PYTHON

Содержание

1. Модули

- `import` — импорт модуля
- `from` — получение имен из модуля
- `imp.reload` — повторная загрузка модуля
- `as` — задание псевдонима для импортированного модуля или атрибута

```
1  import mymodule
2  mymodule.f()
3
4  import mymodule as md
5  md.f()
6
7  from mymodule import f
8  f()
9
10 from mymodule import *
11 f()
```

Импорт (загрузка) с помощью `import mymodule` модуля производится только **один раз**, при этом выполняются следующие действия:

- поиск файла модуля `mymodule.py` (по стандартному пути);
- компиляция в байт-код (`pyc`);
- запуск кода модуля для создания объектов, определенных в нем.

Если файла запускается как главный файл программы, то
`__name__ == '__main__'`,
иначе `__name__ ==` имя модуля.

1.1. **Пакеты модулей.**

Пакет модулей — это каталог с файлом `__init__.py`

```
import package.module1
package.module1.sqr(3)

import package.module1 as md
md.sqr(3)

from package.module1 import sqr
sqr(3)

from package import module1
module1.sqr(3)

from package import *
module1.sqr(3)
```

Пример файла `__init__.py`:

```
print('Welcome to package')
__all__ = ['module1']
```

Если нужен поиск только в пределах пакета:

```
from . import string
```

2.1. **Определение класса.**

```
1 class MyClass:
2     value = 0                  # атрибут класса
3     def method(self, param): # метод
4         self.member = param    # атрибут экземпляра
5         value = 3              # локальная переменная
6         print(self.member, MyClass.value, value)
7
8 ob = MyClass()     # экземпляр класса
9 ob.method('aaa')  # вызов метода
```

В Python объекты бывают:

- объекты классов
- объекты экземпляров

Атрибуты — данные объекта и его методы.

`class` — инструкция, которая создает объект класса.

`self` — ссылка на сам экземпляр класса, первый аргумент метода.

Конструктор:

```
class A:
   def __init__(self, x):
      self.x = x
   def method(self, param):
      print('x = {}'.format(self.x))
```

2.2. **Наследование.**

В Python3 используется только «новый стиль» классов, в котором:

- все классы явно или неявно являются потомками класса `object`.
- допускается множественно наследование.

```
1  class A:
2     def __init__(self):
3        print('init class A')
4
5  class B(A):
6     def __init__(self):
7        A.__init__(self)
8        print('init class B')
9
10 b = B()
```

Наследование атрибутов:

экземпляр-класса-B → класс-B → класс-A

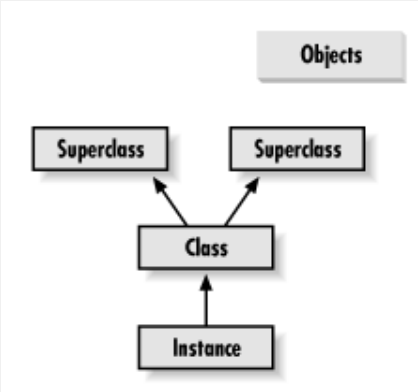

Рис. 1. Дерево наследования

`object.attr` запускает поиск атрибута в дерере наследования.

Атрибуты экземпляров:

- `__class__` — ссылка на класс

Атрибуты классов:

- `__base__` — ссылка на базовый класс (суперкласс)
- `__bases__` — суперклассы

Атрибуты классов и экземпляров:

- `__dict__` — словарь пространства имен

Слоты — ограничить множество разрешенных атрибутов (не доступен `__dict__`):

```
class A:
    __slots__ = ['age', 'name']
```

Методы.

- экземплярные: `obj.method()`
- статические: `MyClass.meth(x)`
- методы класса: `MyClass.meth(x); obj.method()`

```
class MyClass:
    def imethod(self, x): pass  # экземплярный
    def smethod(x): pass        # статический
    def cmethod(cls, x): pass    # метод класса
    smethod = staticmethod(smethod) # не обязат. в Python3
    cmethod = classmethod(cmethod)
```

2.3. **Рекомендации.**

1. Если вы видите класс с двумя методами, включая `__init__`, то это не класс. Достаточно одной функции. В стандартной библиотеке полно готовых классов.

2. Не создавайте новых исключений, если они не нужны (а они не нужны).

2.4. **Магические методы или перегрузка операторов.**

Магические методы — специальные методы класса с именами вида `__method__`, которые вызываются, когда экземпляр участвует во встроенных операциях.

