**Variables and Datatypes**

Every programming language is having own grammar rules just like the other languages we speak.

**Keywords and Identifiers**

Python codes can be divided into identifiers. Identifiers (also referred to as names) are described by the following lexical definitions:

identifier ::= (letter|"\_") (letter | digit | "\_")\*

letter ::= lowercase | uppercase

lowercase ::= "a"..."z"

uppercase ::= "A"..."Z"

digit ::= "0"..."9"

This means *\_abcd* is a valid identifier where as *1sd* is not. The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

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

In Python we don’t specify what kind of data we are going to put in a variable. So you can directly write abc = 1 and abc will become an integer datatype. If you write abc = 1.0 abc will become of floating type. Here is a small program to add two given numbers

**>>>** a = 13

**>>>** b = 23

**>>>** a + b

36

From the above example you can understand that to declare a variable in Python , what you need is just to type the name and the value. Python can also manipulate strings They can be enclosed in single quotes or double quotes like

**>>>** 'India'

'India'

**>>>** 'India**\'**s best'

"India's best"

**>>>** "Hello World!"

'Hello World!'

Reading input from the Keyboard

Generally the real life Python codes do not need to read input from the keyboard. In Python we use raw\_input function to do input. *raw\_input(“String to show”)* , this will return a string as output. Let us write a program to read a number from the keyboard and check if it is less than 100 or not. Name of the program is testhundred.py

*#!/usr/bin/env python*

number = int(raw\_input("Enter an integer: "))

**if** number < 100:

**print** "Your number is smaller than 100"

**else**:

**print** "Your number is greater than 100"

The output

$ ./testhundred.py

Enter an integer: 13

Your number is smaller than 100

$ ./testhundred.py

Enter an integer: 123

Your number is greater than 100

In the next program we are going to calculate investments.

*#!/usr/bin/env python*

amount = float(raw\_input("Enter amount: "))

inrate = float(raw\_input("Enter Interest rate: "))

period = int(raw\_input("Enter period: "))

value = 0

year = 1

**while** year <= period:

value = amount + (inrate \* amount)

**print** "Year *%d* Rs. *%.2f*" % (year, value)

amount = value

year = year + 1

The output

$ ./investment.py

Enter amount: 10000

Enter Interest rate: 0.14

Enter period: 5

Year 1 Rs. 11400.00

Year 2 Rs. 12996.00

Year 3 Rs. 14815.44

Year 4 Rs. 16889.60

Year 5 Rs. 19254.15

Some Examples

Some examples of variables and datatypes:

Average of N numbers

In the next program we will do an average of N numbers.

n = 10

sum = 0

count = 0

while count < n:

number = float(raw\_input(""))

sum = sum + number

count = count + 1

average = float(sum)/n

print "N = *%d* , Sum = *%f*" % (N, sum)

print "Average = *%f*" % average

Temperature conversion

In this program we will convert the given temperature to Celsius from Fahrenheit by using the formula C=(F-32)/1.8

fahrenheit = 0.0

print "Fahrenheit Celsius"

while fahrenheit <= 250:

celsius = ( fahrenheit - 32.0 ) / 1.8 *# Here we calculate the Celsius value*

print "*%5.1f* *%7.2f*" % (fahrenheit , celsius)

fahrenheit = fahrenheit + 25

Multiple assignments in a single line

You can even assign values to multiple variables in a single line, like

**>>>** a , b = 45, 54

**>>>** a

45

**>>>** b

54

Using this swapping two numbers becomes very easy

**>>>** a, b = b , a

**>>>** a

54

**>>>** b

45

To understand how this works, you will have to learn about a data type called *tuple*. We use *comma* to create tuple. In the right hand side we create the tuple (we call this as tuple packing) and in the left hand side we do tuple unpacking into a new tuple.

Below we have another example of tuple unpacking.

**>>>** data = ("John Adams", "USA", "Python")

**>>>** name, country, language = data

**>>>** name

'John Adams'

**>>>** country

'USA'

**>>>** language

'Python'

**Operators and expressions**

In Python most of the lines you will write will be expressions. Expressions are made of operators and operands. An expression is like *2 + 3* .

Operators

Operators are the symbols which tells the Python interpreter to do some mathematical or logical operation. Few basic examples of mathematical operators are given below:

**>>>** 2 + 3

5

**>>>** 23 - 3

20

**>>>** 22.0 / 12

1.8333333333333333

To get floating result you need to the division using any of operand as floating number. To do modulo operation use % operator

**>>>** 14 % 3

2

Example of integer arithmetic

The code

days = int(raw\_input("Enter days: "))

months = days / 30

days = days % 30

print "Months = *%d* Days = *%d*" % (months, days)

In the first line I am taking the input of days, then getting the months and days and at last printing them. You can do it in a easy way

days = int(raw\_input("Enter days: "))

print "Months = *%d* Days = *%d*" % (divmod(days, 30))

The divmod(num1, num2) function returns two values , first is the division of num1 and num2 and in second the modulo of num1 and num2.

**Relational Operators**

You can use the following operators as relational operators

Relational Operators

|  |  |
| --- | --- |
| Operator | Meaning |
| < | Is less than |
| <= | Is less than or equal to |
| > | Is greater than |
| >= | Is greater than or equal to |
| == | Is equal to |
| != | Is not equal to |

Some examples

**>>>** 1 < 2

True

**>>>** 3 > 34

False

**>>>** 23 == 45

False

**>>>** 34 != 323

True

*//* operator gives the floor division result

**>>>** 4.0 // 3

1.0

**>>>** 4.0 / 3

1.3333333333333333

**Logical Operators**

To do logical AND , OR we use *and* ,\*or\* keywords. *x and y* returns *False* if *x* is *False* else it returns evaluation of *y*. If *x* is *True*, it returns *True*.

**>>>** 1 **and** 4

4

**>>>** 1 **or** 4

1

**>>>** -1 **or** 4

-1

**>>>** 0 **or** 4

4

**Shorthand Operator**

*x op = expression* is the syntax for shorthand operators. It will be evaluated like *x = x op expression* , Few examples are

**>>>** a = 12

**>>>** a += 13

**>>>** a

25

**>>>** a /= 3

**>>>** a

8

**>>>** a += (26 \* 32)

**>>>** a

840

shorthand.py example

n = 100

a = 2

while a < n:

print "*%d*" % a

a \*= a

**Expressions**

Generally while writing expressions we put spaces before and after every operator so that the code becomes clearer to read, like

a = 234 \* (45 - 56.0 / 34)

One example code used to show expressions

a = 9

b = 12

c = 3

x = a - b / 3 + c \* 2 - 1

y = a - b / (3 + c) \* (2 - 1)

z = a - (b / (3 + c) \* 2) - 1

print "X = ", x

print "Y = ", y

print "Z = ", z

At first *x* is being calculated. The steps are like this

9 - 12 / 3 + 3 \* 2 -1

9 - 4 + 3 \* 2 - 1

9 - 4 + 6 - 1

5 + 6 - 1

11 - 1

10

Now for *y* and *z* we have parentheses, so the expressions evaluated in different way. Do the calculation yourself to check them.

**Type Conversions**

We have to do the type conversions manually. Like

float(string) -> float value

int(string) -> integer value

str(integer) or str(float) -> string representation

>>> a = 8.126768

>>> str(a)

'8.126768'

evaluateequ.py

This is a program to evaluate 1/x+1/(x+1)+1/(x+2)+ ... +1/n series upto n, in our case x = 1 and n =10

sum = 0.0

for i in range(1, 11):

sum += 1.0 / i

print "*%2d* *%6.4f*" % (i , sum)

In the line *sum += 1.0 / i* what is actually happening is *sum = sum + 1.0 / i*.

quadraticequation.py

This is a program to evaluate the quadratic equation

import math

a = int(raw\_input("Enter value of a: "))

b = int(raw\_input("Enter value of b: "))

c = int(raw\_input("Enter value of c: "))

d = b \* b - 4 \* a \* c

if d < 0:

print "ROOTS are imaginary"

else:

root1 = (-b + math.sqrt(d)) / (2.0 \* a)

root2 = (-b - math.sqrt(d)) / (2.0 \* a)

print "Root 1 = ", root1

print "Root 2 = ", root2

salesmansalary.py

In this example we are going to calculate the salary of a camera salesman. His basic salary is 1500, for every camera he will sell he will get 200 and the commission on the month’s sale is 2 %. The input will be number of cameras sold and total price of the cameras.

basic\_salary = 1500

bonus\_rate = 200

commision\_rate = 0.02

numberofcamera = int(raw\_input("Enter the number of inputs sold: "))

price = float(raw\_input("Enter the total prices: "))

bonus = (bonus\_rate \* numberofcamera)

commission = (commission\_rate \* numberofcamera \* price)

print "Bonus = *%6.2f*" % bonus

print "Commission = *%6.2f*" % commission

print "Gross salary = *%6.2f*" % (basic\_salary + bonus + commission)

**If-else , the control flow**

While working on real life of problems we have to make decisions. Decisions like which camera to buy or which cricket bat is better. At the time of writing a computer program we do the same. We make the decisions using if-else statements, we change the flow of control in the program by using them.

