# Python Beginner Doc

# Chapter 1

## Introduction

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. it has fewer syntactical constructions than other languages.

* Python is processed at runtime by the interpreter, while in some other languages program needs to compile first before executing.
* Python is interactive language. You can interact your lines of code on the python prompt.
* Python also object-oriented language, it basically encapsulates your python code with the objects.

## History

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

## Pros & Cons

### Pros

* Python is easy to use language.
* Python is dynamically typed language.
* In python, everything is object.
* Python has large community support.

### Cons

* Python is slow because of its dynamically typed nature.
* Python global interpreter lock, only a single thread can run access internals at a time.

## Where to Download?

<http://www.python.org/download/>

## How to write your first python program?

### Interactive mode programming

### **Linux –** Open the terminal and typed ‘python’

### $ python

### Python 2.4.3 (#1, Nov 11 2010, 13:34:43) [GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2 Type "help", "copyright", "credits" or "license" for more information.

### >>>

Now type

>>> print "Hello, Python!"

O/P

Hello, Python!

### Script Mode

Create a file with extension **.py**. Type the following line to the file

print "Hello, Python!"

and then save the file with name first.py. Now run the following command on terminal

python location/first.py

## Python Identifiers

Python identifiers is name used to identify function, variable, class name, object name.

### Naming conventions

* Class name start with uppercase letter
* Starting with single leading underscore, it indicates that identifiers is private.
* Starting with double leading underscore, it indicates that identifiers is strongly private.

## Python keywords

|  |  |  |
| --- | --- | --- |
| 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 |

## Lines & Indentation

Python provide no braces to indicate the code blocks. Python use indentation and it is variable, but all statements must be intended with same amount.

## Multiline statements

var = var1 + \

var2 + \

var3

also used with the list, tuple, dictionary. Do not vary about the datatypes, we will cover those in future chapter.

test = [ 1, 2, 3, 4,

5, 6]

## Comments

# This is my python script

## Multiple statement is single line

a = 1; b = 2; c=3

print a,b,c

O/P

1 2 3

## Take Input from user

* raw\_input
  + decision = raw\_input(‘Please enter your decision\n’)
  + After running the above statement user has to write its decision on terminal, his/her decision will be save in the variable **decision.**
  + raw\_input function takes user entry and its type is <str>.
* Input
  + Decision = input(‘Please enter your decision\n’)
  + After running the above statement user has to write its decision on terminal, his/her decision will be save in the variable **decision.**
  + Input function takes user entry and its type will depend on the user entry.

## Command line arguments

python test.py arg1 arg2

test.py

--start –

import sys

# This will print all the argument given

print sys.argv

# This will print the specific argument by their location

print sys.argv[0]

print sys.argv[1]

# Chapter 2

## Variable

Variables are the reserved memory location, you can assign values to them.

Example

a = 2 (a is the integer variable, a holds value 2)

a Memory Location (0xFFFF)

2

a = 100.0 (a is the float variable, a holds value 100.0)

a = ‘Cians’ (a is the string variable, a holds value Cians)

## Data Type

* Numbers
* String
* List
* Tuple
* Dictionary

### Numbers

Number data types hold numeric values.

var = 100

#### Python support 4 types of numerical types:

* int
* long
* float
* complex

### String

String is the contiguous set of characters represented in quotation mark.

var = ‘CiansAnalytics’

**print var[0] # Prints the first character of the string**

O/P

C

**print var[-1] # Print the last character of the string**

O/P

s

**print var[2:6] # Print characters from 3rd to 5th**

O/P

ans

**print var[2:] # Print all character starting from 3rd**

O/P

ansAnalytics

**print var \* 2 # Print string two times**

O/P

CiansAnalyticsCiansAnalytics

**print var + ‘ Gurgaon’ # Print the concatenation of both string**

O/P

CiansAnalytics Gurgaon

## List

A list contains items separated by commas and enclosed within square brackets ([]). List is like arrays in C.

We also can use the slice Operator on list.

list = [1,2,3,4,5,6,7,8]

**print list[0] # Print the first element of the list**

O/P

1

**print list[-1] # Print the last element of the list**

O/P

8

You can add two lists like this

a = [1,2,3,4]

b = [5,6,7]

print a + b # Print the concatenation of both the list

## Tuple

Tuple is another sequence data type that is similar to list.

#### Difference between Tuple and List

* Tuple is non-mutable, while list is mutable
  + Example

list = [1,2,3,4,5]

tup = (1,2,3,4,5)

list[0] = 5 # It will reassign the value at 0 index

print list

O/P

[5,2,3,4,5]

While in tuple this is not possible

tup[0] = 5 # It will show you the TypeError: 'tuple' object does not support item assignment

* Tuple syntax is different from list
  + Example

tup = (1,2,3,4,5)

list = [1,2,3,4,5]

tup = (1,2,3,4,5,6)

**print tup[0] # Print the first element of the tuple**

O/P

1

**print tup[-1] # Print the last element of the tuple**

O/P

6

You can also concatenate the two tuples, can also use the slice operator just like string and list.

## Dictionary

Python dictionary are kind of hash table, it consists of key-value pair. Dictionary key can be any python data type.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

dict = {} # Define the empty dictionary

dict[‘First’] = 1

dict[‘Second’] = 2

print dict # It will print the all key-value pair in the dictionary

O/P

{'Second': 2, 'First': 1}

## Data Type conversion

Int(x[,base]) Converts x to an integer. base specifies the base if x is a string

long(x [,base] ) Converts x to a long integer. base specifies the base if x is a string.

float(x) Converts x to a floating-point number.

complex(real [,imag]) Creates a complex number.

str(x) Converts object x to a string representation.

list(s) Converts s to a list.

tuple(s) Converts s to a tuple.

dict(d) Creates a dictionary. d must be a sequence of (key, value) tuples.

etc

# Chapter 3

### Types of Operator

* Arithmetic
* Comparison
* Assignment
* Logical
* Bitwise
* Membership
* Identity

### Arithmetic Operator

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| + | Addition – Add two numbers | a = 5, b = 10, a+b = 15 |
| - | Subtraction – Subtract two numbers | a = 5, b = 10, a-b = -5 |
| / | Division – Divide two numbers | a = 5, b =10, a/b = 0  a = 5.0, b=10.0, a/b = 0.5 |
| \* | Multiplication – Multiply two numbers | a = 5, b =10, a\*b = 50 |
| % | Modulus - divides the two numbers and return remainder | a = 5, b = 10, a%b = 5 |
| \*\* | Exponent – Do the exponential calculation | a = 5, b = 10, a\*\*b = 9765625 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. | a = 9.9, b = 5. a//b = 1.0 |

### Comparison Operator

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| == | Compare two values if equal then **True** Otherwise **False** | a = 5, b = 10  a == b  False |
| != | Compare two values if not equal then **True** Otherwise False | a = 5, b = 10  a != b  True |
| > | If value of LHS greater than the RHS then **True** Otherwise **False** | a = 5, b = 10  a > b  False |
| < | If value of LHS smaller than the RHS then **True** Otherwise **False** | a = 5, b = 10  a ><b  True |
| >= | If value of LHS greater than or equal to the RHS then **True** Otherwise **False** | a = 5, b = 5  a >= b  True |
| <= | If value of LHS smaller than or equal the RHS then **True** Otherwise **False** | a = 5, b = 5  a <= b  False |

## Assignment Operator

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| = | Assigns values from right side operands to left side operand | c = a + b |
| += | It adds right operand to the left operand and assign the result to left operand | c += a or c = c + a |
| -= | It subtracts right operand from the left operand and assign the result to left operand | c -= a or c = c – a |
| \*= | It multiplies right operand with the left operand and assign the result to left operand | c \*= a or c = c \* a |
| /= | It divides left operand with the right operand and assign the result to left operand | c /= a or c = c/a |
| %= | It takes modulus using two operands and assign the result to left operand | c %= a or c = c % a |
| \*\*= | Performs exponential (power) calculation on operators and assign value to the left operand | c \*\*= a or c = c\*\*a |
| //= | It performs floor division on operators and assign value to the left operand | C //= a or c = c // a |

## Logical Operator

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| & Binary AND | Operator copies a bit to the result if it exists in both operands. | (a & b) = 12 (means 0000 1100) |
| | Binary OR | It copies a bit if it exists in either operand. | (a | b) = 61 (means 0011 1101) |
| ^ Binary XOR | It copies the bit if it is set in one operand but not both | (a ^ b) = 49 (means 0011 0001) |
| ~ Binary Ones compliment | It is unary and has the effect of 'flipping' bits | (~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number. |

## Logical Operator

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| and | If both the operands are True | True and True  True |
| or | If one of the two operands are True | Ture and False  True |
| not | Inverse operands | not True  False |

## Membership Operator

Python membership operator test the membership in the given sequence

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| in | Evaluates to True if it finds a variable in the specified sequence and false otherwise. | a = [1,2,3,4]  1 in a  True |
| not in | Evaluates to False if it finds a variable in the specified sequence and false otherwise. | a = [1,2,3,4]  8 in a  True |

## Identity Operator

Python identity operator compare the memory location of two objects.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. | a = 5, b = 5  a is b  True |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. | a = 5, b = 10  a is b  True |

## Operator precedence

|  |
| --- |
| \*\* |
| ~ ,+, - |
| \* ,/, %, // |
| +, - |
| >>, << |
| & |
| ^ ,| |
| <=, <, >, >=, |
| <>, ==, !=, |
| =, %=, /=, //=, -=, +=, \*=, \*\*= |
| is, is not |
| in, not in |
| not, or and |

# Chapter 4

Note: Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value

### If else Block

Example

a = 5

if a == 5: # This line will compare the LHS with RHS

print True

else:

print False

O/P

True

Else condition block

False

If Condition block

True

Condition

## If elif else block

## 

a = int(raw\_input(‘Enter the Number’))

if a == 10:

print ‘Number is: 10’

elif a == 100:

print ‘Number is: 100’

else:

print ‘Number is other than 10 and 100’

OP

Enter the Number

10

Number is: 10

# Chapter 5

## Loops

A loop statement allows us to execute a statement or group of statements multiple times.

