

Python Functions

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

October 5, 2011

Outline

- 1 Introduction
- 2 Writing functions
- 3 Formal arguments
- 4 Variable scope
- 5 Returning functions
- 6 Functional programming

Outline

- 1 Introduction
- 2 Writing functions
- 3 Formal arguments
- 4 Variable scope
- 5 Returning functions
- 6 Functional programming

Functions

Description

- Functions are the structured or procedural programming way of organizing the logic in your programs.
- Functions: Return a value back to their callers.
- Procedures: Functions that do not return a value.
(Python procedures are implied functions because the interpreter implicitly return a default value of `None`.)

A first procedure

Hello world!

```
In [1]: def hello():  
...:     print "hello world!"  
...:
```

```
In [2]: result = hello()  
hello world!
```

```
In [3]: result
```

```
In [4]: print result  
-----> print(result)  
None
```

```
In [5]: type(result)  
Out[5]: <type 'NoneType'>
```

A first function

Double anything

```
In [1]: def double(arg):  
...:     return arg * 2  
...:
```

```
In [2]: print double(3)  
-----> print(double(3))  
6
```

```
In [3]: print double("abc")  
-----> print(double("abc"))  
abcbcb
```

```
In [4]: print double([1, 2, 3])  
-----> print(double([1, 2, 3]))  
[1, 2, 3, 1, 2, 3]
```

```
In [5]:
```

(Creating the illusion of) Returning multiple items

Example

```
In [1]: def returning_multiple_items():  
...:     return 1, 'a', [1, 2, 3]  
...:  
  
In [2]: result = returning_multiple_items()  
  
In [3]: print result  
-----> print(result)  
(1, 'a', [1, 2, 3])  
  
In [4]: type(result)  
Out[4]: <type 'tuple'>  
  
In [5]: n, c, l = returning_multiple_items()  
  
In [6]: print n, c, l  
-----> print(n, c, l)  
(1, 'a', [1, 2, 3])
```

Arguments

Example

```
In [1]: def connection(host, port):  
...:     # do something ...  
...:     print "host: %s, port: %s" % (host, port)  
...:
```

```
In [2]: connection("monge.univ-mlv.fr", 8080)  
host: monge.univ-mlv.fr, port: 8080
```

```
In [3]: connection(port = 8080, host = "monge.univ-mlv.fr")  
host: monge.univ-mlv.fr, port: 8080
```

```
In [4]: connection()
```

```
TypeError      Traceback (most recent call last)
```

```
.../func/src/<ipython console> in <module>()
```

```
TypeError: connection() takes exactly 2 arguments (0 given)
```

Recursive functions

Example

```
In [1]: def fact(n):
...:     if n == 0:
...:         return 1
...:     else:
...:         return n * fact(n-1)
...:
```

```
In [2]: fact(10)
```

```
Out[2]: 3628800
```

```
In [3]: fact(100)
```

```
Out[3]: 93326215443944152681699238856266700490715968264381621468592963  
        89521759999322991560894146397615651828625369792082722375825118  
        5210916864000000000000000000000000L
```

Recursive functions

Example

```
In [1]: def fact(n):
...:     def fact_aux(n, acc):
...:         if n == 1:
...:             return acc
...:         else:
...:             return fact_aux(n - 1, acc * n)
...:     return fact_aux(n, 1)
...:
```

```
In [2]: fact(10)
```

Out[2]: 3628800

```
In [3]: fact(100)
```

```
Out[3]: 93326215443944152681699238856266700490715968264381621468592963  
        89521759999322991560894146397615651828625369792082722375825118  
        5210916864000000000000000000000000L
```

Recursive functions

Example

```
In [1]: def fact(n):  
...:     def fact_aux(n, acc):  
...:         if n == 1:  
...:             return acc  
...:         else:  
...:             return fact_aux(n - 1, acc * n)  
...:     return fact_aux(n, 1)  
...:
```

```
In [2]: fact(-1)
```

```
func/<ipython console> in fact(n)  
func/<ipython console> in fact(n)  
...  
func/<ipython console> in fact(n)
```

```
RuntimeError: maximum recursion depth exceeded
```

