

Functions and functional programming

Python is not a functional programming language, but it has a lot of features taken from functional programming languages:

- ▶ closures
- ▶ high order functions and decorators
- ▶ generators
- ▶ coroutines
- ▶ list comprehensions

But First... Let's define a function

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

But First... Let's define a function

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

```
s = add(x, y)  
print s
```

But First... Let's define a function

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

```
s = add(x, y)  
print s
```

That's it!!

But First... Let's define a function

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

```
s = add(x, y)  
print s
```

That's it!!

Not actually... There are a couple of things to notice:

- ▶ python is dynamic
- ▶ parameters don't have a specified type
- ▶ neither do we specify the return type

Evaluation Strategy

- ▶ parameters are just names that point to objects
- ▶ if you pass an immutable object, it looks as if it was passed by value

Evaluation Strategy

- ▶ parameters are just names that point to objects
- ▶ if you pass an immutable object, it looks as if it was passed by value

```
def increment_value(x):  
    x = x + 1  
    print x  # Outputs: 4
```

```
a = 3  
increment_value(y)  
print a  # Outputs: 3
```

- ▶ Oups... It didn't actually work...

Evaluation Strategy

- ▶ parameters are just names that point to objects
- ▶ if you pass a mutable object and that object is modified, the changes are going to be visible in the caller

Evaluation Strategy

- ▶ parameters are just names that point to objects
- ▶ if you pass a mutable object and that object is modified, the changes are going to be visible in the caller

```
def increment(values):  
    for i, v in enumerate(values):  
        values[i] = v + 1
```

```
a = [1, 2, 3]  
increment(a)  
print a  # Outputs: [2, 3, 4]
```

Evaluation Strategy

Functions that mutate their input arguments are:

- ▶ said to have **side effects**
- ▶ are best avoided as they might lead to subtle bugs
- ▶ are needed for doing **in-place** changes to large or expensive objects

Evaluation Strategy

- ▶ this is named **call by sharing**
- ▶ it's the same as in languages such as Java or Ruby
- ▶ though Java people name it pass-by-value
- ▶ while Ruby people name it pass-by-reference

Exercises

1. write a function that takes a list of integers and returns the number of even numbers contained in the list
2. write a function that takes a list of integers and returns a new list containing the even numbers from the list
3. write a function that takes a list of integers and **in-place** removes the odd elements

Exercise 3. Take 1

```
def remove_odd(values):  
    for val in values:  
        if val % 2 != 0:  
            values.remove(val)
```

```
a = [1, 1, 1, 2, 4]  
remove_odd(a) # list will be [1, 2, 4]
```

This is wrong!! Never add/remove elements while iterating!!

Exercise 3. Take 2

```
import copy

def remove_odd(values):
    for val in copy.copy(values):
        if val % 2 != 0:
            values.remove(val)

a = [1, 1, 1, 2]
remove_odd(a)  # list will correctly be [2, 4]
```

This works, but the algorithm is $O(n^2)$

Exercise 3. Take 3

```
def remove_odd(values):  
    values[:] = [v for v in values if v % 2 == 0]
```

```
a = [1, 1, 1, 2]  
remove_odd(a)  # list will correctly be [2, 4]
```

More about this when we talk about list comprehensions.

Default parameter values

```
def increment(x, inc=1):  
    return x + inc
```

```
a = 3  
increment(a)  # returns: 4  
increment(a, 2) # returns: 6
```


Default parameter values

- ▶ you can't have a non-default parameter following a default one. That raises **SyntaxError**
- ▶ default parameter values are assigned at function definition and never change

Default parameter values

- ▶ you can't have a non-default parameter following a default one. That raises **SyntaxError**
- ▶ default parameter values are assigned at function definition and never change

```
default = 1
def foo(x=default)
    print x
```

```
default = 2
foo() # Outputs: 1
```

Keyword arguments

```
def make_symlink(target , link_name):  
    do_stuff  
  
make_symlink(target='/foo' , link_name='/bar')
```

