Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» ММФ БГУ

Тема 2. Объект как фундаментальное понятие в Python. Встроенные типы объектов

доц. Лаврова О.А., доц. Щеглова Н.Л., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

февраль, 2024

2.1 Понятие объекта. Атрибуты. Методы. Встроенные типы объектов

Объект в Python — это базовая сущность для представления данных.

К объектам в Python относятся: данные встроенных типов, файлы, функции, модули, классы, экземпляры классов и т.д.

Внутренне объекты представляют собой области выделенной памяти с достаточным пространством для хранения информации об объекте.

У объекта всегда есть атрибуты. **Атрибут** — это имя переменной или имя функции, которое связано с объектом и используется для совершения действий над этим объектом. Все атрибуты объекта составляют **пространство имен объекта**.

Доступ к атрибуту `attr` объекта `obj` производится с применением синтаксиса уточнения `obj.attr` . Доступ к атрибутам объекта является одним из самых распространенных выражений в Python

In [1]:
```
list1 = [1,2];
list1.__class__
```

Out[1]:
```
list
```

Атрибуты объектов могут быть именами переменных или именами функций. Если атрибут объекта является именем функции, то он называется **методом**. Для вызова функции следует указать ее имя, затем в круглых скобках через запятую перечислить аргументы вызова функции

In [2]:
```
list1.append(3); print(list1)
list1.__class__ # атрибут, не являющийся методом
```

```
[1, 2, 3]
```
Out[2]:
```
list
```

Просмотреть пространство имен объекта можно с помощью встроенной функции `dir` . Аргументом функции `dir` является экземпляр объекта или тип объекта

In [3]:
```
print(dir(list1))
dir(list1) == dir(list)
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__del
item__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__'
, '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd__', '__imul__', '__i
nit__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__
rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', '
reverse', 'sort']
```
Out[3]:
```
True
```

Атрибуты, которые начинаются и заканчиваются двумя знаками подчеркивания, *чаще всего* являются **методами, реализующими операции** над объектами (*dunder method*, dunder comes from 'double underscore' *или magic methods*)

In [4]:
```
print([attr for attr in dir(list1) if attr.startswith('_')])
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__del
item__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__'
, '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd__', '__imul__', '__i
nit__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__
rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__']
```

Операция (operator) — это конструкция языка, аналогичная по записи математическим операциям. Для обозначения операции часто используют не алфавитно-цифровые, а специальные символы.

In [5]:
```
a = 1
a + 2, a.__add__(2)
```

```
(3, 3)
```

Встроенные типы объектов

Встроенные типы объектов делятся на **числа** и **коллекции**.

Типы объектов, представляющих числа: **целые числа (`int`)**, **числа с плавающей точкой (`float`)**, **комплексные числа (`complex`)**. **Булевый тип (`bool`)** наследуется от типа `int` , поэтому булевый тип относится к типам числовых объектов.

Коллекции — это наборы объектов, которые делятся на три вида:

Последовательности — это *позиционно упорядоченные* коллекции. Встроенные типы, относящиеся к последовательностям: **строки (`str`)**, **списки (`list`)**, **кортежи (`tuple`)**.

Отображения — это неупорядоченные коллекции объектов, *хранящихся по ключам*. Встроенные типы, относящиеся к отображениям: **словарь (`dict`)**.

Множества — это неупорядоченные коллекции *не повторяющихся* объектов. Встроенные типы, относящиеся к множествам: **множество (`set`)**.

Встроенная функция `type(obj)` возвращает тип объекта `obj`

In [6]:
```
type(1), type('1'), type([1])
```

Out[6]:
```
(int, str, list)
```

Встроенные типы являются частью базового языка Python и всегда доступны для использования.

Встроенные типы Python задействуют уже оптимизированные алгоритмы для работы со структурами данных, которые реализованы на C, чтобы обеспечивать высокое быстродействие.

Встроенные типы объектов не только облегчают программирование, но также являются более мощными и эффективными структурами данных, чем большинство того, что можно создать в Python самостоятельно.

Создание объектов встроенных типов

Создавать объекты можно с помощью *литералов* и с помощью *функций-конструкторов* объектов.

Литерал в Python — это форма записи данных для их представления объектом пределенного типа. Синтаксис литерала определяет тип создаваемого объекта. Приведем пример, как от различных форм записи числа 1, зависит тип создаваемого объекта:

In [7]:
```
print([type(obj) for obj in [1, 1., 1j, '1', [1], (1,), {1}, {1:1}]])
```

```
[<class 'int'>, <class 'float'>, <class 'complex'>, <class 'str'>, <class 'list'>,
<class 'tuple'>, <class 'set'>, <class 'dict'>]
```

Имя **функции-конструктора** совпадает с названием типа создаваемого объекта

In [8]:
```
int(), float(), complex(), str(), list(), tuple(), dict(), set([1])
```

Out[8]:
```
(0, 0.0, 0j, '', [], (), {}, {1})
```

Выражение (expression) в Python — это конструкция языка в виде комбинации литералов, переменных, операций и вызовов функций, которая вычисляется и возвращает значение в виде объекта.

2.2 Встроенные типы объектов, представляющих числа

Примеры создания целых чисел (`int`) в различных системах счисления:

In [9]:
```
1_000_111_111_000, 0b11, 0o11, 0x11, int()
```

Out[9]:
```
(1000111111000, 3, 9, 17, 0)
```

Примеры создания чисел с плавающей точкой (`float`):

In [10]:
```
3.1415, 3e-10, 1E-1, float(2)
```

Out[10]:
```
(3.1415, 3e-10, 0.1, 2.0)
```

С помощью символов `e` или `E` задается экспоненциальная форма записи чисел с плавающей точкой.

Тип `float` является аналогом 64-битного *double* в C/C++.

Числа с плавающей точкой (`float`) не способны представлять определенные значения точно из-за ограниченного количества битов. Это фундаментальная проблема в численном программировании, не уникальная для языка Python

In [11]:
```
1.1 + 2.2 == 3.3, 1.1 + 2.2 - 3.3
```

Out[11]:
```
(False, 4.440892098500626e-16)
```

Примеры создания комплексных чисел (`complex`):

In [12]:
```
3+4j, 1J, complex(1,1)
```

Out[12]:
```
((3+4j), 1j, (1+1j))
```

Мнимая часть комлексных чисел заканчивается символом `j` или `J` .

Действительная и мнимая части комплексного числа доступны через атрибуты `real` и `imag`, соответственно

In [13]:
```
c = 1J; c.real, c.imag
```

Out[13]:
```
(0.0, 1.0)
```

Некоторые арифметические операции

Операция (operator) — это конструкция языка, аналогичная по записи математическим операциям. Для обозначения операции часто используют не алфавитно-цифровые, а специальные символы.

Операция возведения в степень (`**`) реализована для всех числовых типов

In [14]:
```
3**456, 15.**2., (1 + 15.J)**2
```

Out[14]:
```
(3692258952335548868486253430960682864635485853933963934010870460686027831695461222
3011926714496786380566924041472156813147058283837186067759425594341390772627248925 6
3647389406590090689494677802027036224351161724135952 1,
 225.0,
 (-224+30j))
```

Встроенная функция `pow(x,y)` также позволяет возводить число `x` в степень `y`

In [15]:
```
pow(3,456), pow(15.,2.), pow(1 + 15.J,2)
```

Out[15]:
```
(3692258952335548868486253430960682864635485853933963934010870460686027831695461222
3011926714496786380566924041472156813147058283837186067759425594341390772627248925 6
3647389406590090689494677802027036224351161724135952 1,
 225.0,
 (-224+30j))
```

Операция целочисленного деления с округлением в меньшую сторону (`//`) реализована для типов `int` и `float`

In [16]:
```
15//2, 15.2//2
```

Out[16]:
```
(7, 7.0)
```

Операция вычисления остатка от деления (`%`) реализована для типов `int` и `float`

In [17]:
```
15%2, 15.2%2
```

Out[17]:
```
(1, 1.1999999999999993)
```

Если для операций сложения (+), вычитания (-) или умножения (*) в качестве операндов указаны числовые объекты РАЗНЫХ типов, то процесс вычисления следующий. Сначала операнды преобразуются к типу, который представляет наиболее широкое математическое множество, а затем выполняется операция над операндами одинакового типа

In [18]:
```
(23 + 1.2 + 1J)**2
```

Out[18]:
```
(584.64+48.4j)
```

Некоторые встроенные функции для работы с числами

Функции преобразования целого числа (`int`) из десятичной системы счисления в двоичную `bin` , в восьмеричную `oct` , в шестнадцатеричную `hex` системы счисления:

In [19]:
```
bin(3), oct(8), hex(15)
```

Out[19]:
```
('0b11', '0o10', '0xf')
```

Функции преобразования `bin` , `oct` , `hex` возвращают строковый объект

In [20]:
```
?abs
```

```
Signature: abs(x, /)
Docstring: Return the absolute value of the argument.
Type:      builtin_function_or_method
```

In [21]:
```
abs(-1), abs(complex(1,1))
```

Out[21]:
```
(1, 1.4142135623730951)
```

In [22]:
```
?round
```

```
Signature: round(number, ndigits=None)
Docstring:
Round a number to a given precision in decimal digits.

The return value is an integer if ndigits is omitted or None.  Otherwise
the return value has the same type as the number.  ndigits may be negative.
Type:      builtin_function_or_method
```

In [23]:
```
round(10.51), round(10.555555, 4)
```

Out[23]:
```
(11, 10.5556)
```

Модули и расширения для работы с числами

Для работы с типом десятичных чисел (`decimal`) необходимо импортировать модуль `decimal` . **Десятичные числа (`decimal`)** являются числами с плавающей точкой с *фиксированным количеством значащих цифр*. Вычисления с числами типа `decimal` могут иметь лучшую точность, чем вычисления с числами типа `float` .

