ТЕХНОАТОМ

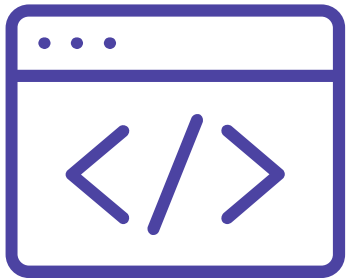
Веб-программирование Python

Лекция 2. Основы Python

Михалев Олег

Сегодня

- Встроенные типы данных
- Словари, списки и множества
- Функции



В Python применяется динамическая типизация. Информация о типе хранится в объекте, а переменная ссылается на объект.

В Python все данные - это объекты.
Даже None.

Каждый объект имеет атрибут **__class__** - класс объекта, получить его также можно вызвав встроенную функцию **type**.

```
1. >>> (1).__class__ == type(1)
2. True
```

Большинство объектов имеют атрибут **__dict__** - внутреннее пространство имен объекта, получить имена в локальной области объекта также можно вызвав встроенную функцию **dir** (но **dir** использует данные **__dict__** только когда не определен специальный метод).

```
1. >>> dir(1)
2. ['__bool__', '__int__', ...]
```

Объекты

ТЕХНОАТОМ

```
>>> (1).__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute '__dict__'
>>> (1).__dir__()
['__bool__', '__int__', ...]
```

В __**dict**__ нет методов и атрибутов класса объекта, все они определяются из __**class**__ и принадлежат именно классу, а не самому объекту.

Некоторые объекты имеют метод **__hash__** - целочисленный хэш-идентификатор объекта, его также можно вызвав встроенную функцию **hash**.

```
1. >>> (1).__hash__() == hash(1.0)
2. True
```

```
>>> hash([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Unmutable типы

"Неизменяемые" типы

Mutable-типы

"Измениямые" типы

"Неизменяемость" - гарантия того, что объект (не переменная) будет оставаться постоянным на протяжении всей работы программы.

Логические значения - unmutable тип

True
False

Логические значения поддерживают операции or, and, и not.

Логические значения поддерживают арифметические и битовые операции, а также операции сравнения, при этом **True** эквивалентно единице, **False** - нулю.

```
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Числа - unmutable тип

Целые числа
Вещественные числа
Комплексные числа

Целые числа могут записываться в двоичной, восьмеричной, десятеричной и шестнадцатеричной системах счисления. По умолчанию используется десятеричная.

```
1. >>> number = 3572 # base 10
2. >>> number = 0b110111110100 # base 2
3. >>> number = 0o6764 # base 8
4. >>> number = 0xdf4 # base 16
```

Целые числа поддерживают арифметические операции:

+	сложение
-	вычитание
*****	умножение
/	деление
******	возведение в степень
//	целочисленное деление
%	деление по модулю

Целые числа поддерживают операции сравнения:

==	равно
!=	не равно
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно

Целые числа поддерживают побитовые операции:

&	бинарный "И"
|	бинарный "Или"
^	бинарный "Исключающее Или"
~	бинарный "Не"
<<	бинарный сдвиг влево
>>	бинарный сдвиг вправо

Целые числа поддерживают логические операции, при этом **False** эквивалентно нулю, а все остальные числа трактуются как **True**.

Вещественные числа записываются в нормальной и экспоненциальной формах.

```
1. >>> number = 3.658 # normal
2. >>> number = 3658e-3 # exp
```

Вещественные числа поддерживают все те же операции, что и целые, за исключением побитовых операций.

Комплексные числа содержат действительную и мнимую часть.
Мнимая часть может указываться без действительной составляющей.

```
1. >>> number = 3+4j
2. >>> number.real
3. 3.0
4. >>> number.imag
5. 4.0
```

Комплексные числа поддерживают все те же операции, что и целые, за исключением побитовых операций и операций сравнения.

Можно использовать числа с фиксированной точностью.

```
1. >>> from decimal import Decimal
2. >>> number = Decimal('0.36')
```

Можно использовать рациональные числа.

```
1. >>> from fractions import Fraction
2. >>> number = Fraction(1, 3) # 1/3
```

Строки - unmutable тип

Стоит помнить, что строки не изменяются (например, в случае конкатенации двух объектов-строк создается третий объект).

Конкатенация и дублирование

```
>>> 'one' + 'two'
'onetwo'
>>> 'one' * 3
'oneoneone'
```

Доступ по индексу и извлечение среза

```
1. >>> ('string')[3]
2. 'i'
3. >>> 'string'[::-1]
4. 'gnirts'
5. >>> ('string')[2:]
6. 'ring'
```

Строки - unmutable тип, изменить символ по индексу нельзя.

