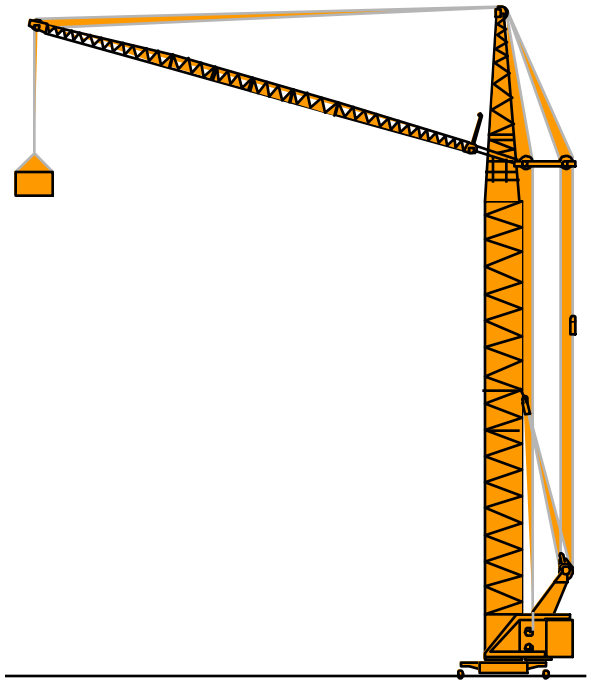
python™ 2 programming

Advanced Functions

Functions

- Contents
  - Variadic functions
  - Assigning default values
  - Named parameters
  - Variables in functions
  - Nested functions
  - Closure Function
  - Decorators
  - Function documentation
  - Lambda functions
- Summary
  - Function attributes



Passing parameters - review

```
def myfunc(file, dir, user='root'):
    print "file: %s, dir: %s, to: %s" % (file,dir,user)
```

- **By position**

```
myfunc('one','two','three')
```

```
file: one, dir: two, to: three
```

- **By default**

```
myfunc('one','two')
```

```
file: one, dir: two, to: root
```

- **Or by name**

```
myfunc(file='one',user='three',dir='two')
```

```
file: one, dir: two, to: three
```

Variadic functions

- **Functions usually have a fixed number of parameters**

```
def myfunc(dir, files):
    print "dir:", dir, "files:", files

myfunc('c:/stuff', 'f1.txt', 'f2.txt', 'f3.txt')
```

```
TypeError: myfunc() takes exactly 2 positional arguments (4 given)
```

- **Variadic functions have a variable number of parameters**
  - **They can be collected into a tuple with a * prefix**
  - **Known as *unpacking***

```
def myfunc(dir, *files):
    print "dir:", dir, "files:", files

myfunc('c:/stuff', 'f1.txt', 'f2.txt', 'f3.txt')
```

```
dir: 'c:/stuff', files: ('f1.txt', 'f2.txt', 'f3.txt')
```

Keyword parameters

- **Look just like the key-value pairs of a dictionary**
  - **Because that is what they are**
- **Prefix a parameter with ** to indicate a dictionary**
  - **Since a dictionary is unordered, then so are the parameters**
  - **May only come at the end of a parameter list**

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

print_vat(vatpc=15, gross=9.55, message='Summary')
```

```
{'gross': 9.55, 'message': 'Summary', 'vatpc': 15}
```

- **Use ** if caller's parameters are in a dictionary**

```
Argsdict = dict(vatpc=15, gross=9.55, message='Summary')
print_vat(**Argsdict)
```

Variables in functions

- **By default, variables used in a function are local**
- **Global variables are defined using `global`**
  - **Are local to the current module, or *namespace***

```
result = 3

def scope_test1():
    result = 42

scope_test1()
print result          →  3

def scope_test2():
    global result
    result = 42

scope_test2()
print result          →  42
```

Nested functions

- **The def statement defines a function object**
  - **This has the same scope as any other object**

```
def outer():
    num = 42

    def inner():
        print num, "in inner"

    inner()
    print num, "in outer"

outer()
inner()
```

```
42 in inner
42 in outer
NameError: name 'inner' is not defined
```

Closure Function

- **This technique by which some data ("Hello") gets attached to the code is called *closure* in Python.**

```
def print_msg(msg):
# This is the outer enclosing function
    def printer():
# This is the nested function
        print(msg)
    return printer  # this got changed