Keyword arguments

```
def make_symlink(target , link_name):  
    do_stuff
```

```
make_symlink(target='/foo' , link_name='/bar')
```

```
make_symlink(link_name='/bar' , target='/foo')
```

Keyword arguments

```
def make_symlink(target , link_name):  
    do_stuff
```

```
make_symlink(target='/foo' , link_name='/bar')
```

```
make_symlink(link_name='/bar' , target='/foo')
```

```
make_symlink('/foo' , link_name='/bar')
```

Keyword arguments

```
def make_symlink(target , link_name):  
    do stuff
```

```
make_symlink(target='/foo' , link_name='/bar')
```

```
make_symlink(link_name='/bar' , target='/foo')
```

```
make_symlink('/foo' , link_name='/bar')
```

```
make_symlink(target='foo' , '/bar')  # SyntaxError !
```

Varargs functions

```
def make_window(parent , *args , **kwargs):  
    print container  
    print args  
    print kwargs
```

```
make_window(1, 2, 3, 4, 5,  
            color='red',  
            modal=False,  
            visible=True)
```

Varargs functions

```
def make_window(parent , *args , **kwargs):  
    print container    # Outputs: 1  
    print args         # Outputs: (2, 3, 4, 5)  
    print kwargs       # Outputs: {'color ': 'red ',  
                        # 'modal ': False ,  
                        # 'visible ': True}  
  
make_window(1, 2, 3, 4, 5,  
            color='red ',  
            modal=False ,  
            visible=True)
```


Variable scope

Python uses function scope:

- ▶ each time a function executes a new local namespace is created
- ▶ the local namespace contains parameters as well as variables defined inside the function

When resolving variables

- ▶ the local namespace is searched
- ▶ If no match is found, the global namespace is searched

Variable scope

```
var = 10
def foo():
    var = 21
foo()
print var  # Outputs: 10
```

Variable scope

```
var = 10
def foo():
    global var
    var = 21
foo()
print var  # Outputs: 21
```

Nested functions

```
def countdown(initial , msg):  
  
    def show_msg():  
        print '%s_%d' % (msg, n)  
  
    for n in xrange(initial , 0, -1):  
        show_msg()
```

```
countdown(2, 'at:')
```

```
# Output:
```

```
# at:2
```

```
# at:1
```

```
# at:0
```

Functions as first class citizens

What this means:

- ▶ functions can be passed as parameters
- ▶ functions can be return values

Functions as first class citizens

What this means:

- ▶ functions can be passed as parameters
- ▶ functions can be return values

```
def compare(x, y):  
    return cmp(x.lower(), y.lower())
```

```
sorted(['B', 'c', 'a'], compare)  
# Returns ['a', 'B', 'c']
```

Closures

A closure is a function that is packaged together with the surrounding environment

Closure example

Closures can be used for delayed evaluation

```
from urllib import urlopen
```

```
def page(url):  
    def get():  
        return urlopen(url).read()  
    return get
```

```
get_python = page('http://python.org')  
get_jython = page('http://jython.org')
```

```
pydata = get_python()  # Fetches http://python.org  
jydata = get_jython()  # Fetches http://jython.org
```


Closure

Closure can be used for preserving state across function calls.

```
def counter(initial_value):  
    counter = initial_value  
    def inc():  
        counter += 1  
        return counter  
    def dec():  
        counter -= 1  
        return counter  
    return inc, dec
```

```
inc, dec = counter(10)  
inc()  # Returns: 11  
inc()  # Returns: 12  
dec()  # Returns: 11
```

Alternative implementation

The classic way of implementing the previous example would have been by using a class

```
class Counter(object):  
  
    def __init__(self, initial_value):  
        self.counter = initial_value  
  
    def inc(self):  
        self.counter += 1  
        return self.counter  
  
    def dec(self):  
        self.counter -= 1  
        return self.counter
```