Поиск подстроки в строке можно осуществить с помощью оператора in, или воспользоваться методом find.

```
1. >>> 'substr' in 'string with substr'
2. True
3. >>> 'string with substr'.find('substr') >= 0
4. True
```

Строки - unmutable тип, изменять их эффективнее используя списки.

```
1. >>> result = []
2. >>> for i in range(5):
3. ...     result.append('%d' % i)
4. ...
5. >>> ', '.join(result)
6. '0, 1, 2, 3, 4'
```

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов.

```
1. >>> ls = list()
2. >>> ls = []
```

Списки (а также множества и словари) можно задавать списковыми включениями (comprehensions).

```
>>> [i ** 2 for i in range(5)]
[0, 1, 4, 9, 16]
```

Списковые включения могут содержать и более сложную логику.

```
>>> [
...     i + j
...     for i in 'ab'
...     for j in '1234'
...     if j != '1'
... ]
[
  'a2', 'a3','a4',
  'b2', 'b3', 'b4'
]
```

Доступ по индексу и извлечение среза

```
1. >>> ls = [1, 2, 3, 4, 5]
2. >>> ls[3]
3. 4
4. >>> ls[2:4]
5. [3, 4]
6. >>> ls[::2]
7. [1, 3, 5]
```

Удаление элемента по индексу

```
1. >>> ls = [1, 2, 3, 4, 5]
2. >>> del ls[2]
3. >>> ls
4. [1, 2, 4, 5]
```

Списковые методы:

append(<элемент>) - добавление элемента в конец
insert(<позиция>, <элемент>) - добавление элемента на позицию
extend(<список>) - добавление всех элементов указанного списка
remove(<элемент>) - удаление по значению
pop(<позиция>) - "выталкивание" элемента по индексу
sort(<функция сортировки>) - сортировка списка
copy() - копирование списка
clear() - очистка списка

Поиск элемента в списке можно осуществить с помощью оператора **in**, или воспользоваться методом **index**.

```
1. >>> 3 in [1, 2, 3, 4]
2. True
3. >>> [1, 2, 3, 4].index(3) >= 0
4. True
```

Кортежи в Python - упорядоченные неизменяемые коллекции объектов произвольных типов.

```
1. >>> ts = tuple()
2. >>> ts = (2,)
```

Кортежи - неизменяемый аналог списка, для них доступны все методы "только чтение". А еще они занимают меньше места в памяти.

Множества в Python - неупорядоченные изменяемые коллекции объектов произвольных типов, содержащие неповторяющиеся элементы.

```
1. >>> ss = set()
2. >>> ss.add('value 1')
3. >>> ss.add('value 2')
4. >>> ss.add('value 2')
5. >>> ss
6. {'value 1', 'value 2'}
```

Поведение множеств в Python во многом опирается на теорию множеств и их операции.

a == b множества равны
a <= b a включено в b
a >= b b включено в a

Поведение множеств в Python во многом опирается на теорию множеств и их операции.

|	объединение
&	пересечение
-	разность
^	симметрическая разность

Важно заметить, что элементами множеств могут быть только объекты "неизменяемых" типов.

```
>>> ss = set()
>>> ss.add([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Как и в случае со списками, для множеств есть неизменяемый аналог.

```
1. >>> frozenset([1, 2, 3, 3, 4])
2. frozenset({1, 2, 3, 4})
```

Словари в Python - неупорядоченные изменяемые коллекции произвольных объектов с доступом по ключу.

```
1. >>> ds = dict()
2. >>> ds = {}
```

Словарь устроен как хэш-таблица, и ключом могут выступать только объекты "неизменяемых" типов.

```
1. >>> ds = {'key': 'value'}
2. >>> ds['key']
3. 'value'
4. >>> ds[5] = 4.6
5. >>> ds
6. {'key': 'value', 5: 4.6}
```

Получать значения безопаснее методом **get**, в противном случае, если ключ не задан, мы получим исключение **KeyError**.

```
1. >>> ds = {'key': 'value'}
2. >>> ds.get('unknown')
3. None
```

Перебор словаря можно осуществлять напрямую в цикле **for**, или получать для этих целей конкретные последовательности с помощью методов **keys** (ключи), **values** (значения) или **items** (последовательность кортежей ключ-значение).

Модуль **collections** также содержит полезные типы данных - **namedtuple** (позволяющий ассоциировать позиции кортежа с удобными именами), **defaultdict** (позволяющий использовать значения по умолчанию для отсутсвующих ключей) и **deque** (предоставляющий интерфейс очередей).

Программа - представленная в объективной форме совокупность данных и команд, функционирующая для получения определенного результата

Программа
Модули
Функции
Инструкции
+
Данные

Модули и функции - это средство для группировки наборов инструкций, согласно представлению о разрабатываемой программной системе

- Иерархическая структура
- Постоянство признака разбиения
- Полнота
- **Простота**

Оператор **import** позволяет загрузить модуль, оператор **from** и конструкция **from <имя модуля> import** – загрузить определенные имена из модуля

```
1. >>> import math
2. >>> print(math.pi)
3. 3.141592653589793
4. >>> from math import sin
5. >>> sin(90)
6. 0.8939966636005579
```

Инструкция **def** создает функцию и связывает ее с именем – функция не существует, пока интерпретатор не выполнит ее. Каждая функция созданная с помощью **def** имеет имя, список аргументов и блок исполняемого кода, возвращающий результат при помощи оператора **return**.

