, чтобы она вывела свою версию:

```
---in↓---
cursor = connection.cursor() #любое взаимодействие с БД
происходит через указатель на нее (курсор)
cursor.execute('SELECT SQLITE_VERSION()') #вот так можно
делать запросы к базе данных
data = cursor.fetchall()[0] #при помощи метода fetchall()
мы получаем ответ на сделанный запрос.
                                #Запишем его в переменную data
print ("SQLite version: {}".format(data)) #при помощи
функции print() выведем произвольный текст
                                #и при помощи метода
.format() вставим в {} нужную переменную
---↑in_out↓---
SQLite version: ('3.31.1',)
---↑out---
```

В коде выше мы вывели не версию Python пакета-«обертки» SQLite, а версию самой СУБД, которую она хранит в служебной таблице «рядом» с «основными» данными. Тем же способом, сделав SELECT-запрос, мы можем получить и те данные, что, собственно, и хранятся в таблице. Например, это могла бы быть стоимость одной из существующих машин в базе данных.

Так как созданная база данных пустая и в ней пока ничего нет, кроме служебных полей (например, полученной ранее версии), то заполним базу информацией. Пусть это будет информация о марках машин и их стоимости (описание данного кода будет представлено далее):

```

---in↓---
cursor.execute("CREATE TABLE if not exists cars (id INT,
name TEXT, price INT)")

cursor.execute("INSERT INTO cars VALUES(1, 'Audi', 52642)")
cursor.execute("INSERT INTO cars
VALUES(2, 'Mercedes', 57127)")
cursor.execute("INSERT INTO cars VALUES(3, 'Skoda', 9000)")
cursor.execute("INSERT INTO cars
VALUES(4, 'Volvo', 29000)")
cursor.execute("INSERT INTO cars
VALUES(5, 'Bentley', 350000)")
cursor.execute("INSERT INTO cars
VALUES(6, 'Citroen', 21000)")
cursor.execute("INSERT INTO cars
VALUES(7, 'Hummer', 41400)")
cursor.execute("INSERT INTO cars
VALUES(8, 'Volkswagen', 21600)")
---↑in_out↓---
<sqlite3.Cursor at 0x7f2f44394260>
---↑out---

```

Курсор в базе данных

Как представлено выше, запросы выполняются через метод `cursor.execute()`, которому передается текстовый запрос на языке SQL.

Курсор — это именованная область памяти, содержащая результирующий набор запроса. В SQLite все манипуляции производятся через курсор. Курсор — это объект, позволяющий по отдельности обрабатывать строки из результирующего набора, возвращенного оператором SELECT.

Оператор CREATE TABLE и таблицы в базе данных

Первой строчкой мы создали таблицу `cars` (при помощи оператора `CREATE TABLE`), состоящую из трех полей:

- `id` — уникальный номер, целочисленный тип `INT`;
- `name` — марка машины, текстовый тип `TEXT`;
- `price` — стоимость машины, целочисленный тип `INT`.

При этом при создании таблицы мы указали ключевое слово `if not exists`, это означает, что таблица будет создана при условии ее отсутствия. Наша база данных была пустой, поэтому таблица `cars` будет создана, условие `if not exists` было указано только для расширения «кругозора» обучающегося.

Таблицы — это и есть объекты, которые хранят все данные в базах данных. Обычно данные разделяются по таблицам и указывается связь полей одних таблиц с полями других.

Например, можно создать таблицу «заказ», которая будет связана по ключевым полям (которые указаны как внешний ключ) с таблицей «товары», и при этом в каждой новой записи в таблице «заказ» не нужно будет хранить информацию обо всех возможных «товарах», а будет храниться только `id` тех товаров, которые действительно указаны в этом заказе.

Бывает так, что объем хранимой в базе данных информации очень большой или же необходимо увеличить скорость работы СУБД, поэтому базы данных бывают распределенными, т.е. когда таблицы логически единой базы данных хранятся на разных серверах/файлах.

Например, шардинг (англ. *shard* — термин без дословного перевода) — это метод разделения и хранения единого логического набора данных в виде множества таблиц (или даже их отдельных полей), разнесенных по разным серверам

Оператор INSERT

Вернемся к примеру кода, представленному ранее. После создания таблицы выполняется ряд запросов, содержащих оператор `INSERT`, который в таблицу `cars` помещает несколько записей, каждая из которых содержит три значения: уникальный номер, марку

машины и стоимость машины. Тем самым мы наполнили таблицу некоторым количеством данных.

Давайте сделаем запрос всех данных из таблицы `cars`:

```
---in↓---
cursor.execute('SELECT * FROM cars')
data = cursor.fetchall()
print (data)
---↑in_out↓---
[(1, 'Audi', 52642), (2, 'Mercedes', 57127), (3, 'Skoda',
9000), (4, 'Volvo', 29000), (5, 'Bentley', 350000),
(6, 'Citroen', 21000), (7, 'Hummer', 41400), (8, 'Volkswagen',
21600)]
---↑out---
```

Все данные из таблицы мы записали в переменную `data`, и выше видно, что они действительно корректно выводятся. Переменная `data` теперь хранит записи (строки) таблицы, но не имена ее полей (имена колонок). Поэтому для обращения к данным необходимо указывать индексы. В примере ниже продемонстрировано, что с индексами вывод элемента работает, а с ключами — нет:

```
---in↓(raises-exception)---
print(data[0][1]) # работает
print(data[0]["name"]) # выведет ошибку
---↑in_out↓---
Audi
-----
TypeError                                Traceback (most
recent call last)
~\AppData\Local\Temp\ipykernel_4284\3187772353.py in <module>
      1 print(data[0][1]) # работает
----> 2 print(data[0]["name"]) # выведет ошибку
TypeError: tuple indices must be integers or slices, not str
---↑out---
```

В выводимой ошибке сообщено, что `data` — это кортеж. А, как уже было рассмотрено, у кортежей есть только индексы и нет ключей, как у словарей.

Создание именованного датафрейма из таблицы SQLite

Через функцию `print()` неудобно воспринимать содержимое таблицы данных. Давайте воспользуемся представлением данных, которое нам дает библиотека `pandas`, а именно структура `DataFrame`, которую мы изучали ранее и которая является двумерной маркированной структурой. То есть `DataFrame` идеально подходит для представления и манипуляции табличными данными.

Для начала вспомним о структурах хранения данных в `Pandas`. Основными являются `Series` и `DataFrame`.

`Series` — это проиндексированный одномерный массив значений. Он похож на простой словарь типа `dict`, где имя элемента будет соответствовать индексу, а значение — значению записи.

`DataFrame` — это проиндексированный многомерный массив значений, соответственно, каждый столбец `DataFrame` является структурой `Series`.

Первый способ создания датафрейма из таблицы

В коде ниже создается `DataFrame` с именем `df`, у которого в качестве строк выступают строки из переменной `data`, а в качестве имен столбцов устанавливаются имена из переменной `column_names` (откуда она появляется, рассмотрим после кода). Импортируем модуль `pandas` и создадим структуру `DataFrame`, заполненную данными из таблицы `cars`.

```
---in↓---
import pandas as pd

#получение имени столбцов из описания курсора
column_names=[]
print (cursor.description)
for row in cursor.description:
    column_names.append(row[0])

df = pd.DataFrame(data, columns=column_names) #в качестве
исходных данных берем полученную ранее data
```

```

#проставим имена
колонок, хранящиеся в массиве column_names
print (df['name'][0])
df
---↑in_out↓---
(('id', None, None, None, None, None, None), ('name',
None, None, None, None, None, None), ('price', None,
None, None, None, None, None))
Audi
#часть вывода представлена в таблице ниже (табл. 13)
---↑out---
```

Таблица 13

Содержимое таблицы cars

	id	name	price
0	1	Audi	52642
1	2	Mercedes	57127
2	3	Skoda	9000
3	4	Volvo	29000
4	5	Bentley	350000
5	6	Citroen	21000
6	7	Hummer	41400
7	8	Volkswagen	21600

В этом примере имена столбцов мы получили из `cursor.description` (т.е. из описания курсора), который хранит метаданную информацию из последнего выполненного запроса `cursor.execute()`. В данном случае мы видим, что в описании курсора есть имена колонок, поэтому мы «переписали» их в переменную `column_names`, которую дальше указали как имена колонок `pandas`-датафрейма.

В представленном ранее примере продемонстрировано, что теперь, используя «`df['name'][0]`», мы можем обращаться к полю столбца `name` нулевой строки.

Если бы мы не устанавливали имена столбцов, то пришлось бы использовать «`df[1][0]`» для вывода этого же поля (если хотите попробовать, удалите «`columns=column_names`» и вместо «`print(df['name'][0])`» напишите «`print(df[1][0])`»).

Второй способ создания датафрейма из таблицы

Также имена столбцов можно получить другим способом — из запроса служебной метаданных информации при помощи «PRAGMA table_info(cars)», это продемонстрировано ниже. Итоговый датафрейм полностью аналогичен предыдущему:

```
---in↓---
#или можно было имена столбцов получить из информации
о таблице
cursor.execute('PRAGMA table_info(cars)')
names = cursor.fetchall()
column_names=[]
for index in enumerate(names):
    column_names.append(index[1][1])

df = pd.DataFrame(data, columns=column_names)
df
---↑in_out↓---
#часть вывода — это полная копия таблицы, которая была
в выводе предыдущего примера (табл. 13)
---↑out---
```

Третий способ создания датафрейма из таблицы

Как можно было понять из представленного выше, курсор по умолчанию возвращает данные в виде кортежа кортежей. Мы можем переключить курсор на иной режим работы, т.е. режим, когда вместо кортежей он будет возвращать словари.

У кортежей есть только индексы, а у словарей индексами может быть не только число, но и текстовый ключ. Таким образом, мы можем ссылаться на данные по их именам столбцов:

```
---in↓---
connection.row_factory = sqlite3.Row
cursor = connection.cursor()
cursor.execute("SELECT * FROM cars")
rows = cursor.fetchall()
```



```

for row in rows:
    print ("{} {} {}".format(row[0], row[1], row[2]))
print ()
print ("Выведем 6 строку столбца name: {}".format(rows[6-1]["name"])) #-1 т.к. id не с нуля

connection.row_factory = None #вернем к курсору по умолчанию – "кортеж кортежей"
cursor = connection.cursor() #обновим курсор
---↑in_out↓---
#часть вывода – это полная копия таблицы, которая была
в выводе предыдущего примера (табл. 13)
Выведем 6 строку столбца name: Citroen
---↑out---

```

Операторы и функции SQL (для аналитики данных и вычисления метрик)

Выведем данные графически в виде столбиковой диаграммы при помощи библиотеки `matplotlib`.

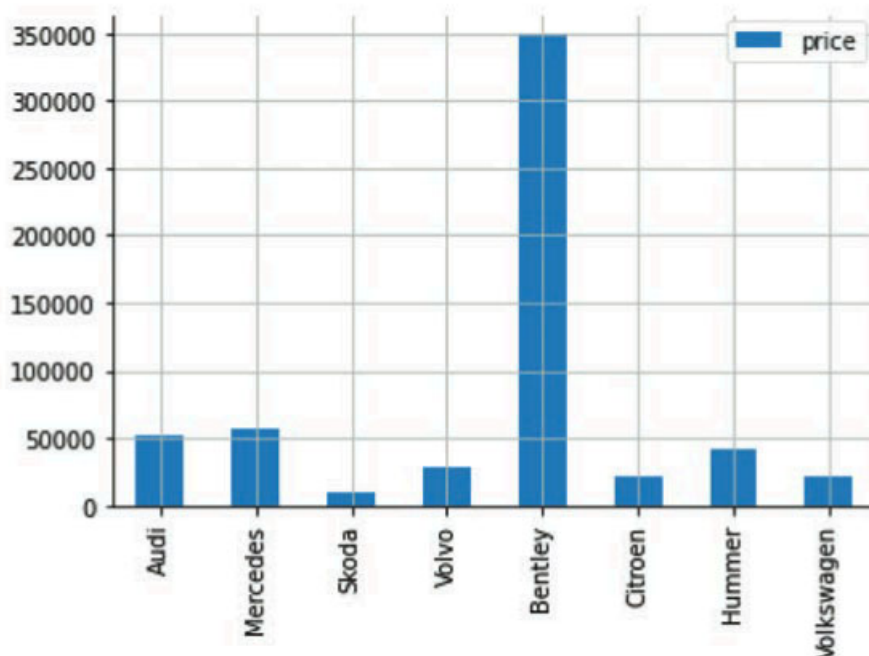


Рис. 54. Данные о ценах автомобилей

```
---in↓---
import matplotlib.pyplot as plt
df.plot(x='name', y='price', kind="bar", grid=True)
---↑in_out↓---
<AxesSubplot:xlabel='name'>
#часть вывода представлена на рисунке ниже (рис. 54)
---↑out---
```

Агрегатные функции

Смотря на данные из вывода представленного выше примера, мы уже можем сделать вывод, что нас может интересовать некоторый ряд стандартных статистических метрик (например, средняя цена автомобилей). Также в продажной сфере бывают метрики «средний чек» и многие другие, возникновение которых обусловлено бизнес-стратегией. Тем самым что угодно можно назвать метрикой.

SQLite предоставляет следующие *агрегатные функции языка SQL*:

- AVG() — возвращает среднее значение группы;
- COUNT() — возвращает количество строк, соответствующих указанному условию;
- MAX() — возвращает максимальное значение в группе;
- MIN() — возвращает минимальное значение в группе;
- SUM() — возвращает сумму значений;
- GROUP_CONCAT(выражение, разделитель) — возвращает строку, которая представляет собой объединение всех ненулевых значений.

Функция GROUP_CONCAT

Рассмотрим сначала самую последнюю в представленном выше списке функцию GROUP_CONCAT(выражение, разделитель), так как она возвращает текст, а текст вряд ли можно отнести к количественному показателю и назвать его метрикой. А дальше мы будем изучать именно метрики.

GROUP_CONCAT объединяет строки, и выражением в ней может быть, например, имя столбца, тогда весь текст со всех строк

сложится и будет разделен разделителем. По умолчанию разделителем является запятая. По реальному примеру проще понять суть этой функции:

```

---in↓---
cursor.execute("SELECT GROUP_CONCAT(name) FROM cars;")
data = cursor.fetchone()
print ("Все доступные марки: {}".format(data[0]))
---↑in_out↓---
Все доступные марки: Audi,Mercedes,Skoda,Volvo,Bentley,
Citroen,Hummer,Volkswagen
---↑out---
```

Функция MIN (метрика «самая дешевая машина»)

Ниже приведен пример использования SQL-функции MIN для вычисления метрики «самая дешевая машина»:

```

---in↓---
cursor.execute("SELECT MIN(price) FROM cars;")
data = cursor.fetchone()
print ("Минимальная цена машины: {}".format(data[0]))

cursor.execute("SELECT name FROM cars WHERE price=(SELECT
MIN(price) FROM cars);")
data = cursor.fetchone()
print ("Марка самой дешевой машины: {}".format(data[0]))
---↑in_out↓---
Минимальная цена машины: 9000
Марка самой дешевой машины: Skoda
---↑out---
```

Функция Count и оператор WHERE. Метрика «количество машин со стоимостью выше 10 000» и вывод их id. Запрос фильтра метрики у пользователя

Ниже приведен пример использования SQL-функции Count для вычисления метрики «количество машин со стоимостью выше 10 000»:

```
---in↓---
cursor.execute("SELECT Count(id) FROM cars WHERE price>10000;")
data = cursor.fetchone()
print ("Сколько машин имеет стоимость выше 10000: {}".format(data[0]))
---↑in_out↓---
Сколько машин имеет стоимость выше 10000: 7
---↑out---
```

Ниже приведен пример запроса, который возвращает уникальный номер машин, цена которых выше 10 000:

```
---in↓---
cursor.execute("SELECT id FROM cars WHERE price>10000
GROUP BY id;")
data = cursor.fetchall()
print(data)

print ("Машины с этим ID имеют стоимость выше 10000:")
for i in data:
    print (i[0], end=',')
---↑in_out↓---
[(1,), (2,), (4,), (5,), (6,), (7,), (8,)]
Машины с этим ID имеют стоимость выше 10000:
1,2,4,5,6,7,8,
---↑out---
```

Функция `input()` позволяет запросить у пользователя необходимую информацию (рассматривали ее ранее). Запустите представленный ниже код и введите интересующую вас минимальную стоимость автомобилей.

```
---in↓---
print ("Начиная с какой стоимости вас интересуют
машины?")
my_price=input()
cursor.execute(f"SELECT id FROM cars WHERE price>{my_price}
GROUP BY id;")
data = cursor.fetchall()
print ("Машины с этим ID имеют стоимость выше {}: ".format
(my_price))
```

```

for i in data:
    print (i[0], end=',')
---↑in_out↓---
Начиная с какой стоимости вас интересуют машины?
10000
Машины с этим ID имеют стоимость выше 10000:
1,2,4,5,6,7,8,
---↑out---
```

Работа с несколькими таблицами (агрегация по внешним ключам)

Создадим таблицу, содержащую имя покупателя и имя товара, которое он купил, и его количество.

```

---in↓---
cursor.execute (" \
CREATE TABLE if not exists orders ( \
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, \
    customer_name TEXT, quantity INT, bought_car_id INT, \
    FOREIGN KEY(bought_car_id) REFERENCES cars (id) \
)")
---↑in_out↓---
<sqlite3.Cursor at 0x7f2f49c762d0>
---↑out---
```

В коде выше мы создали таблицу, у которой есть внешний ключ. То есть обратим внимание на следующую часть:

```

bought_car_id INT, FOREIGN KEY(bought_car_id) REFERENCES
cars (id)
```

где поле `bought_car_id` имеет целочисленный тип `INT`, а далее при помощи атрибута `FOREIGN KEY` указывается, что это поле (поле `bought_car_id`) ссылается на поле `id` таблицы `cars`.

Знак «\» расставлен вручную и служит для переноса строк кода, чтобы сохранить его читаемость.

Именно при помощи внешних ключей и производится связь нескольких таблиц в единую схему данных. Теперь, используя по-

ле `buyed_car_id`, мы имеем возможность объединять запросы в таблицах.

Например, имея таблицы `orders` (заказы) и `cars` (машины), мы можем подсчитывать сумму заказа или выводить имя машин (марку), кроме того, выводить, какие `id` машин содержатся в заказе.

Заполним таблицу тремя заказами и посмотрим на них:

```
---in↓---
cursor.execute("INSERT INTO orders (customer_name,
quantity, buyed_car_id) VALUES ('Pavel', 2, 3)")
cursor.execute("INSERT INTO orders (customer_name,
quantity, buyed_car_id) VALUES ('Ramil', 4, 3)")
cursor.execute("INSERT INTO orders (customer_name,
quantity, buyed_car_id) VALUES ('Maryanna', 1, 6)")

cursor.execute("SELECT * FROM orders")
data = cursor.fetchall()
print (data)
---↑in_out↓---
[(1, 'Pavel', 2, 3), (2, 'Ramil', 4, 3), (3, 'Maryanna', 1, 6)]
---↑out---
```

Эти данные несут в себе «кусочек» информации, если его дополнить данными из таблицы `cars`, то получится представленный в таблице ниже вполне понятный набор данных (табл. 14, как получить такую таблицу, рассмотрим позже).

Таблица 14

Развернутый смысл таблицы `orders` (заказы)

	id заказа	марка машины	цена машины	id машины	Имя покупателя	Количество	id машины	Сумма заказа
0	3	Skoda	9000	1	Pavel	2	3	18000
1	3	Skoda	9000	2	Ramil	4	3	36000
2	6	Citroen	21000	3	Maryanna	1	6	21000

Оператор JOIN

Подсчитаем, на какую сумму было куплено машин в рамках одного заказа, средствами Python. Данный код содержит оператор объединения таблиц LEFT JOIN, который соединит таблицу cars с таблицей orders по условию соответствия orders.buied_car_id=cars.id (т.е. по внешнему ключу).

Так как мы именно к cars присоединяем orders, а не наоборот, то соединение произойдет, даже если заказа не существует, но в этом случае некоторые поля, например количество машин, в заказе будут пустыми, т.е. None.

Конечно, обычно так не делают, и, например, в подобном случае к заказам присоединяют машины. Но пока будем выполнять это таким образом (сделать иначе можно при помощи оператора GROUP BY).

```

---in↓---
cursor.execute(
    "SELECT orders.quantity,cars.name,cars.price FROM
cars LEFT JOIN orders ON orders.buied_car_id=cars.id"
)
data = cursor.fetchall()
df = pd.DataFrame(data)
print (df)
for i in data:
    print("{}:".format(i[1]), int(0 if i[0] is None else
i[0]) * i[2])
---↑in_out↓---
```

	0	1	2
0	NaN	Audi	52642
1	NaN	Mercedes	57127
2	2.0	Skoda	9000
3	4.0	Skoda	9000
4	NaN	Volvo	29000
5	NaN	Bentley	350000
6	1.0	Citroen	21000
7	NaN	Hummer	41400
8	NaN	Volkswagen	21600

Audi: 0

```
Mercedes: 0
Skoda: 18000
Skoda: 36000
Volvo: 0
Bentley: 0
Citroen: 21000
Hummer: 0
Volkswagen: 0
---↑out---
```

В коде выше («`0 if i[0] is None else i[0]`»), если первое поле записи пустое (означает, сколько таких машин в этом конкретном заказе), т.е. имеет тип `None`, то значение суммы этих машин в заказе записывается как нулевое, а если значение в этом поле ненулевое, то происходит умножение количества машин `i[0]` в заказе на стоимость машины `i[2]`. И так для каждой итерируемой записи `data`.

Обратите внимание на вывод кода выше: машина `Skoda` повторяется два раза. Это потому, что она есть в двух разных заказах (каждый из которых сделан разными людьми). Представленный выше код не вычисляет сумму всех заказов всех пользователей и полную сумму заказа одного пользователя. Он определяет сумму записи (строки) в таблице заказа (а она состоит из цены машины и количества машин этой марки в текущей записи).

То есть известно, что в реальном мире человек может заказать разный товар одним заказом. Но в данных, записанных в таблицу, в таблице `orders` нет таких записей, в которых `id` заказа одинаков. Поэтому и в данном коде мы никак не учли, что заказ может состоять из разных товаров (или даже из двух однотипных, если их пробили отдельно по 1 штуке), это не является текущей первоочередной задачей.

Ту же самую логику, что и в примере выше, мы могли реализовать средствами SQL, как представлено ниже, и это выглядит даже проще (т.е. понятнее):

```
---in↓---
```

```
cursor.execute("SELECT name, orders.quantity*cars.price
FROM cars LEFT JOIN orders ON orders.buycar_id=cars.id")
```

```
df = pd.DataFrame(cursor.fetchall())
print (df.fillna('вставьте 0')) #с заменой NaN/None
---↑in_out↓---
```

	0	1
0	Audi	вставьте 0
1	Mercedes	вставьте 0
2	Skoda	18000.0
3	Skoda	36000.0
4	Volvo	вставьте 0
5	Bentley	вставьте 0
6	Citroen	21000.0
7	Hummer	вставьте 0
8	Volkswagen	вставьте 0

```
---↑out---
```

Теперь расширим вывод. Пусть кроме произведения `orders.quantity*cars.price` также выводится все, что только возможно. Для этого после `SELECT` поместим символ «*», который укажет, что необходимо вывести все доступные поля. Также мы добавим «`WHERE orders.quantity*my_cars.price not null`», чтобы данные для «несуществующих» заказов больше не выводились:

```
---in↓---
```

```
cursor.execute("SELECT *, orders.quantity*my_cars.price FROM
cars AS my_cars LEFT JOIN orders ON orders.buycar_id=my_
cars.id WHERE orders.quantity*my_cars.price not null")
data = cursor.fetchall()
print(data)
df = pd.DataFrame(data)
print(df)
---↑in_out↓---
```

```
[(3, 'Skoda', 9000, 1, 'Pavel', 2, 3, 18000), (3, 'Skoda',
9000, 2, 'Ramil', 4, 3, 36000), (6, 'Citroen', 21000, 3,
'Maryanna', 1, 6, 21000)]
```

	0	1	2	3	4	5	6	7
0	3	Skoda	9000	1	Pavel	2	3	18000
1	3	Skoda	9000	2	Ramil	4	3	36000
2	6	Citroen	21000	3	Maryanna	1	6	21000

```
---↑out---
```

Стало больше полезной информации, но без имен колонок результаты непонятны. Поэтому добавим имена колонок уже изученным способом:

```

---in↓---
cursor.execute("SELECT *, orders.quantity*my_cars.price FROM
cars AS my_cars LEFT JOIN orders ON orders.buys_car_id=my_
cars.id WHERE orders.quantity*my_cars.price not null")
data = cursor.fetchall()

column_names = []
for index in enumerate(cursor.description):
    column_names.append(index[1][0])

df = pd.DataFrame(data, columns=column_names)
# в качестве исходных данных берем data
# проставим имена колонок, хранящиеся в массиве column_names
df
---↑in_out↓---
#результат представлен в таблице ниже (табл. 15)
---↑out---
```

Таблица 15

Объединение таблиц cars и orders с выводом суммы

	id	name	price	id	customer_name	quantity	buys_car_id	orders. quantity*cars. price
0	3	Skoda	9000	1	Pavel	2	3	18000
1	3	Skoda	9000	2	Ramil	4	3	36000
2	6	Citroen	21000	3	Maryanna	1	6	21000

Стало практически идеально для восприятия, но мы видим, что имена некоторых столбцов повторяются, а именно — совпали имена столбцов id машины и id заказа (соответственно колонкам).

Оператор AS

Чтобы привести таблицу к идеальному виду, воспользуемся оператором AS, который позволяет как изменять выводимые имена столбцов в итоговой таблице (результате запроса), так и устанавливать имена, по которым таблицы будут объединяться в секции FROM:

```

---in↓---
cursor.execute(
    " \
SELECT \
    orders.id AS 'id заказа', \
    my_cars.name AS 'Марка машины', \
    price AS 'Цена машины', \
    my_cars.id AS 'id машины', \
    customer_name AS 'Имя покупателя', \
    quantity AS 'Кол-во', \
    bought_car_id AS 'id машины', \
    orders.quantity * my_cars.price as 'Сумма заказа' \
FROM \
    cars AS my_cars \
    LEFT JOIN orders ON orders.bought_car_id = my_cars.id \
WHERE \
    orders.quantity * my_cars.price not null"
)
data = cursor.fetchall()
column_names = []
for index in enumerate(cursor.description):
    column_names.append(index[1][0])
df = pd.DataFrame(data, columns=column_names)
df
---↑in_out↓---
#результат работы данного примера уже был представлен
ранее (Таблица 14)
---↑out---
```

Тем самым была получена уже показанная ранее расширенная таблица заказов (см. табл. 14 на стр. 312).

Теперь представим эти данные графически. Для этого у pandas-датафрейма есть встроенный метод `plot`:

```

---in↓---
df.plot(
    x=1, y=7,
    kind="bar",
    title="На какую сумму продано машин",
    xlabel="марка", ylabel="сумма", grid=True
);
df.plot(
    x=1, y=5,
    kind="bar",
    title="Количество проданных машин",
    xlabel="марка", ylabel="кол-во проданных шт.",
    color="red", grid=True
);
---↑in_out↓---
#вывод показан на рисунке ниже (рис. 55)
---↑out---

```

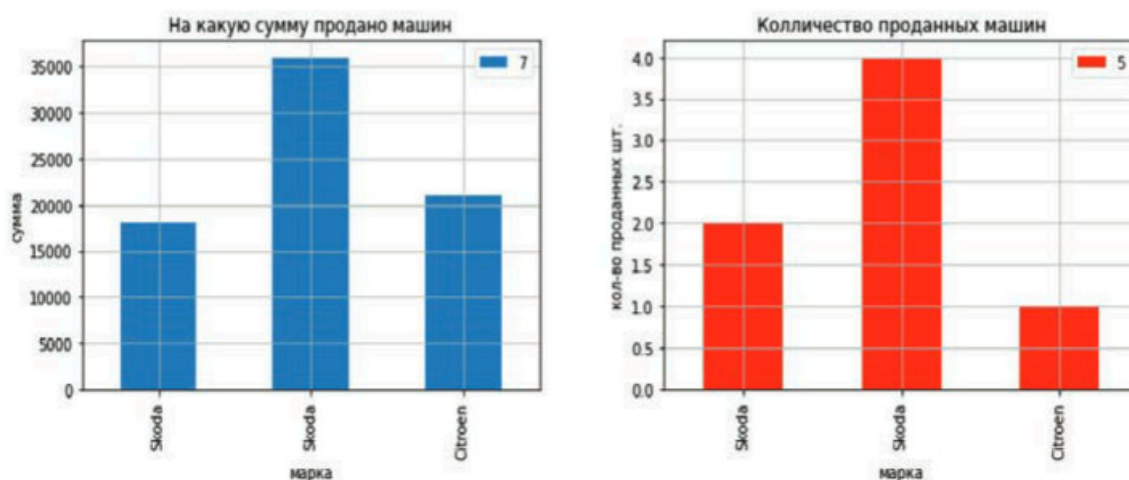


Рис. 55. Данные о проданных автомобилях

Как видим, из-за специфики кода у нас по-прежнему два автомобиля Skoda, и они не сгруппированы. Главная цель всех текущих действий — получить метрики по продаже конкретных марок машин, а для этого все же необходимо группировать результаты.

Оператор GROUP BY

При помощи SQL-оператора GROUP BY производится указание, по каким полям необходимо произвести группировку. Осуществим группировку по полю id машины. При помощи кода ниже мы получим графическое представление метрик «количество» и «выручка» по всем маркам проданных машин:

```

---in↓---
cursor.execute(" \
SELECT \
my_cars.name, SUM(orders.quantity) AS CarsCount, \
SUM(orders.quantity * my_cars.price) as 'Сумма заказа' \
FROM \
cars AS my_cars \
LEFT JOIN orders ON orders.buycar_id = my_cars.id \
WHERE \
orders.quantity * my_cars.price not null \
group by my_cars.id")

data = cursor.fetchall()
df = pd.DataFrame(data)
df.plot(
    x=0, y=2,
    kind="bar",
    title="Выручка по маркам",
    xlabel="марка", ylabel="стоимость",
    color="blue", grid=True,
);
df.plot(
    x=0, y=1,
    kind="bar",
    title="Количество проданных машин",
    xlabel="марка", ylabel="кол-во проданных шт.",
    color="red", grid=True,
);
---↑in_out↓---
#вывод показан на рисунке ниже (рис. 56)
---↑out---
```

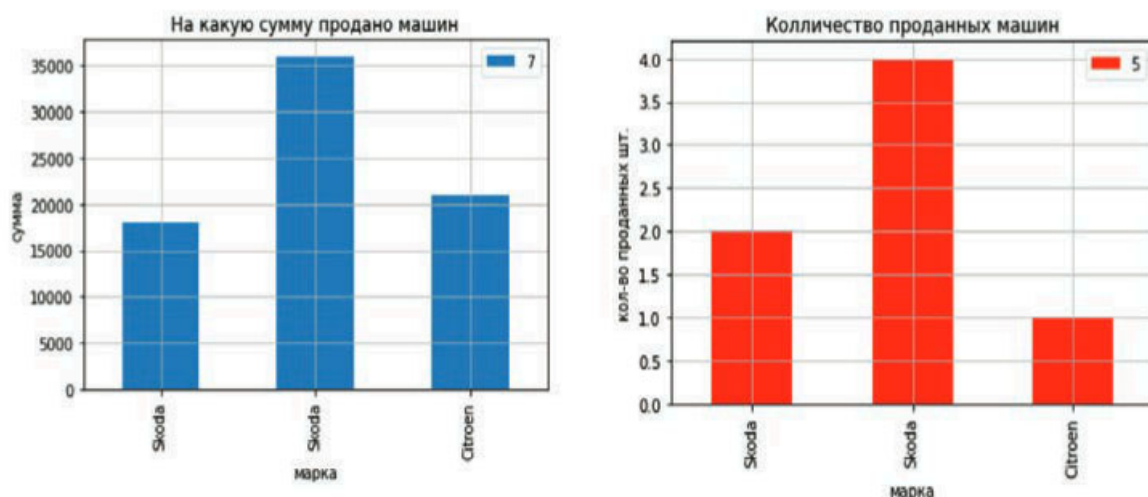


Рис. 56. Метрики о марках проданных автомобилей

При помощи оператора `GROUP BY` вы бы могли создать код (посредством доработки представленного кода), чтобы группировать все записи заказов с одинаковым `id` и тем самым сформировать единый для пользователя заказ. И тогда вы бы увидели, что покупатель в одном заказе заказал разные машины. Но в наших данных таких ситуаций нет, поэтому можете поэкспериментировать самостоятельно.

Типы данных (полей) в SQLite

SQLite поддерживает следующие типы данных, хранимые в ячейках таблиц:

- `NULL` — пустое значение;
- `INTEGER` — целочисленное со знаком (допустимые значения от -2^{63} до 2^{63}). Хранится в 1, 2, 3, 4, 6, или 8 байтах в зависимости от величины числа.
- `REAL` — число с плавающей точкой, хранится как 8 байт в формате `IEEE`;
- `TEXT` — текстовая строка, хранимая в соответствии с настроенной кодировкой базы данных (`UTF-8`, `UTF-16BE` или `UTF-16LE`);
- `BLOB` — блок данных, хранящийся точно в том виде, в котором он был введен. Так называемые бинарные данные.

Используя тип BLOB, вы можете записать в базу данных любые файлы: изображения, песни, Excel-документы или что угодно (допустимого размера). Максимальный допустимый размер от 1 до 2.1 Гб в зависимости от настроек SQLite.

Пример ниже «прочитает» изображение Chrysanthemum50.jpg (можете использовать любое, какое захотите), которое необходимо расположить рядом с этим Jupyter-блокнотом, и запишет его в базу данных, а затем прочитает его из базы данных и выведет. Обратите внимание, что код ниже использует то же соединение, что и код выше, т.е. ту же базу данных, и создаст в ней таблицу cars_img, в поля которой запишет само изображение и некоторые метаданные о нем (расширение, т.е. тип, и имя):

```
---in↓---
import os
import os.path
from os import getcwd, listdir

from IPython.core.display import Image

cursor.execute("CREATE TABLE if not exists cars_img (ID
INTEGER PRIMARY KEY AUTOINCREMENT, PICTURE BLOB, TYPE
TEXT, FILE_NAME TEXT)")

def insert_picture(cursor, picture_file):
    with open(picture_file, "rb") as input_file:
        ablob = input_file.read()
        base = os.path.basename(picture_file)
        afile, ext = os.path.splitext(base)
        sql = """INSERT INTO cars_img
(PICTURE, TYPE, FILE_NAME)
VALUES(?, ?, ?);"""
        cursor.execute(sql, [sqlite3.Binary(ablob), ext,
afile])
        cursor.commit()

picture_file = "Chrysanthemum50.jpg"
insert_picture(connection, picture_file)
```



```
def extract_picture(cursor, picture_id):
    sql = "SELECT PICTURE, TYPE, FILE_NAME FROM cars_img
WHERE id = :id"
    param = {"id": picture_id}
    cursor.execute(sql, param)
    ablob, ext, afile = cursor.fetchone()
    filename = afile + ext
    with open(filename, "wb") as output_file:
        output_file.write(ablob)
    return filename

filename = extract_picture(cursor, 1)
Image(filename="." + filename)
---↑in_out↓---
#в выводе отобразится копия изображения Chrysanthemum50.jpg
---↑out---
```

? Задания для проверки

1. Что такое СУБД SQLite, где она применяется? Каково ключевое отличие от MySQL или PostgreSQL (по информации из сети Интернет или других источников)? Какие существуют типы хранимых данных?

2. В контексте использования SQLite приведите пример Python-кода, при помощи которого создаются новые таблицы, а также пример Python-кода, выполняющего запросы на языке SQL.

3. Кратко опишите, что такое метрика, какие они бывают и какими могут быть. Приведите примеры.

4. Кратко опишите последовательность получения метрики «выручка по маркам автомобилей» и вывод ее в графическом виде.

5. Попробуйте при помощи SQL реализовать запрос, который группирует вывод заказов по уникальному номеру заказа (по id). То есть, например, чтобы можно было вывести, сколько разных машин пользователь заказал одним заказом (записи в таблице заказа разные, но у них один id заказа). Кроме создания этого запроса убедитесь, что в таблице с заказами существуют заказы с одинаковым id (в таблице из занятия таких записей нет, добавьте).

МЕТРИКИ (НА ПРИМЕРЕ КЛЮЧЕВЫХ ПРОДУКТОВЫХ МЕТРИК)⁴²

Цели выполнения работы

Продолжение получения практического представления о бизнес-метриках на примере более сложных входных данных и вычисления для них north star метрик (т.е. метрик «полярной звезды» — наиболее показательных ключевых метрик).

Аналитика по динамике среднего дохода за заказ, коэффициенту новых клиентов, удержанию клиентов, скорости оттока клиентов и др. Получение представления о когортном анализе (аналитика на основе характеристик действий пользователей/клиентов).

Порядок выполнения работы

Бизнес-метрики для продаж

На предыдущем занятии упоминалось, что конкретные наборы метрик (т.е. количественные показатели чего-либо) индивидуальны и разрабатываются (выбираются) на основе специфики предметной отрасли, например в нейронных сетях — это скорость обучения и точность, в датацентрах — это эффективность использования электроэнергии и многие другие из существующих (или потенциально разработанных вами) типов метрик.

