**1.Язык python. Основные определения. Ключевые слова.**

**Python**[[7]](https://ru.wikipedia.org/wiki/Python#cite_note-7) — высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объём полезных функций.

В Python имеются встроенные типы данных: [булевый](https://ru.wikipedia.org/wiki/%D0%91%D1%83%D0%BB%D0%B5%D0%B2%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D0%B8%D0%BF), [строка](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF), [Unicode](https://ru.wikipedia.org/wiki/Unicode)-строка, целое число произвольной точности, число [с плавающей запятой](https://ru.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%B2%D0%B0%D1%8E%D1%89%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D1%8F%D1%82%D0%B0%D1%8F), [комплексное число](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) и некоторые другие. Из [коллекций](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D0%BB%D0%B5%D0%BA%D1%86%D0%B8%D1%8F_(%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)) в Python встроены: [список](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), [кортеж](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D1%80%D1%82%D0%B5%D0%B6_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) (*неизменяемый список*), [словарь](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%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), [множество](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) и другие[[26]](https://ru.wikipedia.org/wiki/Python#cite_note-26). Все значения являются объектами, в том числе функции, методы, модули, классы.

К **неизменяемым** (*immutable*) типам относятся: целые числа (*int*),  числа с плавающей точкой (*float*), комплексные числа (*complex*), логические переменные (*bool*), кортежи (*tuple*), строки (*str*) и неизменяемые множества (*frozen set*).

К **изменяемым** (*mutable*) типам относятся: списки (*list*), множества (*set*), словари (*dict*).

Как уже было сказано ранее, при создании переменной, вначале создается объект, который имеет уникальный идентификатор, тип и значение, после этого переменная может ссылаться на созданный объект.

Неизменяемость типа данных означает, что созданный объект больше не изменяется. Например, если мы объявим переменную*k* = 15, то будет создан объект со значением 15, типа *int* и идентификатором, который можно узнать с помощью функции *id()*.

Если тип данных изменяемый, то можно менять значение объекта. Например, создадим список [1, 2], а потом заменим второй элемент на 3.

В рассмотренном случае, в качестве данных списка, выступают не объекты, а отношения между объектами. Т.е. в переменной a хранятся ссылки на объекты содержащие числа 1 и 3, а не непосредственно сами эти числа.

Добавить новый тип можно написав [класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%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)) (class)

**Объект** — **это** экземпляр класса. Класс можно сравнить с чертежом, по которому создаются **объекты**.

**Кортеж-** неизменяемый список.

a = tuple()

a=() – cоздание кортежа

a=(‘s’, )

a = tuple('hello, world!')

('h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!')

Все [операции над списками](https://pythonworld.ru/tipy-dannyx-v-python/spiski-list-funkcii-i-metody-spiskov.html), не изменяющие список (сложение, умножение на число, методы index() и count() и некоторые другие операции). Можно также по-разному менять элементы местами и так далее.

and as break class continue def global for from accept else if in is not or assert except import while with return true try nonlocal finally pass raise None False True

**Функция** в python - объект, принимающий аргументы и возвращающий значение.

**Операторы** – **это** некий функционал, производящий какие-либо действия, который может быть представлен в виде символов, как например +, или специальных зарезервированных слов.

**Модулем** в Python называется любой файл с программой. Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Подключить модуль можно с помощью инструкции import.

**Отдельный скрипт Python** - Скрипт Python - это исполняемый файл с расширением .py, который можно запустить из командной строки, из специального приложения, такого как PythonWin, или дважды щелкнув файл .py в Проводнике Windows (Windows Explorer).

**Окно Python** может использоваться для выполнения отдельных строк кода на Python с

выводом сообщений о результате в этом же окне.

**List (список)**

Базовая структура данных в python. Элементы в списке хранятся последовательно, каждому из них присвоены индексы, начиная с нуля. В отличие от массива, список может хранить объекты любого типа.

Создание списка

>>> my\_list = [] # Создание пустого списка с помощью литерала списка

>>> my\_list = list() # Создание пустого списка с помощью встроенной функции

>>>

>>> my\_list = [1,2,['a','b'],4,5] # Инициализация списка

>>>

>>> my\_list = list('hello world') # Создание списка из итерируемого объекта

>>> my\_list

['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']

>>>

>>> my\_list = [x for x in range(10)] # Генератор списков в действии

>>> my\_list

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

В каких случаях использовать?

Для хранения элементов, объединенных каким либо признаком. В случае, если изменение элементов и/или расширение списка не предполагается, следует использовать неизменяемый аналог - кортеж.

**Tuple (кортёж)**

Кортеж - это неизменяемый и более быстрый аналог списка. Он защищает хранимые данные от непреднамеренных изменений и может использоваться в качестве ключа в словарях (словарь - ассоциативный массив в python).

Создание кортежа.

>>> my\_tuple = () # Создание кортежа с помощью литерала

>>> my\_tuple = tuple() # Создание кортежа с помощью встроенной функции

>>>

>>> my\_tuple = (1,2,['a','b'],4,5) # Инициализация кортежа

>>>

>>> my\_tuple = tuple('hello world') # Создание кортежа из итерируемого объекта

>>> my\_tuple

('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')

>>>

>>> my\_tuple = tuple(2\*\*x for x in [0, 1, 2, 3]) # Генератор кортежей

>>> my\_tuple

(1, 2, 4, 8)

В каких случаях использовать?

Для хранения данных вместо списка (если они не предполагают изменений).

**Set (множество)**

Множество - это набор уникальных элементов в случайном порядке (неупорядоченный список). Множества примечательны тем, что операция проверки "принадлежит ли объект множеству" происходит значительно быстрее аналогичных операций в других структурах данных.

Создание множества

>>> my\_something = {} # !!! Попытка создать множество при помощи литерала даст нам словарь

>>> type(my\_something)

<class 'dict'>

>>>

>>> my\_set = set() # Создание при помощи встроенной функции

>>>

>>> my\_set = {1,2,3,4,5} # Инициализация множества

>>>

>>> my\_set = set('hello world') # Создания множества из итерируемого объекта

>>> my\_set

{'r', 'o', 'e', 'h', 'd', 'w', 'l', ' '}

>>>

>>> my\_set = {x for x in range(10)} # Генератор множеств

>>> my\_set

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

В каких случаях использовать?

Когда необходимо проверять принадлежит ли значение набору уникальных элементов и отсутствует необходимость поддерживать порядок в данном наборе.

## Ключевые слова

**False** - ложь.

**True** - правда.

**None** - "пустой" объект.

**and** - логическое И.

**with / as** - [менеджер контекста](https://pythonworld.ru/osnovy/with-as-menedzhery-konteksta.html).

**assert** условие - возбуждает исключение, если условие ложно.

**break** - выход из цикла.

**class** - [пользовательский тип](https://pythonworld.ru/osnovy/obektno-orientirovannoe-programmirovanie-obshhee-predstavlenie.html), состоящий из методов и атрибутов.

**continue** - переход на следующую итерацию цикла.

**def** - определение [функции](https://pythonworld.ru/tipy-dannyx-v-python/vse-o-funkciyax-i-ix-argumentax.html).

**del** - удаление объекта.

**elif** - в противном случае, если.

**else** - см. [for/else](https://pythonworld.ru/osnovy/cikly-for-i-while-operatory-break-i-continue-volshebnoe-slovo-else.html) или [if/else](https://pythonworld.ru/osnovy/instrukciya-if-elif-else-proverka-istinnosti-trexmestnoe-vyrazhenie-ifelse.html).

**except** - перехватить исключение.

**finally** - вкупе с инструкцией try, выполняет инструкции независимо от того, было ли исключение или нет.

**for** - цикл for.

**from** - импорт нескольких функций из модуля.

**global** - позволяет сделать значение переменной, присвоенное ей внутри функции, доступным и за пределами этой функции.

**if** - [если](https://pythonworld.ru/osnovy/instrukciya-if-elif-else-proverka-istinnosti-trexmestnoe-vyrazhenie-ifelse.html).

**import** - импорт модуля.

**in** - проверка на вхождение.

**is** - ссылаются ли 2 объекта на одно и то же место в памяти.

**lambda** - определение анонимной функции.

**nonlocal** - позволяет сделать значение переменной, присвоенное ей внутри функции, доступным в объемлющей инструкции.

**not** - логическое НЕ.

**or** - логическое ИЛИ.

**pass** - ничего не делающая конструкция.

**raise** - возбудить исключение.

**return** - вернуть результат.

**try** - выполнить инструкции, перехватывая [исключения](https://pythonworld.ru/tipy-dannyx-v-python/isklyucheniya-v-python-konstrukciya-try-except-dlya-obrabotki-isklyuchenij.html).

**while** - [цикл](https://pythonworld.ru/osnovy/cikly-for-i-while-operatory-break-i-continue-volshebnoe-slovo-else.html) while.

**Yield** это ключевое слово, которое используется примерно как return — отличие в том, что функция вернёт генератор.

**Class**- Оператор определения класса

Инструкция **assert**позволяет производить проверки истинности утверждений, что может быть использовано в отладочных целях.  
Если проверка не прошла, возбуждается исключение AssertionError.

Инструкция **raise** позволяет прервать штатный поток исполнения при помощи возбуждения исключения.  
  
Если после инструкции отсутствует выражение, то повторно поднимается исключение, имеющее место быть в данной области. Если в данной области нет активного исключения, возбуждается

**try**:  
 *# Допустим в функции поднимается FileNotFoundError.*  
 do()  
  
 **except**

**OSError**

:  
 *# Инструкция raise без выражение поднимет FileNotFoundError повторно.*  
 **raise**

Для чего применяется конструкция **with ... as**? Для гарантии того, что критические функции выполнятся в любом случае. Самый распространённый пример использования этой конструкции - открытие файлов. конструкция with ... as гарантирует закрытие файла в любом случае.

Например:

**with** open('newfile.txt', 'w', encoding='utf-8') **as** g:

d = int(input())

print('1 / *{}* = *{}*'.format(d, 1 / d), file=g)

**??2. Операторы языка. Виды операторов. Приоритеты операторов**

**Операторы**– это некий функционал, производящий какие-либо действия, который может быть представлен в виде символов, как например +, или специальных зарезервированных слов.

Операторы могут производить некоторые действия над данными, и эти данные называются **операндами.** В нашем случае 2 и 3 – это операнды.

Арифмитические операторы:

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Примеры** |
| **+** | Сложение - Суммирует значения слева и справа от оператора | 15 + 5 в результате будет 20 20 + -3 в результате будет 17 13.4 + 7 в результате будет 20.4 |
| **-** | Вычитание - Вычитает правый операнд из левого | 15 - 5 в результате будет 10 20 - -3 в результате будет 23 13.4 - 7 в результате будет 6.4 |
| **\*** | Умножение - Перемножает операнды | 5 \* 5 в результате будет 25 7 \* 3.2 в результате будет 22.4 -3 \* 12 в результате будет -36 |
| **/** | Деление - Делит левый операнд на правый | 15 / 5 в результате будет 3 5 / 2 в результате будет 2 (В Python 2.x версии при делении двух целых чисел результат будет целое число) 5.0 / 2 в результате будет 2.5 (Чтобы получить "правильный" результат хотя бы один операнд должен быть float) |
| **%** | ???Деление по модулю - Делит левый операнд на правый и возвращает остаток. | 6 % 2 в результате будет 0 7 % 2 в результате будет 1 13.2 % 5 в результате 3.2 |
| **\*\*** | ???Возведение в степень - возводит левый операнд в степень правого | 5 \*\* 2 в результате будет 25 2 \*\* 3 в результате будет 8 -3 \*\* 2 в результате будет -9 |
| **//** | Целочисленное деление - Деление в котором возвращается только целая часть результата. Часть после запятой отбрасывается.  Результат округляется в меньшую сторону. | 12 // 5 в результате будет 2 4 // 3 в результате будет 1 25 // 6 в результате будет 4  5 // 3 # вернет 1 -5 // 3 # вернет -2 |

## Операторы сравнения в Python:

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Примеры** |
| **==** | Проверяет равны ли оба операнда. Если да, то условие становится истинным. | 5 == 5 в результате будет True True == False в результате будет False "hello" == "hello" в результате будет True |
| **!=** | Проверяет равны ли оба операнда. Если нет, то условие становится истинным. | 12 != 5 в результате будет True False != False в результате будет False "hi" != "Hi" в результате будет True |
| **<>** | Проверяет равны ли оба операнда. Если нет, то условие становится истинным. | 12 <> 5 в результате будет True. Похоже на оператор != |
| **>** | Проверяет больше ли значение левого операнда, чем значение правого. Если да, то условие становится истинным. | 5 > 2 в результате будет True. True > False в результате будет True. "A" > "B" в результате будет False. |
| **<** | Проверяет меньше ли значение левого операнда, чем значение правого. Если да, то условие становится истинным. | 3 < 5 в результате будет True. True < False в результате будет False. "A" < "B" в результате будет True. |
| **>=** | Проверяет больше или равно значение левого операнда, чем значение правого. Если да, то условие становится истинным. | 1 >= 1 в результате будет True. 23 >= 3.2 в результате будет True. "C" >= "D" в результате будет False. |
| **<=** | Проверяет меньше или равно значение левого операнда, чем значение правого. Если да, то условие становится истинным. | 4 <= 5 в результате будет True. 0 <= 0.0 в результате будет True. -0.001 <= -36 в результате будет False. |

## 

## Операторы присваивания в Python:

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Примеры** |
| **=** | Присваивает значение правого операнда левому. | c = 23 присвоит переменной с значение 23 |
| **+=** | Прибавит значение правого операнда к левому и присвоит эту сумму левому операнду. | с = 5 а = 2  с += а равносильно: с = с + а. с будет равно 7 |
| **-=** | Отнимает значение правого операнда от левого и присваивает результат левому операнду. | с = 5 а = 2  с -= а равносильно: с = с - а. с будет равно 3 |
| **\*=** | Умножает правый операнд с левым и присваивает результат левому операнду. | с = 5 а = 2  с \*= а равносильно: с = с \* а. c будет равно 10 |
| **/=** | Делит левый операнд на правый и присваивает результат левому операнду. | с = 10 а = 2  с /= а равносильно: с = с / а. c будет равно 5 |
| **%=** | Делит по модулю операнды и присваивает результат левому. | с = 5 а = 2  с %= а равносильно: с = с % а. c будет равно 1 |
| **\*\*=** | Возводит в левый операнд в степень правого и присваивает результат левому операнду. | с = 3 а = 2  с \*\*= а равносильно: с = с \*\* а. c будет равно 9 |
| **//=** | Производит целочисленное деление левого операнда на правый и присваивает результат левому операнду. | с = 11 а = 2  с //= а равносильно: с = с // а. c будет равно 5 |

## Побитовые операторы в Python:

Побитовые операторы предназначены для работы с данными в битовом (двоичном) формате. Предположим, что у нас есть два числа a = 60; и b = 13. В двоичном формате они будут иметь следующий вид:

a = 0011 1100

b = 0000 1101

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Примеры** |
| **&** | Бинарный "И" оператор, копирует бит в результат только если бит присутствует в обоих операндах. | (a & b) даст нам 12, которое в двоичном формате выглядит так 0000 1100 |
| **|** | Бинарный "ИЛИ" оператор копирует бит, если тот присутствует в хотя бы в одном операнде. | (a | b) даст нам 61, в двоичном формате 0011 1101 |
| **^** | Бинарный "Исключительное ИЛИ" оператор копирует бит только если бит присутствует в одном из операндов, но не в обоих сразу. | (a ^ b) даст нам 49, в двоичном формате 0011 0001 |
| **~** | Бинарный комплиментарный оператор. Является унарным (то есть ему нужен только один операнд) меняет биты на обратные, там где была единица становиться ноль и наоборот. | (~a ) даст в результате -61, в двоичном формате выглядит 1100 0011. |
| **<<** | Побитовый сдвиг влево. Значение левого операнда "сдвигается" влево на количество бит указанных в правом операнде. | a << 2 в результате даст 240, в двоичном формате 1111 0000 |
| **>>** | Побитовый сдвиг вправо. Значение левого операнда "сдвигается" вправо на количество бит указанных в правом операнде. | a >> 2 даст 15, в двоичном формате  0000 1111 |

## 

## Логические операторы в Python:

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Примеры** |
| **and** | Логический оператор "И". Условие будет истинным если оба операнда истина. | True and True равно True. True and False равно False. False and True равно False. False and False равно False. |
| **or** | Логический оператор "ИЛИ". Если хотя бы один из операндов истинный, то и все выражение будет истинным. | True or True равно True. True or False равно True. False or True равно True. False or False равно False. |
| **not** | Логический оператор "НЕ". Изменяет логическое значение операнда на противоположное. | not True равно False. not False равно True. |

## 

## Операторы членства в Python:

В добавок к перечисленным операторам, в **Python** присутствуют, так называмые, операторы членства, предназначенные для проверки на наличие элемента в составных типах данных, таких, как [строки, списки, кортежи или словари](http://pythonicway.com/index.php/python-data-types):

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Примеры** |
| **in** | ??????Возвращает истину, если элемент присутствует в последовательности, иначе возвращает ложь. | "cad" in "cadillac" вернет True. 1 in [2,3,1,6] вернет True. "hi" in {"hi":2,"bye":1} вернет True.  2 in {"hi":2,"bye":1} вернет False (в словарях проверяется наличие в ключах, а не в значениях). |
| **not in** | Возвращает истину если элемента нет в последовательности. | Результаты противоположны результатам оператора in. |

## 

## Операторы тождественности в Python:

Операторы тождественности сравнивают размещение двух объектов в памяти компьютера.

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Примеры** |
| **is** | Возвращает истину, если оба операнда указывают на один объект. | x is y вернет истину, если id(x) будет равно id(y). |
| **is not** | Возврашает ложь если оба операнда указывают на один объект. | x is not y, вернет истину если id(x) не равно id(y). |

## Приоритет операторов в Python

В следующей таблице описан приоритет выполнения операторов в **Python** от наивысшего (выполняется в первую очередь) до наинизшего.

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| **\*\*** | Возведение в степень |
| **~** | Комплиментарный оператор |
| **\* / % //** | Умножение, деление, деление по модулю, целочисленное деление. |
| **+ -** | Сложение и вычитание. |
| **>> <<** | Побитовый сдвиг вправо и побитовый сдвиг влево. |
| **&** | Бинарный "И". |
| **^ |** | Бинарный  "Исключительное ИЛИ" и бинарный "ИЛИ" |
| **<= < > >=** | Операторы сравнения |
| **<> == !=** | Операторы равенства |
| **= %= /= //= -= += \*= \*\*=** | Операторы присваивания |
| **is is not** | Тождественные операторы |
| **in not in** | Операторы членства |
| **not or and** | Логические операторы |

(13 приоритетов)

А) Арифметические операторы (+,-,\*,/,%..)

Б) Операторы сравнения(==,!=,>,<,>=,<=)

В) Операторы присваивания(=,\*=,+=,-=,/=,\*\*=,//=)

Г) Побитовые операторы(&,|,^,>>,<<,~…)

Д) Логические операторы(and,or,not)

Е) Операторы членства(in,not in)

Ж) Операторы тождественности(is,is not)

Приоритет:

\*\*

~

\*/%//

+ -

<< >>

&

^|

<= >= > <

<>==!=

= %= \*= += -=

Is is not

In not in

Not or and

**???3. Встроенные функции языка python.**

abs() , ord()- числовой код символа, chr()- символ по числовому коду, complex(), format(), input(),

float(), round(), int(), map(), len(), max(), min(), print(), pow(), sorted(), range()

## Встроенные функции, выполняющие преобразование типов

**bool**(x) - преобразование к типу bool, использующая стандартную процедуру [проверки истинности](https://pythonworld.ru/osnovy/instrukciya-if-elif-else-proverka-istinnosti-trexmestnoe-vyrazhenie-ifelse.html). Если х является ложным или опущен, возвращает значение False, в противном случае она возвращает True.

