



Python Programming

Why Python



Simple and Easy to Learn



Python is simple and easy to learn, read and write



Free and Open Source

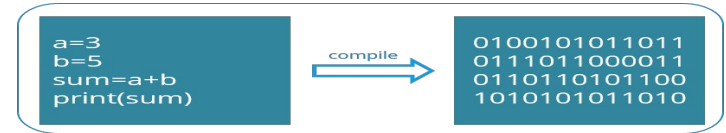


Python is an example of a FLOSS (Free/Libre and Open Source Software) which means one can freely distribute copies of this software, (such as its source code); modify it, etc.

Why Python

High-level Language

One does not need to bother about the low-level details like memory allocation, etc. while writing a Python script



Portable



Supported by many platforms like Linux, Windows, FreeBSD, Macintosh, Solaris, BeOS, OS/390, PlayStation, Windows CE, etc.

Why Python

Supports different programming paradigms

Python supports procedure-oriented programming as well as object-oriented programming



Procedure
Oriented



Object
Oriented

Extensible



C/C++



Python code can invoke C and C++ libraries, can be called from any C++ programs, can integrate with Java and .NET components

Who is Using Python

The popular YouTube video sharing system is largely written in Python



Google makes extensive use of Python in its web search system



Dropbox storage service codes both its server and client software primarily in Python



The Raspberry Pi single-board computer promotes Python as its educational language



COMPANIES USING PYTHON



BitTorrent peer-to-peer file sharing system began its life as a Python Program



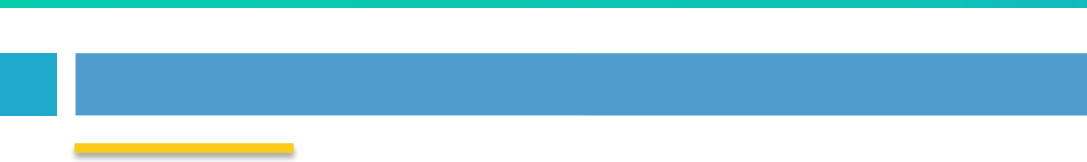
NASA uses Python for specific Programming Tasks



The NSA uses Python for cryptography and intelligence analysis

NETFLIX

Netflix and Yelp have both documented the role of Python in their software infrastructures



“The best time to plant a tree was ten years ago; the second best time is today”

-- Ancient Chinese Proverbs



Documentation

PEPs

<https://www.python.org/dev/peps/>

Refer PEP8 (coding standards), PEP20 (gives useful advice – import this)

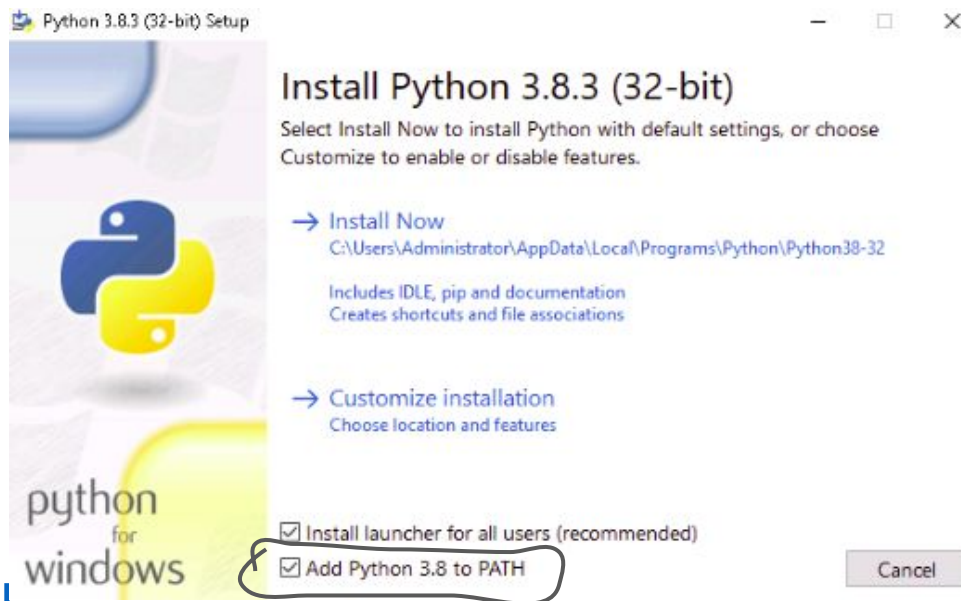
Python Documentation

<https://docs.python.org/>

Installation

<https://www.python.org/>

<https://www.anaconda.com/products/individual>



Variables

`X = 2` □ Variable. A container which holds some value

`X = X + 3`

`Y = 440 * 12`

- No data type declaration
- Mutable – list, dict, set
- Immutable – string, number, tuple
- Important data types – boolean, int, float, strings, bytes, lists, tuples, sets, dicts, None.
- Everything is an object so there are types like module, function, class, method, file, etc.
- `>>> size < 0`. True is 1 and False is 0.

