Iterators

An iterator is an object that contains a countable number of values. An iterator is an object that can be iterated upon, meaning that you can traverse through all the values. Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

- All these objects have a iter() method which is used to get an iterator.
- The next() method also allows you to do operations, and must return the next item in the sequence.
- Even strings are iterable objects, and can return an iterator.

In [1]:
```
1 mytuple = ("apple", "banana", "cherry")
2 myit = iter(mytuple)
3 myit
```

Out[1]:
```
<tuple_iterator at 0x7f81f40540d0>
```

In [2]:
```
1 print(next(myit))
2 print(next(myit))
3 print(next(myit))
4 print(next(myit))
5 print(next(myit))
6
7 print(next(myit))
8 print(next(myit))
```

```
apple
banana
cherry

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-2-94c4be05c624> in <module>
      2 print(next(myit))
      3 print(next(myit))
----> 4 print(next(myit))
      5 print(next(myit))
      6

StopIteration:
```

StopIteration: The example above would continue forever if you had enough next() statements, or if it was used in a for loop. To prevent the iteration to go on forever, we can use the StopIteration statement.

Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Why functions?

- code resuability
- Code Modularity

- Namespaces: The interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

Functions are **first class objects**, can be used as arguments: We can pass them to other functions as arguments, return them from other functions as values, and store them in variables and data structures.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, etc.

In [3]:
```
# Creating a function: use def keyword

def my_fun():
    print("This is a function")

my_fun()
```

```
This is a function
```

In [4]:
```
# Calling a function: use the function name followed by parenthesis

def my_fun():
  print("This is a function")

my_fun()
```

```
This is a function
```

Arguments: Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

Arguments are often shortened to args in Python documentations.

The terms parameter and argument can be used for the same thing: information that are passed into a function.

From a function's perspective:

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.

In [5]:
```
# Here, 'name' is the parameter and 'Ram', 'Shyam' and 'Sita' are arguments.
def my_fun(name):
  print(name + " is an employee.")

my_fun("Ram")
my_fun("Shyam")

b = "Sita"
my_fun(b)
```

```
Ram is an employee.
Shyam is an employee.
Sita is an employee.
```

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

In [6]:
```
def my_fun(name, age):
  print(name + ' ' + str(age))

my_fun('Ram', 19)
# my_fun('Shyam')
# my_fun('Sita', 20, 'Hello')
```

```
Ram 19
```

Arbitrary Arguments, *args: If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition. This way the function will receive a tuple of arguments, and can access the items accordingly

In [7]:
```
def my_function(*names):
  print("The given names are " + names[1])

my_function("Emil", "Tobias", "Linus")
my_function("Emil", "Ram", "Tobias", "Linus", "Shyam", "Sita")
```

```
The given names are Tobias
The given names are Ram
```

Keyword Arguments: You can also send arguments with the key = value syntax. This way the order of the arguments does not matter.

In [8]:
```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

```
The youngest child is Linus
```

Arbitrary Keyword Arguments, **kwargs: If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition. This way the function will receive a dictionary of arguments, and can access the items accordingly:

In [9]:
```
def my_function(**kid):
    print("His last name is " + kid["lname"] + ' ' + str(kid['age']))

my_function(fname = "Tobias", lname = "Refsnes", age = 10)
```

```
His last name is Refsnes 10
```

Default Parameter Value: The following example shows how to use a default parameter value. If we call the function without argument, it uses the default value:

In [10]:
```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

In [11]:
```
def my_function(age = "Not Defined"):
  print("The age is " + str(age))

my_function(18)
my_function(25)
my_function()
my_function(30)
```

```
The age is 18
The age is 25
The age is Not Defined
The age is 30
```

Passing a List as an Argument: You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function. E.g. if you send a List as an argument, it will still be a List when it reaches the function.

