Advance Python Tutorials

Meher Krishna Patel

Created on : Octorber, 2017

Last updated : October, 2018

Table of contents

Chapter 1

Python review

1.1 Introduction

Python is the programming language which can be used for various purposes e.g. web design, mathematical modeling, creating documents and game designs etc. In this chapter, we will review some of the basic features of Python. Then, from next chapter we will write some good coding styles with advance features of Python.

Note: This chapter presents a short review of Python along with some good coding practices. Actual tutorial begins from next chapter. If you have basic knowledge of Python and OOPs then you can skip this chapter.

1.2 Download and Installation

We will use Python 3 in this tutorial. Also, these codes may not run in Python 2.x (e.g. 2.7 or 2.9 etc.) versions.

Further, there are several useful libraries available for Python. For example, Numpy and Scipy libraries contain various mathematical functions which are very useful in simulations. Matplotlib library is required to plot results and save these in various formats e.g. PDF, JPEG or PS etc. Lastly, SPYDER environment is very helpful for storing the results obtained by the simulations. Also, SPYDER can save data as '.mat' file, which can be used by MATLAB software as well. All these topics are briefly discussed in this tutorial.

For installation, simplest option is to download and install Anaconda software, as it contains various useful Python libraries (including the libraries which are mentioned in above paragraph)

1.3 Basics

In this section, 'print' command is used to show the code execution in Python. In Python, code can be run in two ways, i.e. through 'Python shell' or 'by executing the Python files', which are discussed next.

1.3.1 Run code from Python shell (>>>)

Go to terminal/command-prompt and type Python as shown in Listing 1.1. This command will start the Python shell with three 'greater than' signs i.e. >>>. Now, we can write our first command i.e. **print('Hello World')**, which prints the 'Hello World' on the screen as shown in the listing. Further in line 4, two placeholder '%s' are used, which are replaced by the two words i.e. World and Python as shown in line 5. Also, After executing the command, >>> appear again, which indicates that Python shell is ready to get more commands.

Note: Choose correct command to open Python3

```
$ python   (Linux)

or

$ python3    (Linux)

or

C:\>python   (in windows)
```

Listing 1.1: Hello World

```
>>>
>>> print("Hello World")
Hello World
>>> print("Hello %s, simulation can be done using %s." % ("World", "Python"))
Hello World, simulation can be done using Python.
>>>
```

1.3.2 Running Python Files

We can also save the code in Python file with extension '.py'; which can be run from Python shell. For example, let a file 'hello.py' is saved in the folder name as 'PythonCodes' at C: drive. Content of the 'hello.py' is shown in Listing 1.2. '#' in the file is used for comments (which increases the readability of the code), and has no effect in the execution of the code. To run the 'hello.py', we need to locate the folder 'PythonCodes' (where we saved the hello.py file) and then execute the file as shown in Listing 1.3.

Note that Python codes work on indentations i.e. each block of code is defined by the line indentation (or spaces). Any wrong space will result in the error which can be seen by uncommenting the line 6 Listing 1.2,

Listing 1.2: hello.py

```
# hello.py: prints "Hello World"
# save this file to any location e.g. C:\>PythonCodes
print("Hello World")

## following line will be error because of extra space in the beginning
#   print("Hello World with spaces is error") # error
```

Listing 1.3: Run hello.py

```
$ python hello.py
Hello World
```

Note: Since this method stores the code which can be used later, therefore this method is used throughout the tutorial. Please read the comments in the Listings to understand the codes completely.

1.3.3 Variables

Variables are used to store some data as shown in Listing 1.4. Here, two variables a and b are defined in line 3 and 4 respective, which store the values 3 and 6 respectively. Then various operations are performed on those variables.

Listing 1.4: Variables

```
#variableEx.py: a and b are the variables
# which stores 3 and 5 respectively
a=3
b=6

# following line will add value of a i.e. 3 with 6
print(a+6) # 9

# following line will perform (3+5)/2
print((a+b)/2) # 4.5

# following line will perform (3+5)/2
# and then display the interger value only
print(int((a+b)/2)) # 4
```

1.3.4 Built-in object types

Table 1.1 shows the various built-in object types available in Python. Various operations can be performed on these object depending on their types e.g. add operations can be performed on number-object-type, or collection of data (e.g. username-password-email) can be created using list-object-type etc. All these object types are discussed in this section.

Table 1.1: Built-in object types

Object type	Exmaple
Number	“3.14, 5, 3+j2”
String	” ‘Python’, ‘make your code’”
List	“[1, ‘username’, ‘password’, ‘email’]”
Tuple	“(1, ‘username’, ‘password’, ‘email’)”
Dictionary	“{‘subject’:Python, ‘subjectCode’:1234}”
File	“text=opern(‘Python’, ‘r’).read()”

1.3.5 Numbers

Python supports various number types i.e. integer, float (decimal numbers), octal, hexadecimal and complex numbers as shown in Listing 1.5. The list also shows the method by which one format can be converted to another format.

Listing 1.5: Number formats

```
#numFormat.py
a = 11 #integer
print(hex(a)) #0xb

b = 3.2 # float(decimal)
## print(oct(b)) # error: can't convert float to oct or hex.
# integer can be converted, therefore first convert float to int
# and then to hex/oct
print(oct(int(b))) #0o3

d = 0X1A # hexadecimal: '0X' is used before number i.e. 1A
print(d) # 26

#add hex and float
print(b+d) #29.2
```

(continues on next page)

(continued from previous page)

```
16
17 c = 0o17 # octal: '0o' is used before number i.e. 17
18 # print command shows the integer value
19 print(c) # 15
20 #to see octal form use `oct'
21 print(oct(c)) #0o17
22
23 e = 3+2j # imagenary
24 print(e) #(3+2j)
25 print(abs(e)) #3.6055512754639896
26 # round above value upto 2 decimal
27 r=round(abs(e),2)
28 print(r) #3.61
```

1.3.6 String

String can be very useful for displaying some output messages on the screen as shown Listing 1.6. Here messages are displayed on screen (using 'input' command) to get inputs from user and finally output is shown using %s placeholder (see line 26 for better understand of %s).

Listing 1.6: Strings

```
1  #strEx.py
2  firstName = "Meher" #firstName is variable of string type
3  print(firstName) # Meher
4  fullName = "Meher Krishna Patel"
5  print(fullName) # Meher Krishna Patel
6
7  #input is used to take input from user
8  score1 = input("Enter the score 1: ")  #enter some value e.g. 12
9  score2 = input("Enter the score 2: ")  #enter some value e.g. 36
10 totalString = score1 + score2 # add score1 and score2
11 messageString = "Total score is %s"
12 #in below print, totalstring will be assinge to %s of messageString
13 print(messageString % totalString)  # 1236 (undesired result)
14
15 #score1 and score2 are saved as string above
16 #first we need to convert these into integers as below
17 totalInt = int(score1) + int(score2)# add score1 and score2
18 messageString = "Total score is %s"
19 print(messageString % totalInt)  # 48
20
21 #change the input as integer immediately
22 score1 = int(input("Enter the score 1: "))  #enter some value e.g. 12
23 score2 = int(input("Enter the score 2: ")) #enter some value e.g. 36
24 total = score1 + score2 # add score1 and score2
25 messageString = "score1(%s) + score2[%s] =  %s"
26 print(messageString % (score1, score2, total)) #score1(12) + score2[36] =  48
```

1.3.7 List

Variables can store only one data, whereas list can be used to store a collection of data of different types. A list contains items separated by commas, and enclosed within the square brackets [ ]. Listing 1.7 defines a list along with access to it's elements.

Listing 1.7: List

```
#listEx.py
a = [24, "as it is", "abc", 2+2j]
# index start with 0
# i.e. location of number 24 is '0' in the list
print(a[0]) # 24
print(a[2]) # abc

#replace 'abc' with 'xyz'
a[2]='xyz'
print(a[2]) # xyz

# Add 20 at the end of list
a.append(20)
print(a) # [24, 'as it is', 'abc', (2+2j), 20]
```

1.3.8 Tuple

A tuple is similar to the list. A tuple consists of values separated by commas as shown in Listing 1.8. Tuple can be considered as 'read-only' list because it's values and size can not be changed after defining it.

Listing 1.8: Tuple

```
#tupleEx.py
a = 24, "as it is", "abc", 2+2j

## some times () brackets are used to define tuple as shown below
#a = (24, "as it is", "abc", 2+2j)

# index start with 0
# i.e. location of number 24 is '0' in the list
print(a[0]) # 24
print(a[2]) # abc

##Following lines will give error,

##as value can be changed in tuple
#a[2]='xyz' # error

##as size of the tuple can not be changed
# a.append(20) # error
```

1.3.9 Dictionary

Dictionary can be seen as unordered list with key-value pairs. In the other works, since list's elements are ordered therefore it's elements can be access through index as shown in Listing 1.7. On the other hand, location of the elements of the dictionary get changed after defining it, therefore key-value pairs are required, and the values can be accessed using keys as shown in Listing 1.9.

Listing 1.9: Dictionary

```
#dictEx.py
myDict = {}          # define new dictionary
myDict[1] = "one"   # 1 is called key; "one" is called value
myDict['a'] = "alphabet"

print(myDict)             # {1: 'one', 'a': 'alphabet'}
```

(continues on next page)

(continued from previous page)

```
print(myDict.items())  # dict_items([(1, 'one'), ('a', 'alphabet')])
print(myDict.keys())   # dict_keys([1, 'a'])
print(myDict.values()) # dict_values(['one', 'alphabet'])

print(myDict[1]) #one
print(myDict['a']) # alphabet

# add key-value while creating the dictionary
subjectDict = {'py': 'Python', 'np': 'Numpy', 'sp':'Scipy'}
print(subjectDict) # {'py': 'Python', 'sp': 'Scipy', 'np': 'Numpy'}
```

1.4 Number conversion

Following are the patterns to convert the numbers in different formats.

1.4.1 Direct conversion

```
>>> # decimal to binary conversion with 6 places
>>> print('{:06b}'.format(10))
001010

>>> # if '6b' is used instead of '06b', then initial 0 will not be displayed.
>>> print('{:6b}'.format(10))
  1010

>>> # decimal to hex conversion with 6 places
>>> print('{:06x}'.format(10))
00000a

>>> # binary to hexadecimal with 3 places
>>> print('{:03x}'.format(0b1111))
00f
```

1.4.2 zfill

```
>>> # {:b} = binary
>>> print("{:b}".format(10).zfill(15))  # number = 10, total places = 15
000000000001010

>>> x = 10

>>> print("{:b}".format(x).zfill(15))  # number = x, total places = 15
000000000001010

>>> # {:x} = hexadecimal
>>> print("{:x}".format(x).zfill(15))  # number = x, total places = 15
00000000000000a

>>> # {:o} = octal
>>> print("{:o}".format(x).zfill(15))  # number = x, total places = 15
000000000000012


>>> # {:d} = decimal, 0x11 = 17
>>> print("{:d}".format(x).zfill(0x11))  # number = x, total places = 17
```

(continues on next page)

(continued from previous page)

```
00000000000000010

>>> # {:d} = decimal, 0b11 = 3
>>> print("{:d}".format(x).zfill(0b11))  # number = x, total places = 3
010

>>> # {:d} = decimal, 0o11 = 9
>>> print("{:d}".format(x).zfill(0o11))  # number = x, total places = 9
000000010
```

1.5 Control structure

In this section, various simple Python codes are shown to explain the control structures available in Python.

1.5.1 if-else

'If-else' statements are used to define different actions for different conditions. Symbols for such conditions are given in Table 1.2, which can be used as shown in Listing 1.11. Three examples are shown in this section for three types of statements i.e. *if*, *if-else* and *if-elif-else*.

Table 1.2: Symbols for conditions

Condition	Symbol
equal	==
not equal	!=
greater	>
smaller	<
greater or equal	>=
smaller or equal	<=

1.5.1.1 If statement

In this section, only if statement is used to check the even and odd number.

Listing 1.10 checks whether the number stored in variable 'x' is even or not.

Listing 1.10: If statement

```
1  #ifEx.py
2  x = 2
3  # brackets are not necessary (used for clarity of the code)
4  if (x%2 == 0): # % sign gives the value of the remainder e.g. 5%3 = 2
5      #if above condition is true, only then following line will execute
6      print('Number is even.')
7
8  #this line will execute always as it is not inside 'if'
9  print('Bye Bye')
10
11 '''
12 Number is even.
13 Bye Bye
14 '''
15
16 '''
17 if you put x = 3, then output will be,
```

(continues on next page)

(continued from previous page)

```
18 Bye Bye
19 i.e. Number is even will not be printed as 'if' condition is not satisfied.
20 '''
```

Explanation Listing 1.10

An if statement is made up of the 'if' keyword, followed by the condition and colon (:) at the end, as shown in line 4. Also, the line 6 is indented which means it will execute only if the condition in line 4 is satisfied (see Listing 1.13 for better understanding of indentation). Last print statement has no indentation, therefore it is not the part of the 'if' statement and will execute all the time. You can check it by changing the value of 'x' to 3 in line 2. Three quotes (in line 11 and 14) are used to comment more than one lines in the code, e.g. '" results here '" is used at the end of the code (see line 11-20) which contain the results of the code.

Warning: Note that, the comments inside the '" is known as 'Docstrings', which are displayed when help command is used. Therefore, do not use it for multiline comments. It is better to use # at each line.

1.5.1.2 Multiple If statements

Listing 1.11 checks the **even and odd** numbers using 'if' statements. Since there are two conditions (even and odd), therefore two 'if' statements are used to check the conditions as shown in Listing 1.11. Finally output is shown as the comments at the end of the code.

Listing 1.11: Multiple If statements

```
1  #ifOnly.py: uses multiple if to check even and odd nubmer
2
3  # "input" command takes input as string
4  # int is used for type conversion
5  x=int(input('Enter the number:\t')) #\t is used for tab in the output
6
7  # % is used to calculate remainder
8  if x%2==0:
9      print ("Number is even")
10 if x%2!=0:
11     print ("Number is odd")
12
13 '''
14 Output-1st run:
15 Enter the number:   10
16 Number is even
17
18 Output-2nd run:
19 Enter the number:   15
20 Number is odd
21 '''
```

Explanation Listing 1.11

This code demonstrate that one can use multiple 'if' conditions in codes. In this code, value of 'x' is taken from using line 5 in the code. 'int' is used in this line because 'input' command takes the value as string and it should be changed to integer value for mathematical operations on it.

1.5.1.3 If-else

As we know that a number can not be even and odd at the same time. In such cases we can use 'if-else' statement.

Code in Listing 1.11 can be written using If-else statement as show in Listing 1.12.

Listing 1.12: If-else statement

```
1  # ifelse1.py: use if-else to check even and odd nubmer
2
3  # "input" command takes input as string
4  x= int(input('Enter the number:\t'))
5
6  # % is used to calculate remainder
7  if x%2==0:
8      print("Number is even")
9  else:
10     print("Number is odd")
11
12 '''
13 Output-1st run:
14 Enter the number:   10
15 Number is even
16
17 Output-2nd run:
18 Enter the number:   15
19 Number is odd
20 '''
```

Explanation Listing 1.12

Line 4 takes the input from the user. After that remainder is checked in line 7. If condition is true i.e. remainder is zero, then line 8 will be executed; otherwise print command inside 'else' statement (line 10) will be executed.

1.5.1.4 If-elif-else

In previous case, there were two contions which are implemented using if-else statement. If there are more than two conditions then 'elif' block can be used as shown in next example. Further, 'If-elif-else' block can contain any number of 'elif' blocks between one 'if' and one 'else' block.

Listing 1.13 checks whether the number is divisible by 2 and 3, using nested 'if-else' statement.

Listing 1.13: If-elif-else statement

```
1  #elif.py: checks divisibility with 2 and 3
2
3  # "int(input())" command takes input as number
4  x=int(input('Enter the number:\t'))
5
6  # % is used to calculate remainder
7  if x%2==0: #check divisibility with 2
8      if x%3==0: # if x%2=0, then check this line
9          print("Number is divisible by 2 and 3")
10     else:
11         print("Number is divisible by 2 only")
12         print("x%3= ", x%3)
13 elif x%3==0: #check this if x%2 is not zero
14     print("Number is divisible by 3 only")
15 else:
16     print("Number is not divisible by 2 and 3")
17     print("x%2= ", x%2)
18     print("x%3= ", x%3)
19 print("Thank you")
20
21 '''
```

(continues on next page)

(continued from previous page)

```
output 1:
Enter the number:   12
Number is divisible by 2 and 3
Thank you

output 2:
Enter the number:   8
Number is divisible by 2 only
x%3=  2
Thank you

output 3:
Enter the number:   7
Number is not divisible by 2 and 3
x%2=  1
x%3=  1
Thank you

output 4:
Enter the number:   15
Number is divisible by 3 only
Thank you
'''
```

Explanation Listing 1.13

Let's discuss the indentation first. First look at the indentation at lines '7' and '8'. Since line '8' is shifted by one indentation after line '7', therefore it belongs to line '7', which represents that Python-interpreter will go to line '8' only if line '7' is true. Similarly, print statements at line '11' and '12' are indented with respect to line '10', therefore both the print statement will be executed when Python-interpreter reaches to 'else' condition.

Now we will see the output for 'x=12'. For 'x=12', 'if' statement is satisfied at line '7', therefore Python-interpreter will go to line '8', where the divisibility of the number is checked with number '3'. The number is divisible by '3' also, hence corresponding print statement is executed as shown in line '24'. After this, Python-interpreter will exit from the 'if-else' statements and reached to line '19' and output at line '25' will be printed.

Lets consider the input 'x=7'. In this case number is not divisible by '2' and '3'. Therefore Python-interpreter will reached to line '15'. Since lines '16', '17' and '18' are indented with respect to line '15', hence all the three line will be printed as shown in line '35-37' . Finally, Python-interpreter will reach to line '19' and print this line also.

Lastly, use 15 as the input number. Since it is not divided by 2, it will go to elif statement and corresponding print statement will be executed.

Since there are three conditions in this example. therefore 'elif' statement is used. Remember that 'if-else' can contain only one 'if' and one 'else' statement , but there is no such restriction of 'elif' statement. Hence, if there higher number of conditions, then we can increase the number of 'elif' statement.

Listing 1.13 can be written as Listing 1.14 and Listing 1.15 as well. Here 'or' and 'and' keywords are used to verify the conditions. The 'and' keyword considers the statement as true, if and only if, all the conditions are true in the statement; whereas 'or' keyword considers the statement as true, if any of the conditions are true in the statement.

Listing 1.14: 'and' logic

```
#andLogic.py: check divisibility with 2 and 3
x=int(input('Enter the number:\t'))

if x%2==0 and x%3==0: #check divisibility with both 2 and 3
    print("Number is divisible by 2 and 3")
```

(continues on next page)

(continued from previous page)

```
elif x%2==0: #check this if x%2 is not zero
    print("Number is divisible by 2 only")
elif x%3==0: #check this if x%3 is not zero
    print("Number is divisible by 3 only")
else:
    print("Number is not divisible by 2 and 3")
    print("x%2= ", x%2)
    print("x%3= ", x%3)
print("Thank you")
```

Listing 1.15: 'and' and 'or' logic

```
#orLogic.py: check divisibility with 2 and 3
x=int(input('Enter the number:\t'))

# % is used to calculate remainder
if x%2==0 or x%3==0: #check divisibility with 2 or 3
    if x%2==0 and x%3==0: #check if divided by both 2 and 3
        print("Number is divisible by 2 and 3")
    elif x%2==0: #check this if x%2 is not zero
        print("Number is divisible by 2 only")
    elif x%3==0: #check this if x%3 is not zero
        print("Number is divisible by 3 only")
else:
    print("Number is not divisible by 2 and 3")
    print("x%2= ", x%2)
    print("x%3= ", x%3)
print("Thank you")
```

1.5.2 While loop

'While' loop is used for recursive action, and the loop repeat itself until a certain condition is satisfied.

