



Встроенные типы данных



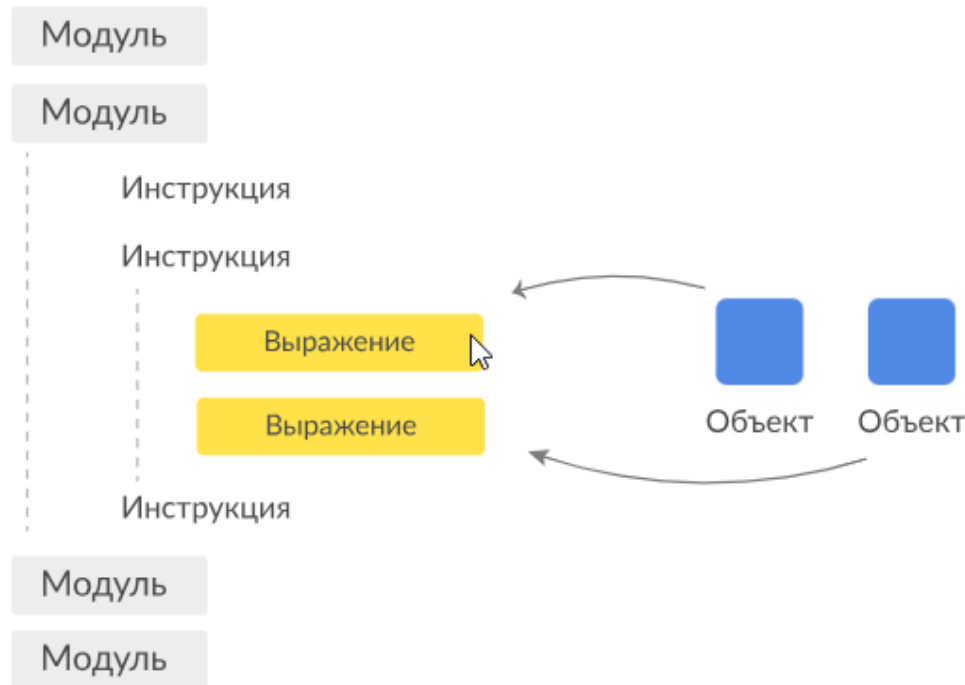
Типы данных в Python

int	float	bool	str	list	dict	set	file
complex			tuple		range	None	

Общий вид программы

1. Программа состоит из модулей;
2. Модуль, в свою очередь, представляет собой набор инструкций;
3. Инструкции содержат выражения;
4. Выражения служат для создания и обработки объектов.

Программа



Объекты — базовое понятие в Python

Все данные в Python являются объектами. Они могут создаваться нами вручную, либо быть изначально встроенными на уровне языка. Объект можно охарактеризовать, как особую область памяти, где хранятся некоторые значения и определённые для этих значений операции.

Динамическая типизация

Всё множество языков программирования можно разделить на две составляющие:

- типизированные языки;
- нетипизированные (бестиповые) языки.

Нетипизированные языки в основной своей массе сосредоточены на низком уровне, где большинство программ напрямую взаимодействует с железом. Так как компьютер "мыслит" нулями и единицами, различия между строкой и, допустим, классом для него будут заключаться лишь в наборах этих самых 0 и 1. В связи с этим, внутри бестиповых языков, близких к машинному коду, возможны любые операции над какими угодно данными. Результат на совести разработчика.

Python — типизированный язык. А, раз в нём определено понятие "типа", то должен существовать и процесс распознавания и верификации этих самых "типов". В противном случае вероятны ситуации, когда логика кода окажется нарушенной, а программа выполнится некорректно.

Таким процессом и является типизация. В ходе её выполнения происходит подтверждение используемых типов и применение к ним соответствующих ограничений. Типизация может быть **статической** и **динамической**. В первом случае, проверка выполняется во время компиляции, во втором — непосредственно во время выполнения программного кода.

Python — язык с динамической типизацией. И здесь, к примеру, одна и та же переменная, при многократной инициализации, может являть собой объекты разных типов:

```
a = 1
print(type(a))
#>> <class 'int'>
```

```
a = 'one'
print(type(a))
#>> <class 'str'>
```

```
a = {1: 'one'}
print(type(a))
#>> <class 'dict'>
```

В языке со
статической
типизацией
такой фокус не
пройдёт:

```
# // код на C++
# int main() {
#   int b = 2;
#   cout << b << "\n";
#   b = "two";
#   cout << b << "\n";
#   return 0;
# }
```

```
#>> error: invalid conversion from 'const char*' to 'int' [-fpermissive]
# b = "two";
```

Адепты и приверженцы разных языков часто спорят о том, что лучше: динамическая типизация или статическая, но, само собой, преимущества и недостатки есть и там, и там.

К плюсам динамической типизации можно отнести:

Создание разнородных коллекций

Благодаря тому, что в Python типы данных проверяются прямоком во время выполнения программного кода, ничто не мешает создавать коллекции, состоящие их элементов разных типов. Причём делается это легко и просто

```
# список, элементами которого являются строка, целое число и кортеж  
variety_list = ['String', 42, (5,25)]
```

Абстрагирование в алгоритмах

Создавая на Питоне, предположим, функцию сортировки, можно не писать отдельную её реализацию для строк и чисел, поскольку она и так корректно отработает на любом компарируемом множестве.

Простота изучения

Не секрет, что изучать Питон с нуля гораздо легче, чем, например, **Java**. И такая ситуация будет наблюдаться не только для этой пары. Языки с динамической типизацией в большинстве своём лучше подходят в качестве учебного инструмента для новичков в программировании.

К минусам же динамической проверки типов можно отнести такие моменты, как:

Ошибки

Ошибки типизации и логические ошибки на их основе. Они достаточно редки, однако зачастую весьма сложно отлавливаемые. Вполне реальна ситуация, когда разработчик писал функцию, подразумевая, что она будет принимать числовое значение, но в результате воздействия тёмной магии или банальной невнимательности, ей на вход поступает строка и ...функция отработывает без ошибок выполнения, однако её результат, — ошибка, сам по себе. Статическая же типизация исключает такие ситуации априори.

Оптимизация

Статически типизированные языки обычно работают быстрее своих динамических братьев, поскольку являются более "тонким" инструментом, оптимизация которого, в каждом конкретном случае, может быть настроена более тщательно и рационально.

Так или иначе, сказать, что "одно лучше другого" нельзя. Иначе "другого" бы не было. Динамически типизированные языки экономят уйму времени при кодировании, но могут обернуться неожиданными проблемами на этапе тестирования или, куда хуже, продакшена. Однако вряд ли кто-то будет спорить с тем, что динамический **Python** куда более дружелюбный для новичков, нежели статический **C++**.