In [24]:
```
import decimal as dec
print(0.1+0.2, dec.Decimal('0.1')+dec.Decimal('0.2'))
```

```
0.30000000000000004 0.3
```

Для работы с типом рациональных чисел (`fraction`) необходимо импортировать модуль `fractions` . **Рациональные числа (`fraction`)** явно хранят числитель и знаменатель, что позволяет избежать ряда неточностей и ограничений в вычислениях с числами с плавающей точкой (`float` , `decimal`).

Невстроенные типы числовых объектов `decimal` и `fraction` предлагают способы получения более точных результатов вычислений (например, необходимые в финансовых приложениях), хотя ценой снижения скорости вычислений и более многословного кода.

Более сложные инструменты для работы с числами содержатся в модуле `math` .

Модуль `cmath` предназначен для работы с комплексными числами.

Модуль `random` выполняет генерацию случайных чисел и случайный выбор.

Расширение `numpy` предоставляет инструменты для эффективного численного программирования на основе специального типа данных массив `ndarray` .

2.3 Встроенные типы последовательностей: str, list, tuple

Последовательность — это *позиционно упорядоченная* коллекция объектов.

Строка (`str`) — это последовательность символов для хранения и представления текста, которую *нельзя изменять* после ее создания (read-only).

Встроенный тип объектов-символов не определен в Python (отсутствует аналог, например, типа `char` в C, C++).

Список (`list`) -- это последовательность объектов произвольных типов, которую *можно изменять* после ее создания.

Кортежи (`tuple`) -- это последовательность объектов произвольных типов, которую *нельзя изменять* после ее создания (read-only).

Создание последовательностей

Литерал, который создает объект типа список имеет вид `[elem1, elem2,...,elemN]`, где элементами списка могут быть объекты Python произвольного типа. Создание списка также возможно с помощью функции-конструктора `list()`

In [25]:
```
[1, [1+2j, [4e1]], 4.5], list(range(3))
```

Out[25]:
```
([1, [(1+2j), [40.0]], 4.5], [0, 1, 2])
```

Квадратные скобки `[...]` используются также для отображения объектов типа список.

Список поддерживает произвольное вложение объектов.

Литерал, который создает объект типа кортеж имеет вид `(elem1, elem2,...,elemN)`, где элементами кортежа могут быть объекты Python произвольного типа. Создание кортежа также возможно с помощью фукнции-конструктора `tuple()`

In [26]:
```
(),(1, ([2], (6,7)), 4.5), tuple([1,20,3])
```

Out[26]:
```
((), (1, ([2], (6, 7)), 4.5), (1, 20, 3))
```

Круглые скобки используются также для отображения объектов типа кортеж.

Кортеж поддерживает произвольное вложение объектов.

При создании одноэлементных кортежей необходимо обязательно указывать запятую после элемента

In [27]:
```
(1,)
```

Out[27]:
```
(1,)
```

Круглые скобки, окружающие элементы кортежа, часто можно опускать; запятые — это то, что фактически определяет кортеж

In [28]:
```
1, 2, 3
```

Out[28]:
```
(1, 2, 3)
```

Строковые объекты имеют много способов их записи в коде. Строки в Python разрешено заключать в 'одинарные' или "двойные" кавычки. Различные кавычки обозначают то же самое, но позволяют встраивать в строковый объект кавычки другого вида. Создание строки также возможно с помощью фукнции-конструктора `str()`

In [29]:
```
'spa"m', "Bob's", str(1)
```

Out[29]:
```
('spa"m', "Bob's", '1')
```

Для читабельности кода при создании строк предпочтение отдается одинарным кавычкам.

Пустая строка задается парой кавычек (одинарных или двойных), между которыми ничего нет или вызовом функции-конструктора `str` без аргументов:

In [30]:
```
'', str()
```

Out[30]:
```
('', '')
```

Многострочные строковые литералы задаются в Python с помощью утроенных кавычек

In [31]:
```
"""
1
2
"""
```

Out[31]:
```
'\n1\n2\n'
```

Текст между утроенными кавычками собирается в единственную строку с дополнительными символами новой строки `\n` в местах, где в коде присутствуют разрывы строк. Строки в утроенных кавычках часто применяются для задания **строк документации** для объектов Python.

В строках может использоваться управляющий символ обратной косой черты `\` для введения специальных символов, управляющих отображением содержимого строки и которые не могут быть легко набраны на клавиатуре. Символ \ и один или более следующих за ним символов в строковом литерале в результирующем строковом объекте заменяются символом, который имеет значение, указанное управляющей последовательностью.

Например, два символа `\n` обозначают переход на новую строку, `\t` заменяется символом табуляции

In [32]:
```
print('1 \t 2 \n 3')
```

```
1 	 2 
 3
```

Для явного представления символа обратной косой черты в строке его необходимо удваивать `\\`

In [33]:
```
print('\\')
```

```
\
```

Если Python не распознает символ после \ в качестве допустимого управляющего кода, то он просто оставляет обратную косую черту в результирующей строке

In [34]:
```
print('\c \xc6')
```

```
\c Æ
```

Для создания строк, отменяющих действие управляющих символов обратной косой черты, используется символ `r` или `R` перед открывающей кавычкой строки. В результате Python сохраняет управляющие символы так, как они набраны

In [35]:
```
print('1\n2\t\t3\n', r'1\n2\t\t3\n')
```

```
1
2		3
 1\n2\t\t3\n
```

Для создания байтовых строк используется символ `b` или `B` перед открывающей кавычкой строки

Байтовая строка (bytes) — это последовательность символов для хранения и представления *байтовой* информации, используемой в файлах медиа-данных и при передаче через сеть.

Для создания строк Unicode используется символ `u` или `U` перед открывающей кавычкой строки

In [36]:
```
b'a\x01c', u'sp\xc4m'
```

Out[36]:
```
(b'a\x01c', 'spÄm')
```

Строки в Python имеют фиксированные размеры и не могут быть изменены после создания!

Операция индексации `[]`

Последовательности поддерживают операцию индексации, или доступа к элементам по смещению. Индексация элементов последовательностей начинается с нуля. Для доступа к элементу последовательности индекс элемента указывается в квадратных скобках `X[i]`, что означает предоставить из последовательности X содержимое для позиции `i`.

In [37]:
```
list1 = [2,3.3,'str']; list1[1]
```

Out[37]:
```
3.3
```

In [38]:
```
str1 = 'String'; str1[0]
```

Out[38]:
```
'S'
```

In [39]:
```
tuple1 = (1, str1, list1); tuple1[1]
```

Out[39]:
```
'String'
```

Обратите внимание, что результатом индексации элементов строки всегда является строка

In [40]:
```
str1[0][0], str1[0][0][0]
```

Out[40]:
```
('S', 'S')
```

Возможна индексация элементов последовательности отрицательными значениями. В этом случае доступ к элементам последовательности осуществляется с конца

In [41]:
```
str1, str1[-1]
```

Out[41]:
```
('String', 'g')
```

In [42]:
```
list1, list1[-2]
```

Out[42]:
```
([2, 3.3, 'str'], 3.3)
```

In [43]:
```
tuple1, tuple1[-3]
```

Out[43]:
```
((1, 'String', [2, 3.3, 'str']), 1)
```

Индексация за пределами последовательности приводит к ошибке

In [44]:
```
str1
```

Out[44]:
```
'String'
```

In [45]:
```
str1[10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[45], line 1
----> 1 str1[10]

IndexError: string index out of range
```

В дополнение к позиционной индексации элементов последовательности в Python поддерживается обобщенная форма индексации — нарезание. **Нарезание** представляет собой способ извлечения ЧАСТИ последовательности или СРЕЗА за один раз.

Срез `X[i:j]` означает "предоставить из последовательности `X` все содержимое, начиная с позиции `i` и заканчивая позицией `j`, *не включая содержимое для позиции `j`*".

In [46]:
```
str1, str1[1:3], str1[2:-1]
```

Out[46]:
```
('String', 'tr', 'rin')
```

Срез возвращает новый объект последовательности.

Если левая позиция среза не указана `X[:j]`, тогда по умолчанию левая позиция равна 0. Если правая позиция среза не указана `X[i:]`, тогда по умолчанию правая позиция равна количеству элементов нарезаемой последовательности.

