A Quick Python Primer

Chandan Anand Purohit

Published with GitBook

Table of Contents

A Quick Python Primer

This is a short but informative book on python which intends to cover many of the language features, modules and popular libraries. This book is meant for the impatient programmers who want to quickly get acquanted with python.

Python is a free,open-source,general-purpose programming language which was started by Guido Von Rossum in the the late 1980s. Since then it is being driven by open-source contributions along with Rossum himself.

Python has a humane syntax and teaches good programming practices and hence is considered ideal for first time programmers. In its current avatar as Python 3 it includes many useful modules. Some of these are tkinter for gui, sqlite3 for database and statistics for basic statistical operations. With these libraries it is now possible to build a complete database-driven GUI application with just pure python.

Python is used in almost all areas of engineering and science. To name a few game development, embedded programming, testing, data analysis, web development and many more. Let us dive straight into this fascinating world of python.

Basic Data Types

IN THIS CHAPTER, we introduce a very small subset of Python. While small, it is broad enough to start doing interesting things right away. In the next chapters we fill in the details. We begin by using Python as a calculator that evaluates algebraic expressions. We then introduce variables as a way to “remember” results of such evaluations. Finally we show how Python works with values other than numbers: values to represent logical values true and false, text values, and lists of values.

Having seen the core types of data supported by Python, we take a step back and define precisely the concept of a data type and that of an object that stores a value of a given type. With data stored in objects, we can ignore how the data is represented and stored in the computer and work only with the abstract but familiar properties that the object’s type makes explicit. In addition to the core, built-in data types, Python comes with a large library of additional types organized into modules.

Expressions, Variables

Before we start, you need to start a new python interpreter session. Python interpreter is a software which can intrepret and run python programs. It runs through your python program line-by-line executing each one at a time. Thus you can either open up an interactive interpreter session and try out python instructions one at a time or feed your entire python program as input to the interpreter. We will be working in the interactive interpreter in this chapter.

Starting interactive interpreter

Linux:

```
$python3 [enter]
```

Windows:

- install python3
- open IDLE ide and launch interpreter or in the command line enter python followed by enter key

Once the interpreter is running you will see a prompt like

```
>>>
```

Note: It is better to use ipython console than the default python console as it provides many features including indentation out-of-the-box

You can now start typing in any python command here: Eg:

```
>>>1 + 2
3
>>>2 * 3
6
>>>2 - 3
-1
```

Here we tested +,- and * operators for addition,subtraction and multiplication respectively. The interpreter echoes the output of the expression evaluated. Let us see the division operator now.

```
>>>2 / 3

0.6666666666666666

>>>2 // 3

0
```

There are two division operators in python 3, **/** for real division (gives actual quotient) and **//** gives the truncated result.

Python also has a power operator ******

```
>>>2 ** 3

8
```

% is the modulus operator

```
>>>3.5 % 2

1.5
```

Python also has many built-in functions to perform common tasks like abs(),max(),min() etc.

```
>>> abs(-1)

1

>>>max(1,2,3)

3
```

Thus, integer(int),floating point(float) are amongst the basic data types in python. We can check type of any variable or constant like so.

```
>>>type(3.5)

class 'float'
```

Note: If you observe here everything is an object in python, including ints and floats

There are no ++ or -- operators in python but +=,-=,*=,/= are present.

Variables

It is very easy to use variables in python. You dont need to worry about types. Python takes care of that for you. It is a dynamically typed language.

```
>>> a = 10;b=20
>>>print(a,b)
10 20
```

Here we used the print( ) function to print the results on the screen. In python variable names are just names, they can be names of anything. For example the value in variable a is now 10. We can change it to anything and python will not complain.

```
>>>type(a)
class 'int'
>>>a = 2.5
>>>type(a)
class 'float'
```

Python also has a NoneType. Let us look at an example:

```
>>> print(print('hello'))
hello
None
```

In the example above, the print statement returns None and thus None gets printed. None evaluates to False when used in boolean context.

Swapping the python way

Since python is extremely humane language and because variable names are indeed just names swapping values of two variables is just a one-liner!!

```
>>>a,b = 2.4,5
>>>a
2.4
>>>b
5
>>>b,a = a,b
>>>print(a,b)
5 2.4
```

Variable names

The characters making up a variable name can be lowercase and uppercase letters from the alphabet (a through z and A through Z), the underscore character (_), and, except for the first character, digits 0 through 9:

- myList and _list are OK, but 5list is not.
- list6 and l_2 are OK, but list-3 is not.
- mylist and myList are different variable names.

