Python Revision

## 1. Description of python:

High level, general purpose programming language. When I say high level, I mean that there is emphasis on code readability.

a. **Open source**: Platform independent and has libraries for almost everything.

b. **Interpreted**: It is an interpreted (creates objects during run time instead of during compilation)

c. **Dynamic** type memory allocation. (Allocation is done during program execution)

d. **Integrated:** It can be easily integrated with other languages like C, C++, Java, etc.

e. **Interactive**: Command line shell gives immediate feedback.

f. **Object oriented**: Programs are organized around data rather than functions and logic.

g. **Scalable**: Provides better structure and support for large programmes

h. **Database**: Provides interface to all major commercials databases

i. **Inheritance** is a mechanism in which one class acquires the property of another class. For example, a child inherits the traits of his/her parents. With inheritance, we can reuse the fields and methods of the existing class.

j. **Exception** **handling** – It is a mechanism which allows us to catch some expected errors and avoid crashing of your programs.

Exception is an error that happens during execution of a program. This avoids program to crash.

When you know your code can produce an error then you use this. Let’s say you code requires users to insert some integers. Here you know if the user inputs a string it would throw an error. In such scenarios you use exception handling.

In Python, exceptions can be handled using a try statement.

The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

try:

fh = open("testfile", "w")

fh.write("This is my test file for exception handling!!")

except IOError:

print "Error: can\'t find file or read data"

else:

print "Written content in the file successfully"

fh.close()

## 2. Data Types:

a. **Im**mutable Data types: TSN (Tuple, String, Number) – can’t be edited

b. **M**utable Data types: LSD (List, Set, Dictionary)- can be edited

Both of them store elements of different data types.

a. List []:

Lists can be edited.

Lists are **slower** than tuples.

Lists cannot be used as Dictionary keys (Do not support Hash functions)

b. Tuples ():

Tuples cannot be edited.

Tuples are faster than lists.

Tuples can be used as Dictionary keys. This is because it supports Hash functions which is used in Dictionary lookup

· Lists have order. They are ordered sequences, typically of the same type of object. Ie: all user names ordered by creation date, ["Seth", "Ema", "Eli"]

· Tuples have structure. Different data types may exist at each index. Ie: a database record in memory,

(2, "Ema", "2020–04–16") # id, name, created\_at

c. Dictionaries {}:

They are key value pairs.

They can be edited

Key has to be unique. Values can be repeated.

All three (Lists, tuples, dictionary) of them can be nested.

## 4. Shallow copy vs Deep copy:

a. Shallow: Changes made to the new copy affect the original copy. (li2 = copy.copy(li1) )

b. Deep: Changes made to the new copy do not affect the original copy. (li3 = copy.deepcopy(li1))

## 

## 5. Python Memory management

Python mainly uses dynamic memory allocation i.e memory is allocated at runtime. (For static it is at compile time). Heap (tree based data structure) is used for dynamic memory allocation. All the python objects and data structures are located on this private heap space. The programmer does not have access to it. The python memory manager takes care of this.

Everything in Python is an object.

Python also has an inbuilt garbage collector which recycles the unused memory. Whenever objects are no longer needed the python memory manager automatically reclaims memory from them.

*Compile-time is the time at which the source code is converted into an executable code while the run time is the time at which the executable code is started running*

*.*

## 6. Help and Dir

Help () is used to display the documentation.

Dir () is used to display the defined symbols.

## 7. **NumPy**

NumPy stands for ‘Numerical Python’ or ‘Numeric Python’. It is an **open source module** of Python which provides fast mathematical computation on arrays and matrices.

· Arrays are homogenous meaning they can have only the same data types. They also need to have items of the same size.

**NumPy can be imported into Python using:** *import numpy as np*

Some of the important attributes of a NumPy object are:

arr = np.array([1, 2, 3, 4, 5]) #define an array

arr=[] #empty array

np.zeros(3) – 1D array of length 3 all zeros

np.zeros((2,3))-2D array of all zeros

np.full((3,4),2) – 3×4 array with all values 2

np.random.rand(3,5) – 3×5 array of random floats between 0-1

np.ones((3,4)) – 3×4 array with all values 1

np.eye(4) – 4×4 array of 0 with 1 on diagonal

b=np.array([(1,2,3,4),(7,8,9,10)],dtype=int) #creating a 2D array

size – Returns number of elements in array

shape – Returns dimensions of array(rows, columns)

array.**dtype** – Returns type of elements in array

Addition: np.add(a,b)

Subtraction:np.subtract(a,b)

Multiplication: np.multiply(a,b)

Division: np.divide(a,b)

Exponentiation: np.exp(a)

Square Root: np.sqrt(b)

np.copy(array) – Copies array to new memory array.

view(dtype) – Creates view of array elements with typedtype

array.sort() – Sorts array

array.sort(axis=0) – Sorts specific axis of array

array.reshape(2,3) – Reshapes array to 2 rows, 3 columns without changing data.