`>>> True + False`

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Data Types

- Numbers – Python supports both integers and floating point numbers. There is no type declaration to distinguish them.

```
>>> type(size)    >>> isinstance()
```

- Built-in Data types

- Number (int, float, complex)
- String
- Tuple
- List
- Dict
- Set
- Boolean
- None

- Custom Data Types (An Object)

Comments and Documentation String

- ▣ # (hash) is used for single line comments
- ▣ It is format to define the documentation of a program.
- ▣ A special variable `__doc__` is used to print the documentation of a function, class or program.

```
def approximate_size(size, a_kb_is_1024=True):  
    '''Convert a file size to human-readable form.  
  
    Keyword arguments:  
    size -- file size in bytes  
    a_kb_is_1024 -- if True(default), use multiples of 1024  
                   if False, use multiples of 1000  
  
    Returns: string  
    '''
```

- ▣ It is valid only if written before the first executable command

Core Objects

- ▣ Numbers
- ▣ Strings
- ▣ List
- ▣ Tuple
- ▣ Dictionary
- ▣ Set
- ▣ Files
- ▣ None
- ▣ Boolean

Operators

- Arithmetic Operators `+`, `-`, `*`, `/`, `**`, `%`, `//`
- Comparison Operators `==`, `!=`, `>`, `<`, `>=`, `<=`
- Condition operators `and`, `or`
- Boolean Operators `True`, `False`
- Assignment Operators `=`, `+=`, `*=`, `-=` ...
- Binary Operators `<<`, `>>`, `&`, `|`, `^`
- Identity Operators `is`, `is not`
- Membership Operators `in`, `not in`

```
>>> 11 / 2
```

```
>>> 11 // 2
```

```
>>> -11 // 2
```

Operators

```
>>> 11.0 // 2
```

```
>>> 11 ** 2
```

```
>>> 11 % 2
```

- / ▫ performs floating point division.

- Lists – much more than array in java. Holds arbitrary objects and can expand dynamically as new items are added

```
A_list = [1,4, 'abc', 2.5, 'a' ]
```

Negative index accesses items from the end of the list counting backwards

Slicing a list – `a_list[1:3]`, `a_list[:3]`, `a_list[3:]`, `a_list[:]`

`A_list.append(True)`, `a_list + [0,1,2,3]`, `a_list.extend([4,5])`. Value in `a_list`, `a_list.count(4)`

- `del` – by index, `remove()` – by value.

Python Data Structures

- Tuples – A tuple is an immutable list. A tuple can not be changed in any way once it is created.
- Cant add/remove elements to tuple. Tuples have no `append()` or `extend()`
- Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of list
- Lists can't be used as dict keys but tuples can be. Since it is write protected.
- Sets – A set is an unordered “bag” of unique values. A single set can contain values of any datatype. can find union, intersection on set
 `a_set = {1}` or `a_set = {1, 2}`, `a_set = set(a_list)`
 Empty set `a_set = set()`. `A_set.add(1)`

Python Data Structures

```
a_set = {1, 2, 3}
```

```
a_set.update({2, 4, 6})
```

```
>>> a_set
```

```
{1, 2, 3, 4, 6}
```

- Dicts – unordered set of key-value pairs.
- None – special constant in Python (null value). None is not the same as False. None is not 0. None is not an empty string. Comparing None to anything other than None will always return False.
- Type(None), None == False

Operators

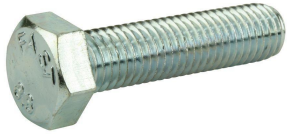
- ******
- **~ + - * / % //**
- **>> <<**
- **&**
- **^ |**
- **<= < > >=**
- **== !=**
- **%= /= //= -= += *= **=**
- **Is, is not**
- **In, not in**
- **And, or, not**

Built-in Functions

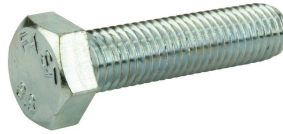
Abs(), chr(), dict(), float(), format(), help(), hex(), id(), input(), int(), isinstance(), len(), list(), map(), max(), min(), next(), open(), ord(), print(), range(), set(), sorted(), str(), sum(), super(), tuple(), type(), xrange(), zip(), etc