### While loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

Start

Condition

Conditional block

Execute until condition is True

Example

Problem Statement: print numbers from 1 to 10 using while loop

--start—

status = True

i = 1

while status:

print i

if i == 10:

status = False

i++

--end—

OP

1

2

3

4

5

6

7

8

9

10

## Infinite loop

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

Example

--start—

var = 1

while var == 1:

print ‘Your Number is: ’+str(var)

print ‘End of the loop’

--end—

OP

Your Number is: 1

Your Number is: 1

Your Number is: 1

.

.

to infinite

Note: You can also use the else statement with while loop

Example

--start—

user = int(raw\_input(‘Enter any number\n’))

while user == 10:

print ‘You enter the correct number’

else:

print ‘You entered the wrong number’

--end—

OP

Enter any number

20

You entered a wrong number

## For loop

For loop is used to iterate over a sequence like list, tuple, string.

### Syntax

--start—

for iterate in sequence:

print iterate

--end—

OP

Above example will print the all values one by one from the sequence

### Block Diagram

Start

for iterate in sequence

Item from sequence

Execute the statement

Iterate for next item print the the iterate If no item left

Example

### Iterate with String

--start—

value = ‘Cians’

for letter in value:

print letter

--end—

OP

C

i

a

n

s

### Iterate with list

--start—

value = [‘Cians’,’Analytics’,’Gurgaon’]

for name in value:

print value

--end—

OP

Cians

Analytics

Gurgaon

### Iterate with tuple

--start—

value = (‘Cians’,’Analytics’,’Gurgaon’)

for name in value:

print name

--end—

OP

Cians

Analytics

Gurgaon

## Nested loops

### while

--start—

while i == 0:

while j == 0:

print ‘Nested statement’

--end—

### for

--start—

for iterate1 in sequence1:

for iterate2 in sequence2:

print ‘Nested block’

--end—

## Loop Control statements

|  |  |
| --- | --- |
| break | Terminate the loop statement |
| pass | This statement is used when, you do not want to execute any statement |
| continue | Skip the current iteration and shift to next iteration |

# Chapter 6

## Function

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions such as print() and but you can also create your own functions. These functions are called user-defined functions

## Defining a function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

* Function blocks begin with the keyword def followed by the function name and parentheses ( ( ) ).
* Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
* The first statement of a function can be an optional statement - the documentation string of the function or docstring.
* The code block within every function starts with a colon (:) and is indented.
* The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

### Syntax

--start—

def first\_function(parameters):

------------------------------

function body

statements

------------------------------

return statement

--end—

### Example

#### Statement – create a function to add two numbers

--start—

Def add(num1, num2):

return num1 + num2

--end—

#### How to call a function

add(1,2) # You can call the add function by simply writing the function name with correct argument given inside the parentheses ().

##### Example

--start—

user = raw\_input(‘Enter the two numbers by comma separated for addition\n’).split(‘,’)

num1 = int(user[0])

num2 = int(user[1])

op = add(num1, num2)

print op

--end—

OP

Enter the two numbers by comma separated for addition

1,2

3

# Chapter 7

## Module

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

## How to create a Module

* Create a file with any name other than the python keyword and save that file with .py extension.
* Now open the file and write some python codes.

### Example

* Create a file name first\_module.py
* Write the below code into the module

--start—

def first\_mod():

print ‘This is my first module’

--end—

## How to import the module/use the module code

To use the module code, you need to import the module to current environment. So, the python interpreter knows what will going to run.

### Example

--start—

Import first\_module

First\_module.first\_mod()

--end—

OP

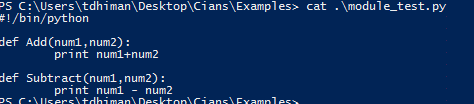
This is my first module

#### Other statements for import

* from module import abc
  + This statement will only import a single function/class from the whole module

Example

We have a module, which contain two functions **Add, Subtract** and its name is operation.py



--start—

def Add(num1, num2):

return num1-num2

def Subtract(num1, num2):

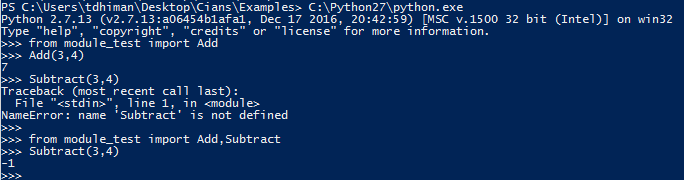
return num1-num2

--end—

Now if you write

from operation import Add

The above statement will only import the single function Add

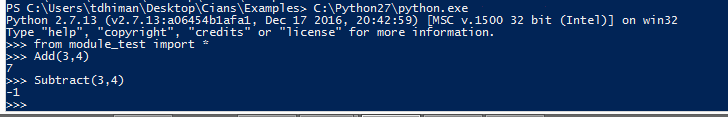


* from module import \*
  + The above statement will import all the function/class to current environment.

Example

From operation import \*

Now we have imported both Add and Subtract function



## Location Modules

When you import a module, the Python interpreter searches for the module in the following sequences

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

## Package

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and sub packages.

Start.py

Package1/

abc.py/

add()

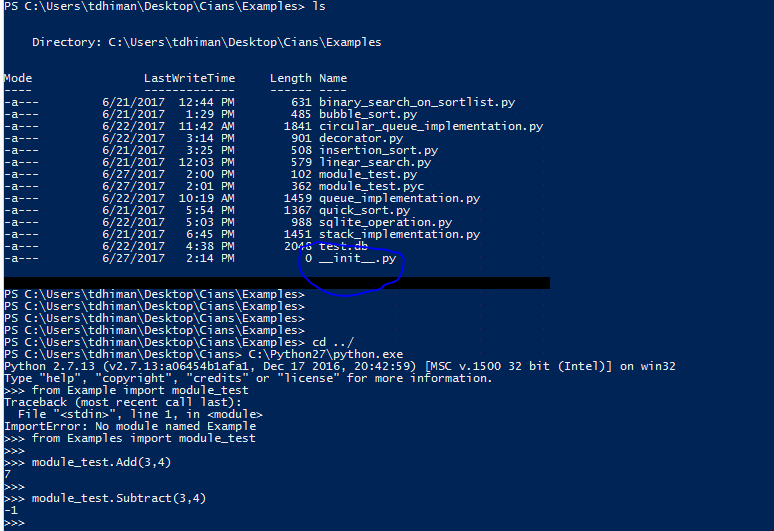
subtract()

ced.py/

multiply()

divide()

If you want to use the abc.py and ced.py module in to start.py, You need to create a file \_\_init\_\_.py into the Package1 directory.



Start.py

Package1/

\_\_init\_\_.py

abc.py/

add()

subtract()

ced.py/

multiply()

divide()

**start.py**

--start—

from Package1 import abc.py, ced.py

print abc.add(3,4)

print abc.Subtract(10,5)

print ced.mutiply(3,4)

print ced.divide(10,2)

--end—

OP

7

5

12

5

## Open and Closing files

To open a file for reading or writing, you need to create an object like this.

file = open(file\_name, access\_mode, buffering)

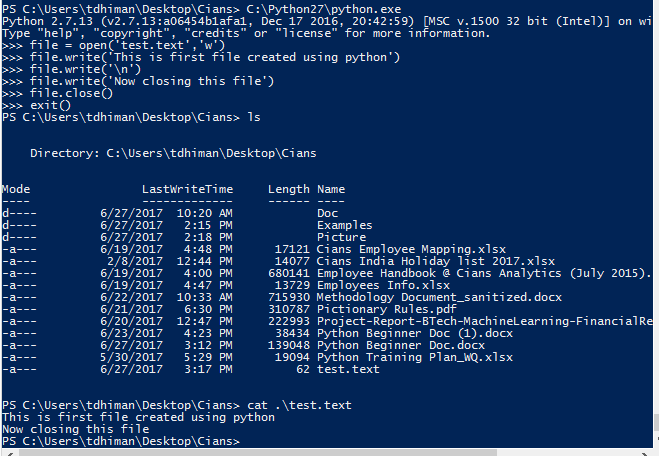
**file\_name** – It’s a string variable contain the name of the file.