Outline

- 1 Introduction
- 2 Writing functions**
- 3 Formal arguments
- 4 Variable scope
- 5 Returning functions
- 6 Functional programming

The simplest function

Example

```
In [1]: def f():  
...:     pass  
...:
```

```
In [2]: f()
```

```
In [3]: print f()  
-----> print(f())
```

```
None
```

```
In [4]: type(f)
```

```
Out[4]: <type 'function'>
```

```
In [5]:
```

Forward references

Example

```
In [1]: def f():  
...:     g()  
...:
```

```
In [2]: f()
```

NameError Traceback (most recent call last)

```
.../<ipython console> in <module>()  
.../<ipython console> in f()
```

NameError: global name 'g' is not defined

```
In [3]: def g():  
...:     pass  
...:
```

```
In [4]: f()
```

```
In [5]: g()
```

```
In [6]:
```

Function attributes

Description

- Dotted-attribute notation.
- Closely related to namespaces.

Example

```
In [1]: def f():  
...:     ' a docstring for function f()'  
...:     pass  
...:  
In [2]: def g():  
...:     pass  
...:  
In [3]: f.__doc__  
Out[3]: ' a docstring for function f()'  
In [4]: g.__doc__  
In [5]: g.__doc__ = 'some docstring for function g()'  
In [6]: g.__doc__  
Out[6]: 'some docstring for function g()'
```

Function attributes

Listing all attributes

```
In [1]: def f():  
...:     pass  
...:
```

```
In [2]: dir(f)
```

```
Out[2]:
```

```
['__call__', '__class__', '__closure__', '__code__', '__defaults__',  
'__delattr__', '__dict__', '__doc__', '__format__', '__get__',  
'__getattr__', '__globals__', '__hash__', '__init__', '__module__',  
'__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',  
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',  
'func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc',  
'func_globals', 'func_name']
```

Passing functions

Description

- In Python, functions are like any other object: they can be referenced (accessed or aliased to other variables), passed as arguments to functions, ...
- Unique characteristic: they are callable.

Example

```
In [1]: def f():  
...:     print 'in f()'  
...:  
In [2]: f()  
in f()  
In [3]: g = f  
In [4]: g()  
in f()  
In [5]:
```

Passing functions

Example

```
In [1]: def f():  
...:     print 'in f()'  
...:  
In [2]: def g(arg_func):  
...:     arg_func()  
...:  
In [3]: g(f)  
in f()  
In [4]: g(g)
```

TypeError Traceback (most recent call last)

.../<ipython console> in <module>()

.../<ipython console> in g(arg_func)

TypeError: g() takes exactly 1 argument (0 given)

In [5]:

Passing functions

Example

```
In [1]: def convert(func, x):  
...:     'conv a number'  
...:     return func(x)  
...:  
In [2]: convert(int, 12.34)  
Out[2]: 12  
In [3]: convert(long, 1234)  
Out[3]: 1234L  
In [4]: convert(float, 1234)  
Out[4]: 1234.0  
In [5]: convert(str, 1234)  
Out[5]: '1234'  
In [6]:
```

Passing functions

Example

```
In [1]: def apply_func(func, x):
...:     return func(x)
...:
```

```
In [2]: apply_func(str, 1234)
```

```
Out[2]: '1234'
```

```
In [3]: apply_func('a string', 1234)
```

```
-----
TypeError Traceback (most recent call last)
```

```
.../<ipython console> in <module>()
```

```
.../<ipython console> in apply_func(func, x)
```

```
TypeError: 'str' object is not callable
```

```
In [5]: callable('a sting')
```

```
Out[5]: False
```

```
In [6]: def apply_func(func, x):
...:     if callable(func):
...:         return func(x)
...:
```

```
In [7]: apply_func('a string', 1234)
```

```
In [8]:
```

Outline

- 1 Introduction
- 2 Writing functions
- 3 Formal arguments**
- 4 Variable scope
- 5 Returning functions
- 6 Functional programming