In [47]:
```
str1, str1[1:], str1[:3], str1[:]
```

Out[47]:
```
('String', 'tring', 'Str', 'String')
```

Срез за границами последовательности не приводит к ошибке, так как позиции, выходящие за границы последовательности, корректируются

In [48]:
```
str1, str1[-100:100], str1[10:]
```

Out[48]:
```
('String', 'String', '')
```

Результатом индексации элементов последовательности `X[i:j]` в обратном порядке, когда нижняя граница больше верхней `i>j`, является пустая последовательность:

In [49]:
```
str1[2:1], list1[3:1], tuple[4:1]
```

Out[49]:
```
('', [], tuple[slice(4, 1, None)])
```

В Python поддерживается также **расширенное нарезание** `X[i:j:k]` с шагом `k`, значение которого по умолчанию равно `1`:

In [50]:
```
str1, str1[::2], str1[1::3], str1[::-1]
```

Out[50]:
```
('String', 'Srn', 'tn', 'gnirtS')
```

Операция индексации `[...]` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция индексации `[]` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`).

Операция `+` для последовательностей реализует их конкатенацию

In [51]:
```
str1, str1 + str1
```

Out[51]:
```
('String', 'StringString')
```

In [52]:
```
list1, list1 + list1
```

Out[52]:
```
([2, 3.3, 'str'], [2, 3.3, 'str', 2, 3.3, 'str'])
```

In [53]:
```
tuple1, tuple1 + tuple1
```

Out[53]:
```
((1, 'String', [2, 3.3, 'str']),
 (1, 'String', [2, 3.3, 'str'], 1, 'String', [2, 3.3, 'str']))
```

В выражении с операцией сложения `+` для последовательностей в качестве операндов могут быть использованы только объекты одного и того же типа. Нельзя использовать последовательности различных типов в операции сложения `+` из-за **строгой типизации объектов** в Python. Автоматическое преобразование типов не производится в Python.

In [54]:
```
[1] + '1'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[54], line 1
----> 1 [1] + '1'

TypeError: can only concatenate list (not "str") to list
```

Следует отметить, что для числовых объектов при выполнении операции `+` допускается смешение типов операндов:

In [55]:
```
1+1.1
```

Out[55]:
```
2.1
```

Операция сложения `+` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция сложения `+` определена в Python, где в качестве обоих операндов могут испольоваться

- числовые объекты (`int` , `float` , `complex`) или
- строки (`str`) или
- списки (`list`) или
- кортежи (`tuple`).

Операция `*` для последовательности реализует ее повторение

В операции `*` одним из операндов является последовательность, вторым -- целое число, которое определяет число повторений.

In [56]:
```
str1, str1*4
```

Out[56]:
```
('String', 'StringStringStringString')
```

In [57]:
```
list1, list1*2
```

Out[57]:
```
([2, 3.3, 'str'], [2, 3.3, 'str', 2, 3.3, 'str'])
```

In [58]:
```
tuple1, 0*tuple1
```

Out[58]:
```
((1, 'String', [2, 3.3, 'str']), ())
```

Операция умножения `*` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция умножения `*` определена в Python, где в качестве обоих операндов могут испольоваться

- числовые объекты (`int` , `float` , `complex`) или
- строка (`str`) и целое число или
- список (`list`) и целое число или
- кортеж (`tuple`) и целое число.

Операция `len` вычисления числа элементов

In [59]:
```
str1 = 'String'; len(str1)
```

Out[59]:
```
6
```

In [60]:
```
list1 = range(10); len(list1)
```

Out[60]:
```
10
```

In [61]:
```
tuple1 = (1, str1, list1); len(tuple1)
```

Out[61]:
```
3
```

Операция `len(obj)` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `len(obj)` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`).

Операция `in` проверки принадлежности

Операция `in` (`not in`) проверки принадлежности (не принадлежности) возвращает булевый объект `True` или `False` .

Операция `elem in sequence` проверяет принадлежность элемента `elem` последовательности `sequence`

In [62]:
```
str1 = 'String'; 't' in str1
```

Out[62]:
```
True
```

In [63]:
```
list1 = range(10); 10 not in list1
```

Out[63]:
```
True
```

In [64]:
```
tuple1 = (1, str1, list1); 2 in tuple1
```

Out[64]:
```
False
```

Операция `substring in string` проверяет принадлежность подстроки `substring` строковому объекту `string`

In [65]:
```
str1 = 'String'; 'ing' in str1
```

Out[65]:
```
True
```

Операция `in` (`not in`) может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `in` (`not in`) реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)

Встроенные функции для работы со строками

Функция `ord(s)` преобразовывает односимвольную строку `s` в целочисленный код символа в кодировочной таблице

In [66]:
```
?ord
```

```
Signature: ord(c, /)
Docstring: Return the Unicode code point for a one-character string.
Type:      builtin_function_or_method
```

In [67]:
```
ord('й')
```

Out[67]:
```
1081
```

Функция `chr(code)` выполняет обратное преобразование, принимая целочисленный код символа `code` и возвращая символ, соответствующий коду

In [68]:
```
?chr
```

```
Signature: chr(i, /)
Docstring: Return a Unicode string of one character with ordinal i; 0 <= i <= 0x10f
fff.
Type:      builtin_function_or_method
```

In [69]:
```
chr(1081)
```

Out[69]:
```
'й'
```

Модуль стандартной библиотеки `re` предназначен для обработки строк на основе образцов.

2.4 Встроенный тип отображения: dict

Отображение — это неупорядоченная коллекция элементов в виде пары объектов *ключ: значение*.

Словарь (`dict`) — это единственный встроенный тип Python, который относится к отображениям.

Создание словаря

Литерал, который создает объект типа словарь имеет вид `{key1: value1, key2: value2,...,keyN: valueN}` , элементы словаря задаются парами объектов `ключ: значение`

In [70]:
```
{1: 1, 'second': [2], (3,): {3:3}}
```

Out[70]:
```
{1: 1, 'second': [2], (3,): {3: 3}}
```

Ключами словаря могуть быть только числа (`int` , `float` , `complex`), строки(`str`), кортежи (`tuple`). Значениями словаря могут быть объекты произвольных типов. Словарь поддерживает произвольное вложение объектов.

Каждый ключ словаря способен иметь только одно ассоциированное значение

In [71]:
```
{1:1, 1:2, 1:3}
```

Out[71]:
```
{1: 3}
```

Одинаковое значение может храниться под любым количеством ключей в словаре

In [72]:
```
{1:1, 2:1, 3:1}
```

Out[72]:
```
{1: 1, 2: 1, 3: 1}
```

Создать словарь также можно с помощью функции-конструктора `dict`

In [73]:
```
dict(one=1, two=2), dict(zip(['one', 'two'], [1, 2]))
```

Out[73]:
```
({'one': 1, 'two': 2}, {'one': 1, 'two': 2})
```

In [74]:
```
?zip
```

```
Init signature: zip(self, /, *args, **kwargs)
Docstring:
zip(*iterables, strict=False) --> Yield tuples until an input is exhausted.

   >>> list(zip('abcdefg', range(3), range(4)))
   [('a', 0, 0), ('b', 1, 1), ('c', 2, 2)]

The zip object yields n-length tuples, where n is the number of iterables
passed as positional arguments to zip().  The i-th element in every tuple
comes from the i-th iterable argument to zip().  This continues until the
shortest argument is exhausted.

If strict is true and one of the arguments is exhausted before the others,
raise a ValueError.
Type:           type
Subclasses:
```

Операция индексации [ ]

Словари не поддерживают позиционное упорядочение слева направо, как последовательности. Обращение к значению словаря/индексация словаря происходит по ключу.

Внутренне словари реализованы как хеш-таблицы, обеспечивая очень быстрое извлечение элементов.

Выражением индексации словаря по ключу являются квадратные скобки `[]`.
Операция индексации позволяет извлекать и изменять значения, связанные с ключами.

In [75]:
```
dict1 = {'one': 1}

dict1['one'] = 11 # изменение значения с использованием индексации
dict1['one'] # извлечение значения с использованием индексации
```

Out[75]:
```
11
```

Присваивание элементу словаря, проиндексированному по несуществующему ключу, приводит к созданию НОВОГО элемента словаря с указанным ключом и указанным значением:

In [76]:
```
dict1['new'] = 2
dict1
```

Out[76]:
```
{'one': 11, 'new': 2}
```

При идексации по ключу, который является кортежем, круглые скобки для обозначения кортежа можно опускать:

In [77]:
```
dict1[1,2] = 12
dict1
```

Out[77]:
```
{'one': 11, 'new': 2, (1, 2): 12}
```

Индексация по несуществующему ключу словаря приводит к ошибке:

In [78]:
```
dict1['two']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[78], line 1
----> 1 dict1['two']

KeyError: 'two'
```

Обращение к элементу словаря по несуществующему ключу с помощью словарного метода `get` не приведет к ошибке и вернет `None`:

In [79]:
```
print(dict1.get('two'))
```

```
None
```

Словарь не имеет фиксированных размеров.