Listing 1.16 uses 'while' loop to print numbers 1 to 5. For printing numbers upto 5, value of initial number should be increased by 1 at each iteration, as shown in line 7.

Listing 1.16: While loop

```
#WhileExample1.py: Print numbers upto 5

n=1 #initial value of number
print("Numbers upto 5: ")
while n<6:
    print(n, end=" "), #end=" ": to stop row change after print
    n=n+1
print("\nCode ended at n =  %s" % n)

'''
output:
Numbers upto 5:
1 2 3 4 5
Code ended at n =  6
'''
```

Explanation Listing 1.16

In the code, line '3' sets the initial value of the number i.e. 'n=1'. Line '5' indicates that 'while' loop will be executed until 'n' is less than 6. Next two lines i.e. line '6' and '7', are indented with respect to line '5'. Hence these line will be executed if and only if the condition at line '5' is satisfied.

Since the value of 'n' is one therefore while loop be executed. First, number 1 is printed by line '6',

then value of 'n' is incremented by 1 i.e. 'n=2'. 'n' is still less than 6, therefore loop will be executed again. In this iteration value of 'n' will become 3, which is still less than 6. In the same manner, loop will continue to increase the value of 'n' until 'n=6'. When 'n=6', then loop condition at line '5' will not be satisfied and loop will not be executed this time. At this point Python-interpreter will reach to line '8', where it will print the value stored in variable 'n' i.e. 6 as shown in line '14'.

Also, look at 'print' commands at lines '3, 6' and '8'. At line '6', "end = ' '" is placed at the end, which results in no line change while printing outputs as shown at line '13'. At line '8', '\n' is used to change the line, otherwise this print output will be continued with output of line '6'.

1.5.3 For loop

Repetitive structure can also be implemented using 'for' loop. **For** loop requires the keyword **in** and some **sequence** for execution. Lets discuss the **range** command to generate sequences, then we will look at 'for' loop. Some outputs of 'range' commands are shown in Listing 1.17.

Listing 1.17: Range command

```
1  >>> range(5)
2  [0, 1, 2, 3, 4]
3
4  >>> range(1,4)
5  [1, 2, 3]
6
7  >>> range(11, 19, 2)
8  [11, 13, 15, 17]
9
10 >>> range(15, 7, -2)
11 [15, 13, 11, 9]
```

Explanation Listing 1.17

From the outputs of 'range' commands in the listing, it is clear that it generates sequences of integers. Python indexing starts from zero, therefore command 'range(5)' at line '3' generates five numbers ranging from '0' to '4'.

At line '6', two arguments are given in 'range' commands i.e. '1' and '4'. Note that output for this at line '7' starts from '1' and ends at '3' i.e. last number is not included by Python in the output list.

At line '9' and '12', three arguments are provided to 'range' function. In these cases, third argument is the increment value e.g. line '12' indicates that the number should start from '15' and stop at number '7' with a decrement of '2' at each step. Note that last value i.e. '7' is not included in the output again. Similarly, output of line '9' does not include '19'.

Listing 1.18 prints numbers from '1' to '5' in forward and reverse direction using range command.

Listing 1.18: For loop

```
1  #ForExample1.py: Print numbers 1-5
2
3  print("Numbers in forward order")
4  for i in range(5):
5      print(i+1, end=" ")
6  print("\nFinal value of i is: ", i)
7
8  print ("\nNumbers in reverse order")
9  for j  in range(5, 0, -1):
10     print(j, end=" "),
11 print("\nFinal value of i is: ", j)
12
13 '''
14 outputs:
```

(continues on next page)

(continued from previous page)

```
Numbers in forward order
1 2 3 4 5
Final value of i is:  4

Numbers in reverse order
5 4 3 2 1
Final value of i is:  1
'''

fruits=["apple", "banana", "orange"]
print("List of fruits are shown below:")
for i in fruits:
    print(i)
'''
List of fruits are shown below:
apple
banana
orange
'''
```

Explanation Listing 1.18

At line '4', command 'range(5)' generates the five numbers, therefore loop repeats itself five times. Since, output of range starts from '0', therefore 'i' is incremented by one before printing. Line '6' shows that the variable 'i' stores only one value at a time, and the last stored value is '4' i.e. last value of 'range(5)'.

At line '9', variable 'j' is used instead of 'i' and range command generates the number from 1 to 5 again but in reverse order. Note that number of iteration in for loop depends on the number of elements i.e. length of the 'range' command's output and independent of the element values. Line '10' prints the current value of 'j' at each iteration. Finally, line '15' prints the last value stores in variable 'j' i.e. 'j=1', which is the last value generated by command 'range(5,0,-1)'.

Code in line '24' shows that, how the values are assigned to the iterating variable 'i' from a list. The list 'fruits' contains three items, therefore loop will execute three times; and different elements of the list are assigned to variable 'i' at each iteration i.e. apple is assign first, then banana and lastly orange will be assigned.

1.6 Function

Some logics may be used frequently in the code, which can be written in the form of functions. Then, the functions can be called whenever required, instead of rewriting the logic again and again.

In Listing 1.19, the function 'addTwoNum' adds two numbers.

Listing 1.19: Function

```
#funcEx.py
def addTwoNum(a, b):
    sum = a+b
    return(sum)

result = addTwoNum(3,4)
print("sum of numbers is %s" % result)

'''
sum of numbers is 7
'''
```

Explanation Listing 1.19

In Python, function is defined using keyword 'def'. Line 2 defines the function with name 'addTwoNum' which takes two parameter i.e. a and b. Line 3 add the values of 'a' and 'b' and stores the result in variable 'sum'. Finally line 4 returns the value to function call which is done at line 6.

In line 6, function 'addTwoNum' is called with two values '4' and '5' which are assigned to variable 'a' an 'b' respectively in line 2. Also, function returns the 'sum' variable from line 4, which is stored in variable 'results' in line 6 (as line 6 called the function). Finally, line 7 prints the result.

In Listing 1.20, the function is defined with some default values; which means if user does not provide all the arguments' values, then default value will be used for the execution of the function.

Listing 1.20: Function with default arguments

```
1  #funcEx2.py: default argument can be defined only after non-default argument
2  # e.g. addTwoNum(num1=2, num2): is wrong. b must have some defined value
3  def addTwoNum(num1, num2=2):
4      return(num1+num2)
5
6  result1 = addTwoNum(3)
7  print("result1=%s" % result1)
8
9  result2 = addTwoNum(3,4)
10 print("result2=%s" % result2)
11
12 '''
13 result1=5
14 result2=7
15 '''
```

Explanation Listing 1.20

Function of this listing is same as Listing 1.19. Only difference is that the line 3 contains a default value for num2 i.e. 'num2 = 2'. Default value indicates that, if function is called without giving the second value then it will be set to 2 by default, as shown in line 6. Line 6 pass only one value i.e. 3, therefore num1 will be assign 3, whereas num2 will be assigned default value i.e. 2. Rest of the the working is same as Listing 1.19.

Note: There are various other important Python features e.g. classes, decorators and descriptors etc. which are not explained here as we are not going to use these in the coding. Further, using these features we can make code more efficient and reusable along with less error-prone.

1.7 Numpy, Scipy and Matplotlib

In this section, we will use various Python libraries, i.e. **Numpy**, **Scipy** and **Matplotlib**, which are very useful for scientific computation. With Numpy library, we can define array and matrices easily. Also, it contains various useful mathematical function e.g. random number generator i.e. 'rand' and 'randn' etc. Matplotlib is used for plotting the data in various format. Some of the function of these libraries are shown in this section. Further, Scipy library can be used for more advance features e.g. complementary error function (erfc) and LU factorization etc.

Listing 1.21 generates the sine wave using Numpy library; whereas the Matplotlib library is used for plotting the data.

Listing 1.21: Sine wave using Numpy and Matplotlib, Fig. 1.1

```
1  # numpyMatplot.py
2  import numpy as np
3  import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
4  # np.linspace: devide line from 0 to 4*pi into 100 equidistant points
5  x = np.linspace(0, 4*np.pi, 100)
6  sinx = np.sin(x) # find sin(x) for above 100 points
7  plt.plot(x,sinx) # plot (x, sin(x))
8  plt.xlabel("Time") # label for x axis
9  plt.ylabel("Amplitude") # label for y axis
10 plt.title('Sine wave') # title
11 plt.xlim([0, 4*np.pi]) # x-axis display range
12 plt.ylim([-1.5, 1.5]) # y-axis display range
13 plt.show() # to show the plot on the screen
```

Explanation Listing 1.21

First line import numpy library to the code. Also, it is imported with shortname 'np'; which is used in line 5 as 'np.linspace'. If line 2 is written as 'import numpy', then line 5 should be written as 'numpy.linspace'. Further, third line import 'pyplot' function of 'matplotlib' library as plt. Rest of the lines are explained as comments in the listing.

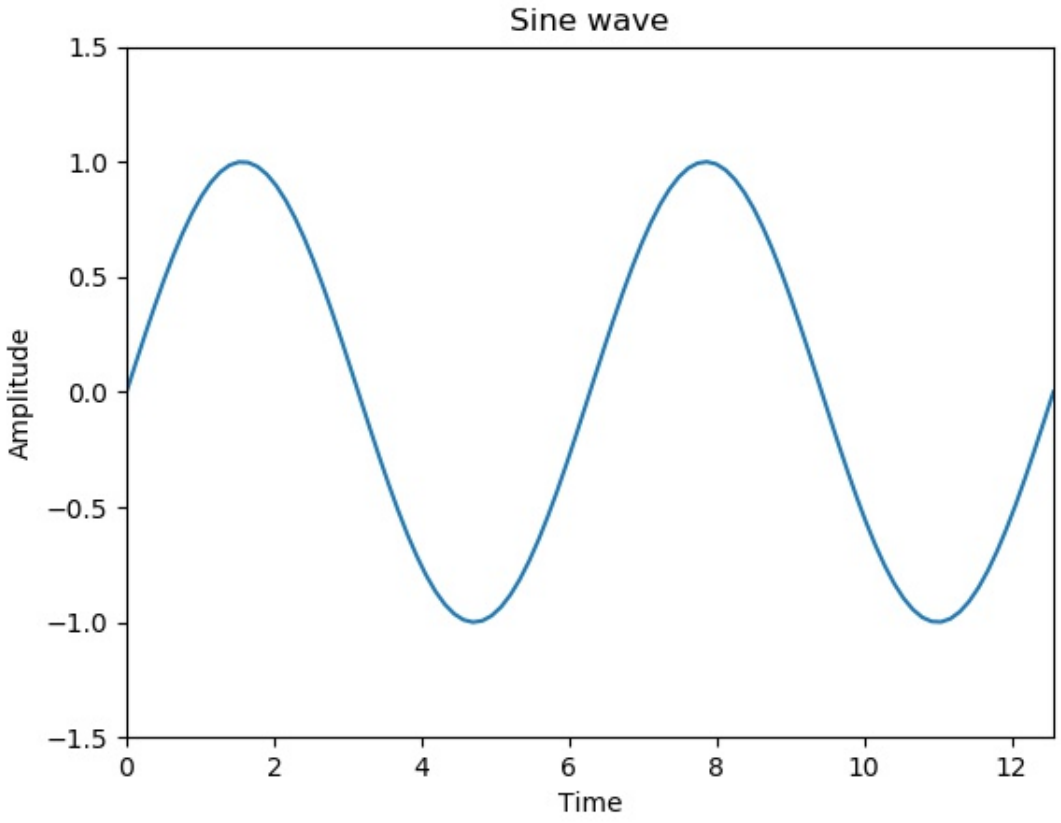

Fig. 1.1: Sine wave using Numpy and Matplotlib, Listing 1.21

1.7.1 Arrays

Arrays can be created using 'arange' and 'array' commands as shown below,

1.7.1.1 arange

One dimensional array can be created using 'arange' option as shown below,

Listing 1.22: arange

```
# arangeEx.py
import numpy as np
a=np.arange(1,10) # last element i.e. 10 is not included in result
print(a) # [1 2 3 4 5 6 7 8 9]
print(a.shape) # (9,) i.e. total 9 entries

b=np.arange(1,10,2) # print 1 to 10 with the spacing of 2
print(b) # [1 3 5 7 9]
print(b.shape) # (5,) i.e. total 9 entries

c=np.arange(10, 2, -2) # last element 2 is not included in result self
print(c) # [10  8  6  4]
```

1.7.1.2 array

Multidimensional array can not be created by 'arange'. Also, 'arange' can only generate sequences and can not take user-defined data. These two problems can be solved by using 'array' option as shown in Listing 1.23,

Listing 1.23: array

```
# arrayEx.py
import numpy as np

a= np.array([1, 8, 2])
print(a) # [1 8 2]
print(np.shape(a)) # (3,)

b=np.array([
    [1, 2],
    [4, 3],
    [6, 2]
])
# b can be written as follow as well, but above is more readable
# b=np.array([[1, 2],[4, 3]])
print(np.shape(b)) #(3, 2) i.e. 3 row and 2 column

# row of array can have different number of elements
c=np.array([[np.arange(1,10)],[np.arange(11, 16)]])
print(c)
'''
[[array([1, 2, 3, 4, 5, 6, 7, 8, 9])]
     [array([11, 12, 13, 14, 15])]]
'''
```

1.7.1.3 Matrix

Similar to array, we can define matrix using 'mat' function of numpy library as shown in Listing 1.24. Also, LU factorization of the matrix is shown in Listing 1.25 using Scipy library. There are differences in results of mathematical operations on the matrices defined by 'array' and 'mat', as shown in the Listing 1.25; e.g. 'dot' function is required for matrix multiplication of array; whereas '*' sign performs matrix multiplication for 'mat' function.

Listing 1.24: Matrix

```
# matrixEx.py
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from scipy.linalg import lu

a= np.mat('1, 2; 3, 2; 2, 3') # define matrix
# print(np.shape(a)) # (3, 2)

aT = np.transpose(a) # transpose of matrix 'a'
# print(np.shape(aT)) # (2, 3)

# eye(n) is used for (nxn) Identity matrix
b=2*np.eye(3) # 2 * Identity matrix
# print(np.shape(b)) # (3, 3)

c = b*a
# print(np.shape(c)) # (3, 2)

l= lu(a)
print(l)
```

Listing 1.25: LU Factorization of Matrix

```
# scipyEx.py
import numpy as np
# import LU factorization command from scipy.linalg
from scipy.linalg import lu

#define matrix 'a'
a= np.mat('1, 1, 1; 3, 4, 6; 2, 5, 4') # define matrix

# perform LU factorization and
# save the values in p, l and u as it returns 3 values
[p, l, u]= lu(a)

# print values of p, l and u
print("p = ", p)
print("l = ", l)
print("u = ", np.round(l,2))


print("Type of P: ", type(p)) #type of p: ndarray
# p*l*u will give wrong results
# because types are not matrix (but ndarray) as shown above
r = p.dot(l).dot(u)
print("r = ", r)

#for p*l*u we need to change the ndarray to matrix type as below,
print("Type of P after np.mat: ", type(np.mat(p)))
m = np.mat(p)*np.mat(l)*np.mat(u)
print("m = ", m)

'''
Outputs:

p =  [[ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]]

l =  [[ 1.          0.          0.        ]
 [ 0.66666667  1.          0.        ]
 [ 0.33333333 -0.14285714  1.        ]]

u =  [[ 1.    0.    0.  ]
```

(continues on next page)

(continued from previous page)

```
 [ 0.67  1.    0.  ]
 [ 0.33 -0.14  1.  ]]

Type of P:  <class 'numpy.ndarray'>

r =  [[ 1.  1.  1.]
 [ 3.  4.  6.]
 [ 2.  5.  4.]]

Type of P after np.mat:  <class 'numpy.matrixlib.defmatrix.matrix'>

m =  [[ 1.  1.  1.]
 [ 3.  4.  6.]
 [ 2.  5.  4.]]
'''
```

1.8 Good practices

As oppose to other programming languages, Python provides various ways to iterate over the list, which are shown in this section.

1.8.1 Avoid range command

```
# multiply 2 to all elements of arr
arr = [10, 20, 30]

# bad practice
for i in range(len(arr)):
    print(2*arr[i])  # 20, 40, 60

# good practices
for i in arr:
    print(2*i)  # 20, 40, 60

# print in reverse order
for i in reversed(arr):
    print(2*i)  # 60, 40, 20
```

1.8.2 Enumerate

In previous case, we do not have the access over index. Use 'enumerate' to get access to index as well,

```
# multiply 2 to all elements of arr
arr = [10, 20, 30]

for i, a in enumerate(arr):
    print(i, ':', 2*a)
    # 0 : 20
    # 1 : 40
    # 2 : 60
```

1.8.3 Loop in sorted order

```
# multiply 2 to all elements of arr,
# but in sorted order
arr = [10, 30, 50, 20]

for i in sorted(arr):
    print(2*i)  # 20, 40, 60, 100

# in reversed sorted order
for i in sorted(arr, reverse=True):
    print(2*i)  # 100, 60, 40, 20
```

1.8.4 Loop over keys

```
dc = { 'Toy':3, 'Play':4, 'Games':5}

# print keys of dictionaries
for d in dc:
    print(d)
```

1.8.5 Loop over keys and values

```
dc = { 'Toy':3, 'Play':4, 'Games':5}

# print keys, values of dictionaries
for k, v in dc.items():
    print(k, v)
    # Toy 3
    # Play 4
    # Games 5
```

1.8.6 Create dictionaries from lists

```
k = ['Toy', 'Game', 'Tiger']
v = ['Toys', 'Games', 'Tigers']

#create dict
dc = dict(zip(k, v))
print(dc)
# {'Game': 'Games', 'Tiger': 'Tigers', 'Toy': 'Toys'}

d = dict(enumerate(v))
print(d)
# {0: 'Toys', 1: 'Games', 2: 'Tigers'}
```

1.8.7 Looping and modifying the list simultaneously

We need to make a copy of the list for such operations as shown below,

```
# loopUpdate.py

animals = ['tiger', 'cat', 'dog']
am = animals.copy()
```

(continues on next page)

(continued from previous page)

```
# below line will go in infinite loop
# for a in animals:
for a in am:
    if len(a) > 3:
        animals.append(a)

print(animals)
```

Or we can use 'animals[:]' in the for loop, instead of 'animal' as shown below,

```
# loopUpdate.py

animals = ['tiger', 'cat', 'dog']

for a in animals[:]:
    if len(a) > 3:
        animals.append(a)

print(animals)
```

1.8.8 Check items in the list

The 'in' keyword can be used with 'if' statement, to check the value in the list,

```
# loopUpdate.py

def testNumber(num):
    if num in [1, 3, 5]:
        print("Thanks")
    else:
        print("Number is not 1, 3 or 5")

testNumber(3)
testNumber(4)
```

1.8.9 Unpacking

Any iterable i.e. list, tuple or set can be unpacked using assignment operator as below,

```
>>> x = [1, 2, 3]
>>> a, b, c = x
>>> a
1
>>> b
2
```

```
>>> student  = ["Tom", 90, 95, 98, 30]
>>> Name, *Marks, Age = student
>>> Marks
[90, 95, 98]
```

```
>>> y = (1, "Two", 3, ("Five", "Six", "Seven"))
>>> a, *b, (*c, d) = y
>>> d
'Seven'
```

(continues on next page)

(continued from previous page)

```
>>> c
['Five', 'Six']
```

1.8.10 Update variables

```
x = 3
y = 2
z = 5
x, y, z = y, z, x

print(x, y, z)  # 2, 5, 3
```

1.9 Object oriented programming

Object oriented programming (OOP) increases the re-usability of the code. Also, the codes become more manageable than non-OOP methods. But, it takes proper planning, and therefore longer time, to write the codes using OOP method. In this chapter, we will learn various terms used in OOP along with their usages with examples.