Positional arguments

Example

```
In [1]: def f(arg):
...:     print 'hello', arg  # defined for only 1 argument
...:
```

```
In [2]: f()  # 0 argument ...
```

```
-----
TypeError Traceback (most recent call last)
.../<ipython console> in <module>()
TypeError: f() takes exactly 1 argument (0 given)
```

```
In [3]: f('world')  # 1 argument, that's fine
hello world
```

```
In [4]: f('cruel', 'world')  # two many arguments ...
```

```
-----
TypeError Traceback (most recent call last)
.../<ipython console> in <module>()
TypeError: f() takes exactly 1 argument (2 given)
```

```
In [5]:
```

Default arguments

```
func(pos_args, def_arg1 = val1, def_arg2 = val2, ...)
```

Example

```
In [1]: def tax_me(cost, rate = .055):
...:     return cost + (cost * rate)
...:
```

```
In [2]: tax_me(100)
```

```
Out[2]: 105.5
```

```
In [3]: tax_me(100, .195)
```

```
Out[3]: 119.5
```

```
In [4]: tax_me(rate = .195, cost = 100)
```

```
Out[4]: 119.5
```

```
In [5]: def tax_me(rate = .055, cost):
...:     return cost + (cost * rate)
```

```
-----
File "<ipython console>", line 1
```

```
SyntaxError: non-default argument follows
              default argument (<ipython console>, line 1)
```

```
In [6]:
```

Default arguments

Example

```
In [1]: def conn(host='monge.univ-mlv.fr', database='sqlbase', port=1234):
...:     print 'host=%s,database=%s,port=%s' % (host, database, port)
...:

In [2]: conn() # use default arguments
host=monge.univ-mlv.fr,database=sqlbase,port=1234

In [3]: conn('localhost', 'my_sqlbase', 9999) # custom conn
host=localhost,database=my_sqlbase,port=9999

In [4]: conn('localhost', 'my_sqlbase') # use default port
host=localhost,database=my_sqlbase,port=1234

In [5]: conn('localhost') # use default database and default port
host=localhost,database=sqlbase,port=1234

In [6]: conn(port = 9999, database = 'my_sqlbase') # use default host
host=monge.univ-mlv.fr,database=my_sqlbase,port=9999

In [7]:
```

Default arguments

Example

```
n [1]: def conn(host, database='sqlbase', port='1234'):
...:     print 'host=%s,database=%s,port=%s' % (host, database, port)
...:
In [2]: conn()
```

TypeError Traceback (most recent call last)

```
.../<ipython console> in <module>()
TypeError: conn() takes at least 1 argument (0 given)
```

```
In [3]: conn('monge.univ-mlv.fr') # use default arguments
host=monge.univ-mlv.fr,database=sqlbase,port=1234
In [4]: conn('monge.univ-mlv.fr', database='flat') # use default port
host=monge.univ-mlv.fr,database=flat,port=1234
In [5]: conn('monge.univ-mlv.fr', port=9999) # use default database
host=monge.univ-mlv.fr,database=sqlbase,port=9999
In [6]: conn('monge.univ-mlv.fr', 9999) # !!!!!!!!!!!
host=monge.univ-mlv.fr,database=9999,port=1234
In [7]:
```

Modifying arguments

Example

```
In [1]: def f(arg):  
...:     print 'in f()', arg = ', arg  
...:     arg[0] = 'x'  
...:     print 'in f()', arg = ', arg  
...:
```

```
In [2]: l = ['a', 'b', 'c']
```

```
In [3]: f(l)  
in f(), arg = ['a', 'b', 'c']  
in f(), arg = ['x', 'b', 'c']
```

```
In [4]: l  
Out[4]: ['x', 'b', 'c']
```

```
In [5]:
```

Modifying arguments

Example

```
In [1]: def f(arg):  
...:     print '1: ', arg  
...:     arg = arg * 2  
...:     print '2: ', arg  
...:
```

```
In [2]: l = [1, 2, 3]
```

```
In [3]: f(l)  
1:  [1, 2, 3]  
2:  [1, 2, 3, 1, 2, 3]
```

```
In [4]: l  
Out[4]: [1, 2, 3]
```

