Питон
Курс лекций
Лекция седьмая

© Зайцев Вадим Валерьевич, 2002–2010,
spider.vz@gmail.com

4.2 Понятие подпрограммы

В наше время существуют два принципиально разных подхода к реализации подпрограмм:

1. **Процедура** — это имеющая собственное имя часть программы, которая при вызове получает некоторые параметры и в соответствии с ними изменяет окружение, после чего возвращает управление в точку вызова. Процедура также может изменять собственные параметры (если их больше нуля). Такой тип подпрограммы широко используется в архаичных языках программирования (Фортран, Паскаль) и подчас (в том случае, если изменение окружения эквивалентно передаче данных через переменную, как в языке Форт) полностью равносилен следующему.

2. **Функция** — это имеющая имя часть программы, которая при вызове получает некоторые параметры и в соответствии с ними возвращает своё значение, не меняя окружение. Это определение куда ближе к математическому понятию функции.

В широко распространённых языках программирования эти подходы присутствуют, будучи перемешаны в том или ином соотношении. Паскаль, например, использует термины *процедура* и *функция*, но функции в нём могут изменять окружение. В Си++ любая подпрограмма является функцией, но может возвращать значение типа `void` (пусто), превращаясь таким образом в процедуру. В Лиспе процедур мало, все они стандартны (например, процедуры вывода на экран) и называются *псевдофункциями*.

Существуют, безусловно, языки программирования, идущие в следовании тому или иному подходу дальше, чем этот подход планировал. Например, в языке ассемблера совсем не обязательно возвращать управление из процедуры, либо это можно сделать в место, отличное от точки вызова.

В чисто функциональных языках (Лисп, МЛ, КЛОС) функции не нужно (хотя и можно) иметь имя.

Также понятно, что хорошо бы остановится где-то посередине, имя возможность применять подходы сообразно стоящей задаче. Одну подпрограмму, огранизующую соединение с сервером для дальнейшего обмена данными, логично было бы организовать процедурой, а другую подпрограмму, вычисляющую синус, — функцией.

В питоне процедуры и функции определяются весьма сходными конструкциями, но используются, конечно, по-разному. Вот так определяется процедура:

```
def␣becool(boy):
␣print␣boy,'is␣cool'
```

А так используется:

```
becool('Python')
```

Вот так определяется функция:

```
def␣logn(n,x)
␣return␣log(x)/log(n)
```

а так используется:

```
print logn(20,x)+sin(x)
```

Как видно из определений, ключевое отличие состоит в слове return — это и есть то самое *возвращение значения*, о котором уже была речь. Его формат таков:

```
return <значение>
```

Можно возвращать и несколько значений, перечисляя их через запятую — в этом случае питон, не изменяя самому себе, возвращает единым значением неявно создаваемый кортеж. Такую функцию можно использовать двояко:

```
def powers(x):
 return x*x,x*x*x,x*x*x*x
X=powers(2)
X2,X3,X4=powers(3)
```

В первом случае в `X` загружается целиком весь кортеж, во втором же — он декомпозиционируется и распадается на три элемента. При этом используются обычные правила присваивания кортежей, рассмотренные нами ранее.

Питон позволяет пользоваться подпрограммами, меняющими свою сущность от запуска к запуску. Например, вот так:

```
def br(a):
 if (a):
  return '('+str(a)+')'
 else:
  print "Error in br("+`a`+')'
```

При использовании функции как процедуры в программном режиме её значение теряется, а в интерактивном — выдаётся на экран. При использовании процедуры как функции считается, что она автоматически возвращает значение `None`.

Если вы хотите стабильно использовать вашу подпрограмму как функцию, позаботьтесь о том, чтобы при любом прохождении через её тело встречался только один return.

Следует твёрдо помнить, что return кроме возвращения значения прерывает выполнение функции (и возвращает управление в точку вызова) и производить все необходимые вычисления до него. Экстренный возврат из процедуры может быть записан как return без параметров:

```
return
```

Рассмотрев оператор `def`, мы затронули один важный момент, без разбора которого было бы немыслимо идти дальше.

4.3 Область действия имён переменных

Что будет, если в программе объявить некую переменную, а потом внутри функции попробуем ей воспользоваться? Получится у нас изменить её значение? Правильный ответ: просто так не получится, но можно, если постараться.

Под термином *область действия имён переменных* мы будем понимать область видимости используемых переменных. В более ранних языках программирования, часть из которых уже канула в Лету, а часть каким-то образом зацепилась за действительность, глобальные переменные были единственным средством создания переменных. Если даже переменная создавалась внутри функции, это просто была ещё одна глобальная переменная. Позже появилась концепция локальной переменной, не видной снаружи. Глобальные переменные всё же могли быть как прочитаны, так и изменены любой функцией. Отношение классиков теории программирования к этому вопросу не было единогласным. Эдсгер Дейкстра считал использование глобальных переменных в подпрограммах одним из самых ужасных нарушений дисциплины программирования, а Альфред Ахо при создании компиляторов не только допускал глобальные переменные как средство передачи информации от подпрограммы к подпрограмме, но и, можно сказать, пропагандировал его своими исходниками. Такие споры связаны с тем, что использующая глобальные переменные подпрограмма не является замкнутой системой, и при разных запусках с одинаковыми параметрами может возвращать разные результаты. С математической точки зрения, это превращает язык программирования в язык, порождаемый контекстно-зависимой грамматикой. Мы не будем вдаваться в лингвистические теории, но переход от контекстно-свободных грамматик к контекстно-зависимым существенно усложняет работу проектировщику компилятора.

