ПЕРМСКИЙ
ГОСУДАРСТВЕННЫЙ
НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ

Р. В. Гарафутдинов

PYTHON ДЛЯ АНАЛИЗА ДАННЫХ

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«ПЕРМСКИЙ ГОСУДАРСТВЕННЫЙ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»

Р. В. Гарафутдинов

PYTHON ДЛЯ АНАЛИЗА ДАННЫХ

Допущено методическим советом
Пермского государственного национального
исследовательского университета в качестве
учебного пособия для студентов, обучающихся
по направлениям подготовки бакалавров «Экономика»,
«Менеджмент», «Бизнес-информатика»,
«Торговое дело»

Пермь 2024

УДК 004.43(075.8)
ББК 32.973я73
Г20

Гарафутдинов Р. В.

Г20 Python для анализа данных [Электронный ресурс] : учебное пособие / Р. В. Гарафутдинов ; Пермский государственный национальный исследовательский университет. – Электронные данные. – Пермь, 2024. – 9,19 Мб ; 276 с. – Режим доступа: http://www.psu.ru/files/docs/science/books/uchebnie-posobiya/Garafutdinov-Python-dlya-analiza-dannyh.pdf. – Заглавие с экрана.

ISBN 978-5-7944-4096-6

Цель учебного пособия – помочь студентам, изучающим курс «Python для анализа данных», приобрести системные знания по основам программирования на языке Python и освоить работу со стандартными возможностями языка и его библиотеками на уровне, достаточном для решения задач обработки и анализа данных. Рассматриваются базовые элементы и синтаксические конструкции языка Python 3. Описываются основные методы и приемы работы с рядами и табличными данными с применением популярных Python-библиотек. Издание содержит тематический план, теорию с примерами программного кода, задания для самостоятельной работы и список использованных источников.

Учебное пособие подготовлено на основе опыта преподавания дисциплины студентам экономического факультета Пермского государственного национального исследовательского университета и факультета социально-экономических и компьютерных наук Национального исследовательского университета «Высшая школа экономики» (Пермь).

УДК 004.43(075.8)
ББК 32.973я73

Издается по решению ученого совета экономического факультета Пермского государственного национального исследовательского университета

Рецензенты: кафедра информационных технологий в бизнесе Пермского филиала Национального исследовательского университета «Высшая школа экономики» (и.о. зав. кафедрой – д-р пед. наук, профессор ***Е. Г. Плотникова***);
начальник управления организации научных исследований Пермского национального исследовательского политехнического университета, канд. экон. наук, доцент ***А. О. Алексеев***

ISBN 978-5-7944-4096-6

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Существует многообразие инструментов анализа данных. Они представляют собой различные программные решения, включая универсальные системы (такие как табличный процессор Microsoft Excel и его аналоги), позволяющие выполнять широкий круг задач обработки одномерных и табличных данных, и специализированные прикладные пакеты, ориентированные на анализ данных непосредственно (такие как Gretl, Statistica, EViews и т.п.). Особняком среди этих инструментов стоят языки программирования высокого уровня, наиболее популярными из которых при решении задач такого рода являются R и Python. Оба этих языка (которые зачастую незаслуженно именуют прикладными пакетами, подразумевая не то сам интерпретатор, не то среду разработки, будь то RStudio или PyCharm), будучи инструментами принципиально иного класса, нежели действительные прикладные пакеты, предоставляют фундаментально более широкие возможности обработки и анализа данных. Готовое приложение, пусть и очень функциональное, учитывающее множество сценариев работы с данными, ограничено фантазией и «представлениями о прекрасном» разработчиков этого приложения. Навыки же алгоритмизации и программирования и знание какого-либо популярного в сфере наук о данных (Data Science) языка снимает практически все ограничения с исследователя: потолок его возможностей по манипуляции с данными устанавливается лишь его собственной квалификацией (и фантазией). Цель данного пособия – вручить читателю описанный ультимативный инструмент и обучить основам его использования.

Язык R широко популярен среди аналитиков данных и других специалистов по Data Science. Если сравнивать его с Python, можно сказать, что R является более «высокоуровневым»: он рассчитан на специалистов, более близких к предметной области, в которой они работают, чем к информационным технологиям; можно сказать, что R – это «язык программирования для непрограммистов».

Настоящее учебное пособие посвящено другому языку – Python, который более универсален, в чем-то более традиционен в сравнении с R, но при этом не менее прост в освоении и совершенно точно предлагает разработчику не меньше возможностей. Более того, по мнению автора, навыки и знания, полученные при изучении Python, универсальнее и позволяют с большей легкостью переходить на использование других языков программирования, чем это позволяет знание языка R, являющегося в достаточной мере специфическим. По данным рейтинга языков программирования, ежегодно составляемого журналом IEEE Spectrum, Python по состоянию на 2023 год является самым распространенным и востребованным языком [16]. Следует отметить, что в пособии рассматривается язык

Python актуальной версии 3 как наиболее современный и активно развивающийся на момент написания данного материала.

Пособие состоит из двух разделов. В первом разделе излагаются основы программирования на языке Python. В нем в меру глубоко рассматриваются теоретические моменты, связанные с алгоритмизацией и программированием, и приводится большое количество примеров программного кода с подробными пояснениями. Второй раздел посвящен библиотекам Python, широко применяемым при анализе данных (NumPy – для векторизованной обработки многомерных массивов; pandas – для работы с рядами и таблицами; Matplotlib и seaborn – для визуализации). Он в меньшей степени ориентирован на теорию, в большей – на практические примеры обработки данных с использованием инструментов, предоставляемых этими библиотеками.

Пособие предназначено для студентов экономического факультета Пермского государственного национального исследовательского университета направлений «Экономика», «Менеджмент», «Бизнес-информатика», «Торговое дело», а также «Прикладная математика и информатика» всех форм обучения.

Издание может быть рекомендовано для широкого круга студентов, аспирантов и преподавателей, желающих овладеть языком программирования Python как инструментом, позволяющим решать задачи обработки и анализа данных, построения моделей машинного обучения и любые другие, исследовательские и практические.

Для работы с данным пособием необходим достаточно высокий уровень компьютерной грамотности. Желательно (но не является критически необходимым) знание некоторых разделов следующих учебных дисциплин: информатика, высшая математика (в части линейной алгебры), теория вероятностей и математическая статистика.

Для достижения наилучшего результата пособие следует изучать последовательно, от главы к главе. Некоторые примеры и задания для самостоятельного выполнения являются сквозными. Например, в примере (задании) могут использоваться данные, подготовленные в предыдущих параграфах (на это есть указания в тексте).

Все приведенные в пособии программные коды доступны в виде блокнотов Google Colab по следующим ссылкам: глава 3, глава 4, глава 5, глава 6, глава 7, глава 8, глава 9, глава 10, глава 11, глава 12, глава 13, глава 14. Эти блокноты подготовлены автором для удобства читателя и не содержат дополнительного материала. Если платформа Google Colaboratory в силу каких-либо причин перестанет функционировать, на доступность материалов пособия это не повлияет.

В тексте использованы следующие обозначения программного кода и различных элементов языка Python (табл. 1).

Таблица 1

Условные обозначения в тексте пособия

Описание	Пример

Программный код, который можно скопировать и выполнить, вместе с отображаемым результатом его выполнения. Используются те же цвета, что в редакторе Colab. Результат может отсутствовать в двух случаях: если код ничего не выводит; когда результирующее значение для экономии места приведено в комментарии. Если вывод кода слишком объемный и для понимания примера не требуется приводить его целиком, используется сокращение вывода с помощью символа многоточия …	`# результат приведен в области`
	`# вывода под ячейкой`
	`print("Привет, мир!")`
	`Привет, мир!`
	
	`# результат приведен`
	`# в комментарии`
	`round(3.7)  # 4`

Описание формата вызова функции с перечислением наиболее важных аргументов. В треугольные скобки <> заключены текстовые описания элементов команды	`print(<объект 1>, <объект 2>,`
	`  ..., <объект N>,`
	`  sep=<разделитель>,`
	`  end=<строка,`
	`  завершающая вывод>)`

Описание формата сложного оператора (конструкции) языка. Ключевые слова выделены цветом	`while <условие>:` `  команды тела цикла`

Фрагмент программного кода в основном тексте, в том числе отдельные переменные и литералы, функции, типы данных, атрибуты и методы объектов	`x = 2 * 3`
	`round()`
	`True`
	`str`

Оператор языка	`continue`
Название библиотеки или модуля	NumPy
Тип исключения (ошибки)	TypeError

РАЗДЕЛ 1. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА PYTHON

Предисловие к первому разделу

Прежде чем приступать к аналитике и формированию выводов на основе исследования данных, следует освоить *инструментарий* – программные средства, предоставляющие необходимые возможности.

Программных решений для анализа данных существует много. Очевидно, что читатель данного пособия уже сделал свой выбор в пользу языка программирования Python, и автором этот выбор горячо поддерживается. Действительно, владение навыками программирования на самом популярном языке позволяет реализовать любые задачи, связанные с обработкой данных, и не только применять существующие методы и методики, но и разрабатывать собственные. Есть, однако, у данного инструментария недостаток: *достаточно высокая сложность использования*. Если мы посмотрим на примеры программного кода для решения какой-нибудь в меру типовой задачи (скажем, построение столбчатой диаграммы с группировкой), которые в изобилии можно найти как в документации к библиотекам, так и на тематических ресурсах в интернете, велика вероятность того, что объем и сложность этого кода нас неприятно удивят. Во всяком случае, состоять он будет не из одной строчки, а даже если из одной, прочитать эту строчку и понять, что в ней происходит, без специальных знаний окажется задачей нетривиальной. Чем более универсален, менее «однокнопочен» инструмент, тем сложнее его эффективное использование. Последний из «трех законов» фантаста А. Кларка гласит: «Достаточно развитая технология неотличима от волшебства» [20]. Но даже юному волшебнику Гарри Поттеру из одноименной серии романов Дж. К. Роулинг пришлось несколько лет постигать искусство владения своей волшебной палочкой. Язык Python и его библиотеки, несомненно, могут стать «волшебной палочкой» для специалиста по наукам о данных, но для этого требуется время, определенная доля усилий и, конечно, желание.

Первый раздел пособия посвящен введению в программирование на языке Python с нуля. Автор категорически рекомендует изучить этот раздел, прежде чем переходить к инструментам, ориентированным на анализ данных непосредственно. Лишь после освоения основ языка программный код, приводящий к нужному результату, не будет восприниматься как «магия» с неизвестным механизмом действия. Если разобраться в принципах написания кода и ознакомиться с основными категориями программирования на Python, такими как объекты, методы, функции, коллекции и так далее, окажется, что все эти непонятные записи далеко не так сложны и запутаны, как могло показаться на первый взгляд. Во всем есть логика, код всегда составлен по определенным правилам. И тогда волшебная палочка заработает в полную силу.

1. Введение в алгоритмизацию и программирование

Первая глава знакомит читателя с основами работы компьютера, описывает процесс программирования как вид деятельности, пытается ответить на сакраментальный вопрос «а что здесь вообще происходит?». Также в ней излагаются общие сведения о языке программирования Python, его особенностях и достоинствах.

1.1. Высокоуровневые языки программирования

Алгоритм

Компьютер – это машина, обрабатывающая информацию. Обработка информации – это все, что она умеет делать.

Чтобы компьютер решил какую-то полезную задачу, необходимо ему эту задачу доходчиво объяснить, то есть дать четкую инструкцию на понятном компьютеру языке.

Алгоритм – это последовательность команд, предназначенная *для исполнителя*, в результате выполнения которой он должен решить поставленную задачу. Исполнителем может быть человек или машина.

Примерами алгоритмов для исполнителя-человека являются: инструкция по делению в столбик, рецепт варки борща, инструкция по ремонту холодильника.

Алгоритм для исполнителя-машины (компьютера), записанный на специальном языке, понятном исполнителю (компьютеру), называется ***компьютерной программой***, а этот язык – ***языком программирования***.

Алгоритм обладает рядом *свойств*:

- ✓ должен описываться на формальном языке, исключающем неоднозначность толкования;
- ✓ должен быть понятен исполнителю;
- ✓ исполнитель должен уметь выполнять все команды, составляющие алгоритм;
- ✓ множество возможных команд, из которых состоит алгоритм, конечно и изначально строго задано;
- ✓ свойство *конечности (результативности)*: алгоритм должен приводить к решению задачи (получению результата) за конечное число шагов;
- ✓ свойство *массовости*: алгоритм должен быть применим для некоторого класса задач, различающихся исходными данными;
- ✓ всегда имеет входные и выходные данные.

Устройство компьютера

Компьютер состоит из *двух важнейших частей*: вычислительный блок и память.

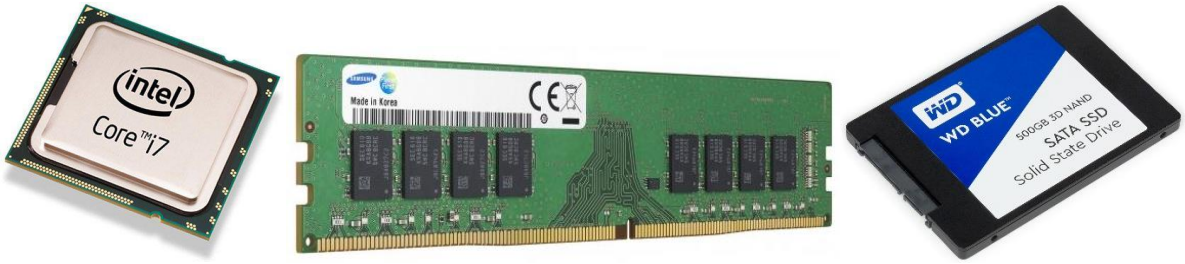
1. ***Вычислительный блок*** – это центральный процессор (Intel Pentium, Core i7, AMD Athlon, Ryzen и т.д.) (рис. 1, *а*). Именно он обеспечивает работу компьютера, то есть исполнение программ. Можно сказать, что компьютер без процессора – не более чем «груда железа».

2. ***Память*** – это хранилище данных. Компьютер занимается обработкой информации, он не создает эту информацию из ничего. Данные поступают на вход компьютера, выполняются некоторые преобразования, и что-то получается на выходе. Все эти данные нужно где-то хранить, и хранятся они в памяти.

Выделяют *два вида памяти*:

1. Энергозависимая память (оперативная память, оперативное запоминающее устройство, ОЗУ) (рис. 1, *б*). Хранит данные только при подаче электропитания. Когда говорят, что в компьютере 4 или 8 гигабайт памяти, как правило, речь ведут именно об ОЗУ.

2. Энергонезависимая память (внешняя память, внешнее запоминающее устройство, ВЗУ) (рис. 1в). Хранит данные вне зависимости от подачи электропитания. К устройствам памяти данного вида относятся жесткие диски (HDD) и твердотельные накопители (SSD), но также к ним можно отнести различные флешки, включая карты памяти, компакт-диски и т.п.



а) б) в)

Рис. 1. Комплектующие компьютера: *а* – процессор Intel Core i7;
б – модуль памяти DDR4; *в* – SSD-накопитель
Источник: https://www.ixbt.com

ОЗУ – быстрая память, операции с ней (чтение и запись) происходят даже не в разы, а на порядки быстрее, чем с ВЗУ. Можно сказать, что оперативная память предназначена для текущей, оперативной работы с данными (тут важна скорость), а внешняя – для их долгосрочного хранения (тут больше важна емкость, объем дискового пространства).

При выполнении программы сам ее код и данные, с которыми она работает, находятся в оперативной памяти. Например, необходимо поместить в память

число 5, чтобы затем выполнить с ним некоторые математические манипуляции. На языке Python эта операция выглядит так:

```
a = 5
```

В результате ее выполнения в оперативной памяти создастся ячейка под названием `a`, в которой будет храниться число 5, и храниться оно там будет до тех пор, пока память не будет очищена (пока программа выполняется).

Если нужно хранить значение переменной `a` в течение длительного времени, его необходимо записать во внешнюю энергонезависимую память – в файл на диске (например, текстовый).

Высокоуровневые языки

Все данные в компьютере хранятся в виде чисел. Компьютер – это вычислительная машина, очень продвинутый калькулятор. Все, что он умеет, – производить математические операции над какими-то числами и считывать (записывать) данные из памяти (в память). Как уже было отмечено, занимается этим центральный процессор.

Строго говоря, только выполнять операции с данными процессор и умеет, и компьютерные вычисления именно так и реализуются. Например, *очень упрощенно и условно* операция сложения чисел A и B выглядит так: в каждом разряде ячейки B устанавливается состояние, соответствующее сумме цифр слагаемых, набранных первоначально в A и B в этом разряде, и переходной единицы из предыдущего разряда, если она есть; в результате в ячейке B оказывается сумма чисел [21].

Любые действия, осуществляемые компьютером, в конечном счете состоят из подобных элементарных операций. Этих операций *очень* много. Чтобы получить результат любого математического выражения, нужно перебрать и обработать множество ячеек с данными.

Для программирования первых компьютеров нужно было указывать каждую такую элементарную операцию, то есть отдавать команду компьютеру напрямую: скопируй данные из этой ячейки памяти в другую ячейку, с той ячейкой сделай что-нибудь еще и т.д. Программы состояли из последовательностей отверстий на *перфокартах* (старинных носителях данных) (рис. 2).

Конечно, работа программиста была тяжела и трудоемка. Описанное программирование является ***низкоуровневым***. Под низким уровнем подразумевается, что программист практически напрямую отдает команды процессору, общается с ним на очень низком уровне, в так называемых *машинных кодах*.

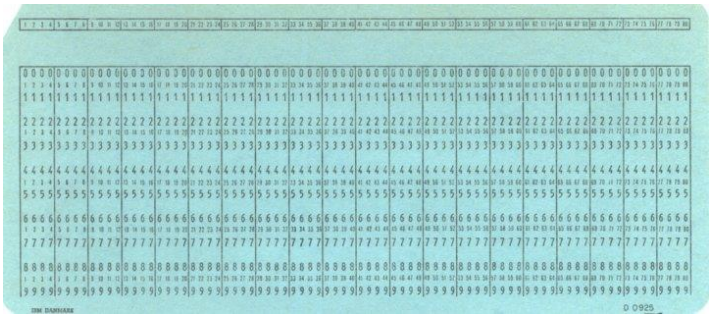


Рис. 2. Перфокарта формата IBM
Источник: https://ru.wikipedia.org

Затем появился *язык ассемблера* – язык тоже низкоуровневый, но все-таки более высокого уровня, чем машинные коды. У разных моделей процессоров были собственные отличающиеся системы команд, и соответственно, под каждый процессор надо было писать специфические для него программы. Выглядит программа на ассемблере так:

```
org 0x100
mov ah, 0x9
mov dx, hello
int 0x21
```

Принцип программирования на ассемблере все тот же: программа содержит команды процессору, предписывающие ему осуществлять некоторые операции с теми или иными данными в ячейках памяти. На ассемблерах пишут и сегодня (на момент издания данного пособия), это требуется в некоторых специфических задачах.

Шло время, появились *унифицированные системы команд* для разных процессоров. В частности, все современные процессоры производства компаний Intel и AMD понимают систему команд x86 и ее расширение x86_64. Уровень абстракции (то есть уровень, на котором происходит общение программиста с компьютером) вырастал, а требования к квалификации программистов падали. Появились *языки программирования высокого уровня*.

Программист, пишущий на таком языке, оперирует не ячейками памяти, с которыми надо что-то сделать, а более абстрактными структурами и конструкциями данных. Язык высокого уровня похож на естественный язык – в особенности для англоговорящих программистов. Впрочем, вот пример кода на языке программирования 1С 8:

```
Переменная = Новый Массив;
Переменная.Добавить(22); // в ячейку 1 записать значение 22
Переменная.Добавить(33); // в ячейку 2 записать значение 33
Сообщить(Переменная[0]); // вывести ячейки 1
```

Но процессор все равно понимает только свои машинные коды. «Высокоуровневость» языка заключается в том, что между непосредственно оборудованием (процессором) и программистом, пишущим код, есть специальные промежуточные слои, программные прослойки, которые берут программный код на языке высокого уровня, понятный человеку, и преобразуют его в команды, понятные процессору. Таким «переводчиком с программистского языка на процессорный» является ***транслятор языка программирования***.

Транслятор бывает *двух видов* (рис. 3).

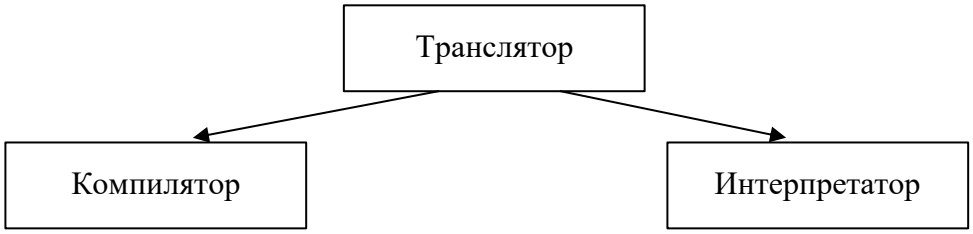


Рис. 3. Виды трансляторов

Компилятор переводит целиком весь код программы в машинные коды, на выходе получается исполняемый файл, который можно запускать на выполнение. *Интерпретатор* последовательно транслирует в машинный код каждую команду программы и сразу выполняет ее. Недостаток интерпретатора: он более медленный, потому что приходится транслировать каждую команду отдельно, а не всю программу разом, как это делает компилятор. Python является *интерпретируемым языком*.

Тенденции в сфере информационных технологий таковы, что уровень абстракции языков растет и программировать становится все проще: от программиста уже не требуется разбираться в тонкостях работы компьютера, порог вхождения в профессию падает. Существуют даже визуальные языки программирования, не требующие непосредственного кодирования. Программа на таком языке представляет собой набор графических объектов, составляемый при помощи мыши (рис. 4).

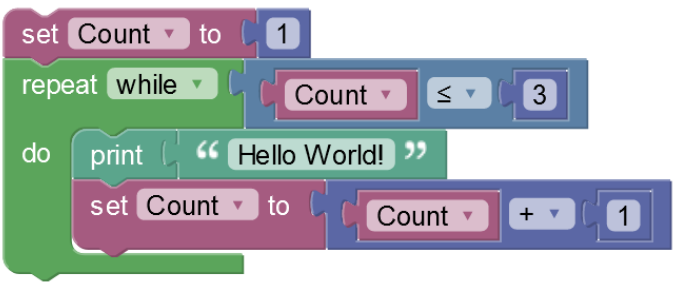


Рис. 4. Пример фрагмента программы на языке Google Blockly
Источник: https://developers.google.com

Ознакомившись с понятием высокоуровневого языка, перейдем к языку программирования Python.

1.2. Язык Python

Общие сведения о языке

Python – универсальный язык программирования высокого уровня. Несмотря на то, что на логотипе языка изображены две змейки, назван он не в честь этих представителей животного царства, а в честь британского телешоу «Летающий цирк Монти Пайтона» (рис. 5).

а) б)

Рис. 5. Происхождение названия языка Python: *а* – логотип Python; *б* – кадр из скетча Монти Пайтон «Мертвый попугай»
Источник: https://ru.wikipedia.org

Одна из целей создателей языка – сделать его «забавным для использования». И действительно, Python существенно отличается от большинства других языков своей простотой, изяществом кода и оригинальностью синтаксиса.

По версии IEEE[1] Spectrum Python уже несколько лет является *самым популярным языком программирования в мире* (по данным на 2023 г. (рис. 6).

Python создан в 1991 году. Создателем языка является *Гвидо ван Россум*, голландский программист, лауреат премии Free Software Award 2001 г. за вклад в развитие свободного программного обеспечения.

Python пережил несколько поколений архитектурных изменений и по сей день продолжает быть актуальным. На практике можно встретить программы, написанные на языке двух версий: версии 2 и версии 3. На данный момент актуальна и активно развивается именно *версия 3*. Текущий релиз – 3.12.1 (по состоянию на конец декабря 2023 г.). Последней официально поддерживаемой версией второго поколения Python была 2.7.18.

[1] IEEE (институт инженеров электротехники и электроники) – некоммерческая инженерная ассоциация, разрабатывающая широко применяемые в мире стандарты по радиоэлектронике, электротехнике и аппаратному обеспечению вычислительных систем и сетей.

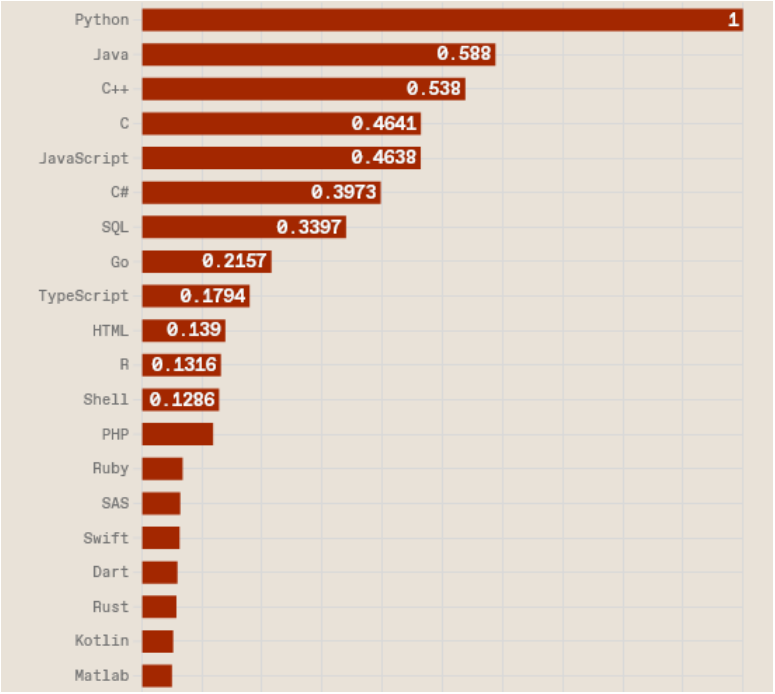


Рис. 6. Рейтинг популярности языков программирования в 2023 г.
Источник: [16]

Следует разделять понятия *язык Python* и *интерпретатор Python*. Язык программирования – это набор правил и ключевых слов, с помощью которых записываются программы, интерпретатор же – это компьютерная программа, которая исполняет программный код, записанный на языке. Когда говорят, что была выпущена новая версия Python, имеют в виду как саму версию языка (он постоянно обновляется, дорабатывается), так и интерпретатор, предназначенный для этой версии. Например, интерпретатор Python 3.9 предполагает, что исполняемые им программы написаны на языке версии 3.9, и при интерпретации ведет себя так, как предписано спецификацией данной версии Python. Программы, написанные на Python 2, не будут корректно работать, если попытаться их запустить с помощью интерпретатора Python 3 (скорее всего, они не запустятся вовсе из-за несовпадения синтаксиса). Программы же, написанные на Python 3.5, будут работать с интерпретатором Python 3.10 (обратное не гарантируется). Новые версии языка (в рамках одной мажорной версии, которой является третья) сохраняют совместимость со старыми версиями, просто новые обладают рядом возможностей, которых не было в старых.

Перечислим некоторые важные *особенности Python*:

✓ является изначально объектно-ориентированным (каждая переменная является ссылкой на объект);

✓ является интерпретируемым (эталонная реализация интерпретатора – CPython, но существуют и другие);

✓ характеризуется строгой динамической типизацией (ее еще называют «утиной» по принципу «утиного теста»: если что-то выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка);

✓ имеет автоматическое управление памятью (сборку мусора);

✓ характеризуется выделением блоков кода пробельными отступами.

К *преимуществам* Python относятся следующие его качества:

✓ универсальность (на Python можно написать программы для решения практически любой задачи);

✓ кроссплатформенность (существуют интерпретаторы Python для различных программных и аппаратных платформ: персональных компьютеров, смартфонов, сетевых устройств и т.д.);

✓ огромное сообщество разработчиков;

✓ широкий выбор готовых библиотек (в том числе для научных расчетов, анализа данных, машинного обучения и т.д.);

✓ минималистичность и простота программ;

✓ пологая кривая обучения (чтобы суметь решить задачу, имеющую практическое значение, достаточно освоить азы языка, что можно сделать в короткие сроки; по мере углубления знаний сложность и полезность создаваемых программ могут возрастать);

✓ язык способствует написанию красивого и понятного кода (это обусловлено как самим синтаксисом Python, так и правилами программирования на нем, описанными в Дзене и PEP8, – об этом ниже).

Основным недостатком Python, с точки зрения автора, является то, что он *интерпретируемый*, а следовательно, *медленный* (в сравнении с компилируемыми языками). Однако Python использует скомпилированные функции и объекты, написанные на языке C, а также промежуточный байт-код, который создается на основе исходных Python-скриптов. Эти факторы позволяют несколько нивелировать данный недостаток и повысить производительность исполнения программ. Не стоит забывать, что разработчиками активно ведутся работы по оптимизации и ускорению интерпретатора.

Дзен Python

Принципы языка Python, его философия сформулированы в тексте под названием ***The Zen of Python***. Приведем его перевод на русский язык [19]:

1. Красивое лучше, чем уродливое.
2. Явное лучше, чем неявное.
3. Простое лучше, чем сложное.

4. Сложное лучше, чем запутанное.
5. Плоское лучше, чем вложенное.
6. Разреженное лучше, чем плотное.
7. Читаемость имеет значение.
8. Особые случаи не настолько особые, чтобы нарушать правила.
9. При этом практичность важнее безупречности.
10. Ошибки никогда не должны замалчиваться.
11. Если они не замалчиваются явно.
12. Встретив двусмысленность, отбрось искушение угадать.
13. Должен существовать один и желательно только один очевидный способ сделать это.
14. Хотя он поначалу может быть и не очевиден, если вы не голландец.
15. Сейчас лучше, чем никогда.
16. Хотя никогда зачастую лучше, чем прямо сейчас.
17. Если реализацию сложно объяснить – идея плоха.
18. Если реализацию легко объяснить – идея, возможно, хороша.
19. Пространства имен – отличная штука! Будем делать их больше!

Конечно, многие из пунктов Дзена могут быть непонятны начинающему программисту, однако по мере изучения языка все становится на свои места. И чтобы стать хорошим «питонистом» (программистом на Python), следует этих принципов придерживаться.

Стандарт PEP8

Важнейшей рекомендацией по стилевому оформлению программ на языке Python является *стандарт PEP8*: он описывает правила именования и структурирования кода, принятые в сообществе Python-разработчиков.

Приведем некоторые правила PEP8:

✓ для выделения блоков кода следует использовать отступы из четырех пробелов;

✓ рекомендуется использовать строки, длина которых не превышает диапазон 80–120 символов (первые мониторы были ограничены 80 символами в ширину);

✓ функции верхнего уровня и определения классов следует отделять от основного кода программы двумя пустыми строками;

✓ следует использовать пустые строки для логического разделения программы на блоки;

✓ следует использовать одиночные пробелы с каждой стороны у операторов присваивания, сравнения и логических операторов.

Подробнее о стандарте PEP8 и его требованиях к коду можно прочитать в следующих источниках: документ на английском языке [13], выжимка на русском языке [14].

Приступая к программированию на Python

Чтобы написать и выполнить программу на Python, необходимы следующие программные компоненты, установленные на компьютере:

1. *Редактор программного кода.* Это текстовый редактор, предоставляющий возможности для удобного написания программ на языке программирования (в нашем случае Python).
2. *Транслятор языка.* Это программа-переводчик, «транслирующая» (переводящая) программу с высокоуровневого языка программирования в низкоуровневый язык, понятный процессору. Как мы уже выяснили, в случае Python используется следующая разновидность транслятора: интерпретатор.

Интерпретатор Python можно скачать по ссылке [1].

Удобной средой разработки, имеющей в составе редактор кода, является PyCharm (ссылка для скачивания: [23]). PyCharm – инструмент, предназначенный для опытных Python-разработчиков, эта среда предоставляет программисту широкие возможности, включая отладчик и инструменты версионирования программ, однако начинающему установить ее и разобраться, как в ней работать, не так просто.

Мы будем писать код в онлайн-среде разработки Google Colab. Знакомству с данной платформой посвящена вторая глава учебного пособия.

Стоит отметить, что интерпретатор Python (программа python.exe) имеет так называемый *интерактивный режим работы*, который позволяет писать и выполнять команды языка в командной строке по одной. В этом режиме длинные, сложные программы не пишут, однако его удобно использовать как инструмент решения некоторых задач (например, в качестве продвинутого калькулятора) (рис. 7).

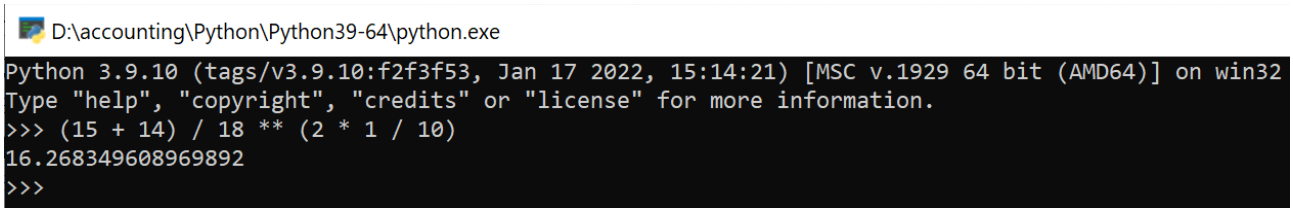


Рис. 7. Интерпретатор Python в интерактивном режиме

Прежде чем приступить к изучению основ Python, познакомимся со средой разработки Google Colab. Это необходимо, чтобы выполнять код из примеров и практиковаться в программировании.

1.3. Задания для самостоятельной работы

Ответьте своими словами на следующие вопросы:

1. Приведите по два-три примера алгоритма для: исполнителя-человека; исполнителя-компьютера.
2. Какими свойствами характеризуется алгоритм?
3. Что такое язык программирования?
4. Что такое компьютерная программа?
5. Каковы функции блока компьютера, который называется «центральный процессор»?
6. В чем различие оперативной и внешней памяти?
7. В чем отличие интерпретатора от компилятора?
8. Каковы преимущества языка Python, обеспечившие его широкую популярность?
9. Что такое «Дзен Python» и стандарт PEP8? Являются ли они наборами строгих предписаний или только рекомендациями программисту?
10. Какие программные компоненты являются минимально необходимыми для того, чтобы написать и выполнить компьютерную программу?

2. Среда разработки Google Colaboratory

Вторая глава целиком посвящена основам работы в облачной среде Google Colab. Она содержит материал, знание которого необходимо для выполнения всех приведенных примеров программного кода.

2.1. Общие сведения о платформе

Чтобы программировать на Python, используют набор программных инструментов, который носит название ***интегрированной среды разработки*** (**I**ntegrated **D**evelopment **E**nvironment, **IDE**). Таких сред разработки для языка Python существует достаточно большое количество. Рассмотрим одну из них: Google Colaboratory. Эта IDE отличается простотой настройки и удобством использования, чем обусловлена ее популярность в среде Python-программистов, аналитиков данных и специалистов по машинному обучению.

Google Colaboratory (широко используется сокращение *Colab*) – облачная платформа для разработки на языке Python от компании – технологического гиганта Google. К ее *преимуществам* можно отнести:

- ✓ бесплатность использования;
- ✓ отсутствие необходимости сложной и длительной настройки программного обеспечения на компьютере: для работы нужен лишь веб-браузер одной из последних версий (например, Google Chrome, Mozilla Firefox, Яндекс.Браузер) и доступ к интернету;
- ✓ большое количество предустановленных программных библиотек и простота установки новых;
- ✓ возможность предоставлять доступ к написанным программам другим интернет-пользователям.

2.2. Основы работы в Colab

Для работы в Colaboratory необходимо пройти по ссылке https://colab.research.google.com/?hl=ru. Эту же ссылку легко получить, сделав запрос *google colab* в любой поисковой системе (Google, Яндекс и т.д.). Откроется приветственная страница (рис. 8).

Для работы в Colab требуется наличие аккаунта Google (почтового ящика в домене @gmail.com). Нажмем кнопку «Войти» и увидим стандартную страницу входа (рис. 9).

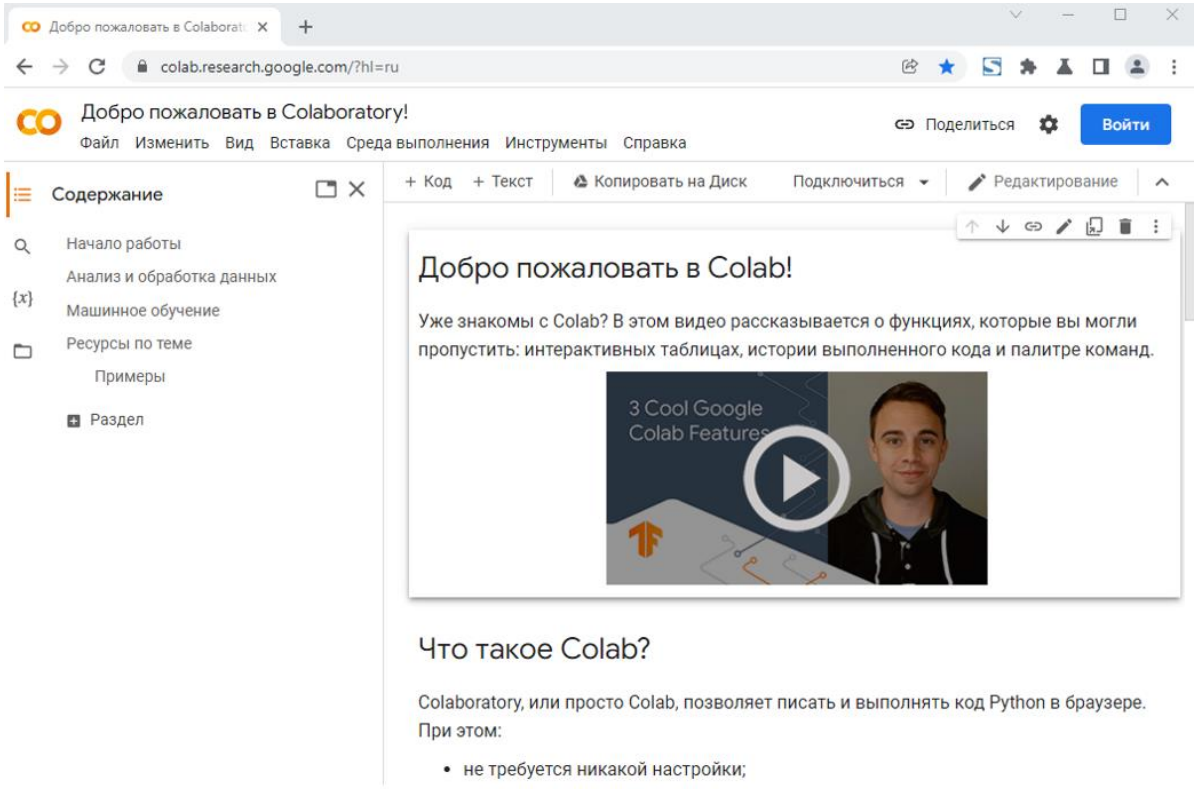


Рис. 8. Приветственная страница Google Colab

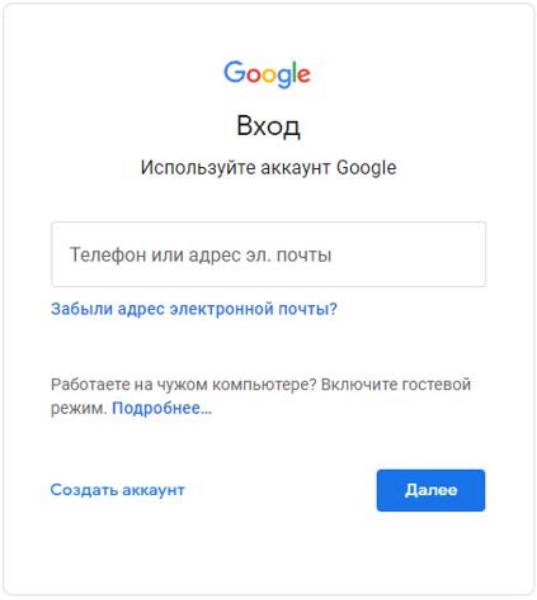


Рис. 9. Страница аутентификации Google

Следует ввести свои учетные данные либо зарегистрировать новый аккаунт, нажав на кнопку «Создать аккаунт». После аутентификации создадим новый файл, в котором будем писать наши программы на Python, – он называется ***блокнот*** (рис. 10).

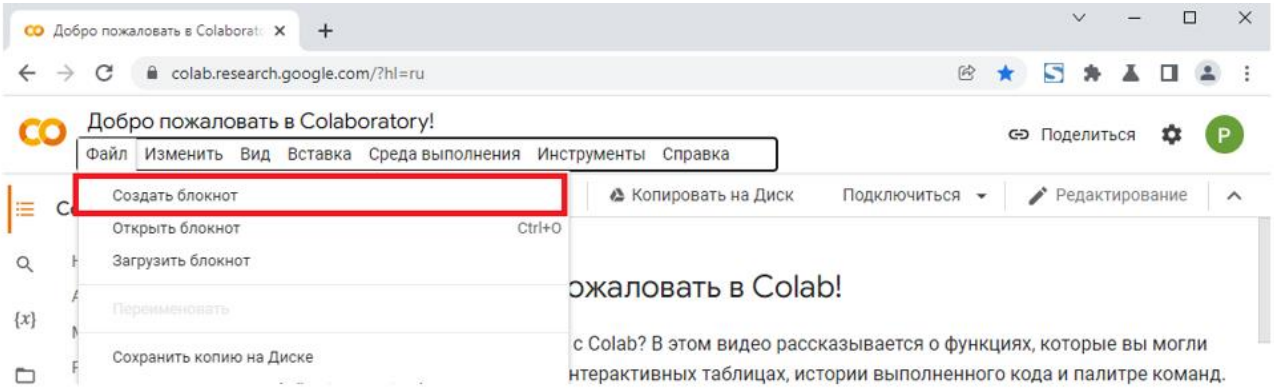


Рис. 10. Создание нового блокнота Colab

После этого откроется основное окно среды разработки Colab (рис. 11).

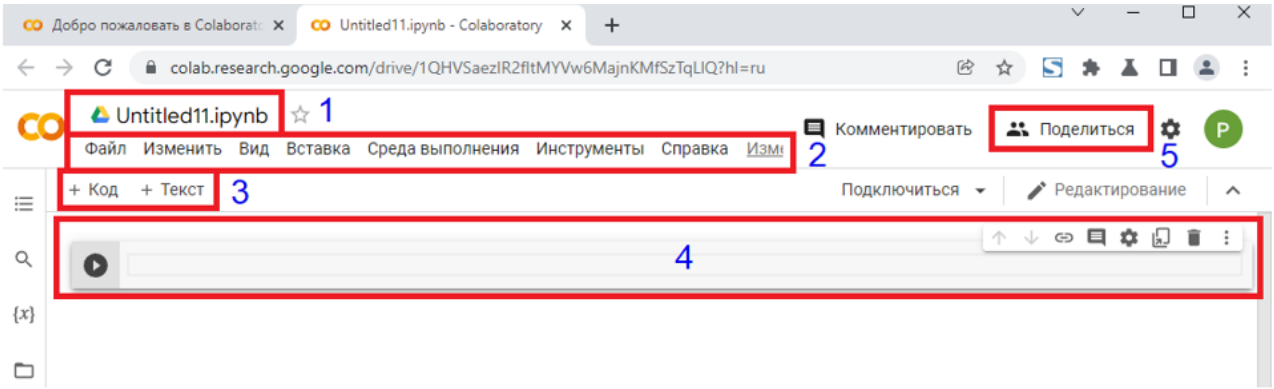


Рис. 11. Интерфейс блокнота

Изучим элементы интерфейса блокнота.

1. *Название блокнота.* Блокнот – это файл с расширением *.ipynb*, который содержит код нашей программы и результаты его выполнения. Для удобства его можно переименовать.

2. *Основное меню.* Пункт «Файл» позволяет создавать блокноты, открывать ранее созданные, скачивать их на локальный компьютер и т.п. Пункт «Среда выполнения» позволяет управлять выполнением ячеек с программным кодом.

3. *Кнопки добавления ячеек.* Можно добавлять ячейки как с кодом, так и с текстом (подробнее о типах ячеек ниже).

4. *Основное рабочее поле с ячейками.* Именно тут производится наполнение блокнота содержимым – программным кодом или форматированным текстом.

5. *Кнопка «Поделиться».* Она позволяет предоставлять доступ к блокноту другим пользователям.

Следует отметить особенность IDE Colab, которая идет вразрез с концепцией модулей программного кода, традиционной для программирования. Дело в

том, что блокноты Colab состоят не только из программного кода. Обычный файл с исходным кодом (модуль) выполняется при запуске целиком от первой до последней строки. Блокнот же состоит из ячеек двух видов: с кодом и с форматированным текстом. *Ячейки с кодом* являются аналогами модулей и выполняются целиком, от начала до конца, при этом их в блокноте может быть несколько. *Ячейки с текстом* позволяют вставлять в блокнот собственно текст, тем или иным образом отформатированный, но не только: они могут включать также математические формулы (используется язык TeX) и изображения. Эта особенность позволяет использовать Colab не только как удобное средство разработки программ на Python, но и как инструмент создания презентаций и прочих учебно-методических материалов. Интеграция с программным кодом и возможность его выполнения здесь и сейчас раздвигает возможности преподавания и вкладывает новые смыслы в понятие «интерактивный учебник».

Напишем первую простейшую программу на Python. При создании нового блокнота одна ячейка с кодом в нем уже содержится. Также можно добавить новую ячейку, нажав на кнопку «+ Код». По принятой у программистов традиции первая написанная программа выводит на экран фразу «Привет, мир!». Напишем такую программу:

```
print("Привет, мир!")
```

Текст программы (ее исходный код) написан. Теперь следует его выполнить, для чего нажмем на кнопку с треугольником в кружке, которая появляется в левой части ячейки при наведении на нее указателя мыши. В результате ячейка будет запущена на выполнение – это означает, что написанный в ней код выполнится целиком, с первой до последней строки (так же, как выполняется код в традиционных модулях). Результат выполнения ячейки приведен на рис. 12.

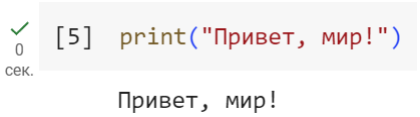


Рис. 12. Ячейка с кодом и результат ее выполнения

Зеленая галочка слева означает, что операция успешно выполнена. В области под ячейкой можно увидеть вывод программы на экран – в нашем случае это строка приветствия миру.

Опишем еще одну особенность выполнения программ в среде Colab, обусловленную структурой блокнотов, состоящей из множества ячеек с кодом. В отличие от традиционных IDE, где данные, с которыми взаимодействует про-

грамма, размещаются в оперативной памяти и хранятся в ней только до окончания выполнения программы (после чего память очищается), блокнот имеет состояние, которое хранится и изменяется в течение всего срока работы с ним (пока он не будет закрыт). Выполнение ячеек с кодом обновляет это состояние, при этом оно хранится даже в моменты, когда никакой код фактически не выполняется.

Продемонстрируем описанную особенность на примере. Для этого добавим еще четыре ячейки, в каждую из которых поместим одну из строк следующего кода (на содержимое кода внимание пока можно не обращать).

```
newlist = [1, 2, 3]  # создание списка из трех целых чисел
newlist[0] = 10  # перезапись первого элемента другим числом
newlist[1] = 20  # перезапись второго элемента другим числом
newlist[2] = 30  # перезапись третьего элемента другим числом
```

Также для демонстрации в конце каждой ячейки поместим вывод значения объекта `newlist` на экран. (Со списками и другими сложными типами данных мы познакомимся в дальнейшей части пособия, а пока что разбираемся с особенностями Colab.)

```
newlist = [1, 2, 3]  # создание списка из трех целых чисел      1
print(newlist)
```

```
newlist[0] = 10  # перезапись первого элемента другим числом    2
print(newlist)
```

```
newlist[1] = 20  # перезапись второго элемента другим числом    3
print(newlist)
```

```
newlist[2] = 30  # перезапись третьего элемента другим числом   4
print(newlist)
```

Выполним указанные ячейки в следующем порядке: 1, 3, 4, 2 (рис. 13).

Как можно заметить, список, созданный в ячейке 1, был помещен в память (стал частью состояния блокнота), и последующие ячейки последовательно изменяли его содержимое, причем в том порядке, в каком они были выполнены. Свой итоговый вид [10, 20, 30] список принял после выполнения ячейки 2, потому что оно было осуществлено последним. Порядок выполнения ячеек с кодом может быть совершенно произвольным. Для большей наглядности изобразим эту схему с помощью следующей иллюстрации (рис. 14).

```
[1]  newlist = [1, 2, 3]  # создание списка из трех целых чисел      1
     print(newlist)

     [1, 2, 3]

[4]  newlist[0] = 10  # перезапись первого элемента другим числом    2
     print(newlist)

     [10, 20, 30]

[2]  newlist[1] = 20  # перезапись второго элемента другим числом    3
     print(newlist)

     [1, 20, 3]

[3]  newlist[2] = 30  # перезапись третьего элемента другим числом   4
     print(newlist)

     [1, 20, 30]
```

Рис. 13. Результат выполнения ячеек в произвольном порядке

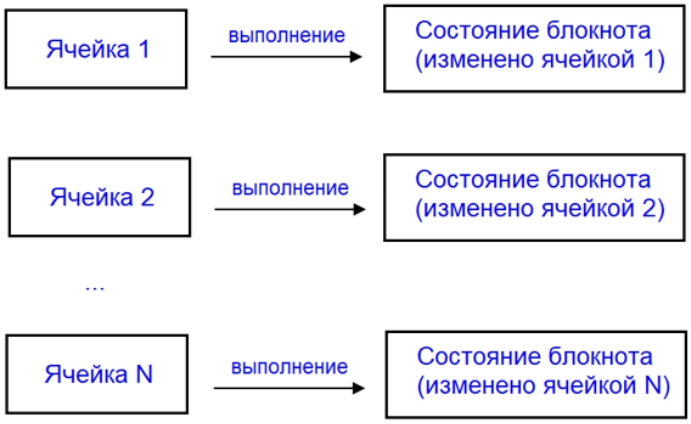


Рис. 14. Изменение состояния блокнота ячейками

Состояние блокнота (то есть значения переменных, используемых в программе) в каждый момент времени определяется результатами выполнения последней вызванной ячейки с кодом. Можно заметить, что после успешного выполнения ячейки справа от зеленой галочки появляется число в квадратных скобках. Это порядковый номер вызова ячейки. Первая выполненная в блокноте ячейка получает номер 1, вторая – 2, двести восемьдесят шестая – 286, и т.д. Эта нумерация позволяет определять порядок вызова ячеек и то, какая из них была выполнена последней (ячейка с максимальным номером) и тем самым определила текущее состояние блокнота. Очень важно понимать этот механизм, чтобы не столкнуться с неожиданными (но в действительности логичными и закономерными) результатами выполнения программы в Colab.

Также следует знать о том, что после закрытия блокнота вывод программ в ячейках сохраняется. Но это не относится к состоянию блокнота: в случае бездействия пользователя оно хранится некоторое время, после чего сбрасывается,

все объекты в памяти исчезают. Выглядит блокнот в исходном состоянии следующим образом (рис. 15).

```
[ ]  newlist = [1, 2, 3]  # создание списка из трех целых чисел      1
     print(newlist)

     [1, 2, 3]

[ ]  newlist[0] = 10  # перезапись первого элемента другим числом    2
     print(newlist)

     [10, 20, 30]

[ ]  newlist[1] = 20  # перезапись второго элемента другим числом    3
     print(newlist)

     [1, 20, 3]

[ ]  newlist[2] = 30  # перезапись третьего элемента другим числом   4
     print(newlist)

     [1, 20, 30]
```

Рис. 15. Блокнот в исходном состоянии

Как можно заметить, зеленые галки и нумерация слева от ячеек пропали. Сбросить состояние можно и вручную с помощью пункта меню «Перезапустить сеанс» (рис. 16).

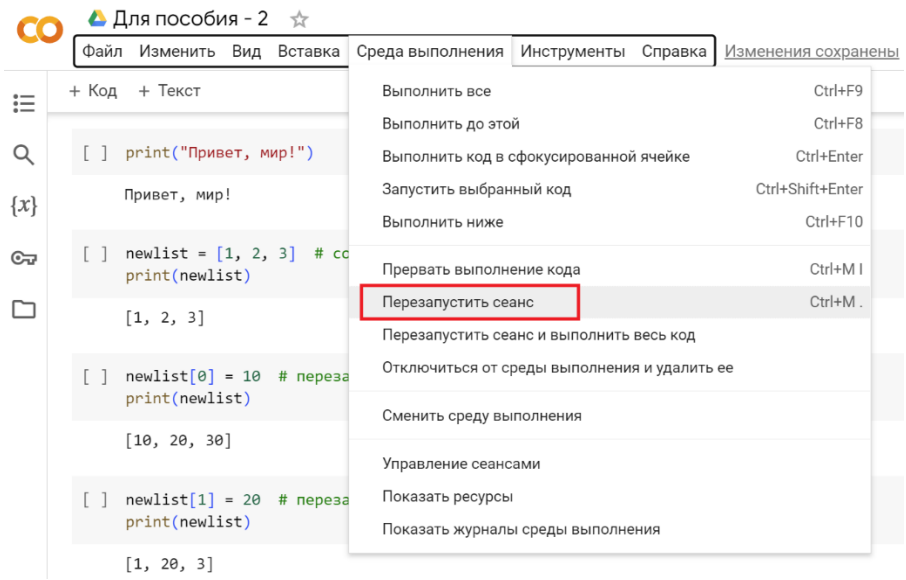


Рис. 16. Сброс состояния блокнота вручную

В правой части активной ячейки (той, в которой находится курсор и с которой происходит взаимодействие) отображается панель, позволяющая изменять позицию ячейки (двигать вверх-вниз) и удалять ячейку (рис. 17).

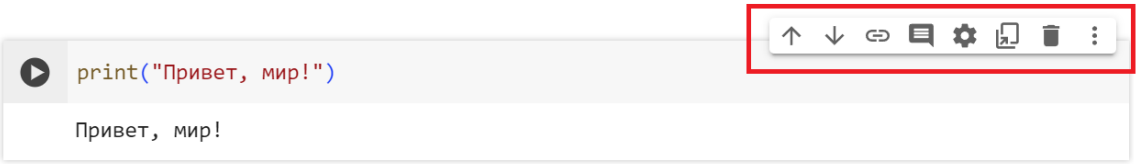


Рис. 17. Панель управления ячейкой

Установка сторонних библиотек

Библиотеки функций – это наборы уже кем-то написанного программного кода, позволяющего решать те или иные задачи. Программирование на 90 % состоит из использования готового кода, потому что абсолютное большинство решаемых программистом узких задач (таких как сортировка массива, вычисление коэффициента корреляции или построение модели линейной регрессии методом наименьших квадратов) уже было кем-то решено, причем, скорее всего, более качественно, чем это смог бы сделать программист, решая такую задачу своими силами впервые. Как уже было отмечено, Google Colab имеет большое количество предустановленных библиотек. Однако невозможно предустановить абсолютно все библиотеки (их очень много), поэтому в Colab предусмотрен механизм установки сторонних библиотек. Осуществляется установка путем выполнения в ячейке с кодом следующей команды:

```
!pip install <название библиотеки>
```

Попробуем установить библиотеку PyMorphy2, предназначенную для морфологического анализа текстов:

```
!pip install pymorphy2
```

Если после выполнения в ячейке появится зеленая галочка, значит, установка выполнилась успешно, теперь эту библиотеку можно подключить и использовать в своей программе.

Если затребованная библиотека уже установлена в Colab, система об этом напишет, и ничего не произойдет.

Стоит отметить, что команда `!pip` не является инструкцией языка Python. Pip – это специальная программа, которая позволяет управлять установкой библиотек (так называемый пакетный менеджер; пакет – это другое название библиотеки функций). Эта программа исполняется в облаке, то есть где-то на серверах Google. Чтобы вызывать такого рода системные программы, в Colab предусмотрен специальный механизм – написание команд с лидирующим символом знака восклицания (`!`). Соответственно, работает этот механизм только в среде

Google Colab. В других IDE попытка выполнить подобную инструкцию внутри программы на Python приведет к ошибке.

Также следует иметь в виду, что при очищении состояния блокнота установленные пользователем библиотеки *удаляются*. При следующем выполнении программы их необходимо будет установить повторно.

Предоставление доступа к блокноту

Colab позволяет делиться собственными блокнотами с другими пользователями. Настроить совместный доступ к блокноту позволяет кнопка «Поделиться», открывающая соответствующее меню (рис. 18).

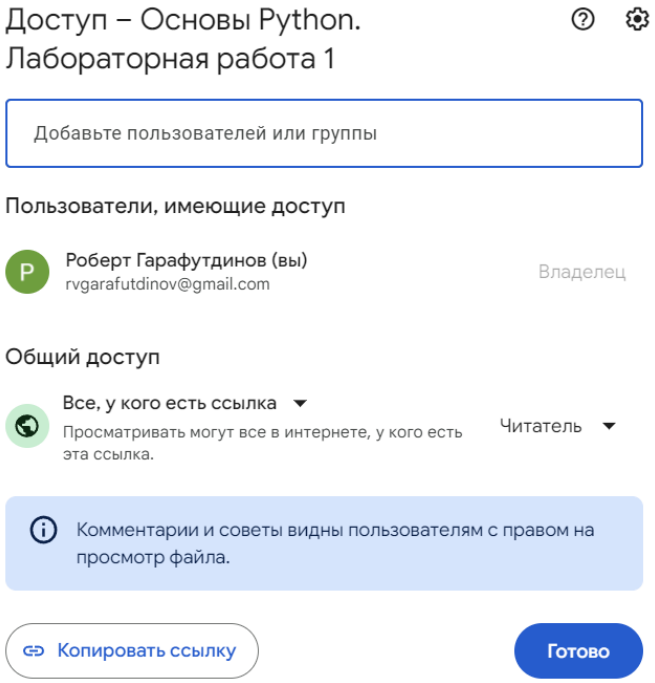


Рис. 18. Меню настройки совместного доступа

В меню настройки доступа можно либо предоставить доступ к блокноту конкретным пользователям (добавив их почтовые адреса в список), либо открыть доступ всем по ссылке. При этом можно выбрать одну из трех ролей: читатель, комментатор и редактор. *Читатель* может только просматривать оригинальный блокнот и создавать его копию, *комментатор* может оставлять комментарии, а *редактор* может вносить в блокнот изменения.

Чтобы открыть общий доступ к блокноту на чтение по ссылке, следует выбрать пункт «Все, у кого есть ссылка» и указать роль «Читатель». Сформированную ссылку можно скопировать и распространить. Любой пользователь, имеющий данную ссылку, сможет просмотреть блокнот, причем даже без входа в аккаунт Google. А войдя в аккаунт, он сможет создать свою собственную копию блокнота, выполнять и редактировать ее по своему усмотрению.

2.3. Задания для самостоятельной работы

Выполните следующие задания:

1. Войдите в Google Colab (при необходимости зарегистрируйте учетную запись Google). Создайте новый блокнот.

2. Создайте ячейку с кодом, напишите в ней указанную команду и выполните ее.

```
print("Привет, мир! Я изучаю Python!")
```

3. Создайте ячейку с текстом. Поэкспериментируйте со следующими элементами, вставляя их в ячейку: обычный текст с различным форматированием (полужирный, курсив), заголовки разного размера, маркированный и нумерованный списки, гиперссылка, изображение. Используйте палитру компонентов в верхней части ячейки.

4. Настройте общий доступ к блокноту по ссылке с правами на чтение. Убедитесь, что доступ открыт, пройдя по ссылке на блокнот с другого компьютера или телефона.

3. Синтаксис Python и основные конструкции программы

Данная глава посвящена первому практическому знакомству с языком Python, его базовым понятиям и написанию простейших программ.

3.1. Переменные

Данные, с которыми работает программа, хранятся в оперативной памяти. Память состоит из ячеек, которые имеют свои адреса. Если дать ячейке памяти какое-то имя и что-то в нее записать, появится ***переменная*** – поименованная область памяти, хранящая некоторое значение.

Создадим несколько переменных:

```
a = 15
perem = "Привет"
variable = [1, 2, 9]
```

После выполнения кода выше в оперативной памяти появятся поименованные ячейки с данными (рис. 19).

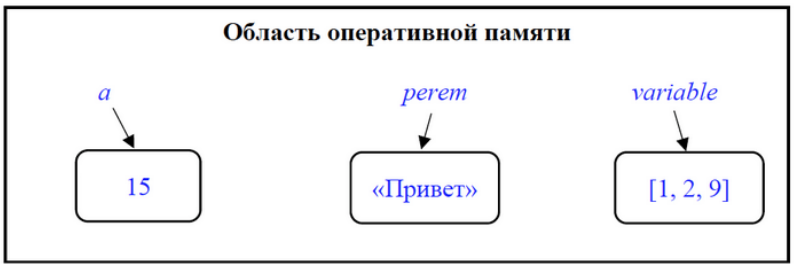


Рис. 19. Переменные в памяти

Можно провести аналогию с математическими переменными, содержащими некоторые числа. Переменная в программировании – это воплощение на практике данного абстрактного понятия из математики.

Имя переменной программист выбирает сам. Приведем несколько правил именования переменных в Python.

1. Имя переменной состоит из одного или нескольких слов на английском языке, причем эти слова должны информировать программиста о содержимом переменной. Примеры информативных имен: `count`, `address_of_office`, `date_of_birth`.
2. Имя должно начинаться с маленькой буквы.
3. Имя не может начинаться с цифры.
4. Не может быть именем *ключевое слово* Python: интерпретатор не сможет правильно обработать такой код, потому что будет считать, что в программе выполняется определенная стандартная команда, а не объявляется переменная.

Примеры ключевых слов: `continue`, `raise`, `pass`, `def`, `return`, `while`, `True`, `False`. Ключевые слова, как правило, выделяются средой разработки цветом.

5. Регистр символов в имени имеет значение (`data` и `dATa` – разные переменные).

В различных языках программирования существуют договоренности о стиле именования переменных в случае, когда имя состоит из нескольких слов. В С-подобных языках (C++, C#, Java и некоторых других) принята так называемая верблюжья нотация (camel case): первое слово пишется с маленькой буквы, а все последующие – с большой. Пример: `newVariableInJava`. В Python правилом хорошего тона является использование «змеиной нотации»: каждое слово в имени переменной пишется с маленькой буквы, слова разделяются подчеркиванием. Пример: `new_cool_variable`.

3.2. Оператор присваивания

Оператор языка программирования – это специальный символ или ключевое слово, сообщающие транслятору о необходимости выполнения той или иной операции с данными.

Фундаментальной и важнейшей операцией в программировании является ***операция присваивания***. Она позволяет помещать те или иные значения (данные) в переменные. На языке Python присваивание записывается так:

```
<имя переменной> = <выражение>
```

Выражением в программировании называется совокупность переменных, литералов, знаков операций, имен функций, скобок, которая может быть вычислена в соответствии с синтаксисом языка программирования. Результатом вычисления выражения является ***значение*** – данные определенного типа. Другое название значения (данных) – ***объект***.

Литералом называется запись в исходном коде программы, представляющая собой фиксированное значение, которое можно ввести с клавиатуры.

Приведем несколько примеров использования оператора присваивания:

```
a = 5  # в переменную a помещается число 5
b = (a + 10) / a  # в переменную b помещается значение выражения (a+10)/a
```

Если с первой командой все понятно, то вторая выполняется в два этапа: сначала вычисляется значение выражения в правой части `(a+10)/a`, а затем оно помещается в левую часть (переменную `b`).

Примеры выражений (записей, которые могут находиться в правой части оператора присваивания):

```
3  # выражение состоит из одного числового значения 3, заданного литералом
(a + 10) / (b - a)  # арифметическое выражение
round(a)  # выражение, состоящее из вызова функции
(a + b) ** 2 + 1 / abs(b)  # выражение, включающее переменные, литералы,
# знаки арифметических операций и вызов функции
```

При выполнении программы выражение выполняется в соответствии с приоритетами операций, из которых оно состоит, и «превращается» в некоторое результирующее значение того или иного типа (объект).

Множественное присваивание

В Python существует механизм присваивания одного значения нескольким переменным сразу. Выглядит соответствующая конструкция так:

```
<имя переменной 1> = <имя переменной 2> = ... = <выражение>
```

Например, нужно создать несколько переменных со значением 0. Код:

```
x = y = z = 0
```

В результате в памяти появятся три переменные: `x`, `y`, `z`, каждая равна 0.

3.3. Комментарии

Комментарии – это текстовые записи в программе, которые частью программы не являются и игнорируются транслятором при ее выполнении. Они поясняют те или иные фрагменты программного кода и нужны для того, чтобы пользователь, читающий программу, разобрался, что она делает.

Однострочный комментарий обозначается символом `#`. Весь текст, расположенный после `#`, считается комментарием.

```
a = 1  # отступ - не менее 2 пробелов
```

Многострочные комментарии заключены в тройные кавычки (двойные или одинарные): `"""..."""`. Следует отметить, что многострочных комментариев как отдельного элемента синтаксиса в Python не предусмотрено. С точки зрения интерпретатора строка в тройных кавычках является *строковым объектом* (значением, данными). Ее можно поместить в переменную или как-то иначе обработать. А можно использовать и в качестве комментария.

```
""" Эту строку можно
использовать как
многострочный ком-
ментарий """
```

Помимо поясняющей функции комментарии имеют функцию отладочную: с их помощью можно «выключать» определенные фрагменты кода. Это может быть полезно, когда написанный код не хочется удалять совсем, однако в данный момент он в программе не требуется.

3.4. Типы данных

Каждое значение в памяти компьютера имеет строго определенный тип. От типа данных зависит, как именно организовано хранение данных памяти и какие операции с ними можно совершать.

Условно можно выделить следующие типы данных в программировании:

- ✓ *простые* (элементарные, скалярные);
- ✓ *сложные* (структурированные).

К *простым* типам данных будем относить следующие:

- ✓ число;
- ✓ строка;
- ✓ логическое (булево) значение.

Числа могут быть целыми (`int`), вещественными (`float`) и комплексными (`complex`). Последние рассматривать в данном пособии не будем.

```
10  # целое положительное число, заданное литералом
-7  # целое отрицательное число, заданное литералом
10.75  # вещественное число, заданное литералом
3+05j  # комплексное число, заданное литералом
```

Строка (`str`) – это последовательность символов[2]. Строковый литерал задается с помощью кавычек (двойных или одинарных).

```
"Это значение строкового типа"  # строка, заданная литералом
```

[2] В действительности строковый тип относится к структурированным типам, потому что он не является элементарным (это последовательность отдельных элементов – символов). Однако, по мнению автора, на начальном этапе изучения Python не следует слишком погружаться в нюансы, а строковый тип данных – один из первых, с которым приходится столкнуться при написании программ. Ввиду этого пока будем считать его «простым».

Комбинирование кавычек разного типа позволяет включать символ кавычки в само содержимое строки. В этом случае включаемые кавычки должны быть вида, отличного от вида внешних кавычек, задающих строковый литерал.

```
"Привет, 'мир'!"  # использование внутри строки одинарных кавычек
'Привет, "мир"!'  # использование внутри строки двойных кавычек
```

Булево (логическое) значение (`bool`) – это специальный тип данных, позволяющий использовать в программировании аппарат математической логики. Булево значение определяет *истинность или ложность высказывания* и бывает двух видов: истина (`True`) и ложь (`False`). Напомним, что регистр символов в Python имеет значение: `True` является ключевым словом, а `true` – нет.

```
a = True  # задание булева значения "Истина" литералом
b = False  # задание булева значения "Ложь" литералом
```

Существует еще один элементарный тип данных – `NoneType` (пустой тип). Значение этого типа может быть только одним: `None`. `None` означает, что какой-то объект в памяти есть, но он не содержит никаких данных. Например, можно создать переменную, выделить под нее ячейку памяти, но при этом никаких осмысленных данных туда не записывать. В результате мы укажем, что переменная существует, но является пустой. Вновь не забудем о важности регистра (`none` не литерал).

```
a = None  # создание пустой переменной
```

Существует способ узнать тип того или иного значения. Для этого служит функция `type()` (рис. 20). О функциях – чуть ниже.

