Harokopio University
Department of Informatics and Telematics

Introduction to Python

Object-oriented programming II

Dr. John Violos

urls:

- https://www.learnpython.org
- https://www.w3schools.com/python/python_variables.asp
- https://www.educba.com/java-vs-python/
- http://net-informations.com/python/iq/objects.htm
- https://www.programiz.com/python-programming/methods/built-in/classmethod
- https://www.geeksforgeeks.org/polymorphism-in-python/
- https://www.geeksforgeeks.org/g-fact-34-class-or-static-variables-in-python/
- https://www.tutorialspoint.com/python/python_multithreading.htm
- https://www.pythonforbeginners.com/basics/list-comprehensions-in-python
- https://docs.python.org/3/library/threading.html

Python code is much more **compact**

Java	Python

Longer lines of code as compared to Python
public class EduCba
{
public static void main (String []
args)
{
System.out.println(“Hello EduCBA”);
}
}
	print (“Hello EduCBA”)

Indentations=Nesting

Java	Python
At the end of the statement if you miss **semicolon** it throws an error. In Java you must define particular block using curly braces without it code won't work.	In python, statement **do not need a semicolon to end**. **Each line represents a new statement** In python, you have never seen a sight of curly braces (wrapping blocks) but indentation is mandatory in python. Indentation also improves readability of code.

Dynamic typed

Java	Python

In java you must declare type of the data.
class Example
{
public static void main (String [] args)
{
int x=10;
System.out.println(x);
}
}
	Python codes are dynamic typed. This means that you don't need to declare a type of the variable this is known as duck typing.
	X = 45
	site = "educba.com"

Speed

Java	Python
In terms of speed, Java is faster. Whenever in projects speed matters the java is best.	It is **slower** because python is an **interpreter** and also it **determines the type of data at runtime**.

Portability

Java	Python
Due to the high popularity of Java, JVM (Java Virtual Machine) is available almost everywhere.	Python is also portable but in front of java, python is not popular.

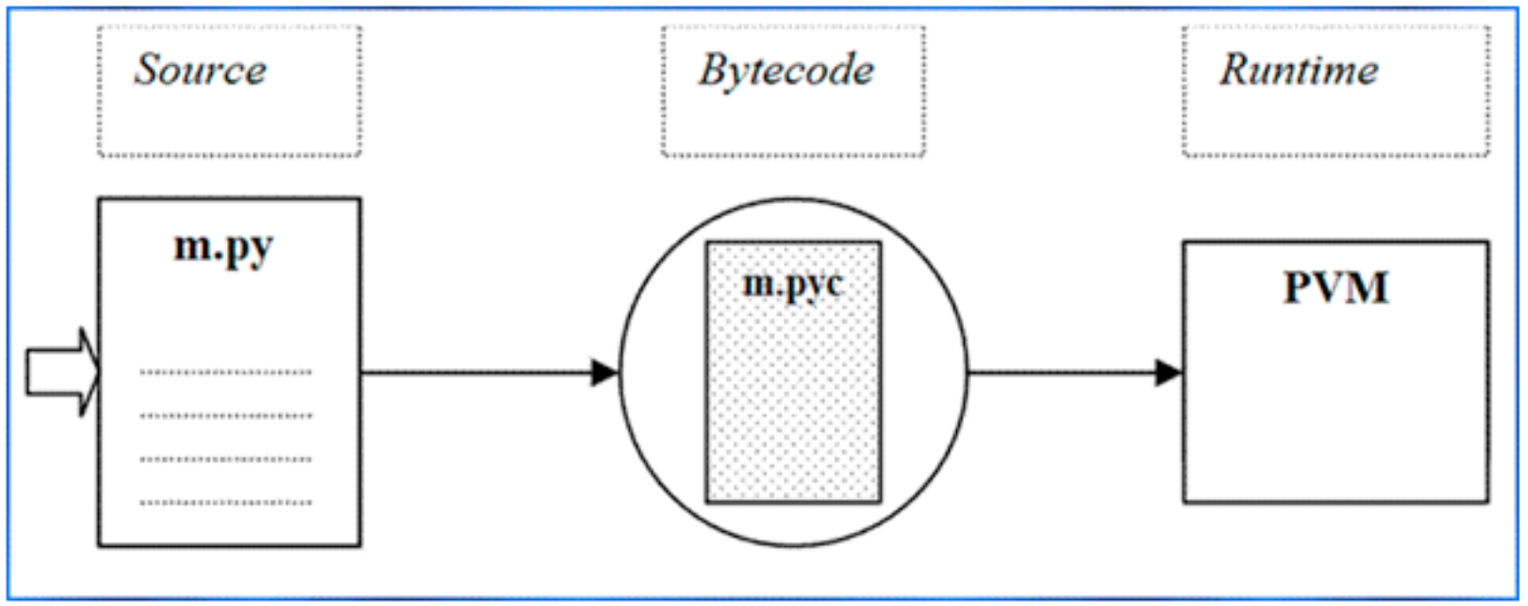
Databases

Java	Python
(JDBC)Java Database Connectivity is most popular and widely used to connect with database.	Python's database access layers are weaker than Java's JDBC. This is why it rarely used in enterprises.