Питон в этом плане более прогрессивен. В каждый момент выполнения программы (или ввода инструкций в интерактивном режиме) в питоне существуют две области действия имён переменных: глобальная и локальная. Первая относится к программе в целом, вторая — к текущей подобласти: телу функции, содержанию объекта, и т.д. Без дополнительных телодвижений изнутри функции глобальные переменные **не видны**. Например, после выполнения следующей функции:

```
x=1
def change(): x=-1
change()
```

значение глобальной переменной `x` не изменится. Вместо этого в локальной области будет создана новая переменная, имя которой совпадёт с именем глобальной переменной. Если же попытаться проверить значение глобальной переменной до попытки изменения её значения, будет выдано значение именно глобальной:

```
x=1
def trytoget(): print x
trytoget()
```

Таким образом, чтение глобальных переменных не считается питоном нарушением стиля, а запись в них данных — считается. Но, используя оператор `global`, можно обойти и это ограничение. Так,

```
x=1
def incr():
 x+=1
incr()
```

вызовет ошибку «обращение к локальной переменной до первого присваивания» (*local variable 'x' referenced before assignment*), а:

```
x=1
def incr():
 global x
 x+=1
incr()
```

будет работать. При определении вложенных функций локальная область не становится глобальной для подфункции, так что локальные области вложенных друг в друга подпрограмм не коррелируют.

Функции в питоне являются полноправными типами данных, поэтому их можно присваивать друг другу:

```
def a(x,y):return x+y+2
b=a
```

После чего `a` и `b` будут указывать на одну и ту же функцию, и она будет существовать до тех пор, пока не выполнить оператор `del` по отношению и к `a`, и к `b`. Такие правила существования справедливы для всех объектов, о чём мы узнаем уже через несколько лекций.

4.4 Особые приёмы работы с функциями

Таких приёмов насчитывается пять штук. Это именованные параметры, необязательные для указания параметры, параметры с неизвестной длиной и параметры с неизвестными именами. Ещё один приём, лямбда-исчисление, будет рассмотрен нами отдельно, так как он стоит этого.

- **Именованные параметры** — это механизм, позволяющий при вызове подпрограммы менять местами её параметры, зная их имена. Например, мы объявили функцию:

```
def qualify(author,name,quality):
 print author+"'s", name, 'is a', quality, 'book.'
```

Её можно вызывать так:

```
qualify('G.Booch', 'OOA&D with Applications', 'good')
```

А можно — так:

```
qualify(quality='good', name='OOA&D with Applications',
author='G.Booch')
```

Результат будет одинаковым: `G.Booch's OOA&D with Applications is a very good book.`.

Можно комбинировать этот подход с обычным перечислением параметров по порядку, но при этом именованные параметры должно стоять после всех обычных:

`qualify(author='G.Booch', 'OOA&D', 'good')` — нельзя

`qualify('G.Booch', quality='good',name='OOA&D')` — можно

- **Необязательные для указания параметры** — это механизм, позволяющий при вызове подпрограммы указывать значения только критических параметров, без которых она работать не будет, имея, тем не менее, возможность указания их при желании. Это реализуется путём указания значений по умолчанию для всех необязательных параметров при определении функции:

```
def qualify(author,name="book quality="bad"):
 print author+\s name, 'is a', quality, 'book.'
```

Теперь наша функция может принимать от одного до трёх параметров, причём, воспользовавшись предыдущим приёмом, можно задать качество книги без указания названия:

```
qualify('G.Booch',qualify='very good')
```

Рекомендуется при использовании необязательных для указания параметров присваивать им значения только неизменяемых типов (строк, чисел, кортежей), потому что используемое по умолчанию значение присваивается только один раз. Например, у вас есть функция:

```
def addel(n,x=[]):
 x.append(n)
 print x
```

Теперь попробуем запустить её несколько раз:

```
addel(2,[3,4]) # даёт [3,4,2] - правильно
addel(1) # даёт [1] - пока правильно
addel(2) # даёт [1,2] - неправильно!
addel(3) # даёт [1,2,3] - уж совсем неправильно!
```

Вместо этого следует пользоваться проверкой внутри тела функции, менее удобной, но работающей правильно.

- **Параметры неизвестной длины** — это механизм, позволяющий реализовывать подпрограммы, полностью инвариантные относительно количества предоставляемых им параметров. Записывается это так:

```
def qualifyAuthors(*several):
 for one in several:
  qualify(one)
```

При этом, как вы поняли, все параметры питон собирает в один кортеж и его отдаёт в качестве указанной переменной. Кортеж, безусловно, может быть пуст.

- **Непредусмотреные параметры** — это механизм, позволяющий реализовывать подпрограммы, стойкие к лишним параметрам и инвариантные относительно их имён. При этом ещё однам именем обозначается словарь, который либо пуст, если все параметры предусмотрены, либо состоит из пар название-значение. Синтаксис таков:

```
def qualify(author,name="book",quality="bad",**aux):
 print author+"'s", name, 'is a', quality,
 if aux:
  print 'book,',
  for a in aux.keys():
   print a,'works in',aux[a],',',
  print 'as one can read.'
 else:
  print 'book.'
```

Тогда нашей весьма выросшей процедурой можно пользоваться вот так:

```
qualify('G.Booch','OOA&D','very good')
qualify('A.V.Aho,R.Sethi,J.D.Ullman', quality='perfect',
name='Dragon Book', Aho='AT&T Bell Labs',Sethi='AT&T Bell
Labs',Ullman='Stanford University')
```

что выдаст:

```
G.Booch's OOA&D is a very good book.
A.V.Aho,R.Sethi,J.D.Ullman's Dragon Book is a perfect book,
Aho works in AT&T Bell Labs, Sethi works in AT&T Bell Labs,
Ullman works in Stanford University, as one can read.
```