**Python lessons**

1. To print : print(“Hello world!”)

Substitute parameters:

name = "John"

age = 23

print("%s is %d years old." % (name, age))

%s, %d, %f, %.2f -> prints float number upto 2 decimal places, %x/%X -> hexadecimal numbers

1. Multiple assignment: a,b = 3,4
2. No need to declare data type: x=156.0 or y=”Hello”
3. Mixing operators between numbers and strings is not supported: print(2+”ok”)

If statement. No curly braces, only indentation using tab:

a=1

if a == 1:

print(“yes”)

elif a<1:

print(“less”)

else :

print(“more”)

1. Lists in python.
2. We can append any object in the list.
3. Accessing index not in list, generates an exception(Index error)
4. Add lists using + operator: list1 + list2
5. Repeat a list: list1\*2

newList = [], newList=[1,2,3]

newList.append(1)

print(newList[1])

list1 = [1,2]

list2 = [3,4]

list3 = list1 + list2

print(list3) -> [1,2,3,4]

list1 = list1\*2

print(list1) -> [1,2,1,2]

1. 2\*\*3 is 2 raised to power 3 and 2\*3 is multiplication of 2 and 3
2. “hello”\*3 gives a string “hellohellohello”
3. (1,2,”name”) -> tuple. It is a fixed size list and is immutable.
4. len(“name”) -> 4, “name”.index(“a”) -> 1
5. String slicing

astring=”hello”

astring[1:4] -> ell, astring[1:] -> ello, astring[:4]->hell , astring[-4:-1]->ell, astring [::2]->hlo, astring[::-1]->olleh

1. String splitting

astring=”hello world!”

astring.split(“ “)->[“hello”, “world!”]

1. Conditional operators

a=1 b=2

|  |  |
| --- | --- |
| if a==1 and b<3 :  print(“true”) -> true | if a>1 or b<3 :  print(“false”) -> false |

1. “in” operator

name = "John"

if name in ["John", "Rick"]:

print("Your name is either John or Rick.")

1. “is” operator compares two objects

x = [1,2,3]

y = [1,2,3]

print(x == y) -> True

print(x is y) -> False

1. “not” operator inverts the Boolean expression. It makes true all false expressions. Following are considered false:-
2. An empty string: ""
3. An empty list: []
4. The number zero: 0
5. The false boolean variable: False
6. For loop

names=[“rick”, “john”]

for name in names:

print(name)

for x in range(5):

print(x) #prints 0 to 4

else:

print(“loop terminated at %d“%x) #prints 4

else clause is executed when for condition fails. It is skipped only in the case of break.

1. Defining functions

def sum\_two\_numbers(a,b):

return a+b #custom functions definition

a=2

b=3

print(“the sum of %d and %d is %d”%(a,b, sum\_two\_numbers(a,b))) #calling the function

The function must be defined and then called.

1. Defining Classes and objects

class MyClass:

variable = "blah"

def function(self):

print("This is a message inside the class.")

myobject = MyClass()

myobject.function() #prints This is a message inside the class.

print(myobject.variable) #prints blah

1. Dictionaries have key-value pairs

phonebook = {}

|  |  |  |
| --- | --- | --- |
| phonebook["John"] = 938477566  phonebook["Jack"] = 938377264  phonebook["Jill"] = 947662781 | **OR** | phonebook = {  "John" : 938477566,  "Jack" : 938377264,  "Jill" : 947662781  } |

print(phonebook) #prints {'Jack': 938377264, 'John': 938477566, 'Jill': 947662781}

same key can’t be added, only the value updates

**Iterating a dictionary**

phonebook = {"John" : 938477566,"Jack" : 938377264,"Jill" : 947662781}

for name, number in phonebook.items():

print("Phone number of %s is %d" % (name, number))

Deleting item from dictionary: phonebook.pop(“John”) or del phonebook[“John”]

To check for a key:

If “John” in phonebook:

print(“John is here”)

if “John” not in phonebook:

print(“john is not here”)

1. Importing other modules in current workspace:

Eg. To import draw.py

import draw

usage: draw.draw\_circle(a) #calling draw\_circle function written in draw.py file

to import draw\_circle from draw.py

from draw import draw\_circle as circle #custom import name for the function in this workspace.

Custom import name

if visual\_mode:

# in visual mode, we draw using graphics

import draw\_visual as draw

else:

# in textual mode, we print out text

import draw\_textual as draw

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

1. Extending module path

You could either use the environment variable PYTHONPATH to specify additional directories to look for modules in:

PYTHONPATH=/foo python game.py OR sys.path.append("/foo")

This will execute the file game.py and will enable the script to load modules from the foo directory as well as the local directory.

