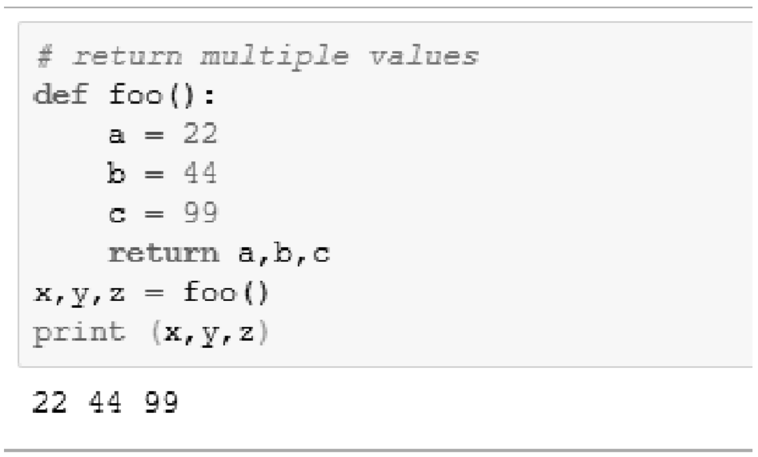
Python World ++

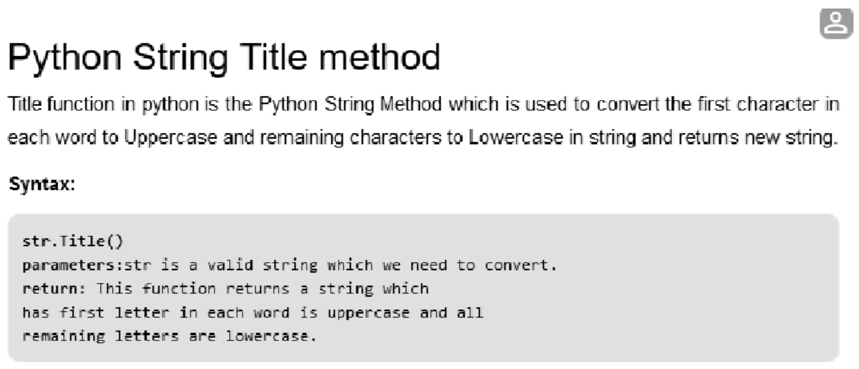
Advanced Technique and Tools

Functions return multiple values



```
# return multiple values
def foo():
    a = 22
    b = 44
    c = 99
    return a,b,c
x,y,z = foo()
print (x,y,z)
```

```
22 44 99
```



Python String Title method

Title function in python is the Python String Method which is used to convert the first character in each word to Uppercase and remaining characters to Lowercase in string and returns new string.

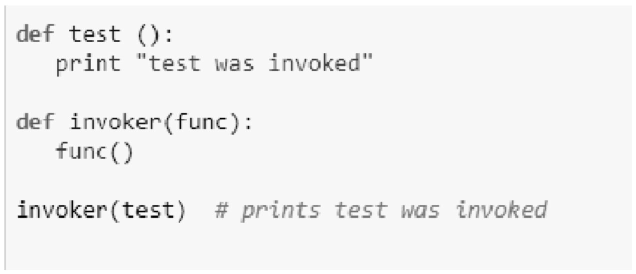
Syntax:

```
str.Title()
parameters:str is a valid string which we need to convert.
return: This function returns a string which
has first letter in each word is uppercase and all
remaining letters are lowercase.
```

Passing Functions to functions

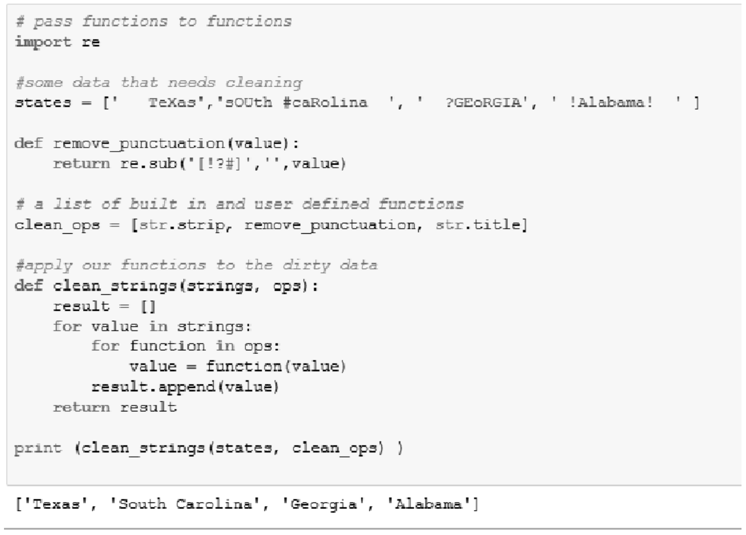
Functions are first-class objects in python. you can pass them around, include them in dicts, lists, etc. Just don't include the parenthesis after the function name.