Relational Operator

- Relational operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Equal objects are interchangeable



`==`



True

```
>>> A = 20
```

```
>>> A == 30
```

Indentation

- Requires readable code
- No clutter
- Human and computer can't get out of sync
- Prefer four spaces. Never mix spaces and tabs
- Be consistent on consecutive lines

Conditional Statements

```
if Expression:  
    print('It is True')
```

```
if h > 50:  
    print("Greater than 50")  
else:  
    if h < 20:  
        print("Less than 20")  
    else:  
        print("Between 20 and 50")
```

- Python provides **elif** keyword to eliminate the need for nested if ... else structures in many cases.
- Flat is better than nested

For loop

Repeats a set of statements over a group of values

```
for var in group_of_values:  
    print('Statement inside for loop')
```

The range specifies a range of integers

```
range(n)  
range(n, m, s)
```

```
for x in range(10):  
    print(x)
```

```
for x in range(5, 0, -1):  
    print(x)
```

While loop

Executes a group of statements as long as a condition is True

Good for indefinite loops (repeat an unknown number of times)

while Expression:

```
print('Loop while Expression is True')
```

```
count = 5
```

```
while count:
```

```
    print(count)
```

```
    count -= 1
```

break , continue, pass

break keyword terminates the innermost loop, transferring execution to the first statement after the loop

Continue – jump to the next iteration skipping all statements after continue keyword

Pass – do nothing. It's a filler

String

String Functions

String formatting, Indexing,

Slicing.

```
name[startIndex : endIndex : Step]
```

We can use str constructor to create string representation of other types like int, float etc

```
>>> str(23)
```

String in python are sequence types. We can access using indices.

No character type. 'x' and 'a long string' both are strings

```
>>> help(str)
```

```
>>> dir(str)
```


String

Python strings are unicode. Default encoding is utf-8. Immutable sequence of unicode characters

```
>>> "first" "second"
```

Multiline Strings

Windows – carriage return + new line `\r\n`

Linux – `\n`

Python doesn't need to consider.. It has universal new line `\n`

Escape Sequences

`\n`, `\t`, `\\`, `\b`, `\r` etc

When dealing with some strings like windows paths or regular expressions we use `\` extensively. Python had option to use raw string

E.g. `path = r'C:\Users\Ganesh\Documents\sample.txt'`

String

Characters in a string are numbered with indexes starting at 0

```
>>> name = 'Dr. Reddy'
```

```
name[0], name[1]..
```

String functions

Slicing

```
name[startIndex : endIndex : step]
```

User Input

- ▣ We can instruct python to pause and read data from user using input() function. Input() functions returns a string.

```
name = input('Enter file name: ')
hd= open(name, 'r')
Counts = dict()
For line in hd:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1
```

```
Inp = input('Floor Num: ')
Actual_floor = int(Inp) + 1
```

import

- Import search path
- Import xyz – copy the module. From xyz import fun – copy fun
- Use standard boiler plate in every python program. It is mandatory in case of multithreading or multiprocessing

```
if __name__ == '__main__':
```

List

- Much more than arrays in C. Holds arbitrary data

```
lst = ['a', 1.2, 8, True, 101]
```

- **Can be indexed and slices**

```
lst[start : End : Step]
```

Just like strings, second number is “up to but not including”

- Lists can be manipulated

```
a[2] = a[2] + 22
```

```
a[0:2] = [9,8]
```

```
len(a)
```

Tuple

- Comma separated values in a bracket
- Immutable – so more efficient. Once created, can't alter its contents.
- Simple and more efficient in terms of memory use and performance than lists. If we want temp variables, prefer tuple over list
- Arbitrary items in tuple

`Mydict.items()` ▫ returns list of (key, value) tuples.

Dict

- Dictionary is a built-in object which holds key-value pair (item). Dictionary holds items in random order like hash mapping — Unordered collection
- Key → has to be immutable and unique → string, number. We index the things in dict with a lookup tag
- Values → can be anything
- Dict is mutable and dynamic in nature. No order

`D = dict(name='Bob', age=40)` -- alternate construction

`D = dict([('name', 'Bob'), ('age', 40)])`

`D = dict(zip(keylist, valuelist))`

`del d` → delete dict `del d['a']` → delete an item

Set

- Collection of unique elements – unordered collections of items. Mutable. Can't be indexed.
- No duplicates

```
>>> my_set = {2,3,5,7,8,2,3}
```

Union – $\text{set1} \mid \text{set2}$ --- combined all elements but will have common

Intersection – $\text{set} \ \& \ \text{set2}$ --- non common elements

Difference – $\text{set1} - \text{set2}$

Boolean, None, Membership

- True and False are predefined objects in Python, when comparing the values or doing comparison operator Python returns either True or False.
- None is an object in Python. It is similar to NULL in db/C++
- Membership operators test for membership in a sequence or an iterables such as strings, lists, tuples..