На данном занятии продолжается рассмотрение бизнес-метрик для отрасли продаж, так как это одна из наиболее популярных и показательных отраслей при аналитике данных. Однако теперь метрики будут более сложными и для более сложных данных. Будут рассмотрены не все возможные типы метрик, а метрики категории «полярная звезда».

⁴² Разработано на основе англоязычных статей: источников [43] и [44].

Метрика north star (NSM, метрика «полярной звезды») — это показатель, который лучше всего отражает основную ценность продукта для пользователей. Правильно выбранная метрика «полярной звезды» может обеспечить компании стабильный и устойчивый рост в долгосрочном периоде. Другими словами, NSM — ключевой показатель успеха для продуктовой команды.

Показатель NSM должен включать в себя **три основных параметра**:

- *доходность* — по показателю видно, сколько зарабатывает компания;

- *ценность для пользователей* — показатель отражает основную ценность продукта для клиентов;

- *измеримость* — показатель легко измерим.

Метрика North Star на сайте высшего учебного заведения в разделе «Поступление» — это количество поданных заявлений от абитуриентов, метрика North Star для медиасервиса — это ежедневные активные пользователи или то время, в которое они просматривают контент; для магазина — конверсия продаж и т.п.

Приступим к практическим примерам вычисления продуктовых метрик для реальных данных.

Подключение библиотек, загрузка и предобработка данных

В нашем примере мы будем использовать образец набора данных розничной онлайн-торговли (доступный для скачивания в источнике [45]). Для розничной торговли через Интернет в качестве метрики North Star мы можем выбрать ежемесячный доход. Посмотрим, как выглядят наши данные.

Начнем с импорта необходимых нам библиотек и чтения данных из CSV-датасета с помощью библиотеки pandas:

```
---in↓---  
# для преобразования даты и времени из строк в объекты  
datetime и timedelta  
from datetime import datetime, timedelta
```

```

import numpy as np
import pandas as pd
import plotly.graph_objs as go # для настройки графиков
import plotly.offline as pyoff # для построения графиков

# чтобы выводить несколько pandas-датафреймов
одновременно как таблицы
from IPython.display import display

pyoff.init_notebook_mode() # использовать вывод графиков
внутри ноутбука

tx_data = pd.read_csv("data.csv", encoding="ISO-8859-1")
# открываем данные

# переименовываем столбцы, чтобы язык их названия
сменился на русский
tx_data.rename(
    columns={
        "InvoiceNo": "Номер заказа",
        "StockCode": "Код товара",
        "Description": "Описание",
        "Quantity": "Количество",
        "InvoiceDate": "Дата заказа",
        "UnitPrice": "Цена за единицу",
        "CustomerID": "ID покупателя",
        "Country": "Страна",
    },
    inplace=True, # inplace - чтобы данные сейчас не вывелись
)
---↑in_out↓---
---↑out---
```

Вот как выглядят первые 10 строк наших данных:

```

---in↓---
tx_data.head(10)
---↑in_out↓---
#вывод представлен в таблице ниже (табл. 16)
---↑out---
```

Таблица 16

Исходный набор данных tx_data

	Номер заказа	Код товара	Описание	Количество	Дата заказа	Цена за единицу	ID покупателя	Страна
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/2010 8:26	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	12/1/2010 8:26	3.39	17850.0	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/2010 8:26	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/2010 8:26	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/2010 8:26	3.39	17850.0	United Kingdom
5	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	12/1/2010 8:26	7.65	17850.0	United Kingdom
6	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	12/1/2010 8:26	4.25	17850.0	United Kingdom
7	536366	22633	HAND WARMER UNION JACK	6	12/1/2010 8:28	1.85	17850.0	United Kingdom
8	536366	22632	HAND WARMER RED POLKA DOT	6	12/1/2010 8:28	1.85	17850.0	United Kingdom
9	536367	84879	ASSORTED COLOUR BIRD ORNAMENT	32	12/1/2010 8:34	1.69	13047.0	United Kingdom

У нас есть вся важная информация, которая нам нужна:

- ID покупателя;
- цена за единицу товара;
- количество;
- дата заказа.

Обладая всеми этими параметрами, мы можем вычислять достаточно различные метрики, многие из которых могут являться наиболее показательными и ключевыми.

Посмотрим общую информацию о данных, среди которой:

- `RangeIndex` — количество строк (записей);
- «`dtypes: float64(2), int64(1), object(5)`» — типы данных в ячейках.

```

---in↓---
tx_data.info()
---↑in_out↓---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
#   Column                Non-Null Count  Dtype
---  -
0   Номер заказа          541909 non-null object
1   Код товара            541909 non-null object
2   Описание              540455 non-null object
3   Количество            541909 non-null int64
4   Дата заказа           541909 non-null object
5   Цена за единицу       541909 non-null float64
6   ID покупателя         406829 non-null float64
7   Страна                541909 non-null object
dtypes: float64(2), int64(1), object(5)
memory usage: 33.1+ MB
---↑out---
```

Как мы видим, часть полей имеет тип `object` (неопределенный). Так произошло, потому что в некоторых данных есть дефекты, т.е., например, в одной записи в этой ячейке число, а в другой — текст.

Давайте взглянем на записи под номером 20 576 и 20 577. В записи 20 576 в номере счета есть ошибка — лишняя буква С. Просмотреть эти данные и увидеть эту же ошибку вы можете, открыв данный CSV-файл в Excel.

```
---in↓---
print(tx_data.loc[20576])
print()
print(tx_data.loc[20577])
---↑in_out↓---
```

Номер заказа	C538066
Код товара	22083
Описание	PAPER CHAIN KIT RETROSPOT
Количество	-1
Дата заказа	12/9/2010 13:58
Цена за единицу	2.95
ID покупателя	15738.0
Страна	United Kingdom

Name: 20576, dtype: object

Номер заказа	538067
Код товара	84944
Описание	SET OF 6 KASHMIR FOLKART BAUBLES
Количество	1
Дата заказа	12/9/2010 13:59
Цена за единицу	4.25
ID покупателя	15288.0
Страна	United Kingdom

Name: 20577, dtype: object

```
---↑out---
```

Пока что оставим дефектные записи без изменения. Еще раз увидеть размерность наших данных (количество записей, количество столбцов) мы можем следующим образом:

```
---in↓---
tx_data.shape
---↑in_out↓---
```

(541909, 8)

```
---↑out---
```

Давайте немного обработаем наши данные и посмотрим *описательную статистику* по ним, которая включает в себя:

- count — количество ненулевых полей;
- mean — среднее значение;
- min — минимальное значение;
- 25%, 75% — 2-й и 3-й квартиль, или 25-й и 75-й процентиль;
- 50% — 50-й процентиль, или то же самое, что и медиана;
- max — максимальное значение.

Все эти статистические показатели мы уже рассматривали в разделе 3 «Статистика в Python». Кратко напомним, что означают некоторые из них:

- медиана представляет собой среднюю точку распределения: половина наблюдений (отсортированных по возрастанию) расположена над ней, а другая половина — под ней (медиана чисел 3, 4, 5, 6 и 102 составляет 5);

- квартиль — процентиль, который делит ряд значений на 4 равные части. Например: первый квартиль (25 %) это 25-й процентиль, а второй квартиль — это медиана, или 50-й процентиль;

- процентиль (персентиль) — это такое значение X_p из массива (или в терминологии статистики это называется совокупностью значений), что значения p -й части совокупности меньше или равны этому значению X_p .

Преобразуем поля столбца «Дата заказа» из текстового типа данных в тип `datetime`, чтобы мы могли использовать встроенный для `datetime` функционал по преобразованию дат и времени и их манипулированию, и заодно выведем описательную статистику:

```
---in↓---
# преобразование типа поля "Дата заказа" из строкового
в тип "datetime".
tx_data["Дата заказа"] = pd.to_datetime(tx_data["Дата
заказа"])

# вывод описания столбца 'Дата заказа'
tx_data["Дата заказа"].describe(datetime_is_numeric=True)
```

```
---↑in_out↓---
count                               541909
mean      2011-07-04 13:34:57.156386048
min                2010-12-01 08:26:00
25%                2011-03-28 11:34:00
50%                2011-07-19 17:17:00
75%                2011-10-19 11:27:00
max                2011-12-09 12:50:00
Name: Дата заказа, dtype: object
---↑out---
```

Создадим новый столбец «ГодМесяц Заказа», который упростит нам восприятие полной отметки времени из столбца «Дата заказа». Полная отметка времени нам не пригодится, поэтому мы и будем вместо нее использовать «ГодМесяц Заказа»):

```
---in↓---
# создание поля "ГодМесяц Заказа" для простоты восприятия
# вывода и его визуализации
tx_data["ГодМесяц Заказа"] = tx_data["Дата заказа"].map(
    lambda date: 100 * date.year + date.month
)

tx_data.head(3)
---↑in_out↓---
#вывод представлен в таблице ниже (табл. 17)
---↑out---
```

Метрика «ежемесячный доход»

Наиболее важным уравнением метрики, т.е. метрикой «полярной звезды», может являться уравнение суммарного дохода с продаж, вычисляемое по формуле ниже (формула (22)):

$$\begin{aligned} \text{Доход} = & \text{Количество клиентов} \times \text{Количество заказов} \times \\ & \times \text{Средний доход за заказ.} \end{aligned} \quad (22)$$

Таблица 17

Добавлен столбец «ГодМесяц заказа»

	Номер заказа	Код товара	Описание	Количество	Дата заказа	Цена за единицу	ID покупателя	Страна	ГодМесяц заказа
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom	201012
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom	201012
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom	201012

Метрики (на примере ключевых продуктовых метрик)

Реализуем данную метрику. Создадим датафрейм `tx_revenue`, который показывает ежемесячный доход. Для этого мы в столбец «Доход» запишем произведение «Цена за единицу» и «Количество», а затем сгруппируем все записи таким образом, чтобы «Доход» был не для каждого отдельного заказа, а суммарно по всем заказам за месяц (т.е. получим помесечный доход):

```
---in↓---
# рассчитать доход для каждой строки и создать новый
# датафрейм с помощью столбцов "ГодМесяц Заказа" и ""Доход""
tx_data["Доход"] = tx_data["Цена за единицу"] * tx_
data["Количество"]
# сгруппируем записи по полю "ГодМесяц Заказа", найдя при
# этом сумму доходов по этим полям
tx_revenue = tx_data.groupby(["ГодМесяц Заказа"])
["Доход"].sum().reset_index()
tx_revenue.head(4)
---↑in_out↓---
#вывод представлен в таблице ниже (табл. 18)
---↑out---
```

Таблица 18

**Первые 4 строки таблицы `tx_revenue`
(ежемесячный доход)**

	ГодМесяц Заказа	Доход
0	201012	748957.020
1	201101	560000.260
2	201102	498062.650
3	201103	683267.080

Следующий шаг — визуализация `tx_revenue`, выведем линейный график при помощи библиотеки интерактивных графиков `plotly`:

```
---in↓---
plot_data = [ go.Scatter(x=tx_revenue["ГодМесяц Заказа"],
                        y=tx_revenue["Доход"], )]
```

```

plot_layout = go.Layout(xaxis={"type": "category"},
title="Месячный Доход")

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 57)
---↑out---
```

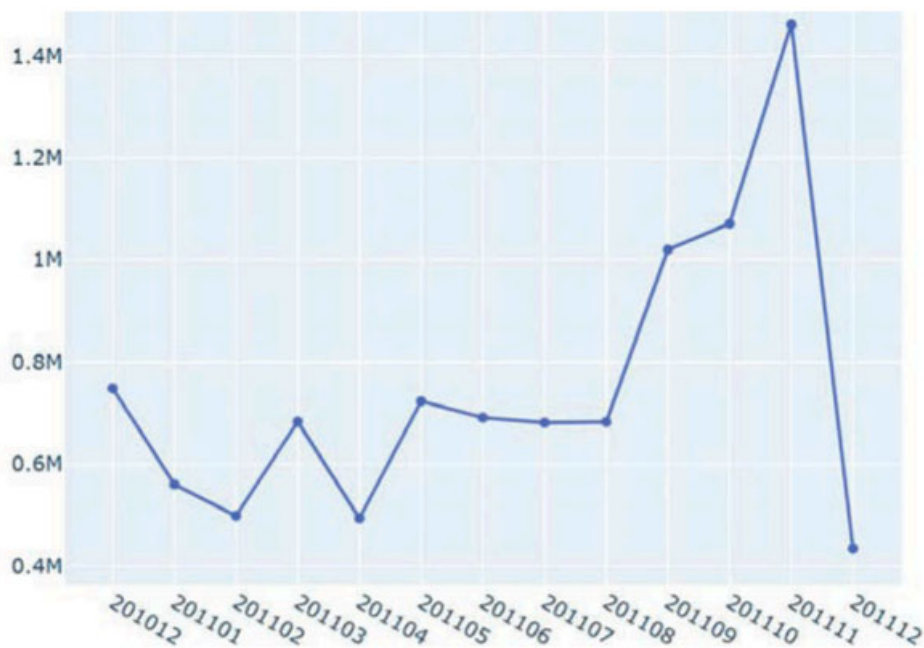


Рис. 57. Метрика «ежемесячный доход»

Тем самым мы получили графическое представление метрики «ежемесячный доход». Видно, что выручка растет, особенно после августа 2011 г. (но стоит отметить, что данные за декабрь неполные, так как декабрь — условно говоря, текущий месяц, а год еще не кончился).

Метрика «ежемесячный темп роста дохода»

Абсолютные цифры ежемесячного дохода выглядят допустимо (т.е. «на вид» без ошибок, график из вывода кода выше (рис. 57)). Давайте разберемся, какой у нас ежемесячный темп роста дохода:

```
---in↓---
# Функция pct_change () показывает процентное изменение
# между текущим и предыдущим элементом (месяцем).
tx_revenue["Месячный Рост"] = tx_revenue["Доход"].pct_
change()

# показ первых 5ти строк
display(tx_revenue.head())

# визуализация - линейный график
plot_data = [
    go.Scatter(
        x=tx_revenue.query("`ГодМесяц Заказа` < 201112")
["ГодМесяц Заказа"],
        y=tx_revenue.query("`ГодМесяц Заказа` < 201112")
["Месячный Рост"],
    )
]
plot_layout = go.Layout(
    xaxis={"type": "category"},
    yaxis={"tickformat": ",.0%"},
    title="Ежемесячный темп роста",
)
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
#вывод представлен ниже в таблице и на рисунке (табл. 19
и рис. 58 соответственно)
---↑out---
```

*Таблица 19***Первые 5 строк tx_revenue со столбцом «Месячный рост»**

	ГодМесяц Заказа	Доход	Месячный Рост
0	201012	748957.020	NaN
1	201101	560000.260	-0.252293
2	201102	498062.650	-0.110603
3	201103	683267.080	0.371850
4	201104	493207.121	-0.278163

Все выглядит хорошо (см. вывод примера выше) — в среднем имеется позитивная динамика по месячному доходу. В предыдущем месяце (см. 201111) мы увидели рост на 36.5 % (декабрь исключен в коде, так как он еще не завершен).

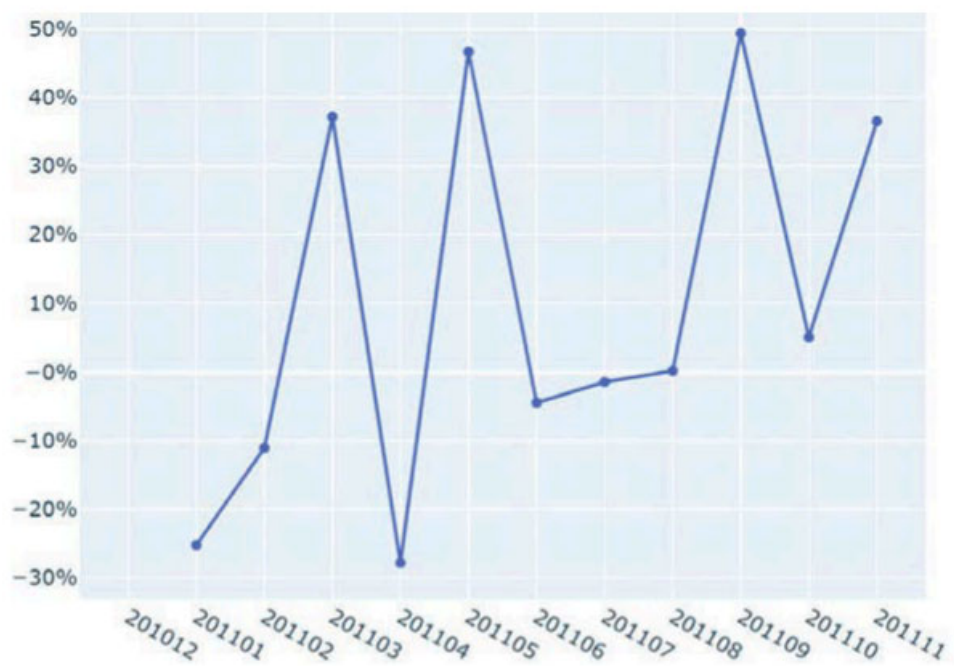


Рис. 58. Метрика «ежемесячный темп роста дохода»

Теперь нам необходимо определить, что именно произошло в апреле (почему произошел спад дохода). Было ли это из-за меньшего числа активных клиентов или активные клиенты просто сделали меньше заказов? Может, они начали покупать более дешевые товары? Чтобы ответить на эти вопросы, перейдем к вычислению более детальных метрик.

Метрика «ежемесячный доход по странам»

Предположим, что в каких-либо странах были введены санкции на некоторые виды продаваемого продукта. Чтобы получить анализ дохода по конкретным странам, воспользуемся суммированием по полям «Страна» и «ГодМесяц Заказа».

В коде ниже при группировке месячные доходы стран еще и одновременно сортируются по убыванию.

Стоит отметить, что не обязательно в данных для каждой страны получится значение дохода для каждого из месяцев, возможно, в какие-либо месяцы дохода от некоторых стран не было. Но в данном случае сортировка месячного дохода происходит по всем странам одновременно, поэтому не обязательно, что некоторые месяцы отсутствуют, скорее всего, они идут «вперемешку».

```
---in↓---
# Получаем сумму доходов по странам, ascending=False
- сортировка по убыванию
month_revenue_country=tx_data.groupby(["Страна", "ГодМесяц
Заказа"])[ "Доход"].sum().sort_values(ascending=False).
astype(int)
month_revenue_country=month_revenue_country.reset_index()

#У каждой страны не обязательно по 13 месяцев и к тому же
#month_revenue_country отсортирован по доходу, а не по
странам
#UK на два порядка лидирует по доходам, так что у нее 13
месяцев подряд
print('month_revenue_country.head(3)')
display (month_revenue_country.head(3))
print('month_revenue_country[12:].head(5)')
display (month_revenue_country[12:].head(5))

# визуализация
plot_data = [
    go.Bar(
        x=(month_revenue_country.loc[month_revenue_
country['Страна'] == "United Kingdom"])[ 'ГодМесяц Заказа'],
        y=(month_revenue_country.loc[month_revenue_
country['Страна'] == "United Kingdom"])[ 'Доход'],
        name="United Kingdom",
    ),
]
plot_layout = go.Layout(xaxis={"type": "category"},
title="Месячный Доход United Kingdom")
fig = go.Figure(data=plot_data, layout=plot_layout)
fig.update_xaxes(categoryorder='category ascending')
pyoff.iplot(fig)
```



```

plot_data = [
    go.Bar(
        x=(month_revenue_country.loc[month_revenue_
country['Страна'] == "Netherlands"])[ 'ГодМесяц Заказа'],
        y=(month_revenue_country.loc[month_revenue_
country['Страна'] == "Netherlands"])[ 'Доход'],
        name="Netherlands",
    ),
    go.Bar(
        x=(month_revenue_country.loc[month_revenue_
country['Страна'] == "Germany"])[ 'ГодМесяц Заказа'],
        y=(month_revenue_country.loc[month_revenue_
country['Страна'] == "Germany"])[ 'Доход'],
        name="Germany",
    ),
]
plot_layout = go.Layout(xaxis={"type": "category"},
title="Месячный Доход Netherlands и Germany")
fig = go.Figure(data=plot_data, layout=plot_layout)
fig.update_xaxes(categoryorder='category ascending')
pyoff.iplot(fig)
---↑in_out↓---
month_revenue_country.head(3)
#часть вывода представлена в таблице ниже (табл. 20)
month_revenue_country[12:].head(5)
#часть вывода представлена в таблице ниже (табл. 21)
#часть вывода представлена на рисунке ниже (рис. 59)
---↑out---
```

Таблица 20

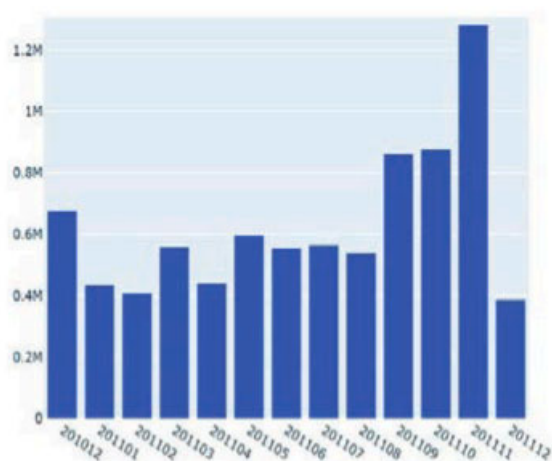
**Выборка помесечных доходов с 0 до 2
(по убыванию)**

	Страна	ГодМесяц Заказа	Доход
0	United Kingdom	201111	1282805
1	United Kingdom	201110	877438
2	United Kingdom	201109	862018

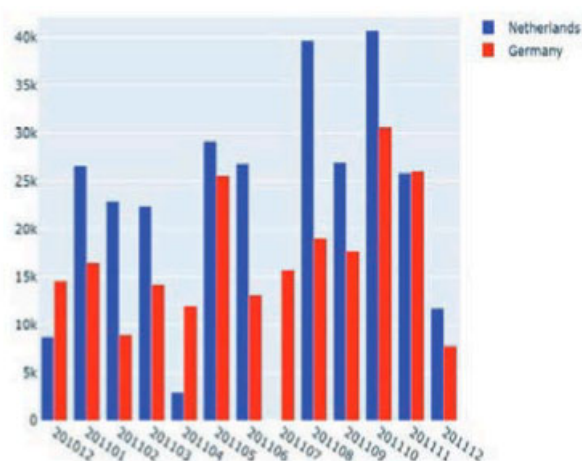
Таблица 21

**Выборка помесечных доходов с 12 до 16
(по убыванию)**

	Страна	ГодМесяц Заказа	Доход
12	United Kingdom	201112	388735
13	EIRE	201107	42740
14	EIRE	201109	42639
15	Netherlands	201110	40708
16	Netherlands	201108	39655



а



б

Рис. 59. Метрика «ежемесячный доход по странам»:
а — United Kingdom; б — Netherlands и Germany

Из вывода кода мы видим значения дохода по конкретным странам. Можно произвести расследование на основе этого кода, просмотреть все страны и сделать вывод: возникли ли проблемы с конкретными странами или наш доход снизился по каким-либо другим причинам, которые могут быть выявлены после при вычислении других метрик. Пока что перейдем далее и оставим аналитику по странам для самостоятельного анализа.

Метрика «ежемесячная активность клиентов»

Чтобы увидеть подробную информацию о клиентах, активных ежемесячно, мы будем следовать шагам, подобным тем, которые мы проделали для ежемесячного дохода.

Начиная с текущего раздела мы сосредоточимся только на данных Великобритании, потому что для этой страны больше всего данных (и по этой причине она наверняка может оказаться самой показательной).

Мы можем получить активных клиентов за месяц, подсчитав уникальные идентификаторы клиентов (`id`):

```
---in↓---
# создание нового датафрейма "только клиенты из
Великобритании"
tx_uk = tx_data.query("Страна=='United Kingdom'").reset_
index(
    drop=True
) # drop=True - не вставлять индексы

# создание датафрейма "активные клиенты" путем подсчета
ID клиентов
# nunique() - Вернуть pandas series с количеством
уникальных элементов, игнорируя пустые (NaN) значения
tx_monthly_active = (
    tx_uk.groupby("ГодМесяц Заказа")["ID покупателя"].
nunique().reset_index()
)
tx_monthly_active=tx_monthly_active.rename(
    columns={
        "ID покупателя": "Сколько покупателей",
    },
)
print('Число активных покупателей из United Kingdom
помесячно (первые 3)')
display(tx_monthly_active.head(3))

# построение графика
plot_data = [
    go.Bar(
```

```

        x=tx_monthly_active["ГодМесяц Заказа"],
        y=tx_monthly_active["Сколько покупателей"],
    )
]

plot_layout = go.Layout(
    xaxis={"type": "category", "title": "ГодМесяц Заказа"},
    yaxis={"title": "Количество покупателей"},
    title="Активные в месяце покупатели",
)
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
#часть вывода представлена ниже в таблице и на рисунке
(табл. 22 и рис. 60)
Число активных покупателей из United Kingdom ежемесячно
---↑out---
```

Таблица 22

Число активных покупателей из UK ежемесячно (первые 3)

	ГодМесяц Заказа	Сколько покупателей
0	201012	871
1	201101	684
2	201102	714

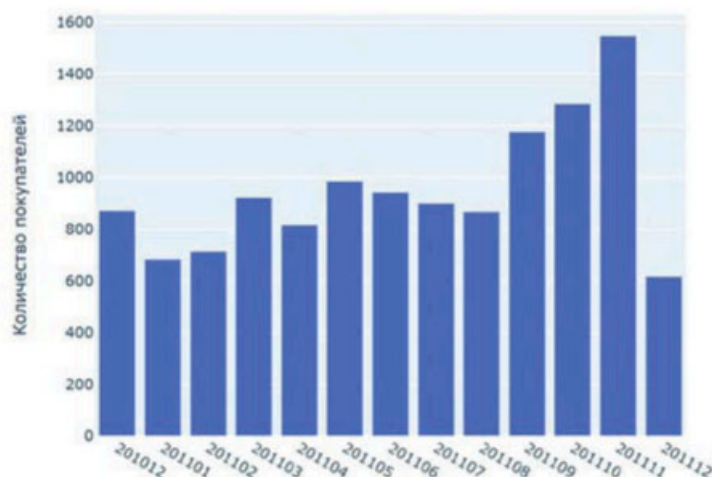


Рис. 60. Метрика «ежемесячная активность клиентов»

В апреле количество активных клиентов в месяц упало с 923 до 817 (–11.5 %).

Можно изучить корреляцию (схожесть/соотношение/зависимость) этой тенденции с метрикой «ежемесячный темп роста дохода по странам» (не вычислялась) и сравнить с данными Великобритании. Если эти «проценты» точно совпадут, то значит проблема с апрельским доходом Великобритании связана только с активностью клиентов.

Выведем среднее месячное количество покупателей:

```
---in↓---
tx_monthly_active["Сколько покупателей"].mean()
---↑in_out↓---
948.4615384615385
---↑out---
```

Теперь нам известно еще и то, сколько клиентов в каждом месяце в среднем за годовую статистику у фирмы.

Метрика «ежемесячное количество заказов»

Для вычисления метрики «ежемесячное количество заказов» мы применим тот же код, что и для метрики «ежемесячная активность клиентов», используя поле «Количество» вместо «ID покупателя».

```
---in↓---
# создать новый датафрейм для заказа с использованием
поля количества
tx_monthly_sales = tx_uk.groupby("ГодМесяц Заказа")
["Количество"].sum().reset_index()

plot_data = [
    go.Bar(
        x=tx_monthly_sales["ГодМесяц Заказа"],
        y=tx_monthly_sales["Количество"], )

plot_layout = go.Layout(
    xaxis={"type": "category"}, title="Суммарное месячное
количество заказов"
)
```



```
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 61)
---↑out---
```

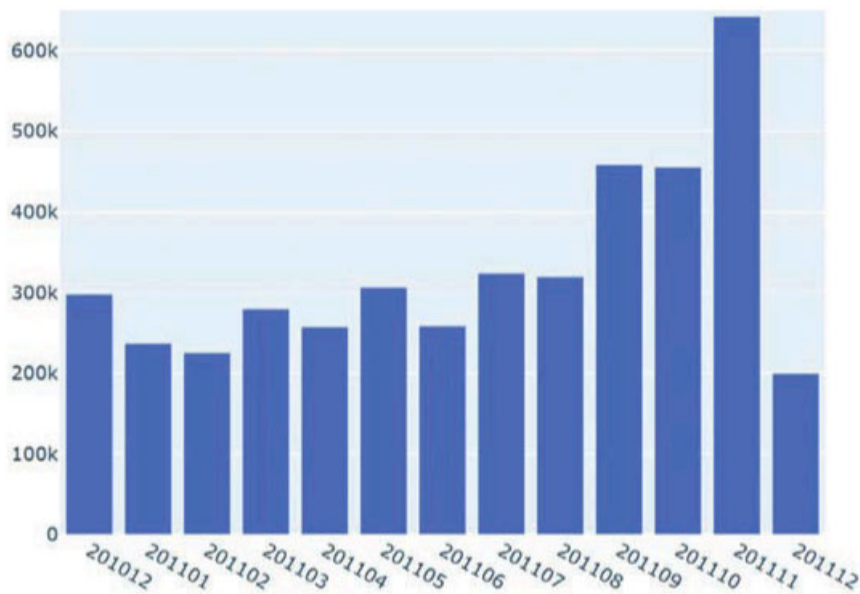


Рис. 61. Метрика «ежемесячное количество заказов»

Форма графика из вывода текущего кода (рис. 61) аналогична графику ежемесячного дохода по Великобритании (рис. 59), так как количество ежемесячных заказов в некоторой степени характеризует и ежемесячный доход.

Выведем среднее месячное количество товара в заказе:

```
---in↓---
round(tx_monthly_sales["Количество"].mean(), 2)
---↑in_out↓---
327986.85
---↑out---
```

Как и ожидалось, количество заказов также снизилось в апреле (с 279 тыс. до 257 тыс. (–8 %)). Мы знаем, что количество активных клиентов напрямую повлияло на уменьшение количества заказов.

Метрика «ежемесячный средний доход за заказ»

Теперь проверим средний доход за заказ. Чтобы получить эти данные, нам нужно рассчитать средний доход за каждый месяц:

```

---in↓---
# создать новый датафрейм для среднего дохода, взяв его
# среднее значение
tx_monthly_order_avg = tx_uk.groupby("ГодМесяц Заказа")
["Доход"].mean().reset_index()

# график
plot_data = [
    go.Bar(
        x=tx_monthly_order_avg["ГодМесяц Заказа"],
        y=tx_monthly_order_avg["Доход"],
    )
]

plot_layout = go.Layout(xaxis={"type": "category"},
title="Средний доход с заказа")
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 62)
---↑out---

```

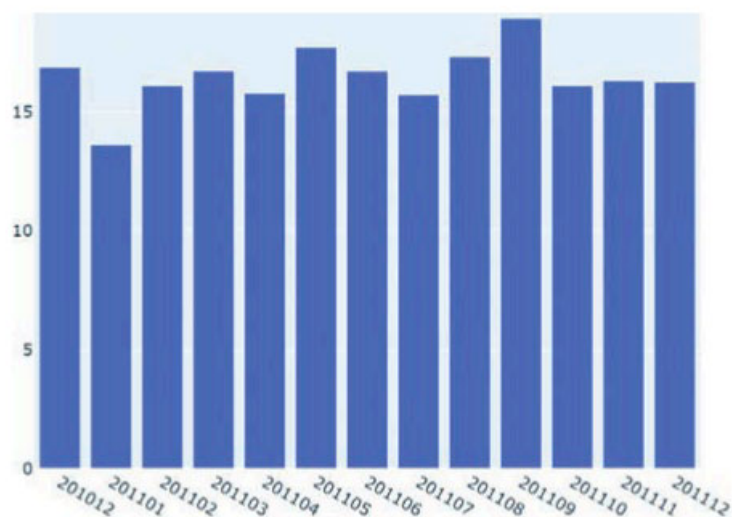


Рис. 62. Метрика «средний доход за заказ»

Даже среднемесячный объем дохода с заказов в апреле снизился (с 16.7 до 15.8 условных единиц валюты). Мы наблюдали замедление во всех метриках, влияющих на нашу «полярную звезду».

Выведем средний доход с одного заказа (так как в заказе несколько товаров):

```
---in↓---  
round(tx_monthly_order_avg["Доход"].mean(), 2)  
---↑in_out↓---  
16.47  
---↑out---
```

Мы рассмотрели наши основные показатели. Конечно, их гораздо больше, и все они варьируются в зависимости от отрасли. Продолжим изучение некоторых других важных *показателей*:

- *коэффициент новых клиентов* — хороший индикатор того, теряем ли мы наших существующих клиентов или не можем привлечь новых;

- *уровень удержания* — «лидер» показателей. Указывает, сколько клиентов мы удерживаем в течение определенного периода времени.

Будут рассмотрены примеры для ежемесячного коэффициента удержания и коэффициента удержания на основе когорт (т.е. для групп клиентов на основе признаков).

Метрика «коэффициент новых клиентов»

Сначала мы должны определить, что такое новый клиент. В нашем наборе данных мы можем предположить, что новый клиент — это тот, кто сделал свою первую покупку в указанном нами временном окне. Для данного примера мы будем делать это ежемесячно.