**Sorted()-** сортировка

**complex**([real[, imag]]) - преобразование к [комплексному числу](https://pythonworld.ru/tipy-dannyx-v-python/chisla-int-float-complex.html#complex).

x = complex(1, 2)

**print**(x)

(1+2j)

**>>> print(x.imag)** *# Мнимая часть*

2.0

**>>> print**(**x.real)** *# Действительная часть*

1.0

*# Комплексные числа нельзя сравнить, Но можно проверить на равенство*, находить *Модуль комплексного числа*, *Возводить его в степень*

**dict**([object]) - преобразование к [словарю](https://pythonworld.ru/tipy-dannyx-v-python/slovari-dict-funkcii-i-metody-slovarej.html).

**float**([X]) - преобразование к [числу с плавающей точкой](https://pythonworld.ru/tipy-dannyx-v-python/chisla-int-float-complex.html#float). Если аргумент не указан, возвращается 0.0.

**int**([object], [основание системы счисления]) - преобразование к [целому числу](https://pythonworld.ru/tipy-dannyx-v-python/chisla-int-float-complex.html#int).

**list**([object]) - создает [список](https://pythonworld.ru/tipy-dannyx-v-python/spiski-list-funkcii-i-metody-spiskov.html).

**object**() - возвращает безликий объект, являющийся базовым для всех объектов.

**range**([start=0], stop, [step=1]) - арифметическая прогрессия от start до stop с шагом step.

**set**([object]) - создает [множество](https://pythonworld.ru/tipy-dannyx-v-python/mnozhestva-set-i-frozenset.html).

**slice**([start=0], stop, [step=1]) - объект среза от start до stop с шагом step.

**str**([object], [кодировка], [ошибки]) - строковое представление объекта. Использует метод \_\_str\_\_.

**tuple**(obj) - преобразование к [кортежу](https://pythonworld.ru/tipy-dannyx-v-python/kortezhi-tuple.html).

## Другие встроенные функции

**abs**(x) - Возвращает абсолютную величину (модуль числа).

**all**(последовательность) - Возвращает True, если все элементы истинные (или, если последовательность пуста).

**any**(последовательность) - Возвращает True, если хотя бы один элемент - истина. Для пустой последовательности возвращает False.

print (any([False, False, False, False])) # False

print (any([False, True, False, False])) #True

**bin**(x) - Преобразование целого числа в двоичную строку.

**chr**(x) - Возвращает односимвольную строку, код символа которой равен x.

**divmod**(a, b) - Возвращает частное и остаток от деления a на b.

**????**

**help**([object]) - Вызов встроенной справочной системы.

**id**(object) - Возвращает "адрес" объекта. Это целое число, которое гарантированно будет уникальным и постоянным для данного объекта в течение срока его существования.

**input**([prompt]) - Возвращает введенную пользователем строку. Prompt - подсказка пользователю.

**len**(x) - Возвращает число элементов в указанном объекте.

**map**(function, iterator) - Итератор, получившийся после применения к каждому элементу последовательности функции function.

**map**. Случаются ситуации, когда нужно применить какую-либо функцию к каждому элементу списка

new\_list = list(map(int, old\_list))

a,b,c=map(int,input('').split(''))

**def** f**(**x**):**

**return** x**\***x

**print** map**(**f**,** nums**)**

**max**(iter, [args ...] \* [, key]) - Максимальный элемент последовательности.

**min**(iter, [args ...] \* [, key]) - Минимальный элемент последовательности.

**open**(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True) - Открывает файл и возвращает соответствующий поток.

**ord**(с) - Код символа.

**pow**(x, y[, r]) - ( x \*\* y ) % r.

**print**([object, ...], \*, sep=" ", end='\n', file=sys.stdout) - Печать.

**round**(X [, N]) - Округление до N знаков после запятой.

**sum**(iter, start=0) - Сумма членов последовательности.

numbers = [1,2,3,4,5,1,4,5]

# start parameter is not provided

Sum = sum(numbers)

print(Sum)

# start = 10

Sum = sum(numbers, 10)

print(Sum)

**type**(object) - Возвращает тип объекта.

**4. Модуль random. Работа со случайными числами.**

**Модулем** в Python называется любой файл с программой. Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Подключить модуль можно с помощью инструкции import.

Модуль **random** предоставляет функции для генерации случайных чисел, букв, случайного выбора элементов последовательности.

-------------------------------

**random.getrandbits(N)** - возвращает N случайных бит.

**random.randrange(start, stop, step)** - возвращает случайно выбранное число из последовательности.

**random.randint(A, B)** - случайное целое число N, A ≤ N ≤ B.

**random.choice(sequence)** - случайный элемент непустой последовательности.

**random.shuffle(sequence, [rand])** - перемешивает последовательность (изменяется сама последовательность). Поэтому функция не работает для неизменяемых объектов.

**random.sample(population, k)** - список длиной k из последовательности population.

**random.random()** - случайное число от 0 до 1.

**random.uniform(A, B)** - случайное число с плавающей точкой, A ≤ N ≤ B (или B ≤ N ≤ A).

-------------------------------

Модуль random управляет генерацией случайных чисел. Его основные функции:

* **random()**: генерирует случайное число от 0.0 до 1.0
* **randint()**: возвращает случайное число из определенного диапазона
* **randrange()**: возвращает случайное число из определенного набора чисел
* **shuffle()**: перемешивает список
* **choice()**: возвращает случайный элемент списка

Функция **random()** возвращает случайное число с плавающей точкой в промежутке от 0.0 до 1.0. Если же нам необходимо число из большего диапазона, скажем от 0 до 100, то мы можем соответственно умножить результат функции random на 100.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import random    number = random.random()  # значение от 0.0 до 1.0  print(number)  number = random.random() \* 100  # значение от 0.0 до 100.0  print(number) |

Функция **randint(min, max)** возвращает случайное целое число в промежутке между двумя значениями min и max.

|  |  |
| --- | --- |
| 1  2  3  4 | import random    number = random.randint(20, 35)  # значение от 20 до 35  print(number) |

Функция **randrange()** возвращает случайное целое число из определенного набора чисел. Она имеет три формы:

* randrange(stop): в качестве набора чисел, из которых происходит извлечение случайного значения, будет использоваться диапазон от 0 до числа stop
* randrange(start, stop): набор чисел представляет диапазон от числа start до числа stop
* randrange(start, stop, step): набор чисел представляет диапазон от числа start до числа stop, при этом каждое число в диапазоне отличается от предыдущего на шаг step

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import random    number = random.randrange(10)  # значение от 0 до 10  print(number)  number = random.randrange(2, 10)  # значение в диапазоне 2, 3, 4, 5, 6, 7, 8, 9, 10  print(number)  number = random.randrange(2, 10, 2)  # значение в диапазоне 2, 4, 6, 8, 10  print(number) |

### Работа со списком

Для работы со списками в модуле random определены две функции: функция **shuffle()** перемешивает список случайным образом, а функция **choice()** возвращает один случайный элемент из списка:

|  |  |
| --- | --- |
| 1  2  3  4  5 | numbers = [1, 2, 3, 4, 5, 6, 7, 8]  random.shuffle(numbers)  print(numbers)  random\_number = random.choice(numbers)  print(random\_number) # 1 |

**5. Функции ввода и вывода.**

input() – возвращает текстовую строку

print(…,sep=’’,end=’’)

# **Вывод данных в консоль**

Один из самых распространенных способов вывести данные в Python – это напечатать их в консоли. Для вывода данных в консоль используется функция print.

Рассмотрим основные способы использования данной функции.

>>> print("Hello")

Hello

>>> print("Hello, " + "world!")

Hello, world!

>>> print("Age: " + str(**23**))

Age: 23

По умолчанию, для разделения элементов в функции print используется пробел.

>>> print("A", "B", "C")

A B C

Для замены разделителя необходимо использовать параметр sep функции print.

print("A", "B", "C", sep="#")

A#B#C

В качестве конечного элемента выводимой строки, используется символ перевода строки.

>>> for i in range(**3**):

print("i: " + str(i))

i: 0

i: 1

i: 2

Для его замены используется параметр end.

>>> for i in range(**3**):

print("[i: " + str(i) + "]", end=" -- ")

[i: 0] -- [i: 1] -- [i: 2] --

# **Ввод данных с клавиатуры**

Для считывания вводимых с клавиатуры данных используется функция input().

>>> input()

test

'test'

Для сохранения данных в переменной используется следующий синтаксис.

>>> a = input()

hello

>>> print(a)

hello

Если считывается с клавиатуры целое число, то строку, получаемую с помощью функции input(), можно передать сразу в функцию int().

>>> val = int(input())

123

>>> print(val)

123

>>> type(val)

<*class* 'int'>

Для вывода строки-приглашения, используйте ее в качестве аргумента функции input().

>>> tv = int(input("input number: "))

input number: 334

>>> print(tv)

334

**Преобразование строки в список осуществляется с помощью метода split(), по умолчанию, в качестве разделителя, используется пробел.**

>>> l = input().split()

1 2 3 4 5 6 7

>>> print(l)

['1', '2', '3', '4', '5', '6', '7']

Разделитель можно заменить, указав его в качестве аргумента метода split().

>>> nl = input().split("-")

1-2-3-4-5-6-7

>>> print(nl)

['1', '2', '3', '4', '5', '6', '7']

Для считывания списка чисел с одновременным приведением их к типу int можно воспользоваться вот такой конструкцией.

>>> nums = list(map(int, input().split()))

1 2 3 4 5 6 7

>>> print((nums))

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

**6. Функция вывода. Форматирование вывода.**

print(’{:5d} or {:7.3f} or {:9.3e}’ . format())

**format**(value[,format\_spec]) - [форматирование строки](https://pythonworld.ru/osnovy/formatirovanie-strok-metod-format.html)). **(%)**

### Именованные параметры

В форматируемой строке мы можем определять параметры, в методе format() передавать для этих параметров значения:

|  |  |
| --- | --- |
| 1  2  3  4 | text = "Hello, {first\_name}.".format(first\_name="Tom")  print(text)     # Hello, Tom.  info = "Name: {name}\t Age: {age}".format(name="Bob", age=23)  print(info)     # Name: Bob  Age: 23  Если для подстановки требуется только один аргумент, то значение - сам аргумент: |

**>>>** 'Hello, {}!'.format('Vasya')

'Hello, Vasya!'

**Вы можете передать методу любое значение, которое необходимо вставить в строку. При запуске программы это значение будет добавлено в строку в то место, где находятся фигурные скобки.**

Например:

print("I have {} apples.".format(5))  
I have 5 apples.

Также можно сохранить строку с заменителем в переменной:

**apple\_string = "I have {}."  
print(apple\_string.format("5 apples"))**I have 5 apples.

Теперь метод format вставил на место **заменителя** строку «5 apples».

## Множественная замена

В строке можно использовать несколько заменителей. Попробуйте добавить в строку ещё одну пару фигурных скобок, а в метод – ещё одно значение.

new\_apple\_string = "I love {} {}."                 #2 {} заменителя  
print(new\_apple\_string.format("red", "apples"))    #Значения нужно разделить запятой  
I love red apples.

Метод str.format() может передавать несколько строк; для этого нужно разделить их запятыми, как показано в примере выше.

Попробуйте добавить в строку ещё несколько заменителей:

my\_string = "I love {} {}, and have {} {}."               #4 {} заменителя  
print(my\_string.format("red", " apples", 5, "apples"))    #4 строки в методе  
I love red apples, and have 5 apples.

## Позиционные и именованные аргументы

**Если фигурные скобки передаются без каких-либо параметров, Python вставит значения, переданные с помощью метода str.format (), по порядку,**

print("My friend {} has a pet {}!".format("Sam", " winged dinosaur"))  
My friend Sam has a pet winged dinosaur!

"Sam", " winged dinosaur"

Эти данные являются кортежем. **Любое значение кортежа можно вызвать с помощью индекса – порядкового номера (начиная с 0).**

**>>>** '{0}, {1}, {2}'.format('a', 'b', 'c')

'a, b, c'

**>>>** '{}, {}, {}'.format('a', 'b', 'c')

'a, b, c'

**>>>** '{2}, {1}, {0}'.format('a', 'b', 'c')

'c, b, a'

**>>>** '{2}, {1}, {0}'.format(\*'abc')

'c, b, a'

**>>>** '{0}{1}{0}'.format('abra', 'cad')

'abracadabra

### **Подстановки**

Еще один способ передачи форматируемых значений в строку представляет использование подстановок или специальных плейсхолдеров, на место которых вставляются определенные значения. Для форматирования мы можем использовать следующие плейсхолдеры:

* **s**: для вставки строк
* **d**: для вставки целых чисел
* **f**: для вставки дробных чисел. Для этого типа также можно определить через точку количество знаков в дробной части.
* **%**: умножает значение на 100 и добавляет знак процента
* **e**: выводит число в экспоненциальной записи

**Общий синтаксис плейсхолдера следующий:**

|  |  |
| --- | --- |
| **1** | **{:плейсхолдер}** |

В зависимости от плейсхолдера можно добавлять дополнительные параметры. Например, для форматирования чисел float можно использовать следующие параметры

|  |  |
| --- | --- |
| 1 | {:[количество\_символов][запятая][.число\_знаков\_в\_дробной\_части] плейсхолдер} |

При вызове метода format в него в качестве аргументов передаются значения, которые вставляются на место плейсхолдеров:

|  |  |
| --- | --- |
| 1  2  3  4 | welcome = "Hello {:s}"  name = "Tom"  formatted\_welcome = welcome.format(name)  print(formatted\_welcome)        # Hello Tom |

В качестве результата метод format() возвращает новую отформатированную строку.

Форматирование целых чисел:

|  |  |
| --- | --- |
| 1  2  3  4 | source = "{:d} символов"  number = 5  target = source.format(number)  print(target)   # 5 символов |

**Если форматируемое число больше 999, то мы можем указать в определении плейсхолдера, что мы хотим использовать запятую в качестве разделителя разрядов:**

|  |  |
| --- | --- |
| **1**  **2** | **source = "{:,d} символов"**  **print(source.format(5000))   # 5,000 символов** |

Для дробных чисел, то есть таких, которые представляют тип float, перед кодом плейсхолдера после точки можно указать, сколько знаков в дробной части мы хотим вывести:

**Число округляется**

|  |  |
| --- | --- |
| 1  2  3  4  5 | number = 23.8589578  print("{:.2f}".format(number))   # 23.86  print("{:.3f}".format(number))   # 23.859  print("{:.4f}".format(number))   # 23.8590  print("{:,.2f}".format(10001.23554))    # 10,001.24 |

Еще один параметр позволяет установить минимальную ширину форматируемого значения в символах:

|  |  |
| --- | --- |
| 1  2 | print("{:10.2f}".format(23.8589578))    #     23.86  print("{:8d}".format(25))               #      25 |

Для вывода процентов лучше воспользоваться кодом "%":

|  |  |
| --- | --- |
| 1  2  3  4 | number = .12345  print("{:%}".format(number))        # 12.345000%  print("{:.0%}".format(number))      # 12%  print("{:.1%}".format(number))      # 12.3% |

Для вывода числа в экспоненциальной записи применяется плейсхолдер "e":

|  |  |
| --- | --- |
| 1  2  3  4 | number = 12345.6789  print("{:e}".format(number))        # 1.234568e+04  print("{:.0e}".format(number))      # 1e+04  print("{:.1e}".format(number))      # 1.2e+04 |

### **Форматирование без метода format**

В принципе для форматирования значений мы можем обойтись и без метода format, применяя следующий синтаксис:

|  |  |
| --- | --- |
| 1 | строка%(параметр1, параметр2,..параметрN) |

То есть в начале идет строка, которая содержит те же плейсхолдеры, которые были рассмотрены выше (за исключением плейсхолдера %), после строки ставится знак процента %, а затем список значений, которые вставляются в строку. Фактически знак процента представляют операцию, в результате которой образуется новая строка:

|  |  |
| --- | --- |
| 1  2 | info = "Имя: %s \t Возраст: %d" % ("Tom", 35)  print(info)   # Имя: Tom     Возраст: 35 |

Рядом с плейсхолдером указывается знак процента и в отличие от функции format здесь **не требуются фигурные скобки.**

Причем способы форматирования чисел здесь также применяются:

|  |  |
| --- | --- |
| 1  2 | number = 23.8589578  print("%0.2f  - %e" % (number, number))   # 23.86  - 2.385896e+01 |

## Выравнивание строк

>>> a = "Hello"

>>> a.center(10)

' Hello '

>>> a.rjust(10)

' Hello'

>>> a.ljust(10)

'Hello '

>>> a.ljust(10,'.')

'Hello.....'

>>> a.center(10,'.')

'..Hello...'

>>> "%s" % (a.center(19))

' Hello '

**??7. Условные операторы. Полные условные операторы. Неполные условные операторы. Тернарный оператор условия. Примеры использования.**

if .. else – полный

If – неполный

m = x if x > y else y

## ****Условный оператор ветвления****if

Оператор ветвления if  позволяет выполнить определенный набор инструкций в зависимости от некоторого условия. Возможны следующие варианты использования.

### **Конструкция**if

Синтаксис оператора if выглядит так.

if выражение:

инструкция\_**1**

инструкция\_**2**

...

инструкция\_n

После оператора if  записывается выражение. Если это выражение истинно, то выполняются инструкции, определяемые данным оператором. Выражение является истинным, если его результатом является число не равное нулю, непустой объект, либо логическое True. После выражения нужно поставить двоеточие “:”.

**ВАЖНО**: блок кода, который необходимо выполнить, в случае истинности выражения, отделяется четырьмя пробелами слева!

Примеры:

?????

if **1**:

print("hello 1")

Напечатает: hello 1

a = **3**

if a == **3**:

   print("hello 2")

Напечатает: hello 2

a = **3**

if a > **1**:

print("hello 3")

Напечатает: hello 3

??????

lst = [**1**, **2**, **3**]

if lst :

print("hello 4")

Напечатает: hello 4

### **Конструкция**if – else

Бывают случаи, когда необходимо предусмотреть альтернативный вариант выполнения программы. Т.е. при истинном условии нужно выполнить один набор инструкций, при ложном – другой. Для этого используется конструкция if – else.

if выражение:

инструкция\_1

инструкция\_2

...

инструкция\_n

else:

инструкция\_a

инструкция\_b

...

инструкция\_x

Примеры.

a = **3**

if a > **2**:

print("H")

else:

print("L")

Напечатает: H

a = **1**

if a > **2**:

   print("H")

else:

   print("L")

Напечатает: L

Условие такого вида можно записать в строчку, в таком случае оно будет представлять собой [**тернарное выражение**](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%80%D0%BD%D0%B0%D1%80%D0%BD%D0%B0%D1%8F_%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F)**.**

a = **17**

b = **True** if a > **10** else **False**

print(b)

В результате выполнения такого кода будет напечатано: True

### **Конструкция**if – elif – else

Для реализации выбора из нескольких альтернатив можно использовать конструкцию if – elif – else.

if выражение\_1:

инструкции\_(блок\_1)

elif выражение\_2:

инструкции\_(блок\_2)

elif выражение\_3:

инструкции\_(блок\_3)

else:

инструкции\_(блок\_4)

Пример.

a = int(input("введите число:"))

if a < **0**:

print("Neg")

elif a == **0**:

print("Zero")

else:

print("Pos")

Если пользователь введет число меньше нуля, то будет напечатано “Neg“, равное нулю – “Zero“, большее нуля – “Pos“.

**Тернарный оператор**– используется в строчку для задания условий в присваивании значения переменной.

cat\_say = “Mew”  
me\_say = “Hi,cat” if cat\_say == “Mew” or cat\_say == “Myavki” else “Who are you?”  
print(me\_say)  
Hi,cat *#результат*

Тернарный оператор по сути включает в себя имя переменной, которой мы присваиваем значение и внутри этой же строки задаем условия, описываемые операторами if/or/else. Не обязательно использовать все три if/or/else оператора, в условии может использоваться просто if/else.