In [12]:
```
def my_function(food):
    print(food)

fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

```
['apple', 'banana', 'cherry']
```

In [13]:
```
def my_function(food):
    for x in food:
        print(x)

fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

```
apple
banana
cherry
```

In [14]:
```
def my_fun1(job):
    for x in job:
        print(x)

def my_fun2(job):
    print(job.keys())

def my_fun3(job):
    print(job.values())

def my_fun4(job):
    for x in job:
        print(x, job[x])

my_job = {'Ram': 'Advocate', 'Sita': 'Teacher', 'Shyam': 'Doctor'}

my_fun1(my_job)
print()

my_fun2(my_job)
print()
my_fun3(my_job)

print()
my_fun4(my_job)
```

```
Ram
Sita
Shyam

dict_keys(['Ram', 'Sita', 'Shyam'])

dict_values(['Advocate', 'Teacher', 'Doctor'])

Ram Advocate
Sita Teacher
Shyam Doctor
```

Return Values: To let a function return a value, use the return statement

In [15]:
```
def my_function(x):
  return(2*x)

print(my_function(3))

a = my_function(5)
print(a)
```

```
6
10
```

The pass Statement: function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

In [16]:
```
def my_fun():

print("Hello")
my_fun()
print("World")
```

```
  File "<ipython-input-16-66f9f1934eb0>", line 3
    print("Hello")
    ^
IndentationError: expected an indented block
```

In [17]:
```
def my_fun():
    pass

print("Hello")
my_fun()
print("World")
```

```
Hello
World
```

Recursion: Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In [18]:
```
# What if thee is no 'if' statement?

def demo_recursion(x):
    if(x>0):
        print(x)
        x-=2
        demo_recursion(x)

a = 15
demo_recursion(a)
```

```
15
13
11
9
7
5
3
1
```

Scope

A variable is only available from inside the region it is created. This is called scope.

Local Scope: A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

In [19]:
```
def myfunc():
    number = 300
    print(number)

myfunc()
print(number)
```

```
300

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-ff79aa231e91> in <module>
      4
      5 myfunc()
----> 6 print(number)

NameError: name 'number' is not defined
```

Global Scope: A variable created in the main body of the Python code is a global variable and belongs to the global scope. Global variables are available from within any scope, global and local.

In [20]:
```
number = 300

def myfunc():
    print(number)

myfunc()
print(number)
```

```
300
300
```

Naming Variables: If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function)

In [21]:
```
x = 300

def myfunc():
    x = 200
    print(x)

myfunc()
print(x)
```

```
200
300
```

Global Keyword: If you need to create a global variable, but are stuck in the local scope, you can use the global keyword. The global keyword makes the variable global.

In [22]:
```
def myfunc():
    global x
    x = 300

myfunc()
print(x)
```

```
300
```

In [23]:
```
# Also, use the global keyword if you want to make a change to a global variable inside a functi

x = 300

def myfunc():
    global x
    x = 200
    print(x)

myfunc()
print(x)
```

```
200
200
```

Inner Functions:

A function which is defined inside another function is known as inner function or nested function. Nested functions are able to access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function. This process is also known as Encapsulation.

In [24]:
```
def myfunc():
    digit = 300

    def myinsidefunc():
        print(digit)

    myinsidefunc()

myfunc()
# myinsidefunc()
# print(digit)
```

```
300
```

Scope of variable in nested function: As explained in the example above, the variable 'digit' is not available outside the function, but it is available for any function inside the function. We can change value of a variable in outer function inside the inner function.

In [25]:
```
def myfunc():
    digit = 300

    def myinsidefunc():
        digit = 500
        print(digit)

    myinsidefunc()

myfunc()
```

```
500
```

Closures: https://www.programiz.com/python-programming/closure (https://www.programiz.com/python-programming/closure)

dir() Function

dir() is a powerful inbuilt function in Python3, which returns list of the attributes and methods of any object (say functions , modules, strings, lists, dictionaries etc.)

dir() tries to return a valid list of attributes of the object it is called upon. Also, dir() function behaves rather differently with different type of objects, as it aims to produce the most relevant one, rather than the complete information.