**Разница между атомарными и
структурными типами данных**

Все типы данных Python классифицируются следующим образом:

1.Атомарные (они же типы-значения в C#):

- none
- bool (логический тип данных)
- числа
 - int (целое число)
 - float (число с плавающей точкой)
 - complex (комплексное число)
- str (строка)
- Файл

2.Ссылочные (они же ссылочные типы в C#):

- list (список)
- tuple (кортеж)
- dict (словарь)
- set (множество)
- def (функция)
- class (класс)

Атомарные объекты, при их присваивании, передаются по значению, а ссылочные — по ссылке

К **неизменяемым (immutable)** типам относятся:

- целые числа (int),
- числа с плавающей точкой (float),
- комплексные числа (complex),
- логические переменные (bool),
- кортежи (tuple),
- строки (str),
- неизменяемые множества (frozen set).

То есть, почти все **атомарные** типы являются неизменяемыми, плюс к ним добавляются кортежи и неизменяемые множества.

К **изменяемым (mutable)** типам относятся:

- списки (list),
- множества (set),
- словари (dict).

Т.е. почти все **ссылочные** типы по своим характеристикам являются изменяемыми.

Атомарные объекты, при их присваивании, передаются по значению, а
ссылочные — по ссылке

```
# пример присваивания атомарного объекта
atom = 3
btom = atom
atom = 2

print(atom)
#>> 2

print(btom)
#>> 3
```

Из результатов видно, что переменной btom было присвоено именно значение, содержащееся в atom, а не ссылка, указывающая на область памяти.

```
# пример присваивания ссылочного объекта
link = ['Ipona', 'Master Sword']
alin = link
link[0] = 'Zelda'

print(link)
#>> ['Zelda', 'Master Sword']

print(alin)
#>> ['Zelda', 'Master Sword']
```

Поскольку списки — это ссылочные объекты, то вполне закономерно, что после присваивания переменной `link` переменной `alin` передалась именно ссылка на объект `list`-а и, при печати, на экран были выведены две одинаковые надписи.

```
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021,  
Type "help", "copyright", "credits" or "license"  
>>> a = 2  
>>> b = a  
>>> a = 3  
>>> b  
2  
>>>
```

```
>>> a = [1,2,3]  
>>> b = a  
>>> a[0] = 6  
>>> b  
[6, 2, 3]  
>>>
```



```
>>> a = [1,2,3]
>>> b = a
>>> a[0] = 6
>>> b
[6, 2, 3]
>>> id(a)
2298151680704
>>> id(b)
2298151680704
>>> _
```

Указывает на одну и ту же область памяти.
Идентичные объекты

Числовые типы

"Все сущее есть Число" — сказал однажды мудрый грек по имени Пифагор. Числа — важнейший и фундаментальнейший из всех типов данных для всех языков программирования. В Python для их представления служит числовой тип данных.

int (целое число)

Это числа без дробной части, которые, говоря математическим языком, являются расширением натурального ряда, дополненного нулём и отрицательными числами.

Целые числа в Python представлены только одним типом — **PyLongObject**, реализация которого лежит в **longobject.c**, а сама структура выглядит так:

```
struct _longobject {  
    PyObject_VAR_HEAD  
    digit ob_digit[1]; # массив цифр  
};
```

Любое целое число состоит из массива цифр переменной длины, поэтому в Python 3 в переменную типа **int** может быть записано число неограниченной длины. Единственное ограничение длины — это размер оперативной памяти.

float (число с плавающей точкой)

Действительные или вещественные числа придуманы для измерения непрерывных величин. В отличие от математического контекста, ни один из языков программирования не способен реализовать бесконечные или иррациональные числа, поэтому всегда есть место приближению с определенной точностью, из-за чего возможны такие ситуации:

```
print(0.3 + 0.3 + 0.3)
#>> 0.8999999999999999
```

```
print(0.3 * 3 == 0.9)
#>> False
```

Информацию о точности и внутреннем представлении float для вашей системы можно получить из `sys.float_info`

```
#> >>> import sys
#> >>> sys.float_info
#> sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
#> min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
#> mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Если нужна высокая точность обычно используют модули `Decimal` и `Fraction`.

complex (комплексное число)

Привет высшей математике! Как вещественный ряд расширяет множество рациональных чисел, так и ряд комплексных чисел расширяет множество вещественных. Показательной особенностью комплексного ряда является возможность извлечения корня из отрицательных чисел.

```
# пример комплексного числа
```

```
z = complex(1, 2)
```

```
print(z)
```

```
#>> (1+2j)
```

```
# вещественная часть
```

```
print(z.real)
```

```
#>> 1.0
```

```
# мнимая часть
```

```
print(z.imag)
```

```
#>> 2.0
```

```
# сопряженное комплексное число
```

```
print(z.conjugate())
```

```
#>> (1-2j)
```

операция сравнения для комплексных чисел не определена

bool (логический тип данных)

- Истина (True);
- Ложь (False).

```
# пример bool  
pravda = True  
lozh = False
```

Переменные логического типа нужны для реализации ветвлений, они применяются для установки флажков, фиксирующих состояния программы, а также используются в качестве возвращаемых значений для функций, названия которых, зачастую, начинаются на **"is"** (**isPrime**, **isEqual**, **isDigit**). То есть тех, которые, на человеческом языке, отвечали бы на вопрос одним словом "Да" или "Нет".

Конвертация типов чисел

Конвертация типа — это метод конвертации числа из одного типа в другой. Для этого можно использовать функции `float()`, `int()` и `complex()`.

```
x = 1 # создание целого числа
```

```
y = 2.0 # создание числа с плавающей точкой
```

```
z = 2+3j # создание комплексного числа
```

```
a = float(x) # преобразование целого числа в число с плавающей точкой
```

```
b = int(x) # преобразование числа с плавающей точкой в целое число
```

```
c = complex(x) # преобразование целого числа в комплексное
```

```
d = complex(y) # преобразование числа с плавающей точкой в комплексное
```

```
print(a, type(a))
```

```
print(b, type(b))
```

```
print(c, type(c))
```

```
print(d, type(d))
```

```
#>>
```

```
#> 1.0
```

```
#> 1
```

```
#> (1+0j)
```

```
#> (2+0j)
```

Случайные числа

В Python нативной поддержки случайных чисел нет, но есть встроенный модуль `random`, который используется для работы с целыми числами.

