Chapter - 1

Getting started with python

What is Python

Python is a general purpose programming language created by Guido Van Rossum. Python is most praised for its elegant syntax and readable code, if you are just beginning your programming career python suits you best. With python you can do everything from GUI development, Web application, System administration tasks, Financial calculation, Data Analysis, Visualization and list goes on.

Python is interpreted language

Yes, python is interpreted language, when you run python program an interpreter will parse python program line by line basis, as compared to compiled languages like C or C++, where compiler first compiles the program and then start running.

Now you may ask, so what's the difference ??

Difference is that interpreted languages are little bit slow as compared to compiled languages. Yes, you will definitely get some performance benefits if you write your code in compiled languages like C or C++.

But writing codes in such languages is a daunting task for beginner. Also in such languages you need to write even most basic functions like calculate the length of the array, split the string etc. For more advanced tasks sometimes you need to create your own data structures to encapsulate data in the program. So in C/C++ before you actually start solving your business problem you need to take care of all minor details. This is where python comes, in python you don't need to define any data structure, no need to define small utility functions because python has everything to get you started.

Moreover python has hundreds of libraries available at https://pypi.python.org/ which you can use in your project without reinventing the wheel.

Python is Dynamically Typed

In python you don't need to define variable data type ahead of time, python automatically guesses the data type of the variable based on the type of value it contains. For e.g

```
myvar = "Hello Python"
```

In the above line `"Hello Python"` is assigned to `myvar` , so the type of `myvar` is `string`.

Note that in python you do not need to end a statement with a semicolon (;) .

Suppose little bit later in the program we assign `myvar` a value of `1` i.e

```
myvar = 1
```

now `myvar` is of type `int`.

Python is strongly typed

If you have programmed in php or javascript. You may have noticed that they both convert data of one data type to other data type automatically.

For example in JavaScript

```
1 + "2"
```

will be `"12"`, here `1` will be converted to string and concatenated to `"2"` , which results in `"12"`

, which is a `string`. In Python automatic conversions are not allowed, so:

```
1 + "2"
```

will produce an error.

Write less code and do more

Python codes are usually 1/3 or 1/5 of the java code. It means we can write less code in Python to achieve the same thing as in Java.

In python to read a file you only need 2 lines:

```
with open("myfile.txt") as f:
    print(f.read())
```

Who uses python

Python is used by many large organization like Google, NASA, Quora, HortonWorks and many others.

Okay what i can start building in python ?

Pretty much anything you want. For e.g

- GUI application.
- Create Websites.
- Scrape data from website.
- Analyse Data.
- System Administration Task.
- Game Development.

and many more ...

In the next chapter we will learn how to Install python.

Chapter – 2

Installing Python3

This tutorial focuses on Python 3. Most Linux distribution for e.g Ubuntu 14.04 comes with python 2 and 3 installed, here is the download link. If you are using some other linux distribution see this link for installation instructions. Mac also comes with python 2 and python 3 installed (if not see this link for instructions), but this is not the case with windows.

Note: Throughout this tutorial i will give instructions wherever necessary for Windows and Ubuntu 14.04 only.

Installing Python 3 in Windows

To install python you need to download python binary from https://www.python.org/downloads/, specifically we will be using python 3.4.3 which you can download from here . While installing remember to check "Add Python.exe to path" (see the image below).

Now you have installed python, open command prompt or terminal and type python . Now you are in python shell.

```
C:\Users\THEPYTHONGURU>python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To test everything is working fine type the following command in the python shell.

```
print("Hello World")
```

```
C:\Users\THEPYTHONGURU>python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> print("Hello World")
Hello World
>>>
```

Expected output:

```
Hello World
```

If you are using Ubuntu 14.04 which already comes with python 2 and python 3, you need to enter `python3` instead of just `python` to enter python 3 shell.

```
tim@MyUbuntu: ~
tim@MyUbuntu:~$ python3
Python 3.4.0 (default, Jun 19 2015, 14:18:46)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Installing Text Editor

To write python programs you will need a text editor, you can use text editor like notepad. If you want to use full-fledged text editor then use notepad++ or sublime text. Download and install text editor of you choice.

Now you have successfully installed python 3 and text editor and ready to move on to the next chapter, where we will learn different ways of running python programs.

Chapter – 3

Running python programs

You can run python programs in two ways, first by typing commands directly in python shell or run program stored in a file. But most of the time you want to run programs stored in a file.

Lets create a file named hello.py in your documents directory i.e `C:\Users\YourUserName\Documents` using notepad (or any other text editor of your choice) , remember python files have `'.py'` extension, then write the following code in the file.

```
print("Hello World")
```

In python we use print function to display string to the console. It can accept more than one arguments. When two or more arguments are passed, print function displays each argument separated by space.

```
print("Hello", "World")
```

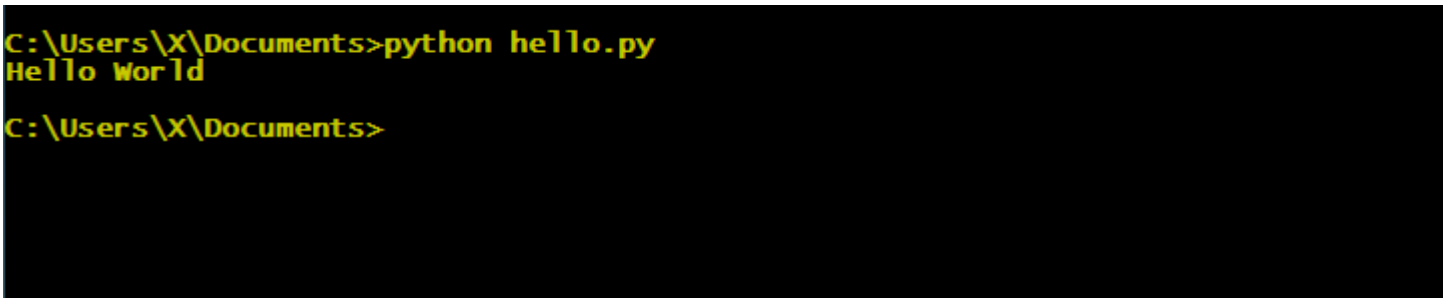
Expected output

```
Hello World
```

Now open terminal and change current working directory to `C:\Users\YourUserName\Documents` using `cd` command.

To run the program type the following command.

```
python hello.py
```



If everything goes well, you will get the following output.

```
Hello World
```

Getting Help

Sooner or later while using python you will come across a situation when you want to know more about some method or functions. To help you Python has `help()` function, here is how to use it.

Syntax:

To find information about class: `help(class_name)`

To find more about method belong to class: `help(class_name.method_name)`

Suppose you want to know more about int class, go to Python shell and type the following command.

```
>>> help(int)
```

Help on class int in module builtins:

```
class int(object)
| int(x=0) -> integer
| int(x, base=10) -> integer
|
| Convert a number or string to an integer, or return 0 if no arguments
| are given. If x is a number, return x.__int__(). For floating point
| numbers, this truncates towards zero.
|
| If x is not a number or if base is given, then x must be a string,
| bytes, or bytearray instance representing an integer literal in the
```

```
| given base. The literal can be preceded by '+' or '-' and be surrounded
| by whitespace. The base defaults to 10. Valid bases are 0 and 2-36.
| Base 0 means to interpret the base from the string as an integer literal.
| >>> int('0b100', base=0)
| 4
|
| Methods defined here:
|
| __abs__(self, /)
| abs(self)
|
| __add__(self, value, /)
| Return self+value.
```

as you can see `help()` function spits out entire `int` class with all the methods, it also contains description where needed.

Now suppose you want to know arguments required for `index()` method of `str` class, to find out you need to type the following command in the python shell.

```
>>> help(str.index)
```

Help on method_descriptor:

```
index(...)
S.index(sub[, start[, end]]) -> int
Like S.find() but raise ValueError when the substring is not found.
```

In the next chapter we will learn about data types and variables in python.

Chapter – 4

DataType and Variables

Variables are named locations which are used to store references to the object stored in memory. The names we choose for variables and functions are commonly known as **Identifiers**. In python Identifiers must obey the following rules.

1. All identifiers must start with letter or underscore (_) , you can't use digits. For e.g `my_var` is valid identifier while `1digit` is not.
2. Identifiers can contain letters, digits and underscores (_).
3. They can be of any length.
4. Identifier can't be a keyword (keywords are reserved words that Python uses for special purpose).Following are Keywords in python 3.

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
pass       else       import     assert
break      except     in         raise
```

Assigning Values to Variables

Values are basic things that programs works with. For e.g `1` , `11` , `3.14` , `"hello"` are all values. In programming terminology they are also commonly known as literals. Literals can be of different type for e.g `1`, `11` are of type `int` , 3.14 is `float` and `"hello"` is string . **Remember in python everything is object even basic data types like int, float, string, we will elaborate more on this in later chapters**.

In python you don't need to declare types of variable ahead of time. Interpreter automatically detects the type of the variable by the data it contains. To assign value to a variable equal sign (`=`) is used. `=` is also known as assignment operator.

Following are some examples of variable declaration:

```
x = 100                         # x is integer
pi = 3.14                       # pi is float
empname = "python is great"     # empname is string
a = b = c = 100                 # this statement assign 100 to c, b and a.
```

Note: In the above code `x` stores reference to the `100` (which is an `int` object) , `x` don't store `100` itself.

In Python comments are preceded by a pound sign (#). Comments are not programming statements that python interpreter executes while running the program. Comments are used by programmers to remind themselves how the program works. They are also used to write program documentation.

```
#display hello world
print("hello world")
```

Simultaneous Assignments

Python allow simultaneous assignment syntax like this:

```
var1, var2, ..., varn = exp1, exp2, ..., expn
```

this statements tells the python to evaluate all the expression on the right and assign them to the corresponding variables on the left. Simultaneous Assignments is helpful to swap values of two variables. For e.g

```
>>> x = 1
>>> y = 2


>>> y, x = x, y # assign y value to x and x value to y
```

Expected Output:

```
>>> print(x)
2
>>> print(y)
1
```

Python Data Types

Python has 5 standard data types namely.

a) Numbers
b) String
c) List
d) Tuple
e) Dictionary
f) Boolean - In Python True and False are boolean literals. But the following values are also considered as false.

- 0 - zero , 0.0 ,
- [] - empty list , () - empty tuple , {} - empty dictionary , ''
- None

Receiving input from Console

`input()` function is used to receive input from the console.

Syntax: `input([prompt]) -> string`

`input()` function accepts an optional string argument called prompt and returns a string.

```
>>> name = input("Enter your name: ")
>>> Enter your name: tim
>>> name
'tim'
```

Note that `input()` returns string even if you enter a number, to convert it to an integer you can use `int()` or `eval()` .