**Важно**: тернарный условный оператор в Python может использоваться как со строками, так и с числами.

**8. Условные операторы. Множественный выбор. Вложенные операторы условия. Примеры использования.**

if….

elif….

else..

if ….:

if..

else

Ранее мы рассмотрели работу условного оператора if. С помощью его расширенной версии if-else можно реализовать две отдельные ветви выполнения. Однако алгоритм программы может предполагать выбор больше, чем из двух путей, например, из трех, четырех или даже пяти. В данном случае следует говорить о необходимости множественного ветвления.

в язык Python встроена возможность настоящего **множественного ветвления на одном уровне вложенности, которое реализуется с помощью веток elif**.

Слово "elif" образовано от двух первых букв слова "else", к которым присоединено слово "if". Это можно перевести как "иначе если".

В отличие от else, **в заголовке elif обязательно должно быть логическое выражение** также, как в заголовке if.

old = int(input('Ваш возраст: '))

**print**('Рекомендовано:', end=' ')

**if** 3 <= old < 6:

**print**('"Заяц в лабиринте"')

**elif** 6 <= old < 12:

**print**('"Марсианин"')

**elif** 12 <= old < 16:

**print**('"Загадочный остров"')

**elif** 16 <= old:

**print**('"Поток сознания"')

Обратите внимание, в конце, после всех elif, может использоваться одна ветка else для обработки случаев, не попавших в условия ветки if и всех elif.

В процессе разработки может возникнуть ситуация, в которой после одной истинной проверки следует сделать еще несколько. В таком случае необходимо **использовать вложенные условные конструкции.** То есть одну **if...elif...else**конструкцию внутри другой.

Например:

[?](http://pythonicway.com/python-conditionals)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | if num > 100:      if num < 150:          print "Число больше ста, но меньше ста пятидесяти"      elif num < 200:          print "Число больше ста, но меньше двухсот"  elif num > 50:      if num < 90:          print "Число больше пятидесяти, но меньше девяноста"      else:          print "Число больше пятидесяти и больше девяноста"  else:      print "Число меньше пятидесяти" |

Логика выполнения вложенных условных конструкций та же, что и у обычных. Главное не запутаться с отступами и порядком выполнения сравнений.

**9. Операторы цикла. Оператор цикла с условием. Операторы break и continue. Примеры использования.**

**while**

Оператор continue переводит цикл на следующую итерацию, пропуская всё "оставшееся" тело цикла.

Оператор break завершает цикл.

Инструкция **while** в **Python** повторяет указанный блок кода до тех пор, пока указанное в цикле условие будет оставаться истинным.

Цикл while в Python записывается следующим образом:

|  |  |
| --- | --- |
| 1  2 | while условие:      выражение |

 При этом выражением может быть как одна так и несколько инструкций. Условием может быть любое истинное или ненулевое значение. Выражение будет повторяться, пока условие будет истинным.

Когда условие становится ложным интерпретатор переводит выполнение программы на строку, следующую за циклом.

**Цикл while** становится **бесконечным** в случае, когда условие цикла никогда не становится ложным. Примером задачи для реализации которой необходим **бесконечный цикл** может быть, например, создание программы "Часы", которая бесконечно будет обновлять и отображать время.

Здесь мы имеем дело с бесконечным циклом, так как условие **while True** всегда истинно и всегда будет выполняться. Это популярный прием для создания программ, которые должны выполняться неопределенно долго.Выходим с помощью break

*break - Производит переход за пределы объемлющего цикла (всей инструкции цикла).*

#### continue - Производит переход в начало цикла (в строку заголовка).

#### pass**-**Ничего не делает: это пустая инструкция, используемая как заполнитель.

#### Блокelse **-**Выполняется, только если цикл завершился обычным образом (без

#### использования инструкции break).

#### while <test1>:

#### <statements1>

#### if <test2>: break    # Выйти из цикла, пропустив часть else

#### if <test3>: continue # Перейти в начало цикла, к выражению

#### # test1

#### else:

#### <statements2>        # Выполняется, если не была использована

#### # инструкция ‘break’

#### -----------------------------------------------------------------

#### **Бесконечный цикл, который ничего не делает:**

#### while 1:

#### pass

#### **continue:**

Оператор continue переводит цикл на следующую итерацию, пропуская всё "оставшееся" тело цикла.

#### **Пример, обход нечетных чисел:**

#### x = 10

#### while x: #выполняется пока x>0

#### x = x-1                   # Или, x -= 1

#### if x % 2 != 0:

#### continue   # Нечетное? – пропустить вывод

#### print(x, end=’ ‘)

#### -----------------------------------------------------------------

#### **break:**

#### Инструкция break вызывает немедленный выход из цикла.

#### Ввод данных с помощью функции input и производится

#### выход из цикла, если в ответ на запрос имени будет введена строка «stop»:

#### >>> while 1:

#### ...     name = input(‘Enter name:’)

#### ...     if name == ‘stop’: break

#### ...     age = input(‘Enter age: ‘)

#### ...     print(‘Hello’, name, ‘=>’, int(age) \*\* 2)

**10. Операторы цикла. Оператор цикла с итератором. Примеры использования.**

for переменная in последовательность

range(a,b,h)

### **Цикл for**

### Цикл for вызывается для каждого числа в некоторой коллекции чисел. Коллекция чисел создается с помощью функции **range()**. Формальное определение цикла for:

|  |  |
| --- | --- |
|  | for int\_var in функция\_range:    инструкции Инструкция **for способна работать**  со строками, списками, кортежами, с другими встроенными объектами,  поддерживающими возможность выполнения итераций. **Общий формат:**for <target> in <object>:   # Связывает элементы объекта с                            # переменной цикла    <statements1>       # Повторяющееся тело цикла: использует                       # переменную циклаelse:    <statements2>       # Если не попали на инструкцию ‘break’>>> for x in [“spam”, “eggs”, “ham”]:...     print(x, end=’ ‘)...#spam eggs ham |

При выполнении цикла Python последовательно получает все числа из коллекции, которая создается функцией range, и сохраняет эти числа в переменной int\_var. При первом проходе цикл получает первое число из коллекции, при втором - второе число и так далее, пока не переберет все числа. Когда все числа в коллекции будут перебраны, цикл завершает свою работу.

Рассмотрим на примере вычисления факториала:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #! Программа по вычислению факториала    number = int(input("Введите число: "))  factorial = 1  for i in range(1, number+1):      factorial \*= i  print("Факториал числа", number, "равен", factorial) |

#### **Функция range**

Функция range имеет следующие формы:

* **range(stop)**: возвращает все целые числа от 0 до stop
* **range(start, stop)**: возвращает все целые числа в промежутке от start (включая) до stop (не включая).
* **range(start, stop, step)**: возвращает целые числа в промежутке от start (включая) до stop (не включая), которые увеличиваются на значение step

Примеры вызовов функции range:

|  |  |
| --- | --- |
| 1  2  3  4 | range(5)            # 0, 1, 2, 3, 4  range(1, 5)         # 1, 2, 3, 4  range(2, 10, 2)     # 2, 4, 6, 8  range(5, 0, -1)     # 5, 4, 3, 2, 1 |

Например, выведем последовательно все числа от 0 до 4:

|  |  |
| --- | --- |
| 1  2 | for i in range(5):      print(i, end=" ") |

### Вложенные циклы

Одни циклы внутри себя могут содержать другие циклы. Рассмотрим на примере вывода таблицы умножения:

|  |  |
| --- | --- |
| 1  2  3  4 | for i in range(1, 10):      for j in range(1, 10):          print(i \* j, end="\t")      print("\n") |

Внешний цикл for i in range(1, 10) срабатывает 9 раз, так как в коллекции, возвращаемой функцией range, 9 чисел. Внутренний цикл for j in range(1, 10) срабатывает 9 раз для одной итерации внешнего цикла, и соответственно 81 раз для всех итераций внешнего цикла.

**11. Модуль math. Основные функции модуля. Примеры использования функций.**

sqrt(), sin(), cos(),tan(), floor() – округление вниз , factorial(), exp(), log(), acos() , asin(), atan(),

ceil-округление до ближайшего большего числа, degrees() – радианы в градусы, radians()

**Модулем** в Python называется любой файл с программой. Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Подключить модуль можно с помощью инструкции import.

Модуль math – один из наиважнейших в Python. Этот модуль предоставляет обширный функционал для работы с числами.

**math.ceil**(X) – округление до ближайшего большего числа.

**math.floor**(X) - округление вниз.

**math.fabs**(X) - модуль X.

**math.factorial**(X) - факториал числа X.

**math.fmod**(X, Y) - остаток от деления X на Y.

**math.fsum**(последовательность) - сумма всех членов последовательности. Эквивалент встроенной функции sum(), но math.fsum() более точна для чисел с плавающей точкой.

**math.isfinite**(X) **- является ли X числом. (True/False)**

**math.isinf**(X) **- является ли X бесконечностью**.

Isnan- не число

**math.trunc**(X) - **усекает значение X до целого.**

**math.exp**(X) - **eX.**

**math.log**(X, [base]) - логарифм X по основанию base. Если base не указан, вычисляется натуральный логарифм.

**math.pow**(X, Y) - XY.

**math.sqrt**(X) - квадратный корень из X.

**math.acos**(X) - арккосинус X. В радианах.

**math.asin**(X) - арксинус X. В радианах.

**math.atan**(X) - арктангенс X. В радианах.

**math.cos**(X) - косинус X (X указывается в радианах).

**math.sin**(X) - синус X (X указывается в радианах).

**math.tan**(X) - тангенс X (X указывается в радианах).

**math.hypot**(X, Y) - **вычисляет гипотенузу треугольника с катетами X и Y (math.sqrt(x \* x + y \* y)).**

**math.degrees**(X) - конвертирует радианы в градусы.

**math.radians**(X) - конвертирует градусы в радианы.

**math.cosh**(X) - **вычисляет гиперболический косинус.**

**math.sinh**(X) - вычисляет гиперболический синус.

**math.tanh**(X) - вычисляет гиперболический тангенс.

**math.acosh**(X) **- вычисляет обратный гиперболический косинус.**

**math.asinh**(X) - вычисляет обратный гиперболический синус.

**math.atanh**(X) - вычисляет обратный гиперболический тангенс.

**math.pi** - pi = 3,1415926...

**math.e** - e = 2,718281...

import math

# возведение числа 2 в степень 3

n1 = math.pow(2, 3)

print(n1)  # 8

# ближайшее наибольшее целое число

print(math.ceil(4.56))  # 5

# ближайшее наименьшее целое число

print(math.floor(4.56))  # 4

# перевод из радиан в градусы

print(math.degrees(3.14159))  # 180

# перевод из градусов в радианы

print(math.radians(180))   # 3.1415.....

# косинус

print(math.cos(math.radians(60)))  # 0.5

# cинус

print(math.sin(math.radians(90)))   # 1.0

import math

radius = 30

# площадь круга с радиусом 30

area = math.pi \* math.pow(radius, 2)

print(area)

**12. Строки. Основные функции и методы работы со строками. Пример программы.**

len(), S.split(), S.isdigit(), S.isalpha(), S.islower(), S.isupper(), S.upper(), S.lower(), S.strip(), S[i:j:step],

S\*3, S1+S2

**Строки в Python** – тип данных, позволяющий обрабатывать текстовую информацию. Длина введенного текста может ограничиваться только ресурсами компьютера. Необходимо помнить, что строки – неизменяемый тип данных. Все происходящее с изначальным текстом присваивается новой переменной.

основные методы строк, которые мы можем применить в приложениях:

* **isalpha(str)**: возвращает True, если строка состоит только из алфавитных символов
* **islower(str)**: возвращает True, если строка состоит только из символов в нижнем регистре
* **isupper(str)**: возвращает True, если все символы строки в верхнем регистре
* **isdigit(str)**: возвращает True, если все символы строки - цифры
* **isnumeric(str)**: возвращает True, если строка представляет собой число
* **startwith(str)**: возвращает True, если строка начинается с подстроки str
* **endwith(str)**: возвращает True, если строка заканчивается на подстроку str
* **lower()**: переводит строку в нижний регистр
* **upper()**: переводит строку в вехний регистр
* **title()**: начальные символы всех слов в строке переводятся в верхний регистр
* **capitalize()**: переводит в верхний регистр первую букву только самого первого слова строки
* **lstrip()**: удаляет начальные пробелы из строки
* **rstrip()**: удаляет конечные пробелы из строки
* **strip()**: удаляет начальные и конечные пробелы из строки
* **ljust(width)**: если длина строки меньше параметра width, то справа от строки добавляются пробелы, чтобы дополнить значение width, а сама строка выравнивается по левому краю
* **rjust(width)**: если длина строки меньше параметра width, то слева от строки добавляются пробелы, чтобы дополнить значение width, а сама строка выравнивается по правому краю
* **center(width)**: если длина строки меньше параметра width, то слева и справа от строки равномерно добавляются пробелы, чтобы дополнить значение width, а сама строка выравнивается по центру
* **find(str[, start [, end])**: возвращает индекс подстроки в строке. Если подстрока не найдена, возвращается число -1
* **replace(old, new[, num])**: заменяет в строке одну подстроку на другую
* **split([delimeter[, num]])**: разбивает строку на подстроки в зависимости от разделителя
* **join(strs)**: объединяет строки в одну строку, вставляя между ними определенный разделитель

Например, если мы ожидаем ввод с клавиатуры числа, то перед преобразованием введенной строки в число можно проверить, с помощью метода isnumeric() введено ли в действительности число, и если так, то выполнить операцию преобразования:

|  |  |
| --- | --- |
| 1  2  3  4 | string = input("Введите число: ")  if string.isnumeric():      number = int(string)      print(number) |

Проверка, начинается или оканчивается строка на определенную подстроку:

|  |  |
| --- | --- |
| 1  2  3  4 | file\_name = "hello.py"    starts\_with\_hello = file\_name.startswith("hello")   # True  ends\_with\_exe = file\_name.endswith("exe")           # False |

Удаление пробелов в начале и в конце строки:

|  |  |
| --- | --- |
| 1  2  3 | string = "   hello  world!  "  string = string.strip()  print(string)           # hello  world! |

Дополнение строки пробелами и выравнивание:

|  |  |
| --- | --- |
| 1  2 | print("iPhone 7:", "52000".rjust(10))  print("Huawei P10:", "36000".rjust(10)) |

## Базовые операции

* (сложение)

**>>>** S1 = 'spam'

**>>>** S2 = 'eggs'

**>>> print**(S1 + S2)

'spameggs'

* Дублирование строки

**>>> print**('spam' \* 3)

spamspamspam

* Длина строки (функция len)

**>>>** len('spam')

4

* Доступ по индексу

**>>>** S = 'spam'

**>>>** S[0]

's'

**>>>** S[2]

'a'

**>>>** S[-2]

'a'

Как видно из примера, в Python возможен и доступ по отрицательному индексу, при этом отсчет идет от конца строки.

* Извлечение среза

Оператор извлечения среза: [X:Y]. X – начало среза, а Y – окончание;

символ с номером Y в срез не входит. По умолчанию первый индекс равен 0, а второй - длине строки.

**>>>** s = 'spameggs'

**>>>** s[3:5]

'me'

**>>>** s[2:-2]

'ameg'

**>>>** s[:6]

'spameg'

**>>>** s[1:]

'pameggs'

**>>>** s[:]

'spameggs'

Кроме того, можно задать шаг, с которым нужно извлекать срез.

**>>>** s[::-1]

'sggemaps'

**>>>** s[3:5:-1]

''

**>>>** s[2::2]

'aeg'

## Другие функции и методы строк

При вызове методов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку.

**>>>** s = 'spam'

**>>>** s[1] = 'b'

Traceback (most recent call last):

File "", line 1, in

s[1] = 'b'

TypeError: 'str' object does not support item assignment

**>>>** s = s[0] + 'b' + s[2:]

**>>>** s

'sbam'

Поэтому все строковые методы возвращают новую строку, которую потом следует присвоить переменной.

## Таблица "Функции и методы строк"

| **Функция или метод** | **Назначение** |
| --- | --- |
| **S = 'str'; S = "str"; S = '''str'''; S = """str"""** | [Литералы строк](https://pythonworld.ru/tipy-dannyx-v-python/stroki-literaly-strok.html) |
| **S = "s\np\ta\nbbb"** | Экранированные последовательности |
| **S = r"C:\temp\new"** | Неформатированные строки (подавляют экранирование) |
| **S = b"byte"** | Строка [байтов](https://pythonworld.ru/tipy-dannyx-v-python/bajty-bytes-i-bytearray.html) |
| **S1 + S2** | Конкатенация (сложение строк) |
| **S1 \* 3** | Повторение строки |
| **S[i]** | Обращение по индексу |
| **S[i:j:step]** | Извлечение среза |
| **len**(S) | Длина строки |
| **S.find**(str, [start],[end]) | Поиск подстроки в строке. Возвращает номер первого вхождения  или -1 |
| **S.rfind**(str, [start],[end]) | Поиск подстроки в строке. Возвращает номер последнего  вхождения или -1 |
| **S.index**(str, [start],[end]) | Поиск подстроки в строке. Возвращает номер первого вхождения  или вызывает ValueError |
| **S.rindex**(str, [start],[end]) | Поиск подстроки в строке. Возвращает номер последнего  вхождения или вызывает ValueError |
| **S.replace**(шаблон, замена) | Замена шаблона |
| **S.split**(символ) | Разбиение строки по разделителю |
| **S.isdigit**() | Состоит ли строка из цифр |
| **S.isalpha**() | Состоит ли строка из букв |
| **S.isalnum**() | Состоит ли строка из цифр или букв |
| **S.islower**() | Состоит ли строка из символов в нижнем регистре |
| **S.isupper**() | Состоит ли строка из символов в верхнем регистре |
| **S.isspace**() | Состоит ли строка из неотображаемых символов  (пробел, символ перевода страницы ('\f'),  "новая строка" ('\n'),  "перевод каретки" ('\r'), "горизонтальная табуляция" ('\t') и  "вертикальная табуляция" ('\v')) |
| **S.istitle**() | Начинаются ли слова в строке с заглавной буквы |
| **S.upper**() | Преобразование строки к верхнему регистру |
| **S.lower**() | Преобразование строки к нижнему регистру |
| **S.startswith**(str) | Начинается ли строка S с шаблона str |
| **S.endswith**(str) | Заканчивается ли строка S шаблоном str |
| **S.join**(список) | Сборка строки из списка с разделителем S |
| **ord**(символ) | Символ в его код ASCII |
| **chr**(число) | Код ASCII в символ |
| **S.capitalize**() | Переводит первый символ строки в верхний регистр,  а все остальные в нижний |
| **S.center**(width, [fill]) | Возвращает отцентрованную строку, по краям которой  стоит символ  fill (пробел по умолчанию) |
| **S.count**(str, [start],[end]) | Возвращает количество непересекающихся вхождений  подстроки в диапазоне [начало, конец]  (0 и длина строки по умолчанию) |
| **S.expandtabs**([tabsize]) | Возвращает копию строки, в которой все символы  табуляции  заменяются одним или несколькими пробелами,  в зависимости от текущего столбца. Если TabSize не указан,  размер  табуляции полагается равным 8 пробелам |
| **S.lstrip**([chars]) | Удаление пробельных символов в начале строки |
| **S.rstrip**([chars]) | Удаление пробельных символов в конце строки |
| **S.strip**([chars]) | Удаление пробельных символов в начале и в конце строки |
| **S.partition**(шаблон) | Возвращает кортеж, содержащий часть перед первым  шаблоном,  сам шаблон, и часть после шаблона.  Если шаблон не найден, возвращается кортеж,  содержащий саму  строку, а затем две пустых строки |
| **S.rpartition**(sep) | Возвращает кортеж, содержащий часть перед последним  шаблоном,  сам шаблон, и часть после шаблона.  Если шаблон не найден, возвращается кортеж, содержащий две  пустых строки, а затем саму строку |
| **S.swapcase**() | Переводит символы нижнего регистра в верхний,  а верхнего –  в нижний |
| **S.title**() | Первую букву каждого слова переводит в верхний регистр,  а все остальные в нижний |
| **S.zfill**(width) | Делает длину строки не меньшей width, по необходимости  заполняя первые символы нулями |
| **S.ljust**(width, fillchar=" ") | Делает длину строки не меньшей width, по необходимости  заполняя последние символы символом fillchar |
| **S.rjust**(width, fillchar=" ") | Делает длину строки не меньшей width, по необходимости  заполняя первые символы символом fillchar |
| **S.format**(\*args, \*\*kwargs) | [Форматирование строки](https://pythonworld.ru/osnovy/formatirovanie-strok-metod-format.html) |

**13. Списки. Создание списков. Генераторы списков.**

X = list()

L = input('Введите массив в одну строку: ').split()

Y = [k for k in range(7)]

Y = [k for k in range(7) if k % 3]

L = [L\*2 for L in ‘IU7’]

Для работы с наборами данных Python предоставляет такие встроенные типы как списки, кортежи и словари.