**Access\_mode** – Access mode determines the mode in which the file will be opened, i.e. reading, writing, appending.

**buffering** – If buffering value is 0, means no buffering takes place, value greater than zero will indicate the buffering will be done based on defined buffered size. If value is zero or negative means buffering will not be happen.

|  |  |
| --- | --- |
| r | Open file only in read only mode |
| rb | Open file in ready only mode for binary file |
| r+ | Open file for both reading and writing |
| rb+ | Open file for both reading and writing for binary file |
| w | Open file in write only mode, if the file does not exist, then it will create the file with the file\_name |
| wb | Open file in write only mode for binary file |
| w+ | Open file for both read only and write only |
| wb+ | Open file for both read only and write only for binary file |
| a | Open file in append mode, File pointer will be at end of the file |
| ab | Open file in append mode for binary file |
| a+ | Open file in appending and reading |
| ab+ | Open file in appending and reading for binary file. |

### Example



# Chapter 8

## Exception

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them:

* Exception Handling
* Assertions

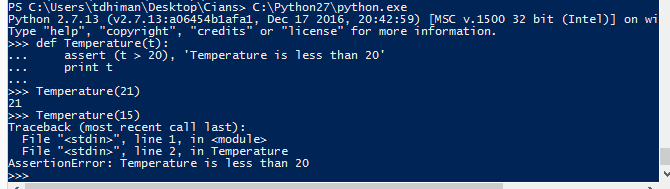
## Assertion

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program. The easiest way to think of an assertion is to liken it to a raise-if statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

### Syntax

assert (Condition check), ‘Message to print’

### Example



## Exception

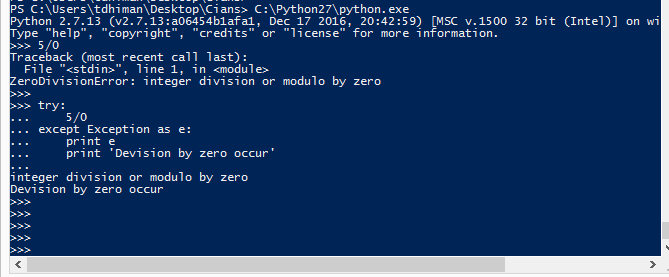
When python program encounters a situation that it cannot understand, then the python code will raise the exception.

## Handle the Exception

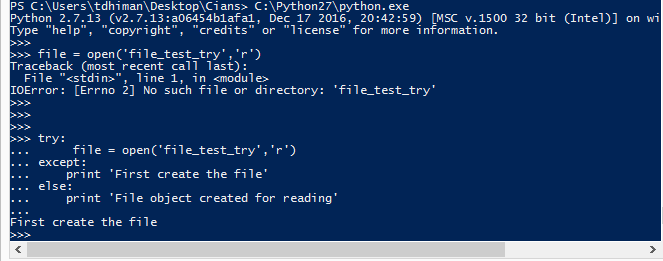
### Try and catch

Sometimes code can go wrong in future, in that case we can write a block of code over the suspicious code by using try and catch.

Examples – You are opening a file name test.txt. But this file is deleted by mistake, then you code will throw the Exception. In these kinds of cases you can catch the exception and execute the necessary remedy.



### Try catch else



You can also catch multiple type of Exceptions

--start—

try:

# block of code can throw more than a type of exception

except IOError as e:

print e

except KeyError as e:

print e

except:

Print ‘Exception occur other than the IOError or KeyError’

--end—

## try finally

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not.

--start—

try:

-----------------------

Block of code

Will throw exception

Will not executes

-----------------------

finally:

This block would always be executed

--end—

# Python Advanced Doc

Note: Please read Chapter 10 before Chapter 9 to better understand the python @property

# Chapter 9

## Python @property

Python has a great concept called property which makes the life of an object-oriented programmer much simple.

Before going deep into property first understand the need of property.

Example

**class Celsius**:  
 **def** \_\_init\_\_(self, temperature=0):  
 self.temperature = temperature  
  
 **def to\_fahrenheit**(self):  
 **return** (self.temperature \* 1.8) + 32

OP

>>> # create new object

>>> man = Celsius()

>>> # set temperature

>>> man.temperature = 37

>>> # get temperature

>>> man.temperature

37

>>> # get degrees Fahrenheit

>>> man.to\_fahrenheit()

98.60000000000001

The extra decimal places when converting into Fahrenheit is due to the floating-point arithmetic error (try 1.1 + 2.2 in the Python interpreter).

Whenever we assign or retrieve any object attribute like temperature, as show above, Python searches it in the object's \_\_dict\_\_ dictionary.

>>> man.\_\_dict\_\_

{'temperature': 37}

Now imagine if we need to implement the set temperature function that, the value of temperature cannot go below the -273 degrees Celsius.

Now we will add getter and setter to implement the situation.

**class Celsius**:  
 **def** \_\_init\_\_(self, temperature=0):  
 self.set\_temperature(temperature)  
  
 **def to\_fahrenheit**(self):  
 **return** (self.temperature \* 1.8) + 32  
  
 **def get\_temperature**(self):  
 **return** self.\_temperature  
  
 **def set\_temperature**(self, temperature):

value = to\_fahrenheit(temperature)  
 **if** value < -273:  
 **raise** ValueError("Temperature cannot go below -273")  
 **else**:  
 self.\_temperature = temperature

OP

>>> c = Celsius(-277)

Traceback (most recent call last):

...

ValueError: Temperature cannot go below -273

>>> c = Celsius(37)

>>> c.get\_temperature()

37

>>> c.set\_temperature(10)

>>> c.set\_temperature(-300)

Traceback (most recent call last):

...

ValueError: Temperature cannot go below -273

Everything is running fine with the above code, but in the above we are receiving the \_temperature variable that is private variable in python.

But in actual private variables don't exist in Python. There are simply norms to be followed. The language itself don't apply any restrictions.

>>> c.\_temperature = -300

>>> c.get\_temperature()

-300

Here the above statement, we found out that even we setup a setter function to set the value, anyone can able to assign the values which are less than -273 degrees Celsius.

We can access the private or strongly private variable of class using its object, so because of this another python rule we are not able to implement the current problem in right manner, to implement such problem we need to follow different rule of python called @property.

## The power of @property

Now we are implementing the above problem in pythonic way by using property.

**class Celsius**:  
 **def** \_\_init\_\_(self, temperature = 0):  
 self.temperature = temperature  
  
 **def to\_fahrenheit**(self):  
 **return** (self.temperature \* 1.8) + 32  
  
 **def get\_temperature**(self):  
 **print**("Getting value")  
 **return** self.\_temperature  
  
 **def set\_temperature**(self, temperature):

value = **to\_fahrenheit(temperature)**  
 **if** value < -273:  
 **raise** ValueError("Temperature below -273 is not possible")  
 **else**:  
 self.\_temperature = value  
  
 temperature = property(get\_temperature, set\_temperature)

>>> c = Celsius()

We added a print() function inside get\_temperature() and set\_temperature() to clearly observe that they are being executed.

The last line of the code, makes a property object temperature. Simply put, property attaches some code (get\_temperature and set\_temperature) to the member attribute accesses (temperature).

Any code that retrieves the value of temperature will automatically call get\_temperature()instead of a dictionary (\_\_dict\_\_) look-up. Similarly, any code that assigns a value to temperature will automatically call set\_temperature(). This is one cool feature in Python.

## Property function syntax

property(fget=None, fset=None, fdel=None, doc=None)

fget – is the function to get the attributes

fset – is the function to set the attributes

fdel – is the function to delete the attributes

doc – is a string like comment.

### Other way to write the above program:

**class** Celsius:  
 **def** \_\_init\_\_(self, temperature = 0):  
 self.\_temperature = temperature  
  
 **def to\_fahrenheit**(self):  
 **return** (self.temperature \* 1.8) + 32  
  
 @property  
 **def temperature**(self):  
 **print**("Getting value")  
 **return** self.\_temperature  
  
 @temperature.setter  
 **def temperature**(self, value):  
 **if** value < -273:  
 **raise** ValueError("Temperature below -273 is not possible")  
 **else**:  
 self.\_temperature = value

# Chapter 10

## OOP (Object oriented programming)

There is way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the *object-oriented* programming paradigm.

Major principles of object-oriented programming system are given below

* Object
* Class
* Method
* Inheritance
* Polymorphism
* Data Abstraction
* Encapsulation

### Object

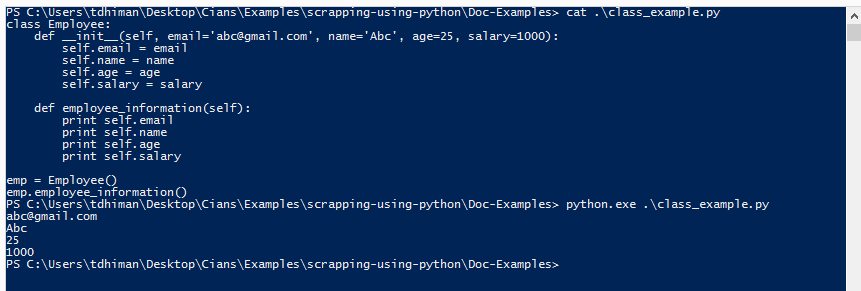
Object is an entity that has state and behavior. It may be anything. It may be physical and logical. For example: mouse, keyboard, chair, table, pen etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute \_\_doc\_\_, which returns the doc string defined in the function source code.

### Class

Class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class then it should contain an attribute and method i.e. an email id, name, age, salary etc.

#### Syntax



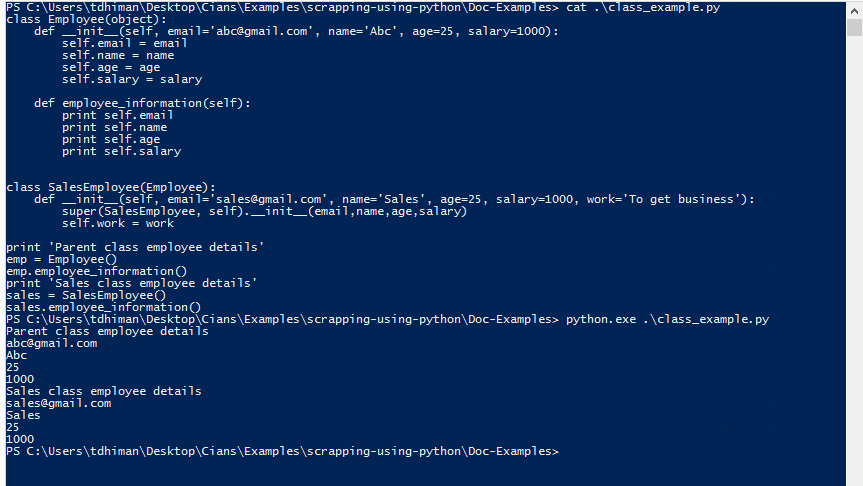
### Method

Method is a function that is associated with an object. In Python, method is not unique to class instances. Any object type can have methods.

