Лекция 4
Модули и скрипты

9 марта 2017 г.

Декораторы (использование)

Пример использования декоратора

```
def my_func(x):
    if x > 5:
        return 25
    else:
        return x**2

data = numpy.array([0.5, 8, 4.1, 25.2])
print my_func(data)
```

Пример использования декоратора

```
def my_func(x):
    if x > 5:
        return 25
    else:
        return x**2

data = numpy.array([0.5, 8, 4.1, 25.2])
print my_func(data)
```

```
ValueError: The truth value of an array with
more than one element is ambiguous. Use a.any()
or a.all()
```

Пример использования декоратора

```
@numpy.vectorize
def my_func(x):
    if x > 5:
        return 25
    else:
        return x**2

data = numpy.array([0.5, 8, 4.1, 25.2])
print my_func(data)
```

```
[  0.25  25.    16.81  25.  ]
```

Синтаксис декорации функции

Общая форма

```
@DECORATOR(DECORATOR_ARGS)
def FUNCTION(FUNCTION_ARGS):
    ...
```

- Модифицирует функцию после ее создания
- Может принимать аргументы
- Используется для упрощения кода
- (как писать — позже)

Декоратор `staticmethod`

`staticmethod`

- Применяется к методу класса
- Делает метод статическим
- Позволяет игнорировать экземпляр (`self`)

Декоратор staticmethod

```
class A(object):
    @staticmethod
    def f(a, b):
        return a + b

a = A()
print a.f(1, 2)
print A.f(1, 2)
```

```
3
3
```

Модули и пространства имен

Модуль как объект

```
>>> import math
>>> math
<module 'math' (built-in)>
```

Модуль как объект

```
>>> import math
>>> math
<module 'math' (built-in)>

>>> math.floor
<built-in function floor>
>>> math.ceil
<built-in function ceil>
```

Атрибуты модуля

```
>>> from math import floor
>>> math
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>> floor
<built-in function floor>
```

Пространства имен

Пространство имен (namespace) — логическое объединение идентификаторов (имен).
Одинаковые имена могут иметь разный смысл в разных пространствах имен.

Пространства имен

Пространство имен (namespace) — логическое объединение идентификаторов (имен).
Одинаковые имена могут иметь разный смысл в разных пространствах имен.

Атрибуты любого объекта — пространство имен.
(в т.ч. модулей)

Создание модулей

Пример модуля

Файл `count.py`

```
def count_lines(filename):
    with open(filename) as f:
        count = 0
        for line in f:
            count += 1
    return count
```

Использование модуля

(в той же директории)

```
>>> import count
>>> count.count_lines('some-file.txt')
181
```

Поиск модуля

```
import module_name
```

Поиск модуля

```
import module_name
```

- Поиск `module_name.py` в текущей директории.

Поиск модуля

```
import module_name
```

- Поиск `module_name.py` в текущей директории.
- Поиск `module_name.py` в директориях из списка PYTHONPATH.

PYTHONPATH и sys.path

PYTHONPATH — директории установленных библиотек.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python2.7', ...]
```

Изменение sys.path

(вне директории с `count.py`)

```
>>> import count
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named count
```

Изменение sys.path

(вне директории с `count.py`)

```
>>> import count
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named count

>>> sys.path.insert(0, 'dir/with/count')
>>> import count
```

Скомпилированные файлы

(в директории с `count.py`)

```
import count
...
```

Скомпилированные файлы

(в директории с `count.py`)

```
import count
...
```

В директории появился файл `count.pyc`

Скомпилированный код для ускорения последующих запусков.

Скрипты как модули

Файл `count.py`

```
import sys

def count_lines(filename):
    ...

if len(sys.argv) == 1:
    print "Not enough arguments."
else:
    print count_lines(sys.argv[1])
```

Скрипты как модули

В той же директории

```
$ python count.py
Not enough arguments.
$ python count.py some-file.txt
319
```

Скрипты как модули

В той же директории

```
$ python count.py
Not enough arguments.
$ python count.py some-file.txt
319
```

```
>>> import count
Not enough arguments.
>>> count.count_lines('some-file.txt')
319
```

Имя модуля

Атрибут модуля `__name__`.

- Отражает имя, с которым наш модуль импортировали.
  (почти всегда — имя файла без расширения).

Имя модуля

Атрибут модуля `__name__`.

- Отражает имя, с которым наш модуль импортировали.
  (почти всегда — имя файла без расширения).
- Когда модуль не импортировали (т.е. когда он главный) равно `__main__`.

Проверка имени модуля

Файл `count.py`

```
import sys

def count_lines(filename):
    ...

if __name__ == '__main__':
    if len(sys.argv) == 1:
        print "Not enough arguments."
    else:
        print count_lines(sys.argv[1])
```

Проверка имени модуля