```
>> age = int(input("Enter your age: "))
Enter your age: 22
>>> age
22
>>> type(age)
<class 'int'>
```

Importing modules

Python organizes codes using module. Python comes with many in built modules ready to use for e.g there is a `math` module for mathematical related functions, `re` module for regular expression and so on. But before you can use them you need to import them using the following syntax:

```
import module_name
```

You can also import multiple module using the following syntax:

```
import module_name_1, module_name_2
```

here is an example

```
>>> import math
>>> math.pi
3.141592653589793
```

First line import all functions, classes, variables, constant in the math module. To access anything inside math module we first need to write module name followed by (.) and then name of class, function, constant or variable. In the above example we are accessing a constant called `pi` in `math` module

In next chapter we will cover numbers in python.

Chapter – 5

Python numbers

This data type supports only numerical values like `1` , `31.4` .

Python 3 support 3 different numerical types.

1. `int` - for integer values like `45` .
2. `float` - for floating point values like `2.3` .
3. `complex` - for complex numbers like `3+2j` .

Integers

Integer literals in python belong to `int` class.

```
>>> i = 100
>>> i
100
```

Floats

Floating points are values with decimal point like.

```
>>> f = 12.3
>>> f
12.3
```

One point to note that when one of the operands for numeric operators is a float value then the result will be in float value.

```
>>> 3 * 1.5
4.5
```

Complex number

As you may now complex number consists of two parts real and imaginary, and is denoted by `j` .You can define complex number like this:

```
>>> x = 2 + 3j # where 2 is the real part and 3 is imaginary
```

Determining types

Python has `type()` inbuilt function which is use to determine the type of the variable.

```
>>> x = 12
>>> type(x)
<class 'int'>
```

Python operators

Python has the different operators which allows you to carry out required calculations in your program.

Name	*Meaning*	*Example*	*Result*
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Float Division	1 / 2	0.5
//	Integer Division	1 // 2	0
**	Exponentiation	4 ** 0.5	2.0
%	Remainder	20 % 3	2

`+` , `-` and `*` works as expected, remaining operators require some explanation.

`/` **- Float Division** : `/` operator divides and return result as floating point number means it will always return fractional part. For e.g

```
>>> 3/2
1.5
```

`//` **- Integer Division** : `//` perform integer division i.e it will truncate the decimal part of the answer and return only integer.

```
>>> 3//2
1
```

** **- Exponentiation Operator** : This operator helps to compute a^b (a raise to the power of b). Let's take an example:

```
>>> 2 ** 3 # is same as 2 * 2 * 2
8
```

% **operator** : % operator also known as remainder or modulus operator. This operator return remainder after division. For e.g:

```
>>> 7 % 2
1
```

Operator Precedence

In python every expression are evaluated using operator precedence. Let's take an example to make it clear.

```
>>> 3 * 4 + 1
```

In the above expression which operation will be evaluated first addition or multiplication? To answer such question we need to refer to operator precedence list in python. Image below list python precedence order from high to low.

Operator	Description
`()`	Parentheses (grouping)
`f(args...)`	Function call
`x[index:index]`	Slicing
`x[index]`	Subscription
`x.attribute`	Attribute reference
`**`	Exponentiation
`~x`	Bitwise not
`+x, -x`	Positive, negative
`*, /, %`	Multiplication, division, remainder

+, -	Addition, subtraction
<<, >>	Bitwise shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
in, not in, is, is not, <, <=, >, >=, <>, !=, ==	Comparisons, membership, identity
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
lambda	Lambda expression

as you can see in table above * is above + , so * will occur first then addition. Therefore the result of the above expression will be 13 .

```
>>> 3 * 4 + 1
>>> 13
```

Let's,take one more example to illustrate one more concept.

```
>>> 3 + 4 - 2
```

In above expression which will occur first addition or subtraction. As we can see from the table + and - have same precedence, then they will be evaluated from left to right, i.e addition will be applied first then subtraction.

```
>>> 3 + 4 - 2
>>> 5
```

The only exception to this rule is assignment operator (=) which occur from right to left.

```
a = b = c
```

You can change precedence by using parentheses ( ) , For e.g

```
>>> 3 * (4 + 1)
>>> 15
```

As you can see from the precedence table ( ) has highest priority so in expression 3 * (4 + 1) , (4 + 1) is evaluated first then multiplication. Hence you can use ( ) to alter order of precedence.

Augmented Assignment Operator

These operator allows you write shortcut assignment statements. For e.g:

```
>>> count = 1
>>> count = count + 1
>>> count
2
```

by using Augmented Assignment Operator we can write it as:

```
>>> count = 1
>>> count += 1
>>> count
2
```

similarly you can use `-` , `%` , `//` , `/` , `*` , `**` with assignment operator to form augmented assignment operator.

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Float division assignment	`i /= 8`	`i = i / 8`
`//=`	Integer division assignment	`i //= 8`	`i = i // 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`
`**=`	Exponent assignment	`i **= 8`	`i = i ** 8`

In the next chapter we will learn about python strings.

Chapter – 6

Python Strings

Strings in python are contiguous series of characters delimited by single or double quotes. Python don't have any separate data type for characters so they are represented as a single character string.

Creating strings

```
>>> name = "tom" # a string
>>> mychar = 'a' # a character
```

you can also use the following syntax to create strings.

```
>>> name1 = str() # this will create empty string object
>>> name2 = str("newstring") # string object containing 'newstring'
```

Strings in python are immutable.

What this means to you is that once string is created it can't be modified. Let's take an example to illustrate this point.

```
>>> str1 = "welcome"
>>> str2 = "welcome"
```

here `str1` and `str2` refers to the same string object `"welcome"` which is stored somewhere in memory. You can test whether `str1` refers to same object as `str2` using `id()` function.

What is id() : Every object in python is stored somewhere in memory. We can use `id()` to get that memory address.

```
>>> id(str1)
78965411

>>> id(str2)
78965411
```

As both `str1` and `str2` points to same memory location, hence they both points to the same object.

Let's try to modify `str1` object by adding new string to it.

```
>>> str1 += " mike"
>>> str1
welcome mike
>>> id(str1)
>>> 78965579
```

As you can see now `str1` points to totally different memory location, this proves the point that concatenation doesn't modify original string object instead it creates a new string object. Similarly Number (i.e `int` type) is also immutable.

Operations on string

String index starts from `0` , so to access the first character in the string type:

```
>>> name[0] #
t
```

`+` operator is used to concatenate string and `*` operator is a repetition operator for string.

```
>>> s = "tom and " + "jerry"
>>> print(s)
```

```
tom and jerry
>>> s = "this is bad spam " * 3
>>> print(s)
this is bad spam this is bad spam this is bad spam
```

Slicing string

You can take subset of string from original string by using `[]` operator also known as slicing operator.

Syntax: `s[start:end]`

this will return part of the string starting from index `start` to index `end - 1` .

Let's take some examples.

```
>>> s = "Welcome"
>>> s[1:3]
el
```

Some more examples.

```
>>> s = "Welcome"
>>> s[ : 6]
'Welcom'
```

```
>>> s[4 : ]
'ome'
```

```
>>> s[1 : -1]
'elcom'
```

Note: `start` index and `end` index are optional. If omitted then the default value of `start` index is `0` and that of `end` is the last index of the string.

ord() and chr() Functions

`ord()` - function returns the ASCII code of the character.

`chr()` - function returns character represented by a ASCII number.

```
>>> ch = 'b'
>>> ord(ch)
98
>>> chr(97)
'a'
>>> ord('A')
65
```

String Functions in Python

FUNCTION NAME	FUNCTION DESCRIPTION
len()	returns length of the string
max()	returns character having highest ASCII value
min()	returns character having lowest ASCII value

```
>>> len("hello")
5

>>> max("abc")
'c'

>>> min("abc")
'a'
```

`in` and `not in` operators

You can use `in` and `not in` operators to check existence of string in another string. They are also known as membership operator.

```
>>> s1 = "Welcome"
>>> "come" in s1
True
>>> "come" not in s1
False
>>>
```

String comparison

You can use (`>` , `<` , `<=` , `<=` , `==` , `!=`) to compare two strings. Python compares string lexicographically i.e using ASCII value of the characters.

Suppose you have `str1` as `"Jane"` and `str2` as `"Jake"` . The first two characters from `str1` and `str2` (`J` and `J`) are compared. As they are equal, the second two characters are compared. Because they are also equal, the third two characters (`n` and `k`) are compared. And because `'n'` has greater ASCII value than `'k'` , `str1` is greater than `str2` .

Here are some more examples:

```
>>> "tim" == "tie"
False
>>> "free" != "freedom"
True
>>> "arrow" > "aron"
True
>>> "green" >= "glow"
True
>>> "green" < "glow"
False
>>> "green" <= "glow"
False
>>> "ab" <= "abc"
True
>>>
```

Iterating string using for loop

String is a sequence type and also iterable using for loop (to learn more about for loop click here).

```
>>> s = "hello"
>>> for i in s:
... print(i, end="")
hello
```

Note: By default `print()` function prints string with a newline , we change this behavior by supplying a second argument to it as follows.

```
print("my string", end="\n")  #this is default behavior
print("my string", end="")    # print string without a newline
print("my string", end="foo")  # now print() will print foo after every
string
```

Testing strings

String class in python has various inbuilt methods which allows to check for different types of strings.

METHOD NAME	METHOD DESCRIPTION
isalnum()	Returns True if string is alphanumeric
isalpha()	Returns True if string contains only alphabets
isdigit()	Returns True if string contains only digits
isidentifier()	Return True is string is valid identifier
islower()	Returns True if string is in lowercase
isupper()	Returns True if string is in uppercase
isspace()	Returns True if string contains only whitespace

```
>>> s = "welcome to python"
>>> s.isalnum()
False
>>> "Welcome".isalpha()
True
>>> "2012".isdigit()
True
>>> "first Number".isidentifier()
False
>>> s.islower()
True
>>> "WELCOME".isupper()
True
>>> "  \t".isspace()
True
```

Searching for Substrings

METHOD NAME	METHODS DESCRIPTION:
endswith(s1: str): bool	Returns True if strings ends with substring s1
startswith(s1: str): bool	Returns True if strings starts with substring s1
count(substring): int	Returns number of occurrences of substring the string
find(s1): int	Returns lowest index from where s1 starts in the string, if string not found returns -1
rfind(s1): int	Returns highest index from where s1 starts in the string, if string not found returns -1

```
>>> s = "welcome to python"
>>> s.endswith("thon")
True
>>> s.startswith("good")
False
>>> s.find("come")
```

```
3
>>> s.find("become")
-1
>>> s.rfind("o")
15
>>> s.count("o")
3
>>>
```

Converting Strings

METHOD NAME	METHOD DESCRIPTION
capitalize(): str	Returns a copy of this string with only the first character capitalized.
lower(): str	Return string by converting every character to lowercase
upper(): str	Return string by converting every character to uppercase
title(): str	This function return string by capitalizing first letter of every word in the string
swapcase(): str	Return a string in which the lowercase letter is converted to uppercase and uppercase to lowercase
replace(old, new): str	This function returns new string by replacing the occurrence of old string with new string

```
s = "string in python"
>>> s1 = s.capitalize()
>>> s1
'String in python'
>>> s2 = s.title()
>>> s2
'String In Python'
>>> s = "This Is Test"
```

```
>>> s3 = s.lower()
>>> s3
'this is test'
>>> s4 = s.upper()
>>> s4
'THIS IS TEST'
>>> s5 = s.swapcase()
>>> s5
'tHIS iS tEST'
>>> s6 = s.replace("Is", "Was")
>>> s6
'This Was Test'
>>> s
'This Is Test'
>>>
```

In next chapter we will learn about python lists.

Chapter – 7

Python Lists

List type is another sequence type defined by the list class of python. List allows you add, delete or process elements in very simple ways. List is very similar to arrays.

Creating list in python

You can create list using the following syntax.

