**Generators:**

Generators are very easy to implement, but a bit difficult to understand.

Generators are used to create iterators, but with a different approach. Generators are simple functions which return an iterable set of items, one at a time, in a special way.

When an iteration over a set of item starts using the for statement, the generator is run. Once the generator's function code reaches a "yield" statement, the generator yields its execution back to the for loop, returning a new value from the set. The generator function can generate as many values (possibly infinite) as it wants, yielding each one in its turn.

Here is a simple example of a generator function which returns 7 random integers:

­import random

def lottery():

# returns 6 numbers between 1 and 40

for i in range(6):

yield random.randint(1, 40)

# returns a 7th number between 1 and 15

yield random.randint(1,15)

for random\_number in lottery():

print("And the next number is... %d!" %(random\_number))

# Output:

# And the next number is... 40!

# And the next number is... 36!

# And the next number is... 24!

# And the next number is... 9!

# And the next number is... 14!

# And the next number is... 3!

# And the next number is... 3!

This function decides how to generate the random numbers on its own, and executes the yield statements one at a time, pausing in between to yield execution back to the main for loop.

**Generators - Exercise:**

Write a generator function which returns the Fibonacci series. They are calculated using the following formula: The first two numbers of the series are always equal to 1, and each consecutive number returned is the sum of the last two numbers. Hint: Can you use only two variables in the generator function? Remember that assignments can be done simultaneously. The code

a = 1

b = 2

a, b = b, a

will simultaneously switch the values of a and b.

# fill in this function

def fib():

pass

# This is a null statement which does nothing when executed.

# This is useful as a placeholder.

# testing code

import types

if type(fib()) == types.GeneratorType:

print("Good. The fib function is a generator.")

counter = 0

for n in fib():

print(n)

counter += 1

if counter == 10:

break

My solution:

def fib():

a = 1

b = 1

for i in range(2):

yield 1

while 1:

old\_b = b

b = a + b

a = old\_b

yield b

Solution:

def fib():

a, b = 1, 1

while 1:

yield a

a, b = b, a + b

Output:

Good. The fib function is a generator.

1

1

2

3

5

8

13

21

34

55

**List Comprehensions:**

List comprehension is a very powerful tool, which creates a new list based on another list, in a single, readable line.

For example, let's say we need to create a list of integers which specify the length of each word in a certain sentence, but only if the word is not the word "the".

sentence = "the quick brown fox jumps over the lazy dog"

words = sentence.split()

word\_lengths = []

for word in words:

if word != "the":

word\_lengths.append(len(word))

print(words)

print(word\_lengths)

#Output:

['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

[5, 5, 3, 5, 4, 4, 3]

Using a list comprehension, we could simplify this process to this notation:

sentence = "the quick brown fox jumps over the lazy dog"

words = sentence.split()

word\_lengths = [len(word) for word in words if word != "the"]

print(words)

print(word\_lengths)

# The output is the same as the previous version of this code

**List Comprehensions - Exercise:**

Using a list comprehension, create a new list called "newlist" out of the list "numbers", which contains only the positive numbers from the list, as integers.

numbers = [34.6, -203.4, 44.9, 68.3, -12.2, 44.6, 12.7]

newlist = []

print(newlist)

Solution:

numbers = [34.6, -203.4, 44.9, 68.3, -12.2, 44.6, 12.7]

newlist = [int(x) for x in numbers if x > 0]

print(newlist)

Output:

[34, 44, 68, 44, 12]

**Multiple Function Arguments:**

Every function in Python receives a predefined number of arguments, if declared normally, like this:

def myfunction(first, second, third):

# do something with the 3 variables...

It is possible to declare functions which receive a variable number of arguments, using the following syntax:

def foo(first, second, third, \*therest):

print("First: %s" % first)

print("Second: %s" % second)

print("Third: %s" % third)

print("And all the rest... %s" % list(therest))

The "therest" variable is a list of variables, which receives all arguments which were given to the "foo" function after the first 3 arguments. So calling "foo(1,2,3,4,5)" will print out:

First: 1

Second: 2

Third: 3

And all the rest... [4, 5]

It is also possible to send functions arguments by keyword, so that the order of the argument does not matter, using the following syntax.

def bar(first, second, third, \*\*options):

if options.get("action") == "sum":

print("The sum is: %d" %(first + second + third))

if options.get("number") == "first":

return first

result = bar(1, 2, 3, action = "sum", number = "first")

print("Result: %d" %(result))

# Output:

# The sum is: 6

# Result: 1

The "bar" function receives 3 arguments. If an additional "action" argument is received, and it instructs on summing up the numbers, then the sum is printed out. Alternatively, the function also knows it must return the first argument, if the value of the "number" parameter, passed into the function, is equal to "first".