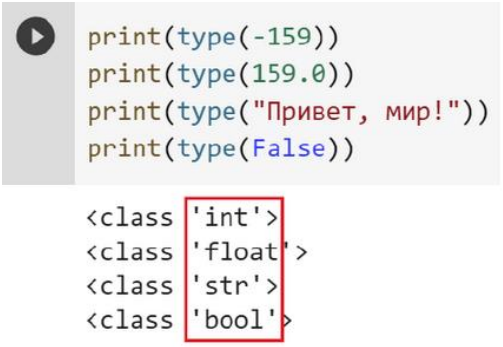


Рис. 20. Вывод типов данных нескольких значений

Преобразование типов

В процессе программирования часто приходится выполнять операции над значениями разных типов (например, складывать строку и число). Как упоминалось выше, от типа данных зависит, какие операции можно выполнять с этими данными. Строки можно складывать со строками, числа – с числами, но строки с числами – нельзя. Однако иногда это необходимо, и вскоре мы столкнемся с такими ситуациями. Ввиду этого требуется механизм преобразования данных одного типа в данные другого типа. В Python преобразование типов осуществляется с помощью функций следующего вида:

```
<название результирующего типа>(<значение исходного типа>)
```

Рассмотрим несколько примеров преобразования типов:

```
int("15")  # преобразование строки в целое число, результат 15
float("-15.997")  # строка в вещественное число, результат - 15.997
```

Строка, содержащая запись числа, преобразуется в значение числового типа лишь в том случае, если эта запись соответствует литералу числа, иначе произойдет ошибка выполнения. Так, `15` является корректным литералом типа `int`, а `-15.997` – корректным литералом типа `float`. При этом целочисленный литерал может быть интерпретирован как `float`, потому что все его символы (цифры и знак минус) допустимы для литерала этого типа и в математике множество вещественных чисел включает в себя множество целых.

```
float("15")
```
```
15.0
```

Однако *не наоборот*: литерал `float` не может быть интерпретирован как целочисленный, он содержит недопустимые для типа `int` символы (точку).

```
int("-15.997")
```
```
...
ValueError: invalid literal for int() with base 10: '-15.997'
```

Значение `float` может быть преобразовано в значение `int`, при этом дробная часть отбрасывается (математически вещественные числа отличаются от целых наличием дробной части, а у `int` она отсутствует, ее негде хранить).

```
int(-15.997)  # вещественное число в целое, результат -15
```

Подчеркнем: дробная часть просто *отбрасывается*, округления не происходит (например, число 3,7 будет преобразовано в 3).

```
int(True)  # логическое значение в целое число, результат 1
```

В `int` могут быть преобразованы значения логического типа по следующему правилу: `True` → `1`, `False` → `0`. Функция `float()` тоже работает с `bool`, она возвращает вещественные значения `1.0` и `0.0`.

```
str(10)  # целое число в строку, результат "10"
```

Значение любого типа может быть преобразовано в строку. Результатом такого преобразования является строковое представление этого значения. Для *простых* типов оно совпадает с их литералами, значения же *сложных* типов имеют свои собственные строковые представления. Преобразование в строку может быть осуществлено *явно* при вызове функции `str()`, а также *неявно* при попытке интерпретации значения как строкового. Последнее происходит при выводе значения на экран при помощи функции `print()`, а также при конструировании F-строк (об этом в параграфе 3.6).

Отдельным пунктом стоит упомянуть *булификацию* – преобразование значений любого типа в логическое значение. Эта процедура может выполняться *явно* при вызове функции `bool()`, а также *неявно* при попытке интерпретации значения как логического, когда в определенном месте программы (например, в условии операторов `if` и `while`, см. главу 4) ожидается значение булева типа, но оказывается значение иного типа.

Приведем *общее правило булификации*. Значение интерпретируется как **истина** (`True`) *всегда*, кроме следующих случаев:

- ✓ значение имеет числовой тип и равно 0;
- ✓ значение является пустой коллекцией, не содержащей элементов (см. параграф 5.7);
- ✓ значение пустое (`None`).

```
bool(123)  # ненулевое число: True
bool(0.0)  # нулевое число любого типа: False
bool([1, 2, 3])  # непустая коллекция (список): True
bool(())  # пустая коллекция (кортеж): False
bool(None)  # False
```

3.5. Функции

Функции – крайне важный элемент любого языка программирования. Все программирование построено на работе с функциями.

Функция – это программа, вложенная в основную (родительскую) программу, то есть *подпрограмма*.

Любая программа является реализацией некоторого алгоритма. Алгоритм, в свою очередь, имеет *входные (исходные)* и *выходные (результирующие) данные*. Соответственно, функция также имеет входные и выходные данные (рис. 21).

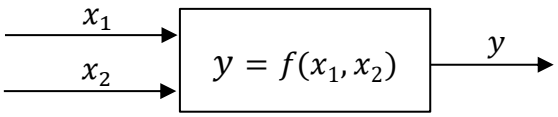


Рис. 21. Схематичное изображение функции двух аргументов $y = f(x_1, x_2)$

Можно сказать, что функция – это фрагмент программного кода, который при вызове выполняется весь целиком, от начала до конца. Этот код имеет имя, принимает некоторые данные на вход, как-то их обрабатывает и возвращает некоторый результат.

Основное предназначение функций – избежать дублирования кода в программе при многократном выполнении одного и того же алгоритма с различными входными данными. Например, при вычислении коэффициента корреляции между двумя наборами чисел нужно выполнить достаточно много математических вычислений, и программа, которая это делает, займет не одну строку кода. При этом в процессе корреляционного анализа часто необходимо вычислить не одно значение коэффициента, а несколько (для разных выборок). Входные данные будут отличаться, но алгоритм (методика вычислений) останется неизменным. Нужно ли будет дублировать один и тот же код, чтобы несколько раз вычислить коэффициент корреляции? Нет, если использовать функцию, которая это делает.

Функции бывают *стандартные* (уже готовые) и *пользовательские*. Пользовательские функции программист пишет сам, чтобы впоследствии использовать. Стандартные функции уже кем-то написаны (профессиональными разработчиками), и программист может их использовать в своей программе. Множество стандартных функций поставляется вместе с интерпретатором Python и доступно для использования программисту изначально.

Примеры стандартных функций – `int()` и `round()`:

```
int("3")  # функция преобразования строки в целое число
round(3.14159)  # функция округления вещественного числа
```

Вызов функции:

```
<имя функции>(<аргумент 1>, <аргумент 2>, ...)
```

Входные данные функции представлены *аргументами (входными параметрами)* – значениями того или иного типа. В качестве аргументов используются выражения. Теоретически аргументов может быть неограниченное количество (а может не быть ни одного).

Выходные данные функции – это значение, которое является результатом ее выполнения (то есть результатом выполнения алгоритма, который реализован данной функцией). Говорят, что функция ***возвращает*** то или иное значение, поэтому выходные данные называют ***возвращаемым значением*** функции. Возвращаемое значение – это значение (объект) того или иного типа (простого или сложного). Возвращаемое значение может быть *максимум одно*. Причем это не является ограничением: не стоит забывать, что типы данных бывают сложные (их значения состоят из нескольких элементов). Также бывают функции, которые вообще ничего не возвращают.

Зная, что выражения могут включать в себя вызов функций, не будем удивляться, встретив в программе каскадные (вложенные) вызовы функций (рис. 22).

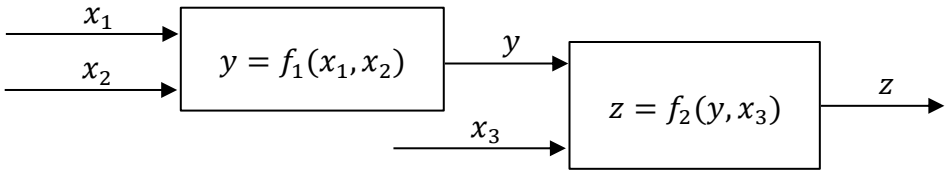


Рис. 22. Схематичное изображение каскадного вызова функций

В коде такой каскадный вызов может выглядеть следующим образом. Будем считать, что переменные `x1`, `x2`, `x3` и функции `f1()`, `f2()` в программе объявлены заранее и доступны для обращения.

```
z = f2(f1(x1, x2), x3)
```

Вызов таких вложенных друг в друга функций происходит «изнутри наружу», начинается с самой глубоко вложенной функции. Очевидно, что прежде, чем выполнится функция `f2()`, должно быть вычислено значение ее первого аргумента, а для этого должна быть выполнена функция `f1()`. Можно отметить, что значение, на рисунке обозначенное `y`, являющееся результатом выполнения функции `f1()`, не записывается ни в какую переменную, а сразу же посту-

пает на вход функции `f2()`. Стиль программирования, в котором активно используется такой «конвейерный» подход (результаты одних функций незамедлительно подаются на вход другим), называется *функциональным*.

3.6. Ввод-вывод данных

Любой алгоритм (а значит, и программа) имеет входные и выходные данные. Входные данные необходимо откуда-то получить.

Источники входных данных (способы ввода):
- ✓ явное задание в тексте программы через литералы;
- ✓ ручной ввод с клавиатуры (с консоли) во время выполнения программы;
- ✓ загрузка из файла на диске;
- ✓ загрузка по сети (например, из Интернета);
- ✓ другие (например, использование данных о текущей дате-времени).

Виды выходных данных (способы вывода):
- ✓ вывод на экран монитора (в консоль);
- ✓ сохранение в файл на диске;
- ✓ отправка по сети (например, отправка сообщения в мессенджере);
- ✓ вывод на печать (на принтер);
- ✓ другие.

Для *ввода данных* с клавиатуры в Python применяется функция `input()`.

```
input(<приглашение к вводу>)
```

Приглашение к вводу – это строковое значение, которое выводится в качестве запроса при вводе. Приглашение к вводу можно не задавать, в этом случае в качестве него будет использована пустая строка, и пользователь, увидев мигающий курсор, должен будет сам догадаться, что программа ожидает от него ввода с клавиатуры. Таким образом, функция `input()` является примером функции, которая может не иметь аргументов.

```
text = input("Введите ваше имя: ")  # ввод строки с клавиатуры
# и сохранение ее в переменную text
```
```
Введите ваше имя: Святослав
```

Результат выполнения следующего кода будет полностью аналогичен предыдущему:

```
invitation = "Введите ваше имя: "
text = input(invitation)
```
```
Введите ваше имя: Святослав
```

Здесь мы всего лишь поместили строку приглашения к вводу в переменную, которую затем использовали при вызове функции `input()`. Однако данный случай – пример ситуации, когда нецелесообразно создавать отдельную переменную для хранения объекта. Тем не менее сделать так никем не запрещено.

Отметим важный нюанс: ввести с клавиатуры с помощью `input()` можно *только значение строкового типа*. Даже набираемые на клавиатуре цифры будут сохранены в памяти как строка. Это значит, что для выполнения арифметических операций над числами, вводимыми с клавиатуры, необходимо преобразовать тип введенных значений из строкового в числовой.

```
# производится попытка преобразовать тип введенного с клавиатуры
# значения в целочисленный (int). Если введенная строка не содержит числа
# (состоит не только из цифр, точки и знака -), произойдет ошибка
number = int(input("Введите целое число: "))
```

Вывод на экран в Python реализуется при помощи функции `print()`:

```
print(<объект 1>, <объект 2>, ..., <объект N>,
      sep=<разделитель>, end=<строка, завершающая вывод>)
```

Входными данными этой функции помимо объектов, которые требуется вывести на экран, являются строковые значения разделителя и строки, завершающей вывод.

Разделитель (значение аргумента функции `sep`) – это строка, которая подставится между выводимыми объектами (если их больше одного). По умолчанию разделителем является пробел, то есть значения выводятся в строку через пробел.

Строка, завершающая вывод (значение аргумента функции `end`), – это строка, которая добавится в конец последнего выводимого объекта. По умолчанию разделителем является символ переноса строки `\n`, то есть каждый вызов функции `print()` выводит данные начиная с новой строки.

```
print("abc", 10, -99.9, sep=";", end="|")
print("Эта надпись выводится в той же строке, что и предыдущие 3 значения")
```
```
abc;10;-99.9|Эта надпись выводится в той же строке, что и предыдущие 3 значения
```

Внутри строки могут использоваться *специальные символы*:

`\n` – символ переноса строки.

`\t` – символ табуляции.

```
print("1-я строка\n2-я строка\n3-я строка\n\t4-я строка с отступом")
```
```
1-я строка
2-я строка
3-я строка
        4-ая строка с отступом
```

В Colab возможен вывод значения на экран *без функции* `print()`. Это работает, когда значение находится в последней строке ячейки и никак не обрабатывается: не записывается в переменную с помощью оператора присваивания и не подается на вход функции. То же самое происходит при работе с интерпретатором в *интерактивном режиме*: результат вычисления выражения (объект) сразу же выводится на экран (см. рис. 7).

```
number = 1
text = "Текстовая строка"
text
```
```
'Текстовая строка'
```

Как можно заметить, объект в последней строке выводится на экран без вызова специальной функции. Это работает только для последней строки. Если ячейка содержит несколько строк с объектами, которые необходимо вывести, применение функции `print()` обязательно (рис. 23).

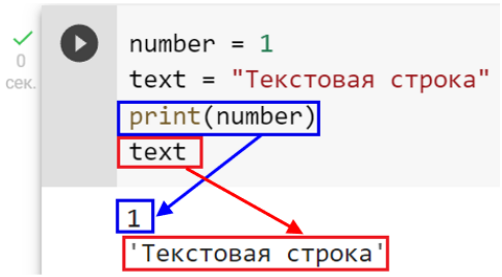


Рис. 23. Вывод нескольких значений на экран в разных строках

Режимы обработки и конструирование строк

Строковые литералы могут обрабатываться интерпретатором по-разному в зависимости от выбранного режима обработки, который задается с помощью *префиксов* (специальных символов в начале литерала – перед открывающими кавычками): `f`, `r`, `u`, `b` (также `F`, `R`, `U`, `B`).

Интерполяция строк (F-строки) – это механизм, позволяющий внедрять в строку (строковый литерал) значения выражений, не заботясь о преобразовании типов (оно происходит автоматически). F-строка имеет следующий вид:

```
f"текст... {<выражение>} текст..."
```

Префикс `f` включает возможность использования фигурных скобок как элементов синтаксиса языка. В эти скобки, расположенные внутри строкового литерала, помещаются выражения, результаты вычисления которых внедряются в строку.

```
Moscow_age = 874  # значение типа int
Perm_age = 298  # значение типа int

# конструирование строки с помощью соединения нескольких подстрок
text1 = "Москве " + str(Moscow_age) +" лет, а Перми " + str(Perm_age) + " лет"
# конструирование строки с помощью интерполяции
text2 = f"Москве {Moscow_age} лет, а Перми {Perm_age} лет"

print(text1)  # вывод строки, сконструированной соединением подстрок
print(text2)  # вывод строки, сконструированной интерполяцией
```

```
Москве 874 лет, а Перми 298 лет
Москве 874 лет, а Перми 298 лет
```

Внутри выражений в фигурных скобках могут быть и строковые литералы, в этом случае кавычки внешней и внедряемых строк должны быть *разного вида*:

```
print(f"Москва {'старше'} Перми на {abs(Moscow_age - Perm_age)} лет")
print(f'Москва {"старше"} Перми на {abs(Moscow_age - Perm_age)} лет')
```

```
Москва старше Перми на 576 лет
Москва старше Перми на 576 лет
```

R-строка (raw-строка) – строковый литерал, который обрабатывается «как есть», без поиска в нем специальных символов.

```
# F-строка
print(f"Москве {Moscow_age} лет, а\nПерми\t{Perm_age} лет")
```

```
Москве 874 лет, а
Перми    298 лет
```

```
# R-строка
print(r"Москве {Moscow_age} лет, а\nПерми\t{Perm_age} лет")
```

```
Москве {Moscow_age} лет, а\nПерми\t{Perm_age} лет
```

Префиксы можно комбинировать:

```
# интерполяция сработает (благодаря f), а табуляция нет (из-за r)
print(fr"{2}x{2}\t=\t{2 * 2}")
```

```
2x2\t=\t4
```

Косая черта \ является *экранирующим символом* в строках. С помощью нее задаются специальные символы табуляции и переноса строки, также она может использоваться, например, для включения внутрь строкового литерала кавычек *того же типа*, что и внешние кавычки.

```
print("Привет, \"мир\"!")  # еще один способ включить кавычки в строку
```
```
Привет, "мир"!
```

Напишем простую программу, которая запрашивает анкетные сведения о пользователе и красиво выводит их на экран:

```
name = input("Как вас зовут? ")  # ввод имени
age = input("Сколько вам лет? ")  # ввод возраста
city = input("В каком городе вы живете? ")  # ввод города
print()  # вывод пустой строки (чтобы отделить ввод от вывода)
print("=== ДАННЫЕ ОБ ОБЪЕКТЕ ===")  # вывод заголовка анкеты
print("Имя:", name)  # вывод имени
print(f"Возраст: {age}")  # вывод возраста
print(f"Место постоянной дислокации: {city}")  # вывод города
print("Статус: вооружен и опасен")  # вывод строковой константы
print("=========================")  # вывод линии для красоты
```
```
Как вас зовут? Ипполит
Сколько вам лет? 25
В каком городе вы живете? Петушки

=== ДАННЫЕ ОБ ОБЪЕКТЕ ===
Имя: Ипполит
Возраст: 25
Место постоянной дислокации: Петушки
Статус: вооружен и опасен
=========================
```

3.7. Математические вычисления

Как уже было сказано, в зависимости от типа со значением можно производить различные операции. Рассмотрим операции над числами (арифметические). Основные арифметические операции в Python приведены в табл. 2.

Таблица 2

Математические операции в Python

Тип операции	Оператор	Тип результата	Пример
Сложение	+	Если оба операнда `int`, то результат `int`, иначе `float`	`2 + 3.5  # 5.5`
Вычитание	-	Если оба операнда `int`, то результат `int`, иначе `float`	`2 - 3  # -1`
Умножение	*	Если оба операнда `int`, то результат `int`, иначе `float`	`2 * 3.5  # 7.0`
Деление без остатка	//	Если оба операнда `int`, то результат `int`, иначе `float`	`7 // 3  # 2` `7 // 3.6  # 1.0`

Остаток от деления	%	Если оба операнда `int`, то результат `int`, иначе `float`	`7 % 3  # 1`
			`7 % 3.6  # 3.4`
			`7 % 3.5  # 0.0`

Вещественное деление	/	`float` (результат – всегда вещественное число, даже если математически частное является целым)	`6 / 2  # 3.0`
Возведение в степень	**	Если оба операнда `int`, то результат `int`, иначе `float`	`3 ** 2  # 9`

Не будет лишним пояснить принцип действия операций `//` и `%`. *Вещественное деление* `/` работает как обычное деление в математике, и его результат совпадает с тем, который мы интуитивно ожидаем получить (за исключением того нюанса, что результат всегда вещественный). Результатом же *деления без остатка* является число, равное тому, сколько раз делитель помещается в делимом целиком. Например, 3,6 целиком помещается в 7 только один раз, поэтому результатом выражения `7 // 3.6` является единица (причем типа `float`, потому что второй операнд вещественный). Фактически результат деления без остатка равен результату вещественного деления с обнуленной (или отброшенной совсем в случае целочисленных операндов) дробной частью. Результатом операции *получения остатка от деления* является собственно остаток – число, которое получится, если из делимого вычесть результат деления без остатка. Например, 3,6 целиком помещается в 7 один раз, остается 3,4, а 3,5 помещается в 7 дважды, остаток 0. Удобнее всего использовать операции `//` и `%` с *целыми числами*, ведь из школьного курса математики мы точно знаем, что 7 на 3 делится с остатком, результатом будет 2, а остаток равен 1.

Как нам уже известно, выражение вычисляется не разом, а *поэтапно* в соответствии с приоритетами операций, из которых оно состоит. Перечислим *приоритеты математических операций* в составе выражения:

1. Операции в скобках.
2. Вызов функций.
3. Возведение в степень.
4. Взятие числа с противоположным знаком (–).
5. Умножение, деление, остаток от деления.
6. Сложение, вычитание.

Если операции, имеющие *равный приоритет*, расположены в выражении подряд, они выполняются по порядку *слева направо*. Такие операции называются ***левоассоциативными***.

```
# сложение и вычитание - левоассоциативные операции
5 - 8 + 14  # сначала вычислится 5 - 8 (будет -3), потом -3 + 14
```
```
11
```

Однако есть исключение: операция возведения в степень является ***правоассоциативной*** и выполняется *справа налево*. Данная логика продиктована математикой: в выражении x^{y^z} сначала вычисляется именно y^z, а уже затем x возводится в степень – результат первого шага.

```
# возведение в степень - правоассоциативная операция
2 ** 4 ** 1  # сначала вычислится 4 ** 1 (будет 4), потом 2 ** 4
```
```
16
```

Рассмотрим пример вычисления сложного арифметического выражения $y = \frac{a+bx+x^2-3^{x+\frac{a}{3}}}{\ln(\sqrt{x+1}-\sqrt[3]{x})}$.

```
import math  # подключим библиотеку с математическими функциями

# прежде чем обращаться к переменным, надо задать им какое-то значение
# введем значения переменных с клавиатуры, преобразуя в тип float
a = float(input("Введите переменную a: "))
b = float(input("Введите переменную b: "))
x = float(input("Введите переменную x: "))

# длинные команды можно переносить на новую строку с помощью символа \ в конце
y = (a + b * x + x ** 2 - 3 ** (x + a / 3)) / \
math.log(math.sqrt(x + 1) - x ** (1 / 3))

y  # выведем результат вычислений (значение y) на экран
```

Выполним эту программу, подставив в качестве значений $a = 2$, $b = -3$, $x = 0{,}5$, и убедимся, что ответ получился `3.3899710923743247`.

Запись выражения, вычисляющего `y`, получилась довольно-таки устрашающей. Разберемся, как происходит вычисление (рис. 24).

```
      6   5   7   3   8   4     2   1     15       14        10     9    13  12    11
y = (a + b * x + x ** 2 - 3 ** (x + a / 3)) / math.log(math.sqrt(x + 1) - x ** (1 / 3))
```

Рис. 24. Порядок выполнения операций при вычислении выражения

Итак, в выражении есть дробь, то есть деление. Поскольку синтаксис языка программирования ограничен символами, которые можно ввести с клавиатуры, математическую формулу любой сложности приходится записывать в одну строку теми средствами, какие есть. Сначала должно выполниться вычисление выражения в числителе, затем в знаменателе, а потом должно произойти деление числителя на знаменатель. В числителе сложное выражение, поэтому возьмем его в скобки. Показатель степени выражения $3^{x+\frac{a}{3}}$ тоже является сложным, поэтому и его следует взять в скобки. Очевидно, что вычисление начнется отсюда. Приведем последовательность вычислений с комментариями (табл. 3, 4). Серым цветом выделены операнды – значения, участвующие в вычислении операции текущего шага.

Таблица 3

Вычисление сложного математического выражения (числитель)

Шаг	Операция	Результат	Комментарий
1	`a / 3`	`0.6667…`	Приоритет деления выше, чем сложения.
2	`x + a / 3`	`1.1667…`	
3	`x ** 2`	`0.25`	После скобок самая приоритетная операция – возведение в степень.
4	`3 ** (x + a / 3)`	`3.6028…`	
5	`b * x`	`-1.5`	Затем идет умножение.
6	`a + b * x`	`0.5`	И только потом – сложение…
7	`a + b * x + x ** 2`	`0.75`	
8	`a + b * x + x ** 2 - 3 ** (x + a / 3)`	`-2.8528…`	…и вычитание.

После шага 8 вычисление числителя завершено, результатом его является некоторое значение числового вещественного типа. Переходим к знаменателю (см. табл. 4).

Таблица 4

Вычисление сложного математического выражения (знаменатель)

Шаг	Операция	Результат
9	`x + 1`	`1.5`
10	`math.sqrt(x + 1)`	`1.2247…`
11	`1 / 3`	`0.3333…`
12	`x ** (1 / 3)`	`0.7937…`
13	`math.sqrt(x + 1) - x ** (1 / 3)`	`0.4310…`
14	`math.log(math.sqrt(x + 1) - x ** (1 / 3))`	`-0.8415…`

Здесь производится расчет натурального логарифма, для чего применяется функция `log()` из библиотеки math. О подключаемых функциях поговорим чуть ниже, а пока просто вспомним, что вызов функций в составе выражения имеет наивысший приоритет (после скобок, хотя автору не приходит в голову пример выражения, в котором операция в скобках и вызов функции конкурировали бы между собой за первое место в очереди выполнения). Чтобы вычислить результат выполнения функции, нужно получить значение ее аргумента, которым (аргументом) является выражение $\sqrt{x+1} - \sqrt[3]{x}$. Обратимся к нему. Для извлечения квадратного корня в библиотеке math заготовлена функция `sqrt()`. Однако для извлечения корня произвольной степени готовой функции там нет, поэтому для решения данной проблемы следует вспомнить математику и тот факт, что $\sqrt[3]{x} = x^{\frac{1}{3}}$. Можно спросить: для чего дробь $1/3$ записывать как отдельную операцию в скобках? Дело в том, что это дробное число иррационально, обладает бесконечной дробной частью, и в тексте программы его невозможно записать точно.

Скобки же необходимы для того, чтобы сначала выполнилось деление и только затем возведение в степень (которое имеет более высокий приоритет).

Последней, пятнадцатой по счету, операцией выполнится деление числителя (значения, полученного по итогам операции 8) на знаменатель (значение, полученное по итогам операции 14). И, конечно, результат этого деления (объект – значение вещественного типа `3.389971092374324 7`) будет записан в переменную `y`, что можно считать шестнадцатой операцией в рассмотренной нами комплексной команде (инструкции) программы.

Комбинированное (составное) присваивание

Зачастую в программировании результат выполнения математической операции записывается в переменную, являющуюся одним из операндов. Типичные сценарии, в которых применяется такой подход (организация счетчиков и «копилок»), описаны в разделе про циклы (см. параграф 4.3), а пока разберемся с механизмом комбинированного присваивания.

Предположим, нужно вычислить сумму значений двух переменных и записать результат в первую переменную:

```
a = 3
b = 5

a = a + b  # значение выражения a + b запишем в переменную a
a  # убедимся, что a = 8
```

```
8
```

В случае присваивания сначала вычисляется выражение в правой части, затем результат записывается в переменную в левой части. Так и здесь: сначала вычисляется `a + b`, при этом переменная `a` хранит ранее записанное значение 3, оно и используется в вычислениях, и только после этого `a` перезаписывается новым значением 8. Очевидно, что на момент выполнения команды переменная `a` должна существовать и что-то хранить. Существует *специальная версия оператора присваивания* для случаев, когда результат операции помещается в переменную – левый операнд:

```
a += b  # результат аналогичен выражению a = a + b, только запись короче
```

Синтаксис комбинированного присваивания выглядит так:

```
<переменная - левый операнд> <знак операции>= <выражение - правый операнд>
```

Приведем несколько примеров:

```
a -= 1  # a = a - 1
a /= b ** 2  # a = a / b ** 2
a **= (b + 1) * 5  # a = a ** ((b + 1) * 5)
```

В последнем примере следует обратить внимание на приоритеты: сначала вычисляется значение выражения – правого операнда `(b + 1) * 5`, только потом осуществляется возведение левого операнда в степень. Если бы данная команда была записана с помощью обычного оператора присваивания, а не составного, вычисление показателя степени было бы необходимо взять в скобки (ведь у операции `**` более высокий приоритет, чем у операции `*`).

3.8. Подключение функций из библиотек

Существует многообразие подключаемых библиотек готовых функций на все случаи жизни (с понятием «библиотека» мы уже сталкивались ранее и даже рассматривали их установку в главе 2). Одной из библиотек, доступных программисту изначально, является math – библиотека с функциями для математических вычислений.

Подключение уже установленной в системе библиотеки осуществляется с помощью ключевого слова `import`, после которого через запятую перечисляются модули (один или больше):

```
import <модуль 1>, <модуль 2>, ...
```

Что такое ***модуль***? Мы уже упоминали, что это файл с исходным кодом, содержащий описание функций. Можно было бы не разделять понятия *библиотеки* и *модуля*, использовать эти термины как синонимы (зачастую это оправданно), если бы не тот факт, что название библиотеки, под которым она опубликована в репозитории, не всегда совпадает с именем ее файла-модуля. Примером подобной библиотеки является BeautifulSoup: модуль ее актуальной (на дату написания этого материала) версии называется bs4. Кроме того, нередко объемные сложные библиотеки состоят из нескольких модулей, имеющих разные названия.

Ключевое слово `as` позволяет задавать подключаемому модулю псевдоним, по которому к нему можно обращаться в тексте программы. Это удобно, если название модуля слишком длинное и сложное.

```
import math  # подключение библиотеки math
import math as mt  # использование псевдонима
```

Для обращения к функции из подключаемой библиотеки используется следующий синтаксис:

```
<название модуля (или его псевдоним)>.<название функции>
```

```
math.sqrt(144)  # вызовем функцию sqrt() для вычисления корня из 144
mt.sqrt(144)  # вызовем ту же функцию, обратившись по псевдониму
```

Возможно подключение не модуля целиком, а *только определенных функций* из него. Для этого служит синтаксическая конструкция `from … import`:

```
from <название модуля> import <функция 1>, <функция 2>, ...
```

В этом случае в *пространстве имен* программы (это область, в которой перечислены названия всех объектов программы в памяти, доступных для обращения) появятся имена подключенных функций, и для их вызова не понадобится указывать родительский модуль через точку.

```
from math import sqrt, floor, ceil  # подключим несколько функций
x = sqrt(16)  # извлечение квадратного корня
y = floor(123.45)  # округление вниз
z = ceil(123.45)  # округление вверх
```

Вместо перечисления функций можно подставить значок звездочки `*`, это добавит в пространство имен все функции, содержащиеся в модуле:

```
from math import *
```

Однако злоупотреблять данной возможностью не следует, потому что среди подключенных функций могут быть функции с именами, совпадающими с именами уже существующих в памяти объектов. Ошибки при этом не возникнет, но может возникнуть путаница: при обращении к объекту по имени будет осуществляться доступ к тому объекту, какой был добавлен в пространство имен *последним*. Продемонстрируем это правило на примере:

```
sqrt = 3  # объявим переменную sqrt
from math import *  # подключим все функции из math, в том числе и sqrt()
sqrt(9)  # попробуем вызвать функцию sqrt
```

```
3.0
```

Несмотря на то, что сначала в программе была объявлена переменная `sqrt`, позже данное имя было переиспользовано для стандартной функции `math.sqrt()`. А вот обратный случай:

```
from math import *  # подключим все функции из math, в том числе и sqrt()
sqrt = 3  # объявим переменную sqrt
sqrt(9)  # попробуем вызвать функцию sqrt
```

```
...
----> 3 sqrt(9)  # попробуем вызвать функцию sqrt

TypeError: 'int' object is not callable
```

В строке с попыткой вызова функции `sqrt()` произошла ошибка, потому что после подключения этой функции в программе была создана переменная, «присвоившая» себе имя `sqrt`, а переменная функцией не является, вызвать ее для выполнения нельзя, о чем и сообщает интерпретатор в описании ошибки.

Таким образом, использовать значок `*` при подключении всех функций модуля следует лишь тогда, когда нет сомнений, что эти функции имеют длинные специфические названия, которые уж точно не будут нами использованы в качестве имен переменных, а также не встречаются среди подключаемых в программе функций из других модулей.

Помимо функций в библиотеках могут встречаться и переменные (правильнее называть их *константами*, ведь их значение заранее прописано в модуле и не меняется в процессе выполнения программы). Например, в модуле math содержится несколько математических констант:

```
math.inf  # плюс бесконечность
-math.inf  # минус бесконечность (полученная путем добавления -)
math.pi  # число Пи (3.14...)
math.e  # экспонента (2.71...)
```

Полный перечень функций и констант в том или ином модуле можно получить, изучив документацию по библиотеке (документация math доступна по ссылке [8]), а также вызвав список доступных для обращения объектов, поставив точку после названия модуля и чуть подождав (рис. 25). Этот список не входит на экран целиком, однако его можно проматывать вниз-верх.

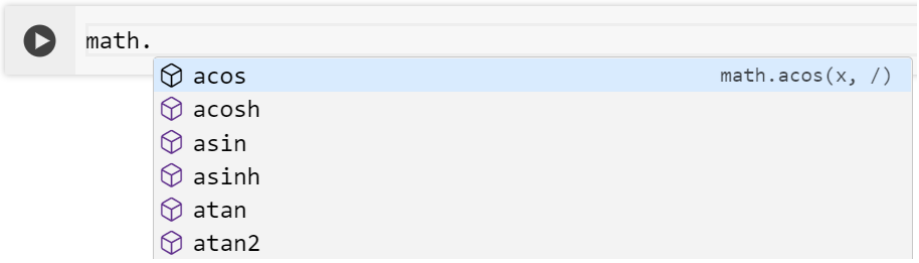


Рис. 25. Всплывающий список функций и констант модуля math в Colab

3.9. Операции сравнения чисел

Часто в программировании возникает необходимость сравнения нескольких значений между собой. Забегая вперед, отметим, что операции сравнения определены не только для чисел, но о сравнении объектов других типов поговорим в следующих главах. В Python существуют следующие операторы сравнения (таблица 5):

Таблица 5

Операторы сравнения в Python

Тип операции	Оператор	Пример
Проверка равенства	==	`-5 == 9  # False`
Проверка неравенства	!=	`-5 != 9  # True`
Проверка того, что левый операнд строго больше	>	`-5 > 9  # False`
Проверка того, что левый операнд нестрого больше	>=	`-5 >= 9  # False`
Проверка того, что левый операнд строго меньше	<	`-5 < 9  # True`
Проверка того, что левый операнд нестрого меньше	<=	`-5 <= 9  # True`

Результатом операции сравнения является *значение логического типа* (`bool`).

```
print("Четыре ли дважды два?", 2 * 2 == 4)
```
```
Четыре ли дважды два? True
```

Целое и вещественное числа считаются равными, если они равны математически (целые части совпадают, а дробная часть вещественного числа нулевая).

```
print("Четыре типа int равно четырем типа float?", 4 == 4.0)
```
```
Четыре типа int равно четырем типа float? True
```

Как можно заметить, в составе выражения операции сравнения имеют *более низкий приоритет*, чем арифметические. Исходя из этого, в примере сначала вычисляется произведение `2 * 2`, а затем производится сравнение полученного результата в левой части выражения (левого операнда) со значением в правой части (правого операнда).

В Python существует механизм не только парных, но и *множественных* сравнений. С его помощью удобно проверять вхождение некоторого значения в диапазон.

```
a = 4
3 <= a < 10  # проверка вхождения a в диапазон [3; 10)
```
```
True
```

Допустима и такая конструкция: `1 < 2 < 3 < 4`. Результат данного выражения будет истинным лишь в том случае, если каждое отдельное сравнение тоже будет истинным.

```
1 < 2 < 3 < 4  # True, потому что 1 < 2, 2 < 3 и 3 < 4
1 < 99 < 3 < 4  # False, потому что 1 < 99, 3 < 4, но 99 > 3
```

3.10. Особенности работы с вещественными числами

Целые числа в Python (тип данных `int`) могут быть неограниченной величины (неограниченной гипотетически; объем памяти в компьютере все-таки ограничен) и хранятся *точно*. Для визуального разделения групп разрядов длинных целых чисел можно использовать символ подчеркивания _.

```
x = 123_456_789_123_456_789_123_456_789_123_456_789_123_456_789_123_456
```

Вещественные числа (тип `float`) имеют точность 64 бита (8 байт), что позволяет хранить значения из диапазона от $\pm 1{,}7 \cdot 10^{-308}$ до $\pm 1{,}7 \cdot 10^{308}$. Таким образом, точность вещественных чисел в Python *ограничена*, они могут храниться *приближенно*. Это ограничение может приводить к погрешностям при вычислениях. Взглянем на результат следующего выражения:

```
0.1 + 0.2
```
```
0.30000000000000004
```

Математически сумма 0,1 и 0,2 равна 0,3. Однако программирование – это не абстрактная теоретическая математика (жизнь – это не теория), ему присущи ограничения, обусловленные практической реализацией хранения и обработки данных в памяти компьютера. И сумма указанных чисел окажется равна значению, пусть незначительно, но отличающемуся от ожидаемого результата. Казалось бы, в чем здесь проблема? Погрешность настолько мала, что ею можно пренебречь. Зачастую это так, но не всегда. Предположим, нам необходимо удостовериться, что результат некоторых вычислений равен конкретному значению. В зависимости от того, так это или нет, в дальнейшем будем предпринимать какие-то действия.

```
0.1 + 0.2 == 0.3
```
```
False
```

И выясняется, что сумма 0,1 и 0,2 и значение 0,3 в Python не равны между собой! Следовательно, выполнится действие, предполагающее, что результат

вычислений не равен 0,3 (что является явной ошибкой). Помочь в данной ситуации может функция `math.isclose()`, предназначенная для проверки равенства двух вещественных чисел. Она вычисляет разность между числами, и если эта разность меньше погрешности 10^{-9} (величину погрешности можно задать), числа считаются равными.

```
math.isclose(0.1 + 0.2, 0.3)
```
```
True
```

Не все вещественные числа хранятся приближенно. В памяти компьютера они представлены в виде $\frac{\text{мантисса}}{\text{степень двойки}}$, где мантисса – некоторое целое число. Например, значение 0,125 хранится в виде $\frac{1}{8} = \frac{1}{2^3}$, и следовательно, оно хранится точно. Числа, для которых не удается подобрать комбинацию мантиссы и степени двойки, позволяющую получить их отношение, в точности равное числу, представлены в памяти приближенно. Посмотреть на комбинацию, используемую Python для хранения того или иного вещественного числа, можно с помощью метода `.as_integer_ratio()`. Подробнее о методах поговорим в следующих главах.

```
0.125.as_integer_ratio()  # 0.125 хранится как 1/8
```
```
(1, 8)
```

```
0.3.as_integer_ratio() # 0.3 хранится как 5404319552844595/18014398509481984
# (в знаменателе 2 в степени 54)
```
```
(5404319552844595, 18014398509481984)
```

Ради любопытства проверим, как хранится в памяти сумма 0,1 и 0,2:

```
(0.1 + 0.2).as_integer_ratio()
```
```
(1351079888211149, 4503599627370496)
```

Заметно, что для 0,1 + 0,2 комбинация мантиссы и степени двойки не та, что используется для хранения 0,3. Правила, по которым выбирается эта комбинация, рассматривать не будем, это технические нюансы Python, однако об особенностях работы с вещественными числами забывать не следует.

Можно посмотреть на хранимое в памяти значение числа с высокой точностью, используя функцию `format()`. Вторым аргументом этой функции является строка с описанием правил форматирования числового значения при выводе на экран. Запись `".xf"` оставляет x знаков после запятой. Возвращаемым значением функции `format()` является строка:

```
format(0.3, ".20f")  # вывод 20 знаков после запятой
```
```
'0.29999999999999998890'
```

Особенности округления в Python

Для округления служит функция `round()`, выполняющая округление вещественного числа до заданного разряда по математическим правилам. Если номер разряда не указывать, выполняется округление до целого:

```
round(<число>, <номер разряда дробной части>)
```

В работе `round()` есть особенность, связанная с неточным хранением вещественных чисел в памяти. Взглянем на примеры:

```
round(1.55, 1)  # 1.6
round(2.55, 1)  # 2.5
```

По известным нам правилам если после разряда, до которого выполняется округление, следует цифра 5, округлять нужно вверх, иначе вниз. Согласно этой логике, результат округления 2,55 до десятых должен составить 2,6 (ведь с 1,55 все сработало как ожидалось). Однако `round()` возвращает 2,5. Чтобы разобраться в причинах такого поведения функции, выведем округляемые числа в формате с высокой точностью (20 знаков после запятой):

```
format(1.55, ".20f")  # 1.55000000000000004441
format(2.55, ".20f")  # 2.54999999999999982236
```

И все становится на свои места: второй разряд (сотые) у 1,55 составляет 5, поэтому округление происходит вверх, а у 2,55 второй разряд равен 4, поэтому округляется вниз. Из этого следует, что результат `round()` при округлении числа до какого-либо дробного разряда в общем случае *непредсказуем*. И дело тут не в специфике самой функции, а в особенностях хранения вещественных чисел в Python.

Однако это не единственный нюанс работы функции `round()`. При округлении чисел с дробной частью 0,5 *до целых* работает ***округление до ближайшего четного*** (такой способ округления еще называют *банковским*).

```
round(2.5)  # 2, потому что выбор из 2 и 4, 2 ближе
round(3.5)  # 4, потому что выбор из 2 и 4, 4 ближе
round(196.5)  # 196, потому что выбор из 196 и 198, 196 ближе
round(197.5)  # 198, потому что выбор из 196 и 198, 198 ближе
```

Все эти нюансы (неточное хранение вещественных чисел, особенности округления) важны при выполнении вычислений, требующих высокой точности: например, научных или финансовых.

Экспоненциальная запись числа

В процессе программирования (и не только: например, при работе в Excel) нередко приходится сталкиваться с выводом вещественных чисел в *экспоненциальной (инженерной) записи*. ***Экспоненциальная запись числа*** – это метод представления очень больших или очень малых чисел. Основная цель использования такой записи – упростить чтение этих чисел человеком. В экспоненциальной форме число представлено следующим образом: $M \cdot 10^p$, где M – мантисса, p – порядок. В Python данная запись выглядит так: $Me \pm p$. Например, $7{,}1e + 22 = 7{,}1 \cdot 10^{22}$. Порядок степени может быть и отрицательным: $7{,}1e - 22 = 7{,}1 \cdot 10^{-22}$.

```
710000000000000000000.0  # очень большое число
```
```
7.1e+22
```

```
0.000000000000000000071  # очень малое число
```
```
7.1e-22
```

3.11. Исключительные ситуации (ошибки)

Любой программист в процессе программирования вне зависимости от своего опыта и стажа допускает ошибки. Можно выделить *три вида* ошибок в программировании:

- ✓ логические ошибки;
- ✓ ошибки синтаксиса;
- ✓ ошибки выполнения.

Логические ошибки – это ошибки в алгоритме, в логике работы программы. Программа, содержащая такие ошибки, может корректно выполняться и выдавать результат, однако этот результат не будет совпадать с ожидаемым, то есть программа работает неправильно: например, неверно вычисляет значение факториала. Другим примером логических ошибок является *зацикливание* (см. параграф 4.3). Ошибки такого рода являются наиболее проблемными и трудноустранимыми, их мы рассматривать не станем.

Ошибки синтаксиса и выполнения выявляет сам транслятор языка и при их обнаружении выдает соответствующие сообщения. В программировании обнаружение таких ошибок в программе называют ***исключительными ситуациями*** (исключениями, exceptions).

Ошибки синтаксиса

Ошибки синтаксиса (SyntaxError) возникают при нарушении синтаксиса языка. Например, в приведенном коде (рис. 26) операция сложения записана не до конца, что является грубым нарушением ее синтаксиса.

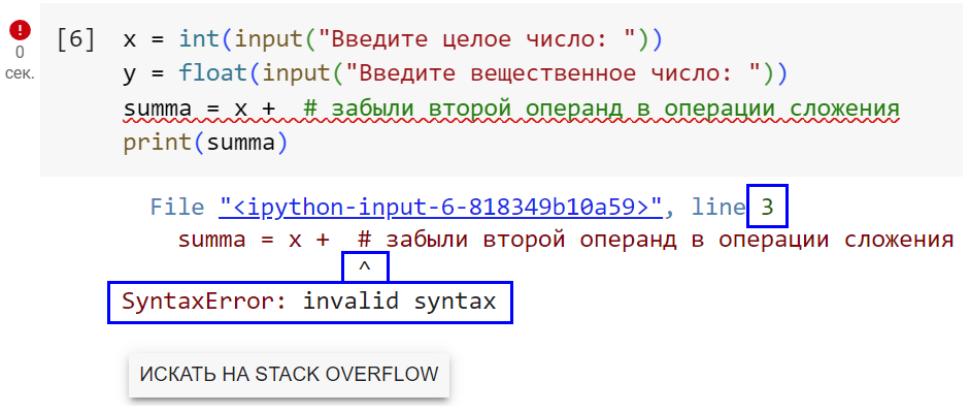
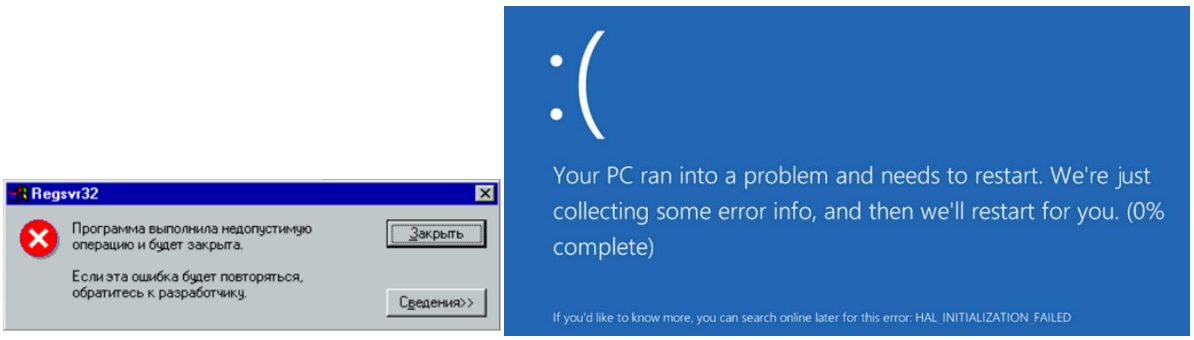


Рис. 26. Ошибка синтаксиса в Colab

Перед запуском кода на выполнение интерпретатором производится его синтаксический анализ. При наличии в коде синтаксических ошибок программа *не запускается вовсе*, и интерпретатор выдает *диагностическое сообщение*, содержащее номер строки программы с ошибкой (в примере выше – 3), указание на конкретное место в строке символом ^, а также тип и описание ошибки на английском языке. Ошибки синтаксиса находить и устранять проще остальных ошибок: они обнаруживаются при запуске программы, и достаточно лишь внимательно изучить указанное место в коде, чтобы внести коррективы. Более того, при использовании специализированного редактора кода, имеющего собственный синтаксический анализатор (например, редактора ячеек с кодом в Colab или редактора кода большинства сред разработки на Python), некорректные строки *подчеркиваются* сразу после их написания (еще до запуска кода).

Ошибки выполнения

Ошибки выполнения возникают в процессе исполнения программы. Предугадать их наличие в коде заранее очень сложно и, как правило, невозможно вовсе. При возникновении такой ошибки происходит *исключительная ситуация*, и программа досрочно завершает работу. На рис. 27 приведены сообщения о подобных ошибках, с ними в том или ином виде сталкивались все пользователи персональных компьютеров.



а *б*

Рис. 27. Сообщения об ошибках выполнения: *а* – ошибка программы в Windows 98; *б* – «синий экран смерти» в Windows 10[3]

Примерами ситуаций, в которых произойдет ошибка выполнения, являются следующие: некорректный ввод с клавиатуры (рис. 28), деление на ноль.

```
[12] print("Здравствуйте!")
     x = int(input("Введите целое число: "))
     print("Введенное число:", x)

     Здравствуйте!
     Введите целое число: пятнадцать
     ---------------------------------------------------------------------------
     ValueError                                Traceback (most recent call last)
     <ipython-input-12-ae570d10d42c> in <cell line: 2>()
           1 print("Здравствуйте!")
     ----> 2 x = int(input("Введите целое число: "))
           3 print("Введенное число:", x)

     ValueError: invalid literal for int() with base 10: 'пятнадцать'

     ИСКАТЬ НА STACK OVERFLOW
```

Рис. 28. Ошибка выполнения в Colab

Как можно заметить, исключительная ситуация (ошибка выполнения) происходит в конкретной строке программы, код до этой строки *выполняется в штатном режиме*. Диагностическое сообщение интерпретатора в данном случае отличается от сообщения об ошибке синтаксиса. Оно включает указание в виде стрелочки `---->` на конкретную строку с командой, при выполнении которой произошла ошибка. Также выводится тип и описание ошибки, при этом описание более информативно, чем просто «некорректный синтаксис». Так, в примере выше ошибка произошла при выполнении функции `int()` и была вызвана

[3] К так называемому синему экрану смерти в операционной системе Windows приводят критические ошибки, возникающие при работе системных программ. Такие ошибки обычно сигнализируют о проблемах с оборудованием компьютера, препятствующих его дальнейшей нормальной работе, и необходимости перезагрузки. Получить такой эффект в обычной прикладной программе на Python не очень просто; тем не менее подобные ошибки можно отнести к классу ошибок выполнения.

передачей в функцию входного строкового значения, которое невозможно интерпретировать как целое число. Конечно, предвидеть заранее, что пользователь программы введет с клавиатуры некорректное значение, не удастся.

Перечислим некоторые *типы ошибок выполнения* и ситуации, в которых они возникают. За полным перечнем стандартных исключений в Python можно обратиться к документации языка.

TypeError: операция применена к объекту несоответствующего типа. Не забудем: тип данных строго определяет, какие операции можно выполнять со значениями этого типа.

```
"Мне " + 19 + "лет"  # попытка сложить строку и число
```
```
...
TypeError: can only concatenate str (not "int") to str
```

ValueError: функция получает аргумент правильного типа, но некорректного значения. Такая ситуация типична при использовании математических функций: значение аргумента может выходить за пределы области определения данной функции.

```
import math
math.sqrt(-4)  # попытка извлечь квадратный корень из отрицательного числа
# причем та же операция командой (-4) ** 0.5 сработает,
# результатом будет комплексное число (complex)
```
```
...
ValueError: math domain error
```

ZeroDivisionError: деление на ноль. Такая ошибка может возникнуть при использовании всех видов деления: вещественного `/`, без остатка `//`, а также получения остатка от деления `%`.

```
5 / 0  # попытка поделить на ноль
```
```
...
ZeroDivisionError: division by zero
```

NameError: не обнаружена переменная (или функция) с указанным именем. Прежде чем обращаться к объекту по имени, объект с таким именем нужно в программе объявить (например, создать переменную или подключить функцию из библиотеки).

```
age = 19  # создали переменную age,
print(Age)  # но вывести на экран пытаемся Age (регистр символов важен)
```
```
...
NameError: name 'Age' is not defined
```

IndexError: индекс за границами диапазона. Такая ошибка часто происходит при работе с индексированными коллекциями (см. главу 5): например, при попытке обратиться к пятнадцатому элементу списка, в котором элементов всего десять.

KeyError: несуществующий ключ. Эта ошибка часто происходит при работе со словарями (см. параграф 10.2) в случае обращения к элементу с ключом, которого в словаре нет.

Обработка исключений

Алгоритм обладает свойством *массовости*, он способен решать задачу при множестве различающихся входных данных. Из этого следует практическая невозможность предусмотреть и заранее определить все данные, которые могут оказаться на входе (например, множество чисел бесконечно). А значит, простор для возникновения ошибок выполнения очень широк: на вход программе (особенно сложной и объемной, каковыми в той или иной степени являются все программы, имеющие практическую полезность) может попасть «что угодно». Чтобы свести количество ошибок выполнения к минимуму, программу тщательно тестируют на различных наборах входных данных. Однако исключить абсолютно все ошибки возможно *далеко не всегда*. В языках программирования и в Python в частности эта особенность реального мира предусмотрена. Существуют специальные средства, позволяющие обрабатывать исключительные ситуации так, чтобы они не приводили к нештатному завершению программы.

За обработку ошибок в Python отвечает оператор `try`, который записывается в виде следующей конструкции:

```
try:
  "рискованный" участок кода
except <тип ошибки 1>:
  реакция на ошибку типа 1
except <тип ошибки 2>:
  реакция на ошибку типа 2
...
else:
  код, который выполнится, если ошибок не было
finally:
  код, который выполнится в конце всей конструкции безусловно
```

«Рискованный» участок кода в блоке `try` содержит команды, во время выполнения которых возможно возникновение ошибок. Блоки `except`, которых может быть несколько, содержат код – реакцию на возникновение ошибки указанного типа. При этом тип ошибки может быть *не указан вовсе*: в этом случае будут перехватываться все возможные ошибки. В блок `else` помещается код, который выполнится в случае отсутствия ошибок при выполнении кода в блоке `try`, а

блок `finally` содержит код, который выполнится после обработки ошибок всегда – даже если ошибок выполнения не было. (О *блоках кода*, выделяемых отступами, поговорим подробнее в параграфе 4.2.)

В своем минимальном виде конструкция `try` выглядит так:

```
try:
  result = 10 / 0  # на ноль делить нельзя, здесь всегда будет ошибка
except:
  print("Произошла ошибка!")
```

```
Произошла ошибка!
```

Как можно заметить, исключительная ситуация возникла (потому что в блоке `try` всегда будет происходить деление на ноль: делитель задан константой, а не переменной), однако программа не «сломалась», просто вывелось сообщение об ошибке, то есть выполнился код в блоке `except`. Рассмотрим пример использования полной конструкции `try`:

```
from math import asin  # функция, вычисляющая арксинус

x = int(input("Введите целое число: "))
try:
  result = asin(10 / x)
except ZeroDivisionError:
  print("Ошибка: деление на ноль!")
except ValueError:
  print("Ошибка: некорректное значение аргумента функции!")
else:
  print("Ошибок не было")
finally:
  print("Обработка \"рискованного\" кода завершена")
```

```
Введите целое число: 0
Ошибка: деление на ноль!
Обработка "рискованного" кода завершена
```

В данном случае к ошибке вновь привело деление на ноль, обусловленное вводом 0 в качестве делителя с клавиатуры. При этом выполнился блок `except`, перехватывающий ошибку ZeroDivisionError. Код в блоке `finally` также выполнился в конце.

Взглянем на вывод той же программы при других введенных значениях переменной `x`:

```
Введите целое число: 1
Ошибка: некорректное значение аргумента функции!
Обработка "рискованного" кода завершена
```

При вводе 1 ошибка произошла уже не при делении, а при выполнении функции `asin()`: значение `10.0`, являющееся результатом деления `10 / 1`, не

попадает в область определения арксинуса, и выполнился блок кода, перехватывающий ошибку ValueError.

Если же ввести такое значение `x`, при котором ошибки в коде блока `try` не произойдет вовсе, выполнятся блоки `else` и, конечно, `finally`, результат будет следующим:

```
Введите целое число: 15
Ошибок не было
Обработка "рискованного" кода завершена
```

Оператор `try` позволяет значительно повысить отказоустойчивость программы, снизить количество ошибок выполнения при ее работе, однако злоупотреблять им не следует. Если в программе произошла действительно важная ошибка, из-за которой дальнейшее корректное выполнение алгоритма *невозможно* (например, не создалась нужная переменная или того хуже – в переменную не записалось нужное значение), нельзя просто подавить это исключение и позволить программе работать дальше как ни в чем не бывало: к благоприятным последствиям это не приведет. Некоторую осмысленную обработку исключения сделать необходимо: так, в ряде ситуаций допустимым может быть задание подходящего значения переменной, которое не нарушит дальнейшую работу алгоритма, при обработке ошибки в блоке `except`. Еще одним вопиющим проявлением небрежности является объявление *всего кода программы* потенциально рискованным и помещение его целиком в блок `try`. Такое применение данного инструмента в большинстве случаев является бесполезным, потому что нельзя всю программу считать нашпигованной ошибками (в конце концов, надо писать хорошие программы, тщательно их тестировать), следует точечно обрабатывать отдельные небольшие участки кода и прописывать адекватную реакцию на *возможные при их выполнении* специфические ошибки.

3.12. Задания для самостоятельной работы

Решите следующие задачи:

1. Напишите программу, которая запрашивает имя пользователя и выводит следующую строку:

```
Приветствую тебя, о "<введенное имя>"!
```

2. Напишите программу, которая запрашивает сведения о пользователе (имя, возраст, возраст сестры, название родного города) и выводит их на экран в следующем виде. Учитывать направление разницы в возрасте (старше или

младше) не требуется: расчет следует вести исходя из предпосылки старшинства сестры.

```
=== АНКЕТА ===
     Имя: <введенное имя>
     Возраст: <введенный возраст>
     Сестра старше на <разница в возрасте> лет
     Родной город: <введенный город>
==============
```

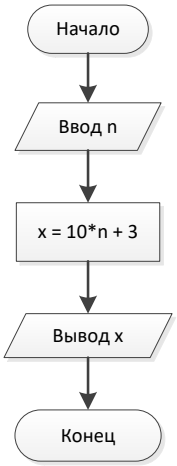
3. Введите с клавиатуры два целых числа. Вычислите и выведите на экран их сумму, разность, произведение, результат вещественного деления первого числа на второе, результат деления без остатка первого числа на 10, результат получения остатка от деления второго числа на 10. Используйте приглашения к вводу (чтобы пользователь понимал, чего ждет от него программа). Проверяйте корректность ввода с помощью оператора `try`.

4. Условия и циклы

В этой главе описываются фундаментальные алгоритмические конструкции, без которых не обходится практически ни одна программа.

4.1. Основные алгоритмические структуры

Алгоритм может быть *линейным*, когда все его шаги (команды) выполняются строго друг за другом. Примером может быть алгоритм, вычисляющий значение математического выражения. На рис. 29 представлен такой алгоритм в виде *блок-схемы* (это один из способов графического представления алгоритма).



а

```
n = int(input("Введите n: "))
x = 10 * n + 3
print(x)
```

б

Рис. 29. Блок-схема (*а*) и программная реализация на Python (*б*) линейного алгоритма, вычисляющего значение выражения $10n + 3$

Строго линейными от начала до конца бывают только простейшие, самые примитивные алгоритмы. Как правило, алгоритм решения какой-либо практической задачи имеет более сложную структуру, включающую помимо линейных участков кода еще *ветвления* и *циклы*.

Всего существует *три основных алгоритмических структуры*:

1. *Линейное выполнение*. Шаги алгоритма выполняются последовательно друг за другом.
2. *Ветвление*. В зависимости от истинности некоторого условия выполняется тот или иной набор команд алгоритма.
3. *Цикл*. Один и тот же фрагмент алгоритма выполняется несколько раз подряд.

Две последние рассмотрим подробно.

4.2. Условный оператор if

Ветвление – это развилка в алгоритме: можно пойти одним путем (выполнить одну последовательность шагов), а можно другим (выполнить другую последовательность шагов). Одновременно эти два пути не могут быть пройдены за один проход по алгоритму, они *взаимоисключающие*: в зависимости от выполнения некоторого условия идем либо сюда, либо туда.

В качестве примера ветвления рассмотрим фрагмент алгоритма получения второго высшего образования (рис. 30).

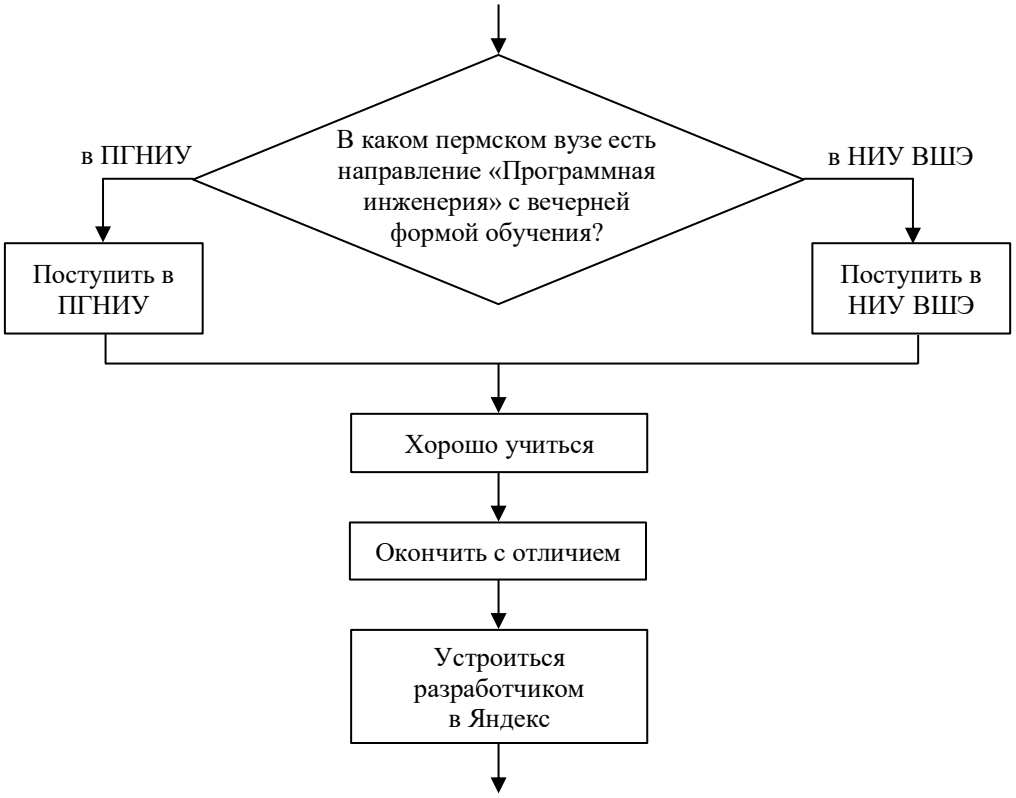


Рис. 30. Блок-схема алгоритма с ветвлением

Проверка условия должна представлять собой некоторый *вопрос*, ответами на который являются либо «да» / «нет», либо какие-то варианты дальнейших действий. Как правило, в программировании вопрос сформулирован с вариантами ответов «да» / «нет», то есть ветвление *двоичное* (развилка на два пути) (рис. 31).

Рис. 31. Двоичное ветвление

Чтобы организовать *множественное ветвление* (развилку на несколько путей), можно использовать вложенные друг в друга двоичные ветвления (рис. 32).

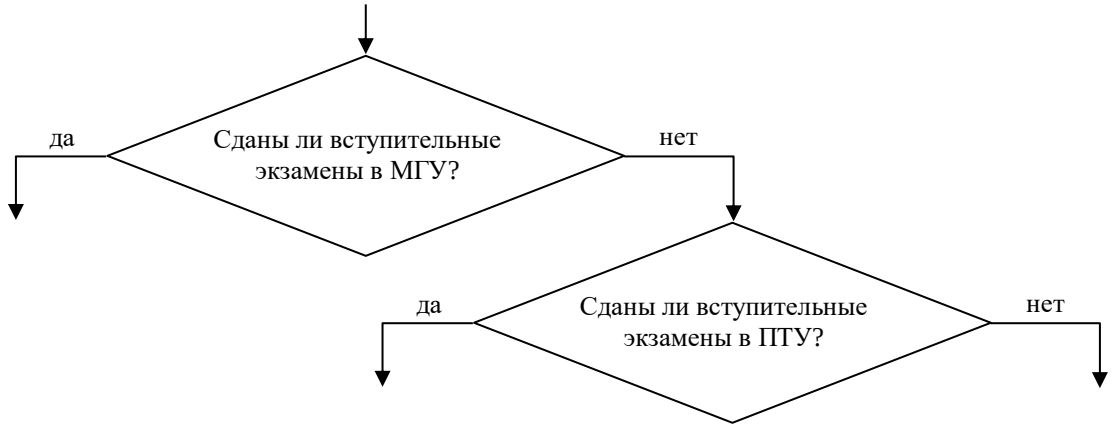


Рис. 32. Множественное ветвление (развилка на три пути)

Реализовать такую развилку с выбором в Python помогает условный оператор `if`:

```
if <условие 1>:
  команды первого блока кода (выполнятся, если истинно условие 1)
elif <условие 2>:
  команды второго блока кода (выполнятся, если истинно условие 2)
elif <условие 3>:
  команды третьего блока кода (выполнятся, если истинно условие 3)
...
else:
  команды блока кода, которые выполнятся при ложности всех условий
```

Условие – это выражение, результат вычисления которого имеет *логический тип* (`bool`) либо может быть интерпретирован как значение логического типа (вспомним механизм булификации, см. параграф 3.4). Ключевые слова `elif` и `else` в команде могут отсутствовать. Приведем пример простейшей команды с условным оператором:

```
age = 15
if age < 18:
  print("Вы несовершеннолетний, спиртного не продадим!")
```

```
Вы несовершеннолетний, спиртного не продадим!
```

Значение выражения `age < 18` равно `True`, потому что в переменной `age` содержится число 15. Как следствие этого, условие выполняется, и команда `print()` выводит сообщение на экран. При `age` ≥ 18 команды блока бы не выполнились и ничего не вывелось.

Сделаем небольшое отступление и разберемся с понятием *блок кода*. Мы уже сталкивались с ним, изучая оператор `try` (см. параграф 3.11). ***Блок кода*** – это фрагмент программы, который начинается с двоеточия, а все его команды

отделяются отступами. Вспомним один из пунктов стандарта PEP8: «для выделения блоков кода следует использовать отступы из четырех пробелов» (см. параграф 1.2). Здесь речь как раз идет о блоках кода, которые в программировании на Python встречаются постоянно (рис. 33).

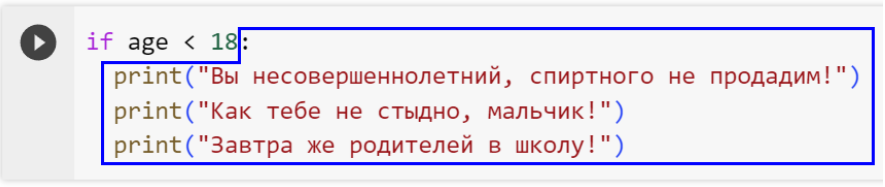


Рис. 33. Блок кода внутри условного оператора

Отметим, что в Google Colab после двоеточия при переходе на новую строку (клавиша Enter) отступы проставляются автоматически, и при этом используется не четыре пробела, как рекомендовано, а два. Количество пробелов, составляющих отступ, может быть произвольным: главное, чтобы все отступы одного блока кода имели *одно и то же* количество пробелов. В противном случае программа не запустится из-за ошибок синтаксиса (рис. 34).

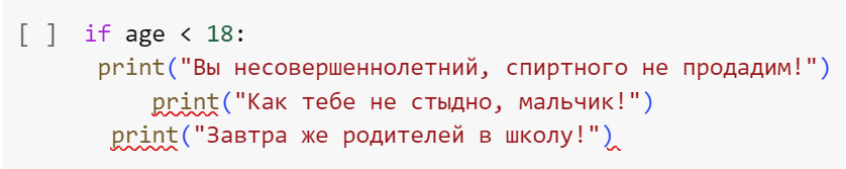


Рис. 34. Нарушение одинаковой длины отступов в одном блоке кода

Команды основной программы никаких отступов не имеют, это верхний, *первый* уровень кода. Блок кода внутри основной программы образует *второй* уровень кода, а блок кода внутри второго уровня – третий *уровень* кода. И таких вложенных друг в друга блоков может быть множество, при этом выстраивается наглядная иерархическая структура.

```
if age < 18:
  print("Вы несовершеннолетний, спиртного не продадим!")
  if age > 16:
    print("Хоть Вам и больше 16, все равно не продадим!")
```

Вспомним, что в качестве одного из преимуществ языка Python мы приводили следующее: красивый и понятный код программ. Структурирование кода с помощью отступов, будучи неотъемлемой особенностью синтаксиса языка (в силу чего не поддается игнорированию программистом при всем его гипотетическом желании), относится, по мнению автора, к аргументам в пользу выдвину-

того тезиса о красоте кода. При этом нужно быть внимательным. Сравните приведенные ниже фрагменты кода: в зависимости от количества отступов вторая команда `print()` относится к одному из двух блоков кода или к основной программе.

```
age = 15

# оба сообщения выведутся только в случае выполнения обоих условий
# при age=15 не выведется ничего, потому что условия не выполняются
if age < 18:
  if age > 16:
    print("Хоть Вам и больше 16, но спиртного не продадим!")
    print("Как тебе не стыдно, мальчик!")
```

```
age = 15

# первое сообщение выведется при выполнении обоих условий
# второе сообщение выведется при выполнении первого условия (age<18)
# (оно относится к блоку первого if, но не относится к блоку второго)
if age < 18:
  if age > 16:
    print("Хоть Вам и больше 16, но спиртного не продадим!")
  print("Как тебе не стыдно, мальчик!")
```

```
Как тебе не стыдно, мальчик!
```

```
age = 93

# первое сообщение выведется при выполнении обоих условий
# второе выведется безусловно
# (оно вообще не относится к блокам кода внутри условных операторов)
if age < 18:
  if age > 16:
    print("Хоть Вам и больше 16, но спиртного не продадим!")
print("Как тебе не стыдно, мальчик!")
```

```
Как тебе не стыдно, мальчик!
```

Разобравшись с блоками кода и отступами (незапуск программы из-за слишком вольного обращения с отступами – частая ошибка начинающих программистов на Python!), вернемся к изучению оператора `if`. Пример конструкции с ключевым словом `else`:

```
age = 25
if age < 18:
  print("Вы несовершеннолетний, спиртного не продадим!")
else:
  print("Добро пожаловать, дорогой клиент! Вам виски или мартини?")
```

```
Добро пожаловать, дорогой клиент! Вам виски или мартини?
```

В данном случае `age` > 18, поэтому условие в блоке `if` не выполнится, но присутствует блок `else`, поэтому выведется сообщение про клиента. Наконец, рассмотрим пример с ключевым словом `elif`:

```
age = 18
if age < 18:
  print("Вы несовершеннолетний, спиртного не продадим!")
elif age == 18:
  print("По закону-то Вы совершеннолетний, но может, не стоит?")
else:
  print("Добро пожаловать, дорогой клиент! Вам виски или мартини?")
```
```
По закону-то Вы совершеннолетний, но может, не стоит?
```

Поскольку `age` = 18, выполнится второе условие (первый блок `elif`), и выведется призыв к благоразумию. Напомним, что блоков `elif` может быть несколько, проверяться условия будут сверху вниз до тех пор, пока какое-то из них не выполнится. Если не выполнилось ни одно условие в блоках `if` и `elif`, тогда либо выполнится код в блоке `else`, либо, при отсутствии оного, не произойдет ничего, программа продолжит выполняться дальше.

Существует *краткая однострочная форма условного оператора*. Ключевые слова `if` и `else` могут находиться внутри выражения:

```
<значение, если условие истинно> if <условие> else <значение, если ложно>
```

При этом `else` является обязательной частью: результатом выражения *должно быть* некоторое значение – либо одно, либо другое. Рассмотрим пример: в данном случае в переменную `drink` попадет строка «Буратино» (речь о газированном напитке), потому что условие `age` ≥ 18 не выполнится.

```
age = 15
drink = "Коньяк" if age >= 18 else "Буратино"
drink
```
```
'Буратино'
```

Логические операции и сложные условия

Логические операции (операции булевой алгебры) – это операции, которые можно осуществлять со значениями логического типа (`bool`). С их помощью можно составлять сложные (составные) высказывания и, как следствие, сложные условия. Основных логических операций *три*: конъюнкция, дизъюнкция и отрицание. Для описания принципа действия логических операций обычно используют так называемые *таблицы истинности* (рис. 35).