```
>>> l = [1, 2, 3, 4]
```

here each elements in the list is separated by comma and enclosed by a pair of square brackets ([]). Elements in the list can be of same type or different type. For e.g:

```
l2 = ["this is a string", 12]
```

Other ways of creating list.

```
list1 = list() # Create an empty list
list2 = list([22, 31, 61]) # Create a list with elements 22, 31, 61
```

```
list3 = list(["tom", "jerry", "spyke"]) # Create a list with strings
list5 = list("python") # Create a list with characters p, y, t, h, o, n
```

Note: Lists are mutable.

Accessing elements in list

You can use index operator ([ ]) to access individual elements in the list. List index starts from 0.

```
>>> l = [1,2,3,4,5]
>>> l[1] # access second element in the list
2
>>> l[0] # access first element in the list
1
```

List Common Operations

METHOD NAME	DESCRIPTION
`x in s`	True if element x is in sequence s.
`x not in s`	if element x is not in sequence s.
`s1 + s2`	Concatenates two sequences s1 and s2
`s * n , n * s`	n copies of sequence s concatenated
`s[i]`	ith element in sequence s.
`len(s)`	Length of sequence s, i.e. the number of elements in s.
`min(s)`	Smallest element in sequence s.
`max(s)`	Largest element in sequence s.
`sum(s)`	Sum of all numbers in sequence s.
`for loop`	Traverses elements from left to right in a for loop.

List examples using functions

```
>>> list1 = [2, 3, 4, 1, 32]
>>> 2 in list1
True
>>> 33 not in list1
True
>>> len(list1) # find the number of elements in the list
5
>>> max(list1) # find the largest element in the list
32
>>> min(list1) # find the smallest element in the list
1
>>> sum(list1) # sum of elements in the list
42
```

List slicing

Slice operator (`[start:end]`) allows to fetch sublist from the list. It works similar to string.

```
>>> list = [11,33,44,66,788,1]
>>> list[0:5] # this will return list starting from index 0 to index 4
[11,33,44,66,788]
>>> list[:3]
[11,33,44]
```

Similar to string `start` index is optional, if omitted it will be `0`.

```
>>> list[2:]
[44,66,788,1]
```

`end` index is also optional, if omitted it will be set to the last index of the list.

Note: If `start >= end` ,then `list[start : end]` will return an empty list. If `end` specifies a position which is beyond the last element of the list, Python will use the length of the list for `end` instead.

+ and * operators in list

`+` operator joins the two list.

```
>>> list1 = [11, 33]
>>> list2 = [1, 9]
>>> list3 = list1 + list2
>>> list3
[11, 33, 1, 9]
```

`*` operator replicates the elements in the list.

```
>>> list4 = [1, 2, 3, 4]
>>> list5 = list4 * 3
>>> list5
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

`in` or `not in` operator

`in` operator is used to determine whether the elements exists in the list. On success it returns `True` on failure it returns `False` .

```
>>> list1 = [11, 22, 44, 16, 77, 98]
>>> 22 in list1
True
```

Similarly `not in` is opposite of `in` operator.

```
>>> 22 not in list1
False
```

Traversing list using for loop

As already discussed list is a sequence and also iterable. Means you can use for loop to loop through all the elements of the list.

```
>>> list = [1,2,3,4,5]
>>> for i in list:
... print(i, end=" ")
1 2 3 4 5
```

Commonly used list methods with return type

METHOD NAME	DESCRIPTION
`append(x:object):None`	Adds an element x to the end of the list and returns None.
`count(x:object):int`	Returns the number of times element x appears in the list.
`extend(l:list):None`	Appends all the elements in `l` to the list and returns None.
`index(x: object):int`	Returns the index of the first occurrence of element x in the list
`insert(index: int, x: object):None`	Inserts an element x at a given index. Note that the first element in the list has index 0 and returns None..
`remove(x:object):None`	Removes the first occurrence of element x from the list and returns None
`reverse():None`	Reverse the list and returns None
`sort(): None`	Sorts the elements in the list in ascending order and returns None.
`pop(i): object`	Removes the element at the given position and returns it. The parameter i is optional. If it is not specified, pop() removes and returns the last element in the list.

```
>>> list1 = [2, 3, 4, 1, 32, 4]
>>> list1.append(19)
>>> list1
[2, 3, 4, 1, 32, 4, 19]
>>> list1.count(4) # Return the count for number 4
2
>>> list2 = [99, 54]
>>> list1.extend(list2)
>>> list1
[2, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.index(4) # Return the index of number 4
2
>>> list1.insert(1, 25) # Insert 25 at position index 1
>>> list1
[2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>>
>>> list1 = [2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop(2)
3
>>> list1
```

```
[2, 25, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop()
54
>>> list1
[2, 25, 4, 1, 32, 4, 19, 99]
>>> list1.remove(32) # Remove number 32
>>> list1
[2, 25, 4, 1, 4, 19, 99]
>>> list1.reverse() # Reverse the list
>>> list1
[99, 19, 4, 1, 4, 25, 2]
>>> list1.sort() # Sort the list
>>> list1
[1, 2, 4, 4, 19, 25, 99]
>>>
```

List Comprehension

Note: This topic needs to have a working knowledge of python for loops.

List comprehension provides a concise way to create list. It consists of square brackets containing expression followed by for clause then zero or more for or if clauses.

here are some examples:

```
>>> list1 = [ x for x in range(10) ]
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list2 = [ x + 1 for x in range(10) ]
>>> list2
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> list3 = [ x for x in range(10) if x % 2 == 0 ]
>>> list3
[0, 2, 4, 6, 8]
```

```
>>> list4 = [ x *2 for x in range(10) if x % 2 == 0 ]
[0, 4, 8, 12, 16]
```

In the next tutorial we will learn about python dictionaries.

Chapter – 8

Python Dictionaries

Dictionary is a python data type that is used to store key value pairs. It enables you to quickly retrieve, add, remove, modify, values using key. Dictionary is very similar to what we call associative array or hash on other languages.

Note: Dictionaries are mutable.

Creating Dictionary

Dictionaries can be created using pair of curly braces (`{}`). Each item in the dictionary consist of key, followed by a colon (`:`), which is followed by value. And each item is separated using commas (`,`). Let's take an example.

```
friends = {
'tom' : '111-222-333',
'jerry' : '666-33-111'
}
```

here `friends` is a dictionary with two items. One point to note that key must be of hashable type, but value can be of any type. Each key in the dictionary must be unique.

```
>>> dict_emp = {} # this will create an empty dictionary
```

Retrieving, modifying and adding elements in the dictionary

To get an item from dictionary, use the following syntax:

```
>>> dictionary_name['key']
>>> friends['tom']
'111-222-333'
```

if the key exists in the dictionary, the value will be returned otherwise KeyError exception will be thrown.

To add or modify an item, use the following syntax:

```
>>> dictionary_name['newkey'] = 'newvalue'
>>> friends['bob'] = '888-999-666'
>>> friends
{'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}
```

Deleting Items from dictionary.

```
>>> del dictionary_name['key']
>>>  del friends['bob']
>>>  friends
{'tom': '111-222-333', 'jerry': '666-33-111'}
```

If the key is found then item will be deleted otherwise `KeyError` exception will be thrown.

Looping items in the dictionary

You can use for loop to traverse elements in the dictionary.

```
>>> friends = {
... 'tom' : '111-222-333',
... 'jerry' : '666-33-111'
...}
>>>
>>> for key in friends
... print(key, ":", friends[key])
...
tom : 111-222-333
jerry : 666-33-111
>>>
```

```
>>>
```

Find the length of the dictionary

You can use `len()` function to find the length of the dictionary.

```
>>> len(friends)
2
```

`in` or `not in` operators

`in` and `notin` operators to check whether key exists in the dictionary.

```
>>> 'tom' in friends
True
>>> 'tom' not in friends
False
```

Equality Tests in dictionary

`==` and `!=` operators tells whether dictionary contains same items not.

```
>>> d1 = {"mike":41, "bob":3}
>>> d2 = {"bob":3, "mike":41}
>>> d1 == d2
True
>>> d1 != d2
False
>>>
```

Note: You can't use other relational operators like `<, >, >=, <=` to compare dictionaries.

Dictionary methods

Python provides you several built-in methods for working with dictionaries.

METHODS	DESCRIPTION
popitem()	Returns randomly select item from dictionary and also remove the selected item.
clear()	Delete everything from dictionary
keys()	Return keys in dictionary as tuples
values()	Return values in dictionary as tuples
get(key)	Return value of key, if key is not found it returns None, instead on throwing KeyError exception
pop(key)	Remove the item from the dictionary, if key is not found KeyError will be thrown

```
>>> friends = {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}

>>> friends.popitem()
('tom', '111-222-333')

>>> friends.clear()

>>>  friends
{}

>>> friends = {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}

>>> friends.keys()
dict_keys(['tom', 'bob', 'jerry'])


>>> friends.values()
dict_values(['111-222-333', '888-999-666', '666-33-111'])

>>> friends.get('tom')
'111-222-333'
```

```
>>> friends.get('mike', 'Not Exists')
'Not Exists'


>>> friends.pop('bob')
'888-999-666'


>>> friends
{'tom': '111-222-333', 'jerry': '666-33-111'}
```

In next post we will learn about Python tuples.

Chapter – 9

Python Tuples

In Python Tuples are very similar to list but once a tuple is created, you cannot add, delete, replace, reorder elements.

Note: Tuples are immutable.

Creating a tuple

```
>>> t1 = () # creates an empty tuple with no data
>>> t2 = (11,22,33)
>>> t3 = tuple([1,2,3,4,4]) # tuple from array
>>> t4 = tuple("abc") # tuple from string
```

Tuples functions

Functions like `max` , `min` , `len` , `sum` can also be used with tuples.

```
>>> t1 = (1, 12, 55, 12, 81)
>>> min(t1)
1
>>> max(t1)
81
>>> sum(t1)
161
>>> len(t1)
5
```

Iterating through tuples

Tuples are iterable using for loop.

```
>>> t = (11,22,33,44,55)
>>> for i in t:
... print(i, end=" ")
>>> 11 22 33 44 55
```

Slicing tuples

Slicing operators works same in tuples as in list and string.

```
>>> t = (11,22,33,44,55)
>>> t[0:2]
(11,22)
```

`in` and `not in` operator

You can use `in` and `not in` operators to check existence of item in tuples as follows.

```
>>> t = (11,22,33,44,55)
>>> 22 in t
True
>>> 22 not in t
False
```

In next chapter we will learn about python data type conversion.