Мы будем использовать метод `min()`, чтобы найти дату первой покупки для каждого клиента и на основе этого определить новых клиентов. Приведенный ниже код применит эту функцию и покажет ежемесячную разбивку доходов для каждой группы.

```

---in↓---
# создать датафрейм, содержащий идентификатор клиента
и дату первой покупки
tx_min_purchase = tx_uk.groupby("ID покупателя")["Дата
заказа"].min().reset_index()
tx_min_purchase.columns = ["ID покупателя", "Минимальная
дата покупки"]
tx_min_purchase["Мин ГодМесяц покупки"] = tx_min_purchase[
    "Минимальная дата покупки"
].map(lambda date: 100 * date.year + date.month)
print("Дата первой покупки")
display(tx_min_purchase.head(3))

# присоединим столбец даты первой покупки в основной
датафрейм (tx_uk) по ID покупателя
tx_uk = pd.merge(tx_uk, tx_min_purchase, on="ID покупателя")
print("Присоединили дату первой покупки")
display(tx_uk.head(3))

# создадим столбец под названием "Тип пользователя"
и установим все его поля как "новый"
# если дата покупки пользователя больше, чем дата его первой
покупки, то значит установим, что клиент "Существующий"
tx_uk["Тип пользователя"] = "Новый"
tx_uk.loc[
    tx_uk["ГодМесяц Заказа"] > tx_uk["Мин ГодМесяц
покупки"], "Тип пользователя"
] = "Существующий"
print("Те, у которых заказ уже не первый")
display(tx_uk.loc[tx_uk["Тип пользователя"] ==
"Существующий"].head(3))

# рассчитать доход в месяц для каждого типа пользователей
tx_user_type_revenue = (
    tx_uk.groupby(["ГодМесяц Заказа", "Тип
пользователя"])(["Доход"]).sum().reset_index()
)

# фильтрация дат и построение графика
tx_user_type_revenue = tx_user_type_revenue.query(
    "`ГодМесяц Заказа` != 201012 and `ГодМесяц Заказа` !=
201112")

```

```
plot_data = [  
    go.Scatter(  
        x=tx_user_type_revenue.query("`Тип пользователя`  
== 'Существующий'")["ГодМесяц Заказа"  
    ],  
        y=tx_user_type_revenue.query("`Тип пользователя`  
== 'Существующий'")["Доход"],  
        name="Существующий",  
    ),  
    go.Scatter(  
        x=tx_user_type_revenue.query("`Тип пользователя`  
== 'Новый'")["ГодМесяц Заказа"],  
        y=tx_user_type_revenue.query("`Тип пользователя`  
== 'Новый'")["Доход"], name="Новый",),]  
  
plot_layout = go.Layout(  
    xaxis={"type": "category"}, title="Новый или  
существующий покупатель")  
fig = go.Figure(data=plot_data, layout=plot_layout)  
pyoff.iplot(fig)  
---↑in_out↓---  
Дата первой покупки  
#часть вывода представлена в таблице ниже (табл. 23)  
Присоединили дату первой покупки  
#часть вывода представлена в таблице ниже (табл. 24)  
Те, у которых заказ уже не первый  
#часть вывода представлена в таблице ниже (табл. 25)  
#часть вывода представлена на рисунке ниже (рис. 63)  
---↑out---
```

*Таблица 23***Дата первой покупки для первых пяти покупателей**

	ID покупателя	Минимальная дата покупки	Мин ГодМесяц покупки
0	12346.0	2011-01-18 10:01:00	201101
1	12747.0	2010-12-05 15:38:00	201012
2	12748.0	2010-12-01 12:48:00	201012

Таблица 24

Добавлен столбец даты первой покупки

	Номер заказа	Код товара	Описание	Количество	Дата заказа	Цена за единицу	ID покупателя	Страна	ГодМесяц заказа	Доход	Минимальная дата покупки	Мин ГодМесяц покупки
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010- 12-01 08:26:00	2.55	17850.0	United Kingdom	201012	15.30	2010- 12-01 08:26:00	201012
1	536365	71053	WHITE METAL LANTERN	6	2010- 12-01 08:26:00	3.39	17850.0	United Kingdom	201012	20.34	2010- 12-01 08:26:00	201012
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010- 12-01 08:26:00	2.75	17850.0	United Kingdom	201012	22.00	2010- 12-01 08:26:00	201012

Метрики (на примере ключевых продуктовых метрик)

Таблица 25

Столбец «Тип пользователя»

	Номер заказа	Код товара	Описание	Количество	Дата заказа	Цена за единицу	ID покупателя	Страна	ГодМесяц Заказа	Доход	Минимальная дата покупки	Мин ГодМесяц покупки	Тип пользователя
297	C543611	82483	WOOD 2 DRAWER CABINET WHITE FINISH	-1	2011-02-10 14:38:00	4.95	17850.0	United Kingdom	201102	-4.95	2010-12-01 08:26:00	201012	Сущест- вующий
298	C543611	22632	HAND WARMER RED RETROSPOT	-6	2011-02-10 14:38:00	1.85	17850.0	United Kingdom	201102	-11.10	2010-12-01 08:26:00	201012	Сущест- вующий
299	C543611	82483	WOOD 2 DRAWER CABINET WHITE FINISH	-1	2011-02-10 14:38:00	4.95	17850.0	United Kingdom	201102	-4.95	2010-12-01 08:26:00	201012	Сущест- вующий

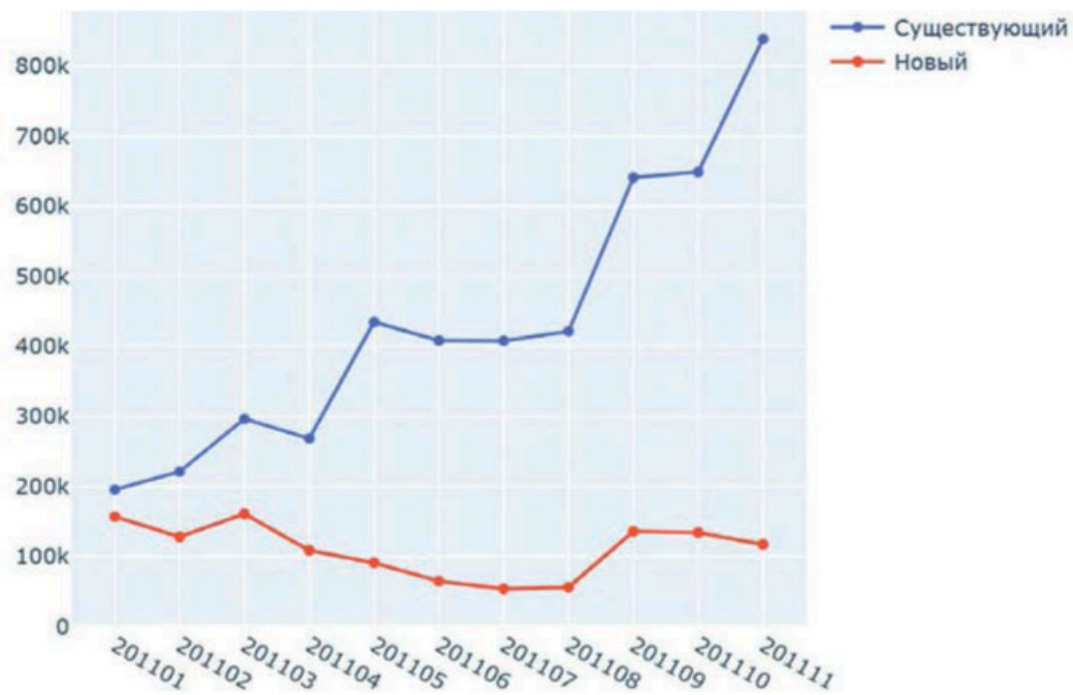


Рис. 63. Графики покупки новыми или существующими покупателями

Существующие клиенты демонстрируют положительную тенденцию, и это говорит нам, что наша клиентская база растет, но новые клиенты имеют небольшую отрицательную тенденцию по приносимому доходу.

Давайте более детально рассмотрим коэффициент новых клиентов. Будем предполагать, что в январе все клиенты были новыми, поэтому не будем выводить его на график.

```

---in↓---
# датафрейм, показывающий коэффициент новых клиентов
# ``- используются, так как русский текст с пробелами
(такая специфика)
tx_user_ratio = (
    tx_uk.query("`Тип пользователя` == 'Новый'")
    .groupby(["ГодМесяц Заказа"])["ID покупателя"]
    .nunique()
    / tx_uk.query("`Тип пользователя` == 'Существующий'")
    .groupby(["ГодМесяц Заказа"])["ID покупателя"]
    .nunique()
)

```

```
tx_user_ratio = tx_user_ratio.reset_index()

# нам также нужно отбросить пустые значения
# (так как новых пользователей в первый месяц 0 и деление
на 0)
print(
    "Обратите внимание, что в нулевом месяце все клиенты
новые, поэтому нужно удалить этот месяц:"
)
tx_user_ratio.rename(
    columns={
        "ID покупателя": "Козф. новых клиентов",
    },
    inplace=True, # inplace - чтобы данные сейчас не
вывелись
)
print("Кэффициент новых клиентов (первые 5)")
display(tx_user_ratio.head(5))

tx_user_ratio = tx_user_ratio.dropna() # dropna()
- удаление пустых значений

plot_data = [
    go.Bar(
        x=tx_user_ratio.query("`ГодМесяц Заказа`>201001
and `ГодМесяц Заказа`<201112")["ГодМесяц Заказа"],
    ),
    go.Bar(
        x=tx_user_ratio.query("`ГодМесяц Заказа`>201101
and `ГодМесяц Заказа`<201112")["Козф. новых клиентов"],
    ),
]

plot_layout = go.Layout(xaxis={"type": "category"},
title="Кэффициент новых клиентов")
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
```

---↑in_out↓---

Обратите внимание, что в нулевом месяце все клиенты новые, поэтому нужно удалить этот месяц:

Коэффициент новых клиентов (первые 5)

#часть вывода представлена в таблице ниже (табл. 26)

#часть вывода представлена на рисунке ниже (рис. 64)

---↑out---

Таблица 26

Коэффициенты новых клиентов (первые 3)

	ГодМесяц Заказа	Козф. новых клиентов
0	201012	NaN
1	201101	1.124224
2	201102	0.904000
3	201103	0.792233

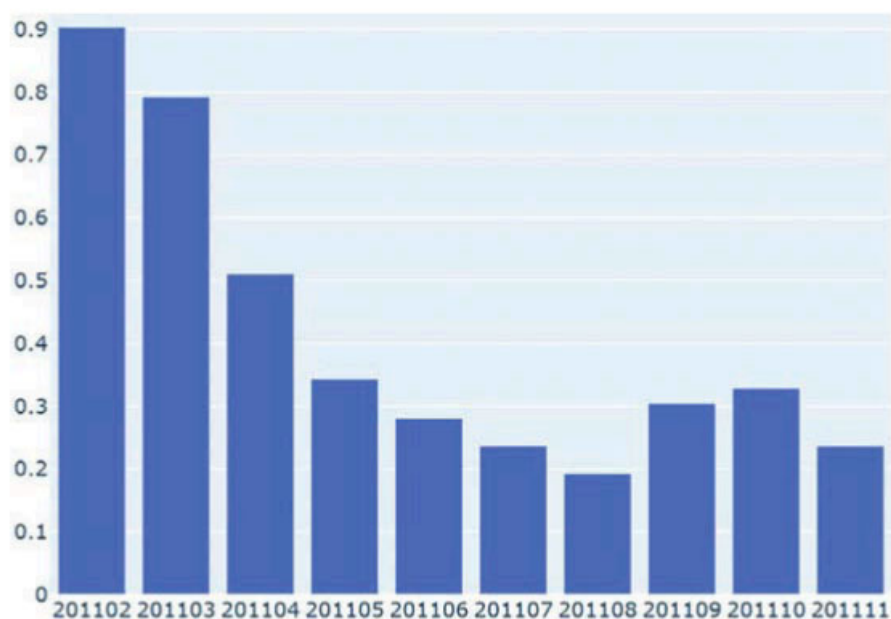


Рис. 64. Метрика «коэффициент новых клиентов»

Коэффициент новых клиентов снизился (в «текущем» периоде, т.е. 201111), как и ожидалось, и составляет около 23 %.

Метрика «ежемесячная ставка удержания»

Следует очень внимательно следить за уровнем удержания, потому что он показывает, насколько популярны ваши услуги и насколько хорошо ваш продукт соответствует рынку. Чтобы визуализировать ежемесячный коэффициент удержания, нам необходимо подсчитать, сколько клиентов было удержано за предыдущий месяц, по формуле, представленной ниже (формула (23)).

$$\text{Ежемесячный коэффициент удержания} = \frac{\text{Удержание клиентов с предыдущего периода}}{\text{Всего активных клиентов}}. \quad (23)$$

Используем функцию `crossstab()` для `pandas`, которая будет первым шагом к вычислению `Retention Rate` (уровень удержания). При помощи `crossstab()` создадим матрицу `tx_retention`, в которой для каждого клиента будет проставлена единица в том месяце, в котором у клиента были заказы.

Далее в цикле для всех месяцев посчитаем разницу между количеством таких «единиц» за текущий и предыдущий месяц и тем самым получим помесечный коэффициент удержания для каждого месяца:

```
---in↓---
# определить, какие пользователи активны, посмотрев на
# ежемесячный доход от них
tx_user_purchase = (
    tx_uk.groupby(["ID покупателя", "ГодМесяц Заказа"])
    ["Доход"].sum().reset_index()
)
print("Доход с пользователя за месяц:")
display(tx_user_purchase.head(3))

# создать матрицу удержания с помощью crosstab()
tx_retention = pd.crosstab(
    tx_user_purchase["ID покупателя"], tx_user_
    purchase["ГодМесяц Заказа"]
```

```

).reset_index()
print("Таблица удержания по каждой записи:")
display(tx_retention.head(3))

# создадим массив словарей, в котором будет храниться
# количество удержанных и общих пользователей за каждый
# месяц
months = tx_retention.columns[2:]
retention_array = []
for i in range(len(months) - 1): # для каждого из месяцев
    retention_data = {} # создадим словарь
    selected_month = months[i + 1] # возьмем следующий
# месяц
    prev_month = months[i] # и предыдущий
    retention_data["ГодМесяц Заказа"] = int(
        selected_month
    ) # и теперь в созданный словарь запишем ГодМесяц
# текущего и
    retention_data["Общее кол-во пользоват."] = tx_
# retention[
        selected_month
    ].sum() # общее кол-во клиентов за этот месяц
    retention_data["Кол-во удержанных пользователей"] =
# tx_retention[
        (tx_retention[selected_month] > 0) &
        (tx_retention[prev_month] > 0)
    ][
        selected_month
    ].sum() # и кол-во удержанных

    retention_array.append(
        retention_data
    ) # затем добавим данные о месяце в итоговый массив
# всех месяцев

# преобразуем итоговый массив в датафрейм и рассчитаем
# коэффициент удержания
tx_retention = pd.DataFrame(retention_array)
tx_retention["Коеф удержания"] = (

```

```
tx_retention["Кол-во удержанных пользователей"]
/ tx_retention["Общее кол-во пользует."]
)
print("Таблица удержания по месяцам:")
display(tx_retention.head(3))

plot_data = [
    go.Scatter(
        x=tx_retention.query("`ГодМесяц Заказа`<201112")
["ГодМесяц Заказа"],
        y=tx_retention.query("`ГодМесяц Заказа`<201112")
["Коэф удержания"],
        name="organic",)
]
plot_layout = go.Layout(
    xaxis={"type": "category"}, title="Ежемесячная ставка
удержания")
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
```

Доход с пользователя за месяц:
#часть вывода представлена в таблице ниже (табл. 27)
Таблица удержания по каждой записи:
#часть вывода представлена в таблице ниже (табл. 28)
Таблица удержания по месяцам:
#часть вывода представлена в таблице ниже (табл. 29)
#часть вывода представлена на рисунке ниже (рис. 65)
---↑out---

*Таблица 27***Доход с пользователя за месяц (первые 3)**

	ID покупателя	ГодМесяц Заказа	Доход
0	12346.0	201101	0.00
1	12747.0	201012	706.27
2	12747.0	201101	303.04

Таблица 28

Таблица удержания по каждой записи (первые 3)

ГодМесяц заказа	ID покупателя	201012	201101	201102	201103	201104	201105	201106	201107	201108	201109	201110	201111	201112
0	12346.0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	12747.0	1	1	0	1	0	1	1	0	1	0	1	1	1
2	12748.0	1	1	1	1	1	1	1	1	1	1	1	1	1

Таблица 29

Таблица удержания по месяцам (первые 3)

	ГодМесяц Заказа	Общее количество пользователей	Количество удержанных пользователей	Коэффициент удержания
0	201102	714	263	0.368347
1	201103	923	305	0.330444
2	201104	817	310	0.379437

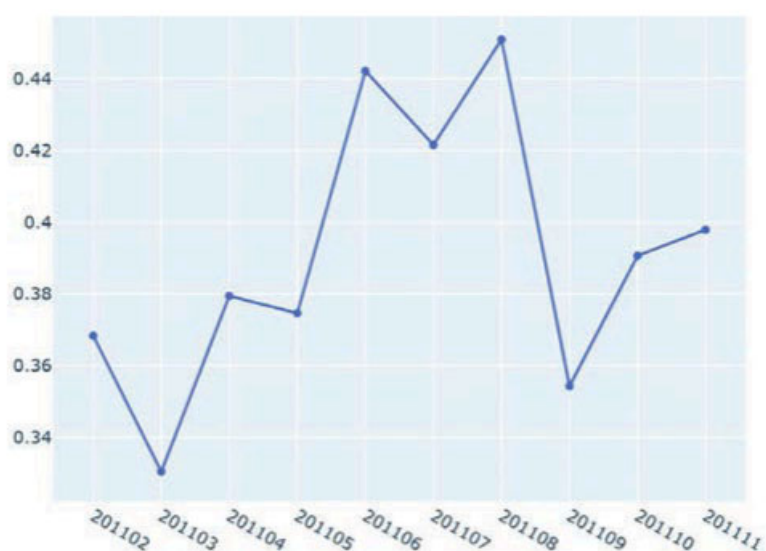


Рис. 65. Метрика «ежемесячная ставка удержания»

Ежемесячный коэффициент удержания значительно вырос с июня по август и впоследствии вернулся к предыдущим уровням.

Метрика «скорость оттока клиентов»

Метрика «скорость оттока клиентов» является обратной метрике «удержание клиентов»:

```

---in↓---
tx_retention["Скорость оттока"] = 1 - tx_retention["Коэф
удержания"]

plot_data = [
    go.Scatter(
        x=tx_retention.query("`ГодМесяц Заказа`<201112")
["ГодМесяц Заказа"],
        y=tx_retention.query("`ГодМесяц Заказа`<201112")
["Скорость оттока"],
        name="organic",
    )]
plot_layout = go.Layout(xaxis={"type": "category"},
title="Месячный отток клиентов")
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 66)
---↑out---
```

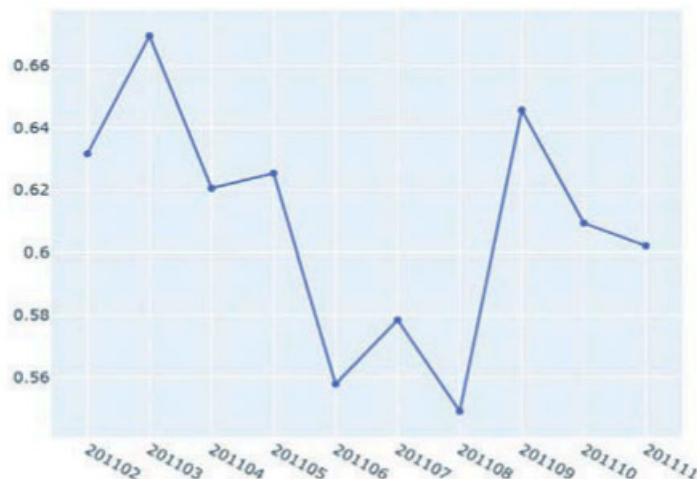


Рис. 66. Метрика «ежемесячный отток клиентов»

Метрика «коэффициент удержания когорт»

Есть еще один способ измерения коэффициента удержания, который позволяет видеть коэффициент удержания для каждой когорты и отслеживать действия этих пользователей на протяжении того или иного периода времени.

Когорта — это группа людей, которые разделяют определенный опыт в течение заданного временного отрезка. Примером когорты является группа пользователей, которые зарегистрировались в сервисе в конкретный день.

Коэффициент удержания когорты — это отношение количества активных пользователей в конце периода к числу активных пользователей в начале этого периода. Определение активного пользователя зависит прежде всего от самого продукта.

Будем разделять когорты по году и месяцу первой покупки и измерять, какой процент клиентов остается после первой покупки или, говоря более простым языком, насколько активно «старые клиенты» продолжают покупать товар.

Возможным недостатком этого подхода является то, что дата-сет не содержит данных о первых клиентах. Другими словами, первая покупка, которую мы видим в этом наборе данных, может не являться реальной первой покупкой данного клиента. Однако невозможно учесть это, не имея доступа ко всему набору исторических данных продавца.

Также недостатком данного подхода может являться то, что данные содержат повторяющихся клиентов (т.е. один клиент может «испортить статистику», если вдруг начнет делать столько заказов, что их количество будет сопоставимо с десятками или даже сотнями заказов других клиентов за рассматриваемые периоды времени).

То, чем мы занимались выше, может называться ретроспективным анализом (а данные могут называться не только историческими, но и ретроспективными).

Ретроспективный (оценочный) анализ — это анализ данных с учетом изменения во времени, начиная от текущего момента времени к какому-либо прошедшему периоду времени.

Приступим к анализу когорт. Сначала мы импортируем недостающие модули и сохраняем только столбцы с удалением повто-

ряющихся в них значений, т.е. один заказ, обозначенный как «Номер заказа», может содержать несколько элементов, обозначенных «Код товара», и тем самым получатся дубли записей с одинаковым номером заказа. Все остальные столбцы не потребуются.

```
---in↓---
# библиотека, чтобы использовать установку атрибутов вместо
# как более скоростная альтернатива применения лямбда-функций
from operator import attrgetter

import matplotlib.pyplot as plt
import seaborn as sns # более высокоуровневое построение
графиков при помощи matplotlib
# т.е. надстройка над matplotlib для построения
статистических графиков

# стерли копии одних и тех же заказов, если в заказе
имелось несколько товаров
# (несколько записей с одним и тем же "ID покупателя",
"Номер заказа" и "Дата заказа")
# при этом еще выкинем все остальные поля, так как они
нам не потребуются
tx_data = tx_data[["ID покупателя", "Номер заказа", "Дата
заказа"]].drop_duplicates()
---↑in_out↓---
---↑out---
```

Теперь добавим колонки «ГодМесяц Заказа» и «Дата первой покупки»:

- «ГодМесяц Заказа» — это усеченная версия поля «Дата заказа», как и было раньше, но теперь это в нем хранится `datetime`-объект (чтобы можно было использовать функционал по манипуляции с датами, например вычитание двух дат друг из друга);

- «Дата первой покупки» — это дата первой покупки данного покупателя.

```
---in↓---
# преобразование типа поля "Дата заказа" из строкового
в тип "datetime".
```

```

tx_data["Дата заказа"] = pd.to_datetime(tx_data["Дата
заказа"])

tx_data["ГодМесяц Заказа"] = tx_data["Дата заказа"].
dt.to_period("М")
tx_data["Дата первой покупки"] = (
    tx_data.groupby("ID покупателя")["Дата заказа"].
transform("min").dt.to_period("М")
)
tx_data.head(5)
---↑in_out↓---
#вывод представлен в таблице ниже (табл. 30)
---↑out---
```

Таблица 30

Колонки «ГодМесяц Заказа» и «Дата первой покупки»

	ID покупателя	Номер заказа	Дата заказа	ГодМесяц Заказа	Дата первой покупки
0	17850.0	536365	2010-12-01 08:26:00	2010-12	2010-12
7	17850.0	536366	2010-12-01 08:28:00	2010-12	2010-12
9	13047.0	536367	2010-12-01 08:34:00	2010-12	2010-12
21	13047.0	536368	2010-12-01 08:34:00	2010-12	2010-12
25	13047.0	536369	2010-12-01 08:35:00	2010-12	2010-12

Затем мы агрегируем (объединяем) данные по когорте (т.е. по дате первой покупки) и по полю «ГодМесяц Заказа» и подсчитываем количество уникальных клиентов в каждой группе.

Кроме того, мы добавляем колонку «period_number» (номер периода), в которой будет указываться количество периодов (месяцев) между месяцем первого заказа и месяцем покупки.

```
---in↓---
tx_data_cohort = (
    tx_data.groupby(["Дата первой покупки", "ГодМесяц
Заказа"])
    .agg(n_customers=("ID покупателя", "nunique"))
    .reset_index(drop=False)
)
tx_data_cohort["period_number"] = (
    tx_data_cohort["ГодМесяц Заказа"] - tx_data_
cohort["Дата первой покупки"]
).apply( # применить какую-либо функцию
    attrgetter(
        "n"
    ) # применим функцию установки атрибутов. Атрибут n
- это количество периодов
) # вместо объекта "непонятного" типа выводить результат
вычитания как количество периодов (т.е. месяцев)
# если сделать apply(attrgetter("n","name")), то будет
(0,M), где 0 - сколько периодов, M - тип периода, т.е.
месяц. 0 месяцев
tx_data_cohort.head(5)
---↑in_out↓---
#вывод представлен в таблице ниже (табл. 31)
---↑out---
```

Таблица 31

**Данные о размере когорт
и соответствующем ему периоде**

	Дата первой покупки	ГодМесяц Заказа	n_customers	period_number
0	2010-12	2010-12	948	0
1	2010-12	2011-01	362	1
2	2010-12	2011-02	317	2
3	2010-12	2011-03	367	3
4	2010-12	2011-04	341	4

Таблица 32

Матрица размера когорт

period_ number / Дата первой покупки	0	1	2	3	4	5	6	7	8	9	10	11	12
2010-12	948.0	362.0	317.0	367.0	341.0	376.0	360.0	336.0	336.0	374.0	354.0	474.0	260.0
2011-01	421.0	101.0	119.0	102.0	138.0	126.0	110.0	108.0	131.0	146.0	155.0	63.0	NaN
2011-02	380.0	94.0	73.0	106.0	102.0	94.0	97.0	107.0	98.0	119.0	35.0	NaN	NaN
2011-03	440.0	84.0	112.0	96.0	102.0	78.0	116.0	105.0	127.0	39.0	NaN	NaN	NaN

Таблица 33

Матрица удержания когорт

period_ number / Дата первой покупки	0	1	2	3	4	5	6	7	8	9	10	11	12
2010-12	1.0	0.381857	0.334388	0.387131	0.359705	0.396624	0.379747	0.354430	0.354430	0.394515	0.373418	0.500000	0.274262
2011-01	1.0	0.239905	0.282660	0.242280	0.327791	0.299287	0.261283	0.256532	0.311164	0.346793	0.368171	0.149644	NaN
2011-02	1.0	0.247368	0.192105	0.278947	0.268421	0.247368	0.255263	0.281579	0.257895	0.313158	0.092105	NaN	NaN
2011-03	1.0	0.190909	0.254545	0.218182	0.231818	0.177273	0.263636	0.238636	0.288636	0.088636	NaN	NaN	NaN

Следующим шагом преобразуем таблицу `tx_data_cohort` таким образом, чтобы каждая из ячеек строк содержала количество покупателей в данной когорте, а столбцами были периоды (месяцы):

```
---in↓---
cohort_pivot = tx_data_cohort.pivot_table(index="Дата первой
покупки", columns="period_number", values="n_customers")
cohort_pivot.head(4)
---↑in_out↓---
#вывод представлен в таблице ниже (табл. 32)
---↑out---
```

Первые значения строк, т.е. значения за нулевой период, — это размер когорт, т.е. количество всех клиентов, сделавших свою первую покупку в данном месяце.

Чтобы получить матрицу удержания, необходимо разделить все значения каждой строки на первое значение строки (размер когорт).

Например, если бы мы хотели посчитать коэффициент удержания за последний, 12 период, мы бы 260.0 разделили на 948.0 и получили, что удержание когорты с условным названием «покупатели, которые с нами уже чуть больше года» составляет 27 %. То есть 27 % покупателей в 12-м периоде совершили покупку⁴³.

Получим из матрицы размера когорт `cohort_pivot` матрицу удержания `retention_matrix`:

```
---in↓---
cohort_size = cohort_pivot.iloc[:, 0]
retention_matrix = cohort_pivot.divide(cohort_size, axis=0)
display(retention_matrix.head(4))
---↑in_out↓---
#вывод представлен в таблице ниже (табл. 33)
---↑out---
```

Используя полученную матрицу, строим ее графическое представление в виде тепловой карты при помощи кода ниже:

⁴³ Имеется в виду, 27 % тех, чей «возраст покупателя» тоже составляет 12 периодов, а другие 73 % «старых» покупателей с возрастом, эквивалентным 12 периодам — воздержались от покупки в 12-м периоде.

```

---in↓---
with sns.axes_style("white"):
    fig, ax = plt.subplots(
        1, 2, figsize=(12, 8), sharey=True, gridspec_
kw={"width_ratios": [1, 11]})

    # матрица удержания
    sns.heatmap(
        retention_matrix,
        mask=retention_matrix.isnull(),
        annot=True,
        fmt=".0%",
        cmap="RdYlGn",
        ax=ax[1],)
    ax[1].set_title("Месячные когорты: удержание
пользователей", fontsize=16)
    ax[1].set(xlabel="№ периода", ylabel="")

    # размер когорты
    cohort_size_df = pd.DataFrame(cohort_size).
rename(columns={0: "размер когорт"})
    sns.heatmap(cohort_size_df, annot=True, cbar=False,
fmt="g", cmap="Greys", ax=ax[0])
    fig.tight_layout()
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 67)
---↑out---

```

В данном коде мы использовали библиотеку `seaborn`, которая является надстройкой над рассмотренной ранее библиотекой `matplotlib` для построения графиков. При помощи `seaborn` в основном можно строить сложные и гибкие графики описательной статистики.

Фактически мы создали две тепловые карты с разными палитрами (см. код выше и его вывод):

- первая — карта, показывающая размер когорты (бело-черная палитра «Greys»);
- вторая — карта удержания (красно-желтая палитра «RdYlGn»).

На изображении из вывода кода выше мы видим, что уже во втором месяце наблюдается резкий спад (см. 1-й номер периода), в среднем около 76 % клиентов не совершают покупок во втором месяце.

Покупатели из первой когорты (2010–12), т.е. наиболее «старые» покупатели, имеют на удивление хороший коэффициент удержания на всех периодах (месяцах). К сожалению, это нельзя сказать о более «молодых» когортах, они имеют коэффициент удержания не более 15 % за последний месяц.

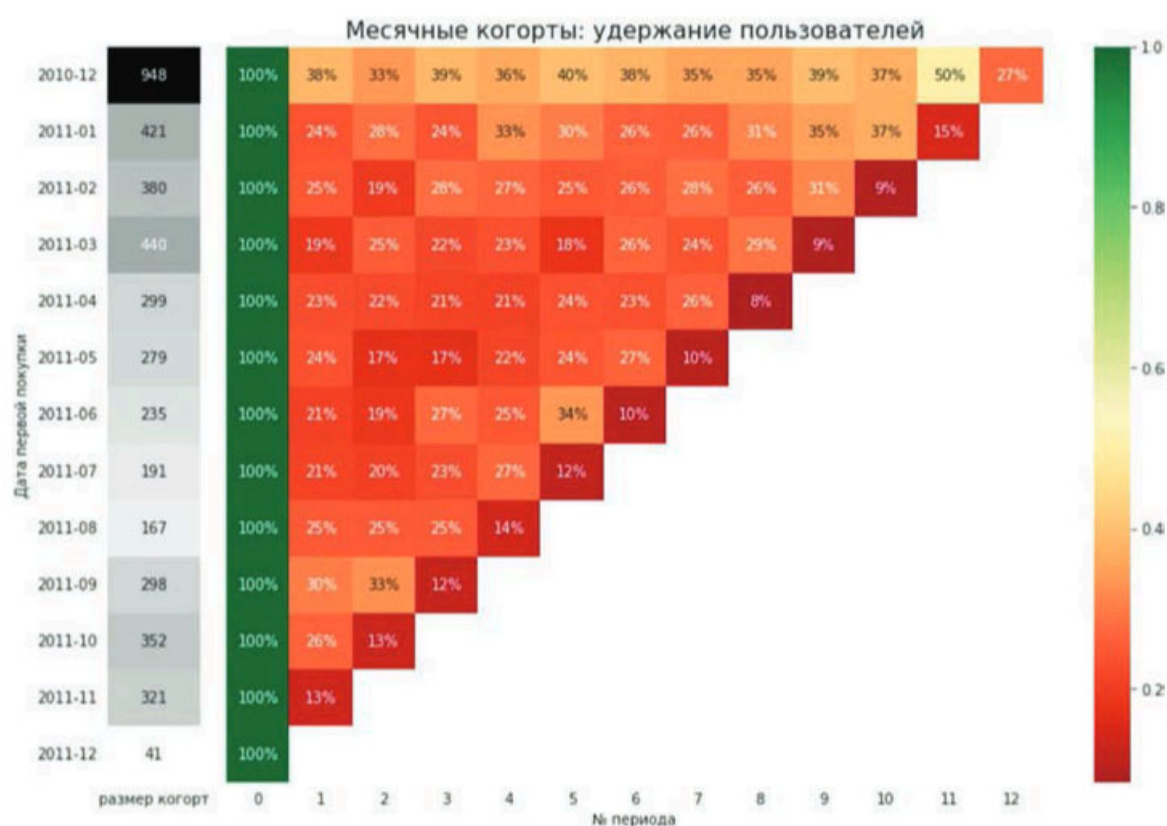


Рис. 67. Метрика «коэффициент удержания когорт»

Через год после первой покупки удержание составляет 50 % (см. первую когорту самых «старых» покупателей с 11-м периодом). Это может быть группа преданных клиентов, которые первыми присоединились к покупкам в данном магазине.

Возможно, продавец сначала рекламировал свой магазин среди своих родственников и знакомых, поэтому они и совершили

покупку первыми и продолжают покупать в этом магазине даже спустя год. Или возможна любая другая причина, о существовании которой можно только догадываться.

По всей матрице мы можем видеть колебания удержания с течением времени. Это может быть вызвано особенностями бизнеса, когда клиенты совершают периодические покупки, за которыми следуют периоды «бездействия».

Как многократно было показано в данном занятии, на основе специфики набора данных можно выбирать инструменты и методы работы с этими данными, а также выстраивать некоторые предположения, которые упрощают процесс аналитики данных и в итоге позволяют лучше понимать бизнес (или анализируемый процесс).

? Задания для проверки

1. Что такое метрика «полярной звезды» и какими показателями она должна обладать? Приведите пример такой метрики.

2. Как считываются данные из CSV-файла и записываются в `pandas`-датафрейм? При помощи какого метода можно изменять имена столбцов?

3. Как вывести «шапку» `pandas`-датафрейма и получить информацию о типе переменных в ячейках? Как вывести конкретную запись по ее индексу?

4. В примерах в занятии используется метод `tx_data.info()`, который показывает типы переменных в полях, но почему-то многие из них имеют тип `object` (т.е. «неопределенный» тип). Так произошло из-за ошибок в данных. Какие ошибки во входных данных вы видите и как бы вы их исправили (например, при помощи каких методов `pandas`)?

5. Опишите кратко последовательность получения любой метрики, рассмотрев то, что вам не совсем понятно, и попытавшись объяснить это себе самостоятельно или при помощи сети Интернет.

6. Дополнительное задание. Сформулируйте гипотезу, для чего может быть использован представленный ниже код и как он

может быть доработан. Было бы полезно вынести данный класс в отдельный модуль и для чего мы могли бы использовать такой модуль при аналитике данных?⁴⁴

```
class Metric4(Metric):  
    def __init__(self, metric_1, metric2):  
        self.metric_1 = metric_1  
        self.metric_2 = metric_2  
    def compute(self):  
        return self.metric_1 + self.metric_2
```

⁴⁴ Для выполнения дополнительного задания вам помогут материалы одного из предыдущих занятий по теме объектно-ориентированного программирования (начиная со стр. 99).

МАШИННОЕ ОБУЧЕНИЕ (АНАЛИТИКА БОЛЬШИХ ДАННЫХ)

TENSORFLOW KERAS. РЕШЕНИЕ МАТЕМАТИЧЕСКИХ ЗАДАЧ⁴⁵

Цели выполнения работы

Изучение основ работы с библиотекой для машинного обучения TensorFlow и надстройкой над ней — Keras. Получение базовых знаний о нейронных сетях и практических навыков работы с ними. Решение простейших математических задач при помощи нейронных сетей.

Порядок выполнения работы

Рассмотрим библиотеку TensorFlow и ее надстройку Keras

TensorFlow

TensorFlow — открытая программная библиотека для машинного обучения, разработанная компанией Google для решения задач построения и тренировки нейронной сети с целью автоматического нахождения и классификации образов. Применяется как для ис-

⁴⁵ Разработано на основе русскоязычной книги — [46] в списке источников и литературы.

следований, так и для разработки собственных продуктов Google. Основной API (англ. *application programming interface* — «программный интерфейс приложения») для работы с библиотекой реализован на языке Python, также существуют реализации для языков C#, C++, Java, JavaScript и др.

Работа с TensorFlow состоит в построении и выполнении графа вычислений. Граф вычислений — это конструкция, которая описывает то, каким образом будут проводиться вычисления. В классическом императивном программировании пишется код, который выполняется построчно. В TensorFlow привычный императивный подход к программированию необходим только для вспомогательных целей. Основа TensorFlow — это создание структуры, задающей порядок вычислений. Программы естественным образом структурируются на две части — составление графа вычислений и выполнение вычислений в созданных структурах.

В TensorFlow граф состоит из плейсхолдеров, переменных и операций. Из этих элементов можно собрать граф, в котором будут вычисляться тензоры. Тензоры — многомерные массивы, они служат «топливом» для графа. Тензором может быть как отдельное число, вектор признаков из решаемой задачи или изображение, так и целый пакетный файл описаний объектов или массив из изображений. Вместо одного объекта мы можем передать в граф массив объектов, и для него будет вычислен массив ответов.

В TensorFlow тензоры являются многомерными массивами, которые текут (англ. *flow*) через узлы графа. Узлами графа являются операции (рис. 68).

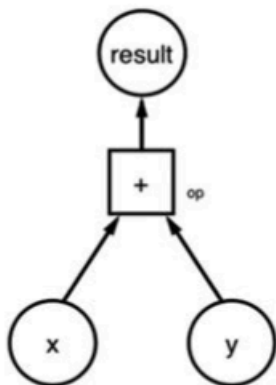


Рис. 68. Граф, выполняющий операцию $X + Y$

Keras

Keras — открытая нейросетевая библиотека, написанная на языке Python, представляющая собой надстройку над фреймворками DeepLearning4j, TensorFlow и Theano (рис. 69)⁴⁶, нацелена на оперативную работу с сетями глубинного обучения, при этом спроектирована так, чтобы быть компактной, модульной и расширяемой. Эта библиотека содержит многочисленные реализации широко применяемых строительных блоков нейронных сетей, таких как слои, целевые и передаточные функции, оптимизаторы, и множество инструментов для упрощения работы с изображениями и текстом. Позволяет работать с TensorFlow не как с графами, а как с абстракциями, т.е. с нейронными сетями.



Рис. 69. TensorFlow с надстройкой Keras

⁴⁶ Изначально Keras был построен «вокруг» библиотеки Theano, затем поддерживал несколько бэкендов (Theano, CNTK, MXNet, Tensorflow), а начиная с 2020 г. осталась возможность работать только с Tensorflow, так как теперь разработкой Keras занимается команда Tensorflow, и Keras стал поставляться вместе с библиотекой Tensorflow. Но возможность использовать другие бэкенды («движки») сохраняется, если использовать Keras версий 2.3 и ниже (по состоянию на январь 2022 г. актуальная версия Keras — 2.8).

Среды тензорных вычислений

TensorFlow способен выполняться на следующих *устройствах*:

- персональных компьютерах общего назначения (возможна работа как на центральном, так и на графических процессорах);
- встраиваемых и мобильных системах (нативно на смартфонах с операционной системой Android/iOS, на одноплатных компьютерах с Raspbian, в веб-браузере);
- специализированных тензорных процессорах (TPU Google, рис. 70), которые Google предоставляет в формате облачной среды вычисления и доступ к которым очень прост даже для новичка: необходимо купить подписку и станет возможным обучать собственные модели на специализированном и предназначенном для этого оборудовании, имеющем наиболее высокую для этих задач производительность (подробнее в источнике [47]).