1. To look for modules implemented in a package, use dir

dir(module\_name) #prints smthng like this: [‘a’,’b’,’c’]

to know more, help(module\_name)

1. Every package in python that is to be included must have \_\_init\_\_.py file. It could be empty too.

Import foo.bar #foo is the package name and bar is the module

If we want to expose only a few modules and not all a package, then we edit \_\_init\_\_.py, overriding \_\_all\_\_ variable:

\_\_init\_\_.py:

\_\_all\_\_ = ["bar"]

1. To find a certain string in the words like in a list:

list1 = [‘hello’, ‘ world’, ‘how’, ‘are’,’you’]

list2=[]

for x in list1:

if “find” in x:

list2.append(x)

print(list2) #this prints a list of all the words in the list1 that contain word “find”

1. **Numpy Arrays**

Great alternatives to python lists, operations on numpy arrays is computationally fast and efficient.

height = [1.87, 1.87, 1.82, 1.91, 1.90, 1.85]

weight = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]

import numpy as np

# Create 2 numpy arrays from height and weight

np\_height = np.array(height)

np\_weight = np.array(weight)

bmi = np\_weight / np\_height \*\* 2 #use arrays as parameters in the equation to compute the result on every element

To subset in numpy arrays:

bmi > 23 # returns Boolean result bmi[bmi > 23] #subsets the array to include only elements that satisfy the condition

1. **Pandas DataFrames**

Pandas is a high-level data manipulation tool developed by Wes McKinney. It is built on the Numpy package and its key data structure is called the DataFrame. DataFrames allow you to store and manipulate tabular data in rows of observations and columns of variables.

Eg. dict = {"country": ["Brazil", "Russia", "India", "China", "South Africa"],

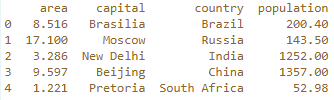
"capital": ["Brasilia", "Moscow", "New Delhi", "Beijing", "Pretoria"],

"area": [8.516, 17.10, 3.286, 9.597, 1.221],

"population": [200.4, 143.5, 1252, 1357, 52.98]}

import pandas as pd

brics = pd.DataFrame(dict) #prints below output



Indexing has been done as 0,1,etc. To change this: brics.index = ["BR", "RU", "IN", "CH", "SA"]

Or to read the data from csv file:

brics = pd.read\_csv(“country.csv”) #read data from a csv file

1. Indexing Pandas DataFrame

brics[‘area’] #gives pandas series

brics[[‘area’]] #gives pandas dataFrame

brics[0:4] #gives only 0-3 rows of the dataFrame

cars.iloc[2]) # Print out observation for 2nd index

cars.loc[['AUS', 'EG']] # Print out observations for Australia and Egypt

1. List Comprehensions

Eg. word\_lengths = [len(word) for word in words if word != "the"]

1. Multiple arguments Functions \*therest is a tuple here. To know length of tuple, len(tuplename)

therest is the varargs here

|  |  |
| --- | --- |
| def foo(first, \*therest) :  print(“first:%s”%first)  print(“rest.. %s”%list(therest))  foo(1,2,3,4) | first:1  rest..[2,3,4] |

1. Use functions as arguments

# \*\*options is used to access the elements of the tuple by the keyword

def bar(first, second, third, \*\*options):

if options.get("action") == "sum":

print("The sum is: %d" %(first + second + third))

if options.get("number") == "first":

return first

result = bar(1, 2, 3, action = "sum", number = "first")

print("Result: %d" %(result))

#o/p: The sum is 6

Result: 1

#action provides the function to be performed and first is the variable to be accessed, this can be changed as per need

1. Generators

Generators are used to create iterators, but with a different approach. Generators are simple functions which return an iterable set of items, one at a time, in a special way. When an iteration over a set of item starts using the for statement, the generator is run. Once the generator's function code reaches a "**yield**" statement, the generator yields its execution back to the for loop, returning a new value from the set. The generator function can generate as many values (possibly infinite) as it wants, yielding each one in its turn. Example below:-

|  |  |
| --- | --- |
| import random  def lottery():  # returns 6 numbers between 1 and 40  for i in range(6): #output ->  yield random.randint(1, 40)  # returns a 7th number between 1 and 15  yield random.randint(1,15)  for random\_number in lottery():  print("And the next number is... %d!" %(random\_number)) |  |

1. Exception handling:

def print\_any\_number():

try:

print(“hey what’s this”)

except Exception: #any exception could be here eg NameError

print(“hell”)

1. Sets in python: sets are lists with no duplicate entries

print(set("my name is Eric and Eric is my name".split())) #{'my', 'Eric', 'name', 'and', 'is'}

a = set(["Jake", "John", "Eric"])

b = set(["John", "Jill"])

To find common elements between two sets: a.intersection(b)

To find elements only in one set : a. symmetric\_difference(b)

a-b: a.difference(b)

union of all sets: a.union(b)

1. To convert json string into json object

import json

jsonObject = json.loads(jsonstring) #converts json string into json object

jsonString = json.dumps(jsonObject) #converts json object to json string

To add a parameter to jsonObject: jsonObject[“age”]=23

To access: jsonObject[“age”]

