Functions, Modules, Packages

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

Modules, Imports

In Python a module is simply a file with the **.py** extension containing Python code:

In [6]:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

In [7]:

```
result
```

Out[7]:

```
7
```

In [8]:

```
pi
```

Out[8]:

```
3.14159
```

Or equivalently:

In [9]:

```
from some_module import f, g, PI
result = g(5, PI)
```

By using the **as** keyword you can give imports different variable names:

In [10]:

```
import some_module as sm
from some_module import PI as pi, g as gf
r1 = sm.f(pi)
r2 = gf(6, pi)
```

It is also possible to import all names from a module into the current namespace by using the following import statement:

In [11]:

```
from some_module import *
```

The **dir()** built-in function returns a sorted list of strings containing the names defined by a module.

In [12]:

```
import math

content = dir(math)
print (content)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acos
h', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cos
h', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floo
r', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfini
te', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'mod
f', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

The **globals()** and **locals()** functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

If you want to reexecute the top-level code in a module, you can use the **reload()** function.

In [17]:

```
import importlib
importlib.reload(some_module)
```

Out[17]:

```
<module 'some_module' from 'D:\\FORTen\\PYTHON TUT TA\\ITself\\MYself\\Python T
utorial Jupyters\\Functions, Modules, Packages\\some_module.py'>
```

Functions

Functions are declared with the **def** keyword and returned from with the **return** keyword:

In [7]:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

If Python reaches the end of a function without encountering a return statement, **None** is returned automatically.

Each function can have *positional* arguments and *keyword* arguments. Keyword arguments are most commonly used to specify default values or optional arguments. In the preceding function, x and y are positional arguments while z is a keyword argument. This means that the function can be called in any of these ways:

```
In [8]: my_function(5, 6, z=0.7)
```

```
Out[8]: 0.06363636363636363
```

```
In [9]: my_function(3.14, 7, 3.5)
```

```
Out[9]: 35.49
```

```
In [10]: my_function(10, 20)
```

```
Out[10]: 45.0
```

- **keyword arguments must follow the positional arguments (if any)**.
- **You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are**.

It is possible to use keywords for passing positional arguments as well. In the preceding example, we could also have written:

```
In [21]: my_function(x=5, y=6, z=7)
         my_function(y=6, x=5, z=7)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-21-254f66f04f61> in <module>()
----> 1 my_function(x=5, y=6, z=7)
      2 my_function(y=6, x=5, z=7)

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

Variable-length arguments

```
In [25]: def printinfo( arg1, *vartuple ):
             "This prints a variable passed arguments"
             print ("Output is: ")
             print (arg1)
             print ("Other: ")
             for var in vartuple:
                 print (var)
```

In [26]:

```
printinfo( 70, 60, 50 )
```

```
Output is:
70
Other:
60
50
```

In [27]:

```
printinfo( 70, 60, 50 , 100 ,20)
```

```
Output is:
70
Other:
60
50
100
20
```

Arbitrary Number of Keyword Parameters

In [32]:

```
def f(**kwargs):
    for key, value in kwargs.items():
        print("The value of {} is {}".format(key, value))
```

In [33]:

```
f()
```

In [34]:

```
f(de="German",en="English",fr="French")
```

```
The value of de is German
The value of en is English
The value of fr is French
```

Docstring

The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does. This string is available to us as __doc__ attribute of the function.

In [19]:

```
def greet(name):
    """This function greets to the person passed in as
    parameter"""
    print("Hello, " + name + ". Good morning!")
```

In [20]:
```
print(greet.__doc__)
```

```
This function greets to the person passed in as
    parameter
```

Namespaces, Scope, and Local Functions

Functions can access variables in two different scopes: *global* and *local*. An alternative and more descriptive name describing a variable scope in Python is a *namespace*.

Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function’s arguments. After the function is finished, the local namespace is destroyed

In [12]:
```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

In [13]:
```
func()
```

In [14]:
```
print(a)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-bca0e2660b9f> in <module>()
----> 1 print(a)

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

In [17]:
```
a = []
def func():
    for i in range(5):
        a.append(i)
```

In [18]:
```
func()
```

In [19]:
```
print(a)
```

```
[0, 1, 2, 3, 4]
```

Assigning variables outside of the function’s scope is possible, but those variables must be declared as global via the **global** keyword:

```
In [23]: x = None
         def bind_a_variable():
             global x
             x = []
```

```
In [24]: bind_a_variable()
```

```
In [25]: print(x)

         []
```

Returning Multiple Values

```
In [26]: def f():
             a = 5
             b = 6
             c = 7
             return a, b, c
```

```
In [27]: a, b, c = f()
```

```
In [28]: a,b,c

Out[28]: (5, 6, 7)
```

```
In [29]: return_value = f()
```

```
In [30]: return_value

Out[30]: (5, 6, 7)
```

```
In [31]: def f():
             a = 5
             b = 6
             c = 7
             return {'a' : a, 'b' : b, 'c' : c}
```

Functions Are Objects

Suppose we were doing some data cleaning (stripping whitespace, removing punctuation symbols, and standardizing on proper capitalization) and needed to apply a bunch of transformations to the following list of strings:

```
In [34]: states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda', 'south carol
```

One way to do this is to use built-in string methods along with the re standard library module for regular expressions:

In [36]:

```
import re
def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('[!#?]', '', value)
        value = value.title()
        result.append(value)
    return result
```

In [37]:

```
clean_strings(states)
```

Out[37]:

```
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

An alternative approach that you may find useful is to make a list of the operations you want to apply to a particular set of strings:

In [40]:

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

clean_ops = [str.strip, remove_punctuation, str.title]
def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

In [41]:

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

Out[41]:

```
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

You can use functions as arguments to other functions like the built-in **map** function, which applies a function to a sequence of some kind:

In [43]:
```
for x in map(remove_punctuation, states):
    print(x)
```

```
 Alabama
Georgia
Georgia
georgia
FlOrIda
south carolina
West virginia
```

Anonymous (Lambda) Functions

Writing functions consisting of a single statement, the result of which is the return value:

In [54]:
```
(lambda x: x * 2)(4)
```

Out[54]:
```
8
```

In [50]:
```
equiv_anon = lambda x: x * 2
```

In [52]:
```
equiv_anon(2)
```

Out[52]:
```
4
```

In [46]:
```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

Out[46]:
```
[8, 0, 2, 10, 12]
```

As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string

In [47]:
```
strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

In [48]:
```
strings.sort(key=lambda x: len(set(list(x))))
```

In [49]:
```
strings
```

Out[49]:
```
['aaaa', 'foo', 'abab', 'bar', 'card']
```

Unlike functions declared with the def keyword, the function object itself is never given an explicit **name** attribute.

Currying: Partial Argument Application

Deriving new functions from existing ones by partial argument application.

In [57]:
```
def add_numbers(x, y):
    return x + y
```

Using this function, we could derive a new function of one variable, add_five, that adds 5 to its argument:

In [59]:
```
add_five = lambda y: add_numbers(5, y)
```

The built-in functools module can simplify this process using the partial function:

In [60]:
```
from functools import partial
add_five = partial(add_numbers, 5)
```

In [61]:
```
add_five(3)
```

Out[61]:
```
8
```

Iterators and Generators

iterator protocol: a generic way to make objects iterable. For example, iterating over a dict yields the dict keys:

In [67]:
```
some_dict = {'a': 1, 'b': 2, 'c': 3}
```

In [68]:
```
for key in some_dict:
    print(key)
```

```
a
b
c
```

When you write for key in some_dict, the Python interpreter first attempts to create an iterator out of some_dict:

In [69]:
```
dict_iterator = iter(some_dict)
```

In [71]:
```
dict_iterator
```

Out[71]:
```
<dict_keyiterator at 0xb0ffc78>
```

An iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as min, max, and sum, and type constructors like list and tuple:

In [72]:
```
list(dict_iterator)
```

Out[72]:
```
['a', 'b', 'c']
```

A *generator* is a concise way to construct a new iterable object. Whereas normal functions execute and return a single result at a time, *generators return a sequence of multiple results lazily*, pausing after each one until the next one is requested. To create a generator, use the **yield** keyword instead of return in a function:

