Python Hand Book

Version – April 2017

Contents

Introduction with Python 3
Why Python? 3
Features of Python 3
Python Productivity 3
OOPS in Python 3
Core python concepts 4
Conditional Statements 4
Looping 5
Pass Statement: 8
String Manipulation: 8
String functions 8
Slicing 11
Lists 12
List Operation: 12
Functions 13
Using Lists as Stacks 14
Using Lists as Queues 15
Tuple 15
Dictionaries 17
Anonymous functions 22
Modules 23
Exception 29
Advanced Python 31
OOPs concept 31
Regular Expression 33
- Modifiers 34

Patterns 35
Database 36
Networking 38
CGI 42
Architecture 43
Get And Post Methods 44
Cookies 44
File Upload 45
MultiThreading 46
Designing 51
Django 51
What is Django? 51
How It Works? 51
Django Installation 52
Django Admin 55
GUI Programming 55
Tkinter Programming 55

Introduction with Python

- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.
- Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development.
- Python supports modules and packages, which encourages program modularity and code reuse.
- The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Why Python?

- **Designed to be easy to learn and master**
    - Clean, clear syntax
    - Very few keywords
- **Highly portable**
    - Runs almost anywhere - high end servers and workstations, down to windows CE
    - Uses machine independent byte-code
- **Extensible**
    - Designed to be extensible using C/C++,
    - allowing access to many external libraries

Features of Python

- **Clean syntax plus high-level data types**
    - Leads to fast coding (First language in many universities abroad!)
- **Uses white-space to delimit blocks**
    - Humans generally do, so why not the language?
    - Try it, you will end up liking it
- **Variables do not need declaration**
    - Although not a type-less language

Python Productivity

- **Reduced development time**
    - code is 2-10x shorter than C, C++, Java
- **Improved program maintenance**
    - code is extremely readable
- **Less training**
    - language is very easy to learn

OOPS in Python

- Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy.
- Overview of OOP Terminology
- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.

- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation:** The creation of an instance of a class.
- **Method:** A special kind of function that is defined in a class definition.
- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading:** The assignment of more than one function to a particular operator.

Programming Style

- Python programs/modules are written as text files with traditionally a .py extension.
- Each Python module has its own discrete namespace.
- Name space within a Python module is a global one.
- Python modules and programs are differentiated only by the way they are called.
- .py files executed directly are programs (often referred to as scripts)
- .py files referenced via the import statement are modules.
- Thus, the same .py file can be a program/script, or a module.

Core python concepts

Conditional Statements

  - If
  - If- else
  - Nested if-else
- Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
- Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome.
- **If statement**
  - It is similar to that of other languages.
  - The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.
- Syntax: if expression:
  Statement
- **If Example:**

```
a=10
if a>0:
        print("A Is Positive Number")
```

If…else Statement

- It is frequently the case that you want one thing to happen when a condition it true, and something else to happen when it is false.
- For that we have the if else statement.
- **Syntax :** if expression:
  Statement
  else:
  Statement
- **If….else Example**

```
a=10
if a>0:
        print(“A Is Positive Number”)
else:
        print(“A Is Negative Number”)
```

Nested if...else Statement

- There may be a situation when you want to check for another condition after a condition resolves to true.
- In such a situation, you can use the nested if construct
- **Syntax :**
  if expression1:
  Statement
  if expression2:
  Statement
  else:
  Statement
- **Nested if example:**

```
a=10
b=20
c=30
if a>b:
        if a>c:
                print(“A Is Greater”)
        else:
                print(“C Is Greater”)
else:
        if b>c:
                print(“B Is Greater”)
        else:
                print(“C Is Greater”)
```

Looping

- A loop statement allows us to execute a statement or group of statements multiple times.
- Python programming language provides following types of loops to handle looping requirements.
  - While Loop

- For Loop

For loop has the ability to iterate over the items of any sequence, such as a list or a string.

- **Syntax :** for iterating_var in sequence:
  Statements(s)
- If a sequence contains an expression list, it is evaluated first.
  Then, the first item in the sequence is assigned to the iterating variable *iterating_var*.
- Next, the statements block is executed.
- Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted
- **For Example**

```
i=1
for i in range(1, 10):
    if i<=5:
        print(i, "Smaller Than Equal Than 5")
    else:
        print(i, "Larger Than 5")
```

While

- A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
- Syntax : while expression:
  Statement
- Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

```
number=23
running=True

while running:
    guess=int (input ("Enter An Integer : "))
    if guess==number:
        print ("Success")
        running=False
    elif guess<number:
        print ("Lower Than Original")
    elif guess>number:
        print ("Higher Than Original")
    else:
        print ("Sorry")
else:
    print("The While Loop Is Over")
print ("Done")
```

Nested loops

- Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.
- Syntax :For
  for iterating_var in sequence:
  for iterating_var in sequence:

Statements(s)
statements(s)

Note: The range () Function

- If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy.
- It generates Arithmetic progressions.

```
for i in range (1, 7):
      for j in range (i):
            print(i, end=' ')
      print ()
```

- Syntax : while
  while expression:
  while expression:
  Statement
  Statement

Nested While Example:

Control Statements

- Loop control statements change execution from its normal sequence.
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements.
  - Break
  - Continue
  - P

```
i=0;j=0
while i<7:
      j=0
      while j<i:
            print(i, end=' ')
            j=j+1
      i=i+1
      print()
```

Break statement:

- It brings control out of the loop and transfers execution to the statement immediately following the loop.

```
for l in "Tops Technology":
    if l=='p' or l=='s':
        break
print ("Current Letter: ", l)
```

Continue Statement :

- It continues with the next iteration of the loop

```
for l in "Tops Technology":
    if l=='p' or l=='s':
        continue
print ("Current Letter: ", l)
```

Pass Statement:

- The pass statement does nothing.
- It can be used when a statement is required syntactically but the program requires no action.
- For example:

```
for l in "Tops Technology":
      pass
print("Current Letter: ", l)
```

String Manipulation:

- Textual data in Python is handled with ”str” objects, or *strings*. Strings are immutable(fixed/rigid) sequences of Unicode code points.
- String literals are written in a variety of ways:
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes".
  - Triple quoted: '''Three single quotes''', """Three double quotes"""
- Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String functions

- **str.capitalize()**
  - Return a copy of the string with its first character capitalized and the rest lowercased.
- **str.casefold()**
  - Return a case folded copy of the string. Case folded strings may be used for caseless matching.
- **str.center(*width*[, *fillchar*])**
  - Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if*width* is less than or equal to len(s).
- **str.count(*sub*[, *start*[, *end*]])**
  - Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*].
  - Optional arguments *start* and *end* are interpreted as in slice notation.
- **str.endswith(*suffix*[, *start*[, *end*]])**

- Return True if the string ends with the specified *suffix*, otherwise return False.
- *Suffix* can also be a tuple of suffixes to look for.
- With optional *start*, test beginning at that position.
- With optional *end*, stop comparing at that position.

- **str.find(*sub*[, *start*[, *end*]])**
  - Return the lowest index in the string where substring *sub* is found within the slice s [start: end].
  - Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 if *sub* is not found.

Note:

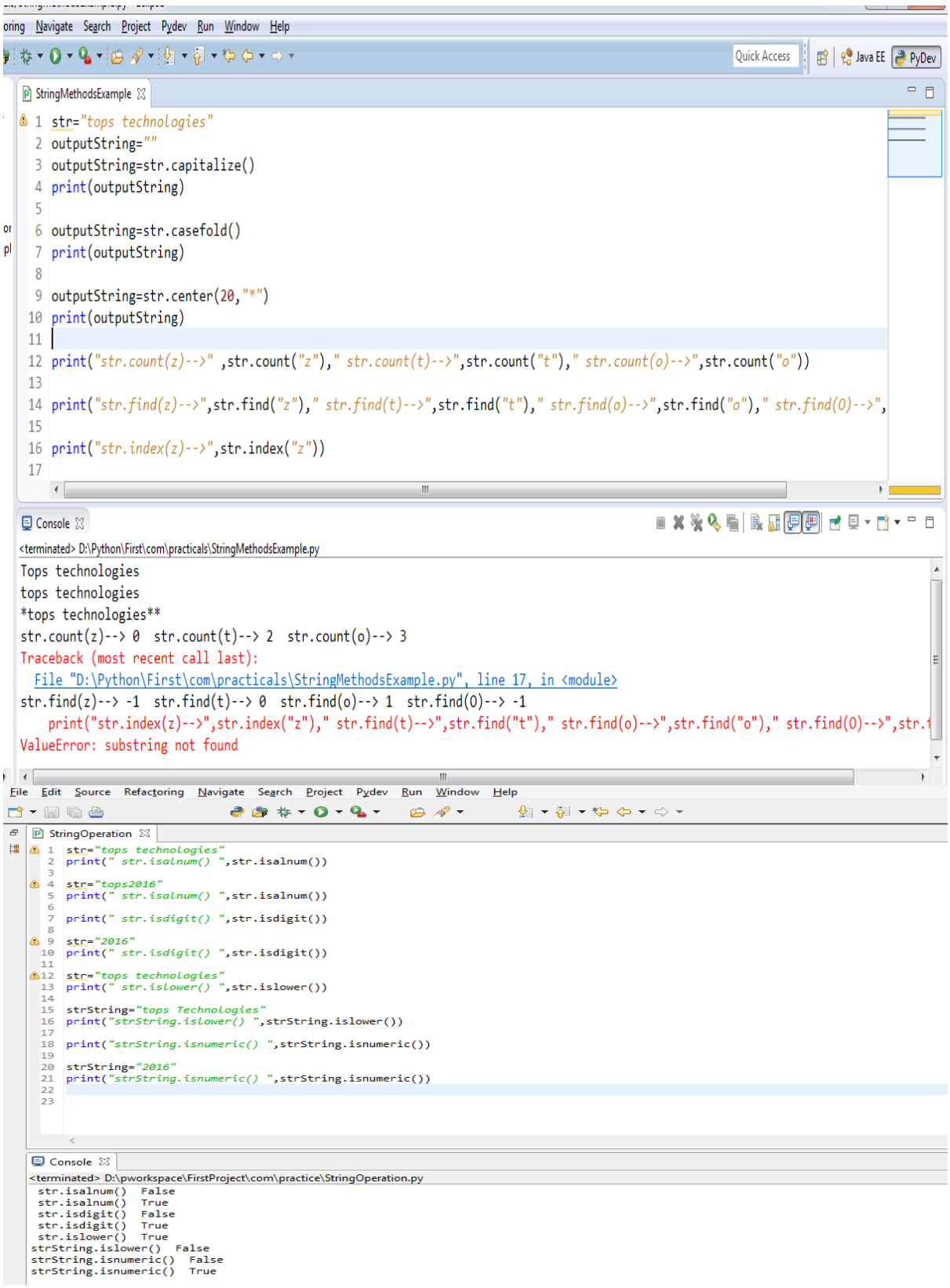
- The find () method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the ”in”operator:
- >>>**>>>** 'Py' **in** 'Python' True
- **str.format(**args*, ***kwargs*)**
  - Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces{}.
  - Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.
- str.index (*sub* [, *start* [, *end*]]) Like find (), but raise Value Error when the substring is not found.
- **str.isalnum()**
  - Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.
  - A character c is alphanumeric if one of the following returns True:
  - **c.isalpha(), c.isdecimal(), c.isdigit(), or c.isnumeric().**
- **str.isidentifier()**
  - Return true if the string is a valid identifier according to the language definition
- **str.islower()**
  - Return true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.
- **str.istitle()**
  - Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.
- **str.isupper()**
  - Return true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.
- **str.join(*iterable*)**

- Return a string which is the concatenation of the strings in the iterable. A TypeError will be raised if there are any non-string values in*iterable*, including bytes objects. The separator between elements is the string providing this method.

- **str.ljust(*width*[, *fillchar*])**
  - Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to len(s).
- **str.lower()**
  - Return a copy of the string with all the cased characters converted to lowercase.
- **str.partition(*sep*)**
  - Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator.
  - If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.
- **str.replace(*old*, *new*[, *count*])**
  - Return a copy of the string with all occurrences of substring *old* replaced by *new*.
  - If the optional argument *count* is given, only the first countoccurrences are replaced.
- **str.split(*sep=None*, *maxsplit=-1*)**

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most maxsplit+1 elements).

  - If *maxsplit* is not specified or -1, then there is no limit on the number of splits
- **str.swapcase()**
  - Return a copy of the string with uppercase characters converted to lowercase and vice versa.
- **str.title()**
  - Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.



Slicing:

- Like other programming languages, it's possible to access individual characters of a string by using array-like indexing syntax. In this we can access each and every element of string through their index number and the indexing starts from 0. Python does index out of bound checking.
- So, we can obtain the required character using syntax, **string_name[index_position]**:

- The positive index_position denotes the element from the starting(0) and the negative index shows the index from the end(-1).
- To extract substring from the whole string then then we use the syntax like
- **string_name[beginning: end : step]**beginning represents the starting index of string end denotes the end index of string which is not inclusive steps denotes the distance between the two words.
  - Example :print x[2:5] # Prints substring stepping up 2nd character
  - Example : print [4:10:2] # from 4th to 10th character
  - print x[-5:-3] # Prints 3rd character from rear from 3 to 5

Lists:

- **Introduction:**
  - Python knows a number of *compound* data types, used to group together other values.
  - The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets.
  - Lists might contain items of different types, but usually the items all have the same type.
  - Example : fruits = ['apples', 'oranges', 'pears', 'apricots']
  - Example :fruits = [*'apples', 1, 'pears', 2]*
  - **Accessing List:**
  - Like strings (and all other built-in sequence type), lists can be indexed and sliced
  - Example: fruits[0]
  - Example :fruits[-3:-1]
  - Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content.

List Operation:

- **1. "in" operator** :- This operator is used to **check if an element is present** in the list or not. Returns true if element is present in list else returns false.
- Example fruits = [*'apples', 'pineapple', 'pears', 'banana']*

if *'apples' in fruits:*

print(*"apple is in list")*

- **2. "not in" operator** :- This operator is used to **check if an element is not present** in the list or not.
- Returns true if element is not present in list else returns false.