Object

- Objects ▫ Everything is object in Python. E.g. int, float, long, complex, etc. `var = 10.5` ▫ created new object of type float
- Functions ▫ `int()`, `float()`, `bin()`, `chr()`, `ord()`, `abs()`, `hex()`, `str()`, `len()`, etc
- floor div --> least integral value. `10.0//3=3.0`. `math.floor()`
- ceil div --> next integral value. `-(-10//3)=3`. `math.ceil()`
- mutable objects (list, dict) – we can modify an element of an object
- Immutable objects (tuple, string) – we can't modify
- operators `+` and `*` are overloaded --> meaning behaves differently based on type of obj. e.g.
`3+1=4`(add), `'ganesh'+'bhure'==>` concat
Name*2

Comprehensions

More readable and faster than standard loops (faster since it works as generators)

List comprehension – a compact way of mapping a list into another list by applying a function to each of the elements of the list.

Dict comprehension – it constructs a dictionary instead of a list

```
{value:key for key, value in a_dict.items()}
```

Set comprehension – sets just have values instead of key:value pairs

```
{x ** 2 for x in a_set}
```

No tuple comprehension

Iterators

Iterator is an object which allows a programmer to traverse through all the elements of a collection

Using iterators functions like enumerate, zip, iter, next

```
days = [ 'mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun' ]
```

```
days_short = [ 'Mo', 'Tu', 'We', 'Th', 'Sa', 'Su' ]
```

Iter -- Creates iterable obj out of a sequence. `i = iter(days).. next(i)` but more useful when iterate using a function – `for line in iter(fh.readline, "")`

Enumerate – reduces code, easy, faster and provides indexes.

Zip – used to combine sequences

Generators

- Generators - special built-in objects which generates values on demand
 - Comprehension
 - Using yield
- Generators are simple way of creating iterators

```
g = ( i for i in 'python' )
```

Functions

- Set of instructions used to perform a single and related actions.
- Function provides better modularity to a program and a high degree of freedom to reuse existing code
- Non-keyword argument should not follow keyword argument
- Cannot have non-default argument after default argument
- By default function returns None. Some rules to be followed when
 - Built-in functions
 - Object functions
 - UDF
 - Module functions

Functions

- Passing immutable objects (like string, number, tuple) acts like call-by-value. (they can't be modified by function)

```
def make_me_rich(balance):  
    balance = 1000000  
account_balance = 500  
make_me_rich(account_balance)  
Print(account_balance)    ▫ prints 500
```

- Passing mutable objects (like dict, list) acts as call by reference. (they gets modified by function)

Functions

Passing mutable objects (like dict, list) acts as call by reference. (they gets modified by function)

```
def talk_to_advisor(tasks):  
    tasks.insert(0, "Publish")  
    tasks.insert(1, "Publish")  
todos = ["Graduate", "Get a job", "Profit!"]  
talk_to_advisor(todos)  
print(todos)  □ can see modified list
```

Multiple Returns

```
def powers(number):  
    return number**2, number**3
```

```
squared, cubed = powers(3)
```


Lambda Functions

- Small, anonymous functions
- Can be passed as arguments where you need a function (like a call back function)

lambda (parameters) : expression

Functional Programming Tools

- **filter(func, sequence)**

```
def func(x):
```

```
    return x%2 != 0 and x%3 != 0
```

```
filter(func, range(2,25))
```

- **map(func, sequence)**

Call func on each item and returns a list of return values

- **reduce(func, sequence)**

Return a single value.

Call binary func on the first two values, then on the result and next item and so on

map

```
map(lambda x: x**3, [1,2,3,4])
```

```
d = [ ['ganesh',10000], ['satish',12000], ['John',11000] ]
```

```
def sal_sort(emp): return(emp[1])
```

```
d.sort(key=sal_sort, reverse=True)
```

```
d.sort(key=lambda x: x[1], reverse=True)
```

5 employeed who belong to 'IT' and having max salary --> key = lambda x: x[1] if x[2]=='IT' else 0

filters the elements of the iterables based on some function.