### Inheritance

**class Employee**(object):  
 **def** \_\_init\_\_(self, email='abc@gmail.com', name='Abc', age=25, salary=1000):  
 self.email = email  
 self.name = name  
 self.age = age  
 self.salary = salary  
  
 **def employee\_information**(self):  
 **print** self.email  
 **print** self.name  
 **print** self.age  
 **print** self.salary  
  
  
**class** SalesEmployee(Employee):  
 **def** \_\_init\_\_(self, email='sales@gmail.com', name='Sales', age=25, salary=1000, work='To get business'):  
 super(SalesEmployee, self).\_\_init\_\_(email,name,age,salary)  
 self.work = work  
  
**print** 'Parent class employee details'  
emp = Employee()  
emp.employee\_information()  
**print** 'Sales class employee details'  
sales = SalesEmployee()  
sales.employee\_information()

Inheritance is a feature of object-oriented programming. It specifies that one object acquires all the properties and behaviors of parent object. By using inheritance, you can define a new class with a little or no changes to the existing class. The new class is known as derived class or child class and from which it inherits the properties is called base class or parent class.

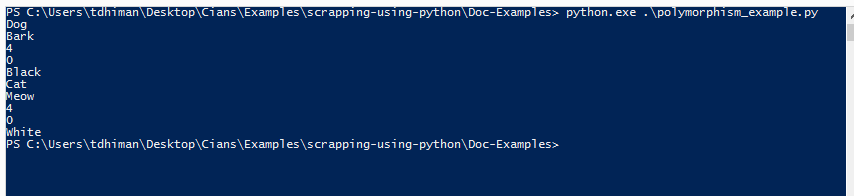


In the above example, you can see the **SalesEmployee** class object inherited the **Employee** class object. **SalesEmployee** class object **emp** also inherited the methods define in its parent class.

### Polymorphism

**class Animal**:  
 **def** \_\_init\_\_(self, animal\_type='Animal', talk='talk', leg=4, hand=0, color='Blue'):  
 self.animal\_type = animal\_type  
 self.talk = talk  
 self.leg = leg  
 self.hand = hand  
 self.color = color  
  
 **def animal\_information**(self):  
 **print** self.animal\_type  
 **print** self.talk  
 **print** self.leg  
 **print** self.hand  
 **print** self.color  
  
  
dog = Animal('Dog', 'Bark', 4, 0, 'Black')  
dog.animal\_information()  
  
cat = Animal('Cat', 'Meow', 4, 0, 'White')  
cat.animal\_information()

Polymorphism is made by two words "poly" and "morphs". Poly means many and Morphs means form, shape. It defines that one task can be performed in different ways. For example: You have a class animal and all animals talk. But they talk differently. Here, the "talk" behavior is polymorphic in the sense and totally depends on the animal. So, the abstract "animal" concept does not actually "talk", but specific animals (like dogs and cats) have a concrete implementation of the action "talk".



### Encapsulation

Encapsulation is also the feature of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

### Data Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things, so that the name captures the core of what a function or a whole program does.

## How to create class and its object

The class statement creates a new class definition.

**class Abc**:  
 *"""  
 Optional class doc string  
 """* methods **and** variable

The class has a documentation string, which can be accessed viaClassName.\_\_doc\_\_.

#### Example

**class Employee**(object):  
 *"""  
 This class demonstrate the class and subclass example  
 """* employee = 0  
   
 **def** \_\_init\_\_(self, email='abc@gmail.com', name='Abc', age=25, salary=1000):  
 self.email = email  
 self.name = name  
 self.age = age  
 self.salary = salary  
  
 **def employee\_information**(self):  
 **print** self.email  
 **print** self.name  
 **print** self.age  
 **print** self.salary  
  
  
**class SalesEmployee**(Employee):  
 **def** \_\_init\_\_(self, email='sales@gmail.com', name='Sales', age=25, salary=1000, work='To get business'):  
 super(SalesEmployee, self).\_\_init\_\_(email,name,age,salary)  
 self.work = work  
  
**print** 'Parent class employee details'  
emp = Employee()  
emp.employee\_information()  
**print** 'Sales class employee details'  
sales = SalesEmployee()  
sales.employee\_information()

* In the above example **employee** is the class variable, its value is shared by all the class objects. This variable accessed through **Employee.employee** (classname.class\_variable).
* The quotation written after the class statement, it is the class doc string. Doc string accessed through **Employee.\_\_doc\_\_** (classname.\_\_doc\_\_). Doc string is used for the documentation or to write about the class working.
* **\_\_init\_\_** method is a special method of the class called constructor. When you will initialize the instance of class called object, then the constructor will execute first from the class.
  + When we created the first object **emp = Employee(),** By this statement you created an object or instance of the class Employee, through this object **emp** reserved the space or memory for the variables (email, name, age, salary) initialize through the constructor.
* **employee\_information()** is the method which is printing the employee information. You can access this only through the object of the class **Employee.** i.e. **emp, sales**

### How to create the object

* **emp** is the class object, you can create the class object by this statement
  + **objectname = classname(argument to send to constructor for initialization)**
* After creating the object, now you can call the class function using this object.
  + objectname.classfunction(argument to send)

### How to access the variable and methods using object

# Create the object of the class  
emp = Employee()  
# To access the variables  
**print** emp.email  
# Or you can change the value of variable using the class instance  
emp.email = 'noname@gmail.com'  
# To access the class methods  
emp.employee\_information()

# Chapter 11

## Regular expression

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.

The module **re** provides full support for Perl-like regular expressions in Python. The **re** module raises the exception re.error if an error occurs while compiling or using a regular expression.

### Regular expression syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a string matches a given regular expression (or if a given regular expression matches a string, which comes down to the same thing).

For more reading about the syntax go to below link

[Regular expression syntax](https://docs.python.org/2/library/re.html%23regular-expression-syntax)

For to practice the regular expressions go to this link [Interactive tutorial](https://regexone.com/lesson/introduction_abcs) before reading.

When you are writing the regular expression, it is recommended that to use raw string instead of regular python string.

Let us take an example if we want to search the **/section** in any text. As we know we need to use backslash before any special character to be treated as the part of normal string. So, our regular expression in this case will be **//section**.

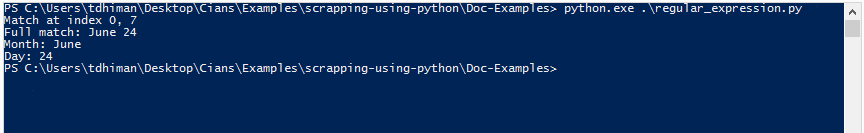
To write this regular expression in python we again need to escape the two backslashes and our final regular expression will be **////section.**

As the above regular expression is difficult to read as there is so many backslashes, in python we can use **r** instead of the two extra backslashes to escape the other backslashes **r”//section”**.

#### Example

**OP**

**import** re  
  
# Lets use a regular expression to match a date string. Ignore  
# the output since we are just testing if the regex matches.  
regex = r"([a-zA-Z]+) (\d+)"  
**if** re.search(regex, "June 24"):  
 # Indeed, the expression "([a-zA-Z]+) (\d+)" matches the date string  
  
 # If we want, we can use the MatchObject's start() and end() methods  
 # to retrieve where the pattern matches in the input string, and the  
 # group() method to get all the matches and captured groups.  
 match = re.search(regex, "June 24")  
  
 # This will print [0, 7), since it matches at the beginning and end of the  
 # string  
 **print** "Match at index %s, %s" % (match.start(), match.end())  
  
 # The groups contain the matched values. In particular:  
 # match.group(0) always returns the fully matched string  
 # match.group(1) match.group(2), ... will return the capture  
 # groups in order from left to right in the input string  
 # match.group() is equivalent to match.group(0)  
  
 # So this will print "June 24"  
 **print** "Full match: %s" % (match.group(0))  
 # So this will print "June"  
 **print** "Month: %s" % (match.group(1))  
 # So this will print "24"  
 **print** "Day: %s" % (match.group(2))  
**else**:  
 # If re.search() does not match, then None is returned  
 **print** "The regex pattern does not match. :("



### Different functions of **re**

#### Match function

##### Syntax

**re.match(pattern, string, flags=0)**

pattern – pattern is your regular expression to be matched in the string

string – will be your string in which you want to match the pattern

flags - You can specify different flags using bitwise OR (|). These are modifiers.

#### Search function

##### Syntax

**re.search(pattern, string, flags=0)**

pattern – pattern is your regular expression to be matched in the string

string – will be your string in which you want to match the pattern

flags - You can specify different flags using bitwise OR (|). These are modifiers.

#### Difference between matches and search

Match function only search the pattern in the beginning of the string while match can search the pattern anywhere in the string.

#### Sub function

This function is used to search and replace the pattern in the string

##### Syntax

**re.sub(pattern, repl, string, max=0)**

pattern – pattern is your regular expression to be matched in the string

repl – This string will replace the matched pattern

string – will be your string in which you want to match the pattern

flags - You can specify different flags using bitwise OR (|). These are modifiers.

