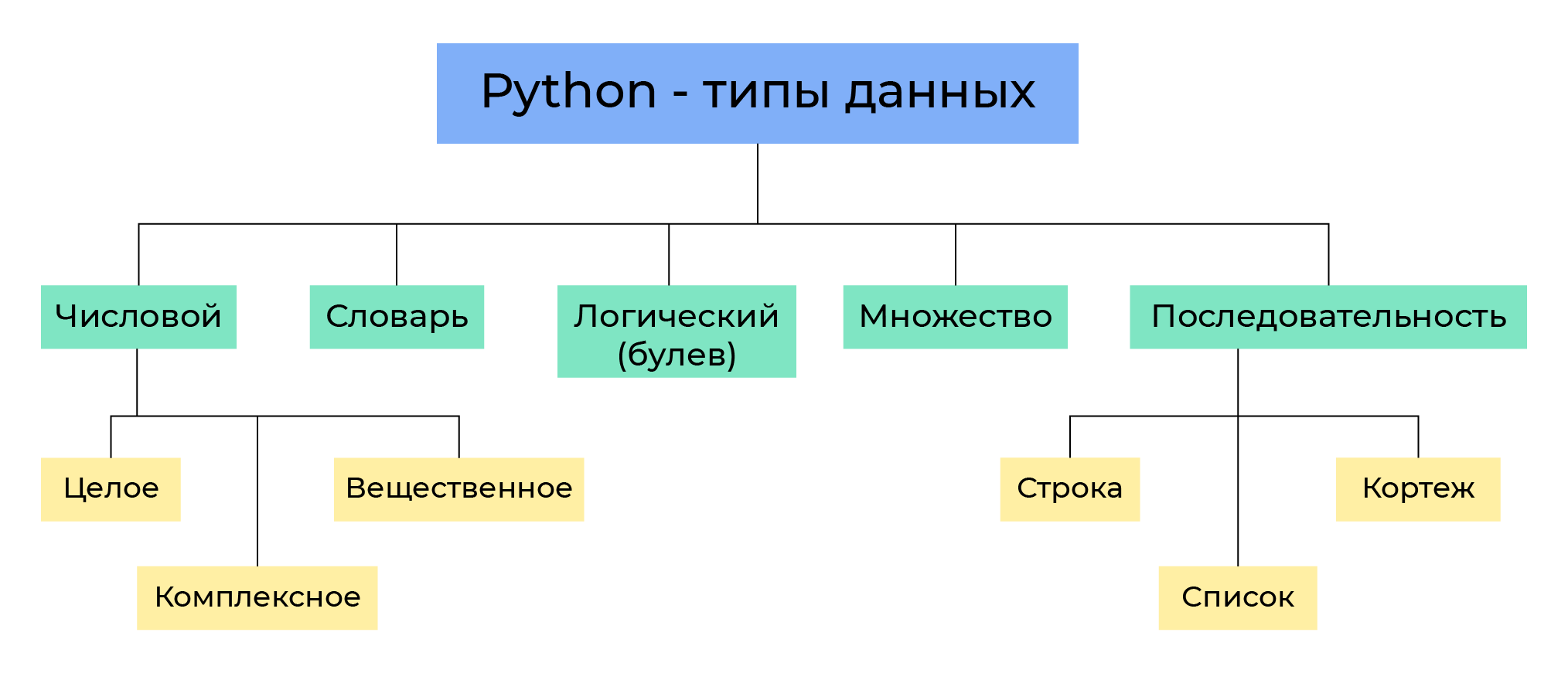
**Тема урока: тип данных bool**

1. Логический тип данных
2. Логические операторы
3. Булевы значения как числа
4. Функции bool(), type(), isinstance()

**Аннотация.** Урок посвящен логическому типу данных bool.

**Типы данных в Python**

В предыдущем [курсе](https://stepik.org/course/58852/syllabus) мы изучали примитивные типы данных, такие как int, float, str, list и т.д. В их числе мы упоминали и о логическом типе данных, однако вскользь. Давайте закроем пробелы текущим уроком, который посвящен целиком и полностью логическому типу данных, который в Python представлен типом bool 😇.

Встроенные типы данных

**Логический тип данных**

Логический тип данных (булев тип, Boolean) — примитивный тип данных в информатике, принимающий два возможных значения, иногда называемых истиной (True) и ложью (False). Присутствует в подавляющем большинстве языков программирования как самостоятельная сущность или реализуется через численный тип данных. В некоторых языках программирования за значение "истина" принимается 11, за значение "ложь" — 00.

Название типа Boolean получило в честь английского математика и логика [Джорджа Буля](https://ru.wikipedia.org/wiki/%D0%91%D1%83%D0%BB%D1%8C,_%D0%94%D0%B6%D0%BE%D1%80%D0%B4%D0%B6), среди прочего занимавшегося вопросами математической логики в середине XIX века.

Джордж Буль

**Логический тип данных в Python**

Логические значения True (истина) и False (ложь) представляют тип данных bool. У этого типа только два возможных значения и два соответствующих литерала: True и False.

Мы активно использовали логический тип данных, когда работали с флагами:

flag = False

или когда использовали условный оператор if-else:

a = 100

b = 17

if b > a:

  print('b больше a')

else:

  print('b не больше a')

Результатом **логического выражения** b > a является булево значение, в данном примере False, так как значение в переменной b меньше значения в переменной a.

Логические выражения можно использовать не только в условном операторе.

Следующий программный код:

print(17 > 7)

print(17 == 7)

print(17 < 7)

выведет:

True

False

False

   Логический тип данных – основа информатики.

**Логические операторы в Python**

Для создания произвольно сложных логических выражений (условий) мы используем три логические операции:

* и (and);
* или (or);
* не (not).

Логические операции используют операнды со значениями True и False и возвращают результат также с логическими значениями. Определённые для объектов типа bool операторы (and, or, not) известны как логические операторы и имеют общеизвестные определения:

* a and b даёт True, если оба операнда True, и False, если **хотя бы один** из них False;
* a or b даёт False, если оба операнда False, и True, если **хотя бы один** из них True;
* not a даёт True, если a имеет значение False, и False, если a имеет значение True.

Следующий программный код:

a = True

b = False

print('a and b is', a and b)

print('a or b is', a or b)

print('not a is', not a)

выведет:

a and b is False

a or b is True

not a is False

**Запомните:** приоритет оператора not выше, чем у оператора and, приоритет которого, в свою очередь, выше, чем у оператора or.

**Булевы значения как числа**

Логические значения в Python можно трактовать как числа. Значению True соответствует число 11, в то время как значению False соответствует 00. Таким образом, мы можем сравнить логические значения с числами:

Следующий программный код:

print(True == 1)

print(False == 0)

выведет:

True

True

Мы можем также применять арифметические операции к логическим значениям.

Следующий программный код:

print(True + True + True - False)

print(True + (False / True))

выведет:

3

1.0

Возможность трактовать булевы выражения как числа на практике используется не так часто. Однако есть один прием, который может оказаться полезным. Поскольку True равно 11, а False равно 00, сложение логических значений вместе – это быстрый способ подсчета количества значений True. Это может пригодиться, когда требуется подсчитать количество элементов, удовлетворяющих условию.

Следующий программный код:

numbers = [1, 2, 3, 4, 5, 8, 10, 12, 15, 17]

res = 0

for num in numbers:

res += (num % 2 == 0)

print(res)

выведет количество **четных элементов** списка numbers, то есть число 5.

**Примечания**

**Примечание 1.** Вместо избыточного кода:

if flag == True:

программисты обычно пишут код:

if flag:

Аналогично, вместо кода

if flag == False:

программисты обычно пишут код:

if not flag:

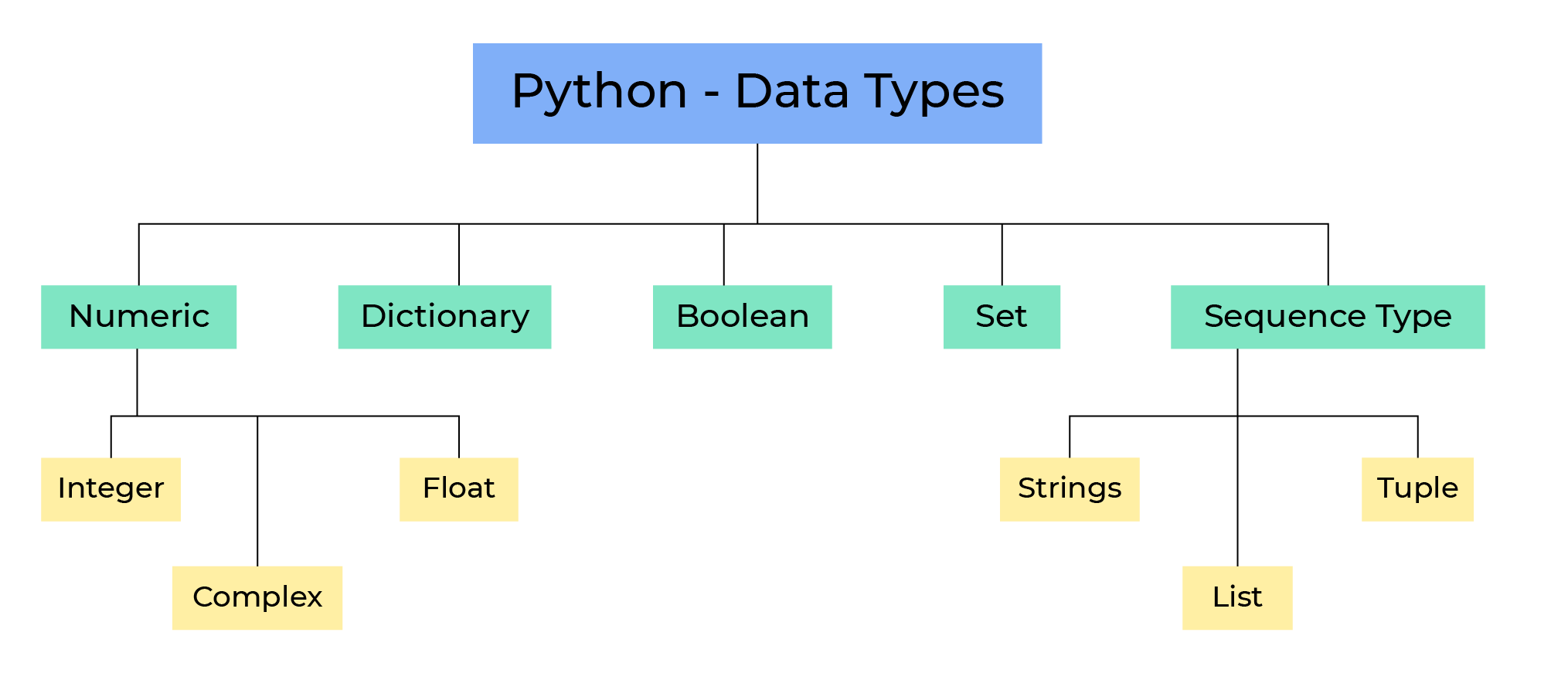
**Примечание 2.** Операторы and и or **ленивые**:

* при вычислении логического выражения x and y, если x == False, то результат всего выражения x and y будет False, так что y **не вычисляется**;
* при вычислении логического выражения x or y, если x == True, то результат всего выражения x or y будет True, и y **не вычисляется**.

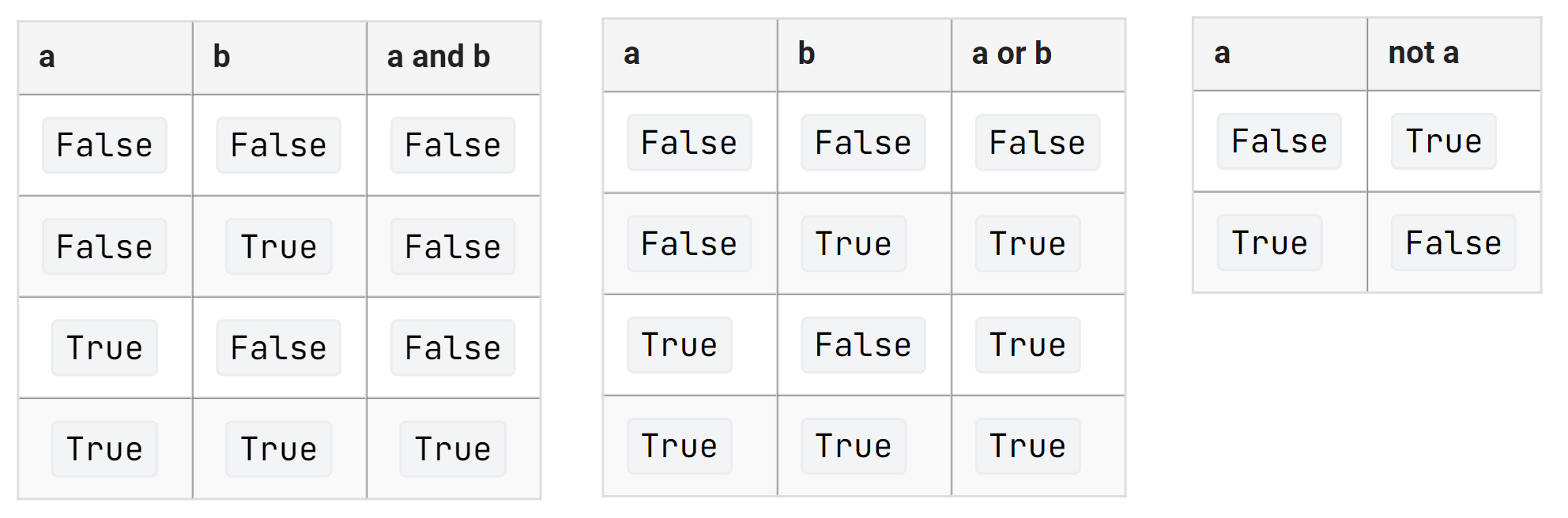
**Примечание 3.** Математическая теория булевой логики определяет, что никакие другие операторы, кроме not, and и or, не нужны. Все остальные операторы на двух входах могут быть указаны в терминах этих трех операторов. Все операторы на трех или более входах могут быть указаны в терминах операторов двух входов.

Фактически, даже наличие пары or и and избыточно. Оператор and может быть определен в терминах not и or, а оператор or может быть определен в терминах not и and. Однако, and и or настолько полезны, что во всех языках программирования есть и то, и другое.

**Примечание 4.** Встроенные типы данных на английском языке.



**Примечание 5.** Приведем таблицы истинности для логических операторов and, or, not:



**Примечание 6.** Посмотреть фильм о Джордже Буле на русском языке можно по [ссылке](https://yadi.sk/i/teF5yPpZ_3uHEg).

**Функция bool()**

Для приведения других типов данных к булеву существует функция bool(), работающая по следующим соглашениям:

* **строки:** пустая строка — ложь (False), непустая строка — истина (True);
* **числа:** нулевое число — ложь (False), ненулевое число (в том числе и меньшее нуля) — истина (True);
* **списки:** пустой список — ложь (False), непустой — истина (True).

Следующий программный код:

print(bool('Beegeek'))

print(bool(17))

print(bool(['apple', 'cherry']))

print(bool())

print(bool(''))

print(bool(0))

print(bool([]))

выведет:

True

True

True

False

False

False

False

   Если функцию bool() вызвать без аргументов, то она вернет значение False.

**Функции, возвращающие булево значение**

Мы можем создавать функции, возвращающие булевы значения (True или False). Такая практика очень полезна. Напишем функцию is\_even(), принимающую одно число и возвращающую значение True, если переданное число четное и False - в противном случае:

def is\_even(num):

return num % 2 == 0

Следующий программный код:

print(is\_even(8))

print(is\_even(7))

выведет:

True

False

   В программировании функция, которая возвращает значение True или False, называется **предикатом**.

**Функция isinstance()**

В языке Python имеется встроенная функция isinstance() для проверки соответствия типа объекта какому-либо типу данных.

Следующий программный код:

print(isinstance(3, int))

print(isinstance(3.5, float))

print(isinstance('Beegeek', str))

print(isinstance([1, 2, 3], list))

print(isinstance(True, bool))

выведет:

True

True

True

True

True

Число 33 – целое число, число 3.53.5 – вещественное число, 'Beegeek' – строка и т.д.

Следующий программный код:

print(isinstance(3.5, int))

print(isinstance('Beegeek', float))

выведет:

False

False

так как число 3.53.5 не целое (float не int), 'Beegeek' - строка, а не вещественное число (str не float).

**Функция type()**

В языке Python имеется встроенная функция type(), позволяющая получить тип указанного в качестве аргумента объекта.

Следующий программный код:

print(type(3))

print(type(3.5))

print(type('Beegeek'))

print(type([1, 2, 3]))

print(type(True))

выведет:

<class 'int'>

<class 'float'>

<class 'str'>

<class 'list'>

<class 'bool'>

Функция type() часто бывает полезна при отладке программного кода, а также в реальном коде, особенно в объектно-ориентированном программировании с наследованием и пользовательскими строковыми представлениями, но об этом позже.

Обратите внимание, что при проверке типов обычно вместо функции type() используется функция isinstance(), так как она принимает во внимание **иерархию типов (ООП)**.

**Тема урока: тип данных NoneType**

1. Тип данных NoneType
2. Литерал None
3. Сравнение None с другими типами данных

**Аннотация.** Урок посвящен типу данных NoneType.

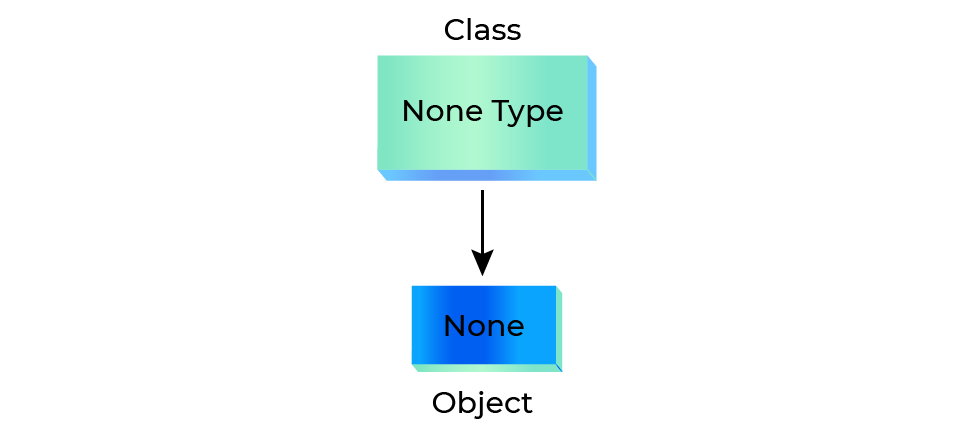
**Пустое значение**

Во многих языках программирования (Java, C++, C#, JavaScript и т.д.) существует ключевое слово null, которое можно присвоить переменным. Концепция ключевого слова null заключается в том, что оно дает переменной нейтральное или "нулевое" поведение.

В языке Python, слово null заменено на None, поскольку слово null звучит не очень дружелюбно, а None относится именно к требуемой функциональности – это ничего и не имеет поведения.

**Литерал None**

Литерал None в Python позволяет представить null переменную, то есть переменную, которая не содержит какого-либо значения. Другими словами, None – это специальная константа, означающая пустоту. Если более точно, то None – это объект специального типа данных NoneType.



Следующий программный код:

var = None

print(type(var))

выведет:

<class 'NoneType'>

Мы можем присвоить значение None любой переменной, однако мы не можем самостоятельно создать другой NoneType объект.

Все переменные, которым присвоено значение None, ссылаются на один и тот же объект типа NoneType. Создание собственных экземпляров типа NoneType недопустимо. Объекты, существующие в единственном экземпляре, называются [синглтонами](https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%B8%D0%BD%D0%BE%D1%87%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)).

**Проверка на None**

Для того, чтобы проверить значение переменной на None, мы используем либо оператор is, либо оператор проверки на равенство ==.

Следующий программный код:

var = None

if var is None: # используем оператор is

  print('None')

else:

  print('Not None')

выведет:

None

Следующий программный код:

var = None

if var == None: # используем оператор ==

  print('None')

else:

  print('Not None')

выведет:

None

Для сравнения переменной с None всегда используйте оператор is. Для встроенных типов поведение is и == абсолютно одинаково, однако с пользовательскими типами могут возникнуть проблемы, так как в Python есть возможность переопределения операторов сравнения в пользовательских типах.

**Сравнение None с другими типами данных**

Сравнение None с любым объектом, отличным от None, дает значение False.

Следующий программный код:

print(None == None)

выведет:

True

Следующий программный код:

print(None == 17)

print(None == 3.14)

print(None == True)

print(None == [1, 2, 3])

print(None == 'Beegeek')

выведет:

False

False

False

False

False

Важно понимать, что следующий программный код:

print(None == 0)

print(None == False)

print(None == '')

выведет:

False

False

False

   Значение None **не отождествляется** с значениями 0, False, ''.

Сравнивать None с другими типами данных можно только на равенство.

Следующий программный код:

print(None > 0)

print(None <= False)

приводит к ошибке:

TypeError: '>' not supported between instances of 'NoneType' and 'int' ('bool')

**Примечания**

**Примечание 1.** Обратите внимание, что функции, не возвращающие значений, на самом деле, в Python возвращают значение None.

def print\_message() :

print('Я - Тимур,')

print('король матана. ')

Мы можем вызвать функцию print\_message() так:

print\_message()

 или так:

res = print\_message()

В переменной res хранится значение None.

**Примечание 2.** Концепция null значений появилась при создании языка ALGOL W великим Чарльзом Хоаром, который позднее назвал собственную идею ошибкой на миллиард долларов. Подробнее можно почитать [тут](http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare).

Чарльз Энтони Ричард Хоар

Чарльз Хоар - автор одного из самых быстрых алгоритмов сортировки, основанной на сравнениях: [быстрая сортировка](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) (QuickSort).

# Тема урока: вложенные списки

1. Вложенные списки
2. Объявление и индексация
3. Функции len(), max(), min()
4. Списочные методы

**Аннотация.** Урок посвящен вложенным спискам, то есть спискам, входящим в качестве элементов в другие списки.

## Введение

Как мы уже знаем, список представляет собой **упорядоченную** последовательность элементов, индексы которых пронумерованы от 00. Элементами списка могут быть любые типы данных – числа, строки, булевы значения и т.д. Например,

numbers = [10, 3]

constants = [3.1415, 2.71828, 1.1415]

countries = ['Russia', 'Armenia', 'Argentina']

flags = [True, False]

Список numbers состоит из 22-х элементов, и каждый из них — **целое число:**

* numbers[0] == 10;
* numbers[1] == 3.

Список constants состоит из 33-х элементов, каждый из которых — **вещественное число:**

* constants[0] == 3.1415;
* constants[1] == 2.71828;
* constants[2] == 1.1415.

Список countries состоит из 33-х элементов, каждый из которых — **строка:**

* countries[0] == 'Russia';
* countries[1] == 'Armenia';
* countries[2] == 'Argentina'.

Список flags состоит из 22-х элементов, и каждый из них — **булево значение:**

* flags[0] == True;
* flags[1] == False.

Мы также говорили, что элементы списка не обязательно должны иметь одинаковый тип данных. Список может содержать значения **разных типов данных**:

info = ['Timur', 1992, 72.5]

Список info содержит строковое значение, целое число и число с плавающей точкой:

* info[0] == 'Timur';
* info[1] == 1992;
* info[2] == 72.5.

Обычно элементы списка содержат данные одного типа, и на практике редко приходится создавать списки, содержащие элементы разных типов данных.

## Вложенные списки

Оказывается, элементами списков могут быть другие списки и в реальной разработке такая конструкция оказывается очень полезной. Такие списки называются **вложенными списками**.

### Создание вложенного списка

Работа с вложенными списками принципиально ничем не отличается от работы со списками, например, чисел или строк. Чтобы создать вложенный список, мы также перечисляем элементы через запятую в квадратных скобках:

my\_list = [[0], [1, 2], [3, 4, 5]]

Переменная my\_list ссылается на список, состоящий из других списков (с вложенными списками).

Поскольку глубина вложенности списка my\_list равна двум, то такой список обычно называют **двумерным списком**. На практике, как правило, мы работаем с **двумерными списками**, реже – с **трехмерными**.

Рассмотрим программный код:

my\_list = [[0], [1, 2], [3, 4, 5]]

print(my\_list)

print(my\_list[0])

print(my\_list[1])

print(my\_list[2])

print(len(my\_list))

Результатом работы такого кода будет:

[[0], [1, 2], [3, 4, 5]]

[0]

[1, 2]

[3, 4, 5]

3

Давайте взглянем на каждую строку приведенного кода поближе.

* строка 11 создает список и присваивает его переменной my\_list. Список имеет три элемента, и каждый элемент тоже является списком:
  + элементом my\_list[0] является список [0];
  + элементом my\_list[1] является список [1, 2];
  + элементом my\_list[2] является список [3, 4, 5].
* строка 33 распечатывает весь список my\_list;
* строка 44 распечатывает элемент my\_list[0];
* строка 55 распечатывает элемент my\_list[1];
* строка 66 распечатывает элемент my\_list[2];
* строка 77 распечатает количество элементов списка my\_list, то есть число 33.

### Индексация

При работе с **одномерными списками** мы использовали **индексацию**, то есть обращение к конкретному элементу по его индексу. Аналогично можно индексировать и вложенные списки:

my\_list = ['Python', [10, 20, 30], ['Beegeek', 'Stepik']]

print(my\_list[0])

print(my\_list[1])

print(my\_list[2])

Результатом работы такого кода будет:

Python

[10, 20, 30]

['Beegeek', 'Stepik']

Так как элементы списка my\_list – строка и списки, их также можно индексировать.

Рассмотрим программный код:

my\_list = ['Python', [10, 20, 30], ['Beegeek', 'Stepik!']]

print(my\_list[0][2]) # индексирование строки 'Python'

print(my\_list[1][1]) # индексирование списка [10, 20, 30]

print(my\_list[2][-1]) # индексирование списка ['Beegeek', 'Stepik!']

print(my\_list[2][-1][-1]) # индексирование строки 'Stepik!'

Результатом работы такого кода будет:

t

20

Stepik!

!

Попытка обратиться к элементу списка по несуществующему индексу:

print(my\_list[3]) # у списка my\_list индексы от 0 до 2

вызовет ошибку:

IndexError: index out of range

### Функции len(), max(), min()

В прошлом курсе мы рассматривали встроенные функции max(), min(), len(), полезные и при работе с вложенными списками (обработке вложенных списков).

#### Функция len()

Рассмотрим программный код:

my\_list = [[0], [1, 2], [3, 4, 5], [], [10, 20, 30]]

print(len(my\_list))

Результатом работы такого кода будет:

5

Обратите внимание, встроенная функция len() возвращает количество элементов (число 55) списка my\_list, а не общее количество элементов во всех списках (число 99).

Если требуется посчитать общее количество элементов во вложенном списке my\_list, мы можем использовать цикл for в связке с функцией len():

total = 0

my\_list = [[0], [1, 2], [3, 4, 5], [], [10, 20, 30]]

for li in my\_list:

total += len(li)

print(total)

Результатом работы такого кода будет:

9

Переменная li последовательно принимает все значения элементов списка my\_list, то есть является сама по себе списком, поэтому мы можем вызывать функцию len() с переданным аргументом li.

#### Функции min() и max()

Функции min() и max() могут работать и со списками. Если этим функциям передается несколько списков, то целиком возвращается один из переданных списков. При этом сравнение происходит поэлементно: сначала сравниваются первые элементы списков. Если они не равны, то функция min() вернет тот список, первый элемент которого меньше, max() – наоборот. Если первые элементы равны, то будут сравниваться вторые и т. д.

Рассмотрим программный код:

list1 = [1, 7, 12, 0, 9, 100]

list2 = [1, 7, 90]

list3 = [1, 10]

print(min(list1, list2, list3))

print(max(list1, list2, list3))

Результатом работы такого кода будет:

[1, 7, 12, 0, 9, 100]

[1, 10]

Функции min() и max() также можно использовать при работе с вложенными списками. Рассмотрим программный код:

list1 = [[1, 7, 12, 0, 9, 100], [1, 7, 90], [1, 10]]

list2 = [['a', 'b'], ['a'], ['d', 'p', 'q']]

print(min(list1))

print(max(list1))

print(min(list2))

print(max(list2))

Результатом работы такого кода будет:

[1, 7, 12, 0, 9, 100]

[1, 10]

['a']

['d', 'p', 'q']

Обратите внимание – элементы вложенных списков в этой ситуации должны быть сравнимы.

Таким образом, следующий код:

my\_list = [[1, 7, 12, 0, 9, 100], ['a', 'b']]

print(min(my\_list))

print(max(my\_list))

приведет к возникновению ошибки:

TypeError: '<' not supported between instances of 'str' and 'int'

## Примечания

**Примечание 1.** Независимо от вложенности списков, нам нужно помнить по возможности все **списочные методы**:

1. метод append() добавляет новый элемент в конец списка;
2. метод extend() расширяет один список другим списком;
3. метод insert() вставляет значение в список в заданной позиции;
4. метод index() возвращает индекс первого элемента, значение которого равняется переданному в метод значению;
5. метод remove() удаляет первый элемент, значение которого равняется переданному в метод значению;
6. метод pop() удаляет элемент по указанному индексу и возвращает его;
7. метод count() возвращает количество элементов в списке, значения которых равны переданному в метод значению;
8. метод reverse()инвертирует порядок следования значений в списке, то есть меняет его на противоположный;
9. метод copy() создает поверхностную копию списка.;
10. метод clear() удаляет все элементы из списка;
11. оператор del позволяет удалять элементы списка по определенному индексу.

**Примечание 2.** Методы строк, работающие со списками:

1. метод split() разбивает строку на слова, используя в качестве разделителя последовательность пробельных символов, символ табуляции (\t) или символ новой строки (\n).
2. метод join() собирает строку из элементов списка, используя в качестве разделителя строку, к которой применяется метод.

**Примечание 3.** Язык Python не ограничивает нас в уровнях вложенности: элементами списка могут быть списки, их элементами могут быть другие списки, элементами которых в свою очередь могут быть другие списки...

**Тема урока: вложенные списки**

1. Создание вложенных списков
2. Считывание вложенных списков
3. Перебор элементов вложенных списков
4. Обработка вложенных списков
5. Вывод вложенных списков

**Аннотация.** Урок посвящен работе с вложенными списками.

**Создание вложенных списков**

Для создания вложенного списка можно использовать литеральную форму записи – перечисление элементов через запятую в квадратных скобках:

my\_list = [[0], [1, 2], [3, 4, 5]]

Иногда нужно создать вложенный список, заполненный по определенному правилу – шаблону. Например, список длиной n, содержащий списки длиной m, каждый из которых заполнен нулями.

Рассмотрим несколько способов решения задачи.

**Способ 1.** Создадим пустой список, потом n раз добавим в него новый элемент – список длины m, составленный из нулей:

n, m = int(input()), int(input()) # считываем значения n и m

my\_list = []

for \_ in range(n):

my\_list.append([0] \* m)

print(my\_list)

Если ввести значения n = 3, m = 5, то результатом работы такого кода будет:

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

Если передать значения n = 5, m = 3, то результатом работы такого кода будет:

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]

**Способ 2.** Сначала создадим список из n элементов (для начала просто из n нулей). Затем сделаем каждый элемент списка ссылкой на другой список из m элементов, заполненный нулями:

n, m = int(input()), int(input()) # считываем значения n и m

my\_list = [0] \* n

for i in range(n):

my\_list[i] = [0] \* m

print(my\_list)

**Способ 3.** Можно использовать генератор списка: создадим список из n элементов, каждый из которых будет списком, состоящих из m нулей:

n, m = int(input()), int(input()) # считываем значения n и m

my\_list = [[0] \* m for \_ in range(n)]

print(my\_list)

В этом случае каждый элемент создается независимо от остальных (заново конструируется вложенный список [0] \* m для заполнения очередного элемента списка).

Обратите внимание, что очевидное решение, использующее операцию умножения списка на число (операция повторения), оказывается неверным:

n, m = int(input()), int(input()) # считываем значения n и m

my\_list = [[0] \* m ] \* n

print(my\_list)

В этом легко убедиться, если присвоить элементу my\_list[0][0] любое значение, например, 17, а затем вывести список на печать:

n, m = int(input()), int(input())

my\_list = [[0] \* m ] \* n

my\_list[0][0] = 17

print(my\_list)

Если ввести значения n = 5, m = 3, то результатом работы такого кода будет:

[[17, 0, 0], [17, 0, 0], [17, 0, 0], [17, 0, 0], [17, 0, 0]]

То есть, изменив значение элемента списка my\_list[0][0], мы также изменили значения элементов my\_list[1][0], my\_list[2][0], my\_list[3][0], my\_list[4][0].

Причина такого поведения кроется в самой природе списков (тип  list). В Python списки – ссылочный тип данных. Конструкция [0] \* m возвращает **ccылку** на список из m нулей. Повторение этого элемента создает список из n ссылок на один и тот же список.

Вложенный список нельзя создать при помощи операции повторения (умножения списка на число). Для корректного создания вложенного списка мы используем способы 1−31−3, отдавая предпочтение способу 33.

**Считывание вложенных списков**

Если элементы списка вводятся через клавиатуру (каждая строка на отдельной строке, всего n строк, числа в строке разделяются пробелами), для ввода списка можно использовать следующий код:

n = 4 # количество строк (элементов)

my\_list = []

for \_ in range(n):

elem = [int(i) for i in input().split()] # создаем список из элементов строки

my\_list.append(elem)

В этом примере мы используем списочный метод append(), передавая ему в качестве аргумента другой список. Так у нас получается список списков.

В результате, если на вход программе подаются строки:

2 4

6 7 8 9

1 3

5 6 5 4 3 1

то в переменной my\_list будет храниться список:

[[2, 4], [6, 7, 8, 9], [1, 3], [5, 6, 5, 4, 3, 1]]

Не забывайте, что метод split() возвращает **список строк, а не чисел**. Поэтому мы предварительно сконвертировали строку в число, с помощью вызова функции int().

Также следует помнить отличие работы списочных методов append() и extend().

Следующий код:

n = 4

my\_list = []

for \_ in range(n):

elem = [int(i) for i in input().split()]

my\_list.extend(elem)

создает одномерный **(!)** список, а не вложенный. В переменной my\_list будет храниться список:

[2, 4, 6, 7, 8, 9, 1, 3, 5, 6, 5, 4, 3, 1]

**Перебор и вывод элементов вложенного списка**

Как мы уже знаем, для доступа к элементу списка указывают индекс этого элемента в квадратных скобках. В случае двумерных вложенных списков надо указать два индекса (каждый в отдельных квадратных скобках), в случае трехмерного списка — три индекса и т. д.

Рассмотрим программный код:

my\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(my\_list[0][0])

print(my\_list[1][2])

print(my\_list[2][1])

Результатом работы такого кода будет:

1

6

8

Когда нужно перебрать все элементы вложенного списка (например, чтобы вывести их на экран), обычно используются **вложенные циклы**.

Рассмотрим программный код:

my\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for i in range(len(my\_list)):

for j in range(len(my\_list[i])):

print(my\_list[i][j], end=' ') # используем необязательный параметр end

print() # перенос на новую строку

 Результатом работы такого кода будет:

1 2 3

4 5 6

7 8 9

Вызов функции print() с пустыми параметрами нужен для того, чтобы переносить вывод на новую строку, после того как будет распечатан очередной элемент (список) вложенного списка.

В предыдущем примере мы перебирали **индексы элементов**, а можно сразу перебирать сами элементы вложенного списка:

my\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in my\_list:

for elem in row:

print(elem, end=' ')

print()

 Результатом работы такого кода будет:

1 2 3

4 5 6

7 8 9

Перебор элементов вложенного списка по индексам дает нам больше гибкости для вывода данных. Например, поменяв порядок переменных i и j, мы получаем иной тип вывода:

my\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for i in range(len(my\_list)):

for j in range(len(my\_list[i])):

print(my\_list[j][i], end=' ') # выводим my\_list[j][i] вместо my\_list[i][j]

print()

 Результатом работы такого кода будет:

1 4 7

2 5 8

3 6 9

**Обработка вложенных списков**

Для обработки элементов вложенного списка, так же как и для вывода его элементов на экран, как правило, используются **вложенные циклы**.

Используем вложенный цикл для подсчета суммы всех чисел в списке:

my\_list = [[1, 9, 8, 7, 4], [7, 3, 4], [2, 1]]

total = 0

for i in range(len(my\_list)):

for j in range(len(my\_list[i])):

total += my\_list[i][j]

print(total)

Или то же самое с циклом не по индексу, а по значениям:

my\_list = [[1, 9, 8, 7, 4], [7, 3, 4], [2, 1]]

total = 0

for row in my\_list:

for elem in row:

total += elem

print(total)

Таким образом, можно обработать элементы вложенного списка практически в любом языке программирования. В Python, однако, можно упростить код, если использовать встроенную функцию sum(), которая принимает список чисел и возвращает его сумму. Подсчет суммы с помощью функции sum() выглядит так:

my\_list = [[1, 9, 8, 7, 4], [7, 3, 4], [2, 1]]

total = 0

for row in my\_list: # в один цикл

total += sum(row)

print(total)

Названия переменных row (строка) и elem (элемент) удобно использовать при переборе вложенного списка по значениям. Названия переменных i и j используются при переборе вложенного списка по индексам.

# Тема урока: матрицы

1. Работа с матрицами
2. Квадратные и прямоугольные матрицы
3. Функции ljust() и rjust()
4. Главная и побочная диагонали

**Аннотация.** Урок посвящен работе с матрицами — прямоугольными таблицами.

## Матрицы

В прошлых уроках мы изучили **вложенные списки**, то есть списки, входящие в качестве элементов в другие списки. Частный случай вложенных списков — матрицы. Это прямоугольные таблицы, заполненные какими-то значениями, обычно числами.



Матрицы часто применяются в математике, так как многие задачи с их помощью гораздо проще сформулировать, записать и решить.

Для работы с матрицами нужно уметь получать элемент �*i*-й строки �*j*-го столбца. Для этого обычно заводят список строк матрицы, где каждая строка — список элементов. Получается вложенный список или список списков. Теперь, чтобы получить определенный элемент, достаточно из списка строк матрицы выбрать �*i*-ю и взять �*j*-й элемент этой строки.

Давайте заведем матрицу размера 3×43×4 (33 строки и 44 столбца), содержащую числа, и получим элемент на позиции (2, 3)(2, 3), то есть элемент второй строки в третьем столбце.

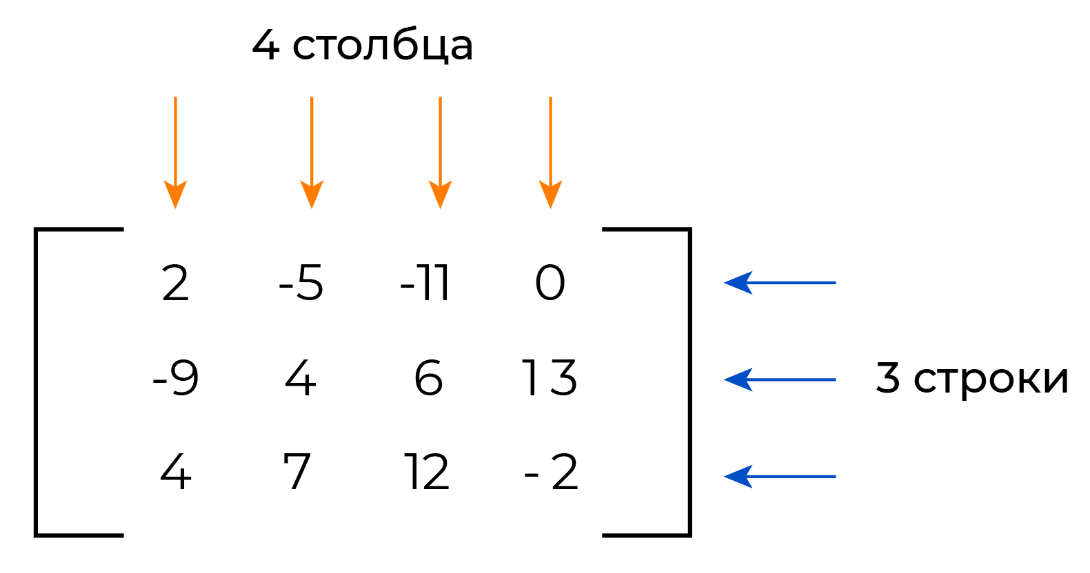
matrix = [[2, -5, -11, 0],

[-9, 4, 6, 13],

[4, 7, 12, -2]]

print(matrix[1][2]) # вывод элемента на позиции (2, 3)

В переменной matrix хранится вся матрица, при этом matrix[1] — список значений во второй строке, matrix[1][2] — элемент в третьем столбце этой строки.



В математике нумерация строк и столбцов начинается с единицы, а не с нуля. По договоренности сначала всегда указывается строка, а затем — столбец. Элемент на �*i*-ой строке, �*j*-м столбце матрицы �*a* обозначается так – ���*aij*​.

## Перебор элементов матрицы

Чтобы перебрать элементы матрицы, необходимо использовать **вложенные циклы**. Например, выведем на экран все элементы матрицы, **перебирая их по строкам**:

rows, cols = 3, 4 # rows - количество строк, cols - количество столбцов

matrix = [[2, 3, 1, 0],

[9, 4, 6, 8],

[4, 7, 2, 7]]

for r in range(rows):

for c in range(cols):

print(matrix[r][c], end=' ')

print()

Результатом работы такого кода будет:

2 3 1 0

9 4 6 8

4 7 2 7

Для перебора элементов матрицы **по столбцам** можно использовать следующий код:

rows, cols = 3, 4 # rows - количество строк, cols - количество столбцов

matrix = [[2, 3, 1, 0],

[9, 4, 6, 8],

[4, 7, 2, 7]]

for c in range(cols):

for r in range(rows):

print(matrix[r][c], end=' ')

print()

Результатом работы такого кода будет:

2 9 4

3 4 7

1 6 2

0 8 7

## Функции ljust() и rjust()

Рассмотрим программный код:

rows, cols = 3, 4 # rows - количество строк, cols - количество столбцов

matrix = [[277, -930, 11, 0],

[9, 43, 6, 87],

[4456, 8, 290, 7]]

for r in range(rows):

for c in range(cols):

print(matrix[r][c], end=' ')

print()

 Результатом работы такого кода будет:

277 -930 11 0

9 43 6 87

4456 8 290 7

Выведенная матрица не сильно похожа на упорядоченный прямоугольник. Элементы матрицы имеют разное количество разрядов и результат вывода получается смазанным. Для решения проблемы удобно использовать строковые методы ljust() и rjust().

### Метод ljust()

Строковый метод ljust() выравнивает текст по ширине, **добавляя пробелы** в конец текста.

Результатом выполнения следующего кода:

print('a'.ljust(3))

print('ab'.ljust(3))

print('abc'.ljust(3))

будет:

a⎵⎵

ab⎵

abc

Исходная строка не обрезается, даже если в ней больше символов, чем нужно.

Результатом выполнения следующего кода:

print('abcdefg'.ljust(3))

будет:

abcdefg

Строковый метод ljust() использует вместо пробела другой символ, если передать ему второй аргумент, необязательный.

Результатом выполнения следующего кода:

print('a'.ljust(5, '\*'))

print('ab'.ljust(5, '$'))

print('abc'.ljust(5, '#'))

будет:

a\*\*\*\*

ab$$$

abc##

### Метод rjust()

Строковый метод rjust() выравнивает текст по ширине, **добавляя пробелы** в начало текста.

Результатом выполнения следующего кода:

print('a'.rjust(3))

print('ab'.rjust(3))

print('abc'.rjust(3))

будет:

⎵⎵a

⎵ab

abc

Исходная строка не обрезается, даже если в ней больше символов, чем нужно.

Результатом выполнения следующего кода:

print('abcdefg'.rjust(3))

будет:

abcdefg

Строковый метод rjust() использует вместо пробела другой символ, если передать ему второй аргумент, необязательный.

Результатом выполнения следующего кода:

print('a'.rjust(5, '\*'))

print('ab'.rjust(5, '$'))

print('abc'.rjust(5, '#'))

будет:

\*\*\*\*a

$$$ab

##abc

Применив метод ljust() для выравнивания столбцов, при выводе таблицы мы получим следующий код:

rows, cols = 3, 4 # rows - количество строк, cols - количество столбцов

matrix = [[277, -930, 11, 0],

[9, 43, 6, 87],

[4456, 8, 290, 7]]

for r in range(rows):

for c in range(cols):

print(str(matrix[r][c]).ljust(6), end='')

print()

Результатом выполнения такого кода будет:

277 -930 11 0

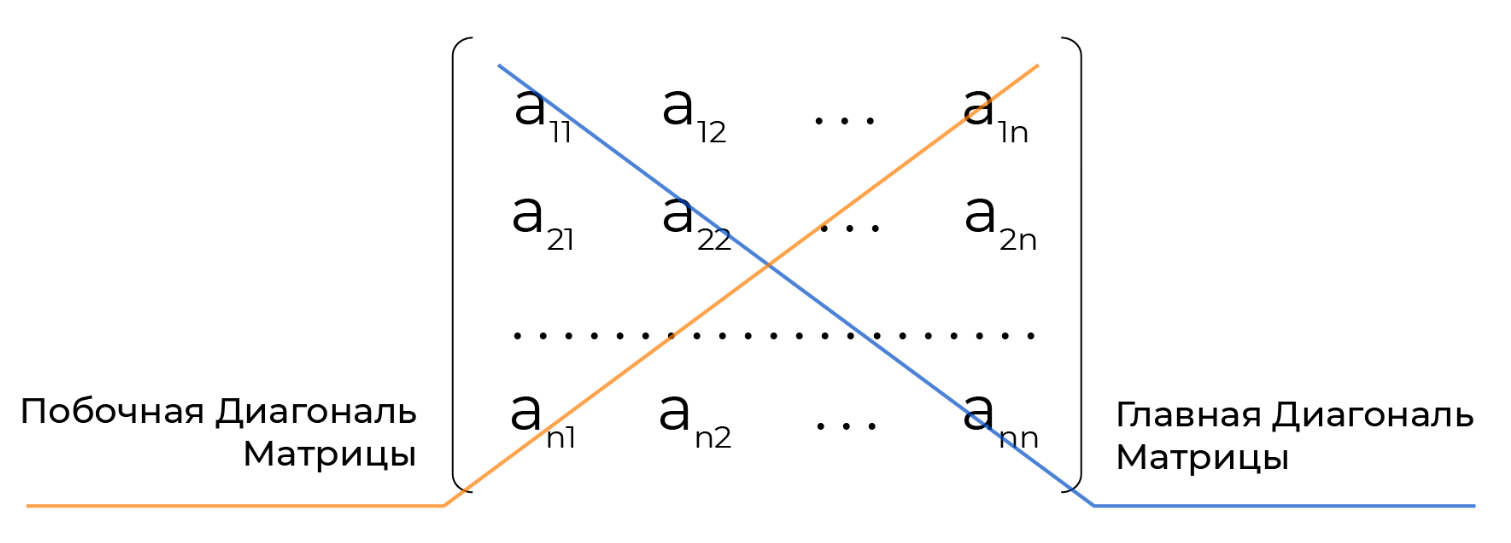
9 43 6 87

4456 8 290 7

## Квадратные матрицы

Матрица с одинаковым количеством строк и столбцов называется **квадратной**. У квадратной матрицы есть две диагонали:

* **главная**: проходит из верхнего левого в правый нижний угол матрицы;
* **побочная**: проходит из нижнего левого в правый верхний угол матрицы.



Элементы с равными индексами i == j находятся на **главной диагонали**. Такие элементы обозначаются matrix[i][i].

Элементы с индексами i и j, связанными соотношением i + j + 1 = n (или j = n - i - 1), где n — размерность матрицы, находятся на **побочной диагонали**.

Таким образом, чтобы установить элементы главной или побочной диагонали, достаточно одного цикла.

Результатом выполнения следующего кода:

n = 8

matrix = [[0]\*n for \_ in range(n)] # создаем квадратную матрицу размером 8×8

for i in range(n): # заполняем главную диагональ единицами, а побочную двойками

matrix[i][i] = 1

matrix[i][n-i-1] = 2

for r in range(n): # выводим матрицу

for c in range(n):

print(matrix[r][c], end=' ')

print()

будет:

1 0 0 0 0 0 0 2

0 1 0 0 0 0 2 0

0 0 1 0 0 2 0 0

0 0 0 1 2 0 0 0

0 0 0 2 1 0 0 0

0 0 2 0 0 1 0 0

0 2 0 0 0 0 1 0

2 0 0 0 0 0 0 1

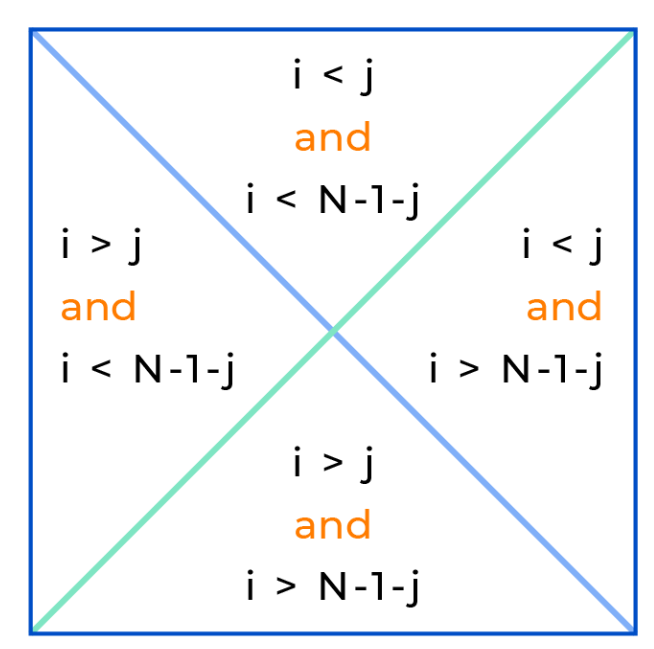
Индексыi и jэлементов на главной диагонали связаны соотношением i = j. Индексы i и jэлементов на побочной диагонали связаны соотношением i + j + 1 = n (или  j = n - i - 1), где n — размерность матрицы.

Заметим также, что:

* если элемент находится выше главной диагонали, то i < j, если ниже - i > j.
* если элемент находится выше побочной диагонали, то i + j + 1 < n, если ниже - i + j + 1 > n.

## Примечания

**Примечание 1.** Чтобы понять, в какой области лежит элемент можно воспользоваться следующей картинкой.



**Примечание 2.** Используйте функцию print\_matrix() для вывода квадратной матрицы размерности n:

def print\_matrix(matrix, n, width=1):

for r in range(n):

for c in range(n):

print(str(matrix[r][c]).ljust(width), end=' ')

print()

**Примечание 3.** Для считывания матрицы из n строк, заполненной числами, удобно использовать следующий код:

n = int(input())

matrix = []

for i in range(n):

temp = [int(num) for num in input().split()]

matrix.append(temp)

**Тема урока: операции над матрицами**

1. Сложение матриц
2. Умножение матрицы на число
3. Умножение матриц

**Аннотация.** Урок посвящен основным операциям над матрицами в математике.

**Сложение матриц**

При сложении матриц �*A* и �*B* получается такая матрица �*C*, каждый элемент которой представляет собой сумму пары соответствующих элементов исходных матриц �*A* и �*B*. Складывать можно только матрицы одинаковой размерности (� ×�*n* ×*m*), с равным количеством строк и столбцов. Таким образом, математически сумма матриц выглядит так:��×�=��×�+��×�*Cn*×*m*​=*An*×*m*​+*Bn*×*m*​Каждый элемент искомой матрицы равен сумме соответствующих элементов заданных матриц:

���=���+���*cij*​=*aij*​+*bij*​**Пример**. Даны две матрицы:

�=[2−3154−2],�=[42−5−413]*A*=[25​−34​1−2​],*B*=[4−4​21​−53​]Найти сумму матриц �+�*A*+*B*.  
**Решение.**Имеем

�+�=[2−3154−2]+[42−5−413]=[6−1−4151]*A*+*B*=[25​−34​1−2​]+[4−4​21​−53​]=[61​−15​−41​]

**Свойства сложения матриц**

1. **Коммутативность** – результат сложения матриц не зависит от их перестановки: �+�=�+�;*A*+*B*=*B*+*A*;
2. **Ассоциативность** – результат сложения матриц не зависит от расстановки скобок: �+(�+�)=(�+�)+�;*A*+(*B*+*C*)=(*A*+*B*)+*C*;
3. **Сложение с нулевой матрицей** – для любой матрицы существует нейтральный элемент, которым является нулевая матрица, сложение с которым не изменяет исходную матрицу. Нулевая матрица – матрица, все элементы которой имеют нулевое значение: �+0=0+�=�;*A*+0=0+*A*=*A*;
4. **Существование противоположной матрицы** – для ненулевой матрицы �*A* всегда есть матрица −�−*A*, сложение с которой даст в результате нулевую матрицу:�+(−�)=0.*A*+(−*A*)=0.