Grouped arguments

Description

- Python allows the programmer to execute a function without explicitly specifying individual arguments in the call.
- Group in a tuple (**non-keyword arguments**).
- Group in a dictionary (**keyword argument**).

```
func(*tuple_grp_nonkw_args, **dict_grp_kw_args)

func(positional_args, kw_args,
     *tuple_grp_nonkw_args, **dict_grp_kw_args)
```

Non-keyword variable arguments

Example

```
n [1]: def f(arg1, arg2='default arg2', *the_rest):
...:     print 'formal arg 1:', arg1
...:     print 'formal arg 2:', arg2
...:     for arg in the_rest:
...:         print 'another arg:', arg
...:

In [2]: f('a') # use default argument, no non-keyword argument
formal arg 1: a
formal arg 2: default arg2

In [3]: f('a', 'b') # no non-keyword argument
formal arg 1: a
formal arg 2: b

In [4]: f('a', 'b', 'c', 'd') # some non-keyword argument
formal arg 1: a
formal arg 2: b
another arg: c
another arg: d

In [5]:
```

Keyword variable arguments

Example

```
In [1]: def f(arg1, arg2='default arg2', **the_rest):
...:     print 'formal arg 1:', arg1
...:     print 'formal arg 2:', arg2
...:     for kw_arg in the_rest.keys():
...:         print 'another arg %s: %s' % (kw_arg, str(the_rest[kw_arg]))
...:
In [2]: f('a')
formal arg 1: a
formal arg 2: default arg2
In [3]: f('a', 'b')
formal arg 1: a
formal arg 2: b
In [4]: f('a', 'b', xtra1='red', xtra2='green')
formal arg 1: a
formal arg 2: b
another arg xtra1: red
another arg xtra2: green
In [5]: f('a', 'b', 'c', xtra='red', xtra2='green') # ... TypeError !
```

Non-Keyword and Keyword variable arguments

Example

```
In [1]: def f(arg1, *nkw, **kw):  
...:     print 'arg1:', arg1  
...:     for nkw_arg in nkw:  
...:         print 'extra nkw arg:', nkw_arg  
...:     for (kw_arg, kw_val) in kw.items():  
...:         print 'extra kw arg %s: %s' % (kw_arg, kw_val)  
...:
```

Non-Keyword and Keyword variable arguments

Example

```
In [2]: f('a')
arg1: a
In [3]: f('a', 'b', 'c')
arg1: a
extra nkw arg: b
extra nkw arg: c
In [4]: f('a', kw_arg1='x', kw_arg2='y')
arg1: a
extra kw arg kw_arg2: y
extra kw arg kw_arg1: x
In [5]: f('a', kw_arg1='x', kw_arg2='y', 'b', 'c')
-----
File "<ipython console>", line 1
SyntaxError: non-keyword arg after keyword arg (<ipython console>, line 1)
In [6]: f('a', 'b', kw_arg1='x')
arg1: a
extra nkw arg: b
extra kw arg kw_arg1: x
```

Partial function application

Description

- Functional programming.
- Default and variable arguments.

Example

```
In [1]: from operator import add, mul    # function interfaces to + and *
In [2]: from functools import partial
In [3]: succ = partial(add, 1)
In [4]: double = partial(mul, 2)
In [5]: mul_3 = partial(mul, 3)
In [6]: [(i, succ(i)) for i in range(1, 20, 3)]
Out[6]: [(1, 2), (4, 5), (7, 8), (10, 11), (13, 14), (16, 17), (19, 20)]
In [7]: [(i, double(i)) for i in range(1, 20, 3)]
Out[7]: [(1, 2), (4, 8), (7, 14), (10, 20), (13, 26), (16, 32), (19, 38)]
In [8]: [(i, mul_3(i)) for i in range(1, 20, 3)]
Out[8]: [(1, 3), (4, 12), (7, 21), (10, 30), (13, 39), (16, 48), (19, 57)]
In [9]:
```

Partial function application

Example

```
In [1]: from functools import partial
```

```
In [2]: int?
```

```
Type:                type
Base Class:          <type 'type'>
String Form:         <type 'int'>
Namespace:           Python builtin
Docstring:
    int(x[, base]) -> integer
```

Convert a string **or** number to an integer, **if** possible. A floating point argument will be truncated towards zero (this does **not** include a string representation of a floating point number!) When converting a string, use the optional base. It **is** an error to supply a base when converting a non-string. If base **is** zero, the proper base **is** guessed based on the string content. If the argument **is** outside the integer **range** a **long object** will be returned instead.