**If statement**

The syntax looks like

if expression:

do this

If the value of expression is true (anything other than zero), do the what is written below under indentation. Please remember to give proper indentation, all the lines indented will be evaluated on the True value of the expression. One simple example is to take some number as input and check if the number is less than 100 or not.

number = int(raw\_input("Enter a number: "))

if number < 100:

print "The number is less than 100"

**Else statement**

Now in the above example we want to print “Greater than” if the number is greater than 100. For that we have to use the else statement. This works when the [\*](http://pymbook.readthedocs.org/en/latest/ifelse.html#id1)if\*statement is not fulfilled.

number = int(raw\_input("Enter a number: "))

if number < 100:

print "The number is less than 100"

else:

print "The number is greater than 100"

Another very basic example

>>> x = int(raw\_input("Please enter an integer: "))

>>> if x < 0:

... x = 0

... print 'Negative changed to zero'

... elif x == 0:

... print 'Zero'

... elif x == 1:

... print 'Single'

... else:

... print 'More'

**Truth value testing**

The elegant way to test Truth values is like

if x:

pass

Warning

Don’t do this

if x == True:

pass

**Looping**

In the examples we used before , sometimes it was required to do the same work couple of times. We use a counter to check how many times the code needs to be executed. This technique is known as looping. First we are going to look into while statement for looping.

**While loop**

The syntax for while statement is like

while condition:

statement1

statement2

The code we want to reuse must be indented properly under the while statement. They will be executed if the condition is true. Again like in if-else statement any non zero value is true. Let us write a simple code to print numbers 0 to 10

>>> n = 0

>>> while n < 11:

... print n

... n += 1

...

In the first line we are setting n = 0, then in the while statement the condition is n < 11, that means what ever line indented below that will execute until n becomes same or greater than 11. Inside the loop we are just printing the value of n and then increasing it by one.

**Fibonacci Series**

Let us try to solve Fibonacci series. In this series we get the next number by adding the previous two numbers. So the series looks like 1,1,2,3,5,8,13 .......

a, b = 0, 1

while b < 100:

print b

a, b = b, a + b

In the first line of the code we are initializing a and b, then looping while b’s value is less than 100. Inside the loop first we are printing the value of b and then in the next line putting the value of b toa and a + b to b in the same line.

If you put a trailing comma in the print statement , then it will print in the same line

a, b = 0, 1

while b < 100:

print b,

a, b = b, a + b

**Power Series**

Let us write a program to evaluate the power series. The series looks like e\*\*x =1+x+x\*\*2/2! +x\*\*3/3! +....+ x\*\*n/n! where 0 < x < 1

x = float(raw\_input("Enter the value of x: "))

n = term = num = 1

sum = 1.0

while n <= 100:

term \*= x / n

sum += term

n += 1

if term < 0.0001:

break

print "No of Times= *%d* and Sum= *%f*" % (n, sum)

In this program we introduced a new keyword called break. What break does is stop the innermost loop. In this example we are using break under the if statement

if term < 0.0001:

break

This means if the value of term is less than 0.0001 then get out of the loop.

**Multiplication Table**

In this example we are going to print the multiplication table up to 10.

i = 1

print "-" \* 50

while i < 11:

n = 1

while n <= 10:

print "*%4d*" % (i \* n),

n += 1

print ""

i += 1

print "-" \* 50

Here we used one while loop inside another loop, this is known as nested looping. You can also see one interesting statement here

print "-" \* 50

In a print statement if we multiply the string with an integer n , the string will be printed n many times. Some examples

>>> print "\*" \* 10

\*\*\*\*\*\*\*\*\*\*

>>> print "#" \* 20

####################

>>> print "--" \* 20

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

>>> print "-" \* 40

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

Some printing \* examples

Here are some examples which you can find very often in lab reports

Design 1

row = int(raw\_input("Enter the number of rows: "))

n = row

while n >= 0:

x = "\*" \* n

print x

n -= 1

Design 2

n = int(raw\_input("Enter the number of rows: "))

i = 1

while i <= n:

print "\*" \* i

i += 1

Design 3

row = int(raw\_input("Enter the number of rows: "))

n = row

while n >= 0:

x = "\*" \* n

y = " " \* (row - n)

print y + x

n -= 1

**Lists**

**List datastructure**

We are going to learn a data structure called list before we go ahead to learn more on looping. Lists can be written as a list of comma-separated values (items) between square brackets.

**>>>** a = [ 1, 342, 2233423, 'India', 'Fedora']

**>>>** a

[1, 342, 2233423, 'India', 'Fedora']

Lists can keep any other data inside it. It works as a sequence too, that means

**>>>** a[0]

1

**>>>** a[4]

'Fedora'

You can even slice it into different pieces, examples are given below

**>>>** a[4]

'Fedora'

**>>>** a[-1]

'Fedora'

**>>>** a[-2]

'India'

**>>>** a[0:-1]

[1, 342, 2233423, 'India']

**>>>** a[2:-2]

[2233423]

**>>>** a[:-2]

[1, 342, 2233423]

**>>>** a[0::2]

[1, 2233423, 'Fedora']

In the last example we used two :(s) , the last value inside the third brackets indicates step. s[i:j:k]means slice of s from i to j with step k.

To check if any value exists within the list or not you can do

>>> a = ['Fedora', 'is', 'cool']

>>> 'cool' in a

True

>>> 'Linux' in a

False

That means we can use the above statement as if clause expression. The built-in function len() can tell the length of a list.

**>>>** len(a)

3

Note

If you want to test if the list is empty or not, do it like this

if list\_name: *#This means the list is not empty*

pass

else: *#This means the list is empty*

pass

**For loop**

There is another to loop by using for statement. In Python the for statement is different from the way it works in C. Here for statement iterates over the items of any sequence (a list or a string). Example given below

>>> a = ['Fedora', 'is', 'powerfull']

>>> for x in a:

... print x,

...

Fedora is powerfull

We can also do things like

>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> for x in a[::2]:

... print x

...

**range() function**

range() is a built-in function. From the help document

range([start,] stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers. range(i, j) returns

[i, i+1, i+2, ..., j-1]; start (!) defaults to 0.

When step is given, it specifies the increment (or decrement).

For example, range(4) returns [0, 1, 2, 3]. The end point is omitted! These are exactly the valid indices for a list of 4 elements. Now if you want to see this help message on your system type help(range) in the Python interpreter.help(s) will return help message on the object s. Examples of range function

**>>>** range(1, 5)

[1, 2, 3, 4]

**>>>** range(1, 15, 3)

[1, 4, 7, 10, 13]

**>>>** range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

**Continue statement**

Just like break we have another statement, continue, which skips the execution of the code after itself and goes back to the start of the loop. That means it will help you to skip a part of the loop. In the below example we will ask the user to input an integer, if the input is negative then we will ask again, if positive then we will square the number. To get out of the infinite loop user must input 0.

while True:

n = int(raw\_input("Please enter an Integer: "))

if n < 0:

continue *#this will take the execution back to the starting of the loop*

elif n == 0:

break

print "Square is ", n \*\* 2

print "Goodbye"

**Game of sticks**

This is a very simple game of sticks. There are 21 sticks, first the user picks number of sticks between 1-4, then the computer picks sticks(1-4). Whoever will pick the last stick will lose. Can you find out the case when the user will win ?

sticks = 21

print "There are 21 sticks, you can take 1-4 number of sticks at a time."

print "Whoever will take the last stick will loose"

while True:

print "Sticks left: " , sticks

sticks\_taken = int(raw\_input("Take sticks(1-4):"))

if sticks == 1:

print "You took the last stick, you loose"

break

if sticks\_taken >= 5 or sticks\_taken <= 0:

print "Wrong choice"

continue

print "Computer took: " , (5 - sticks\_taken) , "n\n"

sticks -= 5

**Data Structures**

Python is having a few built-in data structure. If you are still wondering what is a data structure, then it is nothing a but a way to store data and the having particular methods to retrieve or manipulate it. We already saw lists before, now we will go in depth.