Python supports a Python proprietary data serialization method called pickle (and a faster alternative called cPickle). It is used the same way.

import pickle

pickled\_string = pickle.dumps([1, 2, 3, "a", "b", "c"])

print(pickle.loads(pickled\_string))

1. Creating partial functions, to limit the functionality

**from functools import partial**

def func(u,v,w,x):

return u\*4 + v\*3 + w\*2 + x

f1 = partial(func, 3, 2, 3) #u=3,v=2,w=3 creating partial function f1 out of func with limited number of parameters

print(f1(36)) #x=36

1. Code Introspection

help()

dir()

hasattr()

id()

type()

repr()

callable()

issubclass()

isinstance()

\_\_doc\_\_

\_\_name\_\_

1. Closures

Variables in the function are only read-only in their nested functions. To modify them, use nonlocal keyword.

def print\_msg(number):

def printer():

"Here we are using the nonlocal keyword"

nonlocal number #allows modifying the external variable

number=3

print(number)

printer()

print(number)

print\_msg(9)

Closures can avoid use of global variables and provides some form of data hiding.(Eg. When there are few methods in a class, use closures instead)

1. Decorators allow us to make simple modifications to callable objects like [functions](http://www.learnpython.org/en/Functions), [methods, or classes](http://www.learnpython.org/en/Classes%20and%20Objects). A decorator is just another function which takes a functions and returns one.

def repeater(old\_function):

def new\_function(\*args, \*\*kwds):

old\_function(\*args, \*\*kwds) # we run the old function

old\_function(\*args, \*\*kwds) # we do it twice

return new\_function # we have to return the new\_function, or it wouldn't reassign it to the value

@repeater

def multiply(num1, num2):

print(num1 \* num2)

multiply(2, 3)

@check

def check(old\_function):

def new\_function(arg):

if arg < 0: raise (ValueError, "Negative Argument") # This causes an error, which is better than it doing the wrong thing

old\_function(arg)

return new\_function

1. Tuples

Tuples are immutable lists

Named tuple

from collections import namedtuple

Car = namedtuple(‘Car’, ‘color’, ‘mileage’)

my\_car = Car(‘red’, ‘3456.7’)

print(my\_car.color) #red

print(my\_car.mileage) #3456.7

print(my\_car) #Car(color=’red’,mileage=3456.7)

my\_car.color = ‘blue’

1. A mixin is a special kind of multiple inheritance. There are two main situations where mixins are used:

* You want to provide a lot of optional features for a class.
* You want to use one feature in a lot of different classes.

Mixins are supported via [multiple inheritance](https://en.wikipedia.org/wiki/Multiple_inheritance).  mixin is independent enough that it doesn’t feel the same as a parent class. Mixins aren’t generally used on their own but aren’t abstract classes either.

1. “**with**” statement

It’s handy when you have two related operations which you’d like to execute as a pair, with a block of code in between. The classic example is opening a file, manipulating the file, then closing it:

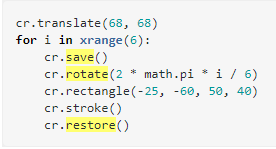
**with** **open**('output.txt', 'w') **as** f:

f.write('Hi there!')

The above with statement will automatically close the file after the nested block of code. The advantage of using a with statement is that it is guaranteed to close the file no matter *how* the nested block exits. If an exception occurs before the end of the block, it will close the file before the exception is caught by an outer exception handler. If the nested block were to contain a return statement, or a continue or break statement, the with statement would automatically close the file in those cases, too. If the class doesn’t support **with** statement, there are two ways to support the with statement: by implementing a context manager class or by writing a generator function.

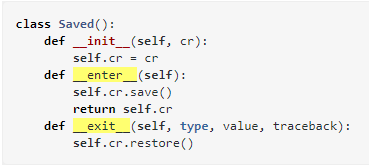
**Read more :** <https://preshing.com/20110920/the-python-with-statement-by-example/>

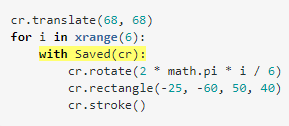
Example to understand ‘with’:



* **Implementing the Context Manager as a Class**

define a class containing an \_\_enter\_\_and \_\_exit\_\_ method





The Saved object is considered to be the [context manager](http://docs.python.org/reference/datamodel.html#context-managers).

[**Context manager**](http://docs.python.org/reference/datamodel.html#context-managers): A *context manager* is an object that defines the runtime context to be established when executing a ‘[with](https://docs.python.org/3/reference/compound_stmts.html#with)’ statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the ‘with’ statement (described in section [The with statement](https://docs.python.org/3/reference/compound_stmts.html#with)), but can also be used by directly invoking their methods. Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

* **Implementing the Context Manager as a Generator**

write a [generator function](http://docs.python.org/tutorial/classes.html#generators)