Example fruits = [*'apples', 'pineapple', 'pears', 'banana']*

if 'strawberries' not in fruits:

print("strawberries is not in list")

- **3 .** With List we can iterate, find the element , concate the list and so more like...

Example :

len([1, 2, 3]) :	Find out the length
[1, 2, 3] + [4, 5, 6] :	Concatenation of 2 lists
['Hi!'] * 4 :	Repetition (['Hi!', 'Hi!', 'Hi!', 'Hi!'])
for x in [1, 2, 3]:	
print x,:	Iteration

- 4. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

Example :

```
>>> squares = []
>>> for x in range(10):
 ... squares.append(x**2)
... >>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
        OR
squares = [x**2 for x in range(10)]
```

```
fruits = ['apples', 'pineapple', 'pears', 'banana']

juiceFruits=['Mausambi','Mango','Orange']

print("Slicing ",fruits[-3:-1])

print("Slicing ",fruits[3:1])

print("Slicing ",fruits[1:4])

fruits=fruits+juiceFruits

print("final list  ",fruits)
```

Functions

Name	Description
len(list)	Gives the total length of the list.
max(list)	Returns item from the list with max value.
min(list)	Returns item from the list with min value.
list(seq)	Converts a tuple into list.

Methods	Description
list.append(x)	Add an item to the end of the list. Equivalent to a [len(a):]=[x].
list.extend(L)	Appends the contents of L to list
list.insert(I,x)	Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0,x) inserts at the front of the list,and a.insert(len(a),x) is equivalent to a.append(x).
list.count(obj)	Returns count of how many times obj occurs in list
list.index(obj)	Returns the lowest index in list that obj appears
List.pop(obj=list[-1]	Removes and returns last object or obj from list.
List.reverse()	Reverses Objects of list in place
List.sort([fun])	Sorts objects of list, use compare function If given
List.remove(obj)	Removes object obj from list

```
fruits = ['apples', 'pineapple', 'pears', 'banana']
lenList=len(fruits)
maxItem=max(fruits)
minItem=min(fruits)
print ("Length of List ",lenList," Max Item",maxItem," Min Item ",minItem)
```

```
fruits = ['apples', 'pineapple', 'pears', 'banana']
if 'apples' in fruits:
        print("apple is in  list")

if 'tomeoto' not in fruits:
        print("tomeoto is not in fruit ")

print("The lenght of fruit list is ",len(fruits))

fruits=fruits+['strawberry','custured apple']

print("After concatination the fruits list is ",fruits)
```

```
squares=[]
for x in range(10):
    squares.append(x**2)
print("Squares 1 :",squares)

squares = [x**2 for x in range(10)]
print("Squares 2 :",squares)
```

```
fruits = ['apples', 'pineapple', 'pears', 'banana']
juiceFruits=['Mausambi','Mango','Orange']
fruits.append('Mango')
print("After append the list is ",fruits)
print("Count apples in List ",fruits.count("apples"))
fruits.extend(juiceFruits)
print("After extend the fruit list is ",fruits)
print("Index of apples ",fruits.index("apples"))
fruits.insert(2, "Guava")
print("Insert Guava in fruit list ",fruits)
print(fruits.pop(1),"<--After pop fruit list ",fruits)
print(fruits.remove("Mango"),"After remove the list is ",fruits)

fruits.reverse()
print("Reverse the list ",fruits)

fruits.sort()
print("Sort the list ",fruits)
```

Using Lists as Stacks

- The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in,first-out”) . To add an item to the top of the stack,use append().

- To retrieve an item from the top of the stack,use pop() with out an explicit index.

```
stackList=[22,11,44]
stackList.append(66)
stackList.append(55)
print(stackList)
print("Pop operation ",stackList.pop())
print("Pop operation ",stackList.pop())
print("Stack ",stackList)
```

Using Lists as Queues

- It is also possible to use a list as a queue,where the first element added is the first element retrieved("first-in,
- first- out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow(because all of the other elements have to be shifted by one).

```
from collections import deque
queue=deque(["Mr. Black","Mr. Smith","Ms. White"])
print("Queue ",queue)
queue.append("Mr. Blue")
print("Queue ",queue)
print(queue.popleft()," After popleft , the queue is ",queue)
print(queue.popleft()," After popleft , the queue is ",queue)
print(queue.pop()," After pop , the queue is ",queue)
```

Tuple

- Introduction
- Accessing tuples
- Operations
- Working
- Functions and Methods

Introduction

- A tuple is a sequence of immutable Python objects.
- Tuples are sequences, just like lists.
- The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.
- Eg fruits=("Mango","Banana","Oranges",23,44)
- Eg numbers=(11,22,33,44)
- Eg fruits="Mango","Banana","Oranges"

Accessing tuples

- There are various ways in which we can access the elements of a tuple.
- We can use the index operator [] to access an item in a tuple.
- Index starts from 0. The index must be an integer.
- Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.
- We can access a range of items in a tuple by using the slicing operator (colon).
  Eg. fruits [2:]
  Example

```
tuple_Items = (4, 2, 3, [6, 5])
print("Tuple ",tuple_Items)
tuple_Items[3][0] = 9
print("Tuple ",tuple_Items)
tuple_Items = ('p','r','o','g','r','a','m','i','z') # tuples can be
reassigned
print("Tuple Items ",tuple_Items)
tuples_NestedItems = ("mouse", [8, 4, 6], (1, 2, 3))
print("Tuple Nested Items  ",tuples_NestedItems)
print("Accessing the 0,3 element from Nested Items
",tuples_NestedItems[0][3]," Accessing the 1,2 element from Nested Items
",tuples_NestedItems[1][2])
del tuples_NestedItems
print("Tuple is deleted")
repeat_tuple=('Hello',) * 3
repeat_tuple1=('Hello') * 3
print("Repeated Tuple Items ",repeat_tuple," Another repeated tuple
",repeat_tuple1)
```

- **Operations**
  - With Tuples we can do concate ,repetation,iterations etc…
  - Example
- **Working**
  - Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once it has been assigned.
  - But if the element is itself a mutable data type like list, its nested items can be changed.
  - We can also assign a tuple to different values (reassignment).
  - Also we cannot delete or remove items from a tuple.
  - But deleting the tuple entirely is possible using the keyword del.

```
tuple_Items = (4, 2, 3, [6, 5])
print("Tuple ",tuple_Items)
tuple_Items[3][0] = 9
print("Tuple ",tuple_Items)
tuple_Items = ('p','r','o','g','r','a','m','i','z') # tuples can be
reassigned
print("Tuple Items ",tuple_Items)
tuples_NestedItems = ("mouse", [8, 4, 6], (1, 2, 3))
print("Tuple Nested Items  ",tuples_NestedItems)
print("Accessing the 0,3 element from Nested Items
",tuples_NestedItems[0][3]," Accessing the 1,2 element from Nested Items
",tuples_NestedItems[1][2])
del tuples_NestedItems
print("Tuple is deleted")
repeat_tuple=('Hello',) * 3
repeat_tuple1=('Hello') * 3
print("Repeated Tuple Items ",repeat_tuple," Another repeated tuple
",repeat_tuple1)
```

u

nctions and methods

- Functions of Tuple

Method	Description
len(tuple)	Gives the total length of the tuple.
count(obj)	Returns count of how many times obj occurs in tuple
tuple(seq)	Converts a seq into tuple.
index(obj)	Returns the lowest index in tuple that obj appears

```
fruits=('Apple','Banana','Watermelon','Strawberry')

juicyFruitsList=['Oranges','Mango','Mausambi','Pineapple']

print("Length of Tuple ",len(fruits))
print("Max in fruits tuple--> ",max(fruits)," Min in fruits tuple -->
",min(fruits))
print("List to  Tuple ",tuple(juicyFruitsList))
```

- **Methods of Tuple**

```
fruits=('Apple','Banana','Watermelon','Strawberry')

print("Count of watermelon item in fruit tuple
",fruits.count("Watermelon"))
print("Index of fruits tuple  ",fruits.index("Watermelon"))
```

Dictionaries

- Introduction
- Accessing values in dictionaries
- Working with dictionaries
- Properties
- Functions

Introduction

- Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”.
- Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys.
- Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.
- You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().
- The main operations on a dictionary are storing a value with some key and extracting the value given the key.
- Like lists they can be easily changed, can be shrunk and grown ad libitum at run time. They shrink and grow without the necessity of making copies. Dictionaries can be contained in lists and vice versa.
- A list is an ordered sequence of objects, whereas dictionaries are unordered sets.
- But the main difference is that items in dictionaries are accessed via keys and not via their position.

```
cityCode={'Ahmedabad':79, 'Mumbai':22,'Pune':20,'Delhi':11}
cityFamousPlace={"Ahmedabad":"IIM","Delhi":"Metro","Mumbai":"Bollywood"}

cityFamousPlaceCopy=cityFamousPlace.copy()
print("copied disctionary ",cityFamousPlaceCopy)

print("Before updating the cityCode is ",cityCode)
cityCode.update(cityFamousPlace)
print("updated cityCode ",cityCode)

print("Values of cityCode ",cityCode.values())
print(" Keys of cityCode ",cityCode.keys())

print("cityCode Items ",cityCode.items())

cityCode.clear()
print("After cityCode clear ",cityCode)
```

- **Accessing values in dictionaries**
  - To access dictionary elements, we can use the familiar square brackets along with the key to obtain its value.
  - It is an error to extract a value using a non-existent key.

```
cityFamousPlace={"Ahmedabad":"IIM",'mumbai':'Juhu',"Delhi":"Metro","Mumbai"
:"Bollywood","Delhi":"Red fort"}
print(cityFamousPlace)

for key in cityFamousPlace:
    print(key,"  ",cityFamousPlace[key])
```

- **Working with dictionaries**

- We can also create a dictionary using the built-in class dict() (constructor).
- We can test if a key is in a dictionary or not using the keyword **in**.
- The membership test is for **keys** only, **not for values**.

Eg. Squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
>>> 1 in squares
True
>>>2 not in squares
True
>>> 49 in squares
False

- We can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry.
- Eg. dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
- You can either remove individual dictionary elements or clear the entire contents of a dictionary.
- You can also delete entire dictionary in a single operation.
- Eg. dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
- del dict['Name']; # remove entry with key 'Name'
- dict.clear(); # remove all entries in dict
- del dict ; # delete entire dictionary

○ Properties of Dictionaries

- Dictionary values have no restrictions.
- They can be any arbitrary Python object, either standard objects or user-defined objects.
- However, same is not true for the keys.
- **(a)** More than one entry per key not allowed.
- Which means no duplicate key is allowed.
- When duplicate keys encountered during assignment, the last assignment wins.

Eg : dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print "dict['Name']: ", dict['Name']
Output: dict['Name']: Manni

(b) Keys must be immutable.

Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Eg: dict = {['Name']: 'Zara', 'Age': 7}

print "dict['Name']: ", dict['Name']
Output : Error (TypeError: list objects are unhashable)

○ Methods and Functions

- Methods

dist.copy()	A dictionary can be copied with the method copy():.
dist.update()	It merges the keys and values of one dictionary into another, overwriting values of the same key.
dist.values()	It returns the list of dictionary dict's values.
dist.keys()	It returns the list of dictionary dict's keys.
dist.items()	It returns the list of dictionary dict's keys,values in tuple pairs.
dist.clear()	Removes all elements of dictionary dict.

```
cityCode={'Ahmedabad':79, 'Mumbai':22,'Pune':20,'Delhi':11}
cityFamousPlace={"Ahmedabad":"IIM","Delhi":"Metro","Mumbai":"Bollywood"}

cityFamousPlaceCopy=cityFamousPlace.copy()
print("copied disctionary ",cityFamousPlaceCopy)

print("Before updating the cityCode is ",cityCode)
cityCode.update(cityFamousPlace)
print("updated cityCode ",cityCode)

print("Values of cityCode ",cityCode.values())
print(" Keys of cityCode ",cityCode.keys())

print("cityCode Items ",cityCode.items())

cityCode.clear()
print("After cityCode clear ",cityCode)
```

○ Functions

- Defining a function
- Calling a function
- Types of functions

- Function Arguments
- Anonymous functions
- Global and local variables
- 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.

Defining a function

- Python gives us many built-in functions like print(), etc. but we can also create our own functions.
- A function in Python is defined by a def statement. The general syntax looks like this:

```
def function-name(Parameter list):
        Statements, i.e. the function body
return [expression]
```

- The keyword ”def” introduces a function *definition.*
- It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.
- The ”return” statement returns with a value from a function.”return” without an expression argument returns None.
- Falling off the end of a function also returns None.
- Example:

```
def checkNoEvenOdd(x):
    if x % 2 == 0:
        print(x, "is even")
    else:
        print(x, "is odd")
```

  - Calling a function
  - Once function define, we can call it directly or in any other function also.
  - Eg. checkNoEvenOdd(20)
  - Output: 20 is even
- Types of functions

Functions can be of	
Built-in functions	Functions that come built into the Python language itself are called built-in functions and are readily available to us. Eg: input(),eval(),print() etc…
User defined functions	Functions that we define ourselves to do certain specific task are referred as user-defined functions. Eg:checkNoEvenOdd(20)

- Function Arguments
  - It is possible to define functions with a variable number of arguments.
  - The function arguments can be
    - Default arguments values
    - Keyword arguments
    - ArbitraryArgumentLists
- **Default arguments values**
  - The most useful form is to specify a default value for one or more arguments.

