Functional Programming

- Python as a "hybrid" language

  - It supports the functional programming paradigm,

  - but equally supports imperative paradigms (both procedural and object-oriented).

- Functions as First class Objects in Python

  - they have attributes and

  - they can be referenced and assigned to variables

- Higher order functions in Python

  - take one or more functions as argument or
  - return a function

Functional Programming (functions as objects)

```
def foo(s):
    'I am a function object'
    print s

bar = foo

bar("hello")
print "my doc string is: ", bar.__doc__
print "my dictionary is: ", bar.__dict__
print "my module name is: ",
bar.__module__
print "my name is: ", bar.__name__
print "my address is:", bar
```

```
hello
my doc string is:  I am a function object
my dictionary is:  {}
my module name is:  __main__
my name is:  foo
my address is: <function foo at 0x7f2788f895f0>
```

Functions as arguments

```
def call_func(f, *args):
    return f(*args)

#call_func takes another function(anonymous)
as its first argument
call_func(lambda x, y: x + y, 4, 5)
```

Having a function as a return value

- Functions can be defined within the scope of another function. Most useful when the inner function is being returned.
- A new instance of the function inner() is created on each call to outer(). That is because it is defined during the execution of outer(). The creation of the second instance has no impact on the first.

```
def outer():
   def inner(a):
      "a nested function"
      return a
   return inner

f1 = outer()
f2 = outer()
print "outer is :", outer
print "inner is :", f1
print "inner is :", f2
```

```
outer is : <function outer at 0x7f452d0c6668>
inner is : <function inner at 0x7f452d0c66e0>
inner is : <function inner at 0x7f452d0c6758>
```

Python closures

A nested function has access to the environment in which it was defined. Remember from above that the definition occurs during the execution of the outer function. Therefore, it is possible to return an inner function that remembers the state of the outer function, even after the outer function has completed execution. This model is referred to as a closure.

```
def outer(a):
   def inner(b):
    return a + b
   return inner

add1 = outer(1) #a is set to 1
print "add1 is ", add1
print "add1(4) is ", add1(4)
print "add1(5) is ", add1(5)
add2 = outer(2) #a is set to 2
print "add2 is ", add2
print "add2(4) is ", add2(4)
print "add2(5) is ", add2(5)
```

```
add1 is  <function inner at 0x7ff29f6df6e0>
add1(4) is  5
add1(5) is  6
add2 is  <function inner at 0x7ff29f6df8c0>
add2(4) is  6
add2(5) is  7
```

Python closures

- A common pattern that occurs while attempting to use closures, and leads to confusion, is attempting to encapsulate an internal variable using an immutable type. When it is re-assigned in the inner scope, it is interpreted as a new variable and fails because it hasn't been defined.
- The standard workaround for this issue is to use a mutable datatype like a list and manage state within that object.

```
#scoping problem in nested functions
def outer():
    count = 0
    def inner():
        count += 1
        return count
    return inner

counter = outer()
#UnboundLocalError: local variable
'count' referenced before assignment
print counter()
```

```
# quick fix for scoping problem
def outer():
    count = [0]
    def inner():
        count[0] += 1
        return count[0]
    return inner

counter = outer()
print counter() #1
```

Mutable vrs. Immutable types

Not all python objects handle changes the same way.

- Some objects are mutable, meaning they can be altered.
- Others are immutable; they cannot be changed but rather return new objects when attempting to update.

Python closures
mutable datatypes?
(list, set, dictionary, user-defined classes)

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

- Primitive-like types are probably immutable.
- Container-like types are probably mutable.

When mutability matters!

(list, set, dictionary, user-defined classes)

```
container = {"hello", "world", "end"}
string_build = ""
for data in container:
    string_build += str(data)
    print "id of string_build is ", id(string_build)
```

id of string_build is 140397490834864
id of string_build is 140397490835152
id of string_build is 140397490840552

```
container = {"hello", "world", "end"}
list_build = []
for data in container:
    list_build.append(str(data))
    print "id of list_build is ", id(list_build)
```

id of list_build is 140633273771072
id of list_build is 140633273771072
id of list_build is 140633273771072

When mutability fails!

Python evaluates default arguments as part of the function definition only once for mutable type

```
def doSomething(param=[]):
    param.append("thing")
    return param
```

```
a1 = doSomething()
print id(a1),"=",a1 #140114778712904 = ['thing']
a2 = doSomething()
print id(a2),"=",a2 #140114778712904 = ['thing', 'thing']
a3 = doSomething()
print id(a3),"=",a3 #140114778712904 = ['thing', 'thing', 'thing']
a4 = doSomething(["passed_1"])
print id(a4),"=",a4 #140114778713408 = ['passed_1', 'thing']
a5 = doSomething(["passed_2"])
print id(a5),"=",a5 #140114778713336 = ['passed_2', 'thing']
```

When mutability fails!