Chapter – 10

Python Datatype conversion

Once in a while you will want to convert data type of one type to another type. Data type conversion is also known as Type casting.

Converting int to float

To convert `int` to `float` you need to use `float()` function.

```
>>> i = 10
>>> float(i)
10.0
```

Converting float to int

To convert `float` to `int` you need to use `int()` function.

```
>>> f = 14.66
>>> int(f)
14
```

Converting string to int

You can also use `int()` to convert string to int.

```
>>> s = "123"
>>> int(s)
123
```

Note: If string contains non numeric character then `int()` will throw `ValueError`.

Converting number to string

To convert number to string you need to use `str()` function.

```
>>> i = 100
>>> str(i)
"100"
>>> f = 1.3
str(f)
'1.3'
```

Rounding numbers

To round numbers you need to use `round()` function.

Syntax: `round(number[, ndigits])`

```
>>> i = 23.97312
>>> round(i, 2)
23.97
```

Next we will cover control statements.

Chapter – 11

Python Control Statements

It is very common for programs to execute statements based on some conditions. In this section we will learn about python `if .. else ...` statement.

But before we need to learn about relational operators. Relational operators allows us to compare two objects.

SYMBOL	DESCRIPTION
<=	smaller than or equal to
<	smaller than
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

The result of comparision will always be a boolean value i.e `True` or `False`. Remember `True` and `False` are python keyword for denoting boolean values.

Let take some examples:

```
>>> 3 == 4
False
>>> 12 > 3
True
```

```
>>> 12 == 12
True
>>> 44 != 12
True
```

Now you are ready to tackle `if statements`.The syntax of `If statement` is:

```
if boolean-expression:
   #statements
else:
   #statements
```

Note: Each statements in the if block must be indented using the same number of spaces, otherwise it will lead to syntax error. This is very different from many other languages like Java, C, C# where curly braces (`{}`) is used.

Now let's see an example

```
i = 10

if i % 2 == 0:
   print("Number is even")

else:
   print("Number is odd")
```

here you can see that if number is even then `"Number is even"` is printed otherwise `"Number is odd"` is printed.

Note: `else` clause is optional you can use only `if` clause if you want, like this

```
if today == "party":
    print("thumbs up!")
```

here when value of today is `"party"` then thumbs up! will get printed, otherwise nothing will print.

If your programs needs to check long list of conditions then you need to use `if-elif-else` statements.

```
if boolean-expression:
   #statements
elif boolean-expression:
   #statements
```

```
elif boolean-expression:
   #statements
elif boolean-expression:
   #statements
else:
   #statements
```

You can add as many `elif` condition as programs demands.

here is an example to illustrate `if-elif-else` statement.

```
today = "monday"


if today == "monday":
   print("this is monday")
elif today == "tuesday":
   print("this is tuesday")
elif today == "wednesday":
   print("this is wednesday")
elif today == "thursday":
   print("this is thursday")
elif today == "friday":
   print("this is friday")
elif today == "saturday":
   print("this is saturday")
elif today == "sunday":
   print("this is sunday")
else:
   print("something else")
```

Nested if statements

You can nest `if statements` inside another `if statements` as follows:

```
today = "holiday"
bank_balance = 25000

if today == "holiday":
   if bank_balance > 20000:
      print("Go for shopping")
   else:
      print("Watch TV")
else:
   print("normal working day")
```

In the next post we will learn about Python Functions.

Chapter – 12

Python Functions

Functions are the re-usable pieces of code which helps us to organize structure of the code. We create functions so that we can run a set of statements multiple times during in the program without repeating ourselves.

Creating functions

Python uses def keyword to start a function, here is the syntax:

```
def function_name(arg1, arg2, arg3, .... argN):
     #statement inside function
```

Note: All the statements inside the function should be indented using equal spaces. Function can accept zero or more arguments(also known as parameters) enclosed in parentheses. You can also omit the body of the function using the pass keyword, like this:

```
def myfunc():
    pass
```

Let's see an example.

```
def sum(start, end):
   result = 0
   for i in range(start, end + 1):
       result += i
   print(result)
```

```
sum(10, 50)
```

Expected output:

```
1230
```

Above we define a function called `sum()` with two parameters `start` and `end`, function calculates the sum of all the numbers starting from `start` to `end`.

Function with return value.

The above function simply prints the result to the console, what if we want to assign the result to a variable for further processing ? Then we need to use the return statement. The `return` statement sends a result back to the caller and exits the function.

```
def sum(start, end):
    result = 0
    for i in range(start, end + 1):
        result += i
    return result


s = sum(10, 50)
print(s)
```

Expected Output:

```
1230
```

Here we are using `return` statement to return the sum of numbers and assign it to variable `s`.

You can also use the `return` statement without a return value.

```
def sum(start, end):
    if(start > end):
        print("start should be less than end")
        return    # here we are not returning any value so a special value
                  # None is returned
    result = 0
    for i in range(start, end + 1):
        result += i
    return result
```

```
s = sum(110, 50)
print(s)
```

Expected Output:

```
start should be less than end
None
```

In python if you do not explicitly return value from a function , then a special value `None` is always returned. Let's take an example

```
def test():   # test function with only one statement
    i = 100

print(test())
```

Expected Output

```
None
```

as you can see `test()` function doesn't explicitly return any value. so `None` is returned.

Global variables vs local variables

Global variables: Variables that are not bound to any function , but can be accessed inside as well as outside the function are called global variables.

Local variables: Variables which are declared inside a function are called local variables.

Let's see some examples to illustrate this point.

Example 1:

```
global_var = 12         # a global variable


def func():
    local_var = 100     # this is local variable
    print(global_var)   # you can access global variables in side function


func()                  # calling function func()
```

```
#print(local_var)        # you can't access local_var outside the function,
                         # because as soon as function ends local_var is
                         # destroyed
```

Expected Output:

```
12
```

Example 2:

```
xy = 100


def cool():
    xy = 200     # xy inside the function is totally different from xy
                 # outside the function
    print(xy)    # this will print local xy variable i.e 200


cool()
print(xy)        # this will print global xy variable i.e 100
```

Expected Output:

```
200
100
```

You can bind local variable in the global scope by using the `global` keyword followed by the names of variables separated by comma (`,`).

```
t = 1


def increment():
    global t   # now t inside the function is same as t outside the function
```

```
    t = t + 1
    print(t) # Displays 2



increment()
print(t) # Displays 2
```

Expected Output:

```
2
2
```

Note that you can't assign a value to variable while declaring them global .

```
t = 1


def increment():
    #global t = 1   # this is error
    global t
    t = 100        # this is okay
    t = t + 1
    print(t) # Displays 101



increment()
print(t) # Displays 101
```

Expected Output:

```
101
101
```

In fact there is no need to declare global variables outside the function. You can declare them global inside the function.

```
def foo():
    global x   # x is declared as global so it is available outside the
               # function
    x = 100


foo()
print(x)
```

Expected Output:

```
100
```

Argument with default values

To specify default values of argument, you just need to assign a value using assignment operator.

```
def func(i, j = 100):
    print(i, j)
```

Above function has two parameter `i` and `j` .`j` has default value of `100` , means we can omit value of `j` while calling the function.

```
func(2) # here no value is passed to j, so default value will be used
```

Expected Output:

```
2 100
func(2, 300) # here 300 is passed as a value of j, so default value will not
             # be used
```

Expected Output:

```
2 300
```

Keyword arguments

There are two ways to pass arguments to method: **positional arguments** and **Keyword arguments**. We have already seen how positional arguments work in the previous section. In this section we will learn about keyword arguments.

Keyword arguments allows you to pass each arguments using name value pairs like this `name=value` . Let's take an example:

```
def named_args(name, greeting):
    print(greeting + " " + name )


named_args(name='jim', greeting='Hello')
Hello jim


# you can pass arguments this way too

named_args(greeting='Hello', name='jim')
Hello jim
```

Mixing Positional and Keyword Arguments

It is possible to mix positional arguments and Keyword arguments, but for this positional argument must appear before any Keyword arguments. Let's see this through an example.

```
def my_func(a, b, c):
    print(a, b, c)
```

You can call the above function in the following ways.

```
# using positional arguments only
my_func(12, 13, 14)


# here first argument is passed as positional arguments while other two as
keyword argument
my_func(12, b=13, c=14)


# same as above
my_func(12, c=13, b=14)


# this is wrong as positional argument must appear before any keyword
argument
# my_func(12, b=13, 14)
```

Expected Output:

```
12 13 14
12 13 14
12 14 13
```

Returning multiple values from Function

We can return multiple values from function using the return statement by separating them with a comma (,). Multiple values are returned as tuples.

```
def bigger(a, b):
    if a > b:
        return a, b
    else:
        return b, a


s = bigger(12, 100)
print(s)
print(type(s))
```

Expected Output:

```
(100, 12)
<class 'tuple'>
```

In the next post we will learn about Python Loops.

Chapter – 13

Python Loops

Python has only two loops: for loop and while loop.

For loop

For loop Syntax:

```
for i in iterable_object:
   # do something
```

Note: all the statements inside for and while loop must be indented to the same number of spaces. Otherwise `SyntaxError` will be thrown.

Let's take an example

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

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

Expected Output

```
1
2
3
4
```

here is how for loop works.

In the first iteration `i` is assigned value `1` then print statement is executed, In second iteration `i` is assigned value `2` then once again `print` statement is executed. This process continues until there are no more element in the list and for loop exists.

`range(a, b)` Function

`range(a, b)` functions returns sequence of integers from `a` , `a + 1` , `a+ 2` .... , `b - 2` , `b - 1` . For e.g

```
for i in range(1, 10):
    print(i)
```

Expected Output:

```
1
2
3
4
5
6
7
8
9
```

You can also use `range()` function by supplying only one argument like this:

```
>>> for i in range(10):
...         print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

here range for loop prints number from `0` to `9`.

`range(a, b)` function has an optional third parameter to specify the step size. For e.g

```
for i in range(1, 20, 2):
    print(i)
```

Expected Output:

```
1
3
5
7
9
11
13
15
17
19
```

While loop

Syntax:

```
while condition:
    # do something
```

While loop keeps executing statements inside it until condition becomes `False`. After each iteration condition is checked and if its `True` then once again statements inside the while loop will be executed.

let's take an example:

```
count = 0


while count < 10:
    print(count)
    count += 1
```

Expected Output:

```
0
1
2
3
4
5
6
7
8
9
```

here while will keep printing until `count` is less than `10`.

`break` statement

`break` statement allows to breakout out of the loop.