Even when a variable name is “legal” (i.e., follows the rules), it might not be a “good” name. Here are some generally accepted conventions for designing good names:

- A name should be meaningful: Name price is better than name p.
- For a multiple-word name, use either the underscore as the delimiter (e.g., temp_varand interest_rate) or use camelCase capitalization (e.g., tempVar, TempVar,interestRate or InterestRate); pick one style and use it consistently throughout your program.
- Shorter meaningful names are better than longer ones.

The below names are used as reserved keywords of the Python language. You cannot use them other than as Python commands.

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

Boolean Expressions and Relational Operators

Python supports boolean types, True and False. Any expression which returns a non-zero result evaluates to True otherwise to False.

```
>>> a = 10; b = 30
>>> a == b
False
>>> a < b
True
```

Python supports logical operators `and or not` .

```
>>> (a > 5) and (b < 40)
True
```

Just like in other languages python performs lazy evaluation of compound expressions.

```
>>> a and b
30
>>> a or b
10
```

We will discuss these in more detail when we discuss decision statements.

Strings

There is no char type in python. Only strings. Thus single quotes and double quotes both can be used for strings.

```
>>>my_name = 'chandan' #same as "chandan"

>>>type(my_name)

class 'str'
```

By the way, pound or hash symbol ie # marks beginning of a comment. The '+' operator when applied to strings concatenates them.

```
>>> my_name + ' '+ 'purohit'

'chandan purohit'
```

Note that the original string did not change ie. the one in my_name. **Strings are immutable**. Thus although we can access characters of a string using indexes like my_name[1],my_name[2] we cannot modify its value as in :

```
>>>my_name[0] = 'b'

TypeError: 'str' object does not support item assignment
```

The '*' operator can be used with strings if one of the operands is an int.

```
>>>2 * 'hello'

'hellohello'
```

With the `in` operator, we can check whether a character/string appears in a string:

```
>>> 'ha' in 'chandan'

True
```

Length of a string can be got from len() function

```
>>>len('aaa') 'ss 3
```

Everything is an object in python. So are strings. We can easily know the methods available on strings through the interpreter console as:

```
>>>dir('ss') #'ss' is just a string, any object can be given to dir
```

Follow along

```
>>> s = 'chandan'

>>> s.upper()

'CHANDAN'

>>>s.count('a') #gives count of number of occurences of 'a' in 'chandan'
```

You can get help for any of these functions by

```
>>>help(s.isalnum) #gives doc about isalnum method in object s
```

split() -> splits a string based on a delimiter if provided else split on space. eg.

```
>>> s = "this is a sentence"

>>> words = s.split()
```

Here we get back list of words in the sentence. We can traverse a string in reverse too with negative indexes. Eg.

```
>>> s[-1] #gives character at the end >>> s[-2] #second last character
```

Strings also support slicing.

```
>>> s = 'hello'

>>> s[1:3]

'el'
```

Slicing syntax is string_object[start_index:stop_index+1:step] All three are optional. Eg.

```
>>> s[:3] #slice s from index 0 to 2

'hel'

>>> s[2:] #slice s from index 2 to end

'llo'

>>> s[:] #slice entire string, so gives back new copy

'hello'

>>> s[::2] # slice s in steps of 2

'hlo'

>>> s[::-1] # slice s in steps of 1 but in reverse order ie get string reverse

'olleh'
```

Experiment with this slicing syntax to become a python wizard.

Palindrome one-liner

Thanks to the powerful slicing in python checking if a string is palindrome is just one-line as:

```
>>> s == s[::-1]
False
```

Strings can be compared using the same relational operators.

```
>>> s > 'hi'
False
```

This performs a dictionary comparison of the two strings.

In Python 3 strings are unicode. So we can have our variables in any language! To get unicode representation of a character use ord(). chr() returns the character corresponding to the code.

Decision Statements

Python just like other languages includes decision statements. However there is no switch statement in python. In this chapter we discuss the if statement and its variants. But before that we need to understand how statements are associated with their respective blocks.