Конъюнкция

a	b	a and b
0	0	0
0	1	0
1	0	0
1	1	1

Дизъюнкция

a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

Отрицание

a	not a
0	1
1	0

Рис. 35. Таблицы истинности основных логических операций
(обозначения: 1 – истина, 0 – ложь)

Конъюнкция (логическое И) реализуется оператором `and`. Результат операции равен `True` тогда и только тогда, когда оба операнда тоже `True`, в остальных случаях `False`.

```
# гулять пойдем только при выполнении обоих условий
# (и температура > 20, и солнце), тогда и только тогда

temperature = 25
sunny = False
walk = (temperature > 20) and sunny  # walk будет False, потому что
# второе условие не выполняется
walk
```

```
False
```

Отметим, что логические операции имеют *более низкий приоритет*, чем операции сравнения, поэтому сравнение можно не выделять скобками, оно все равно выполнится в первую очередь. Однако использовать скобки для наглядности и снижения риска запутаться не возбраняется. Рассмотрим пример с конъюнкцией в условном операторе:

```
temperature = 25
sunny = False
if (temperature > 20) and sunny:
  print("Отличная погода - и тепло, и солнечно! Пойдем гулять!")
else:
  print("Что-то погода не очень, лучше посидим дома.")  # этот блок
```

```
Что-то погода не очень, лучше посидим дома.
```

Дизъюнкция (логическое ИЛИ) реализуется оператором `or`. Результат операции равен `False` тогда и только тогда, когда оба операнда тоже `False`, в остальных случаях `True`.

```
# гулять пойдем при выполнении любого из условий - подойдет
# и температура > 20, и солнце

temperature = 25
sunny = False
walk = (temperature > 20) or sunny  # walk будет True, потому что
# первое условие выполняется (требуется выполнение хотя бы одного из двух)
walk
```

```
True
```

Пример с дизъюнкцией в условном операторе:

```
temperature = 25
sunny = False
if (temperature > 20) or sunny:
  print("Погода сойдет! Жди в 7, пойдем гулять!")  # этот блок
else:
  print("Что-то погода не очень, лучше посидим дома.")
```

```
Погода сойдет! Жди в 7, пойдем гулять!
```

Отрицание (логическое НЕ) реализуется оператором `not`. Это очень простая операция, которая инвертирует значение операнда (`True` превращает в `False` и наоборот).

```
# гулять пойдем, если температура не меньше или равна 20
# (то есть больше 20)

temperature = 25
walk = not (temperature <= 20)  # walk будет True, потому что
# температура больше 20
walk
```

```
True
```

Пример с отрицанием в условном операторе:

```
temperature = 25
if not (temperature <= 20):
  print("На улице тепло, жди в 7, пойдем гулять!")  # этот блок
else:
  print("На улице прохладно, лучше посидим дома.")
```

```
На улице тепло, жди в 7, пойдем гулять!
```

Следует отметить особенность операторов `and` и `or`, связанную с применение механизма булификации (см. параграф 3.4) при их использовании. Несмотря на то, что эти операторы реализуют операции алгебры логики, их операндами могут быть не только значения типа `bool`, но и значения других типов, которые могут быть интерпретированы как логические. С учетом всего изученного данный факт удивления вызывать не должен, однако в случае использования операндов произвольного типа результатом операций является не булево значение, а *один из операндов*. В примере ниже 1 интерпретируется как `True`, 0 как `False`, поэтому результатом конъюнкции является 0 (ложь), а результатом дизъюнкции – 1 (истина). Более глубоко в нюансы работы данных операторов погружаться не будем.

```
1 and 0  # 0
1 or 0  # 1
```

4.3. Оператор цикла while

Еще одной важнейшей алгоритмической структурой является цикл.

Цикл – это повторение фрагмента алгоритма *конечное* число раз. Можно спросить, зачем нужно повторять одни и те же действия несколько раз? Выполнять многократно одинаковые действия *над одинаковыми данными* занятие, действительно, бесполезное: сколько бы раз мы не вычисляли дважды два, результатом всегда будет четыре. Однако существуют задачи (и их огромное количество), которые требуют повтора одних и тех же действий *над разными данными*.

Циклы бывают *двух видов*:

1. *Арифметический цикл.* Выполняется заранее заданное количество раз. Типичным примером задачи, успешно решаемой с помощью арифметического цикла, является перебор всех объектов некоторой выборки. Например, в группе 30 студентов. Нужно обойти каждого и узнать его мнение об изученном курсе. С каждым студентом будут проведены одни и те же мероприятия (будет задан один вопрос и выслушан ответ на него), однако сами студенты будут меняться. При этом их количество известно заранее, до начала опроса – 30.

2. *Цикл с условием.* Выполняется до тех пор, пока истинно некоторое условие. Примером алгоритма, в котором используется такой цикл, является итерационный алгоритм решения какой-нибудь математической задачи оптимизации. Шаги алгоритма повторяются до тех пор, пока не будет достигнута приемлемая величина погрешности. После каждой итерации выполняется проверка: если погрешность больше заданной, вычисления продолжаются, иначе результат считается достигнутым.

Для организации арифметических циклов в Python обычно используют оператор `for`. О нем поговорим в следующей главе, а пока подробно рассмотрим циклы с условием. Приведем пример такого цикла: на рис. 36 изображен фрагмент алгоритма гарантированного достижения выигрыша в лотерею. Процесс покупки лотерейного билета продолжается до тех пор, пока не попадется билет с выигрышем.

Повторяемый набор действий называется ***телом цикла***. Оно выполняется, пока истинно условие, а если условие не выполняется, тело цикла также не выполнится ни разу. Представим себе, что еще до покупки лотерейного билета у нас уже был билет с выигрышем (допустим, мы его нашли или кто-то подарил). Условие не выполнится, тело цикла не выполнится, и мы сразу радостно направимся получать выигрыш.

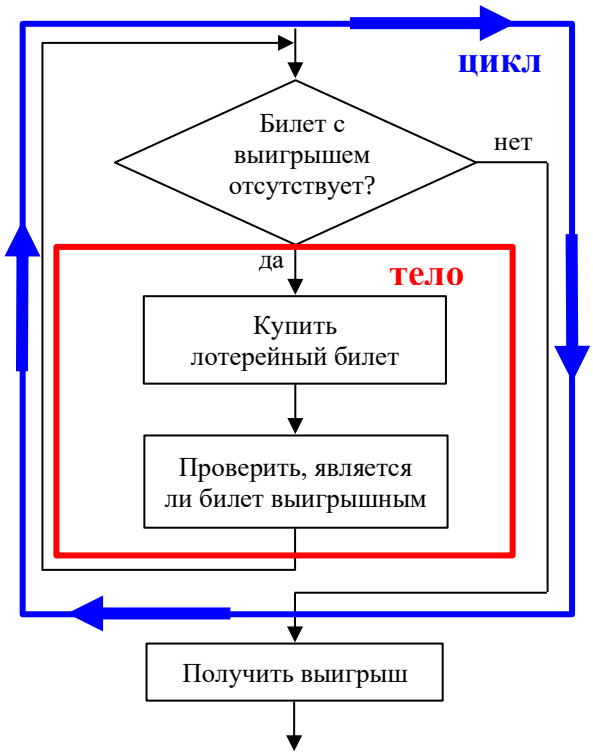


Рис. 36. Блок-схема алгоритма выигрыша в лотерею

При выполнении цикла с условием в теле цикла **обязательно** должны меняться какие-то данные (те, от которых зависит результат проверки условия), иначе условие всегда будет истинным, и выхода из цикла никогда не случится (произойдет так называемое *зацикливание*). Вспомним одно из свойств алгоритма (см. начало первой главы) – *конечность (результативность)*. Как может быть алгоритм бесконечным? Может – если в нем присутствует бесконечный цикл. И это, конечно же, недопустимо. Переменную, которая меняет свое значение в теле цикла, часто называют ***параметром цикла***.

В Python цикл с условием организуется при помощи оператора `while`. Данный оператор записывается в виде следующей конструкции:

```
while <условие>:
  команды тела цикла
else:
  команды, которые выполнятся в случае невыполнения условия
```

Как и в случае оператора `if`, условие – это некоторое выражение с результатом логического типа (механизм булификации также работает). Цикл `while` похож на условный оператор (причем обладает еще более простым синтаксисом): тело цикла выполняется в случае истинности условия, только не один раз, а несколько. Как и набор команд в условном операторе, тело цикла представляет собой *блок кода* с двоеточием и отступами. Как правило, в теле цикла меняется значение переменной, участвующей в вычислении выражения в условии. Блок

`else` позволяет выполнить некоторые действия по завершении работы цикла (включая случай, когда цикл не выполнился ни разу, то есть условие изначально было ложным), но на практике используется редко.

Рассмотрим несколько типичных сценариев применения цикла `while`.

Повтор некоторой операции n раз (организация арифметического цикла):

```
n = 5  # количество итераций (шагов) цикла
i = 0  # переменная - счетчик цикла (данные, которые меняются в цикле)
while i < n:
  print(i)  # действие, которое нужно повторять
  # (в данном случае вывод значения счетчика на экран)
  i += 1  # увеличим счетчик цикла на 1,
  # иначе условие i < n будет выполняться всегда
```

```
0
1
2
3
4
```

Для изменения значения ***счетчиков*** (переменных, которые последовательно в цикле увеличиваются или уменьшаются на некоторую константу) уместно использовать *комбинированное присваивание* (о нем говорилось в параграфе 3.7). Применим код выше для ввода с клавиатуры трех целых чисел:

```
n = 3
i = 0
while i < n:
  x = int(input("Введите " + str(i + 1) + "-е число: "))
  # также строку - приглашение к вводу можно было сформировать с помощью
  # механизма F-строк: f"Введите {i + 1}-е число: "
  i += 1
```

```
Введите 1-е число: 64
Введите 2-е число: -7
Введите 3-е число: 111
```

Введенные числа мы сохраняем в одну и ту же переменную `x`, многократно ее перезаписывая. Если необходима какая-то обработка вводимых чисел, можно выполнять ее в теле цикла после ввода числа. Подсчитаем сумму введенных чисел. Для накопления суммы используем специальную переменную – «копилку»:

```
i = 0
summa = 0  # переменная-копилка
while i < 3:  # переменная n не очень нужна, она всегда равна 3
  x = int(input(f"Введите {i + 1}-е число: "))
  summa += x  # прибавляем x к предыдущему накопленному значению
  i += 1

print(f"Сумма введенных чисел: {summa}")
```

```
Введите 1-е число: 3
Введите 2-е число: -1
Введите 3-е число: 15
Сумма введенных чисел: 17
```

А как быть, если мы не знаем, сколько раз нужно выполнить тело цикла (например, заранее не знаем количество чисел в последовательности)? В этом случае значение, с которым нужно сравнивать счетчик цикла, следует каким-то образом определить до запуска цикла: например, ввести с клавиатуры.

```
i = 0
n = int(input("Введите количество чисел в последовательности: "))
while i < n:
  """команды тела цикла"""
  i += 1  # никогда не забываем увеличивать счетчик цикла!
```

Попробуем применить цикл для подсчета суммы чисел от 1 до 100. Как это сделать? Конечно, можно записать очень длинное выражение в коде программы:

```
# этот код не запустится из-за многоточия
summa = 1 + 2 + 3 + 4 + 5 + ... + 99 + 100
```

Но данное решение очень неэффективное, программисты так не поступают. Следует подумать, и мы увидим, что вычисление суммы из сотни слагаемых можно представить как повторение одного и того же действия: добавления очередного слагаемого в копилку. А ведь мы это уже делали в одном из предыдущих примеров! Только там слагаемые вводились с клавиатуры, а здесь их и вводить не нужно: числа от 1 до 100 – это монотонная последовательность, каждый член которой можно вычислять, прибавляя единицу к предыдущему. Такая увеличивающаяся на 1 переменная у нас тоже уже была: это счетчик цикла. Напишем готовую программу:

```
S = 0  # переменная-копилка: в ней копим сумму чисел
number = 1  # переменная - счетчик цикла и одновременно очередное
# слагаемое (первое слагаемое - единица)
while number <= 100:  # пока счетчик не превысил 100, складываем
  S += number  # добавляем очередное число в копилку
  number += 1  # переходим к следующему числу; если убрать эту строку,
  # счетчик никогда не станет больше 100, и произойдет зацикливание
  # (данные должны меняться!)

print(f"Сумма чисел от 1 до 100 равна {S}")
```

```
Сумма чисел от 1 до 100 равна 5050
```

Отвлечемся от арифметических циклов со счетчиками и рассмотрим пример «чистого» цикла с условием. Пусть необходимо вводить некоторое значение с клавиатуры до тех пор, пока не будет введено «да» (регистр символов важен). В данном случае обязательное изменение данных в теле цикла достигается за счет ввода с клавиатуры новых значений. Отметим, что в этом примере в условии цикла происходит сравнение не чисел, а строк; операторы проверки равенства ==

и неравенства != со строками работают так же, как с числами, а более подробно о сравнении нечисловых объектов поговорим в следующих главах.

```
answer = ""
while answer != "да":  # пока не введено "да", цикл продолжается
 answer = input("Вы кот? ")

print("Я тоже кот. Будем дружить!")
```

```
Вы кот? Нет
Вы кот? Да нет же
Вы кот? Нет!
Вы кот? Да
Вы кот? Да кот я, кот!
Вы кот? угу
Вы кот? Так точно, Ваше Высокопревосходительство!!!
Вы кот? да
Я тоже кот. Будем дружить!
```

Алгоритм поиска максимума (минимума)

Рассмотрим стандартный алгоритм *поиска максимального* (или *минимального*) элемента последовательности чисел в цикле. Идея алгоритма состоит в следующем. Вводится переменная для хранения локального максимума. С этой переменной необходимо поочередно сравнить все числа последовательности, и, если очередной элемент оказывается *больше* предыдущего найденного максимума, значит, это предыдущее значение максимумом не является, его необходимо обновить (актуализировать), записав в переменную максимума значение очередного элемента. Как только будут обработаны все члены последовательности, эта переменная будет содержать истинное значение максимума.

Необходимо ответить на вопрос: каково должно быть *первоначальное значение переменной максимума*? Ведь ее надо сравнивать с членами последовательности, а значит, до начала сравнения некоторое значение в ней уже должно быть (не пустое). К выбору первоначального значения надо подойти ответственно, потому что если оно окажется больше всех чисел последовательности, то максимум *ни разу не обновится*. И результат алгоритма будет некорректным: переменная максимума будет содержать значение, которого даже не было в последовательности. Следовательно, первоначальное значение переменной должно быть:

- ✓ достаточно мало;
- ✓ не больше максимального элемента последовательности.

Для решения задачи выбора подходящего достаточно малого значения существует *два подхода*:

1. Взять в качестве первоначального максимума любой элемент последовательности (как правило, используют *первый* элемент). Этот вариант является *общепринятым и наиболее правильным*.

2. Взять настолько малое значение, что последовательность никогда не будет содержать чисел меньше этого значения, то есть нужно задать минимальное из всех возможных в языке программирования число. В Python это минус бесконечность: `-math.inf`.

Напишем программу поиска максимума последовательности чисел, введенных с клавиатуры. Для задания первоначального значения максимума используем второй подход: возьмем минимально возможное число.

```
import math

n = int(input("Введите количество чисел в последовательности: "))
localmax = -math.inf  # первоначальное значение максимума
i = 0  # счетчик цикла
while i < n:  # пока счетчик не достиг n
  x = int(input(f"Введите {i + 1}-е число: "))  # ввод очередного числа
  if x > localmax:  # если число больше локального максимума,
    localmax = x  # обновим его
  i += 1  # увеличим счетчик на 1

print(f"Максимум последовательности: {localmax}")
```

```
Введите количество чисел в последовательности: 5
Введите 1-е число: -6
Введите 2-е число: 0
Введите 3-е число: 199
Введите 4-е число: -200
Введите 5-е число: 17
Максимум последовательности: 199
```

Алгоритм поиска *минимума* очень похож на описанный выше, отличия лишь в следующем:

✓ первоначальное значение переменной минимума должно быть достаточно *велико* (подойдет плюс бесконечность);

✓ условием обновления переменной минимума является то, что очередной элемент последовательности оказался не больше предыдущего минимума, а *меньше*.

Напишем программу поиска минимума. Для задания первоначального значения минимума используем общепринятый подход: возьмем первое значение последовательности. Для этого придется его ввести отдельно до начала цикла.

```
import math

n = int(input("Введите количество чисел в последовательности: "))
localmin = int(input(f"Введите 1-е число: "))  # первоначальный минимум
i = 1  # счетчик цикла начинается уже не с 0, а с 1
# (ведь одно число ввели, и цикл нужно выполнить на 1 раз меньше)
while i < n:  # пока счетчик не достиг n
  x = int(input(f"Введите {i + 1}-е число: "))  # ввод очередного числа
  if x < localmin:  # если число меньше локального минимума,
    localmin = x  # обновим его
  i += 1  # увеличим счетчик на 1

print(f"Минимум последовательности: {localmin}")
```

```
Введите количество чисел в последовательности: 5
Введите 1-е число: -6
Введите 2-е число: 0
Введите 3-е число: 199
Введите 4-е число: -200
Введите 5-е число: 17
Минимум последовательности: -200
```

Операторы break и continue

Существуют специальные операторы, управляющие выполнением цикла. Их использование имеет смысл и допустимо *только в теле цикла* (внутри соответствующего блока кода); попытка вызвать такой оператор в основном коде программы приведет к ошибке.

Оператор `break` прерывает работу цикла. Часто его используют для организации выхода из цикла при выполнении некоторого условия. Конечно, у цикла есть свое собственное условие продолжения его работы, но в ряде случаев более удобно проверять условие не продолжения, а *прекращения* работы. В случае выхода из цикла через `break` команды блока `else` (при его наличии) не выполняются.

```
# "вечный" цикл (условие всегда истинно)
while True:
  answer = input("Введите число (для выхода введите stop): ")
  if answer == "stop":
    break
```

```
Введите число (для выхода введите stop): 0
Введите число (для выхода введите stop): 5
Введите число (для выхода введите stop): 99.999
Введите число (для выхода введите stop): stop
```

Оператор `continue` прерывает *текущий шаг (итерацию)* цикла и выполняет переход к следующему шагу. В примере ниже требуется ввести пять чисел с клавиатуры, однако из-за `continue` выполнение тела цикла завершается до того, как будет увеличено значение счетчика. В результате произойдет зацикливание.

```
i = 0
while i < 5:
  x = int(input(f"Введите {i + 1}-е число: "))
  continue
  i += 1  # эта команда никогда не выполнится
```

```
Введите 1-е число: 4
Введите 1-е число: 8
Введите 1-е число: 9
Введите 1-е число: 3
Введите 1-е число: 2
Введите 1-е число: 11
Введите 1-е число: 19
...
```

Рассмотрим более осмысленный пример использования `continue`:

```
# подсчитаем количество четных чисел от 5 до 15
number = 4  # первое число (на 1 меньше, потому что потом увеличим)
count = 0  # счетчик четных чисел
while number < 15:
  number += 1  # вычисление очередного числа (это нельзя сделать в конце
  # тела цикла, потому что при нечетных числах код в конце не выполняется)
  if number % 2 != 0:  # если число нечетное
    continue  # перейдем к следующему числу (и следующему шагу цикла)
  # этот код выполнится только в случае четного number
  print(f"Четное число: {number}")
  count += 1  # увеличим счетчик четных чисел на 1
print(f"Количество четных чисел от 5 до 15: {count}")
```

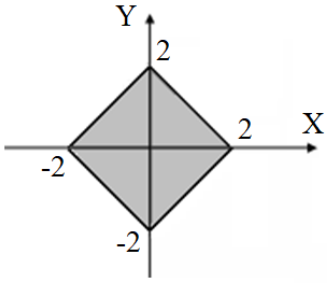
```
Четное число: 6
Четное число: 8
Четное число: 10
Четное число: 12
Четное число: 14
Количество четных чисел от 5 до 15: 5
```

4.4. Задания для самостоятельной работы

Решите следующие задачи:

1. Напишите программу для решения квадратных уравнений. Нужно учесть, что уравнение может иметь два корня, один корень или совсем не иметь корней.

2. Напишите программу, определяющую, принадлежит ли точка заштрихованной фигуре. Координаты точки вводятся с клавиатуры.



3. Доработайте задание про анкету (см. параграф 3.11). Учтите следующие нюансы:

✓ нужно ввести не только возраст родственника, но и его пол, то есть родственником может быть сестра или брат;

✓ родственник может быть старше, младше или того же возраста;

✓ слово «год» имеет различные формы в зависимости от количества лет (1 год; 3 года; 19 лет).

Примеры сообщений о возрасте родственника:
Сестра старше на 5 лет.
Сестра того же возраста.
Брат младше на 10 лет.
Сестра младше на 1 год.
Брат старше на 4 года.

4. Напишите программу, которая вычисляет сумму отрицательных чисел в последовательности из n целых чисел. Длина последовательности n и все ее числа вводятся с клавиатуры.

5. Напишите программу для ввода строк диалога двух человек. Диалог продолжается до тех пор, пока кто-то из собеседников не скажет: «Пока». Каждая фраза собеседников должна вводиться в отдельной строке. По окончании диалога нужно вывести сообщение «Диалог завершен. Всего было сказано <n> фраз.», где n – количество реплик в диалоге. Диалог не может содержать менее двух фраз (иначе его нельзя будет назвать диалогом). Иными словами, нельзя в качестве первой же фразы ввести «Пока», а если введено, нужно сообщить о некорректном вводе и продолжить запрашивать реплики.

Пример диалога:
Здравствуйте, можно в гости?
Да, заходите.
Спасибо.
Чай будете?
Давайте.
Вам сахар нужен?
Да, нужен. Спасибо.
Вы ко мне пришли по делу?
Нет, просто пообщаться.
Ну, вот и пообщались. До свидания!
Пока.
Диалог завершен. Всего было сказано 11 фраз.

5. Коллекции

В данной главе вводится понятие объектно-ориентированного программирования и его важнейших категорий. Описываются сложные типы данных (коллекции) и методы и приемы работы с ними.

5.1. Классы, объекты, методы

Ранее мы упоминали, что типы данных в Python бывают *простые* (к ним относятся числа, строки, логический тип) и *сложные (составные)*. Составные типы отличаются от простых тем, что представляют собой некоторую структуру, объединяющую в себе несколько значений простого или сложного типа. Освоив приемы обработки данных простых типов, познакомимся с типами сложными.

Существуют разные подходы (методологии) программирования. Очень широко распространенным является ***объектно-ориентированное программирование (ООП)*** – это методология, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Центральными понятиями ООП являются *класс* и *объект* (рис. 37). ***Класс*** – это описание структуры данных и методов работы с ними, помещенное в единую «капсулу». Можно сказать, что это *тип данных*, шаблон. ***Объект*** – это конкретный экземпляр класса (можно сказать, *значение* того или иного типа).

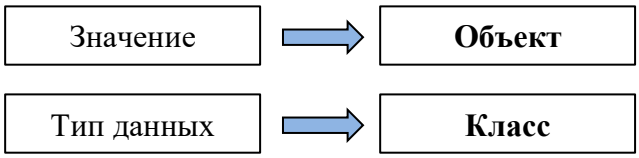


Рис. 37. Соответствие понятий «тип данных», «значение» и терминов ООП

Python – *объектно-ориентированный язык программирования*. Все данные в нем являются **объектами** – экземплярами классов. Даже функции (подпрограммы) – это тоже объекты. А все изученные нами типы данных – это классы.

```
print(type("Привет"))  # строковый объект - это экземпляр класса str
<class 'str'>
```

Отличие **объекта** (экземпляра некоторого *класса*) от **значения** некоторого *типа*, как мы его понимали до текущего момента, состоит в том, что объект может хранить *не только данные*, но и **функции**, то есть алгоритмы обработки этих данных. Такие функции, принадлежащие объекту, называются ***методами*** объекта. От типа данных зависит структура этих данных и операции, которые с ними можно выполнять. Поскольку *на самом деле* типы данных являются классами,

именно класс определяет методы, принадлежащие тому или иному объекту. Например, у класса `str` есть свои методы, которые можно использовать для обработки строковых объектов. А в параграфе 3.10 мы уже упоминали `.as_integer_ratio()` – метод класса `float`.

Обращение к методам осуществляется через *точку*:

```
<объект>.<название метода>(<аргументы метода>)
```

Объект может быть представлен литералом, переменной или выражением (которое после вычисления все равно превратится в некоторое значение).

```
text = ", мир!"

"Привет".find("и")  #вызов метода объекта, заданного литералом строкового типа
text.find("и")  # вызов метода объекта, заданного переменной строкового типа
("Привет" + text).find("и")  # вызов метода объекта, заданного выражением
# с результатом строкового типа
```

Мы знаем, что функция как реализация некоторого алгоритма имеет входные и выходные данные. Метод – это тоже функция, но «привязанная» к данным (своему объекту). Следовательно, метод при вызове всегда имеет входные данные (даже если вызывается без аргументов): входными данными метода *как минимум* выступает его объект.

Есть методы, которые изменяют свой объект, а есть такие, которые изменений в объект не вносят, только извлекают из него данные, как-то их обрабатывают и возвращают результат. В дальнейшем мы познакомимся со многими методами обоих типов.

5.2. Итерируемые объекты (коллекции)

В Python существует разновидность объектов, называемых *итерируемыми*. ***Итерация*** в общем смысле – это какое-либо повторяемое действие (например, очередная итерация цикла). В более узком смысле итерация – это перебор каких-то элементов по очереди.

Итерируемый объект (коллекция, контейнер) – это специальный объект, состоящий из нескольких элементов (можно сказать, *подобъектов*), которые можно **перебрать**. Другими словами, итерируемый объект можно рассматривать как некоторую последовательность элементов, к каждому из которых можно поочередно обратиться (рис. 38). В программировании и информатике вообще упорядоченные последовательности элементов называют ***одномерными массивами***. Еще говорят, что по таким объектам можно *итерироваться*.

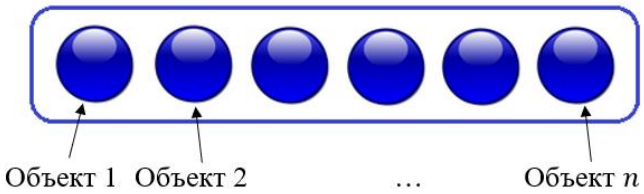


Рис. 38. Коллекция (итерируемый объект), он же одномерный массив

К коллекциям в Python можно отнести *следующие типы данных* (перечень не полный):

- ✓ строка (`str`);
- ✓ диапазон (`range`);
- ✓ кортеж (`tuple`);
- ✓ список (`list`).

Коллекции могут быть **изменяемыми** и **неизменяемыми**. В *изменяемые* коллекции можно добавлять и удалять элементы. Также коллекции могут быть **индексированными** и **неиндексированными**. В *индексированных* коллекциях можно обращаться к любому элементу по его порядковому номеру (индексу).

При работе с коллекциями используются некоторые общие механизмы и приемы, которые мы разберем на примерах конкретных типов итерируемых объектов.

5.3. Строка

Строка (`str`) – **неизменяемый индексированный** тип коллекций. Ранее мы относили строку к простым (элементарным) типам данных, таким как числа и логические значения. Но автор немного схитрил, чтобы не вносить лишней путаницы. Ведь строки – это один из первых типов данных, с которыми приходится иметь дело в процессе обучения программированию.

```
string = "Жизнь прекрасна!"  # с этой строкой будем дальше работать
```

Строка представляет собой последовательность символов. Каждый символ – это тоже строка (значение типа `str`), только единичной длины. На примере строк рассмотрим некоторые возможности, доступные при работе с коллекциями.

Для *определения длины коллекции* (то есть количества элементов в ней) служит функция `len()`.

```
len(string)  # функция вернет числовое значение 16, потому что
# в строке 16 символов (пробел и знак восклицания тоже считается)
16
```

Обращение к элементу на определенной позиции (допустимо для индексированных коллекций) осуществляется с помощью квадратных скобок `[i]`, где `i` – порядковый номер элемента (нумерация начинается с нуля).

```
string[0]  # Ж - первый символ
string[2]  # з - третий символ
string[-1]  # ! - последний (первый с конца) символ
```

Дублирование (многократное повторение) строки реализуется с помощью операции умножения на натуральное число.

```
string * 3  # продублируем строку дважды (исходная строка + 2 дубля)
```
```
'Жизнь прекрасна!Жизнь прекрасна!Жизнь прекрасна!'
```

Сцепление (сложение, конкатенация) строк осуществляется с помощью операции сложения.

```
string + " А также удивительна!"
```
```
'Жизнь прекрасна! А также удивительна!'
```

5.4. Диапазон

Диапазон (`range`) – **неизменяемый**, **индексированный** тип коллекций. Строго говоря, диапазон – это не совсем коллекция. Это достаточно специфический тип данных (модель данных в Python не самая простая, если погружаться глубоко). Для упрощения будем относить диапазон к коллекциям, потому что для него характерны многие присущие им свойства.

Диапазон – это специальный тип данных, позволяющий создавать монотонные целочисленные последовательности. Объект-диапазон создается с помощью функции `range()`. Такой обычно не помещают в переменную. Часто его используют для организации арифметических циклов, а также для перебора числовых последовательностей в цикле (примеры будут ниже).

```
range(10)  # последовательность от 0 до 9
# (не до 10! range(n) генерирует последовательность от 0 до n - 1)
range(1, 11)  # последовательность от 1 до 10
range(0, 11, 2)  # последовательность от 0 до 10 с шагом 2
range(10, 0, -1)  # последовательность от 10 до 1 с шагом -1
```

```
len(range(2, 101, 2))  # длина последовательности четных чисел от 2 до 100
```
```
50
```

```
range(2, 101, 2)[30]  # 31-й элемент последовательности
```
```
62
```

5.5. Кортеж

Кортеж (`tuple`) – **неизменяемый индексированный** тип коллекций. Кортеж представляет собой упорядоченную последовательность значений любых типов. Поскольку он является неизменяемым, после создания кортежа в него нельзя добавлять и из него нельзя удалять элементы.

```
my_tuple = (1, 2, 3)  # создание кортежа перечислением элементов
my_tuple = 1, 2, 3  # то же самое (скобки можно не указывать)
my_tuple
```
```
(1, 2, 3)
```

```
# в кортеже можно хранить объекты разных типов, в т.ч. другие кортежи
1, 3.14, "abc", (True, False)
```
```
(1, 3.14, 'abc', (True, False))
```

```
(5,)  # единичный кортеж (из одного элемента), скобки можно опустить
```
```
(5,)
```

```
(0,) * 10  # кортеж из 10 нулей (дублирование работает)
```
```
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

```
my_tuple[1]  # обращение ко 2-му элементу кортежа
```
```
2
```

```
len(my_tuple)  # длина кортежа (количество элементов)
```
```
3
```

```
# сцепление (сложение, конкатенация) кортежей - по аналогии со строками
my_tuple + (5, 0, 7) + (0,) * 3
# как можно заметить, дублирование кортежа имеет более высокий приоритет
# (как и положено умножению * относительно сложения +)
```
```
(1, 2, 3, 5, 0, 7, 0, 0, 0)
```

Упаковка и распаковка значений кортежа

На примере кортежей рассмотрим очень важный и полезный механизм, поддерживаемый коллекциями в Python:

```
a = 1
b = 5
c = 9
# процедура создания кортежа еще называется упаковкой значений в кортеж
my_tuple = a, b, c
my_tuple
```
```
(1, 5, 9)
```

```
# но если есть упаковка, то должна быть и распаковка?
c, b, a = my_tuple  # количество переменных в левой части
# должно соответствовать длине кортежа
print(a, b, c)
```
```
9 5 1
```

В действительности *не всегда* при распаковке кортежа в переменные количество переменных в левой части оператора присваивания должно строго соответствовать длине кортежа. Переменных может быть *меньше*, чем элементов в кортеже. В этом случае в переменную, отмеченную специальным образом (звездочкой `*`), попадут все «лишние» элементы кортежа в виде *списка* (об этом типе данных чуть ниже). Проще всего понять описанный механизм на примерах:

```
new_tuple = 10, 20, 30, 40, 50  # создадим новый кортеж
new_tuple
```
```
(10, 20, 30, 40, 50)
```

```
*x1, x2, x3 = new_tuple  # в x2, x3 попали 2 последних элемента,
# все остальные попали в x1 в виде списка
print(x1, x2, x3)
```
```
[10, 20, 30] 40 50
```

```
x1, *x2 = new_tuple  # в x1 попал 1-й элемент,
# все остальное попало в x2 в виде списка
print(x1, x2)
```
```
10 [20, 30, 40, 50]
```

```
x1, x2, *x3, x4 = new_tuple  # в x1, x2 попали 2 первых элемента,
# в x4 попал последний элемент, остальное попало в x3 в виде списка
print(x1, x2, x3, x4)
```
```
10 20 [30, 40] 50
```

Механизмы упаковки (распаковки) объектов в кортеж (из кортежа) делают возможной работу знаменитой команды (ею часто демонстрируют специфику языка Python), позволяющей *обменять значениями две переменные*:

```
a = 10
b = "hi"

a, b = b, a  # в переменную a попадет значение из b, a в b - из a

print(a, b)
```
```
hi 10
```

```
# поменяем значения переменных обратно, используя
# более классический способ - через третью переменную

temp = a
a = b
b = temp
print(a, b)
```
```
10 hi
```

Некоторые методы кортежа

Метод `.count()` возвращает количество вхождений элемента в кортеж (сколько раз искомый элемент встречается в кортеже).

```
print(new_tuple)  # вспомним содержимое new_tuple
new_tuple.count(10)  # 10 встречается 1 раз
```
```
(10, 20, 30, 40, 50)
1
```

```
(new_tuple + (10, 10)).count(10)  # а в новом сконструированном кортеже
# 10 встречается уже трижды
```
```
3
```

```
new_tuple.count(999)  # 999 не встречается ни разу
```
```
0
```

Метод `.index()` возвращает позицию первого вхождения элемента в кортеж. Если искомый элемент отсутствует в кортеже, возникнет ошибка ValueError.

```
new_tuple.index(30)  # 30 является 3-м (нумерация с 0) элементом
```
```
2
```

5.6. Список

Список (`list`) – **изменяемый индексированный** тип коллекций. Список очень похож на кортеж – с той разницей, что он *изменяемый*. В список можно добавлять элементы, из него можно удалять элементы, а также элементы списка можно перезаписывать.

```
my_list = [1, 2, 3]  # создание списка перечислением элементов
# (квадратные скобки обязательны - в отличие от создания кортежа)
my_list
```
```
[1, 2, 3]
```

Со списками работают все рассмотренные выше приемы работы с кортежами. Но ввиду изменяемости списка у него есть и свои специфические методы.

Метод `.append()` добавляет элемент в конец списка, при этом сам по себе данный метод ничего не возвращает, он просто модифицирует список.

```
my_list.append(4)
my_list
```
```
[1, 2, 3, 4]
```

Метод `.extend()` фактически выполняет сцепление (конкатенацию) списков, добавляя в конец исходного списка не один элемент, а все элементы другого списка.

```
my_list.extend([5, 6, 7])
my_list
```
```
[1, 2, 3, 4, 5, 6, 7]
```

Метод `.remove()` удаляет первое вхождение элемента. Если искомого элемента в списке нет, возникнет ошибка.

```
my_list.remove(5)  # удалим из списка значение 5
my_list
```
```
[1, 2, 3, 4, 6, 7]
```

Метод `.insert()` вставляет новый элемент в список на указанную позицию.

```
my_list.insert(4, 50)  # вставим 50 на 5-ю позицию (нумерация с 0)
my_list
```
```
[1, 2, 3, 4, 50, 6, 7]
```

Метод `.pop()` удаляет из списка элемент на заданной позиции (если не задавать позицию, то последний), при этом возвращает этот удаленный элемент.

```
print(my_list.pop(0))  # удалим 1-й элемент и убедимся, что метод его вернул
my_list
```
```
1
[2, 3, 4, 50, 6, 7]
```

Возможна перезапись элементов списка.

```
print(my_list)  # выведем список до модификации
my_list[3] = 5  # 4-й элемент 50 заменим на 5
print(my_list)  # выведем список после модификации
```
```
[2, 3, 4, 50, 6, 7]
[2, 3, 4, 5, 6, 7]
```

Особенности работы присваивания с изменяемыми объектами

Метод `.copy()` создает *копию списка в памяти*. Этот момент рассмотрим подробнее.

Операция присваивания = с изменяемыми объектами работает не так, как с неизменяемыми.

```
a = 15  # в переменную a поместим число 15
variable = [1, 2, 9]  # в переменную variable поместим список
# (не забудем, что список – объект изменяемый)
a = variable  # перезапишем переменную a, поместив в нее значение variable
```

Создалась ли в результате выполненного кода в переменной `a` копия списка из переменной `variable`? Нет! Переменные `a` и `variable` теперь указывают на *один и тот же объект в памяти* (рис. 39).

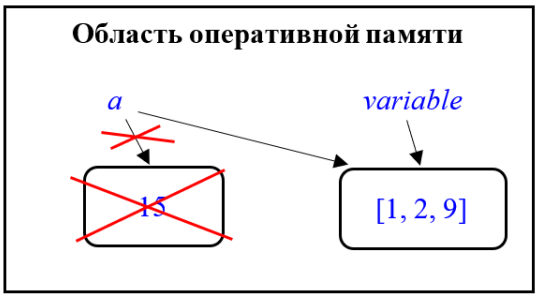


Рис. 39. Назначение ссылки на изменяемый объект в результате присваивания

Если изменим список в переменной `a`, изменим список и в переменной `variable`, потому что это один и тот же список.

```
a.pop()  # удалим последний элемент (3) из a
print(a, variable)  # видим, что он удалился и из variable тоже
```
```
[1, 2] [1, 2]
```

```
# метод списка copy() создает новый объект в памяти - копию списка
variable = [1, 2, 9]
a = variable.copy()
# теперь a и variable - это 2 разных списка. Модифицируем a
a.pop()
print(a, variable)  # как видим, список в variable остался неизменным
```
```
[1, 2] [1, 2, 9]
```

5.7. Преобразование типов коллекций

Коллекции, так же как и иные типы данных, подвержены преобразованию типов. Зачастую коллекцию одного типа можно преобразовать в коллекцию другого типа. Это *не абсолютно универсальное правило*: нельзя преобразовать что угодно во что угодно. Например, список не удастся превратить в строку, элементы которой в точности соответствуют элементам исходного списка (не забудем, что строка состоит из символов, а список может состоять из значений любого типа). Однако многие комбинации исходных и результирующих типов работают. Рассмотрим некоторые из них:

```
tuple([1, 2, 3])  # преобразование списка в кортеж
```
```
(1, 2, 3)
```

```
list((1, 2, 3))  # преобразование кортежа в список
```
```
[1, 2, 3]
```

```
list(range(1, 11))  # преобразование диапазона в список
```
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
tuple("привет")  # преобразование строки в кортеж
# (элементами кортежа являются символы строки)
```
```
('п', 'р', 'и', 'в', 'е', 'т')
```

Здесь же уместно упомянуть и *булификацию* коллекций. Правило, по которому она осуществляется, нам уже известно (см. параграф 3.4): *непустые* коллекции интерпретируются как `True`, *пустые* – как `False`. Приведем несколько примеров в дополнение к рассмотренным ранее:

```
bool("Привет!")  # непустая строка: True
bool(" ")  # непустая строка из одного пробела: True
bool("")  # пустая строка: False
bool([1, 2, 3])  # непустой список: True
bool(range(10, 0))# пустой диапазон (последовательность от 10 до 0): False
```

5.8. Передача в функцию элементов коллекции как аргументов

Выше заходила речь о распаковке «лишних» элементов кортежа в список с помощью звездочки `*`. Рассмотрим еще один механизм, в котором применяется звездочка. С ее помощью можно передавать коллекцию в функцию не как единственный комплексный объект, а как набор аргументов, которыми выступают отдельные элементы коллекции.

```
spisok = ["Veni", "vidi", "vici"]  # создадим список строк
print(*spisok)  # передадим эти строки в функцию как отдельные аргументы
```
```
Veni vidi vici
```

Команда выше аналогична следующей, но короче и универсальнее:

```
print(spisok[0], spisok[1], spisok[2])
```
```
Veni vidi vici
```

```
# используем пользовательские разделитель и строку окончания вывода
print(*spisok, sep=", ", end="!")
```
```
Veni, vidi, vici!
```

Работает рассмотренный механизм с любой функцией, а не только с `print()`. Однако следует иметь в виду, что если функция ожидает на входе определенное количество аргументов, то и передаваемая со звездочкой коллекция должна содержать такое же количество элементов, иначе возникнет несоответствие и ошибка выполнения.

5.9. Оператор цикла for

Оператор `for` – это еще один оператор цикла в Python (помимо уже рассмотренного `while`). Он предназначен для *обхода коллекций* (итерируемых объектов), то есть поочередного обращения к каждому их элементу. Данный оператор записывается в виде следующей конструкции:

```
for <переменная - параметр цикла> in <коллекция>:
  команды тела цикла
```

На каждом шаге цикла в переменную, являющуюся параметром цикла, попадает очередной элемент перебираемой коллекции.

```
mylist = [2, 4, 6, 8, 11]  # создадим список из пяти чисел
for elem in mylist:  # переберем его элементы один за другим
  print(elem)  # каждый элемент выведем на экран
```

```
2
4
6
8
11
```

```
# переберем строку посимвольно
phrase = "Широка страна моя родная!"
for symb in phrase:
  print(f"Символ '{symb}' встречается в строке {phrase.count(symb)} раз")
```

```
Символ 'Ш' встречается в строке 1 раз
Символ 'и' встречается в строке 1 раз
Символ 'р' встречается в строке 3 раз
Символ 'о' встречается в строке 3 раз
Символ 'к' встречается в строке 1 раз
Символ 'а' встречается в строке 4 раз
Символ ' ' встречается в строке 3 раз
Символ 'с' встречается в строке 1 раз
...
```

Очень часто `for` вместе с диапазоном `range` используют для организации *арифметических циклов*.

```
# создадим цикл, который выполнится n раз
n = int(input("Сколько раз выполнить цикл? "))

for i in range(n):
  print(f"{i + 1}-й шаг цикла")
```

```
Сколько раз выполнить цикл? 7
1-й шаг цикла
2-й шаг цикла
3-й шаг цикла
4-й шаг цикла
5-й шаг цикла
6-й шаг цикла
7-й шаг цикла
```

Как и в случае с `while`, в теле цикла `for` допустимо использование операторов `break` и `continue`.

5.10. Задания для самостоятельной работы

Решите следующие задачи.

1. Сгенерируйте указанную последовательность чисел с помощью функции `range()`: –10, –7, –4, –1, 2, 5, 8, 11, 14. Сформируйте кортеж из квадратов элементов этой последовательности, а также список значений функции $y = x^3 + 2x - 5$, где x – элемент последовательности. Для перебора последовательности используйте оператор `for`. Учитывая, что кортеж – неизменяемая коллекция (в него нельзя добавлять новые элементы), подумайте, как можно в процессе обхода последовательности сконструировать кортеж квадратов.

2. Напишите программу для нахождения суммы четных элементов списка целых чисел. Длину списка и числа введите с клавиатуры. Решение задачи осуществите в два этапа:

1) сформируйте последовательность в виде списка;
2) выполните обработку списка в соответствии с заданием.

6. Вложенные коллекции

Данная глава продолжает знакомить читателя с особенностями использования и обработки коллекций. В ней описываются сложные иерархические структуры данных, которые можно конструировать при помощи вложенных друг в друга коллекций.

6.1. Двумерные массивы в Python

Мы уже знаем, как в Python можно хранить не просто одно скалярное значение (например, число или логическое значение), а несколько таких значений в виде последовательности. Для этого предназначены специальные типы данных – *коллекции*. Наверное, наиболее используемой коллекцией в Python является *список*, потому что он изменяемый и может содержать в себе объекты любого типа. С помощью списка можно хранить упорядоченный одномерный набор объектов. Такие структуры данных еще называют *рядами, одномерными массивами, векторами* (см. рис. 38). А как быть, если нужно хранить не одномерный ряд значений, а **таблицу** (матрицу, двумерный массив)?

Таблицу можно представить как *последовательность* строк, причем каждая строка в свою очередь является *последовательностью* ячеек. Можно сказать, что таблица – это **последовательность последовательностей** элементов. Если в этом словосочетании заменить слово «последовательность» на слово «список», получится **список списков** – структура данных, с помощью которой можно создать таблицу в Python (рис. 40).

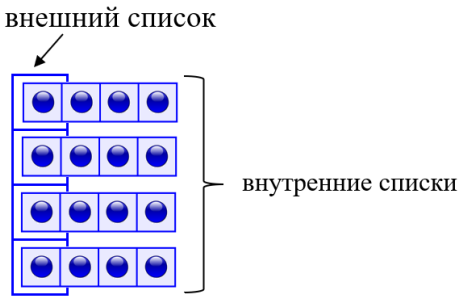


Рис. 40. Список списков

Пусть имеется таблица (рис. 41).

3	6	9	12
a	b	c	d
Истина	Ложь	Истина	Истина
0,1	0,2	0,3	0,4

Рис. 41. Таблица значений разных типов

Опишем ее в Python:

```
# поместим внутренние списки в переменные
internal_list_1 = [3, 6, 9, 12]
internal_list_2 = ["a", "b", "c", "d"]
internal_list_3 = [True, False, True, True]
internal_list_4 = [0.1, 0.2, 0.3, 0.4]

# зададим внешний список перечислением элементов (внутренних списков)
# иногда длинные команды удобнее записывать в несколько строк
# (например, перечислили элементы списка не в строку, а в столбец)
external_list = [internal_list_1,
                 internal_list_2,
                 internal_list_3,
                 internal_list_4]

# посмотрим, как выглядит двухуровневый список при его выводе на экран
external_list
```

```
[[3, 6, 9, 12],
 ['a', 'b', 'c', 'd'],
 [True, False, True, True],
 [0.1, 0.2, 0.3, 0.4]]
```

Для хранения внутренних списков промежуточные переменные можно и не использовать.

```
# обратите внимание на скобки в начале и в конце выражения:
# в начале [[ первая открывающая скобка относится ко внешнему списку,
# вторая - к первому внутреннему,
# в конце ]] первая закрывающая скобка относится к последнему
# внутреннему списку, вторая - к внешнему
external_list = [[3, 6, 9, 12],
                 ["a", "b", "c", "d"],
                 [True, False, True, True],
                 [0.1, 0.2, 0.3, 0.4]]

# а можно записать и в таком более наглядном виде,
# он не позволит запутаться со скобками
external_list = [
  [3, 6, 9, 12],
  ["a", "b", "c", "d"],
  [True, False, True, True],
  [0.1, 0.2, 0.3, 0.4]
]
```

Есть еще один способ создать внешний список: не одной командой за раз, а последовательным добавлением элементов.

```
external_list = []  # сначала создадим пустой список

external_list.append(internal_list_1)  # добавим в конец внутренний список 1
external_list.append(internal_list_2)  # и все остальные
external_list.append(internal_list_3)
external_list.append(internal_list_4)
```

Не станем забывать, что элементом списка может быть *объект любого типа* (например, кортеж).

```
# добавим в нашу таблицу еще одну строку в виде кортежа
external_list.append((10, 20, 30, 40))
# в данном случае при задании кортежа перечислением элементов
# скобки обязательны!
# сравните 2 записи:
#   external_list.append((10, 20, 30, 40))
# и
#   external_list.append(10, 20, 30, 40)
# вторая команда означает, что мы вызываем метод append()
# с 4 числовыми аргументами 10, 20, 30, 40, и она не сработает,
# потому что метод append() принимает только один аргумент
# (добавляемый объект), а не 4

external_list
```

```
[[3, 6, 9, 12],
 ['a', 'b', 'c', 'd'],
 [True, False, True, True],
 [0.1, 0.2, 0.3, 0.4],
 (10, 20, 30, 40)]
```

Можно ли *список* списков превратить в *кортеж* списков? Можно! Обратите внимание на внешние скобки – они стали круглыми, что указывает на тип данных объектов «кортеж».

```
tuple(external_list)
```

```
([3, 6, 9, 12],
 ['a', 'b', 'c', 'd'],
 [True, False, True, True],
 [0.1, 0.2, 0.3, 0.4],
 (10, 20, 30, 40))
```

Функция преобразования типа `tuple()` применилась только к *внешнему* списку (рис. 42).

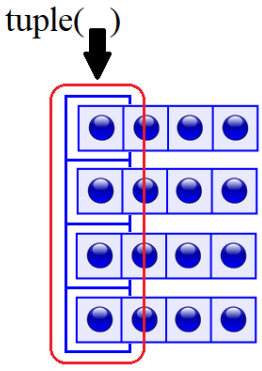


Рис. 42. Изменение типа внешней коллекции

Это логично, потому что для Python список списков не более чем *список*, то есть упорядоченный *одномерный* набор объектов. То, что эти вложенные объекты в свою очередь являются коллекциями, в данном случае не важно. Можно

изменить и тип вложенных коллекций, однако для этого придется перебирать их в цикле как элементы внешнего списка и каждый обрабатывать по отдельности.

Список списков (и любая коллекция коллекций вообще) – это структура, не имеющая строгих ограничений формы. В примере выше с помощью списка списков была описана *таблица*, то есть прямоугольная матрица $n \times m$. Все строки этой таблицы (всего n) имеют одинаковую длину (равное количество элементов) m. Однако никто не запретит в качестве вложенных объектов списка использовать коллекции *разной длины* и даже не коллекции вовсе, а простые *скалярные значения.* Ранее мы убедились, что элементами коллекции могут быть другие коллекции. Но хранение вложенных коллекций не накладывает ограничений на допустимые типы элементов внешней коллекции.

```
# добавим во внешний список:
ext_list = [
  tuple(range(3)),  # кортеж целых чисел длины 3
  42.875,  # вещественное число
  list("привет")  # список символов длиной 6
]
ext_list
```

```
[(0, 1, 2), 42.875, ['п', 'р', 'и', 'в', 'е', 'т']]
```

Очень важно на данном этапе понять, что Python позволяет конструировать объекты любой сложности и *степени вложенности*. Почему мы создаем только списки списков? Почему бы не создать **список кортежей списков кортежей списков списков**? Сконструируем такой объект, а, чтобы не запутаться в его структуре, объект каждого уровня составим из одного элемента:

```
too_complex_list = [([([[0]],)],)]
too_complex_list
```

```
[([([[0]],)],)]
```

При выводе выглядит такая структура достаточно специфически. Что за ужасное нагромождение скобок? Разберемся. Для этого представим данную структуру в более удобоваримом виде (рис. 43).

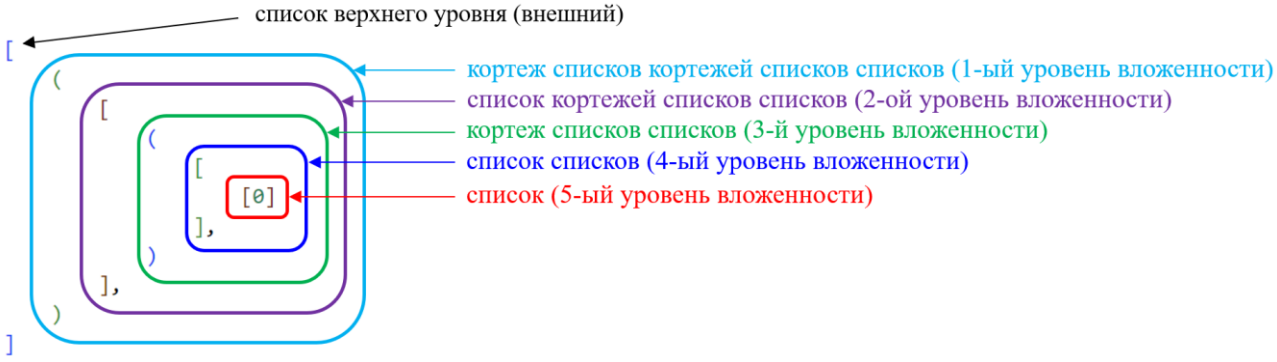


Рис. 43. Многоуровневая структура из вложенных коллекций

Почему в конце записи некоторых объектов стоят запятые? Вспомним, как задается кортеж из одного элемента: `(<значение>,)` – в отличие от списка из одного элемента: `[<значение>]`. На шестом уровне вложенности находится простое целочисленное значение – 0 (элемент списка).

Конечно, эту структуру данных мы привели лишь как демонстрацию того, что подобные сложные конструкции существуют и для Python чем-то недопустимым не являются. На практике работать с такими многоуровневыми коллекциями очень тяжело. Часто для хранения сложных иерархий данных в Python применяются *словари* (`dict`), об этом типе данных поговорим в следующих главах. Также стоит отметить существование специальных типов для хранения многомерных массивов (тензоров) и таблиц. Чтобы включить в программе поддержку этих типов, необходимо использование библиотек, расширяющих стандартные возможности языка, а именно NumPy (поддерживает многомерные массивы `ndarray`) и pandas (поддерживает табличный тип данных `DataFrame`). Возможности данных библиотек обсудим во втором разделе пособия.

6.2. Обращение к внутренним элементам вложенных коллекций

Рассмотрим вопрос о том, как получить доступ к внутренним элементам списка списков.

```
ext_list  # вспомним содержимое данного списка
```
```
[(0, 1, 2), 42.875, ['п', 'р', 'и', 'в', 'е', 'т']]
```

Этот список состоит из трех элементов:

1. Кортеж `(0, 1, 2)`.
2. Вещественное число `42.875`.
3. Список символов `['п', 'р', 'и', 'в', 'е', 'т']`.

```
print(ext_list[0])  # получим кортеж как обычный 1-й элемент списка
print(ext_list[1])  # получим число как 2-й элемент списка
print(ext_list[2])  # получим список символов как 3-й элемент списка
```
```
(0, 1, 2)
42.875
['п', 'р', 'и', 'в', 'е', 'т']
```

Вспомним о *распаковке элементов* в переменные (это работает с любыми коллекциями, а не только с кортежами):

```
elem1, elem2, elem3 = ext_list  # выполним распаковку списка в переменные
# теперь эти переменные содержат значения - элементы списка ext_list
elem1  # выведем содержимое 1-й переменной
```
```
(0, 1, 2)
```

Строго говоря, утверждение «переменные содержат элементы списка» не вполне точно. Можно вспомнить о том, как работает оператор присваивания с изменяемыми и неизменяемыми объектами (см. параграф 5.6). В нашем примере `elem1`, `elem2` хранят *копии* первого и второго элементов внешнего списка соответственно, потому что это кортеж и число, то есть значения *неизменяемых* типов. А вот `elem3` является *ссылкой* на третий элемент списка, потому что это список – *изменяемый* объект. Однако в рассматриваемом случае можно об этих тонкостях не задумываться. Мы поместили элементы внешнего списка в отдельные переменные, теперь к этим объектам можно обращаться по именам переменных:

```
elem1[1]  # обратимся ко 2-му элементу кортежа в переменной elem1
```
```
1
```

```
# кортеж elem1 хранит значение 1-го элемента списка ext_list,
# и к нему мы ранее обращались так: ext_list[0]
# теперь обратимся ко 2-му элементу кортежа как 1-го элемента списка:
ext_list[0][1]
```
```
1
```

```
print(ext_list[0][-1])  # последний элемент 1-го элемента списка (кортежа)
print(ext_list[-1][3])  # 4-й элемент последнего элемента списка (списка)
print(ext_list[2][2])  # 3-й элемент 3-го элемента списка (списка)
```
```
2
в
и
```

```
# а что со 2-м элементом внешнего списка? Это вещественное число
ext_list[1]  # обратимся к этому числу
```
```
42.875
```

```
ext_list[1][0]  # попытка же обратиться к 1-му элементу числа
# приведет к ошибке, потому число - не коллекция и не содержит элементов
```
```
...
TypeError: 'float' object is not subscriptable
```

```
# вспомним о многоуровневой структуре, описанной ранее
too_complex_list
```
```
[([([[0]],)],)]
```

```
# обратимся к числовому значению в ее глубине (всюду указываем 1-й элемент,
# потому что на всех уровнях элемент только один)
too_complex_list[0][0][0][0][0][0]
```
```
0
```

```
# но нулю одиноко, добавим в этот глубоко спрятанный список еще два числа
too_complex_list[0][0][0][0][0].extend([1, 2])
too_complex_list  # теперь внутри находится список чисел длины 3
```
```
[([([0, 1, 2]],)],)]
```

6.3. Поэлементный обход вложенных коллекций

Мы знаем, что для поэлементного обхода коллекций существует специальный тип оператора цикла – `for`. Поможет он и в переборе элементов сложных структур данных. Вновь обратимся к аналогии с таблицей.

Как перебрать все ячейки таблицы? Таблица – это последовательность строк, а каждая строка – последовательность ячеек. Обходить последовательность мы умеем. Как минимум труда не вызовет перебор строк. Что является перебираемым элементом при переборе строк? Очевидно, сама строка и является.

```
external_list  # вспомним данный список, по форме соответствующий таблице 5x4
```
```
[[3, 6, 9, 12],
 ['a', 'b', 'c', 'd'],
 [True, False, True, True],
 [0.1, 0.2, 0.3, 0.4],
 (10, 20, 30, 40)]
```

```
# переберем строки этой таблицы (то есть элементы внешнего списка)
for row in external_list:
  print(row)  # каждую строку (элемент) выведем на экран
```
```
[3, 6, 9, 12]
['a', 'b', 'c', 'd']
[True, False, True, True]
[0.1, 0.2, 0.3, 0.4]
(10, 20, 30, 40)
```

На каждом шаге цикла в переменную `row` попадает очередной элемент внешнего списка, то есть *коллекция* (список или кортеж). Почему бы эту коллекцию тоже не перебрать поэлементно? Конечно, для этого нам вновь понадобится цикл `for`. Только запускать его мы станем не один раз, а несколько – для каждой строки таблицы.

```
for row in external_list:  # внешний цикл - перебирает строки
  for cell in row:  # внутренний цикл - перебирает ячейки строки
    print(cell)  # выводим значение ячейки на экран
```
```
3
6
9
12
a
b
c
d
True
...
```

Все ячейки (элементы внутренних коллекций) вывелись в столбик – по одной в строке. Это обусловлено тем, что функция `print()` в конце вывода по

умолчанию добавляет символ переноса строки `\n` (это нам уже известно из параграфа 3.6). Соотнесем результат вывода с содержимым внешнего списка `external_list` (рис. 44).

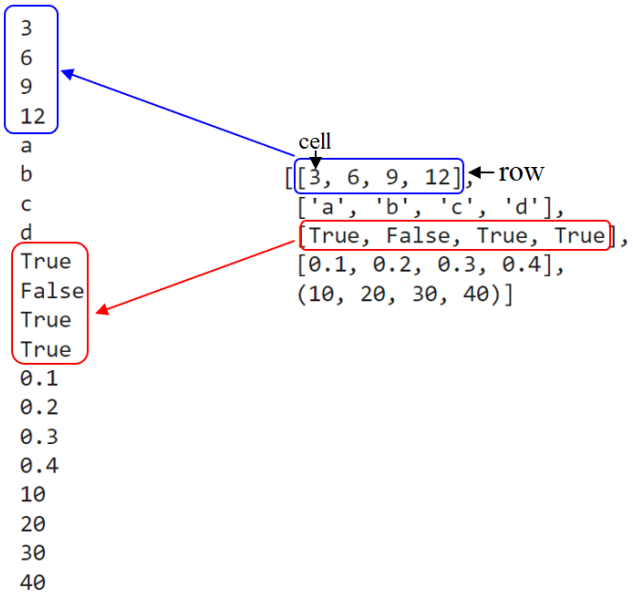


Рис. 44. Содержимое многоуровневого списка при поэлементном выводе

Хорошо бы для красоты вывести эту таблицу в виде, более похожем на таблицу: в несколько столбцов. Для этого элементы каждой строки будем выводить в строку через пробел, а по окончании строки выводить символ переноса:

```
for row in external_list:  # внешний цикл - перебирает строки
  for cell in row:  # внутренний цикл - перебирает ячейки строки
    # выводим значение ячейки на экран, в конце вывода - пробел
    print(cell, end=" ")
  # в конце вывода элементов строки - символ переноса
  print()  # (его выведет print() без параметров)
```

```
3 6 9 12
a b c d
True False True True
0.1 0.2 0.3 0.4
10 20 30 40
```

Следует обратить внимание на отступ в строке со второй функцией `print()`: эта команда относится к телу внешнего цикла, а не внутреннего, и находится в блоке кода первого `for`.

Аналогичного вывода можно достичь без использования вложенного цикла, вспомнив о механизме передачи элементов коллекции в функцию как отдельных аргументов и том факте, что функция `print()` по умолчанию выводит объекты через пробел.

```
# данная запись короче и изящнее
for row in external_list:
  print(*row)
```

```
3 6 9 12
a b c d
True False True True
0.1 0.2 0.3 0.4
10 20 30 40
```

Несмотря на прямоугольную форму нашей таблицы (все строки имеют равную длину), вывод все равно не очень красив. Воспользуемся для разделения ячеек вместо пробела символом табуляции `\t`:

```
for row in external_list:
  for cell in row:
    print(cell, end="\t")
  print()
```

```
3	6	9	12
a	b	c	d
True	False	True	True
0.1	0.2	0.3	0.4
10	20	30	40
```

Этот вид куда как больше соответствует табличному представлению.

```
# того же результата можно добиться и так:
for row in external_list:
  print(*row, sep="\t")
```

```
3	6	9	12
a	b	c	d
True	False	True	True
0.1	0.2	0.3	0.4
10	20	30	40
```

Попрактикуемся в применении вложенных циклов, выведя на экран таблицу умножения. Значения в ячейках этой таблицы представляют собой парные произведения всех чисел из диапазона от 1 до 9. Можно сказать, что это *произведения номеров строки и столбца* (рис. 45).

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Рис. 45. Таблица умножения

Можно перебрать все номера строк (от 1 до 9) и для каждой строки перебрать номера столбцов (от 1 до 9). Номера строк и столбцов представляют собой монотонную последовательность, ее можно сгенерировать функцией `range()`:

```
for i_row in range(1, 10):  # перебираем номера строк
  for i_col in range(1, 10):  # для каждой строки перебираем номера столбцов
    print(i_row * i_col, end="\t")
  print()
```

```
1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81
```

6.4. Функция enumerate() и for с несколькими параметрами

Функция `enumerate()` нумерует элементы коллекции. Фактически она превращает каждый элемент коллекции в кортеж из двух элементов вида `(<порядковый номер элемента начиная с нуля>, <элемент>)`. Взглянем на пример (код ниже и рис. 46).

```
names = ["Иванов", "Петров", "Сидоров"]

# результатом функции enumerate() является специфический объект - итератор
# (изучать итераторы не будем, просто воспользуемся тем, что итератор
# можно преобразовать в список)
list(enumerate(names))
```

```
[(0, 'Иванов'), (1, 'Петров'), (2, 'Сидоров')]
```

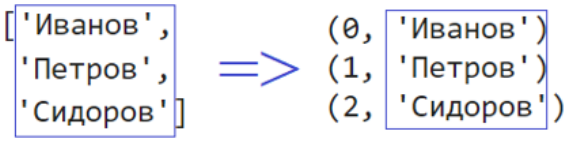


Рис. 46. Демонстрация работы функции enumerate()

Для чего такая нумерация элементов коллекции может быть полезна? Например, для того, чтобы при их переборе на каждом шаге знать порядковый номер текущего элемента. Конечно, можно хранение порядкового номера реализовать с помощью переменной-счетчика (параметра цикла), а к самому элементу обращаться по его номеру.

```
for i in range(len(names)):
  print(i, names[i])
```

```
0 Иванов
1 Петров
2 Сидоров
```

Если же мы будем перебирать не номера элементов, а элементы напрямую (как чаще всего и поступают), то для хранения номера понадобится завести отдельную переменную-счетчик.

```
i = 0  # счетчик
for name in names:
  print(i, name)
  i += 1  # не забудем вручную увеличить счетчик на 1
```

```
0 Иванов
1 Петров
2 Сидоров
```

Вот здесь и пригодится `enumerate()`. Эта функция конструирует последовательность кортежей с номером элемента и самим элементом, а значит, на каждом шаге цикла в переменную попадет такой кортеж.

```
for i_name in enumerate(names):
  print(i_name)
```

```
(0, 'Иванов')
(1, 'Петров')
(2, 'Сидоров')
```

```
# обратимся по отдельности к номеру элемента и самому элементу
for i_name in enumerate(names):
  print(i_name[0],  # номер элемента - 1-й элемент кортежа
        i_name[1])  # сам элемент - 2-й элемент кортежа
```

```
0 Иванов
1 Петров
2 Сидоров
```

Эти перебираемые в цикле кортежи можно распаковать в переменные.

```
for i_name in enumerate(names):
  i, name = i_name
  print(i, name)
```

```
0 Иванов
1 Петров
2 Сидоров
```

Данный механизм распаковки в переменные кортежей, являющихся элементами перебираемых коллекций, реализован в самом синтаксисе оператора `for`. Таким образом, `for` поддерживает *несколько параметров*.

```
for i, name in enumerate(names):
  print(i, name)
```

```
0 Иванов
1 Петров
2 Сидоров
```

Количество параметров `for` должно соответствовать размеру перебираемых кортежей.

```
# список кортежей из 3 элементов
humans = [
  (1, "Иванов", "М"),
  (2, "Петров", "М"),
  (3, "Сидорова", "Ж")
]

# соответственно используем 3 параметра цикла for
for i, name, gender in humans:
  print(i, name, gender)
```

```
1 Иванов М
2 Петров М
3 Сидорова Ж
```

Вспомним, однако, особенности распаковки кортежей с использованием звездочки `*`. С оператором `for` этот механизм тоже работает.

```
for i, *other in humans:
  print(i, other)  # список other содержит все, что не попало в i
```

```
1 ['Иванов', 'М']
2 ['Петров', 'М']
3 ['Сидорова', 'Ж']
```

```
for *other, gender in humans:
  print(other, gender)  # other содержит все, что не попало в gender
```

```
[1, 'Иванов'] М
[2, 'Петров'] М
[3, 'Сидорова'] Ж
```

6.5. Простое и глубокое копирование

Излагая нюансы работы с иерархическими структурами данных (коллекциями коллекций), нельзя не упомянуть важную особенность, связанную с копированием составных объектов в памяти. В параграфе 5.6 был описан списковый метод `.copy()`, создающий копию списка. Попробуем применить его для копирования списка списков:

```
ext_list  # вспомним содержимое списка ext_list
```

```
[(0, 1, 2), 42.875, ['п', 'р', 'и', 'в', 'е', 'т']]
```

```
ext_list_copy = ext_list.copy()  # скопируем его в другую переменную
ext_list_copy[0] = True  # перезапишем первый элемент списка-копии
ext_list  # убедимся, что оригинал не изменился
```

```
[(0, 1, 2), 42.875, ['п', 'р', 'и', 'в', 'е', 'т']]
```

```
# теперь перезапишем первый элемент внутреннего списка списка-копии
ext_list_copy[-1][0] = True
ext_list  # взглянем на оригинал
```

```
[(0, 1, 2), 42.875, [True, 'р', 'и', 'в', 'е', 'т']]
```

Как оказалось, метод `.copy()` скопировал только внешний список, а на его элемент – изменяемую коллекцию (в нашем примере – вложенный список символов) всего лишь создал ссылку. Полученная копия оказалась неполноценной: скопировался только первый уровень иерархии данных. Такое копирование называется ***простым*** (рис. 47).

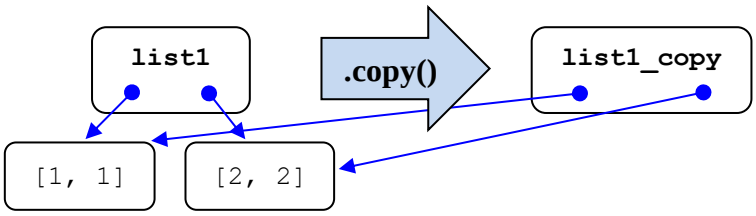


Рис. 47. Простое копирование списка списков list1 в переменную list1_copy

В Python есть и другой вид копирования – ***глубокое***. Оно создает полную независимую копию всей структуры данных, копирование осуществляется на всех уровнях (рис. 48). Чтобы выполнить глубокое копирование, необходимо воспользоваться функцией `deepcopy()` из модуля copy. В этом же модуле есть и функция для простого копирования – `copy()` (не следует путать с одноименным списковым методом).

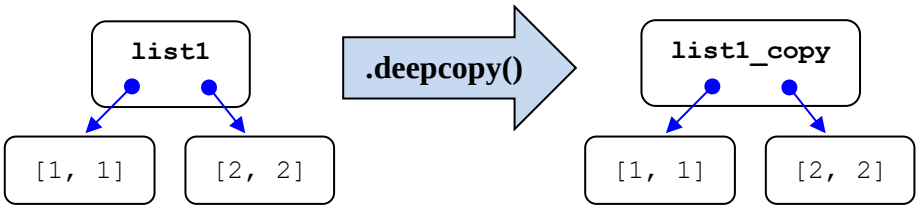


Рис. 48. Глубокое копирование списка списков list1 в переменную list1_copy

Чтобы результат приведенного ниже кода совпал с показанным, необходимо перед его выполнением заново создать список `ext_list` оригинального содержимого.

```
import copy  # подключим модуль copy
ext_list_copy = copy.deepcopy(ext_list)  # выполним глубокое копирование
ext_list_copy[-1][0] = True # перезапишем элемент внутреннего списка копии
ext_list  # убедимся, что оригинал не изменился
```

```
[(0, 1, 2), 42.875, ['п', 'р', 'и', 'в', 'е', 'т']]
```

6.6. Задания для самостоятельной работы

Решите следующие задачи:

1. Доработайте программу, конструирующую таблицу умножения, следующим образом:

1) сначала поместите таблицу в список списков;

2) затем выведите элементы этого списка в табличном виде с разделением столбцов табуляцией;

3) сделайте так, чтобы единица в первой строке и первом столбце не отображалась (см. пример ниже).

```
      2     3     4     5     6     7     8     9
2     4     6     8     10    12    14    16    18
3     6     9     12    15    18    21    24    27
...
```

2. Напишите программу для вычисления определителя матрицы третьего порядка (конструируемая таблица должна иметь размерность 3×3). Все элементы матрицы введите с клавиатуры и сохраните их в список списков. Выведите на экран саму матрицу в табличном виде и вычисленное значение определителя.