**Умножение матрицы на число**

Операция умножения матрицы �*A* на число �*k* заключается в построении матрицы ��=[�⋅���]*kA*=[*k*⋅*aij*​]. Умножать на число можно матрицы любого размера. В результате умножения получается матрица того же размера, что исходная.

Произведение матрицы �*A* на число �*k* – результирующая матрица �=��*B*=*kA* того же размера, полученная умножением каждого из элементов ���*aij*​ исходной матрицы на заданное число.

**Пример.** Даны матрица �*A* и число �*k*:

�=[5−2431−3],�=7*A*=[53​−21​4−3​],*k*=7Найти произведение матрицы и числа.  
**Решение.**Имеем

�⋅�=7⋅[5−2431−3]=[35−1428217−21]*k*⋅*A*=7⋅[53​−21​4−3​]=[3521​−147​28−21​]

**Свойства умножения матрицы на число**

1. Единица – нейтральное число умножения любой матрицы, результат умножения на нейтральное число – исходная матрица:1×�=�;1×*A*=*A*;
2. Результат умножения любой матрицы на ноль – нулевая матрица, все элементы которой равны нулю:0×�=0;0×*A*=0;
3. Для матриц одного размера и действительного числа выполняется свойство дистрибутивности умножения относительно сложения:�×(�+�)=�×�+�×�;*k*×(*A*+*B*)=*k*×*A*+*k*×*B*;
4. Для любой матрицы и суммы действительных чисел выполняется свойство дистрибутивности:(�+�)×�=�×�+�×�;(*k*+*n*)×*A*=*k*×*A*+*n*×*A*;
5. Для любой матрицы и произведения любых действительных чисел выполняется свойство ассоциативности умножения:(�×�)×�=�×(�×�).(*k*×*n*)×*A*=*k*×(*n*×*A*).

**Умножение матрицы на матрицу**

Умножение двух матриц �*A* и �*B* – вычисление результирующей матрицы �*C*, каждый элемент ���*cij*​ которой равен сумме произведений элементов соответствующих строки первой матрицы ���*air*​ и столбца второй матрицы ���*brj*​.

Одну матрицу можно умножать на другую только тогда, когда **количество столбцов в первой матрице совпадает с количеством строк** во второй матрице. То есть матрицы должны быть согласованы по размерности. Результат умножения матрицы размера �×�*n*×*m* на матрицу размером �×�*m*×*k* – матрица размером �×�*n*×*k*.

Итак, произведение матрицы ��×�*An*×*m*​ на матрицу ��×�*Bm*×*k*​ – матрица ��×�*Cn*×*k*​, элемент ���*cij*​ которой, находящийся в �*i*-ой строке и �*j*-ом столбце, равен сумме произведений элементов �*i*-ой строки матрицы �*A* на соответствующие элементы �*j*-ого столбца матрицы �*B*.

Каждый элемент матрицы ��×�*Cn*×*k*​ равен:

���=∑�=1����⋅���=��1⋅�1�+��2⋅�2�+…+���⋅���,*cij*​=*r*=1∑*m*​*air*​⋅*brj*​=*ai*1​⋅*b*1*j*​+*ai*2​⋅*b*2*j*​+…+*aim*​⋅*bmj*​,где �*r* принимает значение от 11 до �*m*.

**Пример.** Даны две матрицы �*A* и �*B*:

�=[2−3154−2],�=[−752−143]*A*=[25​−34​1−2​],*B*=⎣⎡​−724​5−13​⎦⎤​Найти произведение матриц �×�*A*×*B*.  
**Решение.**При перемножении матрицы �2×3*A*2×3​ на матрицу �3×2*B*3×2​ мы получаем матрицу �2×2*C*2×2​. Имеем:

�11=2⋅(−7)+(−3)⋅2+1⋅4=−16,�12=2⋅5+(−3)⋅(−1)+1⋅3=16�21=5⋅(−7)+4⋅2+(−2)⋅4=−35,�22=5⋅5+4⋅(−1)+(−2)⋅3=15*c*11​=2⋅(−7)+(−3)⋅2+1⋅4=−16,*c*12​=2⋅5+(−3)⋅(−1)+1⋅3=16*c*21​=5⋅(−7)+4⋅2+(−2)⋅4=−35,*c*22​=5⋅5+4⋅(−1)+(−2)⋅3=15

Таким образом

�⋅�=[2−3154−2]⋅[−752−143]=[−1616−3515]*A*⋅*B*=[25​−34​1−2​]⋅⎣⎡​−724​5−13​⎦⎤​=[−16−35​1615​]

**Свойства умножения матриц**

1. **Ассоциативность** – (*A*× *B*)× *C* =*A*×(*B*× *C*);
2. **Дистрибутивность** –*A*×(*B*+*C*)=*A*×*B*+*A*×*C*,(*A*+*B*)×*C*=*A*×*C*+*B*×*C*;
3. **Ассоциативность и коммутативность относительно умножения на число** (*k*×*A*)×*B*=*k*×(*A*×*B*)=*A*×(*k*×*B*);
4. **В общем случае умножение матриц не коммутативно** –;*A*×*B*=*B*×*A*;

**Примечания**

**Примечание 1.** [Демонстрация](http://matrixmultiplication.xyz/) операции перемножения матриц.

**Тема урока: кортежи**

1. Тип данных tuple
2. Особенности работы с кортежами

**Аннотация.** Урок посвящен кортежам (тип данных tuple).

Мы изучили списки и строки. Списки – изменяемые коллекции, строки – неизменяемые последовательности Unicode символов. В Python имеются и неизменяемые последовательности содержащие, в отличие от строк, абсолютно произвольные данные. Такие коллекции называются **кортежами** (tuple, читается "тюпл").

**Кортежи**

Рассмотрим следующий программный код:

my\_list = [1, 2, 3, 4, 5]

Мы объявили список чисел и присвоили его переменной my\_list. Содержимое списка можно изменять.

Следующий программный код:

my\_list = [1, 2, 3, 4, 5]

my\_list[0] = 9

my\_list[4] = 7

print(my\_list)

выведет:

[9, 2, 3, 4, 7]

Заменив квадратные скобки при объявлении списка на круглые, мы объявляем кортеж:

my\_tuple = (1, 2, 3, 4, 5)

Кортежи по своей природе (задумке) – **неизменяемые аналоги списков**. Поэтому программный код:

my\_tuple = (1, 2, 3, 4, 5)

my\_tuple[0] = 9

my\_tuple[4] = 7

print(my\_tuple)

приводит к ошибке

TypeError: 'tuple' object does not support item assignment

   Кортеж (tuple) – ещё один вид коллекций в Python. Похож на список, но, в отличие от списка, **неизменяемый**.

В литеральной форме кортеж записывается в виде последовательности элементов в круглых скобках, а список – в квадратных.

**Примеры кортежей**

empty\_tuple = () # пустой кортеж

point = (1.5, 6.0) # кортеж из двух чисел

names = ('Timur', 'Ruslan', 'Roman') # кортеж из трех строк

info = ('Timur', 'Guev', 28, 170, 60, False) # кортеж из 6 элементов разных типов

nested\_tuple = (('one', 'two'), ['three', 'four']) # кортеж из кортежа и списка

* в переменной empty\_tuple хранится пустой кортеж;
* в переменной point хранится кортеж, состоящий из двух вещественных чисел (такой кортеж удобно использовать для представления точки на координатной плоскости);
* в переменной names хранится кортеж, содержащий три строковых значения;
* в переменной info содержится кортеж, содержащий 66 элементов разного типа (строки, числа, булевы переменные);
* в переменной nested\_tuple содержится кортеж, содержащий другой кортеж и список.

Кортежи могут хранить и содержать в себе объекты любых типов (даже составных) и поддерживают неограниченное количество уровней вложенности.

**Кортеж с одним элементом**

Для создания кортежа с единственным элементом после значения элемента ставят замыкающую запятую:

my\_tuple = (1,)

print(type(my\_tuple)) # <class 'tuple'>

Если запятую пропустить, то кортеж создан не будет. Например, приведенный ниже код просто присваивает переменной my\_tuple целочисленное значение 1:

my\_tuple = (1)

print(type(my\_tuple)) # <class 'int'>

**Зачем использовать кортеж вместо списка?**

Списки могут делать то же, что кортежи, и даже больше. Но **неизменяемость** **кортежей** обеспечивает им особые свойства:

* **скорость**– кортежи быстрее работают, так как из-за неизменяемости хранятся в памяти иначе, и операции с их элементами выполняются заведомо быстрее, чем с компонентами списка. Одна из причин существования кортежей – производительность. Обработка кортежа выполняется быстрее, чем обработка списка, поэтому кортежи удобны для обработки большого объема неизменных данных.
* **безопасность**– неизменяемость превращает их в идеальные константы. Заданные кортежами константы делают код более читаемым и безопасным. Кроме того, в кортеже можно безопасно хранить данные, не опасаясь, что они будут случайно или преднамеренно изменены в программе.

В Python существуют операции, требующие применения кортежа. По мере освоения языка Python вы будете чаще сталкиваться с кортежами.

**Примечания**

**Примечание 1.** Мы уже  сталкивались с кортежами, когда изучали функции, возвращающие несколько значений. Такие функции возвращают именно кортежи.

Рассмотрим функцию get\_powers(), которая принимает в качестве аргумента число и возвращает его 22, 33 и 44 степень.

def get\_powers(num):

return num\*\*2, num\*\*3, num\*\*4

Результатом выполнения следующего кода:

result = get\_powers(5)

print(type(result))

print(result)

будет:

<class 'tuple'>

(25, 125, 625)

**Примечание 2.** Списки предназначены для объединения неопределенного количества однородных сущностей. Кортежи, как правило, объединяют под одним именем несколько разнородных объектов, имеющих различный смысл.

**Примечание 3.** Тот факт, что кортеж является неизменяемым вовсе не означает, что мы не можем поменять содержимое списка в кортеже.

Приведенный ниже код:

my\_tuple = (1, 'python', [1, 2, 3])

print(my\_tuple)

my\_tuple[2][0] = 100

my\_tuple[2].append(17)

print(my\_tuple)

выводит:

(1, 'python', [1, 2, 3])

(1, 'python', [100, 2, 3, 17])

При этом важно понимать: меняется список, а не кортеж. Списки являются ссылочными типами данных, поэтому в кортеже хранится ссылка на список, которая не меняется при изменении самого списка.

# Тема урока: кортежи

1. Функция tuple()
2. Особенности кортежей
3. Методы кортежей
4. Вложенные кортежи

**Аннотация.** Урок посвящен кортежам (тип данных tuple).

## Функция tuple()

Встроенная функция list() может применяться для преобразования **кортежа в список**.

Приведенный ниже код:

number\_tuple = (1, 2, 3, 4, 5)

number\_list = list(number\_tuple)

print(number\_list)

выводит:

[1, 2, 3, 4, 5]

Встроенная функция tuple()  может применяться для преобразования **списка в кортеж**.

Приведенный ниже код:

str\_list = ['один', 'два', 'три']

str\_tuple = tuple(str\_list)

print(str\_tuple)

выводит:

('один', 'два', 'три')

Аналогичным образом мы можем создать кортеж на основании строки.

Приведенный ниже код:

text = 'hello python'

str\_tuple = tuple(text)

print(str\_tuple)

выводит:

('h', 'e', 'l', 'l', 'o', ' ', 'p', 'y', 't', 'h', 'o', 'n')

Обратите внимание, что символ пробел содержится в кортеже str\_tuple.

Преобразование строки в список позволяет получить список символов строки. Это может быть полезно, например, когда надо изменить один символ строки:

s = 'симпотичный'

print(s)

a = list(s)

a[4] = 'а'

s = ''.join(a)

print(s)

Приведенный выше код выводит:

симпотичный

симпатичный

С этой же целью может потребоваться преобразование кортежа в список:

writer = ('Лев Толстой', 1827)

print(writer)

a = list(writer)

a[1] = 1828

writer = tuple(a)

print(writer)

Приведенный выше код выводит:

('Лев Толстой', 1827)

('Лев Толстой', 1828)

## Особенности кортежей

Кортежи поддерживают те же операции, что и списки, за исключением изменяющих содержимое.

Кортежи поддерживают:

* доступ к элементу по индексу (только для получения значений элементов);
* методы, в частности index(), count();
* встроенные функции, в частности len(), sum(), min() и max();
* срезы;
* оператор принадлежности in;
* операторы конкатенации (+) и повторения (\*).

### Функция len()

**Длиной кортежа** называется количество его элементов. Для того, чтобы посчитать длину кортежа мы используем встроенную функцию len().

Следующий программный код:

numbers = (2, 4, 6, 8, 10)

languages = ('Python', 'C#', 'C++', 'Java')

print(len(numbers)) # выводим длину кортежа numbers

print(len(languages)) # выводим длину кортежа languages

print(len(('apple', 'banana', 'cherry'))) # выводим длину кортежа, состоящего из 3 элементов

выведет:

5

4

3

### Оператор принадлежности in

Оператор in позволяет проверить, содержит ли кортеж некоторый элемент.

Рассмотрим следующий код:

numbers = (2, 4, 6, 8, 10)

if 2 in numbers:

print('Кортеж numbers содержит число 2')

else:

print('Кортеж numbers не содержит число 2')

Такой код проверяет, содержит ли кортеж numbers число 22 и выводит соответствующий текст:

Кортеж numbers содержит число 2

Мы можем использовать оператор in вместе с логическим оператором not. Например

numbers = (2, 4, 6, 8, 10)

if 0 not in numbers:

print('Кортеж numbers не содержит нулей')

### Индексация

При работе со списками (строками) мы использовали **индексацию**, то есть обращение к конкретному элементу списка (строки) по его индексу. Аналогично можно индексировать и элементы кортежей.

Для индексации кортежей в Python используются квадратные скобки [], в которых указывается индекс (номер) нужного элемента в кортеже:

Пусть numbers = (2, 4, 6, 8, 10).

Таблица ниже, показывает как работает индексация:

| **Выражение** | **Результат** | **Пояснение** |
| --- | --- | --- |
| numbers[0] | 22 | первый элемент кортежа |
| numbers[1] | 44 | второй элемент кортежа |
| numbers[2] | 66 | третий элемент кортежа |
| numbers[3] | 88 | четвертый элемент кортежа |
| numbers[4] | 1010 | пятый элемент кортежа |

Так же, как и в списках, для нумерации с конца разрешены отрицательные индексы.

| **Выражение** | **Результат** | **Пояснение** |
| --- | --- | --- |
| numbers[-1] | 1010 | пятый элемент кортежа |
| numbers[-2] | 88 | четвертый элемент кортежа |
| numbers[-3] | 66 | третий элемент кортежа |
| numbers[-4] | 44 | второй элемент кортежа |
| numbers[-5] | 22 | первый элемент кортежа |

Как и в списках, попытка обратиться к элементу кортежа по несуществующему индексу:

print(numbers[17])

вызовет ошибку:

IndexError: tuple index out of range

### Срезы

Рассмотрим кортеж numbers = (2, 4, 6, 8, 10).

С помощью среза мы можем получить несколько элементов кортежа, создав диапазон индексов разделенных двоеточием numbers[x:y].

Следующий программный код:

print(numbers[1:3])

print(numbers[2:5])

выводит:

(4, 6)

(6, 8, 10)

При построении среза numbers[x:y] первое число – это то место, где начинается срез (**включительно**), а второе – это место, где заканчивается срез (**невключительно**).

При использовании срезов с кортежами мы также можем опускать второй параметр в срезе numbers[x:] (но поставить двоеточие), тогда срез берется до конца кортежа. Аналогично если опустить первый параметр numbers[:y], то можно взять срез от начала кортежа.

  Срез numbers[:] возвращает копию исходного кортежа.

Как и в списках, можно использовать отрицательные индексы в срезах кортежей.

### Операция конкатенации + и умножения на число \*

Операторы + и \* применяют для кортежей как и для списков.

Следующий программный код:

print((1, 2, 3, 4) + (5, 6, 7, 8))

print((7, 8) \* 3)

print((0,) \* 10)

выводит:

(1, 2, 3, 4, 5, 6, 7, 8)

(7, 8, 7, 8, 7, 8)

(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

Для генерации кортежей, состоящих строго из повторяющихся элементов, умножение на число — самый короткий и правильный путь.

Расширенные операторы += и \*= также можно использовать при работе с кортежами.

Следующий программный код:

a = (1, 2, 3, 4)

b = (7, 8)

a += b # добавляем к кортежу a кортеж b

b \*= 5 # повторяем кортеж b 5 раз

print(a)

print(b)

выводит:

(1, 2, 3, 4, 7, 8)

(7, 8, 7, 8, 7, 8, 7, 8, 7, 8)

### Встроенные функции sum(), min(), max()

Встроенная функция sum() принимает в качестве параметра кортеж чисел и вычисляет сумму его элементов.

Следующий программный код:

numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

print('Сумма всех элементов кортежа =', sum(numbers))

выводит:

Сумма всех элементов кортежа = 55

Встроенные функции min() и max() принимают в качестве параметра кортеж и находят минимальный и максимальный элементы соответственно.

Следующий программный код:

numbers = (3, 4, 10, 3333, 12, -7, -5, 4)

print('Минимальный элемент кортежа =', min(numbers))

print('Максимальный элемент кортежа =', max(numbers))

выводит:

Минимальный элемент кортежа = -7

Максимальный элемент кортежа = 3333

Функции min() и max() можно применять только к кортежам с одним типом данных. Если кортеж содержит разные типы данных, скажем целое число (int) и строку (str), то во время выполнения программы произойдет ошибка.

### Метод index()

Метод index() возвращает индекс первого элемента, значение которого равняется переданному в метод значению. Таким образом, в метод передается один параметр:

* value: значение, индекс которого требуется найти.

Если элемент в кортеже не найден, то во время выполнения происходит ошибка.

Следующий программный код:

names = ('Gvido', 'Roman' , 'Timur')

position = names.index('Timur')

print(position)

выведет:

2

Следующий программный код:

names = ('Gvido', 'Roman' , 'Timur')

position = names.index('Anders')

print(position)

приводит к ошибке:

ValueError: tuple.index(x): x not in tuple

Чтобы избежать таких ошибок, можно использовать метод index() вместе с оператором принадлежности in:

names = ('Gvido', 'Roman' , 'Timur')

if 'Anders' in names:

position = names.index('Anders')

print(position)

else:

print('Такого значения нет в кортеже')

### Метод count()

Метод count() возвращает количество элементов в кортеже, значения которых равны переданному в метод значению.

Таким образом, в метод передается один параметр:

* value: значение, количество элементов, равных которому,  нужно посчитать.

Если значение в кортеже не найдено, то метод возвращает 00.

Следующий программный код:

names = ('Timur', 'Gvido', 'Roman', 'Timur', 'Anders', 'Timur')

cnt1 = names.count('Timur')

cnt2 = names.count('Gvido')

cnt3 = names.count('Josef')

print(cnt1)

print(cnt2)

print(cnt3)

выведет:

3

1

0

Кортежи не поддерживают такие методы, как append(), remove(), pop(), insert(), reverse(), sort(), так как они изменяют содержимое.

## Вложенные кортежи

Подобно спискам, мы можем создавать вложенные кортежи.

Следующий программный код:

colors = ('red', ('green', 'blue'), 'yellow')

numbers = (1, 2, (4, (6, 7, 8, 9)), 10, 11)

print(colors[1][1])

print(numbers[2][1][3])

выводит:

blue

9

# Тема урока: кортежи

1. Перебор кортежей
2. Сравнение кортежей
3. Сортировка кортежей
4. Преобразование кортежа в список и строку
5. Упаковка кортежей
6. Распаковка кортежей
7. Присваивание кортежей

**Аннотация.** Урок посвящен кортежам (тип данных tuple).

## Перебор кортежей

Перебор элементов кортежа осуществляется точно так же как перебор элементов списка.

Для вывода **каждого** из элементов кортежа **на отдельной строке** можно использовать следующий код:

**Вариант 1.** Если нужны индексы элементов:

numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

for i in range(len(numbers)):

print(numbers[i])

**Вариант 2.** Если индексы не нужны:

numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

for num in numbers:

print(num)

Можно также использовать операцию **распаковки кортежа**.

Приведенный ниже код:

numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

languages = ('Python', 'C++', 'Java')

print(\*numbers)

print(\*languages, sep='\n')

выводит:

0 1 2 3 4 5 6 7 8 9 10

Python

C++

Java

## Сравнение кортежей

Кортежи можно сравнивать между собой.

Приведенный ниже код:

print((1, 8) == (1, 8))

print((1, 8) != (1, 10))

print((1, 9) < (1, 2))

print((2, 5) < (6,))

print(('a', 'bc') > ('a', 'de'))

выводит:

True

True

False

True

False

Обратите внимание: операции == и != применимы к любым кортежам, независимо от типов элементов. А вот операции <, >, <=, >= применимы только в том случае, когда соответствующие элементы кортежей имеют один тип.

Приведенный ниже код:

print((7, 5) < ('java', 'python'))

выводит:

TypeError: '<' not supported between instances of 'int' and 'str'

Сравнение кортежей происходит последовательно элемент за элементом, а если элементы равны — просматривается следующий элемент.

## Сортировка кортежей

Как мы помним, списки имеют метод sort(), который осуществляет сортировку на месте, то есть меняет порядок исходного списка. Поскольку кортежи по своей природе неизменяемы, то встроенного метода sort() они не содержат, тем не менее с помощью встроенной функции sorted() (не путать с списочным методом sort()) мы можем сортировать значения в кортежах.

Приведенный ниже код:

not\_sorted\_tuple = (34, 1, 8, 67, 5, 9, 0, 23)

print(not\_sorted\_tuple)

sorted\_tuple = tuple(sorted(not\_sorted\_tuple))

print(sorted\_tuple)

выводит:

(34, 1, 8, 67, 5, 9, 0, 23)

(0, 1, 5, 8, 9, 23, 34, 67)

Обратите внимание, что функция sorted() возвращает список, но с помощью функции tuple() мы приводим результат сортировки к кортежу.

Для сортировки кортежа можно воспользоваться явным преобразованием в список и использовать метод sort():

not\_sorted\_tuple = ('cc', 'aa', 'dd', 'bb')

tmp = list(not\_sorted\_tuple)

tmp.sort()

sorted\_tuple = tuple(tmp)

print(sorted\_tuple)

## Преобразование кортежа в список и строку

Часто на практике нам приходится преобразовывать кортежи в списки и в строки. Для этого используются функции и методы str(), list(), tuple(), join().

### Преобразование кортежа в список и наоборот

**Кортеж можно преобразовать в список** с помощью функции list().

Приведенный ниже код:

tuple1 = (1, 2, 3, 4, 5)

list1 = list(tuple1)

print(list1)

выводит:

[1, 2, 3, 4, 5]

**Список можно преобразовать в кортеж** с помощью функции tuple().

Приведенный ниже код:

list1 = [1, 17.8, 'Python']

tuple1 = tuple(list1)

print(tuple1)

выводит:

(1, 17.8, 'Python')

### Преобразование кортежа в строку и наоборот

**Кортеж можно преобразовать в строку** с помощью строкового метода join().

Приведенный ниже код:

notes = ('Do', 'Re', 'Mi', 'Fa', 'Sol', 'La', 'Si')

string1 = ''.join(notes)

string2 = '.'.join(notes)

print(string1)

print(string2)

выводит:

DoReMiFaSolLaSi

Do.Re.Mi.Fa.Sol.La.Si

Обратите внимание, что для применения строкового метода join() кортеж должен содержать именно строковые элементы. Если элементы кортежа отличны от строк, то требуется предварительно их преобразовать.

**Строку можно преобразовать в кортеж** с помощью функции tuple().

Приведенный ниже код:

letters = 'abcdefghijkl'

tpl = tuple(letters)

print(tpl)

выводит:

('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l')

Обратите внимание, что следующий код:

number = 12345

tpl = tuple(number)

print(tpl)

приведет к ошибке:

TypeError: 'int' object is not iterable

поскольку тип данных int не является итерируемым объектом. Для преобразования числа в кортеж, сначала нужно число преобразовать в строку, а уже потом использовать функцию tuple().

## Упаковка кортежей

**Упаковкой кортежа** называют присваивание его какой-либо переменной.

Приведенный ниже код:

tuple1 = (1, 2, 3)

tuple2 = ('b',)

tuple3 = ('red', 'green', 'blue', 'cyan')

print(type(tuple1))

print(type(tuple2))

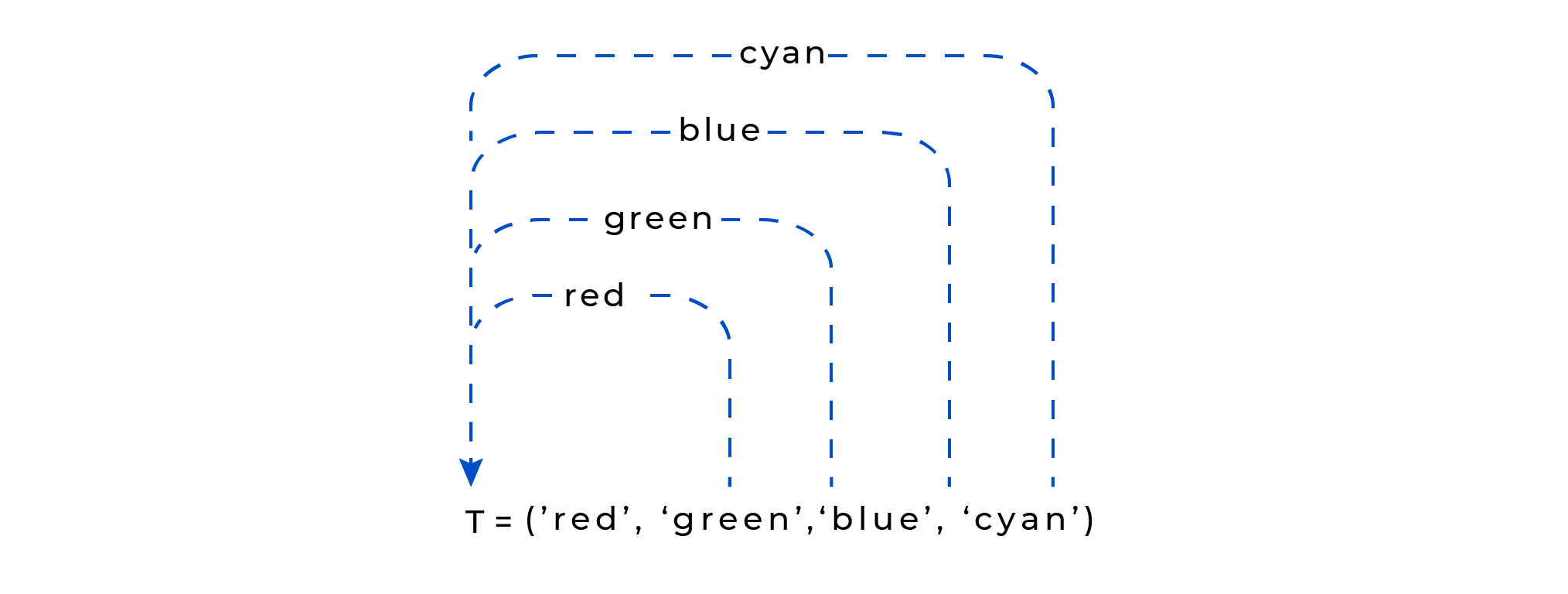
print(type(tuple3))

выводит:

<class 'tuple'>

<class 'tuple'>

<class 'tuple'>



Обратите внимание на то, что упаковка выполняется всегда, когда справа от знака равенства стоит больше одного значения.

Приведенный ниже код автоматически запакует 1, 2, 3 и 'b', в кортежи (1, 2, 3) и ('b', ) и присвоит их значения переменным tuple1 и tuple2:

tuple1 = 1, 2, 3

tuple2 = 'b',

print(type(tuple1))

print(type(tuple2))

выводит:

<class 'tuple'>

<class 'tuple'>

## Распаковка кортежей

Обратная операция, смысл которой в том, чтобы присвоить значения элементов кортежа отдельным переменным, называется **распаковкой кортежа**.

Приведенный ниже код:

colors = ('red', 'green', 'blue', 'cyan')

(a, b, c, d) = colors

print(a)

print(b)

print(c)

print(d)

выводит:

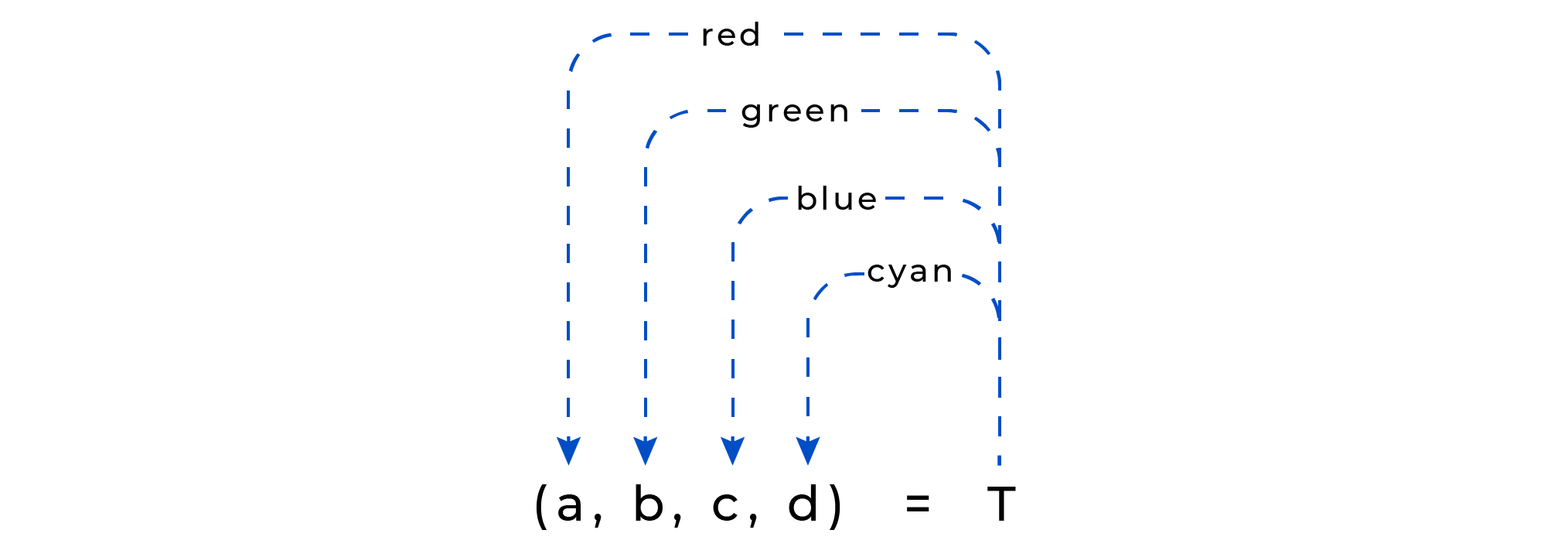
red

green

blue

cyan

В приведенном примере кортеж colors распаковывается в переменные a, b, c, d.



Мы можем опустить скобки слева от знака равенства:

colors = ('red', 'green', 'blue', 'cyan')

a, b, c, d = colors

Количество переменных должно совпадать с числом элементов в кортеже.

Приведенный ниже код:

colors = ('red', 'green', 'blue', 'cyan')

a, b = colors

приводит к ошибке:

ValueError: too many values to unpack

Аналогично, приведенный ниже код:

colors = ('red', 'green', 'blue')

a, b, c, d = colors

приводит к ошибке:

ValueError: not enough values to unpack (expected 4, got 3)

Однако, если необходимо получить лишь какие-то отдельные значения, то в качестве "ненужных" переменных позволено использовать символ нижнего подчеркивания \_.

Приведенный ниже код:

colors = ('red', 'green', 'blue')

a, b, \_ = colors

print(a)

print(b)

выводит:

red

green

Распаковка кортежей очень удобна на практике. По сути мы использовали ее, когда меняли местами значения двух переменных без использования временных переменных.

Приведенный ниже код:

a = 7

b = 17

a, b = b, a

print(a, b)

выводит:

17 7

Сначала вычисляются все значения справа, и лишь затем они кладутся в левую часть оператора присваивания. Поэтому можно менять местами значения переменных a и b, написав: a, b = b, a.

Приведенный ниже код:

a, b, c = 3, 2, 1

b, a, c = c, a, b

print(b, c, a)

выводит:

1 2 3

## \* при распаковке кортежей

Как мы знаем, если при распаковке кортежа число элементов слева и справа не совпадает, то возникает ошибка времени исполнения. Есть способ собрать сразу **несколько значений в одну переменную**. Это делается при помощи звездочки перед именем переменной.

Рассмотрим программный код:

a, b, \*tail = 1, 2, 3, 4, 5, 6

В этом случае в переменной a будет записана единица, в переменной b — двойка, а в переменной tail — список, состоящий из всех аргументов, которые не попали в предыдущие переменные. В данном случае tail будет равен [3, 4, 5, 6].

Учтите, что tail всегда будет списком, даже когда в него попадает лишь один элемент или даже ноль.

Приведенный ниже код:

a, b, \*tail = 1, 2, 3

print(tail)

выводит:

[3]

Приведенный ниже код:

a, b, \*tail = 1, 2

print(tail)

выводит:

[]

Звездочка может быть только у одного аргумента, но необязательно у последнего.

Приведенный ниже код:

\*names, surname = ('Стефани', 'Джоанн', 'Анджелина', 'Джерманотта')

print(names)

print(surname)

выводит:

['Стефани', 'Джоанн', 'Анджелина']

Джерманотта

Аргумент со звездочкой может стоять и посередине.

singer = ('Freddie', 'Bohemian Rhapsody', 'Killer Queen', 'Love of my life', 'Mercury')

name, \*songs, surname = singer

print(name)

print(songs)

print(surname)

выводит:

Freddie

['Bohemian Rhapsody', 'Killer Queen', 'Love of my life']

Mercury

## Примечания

**Примечание 1.** Если вы хотите распаковать единственное значение в кортеже, после имени переменной должна идти запятая.

Приведенный ниже код:

a = 1, # не распаковка, а просто присвоение

b, = 1, # распаковка

print(a)

print(b)

выводит:

(1,)

1

**Примечание 2.**Распаковывать можно не только кортеж, правая сторона может быть любой последовательностью (кортеж, строка или список).

info = ['timur', 'beegeek.org']

user, domain = info # распаковка списка

print(user)

print(domain)

a, b, c, d = 'math' # распаковка строки

print(a)

print(b)

print(c)

print(d)

выводит:

timur

beegeek.org

m

a

t

h

**Примечание 3.** Помимо метода split() строковый тип данных содержит метод partition(). Метод partition() принимает на вход один аргумент sep, разделяет строку при первом появлении sep и **возвращает кортеж**, состоящий из трех элементов: часть перед разделителем, сам разделитель и часть после разделителя. Если разделитель не найден, то кортеж содержит саму строку, за которой следуют две пустые строки.

Приведенный ниже код:

s1 = 'abc-de'.partition('-')

s2 = 'abc-de'.partition('.')

s3 = 'abc-de-fgh'.partition('-')

print(s1)

print(s2)

print(s3)

выведет:

('abc', '-', 'de')

('abc-de', '', '')

('abc', '-', 'de-fgh')

**Примечание 4.** С использованием кортежей многие алгоритмы приобретают достаточно краткую форму. Например, вычисление чисел Фибоначчи может выглядеть следующим образом:

n = int(input())

f1, f2 = 1, 1

for i in range(n):

print(f1)

f1, f2 = f2, f1 + f2

**Примечание 5.** Замечательная серия [статей](https://habr.com/ru/post/319164/) о коллекциях (list, tuple, str, set, dict) в Python.

# Тема урока: множества

1. Множества в математике
2. Числовые множества

**Аннотация.** Урок посвящен основным понятиям и определениям теории множеств.

## Множества в математике

В математике множество – совокупность объектов, понимаемых как единое целое.

При этом предполагается, что объекты данной совокупности можно отличать друг от друга и от объектов, не входящих в эту совокупность. Например, можно говорить:

* о множестве всех студентов данного курса;
* множестве всех языков программирования;
* множестве всех натуральных чисел;
* множестве всех точек данного отрезка.

Студенты данного курса, языки программирования, натуральные числа, точки данного отрезка – элементы соответствующих множеств.

### Обозначения

Обычно множества обозначают большими латинскими буквами: �,�,…*X*,*Y*,…, а элементы множеств – латинскими строчными буквами: �,�,…*x*,*y*,….

Запись )*x* ∈*X*(*x* ∈/*X*) означает, что *x* является (не является) элементом множества *X*.

Элементы множества указываются в фигурных скобках.

Рассмотрим множество вещей{худи,кепка,футболка}. Такое множество содержит три элемента.



### Конечные и бесконечные множества

Рассмотрим три множества:

* *A*={*a*,*b*,*c*,…,*z*} – множество букв английского алфавита;
* *B*={Тимур,Руслан,Роман,Оля} – множество имён авторов данного курса;
* *N*={1,2,3,4,5,…} – множество натуральных чисел.

Первые два множества содержат конечное количество элементов и являются конечными множествами. Третье же множество содержит бесконечно много элементов, поэтому и называется бесконечным множеством.

Мы используем символ ……, чтобы показать, что элементы множества продолжаются. Другими словами, символ …… означает "и так далее". Символ …… можно использовать как в конечных, так и в бесконечных множествах.

### Равенство множеств

Если два множества *X* и *Y* состоят из одних и тех же элементов, то они называются равными *X* =*Y*.

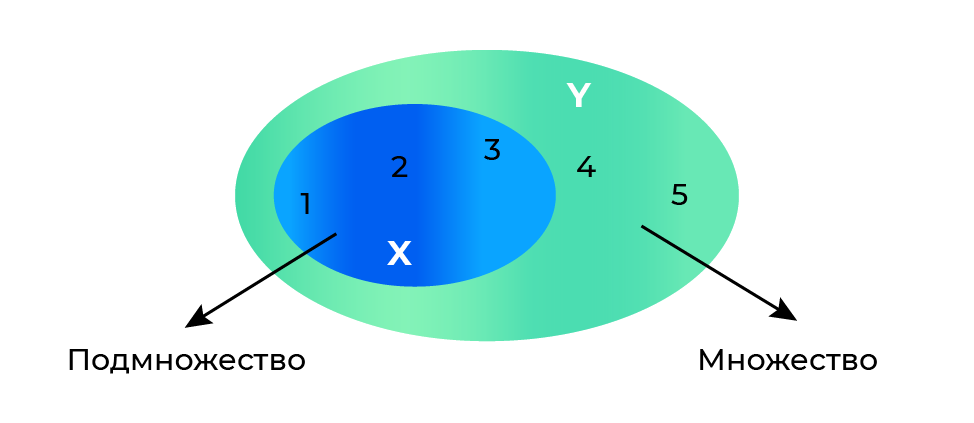
Например, если *X* ={*a*,*b*,*c*,*d*} и *Y* ={*b*,*d*,*c*,*a*}, то *X*=*Y*. Обратите внимание, порядок расположения элементов в записи множеств при их сравнении во внимание не принимается.

Еще один пример,  *X*={1,2,3,4,5} и *Y*={множествонатуральныхчиселменьших6}. Очевидно, такие множества содержат абсолютно одинаковые элементы, поэтому равны.

### Подмножество и надмножество

Если все элементы множества *X* принадлежат также и множеству *Y*, то говорят, что *X* является **подмножеством** *Y*, а записывается это так: *X* ⊂*Y*.

Например,  *X*={1,2,3},*Y*={1,2,3,4,5}. Так как все элементы множества *X* содержатся в множестве *Y*, то мы говорим, что множество *X* является подмножеством множества *Y*.



Множество *X* ={1,2,3,6} не является подмножеством множества *Y*={1,2,3,4,5}, так как элемент 6 не содержится в *Y*.

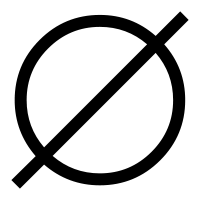
Если множество *X* является подмножеством множества *Y*, то также говорят, что множество *Y* является **надмножеством** множества *X*, а записывается это так: *Y* ⊃*X*.

Заметим, что любое множество также является подмножеством самого себя. Про такое подмножество говорят **нестрогое подмножество:**

* множество 1,2,3} является **нестрогим** подмножеством множества 1,2,3};
* множество {1,2,3} является **строгим** подмножеством множества {1,2,3,4}.

### Пустое множество

Для удобства работы с множествами и записи с их помощью различных математических высказываний, вводится понятие множества, не содержащего ни одного элемента. Оно называется пустым множеством и обозначается ∅∅.



Рассмотрим множество клавиш пианино, находящихся на гитаре. Очевидно такое множество не содержит элементов и является пустым.



Примеры пустых множеств:

* множество лошадей, пасущихся на луне;
* множество точек пересечения двух параллельных прямых на плоскости;
* множество квадратных уравнений, имеющих больше двух корней;
* множество людей, не любящих данный курс (ха-ха).

   Пустое множество является подмножеством любого множества.

## Примечания

**Примечание 1.** Множества – неупорядоченные совокупности, то есть, неважно, в каком порядке указаны элементы множества.

**Примечание 2.** Если множество *X* конечно, то через ∣*X*∣ обозначается количество элементов множества *X*.

**Примечание 3.** Если множество *X* содержит *n* элементов, то оно имеет 2*n* подмножеств, включая пустое множество. Например, множество *X*={*a*,*b*,*c*} содержит 33 элемента и имеет 88 подмножеств:

1. {∅}{∅};
2. {*a*};
3. {*b*};
4. {*c*};
5. {*a*,*b*};
6. {*a*,*c*};
7. {*b*,*c*};
8. {*a*,*b*,*c*}.

**Примечание 4.** Раздел математики, занимающийся множествами, называется теорией множеств. Возникла эта теория во второй половине *XIX* века, главным образом в трудах немецкого математика [Г. Кантора](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BD%D1%82%D0%BE%D1%80,_%D0%93%D0%B5%D0%BE%D1%80%D0%B3). Кантор определял множество как "любое собрание определенных и различимых между собой объектов нашей интуиции или интеллекта, мыслимое как единое целое".

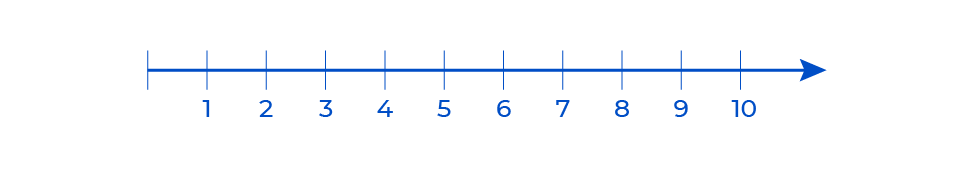
## Числовые множества

К основным числовым множествам математики относятся:

* множество натуральных чисел;
* множество целых чисел;
* множество рациональных чисел;
* множество вещественных чисел;
* множество комплексных чисел.

### Натуральные числа

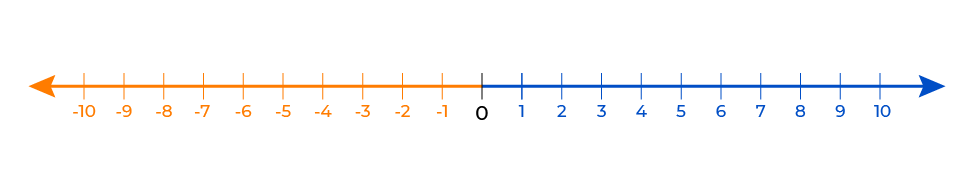
Исторически первыми появились натуральные числа, предназначенные для подсчёта материальных объектов (овец, кур, монет и т.д.). Множество натуральных чисел обозначается буквой N и содержит следующие числа:N={1,2,3,4,5,…}.



   Обратите внимание, число ноль не является натуральным числом.

### Целые числа

Если к множеству натуральных чисел N присоединить те же числа с противоположным знаком и ноль, то получится множество целых чисел. Множество целых чисел обозначается буквой Z и содержит следующие числа: Z={0,±1,±2,±3,±4,±5,…}.Множество натуральных чисел является подмножеством множества целых чисел, поскольку каждый элемент множества N принадлежит множеству Z.



   Множество натуральных чисел – подмножество множества целых чисел N⊂Z.

### Рациональные числа

Рациональным числом в математике называется любое число, представимое в виде частного двух целых чисел с отличным от нуля знаменателем. Множество рациональных чисел обозначается буквой �Q и содержит следующие числа: Q={*nm*​,*m*∈Z,*n* ∈N}.

Множество целых чисел является подмножеством множества рациональных чисел, так как любое целое число можно представить в виде дроби со знаменателем, равным 11.

Любое рациональное число это либо конечная, либо бесконечная периодическая десятичная дробь. К примеру:

* 12=0.521​=0.5 – конечная непериодическая дробь;
* 38=0.37583​=0.375 – конечная непериодическая дробь;
* 13=0.(3)31​=0.(3) – бесконечная периодическая десятичная дробь;
* 711=0.(63)117​=0.(63) – бесконечная периодическая десятичная дробь.

### Иррациональные числа

Не все числа в математике можно представить в виде рационального числа. Примером служат числа:

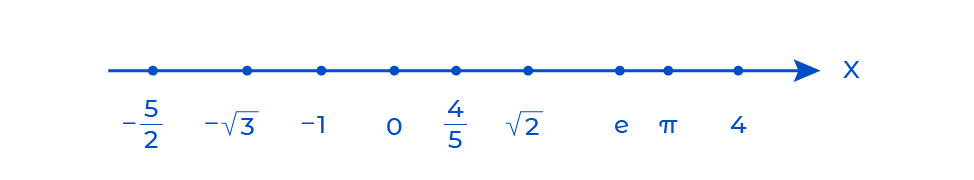
* *π*≈3.1415926535…;
* �*e*≈2.71828182845….

Такие числа называются иррациональными и являются бесконечными непериодическими дробями. Иными словами, в «бесконечных хвостах» иррациональных чисел нет никакой закономерности. Иррациональные числа часто обозначают буквой I.

### Вещественные числа

Объединение рациональных и иррациональных чисел образует множество вещественных чисел. Множество вещественных чисел R определяется так:R=Q∪I,где символ ∪ – означает объединение множеств.

Геометрическая интерпретация множества вещественных чисел – это числовая прямая:



Каждому вещественному числу соответствует определённая точка числовой прямой, и наоборот – каждой точке числовой прямой обязательно соответствует некоторое вещественное число.

   Множество **вещественных** чисел также называют множеством **действительных** чисел.

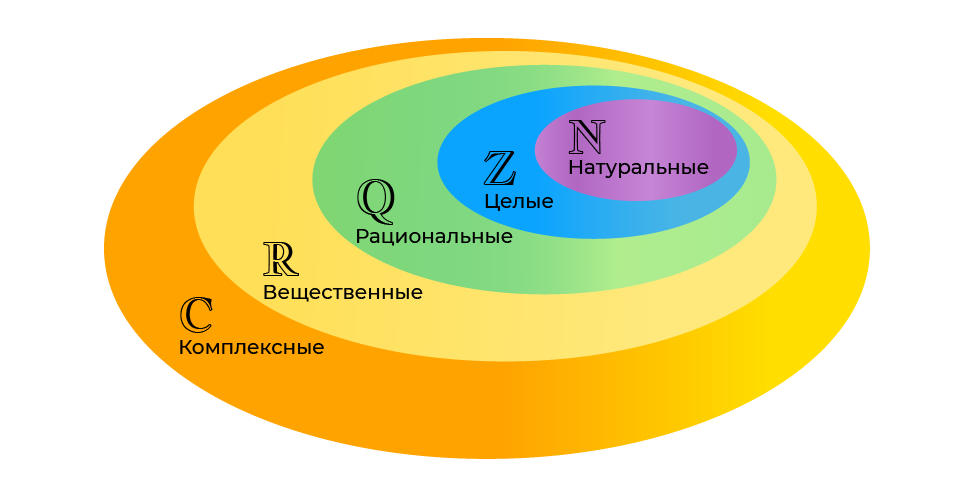
### Множество комплексных чисел

Комплексным числом в математике называется любое число, представимое в виде *a*+*bi*, где *a* и *b* – вещественные числа, а *i*=−1​ – мнимая единица. Множество комплексных чисел обозначается буквой C.

Множество вещественных чисел является подмножеством множества комплексных чисел, так как любое вещественное число можно представить в виде *a*+*bi*, где *b* =0.

### Графическая иллюстрация числовых множеств

Несложно заметить, что каждое следующее множество является надмножеством предыдущего множества, так как содержит все его элементы: N⊂Z⊂Q ⊂R ⊂C.



### Числовые множества при решении уравнений

Рассмотрим алгебраические уравнения, корнями которых являются натуральные, целые, рациональные, вещественные и комплексные числа:

|  |  |  |  |
| --- | --- | --- | --- |
| **Уравнение** | **Корни** | **Множество чисел** | **Обозначение** |
| *x*−3=0 | *x*=3 | Натуральные числа | N |
| *x*+7=0 | *x*=−7 | Целые числа | Z |
| 4*x*−1=0 | *x*=-1/4​ | Рациональные числа | Q |
| *x*2 −2=0 | *x*=±2​ | Вещественные числа | R |
| *x*2 +1=0 | *x*=±*i* | Комплексные числа | C |

## Примечания

**Примечание 1.** Почитать подробнее про числовые множества можно [тут](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%BE#%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE%D0%B2%D1%8B%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%B0).

**Примечание 2.** Почитать подробнее про комплексные числа можно [тут](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BB%D0%B5%D0%BA%D1%81%D0%BD%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE).

# Тема урока: множества

1. Операции над множествами
2. Диаграммы Эйлера-Вена

**Аннотация.** Урок посвящен операциям над множествами и диаграммам Эйлера-Венна.

## Диаграммы Эйлера-Венна

Диаграммы Эйлера-Венна – геометрическое представление множеств. Большой прямоугольник представляет универсальное множество �*U*, а круги в нем – отдельные множества. Круги пересекаются в соответствии с условиями задачи. Точки внутри областей диаграммы — элементы соответствующих множеств. На диаграмме можно заштриховать образованные при пересечении кругов множества.