# Now let's try calling this function.
another = print_msg("Hello")
another()
# Output: Hello
```

- This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current.

When do we have a closure?

- We must have a nested function
- The nested function must refer to a value defined in the enclosing function
- The enclosing function must return the nested function

When to use closures?

- **Closures can avoid the use of global values and provides some form of data hiding.**
- **It can also provide an object oriented solution to the problem (instead of a new class)**

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier
# Multiplier of 3
times3 = make_multiplier_of(3)
# Multiplier of 5
times5 = make_multiplier_of(5)
# Output: 27
print(times3(9))
# Output: 15
print(times5(3))
```

Decorators

- **Decorator is a function that takes another function and extends the behavior of the latter function *without* explicitly modifying it**

```
def my_decorator(some_function):
    def wrapper():
        print("Something is happening before some_function() is called.")
        some_function()
        print("Something is happening after some_function() is called.")
    return wrapper

def just_some_function():
    print("Wow!")
just_some_function = my_decorator(just_some_function)

just_some_function()
```

Using @ “pie” syntax

- **Syntactic sugar**
  - **Python allows you to simplify the calling of decorators using the @ symbol**

```
def just_some_function():
    print("Wow!")
just_some_function = my_decorator(just_some_function)
```

```
@my_decorator
def just_some_function():
    print("Wow!")
```

Decorator - Real World Example

```
import time
def timing_function(some_function):
    """
    Outputs the time a function takes
    to execute.
    """
    def wrapper():
        t1 = time.time()
        some_function()
        t2 = time.time()
        return "Time it took to run the function: " + str((t2 - t1)) + "\n"
    return wrapper
```

```
@timing_function
def my_function():
    num_list = []
    for num in (range(0, 10000)):
        num_list.append(num)
    print("\nSum of all the numbers: " + str((sum(num_list))))
```

```
print(my_function())
```

Function documentation

- **Comments have limited use**
  - **Useful for maintainers, but not designed for users**
- **Python supports** *docstrings*
  - **Used for help() and for automated testing**
  - **Define a bare string at the start of the function**
  - **Or explicitly set the attribute __doc__**

```
def MyFunc1():
    """MyFunc1 has no parameters
        and prints 'Hello'."""
    print "Hello"
```

Use triple quotes over several lines

```
>>> help (MyFunc1)
Help on function MyFunc1 in module __main__:

MyFunc1()
    MyFunc1 has no parameters and prints 'Hello'.
```

Lambda functions

- **Anonymous short-hand functions**
  - **Cannot contain branches or loops**
  - **Can contain *conditional expressions***
  - **Cannot have a `return` statement or assignments**
  - **Last result of the function is the returned value**

```
compare=lambda a,b: -1 if a < b else (+1 if a > b else 0)

x = 42
y = 3

print "a>b",compare(x,y)
```

Parameters

```
a>b 1
```

- **Often used with the `map()` ,`filter()` and `reduce()` built-ins**

Map, Filter and Reduce with Lambdas

- **Map**

```
new_list = map(lambda a: a+1,some_list)
```

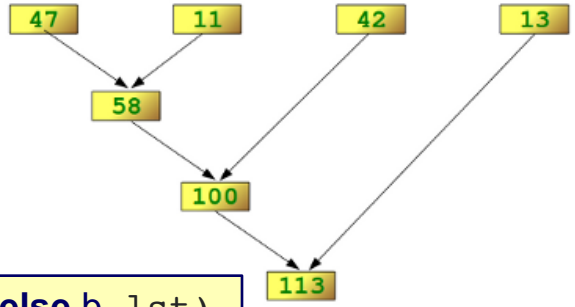
  - Applies an operation to each item in a list
  - Can handle more than one iterable

```
nums = [10,20,30]
inters = [1.17]*3
numstax = map(lambda val,inter:val*inter,nums,inters)
```

- **Filter**
  - filter out all the elements of an iterable, for which the function returns True

```
filter(lambda x: x%2==0,lst)
```

- **Reduce**
  - reduce(function, sequence) continually applies the function (2 args) to the sequence. It then returns a single value




```
lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst)
```

```
reduce(lambda a,b: a if (a > b) else b,lst)
```