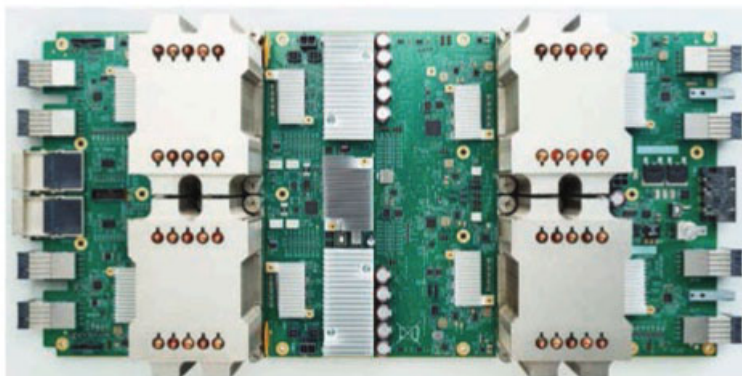


Рис. 70. Тензорный процессор Google

Требования TensorFlow (для Python 3 версии⁴⁷) к операционной системе:

- Ubuntu и Debian с ядром (kernel) версии 4.14 или выше (64-bit);
- macOS 10.12.6 или выше (64-bit, без поддержки GPU);
- Windows 7 или выше (64-bit);
- Raspbian 9.0 или выше.

⁴⁷ Так как существуют реализации TensorFlow и для других языков программирования.

Пример 1. $Y = 3X + 1$ (основы работы с TensorFlow Keras: нейронные сети, их функции активации, веса, смещения и типы ошибок)

Приступим к реализации нейронной сети TensorFlow Keras для решения простейшей математической задачи. Рассмотрим следующие наборы чисел, представленные в таблице ниже (табл. 34).

Таблица 34

Обучающая выборка для $Y = 3X + 1$

X:	-1	0	1	2	3	4
Y:	-2	1	4	7	10	13

Данные пары значений соответствуют результату уравнения $Y = 3X + 1$, и они необходимы для обучения нейронной сети, которую нам предстоит написать.

Начнем с подключения необходимых библиотек. Здесь мы импортируем TensorFlow и назовем его «tf» для простоты использования. Далее мы импортируем библиотеку с именем «numpy», которая помогает легко и быстро представлять наши данные в виде списков. Структура для определения нейронной сети как набора последовательных слоев называется «keras», поэтому мы импортируем ее тоже.

```

---in↓---
import tensorflow as tf #библиотека для машинного обучения
import numpy as np #математическая библиотека (поддержка
многомерных массивов и
                                #высокоуровневых математических
функций над ними
from tensorflow import keras #библиотека высокоуровневого
API для tensorflow и не только нее
from tensorflow.keras.utils import plot_model #утилита
для визуализации полученной модели

```



```
print(tf.__version__)
print(keras.__version__)
print(np.__version__)
---↑in_out↓---
2.3.0
2.4.0
1.21.2
---↑out---
```

Далее мы создадим простейшую из возможных нейронных сетей. У нее будет 1 слой («`keras.layers.Dense()`» — добавление слоя), в котором 1 нейрон (`units=1`), и входная в него величина имеет только размерность 1, т.е. на вход данного слоя приходит только одно значение (`input_shape=[1]` или `input_dim=1` (*dim* — *dimension* — «размерность»)).

Функции активации

Также можно указать и функцию активации этого слоя «`activation='linear'`», но линейная функция активации является значением по умолчанию, поэтому не будем это отдельно указывать. Описание линейной функции и некоторых основных функций активации кратко изложено в таблице ниже (табл. 35)⁴⁸, где X — это входная переменная.

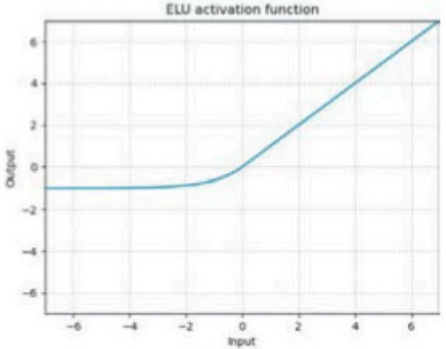
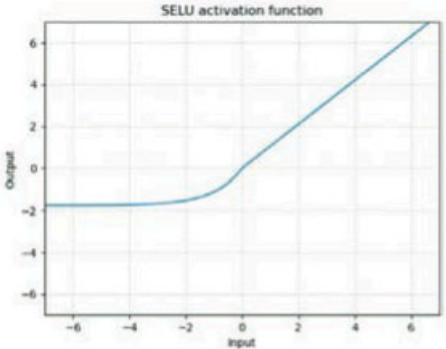
Финальный вид кода для создания описанной модели нейронной сети выглядит так:

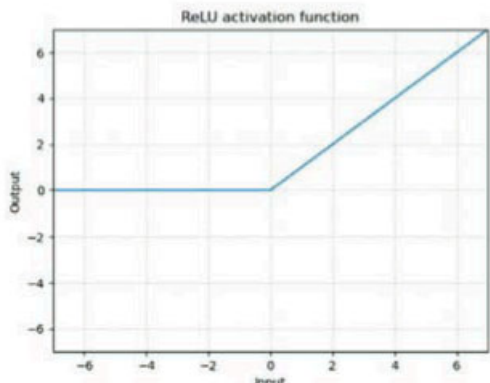
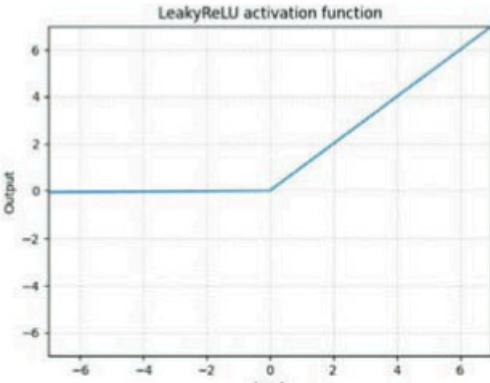
```
---in↓---
#use_bias=True - использовать нейрон смещения (хотя он
и так по умолчанию используется)
model = tf.keras.Sequential(
    [keras.layers.Dense(units=1, input_shape=[1],
use_bias=True)])
---↑in_out↓---
---↑out---
```

⁴⁸ Ознакомиться с доступными функциями активации в полной мере или узнать, как создать свою собственную, можно в официальной документации [48].

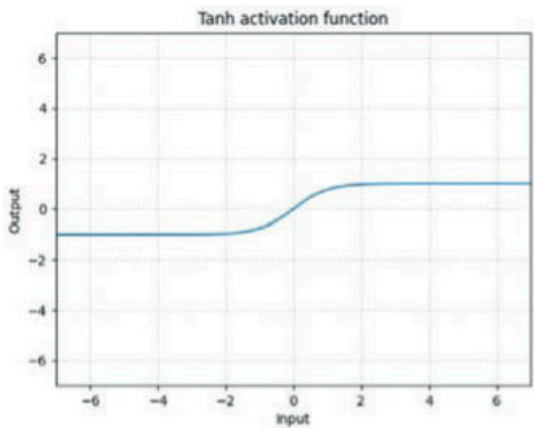
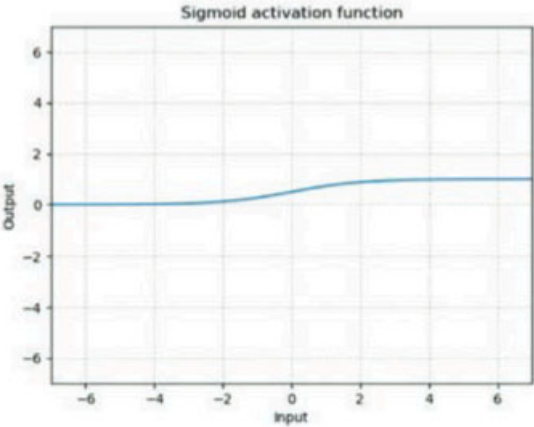
Таблица 35

Краткое описание основных доступных функций активации

Название функции	Область значений	Синтаксис
<p>Экспоненциальная линейная функция (англ. <i>Exponential linear unit, ELU</i>).</p> $\text{ELU}(x) = \begin{cases} \alpha(e^x - 1) & x \leq 0; \\ x & x > 0, \end{cases} \quad (24)$ <p>где $\alpha = 1,67326...$</p> 	$(-\alpha, \infty)$	<code>keras.activations.elu(x, alpha=1.0)</code>
<p>Масштабированная экспоненциальная линейная функция (англ. <i>Scaled exponential linear unit, SELU</i>).</p> $\text{SELU}(x) = \lambda \begin{cases} \alpha(e^x - 1) & x \leq 0; \\ x & x > 0 \end{cases} \quad (25)$ <p>ИЛИ</p> $\text{SELU}(x) = \lambda * (\max(0, x) + \min(0, \alpha * (\exp(x) - 1))),$ <p>где $\lambda = 1,0507...$ и $\alpha = 1,67326...$</p> 	$(-\alpha\lambda, \infty)$	<code>keras.activations.selu(x)</code>

Название функции	Область значений	Синтаксис
<p>Выпрямитель или функция активации ReLU (<i>Rectified Linear Unit</i>). Является функцией активации, определяемой как положительная часть его аргумента.</p> $ReLU(x) = (x)^+ = \max(0, x) \quad (26)$ 	$[0, \infty)$	<code>keras.activations.relu(x)</code>
<p>Линейный выпрямитель с «утечкой» (англ. <i>Leaky rectified linear unit</i>, <i>Leaky ReLU</i>).</p> $LeakyReLU(x) = \max(0, x) + \alpha \cdot \min(0, x)$ <p>или</p> $LeakyReLU(x) = \begin{cases} x, & x \geq 0; \\ \alpha \times x, & \text{иначе,} \end{cases} \quad (27)$ <p>где по умолчанию $\alpha = 0,01$.</p> 	$(-\infty, \infty)$	<code>keras.activations.relu(x, alpha=0.0, max_value=None, threshold=0.0)</code>

Окончание табл. 35

Название функции	Область значений	Синтаксис
<p>Гиперболический тангенс ().</p> $\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \quad (28)$ 	$(-1, 1)$	<code>keras.activations.tanh(x)</code>
<p>Логистическая (сигмоида или гладкая ступенька).</p> $\text{sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)} \quad (29)$ 	$(0, 1)$	<code>keras.activations.sigmoid(x)</code>
<p>Линейная функция активации (или тождественная).</p> $\text{linear}(x) = x \quad (30)$	$(-\infty, \infty)$	<code>keras.activations.linear(x)</code>

Напишем код для компиляции нашей нейронной сети. Когда мы делаем это, мы должны указать две функции: «потери» и «оптимизатор».

Функция «потери» («loss») вычисляет разность «угаданных» (вычисленных) ответов с известными правильными ответами и измеряет, насколько точно произведено решение. Основываясь на результатах функции потерь, TensorFlow «попытается» минимизировать потери, используя заданный алгоритм оптимизации на каждом шаге обучения.

Укажем программе использовать «среднеквадратичную ошибку» («mean_squared_error») для потерь и «стохастический градиентный спуск» («sgd») для оптимизатора (список алгоритмов оптимизации доступен в официальной документации [49]). Обычно применяют именно эти установки параметров потерь и оптимизатора.

```
---in↓---
model.compile(optimizer='sgd', loss='mean_squared_error')
---↑in_out↓---
---↑out---
```

Далее мы создадим обучающую выборку для нашего уравнения при помощи библиотеки `numpy`, которая, как известно, предоставляет нам широкую функциональность по работе с массивами. Для этого используем функцию `np.array`:

```
---in↓---
xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-2.0, 1.0, 4.0, 7.0, 10.0, 13.0], dtype=float)
---↑in_out↓---
---↑out---
```

Код, необходимый для определения нейронной сети, написан. Теперь обучим нашу нейросеть. Процесс обучения нейронной сети, где она «изучает» отношения между X и Y , протекает при вызове метода «`model.fit`».

Здесь программа пройдет цикл, о котором было написано ранее: получит догадку, измерит, насколько она точна (иначе говоря,

потерю), использует оптимизатор, чтобы сделать еще одну догадку и т.д.

TensorFlow повторит описанный цикл обучения заданное количество раз, называемое количеством эпох. Вы указываете столько эпох, сколько достаточно для полного обучения вашей модели (обычно обучение останавливают при достижении заданной точности⁴⁹).

Когда вы запустите этот код, вы увидите, что по мере прогрессирования обучения нейронной сети потери будут уменьшаться с каждой следующей эпохой и станут очень маленькими. Не факт, что для достижения заданной точности нужно 40 эпох, поэтому вы можете поэкспериментировать с их количеством.

Историю обучения мы будем записывать в переменную `history`, чтобы потом, например, построить график уменьшения ошибки в процессе обучения:

```
---in↓---
history = model.fit(xs, ys, epochs=40)
---↑in_out↓---
Epoch 1/40
1/1 [=====] - 0s 977us/step - loss:
107.5874
Epoch 2/40
1/1 [=====] - 0s 0s/step - loss:
84.6572
#вывод обрезан
Epoch 39/40
1/1 [=====] - 0s 0s/step - loss:
0.0416
Epoch 40/40
1/1 [=====] - 0s 978us/step - loss:
0.0385
---↑out---
```

Метод «`model.evaluate()`» возвращает значения потерь при следующих аргументах: входные данные, выходные данные.

⁴⁹ То есть количество эпох обычно указывают вручную — пишут число. Но существует возможность, например, закончить обучение раньше, если точность уже удовлетворительна. Любые подобные манипуляции выполняются при помощи `callback`-функций (см. подробнее в источнике [50]).

То есть мы можем получить значение потерь (среднеквадратичной ошибки) для тех данных, которые подадим.

Этот метод вернет минимальное значение потери обученной модели, которое достигается при последней эпохе обучения, так как при ней модель считается наиболее точной:

```
---in↓---
print(model.evaluate(xs, ys))
---↑in_out↓---
1/1 [=====] - 0s 0s/step - loss:
0.0359
0.03589404746890068
---↑out---
```

После того как вы обучили модель для решения уравнения $Y = 3X + 1$, вы можете использовать метод «`model.predict()`», чтобы он вычислил Y для указанного и ранее неизвестного X :

```
---in↓---
print(model.predict([10.0]))
---↑in_out↓---
[[30.14795]]
---↑out---
```

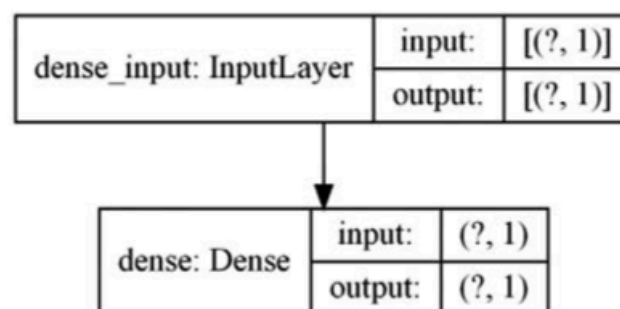


Рис. 71. Структура нейросети для решения уравнения $Y = 3X + 1$

Чтобы сохранить структуру нейронной сети в виде картинки, воспользуемся следующим кодом, и программа сохранит модель в файл `model_example1.png` (и отобразит ее, как представлено на рис. 71, если код выполняется в Jupyter-блокноте):

```

---in↓---
plot_model(model,
            to_file='model_example1.png',
            show_shapes=True,
            show_layer_names=True,
            rankdir='TB',
            expand_nested=False,
            dpi=96)
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 71)
---↑out---

```

Чтобы объяснить, что такое нейросеть и как она работает, выведем значение веса «weight» и значение нейрона смещения «bias». На рисунке после кода (рис. 72) показано графическое объяснение, как работает полученная нейросеть.

```

---in↓---
for layer in model.layers:
    weights = layer.get_weights() # list of numpy arrays
print(weights)
print(f'Weight = {round(weights[0][0][0], 2)}')
print(f'Bias = {round(weights[1][0], 2)}')
[array([[2.8935115]], dtype=float32), array([1.2128329],
dtype=float32)]
---↑in_out↓---
Weight = 2.8900000104904175
Bias = 1.21000000381469727
---↑out---

```

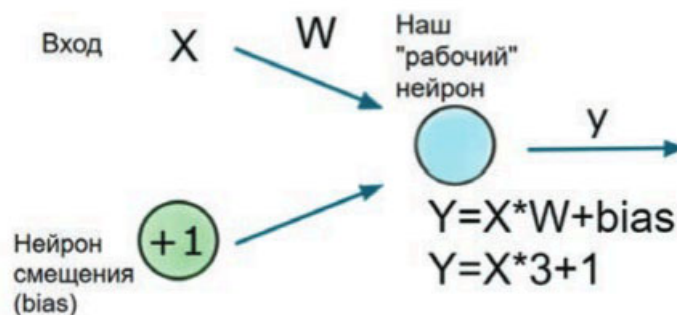


Рис. 72. Структура нейросети для решения уравнения $Y = 3X + 1$

В зависимости от сложности решаемых задач количество нейронов на входных/выходных слоях и на скрытых слоях (на рис. 72) может различаться. Нейроны смещения есть только на скрытых слоях — по 1 нейрону смещения на весь скрытый слой. Слои всегда полносвязные: все нейроны связаны между собой весом, но вес может быть нулевым.

Каждый входной сигнал умножается на вес, а затем подается на каждый нейрон следующего слоя. Далее нейроны суммируют эти взвешенные сигналы, а затем умножают их сумму на функцию активации, добавляют смещение и передают сигнал на каждый нейрон следующего слоя (через новые веса связей).

Смещение задается только для всего слоя целиком. Нейроны смещения по умолчанию включены, но их можно отключить. Обычно про наличие нейронов смещения отдельно не упоминают.

Слои, несущие основную функциональность, часто называют скрытыми слоями.

Для решения данной задачи ($Y = 3X + 1$) достаточно сети с 1 нейроном на скрытом слое. Выведем полную конфигурацию первого слоя нейронной сети, а также график зависимости ошибки обучения от эпохи обучения (рис. 73):

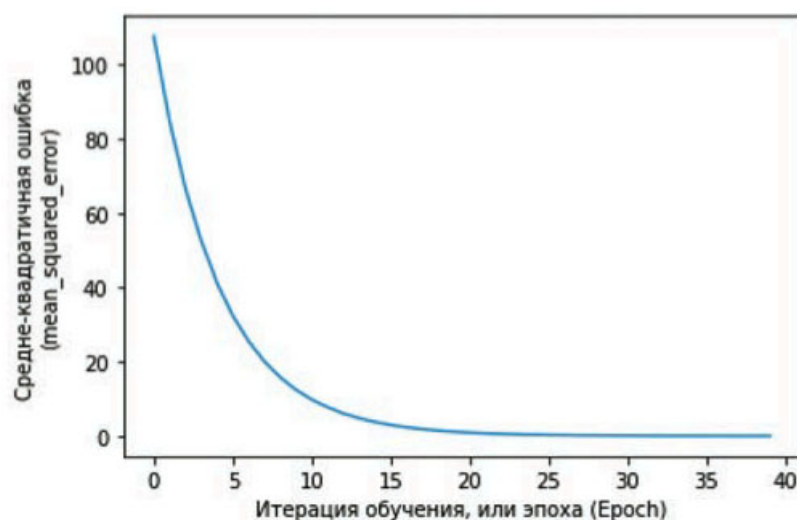


Рис. 73. Метрика «ошибка обучения» для $Y = 3X + 1$

```
---in↓---
import matplotlib.pyplot as plt
%matplotlib inline
```



```

print(history.history.keys())

plt.title('Изменение метрики обучения "ошибка"
в зависимости от итерации обучения\n')
plt.xlabel('Итерация обучения, или эпоха (Epoch)')
plt.ylabel('Среднеквадратичная ошибка\n(mean_squared_
error)')
plt.plot(history.history['loss'])

model.get_config()['layers'][1]
---↑in_out↓---
dict_keys(['loss'])
{'class_name': 'Dense',
 'config': {'name': 'dense',
            'trainable': True,
            'batch_input_shape': (None, 1),
            'dtype': 'float32',
            'units': 1,
            'activation': 'linear',
            'use_bias': True,
            'kernel_initializer': {'class_name': 'GlorotUniform',
                                   'config': {'seed': None}},
            'bias_initializer': {'class_name': 'Zeros', 'config': {}}},
#часть вывода вырезана, а часть – на рисунке ниже (рис. 73)
---↑out---

```

Ошибки (MSE, RMSE и MAE)

Для вычисления среднеквадратической ошибки (MSE) все отдельные остатки регрессии возводятся в квадрат, суммируются, сумма делится на общее число ошибок (формула (31)):

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2, \quad (31)$$

где n — общее число ошибок; Y_i — реальное значение прогнозируемой переменной (или вектора переменных); \hat{Y}_i — предсказанное значение (или вектор значений).

Также существует отдельный подвид — $RMSE = \sqrt{MSE}$, где буква R расшифровывается как root, т.е. корень. Удобство $RMSE$ заключается в том, что она измеряется в тех же единицах, что и Y_i . В этом смысле она является некоторой альтернативой средней абсолютной ошибки MAE , вычисляемой по формуле (32), представленной ниже:

$$MAE = \frac{1}{n} \sum_{i=1}^n |Y_i - \hat{Y}_i|. \quad (32)$$

Разница между $RMSE$ и MAE заключается в том, что первая метрика придает большим выбросам относительно больший вес, так как ошибки возводятся в квадрат, а MAE имеет у всех ошибок одинаковый вес, так как вычисляется линейно (без квадрата). $RMSE$ всегда будет больше или равна MAE ; чем больше разница между ними, тем больше дисперсия в отдельных ошибках в выборке. Если $RMSE = MAE$, то все ошибки имеют одинаковую величину.

Выбор типа ошибки зависит от задачи и критичности ошибок из-за выбросов. Но, пожалуй, по простоте понимания их можно упорядочить от простого уровня по пониманию к сложному в следующем порядке: MAE , $RMSE$ и MSE . Так как первые две измеряются в тех же размерностях, что и предсказанное значение, например, можно утверждать, что если предсказанное число равно 20, а $RMSE$ или MAE равны 2 (а $MSE = RMSE^2 = 4$), то в среднем число 20 может отличаться на 2 в ту или иную сторону.

Обычно при обучении нейронных сетей используют MSE как функцию потерь, так как при больших ошибках на начальном этапе обучения процесс обучения выполняется быстрее. При оценке же нагляднее использовать $RMSE$.

Приведем пример вычисления среднеквадратичной ошибки (MSE) при помощи функциональности `keras`. Вычислим MSE для реальных значений `y_true` и предсказанных `y_pred`⁵⁰:

⁵⁰ Данные значения не имеют ничего общего с рассматриваемым в данном разделе примером. Просто абстрактный пример. Таким же образом вычисляется MSE в процессе обучения или оценивания нейросети.

```

---in↓---
y_true = [[0., 1.], [0., 0.]]
y_pred = [[1., 1.], [1., 0.]]
mse = tf.keras.losses.MeanSquaredError()
mse(y_true, y_pred).numpy()
---↑in_out↓---
0.5
---↑out---
```

Вычисление значения $MSE = 0.5$ осуществляется представленным ниже способом:

```

---in↓---
#**-оператор возведения в степень
((1-0)**2+(1-1)**2+(1-0)**2+(0-0)**2)/4
---↑in_out↓---
0.5
---↑out---
```

Визуализируем различие между обучающей выборкой и предсказанными для входных данных из диапазона $[-1; 5)$ значениями:

```

---in↓---
x_pred = []
y_pred = []
for var in range(-1, 5):
    y_pred.append(model.predict([var])[0][0])
    x_pred.append(var)

import plotly.offline as pyoff
import plotly.graph_objs as go

plot_data = [
    go.Scatter(
        x=x_pred,
        y=y_pred,
        name='Предсказание',
    ),
    go.Scatter(
        x=xs,
```

```

        y=ys,
        name='Реальные данные',
    )]

plot_layout = go.Layout(
    xaxis={"type": "linear"},
    title=
        'График по реальным параметрам и график предсказаний
        (нужно сильно приблизить, чтобы увидеть ошибку)'
)

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 74)
---↑out---
```

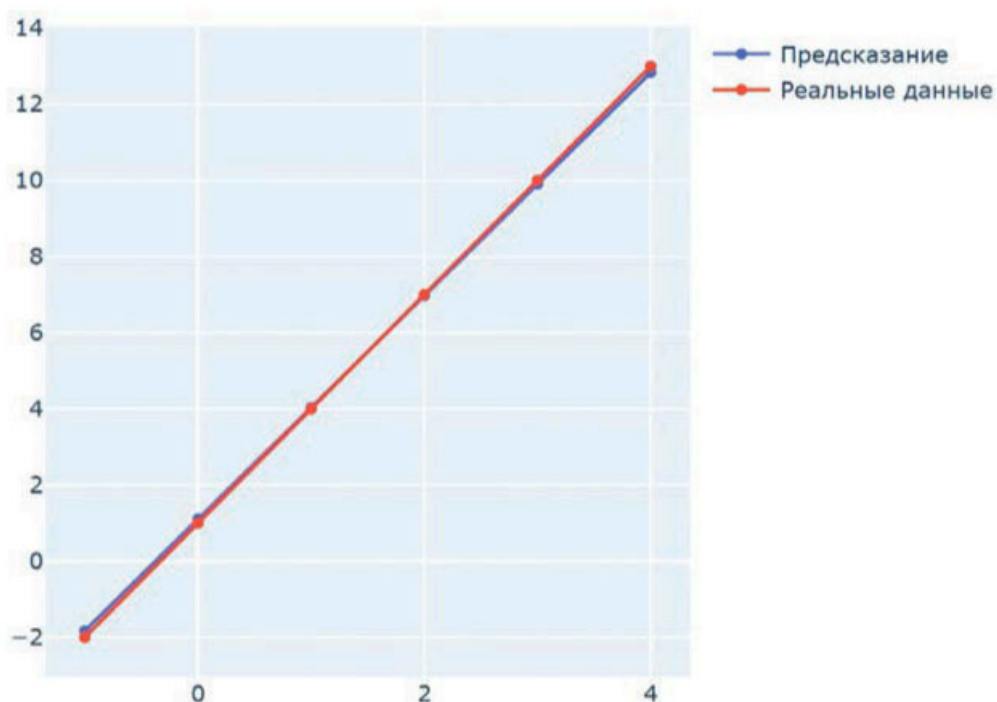


Рис. 74. Сравнение реальных данных с предсказанными для $Y = 3X + 1$

График является интерактивным, так как выводится при помощи plotly. Вы можете увеличить масштаб («приблизить»), чтобы лучше разглядеть отличия двух графиков.

Набор данных, содержащий соответствующие друг другу значения входных и выходных параметров, называют обучающей выборкой (когда речь идет про обучение с учителем). Но часто такой набор разделяют с некоторым соотношением, чтобы получить часть данных для обучения и условно независимую часть данных для оценки точности нейронной сети.

Обычно в обучающую выборку включают 80 % всего датасета, а оставшуюся часть используют как тестовую. В данном случае мы не создаем отдельную тестовую выборку. Это не совсем корректно, но для текущего графика это не сильно принципиально. А вот оценку точности модели при помощи метода `model.evaluate()`, которую ранее мы использовали, следовало бы проводить на отдельном тестовом наборе данных. Но для этого тестовый датасет сначала нужно было бы создать или получить методом отсечения от обучающего, если бы объем обучающей выборки был достаточен для того, чтобы отсечь от нее часть значений.

В итоге написан код на языке Python при помощи фреймворка TensorFlow с надстройкой Keras, являющийся реализацией нейросети для решения уравнения $Y = 3X + 1$.

Далее будет приведено еще три примера для решения математических уравнений при помощи нейронных сетей. Их код отличается только структурой нейросети (например, есть различия в количестве нейронов или типе функций активации) и предобработкой данных, так как их размерность различается (в том числе это влияет и на размерность входного и выходного слоя нейросети). Код для удобства будет представлен единой ячейкой Jupyter.

Пример 2. $Y = 2X$

Ниже представлен пример реализации нейронной сети для решения уравнения $Y = 2X$. Для решения данной функции достаточно 30 нейронов с сигмоидальной функцией активации со 100 эпохами.

Обратите внимание, что в `model.fit()` указан параметр `verbose=0` (с англ. — «многословие»), что означает, что вывод при обучении будет минимальный: не будет выводиться информация по каждой эпохе, а выведется только итоговая потеря обучения. По умолчанию `verbose=1`.

Также используется метод `Keras_backend.clear_session()`, который необходим для очистки TensorFlow и Keras, так как ранее в этом же ноутбуке запускалась другая модель, которая уже была обучена. Именно чтобы «забыть все прошлые наработки» и работать с новой моделью «с чистого листа», и используется данный метод (очищает все графы, веса, начальные условия генератора весов и т.п.).

```
---in↓---
from tensorflow.keras import backend as Keras_backend

Keras_backend.clear_session() #для очистки нумерации
слюев в Keras

model = tf.keras.Sequential([
    keras.layers.Dense(30, input_dim=1,
        activation='sigmoid'),
    #keras.layers.Dense(10, input_dim=30,
activation='sigmoid'), #так бы мы создали второй слой
    keras.layers.Dense(1, activation='linear'
        ) #данный слой необходим, чтобы
из множества выходов
    #сделать лишь один
])

model.compile(optimizer='sgd', loss='mean_squared_error')

ys = []
xs = np.array([
    -10.0, -9.0, -8.0, -7.0, -6.0, -5.0, -4.0, -3.0,
    -2.0, -1.0, 0.0, 1.0, 2.0,
    3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
],
                dtype=float)

def func(x):
    return 2 * x

for i in range(len(xs)):
    #yz[i]=func(xz[i])
```



```

    ys.append(func(xs[i]))
ys = np.array(ys)

print('xs:', xs)
print('ys:', ys)

history = model.fit(xs, ys, epochs=100,
                    verbose=0) #verbose=0-не печатать
ничего
print(model.evaluate(xs, ys))

plt.title(
    'Изменение метрики обучения "ошибка" в зависимости от
итерации обучения\n')
plt.xlabel('Итерация обучения, или эпоха (Epoch)')
plt.ylabel('Среднеквадратичная ошибка\n(mean_squared_
error)')
plt.plot(history.history['loss'])

predict_value = 8.0
print('Число для предсказания: ', predict_value)
print('Предсказание: ', model.predict([predict_value]))

if (func(predict_value) * 1.1 > model.predict([predict_value])
    and func(predict_value) * 0.9 < model.
predict([predict_value])):
    print('Точность для predict_value больше 10%, что
удовлетворительно.')
else:
    print('Точность для predict_value меньше 10%, что НЕ
удовлетворительно.')

plot_model(model,
            to_file='model_example2.png',
            show_shapes=True,
            show_layer_names=True,
            rankdir='TB',
            expand_nested=False,
            dpi=96)
---↑in_out↓---
xs: [-10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.  -2.  -1.
  0.   1.   2.   3.]

```

```

    4.    5.    6.    7.    8.    9.   10.]
ys: [-20. -18. -16. -14. -12. -10.  -8.  -6.  -4.  -2.
    0.    2.    4.    6.
    8.   10.   12.   14.   16.   18.   20.]
1/1 [=====] - 0s 977us/step
- loss: 2.5633
2.5633139610290527
Число для предсказания: 8.0
Предсказание: [[14.920003]]
Точность для predict_value больше 10%, что удовлетворительно.
#часть вывода представлена на рисунках ниже (рис. 75 и рис. 76)
---↑out---
```

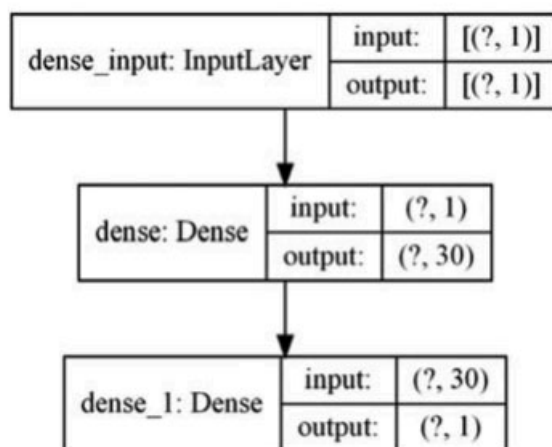


Рис. 75. Структура нейросети для решения уравнения $Y = 2X$

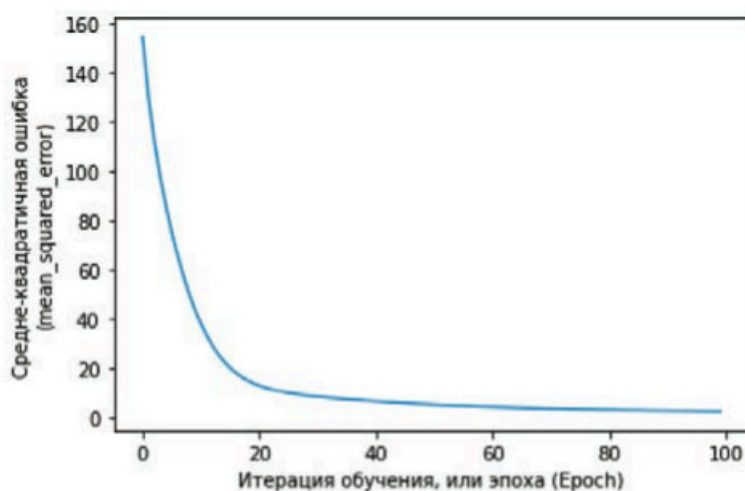


Рис. 76. Метрика «ошибка обучения» для $Y = 2X$

Вывод весов

Выведем веса обученной нейросети при помощи кода ниже, чтобы разобраться более основательно, как работает такая математическая абстракция, как «нейронная сеть». Если представить графически данную нейросеть, то получится результат, похожий на рис. 77.

```

---in↓---
for layer in model.layers:
    weights = layer.get_weights() # list of numpy arrays
print(f'Веса: {np.around(weights[0],1)}')
print(f'Нейрон смещения: {np.around(weights[1],1)}')
---↑in_out↓---
Веса: [[-1.3]
      [-1. ]
      [ 0.8]
      #часть весов вырезана в целях сокращения книги
      [ 1.3]
      [-1.3]]
Нейрон смещения: [0.3]
---↑out---
```

Числа на картинке (рис. 77) могут не соответствовать значениям весов из вывода выше, потому что распределение весов случайно, и в любом случае можно сменить количество эпох обучения, тогда числа тем более будут отличаться.

Также числа могут отличаться даже при запуске с разных компьютеров! Поэтому картинка представлена просто для понимания, а не для поиска в ней ошибок. В любом случае порядок нейронов и их весов не имеет значения — сеть в итоге будет работать одинаково.

Классическое представление структуры нейронных сетей выглядит именно так: изображают нейроны, обозначенные кружками, выстраивают их в вертикальные слои и при помощи весов (линий) связывают нейроны между собой.

Как уже говорилось, обычно нейронные сети всегда полносвязные (иначе было бы сложно их обучать), а скрытыми (или промежуточными) слоями называют все слои, которые не являются входными или выходными.

Скрытыми слоями называются потому, что их входы и выходы условно неизвестны или действительно неизвестны для внешних по отношению к нейронной сети программ и пользователю. Грубо говоря, скрытые слои — это черный ящик, который был обучен, и его остается только использовать, подавая входные значения и получая выходные.