## Операции над множествами

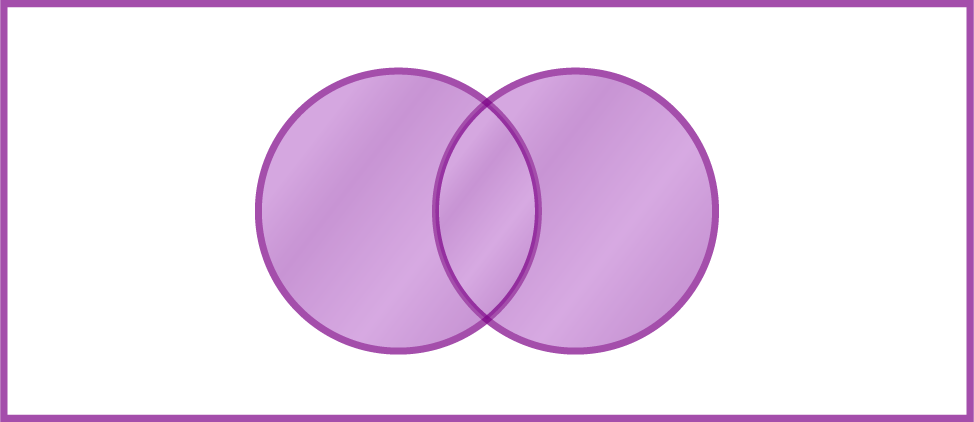
Операции над множествами выполняются для получения новых множеств из уже существующих.

Основные операции над множествами:

* объединение;
* пересечение;
* разность;
* симметрическая разность;
* дополнение.

### Объединение множеств

**Объединение множеств**– множество, состоящее **из элементов, принадлежащих хотя бы одному**из объединяемых множеств.



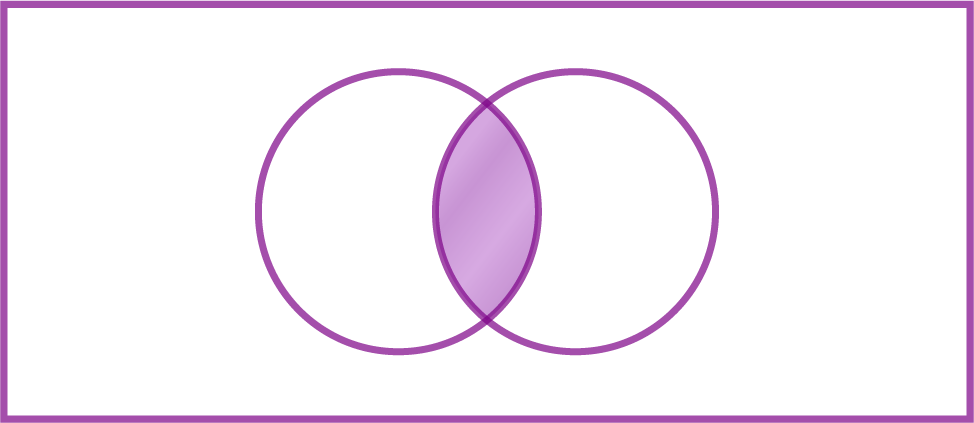
Для объединения множеств используется символ ∪.

Например, если *X*={1,2,3,4,5},*Y*={3,4,7,8,9}, то *X*∪*Y*={1,2,3,4,5,7,8,9}.

   Часто операцию **объединения** множеств отождествляют с операцией **сложения**.

### Пересечение множеств

**Пересечение** множеств – множество, состоящее из элементов, **принадлежащих одновременно каждому** из пересекающихся множеств.



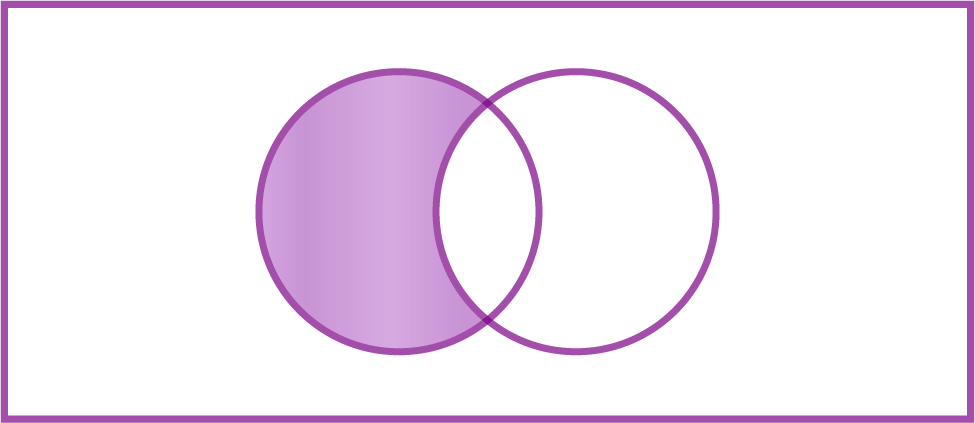
Для пересечения множеств используется символ ∩.

Например, если *X*={1,2,3,4,5},*Y*={3,4,7,8,9}, то *X*∩*Y*={3,4}.

   Часто операцию **пересечения** множеств отождествляют с операцией **умножения**.

### Разность множеств

**Разность** множеств – множество, в которое входят только элементы первого множества, не входящие во второе множество.



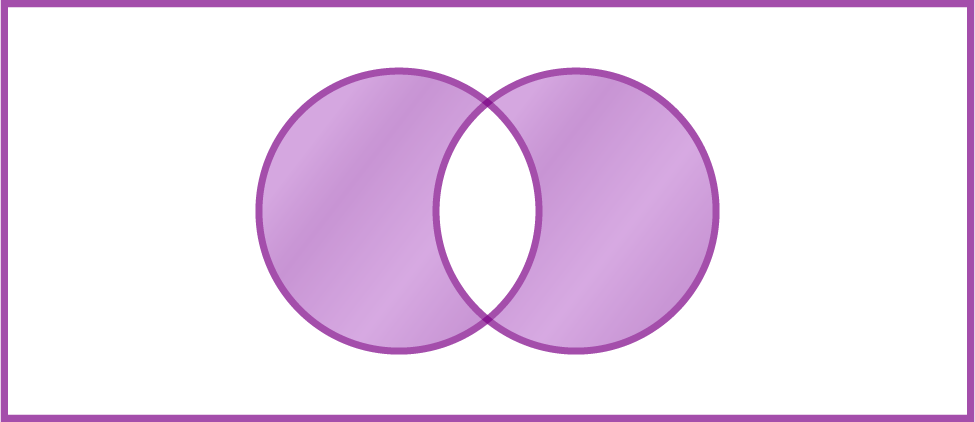
Для разности множеств используется символ ∖.

Например, если *X*={1,2,3,4,5},*Y*={3,4,7,8,9}, то *X*∖ *Y*={1,2,5}.

### Симметрическая разность

**Симметрическая разность** множеств – множество, включающее все элементы исходных множеств, не принадлежащие одновременно обоим исходным множествам.

Другими словами, симметрическая разность это множество (*X*∪*Y*)∖(*X*∩*Y*).

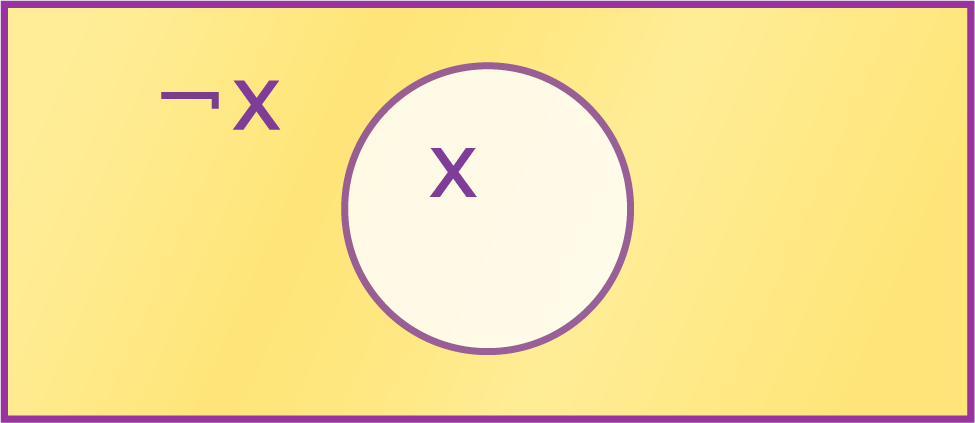


Для симметрической разности множеств используется символ △.

Например, если *X*={1,2,3,4,5},*Y*={3,4,7,8,9}, то *X*△ *Y*={1,2,5,7,8,9}.

### Дополнение

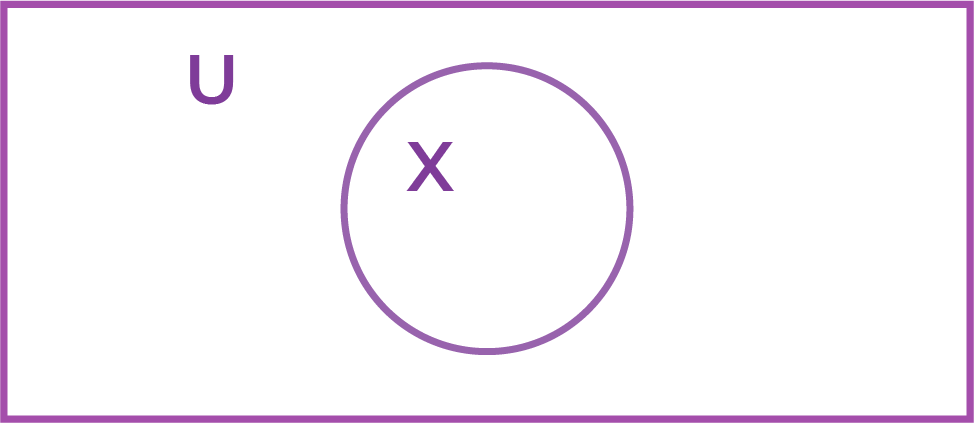
**Дополнение** множества – множество всех элементов, в нем не содержащихся.



Для операции дополнения множества используется символ ¬¬.

## Примечания

**Примечание 1.** Предположим, что изучается некоторая область знаний. Множество всех элементов исследуемой области называется универсальным. На диаграммах универсальное множество обычно изображается множеством точек некоторого прямоугольника плоскости, а принадлежащие ему подмножества – кругами внутри прямоугольника.



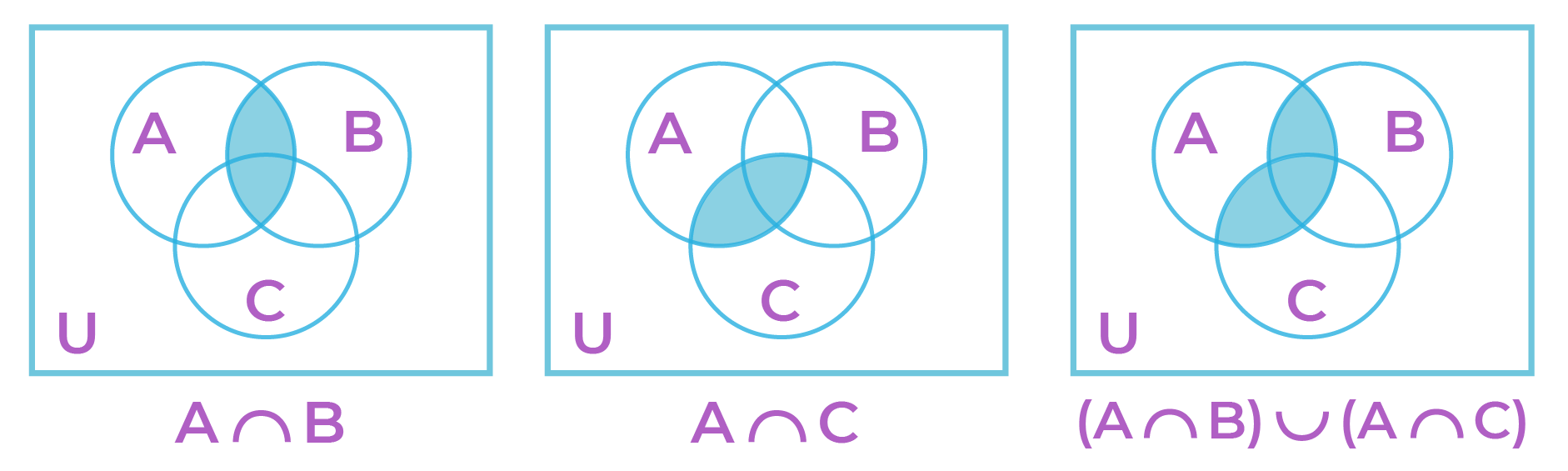
**Диаграммы Эйлера-Венна при решении задач**

Диаграммы Эйлера-Венна применяют для доказательства формул, решения текстовых задач и во многих других случаях.

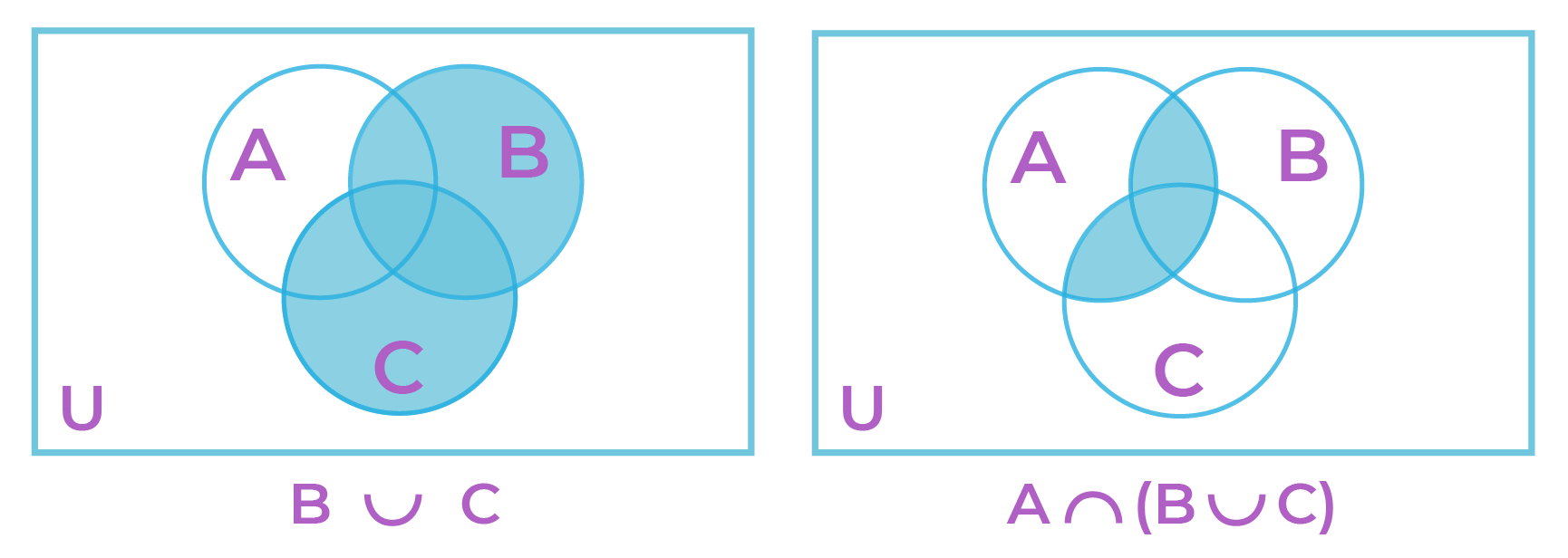
**Задача 1.** Докажите формулу  (*A* ∩*B*)∪(*A*∩*C*)=*A*∩(*B*∪*C*).

**Решение.** Используя диаграмму Эйлера-Венна, покажем, что обеим частям равенства соответствуют одна и та же область.

**Левая часть:**



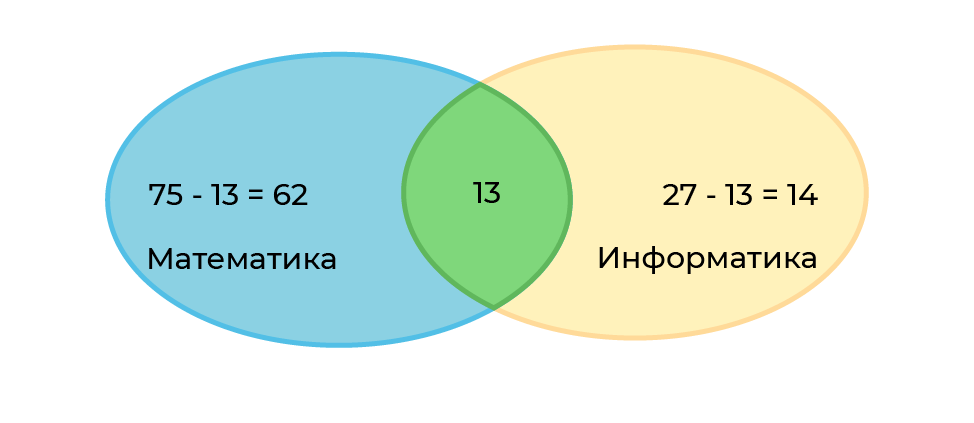
**Правая часть:**



Так как левой и правой частям формулы соответствует одна и та же область на диаграмме Эйлера-Венна, то формула верна.

**Задача 2.** Каждый ученик онлайн-школы BEEGEEK изучает или математику или информатику, или и то и другое одновременно. Всего 75 учеников изучает математику, а 27 – информатику и только 13 – оба предмета. Сколько учеников учится в онлайн-школе BEEGEEK?

**Решение.** Введем обозначения: множество учеников, изучающих математику – М, информатику – И. Изображаем множества на диаграмме Эйлера-Венна в наиболее общем случае.



Рассуждаем следующим образом: оба предмета изучают 13 учеников. Значит только математику изучают 75−13=62 ученика, только информатику изучают 27−13=14 ученика.

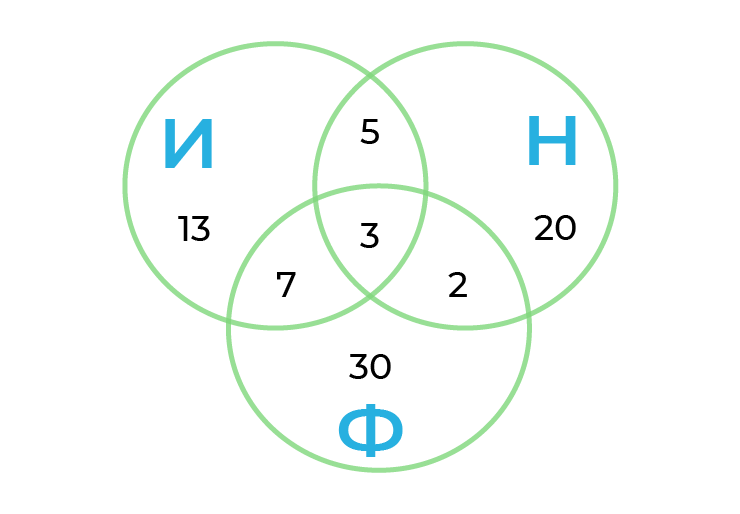
Таким образом всего в школе учится 62+13+14=89 учеников.

**Ответ:** 8989.

**Задача 3.** Опрос  100 студентов дал следующие результаты по количеству изучающих разные иностранные языки: испанский – 28, немецкий – 30, французский – 42, испанский и немецкий – 8, испанский и французский – 10, немецкий и французский – 5, все три языка – 3. Ответьте на вопросы:

1. Сколько студентов не изучает ни одного языка?
2. Сколько студентов изучает один французский язык?

**Решение.** Введем обозначения: множество студентов, изучающих немецкий язык – Н, французский – Ф, испанский – И. Изображаем множества на диаграмме Эйлера-Венна в наиболее общем случае.



Рассуждаем следующим образом: все три языка изучают 33 студента. Значит, одновременно изучают только немецкий и французский 5−3=2, одновременно изучают только немецкий и испанский  8−3=5, одновременно изучают только французский и испанский 10−3=7.

Поскольку всего 28 студентов изучают испанский, то число студентов изучающих только испанский язык 28−3−7−5=13. Аналогично находим число студентов изучающих только немецкий язык 30−3−2−5=20, число студентов изучающих только французский язык 42−3−7−2=30. Находим число студентов, изучающих иностранные языки 13+20+30+3+5+7+2=80. Таким образом, число студентов не изучающих язык, 100−80=20.

**Ответ.** 2020 студентов не изучают ни одного языка, 3030 студентов изучает только французский язык.

**Задача 4.** В языке запросов поискового сервера для обозначения логической операции «ИЛИ» используется символ «|», а для обозначения логической операции «И» — символ «&». Приведены запросы к поисковому серверу. Для каждого запроса указан его код — соответствующая буква от А до Г. Расположите коды запросов слева направо в порядке возрастания количества страниц, которые нашел поисковый сервер по каждому запросу.

| **Код** | **Запрос** |
| --- | --- |
| А | Есенин & Фет |
| Б | (Есенин & Фет) | Тютчев |
| В | Есенин & Фет & Тютчев |
| Г | Есенин | Фет | Тютчев |

**Решение.** Изобразим диаграммы Эйлера-Венна для всех запросов:

|  |  |  |  |
| --- | --- | --- | --- |
| A | Б | В | Г |
| Есенин & Фет | (Есенин & Фет) | Тютчев | Есенин & Фет & Тютчев | Есенин | Фет | Тютчев |
|  |  |  |  |

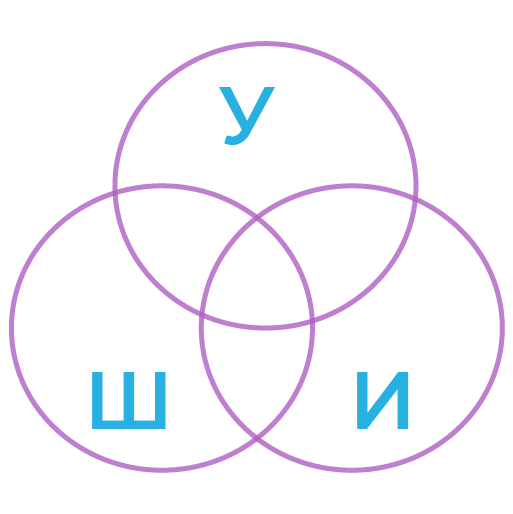
**Ответ:** ВАБГ.

**Задача 5.** В таблице приведены запросы и количество найденных по ним страниц некоторого сегмента сети Интернет.

| **Запрос** | **Найдено страниц (в тысячах)** |
| --- | --- |
| Уэльс & Шотландия | Ирландия | 450 |
| Уэльс & Шотландия | 213 |
| Уэльс & Шотландия & Ирландия | 87 |

Какое количество страниц (в тысячах) будет найдено по запросу **Ирландия**?

**Решение.** Введем обозначения: Уэльс – У, Шотландия – Ш, Ирландия – И. Изображаем множества на диаграмме Эйлера-Венна в наиболее общем случае.



Отметим известные и неизвестные составные высказывания

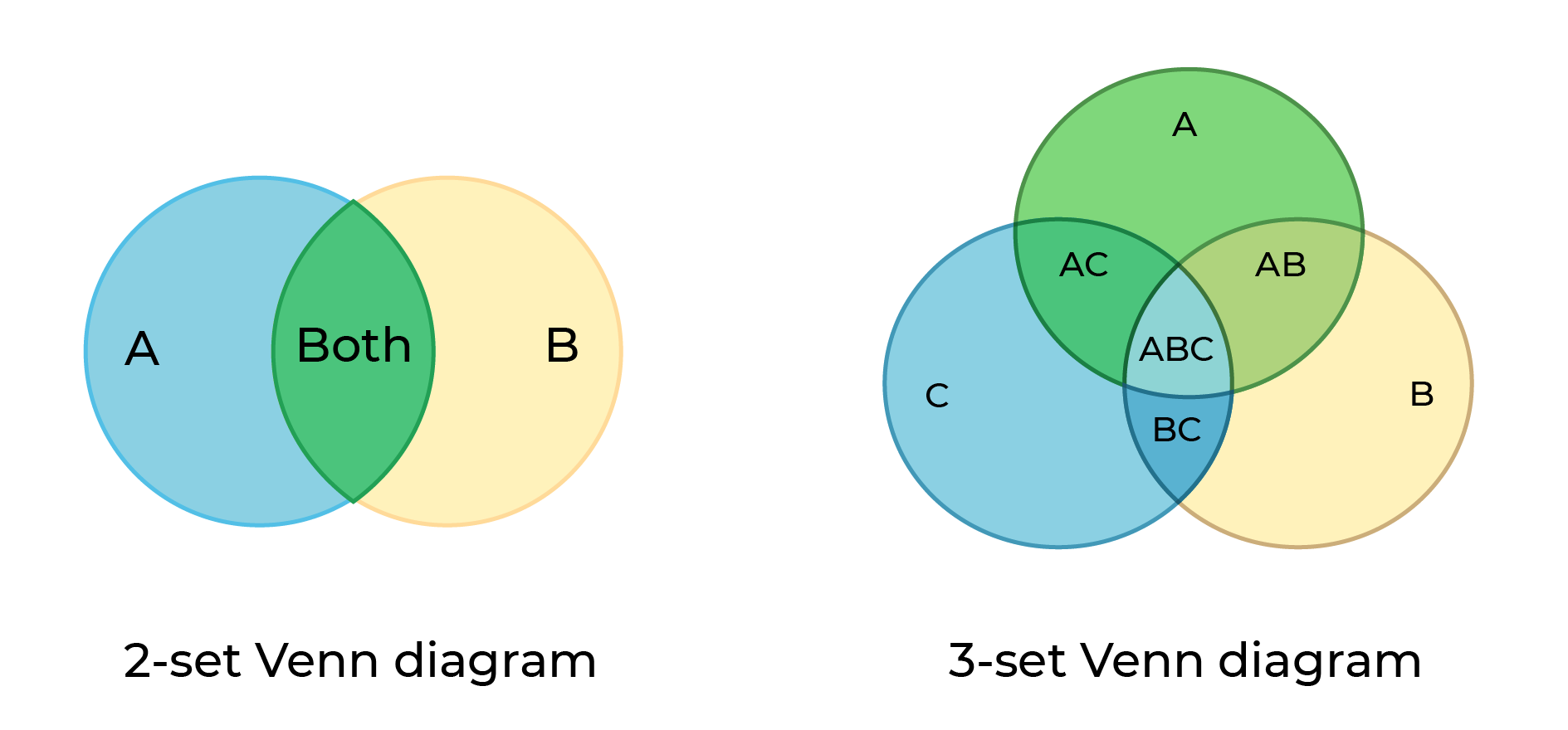
|  |  |  |  |
| --- | --- | --- | --- |
| Уэльс & Шотландия | Ирландия | Уэльс & Шотландия | Уэльс & Шотландия & Ирландия | Ирландия |
|  |  |  |  |

Итак, по запросу Ирландия будет найдено 450−213+87=324 страниц.

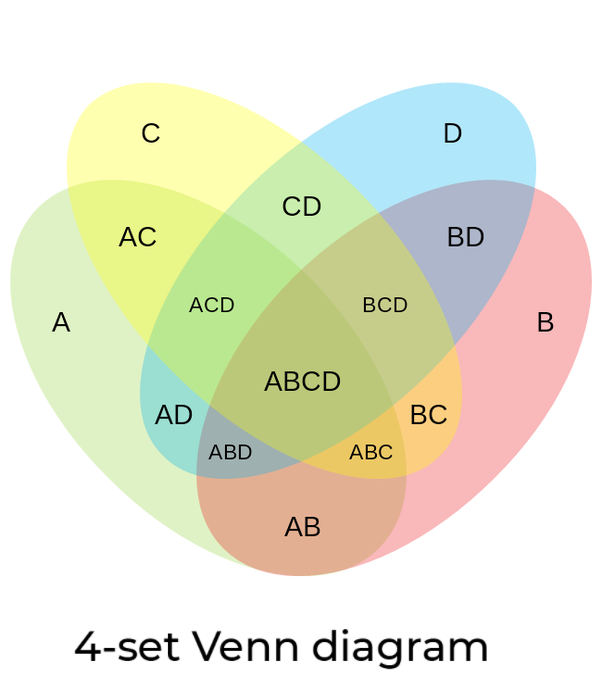
**Ответ:** 324.

**Примечания**

**Примечание 1.** Диаграммы Эйлера-Венна для 2,3 множеств выглядят так:



**Примечание 2.** Диаграмма Эйлера-Венна для 4 множеств выглядят так:



Для 5 и более множеств диаграммы Эйлера-Венна неудобны.

# Тема урока: множества в Python

1. Создание множеств
2. Пустые множества
3. Встроенная функция set()
4. Вывод множеств

**Аннотация.** Начинаем изучение множеств в Python (тип данных set). Этот тип данных аналогичен математическим множествам, он поддерживает быстрые операции проверки наличия элемента в множестве, добавления и удаления элементов, операции объединения, пересечения и многие другие.

## Множества

В прошлых уроках мы изучили три типа коллекций в Python:

* списки – изменяемые коллекции элементов;
* строки – неизменяемые коллекции символов;
* кортежи – неизменяемые коллекции элементов.

Следующий тип коллекций (наборов данных) – **множество**.

Множество – структура данных, организованная так же, как математические множества.

Важно знать:

* все элементы множества различны (уникальны), два элемента не могут иметь одинаковое значение;
* множества неупорядочены, то есть элементы не хранятся в каком-то определенном порядке;
* элементы множества должны относиться к неизменяемым типам данных;
* хранящиеся в множестве элементы могут иметь разные типы данных.

Структура данных (data structure) — программная единица, позволяющая **хранить и обрабатывать** множество однотипных и/или логически связанных данных.

### Создание множества

Чтобы создать множество, нужно перечислить его элементы через запятую в фигурных скобках:

numbers = {2, 4, 6, 8, 10}

languages = {'Python', 'C#', 'C++', 'Java'}

Множество numbers состоит из 55 элементов, и каждый из них — **целое число.**

Множество languages состоит из 44 элементов, каждый из которых — **строка.**

Множества могут содержать значения **разных типов данных**:

info = {'Timur', 1992, 61.5}

Множество info содержит строковое значение, целое число и число с плавающей точкой.

   Не создавайте переменные с именем set. Это очень плохая практика.

### Пустое множество

Создать пустое множество можно с помощью встроенной функции, которая называется set():

myset = set() # пустое множество

Обратите внимание — создать пустое множество с помощью пустых фигурных скобок нельзя:

myset = {} # создается словарь

С помощью пустых фигурных скобок создаются словари, так сложилось исторически, словари появились в Python раньше, чем множества.

   Пустое множество создаётся исключительно через set()

### Вывод множества

Для вывода всего множества можно использовать функцию print():

numbers = {2, 4, 6, 8, 10}

languages = {'Python', 'C#', 'C++', 'Java'}

mammals = {'cat', 'dog', 'fox', 'elephant'}

print(numbers)

print(languages)

print(mammals)

Функция print() выводит на экран элементы множества, в фигурных скобках, разделенные запятыми:

{2, 4, 6, 8, 10}

{'C#', 'Python', 'Java', 'C++'}

{'dog', 'cat', 'fox', 'elephant'}

Обратите внимание: при выводе множества порядок элементов может отличаться от существовавшего при его создании, поскольку множества — неупорядоченные коллекции данных.

### Встроенная функция set()

Встроенная функция set() помимо создания пустого множества может преобразовывать некоторые типы объектов в множества.

В функцию set() можно передать один аргумент. Передаваемый аргумент должен быть итерируемым объектом, таким как список, кортеж или строковое значение. Отдельные элементы объекта, передаваемого в качестве аргумента, становятся элементами множества:

myset1 = set(range(10)) # множество из элементов последовательности

myset2 = set([1, 2, 3, 4, 5]) # множество из элементов списка

myset3 = set('abcd') # множество из элементов строки

myset4 = set((10, 20, 30, 40)) # множество из элементов кортежа

Пустое множество также можно создать передав функции set() в качестве аргумента пустой список, строку или кортеж:

emptyset1 = set([]) # пустое множество из пустого списка

emptyset2 = set('') # пустое множество из пустой строки

emptyset3 = set(()) # пустое множество из пустого кортежа

## Дубликаты при создании множеств

Множества не могут содержать повторяющиеся элементы. Если в функцию set() передать аргумент, содержащий повторяющиеся элементы, то в множестве появится только один из этих повторяющихся элементов.

Приведенный ниже код:

myset1 = {2, 2, 4, 6, 6}

​myset2 = set([1, 2, 2, 3, 3])

myset3 = set('aaaaabbbbccccddd')

print(myset1)

print(myset2)

print(myset3)

выводит (порядок элементов может отличаться):

{2, 4, 6}

{1, 2, 3}

{'b', 'c', 'd', 'a'}

Если требуется создать множество, в котором каждый элемент — строковое значение, содержащее более одного символа, то используем код:

myset = set(['aaa', 'bbbb', 'cc'])

print(myset)

Приведенный выше код выводит (порядок элементов может отличаться):

{'bbbb', 'aaa', 'cc'}

Если же создать множество следующим образом:

myset = set('aaa bbbb cc')

print(myset)

то мы получим (порядок элементов может отличаться):

{' ', 'c', 'a', 'b'}

   Обратите внимание на наличие пробела в качестве элемента множества myset.

## Примечания

**Примечание 1.** Элементы множества могут принадлежать любому **неизменяемому типу данных**, быть числами, строками, кортежами. Элементы изменяемых типов данных не могут входить в множества, в частности, нельзя сделать элементом множества список или другое множество. Требование неизменяемости элементов множества накладывается особенностями представления множеств в Python.

Приведенный ниже код:

myset1 = {1, 2, [5, 6], 7} # множество не может содержать список

myset2 = {1, 2, {5, 6}, 7} # множество не может содержать множество

приводит к ошибке:

TypeError: unhashable type: 'list'

TypeError: unhashable type: 'set'

Однако приведенный ниже код:

myset = {1, 2, (5, 6), 7} # множество может содержать кортеж

работает как полагается.

**Примечание 2.** Документация по множествам доступна по [ссылке](https://docs.python.org/3/library/stdtypes.html#set).

**Примечание 3.** Отличная [статья](https://habr.com/ru/post/516858/) с хабра про множества.

# Тема урока: множества в Python

1. Встроенные функции len(),sum(),min(),max()
2. Оператор принадлежности in
3. Перебор множеств
4. Форматированный вывод множеств
5. Сравнение множеств

**Аннотация.** В этом уроке мы изучим основной функционал при работе с множествами.

## Основы работы с множествами

Работа с множествами очень сильно напоминает работу со списками, поскольку и множества, и списки содержат отдельные элементы, хотя элементы множества уникальны, а списки могут содержать повторяющиеся элементы. Многое из того, что мы делали со списками, доступно и при работе с множествами.

### Функция len()

**Длиной множества** называется количество его элементов. Чтобы посчитать длину множества используют встроенную функцию len() (от слова length – длина).

Следующий программный код:

myset1 = {2, 2, 4, 6, 6}

myset2 = set([1, 2, 2, 3, 3, 4, 4, 5, 5])

myset3 = set('aaaaabbbbccccddd')

print(len(myset1))

print(len(myset2))

print(len(myset3))

выведет:

3

5

4

### Оператор принадлежности in

Оператор in позволяет проверить, содержит ли множество некоторый элемент.

Рассмотрим следующий код:

numbers = {2, 4, 6, 8, 10}

if 2 in numbers:

print('Множество numbers содержит число 2')

else:

print('Множество numbers не содержит число 2')

Такой код проверяет, содержит ли множество numbers число 22 и выводит соответствующий текст:

Множество numbers содержит число 2

Мы можем использовать оператор in вместе с логическим оператором not. Например

numbers = {2, 4, 6, 8, 10}

if 0 not in numbers:

print('Множество numbers не содержит нулей')

Оператор принадлежности in работает **очень быстро** на множествах, намного быстрее чем на списках. Поэтому если требуется часто осуществлять поиск в коллекции уникальных данных, то множество – подходящий выбор.

### Встроенные функции sum(), min(), max()

Встроенная функция sum() принимает в качестве аргумента множество чисел и вычисляет сумму его элементов.

Следующий программный код:

numbers = {2, 2, 4, 6, 6}

print('Сумма всех элементов множества =', sum(numbers))

выводит:

Сумма всех элементов множества = 12

Встроенные функции min() и max() принимают в качестве аргумента множество и находят минимальный и максимальный элементы соответственно.

Следующий программный код:

numbers = {2, 2, 4, 6, 6}

print('Минимальный элемент =', min(numbers))

print('Максимальный элемент =', max(numbers))

выводит:

Минимальный элемент = 2

Максимальный элемент = 6

## Примечания

**Примечание 1.** Индексация и срезы **недоступны** для множеств.

**Примечание 2.** Операция конкатенации + и умножения на число \* **недоступны** для множеств.

## Перебор элементов множества

Перебор элементов множества осуществляется точно так же, как и перебор элементов списка, то есть с помощью цикла for.

Для вывода элементов множества **каждого на отдельной строке** можно использовать следующий код:

numbers = {0, 1, 1, 2, 3, 3, 3, 5, 6, 7, 7}

for num in numbers:

print(num)

Такой код выведет (порядок элементов может отличаться):

0

1

2

3

5

6

7

Мы также можем использовать операцию **распаковки множества**.

Приведенный ниже код:

numbers = {0, 1, 1, 2, 3, 3, 3, 5, 6, 7, 7}

print(\*numbers, sep='\n')

выводит (порядок элементов может отличаться):

0

1

2

3

5

6

7

Не стоит забывать, что множества – неупорядоченные коллекции, поэтому полагаться на порядок вывода элементов не стоит. Если нужно гарантировать порядок вывода элементов (по возрастанию / убыванию), то необходимо воспользоваться встроенной функцией sorted().

Приведенный ниже код:

numbers = {0, 1, 1, 2, 3, 3, 3, 5, 6, 7, 7}

sorted\_numbers = sorted(numbers)

print(\*sorted\_numbers, sep='\n')

будет **гарантированно** выводить элементы множества в порядке возрастания.

Обратите внимание на то, что функция sorted() возвращает отсортированный список, а не множество. Не путайте встроенную функцию sorted() и списочный метод sort(). Множества **не содержат** метода sort().

## Сравнение множеств

Множества можно сравнивать между собой. Равные множества имеют одинаковую длину и содержат равные элементы. Для сравнения множеств используются операторы == и !=.

Приведенный ниже код:

myset1 = {1, 2, 3, 3, 3, 3}

myset2 = {2, 1, 3}

myset3 = {1, 2, 3, 4}

print(myset1 == myset2)

print(myset1 == myset3)

print(myset1 != myset3)

выводит:

True

False

True

## Примечания

**Примечание 1.** Встроенная функция sorted() имеет опциональный параметр reverse. Если установить этот параметр в значение True, произойдет сортировка по убыванию.

Приведенный ниже код:

numbers = {0, 1, 1, 2, 3, 3, 3, 5, 6, 7, 7}

sortnumbers = sorted(numbers, reverse=True)

print(\*sortnumbers, sep='\n')

гарантированно выводит:

7

6

5

3

2

1

0

**Примечание 2.** Код для работы с множествами нужно писать так, чтобы результат его выполнения не зависел от расположения элементов и был одинаковым при любом порядке обхода, последовательного обращения ко всем элементам.

# Тема урока: методы множеств

1. Метод добавления элемента add()
2. Методы удаления элементов remove(), discard(), pop()
3. Метод удаления всех элементов clear()

**Аннотация.** Урок посвящен методам добавления и удаления элементов множеств.

## Добавление элементов

Мы научились создавать множества, элементы которых известны на этапе создания. Следующий шаг – научиться добавлять элементы в уже существующие множества.

### Метод add()

Для добавления нового элемента в множество используется метод add().

Следующий программный код:

numbers = {1, 1, 2, 3, 5, 8, 3} # создаем множество

numbers.add(21) # добавляем число 21 в множество

numbers.add(34) # добавляем число 34 в множество

print(numbers)

выводит (порядок элементов может отличаться):

{1, 2, 3, 34, 5, 8, 21}

   Не забывайте, что порядок элементов при выводе множества абсолютно произвольный.

Обратите внимание, для использования метода add() требуется предварительно созданное множество, при этом оно может быть пустым.

Следующий программный код:

numbers = set() # создаем пустое множество

numbers.add(1)

numbers.add(2)

numbers.add(3)

numbers.add(1)

print(numbers)

выводит (порядок элементов может отличаться):

{1, 2, 3}

Если требуется внести несколько значений в множество, то можно воспользоваться циклом for.

Следующий программный код:

numbers = set() # создаем пустое множество

for i in range(10):

numbers.add(i\*i + 1)

print(numbers)

выводит (порядок элементов может отличаться):

{1, 2, 65, 5, 37, 10, 17, 50, 82, 26}

## Удаление элемента

Для удаления элементов из множества используются методы:

* remove();
* discard();
* pop().

### Метод remove()

Метод remove() — удаляет элемент из множества с генерацией исключения (ошибки) в случае, если такого элемента нет.

Следующий программный код:

numbers = {1, 2, 3, 4, 5}

numbers.remove(3)

print(numbers)

выводит (порядок элементов может отличаться):

{1, 2, 4, 5}

Следующий программный код:

numbers = {1, 2, 3, 4, 5}

numbers.remove(10)

print(numbers)

приводит к возникновению ошибки KeyError, так как элемент 1010 отсутствует в множестве.

### Метод discard()

Метод discard() — удаляет элемент из множества без генерации исключения (ошибки), если элемент отсутствует.

Следующий программный код:

numbers = {1, 2, 3, 4, 5}

numbers.discard(3)

print(numbers)

выводит (порядок элементов может отличаться):

{1, 2, 4, 5}

Следующий программный код:

numbers = {1, 2, 3, 4, 5}

numbers.discard(10)

print(numbers)

не приводит к возникновению ошибки и выводит (порядок элементов может отличаться):

{1, 2, 3, 4, 5}

### Метод pop()

Метод pop() — удаляет и возвращает случайный элемент из множества с генерацией исключения (ошибки) при попытке удаления из пустого множества.

Рассмотрим программный код:

numbers = {1, 2, 3, 4, 5}

print('до удаления:', numbers)

num = numbers.pop() # удаляет случайный элемент множества, возвращая его

print('удалённый элемент:', num)

print('после удаления:', numbers)

Результат работы такого кода случаен, например, такой код может вывести:

до удаления: {1, 2, 3, 4, 5}

удалённый элемент: 1

после удаления: {2, 3, 4, 5}

   Метод pop() можно воспринимать как неконтролируемый способ удаления элементов по одному из множества.

### Метод clear()

Метод clear() удаляет все элементы из множества.

Следующий программный код:

numbers = {1, 2, 3, 4, 5}

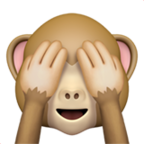
numbers.clear()

print(numbers)

выведет:

set()

В результате получили пустое множество.

Обратите внимание на то, что пустое множество выводится как set(), а не как {}. С помощью {} выводится пустой словарь.

## Примечания

**Примечание 1.** Если мы не изменяли множество, порядок обхода элементов при помощи цикла for не изменится.

**Примечание 2.** После изменения множества (методы add(), remove(), и т.д.) порядок элементов может измениться произвольным образом.

# Тема урока: методы множеств

1. Методы union(), intersection(), difference(), symmetric\_difference()
2. Методы update(), intersection\_update(), difference\_update(), symmetric\_difference\_update()
3. Операторы &, |, -, ^

**Аннотация.** Урок посвящен методам множеств, которые реализуют основные операции над множествами.

## Операции над множествами

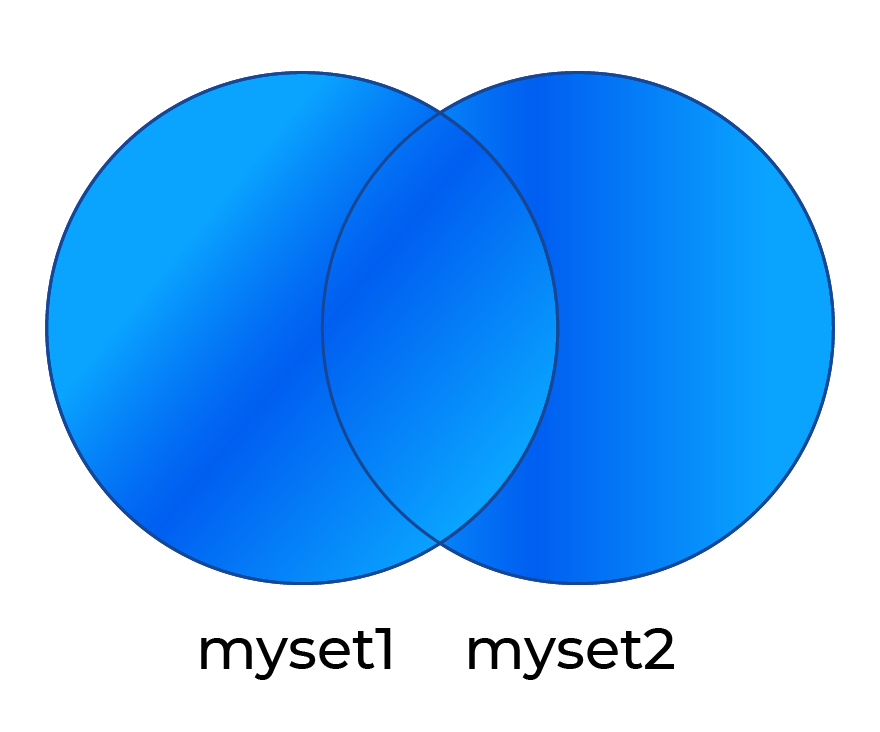
Основные операции над множествами:

* объединение множеств;
* пересечение множеств;
* разность множеств;
* симметрическая разность множеств.

Для каждой операции есть метод и оператор.

### Объединение множеств: метод union()

**Объединение** множеств – это множество, состоящее из элементов, принадлежащих хотя бы одному из объединяемых множеств. Для этой операции существует метод union().

​

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset3 = myset1.union(myset2)

print(myset3)

выводит (порядок элементов может отличаться):

{1, 2, 3, 4, 5, 6, 7, 8}

​Обратите внимание, метод union() **возвращает новое множество** в которое входят все элементы множеств myset1 и myset2. Для изменения текущего множества используется метод update().

Для объединения двух множеств можно также использовать оператор |.

Результат выполнения приведенного ниже кода:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

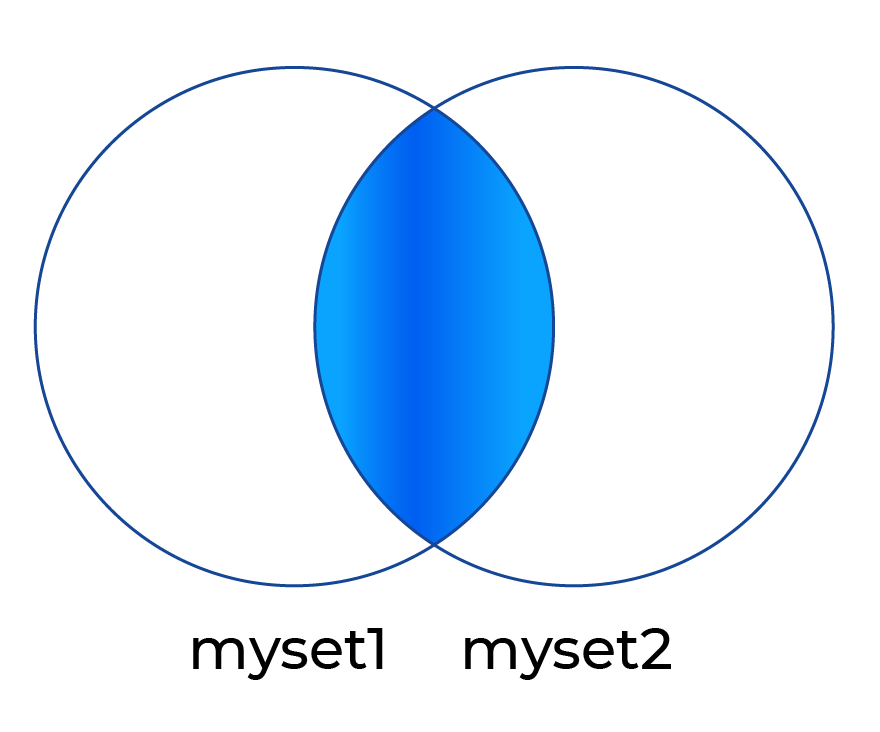
myset3 = myset1 | myset2

print(myset3)

аналогичен предыдущему.

### Пересечение множеств: метод intersection()

**Пересечение** множеств – это множество, состоящее из элементов, принадлежащих одновременно каждому из пересекающихся множеств. Для этой операции существует метод intersection().

​

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset3 = myset1.intersection(myset2)

print(myset3)

выводит (порядок элементов может отличаться):

{3, 4}

​Обратите внимание, метод intersection() возвращает новое множество в которое входят общие элементы множеств myset1 и myset2. Для изменения текущего множества используется метод intersection\_update().

Для пересечения двух множеств можно также использовать оператор &.

Результат выполнения приведенного ниже кода:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

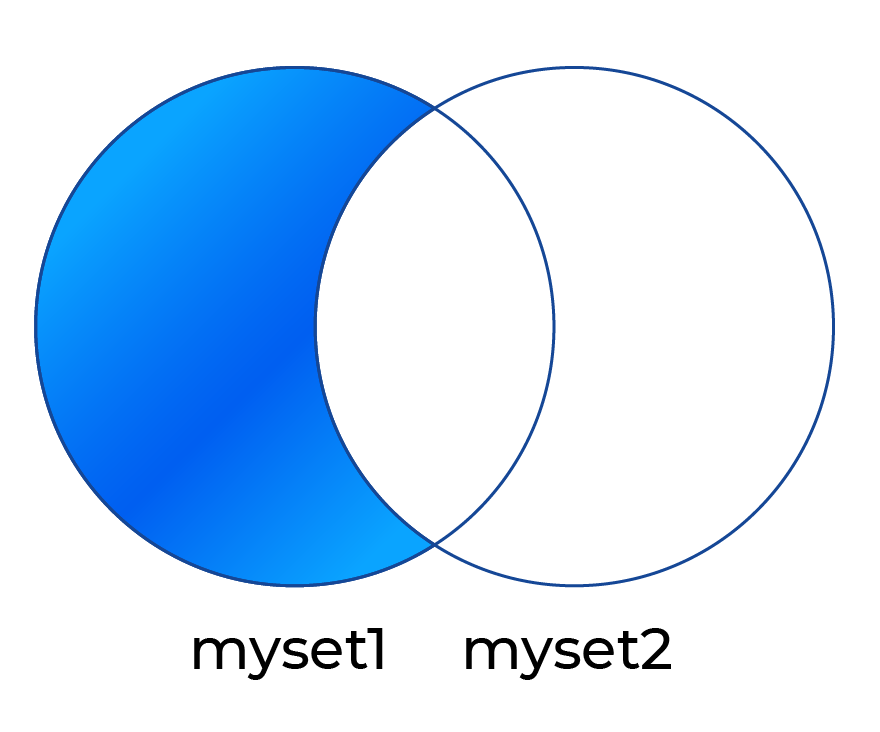
myset3 = myset1 & myset2

print(myset3)

аналогичен предыдущему.

### Разность множеств: метод difference()

**Разность** множеств – это множество, в которое входят все элементы первого множества, не входящие во второе множество. Для этой операции существует метод difference().

​

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset3 = myset1.difference(myset2)

print(myset3)

выводит (порядок элементов может отличаться):

{1, 2, 5}

Для разности двух множеств можно также использовать оператор -.

Результат выполнения приведенного ниже кода:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset3 = myset1 - myset2

print(myset3)

аналогичен предыдущему.

Обратите внимание: для операции разности множеств важен порядок, в котором указаны множества. Если поменять местами myset1 и myset2, нас ожидает совсем другой результат: элементы, входящие в множество myset2 и которых нет в множестве myset1.