```
import random  
print(random.randrange(1, 1000))
```


Встроенные математические функции

```
>>> a = 4
>>> b = 7
>>> a + b # Сложение
11
>>> a - b # Вычитание
-3
>>> a / b # Деление
0.5714285714285714
>>> a ** b # Возведение в степень
16384
>>> a // b # Целочисленное деление
0
>>> a % b # Остаток от деления
4
```

```
>>> a + b
11
>>> a.__add__(b)
11
```

abs(x) — модуль числа

int(x) — преобразование к целому числу

round(x) — округление

pow(x) - возведения числа в степень, тоже что и с помощью оператора **

Модуль math существенно расширяет математические возможности языка

```
import math
```

math.pi — число «пи»

math.sqrt(x) — квадратный корень

math.sin(x) — синус угла, заданного **в радианах**

math.cos(x) — косинус угла, заданного **в радианах**

math.exp(x) — экспонента e^x

math.ln(x) — натуральный логарифм

math.floor(x) — округление «вниз»

math.ceil(x) — округление «вверх»

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__sign', 'cos', 'cosh', 'degrees', 'dist',
, 'hypot', 'inf', 'isclose', 'isfinite',
tafter', 'perm', 'pi', 'pow', 'prod', 'ra
>>> _
```

На что стоит обратить внимание, т.е. что необычного может быть в Python, в отличие от других языков программирования:

1.Динамическую типизацию и неявное преобразование типов.

1.Функция id() позволяет посмотреть на объект с каким идентификатором ссылается данная переменная.

Идентификатор – это некоторое целочисленное значение, посредством которого уникально адресуется объект.

2.Функция type() позволяет определить тип переменной.

3.Изменяемые и неизменяемые типы данных.

4.Размер целого числа ограничивается только объемом памяти компьютера. Однако, скорость работы с большими числами существенно медленнее, чем с числами, которые соответствуют машинному представлению.

5.Модуль math, который существенно расширяет математические возможности языка.

6.Получение срезов строк. Отдельные элементы последовательности, а, следовательно, и отдельные символы в строках, могут извлекаться с помощью оператора доступа к элементам ([]).

7.[f-строки.](#)

Последовательности

Последовательность - упорядоченная коллекция объектов

- **строки** (str) - ссылочный, неизменяемый тип
- **кортежи** (tuple) - ссылочный, неизменяемый тип
- **список** (list) - ссылочный, изменяемый тип
- **множество** (set) - ссылочный, изменяемый тип
- **словарь** (dict) - ссылочный, изменяемый тип

Важной характеристикой всех этих типов данных является то, что они являются итерируемыми, т.е. возможно перебрать все значения последовательности (коллекции).

Строки (str) - неизменяемый тип

Строки в языке программирования Python — это объекты, которые состоят из последовательности символов.

Строковый тип данных — это последовательности символов Unicode любой длины, заключённые в одинарные, двойные или тройные кавычки. Символами могут быть буквы, числа, знаки препинания, специальные символы и так далее. Главное — чтобы их с двух сторон окружали кавычки. В Python текстовый тип данных обозначается буквами str (от англ. string — строка).

```
string = 'Hello, world!'
print(string)
#>> Hello, world!
print(type(string))
#>> <class 'str'>
```

Литералы строк

Литерал — способ создания объектов, в случае строк Питон предлагает несколько основных вариантов:

```
#>>> 'string' # одинарные кавычки  
'string'
```

```
#>>> "string" # двойные кавычки  
'string'
```

```
#>>> """string"""  
'string'
```

```
#>>> '''string'''  
'string'
```

```
#>>> 'book "war and peace"' # разный тип кавычек  
'book "war and peace"'
```

```
#>>> "book 'war and peace'" # разный тип кавычек  
"book 'war and peace'"
```

```
#>>> "book \"war and peace\"" # экранирование  
кавычек одного типа  
'book "war and peace"'
```

```
#>>> 'book \'war and peace\'' # экранирование  
кавычек одного типа  
"book 'war and peace'"
```

Разницы между строками с одинарными и двойными кавычками нет — это одно и то же.

Кодировка строк

В **Python 3** кодировка по умолчанию исходного кода — **UTF-8**. Во второй версии по умолчанию использовалась **ASCII**. Если необходимо использовать другую кодировку, можно разместить специальное объявление на первой строке файла, к примеру:

```
# -*- coding: windows-1252 -*-
```

Максимальная длина строки в Python

Максимальная длина строки зависит от платформы. Обычно это:

$2^{31} - 1$ — для 32-битной платформы;

$2^{63} - 1$ — для 64-битной платформы;

Константа `maxsize`, определенная в модуле `sys`:

```
#>>> import sys  
#>>> sys.maxsize  
# 2147483647
```


Индексация строк

Часто в языках программирования, отдельные элементы в упорядоченном наборе данных могут быть доступны с помощью числового индекса или ключа. Этот процесс называется индексация.

В Python строки являются упорядоченными последовательностями символьных данных и могут быть проиндексированы. Доступ к отдельным символам в строке можно получить, указав имя строки, за которым следует число в квадратных скобках [].

Индексация строк начинается с нуля: у первого символа индекс 0, следующего 1 и так далее. Индекс последнего символа в python — “длина строки минус один”.

s	k	i	l	l	b
0	1	2	3	4	5

Индексы строк также могут быть указаны отрицательными числами. В этом случае индексирование начинается с конца строки: -1 относится к последнему символу, -2 к предпоследнему и так далее.

Обращение по индексу

Для выбора определенного символа из строки можно воспользоваться обращением по индексу, записав его в квадратных скобках:

```
#>>> s = "abcdef"
#>>> s[0]
'a'
#>>> s[2]
'c'
```

```
#>>> s = "abcdef"
#>>> s[-1]
'f'
```

Специальные символы

В строке могут присутствовать специальные символы, такие как перенос строки и т. п.

`\n` – перевод строки

`\r` – возврат каретки

`\t` – знак табуляции

`\"` – двойная кавычка

`\'` – апостроф

`\\` – обратный слеш

Конкатенация строк

Одна из самых распространенных операций со строками — их объединение (конкатенация). Для этого используется знак +, в результате к концу первой строки будет дописана вторая:

```
>>> s1 = "Hello" + " world"  
>>> s2 = " world"  
>>> s1+s2  
#>> 'Hello world world'
```

```
>>> s = "AY"  
>>> s5 = s*5  
#>> 'AYAYAYAYAY'
```

Как удалить строку в Python

Строки, как и некоторые другие типы данных в языке Python, являются неизменяемыми объектами. При задании нового значения строке просто создается новая, с заданным значением. Для удаления строки можно воспользоваться методом `replace()`, заменив ее на пустую строку:

```
>>> s = "test"  
>>> s.replace("test", "")  
#>> ""
```

Сравнение строк

При сравнении нескольких строк рассматриваются отдельные символы и их регистр:

- цифра условно меньше, чем любая буква из алфавита;
- алфавитная буква в верхнем регистре меньше, чем буква в нижнем регистре;
- чем раньше буква в алфавите, тем она меньше;

При этом сравниваются по очереди первые символы, затем — 2-е и так далее.