Code Indentation

In python it is compulsory to indent code because the indentation level decides to which block the statement belongs. There are no braces like { } to define the scope. Here is an example:

```
statement_1
  statement_1_1
  statement_1_2
  statement_1_3
    statement_1_3_1
    statement_1_3_2
  statement_1_4
```

In the above example statement_1_1 belongs to the block statement_1. statement_1_3_1 belongs to statement_1_3 block. If you know any of the conventional languages like C,Java you will realize that indentation level does the task of braces. This will become clearer as you code more.

Due to this rule, python programs are readable by default!! Adding to the humane nature of the language.

Note: You should mix spaces and tabs to give indentation. It is better to make use of an editor like sublime text, atom, notepad++ etc. which does the indentation for you

if statement

Having become familiar with the indentation rule of python let us now look into the if statement.

The syntax for the simple if statement is:

```
if expression:
  statement1
  statement2
```

Python (obviously) has an else which can be associated with the if.

```
if expression:
  statement
else:
  statement
```

Let us write a simple program to check if a number is even or odd

```
number = 10
if number % 2 == 0:
  print('even')
else:
  print('odd')
```

Store this in a file called even_odd.py. If you are using IDLE you can run with the run button or if you want you can run from the command line by changing directory to the directory where your program is stored and typing `python even_odd.py`

Python equivalent of else if is the elif statement.

```
number = 10
if number == 5:
  print('Number equal to 5')
elif number == 10:
  print('Number equal to 10')
else:
  print('Neither 5 or 10')
```

Nesting if statements is also quite easy, just notice the indentation.

```
number = 10
if number > 5:
  if number % 2 == 0:
    print('even and above 5')
  else:
    print('above 5 but not even')
else:
  print('less than 5')
```

Although switch statement and ternary operator are not present in python, there is a cool short hand notation for the simple if statement. Look at the code below:

```
 message_string = None
 a = 10
 message_string = "hello" if a == 10 else "hi"
```

In the code above, since value in a is 10, message_string will have the string "hello".

Loops

In this chapter we discuss looping constructs provided by python, their uniqueness and various ways to use them. Python has for and while loops but there is no do-while loop. The for loop is also different from the ones you may be familiar with already. Let us look into these very powerful statements.

The for loop

As mentioned the for loop in python is different. It is actually a foreach loop. Let us look at an example: How do you print your name 10 times?

```
for i in range(10):
  print('chandan')
```

- the range( ) function is a utility function which can be used to generate a set of numbers within a range. Its syntax is range(start,stop,step)
- range(10) -> generates numbers between 0 and 10 (not including 10). This is NOT exactly what happens, but we can safely assume this way for now.
- range(1,10) -> 1,2,3,..,9
- range(1,10, 2) -> 1,3,5,7,9

In the loop example above the variable i takes on the values 0,1,2,..,9 on the 1st,2nd,3rd,...,10th iteration respectively.

We can nest for loops.

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

We can easily print all characters of a string using a for loop

```
name = 'chandan'
for character in name:
  print(character)
```

This is possible because a string is an iterable object(an object through which you can iterate). We can loop through an object if it is iterable. Python has a useful utility function call zip( ) to iterate through multiple iterables simultaneously. Here is an example to iterate through two strings.

```
for c1,c2 in zip('hello','hate'):
  print(c1,c2)
```

Here is the output of the above program

```
h h
e a
l t
l e
```

One thing to notice here is that zip considers the smallest length iterable for zipping through. Thus the for loop iterates only 4 times ie. length of 'hate'. The for loop is generally preferred when the number for times the loop has to run is fixed.

The while loop

The while loop is generally preferred if the number of times we need to loop is not fixed. It is widely used in event-driven-programming where the number of loop iterations is decided by user events. Here is a simple program to print an even number as long as user does not press 'q' key.

```
start_number = 0
current_number = start_number
while(True):
  print(current_number)
  user_input = input('enter q to quit')
  if user_input == 'q':
    break
  current_number += 2
```

- input( ) -reads user input, returns user input as string
- break - used to abruptly jump out of current enclosing loop

The syntax for a while loop is

```
while expression:
  statement
```

Here value of the expression decides whether the loop body can be entered or not. A popular usage of while loop is to make infinite loop as:

```
while True:
  statement
```

Lists

In this chapter we discuss a very important data structure which is a python built-in data type "List". A list is a linear collection of data. It can store ANYTHING. Lists in python have dynamic size ie. their size is same as the number of contents in the list. Since a list is an object, it has many methods which can be used to manipulate it. Lists are mutable ie. their contents can be changed.

In this chapter, we discuss about all these features of a list and show the most common operations on a list.