- For Class Objects, it returns a list of names of all the valid attributes and base attributes as well.
- For Modules/Library objects, it tries to return a list of names of all the attributes, contained in that module.
- If no parameters are passed it returns a list of names in the current local scope.

Syntax: dir({object})

Object name is optional

In [1]:
```
print(dir())
```

```
['In', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', '_dh', '_i', '_i1', '_ih', '_ii', '_iii', '_oh', 'exit', 'get_ipython',
'quit']
```

In [2]:
```
import random
import math
print(dir())
```

```
['In', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', '_dh', '_i', '_i1', '_i2', '_ih', '_ii', '_iii', '_oh', 'exit', 'get_ipy
thon', 'math', 'quit', 'random']
```

In [3]:
```
# displays the contents of the random library
print(dir(random))
```

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI',
'_Sequence', '_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_accumulate', '_acos', '_bisect', '_ceil', '_cos', '_e', '_
exp', '_inst', '_log', '_os', '_pi', '_random', '_repeat', '_sha512', '_sin', '_sqrt', '_test', '_t
est_generator', '_urandom', '_warn', 'betavariate', 'choice', 'choices', 'expovariate', 'gammavaria
te', 'gauss', 'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randi
nt', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonm
isesvariate', 'weibullvariate']
```

In [4]:
```
# With dictionary: Returns all the available dict methods in the current local scope and common
# attributes of the dictionary

my_job = {'Ram': 'Advocate', 'Sita': 'Teacher', 'Shyam': 'Doctor'}
print(dir(my_job))
```

```
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__init_subcl
ass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__reversed__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclassh
ook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'updat
e', 'values']
```

In [5]:
```
# With list: Returns all the available list methods in current local scope

my_list = ['Ram', 'Advocate', 'Sita', 'Teacher']
print(dir(my_list))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq_
_', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '_
_imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '_
_ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setatt
r__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'coun
t', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

In [6]:
```
class Person:
    name = "John"
    age = 36
    country = "Norway"

print(dir(Person))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age', 'country', 'name']
```

filter() function

The filter() function returns an iterator were the items are filtered through a function to test if the item is accepted or not.

Syntax: filter(function, iterable)

In [7]:
```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)

for x in adults:
    print(x)
```

```
18
24
32
```

Modules

Consider a module to be the same as a code library. A file containing a set of functions you want to include in your application.

To create a module just save the code you want in a file with the file extension .py

- We can use the module by using the import statement.
- When using a function from a module, use the syntax: module_name.function_name.

In [8]:
```
import math
import random
print(dir())
```

```
['In', 'Out', 'Person', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_i2', '_i3', '_i4', '_i5', '_i6', '_i
7', '_i8', '_ih', '_ii', '_iii', '_oh', 'adults', 'ages', 'exit', 'get_ipython', 'math', 'myFunc',
'my_job', 'my_list', 'quit', 'random', 'x']
```

Re-naming a Module: You can create an alias when you import a module, by using the as keyword.

In [9]:
```
# import numpy
# print(numpy.__version__)

import numpy as np
print(np.__version__)
```

```
1.19.4
```

Import From Module: You can choose to import only parts from a module, by using the from keyword. When importing using the from keyword, do not use the module name when referring to elements in the module.

Example: Counter["age"], not collections.Counter["age"]

In [10]:
```
from collections import defaultdict
from collections import Counter
from collections import deque
from collections import namedtuple
from collections import OrderedDict
```

math Module: https://www.w3schools.com/python/module_math.asp (https://www.w3schools.com/python/module_math.asp)

In [11]:
```
import math
a = 5
# math.ceil(a)
# math.floor(a)
math.factorial(a)
```

Out[11]: 120

sys Module: https://www.javatpoint.com/python-sys-module (https://www.javatpoint.com/python-sys-module)

In [12]:
```
import sys
sys.platform
```

Out[12]: 'linux'

os Module

OS module in Python provides functions for interacting with the operating system. OS, comes under Python's standard utility modules. This module provides a portable way of using operating system dependent functionality.