- This creates a function that can be called with fewer arguments than it is defined to allow.
- Eg. def employeeDetails(name,gender='male',age=35)
- This function can be called in several ways:
- giving only the mandatory argument : employeeDetails("Ramesh")
- giving one of the optional arguments: employeeDetails("Ramesh",'Female')
- or even giving all arguments : employeeDetails("Ramesh",'Female',31)
- Note : The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:
- If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
return L
```

```
def testFunction():
    return "This is testFunction "
def testFunction1():
    return 44
def testFunction2():
    return range(10)
print(testFunction())
print(testFunction1())
print(testFunction2())

for i in testFunction2():
    print(i)
```

- **Keyword Arguments**
  - Functions can also be called using keyword arguments of the form kwarg=value.
  - For instance, the following function:
  - def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
  - accepts one required argument (voltage) and three optional arguments (state, action, and type).

```
def testFunction(firstArg,secondArg,thridArg,*args,**kwargs):

print("This is testFunction
",firstArg,secondArg,thridArg,args,kwargs["one"],kwargs["two"])

testFunction(11, 22, 33,44,55,66,77,one=88,two=99)
```

- **ArbitraryArgumentLists**
  - These arguments will be wrapped up in a tuple . Before the variable number of arguments, zero or more normal arguments may occur.
  - Eg : def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
  - Normally, these variadic arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function.

- Any formal parameters which occur after the *args parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

```
def concat(*args, sep="/"):
return sep.join(args)
```

- concat("earth", "mars", "venus") O/P->'earth/mars/venus'
- concat("earth", "mars", "venus", sep=".") O/P->'earth.mars.venus'

```
def testFunction(firstArg,secondArg,thridArg,*args):

    print ("This is testFunction ",firstArg,secondArg,thridArg)
    print("Args ",args)

testFunction(11, 22, 33,44,55,66,77,88,99)
```

```
def testFunction(firstArg,secondArg,thridArg,*args,**kwargs):

      for k in kwargs:
            print(k,kwargs[k])
testFunction(11, 22, 33,44,55,66,77,one=88,two=99)
```

Anonymous functions

- 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.**
- Hence, anonymous functions are also called lambda functions.
- A lambda function has the following 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.
- Python supports a style of programming called *functional programming where you can pass functions to other functions to do stuff.*
- *Example:*
- mult3 = filter(lambda x: x % 3 == 0, [1, 2, 3, 4, 5, 6, 7, 8, 9])
  It sets mult3 to [3, 6, 9], those elements of the original list that are multiples of 3.
  This is shorter (and, one could argue, clearer) than

```
def filterfunc(x):
   return x % 3 == 0
mult3 = filter(filterfunc, [1, 2, 3, 4, 5, 6, 7, 8, 9])
or
mult3 = [x for x in [1, 2, 3, 4, 5, 6, 7, 8, 9] if x % 3 == 0]
```

- Example
- pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
- print(pairs)
- pairs.sort(key=lambda pair: pair[1])
- print(pairs)

```
# Using list comprehension
mult3 = [x for x in [1, 2, 3, 4, 5, 6, 7, 8, 9] if x % 3 == 0]
print("First Method ",mult3)

#Using function
def filterfunc(x):
    return x % 3 == 0
mult3 = list(filter(filterfunc, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
print("Second Method ",mult3)

#Using Lambda
mult3 = filter(lambda x: x % 3 == 0, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print("Third Method ",list(mult3))
```

Global & Local variables

- **Global Variables**
  - Defining a variable on the module level makes it a global variable, you don't need to global keyword.
  - The global keyword is needed only if you want to **reassign the global variables in the function/method.**
- In Python, variables that are only referenced inside a function are implicitly global.
- If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.
- Eg: def set_globvar_to_one():
- global globvar
- # Needed to modify global copy of globvar
- globvar = 1

```
global y
y=100
def testFunction():
    global x
    x=10
    print("x= ",x,"y= ",y)
    x=x+10

def useGlobalVar():
    print("From useGlobalVar x=",x," y= ",y)

testFunction()
useGlobalVar()
```

- **Local variables**
- If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.
- Local variables of functions can't be accessed from outside

Eg : def F()
s="I am local variable"
print (s)
F()
print(s) # Gives NameError: name 's' is not defined

Modules

- **Modules**

1. Importing module

2. Math module
3. Random module
4. Packages
5. Composition

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Within a module, the module's name (as a string) is available as the value of the global variable __name__.

```
def fib(n):
        a,b=0,1
        while b<n:
                print(b, end=' ')
                a,b=b,a+b
        print()
def fib2(n):
        result=[]
        a,b=0,1
        while b<n:
                result.append(b)
                a,b=b,a+b
        return result
fib(10)
print(fib2(10))
```

1. Importing module
    - Modules can import other modules.
    - It is customary but not required to place all import statements at the beginning of a module (or script, for that matter).
    - The imported module names are placed in the importing module's global symbol table.
    - Eg :import fibo
    - **Using the module name we can access functions.**
    - fibo.fib(10)
    - fibo.fib2(10)
    - There is a variant of the import statement that imports names from a module directly into the importing module's symbol table.
    - For example:
    - **from fibo import** fib, fib2
    - fib(500)
    - another way to import is
    - from fibo import *

```
from com.practice.Fibo import *

print(fib2(10))
fib(10)
```

2. Math module
    - This module is always available.
    - It provides access to the mathematical functions defined by the C standard.

- This functions are divided into some categories like Number-theoretic and representation functions, Power and logarithmic functions, Trigonometric functions, Angular conversion, Hyperbolic functions, Special functions.
- Constants
- **math.pi :** The mathematical constant π = 3.141592..., to available precision.
- **math.e** : The mathematical constant e = 2.718281..., to available precision.,
- **math.inf :** A floating-point positive infinity. (For negative infinity, use -math.inf.) Equivalent to the output of float('inf').
- **math.nan :** A floating-point "not a number" (NaN) value. Equivalent to the output of float('nan').

Function	Description
math.ceil(*x*)	Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to x.__ceil__(), which should return an Integral value.
math.factorial(*x*)	Return *x* factorial
math.floor(x)	Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to x.__floor__(), which should return an Integral value.
math.trunc(x)	Return the Real value *x* truncated to an Integral (usually an integer).
math.pow(x,y)	Return x raised to the power y.

```
from math import pi

print([str(round(pi, i)) for i in range(1, 6)])

piList=[]
for i in range(1, 6):
   piList.append(str(round(pi, i)))

print(piList)
```

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
   if not math.isnan(value):
      filtered_data.append(value)

print(filtered_data)
```

Random Module

- Python offers random module that can generate random numbers.
- These are pseudo-random number as the sequence of number generated depends on the seed.

- Random Module Functions

random.randrange(*start*, *stop*[, *step*])	Return a randomly selected element from range(start, stop, step). This is equivalent to choice(range(start, stop, step)), but doesn't actually build a range object.
random.randint(*a*, *b*)	Return a random integer *N* such that a <= N <= b. Alias for randrange(a, b+1).
random.choice(*seq*)	Return a random element from the non-empty sequence *seq*.
random.random()	Return the next random floating point number in the range [0.0, 1.0).

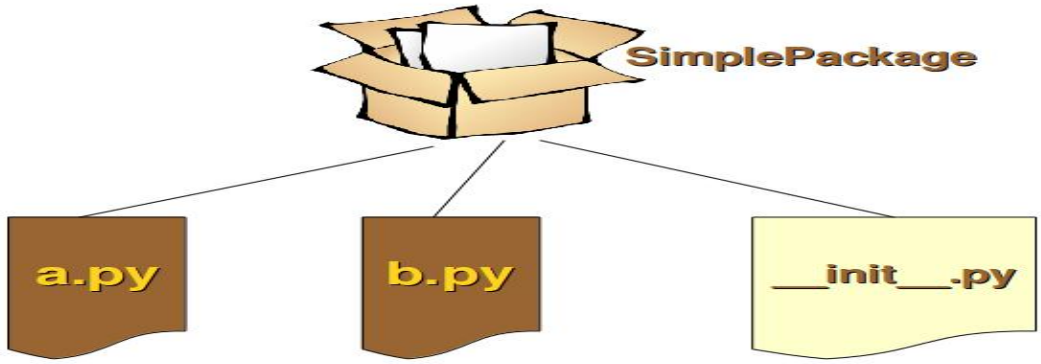
```
import math
print("math.ceil(12.4) : ",math.ceil(12.4))
print("math.degrees(20) : ",math.degrees(20))
print("math.pow(2, 3) : ",math.pow(2, 3))
print("math.trunc(23.8) : ",math.trunc(23.8))
print("math.floor(12.8) : ",math.floor(12.8))
```

And many more…	

```
import random
print("randint()")
for i in range(1,5):
        print(random.randint(2,5))
print("randrange()")
for i in range(1,5):
        print(random.randrange(10,30))
```

Packages

- Packages are a way of structuring Python's module namespace by using "dotted module names".
- For example, the module name A.B designates a submodule named B in a package named A.
- A package is basically a directory with Python files.



- a.py
- def bar():

  print("Hello, function 'bar' from module 'a' calling")

- b.py
- def foo():

  print("Hello, function 'foo' from module 'b' calling")

Calling a abd b module from package (SimplePackage)

```
from SimplePackage import a, b
a.bar()
b.foo()
```

- **Input-Output**
  - Printing on screen
  - Reading data from keyboard
  - Opening and closing file
  - Reading and writing files
  - Functions
- **Printing on screen**
- "print()" is use to print on screen.
- print(value, ..., sep=' ', end='\n', file=sys.stdout)
- Prints the values to a stream, or to sys.stdout by default.
- Optional keyword arguments:
- file: a file-like object (stream); defaults to the current sys.stdout.
- sep: string inserted between values, default a space.
- end: string appended after the last value, default a newline.
- **Reading from keyboard**
- To read data from keyboard "input()" is use.
- The input of the user will be returned as a string without any changes.
- If this raw input has to be transformed into another data type needed by the algorithm, we can use either a casting function or the eval function.

```
inputFromUser=input("Enter Any Value")
int Number=int(inputFromUser)

print(type(int Number))
finalNumber=eval('intNumber+10')
print(finalNumber)
```

- Opening and Closing file
    - “open()” is use to open the file.
    - It returns file object.
    - syntax
    - open(fileName,mode).
    - fileName: Name of the file that we wants to open.
    - mode: ‘r’ (only for reading), ‘w’ (only for writing), ‘a’ (for append) , r+ (for read and write).
    - Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding.
    - If encoding is not specified, the default is platform dependent .
    - ’b’ appended to the mode opens the file in binary mode.
    - **Close the file**
    - call f.close() to close it and free up any system resources taken up by the open file.
- Reading and writing files
    - file.read(size) :This function is use to read a file’s contents.
    - If size is ommited or negative then the entire content is returnd.
    - If the end of the file has been reached, f.read() will return an empty string (’ ’).
    - f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline.
    - For reading lines from a file, you can loop over the file object.
    - This is memory efficient, fast, and leads to simple code:
      for line in f:
      print(line, end='')
    - **f.write(string) ,** writes the contents of string to the file, returning the number of characters written.
    - Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) –before writing them.

      Eg :value = ('the answer', 42)

      s = str(value) # convert the tuple to string

      f.write(s)
- File Functions
    - f.tell() : It returns an integer giving the file object’s current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
    - f.seek(): To change the file object’s position.
    - f.seek(offset, from_what)
    - Eg : f.seek(5)
    - f.seek(-3,2)

```
file=open("Test.txt","r")
currentPosition=file.tell()
print("Current Position ",currentPosition)
fileContent=file.read()

print("File Content ",fileContent)
currentPosition=file.tell()
print("After reading file the Position is ",currentPosition)
```

```
file=open("Test.txt","r")
file.seek(10)
fileContent=file.read()

print("File Content ",fileContent)
```

```
file=open("Test.txt","r+")
content=file.read()
print("Data before Writing ",content)
data=input("Enter data")
file.write(data)
file.seek(0)
content=file.read()
print("Data after Writing ",content)
```

Exception

- Exception Handling
- Exception
- Exception Handling
- Except clause
- Try ? finally clause
- User Defined Exceptions

- **Exception**
  - In python , there are two distinguishable kinds of errors: syntax errors and exceptions.
  - Syntax errors, also known as parsing errors
  - Eg: for i in range(1,10)
  - print(i)
  - Here missing ":" after for is syntax error.
  - Exceptions : Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
  - Errors detected during execution are called exceptions
  - Example (Exception like TypeError,NameError,DivdedByZeroError)
  - Eg : 10*(1/0) →ZeroDivisionError: division by zero
  - Eg : 4+spam*3 → NameError: name 'spam' is not defined
  - Eg: '2'+2 → TypeError: Can't convert 'int' object to str implicitly
- **Exception Handling**
  - Exceptions handling in Python is very similar to Java.
  - But whereas in Java exceptions are caught by catch clauses, we have statements introduced by an "except" keyword in Python.
  - Eg : n = int(input("Please enter a number: "))

- Please enter a number: 23.50
- Exception occurs like
- ValueError: invalid literal for int() with base 10: '23.5'

```
while True:
        try:
                n = input("Please enter an integer: ")
                n = int(n)
                break
        except ValueError:
                print("No valid integer! Please try again ...") print("Great, you successfully
entered an integer!")
```

- Except clause
  - A try statement may have more than one except clause for different exceptions.
  - But at most one except clause will be executed.
  - Eg :
  - import sys
  - try:

```
        ……
        ……
        ……
except IOError:
        ……..
except ValueError:
        …….
except :
        sys.exc_info()[0]
```

```
while True:
   try:
      x = int(input("Please enter a number: "))
      break
   except ValueError:
      print("Oops!  That was no valid number.  Try again...")
```

```
import sys
try:
   f = open('T2est.txt')
   s = f.readline()
   print("s ",s)
   i = int(s.strip())
   print(" i ",i)
except OSError as err:
   print("OS error: {0}".format(err))
except ValueError:
   print("Could not convert data to an integer.")
except:
```

Try …..finally clause

- try statement had always been paired with except clauses. But there is another way to use it as well.
- The try statement can be followed by a finally clause.