Example, for a function named myfunction: myfunction means the function itself, myfunction() means to call the function and get its return value instead.



```
def test ():
   print "test was invoked"

def invoker(func):
   func()

invoker(test)  # prints test was invoked
```

Passing Functions to functions



```
# pass functions to functions
import re

#some data that needs cleaning
states = ['   TeXas','sOUth #caRolina  ', '  ?GEoRGIA', ' !Alabama!  ' ]

def remove_punctuation(value):
    return re.sub('[!?#]','',value)

# a list of built in and user defined functions
clean_ops = [str.strip, remove_punctuation, str.title]

#apply our functions to the dirty data
def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result

print (clean_strings(states, clean_ops) )
```

```
['Texas', 'South Carolina', 'Georgia', 'Alabama']
```

Lambda Functions

- **The lambda function** is used for creating small, one-time and **anonymous function** objects in **Python**.
- **lambda** operator can have any number of arguments, but it can have only one expression.
- It cannot contain any statements and it returns a **function** object which can be assigned to any variable.

map, filter and reduce

map

Mostly lambda functions are passed as parameters to a function which expects a function objects as parameter like map, reduce, filter functions

map

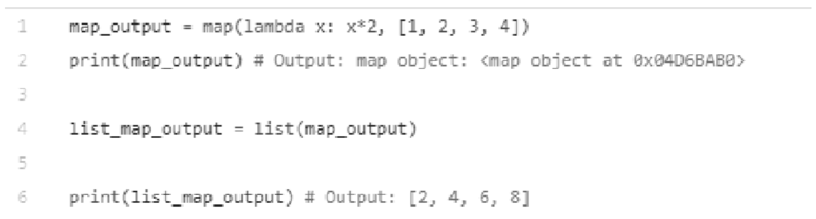
Basic syntax

```
map(function_object, iterable1, iterable2,...)
```

map functions expects a function object and any number of iterables like list, dictionary, etc. It executes the function_object for each element in the sequence and returns a list of the elements modified by the function object.

In Python3, *map* function returns an *iterator* or *map object* which gets lazily evaluate

Neither we can access the elements of the map object with index nor we can use *len()* to find the length of the map object – BUT, ee can force convert the map output i.e. the map object to list

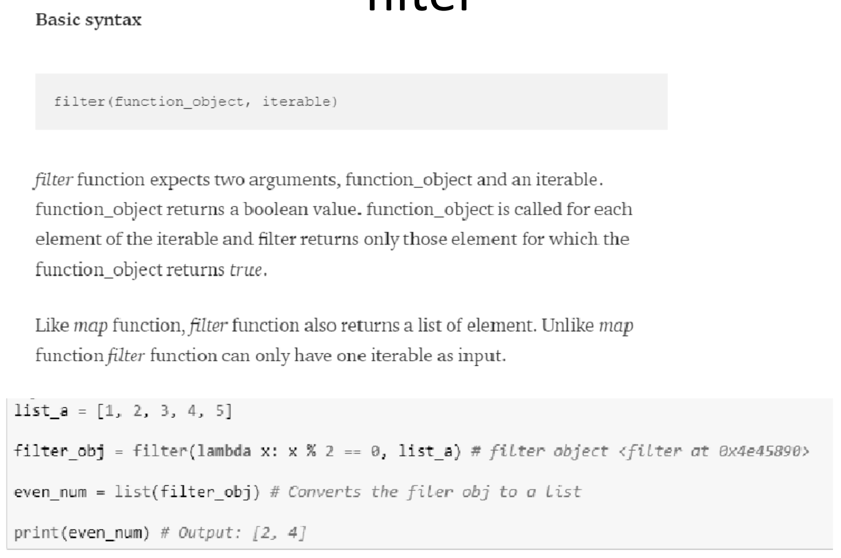


```
map_output = map(lambda x: x*2, [1, 2, 3, 4])
print(map_output) # Output: map object: <map object at 0x04D6BAB0>

list_map_output = list(map_output)

print(list_map_output) # Output: [2, 4, 6, 8]
```