Операция индексации `[...]` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция индексации `[...]` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)

Операции сложения `+` и умножения `*` не реализованы для словарей

Операция (`X | Y`) реализует объединение словарей `X` и `Y` . В случае совпадения ключей в объединенном словаре сохраняется то значение, которое является крайним правым в выражении

In [80]:
```
X = dict(one=1); Y = {'one':'updated', 'two':2}
X | Y
```

Out[80]:
```
{'one': 'updated', 'two': 2}
```

Оператор (`X |= Y`) присваивает словарю `X` результат объединения двух словарей `X` и `Y` :

In [81]:
```
X |= Y; X
```

Out[81]:
```
{'one': 'updated', 'two': 2}
```

Операция `len` вычисления числа элементов

In [82]:
```
dict1, len(dict1)
```

Out[82]:
```
({'one': 11, 'new': 2, (1, 2): 12}, 3)
```

Операция `len(obj)` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `len(obj)` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)

Операция `in` проверки принадлежности

Операция `in` (`not in`) проверки принадлежности (не принадлежности) КЛЮЧА СЛОВАРЮ возвращает булевый объект `True` или `False`.

In [83]:
```
dict1
```

Out[83]:
```
{'one': 11, 'new': 2, (1, 2): 12}
```

In [84]:
```
'one' in dict1, 'three' not in dict1
```

Out[84]:
```
(True, True)
```

In [85]:
```
dict1['one'] in dict1
```

Out[85]:
```
False
```

Операция `in` (`not in`) может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `in` (`not in`) реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)

Кодовые трюки со словарями

Словарные ключи в виде кортежей могут применяться для реализации *разреженных структур данных*, например, представляя таким образом многомерные массивы, где значения хранятся лишь в нескольких позициях

In [86]:
```
matrix = {}
matrix[1,10,100] = 5; matrix[100,100,100] = 500
matrix[(1,10,100)]
```

Out[86]:
```
5
```

В приведенном примере ключи `(i,j,k)` представляют собой индексы элемента трехмерного массива $a_{i,j,k}$.

2.5 Встроенный тип множества: set

Множество (`set`) — это неупорядоченная коллекция не повторяющихся объектов. Элемент встречается во множестве только однажды независимо от того, сколько раз он добавлялся. Множество может содержать только числа, строки и кортежи (неизменяемые объекты). Множество не имеет фиксированных размеров.

Множества подобны словарям без значений. Так как элементы множества неупорядоченные, уникальные и неизменяемые, они ведут себя очень похоже на ключи словаря.

Литерал множества, как и литерал словаря, начинается и заканчивается фигурными скобками `{elem1, elem2, ..., elemN}` . Отличия литералов множества и словаря заключаются в представлении элементов соответствующей коллекции:

In [87]:
```
{1}, {1:1}
```

Out[87]:
```
({1}, {1: 1})
```

Выбор синтаксиса с фигурными скобками для литералов множеств логичен, так как множества очень похожи на ключи в словарях без значений.

Пустое множество создать с помощью литерала нельзя

In [88]:
```
{}, type({})
```

Out[88]:
```
({}, dict)
```

Примеры создания множеств, которые демонстрируют, что повторяющиеся элементы удаляются из множества

In [89]:
```
{1,2,3,3,1}, set('setset')
```

Out[89]:
```
({1, 2, 3}, {'e', 's', 't'})
```

Операции индексации `[]` , сложения `+` и умножения `*` не реализованы для множеств

Операция `len` вычисления числа элементов

In [90]:
```
X = {1,1,2,3}; len(X)
```

Out[90]:
```
3
```

Операция `len(obj)` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `len(obj)` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)
- множеств (`set`)

Операция `in` проверки принадлежности

Операция `in` (`not in`) проверки принадлежности (не принадлежности) возвращает булевый объект `True` или `False` .

In [91]:
```
Y = {4,5}; print(X, Y)
```

```
{1, 2, 3} {4, 5}
```

In [92]:
```
2 in X, X not in Y
```

Out[92]:
```
(True, True)
```

Операция `in` (`not in`) может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `in` (`not in`) реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)
- множеств (`set`)

Операции над множествами: &, |, -, <, <=

Определим два множества:

In [93]:
```
X = set(range(3)); Y = set(range(4))
X, Y
```

Out[93]:
```
({0, 1, 2}, {0, 1, 2, 3})
```

Операция (&) реализует пересечение множеств:

In [94]:
```
print(X, Y, X & Y)
```

```
{0, 1, 2} {0, 1, 2, 3} {0, 1, 2}
```

Операция (|) реализует объединение множеств:

In [95]:
```
print(X, Y, X | Y)
```

```
{0, 1, 2} {0, 1, 2, 3} {0, 1, 2, 3}
```

Операция (-) реализует разность множеств:

In [96]:
```
print(X, Y, X - Y, Y - X)
```

```
{0, 1, 2} {0, 1, 2, 3} set() {3}
```

Операции (< и <=) являются проверкой вложенности множеств:

In [97]:
```
print(X, Y, X > Y, X < X, X <= X)
```

```
{0, 1, 2} {0, 1, 2, 3} False False True
```

Кодовые трюки с множествами

В Python существует подход для удаления повторяющихся элементов ЛЮБОЙ последовательности с помощью использования множества

In [98]:
```
list1 = list(range(5))*2
print(list1)

list1 = list(set(list1))
print(list1)
```

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

В Python существует подход на основе использования множеств для проверки равенства двух последовательностей, когда порядок следования элементов в последовательностях не важен

In [99]:
```
set('abcdef') == set('fedcba')
```

Out[99]:
```
True
```

2.6 Встроенный булевый тип

В Python определен также встроенный булевый тип объектов `bool` . Булевый тип *унаследован* от встроенного типа целых чисел `int` . Как следствие, объекты булевого типа являются числовыми объектами в Python.

In [100…
```
bool.__bases__
```

Out[100]:
```
(int,)
```

Булевый тип имеют только два объекта в Python с именами переменных `True` и `False`, ссылающимися на эти два объекта. Два объекта булевого типа создаются в Python *автоматически*, т.е. программисту не нужно использовать литерал или вызывать конструктор для создания булевых объектов.

In [101…
```
type(True), type(False)
```

Out[101]:
```
(bool, bool)
```

Значениями объектов `True` и `False` являются целые числа 1 и 0, соответственно, которые можно использовать в арифметических операциях:

In [102…
```
True + 1, True*False
```

Out[102]:
```
(2, 0)
```

При отображении булевых объектов отображаются не их значения, а имена переменных:

In [103…
```
print(True, False)
```

```
True False
```

КАЖДЫЙ объект встроенного типа в Python характеризуется своим **булевым значением** True или False:

числа, равные нулю, имеют булево значение False и значение True в противном случае;

пустые коллекции имеют булево значение False и True в противном случае;

объект `None` имеет булево значение False.

Встроенная функция `bool(obj)` возвращает булево значение объекта `obj`

In [104…
```
[bool(x) for x in [0, 10, '', 'a', [], [1], {}, {1:1}, None]]
```

Out[104]:
```
[False, True, False, True, False, True, False, True, False]
```

2.7 Изменяемые и неизменяемые встроенные типы объектов

Все типы данных в Python относятся к одной из двух категорий: изменяемые типы (mutable) и неизменяемые типы (immutable).

Объекты **изменяемых типов** МОЖНО:

- изменять/перезаписывать после создания;
- увеличивать/уменьшать в размере, если тип объектов является коллекцией.

К изменяемым встроенным типам объектов относятся коллекции разных видов: списки (`list`), словари (`dict`), множества (`set`).

In [105…
```
t = [1,2], {1:2}, {1,2}
```

Увеличение размера коллекции

In [106…
```
t[0].append(3); t[1].update({2:3, 3:4}); t[2].update({3,4})
t
```

Out[106]:
```
([1, 2, 3], {1: 2, 2: 3, 3: 4}, {1, 2, 3, 4})
```

Уменьшение размера коллекции

In [107…
```
t[0].pop(); t[1].pop(2); t[2].pop()
t
```

Out[107]:
```
([1, 2], {1: 2, 3: 4}, {2, 3, 4})
```

Остальные встроенные типы объектов относятся к неизменяемым типам: типы числовых объектов `int` , `float` , `complex` и последовательности `str` , `tuple` .

Объекты **неизменяемых типов** НЕЛЬЗЯ:

- изменять/перезаписывать после создания;
- увеличивать/уменьшать в размере, если тип объектов является коллекцией.

Объекты неизменяемых типов более эффективны по памяти, обрабатываются быстрее и дают гарантию того, что их содержимое не будет изменено случайно в коде.

Особенность данных кортежа

Неизменяемость кортежа относится только к верхнему уровню самого кортежа, но не к его содержимому

In [108…
```
t = [1,2], {1:2}, {1,2}; t[0] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[108], line 1
----> 1 t = [1,2], {1:2}, {1,2}; t[0] = 0

TypeError: 'tuple' object does not support item assignment
```

In [109…
```
t[0][0] = 0; print(t)
```

```
([0, 2], {1: 2}, {1, 2})
```

Особенность данных словаря

Ключом для элемента словаря может являться только объект неизменяемого типа: число, строка, кортеж. Значением для элемента словаря может быть объект любого типа

In [110…
```
{1: [1], '1': {1: 2}, (1,): {1,2}}
```

Out[110]:
```
{1: [1], '1': {1: 2}, (1,): {1, 2}}
```

Особенность данных множества

Элементами множества могут являться только объекты неизменяемых типов: число, строка, кортеж

In [111…
```
{1, 'str', (1,)}.add([1])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[111], line 1
----> 1 {1, 'str', (1,)}.add([1])

TypeError: unhashable type: 'list'
```

2.8 Интерфейс объекта: операции и методы

Множество операций и методов для конкретного типа объектов называется **интерфейсом объекта**. После того, как объект создан, работать с объектом можно только через его интерфейс. Это называется **строгой типизацией объектов** в Python.