1.9.1 Class and object

A 'class' is user defined template which contains variables, constants and functions etc.; whereas an 'object' is the instance (or variable) of the class. In simple words, a class contains the structure of the code, whereas the object of the class uses that structure for performing various tasks, as shown in this section.

1.9.2 Create class and object

Class is created using keyword 'class' as shown in Line 4 of Listing 1.26, where the class 'Jungle' is created. **As a rule, class name is started with uppercase letter, whereas function name is started with lowercase letter**. Currently, this class does not have any structure, therefore keyword 'pass' is used at Line 5. Then, at Line 8, an object of class i.e. 'j' is created; whose value is printed at Line 9. This print statement prints the class-name of this object along with it's location in the memory (see comments at Line 9).

Listing 1.26: Create class and object

```
1 #ex1.py
2
3 #class declaration
4 class Jungle:
5     pass
6
7 # create object of class Jungle
8 j = Jungle()
9 print(j)  # <__main__.Jungle object at 0x004D6970>
```

1.9.2.1 Add function to class

Now, we will add one function 'welcomeMessage' in the class. The functions inside the class are known as '**methods**'. The functions inside the class are the normal functions (nothing special about them), as we can see at Lines 5-6. **To use the variables and functions etc. outside the class, we need to create the object of the class first, as shown in Line 9, where object 'j' is created**. When we create teh object of a class, then all the functions and variables of that class is attached to the object and can be used by that object;

e.g. the object 'j' can now use the function 'welcomeMessage' using '.' operator, as shown in Line 10. Also, 'self' is used at Line 6, which is discussed in Section Section 1.9.2.2.

Listing 1.27: Add function to class

```
1  #ex2.py
2
3  #class declaration
4  class Jungle:
5      def welcomeMessage(self):
6          print("Welcome to the Jungle")
7
8  # create object of class Jungle
9  j = Jungle()
10 j.welcomeMessage() # Welcome to the Jungle
```

1.9.2.2 Constructor

The '__init__' method is used to define and initialize the class variables. This ''__init__' method is known as **Constructor** and the variables are known as **attributes**. Note that, the **self** keyword is used in the 'init function' (Line 6) along with the name of the variables (Line 7). Further All the functions, should have first parameter as 'self' inside the class. Although we can replace the word 'self' with any other word, but it is good practice to use the word 'self' as convention.

Explanation Listing 1.28

> Whenever, the object of a class is create then all the attributes and methods of that class are attached to it; and **the constructor i.e. '__init__' method is executed automatically**. Here, the constructor contains one variable i.e. 'visitorName' (Line 7) and one input parameter i.e. 'name' (Line 6) whose value is initialized with 'unknown'. Therefore, when the object 'j' is created at Line 13, the value 'Meher' will be assigned to parameter 'name' and finally saved in 'visitorName' as constructor is executed as soon as the object created. Further, if we create the object without providing the name i.e. 'j = Jungle()', then default value i.e. 'unknown' will be saved in attribute 'visitorName'. Lastly, the method 'welcomeMessage' is slightly updated, which is now printing the name of the 'visitor' (Line 10) as well.

Listing 1.28: Constructor with default values

```
1  #ex3.py
2
3  #class declaration
4  class Jungle:
5      #constructor with default values
6      def __init__(self, name="Unknown"):
7          self.visitorName = name
8
9      def welcomeMessage(self):
10         print("Hello %s, Welcome to the Jungle" % self.visitorName)
11
12 # create object of class Jungle
13 j = Jungle("Meher")
14 j.welcomeMessage() # Hello Meher, Welcome to the Jungle
15
16 # if no name is passed, the default value i.e. Unknown will be used
17 k = Jungle()
18 k.welcomeMessage() # Hello Unknown, Welcome to the Jungle
```

1.9.2.3 Define 'main' function

The above code can be written in 'main' function (Lines 12-19) using standard-boiler-plate (Lines 22-23), which makes the code more readable, as shown in Listing 1.29. This boiler-plate tells the Python-interpretor that the 'main' is the starting point of the code.

Listing 1.29: main function

```
1  #ex4.py
2
3  #class declaration
4  class Jungle:
5      #constructor with default values
6      def __init__(self, name="Unknown"):
7          self.visitorName = name
8
9      def welcomeMessage(self):
10         print("Hello %s, Welcome to the Jungle" % self.visitorName)
11
12 def main():
13     # create object of class Jungle
14     j = Jungle("Meher")
15     j.welcomeMessage() # Hello Meher, Welcome to the Jungle
16
17     # if no name is passed, the default value i.e. Unknown will be used
18     k = Jungle()
19     k.welcomeMessage() # Hello Unknown, Welcome to the Jungle
20
21 # standard boilerplate to set 'main' as starting function
22 if __name__=='__main__':
23     main()
```

1.9.2.4 Keep classes in separate file

To make code more manageable, we can save the class-code (i.e. class Jungle) and application-files (i.e. main) in separate file. For this, save the class code in 'jungleBook.py' file, as shown in Listing 1.30; whereas save the 'main()' in 'main.py' file as shown in Listing 1.31. Since, class is in different file now, therefore we need to import the class to 'main.py file' using keyword '**import**' as shown in Line 4 of Listing 1.31.

Warning: Here, we kept the class and main function in separate files. It is not a good idea keep these small related-codes separate like this. We will learn to manage the code as we will write some big codes in the tutorial.

Listing 1.30: Save classes in separate file

```
1  #jungleBook.py
2
3  #class declaration
4  class Jungle:
5      #constructor with default values
6      def __init__(self, name="Unknown"):
7          self.visitorName = name
8
9      def welcomeMessage(self):
10         print("Hello %s, Welcome to the Jungle" % self.visitorName)
```

Listing 1.31: Import class to main.py

```
1  #main.py
2
3  #import class 'Jungle' from jungleBook.py
4  from jungleBook import Jungle
5
6  def main():
7      # create object of class Jungle
8      j = Jungle("Meher")
9      j.welcomeMessage() # Hello Meher, Welcome to the Jungle
10
11     # if no name is passed, the default value i.e. Unknown will be used
12     k = Jungle()
13     k.welcomeMessage() # Hello Unknown, Welcome to the Jungle
14
15 # standard boilerplate to set 'main' as starting function
16 if __name__=='__main__':
17     main()
```

1.9.3 Inheritance

Suppose, we want to write a class 'RateJungle' in which visitor can provide 'rating' based on their visiting-experience. If we write the class from the starting, then we need define attribute 'visitorName' again; which will make the code repetitive and unorganizable, as the visitor entry will be at multiple places and such code is more prone to error. With the help of inheritance, we can avoid such duplication as shown in Listing 1.32; where class Jungle is inherited at Line 12 by the class 'RateJungle'. Now, when the object 'r' of class 'RateJungle' is created at Line 7 of Listing 1.33, then this object 'r' will have the access to 'visitorName' as well (which is in the parent class).

Listing 1.32: Inheritance

```
1  #jungleBook.py
2
3  #class declaration
4  class Jungle:
5      #constructor with default values
6      def __init__(self, name="Unknown"):
7          self.visitorName = name
8
9      def welcomeMessage(self):
```

(continues on next page)

(continued from previous page)

```
        print("Hello %s , Welcome to the Jungle" % self.visitorName)

class RateJungle(Jungle):
    def __init__(self, name, feedback):
        # feedback (1-10) :  1 is the best.
        self.feedback = feedback # Public Attribute

        # inheriting the constructor of the class
        super().__init__(name)

    # using parent class attribute i.e. visitorName
    def printRating(self):
        print("Thanks %s for your feedback" % self.visitorName)
```

Listing 1.33: Usage of parent-class method and attributes in child-class

```
#main.py

## import class 'Jungle' and 'RateJungle' from jungleBook.py
from jungleBook import Jungle, RateJungle

def main():
    r = RateJungle("Meher", 3)

    r.printRating() # Thanks Meher for your feedback

    # calling parent class method
    r.welcomeMessage() # Hello Meher, Welcome to the Jungle

# standard boilerplate to set 'main' as starting function
if __name__=='__main__':
    main()
```

1.9.4 Polymorphism

In OOP, we can use same name for methods and attributes in different classes; the methods or attributes are invoked based on the object type; e.g. in Listing 1.34, the method 'scarySound' is used for class 'Animal' and 'Bird' at Lines 4 and 8 respectively. Then object of these classes are created at Line 8-9 of Listing 1.35. Finally, method 'scarySound' is invoked at Lines 12-13; here Line 13 is the object of class Animal, therefore method of that class is invoked and corresponding message is printed. Similarly, Line 14 invokes the 'scaryMethod' of class Bird and corresponding line is printed.

Listing 1.34: Polymorphism example with function 'move'

```
#scarySound.py

class Animal:
    def scarySound(self):
        print("Animals are running away due to scary sound.")

class Bird:
    def scarySound(self):
        print("Birds are flying away due to scary sound.")

# scaryScound is not defined for Insect
class Insect:
    pass
```

Listing 1.35: Polymorphism: move function works in different ways for different class-objects

```
#main.py

## import class 'Animal, Bird' from scarySound.py
from scarySound import Animal, Bird

def main():
    # create objects of Animal and Bird class
    a = Animal()
    b = Bird()

    # polymorphism
    a.scarySound() # Animals are running away due to scary sound.
    b.scarySound() # Birds are flying away due to scary sound.

# standard boilerplate to set 'main' as starting function
if __name__=='__main__':
    main()
```

1.9.5 Abstract class and method

Abstract classes are the classes which contains one or more abstract method; and abstract methods are the methods which does not contain any implemetation, but the child-class need to implement these methods otherwise error will be reported. In this way, we can force the child-class to implement certain methods in it. We can define, abstract classes and abstract method using keyword 'ABCMeta' and 'abstractmethod' respectively, as shown in Lines 6 and 15 respectively of Listing 1.36. Since, 'scarySound' is defined as abstractmethod at Line 15-17, therefore it is compulsory to implement it in all the subclasses.

Note: Look at the class 'Insect' in Listing 1.34, where 'scarySound' was not defined but code was running correctly; but now the 'scarySound' is abstractmethod, therefore it is compulsory to implement it, as done in Line 16 of Listing 1.37.

Listing 1.36: Abstract class and method

```
#jungleBook.py

from abc import ABCMeta, abstractmethod

#Abstract class and abstract method declaration
class Jungle(metaclass=ABCMeta):
    #constructor with default values
    def __init__(self, name="Unknown"):
        self.visitorName = name

    def welcomeMessage(self):
        print("Hello %s, Welcome to the Jungle" % self.visitorName)

    # abstract method is compulsory to defined in child-class
    @abstractmethod
    def scarySound(self):
        pass
```

Listing 1.37: Abstract methods are compulsory to define in child-class

```
#scarySound.py

from jungleBook import Jungle

class Animal(Jungle):
    def scarySound(self):
        print("Animals are running away due to scary sound.")

class Bird(Jungle):
    def scarySound(self):
        print("Birds are flying away due to scary sound.")

# since Jungle is defined as metaclass
# therefore all the abstract methods are compulsory be defined in child class
class Insect(Jungle):
    def scarySound(self):
        print("Insects do not care about scary sound.")
```

Listing 1.38: Main function

```
#main.py

## import class 'Animal, Bird' from scarySound.py
from scarySound import Animal, Bird, Insect

def main():
    # create objects of Animal and Bird class
    a = Animal()
    b = Bird()
    i = Insect()

    # polymorphism
    a.scarySound() # Animals are running away due to scary sound.
    b.scarySound() # Birds are flying away due to scary sound.
    i.scarySound() # Insects do not care about scary sound.

# standard boilerplate to set 'main' as starting function
if __name__=='__main__':
    main()
```

1.9.6 Public and private attribute

There is not concept of private attribute in Python. All the attributes and methods are accessible to end users. But there is a convention used in Python programming i.e. if a variable or method name starts with '_', then users should not directly access to it; there must be some methods provided by the class-author to access that variable or method. Similarly, '__' is designed for renaming the attribute with class name i.e. the attribute is automatically renamed as '_className__attributeName'. This is used to avoid conflict in the attribute names in different classes, and is useful at the time of inheritance, when parent and child class has same attribute name.

Listing 1.39 and Listing 1.40 show the example of attributes along with the methods to access them. Please read the comments to understand these codes.

Listing 1.39: Public and private attribute

```
#jungleBook.py

```

(continues on next page)

(continued from previous page)

```
#class declaration
class Jungle:
    #constructor with default values
    def __init__(self, name="Unknown"):
        self.visitorName = name

    def welcomeMessage(self):
        print("Hello %s , Welcome to the Jungle" % self.visitorName)

class RateJungle:
    def __init__(self, feedback):
        # feedback (1-10) :  1 is the best.
        self.feedback = feedback # Public Attribute

        # Public attribute with single underscore sign
        # Single _ signifies that author does not want to acecess it directly
        self._staffRating = 50

        self.__jungleGuideRating = 100 # Private Attribute

        self.updateStaffRating() # update Staff rating based on feedback
        self.updateGuideRating() # update Guide rating based on feedback

    def printRating(self):
        print("Feedback : %s \tGuide Rating: %s \tStaff Rating: %s "
            % (self.feedback, self.__jungleGuideRating, self._staffRating))

    def updateStaffRating(self):
        """ update Staff rating based on visitor feedback"""
        if self.feedback < 5 :
            self._staffRating += 5
        else:
            self._staffRating -= 5

    def updateGuideRating(self):
        """ update Guide rating based on visitor feedback"""
        if self.feedback < 5 :
            self.__jungleGuideRating += 10
        else:
            self.__jungleGuideRating -= 10
```

Listing 1.40: Accessing public and private attributes

```
#main.py

# import class 'Jungle' and 'RateJungle' from jungleBook.py
from jungleBook import Jungle, RateJungle

def main():
    ## create object of class Jungle
    j = Jungle("Meher")
    j.welcomeMessage() # Hello Meher, Welcome to the Jungle

    r = RateJungle(3)
    r.printRating() # Feedback : 3  Guide Rating: 110  Staff Rating: 55

    # _staffRating can be accessed "directly", but not a good practice.
    # Use the method which is provided by the author
    # e.g. below is the bad practice
    r._staffRating = 30 # directly change the _staffRating
    print("Staff rating : ", r._staffRating) # Staff rating :  30
```

(continues on next page)

(continued from previous page)

```
19
20      ## access to private attribute is not allowed
21      ## uncomment following line to see error
22      # print("Jungle Guide rating : ", r.__jungleGuideRating)
23
24      ## private attribute can still be accessed as below,
25      ## objectName._className.__attributeName
26      print ("Guide rating : ",  r._RateJungle__jungleGuideRating) # Guide rating :  110
27
28  # standard boilerplate to set 'main' as starting function
29  if __name__=='__main__':
30      main()
```

1.9.7 Class Attribute

Class attribute is the variable of the class (not of method) as shown in Line 7 of Listing 1.41. This attribute can be available to all the classes without any inheritance e.g. At Line 44, the class Test (Line 41) is using the class-attribute 'sum_of_feedback' of class Jungle (Line 7). Note that, we need to use the class name to access the class attribute e.g. Jungle.sum_of_feedback (Lines 30 and 44).

Listing 1.41: Class attributes and it's access

```
1   #jungleBook.py
2
3   #class declaration
4   class Jungle:
5       # class attribute
6       # use __sum_of_feedback to hide it from the child class
7       sum_of_feedback = 0.0
8
9       #constructor with default values
10      def __init__(self, name="Unknown"):
11          self._visitorName = name # please do not access directly
12
13      def welcomeMessage(self):
14          print("Hello %s, Welcome to the Jungle" % self.visitorName)
15
16      def averageFeedback(self):
17          #average feedback is hided for the the child class
18          self.__avg_feedback = Jungle.sum_of_feedback/RateJungle.total_num_feedback
19          print("Average feedback : ", self.__avg_feedback)
20
21  class RateJungle(Jungle):
22      # class attribute
23      total_num_feedback = 0
24
25      def __init__(self, name, feedback):
26          # feedback (1-10) :  1 is the best.
27          self.feedback = feedback # Public Attribute
28
29          # add new feedback value to sum_of_feedback
30          Jungle.sum_of_feedback += self.feedback
31          # increase total number of feedback by 1
32          RateJungle.total_num_feedback += 1
33
34          # inheriting the constructor of the class
35          super().__init__(name)
36
37      # using parent class attribute i.e. visitorName
```

(continues on next page)

(continued from previous page)

```
38      def printRating(self):
39          print("Thanks %s for your feedback" % self._visitorName)
40
41  class Test:
42      def __init__(self):
43          # inheritance is not required for accessing class attribute
44          print("sum_of_feedback (Jungle class attribute) : ", Jungle.sum_of_feedback)
45          print("total_num_feedback (RateJungle class attribute) : ", RateJungle.total_num_feedback)
```

Listing 1.42: Main program

```
1   #main.py
2
3   ## import class 'Jungle', 'RateJungle' and 'Test' from jungleBook.py
4   from jungleBook import Jungle, RateJungle, Test
5
6   def main():
7       r = RateJungle("Meher", 3)
8       s = RateJungle("Krishna", 2)
9
10      r.averageFeedback() # Average feedback :  2.5
11
12
13      # Test class is using other class attributes without inheritance
14      w = Test()
15      ''' sum_of_feedback (Jungle class attribute) :  5.0
16          total_num_feedback (RateJungle class attribute) :  2
17      '''
18
19  # standard boilerplate to set 'main' as starting function
20  if __name__=='__main__':
21      main()
```

1.9.8 Special methods

There are some special method, which are invoked under certain cases e.g. __init__ method is invoked, when an object of the instance is created. In this section, we will see some more special methods.

1.9.8.1 __init__ and __del__

The __init__ method is invoked when object is created; whereas __del__ is always invoked at the end of the code; e.g. we invoke the 'del' at Line 21 of Listing 1.43, which deletes object 's1' and remaining objects are printed by Line 13. But, after Line 25, there is no further statement, therefore the 'del' command will automatically executed, and results at Lines 31-32 will be displayed. The 'del' command is also known as '**destructor**'.