Iterating over a dictionary using map and lambda

```
# Iterating over a dictionary using map and lambda

dict_a = [{'name': 'python', 'points': 10}, {'name': 'java', 'points': 8}]

map(lambda x : x['name'], dict_a)
# Output: ['python', 'java']

map(lambda x : x['points']*10,  dict_a)
# Output: [100, 80]

map(lambda x : x['name'] == "python", dict_a)
# Output: [True, False]
```

filter



Basic syntax

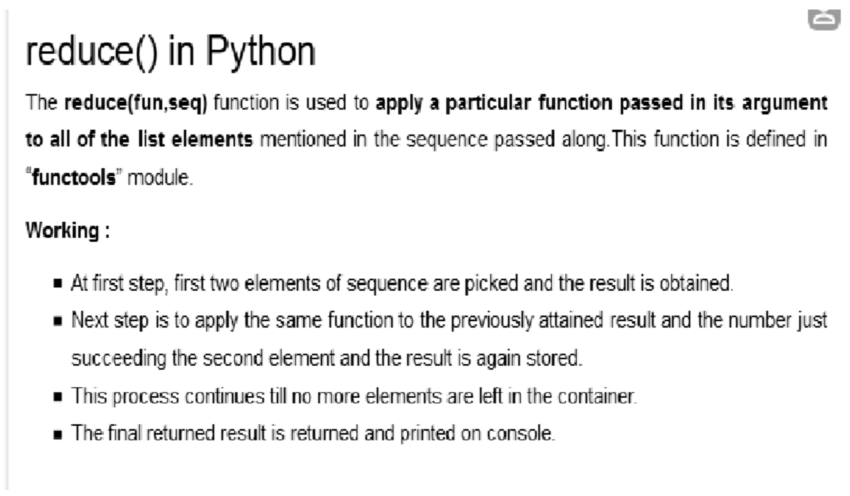
```
filter(function_object, iterable)
```

filter function expects two arguments, function_object and an iterable. function_object returns a boolean value. function_object is called for each element of the iterable and filter returns only those element for which the function_object returns *true*.

Like *map* function, *filter* function also returns a list of element. Unlike *map* function *filter* function can only have one iterable as input.

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

filter_obj = filter(lambda x: x % 2 == 0, list_a) # filter object <filter at 0x4e45890>