**Lists**

**>>>** a = [23, 45, 1, -3434, 43624356, 234]

**>>>** a.append(45)

**>>>** a

[23, 45, 1, -3434, 43624356, 234, 45]

At first we created a list *a*. Then to add *45* at the end of the list we call *a.append(45)* method. You can see that *45* added at the end of the list. Sometimes it may require to insert data at any place within the list, for that we have *insert()* method.

**>>>** a.insert(0, 1) *# 1 added at the 0th position of the list*

**>>>** a

[1, 23, 45, 1, -3434, 43624356, 234, 45]

**>>>** a.insert(0, 111)

**>>>** a

[111, 1, 23, 45, 1, -3434, 43624356, 234, 45]

*count(s)* will return you number of times *s* is in the list. Here we are going to check how many times *45* is there in the list.

**>>>** a.count(45)

2

If you want to remove any particular value from the list you have to use *remove()* method.

**>>>** a.remove(234)

**>>>** a

[111, 1, 23, 45, 1, -3434, 43624356, 45]

Now to reverse the whole list

**>>>** a.reverse()

**>>>** a

[45, 43624356, -3434, 1, 45, 23, 1, 111]

We can store anything in the list, so first we are going to add another list *b* in *a*, then we will learn how to add the values of *b* into *a*.

**>>>** b = [45, 56, 90]

**>>>** a.append(b)

**>>>** a

[45, 43624356, -3434, 1, 45, 23, 1, 111, [45, 56, 90]]

**>>>** a[-1]

[45, 56, 90]

**>>>** a.extend(b) *#To add the values of b not the b itself*

**>>>** a

[45, 43624356, -3434, 1, 45, 23, 1, 111, [45, 56, 90], 45, 56, 90]

**>>>** a[-1]

90

Above you can see how we used *a.extend()* method to extend the list. To sort any list we have *sort()*method.

**>>>** a.sort()

**>>>** a

[-3434, 1, 1, 23, 45, 45, 45, 56, 90, 111, 43624356, [45, 56, 90]]

You can also delete element at any particular position of the list using the del keyword.

>>> del a[-1]

>>> a

[-3434, 1, 1, 23, 45, 45, 45, 56, 90, 111, 43624356]

**Using lists as stack and queue**

Stacks are often known as LIFO (Last In First Out) structure. It means the data will enter into it at the end , and the last data will come out first. The easiest example can be of couple of marbles in an one side closed pipe. So if you want to take the marbles out of it you have to do that from the end where you entered the last marble. To achieve the same in code

**>>>** a

[1, 2, 3, 4, 5, 6]

**>>>** a.pop()

6

**>>>** a.pop()

5

**>>>** a.pop()

4

**>>>** a.pop()

3

**>>>** a

[1, 2]

**>>>** a.append(34)

**>>>** a

[1, 2, 34]

We learned a new method above *pop()*. *pop(i)* will take out the ith data from the list.

In our daily life we have to encounter queues many times, like in ticket counters or in library or in the billing section of any supermarket. Queue is the data structure where you can append more data at the end and take out data from the beginning. That is why it is known as FIFO (First In First Out).

**>>>** a = [1, 2, 3, 4, 5]

**>>>** a.append(1)

**>>>** a

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

**>>>** a.pop(0)

1

**>>>** a.pop(0)

2

**>>>** a

[3, 4, 5, 1]

To take out the first element of the list we are using *a.pop(0)*.

List Comprehensions

List comprehensions provide a concise way to create lists. Each list comprehension consists of an expression followed by a for clause, then zero or more for or if clauses. The result will be a list resulting from evaluating the expression in the context of the for and if clauses which follow it.

For example if we want to make a list out of the square values of another list, then

**>>>** a = [1, 2, 3]

**>>>** [x \*\* 2 **for** x **in** a]

[1, 4, 9]

**>>>** z = [x + 1 **for** x **in** [x \*\* 2 **for** x **in** a]]

**>>>** z

[2, 5, 10]

Above in the second case we used two list comprehensions in a same line.

**Tuples**

Tuples are data separated by comma.

>>> a = 'Fedora', 'Debian', 'Kubuntu', 'Pardus'

>>> a

('Fedora', 'Debian', 'Kubuntu', 'Pardus')

>>> a[1]

'Debian'

>>> for x in a:

... print x,

...

Fedora Debian Kubuntu Pardus

You can also unpack values of any tuple in to variables, like

**>>>** divmod(15,2)

(7, 1)

**>>>** x, y = divmod(15,2)

**>>>** x

7

**>>>** y

1

Tuples are immutable, that means you cannot del/add/edit any value inside the tuple. Here is another example

>>> a = (1, 2, 3, 4)

>>> del a[0]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object doesn't support item deletion

Above you can see Python is giving error when we are trying to delete a value in the tuple.

To create a tuple which contains only one value you have to type a trailing comma.

**>>>** a = (123)

**>>>** a

123

**>>>** type(a)

<type 'int'>

**>>>** a = (123, ) *#Look at the trailing comma*

**>>>** a

(123,)

**>>>** type(a)

<type 'tuple'>

Using the built in function *type()* you can know the data type of any variable. Remember the *len()*function we used to find the length of any sequence ?

**>>>** type(len)

<type 'builtin\_function\_or\_method'>

**Sets**

Sets are another type of data structure with no duplicate items. We can also mathematical set operations on sets.

**>>>** a = set('abcthabcjwethddda')

**>>>** a

set(['a', 'c', 'b', 'e', 'd', 'h', 'j', 't', 'w'])

And some examples of the set operations

**>>>** a = set('abracadabra')

**>>>** b = set('alacazam')

**>>>** a *# unique letters in a*

set(['a', 'r', 'b', 'c', 'd'])

**>>>** a - b *# letters in a but not in b*

set(['r', 'd', 'b'])

**>>>** a | b *# letters in either a or b*

set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])

**>>>** a & b *# letters in both a and b*

set(['a', 'c'])

**>>>** a ^ b *# letters in a or b but not both*

set(['r', 'd', 'b', 'm', 'z', 'l'])

To add or pop values from a set

**>>>** a

set(['a', 'c', 'b', 'e', 'd', 'h', 'j', 'q', 't', 'w'])

**>>>** a.add('p')

**>>>** a

set(['a', 'c', 'b', 'e', 'd', 'h', 'j', 'q', 'p', 't', 'w'])

**Dictionaries**

Dictionaries are unordered set of *key: value* pairs where keys are unique. We declare dictionaries using {} braces. We use dictionaries to store data for any particular key and then retrieve them.

**>>>** data = {'kushal':'Fedora', 'kart\_':'Debian', 'Jace':'Mac'}

**>>>** data

{'kushal': 'Fedora', 'Jace': 'Mac', 'kart\_': 'Debian'}

**>>>** data['kart\_']

'Debian'

We can add more data to it by simply

**>>>** data['parthan'] = 'Ubuntu'

**>>>** data

{'kushal': 'Fedora', 'Jace': 'Mac', 'kart\_': 'Debian', 'parthan': 'Ubuntu'}

To delete any particular *key:value* pair

>>> del data['kushal']

>>> data

{'Jace': 'Mac', 'kart\_': 'Debian', 'parthan': 'Ubuntu'

To check if any *key* is there in the dictionary or not you can use *in* keyword.