- Finally clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a "finally" clause is always executed regardless if an exception occurred in a try block or not.
- Syntax:
  try:
  ……
  ……
  finally :
  ……

- User defined Exception
  - Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.
    class MyNewError(Exception):
    pass

```
class Error(Exception):
   """Base class for other exceptions"""
   pass

class ValueTooSmallError(Error):
   """Raised when the input value is too small"""
   pass

class ValueTooLargeError(Error):
   """Raised when the input value is too large"""
   pass
# our main program
# user guesses a number until he/she gets it right
# you need to guess this number
number = 10
```

raise MyNewError("Something happened in my program")

Advanced Python

- OOPS in Python
- Regular expressions
- CGI
- Database
- Networking
- Multithreading
- GUI Programming
- Designing
- Web Programming
- Connectivity with MySql

OOPs concept

- Class and object
- Attributes
- Inheritance
- Overloading
- Overriding
- Data hiding

- Class and Object

- Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name.
- The class definition looks like :
- class ClassName:
- Statement 1
- Statement 2
- …….
- Statement N
- The statements inside a class definition will usually be function definitions, but other statements are also allowed.
- When a class definition is entered, a new namespace is created, and used as the local scope—thus, all assignments to local variables go into this new namespace.
- In particular, function definitions bind the name of the new function here.

- **Member methods in class**
  - The *class_suite* consists of all the component statements defining class members, data attributes and functions.
  - The class attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
  - Eg. displayDetails()
- **Static methods in Python**
  - Static methods in Python are similar to those found in Java or C++.
  - A static method does not receive an implicit first argument.
  - To declare a static method, use this idiom:

```
class C:

    def f(arg1, arg2, ...): ...

    f = staticmethod(f)
```

staticmethod(function) -> method

  - Convert a function to be a static method.
  - It can be called either on the class (e.g. C.f()) or on an instance
  - (e.g. C().f()). The instance is ignored except for its class.
- **Object**
  - Class objects support two kinds of operations: attribute references and instantiation.
  - Attribute references use the standard syntax used for all attribute references in Python: obj.name
  - Valid attribute names are all the names that were in the class's namespace when the class object was created..
  - Class instantiation uses function notation.
  - Just pretend that the class object is a parameterless function that returns a new instance of the class.
  - For example : x = MyClass()
  - creates a new instance of the class and assigns this object to the local variable x.
- **Inheritance**
  - Python supports inheritance, it even supports multiple inheritance.
  - Classes can inherit from other classes.

- A class can inherit attributes and behaviour methods from another class, called the superclass.
- A class which inherits from a superclass is called a subclass, also called heir class or child class.
- Superclasses are sometimes called ancestors as well.

- Overloading
    - Python supports operator and function overloading.
    - Method overloading
        - Overloading is the ability to define the same method, with the same name but with a different number of arguments and types.
        - It's the ability of one function to perform different tasks, depending on the number of parameters or the types of the parameters.
    - Operator Overloading
        - Python operators work for built-in classes.
        - But same operator behaves differently with different types.
        - For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.
        - This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

Regular Expression

- match function
- Search function
- Matching Vs Searching
- Modifiers
- Patterns

- Regular expressions (called REs,or regexes,or regexpatterns)are essentially a tiny , highly specialized programming language embedded inside Python and made available through the re module.
- Regular expression patterns are compiled into a series of bytecodes.
- The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions.
- Since regular expressions are used to operate on strings, we'll begin with the most common task: matching characters.
- **Match function**
- This function determines if the RE matches at the beginning of the string.
- Syntax
- re.match(pattern,string, flags=0)
- pattern :This is the regular expression to be matched.
- string :This is the string, which would be searched to match the pattern at the beginning of string.
- flags :You can specify different flags using bitwise OR (|).
- **Search function:**
    - Scan through a string, looking for any location where this RE matches.
    - Syntax :re.search(pattern,string, flags=0)
    - pattern :This is the regular expression to be matched.
    - string :This is the string, which would be searched to match the pattern anywhere in the string.
    - flags :You can specify different flags using bitwise OR (|).
- Match vs search

- a pattern. But search attempts this at all possible starting points in the string. Match just tries the first starting point.

- **Modifiers** :

  - Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|).

Modifier	Description
re.I :IGNORECASE,	Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, [A-Z]will match lowercase letters, too, and Spam will match Spam, spam, or spAM.
re.L LOCALE	
re.M :MULTILINE,	Multi-line matching, affecting ^ and $. When this flag is specified, ^ matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the $metacharacter matches either at the end of the string and at the end of each line
re.A ASCII	Make \w, \W, \b, \B, \s and \S perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode patterns, and is ignored for byte patterns.
re.S DOTALL	Makes the '.' special character match any character at all, including a newline; without this flag, '.' will match anything except a newline.
re.XVERBOSE	When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly.

Patterns

- Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Description	
|	Alternation, or the “or” operator.
^	Matches at the beginning of lines.
$	Matches at the end of a line,
\A	Matches only at the start of the string.
\Z	Matches only at the end of the string.
\b	Word boundary. This is a zero-width assertion that matches only at the beginning or end of a word.
\w	Matches the word characters.
\W	Matches the nonword characters.
\d	Matches digits.
\D	Matches non digits.
\s	Matches whitespaces.
\S	Matches nonwhitespaces.
\B	Another zero-width assertion, this is the opposite of \b, only matching when the current position is not at a word boundary.
re{n,m}	Matches at least n and at most m occurrences of preceding expression.

```
import re
import sys
pattern = 'Fred'
file=open("Test.txt","r")
regexp = re.compile(pattern)
for line in file:
   match = regexp.search(line)
   if match:
      print(line)
```

```
Adam
Brenda
Charlie
Dawn
Eric
Erica
....
```

```
import re
test=re.search("[0-9]+", "My Contact Number is 987822227 and 7777234567 ")

#span() returns a tuple with the start and end position

print("Span is ",test.span()," Start at ",test.start(),"Ends at ",test.end())


anotherTest=re.search("[^0-9]+", "My Contact Number is 987822227 and 7777234567 ")
print("Span is ",anotherTest.span()," Start at ",anotherTest.start(),"Ends at ",anotherTest.end())
```

```
import re
test=re.search("[a-z]+", "My Contact Number is 987822227 and 7777234567 ",re.I)
print("Span is ",test.span()," Start at ",test.start(),"Ends at ",test.end())
p=re.compile("(\w*\s*)+", re.I)
anotherTest=re.match(p, "My Contact Number is 987822227 and 7777234567 ")
print("Span is ",anotherTest.span(),"Start at ",anotherTest.start(),"Ends at ",anotherTest.end())
print(anotherTest.group())
```

Database

- Introduction
- Connections
- Executing queries
- Transactions
- Handling error

- **Introduction**
  - SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language.
  - Connections
  - To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the example.db file:
  - **importsqlite3**
  - conn=sqlite3.connect('example.db')
  - You can also supply the special name :memory: to create a database in RAM.
  - **Executing queries**

- Once you have a Connection, you can create a Cursor object and call its execute() method to perform SQL commands.
- There are execute(),executemany(),executescript() methods to execute queries.
- **Transaction**
- By default, the sqlite3 module opens transactions implicitly before a Data Modification Language (DML) statement (i.e.INSERT/UPDATE/DELETE/REPLACE), and commits transactions implicitly before a non-DML, non-query statement .
- **Atomicity:** Either a transaction completes or nothing happens at all.
- **Consistency:** A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation:** Intermediate results of a transaction are not visible outside the current transaction.
- **Durability:** Once a transaction was committed, the effects are persistent, even after a system failure.
- The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()

employees = [ (22,'Mr. Patel',5644.67,'2001-01-12'),
           (33,'Mr. Black',90888.67,'2016-05-25'),
           (44,'Mr. Blue',1234.67,'2011-10-15'),
         ]
c.executemany('INSERT INTO Employee VALUES(?,?,?,?)', employees)

conn.commit()

c.execute("SELECT * FROM Employee")
for row in c:
   print (row)

conn.close()
```

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()

c.execute("SELECT * FROM Employee")
for row in c:
   print (row)
sql = "DELETE FROM EMPLOYEE WHERE salary > '%d'" %(12000)
try:
  # Execute the SQL command
   c.execute(sql)
  # Commit your changes in the database
   conn.commit()
except:
  # Rollback in case there is any error
   conn.rollback()
conn.close()
```

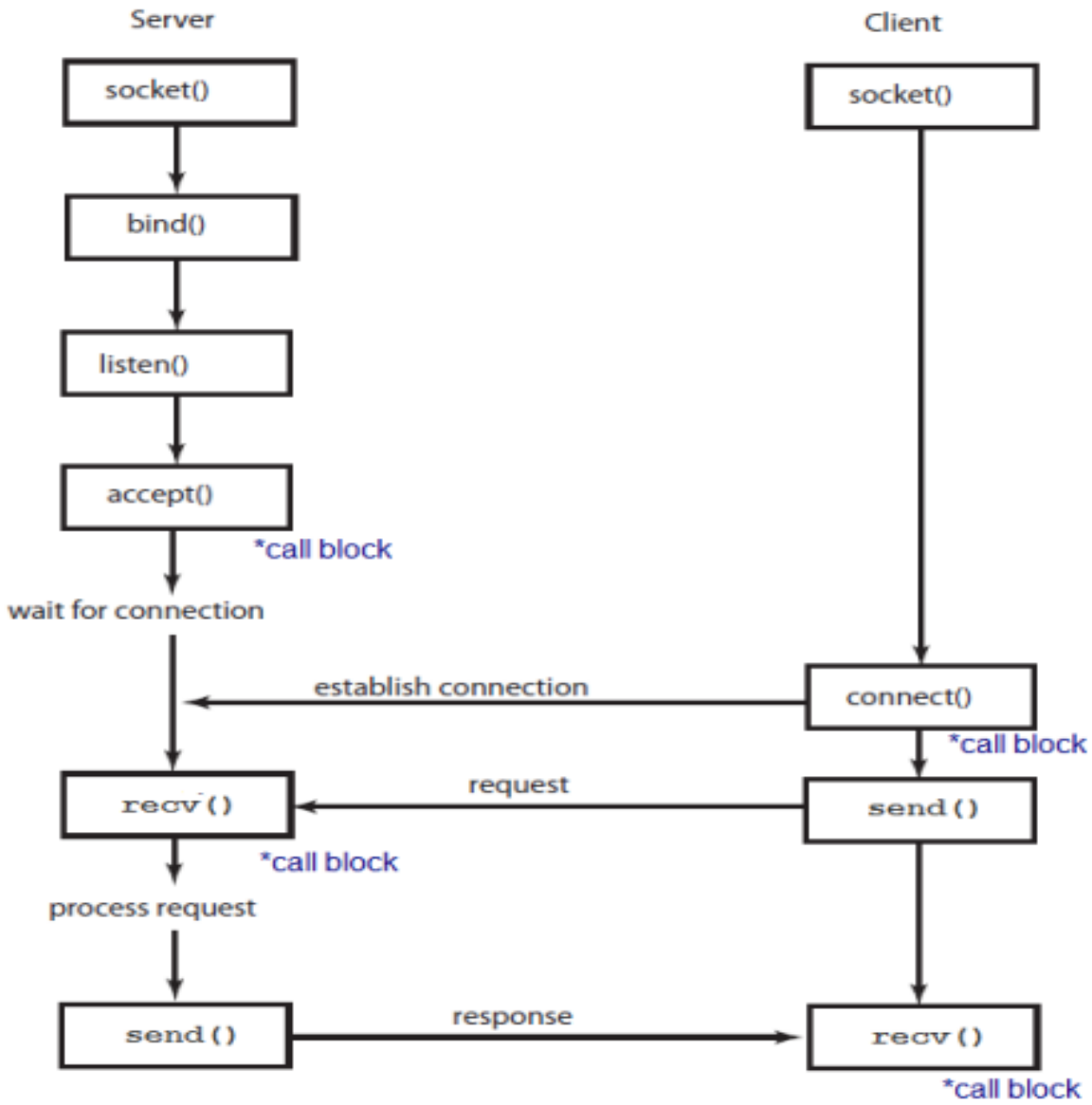
Networking

- Socket
- Socket Module
- Methods
- Client and server
- Internet modules

- **Socket**
  - Sockets are the endpoints of a bidirectional communications channel.
  - Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.
- **Socket Module**
  - To create/initialize a socket, we use the *socket.socket()* method defined in the Python's *socket* module.
  - Its syntax is as follows.
  - sock_obj = socket.socket(socket_family, socket_type, protocol=0)
  - **socket_family:** This is either AF_UNIX or AF_INET,
  - AF_INET: IP version 4 or IPv4
  - AF_UNIX : Unix socket
  - socket_type
- **socket_type:** Defines the types of communication between the two end-points. It can have following values.
  - SOCK_STREAM (for connection-oriented protocols e.g. TCP), or
  - SOCK_DGRAM (for connectionless protocols e.g. UDP).
  - SOCK_RAW (For Raw socket)
- **protocol:** This is usually left out, defaulting to 0.
  - It's usually used with raw sockets. Like IPPROTO_ICMP,
  - IPPROTO_IP,IPPROTO_RAW,IPPROTO_TCP,IPPROTO_UDP.
- **Methods**

Method	Description
accept()	Accept a new connection
bind(address)	Bind to an address and port
close()	Close the socket
connect(address)	Connect to remote socket
fileno()	Return integer file descriptor
getpeername()	Get name of remote machine
getsockname()	Get socket address as (ipaddr,port)
s.getsockopt(...)	Get socket options
s.listen(backlog)	Start listening for connections
accenpt()	Accept a new connection
bind(address)	Bind to an address and port
close()	Close the socket

connect(address)	Connect to remote socket
fileno()	Return integer file descriptor
getpeername()	Get name of remote machine
getsockname()	Get socket address as (ipaddr,port)
s.getsockopt(...)	Get socket options
s.listen(backlog)	Start listening for connections



```
import socket
serversocket = socket.socket(
        socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = socket.gethostname()

port = 9999
# bind to the port
serversocket.bind((host, port))
```

