Python for Informatics
Автор: Dr. Charles R. Severance
Содержание курса: http://pycode.ru/edu/why-python/

Глава 4. Функции

4.1. Вызов функции

В контексте программирования, функцией (function) называется последовательность инструкций, которая выполняет вычисления. Когда вы задаете функцию, то задаете имя и последовательность инструкций. Позже вы сможете вызвать функцию, обратившись к ней по имени. Мы уже встречали примеры вызова функции (function call):

```
>>> type(32)
<type 'int'>
```

Функция носит имя *type*. Выражение в скобках называется аргументом (argument) функции. Аргумент – это значение или переменная, которую мы передаем в функцию в качестве входного параметра. Результатом функции type является тип аргумента.

Проще говоря, функция «берет» аргумент и «возвращает» результат. Результат называется возвращаемым значением (return value).

4.2. Встроенные функции

Python предоставляет несколько важных (built-in) встроенных функций, которые мы можем использовать. Создатели Python написали множество функций, которые решают часто встречающиеся проблемы и включил эти функции в состав Python.

Функции *max* и *min* возвращают соответственно наибольшее и наименьшее значения в списке:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
```

Функция *max* говорит нам о «наибольшем символе» в строке (который функция нам вернула как «w»), функция *min* показывает наименьший символ, которым является пробел.

Другая очень распространенная встроенная функция - *len*, которая говорит нам, сколько элементов содержится в аргументе. Если аргументом функции *len* является строка, то она возвращает количество символов в строке.

```
>>> len('Hello world')

11

>>>
```

Эти функции не исчерпываются просмотром строк, они могут оперировать любым множеством значений, как мы увидим в следующих главах.

Вы должны обращаться с именами встроенных функций, как с зарезервированными словами (т.е. не используйте *max* в качестве имени переменной).

4.3. Функции, преобразующие типы

Python также предоставляет встроенные функции, которые преобразуют значения из одного типа в другой. Функция *int* берет любое значение и преобразует его в целое число, если это возможно:

```
>>> int('32')

32

>>> int('Hello')

ValueError: invalid literal for int(): Hello
```

Функция *int* может преобразовать число с плавающей точкой в целое, но это не округление, а отсечение дробной части:

```
>>> int(3.99999)

3

>>> int(-2.3)

-2
```

Функция *float* преобразует целое и строковое в число с плавающей точкой:

```
>>> float(32)
```

```
32.0
>>> float('3.14159')
3.14159
```

В завершении, функция str преобразует входные аргументы в строки:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4. Случайные числа

Получая одинаковые входные значения, большинство программ каждый раз генерируют одни и те же выходные значения, такие программы можно назвать детерминированными (deterministic). Детерминизм обычно является полезной вещью, но для некоторых приложений необходима непредсказуемость в поведении. Хорошим примером таких приложений являются игры.

Создание действительно недетерминированных программ – дело не простое, но есть пути создания программ, которые похожи на недетерминированные. Одним из таких способов являются алгоритмы (algorithms) генерации псевдослучайных (pseudorandom) чисел. Псевдослучайные числа не настоящие случайные, т.к. они генерируются на основании детерминированных вычислений, но с виду их не отличить от случайных.

Модуль *random* предоставляет функции, которые генерируют псевдослучайные числа (которые далее мы будем называть «случайными»).

Функция *random* возвращает случайное число с плавающей точкой в интервале от 0.0 до 1.0 (включая 0.0 и не включая 1.0). Каждый раз, когда вызывается *random*, вы получаете следующее число в длинной последовательности. Для примера рассмотрим цикл:

```
import random
for i in range(10):
     x = random.random()
     print x
```

Результатом работы программы будет список из следующих 10 случайных чисел в интервале от 0.0 до 1.0.

```
0.301927091705

0.513787075867

0.319470430881

0.285145917252

0.839069045123

0.322027080731

0.550722110248

0.366591677812

0.396981483964

0.838116437404
```

Функция *random* является одной из функций, которая работает со случайными числами. Функция *randint* принимает параметры *low* и *high*, и возвращает целочисленное значение в интервале от *low* до *high* (включая оба).