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset3 = myset2.difference(myset1)

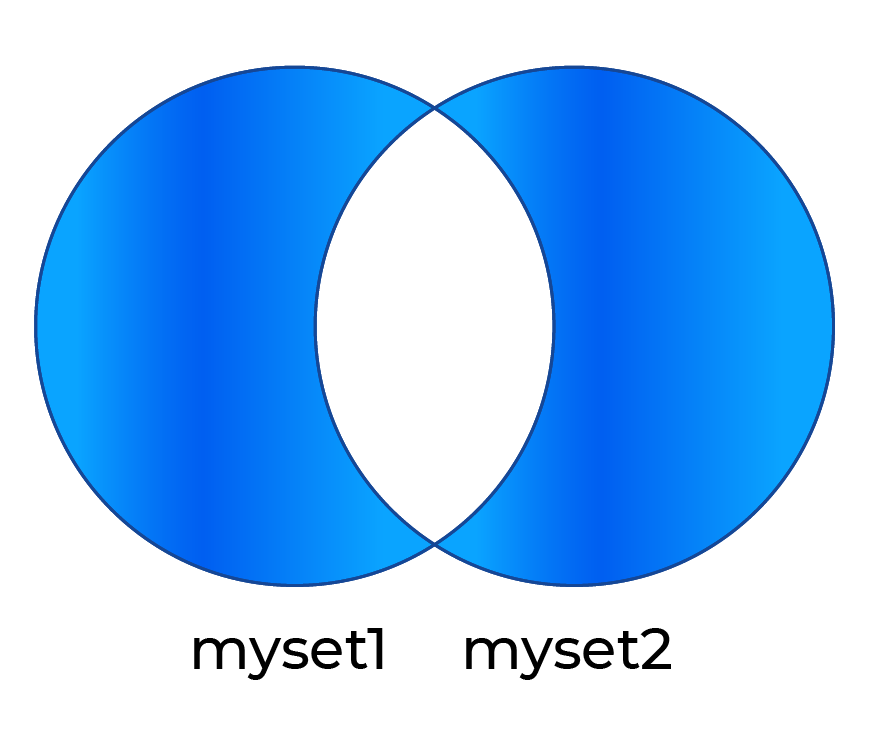
print(myset3)

выводит (порядок элементов может отличаться):

{8, 6, 7}

### Симметрическая разность: метод symmetric\_difference()

**Симметрическая разность** множеств – это множество, включающее все элементы исходных множеств, не принадлежащие одновременно обоим исходным множествам. Для этой операции существует метод symmetric\_difference().

​

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset3 = myset1.symmetric\_difference(myset2)

print(myset3)

выводит (порядок элементов может отличаться):

{1, 2, 5, 6, 7, 8}

Для симметрической разности двух множеств можно также использовать оператор ^.

Результат выполнения приведенного ниже кода:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset3 = myset1 ^ myset2

print(myset3)

аналогичен предыдущему.

​Обратите внимание: для операции симметрической разности порядок множеств не важен, на то она и симметрическая: myset1 ^ myset2 == myset2 ^ myset1.

## Методы множеств, изменяющие текущие множества

Методы union(), intersection(), difference(), symmetric\_difference() не изменяют исходные множества, а возвращают новые. Часто на практике нужно изменять исходные множества. Для таких целей используются парные методы update(), intersection\_update(), difference\_update(), symmetric\_difference\_update().

### Метод update()

Метод update() изменяет исходное множество **по объединению**.

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset1.update(myset2) # изменяем множество myset1

print(myset1)

выводит (порядок элементов может отличаться):

{1, 2, 3, 4, 5, 6, 7, 8}

Аналогичный результат получается, если использовать оператор |=:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset1 |= myset2

print(myset1)

### Метод intersection\_update()

Метод intersection\_update() изменяет исходное множество **по пересечению**.

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset1.intersection\_update(myset2) # изменяем множество myset1

print(myset1)

выводит (порядок элементов может отличаться):

{3, 4}

Аналогичный результат получается, если использовать оператор &=:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset1 &= myset2

print(myset1)

### Метод difference\_update()

Метод difference\_update() изменяет исходное множество **по разности**.

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset1.difference\_update(myset2) # изменяем множество myset1

print(myset1)

выводит (порядок элементов может отличаться):

{1, 2, 5}

Аналогичный результат получается, если использовать оператор -=:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset1 -= myset2

print(myset1)

### Метод symmetric\_difference\_update()

Метод symmetric\_difference\_update() изменяет исходное множество **по симметрической разности**.

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset1.symmetric\_difference\_update(myset2) # изменяем множество myset1

print(myset1)

выводит (порядок элементов может отличаться):

{1, 2, 5, 6, 7, 8}

Аналогичный результат получается, если использовать оператор ^=:

myset1 = {1, 2, 3, 4, 5}

myset2 = {3, 4, 6, 7, 8}

myset1 ^= myset2

print(myset1)

## Примечания

**Примечание 1.** Все основные операции над множествами выполняются двумя способами: при помощи метода или соответствующего ему оператора. Различие заключается в том, что метод может принимать в качестве аргумента не только множество (тип данных set), но и любой итерируемый объект (список, строку, кортеж..).

Приведенный ниже код:

mylist = [2021, 2020, 2019, 2018, 2017, 2016]

mytuple = (2021, 2020, 2016)

mystr = 'abcd'

myset = {2009, 2010, 2016}

print(myset.union(mystr)) # объединяем со строкой

print(myset.intersection(mylist)) # пересекаем со списком

print(myset.difference(mytuple)) # находим разность с кортежем

выводит (порядок элементов может отличаться):

{2016, 'c', 'b', 'a', 'd', 2009, 2010}

{2016}

{2009, 2010}

Приведенный ниже код:

mylist = [2021, 2020, 2019, 2018, 2017, 2016]

mytuple = (2021, 2020, 2016)

mystr = 'abcd'

myset = {2009, 2010, 2016}

print(myset | mystr)

print(myset & mylist)

print(myset - mytuple)

приводит к возникновению ошибок:

TypeError: unsupported operand type(s) for |: 'set' and 'str'

TypeError: unsupported operand type(s) for &: 'set' and 'list'

TypeError: unsupported operand type(s) for -: 'set' and 'tuple'

**Примечание 2.** Некоторые методы (union(), intersection(), difference()) и операторы (|, &, -, ^) позволяют совершать операции над несколькими множествами сразу.

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5, 6}

myset2 = {2, 3, 4, 5}

myset3 = {5, 6, 7, 8}

union1 = myset1.union(myset2, myset3)

union2 = myset1 | myset2 | myset3

difference1 = myset1.difference(myset2, myset3)

difference2 = myset1 - myset2 - myset3 # порядок выполнения слева-направо

print(union1 == union2)

print(difference1 == difference2)

выводит:

True

True

**Примечание 3.** Оператор ^ симметрической разности позволяет использовать несколько множеств, а метод symmetric\_difference() – нет.

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5, 6}

myset2 = {2, 3, 4, 7}

myset3 = {6, 20, 30}

symdifference = myset1 ^ myset2 ^ myset3 # порядок выполнения слева-направо

print(symdifference)

выводит (порядок элементов может отличаться):

{1, 5, 7, 20, 30}

 Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5, 6}

myset2 = {2, 3, 4, 7}

myset3 = {6, 20, 30}

symdifference = myset1.symmetric\_difference(myset2, myset3)

print(symdifference)

приводит к ошибке:

TypeError: symmetric\_difference() takes exactly one argument (2 given)

**Примечание 4.**Таблица соответствия методов и операторов над множествами.

|  |  |
| --- | --- |
| A | B  A.union(B) | Возвращает множество, являющееся объединением множеств A и B |
| A |= B  A.update(B) | Добавляет в множество A все элементы из множества B |
| A & B  A.intersection(B) | Возвращает множество, являющееся пересечением множеств A и B |
| A &= B  A.intersection\_update(B) | Оставляет в множестве A только те элементы, которые есть в множестве B |
| A - B  A.difference(B) | Возвращает разность множеств A и B |
| A -= B  A.difference\_update(B) | Удаляет из множества A все элементы, входящие в B |
| A ^ B  A.symmetric\_difference(B) | Возвращает симметрическую разность множеств A и B |
| A ^= B  A.symmetric\_difference\_update(B) | Записывает в A симметрическую разность множеств A и B |

**Примечание 5.** Приоритет операторов в порядке убывания (верхние операторы имеют более высокий приоритет, чем нижние) имеет вид:

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| - | разность |
| & | пересечение |
| ^ | симметрическая разность |
| | | объединение |

[Тут](https://docs.python.org/3/reference/expressions.html#operator-precedence) можно посмотреть про операторы и их приоритеты в Python.

# Тема урока: методы множеств

1. Методы issuperset(), issubset(), isdisjoint()
2. Сравнение множеств (<, >, <=, >=)

**Аннотация.** Урок посвящен методам множеств и сравнению множеств.

## Подмножества и надмножества

Напомним, что множество set1 является **подмножеством** множества set2, если все элементы первого входят во второе. При этом множество set2 – **надмножество** множества set1.

    Любое множество – подмножество самого себя, про такое подмножество говорят "**нестрогое подмножество**".

### Метод issubset()

Для определения, является ли одно из множеств подмножеством другого, используется метод issubset(). Данный метод возвращает значение True, если одно множество является подмножеством другого, и False, если не является.

Приведенный ниже код:

set1 = {2, 3}

set2 = {1, 2, 3, 4, 5, 6}

print(set1.issubset(set2))

выводит:

True

В этом примере set2 содержит все элементы set1. Это означает, что set1 – подмно­жество set2. Это также означает, что set2 – надмножество set1.

Для определения, является ли одно из множеств подмножеством другого, также применяются операторы <= (нестрогое подмножество) и < (строгое подмножество).

Приведенный ниже код:

set1 = {2, 3}

set2 = {1, 2, 3, 4, 5, 6}

print(set1 <= set2)

аналогичен предыдущему.

### Метод issuperset()

Для определения, является ли одно из множеств надмножеством другого, используется метод issuperset(). Данный метод возвращает значение True, если одно множество является надмножеством другого, в противном случае он возвращает False.

Приведенный ниже код:

set1 = {'a', 'b', 'c', 'd', 'e'}

set2 = {'c', 'e'}

print(set1.issuperset(set2))

выводит:

True

В этом примере set1 содержит все элементы set2. Это означает, что set1 – надмно­жество set2. Это также означает, что set2 – подмножество set1.

Для определения, является ли одно из множеств надмножеством другого, также применяются операторы >= (нестрогое надмножество) и > (строгое надмножество).

Приведенный ниже код:

set1 = {'a', 'b', 'c', 'd', 'e'}

set2 = {'c', 'e'}

print(set1 >= set2)

аналогичен предыдущему.

### Метод isdisjoint()

Для определения отсутствия общих элементов в множествах используется метод isdisjoint(). Данный метод возвращает значение True, если множества не имеют общих элементов, и  False, когда множества имеют общие элементы.

Приведенный ниже код:

set1 = {1, 2, 3, 4, 5}

set2 = {5, 6, 7}

set3 = {7, 8, 9}

print(set1.isdisjoint(set2))

print(set1.isdisjoint(set3))

print(set2.isdisjoint(set3))

выводит:

False

True

False

## Примечания

**Примечание 1.** Методы issuperset(), issubset(), isdisjoint() могут принимать в качестве аргумента не только множество (тип данных set), но и любой итерируемый объект (список, строку, кортеж...). Сами же эти методы могут применяться только ко множеству (тип данных set) или замороженному множеству (тип данных frozenset).

**Примечание 2.** Операторы >, <, >=, <= требуют наличия в качестве операндов множества.

**Примечание 3.** Таблица соответствия методов и операторов над множествами.

|  |  |
| --- | --- |
| set1 <= set2  set1.issubset(set2) | Возвращает True, если set1 является подмножеством set2 |
| set1 >= set2  set1.issuperset(set2) | Возвращает True, если set1 является надмножеством set2 |
| set1 < set2 | Эквивалентно set1 <= set2 and set1 != set2 (строгое подмножество) |
| set1 > set2 | Эквивалентно set1 >= set2 and set1 != set2 (строгое надмножество) |

# Тема урока: генераторы множеств, frozenset

1. Генераторы множеств
2. Неизменяемые множества frozenset

**Аннотация.** Урок посвящен генераторам множеств и типу данных frozenset.

## Генераторы множеств

Пусть требуется создать множество, содержащее цифры введенного числа.

Следующий программный код:

digits = set(int(input()))

приводит к ошибке

TypeError: 'int' object is not iterable

поскольку функция set() принимает в качестве аргумента итерируемый объект, а число (тип данных int) таковым не является.

Следующий программный код:

digits = set(input())

при вводе строки '12345' создает множество символов {'1', '2', '3', '4', '5'}, а не множество цифр {1, 2, 3, 4, 5}.

Для создания требуемого множества, содержащего уникальные цифры введенного числа, нам придется написать код:

digits = set()

for c in input():

digits.add(int(c))

Такой код хоть и не сложен, однако достаточно громоздок.

Для создания множеств в Python можно использовать специальный синтаксис, как при создании [списка](https://stepik.org/lesson/326725/step/1?unit=310010).

Приведенный выше код можно переписать с использованием генератора множеств:

digits = {int(c) for c in input()}

Общий вид генератора множеств следующий:

{выражение for переменная in последовательность},

где  выражение — некоторое выражение, как правило, зависящее от использованной в списочном выражении переменной, которым будут заполнены элементы множества переменная — имя некоторой переменной, последовательность — последовательность значений, которые она принимает (любой итерируемый объект).

### **Примеры использования генератора множеств**

**1. Создать множество, заполненное квадратами целых чисел от 0 до 9, можно так:**

squares = {i \*\* 2 for i in range(10)}

**2. Создать множество, заполненное кубами целых чисел от 10 до 20, можно так:**

cubes = {i \*\* 3 for i in range(10, 21)}

**3. Создать множество, заполненное символами строки, можно так:**

chars = {c for c in 'abcdefg'}

### Условия в генераторе множеств

В генераторах множеств можно использовать условный оператор. Например, если требуется создать множество, заполненное только цифрами некоторой строки, то мы можем написать такой код:

digits = {int(d) for d in 'abcd12ef78ghj90' if d.isdigit()}

## Frozenset

Замороженное множество (frozenset) также является встроенной коллекцией в Python. Обладая характеристиками обычного множества, замороженное множество не может быть изменено после создания.

Кортеж (тип tuple) – неизменяемая версия списка (тип list), а замороженное множество (тип frozenset) – неизменяемая версия обычного множества (тип set).

Для создания замороженного множества используется встроенная функция frozenset(), которая принимает в качестве аргумента другую коллекцию.

Приведенный ниже код:

myset1 = frozenset({1, 2, 3}) # на основе множества

myset2 = frozenset([1, 1, 2, 3, 4, 4, 4, 5, 6, 6]) # на основе списка

myset3 = frozenset('aabcccddee') # на основе строки

print(myset1)

print(myset2)

print(myset3)

выводит:

frozenset({1, 2, 3})

frozenset({1, 2, 3, 4, 5, 6})

frozenset({'e', 'd', 'c', 'b', 'a'})

### Операции над замороженными множествами

Над замороженными множествами можно производить все операции, которые можно производить над обычными множествами:

* объединение множеств: метод union() или оператор |;
* пересечение множеств: метод intersection() или оператор &;
* разность множеств: метод difference() или оператор -;
* симметрическая разность множеств: метод symmetric\_difference() или оператор ^.

Приведенный ниже код:

myset1 = frozenset('hello')

myset2 = frozenset('world')

print(myset1 | myset2)

print(myset1 & myset2)

print(myset1 ^ myset2)

выводит:

frozenset({'l', 'w', 'e', 'h', 'r', 'd', 'o'})

frozenset({'l', 'o'})

frozenset({'d', 'h', 'w', 'e', 'r'})

   Результатом операций над замороженными множествами будут тоже замороженные множества.

## Примечания

**Примечание 1.** Будучи изменяемыми, обычные множества не могут быть элементами других множеств. Замороженные множества являются неизменяемыми, а значит могут быть элементами других множеств.

Приведенный ниже код:

sentence = 'The cat in the hat had two sidekicks, thing one and thing two.'

words = sentence.lower().replace('.', '').replace(',', '').split()

vowels = ['a', 'e', 'i', 'o', 'u']

consonants = {frozenset({letter for letter in word if letter not in vowels}) for word in words}

print(\*consonants, sep='\n')

выводит (порядок элементов может отличаться):

frozenset({'d', 'h'})

frozenset({'h', 't'})

frozenset({'n', 'h', 'g', 't'})

frozenset({'n'})

frozenset({'c', 't'})

frozenset({'n', 'd'})

frozenset({'w', 't'})

frozenset({'s', 'c', 'k', 'd'})

**Примечание 2.** Методы, изменяющие множество, отсутствуют у замороженных множеств:

* add()
* remove()
* discard()
* pop()
* clear()
* update()
* intersection\_update()
* difference\_update()
* symmetric\_difference\_update()

**Примечание 3.** Мы можем сравнивать простые (тип set) и замороженные множества (тип frozenset).

Приведенный ниже код:

myset1 = set('qwerty')

myset2 = frozenset('qwerty')

print(myset1 == myset2)

выведет:

True

# Тема урока: словари в Python

1. Новый тип коллекции
2. Отличия словарей от списков
3. Создание словарей
4. Обращение по ключу
5. Встроенная функция dict()
6. Создание словарей на основе списков и кортежей
7. Пустой словарь
8. Вывод словаря
9. Особенности словарей

**Аннотация.** В этом уроке мы начнем изучение словарей в Python, тип данных – dict. Этот тип данных похож на списки и применяется при решении многих задач.

## Словари

В прошлых уроках мы изучили четыре типа коллекций в Python:

* списки – изменяемые коллекции элементов, индексируемые;
* строки – неизменяемые коллекции символов, индексируемые;
* кортежи – неизменяемые коллекции элементов, индексируемые;
* множества – изменяемые коллекции уникальных элементов, неиндексируемые.

Следующий тип – **словари** – изменяемые коллекции элементов **с произвольными индексами – ключами**. Если в списках элементы индексируются целыми числами, начиная с 0, то в словарях — любыми ключами, в том числе в виде строк.

Как нам уже известно, списки — удобный и самый популярный способ хранения большого количества данных в одной переменной. Списки индексируют все хранящиеся в них элементы, что позволяет быстро обращаться к элементу, зная его индекс.

Приведенный ниже код:

languages = ['Python', 'C#', 'Java', 'C++']

print(languages[0])

print(languages[2])

выводит:

Python

Java

Допустим, мы хотим хранить имя создателя каждого языка программирования. Это можно сделать несколькими способами.

**Способ 1.** Хранить еще один список, где по соответствующему индексу будет находиться имя создателя языка программирования.

Приведенный ниже код:

languages = ['Python', 'C#', 'Java', 'C++']

creators = ['Гвидо ван Россум', 'Андерс Хейлсберг', 'Джеймс Гослинг', 'Бьёрн Страуструп']

print('Создателем языка', languages[0], 'является', creators[0])

выводит:

Создателем языка Python является Гвидо ван Россум

Подход рабочий, но хранить данные в двух коллекциях не очень удобно.

**Способ 2.** Хранить список кортежей с парами значений "язык - имя создателя" в каждом.

Приведенный ниже код:

languages = [('Python', 'Гвидо ван Россум'),

('C#', 'Андерс Хейлсберг'),

('Java', 'Джеймс Гослинг'),

('C++', 'Бьёрн Страуструп')]

print('Создателем языка', languages[2][0], 'является', languages[2][1])

выводит:

Создателем языка Java является Джеймс Гослинг

Тоже рабочий подход, однако не очень эффективный. Придется написать цикл for для поиска по всем элементам списка languages кортежа, первый элемент которого равен искомому (названию языка). Чтобы найти автора языка C++ , нужно будет в цикле пройти мимо Python, C# и Java. Не получится угадать заранее, что язык C++ лежит после них.

Приведенный ниже код:

languages = [('Python', 'Гвидо ван Россум'),

('C#', 'Андерс Хейлсберг'),

('Java', 'Джеймс Гослинг'),

('C++', 'Бьёрн Страуструп')]

for item in languages:

if item[0] == 'C++':

print('Создателем языка', item[0], 'является', item[1])

выводит:

Создателем языка C++ является Бьёрн Страуструп

Списки индексируются целыми числами, но в этом случае удобно было бы находить информацию не по числу, а по строке — названию языка программирования. В списках строки не могут быть индексами, однако в словарях это возможно.

Словарь (тип данных dict), как и список, позволяет хранить много данных. В отличие от списка, в словаре для каждого элемента можно произвольно определить «индекс» — **ключ**, по которому он будет доступен.

Словарь — реализация структуры данных "ассоциативный массив" или "хеш таблица". В других языках аналогичная структура называется map, HashMap, Dictionary.

### Создание словаря

Чтобы создать словарь, нужно перечислить его элементы – пары ключ-значение – через запятую в фигурных скобках, как и элементы множества. Первым указывается ключ, после двоеточия — значение, доступное в словаре по этому ключу.

Приведенный ниже код:

languages = {'Python': 'Гвидо ван Россум',

'C#': 'Андерс Хейлсберг',

'Java': 'Джеймс Гослинг',

'C++': 'Бьёрн Страуструп'}

создает словарь, в котором ключом служит строка — название языка программирования, а значением — имя создателя языка.

### Обращение к элементу словаря

Извлечь значение элемента словаря можно, обратившись к нему по его ключу. Чтобы получить значение по заданному ключу, как и в списках, используем квадратные скобки [] , индексируем по ключу.

**Способ 3.**Приведенный ниже код:

languages = {'Python': 'Гвидо ван Россум',

'C#': 'Андерс Хейлсберг',

'Java': 'Джеймс Гослинг',

'C++': 'Бьёрн Страуструп'}

print('Создателем языка C# является', languages['C#'])

выводит:

Создателем языка C# является Андерс Хейлсберг

В отличие от списков, номеров позиций в словарях нет.

Приведенный ниже код:

languages = {'Python': 'Гвидо ван Россум',

'C#': 'Андерс Хейлсберг',

'Java': 'Джеймс Гослинг',

'C++': 'Бьёрн Страуструп'}

print('Создателем языка C# является', languages[1])

приводит к возникновению ошибки KeyError.

Ошибка KeyError возникнет и при попытке извлечь значение по несуществующему ключу. В качестве ключа можно указать выражение: Python вычислит его значение и обратится к искомому элементу.

**Способ 4.**Приведенный ниже код:

languages = {'Python': 'Гвидо ван Россум',

'C#': 'Андерс Хейлсберг',

'Java': 'Джеймс Гослинг',

'C++': 'Бьёрн Страуструп'}

print('Создателем языка C# является', languages['C' + '#'])

выводит:

Создателем языка C# является Андерс Хейлсберг

### Создание словаря с помощью функции dict()

Если ключи словаря — строки без каких-либо специальных символов, то для создания словаря можно использовать функцию dict().

Приведенный ниже код:

info = dict(name = 'Timur', age = 28, job = 'Teacher')

создает словарь с тремя элементами, ключами которого служат строки 'name', 'age', 'job', а значениями –  'Timur', 28, 'Teacher'.

### Создание словаря на основании списков и кортежей

Создавать словари можно на основе списков кортежей или кортежей списков. Первый элемент списка или кортежа станет ключом, второй — значением.

Приведенный ниже код:

info\_list = [('name', 'Timur'), ('age', 28), ('job', 'Teacher')] # список кортежей

info\_dict = dict(info\_list) # создаем словарь на основе списка кортежей

создает словарь с тремя элементами, где ключи — строки name, age, job, а соответствующие им значения — 'Timur', 28, 'Teacher'.

Аналогично работает приведенный ниже код:

info\_tuple = (['name', 'Timur'], ['age', 28], ['job', 'Teacher']) # кортеж списков

info\_dict = dict(info\_tuple) # создаем словарь на основе кортежа списков

Если необходимо создать словарь, каждому ключу которого соответствует одно и то же значение, можно воспользоваться методом fromkeys().

Приведенный ниже код:

dict1 = dict.fromkeys(['name', 'age', 'job'], 'Missed information')

создает словарь с тремя элементами, где ключи — строки 'name', 'age', 'job', а соответствующие им значения: 'Missed information', 'Missed information', 'Missed information'.

   Если методу fromkeys() не передать второй параметр, то по умолчанию присваивается значение None.

Приведенный ниже код:

dict1 = dict.fromkeys(['name', 'age', 'job'])

создает словарь с тремя элементами, в которых ключи — строки 'name', 'age', 'job', а значения — None, None, None.

### Пустой словарь

**Пустой словарь** можно создать двумя способами:

* с помощью пустых фигурных скобок;
* с помощью функции dict().

Приведенный ниже код:

dict1 = {}

dict2 = dict()

print(dict1)

print(dict2)

print(type(dict1))

print(type(dict2))

создает два пустых словаря и выводит:

{}

{}

<class 'dict'>

<class 'dict'>

Вспомните, что создать пустое множество можно, только используя функцию set() , потому что пустые фигурные скобки зарезервированы для создания словаря.

### Вывод словаря

Для вывода всего словаря можно использовать функцию print():

languages = {'Python': 'Гвидо ван Россум',

'C#': 'Андерс Хейлсберг',

'Java': 'Джеймс Гослинг'}

info = dict(name = 'Timur', age = 28, job = 'Teacher')

print(languages)

print(info)

Функция print() выводит на экран элементы словаря в фигурных скобках, разделенные запятыми:

{'Python': 'Гвидо ван Россум', 'C#': 'Андерс Хейлсберг', 'Java': 'Джеймс Гослинг'}

{'name': 'Timur', 'age': 28, 'job': 'Teacher'}

Начиная с версии Python 3.63.6, словари являются упорядоченными, то есть сохраняют порядок следования ключей в порядке их внесения в словарь.

## Примечания

**Примечание 1.**Словари принципиально отличаются от списков по структуре хранения в памяти. Список — последовательная область памяти, то есть все его элементы (указатели на элементы) действительно хранятся в указанном порядке, расположены последовательно. Благодаря этому и можно быстро «прыгнуть» к элементу по его индексу. В словаре же используется специальная структура данных — **хеш-таблица**. Она позволяет вычислять числовой хеш от ключа и использовать обычные списки, где в качестве индекса элемента берется этот хеш.

**Примечание 2.** В рамках одного словаря каждый ключ уникален.

**Примечание 3.** Словари удобно использовать для хранения различных сущностей. Например, если нужно работать с информацией о человеке, то можно хранить все необходимые сведения, включающие такие разные сущности как "возраст", "профессия", "название города", "адрес электронной почты" в одном словаре  info и легко обращаться к его элементам по ключам:

info = {'name': 'Timur',

'age': 28,

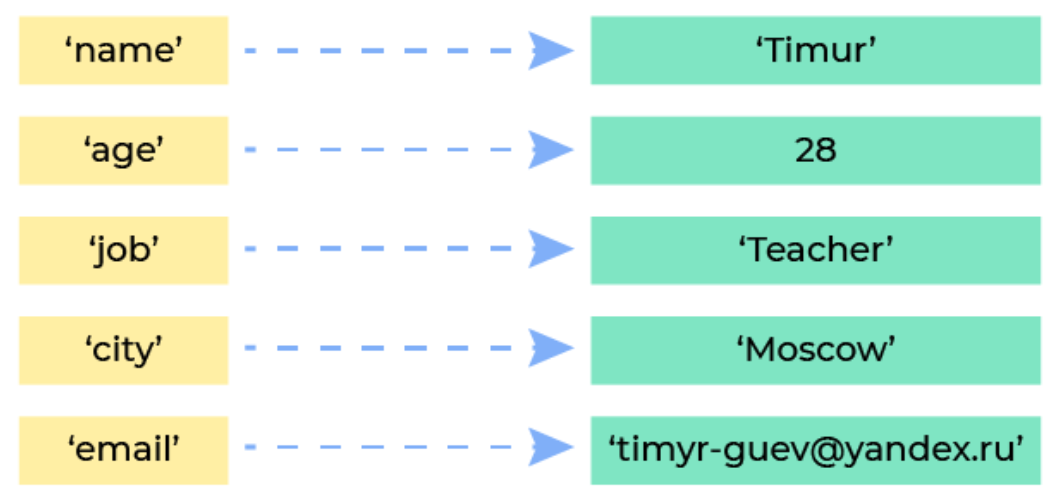
'job': 'Teacher',

'city': 'Moscow',

'email': 'timyr-guev@yandex.ru'}

print(info['name'])

print(info['email'])



**Примечание 4.** Создать словарь на основании двух списков (кортежей) можно с помощью встроенной функции-упаковщика zip(), о которой расскажем позже.

Приведенный ниже код:

keys = ['name', 'age', 'job']

values = ['Timur', 28, 'Teacher']

info = dict(zip(keys, values))

print(info)

выводит (порядок элементов может отличаться):

{'name': 'Timur', 'age': 28, 'job': 'Teacher'}

   В случае несовпадения длины списков функция самостоятельно отсечет лишние элементы.

## Особенности словарей

Словари (тип данных dict) довольно просты, но о нескольких моментах следует помнить при их использовании.

### Ключи должны быть уникальными

Словарь не может иметь два и более значений по одному и тому же ключу. Если при создании словаря (в литеральной форме) указать дважды один и тот же ключ, будет использовано последнее из указанных значений.

Приведенный ниже код:

info = {'name': 'Ruslan',

'age': 28,

'name': 'Timur'}

print(info['name'])

выводит:

Timur

### Ключи должны быть неизменяемым типом данных

Ключом словаря могут быть данные любого неизменяемого типа:

* число;
* строка;
* булево значение;
* кортеж;
* замороженное множество (frozenset);
* ...

Приведенный ниже код создает словарь, ключами которого являются неизменяемые типы данных:

my\_dict = {198: 'beegeek', 'name': 'Bob', True: 'a', (2, 2): 25}

Ключ словаря не может относиться к изменяемому типу данных:

* список;
* множество;
* словарь;
* ...

Приведенный ниже код приводит к возникновению ошибки:

my\_dict = {[2, 2]: 25, {1, 2}: 'python', 'name': 'Bob'}

### Значения могут относиться к любому типу данных, их тип данных произволен

Нет никаких ограничений для значений, хранящихся в словарях. Значения в словарях могут принадлежать к произвольному типу данных и повторяться для разных ключей многократно.

my\_dict1 = {'a': [1, 2, 3], 'b': {1, 2, 3}} # значения – изменяемый тип данных

my\_dict2 = {'a': [1, 2], 'b': [1, 2], 'c': [1, 2]} # значения повторяются

# Тема урока: словари в Python

1. Встроенные функции len(),sum(),min(),max()
2. Оператор принадлежности in
3. Перебор словарей
4. Распаковка словаря
5. Форматированный вывод словарей
6. Сравнение словарей
7. Методы keys(), values() и items()

**Аннотация.** В этом уроке мы изучим основной функционал при работе со словарями.

## Основы работы со словарями

Работа со словарями похожа на работу со списками, поскольку и словари, и списки содержат в качестве отдельных элементов пары: в словарях ключ: значение, в списках индекс: значение.

### Функция len()

**Длиной словаря** называется количество его элементов. Для определения длины словаря используют встроенную функцию len() (от слова length – длина).

Следующий программный код:

fruits = {'Apple': 70, 'Grape': 100, 'Banana': 80}

capitals = {'Россия': 'Москва', 'Франция': 'Париж'}

print(len(fruits))

print(len(capitals))

выведет:

3

2

### Оператор принадлежности in

Оператор in позволяет проверить, содержит ли словарь заданный **ключ**.

Рассмотрим код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

if 'Франция' in capitals:

print('Столица Франции - это', capitals['Франция'])

Такой код проверяет, содержит ли словарь capitals элемент с ключом Франция и выводит соответствующий текст:

Столица Франции - это Париж

   Можно использовать оператор in вместе с логическим оператором not.

Не забывайте, что при обращении по **несуществующему ключу**, возникнет ошибка во время выполнения программы.

Приведенный ниже код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

print(capitals['Англия'])

приводит к возникновению ошибки:

KeyError: 'Англия'

Оператор принадлежности in на словарях работает **очень быстро**, намного быстрее, чем на списках, поэтому если нужен многократный поиск в коллекции данных, словарь – подходящий выбор.

### Встроенные функции sum(), min(), max()

Встроенная функция sum() принимает в качестве аргумента **словарь с числовыми ключами** и вычисляет сумму его ключей.

Следующий программный код:

my\_dict = {10: 'Россия', 20: 'США', 30: 'Франция'}

print('Сумма всех ключей словаря =', sum(my\_dict))

выводит:

Сумма всех ключей словаря = 60

   Для корректной работы функции sum() ключами словаря должны быть именно числа.

Встроенные функции min() и max() принимают в качестве аргумента словарь и находят минимальный и максимальный ключ соответственно, при этом ключ может принадлежать к любому типу данных, для которого возможны операции порядка <, <=, >, >= (числа, строки, и т.д.).

Следующий программный код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

months = {1: 'Январь', 2: 'Февраль', 3: 'Март'}

print('Минимальный ключ =', min(capitals))

print('Максимальный ключ =', max(months))

выводит:

Минимальный ключ = Россия

Максимальный ключ = 3

### Сравнение словарей

Словари можно сравнивать между собой. Равные словари имеют одинаковое количество элементов и содержат равные элементы (ключ: значение). Для сравнения словарей используются операторы == и !=.

Приведенный ниже код:

months1 = {1: 'Январь', 2: 'Февраль'}

months2 = {1: 'Январь', 2: 'Февраль', 3: 'Март'}

months3 = {3: 'Март', 1: 'Январь', 2: 'Февраль'}

print(months1 == months2)

print(months2 == months3)

print(months1 != months3)

выводит:

False

True

True

## Примечания

**Примечание 1.** Обращение по индексу и срезы **недоступны** для словарей.

**Примечание 2.** Операция конкатенации + и умножения на число \* **недоступны** для словарей.

**Примечание 3.** Словари нужно использовать в следующих случаях:

* Подсчет числа каких-то объектов. В этом случае нужно завести словарь, в котором ключи — названия объектов, а значения — их количество.
* Хранение каких-либо данных, связанных с объектом. Ключи — наименования объектов, значения — связанные с ними данные. Например, если нужно по названию месяца определить его порядковый номер, то это можно сделать при помощи словаря num = {'January': 1, 'February': 2, 'March': 3, ...}.
* Установка соответствия между объектами (например, “родитель—потомок”). Ключ — объект, значение — соответствующий ему объект.
* Если нужен обычный список, где максимальное значение индекса элемента очень велико, но при этом используются не все возможные индексы (так называемый “разреженный список”), то для экономии памяти можно использовать словарь.

**Примечание 4.** О том, как устроен словарь (тип dict) в Python можно почитать в [статье](https://habr.com/ru/post/432996/).

**Примечание 5.** Исходный код словаря (тип dict) в Python можно найти [тут](https://github.com/python/cpython/blob/main/Objects/dictobject.c).

## Перебор элементов словаря

Перебор элементов словаря осуществляется так же как перебор элементов списка, с помощью цикла for.

Для вывода ключей словаря **каждого на отдельной строке** можно использовать следующий код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

for key in capitals:

print(key)

 Такой код выведет (порядок элементов может отличаться):

Россия

Франция

Чехия

Для вывода значений словаря **каждого на отдельной строке** можно использовать следующий код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

for key in capitals:

print(capitals[key])

 Такой код выведет (порядок элементов может отличаться):

Москва

Париж

Прага

Для вывода элементов словаря **каждого на отдельной строке** можно использовать следующий код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

for key in capitals:

print('Столица', key, '- это', capitals[key])

Такой код выведет (порядок элементов может отличаться):

Столица Россия - это Москва

Столица Франция - это Париж

Столица Чехия - это Прага

### Методы keys(), values(), items()

Словарный метод keys() возвращает **список ключей** всех элементов словаря.

Следующий программный код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

for key in capitals.keys(): # итерируем по списку ['Россия', 'Франция', 'Чехия']

print(key)

выводит (порядок элементов может отличаться):

Россия

Франция

Чехия

Словарный метод values() возвращает **список значений** всех элементов словаря.

Следующий программный код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

for value in capitals.values(): # итерируем по списку ['Москва', 'Париж', 'Прага']

print(value)

выводит (порядок элементов может отличаться):

Москва

Париж

Прага

Словарный метод items() возвращает список всех элементов словаря, состоящий из кортежей пар (ключ, значение).

Следующий программный код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

for item in capitals.items():

print(item)

выводит (порядок элементов может отличаться):

('Россия', 'Москва')

('Франция', 'Париж')

('Чехия', 'Прага')

Используя магию распаковки кортежей, можно писать такой код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

for key, value in capitals.items():

print(key, '-', value)

Такой код выведет (порядок элементов может отличаться):

Россия - Москва

Франция - Париж

Чехия - Прага

### Распаковка ключей словаря

Если требуется вывести только значение ключей словаря, то мы также можем использовать операцию **распаковки словаря**.

Приведенный ниже код:

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

print(\*capitals, sep='\n')

выводит (порядок элементов может отличаться):

Россия

Франция

Чехия

Начиная с версии Python 3.6 словари являются упорядоченными, то есть сохраняют порядок следования ключей в порядке их внесения в словарь.

### Сортировка словаря

Так как словарь состоит из пар, то и отсортировать его можно как по ключам, так и по значениям.

Сортировка по ключам выполняется с использованием функции sorted().

Приведенный ниже код:

capitals = {'Россия': 'Москва', 'Англия': 'Лондон', 'Чехия': 'Прага', 'Бразилия':'Бразилиа'}

for key in sorted(capitals):

print(key)

будет **гарантированно** выводить ключи словаря в алфавитном порядке, по возрастанию:

Англия

Бразилия

Россия

Чехия

Обратите внимание: функция sorted() возвращает отсортированный список ключей, а не словарь. Не путайте встроенную функцию sorted() и списочный метод sort(). Словари **не содержат** метода sort().

Для сортировки словаря по значениям можно использовать функцию sorted() вместе с методом items().

Приведенный ниже код:

capitals = {'Россия': 'Москва', 'Англия': 'Лондон', 'Чехия': 'Прага', 'Бразилия':'Бразилиа'}

for key, value in sorted(capitals.items(), key = lambda x: x[1]):

print(value)

будет **гарантированно** выводить значения словаря в алфавитном порядке, по возрастанию:

Бразилиа

Лондон

Москва

Прага

Стоит учитывать, что sorted(capitals.items(), key= lambda x: x[1]) возвращает не словарь, а отсортированный по значению список кортежей.

При сортировке словаря по значениям, мы используем анонимную функцию lambda x: x[1], о ней будет рассказано в следующих уроках.

## Примечания

**Примечание 1.** Как мы уже знаем, с помощью оператора принадлежности in можно быстро проверить наличие ключа в словаре. Для проверки наличия значения в словаре можно использовать оператор in вместе с методом values(), который возвращает список всех значений словаря.

**Проверка наличия ключа в словаре:**

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

if 'Россия' in capitals:

print('В словаре есть ключ Россия')

**Проверка наличия значения в словаре:**

capitals = {'Россия': 'Москва', 'Франция': 'Париж', 'Чехия': 'Прага'}

if 'Париж' in capitals.values():

print('В словаре есть значение Париж')

**Примечание 2.** Встроенная функция sorted() имеет опциональный параметр reverse. Если установить этот параметр в значение True, то сортировка будет по убыванию.

**Примечание 3.** Код для работы со словарями нужно писать таким образом, чтобы он правильно работал при любом порядке обхода с помощью цикла for.

**Примечание 4.** Словарные методы items(), keys(), values() возвращают не совсем обычные списки. Типы этих списков -  dict\_items, dict\_keys, dict\_values соответственно, в отличие от обычных списков - list. Методы обычных списков недоступны для списков типа dict\_items, dict\_keys, dict\_values. Используйте явное преобразование с помощью функции list() для получения доступа к методам списков.

# Тема урока: методы словарей

1. Добавление и изменение элементов в словаре
2. Удаление элементов из словаря
3. Методы get(), update()
4. Методы pop(), popitem()
5. Методы clear(), copy()
6. Метод setdefault()

**Аннотация.** В этом уроке мы изучим основные методы словарей.

## Методы словарей

Словари как и списки имеют много полезных методов для упрощения работы с ними и решения повседневных задач. В прошлом уроке, мы уже познакомились с тремя словарными методами:

* метод items() – возвращает словарные пары ключ: значение, как соответствующие им кортежи;
* метод keys() – возвращает список ключей словаря;
* метод values() – возвращает список значений словаря.

### Добавление и изменение элементов в словаре

Чтобы изменить значение по определенному ключу в словаре, достаточно использовать индексацию вместе с оператором присваивания. При этом если ключ уже присутствует в словаре, его значение заменяется новым, если же ключ отсутствует, то в словарь будет добавлен новый элемент.

Приведенный ниже код:

info = {'name': 'Sam',

'age': 28,

'job': 'Teacher'}

info['name'] = 'Timur' # изменяем значение по ключу name

info['email'] = 'timyr-guev@yandex.ru' # добавляем в словарь элемент с ключом email

print(info)

выводит (порядок элементов может отличаться):

{'name': 'Timur', 'age': 28, 'job': 'Teacher', 'email': 'timyr-guev@yandex.ru'}

**Обратите внимание на отличие в поведении словарей и списков:**

* Если в списке lst нет элемента с индексом 77, то попытка обращения к нему, например, с помощью строки кода print(lst[7]) приведет к возникновению ошибки. И попытка присвоить ему значение lst[7] = 100 тоже приведет к возникновению ошибки.
* Если в словаре dct нет элемента с ключом name, то попытка обращения к нему, например, с помощью строки кода print(dct['name']) приведет к возникновению ошибки. Однако попытка присвоить значение по отсутствующему ключу dct['name'] = 'Timur' ошибки не вызовет.

**Решим следующую задачу:** пусть задан список чисел numbers , где некоторые числа встречаются неоднократно. Нужно узнать, сколько именно раз встречается каждое из чисел.

numbers = [9, 8, 32, 1, 10, 1, 10, 23, 1, 4, 10, 4, 2, 2, 2, 2, 1, 10, 1, 2, 2, 32, 23, 23]

Первый код, который приходит в голову имеет вид:

numbers = [9, 8, 32, 1, 10, 1, 10, 23, 1, 4, 10, 4, 2, 2, 2, 2, 1, 10, 1, 2, 2, 32, 23, 23]

result = {}

for num in numbers:

result[num] += 1

Однако, просто так сделать result[num] += 1 нельзя, так как если ключа num в словаре еще нет, то возникнет ошибка KeyError.

Следующий программный код, полностью решает поставленную задачу:

numbers = [9, 8, 32, 1, 10, 1, 10, 23, 1, 4, 10, 4, 2, 2, 2, 2, 1, 10, 1, 2, 2, 32, 23, 23]

result = {}

for num in numbers:

if num not in result:

result[num] = 1

else:

result[num] += 1

Цикл for перебирает все элементы списка numbers и для каждого проверяет, присутствует ли он уже в качестве ключа в словаре result. Если значение отсутствует, значит число встретилось впервые и мы инициируем значение result[num] = 1. Если значение уже присутствует в словаре, увеличим result[num] на единицу.

Этот код можно улучшить с помощью метода get().

### Метод get()

Мы можем получить значение в словаре по ключу с помощью индексации, то есть квадратных скобок. Однако как мы знаем, в случае отсутствия ключа будет происходить ошибка KeyError.

Приведенный ниже код:

info = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

print(info['name'])

выводит:

Bob

Приведенный ниже код:

info = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

print(info['salary'])

приводит к возникновению ошибки:

KeyError: 'salary'

Для того, чтобы избежать возникновения ошибки в случае отсутствия ключа в словаре можно использовать метод get(), способный кроме ключа принимать и второй аргумент — значение, которое вернется, если заданного ключа нет. Когда второй аргумент не указан, то метод в случае отсутствия ключа возвращает None.

Приведенный ниже код:

info = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

item1 = info.get('salary')

item2 = info.get('salary', 'Информации о зарплате нет')

print(item1)

print(item2)

выводит:

None

Информации о зарплате нет

   С помощью словарного метода get(), можно упростить код в задаче о повторяющихся числах.

numbers = [9, 8, 32, 1, 10, 1, 10, 23, 1, 4, 10, 4, 2, 2, 2, 2, 1, 10, 1, 2, 2, 32, 23, 23]

result = {}

for num in numbers:

result[num] = result.get(num, 0) + 1

Цикл for перебирает все элементы списка numbers и для каждого элемента с помощью метода get() мы получаем либо его значение из словаря result, либо значение по умолчанию, равное 00. К данному значению прибавляется единица, и результат записывается обратно в словарь по нужному ключу.

### Метод update()

Метод update() – реализует своеобразную операцию конкатенации для словарей. Он объединяет ключи и значения одного словаря с ключами и значениями другого. При совпадении ключей в итоге сохранится значение словаря, указанного в качестве аргумента метода update().

Приведенный ниже код:

info1 = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

info2 = {'age': 30,

'city': 'New York',

'email': 'bob@web.com'}

info1.update(info2)

print(info1)

выводит (порядок элементов может отличаться):

{'name': 'Bob', 'age': 30, 'job': 'Dev', 'city': 'New York', 'email': 'bob@web.com'}

В Python 3.9 появились операторы | и |= которые реализуют операцию конкатенации словарей.

Приведенный ниже код:

info1 = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

info2 = {'age': 30,

'city': 'New York',

'email': 'bob@web.com'}

info1 |= info2

print(info1)

аналогичен предыдущему коду.

### Метод setdefault()

Метод setdefault() позволяет получить значение из словаря по заданному ключу, автоматически добавляя элемент словаря, если он отсутствует.

Метод принимает два аргумента:

* key: ключ, значение по которому следует получить, если таковое имеется в словаре, либо создать.
* default: значение, которое будет использовано при добавлении нового элемента в словарь.

В зависимости от значений параметров key и default возможны следующие сценарии работы данного метода.

**Сценарий 1.** Если ключ key присутствует в словаре, то метод возвращает значение по заданному ключу (независимо от того, передан параметр default или нет).

Приведенный ниже код:

info = {'name': 'Bob',

'age': 25}

name1 = info.setdefault('name') # параметр default не задан

name2 = info.setdefault('name', 'Max') # параметр default задан

print(name1)

print(name2)

выводит:

Bob

Bob

**Сценарий 2.** Если ключ key отсутствует в словаре, то метод вставляет переданное значение default по заданному ключу.

Приведенный ниже код:

info = {'name': 'Bob',

'age': 25}

job = info.setdefault('job', 'Dev')

print(info)

print(job)

выводит:

{'name': 'Bob', 'age': 25, 'job': 'Dev'}

Dev

При этом если значение default не передано в метод, то вставится значение None.

Приведенный ниже код:

info = {'name': 'Bob',

'age': 25}

job = info.setdefault('job')

print(info)

print(job)

выводит:

{'name': 'Bob', 'age': 25, 'job': None}

None

### Удаление элементов из словаря

Существует несколько способов удаления элементов из словаря:

* оператор del;
* метод pop();
* метод popitem();
* метод clear().

### Оператор del

С помощью оператора del можно удалять элементы словаря по определенному ключу.

Следующий программный код:

info = {'name': 'Sam',

'age': 28,

'job': 'Teacher',

'email': 'timyr-guev@yandex.ru'}

del info['email'] # удаляем элемент имеющий ключ email

del info['job'] # удаляем элемент имеющий ключ job

print(info)

выводит (порядок элементов может отличаться):

{'name': 'Sam', 'age': 28}

​   Если удаляемого ключа в словаре нет, возникнет ошибка KeyError.

### Метод pop()

Оператор del удаляет из словаря элемент по заданному ключу вместе с его значением. Если требуется получить само значение удаляемого элемента, то нужен метод pop().

Следующий программный код:

info = {'name': 'Sam',

'age': 28,

'job': 'Teacher',

'email': 'timyr-guev@yandex.ru'}

email = info.pop('email') # удаляем элемент по ключу email, возвращая его значение

job = info.pop('job') # удаляем элемент по ключу job, возвращая его значение

print(email)

print(job)

print(info)

выводит:

timyr-guev@yandex.ru

Teacher

{'name': 'Sam', 'age': 28}

Единственное отличие этого способа удаления от оператора del — он возвращает удаленное значение. В остальном этот способ идентичен оператору del. В частности, если удаляемого ключа в словаре нет, возникнет ошибка KeyError.

​Чтобы ошибка не появлялась, этому методу можно передать второй аргумент. Он будет возвращен, если указанного ключа в словаре нет. Это позволяет реализовать безопасное удаление элемента из словаря:

surname = info.pop('surname', None)

Если ключа surname в словаре нет, то в переменной surname будет храниться значение None.

### Метод popitem()

Метод popitem() удаляет из словаря **последний добавленный элемент** и возвращает удаляемый элемент в виде кортежа (ключ, значение).

Следующий программный код:

info = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

info['surname'] = 'Sinclar'

item = info.popitem()

print(item)

print(info)

выводит:

('surname', 'Sinclar')

{'name': 'Bob', 'age': 25, 'job': 'Dev'}

​   В версиях Python ниже 3.6 метод popitem() удалял случайный элемент.

### Метод clear()

Метод clear() удаляет все элементы из словаря.

Следующий программный код:

info = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

info.clear()

print(info)

выведет:

{}

### Метод copy()

Метод copy() создает **поверхностную** копию словаря.

Следующий программный код:

info = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

info\_copy = info.copy()

print(info\_copy)

выведет:

{'name': 'Bob', 'age': 25, 'job': 'Dev'}

Не стоит путать копирование словаря (метод copy()) и присвоение новой переменной ссылки на старый словарь.

Следующий программный код:

info = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

new\_info = info

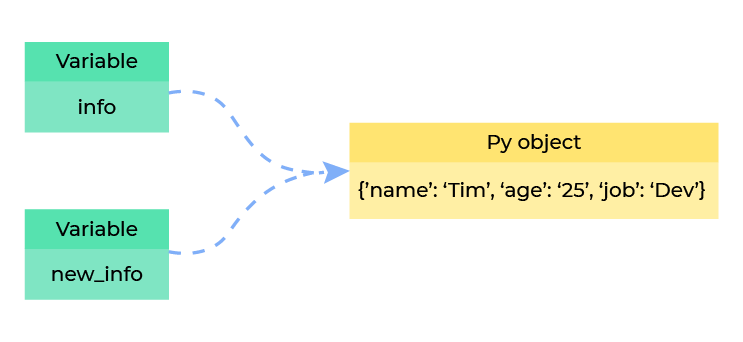
new\_info['name'] = 'Tim'

print(info)

выводит:

{'name': 'Tim', 'age': 25, 'job': 'Dev'}

Оператор присваивания (=) не копирует словарь, а лишь присваивает ссылку на старый словарь новой переменной.



Таким образом, когда мы изменяем словарь new\_info, меняется и словарь info. Если необходимо изменить один словарь, не изменяя второй, используют метод copy().

Следующий программный код:​

info = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

new\_info = info.copy()

new\_info['name'] = 'Tim'

print(info)

print(new\_info)

выводит:

{'name': 'Bob', 'age': 25, 'job': 'Dev'}

{'name': 'Tim', 'age': 25, 'job': 'Dev'}

## Примечания

**Примечание 1.** Словарь можно использовать вместо нескольких вложенных условий, если вам нужно проверить число на равенство. Например вместо

num = int(input())

if num == 1:

description = 'One'

elif num == 2:

description = 'Two'

elif num == 3:

description = 'Three'

else:

description = 'Unknown'

print(description)

можно написать

num = int(input())

description = {1: 'One', 2: 'Two', 3: 'Three'}

print(description.get(num, 'Unknown'))

На практике, такой код встречается достаточно часто. Особенно если в программе необходимо часто осуществлять проверку указанного типа.

# Тема урока: вложенные словари, генераторы словарей

1. Вложенные словари
2. Генераторы словарей

**Аннотация.** Урок посвящен вложенным словарям и генераторам словарей.

## Вложенные словари

Словари могут содержать другие словари, которые сами, в свою очередь, содержат словари, и так далее на любую глубину. Такие словари называются **вложенными словарями**(мы уже сталкивались с вложенными списками и кортежами). Вложенные словари – один из способов представления структурированной информации.

### Создание вложенных словарей

Вложенный словарь создается как обычный, только каждое значение в нем – другой словарь.