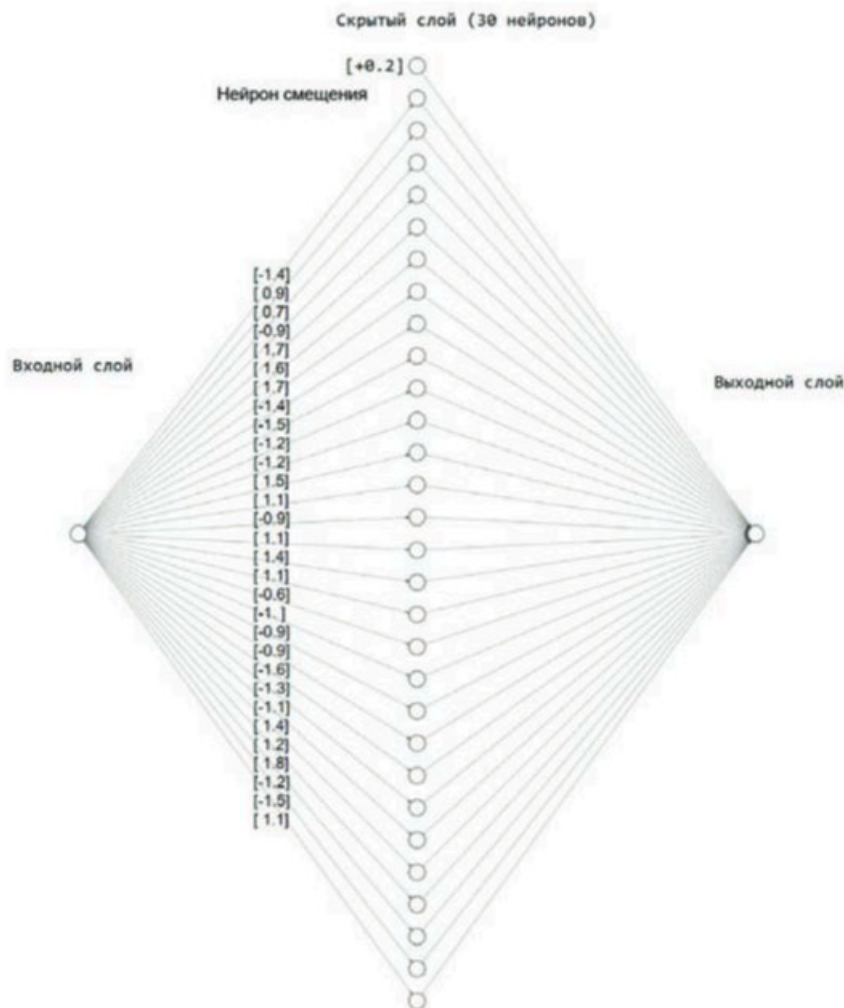


Рис. 77. Структура нейросети для уравнения $Y = 2X$

Выведем график реальных и предсказанных значений (рис. 78):

```

---in↓---
x_pred = []
y_pred = []

```

```

for var in range(-10, 11):
    y_pred.append(model.predict([var])[0][0])
    x_pred.append(var)

plot_data = [
    go.Scatter(
        x=x_pred,
        y=y_pred,
        name='Предсказание',
    ),
    go.Scatter(
        x=xs,
        y=ys,
        name='Реальные данные', )]

plot_layout = go.Layout(
    xaxis={"type": "linear"},
    title='График по реальным параметрам и график
предсказаний (сильно приблизьте, чтобы увидеть ошибку)')

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 78)
---↑out---

```

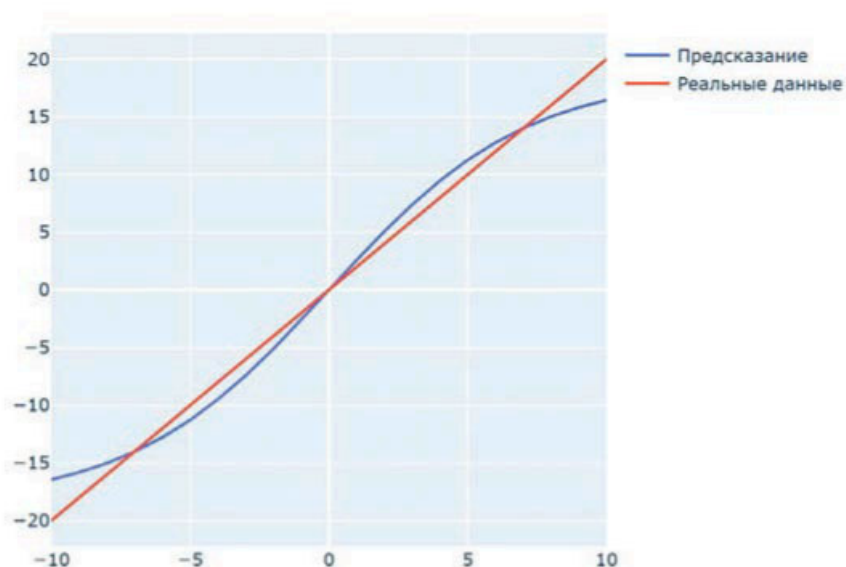


Рис. 78. Сравнение реальных данных с предсказанными для $Y = 2X$

На графике из вывода кода выше (рис. 78) видно, что расхождения реальных и предсказанных значений довольно велики. Но при написании данной книги не стояло цели представить оптимальные для решения поставленных задач структуры нейросетей. Приведены лишь некоторые опорные примеры, на основе которых можно самостоятельно разработать оптимальные решения.

Пример 3. $Y = \sin(X)$

Ниже представлен пример реализации нейронной сети для решения уравнения $Y = \sin(X)$. Для решения данной функции достаточно 50 нейронов с `relu` функцией активации с 300 эпохами.

```
---in↓---
Keras_backend.clear_session() #для очистки нумерации
слоев в Keras

model = tf.keras.Sequential([
    keras.layers.Dense(50, input_dim=1,
                        activation='relu'), #функция
активации "выпрямитель"
    keras.layers.Dense(1, input_dim=50,
                        activation='linear') #данный слой
необходим,
    #чтобы из множества выходов сделать лишь один,
линейная функция активации
])

sgd = keras.optimizers.SGD(learning_rate=0.01,
momentum=0.9,
                           nesterov=True)
#сконфигурируем оптимизатор
#для лучшей оптимизации обучения нейросети для решения
функции sin(x)

model.compile(optimizer=sgd, loss='mean_squared_error')

ys = []
xs = np.arange(-2, 2, 0.1)
```

```

# importing "math" for mathematical operations
import math

def func(x):
    return math.sin(x)

for i in range(len(xs)):
    #yz[i]=func(xz[i])
    ys.append(func(xs[i]))
ys = np.array(ys)

#print('xs:', xs)
#print('ys:', ys)

history = model.fit(xs, ys, epochs=300,
                    verbose=0) #verbose=0-не печатать
ничего
print(model.evaluate(xs, ys))

plt.title(
    'Изменение метрики обучения "ошибка" в зависимости от
итерации обучения\n')
plt.xlabel('Итерация обучения, или эпоха (Epoch)')
plt.ylabel('Среднеквадратичная ошибка\n(mean_squared_
error)')
plt.plot(history.history['loss'])

predict_value = 1.26

print('Число для предсказания: x=', predict_value, ' y=',
func(predict_value))

print('Предсказание: ', model.predict([predict_value]))

if (func(predict_value) * 1.1 > model.predict([predict_value])
    and func(predict_value) * 0.9 < model.
predict([predict_value])):
    print('Точность для predict_value больше 10%, что
удовлетворительно.')
else:
    print('Точность для predict_value меньше 10%, что НЕ
удовлетворительно.')

```

```
plot_model(model,
            to_file='model_example3.png',
            show_shapes=True,
            show_layer_names=True,
            rankdir='TB',
            expand_nested=False,
            dpi=96)

x_pred = []
y_pred = []
for var in range(-20, 21):
    y_pred.append(model.predict([var / 10])[0][0])
    x_pred.append(var / 10)

plot_data = [
    go.Scatter(
        x=x_pred,
        y=y_pred,
        name='Предсказание',
    ),
    go.Scatter(
        x=xs,
        y=ys,
        name='Реальные данные',
    )
]

plot_layout = go.Layout(
    xaxis={"type": "linear"},
    title=
        'График по реальным параметрам и график предсказаний
(сильно приблизьте, чтобы увидеть ошибку)'
)

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
2/2 [=====] - 0s 977us/step
- loss: 0.0019
0.001947292359545827
```

Число для предсказания: $x = 1.26$ $y = 0.9520903415905158$
Предсказание: $[[0.8796497]]$
Точность для predict_value больше 10%, что удовлетворительно.
#часть вывода представлена на рисунках ниже (рис. 79 и рис. 80)
---↑out---

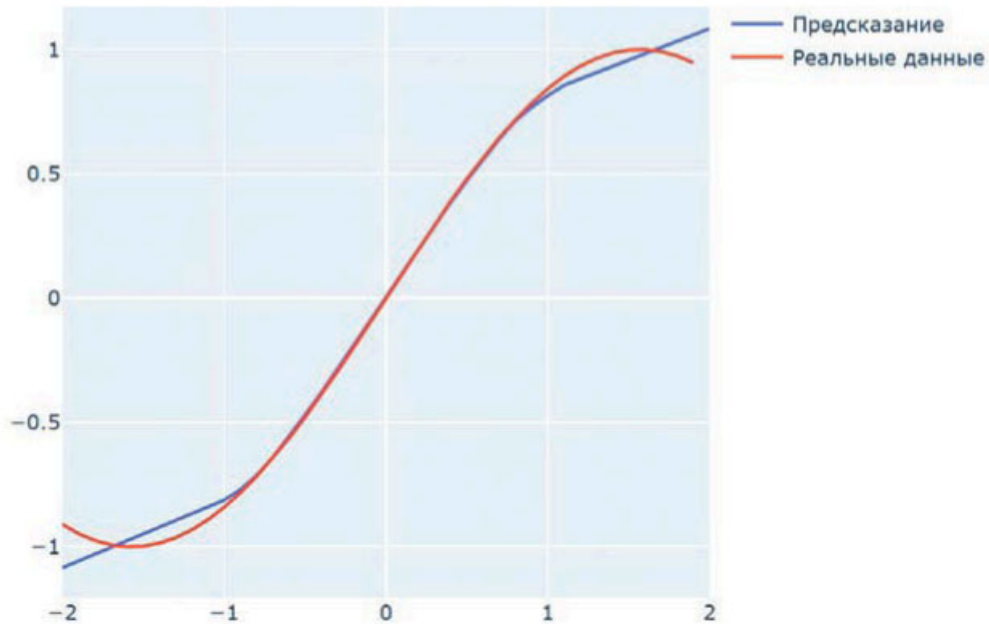


Рис. 79. Сравнение реальных данных с предсказанными для $Y = \sin(X)$

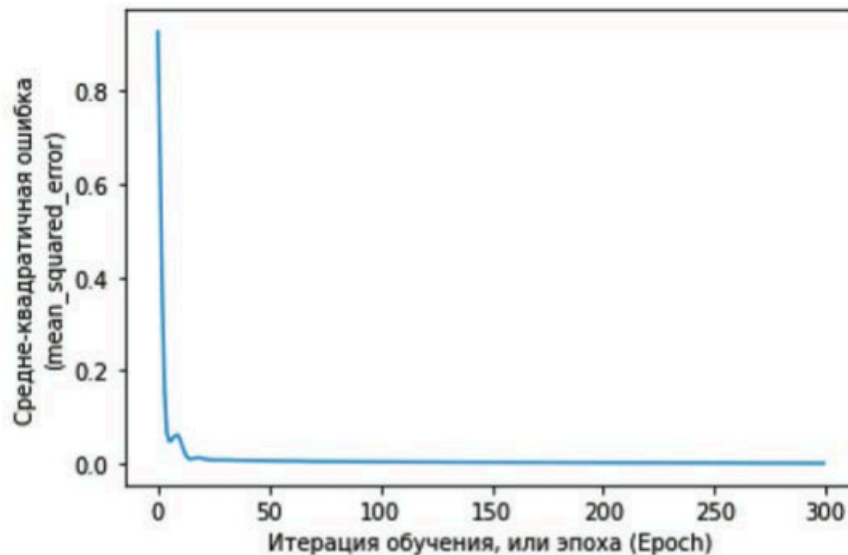


Рис. 80. Метрика «ошибка обучения» для $Y = \sin(X)$

Пример 4. $Y = (K + X) \times X$

Ниже представлен пример реализации нейронной сети для решения уравнения $Y = (K + X) \times X$. Для решения данной функции достаточно 20 нейронов с сигмоидальной функцией активации с 300 эпохами.

```
---in↓---
Keras_backend.clear_session() #для очистки нумерации
слоев в Keras

model = tf.keras.Sequential([
    #keras.layers.Flatten(input_shape=(2,)),
    keras.layers.Dense(units=20, input_dim=2,
activation='sigmoid'),
    #keras.layers.Dense(8, input_dim=20,
activation='sigmoid'),
    keras.layers.Dense(1, activation='linear'
                        ) #данный слой необходим, чтобы
из множества выходов
    #сделать лишь один
])

sgd = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9,
                           nesterov=True)
#сконфигурируем оптимизатор
#для лучшей оптимизации обучения нейросети для решения
функции sin(x)

model.compile(optimizer=sgd, loss='mean_squared_error')

ys = []

xs = np.array([
    -5.9, -5.1, -4.9, -4.1, -3.9, -3.1, -2.9, -2.1, -1.9,
    -0.1, 0.9, 0.1, 1.9,
    1.1, 2.9, 2.1, 3.9, 3.1, 4.9, 4.1, 5.9
],
              dtype=float)
```



```

ks = np.array([
    -5.5, -5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.0,
    -0.5, 0.0, 0.5, 1.0,
    1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0
],
              dtype=float)

def func(k, x):
    return (k + x) * x

for i in range(len(xs)):
    ys.append(func(ks[i], xs[i]))
ys = np.array(ys)

#print('xs:', xs)
#print('ks:', ks)
#print('ys:', ys)

IN = np.array([])
IN = np.column_stack(
    (xs, ks)
) #делает из двух массивов один, где элементы парные
[[x1,k1][x2,k2]...], #именно без запятых между массивами.
Это не то же самое, что списки в Python
#print('IN:', IN)

history = model.fit(IN, ys, epochs=300,
                    verbose=0) #verbose=0-не печатать
ничего

plt.title(
    'Изменение метрики обучения "ошибка" в зависимости от
итерации обучения\n')
plt.xlabel('Итерация обучения, или эпоха (Epoch)')
plt.ylabel('Среднеквадратичная ошибка\n(mean_squared_
error)')
plt.plot(history.history['loss'])

print(model.evaluate(IN, ys))

k = 2.0
x = 3.0
predict_value = np.array([[x, k]])

```

```
print('Числа для предсказания: k=', k, ' x=', x)
#print('predict_value: ', predict_value )

print('Предсказание: ', model.predict(predict_value))

if (func(k, x) * 1.1 > model.predict(predict_value)
    and func(k, x) * 0.9 < model.predict(predict_value)):
    print('Точность для predict_value больше 10%, что
удовлетворительно.')
else:
    print('Точность для predict_value меньше 10%, что НЕ
удовлетворительно.')

plot_model(model,
            to_file='model_example4.png',
            show_shapes=True,
            show_layer_names=True,
            rankdir='TB',
            expand_nested=False,
            dpi=96)

x_pred = []
y_pred = []
k_pred = []

for var in range(-60, 61):
    y_pred.append(model.predict(np.array([[var / 10, var
/ 10]])))[0][0])
    x_pred.append(var / 10)
    k_pred.append(var / 10)

plot_data = [
    go.Scatter(
        x=x_pred,
        y=y_pred,
        name='Предсказание',
    ),
    go.Scatter(
        x=xs,
        y=ys,
        name='Реальные данные',
    )
]
```

```

plot_layout = go.Layout(
    xaxis={"type": "linear"},
    title='График по реальным параметрам и график
предсказаний (сильно приблизьте, чтобы увидеть ошибку)')

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
---↑in_out↓---
1/1 [=====] - 0s 977us/step
- loss: 5.5733
5.573311805725098
Числа для предсказания: k= 2.0  x= 3.0
Предсказание:  [[14.659372]]
Точность для predict_value больше 10%, что удовлетворительно.
#часть вывода представлена на рисунках ниже (рис. 81 и рис. 82)
---↑out---

```

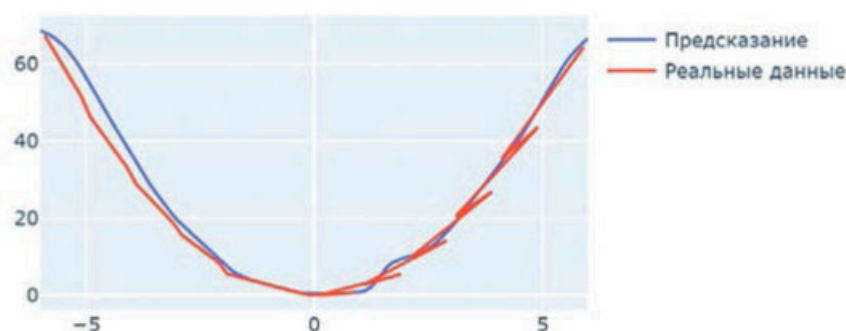


Рис. 81. Сравнение реальных данных с предсказанными для $Y = (K + X) \times X$

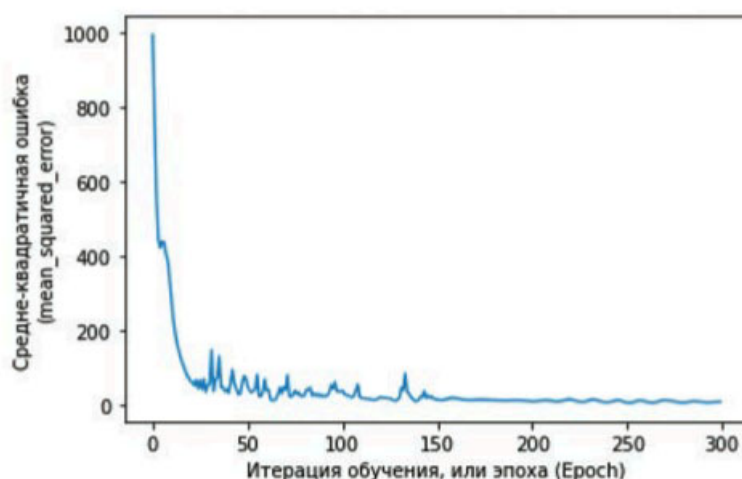


Рис. 82. Метрика «ошибка обучения» для $Y = (K + X) \times X$

Настройки оптимизатора

Обратите внимание, что в 3 и 4 примерах была произведена настройка оптимизатора при помощи представленного ниже кода:

```
---in↓---
sgd = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9,
                             nesterov=True) #сконфигурируем
оптимизатор
#для лучшей оптимизации обучения нейросети для решения
функции sin(x)

model.compile(optimizer=sgd, loss='mean_squared_error')
---↑in_out↓---
---↑out---
```

Здесь указывается *три параметра*:

- `learning_rate` (с англ. — «скорость обучения»). Влияет на скорость обучения наибольшим образом. По умолчанию `learning_rate=0.01`.

На каждой эпохе обучения происходит изменение весов с некоторым шагом, а именно с шагом, равным `learning_rate`. И веса подбираются таким образом (меняясь в большую и меньшую сторону), чтобы ошибка была минимальной. Но чем меньше шаг, тем больше потребуется эпох обучения.

И если шаг слишком маленький, то мы можем попасть в **локальный минимум ошибки**⁵¹, — это ситуация, когда размера шага недостаточно, чтобы при обучении на основе ошибки веса изменились значительно, и тем самым были перепробованы действительно все веса, а не только какая-либо их узкая часть.

Слишком маленькая скорость обучения заставляет алгоритм сходиться очень долго и «застревать» в локальных минимумах, слишком большая — «пролетать» узкие глобальные минимумы или вовсе расходиться.

Но на самом деле для эффективной борьбы с локальными минимумами существует и другой параметр — момент;

⁵¹ Так как нейронные сети обучаются при помощи метода обратного распространения ошибки. См. информацию, доступную в источнике [51].

- `momentum` (момент). Ускоряет градиентный спуск в соответствующем направлении и гасит колебания. По умолчанию `momentum=0`.

Градиентный спуск — метод нахождения локального минимума или максимума функции с помощью движения вдоль градиента. Говоря более простым языком, это некий алгоритм оптимизации, который позволяет быстрее обучать нейронную сеть, не перебирая все возможные варианты весов, а «двигаясь по тем вариантам смены весов», которые дают наибольшую скорость снижения ошибки;

- `nesterov`. Определяет, применять ли ускоренные градиенты Нестерова. Это одно из доступных улучшений метода SGD (градиентного спуска), увеличивающее скорость сходимости. По умолчанию `nesterov=false` (т.е. отключено).

Изучить влияние настроек оптимизатора предлагается самостоятельно или можно использовать значения по умолчанию. Но стоит отметить, что настройки оптимизатора значительно влияют на скорость обучения (количество эпох) и итоговую точность нейросети (ошибку).

В завершение занятия удалим все созданные файлы, чтобы не засорять память компьютера:

```
---in↓---
import os

files = ['model_example1.png', 'model_example2.png',
'model_example3.png', 'model_example4.png']
for f in files:
    if os.path.isfile(f): # если файл существует
        os.remove(f)
---↑in_out↓---
---↑out---
```

Задания для проверки

1. Что дает использование высокоуровневой библиотеки Keras, являющейся надстройкой над TensorFlow?

2. Что такое функции потерь и алгоритм оптимизатора? Какие функции активации доступны в библиотеке Keras? Перечислите 2–3 любых функции и приведите примеры использования.

3. Чем определяется количество входов/выходов у нейросети?
 4. Какова последовательность определения, обучения, использования нейросетевой модели TensorFlow с надстройкой Keras? Опишите кратко алгоритм работы программных кодов (на основе материалов занятия).

5. Верно ли утверждение: если набор данных слишком вариативен или количество обучающих примеров слишком велико, то сеть станет обладать плохой интерполирующей способностью и будет «запоминать» ответы?

6. Реализуйте при помощи нейронной сети (на базе TensorFlow Keras) решение одного из десяти математических уравнений, представленных в таблице ниже (табл. 36). Необходимо по примерам, приведенным в занятии, создать, обучить и протестировать нейросеть для решения данного математического уравнения.

Таблица 36

Варианты индивидуальных заданий

№ варианта	Уравнение	№ варианта	Уравнение
1.1	$Y = kx / 2 + 2$	2.1	$Y = (k + b)x$
1.2	$Y = 2kx + b$	2.2	$Y = kx + 3b$
1.3	$Y = kx + b / 2$	2.3	$Y = 2\sin x$
1.4	$Y = 3\cos x$	2.4	$Y = x / 2 + 2b$
1.5	$Y = x^2 + 1$	2.5	$Y = 2\sin x + b$

Выбор архитектуры нейронной сети (количество слоев, методы оптимизации и функции активации) выполняется самостоятельно методом подбора. Цель работы — создать нейронную сеть, количество эпох обучения которой не превышает 500, количество нейронов на каждом слое — не более 50, количество слоев — не более 2. Количество обучающих примеров не должно быть меньше 20, точность нейронной сети — не хуже $\pm 10\%$.

TENSORFLOW KERAS. КЛАССИФИКАЦИЯ ИЗОБРАЖЕНИЙ⁵²

Цели выполнения работы

Написание программы для распознавания цифр на базе TensorFlow Keras. Создание нейросети и ее обучение по базе данных MNIST. Тестирование MNIST выборкой и пользовательскими картинками.

Порядок выполнения работы

Создание нейросети и использование датасета MNIST

На данном занятии будет создана нейронная сеть, которая классифицирует изображения рукописных цифр. При написании программы будет использоваться надстройка «tf.keras», которая является высокоуровневой API (англ. *Application Programming Interface* — «программный интерфейс приложения») для построения и обучения моделей в TensorFlow.

Подключим все необходимые библиотеки (модули):

```
---in↓---
# Подключаем TensorFlow и tf.keras
import tensorflow as tf
from matplotlib import pyplot as plt
from tensorflow import keras
from tensorflow.keras.utils import plot_model

# Подключаем вспомогательные библиотеки numpy
и matplotlib
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

print(tf.__version__)
```

⁵² Разработано на основе русскоязычной книги — [46] в списке источников и литературы.

```

---↑in_out↓---
2.3.0
---↑out---
```

На данном занятии будет использоваться датасет MNIST который содержит 60 тысяч обучающих монохромных изображений в 10 категориях (а также дополнительно 10 тысяч — для тестирования). На каждом изображении содержится по одной цифре в низком разрешении (28×28 пикселей, рис. 83).

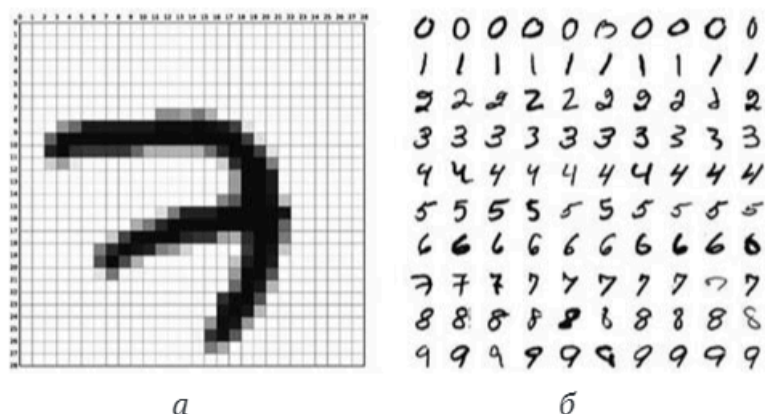


Рис. 83. Датасет *MNIST*:

a — образец, принадлежащий цифре «7»;

б — 100 примеров из обучающего набора *MNIST*

MNIST часто используют как «Hello, world» программ машинного обучения для компьютерного зрения. Он относительно мал и содержит изображения рукописных цифр (0, 1, 2, и т.д.), применяется для проверки корректности работы алгоритма нейросети.

Мы будем использовать 60 000 изображений для обучения нейросети и 10 000 изображений, чтобы проверить, насколько правильно сеть обучилась их классифицировать. Получить доступ к MNIST можно прямо из TensorFlow — импортируем его:

```

---in↓---
from tensorflow.keras.datasets import mnist

fashion_mnist = mnist
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
```

```
# Из диапазона 0-255 преобразуем к диапазону 0-1, чтобы
нейросеть
# работала не со значениями [0;255], а [0;1].
train_images = train_images / 255.0
test_images = test_images / 255.0
---↑in_out↓---
---↑out---
```

Загрузка датасета возвращает четыре массива NumPy:

- массивы «train_images» и «train_labels» являются обучающей (или, как часто говорят за рубежом, тренировочной) выборкой, т.е. данными, на которых модель будет обучаться;
- модель тестируется на проверочном сете (наборе данных), а именно массивах «test_images» и «test_labels».

Изображения являются 28×28 массивами NumPy (т.е. каждое из 60,000 изображений имеет размер 28×28 пикселей), где значение пикселей варьируется от 0 до 255 (под значением пикселей имеется в виду цветность). Метки «labels» — это массив целых чисел от 0 до 9. Они соответствуют классам цифр, изображенных на картинках, что представлено в таблице ниже (табл. 37).

Каждому изображению соответствует единственная метка. Так как названия классов не включены в датасет, сохраним их в переменную `class_names` для дальнейшего использования при классификации изображений:

```
---in↓---
class_names = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
---↑in_out↓---
---↑out---
```

Таблица 37

Метки классов

Метки «labels»	Класс
0	0
1	1
...	...
9	9

Чтобы убедиться, что перед обучением модели данные находятся в правильном формате и мы готовы построить и обучить нейросеть, выведем на экран первые 25 изображений из обучающей выборки и отобразим под ними наименования их классов. Для более простого понимания кода после вывода 25 изображений также выведем всего лишь одно, имеющее индекс 0:

```
---in↓---
# Просмотреть первые 25 изображений из датасета
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
print("первые 25 изображений из датасета:")
plt.show()

# Просмотреть изображение с индексом 0
plt.imshow(train_images[0])
print("изображение с индексом 0:")
plt.show()
---↑in_out↓---
первые 25 изображений из датасета:
#часть вывода представлена на рисунке ниже (рис. 84)
изображение с индексом 0:
#часть вывода представлена на рисунке ниже (рис. 85)
---↑out---
```

Теперь построим модель нейронной сети, т.е. зададим структуру нашей нейросети:

```
---in↓---
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(10, activation="softmax"),])
---↑in_out↓---
---↑out---
```

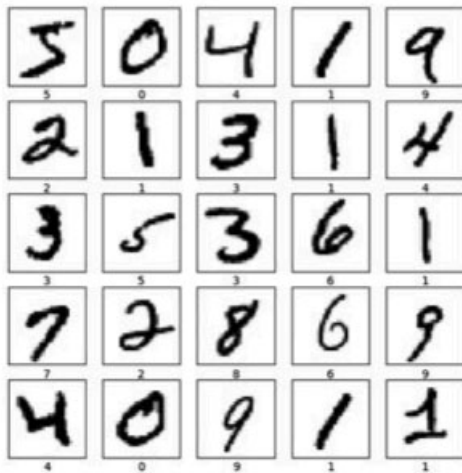



Рис. 84. Первые 25 изображений из датасета `train_images`

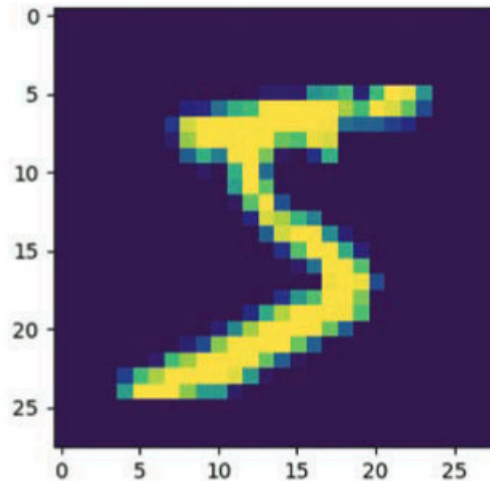


Рис. 85. Изображение `train_images[0]`

Первый слой этой сети — `tf.keras.layers.Flatten` — преобразует формат изображения из двухмерного массива (28×28 пикселей) в одномерный (размерностью $28 \times 28 = 784$ пикселя). Данный слой извлекает строки пикселей из изображения и выстраивает их в один ряд. Этот слой не имеет параметров для обучения, он только переформатирует данные.

После разложения пикселей нейросеть содержит два слоя `tf.keras.layers.Dense`. Это полносвязные нейронные слои.

Первый слой состоит из 128 нейронов с функцией активации `relu`. Данный слой осуществляет основную «логику» по классификации чисел. Нейроны также называются узлами.

Второй слой — 10-узловой с функцией активации `softmax`. Он возвращает массив из 10 вероятностных оценок, дающих в сумме 1. Каждый нейрон второго слоя на выход выводит оценку, указывающую вероятность принадлежности изображения к одному из 10 классов.

Прежде чем модель будет готова для обучения, нам нужно указать еще несколько **параметров**:

- *функция потерь* (`loss function`). Измеряет точность модели во время обучения. Необходимо минимизировать эту функцию, чтобы «направить» модель в верном направлении;
- *оптимизатор* (`optimizer`). Показывает, каким образом обновляется модель на основе входных данных и функции потерь;

- *метрики (metrics)*. Используются для мониторинга обучения и тестирования модели. Наш пример применяет метрику «ассурасу», равную доле правильно классифицированных изображений.

Описанные выше параметры указываются на шаге компиляции модели:

```
---in↓---
model.compile(
    optimizer="adam", loss="sparse_categorical_
crossentropy", metrics=["accuracy"]
)
---↑in_out↓---
---↑out---
```

Теперь обучим нашу нейросеть. Обучение модели нейронной сети требует выполнения следующих **шагов**:

1. Подать обучающую выборку в модель. В этом примере обучающие данные — это массивы `train_images` и `train_labels`.

2. Модель обучится ассоциировать изображения с правильными классами.

3. Сделать прогнозы для проверочных данных (массив `test_images`). Произойдет проверка, соответствуют ли предсказанные классы меткам из массива `test_labels`.

Запустим обучение при помощи вызова метода `model.fit()`. В процессе обучения модели отображаются метрики потери (`loss`) и точности (`accuracy`). Эта модель достигает на тренировочных данных точности, равной приблизительно 0.88 (92–98 %):

```
---in↓---
history = model.fit(train_images, train_labels, epochs=3)

print(history.history.keys())
plt.title('Изменение метрики обучения "ошибка"
в зависимости от итерации обучения\n')
plt.xlabel("Итерация обучения, или эпоха (Epoch)")
plt.ylabel("Среднеквадратичная ошибка\n(mean_squared_
error)")
plt.plot(history.history["loss"])
```

```

---↑in_out↓---
Epoch 1/3
1875/1875 [=====] - 1s 713us/
step - loss: 0.2576 - accuracy: 0.9262
Epoch 2/3
1875/1875 [=====] - 1s 689us/
step - loss: 0.1158 - accuracy: 0.9658
Epoch 3/3
1875/1875 [=====] - 1s 686us/
step - loss: 0.0798 - accuracy: 0.9760
dict_keys(['loss', 'accuracy'])
#часть вывода представлена на рисунке ниже (рис. 86)
---↑out---
```

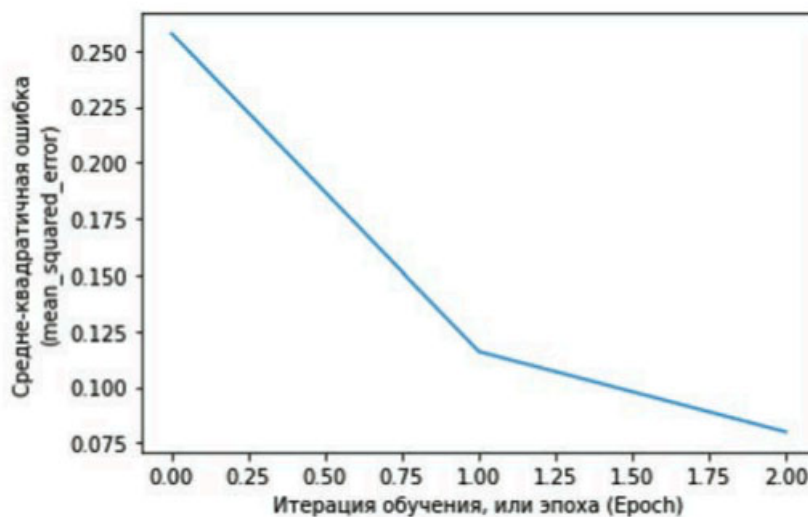


Рис. 86. Метрика «ошибка обучения» в задаче распознавания цифр

Далее сравним, какую точность модель покажет на проверочном датасете относительно тренировочного датасета:

```

---in↓---
test_loss, test_acc = model.evaluate(test_images, test_
labels, verbose=2)
print("Точность на проверочных данных:", test_acc)
---↑in_out↓---
313/313 - 0s - loss: 0.0870 - accuracy: 0.9731
Точность на проверочных данных: 0.9731000065803528
---↑out---
```


Полученная на проверочном сете точность (97.3 %) оказалась немного ниже, чем на тренировочном (97.6 %, см. ассурасу в выводе обучения). Этот разрыв между точностью на обучении и тестировании является примером переобучения (англ. *overfitting*). Переобучение возникает, когда модель машинного обучения показывает на новых данных худший результат, чем на тех, на которых она обучалась.

Теперь, когда модель обучена, мы можем использовать ее, чтобы сделать предсказания по поводу нескольких изображений. Полученная модель предскажет класс цифры для каждого изображения в проверочном датасете. Давайте посмотрим на первое предсказание:

```
---in↓---
predictions = model.predict(test_images)
print(predictions[0])
---↑in_out↓---
[3.8970079e-08 2.0492825e-08 1.6377642e-05 5.9514809e-05
 2.0149856e-11
 1.6520221e-07 9.5718511e-14 9.9992263e-01 4.2380867e-07
 7.3233161e-07]
---↑out---
```

Прогноз представляет из себя массив из 10 чисел. Они описывают «уверенность» модели в том, насколько изображение соответствует каждому из 10 разных видов цифр (или к какому классу принадлежит изображение). Мы можем посмотреть, какой метке соответствует максимальное значение:

```
---in↓---
print(np.argmax(predictions[0])) # вывод наиболее
точного предсказания
print(test_labels[0]) # вывод реальной метки

if np.argmax(predictions[0]) == test_labels[0]:
    print("Угадано верно.")
else:
    print("Нейросеть не угадала.")
---↑in_out↓---
```

```

7
7
Угадано верно.
---↑out---
```

Модель полагает, что на первой картинке изображена цифра 7, или `class_names [7]`. Проверка показывает, что классификация верна.

Построим и выведем картинку, показывающую предсказание:

```

---in↓---
def plot_image(i, predictions_array, true_label, img):
    """Эта функция строит и выводит картинку, хранящуюся
    как список пикселей

    :param i: какое предсказание вывести (индекс)
    :param predictions_array: набор всех предсказаний
    :param true_label: имена классов,
    :param img: "картинки" в виде массива пикселей
    """
    predictions_array, true_label, img = predictions_
array[i], true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = "blue"
    else:
        color = "red"

    plt.xlabel(
        "{} {:.20f}% ({}).format(
            class_names[predicted_label],
            100 * np.max(predictions_array),
            class_names[true_label],
        ),
```



```
        color=color,
    )

def plot_value_array(i, predictions_array, true_label):
    """Эта функция строит и выводит столбиковую диаграмму
    для предсказания

    :param i: какое предсказание вывести (индекс)
    :param predictions_array: набор всех предсказаний
    :param true_label: имена классов
    """
    predictions_array, true_label = predictions_array[i],
    true_label[i]
    # plt.grid(False)
    plt.xticks([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    # plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array,
    color="#777777")
    plt.ylim([0, 1])
    plt.xlim([0, 9])
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color("red")
    thisplot[true_label].set_color("blue")
---↑in_out↓---
---↑out---
```

Воспользуемся написанными функциями `plot_image` и `plot_value_array`, чтобы проверить, как они работают:

```
---in↓---
i = 1
plt.figure(figsize=(6, 3))
plt.subplot(1, 2, 1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1, 2, 2)
plot_value_array(i, predictions, test_labels)
print("Предсказание для test_images[1]")
plt.show()
---↑in_out↓---
```

```

Предсказание для test_images[1]
#часть вывода представлена на рисунке ниже (рис. 87)
---↑out---

```

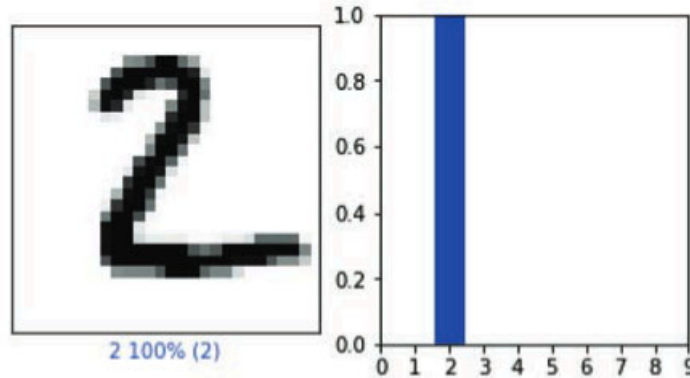


Рис. 87. Предсказание для test_images[1]

Посмотрим еще на несколько изображений с их прогнозами. Цвет верных предсказаний — синий, а неверных — красный. Число — это процент уверенности для предсказанной метки. Стоит отметить, что модель может ошибаться, даже если она очень уверена.

```

---in↓---
# Отображаем первые X тестовых изображений, их
# предсказанную и настоящую метки.
# Корректные предсказания окрашиваем в синий цвет,
# ошибочные — в красный.
num_rows = 5
num_cols = 3
num_images = num_rows * num_cols
plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)
    plot_image(i, predictions, test_labels, test_images)
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)
    plot_value_array(i, predictions, test_labels)
print("Предсказание для test_images[0] - test_images[14]")
plt.show()
---↑in_out↓---
Предсказание для test_images[0] - test_images[14]
#часть вывода представлена на рисунке ниже (рис. 88)
---↑out---

```

Теперь напишем обработчик для собственной картинки рукописной цифры. Входящая картинка может быть любого размера, главное, чтобы она была двухцветной (черно-белая, желательно с минимальной градацией серого (иначе нейронная сеть может не распознать цифру, поскольку ее обучение проводилось исключительно на бело-черных картинках)). Желательно, чтобы картинка была в формате png⁵³, так как далее будет рассмотрено, как создать свою собственную картинку для классификации при помощи фоторедактора gimp.