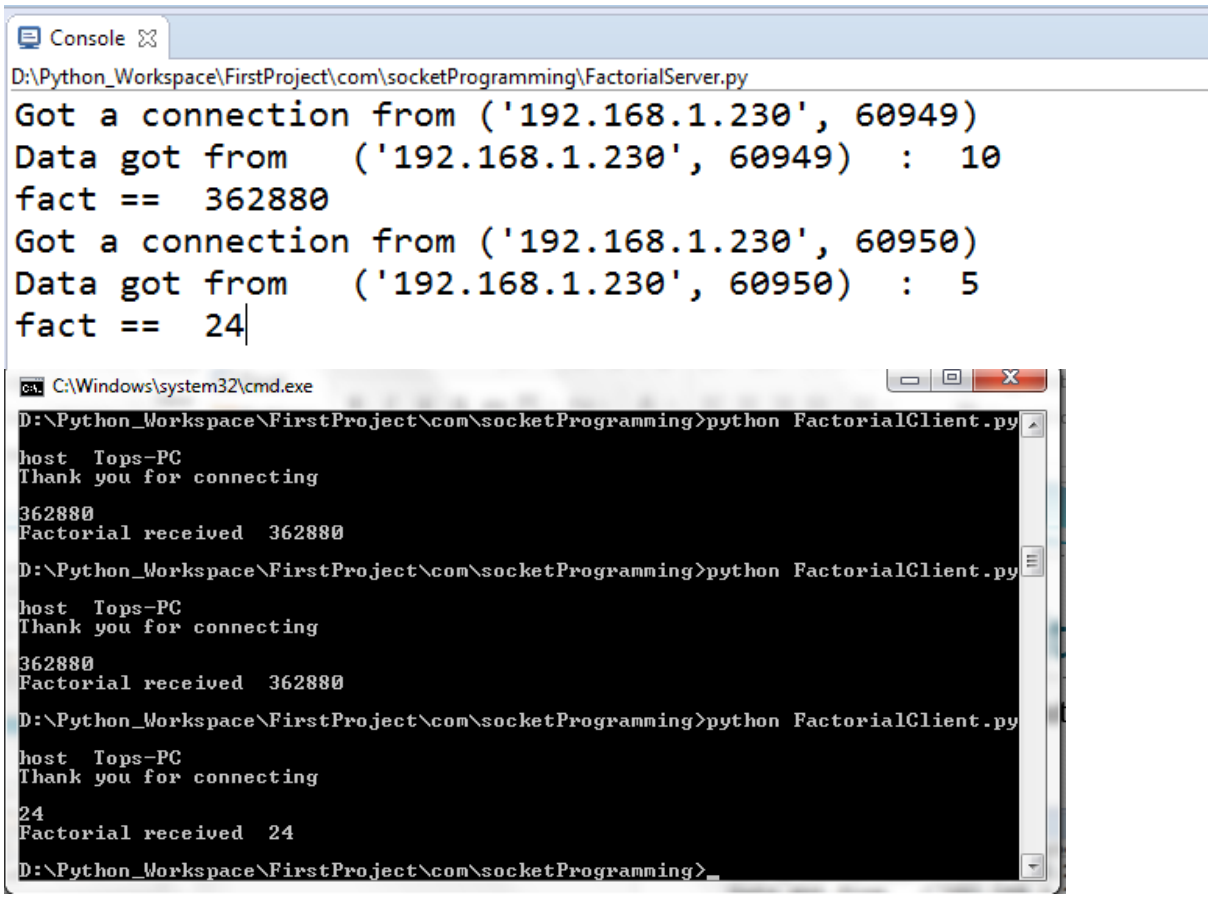
```
import socket

# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = socket.gethostname()
print("host ",host)
port = 9999

# connection to hostname on the port.
s.connect((host, port))
```

```
D:\Python_Workspace\FirstProject\com\socketProgramming>python Example1.py
Got a connection from ('192.168.1.230', 50437)
```

```
D:\Python_Workspace\FirstProject\com\socketProgramming>python Example1_Client.py

host  Tops-PC
Thank you for connecting


D:\Python_Workspace\FirstProject\com\socketProgramming>
```

```
import socket
serversocket = socket.socket(
        socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = socket.gethostname()

port = 9090
# bind to the port
serversocket.bind((host, port))

# queue up to 5 requests
serversocket.listen(5)

while True:
   # establish a connection
   clientsocket,addr = serversocket.accept()
   print("Got a connection from %s" % str(addr))

   msg='Thank you for connecting'+ "\r\n"
   clientsocket.send(msg.encode('ascii'))

   data=clientsocket.recv(1024)

   intData=int(data)
   print("Data got from  ",str(addr), " : ",intData)
   fact=1
   for i in range(1,intData):
      fact=fact*i

   print("fact == ",fact)
   clientsocket.sendto(str(fact).encode('utf-8'),addr)
   clientsocket.close()
```

```
import socket
# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = socket.gethostname()
print("host ",host)
port = 9090

# connection to hostname on the port.
s.connect((host, port))

# Receive no more than 1024 bytes
msg = s.recv(1024)
print (msg.decode('ascii'))

s.send(b'5')
fact = s.recv(1024)
print(int(fact))

print ("Factorial received ",fact.decode('ascii'),"   ")
s.close()
```



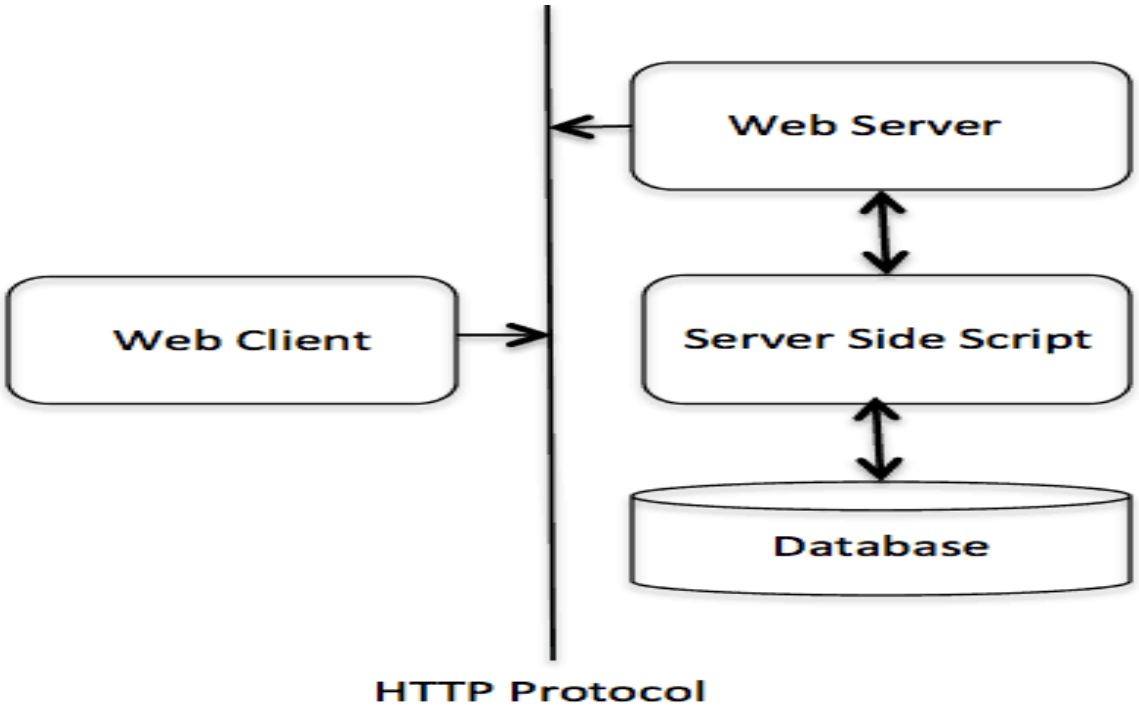
CGI

- Introduction
- Architecture
- CGI environment variable
- GET and POST methods
- Cookies
- File upload

- Introduction
  - CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML <FORM> or <ISINDEX> element.
  - Most often, CGI scripts live in the server's special cgi-bin directory.
  - The HTTP server places all sorts of information about the request (such as the client's hostname, the requested URL, the query string, and lots of other goodies) in the script's shell environment, executes the script, and sends the script's output back to the client.
  - The script's input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the "query string" part of the URL.
  - The output of a CGI script should consist of two sections, separated by a blank line.
  - The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:
  - print ("Content-Type: text/html") # HTML is following
  - print () # blank line, end of headers
  - The second section is usually HTML, which allows the client software to display nicely formatted text with header,
  - in-line images, etc. Here's Python code that prints a simple piece of HTML:

- print ("<TITLE>CGI script output</TITLE>")
- print ("<H1>This is my first CGI script</H1>")
- print ("Hello, world!")

Architecture



Variables	
CONTENT_TYPE	The data type of the content. Used when the client is sending attached content to the server. For example, file upload.
CONTENT_LENGTH	The length of the query information. It is available only for POST requests.
HTTP_COOKIE	Returns the set cookies in the form of key & value pair.
HTTP_USER_AGENT	The User-Agent request-header field contains information about the user agent originating the request. It is name of the web browser.
SERVER_NAME	The server's hostname or IP Address
SERVER_SOFTWARE	The name and version of the software the server is running.

Variables	
PATH_INFO	The path for the CGI script.
QUERY_STRING	The URL-encoded information that is sent with GET method request.
REMOTE_ADDR	The IP address of the remote host making the request. This is useful logging or for authentication.
REMOTE_HOST	The fully qualified name of the host making the request. If this information is not available, then REMOTE_ADDR can be used to get IR address.
REQUEST_METHOD	The method used to make the request. The most common methods are GET and POST.
SCRIPT_FILENAME	The full path to the CGI script.
SCRIPT_NAME	The name of the CGI script.

Get And Post Methods

- Browser uses two methods two pass this information to web server. These methods are GET Method and POST Method.
- The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? Character like
- http://www.xyz.com/cgi-bin/hello.py?key1=value1&key2=value2
- The GET method is the default method to pass information from browser to web server.
- Never use GET method if you have password or other sensitive information to pass to the server.
- The GET method has size limtation: only 1024 characters can be sent in a request string.
- The GET method sends information using QUERY_STRING header and will be accessible in your CGI Program through QUERY_STRING environment variable.

- **Post method**
  - A generally more reliable method of passing information to a CGI program is the POST method.
  - This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message.
  - This message comes into the CGI script in the form of the standard input.
  -

Cookies

- HTTP protocol is a stateless protocol.
- For a commercial website, it is required to maintain session information among different pages.

- For example, one user registration ends after completing many pages.
- In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

- **How It Works?**
  - **Your server sends some data to the visitor's browser in the form of a cookie.**
  - **The browser may accept the cookie.**
  - If it does, it is stored as a plain text record on the visitor's hard drive.
  - Now, when the visitor arrives at another page on your site, the cookie is available for retrieval.
  - Cookies are a plain text data record of 5 variable-length fields:
  - Expires: The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
  - Domain: The domain name of your site.
  - Path: The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
  - Secure: If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
  - Name=Value: Cookies are set and retrieved in the form of key and value pairs.
  - Setting the cookies is done as :
  - print("Set-Cookie:UserId=xyz;\r\n")
  - print("Set-Cookie:Password=xyzPass;\r\n")
  - print("Set-Cookie:Expires=Saturday,22-Sep-2000 23:12:22 GMT;\r\n")
  - Retrieving the cookies

```
if environ.has_key('HTTP_COOKIE'):
        for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):
                (key, value ) = split(cookie, '=');
                 if key == "UserID":
                        user_id = value
                if key == "Password":
                        password = value
```

File Upload

- To upload a file, the HTML form must have the enctype attribute set to multipart/form-data. The input tag with the file type creates a "Browse" button.

```
<html>
<body>
  <form enctype="multipart/form-data"
            action="save_file.py" method="post">
  <p>File: <input type="file" name="filename" /></p>
  <p><input type="submit" value="Upload" /></p>
  </form>
</body>
</html>
```

Example

MultiThreading

- Thread
  - A Thread or a Thread of Execution is defined in computer science as the smallest unit that can be scheduled in an operating system.
  - Threads are usually contained in processes.
  - More than one thread can exist within the same process.
  - Every process has at least one thread, i.e. the process itself.
  - A process can start multiple threads.
- Starting A Thread
  - There are two modules which support the usage of threads in Python:
  - thread
  - &
  - threading
  - It's possible to execute functions in a separate thread with the module Thread.
  - To do this, we can use the function thread.start_new_thread:
  - thread.start_new_thread(function, args[, kwargs])
  - This method starts a new thread and return its identifier.
  - The method call returns immediately and the child thread starts and calls function with the passed list of *agrs*. When function returns, the thread terminates.
  - Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.

Example

- Threading Module
  - The **threading** module constructs higher-level threading interfaces on top of the lower level **_thread** module.
  - Creating Thread using threading module.
  - Define a new subclass of the *Thread* class.
  - Override the *__init__(self [,args])* method to add additional arguments.
  - Then, override the run(self [,args]) method to implement what the thread should do when started.
  - Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls *run()*method.
  - The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

Method	Description
threading.activeCount():	Returns the number of thread objects that are active.
threading.currentThread():	Returns the number of thread objects in the caller's thread control.
threading.enumerate():	Returns a list of all thread objects that are currently active.

- The threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows:

Method	Description
run():	The run() method is the entry point for a thread.
start():	The start() method starts a thread by calling the run method.
join([time]):	The join() waits for threads to terminate.
isAlive():	The isAlive() method checks whether a thread is still executing.
getName():	The getName() method returns the name of a thread.
setName():	The setName() method sets the name of a thread.