```
1. >>> def hello(name):
2. ...   return 'Hello, ' + name
3. ...
4. >>> print(hello('World'))
5. Hello, World
```

```
1. >>> def nope():
2. ...   pass
3. ...
4. >>> print(nope())
```

Аргументы могут быть обязательными или иметь значение по умолчанию, при этом необязательные параметры указываются в конце списка.

```
>>> def hello(name, is_female=False):
...     if is_female:
...         print('Hello, Ms. ' + name)
...     else:
...         print('Hello, Mr. ' + name)
...
```

Функция может принимать переменное количество позиционных аргументов, при этом *args* будет являться кортежем.

```
1. >>> def hello(*args):
2. ...   for name in args:
3. ...       print('Hello, ' + name)
4. ...
5. >>> peoples = ('Alice', 'Bob')
6. >>> hello(*peoples)
7. Hello, Alice
8. Hello, Bob
```

Функция может принимать переменное количество именованных аргументов, при этом *kwargs* будет являться словарем.

```
>>> def hello(**kwargs):
...     for login, name in kwargs.items():
...         print('Hello, ' + name + ' (' + login + ')')
...
>>> peoples = {'alice432': 'Alice', 'xxxSNIPERxxx': 'Bob'}
>>> hello(anonymous='Guest', **peoples)
Hello, Alice (alice432)
Hello, Bob (xxxSNIPERxxx)
```

```
>>> def he(*args, exclamation=True):
...     for name in args:
...             if exclamation:
...                     print('Hello, ' + name + '!')
...             else:
...                     print('Hello, ' + name)
...
>>> he('alice', 'bob')
Hello, alice!
Hello, bob!
>>> he('alice', 'bob', exclamation=False)
Hello, alice!
Hello, bob!
>>> he('alice', 'bob', False)
```

При определении функции ее аргументы должны указываться в следующем порядке:

- Обычные аргументы
- Аргументы со значением по умолчанию
- Кортеж позиционных аргументов переменной длины
- Именованные аргументы
- Словарь именованных аргументов переменной длины

При вызове функции ее аргументы должны передаваться в следующем порядке:

- Позиционные аргументы
- Именованные аргументы
- Кортеж позиционных аргументов переменной длины
- Словарь именованных аргументов переменной длины

Схема сопоставления аргументов:

- Сопоставление неименованных аргументов по позициям
- Сопоставление именованных аргументов по именам
- Сопоставление неименованных аргументов с кортежем
- Сопоставление именованных аргументов со словарем
- Сопоставление значение по умолчанию для именованных аргументов, которые не были найдены

"Изменяемые" объекты передаются по ссылке, "неизменяемые" - по значению. Чтобы гарантированно избежать возможных изменений необходимо либо использовать "неизменяемые аналоги", либо - передавать копии.

Инструкция **lambda** создает анонимную функцию, которая содержит только выражение, возвращающее значение.

```
1. >>> (lambda n: n ** 3)(2)
2. 8
```

Анонимная функция не имеет имени, но может быть вызвана через переменную, которой она присвоена.

```
1. >>> f = lambda n: n ** 3
2. >>> f(3)
3. 8
```

Анонимные функции совместно со списковыми включениями и тернарным оператором условий - довольно мощный синтаксический инструмент, но не стоит переусложнять код.

В Python функции - это тоже объекты. Функции высших порядков (супер-функции) генерируют объекты функций в процессе исполнения.

При этом, благодаря областям видимости, можно использовать замыкания.

```
1. >>> def pow(n):
2. ...     def f(x):
3. ...             return x ** n
4. ...     return f
5. ...
6. >>> sqr = pow(2)
7. >>> sqr(4)
8. 16
```

Опишем функцию-обертку, которая позволит засекать время исполнения кода внутри.

```
>>> from time import time
>>> def timeit(f):
...     def wrapper(*args, **kwargs):
...             start = time()
...             result = f(*args, **kwargs)
...             print(time() - start)
...             return result
...     return wrapper
...
```

Опишем произвольную функцию и обернем ее таймером.

```
1. >>> def f(n):
2. ...     i = 0
3. ...     while i < n:
4. ...             i += 1
5. ...     return n
6. ...
7. >>> f = timeit(f)
8. >>> f(500000)
9. 0.0322 # time
10.500000 # result
```

Но что, если оберток много?

Декораторы помогают сохранить наглядность кода.

```
>>> @timeit
... def f(n):
...     i = 0
...     while i < n:
...             i += 1
...     return n
...
>>> f(500000)
0.0391 # time
500000 # result
```

Декораторы помогают структурировать изменения.

```
1. >>> def decorator(f):
2. ...     print('construct function')
3. ...     def wrapper(*args, **kwargs):
4. ...         print('before function start')
5. ...         result = f(*args, **kwargs)
6. ...         print('after function start')
7. ...         return result
8. ...     return wrapper
```

Спасибо за внимание!

Михалев Олег
mailto:mhalairt@gmail.com