Partial function application

Example

```
In [3]: base2 = partial(int, base=2)
In [4]: base2('0'), base2('1'), base2('101010101'), base2('11111111')
Out[4]: (0, 1, 341, 255)
In [5]: base10 = partial(int, base=10)
In [6]: base10('0'), base10('1'), base10('2'), base10('10')
Out[6]: (0, 1, 2, 10)
In [7]:
```

Outline

- 1 Introduction
- 2 Writing functions
- 3 Formal arguments
- 4 Variable scope**
- 5 Returning functions
- 6 Functional programming

Global versus local variables

Example

```
In [1]: global_string = "a string" # a global variable
In [2]: def f():
...:     local_string = "another string" # a local variable
...:     return global_string + " " + local_string
...:
In [3]: f()
Out[3]: 'a string another string'
In [4]: print local_string
-----> print(local_string)

-----

NameError Traceback (most recent call last)
.../<ipython console> in <module>()

NameError: name 'local_string' is not defined
In [5]:
```

The global statement

Example

```
In [1]: def f():
...:     print "calling f()"
...:     my_variable = 2
...:     print "in f(), my_variable =", my_variable
...:
In [2]: my_variable = 1
In [3]: my_variable
Out[3]: 1
In [4]: f()
calling f()
in f(), my_variable = 2
In [5]: my_variable
Out[5]: 1
In [6]: my_variable = 3
In [7]: f()
calling f()
in f(), my_variable = 2
In [8]:
```

The global statement

Example

```
In [1]: def f():  
...:     print "calling f()"   
...:     global my_variable  
...:     my_variable = 2  
...:     print "in f(), my_variable =", my_variable  
...:
```

The global statement

Example

```
In [2]: my_variable = 1
In [3]: my_variable
Out[3]: 1
In [4]: f()
calling f()
in f(), my_variable = 2
In [5]: my_variable
Out[5]: 2
In [6]: my_variable = 3
In [7]: my_variable
Out[7]: 3
In [8]: f()
calling f()
in f(), my_variable = 2
In [9]: my_variable
Out[9]: 2
```

Number of scopes

Example

```
In [1]: def f():
...:     x = 1
...:     print "in f(), and before calling g(), x is", x
...:     def g():
...:         x = 2
...:         print "in g(),x is", x
...:     g()
...:     print "in f() and after calling g(), x is", x
...:

In [2]: f()
in f(), and before calling g(), x is 1
in g(),x is 2
in f() and after calling g(), x is 1

In [3]:
```

Closures

Description

- A closure combines an inner function's own code and scope along with the scope of an outer function.
- Closures are useful for setting up calculations, hiding states, letting you move around function objects and scope at will.
- The use of closures draws a strong parallel to partial function application.

Closures: you cannot assign to an outer-scope variable

Example

```
>>> def outer(x):  
...     def inner_reads():  
...         # Will return outer's 'x'.  
...         return x  
...     def inner_writes(y):  
...         # Will assign to a local 'x', not the outer 'x'  
...         x = y  
...     def inner_error(y):  
...         # Will produce an error: 'x' is local because of the assignment,  
...         # but we use it before it is assigned to.  
...         tmp = x  
...         x = y  
...         return tmp  
...     return inner_reads, inner_writes, inner_error  
...
```

Closures: you cannot assign to an outer-scope variable

Example

```
>>> inner_reads, inner_writes, inner_error = outer(5)
>>> inner_reads()
5
>>> inner_writes(10)
>>> inner_reads()
5
>>> inner_error(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in inner_error
UnboundLocalError: local variable 'x' referenced before assignment
```