Приведенный ниже код создает словарь для хранения информации о сотрудниках некоторой компании.

info = {'emp1': {'name': 'Timur', 'job': 'Teacher'},

'emp2': {'name': 'Ruslan', 'job': 'Developer'},

'emp3': {'name': 'Rustam', 'job': 'Tester'}}

Тот же самый словарь info может быть создан по другому:

info = dict(emp1 = {'name': 'Timur', 'job': 'Teacher'},

emp2 = {'name': 'Ruslan', 'job': 'Developer'},

emp3 = {'name': 'Rustam', 'job': 'Tester'})

или

ids = ['emp1', 'emp2', 'emp3']

emp\_info = [{'name': 'Timur', 'job': 'Teacher'},

{'name': 'Ruslan', 'job': 'Developer'},

{'name': 'Rustam', 'job': 'Tester'}]

info = dict(zip(ids, emp\_info))

   Число уровней вложенности словарей неограниченно!

### Обращение к элементам вложенного словаря

Для того, чтобы получить значения определенных элементов во вложенном словаре, необходимо указать их ключи в нескольких квадратных скобках подобно тому, как мы получали значения во вложенных списках.

Приведенный ниже код:

info = {'emp1': {'name': 'Timur', 'job': 'Teacher'},

'emp2': {'name': 'Ruslan', 'job': 'Developer'},

'emp3': {'name': 'Rustam', 'job': 'Tester'}}

print(info['emp1']['name'])

print(info['emp2']['job'])

выводит:

Timur

Developer

При попытке обращения по несуществующему ключу возникнет ошибка KeyError. Для того, чтобы избежать возникновения такой ошибки, используйте словарный метод get(), который по умолчанию возвращает значение None, если ключ отсутствует в словаре.

### Изменение вложенных словарей

Чтобы изменить значение определенного элемента во вложенном словаре, необходимо обратиться к его ключу и использовать оператор присвоения (=).

Приведенный ниже код:

info = {'emp1': {'name': 'Timur', 'job': 'Teacher'},

'emp2': {'name': 'Ruslan', 'job': 'Developer'},

'emp3': {'name': 'Rustam', 'job': 'Tester'}}

info['emp1']['job'] = 'Manager'

print(info['emp1'])

выводит:

{'name': 'Timur', 'job': 'Manager'}

### Итерация по вложенным словарям

Итерации по вложенным словарям осуществляются как правило двумя циклами for (иногда достаточно одного цикла).

Приведенный ниже код:

info = {'emp1': {'name': 'Timur', 'job': 'Teacher'},

'emp2': {'name': 'Ruslan', 'job': 'Developer'},

'emp3': {'name': 'Rustam', 'job': 'Tester'}}

for emp in info:

print('Employee ID:', emp)

for key in info[emp]:

print(key + ':', info[emp][key])

print()

выводит:

Employee ID: emp1

name: Timur

job: Teacher

Employee ID: emp2

name: Ruslan

job: Developer

Employee ID: emp3

name: Rustam

job: Tester

Аналогичного результата можно было добиться с помощью метода items():

info = {'emp1': {'name': 'Timur', 'job': 'Teacher'},

'emp2': {'name': 'Ruslan', 'job': 'Developer'},

'emp3': {'name': 'Rustam', 'job': 'Tester'}}

for emp, inf in info.items():

print('Employee ID:', emp)

for key in inf:

print(key + ':', inf[key])

print()

## Генераторы словарей

Пусть требуется создать словарь, ключами которого будут числа от 00 до 55, а значениями – квадраты ключей.

Для создания требуемого словаря можно использовать код:

squares = {}

squares[0] = 0

squares[1] = 1

squares[2] = 4

squares[3] = 9

squares[4] = 16

squares[5] = 25

или:

squares = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

или:

squares = {}

for i in range(6):

squares[i] = i\*\*2

Первые два способа слишком громоздкие и не подходят, когда чисел много. Третий способ полностью решает поставленную задачу, однако его можно переписать в стиле Python с использованием генератора словаря:

squares = {i: i\*\*2 for i in range(6)}

Общий вид генератора словаря следующий:

{ключ: значение for переменная in последовательность}

где переменная — имя некоторой переменной, последовательность — последовательность значений, которые она принимает (любой итерируемый объект), ключ: значение — некоторое выражение, как правило, зависящее от использованной в списочном выражении переменной, которой будут заполнены элементы словаря.

### **Примеры использования генератора словарей**

Генераторы словаря могут выполнять итерации по любому итерируемому объекту: списки, кортежи, строки, словари.

**1. Генератор словаря при итерировании по строке.**

Приведенный ниже код:

dct = {c: c \* 3 for c in 'ORANGE'}

print(dct)

выводит:

{'O': 'OOO', 'R': 'RRR', 'A': 'AAA', 'N': 'NNN', 'G': 'GGG', 'E': 'EEE'}

**2. Для вычисления ключа и значения в генераторе словаря могут быть использованы выражения.**

В следующем примере для вычисления ключа используется метод lower(), а для вычисления значения – метод upper().

Приведенный ниже код:

lst = ['ReD', 'GrEeN', 'BlUe']

dct = {c.lower(): c.upper() for c in lst}

print(dct)

выводит:

{'red': 'RED', 'green': 'GREEN', 'blue': 'BLUE'}

**3. Извлечение из словаря элементов с определенными ключами.**

Приведенный ниже код:

dict1 = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}

selected\_keys = [0, 2, 5]

dict2 = {k: dict1[k] for k in selected\_keys}

print(dict2)

выводит:

{0: 'A', 2: 'C', 5: 'F'}

### Условия в генераторе словарей

В генераторах словарей можно использовать условный оператор.

Приведенный ниже код создает словарь, ключами которого являются четные числа от 00 до 99, а значениями – квадраты ключей:

squares = {}

for i in range(10):

if i % 2 == 0:

squares[i] = i\*\*2

print(squares)

Такой код можно переписать с помощью генератора словаря в виде:

squares = {i: i\*\*2 for i in range(10) if i % 2 == 0}

print(squares)

Не забываем про возможность указания шага в функции range(). Предыдущий код можно записать и без условного оператора: squares = {i: i\*\*2 for i in range(0, 10, 2)}.

### Генераторы вложенных словарей

Мы также можем использовать генераторы словарей для создания вложенных словарей:

Приведенный ниже код:

squares = {i: {j: j\*\*2 for j in range(i + 1)} for i in range(5)}

for value in squares.values():

print(value)

выводит:

{0: 0}

{0: 0, 1: 1}

{0: 0, 1: 1, 2: 4}

{0: 0, 1: 1, 2: 4, 3: 9}

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Тема урока: модуль random

1. Случайные числа
2. Псевдослучайные числа
3. Модуль random

**Аннотация.** Урок посвящен модулю random, который содержит функции по работе с псевдослучайными числами.

## Случайные числа

**Случайные числа** – последовательность чисел, в которой невозможно предсказать следующее число, зная все предыдущие.

**Случайные числа** широко используются в различных задачах программирования:

* в играх (имитация подбрасывания игрального кубика и другие подобные ситуации);
* в программах имитационного моделирования;
* в статистических программах, случайным образом отбирающих данные для анализа;
* в компьютерной безопасности для шифрования уязвимых данных.

Для создания истинно случайных чисел можно бросать монету, игральные кости, или измерять какой-нибудь шумовой сигнал.

## ****Псевдослучайные числа****

Ставить сложные электронные приборы на каждый компьютер для генерации истинно случайных чисел дорого, поэтому математики и программисты создали алгоритмы получения **псевдослучайных чисел**.

Для обычного человека псевдослучайные числа практически не отличаются от случайных, однако они вычисляются по математической формуле, и зная первое число и формулу, можно определить любое следующее.

Python предлагает встроенные функции для работы с псевдослучайными числами. Эти функции хранятся в модуле random в стандартной библиотеке.

## Модуль random

Модуль random предоставляет функции для генерации псевдослучайных чисел, букв и случайного выбора элементов последовательности (списка, строки и т.д.).

Для использования этих функций в начале программы необходимо подключить модуль, что делается командой **import**:

import random

После подключения модуля мы можем использовать его функции.

**Функция randint()**

Функция randint() принимает два обязательных аргумента a и b и возвращает псевдослучайное целое число из отрезка [*a*;*b*].

Следующий код выводит два псевдослучайных целых числа: num1 из отрезка [0;17] и num2 из отрезка [−5;5].

import random

num1 = random.randint(0, 17)

num2 = random.randint(-5, 5)

Левая и правая граница a и b включаются в диапазон генерируемых псевдослучайных чисел. Результатом вызова функции random.randint(2, 9) может быть любое число от 22 до 99, включая 22 и 99.

Следующий код выводит 10 псевдослучайных целых чисел из диапазона [1;100]:

import random

for \_ in range(10):

print(random.randint(1, 100))

Среди этих чисел возможны повторения.

### Функция randrange()

Если вы помните, как применять функцию range(), то почувствуете себя непринужденно с функцией randrange(). Функция randrange() принимает такие же аргументы, что и функция range(). Различие состоит в том, что функция randrange()  
не возвращает саму последовательность чисел. Вместо этого она возвращает случайно выбранное число из последовательности чисел.

Следующий код присваивает переменной num псевдослучайное число в диапазоне от 00 до 99:

import random

num = random.randrange(10)

Аргумент 1010 задает конечный предел последовательности значений. Функция возвратит случайно выбранное число из последовательности чисел от 00 до конечного предела, исключая сам предел.

Следующий код задает начальное значение и конечный предел последовательности:

import random

num = random.randrange(5, 10)

Таким образом в переменной num будет храниться псевдослучайное число в диапазоне от 55 до 99.

Следующий код задает начальное значение, конечный предел и величину шага:

import random

num = random.randrange(0, 101, 10)

Таким образом в переменной num будет храниться псевдослучайное число из последовательности чисел: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

### Функция random()

Функции randint() и randrange() возвращают псевдослучайное **целое число**. А вот функция random() возвращает псевдослучайное число с плавающей точкой (вещественное число). В функцию random() никаких аргументов не передается. Функция random() возвращает случайное число с плавающей точкой в диапазоне от 0.0 до 1.0 (исключая 1.0).

Следующий код выводит случайное число с плавающей точкой из диапазона [0.0;1.0):

import random

num = random.random()

print(num)

### Функция uniform()

Функция uniform() тоже возвращает случайное число с плавающей точкой, но при этом она позволяет задавать диапазон для отбора значений.

Следующий код выводит псевдослучайное число с плавающей точкой из диапазона [1.5;17.3] (включительно):

import random

num = random.uniform(1.5, 17.3)

print(num)

### Функция seed()

Как уже было сказано **псевдослучайные числа** вычисляются на основе некой формулы. Генерация случайных чисел инициируется **начальным значением**. Оно используется в вычислении, возвращающем следующее случайное число в ряду. Когда модуль random импортируется, он получает системное время из внутреннего генератора тактовых импульсов компьютера и использует его как начальное значение. Системное время – целое число, представляющее собой текущую дату и время вплоть до сотой секунды. Если бы всегда использовалось одно и то же начальное значение, функции генерации случайных чисел всегда  возвращали бы один и тот же ряд псевдослучайных чисел. Поскольку системное время меняется каждую сотую долю секунды, можно утверждать, что всякий раз, когда импортируется модуль random, будет создана отличающаяся от предыдущих последовательность случайных чисел.

Вместе с тем, некоторые программы требуют генерации одной и той же последовательности случайных чисел. Для этого можно вызвать функцию seed(), задав начальное значение.

Следующий код генерирует 10 псевдослучайных чисел, и при этом содержит инструкцию, явно устанавливающую начальное значение для генератора случайных чисел:

import random

random.seed(17) # явно устанавливаем начальное значение для генератора случайных чисел

for \_ in range(10):

print(random.randint(1, 100))

Результат выполнения такого кода:

67

54

39

47

38

23

99

91

91

70

Если выполнить код еще раз, получим ту же самую последовательность псевдослучайных чисел.

## Примечания

**Примечание 1.** Подключение модуля следующим образом:

from random import \*

позволяет в дальнейшем не писать название модуля и символ точки при вызове функций модуля.

**Примечание 2.** Функции модуля random на самом деле являются **методами**одноименного класса random.

**Примечание 3.** В Python для генерации псевдослучайных чисел используется один из самых совершенных алгоритмов генерации псевдослучайных чисел – ["вихрь Мерсенна"](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%85%D1%80%D1%8C_%D0%9C%D0%B5%D1%80%D1%81%D0%B5%D0%BD%D0%BD%D0%B0), разработанный в 1997 году. Реализация выполнена на языке *C*, является быстрой и потокобезопасной.

**Примечание 4.** Настоящие случайные числа можно получить с [сайта](https://www.random.org/randomness/). Данный сайт использует атмосферный шум для создания по-настоящему случайных чисел.

**Примечание 5.** Пусть *r* – случайное число из интервала (0;1). Для того, чтобы перевести такое число в интервал (*a*;*b*) можно воспользоваться формулой *a*+(*b*−*a*)⋅*r*.

**Тема урока: модуль random**

1. Метод shuffle()
2. Метод choice()
3. Метод sample()
4. Модуль string

**Аннотация.** Урок посвящен модулю random, в частности, методам работы с последовательностями.

**Метод shuffle()**

Метод shuffle() принимает список в качестве обязательного аргумента и перемешивает его случайным образом.

Следующий код перемешивает список numbers случайным образом, а затем выводит его содержимое:

import random

numbers = [1, 2, 3, 4, 5, 6, 7, 8]

random.shuffle(numbers)

print(numbers)

Результатом работы такого кода может быть:

[4, 7, 8, 1, 2, 3, 6, 5]

**Метод choice()**

Метод choice() принимает список (строку, кортеж) в качестве обязательного аргумента и возвращает **один** случайный элемент.

Следующий код выводит по одному случайному элементу из строки 'BEEGEEK', списка [1, 2, 3, 4] и кортежа  ('a', 'b', 'c', 'd'):

import random

print(random.choice('BEEGEEK'))

print(random.choice([1, 2, 3, 4]))

print(random.choice(('a', 'b', 'c', 'd')))

Результатом работы такого кода может быть:

E

3

c

**Метод sample()**

Метод sample() принимает два обязательных аргумента: первый – список (строка, кортеж, множество), второй – количество случайных элементов. Возвращает **список** из указанного количества уникальных (имеющих разные индексы) случайных элементов.

Результатом работы кода:

import random

numbers = [2, 5, 8, 9, 12]

print(random.sample(numbers, 1))

print(random.sample(numbers, 2))

print(random.sample(numbers, 3))

print(random.sample(numbers, 5))

может быть:

[9]

[12, 5]

[9, 2, 8]

[12, 8, 9, 5, 2]

Количество случайных элементов не должно превышать длину исходного списка (строки). Следующий код:

import random

numbers = [2, 5, 8, 9, 12]

print(random.sample(numbers, 6))

приведет к ошибке:

ValueError: Sample larger than population or is negative

**Модуль string**

Встроенный модуль string раньше использовался для расширения стандартных возможностей (функционала) строкового типа данных str. На текущий момент все функции из модуля string переехали в методы строкового типа данных str, однако в модуле string остались удобные константные строки, которые можно использовать при решении задач.

Приведенный ниже код:

import string

print(string.ascii\_letters)

print(string.ascii\_uppercase)

print(string.ascii\_lowercase)

print(string.digits)

print(string.hexdigits)

print(string.octdigits)

print(string.punctuation)

print(string.printable)

выводит:

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

0123456789

0123456789abcdefABCDEF

01234567

!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~

0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~ \t\n\r\x0b\x0c

**Примечания**

**Примечание 1.** Помимо рассмотренных в уроке методов, модуль random содержит много дополнительных методов. Подробнее о модуле random можно почитать в [документации](https://docs.python.org/3/library/random.html).

**Метод Монте-Карло**

Метод Монте-Карло – группа численных методов, основанных на воспроизведении и статистическом анализе большого числа реализаций случайного процесса. Проводится математическое моделирование случайных процессов и параметры их реализации оцениваются статистически. Метод лежит в основе статистического моделирования.

   Метод Монте-Карло часто называют методом статистических испытаний.

|  |  |  |
| --- | --- | --- |
| Основоположники метода Монте-Карло | | |
| Станислав Улам | Джон фон Нейман | Николас Метрополис |

Статистическое моделирование широко применяется для решения задач из различных областей человеческого знания - биологии, химии, физики, экономики и других. Задачи, где широко используется этот подход:

* численное интегрирование;
* расчеты в системах массового обслуживания;
* расчеты качества и надежности изделий;
* расчеты прохождения нейтронов и других частиц через вещество;
* передача сообщений при наличии помех;
* задачи теории игр;
* задачи динамики разреженного газа;
* задачи дискретной оптимизации;
* задачи финансовой математики.

**Вычисление площадей**

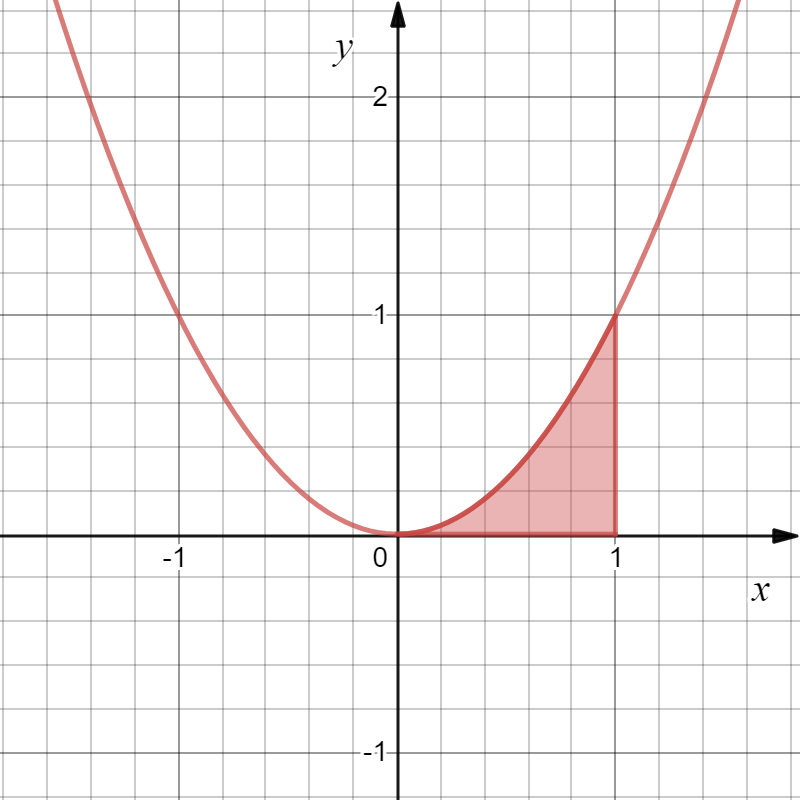
Применим метод Монте-Карло и к задаче вычисления площади геометрической фигуры на плоскости.

Поместим фигуру в прямоугольник и будем наугад бросать точки в этот прямоугольник. Будем исходить из того, что чем больше площадь фигуры, тем чаще в нее будут попадать точки. Таким образом, при большом числе �*n* точек, наугад выбранных внутри прямоугольника, доля точек, содержащихся в данной фигуре �*k*, приближенно равна отношению площади этой фигуры и площади прямоугольника.

Если площадь прямоугольника равна �0*S*0​ и в результате испытаний, из которых при �*k* исходах случайные точки оказались внутри фигуры, то площадь �*S* фигуры будет определяться выражением�=��⋅�0.*S*=*nk*​⋅*S*0​. Рассмотрим алгоритм решения задачи на конкретных примерах.

**Пример 1.** Рассмотрим фигуру, множество точек которой определяется следующей системой неравенств:

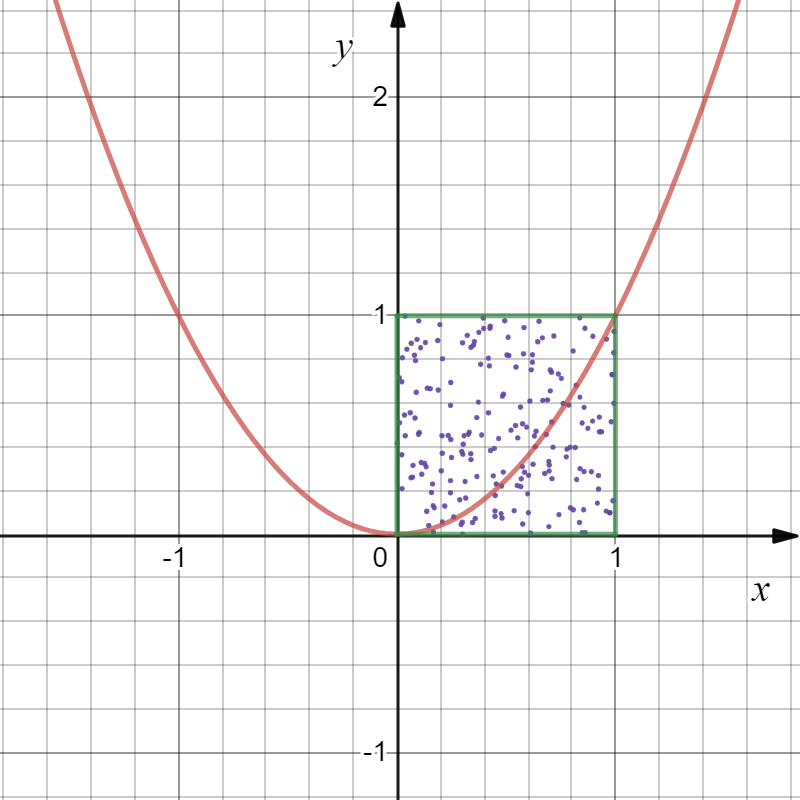
{0≤�≤10≤�≤1�≤�2⎩⎨⎧​0≤*x*≤10≤*y*≤1*y*≤*x*2​Графическое изображение указанной фигуры в плоскости ���*xOy* имеет вид:



Площадь такой фигуры �=13≈ 0.33333*S*=31​≈ 0.33333 (считается через интеграл).

**Алгоритм Монте-Карло:**площадь искомой фигуры составляет часть квадрата со сторонами 11. Площадь такого квадрата равна 11.

1. Генерируем случайные числа �*x* и �*y* равномерно распределенные на отрезке [0;1][0;1]. Это будут координаты случайной точки в квадрате, в которую заключена фигура, площадь которой требуется найти. Полученная точка может как попасть в исследуемую фигуру, так и не попасть.
2. Проверяем принадлежность точки к исследуемой фигуре. Если попадания нет, т.е. не выполняется неравенство �≤�2*y*≤*x*2, то переходим к пункту 1 и генерируем координаты новой точки. Если попадание есть, то фиксируем это попадание, то есть увеличиваем на единицу значение счетчика числа попаданий и снова переходим к пункту 1.



**Примечание.** Заметим, что попадание случайной точки точно на границу фигуры можно отнести как к первому, так и ко второму исходу.

Пункты 11 и 22 следует повторить в цикле достаточно большое количество �*n* раз. От количества повторений напрямую зависит точность результата. После проведения �*n* повторов площадь фигуры найдем по формуле:�=��⋅�0.*S*=*nk*​⋅*S*0​.

Пример программы на Python:

import random

n = 1000

k = 0

s0 = 1

for \_ in range(n):

x = random.uniform(0, 1) # случайное число с плавающей точкой от 0 до 1

y = random.uniform(0, 1) # случайное число с плавающей точкой от 0 до 1

if y <= x\*\*2: # если попадает в нужную область

k += 1

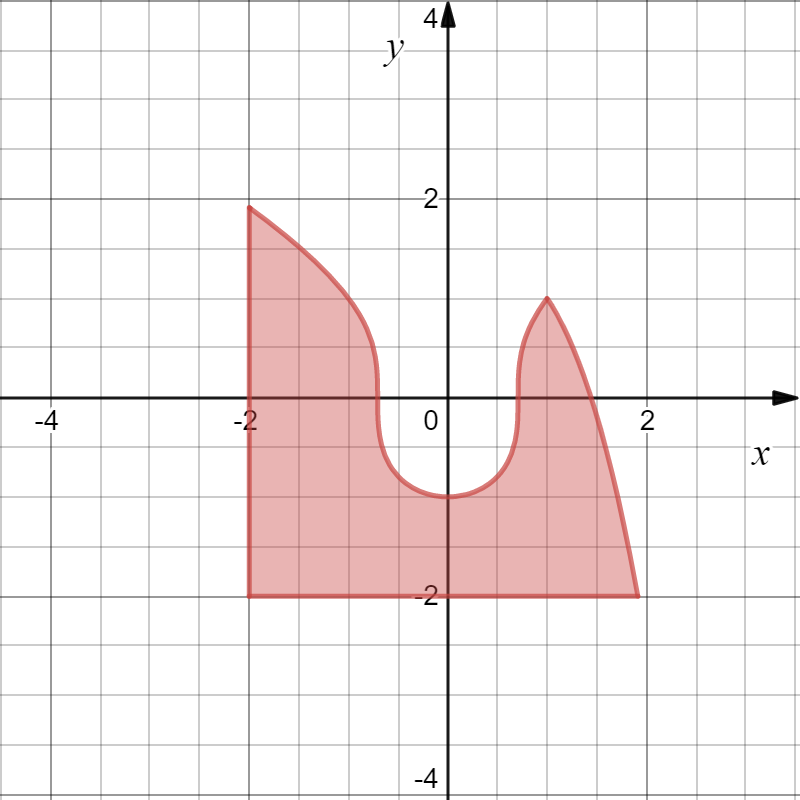
print((k/n)\*s0)

Составим таблицу:

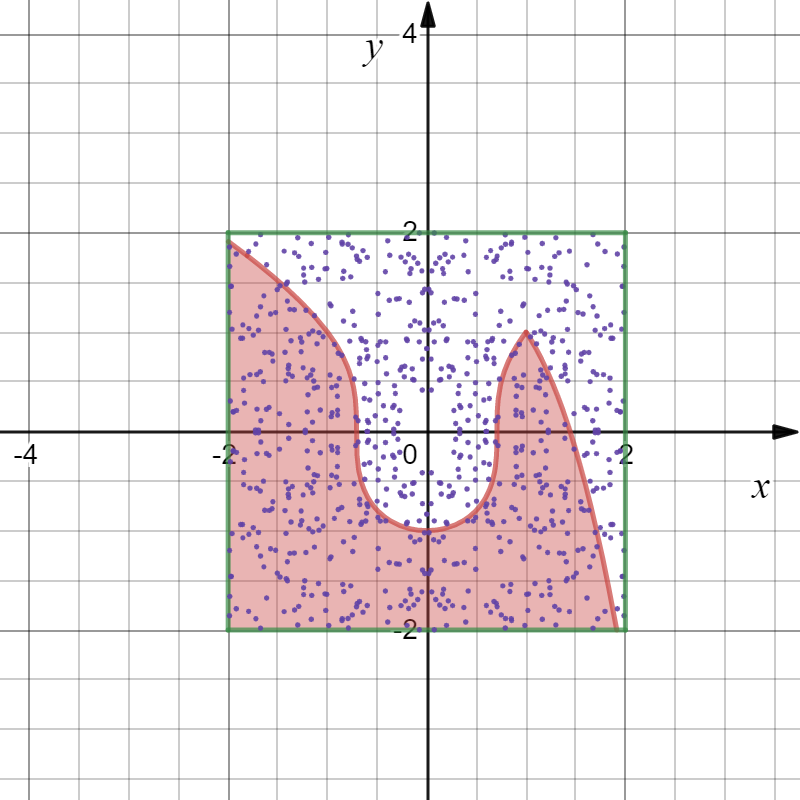
|  |  |
| --- | --- |
| �*n* | �*S* |
| 1010 | 0.60.6 |
| 100100 | 0.370.37 |
| 10001000 | 0.3370.337 |
| 1000010000 | 0.33370.3337 |
| 100000100000 | 0.333290.33329 |
| 10000001000000 | 0.3335710.333571 |
| 1000000010000000 | 0.33328280.3332828 |
| 100000000100000000 | 0.333367070.33336707 |

**Пример 2.** Рассмотрим фигуру, множество точек которой определяется следующей системой неравенств:

{−2≤�≤2−2≤�≤2�3−2�2≤−12�+�3≤3⎩⎨⎧​−2≤*x*≤2−2≤*y*≤2*y*3−2*x*2≤−12*y*+*x*3≤3​Графическое изображение указанной фигуры в плоскости ���*xOy* имеет вид:



Площадь этой фигуры нам заранее известна и равна �=8.38404*S*=8.38404.



**Алгоритм Монте-Карло:**площадь искомой фигуры составляет часть квадрата со сторонами 44. Площадь такого квадрата равна 1616.

1. Генерируем случайные числа �*x* и �*y*, равномерно распределенные на отрезках [−2;2][−2;2]. Это будут координаты случайных точек в квадрате, в который заключена фигура искомой площади. Полученная точка может как попасть в исследуемую фигуру, так и не не попасть.
2. Проверяем принадлежность точки к исследуемой фигуре. Если попадания нет, не выполняется хотя бы одно из  неравенств �3−2�2≤−1*y*3−2*x*2≤−1 или 2�+�3≤32*y*+*x*3≤3, переходим к пункту 11 и генерируем координаты новой точки. Если попадание есть, фиксируем это попадание, то есть увеличиваем на единицу значение счетчика числа попаданий и снова переходим к пункту 11.

import random

n = 1000

k = 0

s0 = 16

for \_ in range(n):

x = random.uniform(-2, 2)

y = random.uniform(-2, 2)

if y\*\*3 - 2\*x\*\*2 <= -1 and 2\*y + x\*\*3 <= 3:

k += 1

print((k/n)\*s0)

Составим таблицу:

|  |  |
| --- | --- |
| �*n* | �*S* |
| 1010 | 9.69.6 |
| 100100 | 8.968.96 |
| 10001000 | 8.2248.224 |
| 1000010000 | 8.38248.3824 |
| 100000100000 | 8.365288.36528 |
| 10000001000000 | 8.3833768.383376 |
| 1000000010000000 | 8.38415848.3841584 |
| 100000000100000000 | 8.384041348.38404134 |

**Примечания**

**Примечание 1.** Свое экзотическое название метод получил от города Монте-Карло в княжестве Монако, известного благодаря казино, поскольку именно рулетка является одним из самых широко известных генераторов случайных чисел. Станислав Улам пишет в своей автобиографии «Приключения математика», что название было предложено Николасом Метрополисом в честь его дяди, азартного игрока.



**Примечание 2.** Активное применение метода началось с появлением ЭВМ, способных выполнять сотни операций для получения необходимых статистических данных. Развитие метода Монте-Карло пришлось на 1950-е годы, когда его использовали ученые из лаборатории ВВС США и исследовательской корпорации RAND, работающие над созданием водородной бомбы, в том числе и гениальный ученый Джон фон Нейман. Неймана считают одним из основателей метода, как, впрочем, и самих ЭВМ.

**Примечание 3.** Аналогичным образом можно вычислять объемы тел в пространстве.

**Примечание 4.** Подробнее о методе Монте-Карло можно почитать [тут](https://ru.qaz.wiki/wiki/Monte_Carlo_method).

# Тема урока: модуль decimal

1. Числа с плавающей точкой float
2. Модуль decimal
3. Тип данных Decimal

**Аннотация.** Урок посвящен модулю decimal и типу данных Decimal.

## Числовые типы данных

В прошлых уроках мы изучили два числовых типа данных, представленных в Python:

* int – целое число;
* float – число с плавающей точкой (аналог вещественного числа в математике).

В Python есть три дополнительных числовых типа данных:

* Decimal – десятичное число, для выполнения точных расчетов;
* Fraction – число, представляющее собой [обыкновенную дробь](https://ru.wikipedia.org/wiki/%D0%94%D1%80%D0%BE%D0%B1%D1%8C_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)#.D0.9E.D0.B1.D1.8B.D0.BA.D0.BD.D0.BE.D0.B2.D0.B5.D0.BD.D0.BD.D1.8B.D0.B5_.D0.B4.D1.80.D0.BE.D0.B1.D0.B8), с заданным числителем и знаменателем;
* Complex – комплексное число.

В этом уроке мы изучим числовой тип данных Decimal, аналог типа данных float на случай более точных вычислений.

## Тип данных float

Рассмотрим программный код:

if 0.3 == 0.3:

print('YES')

else:

print('NO')

Результатом выполнения такого кода будет как и полагается YES.

А теперь рассмотрим программный код:

num = 0.1 + 0.1 + 0.1

if num == 0.3:

print('YES')

else:

print('NO')

Вы будете удивлены, но результатом выполнения такого кода будет NO, так как на самом деле в переменной num хранится что-то типа 0.300000000000000040.30000000000000004.

Из-за ограничений в сохранении точного значения чисел, даже простейшие математические операции могут выдавать ошибочный результат. Поэтому, чтобы сравнивать два float числа мы должны использовать такой код:

num = 0.1 + 0.1 + 0.1

eps = 0.000000001 # точность сравнения

if abs(num - 0.3) < eps: # число num отличается от числа 0.3 менее чем 0.000000001

print('YES')

else:

print('NO')

Такой код выводит, как полагается, значение YES.

Не стоит сравнивать float числа с помощью оператора ==. Для сравнения float чисел нужно использовать указанный выше код.

## Тип данных Decimal

Тип данных Decimal – это класс из стандартного модуля decimal. Он представляет собой число с плавающей точкой, как и float. Однако, Decimal имеет ряд существенных отличий от float.

Тип Decimal создан, чтобы операции над вещественными числами в компьютере выполнялись как в математике, и равенство 0.1+0.1+0.1==0.30.1+0.1+0.1==0.3 было верным.

Точность результатов арифметических действий очень важна для научных вычислений, в сфере финансов и бизнеса. Для таких задач тип данных float не подходит.

В Python тип данных float реализован по стандарту IEEE-754 как число с плавающей точкой двойной точности (6464 бита) с основанием экспоненты равным 22. Реализация таких чисел заложена прямо в железо любого современного процессора. Поэтому float в Python работает как аналогичный тип данных double в таких языках программирования как С#, С++, Java и т.д. И имеет такие же ограничения и «странности». Так как float поддерживается аппаратно, быстродействие при использовании этого типа данных сравнительно велико.

Тип данных Decimal – число с плавающей точкой с основанием экспоненты 1010. Он реализован по стандарту IBM: General Decimal Arithmetic Specification, в свою очередь основанному на стандартах IEEE.

Тип данных Decimal реализован программно, поэтому он в разы медленнее типа данных float, реализованного аппаратно. Сам тип данных Decimal написан на языке С.

Тип данных Decimal оперирует числами с произвольной – задаваемой программистом, но конечной точностью. По умолчанию точность составляет 2828 десятичных знаков.

Тип данных Decimal неизменяемый. Операции над ним приводят к созданию новых объектов, при этом старые не меняются.

Еще одно следствие того, что Decimal реализован программно – его можно на ходу настраивать, как угодно программисту. Для этого есть контекст – объект, содержащий настройки для выполнения операций. Операции, выполняемые в контексте, следуют заданным в нем правилам. Для float все правила фиксированы на аппаратном уровне.

Для типа данных Decimal можно настроить:

* точность выполнения операций в количестве десятичных знаков;
* режимы округления;
* режимы обработки исключительных ситуаций (деление на ноль, переполнение и т. д).

### Создание Decimal чисел

Создать Decimal число можно из обычного целого числа (int), из числа с плавающей точкой (float) или из строки (str).

Приведенный ниже программный код создает Decimal числа на основе целого числа и строки:

from decimal import \*

d1 = Decimal(1)

d2 = Decimal(567)

d3 = Decimal(-93)

d4 = Decimal('12345')

d5 = Decimal('52.198')

print(d1, d2, d3, d4, d5, sep='\n')

и выводит:

1

567

-93

12345

52.198

При создании Decimal чисел из чисел с плавающей точкой (float) возникают проблемы, так как float числа округляются внутри до ближайшего возможного, а Decimal об этом ничего не знает и копирует содержимое float.

Приведенный ниже программный код создает Decimal число на основе числа с плавающей точкой:

from decimal import \*

num = Decimal(0.1)

print(num)

и выводит:

0.1000000000000000055511151231257827021181583404541015625

Не рекомендуется создавать Decimal числа из float чисел. В Decimal попадет уже неправильно округленное число. Создавать Decimal числа нужно из целых чисел, либо из строк!

### Арифметические операции над Decimal числами

Тип данных Decimal отлично интегрирован в язык Python. С Decimal числами работают все привычные операции: сложение, вычитание, умножение, деление, возведение в степень.

Приведенный ниже код:

from decimal import \*

num1 = Decimal('5.2')

num2 = Decimal('2.3')

print(num1 + num2)

print(num1 - num2)

print(num1 \* num2)

print(num1 / num2)

print(num1 // num2)

print(num1 \*\* num2)

выводит:

7.5

2.9

11.96

2.260869565217391304347826087

2

44.34122533787992500412791298

Можно совершать арифметические операции над Decimal и целыми числами (миксовать Decimal и int), но не рекомендуется смешивать их с float.

Приведенный ниже код:

from decimal import \*

num = Decimal('5.2')

print(num + 1)

print(num - 10)

print(num \* 2)

print(num \*\* 4)

выводит:

6.2

-4.8

10.4

731.1616

### Математические функции

Decimal числа можно передавать как аргументы функциям, ожидающим float. Они будут преобразованы во float. К примеру, модуль math, оперирующий float числами, может работать и с Decimal числами.

Приведенный ниже код:

from decimal import \*

from math import \*

num1 = Decimal('1.44')

num2 = Decimal('0.523')

print(sqrt(num1))

print(sin(num2))

print(log(num1 + num2))

выводит:

1.2

0.4994813555186418

0.6744739152943241

Важно понимать, что результатом работы функции модуля math являются float числа, а не Decimal.

Тип данных Decimal содержит некоторые встроенные математические методы, возвращающие значения Decimal.

| **Функция** | **Описание** |
| --- | --- |
| sqrt() | вычисляет квадратный корень из Decimal числа |
| exp() | возвращает ��*ex* для Decimal числа |
| ln() | вычисляет натуральный логарифм (по основанию �*e*) Decimal числа |
| log10() | вычисляет десятичный логарифм (по основанию 1010) Decimal числа |

Приведенный ниже код:

from decimal import \*

num = Decimal('10.0')

print(num.sqrt())

print(num.exp())

print(num.ln())

print(num.log10())

выводит:

3.162277660168379331998893544

22026.46579480671651695790065

2.302585092994045684017991455

1

Обратите внимание на количество знаков в записи чисел. Их 2828, что соответствует точности десятичного числа по умолчанию.

Тип данных Decimal также содержит полезный метод as\_tuple() который возвращает кортеж из 33 элементов:

* sign – знак числа (00 для положительного числа и 11 для отрицательного числа);
* digits – цифры числа;
* exponent – значение экспоненты (количество цифр после точки, умноженное на −1−1),

Приведенный ниже код:

from decimal import \*

num1 = Decimal('-1.4568769017')

num2 = Decimal('0.523')

print(num1.as\_tuple())

print(num2.as\_tuple())

выводит:

DecimalTuple(sign=1, digits=(1, 4, 5, 6, 8, 7, 6, 9, 0, 1, 7), exponent=-10)

DecimalTuple(sign=0, digits=(5, 2, 3), exponent=-3)

Приведенный ниже код:

from decimal import \*

num = Decimal('-1.4568769017')

num\_tuple = num.as\_tuple()

print(num\_tuple.sign)

print(num\_tuple.digits)

print(num\_tuple.exponent)

выводит:

1

(1, 4, 5, 6, 8, 7, 6, 9, 0, 1, 7)

-10

### Работа с контекстом Decimal чисел

Базовые параметры Decimal можно посмотреть в его **контексте**, выполнив функцию getcontext().

Приведенный ниже код:

from decimal import \*

print(getcontext())

выводит:

Context(prec=28, rounding=ROUND\_HALF\_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0,

flags=[], traps=[InvalidOperation, DivisionByZero, Overflow])

Мы видим здесь, что точность 2828 знаков, округление к ближайшему четному, пределы по экспоненте ± 999999± 999999, capitals – это про заглавную Е при печати, включенные ловушки – неправильная операция, деление на ноль, переполнение.

### Точность чисел

Контекстом в Decimal можно управлять, устанавливая свои значения. Например, чтобы управлять точностью Decimal, необходимо изменить параметр контекста prec (от англ. precision – точность). При этом точность вступает в силу только во время арифметических операций, а не при создании самих чисел.

Приведенный ниже код:

from decimal import \*

getcontext().prec = 3 # устанавливаем точность в 3 знака

num = Decimal('3.1415')

print(num)

print(num \* 1)

print(num \* 2)

print(num / 2)

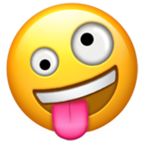
выводит:

3.1415

3.14

6.28

1.57

Обратите внимание на то, что точность вступает в силу только во время арифметических операций, а не при создании самих чисел.

### Округление чисел

Округляют числа Decimal с помощью метода quantize(). Этот метод в качестве первого аргумента принимает объект Decimal, указывающий на формат округления.

Приведенный ниже код:

from decimal import \*

getcontext().prec = 4 # устанавливаем точность числа

num = Decimal('3.1415926535')

print(num.quantize(Decimal('1.000'))) # округление до 3 цифр в дробной части

print(num.quantize(Decimal('1.00'))) # округление до 2 цифр в дробной части

print(num.quantize(Decimal('1.0'))) # округление до 1 цифр в дробной части

выводит:

3.142

3.14

3.1

Если точность округления установлена в 22 , а формат округления Decimal('1.00'), то возникнет ошибка.

Приведенный ниже код:

from decimal import \*

getcontext().prec = 2 # устанавливаем точность округления

num = Decimal('3.1415926535')

print(num.quantize(Decimal('1.00'))) # округление до 2 цифр в дробной части

приводит к возникновению ошибки:

decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]

Чтобы избежать ее, необходимо поменять точность округления на 33 и больше.

Помимо первого параметра, метод quantize() принимает в качестве второго параметра стратегию округления:

* ROUND\_CEILING – округление в направлении бесконечности (Infinity);
* ROUND\_FLOOR – округляет в направлении минус бесконечности (- Infinity);
* ROUND\_DOWN – округление в направлении нуля;
* ROUND\_HALF\_EVEN – округление до ближайшего четного числа, число 6.56.5 округлится не до 77, а до 66;
* ROUND\_HALF\_DOWN – округление до ближайшего нуля;
* ROUND\_UP – округление от нуля;
* ROUND\_05UP – округление от нуля (если последняя цифра после округления до нуля была бы 0 или 5, в противном случае к нулю).

Приведенный ниже код:

from decimal import \*

num = Decimal('3.456')

print(num.quantize(Decimal('1.00'), ROUND\_CEILING))

print(num.quantize(Decimal('1.00'), ROUND\_FLOOR))

выводит:

3.46

3.45

### Сравнение float и Decimal чисел

Выбор между типами данных Decimal и float – поиск компромисса в условиях конкретной задачи.

Если нужно считать очень много (симуляции, физика, графика, игры), имеет смысл отказаться от точности Decimal в пользу скорости и компактности хранения данных float. В бизнесе и финансах считать приходится не так много, но делать это нужно предельно точно, тут имеет смысл посмотреть в сторону Decimal.

|  |  |  |
| --- | --- | --- |
| **Характеристика / тип** | **float** | **Decimal** |
| Реализация | аппаратная | программная |
| Размер | 6464 бит | не ограничен |
| Основание экспоненты | 22 | 1010 |
| Скорость | ✔️ | ❌ |
| Настраиваемость | ❌ | ✔️ |
| Для финансов и бизнеса | ❌ | ✔️ |
| Для симуляций, визуализаций и игр | ✔️ | ❌ |
| Для высокоточных вычислений | ❌ | ✔️ |

## Примечания

**Примечание 1.** Decimal числа можно сравнивать между собой, как обычные числа, причем в отличие от float чисел допускается и точное равенство.

from decimal import \*

num = Decimal('0.1')

if num\*3 == Decimal('0.3'):

print('YES')

else:

print('NO')

**Примечание 2.** Можно сортировать списки с Decimal числами и искать минимум и максимум среди них.

Приведенный ниже код:

from decimal import \*

s = '1.34 3.45 1.00 0.03 9.25'

numbers = [Decimal(i) for i in s.split()]

maximum = max(numbers)

minimum = min(numbers)

numbers.sort()

print(maximum)

print(minimum)

print(numbers)

выводит:

9.25

0.03

[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('3.45'), Decimal('9.25')]

**Примечание 3.** Подробнее о типе данных Decimal можно почитать в официальной документации [тут](https://docs.python.org/3.8/library/decimal.html), [тут](https://docs.python.org/3.8/library/decimal.html#decimal-objects) и [тут](https://www.python.org/dev/peps/pep-0327/).

**Примечание 4.** Подробная статья об устройстве float чисел на [хабре](https://habr.com/post/112953/).

**Примечание 5.** О стандарте Decimal чисел от IBM можно почитать [тут](http://speleotrove.com/decimal/decarith.html).

**Примечание 6.** Чтобы не писать каждый раз название типа, можно использовать следующий код:

from decimal import Decimal as D

num1 = D('1.5') + D('3.2')

num2 = D('1.4') \* D('2.58')

print(num1)

print(num2)

# Тема урока: модуль fractions

1. Модуль fractions
2. Тип данных Fraction

**Аннотация.** Урок посвящен модулю fractions и типу данных Fraction.

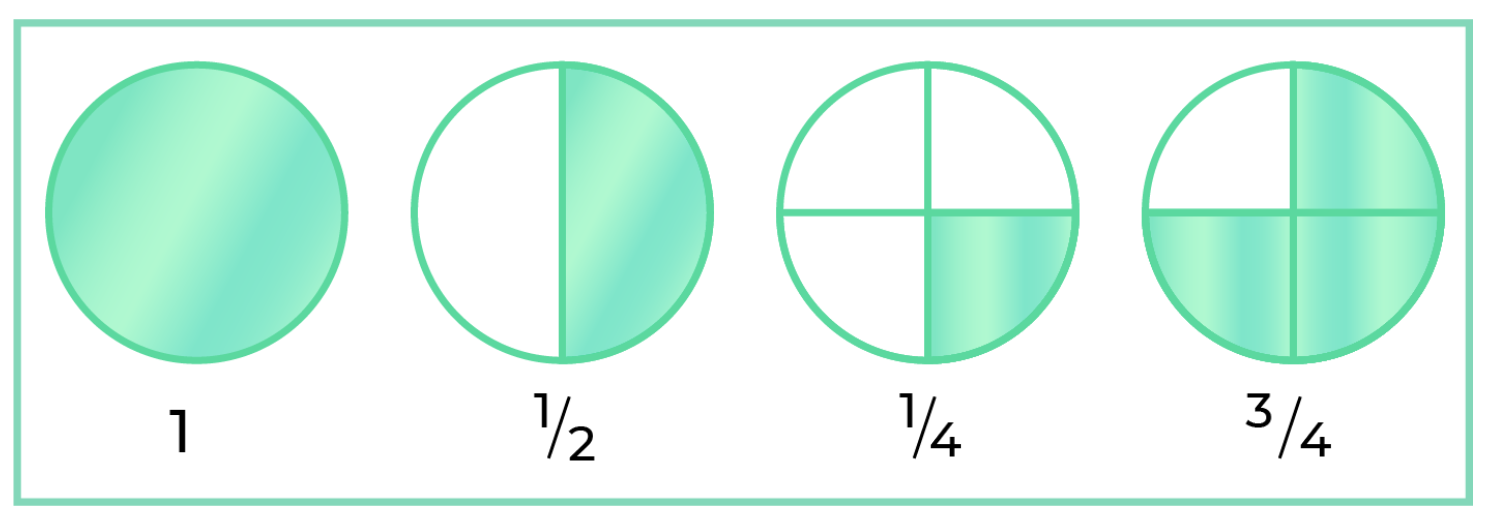
В этом уроке мы изучим числовой тип данных Fraction, который представляет из себя [обыкновенную дробь](https://ru.wikipedia.org/wiki/%D0%94%D1%80%D0%BE%D0%B1%D1%8C_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)#.D0.9E.D0.B1.D1.8B.D0.BA.D0.BD.D0.BE.D0.B2.D0.B5.D0.BD.D0.BD.D1.8B.D0.B5_.D0.B4.D1.80.D0.BE.D0.B1.D0.B8), с заданными числителем и знаменателем;

## Рациональное число

Рациональное число – это число, которое можно представить в виде дроби ��*nm*​, где �,�*m*,*n* соответственно, числитель и знаменатель, которые имеют целочисленное значение, при этом знаменатель не равен нулю.

Например, в дроби 5665​ числитель �=5*m*=5, а знаменатель �=6*n*=6.

   Знаменатель дроби показывает количество равных частей, а числитель дроби показывает, сколько из них взято.



## Тип данных Fraction

Для работы с рациональными числами в Python используется тип данных Fraction. Тип данных Fraction как и Decimal реализован программно, поэтому он в разы медленнее встроенных числовых типов данных int и float. Тип данных Fraction неизменяемый. Операции над данными этого типа приводят к созданию новых объектов, при этом старые не меняются.

Чтобы использовать возможности типа данных Fraction нужно предварительно подключить модуль fractions:

from fractions import Fraction

### Создание Fraction

Создать Fraction число можно несколькими способами:

* из целых чисел, передав значения числителя и знаменателя дроби,
* из строки на основании десятичного представления;
* из строки на основании обыкновенной дроби;
* из числа с плавающей точкой (не рекомендуется).

Приведенный ниже программный код создает Fraction числа на основе целых чисел и строк:

from fractions import Fraction

num1 = Fraction(3, 4) # 3 - числитель, 4 - знаменатель

num2 = Fraction('0.55')

num3 = Fraction('1/9')

print(num1, num2, num3, sep='\n')

и выводит:

3/4

11/20

1/9

Нужно быть очень внимательным при создании Fraction чисел из чисел с плавающей точкой (float), потому что float числа округляются внутри до ближайшего возможного, а Fraction об этом ничего не знает, поэтому копирует содержимое float.

Приведенный ниже программный код создает Fraction число на основе числа с плавающей точкой:

from fractions import Fraction

num = Fraction(0.34)

print(num)

вместо ожидаемого вывода:

17/50

 код выводит:

6124895493223875/18014398509481984

Не рекомендуется создавать Fraction числа из float чисел. В Fraction попадет уже неправильно округленное число. Создавать Fraction числа нужно из целых чисел, либо из строк!

Обратите внимание на то, что при создании рационального числа Fraction, автоматически происходит сокращение числителя и знаменателя дроби.

Приведенный ниже код:

from fractions import Fraction

num1 = Fraction(5, 10)

num2 = Fraction('75/100')

num3 = Fraction('0.25')

print(num1, num2, num3, sep='\n')

выводит:

1/2

3/4

1/4

Так же стоит обратить внимание на вывод дробей, являющихся целыми числами.

Приведенный ниже код:

from fractions import Fraction

num1 = Fraction(5, 1) # 5/1 = 5

num2 = Fraction(23, 23) # 23/23 = 1

print(num1, num2, sep='\n')

выводит:

5

1

### Сравнение Fraction чисел

Fraction числа можно сравнивать между собой точно так же, как и любые другие числа. Доступны 66 основных операторов сравнения:

* >: больше;
* <: меньше;
* >=: больше либо равно;
* <=: меньше либо равно;
* ==:  в точности равно;
* !=: не равно.

Приведенный ниже код:

from fractions import Fraction

num1 = Fraction(1, 2) # 1/2

num2 = Fraction(15, 30) # 15/30=1/2

num3 = Fraction(3, 5) # 3/5

num4 = Fraction(5, 3) # 5/3

num5 = 1

num6 = 0.8

print(num1 == num2)

print(num1 != num4)

print(num2 > num3)

print(num4 <= num1)

print(num1 < num5)

print(num6 > num4)

выводит:

True

True

False

False

True

False

Обратите внимание на то, что мы можем сравнивать Fraction числа и целые числа (числа с плавающей точкой) без явного приведения типов.

### Арифметические операции над Fraction числами

Тип данных Fraction отлично интегрирован в язык Python. С Fraction числами работают все привычные операции: сложение, вычитание, умножение, деление, возведение в степень.