In [13]:
```
import os
```

In [14]:
```
# To get the location of the current working directory os.getcwd() is used.
cwd = os.getcwd()
print(cwd)
```

```
/home/lenovo/Documents/Python Tute
```

os.chdir() method in Python used to change the current working directory to specified path. It takes only a single argument as new directory path.

os.mkdir() method in Python is used to create a directory named path with the specified numeric mode. This method raise FileExistsError if the directory to be created already exists.

os.listdir() method in Python is used to get the list of all files and directories in the specified directory. If we don’t specify any directory, then list of files and directories in the current working directory will be returned.

In [15]:
```
os.listdir()
```

Out[15]:
```
['.ipynb_checkpoints',
 '3. Sets.ipynb',
 'Misc.ipynb',
 '4. Tuples.ipynb',
 'Python Misc.',
 'list1.png',
 'listcompre1.png',
 'listcompre.png',
 '1. Introduction to Python.ipynb',
 '7. Functions and Modules.ipynb',
 'collections.png',
 'strings.png',
 '5. Dictionary.ipynb',
 '2. List.ipynb',
 'chart.jpeg',
 'PDF Notes',
 '6. Collections Module.ipynb',
 'Untitled.ipynb']
```

In [16]:
```
path = os.getcwd()
dir_list = os.listdir(path)
print(dir_list)
```

```
['.ipynb_checkpoints', '3. Sets.ipynb', 'Misc.ipynb', '4. Tuples.ipynb', 'Python Misc.', 'list1.pn
g', 'listcompre1.png', 'listcompre.png', '1. Introduction to Python.ipynb', '7. Functions and Modul
es.ipynb', 'collections.png', 'strings.png', '5. Dictionary.ipynb', '2. List.ipynb', 'chart.jpeg',
'PDF Notes', '6. Collections Module.ipynb', 'Untitled.ipynb']
```

os.remove() method in Python is used to remove or delete a file path. This method can not remove or delete a directory. If the specified path is a directory then OSError will be raised by the method.

In [19]:
```
# path = os.getcwd()
os.remove('test.txt')
```

In [20]:
```
print(os.listdir(path))
```

```
['.ipynb_checkpoints', '3. Sets.ipynb', 'Misc.ipynb', '4. Tuples.ipynb', 'Python Misc.', 'list1.pn
g', 'listcompre1.png', 'listcompre.png', '1. Introduction to Python.ipynb', '7. Functions and Modul
es.ipynb', 'collections.png', 'strings.png', '5. Dictionary.ipynb', '2. List.ipynb', 'chart.jpeg',
'PDF Notes', '6. Collections Module.ipynb', 'Untitled.ipynb']
```

time: https://www.programiz.com/python-programming/time (https://www.programiz.com/python-programming/time)

In [21]:
```
import time
```

In [22]:
```
seconds = time.time()
print("Seconds since epoch =", seconds)
```

```
Seconds since epoch = 1615731143.504409
```

In [23]:
```
# seconds passed since epoch
print("Local time:", time.ctime(time.time()))
```

```
Local time: Sun Mar 14 19:42:23 2021
```

In [24]:
```
for i in range(5):
    print(i)
    time.sleep(2)
```

```
0
1
2
3
4
```

In [25]:
```
result = time.localtime(1615727124.2858176)
print("result:", result)
print("\nyear:", result.tm_year)
print("tm_hour:", result.tm_hour)
```

```
result: time.struct_time(tm_year=2021, tm_mon=3, tm_mday=14, tm_hour=18, tm_min=35, tm_sec=24, tm_w
day=6, tm_yday=73, tm_isdst=0)

year: 2021
tm_hour: 18
```

In [26]:
```
t = (2021,3,14,18,35,24,6,73,0)
result = time.asctime(t)
print("Result:", result)
```

```
Result: Sun Mar 14 18:35:24 2021
```

In [27]:
```
result = time.strftime("%H:%M:%S, %m-%d-%Y", t)
print("Result:", result)
```

```
Result: 18:35:24, 03-14-2021
```

In [ ]:
```

```