Listing 1.43: __init__ and __del__ function

```
1   # delEx.py
2
3   class Student:
4       totalStudent = 0
5
6       def __init__(self, name):
7           self.name = name
8           Student.totalStudent += 1
9           print("Total Students (init) : ", self.totalStudent)
10
```

(continues on next page)

(continued from previous page)

```
11      def __del__(self):
12          Student.totalStudent -= 1
13          print("Total Students (del) : ", self.totalStudent)
14
15  def main():
16      s1 = Student("Meher") # Total Students (init) :  1
17      s2 = Student("Krishna") # Total Students (init) :  2
18      s3 = Student("Patel") # Total Students (init) :  3
19
20      ## delete object s1
21      del s1 # Total Students (del) :  2
22
23      # print(s1.name) # error because s1 object is deleted
24      print(s2.name) # Krishna
25      print(s3.name) # Patel
26
27      ## since there is no further statements, therefore
28      ## 'del' will be executed for all the objects and
29      ## following results will be displayed
30
31      # Total Students (del) :  1
32      # Total Students (del) :  0
33
34  # standard boilerplate to set 'main' as starting function
35  if __name__=='__main__':
36      main()
```

1.9.8.2 __str__

When __str__ is defined in the class, then 'print' statement for object (e.g. print(j) at Line 11 of Listing 1.44), will execute the __str__ statement, instead of printing the address of object, as happened in Listing 1.26. This statement is very useful for providing the useful information about the class using print statement.

Listing 1.44: __str__ method is executed when 'print' statement is used for object

```
1   #strEx.py
2
3   #class declaration
4   class Jungle:
5       def __str__(self):
6           return("It is an object of class Jungle")
7
8   def main():
9       # create object of class Jungle
10      j = Jungle()
11      print(j)  # It is an object of class Jungle
12
13  # standard boilerplate to set 'main' as starting function
14  if __name__=='__main__':
15      main()
```

1.9.8.3 __call__

The __call__ method is executed, when object is used as function, as shown in Line 20 of Listing 1.45; where object 'd' is used as function i.e. d(300).

Listing 1.45: __call__ method is executed when object is used as function

```
# callEx.py

#class declaration
class CalculatePrice:
    # discount in %
    def __init__(self, discount):
        self.discount = discount

    def __call__(self, price):
        discountPrice = price - price*self.discount/100
        return (price, discountPrice)

def main():
    # create object of class CalculatePrice with 10% discount
    d = CalculatePrice(10)

    # using object as function i.e. d(300)
    # since two variables are return by call fuction, therefore
    # unpack the return values in two variables
    price, priceAfterDiscount =  d(300)
    print("Original Price: %s,  Price after discount : %s "
                    % (price, priceAfterDiscount))

    ## or use below method, if you do not want to unpack the return values
    # getPrices =  d(300)
    # print("Original Price: %s,  Price after discount : %s "
    #                 % (getPrices[0], getPrices[1]))

# standard boilerplate to set 'main' as starting function
if __name__=='__main__':
    main()
```

1.9.8.4 __dict__ and __doc__

__dict__ is used to get the useful information about the class (Line 21); whereas __doc__ prints the docstring of the class (Line 30).

Listing 1.46: __dict__ and __doc__

```
#dictEx.py

#class declaration
class Jungle:
    """ List of animal and pet information
        animal = string
        isPet = string
    """
    def __init__(self, animal="Elephant", isPet="yes"):
        self.animal =  animal
        self.isPet = isPet

def main():
    # create object of class Jungle
    j1 = Jungle()
    print(j1.__dict__) # {'isPet': 'yes', 'animal': 'Elephant'}

    j2 = Jungle("Lion", "No")
```

(continues on next page)

(continued from previous page)

```
    print(j2.__dict__) # {'isPet': 'No', 'animal': 'Lion'}

    print(Jungle.__dict__)
    """ {'__doc__': '__doc__': ' List of animal and pet information \n
                        animal = string\n      isPet = string\n    ',
        '__weakref__': <attribute '__weakref__' of 'Jungle' objects>,
        '__module__': '__main__',
        '__dict__': <attribute '__dict__' of 'Jungle' objects>,
        '__init__': <function Jungle.__init__ at 0x00466738>}
    """

    print(Jungle.__doc__)
    """List of animal and pet information
        animal = string
        isPet = string
    """

# standard boilerplate to set 'main' as starting function
if __name__=='__main__':
    main()
```

1.9.8.5 __setattr__ and __getattr__

Method __setattr__ is executed, whenever we set the value of an attribute. __setattr__ can be useful for validating the input-types before assigning them to attributes as shown in Line 9 of Listing 1.47. Please read the comments of Listing 1.47 for better understanding. Similarly, __getattr__ is invoked whenever we try to access the value of an attribute, **which is not in the dictionary**.

Listing 1.47: __setattr__ and __getattr__

```
# setAttr.py
class StudentID:
    def __init__(self, id, name, age = "30"):
        self.id = id
        self.firstName = name
        self.age = age

    # all the init parameters need to be specified in 'setattr'
    def __setattr__(self, name, value):
        if(name == "id"): # setting id
            if isinstance(value, int) and value > 0 :
                self.__dict__["id"] = value
            else:
                # print("Id must be positive integer")
                raise TypeError("Id must be positive integer")
        elif (name == "firstName"): # setting firstName
            self.__dict__["firstName"] = value
        else: # setting age
            self.__dict__[name] = value

    # getattr is executed, when attribute is not found in dictionary
    def __getattr__(self, name):
        raise AttributeError("Attribute does not exist")

def main():
    s1 = StudentID(1, "Meher")
    print(s1.id, s1.firstName, s1.age) # 1 Meher 30

    ## uncomment below line to see the "TypeError" generated by 'setattr'
```

(continues on next page)

(continued from previous page)

```
    # s2 = StudentID(-1, "Krishna", 28)
    """
    Traceback (most recent call last):
    [...]
    raise TypeError("Id must be positive integer")
    """

    s3 = StudentID(1, "Krishna", 28)
    print(s3.id, s3.firstName, s3.age) # 1 Krishna 28

    ## uncomment below line to see the "AttributeError" generated by 'getattr'
    # print(s3.lastName) # following message will be displayed
    """ Traceback (most recent call last):
        [...]
        AttributeError: Attribute does not exist
    """
# standard boilerplate to set 'main' as starting function
if __name__=='__main__':
    main()
```

1.10 Conclusion

In this chapter, we learn various features of Python along with object oriented programming. Also, we learn some of the good coding practices in Python. Further, We saw that there is no concept of private attributes in Python. Lastly, we discuss various special methods available in Python which can enhance the debugging and error checking capability of the code. We will see all these features in details in the subsequent chapters.

Chapter 2

Virtual environment, Package and Distribution

2.1 Virtual environment

First install the virtual environment,

```
$ pip install virtualenv

(for anaconda)
$ conda install virtualenv
```

- Create a virtual environment with name 'pythondsp' as below,

```
(using Internet)
$ virtualenv pythondsp -p python3.6

(create locally)
$ virtualenv pythondsp -p /location/to/bin/python3.6
e.g.
$ virtualenv pythondsp -p /home/anaconda3/bin/python3.6
```

Note: Test the virtualenv. Old version of matplotlib is used, as newer is not installed properly

```
virtualenv test -p python3.6

source test/bin/activate

pip install matplotlib==1.5.3
```

- We need to activate the environment to use it. If it is activate, then the environment name will be shown before the $ sign,

```
$ source location/to/environment/bin/activate
e.g.
$ source /home/env/pythondsp/bin/activate
(pythondsp) $
```

- Next, install packages in the environment. We already used the 'pip' command to install 'virtualenv'. The 'setup.py' method is shown in Section 2.3, where we converted our package to distributable package.

Listing 2.1: Install packages using 'pip' and 'setup.py'

```
$ pip install name_of_package

(install from the file 'requirements.txt')
$ cd location/to/requirements
$ pip install -r requirements.txt

(or using 'setup.py' file)
$ cd location/to/setup.py
$ python setup.py install
```

- See the installed package in the environment,

```
$ pip freeze
```

- Save the list of packages in the 'requirements.txt'

```
$ pip freeze > requirements.txt
```

2.2 Packages

In this section, we will learn to create the 'packages', which are nothing but the collection of modules under the same name.

2.2.1 Location of installed packages

Before creating our own package, let's see the location of the installed packages. The installed packages (using pip command) are saved in the folder 'lib/python3.6/site-packages', as shown below,

```
(for environment 'pythondsp')
pythondsp/lib/python3.6/site-packages

(without environment)
/home/anaconda3/lib/python3.6/site-packages
```

Apart from current working directory, the Python look for the files in the folder 'site-packages'. We will understand this in Section 2.2.4. The complete list of the directories can be seen using 'sys.path' command.

```
(pythondsp) $ python

>>> import sys

>>> sys.path
[
    [...]
    '/home/pythondsp/lib/python3.6/site-packages',
    [...]
]
```

2.2.2 Create files

In this section, we will write some Python codes, and then in next section, we will convert these Python modules into a package.

- Activate the environment, and create a folder with any name e.g. 'bucket' at desired location

```
(pythondsp) $ mkdir -p ~/Desktop/bucket
(pythondsp) $ cd ~/Desktop/bucket
```

Now create two files inside the folder with following contents,

```
# bucket/my_calc.py

def sum2Num(x, y):
    return (x+y)

def diff2Num(x, y):
    return (x-y)
```

```
# bucket/my_work.py

from my_calc import sum2Num, diff2Num

x = 10
y = 5

print("{0} + {1} = {2}".format(x, y, sum2Num(x, y)))
print("{0} - {1} = {2}".format(x, y, diff2Num(x, y)))
```

Next, check execute 'my_work.py' to check the setup,

Note: First 'import my_work' will import everything from the file, therefore 'print' statement will be executed. But the second import will import the values from the 'cache' (not from the file), therefore 'print' statements from the file will not be executed.

```
(pythondsp) $ python my_work.py
10 + 5 = 15
10 - 5 = 5
```

Also, check the below commands,

```
(pythondsp) $ python

>>> from my_calc import sum2Num
>>> sum2Num(2, 4)
6

>>> import my_work
10 + 5 = 15
10 - 5 = 5
>>> import my_work
>>>
```

2.2.3 Packages

In this section, we will convert the Python modules into the package.

Note:

- Package name should be unique, so that it will not collide with other package names.

- Create another folder inside the folder 'bucket' with any desired name e.g. 'wolfpack', and move the 'python files' inside it. After running below commands, we will have following folder structure (after excluding the

folder '__pycache__),

Note: The folder (i.e. 'wolfpack') inside the 'root folder (i.e. bucket)' is called the 'package' and needs special settings to use it with root-folder.

```
bucket/
└── wolfpack
    ├── my_calc.py
    └── my_work.py
```

```
(pythondsp) $ mkdir wolfpack
(pythondsp) $ mv my_work.py my_calc.py wolfpack
```

- Now, run the shell commands again as shown below. Note that the command at Line 4 is working fine, but command at Line 6 is generating error.

Listing 2.2: Import package and error

```
1  (pythondsp) $ python
2
3  >>> from wolfpack.my_calc import sum2Num
4  >>> sum2Num(10, 2)
5  12
6  >>> from wolfpack import my_work
7  Traceback (most recent call last):
8    File "<stdin>", line 1, in <module>
9    File "/home/meher/Desktop/bucket/wolfpack/my_work.py", line 3, in <module>
10     from my_calc import sum2Num, diff2Num
11 ModuleNotFoundError: No module named 'my_calc'
```

Note: The error at Line 6 is 'No module named 'my_calc', as we are **running the command from the 'root directory** (i.e. bucket)', therefore the line 'import my_calc' in 'my_work.py' will look for it in the **root directory only** (not inside the '**wolfpack**'). The error can be removed by modifying the code as below,

In the below code, we used 'from .my_calc import sum2Num, diff2Num'. The 'dot operator' tells python to look in the current directory (i.e. wolfpack), not in the 'root directory'.

```
# bucket/wolfpack/my_work.py

from .my_calc import sum2Num, diff2Num

x = 10
y = 5

print("{0} + {1} = {2}".format(x, y, sum2Num(x, y)))
print("{0} - {1} = {2}".format(x, y, diff2Num(x, y)))
```

- Close the Python terminal and open it again. Then execute the commands and it will work fine,

```
(pythondsp) $ python

>>> from wolfpack import my_work
10 + 5 = 15
10 - 5 = 5
```

Warning:

- Now, the folder 'wolfpack' is a 'package' and the files inside it can not be executed directly, as these files have 'dot' operators in the 'import' statement. Following error will be generated if we run the 'package-module' directly.

```
(pythondsp) $ cd wolfpack/

(pythondsp) $ ls
my_calc.py  my_work.py

(pythondsp) $ python my_work.py
Traceback (most recent call last):
  File "my_work.py", line 3, in <module>
    from .my_calc import sum2Num, diff2Num
ModuleNotFoundError: No module named '__main__.my_calc';
```

- The files in the package module can be executed through root-folders (i.e. bucket) by importing the modules.
- Also, the folder 'wolfpack' can be called as a python-file in the codes i.e. 'import wolfpack'
- We will use following terms for the two folders,
  - Root directory : 'bucket'
  - Package directory : 'wolfpack'

2.2.4 Globally available package

If we 'cut and paste' the package folder (i.e. wolfpack) inside the folder 'site-packages', then it will be available globally in the environment, i.e. it can be imported into any project at any location.

Note: Do not cut and paste the folder now. We will create the package in Section 2.3 and use 'setup.py' command to install the package.

2.2.5 __init__ file

In Listing 2.2, we import the function 'sum2Num' using following command .

```
from wolfpack.my_calc import sum2Num'
```

Currently, we have only two files in the package therefore it's easy to import 'function' like this. But, if we have 100 files with 10 folders in a package, then it will be difficult/inconvenient to remember the import-location of the 'methods'. A better approach is to to use the '__init__.py' file as shown below,

- First go to package folder and create an __init__.py with following content,

```
# wolfpack/__init__.py

# import functions from my_calc
from .my_calc import sum2Num, diff2Num
```

- Now, we can import the commands directly without knowing the file structure,

```
>>> # run from root-folder 'bucket'
>>> from wolfpack import sum2Num, diff2Num
>>> sum2Num(3, 12)
15
```

2.2.6 __all__ in __init__ file

If we want to allow 'import *' option for our package, then we need to add the magic keyword '__all__' in the __init__.py file,

```
1  # wolfpack/__init__.py
2
3  # import sum2Num from my_calc
4  from .my_calc import sum2Num
5
6  # import 'my_calc' and 'my_calc' for 'import *'
7  __all__ = [ "sum2Num",
8              "my_calc",
9              "my_work"
10             ]
```

Now, use the 'import *' command in the Python shell as below. Following items will be imported with * command,

- module 'my_calc'
- module 'my_work'
- function 'sum2Num'

Warning:

- In Line 4 of above code, only 'sum2Num' is imported. If we do not include sum2Num in the '__all__ (Line 7)', then '__all__' will overwrite the Line 4 and 'sum2Num' will not be available. And error (in below code) similar to 'diff2Num(10, 10)' will be shown for sum2Num.

```
(run from the root-folder 'bucket')
(pythondsp) $ python

>>> from wolfpack import *
10 + 5 = 15
10 - 5 = 5
>>>
>>> my_work.x
10
>>> sum2Num(10, 10)
20
>>>
>>> my_calc.diff2Num(25, 5)
20
>>>
>>> diff2Num(10, 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'diff2Num' is not defined
```

2.3 Distribute the package

In this section, we will convert our package to 'distributable package', which can be used by others as well,

Note: We need to create the setup.py file inside the root-folder 'bucket' to generate the 'distribution'. Suppose, if we have the following 'directory structure'.

```
bucket/
├── documentation.txt   # document the package here.
```

(continues on next page)

(continued from previous page)

```
├── my_exec.py          # executable python file
├── my_file1.py         # python file
├── my_file2.py         # python file
├── readme.txt          # add tips here....
├── setup.py            # needs to create a 'distribution'
└── wolfpack            # package
    ├── __init__.py
    ├── item1.py
    └── item2.py
```

Then our setup.py file will be as follows,

```
# setup.py

from distutils.core import setup

setup(name = "wolf",                        # choose any name
    version = "1.0",                        # give version number
    py_modules = ['my_file1', 'my_file2'], # python files
    packages = ['wolfpack'],                # add package here
    scripts = ['my_exec.py'],               # executable
)
```

- Since our project have only 'package' in it, therefore we will use below 'setup.py' configuration,

```
# setup.py

from distutils.core import setup

setup(name = "wolfpack",      # choose any name
    version = "1.0",          # give version number
    packages = ['wolfpack'], # add package here
)
```

- Next, run the setup.py as below and it will create a 'distribution'. It will show some warning as we did not include the 'README.txt' and 'MANIFEST.in' etc.

```
(pythondsp) $ python setup.py sdist

running sdist
running check
warning: check: missing required meta-data: url

warning: check: missing meta-data: either (author and author_email) or (maintainer and maintainer_email)␣
↪must be supplied

warning: sdist: manifest template 'MANIFEST.in' does not exist (using default file list)

warning: sdist: standard file not found: should have one of README, README.txt

writing manifest file 'MANIFEST'
creating wolfpack-1.0
creating wolfpack-1.0/wolfpack
making hard links in wolfpack-1.0...
hard linking setup.py -> wolfpack-1.0
hard linking wolfpack/__init__.py -> wolfpack-1.0/wolfpack
hard linking wolfpack/my_calc.py -> wolfpack-1.0/wolfpack
hard linking wolfpack/my_work.py -> wolfpack-1.0/wolfpack
creating dist
Creating tar archive
```

(continues on next page)

(continued from previous page)

```
removing 'wolfpack-1.0' (and everything under it)
```

- The resultant distribution will be saved as 'wolfpack-1.0.tar.gz' inside the 'dist' folder.
- Extract the 'zipped' file. Then execute the setup.py file in the unzipped folder as below,

```
$ cd dist/wolfpack-1.0
$ python setup.py install
```

- Next, see the 'site-packages' folder for 'wolfpack' package; or use below command to check the list of packages.

```
(pythondsp) $ pip list

[...]
wolfpack (1.0)
```

- Finally, check the working of the package. Create a folder at any location and run the below commands,

```
(pythondsp) $ python

>>> from wolfpack import *
10 + 5 = 15
10 - 5 = 5
>>> sum2Num(10, 10)
20
>>> my_calc.diff2Num(3,1)
2
```

2.4 Conclusion

In this chapter, we learn about the virtual environment and packages. Also, we saw the method by which we can convert the 'package' into the 'distribution'.

Chapter 3

Debugging

3.1 Introdution

In this chapter, the 'sys' module is discussed to read the input from the terminal. Then a little overview of the debugging tool is provided, which will be used in subsequent chapters.

3.2 First code

In the below code, perimeter and area of a circular ring is calculated.

```
# ring.py

from math import pi

metal = "Copper"
radius = 10

perimeter = 2*pi*radius
area = pi * radius**2

print("Metal = ", metal)
print("pi = ", pi)
print("Perimeter = ", perimeter)
print("area = ", area)
```

Following is the output of the above code,

```
$ python ring.py
Metal =  Copper
pi =  3.141592653589793
Perimeter =  62.83185307179586
area =  314.1592653589793
```

3.3 Reading input from command line

In the previous code, the inputs, i.e. radius and metal, are hardwired in the code. In this section, we will modify the code, so that the inputs values can be provided from the termial.