even_num = list(filter_obj) # Converts the filer obj to a list

print(even_num) # Output: [2, 4]
```

reduce



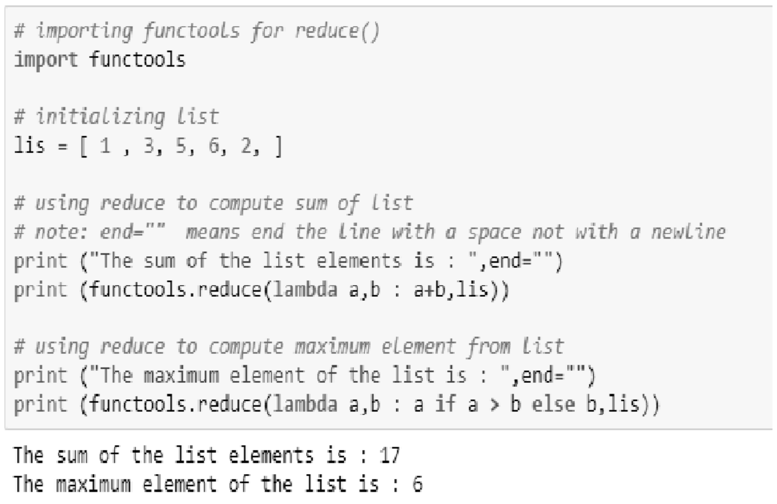
reduce() in Python

The **reduce(fun,seq)** function is used to **apply a particular function passed in its argument to all of the list elements** mentioned in the sequence passed along.This function is defined in "**functools**" module.

Working :

- At first step, first two elements of sequence are picked and the result is obtained.
- Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
- This process continues till no more elements are left in the container.
- The final returned result is returned and printed on console.

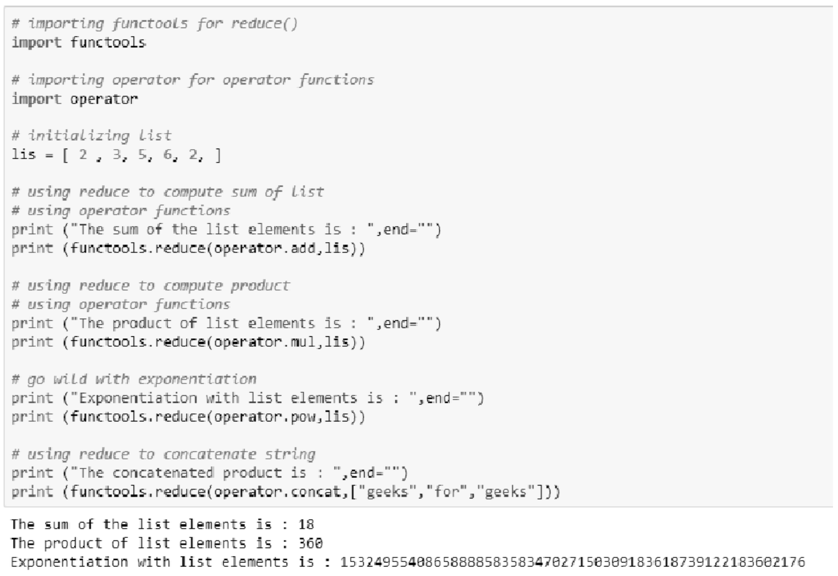
reduce



```
# importing functools for reduce()
import functools

# initializing list
lis = [ 1 , 3, 5, 6, 2, ]

# using reduce to compute sum of list
# note: end=""  means end the line with a space not with a newline
print ("The sum of the list elements is : ",end="")
print (functools.reduce(lambda a,b : a+b,lis))

# using reduce to compute maximum element from list
print ("The maximum element of the list is : ",end="")
print (functools.reduce(lambda a,b : a if a > b else b,lis))
```

```
The sum of the list elements is : 17
The maximum element of the list is : 6
```

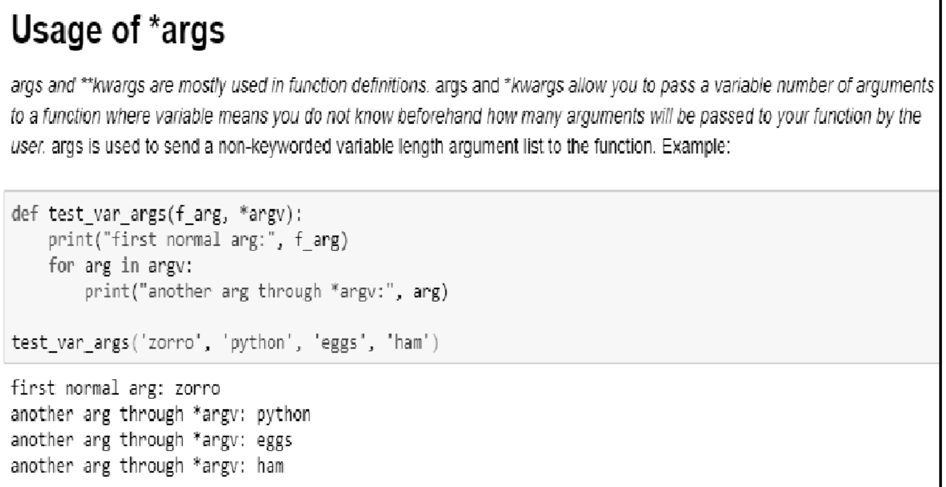
reduce with operator functions



```
# importing functools for reduce()
import functools

# importing operator for operator functions
import operator

# initializing list
lis = [ 2 , 3, 5, 6, 2, ]

# using reduce to compute sum of list
# using operator functions
print ("The sum of the list elements is : ",end="")
print (functools.reduce(operator.add,lis))