7. Обработка строк

В главе 7 подробно рассматривается строковый тип данных и особенности работы с ним. На примере строк описывается механизм срезов, поддерживаемый и другими индексированными коллекциями.

7.1. Срезы

Строка (`str`) – это упорядоченная последовательность символов. Вспомним, что нам известно об этом типе данных. Это *коллекция* (итерируемый объект), она *индексируемая* (к отдельным элементам можно обращаться по порядковому номеру), а также *неизменяемая* (отредактировать строку без создания нового объекта в памяти не получится). Строки можно складывать (сцеплять) – это операция называется «конкатенация» и реализуется оператором сложения `+`.

```
# соберем строку путем конкатенации нескольких более коротких строк
begin = "Привет"  # начало
name = input("Как тебя зовут? ")  # середина
end = "!"  # конец

# соберем строку, не забыв вставить запятую с пробелом между словами
result_str = begin + ", " + name + end
print(result_str)  # выведем собранную строку на экран
```

```
Как тебя зовут? Аристарх
Привет, Аристарх!
```

Используя f-строки, можно обойтись и без конкатенации:

```
result_str = f"{begin}, {name}{end}"
print(result_str)
```

```
Привет, Аристарх!
```

Строку можно дублировать путем умножения на натуральное число. Длину строки (количество символов) можно определить так же, как длину любой коллекции, – с помощью функции `len()`, а к символу на определенной позиции можно обратиться, используя синтаксис `[<номер символа начиная с нуля>]`. Также строки можно обходить циклом `for`.

Посмотрим, что еще можно делать со строками. И начнем с очень важного механизма – ***срезов*** (slicing). Механизм срезов позволяет *извлекать из строки фрагмент (подстроку)*. Срезы работают с любыми индексированными коллекциями (списками, кортежами), но удобнее всего рассмотреть их на примере строкового типа.

Обращение к символу на определенной позиции – это *частный случай* среза.

```
city = "Perm"

city[0]  # 1-й символ - P
city[2]  # 3-й символ - r
city[-1]  # последний (1-й с конца) символ - m
```

В общем случае синтаксис оператора среза включает *два двоеточия*.

```
[<левая граница фрагмента>:<правая граница фрагмента>:<шаг>]
```

Левая граница фрагмента – это номер (позиция) символа, которым начинается извлекаемый фрагмент.

Правая граница фрагмента – это номер (позиция) символа, которым заканчивается извлекаемый фрагмент. При этом правая граница **не включается в срез**: если она равна 10, то последним символом среза окажется символ с номером 9 (то есть десятый по счету; не забываем, что нумерация начинается с нуля). Левая и правая границы могут *отсутствовать*. Отсутствие левой границы означает указание включить в срез строку с самого начала, а отсутствие правой – включить в срез строку до самого конца.

Шаг – это интервал, с которым производится включение символов в срез. Например, при шаге 1 включается каждый первый символ, при шаге 3 – каждый третий (левая граница при этом включается безусловно).

Результатом среза строки, в свою очередь, является строка (последовательность символов, попавших в срез). Взглянем на примеры использования срезов:

```
s = "Hello, World!"  # создадим строковую переменную

s[1:5]    # ello
s[1:5:2]  # el
s[5:1:-1] # ,oll
s[1:]     # ello, World!
s[3:-5]   # lo, W
```

Изучим в подробностях процесс формирования этих срезов (рис. 49).

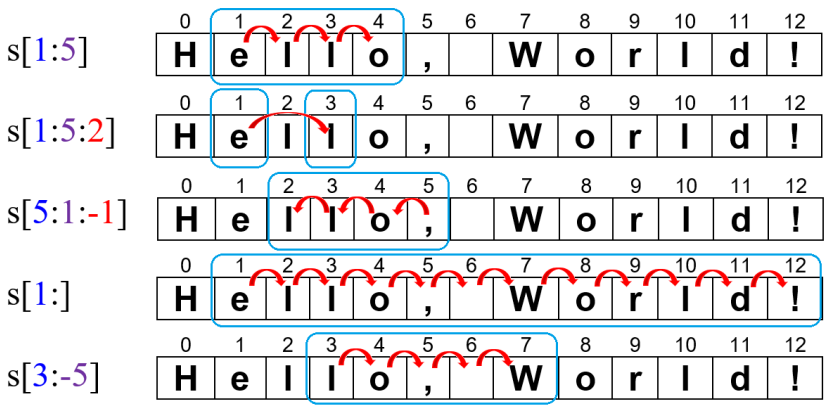


Рис. 49. Формирование срезов

На месте границ могут быть отрицательные числа: они означают отсчет позиции *с конца*. Отрицательный шаг означает, что извлечение среза происходит *справа налево* (обратите внимание на стрелки на рисунке).

```
s[:]  # извлечение фрагмента строки с самого начала до самого конца
# в случае строк такая операция смысла не имеет, а в случае списков она
# аналогична вызову метода copy(), то есть выполняет простое копирование
```
```
Hello, World!
```

```
print(s[::1])  # извлечение фрагмента с начала до конца с шагом 1
# (еще одна "бессмысленная" операция)
```
```
Hello, World!
```

```
print(s[::-1])  # извлечение фрагмента с КОНЦА до НАЧАЛА с шагом -1
# результатом будет разворот строки задом наперед
```
```
!dlroW ,olleH
```

```
print(s[5:1])  # извлечение фрагмента с 6-го символа до
# 2-го НЕ включительно с шагом 1, результат - пустая строка
# это ожидаемо, потому что при положительном шаге правая граница
# должна быть больше левой, иначе срез будет пустым
```
```

```

Сделаем небольшое отступление от изучения строкового типа и рассмотрим примеры использования срезов с *другими индексированными коллекциями*. Результатом среза списка является *список* (последовательность элементов исходного списка, попавших в срез):

```
my_list = ["эне", "бене", "раба", "квинтер", "финтер", "жаба"]
my_list[1:-1]  # фрагмент списка из элементов со 2-го по предпоследний
```
```
['бене', 'раба', 'квинтер', 'финтер']
```

Результатом среза кортежа является *кортеж* (последовательность элементов исходного кортежа, попавших в срез):

```
my_tuple = 4, 6, 8, 3, -5, -5, 0, 9
my_tuple[:7:3]  # фрагмент кортежа из элементов с 1-го по 7-й (с шагом 3)
```
```
(4, 3, 0)
```

Результат среза диапазона (`range`) является *диапазон* (монотонная последовательность чисел, попавших в срез). Впрочем, к диапазону обычно срезы не применяют. Логичнее задать нужные параметры диапазона сразу при его конструировании.

```
range(31)[2::2]  # все четные числа от 1 до 30
```
```
range(2, 31, 2))
```

Результат в виде объекта `range` не очень нагляден, лучше преобразуем его в список:

```
list(range(31)[2::2])
```
```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

А если нужны *нечетные* числа от 1 до 30? Изменим точку отсчета – левую границу:

```
list(range(31)[1::2])
```
```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

Следует обратить внимание, что *срез не может быть «рваным»*, то есть нельзя с помощью среза извлечь из коллекции сборный фрагмент из нескольких частей. Фрагмент может быть только *непрерывным*. Например, нельзя из строки извлечь фрагмент, состоящий из подстрок из символов 2–7, 9–11 и 15–23. Можно извлечь лишь фрагмент 2–23 с определенным постоянным шагом.

7.2. Строковые методы

Познакомимся с некоторыми полезными методами класса `str`. Вспомним, что метод – это функция, «привязанная» к конкретному объекту. Входными данными этой функции всегда выступает объект, которому она принадлежит. Конечно же, методы могут иметь и аргументы, которые задаются при их вызове так же, как для обычных функций.

Мы уже упоминали, что методы могут быть *двух видов*:

1. Методы, которые не меняют свой объект. Они конструируют новое значение и возвращают его как результат своей работы.
2. Методы, которые ничего не возвращают, но изменяют объект. Результатом их работы являются изменения в объекте.

Зная, что строки относятся к *неизменяемым типам данных*, можно догадаться, что строковые методы возвращают результат своей работы, не меняя самой строки.

Метод `.find()` возвращает индекс начала первого вхождения (поиск слева направо) подстроки или -1, если подстрока не найдена. Границы поиска подстроки можно задать, по умолчанию поиск выполняется во всей строке целиком:

```
s = "Будь попрочнее старый таз, длиннее был бы мой рассказ."

s.find("бы")  # подстрока "бы" начинается на 35-м символе (нумерация с 0!)
# Будь попрочнее старый таз, длиннее БЫл бы мой рассказ.
```
```
35
```

Метод `.rfind()` делает то же самое, но ищет первое вхождение *справа налево* (то есть фактически *последнее* вхождение).

```
s.rfind("бы")  #   если искать с конца, то подстрока "бы"
# впервые встречается на 39-м символе
# Будь попрочнее старый таз, длиннее был БЫ мой рассказ.
```
```
39
```

Метод `.replace()` заменяет одну подстроку на другую. По умолчанию замена происходит во всей строке, количество первых замен можно указать в третьем аргументе.

```
s.replace("старый", "новый")  # заменим подстроку "старый" на "новый"
```
```
Будь попрочнее новый таз, длиннее был бы мой рассказ.
```

Замена подстроки на пустую строку `""` равнозначна удалению подстроки.

```
s.replace("старый ", "")  # удалим подстроку "старый " совсем
```
```
Будь попрочнее таз, длиннее был бы мой рассказ.
```

Метод `.count()` возвращает количество вхождений подстроки в строку. Если вхождений не было, вернется 0. По умолчанию поиск осуществляется по всей строке, границы поиска можно задать.

```
s.count("аз")  # сколько раз в строке встречается "аз": 2
s.count("автоваз")  # сколько раз в строке встречается "автоваз": 0
```

Метод `.strip()` удаляет все перечисленные символы в строке-аргументе слева и справа. По умолчанию (при вызове без аргументов) удаляются *пробельные символы* (собственно пробел, табуляция `\t` и перенос строки `\n`).

```
# удалим пробельные символы
"       КТО проживает на ДНЕ океана?\t\n\n".strip()
```
```
'КТО проживает на ДНЕ океана?'
```

```
s.strip("Бу.")  # удалим Б, у, точку
```
```
'дь попрочнее старый таз, длиннее был бы мой рассказ'
```

Есть еще два похожих метода: `.lstrip()` удаляет символы слева, а `.rstrip()` справа.

```
s.rstrip(".,!?-")  # удалим знаки препинания справа
```
```
'Будь попрочнее старый таз, длиннее был бы мой рассказ'
```

Метод `.split()` разбивает строку на подстроки по разделителю, возвращает *список подстрок*. По умолчанию деление выполняется по пробельным символам. Обратите внимание, что в качестве разделителя используется *вся строка-разделитель целиком*. Этим `.split()` отличается от метода `.strip()`, который удаляет все *отдельные символы*, перечисленные в строке-аргументе.

```
s.split()  # получим список слов
```
```
['Будь',
 'попрочнее',
 'старый',
 'таз,',
 'длиннее',
 'был',
 'бы',
 'мой',
 'рассказ.']
```

```
# поделим строку по вертикальной черте
"Я|Ты|Он|Она|Вместе|Дружная|Семья".split("|")
```
```
['Я', 'Ты', 'Он', 'Она', 'Вместе', 'Дружная', 'Семья']
```

С помощью метода `.split()` удобно перебирать слова текста в цикле.

```
# переберем слова предложения в цикле
for word in s.split():
  print(word, end=".. ")
```
```
Будь.. попрочнее.. старый.. таз,.. длиннее.. был.. бы.. мой.. рассказ...
```

Метод `.join()` объединяет строки из списка в одну строку со вставкой строки-заполнителя между ними. Если метод `.split()` делит строку на подстроки, то `.join()` выполняет обратную операцию. Объектом – владельцем метода является строка-заполнитель. Аргументом метода должен быть непременно *список строк*, потому что значения других типов в строку не объединить.

```
# объединим список в строку, вставив между подстроками ! с пробелом
"! ".join(["снип", "снап", "снурре", "пурре", "базелюрре"])
```
```
'снип! снап! снурре! пурре! базелюрре'
```

```
# разобьем строку на слова и сразу же объединим обратно полученный список,
# вставив между словами пробелы
" ".join(s.split())
```
```
'Будь попрочнее старый таз, длиннее был бы мой рассказ.'
```

Методы `.upper()` и `.lower()` преобразуют все буквы строки к верхнему и нижнему регистру соответственно.

```
"универ".upper()  # УНИВЕР
"ПГНИУ".lower()  # пгниу
```

Методы `.isupper()` и `.islower()` проверяют, являются ли все буквы строки заглавными и строчными соответственно. Символы, не имеющие разных регистров (цифры, знаки препинания), игнорируются.

```
"СсСр".isupper()  # нет, не все буквы заглавные: False
"СССР".isupper()  # да, все буквы заглавные: True

"Я люблю тебя, жизнь!".islower()  # нет, не все буквы строчные: False
"я люблю тебя, жизнь!".islower()  # да, все буквы строчные: True
```

Метод `.capitalize()` делает первую букву строки заглавной, а остальные строчными.

```
"кАЖЕТСЯ, Я НАЖАЛ CAPSLOCK".capitalize()
```
```
'Кажется, я нажал capslock'
```

Метод `.swapcase()` меняет регистр букв: строчные становятся заглавными, заглавные – строчными.

```
hello = "пРиВеТ кАк ДеЛа?)))"
hello.swapcase()
```
```
ПрИвЕт КаК дЕлА?)))
```

Приведем эту строку к более привычному виду, применив рассмотренные выше методы:

```
hello.replace("Т к", "Т, к").capitalize().strip(")")
```
```
' Привет, как дела?'
```

Методы `.isnumeric()` и `.isalpha()` проверяют, содержит ли строка только цифры и только буквы соответственно.

```
"347853487".isnumeric()  # да, только цифры: True
"74358ашрпрунге874".isnumeric()  # нет, не только цифры: False

"ПриветWorld".isalpha()  # да, только буквы (латинские и кириллица): True
"Hello, Мир!".isalpha()  # нет, не только буквы: False
```

Некоторые другие возможности обработки строк

Операторы `in` и `not in` проверяют вхождение подстроки в строку. Обратите внимание: `not in` и `not` – это *разные операторы*. Ключевое слово `not` выполняет операцию *отрицания* (инверсии) логического значения, а `not in` проверяет *невхождение* одного значения в другое.

```
"рассказ" in s  # проверка вхождения слова "рассказ" в строку: True
"басня" not in s  # проверка НЕвхождения слова "басня" в строку: True
```

Эти операторы работают не только со строками, но и *с итерируемыми объектами вообще* (в частности, со списками и кортежами), проверяя наличие того или иного значения в коллекции.

```
5 in [1, 2, 3, 4, 6, 7, 0]  # проверка наличия числа 5 в списке
```
```
False
```

Для работы со строками в Python существует модуль string. Он содержит некоторые полезные константы.

```
import string  # подключим модуль string

# строка - набор специальных символов, включая знаки препинания
print(string.punctuation)
```
```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

```
print(string.ascii_letters)  # строка из латинских букв
```
```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
print(string.digits)  # строка из цифр
```
```
0123456789
```

7.3. Задания для самостоятельной работы

Решите следующие задачи:

1. С клавиатуры вводится текст из нескольких слов (не менее двух):

1) посчитайте количество слов в тексте;

2) поменяйте местами первое и второе слова, результат выведите на экран в виде строки (не в виде списка или другой коллекции).

2. С клавиатуры вводится предложение из нескольких слов, разделенных пробелами. Выведите номера слов, состоящих из четного количества букв.

3. Дан текст из нескольких абзацев (абзацы разделены символом переноса строки; каждый абзац состоит минимум из одного предложения). В тексте встречаются знаки вопроса и знаки восклицания:

1) подсчитайте количество: абзацев, предложений во всем тексте, предложений в каждом абзаце;

2) замените все знаки вопроса в тексте знаками восклицания и наоборот (восклицания замените вопросами).

8. Продвинутая обработка коллекций

В данной главе описываются механизмы и функции, позволяющие осуществлять сложную обработку коллекций, такую как фильтрация, сортировка, применение функции ко всем элементам коллекции и т.д.

8.1. Сортировка

Рассмотрим возможности языка Python по обработке коллекций.

Сортировка – это расположение элементов коллекции в порядке возрастания или убывания. Любую коллекцию, элементы которой можно *сравнивать между собой* и про пару этих элементов можно однозначно сказать, что один из них больше другого либо они равны, можно отсортировать. Изучим механизмы сравнения объектов разных типов.

Сравнение чисел

Процедура сравнения целых или вещественных чисел интуитивно понятна и пояснений не требует. Приведем несколько примеров:

```
a = 5
b = -3.7

print(f"{a} > {b}:", a > b)
print(f"{a} >= {b}:", a >= b)
print(f"{a} < {b}:", a < b)
print(f"{a} <= {b}:", a <= b)
print(f"{a} == {b}:", a == b)
print(f"{a} != {b}:", a != b)
```

```
5 > -3.7: True
5 >= -3.7: True
5 < -3.7: False
5 <= -3.7: False
5 == -3.7: False
5 != -3.7: True
```

Сравнение символов

Можно сказать, что при сравнении символов (строк единичной длины) б*о*льшим считается тот, который расположен «дальше по алфавиту». Это интуитивное и вполне корректное объяснение, только нужно определиться: что такое в данном случае *алфавит* и что в нем находится *дальше* – точка или цифра 0?

Все символы в Python хранятся в *специальной кодовой таблице* (именно ее можно условно назвать алфавитом), в которой каждому символу соответствует порядковый номер. Узнать этот номер можно с помощью функции `ord()`.

```
ord("ё")  # числовой код строчной буквы "ё"
```

```
1105
```

Обратное преобразование выполняет функция `chr()`.

```
print(chr(1105))  # символ, соответствующий коду 1105 (строчная "ё")
print(chr(0))  # символ, соответствующий коду 0
print(chr(77))  # символ, соответствующий коду 77
```

```
ё

M
```

Сколько же всего символов в этой таблице? Проверим: станем пытаться преобразовывать числовые коды до тех пор, пока преобразование не приведет к ошибке, что будет означать отсутствие в таблице символа с запрошенным кодом. Чтобы при возникновении ошибки выполнения программа не прекратила свою работу, применим оператор `try`:

```
# переберем очень много кодов в цикле
for code in range(9999999999999):
  try:  # попытаемся выполнить преобразование кода в символ
    chr(code)
  except ValueError:  # в случае ошибки преобразования сообщим об этом
    print(f"{code + 1}-й по счету код не удалось преобразовать в символ.")
    break  # и завершим цикл
```

```
1114113-й по счету код не удалось преобразовать в символ.
```

Правило сравнения символов между собой: больше тот, чей числовой код больше. Можно заметить, что в целом порядок расположения символов в кодовой таблице следующий:

1) пробел;
2) различные спецсимволы;
3) цифры;
4) знаки препинания;
5) заглавные латинские буквы (большие);
6) строчные латинские буквы (маленькие);
7) заглавные кириллические буквы (большие);
8) строчные кириллические буквы (маленькие);
9) иероглифы;
10) различные эмотиконы (смайлики).

```
print("Пробел", ord(" "))
print("#", ord("#"))
print("3", ord("3"))
print("?", ord("?"))
print("F", ord("F"))
print("f", ord("f"))
print("И", ord("И"))
print("и", ord("и"))
print("漢", ord("漢"))
print("😂", ord("😂"))
```

```
Пробел 32
# 35
3 51
? 63
F 70
f 102
И 1048
и 1080
漢 28450
😂 128514
```

```
# убедимся, что китайский иероглиф "python" больше, чем сердечко
"蟒" > "❤"
```
```
True
```

Сравнение индексированных коллекций

К индексированным коллекциям относятся и *строки* (ведь строка – это последовательность символов). При их сравнении действуют следующие *правила*:

1. Если первые элементы коллекций различаются, больше та коллекция, чей первый элемент больше.

```
# "ы" больше, чем латинская "A"
"ы!" > "All mimsy were the borogoves, And the mome raths outgrabe."
```
```
True
```

2. Если первые элементы коллекций одинаковы, проверяются следующие пары элементов до тех пор, пока не найдется различие.

```
"синхрофазотрон" > "синхроимпульс"  # "ф" больше, чем "и"
```
```
True
```

3. Если при сравнении пар элементов одна из коллекций закончится раньше, большей считается та, которая длиннее.

```
# вторая строка целиком включает первую, но длиннее, поэтому она больше
"его воробей" < "его воробейшество"
```
```
True
```

4. Коллекции, в которых не удалось обнаружить различий (содержат одинаковое количество одинаковых элементов), равны между собой.

```
# строки одинаковы (даже при том, что вторая строка конструируется
# из подстрок конкатенацией)
"What were they thinking?!" == "What were they" + " thinking?!"
```
```
True
```

Со списками, кортежами и любыми индексированными коллекциями эти правила работают аналогично.

```
print("Первый элемент первого списка больше:",
      [9] > [8, 7, 6, 5, 4, 3, 2, 1, "огого"])
print("5-й элемент первого списка больше:",
      [8, 7, 6, 5, 50] > [8, 7, 6, 5, 4, 3, 2, 1, "огого"])
print("Второй кортеж включает первый, но длиннее:",
      (8, 7, 6, 5) < (8, 7, 6, 5, 4, 3, 2, 1, "огого"))
print("Кортежи одинаковы:", (8, 7) + (6,) == tuple([8, 7, 6]))
```

```
Первый элемент первого списка больше: True
5-й элемент первого списка больше: True
Второй кортеж включает первый, но длиннее: True
Кортежи одинаковы: True
```

Что произойдет, если очередными сравниваемыми элементами коллекции будут *коллекции*? Элементы-коллекции будут сравниваться между собой *по тем же правилам*, то есть сначала будут сравниваться первые элементы, потом вторые и т.д.

```
# 2-й список больше, потому что "котопес" больше, чем "кот"
[1, 5, "кот", 0] < [1, 5, "котопес"]  # True
# 1-й список больше, потому что "котя" больше, чем "котопес"
# ("я" дальше по алфавиту, чем "о")
[1, 5, "котя", 0] > [1, 5, "котопес"]  # True
```

В Python можно сравнивать любые объекты, типом которых предусмотрена такая операция. Теперь, когда мы разобрались с общими принципами сравнения объектов, вернемся к сортировке коллекций.

Стандартные средства сортировки в Python представлены *двумя инструментами*:

1. Функция `sorted()`.
2. Метод списка `.sort()`.

Они обладают одинаковым набором необязательных аргументов и в целом делают одно и то же, однако функция `sorted()` принимает на вход любую коллекцию и возвращает *список* из ее отсортированных элементов, списковый же метод `.sort()` сортирует свой список, изменяя его, и ничего не возвращает.

```
# функция sorted()
print(sorted([8, 5, 30, -5, 11]))  # сортировка списка
print(sorted((5, 3, 2, 0, 99)))  # сортировка кортежа
print(sorted(range(5, -6, -1)))  # сортировка диапазона
print(sorted("Python"))  # сортировка строки
```

```
[-5, 5, 8, 11, 30]
[0, 2, 3, 5, 99]
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
['P', 'h', 'n', 'o', 't', 'y']
```

```
# метод списка sort()
my_list = [8, 5, 30, -5, 11]
my_list.sort()  # сортируем список
my_list  # как видим, список изменился
```

```
[-5, 5, 8, 11, 30]
```

Для сортировки в Python характерно такое свойство, как ***устойчивость***. Оно означает, что одинаковые элементы коллекции в отсортированном списке будут взаимно расположены так же, как они были расположены до сортировки (рис. 50).

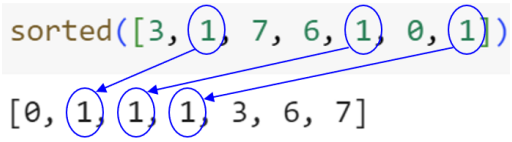


Рис. 50. Демонстрация свойства устойчивости сортировки

Можно спросить: в чем же смысл устойчивости? Единицы идут друг за другом, и какая разница, как они были расположены до сортировки, они же одинаковые! О том, где проявляется и обретает смысл устойчивость, поговорим чуть позже.

Аргумент функции (и метода) сортировки `reverse` (по умолчанию имеет значение `False`) позволяет *обратить порядок* сортировки, то есть сортировать не от меньшего к большему, а от большего к меньшему.

```
sorted(my_list, reverse=True)
```
```
[30, 11, 8, 5, -5]
```

Аргумент `key` позволяет сортировать элементы не по исходным их значениям, а по некоторым производным от них. Иначе говоря, `key` задает так называемый ***ключ сортировки*** – значение, по которому будет производиться сравнение объектов при их упорядочивании. По умолчанию (если не задавать аргумент `key`) в качестве ключа используется само значение объекта. Например, при сортировке списка строк будут сравниваться его элементы – строки. Аргумент `key` позволяет задать *функцию*, которая будет применяться к каждому элементу сортируемой коллекции и возвращать некоторое значение (*ключ*), по которому и будет осуществлена сортировка. Проще всего понять работу этого механизма на примере:

```
my_list_2 = "Python is the best language".split()  # получаем список строк
sorted(my_list_2)  # сортируем строки по их значениям
```
```
['Python', 'best', 'is', 'language', 'the']
```

```
# а если требуется отсортировать строки по их длине?
# нужно применить к каждой строке такую функцию,
# которая будет возвращать ее длину. Такая функция существует - len()
sorted(my_list_2, key=len)
```
```
['is', 'the', 'best', 'Python', 'language']
```

Что тут произошло? Разберемся. К каждому элементу списка (строке) применилась функция `len()`, вернув его длину:

```
'Python'      →    len('Python')      →    6
'is'          →    len('is')          →    2
'the'         →    len('the')         →    3
'best'        →    len('best')        →    4
'language'    →    len('language')    →    8
```

Затем произошла сортировка уже не по исходным значениям (строкам), а по *сконструированным ключам* – длинам этих строк:

```
2 → 'is'
3 → 'the'
4 → 'best'
6 → 'Python'
8 → 'language'
```

Сконструированные ключи, то есть результаты работы функции `len()`, являются *виртуальными*, они нигде не сохраняются, используются только при сортировке. Отсортированный список состоит из элементов исходной коллекции, только расположенных в том порядке, какой предполагается этими виртуальными значениями (ключами).

Обратите внимание, что при указании значения аргумента `key` функция `len()` не вызывается, *скобки писать не нужно*. Это тот случай, когда функция в Python используется как *объект*, а не как алгоритм (см. параграф 5.1). Конечно, в процессе сортировки эта функция будет вызвана для каждого элемента коллекции (этот объект будет подан на вход функции в качестве аргумента), однако для нас эти вызовы функции выполняются неявно.

```
# аргумент reverse также можно использовать
sorted(my_list_2, key=len, reverse=True)
```
```
['language', 'Python', 'best', 'the', 'is']
```

```
# еще один пример конструирования ключей:
# сортировка чисел по абсолютному значению (модулю)
sorted([1, -2, 5, -9, 7], key=abs)
```
```
[1, -2, 5, 7, -9]
```

И вот здесь обретает смысл свойство *устойчивости сортировки*. Представим себе, что в коллекции есть несколько элементов, *разных по значению*, однако *одинаковых по конструируемым ключам*.

```
pet_lover_list = ["Я", "люблю", "котов", "и", "люблю", "собак",
                  "всех", "люблю", "я", "в", "общем"]

sorted(pet_lover_list, key=len)
```

```
['Я',
 'и',
 'я',
 'в',
 'всех',
 'люблю',
 'котов',
 'люблю',
 'собак',
 'люблю',
 'общем']
```

Несколько слов имеют одинаковую длину, и после сортировки они сохранили свой взаимный порядок следования: «Я» идет перед «и», «котов» идет перед «собак», и т.д.

Такая сложная сортировка (по конструируемому ключу) полезна тогда, когда нужно некоторым специальным образом, не «в лоб» сравнивать элементы сортируемой коллекции, однако нет нужды конструировать новую коллекцию из модифицированных необходимым способом элементов, потому что кроме как для целей сортировки эти ключи больше нигде не требуются. Конструирование ключей с помощью аргумента `key` позволяет модифицировать элементы коллекции «на лету».

8.2. Некоторые другие инструменты обработки коллекций

Поиск максимального и минимального значений

Мы уже разбирали стандартные алгоритмы поиска максимума (минимума) в параграфе 4.3 про циклы. Однако в Python есть и готовые функции, выполняющие сортировку коллекций. Этой цели служат функции `max()` и `min()`.

```
my_list_4 = [-5, 11, 9, 0, 4, -8]

print(max(my_list_4))  # максимальный элемент списка: 11
print(min(my_list_4))  # минимальный элемент списка: -8
# перечисление значений как аргументов функции тоже работает
print(max(-5, 11, 9, 0, 4, -8))
```

```
11
-8
11
```

У функций поиска максимума (минимума), как и у функции сортировки, тоже есть аргумент `key`. При поиске максимума (минимума) объекты сравниваются точно так же, как при сортировке, и ключ сравнения можно динамически конструировать.

```
min(my_list_4, key=abs)  # поиск минимума по модулю
```

```
0
```

Свойство, похожее на устойчивость, здесь тоже работает: если максимумов (минимумов) несколько, функция вернет первый из них.

```
# -9 и 9 одинаковы по модулю, но результат именно -9,
# потому что это значение встречается раньше
max([-9, 0, 4, 9, -1], key=abs)
```
```
-9
```

Сумма, произведение и другие статистики

Функция `sum()` возвращает сумму всех элементов коллекции чисел.

```
sum([4, 9, 7])  # передача значений по отдельности не сработает
```
```
20
```

Для подсчета среднего, медианы и других числовых характеристик выборки можно использовать функции из модуля statistics.

```
import statistics
print("Среднее:", statistics.mean([4, 9, 7]))  # среднее арифметическое
print("Медиана:", statistics.median([4, 9, 7, 5, 2, 1]))  # медиана
# о других полезных функциях в statistics можно узнать самостоятельно
```
```
Среднее: 6.666666666666667
Медиана: 4.5
```

Произведение всех элементов коллекции можно получить с помощью функции prod() из модуля math.

```
import math
math.prod([-1, 15, 3])  # -1 * 15 * 3
```
```
-45
```

Множественные конъюнкция и дизъюнкция

Функция `all()` применима к коллекциям значений *логического типа* и возвращает `True` в том случае, если все значения истинны, иначе – `False`. Эта функция осуществляет множественную конъюнкцию. Фактически ею выполняется следующая операция:

```
<значение 1> and <значение 2> and ... and <значение n>
```

```
all((True,) * 9)  # все значения кортежа - True, результат тоже True
all([True, False, True, True])  # есть один False, результат - False
```

Вспомним множественные сравнения (см. параграф 3.9). Механизм их работы можно описать с применением множественной конъюнкции.

```
1 < 2 < 3 < 4  # то же самое: all([1 < 2, 2 < 3, 3 < 4]), результат True
1 < 99 < 3 < 4  # то же самое: all([1 < 99, 99 < 3, 3 < 4]), False
```

В качестве еще одного примера работы функции `all()`, демонстрирующего ее полезность, напишем небольшую программу-опросник:

```
answer_list = []  # список ответов (логических значений)
answer_list.append(input("Вам нравится динамическая типизация? " \
                         "(да/нет): ").strip().lower() == "да")
answer_list.append(input("Вам нравятся компилируемые языки? " \
                         "(да/нет): ").strip().lower() == "нет")
answer_list.append(input("Вам нравится простота синтаксиса? " \
                         "(да/нет): ").strip().lower() == "да")
if all(answer_list):  # если все ответы положительны
  print("Изучайте язык Python - в нем все это есть.")
else:
  print("Python не для Вас, попробуйте C++.")
```

```
Вам нравится динамическая типизация? (да/нет): да
Вам нравятся компилируемые языки? (да/нет): нет
Вам нравится простота синтаксиса? (да/нет): да
Изучайте язык Python - в нем все это есть.
```

Функция `any()` работает аналогично предыдущей, но реализует множественную *дизъюнкцию*. Она возвращает `False` в том случае, если все значения ложны, иначе `True`. Фактически этой функцией выполняется следующая операция:

```
<значение 1> or <значение 2> or ... or <значение n>
```

```
# тестирование соискателей на вакансию программиста
experience = int(input("Введите опыт работы в годах: "))
salary = int(input("Введите желаемую зарплату в тыс. руб.: "))
overtime = input("Вы готовы ради компании работать сверхурочно? " \
                 "(да/нет): ").strip().lower()

# если хотя бы одно из условий выполняется
if any([experience >= 10, salary <= 100, overtime == "да"]):
  print("Вы нам подходите! Отдел кадров на втором этаже.")
else:
  print("Мы вам перезвоним.")
```

```
Введите опыт работы в годах: 20
Введите желаемую зарплату в тыс. руб.: 150
Вы готовы ради компании работать сверхурочно? (да/нет): нет
Вы нам подходите! Отдел кадров на втором этаже.
```

Транспонирование матриц

Функция `zip()` преобразует набор коллекций, перечисленных в качестве аргументов, в набор кортежей, состоящих из элементов этих коллекций на одинаковых позициях. Проще всего понять суть этой операции, взглянув на иллюстрацию (рис. 51).

Рис. 51. Принцип работы функции zip()

Функция `zip()` возвращает объект-итератор, но его можно преобразовать в привычную нам коллекцию (список или кортеж), а также обходить в цикле.

```
matr = list(zip(["1", "2", "3"],  # каждый список передается в функцию zip()
                ["a", "b", "c"],  # как отдельный аргумент
                ["э", "ю", "я"]))
matr
```

```
[('1', 'a', 'э'), ('2', 'b', 'ю'), ('3', 'c', 'я')]
```

С помощью `zip()` и `for` с несколькими параметрами удобно перебирать элементы нескольких коллекций сразу.

```
names = ["Ваня", "Даша", "Петя"]
ages = [20, 23, 17]
for name, age in zip(names, ages):
  print(f"Имя: {name}, возраст: {age}")
```

```
Имя: Ваня, возраст: 20
Имя: Даша, возраст: 23
Имя: Петя, возраст: 17
```

Вспомнив о распаковке коллекций с помощью звездочки, можно применять `zip()` и к *коллекциям коллекций*.

```
# транспонируем матрицу из примера выше обратно
for a, b, c in zip(*matr):
  print(f"{a}\t{b}\t{c}")
```

```
1	2	3
a	b	c
э	ю	я
```

Говоря об инструментах обработки коллекций, логично еще раз упомянуть функцию `enumerate()`, которая уже была рассмотрена в параграфе 6.4.

```
# пронумеруем элементы списка кортежей, начиная с 1
for i, data in enumerate(zip(names, ages), 1):
  print(f"{i}.", *data)  # data - это кортеж вида (имя, возраст)
```

```
1. Ваня 20
2. Даша 23
3. Петя 17
```

8.3. Модуль itertools

Модуль itertools содержит средства для работы с итерируемыми объектами (коллекциями). Рассмотрим некоторые из них.

Функция `combinations()` возвращает набор всех возможных комбинаций (без повторов) элементов входной коллекции в виде кортежей.

```
import itertools

# переберем все бесповторные комбинации букв A, B, C, D по 3 элемента
# возвращаемый функцией итератор преобразуем в список
letters_combinations = list(itertools.combinations("ABCD", r=3))
letters_combinations
```

```
[('A', 'B', 'C'), ('A', 'B', 'D'), ('A', 'C', 'D'), ('B', 'C', 'D')]
```

Функция `combinations_with_replacement()` работает аналогично, только комбинации включают повторы. Если набор комбинаций обрабатывается сразу после генерации «на лету», можно его в список и не преобразовывать, работать как с обычным итерируемым объектом.

```
# переберем все повторные комбинации чисел 1, 2, 3 по 2 элемента
print(*itertools.combinations_with_replacement((1, 2, 3), r=2))
```

```
(1, 1) (1, 2) (1, 3) (2, 2) (2, 3) (3, 3)
```

Функция `permutations()` генерирует бесповторные комбинации, но учитывает порядок следования элементов.

```
print(*itertools.permutations((1, 2, 3), r=2))
```

```
(1, 2) (1, 3) (2, 1) (2, 3) (3, 1) (3, 2)
```

Функция `accumulate()` «накапливает» результат выполнения заданной функции и возвращает последовательность из таких «накопленных» результатов. Формат вызова функции выглядит так:

```
accumulate(<коллекция>, <функция двух аргументов>)
```

Чтобы понять, как работает данная функция, разберем подробно состав последовательности – ее результата. Введем обозначения: $\{x_i\}$ – входная последовательность длины n, $\{y_i\}$ – результат функции `accumulate()`, fun – заданная функция двух аргументов. Последовательность $\{y_i\}$ состоит из *следующих элементов*:

y_1 – первый член входной последовательности (x_1).

y_2 – результат выполнения функции $fun(y_1, x_2)$.

y_3 – результат выполнения функции $fun(y_2, x_3)$.

…

y_n – результат выполнения функции $fun(y_{n-1}, x_n)$.

Если применяемую функцию fun явно не задавать, `accumulate()` возвращает последовательность «нарастающим итогом», то есть в качестве fun используется функция суммирования двух значений. Получим список накопленных сумм чисел от 1 до 5:

```
list(itertools.accumulate(range(1, 6)))  # 1 + 2 + 3 + 4 + 5
```
```
[1, 3, 6, 10, 15]
```

Вычислим результат выражения $((2^3)^4)^5$. Для этого сгенерируем аналогичную предыдущей последовательность, только вместо суммы будет выполняться возведение в степень, для чего используем функцию `math.pow()`.

```
import math

power_list = list(itertools.accumulate(range(2, 6), math.pow))
print("Список результатов возведения в степень:", power_list, sep="\n")
print("Итоговый результат в целочисленном виде:", int(power_list[-1]))
# вспомним, что возведение в степень – правоассоциативная операция,
# поэтому для получения желаемого нами результата нужны скобки
print("Результат ручного возведения в степень:", ((2 ** 3) ** 4) ** 5)
```
```
Список результатов возведения в степень:
[2, 8.0, 4096.0, 1.152921504606847e+18]
Итоговый результат в целочисленном виде: 1152921504606846976
Результат ручного возведения в степень: 1152921504606846976
```

8.4. Функции map(), filter() и генерация списка

Функция `map()` позволяет применить заданную функцию к каждому элементу коллекции. Результатом ее является последовательность (итератор) из преобразованных элементов входной коллекции.

Например, есть список чисел. Требуется получить список их квадратных корней. Для этого можно применить к каждого числу функцию `math.sqrt()` (рис. 52).

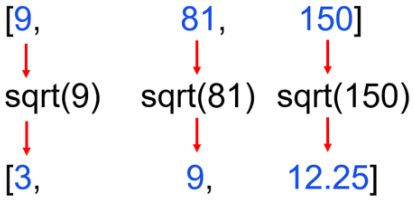


Рис. 52. Принцип работы map(): применение функции ко всем элементам списка

```
from math import sqrt  # функция для извлечения квадратного корня

numbers = [9, 81, 150]  # исходный список чисел
square_roots = list(map(sqrt, numbers))  # результат map() – итератор,
# преобразуем его в привычный список
square_roots
```
```
[3.0, 9.0, 12.24744871391589]
```

Функция `filter()` позволяет выполнить отбор элементов исходной коллекции по заданному условию (отфильтровать). Проверка условия осуществляется с помощью функции, которая применяется к элементу коллекции и возвращает `True` или `False`. Результатом `filter()` является последовательность (итератор) из отобранных элементов входной коллекции.

В качестве примера использования функции отберем только те символы в строке, которые являются цифрами. Для этого применим строковый метод `.isnumeric()`, но применим его *как обычную функцию*, не привязанную к конкретному объекту. В ООП такие методы называются *статическими*, и подробно рассматривать их не будем.

```
text = """Значительное увеличение расходов федерального бюджета на
здравоохранение произошло в 2020 году - в 1,87 раза, с 713 млрд до
1,3 трлн руб."""

digits_list = list(filter(str.isnumeric, text))
print("Список цифр:", digits_list)
```

```
Список цифр: ['2', '0', '2', '0', '1', '8', '7', '7', '1', '3', '1', '3']
```

```
# преобразуем список символов (строк) в список целых чисел
print("Список чисел:", list(map(int, digits_list)))
```

```
Список чисел: [2, 0, 2, 0, 1, 8, 7, 7, 1, 3, 1, 3]
```

```
# совместим отбор и преобразование элементов коллекции в одной команде
print(*map(int, filter(str.isnumeric, text)))
```

```
2 0 2 0 1 8 7 7 1 3 1 3
```

Генерация списка (list comprehension)

Функции `map()` и `filter()` с преобразованием результата в список можно заменить *одной синтаксической конструкцией*. Эта конструкция называется *list comprehension*, в русскоязычных источниках для ее обозначения встречаются следующие термины: генерация (генератор) списка, списковое включение, абстракция списка.

```
[<выражение> for <переменная> in <коллекция> if <условие>]
```

Такая команда позволяет последовательно обойти коллекцию, для каждого ее элемента, удовлетворяющего условию, вычислить выражение и вернуть результат в виде списка. Фильтрующая часть (с ключевым словом `if`) может отсутствовать.

```
# простейшая форма генерации списка: просто создается новый список
# из элементов исходной коллекции (результат - всегда список)
[x for x in (2, 5, 9, 11, 19)]
```

```
[2, 5, 9, 11, 19]
```

Основные отличия *генерации списка* от функций `map()` и `filter()`:

✓ совмещает их функциональность в одной синтаксической конструкции;

✓ не требует для обработки элементов и отбора использовать непременно функции, могут применяться выражения любого состава;

✓ результат представлен в виде списка сразу, его не нужно явно преобразовывать из итератора.

```
# сгенерируем список квадратных корней чисел от 2 до 10
[x ** 2 for x in range(2, 11)]
```
```
[4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
# сгенерируем список квадратных корней НЕЧЕТНЫХ чисел от 2 до 10
[x ** 2 for x in range(2, 11) if x % 2 != 0]
```
```
[9, 25, 49, 81]
```

```
# извлечем числа из текста (см. пример выше) с помощью генерации списка
[int(x) for x in text if x.isnumeric()]
```
```
[2, 0, 2, 0, 1, 8, 7, 7, 1, 3, 1, 3]
```

8.5. Задания для самостоятельной работы

Решите следующие задачи:

1. С клавиатуры в одну строку вводится список вещественных чисел, разделенных запятой. Преобразуйте эту строку в список значений вещественного типа. Используйте два способа: функцию `map()` и генерацию списка (результаты должны быть идентичны).

2. С клавиатуры вводится предложение из нескольких слов. Отсортируйте слова в предложении по длине. Результатом должна быть строка, а не список.

3. С клавиатуры вводится «рваная» (неправильная) матрица чисел (то есть строки могут состоять из разного количества элементов). Организуйте ввод значений любым способом, результат сохраните в список списков. С помощью функций `max()` и `min()` выведите на экран самый длинный и самый короткий вложенные списки.

4. С клавиатуры вводится несколько предложений, нужно сохранить их в список. Отсортируйте слова в каждом предложении по убыванию первой буквы, а затем отсортируйте предложения по количеству слов. Регистр символов не должен оказывать влияния на сортировку.

9. Подробно о функциях

В данной главе подробно рассматривается создание пользовательских функций. Описываются различные возможности языка по работе с функциями.

9.1. Пользовательские функции

Мы уже разбирались с понятием функции в параграфе 3.5, а также активно использовали стандартные функции. Пришло время глубже погрузиться в данную тему и научиться писать функции самим.

Функция в программировании – это подпрограмма, то есть программа, которую можно вызывать из другой программы. Мы знаем, что *программа* – это описание (реализация) некоторого алгоритма на языке программирования. Программа имеет входные и выходные данные. Функция – это тоже программа, которая имеет входные и выходные данные. Ранее мы применяли готовые функции, написанные для нас профессиональными разработчиками.

```
# функция round() округляет вещественное число до заданного разряда
x = 1.23456789
y = round(x, 2)
y
```

```
1.23
```

Данная функция принимает на вход два значения (аргументы x и 2) и возвращает значение-результат y (рис. 53).

Рис. 53. Схема работы стандартной функции round()

Но что за «магия» происходит внутри функции `round()`, как именно число оказалось округлено до двух разрядов? Нам это не очень интересно. Готовая функция для нас является «черным ящиком»: просто пользуемся результатами ее работы, не разбираясь в том, как именно они получились.

Другие примеры готовых функций: `int()`, `float()`, `math.sqrt()`, `math.sin()`, `itertools.accumulate()`, `input()`, `print()`. При этом функции ввода-вывода являются специфическими, они отличаются от других: функция `input()` входные данные получает с клавиатуры (хотя можно указать строку – приглашение к вводу, но этот аргумент необязателен), а результатом работы функции `print()` является вывод данных на экран, сама функция ничего не возвращает.

```
# функция list() принимает на вход коллекцию
# и возвращает список из элементов этой коллекции
input_data = "Python"
output_data = list(input_data)
output_data
```
```
['P', 'y', 't', 'h', 'o', 'n']
```

Но что, если нам нужно написать *свою собственную функцию*? Зачем это может понадобиться? Давайте рассмотрим такой пример. Необходимо написать программу для получения ответа на вопрос: сколькими способами из 25 учеников класса можно выбрать четверых для участия в праздничном концерте? Очевидно, что понадобится формула из комбинаторики для определения числа сочетаний: $C_n^k = \frac{n!}{(n-k)!\cdot k!}$. Формула эта простая, однако при ее применении нужно *трижды* вычислить факториал числа. Запрограммируем вычисление ответа по формуле:

```
n = 25  # общее количество учеников класса
k = 4  # размер комбинации

# вычислим факториал n
n_fact = 1
for i in range(2, n + 1):
  n_fact *= i

# вычислим факториал n - k
nk_fact = 1
for i in range(2, n - k + 1):
  nk_fact *= i

# вычислим факториал k
k_fact = 1
for i in range(2, k + 1):
  k_fact *= i

# наконец, вычислим число сочетаний
C = n_fact // (nk_fact * k_fact)  # используем деление без остатка,
# чтобы результат был типа int (оба операнда целочисленные)
print(f"4 учеников из 25 можно выбрать {C} способами.")
```
```
4 учеников из 25 можно выбрать 12650 способами.
```

В программе нам пришлось *трижды* вычислять факториал. При этом был использован один и тот же алгоритм, отличались только входные данные. В итоге наша программа оказалась *минимум наполовину состоящей из повторяющегося кода* реализации алгоритма вычисления факториала. Было бы хорошо, если бы для вычисления факториала существовала готовая функция! Это бы позволило существенно упростить и сократить код программы[4].

Если бы функция вычисления факториала называлась `fact()`, вычисления выглядели бы так (код не сработает, потому что функция отсутствует):

[4] На самом деле такая функция есть в модуле math, но представим себе, что ее нет.

```
n_fact = fact(n)  # вычислим факториал n
nk_fact = fact(n - k)  # вычислим факториал n - k
k_fact = fact(k)  # вычислим факториал k
C = n_fact // (nk_fact * k_fact)  # наконец, вычислим число сочетаний
```

Или даже так (без использования переменных для хранения промежуточных результатов):

```
C = fact(n) // (fact(n - k) * fact(k))  # этот код тоже не сработает
```

Но готовой функции `fact()` нет. Что ж, напишем ее сами:

```
# эта часть кода - описание пользовательской функции fact()
def fact(value):
  result = 1
  for i in range(2, value + 1):
    result *= i
  return result


# а тут начинается основная программа
n = 25  # общее количество учеников класса
k = 4  # размер комбинации
C = fact(n) // (fact(n - k) * fact(k))  # вычислим число сочетаний
print(f"4 учеников из 25 можно выбрать {C} способами.")
```

```
4 учеников из 25 можно выбрать 12650 способами.
```

Как видим, алгоритм вычисления факториала нам пришлось запрограммировать только один раз. В результате получилась *пользовательская* (то есть написанная нами вручную) функция. Теперь для вычисления факториала достаточно лишь вызвать функцию с нужными значениями аргументов (в данном случае аргумент один – число, факториал которого нужно вычислить).

Разберемся с синтаксисом описания пользовательских функций. Описание функции в Python выглядит так:

```
def <имя функции>(<аргумент 1>, <аргумент 2>, ...):
  команды тела функции
  return <возвращаемое значение>
```

Для описания функции используются ключевые слова `def` и `return`. Фрагмент программы, описывающий алгоритм, реализуемый функцией, называют ***телом функции*** (по аналогии с телом цикла). При описании тела функции используется *блок кода*: начинается с двоеточия, выделяется отступами (как в условном операторе, операторах цикла и некоторых других конструкциях). Ключевое слово `return` завершает выполнение функции, возвращая результат ее работы в *точку вызова* (место в программе, в котором была вызвана функция).

Имена функций составляются по тем же правилам, что и имена переменных. Конечно, совпадающих имен объектов (включая функции) в программе быть не должно. Описание функции может располагаться в любом месте программы. Однако прежде чем использовать функцию, ее нужно описать и загрузить в память. Иначе говоря, описание функции в коде должно располагаться перед ее вызовом.

```
# рассмотрим еще раз функцию, вычисляющую факториал
def fact(value):  # имя функции - fact, аргумент - value
  # тут начинается тело функции, в котором вычисляется факториал числа,
  # содержащегося в переменной value
  result = 1
  for i in range(2, value + 1):
    result *= i
  # вычисленное значение факториала хранится в переменной result
  return result  # и является возвращаемым значением функции
```

При вычислении выражения, содержащего функции, они вызываются в порядке, соответствующем приоритетам операций, и после своего выполнения будто бы превращаются в свой результат (возвращаемое значение) (рис. 54).

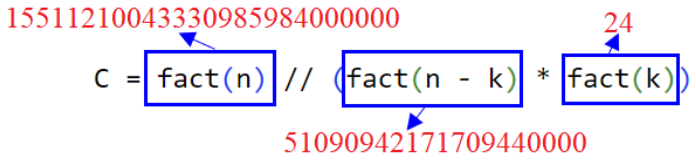


Рис. 54. Вызов функций при вычислении выражения

В нашем примере переменная `k` имеет значение 25. Это число 25 подается на вход функции `fact()`, которая возвращает его факториал (длинное число 15511...), и он участвует в дальнейшем вычислении выражения. Затем вычисляется второй факториал, на этот раз числа $n - k = 25 - 4 = 21$. И наконец, вычисляется третий факториал – числа $k = 4$.

Как можно заметить, на месте любого аргумента функции может быть как значение, заданное литералом или переменной, так и выражение. Если хотя бы один аргумент задан выражением, функция начнет выполняться только тогда, когда все выражения будут вычислены и превратятся в свои результирующие значения.

Что происходит при вызове функции? Начинает выполняться *код ее реализации*, то есть команды, описанные в блоке кода конструкции `def` (тело функции). При этом значения аргументов функции, заданные при ее вызове, передаются в переменные, перечисленные при описании функции в скобках после ее имени. После выполнения функции возвращаемый результат появляется в точке ее вызова как значение определенного типа (рис. 55).

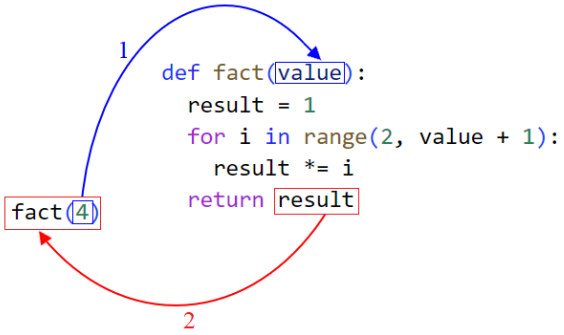


Рис. 55. Обмен данными функции с родительской программой: 1 – передача значений аргументов в функцию; 2 – передача результата в основную программу

Переменные, объявляемые *внутри тела функции*, а также переменные – входные параметры являются *локальными переменными функции* и существуют как объекты в памяти *только во время ее выполнения*. В нашем примере это переменные `value` (аргумент функции) и `result` (объявляется внутри тела функции). Из основной программы к локальным переменным функции обратиться *нельзя*. В этом нет никакого смысла: реализация функции (конструкция `def`) – это подпрограмма, которая во время выполнения основной программы хранится в оперативной памяти, никак себя не проявляет, как бы спит, а «просыпается» только тогда, когда ее вызывают, и начинает обрабатывать те данные, которые подали ей на вход в виде конкретных значений аргументов.

Чтобы использовать (вызвать) пользовательскую функцию, предварительно ее код нужно *выполнить*, то есть загрузить в память. Если были внесены изменения в код функции, его нужно *выполнить снова* (иначе в памяти останется старая версия кода функции). Последнее замечание справедливо для среды Google Colab, в которой ячейки с кодом можно выполнять в любом порядке.

Хорошим тоном является документирование функций с помощью многострочного комментария в начале тела, описывающего, что именно делает функция, какие данные принимает на вход и какой результат возвращает. Этот комментарий называется ***докстрингом***.

```
def foo():
  """Эта функция ничего не делает"""  # докстринг
```

Ключевое слово `return` в теле функции *может отсутствовать*. Это значит, что функция ничего не возвращает. Этот факт может вызвать недоумение: как же в основную программу передается результат работы функции? На самом деле входные параметры (аргументы) и возвращаемое значение – это *не единственные пути обмена данными* функции с основной программой, но в этом разберемся дальше. Если в теле функции отсутствует оператор `return`, она возвращает значение `None`. Убедимся в этом:

```
print(foo())
```
```
None
```

```
# а что возвращает функция print()? Тоже ничего
res = print("Результат работы print() - на экране, а не в памяти компьютера")
print(res)  # res содержит None
```
```
Результат работы print() - на экране, а не в памяти компьютера
None
```

Немного попрактикуемся в написании пользовательских функций.

```
def sum2(a, b):
  """Функция, складывающая 2 числа"""
  return a + b  # возвращает сумму значений a и b


sum2(5, -9)  # вызов функции с аргументами 5 и -9
```
```
-4
```

```
def quadratic_equation(a, b, c):
  """Функция, решающая квадратное уравнение"""
  result = []  # список корней
  D = b ** 2 - 4 * a * c  # дискриминант
  if D > 0:  # два корня - добавляем в список
    result.append((-b - D ** 0.5) / (2 * a))
    result.append((-b + D ** 0.5) / (2 * a))
  elif D == 0:  # один корень
    result.append(-b / (2 * a))
  # а если D < 0, корней нет, и список останется пустым
  return result


print("Корни уравнения 2x^2+5x+3=0:", quadratic_equation(2, 5, 3))
print("Корни уравнения x^2+2x+1=0:", quadratic_equation(1, 2, 1))
print("Корни уравнения -5x^2+2x-3=0:", quadratic_equation(-5, 2, -3))
```
```
Корни уравнения 2x^2+5x+3=0: [-1.5, -1.0]
Корни уравнения x^2+2x+1=0: [-1.0]
Корни уравнения -5x^2+2x-3=0: []
```

9.2. Значения аргументов по умолчанию

Все аргументы функции, перечисленные при ее описании, необходимо задать при ее вызове. Если количество значений, переданных в функцию при вызове, не совпадет с количеством ожидаемых аргументов, возникнет ошибка TypeError:

```
sum2(1, 9, 0)  # ожидается 2 значения, передано 3
```
```
...
TypeError: sum2() takes 2 positional arguments but 3 were given
```

```
sum2(0)  # ожидается 2 значения, передано 1
```
```
...
TypeError: sum2() missing 1 required positional argument: 'b'
```

Однако аргументам можно задать *значения по умолчанию*. ***Значение аргумента по умолчанию*** – это значение, которое используется в случае, если при вызове функции значение данного аргумента не передано.

Обратите внимание: знак = в коде ниже – это не оператор присваивания! В Python одни и те же символы используются для разных действий в зависимости от контекста.

```
def sum3(a, b, c=5):
  """Функция, складывающая 3 числа"""
  # аргумент c имеет значение по умолчанию: 5
  return a + b + c


sum3(10, 10, 10)  # указали все 3 аргумента, получили их сумму: 30
sum3(10, 10)  # указали только 2 аргумента, для c было использовано
# значение по умолчанию 5, результат: 25
```

9.3. Позиционные и именованные аргументы

Аргументы функции, передаваемые при вызове в том порядке, в каком они перечислены в реализации этой функции, называются ***позиционными*** (рис. 56).

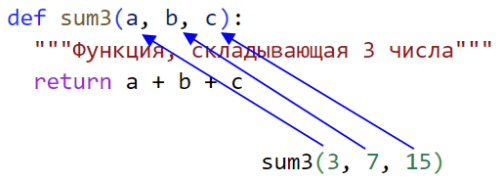


Рис. 56. Вызов функции с позиционными аргументами

В примере на рисунке при выполнении функции в ее локальную переменную `a` попадет целочисленное значение 3, в `b` попадет 7, в `c` попадет 15. Существует способ явно указать аргументы, в которые должно быть передано то или иное значение. Такие аргументы называются ***именованными***.

```
sum3(a=3, b=7, c=15)  # теперь не запутаемся, в какую переменную что попадет
```
```
25
```

Можно вспомнить функции `print()` и `sorted()`, при вызове которых мы использовали именованные аргументы:

```
print("I", "love", "Python", sep=";", end="!\n")
# sep и end - это именованные аргументы
```
```
I;love;Python!
```

```
sorted(["I", "love", "Python"], key=len, reverse=True)
# key и reverse - именованные аргументы
```
```
['Python', 'love', 'I']
```

Именованные аргументы позволяют использовать *произвольный* порядок передачи входных значений в функцию. Если указано имя аргумента, его позиция не важна.

```
sum3(c=7, b=15, a=3)
25
```

Позиционные и именованные аргументы можно *комбинировать*, то есть использовать вместе (рис. 57). Но в этом случае позиционные аргументы должны следовать *строго перед именованными*, а именованные должны задавать *только те значения, которые еще не заданы позиционными*.

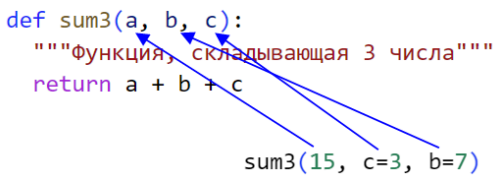


Рис. 57. Вызов функции с именованными аргументами

```
sum3(15, c=3, b=7)
25
```

А вот код из примера ниже не сработает, потому что переменная `a` уже задана позиционным аргументом, а мы вновь пытаемся ее задать именованным аргументом.

```
sum3(15, a=3, b=7)
TypeError: sum3() got multiple values for argument 'a'
```

```
# еще пример правильного использования аргументов обоих типов
print("I", "love", "Python", sep=";", end="!")
I;love;Python!
```

```
# неправильное использование аргументов (позиционные идут за именованными)
print("I", end="!\n\n", "love", "Python", sep=";")
...
SyntaxError: positional argument follows keyword argument
```

9.4. Нефиксированное количество аргументов функции

Мы уже знаем, что при вызове функции можно распаковывать коллекции (итерируемые объекты) в серию ее *позиционных* аргументов с помощью одинарной звездочки `*` (см. параграф 5.8). Вспомним, как это работает:

```
my_list = [1, 2, 3]

print(*my_list)  # это работает аналогично команде ниже
print(my_list[0], my_list[1], my_list[2])

# сравните с передачей списка в функцию как единственного объекта
print(my_list)
```

```
1 2 3
1 2 3
[1, 2, 3]
```

```
# передадим все элементы списка в функцию, она как раз ожидает 3 аргумента
sum3(*my_list)
```

```
6
```

```
# еще примеры
print(*range(15), sep=", ")  # распаковка диапазона
print(*"Ты заходи, если что!", sep="~")  # распаковка строки
```

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
Т~ы~ ~з~а~х~о~д~и~,~ ~е~с~л~и~ ~ч~т~о~!
```

Коллекцию типа *словарь* (`dict`) можно раскрывать в серию *именованных* аргументов с помощью двойной звездочки `**`. (Этот пример пока можно пропустить, вернуться к нему после изучения словарей, см. параграф 10.2.)

```
def func(a, b, c, d):
  """Функция выводит на экран значения своих аргументов"""
  print(f"Аргумент a = {a}")
  print(f"Аргумент b = {b}")
  print(f"Аргумент c = {c}")
  print(f"Аргумент d = {d}")


args_dict = {"a": "AAA", "b": "BBB", "c": "CCC", "d": "DDD"}  # словарь
func(**args_dict)  # в аргументы попали значения с соответствующими ключами
```

```
Аргумент a = AAA
Аргумент b = BBB
Аргумент c = CCC
Аргумент d = DDD
```

Сфокусируемся на одной интересной особенности функции `print()`: она может принимать *разное количество* позиционных аргументов.

```
print(1, 2, 3)  # 3 аргумента
print(*range(10))  # 10 аргументов
print()  # ни одного аргумента вообще (выведется пустая строка)
```

```
1 2 3
0 1 2 3 4 5 6 7 8 9

```

Такой механизм (поддержка нефиксированного количества аргументов) можно задействовать и в пользовательских функциях. Здесь вновь помогут символы звездочки, но добавленные уже не при вызове функции, а при ее *описании*.

Произвольное количество *позиционных* аргументов передается в виде *кортежа*. Для этого используется одинарная звездочка `*`.

```
def sum_many(*args):
  """Функция складывает все значения, переданные в нее"""
  # локальная переменная args - это кортеж всех переданных значений
  result = 0
  for x in args:
    result += x
  return result


sum_many(4, 7, 9, 5.1, -7.777)  # сумма 5 значений: 17.323
sum_many()  # сумма 0 значений: 0
```

Произвольное количество *именованных* аргументов передается в виде *словаря*, в котором ключами являются названия аргументов, а значениями – собственно их значения. Для этого используется двойная звездочка `**`. (К данному примеру можно вернуться после изучения словарей, см. параграф 10.2.)

```
def func_kwargs(**kwargs):
  """Функция выводит на экран словарь
  переданных в нее именованных аргументов"""
  # локальная переменная kwargs - это словарь
  print(kwargs)


func_kwargs(arg1=1, arg2="Ыыы!", arg3=[7, 7, 7])  # передано 3 имен. аргумента
func_kwargs()  # передано 0 именованных аргументов, результат - пустой словарь
```

```
{'arg1': 1, 'arg2': 'Ыыы!', 'arg3': [7, 7, 7]}
{}
```

```
# *args и **kwargs можно скомбинировать (сначала следует *args)

def func_args_kwargs(*args, **kwargs):
  """Функция выводит на экран все переданные аргументы
  в виде кортежа и словаря"""
  print("Кортеж позиционных аргументов:", args)
  print("Словарь именованных аргументов:", kwargs)


# передано 3 позиционных аргумента и 1 именованный
func_args_kwargs(10, 20, -50, arg1="Ууууу!")
print()  # вставим пустую строку, чтобы разделить вывод
func_args_kwargs()  # не передано ни одного аргумента
```

```
Кортеж позиционных аргументов: (10, 20, -50)
Словарь именованных аргументов: {'arg1': 'Ууууу!'}

Кортеж позиционных аргументов: ()
Словарь именованных аргументов: {}
```

Локальные переменные `args` и `kwargs`, хранящие кортеж и словарь из аргументов, могут *называться как угодно*, это обычные переменные; важным элементом синтаксиса языка, включающим возможность передавать произвольное

количество аргументов в функцию, являются *звездочки*, а не имена переменных. Однако именно такие названия (`args`, `kwargs`) являются общепринятыми в Python-сообществе.

9.5. Локальные и глобальные переменные

Как уже было отмечено выше, переменные, объявляемые внутри функции (включая ее аргументы), называются ***локальными***, доступны только изнутри функции и существуют в памяти лишь во время ее выполнения. Переменные основной программы называются ***глобальными***. Во избежание путаницы не следует использовать одинаковые имена для локальных и глобальных переменных.

```
# переменные x (объявляется в теле функции) и y (аргумент функции) -
# это не те же x, y, которые используются в основной программе

def foo(x):
  y = 100  # x, y - локальные переменные функции
  print("Локальная x = ", x, ", локальная y = ", y, sep="")


x, y = 1, -1  # переменные основной программы (глобальные)
foo(0)  # выполнение функции, 0 передается в аргумент x
print("Глобальная x = ", x, ", глобальная y = ", y, sep="")
```

```
Локальная x = 0, локальная y = 100
Глобальная x = 1, глобальная y = -1
```

Из основной программы нельзя обращаться к локальным переменным функции. Однако из функции обращаться к глобальным переменным *можно*. Это логично, потому что в момент выполнения функции все объекты, объявленные до ее вызова в основной программе, никуда не исчезают, продолжают храниться в памяти.

Однако есть *нюанс*: глобальные переменные из функции доступны только для *чтения* (не для записи).

```
def foo2():
  # что это за x? В функции ее не объявляли
  print("Глобальная x =", x)  # это x из основной программы


x = 1  # глобальная x
foo2()  # выполнение функции
print("Глобальная x =", x)

# убедимся, что в функции используется именно глобальная x
x = 999  # изменим ее значение
foo2()
print("Глобальная x =", x)
```

```
Глобальная x = 1
Глобальная x = 1
Глобальная x = 999
Глобальная x = 999
```

Можно попробовать перезаписать значение `x` внутри функции, но тогда у функции возникнет своя собственная локальная переменная, не имеющая связи с глобальной переменной в основной программе (см. предыдущий пример, где внутри функции записывали в `y` значение 100). Однако можно сделать так, чтобы в теле функции возможно было полноценно работать с глобальными переменными, в том числе и перезаписывать их значения. Для этого используется ключевое слово `global`:

```
global <глобальная переменная 1>, <глобальная переменная 2>, ...
```

```
def foo3():
  # перечислим глобальные переменные, которые хотим перезаписывать
  global x, y  # теперь x и y - это переменные из основной программы
  x, y = x * 100, y * 100


x, y = 1, -1  # переменные основной программы (глобальные)
foo3()  # выполним функцию, которая умножает x и y на 100
# убедимся, что значения переменных изменились
print("x = ", x, ", y = ", y, sep="")
```

```
x = 100, y = -100
```

Таким образом, если в функции необходимо переприсваивать глобальные переменные, следует использовать оператор `global` (причем *до* переприсваивания), иначе будут создаваться локальные переменные с таким же именем. Что же касается сложных изменяемых объектов (таких как списки и другие изменяемые коллекции), изнутри функции их можно не только считывать, но и *модифицировать*. Команда `global` для этого не нужна.

```
def add_elem(new_elem):
  """Функция добавляет в список из основной программы новый элемент"""
  my_list.append(new_elem)  # my_list - глобальная переменная


my_list = [2, 5, 7, 11, 13]  # список простых чисел
add_elem(17)  # вызовем функцию для добавления в список нового числа
my_list  # убедимся, что список изменился
```

```
[2, 5, 7, 11, 13, 17]
```

```
# однако чтобы перезаписать переменную my_list, без global не обойтись

def add_elem(new_elem):
  # global my_list
  my_list = new_elem  # my_list - локальная переменная!


my_list = [2, 5, 7, 11, 13]
add_elem(17)  # вызовем функцию
my_list  # убедимся, что список НЕ изменился
```

```
[2, 5, 7, 11, 13]
```

Можно раскомментировать строку с `global` и проверить правильность своего понимания того, как изменилось поведение программы.

Важное замечание: не следует злоупотреблять обращением изнутри функции к глобальным переменным. Впоследствии может понадобиться использовать эту функцию в другой программе (вдруг она окажется очень полезной и удачной), однако в этой новой программе глобальных переменных из старой, с которыми работает функция, может не оказаться. Функция *не должна зависеть от данных родительской программы*, она должна работать только со своими локальными переменными. Если нужно передать данные в функцию, надо их передавать в качестве аргументов. *Это общепринятое правило в программировании.*

9.6. Передача в функцию изменяемых и неизменяемых объектов

Есть еще один важный момент, связанный с передачей в функцию *изменяемых* объектов (например, списков). Можно вспомнить, как с изменяемыми объектами работает *оператор присваивания*: объект не копируется, на него лишь назначается еще одна ссылка. При передаче изменяемого объекта как аргумента в функцию происходит практически *то же самое*: в функции не появляется копия объекта из основной программы, на него просто назначается новая ссылка (локальная переменная-аргумент). Рассмотрим пример:

```
def myfunc(a, b):
  """Функция увеличивает на 1 первый аргумент
  и первый элемент 2-го аргумента-списка"""
  a += 1  # переменная a переприсваивается, поэтому она локальная
  b[0] += 1  # а вот список b является списком y из основной программы


x = 10  # глобальная переменная целочисленного типа
y = [1, 2, 3]  # глобальная переменная-список
myfunc(x, y)  # вызов функции
print(x, y) # убедимся, что значение x не изменилось, а y - изменилось
```

```
10 [2, 2, 3]
```

Вот как организовано хранение данных в коде выше: переменная-аргумент `a` содержит копию целого числа из переменной `x` основной программы, а вот переменная-аргумент `b` является всего лишь ссылкой на объект-список `y` из основной программы (рис. 58).