>>> 'Soumya' in data

False

You must remember that no mutable object can be a *key*, that means you can not use a *list* as a *key*. *dict()* can create dictionaries from tuples of *key,value* pair.

>>> dict((('Indian','Delhi'),('Bangladesh','Dhaka')))

{'Indian': 'Delhi', 'Bangladesh': 'Dhaka'}

If you want to loop through a dict use *iteritems()* method.

>>> data

{'Kushal': 'Fedora', 'Jace': 'Mac', 'kart\_': 'Debian', 'parthan': 'Ubuntu'}

>>> for x, y in data.iteritems():

... print "*%s* uses *%s*" % (x, y)

...

Kushal uses Fedora

Jace uses Mac

kart\_ uses Debian

parthan uses Ubuntu

Many times it happens that we want to add more data to a value in a dictionary and if the key does not exists then we add some default value. You can do this efficiently using *dict.setdefault(key, default)*.

**>>>** data = {}

**>>>** data.setdefault('names', []).append('Ruby')

**>>>** data

{'names': ['Ruby']}

**>>>** data.setdefault('names', []).append('Python')

**>>>** data

{'names': ['Ruby', 'Python']}

**>>>** data.setdefault('names', []).append('C')

**>>>** data

{'names': ['Ruby', 'Python', 'C']}

When we try to get value for a key which does not exists we get *KeyError*. We can use *dict.get(key, default)* to get a default value when they key does not exists before.

>>> data['foo']

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'foo'

>>> data.get('foo', 0)

0

If you want to loop through a list (or any sequence) and get iteration number at the same time you have to use *enumerate()*.

>>> for i, j in enumerate(['a', 'b', 'c']):

... print i, j

...

0 a

1 b

2 c

You may also need to iterate through two sequences same time, for that use *zip()* function.

>>> a = ['Pradeepto', 'Kushal']

>>> b = ['OpenSUSE', 'Fedora']

>>> for x, y in zip(a, b):

... print "*%s* uses *%s*" % (x, y)

...

Pradeepto uses OpenSUSE

Kushal uses Fedora

students.py

In this example , you have to take number of students as input , then ask marks for three subjects as ‘Physics’, ‘Maths’, ‘History’, if the total marks for any student is less 120 then print he failed, or else say passed.

n = int(raw\_input("Enter the number of students:"))

data = {} *# here we will store the data*

languages = ('Physics', 'Maths', 'History') *#all languages*

for i in range(0, n): *#for the n number of students*

name = raw\_input('Enter the name of the student *%d*: ' % (i + 1)) *#Get the name of the student*

marks = []

for x in languages:

marks.append(int(raw\_input('Enter marks of *%s*: ' % x))) *#Get the marks for languages*

data[name] = marks

for x, y in data.iteritems():

total = sum(y)

print "*%s* 's total marks *%d*" % (x, total)

if total < 120:

print "*%s* failed :(" % x

else:

print "*%s* passed :)" % x

matrixmul.py

In this example we will multiply two matrices. First we will take input the number of rows/columns in the matrix (here we assume we are using n x n matrix). Then values of the matrices.

n = int(raw\_input("Enter the value of n: "))

print "Enter values for the Matrix A"

a = []

for i in range(0, n):

a.append([int(x) for x in raw\_input("").split(" ")])

print "Enter values for the Matrix B"

b = []

for i in range(0, n):

b.append([int(x) for x in raw\_input("").split(" ")])

c = []

for i in range(0, n):

c.append([a[i][j] \* b[j][i] for j in range(0,n)])

print "After matrix multiplication"

print "-" \* 10 \* n

for x in c:

for y in x:

print "*%5d*" % y,

print ""

print "-" \* 10 \* n

Here we have used list comprehensions couple of times. *[int(x) for x in raw\_input(“”).split(” ”)]* here first it takes the input as string by *raw\_input()*, then split the result by ” ”, then for each value create one int. We are also using *[a[i][j] \* b[j][i] for j in range(0,n)]* to get the resultant row in a single line.

**Strings**

Strings are nothing but simple text. In Python we declare strings in between “” or ‘’ or ‘’’ ‘’’ or “”” “””. The examples below will help you to understand string in a better way.

>>> s = "I am Indian"

>>> s

'I am Indian'

>>> s = 'I am Indian'

>>> s = "Here is a line \

... split into two lines"

>>> s

'Here is a line split into two lines'

>>> s = "Here is a line \n split in two lines"

>>> s

'Here is a line \n split in two lines'

>>> print s

Here is a line

split in two lines

Now if you want to multiline strings you have to use triple single/double quotes.

>>> s = """ This is a

... multiline string, so you can

... write many lines"""

>>> print s

This is a

multiline string, so you can

write many lines

Different methods available for Strings

Every string object is having couple of buildin methods available, we already saw some of them likes.split(” ”).

**>>>** s = "kushal das"

**>>>** s.title()

'Kushal Das'

title() method returns a titlecased version of the string, words start with uppercase characters, all remaining cased characters are lowercase.

**>>>** z = s.upper()

**>>>** z

'KUSHAL DAS'

**>>>** z.lower()

'kushal das'

upper() returns a total uppercase version whereas lower() returns a lower case version of the string.

**>>>** s = "I am A pRoGraMMer"

>> s.swapcase()

'i AM a PrOgRAmmER'

swapcase() returns the string with case swapped.

**>>>** s = "jdwb 2323bjb"

**>>>** s.isalnum()

False

**>>>** s = "jdwb2323bjb"

**>>>** s.isalnum()

True

Because of the space in the first line isalnum() returned False , it checks for all charecters are alpha numeric or not.

**>>>** s = "SankarshanSir"

**>>>** s.isalpha()

True

**>>>** s = "Sankarshan Sir"

**>>>** s.isalpha()

False

isalpha() checks for only alphabets.

**>>>** s = "1234"

**>>>** s.isdigit() *#To check if all the characters are digits or not*

True

**>>>** s = "Fedora9 is coming"

**>>>** s.islower() *# To check if all chracters are lower case or not*

False

**>>>** s = "Fedora9 Is Coming"

**>>>** s.istitle() *# To check if it is a title or not*

True

**>>>** s = "INDIA"

**>>>** s.isupper() *# To check if characters are in upper case or not*

True

To split any string we have split(). It takes a string as an argument , depending on that it will split the main string and returns a list containing splitted strings.

**>>>** s = "We all love Python"

**>>>** s.split(" ")

['We', 'all', 'love', 'Python']

**>>>** x = "Nishant:is:waiting"

**>>>** x.split(':')

['Nishant', 'is', 'waiting']

The opposite method for split() is join(). It takes a list contains strings as input and join them.

**>>>** "-".join("GNU/Linux is great".split(" "))

'GNU/Linux-is-great'

In the above example first we are splitting the string “GNU/Linux is great” based on the white space, then joining them with “-”.

**Strip the strings**

Strings do have few methods to do striping. The simplest one is strip(chars). If you provide the chars argument then it will strip any combination of them. By default it strips only whitespace or newline characters.

**>>>** s = " abc**\n** "

**>>>** s.strip()

'abc'

You can particularly strip from the left hand or right hand side also using lstrip(chars) or rstrip(chars).

**>>>** s = "www.foss.in"

**>>>** s.lstrip("cwsd.")

'foss.in'

**>>>** s.rstrip("cnwdi.")

'www.foss'

**Finding text**

Stings have some methods which will help you in finding text/substring in a string. Examples are given below:

**>>>** s = "faulty for a reason"

**>>>** s.find("for")

7

**>>>** s.find("fora")

-1

**>>>** s.startswith("fa") *#To check if the string startswith fa or not*

True

**>>>** s.endswith("reason") *#*

True

find() helps to find the first occurrence of the substring given, if not found it returns -1.

**Palindrome checking**

Palindrome are the kind of strings which are same from left or right whichever way you read them. Example “madam”. In this example we will take the word as input from the user and say if it is palindrome or not.

s = raw\_input("Please enter a string: ")

z = s[::-1]

if s == z:

print "The string is a palindrome"

else:

print "The string is not a palindrome"

**Number of words**

In this example we will count the number of words in a given line

s = raw\_input("Enter a line: ")

print "The number of words in the line are *%d*" % (len(s.split(" ")))

**Functions**

Reusing the same code is required many times within a same program. Functions help us to do so. We write the things we have to do repeatedly in a function then call it where ever required. We already saw build in functions like *len()*, *divmod()*.