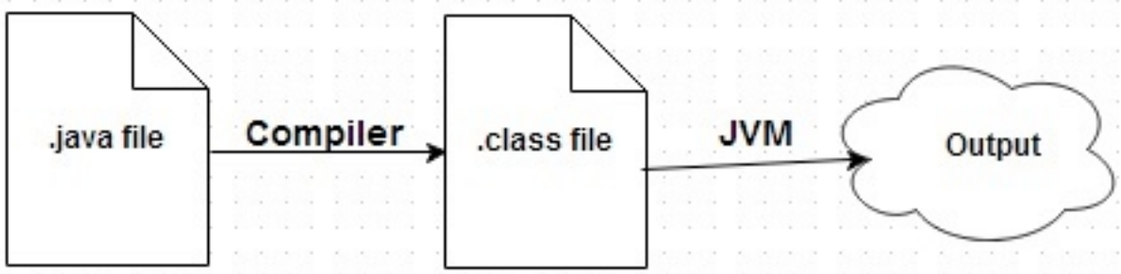
Easy to use

Java	Python
Java is not easy to use as compared to python because there is no dynamic programming concept and codes are longer than python.	Python codes are **shorter than java**. python follows dynamic programming python codes not only easy to use but also **easy to understand** because of indentation. Python has high **code readability**

Python interpreter



Manually compile:
py_compile
compileall




```
>>> import py_compile
>>> py_compile.compile('abc.py')
```

Automatically compile all files in a directory
```
>>> python -m compileall
```

Variables

Text Type: str x=“John” x = ‘John’
Numeric Types: int, float, complex x=15 x=15.0 x= 7+2j
Sequence Types: list, tuple, range l=[1,2,3,4] t=(“male”,”female”) r=range(10)
Mapping Type: dict d={"k1":"v1","k2":"v2"}
Set Types: set, frozenset s={2,4,7}
Boolean Type: bool b=True
x = float(20.5)   #Direct specify the data type
w = **float**(“4.2")   **#casting**
You can get the data type of any object by using the **type()** function:

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
To create a global variable inside a function, you can use the global keyword.

```
def myfunc():
  global x
  x = "fantastic"

myfunc()
print("Python is " + x)
```

Operators

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y
in	Returns True if a sequence with the specified value is present in the object (Collections)	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Arithmetic/Assignment/Comparison Operators same as Java

If else

```
a = 33; b = 34
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b”)

print("A") if a > b else print(“B") #Short Hand If ... Else
print("A") if a > b else print("=") if a == b else print(“B")
```

Nested If same as Java.

Combine conditional statements same as Java.
if a > b and c > a:
if a > b or a > c:

while

```
i = 2
while i < 6:
  print(i)
  if i == 3:
    break # continue
  i += 1
else: #run a block once the condition is no longer true
  print("we get 6”) #it will not run in case of break
```

Python doesn't have do-while loop.

for

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

```
for x in range(2, 30, 3):
  print(x)
else: #a block to be executed when the loop is finished
  print(“end”)

letters = ['a', 'b', ‘c']
numbers = [0, 1, 2]
for l, n in zip(letters, numbers):
  print("val1 "+str(l)+" val2: "+str(n))
```

Function

```
def my_function(x='default value’,y): # It could work as overloading(Υπέρφόρτωση)
  if x=='hi':
    return “5 * y” # It returns String
  else:
    return 10 * y # It returns int

my_function("hi",2)
my_function(y=2,x=“hi")
```

Python is a dynamically typed language, so the concept of overloading simply does not apply to it.
If the number of arguments is unknown, add a * before the parameter name:

```
def my_function2(*kids): # Could work as overloading. kids can be different types
  print("The youngest child is " + kids[2])

my_function2("Emil", "Tobias", "Linus")

def my_function3(country = "Norway"):
  print("I am from " + country)
```

lambda

lambda arguments : expression

small anonymous function that can take any number of arguments, but can only have one expression. Anonymous function is required for a short period of time.