# Chapter 12

## Data Structure with python

These below topics we will cover during this chapter

* Linear search
* Binary search
* Bubble sort
* Queue Implementation
* Stack implementation

### Linear search

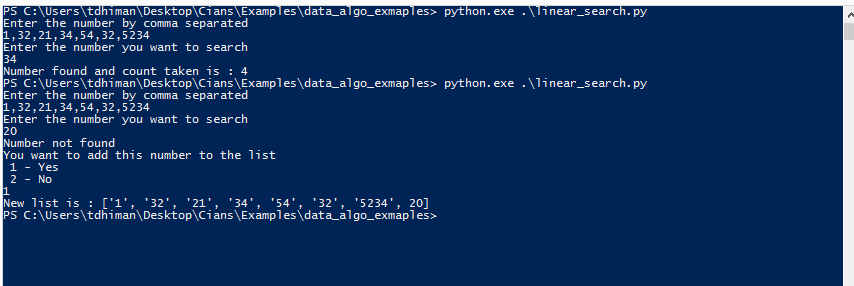
The linear search is used to find an item in a list. The items do not have to be in order. To search for an item, start at the beginning of the list and continue searching until either the end of the list is reached or the item is found.

#### Example or Code

#!/bin/python  
  
position = 0  
  
a = raw\_input('Enter the number by comma separated\n').split(',')  
b = int(raw\_input('Enter the number you want to search\n'))  
  
found = False  
  
**while** position < len(a) **and not** found:  
 **if** int(a[position]) == b:  
 found = True  
 position += 1  
  
**if** found:  
 **print** 'Number found and count taken is : ' + str(position)  
**else**:  
 **print** 'Number not found'  
 dec = raw\_input('You want to add this number to the list\n 1 - Yes\n 2 - No\n')  
 **if** dec == '1':  
 a.append(b)  
 **print** 'New list is : ' + str(a)  
 **else**:  
 **print** 'Number not added to the list : ' + str(a)

* In the above example, **raw\_input** is the function to take input from user. There are two variable **a** will receive the user provided list, **b** will receive the number to be search in the list.
* We initialize a **found** variable to False and loop over the user input list.
* If will iterate to every value from the list until the number not found, then set **found** variable to True.
* Linear search will take less iteration if searched number is present in the beginning of the list, if number present on the last index of the list then linear search will take maximum iteration to search the number.

OP



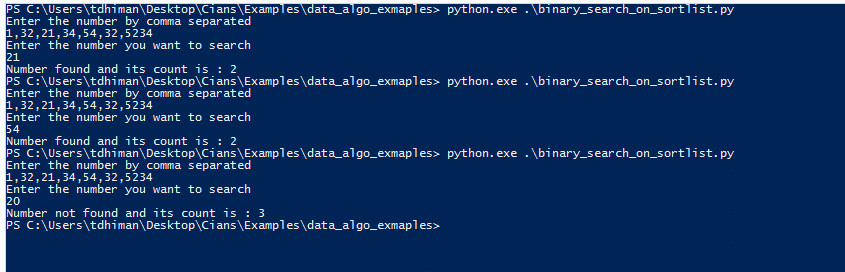
### Binary search

The binary search is used to find an item in an ORDERED list.

To search for an item, look in the middle of the list and see if the number you want is in the middle, above the middle or below the middle. If it is in the middle, you have found the item. If it is higher than the middle value, then adjust the bottom of the list so that you search in a smaller list starting one above the middle of the list. If the number is lower than the middle value, then adjust the top of the list so that you search in a smaller list which has its highest position one less than the middle position.

#!/bin/python  
**import** sys  
  
a = sorted(raw\_input('Enter the number by comma separated\n').split(','), key=**lambda** x: int(x))  
b = int(raw\_input('Enter the number you want to search\n'))  
  
found = False  
first = 0  
last = len(a) - 1  
count = 0  
**while not** found **and** first <= last:  
 count += 1  
 mid = (first + last) / 2  
 **if** int(a[mid]) == b:  
 found = True  
 **else**:  
 **if** first >= last:  
 **print** 'Number not found and its count is : ' + str(count)  
 sys.exit()  
 **else**:  
 **if** int(a[mid]) > b:  
 last = mid - 1  
 **else**:  
 first = mid + 1  
  
**print** 'Number found and its count is : ' + str(count)

**OP**

****

* In the above example, initialize three variables **first,** **last** and **found** initialize to False**. First** will point to the first index of the searching list and **last** will point to the last index of the searching list.
* Program will loop over the user list, and compare the mid element to the search number **b.**
* If the number is matched then while loop will exit else it will check if the searched number is greater or less than the middle number and accordingly change the value of **last** or **first** to create the sub list.

### Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of Ο(n2) where **n** is the number of items.

Example

#!/bin/python  
  
a = raw\_input('Enter the number by comma separated to sort using bubble sort\n').split(',')  
**print** 'Your unsorted list : ' + str(a)  
  
main = 0  
flag = False  
  
**for** iterate **in** range(0, len(a)):  
 main += 1  
 count = 0  
 **for** i **in** a:  
 **if** count < len(a) - 1:  
 **if** int(i) > int(a[count + 1]):  
 temp = i  
 a[count] = a[count + 1]  
 a[count + 1] = temp  
 count += 1  
  
**print** 'Your sorted list : ' + str(a)  
**print** 'Count to sort this list is : ' + str(main) + 'X' + str(count)

### Queue Implementation

A queue is a first in, first out (FIFO) structure. This means that the first item to join the queue is the first to leave the queue. A queue can be implemented using an array (called a list in Python), or using OOP techniques.

A drawing of a cartoon character

Description generated with high confidence

A list implementation for a linear queue will use an append method to add to the queue and a delete method to remove from the queue.

Another type of queue is a circular queue. With this type of queue, if the end of the available spaces is reached, then the next item to be added uses any available spaces at the start of the queue. This is a more efficient use of space.

* In the below example, we define some methods that are useful to implement the queue.
* **isEmpty()** – This method will check whether queue is empty or not.
* **enqueue()** – This method will insert the new element in to the queue.
* **pop() –** This method will pop/remove the value from the queue.
* **peek() –** This method will give the peek value in the queue.
* **show() –** This method will print the all element in the queue.
* **size() –** This method will return the size of the queue.
* **choices() –** This method will provide the user interactive options.

#!/bin/python  
**import** sys  
  
**class Queue**:  
 **def** \_\_init\_\_(self):  
 self.items = []  
  
 **def isEmpty**(self):  
 **return** self.items == []  
  
 **def enqueue**(self, item):  
 self.items.insert(0, item)  
  
 **def pop**(self):  
 **return** self.items.pop()  
  
 **def peek**(self):  
 **try**:  
 **return** self.items[len(self.items) - 1]  
 **except**:  
 **return** None  
  
 **def show**(self):  
 **print** self.items  
  
 **def size**(self):  
 **return** len(self.items)  
  
 **def choices**(self):  
 dec = raw\_input(  
 'Select the choices\n1 - Empty?\n2 - Add\n3 - Pop\n4 - Peek\n5 - size\n6 - show all element\n7 - Exit\n')  
 **try**:  
 **if** int(dec) == 1:  
 **if** self.isEmpty():  
 **print** 'Yes! It is empty'  
 **else**:  
 **print** 'No :) Its not empty'  
 self.choices()  
 **elif** int(dec) == 2:  
 value = raw\_input('Enter the value\n')  
 self.enqueue(value)  
 self.choices()  
 **elif** int(dec) == 3:  
 **print** 'Here is the value : ' + str(self.pop())  
 self.choices()  
 **elif** int(dec) == 4:  
 **print** 'Peek value is : ' + str(self.peek())  
 self.choices()  
 **elif** int(dec) == 5:  
 **print** 'Size of stack is : ' + str(self.size())  
 self.choices()  
 **elif** int(dec) == 6:  
 self.show()  
 self.choices()  
 **elif** int(dec) == 7:  
 **print** 'Exiting application...'  
 sys.exit()  
 **else**:  
 **print** 'Wrong selection due to choice'  
 self.choices()  
 **except** Exception **as** e:  
 **print** 'Wrong selection due to exception'  
 self.choices()  
  
obj = Queue()  
obj.choices()

### Stack implementation

A stack is a last in, first out (LIFO) structure. Items are stored in the stack, but if an item is taken from the stack, it is always the last one that was added.

A screenshot of a cell phone

Description generated with very high confidence

A stack uses a pop method to take the last item added off the stack and a push method to add an item to the stack.

* In the below example, we define some methods that are useful to implement the stack.
* **isEmpty()** – This method will check whether stack is empty or not.
* **push()** – This method will insert the new element in to the stack.
* **pop() –** This method will pop/remove the value from the stack.
* **peek() –** This method will give the peek value in the stack.
* **show() –** This method will print the all element in the stack.
* **size() –** This method will return the size of the stack.
* **choices() –** This method will provide the user interactive options.

#!/bin/python  
**import** sys  
  
**class Stack**:  
 **def** \_\_init\_\_(self):  
 self.items = []  
  
 **def isEmpty**(self):  
 **return** self.items == []  
  
 **def push**(self, item):  
 self.items.append(item)  
  
 **def pop**(self):  
 **return** self.items.pop()  
  
 **def peek**(self):  
 **try**:  
 **return** self.items[len(self.items) - 1]  
 **except**:  
 **return** None  
  
 **def show**(self):  
 **print** self.items  
  
 **def size**(self):  
 **return** len(self.items)  
  
 **def choices**(self):  
 dec = raw\_input(  
 'Select the choices\n1 - Empty?\n2 - Add\n3 - Pop\n4 - Peek\n5 - size\n6 - show all element\n7 - Exit\n')  
 **try**:  
 **if** int(dec) == 1:  
 **if** self.isEmpty():  
 **print** 'Yes! It is empty'  
 **else**:  
 **print** 'No :) Its not empty'  
 self.choices()  
 **elif** int(dec) == 2:  
 value = raw\_input('Enter the value\n')  
 self.push(value)  
 self.choices()  
 **elif** int(dec) == 3:  
 **print** 'Here is the value : ' + str(self.pop())  
 self.choices()  
 **elif** int(dec) == 4:  
 **print** 'Peek value is : ' + str(self.peek())  
 self.choices()  
 **elif** int(dec) == 5:  
 **print** 'Size of stack is : ' + str(self.size())  
 self.choices()  
 **elif** int(dec) == 6:  
 self.show()  
 self.choices()  
 **elif** int(dec) == 7:  
 **print** 'Exiting application...'  
 sys.exit()  
 **else**:  
 **print** 'Wrong selection due to choice'  
 self.choices()  
 **except** Exception **as** e:  
 **print** 'Wrong selection due to exception'  
 self.choices()  
  
obj = Stack()  
obj.choices()

# Chapter 13

## Web Scrapping

Web scraping is data scrapping used for extracting data from websites. In web scrapping we connect to www (world wide web) and extract the data over http (Hyper Text Transfer Protocol) protocol.