Таблица 1. Magic Methods

Имя метода	**Название**	**Когда вызывается**
`__new__(cls,...)`	new	до конструктора
`__init__(self,...)`	конструктор	obj = MyClass(args)
`__del__(self)`	деструктор	при сборке мусора (не при del()!)
`__str__` `__repr__`	вывод, представление	`str(obj), print(obj)` `repr(obj)`
`__call__`	вызов функции	`obj()`
`__len__`	длина	`len(obj)`
`__add__`	сложение	`x + y; x += y`
`__or__`	побитовое ИЛИ	`x | y`
`__contains__`	вхождение	`in obj`

`str(), print()` сначала пытаются вызвать метод `__str__`. В остальных случаях вызывается `__repr__`.

Таблица 2. Magic Methods

Имя метода	**Название**	**Когда вызывается**
`__get__` `__set__`	дескрипторы	`obj.attr` `obj.attr = a`
`__getitem__` `__setitem__`	доступ по индексу	`obj[i]` `obj[i] = a`

`__getattribute__`
`__getattr__`
`__setattr__`
	обращ. к атрибуту	`obj.any`
		`obj.undefined`
		`obj.attr = value`

`__iter__` `__next__`	получение итератора	`for; iter(ojb)` `for; next(obj)`
`__enter__` `__exit__`	менеджеры контекста	`with obj as var:`

```python
try:
   print('try block')
   raise Exception('some_error')
   print('after raise')
except Exception as ex:
   print('there was an exception: ' + str(ex))
else:
   print('there was no exceptions')
finally:
   print('finally block')
```

Иерархия системных исключений:
BaseException ← Exception ← ArithmeticError ← ZeroDivisionError

Подклассы `BaseException`: SystemExit, KeyboardInterrupt, GeneratorExit.
Подклассы `Exception`: ArithmeticError, StopIteration, SyntaxError и др.
`https://docs.python.org/3/tutorial/errors.html`

3.1. **assert.**

`assert` — замена «`if ... raise`» — проверяет условие на истинности. Если не верно, то возбуждает исключение.

```
1    assert count >= 0
```

Основные понятия:

- **контейнер** (`container`) — тип данных, предназначенный для хранения элементов: `list, tuple` и др.
- **итерируемый объект** (`iterable`) — любой объект, который может предоставить *итератор* (с помощью функции `iter()` или метода `__iter__()`).
- **итератор** (`iterator`) — это вспомогательный объект, который возвращает следующий элемент основного «контейнера» (при выполнении функции `next()` или метода `__next__()`);
- **функция-генератор** (`generator function`) — «ленивая» функция, содержащая `yield` (вместо `return`);
- **объект генератора** (`generator object`) — разновидность итерируемого объекта, который создается функцией-генератором либо генераторным выражением.
- **генераторное выражение** (`generator expression`) — выражение вида `(x * x for x in numbers)`;
- **списочное включение** (`list comprehension`) — выражение вида `[x * x for x in numbers]` или `{x * x for x in numbers}`