CreatingThread

```
1 import threading
2 def f():
3     print ('Creating  A Thread')
4     return
5 if __name__ == '__main__':
6     for i in range(3):
7         t = threading.Thread(target=f)
8         t.start()
```

Console

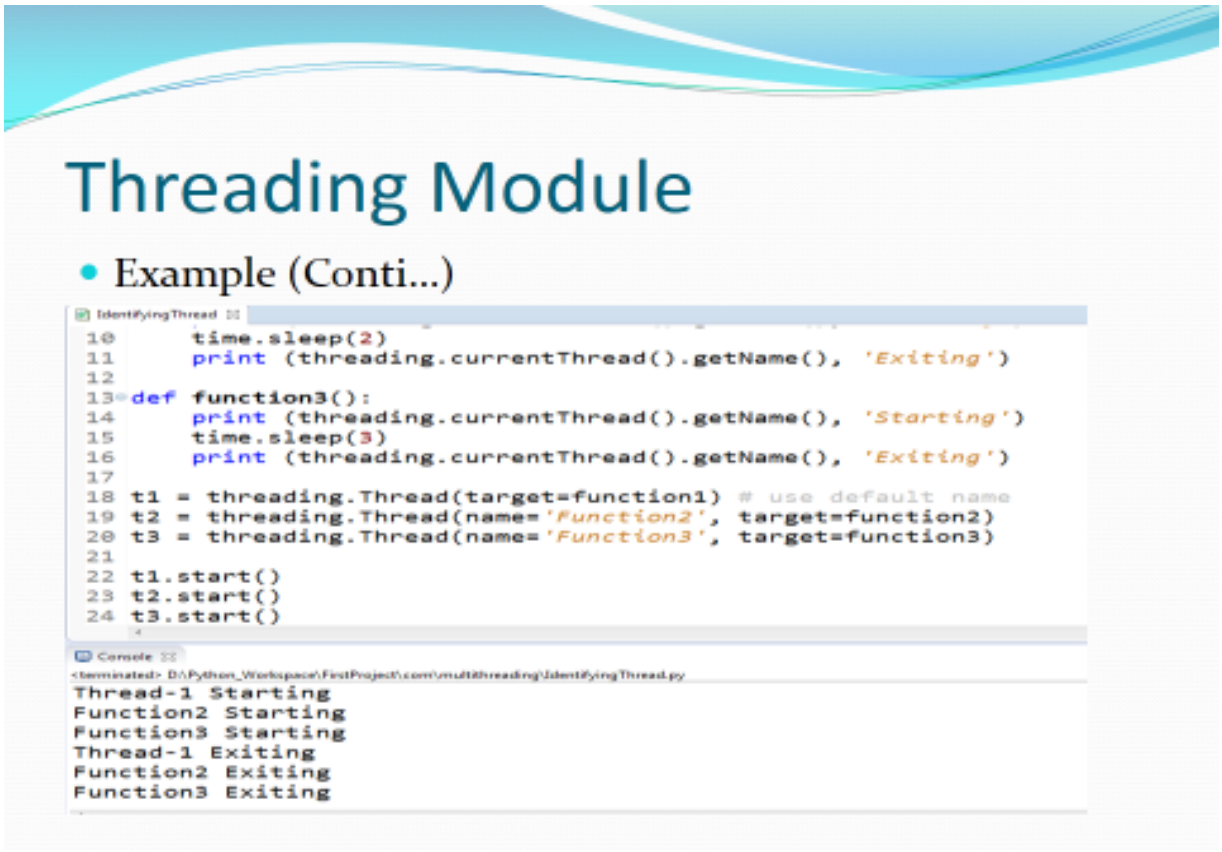
```
<terminated> D:\Python_Workspace\FirstProje
Creating  A Thread
Creating  A Thread
Creating  A Thread
```

IdentifyingThread

```
1 import threading
2 import time
3 def function1():
4     print (threading.currentThread().getName(), 'Starting')
5     time.sleep(1)
6     print (threading.currentThread().getName(), 'Exiting')
7
8 def function2():
9     print (threading.currentThread().getName(), 'Starting')
10    time.sleep(2)
11    print (threading.currentThread().getName(), 'Exiting')
12
13 def function3():
14    print (threading.currentThread().getName(), 'Starting')
15    time.sleep(3)
```

Console

```
<terminated> D:\Python_Workspace\FirstProject\com\multithreading\IdentifyingThread.py
Thread-1 Starting
Function2 Starting
Function3 Starting
Thread-1 Exiting
Function2 Exiting
Function3 Exiting
```



- **Synchronizing Thread**
  - The <*threading*> module has built in functionality to implement locking that allows you to synchronize threads.
  - Locking is required to control access to shared resources to prevent corruption or missed data.
  - You can call *Lock()* method to apply locks, it returns the new lock object.
  - Then, you can invoke the *acquire(blocking)* method of the lock object to enforce threads to run synchronously.
  - The optional *blocking* parameter specifies whether the thread waits to acquire the lock.
  - In case, *blocking* is set to zero, the thread returns immediately with a zero value if the lock can't be acquired and with a 1 if the lock was acquired.
  - In case, *blocking* is set to 1, the thread blocks and wait for the lock to be released.
  - The *release()* method of the lock object is used to release the lock when it is no longer required.

```
import threading
import time
import inspect

class Thread(threading.Thread):
   def __init__(self, t, *args):
      threading.Thread.__init__(self, target=t, args=args)
      self.start()

count = 0
lock = threading.Lock()

def incre():
   global count
   '''Get a list of frame records for a frame and all outer frames.
   These frames represent the calls that lead to the creation of frame.
   The first entry in the returned list represents frame;
   the last entry represents the outermost call on frame's stack.'''
   caller = inspect.getouterframes(inspect.currentframe())[1][3]
   print("Inside %s()" % caller)
   print ("Obtaining lock")
   with lock:
      print ("Lock Obtained")
      count += 1
      time.sleep(2)

def bye():
   while count < 5:
      incre()

def hello_there():
   while count < 5:
      incre()
```

```
Console
<terminated> D:\Python_Workspace\FirstProject\com\multithreading\S
Inside bye()
Inside hello_there()
Obtaining lock
Lock Obtained
Obtaining lock
Inside hello_there()
Obtaining lock
Lock Obtained
Lock Obtained
Inside hello_there()
Obtaining lock
Lock Obtained
Inside bye()
Obtaining lock
Inside hello_there()
Obtaining lock
Lock Obtained
Lock Obtained
```

- **Multithreaded Priority Queue**
  - The *Queue* module allows you to create a new queue object that can hold a specific number of items.

- Sometimes the processing order of the items in a queue needs to be based on characteristics of those items, rather than just the order they are created or added to the queue.

Methods	Description
get():	The get() removes and returns an item from the queue.
qsize() :	The qsize() returns the number of items that are currently in the queue.
put():	The put adds item to a queue.
empty():	The empty( ) returns True if queue is empty; otherwise, False.
full():	The full() returns True if queue is full; otherwise, False.

```
try:
   import Queue as Q  # ver. < 3.0
except ImportError:
   import queue as Q

q = Q.PriorityQueue()
q.put(10)
q.put(1)
q.put(5)
while not q.empty():
   print(q.get())
```

```
import queue as Q
from _functools import cmp_to_key

class Skill(object):
   def __init__(self, priority, description):
      self.priority = priority
      self.description = description
      print ('New Level:', description)
      return
   def __lt__(self, other):
     return ((self.priority < other.priority) and (self.priority < other.priority))

q = Q.PriorityQueue()
q.put(Skill(10, 'Expert'))
q.put(Skill(1, 'Novice'))
q.put(Skill(5, 'Proficient'))


while not q.empty():
   next_level = q.get()
   print( 'Processing level:', next_level.description)
```

Designing

- HTML
- CSS
- JavaScript
- Ajax

Note : This will cover in the Django framework.

Django

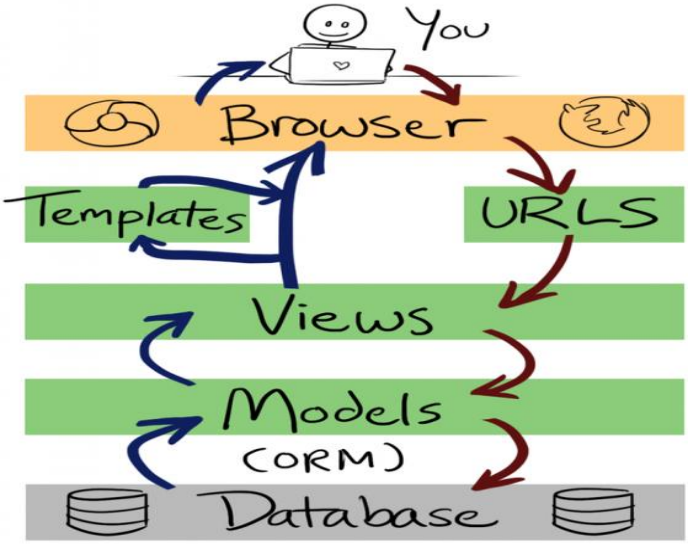
- Virtual Environment
- Installation
- Migration
- Settings.py
- Creating application
- Models

What is Django?

- Django is a free and open source web application framework, written in Python. A web framework is a set of components that helps you to develop websites faster and easier.
- When you're building a website, you always need a similar set of components: a way to handle user authentication (signing up, signing in, signing out), a management panel for your website, forms, a way to upload files, etc.
- To understand what Django is actually for, we need to take a closer look at the servers.
- The first thing is that the server needs to know that you want it to serve you a web page.
- Imagine a mailbox (port) which is monitored for incoming letters (requests). This is done by a web server.
- The web server reads the letter and then sends a response with a webpage.
- But when you want to send something, you need to have some content. And Django is something that helps you create the content.

How It Works?

- When a request comes to a web server, it's passed to Django which tries to figure out what is actually requested.
- It takes a web page address first and tries to figure out what to do. This part is done by Django's urlresolver (note that a website address is called a URL – Uniform Resource Locator – so the name *urlresolver* makes sense).
- It is not very smart – it takes a list of patterns and tries to match the URL. Django checks patterns from top to bottom and if something is matched, then Django passes the request to the associated function (which is called *view*).



Django Installation

- **Virtual environment**
  - Before we install Django we will get you to install an extremely useful tool to help keep your coding environment tidy on your computer.
  - It's possible to skip this step, but it's highly recommended.
  - So, let's create a virtual environment (also called a *virtualenv*). Virtualenv will isolate your Python/Django setup on a per-project basis.
  - This means that any changes you make to one website won't affect any others you're also developing
  - All you need to do is find a directory in which you want to create the virtualenv;
  - For windows ,
  - To create a new virtualenv,you need to open the console and run C:\Python35\python –m venv myvenv.
  - It will look like this:
  - Command-line
  - C:\Users\Name\djangogirls> C:\Python35\python –m venv myvenv
  - where C:\Python35\python is the directory in which python is installed and 'myvenv' is the name of your virtualenv.
  - **Working with virtualenv**
  - The command above will create a directory called 'myvenv' that contains our virtual environment.
  - Start your virtual environment by running :
  - Command line :
  - C:\Users\Name\djangogirls>myvenv\Scripts\activate
  - Now that you have your virtualenv started ,you can install Django.
  - Before we do that, we should make sure we have the latest version of pip, the software that we use to install Django.
  - In the console ,run

    pip install –upgrade pip.

    Then run

    pip install django~=1.9.0

    or

    pip install django
- **Start The Project**
  - The first step is to start a new Django project.
  - Basically, this means that we'll run some scripts provided by Django that will create the skeleton of a Django project for us.
  - This is just a bunch of directories and files that we will use later.
  - On Windows (again, don't forget to add the period (or dot) '.' at the end.
  - Command-line
  - (myvenv)C:\Users\Name\djangogirls>django-admin.py startproject mysite.
  - Here 'django-admin.py' is a script that will create the directories and files for you.
  - You should now have a directory structure which looks like this:

    djangogirls

    |------manage.py

```
|------mysite
        settings.py
    urls.py
        wsgi.py
    __init__.py
```

- Here 'manage.py' is a script that helps with management of the site. With it we will be able (amongst other things) to start a web server on our computer without installing anything else.
- The settings.py file contains the configuration of your website.
- urls.py file contains a list of patterns used by urlresolver.

- **Database Setup**
  - There's a lot of different database software that can store data for your site. We'll use the default one, sqlite3.
  - This is already set up in this part of your mysite/settings.py file.
  - mysite/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'mydatabase',
    }
}
```

- **For Postgres database :**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'todo',
        'USER': 'postgres',
        'PASSWORD': 'admin',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

  - **Now migration is needed for settings configuration.For this we needs to run**
  - python manage.py migrate
  - Starting the web server.
  - (myvenv)~/djangogirls$ python manage.py runserver

- Now all you need to do is check that your website is running. Open your browser (Firefox, Chrome, Safari, Internet Explorer or whatever you use) and enter this address:
- browser
- http://127.0.0.1:8000/

- **Create Application**
  - A model in Django is a special kind of object --it is saved in the database.
  - Model in database as a spreadsheet with columns(fields) and rows (data).
  - Creating an application
  - To create an application we need to run the following command in the console
  - (myvenv)~/djangogirls$ python manage.py startapp blog
  - You will notice that a new 'blog' directory is created and it contains a number of files now.
  - The directories and files in our project should look like this:

```
djangogirls
├── blog
|   ├── __init__.py
|   ├── admin.py
|   ├── apps.py
|   ├── migrations
|   |   └── __init__.py
|   ├── models.py
|   ├── tests.py
|   └── views.py
├── db.sqlite3
├── manage.py
└── mysite
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

  - After creating an application, we also need to tell Django that it should use it. We do that in the file mysite/settings.py.
  - We need to find INSTALLED_APPS and add a line containing 'blog' just above].
  - EG:
  - INSTALLED_APPS = [
  - 'django.contrib.admin',
  - 'django.contrib.auth',
  - 'django.contrib.contenttypes',
  - 'django.contrib.sessions',
  - 'django.contrib.messages',
  - 'django.contrib.staticfiles',
  - 'blog'
  -]

- **Creating Model**
  - Creating a blog post model:
  - In the blog/models.py file we define all objects called Models – this is a place in which we will define our blog post.
  - blog/models.py

- class Post(models.Model):
- …….
- …….
- Create tables for models in your database
- The last step here is to add our new model to our database.
- First we have to make Django know that we have some changes in our model. (We have just created it!)
- Go to your console window and type
- python manage.py makemigrations blog
- Django prepared a migration file for us that we now have to apply to our database. Type
- python manage.py migrate blog

Django Admin

- To add, edit and delete the posts we've just modeled, we will use Django admin.
- Let's open the blog/admin.py file and replace its contents with this:
- blog/admin.py
- from django.contrib import admin
- from .models import Post
- admin.site.register(Post)
- To log in, you need to create a *superuser* - a user account that has control over everything on the site. Go back to the command line, type
- python manage.py createsuperuser and press enter.
- (myvenv) ~/djangogirls$ python manage.py createsuperuser
- Username: admin
- Email address: admin@admin.com
- Password:
- Password (again):
- Superuser created successfully.