List usage

Let us create a simple variable called sample_list and assign a empty list to it.

```
sample_list = []
```

Instead of creating an empty we could have created a list with some values

```
sample_list = [1,2,'hello']
```

A list can contain any python object, be it integer,string, user-defined object etc.

Finding list length

The len function gives length of a list

```
len(sample_list)
```

Accessing list elements

List elements can be accessed using index

```
sample_list[0]
```

We can also use negative indices just like wge saw for strings, to access list elements in reverse order.

```
sample_list[-1] #gives 'hello'
```

Changing list contents

It is very simple to change contents of a list using the familiar index notation. LISTS ARE MUTABLE.

```
sample_list = [1,2,3]
sample_list[1] = 'Hello'
print(sample_list) #prints [1,'Hello',3]
```

Slicing lists

Lists also support slicing, just like strings

```
sample_list[0:2] #returns new list [1,2]
sample_list[::-1] #returns new reversed list ie. ['hello',2,1]
```

Nesting lists

Lists can have anything inside, thus they can have other lists as members

```
sample_list = [[1,2,3],[2,3],1,2]
```

Now we can access the first element in the second nested list

```
sample_list[1][0] # gives 2
sample_list[1] # gives [2,3]
```

Looping through a list

Since lists are iterables we can use the for loop syntax with 'in' keyword to loop through

```
for element in sample_list:
  print(element)
```

Program to sum a list of numbers

```
sum = 0
numbers = [2,3,5,6]
for number in numbers:
  sum += number
print(sum)
```

Although this program works we can use the sum( ) function straightaway

```
numbers = [2,5,6,7]
print(sum(numbers))
```

Appending new elements to a list

We can append new elements to a list using the append( ) method

```
sample_list = [3,4,6]
sample_list.append(20)
print(sample_list) #prints [3,4,6,20]
```

The append( ) method takes object as argument and inserts it at the end of the list

Inserting elements into a list (at any position)

The insert( ) method which is present in all list objects can be used to insert an element anywhere in the list.

```
sample_list = [2,3,4]
sample_list.insert(1,'hello')
print(sample_list) #prints [2,'hello',3,4]
```

The insert method takes 2 arguments,

- index where element has to be inserted
- element to insert

Deleting elements from a list

There are two ways in which we can delete elements from a list

1. Using remove( ) method: We can use the remove( ) method of the list object to locate and remove an element(first occurrence) from a list. eg. `sample_list.remove(2)`
2. Using del: If we know the index of the element we can use del. Eg. `del sample_list[1]`

There are many more methods in the list object. You can refer the documentation.

List comprehension

It is quite common to have problems where we want to generate and return a new list based on some computation. List comprehensions help us do that in a much easier way. Let us see an example: **Write a program which gives list of even numbers from a list of numbers** Here is an implementation without using comprehensions

```
even_numbers = []
given_numbers = [2,3,4,5,6,7]
for number in given_numbers:
  if number % 2 == 0:
    even_numbers.append(number)

print(even_numbers)
```

Here is the same program using list comprehension.

```
even_numbers = [number for number in given_numbers if number % 2 == 0]
print(even_numbers)
```

List comprehension can be used if:

- A new list needs to generated
- Nesting level of for loops is less than or equal to 2
- Only one if statement needs to be evaluated

Sorting list

If a list contains only numbers or strings we can use the sort( ) method of list object to sort the list.

```
sample_list = [3,6,1,2]
sample_list.sort() # default ascending sort
print(sample_list) #prints [1,2,3,6]
sample_list.sort(reverse=True) #descending order sort
```

In case of list of strings, lexicographical ordering of character used for sorting.

We can also define how a particular list needs to be sorted by defining a comparison function and assigning it to the 'key' parameter of sort( ) method. We will discuss this in our chapter on functions.