There are mainly two steps in web scrapping

* Request to URL (Uniform resource locator).
* Parse the useful data from the html content we received after the web request.

### How to request to URL

There are different packages available in python for querying to URL.

* requests
* urllib2

#### requests

**import** requests  
  
response = requests.get('Your url here')

##### How to check the requests status

**import** requests  
  
response = requests.get('Your url here')  
  
**print** response.status\_code

###### Status code

2xx Success

Request code started from 2xx will indicate success. This class of request code indicates that action requested by the client was received, processed, understood and accepted successfully.

3xx Redirection

This class of status code indicates the client must take additional action to complete the request. These status code are used in URL redirections. When one web page is available for more than one URL.

4xx Client errors

This class of status code is intended for situations in which the client seems to have errored.

400 – The server cannot or will not process this request because of some error.

401 – When you are not authorized to request the URL.

##### How to store the HTML content

There are two ways to store the HTML content

* \_content
* text

Note – Recommended to use the **text** to store the data, \_content is the private variable to the class requests.

**import** requests  
  
# This will send the GET request to the specified URL  
response = requests.get('Your url here')  
  
# response.status\_code will print the request return status  
**print** response.status\_code  
  
# This html\_data will store the html data that is requested from the web page  
html\_data = response.text

#### Parsing of HTML data

To parse the data from the HTML content, already some packages available but also you can write your own parser.

##### Available package to parse HTML

* BeautifulSoup
* HTMLParse
* lxml
* html5lib

###### BeautifulSoup

**import** requests  
**from** bs4 **import** BeautifulSoup  
  
# This will send the GET request to the specified URL  
response = requests.get('Your url here')  
  
# response.status\_code will print the request return status  
**print** response.status\_code  
  
# This html\_data will store the html data that is requested from the web page  
html\_data = response.text  
  
# This will convert the html data into beautifulsoup object  
html\_content = BeautifulSoup(html\_data, "html.parser")  
# If you want to parse the whole body tag  
# This will find the body tag  
beautiful\_obj = html\_content.find('body')  
# This will convert it into list of data inside the body tag  
iterator = beautiful\_obj.findChildren()

This above code will find all the content in the body tag and convert it into python list.

We can iterate through this python list and find out the text and other attributes present inside the body.

**import** requests  
**from** bs4 **import** BeautifulSoup  
  
# This will send the GET request to the specified URL  
response = requests.get('Your url here')  
  
# response.status\_code will print the request return status  
**print** response.status\_code  
  
# This html\_data will store the html data that is requested from the web page  
html\_data = response.text  
  
# This will convert the html data into beautifulsoup object  
html\_content = BeautifulSoup(html\_data, "html.parser")  
# If you want to parse the whole body tag  
# This will find the body tag  
beautiful\_obj = html\_content.find('body')  
# This will convert it into list of data inside the body tag  
iterator = beautiful\_obj.findChildren()  
  
**for** tag **in** iterator:  
 # This will print the tag  
 **print** tag  
 # This will print the Text field inside the tag  
 **print** tag.text  
 # To find out the attributes and their values inside the tag  
 **for** attribute\_name, attribute\_value **in** tag.iteritems():  
 # This will print the attribute name  
 **print** attribute\_name  
 # This will print the attribute value  
 **print** attribute\_value

Also added the user interactive program for web scrapping.

**import** json  
**import** csv  
**from** Tkinter **import** \*  
**import** requests  
**import** re  
**from** bs4 **import** BeautifulSoup  
# from Naked.toolshed.shell import execute\_js, muterun\_js  
  
\_\_author\_\_ = 'xxxxx'  
  
"""  
This program is basically a scrapper  
It is used for extracting data from websites.  
It is a form of copying,   
in which specific data is gathered and copied from the web,   
typically stored into a excel, csv or json for later retrieval or analysis.  
"""

# This function is to change the set default encoding of python  
reload(sys)  
sys.setdefaultencoding('utf-8')  
  
  
**class Extract**:  
 **def** \_\_init\_\_(self):  
 *"""  
 This is the initialization of user choices  
 and data list and dictionary  
 """* self.user\_choice = {'url': raw\_input('Enter the url to extract the data\n'), 'tag': {}}  
 self.tag = raw\_input('Please select the tag you want to search in\n'  
 '1 - body\n'  
 '2 - div\n'  
 '3 - a\n'  
 '4 - script\n'  
 '5 - style\n'  
 '6 - form\n'  
 '7 - span\n')  
 **if** int(self.tag) == 1:  
 self.user\_choice['tag']['body'] = {}  
 **pass  
 elif** int(self.tag) == 2:  
 self.user\_choice['tag']['div'] = {}  
 self.attribute = raw\_input('Please select the attribute\n'  
 '1 - class\n'  
 '2 - id\n'  
 '3 - style\n'  
 '4 - No attribute\n')  
 **if** int(self.attribute) == 1:  
 self.user\_choice['tag']['div']['attribute\_value'] = raw\_input('Enter the value of attribute : class\n')  
 self.user\_choice['tag']['div']['attribute\_name'] = 'class'  
 **elif** int(self.attribute) == 2:  
 self.user\_choice['tag']['div']['attribute\_value'] = raw\_input('Enter the value of attribute : id\n')  
 self.user\_choice['tag']['div']['attribute\_name'] = 'id'  
 **elif** int(self.attribute) == 3:  
 self.user\_choice['tag']['div']['attribute\_value'] = raw\_input('Enter the value of attribute : style\n')  
 self.user\_choice['tag']['div']['attribute\_name'] = 'style'  
 **elif** int(self.attribute) == 4:  
 self.user\_choice['tag']['div']['attribute\_value'] = 0  
 self.user\_choice['tag']['div']['attribute\_name'] = 0

**elif** int(self.tag) == 3:  
 self.user\_choice['tag']['a'] = {}  
 self.attribute = raw\_input('Please select the attribute\n'  
 '1 - class\n'  
 '2 - id\n'  
 '3 - No attribute\n')  
 **if** int(self.attribute) == 1:  
 self.user\_choice['tag']['a']['attribute\_value'] = raw\_input('Enter the value of attribute : class\n')  
 self.user\_choice['tag']['a']['attribute\_name'] = 'class'  
 **elif** int(self.attribute) == 2:  
 self.user\_choice['tag']['a']['attribute\_value'] = raw\_input('Enter the value of attribute : id\n')  
 self.user\_choice['tag']['a']['attribute\_name'] = 'id'  
 **elif** int(self.attribute) == 3:  
 self.user\_choice['tag']['a']['attribute\_value'] = 0  
 self.user\_choice['tag']['a']['attribute\_name'] = 0  
**elif** int(self.tag) == 4:  
 self.user\_choice['tag']['script'] = {}  
 **pass  
elif** int(self.tag) == 5:  
 self.user\_choice['tag']['style'] = {}  
 **pass  
elif** int(self.tag) == 6:  
 self.user\_choice['tag']['form'] = {}  
 self.attribute = raw\_input('Please select the attribute\n'  
 '1 - class\n'  
 '2 - id\n'  
 '3 - No attribute\n')  
 **if** int(self.attribute) == 1:  
 self.user\_choice['tag']['form']['attribute\_value'] = raw\_input('Enter the value of attribute : class\n')  
 self.user\_choice['tag']['form']['attribute\_name'] = 'class'  
 **elif** int(self.attribute) == 2:  
 self.user\_choice['tag']['form']['attribute\_value'] = raw\_input('Enter the value of attribute : id\n')  
 self.user\_choice['tag']['form']['attribute\_name'] = 'id'  
 **elif** int(self.attribute) == 3:  
 self.user\_choice['tag']['form']['attribute\_value'] = 0  
 self.user\_choice['tag']['form']['attribute\_name'] = 0  
**elif** int(self.tag) == 7:  
 self.user\_choice['tag']['span'] = {}  
 self.attribute = raw\_input('Please select the attribute\n'  
 '1 - class\n'  
 '2 - id\n'  
 '3 - No attribute\n')  
 **if** int(self.attribute) == 1:  
 self.user\_choice['tag']['span']['attribute\_value'] = raw\_input('Enter the value of attribute : class\n')  
 self.user\_choice['tag']['span']['attribute\_name'] = 'class'  
 **elif** int(self.attribute) == 2:  
 self.user\_choice['tag']['span']['attribute\_value'] = raw\_input('Enter the value of attribute : id\n')  
 self.user\_choice['tag']['span']['attribute\_name'] = 'id'  
 **elif** int(self.attribute) == 3:  
 self.user\_choice['tag']['span']['attribute\_value'] = 0  
 self.user\_choice['tag']['span']['attribute\_name'] = 0  
**else**:  
 **print** 'Your choice not matched, Exiting program ... '  
# self.url = 'http://www.mcdonaldsindia.net/' + str(self.state) + '-store-locator.aspx'  
  
self.data\_dict = {}  
self.final\_list = []