```
import sys

def count_lines(filename):
    ...

def main():
    if len(sys.argv) == 1:
        print "Not enough arguments."
    else:
        print count_lines(sys.argv[1])

if __name__ == '__main__':
    main()
```

Скрипты как модули

В той же директории

```
$ python count.py
Not enough arguments.
$ python count.py some-file.txt
319
```

```
>>> import count
>>> count.count_lines('some-file.txt')
319
```

Обработка аргументов

Модуль argparse

- Модуль для работы с аргументами командной строки
- Обычно берет аргументы из `sys.argv`

Аргументы с argparse

```
import argparse

def count_lines(filename):
    ...

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('filename')
    args = parser.parse_args()
    print count_lines(args.filename)

if __name__ == '__main__':
    main()
```

Вызов с argparse

```
$ python count.py some-file.txt
444
```

Вызов с argparse

```
$ python count.py some-file.txt
444

$ python count.py
usage: count.py [-h] filename
count.py: error: too few arguments
```

Автоматическая справка

```
$ python count.py -h
usage: count.py [-h] filename

positional arguments:
  filename

optional arguments:
  -h, --help  show this help message and exit
```

Описание аргументов

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'filename',
        help='name for input file')
```

Описание аргументов

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'filename',
        help='name for input file')
```

```
$ python count.py -h
usage: count.py [-h] filename

positional arguments:
  filename    name for input file
...
```

Еще аргументы

```
...

def count_symbols(filename, line_index):
    with open(filename) as f:
        current = 0
        for line in f:
            if current == line_index:
                return len(line) - 1
            current += 1

...
```

Еще аргументы

```
parser.add_argument(
    'filename',
    help='Name for input file.')
parser.add_argument(
    '-l', '--line',
    type=int, default=None,
    help='count symbols in line')
args = parser.parse_args()
if args.line is None:
    print count_lines(args.filename)
else:
    print count_symbols(args.filename,
                        args.line)
```

Помощь

```
$ python count.py -h
usage: count.py [-h] [-l LINE] filename

positional arguments:
  filename              name for input file

optional arguments:
  -h, --help            show this help
                        message and exit
  -l LINE, --line LINE  count symbols in
                        line
```

Работа в разных режимах

```
$ python count.py some-file.txt
589

$ python count.py -l 25 some-file.txt
19

$ python count.py -l test
usage: count.py [-h] [-l LINE] filename
count.py: error: argument -l/--line:
invalid int value: 'test'
```

Описание скрипта

```
def main():
    parser = argparse.ArgumentParser()
    parser.description = (
        'Lines and symbols counting '
        'utilities.')
    ...
```

Описание скрипта

```
def main():
    parser = argparse.ArgumentParser()
    parser.description = (
        'Lines and symbols counting '
        'utilities.')
    ...
```

```
$ python count.py -h
usage: count.py [-h] [-l LINE] filename

Lines and symbols counting utilities.

...
```

Другие возможности argparse

- Группировка аргументов
- Аргументы переменного размера
- Взаимодействие между аргументами
- ...

Обработка ошибок

Типы ошибок

- Синтаксические

```
>>> x =
  File "<stdin>", line 1
    x =
      ^
SyntaxError: invalid syntax
```

Типы ошибок

- Синтаксические

```
>>> x =
  File "<stdin>", line 1
    x =
      ^
SyntaxError: invalid syntax
```

- Исключения

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or
modulo by zero
```

Типы исключений

Исключения — объекты.

Типы исключений

Исключения — объекты.

```
>>> x = {}
>>> x['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'

>>> y = [1, 2]
>>> y[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Типы исключений

```
>>> int('qwerty')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with
base 10: 'qwerty'

>>> int([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int() argument must be a string
or a number, not 'list'
```

Обработка исключений

```
try:
    x = [1, 2]
    print x[2]
except IndexError:
    print "Oops!"
```

```
Oops!
```

Конструкция try . . . except

```
try:
    TRY-CLAUSE
except ERROR-CLASS:
    EXCEPT-CLAUSE
```

Конструкция try . . . except

```
try:
    TRY-CLAUSE
except ERROR-CLASS:
    EXCEPT-CLAUSE
```

- Сначала исполняется `TRY-CLAUSE`
- Нет исключений → конец.

Конструкция try . . . except

```
try:
    TRY-CLAUSE
except ERROR-CLASS:
    EXCEPT-CLAUSE
```

- Сначала исполняется `TRY-CLAUSE`
- Нет исключений → конец.
- Есть исключение → обработка.
- Исключение имеет тип `ERROR-CLASS` → выполняется `EXCEPT-CLAUSE`, конец.
- Иначе → исключение «поднимается» дальше.

Примеры обработки исключений