Операции в коде представлены с помощью специальных символов или как встроенные функции

In [112…
```
list1 = list(range(5))
list1[-1], list1+list1, list1*0, len(list1), 5 in list1
```

Out[112]:
```
(4, [0, 1, 2, 3, 4, 0, 1, 2, 3, 4], [], 5, False)
```

Операции в Python выполняются над объектами различных типов (**полиморфизм**):

In [113…
```
'abcd'[-1], [1,2,3][-1], (1,2,3)[-1], {'a':2}['a']
```

Out[113]:
```
('d', 3, 3, 2)
```

Результат выполнения операции или метода с одинаковым именем, определенным для разных типов объектов, зависит от типа объекта, над которым выполняется операция или метод:

- результат выражения `X + Y` (вызов операции сложения) зависит от типа объектов `X` и `Y`;
- в выражении `X.method` результат выполнения метода `method` зависит от типа объекта `X`.

In [114…
```
1 + 2, '1' + '2',
```

Out[114]:
```
(3, '12')
```

Операции в Python реализуются с помощью методов, которые начинаются и заканчиваются двумя знаками подчеркивания (*dunder method*, dunder comes from 'double underscore' или *magic methods*). В языке Python определено фиксированное и неизменяемое соответствие между операцией и 'dunder' методом

In [115…
```
print([attr for attr in dir(list1) if attr.startswith('_')])
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__del
item__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__'
, '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd__', '__imul__', '__i
nit__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__
rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__']
```

In [116…
```
a, b = 1, 2
a + b, a.__add__(b)
```

Out[116]:
```
(3, 3)
```

In [117…
```
a = [7]
a[0], a.__getitem__(0)
```

Out[117]:
```
(7, 7)
```

На практике заменять вызов операции на соответствующий метод НЕ НУЖНО. Это снижает читабельность кода. При этом вызов метода может выполняться медленнее, чем вызов соответствующей операции.

Методы, которые НЕ обрамлены двумя знаками подчеркивания, являются специфичными для различных типов

In [118…
```
a = [1]

a.append(2)
print(a)

a.pop()
print(a)
```

```
[1, 2]
[1]
```

Методы для кортежей

In [119…
```
tuple_method = [attr for attr in dir(tuple) if not attr.startswith('_')]
print(tuple_method)
```

```
['count', 'index']
```

До выхода Python 2.6 и 3.0 у кортежей отсутствовали методы. Это старое соглашение Python для неизменяемых типов, которое по соображениям практичности было нарушено сначала для строк, потом для чисел и кортежей.

Метод `count` подсчета повторяющихся объектов реализован для всех объектов последовательностей (`str` , `list` , `tuple`). Это означает, что метод `count` обладает **полиморфизмом**.

In [120…
```
?tuple.count
```

```
Signature: tuple.count(self, value, /)
Docstring: Return number of occurrences of value.
Type:      method_descriptor
```

In [121…
```
'qqqQ'.count('qq'), [1,1].count(1), (0,0).count(1)
```

Out[121]:
```
(1, 2, 0)
```

Метод `index(el)` вычисления наименьшего индекса указанного элемента последовательности `el` реализован для всех объектов последовательностей (`str` , `list` , `tuple`). Это означает, что метод `index` обладает **полиморфизмом**.

In [122…
```
?tuple.index
```

```
Signature: tuple.index(self, value, start=0, stop=9223372036854775807, /)
Docstring:
Return first index of value.

Raises ValueError if the value is not present.
Type:      method_descriptor
```

In [123…
```
'sss'.index('s'), [1,1].index(1), (0,0).index(0)
```

Out[123]:
```
(0, 0, 0)
```

Методы для списков

In [124…
```
list_method = [attr for attr in dir(list) if not attr.startswith('_')]
print(list_method)
len(list_method)
```

Out[124]:
```
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
11
```

Все списковые методы (кроме `count` и `index`) ИЗМЕНЯЮТ ОБЪЕКТ и возвращают `None` .

Списковые методы добавления объектов:

- `append(el)` добавляет объект `el` в конец списка. *Это самый популярный списковый метод*
- `extend(els)` добавляет несколько объектов из коллекции `els` в конец списка
- `insert(pos, el)` добавляет элемент `el` в позицию `pos`

In [125…
```
list1 = [1,2]; list1.append(3); list1.extend((5,6,7)); list1.insert(0,'first'); lis
```

Out[125]:
```
['first', 1, 2, 3, 5, 6, 7]
```

Списковые методы добавления вполняются быстрее, чем операция сложения `+` для списков.

Списковые методы удаления объектов:

- `pop(pos)` удаляет по по позиции `pos` ; при вызове `pop()` удаляется последний элемент
- `remove(el)` удвляет по значению `el`

In [126…
```
list1 = 2*[1] +5*[2]; print(list1)
list1.pop(0); list1.pop(); print(list1)
list1.remove(1); list1
```

Out[126]:
```
[1, 1, 2, 2, 2, 2, 2]
[1, 2, 2, 2, 2]
[2, 2, 2, 2]
```

Списковый метод `reverse()` переставляет элементы списка в обратном порядке

In [127…
```
?list.reverse
```

```
Signature: list.reverse(self, /)
Docstring: Reverse *IN PLACE*.
Type:      method_descriptor
```

In [128…
```
list1 = list(range(4))
list1.reverse(); list1
```

Out[128]:
```
[3, 2, 1, 0]
```

Списковый метод `sort()` ИЗМЕНЯЕТ список, сортируя его по возрастанию

In [129…
```
list1 = list("Python"); list1.sort(); list1
```

Out[129]:
```
['P', 'h', 'n', 'o', 't', 'y']
```

Встроенная функция `sorted(lst)` создает НОВЫЙ список, сортируя список `lst` по возрастанию

In [130…
```
list1 = [2,1]; print(list1, sorted(list1))
```

```
[2, 1] [1, 2]
```

In [131…
```
?list.sort
```

```
Signature: list.sort(self, /, *, key=None, reverse=False)
Docstring:
Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them,
ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.
Type:      method_descriptor
```

Методы для словарей

In [132…
```
dict_method = [attr for attr in dir(dict) if not attr.startswith('_')]
print(dict_method)
len(dict_method)
```

Out[132]:
```
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault
', 'update', 'values']
11
```

Словарный метод `keys` возвращает *итерируемый объект* с ключами элементов словаря.

Словарный метод `values` возвращает *итерируемый объект* со значениями элементов словаря.

Словарный метод `items` возвращает *итерируемый объект* с кортежами пар (ключ, значение) для элементов словаря.

In [133…
```
dict1 = dict(one=1, two=2)
dict1.keys(), dict1.values(), dict1.items()
```

Out[133]:
```
(dict_keys(['one', 'two']),
 dict_values([1, 2]),
 dict_items([('one', 1), ('two', 2)]))
```

Словарный метод `get(key)` возвращает значение, ассоциированное с ключом `key`, если ключ существует, или `None`, если ключа `key` в словаре нет. При этом индексация словаря по несуществующему ключу является ошибкой.

In [134…
```
print(dict1.get('one'), dict1.get('three'))
dict1['three']
```

```
1 None
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[134], line 2
      1 print(dict1.get('one'), dict1.get('three'))
----> 2 dict1['three']

KeyError: 'three'
```

При условии, что значения, ассоциированные со всеми ключами, первоначально одинаковы, словарь можно создать с помощью метода словаря `fromkeys`, передавая последовательность ключей и одно значение, которое будет ассоциировано со всеми ключами из последовательности

In [135…
```
?dict.fromkeys
```

```
Signature: dict.fromkeys(iterable, value=None, /)
Docstring: Create a new dictionary with keys from iterable and values set to value.
Type:      builtin_function_or_method
```

In [136…
```
dict1={}
dict1.fromkeys('keys')
```

Out[136]:
```
{'k': None, 'e': None, 'y': None, 's': None}
```

Словарный метод `update` объединяет ключи и значения исходного словаря с ключами и значениями другого словаря, обновляя значение ключа исходного словаря при совпадении ключей

In [137…
```
dict2 = dict(one=11, three=3)
print(dict1, dict2)
dict1.update(dict2)
dict1
```

Out[137]:
```
{} {'one': 11, 'three': 3}
{'one': 11, 'three': 3}
```

Методы для множеств

In [138…
```
set_method = [attr for attr in dir(set) if not attr.startswith('_')]
print(set_method)
len(set_method)
```

Out[138]:
```
['add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersectio
n', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove',
'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
17
```

Метод `add` добавляет элемент в множество

In [139…
```
?set.add
```

```
Docstring:
Add an element to a set.

This has no effect if the element is already present.
Type:      method_descriptor
```

In [140…
```
s = {1}; s.add(2); s.add(1); s
```

Out[140]:
```
{1, 2}
```

Метод `update` добавляет несколько элемент в множество

In [141…
```
?set.update
```

```
Docstring: Update a set with the union of itself and others.
Type:      method_descriptor
```

In [142…
```
s = {1}; s.update([True, False, '1']); s
```

Out[142]:
```
{1, '1', False}
```

Методы для строк

In [143…
```
str_method = [attr for attr in dir(str) if not attr.startswith('_')]
print(str_method)
len(str_method)
```

```
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', '
find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal
', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'is
title', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'r
emoveprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'titl
e', 'translate', 'upper', 'zfill']
```

Out[143]: 47

Строковые методы никогда НЕ ИЗМЕНЯЮТ объект. Строковые методы всегда возвращают НОВЫЙ строковый объект!

