Экспресс введение в Python 3.6

Муромцев Никита Андреевич
Выпускник ВМК МГУ,
аспирант геологического факультета МГУ
2017 год

Занятие 4

Консоль Windows, bat-скрипты, функциональное программирование, map, reduce, zip, filter, all, any, декораторы, документирование, классы (наследование, перегрузка, инкапсуляция)

Получение параметров командой строки

Программу можно запускать не только с помощью IDE PyCharm

Python скрипт можно запустить минуя запуск IDE. В случае Unix систем .py скрипт можно сделать исполняемым, и он будет считаться программой. Но во всех системах можно через консоль отдать команду выполнить .py скрипт интерпретатору pythonю

python <путь до файла> [<параметры командной строки>]

Получить список аргументов, с которыми был запущен скрипт, можно с помощью атрибута argv из модуля sys (т.е. sys.argv – список параметров, начиная с названия запускаемого файла)

```
import sys
if __name__ == "__main__":
    for param in sys.argv:
        print (param)
```

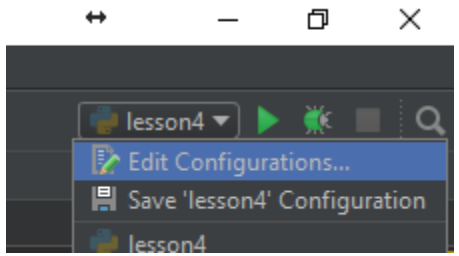
Команду можно сохранить в текстовый файл с расширением .bat, положить в одну папку с нашим скриптом и запускать выполнение скрипта через этот файл

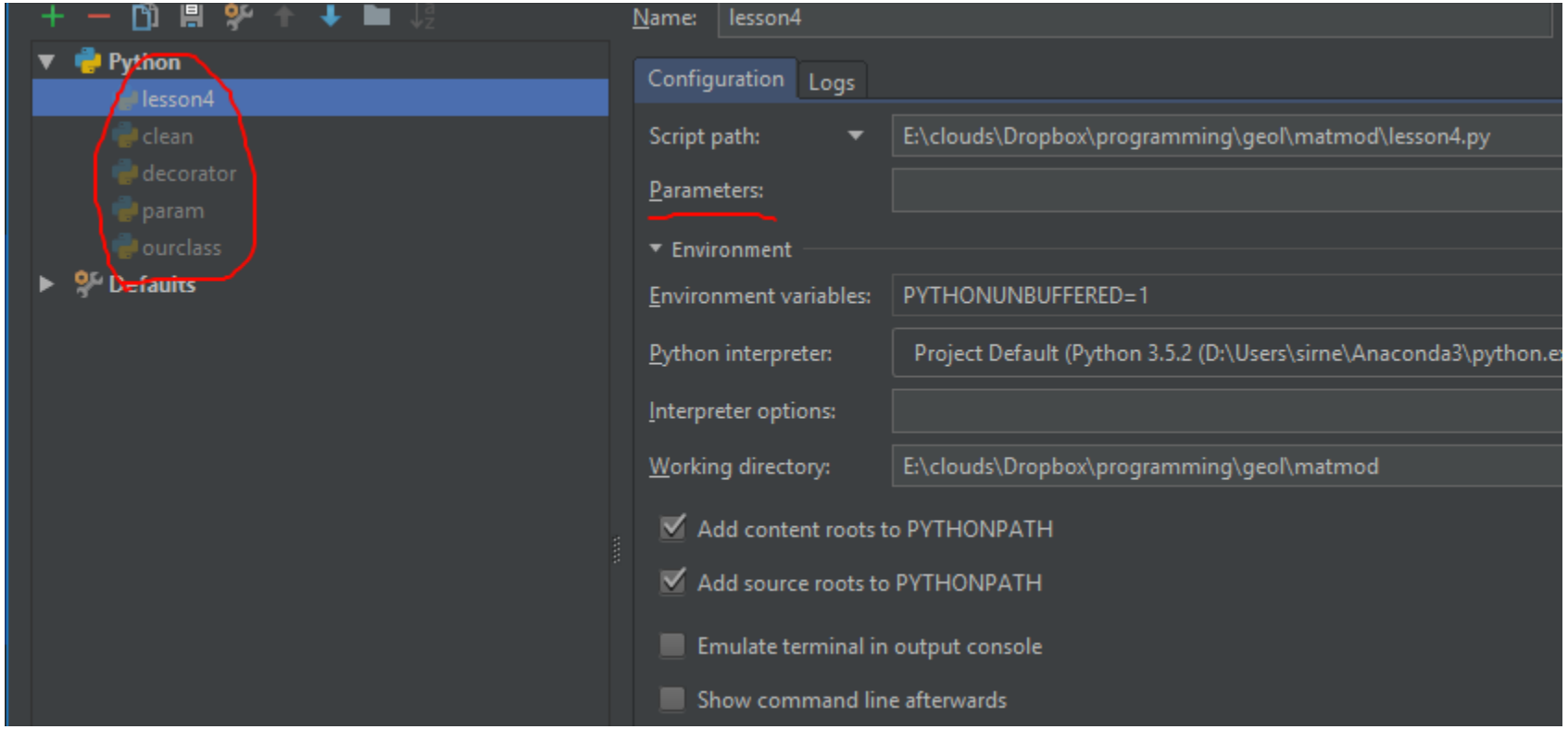
После выполнения команд в BAT-скрипте, окно консоли (CMD) закроется, чтобы отсрочить закрытие используйте команду pause(<количество секунд>)

