Lecture 4 : Python Basics (Part III)

Data Science, DST, UIC

In this lecture, we will cover the following topics:

1. Functions
2. Classes and Objects
3. Exception handling
4. Packages

1 Function

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.

- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes code reusable.

1.1 Built-in Functions

Python interpreter has a number of functions and types built into it that are always available. You can always call the built-in functions directly using their names.

You can find a list of built-in functions here (https://docs.python.org/3/library/functions.html), or use `dir( __builtins__ )`

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

```
Out[1]: ['ArithmeticError',
         'AssertionError',
         'AttributeError',
         'BaseException',
         'BlockingIOError',
         'BrokenPipeError',
         'BufferError',
         'BytesWarning',
         'ChildProcessError',
         'ConnectionAbortedError',
         'ConnectionError',
         'ConnectionRefusedError',
         'ConnectionResetError',
         'DeprecationWarning',
         'EOFError',
         'Ellipsis',
         'EnvironmentError',
         'Exception',
         'False',
         'FileExistsError',
         'FileNotFoundError',
         'FloatingPointError',
         'FutureWarning',
         'GeneratorExit',
         'IOError',
         'ImportError',
         'ImportWarning',
         'IndentationError',
         'IndexError',
         'InterruptedError',
         'IsADirectoryError',
         'KeyError',
         'KeyboardInterrupt',
         'LookupError',
         'MemoryError',
         'ModuleNotFoundError',
         'NameError',
         'None',
         'NotADirectoryError',
         'NotImplemented',
         'NotImplementedError',
         'OSError',
         'OverflowError',
         'PendingDeprecationWarning',
         'PermissionError',
         'ProcessLookupError',
         'RecursionError',
         'ReferenceError',
         'ResourceWarning',
         'RuntimeError',
         'RuntimeWarning',
         'StopAsyncIteration',
         'StopIteration',
         'SyntaxError',
         'SyntaxWarning',
         'SystemError',
         'SystemExit',
         'TabError',
         'TimeoutError',
         'True',
         'TypeError',
         'UnboundLocalError',
         'UnicodeDecodeError',
         'UnicodeEncodeError',
         'UnicodeError',
         'UnicodeTranslateError',
         'UnicodeWarning',
         'UserWarning',
```

'ValueError',
'Warning',
'WindowsError',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',

```
 'print',
 'property',
 'range',
 'repr',
 'reversed',
 'round',
 'set',
 'setattr',
 'slice',
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

Note that the implementation of python built-in functions may not necessarily be using python language. For example, for the sake of performance, the source code of some of the built-in functions are in C.

In [2]:
```
i=20
type(i)
```

Out[2]:
```
int
```

In [3]:
```
abs(-2.0)
```

Out[3]:
```
2.0
```

In [4]:
```
# return smallest item of the given iterable argument, has to be comparable
min([2,3,4])
```

Out[4]:
```
2
```

In [5]:
```
# pow(x,y) Equivalent to x**y
pow(2,8)
```

Out[5]:
```
256
```

In [6]:
```
# round (number, ndigits) Round a number to a given precision in decimal digits.
round(3.14159, 3)
```

Out[6]:
```
3.142
```

In [7]:
```
set("abac")
```

Out[7]:
```
{'a', 'b', 'c'}
```

In [8]:
```
isinstance([1,2], list)
```

Out[8]:
```
True
```

In [9]:
```
# use the built in function help() to check the usage infomation of built in function iter()
help(iter)

#or use ? after the function to get the information
iter?
```

```
Help on built-in function iter in module builtins:

iter(...)
    iter(iterable) -> iterator
    iter(callable, sentinel) -> iterator