Срезы строк

Python также допускает возможность извлечения подстроки из строки, известную как “*string slice*”.

`s[m:n]`

Если `s` это строка, выражение возвращает часть `s`, начинающуюся с позиции `m`, и до позиции `n`, но не включая позицию `n`:

```
>>> s = 'python'
>>> s[2:5]
'tho'
```

```
>>> s = 'python'
>>> s[:4]
'pyth'
>>> s[0:4]
'pyth'
```

Срезы эффективно работают для строк, но в принципе их можно использовать для других структур данных.

Срезы строк (выделение части строки)

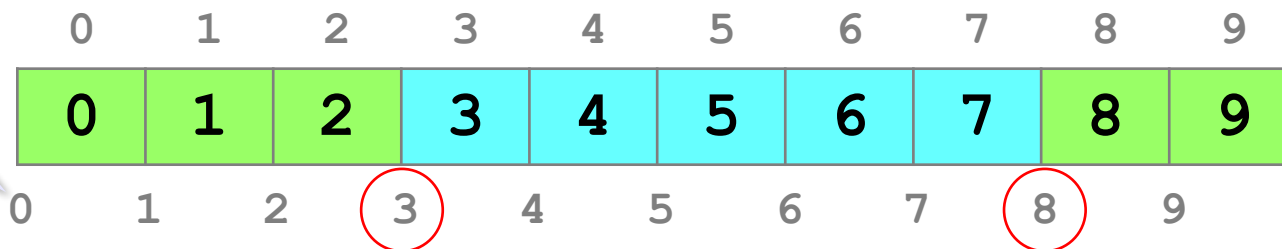
```
s = "0123456789"
```

```
s1 = s[3:8] # "34567"
```

с какого
символа

до какого
(не включая 8)

разрезы



Срезы строк (выделение части строки)

```
s = "0123456789"  
s1 = s[3:8]           # "34567"
```

с какого
символа

до какого
(не включая 8)

```
s = "0123456789"  
s1 = s[:8]            # "01234567"
```

от начала строки

```
s = "0123456789"  
s1 = s[3:]            # "3456789"
```

до конца строки

Срезы строк

Срезы с отрицательными индексами:

```
s = "0123456789"
```

```
s1 = s[:-2] # "01234567"
```

`len(s) - 2`

```
s = "0123456789"
```

```
s1 = s[-6:-2] # "4567"
```

`len(s) - 6`

`len(s) - 2`

```
s1 = s[::-1] # "9876543210"
```

реверс строки

Операции со строками

Удаление:

```
s = "0123456789"
```

```
s1 = s[:3] + s[9:]
```

"012"

"9"

"0129"

Вставка:

```
s = "0123456789"
```

```
s1 = s[:3] + "ABC" + s[3:]
```

"012"

"3456789"

"012ABC3456789"

Встроенные функции строк в python

Функция `ord(c)` возвращает числовое значение для заданного символа.

Функция `chr(n)` возвращает символьное значение для данного целого числа.

Функция `len(s)` возвращает длину строки.

Функция `str(obj)` возвращает строковое представление объекта.

```
>>> ord('a')
97
>>> ord('#')
35
```

```
>>> chr(97)
'a'
>>> chr(35)
'#'
```

```
>>> s = 'Простая строка.'
>>> len(s)
15
```

```
>>> str(49.2)
'49.2'
>>> str(3+4j)
'(3+4j)'
>>> str(3 + 29)
'32'
>>> str('py')
'py'
```

```
>>> s = "Длинная строка, на которой можно показать срезы"
>>> s[3:7] # С 3 элемента по 7 не включительно
'нная'
>>> s[3:] # С 3 до конца
'нная строка, на которой можно показать срезы'
>>> s[:3] # По 3 элемент
'Дли'
>>> s[2:20:2] # Со 2 по 20 с шагом 2
'иньясрк,н '
>>> s[::-2] # Каждый второй
'Диньясрк,н ооо он оааьсеы'
```

```
>>> s = "Длинная строка, на которой можно показать срезы"
>>> s[::-1]
'ызерс ьтазаконжон йороток ан ,акортс яаннилД'
```

Преобразование из строки в другой тип

string → int

Функция `int()` преобразовывает целое число в десятичной системе, заданное как строка, в тип `int`:

```
>>> int("10")  
#>> 10
```

string → float

```
>>> float('1.5')  
#>> 1.5
```

string → list

Самый простой способ преобразования строки в список строк — метод `split()`:

```
>>> 'one two three four'.split()  
#>> ['one', 'two', 'three', 'four']
```

string → bytes

Преобразование строкового типа в байтовый выполняется функцией `encode()` с указанием кодировки:

```
>>> 'Байты'.encode('utf-8')  
#>> b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd1\x8b'
```

string → datetime

Строка в дату преобразовывается функцией `strptime()` из стандартного модуля `datetime`:

```
>>> from datetime import datetime
>>> print(datetime.strptime('Jan 1 2020 1:33PM', '%b %d %Y %l:%M%p'))
#>> 2023-08-18 10:33:00
```

string → dict

Создание словаря из строки возможно, если внутри нее данные в формате `json`. Для этого можно воспользоваться модулем `json`:

```
>>> import json
>>> print(json.loads('{"Russia": "Moscow", "France": "Paris"}'))
#>> {'Russia': 'Moscow', 'France': 'Paris'}
```

string → json

Конвертация объектов Python в объект `json` выполняется функцией `dumps()`:

```
>>> import json
>>> json.dumps("hello")
#>> '"hello"'
```

Стандартные функции

Верхний/нижний регистр:

```
s = "aAbBcC"  
s1 = s.upper()    # "AABVCC"  
s2 = s.lower()    # "aabbcc"
```

Проверка на цифры:

```
s = "abc"  
print ( s.isdigit() )    # False  
s1 = "123"  
print ( s1.isdigit() )    # True
```

Поиск в строках

```
s.find()
```

```
s.rfind()
```

Подсчет количества вхождений

```
s.count ( "c" )
```



```
>>> s = "Строка"
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
 'mod', 'mul', 'ne', 'new', 'reduce', ...]
```

Python Console

```
>>> help(s.swapcase)
Help on built-in function swapcase:

swapcase() method of builtins.str instance
    Convert uppercase characters to lowercase and lowercase characters
    to uppercase.

>>> s.swapcase()
'cTpOkA'
```

list (список) - изменяемый тип

Создание списка в Python может понадобиться для хранения в них коллекции объектов. Списки могут хранить объекты всех типов в одном, в отличие от массива в другом языке программирования. Также размер списка доступен к изменению.