Filter(func, iterable) --> func returns True or False. When func returns True for any value of iterable, that value will be returns by this filter function

reduce

Reduce the iterable to a single element using some function.

E.g. if you want to find the sum of all elements of a list.. Use reduce function

```
import functools
```

```
num = [1,2,3,4]
```

```
functools.reduce(lambda x,y: x+y, num) ----> 10
```

Errors

- Syntax Errors
- Runtime Errors – Needs to be handled to avoid abrupt stops of the program. Handled using try/except
- Logical Error – Use debuggers. Import pdb can be used in program. Pdb.set_trace() OR we can use debugger provided with IDEs.

Logical error E.g.

```
a = 1000000000
if a == 1000000000:
    pass
```

Try...Except

- Try and except is a primary exception catch blocks in python which are useful to handle the runtime exceptions.
- One try can have multiple except. Try/except can be nested
- One try should have at least one except or finally block
- Try/except block can also have else block. It is called if no exception
- Finally is called always whether exception occurred or not. It is a clean up block. Else and finally can be used together. If `sys.exit()` is called, in that case also, it will go in finally block

Working with files and directories

`os.path` contains functions for manipulating filenames and directory names.

E.g.

```
os.getcwd(), os.mkdir(), os.curdir() etc
os.system("ls")
file_like_obj = os.popen('ls')
for root,dirs,files in os.walk('/file/location/')
```

Reading files

- `a_file = open('examples/demo.txt', encoding='utf-8')` # use forward slash even on windows. Path can be network mounted.
 - `a_file.encoding`, `a_file.mode`
- Python converts that sequence of bytes into a seq of characters according to specific character encoding algorithm and return seq of unicode or string.
 - `a_file.read()`

Reading files

```
with open('ex/demo.txt', encoding='utf-8') as fh:
    fh.seek(17)
    a_char= fh.read(1)
    print(a_char)
```

▣ When the with block ends, Python calls `a_file.close()` automatically

```
with open('test.log',mode='w',encoding='utf-8') as fh:
    fh.write('test succeeded')
```

```
with open('test.log',mode='a',encoding='utf-8') as fh:
    fh.write('and again')
```

```
img = open('ex/dog.jpg', mode='rb')
```

Closures and Decorators

- A function which remembers the values in enclosing scopes even if they are not present in memory.

Closure is an inner function that remembers and has access to variables in the local scope in which it was created even after the outer function has finished executing.

- modify/enhance the functions without changing their definition
- implemented as collables which takes callable and returns other callable

Decorators

A function that takes another function as an argument, adds some kind of functionality and returns another function (without altering the original function that was passed)

```
def dec_func(ori_func):  
    def wrapper_func():  
        return ori_func()  
    return wrapper_func  
  
def display():  
    print('Hello')  
  
dec_display = dec_func(display)  
dec_display()
```



```
def dec_func(ori_func):  
    def wrapper_func():  
        return ori_func()  
    return wrapper_func  
  
@dec_func  
def display():  
    print('Hello')  
  
display()
```

Modules and Packages

- Built-in modules
- User Defined Modules
- Repo Modules
- Once imported, .pyc file gets created. ▫ this pyc can be provided in prod..
- Whenever we import a module, python checks if .pyc exists, if not it gets created. If exists, it will check if .py timestamp > .pyc and reload into .pyc. Else ignore.

Manual installation of modules

```
$ python setup.py install
```

Using automation

```
$ pip or easy_install
```

Regex

- For generic search
- Used for data filtration, data cleaning and data transformation
- Regex is used for the complex searches..
E.g. extracting the IP address from a web log, getting the timestamps from log files, email validations, phone number validations, etc.
- Mostly regex operates on string.. Identify the pattern first.
- Some characters are special metacharacters..
- `^ . $ * + ? { } [] \ | ()`

Regex

- . (a period) matches any single character except newline.
- [] ▫ used to specify the character class. Other metacharacters are not active inside this.
- Backslash (\) – used to escape all the metacharacters so you can still match them in patterns
- * used for repeating things. Previous character can be matched zero or more times. E.g. ca*t match 0 or more 'a'. ct, cat, caat, caaaat etc
- + means one or more times. E.g. ca+t will match 'cat' (1 'a'), 'caaat' (3 'a's), but won't match 'ct'
- * matches zero or more times. so whatever's being repeated may not be present at all, while + requires at least one occurrence.