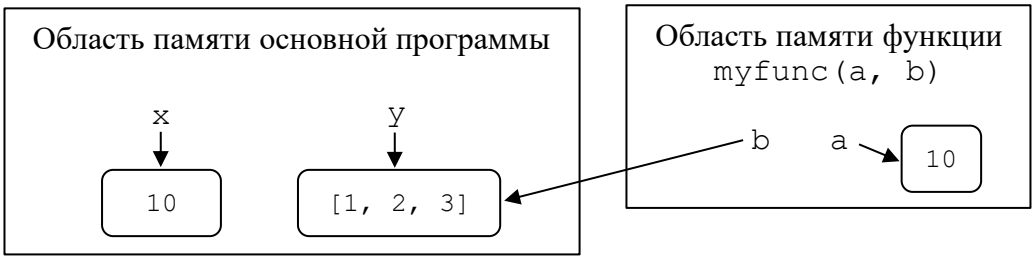


Рис. 58. Передача в функцию изменяемого объекта по ссылке

Строго говоря, списки и другие изменяемые коллекции можно в функцию и не передавать с помощью аргументов, они все равно будут доступны для чтения и модификации из тела функции. Однако такой подход не удовлетворяет описанному выше правилу (передача данных в функцию по возможности должна осуществляться через аргументы).

9.7. Анонимные (лямбда) функции

Вспомним сортировку коллекции по конструируемому ключу. В примере ниже производится сортировка слов в предложении по последней букве. При этом стандартной функции, которая бы извлекала из слова последнюю букву, нет. Придется написать ее самим:

```
def get_last_letter(w):
  """Функция получает на вход строку и возвращает ее последний символ"""
  return w[-1]


words = input("Введите предложение из нескольких слов: ").split()
sorted(words, key=get_last_letter)  # сортируем список слов
```

```
Введите предложение из нескольких слов: есть много способов медитировать
['способов', 'много', 'есть', 'медитировать']
```

Совсем не факт, что функция `get_last_letter()`, извлекающая последний символ из строки, понадобится в программе где-то еще. Однако при сложной сортировке нужна именно *функция*. Другими примерами функций, где значением аргумента может выступать другая функция, являются `map()`, `filter()`, `max()` и `min()`. А ведь главное преимущество функций – возможность избежать дублирования кода – в таких случаях *не используется*. Более того, из-за конструкции `def`, описывающей пользовательскую функцию, в программе появляется несколько лишних строк (не забудем, что стандарт PEP8 рекомендует отделять объявления функций от основного кода двумя пустыми строками).

Существует особая разновидность функций, предназначенная для использования в указанной ситуации, – ***анонимные*** функции (или ***лямбда***-функции). Описываются они с помощью ключевого слова `lambda` и вызываются в том же месте, где описаны:

```
lambda <аргументы функции (может быть несколько)>: <возвращаемое значение>
```

Заменим обычную функцию с именем `get_last_letter()` на лямбда-функцию, которая делает то же самое (код ниже, рис. 59).

```
# можно сказать, что оператор lambda описывает выражение,
# результатом вычисления которого является функция
lambda w: w[-1]
```

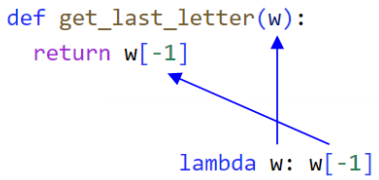


Рис. 59. Сопоставление обычной и анонимной функций

Конечно, обратиться к лямбда-функции из произвольного места программы не выйдет. Она же *анонимная (безымянная)*, как к ней обращаться? Но она и не для этого нужна. Заменим при сортировке обычную функцию на ее лямбда-аналог:

```
# как видим, результат аналогичен, а запись значительно короче
sorted(words, key=lambda w: w[-1])
```

```
['способов', 'много', 'есть', 'медитировать']
```

Пример фильтрации элементов списка с помощью лямбда-функции:

```
# отберем из последовательности только отрицательные числа
numbers = [7, 4, 0, -4, -18, 9, -1, 2]
list(filter(lambda x: x < 0, numbers))
```

```
[-4, -18, -1]
```

Если очень хочется дать имя безымянной функции, это возможно (как и многое в Python – языке, очень лояльном к программисту). Однако записывать лямбда-функции в переменные не принято.

```
is_negative = lambda x: x < 0  # записали объект-функцию в переменную

is_negative(5)  # проверка на отрицательность числа 5: False
is_negative(0)  # проверка на отрицательность числа 0: False
is_negative(-5)  # проверка на отрицательность числа -5: True
```

9.8. Задания для самостоятельной работы

Решите следующие задачи:

1. Напишите функцию для округления вещественных чисел до заданного разряда – собственную реализацию стандартной функции `round()`. Саму функцию `round()` использовать нельзя.

2. Напишите функцию ввода с клавиатуры целого или вещественного числа с проверкой правильности ввода. Если введено целое число, результатом функции должно быть значение типа `int`. Если введено число с дробной частью, результат должен быть типа `float`. Если введено некорректное значение (не число), ввод должен повторяться до тех пор, пока не будет введено число.

3. Напишите лямбда-функцию, определяющую количество слогов в слове. Решение должно подходить как для русского, так и для английского языка.

10. Неиндексированные коллекции и файлы

В данной главе рассматриваются новые типы данных – неиндексированные коллекции. Излагаются основы работы с файлами. Затрагивается обработка структурированных текстовых данных формата JSON.

10.1. Множества

Ранее мы рассматривали такие структуры данных в Python, как *коллекции* (итерируемые объекты), узнали, что коллекции могут быть:

✓ *индексированными* (позволяют обращаться к конкретному элементу по его порядковому номеру) и *неиндексированными*;

✓ *изменяемыми* (их можно модифицировать, например добавлять и удалять элементы) и *неизменяемыми*.

Мы изучили такие структуры данных, как *строка* (`str`), *кортеж* (`tuple`) и *список* (`list`). Поговорим о еще двух типах данных: *множествах* и *словарях*.

Множество (`set`) – ***изменяемый неиндексированный*** тип коллекций. Множество можно охарактеризовать как *неупорядоченный* набор элементов, каждый из которых *уникален* (не повторяется). В математике есть структура с соответствующим названием, и можно сказать, что тип данных `set` наследует свойства этого математического объекта. Создадим объект-множество и добавим в него несколько значений (код ниже, рис. 60).

```
my_set = {3, 5, 8, 2, 0, 1}  # зададим множество перечислением его элементов
my_set
```

```
{0, 1, 2, 3, 5, 8}
```

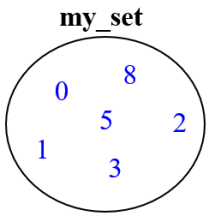


Рис. 60. Представление множества в памяти

Можно заметить, что при выводе на экран содержимого множества его элементы расположились в порядке возрастания. Этот наблюдаемый факт не должен вводить в заблуждение. Множество *не является индексируемой коллекцией*, его элементы *не имеют порядковых номеров*, нельзя напрямую обратиться к конкретному, например, десятому элементу множества. Следовательно, множества *не поддерживают механизм срезов*.

```
empty_set = set()  # создание пустого множества
# обратите внимание, запись {} к аналогичному результату не приведет!
```

Множество является *изменяемой* коллекцией, однако оно может состоять *только из неизменяемых элементов*. Причем изменяемых объектов не должно быть *ни на одном уровне вложенности* (например, кортеж кортежей списков не может быть элементом множества).

```
my_set = {17, "привет", True, (3.33, 6.67, 10.0)}  # так можно
# my_set = {17, "привет", True, [3.33, 6.67, 10.0]}  # а так - нет
```

Множество – коллекция, с ним работают соответствующие инструменты.

```
# функция len()
print("Количество элементов в множестве:", len(my_set))

# перебор элементов в цикле (порядок элементов не гарантирован!)
print("\nЭлементы множества:")
for elem in my_set:
  print(elem)

# распаковка элементов при вызове функции
print("\nЭлементы множества:", *my_set)

# проверка вхождения элемента в множество - операторы in / not in
print("\nЭлемент 17 входит в множество:", 17 in my_set)
print("Элемент 'пока' не входит в множество:", "пока" not in my_set)
```

```
Количество элементов в множестве: 4

Элементы множества:
17
привет
(3.33, 6.67, 10.0)
True

Элементы множества: 17 привет (3.33, 6.67, 10.0) True

Элемент 17 входит в множество: True
Элемент 'пока' не входит в множество: True
```

Добавление и удаление элементов

Метод `.add()` добавляет элемент в множество.

```
my_set.add(999)  # добавим 999
my_set.add(17)  # добавим 17, но 17 в множестве уже есть, ничего не произойдет
my_set
```

```
{(3.33, 6.67, 10.0), 17, 999, True, 'привет'}
```

Методы `.remove()` и `.discard()` удаляют элемент из множества. В случае отсутствия в множестве заданного значения `.remove()` приведет к ошибке, а `.discard()` нет.

```
my_set.remove(17)  # удалим элемент 17
my_set.discard(1000)  # попытаемся удалить элемент 1000 (которого нет)
my_set
```
```
{(3.33, 6.67, 10.0), 999, True, 'привет'}
```

Сравнение множеств

Рассмотрим особенности сравнения множеств между собой. Два множества считаются *равными*, если они состоят из одних и тех же элементов (рис. 61).

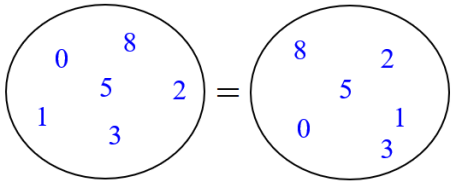


Рис. 61. Равные множества

```
{1, 0, 5, 3, 8, 2} == {8, 0, 5, 2, 1, 3}  # True
{1, 0, 5, 3, 8, 2} != {8, 0, 5, 2, 1, 3}  # False
```

Множество *A меньше* множества *B*, если *B* включает в себя все элементы *A*, но при этом содержит и другие объекты (рис. 62).

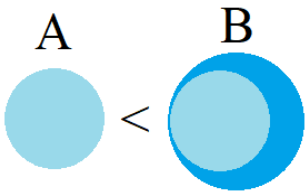


Рис. 62. Множество A является подмножеством B

```
# B полностью включает в себя A, но не исчерпывается им
A = {3, 2.14, "Ы"}
B = {3, 2.14, "Ы", "python", -18}

A < B  # A меньше B - да: True
A <= B  # A меньше либо равно B - да: True
A > B  # A больше B - нет: False
A >= B  # A больше либо равно B - нет: False
```

Также у множеств есть методы для сравнения:

`A.issubset(B)` – проверка, что `A` является *подмножеством* `B`; аналог операции `A <= B`.

`A.issuperset(B)` – проверка, что `A` является *надмножеством* `B`; аналог операции `A >= B`.

`A.isdisjoint(B)` – проверка, что `A` и `B` не имеют общих элементов.

```
print(f"{A} является ПОДмножеством {B}: {A.issubset(B)}")
print(f"{B} является ПОДмножеством {A}: {B.issubset(A)}\n")

print(f"{A} является НАДмножеством {B}: {A.issuperset(B)}")
print(f"{B} является НАДмножеством {A}: {B.issuperset(A)}\n")

print(f"{A} и {B} не имеют общих элементов: {A.isdisjoint(B)}")
# внешние скобки относятся к f-строке, внутренние - к описанию множества
print(f"{{1, 2, 3}} и {{7, 8,9}} не имеют общих элементов: {X.isdisjoint(Y)}")
```

```
{2.14, 3, 'Ы'} является ПОДмножеством {2.14, 3, 'python', 'Ы', -18}: True
{2.14, 3, 'python', 'Ы', -18} является ПОДмножеством {2.14, 3, 'Ы'}: False

{2.14, 3, 'Ы'} является НАДмножеством {2.14, 3, 'python', 'Ы', -18}: False
{2.14, 3, 'python', 'Ы', -18} является НАДмножеством {2.14, 3, 'Ы'}: True

{2.14, 3, 'Ы'} и {2.14, 3, 'python', 'Ы', -18} не имеют общих элементов: False
{1, 2, 3} и {8, 9, 7} не имеют общих элементов: True
```

Может возникнуть вопрос: если множества `A` и `B` состоят из разных элементов, то *какое из них больше*? И ответ – *никакое*. Множество – это такой тип данных, для двух значений которого не всегда можно однозначно сказать, какое из них больше. Это возможно только тогда, когда *одно из множеств является подмножеством другого*.

```
a = {1, 2, 3}
b = {4, 5, 6}

a < b  # нет: False
a > b  # нет: False
a == b  # нет: False
# единственное, что можно утверждать: множества a и b не равны
a != b  # да: True
```

Следовательно, напрашивается вывод: коллекции из элементов-множеств *не поддаются сортировке*, также в них *невозможно найти максимум и минимум*. Функцию сортировки применить можно, это не приведет к ошибке, однако порядок элементов не изменится, то есть сработает свойство устойчивости сортировки (если про два объекта нельзя однозначно сказать, какой из них больше, а какой меньше, то их порядок не должен меняться). Взглянем на пример:

```
# сгенерируем список множеств, которые не сравниваются между собой
list_of_sets = [set(x) for x in "кто проживает на дне океана?!".split()]
list_of_sets  # список до сортировки
```

```
[{'к', 'о', 'т'},
 {'а', 'в', 'е', 'ж', 'и', 'о', 'п', 'р', 'т'},
 {'а', 'н'},
 {'д', 'е', 'н'},
 {'!', '?', 'а', 'е', 'к', 'н', 'о'}]
```

```
sorted(list_of_sets)  # список после сортировки - точно такой же
```
```
[{'к', 'о', 'т'},
 {'а', 'в', 'е', 'ж', 'и', 'о', 'п', 'р', 'т'},
 {'а', 'н'},
 {'д', 'е', 'н'},
 {'!', '?', 'а', 'е', 'к', 'н', 'о'}]
```

В случае поиска максимума (минимума) в коллекции «несравнимых» множеств результатом будет *первый элемент коллекции*:

```
print("Максимум:", max(list_of_sets))
print("Минимум:", min(list_of_sets))
```
```
Максимум: {'т', 'к', 'о'}
Минимум: {'т', 'к', 'о'}
```

Операции алгебры множеств

Как и в математике, в Python над множествами определено несколько специфических операций. Пусть имеются множества `A` и `B`:

```
A = {7, 3, 0, 1, 5, 9, 14}
B = {6, 0, 15, 9, 2}
```

Результатом **объединения** множеств `A` и `B` является множество, состоящее из всех элементов обоих множеств (рис. 63). Данная операция является *симметричной* (не имеет значения порядок операндов).

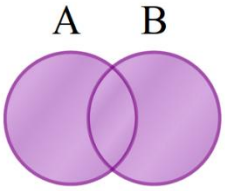


Рис. 63. Объединение множеств

```
A | B  # другой способ - метод union(): A.union(B)
```
```
{0, 1, 2, 3, 5, 6, 7, 9, 14, 15}
```

Результатом **пересечения** множеств `A` и `B` является множество, состоящее из общих элементов обоих множеств (рис. 64). Операция симметрична.

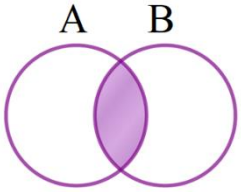


Рис. 64. Пересечение множеств

```
A & B  # другой способ - метод intersection(): A.intersection(B)
```
```
{0, 9}
```

Результатом **вычитания** множества `B` из множества `A` является множество, состоящее только из тех элементов `A`, которых нет в `B` (рис. 65). Операция *не симметрична* (порядок операндов важен).

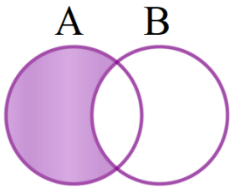


Рис. 65. Вычитание множеств

```
# другой способ - метод difference()
print("A - B =", A - B)  # A.difference(B)
print("B - A =", B - A)  # B.difference(A)
```
```
A - B = {1, 3, 5, 7, 14}
B - A = {2, 6, 15}
```

Результатом **симметрической разности** множеств `A` и `B` является множество, состоящее из уникальных элементов обоих множеств (рис. 66). Операция симметрична.

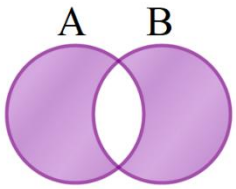


Рис. 66. Симметрическая разность множеств

```
# другой способ - метод symmetric_difference(): A.symmetric_difference(B)
print(A ^ B)

# а еще это то же самое, что разность объединения и пересечения
(A | B) - (A & B)
```
```
{1, 2, 3, 5, 6, 7, 14, 15}
{1, 2, 3, 5, 6, 7, 14, 15}
```

Генерация множества (set comprehension)

По аналогии с генерацией списка существует и *генерация множества*. Синтаксически вторая конструкция отличается от первой фигурными скобками вместо квадратных.

```
# множество, включающее только заглавные буквы исходной строки,
# приведенные к нижнему регистру
{x.lower() for x in "сИнхрОфазОТроН" if x.isupper()}
```
```
{'и', 'н', 'о', 'т', 'ф'}
```

Неизменяемое множество

Можно сказать, что кортеж является неизменяемым аналогом списка. У множества тоже есть неизменяемый аналог: `frozenset`.

```
# сконструировать такое множество можно путем
# преобразования коллекции к типу frozenset
my_frozenset = frozenset({1, 2, 3})
my_frozenset
```

```
frozenset({1, 2, 3})
```

С помощью `frozenset` можно создать *множество множеств*:

```
fA = frozenset([1, "aaa"])
fB = frozenset(["hdfkg", 3.14])
fC = frozenset(["привет"])

set_of_frozensets = {fA, fB, fC}
set_of_frozensets
```

```
{frozenset({1, 'aaa'}), frozenset({'привет'}), frozenset({3.14, 'hdfkg'})}
```

Применение множеств в программировании

Множества могут быть полезны, например для *удаления повторяющихся элементов* в коллекции. Для этого можно преобразовать коллекцию в множество, а затем вновь привести к исходному типу. Условия применения такого способа удаления дубликатов:

- ✓ коллекция состоит из неизменяемых элементов;
- ✓ порядок элементов не имеет значения.

```
# удалим повторяющиеся буквы
list(set(["a", "b", "a", "h", "s", "h", "c"]))
```

```
['s', 'a', 'c', 'h', 'b']
```

Также множества удобно использовать в качестве объектов, к которым применимы *операции алгебры множеств*. Например, можно из двух списков отобрать только те элементы, которые встречаются в обоих из них (выполнить пересечение списков). И вновь не надо забывать, что в процессе преобразования коллекции в множество теряется порядок следования элементов (если, конечно, в исходной коллекции он был).

Решим такую задачу. Петя любит яблоки, апельсины и виноград, а Виолетта – кокосы, ананасы, апельсины и бананы. Нужно сформировать перечень фруктов, которые любят и Петя, и Виолетта. Операция пересечения множеств позволит сделать это с легкостью.

```
Petya = "яблоки", "апельсины", "виноград"
Violetta = "кокосы", "ананасы", "апельсины", "бананы"
print("Оба любят:", set(Petya) & set(Violetta))
```

```
Оба любят: {'апельсины'}
```

10.2. Словари

Словарь (`dict`) – ***изменяемый неиндексированный*** тип коллекций. Словарь можно охарактеризовать как *неупорядоченный* набор пар «ключ – значение», причем ключи *уникальны*. Этот тип данных похож и на множество, и на список.

Если взять *множество* и мысленно к каждому его элементу «прицепить» дополнительный реквизит, получится словарь (рис. 67). Как и элементы множества, ключи словаря не могут повторяться и могут быть только неизменяемыми значениями.

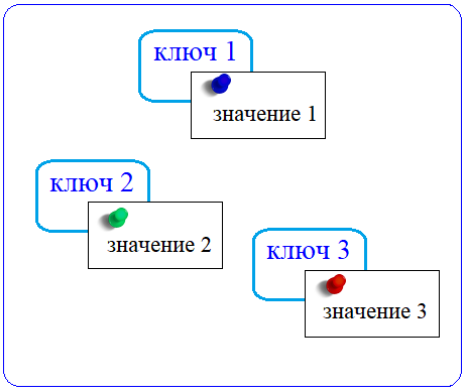


Рис. 67. Сходство словаря с множеством

Если взять *список* и заменить индексы (порядковые номера) его элементов на некоторые метки, тоже получится словарь (рис. 68). Как и в случае списка, к конкретным элементам словаря (значениям) обращаться можно, но уже не по порядковому номеру (ведь порядок отсутствует), а по *метке (ключу)*.

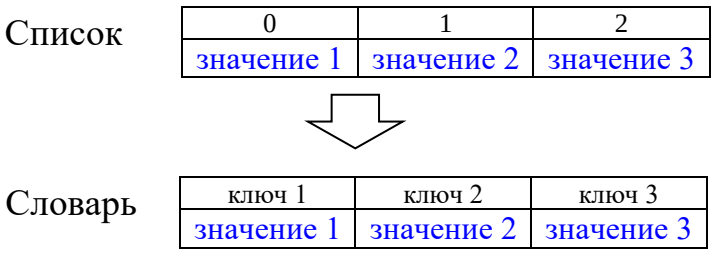


Рис. 68. Сходство словаря со списком

Создание словаря и обращение к его элементам

Рассмотрим способы создания словаря.

```
# создадим словарь путем перечисления элементов
my_dict = {
  "кошка": "cat",  # каждый элемент словаря - это пара значений
  "собака": "dog",  # они записываются через двоеточие
  "птица": "bird"
}

my_dict
```
```
{'кошка': 'cat', 'собака': 'dog', 'птица': 'bird'}
```

```
# создадим пустой словарь; другой способ - функция dict()
empty_dict = {}
# пустые фигурные скобки зарезервированы для словаря, а не для множества
empty_dict
```
```
{}
```

В словарь можно преобразовать коллекцию, элементы которой могут быть интерпретированы как пары «ключ – значение» (то есть являются коллекциями длины 2). Чтобы исполнять роль ключей, первые элементы должны быть *неизменяемыми*.

```
# словарь из списка кортежей из 2 элементов
dict([(0, "Иванов"),
      (1, "Петров"),
      (2, "Сидоров")])
```
```
{0: 'Иванов', 1: 'Петров', 2: 'Сидоров'}
```

```
# как тут не вспомнить функцию enumerate()?
dict(enumerate(["Иванов", "Петров", "Сидоров"]))
```
```
{0: 'Иванов', 1: 'Петров', 2: 'Сидоров'}
```

Для обращения к отдельному элементу словаря используется тот же синтаксис, что для обращения к элементу списка, – квадратные скобки. Только идентификатором элемента является не его порядковый номер, а *ключ*.

```
# обращение к элементу словаря по ключу
my_dict["кошка"]  # обращаемся по ключу - получаем значение
# если такого ключа нет, возникнет ошибка
```
```
'cat'
```

Обход элементов словаря

Как и множество, будучи неиндексированным типом, словарь *не поддерживает срезы*, однако он является коллекцией и обладает поддержкой инструментов для обработки коллекций.

```
# например, поддерживается функция len()
print("Количество элементов в словаре:", len(my_dict))
```
```
Количество элементов в словаре: 3
```

Важная деталь: при работе со словарем как с коллекцией, то есть при обходе его элементов, получаемыми значениями являются не пары «ключ – значение», а *ключи*.

```
# перебор элементов в цикле (порядок элементов не гарантирован!)
print("Ключи словаря:")
for key in my_dict:
  print(key)

# распаковка элементов при вызове функции
print("\nКлючи словаря:", *my_dict)

# проверка вхождения элемента с определенным ключом в словарь
print("\nЭлемент 'птица' входит в словарь:", "птица" in my_dict)
print("Элемент 'dog' не входит в словарь:", "dog" not in my_dict)
```

```
Ключи словаря:
кошка
собака
птица

Ключи словаря: кошка собака птица

Элемент 'птица' входит в словарь: True
Элемент 'dog' не входит в словарь: True
```

```
print(sorted(my_dict))  # сортировка ключей словаря
```

```
['кошка', 'птица', 'собака']
```

```
# сортировка ключей словаря по соответствующим этим ключам значениям
# x - это ключ, обращаемся по нему к значению
sorted(my_dict, key=lambda x: my_dict[x])
```

```
['птица', 'кошка', 'собака']
```

```
# "наибольший" ключ в словаре
# в данном случае ключи - строки, поэтому "наибольший" по алфавиту
max(my_dict)
```

```
собака
```

```
# "наименьший" ключ
# (в данном случае ключи - целые числа, сравниваются как обычно)
min({6: "котик", 0: "свинка", 14: "гадюка"})
```

```
0
```

Добавление и удаление элементов словаря

Для добавления в словарь элемента с заданным ключом используется *оператор присваивания*.

```
# если элемент с таким ключом уже есть, он перезапишется
my_dict["енот"] = "raccoon"
my_dict
```

```
{'кошка': 'cat', 'собака': 'dog', 'птица': 'bird', 'енот': 'raccoon'}
```

Метод `.pop()` удаляет элемент по ключу и возвращает его значение.

```
print("Удаляемое значение:", my_dict.pop("енот"))
my_dict
```

```
Удаляемое значение: raccoon
{'кошка': 'cat', 'собака': 'dog', 'птица': 'bird'}
```

Методы словаря

Метод `.get()` позволяет получить значение по ключу. Если заданного ключа в словаре нет, будет возвращено указанное вторым аргументом значение (по умолчанию `None`).

```
print("Элемент с ключом 'кошка':", my_dict.get("кошка"))  # такой элемент есть
# такого элемента нет, укажем возвращаемое значение явно
print("Элемент с ключом 'бегемот':", my_dict.get("бегемот", "hippo"))
# такого элемента нет, возвращаемое значение оставим по умолчанию
print("Элемент с ключом 'питон':", my_dict.get("питон"))
```

```
Элемент с ключом 'кошка': cat
Элемент с ключом 'бегемот': hippo
Элемент с ключом 'питон': None
```

Метод `.items()` формирует из словаря коллекцию кортежей вида `(<ключ>, <значение>)`.

```
print(my_dict.items(), end="\n")  # коллекция элементов в виде кортежей

# перебор элементов словаря обычным способом
print("\nЭлементы словаря - классический способ перебора:")
for key in my_dict:  # перебираются ключи
  print(key, "-", my_dict[key])  # выводим ключ и его значение

# перебор элементов словаря с помощью items() - более "в Python-стиле"
print("\nЭлементы словаря - более pythonic-способ перебора:")
# перебираются кортежи (ключ, значение)
for key, value in my_dict.items():
  print(key, "-", value)  # выводим ключ и его значение
```

```
dict_items([('кошка', 'cat'), ('собака', 'dog'), ('птица', 'bird')])

Элементы словаря - классический способ перебора:
кошка - cat
собака - dog
птица - bird

Элементы словаря - более pythonic-способ перебора:
кошка - cat
собака - dog
птица - bird
```

Методы `.keys()`, `.values()` позволяют получить коллекции соответственно ключей и значений по отдельности. Как любые итерируемые объекты, их можно преобразовать, например, в обычный список для дальнейшей обработки:

```
print("Ключи словаря:", my_dict.keys())  # выведем коллекцию ключей как есть
# преобразуем коллекцию значений в список
print("Значения словаря:", list(my_dict.values()))
```
```
Ключи словаря: dict_keys(['кошка', 'собака', 'птица'])
Значения словаря: ['cat', 'dog', 'bird']
```

Метод `.update()` дополняет словарь элементами другого словаря. Если в добавляемом словаре присутствуют элементы с ключами, имеющимися в исходном словаре, эти элементы перезапишутся новыми:

```
my_dict.update({"мышь": "mouse", "кошка": "pussycat"})
my_dict  # "мышь" добавилась, а значение "кошки" обновилось
```
```
{'кошка': 'pussycat', 'собака': 'dog', 'птица': 'bird', 'мышь': 'mouse'}
```

Сравнение словарей

Словари можно проверять на *равенство* и *неравенство* между собой. Равными являются словари, состоящие из одинаковых пар «ключ – значение»:

```
{"кот": "cat", "мышь": "mouse"} == {"мышь": "mouse", "кот": "cat"}  # True
{"кот": "cat", "мышь": "mouse"} != {"mouse": "мышь", "cat": "кот"}  # True
```

Операции сравнения словарей >, <, >=, <= *не определены*. Это логично, потому что если для множеств еще можно при определенных условиях установить, какое из них больше, то как сравнивать между собой такие сложные структуры данных, как словари, решительно непонятно. Следовательно, *сортировка* коллекции словарей тоже невозможна.

Генерация словаря (dict comprehension)

Существует и генерация словаря. Синтаксически эта конструкция отличается от генерации множества наличием *двоеточия*, разделяющего ключ и значение.

```
# словарь, содержащий частоты букв в слове
word = "синхрофазотрон"
# ключ - символ, значение - сколько раз он встречается
{x: word.count(x) for x in word}
```
```
{'с': 1,
 'и': 1,
 'н': 2,
 'х': 1,
 'р': 2,
 'о': 3,
 'ф': 1,
 'а': 1,
 'з': 1,
 'т': 1}
```

Применение словарей в программировании

Словари используются по прямому назначению – как хранилища объектов, к которым можно обращаться по некоторым меткам, в том случае, когда порядок объектов значения не имеет. Часто словари применяют для хранения данных в виде сложных иерархических структур различного количества степеней вложенности. Об этом поговорим в следующем параграфе, когда дойдем до формата данных JSON.

Рассмотрим следующий пример использования словарей. Имеется перечень имен всех студентов некоторой группы. Необходимо вывести *порядковые номера* в исходном перечне имен, упорядоченных по частоте (в начале – наиболее популярные имена, в конце – наименее).

```
# список имен
names = ["Иван", "Сергей", "Петр", "Сергей", "Елена", "Сергей",
         "Мария", "Елена", "Мария", "Сергей", "Иван", "Ульяна"]

# имена надо упорядочить по частоте, поэтому создадим словарь частот
positions = {name: names.count(name) for name in names}
positions
```

```
{'Иван': 2, 'Сергей': 4, 'Петр': 1, 'Елена': 2, 'Мария': 2, 'Ульяна': 1}
```

```
# теперь упорядочим ключи словаря по частотам от большей к меньшей
# аргументом лямбда-функции x является кортеж (имя, частота)
sorted_names = sorted(positions.items(), key=lambda x: x[1], reverse=True)
sorted_names  # результат - список кортежей
```

```
[('Сергей', 4),
 ('Иван', 2),
 ('Елена', 2),
 ('Мария', 2),
 ('Петр', 1),
 ('Ульяна', 1)]
```

```
# переберем этот список кортежей
# и для каждого кортежа выведем порядковые номера студентов с таким именем
for name, frequency in sorted_names:
  # порядковые номера получим с помощью генерации списка, нумерация с 1
  print(name, [i + 1 for i in range(len(names)) if names[i] == name])
```

```
Сергей [2, 4, 6, 10]
Иван [1, 11]
Елена [5, 8]
Мария [7, 9]
Петр [3]
Ульяна [12]
```

10.3. Работа с файлами

Программа может работать с данными не только в оперативной, но и во *внешней* памяти, то есть на локальном диске компьютера. Эти данные хранятся в виде ***файлов***.

Бывает *два вида* файлов:

1. *Текстовые*. Хранят данные в виде последовательности символов.
2. *Двоичные (бинарные)*. Хранят данные в виде последовательности байтов.

Текстовые файлы могут быть открыты в любом текстовом редакторе (например, «Блокноте» Windows) и просмотрены человеком как обычный текст. Текстовые данные характеризуются определенной *кодировкой*. Приведем определение данного термина с сайта компании Microsoft. ***Кодировка*** – это схема нумерации, согласно которой каждому текстовому символу в наборе соответствует определенное числовое значение. Кодировка может содержать буквы, цифры и другие символы. В различных языках часто используются разные наборы символов, поэтому многие из существующих кодировок предназначены для отображения наборов символов соответствующих языков.

Двоичные файлы используются для хранения *произвольного содержимого*, любой структурированной и неструктурированной информации (например, изображений, аудио, видео, документов, баз данных и т.д.). Открыть двоичный файл в текстовом редакторе можно, однако ничего осмысленного из него прочитать, скорее всего, не удастся, в текстовом виде содержимое такого файла будет представлять собой мешанину символов.

Общая схема работы с файлами в программе представлена на рис. 69.

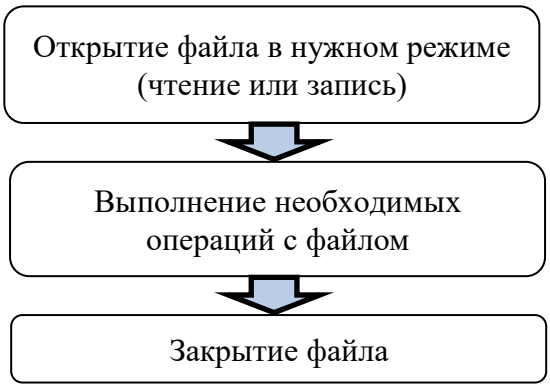


Рис. 69. Порядок работы с файлами в программе

Все, что можно делать с файлом, – это *считывать* из него и *записывать* в него данные. Прежде чем этим заниматься, файл нужно *открыть*, указав при этом, что именно мы хотим делать – читать или записывать. В момент открытия программа сообщает операционной системе компьютера, что данный файл занят, никакая другая программа работать с ним сейчас не должна. После того, как файл будет обработан, его обязательно нужно *закрыть*, то есть сообщить операционной системе, что файл освободился и доступен для открытия какой-нибудь другой программой.

Для открытия файла используется функция `open()`.

```
open(<путь к файлу>, mode=<режим>, encoding=<кодировка>)
```

Возвращаемым значением этой функции является *файловый объект* – специальный объект, «связанный» с файлом на диске, через который и осуществляется работа с этим файлом. Аргумент `mode` позволяет задать режим открытия файла. Его значение – это строка, состоящая из определенных символов (флагов). Возможные режимы:

`r` – чтение (значение по умолчанию; если файл отсутствует, возникнет ошибка);

`w` – запись (если файл отсутствует, он создастся; если файл существует, он перезапишется с начала);

`a` – дозапись (если файл отсутствует, он создастся; если файл существует, данные будут дописываться в его конец);

`+` – чтение и запись;

`t` – обработка файла как текстового (значение по умолчанию);

`b` – обработка файла как двоичного.

Флаги режимов могут комбинироваться: например, строка `rb` означает открытие двоичного файла на чтение. Для лучшего понимания разницы между теми или иными комбинациями флагов можно изучить иллюстрацию (рис. 70) и табл. 6.

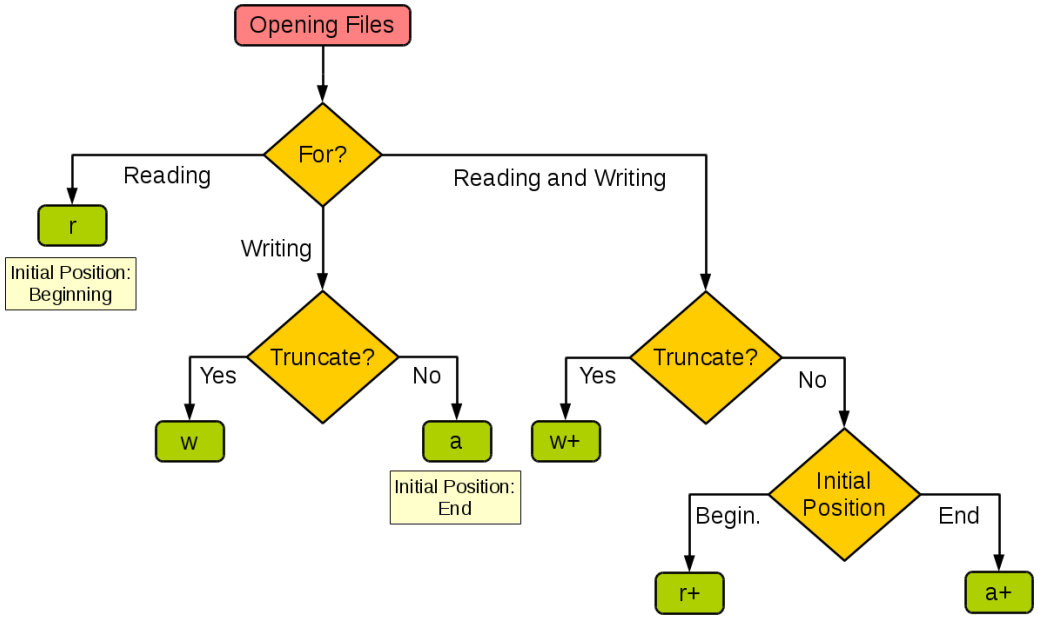


Рис. 70. Путеводитель по режимам открытия файла
Источник: https://stackoverflow.com

Таблица 6

Режимы открытия файла

Режим	r	r+	w	w+	a	a+
Чтение	+	+		+		+
Запись		+	+	+	+	+
Создание			+	+	+	+
Перезапись с нуля			+	+		
Указатель в начале	+	+	+	+		
Указатель в конце					+	+

Параметр `encoding`, задающий кодировку, применим *только для текстовых файлов*. Стандартной (общепринятой) кодировкой является *utf-8*.

Любой файл представляет собой *последовательность байтов* (в случае текстовых файлов байты содержат обычные строковые символы). При *чтении* данных обращение к файлу осуществляется *последовательно*, указатель текущей позиции при этом смещается вперед (рис. 71).

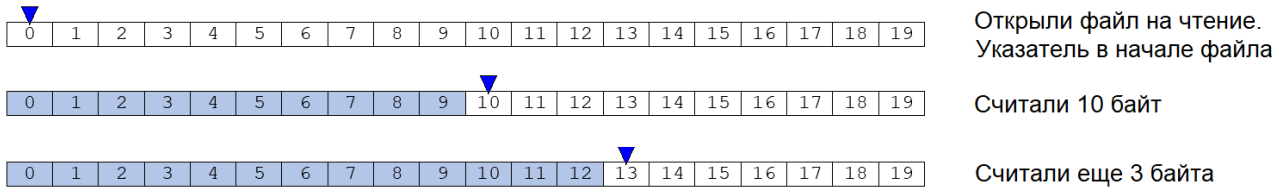


Рис. 71. Последовательное чтение файла

В случае *записи* очередная порция данных добавляется в конец файла. Если файл уже существовал, в зависимости от режима доступа запись данных либо продолжается, либо прежнее содержимое файла затирается, и запись начинается с самого начала.

Для *закрытия* файла используется метод файлового объекта `.close()`.

Текстовые файлы

Рассмотрим приемы работы с текстовыми файлами в Python. Начнем с методов *чтения*.

Метод `.read()` считывает заданное количество символов в строку, по умолчанию весь файл целиком (от текущей позиции указателя до конца файла).

Поработаем с существующим текстовым файлом «ucf.conf» из виртуальной среды Colab. Программный код нашего блокнота исполняется на виртуальной машине с ОС Linux. «/etc/ucf.conf» – это полный путь к файлу в Unix-стиле. По умолчанию режим работы с файлом – `rt` (чтение текстового файла), нам такой и нужен, поэтому можно его явно не задавать. Считаем файл целиком:

```
fobject = open("/etc/ucf.conf", encoding="utf-8")  # откроем файл
data = fobject.read()  # считаем содержимое файла целиком в строку
print(data)  # посмотрим на эту строку
fobject.close()  # не забудем закрыть файл
```

```
#
# This file is a bourne shell snippet, and is sourced by the
# ucf script for configuration.
...
```

Попробуем считывать файл порциями, смещая указатель:

```
fobject = open("/etc/ucf.conf", encoding="utf-8")

data = fobject.read(10)  # считаем первые 10 символов
print(f"'{data}'\n")  # посмотрим на считанную строку (используем кавычки,
# чтобы понять, где ее границы)
data = fobject.read(10)  # считаем вторые 10 символов
print(f"'{data}'\n")  # посмотрим
data = fobject.read()  # считаем остаток до самого конца
print(f"'{data}'")  # посмотрим

fobject.close()
```

```
'#
# This f'

'ile is a b'

'ourne shell snippet, and is sourced by the
# ucf script for configuration.
...
```

Метод `.readline()` считывает заданное число символов текущей строки, по умолчанию всю строку целиком. Отличие от метода `.read()` в том, что чтение происходит *построчно*. Считывание очередной порции данных заканчивается либо по достижении указанного числа символов, либо, если оно не указано, по достижении конца строки `\n`.

```
fobject = open("/etc/ucf.conf", encoding="utf-8")

data = fobject.readline(10)  # считаем первые 10 символов строки
print(f"'{data}'\n")  # 10 символов в первой строке не оказалось
data = fobject.readline(10)  # считаем первые 10 символов следующей строки
print(f"'{data}'\n")  # тут 10 нашлось, указатель остановился посреди строки
data = fobject.readline()  # считаем остаток строки целиком
print(f"'{data}'")  # посмотрим

fobject.close()
```

```
'#
'


'# This fil'

'e is a bourne shell snippet, and is sourced by the
'
```

Метод `.readlines()` считывает строки файла в список, но не больше чем заданное число символов, по умолчанию все строки файла. Если фрагмент из заданного числа символов заканчивается посреди строки, эта строка попадет в список целиком.

```
fobject = open("/etc/ucf.conf", encoding="utf-8")

data = fobject.readlines(100)  # считаем первые 100 символов в список
print(*data, sep="", end="==========\n")  # посмотрим на список
data = fobject.readlines()  # считаем остаток строк файла в список
print(*data[:3], sep="")  # посмотрим на первые 3 элемента списка

fobject.close()
```

```
#
# This file is a bourne shell snippet, and is sourced by the
# ucf script for configuration.
#

# Debugging information: The default value is 0 (no debugging
==========
# information is printed). To change the default behavior, uncomment
# the following line and set the value to 1.
#
```

Метод `.seek()` устанавливает указатель в файле на указанную позицию. Следовательно, текущей позицией указателя можно управлять.

```
fobject = open("/etc/ucf.conf", encoding="utf-8")

fobject.seek(50)  # установим указатель на 51-й символ (нумерация с 0)
data = fobject.read(10)  # считаем 10 символов, начиная с 51-й позиции
print(data)  # посмотрим на эту строку

fobject.close()
```

```
urced by t
```

Файловый объект, возвращаемый функцией `open()`, является коллекцией строк файла (итерируемым объектом). Следовательно, его строки можно последовательно считать в цикле `for`.

```
fobject = open("/etc/ucf.conf", encoding="utf-8")

for line in fobject:
  print(line.strip())  # строки файла оканчиваются символом переноса,
  # во избежание вывода пустых строк избавимся от него

fobject.close()
```

```
#
# This file is a bourne shell snippet, and is sourced by the
# ucf script for configuration.
...
```

После перечисления методов чтения и описания их работы может показаться, что чтение текстового файла – *задача не слишком тривиальная*. Однако при работе с текстовыми файлами небольшого объема (скажем, до одного мегабайта, то есть до миллиона символов) *практичным подходом* будет считывание файла в строку целиком и дальнейшая работа уже не с файловым объектом, а со строкой.

```
fobject = open("/etc/ucf.conf", encoding="utf-8")
data = fobject.read()  # считаем весь файл в строку
fobject.close()  # теперь содержимое файла - в строковой переменной data
```

Конечно, при чтении гигантских текстовых файлов (а такие бывают – например, файлы журналов высоконагруженных серверов) данный подход не годится, слишком велики окажутся затраты машинных ресурсов. В описанном случае подойдет только порционное (например, построчное) считывание.

Далее рассмотрим методы *записи* текстовых файлов.

Метод `.write()` записывает строковые данные в файл. Если необходимо создать многострочный файл (то есть содержащий символы переноса `\n`), эти символы должны присутствовать в записываемой строке.

```
# откроем файл в режиме записи. Флаг "w" нужно задать явно, а флаг "t",
# указывающий на работу с текстовым файлом, присутствует по умолчанию

fobject = open("myfile.txt", mode="w", encoding="utf-8")
# если указать лишь имя, а не полный путь к файлу, будет производиться
# попытка найти такой файл (либо создать в случае записи)
# в рабочем каталоге программы

data = "И на обломках\nСамовластья\nНапишут наши\nИмена"  # текст для записи
fobject.write(data)  # запишем строку в файл

fobject.close()
```

После выполнения приведенного кода файл myfile.txt появится в домашнем каталоге Colab, который можно просмотреть в обозревателе файловой системы, щелкнув по иконке «Файлы» на панели слева (рис. 72).

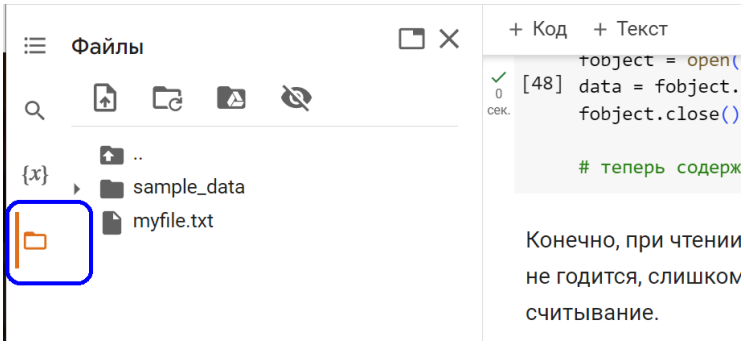


Рис. 72. Файловый обозреватель Colab

Содержимое файла можно просмотреть прямо в браузере, дважды щелкнув по его имени. Файлы из Colab можно скачивать на локальный компьютер, а также загружать в домашний каталог блокнота файлы с компьютера. Однако нужно иметь в виду, что файлы хранятся там лишь до тех пор, пока осуществляется активная работа с блокнотом и его состояние не сброшено, то есть хранилище это *временное*.

Метод `.writelines()` записывает все строки заданного списка строк в файл подряд. Если необходимо создать многострочный файл, каждая строка должна оканчиваться символом переноса `\n`:

```
fobject = open("myfile.txt", mode="w", encoding="utf-8")

# запишем список строк в файл
fobject.writelines(["Погрузитесь в мир\n",
                    "надвигающегося сумасшествия",
                    "и космического\n",
                    "ужаса."])
# изучая содержимое полученного файла, обратим внимание,
# что некоторые строки "слиплись" из-за отсутствия переносов

fobject.close()
```

Хорошо знакомая нам функция `print()` тоже умеет записывать данные в файл. Для этого при ее вызове нужно указать файловый объект в качестве значения аргумента `file`. Вывод работает привычным образом, в том числе срабатывают аргументы `sep` и `end`, только попадает все это не на экран, а в файл.

```
fobject = open("myfile.txt", mode="w", encoding="utf-8")

# запишем строки в файл (в качестве разделителя укажем \n)
print("И на обломках", "Самовластья", "Напишут наши", "Имена", sep="\n",
      file=fobject)
# завершающий вывод символ переноса допишется в конец файла как положено

fobject.close()
```

Оператор with (менеджер контекста)

Контекстный менеджер – это механизм, позволяющий высвободить захваченные ресурсы автоматически по окончании работы с ними. Ресурсом может быть файл на диске, подключение к базе данных, сетевое соединение. Если ресурс корректно не освободить, возможны утечки памяти, потеря данных и другие неприятные эффекты.

Контекстные менеджеры можно создавать. В Python для файлов уже есть готовый. Для использования контекстного менеджера с файлами применяется следующая конструкция:

```
with <файловый объект> as <псевдоним>:
    операции с файлом
```

По завершении выполнения кода в блоке `with` закрытие файла произойдет автоматически. Следовательно, метод файлового объекта `.close()` вызывать *не требуется*.

Рассмотрим следующий пример. Прочитаем файл, к каждой его строке добавим ее порядковый номер и запишем результат в новый файл.

```
# считаем ранее созданный файл с применением менеджера контекста
with open("myfile.txt", encoding="utf-8") as fobject:
  data = fobject.read()  # считываем содержимое файла в переменную-строку

# запишем файл с применением менеджера контекста
with open("myfile2.txt", mode="w", encoding="utf-8") as fobject:
  # перебираем нумерованный список подстрок
  for i, line in enumerate(data.splitlines(), 1):
    # записываем подстроку с номером в файл
    fobject.write(f"{i}. {line}\n")
```

Сериализация объектов (двоичные файлы)

Немного затронем и работу с двоичными файлами. Для обработки двоичных файлов определенной природы (например, аудио, изображений или файлов баз данных) существуют специальные библиотеки. Мы же рассмотрим механизм *сериализации*.

Сериализация – это преобразование произвольного объекта в последовательность байтов, то есть в файл. Если содержимым текстовых файлов является текст, который в оперативной памяти хранится в виде обычных объектов строкового типа, то в двоичные файлы можно записывать данные *любого типа*, включая привычные нам кортежи, списки или словари. Конечно, для корректного чтения и обработки таких файлов программист должен знать, как они устроены (что именно в них записано).

Для сериализации в Python используются модули `pickle` и `json`.

Модуль `pickle` позволяет сохранить *почти* любой (за некоторыми исключениями) объект на диск в виде двоичного файла. Для сохранения и загрузки используются функции `dump()` и `load()`.

```
import pickle  # подключим модуль

# создадим произвольный объект - список кортежей,
# хранящих числа от 1 до 10 и их квадратные корни
tuples = [(i, i ** 0.5) for i in range(1, 11)]

# укажем режим "запись двоичного файла"
with open("myfile.bin", mode="wb") as fobject:
  pickle.dump(tuples, fobject)  # запишем наш список в файл
```

```
# считаем этот файл и поместим его содержимое в переменную
# укажем режим "чтение двоичного файла"
with open("myfile.bin", mode="rb") as fobject:
  obj = pickle.load(fobject)  # запишем загруженный объект в переменную obj

print("Тип данных объекта, загруженного из файла:", type(obj))
print("\nСодержимое объекта:", *obj, sep="\n")
```

```
Тип данных объекта, загруженного из файла: <class 'list'>

Содержимое объекта:
(1, 1.0)
(2, 1.4142135623730951)
(3, 1.7320508075688772)
(4, 2.0)
(5, 2.23606797749979)
(6, 2.449489742783178)
(7, 2.6457513110645907)
(8, 2.8284271247461903)
(9, 3.0)
(10, 3.1622776601683795)
```

Модуль `json` позволяет работать со структурированными текстовыми файлами формата JSON.

JSON (**J**ava**S**cript **O**bject **N**otation) – это текстовый формат обмена данными, пригодный для описания данных сложной структуры. Примером другого подобного формата является **XML**. Будучи текстовым форматом, JSON является человекочитаемым, но обычно используется для обмена данными между различными программными системами. По своей структуре данные в этом формате очень похожи на *словарь* или на *список* (в зависимости от внутренней организации этих данных).

Сложные структуры данных, представляющие собой *словари списков или списки словарей* (и любые комбинации этих структур различных уровней вложенности), можно сериализовать в JSON и десериализовать обратно с помощью функций, аналогичных таковым в `pickle`.

```
import json  # подключим модуль

# считаем некий стандартный файл в формате JSON из хранилища Colab
with open("/content/sample_data/anscombe.json") as fobject:
  data = json.load(fobject)

print("Тип данных объекта, загруженного из файла:", type(data))
print("\nСодержимое объекта (первые 3 элемента):", data[:3], sep="\n")
```

```
Тип данных объекта, загруженного из файла: <class 'list'>

Содержимое объекта (первые 3 элемента):
[{'Series': 'I', 'X': 10.0, 'Y': 8.04}, {'Series': 'I', 'X': 8.0, ...}]
```

Можно заметить, что данный JSON-файл содержит в себе список словарей и именно в такую структуру и десериализуется. Загрузим другой JSON-файл из файловой системы Colab:

```
with open(".config/.last_update_check.json") as fobject:
  data = json.load(fobject)

print("Тип данных объекта, загруженного из файла:", type(data))
print("Содержимое объекта:", data, sep="\n")
```

```
Тип данных объекта, загруженного из файла: <class 'dict'>
Содержимое объекта:
{'last_update_check_time': 1701354287.463242, ...}
```

Этот файл десериализовался в словарь.

Также в модуле `json` есть функция `loads()`, которая преобразует в словарь или список заданную строку, если ее содержимое соответствует формату JSON:

```
# строка в JSON-формате
json_str = """{"Имя": "Вася", "Дети": {"Евкакий": {"Принадлежность": "Сын",
"Возраст": 5}, "Октябрина": {"Принадлежность": "Дочь", "Возраст": 9}}}"""

json.loads(json_str)  # загрузка этой строки в словарь
# (в зависимости от данных может быть и список)
```

```
{'Имя': 'Вася',
 'Дети': {'Евкакий': {'Принадлежность': 'Сын', 'Возраст': 5},
  'Октябрина': {'Принадлежность': 'Дочь', 'Возраст': 9}}}
```

10.4. Задания для самостоятельной работы

Решите следующие задачи:

1. С клавиатуры вводится последовательность чисел через пробел в одну строку. Для каждого числа выведите слово YES, если это число ранее встречалось в последовательности, или NO, если не встречалось. Для хранения уже встреченных чисел используйте множество.

2. В текстовом файле содержится следующий текст:

«Теперь несколько вопросов о земледелии и животноводстве. Скажите, пожалуйста, в течение последних 12 месяцев у Вашей семьи была в пользовании какая-либо земля? Скажите, пожалуйста, в настоящее время у Вашей семьи есть в пользовании какая-либо земля? Сколько всего соток земли у Вашей семьи в настоящее время? Давайте поговорим о том, кому принадлежит эта земля. Сейчас я прочту Вам несколько возможных вариантов, а Вы выберите, пожалуйста, только один ответ. Скажите, пожалуйста, в последние 12 месяцев Ваша семья платила за пользование землей?»

Напишите программу, которая считывает этот файл, отбирает из него только вопросительные предложения, формирует из них диалог, вставляя после каждого вопроса строку «Да» или «Нет», и сохраняет полученный текст в другой файл.

3. Студент готовится к контрольной работе по иностранному языку и по мере прочтения материала для подготовки помечает непонятные слова звездоч-

кой *, ставя ее в конце каждого такого слова. Помогите студенту собрать все непонятные слова и подсчитайте, сколько раз в тексте встречается каждое из них. Необходимо сформировать словарь, в котором ключами являются помеченные звездочками слова, а значениями – их частоты. Полученный словарь нужно сохранить в файл формата JSON. Входной файл для решения этой задачи следует сформировать самостоятельно, для чего взять любой понравившийся текст и отметить в нем некоторые слова указанным образом.

11. Работа с интернет-данными

Одиннадцатая глава посвящена основам извлечения данных из интернет-источников с применением двух популярных подходов. Изложенный материал не имеет непосредственного отношения к анализу данных, является в некоторой степени факультативным и не потребуется читателю при изучении второго раздела пособия. Однако навыки работы с REST API и парсинга веб-страниц хорошо дополняют умения специалиста по Data Science и могут быть полезны во многих ситуациях, связанных с нехваткой или отсутствием необходимых данных.

11.1. Данные в сети Интернет

Прежде чем приступать к анализу данных, необходимо эти данные для анализа где-то найти. Рассмотрим *два типичных случая* (рис. 73):

✓ Существует *уже готовый* набор данных в виде файла (например, форматов Excel или CSV), его достаточно только загрузить в программе. Если это так, то все замечательно, нам повезло. Примером заранее подготовленных данных в файле является статистика по экономическим показателям регионов на сайте Росстата[5].

✓ Второй случай сложнее. Готового набора данных *нет*, его нужно сформировать самостоятельно. Данные размещены на каком-нибудь веб-сайте в виде информации на страницах, которую легко просмотреть человеку, однако применить к ней количественные методы анализа затруднительно. Примером таких данных может быть каталог товаров интернет-магазина[6].

Рис. 73. Проблема поиска данных для анализа

Загружать табличные данные из форматов Excel и CSV мы научимся во втором разделе (в главе 13, посвященной библиотеке pandas), а в данной главе

[5] Регионы России. Социально-экономические показатели. URL: https://rosstat.gov.ru/folder/210/document/13204 (дата обращения: 30.12.2023).

[6] Ситилинк – интернет-магазин техники, электроники, товаров для дома и ремонта. URL: https://www.citilink.ru (дата обращения: 30.12.2023).

рассмотрим некоторые возможности, которые предоставляет Python для получения данных из интернета.

Интернет является гигантским хранилищем информации обо всем на свете. Данные хранятся и обрабатываются на специальных компьютерах – ***серверах***. Если к серверу отправить *правильно* составленный запрос на получение данных, он их предоставит. С процессом отправки запросов и получения ответов от интернет-серверов мы сталкиваемся постоянно. Это происходит, когда мы просматриваем веб-страницы или пользуемся любой программой, взаимодействующей с сетью (например, интернет-мессенджерами, такими как Viber, Telegram и т.д.).

Компьютер (а точнее, программа, работающая на этом компьютере), который отправляет запросы на обслуживание, называется ***клиентом***, а сама модель взаимодействия клиентов и серверов называется *клиент-серверной архитектурой* (рис. 74).

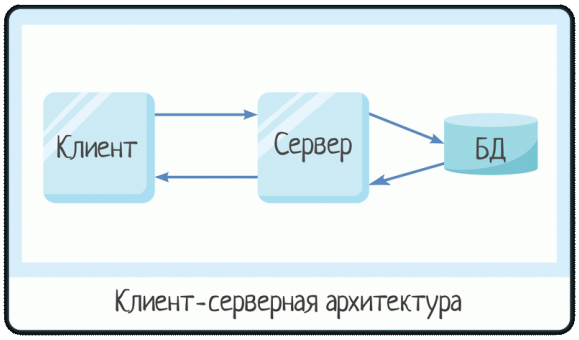


Рис. 74. Клиент-серверная архитектура
Источник: https://habr.com

Управлять процессом взаимодействия с серверами для отправки и получения нужных данных из интернета можно с помощью компьютерной программы. Почти в любом языке программирования предусмотрены соответствующие инструменты.

Существует *два популярных способа* получения данных из интернета:

- ✓ Использование специального программного интерфейса – REST API.
- ✓ Извлечение данных из кода веб-страниц.

Рассмотрим оба из них.

11.2. REST API

Первый способ позволяет отправить к веб-серверу запрос на получение *только нужных* данных. Данные на серверах хранятся в *базах данных* (БД). Веб-сервер, получив корректно составленный в соответствии с некоторыми прави-

лами запрос, извлекает данные из БД и отдает их клиенту. Эти правила называются ***API*** (**A**pplication **P**rogramming **I**nterface) – это специальный протокол, описывающий и стандартизирующий взаимодействие программных систем, в данном случае клиента и сервера.

Достоинство данного способа очевидно: сервер выдает только нужную (запрошенную у него) информацию, причем в виде, удобном для обработки. Использование REST API является *предпочтительным* способом взаимодействия с интернет-серверами. Однако есть у него и *недостатки*:

✓ Не у всех интернет-серверов есть свой API. Примером интернет-сервиса с общедоступным API является Telegram: любой программист может написать программу для отправки сообщений в этом мессенджере, используя возможности REST API.

✓ Не всегда API является общедоступным и бесплатным в использовании. Нередко за доступ к нему приходится платить. Примеры API с платным доступом (возможно бесплатное использование с ограничениями): API сервисов Яндекса (например, Карты, Такси).

Лучше всего изучать такие достаточно сложные технологии, как REST API, в процессе решения практической задачи. Сквозной задачей, разбираемой в данном параграфе, будет **получение данных о вакансиях, размещенных на сайте HeadHunter** – крупнейшей российской компании, занимающейся публикацией объявлений от работодателей и соискателей работы. (Отметим, что все приведенные ниже фрагменты программного кода являются рабочими на момент публикации данного пособия.)

Чтобы правильно составить запрос к серверу и суметь корректно распознать ответ, нужно изучить документацию по API. Документация для HeadHunter доступна по следующей ссылке: [3].

Библиотека Requests

Для отправки запросов к веб-серверам в Python используется библиотека Requests. Эта библиотека не является стандартной, то есть не распространяется вместе с интерпретатором Python, ее необходимо устанавливать отдельно. Однако в Colab она уже предустановлена. Для отправки запросов типа GET (изучение особенностей протокола передачи интернет-данных HTTP и его методов GET и POST выходит за рамки нашего материала) служит функция `get()`:

```
get(<адрес веб-ресурса>, headers=<словарь заголовков запроса>,
                         params=<словарь параметров запроса>)
```

Адрес веб-ресурса – это строка для обращения к API сервера. Ее можно узнать из документации. *Заголовки* запроса (аргумент `headers`) – это структура данных, содержащая служебные параметры запроса (например, сведения о клиенте, отправляющем запрос). *Параметры* запроса (аргумент `params`) – это аналогичная структура, содержащая информацию о конкретных параметрах запроса (указание на то, какие именно данные и в каком виде необходимо получить от сервера).

Функция `get()` возвращает специальный объект – экземпляр класса `Response`, который содержит в себе всю информацию, включая множество служебной, об ответе сервера на запрос. У данного объекта есть *атрибут* `.text` – значение строкового типа, содержащее непосредственно запрошенные данные в текстовом виде. Значение атрибута `.text` представляет собой структурированный текст в определенном стандартном формате, например XML или JSON (эти форматы упоминались в параграфе 10.3).

Небольшая ремарка: ранее с понятием *атрибутов объектов* мы не сталкивались, говорили лишь о *методах* – функциях (некоторых алгоритмах), «привязанных» к объекту. У объектов сложной структуры, а ответ от сервера таким определенно является, помимо методов могут быть ***атрибуты*** – это данные, «привязанные» к объекту. Можно провести следующую аналогию: методы – это *функции*, принадлежащие объекту, а атрибуты – его *переменные*. До сих пор мы рассматривали *сам объект* как некоторые данные. Действительно, при выводе объекта на экран можно наблюдать именно хранимые в нем данные, значения (а вовсе не код его методов). Однако со столь сложно организованными иерархическими объектами, как ответ от сервера (экземпляр класса `Response`), мы в рамках материала еще не сталкивались. Данный объект содержит в себе такое количество информации, что просматривать и обрабатывать ее целиком невозможно, приходится обращаться к отдельным ее фрагментам. Для этого и служат атрибуты. Чтобы узнать об объектах больше, следует предметно изучать объектно-ориентированное программирование, а мы приступим к написанию программы.

Получение данных через API: пример

Прежде всего подключим необходимые библиотеки. API HeadHunter возвращает данные в формате JSON, поэтому воспользуемся функциями из соответствующего модуля:

```
import requests  # подключим библиотеку requests
import json  # подключим библиотеку для работы с данными в формате JSON
```

Прежде всего попробуем получить вообще все открытые вакансии на сайте. Согласно документации, общий адрес для отправки запросов к API такой: https://api.hh.ru. Нас интересуют данные об опубликованных вакансиях, эту информацию можно получить по следующему адресу:

```
url = "https://api.hh.ru/vacancies"  # сохраним адрес запроса в переменную
```

Укажем в качестве заголовка запроса так называемый *user agent*: это параметр, характеризующий отправляющего запрос клиента. Многие веб-сервера отказывают в обработке запросов клиентов, если он не указан в заголовках запроса. Строковое значение *user agent* можно посмотреть, изучив запросы своего веб-браузера (в консоли разработчика) или воспользовавшись специальным сайтом (например, ссылкой [18]).

```
# сохраним user agent в переменную
ug = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 " \
"(KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36"
```

```
# отправим GET-запрос к серверу, указав в заголовках словарь с user agent
response = requests.get(url, headers={"User-Agent": ug})
response  # посмотрим на ответ от сервера
```

```
<Response [200]>
```

Обратите внимание, как отображается при выводе содержание сложного объекта `Response`. Ответ от сервера содержит множество различной информации, и на экран выводится только самое важное – в данном случае код ответа. Код 200 означает, что запрос успешно обработан.

```
response.status_code  # получим код ответа отдельным значением
```

```
200
```

```
response.text  # посмотрим непосредственно на текст ответа
```

```
{"items":[{"id":"88686062","premium":false,"name":"ОФИЦИАНТ (ИГРОВОЙ
КЛУБ)","department":null,"has_test":false,"response_letter_required":false,"ar
ea":{"id":"1","name":"Москва","url":"https://api.hh.ru/areas/1"},"sal-
ary":null,"type":{"id":"open","name":"Открытая"}, ...
```

Значение `.text` представляет собой очень длинную строку формата JSON с данными о вакансиях.

```
# преобразуем JSON-строку в привычный словарь
response_dict = json.loads(response.text)
```

```
# чтобы разобраться со структурой ответа, вновь обратимся к документации
# список вакансий находится в элементе словаря с ключом "items",
vacancies_list = response_dict["items"]
```

```
# а общее количество вакансий - в элементе с ключом "found"
print("Всего вакансий на HeadHunter:", response_dict["found"])

# посмотрим, сколько записей о вакансиях содержится в ответе сервера
print("Количество вакансий в списке:", len(vacancies_list))
```

```
Всего вакансий на HeadHunter: 1414739
Количество вакансий в списке: 20
```

Как оказалось, сервер вернул данные только о 20 вакансиях. То, что ответ содержит *не* абсолютно все вакансии на сайте, вполне ожидаемо: их огромное количество (более миллиона), и ответ с полным их перечнем был бы слишком большим.

```
vacancies_list[0]  # посмотрим на данные о первой вакансии в списке
```

```
{'id': '88686062',
 'premium': False,
 'name': 'ОФИЦИАНТ (ИГРОВОЙ КЛУБ)',
 'department': None,
 'has_test': False,
 'response_letter_required': False,
 'area': {'id': '1', 'name': 'Москва', 'url': 'https://api.hh.ru/areas/1'},
 'salary': None,
...
```

Как видим, элемент списка – это словарь с полным описанием вакансии. Название вакансии хранится в элементе словаря с ключом `"name"`.

```
# переберем все вакансии из списка по названию
for vac in vacancies_list:
  print(vac["name"])
```

```
ОФИЦИАНТ (ИГРОВОЙ КЛУБ)
Менеджер по продажам
Менеджер по развитию продаж HoReCa (рестораны)
Удаленный диспетчер чатов (в Яндекс)
Удаленный специалист службы поддержки (в Яндекс)
...
```

Теперь конкретизируем наш запрос: получим данные по *открытым вакансиям в Перми в сфере разработки программного обеспечения*. Чтобы правильно составить запрос, вновь надо внимательно изучить документацию API. Из нее можно узнать, что справочник всех регионов хранится по адресу https://api.hh.ru/areas, там можно посмотреть коды регионов. В частности, Пермь имеет код 72. За указание конкретного региона по коду отвечает параметр GET-запроса *area*. Адрес справочника отраслей: https://api.hh.ru/industries, разработка ПО имеет код 7.540, параметр запроса для указания отрасли – *industry*.

Важный момент: как правило, сервер выдает не весь запрошенный объем информации разом (вакансий может быть слишком много, мы в этом уже убедились), а делит возвращаемые данные на так называемые *страницы*. Когда мы

просматриваем, например, новости или товары в каталоге интернет-магазина, их перечень помещается на веб-страницу тоже не целиком, а порциями (по 10, 20, 50, 100 элементов, иногда это число можно выбрать самому). Данный механизм деления большого объема выдаваемой сервером информации на порции называется ***пагинацией*** (от англ. *page* – страница).

Сервер HeadHunter тоже выдает данные постранично, и этим можно управлять. В очередной раз изучив документацию, выясним, что можно указать число вакансий на странице (параметр запроса *per_page*), а также номер запрашиваемой страницы (параметр *page*). Таким образом можно перебирать все страницы до тех пор, пока список вакансий, удовлетворяющих условиям, не закончится. Общее количество страниц содержится в элементе словаря-ответа `"pages"`, о чем можно узнать, исследовав содержимое этого словаря.

```
# сконструируем словарь с параметрами запроса
parameters = {
  "area": "72",  # регион - Пермь (значение - строка)
  "industry": "7.540",  # область - разработка ПО (строка)
  "per_page": 100,  # 100 вакансий на странице
  "page": 0  # страница 0 (на самом деле первая,
             # но у API HeadHunter нумерация страниц с нуля)
}

# отправим новый запрос, на этот раз укажем уточняющие параметры
response = requests.get(url, headers={"User-Agent": ug},
                        params=parameters)  # параметры запроса

response_dict = json.loads(response.text)
vacancies_list = response_dict["items"]

# выведем названия первых 10 полученных вакансий
for vac in vacancies_list[:10]:
  print(vac["name"])
```

```
Специалист по ручному тестированию
Гид-экскурсовод (в офисе на предприятии)
Специалист по обработке данных
Бортпроводник
Сторож-охранник
Специалист по тестированию
Frontend-разработчик
Frontend-разработчик (React)
Специалист по кадрам
Специалист поддержки (в чате)
```

Как можно заметить, не все вакансии соответствуют сфере «разработка ПО» (например, какое отношение к разработке имеет сторож-охранник?), однако такие данные предоставляет HeadHunter: видимо, не все вакансии попадают в релевантный раздел.