Adding values:

np.append(array,values) – Appends values to end of array

np.insert(array,4,values) – Inserts values into array before index 4

Removing:

np.delete(array,2,axis=0) – Deletes row on index 2 of array

np.delete(array,3,axis=1) – Deletes column on index 3 of array

Combining:

np.concatenate((array1,array2),axis=0) – Adds array2 as rows to the end of array1

np.concatenate((array1,array2),axis=1) – Adds array2 as columns to end of array1

Splitting:

np.split(array,3) – Splits array into 3sub-arrays

Indexing:

a[0]=5 – Assigns array element on index 0 the value 5

a[2,3]=1 – Assigns array element on index [2][3] the value 1

Subsetting:

a[2]: Returns the element of index 2 in array a.

a[3,5] – Returns the 2D array element on index [3][5]

Slicing:

a[0:4] – Returns the elements at indices 0,1,2,3

a[0:4,3] – Returns the elements on rows 0,1,2,3 at column 3

a[:2] – Returns the elements at indices 0,1

a[:,1] – Returns the elements at index 1 on all rows

## 8. Pandas: -

Pandas is one of the most widely used python **libraries** in data science. It provides high-performance, easy to use structures and data analysis tools. Unlike NumPy library which provides objects for multi-dimensional arrays, Pandas provides in-memory 2d table object called Dataframe. It is like a spreadsheet with column names and row labels.

· Dataframes are heterogenous meaning it can store multiple data types

Hence, with 2d tables, pandas is capable of providing many additional functionalities like creating pivot tables, computing columns based on other columns and plotting graphs.

**Pandas can be imported into Python using:** *import pandas as pd*

Some commonly used data structures in pandas are:

1. **Series objects**: 1D array, similar to a column in a spreadsheet

2. **DataFrame objects:** 2D table, similar to a spreadsheet

3. **Panel objects:** Dictionary of DataFrames, similar to sheet in MS Excel

pd.read\_csv(“filename”)

pd.read\_table(“filename”)

pd.read\_excel(“filename”)

pd.read\_sql(query, connection\_object)

pd.read\_json(json\_string)

df.to\_csv(“filename”)

df.to\_excel(“filename”)

df.to\_sql(table\_name, connection\_object)

df.to\_json(“filename”)

pd.DataFrame(np.random.rand(4,3)) – 3 columns and 4 rows of random floats

df.head(n) – look at first n rows of the DataFrame.

df.tail(n) – look at last n rows of the DataFrame.

df.shape() – Gives the number of rows and columns.

df.info() – Information of Index, Datatype and Memory.

df.describe() –Summary statistics for numerical columns.

df.groupby(column) – Returns a groupby object for values from one column

df.groupby([column1,column2]) – Returns a groupby object values from multiple columns

Mean:df.mean() – mean of all columns

Median:df.median() – median of each column

Standard Deviation:df.std() – standard deviation of each column

Max:df.max() – highest value in each column

Min:df.min() – lowest value in each column

Count:df.count() – number of non-null values in each DataFrame column

Describe:df.describe() – Summary statistics for numerical columns

Histogram: df.plot.hist()

Scatter Plot:df.plot.scatter(x=’column1′,y=’column2′)

Data Cleaning:

pd.isnull() | Checks for null Values, Returns Boolean Arrray (True/ False)

pd.notnull() | Opposite of pd.isnull()

df.dropna() | Drop all rows that contain null values

df.dropna(axis=1) | Drop all columns that contain null values

df.dropna(axis=1,thresh=n) | Drop all rows have have less than n non null values

df.fillna(x) | Replace all null values with x

s.fillna(s.mean()) | Replace all null values with the mean

s.astype(float) | Convert the datatype of the series to float

s.replace(1,'one') | Replace all values equal to 1 with 'one'

s.replace([1,3],['one','three']) | Replace all 1 with 'one' and 3 with 'three'

df.rename(columns=lambda x: x + 1) | Mass renaming of columns

df.rename(columns={'old\_name': 'new\_ name'}) | Selective renaming

df.set\_index('column\_one') | Change the index

df.rename(index=lambda x: x + 1) | Mass renaming of index

df.drop(df[df['Age'] < 25].index, inplace = True)

df.pivot\_table(index=col1,values=[col2,col3],aggfunc=mean) | Create a pivot table that groups by col1 and calculates the mean of col2 and col3

df.groupby(col1).agg(np.mean) | Find the average across all columns for every

df.apply(np.mean) | Apply the function np.mean() across each column

df1.append(df2) | Add the rows in df1 to the end of df2 (columns should be identical)

pd.concat([df1, df2],axis=1) | Add the columns in df1 to the end of df2 (rows should be identical)