In [63]:
```
def squares(n=10):
    print('Generating squares from 1 to {0}'.format(n ** 2))
    for i in range(1, n + 1):
        yield i ** 2
```

When you actually call the generator, no code is immediately executed:

In [65]:
```
gen = squares()
```

It is not until you request elements from the generator that it begins executing its code:

In [74]:
```
for x in gen:
    print(x, end=' ')
```

```
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

Generator expresssions

A concise way to make a generator. This is a generator analogue to list, dict, and set comprehensions; to create one, enclose what would otherwise be a list comprehension within parentheses instead of brackets:

In [78]:
```
gen = (x ** 2 for x in range(100))
```

In [79]:
```
gen
```

Out[79]:
```
<generator object <genexpr> at 0x000000000B10CB48>
```

This is completely equivalent to the following more verbose generator:

```
In [80]: def _make_gen():
             for x in range(100):
                 yield x ** 2
         gen = _make_gen()
```

Generator expressions can be used instead of list comprehensions as function arguments in many cases:

```
In [81]: sum(x ** 2 for x in range(100))

Out[81]: 328350
```

```
In [82]: dict((i, i **2) for i in range(5))

Out[82]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

itertools module

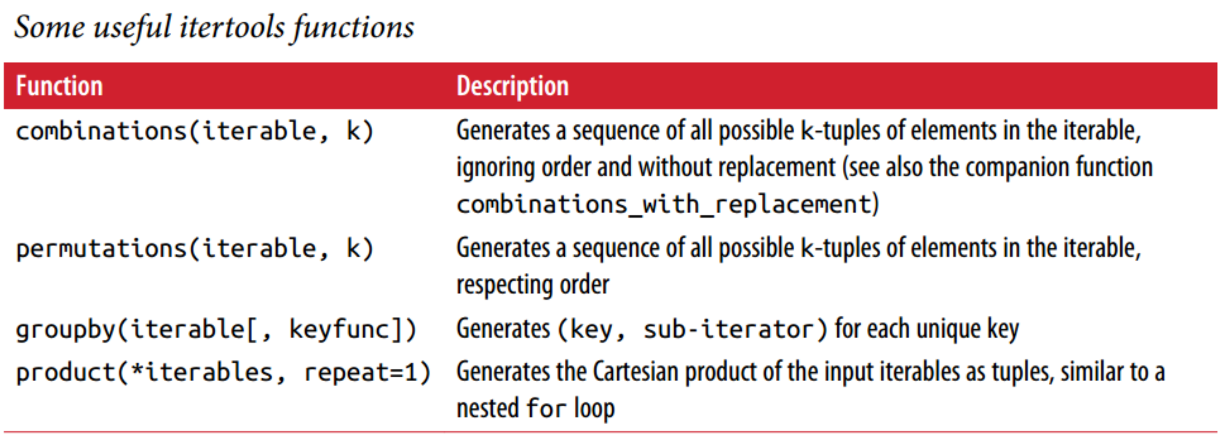
The standard library itertools module has a collection of generators for many common data algorithms. For example, **groupby** takes any sequence and a function, grouping consecutive elements in the sequence by return value of the function. Here's an example:

```
In [87]: import itertools
```

```
In [88]: first_letter = lambda x: x[0]
```

```
In [100]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']
```

```
In [102]: for letter, names in itertools.groupby(names, first_letter):
              print(letter, list(names))
```



Some useful itertools functions

Function	Description
combinations(iterable, k)	Generates a sequence of all possible k-tuples of elements in the iterable, ignoring order and without replacement (see also the companion function combinations_with_replacement)
permutations(iterable, k)	Generates a sequence of all possible k-tuples of elements in the iterable, respecting order
groupby(iterable[, keyfunc])	Generates (key, sub-iterator) for each unique key
product(*iterables, repeat=1)	Generates the Cartesian product of the input iterables as tuples, similar to a nested for loop