Но мы получили только первые 100 вакансий. Как получить полный их список? Возможен следующий подход: перебирать страницы до тех пор, пока запрос не будет выполнен *неуспешно* (сервер вернет ответ с кодом, не равным

200). Конечно, это очень грубый способ, потому что неуспех запроса может быть обусловлен разными факторами – например, отсутствием интернета на клиенте, проблемами на сервере или злоупотреблением (слишком частой отправкой запросов клиентом). К тому же конкретно сервис HeadHunter при обращении к излишне большому номеру страницы по-прежнему сообщает, что запрос выполнен успешно (код ответа 200), просто список вакансий для несуществующих страниц оказывается пустым. Кроме того, общее количество вакансий содержится в элементе словаря-ответа `"found"`, а общее количество страниц – в элементе `"pages"`, что позволяет организовать при переборе страниц цикл не с условием, а арифметический, ведь мы точно знаем, сколько страниц надо перебрать. При работе с разными интернет-сайтами и их API следует поступать по ситуации, пытаться применить различные подходы. В нашем примере же используем в качестве критерия остановки перебора страниц *пустоту списка вакансий*:

```
import time  # модуль для работы со временем

vacancies_names = []  # сформируем список названий вакансий
page = 0
while True:  # запустим "вечный" цикл
  # будем обновлять номер страницы в параметрах запроса
  parameters.update({"page": page})

  print(f"Запрос на получение {page}-й страницы...", end="\t")
  response = requests.get(url, headers={"User-Agent": ug}, params=parameters)
  response_dict = json.loads(response.text)  # ответ в виде словаря
  vacancies_list = response_dict["items"]  # список очередных 100 вакансий

  # сделаем паузу 3 секунды между запросами,
  # чтобы сервер не начал отказывать нам в обслуживании
  time.sleep(3)

  # если список вакансий пуст, прерываем цикл
  # (страницы с вакансиями закончились)
  if len(vacancies_list) == 0:
    print("\tошибка")
    break
  print("\tуспех")
  # переберем названия вакансий и добавим в список названий
  for vac in vacancies_list:
    vacancies_names.append(vac["name"])

  page += 1  # не забудем увеличить номер страницы

print("Получение списка вакансий завершено")
```

```
Запрос на получение 0-й страницы...         успех
Запрос на получение 1-й страницы...         успех
Запрос на получение 2-й страницы...         успех
Запрос на получение 3-й страницы...         успех
Запрос на получение 4-й страницы...         успех
Запрос на получение 5-й страницы...         успех
Запрос на получение 6-й страницы...         успех
Запрос на получение 7-й страницы...         ошибка
Получение списка вакансий завершено
```

Взглянем на содержимое сформированного списка названий вакансий:

```
print("Всего найдено вакансий, удовлетворяющих условиям:",
      len(vacancies_names))
print("\nПоследние 5 вакансий:")
for vac_name in vacancies_names[-5:]:
  print(f"\t{vac_name}")
```

```
Всего найдено вакансий, удовлетворяющих условиям: 614

Последние 5 вакансий:
        Менеджер по продажам МегаФон
        Менеджер по работе с клиентами (удаленно)
        Специалист технической поддержки (b2b)
        Специалист отдела продаж и клиентского сервиса (удаленно)
        Продавец-консультант в салон связи
```

В рассмотренном примере для каждой вакансии мы сохраняли только название. Ответ от сервера содержит полные данные о вакансии, и при необходимости их можно извлечь из словаря с данными ответа.

Как можно заметить, получать данные через REST API достаточно просто, нужно только изучить документацию. От наличия, качества и полноты документации к API во многом зависит трудоемкость работы с ним, но главное, конечно, чтобы API ресурса с интересующими нас данными вообще существовал, позволял эти данные получать и был нам доступен. Так бывает далеко не всегда. В следующем параграфе рассмотрим второй способ получения данных из интернета, более трудоемкий, однако позволяющий извлекать информацию почти с любого интернет-сайта.

11.3. Извлечение данных из веб-страниц

Веб-скрейпинг

Извлечение данных из содержимого страниц интернет-сайтов называют ***веб-скрейпингом*** (или ***парсингом*** веб-страниц). Идея подхода состоит в следующем. Веб-страницы предназначены для просмотра человеком, и, следовательно, любая информация, опубликованная в интернете и доступная на веб-сайтах, *может быть получена*. Веб-страница с информацией есть у любого сайта – в отличие от API, которые есть не у всех веб-ресурсов и (или) не всем доступны. Программа может «притвориться» человеком (а точнее, *веб-браузером*, с помощью которого человек просматривает информацию из интернета) и получить все необходимые данные в виде веб-страницы: сервер эти данные отдаст (во всяком случае, если притвориться достаточно хорошо).

Достоинства веб-скрейпинга:

✓ С его помощью можно *программно* (то есть не вручную через браузер, а с помощью программного кода) получить *любую* информацию, доступную в интернете.

✓ Не требуется изучать документацию для конкретного веб-сайта, технологии формирования веб-страниц стандартны и используются повсеместно.

Но есть и *недостатки*:

✓ Веб-страницы (в отличие от ответов REST API) помимо полезного содержимого включают множество *«лишних» данных*: различные скрипты, рекламные баннеры и т.п. Полезную информацию из кода веб-страниц необходимо добывать.

✓ Современная Всемирная паутина (World Wide Web, WWW) достаточно сложно устроена. Содержимое страниц сайтов формируется динамически, зачастую из множества разных источников. Ввиду этого не все веб-страницы легко подвергаются извлечению данных.

Из написанного следует закономерный *вывод*: если есть возможность для получения информации с веб-ресурса использовать REST API, нужно его использовать. Если же такой возможности нет, придется вооружаться инструментами веб-скрейпинга. Рассмотрим эти инструменты подробнее.

Веб-страницы предназначены для удобного чтения человеком. Однако часто возникает потребность извлечения нужной информации из интернета автоматизированным способом, и здесь «человекоориентированность» веб-сайтов является больше недостатком, чем преимуществом. Представим себе задачу: нужно получать данные об индексе инфляции с сайта Центробанка (рис. 75) и выводить их в какой-нибудь программе (например, в бухгалтерской системе 1С). Как это делать не вручную, открыв страницу и посмотрев на нее (глазами), а *автоматически*?[7]

Исследуем поставленную проблему.

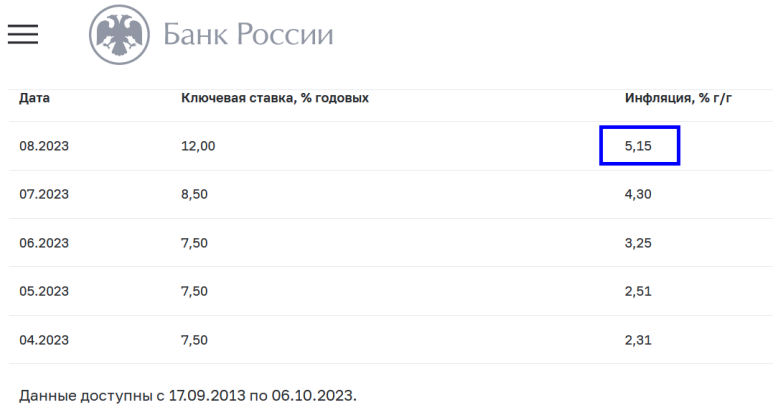


Рис. 75. Данные об инфляции на сайте Центробанка
Источник: https://cbr.ru

[7] Может быть, приведенный пример является не самым показательным, потому что у Центробанка есть открытый REST API (https://www.cbr.ru/lk_uio/guide/rest_api). Однако существует множество веб-сайтов с полезной информацией, у которых никакого API нет, и извлечь с них данные можно только с помощью скрейпинга.

Процесс формирования веб-страницы

С точки зрения компьютера веб-страница является высокоструктурированным объектом, поэтому находить на ней нужную информацию несложно (в том случае, если она там есть).

Веб-страница – это текстовый файл, содержащий разметку на специальном языке **HTML** (**H**yper**T**ext **M**arkup **L**anguage). Визуальное представление веб-страницы, которое отображает веб-браузер, – это результат обработки им HTML-кода страницы (рис. 76). Чтобы посмотреть код любой страницы, следует нажать правой кнопкой мыши в произвольном месте страницы и выбрать соответствующий пункт контекстного меню.

```
344     <main id="content">
345         <div class="offsetMenu">
346             <div class="container-fluid">
347                 <div class="col-md-23 offset-md-1">
348                     <div class="breadcrumbs">
349     <div class="breadcrumbs_item"><a class="breadcrumbs_home" href="/"></a></div>
350         <div class="breadcrumbs_item">
351             <a href="/lk_uio/"><!--noindex-->Личный кабинет участника информационного обмена<!--/noindex--></a>
352         </div>
353         <div class="breadcrumbs_item">
354             <a href="/lk_uio/guide/"><!--noindex-->Инструкции и иная информация о технологии подготовки
355         </div>
356     </div>
357
358                     <!--index-->
359 <h1><span class="referenceable">REST-API сервис</span>
```

Рис. 76. Фрагмент HTML-кода некоторой веб-страницы

Парсинг веб-страниц (от англ. *parse* – разбирать) – это процесс синтаксического анализа HTML-кода и извлечения из него необходимой информации.

Загрузка веб-страницы в браузер происходит следующим образом (рис. 77):

1. Клиентский веб-браузер посылает веб-серверу запрос, содержащий адрес страницы и параметры ее формирования. Как правило, при обычном *интернет-серфинге* (просмотре веб-сайтов) такие запросы посылаются в результате перехода пользователя по гиперссылкам.
2. Веб-сервер в ответ на полученный запрос формирует страницу в виде HTML-кода и возвращает ее клиенту.
3. Клиентский браузер обрабатывает полученный от сервера код и выводит визуальное представление веб-страницы на экран. Осуществляет эту операцию так называемый *браузерный движок* (наиболее популярным в мире по данным на 2023 г. является *Blink* – движок браузера Google Chrome и многих других).

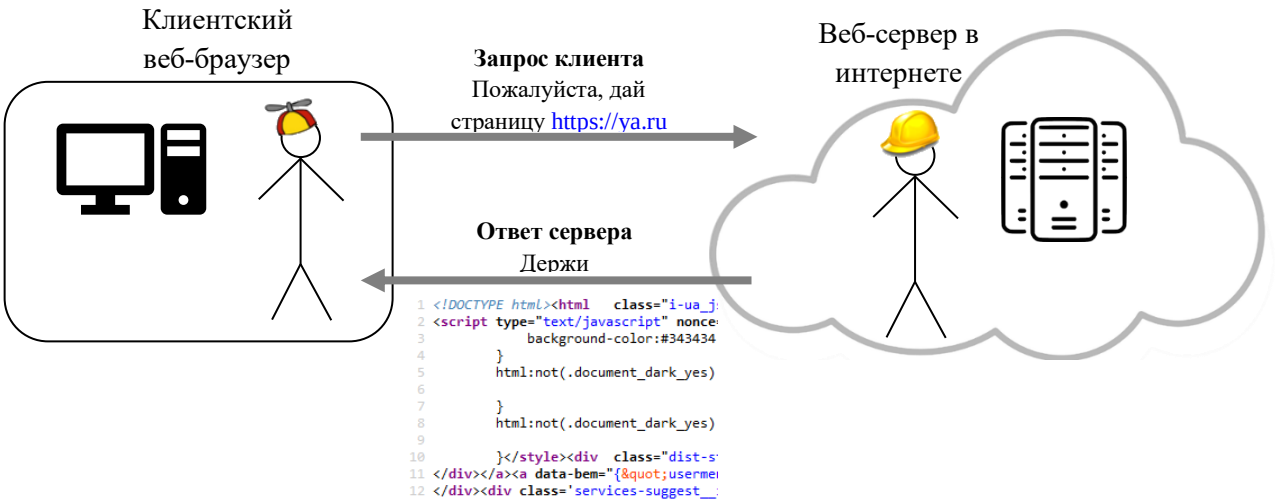


Рис. 77. Процесс получения веб-страницы от сервера
Источник нескольких пиктограмм: https://www.pngwing.com

Процесс получения веб-страниц от веб-сервера полностью аналогичен обращениям к REST API, *с той разницей*, что API возвращает не код веб-страницы, а *структуру данных*, содержащую только ту информацию, которая была запрошена клиентом. Можно сказать, что API предназначен для информационного обмена ***«машина ↔ машина»***. Типичная же обязанность веб-сервера заключается в обеспечении взаимодействия ***«человек ↔ машина»***, и веб-скрейпинг эксплуатирует именно эту схему, программа выдает себя в ней за *человека*.

Устройство веб-страницы

Веб-страница в виде HTML-кода обладает *иерархической структурой* и состоит из элементов языка HTML (тегов).

HTML-теги – это специальные ключевые слова, описывающие данные и их визуальное представление в браузере. Теги выделяются треугольными скобками <>. Бывают *теги-контейнеры*, состоящие из пары ***открывающий-закрывающий тег***, а также *пустые теги*, не имеющие закрывающей части.

Примером тега-контейнера является тег `<a>`, используемый для организации гиперссылок:

```
<a href="Адрес ссылки">Текст ссылки на странице</a>
```

Внутри тега-контейнера может быть не только текст, но и *вложенные теги* (веб-страница – иерархическая структура). Примером пустого тега является тег `<img>`, используемый для вставки изображения на веб-страницу:

```
<img src="Адрес изображения">
```

Теги могут иметь *атрибуты*, принимающие то или иное значение. (Не следует путать атрибуты тега с *атрибутами объекта*, это термины из разных предметных областей, имеющие, впрочем, схожий смысл: атрибут – это характеристика сущности, ее реквизит, свойство.) Например, так выглядит в HTML-коде гиперссылка на главную страницу сайта Mail.ru (рис. 78).

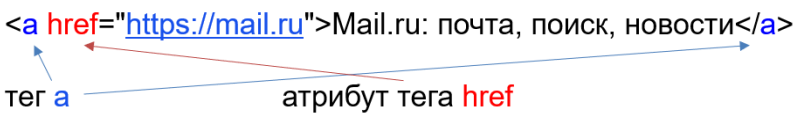


Рис. 78. Тег-контейнер с атрибутом (гиперссылка)

При парсинге HTML-кода совокупность тегов и их атрибутов позволяет находить и извлекать нужные данные.

Библиотека BeautifulSoup

Для парсинга HTML-кода существует Python-библиотека BeautifulSoup. Ее инструменты позволяют обрабатывать текстовую строку с кодом как объект, структурированный определенным образом. Для загрузки HTML-кода используется следующая функция:

```
BeautifulSoup(<строка с кодом>, features="html.parser")
```

Аргумент `features` позволяет задать набор алгоритмов (так называемый *движок*) парсинга страницы (парсер). По умолчанию используется *html.parser*, популярным парсером является *lxml* (чтобы применить этот движок, должна быть установлена одноименная библиотека).

Функция `BeautifulSoup()` возвращает объект – экземпляр класса с таким же именем. Этот объект содержит методы `.find()` и `.find_all()`.

```
find(<тег>, <набор атрибутов тега со значениями>)
```

Метод `.find()` возвращает *первый* извлеченный из кода страницы тег, соответствующий заданным условиям. Метод `.find_all()` отличается тем, что возвращает *коллекцию* всех найденных на странице подходящих тегов.

Каждый элемент страницы (тег) является экземпляром класса `Tag`, из которого, в свою очередь, можно извлечь содержащиеся в нем теги с помощью все тех же методов. Также объект-тег содержит атрибут `.parent`, который является ссылкой на родительский для данного объекта тег-контейнер (в свою очередь, объект `Tag`). Например, в приведенном ниже HTML-коде родительским для тега `<b>` является тег `<head>`:

```
<head>
      <b>Заголовок</b>
</head>
```

Чтобы извлечь содержимое тега-контейнера из объекта `Tag`, можно применить его метод `.get_text()` или обратиться к атрибуту `.text` (с тем же эффектом). При этом из контейнера будет извлечен *только текст:* даже если в нем есть и другие теги, их код будет проигнорирован и в возвращаемое строковое значение не попадет.

Извлечение данных из веб-страницы: пример 1

Рассмотрим работу с BeautifulSoup на примере. Пусть имеется некоторая веб-страница с заголовком и двумя таблицами. Ниже приведены визуальное представление страницы в веб-браузере (рис. 79) и ее HTML-код (рис. 80).

Данные о компании по продаже электросамоваров

Клиенты

Имя	Возраст
Иван	21
Марина	30
Феофил	87

Офисы

Адрес	Количество сотрудников	Наличие платежного терминала
Ул. Куйбышева, 17	2	Нет
Ул. Коминтерна, 555	5	Да

Рис. 79. Простейшая веб-страница

Решим задачу: извлечь заголовок страницы и содержимое второй таблицы («Офисы»). Загрузим код веб-страницы с помощью GET-запроса:

```
import bs4  # модуль библиотеки BeautifulSoup называется bs4
from requests import get  # для удобства подключим сразу функцию get()

# веб-страница учебная, файл с ее кодом опубликован в интернете
# и доступен по прямой ссылке
html_code = get(
  "https://raw.githubusercontent.com/rvgarafutdinov/python_book_2023/" \
  "main/htmlsample.html"
).text

# аргумент features можно опустить, использовать парсер по умолчанию
# сохраним объект - веб-страницу в переменную soup
# (название из документации библиотеки)
soup = bs4.BeautifulSoup(html_code)
soup  # в текстовом виде этот объект представляет собой код страницы
```

```
<!DOCTYPE html>
<html>
<head><title>самовары</title></head>
...
```

```
<!DOCTYPE html>
<html>
<head><title>самовары</title></head>
<body><h4><b>Данные о компании по продаже электросамоваров</b></h4>
<table>
    <tr>
        <td>
            <div class="table1"><h5>Клиенты</h5>
                <table bgcolor="Aquamarine">
                    <tr>
                        <th>Имя</th>
                        <th>Возраст</th>
                    </tr>
                    <tr>
                        <td>Иван</td>
                        <td>21</td>
                    </tr>
                    <tr>
                        <td>Марина</td>
                        <td>30</td>
                    </tr>
                    <tr>
                        <td>Феофил</td>
                        <td>87</td>
                    </tr>
                </table>
            </div>
        </td>
        <td>
            <div class="table2">
                <h5>Офисы</h5>
                <table bgcolor="CornflowerBlue">
                    <tr>
                        <th>Адрес</th>
                        <th>Количество<br>сотрудников</th>
                        <th>Наличие платежного<br>терминала</th>
                    </tr>
                    <tr>
                        <td>Ул. Куйбышева, 17</td>
                        <td>2</td>
                        <td>Нет</td>
                    </tr>
                    <tr>
                        <td>Ул. Коминтерна, 555</td>
                        <td>5</td>
                        <td>Да</td>
                    </tr>
                </table>
            </div>
        </td>
    </tr>
</table>
</body>
</html>
```

Рис. 80. HTML-код веб страницы

Сначала извлечем заголовок страницы. Он находится в первом на странице теге <b> (рис. 81).

```
<!DOCTYPE html>
<html>
<head><title>самовары</title></head>
<body><h4><b>Данные о компании по продаже электросамоваров</b></h4>
<table>
    <tr>
```

Рис. 81. Первый тег <b> на странице

```
# для извлечения первого тега подойдет метод find()
title_tag = soup.find("b")  # получим объект-тег
title_tag  # посмотрим на содержимое данного объекта
```
```
<b>Данные о компании по продаже электросамоваров</b>
```

```
# извлечем содержимое тега (текст заголовка)
title = title_tag.get_text()
title
```
```
'Данные о компании по продаже электросамоваров'
```

Теперь извлечем содержимое второй таблицы, включая шапку и данные в строках. Можно заметить, что на странице *три* тега `<table>`. Нас же интересует только один. Еще раз внимательно посмотрев на код, увидим, что вторая таблица находится внутри тега `<div>`. Но таких тегов тоже больше одного – два. Нам нужен второй из них, который содержит атрибут *class* со значением *table2* (рис. 82).

```
<td>
    <div class="table2">
        <h5>Офисы</h5>
        <table bgcolor="CornflowerBlue">
            <tr>
                <th>Адрес</th>
                <th>Количество<br>сотрудников</th>
                <th>Наличие платежного<br>терминала</th>
            </tr>
            <tr>
                <td>Ул. Куйбышева, 17</td>
                <td>2</td>
                <td>Нет</td>
            </tr>
            <tr>
                <td>Ул. Коминтерна, 555</td>
                <td>5</td>
                <td>Да</td>
            </tr>
        </table>
    </div>
</td>
```

Рис. 82. Второй тег на странице

```
# извлечем нужный тег, атрибуты тега зададим в виде словаря
table2_tag = soup.find("div", {"class": "table2"})
table2_tag  # взглянем на результат
```
```
<div class="table2">
<h5>Офисы</h5>
<table bgcolor="CornflowerBlue">
...
```

Поскольку вторая таблица находится внутри тега `<div class="table2">`, далее будем работать не со всей веб-страницей целиком (объект `soup`), а только с данным тегом (объект `table2_tag`). Можно извлечь тег `<table>`, он здесь такой единственный. Впрочем, в этом нет необходимости, потому что `<div>` почти

целиком состоит из тега <table> и выделение <table> в отдельный объект нецелесообразно.

Извлечем данные из шапки таблицы (заголовки столбцов). Они находятся в первой строке таблицы (первый тег <tr>). Строго говоря, в данном случае для получения заголовков (теги <th>) тоже не обязательно извлекать первый тег <tr>, потому что <th> встречаются только в нем и все теги <th>, которые будут обнаружены в коде рассматриваемого тега , будут относиться к шапке таблицы, лишних не окажется. Но нам все равно понадобится извлечь теги <tr> для доступа к остальным строкам, поэтому извлечем их все (рис. 83).

```
<div class="table2">
    <h5>Офисы</h5>
    <table bgcolor="CornflowerBlue">
        <tr>
            <th>Адрес</th>
            <th>Количество<br>сотрудников</th>
            <th>Наличие платежного<br>терминала</th>
        </tr>
        <tr>
            <td>Ул. Куйбышева, 17</td>
            <td>2</td>
            <td>Нет</td>
        </tr>
        <tr>
            <td>Ул. Коминтерна, 555</td>
            <td>5</td>
            <td>Да</td>
        </tr>
    </table>
</div>
```

Рис. 83. Теги <tr> во втором теге

```
# все <tr>-теги одинаковы, различить их по атрибутам не удастся,
# поэтому получим полный их список
tr_tags = table2_tag.find_all("tr")
# для красоты выведем каждый элемент списка в отдельной строке
print(*tr_tags, sep="\n\n")
```

```
<tr>
<th>Адрес</th>
<th>Количество<br/>сотрудников</th>
<th>Наличие платежного<br/>терминала</th>
</tr>

...
```

Теперь из первой строки (первого тега <tr>) извлечем заголовки (теги <th>) (рис. 84).

```
<tr>
    <th>Адрес</th>
    <th>Количество<br>сотрудников</th>
    <th>Наличие платежного<br>терминала</th>
</tr>
```

Рис. 84. Теги <th> в первом теге <tr>

```
# для извлечения всех тегов контейнера используем метод find_all()
th_tags = tr_tags[0].find_all("th")
th_tags  # коллекция тегов очень похожа на обычный список, но это не он
```

```
[<th>Адрес</th>,
 <th>Количество<br/>сотрудников</th>,
 <th>Наличие платежного<br/>терминала</th>]
```

```
# переберем <th>-теги в цикле и из каждого извлечем содержимое
for th_tag in th_tags:
  # в HTML-коде используется HTML-символ переноса строки <br>,
  # аргумент separator его заменяет на пробел (иначе слова слипнутся)
  col_head = th_tag.get_text(separator=" ")  # заголовок колонки
  print(col_head, end="\t")  # выведем шапку в одну строку
print()
```

```
Адрес   Количество сотрудников Наличие платежного терминала
```

Шапку таблицы получили, теперь выведем остальные строки. Мы знаем, что они содержатся во втором и третьем элементах списка тегов `<tr>`.

```
# переберем теги <tr> в цикле, начиная со второго
for tr_tag in tr_tags[1:]:
  # извлечем отдельные ячейки в строке (теги <td>)
  td_tags = tr_tag.find_all("td")
  # переберем их и выведем содержимое на экран
  for td_tag in td_tags:
    cell_text = td_tag.get_text(separator=" ")  # текст ячейки
    print(cell_text, end="\t")
  print()
```

```
Ул. Куйбышева, 17       2       Нет
Ул. Коминтерна, 555     5       Да
```

Итак, задача решена: нужные данные из HTML-кода извлечены.

Извлечение данных из веб-страницы: пример 2

Теперь попробуем извлечь данные из настоящей веб-страницы в интернете. Пусть это будет актуальная величина инфляции с сайта Центробанка, которая упоминалась выше (см. рис. 75).

```
# адрес веб-страницы с величиной инфляции
url = "https://www.cbr.ru/hd_base/infl"

# отправим запрос к серверу и получим HTML-код веб-страницы

# установим user agent
ug = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 " \
     "(KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36"

response = get(url, headers={"User-Agent": ug})
response  # код 200: запрос успешен
```

```
<Response [200]>
```

```
html_code = response.text  # получим код веб-страницы
# для разнообразия используем стандартный парсер
soup = bs4.BeautifulSoup(html_code)
```

Изучим код веб-страницы в браузере (рис. 85).

```
539     <br>
540 Ключевая ставка Банка России указана на последний день месяца.
541     </p>
542 <div class="table-wrapper">
543   <div class="table">
544     <table class="data">
545       <tr>
546         <th>Дата</th>
547         <th>Ключевая ставка, % годовых</th>
548         <th>Инфляция, % г/г</th>
549         <th>Цель по инфляции, %</th>
550       </tr>
551       <tr>
552         <td>10.2023</td>
553         <td>15,00</td>
554         <td>6,69</td>
555         <td>4,00</td>
556       </tr>
```

Рис. 85. Теги со значением инфляции на странице сайта Центробанка

Необходимо найти такой тег, который содержит интересующие нас данные. Как видим, есть тег `<table>` с атрибутом *class* и его значением *data*. Такой тег на странице только один, и он содержит данные об инфляции. В свою очередь, интересующая нас величина хранится в третьей ячейке второй строки таблицы. Можно предположить, что последняя имеющаяся информация об инфляции всегда будет расположена именно здесь – в первой после шапки строке и третьей ячейке (потому что в первых двух ячейках находятся дата и ключевая ставка). Конкретно третий тег `<td>` получить мы не сможем, потому что он не обладает атрибутами со специфическими значениями, по которым его можно было бы найти, однако можно получить список всех тегов `<td>` и обратиться к третьему его элементу:

```
table_data_tag = soup.find("table", {"class": "data"})  # извлечем тег-таблицу
td_tags = table_data_tag.find_all("td")  # извлечем все теги ячеек из таблицы
inflation_rate = td_tags[2].get_text()  # обратимся к содержимому 3-ей ячейки
print(f"Последняя опубликованная Центробанком величина инфляции: " \
      f"{inflation_rate}%")
```

```
Последняя опубликованная Центробанком величина инфляции: 6,69%
```

Таким образом, на простых примерах мы познакомились с весьма мощным инструментарием, позволяющим собирать данные из интернета. Полученные значения мы выводили на экран, однако никто не запрещает сохранить их в файл на диске. Во втором разделе пособия будет рассмотрена загрузка табличных данных из CSV- и Excel-файлов, и полученными знаниями можно будет воспользоваться, чтобы сформировать свой собственный готовый набор данных для анализа, сохранив собранные в интернете данные в виде такого файла.

11.4. Задания для самостоятельной работы

Решите следующие задачи:

1. Используя REST API, получите с HeadHunter данные о вакансиях в заданной сфере в заданном регионе, отберите из них только вакансии с зарплатой в заданном диапазоне и отсортируйте их список по убыванию. Сферу, регион и границы зарплаты следует ввести с клавиатуры. Результат выведите на экран в виде перечня вакансий, для каждой из которых укажите название, работодателя и данные о вакансии.

2. Выберите веб-страницу на свое усмотрение и извлеките из нее какие-нибудь данные, используя библиотеки requests и BeautifulSoup.

РАЗДЕЛ 2. БИБЛИОТЕКИ ДЛЯ АНАЛИЗА ДАННЫХ

Предисловие ко второму разделу

Чтобы осуществлять анализ данных и применять различные методы искусственного интеллекта и машинного обучения, необходимы собственно *данные*. В сфере наук о данных (Data Science) хранилище, являющееся исходным материалом для анализа, называют ***набором данных*** или ***датасетом*** (data set). В главе 11 мы рассмотрели некоторые способы формирования датасета для анализа путем сбора необходимых данных из интернет-источников. Во втором разделе пособия познакомимся с инструментами обработки готовых датасетов.

Датасет представляет собой набор записей об объектах, представляющих интерес для исследования. Как правило, каждая запись состоит из нескольких реквизитов, содержащих сведения о тех или иных характеристиках объекта, описывающих различные его свойства. Иными словами, данные в датасете имеют *табличный вид*. Таблица – не единственный способ организации данных, но весьма популярный. Очень часто данные распространяются именно в табличном формате (например, в виде уже упоминавшихся CSV-файлов или таблиц Excel).

Удобно ли работать с такими данными средствами чистого Python без дополнительных библиотек? Строго говоря, нет. CSV-файлы являются текстовыми, а считывать и записывать текстовые файлы мы научились, также освоили обработку текстовых данных (данных строкового типа). Но содержимое CSV-файлов представляет собой не обычный, а определенным образом структурированный текст, каждая строка разделена на столбцы с помощью символа-разделителя. Можем ли мы, используя изученный инструментарий, вычислить среднее по столбцу или выполнить отбор записей по определенным критериям? Эти и многие другие операции типичны для анализа данных. Да, можем, однако придется потрудиться, и вряд ли получившийся код будет очень коротким и (или) понятным. Причем чем сложнее будет процедура обработки данных (например, может понадобиться вычислить коэффициент корреляции между двумя столбцами), тем сложнее будет становиться наш код. А с таблицами Excel дела обстоят еще хуже. Книги Excel текстовыми файлами не являются, это файлы двоичные, и данные в них организованы намного сложнее, чем в файлах с разделителями (CSV). Конечно, мы можем все! Но зачем усложнять себе жизнь и использовать для решения задачи неподходящие инструменты при наличии подходящих?

Существуют специальные Python-библиотеки для обработки табличных данных и их визуализации. Второй раздел пособия посвящен библиотекам, дефакто являющимся стандартом среди инструментов анализа данных на Python и позволяющим решать множество задач обработки данных легко и эффективно.

12. Библиотека NumPy и векторизованные вычисления

Первая глава второго раздела посвящена знакомству с библиотекой, ориентированной на научные математические вычисления, – NumPy. Данная библиотека простой в изучении не является, однако пропускать ее рассмотрение нельзя: при работе с NumPy пользователь приобретает навыки работы с многомерными массивами и векторизованными вычислениями, необходимые для эффективной обработки данных.

12.1. Векторизованные операции

Один из важнейших инструментов, предоставляемых языком Python для анализа и обработки табличных данных, – библиотека NumPy.

Рассмотрим следующую задачу: необходимо *сложить два списка поэлементно*. Представим себе, что эти списки содержат баллы неких студентов по двум предметам, и нужно получить сумму баллов каждого студента (рис. 86).

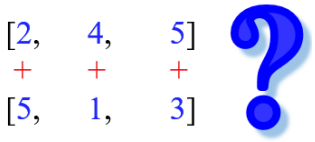


Рис. 86. Списки с баллами двух студентов по трем предметам

В чистом Python можно перебрать элементы обоих списков в цикле, вычислить сумму каждой пары элементов и добавить ее в новый список (рис. 87).

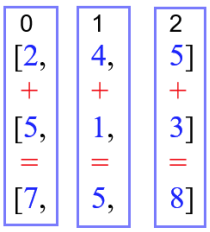


Рис. 87. Формирование списка сумм

Реализуем описанный алгоритм в коде:

```
scores1 = [2, 4, 5]  # список 1
scores2 = [5, 1, 3]  # список 2

total_scores = []  # пустой список для хранения сумм

# переберем индексы списков (целых чисел от 0 до длины списков минус 1)
for i in range(len(scores1)):
  elem_sum = scores1[i] + scores2[i] # получим сумму текущих элементов списков
  total_scores.append(elem_sum)  # добавим ее в конец нового списка

total_scores  # выведем полученный список на экран
```

```
[7, 5, 8]
```

Еще один способ – использование генерации списка и функции `zip()`:

```
total_scores = [elem1 + elem2 for elem1, elem2 in zip(scores1, scores2)]
total_scores
```
```
[7, 5, 8]
```

Как видим, результат аналогичен предыдущему. Код получился короче, но не понятнее (даже наоборот). Хорошо было бы использовать некую *более интуитивную* команду наподобие следующей: `scores1 + scores2`.

В случае обычных списков (`list`) такая операция вполне легитимна. Ее результатом будет более длинный список, состоящий из всех элементов первого списка и добавленных к ним элементов второго списка, то есть произойдет *сцепление списков (конкатенация)*:

```
scores1 + scores2
```
```
[2, 4, 5, 5, 1, 3]
```

Это не тот результат, который нас интересует. Хорошо было бы иметь поддержку в Python операций линейной алгебры, операций с векторами и матрицами. Это позволило бы выполнить поэлементное сложение двух векторов. И такая поддержка есть благодаря библиотеке NumPy.

Библиотека NumPy

NumPy – библиотека с открытым исходным кодом, которая используется для сложных математических и научных вычислений. Ее название образовано от английского словосочетания **Num**erical **Py**thon («числовой Python»). Логотип библиотеки представлен на рис. 88, документация приведена по ссылке [11]. Актуальной (на момент издания пособия) версией NumPy является 1.26.0.

Рис. 88. Логотип NumPy

Особенности библиотеки:

✓ поддержка типа данных «многомерный массив»;

✓ поддержка высокоуровневых инструментов для работы с многомерными массивами, в том числе для выполнения над ними операций линейной алгебры;

✓ библиотека написана на языке C, который является *компилируемым* (в отличие от интерпретируемого Python), и это способствует значительно более

высокой скорости исполнения кода программ, использующих функции NumPy, в сравнении с программами, написанными на чистом Python.

Специфической структурой данных (типом данных), поддержку которой добавляет NumPy, является `ndarray` (*n*-dimensional array, многомерный массив). «Многомерность» массива означает, что у него может быть несколько *измерений* (осей, axes). Для лучшего понимания воспользуемся терминологией линейной алгебры: массив с одним измерением – это *вектор*, с двумя измерениями – *матрица*, с тремя и более измерениями – *тензор* (а с нулевым количеством измерений – *скаляр*, одно число). Эти математические структуры имеют свое воплощение в виде ***NumPy-массивов*** (кроме скаляра: все-таки минимальное количество осей в объекте `ndarray` – 1).

NumPy-массивы поддерживают ***векторизованные (поэлементные) вычисления***. Векторизованные операции производятся с каждым элементом структуры данных (например, вектора или матрицы) *независимо.* В программировании такой подход позволяет при обработке всех элементов структуры данных избавиться от циклов.

Посмотрим, как NumPy-массивы помогут нам сложить два списка.

```
# подключим библиотеку numpy, в Colab она уже предустановлена
import numpy as np  # np - общепринятый псевдоним данной библиотеки

# атрибут объекта-модуля __version__ позволяет узнать номер
np.__version__  # установленной версии библиотеки
```

```
'1.23.5'
```

Как видим, на момент подготовки данного материала в Colab была установлена версия NumPy 1.23.5. Весь приведенный ниже код работает так, как показано, именно на этой версии. В будущих версиях некоторые функции, атрибуты и методы могут быть признаны *устаревшими* (deprecated), о чем интерпретатор будет сообщать при обращении к ним. Обычно в нескольких промежуточных версиях устаревшие функции еще доступны, их применение сопровождается предупреждением с рекомендацией использовать другие функции, а затем, начиная с некоторой версии, они удаляются из библиотеки окончательно.

Создать NumPy-массив можно из любой коллекции путем применения к ней функции `array()`:

```
# создадим одномерный numpy-массив из нашего списка
scores1_array = np.array(scores1)

# выведем его на экран
scores1_array  # так выглядит строковое представление numpy-массива
```

```
array([2, 4, 5])
```

Создадим второй массив и осуществим нужную нам процедуру:

```
# преобразуем в массив второй список
scores2_array = np.array(scores2)

# теперь выполним поэлементное сложение наших массивов с баллами
# результат запишем в третий массив
total_scores_array = scores1_array + scores2_array

total_scores_array  # выведем его содержимое на экран
```

```
array([7, 5, 8])
```

Как видим, массив `total_scores_array` содержит суммы элементов исходных массивов. И сформирован он был с помощью операции сложения +.

Убедимся, что вычисления с NumPy-массивами действительно осуществляются *значительно быстрее*, чем со списками. Для этого определим и сравним время выполнения операций со значениями двух этих типов данных:

```
import time  # модуль для работы со временем

# сформируем 2 очень длинных списка и 2 массива аналогичного им содержания
list1 = list(range(10 ** 7))  # список из 10 млн чисел
list2 = list(range(10 ** 7, 2 * 10 ** 7))  # еще один список

array1 = np.array(list1)  # массив 1
array2 = np.array(list2)  # массив 2
# засечем текущее время, выполним операцию
# и посмотрим на прошедшее количество секунд

# сначала сложим списки
start_time = time.time()
list3 = [elem1 + elem2 for elem1, elem2 in zip(list1, list2)]
print("Сложение списков заняло, с:", format(time.time() - start_time, ".3f"))

# теперь выполним ту же процедуру с массивами
start_time = time.time()
array3 = array1 + array2
print("Сложение массивов заняло, с:", format(time.time() - start_time, ".3f"))
```

```
Сложение списков заняло, сек: 0.835
Сложение массивов заняло, сек: 0.015
```

Как можно заметить, векторизованные операции с массивами выполняются значительно быстрее.

12.2. Операции с массивами

Рассмотрим подробнее некоторые возможности NumPy, связанные с обработкой многомерных массивов.

Векторизованные математические операции

Начнем с математических операций. При выполнении таких операций нужно учитывать, что либо структура данных операндов должна совпадать

(например, вектор можно сложить с вектором аналогичной длины), либо одним из операндов должно быть скалярное значение (число).

```
# создадим два массива для демонстрации
array1 = np.array([10, 20, 30])
array2 = np.array([1, 2, 3])
```

Поэлементное сложение и вычитание:

```
array1 + array2  # array([11, 22, 33])
array1 - array2  # array([ 9, 18, 27])
```

Поэлементное умножение и вещественное деление (можно заметить, что в NumPy-массиве нулевая дробная часть вещественного числа не отображается):

```
array1 * array2  # array([10, 40, 90])
array1 / array2  # array([10., 10., 10.])
```

Поэлементное деление без остатка и нахождение остатка от деления:

```
array1 // array2  # array([10, 10, 10])
array1 % array2  # array([0, 0, 0])
```

Поэлементное возведение в степень (первый операнд содержит основания степени, второй – показатели степени):

```
array1 ** array2  # array([10, 400, 27000])
```

Как уже было сказано, векторизованные операции можно выполнять не только над векторами и иными сложными структурами данных, но и над вектором и *скаляром*, то есть обычным числом. В этом случае скалярный операнд рассматривается как структура данных, аналогичная другому операнду, каждым элементом которой является это число (рис. 89).

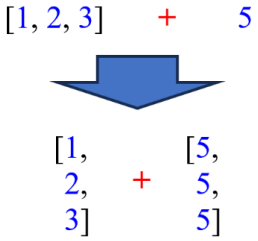


Рис. 89. Выполнение операций над вектором и скаляром

```
# добавление к каждому элементу массива числа 5
array1 + 5  # array([15, 25, 35])

# возведение каждого элемента в квадрат
array1 ** 2  # array([100, 400, 900])
```

Векторизованные операции сравнения и логические операции

Помимо математических возможны и векторизованные *операции сравнения*. Результатом таких операций является вектор булевых (логических) значений. Подобные структуры из двоичных значений еще называют ***битовыми картами*** (bit map).

```
# проверка на равенство
array1 == array2  # array([False, False, False])
# все элементы первого вектора не равны соответствующим элементам второго

# проверка на неравенство
array1 != array2  # array([True, True, True])

# проверка на неравенство
array1 != array2  # array([True, True, True])

# операция "меньше"
array1 < array2  # array([False, False, False])

# и т.д.
```

Также поддерживаются векторизованные *логические операции*, то есть операции над битовыми картами (массивами логических значений).

```
# создадим два массива логических значений
# первый получим путем манипуляций с array1
# (можно отметить, что, как и в чистом Python,
# операции сравнения имеют более низкий приоритет, чем арифметические)
bool_array1 = array1 // 5 >= np.array([5, 0, 17])

# второй логический массив зададим перечислением элементов списка
bool_array2 = np.array([False, True, True])

# посмотрим на полученные массивы (при выводе через print()
# вид numpy-массива несколько отличается от прежнего)
print("Первый массив:", bool_array1)
print("Второй массив:", bool_array2)
```

```
Первый массив: [False  True False]
Второй массив: [False  True  True]
```

Векторизованные логические операции осуществляются с помощью *следующих операторов:*

`&` – конъюнкция;

`|` – дизъюнкция;

`~` – отрицание.

```
# поэлементная конъюнкция (векторизованный аналог and)
bool_array1 & bool_array2  # array([False,  True, False])

# поэлементная дизъюнкция (векторизованный аналог or)
bool_array1 | bool_array2  # array([False,  True, True])

# поэлементное отрицание (векторизованный аналог not)
~bool_array1  # array([ True, False,  True])
```

Векторизованные математические функции

Поддерживаются в NumPy и векторизованные *математические функции*. Они называются аналогично своим скалярным вариантам из модуля math.

```
# извлечение квадратного корня
np.sqrt(array1)  # array([3.16227766, 4.47213595, 5.47722558])

# вычисление натурального логарифма
np.log(array1)  # array([2.30258509, 2.99573227, 3.40119738])

# вычисление десятичного логарифма
np.log10(array1)  # array([1., 1.30103, 1.47712125])

# вычисление двоичного логарифма
np.log2(array1)  # array([3.32192809, 4.32192809, 4.9068906])

# вычисление логарифма с произвольным основанием
np.emath.logn(
  7,  # основание логарифма (допустимо задание оснований вектором)
  array1
)  # array([1.18329466, 1.53950185, 1.7478697 ])

# потенцирование (операция, обратная логарифмированию)
# по основанию - экспоненте (e)
np.exp(array1)  # array([2.20264658e+04, 4.85165195e+08, 1.06864746e+13])

# та же операция с помощью поэлементного возведения e в степень
np.e ** array1  # array([2.20264658e+04, 4.85165195e+08, 1.06864746e+13])

# тригонометрические функции (на примере синуса)
np.sin(array1)  # array([-0.54402111,  0.91294525, -0.98803162])

# округление элементов массива array1 / 3 до 2 знаков
np.round(array1 / 3, 2)  # array([3.33, 6.67, 10.])
```

12.3. Многомерные массивы

Рассмотрим операции с *многомерными массивами* (в основном с *матрицами*; массивы с числом осей более двух сложны для восприятия, поэтому ограничимся демонстрацией нескольких несложных примеров работы с ними).

```
# создадим матрицу из списка списков
matr = np.array(
  [[1, 2, 3, 4],
   [5, 6, 7, 8],
   [9, 10, 11, 12]]
)
matr
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Векторизованные математические операции с многомерными массивами выполняются *аналогично операциям с векторами* (структуры данных должны совпадать либо другим операндом должен быть скаляр):

```
# математические операции над матрицами
matr + matr ** 2 - matr * 5  # результат - матрица тех же размеров
```

```
array([[-3, -4, -3,  0],
       [ 5, 12, 21, 32],
       [45, 60, 77, 96]])
```

Некоторые атрибуты и методы массивов

NumPy-массив, как и любой объект в Python, обладает определенными *атрибутами и методами.* Взглянем на некоторые из них.

Атрибут `.ndim` содержит число осей (измерений) массива.

```
matr.ndim  # 2 (матрица – это двумерный массив)
```

Атрибут `.shape` содержит размеры массива (его форму). Представляет собой кортеж длин массива по каждой из осей. Длина этого кортежа равна числу осей (значению атрибута `.ndim`).

```
matr.shape  # (3, 4) (3 строки, 4 столбца)
```

Атрибут `.size` содержит общее количество элементов массива, равное произведению элементов кортежа `.shape`.

```
matr.size  # 12
```

Весьма полезным и часто применяемым методом является `.reshape()`. Он позволяет *изменять форму массива* (например, может превратить прямоугольную матрицу в вектор-строку или вектор-столбец). Изменение формы массива часто требуется при построении *моделей машинного обучения* (например, функция может требовать на входе данные в виде не «плоского» вектора, а вектора-столбца, то есть матрицы из одного столбца). Аргументами `.reshape()` являются желаемые длины осей, заданные через запятую отдельными значениями (возможно задание кортежем того же вида, что значение атрибута `.shape`). Произведение этих длин должно совпадать с общим количеством элементов в массиве, а данные должны раскладываться на желаемое количество элементов по

осям, в противном случае произойдет ошибка выполнения. Например, массив 3×5 не удастся преобразовать в массив 2×6 (а в массив 5×3 удастся).

```
matr.reshape((4, 3))  # превращение матрицы 3x4 в матрицу 4x3
```
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
matr.reshape((2, 6))  # превращение матрицы 3x4 в матрицу 2x6
```
```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

```
matr.reshape((1, 12))  # превращение матрицы 3x4 в вектор-строку (матрицу
# 1x12). Обратите внимание на двойные скобки: это не одномерный массив!
```
```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
```

```
matr.reshape((12, 1)) #превращение матрицы 3x4 в вектор-столбец (матрицу 12x1)
```
```
array([[ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
...
```

Длину массива по той или иной оси можно явно не задавать, для этого на соответствующей позиции следует написать `-1`:

```
matr.reshape((6, -1))  # изменение числа строк на 6
# (столбцов – столько, сколько получится)
```
```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12]])
```

```
matr.reshape((-1,))  # превращение матрицы в плоский вектор
# (такой длины, какой получится)
```
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

```
# превращение матрицы в трехмерный массив 2x3x2
array3d = matr.reshape((2, 3, 2))
array3d
```
```
array([[[ 1,  2],
        [ 3,  4],
        [ 5,  6]],

       [[ 7,  8],
        [ 9, 10],
        [11, 12]]])
```

Метод `.cumsum()` вычисляет накопленную сумму элементов массива любой размерности, возвращает плоский вектор. Результат аналогичен результату вызова функции `itertools.accumulate()` без аргументов (см. параграф 8.3).

```
matr.cumsum() # результат - вектор (даже если исходный массив многомерный)
array([ 1,  3,  6, 10, 15, 21, 28, 36, 45, 55, 66, 78])
```

Метод `.cumprod()` вычисляет накопленное произведение элементов.

```
matr.cumprod()
array([       1,        2,        6,       24,      120,      720,
         5040,    40320,   362880,  3628800, 39916800, 479001600])
```

Метод `.tolist()` преобразует многомерный массив в иерархическую структуру из списков.

```
matr.tolist()  # матрица превращается в список списков
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

```
array3d.tolist() # трехмерный массив превращается в список списков списков
[[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10], [11, 12]]]
```

Метод `.flatten()` преобразует многомерный массив в одномерный (выполняет «сплющивание» многомерного массива).

```
matr.flatten()  # матрица превращается в плоский вектор
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Обращение к элементам массива и срезы

NumPy-массив является ***изменяемой индексированной*** коллекцией. Для обращения к конкретному элементу массива, как и в случае обычных списков, служат квадратные скобки. Однако массив *многомерный*, поэтому положение элемента может описываться не одним, а несколькими значениями, перечисленными через запятую: `[i, j, k, ..., n]`, где `i`, `j`, `k`, `n` – позиции элемента по соответствующим осям. Поддерживается и механизм *срезов* (по разным осям срезы задаются независимо).

```
array1[0]  # 1-й элемент одномерного массива: 10
array1[1:]  # элементы со 2-го по последний: array([20, 30])
matr[1, 2]  # 3-й элемент во 2-й строке матрицы: 7
matr[:, 1]  # 2-й столбец (плоский вектор): array([2, 6, 10])
matr[:, [1]]  # 2-й столбец (вектор-столбец), номер задан вектором [1]
```

```
array([[ 2],
       [ 6],
       [10]])
```

Еще несколько примеров:

```
matr[[2, 0], :-1]  # матрица из 3-й и 1-й строк и столбцов с первого
# до последнего (не включая последний)
```

```
array([[ 9, 10, 11],
       [ 1,  2,  3]])
```

```
array3d[1, :, :]  # вторая "подматрица" трехмерного массива
```

```
array([[ 7,  8],
       [ 9, 10],
       [11, 12]])
```

Функции для генерации массивов

Рассмотрим некоторые функции для генерации массивов определенного содержимого. С помощью функций `zeros()` и `ones()` можно генерировать массивы из нулей и единиц соответственно:

```
np.zeros((5, 7))  # матрица из нулей 5x7
```

```
array([[0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.]])
```

```
np.ones((3, 2, 4))  # трехмерный массив из единиц 3x2x4
```

```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

Функция `eye()` предназначена для генерации единичной матрицы размером $n \times m$ (не обязательно квадратной).

```
np.eye(3, 4)  # матрица 3x4 с единицами по главной диагонали
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

Некоторые другие функции

Рассмотрим еще несколько функций библиотеки NumPy.

Функция `arange()` является аналогом `range()` с поддержкой вещественных чисел. Предназначена для генерации монотонных последовательностей. Возвращаемое значение – массив.

```
np.arange(0, 5.01, 2.5)  # массив элементов от 0 до 5 с шагом 2.5
```
```
array([0. , 2.5, 5. ])
```

Функция `linspace()` генерирует последовательность от a до b (*включая* b) заданной длины.

```
np.linspace(0, 1, 5)  # левая граница 0, правая 1, всего 5 элементов
```
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Функция `diff()` вычисляет разности элементов массива заданного порядка. По умолчанию вычисляются первые разности.

```
# первые разности: 5-3=2, 9-5=4, -3-9=-12
np.diff([3, 5, 9, -3])  # array([  2,   4, -12])

# вторые разности: 4-2=2, -12-4=-16
np.diff([3, 5, 9, -3], n=2)  # array([  2, -16])
```

Функции `vstack()` и `hstack()` соединяют массивы по вертикали и горизонтали соответственно. Аргументом обеих функций является коллекция соединяемых массивов (кортеж или список).

```
np.vstack(
  (matr[:2, :],  # матрица из первых 2 строк исходной матрицы matr
   matr[2:, :])  # матрица из 3-й строки (вектор-строка) матрицы matr
)  # результат соответствует исходной матрице
```
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```
np.hstack(
  (matr[:, :2],  # матрица из первых 2 столбцов исходной матрицы matr
   matr[:, 2:])  # матрица из вторых 2 столбцов матрицы matr
)  # результат соответствует исходной матрице
```
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

С помощью функции `hstack()` можно выполнить сцепление (конкатенацию) одномерных массивов (ведь обычное сложение + вместо этого формирует массив сумм).

```
# сцепляем 3 массива
np.hstack(
  (array1,
   array2,
   array1 * 3)
)
```
```
array([10, 20, 30,  1,  2,  3, 30, 60, 90])
```

Функции `vsplit()` и `hsplit()` разделяют массивы на n частей по первому измерению по вертикали и горизонтали соответственно. Результатом обеих функций является *список* подмассивов. Длина первого измерения разделяемого массива должна быть кратна n.

```
np.vsplit(matr, 3)#список из 3 "подматриц" исходной матрицы (векторов-строк)
```
```
[array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]]), array([[ 9, 10, 11, 12]])]
```

```
np.hsplit(matr, 2)  # список из 2 "подматриц" исходной матрицы
```
```
[array([[ 1,  2],
        [ 5,  6],
        [ 9, 10]]),
 array([[ 3,  4],
        [ 7,  8],
        [11, 12]])]
```

Функция `roots()` находит корни полинома. Корни могут быть комплексными. Аргументом функции является коллекция коэффициентов полинома. Определим корни квадратного уравнения $2x^2 + 4x - 2 = 0$:

```
np.roots([2, 4, -2])  # свободный член задается последним
```
```
array([-2.41421356,  0.41421356])
```

12.4. Использование генератора случайных чисел

В NumPy поддерживается работа с генератором случайных чисел. Например, можно получить случайную выборку значений из того или иного стандартного распределения в виде массива.

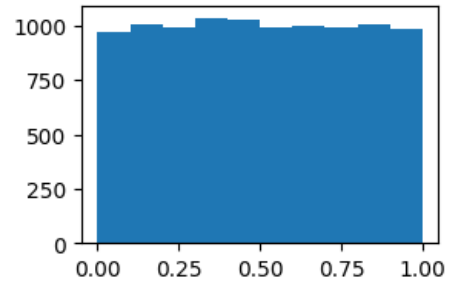
Функция `random.uniform()` генерирует указанное количество значений в заданном диапазоне из *равномерного* распределения.

```
# генерация выборки размером 10000 из равномерного распределения
# в диапазоне от 0 до 1
uniform_array = np.random.uniform(0, 1, 10000)
uniform_array
```
```
array([0.59022788, 0.33550111, 0.08052889, ..., 0.3538479 , 0.33873443,
       0.47746998])
```

Убедимся, что выборка действительно получена из равномерного распределения. Для этого воспользуемся методом визуального анализа и построим гистограмму частот с помощью библиотеки Matplotlib (о ней подробно поговорим в главе 14). Для построения гистограммы используем функцию `hist()`. Заметим, что мы сгенерировали выборку столь большого размера (10 000) для наглядности: в силу *закона больших чисел* свойства выборки (включая и форму ее гистограммы) оказываются тем ближе к свойствам генеральной совокупности, чем

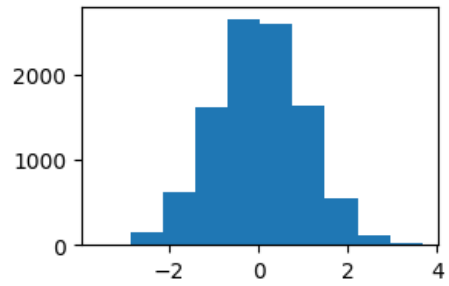
больше ее объем. Гистограмма выборки малого объема не очень похожа на кривую плотности вероятности равномерного распределения:

```
import matplotlib.pyplot as plt  # подключим библиотеку
plt.hist(uniform_array);  # построим гистограмму частот
```



Функция `random.normal()` генерирует выборку из *нормального* распределения с заданными параметрами (по умолчанию математическое ожидание равно 0, дисперсия равна 1). Сведения о функциях для работы с другими распределениями можно найти в документации по ссылке [15].

```
normal_array = np.random.normal(0, 1, 10000)  # сгенерируем выборку
plt.hist(normal_array);  # построим гистограмму частот
```



Функция `random.shuffle()` выполняет перемешивание элементов массива. Она изменяет входной массив и ничего не возвращает. Перемешивание позволяет обеспечить независимость объектов выборки друг от друга, что бывает полезно при построении моделей машинного обучения.

```
shuffled_array = array1.copy()  # создадим копию массива
np.random.shuffle(shuffled_array)  # перемешаем элементы этой копии
shuffled_array
```

```
array([20, 30, 10])
```

Функция `random.choice()` извлекает из массива случайную подвыборку заданного объема (повторную или бесповторную):

```
# случайная подвыборка из 5 элементов последовательности от 1 до 100
# replace=True - повторная, False - бесповторная
np.random.choice(np.arange(1, 101), 5, replace=False)
```

```
array([64, 91, 93, 58, 29])
```

Функция `random.seed()` фиксирует состояние генератора случайных чисел. Это обеспечивает *воспроизводимость результатов генерации*. Аргументом функции является произвольное целочисленное значение. При одном и том же заданном значении все функции библиотеки NumPy, использующие генератор случайных чисел, *всегда выдают один и тот же результат*. Заданное состояние генератора сохраняется в пределах одной ячейки Colab, на другие ячейки действие `seed()` не распространяется.

Следует пояснить, как работает генерация случайных чисел в программировании и почему полученные с ее помощью числа правильнее называть ***псевдослучайными***. Дело в том, что компьютер *не может сгенерировать полностью случайное число*. Эта задача сродни получению результата работы алгоритма *без входных данных* (то есть получению информации из ничего, «из пустоты»). Псевдослучайные числа являются результатом работы некоторого алгоритма, а следовательно, они формируются путем преобразования определенной входной информации. Такой информацией, как правило, является *значение текущего времени*. Компьютерные часы непрерывно идут, время постоянно меняется, счетчик мельчайших долей секунды увеличивается. Следовательно, при разных запусках генератора числа на выходе тоже всегда оказываются разными. Функция же `seed()` (с английского «сеять») подает на вход генератора конкретное заданное значение, что и приводит к соответствующему этому входному значению результату. В каком-то смысле происходит практическая реализация пословицы «что посеешь, то и пожнешь».

```
np.random.seed(123)  # зафиксируем состояние генератора числом 123
print(np.random.uniform(0, 1, 1))  # сгенерируем одно случайное число
print(np.random.normal(0, 1, 2))  # сгенерируем еще два числа
# при каждом запуске этого кода числа будут одни и те же
```

```
[0.69646919]
[-0.95209721 -0.74544106]
```

12.5. Операции линейной алгебры над матрицами

Наконец, рассмотрим функции для выполнения некоторых операций линейной алгебры. Функция `linalg.det()` позволит вычислить определитель квадратной матрицы:

```
np.linalg.det(matr[1:, :-2])  # матрица должна быть квадратной
```

```
-4.000000000000003
```

Функция `linalg.eig()` вычисляет собственные значения и собственные векторы квадратной матрицы. Возвращает кортеж из двух элементов: 1) массив собственных значений длины n; 2) матрица из n столбцов, i-й столбец которой соответствует i-му элементу массива.

```
np.linalg.eig(matr[1:, :-2])  # матрица должна быть квадратной
(array([-0.26208735, 15.26208735]),
 array([[-0.75182561, -0.50473582],
        [ 0.659362  , -0.86327385]]))
```

Функция `linalg.inv()` вычисляет обратную матрицу:

```
np.linalg.inv(matr[1:, :-2])  # матрица должна быть квадратной
array([[-2.5 ,  1.5 ],
       [ 2.25, -1.25]])
```

Метод `.transpose()` формирует транспонированную матрицу. К аналогичному результату приводит обращение к атрибуту массива `.T`. Транспонирование можно использовать для превращения вектора-столбца в вектор-строку и наоборот.

```
matr.transpose()  # matr.T - то же самое
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
```

Функция `dot()` и одноименный метод массива выполняют произведение матриц:

```
np.dot(matr, matr.T)  # matr.dot(matr.T) - то же самое
array([[ 30,  70, 110],
       [ 70, 174, 278],
       [110, 278, 446]])
```

Пример использования NumPy

Вооружившись изученным инструментарием, решим небольшую задачу для закрепления материала по библиотеке NumPy. Пусть дана линейная зависимость $y = 5 + 3x + \varepsilon$, где x – величина, распределенная по закону Стьюдента с числом степеней свободы 5; ε – величина, распределенная по стандартному нормальному закону (математическое ожидание 0, дисперсия 1). Необходимо:

– сгенерировать выборки величин x и ε по 1000 значений в каждой в соответствии с законами их распределения;

– для каждого значения x вычислить y;

– найти оценки коэффициентов зависимости y от x с помощью *метода наименьших квадратов* и убедиться, что метод применен правильно (оценки близки к истинным значениям коэффициентов).

Формула метода наименьших квадратов в матричном виде: $\hat{\beta} = (X^T.X)^{-1}.X^T.Y$, где $\hat{\beta}$ – вектор оценок коэффициентов; X – матрица из столбцов

– значений независимых переменных; Y – вектор значений зависимой переменной. Поскольку зависимость имеет свободный член, матрица X должна включать в себя столбец из единиц.

Напишем код для решения этой задачи. Сначала сгенерируем массив значений x, для чего изучим документацию библиотеки NumPy и выясним, что выборку из распределения Стьюдента можно получить с помощью функции `standard_t()`.

```
x_array = np.random.standard_t(df=5, size=1000)  # массив значений x
```

Затем сгенерируем массив значений ε.

```
eps_array = np.random.normal(size=1000)  # массив значений e
```

Используя векторизованные вычисления, получим массив значений y по приведенной в задании формуле:

```
Y = 3 * x_array + 5 + eps_array  # массив значений y
```

Для применения метода наименьших квадратов необходимо, чтобы массив значений x представлял собой не плоский вектор, а матрицу, столбцами которой являются значения независимых переменных (в нашем случае такая переменная одна – x), а также столбец единиц (он нужен, если аппроксимируемая зависимость включает свободный член). Сформируем матрицу X путем горизонтального соединения двух векторов: вектора-столбца единиц и вектора-столбца значений x, который нужно предварительно получить из ранее сгенерированного плоского вектора.

```
# матрица, полученная путем горизонтального соединения двух векторов:
X = np.hstack([
  np.ones((1000, 1)),  # вектора-столбца из единиц
  x_array.reshape((-1, 1))  # и вектора-столбца значений x
])
```

Наконец, применим метод наименьших квадратов в соответствии с приведенной формулой. Если все сделано правильно, результатом вычислений будет вектор оценок коэффициентов зависимости.

```
# вычислим оценки коэффициентов по формуле
coefs = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Y)
coefs  # выведем массивы оценок на экран
```

```
array([5.00034798, 2.97376521])
```

Как можно заметить, полученные оценки очень похожи на фактические коэффициенты зависимости (5 и 3), а значит, метод наименьших квадратов применен нами правильно.

На этом наше знакомство с библиотекой NumPy завершается. Как мы убедились, важнейшим ее достоинством является предоставление инструментария для выполнения операций не только над скалярными числами, но и над векторами и матрицами (над всеми их элементами разом), что очень удобно при обработке и анализе табличных данных. Однако NumPy предлагает мыслить в терминах *линейной алгебры:* векторов, матриц, многомерных массивов. В анализе данных обычно работают не с матрицами, а с *таблицами*. Кроме того, датасеты далеко не всегда содержат *только числовые* данные (а в названии NumPy даже присутствует слово *numerical*). В следующей главе мы изучим возможности библиотеки pandas, ориентированной непосредственно на анализ и обработку табличных датасетов.

12.6. Задание для самостоятельной работы

Решите следующую задачу. Напишите программу для нахождения корней полинома степени n с помощью библиотеки NumPy.

Вид полинома: $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$.

Необходимые функции, о которых не говорилось в главе 12, найдите в документации самостоятельно. Степень полинома n следует ввести с клавиатуры ($n \geq 10$). Коэффициенты должны быть сгенерированы случайно. Необходимо подобрать такие коэффициенты, чтобы у полинома было не менее пяти вещественных (не комплексных) корней.

13. Библиотека pandas

Данная глава посвящена знакомству с библиотекой pandas – инструментом, которым должен владеть каждый уважающий себя специалист по анализу данных в Python. Рассматриваются основы работы с объектами `Series` и `DataFrame`.

13.1. Ряды (тип данных Series)

Своего рода индустриальным стандартом в анализе табличных данных на языке Python является использование библиотеки ***pandas.*** Ее название образовано от словосочетания **pan**el **da**ta («панельные данные»). Логотип pandas представлен на рис. 90, документация приведена по ссылке [12]. Актуальной (на момент издания пособия) версией pandas является 2.1.4.

Рис. 90. Логотип pandas

Pandas является надстройкой («оберткой», «оболочкой») над изученной нами в прошлой главе библиотекой NumPy**.** Как и положено надстройке, pandas предоставляет более простые и удобные инструменты для обработки данных. Можно сказать, что эта библиотека *более высокоуровневая*, порог вхождения в работу с ней ниже. Если основной категорией NumPy является *многомерный массив,* то в случае pandas это более привычная и понятная *таблица,* также pandas не требует знаний линейной алгебры при использовании своих основных возможностей.

Основные структуры данных pandas:

- ✓ ***Series*** – это одномерный массив (ряд, последовательность, серия).
- ✓ ***DataFrame*** – это двумерный массив (таблица).

В данном параграфе изучим работу с рядами. Структура `Series` является надстройкой над многомерным массивом `ndarray` с *одним* измерением. Она является **изменяемой индексированной** коллекцией и похожа как на список (`list`), так и на словарь (`dict`). К отдельному элементу ряда можно обратиться двумя способами: по его *порядковому номеру* (нумерация начинается с нуля – как в случае списка) и по его *индексу* (метке или ключу – как в случае словаря). Ряд можно представить как своеобразную таблицу размером $1 \times n$ (одна строка, n столбцов) с двойной шапкой (рис. 91).

Позиции (как в списке)	0	1	…	$n-1$
Индексы (как в словаре)	Индекс 1	Индекс 2	…	Индекс n
Значения (хранимые данные)	Значение 1	Значение 2	…	Значение n

Рис. 91. Наглядное представление Series в виде таблицы

Можно вспомнить, что словарь для независимого обращения к ключам и значениям содержал методы `.keys()` и `.values()` соответственно. Объект `Series` же для этой цели имеет два атрибута: `.index` – коллекция *индексов* (меток, ключей) и `.values` – одномерный NumPy-массив *значений*. Рассмотрим работу с рядами на примерах:

```
import pandas as pd  # подключим библиотеку pandas
# псевдоним pd является общепринятым

pd.__version__   # установленная в Colab версия библиотеки
```
```
'1.5.3'
```

На момент подготовки примеров кода в Colab была установлена версия pandas 1.5.3, все полученные результаты справедливы для нее.

Создание нового ряда

Ряд можно создать из другой коллекции с помощью функции `Series()`. Элементы коллекции будут использованы как значения ряда. Индексы можно задать с помощью аргумента `index`. Если перечислены только значения, а индексы явно не заданы, в качестве них используются целые числа от 0 до $n-1$, где n – количество значений.

```
ser1 = pd.Series([10, 20, 30])  # создадим объект Series из списка
ser1  # строковое представление ряда выглядит так
```
```
0    10
1    20
2    30
dtype: int64
```

Изучим представление ряда при выводе на экран. *Левый* столбец содержит *индексы*, *правый* – соответствующие им *значения*. Также выводится тип хранимых данных (`dtype`) – это тип элементов ряда[8]. При выводе на экран NumPy-массива тип его значений не указывается (предполагается, что массивы используются для хранения чисел). Pandas же активно применяется в анализе данных,

[8] `Series` является надстройкой над `ndarray`, внутри ряд – это многомерный массив (в объектно-ориентированном программировании создание «надстроек» над типами данных называется *наследованием*). Атрибут `.dtype`, хранящий тип значений, есть и у NumPy-массивов, а это значит, что массивы могут хранить не только числа, но и данные других типов, просто ранее мы об этом не упоминали.

и эти данные далеко не всегда *количественные*, они могут быть и *качественными*, например строковыми. В нашем примере ряд состоит из трех целочисленных значений, поэтому тип ряда числовой. Однако заметим, что в NumPy и pandas используются не стандартные Python-типы `int` и `float`, а их реализации `int64` и `float64`, поддержка которых обеспечивается библиотекой NumPy. Если в ряде не все значения имеют числовой тип, тогда атрибут `.dtype` имеет значение `object` (это наиболее общий тип данных в Python). В тонкости наследования типов данных (классов) погружаться не станем, но понимать, что именно мы наблюдаем на экране, необходимо.

Для хранения индексов ряда используются специальные типы данных со словом *Index* в названии. В случае, когда индексы заданы монотонной числовой последовательностью, используется тип `RangeIndex`. Значения же хранятся в обычном NumPy-массиве.

```
ser1.index  # атрибут index - индексы ряда (ключи, метки)
```
```
RangeIndex(start=0, stop=3, step=1)
```

```
ser1.values  # атрибут values - значения ряда
```
```
array([10, 20, 30])
```

Будем считать наши данные значениями курсов некоторых валют. Обозначения этих валют используем как индексы ряда.

```
ser1.index = ["EUR", "USD", "JPY"]  # зададим индексы вручную, записав
# в атрибут index список значений (той же длины, что сам ряд)
ser1
# также индексы можно было задать явно при создании ряда:
# ser1 = pd.Series([10, 20, 30], index=["EUR", "USD", "JPY"])
```
```
EUR    10
USD    20
JPY    30
dtype: int64
```

Еще одним способом создания ряда является преобразование его из словаря. В этом случае ключи словаря интерпретируются как *индексы* ряда, значения – как собственно *значения*.

```
pd.Series({"EUR": 10, "USD": 20, "JPY": 30})
```
```
EUR    10
USD    20
JPY    30
dtype: int64
```

Ряд является коллекцией и поддерживает соответствующие инструменты. Например, длину ряда (количество элементов в нем) можно получить с помощью функции `len()`:

```
len(ser1)  # 3
```

Обращение к элементам ряда и срезы

Как было отмечено выше, к отдельным элементам `Series` можно обращаться как по порядковому номеру, так и по индексу.

```
ser1[1]  # обращение ко 2-му элементу по номеру: 20
ser1["USD"]  # обращение к нему же по индексу: 20
```

При этом используется один и тот же синтаксис – квадратные скобки `[x]`. Интерпретатор сам разберется, что такое `x`: позиция или индекс элемента. Однако возможны коллизии: индексы могут совпадать с порядковыми номерами частично или целиком. Например, при создании ряда без явного задания индексов они принимают значения от 0 до длины ряда минус 1, что полностью соответствует позициям. Ввиду этого *в приоритете* обращение по ***индексу***.

```
pd.Series(["A", "B"], index=[1, 0])[1]  # элемент с индексом 1 - первый: A
```

Также `x` интерпретируется как индекс *всегда*, если: 1) `x` не является целым числом; 2) среди индексов встречаются целые числа. Если элемент с индексом `x` отсутствует, возникает ошибка KeyError:

```
# элемент на позиции 2 (третий) есть, но среди индексов есть целое число
pd.Series(["A", "B", "C"], index=[1, "x", "y"])[2]  # ошибка KeyError
```

Если описанные выше условия не выполняются (`x` – целое число и среди индексов целые числа отсутствуют), производится попытка обращения по номеру. Если элемента на указанной позиции нет, возникает ошибка IndexError:

```
ser1[5]  # шестого элемента нет, их всего три: ошибка IndexError
```

Описанный механизм выглядит несколько запутанно, не правда ли? Чтобы избежать неоднозначности, следует использовать атрибуты `.loc` (обращение *строго по индексу*) и `.iloc` (обращение *строго по номеру*).

```
ser1.loc["USD"]  # обращение ко 2-му элементу по индексу: 20
ser1.iloc[1]  # а обращение к нему же по номеру: 20
```

В отличие от списка и словаря, из ряда можно извлекать сразу несколько элементов, перечисляя их индексы или позиции в списке. С NumPy-массивами этот механизм тоже работает (и он встречался в примерах кода), но мы не акцентировали на этой возможности внимания, оставив ее упоминание для текущей главы. Результатом такого отбора будет `Series` (или `ndarray` в случае массива).

```
ser1.loc[["USD", "EUR"]]  # ряд из элементов с индексами USD, EUR
ser1.iloc[[1, 0]]  # результат аналогичен
```

Работают с рядами и *срезы*, причем как по номерам, так и по *индексам*. Срезы по индексам имеют *особенность*: в отличие от обычных срезов по номерам (как они функционируют со списками или NumPy-массивами), в них правая граница *включается*.

```
ser1[:2]  # ряд из первых двух элементов (по номерам)
ser1[:"USD"]  # ряд из элементов до USD включительно (по индексам)
ser1.iloc[1:]  # ряд из элементов со второго (строго по номерам)
ser1.loc["USD":"JPY"]  # ряд из элементов с USD по JPY (строго по индексам)
```

В отличие от ключей словаря, индексы ряда могут быть *неуникальны* (элемент все равно однозначно идентифицируется своим порядковым номером). При наличии элементов с одинаковыми индексами обращение по индексу возвращает не единственное значение, а *ряд*:

```
pd.Series(["A", "B", "C"], index=[0, 0, 1])[0]  # элементов с индексом 0 два
```
```
0    A
0    B
dtype: object
```

Добавление, модификация и удаление элементов ряда

Для добавления нового элемента ряда и перезаписи существующего используется *оператор присваивания* (как в словарях и отчасти в списках):

```
<серия>[<индекс>] = <значение>
```

Если элемента с указанным индексом нет, он создастся и добавится *в конец ряда*. Если такой элемент существует, его значение перезапишется:

```
ser1.loc["USD"] = 60  # присвоим новое значение элементу с индексом USD
ser1  # как видим, значение 20 сменилось на 60
```
```
EUR    10
USD    60
JPY    30
dtype: int64
```

У рядов *отсутствует* метод наподобие спискового `.insert()`, позволяющий вставить новый элемент на заданную позицию (например, посреди ряда). Добиться указанного результата можно различными способами, один из которых использован в примере ниже.

```
# необходимо вставить новый элемент на вторую позицию
ser1["CNY"] = 3  # добавим новый элемент в конец ряда
ser1 = ser1.iloc[[0, 3, 1, 2]]  # изменим порядок элементов, как требуется
ser1
```

```
EUR    10
CNY     3
USD    60
JPY    30
dtype: int64
```

Ряды (как и NumPy-массивы) поддерживают механизм *группового присваивания*. Это не что иное, как *распаковка коллекции* (которую мы рассматривали в параграфе 5.5 на примере кортежей), находящейся в правой части команды, в соответствующие элементы вектора – результата отбора или среза в левой части[9]. Также возможно присваивание *скалярных значений*, в этом случае во все элементы вектора в левой части запишется одно и то же значение (с обычными списками это не сработает из-за отсутствия поддержки ими векторизованных операций).

```
ser1.iloc[[0, -1]] = 0  # извлечем ряд из двух элементов (первого и
# последнего) перечислением номеров и зададим их значения одним числом
ser1
```

```
EUR     0
CNY     3
USD    60
JPY     0
dtype: int64
```

```
ser1.loc[["EUR", "JPY"]] = 70, 5  # извлечем ряд из тех же элементов
# перечислением индексов и зададим значения этих элементов кортежем
ser1
```

```
EUR    70
CNY     3
USD    60
JPY     5
dtype: int64
```

Для удаления элементов ряда служат следующие методы.