Необходимо наличие тестового файла «image.png», который будет открыт и подан в нейросетевую модель для предсказания класса. В нашем случае в данном файле изображена цифра 5.

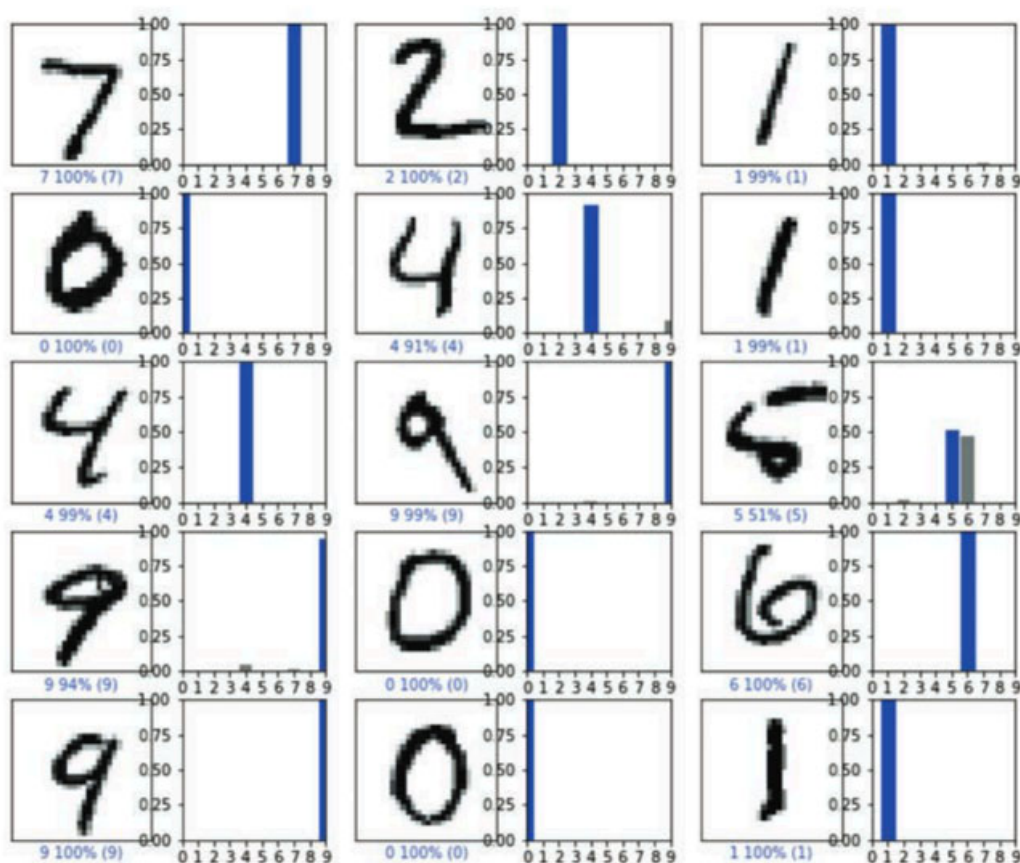


Рис. 88. Предсказание для test_images[0] – test_images[14]

⁵³ Image.open() кроме png поддерживает множество других форматов. Также был протестирован jpg. Стоит отметить, что уровень сжатия сильно влияет на шум в картинке, и она может перестать распознаваться (актуально для обоих форматов).

```

---in↓---
import os
from matplotlib import pyplot as plt
from PIL import Image, ImageFilter

def imageprepare(path):
    """Эта функция открывает черно-белую картинку,
    масштабирует ее до 28×28 (по большей стороне,
    если картинка не квадратная), нормализует пиксели от
    0-255 к 0-1, и возвращает картинку как
    двухмерный list с размером 28×28 (и значениями от 0 до 1)

    :param path: путь к картинке
    :return: картинка как двухмерный list[28][28]
    (и значениями от 0 до 1)
    """
    im = Image.open(path).convert("L") # "L" - один
    цветовой канал, так как оттенки серого
    width = float(im.size[0])
    height = float(im.size[1])
    newImage = Image.new("L", (28, 28), (255)) # создает
    белый холст 28×28

    if width > height: # проверяем какой размер больше
        # Если ширина больше высоты, то Width становится
        28 пикселей, а высоту нормализуем.
        nheight = int(
            round((28.0 / width * height), 0)
        ) # изменить размер высоты в соответствии
        с соотношением ширины (пропорцией)
        if nheight == 0: # редкий случай, но высота
        должна быть минимум 1 пиксель
            nheight = 1
            # изменить размер и резкость
            img = im.resize((28, nheight), Image.ANTIALIAS).
            filter(ImageFilter.SHARPEN)
            wtop = int(round(((28 - nheight) / 2), 0)) #
            вычислить положение по горизонтали
            newImage.paste(
                img, (0, wtop)

```



```
    ) # вставить изображение с измененным размером
на белый холст
    else:
        # все то же самое, но если высота больше, то
        обрезка по ней произойдет
        nwidth = int(round((28.0 / height * width), 0))
        if nwidth == 0: # rare case but minimum is
1 pixel
            nwidth = 1
            img = im.resize((nwidth, 28), Image.ANTIALIAS).
filter(ImageFilter.SHARPEN)
            wleft = int(round(((28 - nwidth) / 2), 0))
            newImage.paste(img, (wleft, 0))

            file_name = os.path.basename(path)
            new_file_name=os.path.splitext(file_name)[:1][0] +
"_cropped"+ os.path.splitext(file_name)[-1]
            print("Обрезанная картинка сохранена в файл:", new_
file_name)
            # сохранить отмасштабированную и обрезанную картинку
в файл
            newImage.save(new_file_name)

            tv = list(
                newImage.getdata()
            ) # получить значения пикселей 0-255, только 1 канал цвета

            # нормализовать пиксели до 0 и 1. 0 – чисто белый,
1 – чисто черный.
            tva = [(255 - x) * 1.0 / 255.0 for x in tv]

            # Теперь мы преобразуем массив 1d размером 784 в массив
2d размером 28×28, чтобы мы могли его визуализировать.
            newArr = [[0 for d in range(28)] for y in range(28)]
# создаем пустой список 28×28
            k = 0
            for i in range(28):
                for j in range(28):
                    newArr[i][j] = tva[k]
                    k = k + 1
            return newArr
```



```
img = imageprepare("./image.png") # file path here
print("Ширина", len(img), "Высота", len(img[0]))
# print(img) #выведет изображение в текстовом формате
---↑in_out↓---
```

Обрезанная картинка сохранена в файл: image_cropped.png
 Ширина 28 Высота 28
 ---↑out---

Представленный выше код откроет картинку `image.png`, отмасштабирует ее к формату 28×28 пикселей и запишет в переменную `img` как список значений от 0 до 1 (в качестве значения цвета пикселя). Размер списка тоже 28×28. На рис. 89 показано, как будет преобразована не квадратная картинка, линиями отмечено, что картинка была отмасштабирована по высоте с сохранением пропорций, а по ширине была уменьшена на тот же коэффициент масштаба, что и по высоте, а далее просто отцентрирована на холсте шириной 28 пикселей.

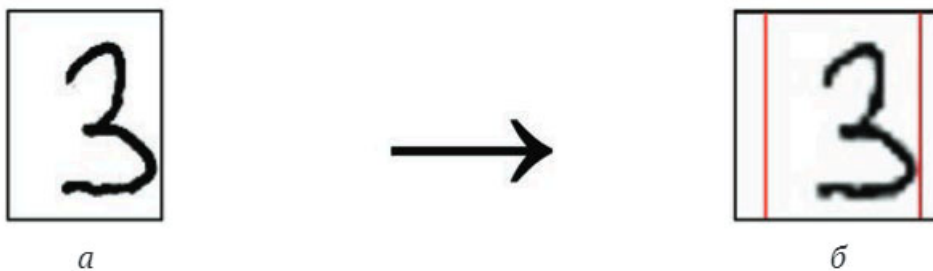


Рис. 89. Преобразование картинки:

а — оригинал (64×94 пикселей); *б* — после преобразования (28×28 пикселей)

Преобразованный вариант сохраняется с тем же именем и расширением, но в имя дописывается «`_cropped`». Например, из «`image.png`» получается «`image_cropped.png`». Обратим внимание, что в картинке `image.png` изображена цифра 5, а преобразование цифры 3 на рис. 89 приведено просто как пример. Про цифру 3 мы поговорим позже.

Модели `tf.keras` оптимизированы для предсказаний по пакетам данных (пакет — `batch` (англ.)), т.е. получения предсказаний для множества сетов входных данных сразу. Таким образом, даже если нужно отклассифицировать всего 1 картинку, то все равно необходимо добавить ее в список:

```
---in↓---
# Добавляем изображение в пакет данных, состоящий только
из одного элемента.
img = np.expand_dims(img, 0)
---↑in_out↓---
---↑out---
```

Предскажем для изображения вероятности принадлежности к каждому из 10 классов цифр. Метод `model.predict()` возвращает нам список списков — по одному для каждой картинке в пакете данных. В данном случае в пакете данных только одна картина, поэтому на выходе мы получим двухмерный массив, у которого на первом уровне содержится единственный массив из 10 предсказаний:

```
---in↓---
predictions_single = model.predict(img)
print(predictions_single)
---↑in_out↓---
[[7.2719617e-05 2.6502952e-04 5.7016878e-04 5.3933412e-02
 4.9496253e-07
 9.3102551e-01 2.7415188e-04 8.2742808e-06 1.1108978e-02
 2.7411713e-03]]
---↑out---
```

Таким образом для рукописной цифры, представленной в вашей картинке (`image.png`), нейронная сеть предскажет степень принадлежности от 0 до 1 к классу, к которому принадлежит эта цифра.

Чтобы получить наибольшую вероятность, можно воспользоваться методом `np.argmax()`, который выведет нам индекс наибольшего значения из массива:

```
---in↓---
print(np.argmax(predictions_single[0]))
---↑in_out↓---
5
---↑out---
```

Для большей наглядности выведем график распределения вероятности принадлежности изображения к классам чисел:

```

---in↓---
plt.subplot(1, 2, 1) # строка, сколько колонок, индекс
                        колонки в графике
plot_image(0, predictions_single, test_labels, img)
plt.subplot(1, 2, 2) # строка, сколько колонок, индекс
                        колонки в графике
plot_value_array(0, predictions_single, test_labels)
print("Предсказание для картинки image.png")
plt.show()
---↑in_out↓---
Предсказание для картинки image.png
#часть вывода представлена на рисунке ниже (рис. 90)
---↑out---

```

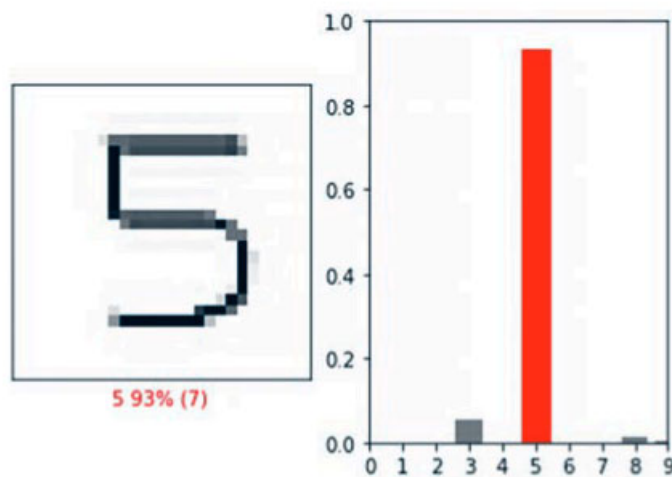


Рис. 90. Предсказание для image.png

Теперь выведем получившуюся модель нейросети:

```

---in↓---
plot_model(
    model,
    to_file="model.png",
    show_shapes=True,
    show_layer_names=True,
    rankdir="TB",
    expand_nested=False,
    dpi=96,
)

```

```
---↑in_out↓---
```

```
#вывод представлен на рисунке ниже (рис. 91)
```

```
---↑out---
```

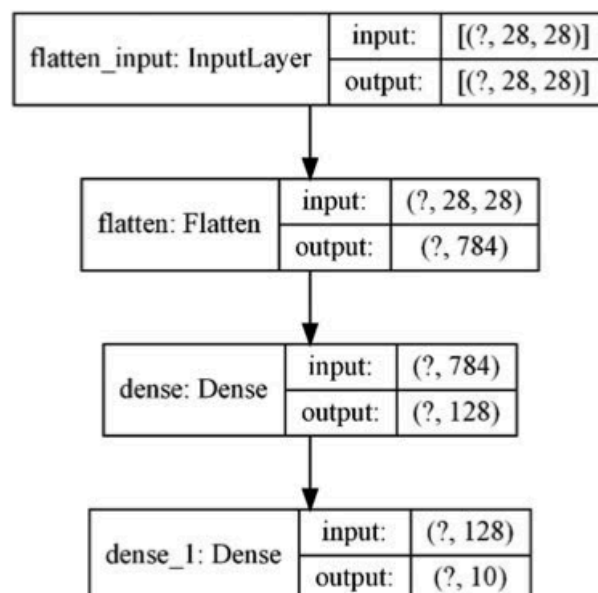


Рис. 91. Модель нейросети для классификации цифр

Модель нейросети для классификации рукописных цифр имеет 128 и 10 основных нейронов на первом и втором слоях соответственно (см. рис. 91). Эти нейроны и производят распознавание цифр. Нейроны на нулевом слое, помеченном как `flatten` (с англ. — «плоский»), — вспомогательные, необходимы для преобразования входного двухмерного массива в одномерный и не имеют ни весов, ни смещений для обучения.

Также мы имеем возможность вывести веса и смещения каждого из слоев. Так как по умолчанию `use_bias=True`⁵⁴ (использование нейронов смещения включено), то кроме весов синапсов (связей) нейронов для каждого слоя есть еще количество смещений, равное количеству выходных синапсов слоя.

⁵⁴ Этот параметр можно было указать для каждого слоя при создании структуры нейросети при помощи кода: `model = keras.Sequential()` (представлен ранее), написав в одном из слоев, например, `keras.layers.Dense(128, activation='relu', use_bias=True)`. Но по умолчанию `True`, поэтому и не было указано.


```

---in↓---
info=[]
for layer_index, layer in enumerate(model.layers):
    info.append(layer.get_weights()) # list of numpy arrays
    if len(info[layer_index])==0:
        print(f"Столько весов/смещений (weights) на
{layer_index} слое: {len(info[layer_index])}")
    if len(info[layer_index])==1:
        print(f"Столько весов (weights) для каждого узла
(нейрона) на {layer_index} слое: {len(info[layer_index]
[0])}")
    if len(info[layer_index])==2:
        print(f"Столько весов (weights) для каждого узла
(нейрона) на {layer_index} слое: {len(info[layer_index][0])}")
        print(f"Столько смещений (bias) для слоя целиком
на {layer_index} слое: {len(info[layer_index][1])}")

layer_number= 2
weight_or_bias=0 #0-веса, 1- смещения
my= info[layer_number][weight_or_bias]
print(f'На последнем слое для каждого из {len(my[0])}
нейронов {len(my)} весов:')
print(f'На последнем слое суммарно {len(my)*len(my[0])}
весов')

weight_or_bias=1 #0 – веса, 1 – смещения
my= info[layer_number][weight_or_bias]
print(f'Но при этом на последнем слое всего {len(my)}
смещений')
print(f'A "my[0]" - это уже значение первого смещения,
т.е. {my[0]}, а не еще один массив, как у весов')
---↑in_out↓---
Столько весов/смещений (weights) на 0 слое: 0
Столько весов (weights) для каждого узла (нейрона) на
1 слое: 784
Столько смещений (bias) для слоя целиком на 1 слое: 128
Столько весов (weights) для каждого узла (нейрона) на
2 слое: 128
Столько смещений (bias) для слоя целиком на 2 слое: 10
На последнем слое для каждого из 10 нейронов 128 весов:
На последнем слое суммарно 1280 весов

```


Но при этом на последнем слое всего 10 смещений
А "my[0]" - это уже значение первого смещения, т.е.
0.1025635302066803, а не еще один массив, как у весов
---↑out---

Конечно, разобраться с весами и смещениями уже не так просто, как на предыдущем занятии, так как может показаться, что все они идут вперемешку. Но если усвоить представленное ниже, то есть шанс понять вывод из примера выше:

- каждый нейрон — это узел графа («операция» над несколькими векторами), и каждый нейрон следующего слоя имеет перед собой количество весов, равное количеству нейронов на предыдущем слое, так как сеть в TensorFlow Keras всегда полносвязная (и в целом обычно нейронные сети всегда полносвязные — все нейроны связаны между собой);

- на слое столько смещений, сколько и нейронов на нем, потому что смещение применяется к выходу каждого нейрона (к результату вычисления функции активации);

- выход каждого нейрона представляет собой математический результат вычисления: сумма входных векторов (значений, умноженных на вес), над которой далее вычисляется функция активации, к которой добавляется значение смещения.

Но стоит отметить, что почти всегда нет никакой необходимости углубляться в конкретные значения весов и смещений. Нейронные сети служат в первую очередь как абстракция для решения сложных зависимостей, поэтому прибегать к ручному графовому анализу не имеет никакого смысла. Достаточно создать нейронную сеть, обучить, а далее использовать полученную модель как черный ящик теми же инструментами, что и создавали.

Подготовка и классификация пользовательской картинки

Теперь попробуем обработать картинку, полученную с камеры. Для этого установите себе на компьютер бесплатный графический редактор gimp с открытым исходным кодом (доступен к скачиванию с официального сайта — источник [52]).

Откроем картинку в *gimp* и откадрируем ее, т.е. обрежем так, чтобы не было ничего лишнего (рис. 92). Рекомендуется после кадрирования получить равное соотношение сторон (т.е. квадрат), так будет получена наибольшая точность при распознавании.



Рис. 92. Кадрирование картинки

Затем обесцветим картинку при помощи верхнего контекстного меню: «Цвет → Обесцвечивание → Обесцвечивание» (или «Цвет → Обесцвечивание», если старая версия *gimp*). Установите среднюю основу оттенков серого, как показано на рисунке ниже (рис. 93).

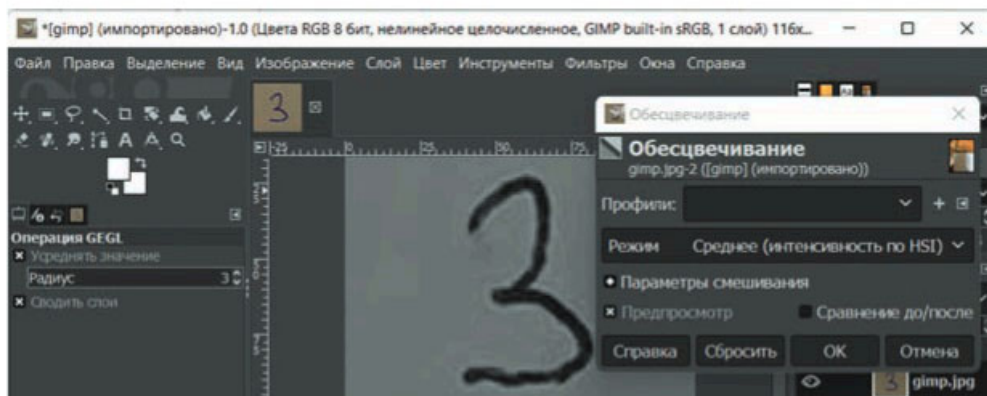


Рис. 93. Обесцвечивание картинки

Настроим яркость/контраст картинки («Цвет → Яркость-Контраст»), чтобы остались только белый и черный цвета (рис. 94).

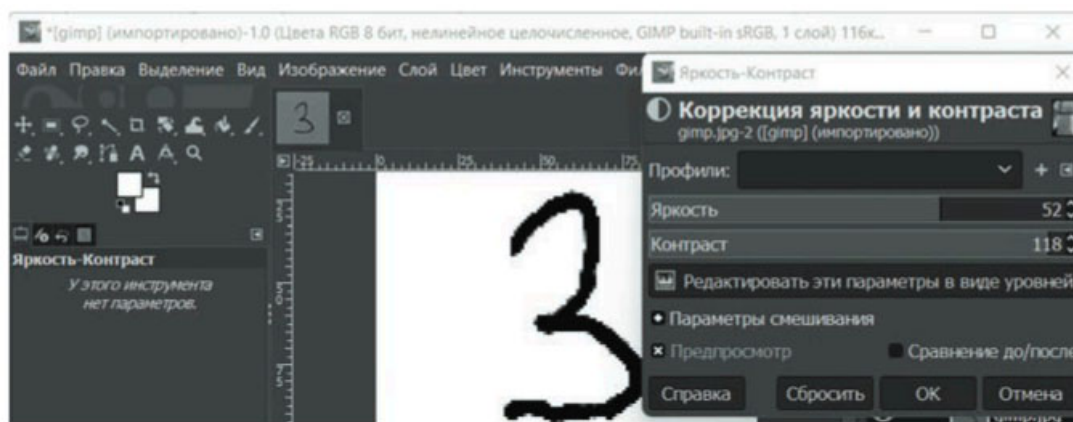


Рис. 94. Изменение яркости/контраста картинки

Теперь экспортируем картинку в формат .png («Файл → Экспортировать как...»), выбрав степень сжатия, равную нулю (рис. 95).

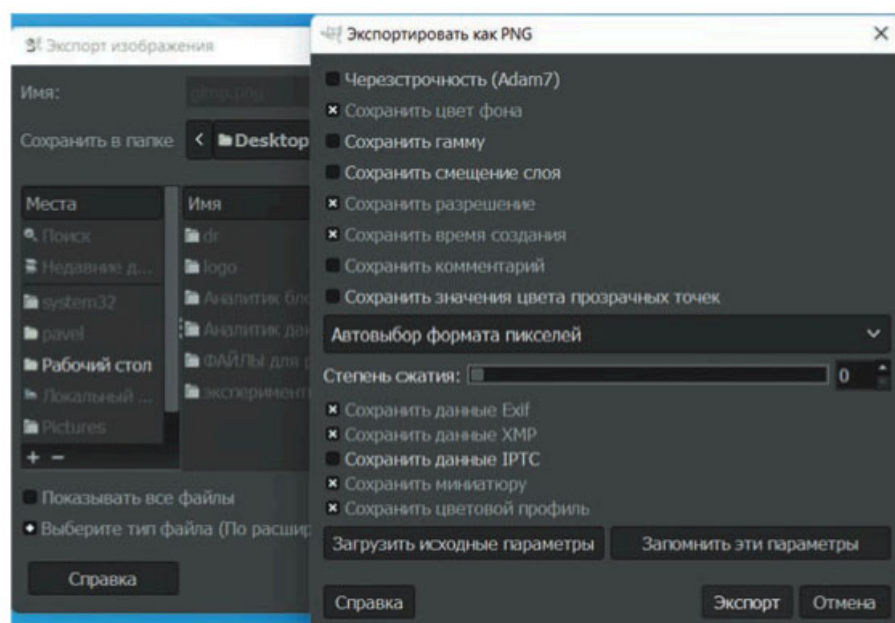


Рис. 95. Экспорт картинки gimp.png

Вставим путь к вашей картинке в код ниже. При выполнении данного кода будет осуществлена классификация написанной в изображении цифры:


```

---in↓---
img = imageprepare("./gimp.png") # file path here
# Добавляем изображение в пакет данных, состоящий только
из одного элемента.
img = np.expand_dims(img, 0)
predictions_single = model.predict(img)
print(predictions_single)

print(np.argmax(predictions_single[0]))

plt.subplot(1, 2, 1)
plot_image(0, predictions_single, test_labels, img)
plt.subplot(1, 2, 2)
plot_value_array(0, predictions_single, test_labels)
plt.show()
---↑in_out↓---
Обрезанная картинка сохранена в файл: gimp_cropped.png
[[5.6594126e-06 1.2847389e-02 1.7059442e-02 9.5942712e-01
 5.2441817e-05
 3.0845616e-03 1.5822765e-05 5.8365677e-04 6.3264631e-03
 5.9757876e-04]]
3
#часть вывода представлена на рисунке ниже (рис. 96)
---↑out---

```

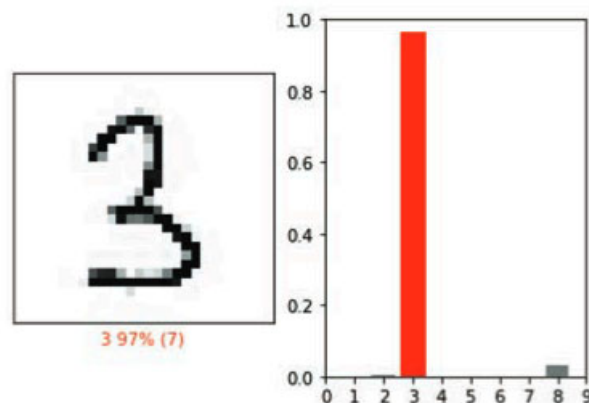


Рис. 96. Предсказание для картинки gimp.png

Представленный выше код такой же, как и рассмотренный ранее в общей части занятия. Он всего лишь продублирован, чтобы вы могли удобно подставить путь к своей картинке и выполнить данный код именно для нее.

Сделаем выводы

На данном занятии была разработана программа для классификации цифр по их изображениям. На основе представленного кода можно создать модель машинного обучения для распознавания любых одиночных объектов на изображениях.

Используя другие датасеты данных, возможно распознавать иные объекты, и даже не потребуется вносить правки в код, если формат обучающего датасета аналогичен рассмотренному на занятии. Например, есть в открытом доступе датасеты, представляющие из себя отдельные буквы (датасет EMNIST) или рисунки одежды (датасет Fashion-MNIST), и формат этих датасетов тоже 28×28 пикселей.

Можно получить распознавание многозначных цифр, если доработать код таким образом, чтобы он мог идентифицировать отдельные сегменты изображения с одиночными цифрами. Самый простой способ реализовать это — требовать разделения цифр от пользователя, который их пишет. То есть требовать писать цифры так, чтобы каждая цифра была на отдельном сегменте изображения, а сделать подобное можно, например, путем разметки листа бумаги квадратными ячейками, как показано на рисунке ниже (рис. 97).



Рис. 97. Сегментированный ввод

Далее код должен обработать каждую такую ячейку и из отдельных цифр выстроить многозначные цифры.

Для распознавания текста существуют специальные программные средства, называемые утилитами для OCR (от англ. *Optical Character Recognition* — «оптическое распознавание символов»). Большинство из них построено именно на моделях машинного обучения. Такие программы позволяют распознавать не сегментированные на отдельные символы изображения, используя специальные алгоритмы, про которые в данной книге упоминаться не будет.

Стоит отметить, что для TensorFlow есть надстройка TensorFlow Object Detection API, представляющая собой набор инструментов для создания моделей, которые способны распознавать несколько классов на одном изображении. Это тоже, к сожалению, не будет рассматриваться в данной книге, но главный принцип заключается в том, что при обучении используются проаннотированные изображения, т.е. изображения, где прямоугольниками выделены разные сегменты, на которых изображены конкретные классы. Например, на рисунке ниже продемонстрировано ручное аннотирование изображений при помощи утилиты labelImg (рис. 98)⁵⁵, на основе которого создаются карты меток, используемые при обучении с Object Detection API.

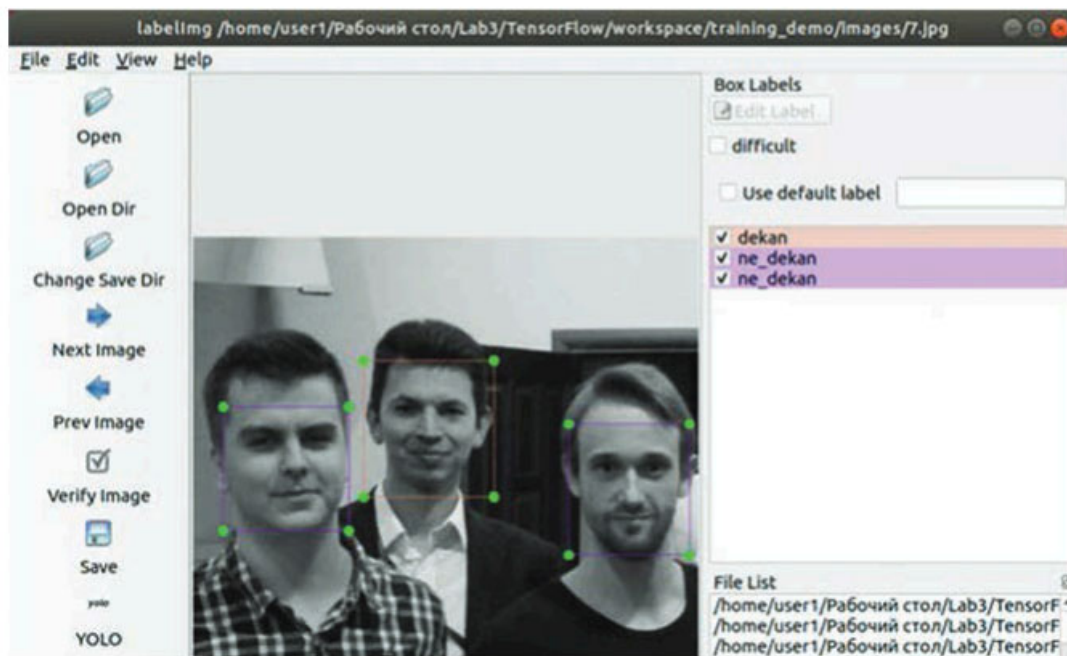


Рис. 98. Ручное аннотирование классов

В завершение занятия удалим все созданные файлы, чтобы не засорять память компьютера:

```
---in↓---
import os
```

⁵⁵ Подробнее представлено в книге, см. источник [46].

```
files = ['gimp_cropped.png', 'image_cropped.png', 'model.png']
for f in files:
    if os.path.isfile(f): # если файл существует
        os.remove(f)
---↑in_out↓---
---↑out---
```

? Задания для проверки

1. Что такое набор данных, или датасет? Для чего он может использоваться и как может задаваться (в виде каких типов данных)?
2. Для чего необходимо разбивать датасет на обучающие выборки и валидирующие (тестировочные)? Что дает тестирование?
3. Что такое классы объектов? Зачем им нужны имена? Влияет ли имя класса на обучение или это дополнительное «удобство» при использовании обученной нейросети?
4. Опишите назначения всех узлов элементов нейросети: входы, синапсы, нейроны, аксоны. Каким узлам соответствуют данные понятия: веса, функции активации?
5. Каким набором вызова функций библиотеки Keras можно определять слои и их параметры (функции активации нейронов на слоях, количество нейронов, тип слоя и прочие параметры)?
6. Подготовьте собственную картинку цифры, используя один из 40 представленных ниже вариантов (рис. 99) или на основе своей собственной фотографии. Протестируйте на ней классификацию при помощи нейросети.

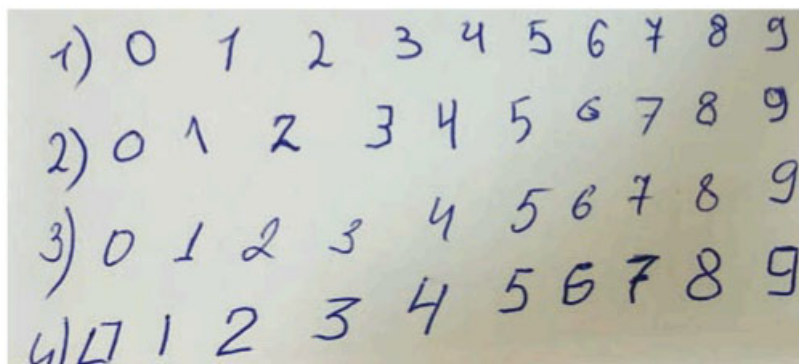


Рис. 99. Варианты индивидуальных заданий

БИБЛИОТЕКА SCIKIT-LEARN. ОБУЧЕНИЕ С УЧИТЕЛЕМ И БЕЗ УЧИТЕЛЯ⁵⁶

Цели выполнения работы

Получение представления о функциональности, доступных методах и объектах библиотеки `Scikit-learn`. Изучение основных принципов практической работы с ними для реализации математических алгоритмов.

Рассмотрение основных положений машинного обучения с учителем и без учителя: метод k -средних, линейная регрессия и деревья решений.

Порядок выполнения работы

Знакомство с библиотекой `Scikit-learn`

Библиотека `Scikit-learn` — самый распространенный выбор для решения задач классического машинного обучения. Она предоставляет широкий выбор алгоритмов обучения с учителем и без учителя.

Обучение с учителем предполагает наличие размеченного датасета, в котором известно значение целевого признака. В то время как обучение без учителя не предполагает наличия разметки в датасете — требуется научиться извлекать полезную информацию из произвольных данных.

Одно из основных преимуществ библиотеки состоит в том, что она работает на основе нескольких распространенных математических библиотек и легко интегрирует их друг с другом. Еще одним преимуществом является широкое сообщество и подробная документация.

⁵⁶ Разработано на основе источников [53], [54], [55], [56] и [57], среди которых есть официальная документация `Scikit-learn` и две русскоязычных статьи.

`Scikit-learn` широко применяется для промышленных систем, в которых применяются алгоритмы классического машинного обучения, для исследований, а также популярна у новичков, которые только делают первые шаги в области машинного обучения.

Для работы `Scikit-learn` используются представленные ниже популярные библиотеки. Почти все они уже были рассмотрены ранее:

- `NumPy` — математические операции и операции над массивами;
- `SciPy` — научно-технические вычисления;
- `Matplotlib` — визуализация данных;
- `SymPy` — символьная математика;
- `Pandas` — обработка, манипуляция и анализ данных.

`Scikit-learn` специализируется на алгоритмах машинного обучения с учителем и без учителя.

Обучение с учителем

Обучение с учителем (англ. *supervised learning*) — это один из способов машинного обучения, в ходе которого испытуемая система принудительно обучается с помощью примеров «стимул — реакция». К обучению с учителем относятся задачи классификации и регрессии.

Классификация — система группировки объектов исследования или наблюдения в соответствии с их общими признаками. При классификации происходит предсказание признака, множество допустимых значений которого ограничено.

Регрессия — выявление зависимости между случайными переменными и математическим выражением, отражающим связь между зависимой переменной и независимыми переменными x_i при условии, что это выражение будет иметь статистическую значимость. В отличие от чисто функциональной зависимости $y = f(x_i)$, когда каждому значению независимой переменной x_i соответствует одно определенное значение величи-

ны y , при регрессионной связи одному и тому же значению x_i могут соответствовать в зависимости от случая различные значения величины y .

Стоит отметить, что существует подвид (т.е. частный случай) обучения с учителем, называемый «обучение с подкреплением». При данном типе обучения испытуемая система (называемая агентом) обучается, взаимодействуя с некоторой средой, которая дает модели отклик на принятые решения. Отклик — это обратная связь. При этом агент не имеет никакой информации о среде и способен обучаться только посредством получения обратной связи от воздействия на нее.

Обучение без учителя

Обучение без учителя (самообучение, спонтанное обучение, англ. *unsupervised learning*) — один из способов машинного обучения, при котором испытуемая система спонтанно обучается выполнять поставленную задачу без вмешательства со стороны экспериментатора.

Как правило, оно пригодно только для задач, в которых известны описания множества объектов (т.е. существуют обучающие выборки) и требуется обнаружить внутренние взаимосвязи или закономерности, существующие между объектами.

Примерами обучения без учителя являются:

- *кластеризация* — задача группировки множества объектов на подмножества (кластеры) таким образом, чтобы объекты из одного кластера были более похожи друг на друга, чем на объекты из других кластеров, по какому-либо критерию;

- *снижение размерности* — представление данных в пространстве меньшей размерности с минимальными потерями полезной информации. Обычно в его основе лежит метод главных компонент;

- *выявление аномалий* — опознавание во время интеллектуального анализа данных редких данных, событий или наблюдений, которые вызывают подозрения ввиду существенного отличия от большей части данных.