Use Immutable types for intended effect

```
def doSomething(param=None):
    if param == None:
        param = []

    param.append("thing")
    return param
```

```
a1 = doSomething()
print id(a1),"=",a1 #140114778713552 = ['thing']
a2 = doSomething()
print id(a2),"=",a2 #140114778713480 = ['thing']
a3 = doSomething()
print id(a3),"=",a3 #140114778656712 = ['thing']
a4 = doSomething(["passed_1"])
print id(a4),"=",a4 #140114778712904 = ['passed_1', 'thing']
a5 = doSomething(["passed_2"])
print id(a5),"=",a5 #140114778713408 = ['passed_2', 'thing']
```

Argument list and keyword arguments

Putting *args and/or **kwargs as the last items in your function definition’s argument list allows that function to accept an arbitrary number of arguments and/or keyword arguments.

Let's divide our work under five sections:

- Understanding what '*' does in a function call.
- Understanding what '*args' mean in a function definition.
- Understanding what '**' does in a function call.
- Understanding what '**kwargs' mean in a function definition.
- Practical examples of where we use 'args', 'kwargs' and why we use it.

https://agiliq.com/blog/2012/06/understanding-args-and-kwargs/

Understanding what '*' does in a function call.

It unpacked the values in list 'l' as positional arguments. And then the unpacked values were passed to function 'fun' as positional arguments.

```
def f(a,b,c):
    print a,b,c
```

```
f(1, 2, 3)
f(*[1,2,3])
f(1,*[2,3])
f(*[2,3])
```

```
1 2 3
1 2 3
1 2 3
TypeError: f() takes exactly 3 arguments (2 given)
```

Understanding what '*args' mean in a function definition.

```
# * for variable number of arguments
def f(*args):
    print "args = ", args
```

```
f(1,2,3)
f(1)
```

```
args =  (1, 2, 3)
args =  (1,)
```

```
def f(a, *args):
    print "a = ", a,"args = ", args
```

```
f(1,2,3)
f(1)
f(1, *[2,3,4,5])
```

```
a =  1 args =  (2, 3)
a =  1 args =  ()
a =  1 args =  (2, 3, 4, 5)
```

args can receive a tuple of any number of arguments.

The objective here is to see how we get a variable number of arguments in a function and pass these arguments to another function.

```
# can take variable number of arguments stored in a tuple called args
def f3(*args):
    # * here indicates unpacking of args to match the positional arguments in
sum
    print "f3:sum =",sum(*args)

def f2(a,b):
    print "f2: two args are ",a,b

# can take variable number of arguments in form of a tuple called args
def f1(*args):
    # * here indicates unpacking of args tuple to corresponding formals a,b of f2
    f2(*args)
    # f3 is passed a tuple as first postional argument
    f3(args)

f1(1,2)
```

```
f2: two args are  1 2
f3:sum = 3
```

Use case

- With *args you can create more flexible code that accepts a varied amount of non-keyworded arguments within your function.
- In simple words *args is used in cases when you don't know how many arguments are going to be passed to the function by the user.

```
def multiply(*args):
    z = 1
    for num in args:
        z *= num
    print(z)

multiply(4, 5)
multiply(10, 9)
multiply(2, 3, 4)
multiply(3, 5, 10, 6)
```

```
20
90
24
900
```

Understanding what '**' does in a function call.

```
def f(a,b,c):
    print a,b,c

f(1,2,3)                    #1 2 3

def f(a, b=2, c=3):
    print a,b,c

f(1)                        #1 2 3

# ** in function call here indicates unpacking of the dictionary to match the named arguments of f
f(1, **{'b':2, 'c':3})      #1 2 3
f(1, 2, **{'c':3})          #1 2 3
```

Understanding what '**' does in a function definition

** in function definition indicates variable number of named arguments packed in a dictionary kwds and passed in key=value format

```
def f (a, **kwds):
    print "a=",a
    for item in kwds:
        print "item=", item, " val=", kwds[item]

f(1, b=2, c=3, d=4, e=5)
```

```
a= 1
item= c  val= 3
item= b  val= 2
item= e  val= 5
item= d  val= 4
```

Ordering Arguments

When ordering arguments within a function or function call, arguments need to occur in a particular order:

- Formal positional arguments
- Variable args (*args)
- Keyword arguments
- Variable keyword args (**kwargs)

```
def example(arg_1, arg_2, *args, **kwargs):pass

def example2(arg_1, arg_2, *args, kw_1="shark",
kw_2="blobfish", **kwargs):pass
```

Decorators

➔Decorators allow you to make simple modifications to callable objects like functions, methods, or classes.