Defining a function

We use *def* keyword to define a function. General syntax is like

def functionname(params):

statement1

statement2

Let us write a function which will take two integers as input and then return the sum.

>>> def sum(a, b):

... return a + b

In the second line with the *return* keyword, we are sending back the value of *a + b* to the caller.

You must call it like

**>>>** res = sum(234234, 34453546464)

**>>>** res

34453780698L

Remember the palindrome program we wrote in the last chapter. Let us write a function which will check if a given string is palindrome or not, then return *True* or *False*.

def palindrome(s):

return s == s[::-1]

s = raw\_input("Enter a string: ")

if palindrome(s):

print "Yay a palindrome"

else:

print "Oh no, not a palindrome"

**Local and global variables**

To understand local and global variables we will go through two examples.

def change(b):

a = 90

print a

a = 9

print "Before the function call ", a

print "inside change function",

change(a)

print "After the function call ", a

First we are assigning *9* to *a*, then calling change function, inside of that we are assigning *90* to *a*and printing *a*. After the function call we are again printing the value of *a*. When we are writing *a = 90* inside the function, it is actually creating a new variable called *a*, which is only available inside the function and will be destroyed after the function finished. So though the name is same for the variable *a* but they are different in and out side of the function.

def change(b):

global a

a = 90

print a

a = 9

print "Before the function call ", a

print "inside change function",

change(a)

print "After the function call ", a

Here by using global keyword we are telling that *a* is globally defined, so when we are changing a’s value inside the function it is actually changing for the *a* outside of the function also.

**Default argument value**

In a function variables may have default argument values, that means if we don’t give any value for that particular variable it will assigned automatically.

>>> def test(a , b=-99):

... if a > b:

... return True

... else:

... return False

In the above example we have written *b = -99* in the function parameter list. That means of no value for *b* is given then b’s value is *-99*. This is a very simple example of default arguments.

You can test the code by

**>>>** test(12, 23)

False

**>>>** test(12)

True

Remember that you can not have an argument without default argument if you already have one argument with default values before it. Like *f(a, b=90, c)* is illegal as *b* is having a default value but after that *c* is not having any default value.

Also remember that default value is evaluated only once, so if you have any mutable object like list it will make a difference. See the next example

>>> def f(a, data=[]):

... data.append(a)

... return data

...

>>> print f(1)

[1]

>>> print f(2)

[1, 2]

>>> print f(3)

[1, 2, 3]

To avoid this you can write more idiomatic Python, like the following

>>> def f(a, data=None):

... if data is None:

... data = []

... data.append(a)

... return data

...

>>> print f(1)

[1]

>>> print f(2)

[2]

**Keyword arguments**

>>> def func(a, b=5, c=10):

... print 'a is', a, 'and b is', b, 'and c is', c

...

>>> func(12, 24)

a is 12 and b is 24 and c is 10

>>> func(12, c = 24)

a is 12 and b is 5 and c is 24

>>> func(b=12, c = 24, a = -1)

a is -1 and b is 12 and c is 24

In the above example you can see we are calling the function with variable names, like *func(12, c = 24)*, by that we are assigning *24* to *c* and *b* is getting its default value. Also remember that you can not have without keyword based argument after a keyword based argument. like

>>> def func(a, b=13, v):

... print a, b, v

...

File "<stdin>", line 1

SyntaxError: non-default argument follows default argument

**Docstrings**

In Python we use docstrings to explain how to use the code, it will be useful in interactive mode and to create auto-documentation. Below we see an example of the docstring for a function called *longest\_side*.

**import** **math**

def longest\_side(a, b):

*"""*

*Function to find the length of the longest side of a right triangle.*

*:arg a: Side a of the triangle*

*:arg b: Side b of the triangle*

*:return: Length of the longest side c as float*

*"""*

return math.sqrt(a\*a + b\*b)

print longest\_side(4, 5)

**Higher-order function**

Higher-order function or a functor is a function which does at least one of the following step inside:

Takes one or more functions as argument.

Returns another function as output.

In Python any function can act as higher order function.

>>> def high(func, value):

... return func(value)

...

>>> lst = high(dir, int)

>>> print lst[-3:]

['imag', 'numerator', 'real']

>>> print lst

**map function**

*map* is a very useful higher order function in Python. It takes one function and an iterator as input and then applies the function on each value of the iterator and returns a list of results.

Example:

>>> lst = [1, 2, 3, 4, 5]

>>> def square(num):

... "Returns the square of a given number."

... return num \* num

...

>>> print map(square, lst)

[1, 4, 9, 16, 25]

**File handling**

A file is some information or data which stays in the computer storage devices. You already know about different kinds of file , like your music files, video files, text files. Python gives you easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

**File opening**

To open a file we use *open()* function. It requires two arguments, first the file path or file name, second which mode it should open. Modes are like

“r” -> open read only, you can read the file but cannot edit / delete anything inside

“w” -> open with write power, means if the file exists then delete all content and open it to write

“a” -> open in append mode

The default mode is read only, ie if you do not provide any mode it will open the file as read only. Let us open a file

**>>>** fobj = open("love.txt")

**>>>** fobj

<open file 'love.txt', mode 'r' at 0xb7f2d968>

**Closing a file**

After opening a file one should always close the opened file. We use method *close()* for this.

**>>>** fobj = open("love.txt")

**>>>** fobj

<open file 'love.txt', mode 'r' at 0xb7f2d968>

**>>>** fobj.close()

Always make sure you *explicitly* close each open file, once its job is done and you have no reason to keep it open. Because - There is an upper limit to the number of files a program can open. If you exceed that limit, there is no reliable way of recovery, so the program could crash. - Each open file consumes some main-memory for the data-structures associated with it, like file descriptor/handle or file locks etc. So you could essentially end-up wasting lots of memory if you have more files open that are not useful or usable. - Open files always stand a chance of corruption and data loss.

**Reading a file**

To read the whole file at once use the *read()* method.

**>>>** fobj = open("sample.txt")

**>>>** fobj.read()

If you call *read()* again it will return empty string as it already read the whole file. readline() can help you to read one line each time from the file.

**>>>** fobj = open("sample.txt")

**>>>** fobj.readline()

'I love Python\n'

**>>>** fobj.readline()

To read all the lines in a list we use *readlines()* method.

**>>>** fobj = open("sample.txt")

**>>>** fobj.readlines()

You can even loop through the lines in a file object.

>>> fobj = open("sample.txt")

>>> for x in f:

... print x,

...

Let us write a program which will take the file name as the input from the user and show the content of the file in the console.

name = raw\_input("Enter the file name: ")

fobj = open(name)

print fobj.read()

fobj.close()

In the last line you can see that we closed the file object with the help of close() method.

**Writing in a file**

Let us open a file then we will write some random text into it by using the write() method.

**>>>** fobj = open("ircnicks.txt", 'w')

**>>>** fobj.write('powerpork**\n**')

**>>>** fobj.write('indrag**\n**')

**>>>** fobj.write('mishti**\n**')

**>>>** fobj.write('sankarshan')

**>>>** fobj.close()

Now read the file we just created

>>> fobj = open('ircnicks.txt')

>>> s = fobj.read()

>>> print s

copyfile.py

In this example we will copy a given text file to another file.

import sys

if len(sys.argv) < 3:

print "Wrong parameter"

print "./copyfile.py file1 file2"

sys.exit(1)

f1 = open(sys.argv[1])

s = f1.read()

f1.close()

f2 = open(sys.argv[2], 'w')

f2.write(s)

f2.close()

Note

This way of reading file is not always a good idea, a file can be very large to read and fit in the memory. It is always better to read a known size of the file and write that to the new file.

You can see we used a new module here *sys*. *sys.argv* contains all command line parameters. Remember *cp* command in shell, after *cp* we type first the file to be copied and then the new file name.