```
>>> random.randint(5, 10)

5

>>> random.randint(5, 10)

9
```

Для выбора случайного элемента из последовательности, можно воспользоваться функцией *choice*:

```
 >>> t = [1, 2, 3]

>>> random.choice(t)

2

>>> random.choice(t)

3
```

Модуль *random* также включает функции, которые генерируют значения от непрерывных распределений, включая Гаусса, экспоненциального, гамма и некоторых других.

4.5. Математические функции

В Python содержится математический модуль (module), который предоставляет большинство популярных математических функций. Перед тем, как его использовать, нам необходимо его импортировать:

```
>>> import math
```

Эта инструкция создает модульный объект (module object), который называется *math*. Если вы выведите на экран модульный объект, то получите некоторую информацию о нем:

```
>>> print math
<module 'math' (built-in)>
```

Модульный объект содержит функции и переменные, определенные в объекте. Для получения доступа к одной из этих функций, вам необходимо задать имя модуля и имя функции, разделенные точкой (period). Этот формат называется точечной нотацией (dot notation).

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)


>>> radians = 0.7
>>> height = math.sin(radians)
```

В первом примере вычисляется логарифм по основанию 10 отношения «сигнал-шум». Модуль *math* также предоставляет функцию *log*, которая вычисляет логарифмы по основанию *e*.

Во втором примере вычисляется синус *radians*. Имя переменной подсказывает, что *sin* и другие тригонометрические функции (*cos*, *tg* и т.д.) принимают аргументы в радианах. Для перевода градусов в радианы, разделим на 360 и умножим на 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

Выражение *math.pi* получает значение переменной *pi* из модуля *math*. Значением этой переменной является приближенное π с точностью до 15 знаков.

4.6. Добавление новых функций

Ранее мы использовали встроенные в Python функции. Теперь мы рассмотрим, как добавлять новые функции. Определение функции (function definition) задает имя новой функции и последовательность инструкций, которые выполняются, когда функция вызывается. Как только мы определили функцию, мы можем многократно ее использовать.

Например,

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'
```

def – это ключевое слово, которое показывает, что далее следует определение функции. Имя функции - *print_lyrics*. Правила наименования функций такие же, как для переменных: возможны буквы, числа и некоторые знаки пунктуации, но первая буква не может быть числом. Вы не можете использовать ключевые (зарезервированные) слова в качестве имен функций. Имена функций и переменных не должны совпадать.

Пустые скобки после имени указывают на то, что функция не принимает аргументов. Позже мы рассмотрим функции, которые принимают входные аргументы.

Первая строка определения функции называется заголовком (header), оставшаяся часть – телом (body) функции. Заголовок заканчивается двоеточием, тело функции имеет отступ. По договоренности, отступ всегда является четырьмя пробелами. Тело функции может содержать любое количество инструкций.

Строки, которые мы выводим на экран, заключены в двойные кавычки. Одиночные и двойные кавычки взаимозаменяемы, большинство людей используют одиночные кавычки, за исключением тех случаев, когда одиночные кавычки встречаются внутри строки.

Если вы будете набирать функцию в интерактивном режиме, то Python для тела функции сделает отступы, а в конце необходимо будет ввести пустую строку.

Определение функции создает переменную с таким же именем.

```
>>> print print_lyrics

<function print_lyrics at 0xb7e99e9c>

>>> print type(print_lyrics)

<type 'function'>
```

Значением *print_lyrics* является функциональный объект (function object), который имеет тип *'function'*.

Синтаксис вызова новой функции похож на вызов встроенной функции:

```
>>> print_lyrics()

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.
```

Однажды определив функцию, вы можете использовать ее внутри других функций. Для примера повторим строку-припев, воспользовавшись новой функцией:

```
def repeat_lyrics():

     print_lyrics()

     print_lyrics()
```

Затем вызовем *repeat_lyrics*:

```
>>> repeat_lyrics()

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.
```

4.7. Определение и использование

Если собрать вместе весь код из предыдущего параграфа, то получим следующее:

```
def print_lyrics():

     print "I'm a lumberjack, and I'm okay."