Список (list) - тип данных, предназначенный для хранения набора или последовательности разных элементов.

Как списки хранятся в памяти?

Базовая C-структура списков в Python (CPython) выглядит следующим образом:

```
# typedef struct {  
#   PyObject_VAR_HEAD  
#   PyObject **ob_item;  
#   Py_ssize_t allocated;  
# } PyListObject;
```

Когда мы создаём список, в памяти под него резервируется объект, состоящий из 3-х частей:

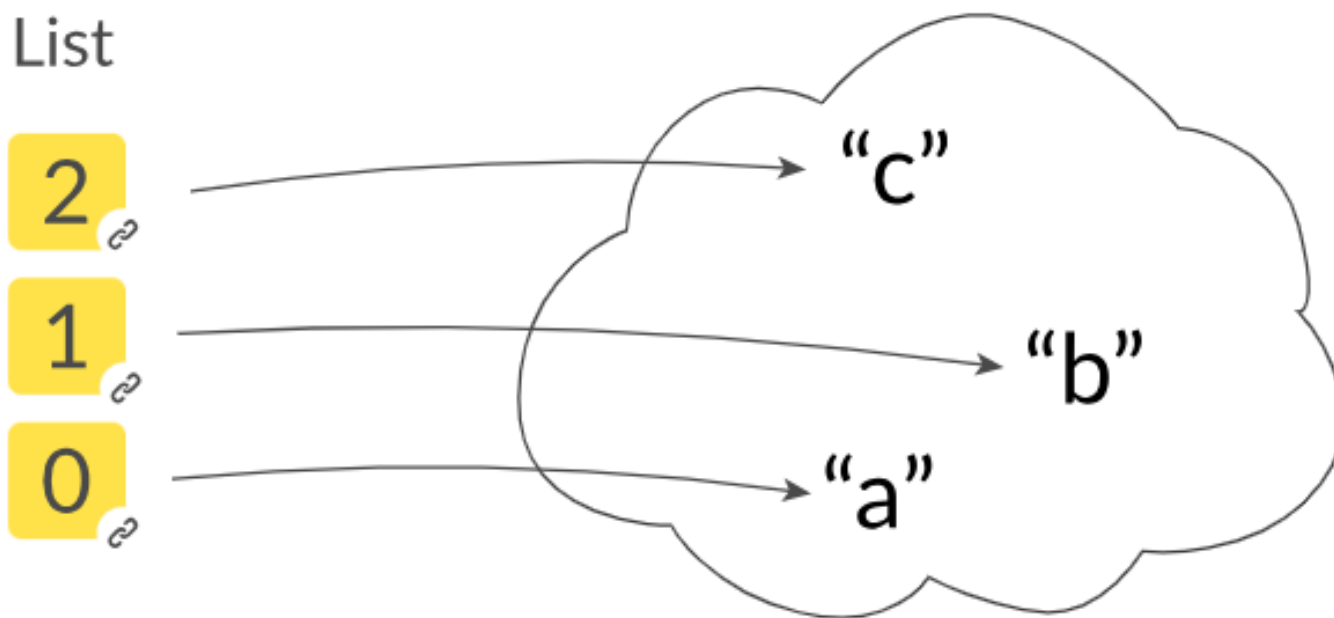
`PyObject_VAR_HEAD` — заголовок;

`ob_item` — массив указателей на элементы списка;

`allocated` — количество выделенной памяти под элементы списка.

Объект списка хранит указатели на объекты, а не на сами объекты

Python размещает элементы списка в памяти, затем размещает указатели на эти элементы. Таким образом, список в Python — это массив указателей.



Список в Python — это массив указателей на элементы, размещенные в памяти

Базовая работа со списками

Срезы (slices) — это подмножества элементов списка. Срез нужен, когда необходимо извлечь часть списка из полного списка.

`elements[START:STOP:STEP]`

В этом случае берётся срез от номера start (включительно) до stop (не включая его), а step — это шаг. По умолчанию start и stop равны 0, step равен 1.

```
>>> elements = [0.1, 0.2, 1, 2, 3, 4, 0.3, 0.4]
>>> int_elements = elements[2:6] # с 2-го элемента включительно по 6-й элемент

>>> print(id(elements), id(int_elements)) # elements и int_elements - 2 разных списка
53219112 53183848

>>> print(elements)
[0.1, 0.2, 1, 2, 3, 4, 0.3, 0.4] # срез не модифицирует исходный список

>>> print(int_elements)
[1, 2, 3, 4]
```

Объявление списка

```
list = []
```

```
>>> elements = [1, 3, 5, 6]
```

```
>>> elements = [1, 3, 5, 6]
```

```
>>> type(elements)  
#>> <class 'list'>
```

```
>>> print(elements)  
#>> [1, 3, 5, 6]
```

Через функцию list()

```
>>> elements = list()
```

```
>>> elements = list()
```

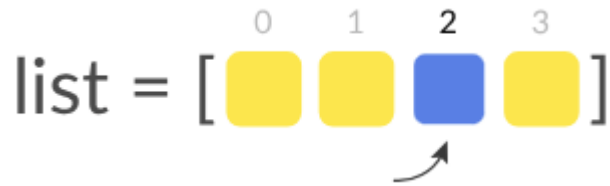
```
>>> type(elements)  
#>> <class 'list'>
```

```
>>> print(elements)  
#>> []
```

Длина списка

Функция len()

Обращение к элементу списка в Python



```
>>> elements = [1, 2, 3, 'word']
```

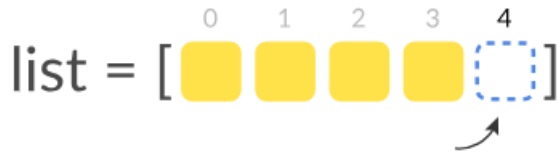
```
>>> elements[3]
```

```
#>> 'word'
```

Индекс — это порядковый номер элемента в списке. Чтобы обратиться к элементу списка, достаточно указать его индекс. Нумерация элементов списка в Python начинается с нуля.