# using reduce to compute product
# using operator functions
print ("The product of list elements is : ",end="")
print (functools.reduce(operator.mul,lis))

# go wild with exponentiation
print ("Exponentiation with list elements is : ",end="")
print (functools.reduce(operator.pow,lis))

# using reduce to concatenate string
print ("The concatenated product is : ",end="")
print (functools.reduce(operator.concat,["geeks","for","geeks"]))
```

```
The sum of the list elements is : 18
The product of list elements is : 360
Exponentiation with list elements is : 1532495540865888858358347027150309183618739122183602176
```

*args



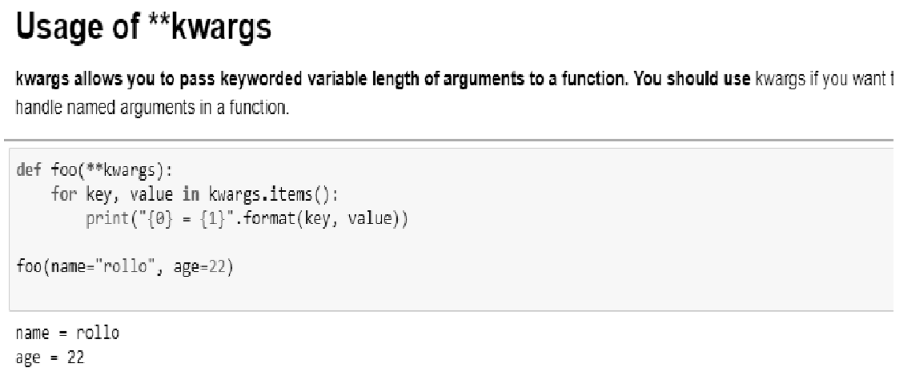
Usage of *args

*args and **kwargs are mostly used in function definitions. args and *kwargs allow you to pass a variable number of arguments to a function where variable means you do not know beforehand how many arguments will be passed to your function by the user.* args is used to send a non-keyworded variable length argument list to the function. Example:

```
def test_var_args(f_arg, *argv):
    print("first normal arg:", f_arg)
    for arg in argv:
        print("another arg through *argv:", arg)

test_var_args('zorro', 'python', 'eggs', 'ham')
```

```
first normal arg: zorro
another arg through *argv: python
another arg through *argv: eggs
another arg through *argv: ham
```

**kwargs

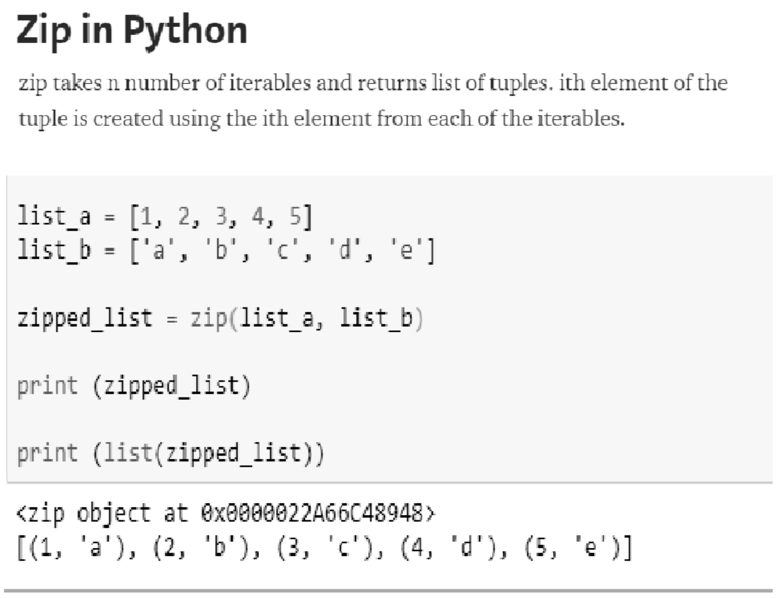


Usage of **kwargs

kwargs allows you to pass keyworded variable length of arguments to a function. You should use kwargs if you want t
handle named arguments in a function.

```
def foo(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

foo(name="rollo", age=22)
```

```
name = rollo
age = 22
```



Zip in Python

zip takes n number of iterables and returns list of tuples. ith element of the tuple is created using the ith element from each of the iterables.