Метод `.drop()` удаляет элемент по индексу (возможно указание нескольких индексов в виде коллекции). Результат работы метода зависит от значения

[9] Со списками это тоже работает. Например, команда `new_list[:2] = (0, 0)` заменит первые два элемента списка `new_list` нулями.

его аргумента `inplace`: если оно не задано или равно `False` (значение по умолчанию), `.drop()` конструирует и возвращает новый объект – ряд с удаленными элементами, не внося изменений в исходный ряд, в противном случае метод модифицирует свой ряд и ничего не возвращает. Надо отметить, что аргумент `inplace` присутствует у многих методов `Series` и `DataFrame`. В дальнейшем мы будем использовать его в примерах, но не станем акцентировать внимание на принципе его действия.

```
ser1.drop(["CNY"], inplace=True)  # удалим элемент CNY
ser1["GEL"] = 20  # вместо него добавим GEL
ser1
```

```
EUR    70
USD    60
JPY     5
GEL    20
dtype: int64
```

Метод `.pop()` удаляет из ряда элемент с указанным индексом и возвращает его значение (действует аналогично одноименному методу словаря).

```
ser1["BYN"] = 50  # добавим новый элемент
ser1.pop("BYN")  # и сразу его удалим
```

```
50
```

Сцепление (конкатенация) рядов

Как и в случае с NumPy-массивами, сцепление рядов производится не оператором + (как сцепление списков), а специальной функцией: `concat()`. Аргументом ее является список сцепляемых рядов.

```
# сцепим два ряда
pd.concat([
  ser1,  # первый ряд
  pd.Series([1, 2])  # второй ряд
])
```

```
EUR    70
USD    60
JPY     5
GEL    20
0       1
1       2
dtype: int64
```

Отметим, что в pandas версии 1.5.3 объект `Series` имеет метод `.append()`, который служит для той же цели – сцепления нескольких рядов. Однако этот метод признан устаревшим (deprecated) и в pandas версии 2 удален. Его использование в версии 1.5.3 сопровождается выводом следующего предупреждающего сообщения:

```
FutureWarning: The series.append method is deprecated and will be removed from
pandas in a future version. Use pandas.concat instead.
```

Ввиду этого для сцепления рядов следует применять функцию `concat()`.

Отбор элементов ряда

Как и NumPy-массивами, рядами поддерживается *механизм отбора элементов по битовым картам*, то есть векторам логических значений (рис. 92).

```
ser1[[True, False, False, True]]  # битовая карта – список из логических
# значений; показывает, какие значения включить в отбор, а какие нет
```

```
EUR    70
GEL    20
dtype: int64
```

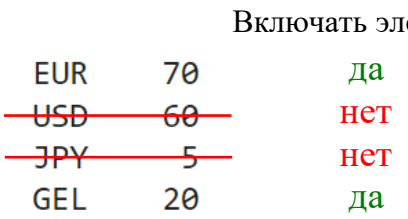


Рис. 92. Механизм отбора по битовым картам

Битовые карты легко формировать с помощью векторизованных операций сравнения и логических операций. Это открывает широкие возможности для *отбора элементов ряда по различным критериям*.

```
ser1 > 20  # битовая карта в виде Series: элементы со значением больше 20
# (в сравнении участвуют именно значения, а не индексы)
```

```
EUR     True
USD     True
JPY    False
GEL    False
dtype: bool
```

```
ser1[ser1 > 20]  # ряд только из элементов больше 20
```

```
EUR    70
USD    60
dtype: int64
```

В параграфе 10.2 упоминалось, что при обходе словарей как коллекций возвращаемыми элементами являются *ключи*, а не значения. В случае рядов ситуация обратная: хранимыми данными в Series являются именно *значения*, а не индексы. Индексы – это лишь *метаданные*; значения первичны, индексы вторичны. Ввиду этого при отборе элементов с помощью векторизованных операций сравнения этот отбор осуществляется по значениям. Однако возможен отбор и по индексам.

```
ser1[ser1.index == "EUR"]  # отбор элементов с индексом EUR
EUR    70
dtype: int64
```

Здесь следует пояснить два момента. Во-первых, векторизованные операции сравнения конструируют битовую карту, то есть *коллекцию*, вектор. Отбор элементов через вектор *всегда приводит к результату векторного типа*, даже если этот вектор состоит из одного элемента (как в примере выше: в отбор попал только один элемент с индексом EUR, однако результатом оказался объект `Series`, а не целое число)[10].

Во-вторых, неважно, каким именно способом была сконструирована битовая карта для отбора элементов ряда. Ее можно задать вручную перечислением логических значений либо сформировать путем сравнения между собой некоторых третьих векторов, не связанных с нашим исходным вектором напрямую. Главное, чтобы длина битовой карты соответствовала длине исходного вектора и для каждого элемента вектора нашлось соответствующее логическое значение, указывающее, нужно ли включать его в отбор. Конечно, необходимо также, чтобы между элементами этого вектора и битовой карты было смысловое соответствие, иначе результат отбора окажется произвольным и бессмысленным. Например, вряд ли стоит отбирать студентов группы A по критерию «студент группы B на той же позиции в списке набрал свыше 70 баллов на экзамене». Да, студентов в группах A и B может быть одинаковое количество, но как они связаны между собой, в особенности когда студенты группы A упомянутый экзамен даже не сдавали? Однако если они все же связаны (гипотетическая ситуация: группа B состоит из братьев или сестер студентов группы A), тогда подобный отбор может обрести смысл: в него попадут те студенты группы A, чьи родственники набрали свыше 70 баллов (при условии соответствия порядка студентов в списках).

```
ser1[ser1.index > "J"]  # отберем элементы с индексами больше J
# JPY попадет, потому что JPY > J (описание механизма см. в параграфе 8.1)
USD    60
JPY     5
dtype: int64
```

Векторизованные логические операции (см. параграф 12.2) позволяют конструировать *сложные условия отбора*. При этом операции сравнения нужно брать в скобки из-за их более низкого приоритета относительно операций `& | ~`.

[10] Подобный принцип справедлив и для срезов: запись `x[0]` вернет первый элемент списка, а `x[:1]` – *список*, содержащий только первый элемент (оцените разницу).

Примеры сложных отборов:

```
# конъюнкция (логическое И)
ser1[(ser1 > 10) & (ser1.index < "M")]  # отберем элементы
# со значением больше 10 И с индексом меньше M (таких элементов два)
```

```
EUR    70
GEL    20
dtype: int64
```

```
# дизъюнкция (логическое ИЛИ)
ser1[(ser1 > 10) | (ser1.index < "M")]  # отберем элементы со значением больше
10 ИЛИ с индексом меньше M (одному из условий удовлетворяют все элементы ряда)
```

```
EUR    70
USD    60
JPY     5
GEL    20
dtype: int64
```

```
# отрицание (логическое НЕ)
ser1[~(ser1.index > "J")]  # отберем элементы с индексом НЕ больше J
```

```
EUR    70
GEL    20
dtype: int64
```

Некоторые операции с рядами

Аналогично NumPy-массивам ряды поддерживают *векторизованные математические операции*, в том числе с константами в качестве операнда.

```
ser1 + ser1 * 2  # сложение исходного ряда и ряда удвоенных значений
```

```
EUR    210
USD    180
JPY     15
GEL     60
dtype: int64
```

Метод `.sort_values()` упорядочивает ряд по *значениям*. Порядок сортировки задается аргументом `ascending` (по умолчанию равен `True` – сортировка по возрастанию). Метод `.sort_index()` упорядочивает ряд по *индексам*.

```
ser1.sort_values()  # упорядочим ряд по возрастанию значений
```

```
JPY     5
GEL    20
USD    60
EUR    70
dtype: int64
```

```
ser1.sort_index(ascending=False)  # упорядочим ряд по индексам по убыванию
```

```
USD    60
JPY     5
GEL    20
EUR    70
dtype: int64
```

Метод `.apply()` применяет функцию ко всем элементам ряда. Конечно, механизм генерации списка (а также множества и словаря), позволяющий выполнить операцию со всеми элементами коллекции, с рядами тоже работает. Однако не забудем, что векторизованные вычисления с рядами (внутри которых NumPy-массивы) выполняются быстрее, чем операции со списками. Кроме того, результатом вызова метода `.apply()` является ряд с тем же набором индексов, что исходный, но с преобразованными значениями, и это может быть удобно. Продемонстрируем работу метода на примере (код ниже, рис. 93).

```
# преобразуем элементы ряда в значения вещественного типа, применив к ним
ser1.apply(float)  # функцию float (она не вызывается, скобки не нужны)
```

```
EUR    70.0
USD    60.0
JPY     5.0
GEL    20.0
dtype: float64
```

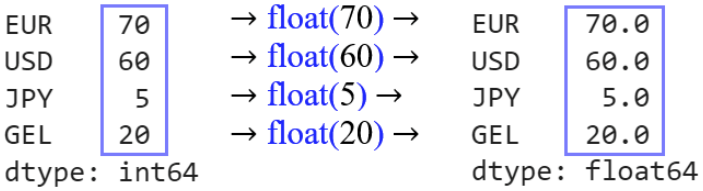


Рис. 93. Принцип действия метода Series apply()

```
ser1.apply(lambda x: -x)  # умножим элементы ряда на -1
```

```
EUR   -70
USD   -60
JPY    -5
GEL   -20
dtype: int64
```

Метод `.astype()` преобразует тип данных ряда.

```
ser1.astype(bool)  # преобразуем тип из целочисленного в логический
# (правила булификации см. в параграфе 3.4)
```

```
EUR    True
USD    True
JPY    True
GEL    True
dtype: bool
```

Иногда полезно преобразовать значения в строки. Это позволит использовать векторизованные *строковые методы*, которые вызываются через специальный атрибут `.str`. Рассмотрим некоторые из них.

Метод `.str.contains()` проверяет, содержит ли каждое из значений ряда заданную подстроку. Возвращает битовую карту. Особенностью этого метода

является поддержка в строке – шаблоне поиска синтаксиса *регулярных выражений* (данный механизм позволяет гибко обрабатывать тексты, его мы рассматривать не будем).

```
# отбор элементов с цифрой 5 в значении
ser1[ser1.astype(str).str.contains("5")]
```

```
JPY    5
dtype: int64
```

```
# отбор элементов с цифрой 5 или 6 в значении
ser1[ser1.astype(str).str.contains("5|6")]  # | - это дизъюнкция
```

```
USD    60
JPY     5
dtype: int64
```

Методы `.str.replace()`, `.str.split()` выполняют те же действия, что и обычные одноименные строковые методы. О других строковых методах `Series` (а также специальных атрибутах помимо `.str`) можно почитать в документации.

```
# замена подстроки "0" на "00"
ser1.astype(str).str.replace("0", "00")
```

```
EUR    700
USD    600
JPY      5
GEL    200
dtype: object
```

```
# преобразование строки в список подстрок по разделителю (значениями ряда
ser1.apply(float).astype(str).str.split(".")  # могут быть списки!)
```

```
EUR    [70, 0]
USD    [60, 0]
JPY     [5, 0]
GEL    [20, 0]
dtype: object
```

Метод `.value_counts()` конструирует *частотную таблицу* значений ряда. Он возвращает ряд, индексами которого являются значения исходного ряда, а значениями – их частоты (***частота*** – величина, показывающая, сколько раз то или иное значение встречается в ряде). Аргумент `sort` упорядочивает значения по убыванию частот (по умолчанию равен `True`). Аргумент `normalize` вместо *абсолютных* частот вычисляет *относительные* (доли, занимаемые значениями в ряде). Аргумент `bins` позволяет указать количество интервалов, на которые разбивается диапазон значений ряда, его использование конструирует *интервальный ряд*. Задание значения `False` аргумента `dropna` (по умолчанию – `True`) включает подсчет и вывод частоты пропусков (`NaN`).

```
# все значения встречаются по одном разу, пропуски отсутствуют
ser1.value_counts(dropna=False)
```

```
70    1
60    1
5     1
20    1
dtype: int64
```

Для лучшей демонстрации работы метода `.value_counts()` сгенерируем другой ряд:

```
import numpy as np  # подключим numpy для генерации выборок
# сгенерируем случайную выборку из распределения хи-квадрат размером 100
ser2 = pd.Series(np.random.chisquare(df=5, size=100).round())
ser2.value_counts()  # построим таблицу абсолютных частот
```

```
3.0     20
5.0     14
1.0     14
2.0     12
6.0     11
4.0     11
7.0      7
9.0      7
8.0      3
11.0     1
dtype: int64
```

```
ser2.value_counts(normalize=True)  # таблица относительных частот (долей)
```

```
3.0     0.20
5.0     0.14
1.0     0.14
2.0     0.12
6.0     0.11
4.0     0.11
7.0     0.07
9.0     0.07
8.0     0.03
11.0    0.01
dtype: float64
```

```
ser2.value_counts(bins=5)  # сгруппируем значения по 5 интервалам
```

```
(0.989, 3.0]    46
(3.0, 5.0]      25
(5.0, 7.0]      18
(7.0, 9.0]      10
(9.0, 11.0]      1
dtype: int64
```

Метод `.unique()` формирует NumPy-массив из уникальных (неповторяющихся) значений ряда:

```
pd.Series([0, 0, 0, 1, 0, 2]).unique()  # получим уникальные значения
```

```
array([0, 1, 2])
```

Перебор элементов ряда

В завершение знакомства с типом `Series` взглянем на примеры кода для перебора элементов ряда в цикле. Впрочем, читатель, внимательно изучавший материал предыдущих глав, нового для себя не откроет. И не стоит забывать, что при возможности использования векторизованных операций (в том случае, если оно оправданно, не усложняет алгоритм и *значительно* не ухудшает читаемость кода) *от циклов нужно отказываться* в их пользу.

```
# перебор элементов по значениям
for value in ser1:
  print(value, end=" ")
```
```
70 60 5 20
```

```
# перебор элементов по индексам (вывод аналогичен предыдущему)
for ind in ser1.index:
  print(ser1.loc[ind], end=" ")
```

```
# перебор элементов по порядковым номерам (вывод аналогичен предыдущему)
for i in range(len(ser1)):
  print(ser1.iloc[i], end=" ")
```

13.2. Таблицы (тип данных DataFrame)

В этом параграфе познакомимся с основным типом данных библиотеки pandas, предназначенным для хранения и обработки табличных данных, и попытаемся разобраться, как этот непростой тип устроен и как его эффективно использовать.

Структура ***DataFrame*** («кадр данных»), как и `Series`, является надстройкой над NumPy-массивом с *двумя* измерениями. Это **изменяемая индексированная** коллекция, элементами которой являются *ряды* (`Series`). В русскоязычном сообществе специалистов в сфере наук о данных распространено следующее название этой структуры – ***датафрейм***, которое мы также будем использовать для удобства и соблюдения единства терминологии. Как и ряд, датафрейм можно наглядно представить в виде *таблицы* размером $n \times m$ (n строк, m столбцов) (рис. 94).

	Столбец 1	Столбец 2	…	Столбец m
Строка 1	Значение 1, 1	Значение 1, 2	…	Значение 1, m
Строка 2	Значение 2, 1	Значение 2, 2	…	Значение 2, m
…	…	…	…	…
Строка n	Значение n, 1	Значение n, 2	…	Значение n, m

Рис. 94. Наглядное представление датафрейма в виде таблицы

В силу своей природы датафрейм очень хорошо подходит для представления табличных данных: строки соответствуют отдельным объектам выборки, а столбцы – их признакам. Каждый столбец датафрейма является объектом `Series` и имеет собственное название, при этом столбцы-ряды обладают *общим набором индексов* (рис. 95).

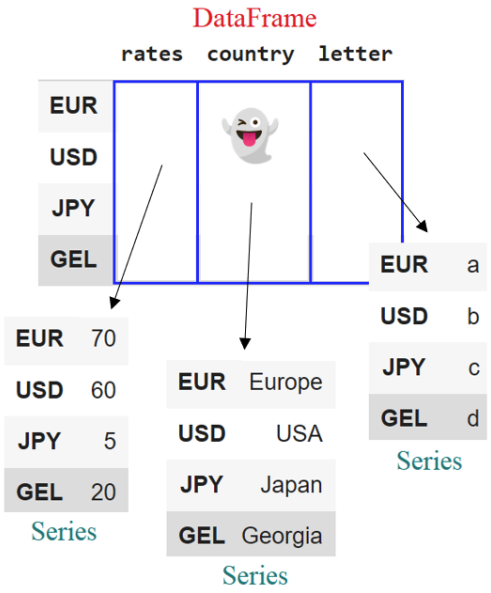


Рис. 95. Представление датафрейма в виде набора рядов-столбцов

Датафрейм является комплексной табличной структурой данных, которую можно рассматривать и как *набор именованных столбцов* (в примере на рис. 95 имена столбцов – *rates*, *country*, *letter*), и как *набор именованных строк* (в примере имена строк, они же индексы – *EUR*, *USD*, *JPY*, *GEL*). По именам можно обращаться как к столбцам, так и к строкам, результатом таких обращений будет ряд-столбец или ряд-строка соответственно. Но *основное* представление датафрейма – коллекция *столбцов*. Можно вспомнить *списки списков* (см. параграф 6.1), в которых внешний список обычно интерпретируется как набор строк, а внутренние – как наборы ячеек, элементов этих строк (см. рис. 40). Датафрейм же имеет перевернутое относительно списка списков представление: элементами внешней коллекции являются именно *столбцы*, а не строки.

Чтобы после всего написанного не запутаться в устройстве датафреймов окончательно, рассмотрим работу с ними на простых примерах. Но прежде упомянем *три важных атрибута* `DataFrame`:

- ✓ `.columns` – коллекция названий колонок;
- ✓ `.index` – коллекция индексов строк;
- ✓ `.values` – двумерный NumPy-массив (матрица) значений.

Создание датафрейма

Ряд `ser1` хранит данные о курсах нескольких валют:

```
ser1  # вспомним содержимое ряда ser1 из предыдущего параграфа
```

```
EUR    70
USD    60
JPY     5
GEL    20
dtype: int64
```

Пусть нашим объектом данных будет *валюта*, которая характеризуется несколькими признаками, в частности ее курсом (например, к рублю) на некоторую неопределенную дату. Но этот признак описывается лишь одним столбцом таблицы с данными о валютах. Добавим второй признак – название страны (региона), в котором данная валюта имеет хождение.

```
# создадим еще один ряд из списка
ser2 = pd.Series(
  ["Europe", "USA", "Japan", "Georgia"],  # перечислим значения ряда
  index=ser1.index  # зададим те же индексы, что у первого ряда
)
ser2
```

```
EUR     Europe
USD        USA
JPY      Japan
GEL    Georgia
dtype: object
```

```
df = pd.DataFrame()  # создадим пустой датафрейм
df["rates"] = ser1  # добавим в него столбец с названием rates, в качестве
# значений возьмем ряд ser1 (при этом индексами строк датафрейма
# стали индексы этого ряда)
df  # получился датафрейм из одного столбца
```

	rates
EUR	70
USD	60
JPY	5
GEL	20

Немного отвлечемся от создания датафрейма и обратим внимание на вывод объекта `DataFrame` на экран. Можно заметить, что это не обычное текстовое представление, характерное для более простых типов данных, включая ряды. Типичный для Colab механизм вывода содержимого объектов без использования функции `print()`, описанный в параграфе 3.6, в случае датафреймов отображает

их в красивом графическом виде[11]. Однако и текстовое представление возможно, для этого применяется `print()`:

```
print(df)  # вывод датафрейма в простом текстовом виде
```

```
     rates
EUR     70
USD     60
JPY      5
GEL     20
```

Существует метод `.to_markdown()`, позволяющий добиться более наглядного табличного вида с отображением границ строк и столбцов (`Series` тоже его поддерживает):

```
print(df.to_markdown())  # "более табличный" текстовый вид
```

```
|     |   rates |
|:----|--------:|
| EUR |      70 |
| USD |      60 |
| JPY |       5 |
| GEL |      20 |
```

Вернемся к конструированию датафрейма и добавим в него второй столбец с названиями стран:

```
df["country"] = ser2  # добавим столбец country, полученный из ряда ser2
df
```

	rates	country
EUR	70	Europe
USD	60	USA
JPY	5	Japan
GEL	20	Georgia

Как и ряд, датафрейм можно получить путем преобразования в него словаря определенной структуры. Ключи данного словаря интерпретируются как *названия столбцов*, а значениями должно быть *содержимое этих столбцов* в виде индексированных коллекций (например, списков или рядов, *но не строк*). Также значение может быть *скалярным* (в том числе строкой), тогда оно продублируется во всех строках таблицы. Для примера сконструируем датафрейм такого же содержимого, как на рис. 95:

[11] Также графическое представление датафреймов, помимо придания данным наглядности, позволяет выполнять с ними различные интерактивные действия, такие как фильтрация, поиск и построение диаграмм. Возможности этого аналитического инструментария, предоставляемого платформой Colab, мы рассматривать не будем.

```
# создадим датафрейм из словаря (не сохраняя его в переменную)
pd.DataFrame({
  "rates": ser1,  # столбец rates (из ряда)
  "country": list(ser2),  # столбец country (из списка)
  "letter": tuple("abcd")  # столбец letter (из кортежа)
})
```

	rates	country	letter
EUR	70	Europe	a
USD	60	USA	b
JPY	5	Japan	c
GEL	20	Georgia	d

Получение общей информации о датафрейме

Как уже было отмечено, атрибуты `.columns`, `.index` и `.values` позволяют получить названия измерений и матрицу значений датафрейма.

```
df.columns  # атрибут columns - названия столбцов (колонок)
```
```
Index(['rates', 'country'], dtype='object')
```

```
df.index  # атрибут index - индексы строк
```
```
Index(['EUR', 'USD', 'JPY', 'GEL'], dtype='object')
```

```
df.values  # атрибут values - матрица значений
```
```
array([[70, 'Europe'],
       [60, 'USA'],
       [5, 'Japan'],
       [20, 'Georgia']], dtype=object)
```

Как и у NumPy-массива, у датафрейма есть атрибут `.shape`, содержащий кортеж с его размерами (количеством строк и столбцов). Если вспомнить, что датафрейм является коллекцией, эти сведения можно получить с помощью универсальной функции `len()`:

```
df.shape  # 4 строки, 2 столбца: (4, 2)
len(df)  # число строк: 4
len(df.columns)  # число столбцов: 2
```

Методы `.head()` и `.tail()` возвращают заданное количество первых и последних соответственно строк датафрейма (по умолчанию – 5).

```
df.head(2)  # вывод первых двух строк датафрейма, последние две - df.tail(2)
```

	rates	country
EUR	70	Europe
USD	60	USA

Метод `.info()` выводит сводную информацию о датафрейме: количество строк и колонок, названия колонок, количество непустых значений в них, типы данных.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4 entries, EUR to GEL
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   rates    4 non-null      int64 
 1   country  4 non-null      object
dtypes: int64(1), object(1)
memory usage: 268.0+ bytes
```

Добавление и удаление данных

Новый *столбец* в датафрейм добавляется тем же способом, что и новый элемент в ряд, только вместо индекса элемента указывается название столбца.

```
df["новая колонка"] = [0] * len(df)  # добавим новый столбец с названием
# "новая колонка", запишем в него список из такого же числа элементов,
# сколько в датафрейме строк (можно и скалярное значение)
df  # столбец добавился в конец
```

	rates	country	новая колонка
EUR	70	Europe	0
USD	60	USA	0
JPY	5	Japan	0
GEL	20	Georgia	0

Чтобы добавить столбец на произвольную позицию, можно использовать метод `.insert()`:

```
<датафрейм>.insert(<позиция>, <название столбца>, <содержимое столбца>)
```

```
df.insert(1, "new_col_2", 7)  # вставим столбец из семерок вторым по счету
df
```

	rates	new_col_2	country	новая колонка
EUR	70	7	Europe	0
USD	60	7	USA	0
JPY	5	7	Japan	0
GEL	20	7	Georgia	0

Для добавления новой строки в датафрейм используется атрибут `.loc`, в скобках указывается индекс строки.

```
df.loc["новая строка"] = 1  # строку также можно задать вектором
# (число элементов должно быть равно числу столбцов) или скаляром
df
```

	rates	new_col_2	country	новая колонка
EUR	70	7	Europe	0
USD	60	7	USA	0
JPY	5	7	Japan	0
GEL	20	7	Georgia	0
новая строка	1	1	1	1

Учитывая, что при создании датафрейма индексы строк по умолчанию имеют значения от 0 до $n-1$, где n – количество строк, удобно при добавлении новой строки использовать следующую конструкцию:

```
<датафрейм>.loc[len(<датафрейм>)] = <содержимое строки>
```

В этом случае индекс новой строки будет соответствовать ее корректному порядковому номеру (нумерация с нуля).

Удаление строк и столбцов выполняется с помощью метода `.drop()`:

```
<датафрейм>.drop(<название строки или столбца>, axis=<измерение>)
```

```
# удаление строки с сохранением изменений
df.drop("новая строка", axis="rows", inplace=True)  # axis=0 - то же самое

# удаление столбца с сохранением изменений
df.drop("новая колонка", axis="columns", inplace=True)  # или axis=1
df.drop("new_col_2", axis=1, inplace=True)

df
```

	rates	country
EUR	70	Europe
USD	60	USA
JPY	5	Japan
GEL	20	Georgia

Извлечение данных из датафрейма

Для обращения к отдельному *столбцу* достаточно указать его название в квадратных скобках. В результате будет получен *ряд* (объект `Series`). Способ со скобками является универсальным, однако есть и другой способ получения столбца: обращение к нему как к *атрибуту датафрейма*. Это сработает лишь в том случае, если название столбца не содержит пробелов и иных символов, недопустимых в именах объектов (см. параграф 3.1).

```
df["rates"]  # извлечем столбец rates, другой способ: df.rates
# получим ряд; отметим, что у ряда может быть атрибут name
```
```
EUR    70
USD    60
JPY     5
GEL    20
Name: rates, dtype: int64
```

Для удобства работы с отдельным столбцом можно поместить его в переменную (фактически назначить ссылку на конкретный объект-столбец). Все модификации ряда будут сполна отражены в столбце датафрейма (ведь объект один и тот же):

```
rates = df.rates  # поместим столбец rates в переменную типа Series
```

Обращение к отдельной *строке* осуществляется через атрибут `.loc`.

```
df.loc["EUR"]  # извлечем строку с индексом EUR, получим ряд
```
```
rates          70
country    Europe
Name: EUR, dtype: object
```

Указание в квадратных скобках не скалярного, а *векторного* значения вернет не ряд, а *датафрейм*. Аналогичный механизм работает с NumPy-массивами, и он уже был нами использован при формировании векторов-строк и векторов-столбцов вместо плоских векторов (см. пример кода в параграфе 12.3).

```
df[["rates"]]  # датафрейм из одного столбца
df.loc[["EUR"]]  # датафрейм из одной строки
```

Извлечение отдельных элементов и фрагментов датафрейма осуществляется по аналогии с извлечением данных из ряда. Отличие в том, что датафрейм является *таблицей*, *матрицей*, а следовательно, положение его элементов описывается *двумя значениями*. Конечно, поддерживаются и *срезы*. Если читатель внимательно изучил материал про извлечение данных из матриц и рядов, общие механизмы соответствующих операций с датафреймами будут ему понятны. Приведем несколько примеров:

```
df.loc["USD", "country"]  # извлечем значение по индексам (меткам): USA
df.iloc[1, 1]  # то же значение по номерам строки и столбца

df.loc[["USD", "JPY"], ["rates", "country"]]  # извлечем подтаблицу
# только из указанных строк и столбцов (по индексам)
# df.iloc[[1, 2], [0, 1]]  # то же самое по номерам
```

	rates	country
USD	60	USA
JPY	5	Japan

```
df.loc["EUR":"USD", "rates":]  # строки EUR-USD, столбцы с rates до конца
df.iloc[:-2, 0:]  # результат аналогичен

df.loc[:, "rates"]  # все строки, столбец rates (первый), результат - ряд
df.iloc[:, [0]]  # все строки, первый столбец (rates), результат - датафрейм
```

Стоит отметить, что удобным *способом удаления строк и столбцов* датафрейма является извлечение подтаблицы, включающей только нужные строки и столбцы, и сохранение ее в переменную, содержащую исходный датафрейм. Иными словами, для удаления строк и столбцов их можно просто *не включать в отбор* с перезаписью датафрейма результатами этого отбора.

```
df["newcol"] = 0  # добавим столбец
df.loc[len(df)] = 0  # добавим строку
# а теперь удалим их: извлечем подтаблицу без последних столбца и строки
df = df.iloc[:-1, :-1]  # и сохраним ее в переменную df
```

Отбор данных по условию

Механизм отбора данных из датафрейма по битовым картам аналогичен таковому для рядов, с той разницей, что критерии отбора для строк и столбцов задаются отдельно.

```
# отберем элементы, удовлетворяющие условиям:
# 1) индекс строки больше U;
# 2) значение в столбце rates больше 20;
# 3) название столбца меньше r
df.loc[
  (df.index > "U") & (df.rates > 20),  # отбор по строкам, условия 1 и 2
  df.columns < "r"  # отбор по столбцам, условие 3
]
# результат - датафрейм из одной строки и одного столбца
```

	country
USD	USA

Как можно заметить, отбор по битовым картам работает как со строками, так и со столбцами, но на практике *по условию отбирают именно строки*, необходимые же столбцы задают перечислением их названий (номеров) или с помощью среза. Дело в том, что векторизованные операции хорошо применимы к столбцам ввиду однородности их значений (как правило, тип всех элементов столбца одинаков), и проблем с конструированием битовых карт с помощью опе-

раторов сравнения не возникает (все значения столбца можно сравнить с константой или со значениями другого вектора того же типа). Строки же состоят из элементов разного типа, применять к ним операции сравнения не только сложнее, но и зачастую лишено смысла.

Еще одним способом отбора строк датафрейма является применение метода `.query()`. Критерии отбора задаются строкой:

```
# отберем строки с индексом <= U и rates <= 50
df.query("(index <= 'U') & (rates <= 50)")
```

	rates	country
JPY	5	Japan
GEL	20	Georgia

Копирование датафреймов

Датафреймы (как и ряды, о чем мы ранее не упомянули) имеют метод `.copy()`, поддерживающий простое и глубокое копирование (вспомнить, в чем между ними разница, можно, вернувшись к параграфу 6.5). Режим задается аргументом `deep`, его значение по умолчанию `True` (глубокое копирование).

Соединение датафреймов

Функция `concat()` умеет соединять не только ряды, но и датафреймы. Аргумент `axis` задает способ соединения таблиц: соединение по строкам (`"index"` или 0) – вертикальное (выполняется по умолчанию), соединение по столбцам (`"columns"` или 1) – горизонтальное. Сконструируем новый датафрейм `df2` на базе `df`:

```
df2 = df.copy().iloc[:, [-1]]  # получим независимую копию первого датафрейма,
# отберем только последний столбец. Также добавим новый столбец
df2["capital"] = "Brussels", "Washington", "Tokyo", "Tbilisi"
# и две новые строки
df2.loc["KZT"] = "Kazakhstan", "Astana"
df2.loc["CNY"] = "China", "Beijing"
df2
```

	country	capital
EUR	Europe	Brussels
USD	USA	Washington
JPY	Japan	Tokyo
GEL	Georgia	Tbilisi
KZT	Kazakhstan	Astana
CNY	China	Beijing

Выполним соединение датафреймов:

```
pd.concat([df, df2])  # соединим по вертикали исходный и новый датафреймы
```

	rates	country	capital
EUR	70.0	Europe	NaN
USD	60.0	USA	NaN
JPY	5.0	Japan	NaN
GEL	20.0	Georgia	NaN
EUR	NaN	Europe	Brussels
USD	NaN	USA	Washington
JPY	NaN	Japan	Tokyo
GEL	NaN	Georgia	Tbilisi
KZT	NaN	Kazakhstan	Astana
CNY	NaN	China	Beijing

```
pd.concat([df, df2], axis=1)  # соединим датафреймы по горизонтали
```

	rates	country	country	capital
EUR	70.0	Europe	Europe	Brussels
USD	60.0	USA	USA	Washington
JPY	5.0	Japan	Japan	Tokyo
GEL	20.0	Georgia	Georgia	Tbilisi
KZT	NaN	NaN	Kazakhstan	Astana
CNY	NaN	NaN	China	Beijing

Как можно заметить, если набор колонок или строк (при вертикальном или горизонтальном сцеплении соответственно) соединяемых таблиц не совпадает, отсутствующие колонки (строки) *заполняются пропусками* (NaN-значениями). Подробнее о пропусках скажем ниже, а сейчас лишь отметим, что они относятся к типу `float64` и их наличие в *целочисленном* столбце датафрейма приводит к изменению его типа на более общий – вещественный. При этом *строковые* столбцы тип не меняют, потому что он у них и так наиболее общий, допускающий любые значения – `object`.

Функция `merge()` позволяет соединять датафреймы *в стиле SQL-запросов*. Существует также одноименный *метод* датафрейма. О реляционных базах данных и языке SQL говорить не станем (хотя датафреймы – это таблицы, поэтому применение к ним операций реляционной алгебры выглядит логичным), приведем лишь несколько примеров использования функции:

```
# объединим датафреймы df и df2 в один по столбцу country
pd.merge(
  left=df, right=df2,  # левый и правый соединяемые датафреймы
  left_on="country",  # общий столбец левого датафрейма
  right_on="country",  # общий столбец правого датафрейма
  how="outer"  # тип соединения - внешнее
)
```

	rates	country	capital
0	70.0	Europe	Brussels
1	60.0	USA	Washington
2	5.0	Japan	Tokyo
3	20.0	Georgia	Tbilisi
4	NaN	Kazakhstan	Astana
5	NaN	China	Beijing

Что сделала функция `merge()`? Она соединила левую и правую таблицы по их общему столбцу *country*, то есть сопоставила строкам левой таблицы строки правой с таким же значением общего столбца. При этом был использован тип соединения «внешнее» (значение `"outer"` аргумента `how`). Внешнее соединение таблиц похоже на *объединение множеств* (см. параграф 10.1): в итоговую таблицу попали *все* строки обеих исходных таблиц, при этом недостающие ячейки (в частности, в первой таблице нет данных о странах Казахстан и Китай, а во второй они есть) заполнились NaN-значениями.

```
# выполним внутреннее соединение df и df2
# (в результат попадут только строки, присутствующие в обеих таблицах)
df.merge(  # df - левый датафрейм
  df2,  # правый датафрейм
  left_on="country", right_on="country",  # соединяем по country
  how="inner"  # тип соединения - внутреннее (похоже на пересечение множеств)
)
```

	rates	country	capital
0	70	Europe	Brussels
1	60	USA	Washington
2	5	Japan	Tokyo
3	20	Georgia	Tbilisi

Векторизованные операции

Как NumPy-массивы и ряды, датафреймы поддерживают векторизованные операции. Они выполняются над всеми ячейками таблицы.

```
# поэлементно сложим датафрейм df и его же с удвоенными значениями ячеек
df + df * 2
```

	rates	country
EUR	210	EuropeEuropeEurope
USD	180	USAUSAUSA
JPY	15	JapanJapanJapan
GEL	60	GeorgiaGeorgiaGeorgia

Прокомментируем результат кода выше на примере строки с курсом японской йены (индекс *JPY*). Значение ячейки *rates* получено следующим образом: `5 + 5 * 2`, значение *country*: `"Japan" + "Japan" * 2`. Операция умножения на целое число работает как с числами, так и со строками, поэтому применима к данному датафрейму.

Применение функции к элементам датафрейма

Датафрейм имеет собственный аналог `Series`-метода `.apply()`, применяющего заданную функцию ко всем элементам ряда, – `.applymap()`. Данный метод применяет функцию ко всем элементам датафрейма и возвращает, соответственно, *датафрейм*.

```
# применим функцию type() ко всем ячейкам датафрейма. Эта функция выбрана
# по той причине, что она работает со значениями любого типа
df.applymap(type)
```

	rates	country
EUR	<class 'int'>	<class 'str'>
USD	<class 'int'>	<class 'str'>
JPY	<class 'int'>	<class 'str'>
GEL	<class 'int'>	<class 'str'>

Метод `.apply()` у датафрейма тоже есть, однако он применяет заданную функцию не ко всем элементам таблицы, а к ее *столбцам* или *строкам*. Результатом является *ряд*. Разберемся, как это работает.

```
# применим функцию max() к столбцам
# измерение задается аргументом axis, по умолчанию axis="rows" или 0
# (это не ошибка: значение аргумента "rows" применяет функцию к столбцам)
df.apply(max)
```

```
rates       70
country    USA
dtype: object
```

Результат выполнения кода выше проще всего понять, взглянув на иллюстрацию (рис. 96). Функция `max()` возвращает максимальное значение коллекции (в данном случае коллекцией выступает *столбец*, имеющий тип `Series`). Столбцов всего два, поэтому результатом является ряд из двух максимальных значений, индексами которых являются имена соответствующих столбцов.

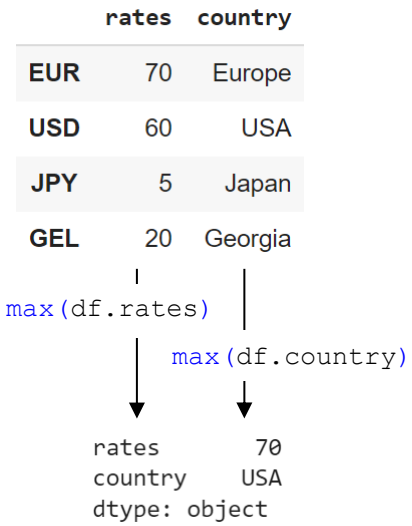


Рис. 96. Применение функции к столбцам датафрейма

```
# применим функцию max() к строкам, для этого предварительно приведем значения
# к типу str (потому что число и строку сравнивать нельзя, а строки можно)
df.astype(str).apply(max, axis="columns")  # или axis=1 - то же самое
```

```
EUR     Europe
USD        USA
JPY      Japan
GEL    Georgia
dtype: object
```

В случае применения функции к строкам входными данными `max()` по-прежнему являются ряды, но уже не столбцы, а *строки*. В остальном механизм аналогичен рассмотренному выше (рис. 97). Если вызывает удивление тот факт, что при сравнении строки из цифр со строкой из букв больше оказывается последняя, можно вспомнить механизм сравнения строк, описанный в параграфе 8.1.

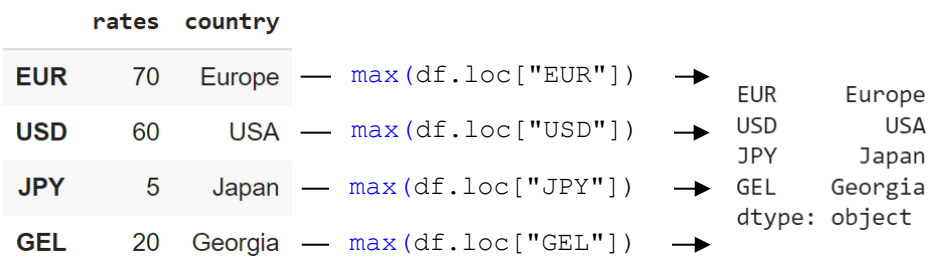


Рис. 97. Применение функции к строкам датафрейма

Замена значений

Метод `.replace()` заменяет одно значение на другое.

```
df.replace(70, 1000)  # заменим в датафрейме 70 на 1000
```

	rates	country
EUR	1000	Europe
USD	60	USA
JPY	5	Japan
GEL	20	Georgia

Возможна замена по словарю следующей структуры:

```
{
  <старое значение 1>: <новое значение 1>,
  <старое значение 2>: <новое значение 2>,
  ...
}
```

```
# заменим два значения по словарю
df.replace({
  "Europe": "Европа",
  "Georgia": "Грузия"
})
```

	rates	country
EUR	70	Европа
USD	60	USA
JPY	5	Japan
GEL	20	Грузия

Переименование строк и столбцов

Метод `.rename()` позволяет переименовывать как столбцы, так и строки (названиями строк являются индексы).

```
# переименуем несколько столбцов и строк
df.rename(
  columns={"rates": "курс валюты", "country": "страна"},
  index={"EUR": "евро", "USD": "доллар", "JPY": "йена", "GEL": "лари"}
)
```

	курс валюты	страна
евро	70	Europe
доллар	60	USA
йена	5	Japan
лари	20	Georgia

Формирование случайной выборки строк

Метод `.sample()` извлекает из датафрейма заданное количество случайных строк (по умолчанию – одну). Аргумент `random_state` фиксирует состояние генератора случайных чисел, то есть делает то же самое, что NumPy-функция `random.seed()`, в рамках одного метода. Данный метод также поддерживается объектами `Series`.

```
# выберем три случайных строки
df.sample(3, random_state=123)  # при значении 123 строки, извлекаемые
# из данного конкретного датафрейма, всегда будут одни и те же
```

	rates	country
GEL	20	Georgia
EUR	70	Europe
USD	60	USA

```
# перемешать датафрейм - все равно что выбрать все его строки
df.sample(len(df))  # размер извлекаемой выборки равен размеру датафрейма
```

	rates	country
GEL	20	Georgia
EUR	70	Europe
USD	60	USA
JPY	5	Japan

Перебор строк и столбцов датафрейма

Несмотря на поддержку датафреймами векторизованных операций (и настоятельная рекомендация по возможности этой поддержкой пользоваться), иногда бывает необходимо перебрать строки или столбцы датафрейма в обычном цикле. Приведем примеры кода, решающего указанную задачу:

```
# перебор строк по индексам
for ind in df.index:
  row = df.loc[ind]  # row - строка датафрейма (Series)

# перебор строк по порядковым номерам
for i in range(len(df)):
  row = df.iloc[i]  # row - строка датафрейма (Series)

# перебор столбцов по названиям
for col in df.columns:  # можно обращаться просто к df, будет то же самое
  column = df[col]  # column - столбец датафрейма (Series)

# перебор столбцов по порядковым номерам
for i in range(len(df.columns)):
  column = df.iloc[:, i]  # column - столбец датафрейма (Series)
```

Некоторые другие методы DataFrame

Рассмотрим ряд методов датафреймов, не упомянутых выше.

Метод `.transpose()` осуществляет транспонирование таблицы (при этом названия столбцов превращаются в индексы строк, и наоборот). Обращение к атрибуту `.T` приводит к аналогичному результату:

```
df.transpose()  # df.T - то же самое
```

	EUR	USD	JPY	GEL
rates	70	60	5	20
country	Europe	USA	Japan	Georgia

Метод `.sort_values()` сортирует строки по значениям в заданном столбце. Для указания столбца используется аргумент `by`. Может быть задано несколько столбцов списком, в этом случае сортировка выполняется по первому столбцу, при наличии в нем одинаковых значений упорядочение происходит по второму столбцу, и т.д. Также существует метод `.sort_index()`, он сортирует строки по индексам; результат его работы в демонстрации и пояснениях не нуждается.

```
# искусственно создадим дубликат в столбце country, чтобы увидеть результат
# сортировки по нескольким столбцам (сначала по стране, потом по курсу)
df.replace("Georgia", "Europe").sort_values(by=["country", "rates"])
```

	rates	country
GEL	20	Europe
EUR	70	Europe
JPY	5	Japan
USD	60	USA

Метод `.duplicated()` выявляет дубликаты (повторяющиеся строки). Возвращает ряд – битовую карту, в которой значением `True` обозначены все повторы.

```
# присоединим к датафрейму две его же строки, они будут помечены как дубликаты
pd.concat([df, df.iloc[::2]]).duplicated()
```

```
EUR    False
USD    False
JPY    False
GEL    False
EUR     True
JPY     True
dtype: bool
```

Метод `.drop_duplicates()` удаляет повторяющиеся строки.

```
# присоединим к датафрейму его же и удалим дубликаты, результат совпадает
pd.concat([df, df]).drop_duplicates()  # с исходным датафреймом
```

	rates	country
EUR	70	Europe
USD	60	USA
JPY	5	Japan
GEL	20	Georgia

Наконец, рассмотрим группу методов датафрейма, вычисляющих некоторые количественные характеристики хранимых в нем данных (*описательные статистики*). Для большей наглядности добавим в наш датафрейм два числовых столбца.

```
df["population"] = [300, 10, 100, 330]  # население страны, млн чел.
df["GDP"] = [90, 15, 80, 150]  # ВВП страны
df
```

	rates	country	population	GDP
EUR	70	Europe	300	90
USD	60	USA	10	15
JPY	5	Japan	100	80
GEL	20	Georgia	330	150

Метод `.sum()` вычисляет сумму по столбцам или строкам. Аргумент `numeric_only` указывает, участвуют в расчетах все столбцы или только числовые. Результатом является ряд.

```
# вычислим сумму по всем столбцам, включая нечисловые
df.sum(axis="rows", numeric_only=False)  # числа сложились, строки сцепились
```

```
rates                             155
country       EuropeUSAJapanGeorgia
population                        740
GDP                               335
dtype: object
```

Перечислим другие подобные методы с тем же набором аргументов:

```
df.mean(axis="rows", numeric_only=True)  # среднее по числовым столбцам
df.median(axis="rows", numeric_only=True)  # медиана по числовым столбцам
df.mode(axis="rows", numeric_only=True)  # мода по числовым столбцам
# (результат - датафрейм, потому что мод в столбце может быть несколько)
df.var(axis="rows", numeric_only=True, ddof=1)  # несмещенная дисперсия
```

```
df.std(axis="rows", numeric_only=True, ddof=1)  # несмещенное стандартное
# отклонение (ddof - это число, которое вычитается из объема выборки
# в знаменателе формул дисперсии и стандартного отклонения)
df.max(axis="rows", numeric_only=True)  # максимум по числовым столбцам
df.min(axis="rows", numeric_only=True)  # минимум по числовым столбцам
# (методами max(), min() аргумент key не поддерживается)
```

Метод `.describe()` выводит некоторые описательные статистики по столбцам в виде датафрейма. Аргумент `include` позволяет задать нужные типы столбцов списком. Возможны также значения `"all"` (все столбцы) и `None` (по умолчанию – только числовые столбцы). Значения, обозначенные процентами, – это первый, второй и третий квартили соответственно. Набор статистик, вычисляемых по нечисловым столбцам, отличается (предлагаем убедиться в этом самостоятельно).

```
df.describe()  # посмотрим на описательные статистики по числовым столбцам
```

	rates	population	GDP
count	4.000000	4.000000	4.000000
mean	38.750000	185.000000	83.750000
std	31.191612	155.026879	55.283361
min	5.000000	10.000000	15.000000
25%	16.250000	77.500000	63.750000
50%	40.000000	200.000000	85.000000
75%	62.500000	307.500000	105.000000
max	70.000000	330.000000	150.000000

Методы `.cov()` и `.corr()` конструируют квадратные матрицы ковариации и корреляции столбцов датафрейма соответственно. У метода `.corr()` есть аргумент `method`, который может принимать в качестве значения строковые константы `"pearson"` (вычислять коэффициент корреляции Пирсона по умолчанию), `"spearman"` и `"kendall"` (ранговые коэффициенты Спирмена и Кенделла соответственно), а также *функцию*. Аргументами этой функции должны быть два ряда, а вот возвращаемое значение коэффициентом корреляции может и не быть. Еще одной особенностью метода `.corr()` является то, что на главной диагонали результирующего датафрейма всегда располагаются *единицы* (то есть эти значения не вычисляются).

```
# матрица корреляции Пирсона между числовыми столбцами
df.corr(numeric_only=True, method="pearson")
```

	rates	population	GDP
rates	1.000000	0.001723	-0.460311
population	0.001723	1.000000	0.881911
GDP	-0.460311	0.881911	1.000000

Обработка пропусков и выбросов

Пропущенные (пустые) значения, или ***пропуски*** – это незаполненные ячейки в массивах данных. В анализе данных нередко бывает так, что часть информации об объекте отсутствует. Например, в группе студентов некоторые уже сдали экзамен по программированию и имеют оценку, а некоторые сдать еще не успели. Датасет с данными о студентах должен включать признак «Оценка по программированию», однако не для всех объектов он будет заполнен. Другой типичный пример данных с пропусками: временные ряды котировок акций не содержат значений на даты праздничных дней по причине отсутствия в эти дни торгов на бирже. В компьютерных науках пропуски в данных обычно обозначаются так называемыми *NaN-значениями* (Not a Number, «не число»). В Python реализация NaN-значения есть в модуле math (`math.nan`), а также в библиотеке NumPy (`numpy.nan`). Оба значения относятся к типу `float`, однако не равны между собой. Рассмотрим несколько методов датафреймов, предназначенных для обработки пропусков.

Метод `.dropna()` удаляет строки или столбцы с пропусками. Аргумент `thresh` задает порог удаления – минимально допустимое количество *непустых* значений: если их меньше порога, строка (столбец) удаляется. По умолчанию удаляются строки (столбцы) при наличии в них хотя бы одного пропуска.

```
# чтобы пропуски появились, добавим их вручную
df_nan = df.copy()  # исходный датафрейм портить не будем, создадим копию
# точечно проставим в ней пропуски
df_nan.iloc[0, 0], df_nan.iloc[0, 2], df_nan.iloc[2, 2] = (np.nan,) * 3
df_nan  # посмотрим на результат
```

	rates	country	population	GDP
EUR	NaN	Europe	NaN	90
USD	60.0	USA	10.0	15
JPY	5.0	Japan	NaN	80
GEL	20.0	Georgia	330.0	150

```
df_nan.dropna(axis="rows")  # удалим строки с пропусками (хотя бы с одним)
```

	rates	country	population	GDP
USD	60.0	USA	10.0	15
GEL	20.0	Georgia	330.0	150

```
# удалим строки, в которых меньше трех непустых значений
df_nan.dropna(axis="rows", thresh=3)
```

	rates	country	population	GDP
USD	60.0	USA	10.0	15
JPY	5.0	Japan	NaN	80
GEL	20.0	Georgia	330.0	150

```
df_nan.dropna(axis="columns")  # удалим столбцы с пропусками
```

	country	GDP
EUR	Europe	90
USD	USA	15
JPY	Japan	80
GEL	Georgia	150

```
# удалим столбцы, в которых меньше трех непустых значений
df_nan.dropna(axis="columns", thresh=3)
```

	rates	country	GDP
EUR	NaN	Europe	90
USD	60.0	USA	15
JPY	5.0	Japan	80
GEL	20.0	Georgia	150

Метод `.fillna()` заполняет пропуски заданным значением.

```
df_nan.fillna(0)  # заполним пропуски нулями
```

	rates	country	population	GDP
EUR	0.0	Europe	0.0	90
USD	60.0	USA	10.0	15
JPY	5.0	Japan	0.0	80
GEL	20.0	Georgia	330.0	150

Метод `.interpolate()` заполняет пропуски с помощью *интерполяции* (вычисления пропущенного значения на основе соседних значений). Простейший способ интерполяции – усреднение двух соседних по столбцу или строке значений.

```
# заполним пропуски интерполяцией (усреднением соседних значений по столбцам)
df_nan.interpolate(axis="rows")
# пропуски в 1-й строке не заполнились, ведь у них только одно значение рядом
```

	rates	country	population	GDP
EUR	NaN	Europe	NaN	90
USD	60.0	USA	10.0	15
JPY	5.0	Japan	170.0	80
GEL	20.0	Georgia	330.0	150

Аномальные значения, или ***выбросы*** – это значения, сильно отличающиеся от других значений набора данных (очень большие или очень малые). В анализе данных применяются различные способы их выявления (например, с помощью построения диаграммы «ящик с усами», этот способ рассмотрим в главе 14) и обработки. Подробное изложений методик работы с выбросами и пропусками выходит за рамки материала данного пособия, поэтому ограничимся упоминанием *двух простых способов* обработки выбросов:

✓ выбросы можно заменить *пропусками* и обрабатывать соответствующими методами;

✓ выбросы можно «сгладить» (заменить на «менее аномальные» значения).

В качестве «сглаженных» значений могут быть использованы *квантили* малых или больших (в зависимости от разновидности выброса) порядков. Например, аномально малое значение может быть заменено квантилем порядка 0,01, а аномально большое – квантилем порядка 0,99.

Метод `.quantile()` вычисляет квантили по столбцам или строкам. Аргумент `q` позволяет задать порядки квантилей (может быть одно число или список). Также одноименный метод есть у рядов.

```
# вычислим квантили порядков 0.01 и 0.99 по числовым столбцам
df.quantile(q=[0.01, 0.99], axis="rows", numeric_only=True)
```

	rates	population	GDP
0.01	5.45	12.7	16.95
0.99	69.70	329.1	148.20

13.3. Загрузка и выгрузка наборов данных (датасетов)

Как уже упоминалось, предназначенные для обработки данные часто хранятся и распространяются в *следующих форматах*:

✓ ***CSV-файлы*** – это текстовые файлы псевдотабличного вида, в которых значения ячеек в строках разделены заданным символом (например, запятой);

✓ книги Excel.

Научимся загружать данные из таких файлов, а также выгружать в файл. Не забудем, что табличные данные в pandas хранятся в виде датафреймов.

CSV-файлы

CSV – это сокращение от английского словосочетания **C**omma-**S**eparated **V**alues («значения, разделенные запятой»). Для *загрузки* файлов с разделителями используется функция `read_csv()`:

```
read_csv(<адрес CSV-файла>, sep=<разделитель>, encoding=<кодировка>)
```

Адрес файла может быть задан как в виде относительного или абсолютного *пути к локальному файлу* на компьютере, так и в виде *прямой ссылки на файл* в интернете. Разделитель – это строка, содержащая символ разделителя колонок таблицы, по умолчанию используется запятая. Аргумент `encoding`, как и при открытии обычного текстового файла (см. параграф 10.3), задает кодировку, значение по умолчанию `"utf-8"`. Изучив документацию, можно обнаружить, что функция `read_csv()` имеет очень много аргументов, устанавливающих те или иные параметры загрузки данных, рассматривать их все не станем.

Для примера загрузим известный датасет «Ирисы Фишера», содержащий информацию о цветках разного вида (источник оригинальных данных: ссылка [4]).

```
# загрузим данные из интернета по прямой ссылке
iris_df = pd.read_csv(
  "https://raw.githubusercontent.com/rvgarafutdinov/" \
  "python_book_2023/main/iris.csv", sep=","
)
iris_df.head()  # выведем первые 5 строк
```

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Следует отметить, что в датафрейм из интернета можно загружать только файлы, *доступные по прямой ссылке*. Доступ по прямой ссылке означает, что веб-сервер при получении запроса от клиента возвращает *только содержимое файла* и ничего лишнего. Убедиться, что ссылка прямая, несложно: при открытии такой ссылки в веб-браузере файл либо начинает скачиваться, либо, если он текстовый, отображается на странице как простой текст (без элементов форматирования, присущих веб-страницам).

Загрузим тот же датасет из файла, расположенного локально. Для этого соединим обе части строки со ссылкой в коде выше (из-за чрезмерной длины пришлось эту строку разделить), откроем полученную ссылку в браузере и сохраним файл iris.csv на компьютер. Загрузить его в домашний каталог блокнота можно через файловый обозреватель Colab (см. рис. 72).

```
# загрузим данные из файла в домашнем каталоге, сохраним в ту же переменную
iris_df = pd.read_csv("iris.csv", sep=",")
```

Далее научимся выгружать данные в CSV. Внесем небольшие изменения в датафрейм, а именно переименуем столбцы и названия цветков:

```
# переименуем столбцы
iris_df.rename(columns={
  "sepal length": "длина чашелистика",
  "sepal width": "ширина чашелистика",
  "petal length": "длина лепестка",
  "petal width": "ширина лепестка",
  "class": "вид цветка"
}, inplace=True)

# переименуем значения
iris_df.replace({
  "Iris-setosa": "ирис щетинистый",
  "Iris-versicolor": "ирис разноцветный",
  "Iris-virginica": "ирис виргинский"
}, inplace=True)

iris_df.sample(3)  # взглянем на результат (извлечем 3 случайные строки)
```

	длина чашелистика	ширина чашелистика	длина лепестка	ширина лепестка	вид цветка
52	6.9	3.1	4.9	1.5	ирис разноцветный
3	4.6	3.1	1.5	0.2	ирис щетинистый
137	6.4	3.1	5.5	1.8	ирис виргинский

Для выгрузки датафрейма в CSV-файл используется метод `.to_csv()`. Аргумент `index` позволяет указать, нужно ли включать в файл столбец индексов (по умолчанию равен `True` – включать).

```
#сохраним датасет в файл "iris_ru.csv" с разделителем ";" и без индексов строк
iris_df.to_csv("iris_ru.csv", sep=";", index=False)
```

Полученный файл, если его скачать на компьютер и открыть в «Блокноте» Windows, выглядит следующим образом (рис. 98).

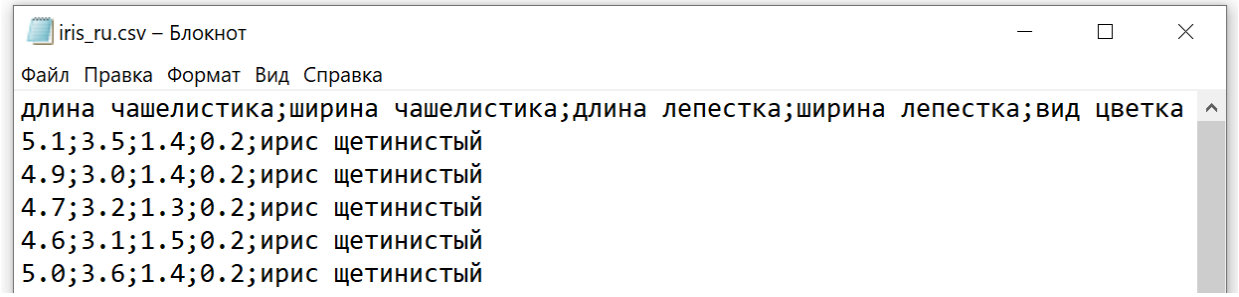


Рис. 98. Выгруженный из датафрейма датасет в CSV-файле

Книги Excel

Microsoft Excel – это табличный процессор, обладающий широкими возможностями по обработке табличных данных. Данные, с которыми он работает, хранятся в так называемых *книгах Excel* (файлах с расширением *.xlsx* или *.xls*). В классическом виде эти данные представляют собой *прямоугольные матрицы* размером $n \times m$, то есть таблицы. Однако Excel позволяет *нарушать* эту прямоугольно-сетчатую структуру, объединяя несколько ячеек в одну, в результате чего общее количество ячеек таблицы оказывается не $n \times m$, а *меньше*. Эта возможность бывает удобна для просмотра данных человеком, позволяет повысить наглядность, однако плохо соотносится с матричным форматом двумерных массивов и датафреймов. Кроме того, Excel, будучи чрезвычайно функционально нагруженным прикладным пакетом с претензиями на некоторую «интеллектуальность», отличается «своенравным» поведением и может, например, при открытии или сохранении файла преобразовывать некоторые записи в ячейках из обычных строк в даты, что может привести к потере важных данных, порой необратимой. Корректно загрузить и обработать в pandas такие «испорченные» файлы не удастся. Ввиду этого использовать книги Excel для хранения результатов обработки данных автором *не рекомендуется*. Здесь можно провести аналогию с JSON- или XML-файлами и HTML-страницами: первые, как и CSV, предназначены для чтения и обработки *машиной*, содержат данные в аутентичном виде, не включают ничего лишнего, в то время как веб-страницы, как и книги Excel, предназначены для просмотра и обработки *человеком* и содержат много информации, не нужной и даже вредной для целей анализа данных, предполагающего использование количественных методов. Однако датасеты часто распространяются в формате Excel, поэтому научиться их загружать необходимо.

Выгрузим наш модифицированный датафрейм в Excel. Для этого служит метод `.to_excel()`. Набор аргументов во многом пересекается с таковым метода `.to_csv()`. Есть возможность задать название листа (аргумент `sheet_name`).

```
# сохраним датасет в файл "iris_ru.xlsx" без индексов строк
iris_df.to_excel("iris_ru.xlsx", sheet_name="Ирисы", index=False)
```

Скачаем файл на компьютер и откроем его в Excel (рис. 99).

	A	B	C	D	E
1	**длина чашелистика**	**ширина чашелистика**	**длина лепестка**	**ширина лепестка**	**вид цветка**
2	5,1	3,5	1,4	0,2	ирис щетинистый
3	4,9	3	1,4	0,2	ирис щетинистый
4	4,7	3,2	1,3	0,2	ирис щетинистый
5	4,6	3,1	1,5	0,2	ирис щетинистый
6	5	3,6	1,4	0,2	ирис щетинистый

Рис. 99. Выгруженный из датафрейма датасет в файле Excel

Теперь попробуем загрузить Excel-файл из интернета. Для загрузки таких файлов служит функция `read_excel()`. Книга Excel может состоять из нескольких листов, поэтому можно указать название нужного нам (по умолчанию загружается первый лист):

```
read_excel(<адрес Excel-файла>, sheet_name=<название листа>)
```

В качестве примера данных сложного для загрузки формата используем сведения о среднемесячной номинальной начисленной заработной плате работников в целом по экономике Российской Федерации в 1991–2023 гг., опубликованные на сайте Росстата [22]. Файл доступен по прямой ссылке, которую можно найти на указанной странице, чем мы и воспользуемся. Можно скачать данный файл на компьютер и сначала открыть его в Excel, а затем загрузить в датафрейм с параметрами по умолчанию и «полюбоваться» результатом. Файлы со статистическими данными, опубликованные на сайте Росстата, обладают особенностями, не позволяющими с легкостью их загружать: они содержат объединенные ячейки, а также сноски с пояснениями. Попытаемся загрузить файл, задав несколько дополнительных параметров:

```
# загрузим данные из интернета по прямой ссылке
rosstat_df = pd.read_excel(
  "https://github.com/rvgarafutdinov/python_book_2023/raw/main/tab1-zpl_09-
2023.xlsx",
  sheet_name="Лист1",
  skiprows=7,  # пропустим первые 7 строк с заголовком и шапкой
```

```
  header=None  # заголовки столбцов расположены в объединенных ячейках
               # не будем их использовать
)
rosstat_df.iloc[[0, 1, 24, 25]]  # взглянем на результат (отберем 4 строки)
```

	0	1	2	3	4	5	6	7	8
0	1991	0.548	…	…	…	…	0.308	0.294	0.337
1	1992	6.000	2.1	4	6.3	12.1	1.400	2.000	2.700
24	2015(1)	34030.000	31566	34703	32983	36692	30929.000	31325.000	32642.000
25	2016	36709.000	34000	37404	35744	39824	32660.000	33873.000	35501.000

Глядя на фрагмент полученного датафрейма, можно отметить, что дополнительного преобразования требуют *как минимум* первый столбец (из-за сносок) и столбцы с третьего по шестой (в них присутствуют пропуски в виде многоточий). Исправлять все эти проблемы средствами pandas мы уже умеем (и в следующем параграфе решим эту задачу), но данный пример хорошо демонстрирует трудности, с которыми иногда приходится сталкиваться при загрузке датасетов из Excel.

13.4. Пример предварительной обработки датасета

Выполним необходимые преобразования, чтобы подготовить к анализу данные о зарплате из предыдущего параграфа (датафрейм `rosstat_df`).

Прежде всего удалим последние четыре строки со сносками:

```
rosstat_df = rosstat_df.iloc[:-4]  # удалим строки невключением их в отбор
rosstat_df.tail()  # посмотрим на последние 5 строк
```

	0	1	2	3	4	5	6	7	8
28	2019	47867.0	43944	48453	45726	51684	42263.0	43062.0	46324.0
29	2020	51344.0	48390	50784	49021	56044	46674.0	47257.0	50948.0
30	2021	57244.0	52143	57275	54133	62828	49516.0	51229.0	55208.0
31	2022	65338.0	60101	63784	61385	71377(2)	55717.0	57344.0	66757.0
32	2023(2)	NaN	66778	73534	70638.5	NaN	63260.0	65094.0	71334.0

Избавимся от пропусков в виде символа многоточия … (обратите внимание: это не три точки подряд, а единый символ).

```
rosstat_df.replace("…", np.nan, inplace=True)  # заменим многоточие на NaN
rosstat_df.head(1)  # убедимся, что замена прошла успешно
```

	0	1	2	3	4	5	6	7	8
0	1991	0.548	NaN	NaN	NaN	NaN	0.308	0.294	0.337

Если внимательно изучить данные, можно заметить, что скобки, указывающие на сноски, есть не только в первом столбце. Удалим скобки в конце всех значений датафрейма, для этого выполним следующие действия:

- ✓ преобразуем все значения в строки;
- ✓ применим к этим строковым значениям функцию, которая делит строку на подстроки по скобке, извлекает первую подстроку (все, что было до скобки) и преобразует ее в значение типа `float` (ведь в данных есть вещественные числа, а также `NaN`);
- ✓ полученный в результате датафрейм пересохраним в ту же переменную:

```
rosstat_df = rosstat_df.astype(str).applymap(lambda s: float(s.split("(")[0]))
```

Первый столбец содержит годы. Выглядит логичным: 1) сделать его целочисленным; 2) записать его значения в индексы строк. Осуществить последнее поможет метод `.set_index()`.

```
# сделаем первый столбец целочисленным
rosstat_df[0] = rosstat_df[0].astype(int)
# сделаем его индексным столбцом
rosstat_df.set_index(0, inplace=True)  # 0 - это название столбца с годами
# удалим название индексного столбца, оставшееся у него со времен,
rosstat_df.index.name = None  # когда он был столбцом обычным
rosstat_df.head(1)  # посмотрим на результат
```

	1	2	3	4	5	6	7	8
1991	0.548	NaN	NaN	NaN	NaN	0.308	0.294	0.337

Осталось задать информативные названия столбцов. Их можно проставить вручную, но используем более интересный способ: попытаемся извлечь их из исходного Excel-файла.

```
# еще раз загрузим датасет и сохраним первые строки в отдельный датафрейм
colnames = pd.read_excel(
  "https://github.com/rvgarafutdinov/python_book_2023/raw/main/tab1-zpl_09-2023.xlsx",
  skiprows=4, header=None,
).head(3)
colnames  # посмотрим, что получилось
```

	0	1	2	3	4	5	6	7	8
0	NaN	В среднем за год	В среднем за кварталы	NaN	NaN	NaN	Январь	Февраль	Март
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	I	II	III	IV	NaN	NaN	NaN

Из полученных записей можно извлечь корректные названия столбцов. Сделаем это следующим образом: в каждом столбце соединим вместе все его непустые значения.

```
# для упрощения кода напишем именованную функцию
def concat_values(ser):
  """Функция принимает на вход ряд и соединяет все его значения вместе,
  предварительно заменив пропуски на пустые строки"""
  return " ".join(ser.replace(np.nan, "")).strip()


# применим описанную функцию к столбцам датафрейма colnames
colnames = colnames.apply(concat_values, axis="rows")
colnames  # результат - ряд
```

```
0
1              В среднем за год
2     В среднем за кварталы  I
3                            II
4                           III
5                            IV
6                       Январь
...
```

Сформированный ряд обладает следующей особенностью: индексы его элементов соответствуют прежним названиям столбцов датафрейма (целые числа от нуля), а значениями являются сконструированные новые названия. Эта структура очень напоминает *словарь замен*, и в таком качестве данный ряд можно использовать.