## Numpy v/s Pandas:

a. **Concept**: Pandas is built on top of numpy.

b. **Memory**: Numpy consumes less memory.

c. **Speed**: Numpy is faster for mathematical computation

d. **Building block**: for numpy is np.array. Building block for pandas is pd.series.

e. **Multiple data types:** Numpy arrays cannot have multiple data types. pandas dataframe can have different data types for each column.

f. **Index**: You cannot define your own index in an array but in pandas series you can.

Numpy is very fast with arrays, matrices, and math. Pandas series have indexes, sometimes it's very useful to sort or join data. Dictionaries are a slow beast, but sometimes it's very handy too. So, as it was already mentioned, it depends on the use case which data types and tools to use.

## 9. List comprehension

Syntax: - listComp = [ expression for item in list if conditional ]

Even\_num\_list = [ x for x in range(20) if x % 2 == 0]

h\_letters = [ letter for letter in 'human' ]

obj = ["Even" if i%2==0 else "Odd" for i in range(10)]

## 10. Lambda Function:

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

lambda *arguments* : *expression*

def myfunc(n):

return lambda a : a \* n

## 11. Explain how the map function works

MAP returns a map object (an iterator) which can iterate over returned values from applying a function to every element in a sequence. The map object can also be converted to a list if required.

# Return double of n

def addition(n):

return n + n

# We double all numbers using map()

numbers = (1, 2, 3, 4)

result = map(addition, numbers)

print(list(result))

Output : [2, 4, 6, 8]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Double all numbers using map and lambda

numbers = (1, 2, 3, 4)

result = map(lambda x: x + x, numbers)

print(list(result))

Output :

[2, 4, 6, 8]

## 12. Explain how the reduce function works

This can be tricky to wrap your head around until you use it a few times. Reduce takes a function and a sequence and **iterates** over that sequence. On each iteration, both the current element and output from the previous element are passed to the function. In the end, a single value is returned.

import functools

# initializing list

lis = [ 1 , 3, 5, 6, 2, ]

# using reduce to compute sum of list

print ("The sum of the list elements is : ",end="")

print (functools.reduce(lambda a,b : a+b,lis))

# using reduce to compute maximum element from list

print ("The maximum element of the list is : ",end="")

print (functools.reduce(lambda a,b : a if a > b else b,lis))

The sum of the list elements is : 17

The maximum element of the list is : 6

## 13. How does string math operation work?

Str1 = ‘hello’

Str2 = ‘world’

Str1 + Str2

>>> helloworld

'cat' \* 3

>>> catcatcat

The string is concatenated to itself 3 times.

## 14. How does list arithmetic operations work?

list1 = ["a", "b", "c"]

list2 = [1, 2, 3]

list3 = [1,0,1,0]

list4 = [5, 10, 20]

list1 +list2

>>> [a, b, c, 1, 2, 3]

list2+list3

>>> [1, 2, 3, 1, 0, 1, 0]

list2\*2

>>> [1,2,3, 1,2,3]

list1\*list2

>>> error

to multiply two list:

method 1:

res\_list = []

for i in range(0, len(list1)):

res\_list.append(list1[i] \* list2[i])

>>> [a, bb, ccc]

method 2: using list comprehension

res\_list = [list2[i] \* list4[i] for i in range(len(list2))]

>>>[5, 20, 60]

*Both list should have same number of items*

To work with elements in the list

list1 = [1, 2, 3]

list2 = [4, 5, 6]

Method1:

new\_list = []

for (item1, item2) in zip(list1, list2):

new\_list.append(item1+item2)

print(new\_list)

>>>[5, 7, 9]

Method2:

new\_list = [a + b for a, b in zip(list1, list2)]

append() Adds an element at the end of the list

clear() Removes all the elements from the list

copy() Returns a copy of the list

count() Returns the number of elements with the specified value

extend() Add the elements of a list (or any iterable), to the end of the current list

index() Returns the index of the first element with the specified value

insert() Adds an element at the specified position

pop() Removes the element at the specified position

remove() Removes the first item with the specified value

reverse() Reverses the order of the list

sort() Sorts the list

## 15. How does Sets operation work?

Set is a collection which is unordered and unindexed. In Python, sets are written with curly brackets.

It can have both numbers and letters.