    Get an iterator from an object.  In the first form, the argument must
    supply its own iterator, or be a sequence.
    In the second form, the callable is called until it returns the sentinel.
```

In [10]:
```
# range return an object that produces a sequence of integers
range(1,10,2)
```

Out[10]:
```
range(1, 10, 2)
```

In [11]:
```
list(range(1,10,2))
```

Out[11]:
```
[1, 3, 5, 7, 9]
```

In [12]:
```
# bin return the binary representation of an integer
bin(255)
```

Out[12]:
```
'0b11111111'
```

In [13]:
```
hex(256)
```

Out[13]:
```
'0x100'
```

1.2 Module functions

Different from built in functions, module functions are defined in third party packages or modules. You must **import** the module or package before you call the functions.

- A module is a file containing Python definitions, functions and statements.
- The file name is the module name with the suffix `.py` appended.

Before you **import** a package or module in your python codes, you need to install them first, the commonly used pacakge management tools include **Pip** or **conda**. In the jupyter notebook, most commonly used packages are already installed so that you can import them directly when necessary.

There are several variants of import statement that imports names from a module:

- `import module1` : import the module1, you can access all the functions defined in module1 by using the module name.
- `import module1 as rename1` : import module1 and rename it to rename1.
- `from module1 import function1` : import the function1 defined in module1.

In [14]:
```
import math
math.sin(math.pi/2)
```

Out[14]:
```
1.0
```

In [15]:
```
# Rename the imported module using as
import math as mt
mt.cos(1.5)
```

Out[15]: 0.0707372016677029

In [16]:
```
#cos(1.5)  # cos is not defined
```

In [17]:
```
from math import cos
cos(1.5)
```

Out[17]: 0.0707372016677029

1.3 Self-defined functions

We can also create our own functions, these functions are called self-defined functions.

Syntax of defining function:

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

Above shown is a function definition which consists of following components.

- Keyword `def` marks the start of function header.
- A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
- An optional `return` statement to return a value from the function.

In [18]:
```
def my_func():
    j=0
    print('hello world '+str(j))
    return
```

Once we have defined a function, we can call it from another function in the program. To call a function we simply type the function name with appropriate parameters.

In [19]:
```
my_func()
```

```
hello world 0
```

use docString to show the help of function

In [20]:
```
def get_name():
    '''get the user name according to the user input msg,
    if input is empty, the default name is Anonymous User'''
    name = input() or 'Anonymous User'
    return name
```

In [21]:
```
get_name?
```

In [22]:
```
help(get_name)
```

```
Help on function get_name in module __main__:

get_name()
    get the user name according to the user input msg,
    if input is empty, the default name is Anonymous User
```

In [23]:
```
get_name()
```

```
Test
```

Out[23]:
```
'Test'
```

The return value of function use `return` statement.

In [24]:
```
def my_func2(i):
    j=i+1
    return j

print(my_func2(5))
```

```
6
```

If a function does not explicitly return a value, then the default return value is 'None'

In [25]:
```
def my_func3(i):
    j=i+1

print(my_func3(5))
```

```
None
```

A function can return multiple values

In [26]:
```
def my_func4(i):
    j=i+1
    return i,j

c = my_func4(5)
a,b = my_func4(5)
print(a)
print(b)
print(my_func4(5))
```

```
5
6
(5, 6)
```

The arguments of a function

From **function definition** perspective, the arguments can be divided into **mandatory(non-default) arguments** and **optional arguments**.

- Arguments without default value are mandatory arguments, which means one must provide a value for the argument when calling the function.
- Arguments with default value are optional arguments, which means if not provided a value when calling the function, default value will be used.
- The optional arguments **must be behind** the mandatory arguments in the definition of functions

In [27]:
```
#there is error, because the optional arguments must behind the mandatory arguments in the defin
ition of functions

# def my_func5(x1,x2,x4=4,x3,x5=5): # x1,x2,x3 are mandatory arguments; x4 and x5 are optional a
rguments
#     print(x1)
#     print(x2)
#     print(x3)
#     print(x4)
#     print(x5)
```

In [28]:
```
#correct definition of function
def my_func5(x1, x2, x3, x4=4, x5=5):
    print(x1, x2, x3, x4, x5)
```

In [29]:
```
#correct call of function
my_func5(1,2,3,7,8)
```