```
count = 0
```

```
while count < 10:
    count += 1
    if count == 5:
         break
    print("inside loop", count)


print("out of while loop")
```

when `count` equals to `5` if condition evaluates to `True` and `break` keyword breaks out of loop.

Expected Output:

```
inside loop 1
inside loop 2
inside loop 3
inside loop 4
out of while loop
```

`continue` statement

When `continue` statement encountered in the loop, it ends the current iteration and programs control goes to the end of the loop body.

```
count = 0


while count < 10:
    count += 1
    if count % 2 == 0:
           continue
    print(count)
```

Expected Output:

```
1
3
5
7
9
```

As you can see when `count % 2 == 0`, `continue` statement is executed which causes the current iteration to end and the control moves on to the next iteration.

In next lesson we will learn about Python mathematical function.

Chapter – 14

Python Mathematical Functions

Python has many inbuilt function.

METHOD	DESCRIPTION
round(number[, ndigits])	rounds the number, you can also specify precision in the second argument
pow(a, b)	Returns a raise to the power of b
abs(x)	Return absolute value of x
max(x1, x2, ..., xn)	Returns largest value among supplied arguments
min(x1, x2, ..., xn)	Returns smallest value among supplied arguments

Below mentioned functions are in math module, so you need to import math module first, using the following line.

```
import math
```

METHOD	DESCRIPTION
ceil(x)	This function rounds the number up and returns its nearest integer
floor(x)	This function rounds the down up and returns its nearest integer
sqrt(x)	Returns the square root of the number
sin(x)	Returns sin of x where x is in radian
cos(x)	Returns cosine of x where x is in radian
tan(x)	Returns tangent of x where x is in radian

Let's take some examples to understand better

```
>>> abs(-22)              # Returns the absolute value
22

>>> max(9, 3, 12, 81)     # Returns the maximum number
81

>>> min(78, 99, 12, 32)   # Returns the minimum number
12
```

```
>>> pow(8, 2)              # can also be written as 8 ** 2
64

>>> pow(4.1, 3.2)          # can also be written as 4.1 ** 3.2
91.39203368671122

>>> round(5.32)            # Rounds to its nearest integer
5

>>> round(3.1456875712, 3) # Return number with 3 digits after decimal point
3.146


>>> import math
>>> math.ceil(3.4123)
4

>>> math.floor(24.99231)
24
```

In next post we will learn how to generate random numbers in python.

Chapter – 15

Python Generating Random numbers

Python `random` module contains function to generate random numbers. So first you need to import `random` module using the following line.

```
import random
```

`random()` Function

`random()` function returns random number `r` such that `0 <= r < 1.0 .`

```
>>> import random
>>> for i in range(0, 10):
... print(random.random())
...
```

Expected Output:

```
0.9240468209780505
0.14200320177446257
0.8647635207997064
0.23926674191769448
0.4436673317102027
0.09211695432442013
0.2512541244937194
0.7279402864974873
0.3864708801092763
0.08450122561765672
```

`randint(a, b)` generate random numbers between `a` and `b` (inclusively).

```
>>> import random
>>> for i in range(0, 10):
... print(random.randint(1, 10))
...
8
3
4
7
1
5
3
```

```
7
3
3
```

Next chapter will cover file handling techniques in python.

Chapter – 16

Python File Handling

We can use File handling to read and write data to and from the file.

Opening a file

Before reading/writing you first need to open the file. Syntax of opening a file is.

```
f = open(filename, mode)
```

`open()` function accepts two arguments `filename` and `mode`. `filename` is a string argument which specify filename along with it's path and `mode` is also a string argument which is used to specify how file will be used i.e for reading or writing. And `f` is a file handler object also known as file pointer.

Closing a file

After you have finished reading/writing to the file you need to close the file using `close()` method like this,

```
f.close()  # where f is a file pointer
```

Different modes of opening a file are

MODES	DESCRIPTION
`"r"`	Open a file for read only
`"w"`	Open a file for writing. If file already exists its data will be cleared before opening. Otherwise new file will be created
`"a"`	Opens a file in append mode i.e to write a data to the end of the file
`"wb"`	Open a file to write in binary mode
`"rb"`	Open a file to read in binary mode

Let's now look at some examples.

Writing data to the file

```
>>> f = open('myfile.txt', 'w')        # open file for writing
>>> f.write('this first line\n')       # write a line to the file
>>> f.write('this second line\n')      # write one more line to the file
>>> f.close()                          # close the file
```

Note: `write()` method will not insert new line (`'\n'`) automatically like print function, you need to explicitly add `'\n'` to write `write()` method.

Reading data from the file

To read data back from the file you need one of these three methods.

METHODS	DESCRIPTION
`read([number])`	Return specified number of characters from the file. if omitted it will read the entire contents of the file.
`readline()`	Return the next line of the file.
`readlines()`	Read all the lines as a list of strings in the file

Reading all the data at once.

```
>>> f = open('myfile.txt', 'r')
>>> f.read() # read entire content of file at once
"this first line\nthis second line\n"
>>> f.close()
```

Reading all lines as an array.

```
>>> f = open('myfile.txt', 'r')
>>> f.readlines() # read entire content of file at once
["this first line\n", "this second line\n"]
>>> f.close()
```

Reading only one line.

```
>>> f = open('myfile.txt', 'r')
>>> f.readline() # read entire content of file at once
"this first line\n"
>>> f.close()
```

Appending data

To append the data you need open file in `'a'` mode.

```
>>> f = open('myfile.txt', 'a')
>>> f.write("this is third line\n")
19
>>> f.close()
```

Looping through the data using for loop

You can iterate through the file using file pointer.

```
>>> f = open('myfile.txt' 'r')
>>> for line in f:
... print(line)
...
this first line
this second line
this is third line

>>> f.close()
```

Binary reading and writing

To perform binary i/o you need to use a module called `pickle` , `pickle` module allows you to read and write data using `load` and `dump` method respectively.

Writing data

```
>> import pickle
>>> f = open('pick.dat', 'wb')
>>> pickle.dump(11, f)
>>> pickle.dump("this is a line", f)
>>> pickle.dump([1, 2, 3, 4], f)
>>> f.close()
```

Reading data

```
>> import pickle
>>> f = open('pick.dat', 'rb')
>>> pickle.load(f)
11
>>> pickle.load()
"this is a line"
>>> pickle.load()
[1,2,3,4]
>>> f.close()
```

If there is no more data to read from the file `pickle.load()` throws `EOFError` or end of file error.

In the next lesson we will learn about classes and objects in python.

Chapter – 17

Python Object and Classes

Creating object and classes

Python is an object-oriented language. In python everything is object i.e `int` , `str` , `bool` even modules, functions are also objects.

Object oriented programming use objects to create programs, and these objects stores data and behaviors.

Defining class

Class name in python is preceded with `class` keyword followed by colon (`:`). Classes commonly contains data field to store the data and methods for defining behaviors. Also every class in python contains a special method called initializer (also commonly known as constructors), which get invoked automatically every time new object is created.

Let's see an example.

```
class Person:

       # constructor or initializer
      def __init__(self, name):
            self.name = name # name is data field also commonly known as
                             # instance variables


      # method which returns a string
      def whoami(self):
            return "You are " + self.name
```

here we have created a class called `Person` which contains one data field called `name` and method `whoami()`.

What is self ??

All methods in python including some special methods like initializer have first parameter `self` . This parameter refers to the object which invokes the method. When you create new object the self parameter in the `__init__` method is automatically set to reference the object you have just created.

Creating object from class

```
p1 = Person('tom') # now we have created a new person object p1
print(p1.whoami())
print(p1.name)
```

Expected Output:

```
You are tom
tom
```

Note: When you call a method you don't need to pass anything to `self` parameter, python automatically does that for you behind the scenes.

You can also change the name data field.

```
p1.name = 'jerry'
print(p1.name)
```

Expected Output:

```
jerry
```

Although it is a bad practice to give access to your data fields outside the class. We will discuss how to prevent this next.

Hiding data fields

To hide data fields you need to define private data fields. In python you can create private data field using two leading underscores. You can also define a private method using two leading underscores.

Let's see an example

```
class BankAccount:


     # constructor or initializer
     def __init__(self, name, money):
        self.__name = name
        self.__balance = money    # __balance is private now, so it is only
                                  # accessible inside the class


    def deposit(self, money):
         self.__balance += money


    def withdraw(self, money):
         if self.__balance > money :
             self.__balance -= money
             return money
         else:
             return "Insufficient funds"


    # method which returns a string

    def checkbalance(self):
         return self.__balance


b1 = BankAccount('tim', 400)
print(b1.withdraw(500))
b1.deposit(500)
print(b1.checkbalance())
print(b1.withdraw(800))
print(b1.checkbalance())
```

Expected Output:

```
Insufficient funds
900
800
100
```

Let's try to access `__balance` data field outside of class.

```
print(b1.__balance)
```

Expected Output:

```
AttributeError: 'BankAccount' object has no attribute '__balance'
```

As you can see now `__balance` is not accessible outside the class.

In next chapter we will learn about operator overloading.

Chapter – 18

Python Operator Overloading

You have already seen you can use + operator for adding numbers and at the same time to concatenate strings. It is possible because + operator is overloaded by both int class and str class. The operators are actually methods defined in respective classes. Defining methods for operators is known as operator overloading. For e.g. To use + operator with custom objects you need to define a method called __add__ .

Let's take an example to understand better

```
import math

class Circle:
    def __init__(self, radius):
        self.__radius = radius


    def setRadius(self, radius):
        self.__radius = radius


    def getRadius(self):
        return self.__radius


    def area(self):
        return math.pi * self.__radius ** 2


    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )
```

```
c1 = Circle(4)
print(c1.getRadius())


c2 = Circle(5)
print(c2.getRadius())


c3 = c1 + c2 # This became possible because we have overloaded + operator by
             # adding a method named __add__

print(c3.getRadius())
```

Expected Output:

```
4
5
9
```

In the above example we have added `__add__` method which allows use to use `+` operator to add two circle objects. Inside the `__add__` method we are creating a new object and returning it to the caller.

python has many other special methods like `__add__` , see the list below.

OPERATOR	FUNCTION	METHOD DESCRIPTION
+	__add__(self, other)	Addition
*	__mul__(self, other)	Multiplication
-	__sub__(self, other)	Subtraction
%	__mod__(self, other)	Remainder
/	__truediv__(self, other)	Division
<	__lt__(self, other)	Less than
<=	__le__(self, other)	Less than or equal to
==	__eq__(self, other)	Equal to
!=	__ne__(self, other)	Not equal to
>	__gt__(self, other)	Greater than
>=	__ge__(self, other)	Greater than or equal to

[index]	__getitem__(self, index)	Index operator
in	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
str	__str__(self)	The string representation

Program below is using some of the above mentioned functions to overload operators.

