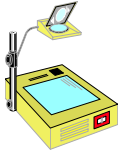




## Python: Chapter 4 Functions



Learning Python, Lutz & Ascher

1

## What's a function?

- **Code reuse**: groups statements so that they can run multiple times in a pgm
- **Program decomposition**: partition program into pieces, each with a well-defined role
- **Parameters**: differ each time code is executed

```
def adder(a, b, c=1, *d) : return a+b+c+d[0]  
def fun() : global x, y; x = 'n';
```

2

## Function syntax

- **def**: creates function object, gives name
- **return**: sends result back
- **global**: declares *module* level variables that are to be assigned; by default, all names declared in a function are local
- Args are passed by **assignment**
- Args, variables and return types are not declared!

3

## Easy function example

```
>>>def times(x, y):  
...     return x * y  
...  
>>>times(2, 4)  
8  
>>>times('cat', 3)  
'catcatcat'
```

Python is dynamically typed!

4

## Intersection example

```
def intersect(s1, s2):  
    result = []  
    for x in s1:  
        if x in s2:  
            result.append(x)  
    return result  
  
Calls:  
>>>x = 'catalog'  
>>>y = 'gallon'  
>>>intersect(x, y)  
['a', 'a', 'l', 'o', 'g']  
>>>intersect([1,2,3], [1,4])  
[1]
```

5

## Scope rules

- When a name is used in a Python program, lookup occurs first in the current namespace
- Assignment attaches a name to a namespace
- 3 namespaces: **LGB**

6

## LGB names

### Built-in (Python)

- predefined: open, len, etc

### Global (module)

- assigned at top level  
- global in function

### Local (function)

- names assigned in function

7

## Scope example

```
x = 99 #global scope
```

```
def fun(y):  
    global sum  
    z = x + y #x is global, z & y are local  
    sum = z  
    return z
```

Closest structure to a declaration in Python. Means a name at top-level of the module

8

## Parameter transmission

```
>>>def change(x, y):  
...     x = 1  
...     y[0] = 'c'  
...  
>>>x = 99  
>>>L=[1,2,3]  
>>>fun(x, L)  
>>>x  
>>>L  
>>>fun(x, 'bat')
```

Immutable objects: since you can't change them, it's like pass by const reference in C++

Mutable objects: can be changed in place

9

## Multiple return values/types

```
>>>def multiple(x, y):  
...     x = 2; y = [3,4]  
...     return x, y  
...  
>>>x = 1  
>>>L = [1,2]  
>>>x, L = multiple(x, L)  
>>>x, L  
(2, [3,4])
```

10

## Special argument matching (optional)

- Default: matched by position, L to R
- number at call must match declaration
- Can also specify match by name, default values and collectors (for extra args)

```
def fun(a, b, c=1, d=2):  
    print a, b, c, d
```

```
fun(1,2,3,4)  
fun(1, 2)  
fun(b=7, a=2)  
fun(c=9, a=1, b=2)
```

11

## Arbitrary argument set functions

```
def intersect(*args):  
    result = []  
    for x in args[0]:  
        for y in args[1:]:  
            if x not in y: break;  
            else: result.append(x)  
    return result  
  
def union(*args):  
    result = []  
    for seq in args:  
        for x in seq:  
            if not x in result: result.append(x)  
    return result
```

12

## Lambda functions

- General form: lambda followed by one or more args, followed by colon and an expression
- Lambda is an expression, not a statement

```
>>>def sum(x, y, z): return x+y+z
>>>fun(2,3,4)
9

>>> f = lambda x, y, z: x+y+z
>>> f(2,3,4)
9
```

13

## Lambda with defaults

```
>>> f = (lambda x='c', y='a', z='t': x+y+z)
>>> f('b')
'bat'
```

14

## Lambda with lists

```
>>>L = [lambda x: x**2, lambda x: x**3, lambda x: x**4]
>>>for f in L:
...     print f(2)
4 8 16
>>>print L[0](3)
9
```

15

## Using the *apply* builtin function

```
>>> f = lambda x, y, z: x+y+z
>>> apply(f, (2,3,4))
9
```

16

## Using the *map* builtin function

```
>>>counters = [1,2,3,4]
>>>updated = []
>>>for x in counters:
...     updated.append(x+10)
>>>updated
[11,12,13,14]

>>>def inc(x): return x + 10
...
>>>map(inc, counters)
[11,12,13,14]

>>>map(lambda x: x + 3), counters
[11,12,13,14]
```

17

## procedures

Return statements are option;  
When a function doesn't return  
A value, it returns None:

```
>>> def procedure(x):
...     print x
...
>>>x = procedure('this is a test')
this is a test
>>>print x
None
```

18

## Design of functions

- Use arguments for inputs, return for output
- Use globals rarely and only when necessary
- Don't change mutable arguments unless user expects it

19

## Functions are objects

```
>>>def output(msg):
...     print msg

>>>x = output
>>>x("Hello world")
Hello world

>>>L = [output, "cat"], (output, "dog")
>>>for (fun, arg) in L:
...     apply(fun, (arg,))
cat
dog
```

20

## Function gotchas: local names

```
x = 77
>>>def output():
...     print x
...     x = 1
...
>>>output()
ERROR!!!
```

Whoa – this is subtle; the assignment, `x = 1`, makes `x` local to `output`, so `x` doesn't exist at the statement `print x` in `output`!

```
Solution:
x = 77
>>>def output():
...     global x
...     print x
...     x = 1
```

21

## Default values and mutable objects

```
>>>def whatsup(x = []):
...     x.append(1)
...     print x
...
>>>whatsup([2])
[2, 1]
>>>whatsup()
[1]
>>>whatsup()
[1,1]
>>>whatsup()
[1,1,1]
```

Only one list

22