Строковый метод `sep.join(obj)` объединяет строки из *итерируемого объекта* `obj` с разделителем `sep` между элементами итерируемого объекта

In [144…
```
s = '+'.join('string'); s
```

Out[144]:
```
's+t+r+i+n+g'
```

Строковый метод `split(sep)` разбивает строку на СПИСОК подстрок по разделителю `sep`

In [145…
```
s.split('+')
```

Out[145]:
```
['s', 't', 'r', 'i', 'n', 'g']
```

Строковый метод `split()` без аргументов разбивает строку на СПИСОК подстрок, удаляя все пробельные символы

In [146…
```
'a   b\nc\td\n'.split()
```

Out[146]:
```
['a', 'b', 'c', 'd']
```

Строковый метод `replace(sub1, sub2)` заменяет одну подстроку `sub1` на другую подстроку `sub2`

In [147…
```
s, s.replace('+','-')
```

Out[147]:
```
('s+t+r+i+n+g', 's-t-r-i-n-g')
```

Строковый метод `replace(sub1, '')` УДАЛЯЕТ подстроку `sub1` из исходной строки

In [148…
```
s, s.replace('+','')
```

Out[148]:
```
('s+t+r+i+n+g', 'string')
```

Строковые методы `upper` , `lower` преобразовывают буквенные символы строки в прописные или строчные буквы; небуквенные символы остаются без изменений

Строковые методы `strip` , `lstrip` , `rstrip` удаляют пробельные символы в начале и/или в конце строки

In [149…
```
'    f    '.strip(), '    f    '.lstrip(), '    f    '.rstrip()
```

Out[149]:
```
('f', 'f    ', '    f')
```

Строковые методы `startswith` , `endswith` проверяют подстроки в начале и в конце строки, соответственно

In [150…
```
s, s.startswith('s-'), '2023_KM2_Lb1.ipynb'.endswith(('py','ipynb'))
```

Out[150]:
```
('s+t+r+i+n+g', False, True)
```

В Python определено 12 строковых методов для проверки содержимого строки, например, `isalpha` , `isdigit`

In [151…
```
s.isalpha(), s.isdigit(), 'acvf'.isalpha(), '1278'.isdigit(), 'ac12'.isalnum()
```

Out[151]:
```
(False, False, True, True, True)
```

Строковый метод `find` ищет подстроку в строке и возвращает минимальный индекс начала подстроки в строке. Строковый метод `find` совпадает по функциональности со строковым методом `index`

In [152…
```
print(s)
s.find('+'), s.index('+')
```

Out[152]:
```
s+t+r+i+n+g
(1, 1)
```

Отличаются результаты выполнения методов `find` и `index` тогда, когда подстрока не содержится в строке

In [153…
```
print(s, s.find('-'))
s.index('-')
```

```
s+t+r+i+n+g -1
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[153], line 2
      1 print(s, s.find('-'))
----> 2 s.index('-')

ValueError: substring not found
```

Строковые операции выполняются слева направо. При этом в процессе выполнения выражения создаются промежуточные объекты

In [154…
```
 ('a'+' bs'*4).upper().split(' ')
```

Out[154]:
```
['A', 'BS', 'BS', 'BS', 'BS']
```

2.9 Переменная

Переменная создается при первом присваивании ей значения. Основной способ создания переменных – это использование оператора присваивания `=` : `<переменная> = <значение переменной>` . При этом значением переменной является объект произвольного типа.

Чтобы переменную можно было использовать, ей обязательно должно быть присвоено значение. Использование переменной до того, как ей было присвоено значение, всегда приведет к ошибке.

In [155…
```
a = 1 # создание переменной
a, d
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[155], line 2
      1 a = 1 # создание переменной
----> 2 a, d

NameError: name 'd' is not defined
```

Понятие типа не связано с переменной, а связано со значением переменной. В процессе выполнения кода значение переменной можно изменять за счет повторных присваиваний одной и той же переменной объектов разных типов. Такая возможность Python называется **динамической типизации** переменных.

In [156…
```
a = 's'; print(a, type(a))
a = [1]; print(a, type(a))
```

```
s <class 'str'>
[1] <class 'list'>
```

Переменные ссылаются на объекты

Переменная (variable, name) в Python является ссылкой на объект. Внутренне ссылка записывается в системной таблице в виде соответствия между именем переменной и адресом ячейки памяти, где располагается объект. Переменная не хранит объект. Переменная в каждый момент выполнения кода ссылается на один определенный объект.

Когда переменная встречается в коде, она ЗАМЕНЯЕТСЯ объектом, на который она в текущий момент ссылается:

In [157…
```
a = 2; b = c = 1
a + b + c*2
```

Out[157]:
```
5
```

Переменная используется для доступа к объекту или изменения объекта, на который переменная ссылается.

Переменные всегда ссылаются на объекты и никогда не ссылаются на другие переменные. В Python нет способа связывания двух переменных

In [158…
```
a = 1
b = a   # a и b ссылаются на один и тот же объект
c = [1] # c ссылается на другой объект
b.append(2)
a, b
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[158], line 4
      2 b = a   # a и b ссылаются на один и тот же объект
      3 c = [1] # c ссылается на другой объект
----> 4 b.append(2)
      5 a, b

AttributeError: 'int' object has no attribute 'append'
```

Адрес области памяти объекта является **уникальным целочисленным идентификатором** объекта. Уникальный идентификатор объекта `obj` может быть получен с помощью функции `id(obj)`

In [159…
```
id(1), id([1]), id({1})
```

Out[159]:
```
(140711162778408, 1683659704832, 1683646938496)
```

In [160…
```
id(a), id(b), id(c)
```

Out[160]:
```
(140711162778408, 140711162778408, 1683660398848)
```

Если переменная ссылается на список, кортеж или словарь, то индексированная переменная являтся ссылкой на соответствующий элемент списка, кортежа или словаря. Списки, кортежи, словари, множества хранят не объекты, а ссылки на объекты, что позволяет этим коллекциям содержать объекты любых типов.

In [161…
```
a = [1, '1', (1,)]
a, a[1] # переменные
```

Out[161]:
```
([1, '1', (1,)], '1')
```

Если объект коллекции содержит ссылку на самого себя, то он называется **циклическим объектом**. Когда обнаруживается цикл в объекте, то он выводится специальном образом без возникновения зацикливания

In [162…
```
a = [1,2]; a.append(a)
a
```

Out[162]:
```
[1, 2, [...]]
```

Если несколько переменных ссылаются на один и тот же объект, то этот объект называется **разделяемым объектом**, а ссылка на один и тот же объект называется **разделяемой ссылкой**

In [163…
```
a = b = c = d = [5]    # [5] -- разделяемый объект
id(a), id(d)           # разделяемая ссылка
```

Out[163]:
```
(1683660399232, 1683660399232)
```

Сравнение переменных

Операции сравнения `<переменная1> is <переменная2>` или `<переменная1> is not <переменная2>` сравнивает уникальные идентификаторы объектов, на которые ссылаются переменные, и возвращает значение `True` в том случае, когда переменные ссылаются на один и тот же объект.

In [164…
```
print(a,b,c)
print(id(a), id(b), id(c))
a is b, a is not c
```

Out[164]:
```
[5] [5] [5]
1683660399232 1683660399232 1683660399232
(True, False)
```

Операции сравнения `<переменная1> == <переменная2>` или `<переменная1> != <переменная2>` сравнивает значения объектов, на которые ссылаются переменные, и возвращает значение `True` в том случае, когда переменные ссылаются на объекты с одинаковыми значениями.

In [165…
```
print(a, b, c)
a == b, a != c
```

Out[165]:
```
[5] [5] [5]
(True, False)
```

Операция сравнения `is` применяется в программах реже, чем операция сравнения `==` .

В Python поддерживается особый **объект-заполнитель** `None` , часто используемый для инициализации переменных

In [166…
```
x = None
y = [None]*5
x, y
```

Out[166]:
```
(None, [None, None, None, None, None])
```

Особенностью объекта `None` является то, что он создается только один раз во время выполнения кода

In [167…
```
x is y[2]
```

Out[167]:
```
True
```

Объект `None` имеет булево значение `False`

In [168…
```
bool(None)
```

Out[168]:
```
False
```

Динамическая типизация переменных

Понятие типа в Python связано с объектом, на который ссылается переменная, а не с переменной. Типы объектов отслеживаются *автоматически* во время выполнения кода. Переменная не располагает какой-либо информацией о типе.

Каждый объект содержит атрибут `__class__` , определяющий тип объекта

In [169…
```
a = [5]
a.__class__
```

Out[169]:
```
list
```

Каждый объект содержит *счетчик ссылок*, который отслеживает количество ссылок, в текущий момент указывающих на объект. Когда *счетчик ссылок* уменьшается до нуля, область памяти объекта *автоматически* попадает в пул свободного пространства, чтобы быть задействованной будущим объектом.