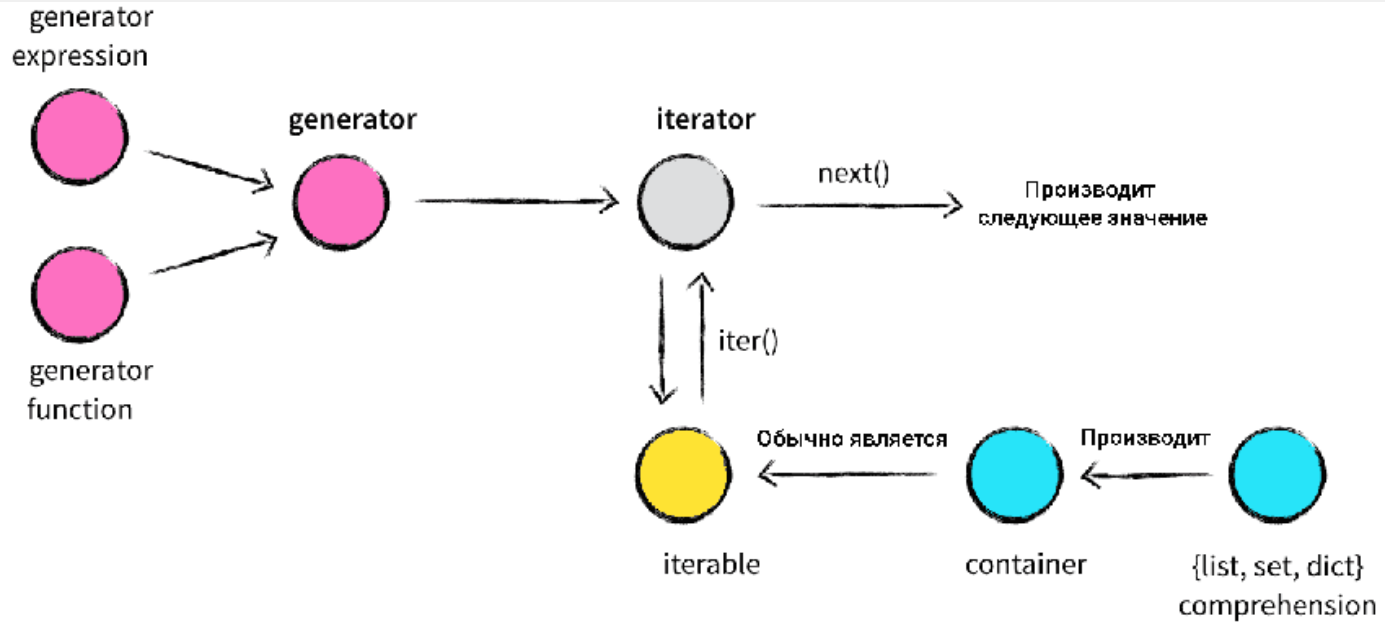

Рис. 2. https://shepetko.com/ru/blog/python-iterable-iterators-generators

4.1. List comprehension.

(чем проще, тем лучше)

```
>>> a = [1, 2, 3, -2]
>>> [x**2 for x in a]
[1, 4, 9, 4]
>>> {x**2 for x in a}
{1, 9, 4}
```

4.2. **Функции-генераторы.**

Функции-генераторы реализуют *отложенные вычисления* и автоматически поддерживают протокол итераций.

Функция-генератор не возвращается значение (`return`), а *поставляет* его (инструкция `yield`):

- при выполнении `yield` функция приостанавливает работу;
- функция передает значение вызывающей программе;
- функция возобновляет работу с того же места (после инструкции `yield`).

```
1 def genfunc():
2    for i in range(0, 5):
3       yield i**2
4 
5 for x in genfunc():
6    print(x)
```

или явно:

```
1 genobj = genfunc()
2 for x in genobj:
3    print(x)
```

Функция-генератор возвращает объект-генератор.

```
>>> genfunc()
<generator object genfunc at 0xb63f25a4>
```

Объект-генератор может использоваться только **один раз** (однократное выполнение итерационного процесса).

Итератор можно создать вручную:

```
>>> genobj = genfunc()  # возвращает объект-генератор
>>> it = iter(genobj)   # it - итератор
>>> next(it)
0
>>> next(it)
1
>>> list(genobj)
[4, 9, 16]
```

Еще один пример:

```
# A generator function that yields 1 for first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

4.3. **Генераторное выражение.**

Генераторное выражение — еще один способ создания объекта-генератора:

```
1 genobj = (i**2 for i in range(0,5))
```

Упражнение:

```
1 g = (print(i) for i in range(0,5))
2 list(g)
```

4.4. **Итерируемые объекты.**

Итератор с помощью `iter()` можно получить либо по генератору (функция-генератор или выражение-генератор), либо по итерируемому объекту.

Итерируемым объект — это объект, который может предоставить итератор. Говорят, что такой объект поддерживающий *интерфейс итератора*.

Итерируемым объектом является:

- большинство контейнеров (напр., list, tuple);
- любой объект, имеющий метод `__iter__()`.

Примеры:

```
1 it = iter(['a','b','c'])
2 next(it)
3 it.__next__()
```

```
class A:
   def __iter__(self):
      return self
   def __next__(self):
      return 'ok'
a = A()
it = iter(a)
print(next(it))
```

В последнем случае итератор совпадает с сами итерируемым объектом, поэтому можно написать (в общем случае это не так):

```
print(next(a))
```

Больше примеров:

1) `range` — это класс, экземплярами которого являются итерируемые объекты (но не контейнеры)

```
>>> r = range(10)
>>> it = iter(r)
>>> next(it)
0
```

2) Цикл `for` реализует итерационный контекст и работает следующим образом:

- пытается вызвать метод `__iter__`, и если он есть, то получает итератор и запускает итерационный процесс;
- в противном случае пытается вызвать метод `__getitem__(self,i)`, многократно применяя операцию индексирования с индексом i от 0 до ∞ — до тех пор, пока не будет сгенерировано исключение `StopIteration`.