```
1 2 3 7 8
```

In [30]:
```
#correct call of function, some arguments are omitted when calling, therefore will use default v
alues
my_func5(1,2,3)
```

```
1 2 3 4 5
```

In [31]:
```
# incorrect call of function, you must provide all the mandatory arguments
# my_func5(1,2)
```

From **function calling** perspective, the arguments can be divided into **positional arguments** and **keyword-only arguments**.

- The positional arguments pass the value to the arguments depending on the position in the argument list of the function definition.
- The keyword-only arguments need the name of arguments to appear explicitly when calling the function.
- The keyword-only arguments can only appear **after** the positional arguments

In [32]:
```
#call the function and pass the value using argument names
# my_func5(1,2,3,4,5) positional arguments
my_func5(x1=1,x3=2,x2=3,x5=6)
```

```
1 3 2 4 6
```

In [33]:
```
#some value are passed by position, some by names
my_func5(1,2,3,x5=5)
```

```
1 2 3 4 5
```

In [34]:
```
# Error! Because positional argument should be always before keyword argument
# my_func5(1,x2=2,3,x5=5)
```

From **function definition** perspective

- the arguments that start with * will take in tuple as their value. If you do not know how many arguments that will be passed into your function, use *.
- the arguments that start with ** will take in dictionary as their value
- the arguments that appear after the arguments that start with * are **all keyword-only arguments**, which means you must provide their names when trying to pass a value to them.

In [35]:
```
def my_func6(x1,*x2,x3=3,x4,x5=5):
    print(x1,x2,x3,x4,x5)
```

In [36]:
```
#value 2 passed to variable x2 becomes a tuple (2,)
my_func6(1,2,x3=3,x4=4)
```

```
1 (2,) 3 4 5
```

In [37]:
```
#value 2,3,3 passed to variable x2 becomes (2,3,3)
my_func6(1,2,3,3,x3=4,x4=4)
```

```
1 (2, 3, 3) 4 4 5
```

In [38]:
```
#values 2,3 are passed to variable x2, not to x3;
#variable x3 uses default value 3;
my_func6(1,2,3,x4=4)
```

```
1 (2, 3) 3 4 5
```

In [39]:
```
def my_func7(x1,x2,*x3,x4,**x5): # argument start with ** must be last argument
    print(x1,x2,x3,x4,x5)

my_func7(1,2,3,3,x4=4,a=1,b=2)
```

```
1 2 (3, 3) 4 {'a': 1, 'b': 2}
```

You can pass the tuple and dictionary variables as the arguments in function call by put * and ** before the variable.

In [40]:
```
dic_args = {"a":1,"b":2}
tu_args = (3,3)
my_func7(1,2,*tu_args,x4=4,**dic_args)
```

```
1 2 (3, 3) 4 {'a': 1, 'b': 2}
```

1.4 The visibility of variables:

1. local variable: defined and visible inside a function
2. global variable: defined anywhere and visible to the whole program. Use it carefully.
3. non-local variable: similar to global variable but used only in embedded functions, will not be discussed here.

In [41]:
```
x=0                # the first x is visible in the scope where it is defined

def my_func7(i):
    x=i            # the second x is visible in the scope where it is defined, which is inside t
he function my_func6(i)
    print(x)       # note that the two 'x' are different variables even they share the same name

my_func7(1)
print(x)
```

```
1
0
```

In [42]:
```
x=0                # the first x is visible in the scope where it is defined
def my_func8(i):
    global x       # a global variable x is defined, which will overwrite the x variable define
d before
    x=i
    print(x)

my_func8(1)
print(x)
```

```
1
1
```

1.5 Pass by value or pass by address

When passing a value to an argument, there are mainly two ways of doing it. One is passing by value, the other is passing by address.

- If we use an **immutable variable** (such as int, float, str, bool, tuple) to pass its value to the argument of a function, it is **passing by value**. When passing by value, the function will not affect the variable that passed its value to the function argument. For example:

In [43]:
```
i = 100
def my_func9(j,k=2): # j = i
    j+=k

my_func9(i)  # we use the variable i, which is an integer (immutable variable), to pass the valu
e to the argument j
print(i)     # the value of variable i remains unchanged
```

