Плюшки Python

Лавров Д.Н.
2018

Содержание

- Менеджеры контекста
- Декораторы
- Перегрузка операторов

Менеджеры контекста

- Инструкция with/as может использоваться как альтернатива известной конструкции try/finally
- Предназначена для выполнения заключительных операций независимо от того, возникло ли исключение на этапе выполнения основного действия.
- Инструкция with поддерживает более богатый возможностями сенарий поведения, позволяющий определять как предварительные, так и заключительные действия для заданного блока программного кода.

Менеджеры контекста

- Синтаксис

```
with expression [as target]:
    БлокОператоров
```

- Здесь предполагается, что выражение возвращает объект, поддерживающий протокол контекстного менеджера
- Этот объект может возвращать значение, которое будет присвоено переменной, если присутствует необязательное предложение **as**.

Менеджеры контекста

Что происходит при выполнении данного блока:

- Выполняется выражение в конструкции **with** ... **as**.
- Загружается специальный метод __exit__ для дальнейшего использования.
- Выполняется метод __enter__. Если конструкция **with** включает в себя слово **as**, то возвращаемое методом __enter__ значение записывается в переменную.
- Выполняется БлокОператоров.
- Вызывается метод __exit__, причём неважно, выполнилось ли БлокОператоров или произошло исключение. В этот метод передаются параметры исключения, если оно произошло, или во всех аргументах значение None, если исключения не было.

Менеджеры контекста

- Некоторые встроенные объекты языка Python были дополнены поддержкой сценария управления контекстом
- Пример

```
with open(r'C:\misc\data') as myfile:
    for line in myfile:
        print(line)
        остальной программный код
```

- После того как инструкция with начнет выполнение, механизм управления контекстом гарантирует, что объект файла, на который ссылается переменная myfile, будет закрыт автоматически, даже если в цикле for во время обработки файла произойдет исключение.

Декораторы

- *Декорирование* – это способ управления функциями и классами.
- Имеют две родственные разновидности:
  - *Декораторы функций* связывают имя функции с другим вызываемым объектом на этапе определения функции, добавляя дополнительный уровень логики, которая управляет функциями и методами или выполняет некоторые действия в случае их вызова.
  - *Декораторы классов* связывают имя класса с другим вызываемым объектом на этапе его определения, добавляя дополнительный уровень логики, которая управляет классами или экземплярами, созданными при обращении к этим классам.

Декораторы

- В двух словах, декораторы предоставляют возможность в конце инструкции def определения функции в случае декораторов функций или в конце инструкции class определения класса в случае декораторов классов добавить автоматически вызываемый программный код.

Декораторы

- Декораторы — это "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код.

Декораторы

```
def decorate(O):
    #Сохраняет или дополняет
    #функцию или класс O
    return O

@decorator
def F(): ... # F = decorator(F)

@decorator
class C: ... # C = decorator(C)
```

Декораторы

```
class tracer:
    # На этапе декорирования @:
    def __init__(self, func):
        # сохраняет оригинальную функцию func
        self.calls = 0 # счётчик
        self.func = func
        # При последующих вызовах: вызывает
        # оригинальную функцию func.
        def __call__(self, *args):
           self.calls += 1
           print('call %s to %s' % (self.calls, self.func.__name__))
           self.func(*args)
@tracer
def spam(a, b, c):  # spam = tracer(spam)
   print(a + b + c) # Обертывает функцию spam объектом декоратора
```

Декораторы

```
def benchmark(func):
    """
     Декоратор, выводящий время, которое заняло
     выполнение декорируемой функции.
    """
     import time
     def wrapper(*args, **kwargs):
         t = time.clock()
         res = func(*args, **kwargs)
         print(func.__name__,time.clock() - t)
         return res
     return wrapper
```

Декораторы

```
def counter(func):
    """
    Декоратор, считающий и выводящий количество вызовов
    декорируемой функции.
    """
    def wrapper(*args, **kwargs):
        wrapper.count += 1
        res = func(*args, **kwargs)
        print("{0} была вызван а: {1} раз".format(func.__name__, \
                                                  wrapper.count))
        return res
    wrapper.count = 0
    return wrapper
```

Декораторы

```
def benchmark(func):
    """
     Декоратор, выводящий время, которое заняло
     выполнение декорируемой функции.
    """
     import time
     def wrapper(*args, **kwargs):
         t = time.clock()
         res = func(*args, **kwargs)
         print(func.___name___,time.clock() - t)
         return res
     return wrapper
```

Декораторы

```
@benchmark
@counter
def reverse_string(string):
  return ''.join(reversed(string))
```

- `print(reverse_string('ABC'))`

```
Вывод результата:
reverse_string функция была вызвана 1 раз
('wrapper', 4.112026556215142e-05)
CBA
```

Декораторы

```
#Реализация Singleton с помощью декортаора
def singleton(class_):
instances = {}
    def getinstance(*args, **kwargs):
        if class_ not in instances:
            instances[class_] = class_(*args, **kwargs)
        return instances[class_]
     return getinstance

@singleton
class MyClass(BaseClass):
    pass
```

Декораторы

- Подробнее у Лутца

Перегрузка операторов

Перегрузка операторов — один из способов реализации полиморфизма, когда мы можем задать свою реализацию какого-либо метода в своём классе.

Перегрузка операторов

Можно перегрузить любую «магическую» функцию, то есть ту, которая не вызываются напрямую, а вызываются встроенными функциями или операторами.

Например

__init__(self[, ...]) - конструктор

__str__(self) - вызывается функциями str, print и format. Возвращает строковое представление объекта.

__lt__(self, other) - x < y вызывает x.__lt__(y).

__le__(self, other) - x ≤ y вызывает x.__le__(y).

__eq__(self, other) - x == y вызывает x.__eq__(y).

__ne__(self, other) - x != y вызывает x.__ne__(y)

__gt__(self, other) - x > y вызывает x.__gt__(y).

__ge__(self, other) - x ≥ y вызывает x.__ge__(y).

Перегрузка операторов

Перегрузка арифметических операторов:

__add__(self, other) - сложение. x + y вызывает x.__add__(y).

__sub__(self, other) - вычитание (x - y).

__mul__(self, other) - умножение (x * y).

__truediv__(self, other) - деление (x / y).

__floordiv__(self, other) - целочисленное деление (x // y).

__mod__(self, other) - остаток от деления (x % y).

__iadd__(self, other) - +=.

__isub__(self, other) - -=.

__imul__(self, other) - *=.

__itruediv__(self, other) - /=.

__ifloordiv__(self, other) - //

Перегрузка операторов

Пример

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

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

    def __add__(self, other):
        return Vector2D(self.x + other.x, \
                        self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self
```

```
    def __sub__(self, other):
        return Vector2D(self.x - other.x,\
                        self.y - other.y)

    def __isub__(self, other):
        self.x -= other.x
        self.y -= other.y
        return self
```

```
a=Vector2D(2,3)
b=Vector2D(3,2)
print(a-b)
```

Ссылки по теме

- М. Лутц. Изучаем Python. 4-е издание. Гл. 33, 38-39.
- https://www.ibm.com/developerworks/ru/library/l-cpdecor/
- https://pythonworld.ru/osnovy/dekoratory.html
- http://toly.github.io/blog/2014/03/05/advanced-design-patterns-in-python/
- https://pythonworld.ru/osnovy/peregruzka-operatorov.html
- https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html

БЛАГОДАРЮ ЗА ВНИМАНИЕ

ВСЕ! КУРС ОКОНЧЕН!