```
import math

class Circle:

    def __init__(self, radius):
        self.__radius = radius
```

```
    def setRadius(self, radius):
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def area(self):
        return math.pi * self.__radius ** 2

    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )

    def __gt__(self, another_circle):
        return self.__radius > another_circle.__radius

    def __lt__(self, another_circle):
        return self.__radius < another_circle.__radius

    def __str__(self):
        return "Circle with radius " + str(self.__radius);

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2
print(c3.getRadius())

print( c3 > c2) # Became possible because we have added __gt__ method
print( c1 < c2) # Became possible because we have added __lt__ method
print(c3) # Became possible because we have added __str__ method
```

Expected Output:

```
4
5
9
```

```
True
True
Circle with radius 9
```

Next lesson is inheritance and polymorphism.

Chapter – 19

Python inheritance and polymorphism

Inheritance allows programmer to create a general class first then later extend it to more specialized class. It also allows programmer to write better code.

Using inheritance you can inherit all access data fields and methods, plus you can add your own methods and fields, thus inheritance provide a way to organize code, rather than rewriting it from scratch.

In object-oriented terminology when class X extend class Y , then Y is called **super class** or **base class** and X is called **subclass or derived class**. One more point to note that only data fields and method which are not private are accessible by child classes, private data fields and methods are accessible only inside the class.

Syntax to create a subclass is:

```
class SubClass(SuperClass):
  # data fields
  # instance methods
```

Let take an example to illustrate the point.

```
class Vehicle:

    def __init__(self, name, color):
        self.__name = name      # __name is private to Vehicle class
        self.__color = color


    def getColor(self):      # getColor() function is accessible to class Car
        return self.__color


    def setColor(self, color):  # setColor is accessible outside the class
        self.__color = color


    def getName(self):          # getName() is accessible outside the class
        return self.__name


class Car(Vehicle):


    def __init__(self, name, color, model):
        # call parent constructor to set name and color
        super().__init__(name, color)
        self.__model = model


    def getDescription(self):
        return self.getName() + self.__model + " in " + \
               self.getColor() + "color"
```

```
# in method getDescrition we are able to call getName(), getColor() because
# they are accessible to child class through inheritance


c = Car("Ford Mustang", "red", "GT350")
print(c.getDescription())
print(c.getName()) # car has no method getName() but it is accessible
# through class Vehicle
```

Expected Output:

```
Ford MustangGT350 in red color
Ford Mustang
```

here we have created base class `Vehicle` and it's subclass `Car` . Notice that we have not defined `getName()` in `Car` class but we are still able to access it, because class `Car` inherits it from `Vehicle` class. In the above code `super()` method is used to call method of the base class. Here is the how `super()` works

Suppose you need to call a method called `get_information()` in the base class from child class , you can do so using the following code.

```
super().get_information()
```

similarly you can call base class constructor from child class constructor using the following code.

```
super().__init__()
```

Multiple inheritance

Unlike languages like Java and C#, python allows multiple inheritance i.e you can inherit from multiple classes at the same time like this,

```
class Subclass(SuperClass1, SuperClass2, ...):
   # initializer
   # methods
```

Let's take an example:

```
class MySuperClass1():
```

```
    def method_super1(self):
        print("method_super1 method called")


class MySuperClass2():

    def method_super2(self):
        print("method_super2 method called")


class ChildClass(MySuperClass1, MySuperClass2):


    def child_method(self):
        print("child method")


c = ChildClass()
c.method_super1()
c.method_super2()
```

Expected Output:

```
method_super1 method called
method_super2 method called
```

As you can see becuase `ChildClass` inherited `MySuperClass1` , `MySuperClass2` , object of `ChildClass` is now able to access `method_super1()` and `method_super2()` .

Overriding methods

To override a method in the base class, sub class needs to define a method of same signature. (i.e same method name and same number of parameters as method in base class).

```
class A():


    def __init__(self):
        self.__x = 1
```

```
    def m1(self):
        print("m1 from A")


class B(A):

    def __init__(self):
        self.__y = 1

    def m1(self):
        print("m1 from B")

c = B()
c.m1()
```

Expected Output:

```
m1 from B
```

Here we are overriding `m1()` method from the base class. Try commenting `m1()` method in `B` class and now `m1()` method from Base class i.e class `A` will run.

Expected Output:

```
m1 from A
```

`isinstance()` function

`isinstance()` function is used to determine whether the object is an instance of the class or not.

Syntax: `isinstance(object, class_type)`

```
>>> isinstance(1, int)
True
>>> isinstance(1.2, int)
False


>>> isinstance([1,2,3,4], list)
True
```

Next chapter Exception Handling.

Chapter – 20

Python Exception Handling

Exception handling enables you handle errors gracefully and do something meaningful about it. Like display a message to user if intended file not found. Python handles exception using `try .. except ..` block.

Syntax:

```
try:
    # write some code
    # that might throw exception
except <ExceptionType>:
    # Exception handler, alert the user
```

As you can see in try block you need to write code that might throw an exception. When exception occurs code in the try block is skipped. If there exist a matching exception type in `except` clause then it's handler is executed.

Let's take an example:

```
try:
    f = open('somefile.txt', 'r')
    print(f.read())
    f.close()
except IOError:
    print('file not found')
```

The above code work as follows:

1. First statement between `try` and `except` block are executed.
2. If no exception occurs then code under `except` clause will be skipped.
3. If file don't exists then exception will be raised and the rest of the code in the `try` block will be skipped
4. When exceptions occurs, if the exception type matches exception name after except keyword, then the code in that `except` clause is executed.

Note: The above code is only capable of handling `IOError` exception. To handle other kind of exception you need to add more `except` clause.

A `try` statement can have more than once `except` clause, It can also have optional `else` and/or `finally` statement.

```
try:
    <body>
except <ExceptionType1>:
    <handler1>
except <ExceptionTypeN>:
    <handlerN>
except:
    <handlerExcept>
else:
    <process_else>
finally:
    <process_finally>
```

`except` clause is similar to `elif` . When exception occurs, it is checked to match the exception type in `except` clause. If match is found then handler for the matching case is executed. Also note that in last `except` clause ExceptionType is omitted. If exception does not match any exception type before the last `except` clause, then the handler for last except `clause` is executed.

Note: Statements under the `else` clause run only when no exception is raised.

Note: Statements in `finally` block will run every time no matter exception occurs or not.

Now let's take an example.

```
try:
    num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
    result = num1 / num2
    print("Result is", result)


except ZeroDivisionError:
    print("Division by zero is error !!")




except SyntaxError:
    print("Comma is missing.Enter numbers separated by comma like this 1, 2")
```

```
except:
    print("Wrong input")


else:
    print("No exceptions")


finally:
    print("This will execute no matter what")
```

Note: The `eval()` function lets a python program run python code within itself, `eval()` expects a string argument.

To learn more about `eval()` see the link http://stackoverflow.com/questions/9383740/what-does-pythons-eval-do

Raising exceptions

To raise your exceptions from your own methods you need to use `raise` keyword like this

```
raise ExceptionClass("Your argument")
```

Let's take an example

```
def enterage(age):
    if age < 0:
        raise ValueError("Only positive integers are allowed")


    if age % 2 == 0:
        print("age is even")
    else:
        print("age is odd")

try:
    num = int(input("Enter your age: "))
    enterage(num)
except ValueError:
    print("Only positive integers are allowed")

except:
    print("something is wrong")
```

Run the program and enter positive integer.

Expected Output:

```
Enter your age: 12
age id even
```

Again run the program and enter a negative number.

Expected Output:

```
Enter your age: -12
Only integers are allowed
```

Using Exception objects

Now you know how to handle exception, in this section we will learn how to access exception object in exception handler code. You can use the following code to assign exception object to a variable.

```
try:
    # this code is expected to throw exception
except ExceptionType as ex:
    # code to handle exception
```

As you can see you can store exception object in variable ex . Now you can use this object in exception handler code

```
try:
    number = eval(input("Enter a number: "))
    print("The number entered is", number)
except NameError as ex:
    print("Exception:", ex)
```

Run the program and enter a number.

Expected Output:

```
Enter a number: 34
The number entered is 34
```

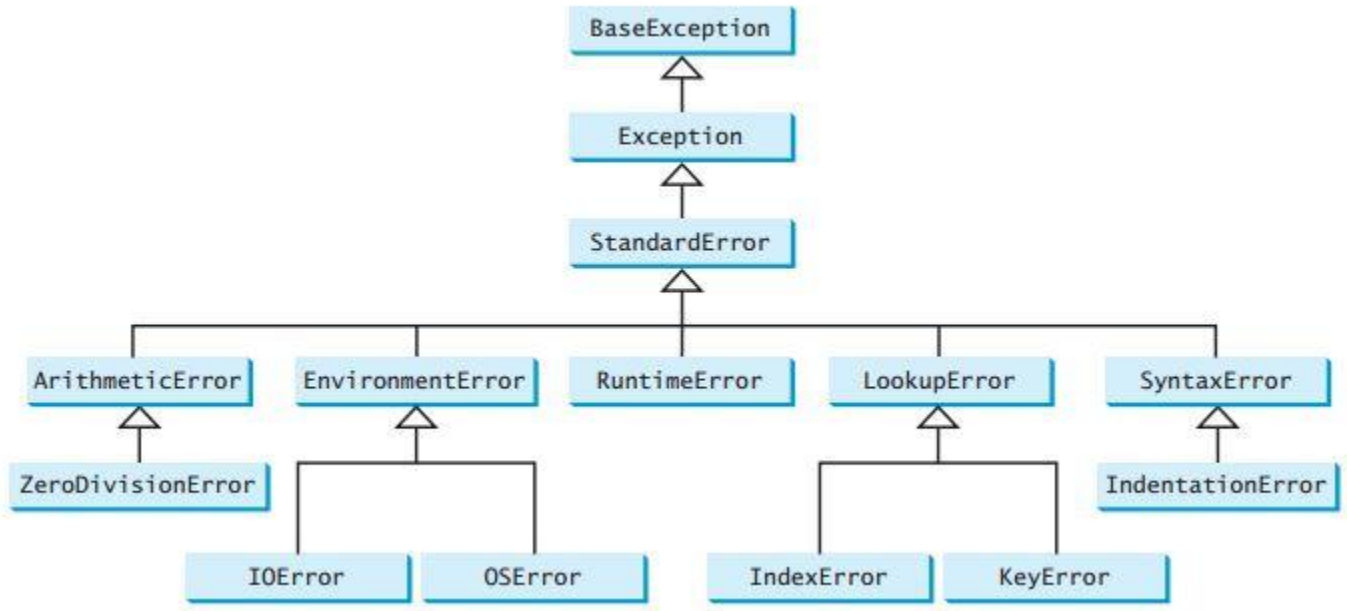
Again run the program and enter a string .

Expected Output:

```
Enter a number: one
Exception: name 'one' is not defined
```

Creating custom exception class

You can create a custom exception class by Extending `BaseException` class or subclass of `BaseException` .



As you can see from most of the exception classes in python extends from the `BaseException` class. You can derive you own exception class from `BaseException` class or from sublcass of `BaseException` like `RuntimeError` .

Create a new file called `NegativeAgeException.py` and write the following code.