```
100
```

- If we use a **mutable variable** (such as list, set, dict) to pass its value to the argument of a function, it is passing by address. When passing by address, the function will affect the variable that passed its value to the function argument. For example:

In [44]:
```
i = [100]

def my_func9(j,k=2):
    j[0]+=k

my_func9(i)  # we use the variable i, which is a list (mutable variable), to pass the value to t
he argument j
print(i)     # the value of variable i will be changed
```

```
[102]
```

The reason that passing by address will change the variable i in the above example is that we are actually passing the address of variable i to the function, so that the function can really change the content that is stored in that address.

2 Classes and Objects

2.1 Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Class Definition Syntax:

```
class ClassName:
    statement(s)
```

- Classes are created by keyword `class`.
- Attributes are the variables that belong to class.
- Attributes are always public and can be accessed using dot (.) operator.

In [45]:
```
class MyClass:
    """A simple example class"""
    def __init__(self):
        pass
    i = 12345
    def my_func(self):
        print(self.i)
        return 'hello world'
```

Class Objects

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values.

- **Class instantiation** uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class, for example:
- **Attribute references** use the standard syntax used for all attribute references in Python: `obj.name`.

In [46]:
```
x = MyClass()  # creates a new instance of the class and assigns this object to the local variable x.
print(x.i)
print(x.my_func()) # MyClass.my_func(x)
```

```
12345
12345
hello world
```

In the above example, `MyClass.i` and `MyClass.my_func` are valid attribute references, returning an integer and a function object, respectively.

self

Note that `self` represents the `instance of the class` (just like `this` in Java). By using the `self` keyword we can access the attributes and methods of the class in python.

- Class methods must have an extra first parameter in method definition, which is the instance the moethod is called on. We do not give a value for this parameter when we call the method, Python provides it.
- `self` is parameter in function and user can use another parameter name in place of it. But it is advisable to use self because it increase the readability of code.

In the above example, when we call a method of this object as `x.my_func()`, this is automatically converted by Python into `MyClass.my_func(x)`.

The instantiation operation creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()` (similar to the constructor in Java)

- It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

In [47]:
```
class MyClass:
#  if no user defined __init__, python will provide __init__(self)
    def __init__(self,number):
        self.i = number

    def get_i(self):
        return self.i

    def set_i(self,num):
        self.i = num

x = MyClass(3) # MyClass.__init__(x,3)
print(x.get_i()) # MyClass.get_i(x)
x.set_i(4) # MyClass.set_i(x,4)
print(x.get_i())
```

```
3
4
```

3 Exception Handling

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: syntax errors and exceptions.

- Syntax errors: also known as parsing errors. The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected.

In [48]:
```
# for i in range(4):
#     print(i++)
```

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal. It is possible to write programs that handle selected exceptions.

Like other languages, python also provides the runtime errors via exception handling method with the help of **try-except**.

Syntax:

```
try:
    statement(s)
except:
    exception handling statement(s)
```

In [49]:
```
a = [1, 2, 3]
try:
    print("Second element = ", a[1])
    # Throws error since there are only 3 elements in array
    print("Fourth element = ", a[3])

except:
    print("Exception occured")
```

```
Second element =  2
Exception occured
```

In the above example, we did not mention any exception in the except clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can **specify** which exceptions an except clause will catch.

- We can also use a tuple of values to specify multiple exceptions in an except clause.

In [50]:
```
try :
    a = 6
    if a < 4:
        res = a/(a-3) # throws ZeroDivisionError for a = 3
    print("Value of res = ",res)  # throws NameError if a >= 4

except(ZeroDivisionError, NameError):
    print("\nError Occurred and Handled")
```