     print 'I sleep all night and I work all day.'
```

```
def repeat_lyrics():
     print_lyrics()
     print_lyrics()
repeat_lyrics()
```

Эта программа содержит два определения функций: *print_lyrics* и *repeat_lyrics*. Определения функций получают управление подобно другим инструкциям, результатом является создание функционального объект.

Инструкции внутри функции не получат управления, пока функция не будет вызвана.

Как и следовало ожидать, вы должны предварительно создать функцию, прежде чем сможете ее выполнить. Иными словами, *определение функции должно быть выполнено перед ее первым запуском*.

4.8. Поток исполнения

Для того чтобы убедиться, что функция определяется до ее первого использования, вы должны знать порядок, в котором выполняются инструкции, он называется потоком исполнения (flow of execution).

Исполнение обычно начинается с первой инструкции программы. Инструкции выполняются по одной сверху вниз.

Определение функций не изменяют порядок исполнения программы, но запомните, что инструкции внутри функции не исполняются до вызова функции.

Вызов функции подобен обходному пути в потоке исполнения. Вместо того чтобы перейти к следующей инструкции, поток переходит в тело функции, выполняет все инструкции внутри тела, и затем возвращается обратно в то место, которое он покинул в момент вызова функции.

Это звучит достаточно просто, пока вы не вспомните, что одна функция может вызывать другую. В то время как в середине одной функции, программа может выполнять инструкции из другой функции. Но во время выполнения этой новой функции, программа может выполнять еще одну функцию!

К счастью, Python следит, где находится, так что всякий раз, когда функция завершается, программа переходит обратно, в место вызова функции. Программа завершится, когда дойдет до конца программы.

В чем мораль этой неприглядной истории? Когда вы читаете программу, вам не всегда хочется читать сверху вниз. Иногда это имеет смысл, если вы следуете за потоком исполнения.

4.9. Параметры и аргументы

Некоторые из встроенных функций требуют передачи входных аргументов. Например, когда вы вызываете *math.sin*, вы передаете число в качестве входного аргумента. Некоторые функции принимают больше одного аргумента, например, в *math.pow* передается два аргумента: основание и степень.

Внутри функции, аргументы, присвоенные переменным, называются параметрами (parameters). Пример определения функции, которой передается аргумент:

```
def print_twice(bruce):
     print bruce
     print bruce
```

Эта функция присваивает аргумент параметру с именем *bruce*. Когда вызывается функция, она дважды выводит на экран значение параметра (что бы это ни было).

Эта функция работает с любыми значениями, которые могут быть выведены на экран.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

Для построения функций, определяемых пользователем, используются те же правила, что и для встроенных функций, поэтому мы можем использовать любое выражение в качестве аргумента для *print_twice*:

```
>>> print_twice('Spam '*4)

Spam Spam Spam Spam

Spam Spam Spam Spam

>>> print_twice(math.cos(math.pi))

-1.0

-1.0
```

Аргумент вычисляется перед вызовом функции, поэтому в примере выражения *'Spam '*4* и *math.cos(math.pi)* вычисляются только один раз.

Вы можете использовать переменные в качестве аргументов:

```
>>> michael = 'Eric, the half a bee.'

>>> print_twice(michael)

Eric, the half a bee.

Eric, the half a bee.
```

Имя переменной, которое мы передаем в качестве аргумента (*michael*), не имеет ничего общего с параметром (*bruce*). Не важно, какое значение было передано в вызывающую функцию; здесь, в *print_twice*, мы называем все *bruce*.

4.10. Плодотворные (fruitful) функции и void-функции

Некоторые из функций, которые мы используем, такие, как математические функции, возвращают результаты (приносят плоды); за неимением лучшего названия, я называю их плодотворными функциями (fruitful functions). Другие функции, подобные *print_twice*, выполняют некоторые действия, но не возвращают значение. Они называются void-функциями.

Когда вы вызываете плодотворную функцию, вы почти всегда хотите сделать что-то с результатом, например, вы можете присвоить его переменной или использовать его как часть выражения:

```
x = math.cos(radians)

golden = (math.sqrt(5) + 1) / 2
```

Когда вы вызываете функцию в интерактивном режиме, Python отображает результат:

```
>>> math.sqrt(5)

2.2360679774997898
```

Но если вы вызываете плодотворную функцию в скрипте и не сохраняете результат выполнения функции в переменной, то возвращаемое значение растворится в тумане!