Exercise

Implement a stack using closures.

```
push , pop = stack()  
  
push(1)  
push(3)  
pop()      # Returns: 3  
push(4)  
pop()      # Returns: 4  
pop()      # Returns: 1
```

High order functions

High order functions are functions that do at least one of

- ▶ take one or more functions as input
- ▶ return a function

High order function example 1

```
def logging_wrapper(func):  
    def wrapped():  
        print 'entering '  
        func()  
        print 'exiting '  
    return wrapped  
  
def foo():  
    print 'fooo '  
  
logged_foo = logging_wrapper(foo)  
logged_foo()  
# entering  
# fooo  
# exiting
```

High order function example 2

```
def logging_wrapper(func):  
    def wrapped():  
        print 'entering '  
        func()  
        print 'exiting '  
    return wrapped
```

```
def foo():  
    print 'fooo '
```

```
foo = logging_wrapper(foo)  
foo()  
# entering  
# fooo  
# exiting
```

Decorators (take 1)

```
def logging_wrapper(func):  
    def wrapped():  
        print 'entering '  
        func()  
        print 'exiting '  
    return wrapped
```

```
@logging_wrapper  
def foo():  
    print 'foo'
```

```
foo()  
# entering  
# foo  
# exiting
```

Decorators (take 2)

```
def logging_wrapper(func):  
    def wrapped(*args, **kwargs):  
        print 'entering '  
        ret_val = func(*args, **kwargs)  
        print 'exiting '  
        return ret_val  
    return wrapped
```

```
@logging_wrapper  
def foo(msg):  
    return 'foo_0%s' % msg
```

```
print foo('bar')  
# entering  
# exiting  
# foo_0 bar
```


Decorators exercise 1

Write a 'timing' decorator that wraps a function and prints how long the function's execution takes

Hint: use the **time** module

```
import time
```

```
started_at = time.time()
```

```
# do stuff
```

```
print time.time() - started_at
```

Decorators exercise 2

Write a decorator that keeps track of how many times decorated functions are being called

```
@count_calls
def foo():
    print 'booo'
```

```
@count_calls
def bar():
    print 'boohoo_yourself'
```

```
foo()
bar()
foo()
get_call_count(foo)  # Returns: 2
get_call_count(bar)  # Returns: 1
```

Generators

- ▶ a generator is a function that produces a sequence of values.
- ▶ the sequence can be then consumed with a **for** loop or by explicitly calling **next** on the returned generator object

Generators

- ▶ a generator is a function that produces a sequence of values.
- ▶ the sequence can be then consumed with a **for** loop or by explicitly calling **next** on the returned generator object

```
def my_range( first , last ):  
    i = first  
    while i < last:  
        yield i  
        i += 1
```

```
for x in my_range(0, 3):  
    print x
```

Outputs: 0 1 2

Generators

- ▶ a generator is a function that produces a sequence of values.
- ▶ the sequence can be then consumed with a **for** loop or by explicitly calling **next** on the returned generator object

```
def my_range( first , last ):  
    i = first  
    while i < last:  
        yield i  
        i += 1
```

```
print sum(my_range(0 , 3))  # Outputs: 4
```

Generators

- ▶ a generator is a function that produces a sequence of values.
- ▶ the sequence can be then consumed with a **for** loop or by explicitly calling **next** on the returned generator object

```
def my_range(first , last ):
    i = first
    while i < last:
        yield i
        i += 1
```

```
gen = my_range(0, 3)
print gen.next()  # Outputs 0
print gen.next()  # Outputs 1
print gen.next()  # Outputs 2
print gen.next()  # raised StopIteration !
```

Generators

```
def my_range(first , last ):
    i = first
    while i < last:
        yield i
        i += 1
```

```
gen = my_range(0, 3)
while True:
    try:
        print gen.next()
    except StopIteration:
        break
```

Endless generators

```
import random

def random_generator():
    while True:
        yield random.random()

random_gen = random_generator()
for rand_nr in random_gen:
    print nr
    if rand_nr > 0.5:
        break
random_gen.close()
```


Exercises

1. Write a generator that takes an integer parameter and yields Fibonacci numbers smaller than the given parameter.
2. Having a binary tree encoded as a tuple (label, left, right) write a generator that yields the labels in pre-order (root, left, right). Write both an iterative and recursive implementation.