Searching a list

We can use the `in` keyword just like we did for strings to search a list for an element. Here is a simple example to search a word in a list.

```
sample_list = ['hello','hi','chandan']
if 'chandan' in sample_list:
  print("found word")
else:
  print('word not found')
```

Tuples

Tuples are immutable data structures in python. Curly braces '(' and ')' are used for tuples. Once a tuple is created we cannot change its contents.

```
sample_tuple = (1,2,3)
sample_tuple[2] = 'Hello' #error
```

Accessing elements

Tuple elements can be accessed just like list and string elements, using indices

```
sample_tuple = (11,12,14,15)
print(sample_tuple[2]) # prints 14
```

Slicing

Tuples also support slicing and just like in strings and lists, a new tuple is created.

```
sample_tuple = (33,44,55,66,77)
print(sample_tuple[:2]) #prints (33,44)
print(sample_tuple[1:5:2]) #prints (44,66)
```

Counting number of occurrences

Tuple objects have a count( ) method which can be used to count the number of occurrences of an element.

```
sample_tuple = (1,1,3,3,5,5,5)
print(sample_tuple.count(5)) # prints 3
```

Getting position of an element

Use index( ) method of tuple object to get the index of an element

```
sample_tuple = (2,3,4)
print(sample_tuple.index(4)) # prints 2
```

Unpacking a tuple

Here is an example of unpacking a tuple

```
number_1, number_2 = (2,3)
print(number_1,number_2) #prints 2 3
```

If number of values in the tuple is more than number of variables, then we need to use a *

```
numbers = (3,4,5,6,7)
first_number, *other_numbers = numbers
print(first_number,other_numbers) # prints 3 [4,5,6,7]
```

If you observe the output other_numbers is **list** of all the other numbers except first one.

Converting to tuple

We can convert a string to a tuple with the tuple( ) function

```
numbers = '23456'
numbers_tuple = tuple(numbers)
print(numbers_tuple) # prints (2,3,4,5,6)
```

Similarly, we can convert a list to a tuple

```
numbers_list = tuple([3,4,5])
```

Sorting

Since tuples are immutable in-place sorting cannot be done. However, we can used the sorted( ) function to get back a new sorted list containing the tuple contents

```
numbers = (4,3,2,1)
sorted_numbers_list = sorted(numbers) # default ascending sort
print(sorted_numbers_list)
```

Searching

Tuples support searching just like lists using the `in` keyword

```
print('hi' in ('hello','hi')) #Prints True
```

Sets

Sets are again important data structures which do not allow duplicate values. Let us look at an example:

```
sample_set = set([1,2,3,3,3,4]) #dont use { } here. it creates a dictionary not set!
print(sample_set) # prints {1,2,3,4}
```

In the example above we create a new set from a list ie. [1,2,3,3,3,4]. However, the new set only has 1,2,3,4. The duplicates have not been saved in the set. SETS DO NOT ALLOW DUPLICATES.

Common set operations

Here are some example which show common set operations:

```
a = set('abracadabra')
b = set('alacazam')
print(a)   # unique letters in a

print(a - b)    # letters in a but not in b {'r', 'd', 'b'}
print(a | b)   # letters in either a or b {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
print(a & b)    # letters in both a and b {'a', 'c'}
print(a ^ b)   # letters in a or b but not both {'r', 'd', 'b', 'm', 'z', 'l'}
```

Searching

Similar to lists, tuples, strings the `in` keyword

```
print('a' in {'a','b','c'}) #prints True
```

Set comprehension

Similar to list comprehension we have set comprehension

```
a = {x for x in 'abracadabra' if x not in 'abc'}
print(a) # prints {'r','d'}
```

Dictionaries

A dictionary is an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary).

Creating empty dictionary

A pair of empty flower braces creates an empty dictionary

```
empty_dict = { }
```

A simple dictionary example

Here is a simple example using dictionaries

```
sample_dict = {'name':'sathvik','estd':2015} # creates a dictionary
print(sample_dict['name']) # prints sathvik
print(sample_dict.get('estd')) # prints 2015
```

As shown we can get a value giving the key to the dictionary. We can either use [] or the get( ) method. Here are some more examples:

```
print(sample_dict.keys()) # prints ['name','estd']
print(sample_dict.values()) # prints ['sathvik',2015]
print('name' in sample_dict) # prints True
```

Using dict( ) constructor

The `dict( )` constructor can be used to create a dictionary from sequences of values. Here are some examples:

```
my_dict1 = dict([('name', 'chandan'), ('age', 30)])
my_dict2 = dict(name='chandan',age=30)
```

Dictionary comprehension

Python 3 also supports dictionary comprehension. It is similar to list and set comprehension except that it now applies to a dictionary. Here is an example: Suppose we have a list of dictionaries, each dictionary containing student data. Suppose we have to pick only those students whose marks is above some threshold.

```
students = [{'marks':20},
            {'marks':21},
            {'marks':20},
            {'marks':23}]

# return all students whose marks is greater than 20
students_filtered = {key:value for key,value for students.items() if value > 20}
print(students_filtered)
```