Альтернатива – библиотека argparse

Параметры командной строки в PyCharm

Потребуется во время разработки, чтобы упростить запуск и отладку





Функциональное программирование

Функциональный подход упростит повторное использование кода

Главное в функциональном коде – отсутствие побочных эффектов.
Т.е. программа не полагается на данные вне контекста функции.

```
a = 0
def increment1():
    global a
    a += 1
```

Нефукциональный код

```
def increment2(a):
    return a + 1
```

Функциональный код

Map

Применяется для выполнения операции над каждым элементом списка

map(<функция>,<набор данных (список)>)
Результат – map элемент, используем list для преобразования

```
name_lengths = list(map(len, ['Маша', 'Петя', ‘Юля'])) # => [4, 4, 3]
squares = list(map(lambda x: x * x, [0, 1, 2, 3, 4])) # => [0, 1, 4, 9, 16]

new_list = list(map(int, ['1', '2', '3', '4', '5', '6', '7'])) # => [1, 2, 3, 4, 5, 6, 7]
```

Если же количество элементов в списках совпадать не будет, то выполнение закончится на минимальном списке:

```
l1 = [1,2,3]
l2 = [4,5]
new_list = list(map(lambda x,y: x + y, l1, l2)) # => [5,7]
```

Reduce

Перерабатывает список в итоговый результат (комбинацию элементов)

reduce(<функция>,<набор данных (список)>, [<стартовое значение>])
Результат - значение

```
s = sum(lambda x,y: x + y, [0, 1, 2, 3, 4]) # => 10

sentences = ['капитан джек воробей',
             'капитан дальнего плавания',
             'ваша лодка готова, капитан']
cap_count = reduce(lambda a, x: a + x.count('капитан'),
                   sentences,
                   0) # => 3
```

Zip

Объединяет списки

zip(<список> , <список>, [<список>])
Объединение закончится на минимальном списке

```
a = [1,2,3]
b = "xyz"
c = (None, True)

res = list(zip(a, b, c))
print (res)

# => [(1, 'x', None), (2, 'y', True)]
```

Filter

Объединяет списки

filter(<функция с булевым результатом (True/False)> , <список>)
Возвращает отфильтрованный список в соответствии с условием

```
mixed = ['мак', 'просо', 'мак', 'мак', 'просо', 'мак', 'просо', 'просо', 'просо', 'мак']
zolushka = list(filter(lambda x: x == 'мак', mixed)) # = > ['мак', 'мак', 'мак', 'мак', 'мак']
```

All и Any

Проверка выполнения условия для всех/хотя бы одного элементов списка

all(<генератор списка/список с логическими элементами (False/True)>)
Применяет ко всем элементам списка одновременно операцию «И»

any(<генератор списка/список с логическими элементами (False/True)>)
Применяет ко всем элементам списка одновременно операцию «ИЛИ»

```
b = [1,2,3,4,5,5,6,7,8]
print(all(x<9 for x in b)) # => True
print(all(x<3 for x in b)) # => False
print(any(x<3 for x in b)) # => True
```

Декораторы

Иногда к выполнению стандартной функции хочется добавить команд

Декоратор – обертка вокруг функции, с помощью него можно легко добавить вывод отладочной информации для всех используемых функций

```
def decor(k):                # k – фунция для декорирования
    def the_wrap(d):         # d – параметр декорируемой функции
        print(type(d))       # команды до декорируемой функции
        r = k(d)             # вызовы декорируемой функции
        pass                 # команды после вызова дек. функции
        return r             # возвращаем результат дек. функции
    return the_wrap          # возвращаем обёртку на функцию

fun = decor(str)             # fun – обернутая str
list = decor(list)           # перегрузили функцию list
```