```
while True:
    try:
        x = int(raw_input("Enter a number: "))
        break
    except ValueError:
        print "Invalid number. Try again."
```

Примеры обработки исключений

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error:", e.strerror
except ValueError:
    print "Data is not an integer"
except:
    print "Unexpected error"
```

Примеры обработки исключений

```
try:
    x = y[5]
except (NameError, IndexError) as e:
    print "Unexpected error"
```

Генерация исключений

```
>>> raise ValueError('qwerty')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: qwerty
```

Генерация исключений

```
>>> raise ValueError('qwerty')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: qwerty
```

```
def get_third(l):
    if len(l) < 3:
        raise ValueError("Too short.")
    else:
        return l[2]
```

Пользовательские исключения

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return str(self.value)

try:
    raise MyError(2*2)
except MyError as e:
    print 'My error, value:', e.value
```

Атрибуты исключений по умолчанию

```
class MyError(Exception):
    pass

try:
    raise MyError(str(2*2))
except MyError as e:
    print 'My error, value:', e.message
```

Иерархии исключений

```
class MyModuleError(Exception):
    pass

class MyIOError(MyModuleError):
    pass

class MyFloatError(MyModuleError):
    pass

try:
    ...
except MyModuleError:
    ...
```

Подходы к обработке ошибок

Look Before You Leap (LBYL)

```
def get_third_LBYL(l):
    if len(l) > 3:
        return l[2]
    else:
        return None
```

Подходы к обработке ошибок

Look Before You Leap (LBYL)

```
def get_third_LBYL(l):
    if len(l) > 3:
        return l[2]
    else:
        return None
```

Easier to Ask for Forgiveness than Permission (EAFP)

```
def get_third_EAFP(l):
    try:
        return l[2]
    except IndexError:
        return None
```

Пример EAFP

```
class CountError(Exception):
    pass

def count_symbols(filename, line_index):
    try:
        with open(filename) as f:
            current = 0
            for line in f:
                if current == line_index:
                    return len(line) - 1
                current += 1
        raise CountError("Line with index " +
            str(line_index) + " not found")
    except IOError:
        raise CountError("File not found")
```

Тестирование

Юнит-тесты

Общая идея

- Разбивать код на независимые части (юниты)
- Тестировать каждую часть отдельно

Юнит-тесты

Общая идея

- Разбивать код на независимые части (юниты)
- Тестировать каждую часть отдельно

Преимущества

- Нужно меньше тестов
- Проще отлаживать

Тесты «вручную»

```
def factorial(n):
    current = 1
    for i in range(1, n + 1):
        current *= i
    return current

def test_1():
    return factorial(1) == 1
def test_2():
    return factorial(5) == 120

if not (test_1() and test_2()):
    print "Tests failed"
```

Модуль unittest

```
import unittest

class TestFactorial(unittest.TestCase):
    def test_1(self):
        self.assertEqual(factorial(1), 1)
    def test_2(self):
        self.assertEqual(factorial(5), 120)

unittest.main()
```

(тесты должны называться `test*`)

Возможности unittest

- Отчеты
  (сколько сломалось, что сломалось)
- В чем проблема
  `assertTrue(), assertEqual()`
  `assertIn(), ...`
- Поиск тестов в директории

Проверка исключений

```
class TestFactorial(unittest.TestCase):
    ...
    def test_float():
        with self.assertRaises(TypeError):
            factorial(2.5)
```

Библиотека pytest

```
def test_1():
    assert factorial(1) == 1

def test_2():
    assert factorial(5) == 120
```

```
$ py.test <filename>
```

(тесты должны называться `test*`)

Проверка кода

assert

Проверка корректности «на лету»

```
def probability_of_smth(...):
    ...
    result = ...

    assert 0 <= result <= 1, \
        "Probability must be in [0, 1]"
    return result
```

assert

```
>>> assert True
>>> assert False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

assert

```
>>> assert True
>>> assert False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError

>>> assert False, "<description>"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: <description>
```

Устройство assert

```
assert CONDITION, TEXT
```

эквивалентно

```
if not CONDITION:
    raise AssertionError(TEXT)
```

Использование assert

- В корректной программе не срабатывают
- Обычно не используется для проверки аргументов

Использование assert

- В корректной программе не срабатывают
- Обычно не используется для проверки аргументов
- Отражают инварианты программы

Использование assert

- В корректной программе не срабатывают
- Обычно не используется для проверки аргументов
- Отражают инварианты программы
- Есть опция, отключающая их проверку для ускорения работы

Утилиты для проверки

(Проверка кода без выполнения)

- pep8.py
- PyChecker
- PyFlakes
- pylint
- (PyCharm)