Functions

Everything is an object in python. So are functions. Functions help us(programmers) organize our code properly. We should remember that each function must have **single** responsibility. Defining functions in python is easy as shown:

```
def addTwo(a,b): #definition
  return a+b

sum = addTwo(3,4) #function call
print(sum)
```

In the above example variables `a` and `b` are local variables, they are local to the function addTwo. Let us consider
another example to illustrate this concept of local variables.

```
number1 = 10
number2 = 20
numbers = [[2,3],2,3,4]

def changeNumber(number1, number2):
  number1 = 30
  number2 = 40
  print('in function changeNumber, number1 = ',number1,'number2= ',number2)

def changeNumbers(numbers):
  numbers[1] = 222


changeNumber(number1,number2)

print('Outside function changeNumber, number1 = ',number1,'number2 = ',number2)

changeNumbers(numbers)
print('After executing changeNumbers ',numbers)
```

This program prints the following output:

```
in function changeNumber, number1 =  30 number2=  40
Outside function changeNumber, number1 =  10 number2 =  20
After executing changeNumbers  [[2, 3], 222, 3, 4]
```

If you observe the output here, values of number1 and number2 have remained the same after executing changeNumber. This shows variables number1 and number2 are local to the function changeNumber. They are passed by value ie. copies of those variables are passed.

On the other hand, numbers is a list and thus it gets passed **by reference**. So changes made inside `changeNumbers` function are visible outside too.

Default arguments

Python allows us to assign default values to arguments of a function. Let us consider an example:

```
def addTwo(a=10,b=20):
  return a+b

print(addTwo()) # adds 10 and 20, prints 30
print(addTwo(20)) # adds 20 and 20, prints 40
print(addTwo(30,30) # adds 30 and 30, prints 60
```

As shown in the example, since both a and b have default values assigned, the call `addTwo()` just adds the default values of a and b. Similarly, the call `addTwo(20)` assigns the value 20 to a and b uses default value, ie 20.

Remember 2 rules when using default arguments:

- non-default argument cannot come after default argument. eg. `def addTwo(a=10,b): #error`
- Python does not support polymorphism the way c++ or java supports ie. we cant have 2 functions with the same name within the same scope. If given, the last function definition will override all others above it.

Recursion

A function can call itself. Here is an example of such a recursive function to calculate the nth fibbonacci number.

```
def fib(n):
  if n == 1:
    return 0
  elif n == 2:
    return 1
  else:
    return fib(n-1) + fib(n-2)
```

In the function above we have made use of the fact that the nth fibonacci number is equal to sum of (n-1)th and (n-2)th fibonacci number

Nesting of functions

We can also nest functions ie. one function definition can be inside another. Here is an example:

```
def outerFunction(s):
  def innerFunction(s):
    print(s)
  innerFunction(s)

outerFunction('sathvik softech')
```

Classes and Objects

Creating a class in python is pretty simple. Here is an example which creates a class called Student.

```
class Student:
  pass
```

To create an instance of the class, just use the class as of it were a function, as shown.

```
student1 = Student()
```

Now we have a new object called student1 which is an instance of Student class. We can give this student a name and age,

```
student1.name = "sathvik"
student1.age = 22
```

However, the name and age attributes are present in only the student1 object, not in all objects created as students. Since every student has a name and age, we can make those attributes part of the class.

```
class Student:
  def __init__(self,name,age):
    self.name = name
    self.age = age
```

Now we can start creating new students with different names and ages as;

```
student1 = Student('sathvik',22)
student2 = Student('chandan',2)
print("Student1 data ",student1.age,student1.name)
print("Student2 data ",student2.age,student2.name);
```

Every method in a class is passed a reference to the instance by default, and thus we need to have a parameter to hold it. We have used 'self' in our example to hold the reference to the current instance.

The __init__ method behaves like a constructor of the Student class, which takes the name and age, and creates a new student object with the given name and age.

Class Variables and Methods

Python also supports creation of class variables. Here is an example for the same

```
class Student:
  count = 0
  def __init__(self):
    Student.count += 1
    print(str(Student.count) + ' students created so far')

s1 = Student() //prints 1 students created so far
s2 = Student() //prints 2 students created so far
```

In this example, we have a Student class which contains an attribute called **count**. This is a class variable. We can access it through the class as `Student.count`. We have incremented this count value in the __init__ function. Class variables are common to all objects. Thus, if a class variable is modified in a object all other objects will also see the modified value.