**Список (list)** представляет тип данных, который хранит набор или последовательность элементов. Для создания списка в квадратных скобках ([]) через запятую перечисляются все его элементы. Во многих языках программирования есть аналогичная структура данных, которая называется массив. Например, определим список чисел:

|  |  |
| --- | --- |
| 1 | numbers = [1, 2, 3, 4, 5] |

Также для создания списка можно использовать конструктор **list()**:

|  |  |
| --- | --- |
| 1  2 | numbers1 = []  numbers2 = list() |

Оба этих определения списка аналогичны - они создают пустой список.

Конструктор list для создания списока может принимать другой список:

|  |  |
| --- | --- |
| 1  2 | numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]  numbers2 = list(numbers) |

Для обращения к элементам списка надо использовать индексы, которые представляют номер элемента в списка. Индексы начинаются с нуля. То есть второй элемент будет иметь индекс 1. Для обращения к элементам с конца можно использовать **отрицательные индексы, начиная с -1.** То есть у последнего элемента будет индекс -1, у предпоследнего - -2 и так далее.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | numbers = [1, 2, 3, 4, 5]  print(numbers[0])   # 1  print(numbers[2])   # 3  print(numbers[-3])  # 3    numbers[0] = 125  # изменяем первый элемент списка  print(numbers[0])   # 125 |

Если необходимо создать список, в котором повторяется одно и то же значение несколько раз, то можно использовать символ звездочки \*. Например, определим список из шести пятерок:

|  |  |
| --- | --- |
| 1  2 | numbers = [5] \* 6  # [5, 5, 5, 5, 5, 5]  print(numbers) |

Кроме того, если нам необходим последовательный список чисел, то для его создания удобно использовать функцию **range**, которая имеет три формы:

* range(end): создается набор чисел от 0 до числа end
* range(start, end): создается набор чисел от числа start до числа end
* range(start, end, step): создается набор чисел от числа start до числа end с шагом step

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers = list(range(10))  print(numbers)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  numbers = list(range(2, 10))  print(numbers)      # [2, 3, 4, 5, 6, 7, 8, 9]  numbers = list(range(10, 2, -2))  print(numbers)      # [10, 8, 6, 4] |

Например, следующие два определения списка будут аналогичны, но за счет функции range мы сокращаем объем кода:

|  |  |
| --- | --- |
| 1  2 | numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]  numbers2 = list(range(1, 10)) |

Список необязательно должен содержать только однотипные объекты**. Мы можем поместить в один и тот же список одновременно строки, числа, объекты других типов данных:**

|  |  |
| --- | --- |
| 1 | objects = [1, 2.6, "Hello", True] |

### Перебор элементов

Для перебора элементов можно использовать как цикл for, так и цикл while.

Перебор с помощью цикла for:

|  |  |
| --- | --- |
| 1  2  3 | companies = ["Microsoft", "Google", "Oracle", "Apple"]  for item in companies:      print(item) |

Здесь вместо функции range мы сразу можем подставить имеющийся список companies.

Перебор с помощью цикла while:

|  |  |
| --- | --- |
| 1  2  3  4  5 | companies = ["Microsoft", "Google", "Oracle", "Apple"]  i = 0  while i < len(companies):      print(companies[i])      i += 1 |

Для перебора с помощью функции **len()** получаем длину списка. С помощью счетчика i выводит по элементу, пока значение счетчика не станет равно длине списка.

### Сравнение списков

Два списка считаются равными, если они содержат один и тот же набор элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]  numbers2 = list(range(1,10))  if numbers == numbers2:      print("numbers equal to numbers2")  else:      print("numbers is not equal to numbers2") |

В данном случае оба списка будут равны.

В языке программирования Python существует специальная синтаксическая конструкция, которая позволяет по определенным правилам создавать заполненные списки. Такие конструкции называются **генераторами списков**. Их удобство заключается в более короткой записи программного кода, чем если бы создавался список обычным способом.

Например, надо создать список, заполненный натуральными числами до определенного числа. "Классический" способ будет выглядеть примерно так:

>>> a = []

>>> **for** i **in** range(1,15):

... a.append(i)

...

>>> a

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

На создание списка ушло три строчки кода. Генератор же сделает это за одну:

>>> a = [i **for** i **in** range(1,15)]

>>> a

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

Здесь конструкция [i **for** i **in** range(1,15)] является генератором списка. Вся конструкция заключается в квадратные скобки, что как бы говорит, что будет создан список. Внутри квадратных скобок можно выделить три части: 1) что делаем с элементом (в данном случае ничего не делаем, просто добавляем в список), 2) что берем (в данном случае элемент i), 3) откуда берем (здесь из объекта range). Части отделены друг от друга ключевыми словами **for** и **in**.

Рассмотрим такой пример:

>>> a = [2,-2,4,-4,7,5]

>>> b = [i\*\*2 **for** i **in** a]

>>> b

[4, 4, 16, 16, 49, 25]

В данном случае в генераторе списка берется каждый элемент из списка a и возводится в квадрат. Таким образом, 1) что делаем - возводим элемент в квадрат, 2) что берем - элемент, 3) откуда берем - из списка a.

>>> a = {1:10, 2:20, 3:30}

>>> b = [i\*a[i] **for** i **in** a]

>>> b

[10, 40, 90]

Здесь берется ключ из словаря, а в генерируемый список добавляется произведение ключа на его значение.

>>> a = {1:10, 2:20, 3:30}

>>> b = [[i,a[i]] **for** i **in** a]

>>> b

[[1, 10], [2, 20], [3, 30]]

>>> c = [j **for** i **in** b **for** j **in** i]

>>> c

[1, 10, 2, 20, 3, 30]

В этом примере генерируемый список b состоит из вложенных списков. Если бы в генераторе были опущены квадратные скобки в выражении [i,a[i]], то произошла бы ошибка. Если все же надо получить одноуровневый список из ключей и значений словаря, надо взять каждый вложенный список и из него взять каждый элемент. Это достигается за счет вложенной конструкции **for**, что демонстрируется в строчке c =[j **for** i **in** b **for** j **in** i]. "Классический" синтаксис для заполнения списка c выглядел бы так:

>>> c = []

>>> **for** i **in** b:

... **for** j **in** i:

... c.append(j)

...

>>> c

[1, 10, 2, 20, 3, 30]

В конец генератора можно добавлять конструкцию **if**. Например, надо из строки извлечь все цифры:

>>> a = "lsj94ksd231 9"

>>> b = [int(i) **for** i **in** a **if** '0'<=i<='9']

>>> b

[9, 4, 2, 3, 1, 9]

Или заполнить список числами, кратными 30 или 31:

>>> a = [i **for** i **in** range(30,250) **if** i%30 == 0 **or** i%31 == 0]

>>> a

[30, 31, 60, 62, 90, 93, 120, 124, 150, 155, 180, 186, 210, 217, 240, 248]

Таким образом, генераторы позволяют создавать списки легче и быстрее. Однако заменить ими достаточно сложные конструкции не получится. Например, когда условие проверки должно включать ветку **else**.

**14. Списки. Основные методы для работы с элементами списка. Добавление элемента, вставки, удаление, поиск.**

L.append(), L.extend(s) – вставляет в конец L список s, L.reverse(), L.index(), L.insert(i,x), L.count() , L.remove(), L.pop(),L.copy()

### Методы и функции по работе со списками

Для управления элементами списки имеют целый ряд методов. Некоторые из них:

* **append(item)**: добавляет элемент item в конец списка
* **insert(index, item)**: добавляет элемент item в список по индексу index
* **remove(item)**: удаляет элемент item. Удаляется только первое вхождение элемента. Если элемент не найден, генерирует исключение ValueError
* **clear()**: удаление всех элементов из списка
* **index(item)**: возвращает индекс элемента item. Если элемент не найден, генерирует исключение ValueError
* **pop([index])**: удаляет и возвращает элемент по индексу index. Если индекс не передан, то просто удаляет последний элемент.
* **count(item)**: возвращает количество вхождений элемента item в список
* **sort([key])**: сортирует элементы. По умолчанию сортирует по возрастанию. Но с помощью параметра key мы можем передать функцию сортировки.
* **reverse()**: расставляет все элементы в списке в обратном порядке

Кроме того, Python предоставляет ряд встроенных функций для работы со списками:

* **len(list)**: возвращает длину списка
* **sorted(list, [key])**: возвращает отсортированный список
* **min(list)**: возвращает наименьший элемент списка
* **max(list)**: возвращает наибольший элемент списка
* **copy()**: копирует список
* **l.extend(s)**

### Добавление и удаление элементов

Для добавления элемента применяются методы append() и insert, а для удаления - методы remove(), pop() и clear().

### Проверка наличия элемента

Если определенный элемент не найден, то методы remove и index генерируют исключение. Чтобы избежать подобной ситуации, перед операцией с элементом можно проверять его наличие с помощью ключевого слова **in**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | companies = ["Microsoft", "Google", "Oracle", "Apple"]  item = "Oracle"  # элемент для удаления  if item in companies:      companies.remove(item)    print(companies) |

Выражение item in companies возвращает True, если элемент item имеется в списке companies. Поэтому конструкция if item in companiesможет выполнить последующий блок инструкций в зависимости от наличия элемента в списке.

**15. Списки. Основные операции со списками. Поиск минимального элемента. Поиск максимального элемента. Нахождение количества элементов. Нахождение суммы и произведения элементов.**

Max min len

### **Минимальное и максимальное значения**

Встроенный функции Python **min()**и**max()** позволяют найти минимальное и максимальное значения соответственно:

|  |  |
| --- | --- |
| 1  2  3 | numbers = [9, 21, 12, 1, 3, 15, 18]  print(min(numbers))     # 1  print(max(numbers))     # 21  for i in range(6):  a = float(input('enter number '))  if a > max:  max = a  if a < min:  min = a |

Чтобы найти размер списка, используйте встроенную функцию **len**:

**Нахождение суммы и произведения элементов:**

a = [0]\*10  
s = 0  
m = 1  
**for** i **in** range(10):  
    a[i] = float(input())  
    s += a[i]  
    m \*= a[i]  
**print**(a)  
**print**(s)  
**print**('%.4f'%m)

Для поиска суммы существует стандартная функция **sum**.

S=sum(list)

**16. Списки. Использование срезов при обработке списков**

Очень часто, надо получить не один какой-то элемент, а некоторый их набор ограниченный определенными простыми правилами — например первые 5 или последние три, или каждый второй элемент — в таких задачах, вместо перебора в цикле намного удобнее использовать так называемый **срез (slice).**  
  
Следует помнить, что взяв элемент по индексу или срезом (slice**) мы не как не меняем исходную коллекцию, мы просто скопировали ее часть для дальнейшего использования** (например добавления в другую коллекцию, вывода на печать, каких-то вычислений). Поскольку сама коллекция не меняется — это применимо как к изменяемым (список) так и к неизменяемым (строка, кортеж) последовательностям.  
  
Синтаксис среза похож на таковой для индексации, но в квадратных скобках вместо одного значения указывается 2-3 через двоеточие:  
my\_collection[start:stop:step] *# старт, стоп и шаг*

**Особенности среза:**

* Отрицательные значения старта и стопа означают, что считать надо не с начала, а с конца коллекции.
* Отрицательное значение шага — перебор ведём в обратном порядке справа налево.
* Если не указан старт **[:stop:step]**— берём с самого начала коллекции, то есть start = 0  
  Если не указан стоп **[start:: step]** — идем до самого конца коллекции, то есть stop = 0
* step = 1, то есть последовательный перебор слева направо указывать не обязательно — это значение шага по умолчанию. В таком случае достаточно указать **[start:stop]**
* Можно сделать даже так [:] — это значит взять коллекцию целиком
* **ВАЖНО**: При срезе, первый индекс входит в выборку, а второй нет! То есть от старта включительно, до стопа, где стоп не включается в результат. Математически это можно было бы записать как [start, stop)

**Примеры срезов в виде таблицы:**

|  |  |
| --- | --- |
| R = x[:] | создает копию списка: [1,2,3,4,5] |
| R1 = x[0:3] | формируется список c 0-го элемента х по : x[0], x[1], x[2]: [1,2,3] |
| R2 = x[1:] | выбираются все элементы, кроме первого: [2,3,4,5] |
| R3 = x[:3] | первые 3 элемента массива: [1,2,3] |
| R4 = x[:-1] | все элементы, кроме последнего: [1,2,3,4] |
| R5 = x[-3:] | три последних элемента: [3,4,5] |
| R6 = x[1:-2] | отсутствуют первый элементы и два последних: [2,3] |

**Чтобы избавится от «магических констант», особенно в случае, когда один и тот же срез надо применять многократно, можно задать константы с именованными срезами с пользованием специальной функции slice()()**

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

**EVEN = slice(1, None, 2)**

**print(my\_list[EVEN]) *# [2, 4, 6]***

Важный момент, на котором не всегда заостряется внимание — с помощью среза можно не только получать копию коллекции, но в случае списка можно также менять значения элементов, удалять и добавлять новые.

#### **Проиллюстрируем это на примерах ниже:**

* Даже если хотим добавить один элемент, необходимо передавать итерируемый объект, иначе будет ошибка TypeError: can only assign an iterable
* my\_list = [1, 2, 3, 4, 5]
* *# my\_list[1:2] = 20 # TypeError: can only assign an iterable*
* my\_list[1:2] = [20] *# Вот теперь все работает*
* print(my\_list) *# [1, 20, 3, 4, 5]*
* Для вставки одиночных элементов можно использовать срез, код примеров есть ниже, но делать так не рекомендую, так как такой синтаксис хуже читать. Лучше использовать методы списка .append() и .insert():

Срез аналоги .append() и insert()

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

my\_list[5:] = [6] *# вставляем в конец — лучше использовать .append(6)*

print(my\_list) *# [1, 2, 3, 4, 5, 6]*

my\_list[0:0] = [0] *# вставляем в начало — лучше использовать .insert(0, 0)*

print(my\_list) *# [0, 1, 2, 3, 4, 5, 6]*

my\_list[3:3] = [25] *# вставляем между элементами — лучше использовать .insert(3, 25)*

print(my\_list) *# [0, 1, 2, 25, 3, 4, 5, 6]*

* Можно менять части последовательности — это применение выглядит наиболее интересным, так как решает задачу просто и наглядно.
* my\_list = [1, 2, 3, 4, 5]
* my\_list[1:3] = [20, 30]
* print(my\_list) *# [1, 20, 30, 4, 5]*
* my\_list[1:3] = [0] *# нет проблем заменить два элемента на один*
* print(my\_list) *# [1, 0, 4, 5]*
* my\_list[2:] = [40, 50, 60] *# или два элемента на три*
* print(my\_list) *# [1, 0, 40, 50, 60]*
* Можно просто удалить часть последовательности
* my\_list = [1, 2, 3, 4, 5]
* my\_list[:2] = [] *# или del my\_list[:2]*
* print(my\_list) *# [3, 4, 5]*

**Выход за границы индекса**

Обращение по индексу по сути является частным случаем среза, когда мы обращаемся только к одному элементу, а не диапазону. Но есть очень важное отличие в обработке ситуации с отсутствующим элементом с искомым индексом.  
  
Обращение к несуществующему индексу коллекции вызывает ошибку:

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

print(my\_list[-10]) *# IndexError: list index out of range*

print(my\_list[10]) *# IndexError: list index out of range*

**А в случае выхода границ среза за границы коллекции никакой ошибки не происходит:**

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

print(my\_list[0:10]) *# [1, 2, 3, 4, 5] — отработали в пределах коллекции*

print(my\_list[10:100]) *# [] - таких элементов нет — вернули пустую коллекцию*

print(my\_list[10:11]) *# [] - проверяем 1 отсутствующий элемент - пустая коллекция, без ошибки*

**17. Списки. Сортировка элементов списка.**

**a=sorted(a,reverse=True, key=funcname (например len))**

**a=sorted(a) #по возрастанию**

**b=list(reversed(b))**

**c.sort()**

**d.reverse()**

**c.sort(reverse=True)**

### **Функция sorted()**

Мы может использовать **функцию sorted()** для вывода списка сортированных элементов любой коллекции для последующее обработки или вывода.

* функция не меняет исходную коллекцию, а возвращает новый список из ее элементов;
* не зависимо от типа исходной коллекции, вернётся список (list) ее элементов;
* поскольку она не меняет исходную коллекцию, ее можно применять к неизменяемым коллекциям;
* Поскольку при сортировке возвращаемых элементов нам не важно, был ли у элемента некий индекс в исходной коллекции, можно применять к неиндексированным коллекциям;
* Имеет дополнительные не обязательные аргументы:  
  **reverse=True** — сортировка в обратном порядке  
  **key=funcname** (начиная с Python 2.4) — сортировка с помощью специальной функции funcname, она может быть как стандартной функцией Python, так и специально написанной вами для данной задачи функцией и лямбдой.

**индексированные коллекции** (их еще называют последовательности — sequences) — **список (list), кортеж (tuple), строку (string).**  
Под индексированностью имеется ввиду, что элементы коллекции располагаются в определённом порядке, каждый элемент имеет свой индекс от 0.

**New=sorted(c)**

my\_list = [2, 5, 1, 7, 3]

my\_list\_sorted = sorted(my\_list)

print(my\_list\_sorted) *# [1, 2, 3, 5, 7]*

my\_set = {2, 5, 1, 7, 3}

my\_set\_sorted = sorted(my\_set, reverse=True)

print(my\_set\_sorted) *# [7, 5, 3, 2, 1]*

Пример сортировки списка строк по длине len() каждого элемента:

my\_files = ['somecat.jpg', 'pc.png', 'apple.bmp', 'mydog.gif']

my\_files\_sorted = sorted(my\_files, key=len)

print(my\_files\_sorted) *# ['pc.png', 'apple.bmp', 'mydog.gif', 'somecat.jpg']*

**a=sorted(a,reverse=True)**

**a=sorted(a) #по возрастанию**

**b=list(reversed(b))**

**c.sort()**

**d.reverse()**

**c.sort(reverse=True)**

**Функция reversed()**

Функция reversed() применяется для последовательностей и работает по другому:

* возвращает генератор списка, а не сам список;
* **если нужно получить не генератор, а готовый список, результат можно обернуть в list() или же вместо reversed() воспользоваться срезом [: :-1];**
* она **не сортирует** элементы, а возвращает их в обратном порядке, то есть читает с конца списка;
* **не применима к неиндексированным коллекциям**
* не позволяет использовать дополнительные аргументы — будет ошибка «TypeError: reversed() does not take keyword arguments».

