Python functions

Reuven M. Lerner, PhD
reuven@lerner.co.il

1

Calling functions

- We call functions in Python with ()
- No parentheses — no function call!

```
x = len('abc')
type(x)
int

x = len
type(x)
builtin_function_or_method
```

2

Defining functions

```
def myfunc():
    print "Hello"

myfunc()   # Prints “Hello”
```

Docstrings

```
def myfunc():
    "Hello function"
    print "Hello"


help(myfunc)  # Shows docstring
print myfunc.__doc__
```

Attributes

- When we ask for myfunc.__doc__, we are looking at the attributes of myfunc
- Every object in Python has attributes
  - Some data, some functions
  - (Although functions are data!)
- Get the attributes of an object with dir()

5

Functions are objects

```
x = myfunc

type(x)

  function

x()

  Hello
```

6

Function parameters

- What if we invoke myfunc with a parameter?

```
myfunc(1)



Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: myfunc() takes no arguments (1
given)
```

7

Arity

- Python knows the arity (number of parameters) of every defined function
- Every function can be defined once, and only once, always taking the same number of params
- You thus cannot define both hello() and hello(name) — the last one defined has priority

8

Add a parameter

```
def myfunc(name):
    "Prints 'hello'"
    print "Hello, {}".format(name)


myfunc('Reuven')
  Hello, Reuven
```

9

Default values

- Must come at the end of the parameter list (relaxed in Python 3)
- May be assigned to any value
- Parameters with default values may be passed, or not, with any name

Explicit parameters

```
def hello(first="FirstName", last="LastName"):
     print "Hello,{} {}".format(first, last)

hello()
  Hello, FirstName LastName
hello('myfirst', 'mylast')
  Hello, myfirst mylast
hello(last='mylast')
  Hello, FirstName mylast
hello(last='mylast', first='myfirst')
  Hello, myfirst mylast
```

Default values

```
def myfunc(name='Reuven'):
    "Prints 'hello'"
    print "Hello, {}".format(name)


myfunc()
    Hello, Reuven
```

12

Flexible parameters

- Python offers two special parameters, which must come at the end (in Python 2)
- *args turns all unmatched parameters into a tuple
- **kwargs turns unmatched key-value pairs into a dict
- These names are traditional, not required

* (“splat”) operator

- In a parameter list, *args means that the “args” parameter will be a tuple of zero or more values
- When invoking a function, *args transforms a list to a tuple of parameters

*args example

```
def test_var_args(farg, *args):

    print "formal arg:", farg

    for arg in args:

        print "another arg:", arg
```

15

Invoking with *args

```
test_var_args("Hello")

test_var_args("Hello", 1)

test_var_args("Hello", 1, 2, 3)
```

**kwargs

- Parameters are passed as name=value
- These name-value pairs are turned into a dictionary (kwargs)
- name becomes a string, value is whatever type you pass
- This gives you infinite flexibility in accepting parameters

**kwargs example

```
def test_var_kwargs(farg, **kwargs):

    print "formal arg:", farg

    for key in kwargs:

        print "arg: {}:{}".format(key, kwargs[key])
```

Invoking with **kwargs

```
test_var_kwargs("Hello")

test_var_kwargs("Hello",
                a="abc",
                b="def")
```

Return values

- You don't have to declare a return value; a function may simply return one.
- A function that fails to return any value actually returns None.
- A function may return any Python object — a number, string, list, tuple, dictionary, object, or function. (Yes, you may return a function!)

Returning example

```
def return_stuff(var):
    return [1, 2,
            {'a':1, 'b':2},
            'string']
```

Multiple return values

- If you return a sequence, it can be assigned to a single variable
- It can also be assigned to multiple variables, each of which gets one element of the sequence

By reference? By value?

- Neither!
- Parameters are passed by reference
- Assigning to a parameter never changes the parameter. It creates a local variable.
- Invoking a method on a parameter, if it is mutable, will change the object in both the function and in the caller’s scope.

Scoping

- Functions introduce the idea of scoping — where variables do and don't exist
- LEGB rule for scoping resolution: Local, Enclosing functions, Global, Built-in
- Loops and conditionals don't open a new variable scope!

Basic scoping

- Variables in functions are local to the function
- Assignment in a function creates a new local variable, masking global/builtins
- Retrieval in a function gets the global variable (or builtin)

Example

```
>> x = 100

>> def foo():

   x = 222



>> foo()

>> x

  100
```

“global” keyword

- If you name a variable with the “global” keyword, then assigning to that variable wil affect the global, rather than create a local variable

“global” example

```
>> x = 100
>> def foo():
   global x
   x = 222
>> foo()
>> x
  222
```

28

Hoisting

- If a local variable is defined in a function, then all references to it are seen as local
- This includes references before the variable's actual definition!

Hoisting error

```
>> x = 100
>> def foo():
   print x
   x = 222
>> foo()       # Error!
```

30