GUI Programming

Tkinter Programming

- Tkinter is the standard GUI library for Python.
- Python when combined with Tkinter provides a fast and easy way to create GUI applications.
- Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

- **Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps –**
  - Import the *Tkinter* module.
  - Create the GUI application main window.
  - Add one or more of the above-mentioned widgets to the GUI application.
  - Enter the main event loop to take action against each event triggered by the user.
  - Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.
  - EG :
  - Button,Canvas,Checkbutton,Entry,Frame,Label,Listbox,Menubutton,Menu etc…
- **Some Important Methods**

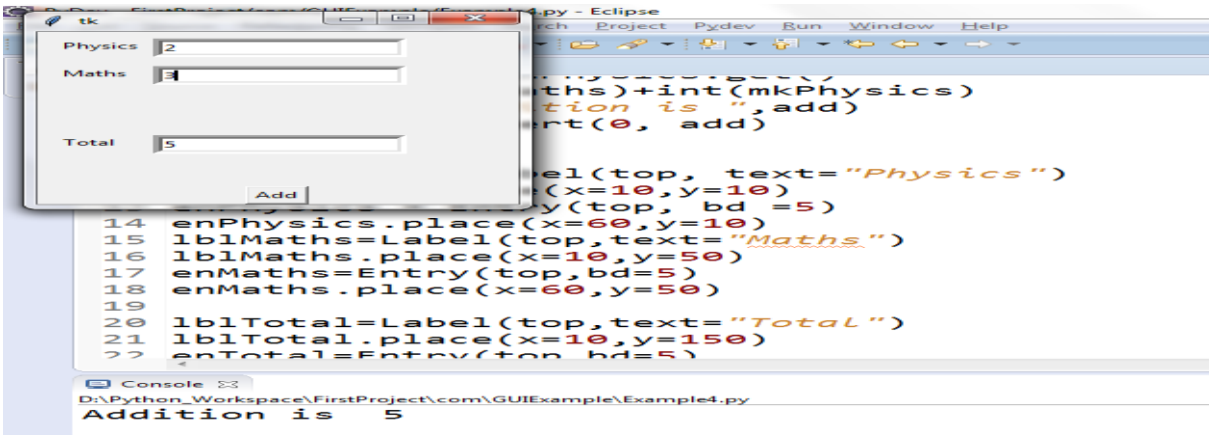
Method	Description
widget.pack(pack_options)	Pack_options can be 'expand','fill', 'side' expand:When set to true, widget expands to fill any space not otherwise used in widget's parent. fill: Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically). side: Determines which side of the parent widget packs against: TOP (default), BOTTOM, LEFT, or RIGHT.

- **Example** bottomframe=Frame(root)
  bottomframe.pack(side=BOTTOM)

Method	Description
widget.place(place_options)	place_options are anchor,bordermode,height,width,relheight,relwidth,relx,rely,x,y anchor : The exact spot of widget other options refer to: may be N, E, S, W, NE, NW, SE, or SW, compass directions indicating the corners and sides of widget; default is NW (the upper left corner of widget) bordermode : INSIDE (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); OUTSIDE otherwise.
widget.place(place_options)	relheight, relwidth : Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget. relx, rely : Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget. x, y : Horizontal and vertical offset in pixels.

```
import queue as Q
from _functools import cmp_to_key

class Skill(object):
   def __init__(self, priority, description):
      self.priority = priority
      self.description = description
      print ('New Level:', description)
      return
   def __lt__(self, other):
     return ((self.priority < other.priority) and (self.priority < other.priority))

q = Q.PriorityQueue()
q.put(Skill(10, 'Expert'))
```

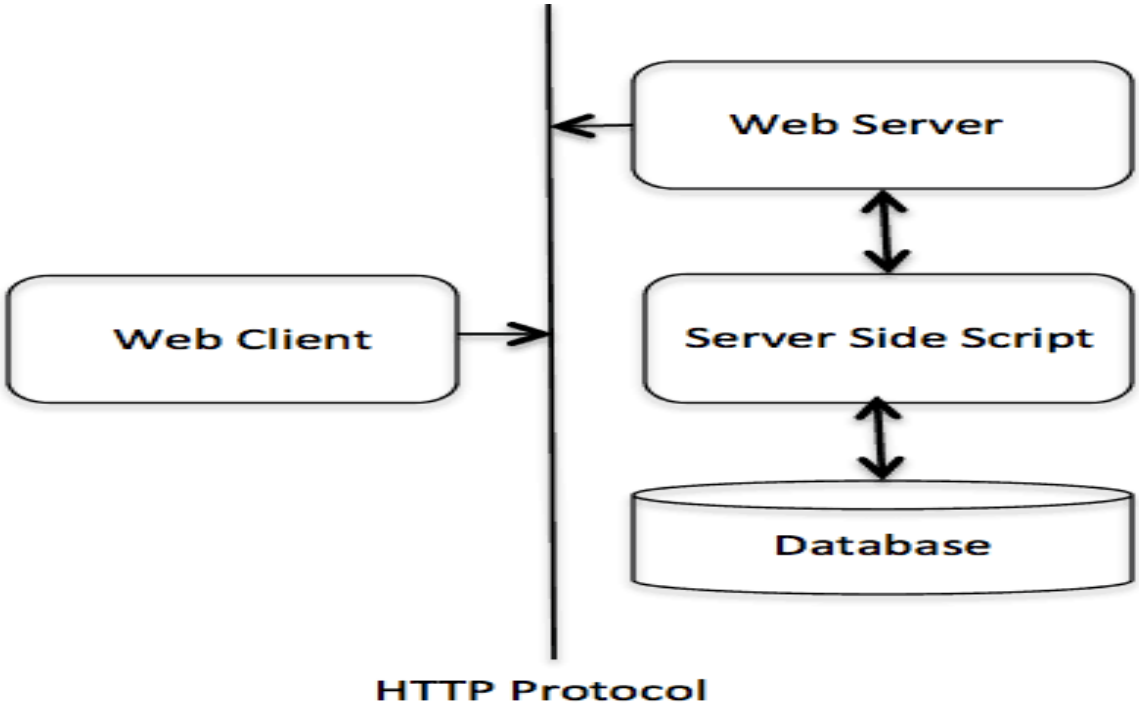
```
from tkinter import *
def addition():
   mkMaths=enMaths.get()
   mkPhysics=enPhysics.get()
   add=int(mkMaths)+int(mkPhysics)
   print ("Addition is ",add)
   enTotal.insert(0, add)
top = Tk()
lblPhysics = Label(top, text="Physics")
lblPhysics.place(x=10,y=10)
enPhysics = Entry(top, bd =5)
enPhysics.place(x=60,y=10)
lblMaths=Label(top,text="Maths")
lblMaths.place(x=10,y=50)
enMaths=Entry(top,bd=5)
enMaths.place(x=60,y=50)

lblTotal=Label(top,text="Total")
lblTotal.place(x=10,y=150)
enTotal=Entry(top,bd=5)
enTotal.place(x=60,y=150)

btnAdd = Button(top, text ="Add",command=addition)
btnAdd.place(x=100, y=100)
btnAdd.pack(side=BOTTOM)
top.geometry("250x250+10+10")
top.mainloop()
```



CGI

- Introduction
- Architecture
- CGI environment variable
- GET and POST methods
- Cookies
- File upload

- **Introduction**
  - Common protocol web servers use to run external programs in response to HTTP requests.
  - Typical uses:
  - forms processing,
  - dynamic content generation
  - Most often, CGI scripts live in the server's special "cgi-bin" directory.
  - The HTTP server places all sorts of information about the request (such as the client's hostname, the requested URL, the query string, and lots of other goodies) in the script's shell environment, executes the script, and sends the script's output back to the client.
  - The script's input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the "query string" part of the URL.
  - The output of a CGI script should consist of two sections, separated by a blank line.
  - The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:
  - print("Content-Type: text/html") # HTML is following
  - print() # blank line, end of headers
  - **The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here's Python code that prints a simple piece of HTML:**
  - **print ("<TITLE>CGI script output</TITLE>")**
  - **print ("<H1>This is my first CGI script</H1>")**
  - **print ("Hello, world!")**
- **Architecture**

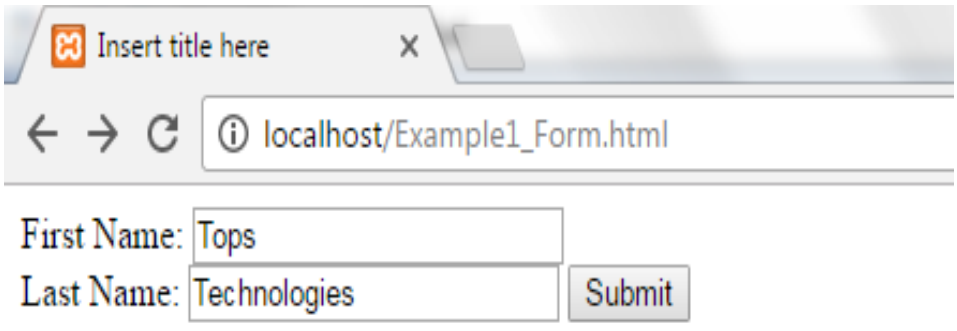


- **CGI environment variable**

Variables	
CONTENT_TYPE	The data type of the content. Used when the client is sending attached content to the server. For example, file upload.
CONTENT_LENGTH	The length of the query information. It is available only for POST requests.
HTTP_COOKIE	Returns the set cookies in the form of key & value pair.
HTTP_USER_AGENT	The User-Agent request-header field contains information about the user agent originating the request. It is name of the web browser.
SERVER_NAME	The server's hostname or IP Address
SERVER_SOFTWARE	The name and version of the software the server is running.
PATH_INFO	The path for the CGI script.
QUERY_STRING	The URL-encoded information that is sent with GET method request.
REMOTE_ADDR	The IP address of the remote host making the request. This is useful logging or for authentication.
REMOTE_HOST	The fully qualified name of the host making the request. If this information is not available, then REMOTE_ADDR can be used to get IR address.
REQUEST_METHOD	The method used to make the request. The most common methods are GET and POST.
SCRIPT_FILENAME	The full path to the CGI script.

SCRIPT_NAME	The name of the CGI script.

- **GET and POST methods**
  - **Browser uses two methods two pass this information to web server. These methods are GET Method and POST Method.**
  - **The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? Character like**
  - **http://www.xyz.com/cgi-bin/hello.py?key1=value1&key2=value2**
  - **The GET method is the default method to pass information from browser to web server.**
  - **Never use GET method if you have password or other sensitive information to pass to the server.**
  - **The GET method has size limtation: only 1024 characters can be sent in a request string.**
  - **The GET method sends information using QUERY_STRING header and will be accessible in your CGI Program through QUERY_STRING environment variable.**
- **Post method**
  - **A generally more reliable method of passing information to a CGI program is the POST method.**
  - **This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message.**
  - **This message comes into the CGI script in the form of the standard input.**
  - **Download xampp**
  - **Edit C:\xampp\apache\conf\httpd.conf file**
    <Directory "C:/xampp/htdocs">
    AddHandler cgi-script .cgi .py
    Options Indexes FollowSymLinks Includes ExecCGI
    Put .py and .html files in "C:\xampp\htdocs" folder.
- **Steps to perform the CGI programs**
  -



- **Example1_Form.html**

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<form action="Example1.py" method="post">
First Name: <input type="text" name="first_name"><br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

- **After submit (Example1.py)**

- **Example**

```
import cgi
# Create instance of FieldStorage
form = cgi.FieldStorage()
# Get data from fields
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')
print("Content-Type: text/html")
print ("<html>")
print ("<head>")
print ("<title>Hello - Second CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2>Hello %s %s</h2>" % (first_name, last_name))
print ("</body>")
print ("</html>")
```

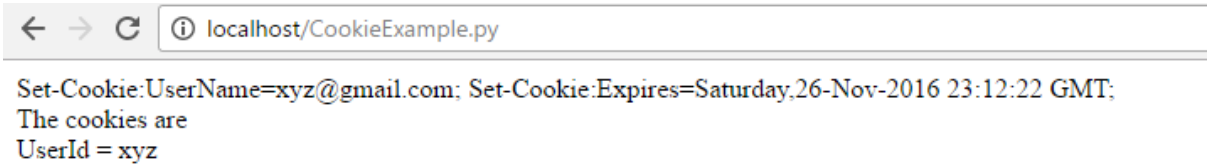
- **Cookies**
  - **HTTP protocol is a stateless protocol.**
  - **For a commercial website, it is required to maintain session information among different pages.**
  - **For example, one user registration ends after completing many pages.**
  - **In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.**

- **How It Works?**
  - **Your server sends some data to the visitor's browser in the form of a cookie.**
  - **The browser may accept the cookie.**
  - **If it does, it is stored as a plain text record on the visitor's hard drive.**
  - **Now, when the visitor arrives at another page on your site, the cookie is available for retrieval.**
- **Cookies are a plain text data record of 5 variable-length fields:**
- **Expires: The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.**
- **Domain: The domain name of your site.**
- **Path: The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.**
- **Secure: If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.**
- **Name=Value: Cookies are set and retrieved in the form of key and value pairs.**