**new=list(reserved(mylist))**

my\_list = [2, 5, 1, 7, 3]

my\_list\_sorted = reversed(my\_list)

print(my\_list\_sorted) *# <listreverseiterator object at 0x7f8982121450>*

**!!!!!print(list(my\_list\_sorted)) *# [3, 7, 1, 5, 2]***

**print(my\_list[::-1]) *# [3, 7, 1, 5, 2] - тот же результат с помощью среза***

**Методы списка .sort() и .reverse()**

**У списка (и только у него)** есть особые методы .**sort() и .reverse()** которые делают тоже самое, что соответствующие функции sorted() и reversed(), но при этом:

* **Меняют сам исходный список, а не генерируют новый;**
* Возвращают None, а не новый список;
* поддерживают те же **дополнительные аргументы;**
* в них не надо передавать сам список первым параметром, более того, если это сделать — будет ошибка — не верное количество аргументов.

my\_list = [2, 5, 1, 7, 3]

my\_list.sort()

print(my\_list) *# [1, 2, 3, 5, 7]*

**a=sorted(a,reverse=True)**

**a=sorted(a) #по возрастанию**

**b=list(reversed(b))**

**c.sort()**

**d.reverse()**

**c.sort(reverse=True)**

**18. Списки. Сортировка. Сортировка вставками. Сортировка выбором.**

**1)Сортировка выбором** — здесь, чтобы отсортировать массив, находим элемент с минимальным значением, затем сравниваем его со значением первой неотсортированной позиции. Если этот элемент меньше, то он становится новым минимумом и их позиции меняются.

 Как и при сортировке вставками, упорядоченная часть массива расположена в начале, в то время как в пузырьковой сортировке она находится в конце.  
  
Пример реализации

Код:

def ssort(array):

for i in range(len(array)):

indxMin = i

for j in range(i+1, len(array)):

if array[j] < array[indxMin]:

indxMin = j

tmp = array[indxMin]

array[indxMin] = array[i] # **array[indxMin], array[i] = array[i], array[indxMin]**

array[i] = tmp

return a

**2)Сортировка вставками:**

элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов.

Код сортировки:

Код:

def isort(array):

for i in range(len(array)):

v = array[i]

j = i

while (array[j-1] > v) and (j > 0):

array[j] = array[j-1]

j = j - 1

array[j] = v

return array

3)**Cортировка Вставками с барьером:**

Для того чтобы сократить количество сравнений, производимых нашей программой,

дополним сортируемый массив нулевой компонентой

и будем записывать в нее поочередно каждый вставляемый элемент . В тех случаях,

когда вставляемое значение окажется меньше, чем a[1], компонента a[0] будет

работать как "барьер", не дающий индексу j выйти за нижнюю границу массива.

Кроме того, компонента a[0] может заменить собою и дополнительную переменную х.

Код:

a1=[3,-5,6,2,1,-100,12,10]

a=[0]\*(len(a1)+1)

a[1:]=a1[:]

print(a)

N=len(a)

for i in range(2,N):

if a[i-1]>a[i]:

a[0]= a[i]

j= i

while a[j-1]>a[0] and j>0:

a[j]= a[j-1]

j= j-1

a[j]= a[0]

print(a[1:])

**???19. Списки. Сортировка вставками. Метод простых вставок. Метод вставок с бинарным поиском. Метод Шелла.+Вставки с барьером.**

***1)Бинарный поиск***

Этот алгоритм представляет из себя оптимизированную версию предыдущего, отличие заключается в том, что при поиске места, на которое надо вставить элемент ai в уже упорядоченную совокупность a0 , …, ai-1 , определяется алгоритмом деления пополам (отсюда и название алгоритма «бинарные вставки» здесь понимаем как «вставка делением пополам»).

(можно ли insert)????

for i in range(1, len(a)):

if a[i] > a[i - 1]:

continue

else:

mn=0

mx=i

while mn < mx:

t = mn + ((mx - mn)//2)

if a[i] > a[t]:

mn = t + 1

else:

mx = t

a.insert(mn, a.pop(i))

**или:**

**(без insert):**

**for i in range (len(a)):**

**x=a[i] #{запомним элемент}**

**l=0 #{левый край}**

**r=i-1 #{правый}**

**while l<=r : # {пока левый не больше правого}**

**m=(l+r)//2 #{находим середину}**

**if x < a[m]:**

**r=m-1 #{если элемент меньше среднего,**

**#правый край левее середины}**

**else:**

**l=m+1 #{иначе левый правее середины}**

**for j in range (i-1,l-1,-1):**

**a[j+1]=a[j] #{сдвигаем массив вправо на 1}**

**a[l]=x #{вставляем элемент на место}**

**2) Сортировка методом шелла**

Сортировка Шелла является несколько измененным вариантом сортировки вставками.  
Сортировка вставками является медленной из-за того, что совершает перемещения только с соседними элементами, в отличии от сортировки Шелла, которая позволяет быстро сделать обмен между элементами, которые находятся далеко друг от друга.  
  
При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d{\displaystyle d} {\displaystyle d}. После этого процедура повторяется для некоторых меньших значений {\displaystyle d}d, а завершается сортировка Шелла упорядочиванием элементов при {\displaystyle d=1} d=1 (то есть обычной [сортировкой вставками](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%BE%D0%B9)). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, [пузырьковой](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BC_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%B0), каждая перестановка двух элементов уменьшает количество [инверсий](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F_(%D0%BF%D0%B5%D1%80%D0%B5%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B0)) в списке максимум на 1, а при сортировке Шелла это число может быть больше).

*Код:*

#### def Shell(A):

#### t = (len(A)//2)

#### while t > 0:

#### for i in range(len(A)-t):

#### j = i

#### while j >= 0 and A[j] > A[j+t]:

#### A[j], A[j+t] = A[j+t], A[j]

#### j -= 1

#### t = (t//2)

**20. Списки. Сортировка. Пузырьковые методы сортировки. Сортировка пузырьком. Сортировка пузырьком с барьером. Метод шейкер-сортировки.**

1)Сортировка методом пузырька:

Суть алгоритма в том, что совершается несколько проходов по массиву. При проходе последовательно сравниваются пары элементов в массиве и в случае несоответствия выбранному порядку меняются местами. Если пары элементов находятся в верном порядке, то ничего не происходит. В результате первого прохода максимальный элемент окажется в конце, то есть всплывет словно пузырек. Затем все повторяется до того момента пока весь массив не будет отсортирован.Последний проход будет по отсортированному массиву.  
Пример кода:

##### Код:

def BubbleSort(A):  
    for j in range(len(A) - 1):  
        for i in range(len(A) - 1 - j):  
            if A[i] > A[i + 1]:  
                A[i], A[i + 1] = A[i + 1], A[i]

**2) Сортировка пузырьком с барьером:**

 если за полный проход в массиве не сделано ни одной перестановки, то его можно считать отсортированным. Что значит «не сделано ни одной перестановки»? Это значит, что все пары соседних чисел расположены «правильно», то есть большее число идет позже меньшего, поэтому они в перестановках не нуждаются. Это позволяет значительно сократить время в случаях, когда более или менее повезло с исходными данными.

Например, в массиве 8, 1, 2, 3, 4, 5, 6 будет вообще достаточно одного прохода, чтобы вытолкнуть восьмерку на последнее место.

def BubbleSort(A):  
    j = len(A) - 1  
    IsNotOrdered = True  
    while IsNotOrdered:  
        IsNotOrdered = False  
        for i in range(0, j):  
            if A[i] > A[i + 1]:  
                A[i], A[i + 1] = A[i + 1], A[i]  
                IsNotOrdered = True  
        j -= 1

Переменная **j** здесь служит для того, чтобы прервать проходы по списку, как только ее значение приблизится к размеру длины строки. Также цикл **for** благодаря **j** сокращается при каждом последующем проходе по **while**. Это оптимизирует алгоритм: последние элементы не просматриваются. Элементы меняются местами лишь в случае, если предыдущий элемент больше последующего.

**3) Шейкер-сортировка:**

Дело в том, что сейчас мы сортируем массив в одном направлении, а можем попробовать поочередно менять направления сортировки. Так мы сможем добиться чтобы элементы (которые далеко находятся от своих отсортированных позиций) быстро смогли занять свои правильные места.

Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства.

***Во-первых***, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения.

***Во-вторых***, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки**. Границы рабочей части массива (то есть части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.**

Лучший случай для этой сортировки — отсортированный массив {\displaystyle O(n)} худший — отсортированный в обратном порядке/{\displaystyle O(n^{2})}Наименьшее число сравнений в алгоритме Шейкер-сортировки {\displaystyle C=N-1}. Это соответствует единственному проходу по упорядоченному массиву (лучший случай)

def shakerSort(a):

#lb, ub границы неотсортированной части массива

k= ub = len(a)-1

lb=1

while ( lb < ub ):

# проход сверху вниз

for j in range (ub, lb-1, -1):

if a[j-1] > a[j]:

a[j-1], a[j] = a[j], a[j-1]

k=j

lb = k

# проход снизу вверх

for j in range (lb, ub+1):

if a[j-1] > a[j]:

a[j-1], a[j] = a[j], a[j-1]

k=j

ub = k

return a

**21. Списки. Сортировка. Метод быстрой сортировки.**

**Быстрая сортировка (Сортировка Хоара):**

#### Одна из самых быстрых сортировок. Идея алгоритма заключается в том, что выбирается опорный элемент, относительно которого будет происходит сортировка. Равные и бОльшие элементы помещаются справа, меньшие – слева. Затем к полученным подмассивам рекурсивно применяются два первых пункта.

* ШАГ 1: Возьмем элемент A[p] за ось и "раскидаем" остальные элементы A[(p+1)..q] по разные стороны от него стороны — меньшие влево, большие --- вправо, то есть переставим элементы подмассива A[p..q] так, чтобы вначале шли элементы меньше либо равные A[p] потом элементы, больше либо равные A[p]. Назовет этот шаг разделением (partition).

* ШАГ 2: Пусть r есть новый индекс элемента A[p]. Тогда, если q - p > 2, вызовем функцию сортировки для подмассивов A[p..(r-1)] и A[(r+1)..q].

Ключевая идея алгоритма заключается в процедуре «partition», которая за линейное время от размера массива, осуществляет такую перестановку элементов, относительно некоторой «оси» — заданного значения, равного одному из значений сортируемого интервала массива, что переставленный массив состоит из трех интервалов, идущих по порядку:

1. Элементы меньшие «оси»
2. Элементы равные «оси»
3. Элементы большие «оси»

**Первый и последний из упомянутых интервалов могут быть неупорядоченными, поэтому далее, они рекурсивно сортируются.**

Эффективность алгоритма существенно зависит от выбора «оси». Например**, если назначать «осью» значение, которое больше или меньше всех элементов, то алгоритм вовсе не будет работать — разбиение никак не будет уменьшать сортируемый интервал и алгоритм зациклится.**

**import** random

**from** random **import** randint

*# процедура*

**def** qSort ( A, nStart, nEnd ):

**if** nStart >= nEnd: **return**

L = nStart; R = nEnd

X = A[(L+R)//2]

**while** L <= R:

**while** A[L] < X: L += 1 *# разделение*

**while** A[R] > X: R -= 1

**if** L <= R:

A[L], A[R] = A[R], A[L]

L += 1; R -= 1

qSort ( A, nStart, R ) *# рекурсивные вызовы*

qSort ( A, L, nEnd )

N=10

A = [randint(1,10) **for** i **in** range(N)]

**print**(A)

*# вызов процедуры*

qSort ( A, 0, N-1 )

**print**('отсортированный', A)

**22. Множества. Определение. Основные операции.**

Множество в python - "контейнер", содержащий не повторяющиеся элементы в случайном порядке.

a = set() – создание множества

Операции:

* **len(s)** - число элементов в множестве (размер множества).
* **x in s** - принадлежит ли x множеству s.
* **set.isdisjoint**(other) - истина, если set и other не имеют общих элементов.
* **set == other** - все элементы set принадлежат other, все элементы other принадлежат set.
* **set.issubset**(other) или **set <= other** - все элементы set принадлежат other.
* **set.union**(other, ...) или **set | other | ...** - объединение нескольких множеств.
* **set.intersection**(other, ...) или **set & other & ...** - пересечение.
* **set.difference**(other, ...) или **set - other - ...** - Возвращает **set**, разницу между множествами
* **set.update**(other, ...); set |= other | ... - объединение.
* **set.add**(elem) - добавляет элемент в множество.
* **set.remove**(elem) - удаляет элемент из множества. KeyError, если такого элемента не существует.
* **set.discard**(elem) - удаляет элемент, если он находится в множестве.
* **set.pop**() - удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым.
* **set.clear**() - очистка множества.

**Set() – изменяемый кортеж, frozenset() – неизменяемый.**

**union** в качестве результата возвращает новое множество, не меняя исходные, а **update** ничего не возвращает, но добавит в первое множество элементы второго.

**Множеством** называется неупорядоченная совокупность **уникальных** значений. В качестве элементов этого набора данных могут выступать любые **неизменяемые объекты**, такие как числа, символы, строки. В отличие от массивов и списков, порядок следования значений не учитывается при обработке его содержимого

создается множество целых чисел под названием a

a = {1, 2, 0, 1, 3, 2} #создание

{0, 1, 2, 3}

Как можно заметить, все элементы полученной последовательности являются уникальными, без повторений.

метод set. Аргументом этой функции может быть набор неких данных или даже строка с текстом, как это показано в следующем примере.

a = set('data')

print(a)

{'d', 'a', 't'}

В результате выполнения этого кода, программа заполняет новое множество уникальными символами из входной строки. Содержимое набора также выводится на экран.

## Использование

Обычно используется для следующих операций:

* Проверка, есть ли данное значение в множестве. Для этого используется in.
* a = {0, 1, 2, 3}
* print(2 in a)
* True
* Наоборот, проверка отсутствия. Используется not in.
* a = {0, 1, 2, 3}
* print(2 not in a)
* False
* Перебор всех элементов.
* for a in {0, 1, 2}:
* print(a)
* 0
* 1
* 2

## Генератор

Для создания множества можно в Python воспользоваться генератором, позволяющих заполнять списки, а также другие наборы данных с учетом неких условий. Следующий код демонстрирует генерацию множества a с циклом for для нескольких чисел.

a = {i for i in [1, 2, 0, 1, 3, 2]}

print(a)

{0, 1, 2, 3}

## Изменение множеств

Для управления содержимым множеств в языке Python присутствуют специальные методы, дающие возможность добавлять и удалять отдельные элементы.

### **Получение размера**

Узнать точное количество элементов, входящих в состав множества, поможет метод len, принимающий в качестве аргумента набор данных. Функция print выводит результат.

a = {0, 1, 2, 3}

print(len(a))

4

### **Добавление элемента**

Чтобы внести новые значения, потребуется вызывать метод add. Аргументом в данном случае будет добавляемый элемент последовательности. В примере кода на Python добавим в множество элемент со значением 4.

a = {0, 1, 2, 3}

a.add(4)

print(a)

{0, 1, 2, 3, 4}

### Удаление элемента

**Для удаления элементов из множества используются следующие функции в Python**(кроме очистки, которая будет рассмотрена ниже):

* remove — удаление элемента с генерацией исключения в случае, если такого элемента нет;
* discard — удаление элемента без генерации исключения, если элемент отсутствует;
* pop — удаление первого элемента, генерируется исключение при попытке удаления из пустого множества.

Избавиться от лишних значений в наборе данных с помощью remove. В качестве входного параметра здесь выступает элемент, который нужно удалить (в примере удалим число со значением 3).

a = {0, 1, 2, 3}

a.remove(3)

print(a)

{0, 1, 2}

По поводу функции pop хотелось бы отметить, что так как множества не упорядочены, то удалится случайный элемент, который будет находиться в памяти первым. Но если они в памяти хранятся в отсортированном виде, что не факт, то скорее всего будет удален элемент с наименьшим значением. Но на это не стоит рассчитывать.

### Полная очистка

Иногда необходимо полностью убрать все элементы. Чтобы не удалять каждый элемент отдельно, используется метод **clear,** не принимающий аргументов. Если вывести содержимое после этой операции, на экране появится только его название.

a = {0, 1, 2, 3}

a.clear()

print(a)

set()

В результате получили пустое множество.

## Сортировка

Порядок следования элементов не учитывается. Поэтому нет смысла говорить о сортировке множеств в Python 3.

Но с другой стороны все дело обстоит не со всем так. Для быстрого поиска элемента, желательно их хранить в памяти в упорядоченном виде.

В начале рассмотрим, что будет с элементами разных типов данных в одном множестве. Такие элементы не должны сортироваться. Если мы будем выводить элементы с помощью команды print, то они выводятся примерно следующим образом:

a = {0, 1, 12, 'b', 'ab', 3, 2, 'a'}

print(a)

{0, 1, 'b', 3, 2, 12, 'ab', 'a'}

Как видим, у нас вывелись не отсортированные значения, если повторить запуск, то порядок будет меняться. Но это только в том случае, если перемешаны элементы разного типа.

Посмотрим, что будет, если попытаемся вывести только числа:

a = {0, 1, 12, 3, 2}

print(a)

{0, 1, 2, 3, 12}

Все элементы выведены упорядоченно. Теперь посмотрим что будет если преобразовать в список:

a = {0, 1, 12, 3, 2}

b = list(a)

print(b)

[0, 1, 2, 3, 12]

Аналогично, в список значения записались отсортированными по возрастанию. **.**

**Получается, что элементы хранятся в памяти в упорядоченном виде, если они одного типа**

Если вам нужно получить отсортированный список из множества, лучше воспользоваться для верности функцией sort. Элементы будут точно отсортированы. Ваш код будет понятен для других.

## Операции над множествами

### **Объединение**

Чтобы объединить все элементы двух разных множеств, стоит воспользоваться методом **union** на одном из объектов.

a = {0, 1, 2, 3}

b = {4, 3, 2, 1}

c = a.union(b)

print(c)

{0, 1, 2, 3, 4}

### Добавление

Чтобы добавить все элементы из одного множества к другому, необходимо вызывать метод update на первом объекте. Таким образом можно перенести уникальные данные из одного набора чисел в другой, как это показано в следующем примере.

a = {0, 1, 2, 3}

b = {4, 3, 2, 1}

a.update(b)

print(a)

{0, 1, 2, 3, 4}

### **Пересечение**

Чтобы найти общие элементы для двух разных множеств, следует применить функцию **intersection,** принимающую в качестве аргумента один из наборов данных

a = {0, 1, 2, 3}

b = {4, 3, 2, 1}

c = a.intersection(b)

print(c)

{1, 2, 3}

### **Разность**

Чтобы вычислить разность для двух разных множеств, необходимо воспользоваться методом **difference.** Функция позволяет найти элементы, уникальные для второго набора данных, которых в нем нет. Следующий код демонстрирует эту операцию.

a = {0, 1, 2, 3}

b = {4, 3, 2, 1}

c = a.difference(b)

print(c)

{0}

## Отношения между множествами

Для определения подмножеств и надмножеств существуют специальные функции, возвращающие True или False в зависимости от результата выполнения.

### Определение подмножества

Чтобы выяснить, является ли множество a подмножествомb, стоит попробовать вывести на экран результат выполнения метода issubset, как в следующем примере. Так как не все элементы набора чисел a присутствуют в b, функция вернет False.

a = {0, 1, 2, 3, 4}

b = {3, 2, 1}

print(a.issubset(b))

False

### **Определение надмножества**

Чтобы узнать, является ли множество a надмножеством b, необходимо вызвать **метод issuperset** и вывести результат его работы на экран. Поскольку все элементы набора чисел b присутствуют в a, функция возвращает True.

a = {0, 1, 2, 3, 4}

b = {3, 2, 1}

print(a.issuperset(b))

True

## Тип frozenset

Множество, содержимое которого не поддается изменению имеет тип **frozenset.** Значения из этого набора нельзя удалить, как и добавить новые. В следующем примере демонстрируется создание при помощи стандартной функции.

a = frozenset({"hello", "world"})

print(a)

frozenset({'hello', 'world'})

Поскольку содержимое frozenset должно всегда оставаться статичным, перечень функций, с которыми такое множество может взаимодействовать, имеет ограничения.

## Преобразование множеств

Иногда возникает необходимость представления уже готовой последовательности значений в качестве совсем другого типа данных. Возможности языка позволяют конвертировать любое множество в строку, словарь или список при помощи стандартных функций.