Приведенный ниже код:

from fractions import Fraction

num1 = Fraction('1/10')

num2 = Fraction('2/3')

print(num1 + num2)

print(num1 - num2)

print(num1 \* num2)

print(num1 / num2)

выводит:

23/30

-17/30

1/15

3/20

Мы также можем совершать арифметические операции над Fraction и целыми числами (миксовать Fraction и int), но не рекомендуется смешивать их с float.

Приведенный ниже код:

from fractions import Fraction

num = Fraction('3/8')

print(num + 1)

print(num - 1)

print(num \* 2)

print(num \*\* 4)

выводит:

11/8

-5/8

3/4

81/4096

Обратите внимание, на то, что операция возведения в степень (\*\*) для Fraction чисел может возвращать вещественный результат.

Приведенный ниже код:

from fractions import Fraction

num1 = Fraction('3/8')

num2 = Fraction('1/2')

print(num1 \*\* num2)

выводит:

0.6123724356957945

По сути тут происходит вычисление числа (38)12=38≈0.6123724356957945(83​)21​=83​​≈0.6123724356957945.

### Математические функции

Fraction числа можно передавать как аргументы функций, ожидающих float. Тогда они будут преобразованы во float. К примеру, модуль math оперирующий float числами, может работать и с Fraction числами.

Приведенный ниже код:

from fractions import Fraction

from math import \*

num1 = Fraction('1.44')

num2 = Fraction('0.523')

print(sqrt(num1))

print(sin(num2))

print(log(num1 + num2))

выводит:

1.2

0.4994813555186418

0.6744739152943241

Важно понимать, что результатом работы функций модуля math являются float числа, а не Fraction.

### Свойства numerator и denominator

Для получения числителя и знаменателя Fraction числа используются свойства numerator и denominator.

Приведенный ниже код:

from fractions import Fraction

num = Fraction('5/16')

print('Числитель дроби равен:', num.numerator)

print('Знаменатель дроби равен:', num.denominator)

выводит:

Числитель дроби равен: 5

Знаменатель дроби равен: 16

В Python 3.8 появился метод as\_integer\_ratio(), который возвращает кортеж, состоящий из числителя и знаменателя данного Fraction числа.

Приведенный ниже код:

from fractions import Fraction

num = Fraction('-5/16')

print(num.as\_integer\_ratio())

выводит:

(-5, 16)

### Метод limit\_denominator()

Метод limit\_denominator() возвращает самую близкую к данному числу рациональную дробь, чей знаменатель не превосходит переданного аргумента.

Приведенный ниже код:

from fractions import Fraction

import math

print('PI =', math.pi)

num = Fraction(str(math.pi))

print('No limit =', num)

for d in [1, 5, 50, 90, 100, 500, 1000000]:

limited = num.limit\_denominator(d)

print(limited)

выводит:

PI = 3.141592653589793

No limit = 3141592653589793/1000000000000000

3

16/5

22/7

267/85

311/99

355/113

3126535/995207

Метод limit\_denominator() позволяет получить очень точные рациональные приближения иррациональных чисел, что очень удобно во многих математических задачах.

## Примечания

**Примечание 1.** Для того, чтобы каждый раз не писать название типа, можно использовать следующий код:

from fractions import Fraction as F

num1 = F('1/5') + F('3/2')

num2 = F('1/4') \* F('2/5')

print(num1)

print(num2)

**Примечание 2.** Полное руководство по данному типу данных находится в [официальной документации](https://docs.python.org/3/library/fractions.html).

**Примечание 3.**В Python нельзя совершать арифметические операции (+, -, \*, /) между типами Decimal и Fraction.

Приведенный ниже код:

from decimal import Decimal

from fractions import Fraction

num1 = Decimal('12.5')

num2 = Fraction(19, 3)

print(num1 + num2)

приводит к ошибке:

TypeError: unsupported operand type(s) for +: 'Decimal' and 'Fraction'

## Комплексные числа в Python

В языке Python есть возможность работать с комплексными числами. Общая форма представления комплексного числа следующая: real + imag j, где

* real – вещественная часть комплексного числа;
* imag – мнимая часть комплексного числа, которая завершается символом j или J.

  Обратите внимание: в Python используется буква j, а не i.

Приведенный ниже код:

z1 = 5 + 7j

z2 = 1j

z3 = -3 + 5J

z4 = 1.5 + 3.2j

print(z1, z2, z3, z4, sep='\n')

print(type(z1))

выводит:

(5+7j)

1j

(-3+5j)

(1.5+3.2j)

<class 'complex'>

### Создание комплексных чисел

Комплексные числа можно создать с помощью литерала, как выше, а можно с помощью функции complex(), которая принимает два аргумента: вещественную и мнимую часть числа, либо строковое представление числа.

Приведенный ниже код:

z1 = -3 + 2j # создание на основе литерала

z2 = complex(6, -8) # z2 = 6 - 8j

z3 = complex(0, 2.5) # z3 = 2.5j

z4 = complex(5, 0) # z4 = 5 + 0j

z5 = complex('3+4j') # создание на основе строки

print(z1, z2, z3, z4, z5, sep='\n')

выводит:

(-3+2j)

(6-8j)

2.5j

(5+0j)

(3+4j)

### Арифметические операции над комплексными числами

Тип данных complex отлично интегрирован в язык Python. С complex числами работают все привычные операции: сложение, вычитание, умножение, деление, возведение в степень.

Приведенный ниже код:

z1 = 1 + 3j

z2 = -3 + 2j

print('z1 + z2 =', z1 + z2)

print('z1 - z2 =', z1 - z2)

print('z1 \* z2 =', z1 \* z2)

print('z1 / z2 =', z1 / z2)

print('z1^20 =', z1\*\*20)

выводит:

z1 + z2 = (-2+5j)

z1 - z2 = (4+1j)

z1 \* z2 = (-9-7j)

z1 / z2 = (0.23076923076923078-0.8461538461538461j)

z1^20 = (9884965888-1512431616j)

Мы также можем совершать арифметические операции над complex и целыми числами (миксовать complex, int, float).

Приведенный ниже код:

z = 1 + 3j

print(z + 5)

print(z - 2)

print(3\*z)

print(z/2)

выводит:

(6+3j)

(-1+3j)

(3+9j)

(0.5+1.5j)

### Методы и свойства комплексных чисел

Для получения действительной и мнимой частей комплексного числа используются свойства real и imag.

Приведенный ниже код:

z = 3+4j

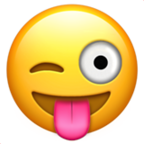
print('Действительная часть =', z.real)

print('Мнимая часть =', z.imag)

выводит:

Действительная часть = 3.0

Мнимая часть = 4.0

Python представляет комплексное число как два вещественных числа, поэтому при выводе у нас появились выражения 3.03.0 и 4.04.0 вместо 33 и 44.

Для нахождения сопряженного комплексного числа, можно использовать метод conjugate().

Приведенный ниже код:

z = 3+4j

print('Сопряженное число =', z.conjugate())

выводит:

Сопряженное число = (3-4j)

## Примечания

**Примечание 1.** Почему используется буква j вместо буквы i, можно почитать [тут](https://stackoverflow.com/questions/24812444/why-are-complex-numbers-in-python-denoted-with-j-instead-of-i).

**Примечание 2.** Для нахождения модуля комплексного числа, используется встроенная функция abs().

Приведенный ниже код:

z = 3+4j

print('Модуль числа =', abs(z))

выводит:

Модуль числа = 5.0

**Примечание 3.**Встроенный модуль math работает с вещественными числами. Для работы с комплексными числами есть модуль cmath. Модуль cmath включает дополнительные функции для использования комплексных чисел.

import cmath

z = 2+3j

print(cmath.phase(z)) # полярный угол

print(cmath.polar(z)) # полярные координаты

Модуль cmath содержит следующие категории функций:

* Экспоненциальные и логарифмические функции
* Квадратные корни
* Тригонометрические функции и их обратные
* Гиперболические функции и их обратные

**Примечание 4.** Документация по модулю cmath [тут](https://docs.python.org/3/library/cmath.html).

**Примечание 5.** Больше примеров работы с комплексными числами [тут](https://www.askpython.com/python/python-complex-numbers).

**Примечание 6.** Для работы с комплексными числами (тип complex) не нужно подключать какой-либо модуль, в отличии от типа Decimal и Fraction.

# Тема урока: необязательные и именованные аргументы

1. Позиционные аргументы
2. Необязательные аргументы
3. Именованные аргументы

**Аннотация.** Урок посвящен необязательным и именованным аргументам.

## Позиционные аргументы

Все ранее написанные нами функции имели **позиционные аргументы**. Такие аргументы передаются без указания имен. Они называются **позиционными**, потому что именно по позиции, расположению аргумента, функция понимает, какому параметру он соответствует.

Рассмотрим следующий код:

def diff(x, y):

return x - y

res = diff(10, 3) # используем позиционные аргументы

print(res)

Такой код выведет число 77. При вызове функции diff() **первому** параметру x будет соответствовать **первый** переданный аргумент, 1010, а **второму** параметру y  — **второй** аргумент, 33.

В Python можно использовать не только позиционные, но и именованные аргументы.

## Именованные аргументы

Аргументы, передаваемые с именами, называются **именованными**. При вызове функции можно использовать имена параметров из ее определения. Исключение составляют списки аргументов неопределенной длины, где используются аргументы со звездочкой, но об этом в следующем уроке. Все функции из предыдущих уроков можно вызывать, передавая им именованные аргументы.

Рассмотрим следующий код:

def diff(x, y):

return x - y

res = diff(x=10, y=3) # используем именованные аргументы

print(res)

Такой код по-прежнему выведет число 77. При вызове функции diff() мы явно указываем, что параметру x соответствует аргумент 1010, а параметру y— аргумент 33.

Использование именованных аргументов позволяет нарушать их позиционный порядок при вызове функции. Порядок упоминания именованных аргументов не имеет значения!

Мы можем вызвать функцию diff() так:

res = diff(y=3, x=10)

и получить тот же результат 77.

Возможность использования именованных аргументов — еще один повод давать параметрам значащие, а не однобуквенные имена.

### Когда стоит применять именованные аргументы

Каких-то строгих правил на этот счёт не существует. Однако широко практикуется такой подход: если функция принимает больше трёх аргументов, нужно хотя бы часть из них указать по имени. Особенно важно именовать значения аргументов, когда они относятся к одному типу, ведь без имен очень трудно понять, что делает функция с подобным вызовом.

Рассмотрим определение функции make\_circle() для рисования круга:

def make\_circle(x, y, radius, line\_width, fill):

# тело функции

Вызвать такую функцию можно так:

make\_circle(200, 300, 17, 2.5, True)

Тут непросто понять, какие параметры круга задают числа 200200, 300300 или 1717.

Сравните:

make\_circle(x=200, y=300, radius=17, line\_width=2.5, fill=True)

Такой код читать значительно проще!

В соответствии с PEP 8 при указании значений именованных аргументов при вызове функции знак равенства не окружается пробелами.

Когда значение аргументов очевидно, можно их не именовать. Да, очевидность относительна, но обычно легко понять, что скрывается за значениями при вызове функции point3d(7, 50, 13) или rgb(7, 255, 45). В первом случае переданные аргументы – координаты точки в трехмерном пространстве, во втором – значения компонент **red**, **green**, **blue** некоторого цвета. Тут можно ориентироваться только на здравый смысл, жестких правил нет.

Мы уже сталкивались с именованными аргументами, когда вызывали функцию print().

Приведенный ниже код:

print('aaaa', 'bbbbb', sep='\*', end='##')

print('cccc', 'dddd', sep='()')

print('eeee', 'ffff', sep='123', end='python')

использует именованные аргументы sep и end и выводит:

aaaa\*bbbbb##cccc()dddd

eeee123ffffpython

Используя именованные аргументы sep и end можно не беспокоиться, какой из них указать первым.

   Напомним, что значение по умолчанию для sep=' ' (символ пробела),  а для end='\n' (символ перевода строки).

### Комбинирование позиционных и именованных аргументов

Мы можем вызывать функции, используя именованные и позиционные аргументы одновременно. Но позиционные значения должны быть указаны **до** любых именованных!

Для функции diff() код:

res = diff(10, y=3) # используем позиционный и именованный аргумент

работает как полагается, при этом параметру x соответствует значение 1010.

Приведенный ниже код:

res = diff(x=10, 3) # используем позиционный и именованный аргумент

приводит к возникновению ошибки SyntaxError: positional argument follows keyword argument.

## Необязательные аргументы

Бывает, что какой-то параметр функции часто принимает одно и то же значение. Например, для функции print() создатели языка Python установили значения параметров sep и end равными символу пробела и символу перевода строки, поскольку эти значения используют наиболее часто.

Другим примером служит функция int() , преобразующая строку в число. Она принимает два аргумента:

* первый аргумент: строка, которую нужно преобразовать в число;
* второй аргумент: основание [системы счисления](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F).

Это позволяет ей считывать числа в различных системах счисления.

Приведенный ниже код, преобразует двоичное число 101101:

num = int('101', 2) # аргумент 2 указывает на то, что число 101 записано в двоичной системе

В переменной num будет храниться число 55, так как 1012=5101012​=510​.

Но чаще всего эта функция используется для считывания из строки чисел, записанных в десятичной системе счисления. Утомительно каждый раз писать 1010 вторым аргументом. В таких ситуациях Python позволяет задавать некоторым параметрам значения по умолчанию. У функции int() второй параметр по умолчанию равен 1010, и потому можно вызывать эту функцию с одним аргументом. Значение второго подставится автоматически.

Чтобы задать **значение параметра по умолчанию**, в списке параметров функции достаточно после имени переменной написать знак равенства и нужное значение.

Параметры со значением по умолчанию идут последними, ведь иначе интерпретатор не смог бы понять, какой из аргументов указан, а какой пропущен, и для него нужно использовать значение по умолчанию.

Рассмотрим все то же определение функции make\_circle(), которая рисует круг:

def make\_circle(x, y, radius, line\_width, fill):

# тело функции

Поскольку обычно нам нужно рисовать круг с шириной линии, равной 11 с заливкой, то логично установить данные значения в качестве значений по умолчанию:

def make\_circle(x, y, radius, line\_width=1, fill=True):

# тело функции

Теперь для того, чтобы нарисовать стандартный круг, то есть круг имеющий ширину линии, равную 11 с заливкой, мы вызываем функцию так:

make\_circle(100, 50, 20)

или так:

make\_circle(x=100, y=50, radius=20)

Если вам хочется поменять ширину линии и заливку, то вы легко можете это сделать:

make\_circle(x=100, y=50, radius=20, fill=False) # line\_width=1, fill=False

make\_circle(x=100, y=50, radius=20, line\_width=3) # fill=True, line\_width=3

make\_circle(x=100, y=50, radius=20, line\_width=5, fill=False) # line\_width=5, fill=False

   В соответствии с стандартом PEP 8 и при объявлении аргументов по умолчанию пробел вокруг знака равенства не ставят.

### Изменяемые типы в качестве значений по умолчанию

При использовании **изменяемых типов данных** в качестве значения параметра по умолчанию можно столкнуться с неожиданными результатами работы функции.

Рассмотрим определение функции append(), где в качестве значения по умолчанию используется изменяемый тип данных (список, тип list):

def append(element, seq=[]):

seq.append(element)

return seq

Вызывая функцию append() следующим образом:

print(append(10, [1, 2, 3]))

print(append(5, [1]))

print(append(1, []))

print(append(3, [4, 5]))

получим ожидаемый вывод:

[1, 2, 3, 10]

[1, 5]

[1]

[4, 5, 3]

А если вызовем функцию append() так:

print(append(10))

print(append(5))

print(append(1))

 получим **не совсем** ожидаемый вывод:

[10]

[10, 5]

[10, 5, 1]

Что происходит? Значение по умолчанию для параметра создается единожды при определении функции (обычно при загрузке модуля) и становится атрибутом (свойством) функции. Поэтому, если значение по умолчанию изменяемый объект, то его изменение повлияет на каждый следующий вызов функции.

Чтобы посмотреть значения по умолчанию, можно использовать атрибут \_\_defaults\_\_.

Приведенный ниже код:

def append(element, seq=[]):

seq.append(element)

return seq

print('Значение по умолчанию', append.\_\_defaults\_\_)

выводит:

Значение по умолчанию ([],)

Приведенный ниже код:

def append(element, seq=[]):

seq.append(element)

return seq

print('Значение по умолчанию', append.\_\_defaults\_\_)

print(append(10))

print('Значение по умолчанию', append.\_\_defaults\_\_)

print(append(5))

print('Значение по умолчанию', append.\_\_defaults\_\_)

print(append(1))

print('Значение по умолчанию', append.\_\_defaults\_\_)

выводит:

Значение по умолчанию ([],)

[10]

Значение по умолчанию ([10],)

[10, 5]

Значение по умолчанию ([10, 5],)

[10, 5, 1]

Значение по умолчанию ([10, 5, 1],)

Для решения проблемы можно использовать константу None в качестве значения параметра по умолчанию, а в теле функции устанавливать нужное значение:

def append(element, seq=None):

if seq is None:

seq = []

seq.append(element)

return seq

Вызывая функцию append() следующим образом:

print(append(10))

print(append(5))

print(append(1))

 получим **ожидаемый вывод**:

[10]

[5]

[1]

   Подход, основанный на значении None, общепринятый в Python.

## Примечания

**Примечание 1.** При написании функций стоит указывать более важные параметры первыми.

**Примечание 2.** Именованные аргументы часто используют вместе со значениями по умолчанию.

**Примечание 3.** Именованные и позиционные аргументы не всегда хорошо ладят друг с другом. При вызове функции позиционные аргументы должны обязательно идти перед именованными аргументами.

**Примечание 4.** Параметр в программировании — принятый функцией аргумент. Термин «аргумент» подразумевает, что конкретно и какой конкретной функции было передано, а параметр — в каком качестве функция применила это принятое. То есть вызывающий код передает аргумент в параметр, который определен в описании (заголовке) функции.

* **P**arameter → **P**laceholder (заполнитель принадлежит имени функции и используется в теле функции);
* **A**rgument → **A**ctual value (фактическое значение, которое передается при вызове функции).

Подробнее можно почитать [тут](https://stackoverflow.com/questions/156767/whats-the-difference-between-an-argument-and-a-parameter).

**Примечание 5.** Отличная [статья](https://habr.com/ru/company/ruvds/blog/515678/) на хабре про аргументы функций в Python.

# Тема урока: функции с переменным количеством аргументов

1. Аргументы \*args
2. Аргументы \*\*kwargs
3. Keyword-only аргументы

**Аннотация.** Урок посвящен функциям с переменным количеством аргументов.

## Переменное количество аргументов

Вспомним функцию print(), которой мы пользуемся почти в каждой задаче.

Приведенный ниже код:

print('a')

print('a', 'b')

print('a', 'b', 'c')

print('a', 'b', 'c', 'd')

выводит:

a

a b

a b c

a b c d

Функция print() принимает столько аргументов, сколько ей передано. Причем функция print() делает полезную работу, даже когда вызывается вообще без аргументов. В этом случае она переносит каретку печати на новую строку.

Было бы здорово научиться писать свои собственные функции, способные принимать переменное количество аргументов. Для этого потребуется специальный, совсем не сложный во всех смыслах, синтаксис.

Рассмотрим определение функции my\_func():

def my\_func(\*args):

print(type(args))

print(args)

my\_func()

my\_func(1, 2, 3)

my\_func('a', 'b')

Приведенный выше код выводит:

<class 'tuple'>

()

<class 'tuple'>

(1, 2, 3)

<class 'tuple'>

('a', 'b')

В заголовке функции my\_func() указан всего один параметр args, но со звездочкой перед ним. Звездочка в определении функции означает, что переменная (параметр) args получит в виде кортежа все аргументы, переданные в функцию при ее вызове от текущей позиции и до конца.

При описании функции можно использовать **только один** параметр помеченный звездочкой, причем располагаться он должен в конце списка параметров, иначе последующим параметрам не достанется значений.

Приведенный ниже код:

def my\_func(\*args, num):

print(args)

print(num)

не является рабочим, так как параметр со звездочкой указан раньше позиционного num.

Приведенный ниже код:

def my\_func(num, \*args):

print(args)

print(num)

my\_func(17, 'Python', 2, 'C#')

связывает с переменной num значение 1717, а с переменной args значение кортежа ('Python', 2, 'C#') и выводит:

('Python', 2, 'C#')

17

Помеченный звездочкой параметр \*args нормально переживает и отсутствие аргументов, в то время как позиционные параметры всегда обязательны.

Приведенный ниже код:

my\_func(17)

связывает с переменной num значение 1717, а с переменной args значение пустого кортежа () и выводит:

()

17

Обратите внимание: функция my\_func() принимает несколько аргументов, но как минимум один аргумент должен быть передан обязательно. Этот первый аргумент станет значением переменной num, а остальные аргументы сохранятся в переменной args. Подобным образом можно делать любое количество обязательных аргументов.

Параметр args в определении функции пишется после позиционных параметров перед первым параметром со значением по умолчанию.

### Передача аргументов в форме списка и кортежа

Иногда хочется сначала сформировать набор аргументов, а потом передать их функции. Тут поможет оператор распаковки коллекций, который также обозначается звездочкой \*.

Вспомним, что встроенная функция sum() принимает на вход коллекцию чисел (список, кортеж, и т.д).

Приведенный ниже код:

sum1 = sum([1, 2, 3, 4]) # считаем сумму чисел в списке

sum2 = sum((10, 20, 30, 40)) # считаем сумму чисел в кортеже

print(sum1, sum2)

выводит:

10 100

Однако функция sum() не может принимать переменное количество аргументов.

Приведенный ниже код:

sum1 = sum(1, 2, 3, 4)

print(sum1)

приводит к возникновению ошибки:

TypeError: sum expected at most 2 arguments, got 4

Напишем свою версию функции sum(), функцию my\_sum(), которая принимает переменное количество аргументов и вычисляет их сумму:

def my\_sum(\*args):

return sum(args) # args - это кортеж

Приведенный ниже код:

print(my\_sum())

print(my\_sum(1))

print(my\_sum(1, 2))

print(my\_sum(1, 2, 3))

print(my\_sum(1, 2, 3, 4))

выводит:

0

1

3

6

10

Мы также можем вызывать нашу функцию my\_sum() , передавая ей списки или кортежи, предварительно распаковав их.

Приведенный ниже код:

print(my\_sum(\*[1, 2, 3, 4, 5])) # распаковка списка

print(my\_sum(\*(1, 2, 3))) # распаковка кортежа

выводит:

15

6

Более того, часть аргументов можно передавать непосредственно и даже коллекции подставлять не только по одной.

Приведенный ниже код:

print(my\_sum(1, 2, \*[3, 4, 5], \*(7, 8, 9), 10))

выводит:

49

По соглашению, параметр, получающий подобный кортеж значений, принято называть args (от слова arguments). Старайтесь придерживаться этого соглашения.

## Получение именованных аргументов в виде словаря

Позиционные аргументы можно получать в виде \*args, причём произвольное их количество. Такая возможность существует и для именованных аргументов. Только именованные аргументы получаются в виде словаря, что позволяет сохранить имена аргументов в ключах.

Рассмотрим определение функции my\_func():

def my\_func(\*\*kwargs):

print(type(kwargs))

print(kwargs)

Приведенный ниже код:

my\_func()

my\_func(a=1, b=2)

my\_func(name='Timur', job='Teacher')

выводит:

<class 'dict'>

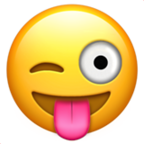
{}

<class 'dict'>

{'a': 1, 'b': 2}

<class 'dict'>

{'name': 'Timur', 'job': 'Teacher'}

По соглашению параметр, получающий подобный словарь, принято называть kwargs (от словосочетания keyword arguments). Старайтесь придерживаться этого соглашения.

Параметр  \*\*kwargs пишется в самом конце, после последнего аргумента со значением по умолчанию. При этом функция может содержать и \*args и \*\*kwargs параметры.

Рассмотрим определение функции, которая принимает все виды аргументов.

def my\_func(a, b, \*args, name='Gvido', age=17, \*\*kwargs):

print(a, b)

print(args)

print(name, age)

print(kwargs)

Приведенный ниже код:

my\_func(1, 2, 3, 4, name='Timur', age=28, job='Teacher', language='Python')

my\_func(1, 2, name='Timur', age=28, job='Teacher', language='Python')

my\_func(1, 2, 3, 4, job='Teacher', language='Python')

выводит (пустая строка вставлена для наглядности):

1 2

(3, 4)

Timur 28

{'job': 'Teacher', 'language': 'Python'}

1 2

()

Timur 28

{'job': 'Teacher', 'language': 'Python'}

1 2

(3, 4)

Gvido 17

{'job': 'Teacher', 'language': 'Python'}

Не нужно пугаться, в реальном коде функции редко используют все эти возможности одновременно. Но понимать, как каждая отдельная форма объявления аргументов работает, и как такие формы сочетать — очень важно.

Для лучшего понимания, поэкспериментируйте с передачей аргументов. Правила использования аргументов довольно сложно описывать, но на практике мы редко сталкиваемся с проблемами.

### Передача именованных аргументов в форме словаря

Как и в случае позиционных аргументов, именованные можно передавать в функцию "пачкой" в виде словаря. Для этого нужно перед словарём поставить две звёздочки.

Рассмотрим определение функции my\_func():

def my\_func(\*\*kwargs):

print(type(kwargs))

print(kwargs)

Приведенный ниже код:

info = {'name':'Timur', 'age':'28', 'job':'teacher'}

my\_func(\*\*info)

выводит:

<class 'dict'>

{'name': 'Timur', 'age': '28', 'job': 'teacher'}

Рассмотрим еще один пример определения функции print\_info(), распечатывающей информацию о пользователе.

def print\_info(name, surname, age, city, \*children, \*\*additional\_info):

print('Имя:', name)

print('Фамилия:', surname)

print('Возраст:', age)

print('Город проживания:', city)

if len(children) > 0:

print('Дети:', ', '.join(children))

if len(additional\_info) > 0:

print(additional\_info)

Приведенный ниже код:

children = ['Бодхи Рансом Грин', 'Ноа Шэннон Грин', 'Джорни Ривер Грин']

additional\_info = {'height':163, 'job':'actress'}

print\_info('Меган', 'Фокс', 34, 'Ок-Ридж', \*children, \*\*additional\_info)

выводит:

Имя: Меган

Фамилия: Фокс

Возраст: 34

Город проживания: Ок-Ридж

Дети: Бодхи Рансом Грин, Ноа Шэннон Грин, Джорни Ривер Грин

{'height': 163, 'job': 'actress'}

При подстановке аргументов "разворачивающиеся" наборы аргументов вроде \*positional и \*\*named можно указывать вперемешку с аргументами соответствующего типа: \*positional с позиционными, а \*\*named — с именованными. И, конечно, же, все именованные аргументы должны идти после всех позиционных!

## Keyword-only аргументы

В Python 3 добавили возможность пометить именованные аргументы функции так, чтобы вызвать функцию можно было, только передав эти аргументы по именам. Такие аргументы называются keyword-only и их нельзя передать в функцию в виде позиционных.

Рассмотрим определение функции make\_circle():

def make\_circle(x, y, radius, \*, line\_width=1, fill=True):

Здесь \* выступает разделителем: отделяет обычные аргументы (их можно указывать по имени и позиционно) от строго именованных.

Приведенный ниже код работает как и полагается:

make\_circle(10, 20, 5) # x=10, y=20, radius=5, line\_width=1, fill=True

make\_circle(x=10, y=20, radius=7) # x=10, y=20, radius=7, line\_width=1, fill=True

make\_circle(10, 20, radius=10, line\_width=2, fill=False) # x=10, y=20, radius=10, line\_width=2, fill=False

make\_circle(x=10, y=20, radius=17, line\_width=3) # x=10, y=20, radius=17, line\_width=3, fill=True

То есть аргументы x, y и radius могут быть переданы в качестве как позиционных, так и именованных аргументов. При этом аргументы line\_width и fill могут быть переданы **только** как именованные аргументы.

Приведенный ниже код:

make\_circle(10, 20, 15, 20)

make\_circle(x=10, y=20, 15, True)

make\_circle(10, 20, 10, 2, False)

приводит к возникновению ошибок.

Этот пример неплохо демонстрирует подход к описанию аргументов. Первые три аргумента — координаты центра круга и радиус. Координаты центра и радиус присутствуют у круга всегда, поэтому обязательны и их можно не именовать. А вот line\_width и fill — необязательные аргументы, ещё и получающие ничего не говорящие значения. Вполне логично ожидать, что вызов вида make\_circle(10, 20, 5, 3, False) мало кому понравится! Ради ясности аргументы line\_width и fill  и объявлены так, что могут быть указаны только явно через имя.

Мы также можем объявить функцию, у которой будут только строго именованные аргументы, для этого нужно поставить звёздочку в самом начале перечня аргументов.

def make\_circle(\*, x, y, radius, line\_width=1, fill=True):

Теперь для вызова функции make\_circle() нам нужно передать значения всех аргументов явно через их имя:

make\_circle(x=10, y=20, radius=15) # line\_width=1, fill=True

make\_circle(x=10, y=20, radius=15, line\_width=4, fill=False)

Такой разделитель можно использовать только один раз в определении функции. Его нельзя применять в функциях с неограниченным количеством позиционных аргументов \*args.

## Примечания.

**Примечание 1.** Специальный синтаксис \*args и \*\*kwargs в определении функции позволяет передавать функции переменное количество позиционных и именованных аргументов. При этом args и kwargs **просто имена.**Вы не обязаны их использовать, можно выбрать любые, однако среди Python программистов приняты именно эти.

**Примечание 2.** Вы можете использовать одновременно \*args и \*\*kwargs в одной строке для вызова функции. В этом случае **порядок имеет значение**. Как и аргументы, не являющиеся аргументами по умолчанию, \*args должны предшествовать и аргументам по умолчанию, и \*\*kwargs. Правильный порядок параметров:

1. позиционные аргументы,
2. \*args аргументы,
3. \*\*kwargs аргументы.

def my\_func(a, b, \*args, \*\*kwargs):

# Тема урока: парадигмы программирования

1. Парадигмы программирования
2. Императивное программирование
3. Структурное программирование
4. Объектно-ориентированное программирование
5. Логическое программирование
6. Функциональное программирование

**Аннотация.** Урок посвящен основным парадигмам программирования.

## Парадигмы программирования

Парадигма программирования (подход к программированию) — совокупность идей и понятий, определяющих стиль написания компьютерных программ.

   Парадигма – устоявшаяся система научных взглядов, в рамках которой ведутся исследования (Т. Кун).

Парадигма программирования определяется:

* вычислительной моделью;
* базовой программной единицей (-ами);
* методами разделения абстракций.

Язык программирования не обязательно использует единственную парадигму. Существуют мультипарадигменные языки. Создатели таких языков считают, что ни одна парадигма не может быть одинаково эффективной для всех задач, и следует позволять программисту выбирать лучшую для решения каждой.

   Язык программирования Python мультипарадигменный.

Основные парадигмы программирования:

* императивное;
* структурное;
* объектно-ориентированное;
* функциональное;
* логическое.

Вычислительная техника создавалась для решения математических задач — расчета баллистических траекторий, численного решения уравнений и т.д. Для этого же предназначены первые языки программирования, такие как Fortran, Алгол, реализованные в парадигме **императивного программирования**.

### Императивное программирование

Императивное программирование (ИП) характеризуется тем, что:

* в исходном коде программы записаны инструкции (команды);
* инструкции должны выполняться последовательно;
* данные, полученные при выполнении инструкции, могут записываться в память;
* данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями.

Императивная программа похожа на приказы (англ. imperative — приказ, повелительное наклонение), выражаемые повелительным наклонением в естественных языках. Это последовательность команд, выполняемых процессором.

При императивном подходе к составлению кода широко используется присваивание. Наличие операторов присваивания увеличивает сложность модели вычислений и создает условия для специфических ошибок императивных программ.

Основные механизмы управления:

* последовательное исполнение команд;
* использование именованных переменных;
* использование оператора присваивания;
* использование ветвления (оператор if);
* использование безусловного перехода (оператор goto 😈).

Ключевой идеей императивного программирования является работа с переменными, как с временным хранением данных в оперативной памяти.

### Структурное программирование

Струк­тур­ная парадигма программирования на­це­ле­на на со­кра­ще­ние вре­ме­ни раз­ра­бот­ки и уп­ро­ще­ние под­держ­ки про­грамм за счёт ис­поль­зо­ва­ния блоч­ных опе­ра­то­ров и под­про­грамм. От­ли­чительная чер­та струк­тур­ных про­грамм — от­каз от опе­ра­то­ра безусловного пе­ре­хо­да (goto 🙏), который широко использовался в 19701970-х годах.

Основные механизмы управления:

* последовательное исполнение команд;
* использование именованных переменных;
* использование оператора присваивания;
* использование ветвления (оператор if);
* использование циклов;
* использование подпрограмм (функций).

В структурном программировании программа по возможности разбивается на маленькие подпрограммы (функции) с изолированным контекстом.

   Парадигму структурного программирования предложил нидерландский ученый Эдсгер Дейкстра.

### Объектно-ориентированное программирование

В объектно-ориентированной парадигме про­грам­ма раз­би­ва­ет­ся на объ­ек­ты – струк­ту­ры дан­ных, со­стоя­щие из по­лей, опи­сы­ваю­щих со­стоя­ние, и ме­то­дов – функций, при­ме­няе­мых к объ­ек­там для изме­не­ния или за­про­са их со­стоя­ния.

Объ­ект­но-ори­ен­ти­ро­ван­ную парадигму программирования под­дер­жи­ваю­т:

* Python;
* C#;
* Java;
* C++;
* JavaScript;
* и другие.

Основные механизмы управления:

* абстракция;
* класс;
* объект;
* полиморфизм;
* инкапсуляция;
* наследование.

### Логическое программирование

При использовании логического программирования программа содержит описание проблемы в терминах фактов и логических формул, а решение проблемы система находит с помощью механизмов логического вывода.

В конце 6060-х годов XX века Корделл Грин предложил использовать резолюцию как основу логического программирования. Алан Колмеро создал язык логического программирования [Prolog](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%BB%D0%BE%D0%B3_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) в 19711971 году. Логическое программирование пережило пик популярности в середине 8080-х годов XX века, когда было положено в основу проекта разработки программного и аппаратного обеспечения вычислительных систем пятого поколения.

Важное его преимущество — достаточно высокий уровень машинной независимости, а также возможность откатов, возвращения к предыдущей подцели при отрицательном результате анализа одного из вариантов в процессе поиска решения.

Один из концептуальных недостатков логического подхода — специфичность класса решаемых задач.

Недостаток практического характера — сложность эффективной реализации для принятия решений в реальном времени, скажем, для систем жизнеобеспечения.

### Функциональное программирование

Основной инструмент **функционального программирования** (ФП) — математические функции.

Математические функции выражают связь между исходными данными и итогом процесса. Процесс вычисления также имеет вход и выход, поэтому функция — вполне подходящее и адекватное средство описания вычислений. Именно этот простой принцип положен в основу функциональной парадигмы программирования.

   Функциональное программирование (ФП) — [декларативная](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) парадигма программирования.

Функциональная программа — набор определений функций. Функции определяются через другие функции или рекурсивно через самих себя. При выполнении программы функции получают аргументы, вычисляют и возвращают результат, при необходимости вычисляя значения других функций.

Как преимущества, так и недостатки данной парадигмы определяет модель вычислений без состояний. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты, то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путем присваивания значений переменным, в функциональных достигается передачей выражений в параметры функций. В результате чисто функциональная программа не может изменять имеющиеся данные, а может лишь порождать новые копированием и/или расширением старых. Следствие того же — отказ от циклов в пользу [рекурсии](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F).

Сильные стороны функционального программирования:

* повышение надёжности кода;
* удобство организации модульного тестирования;
* возможности оптимизации при компиляции;
* возможности параллелизма.

Недостатки: отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный сборщик мусора.

Основные идеи функционального программирования:

* **неизменяемые переменные** — в функциональном программировании можно определить переменную, но изменить ее значение нельзя;
* **чистая функция** — это функция, результат работы которой предсказуем. При вызове с одними и теми же аргументами, такая функция всегда вернет одно и то же значение. Про такие функции говорят, что они не вызывают побочных эффектов;
* **функции высшего порядка** — могут принимать другие функции в качестве аргумента или возвращать их;
* **рекурсия** — поддерживается многими языками программирования, а для функционального программирования обязательна. Дело в том, что в языках ФП отсутствуют циклы, поэтому для повторения операций служит рекурсия. Использование рекурсии в языках ФП оптимизировано, и происходит быстрее, чем в языках императивного программирования;
* **лямбда-выражения** — способ определения анонимных функциональных объектов.

## Примечания

**Примечание 1.** Термин «парадигма программирования» впервые применил в 1978 году Роберт Флойд. В своей лекции при получении премии Тьюринга он отметил, что в программировании можно наблюдать явление, подобное парадигмам Куна, но, в отличие от них, парадигмы программирования не взаимоисключающие: если прогресс искусства программирования в целом требует постоянного изобретения и усовершенствования парадигм, то совершенствование искусства программиста требует расширения репертуара парадигм.

Таким образом, по мнению Роберта Флойда, в отличие от парадигм в научном мире, описанных Куном, парадигмы программирования могут сочетаться, обогащая инструментарий программиста.

**Примечание 2.** В основе императивных, структурных, объектно-ориентированных языков программирования лежит [машина Тьюринга](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%B0_%D0%A2%D1%8C%D1%8E%D1%80%D0%B8%D0%BD%D0%B3%D0%B0), разработанная [Аланом Тьюрингом](https://ru.wikipedia.org/wiki/%D0%A2%D1%8C%D1%8E%D1%80%D0%B8%D0%BD%D0%B3,_%D0%90%D0%BB%D0%B0%D0%BD).

**Примечание 3.** В основе функциональных языков программирования лежит модель лямбда-исчислений, разработанная [Алонзо Чёрчем](https://ru.wikipedia.org/wiki/%D0%A7%D1%91%D1%80%D1%87,_%D0%90%D0%BB%D0%BE%D0%BD%D0%B7%D0%BE).

# Тема урока: функции как объекты

1. Функции как объекты
2. Функции в качестве аргументов других функций
3. Встроенные функции, принимающие функции в качестве аргументов
4. Функции в качестве возвращаемых значений других функций

**Аннотация.** Урок посвящен основным концепциям функционального программирования.

### Функции как объекты

До сих пор мы рассматривали функции как совершенно отдельный элемент языка со своим синтаксисом и механизмом работы. Но, оказывается, функции также что-то вроде особого типа объектов. Бывают числа, строки, списки, кортежи, словари, множества. А бывают — функции. У каждого из этих типов есть свои операции, свой синтаксис, но все они — объекты.

Напомним, что язык Python мультипарадигменный, он одинаково хорошо поддерживает и функциональную и объектно-ориентированную парадигмы программирования.

Приведенный ниже код:

num = 17

numbers = [1, 2, 3]

colors = (1, 2, 3)

name = 'Python'

print(type(num))

print(type(numbers))

print(type(colors))

print(type(name))

выводит:

<class 'int'>

<class 'list'>

<class 'tuple'>

<class 'str'>

Таким образом, целое число — объект типа int, список — объект типа list, кортеж — объект типа tuple, строка — объект типа str и т.д.

Любая функция в языке Python — объект типа function.

Приведенный ниже код:

print(type(print))

print(type(sum))

print(type(abs))

выводит:

<class 'builtin\_function\_or\_method'>

<class 'builtin\_function\_or\_method'>

<class 'builtin\_function\_or\_method'>

Python выдает нам строку <class 'builtin\_function\_or\_method'>, которая поясняет, что print, sum, abs — встроенные функции языка, или методы.

Обратите внимание: скобки не ставим при передаче аргумента в функцию type(), мы не вызываем функцию, а передаем ее название.

Мы также можем объявить свою функцию, которая тоже является объектом.

Приведенный ниже код:

def hello():

print('Hello from function')

print(type(hello))

выводит:

<class 'function'>

​   ​​​​​​В языке Python все объект: число, строка, список, кортеж, множество, словарь, даже функция.

Поскольку функции тоже объекты, работать с ними можно и как с объектами: записывать их в переменные, передавать в качестве аргументов другим функциям, возвращать из функций и т.д.

Приведенный ниже код:

def hello():

print('Hello from function')

func = hello # присваиваем переменной func функцию hello

func() # вызываем функцию

выводит:

Hello from function

Таким образом, теперь переменную func можно использовать как функцию hello().

Рассмотрим еще один пример. Если по какой-либо причине вам не нравится название встроенной функции print(), которая выводит указанный текст на экран, можно написать такой код:

writeln = print # как в языке Pascal 😀

writeln('Hello world!')

writeln('Python')

Такой код выводит:

Hello world!

Python

Возможность записать функцию в переменную позволяет гибко управлять тем, какую функциональность мы хотим использовать. В одну и ту же переменную можно записать разные варианты поведения и менять их при необходимости. При этом не только не нужно будет менять код по всей программе, но и не придется даже изменять код функций. Достаточно переменной присвоить вместо одной функции — другую.

Представим себе ситуацию, когда необходимо выполнить некую функцию, если задано имя команды. Для простоты предположим, если пришла команда start — надо выполнить функцию start(), если команда stop — функцию stop(), если команда pause — функцию pause().

Такую логику легко можно написать при помощи условного оператора if .. elif.

def start():

# тело функции start

pass

def stop():

# тело функции stop

pass

def pause():

# тело функции pause

pass

command = input() # считываем название команды

if command == 'start':

start()

elif command == 'stop':

stop()

elif command == 'pause'

pause()

Однако если команд будет много или если их количество будет увеличиваться, то оператор if получится слишком громоздким. В этом случае можно создать словарь, где ключом служит название команды, а значением — соответствующая функция.

Приведенный ниже код решает ту же самую задачу, но более гибко:

def start():

# тело функции start

pass

def stop():

# тело функции stop

pass

def pause():

# тело функции pause

pass

commands = {'start': start, 'stop': stop, 'pause': pause} # словарь соответствия команда → функция

command = input() # считываем название команды

commands[command]() # вызываем нужную функцию через словарь по ключу

Таким образом, вся логика обработчика команды сводится к очень простой строке:

commands[command]()

где переменная command — имя команды, которую надо выполнить. Такой код намного гибче!

### Функции в качестве аргументов других функций

Возможность присваивать имя функции переменной позволяет, в частности, передавать имя функции аргументом другой функции. Это доступно во многих языках, но в Python проще, благодаря его гибкой типизации.

Это позволяет легко и красиво решать сложные задачи. Классическая иллюстрация – математические задачи, где функциональная зависимость играет роль внешнего фактора.

Например, есть программная функция построения графика для заданной математической функции. Если нужно строить графики многих математических функций, то каждый раз придется писать новую функцию, или модифицировать имеющуюся.  
  
Но логика построения графика функции практически не зависит от типа математической функции, поэтому можно рассматривать математическую функцию как аргумент программной функции построения графика. Определим функцию plot(), которая принимает 33 аргумента: f – функцию, для которой хотим построить график, и a, b – границы диапазона построения графика.

def plot(f, a, b):

При запуске функции plot() мы можем указать, для какой именно функции строим график. Например, пусть у нас есть следующие математические функции:

def square\_add\_one(x):

return x\*x + 1

def cube\_add\_square(x):

return x\*\*3 + x\*\*2

Чтобы нарисовать график функции �=�2+1*y*=*x*2+1 в диапазоне [1;10][1;10], достаточно выполнить функцию plot() со следующими аргументами: plot(square\_add\_one, 1, 10).

Аналогично, вызов функции plot(cube\_add\_square, -10, 10) построит график функции �=�3+�2*y*=*x*3+*x*2 в диапазоне [−10;10][−10;10].

|  |  |
| --- | --- |
| График функции �=�2+1*y*=*x*2+1 | График функции �=�3+�2*y*=*x*3+*x*2 |

Логику работы функции plot() не нужно менять, достаточно передать ей функциональный объект, используемый внутри. Такой подход работает во многих популярных библиотеках Python. В библиотечные функции мы передаем ту логику, которая нужна нам, вместо стандартной, реализованной в библиотеке.

Функции, способные в качестве аргумента принимать или/и возвращать другие функции, называются функциями **высшего порядка**.

### Встроенные функции, принимающие функции в качестве аргументов

В прошлых уроках мы изучили встроенные функции:

* min(): поиск минимального элемента;
* max(): поиск максимального элемента;
* sorted(): сортировка данных.

Не путайте списочный метод sort() и встроенную функцию sorted(). Они работают одинаково, но списочный метод sort() сортирует список на месте, а функция sorted() возвращает новый, отсортированный список.

Продемонстрируем работу функций на примере:

numbers = [10, -7, 8, -100, -50, 32, 87, 117, -210]

print(max(numbers))

print(min(numbers))

print(sorted(numbers))

Приведенный выше код выводит:

117

-210

[-210, -100, -50, -7, 8, 10, 32, 87, 117]

Но что, если мы хотим написать код для поиска **максимального по модулю** элемента списка numbers? И вообще, сравнивать элементы не стандартным способом, а более специфическими?

На этот случай все выше перечисленные встроенные функции могут принимать необязательный аргумент key – функцию, определяющую условия сравнения элементов. Другими словами, значение key должно быть функцией, принимающей один аргумент и возвращающей на его основе ключ для сравнения.

   Функция, определяющая условия сравнения элементов, называется **компаратор** (compare – сравнивать).

Встроенные функции min(), max(), sorted() – функции высшего порядка, так как принимают в качестве аргумента функцию сравнения элементов.

Продемонстрируем вышесказанное на примере кода:

numbers = [10, -7, 8, -100, -50, 32, 87, 117, -210]

print(max(numbers, key=abs)) # указываем функцию abs в качестве компаратора

print(min(numbers, key=abs)) # указываем функцию abs в качестве компаратора

print(sorted(numbers, key=abs)) # указываем функцию abs в качестве компаратора

Приведенный выше код выводит:

-210 # максимальный по модулю элемент

-7 # минимальный по модулю элемент

[-7, 8, 10, 32, -50, 87, -100, 117, -210] # сортировка на основании модулей элементов

Рассмотрим еще один пример.

Пусть в списке points хранятся в виде кортежей координаты точек плоскости в двумерной биполярной системе координат.

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

При использовании встроенной функции sorted() (или списочного метода sort()) сортировка пройдет по первым значениям пар кортежа, а в случае их совпадения – по вторым.

Таким образом, приведенный ниже код:

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

points.sort() # сортируем список точек на месте

print(points)

выводит:

[(-10, 15), (1, -1), (1, 5), (2, -4), (2, 3), (7, 18), (10, 9)]

Рассмотрим следующий код:

def compare\_by\_second(point):

return point[1]

def compare\_by\_sum(point):

return point[0] + point[1]

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

print(sorted(points, key=compare\_by\_second)) # сортируем по второму значению кортежа

print(sorted(points, key=compare\_by\_sum)) # сортируем по сумме кортежа

Он выводит:

[(2, -4), (1, -1), (2, 3), (1, 5), (10, 9), (-10, 15), (7, 18)]

[(2, -4), (1, -1), (2, 3), (-10, 15), (1, 5), (10, 9), (7, 18)]

### Функции в качестве возвращаемых значений других функций

Объектная сущность функций позволяет и передавать их в качестве аргументов в другие функции, и возвращать одни функции из других. То есть, функции могут быть результатом работы других функций, что позволяет писать генераторы функций, возвращающие функции в зависимости от передаваемых им аргументов.

Рассмотрим код, где функция generator() возвращает функцию hello() в качестве результата своей работы.

def generator():

def hello():

print('Hello from function!')

return hello

Результат работы функции generator() можно записать в переменную, и использовать эту переменную как функцию.

Приведенный ниже код:

func = generator()

func()

выводит:

Hello from function!

   В Python можно определять функцию внутри функции, ведь функция это объект.

Приведенный выше пример не очень информативен, но идею можно использовать и для построения более мощных генераторов функций. Например, рассмотрим семейство функций — квадратных трехчленов. Все эти функции имеют один и тот же вид �(�)=��2+��+�,*f*(*x*)=*ax*2+*bx*+*c*,но поведение конкретного квадратного трехчлена зависит от значения параметров �,�,�*a*,*b*,*c*. Мы можем написать генератор функций, который по параметрам �,�,�*a*,*b*,*c*, построит и вернет нам конкретный квадратный трехчлен:

def generator\_square\_polynom(a, b, c):

def square\_polynom(x):

return a\*x\*\*2 + b\*x + c

return square\_polynom

Приведенный ниже код:

f = generator\_square\_polynom(a=1, b=2, c=1)

g = generator\_square\_polynom(a=2, b=0, c=-3)

h = generator\_square\_polynom(a=-3, b=-10, c=50)

print(f(1))

print(g(2))

print(h(-1))

выводит:

4

5

57

Другими словами мы построили функции �(�)=�2+2�+1,�(�)=2�2−3,ℎ(�)=−3�2−10�+50*f*(*x*)=*x*2+2*x*+1,*g*(*x*)=2*x*2−3,*h*(*x*)=−3*x*2−10*x*+50.

Обратите внимание на то, что внутренняя функция square\_polynom() использует параметры внешней функции generator\_square\_polynom(). Такую вложенную функцию называют **замыканием**.

Замыкания – вложенные функции, ссылающиеся на переменные, объявленные вне определения этой функции, и не являющиеся её параметрами.

## Примечания

**Примечание 1.** Функция sorted() и списочный метод sort() помимо необязательного аргумента key принимают еще аргумент reverse, который по умолчанию имеет значение False, что соответствует сортировке по возрастанию. Если значение reverse установить в True, произойдет сортировка по убыванию.

**Примечание 2.** Сортировка при помощи функции sorted() и списочного метода sort() [стабильна](https://ru.wikipedia.org/wiki/%D0%A3%D1%81%D1%82%D0%BE%D0%B9%D1%87%D0%B8%D0%B2%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0), то есть гарантирует неизменность взаиморасположения равных между собой элементов.

Приведенный ниже код:

def comparator(item):

return item[0]

data = [('red', 1), ('blue', 2), ('green', 5), ('blue', 1)]

data.sort(key=comparator) # сортируем по первому полю

print(data)

выводит:

[('blue', 2), ('blue', 1), ('green', 5), ('red', 1)]

Обратите внимание, что две записи с 'blue' сохранили начальный порядок.

**Примечание 3.** Функции max() и min() возвращают **первый** максимальный или минимальный элемент, если таковых несколько.

**Примечание 4.** Надо четко понимать что сделает код print(input()) и почему это отличается от print(input). Код print(input()) запросит у пользователя текст и тут же его напечатает, т.к. функция input() будет вызвана. Код print(input) выдаст нам текстовое представление этой функции: строку built-in function input, которая поясняет, что input — встроенная функция языка.

# Тема урока: функции высшего порядка

1. Функции высшего порядка
2. Самописные функции map(), filter(), reduce()

**Аннотация.** Урок посвящен функциям высшего порядка.

## Функции высшего порядка

Как уже сказано, функции, которые принимают или/и возвращают другие функции, называются **функциями высшего порядка**.

Давайте реализуем простейшую функцию высшего порядка:

def high\_order\_function(func): # функция высшего порядка, так как принимает функцию

return func(3)

def double(x): # обычная функция = функция первого порядка

return 2\*x

def add\_one(x): # обычная функция = функция первого порядка

return x + 1

Здесь функция high\_order\_function() принимает другую функцию на входе и возвращает результат её вызова с аргументом, равным 33.

Приведенный ниже код:

print(high\_order\_function(double))

print(high\_order\_function(add\_one))

выводит:

6

4

Функции первого порядка принимают и возвращают "обычные" значения, не функции, функции высшего порядка принимают или/и возвращают другие функции.

## Функции высшего порядка для обработки набора данных

Часто функции высшего порядка используются для обработки наборов данных. Рассмотрим три важные функции высшего порядка:

* map();
* filter();
* reduce().