```
math.sqrt(5)
```

Этот скрипт вычисляет квадратный корень из 5, но так как он не сохраняет результат в переменной и не отображает результат, то это не очень полезно.

Void-функции могут отображать что-то на экране или производить другие действия, но они не возвращают значение. Если вы попытаетесь присвоить результат выполнения такой функции переменной, то получите специальное значение, называемое *None*.

```
>>> result = print_twice('Bing')

Bing

Bing

>>> print result

None
```

Значение *None* – это не тоже самое, что строка *'None'*. Это специальное значение, которое имеет собственный тип:

```
>>> print type(None)

<type 'NoneType'>
```

Для возврата результата из функции, мы используем инструкцию *return* в функции. Для примера, мы можем создать простую функцию с именем *addtwo*, которая складывает два числа и возвращает результат.

```
def addtwo(a, b):

     added = a + b

     return added

x = addtwo(3, 5)
```

```
print x
```

Когда скрипт выполнится, инструкция *print* напечатает «8». Внутри функции параметры *a* и *b,* в момент выполнения, будут содержать значения 3 и 5 соответственно. Функция подсчитывает сумму двух чисел и помещает ее в локальную переменную *added*. Затем с помощью инструкции *return* результат вычисления возвращается в вызываемый код, как результат выполнения функции. Далее результат присваивается переменной *x* и выводится на экран.

4.11. Зачем нужны функции?

Есть несколько причин, на основании которых программу стоит разбивать на функции.

- Создание новой функции предоставляет вам возможность присвоить имя группе инструкций. Это позволит упростить чтение, понимание и отладку программы.
- Функции позволяют сократить код программы, благодаря ликвидации повторяющихся участков кода. Позже, вы сможете вносить коррективы только в одном месте.
- Разбиение длинной программы на функции позволяет одновременной отлаживать отдельные части, а затем собрать их в единое целое.
- Хорошо спроектированная функция может использоваться во множестве программ. Однажды написав и отладив функцию, вы можете использовать ее множество раз.

4.13. Словарь

Алгоритм (algorithm): обобщенный подход к решению категории проблем.

Аргумент (argument): значение, передаваемое функции, в момент ее вызова. Это значение присваивается соответствующему параметру в функции.

Тело функции (body): последовательность инструкций внутри определения функции.

Детерминистический (deterministic): относится к программе, которая выполняет одинаковые действия при одинаковых входных значениях.

Точечная нотация (dot notation): синтаксис для вызова функции из другого модуля.

Поток исполнения (flow of execution): порядок, в котором выполняются инструкции во время работы программы.

Плодотворная функция (fruitful function): функция, которая возвращает значение.

Функция (function): это именованная последовательность инструкций, которая выполняет некоторые полезные действия. Функции могут иметь или не иметь аргументы, могут возвращать или не возвращать результат.

Вызов функции (function call): инструкция, которая выполняет функцию. Она содержит имя функции и список аргументов.

Определение функции (function definition): инструкция, которая создает новую функцию; задает имя функции, параметры и инструкции для выполнения.

Функциональный объект (function object): значение, созданное определением функции. Имя функции является переменной, которая ссылается на функциональный объект.

Заголовок функции (header): первая строка в определении функции.

Инструкция импорта (import statement): инструкция, которая читает файл модуля и создает модульный объект.

Модульный объект (module object): значение, которое создает инструкция импорта, для предоставления доступа к данным и коду, определенном в модуле.

Параметр (parameter): имя, используемое внутри функции, которое ссылается на передающееся в качестве аргумента значение.

Псевдослучайный (pseudorandom): относится к последовательности чисел, которые похожи на случайные, но генерируются детерминированной программой.

Возвращаемое значение (return value): результат выполнения функции.

Void-функция (void function): функция, которая не возвращает результирующего значения.

4.14. Упражнения

1. Что называется функцией?
2. Приведите примеры встроенных функций.
3. Как в Python добавлять свои функции?
4. Чем отличаются параметры от аргументов функции?
5. Зачем нужны функции?