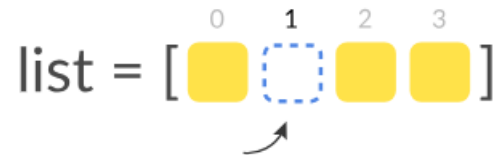
Добавление в список

```
list.append(x)
```



Добавление в список на указанную позицию

```
list.insert(1, x)
```



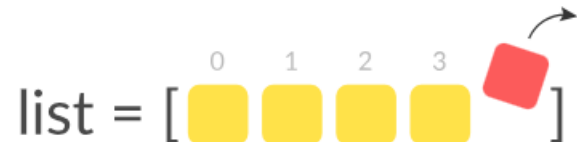
Изменение элементов списка

```
elements[2] = 8
```



Удаление элемента из списка

```
del list[4]
```



```
>>> lst = [23, 4, 324]
>>> lst.append(17)
>>> lst
[23, 4, 324, 17]
>>> lst.insert(1, 17)
>>> lst
[23, 17, 4, 324, 17]
>>> lst.pop()
17
>>> lst
[23, 17, 4, 324]
>>> lst.pop(0)
23
>>> lst
[17, 4, 324]
>>> |
```

Как проверить наличие элемента в списке

Для того чтобы проверить существование какого-либо элемента в списке, нужно воспользоваться оператором *in*.

```
>>> elements = ['слон', 'кот', 'лошадь', 'змея', 'рыба']
>>> if 'кот' in elements:
    print('meow')

#>> meow
```

Объединение списков

list = [ ⁰  ¹  ²] + [ ⁰  ¹  ²]

Списки в Python можно объединять с помощью оператора + или метода extend.

```
>>> a = [1, 3, 5]
>>> b = [1, 2, 4, 6]
>>> print(a + b)
#>> [1, 3, 5, 1, 2, 4, 6]

>>> hello = ["h", "e", "l", "l", "o"]
>>> world = ["w", "o", "r", "l", "d"]
>>> hello.extend(world) # extends не возвращает
# новый список, а дополняет текущий
>>> print(hello)
#>> ['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']
```


Копирование списка Python

Если скопировать список оператором `=`, вы скопируете не сам список, а только его ссылку.

Для копирования списков можно использовать несколько вариантов:

- **`elements.copy()`** — встроенный метод `copy` (доступен с Python 3.3);
- **`list(elements)`** — через встроенную функцию `list()`;
- **`copy.copy(elements)`** — функция `copy()` из пакета `copy`;
- используя срезы.

Цикл по списку



```
elements = [1, 2, 3, "meow"]
```

```
for el in elements:
```

```
    print(el)
```

```
#>> 1
```

```
#>> 2
```

```
#>> 3
```

```
#>> meow
```

list -> str

Перевод списка в строку осуществляется с помощью функции join().

```
>>> fruits = ["яблоко", "груша", "ананас"]
```

```
>>> print(', '.join(fruits))
```

```
яблоко, груша, ананас
```

list -> dict

```
>>> elements = [['1', 'a'], ['2', 'b'], ['3', 'c']]
```

```
>>> my_dict = dict(elements)
```

```
>>> print(my_dict)
```

```
#> {'1': 'a', '2': 'b', '3': 'c'}
```

list -> json

JSON — это JavaScript Object Notation. В Python находится встроенный модуль json для кодирования и декодирования данных JSON. С применением метода json.dumps(x) можно запросто преобразовать список в строку JSON.

```
>>> import json
```

```
>>> json.dumps(['word', 'eye', 'ear'])
```

```
'["word", "eye", "ear"]'
```

Кортежи (tuple) - неизменяемый тип

По своей природе они очень схожи со списками, но, в отличие от последних, являются неизменяемыми.

Особенности кортежей:

- они упорядочены по позициям;
- tuple могут хранить и содержать внутри себя объекты любых типов (и даже составных);
- доступ к элементам происходит по смещению, а не по ключу;
- в рамках этой структуры данных определены все операции, основанные на применении смещения (индексирование, срез);
- кортежи поддерживают неограниченное количество уровней вложенности;
- кортежи хранят указатели на другие объекты, а значит их можно представлять, как массивы ссылок;
- они позволяют очень просто менять местами значения двух переменных.

Зачем использовать кортеж вместо списка?

- **Неизменяемость** — именно это свойство кортежей, порой, может выгодно отличать их от списков.
- **Скорость** — кортежи быстрее работают. По причине неизменяемости кортежи хранятся в памяти особым образом, поэтому операции с их элементами выполняются заведомо быстрее, чем с компонентами списка.
- **Безопасность** — неизменяемость также позволяет им быть идеальными кандидатами на роль констант. Константы, заданные кортежами, позволяют сделать код более читаемым и безопасным.
- **Использование tuple в других структурах данных** — кортежи применимы в отдельных структурах данных, от которых python требует неизменяемых значений. Например ключи словарей (dicts) должны состоять исключительно из данных immutable-типа.

Кроме того, кортежи удобно использовать, когда необходимо вернуть из функции несколько значений.

```
# пустой кортеж
empty_tuple = ()

# кортеж из 4-х элементов разных типов
four_el_tuple = (36.6, 'Normal', None, False)

# пример tuple, что содержит вложенные элементы
nested_elem_tuple = (('one', 'two'), ['three', 'four'],
                     {'five': 'six'}, (('seven', 'eight'), ('nine', 'ten')))
print(nested_elem_tuple)

#>> (('one', 'two'), ['three', 'four'],
# {'five': 'six'}, (('seven', 'eight'), ('nine', 'ten')))
```

Метод **index()** позволяет получить индекс элемента. Достаточно передать нужное значение элемента, как аргумент метода.

Метод **count()** ведёт подсчет числа вхождений элемента в кортеж.

Создание

`tuple= ()`

```
literal_creation = ('any', 'object')
```

```
print(literal_creation)
```

```
#>> ('any', 'object')
```

```
print(type(literal_creation))
```

```
#>> <class 'tuple'>
```

Упаковка

`tuple = (,)`

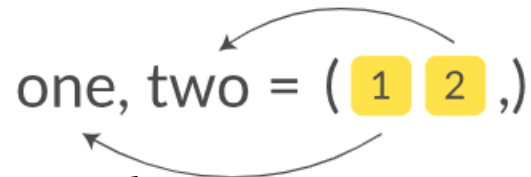
Упаковкой кортежа называют присваивание его какой-то переменной, что, по сути, совпадает с операцией объявления.