В языке Python эти функции уже реализованы, однако для лучшего понимания их работы мы сначала напишем свои версии этих функций, и уже после этого поговорим о встроенных реализациях.

Функции высшего порядка map(), filter() и reduce() довольно широко распространены в функциональном программировании и часто применяются программистами.

### Функция map()

При работе со списками часто требуется применить одно и то же преобразование к каждому элементу. Можно написать цикл, содержащий нужное преобразование.

Например, для преобразования списка чисел в список их квадратов, код может выглядеть так:

def f(x):

return x\*\*2 # тело функции, которая преобразует аргумент x

old\_list = [1, 2, 4, 9, 10, 25]

new\_list = []

for item in old\_list:

new\_item = f(item)

new\_list.append(new\_item)

print(old\_list)

print(new\_list)

Результатом работы такого кода будет:

[1, 2, 4, 9, 10, 25]

[1, 4, 16, 81, 100, 625]

Несложно понять, что цикл будет выглядеть одинаково практически во всех случаях. Меняться будет только преобразование, то есть применяемая функция f(). Так почему бы не обобщить код, чтобы функция была параметром? Так и сделаем:

def map(function, items):

result = []

for item in items:

new\_item = function(item)

result.append(new\_item)

return result

Теперь мы можем совершать преобразования, используя функцию высшего порядка map().

Приведенный ниже код:

def square(x):

return x\*\*2

def cube(x):

return x\*\*3

numbers = [1, 2, -3, 4, -5, 6, -9, 0]

strings = map(str, numbers) # используем в качестве преобразователя - функцию str

abs\_numbers = map(abs, numbers) # используем в качестве преобразователя - функцию abs

squares = map(square, numbers) # используем в качестве преобразователя - функцию square

cubes = map(cube, numbers) # используем в качестве преобразователя - функцию cube

print(strings)

print(abs\_numbers)

print(squares)

print(cubes)

выводит:

['1', '2', '-3', '4', '-5', '6', '-9', '0']

[1, 2, 3, 4, 5, 6, 9, 0]

[1, 4, 9, 16, 25, 36, 81, 0]

[1, 8, -27, 64, -125, 216, -729, 0]

Функция называется "map" то есть "отобразить". Название пришло из математики, где так называется функция, отображающая одно множество значений в другое путём преобразования всех элементов с помощью некой трансформации.

Реализованную нами функцию map() можно использовать как альтернативную возможность для преобразования типов элементов любого списка. Раньше мы решали такую задачу с помощью списочных выражений. Теперь можем использовать и функцию map().

Приведенный ниже код, при условии, что функция map() определена как указано выше:

strings = ['10', '12', '-4', '-9', '0', '1', '23', '100', '99']

numbers1 = [int(c) for c in strings] # используем списочное выражение для преобразования

numbers2 = map(int, strings) # используем функцию map() для преобразования

print(numbers1)

print(numbers2)

выводит:

[10, 12, -4, -9, 0, 1, 23, 100, 99]

[10, 12, -4, -9, 0, 1, 23, 100, 99]

### Цепочки преобразований

Мы также можем строить цепочки преобразований, несколько раз вызывая функцию map().

Приведенный ниже код, при условии, что функция map() определена как указано выше:

numbers = ['-1', '20', '3', '-94', '65', '6', '-970', '8']

new\_numbers = map(abs, map(int, numbers))

print(new\_numbers)

выводит:

[1, 20, 3, 94, 65, 6, 970, 8]

То есть, сначала мы преобразуем список строк в список чисел с помощью кода map(int, numbers), получая список [-1, 20, 3, -94, 65, 6, -970, 8]. Далее с помощью еще одного вызова функции map() трансформируем полученный список в список [1, 20, 3, 94, 65, 6, 970, 8].

### Функция filter()

Другая популярная задача при работе со списками: отобрать часть элементов списка по определенному критерию. Функция высшего порядка для решения такой задачи называется filter().

   Функция-критерий, которая возвращает значение True или False , называется предикатом.

Реализация такой функции может выглядеть так:

def filter(function, items):

result = []

for item in items:

if function(item):

result.append(item) # добавляем элемент item если функция function вернула значение True

return result

Наша функция filter() применяет предикат function к каждому элементу и добавляет в итоговый список только те элементы, для которых предикат вернул True.

Например, чтобы из исходного списка чисел получить список с элементами, большими 1010, можно написать такой код:

def is\_greater10(num): # функция возвращает значение True если число больше 10 и False в противном случае

return num > 10

numbers = [12, 2, -30, 48, 51, -60, 19, 10, 13]

large\_numbers = filter(is\_greater10, numbers) # список large\_numbers содержит элементы, большие 10

print(large\_numbers)

Этот код, при условии, что функция filter() определена как указано выше выводит:

[12, 48, 51, 19, 13]

Рассмотрим еще пару примеров применения реализованной функции filter().

Приведенный ниже код, при условии, что функция filter() определена как указано выше:

def is\_odd(num):

return num % 2

def is\_word\_long(word):

return len(word) > 6

numbers = list(range(15))

words = ['В', 'новом', 'списке', 'останутся', 'только', 'длинные', 'слова']

odd\_numbers = filter(is\_odd, numbers)

large\_words = filter(is\_word\_long, words)

print(odd\_numbers)

print(large\_words)

выводит:

[1, 3, 5, 7, 9, 11, 13]

['останутся', 'длинные']

### Функция reduce()

Реализованные нами функции map() и filter() работали с отдельными элементами списка независимо. Но встречаются циклы с агрегацией результата — формированием одного результирующего значения при комбинации элементов с использованием аргумента-аккумулятора.

Типичные примеры агрегации — сумма всех элементов списка или их произведение.

Приведенный ниже код:

numbers = [1, 2, 3, 4, 5]

total = 0

product = 1

for num in numbers:

total += num

product \*= num

print(total)

print(product)

вычисляет сумму и произведение элементов списка и выводит:

15

120

С точки зрения математики сумма 1+2+3+4+51+2+3+4+5 может быть выражена как:

(((((0+1)+2)+3)+4)+5).(((((0+1)+2)+3)+4)+5).Ноль здесь тот самый аккумулятор, точнее его начальное значение. Он не добавляет к сумме ничего, поэтому может служить отправной точкой. А еще будет результатом, если входной список пуст.

Несложно понять, что этот цикл будет выглядеть одинаково практически во всех случаях. Меняться будет только начальное значение аккумулятора (00 для суммы, 11 для произведения и т.д.) и операция, которая комбинирует элемент и аккумулятор. Так почему бы не обобщить этот код? Так и сделаем:

def reduce(operation, items, initial\_value):

acc = initial\_value

for item in items:

acc = operation(acc, item)

return acc

Приведенный ниже код, при условии, что функция reduce() определена как указано выше:

def add(x, y):

return x+y

def mult(x, y):

return x\*y

numbers = [1, 2, 3, 4, 5]

total = reduce(add, numbers, 0)

product = reduce(mult, numbers, 1)

print(total)

print(product)

выводит:

15

120

## Примечания

**Примечание 1.** Мы с вами реализовали три функции:

* map(): преобразование элементов списка;
* filter(): фильтрация элементов списка;
* reduce(): агрегация элементов списка.

Каждая функция имеет меньшую мощность, чем цикл for. Цикл for позволяет гибко управлять процессом итерации, мы можем использовать даже команды break и continue. Возникает резонный вопрос: зачем нужны отдельные функции, когда можно обойтись циклом?

Во-первых, такие функции — часть функционального подхода.

Во-вторых, каждая такая функция делает единственную работу, что значительно упрощает рассуждение о коде, его чтение и написание. Взглянув на имя функции можно понять, что filter() отфильтрует, а map() — преобразует элементы. Более того, по построению функция filter() не меняет элементы, а лишь отбрасывает их часть. А функция map() меняет значение элементов, но не меняет их количество и позиции.

**Примечание 2.** В математике определенная нами функция reduce() называется левая свёртка (left fold), по сути мы сворачиваем список в одно значение, начиная слева. Существует ещё и правая свёртка (right fold). В большинстве случаев обе свёртки дают одинаковый результат, если применяемая операция [ассоциативна](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)).

# Тема урока: встроенные функции map(), filter(), reduce()

1. Встроенные функции map(), filter(), reduce()
2. Модуль operator

**Аннотация.** Урок посвящен встроенным функциям map(), filter() и reduce() и модулю operator.

## Встроенные функции map(), filter(), reduce()

Язык Python имеет встроенные реализации функций высшего порядка map(), filter() и reduce() , которые намного удобнее, чем наши собственные версии.

### Встроенная функция map()

Встроенная функция map() имеет сигнатуру map(func, \*iterables). В отличие от нашей версии из прошлого урока, встроенная функция map() может принимать сразу несколько последовательностей, переменное количество аргументов.

В качестве параметра func указывается функция, которой будет передаваться текущий элемент последовательности. Внутри функции func необходимо вернуть новое значение. Для примера прибавим к каждому элементу списка число 77.

Приведенный ниже код:

def increase(num):

return num + 7

numbers = [1, 2, 3, 4, 5, 6]

new\_numbers = map(increase, numbers) # используем встроенную функцию map()

print(new\_numbers)

выведет не список, а специальный объект:

<map object at 0x...>

Такой объект похож на список тем, что его можно перебирать циклом for, то есть итерировать. Такие объекты в Python называют **итераторами**.

Приведенный ниже код:

def increase(num):

return num + 7

numbers = [1, 2, 3, 4, 5, 6]

new\_numbers = map(increase, numbers)

for num in new\_numbers: # итерируем циклом for

print(num)

выводит:

8

9

10

11

12

13

Чтобы получить из итератора список, нужно воспользоваться функцией list():

new\_numbers = list(map(increase, numbers))

Функция map() возвращает объект, поддерживающий итерации, а не список. Чтобы получить из него список, необходимо результат передать в функцию list().

Функции map() можно передать несколько последовательностей. В этом случае в функцию обратного вызова func будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковых позициях.

Приведенный ниже код суммирует элементы трех списков:

def func(elem1, elem2, elem3):

return elem1 + elem2 + elem3

numbers1 = [1, 2, 3, 4, 5]

numbers2 = [10, 20, 30, 40, 50]

numbers3 = [100, 200, 300, 400, 500]

new\_numbers = list(map(func, numbers1, numbers2, numbers3)) # преобразуем итератор в список

print(new\_numbers)

и выводит:

[111, 222, 333, 444, 555]

Если в последовательностях разное количество элементов, то последовательность с минимальным количеством элементов становится ограничителем.

Приведенный ниже код:

def func(elem1, elem2, elem3):

return elem1 + elem2 + elem3

numbers1 = [1, 2, 3, 4]

numbers2 = [10, 20]

numbers3 = [100, 200, 300, 400, 500]

new\_numbers = list(map(func, numbers1, numbers2, numbers3)) # преобразуем итератор в список

print(new\_numbers)

выводит:

[111, 222]

Встроенная функция map() реализована очень гибко. В качестве последовательностей мы можем использовать: списки, строки, кортежи, множества, словари.

Приведем пример удобного использования встроенной функции map(), которой передано две последовательности.

Приведенный ниже код:

circle\_areas = [3.56773, 5.57668, 4.31914, 6.20241, 91.01344, 32.01213]

result1 = list(map(round, circle\_areas, [1]\*6)) # округляем числа до 1 знака после запятой

result2 = list(map(round, circle\_areas, range(1, 7))) # округляем числа до 1,2,...,6 знаков после запятой

print(circle\_areas)

print(result1)

print(result2)

выводит:

[3.56773, 5.57668, 4.31914, 6.20241, 91.01344, 32.01213]

[3.6, 5.6, 4.3, 6.2, 91.0, 32.0]

[3.6, 5.58, 4.319, 6.2024, 91.01344, 32.01213]

Встроенная функция round(x, n=0) принимает два числовых аргумента x и n и округляет переданное число x до n цифр после десятичной запятой. Значением по умолчанию для n является 00.

### Встроенная функция filter()

Встроенная функция filter() имеет сигнатуру filter(func, iterable). В отличие от нашей реализации из прошлого урока она может принимать любой итерируемый объект (список, строку, кортеж, и т.д.).

В качестве параметра func указывается ссылка на функцию, которой будет передаваться текущий элемент последовательности. Внутри функции func необходимо вернуть значение True или False. Для примера, удалим все отрицательные значения из списка.

Приведенный ниже код:

def func(elem):

return elem >= 0

numbers = [-1, 2, -3, 4, 0, -20, 10]

positive\_numbers = list(filter(func, numbers)) # преобразуем итератор в список

print(positive\_numbers)

выводит:

[2, 4, 0, 10]

Обратите внимание: функция filter() как и функция map() возвращает не список, а специальный объект, который называется итератором. Итераторы можно перебрать с помощью цикла for, либо преобразовать в список.

Встроенной функции filter() можно в качестве первого параметра func передать значение None. В таком случае каждый элемент последовательности будет проверен на соответствие значению True. Если элемент в логическом контексте возвращает значение False, то он не будет добавлен в возвращаемый результат.

   О преобразовании основных типов в булево значение (True/False) можно почитать в [этом](https://stepik.org/lesson/440563/step/1?unit=430761) уроке.

Приведенный ниже код:

true\_values = filter(None, [1, 0, 10, '', None, [], [1, 2, 3], ()])

for value in true\_values:

print(value)

выводит:

1

10

[1, 2, 3]

В данном случае, значения списка: 0, '', None, [], () позиционируется как False, а значения 1, 10, [1, 2, 3] как True.

### Функция reduce()

Для использования функции reduce() необходимо подключить специальный модуль functools. Функция reduce() имеет сигнатуру reduce(func, iterable, initializer=None). Если начальное значение не установлено, то в его качестве используется первое значение из последовательности iterable.

Приведенный ниже код:

from functools import reduce

def func(a, b):

return a + b

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

total = reduce(func, numbers, 0) # в качестве начального значения 0

print(total)

выводит:

55

Обратите внимание на то, что мы могли вызвать функцию так:

total = reduce(func, numbers) # в качестве начального значения первый элемент списка numbers

Функция reduce() во второй версии языка Python была встроенной, но в Python 3 ее решили перенести в модуль functools. Функция reduce() как и функции map() и filter() может принимать любой итерируемый объект.

## Модуль operator

Чтобы не писать каждый раз функции, определяющие такие стандартные математические операции как сумма или произведение:

def sum\_func(a, b):

return a + b

def mult\_func(a, b):

return a \* b

можно использовать уже реализованные функции из модуля operator.

Неполный список функций из модуля operator выглядит так:

|  |  |  |
| --- | --- | --- |
| **Операция** | **Синтаксис** | **Функция** |
| Addition | a + b | add(a, b) |
| Containment Test | obj in seq | contains(seq, obj) |
| Division | a / b | truediv(a, b) |
| Division | a // b | floordiv(a, b) |
| Exponentiation | a \*\* b | pow(a, b) |
| Modulo | a % b | mod(a, b) |
| Multiplication | a \* b | mul(a, b) |
| Negation (Arithmetic) | -a | neg(a) |
| Subtraction | a - b | sub(a, b) |
| Ordering | a < b | lt(a, b) |
| Ordering | a <= b | le(a, b) |
| Equality | a == b | eq(a, b) |
| Difference | a != b | ne(a, b) |
| Ordering | a >= b | ge(a, b) |
| Ordering | a > b | gt(a, b) |

Рассмотрим примеры использования функций из модуля operator.

Приведенный ниже код:

from operator import \* # импортируем все функции

print(add(10, 20)) # сумма

print(floordiv(20, 3)) # целочисленное деление

print(neg(9)) # смена знака

print(lt(2, 3)) # проверка на неравенство <

print(lt(10, 8)) # проверка на неравенство <

print(eq(5, 5)) # проверка на равенство ==

print(eq(5, 9)) # проверка на равенство ==

выводит:

30

6

-9

True

False

True

False

Приведенный ниже код:

from functools import reduce

import operator

words = ['Testing ', 'shows ', 'the ', 'presence', ', ', 'not ', 'the ', 'absence ', 'of ', 'bugs']

numbers = [1, 2, -6, -4, 3, 9, 0, -6, -1]

opposite\_numbers = list(map(operator.neg, numbers)) # смена знаков элементов списка

concat\_words = reduce(operator.add, words) # конкатенация элементов списка

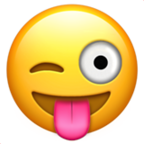
print(opposite\_numbers)

print(concat\_words)

Выводит:

[-1, -2, 6, 4, -3, -9, 0, 6, 1]

Testing shows the presence, not the absence of bugs

Модуль operator реализован на языке C, поэтому функции этого модуля работают в разы быстрее, чем самописные функции в Python.

## Примечания

**Примечание 1.** Итераторы – важная концепция языка Python. Нужно помнить:

* итераторы можно обойти циклом for;
* итератор можно преобразовать в список или кортеж, с помощью функций list() и tuple();
* итератор можно **распаковать** с помощью \* 😎.

Приведенный ниже код:

numbers = [1, 10, -9, 8, 9, 345, -32, -89, 2]

map\_obj = map(abs, numbers)

print(\*map\_obj) # распаковываем

выводит:

1 10 9 8 9 345 32 89 2

**Примечание 2.** Если нам нужны строковые методы в виде функций, мы можем получить их через название типа str.

Приведенный ниже код:

pets = ['alfred', 'tabitha', 'william', 'arla']

chars = ['x', 'y', '2', '3', 'a']

uppered\_pets = list(map(str.upper, pets))

capitalized\_pets = list(map(str.capitalize, pets))

only\_letters = list(filter(str.isalpha, chars))

print(uppered\_pets)

print(capitalized\_pets)

print(only\_letters)

выводит:

['ALFRED', 'TABITHA', 'WILLIAM', 'ARLA']

['Alfred', 'Tabitha', 'William', 'Arla']

['x', 'y', 'a']

Аналогично можно получить методы других типов данных в виде функций.

**Примечание 3.** Подробнее прочитать про модуль operator можно в официальной [документации](https://docs.python.org/3/library/operator.html).

**Примечание 4.** В уроке, посвященном списочным выражениям, мы разбирали конструкции очень похожие на действие функций map() и filter(). Списочное выражение можно рассматривать как комбинацию фильтрации и трансформации: сначала выполняется фильтрация, затем — трансформирование.

# Тема урока: анонимные функции

1. Определение анонимных функций
2. Условный оператор в теле анонимной функции
3. Передача аргументов в анонимную функцию
4. Ограничения анонимных функций

**Аннотация.** Урок посвящен работе с анонимными функциями.

В языке Python можно создавать функции с уникальными именами, но иногда имя не требуется. Например, функциям, использующимся единственный раз. В таких случаях применяют анонимные функции.

## Анонимные функции

Помимо стандартного определения функции, состоящего из ее заголовка с ключевым словом def и блока кода – тела функции, в Python можно создавать короткие однострочные функции с использованием оператора lambda. Это анонимные функции или лямбда-функции.

   Анонимные функции – функции с телом, но без имени.

Общий формат определения анонимной функции: lambda список\_параметров: выражение.

Тут список\_параметров – список параметров через запятую, выражение – значение, либо код, дающий значение.

   Параметры анонимных функций, в отличие от обычных, не нужно заключать в скобки.

Следующие два определения функций эквивалентны:

def standard\_function(x): # стандартное объявление функции

return x\*2

lambda\_function = lambda x: x\*2 # объявление анонимной функции

Приведенный ниже код:

print(standard\_function(7))

print(lambda\_function(7))

выводит:

14

14

Рассмотрим еще примеры. Приведенный ниже код:

f1 = lambda: 10 + 20 # функция без параметров

f2 = lambda х, у: х + у # функция с двумя параметрами

f3 = lambda х, у, z: х + у + z # функция с тремя параметрами

print(f1())

print(f2(5, 10))

print(f3(5, 10, 30))

выводит:

30

15

45

Когда применение анонимных функций оправдано:

* однократное использование функции;
* передача функций в качестве аргументов другим функциям;
* возвращение функции в качестве результата другой функции.

### Однократное использование функции

В одном из прошлых уроков нам приходилось сортировать список кортежей. Как мы уже знаем, встроенная  функция sorted() (или списочный метод sort()) сортируют по первым значениям кортежей, а в случае их совпадения, по вторым и т.д. Для сортировки, отличной от стандартной нам приходилось создавать отдельные функции-компараторы для сравнения элементов:

def compare\_by\_second(point):

return point[1]

def compare\_by\_sum(point):

return point[0] + point[1]

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

print(sorted(points, key=compare\_by\_second)) # сортируем по второму значению кортежа

print(sorted(points, key=compare\_by\_sum)) # сортируем по сумме кортежа

Очевидно, что такие функции как compare\_by\_second() и compare\_by\_sum() не особо нужны вне контекста сортировки, поэтому логично их заменить на анонимные функции:

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

print(sorted(points, key=lambda point: point[1])) # сортируем по второму значению кортежа

print(sorted(points, key=lambda point: point[0] + point[1])) # сортируем по сумме элементов кортежа

   Название аргумента point в анонимной функции можно заменить на любое другое.

### Передача анонимных функций в качестве аргументов другим функциям

Функции высшего порядка map(), filter() и reduce() идеально подойдут для демонстрации удобства анонимных функций в качестве аргументов других функций.

Теперь нет необходимости делать преобразующую элементы функцию отдельно определенной именованной функцией.

Приведенный ниже код:

numbers = [1, 2, 3, 4, 5, 6]

new\_numbers1 = list(map(lambda x: x+1, numbers)) # увеличиваем на 1

new\_numbers2 = list(map(lambda x: x\*2, numbers)) # удваиваем

new\_numbers3 = list(map(lambda x: x\*\*2, numbers)) # возводим в квадрат

print(new\_numbers1)

print(new\_numbers2)

print(new\_numbers3)

выводит:

[2, 3, 4, 5, 6, 7]

[2, 4, 6, 8, 10, 12]

[1, 4, 9, 16, 25, 36]

Приведенный ниже код:

strings = ['a', 'b', 'c', 'd', 'e']

numbers = [3, 2, 1, 4, 5]

new\_strings = list(map(lambda x, y: x\*y, strings, numbers))

print(new\_strings)

выводит:

['aaa', 'bb', 'c', 'dddd', 'eeeee']

Рассмотрим примеры использования анонимных функций в качестве аргумента функции filter().

Приведенный ниже код:

numbers = [-1, 2, -3, 4, 0, -20, 10, 30, -40, 50, 100, 90]

positive\_numbers = list(filter(lambda x: x > 0, numbers)) # положительные числа

large\_numbers = list(filter(lambda x: x > 50, numbers)) # числа, большие 50

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers)) # четные числа

print(positive\_numbers)

print(large\_numbers)

print(even\_numbers)

выводит:

[2, 4, 10, 30, 50, 100, 90]

[100, 90]

[2, 4, 0, -20, 10, 30, -40, 50, 100, 90]

Приведенный ниже код:

words = ['python', 'stepik', 'beegeek', 'iq-option']

new\_words1 = list(filter(lambda w: len(w) > 6, words)) # слова длиною больше 6 символов

new\_words2 = list(filter(lambda w: 'e' in w, words)) # слова содержащие букву e

print(new\_words1)

print(new\_words2)

выводит:

['beegeek', 'iq-option']

['stepik', 'beegeek']

Рассмотрим примеры использования анонимных функций в качестве аргументов функции reduce().

Приведенный ниже код:

from functools import reduce

words = ['python', 'stepik', 'beegeek', 'iq-option']

numbers = [1, 2, 3, 4, 5, 6]

summa = reduce(lambda x, y: x + y, numbers, 0)

product = reduce(lambda x, y: x \* y, numbers, 1)

sentence = reduce(lambda x, y: x + ' loves ' + y, words, 'Everyone')

print(summa)

print(product)

print(sentence)

выводит:

21

720

Everyone loves python loves stepik loves beegeek loves iq-option

### Возвращение функции в качестве результата другой функции

Анонимные функции могут быть результатом работы других функций.

Приведенный ниже код по значениям �,�,�*a*,*b*,*c* строит и возвращает квадратный трехчлен:

def generator\_square\_polynom(a, b, c):

def square\_polynom(x):

return a\*x\*\*2 + b\*x + c

return square\_polynom

Такой код можно переписать так:

def generator\_square\_polynom(a, b, c):

return lambda x: a\*x\*\*2 + b\*x + c

Этот пример показывает, что анонимные функции являются **замыканиями**: возвращаемая функция запоминает значения переменных a, b, c из внешнего окружения.

## Условный оператор в теле анонимной функции

В теле анонимной функции не получится выполнить несколько действий и не получится использовать многострочные конструкции вроде циклов for и while. Однако можно использовать тернарный условный оператор.

Приведенный ниже код:

numbers = [-2, 0, 1, 2, 17, 4, 5, 6]

result = list(map(lambda x: 'even' if x % 2 == 0 else 'odd', numbers))

print(result)

выводит:

['even', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'even']

Общий вид тернарного условного оператора в теле анонимной функции выглядит так:

значение1 if условие else значение2

Если условие истинно, возвращается значение1 , если нет – значение2.

Анонимные функции не могут содержать многострочных конструкций, зато их легко читать. Этого сложно было бы добиться, разреши авторы "многострочные" анонимные функции.

## Передача аргументов в анонимную функцию

Как и обычные функции, определенные с помощью ключевого слова def , анонимные функции поддерживают все способы передачи аргументов:

* позиционные аргументы;
* именованные аргументы;
* переменный список позиционных аргументов (\*args);
* переменный список именованных аргументов (\*\*kwargs);
* обязательные аргументы (\*).

Приведенный ниже код:

f1 = lambda x, y, z: x + y + z

f2 = lambda x, y, z=3: x + y + z

f3 = lambda \*args: sum(args)

f4 = lambda \*\*kwargs: sum(kwargs.values())

f5 = lambda x, \*, y=0, z=0: x + y + z

print(f1(1, 2, 3))

print(f2(1, 2))

print(f2(1, y=2))

print(f3(1, 2, 3, 4, 5))

print(f4(one=1, two=2, three=3))

print(f5(1))

print(f5(1, y=2, z=3))

выводит:

6

6

6

15

6

1

6

## Ограничения анонимных функций

Особенности и ограничения анонимных функций в Python:

* анонимная функция может содержать только выражение, и не может включать в свое тело операторы;
* в теле анонимной функции такие операторы, как return, pass, assert или raise, вызовут исключение SyntaxError;
* анонимная функция пишется как одна строка исполнения;
* анонимная функция может быть немедленно вызвана 🤓.

## Примечания

**Примечание 1.** Интересная особенность анонимных функций (лямбда-функций) – они являются выражениями. После определения лямбда-функции ее можно сразу же вызвать.

Приведенный ниже код:

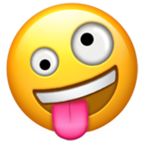
print((lambda х, у: х + у)(5, 10)) # 5 + 10

print(1 + (lambda x: x\*5)(10) + 2) # 1 + 50 + 2

выводит:

15

53

   Анонимные функции можно конструировать и тут же вызывать. На практике это редко применяется.

**Примечание 2.** В лямбда исчислении, часто применяемом в разработке языков программирования, все функции — анонимные, поэтому анонимные функции во многих языках тоже называют "лямбдами" или "лямбда-функциями".

**Примечание 3.** В Python анонимные функции — лишь сокращенная запись функции, поэтому приведенный ниже код:

f = lambda x: x + 1

print(type(f))

выводит:

<class 'function'>

То есть, анонимные функции имеют такой же тип, как и обычные функции.

**Примечание 4.** Анонимные функции **очень часто** используются вместе со встроенными функциями map(), filter(), reduce(), sorted(), max(), min() и т.д.

# Тема урока: функции any(), all(), zip(), enumerate()

1. Функция all()
2. Функция any()
3. Функция enumerate()
4. Функция zip()

**Аннотация.** Урок посвящен встроенным функциям all(), any(), zip(), enumerate().

## Функции all() и any()

При работе с коллекциями часто приходится определять, выполняется ли некоторое условие одновременно для всех элементов последовательности или хотя бы для одного. Для этого существуют две встроенные функции all() и any().

### Функция all()

Встроенная функция all() возвращает значение True, если **все элементы** переданной ей последовательности (итерируемого объекта) истинны (приводятся к значению True), и False в противном случае.

Сигнатура функции следующая: all(iterable). В качестве iterable может выступать любой итерируемый объект:

* список;
* кортеж;
* строка;
* множество;
* словарь и т.д.

Приведенный ниже код:

print(all([True, True, True])) # возвращает True, так как все значения списка равны True

print(all([True, True, False])) # возвращает False, так как не все значения списка равны True

выводит:

True

False

Напомним, что в Python все следующие значения приводятся к значению False:

* константы None и False;
* нули всех числовых типов данных: 0, 0.0, 0j, Decimal(0), Fraction(0, 1);
* пустые коллекции: '', (), [], {}, set(), range(0).

Приведенный ниже код:

print(all([1, 2, 3]))

print(all([1, 2, 3, 0, 5]))

print(all([True, 0, 1]))

print(all(('', 'red', 'green')))

print(all({0j, 3+4j}))

выводит:

True

False

False

False

False

При работе со словарями функция all() проверяет на соответствие параметрам True ключи словаря, а не их значения.

Приведенный ниже код:

dict1 = {0: 'Zero', 1: 'One', 2: 'Two'}

dict2 = {'Zero': 0, 'One': 1, 'Two': 2}

print(all(dict1))

print(all(dict2))

выводит:

False

True

Обратите внимание: если переданный итерируемый объект пустой, то функция all() возвращает значение True.

Приведенный ниже код:

print(all([])) # передаем пустой список

print(all(())) # передаем пустой кортеж

print(all('')) # передаем пустую строку

print(all([[], []])) # передаем список, содержащий пустые списки

выводит:

True

True

True

False

### Функция any()

Встроенная функция any() возвращает значение True, если **хотя бы один элемент** переданной ей последовательности (итерируемого объекта) является истинным (приводится к значению True), и False в противном случае.

Сигнатура функции следующая: any(iterable). В качестве iterable может выступать любой итерируемый объект:

* список;
* кортеж;
* строка;
* множество;
* словарь и т.д.

Приведенный ниже код:

print(any([False, True, False])) # возвращает True, так как есть хотя бы один элемент, равный True

print(any([False, False, False])) # возвращает False, так как нет элементов, равных True

выводит:

True

False

Приведенный ниже код:

print(any([0, 0, 0]))

print(any([0, 1, 0]))

print(any([False, 0, 1]))

print(any(['', [], 'green']))

print(any({0j, 3+4j, 0.0}))

выводит:

False

True

True

True

True

При работе со словарями функция any() проверяет на соответствие True ключи словаря, а не их значения.

Приведенный ниже код:

dict1 = {0: 'Zero'}

dict2 = {'Zero': 0, 'One': 1}

print(any(dict1))

print(any(dict2))

выводит:

False

True

Обратите внимание: если переданный объект пуст, то функция any() возвращает значение False.

Приведенный ниже код:

print(any([])) # передаем пустой список

print(any(())) # передаем пустой кортеж

print(any('')) # передаем пустую строку

print(any([[], []])) # передаем список, содержащий пустые списки

выводит:

False

False

False

False

### Функции all() и any() в связке с функцией map()

Функции all() и any() могут быть полезны в комбинации с функцией map(), которая может преобразовывать элементы последовательности (итерируемого объекта) к значению True/False в соответствии с неким условием.

Приведенный ниже код, проверяет, **все ли элементы** списка numbers больше 1010:

numbers = [17, 90, 78, 56, 231, 45, 5, 89, 91, 11, 19]

result = all(map(lambda x: True if x > 10 else False, numbers))

if result:

print('Все числа больше 10')

else:

print('Хотя бы одно число меньше или равно 10')

и выводит:

Хотя бы одно число меньше или равно 10

так как список numbers содержит число 55, которое не больше чем 1010.

Лямбда функцию, которая преобразует элементы списка numbers в значения True/False можно упростить следующим образом:

result = all(map(lambda x: x > 10, numbers))

Приведенный ниже код, проверяет, что **хотя бы один элемент** списка четное число:

numbers = [17, 91, 78, 55, 231, 45, 5, 89, 99, 11, 19]

result = any(map(lambda x: x % 2 == 0, numbers))

if result:

print('Хотя бы одно число четное')

else:

print('Все числа нечетные')

и выводит:

Хотя бы одно число четное

так как список numbers содержит четное число 7878.

## Функция enumerate()

Встроенная функция enumerate() возвращает кортеж из индекса элемента и самого элемента переданной ей последовательности (итерируемого объекта).

Сигнатура функции следующая: enumerate(iterable, start). В качестве iterable может выступать любой итерируемый объект:

* список;
* кортеж;
* строка;
* множество;
* словарь и т.д.

С помощью необязательного параметра start можно задать начальное значение индекса. По умолчанию значение параметра start = 0, то есть счет начинается с нуля.

Приведенный ниже код:

colors = ['red', 'green', 'blue']

for pair in enumerate(colors):

print(pair)

выводит:

(0, 'red')

(1, 'green')

(2, 'blue')

Если счет нужно начать с отличного от нуля числа, то нужно передать значение аргумента start.

Приведенный ниже код:

colors = ['red', 'green', 'blue']

for pair in enumerate(colors, 100):

print(pair)

выводит:

(100, 'red')

(101, 'green')

(102, 'blue')

Обратите внимание, функция enumerate() возвращает не список, а специальный объект, который называется итератором. Такой объект похож на список тем, что его можно перебирать циклом for, то есть итерировать. Итератор можно преобразовать в список с помощью функции list().

Приведенный ниже код:

colors = ['red', 'green', 'blue']

pairs = enumerate(colors)

print(pairs)

print(list(pairs))

выводит:

<enumerate object at 0x...>

[(0, 'red'), (1, 'green'), (2, 'blue')]

Мы также можем использовать распаковку кортежей при итерировании с помощью цикла for.

Приведенный ниже код:

colors = ['red', 'green', 'blue']

for index, item in enumerate(colors):

print(index, item)

выводит:

0 red

1 green

2 blue

Такой код является альтернативой коду:

colors = ['red', 'green', 'blue']

for i in range(len(colors)):

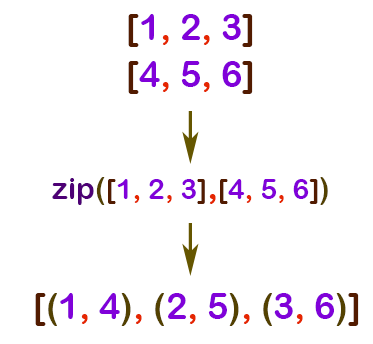
print(i, colors[i])

## Функция zip()

Встроенная функция zip() объединяет отдельные элементы из каждой переданной ей последовательности (итерируемого объекта) в кортежи.

Сигнатура функции следующая: zip(\*iterables). В качестве iterable может выступать любой итерируемый объект:

* список;
* кортеж;
* строка;
* множество;
* словарь и т.д.



Приведенный ниже код:

numbers = [1, 2, 3]

words = ['one', 'two', 'three']

for pair in zip(numbers, words):

print(pair)

выводит:

(1, 'one')

(2, 'two')

(3, 'three')

Функция zip(), как и функция enumerate() возвращает не список, а специальный объект, который называется итератором. Такой объект похож на список тем, что его можно перебирать циклом for, то есть итерировать. Итератор можно преобразовать в список с помощью функции list().

Приведенный ниже код:

numbers = [1, 2, 3]

words = ['one', 'two', 'three']

result = zip(numbers, words)

print(result)

print(list(result))

выводит:

<zip object at 0x...>

[(1, 'one'), (2, 'two'), (3, 'three')]

Мы можем передавать функции zip() сколько угодно итерируемых объектов.

Приведенный ниже код:

numbers = [1, 2, 3]

words = ['one', 'two', 'three']

romans = ['I', 'II', 'III']

result = zip(numbers, words, romans)

print(list(result))

выводит:

[(1, 'one', 'I'), (2, 'two', 'II'), (3, 'three', 'III')]

Мы можем передать функции zip() даже один итерируемый объект.

Приведенный ниже код:

numbers = [1, 2, 3]

result = zip(numbers)

print(list(result))

выводит:

[(1,), (2,), (3,)]

Если функции zip() передать итерируемые объекты, имеющие разную длину, то объект с наименьшим количеством элементов определяет итоговую длину.

Приведенный ниже код:

numbers = [1, 2, 3, 4]

words = ['one', 'two']

romans = ['I', 'II', 'III']

result = zip(numbers, words, romans)

print(list(result))

выводит:

[(1, 'one', 'I'), (2, 'two', 'II')]

### Частые сценарии использования функции zip()

**Сценарий 1.** Функция zip() удобна для создания словарей, когда ключи и значения находятся в разных списках.

Приведенный ниже код:

keys = ['name', 'age', 'gender']

values = ['Timur', 28, 'male']

info = dict(zip(keys, values))

print(info)

выводит:

{'name': 'Timur', 'age': 28, 'gender': 'male'}

**Сценарий 2.** Функция zip() удобна для одновременного (параллельного) итерирования сразу по нескольким коллекциям.

Приведенный ниже код:

name = ['Timur', 'Ruslan', 'Rustam']

age = [28, 21, 19]

for x, y in zip(name, age):

print(x, y)

выводит:

Timur 28

Ruslan 21

Rustam 19

## Примечания

**Примечание 1.** Итераторы – важная концепция языка Python. Нужно помнить:

* итераторы можно обойти циклом for;
* итератор можно преобразовать в список или кортеж, с помощью функций list() и tuple();
* итератор можно распаковать с помощью \*.

**Примечание 2.** Реализация встроенных функций all() и any() выглядит **примерно** так:

def all(iterable):

for item in iterable:

if not item:

return False

return True

def any(iterable):

for item in iterable:

if item:

return True

return False

**Примечание 3.** Мы можем использовать одновременно функции zip() и enumerate():

Приведенный ниже код:

list1 = ['a1', 'a2', 'a3']

list2 = ['b1', 'b2', 'b3']

for index, (item1, item2) in enumerate(zip(list1, list2)):

print(index, item1, item2)

выводит:

0 a1 b1

1 a2 b2

2 a3 b3

**Тема урока: файловый ввод и вывод**

1. Понятие файла
2. Файловый ввод и вывод
3. Работа с файлами
4. Типы файлов
5. Методы доступа к файлам
6. Имена файлов
7. Кодировка файлов

**Аннотация.** Урок посвящен основным понятиям, связанным с файлами.

**Файловый ввод и вывод**

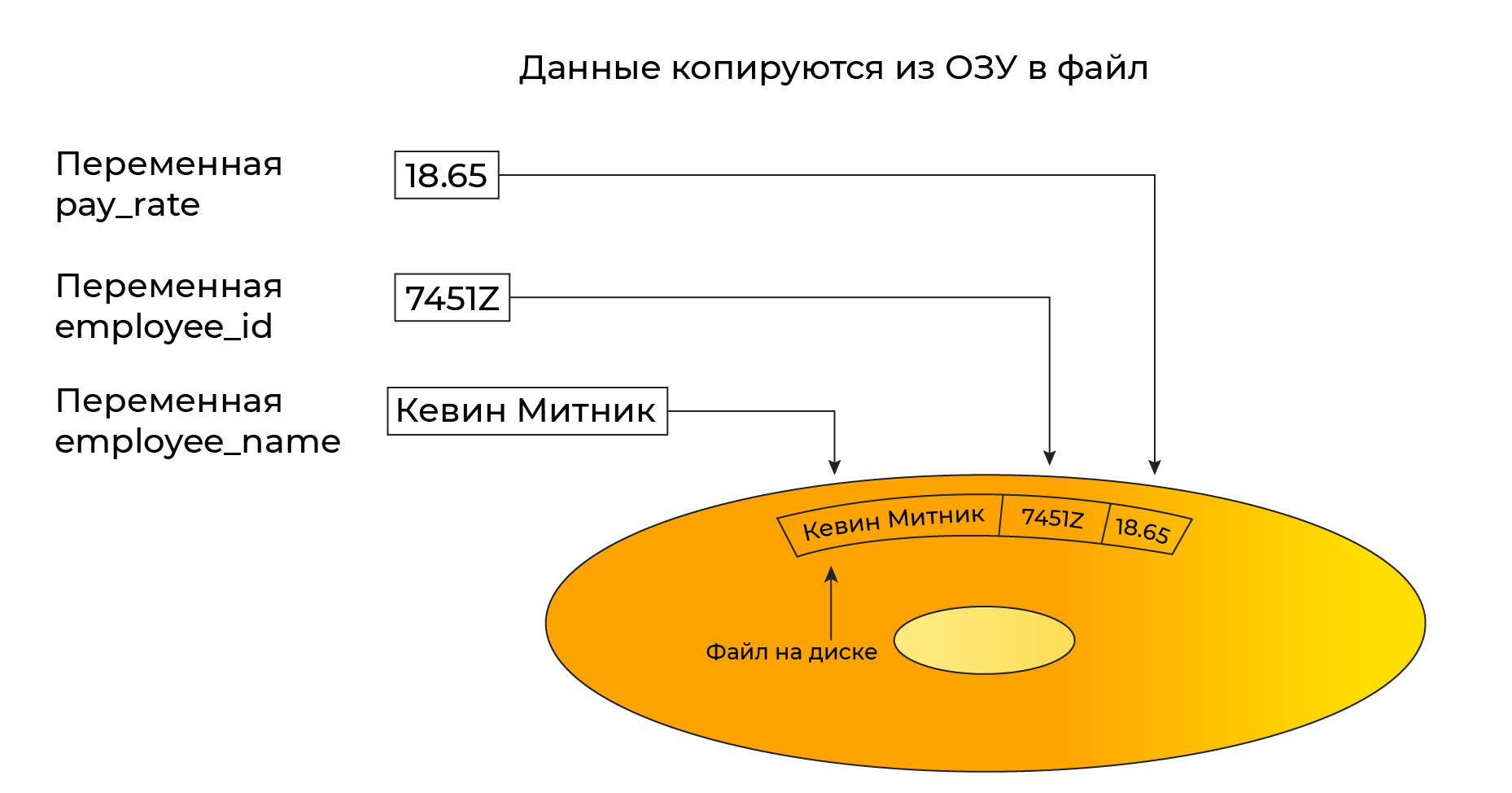
Программы, которые мы писали до сих пор, требовали повторного ввода данных при каждом запуске, потому что, как только программа заканчивает свою работу, данные для переменных исчезают из оперативной памяти. Если их нужно сберечь между выполнениями программы, требуется запись. Данные записываются в файл, обычно сохраняющийся на диске компьютера.

   Файл (англ. *file*) — именованная область данных на носителе информации.

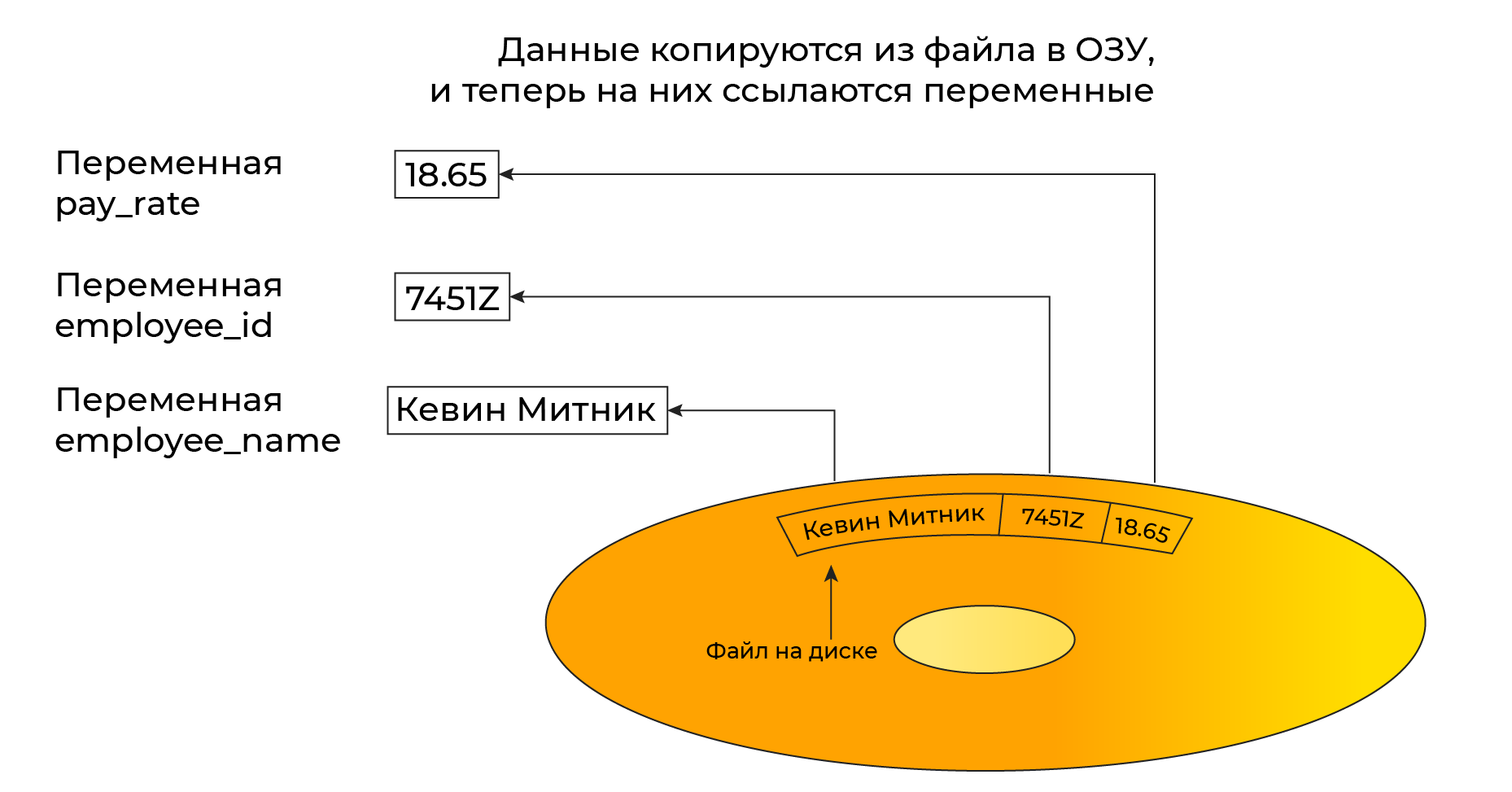
Сохраненные в файле данные обычно остаются в нем после завершения работы программы, их можно позже извлечь и использовать.

Когда программе нужно сохранить данные для дальнейшего использования, она пишет их в файл. Из файла записанные данные можно считать.

Программисты называют такой процесс сохранения данных — **запись данных** в файл. Когда часть данных пишется в файл, она копируется из переменной, находящейся в оперативной памяти. Файл, куда сохраняются данные, называется **файл вывода**, потому что программа помещает в него выходные данные.



Процесс извлечения данных из файла называется **чтением данных** из файла. Данные считываются из **файла ввода**. Программа извлекает входные данные из этого файла. Когда порция данных считывается из файла, она копируется в оперативную память, где на нее ссылается переменная.



**Работа с файлами**

Когда в программе используется файл, как правило требуется выполнить три шага:

1. **Открыть файл**. В процессе открытия файла создается связь между файлом и программой. Открытие файла вывода обычно создает файл на диске и позволяет программе записать в него данные. Открытие файла ввода позволяет программе прочитать данные из файла.
2. **Обработать файл.** На этом шаге данные либо записываются в файл (если это файл вывода), либо считываются из файла (если это файл ввода).
3. **Закрыть файл.** После использования файла программой **его нужно закрыть**, тем самым освободить ресурс и разорвать связь файла с программой.

**Типы файлов**

Существует два типа файлов: **текстовые** и **двоичные (бинарные)**. Текстовый файл содержит данные, которые были закодированы в виде текста при помощи такой схемы кодирования, как ASCII или Юникод. Даже если файл содержит числа, эти числа в файле хранятся как набор символов. В результате файл можно открыть и просмотреть в текстовом редакторе, таком как Блокнот.

   Рекомендуем использовать текстовый редактор [Notepad++](https://notepad-plus-plus.org/downloads/).

Двоичный файл содержит данные, которые не были преобразованы в текст. Данные, которые помещены в двоичный файл, предназначены только для чтения программой, и такой файл невозможно просмотреть в текстовом редакторе.

Python позволяет работать и с текстовыми, и с двоичными файлами, но в рамках этого курса мы будем работать с текстовыми файлами, чтобы вы могли использовать текстовый редактор для исследования файлов, создаваемых вашими программами.

   Разделение файлов на текстовые и бинарные искусственное, так как любой текстовый файл бинарен.

**Методы доступа к файлам**

Большинство языков программирования обеспечивает два способа получения доступа к данным в файле:

* **последовательный,**
* **прямой**или **произвольный**.

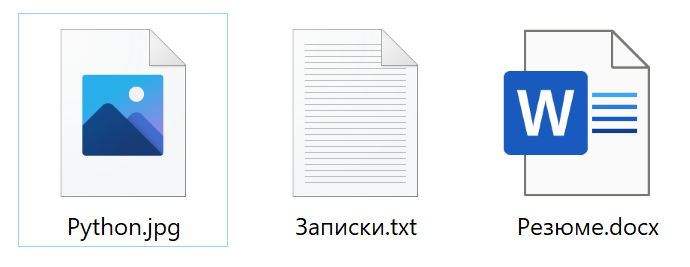
Последовательный, как при проигрывании кассет с записью на пленке, выдает порции информации одну за другой. При работе с таким файлом не получится перескочить сразу к нужной части данных, сначала придется прочитать все предыдущие.

При работе с файлом с прямым или произвольным доступом можно перескочить непосредственно к любой порции данных, не читая предыдущие. Как проигрыватель компакт-дисков или МР3-плеер перескакивает сразу к любой песне.

**Имена файлов**

Большинство пользователей компьютеров привыкли, что файлы определяются по имени. Когда создаете документ с помощью текстового редактора и сохраняете его в файле, указываете имя файла. Если исследуете содержимое диска с помощью **проводника Windows**, видите список имен файлов.

На рисунке показано, как в Windows могут выглядеть значки файлов с именами Python.jpg, Записки.tхt и Резюме.dосх.



У каждой операционной системы собственные правила именования файлов. Многие системы поддерживают использование расширений файлов, т.е. коротких последовательностей символов, которые расположены в конце имени файла и предваряются точкой. Файлы, изображенные на рисунке, имеют расширения jpg, txt и docx. Расширение обычно говорит о типе данных в файле. Например, расширение jpg сообщает, что файл содержит графическое изображение, сжатое согласно стандарту JPEG. Расширение txt свидетельствует, что в файле текст. Расширение docx информирует о наличии в файле документа Microsoft Word.