```
# переименуем столбцы по словарю, в качестве словаря используем ряд
# (это сработает, ведь ряд похож на словарь с упорядоченными ключами)
rosstat_df.rename(columns=colnames, inplace=True)
rosstat_df.head(2)
```

	В среднем за год	В среднем за кварталы I	II	III	IV	Январь	Февраль	Март	Апрель	Май	Июнь	Июль
1991	0.548	NaN	NaN	NaN	NaN	0.308	0.294	0.337	0.373	0.438	0.493	0.541
1992	6.000	2.1	4.0	6.3	12.1	1.400	2.000	2.700	3.100	3.700	5.100	5.500

Напрашивается некоторая доработка полученного датафрейма (в частности, «для красоты» уместно отредактировать названия столбцов с квартальными данными). Это упражнение можно добавить к заданиям для самостоятельной работы.

13.5. Задания для самостоятельной работы

Решите следующие задачи:

1. Загрузите в датафрейм данные по объявлениям о продаже квартир по ссылке. Далее работайте с этим датафреймом.

2. Создайте новый столбец, содержащий значения столбца *plan*, кодированные числами по следующим правилам:

1) если значение содержит число, нужно использовать это число в качестве кода (например, «3-комнатная квартира» кодируется числом 3). Если несколько значений содержат одно и то же число, следует это число произвольно изменить, чтобы коды не повторялись;

2) числовые коды для остальных значений (не содержащих чисел) следует сгенерировать случайным образом, при этом сгенерированные коды не должны пересекаться с кодами, полученными из значений с числами;

3) одному и тому же значению должен соответствовать один и тот же код (например, если значение «Студия» получило код 99, то и все записи с таким значением должны иметь код 99).

3. Добавьте новую строку, ячейки которой заполните максимальными значениями по столбцам: самый большой этаж, квартира с наибольшим числом комнат, наибольшие цена и площадь. При этом максимальные значения должны быть аутентичными, то есть текстовыми, а не числовыми (например, «5-комнатная квартира», а не 5). Значение района (признака *address*) следует выбрать случайным образом из имеющихся. Значение добавленного столбца с кодом должно соответствовать найденному максимуму столбца *plan*, а не максимальному коду в новом столбце (например, если максимум – «5-комнатная квартира», то значение кода должно быть 5, даже если в столбце есть код 99).

4. Отберите только те строки, в которых площадь квартиры составляет от 50 до 60 м², стоимость составляет от 4 до 5 млн, а этаж является минимальным в указанной категории (то есть среди строк, отобранных по первым двум критериям).

5. Создайте новый датафрейм на базе исходного, отобрав из того любым способом такие строки (в количестве не менее 50), чтобы выборочное распределение столбца *square* имело следующие параметры: среднее от 47,3 до 47,6, стандартное отклонение от 13,3 до 13,6. Сам столбец *square* должен остаться оригинальным (текстовым, а не числовым).

14. Визуализация

Перед исследованием данных с помощью количественных методов полезно посмотреть на данные глазами, то есть изучить их графическое представление. Методом визуального анализа могут быть выявлены многие закономерности. В некоторых случаях зависимости в данных видны невооруженным глазом и не требуют подтверждения с помощью математических методов (например, статистических тестов, корреляционного или регрессионного анализа), в других случаях количественные методы могут подтвердить или опровергнуть предварительные выводы, сделанные «на глаз».

Существуют различные виды диаграмм, позволяющие представлять данные графически. Для их построения могут быть использованы разные Python-библиотеки. В завершающей главе пособия мы рассмотрим наиболее известную и популярную из них – Matplotlib. Также коснемся функций библиотеки seaborn, потому что некоторые стандартные диаграммы с их помощью построить намного проще.

14.1. Библиотека Matplotlib

Matplotlib – библиотека с открытым исходным кодом, являющаяся одним из самых популярных средств визуализации данных в Python. Она поддерживает работу с двумерной и трехмерной графикой, хорошо подходит для создания статичных, а также анимированных и интерактивных изображений. Название образовано, *вероятно* (автору не удалось найти достоверной информации), от словосочетания «**Mat**hematics **plot**ting **lib**rary» («библиотека для построения математических графиков»). Логотип библиотеки представлен на рис. 100, документация приведена по ссылке [9]. Актуальной (на момент издания пособия) версией Matplotlib является 3.8.2.

Рис. 100. Логотип Matplotlib

Сделаем небольшую, но важную ремарку: Matplotlib является очень мощным инструментом, обладающим широкими возможностями визуализации, однако и достаточно сложным в освоении. Автор видит своей задачей в рамках данного пособия дать читателю некоторую вводную по использованию библиотеки, достаточную для быстрого построения основных графиков, применяемых в анализе данных. Более детальное рассмотрение инструментов визуализации в Python можно найти, например, в книге [1].

Функции для построения диаграмм с помощью библиотеки содержатся в ее модуле pyplot, который подключают следующим образом:

```
# подключим библиотеку matplotlib
import matplotlib.pyplot as plt  # plt - общепринятый псевдоним модуля

import matplotlib  # подключим основной модуль, чтобы узнать версию
matplotlib.__version__  # установленная в Colab версия библиотеки
```

```
'3.7.1'
```

При построении диаграмм в Matplotlib они помещаются на ***холст*** (область, содержащую изображения), который создается автоматически при создании диаграммы. *Размерами холста* можно управлять с помощью изменения значения `"figure.figsize"` в словаре параметров визуализации:

```
plt.rcParams["figure.figsize"] = <ширина в дюймах>, <длина в дюймах>
```

Количество пикселов, которому соответствует заданная длина или ширина, можно вычислить, умножив его на величину DPI (от англ. *dots per inch* – точек на дюйм), которая по умолчанию составляет 100.

На холст можно поместить несколько диаграмм. Для построения новой диаграммы вызывается соответствующая функция. Вызов нескольких таких функций подряд добавит на холст соответствующие диаграммы. Очередностью расположения диаграмм на холсте можно управлять при помощи аргумента функций построения `zorder`. Это позволяет, например, поместить график с маркерами точек поверх соединяющей эти точки ломаной, которая является графиком иного типа и конструируется с помощью другой функции.

Рассмотрим некоторые функции, позволяющие настроить те или иные параметры холста. Примеры их использования приведем ниже, вместе с примерами построения диаграмм конкретных типов, а пока лишь опишем предназначение данных функций, чтобы впоследствии не акцентировать на нем внимание. Функция `grid()` включает отображение сетки. Команда

```
plt.rcParams["axes.axisbelow"] = True
```

располагает сетку и оси координат *за фигурами диаграмм*, позади них (по умолчанию сетка их перекрывает). Функции `xticks()`, `yticks()` позволяют вручную задать значения на осях координат x, y соответственно. Функции `xlim()`, `ylim()` устанавливают границы отображаемой области холста по x, y (сам холст с диаграммами может быть бо́льших размеров, но отображаться будет лишь указанная его часть). Функции `ylabel()`, `xlabel()` задают подписи осей. Функция `title()` задает заголовок всей диаграммы (композиции диаграмм) целиком.

Некоторые функции, устанавливающие параметры текстовых меток, имеют аргумент `size` для указания размера шрифта.

Функции, выполняющие построение диаграмм и задающие их параметры, возвращают специфические объекты, содержимое которых, если они сконструированы в последней строке ячейки Colab, выводится на экран в виде текстового представления объекта. Как правило, вывод этих данных является лишним. Хороший способ его подавить – добавление в конце последней строки ячейки точки с запятой (`;`). Также можно использовать функцию `show()`, отображающую холст со всеми диаграммами, которая ничего не возвращает и не создает лишнего вывода. В более традиционных средах разработки на Python, таких как PyCharm, в которых код программы содержится в модулях, а не в блокнотах, вызов `show()` *необходим* для отображения построенных диаграмм. Однако в Colab изображение выводится и без использования данной функции.

Рассмотрим построение диаграмм нескольких основных типов, применяемых в анализе данных. Как известно, данные могут быть *количественные* (например, заработная плата, рост человека или длина лепестка ириса) и *качественные*, или *категориальные* (марка автомобиля или цвет рубашки). Большая часть рассмотренных диаграмм предназначена для визуализации количественных данных, но работы с качественными тоже коснемся.

Диаграмма рассеяния (точечная диаграмма, scatter plot)

Этот вид диаграмм часто используется для отображения *зависимости между двумя величинами*. Каждому наблюдению (объекту выборки) соответствует точка, координаты которой равны значениям двух признаков этого наблюдения. Если предполагается, что один из признаков зависит от другого, значения *независимого* признака откладываются по оси x, а значения *зависимого* – по оси y. В простейшем случае по оси x откладываются *порядковые номера* объекта в выборке (объект 1, объект 2, объект 3; эти данные не несут содержательной нагрузки, являются лишь некоторыми метками), по оси y – значения признака объектов. Диаграмма рассеяния может служить визуальным методом *выявления выбросов*: таковыми можно считать точки, далеко расположенные от основной массы точек диаграммы (рис. 101).

Для построения диаграммы рассеяния используется функция `scatter()`:

```
plt.scatter(x=<значения x-координат>, y=<значения y-координат>)
```

Значения координат точек задаются *индексированными коллекциями* или вещественным числом (если точка одна).

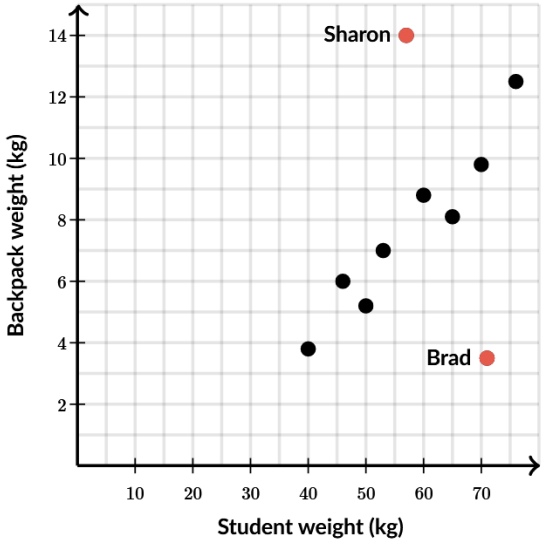


Рис. 101. Выбросы на диаграмме рассеяния
Источник: https://www.khanacademy.org

Попробуем построить диаграмму, отображающую зависимость веса рюкзака студента от веса самого студента, как на рис. 101. Для этого подготовим данные в виде датафрейма:

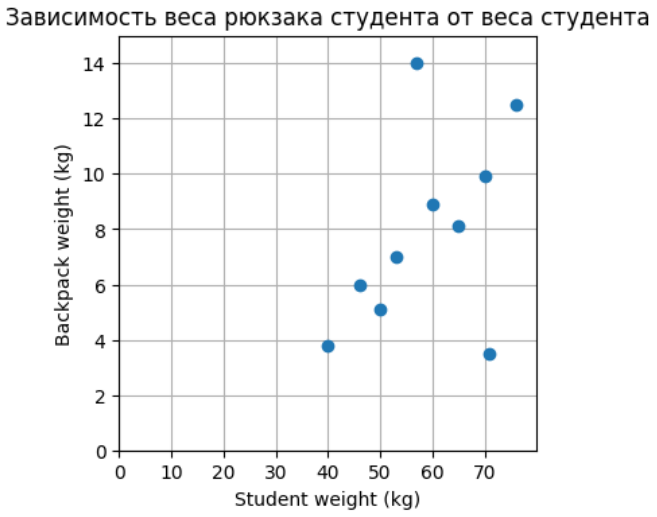
```
import pandas as pd  # подключим pandas

backpacks_df = pd.DataFrame(
  {
    # веса студентов
    "student_weight": [40, 46, 50, 53, 57, 60, 65, 70, 71, 76],
    # веса рюкзаков
    "backpack_weight": [3.8, 6, 5.1, 7, 14, 8.9, 8.1, 9.9, 3.5, 12.5]
  },
  index=["Анна", "Алена", "Ксения", "Ольга", "Sharon",  # имена студентов
         "Антон", "Петр", "John", "Brad", "Сергей"]
)
backpacks_df.head(3)
```

	student_weight	backpack_weight
Анна	40	3.8
Алена	46	6.0
Ксения	50	5.1

Выполним построение диаграммы на основе подготовленных данных:

```
plt.rcParams["figure.figsize"] = 4, 4  # установим размеры холста

# добавим на холст диаграмму рассеяния
plt.scatter(
  x=backpacks_df.student_weight,  # веса студентов
  y=backpacks_df.backpack_weight,  # веса рюкзаков
)
```

```
# установим положение осей и сетки позади фигур диаграмм (если с первого раза
plt.rcParams["axes.axisbelow"] = True  # не сработало, выполните код повторно)
plt.grid()  # включим отображение сетки

plt.xticks(range(0, 80, 10))  # зададим значения на оси x
plt.yticks(range(0, 15, 2))  # зададим значения на оси y

plt.xlim(0, 80)  # ограничим диапазон оси x
plt.ylim(0, 15)  # ограничим диапазон оси y

plt.xlabel("Student weight (kg)")  # установим подпись оси x
plt.ylabel("Backpack weight (kg)")  # установим подпись оси y

# добавим название всей диаграммы и не забудем подавить вывод объекта (;)
plt.title("Зависимость веса рюкзака студента от веса студента");
```

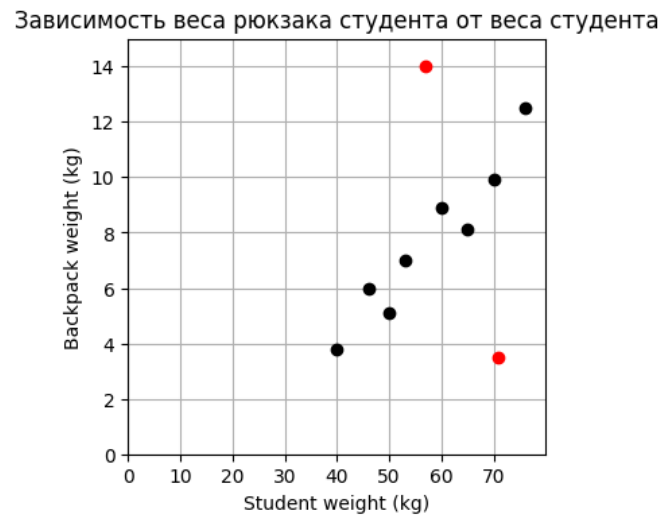


Цвет диаграммы устанавливается автоматически, однако его можно задать явно при помощи аргументов функции построения `color` и `c`. Текстовые названия тех или иных цветов приведены по ссылке [6].

Чтобы достичь еще большего сходства нашей диаграммы с изображенной на рис. 101, изменим ее цвета. Основная масса точек – черные, но объекты-выбросы (Sharon и Brad) должны быть красными. Рассмотрим *два способа* раскрасить точки разными цветами. Первый из них: поместить на холст не одну диаграмму с точками, а *две* – с обычными точками черного цвета и с точками-выбросами красного цвета.

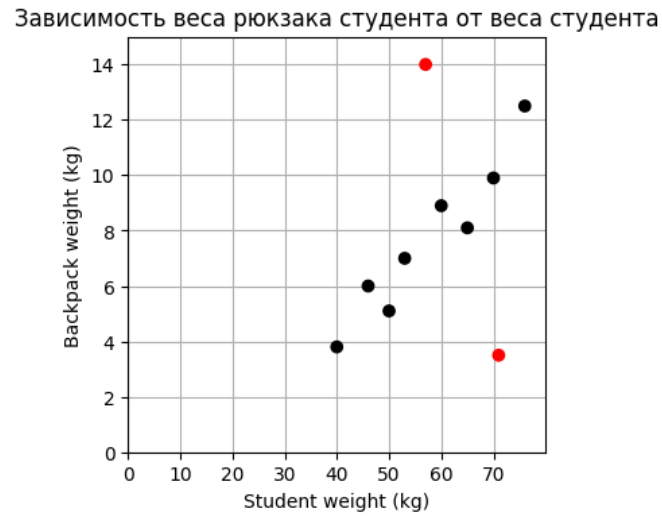
```
# сформируем битовую карту объектов-выбросов
# (строковый индекс поддерживает строковые методы)
outliers = backpacks_df.index.str.contains("Sharon|Brad")

# обычные точки
plt.scatter(
  x=backpacks_df[~outliers].student_weight,
  y=backpacks_df[~outliers].backpack_weight,
  color="black"  # цвет черный
)
```

```
# точки-выбросы
plt.scatter(
  x=backpacks_df[outliers].student_weight,
  y=backpacks_df[outliers].backpack_weight,
  color="red"  # цвет красный
)

# остальной код ячейки не изменился, не будем его приводить
```



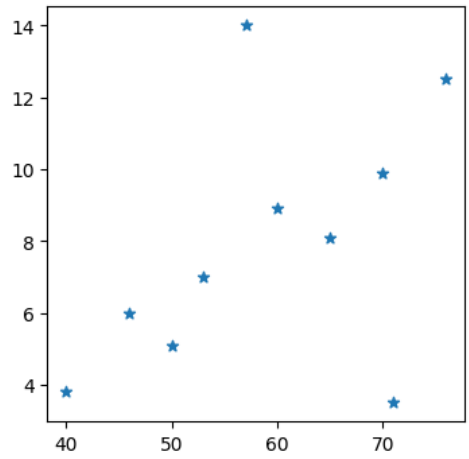
Второй способ: применить аргумент функции `c`, позволяющий задавать цвета точек в виде индексированной коллекции. Элементами этой коллекции могут быть названия цветов, а также их числовые коды.

```
plt.scatter(
  x=backpacks_df.student_weight,
  y=backpacks_df.backpack_weight,
  # в битовой карте выбросы обозначены True. Заменим True/False
  # на названия цветов, но так как метода replace() у numpy-массива нет,
  # преобразуем массив в ряд
  c=pd.Series(outliers).replace({False: "black", True: "red"})
)

# остальной код ячейки не изменился, не будем его приводить
```



Форму точек можно задавать с помощью аргумента `marker`. О допустимых формах точек и их обозначениях можно почитать по ссылке [7].

```
plt.scatter(
  x=backpacks_df.student_weight,
  y=backpacks_df.backpack_weight,
  marker="*",  # точка в виде звездочки
);
```



Линейный график (line chart)

Диаграмма данного типа представляет собой ломаную линию, соединяющую точки (сами точки при этом не отображаются). Линейный график часто используется для визуализации временных рядов, а также в качестве *полигона (многоугольника) частот*. В последнем случае по оси x располагаются значения показателя, а по оси y – их частоты. Для построения такой диаграммы используется функция `plot()`:

```
plt.plot(<значения x-координат>, <значения y-координат>)
```

Координаты задаются индексированными коллекциями в виде *позиционных* аргументов, при этом x-координаты могут не указываться, в качестве них по умолчанию используются значения от 0 до $n-1$, где n – количество точек.

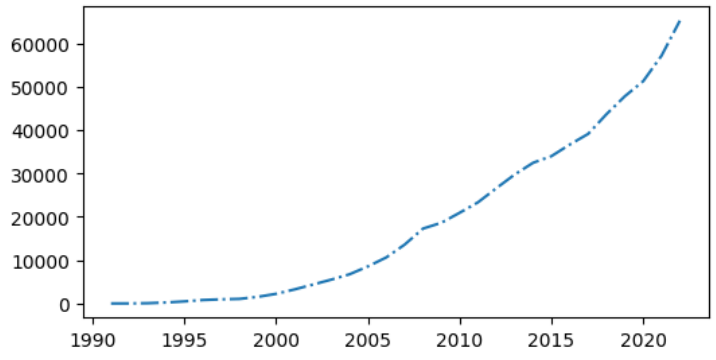
Построим линейный график, отображающий динамику средней зарплаты в России по годам (датафрейм `rosstat_df` был подготовлен в параграфе 13.4).

```
plt.rcParams["figure.figsize"] = 6, 3  # изменим размеры холста

plt.plot(
  rosstat_df.index,  # значения по оси x
  rosstat_df["В среднем за год"],  # значения по оси y
)
# в данном случае значения y - ряд (Series), у ряда есть индексы,
# можно было значения x не указывать, были бы использованы индексы ряда
```

```
plt.xlabel("Год")  # подпись оси x
plt.ylabel("Зарплата, руб.")  # подпись оси y
plt.title("Динамика средней зарплаты по годам");
```

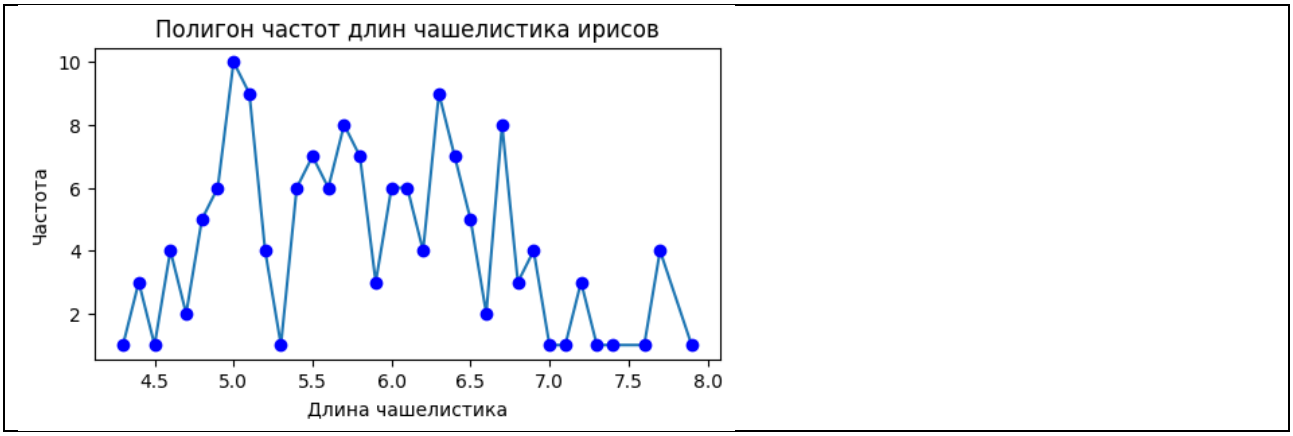
Стиль линии можно задавать с помощью аргумента `linestyle`. Допустимые стили и их обозначения описаны по ссылкам [5; 10].

```
plt.plot(
  rosstat_df["В среднем за год"],
  linestyle="-."  # кривая в виде пунктира с точкой
);
```



Также построим полигон частот длин чашелистика цветка из датасета «Ирисы Фишера» (датафрейм `iris_df` был загружен в параграфе 13.3). Для этого сконструируем частотную таблицу длин чашелистика ирисов, упорядоченную по значению длины. Если не упорядочить, точки будут соединяться в неправильном порядке (не по возрастанию), и график получится некорректный.

```
# сохраним упорядоченную частотную таблицу в отдельную переменную (ряд)
freq_table = iris_df["sepal length"].value_counts().sort_index()
# построим линейный график (x - индексы, y - значения)
plt.plot(freq_table, zorder=1)  # положение на холсте - снизу
# добавим точки (диаграмму рассеяния), положение на холсте - сверху
plt.scatter(x=freq_table.index, y=freq_table, color="blue", zorder=2)

plt.xlabel("Длина чашелистика")  # подпись оси x
plt.ylabel("Частота")  # подпись оси y
plt.title("Полигон частот длин чашелистика ирисов");
```



Полигон частот используется для *визуализации выборочного распределения данных*. Например, по графику из примера выше заметно, что чаще всего (10 раз) в датасете встречаются ирисы с длиной чашелистика 5.

Столбчатая диаграмма (bar chart)

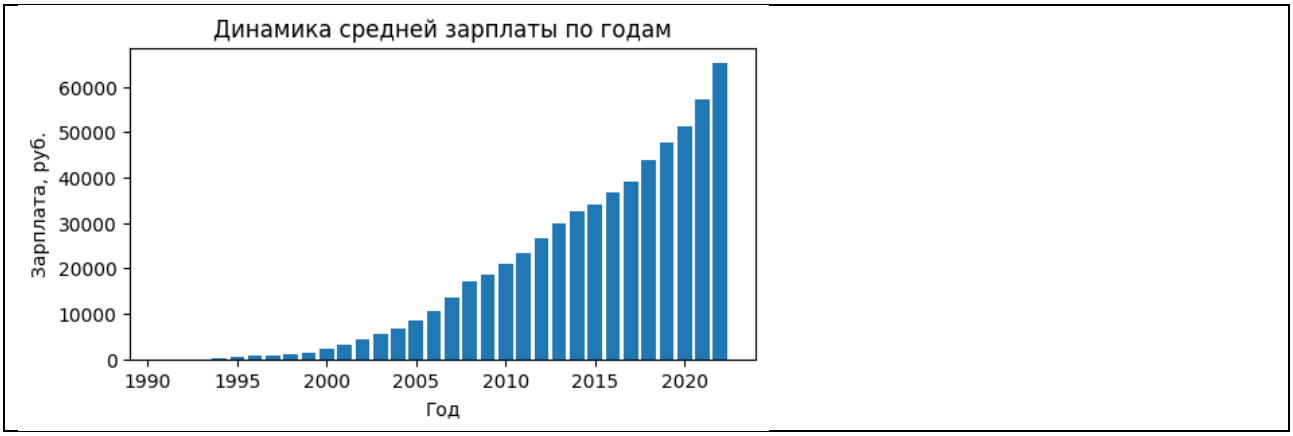
Столбчатая диаграмма по смыслу отображаемой информации похожа на диаграмму рассеяния, только объекты вместо точек представлены столбцами определенной высоты (длины). Столбцы располагаются вертикально или горизонтально. Этот вид диаграмм обычно используется для сравнения (сопоставления) нескольких величин. Иногда на столбчатые диаграммы помещают по несколько значений для каждой из сравниваемых категорий. Для построения используются функции `bar()` и `barh()` с однотипным набором аргументов.

```
plt.bar(x=<значения по оси x>, height=<высоты столбцов>,
                               width=<ширина (ширины) столбцов>)
```

Значения по оси x и значения высоты столбцов задаются индексированными коллекциями. Высоту столбцов можно задать *вещественным числом* (если нужны столбцы равной высоты). В свою очередь, ширину столбцов можно задать *коллекцией* (если нужны столбцы разной ширины).

Построим вертикальную столбчатую диаграмму, отображающую динамику средней зарплаты:

```
# вертикальная столбчатая диаграмма, функция bar()
plt.bar(
  x=rosstat_df.index,  # значения по оси x
  height=rosstat_df["В среднем за год"],  # высоты столбцов
  width=0.8  # ширина столбца
)

plt.xlabel("Год")  # подпись оси x
plt.ylabel("Зарплата, руб.")  # подпись оси y
plt.title("Динамика средней зарплаты по годам");
```



Поработаем с категориальными данными. Построим горизонтальную столбчатую диаграмму долей брендов смартфонов на российском рынке [24].

```
# данные о долях брендов смартфонов на российском рынке
smartphones_df = pd.DataFrame({
  'Realme': [8, 21, 20],
  'Honor': [4, 4, 19.4],
  'Xiaomi': [29, 24, 19],
  'Tecno': [0, 11, 17],
  'Samsung': [39, 24, 12],
  'Apple': [12, 10, 8],
  'ZTE': [6, 4.3, 1]
}, index=[2021, 2022, 2023])

#сумма по годам не равна 100%, недостающие проценты добавим в столбец "Прочие"
smartphones_df["Прочие"] = 100 - smartphones_df.apply(sum, axis=1)
smartphones_df
```

	Realme	Honor	Xiaomi	Tecno	Samsung	Apple	ZTE	Прочие
2021	8	4.0	29	0	39	12	6.0	2.0
2022	21	4.0	24	11	24	10	4.3	1.7
2023	20	19.4	19	17	12	8	1.0	3.6

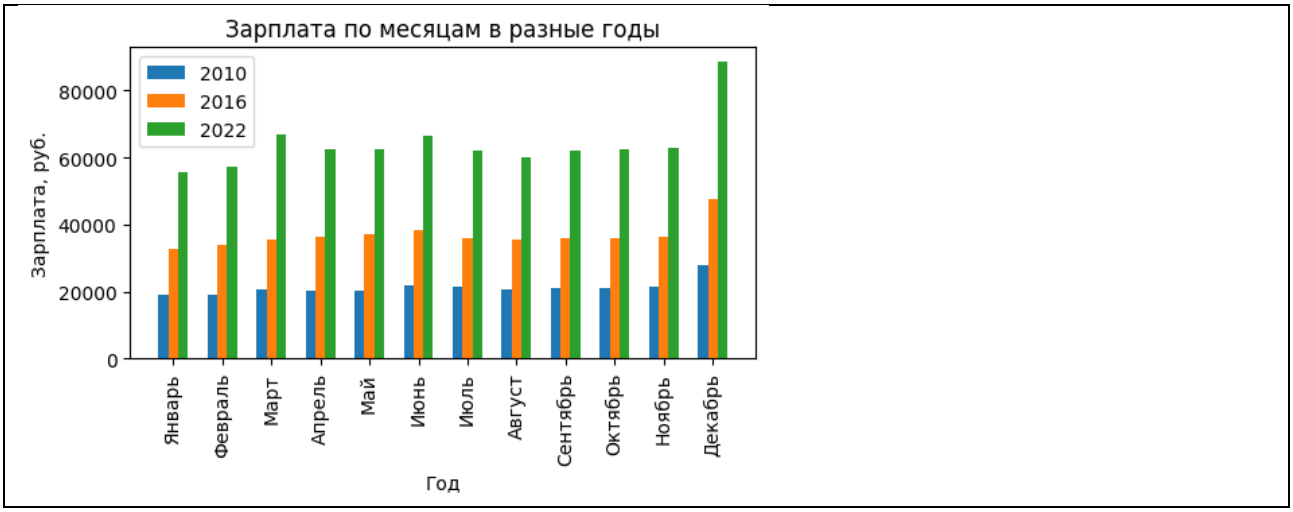
Функция `barh()` работает аналогично предыдущей, но названия ее аргументов несколько отличаются. Обратите внимание, что для добавления меток столбцов на диаграмму нам пришлось сохранить возвращаемый функцией построения объект в переменную и воспользоваться функцией `bar_label()`.

```
plt.rcParams["figure.figsize"] = 5, 4  # изменим размеры холста

# возьмем данные 2023 года (ряд), упорядочим бренды по долям
top2023_df = smartphones_df.loc[2023].sort_values()

# горизонтальная столбчатая диаграмма, функция barh()
bar = plt.barh(
  y=top2023_df.index,  # значения по оси y (названия брендов)
  width=top2023_df,  # длины столбцов (доли)
  height=0.8,  # ширина (толщина) столбца
  color="lightseagreen"  # цвет - оттенок зеленого
)
```

```
# добавим рядом со столбцами их метки (доли) в отформатированном виде
plt.bar_label(bar, labels=top2023_df.apply(lambda s: f"{s:.1f}%"))
# если не расширить область построения, метки в нее не войдут
plt.xlim(0, top2023_df.max() + 3)
plt.title("Доли брендов смартфонов на российском рынке в 2023 году");
```

Построим еще одну диаграмму, отображающую средние зарплаты за три разных года по месяцам. С помощью функции `bar()` это можно сделать следующим образом: поместить на холст три диаграммы с нужными данными, каждую последующую при этом смещать вправо по оси x. Для этого в качестве значений по x будем использовать не метки месяцев, а массив позиций столбцов в дюймах, названия месяцев же нанесем после добавления всех диаграмм.

```
plt.rcParams["figure.figsize"] = 6, 3  # изменим размеры холста

# для добавления диаграммы на холст напишем функцию
def add_bar(x_data, heights, label, width=0.2):
  """Функция для построения столбчатой диаграммы с заданными параметрами"""
  plt.bar(
    x=x_data,  # значения по оси x
    height=heights,  # высоты столбцов
    width=width,  # ширина столбца
    label=label  # метка диаграммы
  )

# добавим три диаграммы, значения зарплат в нужном году извлечем из датафрейма
add_bar(np.arange(0.8, 12), rosstat_df.loc[2010].iloc[5:], "2010")
# столбцы второй диаграммы смещены вправо на ширину одного столбца
add_bar(np.arange(1, 13), rosstat_df.loc[2016].iloc[5:], "2016")
# столбцы третьей диаграммы смещены вправо на ширину двух столбцов
add_bar(np.arange(1.2, 13), rosstat_df.loc[2022].iloc[5:], "2022")

# нанесем на ось x метки названия месяцев с помощью аргумента labels,
# но текстовые метки без самих значений задать нельзя, поэтому используем
# позиции столбцов 2016 года. Также укажем, что метки надо расположить
# вертикально (иначе соседние метки наложатся друг на друга)
plt.xticks(np.arange(1, 13), labels=rosstat_df.columns[5:],
           rotation='vertical')
plt.xlabel("Год")
plt.ylabel("Зарплата, руб.")
plt.title("Зарплата по месяцам в разные годы")
plt.gca().legend();  # включим отображение легенды
```

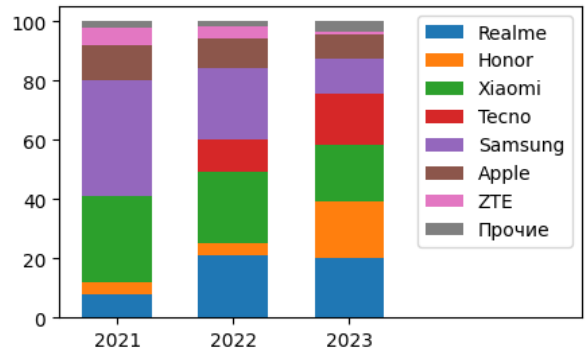


Можно заметить, что для формирования и отображения так называемой легенды (области с условными обозначениями различных данных на диаграммах) мы использовали аргумент функции построения `label` (поддерживается функциями всех диаграмм Matplotlib), а также метод `.legend()` объекта `Axes`, возвращаемого функцией `gca()`. Разбирать в подробностях работу с этим объектом не будем, просто запомним, что легенду можно включать таким способом.

Построим еще одну разновидность столбчатой диаграммы, в которой каждый столбец включает в себя распределение данных по категориям, – *столбчатую диаграмму с накоплением* (stacked bar chart). Вновь обратимся к долям брендов смартфонов на российском рынке (категориальные данные) за три года. У функции `bar()` есть аргумент `bottom`, с помощью которого можно задать значения высоты (значения по оси y), на которых будут располагаться основания столбцов, то есть столбцы можно «приподнять» над осью x. Это позволяет формировать составные столбцы, при каждом вызове функции построения помещая на холст фрагмент столбца с долей очередной категории на высоте, равной накопленной величине долей предыдущих категорий. Аналогичный по принципу действия аргумент функции `barh()` называется `left`, он позволяет смещать столбцы вправо от оси y.

```
plt.rcParams["figure.figsize"] = 5, 3  # изменим размеры холста

# поместим на диаграмму данные первого бренда
plt.bar(
  x=smartphones_df.index,  # значения по оси x (годы)
  height=smartphones_df.iloc[:, 0],  # высоты столбцов (доли первого бренда)
  width=0.6,  # ширина столбца
  label=smartphones_df.columns[0]  # метка диаграммы (название бренда)
)
```

```
# в цикле поместим на диаграмму данные остальных брендов,
# "приподнимая" столбцы очередного бренда
for i in range(1, len(smartphones_df.columns)):  # начнем со второго
  plt.bar(
    x=smartphones_df.index,  # значения по оси x (годы)
    height=smartphones_df.iloc[:, i],  # высоты столбцов (доли i-го бренда)
    width=0.6,  # ширина столбца
    # высоты, на которых расположены основания столбцов
    # (равны накопленным высотам столбцов предыдущих брендов)
    bottom=smartphones_df.iloc[:, :i].sum(axis=1),
    label=smartphones_df.columns[i]  # метка диаграммы (название бренда)
  )

plt.xticks(smartphones_df.index)  # метки на оси x (годы)
# расширим область построения, чтобы оставить место для легенды
plt.xlim(2020.5, 2025)
plt.gca().legend();  # отобразим легенду
```



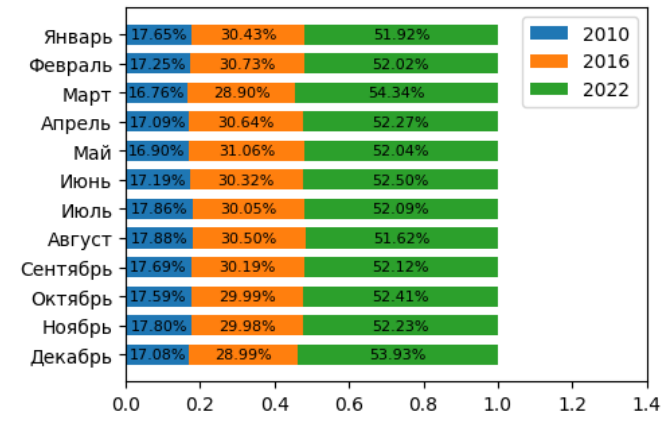
В данном случае диаграмма с накоплением получилась *нормированной* в силу самих данных: совокупная доля брендов в датафрейме в каждом году составляет 100 %. Представим в аналогичном виде с накоплением помесячные данные о зарплатах с распределением по годам. Чтобы столбцы оказались нормированы (были равной длины), выполним нормализацию данных вручную.

```
# поместим нужные данные в отдельный датафрейм и нормализуем
df_norm = rosstat_df.loc[[2010, 2016, 2022]].iloc[:, -12:]
# для нормализации разделим значение в каждой ячейке на сумму по столбцу
df_norm = df_norm / df_norm.sum()
df_norm
```

	Январь	Февраль	Март	Апрель	Май	Июнь	Июль	Август
2010	0.176471	0.172515	0.167599	0.170898	0.168983	0.171866	0.178582	0.178805
2016	0.304338	0.307283	0.288985	0.306378	0.310568	0.303176	0.300537	0.305045
2022	0.519191	0.520202	0.543416	0.522724	0.520449	0.524958	0.520881	0.516150

```
plt.rcParams["figure.figsize"] = 5, 3.6  # изменим размеры холста

# для добавления диаграммы на холст напишем функцию
def add_stacked_barh(y_data, widths, left, label, height=0.7):
  """Функция для построения столбчатой диаграммы с накоплением
  с заданными параметрами"""
  barh = plt.barh(
    y=y_data,  # значения по оси y (месяцы)
    width=widths,  # ширины столбцов (доли текущего года)
```

```
    # отступы слева, с которых начинаются столбцы
    left=left,
    height=height,  # высота столбца
    label=label  # метка диаграммы (год)
  )
  # добавим метки (доли) сегментов столбцов
  plt.bar_label(
    barh,
    labels=widths.apply(lambda s: f"{s:.2%}"),  # в отформатированном виде
    label_type="center",  # расположение меток - внутри сегментов столбцов
    size=8 # укажем размер шрифта метки (по умолчанию не помещается в сегмент)
  )

months = df_norm.columns  # названия месяцев

# добавим три диаграммы, первая - 2010 год
add_stacked_barh(months, df_norm.loc[2010], 0, "2010")
# вторая - 2016 год, столбцы смещены вправо на величину 2010 года
add_stacked_barh(months, df_norm.loc[2016], df_norm.loc[2010], "2016")
# третья - 2022 год, столбцы смещены вправо на сумму 2010 и 2016 гг.
add_stacked_barh(months, df_norm.loc[2022],
                 df_norm.loc[2010] + df_norm.loc[2016], "2022")

# расширим область построения, чтобы оставить место для легенды
plt.xlim(0, 1.4)
plt.gca().invert_yaxis()  # обратим порядок расположения месяцев на диаграмме
plt.gca().legend();  # отобразим легенду
```



Гистограмма частот (histogram)

Диаграмма данного вида очень похожа на столбчатую диаграмму, однако она используется не для сравнения величин, а для визуализации выборочного распределения данных (как и полигон частот). В отличие от полигона, на гистограмме по оси x располагаются не конкретные значения показателя, а *диапазоны его значений (интервалы)*. Следовательно, гистограмма является графическим изображением *интервального ряда* (см. `Series`-метод `.value_counts()` с аргументом `bins`, параграф 13.1). На оси x располагаются границы интервалов, на которые разбит размах значений показателя, на оси y – частоты, с которыми конкретные значения показателя попадают в тот или иной интервал. Гистограмма состоит из столбцов шириной, соответствующей ширине интервала, и высотой,

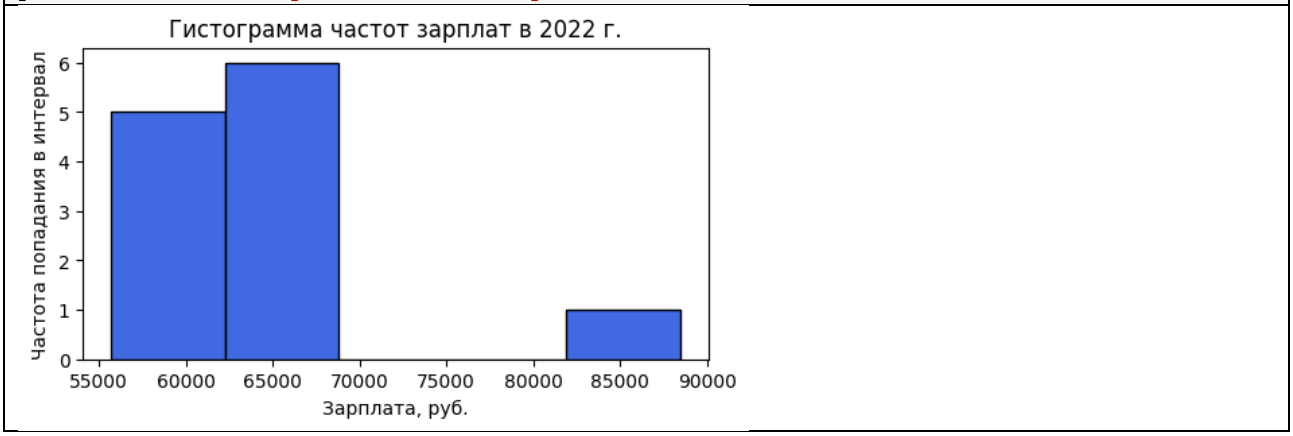
соответствующей его частоте. Подобную диаграмму, отражающую распределение данных, можно построить и с помощью функции `bar()`, однако для построения гистограмм существует специальная функция `hist()`, которая вычисляет границы интервалов и частоты автоматически, поэтому значительно удобнее в использовании:

```
plt.hist(x=<значения показателя>, bins=<количество или границы интервалов>)
```

Значения показателя могут быть заданы коллекцией, в том числе неиндексированной (порядок элементов не важен). Интервалы можно задать либо перечислением их границ в виде индексированной коллекции, либо их количеством, в последнем случае границы вычисляются автоматически.

Построим гистограмму, отображающую распределение средней зарплаты в разные месяцы 2022 г.:

```
plt.rcParams["figure.figsize"] = 6, 3  # изменим размеры холста

plt.hist(
  rosstat_df.loc[2022].iloc[5:],  # данные
  bins=5,  # количество интервалов (столбцов)
  color="royalblue",  # цвет столбца
  linewidth=1,  # толщина границ столбца
  edgecolor="black"  # цвет границ столбца
)

plt.xlabel("Зарплата, руб.")
plt.ylabel("Частота попадания в интервал")
plt.title("Гистограмма частот зарплат в 2022 г.");
```



По диаграмме можно сделать вывод, что в течение одиннадцати месяцев из двенадцати величина средней зарплаты попадала в диапазон примерно от 56 до 68 тыс. руб., а в одном месяце (очевидно, декабре, когда на предприятиях выплачивают годовые премии) она составила примерно от 82 до 88 тыс. руб. Точнее по этой гистограмме сказать нельзя, однако можно воспользоваться функцией `xticks()` для ручного задания отображаемых на оси x значений, в качестве

которых можно взять вычисленные границы интервалов. С вопросом, как получить эти значения, автор предлагает читателю разобраться самостоятельно, ибо не сомневается, что для того, кто дошел до текущего параграфа, решение указанной задачи трудностей не вызовет.

Круговая диаграмма (pie chart)

Диаграмма данного вида демонстрирует, какую долю та или иная категория занимает в составе целого. Ее англоязычное название отсылает к разрезанному на куски круглому пирогу. Эта диаграмма по смыслу отображаемой информации похожа на один столбец столбчатой диаграммы с накоплением, она позволяет визуально оценить распределение категориальных данных. Для построения круговой диаграммы используется функция `pie()`:

```
plt.pie(<доли секторов>, labels=<метки секторов>, colors=<цвета секторов>)
```

Доли задаются индексированной коллекцией числовых значений, каждое из которых, нормированное их суммой, обозначает долю, занимаемую сектором круга. Например, если доли заданы списком `[0.4, 0.2, 0.5]`, то первый сектор круга займет 40% общей площади, второй – 20 %, третий – 50 %. Если доли заданы списком `[10, 20, 20]`, то первый сектор займет 20 % (именно столько составляет процент значения 10 в общей сумме 50), оставшиеся – по 40 % каждый. Метки задаются коллекцией значений, которыми будут подписаны секторы круга, перечисленные в том же порядке, что и доли. Цвета секторов задаются коллекцией названий цветов.

Построим круговую диаграмму долей ирисов того или иного вида. В оригинальном наборе данных присутствует по 50 записей об ирисах каждого вида, поэтому, чтобы распределение не было таким безупречно равномерным, сформируем случайную выборку из 100 элементов, по ней и будем строить диаграмму:

```
# сформируем частотную таблицу видов ирисов (ряд)
freq_table = iris_df["class"].sample(100, random_state=42).value_counts()

# построим круговую диаграмму
plt.pie(
  freq_table,  # доли секторов (частоты видов ирисов)
  labels=freq_table.index,  # метки секторов
  colors=["dodgerblue", "deepskyblue", "cornflowerblue"],  # цвета секторов
  autopct="%.1f%%",  # формат метки (один дробный разряд и знак процента)
  explode=[0.1, 0, 0] # расстояния секторов от центра круга (для их выделения)
)
plt.title("Доли ирисов разных видов в наборе данных");
```

Ящик с усами (box plot)

Такой вид диаграммы в удобной форме отображает медиану (или среднее), нижний и верхний квартили, минимальное и максимальное значения выборки, а также *выбросы*, выявленные по критерию *межквартильного размаха*. Расстояния между различными частями ящика позволяют определить степень разброса (дисперсии) и асимметрии данных и выявить выбросы. Часто несколько таких ящиков, построенных на различных выборках, помещают рядом, чтобы визуально сравнивать распределения этих выборок. Ящики можно располагать как горизонтально, так и вертикально. Для построения ящика с усами предназначена функция `boxplot()`:

```
plt. boxplot(<набор или коллекция наборов значений>)
```

Анализируемая выборка может быть задана коллекцией, в том числе неиндексированной. Если необходимо поместить на диаграмму ящики, построенные по нескольким выборкам, тогда нужно передать эти выборки как элементы списка (или другой индексированной коллекции).

Построим ящики с усами для средней зарплаты по месяцам в 2020–2022 гг.:

```
plt.boxplot(
  [
    rosstat_df.loc[2020].iloc[5:],  # зарплата в 2020 г.
    rosstat_df.loc[2021].iloc[5:],  # зарплата в 2021 г.
    rosstat_df.loc[2022].iloc[5:]  # зарплата в 2022 г.
  ],
  labels=[2020, 2021, 2022]  # метки ящиков на оси x
)

plt.xlabel("Год")
plt.ylabel("Зарплата, руб.")
plt.title("Распределение средней зарплаты в виде ящиков с усами");
```

Значения выборки, оказывающиеся за пределами «усов» (линий – границ ящика), отображаются окружностями малого диаметра и считаются выбросами. В нашем примере явными выбросами (аномально большими значениями) являются величины зарплаты в декабре.

14.2. Библиотека seaborn

Seaborn является высокоуровневой надстройкой над Matplotlib (подобно тому как pandas является надстройкой над NumPy). Логотип библиотеки представлен на рис. 102, документация доступна по ссылке [17]. Актуальной (на момент издания пособия) версией seaborn является 0.13.0.

Рис. 102. Логотип seaborn

Стоит отметить, что параметры холста при использовании функций seaborn настраиваются так же, как при использовании Matplotlib.

```
# подключим библиотеку seaborn
import seaborn as sns  # sns - общепринятый псевдоним

sns.__version__  # установленная в Colab версия библиотеки
```

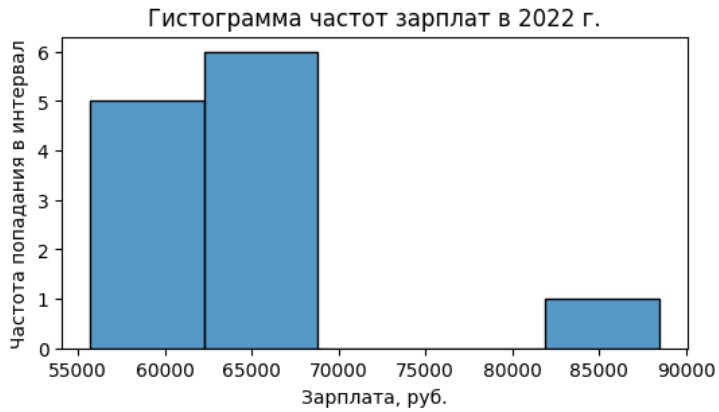
```
'0.12.2'
```

Рассмотрим несколько функций seaborn для построения диаграмм.

Гистограмма частот (histogram)

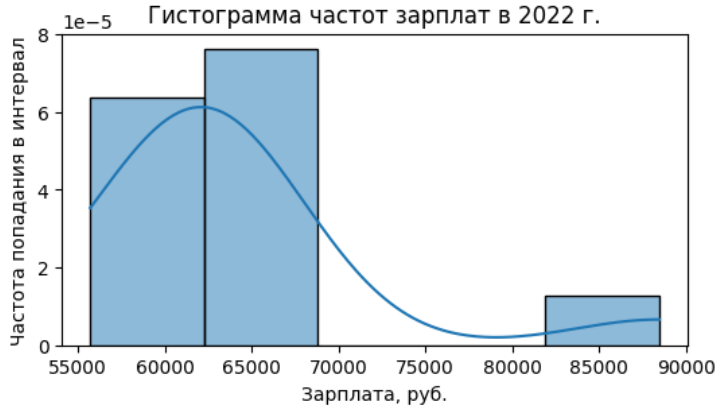
В seaborn гистограмму можно построить с помощью функции `histplot()`. Построим гистограмму частот зарплат, аналогичную построенной ранее с помощью `plt.hist()`:

```
# гистограмма, 5 интервалов, цвета стандартные
sns.histplot(rosstat_df.loc[2022].iloc[5:], bins=5)

plt.xlabel("Зарплата, руб.")
plt.ylabel("Частота попадания в интервал")
plt.title("Гистограмма частот зарплат в 2022 г.");
```



Данная функция позволяет также наложить на гистограмму *кривую оценки плотности вероятности случайной величины* (**KDE**, **K**ernel **D**ensity **E**stimation – «ядерная оценка плотности»). В прикладном анализе данных нередко строят такие совмещенные диаграммы при визуальном изучении распределения выборки. Построить гистограмму с наложенной кривой KDE можно и только средствами Matplotlib, но это более трудоемко.

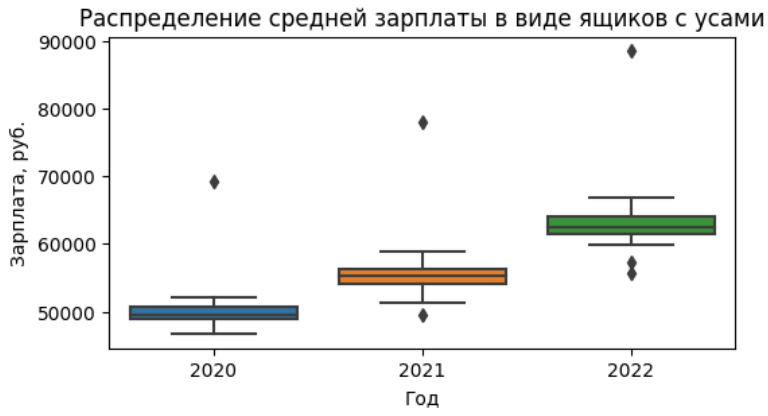
```
# гистограмма, 5 интервалов, цвета стандартные
sns.histplot(
  rosstat_df.loc[2022].iloc[5:], bins=5,
  kde=True, stat="density"  # включим отображение кривой оценки плотности
)

# остальной код ячейки не изменился, не будем его приводить
```



Ящик с усами (box plot)

Данная диаграмма может быть построена с помощью функции с таким же, как в Matplotlib, названием `boxplot()`:

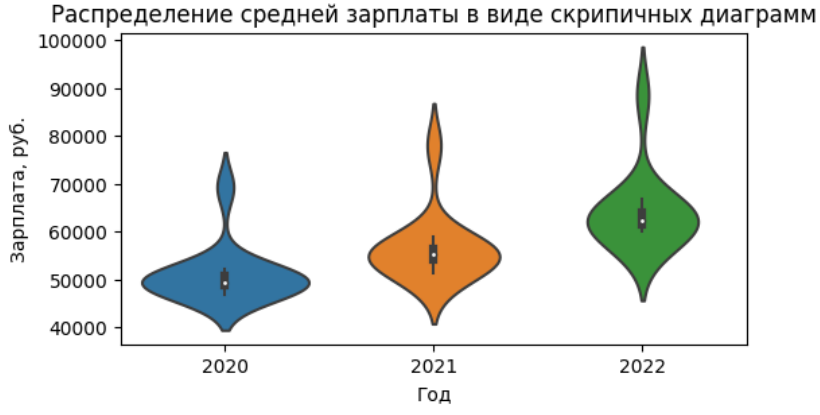
```
sns.boxplot([
  rosstat_df.loc[2020].iloc[5:],  # зарплата в 2020 г.
  rosstat_df.loc[2021].iloc[5:],  # зарплата в 2021 г.
  rosstat_df.loc[2022].iloc[5:]  # зарплата в 2022 г.
])

plt.xticks(range(3), labels=[2020, 2021, 2022])  # метки ящиков на оси x
plt.xlabel("Год")
plt.ylabel("Зарплата, руб.")
plt.title("Распределение средней зарплаты в виде ящиков с усами");
```



Скрипичная диаграмма (violin plot)

Эта диаграмма представляет собой ящик с усами и наложенные на него с обеих сторон зеркально отраженные кривые KDE. Однако, в отличие от ящика, скрипичная диаграмма *не отображает выбросы*, поэтому ее используют для визуализации и сравнения распределений выборок данных, но не для выявления аномальных значений. Для построения такой диаграммы служит функция `violinplot()`. Набор основных аргументов аналогичен таковому для `boxplot()`.

```
sns.violinplot([
  rosstat_df.loc[2020].iloc[5:],  # зарплата в 2020 г.
  rosstat_df.loc[2021].iloc[5:],  # зарплата в 2021 г.
  rosstat_df.loc[2022].iloc[5:]  # зарплата в 2022 г.
])

plt.xticks(range(3), labels=[2020, 2021, 2022])  # метки на оси x
plt.xlabel("Год")
plt.ylabel("Зарплата, руб.")
plt.title("Распределение средней зарплаты в виде скрипичных диаграмм");
```



Тепловая карта (heat map)

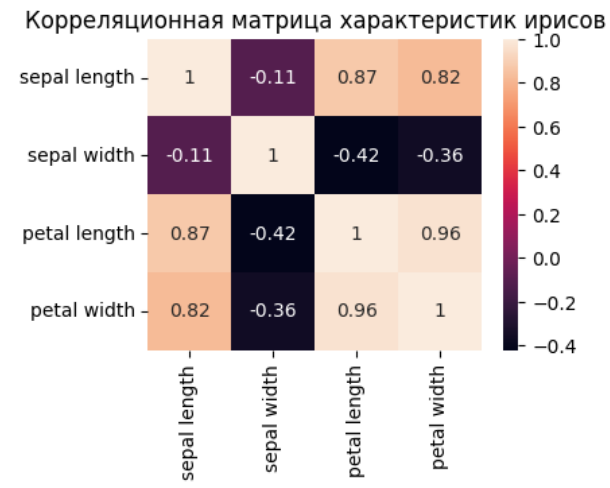
Эта диаграмма предназначена для визуализации таблиц (матриц). Она отображает значения в ячейках таблицы при помощи *цветовой индикации*. Крайним значениям показателя (максимальному и минимальному) в наборе данных поставлены в соответствие определенные цвета, и чем ближе значение в ячейке к крайнему, тем ближе его цвет к цвету этого крайнего значения. Такой способ отображения удобен при визуальном анализе индивидуальных числовых значений в больших таблицах (примером может служить матрица корреляции множества показателей между собой), потому что глазу проще воспринимать цвет, а не мелкие цифры. Для построения тепловой карты служит функция `heatmap()`:

```
sns.heatmap(<матрица (таблица) значений>,
            annot=<отображение значений в ячейках>)
```

Матрица может быть задана как двумерной структурой данных (NumPy-массивом, датафреймом), так и списком списков. Значение `annot` по умолчанию `False`, то есть числовое значение в ячейках тепловой карты не отображается.

Построим тепловую карту по корреляционной матрице характеристик ирисов:

```
plt.rcParams["figure.figsize"] = 4, 3  # изменим размеры холста

sns.heatmap(
  iris_df.corr(numeric_only=True),  # матрица корреляции характеристик ирисов
  annot=True  # включим отображение значений в ячейках
)

plt.title("Корреляционная матрица характеристик ирисов");
```



По цветовой шкале в правой части видно, что чем ближе значение к единице, тем оно светлее. Значения на главной диагонали матрицы корреляции неинформативны (там всегда единицы), а среди прочих наиболее светлой является

ячейка со значением корреляции длины и ширины лепестка, что свидетельствует о высокой степени линейной связи между этими показателями.

14.3. Несколько диаграмм на холсте

Иногда бывает удобно расположить на одном холсте несколько диаграмм в виде сетки $n \times m$, где n, m – количество строк и столбцов соответственно. Добиться этого поможет функция `subplot()`.

```
plt.subplot(<n>, <m>, <порядковый номер диаграммы>)
```

Нумерация диаграмм в сетке начинается с единицы. Например, в сетке 2×3 может быть размещено до шести диаграмм (можно и меньше), пронумерованных по порядку с левой верхней до правой нижней (рис. 103).

Диаграмма **1**	Диаграмма **2**	Диаграмма **3**
Диаграмма **4**	Диаграмма **5**	Диаграмма **6**

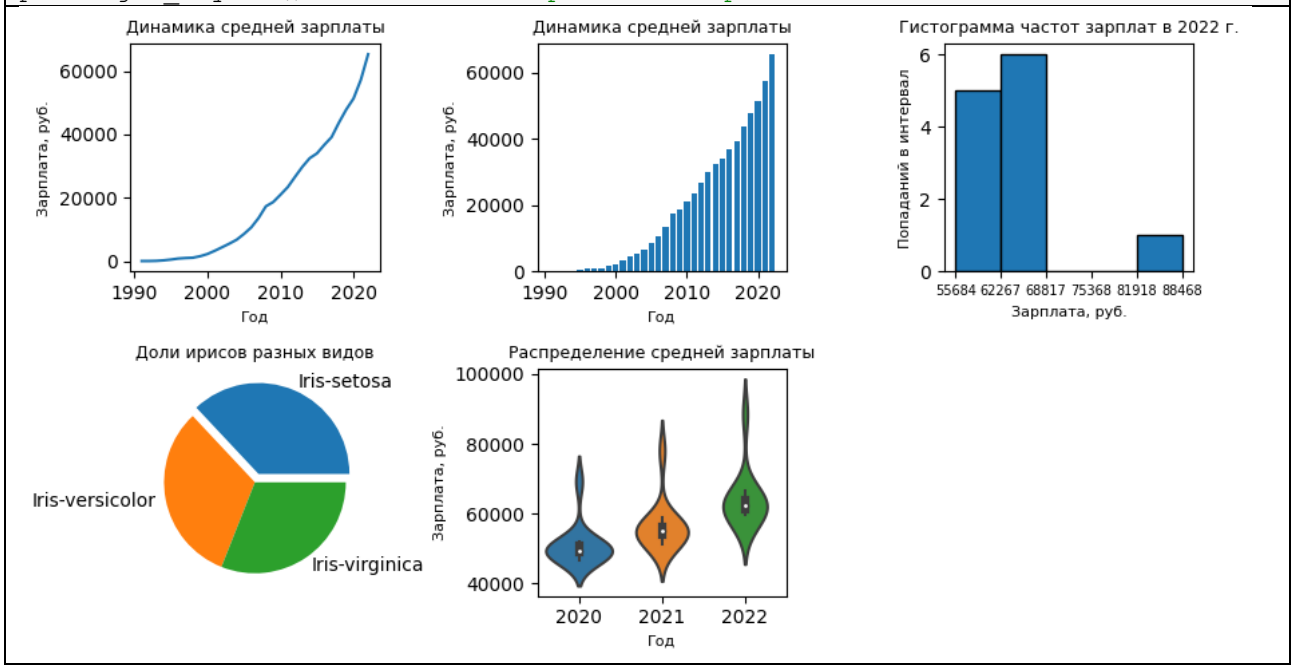
Рис. 103. Нумерация диаграмм в сетке 2×3

Функция `subplot()` должна быть вызвана для *каждой* диаграммы перед функцией ее построения и установкой параметров холста этой диаграммы (таких как заголовок, подписи осей и т.п.).

Разместим на холсте пять диаграмм из ранее рассмотренных примеров. Датафреймы использованы те же.

```
plt.rcParams["figure.figsize"] = 9, 5  # изменим размеры холста

def set_labels(title, xlabel=None, ylabel=None):
  """Функция для установки заголовка диаграммы и подписей осей"""
  plt.xlabel(xlabel, size=8)  # подпись оси x, размер меток
  plt.ylabel(ylabel, size=8)  # подпись оси y, размер меток
  plt.title(title, size=9)  # заголовок диаграммы, размер меток


plt.subplot(2, 3, 1)  # сетка 2 на 3, диаграмма 1
# линейный график
plt.plot(rosstat_df.index, rosstat_df["В среднем за год"])
set_labels("Динамика средней зарплаты", "Год", "Зарплата, руб.")

plt.subplot(2, 3, 2)  # сетка 2 на 3, диаграмма 2
# столбчатая диаграмма
plt.bar(x=rosstat_df.index, height=rosstat_df["В среднем за год"],  width=0.8)
set_labels("Динамика средней зарплаты", "Год", "Зарплата, руб.")

plt.subplot(2, 3, 3)  # сетка 2 на 3, диаграмма 3
# гистограмма
plt.hist(rosstat_df.loc[2022].iloc[5:], bins=5, linewidth=1,edgecolor="black")
```

```
# зададим вручную значения по x - границы интервалов
plt.xticks([bin.left for bin in rosstat_df.loc[2022].iloc[5:]. \
           value_counts(bins=5, sort=False).index] + \
           [rosstat_df.loc[2022].iloc[5:].max()], size=7)
set_labels("Гистограмма частот зарплат в 2022 г.", "Зарплата, руб.",
           "Попаданий в интервал")

plt.subplot(2, 3, 4)  # сетка 2 на 3, диаграмма 4
# круговая диаграмма
plt.pie(freq_table, labels=freq_table.index, explode=[0.1, 0, 0])
set_labels(title="Доли ирисов разных видов")

plt.subplot(2, 3, 5)  # сетка 2 на 3, диаграмма 5
# скрипичная диаграмма
sns.violinplot([rosstat_df.loc[2020].iloc[5:],
                rosstat_df.loc[2021].iloc[5:],
                rosstat_df.loc[2022].iloc[5:]])
plt.xticks(range(3), labels=[2020, 2021, 2022])
set_labels("Распределение средней зарплаты", "Год", "Зарплата, руб.")

plt.tight_layout();  # чтобы диаграммы не пересекались
```



Заметим, что была использована функция `tight_layout()`, которая подбирает параметры отображения диаграмм так, чтобы различные их элементы не пересекались.

14.4. Задания для самостоятельной работы

Решите следующие задачи на построение диаграмм. На всех диаграммах задайте заголовок и подписи осей, поэкспериментируйте с кастомизацией (цветами, формами маркеров и линий, размерами текстовых меток и т.д.).

1. Постройте график синусоиды при помощи диаграммы рассеяния (scatter plot).

2. Постройте линейный график (line chart) функции $y = x^3 + 4x^2 - 5x + 7$ на промежутке [–15; 15], шаг приращения аргумента 0,1.

Для выполнения следующих заданий используйте датасет с объявлениями о продаже квартир из параграфа 13.5, преобразуйте столбцы датафрейма в числовой вид.

3. Постройте гистограмму (histogram), показывающую, с какой частотой встречаются объявления о продаже квартир в том или ином ценовом диапазоне, количество диапазонов (интервалов) выберите самостоятельно.

4. Постройте столбчатую диаграмму с накоплением (stacked bar chart), показывающую соотношение видов квартир (квартира с определенным числом комнат, студия, …) в каждом из районов.

5. Проверьте наличие выбросов в данных по признаку «площадь квартиры» с помощью ящика с усами.

6. Сформируйте матрицу корреляции признаков «этаж», «цена» и «площадь» и отобразите ее в виде тепловой карты.

7. Сравните распределение цен квартир в каждом из районов, построив для каждого района скрипичную диаграмму. Все диаграммы поместите на один холст.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Абдрахманов М.И.* Python. Визуализация данных. Matplotlib. Seaborn. Mayavi. – Devpractice Team, 2020. 412 с.
2. Download Python. [Электронный ресурс]. URL: https://www.python.org/downloads (дата обращения: 30.12.2023).
3. HeadHunter API: документация и библиотеки. [Электронный ресурс]. URL: https://github.com/hhru/api (дата обращения: 30.12.2023).
4. Iris – UCI Machine Learning Repository. [Электронный ресурс]. URL: https://archive.ics.uci.edu/dataset/53/iris (дата обращения: 30.12.2023).
5. Linestyles. [Электронный ресурс]. URL: https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html (дата обращения: 30.12.2023).
6. List of named colors. [Электронный ресурс]. URL: https://matplotlib.org/stable/gallery/color/named_colors.html#css-colors (дата обращения: 30.12.2023).
7. Marker reference. [Электронный ресурс]. URL: https://matplotlib.org/stable/gallery/lines_bars_and_markers/marker_reference.html (дата обращения: 30.12.2023).
8. math – Mathematical functions. [Электронный ресурс]. URL: https://docs.python.org/3/library/math.html (дата обращения: 30.12.2023).
9. Matplotlib documentation. URL: https://matplotlib.org/stable/index.html (дата обращения: 30.12.2023).
10. matplotlib.lines.Line2D. [Электронный ресурс]. URL: https://matplotlib.org/stable/api/_as_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D.set_linestyle (дата обращения: 30.12.2023).
11. NumPy user guide. [Электронный ресурс]. URL: https://numpy.org/doc/stable/user/index.html (дата обращения: 30.12.2023).
12. pandas documentation. [Электронный ресурс]. URL: https://pandas.pydata.org/docs (дата обращения: 30.12.2023).
13. PEP 8 – Style Guide for Python Code. [Электронный ресурс]. URL: https://peps.python.org/pep-0008 (дата обращения: 30.12.2023).
14. PEP8. Коротко о главном. [Электронный ресурс]. URL: https://defpython.ru/pep8__korotko_o_glavnom (дата обращения: 30.12.2023).
15. Random Generator. [Электронный ресурс]. URL: https://numpy.org/doc/stable/reference/random/generator.html (дата обращения: 30.12.2023).

16. The Top Programming Languages 2023 – IEEE Spectrum. [Электронный ресурс]. URL: https://spectrum.ieee.org/the-top-programming-languages-2023 (дата обращения: 30.12.2023).

17. User guide and tutorial. [Электронный ресурс]. URL: https://seaborn.pydata.org/tutorial.html (дата обращения: 30.12.2023).

18. What is my user agent? – WhatIsMyBrowser.com. [Электронный ресурс]. URL: https://www.whatismybrowser.com/detect/what-is-my-user-agent (дата обращения: 30.12.2023).

19. Дзен Пайтона. [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Дзен_Пайтона (дата обращения: 30.12.2023).

20. Законотворчество Артура Кларка. [Электронный ресурс]. URL: https://www.nkj.ru/archive/articles/41906 (дата обращения: 30.12.2023).

21. Описание основных узлов: Арифметический узел. [Электронный ресурс]. URL: https://www.computer-museum.ru/histussr/m1_otchet_4.htm (дата обращения: 30.12.2023).

22. Росстат – Рынок труда, занятость и заработная плата. [Электронный ресурс]. URL: https://rosstat.gov.ru/labor_market_employment_salaries (дата обращения: 30.12.2023).

23. Скачать PyCharm: IDE для профессиональной разработки на Python от JetBrains. [Электронный ресурс]. URL: https://www.jetbrains.com/ru-ru/pycharm/download (дата обращения: 30.12.2023).

24. Смартфоны (рынок России). [Электронный ресурс]. URL: https://www.tadviser.ru/index.php/Статья:Смартфоны_(рынок_России) (дата обращения: 30.12.2023).

Учебное издание

Гарафутдинов Роберт Викторович

Python для анализа данных

Учебное пособие

Редактор *Е. Б. Денисова*
Корректор *Д. Е. Булатова*
Компьютерная вёрстка: *Р. В. Гарафутдинов*

Объем данных 9,19 Мб
Подписано к использованию 26.02.2024

Размещено в открытом доступе
на сайте www.psu.ru
в разделе НАУКА / Электронные публикации
и в электронной мультимедийной библиотеке ELiS

Управление издательской деятельности
Пермского государственного
национального исследовательского университета
614068, г. Пермь, ул. Букирева, 15