The first value in *sys.argv* is the name of the command itself.

import sys

print "First value", sys.argv[0]

print "All values"

for i, x in enumerate(sys.argv):

print i, x

Here we used a new function *enumerate(iterableobject)*, which returns the index number and the value from the iterable object.

**Random seeking in a file**

You can also randomly move around inside a file using *seek()* method. It takes two arguments , offset and whence. To know more about it let us read what Python help tells us

seek(...) seek(offset[, whence]) -> None. Move to new file position. Argument offset is a byte count. Optional argument whence defaults to 0 (offset from start of file, offset should be >= 0); other values are 1 (move relative to current position, positive or negative), and 2 (move relative to end of file, usually negative, although many platforms allow seeking beyond the end of a file). If the file is opened in text mode, only offsets returned by tell() are legal. Use of other offsets causes undefined behavior. Let us see one example

**>>>** fobj = open('/tmp/tempfile', 'w')

**>>>** fobj.write('0123456789abcdef')

**>>>** fobj.close()

**>>>** fobj = open('/tmp/tempfile')

**>>>** fobj.tell() *#tell us the offset position*

0L

**>>>** fobj.seek(5) *# Goto 5th byte*

**>>>** fobj.tell()

5L

**>>>** fobj.read(1) *#Read 1 byte*

'5'

**>>>** fobj.seek(-3, 2) *# goto 3rd byte from the end*

**>>>** fobj.read() *#Read till the end of the file*

'def'

**Count spaces, tabs and new lines in a file**

Let us try to write an application which will count the spaces, tabs, and lines in any given file.

import os, sys

def parse\_file(path):

*"""*

*Parses the text file in the given path and returns space, tab & new line*

*details.*

*:arg path: Path of the text file to parse*

*:return: A tuple with count of spacaes, tabs and lines.*

*"""*

fd = open(path)

i = 0

spaces = 0

tabs = 0

for i,line in enumerate(fd):

spaces += line.count(' ')

tabs += line.count('\t')

*#Now close the open file*

fd.close()

*#Return the result as a tuple*

return spaces, tabs, i + 1

def main(path):

*"""*

*Function which prints counts of spaces, tabs and lines in a file.*

*:arg path: Path of the text file to parse*

*:return: True if the file exits or False.*

*"""*

if os.path.exists(path):

spaces, tabs, lines = parse\_file(path)

print "Spaces *%d*. tabs *%d*. lines *%d*" % (spaces, tabs, lines)

return True

else:

return False

if \_\_name\_\_ == '\_\_main\_\_':

if len(sys.argv) > 1:

main(sys.argv[1])

else:

sys.exit(-1)

sys.exit(0)

You can see that we have two functions in the program , *main* and *parse\_file* where the second one actually parses the file and returns the result and we print the result in *main* function. By splitting up the code in smaller units (functions) helps us to organize the codebase and also it will be easier to write test cases for the functions.

**Exceptions**

In this chapter we will learn about exceptions in Python and how to handle them in your code.

Any error which happens during the execution of the code is an exception. Each exception generally shows some error message.

**NameError**

When one starts writing code, this will be one of the most command exception he/she will find. This happens when someone tries to access a variable which is not defined.

>>> print kushal

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'kushal' is not defined

The last line contains the details of the error message, the rest of the lines shows the details of how it happened (or what caused that exception).

**TypeError**

*TypeError* is also one of the most found exception. This happens when someone tries to do an operation with different kinds of incompatible data types. A common example is to do addition of Integers and a string.

>>> print 1 + "kushal"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

How to handle exceptions?

We use *try...except* blocks to handle any exception. The basic syntax looks like

try:

statements to be inside try clause

statement2

statement3

...

except ExceptionName:

statements to evaluated in case of ExceptionName happens

It works in the following way:

First all lines between *try* and *except* statements.

If *ExceptionName* happens during execution of the statements then *except* clause statements execute

If no exception happens then the statements inside *except* clause does not execute.

If the *Exception* is not handled in the *except* block then it goes out of *try* block.

The following examples showcase these scenarios.

>>> def get\_number():

... "Returns a float number"

... number = float(raw\_input("Enter a float number: "))

... return number

...

>>>

>>> while True:

... try:

... print get\_number()

... except ValueError:

... print "You entered a wrong value"

...

**KeyboardInterrupt**

As the first input I provided a proper float value and it printed it back, next I entered a value with a comma, so the *except* clause executed and the print statement executed.

In the third time I pressed *Ctrl+c* which caused a *KeyboardInterrupt*, which is not handled in the *except* block so the execution stopped with that exception.

An empty *except* statement can catch any exception. Read the following example:

>>> try:

... raw\_input() *# Press Ctrl+c for a KeyboardInterrupt*

... except:

... print "Unknown Exception"

...

**Raising exceptions**

One can raise an exception using *raise* statement.

>>> raise ValueError("A value error happened.")

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: A value error happened.

We can catch these exceptions like any other normal exceptions.

>>> try:

... raise ValueError("A value error happened.")

... except ValueError:

... print "ValueError in our code."

...

**Using finally for cleanup**

If we want to have some statements which must be executed under all circumstances, we can use *finally* clause, it will be always executed before finishing *try* statements.

>>> try:

... fobj = open("hello.txt", "w")

... res = 12 / 0

... except ZeroDivisionError:

... print "We have an error in division"

... finally:

... fobj.close()

... print "Closing the file object."

...

**Closing the file object.**

In this example we are making sure that the file object we open, must get closed in the *finally* clause.

**Class**

**Your first class**

Before writing your first class, you should know the syntax. We define a class in the following way.

class nameoftheclass(parent\_class):

statement1

statement2

statement3

In the statements you can write any Python statement, you can define functions (which we call methods of a class).

>>> class MyClass(object):

... a = 90

... b = 88

...

>>> p = MyClass()

>>> p

<\_\_main\_\_.MyClass instance at 0xb7c8aa6c>

In the above example you can see first we are declaring a class called MyClass, writing some random statements inside that class. After the class definition, we are creating an object p of the classMyClass. If you do a dir on that

**>>>** dir(p)

['\_\_doc\_\_', '\_\_module\_\_', 'a', 'b']

you can see the variables a and b inside it.

**\_\_init\_\_ method**

\_\_init\_\_ is a special method in Python classes, it is the constructor method for a class. In the following example you can see how to use it.

class Student(object):

*"""*

*Returns a ```Student``` object with the given name, branch and year.*

*"""*

def \_\_init\_\_(self, name, branch, year):

self.name = name

self.branch = branch

self.year = year

print "A student object is created"

def print\_details(self):

*"""*

*Prints the details of the student.*

*"""*

print "Name:", self.name

print "Branch:", self.branch

print "Year:", self.year

\_\_init\_\_ is called whenever an object of the class is constructed. That means when ever we will create a student object we will see the message “A student object is created” in the prompt. You can see the first argument to the method is self. It is a special variable which points to the current object (like this in C++). The object is passed implicitly to every method available in it, but we have to get it explicitly in every method while writing the methods. Example shown below.

>>> std1 = Student()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: \_\_init\_\_() takes exactly 4 arguments (1 given)

>>> std1 = Student('Kushal','CSE','2005')

A student object is created

In this example at first we tried to create a Student object without passing any argument and Python interpreter complained that it takes exactly 4 arguments but received only one (self). Then we created an object with proper argument values and from the message printed, one can easily understand that \_\_init\_\_ method was called as the constructor method.

Now we are going to call print\_details() method.

**>>>** std1.print\_details()

Name: Kushal

Branch: CSE

Year: 2005

**Inheritance**

In general we human beings always know about inheritance. In programming it is almost the same. When a class inherits another class it inherits all features (like variables and methods) of the parent class. This helps in reusing codes.