Стоит обратить внимание 2 момента:

- Выражения `some_tuple = (11, 12, 13)` и `some_tuple = 11, 12, 13` тождественны.
- Для объявления кортежа, включающего один единственный элемент, нужно использовать завершающую запятую.

Распаковка

`one, two = (1 2 ,)`



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

```
night_sky = 'Moon', 'Stars'  
moon, _ = night_sky  
print(moon)
```

```
#>> Moon
```


Удаление

Добавить или удалить элемент содержащийся в tuple нельзя, по причине всё той же неизменяемости. Однако сам кортеж стереть с цифрового лица Земли возможно.

Оператор `del` к нашим услугам:

```
some_useless_stuff = ('sad', 'bad things', 'trans fats')
del some_useless_stuff
print(some_useless_stuff)
```

```
#>>
```

```
#> Traceback (most recent call last):
```

```
#> print(some_useless_stuff)
```

```
#> NameError: name 'some_useless_stuff' is not defined
```

Словарь (dict) - изменяемый тип

Словари в Python можно считать реализацией структуры данных, более известной как ассоциативный массив.

Словарь (dictionary) - это тип данных, представляющий собой неупорядоченный набор пар ключ:значение. (при этом каждый ключ, в рамках одного словаря, является уникальным).

```
# литерал словаря в Python, где first_key и second_key - ключи,  
# а 1 и 2 - соответственно ассоциированные с ними значения
```

```
{'first_key': 1, 'second_key': 2}
```

Базовая работа со словарями

Объявление словаря

```
example_dict = {}  
print(type(example_dict))  
  
#>> <class 'dict'>
```

dict = {**k**:**v**}

Добавление нового элемента в словарь

```
superhero_dict = {'dc_hero': 'Flash'}  
  
superhero_dict['dark_horse_hero'] = 'Hellboy'  
print(superhero_dict)  
  
#>> {'dc_hero': 'Flash', 'dark_horse_hero': 'Hellboy'}
```

Удаление элемента из словаря

```
# запись "'dark_horse_hero': 'Hellboy'" исчезнет. Прости, Красный!  
del superhero_dict['dark_horse_hero']  
print(superhero_dict)  
  
#>> {'dc_hero': 'Batwoman'}
```

Перебор словаря в Python

```
iter_dict = {'key_b': 1, 'key_d': 0, 'key_e': -2, 'key_c': 95, 'key_a': 13}
for key in iter_dict:
    print(key, end=' ')

#>> key_b key_d key_e key_c key_a
```

```
iter_dict = {'key_b': 1, 'key_d': 0, 'key_e': -2, 'key_c': 95, 'key_a': 13}
for item in iter_dict.items():
    print(item, end=' ')

#> ('key_b', 1)('key_d', 0)('key_e', -2)('key_c', 95)('key_a', 13)
```

Объединение словарей

```
showcase_1 = {'Apple': 2.7, 'Grape': 3.5, 'Banana': 4.4}
```

```
showcase_2 = {'Orange': 1.9, 'Coconut': 10}
```

```
showcase_1.update(showcase_2)
```

```
print(showcase_1)
```

```
#>> {'Apple': 2.7, 'Grape': 3.5, 'Banana': 4.4, 'Orange': 1.9, 'Coconut': 10}
```

Ограничения

Создавая словарь, вы не должны забывать о некоторых ограничениях, накладываемых, в основном, на его ключи.

- Данные, представляющие собой ключ словаря, должны быть уникальны внутри множества ключей этого словаря. Проще говоря, не должно быть двух одинаковых ключей;
- Ключ должен быть объектом неизменяемого типа, то есть строкой, числом или кортежем. Если говорить строже, то объект содержащий ключ должен быть `hashable`. То есть иметь хеш-значение, которое не меняется в течение его жизненного цикла;
- На значения нет никаких ограничений. Максимальный уровень свободы. Они не обязаны быть ни уникальными, ни неизменяемыми, поэтому могут себе позволить быть какими угодно.

dict → json

```
import json
```

Существует два схожих метода:

- **dump()** позволит вам конвертировать питоновские словари в json объекты и сохранять их в файлы на вашем компьютере. Это несколько напоминает работу с csv.
- **dumps()** запишет словарь в строку Python, но согласно json-формату.

```
phonebook = dict(j_row='John Connor', s_row='Sarah Connor')  
phonebook_json = json.dumps(phonebook)  
print(phonebook_json)
```

```
#>> {"j_row": "John Connor", "s_row": "Sarah Connor"}  
print(type(phonebook_json))  
#>> <class 'str'>
```


Множество (set) - изменяемый тип

Множества (set) в питоне представлены как **неупорядоченные** коллекции **уникальных** и **неизменяемых** объектов.

Множества:

- Дают возможность быстро удалять дубликаты, поскольку, по определению, могут содержать только уникальные элементы;
- Позволяют, в отличие от других коллекций, выполнять над собой ряд математических операций, таких как объединение, пересечение и разность множеств.

```
pangram = {'съешь же ещё этих мягких французских булок, да выпей чаю'}  
print(pangram)  
#>> {'съешь же ещё этих мягких французских булок, да выпей чаю'}
```

```
pangram_second = set('съешь же ещё этих мягких французских булок, да выпей чаю')  
print(pangram_second)  
# попить чаю с функцией set(), к сожалению, не выйдет  
#>> {'щ', 'ь', 'э', 'н', 'з', 'с', 'м', ' ', 'р', 'о', 'ю', 'ш',  
#  'ё', 'к', 'у', 'е', 'л', 'в', 'г', 'ы', 'ъ', 'х', 'ж', 'ц',  
#  'п', 'и', ',', 'ч', 'а', 'т', 'й', 'ф', 'д', 'я', 'б'}
```

Frozenset

Frozen set-ы, по сути, отличаются от обычных лишь тем, что являются неизменяемым типом данных, в то время, как простой set возможно изменять.

```
ordinary_set = set()
frozen_set = frozenset()

print(ordinary_set == frozen_set)
#>> True

# на них также определены теоретико-множественные операции
print(type(ordinary_set - frozen_set))
#>> <class 'set'>

# отличие кроется в неизменяемости
ordinary_set.add(1)
print(ordinary_set)
#>> {1}

frozen_set.add(1)

#>> Traceback(most recent call last):
#>>   frozen_set.add(1)
#>> AttributeError: 'frozenset' object has no attribute 'add'
```

Пересечение

```
overlap = set(unbreakable_diamond) & set(golden_wind)
```

Добавление элемента

Для добавления нового элемента в существующий набор используем метод **add(x)**.