Example:

```
tree= ( 'b' ,  
        ( 'a' ,  
          ( 'q' , None , None ) ,  
          None ) ,  
        ( 'z' ,  
          ( 'c' , None , None ) ,  
          ( 'zz' , None , None ) ) )
```

```
for label in iterate(tree):  
    print label
```

Output: b, a, q, z, c, zz

Iterative solution

```
def iterator(bin_tree):  
    node_stack = [bin_tree]  
    while node_stack:  
        node = node_stack.pop()  
        if not node:  
            continue  
        label, left, right = node  
        yield label  
        node_stack.append(right)  
        node_stack.append(left)
```

Recursive solution

```
def iterator(bin_tree):  
    if not bin_tree:  
        return  
    label, left, right = bin_tree  
    yield label  
    for label in iterator(left):  
        yield label  
    for label in iterator(right):  
        yield label
```

Exercises pt. 2

1. write a generator that flattens a list that might have any level of nesting.

Example:

```
for x in flatten([1, [2, 3], [[4, 5], 6, [[[7]]]]):  
    print x
```

Output: 1 2 3 4 5 6

Hint: use `isinstance` for checking if an object is a list or not

Exercise solution

```
def flatten(lst):  
    for item in lst:  
        if isinstance(item, list):  
            for sub_item in flatten(item):  
                yield sub_item  
        else:  
            yield item
```

Piping generators

```
def grep(lines, word):  
    for line in lines:  
        if word in line:  
            yield line  
  
f = open('/etc/passwd')  
lines = grep(f, 'foo')  
lines = grep(lines, 'bar')  
for line in lines:  
    print line  
f.close()
```

Equivalent: cat some_file | grep foo | grep bar

Piping generators

```
def grep(lines , word):  
    for line in lines:  
        if word in line:  
            yield line  
  
f = open('/etc/passwd')  
try:  
    lines = grep(f, 'foo')  
    lines = grep(lines, 'bar')  
    for line in lines:  
        print line  
finally:  
    f.close()
```

Piping generators

```
def grep(lines , word):  
    for line in lines:  
        if word in line:  
            yield line  
  
with open('/etc/passwd') as f:  
    lines = grep(f, 'foo')  
    lines = grep(lines, 'bar')  
    for line in lines:  
        print line
```


Piping generators

```
def grep(lines , word):  
    for line in lines:  
        if word in line:  
            yield line  
  
for ln in grep(grep(open('passwd'), 'foo'), 'bar'):  
    print ln
```

List comprehensions

```
nums = [1, 2, 3, 4, 5]  
times_two = [x * 2 for x in nums]
```

```
print times_two
```

```
# Outputs: [2, 4, 6, 8, 10]
```

List comprehensions

```
nums = [1, 2, 3, 4, 5]  
times_two = [x * 2 for x in nums if x % 2 == 0]
```

```
print times_two
```

```
# Outputs: [4, 8]
```

Exercise

- ▶ write a list comprehension statement that given a list of string excludes the ones longer than 5 characters and makes the shorter ones uppercase

List comprehensions

```
sentences = [ 'mama_are_mere', 'tata_are_pere' ]  
words = []  
for sentence in sentences:  
    for w in sentence.split():  
        words.append(w.upper())  
  
print words  
# Outputs: [ 'MAMA', 'ARE', 'MERE', 'TATA',  
            'ARE', 'PERE' ]
```

List comprehensions

```
sentences = ['mama_are_mere', 'tata_are_pere']  
  
words = [w.upper() for sentence in sentences  
          for w in sentence.split()]  
  
print words  
# Outputs: ['MAMA', 'ARE', 'MERE', 'TATA',  
            'ARE', 'PERE']
```