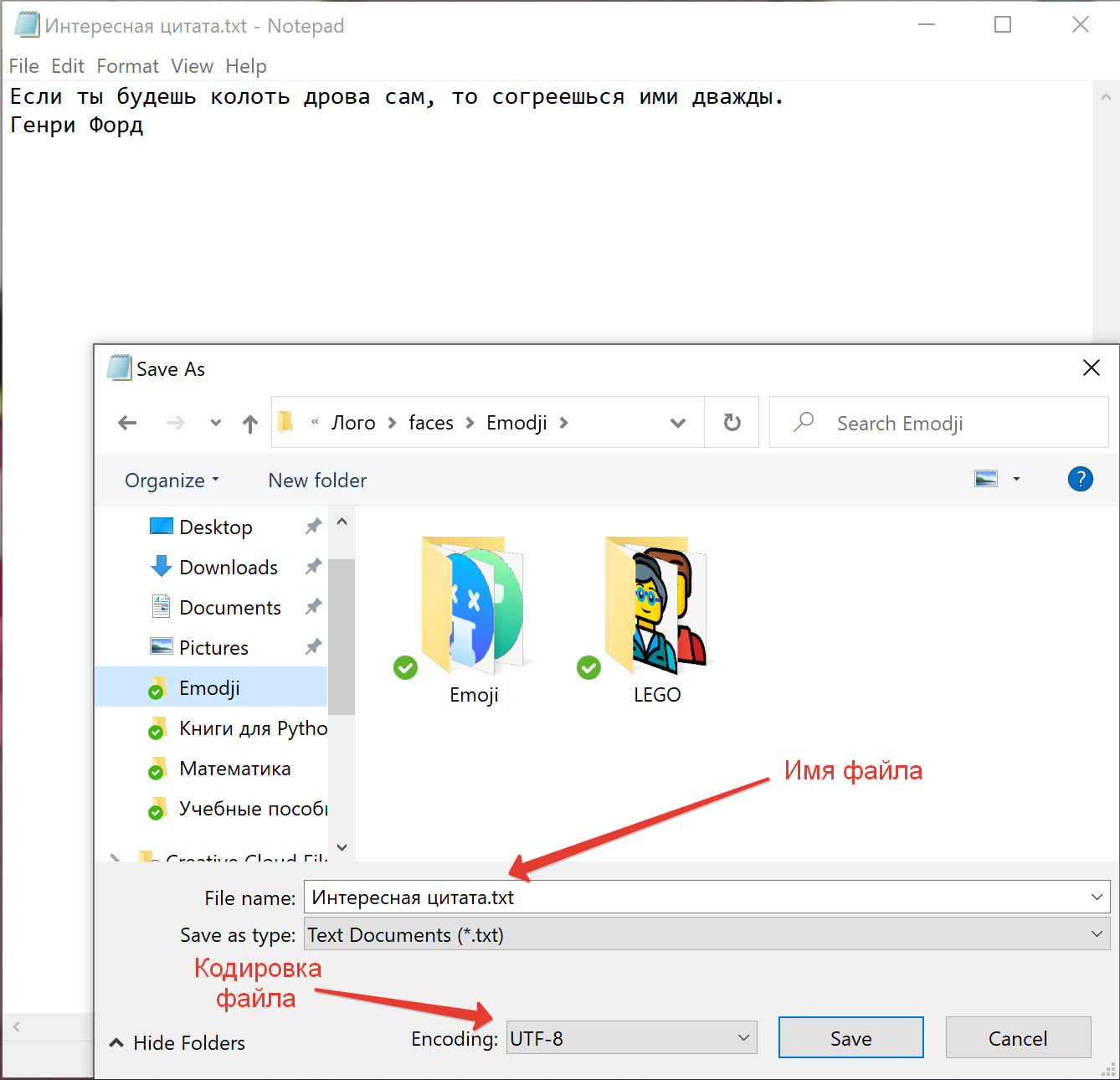
**Кодировка файлов**

В [прошлом курсе](https://stepik.org/lesson/313439/step/1?unit=295959) мы говорили о том, как строки текста хранятся в памяти компьютера, таблицах символов ASCII и Юникод, а также о кодировке UTF-8.

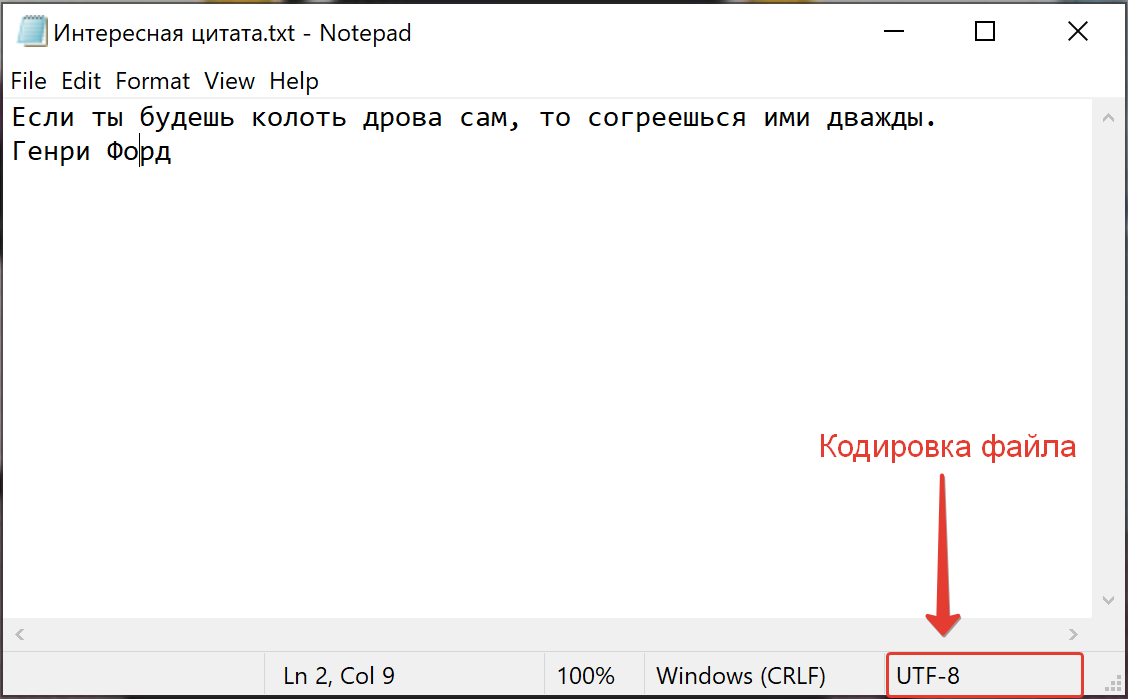
Кодировка UTF-8 самая распространенная, рекомендуем использовать именно ее в качестве кодировки по умолчанию для текстовых файлов.

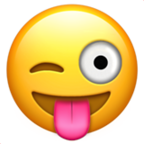
UTF-8 — сложная кодировка, на обозначение одного символа в ней может использоваться от одного до шести байт. Подробнее про эту кодировку можно почитать в [википедии](https://ru.wikipedia.org/wiki/UTF-8).

В операционной системе Windows до сих пор используется однобайтовая кодировка Windows-1251 (😨). Чтобы избежать проблем при работе с текстовыми файлами в Windows нужно явно указывать кодировку. В редакторе Notepad («Блокнот») можно указывать кодировку при сохранении файла.

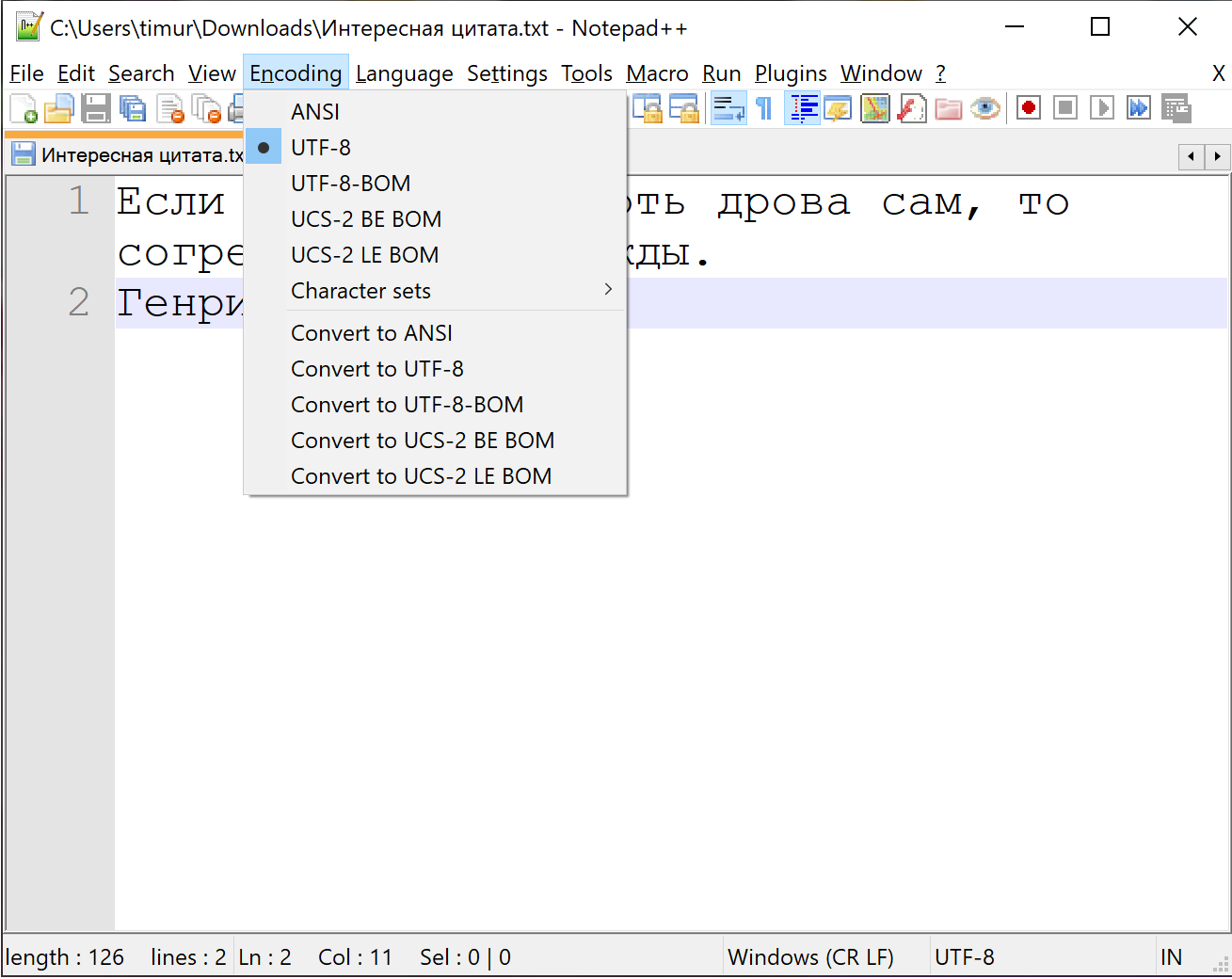


Если после такого сохранения открыть файл Интересная цитата.txt, то мы увидим нужную кодировку.



При работе с ОС Linux и MacOS таких проблем не возникает вовсе, поскольку в них кодировка UTF-8 применяется по умолчанию.

В редакторе Notepad++ кодировка файла устанавливается так:



**Относительные и абсолютные пути**

Путь файла (или путь к файлу) — последовательное указание имен папок, через которые надо пройти, чтобы добраться до объекта.

Пути к файлу бывают двух типов:

* абсолютные;
* относительные.

**Абсолютный путь** – полный путь к файлу, показывающий **точное место** его расположения. Он всегда один и тот же, пока файл не перемещен.

Примеры абсолютного пути:

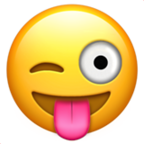
* D:\Data\MyFiles\picture.png;
* С:\MyPrograms\Python\script.py;
* C:\Users\timur\YandexDisk\BeeGeek\Python\book.pdf.

Указывая абсолютный путь на компьютере, обязательно нужно указывать диск, а также использовать \ (для Windows) в качестве разделителя имен папок.

В unix-подобных ОС, например, в Linux и Mac OS для отделения имен папок используется прямой слеш /, а не обратный, как в Windows.

**Относительный путь** – привязан к какой-либо "отправной точке" и указан по отношению к ней.

Например, у нас есть картинка picture.png, которая хранится на диске D. Абсолютный путь к ней будет D:\Data\MyFiles\picture.png, а относительно папки Data можно указывать MyFiles\picture.png.

Абсолютный путь показывает точное местонахождение файла, а относительный показывает путь к файлу от какой-либо "отправной точки".

**Примечания**

**Примечание 1.** Конкретная физическая организация файлов, их группировка по папкам, устройство процедур доступа к информации, механизмы кеширования очень сильно зависят от операционной системы и применяемой в ней файловой системы. Как правило, при работе с файлами прикладные программисты работают на верхнем уровне **абстракции**.

**Примечание 2.** Файлы обычно располагаются на носителях, работающих медленнее, чем оперативная память. Поэтому работа с ними идет в буферизированном режиме. Даже если вы запросите один байт из файла, считается целый блок (до нескольких килобайт). Он переместится в буфер оперативной памяти. Дальше файл читается оттуда, поскольку это быстрее и экономит ресурсы чтения/записи внешних носителей.

**Примечание 3.** Примеры однобайтовых кодировок:

* [Windows-1251](https://ru.wikipedia.org/wiki/Windows-1251);
* [cp-866](https://ru.wikipedia.org/wiki/CP866);
* [КОИ-8](https://ru.wikipedia.org/wiki/%D0%9A%D0%9E%D0%98-8).

**Примечание 4.** Прочитать про относительные и абсолютные пути можно [тут](http://phpfaq.ru/newbie/paths) и [тут](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D1%82%D1%8C_%D0%BA_%D1%84%D0%B0%D0%B9%D0%BB%D1%83).

# Тема урока: работа с текстовыми файлами, чтение данных

1. Открытие и закрытие файлов
2. Указание кодировки
3. Чтение данных из файла

**Аннотация.** Урок посвящен работе с файлами. Изучим функцию open() и метод close(), узнаем как задавать правильную кодировку и научимся читать информацию с помощью методов read(), readline(), readlines().

## Работа с файлами в Python

Высокоуровневый язык программирования Python предоставляет своим пользователям массу полезных средств для взаимодействия с файлами. Встроенные функции и методы позволяют создавать файлы, читать из них данные, а также всячески манипулировать их содержимым.

### Открытие файла

Для открытия файлов в Python существует функция open(). Она создает файловый объект и связывает его с файлом на диске. Общий формат применения функции open():

файловая\_переменная = open(имя\_файла, режим\_доступа)

Здесь

* файловая переменная – имя переменной, которая ссылается на файловый объект;
* имя\_файла – строковый литерал, задающий имя файла;
* режим\_доступа – строковый литерал, задающий режим доступа (чтение, запись, и т.д.), в котором файл будет открыт.

Ниже представлены строковые литералы (символы), используемые для задания режима доступа.

| **Стр. литерал** | **Режим** | **Описание** |
| --- | --- | --- |
| 'r' | Read (чтение) | Открыть файл только для чтения. Такой файл не может быть изменен. |
| 'w' | Write (запись) | Открыть файл для записи. Если файл уже существует, стереть его содержимое. Если файл не существует, он будет создан. |
| 'a' | Append (добавление) | Открыть файл для записи. Данные будут добавлены в конец файла. Если файл не существует, он будет создан. |
| 'r+' | Read + Write | Открыть файл для чтения и записи. В этом режиме происходит частичная перезапись содержимого файла. |
| 'x' | Create (создание) | Создать новый файл. Если файл существует, произойдет ошибка. |

Предположим, файл students.txt содержит данные о студентах, и мы хотим открыть его для чтения.

Это можно сделать с помощью строки кода:

student\_file = open('students.txt', 'r')

По умолчанию режим доступа (второй аргумент функции open()) определен для чтения (литерал 'r'), поэтому файл students.txt можно открыть для чтения так:

student\_file = open('students.txt') # по умолчанию режим доступа для чтения ('r')

В результате исполнения этой инструкции будет открыт файл students.txt и переменная student\_file будет ссылаться на файловый объект, который можно использовать для чтения данных из файла.

Обратите внимание, что в переменную student\_file в примере выше не попадает содержимое файла students.txt. Фактически это ссылка на файл, ее еще называют **дескриптор файла**.

Предположим, надо создать файл с именем sales.txt и записать в него данные о продажах. Это можно сделать с помощью строки кода:

sales\_file = open('sales.txt', 'w')

После исполнения этого кода будет создан файл sales.txt и переменная sales\_file будет ссылаться на файловый объект, который можно использовать для записи в него данных.

### Указание места расположения файла

Когда в функцию open() передается имя файла без указания пути, интерпретатор Python исходит из предположения, что место расположения файла то же, что у исполняемой программы. Например, программа расположена в папке C:\Users\Documents\Python. Если программа выполняется и исполняет инструкцию:

customer\_file = open('customers.txt', 'r')

то файл customer.txt программа станет искать в папке C:\Users\Documents\Python.

Аналогично, если программа выполняется, и она исполняет инструкцию:

sales\_file = open('sales.txt', 'w')

то файл sales.txt создается в той же папке.

Если имя файла не содержит путь, то используется относительный путь, относительно папки, где находится исполняемая программа.

Если требуется открыть файл, расположенный в другом месте, нужно указать путь и имя файла в аргументе, передаваемом в функцию open().

Приведенный ниже код создает файл test.txt в папке C:\Users\temp:

test\_file = open('C:\\Users\\temp\\test.txt', 'w')

Обратите внимание: символ \ является специальным символом в Python и его нужно экранировать (\\), чтобы интерпретатор Python рассматривал обратную косую черту как обычный символ.

Вместо экранирования символов можно использовать сырые строки (raw strings). Для этого следует снабдить строковый литерал префиксом в виде буквы r.

test\_file = open(r'C:\Users\temp\test.txt', 'w')

Приведенный выше код создает файл test.txt в папке C:\Users\temp. Префикс r указывает на то, что строковый литерал является сырым (неформатированным).

Механизм сырых строк очень удобен не только при работе с файлами.

Приведенный ниже код:

path = 'C:\new\text.txt'

print(path)

выводит:

C:

ew ext.txt

поскольку символы \n и \t интерпретируются как перенос строки и табуляция.

Приведенный ниже код:

path = r'C:\new\text.txt'

print(path)

выводит содержимое строки path:

C:\new\text.txt

Чтобы сделать работу с файлами универсальнее, в путях файлов в Windows в Python-программах рекомендуется ставить прямой слеш (/). В наших примерах мы так и будем делать:

file1 = open(r'C:/Users/temp/test.txt') # используем прямой слеш / (абсолютный путь)

file2 = open(r'temp/data.txt') # используем прямой слеш / (относительный путь)

### Кодировка

Открыть файл, содержащий только латиницу и цифры, можно так:

file = open('info.txt', 'r')

При работе с текстом на русском языке нужно указать кодировку, для этого служит параметр encoding:

file = open('info.txt', 'r', encoding='utf-8')

   Указание кодировки при открытии файла – хороший тон. Рекомендуем придерживаться этого правила.

Чтобы получить кодировку открытого файла используют файловое свойство encoding.

Приведенный ниже код:

file1 = open('students.txt', 'w')

file2 = open('customers.txt', 'w', encoding='utf-8')

print(file1.encoding)

print(file2.encoding)

file1.close()

file2.close()

выводит на компьютере с операционной системой Windows:

cp1252

utf-8

### Закрытие файлов

После окончания работы с файлом его необходимо закрыть. Для этого есть несколько причин:

* если файл изменялся, это позволит корректно его сохранить;
* если открытый файл потребуется другим программам, ваша программа может его блокировать;
* не стоит держать в памяти лишние, уже не нужные, данные;
* удалить открытый кем-то файл проблематично.

Для закрытия файла используется файловый метод close():

file = open('info.txt', 'r') # открываем файл с именем info.txt для чтения

# работаем с содержимым файла info.txt

file.close() # закрываем файл после окончания работы

Чтобы проверить открыт файл или закрыт можно использовать файловое свойство (атрибут) closed.

Приведенный ниже код:

file1 = open('students.txt', 'w')

file2 = open('customers.txt', 'w')

file1.close()

print(file1.closed)

print(file2.closed)

file2.close()

выводит:

True

False

Обратите внимание на то, что при вызове метода мы используем скобки: close(), а при вызове свойства (атрибута) скобок нет closed. Методы совершают действия, а свойства возвращают информацию о объекте (привет ООП 😉).

## Чтение содержимого файла

Как уже сказано, при открытии файла с помощью функции open() в файловую переменную попадает не содержимое файла, а ссылка на файл (дескриптор файла).

Приведенный ниже код:

file = open('info.txt', 'w', encoding='utf-8') # открываем файл для записи

print(file)

выводит:

<\_io.TextIOWrapper name='info.txt' mode='w' encoding='utf-8'>

Для чтения содержимого открытого для чтения файла используются три файловых метода:

* read() – читает все содержимое файла;
* readline() – читает одну строку из файла;
* readlines() – читает все содержимое файла и возвращает список строк.

Предположим, в папке с исполняемой программой есть текстовый файл languages.txt с содержимым:

Python

Java

Javascript

C#

C

C++

PHP

R

Objective-C

### Метод read()

Файловый метод read() считывает все содержимое из файла и возвращает строку, которая может содержать символы перехода на новую строку '\n'.

Приведенный ниже код:

file = open('languages.txt', 'r', encoding='utf-8')

content = file.read()

file.close()

считывает содержимое файла languages.txt в переменную content. В переменной content будет содержаться строка 'Python\nJava\nJavascript\nC#\nC\nC++\nPHP\nR\nObjective-C'.

Таким образом метод read() считывает все содержимое файла, **включая переносы строк**:

Если методу read() передать целочисленный параметр, то будет считано не более заданного количества символов. Например, считывать файл посимвольно можно при помощи метода read(1).

### Метод readline()

Файловый метод readline() считывает одну строку из файла (до символа конца строки '\n'), при этом возвращается считанная строка вместе с символом '\n'. Если считать строку не удалось – достигнут конец файла и больше строк в нем нет, возвращается пустая строка.

Приведенный ниже код:

file = open('languages.txt', 'r', encoding='utf-8')

language = file.readline()

file.close()

считывает содержимое первой строки файла languages.txt в переменную language. В переменной language будет содержаться строка 'Python\n'.

Для удаления символа '\n' из конца считанной строки удобно использовать строковый метод rstrip().

Приведенный ниже код:

line = 'Python\n'

line = line.rstrip()

удаляет символ \n из содержимого строки line, в результате чего в переменной line содержится строка 'Python'.

   Если вдруг вы забыли о строковых методах, освежить знания можно [тут](https://stepik.org/lesson/296416/step/1?unit=278136), [тут](https://stepik.org/lesson/303083/step/1?unit=284990) и [тут](https://stepik.org/lesson/303084/step/1?unit=284991) 🥰.

Прочитать содержимое всего файла построчно можно двумя способами.

**С помощью цикла while:**

file = open('languages.txt', 'r', encoding='utf-8')

line = file.readline() # считываем первую строку

while line != '': # пока не конец файла

print(line.strip()) # обрабатываем считанную строку

line = file.readline() # читаем новую строку

file.close()

**С помощью цикла for (предпочтительный вариант):**

file = open('languages.txt', 'r', encoding='utf-8')

for line in file:

print(line.strip())

file.close()

Метод readline() довольно удобен, когда мы хотим управлять процессом чтения из файла, особенно если файл очень большой и его полное считывание может привести к нехватке памяти.

### Метод readlines()

Файловый метод readlines() считывает все строки из файла и возвращает список из всех считанных строк (одна строка — один элемент списка). При этом, каждая строка в списке заканчивается символом переноса строки  '\n'😅.

Приведенный ниже код:

file = open('languages.txt', 'r', encoding='utf-8')

languages = file.readlines()

file.close()

считывает содержимое файла languages.txt в переменную languages. В переменной languages будет содержаться список:

['Python\n', 'Java\n', 'Javascript\n', 'C#\n', 'C\n', 'C++\n', 'PHP\n', 'R\n', 'Objective-C']

Чтобы удалить символ '\n' можно использовать списочное выражение:

languages = [line.strip() for line in file.readlines()]

либо функцию map()🤩:

languages = list(map(str.strip, file.readlines()))

либо анонимную функцию:

languages = list(map(lambda line: line.strip(), file.readlines()))

Если передать в функцию list() ссылку на файловый объект list(file) , получим тот же результат, что при вызове метода file.readlines().

## Примечания

**Примечание 1.** Язык Python позволяет работать не только с текстовыми, но и с бинарными файлами. Ниже представлены строковые литералы, которые можно использовать для задания режима обработки файла.

| **Символ** | **Режим** | **Описание** |
| --- | --- | --- |
| 't' | Текстовый режим (значение по умолчанию) | Работа с текстовым файлом |
| 'b' | Бинарный режим | Работа с бинарными файлами (картинки, звук и т.д.) |

Режим обработки файла указывается **после** режима доступа к файлу.

Приведенный ниже код открывает файл file.dat в режиме чтения как бинарный файл.

file = open('file.dat', 'rb')

По умолчанию функция open() использует литерал 'rt', то есть файл открывается для чтения в текстовом  режиме.

Таким образом, открыть текстовый файл для чтения можно так open('info.txt') или так open('info.txt', 'r') или так open('info.txt', 'rt').

**Примечание 2.** Так как Python — язык с автоматическим управлением памятью, все файлы закрываются автоматически после успешного завершения программы или когда удаляется последняя ссылка на файловый объект. Но важно все равно закрывать файл, как только он перестает быть нужным. Это помогает избегать конфликтов совместного доступа и риска получить испорченный файл, если программа завершится аварийно.

Файл должен быть закрыт сразу после того, как все нужное из него прочитано или в него записано.

**Примечание 3.** Еще раз обратите внимание на то, что в путях до файла используются прямые слеши /. Можно использовать и обратные, но тогда их придется экранировать либо применять модификатор сырой строки r. Кроме того, в unix-подобных операционных системах принято использовать именно прямой слеш.

**Примечание 4.** Существуют специальные символы:

* \n – перемещает позицию печати на одну строку вниз;
* \r – перемещает позицию печати в крайнее левое положение строки.

Приведенный ниже код:

print('aaaaaa\nbb')

выводит:

aaaaaa

bb

 Приведенный ниже код:

print('aaaaaa\rbb')

выводит:

bbaaaa

Приведенный ниже код:

print(ord('\n'))

print(ord('\r'))

выводит:

10

13

**Примечание 5.** После того, как файл (file) открыт, можно получить различную относящуюся к нему информацию. Три полезных атрибута (свойства):

| **Атрибут (свойство)** | **Описание** |
| --- | --- |
| file.closed | возвращает истину (True), если файл закрыт, иначе возвращает ложь (False) |
| file.mode | возвращает режим доступа с помощью которого был открыт файл |
| file.name | возвращает имя файла |

# Тема урока: работа с текстовыми файлами

1. Позиция (курсор) в файле
2. Менеджеры контекста

**Аннотация.** Урок посвящен работе с файлами: позиция считывания в файле, методы seek(), tell(), а также работа с менеджером контекста.

## Позиция в файле

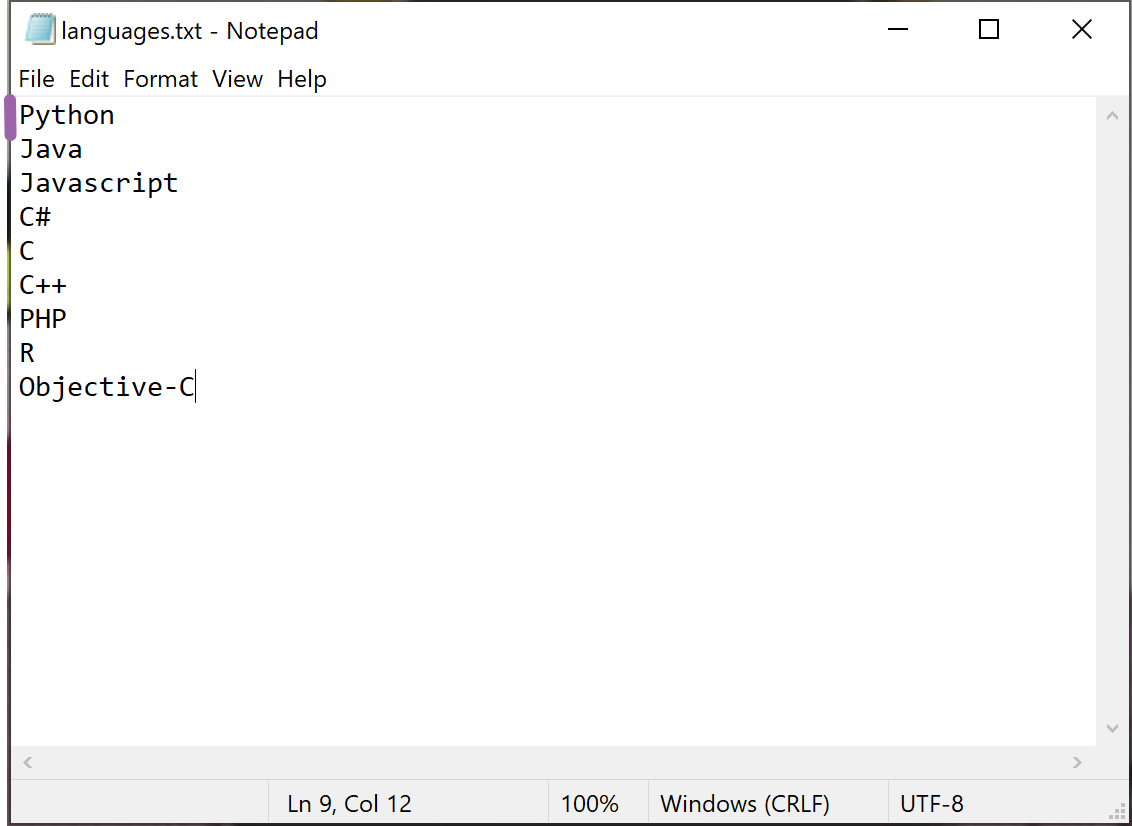
Когда мы читаем текст из файла с помощью методов read() или readlines() происходит перемещение **текущей позиции** в конец файла. При использовании метода readline() текущая позиция перемещается на следующую строку файла.

При открытии файла текущая позиция всегда равна нулю – указывает на первый символ текста. При прочтении файла до конца с помощью вызова методов read(), readlines() позиция перемещается в конец файла и последующие чтения ничего не дают.

Вызов методов read(), readlines(), readline() перемещает текущую позицию туда, где завершилось чтение. Для методов read() и readlines() это конец файла, для метода readline() – следующая строка после прочитанной.

Текущую позицию обычно называют "**курсор"**.

Предположим, у нас есть файл languages.txt. Когда мы его открываем, курсор находится в начале файла, в нулевой позиции, это выглядит примерно так:



Если мы считаем две строки с помощью метода readline():

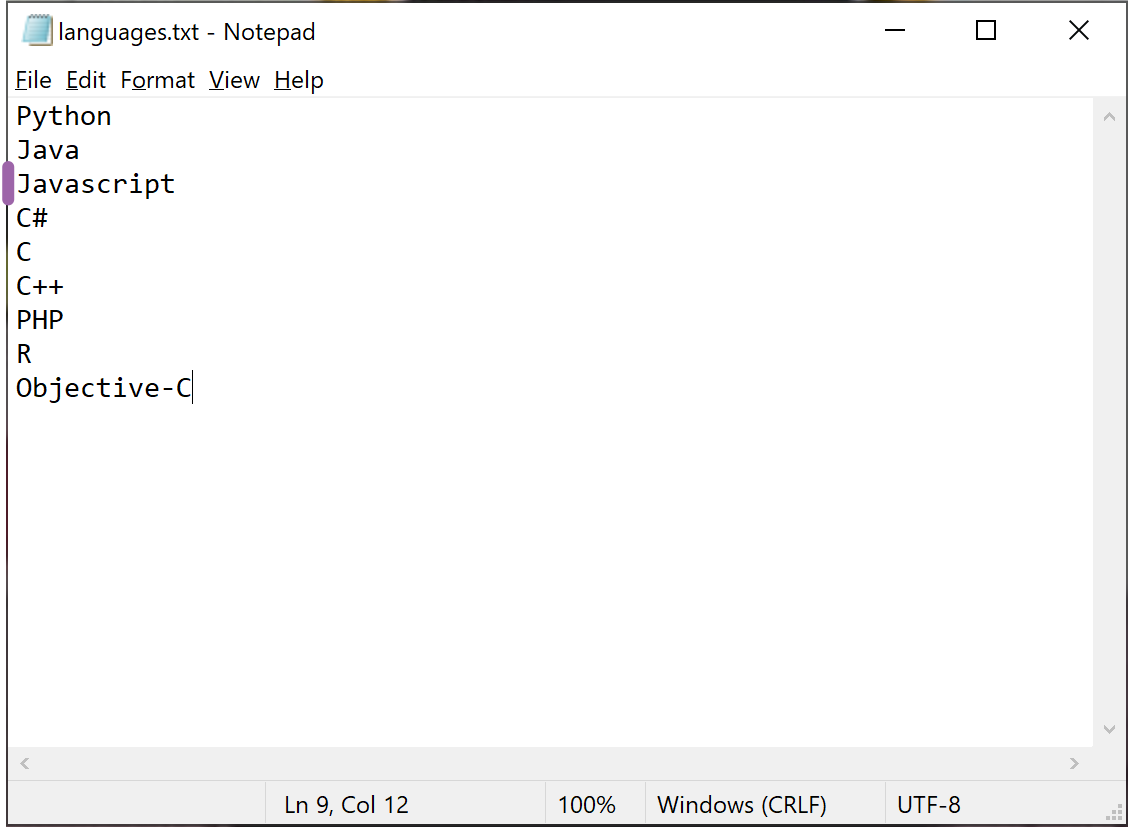
file = open('languages.txt', 'r', encoding='utf-8')

line1 = file.readline()

line2 = file.readline()

file.close()

курсор переместится в начало третьей строки:



Чтение всегда происходит слева направо от курсора. Таким образом, если после двух вызовов метода readline() вызвать метод read(), он считает не весь файл, а только оставшиеся строки:

file = open('languages.txt', 'r', encoding='utf-8')

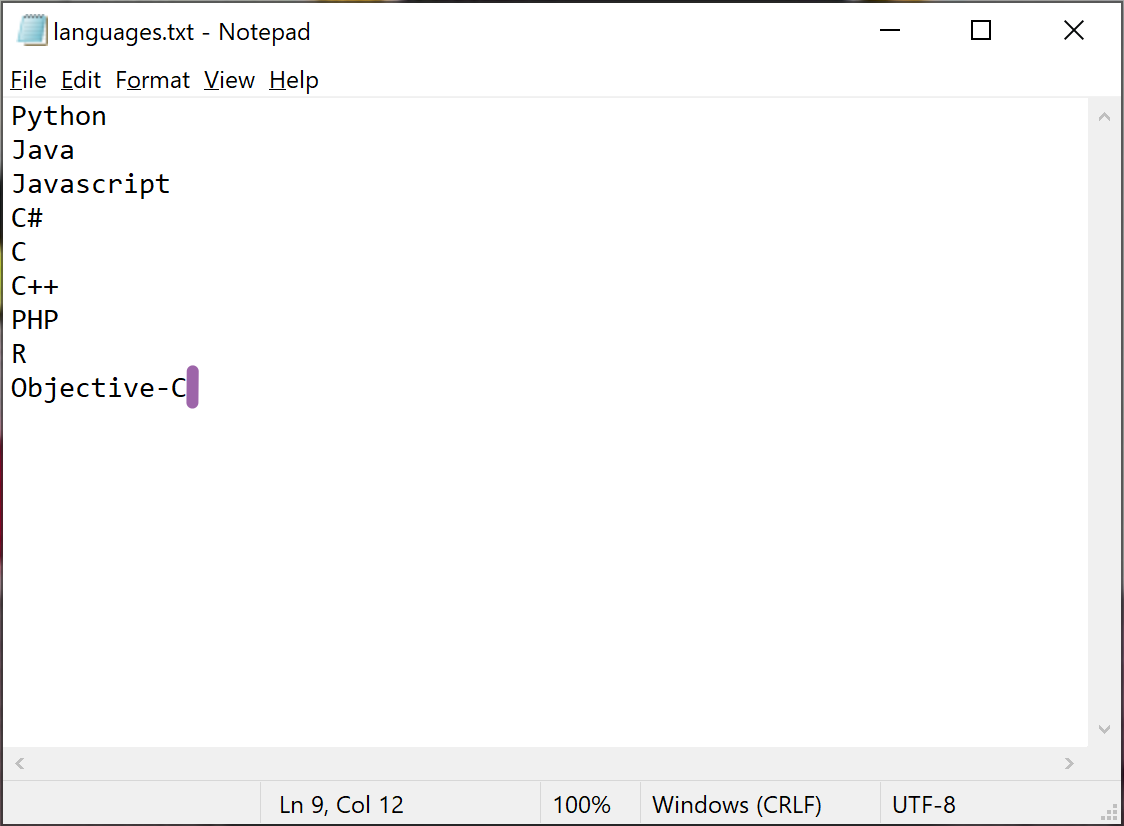
line1 = file.readline()

line2 = file.readline()

remaining\_lines = file.read() # считывание начинается с 3 строки до конца файла

file.close()

После того, как мы считали все строки файла, курсор находится в конце.



После завершения чтения мы больше не можем считать ни одного символа из файла. Все последующие вызовы методов read() или readline() будут приводить к считыванию пустой строки.

Для повторного чтения данных из файла, можно:

* переоткрыть файл, тогда курсор снова попадёт в начало;
* переместить курсор с помощью файлового метода seek().

### Файловый метод seek()

Файловый метод seek() задаёт позицию курсора в байтах от начала файла. Чтобы перевести курсор в самое начало файла необходимо вызвать метод seek(), передав ему в качестве аргумента значение 00.

Приведенный ниже код:

file = open('languages.txt', 'r', encoding='utf-8')

line1 = file.readline()

file.seek(0) # переводим курсор в самое начало

line2 = file.readline()

print(line1, line2)

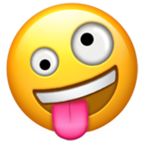
file.close()

выводит:

Python

Python

Метод seek() **не очень** полезен при работе с текстовыми файлами, так как не учитывает разделение текста на строки. А вот при работе с файлами в двоичном режиме умение работать с позицией и смещениями очень важно!

Будьте аккуратны с символами, использующими более 11 байта (кириллица в кодировке utf-8), обращение к "промежуточному" байту может вызвать ошибку.

Если метод seek() устанавливает курсор (текущую позицию), то метод tell() получает ее.

Приведенный ниже код:

file = open('languages.txt', 'r', encoding='utf-8')

print(file.tell())

line1 = file.readline()

print(file.tell())

file.close()

выводит:

0

8

В самом начале курсор (текущая позиция) равен нулю, после считывания первой строки, курсор смещается на 88 байт (по байту на каждый из символов 'P', 'y', 't', 'h', 'o', 'n' и два байта на символ перевода строки '\n').

## Менеджер контекста

Как уже сказано, важно своевременно закрывать файлы с помощью метода close(). Закрытие файлов вручную, а также отдача закрытия на откуп среде исполнения, обладают существенным недостатком: если между открытием файла и его закрытием произойдёт ошибка, в лучшем случае файл окажется открыт слишком долго, а в худшем случае часть данных не сохранится.

Хочется иметь возможность автоматически закрывать файл сразу после окончания работы с ним и осуществлять закрытие даже при возникновении ошибки. Файловые объекты уже умеют работать в таком режиме, но для этого их нужно использовать как **менеджеры контекста**.

Менеджер контекста — объект, реализующий одноименный протокол. Объекты, реализующие этот протокол, позволяют использовать следующий специальный синтаксис:

with object as name:

# Здесь нам доступен ресурс name.

# Это тело with-блока.

# А здесь ресурс name уже освобождён, даже если в теле with-блока произошла ошибка.

Весь код в теле with-блока работает "в контексте". Чаще всего контекст подразумевает выделение некоего ресурса, например, файла. По выходу из контекста ресурс автоматически освобождается, даже если при выполнении блока возникло исключение.

Как только закончится код, оформленный с отступами в with (аналогичные отступы в циклах или функциях), это будет означать, что контекст закончился, и Python автоматически закроет файл.

Приведенный ниже код:

file = open('languages.txt', 'r', encoding='utf-8')

for line in file:

print(line)

file.close() # ручное закрытие файла

print('Файл закрыт')

можно переписать в виде:

with open('languages.txt', 'r', encoding='utf-8') as file:

for line in file:

print(line)

# автоматическое закрытие файла

print('Файл закрыт')

Обратите внимание: при использовании менеджера контекста не требуется использовать метод close().

   При работе с файлами желательно всегда использовать менеджер контекста. Это делает программу надежнее.

## Примечания

**Примечание 1.** Подробнее о файловом методе seek() можно почитать в [документации](https://docs.python.org/3/library/io.html#io.TextIOBase.seek).

**Примечание 2.**В современных операционных системах файловый ввод-вывод устроен достаточно сложно. Для обеспечения максимального быстродействия чтения и записи в файлы, а также контроля безопасности этого процесса, большинство операционных систем не позволяют программам напрямую работать с диском. Операционная система предоставляет программам специальные объекты — файловые дескрипторы (функция open() возвращает как раз файловый дескриптор). Имея файловый дескриптор, можно записывать и читать данные, не задумываясь о файловой системе. Файловые дескрипторы удобны, но на создание каждого расходуется достаточно большое количество ресурсов. Поэтому у операционной системы есть общий лимит на количество одновременно использующихся файловых дескрипторов. И при этом каждая программа имеет свой собственный лимит. Как только программа исчерпает доступное ей количество дескрипторов, следующая попытка открыть очередной файл закончится с ошибкой. Программисту важно следить за тем, сколько файлов программа открывает в каждый момент и закрывает ли она их своевременно. Используйте менеджер контекста with и жизнь станет проще 😋.

**Примечание 3.** С помощью менеджера контекста можно работать с несколькими файлами.

with open('input.txt', 'r') as input\_file, open('output.txt', 'w') as output\_file:

# обработка файлов

# Тема урока: работа с текстовыми файлами

1. Запись данных в файл
2. Файловые методы write() и writelines()

**Аннотация.** Урок посвящен работе с файлами. Мы изучим методы записи в файл write() и writelines().

## Запись данных в файлы

Файлы могут быть открыты для чтения, а могут для записи данных. Ниже представлены строковые литералы для задания режима доступа.

| **Стр. литерал** | **Режим** | **Описание** |
| --- | --- | --- |
| 'r' | Read (чтение) | Открыть файл только для чтения. Такой файл не может быть изменен. |
| 'w' | Write (запись) | Открыть файл для записи. Если файл уже существует, то стереть его содержимое. Если файл не существует, он будет создан. |
| 'a' | Append (добавление) | Открыть файл для записи. Все записываемые в файл данные будут добавлены в его конец. Если файл не существует, то он будет создан. |
| 'r+' | Read + Write | Открыть файл для чтения и записи. В этом режиме происходит частичная перезапись содержимого файла с самого начала. |
| 'x' | Create (создание) | Создать новый файл. Если файл уже существует, произойдет ошибка. |

Для записи используются два файловых метода:

* write() – записывает переданную строку в файл;
* writelines() – записывает переданный список строк в файл.

### Метод write()

Общий формат применения файлового метода write():

файловая\_переменная.writе(строковое\_значение)

Здесь

* файловая переменная – это имя переменной, которая ссылается на файловый объект;
* строковое значение – это символьная последовательность, которая будет записана в файл.

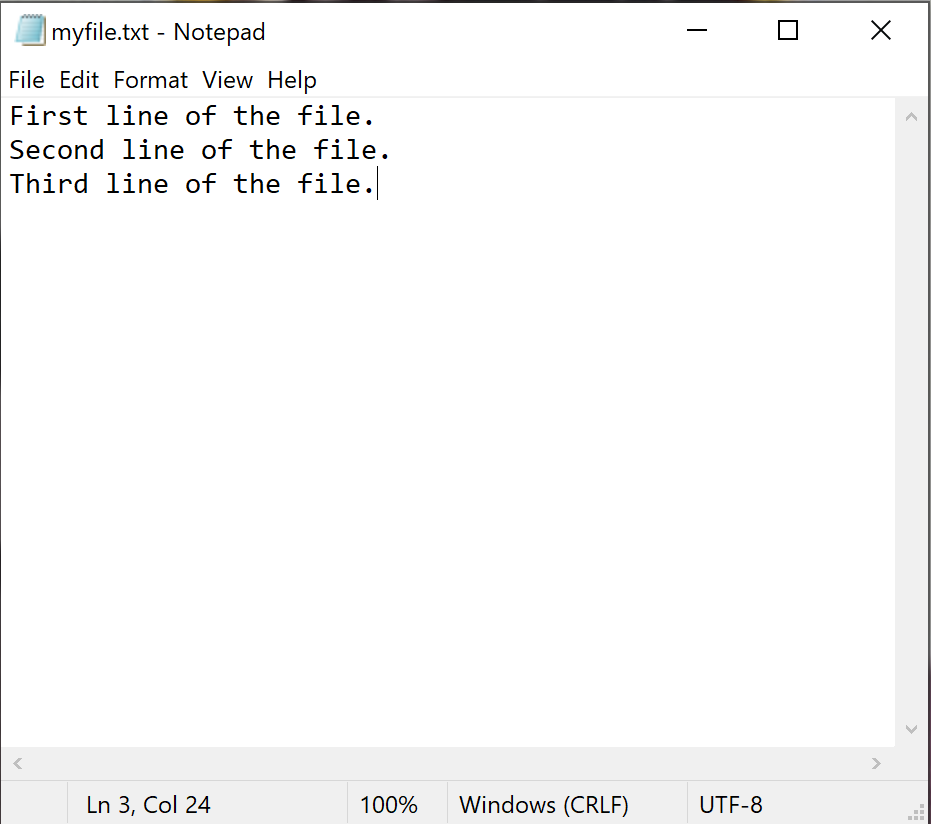
   Для записи данных в файл он должен быть открыт для записи (режимы 'w', 'а', 'r+'), иначе произойдет ошибка.

Рассмотрим текcтовый файл myfile.txt, содержащий следующие строки:

First line of the file.

Second line of the file.

Third line of the file.



Если файл открыт в режиме 'w', то его содержимое сначала полностью стирается, а уже затем в него добавляются данные.

После выполнения следующего кода:

with open('myfile.txt', 'w', encoding='utf-8') as file:

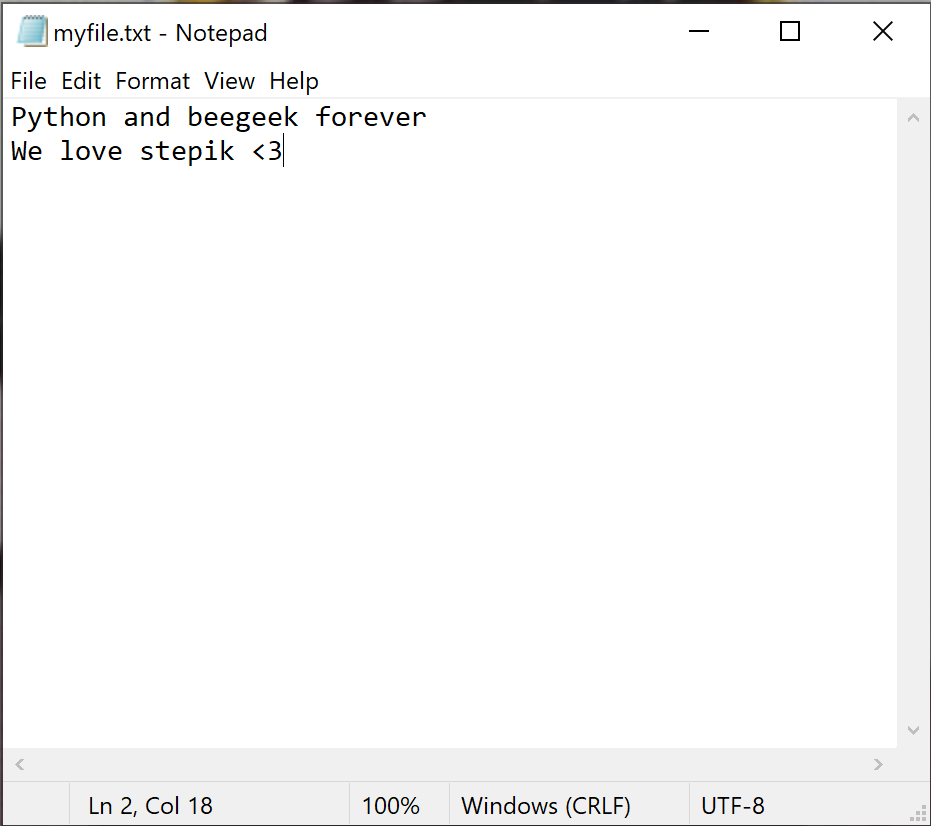
file.write('Python and beegeek forever\n')

file.write('We love stepik <3')

файл myfile.txt будет содержать:

Python and beegeek forever

We love stepik <3



Если файл открыт в режиме 'a', то запись происходит в самый конец файла.

После выполнения следующего кода:

with open('myfile.txt', 'a', encoding='utf-8') as file:

file.write('Python and beegeek forever\n')

file.write('We love stepik <3')

файл myfile.txt будет содержать:

First line of the file.

Second line of the file.

Third line of the file.Python and beegeek forever

We love stepik <3

Если файл открыт в режиме 'r+', то происходит частичная перезапись его содержимого.

После выполнения следующего кода:

with open('myfile.txt', 'r+', encoding='utf-8') as file:

file.write('Python and beegeek forever\n')

file.write('We love stepik.')

файл myfile.txt будет содержать:

Python and beegeek forever

We love stepik. file.

Third line of the file.

### Метод writelines()

Последовательные вызовы метода write() дописывают текст в конец файла.

Приведенный ниже код создает файл philosophers.txt и записывает в него три строки текста:

with open('philosophers.txt', 'w', encoding='utf-8') as file:

file.write('Джoн Локк\n')

file.write('Дэвид Хьюм\n')

file.write('Эдмyнд Берк\n')

На практике часто приходится записывать в файл содержимое целого списка. Это можно сделать с помощью цикла или метода writelines(), что удобнее. Метод writelines() принимает в качестве аргумента список строк и записывает его в файл.

Приведенный ниже код создает файл philosophers.txt и записывает в него содержимое списка philosophers:

philosophers = ['Джoн Локк\n', 'Дэвид Хьюм\n', 'Эдмyнд Берк\n']

with open('philosophers.txt', 'w', encoding='utf-8') as file:

file.writelines(philosophers)

Обратите внимание, что все записанные в файл строковые значения оканчиваются символом '\n', экранированной последовательностью новой строки. Символ '\n' не только отделяет находящиеся в файле значения друг от друга, но и обеспечивает появление каждого из них на отдельной строке во время просмотра данных в текстовом редакторе.

Такой вариант записи предпочтителен, когда нужно записать большой объем текста, который вы получаете и обрабатываете строчка за строчкой. Можно предварительно накопить весь текст в одну большую строку, однако для этого может потребоваться большой объём памяти. Гораздо лучше записывать строчки по мере готовности и writelines для этого подходит идеально!

### Запись в файл с помощью функции print()

Для записи данных в файл можно также использовать встроенную функцию print(). Для этого нужно передать ей еще один именованный аргумент file, указывающий на открытый файл. При этом функция print() автоматически добавляет переход на новую строку.

Приведенный ниже код:

with open('philosophers.txt', 'w', encoding='utf-8') as output:

print('Джoн Локк', file=output)

print('Дэвид Хьюм', file=output)

print('Эдмyнд Берк', file=output)

создает файл philosophers.txt с содержимым:

Джoн Локк

Дэвид Хьюм

Эдмyнд Берк

Мы можем использовать всю мощность встроенной функции print() для форматирования выводимого текста.

Приведенный ниже код:

with open('philosophers.txt', 'w', encoding='utf-8') as output:

print('Джoн Локк', 'Дэвид Хьюм', 'Эдмyнд Берк', sep='\*\*\*', file=output)

создает файл philosophers.txt с содержимым:

Джoн Локк\*\*\*Дэвид Хьюм\*\*\*Эдмyнд Берк

Не забывайте, что файловые методы write() и writelines() не добавляют переход на новую строку, поэтому для перехода на новую строку в файле необходимо явно добавить символ '\n'.

## Примечания

**Примечание 1.** В некоторых операционных системах невыполнение операции закрытия файла может привести к потере данных. Данные сначала пишутся в буфер – небольшую область временного хранения в оперативной памяти. Когда буфер заполняется, система записывает его содержимое в файл. Это увеличивает производительность системы, потому что запись данных в оперативную память быстрее записи на диск. Процесс закрытия файла записывает любые несохраненные данные из буфера в файл. Чтобы принудительно записать содержимое буфера в файл, используется файловый метод flush().

**Примечание 2.** Используйте конструкцию with для чтения и записи файлов. Закрывать файлы — полезная привычка, и если вы используете команду with при работе с файлами, вам не придется беспокоиться об их закрытии. Команда with автоматически закрывает файл за вас.