```
list_a = [1, 2, 3, 4, 5]
list_b = ['a', 'b', 'c', 'd', 'e']

zipped_list = zip(list_a, list_b)

print (zipped_list)

print (list(zipped_list))
```

```
<zip object at 0x0000022A66C48948>
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

Iterators, Iterables, Generators

Iteration

Iteration is a general term for taking each item of something, one after another. Any time you use a loop, explicit or implicit, to go over a group of items, that is iteration.

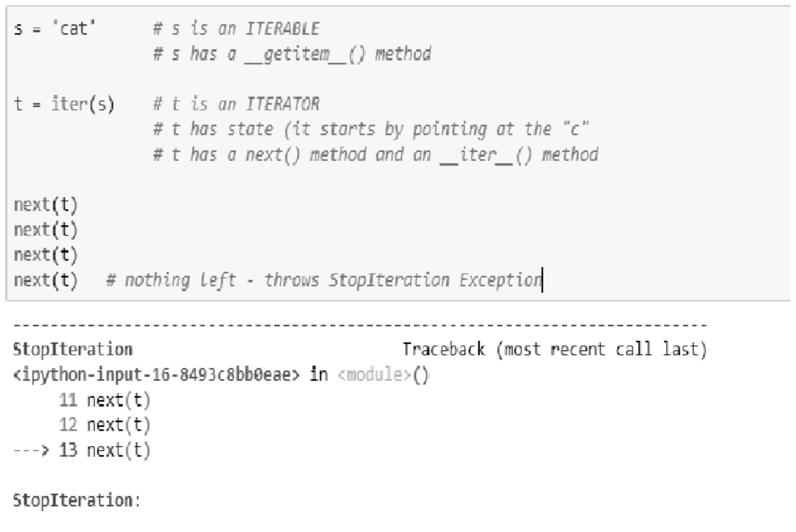
In Python, **iterable** and **iterator** have specific meanings.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, Python **iterator object** must implement two special methods, `__iter__()` and `__next__()`, collectively called the **iterator protocol**.

An object is called **iterable** if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.

The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.



```
s = 'cat'      # s is an ITERABLE
               # s has a __getitem__() method

t = iter(s)    # t is an ITERATOR
               # t has state (it starts by pointing at the "c"
               # t has a next() method and an __iter__() method

next(t)
next(t)
next(t)
next(t)   # nothing left - throws StopIteration Exception
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-16-8493c8bb0eae> in <module>()
     11 next(t)
     12 next(t)
---> 13 next(t)

StopIteration:
```

for loops – can do iteration

```
>>> for element in my_list:
...     print(element)
...
4
7
0
3
```

any iterable can be used here

In fact the `for` loop can iterate over any iterable. Let's take a closer look at how the `for` loop is actually implemented in Python.

```
for element in iterable:
    # do something with element
```

Is actually implemented as.

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

So internally, the `for` loop creates an iterator object, `iter_obj` by calling `iter()` on the iterable.

Building Your Own Iterator in Python

Building an iterator from scratch is easy in Python. We just have to implement the methods `__iter__()` and `__next__()`.

The `__iter__()` method returns the iterator object itself. If required, some initialization can be performed.

The `__next__()` method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise `StopIteration`.

```
class PowTwo:
    def __init__(self, max=0):
        self.max = max

    def __iter__(self):
        self.n = 0  #upon creation we set counter and return instance (self)
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration

c1 = PowTwo(5)
for k in c1:
    print (k)
```