Set does not allow duplicate data.

thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)

>>> {'cherry', 'orange', 'apple', 'banana'}

thisset.remove("banana")

print(thisset)

>>> {'orange', 'apple', 'cherry'}

Remove the last item by using the pop() method:

thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)

>>> {"apple", "banana"}

Sets are unordered, so when using the pop() method, you will not know which item that gets removed.

The union() method returns a new set with all items from both sets:

set1 = {"a", "b" , "c"}

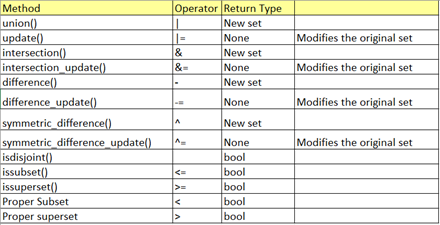
set2 = {1, 2, 3}

set3 = set1.union(set2)

print(set3)

>>>{1, 2, 'b', 'a', 3, 'c'}

*Once again we won't have any clue about the sequence.*



## 16. Dictionary working in Python

Python dictionary is an **unordered** collection of items. Each item of a dictionary has a key/value pair.

Dictionaries are optimized to retrieve values when the key is known.

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

# Dictionary Comprehension

squares = {x: x\*x for x in range(6)}

# Create a dict:

dictionary with integer keys

my\_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys

my\_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()

my\_dict = dict({1:'apple', 2:'ball'})

Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method.

If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

# get vs [] for retrieving elements

my\_dict = {'name': 'Jack', 'age': 26}

# Output: Jack

print(my\_dict['name'])

# Output: 26

print(my\_dict.get('age'))

# Trying to access keys which doesn't exist throws error

# Output None

print(my\_dict.get('address'))

# KeyError

print(my\_dict['address'])

# Changing and adding Dictionary Elements

my\_dict = {'name': 'Jack', 'age': 26}

# update value

my\_dict['age'] = 27

# For getting values

for v in spam.values():

>>> print(v)

# For getting keys

for k in spam.keys():

>>> print(k)

# For getting both

for i in spam.items():

>>> print(i)

# For getting both

for k, v in spam.items():

>>> print('Key: {} Value: {}'.format(k, str(v)))

Key: age Value: 42

Key: color Value: red

# Checking Whether a Key or Value Exists in a Dictionary

'name' **in** spam.keys()

True

# to merge two dictionaries

Step1

def Merge(dict1, dict2):

return(dict2.**update**(dict1))

>>>print(dict2)

Step2

def Merge(dict1, dict2):

res = {**\*\***dict1, \*\*dict2}

return res

clear() Removes all the elements from the dictionary

copy() Returns a copy of the dictionary

pop() Removes the element with the specified key

popitem() Removes the last inserted key-value pair

## 17. What is the difference between lists and arrays?

*Note: Python’s standard library has an array object but here I’m specifically referring to the commonly used Numpy array.*

· Lists exist in python’s standard library. Arrays are defined by Numpy.

· Lists can be populated with **different types** of data at each index. Arrays require **homogeneous** elements.

· **Arithmetic** on lists adds or removes elements from the list. Arithmetic on arrays functions per linear algebra.

· Arrays also use less **memory** and come with significantly more functionality.

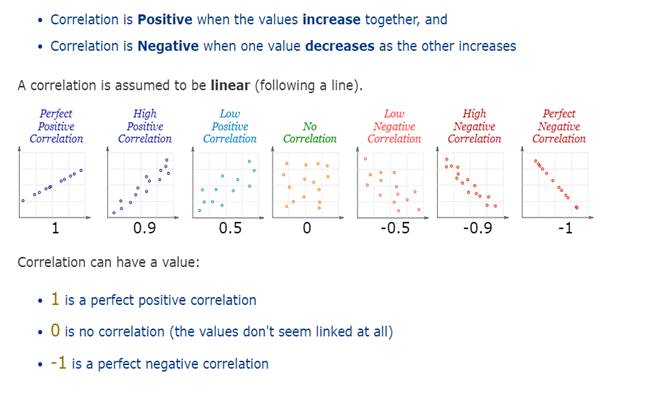
· We **concatenate** lists using (list1 + list 2) . While arrays are concatenated using np.concatenate(array1, array2)

## 

## 18. How do any() and all() work?

**Any** takes a sequence and returns true if any element in the sequence is true. **All** returns true only if all elements in the sequence are true.

## 19. Correlation ?



## 20. Generators?

## Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop.

## 21. Which is the most affected by outliers?

Mean, median and mode are measures of central tendency. Mean is the only measure of central tendency that is always **affected** by an **outlier**. Mean (the average) is the **most** popular measure of central tendency.