➔They perform common pre + post function call tasks, such as:

- Caching
- Timing
- Counting function calls
- Access rights

Decorators

A decorator is just another function which takes a functions and returns one. Python makes creating and using decorators a bit cleaner and nicer for the programmer through some syntactic sugar using @.

```
def decorator(f):
    def wrapper(arg):
        'add a wrapper around f'
        return f("Only this thing: " + arg)
    return wrapper
```

```
### code1 ###
@decorator
def function(arg):return arg
print function("hello")
```

=

```
### code2 ###
def function(arg):return arg

function = decorator(function)
print function("hello")
```

Output:

Only this thing: hello

Python decorators(Changing the input)

```
def double_in(old):
    def wrapper(arg):
        return old(2*arg)
    return wrapper
```

```
def function(arg): return arg % 3
function = double_in(function)
print function(2)
```

```
# other way of writing the above code
@double_in
def function (arg):return arg % 3
print function(2)
```

Python decorators(Changing the output)

```
def double_out(old):
    def wrapper(arg):
        return 2 * old(arg)
    return wrapper
```

```
def function(arg): return arg % 3
function = double_out(function)
print function(2)
```

```
# other way of writing the above code
@double_out
def function (arg):return arg % 3
print function(2)
```

Decorators (variable number of args)

```
def decorator(old):
    def wrapper(*args, **kwds):
        # preprocessing
        ret = old(*args, **kwds)
        # postprocessing
        return ret
    return wrapper

@decorator
def function(*args):
    print "Hello World!:", args

function("name1","name2","name3")
```

Decorators are usually generic, so you can't specify the arguments upfront.

```
Hello World!: ('name1', 'name2', 'name3')
```

Decorators(changing input and output both)

```
def decorator(old):
    def wrapper(*args, **kwds):
        # preprocessing
        new_args = []
        for arg in args:
            new_args.append("pre-" + arg)

        #calling the old function
        ret = old(*new_args, **kwds)

        # postprocessing
        new_args = []
        for arg in ret:
            new_args.append(arg + "-post")
        return new_args
    return wrapper
```

```
def function(a, b, c):
    return [a,b,c]

print function("foo", "bar", "baz")
function = decorator(function)
print function("foo", "bar", "baz")
```

```
@decorator
def function(a, b, c):
    return [a,b,c]

print function("foo", "bar", "baz")
```

Output:
['pre-foo-post', 'pre-bar-post', 'pre-baz-post']

Decorators(timing)

```
import time
def time_decorator(old):
   def time_wrapper(*args, **kwds):
        t1 = time.time()
        ret =  old(*args, **kwds)
        t2 = time.time()
        print "time taken to execute method ", old.__name__, " is ",
(t2-t1), 'ms'
     return ret
   return time_wrapper

@time_decorator
def function(a, b, c): return a*b*c

mul = function(27653, 3156, 4298)
print "product is ", mul
```

time taken to execute method function is 5.00679016113e-06 ms
product is 375098786664

Decorators(counter to count number of calls made to a function)

```
def count_decorator(old):
   count = [0] #initialize count once before returning the wrapper function
   def count_wrapper(*args, **kwds):
      count[0] += 1
      print "count is ", count[0]
      return old(*args,  **kwds)
   return count_wrapper

@count_decorator
def function (a,b,c): return a+b+c

function (1,2,3)
function (1,2,3)
function (1,2,3)
function (1,2,3)
function (1,2,3)
```

```
count is  1
count is  2
count is  3
count is  4
count is  5
```

Decorators

Using classes

```
import time
class TIMED(object):
    def __init__(self, f): self.f = f

    def __call__(self, *args):
        start = time.time()
        ret = self.f(*args)
        stop = time.time()
        print "time taken to {0} is {1} ms.".format(self.f.func_name, 1000*(stop-start))
        return ret
```

```
@TIMED
def div(x,y): return x/y

div(938504395, 84775845)

@TIMED
def mul(x,y,z): return x*y*z

mul(27653, 3156, 4298)
```

Output

time taken to div is 0.00190734863281 ms.
time taken to mul is 0.000953674316406 ms.

Decorators
On methods

```
def p_decorate(func):
  def func_wrapper(self):
     return "<p>{0}</p>".format(func(self))
  return func_wrapper

class Person(object):
   def __init__(self):
      self.name = "Bunny"
      self.family = "Foo"

   @p_decorate
   def get_fullname(self):
      return self.name+" "+self.family

my_person = Person()
print my_person.get_fullname() #<p>Bunny Foo</p>
```

Decorators

Multiple decorators

```
def p_decorate(func):
   def func_wrapper(name):
       return "<p>{0}</p>".format(func(name))
   return func_wrapper

def strong_decorate(func):
    def func_wrapper(name):
        return "<strong>{0}</strong>".format(func(name))
    return func_wrapper

def div_decorate(func):
    def func_wrapper(name):
        return "<div>{0}</div>".format(func(name))
    return func_wrapper

@div_decorate
@p_decorate
@strong_decorate
def greet(name):
   return "hello {0}".format(name)
print greet("Bunny") #<div><p><strong>hello Bunny</strong></p></div>
```

=

```
def greet(name):
   return "hello {0}".format(name)
greet = div_decorate(p_decorate(strong_decorate(greet)))
print greet("Bunny") #<div><p><strong>hello Bunny</strong></p></div>
```

Decorators

Passing arguments to decorators

3 decorators(div_decorate, p_decorate, strong_decorate) each with the same functionality but wrapping the string with different tags. Why not have a more general implementation for one that takes the tag to wrap with as a string?

```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))
        return func_wrapper
    return tags_decorator
```

```
@tags("div")
@tags("p")
@tags("strong")
def greet(name):
   return "hello {0}".format(name)

print greet("Bunny") #<div><p><strong>hello Bunny</strong></p></div>
```