### **Строка**

Для преобразования множества в строку используется конкатенация текстовых значений, которую обеспечивает функция join. В этом случае ее аргументом является набор данных в виде нескольких строк. Запятая в кавычках выступает в качестве символа, разделяющего значения. Метод type возвращает тип данных объекта в конце приведенного кода.

a = {'set', 'str', 'dict', 'list'}

b = ','. join(a)

print(b)

print(type(b))

set,dict,list,str

<class 'str'>

### Словарь

Чтобы получить из множества [словарь](https://all-python.ru/osnovy/slovari.html), следует передать функции dict набор из нескольких пар значений, в каждом из которых будет находиться ключ. Метод print демонстрирует на экране содержимое полученного объекта, а type отображает его тип.

a = {('a', 2), ('b', 4)}

b = dict(a)

print(b)

print(type(b))

{'b': 4, 'a': 2}

<class 'dict'>

Следует отметить, что каждый элемент для такого преобразования – [кортеж](https://all-python.ru/osnovy/kortezh.html) состоящий из двух значений:

1. ключ будущего словаря;
2. значение, соответствующее ключу.

### **Список**

По аналогии с предыдущими преобразованиями можно получить [список](https://all-python.ru/osnovy/spiski.html) неких объектов. На этот раз используется метод list, получающий в качестве аргумента множество a. На выходе функции print отображаются уникальные значения для изначального набора чисел.

a = {1, 2, 0, 1, 3, 2}

b = list(a)

print(b)

print(type(b))

[0, 1, 2, 3]

<class 'list'>

| Название | Назначение |
| --- | --- |
| Len | Получение размера |
| Add | Добавление элемента |
| Remove | Удаление элемента |
| Clear | Очистка |
| Union | Объединение |
| Update | Добавление всех элементов одного множества в другое |
| Intersection | Нахождение множества, элементы которого находятся на пересечении двух множеств |
| Difference | Нахождение множества, элементы которого входят в первое, но не входят во второе множество |
| Issubset | Проверка, является ли множество подмножеством |
| Issuperset | Проверка, является ли множество надмножеством |

**union** в качестве результат возвращает новое множество, не меняя исходные, а **update** ничего не возвращает, но добавит в первое множество элементы второго.

**23. Словари. Понятие ключей и значений. Создание словарей. Основные методы словарей.**

**Словарь (dict)** представляет собой структуру данных (которая ещё называется ассоциативный массив), предназначенную для хранения произвольных объектов с доступом по ключу. Данные в словаре хранятся в формате **ключ – значение.**

В словаре аналогом индекса является ключ, при этом ответственность за его формирование ложится на программиста.

## Создание, изменение, удаление словарей и работа с его элементами

**Создание словаря**Пустой словарь можно создать, используя функцию dict(), либо просто указав пустые фигурные скобки.

>>> d1 = dict()

>>> print(type(d1))

<*class* 'dict'>

>>> d2 = {}

>>> print(type(d2))

<*class* 'dict'>

Если необходимо создать словарь с заранее подготовленным набором данных, то можно использовать один из перечисленных выше подходов, но с перечислением групп ключ-значение.

>>> d1 = dict(Ivan="manager", Mark="worker")

>>> print(d1)

{'Mark': 'worker', 'Ivan': 'manager'}

>>> d2 = {"A1":"123", "A2":"456"}

>>> print(d2)

{'A2': '456', 'A1': '123'}

Добавление и удаление элемента  
Чтобы **добавить** элемент в словарь нужно указать новый ключ и значение.

>>> d1 = {"Russia":"Moscow", "USA":"Washington"}

>>> d1["China"]="Beijing"

>>> print(d1)

{' Russia': 'Moscow', 'China': 'Beijing', 'USA': 'Washington'}

Для **удаления** элемента из словаря можно воспользоваться командой **del.**

>>> d2 = {"A1":"123", "A2":"456"}

>>> del d2["A1"]

>>> print(d2)

{'A2': '456'}

Работа со словарем  
Проверка наличия ключа в словаре производится с помощью оператора **in.**

>>> d2 = {"A1":"123", "A2":"456"}

>>> "A1" in d2

**True**

>>> "A3" in d2

**False**

Доступ к элементу словаря, осуществляется как же как доступ к элементу списка, только в качестве индекса указывается ключ.

>>> d1 = {"Russia":"Moscow", "USA":"Washington"}

>>> d1["Russia"]

'Moscow'

## Методы словарей

У словарей доступен следующий набор методов.

**clear()**  
Удаляет все элементы словаря.

>>> d2 = {"A1":"123", "A2":"456"}

>>> print(d2)

{'A2': '456', 'A1': '123'}

>>> d2.clear()

>>> print(d2)

{}

**copy()**  
Создается новая копия словаря.

>>> d2 = {"A1":"123", "A2":"456"}

>>> d3 = d2.copy()

>>> print(d3)

{'A1': '123', 'A2': '456'}

>>> d3["A1"]="789"

>>> print(d2)

{'A2': '456', 'A1': '123'}

>>> print(d3)

{'A1': '789', 'A2': '456'}

**fromkeys(seq[, value])**  
Создает новый словарь с ключами из seq и значениями из value. По умолчанию value присваивается значение None.

***get(key)***  
Возвращает значение из словаря по ключу key.

>>> d = {"A1":"123", "A2":"456"}

>>> d.get("A1")

'123'

***items()***  
Возвращает элементы словаря (ключ, значение) в отформатированном виде.

>>> d = {"A1":"123", "A2":"456"}

>>> d.items()

dict\_items([('A2', '456'), ('A1', '123')])

***keys()***  
Возвращает ключи словаря.

>>> d = {"A1":"123", "A2":"456"}

>>> d.keys()

dict\_keys(['A2', 'A1'])

***pop(key[, default])***  
Если ключ key есть в словаре, то данный элемент удаляется из словаря и возвращается значение по этому ключу, иначе будет возвращено значение default. Если default не указан и запрашиваемый ключ отсутствует в словаре, то будет вызвано исключение KeyError.

>>> d = {"A1":"123", "A2":"456"}

>>> d.pop("A1")

'123'

>>> print(d)

{'A2': '456'}

***popitem()***  
Удаляет и возвращает пару (ключ, значение) из словаря. Если словарь пуст, то будет вызвано исключение KeyError.

>>> d = {"A1":"123", "A2":"456"}

>>> d.popitem()

('A2', '456')

>>> print(d)

{'A1': '123'}

***setdefault(key[, default])***  
Если ключ key есть в словаре, то возвращается значение по ключу. Если такого ключа нет, то в словарь вставляется элемент с ключом key и значением default, если default не определен, то по умолчанию присваивается None.

>>> d = {"A1":"123", "A2":"456"}

>>> d.setdefault("A3", "777")

'777'

>>> print(d)

{'A2': '456', 'A3': '777', 'A1': '123'}

>>> d.setdefault("A1")

'123'

>>> print(d)

{'A2': '456', 'A3': '777', 'A1': '123'}

***update([other])***  
Обновить словарь парами (key/value) из other, если ключи уже существуют, то обновить их значения.

>>> d = {"A1":"123", "A2":"456"}

>>> d.update({"A1":"333", "A3":"789"})

>>> print(d)

{'A2': '456', 'A3': '789', 'A1': '333'}

***values()***  
Возвращает значения элементов словаря.

>>> d = {"A1":"123", "A2":"456"}

>>> d.values()

dict\_values(['456', '123'])

**или:**

**Словари в Python** - неупорядоченные коллекции произвольных объектов с доступом по ключу. Их иногда ещё называют ассоциативными массивами или хеш-таблицами.Пара фигурных скобок {} создает пустой словарь. В отличие от последовательностей, доступ к элементам словаря производится по ключу, а не по индексу, ключ может быть любого типа, ключ не допускает изменений.

Основные операции над словарем — сохранение с заданным ключом и извлечение по нему значения. Также можно удалить пару key: value с помощью инструкции del.

Метод keys() для словаря возвращает список всех используемых ключей в произвольном порядке; для сортировки списка нужно применить метод sort(). Для определения наличия определенного ключа есть метод has\_key(), который в версии 3.0 успеет устареть — вместо него есть оператор in. Добавление нового объекта в словарь не требует предварительных проверок: если ранее ключу уже соответствовало некоторое значение, оно будет перезаписано.

Пример — словарь в качестве телефонного справочника:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | >>> dic = {'vanya' : 23323223, 'smith' : 32232332}  >>> dic['fedya'] = 33332222  >>> dic  {'vanya': 23323223, 'fedya': 33332222, 'smith': 32232332}  >>> dic['smith']  32232332  >>> del dic['vanya']  >>> dic  {'fedya': 33332222, 'smith': 32232332}  >>> dic.keys()  ['fedya', 'smith']  >>> dic.has\_key('fedya')  True |

Создать словарь можно несколькими способами:

1. Обычное выражение — оно удобно, если словарь статичен:

|  |  |
| --- | --- |
| 1 | D = {'name': 'mel', 'age': 45} |

1. Динамический вариант создания на лету:

|  |  |
| --- | --- |
| 1  2  3 | D = {}  D['name'] = 'mel'  D['age'] = 45 |

1. С помощью функции dict() — ключи при этом должны быть строками. С помощью этой функции можно избавить себя от обязательного условия заключать ключ в кавычки. В примере приведены четыре варианта создания одного и того же словаря:

|  |  |
| --- | --- |
| 1  2  3  4 | **d1 = dict(id=1948, name="Washer", size=3)**  **d2 = dict({"id": 1948, "name": "Washer", "size": 3})**  **d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])**  **d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))** |

1. С помощью fromkeys() — создает словарь по списку ключей с пустыми значениями:

|  |  |
| --- | --- |
| 1 | D = {}.fromkeys(['name', 'age'],123) |

1. С помощью конструктора:

|  |  |
| --- | --- |
| 1 | d = dict((x, x\*\*2) for x in xrange(5)) |

2. Функции/методы словаря

dict() — создание словаря;

len() — возвращает число пар;

clear() — удаляет все значения из словаря;

copy() — создает псевдокопию словаря;

deepcopy() — создает полную копию словаря;

fromkeys() — создание словаря;

get() — получить значение по ключу;

has\_ key() — проверка значения по ключу;

items() — возвращает список значений;

iteriyems() — возвращает итератор;

keys() — возвращает список ключей;

iterkeys() — возвращает итератор ключей;

pop() — извлекает значение по ключу;

popitem() — извлекает произвольное значение;

update() — изменяет словарь;

values() — возвращает список значений;

itervalues() — возвращает итератор на список значений.

in — оператор, проверяет наличие значения по ключу;

del — оператор, удаляет пару по ключу;

dict() — конструирует словарь с помощью последовательности.

Например, создать словарь с помощью списка кортежей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> items = [('name','sveta'),('age',20)]  >>> d = dict(items)  >>> d  {'age': 20, 'name': 'sveta'}    >>> len(d)  2 |

in() — оператор проверки вхождения.

Пример: база данных может быть заполнена в виде словаря.

Проверить наличие в базе данных телефона по имени:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | people = {'Alice': {'phone': '2341', 'addr': 'Foo drive 23' },            'Beth':  {'phone': '9102', 'addr': 'Bar street 42'}}  name = 'Alice'  key = 'phone'  if name in people:    print "%s phone is %s" % (name, people[name][key])  >>> Alice phone is 2341    copy() |

Пример создания копии словаря:

|  |  |
| --- | --- |
| 1  2  3  4 | >>> x = {"user":'admin','attr':[1,2,3]}  >>> y = x.copy()  >>> y  {'user': 'admin', 'attr': [1, 2, 3]} |

Метод copy() не делает полного копирования: если мы, например, сделаем операцию:

|  |  |
| --- | --- |
| 1 | >>> x['attr'].remove(1) |

то с удивлением обнаружим, что удаление атрибута произойдет также и в копии.

Чтобы этого не произошло, нужно использовать метод deepcopy().

|  |  |
| --- | --- |
| 1  2 | >>> from copy import deepcopy  >>> y = x.deepcopy() |

fromkeys() — создает словарь по заданным ключам с пустыми значениями:

|  |  |
| --- | --- |
| 1  2 | >>> {}.fromkeys(['name', 'age'])  {'age': None, 'name': None} |

Можно все значения заполнить по умолчанию:

|  |  |
| --- | --- |
| 1  2 | >>> {}.fromkeys(['name', 'age'],123)  {'age': 123, 'name': 123} |

get() — получает значение по ключу, в случае отсутствия дает None:

|  |  |
| --- | --- |
| 1  2  3 | >>> d = {}  >>> print d.get('name')  None |

has\_key() — проверяет, есть ли в словаре значение по данному ключу:

|  |  |
| --- | --- |
| 1  2  3 | >>> d = {}  >>> d.has\_key('name')  False |

items() — возвращает список значений:

|  |  |
| --- | --- |
| 1  2 | for key, value in d.items():          print(key, value) |

iteriyems() — возвращает итератор — выдает тот же результат:

|  |  |
| --- | --- |
| 1  2 | >>> for k, v in d.iteritems():  ...     print k, v |

keys() — возвращает список ключей;

iterkeys() — возвращает итератор ключей:

|  |  |
| --- | --- |
| 1  2  3  4 | >>> d.keys()  ['url', 'title']  >>> d.iterkeys()  <dictionary-keyiterator object at 0xb7c4dd00> |

pop() — извлекает значение по ключу с последующим удалением:

|  |  |
| --- | --- |
| 1  2  3 | >>> d.pop('title')  >>> d  {'url': 'http://www.python.org'} |

popitem() — извлекает произвольное значение с последующим удалением:

|  |  |
| --- | --- |
| 1  2  3  4 | >>> d = {'title': 'Python Web Site', 'url': 'http://www.python.org', 'www': 'python'}  >>> d.popitem()  >>> d  {'www': 'python', 'title': 'Python Web Site'} |

update() — изменяет значение по ключу:

|  |  |
| --- | --- |
| 1  2  3  4 | >>> d2 = {'www':'python.org'}  >>> d.update(d2)  >>> d  {'www': 'python.org', 'title': 'Python Web Site'} |

values() — возвращает список значений:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> d={}  >>> d[1]=1  >>> d[2]=2  >>> d[3]=3  >>> d  {1: 1, 2: 2, 3: 3}  >>> d.values()  [1, 2, 3] |

del — оператор удаляет пару ключ: значение по ключу:

|  |  |
| --- | --- |
| 1  2  3 | >>> del d[2]  >>> d  {1: 1, 3: 3} |

3. Операции

Поскольку словари представляют собой мапы (map), а не последовательности, к ним нельзя применить конкатенацию или срезы.

**К словарям можно применять стандартные операторы сравнения:**

|  |  |
| --- | --- |
| 1 | <, <=, ==, !=, >=, > |

Для того чтобы сделать проход по ключам словаря, используем **for:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | >>> table = {'Python': 'Guido van Rossum',  ...          'Perl':     'Larry Wall',  ...          'Tcl':      'John Ousterhout' }  >>> for lang in table:  ...     print(lang, table[lang])  ..  >>> Tcl     John Ousterhout  >>> Python  Guido van Rossum  >>> Perl    Larry Wall |

Словари хорошо подходят для хранения многомерных массивов или матриц:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | >>> Matrix = {}  >>> Matrix[(2, 3, 4)] = 88  >>> Matrix[(7, 8, 9)] = 99  >>>  >>> X = 2; Y = 3; Z = 4  >>> Matrix[(X, Y, Z)]  88  >>> Matrix  {(2, 3, 4): 88, (7, 8, 9): 99} |

С помощью словарей можно хранить структурированную информацию в виде записей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> man = {'name': 'Serg',  ...        'jobs': ['programmer', 'writer'],  ...        'web': 'www.iakovlev.org',  ...        'home': {'city': 'Moscow', 'zip':129000}}  >>> man['name']  Serg  >>> man['jobs'][1]  'writer' |

24 Матрицы. Создание матрицы. Ввод и вывод матрицы. Выполнение операций с элементами матрицы.

Для работы с **матрицами** также используются списки. Каждый элемент списка-матрицы содержит вложенный список.

Таким образом, получается структура из вложенных списков, количество которых определяет количество строк матрицы, а число элементов внутри каждого вложенного списка указывает на количество столбцов в исходной матрице.

# Создание матрицы

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

a = [[0] \* m for I in range(n)] #m- столбцы, n-строки

# Ввод двумерного массива

n = int(input())

a = []

for i in range(n):

a.append([int(j) for j in input().split()])

ИЛИ при помощи ГЕНЕРАТОРА:

n = int(input())

a = [[int(j) for j in input().split()] for i in range(n)]

# Вывод двумерного массива:

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

for i in range(len(a)):

for j in range(len(a[i])):

print(a[i][j], end=' ')

print()

ИЛИ:

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

for row in a:

for elem in row:

print(elem, end=' ')

print()

или:

**Для инициализации элементов матрицы** случайными числами используется алгоритм:

|  |  |
| --- | --- |
| 23456 | **import** random**for** i **in** range(N):**for** j **in** range(M):matrix[i][j] = random.randint ( 30, 60 )**print** ( "{:4d}".format(matrix[i][j]), end = "" )**print**() |

###### **Обработка элементов двумерного массива:**

Нумерация элементов двумерного массива, как и элементов одномерного массива, начинается с нуля.  
Т.е. matrix[2][3] — это элемент третьей строки четвертого столбца.

**Пример обработки элементов матрицы:**  
Найти произведение элементов двумерного массива.

|  |  |
| --- | --- |
| 1  2  3  4  5 | p = 1  **for** i **in** range(N):  **for** j **in** range(M):  p \*= matrix[i][j]  **print** (p) |

**Пример:**  
Найти сумму элементов двумерного массива.

|  |  |
| --- | --- |
| 1  2  3  4 | s = 0  **for** row **in** matrix:  s += sum(row)  **print** (s) |

Для поиска суммы существует стандартная функция **sum**.

### **Вложенные генераторы двумерных массивов**

Для создания двумерных массивов можно использовать вложенные генераторы, разместив генератор списка, являющегося строкой, внутри генератора всех строк. Напомним, что сделать список из n строк и m столбцов можно при помощи генератора, создающего список из n элементов, каждый элемент которого является списком из m нулей:

[[0] \* m for i in range(n)]

[[0 for j in range(m)] for i in range(n)]

Но если число 0 заменить на некоторое выражение, зависящее от i (номер строки) и j (номер столбца), то можно получить список, заполненный по некоторой формуле.

Например, пусть нужно задать следующий массив (для удобства добавлены дополнительные пробелы между элементами):



0  0  0  0  0  0

0  1  2  3  4  5

0  2  4  6  8 10

0  3  6  9 12 15

0  4  8 12 16 20

В этом массиве n = 5 строк, m = 6 столбцов, и элемент в строке i и столбце j вычисляется по формуле: a[i][j] = i \* j.

Для создания такого массива можно использовать генератор:

[[i \* j for j in range(m)] for i in range(n)]

**????25. Матрицы. Квадратные матрицы. Обработка верхне- и нижнетреугольных матриц. Работа с диагональными элементами матрицы**.

j<i – элементы расположены ниже главной диагонали;  
j>i – элементы расположены выше главной диагонали;  
i+j-1 < n– элементы расположены над побочной диагональю;(n- столбцы)  
i+j-1 > n– элементы расположены под побочной диагональю;

[i][i] - главная диагональ

[i][n-1-i] – побочная диагональ

#### Заполнение квадратной матрицы

В терминах языка программирования Python матрица - это список с вложенными списками одинаковой длины. Если говорить о квадратной матрице (а только в этом случае имеет смысл говорить о ее диагоналях), то количество вложенных списков и количество элементов в них должно совпадать. Например, список [[12,45,18][0,3,10][21,18,6]] можно назвать квадратной матрицей.

Пусть квадратная матрица заполняется случайными числами и ее размерность будет равна 10 (т. е. 10 строк и 10 столбцов). Перед заполнением матрицы, создается пустой список. Во внешнем цикле осуществляется проход по строкам, которые представляют собой вложенные списки. Сначала создается пустой вложенный список, потом во внутреннем цикле он заполняется случайными числами, после чего уже заполненный добавляется в основной список. При заполнении матрицы может также осуществляться вывод ее на экран.