```
class NegativeAgeException(RuntimeError):
    def __init__(self, age):
        super().__init__()
        self.age = age
```

Above code creates a new exception class named `NegativeAgeException` , which consists of only constructor which call parent class constructor using `super().__init__()` and sets the age .

Using custom exception class

```
def enterage(age):
    if age < 0:
        raise NegativeAgeException("Only positive integers are allowed")

    if age % 2 == 0:
        print("age is even")
    else:
        print("age is odd")


try:
    num = int(input("Enter your age: "))
    enterage(num)
except NegativeAgeException:
    print("Only positive integers are allowed")
except:
    print("something is wrong")
```

In the next post we will learn about Python Modules.

Chapter – 21

Python Modules

Python module is a normal python file which can store function, variable, classes, constants etc. Module helps us to organize related codes . For e.g `math` module in python has mathematical related functions.

Creating module

Create a new file called `mymodule.py` and write the following code.

```
foo = 100


def hello():
    print("i am from mymodule.py")
```

as you can see we have defined a global variable `foo` and a function `hello()` in our module. Now to use this module in our programs we first need to import it using import statement like this

```
import mymodule
```

now you can use variable and call functions in the `mymodule.py` using the following code.

```
import mymodule
print(mymodule.foo)
print(mymodule.hello())
```

Expected Output:

```
100
i am from mymodule.py
```

Remember you need to specify name of module first to access it's variables and functions, failure to so will result in error.

Using from with import

Using import statements imports everything in the module, what if you want to access only specific function or variable ? This is where `from` statement comes, here is how to use it.

```
# this statement import only foo variable from mymodule
from mymodule import foo

print(foo)
```

Expected output:

```
100
```

Note: In this case you don't need to specify module name to access variables and function.

dir() method

`dir()` is an in-built method used to find all attributes (i.e all available classes, functions, variables and constants) of the object. As we have already discussed everything in python is object, we can use `dir()` method to find attributes of the module like this:

```
dir(module_name)
```

`dir()` returns a list of string containing the names of the available attributes.

```
>>> dir(mymodule)
['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'foo', 'hello']
```

As you can see besides `foo` and `hello` there are additional attributes in the `mymodule` . These are in-built attributes which python provides to all the modules automatically.

Congratulations you have completed all building blocks you need to master python !!

Chapter – 22

Python *args and **kwargs

What is *args ??

`*args` allows us to pass variable number of arguments to the function. Let's take an example to make this clear.

Suppose you created a function to add two number like this.

```
def sum(a, b):
    print("sum is", a+b)
```

As you can see this program only accepts two numbers, what if you want to pass more than two

arguments, this is where `*args` comes into play.

```
def sum(*args):
    s = 0
    for i in args:
        s += i
    print("sum is", s)
```

Now you can pass any number of arguments to the function like this,

```
>>> sum(1, 2, 3)
6
>>> sum(1, 2, 3, 4, 5, 7)
22
>>> sum(1, 2, 3, 4, 5, 7, 8, 9, 10)
49
>>> sum()
0
```

Note: name of `*args` is just a convention you can use anything that is a valid identifier. For e.g `*myargs` is perfectly valid.

What is **kwargs ?

`**kwargs` allows us to pass variable number of keyword argument like this
`func_name(name='tim', team='school')`

```
def my_func(**kwargs):
    for i, j in kwargs.items():
        print(i, j)

my_func(name='tim', sport='football', roll=19)
```

Expected Output:

```
sport football
roll 19
name tim
```

Using `*args` and `**kwargs` in function call

You can use `*args` to pass elements in an iterable variable to a function. Following example will clear everything.

```
def my_three(a, b, c):
    print(a, b, c)


a = [1,2,3]
my_three(*a) # here list is broken into three elements
```

Note: This works only when number of argument is same as number of elements in the iterable variable.

Similarly you can use `**kwargs` to call a function like this

```
def my_three(a, b, c):
    print(a, b, c)


a = {'a': "one", 'b': "two", 'c': "three" }
my_three(**a)
```

Note: For this to work 2 things are necessary:

1. Names of arguments in function must match with the name of keys in dictionary.
2. Number of arguments should be same as number of keys in the dictionary.

Chapter – 23

Python Generators

Generators are function used to create iterators, so that it can be used in the for loop.

Creating Generators

Generators are defined similar to function but there is only one difference, we use yield keyword to return value used for each iteration of the for loop. Let's see an example where we are trying to clone python's built-in `range()` function.

```
def my_range(start, stop, step = 1):
    if stop <= start:
        raise RuntimeError("start must be smaller than stop")
    i = start
    while i < stop:
        yield i
        i += step


try:
    for k in my_range(10, 50, 3):
        print(k)
except RuntimeError as ex:
    print(ex)
except:
    print("Unknown error occurred")
```

Expected Output:

```
10
13
16
19
22
25
28
31
34
```

```
37
40
43
46
49
```

Here is how `my_range()` works:

In for loop `my_range()` function get called, it initializes values of the three arguments(`start` , `stop` and `step`) and also checks whether stop is smaller than or equal to start , if it is not then `i` is assigned value of start . At this point `i` is `10` so while condition evaluates to `True` and while loop starts executing. In next statement `yield` transfer control to the for loop and assigns current value of `i` to variable `k` , inside the for loop print statement get executed, then the control again passes to line 7 inside the function `my_range()` where `i` gets incremented. This process keeps on repeating until `i < stop` .

Chapter – 24

Python Regular Expression

Regular expression is widely used for pattern matching. Python has built-in support for regular function. To use regular expression you need to import re module.

```
import re
```

Now you are ready to use regular expression.

`re.search()` Method

`re.search()` is used to find the first match for the pattern in the string.

Syntax: `re.search(pattern, string, flags[optional])`

`re.search()` method accepts pattern and string and returns a match object on success or `None` if no match is found. `match object` has `group()` method which contains the matching text in the string.

You must specify the pattern using **raw strings** i.e prepending string with `r` like this.

```
r'this \n'.
```

All the special character and escape sequences loose their special meanings in raw string so \n is not a newline character, it's just backslash \ followed by n .

```
>>> import re
>>> s = "my number is 123"
>>> match = re.search(r'\d\d\d', s)
>>> match
<_sre.SRE_Match object; span=(13, 16), match='123'>
>>> match.group()
'123'
```

above we have use `\d\d\d` as pattern. `\d` in regular expression matches a single digit, so

`\d\d\d` will match digits like 111 , 222 , 786 it will not match 12 , 1444 .

Basic patterns used in regular expression

SYMBOL	DESCRIPTION
.	dot matches any character except newline
\w	matches any word character i.e letters, alphanumeric, digits and underscore (_)
\W	matches non word characters
\d	matches a single digit
\D	matches a single character that is not a digit
\s	matches any white-spaces character like \n, \t, spaces
\S	matches single non white space character
[abc]	matches single character in the set i.e either match a, b or c
[^abc]	match a single character other than a, b and c
[a-z]	match a single character in the range a to z.
[a-zA-Z]	match a single character in the range a-z or A-Z
[0-9]	match a single character in the range 0-9
^	match start at beginning of the string
$	match start at end of the string
+	matches one or more of the preceding character (greedy match).
*	matches zero or more of the preceding character (greedy match).

Let take one more example:

```
import re
s = "tim email is tim@somehost.com"
match = re.search(r'[\w.-]+@[\w.-]+', s)


# the above regular expression will match a email address


if match:
    print(match.group())
else:
    print("match not found")
```

here we have used `[\w.-]+@[\w.-]+` pattern to match an email address. On success `re.search()` returns an `match object` , and its `group()` method will contain the matching text.

Group capturing

Group capturing allows to extract parts from the matching string. You can create groups using parentheses `()` . Suppose we want to extract username and host name from the email address in the above example. To do this we need to add `()` around username and host name like this.

```
match = re.search(r'([\w.-]+)@([\w.-]+)', s)
```

Note that parentheses will not change what the pattern will match. If the match is successful then match.group(1) will contain the match from the first parentheses and match.group(2) will contain the match from the second parentheses.

```
import re
s = "tim email is tim@somehost.com"
match = re.search('([\w.-]+)@([\w.-]+)', s)

if match:
    print(match.group()) ## tim@somehost.com (the whole match)
    print(match.group(1)) ## tim (the username, group 1)
    print(match.group(2)) ## somehost (the host, group 2)
```

`findall()` Function

As you know by now `re.search()` find only first match for the pattern, what if we want to find all matches in string, this is where `findall()` comes into the play.

Syntax: `findall(pattern, string, flags=0[optional])`

On success it returns all the matches as a list of strings, otherwise an empty list.

```
import re
s = "Tim's phone numbers are 12345-41521 and 78963-85214"
match = re.findall(r'\d{5}', s)


if match:
    print(match)
```

Expected Output:

```
['12345', '41521', '78963', '85214']
```

you can also use group capturing with `findall()` , when group capturing is applied then `findall()` returns a list of tuples where tuples will contain the matching groups. An example will clear everything.

```
import re
s = "Tim's phone numbers are 12345-41521 and 78963-85214"
match = re.findall(r'(\d{5})-(\d{5})', s)
print(match)


for i in match:
    print()
    print(i)
    print("First group", i[0])
    print("Second group", i[1])
```

Expected Output:

```
[('12345', '41521'), ('78963', '85214')]

('12345', '41521')
First group 12345
Second group 41521
```

```
('78963', '85214')
First group 78963
Second group 85214
```

Optional flags

Both `re.search()` and `re.findall()` accepts and optional parameter called flags. flags are used to modify the behavior of the pattern matching.

FLAGS	DESCRIPTION
re.IGNORECASE	Ignores uppercase and lowercase
re.DOTALL	Allows (.) to match newline, be default (.) matches any character except newline
re.MULTILINE	This will allow ^ and $ to match start and end of each line

Using `re.match()`

`re.match()` is very similar to `re.search()` difference is that it will start looking for matches at the beginning of the string.

```
import re
s = "python tuts"
match = re.match(r'py', s)
if match:
    print(match.group())
```

You can accomplish the same thing by applying `^` to a pattern with `re.search()` .

```
import re
s = "python tuts"
match = re.search(r'^py', s)
if match:
    print(match.group())
```

This completes everything you need to know about re module in python.

Chapter – 25

Installing packages in python using PIP

PIP is a package management system used to install packages from repository. You can use pip to install various software packages available on http://pypi.python.org/pypi. pip is much similar to composer in php. Pip is a recursive acronym which stands for pip installs packages.

Installing pip

Python 2.7.9 and later (python2 series), and Python 3.4 and later (python 3 series) already comes with pip.

To check your python version you need to enter the following command :

```
python -V
```

If your python version do not belong to any of above mentioned versions then you need to manually install pip (see links below) .

Click here for windows installation instructions .
Click here for linux instructions .

Installing packages

Suppose you want to install a package called requests (which is used to make HTTP requests). You need to issue the following command.