Closures: work around this limitation by using a mutable container type

Example

```
>>> def outer(x):  
...     x = [x]  
...     def inner_reads():  
...         # Will return outer's x's first (and only) element.  
...         return x[0]  
...     def inner_writes(y):  
...         # Will look up outer's x, then mutate it.  
...         x[0] = y  
...     def inner_error(y):  
...         # Will now work, because 'x' is not assigned to, just referenced.  
...         tmp = x[0]  
...         x[0] = y  
...         return tmp  
...     return inner_reads, inner_writes, inner_error  
...
```

Closures: work around this limitation by using a mutable container type

Example

```
>>> inner_reads, inner_writes, inner_error = outer(5)
>>> inner_reads()
5
>>> inner_writes(10)
>>> inner_reads()
10
>>> inner_error(15)
10
>>> inner_reads()
15
```

Closures

Example

```
In [1]: def counter(start_at = 0):  
...:     count = [start_at]  
...:     def incr():  
...:         count[0] += 1  
...:         return count[0]  
...:     return incr  
...:
```

Closures

Example

```
In [2]: count = counter()
In [3]: print count()
-----> print(count())
1
In [4]: print count()
-----> print(count())
2
In [5]: count = counter(5)
In [6]: print count()
-----> print(count())
6
In [7]: print count()
-----> print(count())
7
In [8]:
```

Outline

- 1 Introduction
- 2 Writing functions
- 3 Formal arguments
- 4 Variable scope
- 5 Returning functions**
- 6 Functional programming

Tracing functions

Example

```
def trace(f):  
    def wrap(*args):  
        print "before calling " + f.__name__  
        f(args)  
        print "after calling " + f.__name__  
    return wrap  
  
def g(*args):  
    print "in g"  
  
g('a', 'b')  
h = trace(g)  
h('a', 'b')
```

Output

in g before calling g in g after calling g

Composition

Example

```
In [1]: def compose(f, g):  
...:     "Return a function f(g(x)) given two functions f,g"  
...:     def h(x):  
...:         return f(g(x))  
...:     return h  
...:  
In [2]: def f(x): return x+1  
...:  
In [3]: def g(x): return 2*x  
...:
```

Composition

Example

```
In [4]: s = compose(f,g)           # store the composition
In [5]: s
Out[5]: <function h at 0x8374454>  # s is a function !
In [6]: s(3)                       # apply s
Out[6]: 7
In [7]: compose(f, g)(3)           # do not store the composition
Out[7]: 7
In [8]: r = s                       # alias
In [9]: r(3)
Out[9]: 7
In [10]:
```

Composition

Example

```
def f(x):  
    return x+1  
  
def power(f, n):  
    "Return a function of f composed with itself n times"  
    def identity(x):  
        return x  
    if n == 0:  
        return identity  
    f_res = f  
    for _ in range(n-1):  
        f_res = compose(f_res, f)  
    return f_res  
  
print f(f(f(f(1))))           # print 5  
power4 = power(f, 4)  
print power4(1)                # print 5  
print power(f, 0)(1)           # print 1
```

Outline

- 1 Introduction
- 2 Writing functions
- 3 Formal arguments
- 4 Variable scope
- 5 Returning functions
- 6 Functional programming**

filter

Example

```
In [1]: def modulo(m):  
...:     def inner(x):  
...:         return x % m == 0  
...:     return inner  
...:  
In [2]: l = range(10)  
In [3]: filter(modulo(2), l)  
Out[3]: [0, 2, 4, 6, 8]  
In [4]: filter(modulo(3), l)  
Out[4]: [0, 3, 6, 9]  
In [5]: filter(modulo(4), l)  
Out[5]: [0, 4, 8]  
In [6]: filter(modulo(5), l)  
Out[6]: [0, 5]  
In [7]:
```

filter

Coding filter

```
In [1]: def my_filter(bool_func, seq):  
...:     filtered_seq = [] # start with the empty seq  
...:     for x in seq:  
...:         if bool_func(x):  
...:             filtered_seq.append(x)  
...:     return filtered_seq  
...:  
In [2]: def is_odd(x):  
...:     return x % 2  
...:  
In [3]: my_filter(is_odd, range(10))  
Out[3]: [1, 3, 5, 7, 9]  
In [4]:
```