- For this, we need to import 'sys' module.

Note:

– The 'sys.argv' pass the command line argument to Python script.
– The argv[0] is the file name.
– Also, the arguments are passed as string-format, therefore these need to be converted into proper format, as done in Line 16, where the argv[2] is converted into 'float' format.

```
1  # ring.py
2
3  import sys
4
5  from math import pi
6
7  if len(sys.argv) != 3:  # display error message for missing arguments
8      raise SystemExit("usage : ring.py \"metal\" radius")
9
10 # sys.argv[0] is the file name
11 # metal = "Copper"
12 metal = sys.argv[1]
13
14 # radius = 10
15 # input is read as string therefore it is converted into float
16 radius = float(sys.argv[2])
17
18 perimeter = 2*pi*radius
19 area = pi * radius**2
20
21 print("Metal =", metal)
22 # print("pi =", pi)
23 print("Perimeter =", perimeter)
24 print("area =", area)
```

- Now, run the command and we will get the following outputs.

```
$ python ring.py
usage : ring.py "metal" radius

$ python ring.py "Gold" 2
Metal = Gold
Perimeter = 12.566370614359172
area = 12.566370614359172
```

3.4 Debugging

In this section, two basic methods are shown for debugging the code. In the later chapters, we will see use some advance topics such as decorator and descriptor etc. to debug the design.

3.4.1 Run script and go to Python shell

One of the way to debug the code is to use Python shell. For this we can use the '-i' option as shown below. After executing of the code using 'i' option, the Python shell will be open, where we can check the various values or behavior of the implemented logics. We will use this method extensively in the tutorial.

```
$ python -i ring.py "Gold" 2
Metal = Gold
Perimeter = 12.566370614359172
area = 12.566370614359172
>>> print(sys.argv) # print the arguments read from terminal
['ring.py', 'Gold', '2']
```

(continues on next page)

(continued from previous page)

```
>>> print(metal)
Gold
```

3.4.2 Python debugger (pdb)

Another way to debug the code is to use the 'pdb', as shown below,

- Here, python debugger module is imported at Line 3.
- Next, pdb.set_trace is used to set the starting location for the debugging, i.e. the code will stop after reaching this point i.e. Line 15 here.

```
1  # ring.py
2
3  import pdb # Python Debugger
4  import sys
5
6  from math import pi
7
8  if len(sys.argv) != 3: # display error message for missing arguments
9      raise SystemExit("usage : ring.py \"metal\" radius")
10
11 # print arguments
12 print("Entered values: ", sys.argv)
13
14 # manual debugging starts from here
15 pdb.set_trace()
16
17 # sys.argv[0] is the file name
18 # metal = "Copper"
19 metal = sys.argv[1]
20
21 # radius = 10
22 # input is read as string therefore it is converted into float
23 radius = float(sys.argv[2])
24
25 perimeter = 2*pi*radius
26 area = pi * radius**2
27
28 print("Metal =", metal)
29 # print("pi =", pi)
30 print("Perimeter =", perimeter)
31 print("area =", area)
```

- Now, run the code as below. Press 's' and then enter to execute the next line.

```
$ python ring.py "Gold" 2
Entered values:  ['ring.py', 'Gold', '2']
> /ring.py(19)<module>()
-> metal = sys.argv[1]
(Pdb) s
> /ring.py(23)<module>()
-> radius = float(sys.argv[2])
```

3.5 Underscore operator (_)

The Underscore operator stores the last value of the calculation and can be very useful while debugging the code in Python shell.

```
>>> 3 + 2
5
>>> _ * 2
10
```

3.6 Conclusion

In this chapter, we saw the debugging tools which will be used in subsequent chapters. Also, the module 'sys' is discussed for reading the inputs from the terminal.

Chapter 4

Print statement

4.1 Introduction

In this chapter, the print statement is discussed to print the output in nice format.

4.2 Expansion calculation

In the below code, number of days are calculate in which the diameter of the ring becomes greater than or equal to 10 cm.

```
# expansion.py

# find the number of days when radius = 10 cm due to heat-expansion

import sys

if len(sys.argv) != 3: # display error message for missing arguments
    raise SystemExit("usage : ring.py \"metal\" radius")

# sys.argv[0] is the file name
metal = sys.argv[1]

# input is read as string therefore it is converted into float
radius = float(sys.argv[2])

# list of expansion rate for different metal
rate = [0.4, 0.08, 0.05] # [Copper, Gold, Iron]

day = 0
while radius < 10:
    # multiply by correct expansion rate
    if metal == "Copper":
        expansion = radius * rate[0]
    elif metal == "Gold":
        expansion = radius * rate[1]
    elif metal == "Iron":
        expansion = radius * rate[2]
    else:
        print("Enter the correct metal")
        break

    # new radius
```

(continues on next page)

(continued from previous page)

```
    radius += expansion
    day += 1 # increment the number of days by one

# print the number of days
print("Number of days =", day)
```

Following are the outputs for different metals with same radius,

```
$ python expansion.py "Gold" 8
Number of days = 3

$ python expansion.py "Iron" 8
Number of days = 5

$ python expansion.py "Copper" 8
Number of days = 1

$ python expansion.py "Silver" 8
Enter the correct metal
Number of days = 0
```

4.3 Print the expansion

In the below code, the new radius after expansion is printed on the daily basis,

```
# expansion.py

# find the number of days when radius = 10 cm due to heat-expansion

import sys

if len(sys.argv) != 3: # display error message for missing arguments
    raise SystemExit("usage : ring.py \"metal\" radius")

# sys.argv[0] is the file name
metal = sys.argv[1]

# input is read as string therefore it is converted into float
radius = float(sys.argv[2])

# list of expansion rate for different metal
rate = [0.4, 0.08, 0.05] # [Copper, Gold, Iron]

day = 0

print("day, expansion, radius")
while radius < 10:
    # multiply by correct expansion rate
    if metal == "Copper":
        expansion = radius * rate[0]
    elif metal == "Gold":
        expansion = radius * rate[1]
    elif metal == "Iron":
        expansion = radius * rate[2]
    else:
        print("Enter the correct metal")
        break

```

(continues on next page)

(continued from previous page)

```
    # new radius
    radius += expansion
    day += 1 # increment the number of days by one

    # print the data
    print(day, expansion, radius)

# print the number of days
# print("Number of days =", day)
```

Following is the output for Gold ring,

```
$ python expansion.py "Gold" 8
day, expansion, radius
1 0.64 8.64
2 0.6912 9.3312
3 0.746496 10.077696000000001
```

4.4 Formatted output

In the below code, the 'format' option of print statement is used to display the output in more readable form,

```
# expansion.py

# find the number of days when radius = 10 cm due to heat-expansion

import sys

if len(sys.argv) != 3: # display error message for missing arguments
    raise SystemExit("usage : ring.py \"metal\" radius")

# sys.argv[0] is the file name
metal = sys.argv[1]

# input is read as string therefore it is converted into float
radius = float(sys.argv[2])

# list of expansion rate for different metal
rate = [0.4, 0.08, 0.05] # [Copper, Gold, Iron]

day = 0

## use any of the below
## below is not in good format
# print("{0}, {1}, {2}".format("day", "expansion", "radius"))

## "> right aligned" "< left aligned"
## 5s : string with width 5
print("{:>5s} {:>10s} {:>7s}".format("day", "expansion", "radius"))

## old style
# print("%5s %10s %7s" % ("day", "expansion", "radius"))

while radius < 10:
    # multiply by correct expansion rate
    if metal == "Copper":
        expansion = radius * rate[0]
    elif metal == "Gold":
```

(continues on next page)

(continued from previous page)

```
        expansion = radius * rate[1]
    elif metal == "Iron":
        expansion = radius * rate[2]
    else:
        print("Enter the correct metal")
        break

    # new radius
    radius += expansion
    day += 1 # increment the number of days by one

    ## print the data
    ## 5d : 5 digits
    ## 7.2f : 7 digits with 2 decimal points
    print("{:>5d} {:>10.5f} {:>7.2f}".format(day, expansion, radius))

# print the number of days
# print("Number of days =", day)
```

- Following is the outputs of the print statements, which looks better than the output in the previous section,

```
$ python expansion.py "Gold" 8
  day  expansion  radius
    1    0.64000    8.64
    2    0.69120    9.33
    3    0.74650   10.08
```

4.5 Saving results in file

There are two ways to save the results in the file.

- In the first method, we can redirect the output of the print statement to the file (instead of printing then on the terminal). Note that the '>' sign will overwrite the file contents, whereas the '>>' sign will append the content at the end of the file. Also, please see the Unix Guide to learn more commands like 'cat' which is used to display the content of the file.

```
$ python expansion.py "Gold" 8 > data.txt

$ cat data.txt
  day  expansion  radius
    1    0.64000    8.64
    2    0.69120    9.33
    3    0.74650   10.08

$ python expansion.py "Gold" 8 >> data.txt

$ cat data.txt
  day  expansion  radius
    1    0.64000    8.64
    2    0.69120    9.33
    3    0.74650   10.08
  day  expansion  radius
    1    0.64000    8.64
    2    0.69120    9.33
    3    0.74650   10.08
```

- In the other method, we can open the file in the Python script and then save the data in the file, as shown below. More file operations will be discussed in next chapter.

Note: This method is better than previous method, as we can select the outputs which should

be written in the file. For example, Line 44 will not be printed in the file, if the wrong input values are provided during execution of the script.

```
# expansion.py

# find the number of days when radius = 10 cm due to heat-expansion

import sys

if len(sys.argv) != 3: # display error message for missing arguments
    raise SystemExit("usage : ring.py \"metal\" radius")

# sys.argv[0] is the file name
metal = sys.argv[1]

# input is read as string therefore it is converted into float
radius = float(sys.argv[2])

# list of expansion rate for different metal
rate = [0.4, 0.08, 0.05] # [Copper, Gold, Iron]

day = 0

out_file = open("expansion.txt", "w") # open file in write mode

## use any of the below
## below is not in good format
# print("{0}, {1}, {2}".format("day", "expansion", "radius"))

## "> right aligned" "< left aligned"
## 5s : string with width 5
print("{:>5s} {:>10s} {:>7s}".format("day", "expansion", "radius"),
                file = out_file)

## old style
# print("%5s %10s %7s" % ("day", "expansion", "radius"))

while radius < 10:
    # multiply by correct expansion rate
    if metal == "Copper":
        expansion = radius * rate[0]
    elif metal == "Gold":
        expansion = radius * rate[1]
    elif metal == "Iron":
        expansion = radius * rate[2]
    else:
        print("Enter the correct metal")
        break

    # new radius
    radius += expansion
    day += 1 # increment the number of days by one

    ## print the data
    ## 5d : 5 digits
    ## 7.2f : 7 digits with 2 decimal points
    print("{:>5d} {:>10.5f} {:>7.2f}".format(day, expansion, radius),
                file = out_file)

# print the number of days
# print("Number of days =", day)
```

- Now, execute the script and see the content of the file as below,

```
$ python expansion.py "Gold" 8

$ cat expansion.txt
  day  expansion  radius
    1    0.64000    8.64
    2    0.69120    9.33
    3    0.74650   10.08
```

4.6 Alter the printing sequence

We can alter the printing location of the arguments as below,

```
# {0}, {1}, {2} are the position i.e. 0th argument i.e. 'day'
# will go in first position and so on.
print("{0}, {1}, {2}".format("day", "expansion", "radius"))

# same as above but with modified locations i.e. parameter 2 (i.e. day)
# will be printed at first position i.e. {2}
print("{2}, {0}, {1}".format("expansion", "radius", "day"))


# same can be done with following formatting
print("{:>5s} {:>10s} {:>7s}".format("day", "expansion", "radius"))

# same as above, but location is defined
print("{2:>5s} {0:>10s} {1:>7s}".format("expansion", "radius", "day"))
```

4.7 Conclusion

In this chapter, we saw various print statements. Also, we learn the methods by which can save the outputs in the file. In the next chapter, we will learn some more file operations.

Chapter 5

CSV module

5.1 Introduction

In this chapter, we will see some of the features of the Python in-built CSV module. This module can be quite useful for processing the files as shown in this chapter.

5.2 Basic file operations

In this section, we will perform some operations on the file without using the CSV module. For this first create a file 'price.csv' with following contents in it,

```
date,metal,radius,price,quantity
"2016-06-12","Gold",5.5,80.99,1
"2015-07-13","Silver",40.3,5.5,3
"2016-01-21","Iron",9.2,14.29,8
"2014-03-23","Gold",8,120.3,2
"2017-09-11","Copper",4.1,70.25,12
"2011-01-20","Iron",3.25,10.99,3
```

5.2.1 Open and close the files

- Next go to the folder, where the file 'price.csv' is saved and open Python shell there. And run the following commands.

```
>>> f = open("price.csv", 'r')
>>> f # it is a buffered text stream
<_io.TextIOWrapper name='price.csv' mode='r' encoding='UTF-8'>
>>> data = f.read() # read the buffer into data
>>> print(data) # print the data
date,metal,radius,price,quantity
"2016-06-12","Gold",5.5,80.99,1
"2015-07-13","Silver",40.3,5.5,3
"2016-01-21","Iron",9.2,14.29,8
"2014-03-23","Gold",8,120.3,2
"2017-09-11","Copper",4.1,70.25,12
"2011-01-20","Iron",3.25,10.99,3
```

- We can access and print the individual lines as well, as shown below,

```
>>> r = open('price.csv', 'r') # open in read mode
>>> for line in r:
```

(continues on next page)

(continued from previous page)

```
...     print(line)
...
date,metal,radius,price,quantity

"2016-06-12","Gold",5.5,80.99,1

"2015-07-13","Silver",40.3,5.5,3

"2016-01-21","Iron",9.2,14.29,8

"2014-03-23","Gold",8,120.3,2

"2017-09-11","Copper",4.1,70.25,12

"2011-01-20","Iron",3.25,10.99,3
```

- Next close the file. Once the file is closed than we can not perform further operation on buffer.

```
>>> f.close()
>>> r.close()

>>> for line in r: # file is closed, therefore can not be accessed
...     print(line)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.

>>> r
<_io.TextIOWrapper name='price.csv' mode='r' encoding='UTF-8'>
```

5.2.2 with - open statement

In this section, 'with - open' statement is used to read the file.

Note: Do not forget to close the file using close() command. Further, file will be closed automatically, if it is open using 'with' statement, as shown in this section. This method is quite useful when we write the codes in the files, instead of Python-shell.

- In the below code, the file is open using 'with' statement therefore it will be closed as soon as the 'with' statement ends. Therefore buffer will not be accessible outside the 'with' statement. For example in the below code, Line 13 generates the error, as buffer 'w' is outside the 'with' statement.

```
1  >>> with open('price.csv', 'r') as w:
2  ...     data = w.read()
3  ...
4  >>> print(data) # print the data
5  date,metal,radius,price,quantity
6  "2016-06-12","Gold",5.5,80.99,1
7  "2015-07-13","Silver",40.3,5.5,3
8  "2016-01-21","Iron",9.2,14.29,8
9  "2014-03-23","Gold",8,120.3,2
10 "2017-09-11","Copper",4.1,70.25,12
11 "2011-01-20","Iron",3.25,10.99,3
12
13
14
15 >>> for lines in w:  # file is already closed
```

(continues on next page)

(continued from previous page)

```
...     print(lines)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

5.3 Strings operations

We need to perform string-operations to the data for the further processing the data e.g. extracting the lines which contains "Gold" etc. In this section, we will see some of the string operations and the perform these operations on the file 'print.csv'.

```
>>> m = "Hello World"
>>> print(m)
Hello World
>>> m[0] # print first character
'H'
>>> m[0:2] # print first 2 characters
'He'

>>> m[-1] # print last character
'd'
>>> m[-3:-1] # print 2nd and 3rd last (but not the last)
'rl'
>>> m[-3:] # print last 3 characters
'rld'
```

Lets see some more string operations as below. Please read the comments in the codes.

Commands	Description
strip()	remove end line character i.e. n
strip("")	remove "
replace("", '-')	replace " with -
split(",")	make list for data with separator ','

```
>>> f = open('price.csv', 'r')
>>> for line in f:
...     print(line)
...
date,metal,radius,price,quantity

"2016-06-12","Gold",5.5,80.99,1

"2015-07-13","Silver",40.3,5.5,3

"2016-01-21","Iron",9.2,14.29,8

"2014-03-23","Gold",8,120.3,2

"2017-09-11","Copper",4.1,70.25,12

"2011-01-20","Iron",3.25,10.99,3


>>> line # loop store only one value
```

(continues on next page)

(continued from previous page)

```
'"2011-01-20","Iron",3.25,10.99,3\n'

>>> # remove the end line character i.e. \n
>>> line.strip()
'"2011-01-20","Iron",3.25,10.99,3'
>>> line # strip operation does not save automatically
'"2011-01-20","Iron",3.25,10.99,3\n'
>>> line = line.strip() # save the split operation
>>> line
'"2011-01-20","Iron",3.25,10.99,3'

>>> line.replace('"', '-') # replace " with -
'-2011-01-20-,-Iron-,3.25,10.99,3'

>>> # create list and split at comma
>>> columns = line.split(',')
>>> columns
['"2011-01-20"', '"Iron"', '3.25', '10.99', '3']
>>> type(columns)
<class 'list'>

>>> # acess columns
>>> columns[0]
'"2011-01-20"'

>>> # remove " from the data
>>> for i, col in enumerate(columns):
...     columns[i] = col.strip('"')
...
>>> columns
['2011-01-20', 'Iron', '3.25', '10.99', '3']
>>> columns[0]
'2011-01-20'

>>> # all the items are string, therefore multiplaction can not be performed
>>> total_price = columns[3]*columns[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'

# convert string to 'float' and multiple
>>> total_price = float(columns[3])*int(columns[4])
>>> total_price
32.97
```

5.4 Perform calculations

In this section, we will write a Python script which will use all the operations of the previous section, to calculate the total cost of all the items.

```
1  # price.py
2
3  total_price = 0 # for all items in the list
4
5  with open('price.csv', 'r') as f: # open file in read mode
6      header = next(rows) # skip line 1 i.e. header
7      for line in f:
8          line = line.strip() # remove \n
```

(continues on next page)

(continued from previous page)

```
        columns = line.split(',') # split line on ,
        columns[0] = columns[0].strip('"') # remove " for metal
        columns[1] = float(columns[1]) # radius
        columns[2] = float(columns[2]) # price
        columns[3] = int(columns[3]) # quantity
        columns[4] = columns[4].strip('"') # date

        total_price += columns[2] * columns[3]

print("Total price =", total_price)
```

Following is the output of above listing,

```
$ python price.py
Total price = 1328.38
```

Important: Note that, 'next(rows)' is used in Listing 6.1 which return the next row of the reader's iterable object as a list. Since, the first row is extracted and save in the header, therefore it will not be available inside the for loop.

5.5 Problem with current method

Create a file 'price2.csv' with following contents. Note that the contents of this file and the 'price.csv' are the same expect for the date, which is different format.