Function	Description

abs() (ref_func_abs.asp)	Returns the absolute value of a number
all() (ref_func_all.asp)	Returns True if all items in an iterable object are true
any() (ref_func_any.asp)	Returns True if any item in an iterable object is true
ascii() (ref_func_ascii.asp)	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin() (ref_func_bin.asp)	Returns the binary version of a number
bool() (ref_func_bool.asp)	Returns the boolean value of the specified object
bytearray() (ref_func_bytearray.asp)	Returns an array of bytes
bytes() (ref_func_bytes.asp)	Returns a bytes object
callable() (ref_func_callable.asp)	Returns True if the specified object is callable, otherwise False
chr() (ref_func_chr.asp)	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile() (ref_func_compile.asp)	Returns the specified source as an object, ready to be executed
complex() (ref_func_complex.asp)	Returns a complex number
delattr() (ref_func_delattr.asp)	Deletes the specified attribute (property or method) from the specified object
dict() (ref_func_dict.asp)	Returns a dictionary (Array)
dir() (ref_func_dir.asp)	Returns a list of the specified object's properties and methods
divmod() (ref_func_divmod.asp)	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate() (ref_func_enumerate.asp)	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval() (ref_func_eval.asp)	Evaluates and executes an expression
exec() (ref_func_exec.asp)	Executes the specified code (or object)
filter() (ref_func_filter.asp)	Use a filter function to exclude items in an iterable object
float() (ref_func_float.asp)	Returns a floating point number
format() (ref_func_format.asp)	Formats a specified value
frozenset() (ref_func_frozenset.asp)	Returns a frozenset object
getattr() (ref_func_getattr.asp)	Returns the value of the specified attribute (property or method)
globals() (ref_func_globals.asp)	Returns the current global symbol table as a dictionary
hasattr() (ref_func_hasattr.asp)	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system
hex() (ref_func_hex.asp)	Converts a number into a hexadecimal value
id() (ref_func_id.asp)	Returns the id of an object
input() (ref_func_input.asp)	Allowing user input
int() (ref_func_int.asp)	Returns an integer number

isinstance() (ref_func_isinstance.asp)	Returns True if a specified object is an instance of a specified object
issubclass() (ref_func_issubclass.asp)	Returns True if a specified class is a subclass of a specified object
iter() (ref_func_iter.asp)	Returns an iterator object
len() (ref_func_len.asp)	Returns the length of an object
list() (ref_func_list.asp)	Returns a list
locals() (ref_func_locals.asp)	Returns an updated dictionary of the current local symbol table
map() (ref_func_map.asp)	Returns the specified iterator with the specified function applied to each item
max() (ref_func_max.asp)	Returns the largest item in an iterable
memoryview() (ref_func_memoryview.asp)	Returns a memory view object
min() (ref_func_min.asp)	Returns the smallest item in an iterable
next() (ref_func_next.asp)	Returns the next item in an iterable
object() (ref_func_object.asp)	Returns a new object
oct() (ref_func_oct.asp)	Converts a number into an octal
open() (ref_func_open.asp)	Opens a file and returns a file object
ord() (ref_func_ord.asp)	Convert an integer representing the Unicode of the specified character
pow() (ref_func_pow.asp)	Returns the value of x to the power of y
print() (ref_func_print.asp)	Prints to the standard output device
property()	Gets, sets, deletes a property
range() (ref_func_range.asp)	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed() (ref_func_reversed.asp)	Returns a reversed iterator
round() (ref_func_round.asp)	Rounds a numbers
set() (ref_func_set.asp)	Returns a new set object
setattr() (ref_func_setattr.asp)	Sets an attribute (property/method) of an object
slice() (ref_func_slice.asp)	Returns a slice object
sorted() (ref_func_sorted.asp)	Returns a sorted list
@staticmethod()	Converts a method into a static method
str() (ref_func_str.asp)	Returns a string object
sum() (ref_func_sum.asp)	Sums the items of an iterator
tuple() (ref_func_tuple.asp)	Returns a tuple
type() (ref_func_type.asp)	Returns the type of an object
vars() (ref_func_vars.asp)	Returns the __dict__ property of an object
zip() (ref_func_zip.asp)	Returns an iterator, from two or more iterators

```
In [ ]:
```