Regex

- $\{ \}$ □ $\{m,n\}$. There must be at least m repetitions, and at most n .
- $\{x\}$ - Repeat exactly x number of times.
- $\{x,\}$ - Repeat at least x times or more.
- $\{x, y\}$ - Repeat at least x times but no more than y times.
- e.g. $a\{6\}$ will match exactly six 'a'
- E.g. For example, $a/\{1,3\}b$ will match 'a/b', 'a//b', and 'a///b'. It won't match 'ab', which has no slashes, or 'a////b', which has four
- $\backslash w$ □ matches any single digit, letter or underscore.
- $\backslash t$ □ matches tab. $\backslash n$ □ matches new line
- $\backslash d$ □ matches decimal digit 0-9

Regex

- `re.match()` – checks for a match only at the beginning of the string
- `re.search()` – scan and checks for a match anywhere in the string. Searches for first occurrence of pattern
- `re.findall()` – Finds all the possible matches in the entire sequence and returns them as a list of strings
- `re.sub()` – substitute function. Search and replace
- `re.compile()` – Compiles a regular expression pattern into a regular expression object. Efficient when u want to reuse a pattern.

`match` checks for a match only at the beginning of the string, while `search` checks for a match anywhere in the string

Serializing Python Objects - pickle

- You have a data structure in memory that you want to save, reuse, or send to someone else. E.g. many games save the state where you left. In this case, a data structure that captures “your progress so far” needs to be stored on disk when you quit, then loaded from disk when you relaunch

- Pickle module is used for the same. It is fast, written in C.

```
import pickle
```

```
With open('data.pickle', 'wb') as f:
```

```
    pickle.dump(myDict, f)
```

- The `dump()` function in the pickle module takes a serializable Python data structure, serializes it into a binary, Python-specific format using the latest version of the pickle protocol, and saves it to an open file.

Serializing Python Objects - pickle

- ▢ Loading data from pickle file.

```
import pickle
with open('data.pickle', 'rb') as f:
    entry = pickle.load(f)
```

- ▢ Entry will be new dict created since data.pickle were loaded with dict data.

- Pickling without file

```
b = pickle.dumps(entry)
type(b)
entry3 = pickle.loads(b)
entry3 == entry
```

Serializing Python Objects - pickle

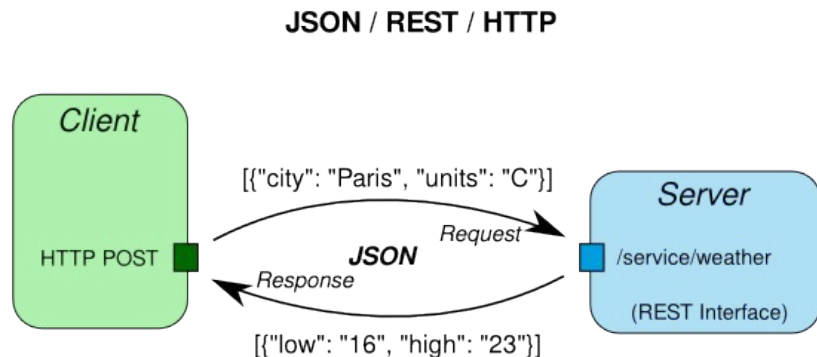
```
from sklearn.externals import joblib
```

```
joblib.dump(regressor, open('model.j', 'wb'))
```

```
model = joblib.load(open('model.j', 'rb'))
```

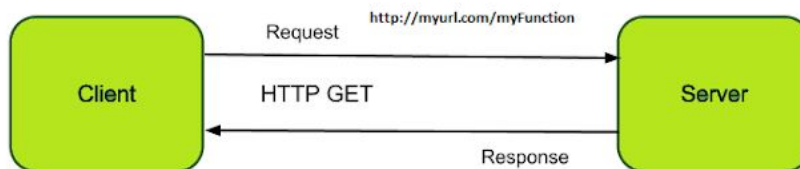
JSON

- Javascript Object Notation
- Data format. A way of structuring data to interchange.
- Created in 2001 and specs in 2006, developed based on JS syntax
- Language and platform independent



Requests

- Human friendly http library used to get requests from the web pages. One of the most downloaded library.
- Do not have to manually add query strings to URLs -- Pip install requests
- GET -> to request the data from server. `Resp = requests.get(url)`
- POST -> to submit data to be processed to the server
 `payload = {'key1': 'value1', 'key2': 'value2'}`
 `Resp = requests.post(url, data=payload)`



Numpy

NumPy is a library written for scientific computing and data analysis. It stands for ***numerical python***.