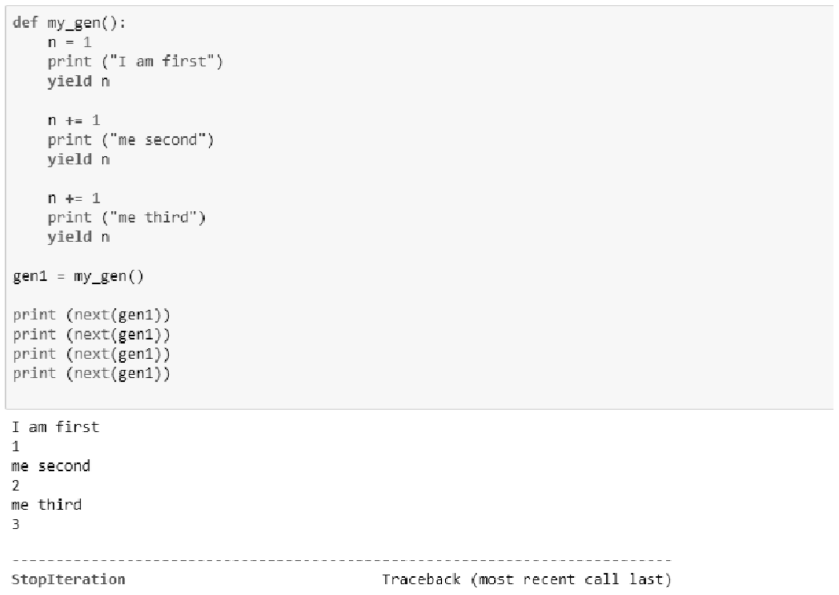
```
1
2
4
8
16
32
```

Generators

There is a lot of overhead in building an iterator in Python; we have to implement a class with __iter__() and __next__() method, keep track of internal states, raise StopIteration when there was no values to be returned etc. -- This is both lengthy and counter intuitive. Generator comes into rescue in such situations.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

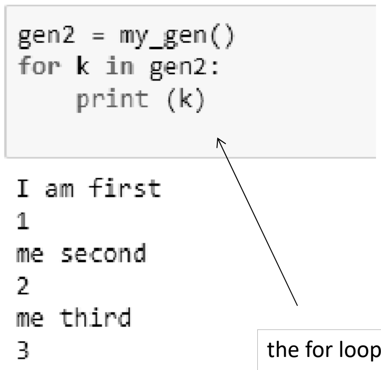
Generator function contains one or more yield statement.
• When called, it returns an object (iterator) but does not start execution immediately.
• Methods like **iter**() and **next**() are implemented automatically. So we can iterate through the items using next().
• Once the function yields, the function is paused and the control is transferred to the caller.
• Local variables and their states are remembered between successive calls.
• Finally, when the function terminates, StopIteration is raised automatically on further calls.



```
def my_gen():
    n = 1
    print ("I am first")
    yield n

    n += 1
    print ("me second")
    yield n

    n += 1
    print ("me third")
    yield n

gen1 = my_gen()

print (next(gen1))
print (next(gen1))
print (next(gen1))
print (next(gen1))
```

```
I am first
1
me second
2
me third
3

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
```

Generators work best with for loops



```
gen2 = my_gen()
for k in gen2:
    print (k)
```

```
I am first
1
me second
2
me third
3
```

the for loop catches the StopIteration Exception

Comprehensions

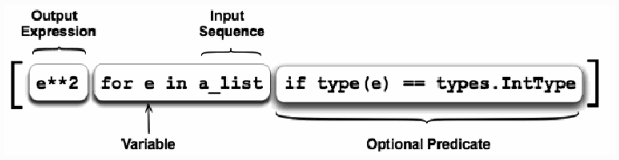
List Comprehensions

A list comprehension consists of the following parts:

- An Input Sequence.
- A Variable representing members of the input sequence.
- An Optional Predicate expression.
- An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate.

Say we need to obtain a list of all the integers in a sequence and then square them:

```
a_list = [1, '4', 9, 'a', 0, 4]

squared_ints = [ e**2 for e in a_list if type(e) == types.IntType ]

print squared_ints
# [ 1, 81, 0, 16 ]
```



LIST COMPREHENSIONS

List comprehensions are the best known and most widely used. Let's start with an example.