**def write\_csv**(self, data):  
 *"""  
 This function will convert the dictionary/json data in to csv  
 By default, the classes in the csv module use Windows-style line terminators (\r\n) rather than Unix-style (\n)  
 So because of this we used line terminator='\n'  
 """* **with** open('output.csv', 'w') **as** out:  
 writer = csv.DictWriter(out, data[0].keys(), delimiter=',', lineterminator='\n')  
 writer.writeheader()  
 **for** row **in** data:  
 writer.writerow(row)  
  
**def request\_to\_web**(self):  
 *"""* ***:return****: a beautiful soup object from html pages  
  
 This function will request to url using requests module  
 Then parse the html content using BeautifulSoup  
 It will extract the content on the basis of tag and class provided by the user  
 It will return beautiful soup object  
 """* **if** self.user\_choice:  
 page = requests.get(self.user\_choice['url'])  
 html\_content = page.text  
 parsed\_html = BeautifulSoup(html\_content, "html.parser")  
 **print** self.user\_choice['tag'].keys()[0]  
 **if** self.user\_choice['tag'].keys()[0] == 'body':  
 beautiful\_obj = parsed\_html.find(self.user\_choice['tag'].keys()[0])  
 iterator = beautiful\_obj.findChildren()  
 **elif** self.user\_choice['tag'][self.user\_choice['tag'].keys()[0]] == {} **or** \  
 (self.user\_choice['tag'][self.user\_choice['tag'].keys()[0]]['attribute\_value'] == 0 **and** self.user\_choice['tag'][self.user\_choice['tag'].keys()[0]]['attribute\_value'] == 0):  
 beautiful\_obj = parsed\_html.body.find\_all(self.user\_choice['tag'].keys()[0])  
 iterator = beautiful\_obj  
 **else**:  
 beautiful\_obj = parsed\_html.body.find\_all(self.user\_choice['tag'].keys()[0],  
 attrs={  
 self.user\_choice['tag']  
 [self.user\_choice['tag'].keys()[0]]  
 ['attribute\_name']  
 : self.user\_choice['tag']  
 [self.user\_choice['tag'].keys()[0]]  
 ['attribute\_value']})  
 iterator = beautiful\_obj  
 **else**:  
 **raise** Exception('Url missing !!')  
  
 # printing the parse data, To show user whether data is correct or not?  
  
 # print 'Here is below your parse content'  
 # print '------------------------------------------------------------------------------------------------------'  
  
 data = '-------------------------------------------------------------------------------------------------------'

html\_dictionary = {}  
information\_count = 1  
**for** tag **in** iterator:  
 # print tag  
 html\_dictionary[information\_count] = {}  
 # print 'Text : '+str(tag.text.strip())  
 html\_dictionary[information\_count]['Text'] = str(tag.text.strip())  
 data += 'Text : ' + str(tag.text.strip()) + '\n'  
 **for** key, value **in** tag.attrs.iteritems():  
 # print 'Key : '+str(key).strip()  
 html\_dictionary[information\_count][key] = value  
 data += 'Key : ' + str(key).strip() + '\n'  
 # print 'Value : '+str(value).strip()  
 data += 'Value : ' + str(value).strip() + '\n'  
  
 data += '--------------------------------------------------------------------------------------------------'  
 # print '--------------------------------------------------------------------------------------------------'  
 information\_count += 1  
  
**for** key **in** html\_dictionary.iterkeys():  
 **print** key  
 **for** key1 **in** html\_dictionary[key].iterkeys():  
 **print** str(key1)+':'+str(html\_dictionary[key][key1])  
  
user\_input = raw\_input('If your content not good then restart the search\n1 - Y\n2 - N\n')  
**if** int(user\_input) == 1:  
 **print** 'Run this application again'  
 **print** 'Exiting the application'  
 sys.exit()  
**elif** int(user\_input) == 2:  
 **pass  
else**:  
 **print** 'You selected a wrong option'  
 **print** 'Exiting the application'  
 sys.exit()  
  
**while** True:  
 operator = raw\_input(  
 'Are you want to apply operator\n'  
 '1 - between\n'  
 '2 - split\n'  
 '3 - Text\n'  
 '4 - link\n'  
 '5 - class\n'  
 '6 - id\n'  
 '7 - script\n'  
 '8 - want to proceed for final parsing?\n'  
 )  
 **if** int(operator) == 1:  
 extracted = self.apply\_operator(data, 'between')  
 self.populate\_data\_to\_tkinter\_text\_field(extracted)  
 **elif** int(operator) == 2:  
 **try**:  
 delimiter = raw\_input('Write your separator and index by comma separated\n').split(',')  
 **except** Exception **as** e:  
 **print** 'Your input is wrong'  
 sys.exit()  
 extracted = extracted.split(delimiter[0])[int(delimiter[1])]  
 self.populate\_data\_to\_tkinter\_text\_field(extracted)

**elif** int(operator) == 3:  
 extracted = self.apply\_operator(html\_dictionary, 'Text')  
 self.populate\_data\_to\_tkinter\_text\_field(extracted)  
 **elif** int(operator) == 4:  
 extracted = self.apply\_operator(html\_dictionary, 'link')  
 self.populate\_data\_to\_tkinter\_text\_field(extracted)  
 **elif** int(operator) == 5:  
 extracted = self.apply\_operator(html\_dictionary, 'class')  
 self.populate\_data\_to\_tkinter\_text\_field(extracted)  
 **elif** int(operator) == 6:  
 extracted = self.apply\_operator(html\_dictionary, 'id')  
 self.populate\_data\_to\_tkinter\_text\_field(extracted)  
 **elif** int(operator) == 7:  
 extracted = self.apply\_operator(html\_dictionary, 'script')  
 self.populate\_data\_to\_tkinter\_text\_field(extracted)  
 **elif** int(operator) == 8:  
 user = raw\_input('How you want to parse the data\n1 - json\n2 - Other\n')  
 **if** int(user) == 1:  
 data = self.json\_parsing(extracted)  
 **print** data  
 **try**:  
 user\_dict\_dec = raw\_input(  
 'You want to write full\n'  
 '1 - Full dictionary or json\n2 - or specific key, & enter the key\n'  
 ).split(',')  
 **except** Exception **as** e:  
 **print** 'Your input was not correct ' + str(e)  
 sys.exit()  
 **if** data **and** int(user\_dict\_dec[0]) == 1:  
 **try**:  
 self.write\_csv(data)  
 **print** 'Your csv file is generated.'  
 **except**:  
 **pass** sys.exit()  
 **elif** data **and** int(user\_dict\_dec[0]) == 2:  
 **try**:  
 self.write\_csv(data[user\_dict\_dec[1]])  
 **print** 'Your csv file is generated'  
 **except**:  
 **pass** sys.exit()  
 **else**:  
 **print** type(extracted), extracted  
 self.write\_csv(json.loads(extracted))  
 sys.exit()  
 **else**:  
 **pass  
  
 return** beautiful\_obj  
  
@staticmethod  
**def json\_parsing**(data):  
 **try**:  
 **return** json.loads(data)  
 **except**:  
 **return** None

**def apply\_operator**(self, data, operator):  
 **if** operator == 'between':  
 first\_value = raw\_input('Enter the first value\n')  
 last\_value = raw\_input('Enter the last value\n')  
 search\_pattern = first\_value + '(.\*?)' + last\_value  
 main = re.findall(search\_pattern, data, re.DOTALL)[0]  
 **return** main  
 **elif** operator == 'Text':  
 text\_list = []  
 **for** key **in** data.iterkeys():  
 local\_dict = {}  
 **try**:  
 local\_dict['Name'] = str(data[key][operator].split('\r\n')[0])  
 local\_dict['Address'] = str(data[key][operator].split('\r\n')[1].lstrip().replace('\xc2\xa0', ' '))  
 **except** Exception **as** e:  
 local\_dict = {'Name': str(data[key][operator]), 'Address': str(e)}  
 **print** local\_dict  
 text\_list.append(local\_dict)  
 **print** text\_list  
 self.populate\_data\_to\_tkinter\_text\_field(text\_list)  
 **return** json.dumps(text\_list)  
 **elif** operator == 'link':  
 text\_list = []  
 **for** key, value **in** data.iteritems():  
 local\_dict = {}  
 local\_dict = {'Link': value['href'], 'Text': value['Text']}  
 text\_list.append(local\_dict)  
  
 **print** text\_list  
 **return** json.dumps(text\_list)  
 **elif** operator == 'class':  
 text\_list = []  
 **for** key, value **in** data.iteritems():  
 local\_dict = {}  
 **if** operator **in** value:  
 local\_dict = {'class': value['class'], 'Text': value['Text']}  
 text\_list.append(local\_dict)  
 **else**:  
 **pass  
  
 print** text\_list  
 **return** json.dumps(text\_list)  
 **elif** operator == 'id':  
 text\_list = []  
 **for** key, value **in** data.iteritems():  
 local\_dict = {}  
 **if** operator **in** value:  
 local\_dict = {'id': value['class'], 'Text': value['Text']}  
 text\_list.append(local\_dict)  
 **else**:  
 **pass  
  
 print** text\_list  
 **return** json.dumps(text\_list)