Другой способ оборачивания функции:

```
@decor
def func(s):
    print("test", s)
    return s
```

Документирование кода

Помогает понять что делает функция/модуль другим разработчикам

Для документирования используются тройные кавычки

```
def func():
    """Документация"""
        pass
```

- Строки документирования идут сразу после заголовка функции/класса/метода, в начале файла
- Необходимо описываться что функция делает с получаемыми аргументами и что возвращает.

Посмотрите соглашения, которые используются при документировании

Документацию по объекту можно посмотреть с помощью атрибута __doc__:

```
print(str.__doc__)
```

Классы и объекты

Работаем в программе с объектами, а не списком операций

Пример класса

```
class man():
    age = 0 # переменная класса
    name = "" # переменная класса

    def __init__(self, n="", a=0): # конструктор запускается при создании объекта
        self.name = n # self - это объект для которого вызывается контруктор
        self.age = a

    def ages(self):
        print("возраст:", self.age) # вывести значение age для данного объекта

    def names(self):
        print("имя", self.name) # вывести значение name для данного объекта

a = man() # a - объект класса man со значениями по-умолчанию
a.age = 29 # поменяли нашему человеку возраст
a.names() # вывели имя человека
a.height = 185 # добавили в объект данного класса атрибут height
```

Классы и объекты - наследование

self - обязательный аргумент, содержащий в себе экземпляр класса, передающийся при вызове метода, поэтому этот аргумент должен присутствовать во всех методах класса.

```
class worker(man):
    def __init__(self, n="", a=0, j=None):
        # Необходимо вызвать метод инициализации родителя
        # В Python 3.x это делается при помощи функции super()
        super().__init__(n, a)
        self.job = j

    def jobs(self):
        print("возраст:", self.job) # вывести значение job для данного объекта

b = worker ("Вася", 23, "Газпром")
b.ages() # объект унаследовал методы класса-родителя
b.jobs() # объект имеет свои методы
a.jobs() # при вызове этих методов для объектов рожительского класса выведется ошибка
```

Концепция наследования в программировании позволяет сократить время разработки, упростить процесс написания программы как сейчас, так и в будущем, когда заказчик захочет внести "небольшие правки" в проект. Правильно спроектированный класс это, кроме прочего, гибкая структура, которую вы свободно сможете изменять, дополнять в будущем. Важно помнить, что иногда проще создать дополнительный базовый класс, наследовать от него и расширять по мере необходимости, чем сразу написать готовый класс.

Классы и объекты - инкапусуляция

Строки в апострофах и в кавычках - одно и то же

```
class A:
    def _private(self):
        # нижнее подчеркивание, чтобы указать, что метод/атрибут
        приватный, т.е. попросить его не трогать из вне
        print("Это приватный метод!")

    def __veryPrivate(self):
        print("Это очень приватный метод!")
        # двойное подчеркивание в начале имени атрибута даёт большую
        защиту: атрибут становится недоступным по этому имени
        # полностью это не защищает, так как атрибут всё равно остаётся
        доступным под именем _ИмяКласса__ИмяАтрибута

a = A()
a._private() # => Это приватный метод!
a.__Private() # ошибка
a__A__veryPrivate() # работает
```

Перегрузка операторов и полиморфизм

Строки в апострофах и в кавычках - одно и то же

Чтобы вывести список методов класса используем функцию dir()

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):  # перегрузили оператор сложения, т.е. +
        return Vector2D(self.x + other.x, self.y + other.y)

    def __str__(self):
        return '({}, {})'.format(self.x, self.y)


class V2D(Vector2D):
    def __init__(self, x, y):
        super().__init__(x, y)

    def __add__(self, other):  # перегрузили оператор сложения, т.е. +
        print("Перегрузили метод")
        return super().__add__(other)  # super помогает вызывать вообще любой
        метод родительского класса
```

```
a = Vector2D(1, 1)
b = Vector2D(2, 2)
c = V2D(3, 3)
d = V2D(4, 4)

print(a + b)
print(c + d)
print(c + b)
```

```
Вывод:
(3, 3)
Перегрузили метод
(7, 7)
Перегрузили метод
(5, 5)
```

Список доступных для перегрузки стандартных методов