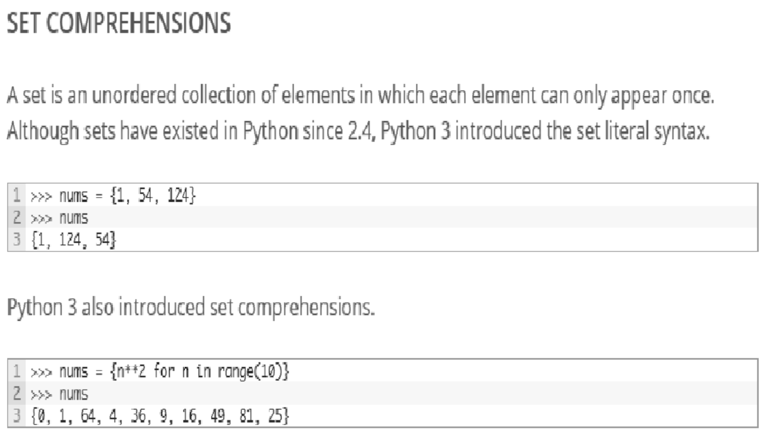
A common programming task is to iterate over a list and transform each element in some way, e.g:

```
>>> squares = []
>>> for num in range(10):
        squares.append(num**2)
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The basic syntax for list comprehensions is this: **[EXPRESSION FOR ELEMENT IN SEQUENCE]**.

Set Comprehensions



SET COMPREHENSIONS

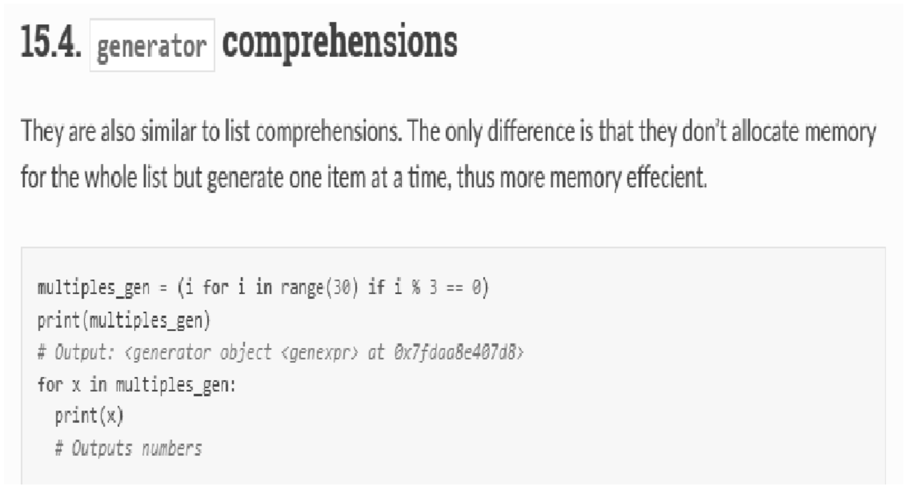
A set is an unordered collection of elements in which each element can only appear once. Although sets have existed in Python since 2.4, Python 3 introduced the set literal syntax.

```
>>> nums = {1, 54, 124}
>>> nums
{1, 124, 54}
```

Python 3 also introduced set comprehensions.

```
>>> nums = {n**2 for n in range(10)}
>>> nums
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

generator comprehensions



15.4. `generator` **comprehensions**

They are also similar to list comprehensions. The only difference is that they don't allocate memory for the whole list but generate one item at a time, thus more memory effecient.

```
multiples_gen = (i for i in range(30) if i % 3 == 0)
print(multiples_gen)
# Output: <generator object <genexpr> at 0x7fdaa8e407d8>
for x in multiples_gen:
  print(x)
  # Outputs numbers
```

Generator Comprehension

The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses. The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time. They are kind of lazy, producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
my_list = [1,3,6,10]

#square brackets -  list comprehension
list1 = [x**2 for x in my_list]
print (list1)

#round braces for a LAZY generator comprehension
zz = (x**2 for x in my_list)

print (zz)  #zz is a generator object

print (next(zz))

print (list(zz))  #here we get was is left of our generator
```

```
[1, 9, 36, 100]
<generator object <genexpr> at 0x0000022A66CB6678>
1
[9, 36, 100]
```

Another common task is to filter a list and create a new list composed of only the elements that pass a certain condition. The next snippet constructs a list of every number from 0 to 9 that has a modulus with 2 of zero, i.e. every even number.

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
```

Using an IF-ELSE construct works slightly differently to what you might expect. Instead of putting the ELSE at the end, you need to use the ternary operator – **x if y else z.**

The following list comprehension generates the squares of even numbers and the cubes of odd numbers in the range 0 to 9.

```
>>> [x**2 if x % 2 == 0 else x**3 for x in range(10)]
[0, 1, 4, 27, 16, 125, 36, 343, 64, 729]
```