- **Example (SetCookiesExample.py)**

```
#!C:/Python35/python.exe
from os import environ
import cgi, cgitb

print("Content-Type: text/html")
print("Set-Cookie:UserId=xyz;\r\n")
print("Set-Cookie:Expires=Saturday,26-Nov-2016 23:12:22 GMT;\r\n")
handler = {}
if 'HTTP_COOKIE' in environ:
    cookies = environ['HTTP_COOKIE']
    cookies = cookies.split('; ')

    for cookie in cookies:
        cookie = cookie.split('=')
        handler[cookie[0]] = cookie[1]
print("<br> The cookies are <br>")
for k in handler:
    print(k + " = " + handler[k] + "<br>")
```

- **Output**

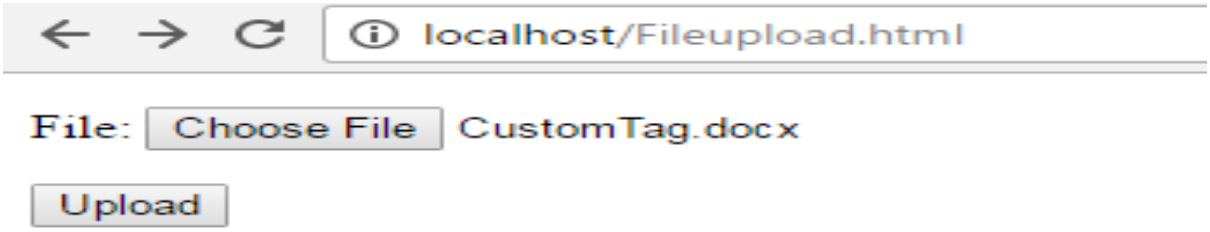


- **File Upload**
  - **To upload a file, the HTML form must have the enctype attribute set to multipart/form-data. The input tag with the file type creates a "Browse" button.**

- **FileUpload.html**

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
 <form enctype="multipart/form-data"
                      action="save_file.py" method="post">
   <p>File: <input type="file" name="filename" /></p>
   <p><input type="submit" value="Upload" /></p>
   </form>
</body>
</html>
```

- **Examples**



- **save._file.py**

```
#!C:/Python35/python.exe
import cgi, os
import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# Get filename here.
fileitem = form['filename']

# Test if the file was uploaded
if fileitem.filename:
   # strip leading path from file name to avoid
   # directory traversal attacks
   fn = os.path.basename(fileitem.filename)
   open('D:/New folder/' + fn, 'wb').write(fileitem.file.read())

   message = 'The file "' + fn + '" was uploaded successfully'

else:
   message = 'No file was uploaded'

print ("""\
Content-Type: text/html\n
<html>
<body>
   <p>%s</p>
</body>
</html>
""" % (message,))
```

- **Multithreading**
  - **Thread**
  - **Starting a thread**
  - **Threading module**

- Synchronizing threads
- Multithreaded Priority Queue

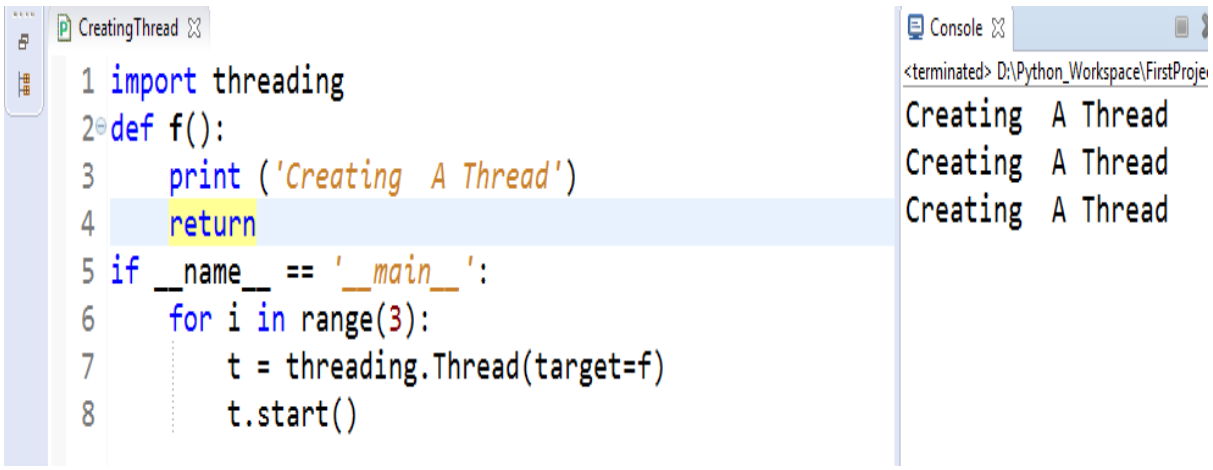
- Thread
  - A Thread or a Thread of Execution is defined in computer science as the smallest unit that can be scheduled in an operating system.
  - Threads are usually contained in processes.
  - More than one thread can exist within the same process.
  - Every process has at least one thread, i.e. the process itself.
  - A process can start multiple threads.
- Starting Thread
  - There are two modules which support the usage of threads in Python:
  - thread
  - &
  - threading
  - It's possible to execute functions in a separate thread with the module Thread.
  - To do this, we can use the function thread.start_new_thread:
  - thread.start_new_thread(function, args[, kwargs])
  - This method starts a new thread and return its identifier.
  - The method call returns immediately and the child thread starts and calls function with the passed list of *agrs*. When function returns, the thread terminates.
  - Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.
- Threading module
  - The threading module constructs higher-level threading interfaces on top of the lower level _thread module.
  - Creating Thread using threading module.
  - Define a new subclass of the *Thread* class.
  - Override the *__init__(self [,args])* method to add additional arguments.
  - Then, override the run(self [,args]) method to implement what the thread should do when started.
  - Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls *run()*method.
- Threading Module
  - The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

Method	Description
threading.activeCount():	**Returns the number of thread objects that are active.**
threading.currentThread():	**Returns the number of thread objects in the caller's thread control.**
threading.enumerate():	**Returns a list of all thread objects that**

	are currently active.

Method	Description
run():	The run() method is the entry point for a thread.
start():	The start() method starts a thread by calling the run method.
join([time]):	The join() waits for threads to terminate.
isAlive():	The isAlive() method checks whether a thread is still executing.
getName():	The getName() method returns the name of a thread.
setName():	The setName() method sets the name of a thread.

- **Example**



- **Example**

IdentifyingThread

```
1  import threading
2  import time
3  def function1():
4      print (threading.currentThread().getName(), 'Starting')
5      time.sleep(1)
6      print (threading.currentThread().getName(), 'Exiting')
7
8  def function2():
9      print (threading.currentThread().getName(), 'Starting')
10     time.sleep(2)
11     print (threading.currentThread().getName(), 'Exiting')
12
13 def function3():
14     print (threading.currentThread().getName(), 'Starting')
15     time.sleep(3)
16     print (threading.currentThread().getName(), 'Exiting')
17
18 t1 = threading.Thread(target=function1) # use default name
19 t2 = threading.Thread(name='Function2', target=function2)
20 t3 = threading.Thread(name='Function3', target=function3)
21
22 t1.start()
23 t2.start()
24 t3.start()
```

Console

<terminated> D:\Python_Workspace\FirstProject\com\multithreading\IdentifyingThread.py

```
Thread-1 Starting
Function2 Starting
Function3 Starting
Thread-1 Exiting
Function2 Exiting
Function3 Exiting
```

- **Synchronizing threads**
  - The <*threading*> module has built in functionality to implement locking that allows you to synchronize threads.
  - Locking is required to control access to shared resources to prevent corruption or missed data.
  - You can call *Lock()* method to apply locks, it returns the new lock object.
  - Then, you can invoke the *acquire(blocking)* method of the lock object to enforce threads to run synchronously.
  - The optional *blocking* parameter specifies whether the thread waits to acquire the lock.
  - In case, *blocking* is set to zero, the thread returns immediately with a zero value if the lock can't be acquired and with a 1 if the lock was acquired.
  - In case, *blocking* is set to 1, the thread blocks and wait for the lock to be released.
  - The *release()* method of the lock object is used to release the lock when it is no longer required.
- **Example**

TimerObject SynchonizationExample

```
1  import threading
2  import time
3  import inspect
4
5  class Thread(threading.Thread):
6      def __init__(self, t, *args):
7          threading.Thread.__init__(self, target=t, args=args)
8          self.start()
9
10 count = 0
11 lock = threading.Lock()
12
13 def incre():
14     global count
15     '''Get a list of frame records for a frame and all outer frames.
16     These frames represent the calls that lead to the creation of frame.
17     The first entry in the returned list represents frame;
```

TimerObject SynchonizationExample

```
18     the last entry represents the outermost call on frame's stack.'''
19     caller = inspect.getouterframes(inspect.currentframe())[1][3]
20     print("Inside %s()" % caller)
21     print ("Obtaining lock")
22     with lock:
23         print ("Lock Obtained")
24         count += 1
25         time.sleep(2)
26
27 def bye():
28     while count < 5:
29         incre()
30
31 def hello_there():
32     while count < 5:
33         incre()
34
35 def main():
36     hello = Thread(hello_there)
37     goodbye = Thread(bye)
38
39
40 if __name__ == '__main__':
41     main()
```

Console

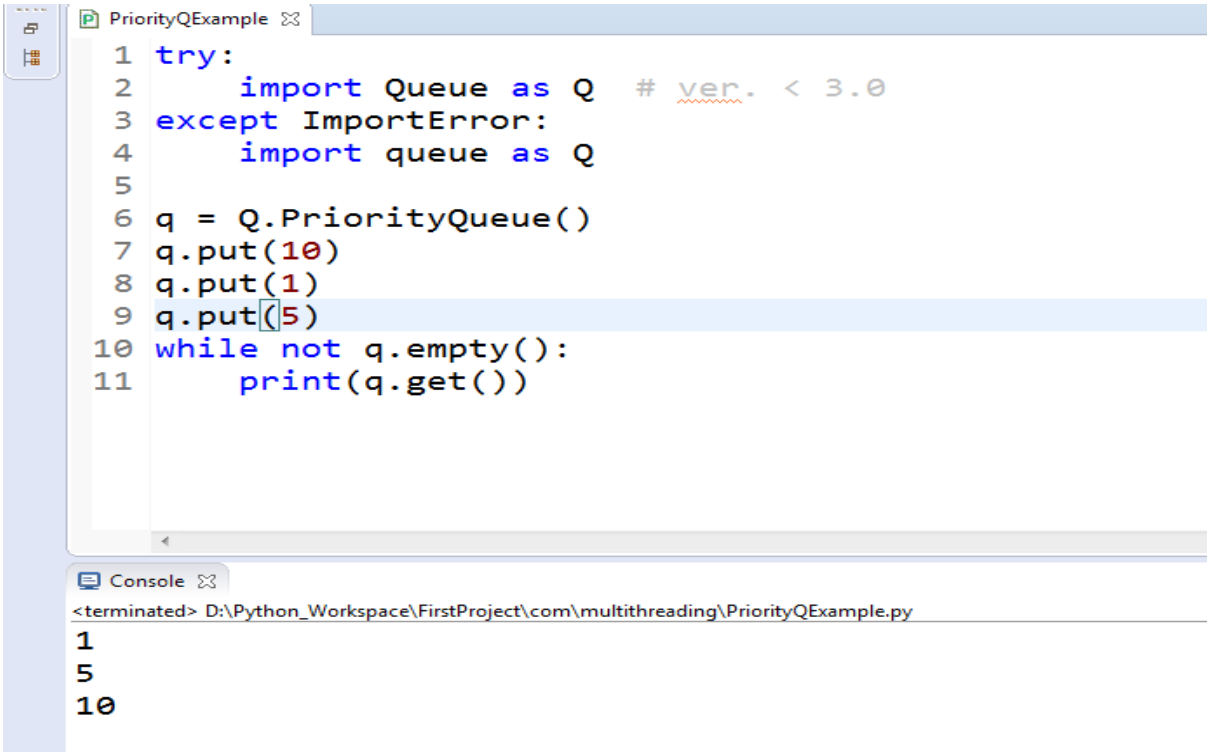
<terminated> D:\Python_Workspace\FirstProject\com\multithreading\S

```
Inside bye()
Inside hello_there()
Obtaining lock
Lock Obtained
Obtaining lock
Inside hello_there()
Obtaining lock
Lock Obtained
Lock Obtained
Inside hello_there()
Obtaining lock
Lock Obtained
Inside bye()
Obtaining lock
Inside hello_there()
Obtaining lock
Lock Obtained
Lock Obtained
```

- **Multithreaded Priority Queue**
  - **The *Queue* module allows you to create a new queue object that can hold a specific number of items.**
  - **Sometimes the processing order of the items in a queue needs to be based on characteristics of those items, rather than just the order they are created or added to the queue.**

Methods	Description
get():	The get() removes and returns an item from the queue.
qsize() :	The qsize() returns the number of items that are currently in the queue.
put():	The put adds item to a queue.
empty():	The empty( ) returns True if queue is empty; otherwise, False.
full():	The full() returns True if queue is full; otherwise, False.

- **Example**



- **Multithreaded Priority Queue**

```
import queue as Q
from _functools import cmp_to_key

class Skill(object):
    def __init__(self, priority, description):
        self.priority = priority
        self.description = description
        print ('New Level:', description)
        return
    def __lt__(self, other):
        return ((self.priority < other.priority) and (self.priority < other.priority))

q = Q.PriorityQueue()
q.put(Skill(10, 'Expert'))
q.put(Skill(1, 'Novice'))
q.put(Skill(5, 'Proficient'))


while not q.empty():
    next_level = q.get()
    print( 'Processing level:', next_level.description)
```