```
pip install requests # this will install latest request package
```

```
pip install requests==2.6.0 # this will install requests version 2.6.0
```

```
pip install requests>=2.6.0 # specify a minimum version
```

Note: `pip.exe` is stored under `C:\Python34\Scripts` , so you need to go there to install packages. Alternatively add the whole path to PATH environment variable. This way you can access pip from any directory.

Uninstalling packages

To uninstall the package use the command below.

```
pip uninstall package_name
```

Upgrade Package

```
pip install --upgrade package_name
```

Searching for a package

```
pip search "your query"
```

Note: You don't need to add quotes around your search term.

Listing installed packages

```
pip list
```

above command will list all the installed packages.

Listing outdated installed packages

```
pip list –outdated
```

Details about installed packages

You can use the following command to get information about a installed package, i.e Package name, version, location, dependencies.

```
pip show package_name
```

Chapter – 26

Python virtualenv Guide

Note: This tutorial need pip, if you have not already done so, first go through installing pip .

virtualenv is a tool used to separate different dependencies required by the projects. While working on multiple projects it's a common issue that one project need a version of package that is completely different from the other one, virtualenv helps us to resolve such kind of issues. It also helps to prevent polluting global site package.

Installing virtualenv

virtualenv is just a package available at pypi, you can use pip to install virtualenv.

```
pip install virtualenv
```

After installation you may need to add `C:\Python34\Scripts` to your PATH environment variable.

This way commands like pip, virtualenv will become available in any directory level.

Creating a Virtual Environment

Create a new directory called `python_project` and change current working directory to `python_project` .

```
mkdir python_project
cd python_project
```

To create a virtual environment inside `python_project` you need to issue the following command.

```
virtualenv my_env
```

This will create a new folder `my_env` inside `python_project` . This folder will contain a copy of python executables and pip library used to install packages. Here we have used `my_env` as name, but

you can use anything you want. Now your virtual environment is ready to use, you just need to activate it.

There is one point in this tutorial we have installed virtualenv using python 3.4 suppose you also have python 2.7 and want to create a virtual environment that use python 2.7 instead of 3.4, you can do so using the following command.

```
virtualenv -p c:\Python27/python.exe my_env
```

Activating Virtual Environment

If you are on windows you need to execute the following command.

```
my_env\Scripts\activate.bat
```

On Linux enter this.

```
source my_env/bin/activate
```

After issuing the above command your command prompt string will change and will look something like,

```
( my_env ) Path_to_the_project: $
```

Notice (`my_env`) , this indicates that you are now operating under virtual environment.

Now you virtual environment is activated. Anything you install here will be used by this project only.

Let's try to install requests package.

In Windows enter the following code.

```
my_env\Scripts\pip.exe install requests
```

You can't use use just `pip install requests` in windows because it would execute the global pip if you have added `C:\Python34\Scripts` to your PATH environment variable. If you have not added then you will get an error.

Similarly in Linux you need to execute the following code

```
my_env\Scripts\pip install requests
```

Deactivating Virtual Environment

To deactivate virtual environment you need to use the following command.

```
deactivate
```

This command will put you back in system's default python interpreter, where we can install the package in the global site package.

You should now able to see the motivation behind using virtualenv. It helps us to organise the needs of projects without conflicting with each other.

Chapter – 27

Python recursive functions

When a function call itself is knows as recursion. Recursion works like loop but sometimes it makes more sense to use recursion than loop. You can convert any loop to recursion.

Here is how recursion works. A recursive function calls itself. As you you'd imagine such a process would repeat indefinitely if not stopped by some condition. This condition is known as base condition. A base condition is must in every recursive programs otherwise it will continue to execute forever like an infinite loop.

Overview of how recursive function works

1. Recursive function is called by some external code.
2. If the base condition is met then the program do something meaningful and exits.
3. Otherwise, function does some required processing and then call itself to continue recursion.

Here is an example of recursive function used to calculate factorial.

Factorial is denoted by number followed by (!) sign i.e 4!

For e.g

```
4! = 4 * 3 * 2 * 1
2! = 2 * 1
0! = 1
```

Here is an example

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

```
print(fact(0))
print(fact(5))
```

Expected Output:

```
1
120
```

Now try to execute the above function like this

```
print(fact(2000))
```

You will get

```
RuntimeError: maximum recursion depth exceeded in comparison
```

This happens because python stop calling recursive function after 1000 calls by default. To change this behavior you need to amend the code as follows.

```
import sys
sys.setrecursionlimit(3000)


def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)


print(fact(2000))
```

Chapter – 28

What is if __name__ == "__main__" ??

Every module in python has a special attribute called `__name__` . The value of `__name__` attribute is set to `'__main__'` when module run as main program. Otherwise the value of `__name__` is set to contain the name of the module.

Consider the following code for better understanding.

```
# file my_module.py

foo = 100

def hello():
    print("i am from my_module.py")


if __name__ == "__main__":
    print("Executing as main program")
    print("Value of __name__ is: ", __name__)
    hello()
```

Here we have defined a new module my_module . We can execute this module as main program by

entering the following code

```
python my_module.py
```

Expected Output:

```
Executing as main program
Value of __name__ is: __main__
i am from my_module.py
```

As you can see now if statement in `my_module` fails to execute because the value of `__name__` is set to `'my_module'` .

Chapter – 29

Python Lambda Function

Python allows you to create anonymous function i.e function having no names using a facility called lambda function.

lambda functions are small functions usually not more than a line. It can have any number of arguments just like a normal function. The body of lambda functions is very small and consists of only one expression. The result of the expression is the value when the lambda is applied to an argument. Also there is no need for any return statement in lambda function.

Let's take an example:

Consider a function multiply()

```
def multiply(x, y):
    return x * y
```

This function is too small, so let's convert it into a lambda function.

To create a lambda function first write keyword lambda followed by one of more arguments separated by comma, followed by colon sign (:), followed by a single line expression.

```
r = lambda x, y: x * y
r(12, 3)   # call the lambda function
```

Expected Output:

```
36
```

Here we are using two arguments x and y , expression after colon is the body of the lambda function. As you can see lambda function has no name and is called through the variable it is assigned to.

You don't need to assign lambda function to a variable.

```
(lambda x, y: x * y)(3,4)
```

Expected Output:

```
12
```

Note that lambda function can't contain more than one expression.

Chapter – 30

Python String Formatting

`format()` method allows you format string in any way you want.

Syntax: `template.format(p1, p1, .... , k1=v1, k2=v2)`

template is a string containing format codes, format() method uses it's argument to substitute value for each format codes. For e.g

```
>>> 'Sam has {0} red balls and {1} yellow balls'.format(12, 31)
```

`{0}` and `{1}` are format codes. The format code `{0}` is replaced by the first argument

of `format()` i.e `12` , while `{1}` is replaced by the second argument of `format()` i.e `31` .

Expected Output:

```
Sam has 12 red balls and 31 yellow balls
```

This technique is okay for simple formatting but what if you want to specify precision in floating point number ? For such thing you need to learn more about format codes. Here is the full syntax of format codes.

Syntax: `{[argument_index_or_keyword]:[width][.precision][type]}`

type can be used with format codes

Format codes,	Description
d,	for integers
f,	for floating point numbers
b,	for binary numbers
o,	for octal numbers
x,	for octal hexadecimal numbers
s,	for string
e,	for floating point in exponent format

Following examples will make things more clear.

Example 1:

```
>>> "Floating point {0:.2f}".format(345.7916732)
```

Here we specify `2` digits of precision and `f` is used to represent floating point number.

Expected Output:

```
Floating point 345.79
```

Example 2:

```
>>> import math
>>> "Floating point {0:10.3f}".format(math.pi)
```

Here we specify `3` digits of precision, `10` for width and `f` for floating point number.

Expected Output:

```
Floating point 3.142
```

Example 3:

```
"Floating point pi = {0:.3f}, with {1:d} digit precision".format(math.pi, 3)
```

here `d` in `{1:d}` represents integer value.

Expected Output:

```
Floating point pi = 3.142, with 3 digit precision
```

You need to specify precision only in case of floating point numbers if you specify precision for integer ValueError will be raised.

Example 5:

```
'Sam has {1:d} red balls and {0:d} yellow balls'.format(12, 31)
```

Expected Output:

```
Sam has 31 red balls and 12 yellow balls
```

Example 6:

```
"In binary 4 is {0:b}".format(4) # b for binary, refer to Fig 1.1
```

Expected Output:

```
In binary 4 is 100
```

Example 7:

```
array = [34, 66, 12]
"A = {0}, B = {1}, C = {2}".format(*array)
```

Expected Output:

```
'A = 34, B = 66, C = 12'
```

Example 8:

```
d = {
  'hats' : 122,
  'mats' : 42
}
```

Expected Output:

```
"Sam had {hats} hats and {mats} mats".format(**d)
```

`format()` method also supports keywords arguments.

'Sam has {red} red balls and {green} yellow balls'.format(red = 12, green = 31)

Note while using keyword arguments we need to use arguments inside {} not numeric index.

You can also mix position arguments with keywords arguments

```
'Sam has {red} red balls, {green} yellow balls \
and {0} bats'.format(3, red = 12, green = 31)
```

`format()` method of formatting string is quite new and was introduced in python 2.6 . There is another old technique you will see in legacy codes which allows you to format string using % operator instead of `format()` method.

Let's take an example.

```
"%d pens cost = %.2f" % (12, 150.87612)
```

Here we are using template string on the left of `%` . Instead of `{}` for format codes we are using `%` . On the right side of `%` we use tuple to contain our values. `%d` and `%.2f` are called as format specifiers, they begin with `%` followed by character that represents the data type. For e.g `%d` format specifier is a placeholder for a integer, similarly `%.2f` is a placeholder for floating point number.

So `%d` is replaced by the first value of the tuple i.e `12` and `%.2f` is replaced by second value i.e `150.87612` .

Expected Output:

```
12 pens cost = 150.88
```

Some more examples

Example 1:

New: `"{0:d} {1:d} ".format(12, 31)`

Old: `"%d %d" % (12, 31)`

Expected Output:

```
12 31
```

Example 2:

New: `"{0:.2f} {1:.3f}".format(12.3152, 89.65431)`

Old: `"%.2f %.3f" % (12.3152, 89.65431)`

Expected Output:

```
12.32 89.654
```

Example 3:

New: `"{0:s} {1:o} {2:.2f} {3:d}".format("Hello", 71, 45836.12589, 45 )`

Old: `"%s %o %.2f %d" % ("Hello", 71, 45836.12589, 45 )`

Expected Output:

```
Hello 107 45836.13 45
```