In the next example we first create a class called Person and create two sub-classes Student and Teacher. As both of the classes are inherited from Person class they will have all methods of Person and will have new methods and variables for their own purpose.

student\_teacher.py

class Person(object):

*"""*

*Returns a ```Person``` object with given name.*

*"""*

def \_\_init\_\_(self, name):

self.name = name

def get\_details(self):

"Returns a string containing name of the person"

return self.name

class Student(Person):

*"""*

*Returns a ```Student``` object, takes 3 arguments, name, branch, year.*

*"""*

def \_\_init\_\_(self, name, branch, year):

Person.\_\_init\_\_(self, name)

self.branch = branch

self.year = year

def get\_details(self):

"Returns a string containing student's details."

return "*%s* studies *%s* and is in *%s* year." % (self.name, self.branch, self.year)

class Teacher(Person):

*"""*

*Returns a ```Teacher``` object, takes a list of strings (list of papers) as*

*argument.*

*"""*

def \_\_init\_\_(self, name, papers):

Person.\_\_init\_\_(self, name)

self.papers = papers

def get\_details(self):

return "*%s* teaches *%s*" % (self.name, ','.join(self.papers))

person1 = Person('Sachin')

student1 = Student('Kushal', 'CSE', 2005)

teacher1 = Teacher('Prashad', ['C', 'C++'])

print person1.get\_details()

print student1.get\_details()

print teacher1.get\_details()

In this example you can see how we called the \_\_init\_\_ method of the class Person in both Student and Teacher classes’ \_\_init\_\_ method. We also reimplemented get\_details() method of Person class in both Student and Teacher class. So, when we are calling get\_details() method on the teacher1 object it returns based on the object itself (which is of teacher class) and when we call get\_details()on the student1 or person1 object it returns based on get\_details() method implemented in its own class.

**Multiple Inheritance**

One class can inherit more than one classes. It gets access to all methods and variables of the parent classes. The general syntax is:

class MyClass(Parentclass1, Parentclass2,...):

def \_\_init\_\_(self):

Parentclass1.\_\_init\_\_(self)

Parentclass2.\_\_init\_\_(self)

...

...

**Deleting an object**

As we already know how to create an object, now we are going to see how to delete an Python object. We use del for this.

>>> s = "I love you"

>>> del s

>>> s

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 's' is not defined

del actually decreases reference count by one. When the reference count of an object becomes zero the garbage collector will delete that object.

**Getters and Setters in Python**

One simple answer, don’t. If you are coming from other languages (read Java), you will be tempted to use getters or setters in all your classes. Please don’t. Just use the attributes directly. The following shows a direct example.

>>> class Student(object):

... def \_\_init\_\_(self, name):

... self.name = name

...

>>> std = Student("Kushal Das")

>>> print std.name

Kushal Das

>>> std.name = "Python"

>>> print std.name

Python

**Properties**

If you want more fine tuned control over data attribute access, then you can use properties. In the following example of a bank account, we will make sure that no one can set the money value to negative and also a property called inr will give us the INR values of the dollars in the account.

class Account(object):

*"""The Account class,*

*The amount is in dollars.*

*"""*

def \_\_init\_\_(self, rate):

self.\_\_amt = 0

self.rate = rate

@property

def amount(self):

"The amount of money in the account"

return self.\_\_amt

@property

def inr(self):

"Gives the money in INR value."

return self.\_\_amt \* self.rate

@amount.setter

def amount(self, value):

if value < 0:

print "Sorry, no negative amount in the account."

return

self.\_\_amt = value

if \_\_name\_\_ == '\_\_main\_\_':

acc = Account(61) *# Based on today's value of INR :(*

acc.amount = 20

print "Dollar amount:", acc.amount

print "In INR:", acc.inr

acc.amount = -100

print "Dollar amount:", acc.amount

**Modules**

In this chapter we are going to learn about Python modules.

**Introduction**

Up until now, all the code we wrote in the Python interpreter was lost when we exited the interpreter. But when people write large programs they tend to break their code into multiple different files for ease of use, debugging and readability. In Python we use modules to achieve such goals. Modules are nothing but files with Python definitions and statements. The module name, to import, has the same name of the Python file without the .py extension.

You can find the name of the module by accessing the \_\_name\_\_ variable. It is a global variable.

Now we are going to see how modules work. Create a file called bars.py. Content of the file is given bellow.

*"""*

*Bars Module*

*============*

*This is an example module with provide different ways to print bars.*

*"""*

def starbar(num):

*"""*

*Prints a bar with \**

*:arg num: Length of the bar*

*"""*

print '\*' \* num

def hashbar(num):

*"""*

*Prints a bar with #*

*:arg num: Length of the bar*

*"""*

print '#' \* num

def simplebar(num):

*"""*

*Prints a bar with -*

*:arg num: Length of the bar*

*"""*

print '-' \* num

Now we are going to start the Python interpreter and import our module.

>>> import bars

>>>

This will import the module bars. We have to use the module name to access functions inside the module.

**>>>** bars.hashbar(10)

##########

**>>>** bars.simplebar(10)

----------

**>>>** bars.starbar(10)

\*\*\*\*\*\*\*\*\*\*

**Importing modules**

There are different ways to import modules. We already saw one way to do this. You can even import selected functions from modules. To do so:

>>> from bars import simplebar, starbar

>>> simplebar(20)

**Default modules**

Now your Python installation comes with different modules installed, you can use them as required and install new modules for any other special purposes. In the following few examples we are going to see many examples on the same.

**>>>** help()

Welcome to Python 2.7! This is the online help utility.

If this is your first time using Python, you should definitely check out

the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing

Python programs and using Python modules. To quit this help utility and

return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",

"keywords", or "topics". Each module also comes with a one-line summary

of what it does; to list the modules whose summaries contain a given word

such as "spam", type "modules spam".

help> modules

The above example shows how to get the list of all installed modules in your system. I am not pasting them here as it is a big list in my system :)

You can also use help() function in the interpreter to find documentation about any module/classes. Say you want to know all available methods in strings, you can use the following method

**>>>** help(str)

**Module os**

os module provides operating system dependent functionality. You can import it using the following import statement.

>>> import os

getuid() function returns the current process’s effective user’s id.

**>>>** os.getuid()

500

getpid() returns the current process’s id. getppid() returns the parent process’s id.

**>>>** os.getpid()

16150

**>>>** os.getppid()

14847

uname() returns different information identifying the operating system, in Linux it returns details you can get from the uname command. The returned object is a tuple, (sysname, nodename, release, version, machine)

**>>>** os.uname()

('Linux', 'd80', '2.6.34.7-56.fc13.i686.PAE', '#1 SMP Wed Sep 15 03:27:15 UTC 2010', 'i686')

getcwd()returns the current working directory. chdir(path) changes the current working directory to path. In the example we first see the current directory which is my home directory and change the current directory to /tmp and then again checking the current directory.

**>>>** os.getcwd()

'/home/kushal'

**>>>** os.chdir('/tmp')

**>>>** os.getcwd()

'/tmp'

So let us use another function provided by the os module and create our own function to list all files and directories in any given directory.

def view\_dir(path='.'):

*"""*

*This function prints all files and directories in the given directory.*

*:args path: Path to the directory, default is current directory*

*"""*

names = os.listdir(path)

names.sort()

for name in names:

print name,

**Using the view\_dir example.**

**>>>** view\_dir('/')

.readahead bin boot dev etc home junk lib lib64 lost+found media mnt opt

proc root run sbin srv sys tmp usr var

**Collections module**

In this chapter we will learn about a module called Collections. This module implements some nice data structures which will help you to solve various real life problems.

>>> import collections

This is how you can import the module, now we will see the available classes which you can use.

**Counter**

Counter is a dict subclass which helps to count hashable objects. Inside it elements are stored as dictionary keys and counts are stored as values which can be zero or negative.

Below we will see one example where we will find occurrences of words in the Python LICENSE file.

Counter example

>>> from collections import Counter

>>> import re

>>> path = '/usr/share/doc/python-2.7.3/LICENSE'

>>> words = re.findall('\w+', open(path).read().lower())

>>> Counter(words).most\_common(10)

[('2', 97), ('the', 80), ('or', 78), ('1', 76), ('of', 61), ('to', 50), ('and', 47), ('python', 46), ('psf', 44), ('in', 38)]

Counter objects has a method called elements which returns an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order.

**>>>** c = Counter(a=4, b=2, c=0, d=-2)

**>>>** list(c.elements())

['a', 'a', 'a', 'a', 'b', 'b']

most\_common is a method which returns most common elements and their counts from the most common to the least.