```
date,metal,radius,price,quantity
"Jun 12, 2016","Gold",5.5,80.99,1
"Jul 13, 2015","Silver",40.3,5.5,3
"Jan 21, 2016","Iron",9.2,14.29,8
"Mar 23, 2014","Gold",8,120.30,2
"Sep 11, 2017","Copper",4.1,70.25,12
"Jan 20, 2011","Iron",3.25,10.99,3
```

Now read the file 'price2.csv' in 'price.py' as below,

```
# price.py

total_price = 0 # for all items in the list

with open('price2.csv', 'r') as f: # open file in read mode
    header = next(rows) # skip line 1 i.e. header
    for line in f:
        line = line.strip() # remove \n
        columns = line.split(',') # split line on ,
        columns[0] = columns[0].strip('"') # date
        columns[1] = columns[1].strip('"') # remove " for metal
        columns[2] = float(columns[2]) # radius
        columns[3] = float(columns[3]) # price
        columns[4] = int(columns[4]) # quantity

        total_price += columns[3] * columns[4]

print("Total price = %10.2f" % total_price)
```

Next execute the code and following error will be generated.

```
$ python -i price.py
Traceback (most recent call last):
  File "price.py", line 12, in <module>
    columns[2] = float(columns[2]) # radius
ValueError: could not convert string to float: '"Gold"'
```

This error is generated because the date has one comma, because of which an additional column is added to the list, as shown below,

```
>>> columns
['Jun 12', ' 2016', '"Gold"', '5.5', '80.99', '1']
```

Note: One way to remove this problem is to redesign the code according to new date format. Note that, the dates are in standard formats in both the cases. Therefore we should think like this, "This is not a new problem as everything is in standard format, therefore there must be an standard way to solve this problem". And look for the standard library or third party packages to solve the problem. Currently, we can solve this problem using CSV module, or we can use Pandas-library which is quite powerful to solve the problems in data-processing. Click here to learn the Pandas.

5.6 CSV module

In this section, we will see two functionalities of the CSV module i.e. 'csv.reader' and 'csv.DictReader'.

5.6.1 csv.reader

- Before modifying the Python script, let us see the functionality of the CSV module. Note that, in the below outputs, the stripping and splitting operations are performed by the CSV module itself.

```
>>> import csv
>>> f = open('price2.csv', 'r')
>>> rows = csv.reader(f) # read the file using csv
>>> for row in rows:
...     print(row)
...
['date', 'metal', 'radius', 'price', 'quantity']
['Jun 12, 2016', 'Gold', '5.5', '80.99', '1']
['Jul 13, 2015', 'Silver', '40.3', '5.5', '3']
['Jan 21, 2016', 'Iron', '9.2', '14.29', '8']
['Mar 23, 2014', 'Gold', '8', '120.30', '2']
['Sep 11, 2017', 'Copper', '4.1', '70.25', '12']
['Jan 20, 2011', 'Iron', '3.25', '10.99', '3']
```

- Following is the Python script which can perform the calculation on both the files i.e. 'price.csv' and 'price2.csv'.

Listing 5.1: price calculation using csv module

```
# price.py

import csv

total_price = 0 # for all items in the list

with open('price2.csv', 'r') as f: # open file in read mode
    rows = csv.reader(f)
    header = next(rows) # skip line 1 i.e. header
```

(continues on next page)

(continued from previous page)

```
    for row in rows:
        row[3] = float(row[3]) # price
        row[4] = int(row[4]) # quantity

        total_price += row[3] * row[4]

print("Total price = %10.2f" % total_price)
```

- Run the above script and we will get the following output,

```
$ python price.py
Total price =    1328.38
```

Important: Note that, when we use standard library then lots of task are reduced e.g. here we need not to perform any cleaning operation i.e. removing double-quotes and commas etc. Also, the code is shorter and cleaner when we used the CSV module.

5.6.2 csv.DictReader

The '.DictReader' option is same as the '.reader', but it maps the information into the dictionary, which enhances the data processing capabilities.

Note:

- '.reader' returns a reader-object which iterates over the line of the csv file.
- '.DictReader' is similar to '.reader' but maps the information in the dictionary.
- In the below example, Python data structure are used, e.g. List and Set etc., which are discussed in next chapter.
- Further, we will some more examples of '.DictReader' after learning the 'data structure' and 'functions'.

```
>>> import csv
>>> f = list(csv.DictReader(open('price.csv'))) # read DictReader in list
>>> f[0]  # first item in the list

OrderedDict([('date', '2016-06-12'), ('metal', 'Gold'), ('radius', '5.5'),
    ('price', '80.99'), ('quantity', '1')])

>>> [row['metal'] for row in f] # display all values i.e. List
['Gold', 'Silver', 'Iron', 'Gold', 'Copper', 'Iron']

>>> {row['metal'] for row in f} # display unique values i.e. Set
{'Silver', 'Copper', 'Iron', 'Gold'}

>>> g = [row for row in f if row['metal'] == 'Gold'] # read Gold entries
>>> len(g)  # total gold entries
2

>>> for item in g:  # print radius, price and quantity
...     print(item['radius'], item['price'], item['quantity'])
...
5.5 80.99 1
8 120.3 2
```

5.7 Conclusion

In this chapter, we saw the various ways to read the files. Also, we learn the usage of CSV module. Lastly, we saw some of the data structure available in Python, which will be discussed in details after the next chapter. In the next chapter, we will discussed the “functions” and “error handling”.

Chapter 6

Functions

6.1 docstring

The content between the ''' is known as 'docstring', which are displayed by the 'help' function as shown below. Press 'q' to exit from help-screen.

```
>>> def add2Num(x, y):
...     ''' add two numbers : x, y '''
...     print(x+y)
...
>>> add2Num(2, 3)
5
>>> help(add2Num)
Help on function add2Num in module __main__:

add2Num(x, y)
    add two numbers : x, y
```

6.2 types of docstring

There are two standard format of creating the docstrings, i.e. Numpy style and Goole style, which are supported by Sphinx-documentation for generating the auto-documentation of the project.

6.3 Convert previous code into function

6.3.1 Conversion

Now, we will convert the code in Listing 5.1 into function. Conversion process is quite simple, as shown in Listing 6.1, where function with docstring is defined at Lines 5-6. Ther previous code is indented and finally a return statement is added in Line 20. Lines 22-24 calls the function and print the output. Lastly, Lines 28-29 are the standard boilerplate to set the function 'main' as the entry point.

Listing 6.1: price calculation using function

```
1 # price.py
2
3 import csv
4
5 def ring_cost(filename):
```

(continues on next page)

(continued from previous page)

```
    ''' calculate the total cost '''

    total_price = 0 # for all items in the list

    with open(filename, 'r') as f: # open file in read mode
        rows = csv.reader(f)
        header = next(rows) # skip line 1 i.e. header
        for row in rows:
            row[3] = float(row[3]) # price
            row[4] = int(row[4]) # quantity

            total_price += row[3] * row[4]

    # print("Total price = %10.2f" % total_price)
    return total_price  # return total_price

def main():
    total = ring_cost('price.csv')  # function call
    print("Total price = %10.2f" % total) # print value

# standard boilerplate
# main is the starting function
if __name__ == '__main__':
    main()
```

```
$ python price.py
Total price =    1328.38
```

6.3.2 Debugging

In Listing 5.1, we have to manually change the file name in the 'price.py' file; whereas in Listing 6.1, we can pass the filename as the parameter in the Section 3.4.1, as shown below,

```
$ python -i price.py
Total price =    1328.38
>>>
>>> ring_cost('price.csv')
1328.3799999999999
>>> ring_cost('price2.csv')
1328.3799999999999
>>>
```

Note: In the above command, 'python -i price.py', the main() function is called. And after entering the Python shell, we can call the function directly i.e. ring_cost('price.csv'), and the corresponding 'return value', i.e. 1328.3799999999999, will be printed in the shell.

6.4 glob module

'glob' module can be used to select the files for further processing, as shown in this section. To understand it, first create some files as below,

```
$ touch data1.txt data2.txt data3.txt data_1.txt data_2.txt data_3.txt data1.csv data2.csv data3.csv
```

Next, open the Python shell and see the following function of 'glob' module,

```
>>> glob.glob('*.csv') # find all csv files
['data3.csv', 'data2.csv', 'data1.csv']

>>> glob.glob('data*.txt') # select all txt file which starts with 'data'
['data_3.txt', 'data_2.txt', 'data2.txt', 'data_1.txt', 'data3.txt', 'data1.txt']

>>> # select txt files which have one character between 'data' and '.txt'
>>> glob.glob('data?.txt')
['data2.txt', 'data3.txt', 'data1.txt']

>>> glob.glob('data[0-2]*.csv') # select csv file with numbers 0,1,2 after 'data'
['data2.csv', 'data1.csv']
```

Note:

- The 'glob' module returns the 'list'.
- The list is in unordered form.

6.4.1 Price calculation on files using 'glob'

Now, we will use the glob module to perform 'price calculation' on files using 'glob' module. First open the Python shell without using '-i' operation.

```
$ python
```

Since we did not use the '-i' operation to open the shell, therefore we need to import the function 'ring_cost' in the shell, as shown below,

```
>>> import glob
>>> from price import ring_cost
>>>
>>> files = glob.glob('pri*.csv')
>>> for file in files:
...     print(file, ring_cost(file))
...
price.csv 1328.3799999999999
price2.csv 1328.3799999999999
```

Warning: Note that we need to restart python shell every time, we made some changes in the code. This is required as the 'import' statement loads all the data at the first time; and when we re-import the modules then it is fetched from the cache.

Note that, the 'glob' returns a list, therefore we can extend the list using various listing operation. This can be useful when we have files names with different names, but required same operations e.g. we want to perform price calculation on another set of items which has same columns as price.csv file. List can be modified as below,

```
>>> files = glob.glob('pri*.csv')
>>> files2 = glob.glob('pri*.csv')
>>> files.extend(files2)  # extend list
>>> files
['price.csv', 'price2.csv', 'price.csv', 'price2.csv']
>>> files.append('price2.csv')  # append data
>>> files
['price.csv', 'price2.csv', 'price.csv', 'price2.csv', 'price2.csv']
```

Chapter 7

Data types and objects

7.1 Introduction

In this chapter, we will see various data types available in the Python. Also, we will use these data types for processing the data (also known as data mining).

Everything in Python is the object e.g. builtin types numbers, strings, list, set and dictionary etc. are objects. Further, user define objects can be created using 'classes'. This chapter describes the working of Python object model along with builtin data types.

Note: This chapter contains lots of details about the data objects and data types. We can skip this chapter at this moment, except Section 7.5, and reference back to it, whenever required.

7.2 Identity and type

Each object has an identity in the form of integer and a type. The identity corresponds to the location of the object in the memory.

```
>>> x = 2
>>> id(x)  # id is unique memory address
3077862864
>>> type(x)
<class 'int'>
>>>

>>> # type of class is 'type'
>>> class Circle(object):
...     pass
...
>>> id(Circle)
3070212412
>>> type(Circle)
<class 'type'>
>>>

>>> # type of builtin-type is 'type'
>>> type(int)
<class 'type'>
>>>
```

Note: The type of classes and builtin types e.g. int and float are type as shown in above code.

7.2.1 'is' operator

- The 'is' operator is used to compare the identity of two objects,

```
>>> a = [1, 2, 3]
>>> b = a # b is new name for a

>>> # a and b have same id
>>> id(a)
3070184940
>>> id(b)
3070184940

>>> # since a and b are same, hence if modify b, a also changes
>>> b.append(3)
>>> a
[1, 2, 3, 3]
>>>

>>> # 'is' returns true if 'id' is same
>>> a is b
True
>>> b is a
True
>>>
```

7.2.2 Comparing objects

Objects can be compared based on values, id and types as shown below,

```
>>> def compare(a, b):
...     if a is b:
...             print(" objects have same id")
...     if a == b:
...             print(" values of objects are same")
...     if type(a) is type(b):
...             print(" types of objects are same")
...

>>> x=[1, 2, 3]
>>> y=[1, 2, 3]
>>> id(x)
3070183308
>>> id(y)
3070185004
>>> compare(x, y)
 values of objects are same
 types of objects are same


>>> compare(2, 2)
objects have same id
values of objects are same
types of objects are same
```

(continues on next page)

(continued from previous page)

```
>>> compare(2, 3)
types of objects are same

>>> x = 3
>>> y = 3
>>> compare(x, y)
 objects have same id
 values of objects are same
 types of objects are same
>>> id(x)
3077862880
>>> id(y)
3077862880
```

7.2.3 isinstance

Note: The type of an object is **itself an object**, which is known as object's class. This object (i.e. object's class) is unique for a given type, hence can be compared using 'is' opeartor as shown below,

```
>>> x = [1, 2, 3]
>>> y = [2, 3]
>>> if type(x) is list:
...     x.append(y)
...
>>> x
[1, 2, 3, [2, 3]]
>>>
```

- Above operation can be performed using **isinstance(object, type)** as shown below,

```
>>> x = [1, 2, 3]
>>> y = [2, 3]
>>> if isinstance(x, list):
...     x.append(y)
...
>>> x
[1, 2, 3, [2, 3]]
>>>
```

7.3 Reference count and garbage collection

All the objects are reference counted. This reference count is increased whenever a new name is assigned to object or placed in a container e.g. list, tuple or dictionary etc.

7.3.1 getrefcount

getrefcount can be used to see the reference count of an object.I

```
>>> r = 20
>>> import sys
>>> sys.getrefcount(r)
15
>>> o = r
```

(continues on next page)

(continued from previous page)

```
>>> sys.getrefcount(r)
16
>>> c = []
>>> c.append(r)
>>> c
[20]
>>> sys.getrefcount(r)
17
>>>
```

7.3.2 Garbage collection

The 'del' command decrease the value of reference count; when this value reaches to zero, the object is garbage collected as shown below,

```
>>> del c
>>> sys.getrefcount(r)
16

>>> del o
>>> sys.getrefcount(r)
15

>>> del r
```

7.3.3 Shallow and deep copy

Note: In Section 7.2.1, we assigned b = a, which created a reference for a as b. However, this behaviour is quite different for mutuable objects e.g. list and dictionary. There are two types of reference for these cases i.e. shallow copy and deep copy.

Shallow copy

- In shallow reference, inner list or dictionary shared the data between two references as shown below. Note that, a and b are not same i.e. 'a is b' results as False, but still change in b results in change in a.

```
>>> a = [1, 2, [10, 20]]
>>> b = list(a)  # create shallow copy i.e. [10, 20] are still shared
>>> b
[1, 2, [10, 20]]

>>> a is b
False

>>> b.append(30)
>>> a   # no change in a
[1, 2, [10, 20]]

>>> b[2][0] = 100
>>> a  # value of a is changed
[1, 2, [100, 20]]
>>>
```

Deep copy

Above problem can be solved using 'deep copy'. Deep copy creates a new object and recursively copies all the objects it contains.

```
>>> import copy
>>> a = [1, 2, [3, 4]]

>>> b = copy.deepcopy(a)

>>> a is b
False

>>> b[2][0]=100
>>> b
[1, 2, [100, 4]]
>>> a  # a is not changed
[1, 2, [3, 4]]
>>>
```

7.4 First class object

All objects in python are said to be 'first class' i.e. all objects that can be named by an identifier have equal status. In below code, 'ty' is assigned the object 'str'; now 'ty' can be used in place of str as shown below,

```
>>> ty=str
>>> ty = int
>>> ty('10')
10
>>>
```

This feature is quite useful for writing compact and flexible code. In below code, list comprehension is used to change the data type of 'stock',

```
>>> # here, stock is string, we want to split it and
>>> # convert the data in to correct format i.e. string, int and float
>>> stock = "GOOG, 100, 20.3"
>>> field_types = [str, int, float]
>>> split_stock = stock.split(',')   # split data and get a list
>>> split_stock
['GOOG', ' 100', ' 20.3']

>>> # change the format
>>> stock_format = [ty(val) for ty, val in zip(field_types, split_stock)]
>>> stock_format   # format = [str, int, float]
['GOOG', 100, 20.3]
```

7.5 Builtin types for representing data

There are various types of data types, which can be divided as shown in Table 7.1.

Table 7.1: Builtin datatypes

Category	Type name	Description
None	NoneType	x = None
Numbers	int	integer
	float	floating point
	complex	complex number
	boolean	True or False
Sequences	str	character string
	list	list
	tuple	tuple
Mapping	dict	Dictionary
Sets	set	Mutable set
	frozenset	Immutable set

7.5.1 None

None object is usually used with the function for setting default value for the kwargs e.g. def add(a=None, b=None) etc. Further, if None is return by a function than it is considered as 'False' in Boolean expressions,

```
>>> x = None
>>> type(x)
<class 'NoneType'>
>>>

>>> # input arguments values as None
>>> def add2Num(a=None, b=None):
...     c = a + b
...     print(c)
...     return None
...
>>> s = add2Num(3, 2) # None is return
5

>>> type(s)
<class 'NoneType'>

>>> # None is considered as False
>>> if s:
...     print("s has some value")
... else:
...     print("s is None")
...
s is None
>>>
```

- isinstance can not be used with None type,

```
>>> isinstance(x, None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() arg 2 must be a type or tuple of types
```

7.5.2 Numbers

- Complex numbers are represented as a pair of two **floating-point** number. Real part and complex part of complex number can be extracted as below,

```
>>> c = 3 + 3j
>>> c.real
3.0
>>> c.imag
3.0
>>> c.conjugate()
(3-3j)
>>>
```

7.5.3 Sequences

Sequences are the set objects which are indexed by non-negative integers. Python has three types of sequences i..e. string, list and tuple. Strings and tuples are immutable objects, whereas list is mutable object.

Table 7.2 shows the operations which are applicable to all the sequences,

Table 7.2: Common operations to all sequences

Operation	Description	example s = [5, 10, 15, 20, 25]
s[i]	returns element i	s[1] # 10
s[i:j]	returns a slice	s[0:3] # [5, 10, 15]
s[i:j:stride]	returns a extended slice	s[0:3:2] # [5, 15]
len(s)	Number of elements	len(s) # 5
min(s)	mininum value in s	min(s) # 5
max(s)	maximum value in s	max(s) # 25
sum(s)	sum of all elements	sum(s) # 75
sum(s, [, initial])	sum of elements + initial value	sum(s, 3) # 78
all(s)	return True, if all true in s	
any(s)	return Trun, if any true in s	

'all' and 'any' example

```
>>> x = (1, 2, 3, 4, 5) # tuple

>>> x[0:3]
(1, 2, 3)

>>>  # True, as all items in x are greater than zero
>>> all(item>0 for item in x)
True

>>>  # False, as all items in x are not greater than 4
>>> all(item>4 for item in x)
False

>>>  # True, as some items in x are greater than 4
>>> any(item>4 for item in x)
True
>>>
```

- Some string operations are shown below,

```
>>> s = 'Meher Krishna Patel'
>>> s[:5]   # 0 to 4
'Meher'

>>> s[:-5]  # 0 to sixth last
```

(continues on next page)

(continued from previous page)

```
'Meher Krishna '

>>> s[-5:]  # fifth last to end
'Patel'
>>>
```

- Table 7.3 shows the operations which are applicable to mutable sequences only, i.e. string and list,

Table 7.3: Operations for mutuable sequences only

item	description
s[i] = v	item assignment, v = value
s[i, j] = t	slice assignment, t = tuple/list
s[i:j:stride] = t	extended slice assignment
del s[i]	item deletion
del s[i:j]	slice deletion
del s[i:j:stride]	extended slice deletion

- Some of the operations of Table 7.3 are shown below,

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> x[0] = 20  # value assignment
>>> x
[20, 2, 3, 4, 5, 6, 7, 8]
>>>
>>> x[2:5] = -10, -20, -30  # slice assignment
>>> x
[20, 2, -10, -20, -30, 6, 7, 8]
>>>
>>> x[3:6] = [-100, -200, -300]
>>> x
[20, 2, -10, -100, -200, -300, 7, 8]
>>>
>>> x[0:5:3] = 0, 0   # extended slice assignment
>>> x
[0, 2, -10, 0, -200, -300, 7, 8]
>>>
>>> del x[3:5]   # slice deletion
>>> x
[0, 2, -10, -300, 7, 8]
>>>
```

7.5.4 List

List is the mutable sequences, which is often used to store the data of same type. It can be seen as the columns of a spreadsheet.

