



Функции, функциональное программирование

Юрова Анна, группа 222

Парадигма программирования.

- — это совокупность идей и понятий, определяющая стиль написания программ.

Императивное программирование

- Программы представляют собой последовательность действий с условными и безусловными переходами.

Декларативное программирование

- Способ, описывающий, не как решить задачу, а что нужно получить. Описывается спецификация программы.
- Частный случай: функциональное программирование.

Функциональное программирование

- Парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании (в отличие от функций как подпрограмм в процедурном программировании).

Ключевой вопрос:

- **ЧТО** нужно вычислить, а не **КАК**.

Концепции.

- **Функции высших порядков**

функции, которые могут принимать в качестве аргументов и возвращать другие функции.

- **Чистые функции**

функция может управлять только выделенной для неё памятью, не модифицируя память вне своей области.

- **Ленивость**

функция не производит вычислений до тех пор, пока их результат не будет необходим в работе программы.

- **Рекурсия**

цикл организуется в виде рекурсии. переполнения стека можно избежать при помощи хвостовой рекурсии.

Модель вычисления без состояний.

● Императивная программа:

- на любом этапе исполнения имеет состояние (совокупность значений всех переменных).
- последовательность инструкций, описывающих **КАК** компьютер должен решать задачу, основываясь на состоянии, изменяемом шаг за шагом.

● Функциональная программа:

- ни целиком, ни частями СОСТОЯНИЯ не имеет.
- описывает, **ЧТО** должно быть вычислено. Является выражением, определенным в терминах заранее заданных функций и функций, заданных пользователем. Величина этого выражения – результат программы.

Особенности.

- чисто функциональная программа не может изменять уже имеющиеся у неё данные, а может лишь порождать новые путём копирования и/или расширения старых.
- в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: **ВЫХОДНЫЕ ДАННЫЕ ЗАВИСЯТ ТОЛЬКО ОТ ВХОДНЫХ.**

Сильные стороны.

- Повышение надёжности кода

за счёт чёткой структуризации и отсутствия необходимости отслеживания побочных эффектов.

- Удобство организации модульного тестирования

возможность протестировать каждую функцию в программе, просто вычислив её от различных наборов значений аргументов.

- Возможности оптимизации при компиляции

так как не задана последовательность выполнения.

- Возможности параллелизма.

всегда допустимо параллельное вычисление двух различных параметров . Порядок их вычисления не может оказать влияния на результат вызова.

Недостатки и как с ними бороться.

- Отсутствие присваиваний и замена их на порождение новых данных приводят к большим затратам памяти и вычислений при копировании.
- Сборщик мусора.
специальный код, который периодически освобождает память, удаляя объекты, которые уже не будут востребованы приложением

Python.

- Python поддерживает конструкции функционального программирования, которые можно сочетать с императивными.

Функции в Python.

```
def <name>(arg1, arg2, ..., argN) :  
    '''строка документации'''  
    <statements>
```

```
def <name>(arg1, arg2, ..., argN) :  
    ...  
    return <value>
```

- Тип возвращаемого значения определяется аргументами, которые передаются функции.
- Без аргументов **return** используется для выхода из функции без возвращаемого значения.
- В этом случае возвращается значение None.

Функции в Python.

- определение функции происходит во время выполнения.
- имя функции не является однозначно определенным. Важен только *объект* на который ссылается имя.

```
othername = func    #связывание объекта  
функции
```

с именем

```
othername ()        #вызов функции
```

Функции в Python.

1. `def func(x, y):`
2. `return x**2 + y**2`

```
func = lambda x, y: x**2 + y**2
```

Определение и использование на месте :

```
>>> (lambda x: x+2) (5)
```

```
7
```

Функции в Python. Аргументы.

1. `def func(x, y, z=7)`
2. `return x+y+z`
- 3.
4. `print func(1, y=3) # 1+3+7=11`

Области видимости.

- Имена, определяемые внутри инструкции `def`, видны **ТОЛЬКО** программному коду внутри инструкции `def`. К ним нельзя обратиться за пределами функции.
- Имена, определяемые внутри инструкции `def`, не вступают в конфликт с именами, находящимися за пределами инструкции `def`.

Области видимости.

1. #Глобальная область видимости.
2. `X = 99` # X и func определены в модуле:
3. глобальная область
4. `def func(Y):` # Y и Z определены в функции:
5. локальная область
6. #Локальная область видимости
7. `Z = X + Y` # X – глобальная переменная
8. `return Z`
- 9.
10. `func(1)` #result = 100

Области видимости. Инструкция *global*.