#### Вызов функции

Поскольку функция должна уметь по выбору вычислять как сумму главной, так и побочной диагонали, то в нее необходимо передавать не только список, но и "сигнал" о том, по какому пути ей идти. Если функция будет возвращать полученную сумму в основную ветку программы, то вызов функции надо связать с переменной или поместить как аргумент в функцию print().

Перед вызовом функции надо получить значение переменной-"сигнала". Сумму какой диагонали вычислять, может выбрать пользователь. Например, если он вводит символ '1', то вычислять главную диагональ, если '2', то побочную. Так как пользователь может ввести не только эти символы, а какие угодно, то разумно заключить вызов функции и вывод результата на экран в конструкцию if, условием выполнения которой будет правильно введенный символ.

#### Функция

При написании функции надо сделать выбор, будет ли она выводить результат на экран в своем теле или возвращать в основную ветку программы. Пусть функция возвращает результат, в таком случае она должна содержать оператор return (обычно, но не обязательно, в конце тела функции), после которого указывается возвращаемое значение.

Главная диагональ квадратной матрицы идет от левого верхнего угла до нижнего правого, и, следовательно, каждый элемент такой диагонали имеет совпадающие первый и второй индексы. При этом индексы следующего элемента диагонали отличаются от предыдущего на 1. Т.е. первый элемент главной диагонали имеет индексы [0][0], второй - [1][1] и т.д. Поэтому для нахождения суммы элементов главной диагонали, надо построчно перебрать матрицу и в каждой строке взять элемент, номер которого совпадает с номером строки, и добавить его значение в общую сумму.

Побочная диагональ идет из верхнего правого угла в нижний левый. Индексы ее элементов обратны. В первой строке это будет последний элемент, во второй - предпоследний и т.д. Если i - это номер строки, равный в начале 0, то второй индекс будет вычисляться как (N-1)-i. Выражение (N-1) - это индекс последнего элемента при индексации с нуля.

**print**('Элементы ниже главной диагонали:')

*# ниже главной*

**for** i **in** range(rows):

**for** j **in** range(cols):

**if** j<i:

**print**(matrix[i][j], end=' ')

**print**()

**print**('Элементы выше главной диагонали:')

*# выше главной*

**for** i **in** range(rows):

**for** j **in** range(cols):

**if** j>i:

**print**(matrix[i][j], end=' ')

**print**()

**print**('Элементы выше побочной диагонали:')

*# выше побочной*

**for** i **in** range(rows):

**for** j **in** range(cols):

**if** i < cols - j - 1:

**print**(matrix[i][j], end=' ')

**print**()

**print**('Элементы ниже побочной диагонали:')

*# ниже побочной*

**for** i **in** range(rows):

**for** j **in** range(cols):

**if** i > cols - j - 1:

**print**(matrix[i][j], end=' ')

**print**()

a=[[0]\*3 for i in range(3)]

n=3

for i in range (3):

a[i][n-i-1]=1 #побочная диагональ = 1

for i in range (3):

a[i][i]=2 #главная диагональ = 2

for q in a:

print(q)

**26. Функции. Создание функции. Аргументы функции. Возвращаемое значение.**

При помощи def():

def add(x, y):

return x + y

**Функция в python** - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции **def**.

**Если в функции нет return, то она называется** **процедурой.**

**Может иметь несколько return, но после выполнения любого из них, функция заканчивает работу.**

Определим простейшую функцию:

**def** add(x, y):

**return** x + y

Инструкция **return** говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y.

Теперь мы ее можем вызвать:

**>>>** add(1, 10)

11

**>>>** adod('abc', 'def')

'abcdef'

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

**>>> def** newfunc(n):

**...**  **def** myfunc(x):

**...**  **return** x + n

**...**  **return** myfunc

**>>>** new = newfunc(100) *# new - это функция*

**>>>** new(200)

300

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение [None](https://pythonworld.ru/tipy-dannyx-v-python/none.html):

**>>> def** func():

**...**  **pass**

**...**

**>>> print**(func())

None

## Аргументы функции

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с **произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.**

**>>> def** func(a, b, c=2): *#* ***c - необязательный аргумент***

**...**  **return** a + b + c

**>>>** func(1, 2) *# a = 1, b = 2, c = 2 (по умолчанию)*

5

**>>>** func(1, 2, 3) *# a = 1, b = 2, c = 3*

6

**>>>** func(a=1, b=3) *# a = 1, b = 3, c = 2*

6

**>>>** func(a=3, c=6) *# a = 3, c = 6, b не определен*

Traceback (most recent call last):

File "", line 1, in

func(a=3, c=6)

TypeError: func() takes at least 2 arguments (2 given)

Функция также может принимать **переменное количество позиционных аргументов**, тогда перед именем ставится \*:

**>>> def** func(\*args):

**...**  **return** args

**...**

**>>>** func(1, 2, 3, 'abc')

(1, 2, 3, 'abc')

**>>>** func()

()

**>>>** func(1)

(1,)

Как видно из примера**, args - это**[**кортеж**](https://pythonworld.ru/tipy-dannyx-v-python/kortezhi-tuple.html) из всех переданных аргументов функции, и с переменной можно работать так же, как и с кортежем.

Функция может принимать и произвольное число **именованных** аргументов, тогда перед именем ставится \*\*:

**>>> def** func(\*\*kwargs):

**...**  **return** kwargs

**...**

**>>>** func(a=1, b=2, c=3)

{'a': 1, 'c': 3, 'b': 2}

**>>>** func()

{}

**>>>** func(a='python')

{'a': 'python'}

В переменной **kwargs у нас хранится**[**словарь**](https://pythonworld.ru/tipy-dannyx-v-python/slovari-dict-funkcii-i-metody-slovarej.html).

В **Python** две базовых области видимости переменных:

* **Глобальные переменные**
* **Локальные переменные**

Переменные объявленные внутри тела функции имеют **локальную** область видимости, те что объявлены вне какой-либо функции имеют **глобальную** область видимости.

Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

Например:

[?](http://pythonicway.com/python-functions)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | # глобальная переменная age  age = 44    def info():      print age # Печатаем глобальную переменную age    def local\_info():      age = 22 # создаем локальную переменную age      print age    info() # напечатает 44  local\_info() # напечатает 22 |

Важно помнить, что для того чтобы получить доступ к глобальной переменной, достаточно лишь указать ее имя. Однако, если перед нами стоит задач а **изменить глобальную переменную** внутри функции - необходимо использовать ключевое слово **global**.

Например:

[?](http://pythonicway.com/python-functions)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # глобальная переменная age  age = 13    # функция изменяющая глобальную переменную  def get\_older():      global age      age += 1    print age # напечатает 13  get\_older() # увеличиваем age на 1  print age # напечатает 14 |

**27 Функции. Lambda-функции. Рекурсивные функции.**

Func = lambda x, y: x + y

Func (1,2)

(lambda x, y: x + y)(1,2)

## Анонимные функции, инструкция lambda

**Анонимные функции** могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные фун кции создаются с помощью инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией def func()

**>>>** func = **lambda** x, y: x + y

**>>>** func(1, 2)

3

**>>>** func('a', 'b')

'ab'

**>>>** (**lambda** x, y: x + y)(1, 2)

3

**>>>** (**lambda** x, y: x + y)('a', 'b')

'ab'

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

**>>>** func = **lambda** \*args: args

**>>>** func(1, 2, 3, 4)

(1, 2, 3, 4)

## Рекурсия

**Рекурсией** в программировании называется ситуация, в которой функция вызывает саму себя. Классическим примером рекурсии может послужить функция вычисления факториала числа.

Напомним, что факториалом числа, например, 5 является произведение всех натуральных (целых) чисел от 1 до 5. То есть, 1 \* 2 \* 3 \* 4 \* 5

**Рекурсивная функция вычисления факториала на языке Python будет выглядеть так**:

[?](http://pythonicway.com/python-functions)

|  |  |
| --- | --- |
| 1  2  3  4  5 | def fact(num):      if num == 0:          return 1 # По договоренности факториал нуля равен единице      else:          return num \* fact(num - 1) # возвращаем результат произведения num и результата  возвращенного функцией fact(num - 1) |

Однако следует помнить, что использование рекурсии часто может быть неоправданным. Дело в том, что в момент вызова функции в оперативной памяти компьютера резервируется определенное количество памяти, соответственно чем больше функций одновременно мы запускаем - тем больше памяти потребуется, что может привести к переполнению стека (stack overflow) и программа завершится аварийно, не так как предполагалось. Учитывая это, там где это возможно, вместо рекурсии лучше применять [циклы](http://pythonicway.com/python-loops).

28. Файлы. Текстовые файлы. Открытие файла. Режимы доступа к файлу.

F = open(‘text.txt’, ‘r’)

Режимы:

'r’ – открытие на чтение

‘w’ – открытие на запись

‘x’ – открытие на запись, если файла не существует

‘a’ – открытие на дозапись

‘b’ – открытие в двоичном режиме

**Те́кстовый файл** — компьютерный [файл](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), содержащий [текстовые данные](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5). Текстовым файлам противопоставляются [двоичные (бинарные) файлы](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB), в которых содержатся данные, не рассчитанные на интерпретацию в качестве текстовых (например, файлы, хранящие текст в закодированном или сжатом виде, или хранящие не текст, а звук, изображение или иные данные).

В отличие от термина «[текстовые данные](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5)» («текстовый формат данных»), характеризующего содержимое данных, термин «текстовый файл» относится к файлу и характеризует его как контейнер, хранящий такие данные.

## Создание и открытие

Чтобы получить возможность работать с файлом в Python 3, для начала его необходимо создать. Сделать это можно стандартными средствами операционной системы, перейдя в нужный каталог и создав новый документ с форматом txt. Однако аналогичное действие выполняется и с помощью метода open в языке программирования Python, которому надо передать в качестве параметров название файла и режим его обработки.

Следующий код демонстрирует получение переменной file ссылки на новый документ. Если запустить эту программу, она создаст текстовый файл test.txt в папке, где хранится исходный код.

file = open("test.txt", "w")

file.close()

Если же файл с указанным именем test.txt уже существует в каталоге с кодом, программа просто продолжит работу с ним, не создавая новый документ. Как можно заметить, имя файла является первым параметром метода open. Сразу за ним следует специальная буква, которая обозначает метод обработки данных. В данном случае “w” означает write, то есть запись. Подробнее обо всех доступных режимах работы будет немного дальше, а сейчас важно усвоить, что **после выполнения любых манипуляций над файлом, его обязательно следует закрыть с помощью функции close, чтобы гарантированно избежать потери информации**.

В предыдущем примере для доступа к файлу был использован относительный путь, который не содержит в себе исчерпывающих сведений о местоположении объекта на жестком диске. Для того, чтобы задать их, необходимо в качестве первого аргумента функции open прописать абсолютный путь. В данном случае документ test.txt будет находиться в корневом каталоге на диске D, а не в папке программы.

file = open(r"D:\test.txt", "w")

file.close()

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

Режимы могут быть объединены, то есть, к примеру, 'rb' - чтение в двоичном режиме. По умолчанию режим равен 'rt'.

И последний аргумент, encoding, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку.

29. Файлы. Текстовые файлы. Чтение файла. Запись в файл. Поиск в файле.

# Чтение файла

F = open(‘text.txt’)

f=F.read()

print(f)

# Запись в файл

L = [str(i) + str(i-1) for I in range(20)]

F = open(‘text.txt’, ‘w’)

For index in L:

F.write(index + ‘\n’)

F.close()

### Чтение файла

Для чтения файла он открывается с режимом r (Read), и затем мы можем считать его содержимое различными методами:

* **readline()**: считывает одну строку из файла
* **read()**: считывает все содержимое файла в одну строку
* **readlines()**: считывает все строки файла в список

Например, считаем выше записанный файл построчно:

|  |  |
| --- | --- |
| 1  2  3 | with open("hello.txt", "r") as file:      for line in file:          print(line, end="") |

Несмотря на то, что мы явно не применяем метод readline() для чтения каждой строки, но в при переборе файла этот метод автоматически вызывается для получения каждой новой строки. Поэтому в цикле вручную нет смысла вызывать метод readline. И поскольку строки разделяются символом перевода строки "\n", то чтобы исключить излишнего переноса на другую строку в функцию print передается значение end="".

Теперь явным образом вызовем метод readline() для чтения отдельных строк:

|  |  |
| --- | --- |
| 1  2  3  4  5 | with open("hello.txt", "r") as file:      str1 = file.readline()      print(str1, end="")      str2 = file.readline()      print(str2) |

Консольный вывод:

##### hello world

##### good bye, world

Метод readline можно использовать для построчного считывания файла в цикле while:

|  |  |
| --- | --- |
| 1  2  3  4  5 | with open("hello.txt", "r") as file:      line = file.readline()      while line:          print(line, end="")          line = file.readline() |

Если файл небольшой, то его можно разом считать с помощью метода **read()**:

|  |  |
| --- | --- |
| 1  2  3 | with open("hello.txt", "r") as file:      content = file.read()      print(content) |

И также применим метод **readlines()** для считывания всего файла в список строк:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | with open("hello.txt", "r") as file:      contents = file.readlines()      str1 = contents[0]      str2 = contents[1]      print(str1, end="")      print(str2) |

При чтении файла мы можем столкнуться с тем, что его кодировка не совпадает с ASCII. В этом случае мы явным образом можем указать кодировку с помощью параметра **encoding**:

|  |  |
| --- | --- |
| 1  2  3 | filename = "hello.txt"  with open(filename, encoding="utf8") as file:      text = file.read() |

**Если вы хотите включить \n :**

**with open(fname) as f: content = f.readlines()**

**Если вы не хотите использовать \n :**

**with open(fname) as f: content = f.read().splitlines()**

## Запись

В Python 3 запись в файл осуществляется с помощью метода **write.** Метод вызываем у объекта, который ссылается на существующий файл. Важно помнить, что для этого следует предварительно открыть документ с помощью функции open и указать режим записи символом “w”. Метод write принимает в качестве аргумента данные, которые нужно поместить в текстовый файл. Следующий пример кода показывает запись строки “hello”.

file = open("test.txt", "w")

file.write("hello")

file.close()

Если необходимо добавить новую информацию к записанным ранее данным, следует заново вызвать функцию open, указав ей в качестве режима работы символ “a”. В противном случае все сведения из файла test.txt будут полностью удалены. В приведенном ниже примере кода текстовый документ открывается для дополнительной записи, после чего в него помещается строковый литерал “ world” с пробелом вначале. Таким образом в test.txt будет располагаться “hello world”. После всего этого не нужно забывать об обязательном закрытии файла.

file = open("test.txt", "a")

file.write(" world")

file.close()

**Поиск:**

Чтобы напечатать все строки, которые содержат заданное слово в данном текстовом файле, закодированном в utf-8 кодировке:

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

from \_\_future\_\_ import print\_function

import io

word = u'привет'

with io.open('/path/to/file.txt', encoding='utf-8') as file:

for line in file:

if word in line:

print(line, end='')

* данный метод закрывает входной файл (with-инструкция), не рассчитывая на особенности уборки мусора в реализации интерпретатора или возникновения исключений (ошибок)
* **считывание файла идёт построчно без загрузки всего файла в память**

30. Файлы. Текстовые файлы. Итерационное чтение содержимого файла.

F = open(‘text.txt’,’r’)

For I in F:

Print(i)

1. file = open("file.txt")

for line in file:

#do something

В первом вы выполняете итерацию по файлу, строка за строкой. В этом случае все данные файла не считываются в память, а только текущая строка считывается в память, это полезно для обработки очень больших файлов.

1. file = open("file.txt")

contents = file.read()

for line in contents:

# do something

Во втором, file.read() возвращает полные данные файла в виде строки, когда вы выполняете итерацию по ней, вы фактически выполняете итерацию по символу данных файла по символу. Это считывает полные данные файла в память.

Второй случай читает содержимое файла в одну большую строку. Если вы перебираете строку, вы получаете каждый символ по очереди. Если вы хотите получить каждую строку по очереди, вы можете сделать это:

for line in contents.split('\n'):

# do something

Или вы можете читать содержимое в виде списка строк, используя readlines() вместо read().

with open('file.txt','r') as fin:

lines = fin.readlines()

for line in lines:

# do something

Итерация

Итерация по файлу является базовой операцией и имеет множество вариантов. Использование функции read() для байтового чтения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | f = open(filename)  while True:    char = f.read(1)    if not char: break    process(char)  f.close() |

Построчное чтение текстовых файлов и функция readline():

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | f = open(filename)  while True:    line = f.readline()    if not line: break    process(line)  f.close() |

Файл сам может выступать в роли итератора:

|  |  |
| --- | --- |
| 1  2 | for line in open(filename):    process(line) |

**??31. Файлы. Бинарные файлы. Сериализация данных. Модуль pickle.**

File = open(‘text.txt’,’wb’)

Dict1 = {‘0’: kek, ‘1’:mem}

dump( Dict1,file) – записывает сериализованный объект в файл

File = open(‘text.txt’,’rb’)

load(file) – загружает объект из файла

Бинарные файлы в отличие от текстовых хранят информацию в виде набора байт. Для работы с ними в Python необходим встроенный модуль **pickle**. Этот модуль предоставляет два метода:

* **dump(obj, file)**: записывает объект obj в бинарный файл file
* **load(file)**: считывает данные из бинарного файла в объект

При открытии бинарного файла на чтение или запись также надо учитывать, что нам нужно применять режим "b" в дополнение к режиму записи ("w") или чтения ("r"). Допустим, надо надо сохранить два объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import pickle    FILENAME = "user.dat"    name = "Tom"  age = 19    with open(FILENAME, "wb") as file:      pickle.dump(name, file)      pickle.dump(age, file)    with open(FILENAME, "rb") as file:      name = pickle.load(file)      age = pickle.load(file)      print("Имя:", name, "\tВозраст:", age) |

С помощью функции dump последовательно записываются два объекта. Поэтому при чтении файла также последовательно посредством функции load мы можем считать эти объекты. Консольный вывод программы:

# Имя: Tom Возраст: 28

Подобным образом мы можем сохранять и извлекать из файла наборы объектов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import pickle    FILENAME = "users.dat"    users = [      ["Tom", 28, True],      ["Alice", 23, False],      ["Bob", 34, False]  ]    with open(FILENAME, "wb") as file:      pickle.dump(users, file)      with open(FILENAME, "rb") as file:      users\_from\_file = pickle.load(file)      for user in users\_from\_file:          print("Имя:", user[0], "\tВозраст:", user[1], "\tЖенат(замужем):", user[2]) |

В зависимости от того, какой объект мы записывали функцией dump, тот же объект будет возвращен функцией load при считывании файла.

Консольный вывод:

Имя: Tom Возраст: 28 Женат(замужем): True

Имя: Alice Возраст: 23 Женат(замужем): False

Имя: Bob Возраст: 34 Женат(замужем): False

**32.** NumPy — это библиотека языка Python, добавляющая поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.

## Установка NumPy

На linux - пакет python3-numpy (или аналогичный для вашей системы), или через [pip](https://pythonworld.ru/osnovy/pip.html). Ну или же собирать из исходников <https://sourceforge.net/projects/numpy/files/NumPy/>.

На Windows на том же сайте есть exe установщики. Или, если возникают проблемы, рекомендую ещё хороший сборник библиотек <http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy>.

## Начинаем работу

Основным объектом NumPy является однородный многомерный массив (в numpy называется numpy.ndarray). Это многомерный массив элементов (обычно чисел), одного типа.

Наиболее важные атрибуты объектов ndarray:

**ndarray.ndim** - число измерений (чаще их называют "оси") массива.

**ndarray.shape** - размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси. Для матрицы из n строк и m столбов, shape будет (n,m). Число элементов кортежа shape равно ndim.