Функциональность Scikit-learn

Библиотека Scikit-learn реализует следующие **основные методы**:

- *линейные* — модели, задача которых построить разделяющую или аппроксимирующую гиперплоскость (для классификации и регрессии соответственно);

- *метрические* — модели, которые вычисляют расстояние по одной из метрик между объектами выборки и принимают решения в зависимости от этого расстояния (например, метод *K*-ближайших соседей);

- *деревья решений* — обучение моделей, базирующихся на множестве условий, оптимально выбранных для решения задачи;

- *ансамблевые методы* — методы, основанные на деревьях решений, которые комбинируют мощь множества деревьев и таким образом повышают их качество работы, а также позволяют производить отбор признаков (*бустинг*, *бэггинг*, *случайный лес*, *мажоритарное голосование*);

- *нейронные сети* — комплексный нелинейный метод для задач регрессии и классификации;

- *метод опорных векторов* (англ. *Support Vector Machine, SVM*) — нелинейный метод, который обучается определять границы принятия решений;

- *наивный Байес* — прямое вероятностное моделирование для задач классификации;

- *метод главных компонент* (англ. *Principal Component Analysis, PCA*) — линейный метод понижения размерности и отбора признаков;

- *стохастическое вложение соседей с t -распределением* (англ. *t-distributed Stochastic Neighbor Embedding, t-SNE*) — нелинейный метод понижения размерности;

- *K-средних* — самый распространенный метод для кластеризации, требующий на вход число кластеров, по которым должны быть распределены данные;

- *кросс-валидация* — метод, при котором для обучения используется весь датасет (в отличие от разбиения на выборки *train/test*), однако обучение происходит многократно, и в качестве ва-

лидационной выборки на каждом шаге выступают разные части датасета. Итоговый результат является усреднением полученных результатов;

- *поиск по сетке* (англ. *Grid Search*) — метод для нахождения оптимальных гиперпараметров⁵⁷ модели путем построения сетки из значений гиперпараметров и последовательного обучения моделей со всеми возможными комбинациями гиперпараметров из сетки.

Выше представлен лишь базовый список. Помимо перечисленного, Scikit-learn содержит функции для расчета значений метрик, выбора моделей, предварительной обработки данных (препроцессинга) и др. Полный список доступен в официальной документации [53]. Также на официальном сайте [58] есть большое количество примеров, которые можно скачать единым архивом, открыть в JupyterLab и изучить⁵⁸.

Давайте рассмотрим несколько базовых примеров работы с данной библиотекой.

Кластерный анализ (общий обзор, K-средних, линейная регрессия и деревья решений)

Кластерный анализ, или **кластеризация** — это задача группировки набора объектов таким образом, чтобы объекты в одной группе (называемой кластером) были более похожи (в некотором

⁵⁷ **Гиперпараметр** — это параметр, который не настраивается во время обучения модели. Пример гиперпараметра — шаг градиентного спуска, он задается перед обучением. Пример параметров — веса градиентного спуска, они изменяются и настраиваются во время обучения.

⁵⁸ Все примеры при помощи кнопки `launch binder` можно запустить в облачном Jupyter-блокноте без каких-либо дополнительных манипуляций с окружением — прямо с сайта-документации. Стоит отметить, что наличие готовых к запуску примеров — это редкость. Возможно, из описанных в книге библиотек Scikit-learn имеет наиболее доступную к пониманию документацию. Также доступен блокнот для распознавания рукописных цифр посредством этой библиотеки — аналог уже рассмотренного решения, в котором использовался TensorFlow Keras.

смысле) друг на друга, чем на объекты в других группах (кластерах). Это основная задача исследовательского анализа данных и общий метод статистического анализа данных, используемый во многих областях, включая распознавание образов, анализ изображений, поиск информации, биоинформатику, сжатие данных, компьютерную графику и машинное обучение.

Ниже показан результат кластерного анализа по признаку геометрического положения квадратов (рис. 100). Принадлежность к одному из трех кластеров отмечена цветом.

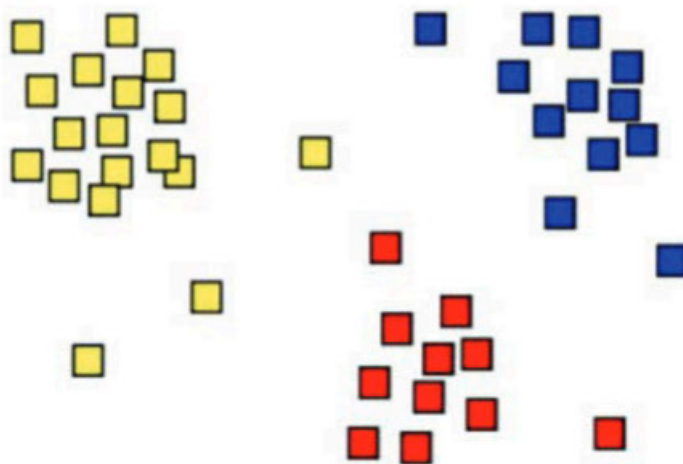


Рис. 100. Кластеризация на три группы

Кластеризация немаркированных данных, т.е. когда заранее неизвестно, к какому классу принадлежат данные, может выполняться с помощью модуля `sklearn.cluster`.

Обзор доступных методов кластеризации⁵⁹

В таблице ниже представлены основные доступные методы кластеризации с их краткими характеристиками и описанием (табл. 38). На рисунке после данной таблицы приведено графическое сравнение результата работы данных методов (рис. 101). На рисунке названия методов даны на английском языке, так как столбцы рисунка соответствуют строкам таблицы.

⁵⁹ Полный обзор с примерами доступен на английском языке в источнике [59].

Таблица 38

Доступные методы кластеризации

Название метода	Параметры	Масштабируемость	Вариант использования	Геометрия
K -средние	Количество кластеров	Очень большое количество примеров, среднее количество кластеров (при пакетном обучении MiniBatch)	Универсальный, плоская геометрия, не слишком много кластеров, индуктивный метод ¹	Расстояния между точками
Метод распространения близости	Демпфирование, схожесть	Не масштабируется с помощью количества примеров	Множество кластеров, неравномерные размеры кластеров, неплоская геометрия, индуктивный метод	Длина графа (например, до ближайшего соседа)
Сдвиг среднего значения	Пропускная способность	Не масштабируется с помощью количества примеров	Множество кластеров, неравномерные размеры кластера, неплоская геометрия, индуктивный метод	Расстояния между точками
Спектральная кластеризация	Количество кластеров	Среднее количество примеров и малое количество кластеров	Мало кластеров, одинаковые размеры кластеров, неплоская геометрия, трансдуктивный метод ²	Длина графа
Иерархическая кластеризация	Количество кластеров или порог расстояния	Большое количество примеров и кластеров	Много кластеров, возможны ограниченные зависимости между данными, трансдуктивный метод	Расстояния между точками
Агломеративная кластеризация	Количество кластеров / порог, тип связи	Большое количество примеров и кластеров	Множество кластеров, возможны ограниченные зависимости, неевклидовы расстояния ³ , трансдуктивный метод	Любое попарное расстояние

Окончание табл. 38

Название метода	Параметры	Масштабируемость	Вариант использования	Геометрия
DBSCAN	Близость соседа	Очень большое количество примеров и среднее количество кластеров	Неплоская геометрия, неодинаковые размеры кластеров, трансдуктивный метод	Расстояния между ближайшими точками
Оптический метод кластеризации	Минимальное членство в кластере	Большое количество примеров и кластеров	Неплоская геометрия, неравномерные размеры кластеров или их переменная плотность, трансдуктивный метод	Расстояния между точками
Гауссовские смеси	Много параметров	Не масштабируется	Плоская геометрия, подходит для оценки плотности, индуктивный метод	Расстояние Махаланобиса до центров
BIRCH	Фактор ветвления, порог, кластеризатор	Большое количество примеров и кластеров	Большой набор данных, удаление выбросов, сокращение данных, индуктивный метод	Евклидово расстояние между точками

Примечания:

¹ **Индуктивный метод** — обучение по прецедентам, или индуктивное обучение, основано на выявлении общих эмпирических закономерностей в данных.

² **Трансдуктивный метод** — обучение с частичным привлечением учителя, когда прогноз предполагается делать только для частной тестовой выборки на основе частной обучающей выборки (т.е. без систематичности предсказаний). На основе одной песни можно стенировать похожую.

³ Обычно используются евклидовы расстояния (евклидова метрика) между векторами, вычисляемые по теореме Пифагора. К неевклидовым относятся следующие расстояния: манхэттенское, Минковского, Махаланобис, косинусное сходство, жаккардовое и др. Использование конкретных типов расстояния зависит от характера данных — размер, тип и др. (например, если необходимо определить схожесть между двумя текстами, то используется косинусное сходство).

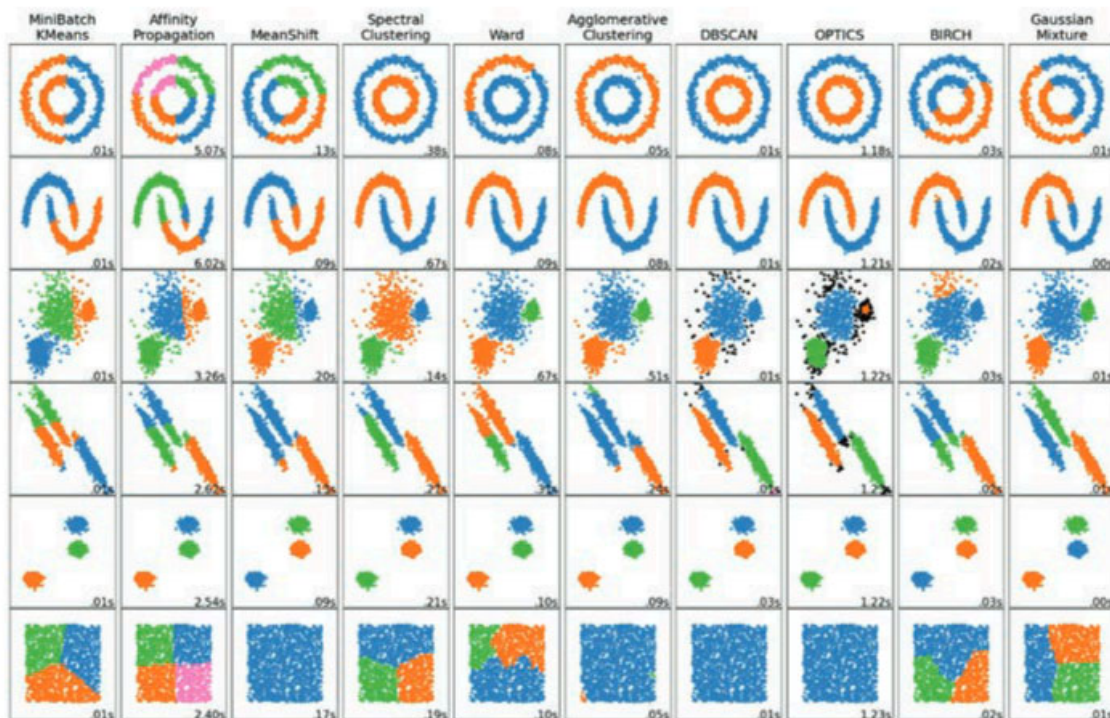


Рис. 101. Различные методы кластеризации

Давайте рассмотрим наиболее часто применяющийся алгоритм кластеризации — метод К-средних.

К-средних (K-means)

Алгоритм К-средних делит набор данных на непересекающиеся кластеры, каждый из которых описывается средним центром массы в кластере. Средние значения обычно называют «центроидами» кластера.

Действие алгоритма таково, что он стремится минимизировать суммарное квадратичное отклонение точек кластеров от центров этих кластеров (формула (33)). Это также называется инерцией.

$$\sum_{i=0}^n \min_{\mu \in C} (\|x_i - \mu_j\|^2), \quad (33)$$

где n — число кластеров; x — точка данных; $i = 1, 2, \dots$; μ_j — центры масс всех векторов из множества непересекающихся кластеров C .

По аналогии с методом главных компонент центры кластеров называются также главными точками, а сам метод называется методом главных точек и включается в общую теорию главных объектов, обеспечивающих наилучшую аппроксимацию данных.

Обратим внимание, что центр масс, как правило, не совпадает с точками данных, хотя они и живут в одном пространстве.

Инерцию можно определить как меру того, насколько кластеры внутренне связаны. Данный алгоритм обладает следующими **недостатками**:

- инерция предполагает, что кластеры выпуклые и изотропные, что не всегда так. Алгоритм плохо реагирует на удлинённые кластеры или связи неправильной формы;

- инерция — это ненормализованная метрика: нам всего лишь известно, что более низкие значения лучше, а ноль — оптимально. Но в очень многомерных пространствах евклидовы расстояния имеют тенденцию становиться раздутыми (это пример так называемого «проклятия размерности»). Выполнение алгоритма уменьшения размерности, такого как анализ главных компонент (РСА) перед кластеризацией К-средних, может облегчить эту проблему и ускорить вычисления. РСА также доступен в Scikit-learn.

Пример кластеризации К-средних

В этом примере мы рассмотрим, как реализовать кластеризацию методом К-средних с помощью Scikit-learn и pandas. Начнем с подключения необходимых модулей:

```
---in↓---
from sklearn.cluster import KMeans
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import style
%matplotlib inline
---↑in_out↓---
---↑out---
```

Создание исходных данных

Первый шаг — создать исходные данные вручную или же сгенерировать их. Мы создадим данные вручную, пусть это будет двумерный массив значений x и y . В этом наборе данных они имеют только две оси (но метод К-средних может использоваться с любым количеством осей):

```

---in↓---
data = pd.DataFrame([[1, 2], [5, 8], [1.5, 1.8], [8, 8],
[1, 0.6],
                        [9, 11]], columns=['x', 'y'])
print( data )

plt.scatter(data.x,data.y)
plt.show()
---↑in_out↓---
      x      y
0  1.0    2.0
1  5.0    8.0
2  1.5    1.8
3  8.0    8.0
4  1.0    0.6
5  9.0   11.0
#часть вывода представлена на рисунке ниже (рис. 102)
---↑out---
```

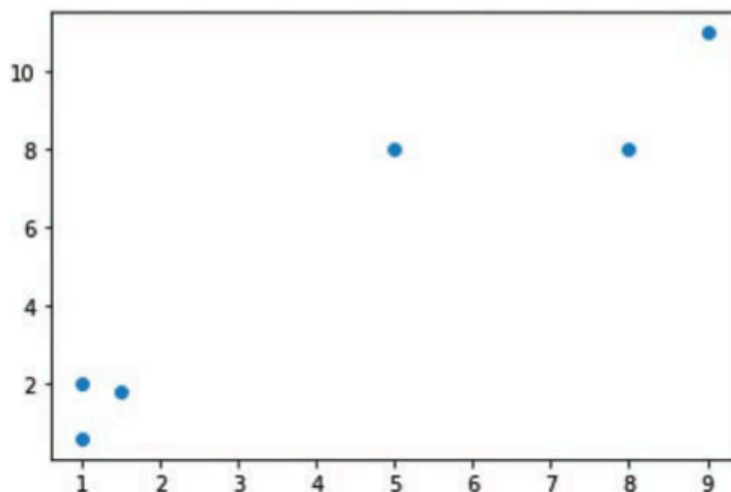


Рис. 102. Вручную созданные данные (точки)

Построение модели

Как и в случае с обучением с учителем, например, когда мы с вами работали с нейронными сетями в TensorFlow, вы сначала создаете модель, а затем вызываете метод запуска обучения `fit()`, используя свой источник данных. Теперь модель заполнена центроидами и метками. К ним можно получить доступ через свойства `cluster_centers_` и `labels_` соответственно.

```
---in↓---
kmeans = KMeans(n_clusters=2).fit(data.values)

centroids = kmeans.cluster_centers_
labels = kmeans.labels_

print(centroids) #выведет X Y центроидов
print(labels)
---↑in_out↓---
[[7.33333333 9.          ]
 [1.16666667 1.46666667]]
[1 0 1 0 1 0]
---↑out---
```

Визуализация кластеров

Напишем код, который визуализирует кластеры и крестами помечает их центры. Чтобы различать принадлежность точек к каждому кластеру, построим группы данных на отдельных осях (имеется в виду виртуальных), а также зададим им ключевые слова, цвета и метки:

```
---in↓---
data['labels'] = labels

group1 = data[data['labels']==1].plot( kind='scatter',
x='x', y='y', color='DarkGreen', label="Группа 1" )
group2 = data[data['labels']==0].plot( kind='scatter',
x='x', y='y', color='Brown', ax=group1, label="Группа 2" )
```



```
plt.scatter(centroids[:, 0], centroids[:, 1], marker = "x")
plt.show()
---↑in_out↓---
```

#вывод представлен на рисунке ниже (рис. 103)

```
---↑out---
```

Класс KMeans также имеет метод `predict()`, который можно использовать для прогнозирования (предсказания, англ. — *predict*) принадлежности точки к конкретному кластеру.

```
---in↓---
```

```
print ('Предсказание для точки (3,3):')
kmeans.predict([[3,3]])[0]
---↑in_out↓---
```

Предсказание для точки (3,3):

```
1
---↑out---
```

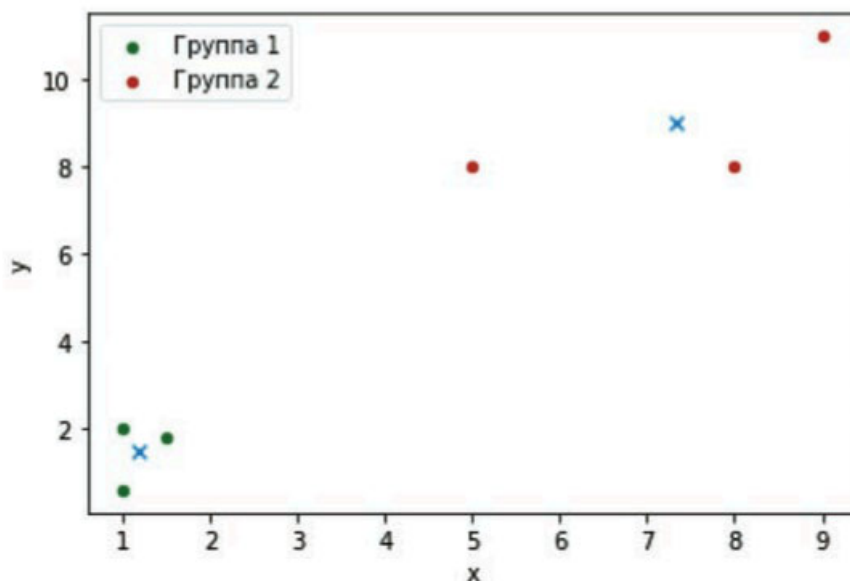


Рис. 103. Кластеры и их центры

Линейная регрессия

Регрессия прогнозирует непрерывные выходы значений, а классификация — дискретные. Например, прогнозирование стоимости дома в долларах является проблемой регрессии, тогда как прогно-

зирование злокачественной или доброкачественной опухоли — проблемой классификации.

Термин «линейность» в алгебре относится к линейной зависимости между двумя или более переменными. Если мы отобразим это соотношение в двухмерном пространстве (в данном случае между двумя переменными), мы получим прямую линию.

Давайте рассмотрим код, в котором мы определим линейную зависимость между количеством часов, которые студент учится, и процентом оценок, которые студент получает на экзамене. Если мы нанесем независимую переменную «часы» на ось x , а зависимую переменную «оценка» на ось y , линейная регрессия даст нам прямую линию, которая наилучшим образом соответствует точкам данных, как это показано на рисунке ниже (рис. 104).

Известно из школьного курса математики, что уравнение прямой имеет вид $y = kx + b$, где b — точка пересечения, а k — коэффициент наклона прямой линии. Данное уравнение представляет собой двухмерный пример линейной регрессии. И в этом случае алгоритм линейной регрессии дает нам оптимальное значение для точки пересечения и наклона (в двух измерениях).

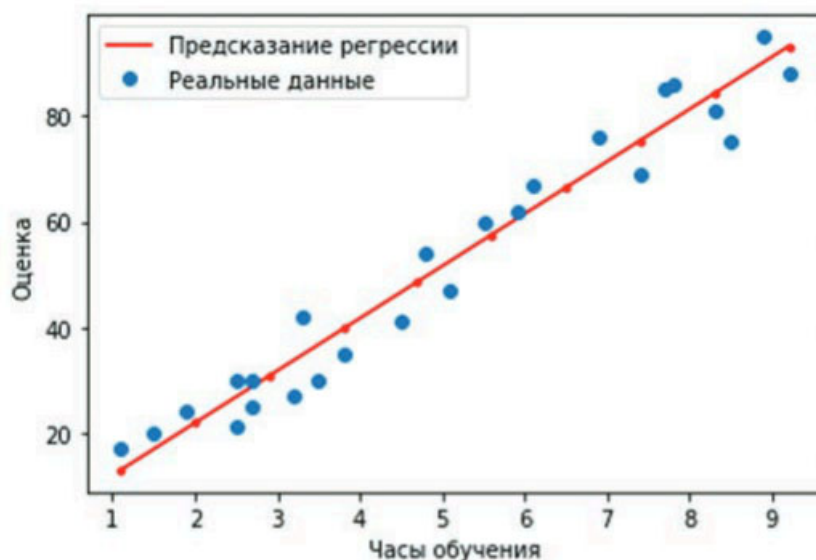


Рис. 104. Пример линейной регрессии

Переменные y и x остаются неизменными, поскольку они являются функциями данных и не могут быть изменены. Значения,

которые мы можем контролировать, — это точка пересечения и наклон. В зависимости от значений точки пересечения и наклона может быть несколько прямых линий.

По сути, алгоритм линейной регрессии производит вычисление уравнений с разными параметрами и возвращает те параметры наклона и сдвига, с которыми достигается наименьшая ошибка.

Эту же концепцию можно распространить на случаи, когда существует более двух переменных. Это называется **множественной линейной регрессией**. Например, необходимо спрогнозировать цену дома на основе его площади, количества спален, среднего дохода людей в этом районе, возраста дома и т.д. В этом случае зависимая переменная зависит от нескольких независимых переменных. Модель регрессии, включающая несколько переменных, выражается представленной ниже формулой (34):

$$y = b_0 + x_1 b_1 + x_2 b_2 + x_3 b_3 + \dots + x_n b_n, \quad (34)$$

где b_0 — единственный член (коэффициент); n — количество зависимых параметров x (и их коэффициентов b).

Это уравнение гиперплоскости. Модель линейной регрессии в двух измерениях — это прямая линия, в трех измерениях — это плоскость, а в более чем трех измерениях — гиперплоскость.

Вернемся к примеру с оценками и временем обучения студента, т.е. к двумерной линейной регрессии. В этой задаче мы прогнозируем процент оценок, которые, как ожидается, получит студент, на основе количества часов, в которые он обучался. Так как модули `pandas` и `matplotlib` мы уже подключаем при работе с методом k -средних, то можем приступить сразу к открытию и визуализации исходных данных.

Набор данных

Нам необходим датасет `student_scores.csv`, содержащий оценки студента (если он вам недоступен совместно с архивом файлов данной книги, то скачать данный датасет можно в источнике [60]). Когда мы убедились в наличии данного файла в текущем каталоге, откроем его с помощью `pandas`, выведем его раз-

мерность (п. 1 вывода) и первые три записи из него (п. 2). Также важно посмотреть на статистические детали данных, в частности, например, минимальные и максимальные значения, чтобы приблизительно понимать диапазоны значений (п. 3).

```
---in↓---
dataset = pd.read_csv('./student_scores.csv')
print ('1)shape', dataset.shape)
print ('2)head(3):\n',dataset.head(3))
print ('3)describe():\n',dataset.describe())
---↑in_out↓---
```

```
1)shape (25, 2)
2)head(3):
      Hours  Scores
0      2.5      21
1      5.1      47
2      3.2      27
3)describe():
      Hours      Scores
count  25.000000  25.000000
mean    5.012000  51.480000
std     2.525094  25.286887
min     1.100000  17.000000
25%     2.700000  30.000000
50%     4.800000  47.000000
75%     7.400000  75.000000
max     9.200000  95.000000
---↑out---
```

Давайте отобразим наши точки данных на двумерном графике для того, чтобы взглянуть на наш набор данных визуально и определить, сможем ли мы вручную найти какую-либо зависимость между данными:

```
---in↓---
dataset.plot(x='Hours', y='Scores', style='o') #style='o'
- чтобы рисовалась точка, а не линия от прошлой точки
  к текущей
plt.title('Часы vs Оценка')
```



```
plt.xlabel('Часы обучения')
plt.ylabel('Оценка')
plt.show()
---↑in_out↓---
```

#вывод примера представлен на рисунке ниже (рис. 105)

```
---↑out---
```

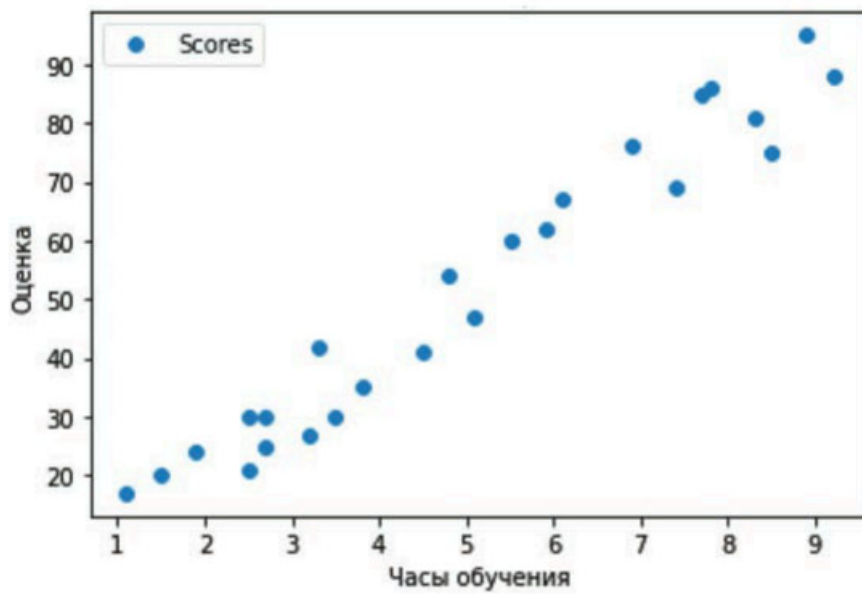


Рис. 105. Датасет (оценки в зависимости от часов обучения)

Из выведенного кодом графика мы можем увидеть, что существует некая линейная зависимость между количеством часов обучения и процентом набранных баллов.

Подготовка данных

Теперь у нас есть представление о наших данных. Следующим шагом является разделение данных на «атрибуты» и «метки». Атрибуты — это независимые переменные, а метки — это зависимые переменные, значения которых должны быть предсказаны.

В нашем наборе данных всего два столбца. Мы хотим предсказать оценку в зависимости от часов обучения. Поэтому наш набор атрибутов будет состоять из столбца «Часы», а меткой будет столбец «Оценка». Чтобы извлечь атрибуты и метки, выполним следующий код:

```
---in↓---
X=dataset.iloc[:, :-1].values #последний столбец, т.е. оценки
y=dataset.iloc[:, 1].values #первый столбец, т.е. часы
---↑in_out↓---
---↑out---
```

Теперь атрибуты хранятся в переменной X, а переменная Y содержит метки. Напоминаем, что «:» — это знак среза, и так как не указано индексов, то в срез попадут все строки. Индекс «-1» — это первый с конца элемент, или последний столбец, и до него не включительно мы сделали срез⁶⁰.

Теперь, когда у нас есть атрибуты и метки, следующим шагом будет разделение этих данных на обучающий и тестовый наборы. Мы сделаем это с помощью встроенного в Scikit-learn метода `train_test_split()`⁶¹:

```
---in↓---
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=0)
---↑in_out↓---
---↑out---
```

Приведенный выше код разделяет: 80 % данных — на обучающий набор, а 20 % данных — на тестовый. Именно параметр `test_size` используется для задания соотношения обучающего и тестового набора.

Обучение

Для импорта модели линейной регрессии и запуска машинного обучения выполним следующий код:

⁶⁰ Это то же самое, как если бы мы просто взяли столбец с нулевым индексом. Но «`dataset.iloc[:, 0].values`» вернет «`[..., ..., ...]`», а «`dataset.iloc[:, :-1].values`» вернет «`[..., [...], [...], [...]]`» — и нам такая форма как раз и нужна. Поэтому было сделано так, в целях краткости кода.

⁶¹ Метод `train_test_split()` случайным образом разбивает данные на обучающие и тестовые наборы, и такие разбиения каждый новый раз будут отличаться.

```

---in↓---
from sklearn.linear_model import LinearRegression

regressor = LinearRegression()
regressor.fit(X_train, y_train);
---↑in_out↓---
---↑out---
```

При помощи `Scikit-learn` очень просто реализовать модели линейной регрессии, поскольку все, что необходимо сделать, — это импортировать класс `LinearRegression`, создать его экземпляр и вызвать метод `fit()`, передав ему обучающую выборку.

Модель линейной регрессии в ходе своего обучения найдет наиболее подходящее значение для коэффициента сдвига и наклона, с которыми уравнение прямой наилучшим образом соответствует данным. Чтобы увидеть значение точки пересечения и наклона, вычисленное алгоритмом линейной регрессии для нашего набора данных, выполним следующий код:

```

---in↓---
print("Смещение (значение точки пересечения): ",
regressor.intercept_)
print("Коэффициент наклона: ", regressor.coef_[0])
---↑in_out↓---
Смещение (значение точки пересечения):  2.018160041434662
Коэффициент наклона:  9.91065648064224
---↑out---
```

После того как мы подставим эти значения в уравнение прямой, станет возможным сравнить результаты с исходными данными, но мы сделаем это немного позже — на этапе, когда построим график предсказаний в сравнении с графиком реальных данных.

Прогнозы

Теперь, когда мы обучили модель, пришло время протестировать ее и сделать некоторые прогнозы. Для этого мы воспользуемся нашими тестовыми данными и посмотрим, насколько точно

алгоритм предсказывает оценку. Чтобы сделать прогнозы на тестовых данных, выполним код, представленный ниже:

```
---in↓---
#Y_pred – это массив numpy, который содержит все
предсказанные значения для входных значений в серии X_test.
y_pred = regressor.predict(X_test)

df = pd.DataFrame({'Действительное': y_test,
                   'Предсказанное': y_pred})
df
print (df)
---↑in_out↓---
   Действительное  Предсказанное
0                20      16.884145
1                27      33.732261
2                69      75.357018
3                30      26.794801
4                62      60.491033
---↑out---
```

Наша модель не очень точна, но все же прогнозируемые оценки близки к фактическим. Визуализируем предсказания в сравнении с реальными данными при помощи кода, представленного ниже.

В данном коде вместо тестовых данных мы сгенерируем 10 дискретных значений часов, при этом минимальным часом будет являться такое же минимальное количество часов, сколько и в исходном датасете. Соответственно, так же и с максимальным.

Сейчас для вывода прямой предсказаний линейной регрессии нам не обязательно использовать именно тестовые данные, так как тестовые данные нужны для оценки точности модели, а на текущем этапе мы пока что посмотрим на визуальное представление предсказаний и реальных данных. К математической оценке точности мы перейдем позже.

```
---in↓---
import numpy as np
```



```

min=dataset['Hours'].min(); max=dataset['Hours'].max()
x_tests=np.linspace(min, max, 10) #сгенерируем 10 точек для
предсказания
x_tests=x_tests.reshape(-1, 1) # из [] в [[]]
predict_values=pd.DataFrame( {'Hours': x_tests.
reshape(-1), 'Scores':regressor.predict(x_tests)} )
print(predict_values.head(3)) #выведем первые 3 предсказания
ax = predict_values.plot(x='Hours', y='Scores', color='r',
style='.-', label='Предсказание регрессии')
print("Коэффициенты:", regressor.coef_, 'независимый член',
regressor.intercept_)
print("То есть в одномерном случае  $y=kx+b$ :  $y=$ ", regressor.
coef_[0], '*x+', regressor.intercept_, sep='')
print("Например см. 1 строчку предсказаний  $y=9.91*1.1+2.02=$ "
,9.91*1.1+2.018)

dataset.plot(ax=ax, x='Hours', y='Scores', style='o',
label='Реальные данные') #style='o' - чтобы рисовалась
точка, а не линия от прошлой точки к текущей
plt.title('Часы vs Оценка'); plt.xlabel('Часы обучения');
plt.ylabel('Оценка')
plt.show()
---↑in_out↓---
  Hours  Scores
0  1.1 12.919882
1  2.0 21.839473
2  2.9 30.759064
Коэффициенты: [9.91065648] независимый член
2.018160041434662
То есть в одномерном случае  $y=kx+b$ :  $y=9.91065648064224$ 
*x+2.018160041434662
Например, см. 1 строчку предсказаний  $y=9.91*1.1+2.02=12.919$ 
#часть вывода представлена в начале раздела о линейной
регрессии (Рис. 104)
---↑out---
```

Полученное значение смещения вербально означает, что за каждый час изменение оценки составляет около 9.91 %. Или, проще говоря, если студент учится на один час больше, чем готовил-

ся к экзамену ранее, он может рассчитывать на повышение на 9.91 % баллов за экзамен сравнительно с потенциально полученными студентом баллами, если бы он готовился на час меньше.

*Оценка алгоритма
(коэффициент детерминации, или R^2)*

Последний шаг — оценить алгоритм. Этот шаг особенно важен для сравнения того, насколько хорошо разные алгоритмы работают с конкретным набором данных. Для оценки алгоритмов регрессии обычно используются три рассмотренных ранее метрики на занятии «TensorFlow Keras. Решение математических задач» в разделе «Ошибки (MSE, RMSE и MAE)» на стр. 381, но также важным показателем является R^2 (R-квадрат, или коэффициент детерминации).

Коэффициент детерминации — это доля дисперсии зависимой переменной, объясняемая рассматриваемой моделью. То есть если модель «объясняет», т.е. предсказывает, правильно все 100 % возможных выходных значений, то коэффициент детерминации равен единице. Если же он равен 0, это означает, что связь между переменными регрессионной модели отсутствует, а если R^2 меньше нуля, то вместо модели для оценки значения выходной переменной будет более точным использование простого среднего ее наблюдаемых значений, т.е. модель выдает совершенно неправильные результаты — более неправильные, чем если бы это было среднее число. Обычно модель считается адекватной при коэффициенте детерминации, равном 0.9 и выше.

Давайте выведем и проанализируем все четыре метрики. Для этого в Scikit-learn доступен модуль `metrics`:

```
---in↓---
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_
error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_
error(y_test, y_pred))
```

```

print('Root Mean Squared Error:', np.sqrt(metrics.mean_
squared_error(y_test, y_pred)))
print('r2_score: ', metrics.r2_score(y_test, y_pred))
---↑in_out↓---
Mean Absolute Error: 4.183859899002982
Mean Squared Error: 21.598769307217456
Root Mean Squared Error: 4.647447612100373
r2_score: 0.9454906892105354
---↑out---
```

Мы видим, что значение корня из среднеквадратичной ошибки (т.е. *RMSE*) 4.64, что составляет менее 10 % от среднего значения оценок всех студентов, т.е. 51.48 (его мы узнали в самом начале при открытии датасета). Это означает, что наш алгоритм имеет неплохую точность.

Мы изучили линейную регрессию с участием двух переменных. Почти все проблемы реального мира, с которыми вы столкнетесь, будут иметь более двух переменных. Для решения подобных задач применяется множественная линейная регрессия, т.е. множественной называют линейную регрессию, в модели которой число независимых переменных два или более. Отличие между простой и множественной линейной регрессией заключается в том, что вместо линии регрессии в ней применяется гиперплоскость (как уже было сказано).

Преимущество множественной линейной регрессии⁶² по сравнению с простой заключается в том, что использование в модели нескольких входных переменных позволяет увеличить долю объясненной дисперсии выходной переменной, и таким образом улучшается точность модели. То есть при добавлении в модель каждой новой переменной коэффициент детерминации растет.

Деревья решений (decision tree)

Использование деревьев решений для прогнозного анализа имеет **ряд преимуществ**:

1. Деревья решений могут быть использованы для прогнозирования как непрерывных, так и дискретных значений, т.е. они хо-

⁶² Попробовать поработать с множественной регрессией вы можете на основе источника [61]. Датасет для него можно скачать в источнике [62].

рошо работают как для задач регрессии, так и для задач классификации.

2. Они требуют относительно меньших усилий для обучения алгоритма.

3. Они могут быть использованы для классификации нелинейно разделимых данных.

4. Они очень быстры и эффективны по сравнению с другими алгоритмами классификации.

Дерево решений — это один из наиболее часто и широко используемых алгоритмов контролируемого машинного обучения, который может выполнять как регрессионные, так и классификационные задачи. «Интуиция» (логика), лежащая в основе алгоритма дерева решений, проста, но в то же время очень мощна.

Для каждого атрибута в наборе данных алгоритм дерева решения формирует узел, где наиболее важный атрибут помещается в корневой узел (т.е. остается в текущем узле как готовое решение). Если условие корневого узла не выполняется, то данные переходят в следующий узел, и это продолжается до тех пор, пока не будет достигнут конечный узел, также содержащий предсказание или результат дерева решений.