In [170…
```
a = b = c = d = [5]
import sys
sys.getrefcount(a)
```

Out[170]:
```
5
```

In [171…
```
?sys.getrefcount
```

```
Signature: sys.getrefcount(object, /)
Docstring:
Return the reference count of object.

The count returned is generally one higher than you might expect,
because it includes the (temporary) reference as an argument to
getrefcount().
Type:      builtin_function_or_method
```

Когда утрачивается последняя ссылка на объект, например, за счет присваивания единственной переменной, ссылающейся на объект, чего-нибудь другого, то вся область памяти, занимаемая исходным объектов, очищается *автоматически* (сборка мусора). **Автоматическое управление памятью** является спецификой языка Python.

Однако, небольшие целые числа и строки *кэшируются* и используются повторно при инициализации переменных. Такой подход является одним из способов оптимизации скорости выполнения кода на Python

In [172…
```
a = 10; print(id(a))
a = 1; b = 10; print(id(b))
```

```
140711162778696
140711162778696
```

Python является **динамически типизированным** языком программирования. В процессе выполнения кода одна и та же переменная может ссылаться на объекты различных типов в разные моменты времени. C++ и Java являются статически типизированными языками программирования.

Копирование объектов

Для того, чтобы различные переменные не ссылались на один и тот же объект, нужно создавать копии объектов. Существует три способа создания копий объектов.

Способ 1 (универсальный). Копирование объектов возможно с помощью вызова соответствующих функций-конструкторов.

Создаем копию исходного объекта с помощью функции конструкторв; убеждаемся в том, что уникальные идентификаторы исходного и построенного объектов не совпадают

In [173…
```
obj = [1, 2, 3]; print(obj, id(obj))
obj_copy = list(obj); obj_copy = obj.__class__(obj) # два способа
print(obj_copy, id(obj_copy))
```

```
[1, 2, 3] 1683659952384
[1, 2, 3] 1683659910272
```

Создадим копии для нескольких объектов различных типов:

In [174…
```
objs1 = ('String', [1, 2, 3], (4,5,6),
         {'a':1, 'b':2, 'c':3}, {1,4,'s',(5,)})

objs1_copy = tuple(map(lambda x: x.__class__(x), objs1))
objs1_copy
```

Out[174]:
```
('String', [1, 2, 3], (4, 5, 6), {'a': 1, 'b': 2, 'c': 3}, {(5,), 1, 4, 's'})
```

Способ 2 (для последовательностей). Копирование последовательностей (`str` , `list` , `tuple`) возможно посредством операции среза по всем элементам `[:]` .

In [175…
```
objs2 = 'String', [1, 2, 3], (4,5,6)

objs2_copy = tuple(map(lambda x: x[:], objs2))
objs2_copy
```

Out[175]:
```
('String', [1, 2, 3], (4, 5, 6))
```

Способ 3 (для изменяемых объектов). Копирование изменяемых объектов (списков (`list`), словарей (`dict`) и множеств (`set`)) возможно с помощью метода `copy`

In [176…
```
objs3 = ([1, 2, 3], {'a':1, 'b':2, 'c':3}, {1,4,'s',(5,)})

objs3_copy = tuple(map(lambda x: x.copy(), objs3))
objs3_copy
```

Out[176]:
```
([1, 2, 3], {'a': 1, 'b': 2, 'c': 3}, {(5,), 1, 4, 's'})
```

Правила именования переменных

1. Имена переменных могут начинаться с подчеркивания `_` или буквы, за которой может следовать любое количество букв, цифр или символов подчеркиваний.

1. Регистр символов имеет значение. Например, имена модулей рекомендовано начинать с буквы нижнего регистра, а имена классов -- с буквы верхнего регистра.

1. Зарезервированные имена, которые перечислены в модуле `builtins` , см. также в справочной системе https://docs.python.org/3.11/reference/lexical_analysis.html#keywords, НЕЛЬЗЯ использовать в качестве имен переменных. При переопределении зарезервированного имени предупреждение выдаваться не будет

In [177…
```
print(abs(-1))
abs = 5
print(abs(-1))
```

```
1
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[177], line 3
      1 print(abs(-1))
      2 abs = 5
----> 3 print(abs(-1))

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

1. Имена переменных с двумя подчеркиваниями `__` в начале и конце имени (dunder attribute) обычно имеют особый смысл для интерпретатора Python, поэтому необходимо избегать использования такого именования для собственных переменных.

2.10 Выражение спискового включения

Списковое включение (**list comprehension**) представляет собой способ построения НОВОГО списка за счет выполнения *выражения* на каждом элементе последовательности (итерируемого объекта), по одному за раз, слева направо.

Выражение спискового включения записывается в квадратных скобках и содержит конструкцию цикла `for ... in ...` .

Списковое включение имеет следующий синтаксический шаблон:

```
[expr(elem) for elem in sequence]
```

In [178…
```
[x+'0' # выражение
 for x in 'string'] # конструкция цикла с проходом по последовательности
```

Out[178]:
```
['s0', 't0', 'r0', 'i0', 'n0', 'g0']
```

In [179…
```
[x**2+x-3 # выражение
 for x in range(5)] # конструкция цикла с проходом по итерируемому объекту
```

Out[179]:
```
[-3, -1, 3, 9, 17]
```

Длина результарующего списка совпадает с длиной итерируемого объекта.

Целесообразно применять списковое включение тогда, когда необходимо *выполнить выражение на каждом элементе* итерируемого объекта и *получить список в результате*.

```
[expr(elem) for elem in sequence]
```

Если для каждого элемента итерируемого объекта необходимо выполнять более сложные действия (присваивания, вывод на экран и др.), то нужно использовать циклы.

Не используйте функцию `print` в выражении спискового включения, так как `print` возвращает `None`

In [180…
```
[x**2 for x in range(5)]
```

Out[180]:
```
[0, 1, 4, 9, 16]
```

In [181…
```
[print(x**2) for x in range(5)]
```

```
0
1
4
9
16
```

Out[181]: [None, None, None, None, None]

Синтаксис списковых включений позволяет дополнительно задавать ограничения на элементы из последовательности (итерируемого объекта) с использованием конструкции выбора `if` , на которых будет выполняться выражение.

Списковое включение с условием имеет следующий синтаксический шаблон:

```
[expr(elem) for elem in sequence if condition(elem)]
```

In [182…

```
[x+'0'
 for x in 'string'
 if x!='s']
```

Out[182]: ['t0', 'r0', 'i0', 'n0', 'g0']

Списковое включение с условием имеет следующий синтаксический шаблон:

```
[expr(elem) for elem in sequence if condition(elem)]
```

Списковое включение может содержать несколько конструкций выбора `if` , следующих друг за другом

In [183…

```
[i
 for i in range(100)
 if i > 10
 if i < 30
 if i % 2]
```

Out[183]: [11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

Синтаксис списковых включений позволяет организовать *несколько уровней вложенности* для прохода по нескольким последовательностям.

Списковое включение с двумя уровнями вложенности имеет следующий синтаксический шаблон:

```
[expr(elem_ext,elem_int) for elem_ext in sequence_ext for elem_int in
sequence_int]
```

In [184…

```
words = ['word1', 'word2', 'word3']
letters = [f'{letter} in {word}'
           for word in words
           for letter in word]
print(letters)
```