**ndarray.size** - количество элементов массива. Очевидно, равно произведению всех элементов атрибута shape.

**ndarray.dtype** - объект, описывающий тип элементов массива. Можно определить dtype, используя стандартные типы данных Python. NumPy здесь предоставляет целый букет возможностей, как встроенных, например: bool\_, character, int8, int16, int32, int64, float8, float16, float32, float64, complex64, object\_, так и возможность определить собственные типы данных, в том числе и составные.

**ndarray.itemsize** - размер каждого элемента массива в байтах.

**ndarray.data** - буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

## Создание массивов

В NumPy существует много способов создать массив. Один из наиболее простых - создать массив из обычных списков или кортежей Python, используя функцию numpy.array() (запомните: array - функция, создающая объект типа ndarray):

>>>

**>>> import** **numpy** **as** **np**

**>>>** a = np.array([1, 2, 3])

**>>>** a

array([1, 2, 3])

**>>>** type(a)

<class 'numpy.ndarray'>

Функция array() трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности (но можно и переопределить его в момент создания).

>>>

**>>>** b = np.array([[1.5, 2, 3], [4, 5, 6]])

**>>>** b

array([[ 1.5, 2. , 3. ],

[ 4. , 5. , 6. ]])

Можно также переопределить тип в момент создания:

>>>

**>>>** b = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=np.complex)

**>>>** b

array([[ 1.5+0.j, 2.0+0.j, 3.0+0.j],

[ 4.0+0.j, 5.0+0.j, 6.0+0.j]])

Функция array() не единственная функция для создания массивов. Обычно элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип создаваемого массива — float64).

Функция zeros() создает массив из нулей, а функция ones() — массив из единиц. Обе функции принимают кортеж с размерами, и аргумент dtype:

>>>

**>>>** np.zeros((3, 5))

array([[ 0., 0., 0., 0., 0.],

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

[ 0., 0., 0., 0., 0.]])

**>>>** np.ones((2, 2, 2))

array([[[ 1., 1.],

[ 1., 1.]],

[[ 1., 1.],

[ 1., 1.]]])

Функция eye() создаёт единичную матрицу (двумерный массив)

>>>

**>>>** np.eye(5)

array([[ 1., 0., 0., 0., 0.],

[ 0., 1., 0., 0., 0.],

[ 0., 0., 1., 0., 0.],

[ 0., 0., 0., 1., 0.],

[ 0., 0., 0., 0., 1.]])

Функция empty() создает массив без его заполнения. Исходное содержимое случайно и зависит от состояния памяти на момент создания массива (то есть от того мусора, что в ней хранится):

>>>

**>>>** np.empty((3, 3))

array([[ 6.93920488e-310, 6.93920488e-310, 6.93920149e-310],

[ 6.93920058e-310, 6.93920058e-310, 6.93920058e-310],

[ 6.93920359e-310, 0.00000000e+000, 6.93920501e-310]])

**>>>** np.empty((3, 3))

array([[ 6.93920488e-310, 6.93920488e-310, 6.93920147e-310],

[ 6.93920149e-310, 6.93920146e-310, 6.93920359e-310],

[ 6.93920359e-310, 0.00000000e+000, 3.95252517e-322]])

Для создания последовательностей чисел, в NumPy имеется функция arange(), аналогичная встроенной в Python range(), только вместо списков она возвращает массивы, и принимает не только целые значения:

>>>

**>>>** np.arange(10, 30, 5)

array([10, 15, 20, 25])

**>>>** np.arange(0, 1, 0.1)

array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])

Вообще, при использовании arange() с аргументами типа float, сложно быть уверенным в том, сколько элементов будет получено (из-за ограничения точности чисел с плавающей запятой). Поэтому, в таких случаях обычно лучше использовать функцию linspace(), которая вместо шага в качестве одного из аргументов принимает число, равное количеству нужных элементов:

>>>

**>>>** np.linspace(0, 2, 9) *# 9 чисел от 0 до 2 включительно*

array([ 0. , 0.25, 0.5 , 0.75, 1. , 1.25, 1.5 , 1.75, 2. ])

fromfunction(): применяет функцию ко всем комбинациям индексов

>>>

**>>> def** f1(i, j):

**...**  **return** 3 \* i + j

**...**

**>>>** np.fromfunction(f1, (3, 4))

array([[ 0., 1., 2., 3.],

[ 3., 4., 5., 6.],

[ 6., 7., 8., 9.]])

**>>>** np.fromfunction(f1, (3, 3))

array([[ 0., 1., 2.],

[ 3., 4., 5.],

[ 6., 7., 8.]])

## Печать массивов

Если массив слишком большой, чтобы его печатать, NumPy автоматически скрывает центральную часть массива и выводит только его уголки.

>>>

**>>> print**(np.arange(0, 3000, 1))

[ 0 1 2 ..., 2997 2998 2999]

Если вам действительно нужно увидеть весь массив, используйте функцию numpy.set\_printoptions:

np.set\_printoptions(threshold=np.nan)

И вообще, с помощью этой функции можно настроить печать массивов "под себя". Функция numpy.set\_printoptions принимает несколько аргументов:

*precision* : количество отображаемых цифр после запятой (по умолчанию 8).

*threshold* : количество элементов в массиве, вызывающее обрезание элементов (по умолчанию 1000).

*edgeitems* : количество элементов в начале и в конце каждой размерности массива (по умолчанию 3).

*linewidth* : количество символов в строке, после которых осуществляется перенос (по умолчанию 75).

*suppress* : если True, не печатает маленькие значения в scientific notation (по умолчанию False).

*nanstr* : строковое представление NaN (по умолчанию 'nan').

*infstr* : строковое представление inf (по умолчанию 'inf').

**34. Модуль matplotlib. Построение графиков в декартовой системе координат. Управление областью рисования.**

##### Python. Графики

##### В качестве инструмента построения графиков рассмотрим библиотеку **matplotlib** и её модуль [pyplot](https://matplotlib.org/api/pyplot_api.html). Он предназначен для работы с двумерными фигурами и умеет строить большое число разнообразных типов графиков, обеспечивая при этом **Matlab**-подобный интерфейс. Простейший график можно создать с помощью следующих команд.

##### **import** numpy **as** np   **import** matplotlib.pyplot **as** plt    # векторы абсцисс и ординат    X = np.linspace(0, 2\*np.pi, 50)   Y = np.sin(X)   # создание графика и его отображение   plt.plot(X,Y)     plt.show()

##### Как видно, отрисовка происходит через обращение к модулю **pyplot** (plt). График строит функция plot(), а отображает - show().

##### [https://4.bp.blogspot.com/-DU6SHj1gw9Y/WwT8c3VeOrI/AAAAAAAABIk/BblfeVVS-wgZut3xTG_VYy4Ou10V22wqwCLcBGAs/s320/Screenshot_2018-05-23_08-29-52.png](https://4.bp.blogspot.com/-DU6SHj1gw9Y/WwT8c3VeOrI/AAAAAAAABIk/BblfeVVS-wgZut3xTG_VYy4Ou10V22wqwCLcBGAs/s1600/Screenshot_2018-05-23_08-29-52.png)

##### 

##### Для спецификации параметров линии можно использовать как "стандартные" коды, так и именованые аргументы функции plot(). Например, следующий фрагмент кода изменяет цвет и структуру линии, а также ограничивает диапазон отображения.

##### plt.plot(X,Y,'r--')

##### plt.xlim((1, 3))       # предел по X - от 1 до 3

##### plt.ylim((0, 1.5))    # предел по Y - от 0 до 1.5

##### plt.show()

##### [https://1.bp.blogspot.com/-K0YfyiykdFc/WwT_RRzgErI/AAAAAAAABIw/UowrIN3ZXT4YPCZyF6EyphlYsmudmcyngCLcBGAs/s320/Screenshot_2018-05-23_08-41-13.png](https://1.bp.blogspot.com/-K0YfyiykdFc/WwT_RRzgErI/AAAAAAAABIw/UowrIN3ZXT4YPCZyF6EyphlYsmudmcyngCLcBGAs/s1600/Screenshot_2018-05-23_08-41-13.png)

##### Чтобы объединить несколько кривых на одном графике, просто перечислите их в функции plot(), при этом для каждой из них можно задать собственные парамеры кривых с помощью соответствующих "кодов". Следующий пример не только отрисовывает два графика в одном окне, но также делает подписи осей и отображает заголовок.

##### plt.plot(X,Y, X,np.cos(X))       # объединение 2-х графиков

##### plt.legend(('sin','cos'))           # подписи

##### plt.title('Trigonometry')        # заголовок

##### plt.xlabel('Time, s')               # наименование оси абсцисс

##### plt.ylabel('Amplitude, c.u.')  # наименование оси ординат

##### plt.show()

##### [https://2.bp.blogspot.com/-jTOUdx7AM0g/WwUBFv2YDTI/AAAAAAAABI8/HWiUe1eCo8870DCYk0U9ZFJnFe75QmJEACLcBGAs/s320/Screenshot_2018-05-23_08-47-47.png](https://2.bp.blogspot.com/-jTOUdx7AM0g/WwUBFv2YDTI/AAAAAAAABI8/HWiUe1eCo8870DCYk0U9ZFJnFe75QmJEACLcBGAs/s1600/Screenshot_2018-05-23_08-47-47.png)

##### Если необходимо отобразить графики по-отдельности, нужно для каждого изображения создавать отдельное окно с помощью функции figure().

##### plt.plot(X,Y)                     # первый график

##### plt.figure()                       # новое окно

##### plt.plot(X,np.cos(X))        # второй график

##### plt.show()

##### Другой вариант - отображение графиков в одном окне, но разных областях. Для этого служит функция subplot(), которая делит окно на заданное число частей и управляет доступом к ним. Если число частей меньше десяти, то можно использовать нотацию в виде трёхзначного числа:

##### строки-столбцы-номер\_графика

##### Следующий пример строит два графика, расположенных горизонтально (цифра 1 соответствует одной строке, цифра 2 - двум столбцам, последняя цифра определяет область для отображения).

##### plt.subplot(121)

##### plt.plot(X,Y)

##### plt.subplot(122)

##### plt.plot(X,np.cos(X))

##### plt.show()

##### [https://2.bp.blogspot.com/-LrqQmYJvHQY/WwUE9YegnXI/AAAAAAAABJI/hwxwPb7STUk1pnlNPGs8qoYPvzHJIL2oACLcBGAs/s320/Screenshot_2018-05-23_09-06-29.png](https://2.bp.blogspot.com/-LrqQmYJvHQY/WwUE9YegnXI/AAAAAAAABJI/hwxwPb7STUk1pnlNPGs8qoYPvzHJIL2oACLcBGAs/s1600/Screenshot_2018-05-23_09-06-29.png)

# -\*- coding: UTF-8 -\*-

import matplotlib.pyplot as plt

# Значения по оси X

X = [20.0, 40.0, 60.0, 80.0, 100.0]

# Набор значений по оси Y

Y\_10 = [0.97252, 0.94238, 0.89927, 0.85197, 0.79784]

Y\_20 = [0.96864, 0.93518, 0.89113, 0.84344, 0.78934]

Y\_30 = [0.96395, 0.92770, 0.88278, 0.83473, 0.78075]

# Строим диаграмму

# Задаем исходные данные для каждой линии диаграммы, внешний вид линий и маркеров.

# Функция plot() возвращает кортеж ссылок на объекты класса matplotlib.lines.Line2D

line\_10, line\_20, line\_30 = plt.plot(X, Y\_10, 'bD:', X, Y\_20, 'r^:', X, Y\_30, 'go:')

# Задаем интервалы значений по осям X и Y

plt.axis([15.0, 105.0, 0.75, 1.0])

# Задаем заголовок диаграммы

plt.title(u'Зависимость плотности водных растворов этилового спирта от температуры')

# Задаем подписи к осям X и Y

plt.xlabel(u'Массовая доля этилового спирта, %')

plt.ylabel(u'Плотность, г/мл')

# Задаем исходные данные для легенды и ее размещение

plt.legend( (line\_10, line\_20, line\_30), (u'Температура 10 \u00b0C', u'Температура 20 \u00b0C', u'Температура 30 \u00b0C'), loc = 'best')

# Включаем сетку

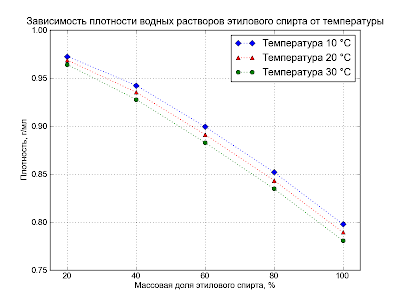
plt.grid()

# Сохраняем построенную диаграмму в файл

# Задаем имя файла и его тип

plt.savefig('spirit.png', format = 'png')

В результате выполнения скрипта файл spirit.png должен содержать такое изображение.

**[](http://lh4.ggpht.com/_oMYflydwAbM/SoKU_3j2KlI/AAAAAAAAAFE/OdGveMub4vs/spirit.png)**

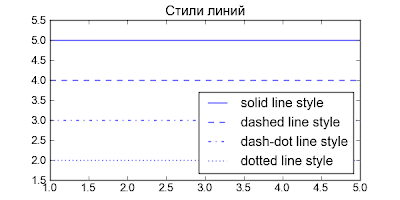
Комментарии к коду.

Функция plot() строит 2D диаграмму - задает исходные данные и внешний вид линии (линий) диаграммы. Функция принимает произвольное число наборов аргументов вида X\_set, Y\_set, format\_string. Где X\_set и Y\_set типа list или numpy.array - значения откладываемые по осям X и Y соответственно, format\_string - строка форматирования (не обязательный аргумент набора), определяет стиль и цвет линии. Строка форматирования (позаимствована из системы Matlab) включает (в произвольном порядке) три подстроки определяющих стиль линии, стиль маркера, цвет. Например 'g^:' = 'g' + '^' + ':' - цвет зеленый, маркер треугольник вершиной вверх, линия состоит из точек.

Ниже в таблицах приведены возможные значения для каждой из подстрок строки форматирования, а на рисунках представлены результаты их применения.

### Стили линий

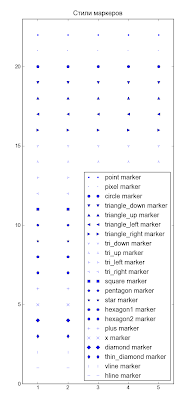
|  |  |  |
| --- | --- | --- |
| **Символ** | **Описание** | |
| '-' | solid line style | непрерывная линия |
| '--' | dashed line style | линия из штрихов |
| '-.' | dash-dot line style | чередование штрихов и точек |
| ':' | dotted line style | линия из точек |

**[](http://lh6.ggpht.com/_oMYflydwAbM/SoLHt41Mc8I/AAAAAAAAAGE/IL9VYKVldfw/line_style.png)**

### Стили маркеров

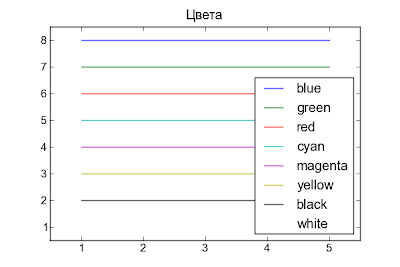
Я умышленно не стал переводить названия маркеров из документации, дабы не придумывать как по русски выразить различия между point markerи pixel marker или как обозвать tri\_right marker с thin\_diamond marker.

|  |  |
| --- | --- |
| **Символ** | **Описание** |
| '.' | point marker |
| ',' | pixel marker |
| 'o' | circle marker |
| 'v' | triangle\_down marker |
| '^' | triangle\_up marker |
| '<' | triangle\_left marker |
| '>' | triangle\_right marker |
| '1' | tri\_down marker |
| '2' | tri\_up marker |
| '3' | tri\_left marker |
| '4' | tri\_right marker |
| 's' | square marker |
| 'p' | pentagon marker |
| '\*' | star marker |
| 'h' | hexagon1 marker |
| 'H' | hexagon2 marker |
| '+' | plus marker |
| 'x' | x marker |
| 'D' | diamond marker |
| 'd' | thin\_diamond marker |
| '|' | vline marker |
| '\_' | hline marker |

**[](http://lh6.ggpht.com/_oMYflydwAbM/SoLTCaNmfwI/AAAAAAAAAGI/zKRlRofVktI/marker_style.png)**

### Цвета

|  |  |  |
| --- | --- | --- |
| **Символ** | **Цвет** | |
| 'b' | blue | синий |
| 'g' | green | зеленый |
| 'r' | red | красный |
| 'c' | cyan | бирюзовый |
| 'm' | magenta | пурпурный |
| 'y' | yellow | желтый |
| 'k' | black | черный |
| 'w' | white | белый |

**[](http://lh3.ggpht.com/_oMYflydwAbM/SoPtOZ6C0oI/AAAAAAAAAGM/e9IV3edeyRc/line_color.png)**

Функция plot() возвращает кортеж ссылок на объекты класса matplotlib.lines.Line2D. Каждый из объектов кортежа представляет линию на диаграмме. Посредством методов объектов можно менять свойства (внешний вид) любой из линий. В нашем случае кортеж ссылок в дальнейшем передается функции legend().

Функция axis() задает свойства осей диаграммы. В частности с её помощью можно установит масштаб осей. Для этого функции необходимо передать одни аргумент - список из четырех значений. Функция рассматривает такой список как [x\_min, x\_max, y\_min, y\_max], где x\_min, y\_minминимальные, x\_max, y\_max максимальные значения откладываемые по осям X и Y соответственно.

Функция может принимать в качестве одного аргумента и строку. Возможные значения такого аргумента и оказываемые ими действия представлены в таблице.

|  |  |
| --- | --- |
| **Аргумент** | **Действие** |
| 'off' | Выключает отображение осей (вместе с подписями) |
| 'equal' | Изменяет масштаб по осям так, чтобы равному изменению x или y соответсвовала одинаковая длинна на диаграмме. Как сказано в документации "круг превращается в круг". |
| 'scaled' | Тот же эффект, что у 'equal' достигается изменением соотношения сторон диаграммы. |
| 'tight' | Масштаб по осям изменяется так, что бы отображались все данные. Если все данные уже находятся в поле диаграммы, масштаб изменяется так, чтобы линии располагались в центре диаграммы. |

Функция legend() задает свойства и расположения легенды диаграммы. В примере использован вызов функции принимающий в качестве первых двух аргументов кортежи - первый содержит ссылки на объекты типа Line2D, второй соответствующие им подписи. Именованный аргумент loc задает расположение легенды, может быть или строкой или числом. Возможные варианты значений аргумента loc представлены в таблице.

|  |  |  |
| --- | --- | --- |
| **Значения аргумента loc** | | **Расположение легенды** |
| **Строка** | **Число** |
| 'best' | 0 | Оптимальное расположение легенды определяется автоматически. Значение по умолчанию. |
| 'upper right' | 1 | сверху справа |
| 'upper left' | 2 | сверху слева |
| 'lower left' | 3 | снизу слева |
| 'lower right' | 4 | снизу справа |
| 'right' | 5 | справа |
| 'center left' | 6 | в центре слева |
| 'center right' | 7 | в центре справа, не отличается от 'right' |
| 'lower center' | 8 | в сентер снизу |
| 'upper center' | 9 | в центре сверху |
| 'center' | 10 | в центре диаграммы |

Сформировать легенду можно и другим (на мой взгляд менее громоздким) способом. Функция plot() (если задано построение только одной линии) принимает именованный аргумент label. Это строка, подпись к линии, на основании которой вызов функции legend() без аргументов добавляет к диаграмме легенду. Вышеприведенный скрипт мог быть написан и так:

# Строим диаграмму

# Задаем исходные данные для каждой линии диаграммы, внешний вид линий и маркеров, подписи.

plt.plot(X, Y\_10, 'bD:',  label = u'Температура 10 \u00b0C')

plt.plot(X, Y\_20, 'r^:',  label = u'Температура 20 \u00b0C')

plt.plot(X, Y\_30, 'go:',  label = u'Температура 30 \u00b0C')

# Добавляем к диаграмме легенду

plt.legend(loc = 'best')

**35. Модуль matplotlib. Построение гистограмм и круговых диаграмм.**