Exercises

- ▶ Write a list comprehension statement that flattens a list of lists but skips nested lists that have a single element

Example :

`[[1, 2], [3], [4, 5]]` should be transformed to `[1, 2, 4, 5]`

List comprehensions

List comprehensions can be used for building generators as well

```
sentences = [ 'mama_are_mere' , 'tata_are_pere' ]
```

```
words = (w.upper() for sentence in sentences  
          for w in sentence.split())
```

```
for word in words:  
    print words
```


In-place list processing

```
numbers = [1, 2, 3, 4, 5]
numbers[:] = [x * x for x in numbers if x % 2 == 0]

print numbers
# Outputs: [4, 16]
```

Using list comprehensions for initializing container objects

```
words = ['foo', 'barbaz', 'oups']  
d = dict((w, len(w)) for w in words)  
  
print d  # Outputs: {'foo': 3, 'barbaz': 6, 'oups':
```

Lambdas

- ▶ lambdas are anonymous functions
- ▶ can have a single expression
- ▶ use-full for short callbacks

lambda $x, y: x + y$

Using lambdas with filter

```
numbers = [-7, 3, 4, -8, 9]
```

```
positive_nums = [x for x in numbers if x >= 0]
```

is the equivalent of

```
positive_nums = filter(lambda x: x >= 0, numbers)
```

Using lambdas with map

```
numbers = [-7, 3, 4, -8, 9]
```

```
squares = [x * x for x in numbers]
```

is the equivalent of

```
squares = map(lambda x: x * x, numbers)
```

Using lambdas with reduce

```
numbers = [1, 2, 3, 4]

print reduce(lambda x, y: x + y, numbers)
# Outputs: 10
```

Exercise

Write a function that takes a list of words. For each word:

- ▶ if is shorter than 5 chars, remove the vowels
- ▶ if is longer or equal to 5 chars, remove the consonants

Order the resulting names alphabetically, and then return the concatenated string.

Try to be as 'functional' as possible.

Solution

Write a filter that takes a word and:

- ▶ if is shorter than 5 chars, remove the vowels
- ▶ if is longer or equal to 5 chars, remove the consonants

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

```
w = 'anamaria'
```

```
filter(lambda c: (len(w)<5)^(c in vowels), w)  
# Outputs: aaaia
```


Solution

Write a filter that takes a word and:

- ▶ if is shorter than 5 chars, remove the vowels
- ▶ if is longer or equal to 5 chars, remove the consonants

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

```
w = 'asfe'
```

```
filter(lambda c: (len(w)<5)^(c in vowels), w)
```

```
# Outputs: sf
```

Solution

Write a map that takes a list of words. For each word:

- ▶ if is shorter than 5 chars, remove the vowels
- ▶ if is longer or equal to 5 chars, remove the consonants

```
vowels = ['a', 'e', 'i', 'o', 'u']  
list_of_words = ['asfe', 'anamaria']
```

```
map(  
    lambda w: filter(  
        lambda c: (len(word)<5)^(c in vowels), w),  
    list_of_words)
```

Outputs: ['sf', 'aaaia']

Solution

Now we sort the result of the previous map

```
vowels = ['a', 'e', 'i', 'o', 'u']  
list_of_words = ['anamaria', 'asfe']  
  
sorted(map(  
    lambda w: filter(  
        lambda c: (len(w)<5)^(c in vowels), w),  
    list_of_words))
```

Outputs: ['aaaia', 'sf']

Final Solution

```
vowels = ['a', 'e', 'i', 'o', 'u']  
list_of_words = ['anamaria', 'asfe']  
  
reduce(  
    lambda x, y: x + y,  
    sorted(map(  
        lambda w: filter(  
            lambda c: (len(w) < 5) ^ (c in vowels), w),  
        list_of_words)))
```

Outputs: 'aaaiasf'