**elif** operator == 'script':  
 """  
 for key, value in data.iteritems():  
 tmp = open('file.js', 'w')  
 tmp.write(value['Text'])  
 tmp.close()  
 response = muterun\_js('file.js')  
 if response.exitcode == 0:  
 print(response.stdout)  
 else:  
 sys.stderr.write(response.stderr)  
 """  
 **pass** @staticmethod  
 **def populate\_data\_to\_tkinter\_text\_field**(value):  
 root = Tk()  
 scroll = Scrollbar(root)  
 text = Text(root, height=20, width=200)  
 scroll.pack(side=RIGHT, fill=Y)  
 text.pack(side=LEFT, fill=Y)  
 scroll.config(command=text.yview)  
 text.config(yscrollcommand=scroll.set)  
 text.insert(END, value)  
 mainloop()  
  
 **def converting\_to\_dict**(self, parsed\_data):  
 *"""* ***:return****: dictionary or json from the beautiful soup object  
 This function will take the parsed content from the html  
 and convert into the json or dictionary  
 """* **for** store **in** parsed\_data:  
 single = store.find\_all("p")  
 **for** value **in** single:  
 local\_dict = {}  
 val = value.text.split('\r\n')  
 local\_dict['Name'] = val[0].strip()  
 local\_dict['Address'] = val[1].strip()  
 self.final\_list.append(local\_dict)  
  
 self.data\_dict['locations'] = self.final\_list  
 **return** self.data\_dict  
  
  
**if** \_\_name\_\_ == "\_\_main\_\_":  
 ext = Extract()  
 parse = ext.request\_to\_web()  
 data\_dict = ext.converting\_to\_dict(parse)  
 ext.write\_csv(data\_dict['locations'])

## Scrapping using POST request (Dynamic pages):

### Scrap the historical data from **Nasdaq Stock Market:** [Link](http://www.nasdaq.com/symbol/ddd/historical)

#### Problem: Extract the stock data of last 3 years

A screenshot of a social media post

Description generated with very high confidence

##### Steps for web scrapping

* Right click and open the inspect (developer page)

A screenshot of a social media post

Description generated with very high confidence

* Go to Network section and click on **XHR** 
  + Here you can see your http request format and the response.
* Now select the **3 years** from the dropdown section and you will see the ajax call on developer interface as **historical**.
* Now click on latest historical call and check the Headers, Response sections.

A screenshot of a social media post

Description generated with very high confidence

A screenshot of a computer

Description generated with very high confidence

* In the Headers section, you need to see the highlighted section in the given figures
  + Request URL
    - This is the URL to send the request
  + Request Method
    - Type of http request
  + Content-type
    - Type of headers data
  + Request Payload
  + Request data
* Now check the Response tab and you will find out the data is inserted inside the HTML table under the tag **<tbody>**.

A screenshot of a social media post

Description generated with very high confidence

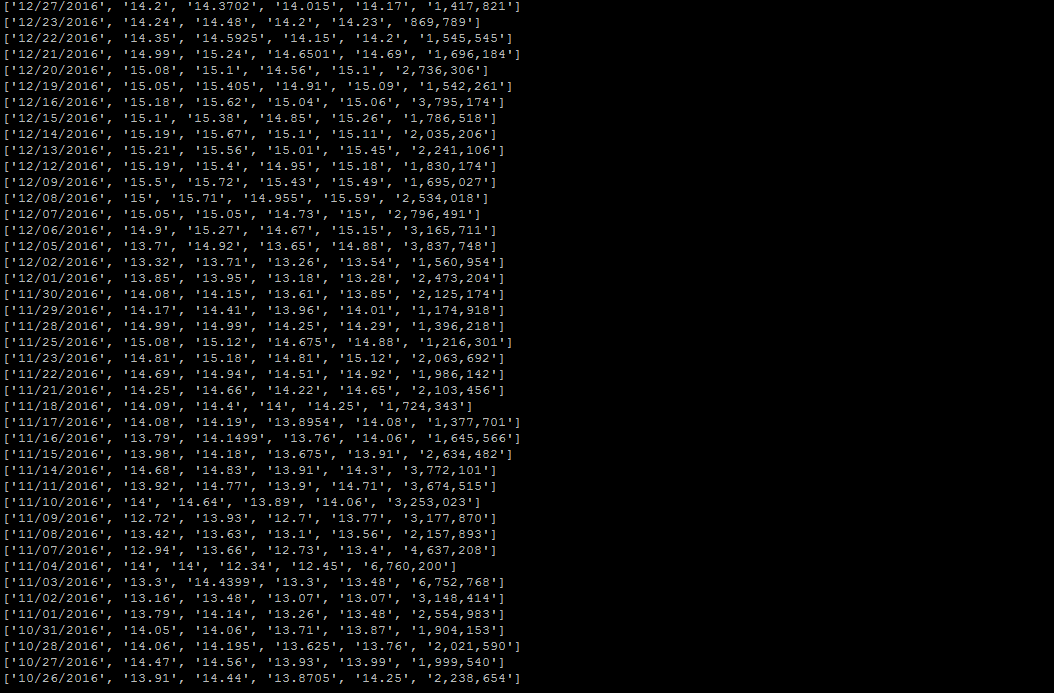
* Now we have all required information to extract data
  + Request URL
  + Request Type
  + Request Header
  + Data type
  + Response data format
* Now as we know the Request Type is POST. We need to send the post request to the Request URL. But before that we need to create the POST request header and data. Let’s start the coding

A screenshot of a cell phone

Description generated with very high confidence

OP

* Your output will print in below format



# Chapter 14

## Lambda function

In Python, anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

## Syntax

*lambda arguments: expression*

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

### Example

double = **lambda** x: x\*2  
  
**print** double(10)

**OP**

20

Above example will double the value given to the lambda function. Here **x** is the argument and **x\*2** is the expression. Function has no name but it will return the **x\*2**.

### Lambda with Other Functions

#### filter()

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True

##### Syntax

filter(function as argument, list)

filter function takes two arguments, first argument is the function type and second argument is the list.

###### Example

filter\_example = filter(**lambda** x: (x % 2 == 0), [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])  
  
**print** filter\_example

OP

[2, 4, 6, 8, 20]

The above example will evaluate every element of the [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] list through the lambda function defines and return the list of elements, which are perfectly dividend of 2.

#### Map()

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item

##### Syntax

map(function as argument, list)

The map() function in Python takes in a function and a list.

##### Example

map\_example = map(**lambda** x: (x % 2 == 0), [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])  
  
**print** map\_example

**OP**

[False, True, False, True, False, True, False, True, False, True]

# Basic SQL Operators and Syntax

## SELECT

The most basic SELECT statement has only 2 parts: (1) what columns you want to return and (2) what table(s) those columns come from.

If we want to retrieve all the information about all the customers in the Employees table, we could use the asterisk (\*) as a shortcut for all the columns, and our query looks like

**SELECT \* FROM Employees**

If you want some specific columns

**SELECT ID, Name, Gender, City FROM Employees**

## WHERE

The next thing we want to do is to start limiting, or filtering, the data we fetch from the database. By adding a **WHERE**clause to the **SELECT** statement, we add one (or more) conditions that must be met by the selected data. This will limit the number of rows that answer the query and are fetched. In many cases, this is where most of the "action" of a query takes place.

We can continue with our previous query, and limit it to only those employees living in London:

**SELECT ID, Name, Gender, City FROM Employees**

**WHERE City = 'London'**

If you want the employee not living in London

**SELECT ID, Name, Gender, City FROM Employees**

**WHERE City <> 'London'**

If you want to use the **AND** & **OR** operator with where clause.

**SELECT ID, Name, Gender, City, HireDate FROM Employees**

**WHERE (HireDate >= ‘1-Jan-2000’) AND (HireDate <= ‘1-Jan-2010’)**

you can use **BETWEEN** operatorfor above statement

**SELECT ID, Name, Gender, City, HireDate FROM Employees**

**WHERE HireDate BETWEEN ‘1-Jan-2000’ AND ‘1-Jan-2010’**

**OR** operator use

**SELECT ID, Name, Gender, City, HireDate FROM Employees**

**WHERE City=’London’ OR City=’Delhi’**

You can use **IN** operatorfor above statement

**SELECT ID, Name, Gender, City, HireDate FROM Employees**

**WHERE City IN (’London’, ’Delhi’)**

You can use **NOT** operator with IN operator

**SELECT ID, Name, Gender, City, HireDate FROM Employees**

**WHERE City NOT IN (’London’, ’Delhi’)**

The above statement will give the output of all employee details which are living except London and Delhi.

## LIKE

**LIKE**operator allows us to perform basic pattern-matching using wildcard characters. For Microsoft SQL Server, the wildcard characters are defined as follows:

* **WHERE** FirstName **LIKE** '\_im' finds all three-letter first names that end with 'im' (e.g. Jim, Tim).
* **WHERE** LastName **LIKE** '%stein' finds all employees, whose last name ends with 'stein'
* **WHERE** LastName **LIKE** '%stein%' finds all employees, whose last name includes 'stein' anywhere in the name.
* **WHERE** FirstName **LIKE** '[JT]im' finds three-letter first names that end with 'im' and begin with either 'J' or 'T' (that is, *only* Jim and Tim)
* **WHERE** LastName **LIKE** 'm[^c]%' finds all last names beginning with 'm' where the following (second) letter is not 'c'.

**SELECT ID, Name, LastName, HireDate, City FROM Employees**

**WHERE (Name NOT LIKE 'M%') AND (Name NOT LIKE 'A%')**

The above query will print details of employees excluded the employee name start with A and M.

## ORDER

To sort the data rows, we use ORDER BY clause.

**SELECT ID, Name, HireDate, City FROM Employees**

**ORDER BY City**

The above statement will print the employee details in ascending order of their city column.

If you want the data descending order you can write your query like

**SELECT ID, Name, HireDate, City FROM Employees**

**ORDER BY City DESC**