Python

Урок 6. Декораторы, Исключения, Итераторы

Декораторы

Декораторы — обычные функции, которые принимают в качестве аргумента другую функцию.

```
>>> def decorator(func):
        def decorated(*args, **kwargs):
            print('before')
            result = func(*args, **kwargs)
            print('after')
            return result
        return decorated

>>> def s(a,b):
        return a+b

>>> s = decorator(s) # "оборачиваем" функцию
>>> s(1, 2)
before
3
after
```

Более удобный способ обернуть функцию через символ @:

```
@decorator
def m(a,b):
    print (a*b)
```

Запись вида

```
@f1
def func(): pass
```

эквивалентна

```
def func(): pass
func = f1(func)
```

Декораторов может быть несколько. В этом случае они «выполняются» сверху вниз.

```
@decorator
@timer
@pause
def func(x, y):
    return x + y
```

В декоратор можно передавать параметры:

```
def pause(t):
    def wrapper(f):
        def tmp(*args, **kwargs):
            time.sleep(t)
            return f(*args, **kwargs)
        return tmp

    return wrapper

@pause(2)
def func(x, y):
    return x + y
```

Запись вида:

```
@f1(123)
def func(): pass
```

эквивалентна

```
def func(): pass
func = f1(123)(func)
```

Использование декораторов в классах

Использование декораторов на методах классов ничем не отличается от использования декораторов на обычных функциях.

Для классов также есть предопределённые декораторы с именами staticmethod и classmethod. Они предназначены для задания статических методов и методов класса соответственно. Вот пример их использования:

```
class TestClass(object):
    @classmethod
    def f1(cls):
        print cls.__name__

    @staticmethod
    def f2():
        pass

class TestClass2(TestClass):
    pass

TestClass.f1() # печатает TestClass
TestClass2.f1() # печатает TestClass2

a = TestClass2()
a.f1() # печатает TestClass2
```

Статический метод (обёрнутый декоратором staticmethod) в принципе соответствует статическим методам в C++ или Java. А вот метод класса — это нечто более интересное. Первым аргументом такой метод получает класс (не экземпляр!), это происходит примерно так же, как с обычными методами, которые первым аргументом получают референс на экземпляр класса. В случае, когда метод класса вызывается на инстансе, первым параметром передаётся актуальный класс инстанса, это видно на примере выше: для порождённого класса передаётся именно порождённый класс.

Исключения

Ошибки бывают 2 типов: ошибки синтаксиса и исключения.

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

Даже если выражение синтаксически корректно, ошибка может возникнуть во

время выполнения. Такие ошибки называются "исключениями".

```
>>> 10*(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> 4+spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

```
>>> '2'+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Инструкции обработки исключений

Исключения — по сути, являются событиями, способными изменить ход выполнения программы. Исключения в языке Python возбуждаются автоматически, когда программный код допускает ошибку, а также могут возбуждаться и перехватываться самим программным кодом.

Обрабатываются исключения четырьмя инструкциями:

try/except
: Перехватывает исключения, возбужденные интерпретатором или вашим программным кодом, и выполняет восстановительные операции.

try/finally
: Выполняет заключительные операции независимо от того, возникло исключение или нет.

raise
: Дает возможность возбудить исключение программно.

assert
: Дает возможность возбудить исключение программно, при выполнении определенного условия.

Таким образом, если для вас нежелательно, чтобы программа завершалась, когда интерпретатор возбуждает исключение, достаточно просто перехватить его, обернув участок программы в инструкцию try.

Полный формат инструкции try:

```
try:
    <statements> # Сначала выполняются эти действия
except <name1>:
    <statements> # Запускается, если возникло исключение name1
except (name2, name3):
    <statements> # Запускается, если возникло любое
                 # из заданных исключений
except <name4> as <data>:
    <statements>  # Запускается в случае исключения name4
                  # и получает экземпляр исключения
except:
    <statements>  # Запускается для всех остальных исключений
else:
    <statements>  # Запускается, если в блоке try не возникло
 исключения
finally:
    <statements>  # Запускается в любом случае, возникло
 исключение или нет
```

Простой блок try/except

Возможно писать программы, обрабатывающие исключения:

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

Несколько except

В конструкции try может быть несколько except:

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Инструкция try/else

Назначение предложения else в инструкции try на первый взгляд не всегда очевидно для тех, кто только начинает осваивать язык Python. Тем не менее без этого предложения нет никакого другого способа узнать (не устанавливая и не проверяя флаги) – выполнение программы продолжилось потому, что исключение в блоке try не было возбуждено, или потому, что исключение было перехвачено и обработано:

```
try:
    ...выполняемый код...
except IndexError:
    ...обработка исключения...
# Программа оказалась здесь потому, что исключение было
# обработано или потому, что его не возникло?
```

Точно так же, как предложение else в операторах цикла делает причину выхода из цикла более очевидной, предложение else в инструкции try однозначно и очевидно сообщает о произошедшем:

```
try:
    ...выполняемый код...
except IndexError:
    ...обработка исключения...
else:
    ...исключение не было возбуждено...
```

Инструкции из else выполняются, если не возникло исключения:

```
filename = 'somefile.txt'

try:
    f = open(filename, 'r')
except IOError:
    print('cannot open', filename)
else:
    print(filename, 'has', len(f.readlines()), 'lines')
    f.close()
```

Возбуждение исключений

Исключения могут возбуждаться интерпретатором или самой программой и могут перехватываться или не перехватываться. Чтобы возбудить исключение вручную, достаточно просто выполнить инструкцию **raise**. Исключения, определяемые программой, перехватываются точно так же, как и встроенные исключения.

При самостоятельном возбуждении исключений можно использовать переменные:

```
try:
    raise Exception('spam', 'eggs')
except Exception as inst:
    print(type(inst))    # <class 'Exception'>
    print(inst.args)     # аргументы хранятся в .args
    print(inst)          # __str__ выводит .args
    x, y = inst.args     # распаковываем аргументы
    print('x =', x)
    print('y =', y)
```

Создание собственных исключений

Ислючения должны наследоваться от класса Exception или его потомков

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

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

Инструкция try/finally

```
try:
    <statements> # Выполнить эти действия первыми
finally:
    <statements> # Всегда выполнять этот блок кода при выходе
```

Инструкция finally всегда выполняется при выходе из блока try, не смотря на то, возникло исключение или нет.

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
divide(2, 1)
result is 2.0
executing finally clause

divide(2, 0)
division by zero!
executing finally clause
```

В реальных приложениях инструкцию finally удобно применять для освобождения ресурсов (таких как соединения по сети), не смотря на то, было ли успешным их использование или нет.

Документация по встроенным исключениям:
http://docs.python.org/3/library/exceptions.html#bltin-exceptions

Итераторы и генераторы

Итераторы

Когда вы создаёте список, вы можете считывать его элементы один за другим — это называется итерацией:

```
>>> mylist = [1, 2, 3]
>>> for i in mylist :
...    print(i)
1
2
3
```

Mylist является итерируемым объектом.

Всё, к чему можно применить конструкцию «for… in...», является итерируемым объектом: списки, строки, файлы… Это удобно, потому что можно считывать из них значения сколько потребуется — однако все значения хранятся в памяти, а это не всегда желательно, если у вас много значений.

Итератор
: Объект, представляющий поток данных. Последовательное ображещение к методу __**next**__() или передача его встроенной функции next() возвращает последовательно данные из потока.
Когда данных не остается — вызывается исключение **StopIteration**.
Итератор также должен иметь метод __**iter**__(), возвращающий сам итератор.

Питон поддерживает концепцию итерации содержимого контейнера. Она позволяют делать итерации по объектам, определенным пользовалем.

Контейнеру необхоимо определить один метод, чтобы обеспечить поддержку итерации:

container.__**iter**__()
: Возвращает итерируемый объект. Объект должен поддерживать протокол итератора, описанный ниже.

Сами итерируемые объекты должны следующие 2 метода, составляющие протокол итератора:

iterator.__**iter**__()

Возвращает сам объект итератора.

iterator.__**next**__()
Возвращает следующий элемент из контейнера. Если больше нет элементов — вызывается исключение StopIteration.

Генераторы

Генераторы это тоже итерируемые объекты, но прочитать их можно лишь один раз. Это связано с тем, что они не хранят значения в памяти, а генерируют их на лету:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator :
...     print(i)
0
1
4
```

Генератор
Функция, которая возвращает итератор. Она выглядит как обычная функция, за исключением того, что она содержит оператор **yield**, возвращающий серию значений, используемых в цикле или через функцию next(). Каждый оператор yield временно замораживает процесс, запоминая позицию выполнения (включая внутренние переменные).

```
# генератор чисел Фибоначчи
def fibonacci(max):
    a, b = 1, 2
    while a < max:
        yield a
        a, b = b, a+b
```