Note: list(s) command converts any iterable to a list. If iterable is already a list, then list(s) command will create a new list with shallow copy of 's' as discussed in Section 7.3.3.

Table 7.4 shows various methods supported by list object,

Table 7.4: List methods

Method	Description
list(s)	convert s to list
s.append(x)	append element x, to the end of s
s.extend(x)	append new list x, to the end of s
s.count(x)	count occurances of x in s
s.index(x, [, start [, stop]])	return the smallest i where s[i] == x. start and stop optionally specify the starting and ending for the search
s.insert(i, x)	insert x at index i
s.pop([i])	pops the element index i. If 'i' is ommited then last value popped out
s.remove(x)	search for first occurrence of x and remove it
s.reverse()	reverse the order of list
s.sort([key [, reverse])	Sorts item fo s. 'key' and 'reverse' should be provided as keywords

- Some of the examples of commands in Table 7.4 are listed below,

```
>>> s = [1, 2, 3]
>>> s.append(4)
>>> s
[1, 2, 3, 4]
>>> s.append([3, 4])
>>> s
[1, 2, 3, 4, [3, 4]]
>>>
>>> x = [1, 2, 3]
>>> x.extend([1, 5])
>>> x
[1, 2, 3, 1, 5]
>>> x.count(1)
2
>>> x.index(3)
2
>>> x.insert(2, -10)
>>> x
[1, 2, -10, 3, 1, 5]
>>>
>>> x.pop()
5
>>> x
[1, 2, -10, 3, 1]
>>> x.pop(2)
-10
>>> x
[1, 2, 3, 1]
>>>
>>> x.remove(1)
>>> x
[2, 3, 1]
>>>
>>> s.reverse()
>>> s
[4, 3, 2]
>>>
```

Note: 's.sort()' method sorts and modifies the original list; whereas 'sorted()' option sorts the contents of the list, but does not modify the original list, therefore we need to save it manuall. Further, if we try to save the outcome of s.sort() in some other list, it will not work as shown below

```
>>> name_age = [ ['Tom', 20], ['Jerry', 15], ['Pluto', 25] ]
>>> name_age
[['Tom', 20], ['Jerry', 15], ['Pluto', 25]]
>>> name_age.sort(key=lambda name: name[0])   # sort by name
>>> name_age
[['Jerry', 15], ['Pluto', 25], ['Tom', 20]]

>>> name_age.sort(key=lambda age: age[1])     # sort by age
>>> name_age
[['Jerry', 15], ['Tom', 20], ['Pluto', 25]]
>>>

>>> x = []
>>> x = name_age.sort(key=lambda age: age[1])  # can not save the output
>>> x  # nothing is saved in x

>>> # use sorted() to save outputs to another list
>>> y = sorted(name_age, key=lambda age: age[1])   # sorted() : by age
>>> y
[['Jerry', 15], ['Tom', 20], ['Pluto', 25]]

>>> y = sorted(name_age, key=lambda name: name[1])    # sorted() : by name
>>> y
[['Jerry', 15], ['Tom', 20], ['Pluto', 25]]
```

7.5.5 Strings

Various strings methods are shown in Table 7.5. As oppose to list methods, string methods do not modify the underlying string data. In the other words, the string methods return the new string which can be saved in new string object.

Table 7.5: String methods

Method	Description
s.capitalize()	capitalize the first letter
s.center(width, [,pad])	centers the string in a field of length width. pad is a padding character
s.count(sub, [, start [, end]])	counts occurrence of the specified substring sub
s.endswith(suffix, [, start [, end]])	checks for the end for a suffix
s.expandtabs([tabsize])	replace tabs with spaces
s.find(sub [, start [, end]]) s.rfind(sub [, start [, end]])	find the first occurrence of substring sub find last occurrence of substring sub
s.format(* args, ** kwargs)	format string
s.index(sub, [, start [, end]]) s.rindex(sub, [, start [, end]])	find the first occurrence of sub find the last occurrence of sub
s.isalnum()	checks whether all characters are alphanumerics
s.isalpha()	checks whether all characters are alphabets
s.isdigit()	checks whether all characters are digits
s.islower()	checks whether all characters are lowercase
s.isspace()	checks whether all characters are spaces
s.istitle()	checks whether all characters are titlecased
s.isupper()	checks whether all characters are uppercase
s.join(t)	joins string in sequence t with s as separator
s.ljust(width [, fill]) s.rjust(width [, fill])	left or right align s in a string of size 'width'
s.lower() s.upper()	change string to lower or upper case
s.lstrip([chrs])	remove leading white space or [chrs] if provided
s.partition(sep) s.partition(sep)	partitions a string based on a seperator string 'sep'. It returns a tuple (head, sep, tail) or s(s, '', '') if seperator is not provided partitions a string but search from the last.
s.replace(old, new [, maxreplace])	replace a substring
s.split([sep [, maxsplit]]) s.rsplit([sep, [,maxsplit]])	splits a string using 'sep' as delimiter. maxsplit is the maximum number of split. r is used for checking from the end
s.splitlines([keepends])	splits the string into a list of line. If keepends is 1, then trailing newlines are preserved
s.startswith(prefix, [,start [,end]])	checks whether string starts with prefix
s.strip([chars]) s.rstrip([chars])	removes leading and trailing white spaces or chars if provided
s.swapcase()	changes the case of string
s.title()	return title-case version of string
s.zfill(width)	pads a string with zeros on the left up to specified width

- Some of the examples of commands in Table 7.5 are listed below,

```
>>> s = "meher krishna patel"
>>> s.capitalize()
'Meher krishna patel'
>>>

>>> s.center(30)
'     meher krishna patel      '
>>> s.center(50)
'               meher krishna patel                '
```

(continues on next page)

(continued from previous page)

```
>>>
>>> s.center(30, '#')
'#####meher krishna patel######'
>>>

>>> s.count('e')
3
>>> s.count('eh')
1
>>>

>>> s.endswith('Patel')
False
>>> s.endswith('Patel ')  # space added at the end
False
>>>
>>> s.endswith('er', 0, 5)  # Meher, check from 0 to 4
True
>>>

>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {0}, {1}'.format('a', 'b', 'c')
'c, a, b'
>>> '{name}, {age}'.format(name='Meher', age=30)   # keywords
'Meher, 30'

>>> s.index('e')
1
>>> s.index('e', 4)
17

>>> t = ' '   # space seperator
>>> t.join(s)
'm e h e r   k r i s h n a   p a t e l'
>>>

>>> name = '    ## Meher'
>>> name.lstrip()
'## Meher'
>>> name.lstrip().lstrip('##')
' Meher'
>>>

>>> s.partition(' ')
('meher', ' ', 'krishna patel')
>>> s.partition('krishna')
('meher ', 'krishna', ' patel')
>>>

>>> s.replace('e', 'E')
'mEhEr krishna patEl'
>>> s.replace('e', 'E', 2)
'mEhEr krishna patel'
>>>

>>> s.split(' ')
['meher', 'krishna', 'patel']
>>>

>>> l = 'My name is Meher.\nWhat is your name?'
```

(continues on next page)

(continued from previous page)

```
>>> print(l)
My name is Meher.
What is your name?
>>> l.splitlines()
['My name is Meher.', 'What is your name?']
>>>
>>> l.splitlines(1)
['My name is Meher.\n', 'What is your name?']
>>>

>>> n = '   #meher krishna patel#  '
>>> n.strip()
'#meher krishna patel#'
>>> n.strip().strip('#')
'meher krishna patel'

>>> s.zfill(30)
'00000000000meher krishna patel'
>>>
```

7.5.6 Mapping types

A mapping object represents an arbitrary collection of objects that are indexed by another collection of arbitrary key values. Unlike sequences, a mapping object can be indexed by numbers, strings and other objects. Further, the mappings are mutable.

Dictionaries are the only builtin mapping types in python. Table 7.6 shows the list of various dictionary operations,

Table 7.6: Methods and operations on dictionaries,

Item	Description
len(m)	number of item in m
m[k]	returns value of key k
del m[k]	delete key k
k in m	return True if key k exist
m.clear()	remove all item from m
m.copy()	make a copy of m
m.fromkeys(s, [,val])	create a new dictionary with keys from sequnece s. 'None' or val (if provided) is filled as values to all keys
m.get[k, [,msg]]	returns m[k]; if not found return msg.
m.items() m.keys() m.values()	returns items, keys or values These are return as iterator
m.pop(k [,msg])	pops m[k] if found; otherwise shows msg if provided, else keyerror
m.popitem()	remove one key-value at random
m.setdefault(k [,v])	same as m.get() but set the key with value None/v, if not found (instead of keyerror)
m.update(b)	add all objects of b to m

- Some of the examples of commands in Table 7.6 are listed below,

```
>>> m = { 'name' : 'AA',
...       'shares' : 100,
...       'price' : 300.2
... }
>>>
>>> len(m)
3

>>> m['name']
```

(continues on next page)

(continued from previous page)

```
'AA'

>>> m['name'] = 'GOOG'
>>> m['name']
'GOOG'

>>> del m['shares']
>>> len(m)
2

>>> 'shares' in m
False
>>> 'name' in m
True

>>> m.clear()
>>> m
{}

>>> m = { 'name' : 'Tiger', 'age' : 14 }
>>> m
{'name': 'Tiger', 'age': 14}
>>> c = m.copy()
>>> c
{'name': 'Tiger', 'age': 14}

>>> stock = ['name', 'shares', 'price']
>>> n={}
>>> n.fromkeys(stock)
{'name': None, 'shares': None, 'price': None}
>>> n   # note that results are not stored in n
{}

>>> o={}
>>> o = o.fromkeys(stock, 0)
>>> o
{'name': 0, 'shares': 0, 'price': 0}

>>> o.get('name')
 0
 >>> o.get('rise', 3)
 3
 >>> o.get('name', 3)
 0

>>> o.pop('price')
0
>>> o
{'name': 0, 'shares': 0}
>>> o.pop('price', 'not found')
'not found'

>>> m
{'name': 'AA', 'shares': 100}
>>> m.setdefault('price', 200.3)
200.3
>>> m
{'name': 'AA', 'shares': 100, 'price': 200.3}
>>> m.setdefault('shares', 80)
100
>>> m
```

(continues on next page)

(continued from previous page)

```
{'name': 'AA', 'shares': 100, 'price': 200.3}
>>> m.setdefault('name')
'AA'

>>> b= {'rise': 10, 'fall':0}
>>> m.update(b)
>>> m
{'rise': 10, 'name': 'AA', 'fall': 0, 'shares': 100, 'price': 200.3}
>>>
```

Note: m.items(), m.keys() and m.values() returns the result as iterator, which can be changed into list using 'list()' method,

```
>>> m = {'name': 'AA', 'shares':100, 'value':200.2}
 >>> m
 {'name': 'AA', 'shares': 100, 'value': 200.2}

 >>> i = m.items()
 >>> i
 dict_items([('name', 'AA'), ('shares', 100), ('value', 200.2)])

 >>> i = list(i)
 >>> i
 [('name', 'AA'), ('shares', 100), ('value', 200.2)]

 >>> v = list(m.values())
 >>> v
 ['AA', 100, 200.2]
 >>>
```

7.5.7 Set types

- Set is an unordered collection of unique items.
- Unlike sequences, set does not provide indexing.
- Unlike dictionary, set does not provide key associated with values.
- The items placed in the set must be immutable e.g. list can not be saved in set as shown below,

```
>>> s = set()
>>> type(s)
<class 'set'>

>>> s.add([1, 2, 3])    #  mutable objects e.g. list can not be saved in set
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

>>> s.add((1, 2, 3))   # immutable objects e.g.tuple can be saved in set
>>> s
{(1, 2, 3)}

>>> s.add(5)
>>> s
{5, (1, 2, 3)}
>>>
```

- Sets are of two types i.e. 'set' and 'frozenset'. Above codes are the example of set, which is mutable in nature; whereas 'frozenset' is immutable i.e. add operation can not be performed on frozenset.

Table 7.7 shows various operations that can be performed on both sets and frozensets,

Table 7.7: Methods and operations for Set types

Item	Description
len(s)	total number of items in s
s.copy()	makes a copy of s
s.difference(t)	return items in s but not in t. The 't' can be any iterator e.g. list, tuple or set
s.intersection(t)	returns common in both s and t
s.isdisjoint(t)	returns 'True' if nothingis common in s and t
s.isubset(t)	returns 'True' if s is subset of t
s.issuperset(t)	returns 'True' is s is superset of t
s.symmetric_difference(t)	returns all the items from s and t, which are not common in s and t
s.union(t)	returns all items from s and t

- Some of the operations of Table 7.7 are shown below,

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

>>> s = set(a) # change 'a' to set
>>> s  # duplicate entries are removed
{1, 2, 3, 4, 5}

>>> t = [1, 2, 3]

>>> s.difference(t)
{4, 5}

>>> t = set(t)
>>> s.difference(t)
{4, 5}
>>>
```

- Mutable sets provides some additional methods as shown in Table 7.8. Operation on this table will update the set after execution of the commands.

Table 7.8: Methods for mutable set type

Item	Description
s.add(item)	add item to s
s.clear()	remove all items from s
s.difference_update(t)	remvoes all items from s, which are in t
s.discard(item)	remove item form s. No error if item not present
s.intersection_update	computes the intersetion of s and t
s.pop()	remove an element from s
s.remove(item)	remove item from s. If not found, raise keyerror
s.symmetric_difference_update(t)	save uncommon entries in s and t to s
s.update(t)	add all entries of iterator 't' to s

- Some of the operations of Table 7.8 are shown below,

```
>>> s = {1, 2, 3, 4, 5}
>>> t = [1, 2, 3, 11, 12]

>>> s.add(9)
>>> s
{1, 2, 3, 4, 5, 9}

>>> s.symmetric_difference_update(t)
>>> s
```

(continues on next page)

(continued from previous page)

```
{4, 5, 9, 11, 12}
>>>
```

7.6 Builtin types for representing program structure

Everything in python is the object including functions, classes and modules. Therefore functions, classes and modules can be manipulated as data. Table 7.9 shows various types that are used to represent various element of a program itself,

Table 7.9: Builtin types for program structure

Type category	Type name	Description
Callable	types.BuiltinFunctionType	builtin function or method
	type	type of builtin types and classes
	object	Ancestor of all types and classes
	types.FunctionType	user defined funtions
	types.MethodType	class method
Modules	types.ModuleType	Module
Classes	object	Ancestor of all types and classes
Types	type	Type of builtin types and classes

Note: In Table 7.9, 'object' and 'type' are appeared twices because 'classes' and 'types' are callable as function.

7.6.1 Callable types

Callable types represent objects which can be called as function e.g. builtin functions, user-define functions, instance method and classes etc.

7.6.1.1 User-defined functions

User-defined functions are callable functions that are created at module level by using 'def' statement or with 'lambda' operator, as shown below,

```
>>> def add2Num(x, y):
...     return(x+y)
...
>>> diff2Num = lambda x, y: x-y
>>>
>>> add2Num(3, 2)
5
>>> diff2Num(3, 2)
1
>>>
```

- Various attributes of user-defined functions are shown in Table 7.10,

Table 7.10: Attributes for user-defined function 'f'

Attributes	Description
f.__doc__	documentation string
f.__name__	function name
f.__dict__	dictionary containing function attributes
f.__code__	byte-compiled code
f.__defaults__	tuple containing default arguments
f.__globals__	dictionary defining the global namespace
f.__closure__	tuple containing data related to nested scope

- Some of the operations of Table 7.10 are shown below,

```
>>> def mathEx(a, b):
...     """ calculate (a+b)*x """
...     global x
...     c = (a + b) * x
...     return c
...

>>> mathEx(2, 4)
18

>>> mathEx.__doc__
' calculate (a+b)*x '

>>> mathEx.__globals__
{'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__builtins__': <module 'builtins' (built-in)>,
'__spec__': None, '__package__': None,
'__name__': '__main__',
'x': 3, 'mathEx': <function mathEx at 0xb70577c4>,
'__doc__': None}
>>>
```

7.6.1.2 Methods

Methods are the functions that are defined inside the class. There are three types of methods i.e. instance method, class method and static method. Following is an example of all three methods,

```
>>> class Spam():
...     num = 10    # class variable
...     def __init__(self, num = 3):
...             self.num = num
...
...     def imethod(self):  # print instance variable
...             print("imethod ", self.num)
...
...     @classmethod
...     def cmethod(cls):  # print class variable
...             print("cmethod ", cls.num)
...
...     @staticmethod
...     def smethod(num):  # print variable defined in the file
...             print("smethod ", num)


>>> s = Spam()

>>> s.imethod()
```

(continues on next page)

(continued from previous page)

```
imethod  3

>>> s.cmethod()
cmethod  10

>>> num = -40
>>> s.smethod()
smethod  -40
```

- Note that, calling a function a two step process as shown below.

```
>>> i = s.imethod    # i is method
>>> type(i)
<class 'method'>

>>> i()              # make function call using ()
imethod  3

>>> iu = Spam.imethod   # function (not method)
>>> type(iu)
<class 'function'>

>>> iu(s)         # instance of the class must be passed in function
imethod  3

>>> type(i)             # instance-method is 'method
<class 'method'>
>>> type(c)             # classmethod is 'method'
<class 'method'>
>>> type(l)             # staticmethod is 'function'
<class 'function'>

>>> c = s.cmethod
>>> c()
cmethod  10
>>> l = s.smethod
>>> l()
smethod  -40
>>> cu = Spam.cmethod
>>> cu()
cmethod  10
>>> lu = Spam.smethod
>>> lu()
smethod  -40
```

Note: Method (s.imethod) is a callable object that wraps both a function and the associated instance. When we call the method, the instance is passed to the method as the first parameter (self); but does not invoked teh function call operator. Then, () can be used to invoke the function call as shown in above example; whereas, the function (Spam.imethod) wraps only method function (not the instance), therefore instance need to passed explicitly.

Table 7.11 shows the attributes available for the method-objects,

Table 7.11: Attributes for the methods

Attributes	Description
m.__doc__	documentation string
m.__name__	method name
m.__class__	name of the class where method is defined
m.__func__	function object implementing the method
m.__self__	instance associated with the method (None if unbound

7.6.1.3 Classes and instances are callable

- Class objects and instances operate as callable objects. A class object is created by the 'class' statement and is called as function in order to create a new instance. The arguments passed in the to the function are passed to __init__() method, which initialize the newly created instance.
- An instance can emulate the function if it defines the special method __call__(). If a method is defined for an instance 'f', then f.methodname(args) actually invokes the method f.methodname.__call__(args).