```
1  class A:
2     def __getitem__(self, i):
3        if i > 1000:
4           raise StopIteration
5        return i
6  a = A()
7  for x in a:
8     print('x = ' + str(x))
```

(получается, достаточно иметь метод getitem, чтобы быть iterable)

Но лучше как-то так:

```
class Iterator:
   def __init__(self):
      self.k = 0
   def __next__(self):
      print('k = ' + str(self.k))
      self.k += 1
      if self.k > 1000:
         raise StopIteration
      return self.k

class A:
   def __iter__(self):
      return Iterator()

a = A()
for x in a:
   print(x)
```

4.5. Функции enumerate и sorted.

- `enumerate(iterable)` — нумерует элементы в iterable
- `sorted(iterable, key=None, reverse=False)`

Примеры:

```
sequence = [1, 2, 7, 19]

idx = 0
for item in sequence:
   print(idx, item)
   idx += 1

for idx, item in enumerate(sequence):
   print(idx, item)
```

4.6. Модуль itertools.

- `itertools.chain()` — объединяет два разных итератора в один:
  `for i in itertools.chain(it1, it2)`.
- `itertools.cycle()` — бесконечно повторяет заданную последовательность:
  `for i in itertools.cycle([1,2,3])`.

Императивный стиль — циклы и изменяемые переменные.

Например, найти $\sum_{i=1}^{N} x_i^3$:

```
a = [3, -2, 1, 4, 2, 0, 3]
s = 0
for x in a:
    s += x**3
```

Функциональный стиль — чистые функции, неизменяемые переменные, нет циклов while, for и т.п.

```
m = map(lambda x: x**3, a)
s = reduce(lambda x, y: x+y, m)
```

Фунциональный подход используется при программировании на основе *событий* («reactive»), а также при параллельном программировании.

Чистыми функциональными языками являются **ML**. Последнии версии C# (начиная с версии 3), JAVA (с версии 8) и C++11 поддерживают некоторые элементы функционального программирования.

5.1. **Средства функционального программирования на Python.** Функции в python — объекты первого порядка.

5.2. **lambda-выражения.**

`lambda arg1, arg2,..., argN : выражение`

```
add = lambda x, y, z: x + y + z
```

5.3. **Функция map.**

`map(func, iterable)` — применяет `func` ко всем элементам `iterable`, и возвращает «список» результатов.

Примеры:

```
map(str, [1, 2, 3])
map(lambda x: x**2, [1, 2, 3])
```

5.4. **Функция filter.**

`filter(func, iterable)` — применяет `func` ко всем элементам `iterable`, и возвращает «список» из элементов, для которых функция `func` вернула `True`.

Примеры:

```
filter(lambda x: x % 2 == 0, sequence)
```

5.5. **Функция reduce.**

`reduce(func, [a1,a2,a3,...]` — вычисляет `func(func(func(a1, a2), a3)...)`,

где `func` — функция двух переменных.

Примеры:

```
1 from functools import reduce
2 reduce(lambda x,y: x*y, [1,2,3,4])
```

Вычисление наибольшего элемента в списке при помощи reduce:

```
1 from functolls import reduce
2 items = [1, 24, 17, 14, 9, 32, 2]
3 all_max = reduce(lambda a,b: a if (a > b) else b, items)
4
5 print (all_max)
```

5.6. **Функция zip.** Функция `zip()` объединяет в кортежи элементы из последовательностей переданных в качестве аргументов:

```
a = [1, 2, 3]
b = "xyz"
c = (None, True)
res = list(zip(a, b, c))
print (res)
# [(1, 'x', None), (2, 'y', True)]
```

5.7. **Замыкания.**

Смысл замыкания состоит в том, что определение функции «замораживает» окружающий ее контекст на момент определения:

```
def multiplier(n):    # multiplier возвращает функцию умножения на n
    def mul(k):
        return n * k
    return mul

mul3 = multiplier(3)  # mul3 - функция, умножающая на 3
print(mul3(3), mul3(5))
```

5.8. **Частичное применение функции.**

Частичное применение функции предполагает на основе функции N переменных определение новой функции с меньшим числом переменных $M < N$, при этом остальные $N - M$ переменных получают фиксированные «замороженные» значения:

```
1 from functools import partial
2 def mulPart(a, b):    # частичное применение функции
3     return a * b
4 res = partial(mulPart,3)
```

5.9. **Карринг.**

Карринг (или каррирование, currying) — преобразование функции от многих переменных в функцию, берущую свои аргументы по одному.