Lambda as a sort key

- **Takes the element to be compared**
- **Returns the key in the correct format**
  - Sort each country by the second field, population

```
countries=[]
for line in open('country.txt') :
    countries.append(line.split(','))

countries.sort(key=lambda c: (int(c[1])))

for line in countries:
    print ','.join(line),
```

```
Antarctica,0,-,-,Antarctica,1961,-,-,-
Arctica,0,-,-,Arctic Region,-,-,-,-
Pitcairn Islands,46,Adamstown,?,Oceania,-,...
Christmas Island,396,The Settlement,?,Oceania,...
Johnston Atoll,396,-,-,Oceania,-,US Dollar,-,...
```

Lambda in re.sub

- The re.sub method can take a function as a replacement
  - Passes a match object to the function
  - The return value is the value substituted

```
import re
codes = {}

names = ['zero','wun','two','tree','fower','fife','six','seven',
         'ait','niner','alpha','bravo','charlie','delta','echo',
         'foxtrot','golf','hotel','india','juliet','kilo','lima',
         'mike','november','oscar','papa','quebec','romeo',
         'sierra','tango','uniform','victor','whisky','xray',
         'yankee','zulu']

for key in (xrange(0,10)):
    codes[str(key)] = names[key]
for key in (xrange(ord('A'),ord('Z')+1)):
    codes[chr(key)] = names[key - ord('A')+10]

reg = 'WG07 OKD'

result = re.sub(r'(\w)',
                lambda m: codes[m.groups()[0]]+' ', reg)
```

Summary

- **A function is a defined object**
  - **Variables have local scope unless `global` is used**
  - **Other functions can be nested within**
- **Parameters are declared local variables**
  - **May be assigned defaults, from the right**
  - ****arg* means unpack to a tuple**
  - *****arg* means unpack to a dictionary**
- **Can return any object**
  - **Including lists and dictionaries**
- **Can include a *docstring* - a bare string at the start**
- **Short, inline, anonymous functions can be defined using lambda**

More on named parameters

```
import sys

def myfunc(**UserArgs):
    args = {'country':'England','town':'London',
            'currency':'Pound', 'language':'English'}

    diff = set(UserArgs.keys()) - set(args.keys())
    if diff:
        print >> sys.stderr,"Invalid arguments:",tuple(diff)
        return

    args.update(UserArgs)
    print args

mydict = dict(town = 'Glasgow', country = 'Scotland')
myfunc(**mydict)

myfunc(twn = 'Glasgow', county = 'Scotland')
```

```
{'town': 'Glasgow', 'currency': 'Pound',
                  'language': 'English', 'country': 'Scotland'}
Invalid arguments: ('county', 'twn')
```

Set operators reminder

- Includes set operators
  - Can use a method call instead

```
setA = set('John')
setB = set('Jane')
```

Operator	Method	Returns a new set containing
&	`setA.intersection(setB)`	Each item that is in both sets
|	`setA.union(setB)`	All items in both sets
-	`setA.difference(setB)`	Items in setA not in setB
^	`setA.symmetric_difference (setB)`	Items that occur in one set only

```
print setA & setB
print setA | setB
print setA - setB
print setA ^ setB
```

```
set(['J', 'n'])
set(['a', 'e', 'h', 'J', 'o', 'n'])
set(['h', 'o'])
set(['a', 'e', 'h', 'o'])
```

Function attributes

func_name	The function's name.
__name__	As func_name (compatible with Python 3)
func_doc	The *docstring* defined in the function's source code.
__doc__	As func_doc (compatible with Python 3)
func_defaults	A tuple containing default argument values for those arguments that have default values.
func_dict	The namespace supporting arbitrary function attributes.
func_closure	A tuple of cells that contain bindings for the function's free variables.
func_globals	Reference to the global namespace of the module in which the function was defined.
func_code	Code object representing the compiled function body.

Function annotation (python 3)

- **Similar to inline comments**
  - **Not supported for lambda functions**

```
def print_vat (*,
           gross:   "Gross amount (including VAT)"=0,
           vatpc:   "VAT in percentage terms"=17.5,
           message: "Free text"='Summary:') \
           -> "No usable return value":
```

- **Function attribute `__annotations__` gives details**

```
for kv in print_vat.__annotations__.items():
    print kv
```

```
('gross', 'Gross amount (including VAT)')
('message', 'Free text')
('vatpc', 'VAT in percentage terms')
('return', 'No usable return value')
```