Class methods

Class methods or static methods can be created using the @staticmethod decorator as shown in the following example:

```
class Student:
  count = 0
  def __init__(self):
    Student.count += 1

  @staticmethod
  def getStudentCount(self):
    print(Student.count)

s1 = Student() //prints 1 students created so far
s2 = Student() //prints 2 students created so far

Student.getStudentCount() //prints 2
```

Just like in other object oriented programming languages, class variables and methods can be accessed via the class names themselves.

Inheritance

Let us consider a simple example to illustrate inheritance. Suppose we have a Animal class defined as follows:

```
class Animal:
  def __init__(self,name,weight,age):
    self.name = name
    self.weight = weight
    self.age = age
  def set_name(self, new_name):
    self.name = new_name
```

If we create a new class Tiger which is a Animal as shown:

```
class Tiger(Animal): # Tiger is-a Animal
  pass
```

Here we have used inheritance to say that Tiger is a Animal. Now, we can access methods and variables from the Animal class.

```
t = Tiger('sathvik',60,15)
print(t.age,t.name,t.weight) #60 sathvik 15
```

Although Tiger class does not define set_name method, age,name and weight attributes, it has these due to inheritance.

Python also supports multiple inheritance, here is an example, Suppose we have a class called Vehicle, LandVehicle and WaterVehicle are its children. Suppose we want a new vehicle AmphibiousVehicle which is both a land and water vehicle. Here is how our class definitions may look like:

```
class Vehicle:
  def __init__(self,name,number):
    self.name = name
    self.number = number

class LandVehicle(Vehicle):
  def __init__(self,name,number):
    super().__init__(name,number)

class WaterVehicle(Vehicle):
  def __init__(self,name,number):
    super().__init__(name,number)

class AmphibiousVehicle(LandVehicle,WaterVehicle):
  def __init__(self,name,number):
    super().__init__(name,number)


am = AmphibiousVehicle('hoverBoat',2233)
```

We will not talk about the potential flaws in this implementation, but nevertheless, this is how multiple inheritance is possible in python.

Operator Overloading

Python also supports operator overloading. For example, suppose we have a list of students and we want to be able to use the + and - operators to add and subtract students. Why? Well, our definition of adding students is that we add only the marks. So,

```
class Student:
  def __init__(self,marks):
    self.marks = marks

s1 = Student(20)
s2 = Student(30)
print(s1 + s2) #error, + operator does not support adding of Student objects
```

If we really want this, we need to overload the __add__ method in the Student class.Here is how to do it,

```
class Student:
  def __init__(self,marks):
    self.marks = marks
  def __add__(self,st):
    return self.marks + st.marks

s1 = Student(20)
s2 = Student(30)
print(s1 + s2) #prints 50
```

Similarly we can overload other operators like *,== etc by implementing the __mul__,__eq__ etc.

Modules

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable **name**. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the **name** set to "**main**". That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file. **If the module is imported, the code is not run.**

File I/O

open() returns a file object, and is most commonly used with two arguments: open(filename, mode).

```
>>> f = open('workfile', 'w')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. mode can be 'r' when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The mode argument is optional; 'r' will be assumed if it’s omitted.

Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent. 'b' appended to the mode opens the file in binary mode: now the data is read and written in the form of bytes objects. This mode should be used for all files that don’t contain text.

To read a file’s contents, call f.read(size), which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). size is an optional numeric argument. When size is omitted or negative, the entire contents of the file will be read and returned; it’s your problem if the file is twice as large as your machine’s memory. Otherwise, at most size bytes are read and returned. If the end of the file has been reached, f.read() will return an empty string ('').

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use list(f) or f.readlines().

f.write(string) writes the contents of string to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

It is good practice to use the with keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent try-finally blocks:

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Saving structured data with json

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the read() method only returns strings, which will have to be passed to a function like int(), which takes a string like '123' and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called json can take Python data hierarchies, and convert them to string representations; this process is called serializing. Reconstructing the data from the string representation is called deserializing. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

If you have an object x, you can view its JSON string representation with a simple line of code:

```
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the dumps() function, called dump(), simply serializes the object to a text file. So if f is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

To decode the object again, if f is a text file object which has been opened for reading:

```
x = json.load(f)
```

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the json module contains an explanation of this.