Представьте себе ситуацию, когда человек просит вас одолжить ему машину на день, и вы должны принять решение, сделать это или нет. Есть несколько факторов, которые помогают определить ваше решение, примеры некоторых из них перечислены ниже:

1. Является ли этот человек близким другом или просто знакомым? Если человек просто знакомый, то отклоните просьбу; если человек друг, то переходите к следующему шагу.

2. Человек просит машину в первый раз? Если да, одолжите машину, в противном случае переходите к следующему шагу.

3. Была ли машина повреждена в прошлый раз, когда он вернул машину? Если да, отклоните просьбу; если нет, одолжите ему машину.

Дерево решений для вышеупомянутого сценария выглядит представленным на рисунке ниже образом (рис. 106).

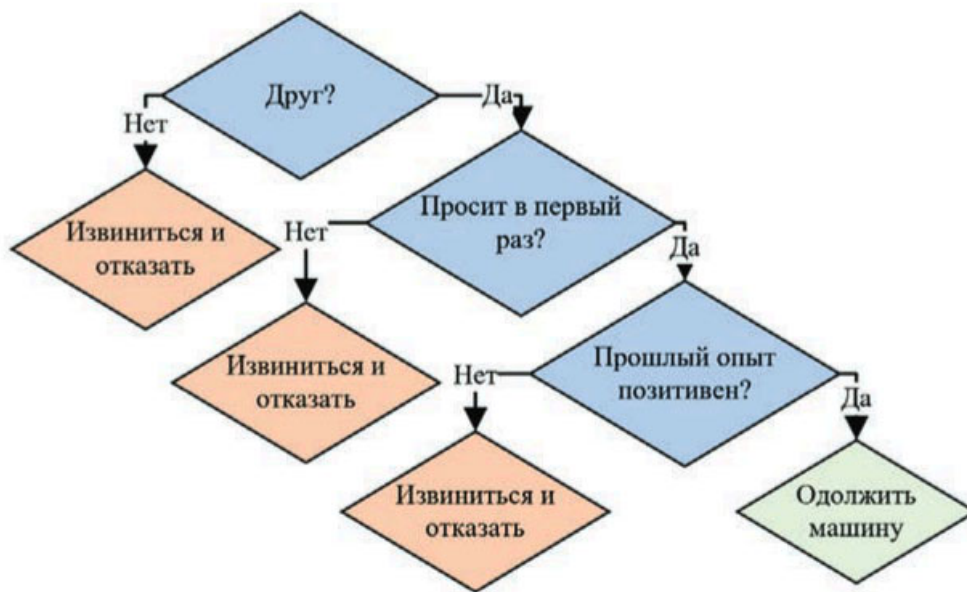


Рис. 106. Дерево решений

Последовательность работы с деревьями решений для классификации и регрессии в `Scikit-learn` аналогична списку действий, которые были выполнены и с другими рассмотренными ранее алгоритмами.

Сначала мы загрузим наш датасет, затем мы инициализируем дерево решений для классификации. Далее запустим обучение при помощи метода `fit()`, передав ему обучающую выборку и протестируем работу модели.

В качестве набора данных будет использован датасет ирисов Фишера. Данный датасет встроен в библиотеку `Scikit-learn` ввиду того, что является академическим, аналогично ранее рассмотренному датасету `MNIST`, встроенному в `TensorFlow`.

Ирисы Фишера состоят из данных о 150 экземплярах ириса, по 50 экземпляров из трех видов:

- ирис щетинистый (`iris setosa`);
- ирис виргинский (`iris virginica`);
- ирис разноцветный (`iris versicolor`).

Для каждого экземпляра измерены четыре характеристики (в сантиметрах):

1. Длина наружной доли околоцветника (`sepal length`).
2. Ширина наружной доли околоцветника (`sepal width`).

3. Длина внутренней доли околоцветника (petal length).

4. Ширина внутренней доли околоцветника (petal width).

Подключим все необходимые модули, загрузим датасет и обучим модель:

```
---in↓---
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree

# Загрузить данные
iris = load_iris()

#создадим и обучим классификатор на основе дерева решений
clf = DecisionTreeClassifier()
clf = clf.fit(iris.data, iris.target)
---↑in_out↓---
---↑out---
```

Scikit-learn позволяет визуализировать дерево решений. Для этого есть несколько настраиваемых опций, которые помогут визуализировать узлы принятия решений и разбить изученную модель, что очень полезно для понимания того, как она работает. Ниже мы раскрасим узлы на основе имен признаков и отобразим информацию о классе и объектах каждого узла.

```
---in↓---
plt.figure(figsize=((27, 13)))
plot_tree(
    clf,
    filled=True,
    feature_names=iris.feature_names,
    class_names=iris.target_names,
    rounded=True,)
plt.show()
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 107)
---↑out---
```

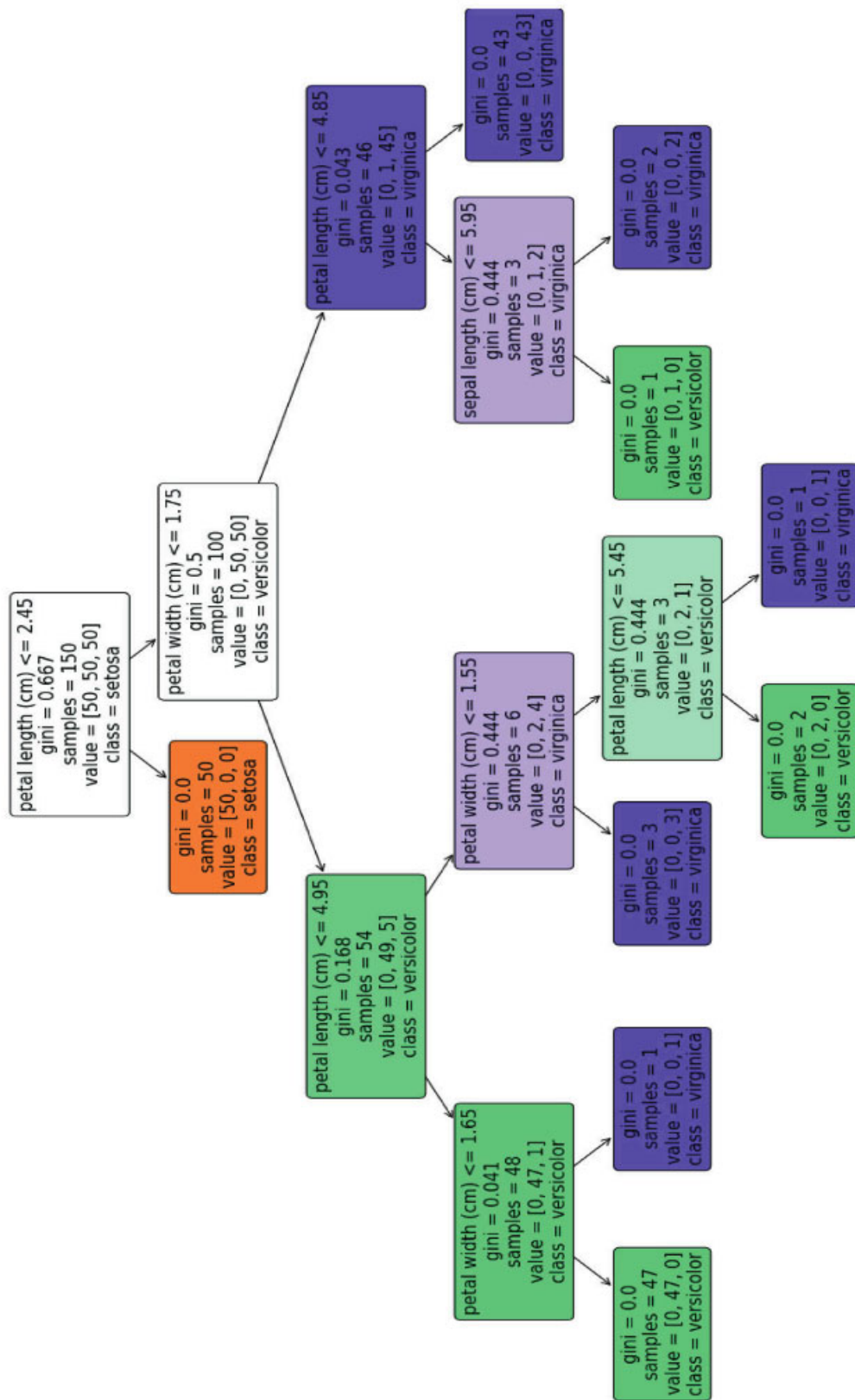


Рис. 107. Дерево решений (классификация ирисов)

Обратите внимание на коэффициент Джини (`gini`) на рисунке из вывода примера выше. Коэффициент Джини — это статистический показатель степени расслоения по какому-либо изучаемому признаку. Он измеряет степень или вероятность того, что конкретная переменная будет неправильно классифицирована (не удовлетворит условию), когда она выбрана случайным образом. Коэффициент Джини лежит в диапазоне от 0 до 1, а если он равен 0.5, то значит, элементы распределены (распределяются) равномерно на классы.

Например, посмотрите на самый верхний узел, у него `gini=0.667`. Если объяснять графически, то это означает, что 33 % (`samples=50`) элементов распределяются влево, а остальные 67 % (`samples=100`) — вправо. Имеется в виду, что `samples=150` корневого узла разделяется в пропорции 0.667. И далее происходит подобное аналогично на каждом узле.

Построим поверхность принятия решений дерева решений, обученного парам характеристик ирисов. Всего характеристик 4, значит, попарных сочетаний без повторений будет 6. Для каждой пары характеристик цветка модель изучает границы решения, состоящие из комбинаций простых пороговых правил, выведенных из обучающих выборок.

```
---in↓---
# параметры
n_classes = 3
plot_colors = "ryb"
plot_step = 0.02

plt.rcParams["figure.figsize"] = (20,10)
plt.rcParams.update({'font.size': 22})

for pairidx, pair in enumerate([[0, 1], [0, 2], [0, 3],
[1, 2], [1, 3], [2, 3]]):
    # Мы берем только две соответствующие оси
    X = iris.data[:, pair]
    y = iris.target

    clf_pair = DecisionTreeClassifier().fit(X, y) # Обучение
```



```

# Построение границы решений
plt.subplot(2, 3, pairidx + 1)

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid( #генерация точек, которые
    распределены как сетка
        np.arange(x_min, x_max, plot_step), np.arange
(y_min, y_max, plot_step)
    )

Z = clf_pair.predict(np.c_[xx.ravel(), yy.ravel()])
#ravel()- сделать массив плоским
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu)
#z - значение высоты на которой рисуется контур

plt.xlabel(iris.feature_names[pair[0]])
plt.ylabel(iris.feature_names[pair[1]])

# Построение точек обучения
for i, color in zip(range(n_classes), plot_colors):
    idx = np.where(y == i)
    plt.scatter(
        X[idx, 0],
        X[idx, 1],
        c=color,
        label=iris.target_names[i],
        edgecolor="black",
        s=20, ) #размер маркера
plt.suptitle("Поверхность принятия решений дерева решений
с использованием парных функций")
plt.legend(bbox_to_anchor=(1.55, 1.35)) #отобразить
названия классов правее
---↑in_out↓---
#вывод представлен на рисунке ниже (рис. 108)
---↑out---
```

На поверхности принятия решения цветами закрашены области кластеризации. Видно, что они вполне однозначны, и по данной поверхности можно достаточно удобно по сочетанию двух при-

знаков определить, к какому кластеру относится цветок с теми или иными двумя признаками.

Закрепим полученное знание, для этого выведем, сколько в последней паре элементов в каждом кластере. Отметим, что вы их можете посчитать на поверхности принятия решений, но по факту больше половины парных значений повторяются, есть даже те, что повторяются 8 раз (а всего, напомним, в датасете по 50 цветов в каждом из трех классов) и поэтому на поверхности принятия решений вы их не увидите (перекроют друг друга).

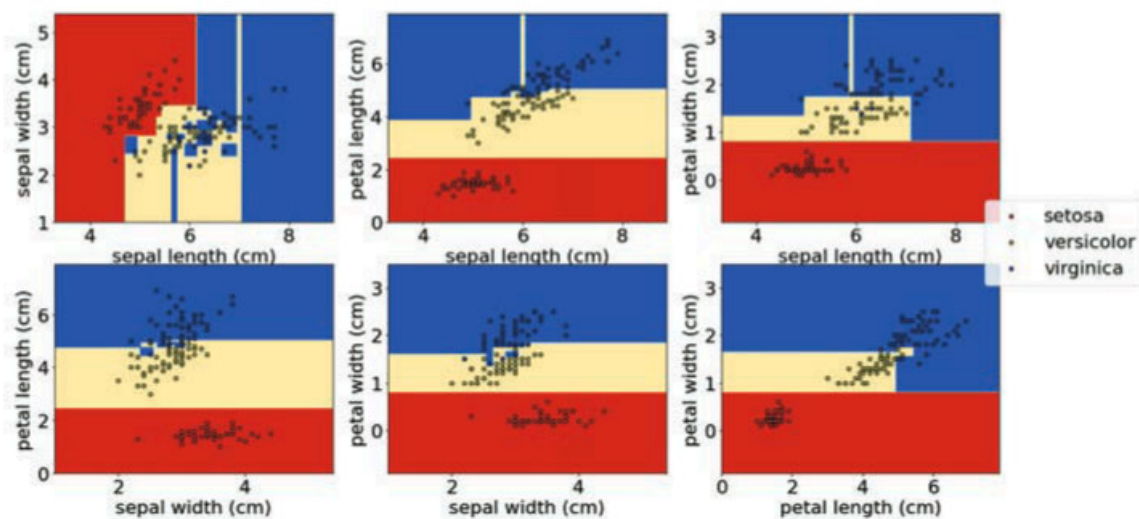


Рис. 108. Поверхность принятия решений с использованием парных функций

Как итог, выведем не только обычное количество элементов в кластере последней пары, но и, например, количество уникальных *setosa* в этой паре. Сравните полученное значение 22 с поверхностью принятия решений методом подсчета точек на 6-й диаграмме.

```
---in↓---
y_pred_last= clf_pair.predict(iris.data[:, pair])
#получим предсказания для последней пары

setosa_unique=len(np.unique(iris.data[:50, pair], return_counts=True, axis=0)[1]) #посчитаем уникальные строки
с данными характеристиками
```

```

print ('В последней паре у кластеров столько элементов:')
print ('0)setosa',(y_pred_last==0).sum(), ', но по факту уникальных только', setosa_unique, 'см. кол-во setosa на 6-м графике' )
print ('1)versicolor',(y_pred_last==1).sum())
print ('2)virginica',(y_pred_last==2).sum())
---↑in_out↓---
В последней паре у кластеров столько элементов:
0)setosa 50 , но по факту уникальных только 22 см. кол-во setosa на 6-м графике
1)versicolor 49
2)virginica 51
---↑out---
```

Обратите внимание, что имеются расхождения по количеству цветов в кластерах: должно было быть по 50 штук в каждом кластере, как и в исходном датасете:

```

---in↓---
print (np.unique(iris.target, return_counts=True, axis=0))
print ('[индексы кластеров], [количество элементов в кластерах]')
---↑in_out↓---
(array([0, 1, 2]), array([50, 50, 50], dtype=int64))
[индексы кластеров], [количество элементов в кластерах]
---↑out---
```

Это вызвано тем, что мы обучали модель только 6-й парой характеристик. Ранее упоминалось, что чем больше различных параметров, тем выше сходимость модели и ее коэффициент детерминации (r^2_score). Давайте проверим это. Ниже мы выведем ошибки для модели, обученной только 6-й парой, и модели, обученной всеми параметрами.

```

---in↓---
print('При обучении только по 6 паре характеристик есть небольшая ошибка кластеризации:')
print('(имеется в виду при проверке всем датасетом)')
print('Mean Absolute Error:', metrics.mean_absolute_error(iris.target, y_pred_last))
```

```
print('Mean Squared Error:', metrics.mean_squared_error(iris.target, y_pred_last))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(iris.target, y_pred_last)))
print('r2_score: ', metrics.r2_score(iris.target, y_pred_last))
```

```
y_pred_all= clf.predict(iris.data) #получим предсказания для всех данных
```

```
print('\nПри обучении на всех данных ошибок кластеризации, как ни странно, нет:')
```

```
print('Mean Absolute Error:', metrics.mean_absolute_error(iris.target, y_pred_all))
```

```
print('Mean Squared Error:', metrics.mean_squared_error(iris.target, y_pred_all))
```

```
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(iris.target, y_pred_all)))
```

```
print('r2_score: ', metrics.r2_score(iris.target, y_pred_all))
```

```
---↑in_out↓---
```

При обучении только по 6 паре характеристик есть небольшая ошибка кластеризации:

(имеется в виду при проверке всем датасетом)

Mean Absolute Error: 0.006666666666666667

Mean Squared Error: 0.006666666666666667

Root Mean Squared Error: 0.08164965809277261

r2_score: 0.99

При обучении на всех данных ошибок кластеризации, как ни странно, нет:

Mean Absolute Error: 0.0

Mean Squared Error: 0.0

Root Mean Squared Error: 0.0

r2_score: 1.0

```
---↑out---
```

Действительно, мы видим, что при обучении с использованием всех параметров точность модели выше (в частности, обратите внимание на коэффициент детерминации `r2_score`). В данном случае точность абсолютна — нет ошибок предсказаний. Это выглядит очень странно, но подтверждается информацией из се-

ти «Интернет»: дерево решений обучается на датасете ирисов очень точно.

Так как коэффициент детерминации, который является долей объясненных дисперсий, равен 1, значит, все дисперсии объяснены.

Но мы обучали и оценивали модель на одних и тех же данных — хорошо, она всем им обучилась и все решила правильно. Но грамотным подходом при оценивании является оценивание при помощи новых данных, т.е. при помощи тестовой выборки, мы уже рассматривали это ранее в примере линейной регрессии:

```

---in↓---
X_train, X_test, y_train, y_test = train_test_split(iris.
data, iris.target, test_size=0.2, random_state=0)
clf = DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)

y_pred_all= clf.predict(X_test) #получим предсказания для
всех данных
print('\nПри обучении на 80% данных и тесте на 20% ошибок
все равно нет:')
print('Mean Absolute Error:', metrics.mean_absolute_
error( y_test, y_pred_all))
print('Mean Squared Error:', metrics.mean_squared_error(
y_test, y_pred_all))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_
squared_error( y_test, y_pred_all)))
print('r2_score: ', metrics.r2_score( y_test,y_pred_all))
---↑in_out↓---
При обучении на 80% данных и тесте на 20% ошибок все
равно нет:
Mean Absolute Error: 0.0
Mean Squared Error: 0.0
Root Mean Squared Error: 0.0
r2_score: 1.0
---↑out---
```






Теперь точно все выполнено правильно. Видимо, мы сделали правильный вывод: модель очень точна, даже на первый взгляд







пугающе точно, совсем нет ошибок, даже минимальных. Но они появляются, если, например, изменить соотношение тестовой и обучающей выборки.

? Задания для проверки







1. Опишите, какие возможности предоставляет библиотека `Scikit-learn`. Чем она схожа и чем отличается с `TensorFlow`?
2. Что такое кластерный анализ и где применяется? Чем он отличается от классификации?
3. Линейная регрессия. Проведите эксперимент с собственным датасетом (к примеру, подготовьте в `Excel` датасет случайных значений, лежащих возле какого-либо уравнения).
4. Проведите и проанализируйте свои собственные эксперименты с `K-средних` и деревом решений.







БУДЕТ ПОЛЕЗНО







№	Источник	QR-код (если применимо)
1.	Яндекс.Диск // Яндекс : [сайт]. — URL: https://disk.yandex.ru/d/KcP6tDe2y_2kLw (дата обращения: 25.10.2022).	
2.	Anaconda. The World's Most Popular Data Science Platform. Documentation // Anaconda : [site]. — URL: https://www.anaconda.com/ (дата обращения: 17.10.2022).	
3.	Anaconda Installers and Packages. Index // Anaconda : [site]. — URL: https://repo.anaconda.com/archive/ (дата обращения: 17.10.2022).	
4.	PyCharm: the Python IDE for Professional Developers // JetBrains : [site]. — URL: https://www.jetbrains.com/pycharm/ (дата обращения : 17.10.2022).	
5.	Conda // Conda : Documentation : [site]. — URL: https://docs.conda.io/en/latest/ (дата обращения: 17.10.2022).	







№	Источник	QR-код (если применимо)
6.	Pip // Pip : Documentation : [site]. — URL: https://pip.pypi.org/en/stable/ (дата обращения: 17.10.2022).	
7.	Python // Python Official Site: Introduction to Python : [site]. — URL: https://www.python.org/ (дата обращения: 17.10.2022).	
8.	Jupyter: Project Documentation // Jupyter : [site]. — URL: https://docs.jupyter.org/en/latest/ (дата обращения: 17.10.2022).	
9.	Package repository for anaconda // Anaconda. org : [site]. — URL: https://anaconda.org/anaconda/repo (дата обращения: 17.10.2022).	
10.	The Python Package Index (PyPI) // PyPI : [site]. — URL: https://pypi.org/ (дата обращения: 17.10.2022).	
11.	Installing Python Packages from a Jupyter Notebook // Pythonic Perambulations : [site]. — URL: https://jakevdp.github.io/blog/2017/12/05/installing-python-packages-from-jupyter/ (дата обращения: 17.10.2022).	






№	Источник	QR-код (если применимо)
12.	Download // Graphviz : [site]. — URL: https://graphviz.org/download/ (дата обращения: 17.10.2022).	
13.	GitHub: Python3_Jupyter_Notebook // GitHub : [site]. — URL: https://github.com/jvdkwast/Python3_Jupyter_Notebook (дата обращения: 17.10.2022).	
14.	PEP 8 – Style Guide for Python Code // PEP Index : [site]. — URL: https://www.python.org/dev/peps/pep-0008/ (дата обращения: 17.10.2022).	
15.	Объектно-ориентированное программирование на Python // Викиучебник : [сайт]. — URL: https://ru.wikibooks.org/wiki/Python/Объектно-ориентированное_программирование_на_Python (дата обращения: 17.10.2022).	
16.	The Python Tutorial // Python : [site]. — URL: https://docs.python.org/3/tutorial/ (дата обращения: 17.10.2022).	
17.	Caret // Wikipedia, The Free Encyclopedia : [site]. — URL: https://wikipedia.org/wiki/Caret (дата обращения: 17.10.2022).	







№	Источник	QR-код (если применимо)
18.	Регулярные выражения в Python // python-scripts : [site]. — URL: https://python-scripts.com/import-re-regular-expression (дата обращения: 17.10.2022).	
19.	Multiprocessing — Process-based parallelism // Python 3.10.8 documentation : [site]. — URL: https://docs.python.org/3/library/multiprocessing.html (дата обращения: 17.10.2022).	
20.	Python 3 для начинающих // Pythonworld: NumPy : [site]. — URL: https://pythonworld.ru/numpy (дата обращения: 17.10.2022).	
21.	NumPy: Reference // NumPy : [site]. — URL: https://numpy.org/doc/stable/reference/ (дата обращения: 17.10.2022).	
22.	Data type objects (dtype) // NumPy Manual : [site]. — URL: https://numpy.org/doc/stable/reference/arrays.dtypes.html (дата обращения: 17.10.2022).	
23.	Numpy.set_printoptions // NumPy Manual : [site]. — URL: https://numpy.org/doc/stable/reference/generated/numpy.set_printoptions.html (дата обращения: 17.10.2022).	





№	Источник	QR-код (если применимо)
24.	Mathematical functions [Электронный ресурс] // NumPy Manual : [site]. — URL: https://docs.scipy.org/doc/numpy/reference/routines.math.html (дата обращения: 17.10.2022).	
25.	Изучаем pandas. Урок 1. Введение в pandas и его установка // Devpractice : [site]. — URL: https://devpractice.ru/pandas-start-part1/ (дата обращения: 17.10.2022).	
26.	Pandas documentation // Pandas : [site]. — URL: https://pandas.pydata.org/docs/index.html (дата обращения: 17.10.2022).	
27.	NumPy Statistical Functions with Examples // Data-flair : [site]. — URL: https://data-flair.training/blogs/numpy-statistical-functions/ (дата обращения: 17.10.2022).	
28.	SciPy Documentation // SciPy : [site]. — URL: https://docs.scipy.org/doc/scipy/ (дата обращения: 17.10.2022).	
29.	API Reference // Matplotlib : [site]. — URL: https://matplotlib.org/stable/api/index.html (дата обращения: 17.10.2022).	




№	Источник	QR-код (если применимо)
30.	Plotly Python Open Source Graphing Library // Plotly : [site]. — URL: https://plotly.com/python/ (дата обращения: 17.10.2022).	
31.	Statistical functions (scipy.stats) // SciPy Manual : [site]. — URL: https://docs.scipy.org/doc/scipy/reference/stats.html (дата обращения: 17.10.2022).	
32.	Statistics (scipy.stats) // SciPy Manual : [site]. — URL: https://docs.scipy.org/doc/scipy/tutorial/stats.html (дата обращения: 17.10.2022).	
33.	Введение в описательную статистику // machinelearningmastery : [site]. — URL: https://www.machinelearningmastery.ru/intro-to-descriptive-statistics-252e9c464ac9/ (дата обращения: 17.10.2022).	
34.	Built-in magic commands // IPython documentation : [site]. — URL: https://ipython.readthedocs.io/en/stable/interactive/magics.html (дата обращения: 17.10.2022).	
35.	Семен Лукашевский. Погружаемся в статистику вместе с Python. Часть 1. Z-статистика и p-value // Хабр : [сайт]. — URL: https://habr.com/ru/post/557424/ (дата обращения: 17.10.2022).	

№	Источник	QR-код (если применимо)
36.	Scipy.stats.gamma // SciPy Manual : [site]. — URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gamma.html (дата обращения: 17.10.2022).	
37.	Numpy.random.Generator.gamma // NumPy Manual : [site]. — URL: https://numpy.org/doc/stable/reference/random/generated/numpy.random.Generator.gamma.html (дата обращения: 17.10.2022).	
38.	Нормальное распределение // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/Нормальное_распределение (дата обращения: 17.10.2022).	
39.	Scipy.stats.expon // SciPy Manual : [site]. — URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.expon.html (дата обращения: 17.10.2022).	
40.	Гамма-распределение // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/Гамма-распределение (дата обращения: 17.10.2022).	
41.	Understanding Boxplots: How to Read and Interpret a Boxplot // Built In : [site]. — URL: https://builtin.com/data-science/boxplot (дата обращения: 17.10.2022).	

№	Источник	QR-код (если применимо)
42.	Sqlite3 — DB-API 2.0 interface for SQLite databases // Python : [site]. — URL: https://docs.python.org/3/library/sqlite3.html (дата обращения: 17.10.2022).	
43.	Bariş Karaman. Know Your Metrics // Towards data science : [site]. — URL: https://towardsdatascience.com/data-driven-growth-with-python-part-1-know-your-metrics-812781e66a5b (дата обращения: 17.10.2022).	
44.	Eryk Lewinson. A step-by-step introduction to Cohort Analysis in Python // Towards data science : [site]. — URL: https://towardsdatascience.com/a-step-by-step-introduction-to-cohort-analysis-in-python-a2cbbd8460ea (дата обращения: 17.10.2022).	
45.	Online Retail Data Set // Kaggle : [site]. — URL: https://www.kaggle.com/vijayuv/onlineretail (дата обращения: 17.10.2022).	
46.	Использование библиотеки TensorFlow для машинного обучения при решении задач построения, обучения, оценки и использования нейронных сетей. Лабораторный практикум : учебное пособие / А.П. Никищечкин, П.И. Меликов. — Москва : Изд-во МГТУ «СТАНКИН», 2021. — 94 с .	печатное издание
47.	System Architecture. Cloud TPU // Google Cloud : [site]. — URL: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm (дата обращения: 17.10.2022).	

№	Источник	QR-код (если применимо)
48.	Layer activation functions // Keras documentation : [site]. — URL: https://keras.io/api/layers/activations/ (дата обращения: 17.10.2022).	
49.	Optimizers // Keras documentation : [site]. — URL: https://keras.io/api/optimizers/ (дата обращения: 17.10.2022).	
50.	Writing your own callbacks // TensorFlow Core : [site]. — URL: https://www.tensorflow.org/guide/keras/custom_callback (дата обращения: 17.10.2022).	
51.	Метод обратного распространения ошибки // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/Метод_обратного_распространения_ошибки (дата обращения: 17.10.2022).	
52.	GIMP — Downloads // GIMP : [site]. — URL: https://www.gimp.org/downloads/ (дата обращения: 17.10.2022).	
53.	User Guide // Scikit-learn : [site]. — URL: https://scikit-learn.org/stable/user_guide.html (дата обращения: 17.10.2022).	

№	Источник	QR-код (если применимо)
54.	Введение в Scikit-learn // Neurohive : [site]. — URL: https://neurohive.io/ru/osnovy-data-science/vvedenie-v-scikit-learn/ (дата обращения: 17.10.2022).	
55.	K-Means Clustering Example // Notebook community : [site]. — URL: https://notebook.community/cgivre/oreilly-sec-ds-fundamentals/Notebooks/Unsupervised/K-Means%20Clustering%20Example (дата обращения: 17.10.2022).	
56.	Simple Linear Regression // Kaggle : [site]. — URL: https://www.kaggle.com/ahmedimamlos/simple-linear-regression (дата обращения: 17.10.2022).	
57.	Деревья решений в Python с Scikit-Learn // Pythobyte : [site]. — URL: https://pythobyte.com/decision-trees-in-python-with-scikit-learn-9e8b0826/ (дата обращения: 17.10.2022).	
58.	Examples // scikit-learn documentation : [site]. — URL: https://scikit-learn.org/stable/auto_examples/index.html (дата обращения: 17.10.2022).	
59.	Clustering // scikit-learn documentation : [site]. — URL: https://scikit-learn.org/stable/modules/clustering.html (дата обращения: 17.10.2022).	

№	Источник	QR-код (если применимо)
60.	Student_scores // Kaggle : [site]. — URL: https://www.kaggle.com/kamleshhsam/student-scores (дата обращения: 17.10.2022).	
61.	Linear Regression in Python with Scikit-Learn // Stack Abuse : [site]. — URL: https://stackabuse.com/linear-regression-in-python-with-scikit-learn (дата обращения: 17.10.2022).	
62.	Petrol consumption // Kaggle : [site]. — URL: https://www.kaggle.com/harinir/petrol-consumption (дата обращения: 17.10.2022).	



IPR MEDIA
ИЗДАТЕЛЬСТВО

С 2005 года выпускаем качественную литературу, развиваем инновационные платформы и меняем систему образования

ЭКОСИСТЕМА IPR SMART

IPR SMART – цифровой образовательный ресурс, большая электронная библиотека учебных и практических изданий, а также современные, удобные сервисы для преподавания и обучения, в том числе с применением адаптивных технологий. Включает более 155 000 книг, журналов, аудиозданий.



МОБИЛЬНЫЕ ПРИЛОЖЕНИЯ



IPR SMART Mobile Reader

Мобильное приложение полностью обеспечивает эффективную и удобную работу пользователей с книгами на смартфоне или планшете.



IPR BOOKS-WV Reader

Мобильное приложение для людей с ограниченными возможностями по зрению.

Приложения работают на операционных системах iOS и Android



DataLIB

– первая образовательная платформа по освоению цифровых компетенций – более 3 500 изданий по информационным и сквозным цифровым технологиям, а также набор интерактивных инструментов для преподавания и обучения (SMART-курсы, модуль «Цифровая кафедра», система групповой работы, лекторий).



РКИ

– современный цифровой ресурс для обучения иностранных студентов и абитуриентов, содержащий издания, онлайн-курсы, тесты, аудио- и видеоматериалы.



**Заинтересованы
в сотрудничестве с нами?**

**Следите за новостями
в социальных сетях**



8 800 555 22 35

Звонок бесплатный по всей России



izdat@iprmedia.ru

Ответим на все Ваши вопросы



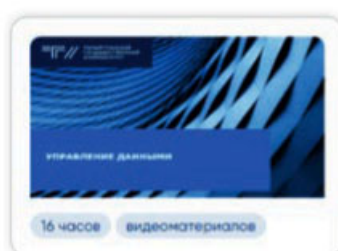
@iprmedia



@ipredu_online

Получайте образование уже сегодня на платформе онлайн-обучения DATALIB.RU

- ✓ Онлайн-курсы и другой контент по всем сквозным цифровым технологиям и направлениям подготовки
- ✓ Топовые образовательные программы от ведущих университетов страны
- ✓ Лекторий с лучшими спикерами – экспертами-практиками и преподавателями
- ✓ Возможность выстроить свою образовательную траекторию на платформе и обучаться у лучших
- ✓ Образовательные сертификаты всем слушателям



16 часов видеоматериалов

Управление данными



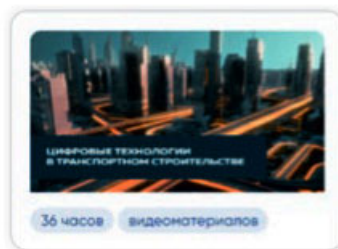
10 часов видеоматериалов

Data Presentation with Pandas



23 часа видеоматериалов

Анализ и визуализация данных.
Уровень Junior



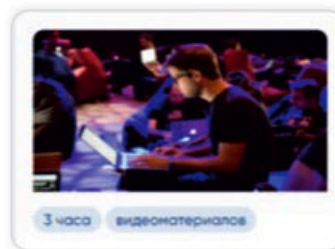
36 часов видеоматериалов

Цифровые технологии
в транспортном строительстве



10 часов видеоматериалов

Профориентация
в цифровой экономике



3 часа видеоматериалов

Цифровые ресурсы
в образовательной деятельности



Получите
тестовый
доступ



8 800 555 22 35



marketing@iprmedia.ru



Международное издательство,
выпускающее учебные, научные
и практические издания на разных языках
для всех уровней образования



– международная платформа для подготовки высоко-
квалифицированных кадров с цифровой библиотекой,
онлайн-курсами и digital-сервисами для обучения. Ресурс
объединяет лучшие международные практики на одной платформе.



- ✓ Самая большая национальная цифровая библиотека учебного и научного контента для университета и колледжей
- ✓ Только лицензионный контент на казахском, английском и русском языках
- ✓ Доступные интерактивные инструменты для обучения: онлайн-курсы и Лекторий
- ✓ Современные сервисы для преподавания и обучения
- ✓ Удобный модуль для выстраивания «Цифрового университета»

Станьте автором международного уровня:

- ① выгодные условия сотрудничества для авторов
- ② затраты по производству и популяризации изданий EDP Hub берет на себя
- ③ индивидуальный подход к каждой работе
- ④ публикация изданий в электронном виде на международных платформах
- ⑤ издание книг в печатном виде
- ⑥ профессиональная редакционно-издательская подготовка, присвоение ISBN
- ⑦ присвоение DOI, повышение видимости и увеличение цитируемости изданий
- ⑧ возможность участия в международных конкурсах авторских работ, ежегодно организуемых издательством

**Направьте свою авторскую
работу или откройте
тестовый доступ к платформе**



edp_hub@mail.kz



Издательство «Профобразование» – издательство учебной и профильной литературы для среднего профессионального образования. Издания включены в ПОП в качестве основной и дополнительной литературы (более 700 рекомендаций ФУМО СПО)

ПАРТНЕРСКИЕ ПРОЕКТЫ



– одна из крупнейших в России специализированных библиотек для учреждений СПО. В цифровой библиотеке доступны к подключению более 8 000 учебных изданий, журналы, онлайн-курсы, тесты, аудио- и видеоматериалы, а также учебники издательства «Просвещение», входящие в Федеральный перечень учебников.



– современное решение для проверок работ на объем заимствований и банк электронных портфолио обучающихся. Безопасный и надежный репозиторий данных, а также работ студентов и преподавателей.



**Заинтересованы
в сотрудничестве с нами?**

**Следите за новостями
в социальных сетях**



8 800 511 14 70

Звонок бесплатный по всей России



office@profspo.ru

Ответим на все Ваши вопросы



@profobrazovanie_official



@profobrexp

Издавайте книги вместе с нами!

IPR MEDIA, «Профобразование» и «Вузовское образование» – ведущие издательства, с 2005 года выпускающие высококачественную учебную и практическую литературу для высшего и среднего профессионального образования, а также практическую литературу по актуальным востребованным темам для широкого круга лиц

Преимущества работы для авторов:

- 1 индивидуальный подход
- 2 профессиональная редакционно-издательская подготовка и оперативное издание
- 3 публикация изданий в электронном и печатном виде
- 4 качественное полиграфическое исполнение изданий
- 5 выплата вознаграждения
- 6 бесплатные авторские экземпляры
- 7 рост публикационной активности и цитируемости
- 8 присвоение изданиям ISBN и DOI, передача данных в РИНЦ
- 9 защита изданий от незаконного использования и распространения
- 10 продвижение авторов и их изданий, в том числе через маркетплейсы, конкурсы, мероприятия

Хотите издать книгу?

**Свяжитесь с нами,
чтобы обсудить условия**



8 800 555 22 35

доб. 208, 227, 229



izdat@iprmedia.ru

Рекомендуемые новинки



Купить книгу



8 800 555 22 35 (доб. 214)



books@iprmedia.ru