Удаление и очистка

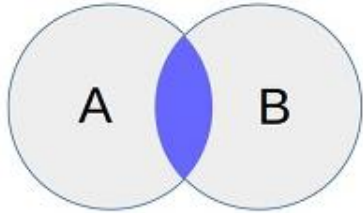
Очистить и свести уже существующий сет к пустому не составит никаких проблем благодаря методу **clear()**

Метод **remove()** удаляет элемент `elem` из `set`-а. В случае отсутствия `elem` в наборе интерпретатор выбрасывает исключение

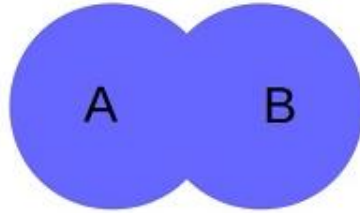
Метод **discard()** производит предельно схожую с `remove()` операцию с той лишь разницей, что, в случае отсутствия элемента в коллекции, исключение не возникает

Метод **pop()** удаляет и возвращает случайный элемент множества.

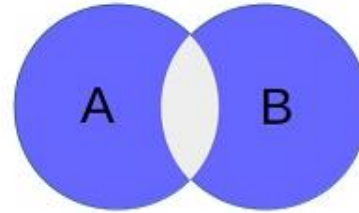
Пересечение $A \cap B$



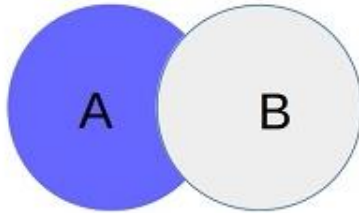
Объединение $A \cup B$



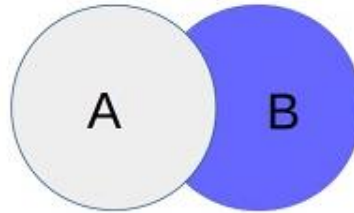
Симметричная разность $A \oplus B$



Разница $A - B$



Разница $B - A$



Создание множества

$a = \{1, 2, 3, 4\}$

$b = \{3, 4, 5, 6\}$

Объединение множеств (`union`)

Объединение возвращает множество, содержащее все элементы обоих множеств.

```
c = a.union(b)
```

или можно использовать оператор |

```
c = a | b
```

```
print(c) # {1, 2, 3, 4, 5, 6}
```

Пересечение множеств (`intersection`)

Пересечение возвращает множество, содержащее только общие элементы двух множеств.

```
c = a.intersection(b)
```

или можно использовать оператор &

```
c = a & b
```

```
print(c) # {3, 4}
```

Разность множеств (`difference`)

Разность возвращает множество, содержащее элементы, которые находятся только в одном множестве.

```
c = a.difference(b)
```

или можно использовать оператор -

```
c = a - b
```

```
print(c) # {1, 2}
```

Симметрическая разность множеств (`symmetric_difference`)

Симметрическая разность возвращает множество, содержащее элементы, которые находятся в одном из множеств, но не в обоих сразу.

```
c = a.symmetric_difference(b)
```

или можно использовать оператор ^

```
c = a ^ b
```

```
print(c) # {1, 2, 5, 6}
```

Проверка подмножества (`issubset`)

Возвращает `True`, если все элементы одного множества содержатся в другом множестве.

```
c = {1, 2}
```

```
print(c.issubset(a)) # True
```

Проверка надмножества (`issuperset`)

Возвращает `True`, если множество содержит все элементы другого множества.

```
print(a.issuperset(c)) # True
```

Проверка пересечения множеств (`isdisjoint`)

Возвращает `True`, если два множества не содержат общих элементов.

```
print(a.isdisjoint(b)) # False
```

range object (a type of iterable)

Крутая особенность языка Python состоит в наличии в нём встроенной функции `range()`, которая способна генерировать непрерывную последовательность целых чисел:

```
r = range(10)
print(r)
#>> range(0, 10)

for i in r:
    print(i, end=' ')
#>> 0 1 2 3 4 5 6 7 8 9
```

Работа с файлами

Чтобы начать работу с файлами, нужно вызвать функцию `open()` и передать ей в качестве аргументов имя файла из внешнего источника и строку, описывающую режим работы функции:

```
f = open('filename.txt', 'w')
```

В этом примере `f` — переменная-указатель на текстовый файл `.txt`.

Операции с файлами могут быть разными, а, следовательно, разными могут быть и режимы работы с ними:

- `r` — выбирается по умолчанию, означает открытие файла для чтения;
- `w` — файл открывается для записи (если не существует, то создаётся новый);
- `x` — файл открывается для записи (если не существует, то генерируется исключение);
- `a` — режим записи, при котором информация добавляется в конец файла, а не затирает уже имеющуюся;
- `b` — открытие файла в двоичном режиме;
- `t` — ещё одно значение по умолчанию, означающее открытие файла в текстовом режиме;
- `+` — читаем и записываем.

Запись в файл:

```
file.write()
```


None

None — специальный объект внутри Питона. Он означает пустоту, всегда считается "Ложью" и может рассматриваться в качестве аналога NULL для языка C/C++. Помимо этого, None возвращается функциями, как объект по умолчанию.

Работа с типами в Python

Как проверить тип данных

```
# достаточно воспользоваться встроенной функцией type()
my_list = {'Python': 'The best!'}

print(type(my_list))
#>> > <class 'dict'>
```

Функция `isinstance()` возвращает булево значение, говорящее о том, принадлежит объект к определенному классу или нет:

```
num = 4.44
print(isinstance(num, float))

#>> > True
```

F-строки в Python

[f-string](#) - это строка с префиксом 'f' или 'F', которая содержит выражения внутри фигурных скобок {}. Выражения заменяются их значениями.

```
>>> name = "Дмитрий"
>>> age = 25
>>> print(f"Меня зовут {name} Мне {age} лет.")
>>> Меня зовут Дмитрий. Мне 25 лет.
```

f-строки также поддерживают расширенное форматирование чисел:

```
>>> from math import pi
>>> print(f"Значение числа pi: {pi:.2f}")
>>> Значение числа pi: 3.14
```

Они поддерживают базовые арифметические операции. Да, прямо в строках:

```
>>> x = 10
>>> y = 5
>>> print(f"{x} x {y} / 2 = {x * y / 2}")
>>> 10 x 5 / 2 = 25.0
```

С помощью f-строк можно форматировать дату без вызова метода strftime():

```
>>> from datetime import datetime as dt
>>> now = dt.now()
>>> print(f"Текущее время {now:%d.%m.%Y %H:%M}")
>>> Текущее время 24.02.2017 15:51
```