The most basic object in NumPy is the ndarray, or simply an array which is an **n-dimensional, homogeneous array**. By homogenous, we mean that all the elements in a NumPy array have to be of the same data type, which is commonly numeric (float or integer).

- core scientific computing pkg used for data analytics. -- numeric computing package. Numpy is a linear algebra - Matrix manipulation library
- It provides high performance multidimensional array objects [object for multidimensional arrays and matrices].
- provides functions to perform advanced mathematical and statistical operations on these objects

Numpy

`Np.unique(arr)`

`Arr = np.arange(20,27)`

For item in arr: -- for idx, item in enumerate(arr)

Algebra

`Mat = np.array([[1,2] [3,4]])`

`Np.linalg.inv(mat), mat.transpose()`

`Arr1 * arr2` – element by element multiplication

`Arr1.dot(arr2)` – actual matrix multiplication

1D array

```
>>> import numpy as np
>>> x = np.arange(2, 5).reshape(3)
>>> x
array([ 2, 3, 4])
>>>
```



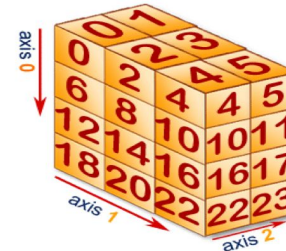
2D array

```
>>> import numpy as np
>>> x = np.arange(2, 10).reshape(2, 4)
>>> x
array([[ 2, 3, 4, 5],
       [ 6, 7, 8, 9]])
>>>
```



3D array

```
>>> import numpy as np
>>> x = np.arange(24).reshape(4, 3, 2)
>>> x
array([[[ 0, 1], [ 6, 7], [12, 13], [18, 19]],
       [[ 2, 3], [ 8, 9], [14, 15], [20, 21]],
       [[ 4, 5], [10, 11], [16, 17], [22, 23]]])
>>>
```



Numpy

$$2x + y = 1$$

$$X + y = 2$$

`M0 = np.array([[2,1], [1, 1]])` □ coefficient of left side of equations

`M1 = np.array([1,2])` □ right side of equation

`Np.linalg.solve(M0, M1)`

`Arr2 = arr1` □ a new numpy array won't be created, it's a view created. If any element changed in `arr1`, `arr2` also has changed one. `Arr2` is actually a view not COPY

`Id(arr1), id(arr2)` □ both same

`Arr2 = arr1.copy()` □ actual copy of memory gets created

Pandas

Data analysis - logical techniques to describe, evaluate data and discover useful information, provide insight, suggest conclusion and support decision making

Pandas --> python package for data analysis. It provides the builtin data structures which simplify the manipulation and analysis of data sets

Two primary data structures of pandas -

Series (1D) --> named python list -- dict with list as value. E.g. { 'grades': [50, 90, 100, 45] }

DataFrame (2D) --> dict of series

Pandas -- used to import, organise and process data -- provides tabular structure (df)

Pandas

- A tool for data analysis - logical techniques to describe, evaluate data and discover useful information, provide insight, suggest conclusion and support decision making
- Like a excel in python
- Python good for data munging and preparation. analysis and modeling to be done in R then provide back to python for further processing. Pandas good for data analysis and modeling to be done within python.
- High-performance, easy-to-use data structures and data analysis tools.
- Python package for data analysis. It provides the built-in data structures which simplify the manipulation and analysis of data sets

Pandas features

A series is similar to a 1-D numpy array, and contains scalar values of the same type (numeric, character, datetime etc.). A data frame is simply a table where each column is a pandas series.

Series, DataFrames, Panel data types

Csv, excel, sql

Handling missing data

Reshaping data

Slicing, indexing, sub-setting, merging, joining

Optimized for performance

Pandas

Data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)

two primary data structures of pandas -

Series (1D) --> named python list -- dict with list as value. E.g. { 'grades': [50, 90, 100, 45] }

Dataframe (2D) --> dict of series

```
pd.Series([1,2,3,4])
```

```
pd.Series([1,2,3,4], index=['a','b','c','d'])
```

Two primary data structures of pandas -

Series (1D) --> named python list -- dict with list as value. E.g. { 'grades': [50, 90, 100, 45] }

Dataframe (2D) --> dict of series

Pandas

Pandas -- used to import, organise and process data -- provides tabular structure (df)

```
data = { 'year': [2008, 2012, 2016], 'attendees': [112, 321, 729], 'avg_age': [24, 41, 29] }
```

```
df.DataFrame(data)
```

```
type(df['year']) --> pandas series object.
```

```
Df['year'] < 2013 --> gives boolean (True or False) all with year less than 2013.
```

```
df[ df['year'] < 2013 ] --> Boolean indexing
```

```
df.loc(idx2, idx5) --> index 2 to index 5 --> label based
```

```
df.iloc[0:3] --> position oriented
```