1. `y, z = 1, 2` #Глобальные переменные
2. в модуле.
3. `def all_global():`
4. `global x` #объявляется глобальной
5. для присваивания
6. `x = y + z`

Функции в Python.

- Функции в Python являются объектами первого класса, то есть, они могут употребляться в программе наравне с объектами других типов данных.

List comprehension.

(Списочные выражения)

1. `range(5)`

2. `#[0, 1, 2, 3, 4]`

3. `L = [a ** 2 for a in range(5)]`

4. `#[0, 1, 4, 9, 16]`

5. `L = [a**2 for a in range(5) if a % 2 == 1]`

6. `#[1, 9]`

7. `L = [a1 + a2 for a1 in ['a','b','c'] for a2 in ['x','y']]`

8. `#['ax', 'ay', 'bx', 'by', 'cx', 'cy']`

ФУНКЦИИ ВЫСШИХ ПОРЯДКОВ. Map().

1. `list1 = [1, 3, 5]`
2. `list2 = [1, 2, 3]`
3. `def inc(x):`
4. `return x+10`
5. `map(inc, list1)`
6. `# [11, 13, 15]`
7. `map(lambda x, y: x**y, list1, list2)`
8. `# [1, 9, 125]`
9. `[x * y for x, y in zip(list1, list2)]`
10. `# [1, 9, 125]`

Функции высших порядков. Filter()

```
list = [9, 1, -4, 3, 8]
filter(lambda x: x < 5, list)
#[1, -4, 3]
```

```
list = [9, 1, -4, 3, 8]
[x for x in list if x < 5]
# [1, -4, 3]
```

Функции высших порядков. Reduce()

```
list = [2, 3, 4, 5, 6]
```

```
reduce(lambda res, x: res*x, list, 1)
```

```
#720 = (((1*2)*3)*4)*5)*6
```

```
reduce(lambda res, x: res*x, [], 1)
```

```
#1
```

```
reduce(lambda res, x: [x]+res, [1, 2, 3, 4], [])
```

```
#[4, 3, 2, 1]
```

```
reversed([1, 2, 3, 4, 5])
```

```
#[5, 4, 3, 2, 1]
```

Функции высших порядков. Apply()

```
def f(x, y, z, a=None, b=None):  
    print x, y, z, a, b
```

```
apply(f, [1, 2, 3], {'a': 4, 'b': 5})
```

```
#1 2 3 4 5
```

```
f(*[1, 2, 3], **{'a': 4, 'b': 5})
```

```
1 2 3 4 5
```

Замыкания.

```
def make_adder(x):  
    def adder(n):  
        return x + n # захват переменной "x" из  
                       # внешнего контекста  
    return adder
```

То же самое, но через безымянные функции:

```
make_adder = lambda x: ( lambda n: ( x + n ) )  
f = make_adder(10)  
  
print f(7) # 17  
print f(-4) # 6
```

Замыкания. *Partial.*

```
from functools import partial  
def add(a, b):  
    return a + b
```

```
add1 = partial(add, b=1)
```

```
print add1(2)
```

```
#3
```


Итераторы.

```
it = enumerate(sorted("PYTHON"))  
    it.next()
```

```
# (0, 'H')
```

```
print list(it)
```

```
# [(1, 'N'), (2, 'O'), (3, 'P'), (4, 'T'), (5, 'Y')]
```

Итератор – объект, позволяющий программисту перебирать все элементы коллекции без учёта её особенностей реализации.

В модуле **itertools** есть функции для работы с итераторами, позволяющие кратко и эффективно выразить требуемые процессы обработки списков.

Генераторы.

- **генераторы** — функции, сохраняющие внутреннее состояние: значения локальных переменных и текущую инструкцию

```
def gen_fibonacci(n=100):  
    a, b = 0, 1  
    while a < n:  
        yield a #вместо return  
        a, b = b, a+b # напечатать все  
        числа Фибоначчи < 1000  
for i in gen_fibonacci(1000):  
    print i
```

Генераторные выражения.

- **генераторные выражения** — выражения, дающие в результате генератор
- *Генераторные выражения* позволяют сэкономить память там, где иначе требовалось бы использовать список с промежуточными результатами.

```
sum(i for i in xrange(1, 100) if i % 2 != 0)
```

```
#2500
```

Литература.

1. Марк Лутц. Изучаем Питон. 3-е издание.
2. http://ru.wikipedia.org/wiki/Функциональное_программирование_на_Питоне
3. http://ru.wikipedia.org/wiki/Функциональное_программирование
4. Г. Россум, Ф.Л.Дж.Дрейк, Д.С.Откидач «Язык программирования Python»
5. Журнал «Практика функционального программирования» <http://fprog.ru/>