**>>>** Counter('abracadabra').most\_common(3)

[('a', 5), ('r', 2), ('b', 2)]

**defaultdict**

defaultdict is a dictionary like object which provides all methods provided by dictionary but takes first argument (default\_factory) as default data type for the dictionary. Using defaultdict is faster than doing the same using dict.set\_default method.

defaultdict example

>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

>>> d = defaultdict(list)

>>> for k, v in s:

... d[k].append(v)

...

>>> d.items()

[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

In the example you can see even if the key is not there in the defaultdict object, it automatically creates an empty list. list.append then helps to append the value to the list.

**namedtuple**

Named tuples helps to have meaning of each position in a tuple and allow us to code with better readability and self-documenting code. You can use them in any place where you are using tuples. In the example we will create a namedtuple to show hold information for points.

Named tuple

**>>> from** **collections** **import** namedtuple

**>>>** Point = namedtuple('Point', ['x', 'y']) *# Defining the namedtuple*

**>>>** p = Point(10, y=20) *# Creating an object*

**>>>** p

Point(x=10, y=20)

**>>>** p.x + p.y

30

**>>>** p[0] + p[1] *# Accessing the values in normal way*

30

**>>>** x, y = p *# Unpacking the tuple*

**>>>** x

10

**>>>** y

20

**Iterators, generators and decorators**

In this chapter we will learn about iterators, generators and decorators.

**Iterators**

Python iterator objects required to support two methods while following the iterator protocol.

\_\_iter\_\_ returns the iterator object itself. This is used in for and in statements.

next method returns the next value from the iterator. If there is no more items to return then it should raise StopIteration exception.

class Counter(object):

def \_\_init\_\_(self, low, high):

self.current = low

self.high = high

def \_\_iter\_\_(self):

'Returns itself as an iterator object'

return self

def next(self):

'Returns the next value till current is lower than high'

if self.current > self.high:

raise StopIteration

else:

self.current += 1

return self.current - 1

Now we can use this iterator in our code.

>>> c = Counter(5,10)

>>> for i in c:

... print i,

...

5 6 7 8 9 10

Remember that an iterator object can be used only once. It means after it raises StopIteration once, it will keep raising the same exception.

>>> c = Counter(5,6)

>>> next(c)

5

>>> next(c)

6

>>> next(c)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 11, in next

StopIteration

>>> next(c)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 11, in next

**StopIteration**

Using the iterator in for loop example we saw, the following example tries to show the code behind the scenes.

>>> iterator = iter(c)

>>> while True:

... try:

... x = iterator.next()

... print x,

... except StopIteration as e:

... break

...

5 6 7 8 9 10

**Generators**

In this section we learn about Python generators. They were introduced in Python 2.3. It is an easier way to create iterators using a keyword yield from a function.

>>> def my\_generator():

... print "Inside my generator"

... yield 'a'

... yield 'b'

... yield 'c'

...

>>> my\_generator()

<generator object my\_generator at 0x7fbcfa0a6aa0>

In the above example we create a simple generator using the yield statements. We can use it in a for loop just like we use any other iterators.

>>> for char in my\_generator():

... print char

...

Inside my generator

a

b

c

In the next example we will create the same Counter class using a generator function and use it in a for loop.

>>> def counter\_generator(low, high):

... while low <= high:

... yield low

... low += 1

...

>>> for i in counter\_generator(5,10):

... print i,

...

5 6 7 8 9 10

Inside the while loop when it reaches to the yield statement, the value of low is returned and the generator state is suspended. During the second next call the generator resumed where it froze before and then the value of low is increased by one. It continues with the while loop and comes to the yield statement again.

When you call an generator function it returns a \*generator\* object. If you call \*dir\* on this object you will find that it contains \_\_iter\_\_ and \*next\* methods among the other methods.

**>>>** c = counter\_generator(5,10)

**>>>** dir(c)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_',

'\_\_name\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_',

'\_\_subclasshook\_\_', 'close', 'gi\_code', 'gi\_frame', 'gi\_running', 'next', 'send', 'throw']

We mostly use generators for laze evaluations. This way generators become a good approach to work with lots of data. If you don’t want to load all the data in the memory, you can use a generator which will pass you each piece of data at a time.

One of the biggest example of such example is os.path.walk() function which uses a callback function and current os.walk generator. Using the generator implementation saves memory.

We can have generators which produces infinite values. The following is a one such example.

>>> def infinite\_generator(start=0):

... while True:

... yield start

... start += 1

...

>>> for num in infinite\_generator(4):

... print num,

... if num > 20:

... break

...

4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

If we go back to the example of my\_generator we will find one feature of generators. They are not re-usable.

>>> g = my\_generator()

>>> for c in g:

... print c

...

Inside my generator

a

b

c

>>> for c in g:

... print c

...

One way to create a reusable generator is Object based generators which does not hold any state. Any class with a \_\_iter\_\_ method which yields data can be used as a object generator. In the following example we will recreate out counter generator.

**>>> class** **Counter**(object):

**...**  **def** \_\_init\_\_(self, low, high):

**...**  self.low = low

**...**  self.high = high

**...**  **def** \_\_iter\_\_(self):

**...**  counter = self.low

**...**  **while** self.high >= counter:

**...**  **yield** counter

**...**  counter += 1

**...**

**>>>** gobj = Counter(5, 10)

**>>> for** num **in** gobj:

**...**  **print** num,

**...**

5 6 7 8 9 10

**>>> for** num **in** gobj:

**...**  **print** num,

**...**

5 6 7 8 9 10

**Generator expressions**

In this section we will learn about generator expressions which is a high performance, memory efficient generalization of list comprehensions and generators.

For example we will try to sum the squares of all numbers from 1 to 99.

>>> sum([x\*x for x in range(1,10)])

The example actually first creates a list of the square values in memory and then it iterates over it and finally after sum it frees the memory. You can understand the memory usage in case of a big list.

We can save memory usage by using a generator expression.

sum(x\*x for x in range(1,10))

The syntax of generator expression says that always needs to be directly inside a set of parentheses and cannot have a comma on either side. Which basically means both the examples below are valid generator expression usage example.

>>> sum(x\*x for x in range(1,10))

285

>>> g = (x\*x for x in range(1,10))

>>> g

<generator object <genexpr> at 0x7fc559516b90>

We can have chaining of generators or generator expressions. In the following example we will read the file \*/var/log/cron\* and will find if any particular job (in the example we are searching for anacron) is running successfully or not.

We can do the same using a shell command tail -f /var/log/cron |grep anacron

>>> jobtext = 'anacron'

>>> all = (line for line in open('/var/log/cron', 'r') )

>>> job = ( line for line in all if line.find(jobtext) != -1)

>>> text = next(job)

>>> text

"May 6 12:17:15 dhcp193-104 anacron[23052]: Job `cron.daily' terminated\n"

>>> text = next(job)

>>> text

'May 6 12:17:15 dhcp193-104 anacron[23052]: Normal exit (1 job run)\n'

>>> text = next(job)

>>> text

'May 6 13:01:01 dhcp193-104 run-parts(/etc/cron.hourly)[25907]: starting 0anacron\n'

You can write a for loop to the lines.

**Closures**

Closures are nothing but functions that are returned by another function. We use closures to remove code duplication. In the following example we create a simple closure for adding numbers.

>>> def add\_number(num):

... def adder(number):

... 'adder is a closure'

... return num + number

... return adder

...

>>> a\_10 = add\_number(10)

>>> a\_10(21)

31

>>> a\_10(34)

44

**>>>** a\_5 = add\_number(5)

**>>>** a\_5(3)

8

adder is a closure which adds a given number to a pre-defined one.

**Decorators**

Decorator is way to dynamically add some new behavior to some objects. We achieve the same in Python by using closures.

In the example we will create a simple example which will print some statement before and after the execution of a function.

>>> def my\_decorator(func):

... def wrapper(\*args, \*\*kwargs):

... print "Before call"

... result = func(\*args, \*\*kwargs)

... print "After call"

... return result

... return wrapper

...

>>> @my\_decorator

... def add(a, b):

... "Our add function"

... return a + b

...

>>> add(1, 3)