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Anonymous function inside another function

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Class

Python classes provide all the standard features of Object Oriented Programming Language.
A Python class defines a **data type** , which contains variables, properties and methods. A class describes the abstract characteristics of a real-life thing.
self (Python) = this (Java)

```
class CSStudent:
    stream = 'cse'                          # Class Variable
    def __init__(self,name,roll):
        self.name = name                    # Instance Variable
        self.roll = roll                    # Instance Variable
```

Objects

Python's Objects are **instances** of classes created at run-time.
All Python objects have a **unique identity** . The built-in function id() returns the identity of an object as an integer. This integer usually corresponds to the object's location in memory, although this is specific to the **Python** implementation and the platform being used. The "is" operator compares the identity of two objects.

Access modifiers

```
class employeePublic:
    def __init__(self, name, sal): #constructor
        self.name=name  # public attribute
        self.salary=sal # public attribute

class employeeProtected:
    def __init__(self, name, sal):
        self._name=name  # protected attribute
        self._salary=sal # We are all adults here

class employeePrivate:
    def __init__(self, name, sal):
        self.__name=name  # private attribute
        c__salary=sal # private attribute
    def __fun(self):
        print("Private method”)
```

@ classmethod

A method that is bound to a class rather than its object. It doesn't require creation of a class instance

```
class person:
    totalObjects=0
    def __init__(self):
        person.totalObjects=person.totalObjects+1

    @classmethod
    def showcount(cls): # cls, which refers to the person class
        print("Total objects: ",cls.totalObjects)

p1=person()
p2=person()
person.showcount()
p1.showcount() # method can be called using an object also
```

@ staticmethod

Static method **knows nothing about the class** and **just deals with the parameters**
Class method works with the class since its parameter is always the class itself.
A static method does not receive an implicit first argument.
A static method is bound to the class and not the object of the class.
A static method **can't access or modify class state.**
It is present in a class because it makes sense for the method to be present in class.

```
class person:
    @staticmethod
    def greet(name):
        print("Hello!"+name)
p1.greet()
```

Inheritance

Python **Inheritance** enable us to define a class that takes all the functionality from parent class and allows us to add more. Inheritance is used to specify that one class will get most or all of its features from its **parent class**.
Python supports **multiple inheritances**, unlike Java
Python does **not** have any equivalent of **interfaces**

```
class Student(Person, SecondParrent):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the
class of", self.graduationyear)
```

Abstract Classes: Inherent from **ABC** and decorator @abstractmethod

Override

In Python method overriding (Υπέρβαση) occurs by **simply defining in the child class a method with the same name of a method in the parent class**. When you define a method in the object you make this latter able to satisfy that method call, so the implementations of its ancestors do not come in play.

```
class Parent(object):
        def __init__(self):
            self.value = 4
        def get_value(self):
            return self.value

class Child(Parent):
        def get_value(self):
            return self.value + 1
```

Polymorphism(1/2)

Polymorphism means same function name is used for different types.

1. Polymorphism out of Inheritance.
   Python can use **two different class types**, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. **We assume that these methods actually exist in each class**.

2. Polymorphism with Inheritance like Java

3. Polymorphism with a Function and objects:
   It is also possible to create a function that can take any object, allowing for polymorphism.

Polymorphism(1/2)

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")
    def language(self):
        print("Hindi is the most widely spoken language of India.")
    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")
    def language(self):
        print("English is the primary language of USA.")
    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Polymorphism(2/2)

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")
    def language(self):
        print("Hindi is the most widely spoken language of India.")
    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")
    def language(self):
        print("English is the primary language of USA.")
    def type(self):
        print("USA is a developed country.")

def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()
func(obj_ind); func(obj_usa)
```

Exceptions

```
try:
  print(x)
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except NameError: #the try block raises a NameError
  print("Variable x is not defined")
except:
  print("Something went wrong") #general any exception
else: #a block of code to be executed if no errors were raised
  print("Nothing went wrong")
finally: #will be executed regardless if the try block raises an error or not
  print("The 'try except' is finished")

if x < 0:
  raise Exception("Sorry, no numbers below zero")

class MyError(Exception): #User-defined Exceptions
    def __init__(self, value): # Constructor or Initializer
        self.value = value
    def __str__(self): # __str__ is to print() the value
        return(repr(self.value))
```

Files

```
f = open("demofile2.txt", “a") #append
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())

import pickle #serializing & de-serializing of python objects
example_dict = {1:"6",2:"2",3:"f"}
pickle_out = open("dict.pickle","wb")
pickle.dump(example_dict, pickle_out)
pickle_out.close()
pickle_in = open("dict.pickle","rb")
example_dict = pickle.load(pickle_in)
print(example_dict)
```

Threads(1/2)