Pandas

Creating pandas df by importing data from csv file

```
data = pd.read_csv('file.csv')
```

```
data['country'] OR data.country
```

```
afghanistan = data[data.country == 'afghanistan']
```

set(data.continent) --> gives unique continents in the data

```
data_2007 = data[data.year == 2007] ----- asia_2007 = data_2007[data_2007.continent == 'asia']
```

```
asia_2007.gdpPerCapita.mean() ----- (and median())
```

```
plt.hist(asia_2007.gdpPerCapita) ---- create histogram
```

```
plt.show()
```

Pandas

```
import pandas as pd
```

pd.read_csv('file.csv', nrows=5) --> default indexing on dataframe. So use index_col='id' param --> column id will be used as index

```
pd.read_csv('file.csv', nrows=5, index_col='id', usecols=['id','name', ....])
```

```
COLS_TO_USE = ['id', 'name', 'year', 'dept']
```

```
pd.read_csv('file.csv', usecols=COLS_TO_USE, index_col='id')
```

```
pd.to_pickle(os.path.join '..', 'data_frame.pickle'))
```

```
plt.plot(df['year'], df['attendees'])
```

Matplotlib

- Python 2D plotting library which produces publication quality figures in a variety of hard copy formats
- A set of functionalities similar to those of MATLAB
- Used for line plots, scatter plots, barcharts, histograms, pie charts etc.

AWS Automation with boto3

- Python SDK for AWS
- Boto3 allows us to directly create, update and delete AWS services from python scripts

Pytest

Pytest is a python based testing framework, which is used to write and execute test codes. In the present days of REST services, pytest is mainly used for API testing even though we can use pytest to write simple to complex tests, i.e., we can write codes to test API, database, UI, etc.

Advantages of Pytest

The advantages of Pytest are as follows:

- Pytest can run multiple tests in parallel, which reduces the execution time of the test suite.
- Pytest has its own way to detect the test file and test functions automatically, if not mentioned explicitly.
- Pytest allows us to skip a subset of the tests during execution.
- Pytest allows us to run a subset of the entire test suite.
- Pytest is free and open source.
- Because of its simple syntax, pytest is very easy to start with.

Pytest environment setup

Command to install pytest:

Below command will install the latest version of pytest

- `pip install pytest`

Install the specific version of pytest

- `pip install pytest==6.2.4`

Command to confirm the pytest installed

- `pytest -h`

Pytest writing the basic Test

Below are the test case example of checking square root and square of the number

```
import math
def test_sqrt():
    num=25
    assert math.sqrt(num) == 5
def testsquare():
    num=7
    assert 7*7 == 40
def tesequality():
    assert 10 == 11
```

Running Test cases

Running multiple test cases at a time

- Goto the test directory path and execute the following command
- Execute the command `pytest`

Running subset of test cases

- Execute the command `Pytest -k subset name`

Running group of test cases

- Use `@pytest.mark.modulename` above the test case
- Execute the command `Pytest -m marker name`

Pytest fixtures

- Fixtures are functions, which will run before each test function to which it is applied. Fixtures are used to feed some data to the tests such as database connections, URLs to test and some sort of input data.
- Therefore, instead of running the same code for every test, we can attach fixture function to the tests and it will run and return the data to the test before executing each test

Pytest fixtures example

```
@pytest.fixture
```

```
def input_data():
```

```
    a = 10
```

```
    return a
```

```
def test_even(input_data):
```

```
    assert input_data % 2 == 0
```

A function needs to mark with

```
@pytest.fixture
```

Pytest conftest.py

- Create the new file called conftest.py
- Move the fixture code to common file called conftest.py
- All the test cases which requires the common data can have access to it
- It helps to reuse the common data

`@pytest.fixture`

`def input_data():`

`a = 10`

`return a`

A function needs to mark with

`@pytest.fixture`

Pytest parameterized test cases

- Parameterizing of a test is done to run the test against multiple sets of inputs. We can do this by using the following marker

```
@pytest.mark.parametrize("num,output",[(1,1),(2,2),(3,45),(4,4)])
```

```
def test_check_params(num, output):
```

```
    assert num == output
```

Pytest code coverage

Pkg name: **pip install coverage**

RUN Command: coverage run -m pytest directory/file name

Report: coverage report -m

HTML Report: coverage html