```
def spam(x, y):
    print( 'arg1 =', x, ' arg2 =', y)

spam1 = lambda x: lambda y: spam(x, y)

def spam2(x):
    def new_spam(y):
        return spam(x, y)
    return new_spam

spam1(2)(3)      # карринг
spam2(2)(3)
```

5.10. **Функторы.**

Функтор — это объект класса, в котором определен метод с именем `__call__()`.

```
class mulFunctor:
    def __init__(self, val1):
        self.val1 = val1
    def __call__(self, val2):
        return self.val1 * val2
func = mulFunctor(3)
print(func(4))
```

6.1. **Декораторы.**

Метафункция — это функция или объект, который управляет другой функцией.

Пример:

```
1  def decor(f):    # метафункция
2     newf = lambda x: '<<< ' + str(f(x)) + ' >>>'
3     return newf
4
5  def func(x):     # наша функция
6     return(2*x)
7
8  print('func(3) = ' + str(func(3)))
9  func = decor(func)  # декорирование
10 print('func(3) = ' + str(func(3)))
```

В Python используется следующий (более явный) синтаксис — на основе декораторов:

```
@decor
def square(x):
   return(x**2)
```

что аналогично:

```
def square(x):
   return(x**2)
square = decor(square)
```

Декоратор (decorator) — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту (например, классу или функции).

Декораторы функций — это «обёртки» функций, которые дают нам возможность изменить поведение функции, не изменяя ее код.

Декораторы функций обеспечивают способ определения специальных режимов работы функций, обертывая их дополнительным слоем логики, реализованной в виде других функций.

Вложенные декотораторы:

```
@A
@B
@C
def f(): ...
```

работает как

```
f = A(B(C(f)))
```

Замечания:

1) Декоратором может быть класс:

```
class decor():
   def __init__(self, func):
      self.func = func
   def __call__(self, *args):
      return '<<< ' + str(self.func(args[0])) + ' >>>'
```

2) Декоратор может иметь параметры:

```
@paramdecor(a, b)
def f(): ...
```

Это эквивалентно

```
f = paramdecor(a,b)(f)
```

Пример из жизни:

```
app = Flask(__name__)
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

6.2. **Метаклассы.**

Метаклассы для классов подобны декораторам для функций. Они позволяют управлять процессом создания экземпляров классов или самих классов.

По умолчанию, любой **класс** — это *объект* класса `type`:

`class = type(classname, superclasses, attributedict)`

`type` — это пример метакласса.

Можно определить собственный метакласс, например с именем `Meta` и использовать его при создании своих классов:

```
class MyClass(metaclass=Meta):
   pass
```

В этом случае класс MyClass создается как объект класса Meta:

`class = Meta(classname, superclasses, attributedict)`

6.3. **Абстрактные суперклассы.**

```
from abc import ABCMeta, abstractmethod
class Super(metaclass=ABCMeta):
   @abstractmethod
   def method(self):
      pass
```

Singleton — данный паттерн позволяет создать всего один экземпляр класса. Используется метод `__new__`:

```
class Singleton(object):
  def __new__(cls, *args, **kw):
      if not hasattr(cls, '_instance'):
        orig = super(Singleton, cls)
        cls._instance = orig.__new__(cls, *args, **kw)
      return cls._instance
```

8. ПРИЛОЖЕНИЕ

8.1. **Полезные функции.**

`exec(obj[, globals[, locals]])` — динамически исполняет указанный код. Здесь obj — строка кода, либо объект кода.

`eval(expression, globals=None, locals=None)` — разбирает и исполняет указанное выражение. Возвращает какой-то результат.

Например, `x = 1; eval('x+1')`.

Функциям eval() и exec() можно передавать результаты функций globals() и locals().

8.2. **Рецепты.**

Определить индекс минимального элемента в списке values.

```
1 import operator
2 min_index, min_value = min(enumerate(values), key=operator.itemgetter(1))
3 max_index, max_value = max(enumerate(values), key=operator.itemgetter(1))
```

Список литературы

[1] Лутц, М. Изучаем Python / М. Лутц. — 4-ое изд. — Пер. с англ. — СПб.: Символ-Плюс, 2011. — 1280 с.

[2] Сузи Р. А. Python : [полное руководство]. — Санкт-Петербург [и др.] : БХВ-Петербург, 2002. — 748 с. [чз]

[3] Силен Д. Основы Data Science и Big Data. Python и наука о данных / [пер. с англ. Е. Матвеева]. — Санкт-Петербург [и др.] : Питер, 2017. — 336 с.