```
import thread
def function_to_run( threadName, delay): #A function for the thread
      print "%s: %s" % ( threadName, time.ctime(time.time()) )

try: # Create two threads as follows
   thread.start_new_thread( function_to_run, ("Thread-1", 2, ) )
   thread.start_new_thread( function_to_run, ("Thread-2", 4, ) )
except:
   print (“Error: unable to start thread")
```

Threading Module: Define a new subclass of the *Thread* class

run() – The run() method is the entry point for a thread.

start() – The start() method starts a thread by calling the run method.

join([time]) – The join() waits for threads to terminate.

isAlive() – The isAlive() method checks whether a thread is still executing.

getName() – The getName() method returns the name of a thread.

setName() – The setName() method sets the name of a thread.

Threads(2/2)

```
import threading
class myThread (threading.Thread):
   def __init__(self, threadID, name, counter): # To add additional arguments
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self): #To implement what the thread should do when started
      print "Starting " + self.name
      threadLock.acquire() # Get lock to synchronize threads
      print_time(self.name, self.counter, 3)
      threadLock.release() # Free lock to release next thread
def print_time(threadName, delay, counter):
   print "%s: %s" % (threadName, time.ctime(time.time()))
threadLock = threading.Lock()
threads = []
thread1 = myThread(1, "Thread-1", 1) # Create new threads
thread2 = myThread(2, "Thread-2", 2)
thread1.start(); thread2.start() # Start new Threads
threads.append(thread1); threads.append(thread2) # Add threads to thread list
for t in threads: # Wait for all threads to complete
    t.join() # waits for threads to terminate
print "Exiting Main Thread"
```

arrays(lists)

```
cars = ["Smart", "Volvo", "BMW"]
x = cars[0] #Get the value of the first array item
cars[0] = "Mini Cooper"# Modify the value of an array item
l = len(cars) #The number of elements in the cars array
for x in cars: #Looping Array Elements
  print(x)
cars.append("Honda") #add an element to an array
cars.pop(0) #To remove an element from the array
cars.remove("Volvo") #to remove an element. no equals(Java)
cars.clear() #Removes all the elements from the list
cars.copy() #Returns a copy of the list
cars.count("Smart") #Returns a copy of the list
cars.extend("Chrysler") #Add the elements of an iterable, to the end
cars.insert("VW") #Adds an element at the specified position
cars.reverse() #Reverses the order of the list
cars.sort() #Sorts the list
list_b=[2,3,45,6, True,"John",True,datetime.datetime.now()]
```

List Comprehensions

```
new_list = [expression(i) for i in old_list if filter(i)]
```

tuple & set

A tuple is a collection which is ordered and unchangeable

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango")
print(thistuple[2:5])
thistuple[-4:-1]
if "apple" in thistuple:
myit = iter(thistuple) # to get an iterator
print(next(myit))
```

A set is a collection which is unordered and unindexed (Java: HashSet)

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
thisset.add("orange")
thisset.update(["orange", "mango", "grapes"]) #Add multiple items to a set
thisset.remove("banana")
x = thisset.pop() #Remove the last item
set3 = set1.union(set2) #intersection() difference() symmetric_difference()
```

dictionary

Collection which is unordered, changeable and indexed (Java:HashMap)

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
x = thisdict.get("model")
thisdict["year"] = 2018 # Change Value
for x in thisdict: #Loop Through a Dictionary
  print(x) # Print all key names in the dictionary, one by one
  print(thisdict[x]) # Print all values in the dictionary
thisdict.values() #to return values of a dictionary
for x, y in thisdict.items(): #Loop through both keys and values
  print(x, y)
if "model" in thisdict: #Check if "model" is in the dictionary
thisdict["color"] = "red" #Adding an item to the dictionary
thisdict.pop("model") #removes the item with the specified key name
thisdict.popitem() # removes the last inserted item
del thisdict["model"] #removes the item with the specified key name
```

Json

storing and exchanging data From JSON string -> dict

```
import json
x = '{ "name":"John", "age":30, "city":"New York"}' # some JSON:
y = json.loads(x) # parse x:
print(y["age"]) # the result is a Python dictionary:
```

From dict, list, tuple, string, int, float, True, False -> JSON str

```
x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}
print(json.dumps(x)) #from dict to JSON
```

mysql

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase")

mycursor = mydb.cursor()
sql = "SELECT * FROM customers WHERE address LIKE '%way%'"
mycursor.execute(sql)

myresult = mycursor.fetchall()
for x in myresult:
  print(x)

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)
mydb.commit()
```

Module

A module is **a file consisting of Python code**. A module can define **functions**, **classes** and **variables**. It is a ***module_file.py***

We can put **multiple classes** in single module to make it easy to read and follow the flow of the program.

A **package** is a collection of python modules under a common namespace.
A directory of Python module(s)

import module1
from module import function

Python interpreter searches for the module in the following sequences –

- The **current directory**
- If the module isn't found, It searches each directory in the shell variable **PYTHONPATH**.
- If all else fails, Python checks the **default path**. On UNIX,default path is: /usr/local/lib/python/