

Functions

Functions

def statement

Defines a new function

```
def fname(args):  
    statements
```

Example:

```
def divide(a,b):  
    return a/b
```

Functions

The def statement

Return the remainder of a/b

```
def remainder(a,b):  
    q = a/b  
    r = a - q*b  
    return r
```

Now use it

```
a = remainder(42,5)    # a = 2
```

Returning multiple values

```
def divide(a, b):  
    q = a / b  
    r = a - q * b  
    return q ,r
```

```
x, y = divide(42, 5)      # x = 8, y = 2
```

Calling a function

Calling a function

```
x = divide(7,3)
```

Calling a function with keyword arguments

```
x = divide(b=3, a=7)
```

Exact same result as before, argument names specified explicitly.

Default Arguments

Function definition with default values

```
def spam(a,b, c=37, d="Guido"):
    print a, b, c, d          # statements
...
```

spam(2,3)

spam(2,3,4)

spam(2,3,4,"Dave")

spam(2,3,d="Dave")

General Comments:

1. No non-optional arguments can follow an optional argument.
2. Can mix non-keyword and keyword arguments together when calling (with care).
3. Value of an optional argument is determined at time of function definition.

Value of an optional argument is determined at time of function definition

x = 10

```
def spam(a = x):  
    print a
```

x = 20

spam()

Produces ??

Scoping Rules

Local vs. Global Scope:

- Functions execute within a local and global scope.
- Local scope is just the function body.
- Global scope is the module in which the function was defined.
- Variable lookups first occur in local scope, then in global scope.
- All variable assignments are made to the local scope.

LGB = Local --> Global --> Built-in

Local vs. Global Scope

x = 4

```
def foo():
```

```
    x = x + 1
```

Creates a local copy of x

```
foo()
```

```
print x
```

Outputs '?'

The **global** statement declares a variable name to refer to global scope.

```
def foo():
```

```
    global x
```

```
    x = x + 1
```

Modifies the global x

Parameter Passing and Return Values

- Parameters are passed by reference
- This has certain implications for lists, dictionaries, and similar types

```
def foo(x):  
    x[2] = 10
```

```
a = [1,2,3,4,5]
```

```
foo(a)
```

```
print a
```

```
# Outputs '[]'
```

Return values

- Values are returned as a **tuple** of results.

```
def foo():  
    ...  
    return (x,y,z)
```

```
d = foo()           # Gets a tuple with three values  
(a,b,c) = foo()    # Assigns values to a, b, and c.  
a,b,c = foo()      # Same thing
```

In Python:

functions are first-class objects

This means:

you can manipulate functions like all other types (integers, floats, lists, etc...)

```
def add(a,b):  
    return a+b
```

```
a = add
```

```
print a(3,4)
```

Default Arguments

Function definition with default values

```
def spam(a,b, c=37, d="Guido"):
    print a, b, c, d          # statements
...
```

spam(2,3)

spam(2,3,4)

spam(2,3,4,"Dave")

spam(2,3,d="Dave")

General Comments:

1. No non-optional arguments can follow an optional argument.
2. Can mix non-keyword and keyword arguments together when calling (with care).
3. Value of an optional argument is determined at time of function definition.

Variable Length Arguments

- A function accepting variable number of arguments

```
def printf(fmt, *args):  
    print fmt % args
```

- Extra arguments are passed as a **tuple** in args

Accepting an **arbitrary** set of **keyword** arguments

```
def foo(**kwargs):  
    print kwargs
```

- **kwargs** is a **dictionary** mapping argument names to values.

Accepting both **positional** and **keyword** arguments

```
def foo(arg1, *vargs, **kwargs):  
    print arg1, vargs, kwargs    #statements
```

```
foo(1,2,3,4,name="Guido")
```

```
#arg1 = 1, vargs = (2,3,4), kwargs= {'name':'Guido'}
```

The lambda operator

- You can also create anonymous functions with lambda

```
a = lambda x,y: x+y
```

...

```
b = a(3,4)
```

- This is particularly useful for callback functions
- However, lambda can only be used to specify simple expressions.

The apply Function

- The `apply(func [,args [,kwargs]])` function can be used to call a function
- `args` is a tuple of positional arguments.
- `kwargs` is a dictionary of keyword arguments.

```
foo(3, 'x', name='Dave', id=12345)
```

```
apply(foo, (3, 'x'), {'name': 'Dave', 'id': 12345 })
```

```
# These two statements are exactly the same
```

Why would you use this?

Sometimes it is useful to manipulate arguments and call another function

```
def fprintf(file,fmt, *args):  
    file.write(fmt % args)
```

```
def printf(fmt, *args):  
    apply(fprintf, (sys.stdout,fmt)+args)
```

The map Function

- **map(func, s)** applies a function to each element of a sequence

a = [1,2,3,4,5,6]

```
def foo(x):  
    return 3*x
```

b = map(foo,a) # b = [3,6,9,12,15,18]

Alternatively...

b = map(lambda x: 3*x, a) # b = [3,6,9,12,15,18]

Special cases:

- **map can be applied to multiple lists.**

b = map(func, s1, s2, ... sn)

- **In this case, func must take n arguments.**
- **If the function is None, the identity function is assumed.**

The reduce Function

- `reduce(func, s)` collects data from a sequence and returns a single value

`a = [1,2,3,4,5,6]`

```
def sum(x,y):  
    return x+y
```

`b = reduce(sum, a)` # `b = (((((1+2)+3)+4)+5)+6) = 21`

Alternatively...

`b = reduce(lambda x,y: x+y, a)`

The function given to reduce must accept two arguments and return a single result (suitable for reuse as one of the arguments).

The filter Function

filter(func, s) filters the elements of a sequence

a = [1,2,3,4,5,6]

b = filter(lambda x: x < 4, a) # b = [1,2,3]

- **If func is None, filter() returns all of the elements that evaluate to true.**

Exec, Eval, and Execfile

- The eval function

Evaluates a string as a Python expression

```
a = eval("3*math.sin(3.5+x)+7.2")
```

- The exec statement

Executes a string containing arbitrary Python code

```
a = [3,5,10,13]
```

```
exec "for i in a: print i"
```

- The execfile function

Executes the contents of a file

```
execfile("foo.py")
```

Note: exec is a statement. eval and execfile are functions

The End