```
['w in word1', 'o in word1', 'r in word1', 'd in word1', '1 in word1', 'w in word2'
, 'o in word2', 'r in word2', 'd in word2', '2 in word2', 'w in word3', 'o in word3
', 'r in word3', 'd in word3', '3 in word3']
```

Списковые включения могут быть вложенными друг в друга

In [185…
```
rows = 3; columns = 5
matrix = [[f'a{i}{j}'
           for j in range(columns)]
          for i in range(rows)]
matrix
```

Out[185]:
```
[['a00', 'a01', 'a02', 'a03', 'a04'],
 ['a10', 'a11', 'a12', 'a13', 'a14'],
 ['a20', 'a21', 'a22', 'a23', 'a24']]
```

В Python 3.X и 2.7 не только списки, но и словари, множества и *генераторные объекты* могут быть построены по аналогии с синтаксисом списковых включений

In [186…
```
{x.upper() for x in 'strings'*2}
```

Out[186]:
```
{'G', 'I', 'N', 'R', 'S', 'T'}
```

In [187…
```
{i: x for (i,x) in enumerate('string')}
```

Out[187]:
```
{0: 's', 1: 't', 2: 'r', 3: 'i', 4: 'n', 5: 'g'}
```

In [188…
```
(x for x in 'string')
```

Out[188]:
```
<generator object <genexpr> at 0x000001887FE4B030>
```

Включения относят к инструментам функционального программирования в Python. Похожий инструмент существует, например, в языке функционального программирования Haskell, а также в Maple-языке, используемом в одноименной математической системе символьных вычислений.

Включения выполняются со скоростью кода на языке C внутри интерпретатора, которая гораздо выше, чем скорость выполнения байт-кода циклов `for` внутри PVM.

В случае применения включений нужно стараться сохранять их простыми. Для более сложных задач нужно использовать оператор циклов `for` .

2.11 Форматированный вывод строк

Существует три способа форматированного вывода строк на основе *подстановок* в исходную форматную строку объектов произвольных типов согласно шаблону форматной строки:

- бинарная операция `%` : `'...%s...%d...' % (obj1, obj2)`
- строковый метод `format` : `'...{}...{}...'.format(obj1, obj2)`
- форматированный строковый литерал `f'...{x}...{y}...'`

Способ на основе бинарной операции `%` существует с момента создания Python. Он основан на синтаксисе функции `printf` языка C и часто встречается в коде Python. Синтаксис функции `printf` можно посмотреть, например, по ссылке https://ru.wikipedia.org/wiki/Printf.

Способ на основе строкового метода `format` появился в версии Python 2.6. Он частично совпадает по функциональности с использованием операции `%` .

Способ на основе форматированного строкового литерала (*f-string*) появился в версии Python 3.6.

Приведем пример создания строкового объекта на основе подстановок в исходную форматную строку пяти объектов различных встроенных типов `(1, 2.34, 5.6J, [7], {8})`

In [189…
```
objs = (1, 2.34, 5.6J, [7], {8})
```

In [190…
```
'int: %s, float: %s, complex: %s, list: %s, set: %s' % objs
```

Out[190]:
```
'int: 1, float: 2.34, complex: 5.6j, list: [7], set: {8}'
```

`%s` — это спецификатор формата для преобразования произвольного объекта в строку. Спецификатор формата также указывает место вставки значения объекта. Python поддерживает все спецификаторы формата функции `printf` языка C. Другие спецификаторы формата можно посмотреть, например, по ссылке https://ru.wikipedia.org/wiki/Printf.

Строковый метод `format` `'...{}...{}...'.format(obj1, obj2)` НЕ основан на синтаксисе функции `printf` языка C.

Фигурные скобки `{}` внутри исходной форматной строки, к которой применяется метод `format` , обозначают место подстановки объекта. Аргументы функции `format` могут подставляться в исходную форматную строку по порядку следования аргументов (`{}`), по позиции аргумента (например, `{1}`) или по ключу аргумента (например, `{key}`)

In [191…
```
'int: {}, float: {}, complex: {}, list: {}, set: {}'.format(*objs)
```

Out[191]:
```
'int: 1, float: 2.34, complex: 5.6j, list: [7], set: {8}'
```

In [192…
```
'int: {0}, float: {1}, complex: {c}, list: {l}, set: {s}'.format(1, 2.34, c=5.6J, l
```

Out[192]:
```
'int: 1, float: 2.34, complex: 5.6j, list: [7], set: {8}'
```

Фигурные скобки `{}` внутри форматированного строкового литерала (*f-string*) обозначают место подстановки объекта. В фигурных скобках может быть указано ВЫРАЖЕНИЕ, результат вычисления которого будет подставлен в соответствующем месте.

In [193…
```
f'int: {objs[0]}, float: {objs[1]}, complex: {objs[2].real}, list: {len(objs[3])},
```
Out[193]:
```
'int: 1, float: 2.34, complex: 0.0, list: 1, set: {8}'
```

Сравним три способа форматированного вывода стро по времени выполнения

In [194…
```
%timeit 'int: %s, float: %s, complex: %s, list: %s, set: %s' % objs
```
```
6.44 µs ± 137 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [195…
```
%timeit 'int: {}, float: {}, complex: {}, list: {}, set: {}'.format(*objs)
```
```
7.96 µs ± 382 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [196…
```
%timeit f'int: {objs[0]}, float: {objs[1]}, complex: {objs[2]}, list: {objs[3]}, se
```
```
7.4 µs ± 122 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Пример подстановки имени переменной и ее значения `f'...{var=}...:...'` с применением дополнительного форматирования числового результата

In [197…
```
f'{objs[0]=:10}', f'{objs[0] = :.1%}'
```
Out[197]:
```
('objs[0]=         1', 'objs[0] = 100.0%')
```

Пример подстановки результата вычисления выражения `f'...{expression}...:...'` с применением дополнительного форматирования числового результата

In [198…
```
f'{(objs[0]+objs[1]**2)%2:.3e}'
```
Out[198]:
```
'4.756e-01'
```

При использовании форматированного строкового литерала `f'...'` определение исходной форматной строки и подстановка происходят в одном выражении. Строковый метод `format` позволяет определить исходную форматную строку в одной части кода, а сделать подстановку в других частях кода

In [199…
```
template = "http://km.mmf.bsu.by/courses/{year}/{course}.html"

# More code here

url = template.format(year='2023', course='km2python'); url
```
Out[199]:
```
'http://km.mmf.bsu.by/courses/2023/km2python.html'
```

2.12 Файловый объект

Файловый объект в Python создается вызовом встроенной функции `open(file, mode)`, аргументами которой являются имя внешнего файла `file` в виде строки и режим обработки файла `mode` в виде строки:

In [200…
```
file = open('tmp.txt','rt'); print(file)
```

```
<_io.TextIOWrapper name='tmp.txt' mode='rt' encoding='cp1251'>
```

Созданный файловый объект `file` предоставляет интерфейс для работы с внешним файлом на компьютере.

Python 3.X проводит четкое различие между текстовыми и двоичными данными в файлах.

- **Текстовые файлы** представляют содержимое в виде строк типа `str`, автоматически выполняя кодирование и декодирование Unicode при записывании и чтении данных.
- **Двоичные файлы** представляют содержимое в виде специальных строк `bytes` и позволяют получать доступ к содержимому файла, не изменяя его.

Встроенный тип данных **байтовая строка (bytes)** — это последовательность символов для хранения и представления *байтовой* информации, используемой в файлах медиа-данных и при передаче через сеть.

Некоторые режимы обработки файла:

- `'r'` — открыть для чтения; *является значением по умолчанию*
- `'w'` — открыть для записи; если файл существует, то содержимое файла удаляется
- `'a'` — открыть для до записи в конец файла; если файл существует, то содержимое файла не удаляется
- `'b'` — открыть в двоичном режиме
- `'t'` — открыть в текстовом режиме; *является значением по умолчанию*
- `'+'` — открыть для чтения и записи

Различные режимы обработки файла могут быть объединены в одну строку. Например, режим обработки файла `'rb'` означает открыть для чтения в двоичном режиме. По умолчанию режим равен `'rt'` -- открыть для чтения в текстовом режиме.

После окончания работы с файлом он должен быть закрыт с помощью метода `close()` файлового объекта

In [201…
```
file.close()
```

Чтение файла и запись в файл

Метод `read()` файлового объекта читает ВЕСЬ файл в строковый объект типа `str` или `bytes` в зависимости режима обработки файла

In [202…
```
file = open('tmp.txt','rb')
print(file.read())
file.close()
```

```
b'the first line\r\nthe second line\r\nthe last line\r\n'
```

При вызове метода в виде `read(num)` из файла считывается `num` символов

In [203…
```
file = open('tmp.txt','rb')
print(file.read(10))
file.close()
```

```
b'the first '
```

Метод `readline` файлового объекта читает строку файла до символа `\n` включительно

In [204…
```
file = open('tmp.txt')
file.readline()
```

Out[204]:
```
'the first line\n'
```

In [205…
```
print(file.readline())
file.close()
```

```
the second line
```

Эффективным способом построчного чтения файла в Python является использование цикла `for` для файлового объекта, не вызывая метод `readline`

In [206…
```
file = open('tmp.txt')
for line in file:
    print(line)
file.close()
```

```
the first line

the second line

the last line
```

Метод `readlines` файлового объекта читает весь файл в список строк

In [207…
```
file = open('tmp.txt')
print(file.readlines())
file.close()
```

```
['the first line\n', 'the second line\n', 'the last line\n']
```

Обратите внимание, что проход по файловому объекту можно сделать только один раз! Поэтому, в частности, перед вызовами методов `read`, `readlines` необходимо повторное создание файлового объект с помощью функции `open`.

Метод `write(string)` файлового объекта записывает одну строку `string` в файл

In [208…
```
file = open('tmp.txt','a') # открыть для дозаписи в конец файла
file.write(str(100) + '\n')
file.close()
file = open('tmp.txt')
print(file.read())
file.close()
```

```
the first line
the second line
the last line
100
```

Обратите внимание, что аргументом метода `write` является строка, автоматического преобразования типов происходить не будет. Также символ конца строки `\n` автоматически добавляться не будет при повторном вызове функции `write`.

Метод `writelines(list_of_strings)` файлового объекта записывает список строк `list_of_strings` в файл

In [209…
```
file = open('tmp.txt','a') # открыть для дозаписи в конец файла
file.writelines(['200 \n', '300 \n'])
file.close()
file = open('tmp.txt','r')
print(file.read())
file.close()
```

```
the first line
the second line
the last line
100
200
300
```

Оператор `with` для файлового объекта гарантирует закрытие файла без вызова метода `close()` для файлового объекта. Файл закроется *автоматически* после выполнения вложенного блока оператора `with`.

In [210…
```
with open('tmp.txt') as file:
    for line in file:
        print(line, end="")
```

```
the first line
the second line
the last line
100
200
300
```