```
Error Occurred and Handled
```

The **try-except** statement has an optional **else** clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances, **finally**. The finally clause runs whether or not the try statement produces an exception.

In [51]:
```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!") # execute if exceptio occurs
    else:
        print("result is", result) # execute if no exception occur
    finally:
        print("executing finally clause")   # execute regardless of whether there is exception o
r not
```

In [52]:
```
divide(3,4)
```

```
result is 0.75
executing finally clause
```

In [53]:
```
divide(2,0)
```

```
division by zero!
executing finally clause
```

14. Packages

In Python, a .py file can be a module, it may contain codes for classes, functions and other statements. Multiple modules together with a `__init__.py` file (optional, used for execution of package initialization code) make a package.

- Packages provide a mechanism for grouping and organizing modules.
- Packages allow for a hierarchical structuring of the module namespace using **dot notation**.

Consider the following example, which is a directory named `pkg` that contains two modules `mod1.py` and `mod2.py`

```
pkg
    mod1.py
    mod2.py
```

The contents of the modules are:

In [54]:
```
# mod1.py
def foo():
    print("mod1.foo()")

# mode2.py
def bar():
    print("mode2.bar()")
```

Given this structure, put the `pkg` directory resides in a location where is can be found (in one of the directories contained in `sys.path`), you can refer to the two modules with dot notation (`pkg.modd1`, `pkg.mod2`) and import them.

- `import package.module`
- `from package import module`

In [55]:
```
# first check the paths
# import sys
# sys.path

# put the pkg directory to one of the path
import pkg.mod1
pkg.mod1.foo()

from pkg import mod2
mod2.bar()
```

```
mod1.foo()
mode2.bar()
```

To check all the built in modules of your python version, use the following codes

In [56]:
```
import sys
sys.builtin_module_names
```

Out[56]:
```
('_abc',
 '_ast',
 '_bisect',
 '_blake2',
 '_codecs',
 '_codecs_cn',
 '_codecs_hk',
 '_codecs_iso2022',
 '_codecs_jp',
 '_codecs_kr',
 '_codecs_tw',
 '_collections',
 '_csv',
 '_datetime',
 '_functools',
 '_heapq',
 '_imp',
 '_io',
 '_json',
 '_locale',
 '_lsprof',
 '_md5',
 '_multibytecodec',
 '_opcode',
 '_operator',
 '_pickle',
 '_random',
 '_sha1',
 '_sha256',
 '_sha3',
 '_sha512',
 '_signal',
 '_sre',
 '_stat',
 '_string',
 '_struct',
 '_symtable',
 '_thread',
 '_tracemalloc',
 '_warnings',
 '_weakref',
 '_winapi',
 'array',
 'atexit',
 'audioop',
 'binascii',
 'builtins',
 'cmath',
 'errno',
 'faulthandler',
 'gc',
 'itertools',
 'marshal',
 'math',
 'mmap',
 'msvcrt',
 'nt',
 'parser',
 'sys',
 'time',
 'winreg',
 'xxsubtype',
 'zipimport',
 'zlib')
```

Installation

For non built-in packages, one can install them in one's own python development environment. Depending on the python developing environment you are using, you can either use:

- # `pip install package_name` or
- # `conda install package_name`

You can also list the currently installed packages using:

- # `pip list` or
- # `conda list`

Frequently used packages in python for data scientist are the following:

- Numpy: Processing with high dimension arrays and matrix
- Pandas: Processing with data frames
- matplotlib: Data visualization
- StatsModels: Statistical models
- Scikit-learn, TensorFlow: Machine learning
- Scrapy: Web crawling
- pySpark: Spark programming
- NLTK, spaCy: Natural Language Processing
- Random: Generating random numbers
- etc.