```
>>> class Foo(object):
...     def __init__(self, val):
...             self.value =  val
...
...     def addValue(self, num):
...             print(self.value + num)
...
>>> f = Foo(3)       # class is called as function to create new instance
>>> f.addValue(4)
7

>>> f.addValue.__call__(5)
8

>>> i = f.addValue
>>> i.__call__(3)
6
```

7.6.2 Class, types and instances

When we define a class, then the class definition produces an object of type 'type', as shown below. Table 7.12 shows the various attributes available for the class,

```
>>> class Foo():
...     pass
...
>>> type(Foo)
<class 'type'>
>>>
```

Table 7.12: Class attributes

Attributes	Description
t.__doc__	documentation string
t.__name__	class name
t.__bases__	tuple of base classes
t.__dict__	dictionary holding the class methods and variables
t.__module__	module name in which the class is defined
t.__abstractmethods__	set of abstract method names

The instance of the class has some special attributes a shown in Table 7.13,

Table 7.13: Instance attributes

Attributes	Description
i.__class__	name of the class for instance
i.__dict__	dictionary holding the instance data

Note: The __dict__ attribute is normally where all the data associated with an instance is stored. However this behaviour can be changed by using __slots__ , which is a more efficient way for handling the large number of instances. In that case, __dict__ attribute will not be available for the instance.

7.6.3 Modules

Module type is a container that holds object loaded with 'import' statement. Whenever an attribute of the module is references, then corresponding dictionary is invoked e.g. m.x invokes m.__dict__['x']. Table 7.14 shows the various attributes available for module.

Table 7.14: Module attributes

Attributes	Description
m.__dict__	dictionary associated with module
m.__doc__	documenatation string
m.__name__	name of module
m.__file__	file from which module is loaded
m.__path__	fully qualified package name

7.7 Special methods

In this section, various special method are listed which handle different situations.

7.7.1 Object creation and destruction

Table 7.15 shows three methods which are used to create and delete the objects.

Table 7.15: Object creation and destruction

Method	Description
__new__(cls [,*args [,**kwargs]])	called to create a new object
__init__(self, [,*args [,**kwargs]])	called to initialize a new instance
__del__(self)	called to delete an instance

7.7.2 String representation

Table 7.16 shows three methods which are used with string objects,

Table 7.16: String representation

Method	Description
__format__(self, format_spec)	creates a formatted string
__repr__(self)	creates a string representation of an object
__str__(self)	create a simple string representation

7.7.3 Type checking

Table 7.17 shows two methods which are used for type checking,

Table 7.17: Type checking

Method	Result
__instancecheck__(cls, object)	isinstance(object, cls)
__subclasscheck__(cls, sub)	issubclass(sub, cls)

7.7.4 Attribute access

Table 7.18 shows the methods which are used to access the attribute using dot operator

Table 7.18: Attribute access

Method	Description
__getattribute__(self, name)	returns attribute self.name
__getattr__(self, name)	returns attribute self.name if not found through normal attribute lookup or raise error
__setattr__(self, name, value)	sets value of the attribute
__delattr__(self, name)	delete the attribute

Note: Whenever an attribute is accessed, then __getattribute__ is invoked. If attribute is not found, then __getattr__ is invoked. The, default behavior of __getattr__ is to raise 'AttributeError'.

7.7.5 Descriptors

A subtle aspect of attribute manipulation is that sometimes the attributes of an object are wrapped with an extra layer of logic with get, set and delete operations. This can be achieved with descriptors with three options which are listed in Table 7.19.

Table 7.18 shows the methods which are used to

Table 7.19: Special methods for Descriptor objects

Method	Description
__get__(self, instance, cls)	return the attribute value or raise error
__set__(self, instance, value)	set the attribute to value
__delete__(self, instance)	delete the attribute

7.7.6 Sequence and mapping methods

Table 7.20 shows the methods which are used to emulate the sequences and mapping objects

Table 7.20: Methods for sequences and mappings

Method	Description
__len__(self)	returns length of self
__getitem__(self, key)	returns self[key]
__setitem__(self, key, value	sets self[key]=value
__delitem__(self, key)	deletes self[key]
__contains__(self, obj)	returns True if obj is in self

7.7.7 Iteration

Table 7.21 shows the methods which are used with iterators. Further, following example is added for better understanding of the table,

```
>>> x = [1, 2, 3, 4, 5, 6]
>>> y = x.__iter__()
>>> y.__next__()
1
>>> y.__next__()
2
>>>

>>> # complete code for iteration
>>> x = [1, 2, 3, 4, 5, 6]
>>> y = x.__iter__()
>>> while 1:
...     try:
...         print(y.__next__(), end=", ")
...     except StopIteration:
...         break
...

1, 2, 3, 4, 5, 6,    # outputs
```

Table 7.21: Method for iterator

Method	Description
obj.__iter__()	returns the iterator object
iter.__next__()	return the next object or raise StopIteration error

7.7.8 Callable interface

An instance can be used as function if the class method contains __call__ method,

```
>>> class Foo:
...     def __init__(self, num):
...         self.num = num
...
...     def __call__(self, greeting):
...         print("{}, The name of the class is '{}'"
...               .format(greeting, self.__class__.__name__))
...
>>>

>>> f = Foo(3)
>>> f('Hello')      # instance is used as function
Hello, The name of the class is 'Foo'

>>> ############### another example ##########################

>>> class Spam:
...     def __init__(self, a=0, b=0):
...         self.a = a
...         self.b = b
...
...     def __call__(self, a=10, b=20):
...         self.a = a
...         self.b = b
...
```

(continues on next page)

(continued from previous page)

```
>>> # invoke __init__
>>> s = Spam()
>>> print("__init__ : ", s.a, s.b)
__init__ :  0 0
>>>
>>> # invoke __call__
>>> s(1, 2)
>>> print("__call__ : ", s.a, s.b)
__call__ :  1 2
```

7.7.9 Context management protocol

The 'with' statement allows statements to execute under the control over another object known as "context manager". The context manager has two methods i.e. __enter__ and __exit__ as shown in Table 7.22.

Table 7.22: Methods for context management

Methods	Description
__enter__(self)	called when entering a new context
__exit__(self, type, value, tb)	called when leaving the context. If an exception occurred, 'type', 'value' and 'tb' have the exception type, value and traceback information

- Primary use of the context management interface is to allow for simplified resource control on objects involving system state e.g. files, networks connection and database etc. By implementing this interface, an object can safely clean up resources when execution leaves a context in which the object is being used.

7.7.10 Object inspection and dir()

The dir() function is used to inspect the objects, which returns all the attributes of the objects. To provide only useful information about the object, it can be overwritten using __dir__(self) as shown below,

```
>>> class Spam:
...     def __init__(self, a=0, b=0):
...         self.a = a
...         self.b = b
...
...     def __call__(self, a=10, b=20):
...         self.a = a
...         self.b = b
...
...     def __dir__(self):
...         return ['__init__', '__call__', 'add more']
...
>>> s = Spam()
>>> dir(s)
['__call__', '__init__', 'add more']
>>>
```

Chapter 8

Exception handling

8.1 Introduction

Exception handling is the process of handling the run time error. Error may occur due to missing data and invalid data etc. These error can be catch in the runtime using try-except block and then can be processed according to our need. This chapter presents some of the examples of error handling.

For this first create a new file with missing data in it, as shown below. Here 'price' column is empty for Silver,

```
$ cat price_missing.csv

date,metal,radius,price,quantity
"2016-06-12","Gold",5.5,80.99,1
"2015-07-13","Silver",40.3,,3
"2016-01-21","Iron",9.2,14.29,8
"2014-03-23","Gold",8,120.3,2
"2017-09-11","Copper",4.1,70.25,12
"2011-01-20","Iron",3.25,10.99,3
```

Now try to calculate the total price for this file using 'ring_cost' function. A ValueError will be displayed as shown below,

```
>>> from price import ring_cost
>>> ring_cost('price_missing.csv')
Traceback (most recent call last):
    [...]
    row[3] = float(row[3]) # price
ValueError: could not convert string to float:
```

8.2 try-except block

The problem discussed in above section can be solved using try-except block. In this block, the 'try' statement can be used to try the string to float/int conversion; and if it fails then 'except' block can be used to skip the processing of that particular row, as shown below,

```
# price.py

import csv

def ring_cost(filename):
    ''' calculate the total cost '''
```

(continues on next page)

(continued from previous page)

```
    total_price = 0 # for all items in the list

    with open(filename, 'r') as f: # open file in read mode
        rows = csv.reader(f)
        header = next(rows) # skip line 1 i.e. header
        for row in rows:
            try:
                row[3] = float(row[3]) # price
                row[4] = int(row[4]) # quantity
            except ValueError: # process ValueError only
                print("Invalid data, row is skipped")
                continue
            total_price += row[3] * row[4]

    # print("Total price = %10.2f" % total_price)
    return total_price  # return total_price

def main():
    total = ring_cost('price.csv')  # function call
    print("Total price = %10.2f" % total) # print value

# standard boilerplate
# main is the starting function
if __name__ == '__main__':
    main()
```

Now process the file again and the processing will skip the invalid line and display the total price.

```
>>> from price import ring_cost
>>> ring_cost('price_missing.csv')
Invalid data, row is skipped
1311.8799999999999
```

8.3 Report error

In previous section, the invalid data was ignored and a message was printed. But it is better give some details about the error as well, which is discussed in this section.

8.3.1 Type of error

In the below code, the type of the error is printed on the screen.

```
# price.py

import csv

def ring_cost(filename):
    ''' calculate the total cost '''

    total_price = 0 # for all items in the list

    with open(filename, 'r') as f: # open file in read mode
        rows = csv.reader(f)
        header = next(rows) # skip line 1 i.e. header
        for row in rows:
            try:
                row[3] = float(row[3]) # price
```

(continues on next page)

(continued from previous page)

```
                row[4] = int(row[4]) # quantity
            except ValueError as err: # process ValueError only
                print("Invalid data, row is skipped")
                print('Reason :', err)
                continue
            total_price += row[3] * row[4]

    # print("Total price = %10.2f" % total_price)
    return total_price  # return total_price

def main():
    total = ring_cost('price.csv')  # function call
    print("Total price = %10.2f" % total) # print value

# standard boilerplate
# main is the starting function
if __name__ == '__main__':
    main()
```

Note: Do not forget to restart the Python shell after changing the code.

```
>>> from price import ring_cost
>>> ring_cost('price_missing.csv')
Invalid data, row is skipped
Reason : could not convert string to float:
1311.8799999999999
```

8.3.2 Location of error

We can use the 'enumerate' to display the location of the error.

```
# price.py

import csv

def ring_cost(filename):
    ''' calculate the total cost '''

    total_price = 0 # for all items in the list

    with open(filename, 'r') as f: # open file in read mode
        rows = csv.reader(f)
        header = next(rows) # skip line 1 i.e. header
        for row_num, row in enumerate(rows, start=1): # start from 1, not 0)
            try:
                row[3] = float(row[3]) # price
                row[4] = int(row[4]) # quantity
            except ValueError as err: # process ValueError only
                print("Invalid data, row is skipped")
                print('Row: {}, Reason : {}'.format(row_num, err))
                continue
            total_price += row[3] * row[4]

    # print("Total price = %10.2f" % total_price)
    return total_price  # return total_price

def main():
```

(continues on next page)

(continued from previous page)

```
    total = ring_cost('price.csv')  # function call
    print("Total price = %10.2f" % total) # print value

# standard boilerplate
# main is the starting function
if __name__ == '__main__':
    main()
```

Now run the code again and it will display the location of the error,

```
>>> from price import ring_cost
>>> ring_cost('price_missing.csv')
Invalid data, row is skipped
Row: 2, Reason : could not convert string to float:
1311.8799999999999
```

8.4 Catch all error (bad practice)

In previous sections, we catch the specific error i.e. 'ValueError'. We can catch all types of errors as well using 'Exception' keyword, but this may result in misleading messages, as shown in this section.

First replace the 'ValueError' with 'Exception' as below,

```
# price.py

import csv

def ring_cost(filename):
    ''' calculate the total cost '''

    total_price = 0 # for all items in the list

    with open(filename, 'r') as f: # open file in read mode
        rows = csv.reader(f)
        header = next(rows) # skip line 1 i.e. header
        for row_num, row in enumerate(rows, start=1): # start from 1, not 0)
            try:
                row[3] = float(row[3]) # price
                row[4] = int(row[4]) # quantity
            except Exception as err: # process ValueError only
                print("Invalid data, row is skipped")
                print('Row: {}, Reason : {}'.format(row_num, err))
                continue
            total_price += row[3] * row[4]

    # print("Total price = %10.2f" % total_price)
    return total_price  # return total_price

def main():
    total = ring_cost('price.csv')  # function call
    print("Total price = %10.2f" % total) # print value

# standard boilerplate
# main is the starting function
if __name__ == '__main__':
    main()
```

Now run the code and it will work fine as shown below,

```
>>> from price import ring_cost
>>> ring_cost('price_missing.csv')
Invalid data, row is skipped
Row: 2, Reason : could not convert string to float:
1311.8799999999999
```

Next, replace the 'int' with 'integer' at Line 16, i.e. we are introducing error in the code.

```
# price.py

import csv

def ring_cost(filename):
    ''' calculate the total cost '''

    total_price = 0 # for all items in the list

    with open(filename, 'r') as f: # open file in read mode
        rows = csv.reader(f)
        header = next(rows) # skip line 1 i.e. header
        for row_num, row in enumerate(rows, start=1): # start from 1, not 0)
            try:
                row[3] = float(row[3]) # price
                row[4] = integer(row[4]) # quantity
            except Exception as err: # process ValueError only
                print("Invalid data, row is skipped")
                print('Row: {}, Reason : {}'.format(row_num, err))
                continue
            total_price += row[3] * row[4]

    # print("Total price = %10.2f" % total_price)
    return total_price  # return total_price

def main():
    total = ring_cost('price.csv')  # function call
    print("Total price = %10.2f" % total) # print value

# standard boilerplate
# main is the starting function
if __name__ == '__main__':
    main()
```

Now run the code again and it will give display following messages, which has no relation with the actual error. Actual error is the 'integer' at line 16; since the conversion operation can not be performed now (due to invalid type 'integer'), therefore error is catch for all the rows and the messages are generated for each row. Therefore it is very bad idea to catch errors using "Exception" keyword.

```
>>> from price import ring_cost
>>> ring_cost('price_missing.csv')
Invalid data, row is skipped
Row: 1, Reason : name 'integer' is not defined
Invalid data, row is skipped
Row: 2, Reason : could not convert string to float:
Invalid data, row is skipped
Row: 3, Reason : name 'integer' is not defined
Invalid data, row is skipped
Row: 4, Reason : name 'integer' is not defined
Invalid data, row is skipped
Row: 5, Reason : name 'integer' is not defined
Invalid data, row is skipped
Row: 6, Reason : name 'integer' is not defined
0
```

8.5 Silencing error

Sometimes it is undesirable to display the error messages of the except blocks. In such cases, we want to silent the error messages, which is discussed in this section.

First, undo the changes made in previous section, i.e. replace 'integer' with 'int' and 'Exception' with 'ValueError'.

Now, we will consider the following three cases to handle the error,

1. silent : do not display error message
2. warn : display the error message
3. stop : stop execution of code, if error is detected

For this one positional argument 'mode' is defined, whose default value is set to 'warn', and then put the 'print' statement inside the 'if-else' block. It is good idea to set the default value of 'mode' to 'warn' as we do not want to pass the error silently.

Listing 8.1: Silencing error

```
# price.py

import csv

# warn is kept as default, as error should not be passed silently
def ring_cost(filename, mode='warn'):
    ''' calculate the total cost '''

    total_price = 0 # for all items in the list

    with open(filename, 'r') as f: # open file in read mode
        rows = csv.reader(f)
        header = next(rows) # skip line 1 i.e. header
        for row_num, row in enumerate(rows, start=1): # start from 1, not 0)
            try:
                row[3] = float(row[3]) # price
                row[4] = int(row[4]) # quantity
            except ValueError as err: # process ValueError only
                if mode == 'warn':
                    print("Invalid data, row is skipped")
                    print('Row: {}, Reason : {}'.format(row_num, err))
                elif mode == 'silent':
                    pass # do nothing
                elif mode == 'stop':
                    raise # raise the exception
                continue
            total_price += row[3] * row[4]

    # print("Total price = %10.2f" % total_price)
    return total_price  # return total_price

def main():
    total = ring_cost('price.csv')  # function call
    print("Total price = %10.2f" % total) # print value

# standard boilerplate
# main is the starting function
if __name__ == '__main__':
    main()
```

Below are the outputs for each of the cases .. code-block:: python

```
>>> from price import ring_cost
>>> ring_cost('price_missing.csv')  # default 'warn'
```

(continues on next page)

(continued from previous page)

```
Invalid data, row is skipped
Row: 2, Reason : could not convert string to float:
1311.8799999999999
```

```
>>> ring_cost('price_missing.csv', mode='warn')
Invalid data, row is skipped
Row: 2, Reason : could not convert string to float:
1311.8799999999999
```

```
>>> ring_cost('price_missing.csv', mode='silent')
1311.8799999999999
```

```
>>> ring_cost('price_missing.csv', mode='stop')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/media/dhriti/work/git/advance-python-tutorials/codes/price.py", line 16, in ring_cost
    row[3] = float(row[3]) # price
ValueError: could not convert string to float:
```

8.6 List of Exception in Python

Following is the list of exceptions available in Python,

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
           +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
```

(continues on next page)

(continued from previous page)

```
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

8.7 Conclusion

In this chapter, we saw various ways to handle the error along with some good practices. In next chapter, we will discuss the 'data manipulation' techniques using various data structures available in Python.

Chapter 9

Data mining

Processing the data from a large file and finding patterns in it is known as data-mining. Data mining required lots of data cleaning and data transformation operations. In this section, we will see some of these operations.

9.1 Building data structure from file

In the previous chapters, we read the data from the file and then process the data. In this section, we will save the data in a list/dictionary and then use these data structure to process the data. We will see that the data processing operations become easier when the data is converted in the form of dictionary.

Important: In previous chapter, we read the file and calculated the total prices inside the function 'ring_cost'. The problem with this method is that the read data is no longer available (after the return statement) for the further processing.

Therefore, it is good idea to save the results in a list or dictionary, so that it will be available for other functions as well, as shown in this chapter.

Lets see the contents of the 'price.csv' file again,

```
$ cat price.csv
date,metal,radius,price,quantity
"2016-06-12","Gold",5.5,80.99,1
"2015-07-13","Silver",40.3,5.5,3
"2016-01-21","Iron",9.2,14.29,8
"2014-03-23","Gold",8,120.3,2
"2017-09-11","Copper",4.1,70.25,12
"2011-01-20","Iron",3.25,10.99,3
```

9.1.1 Save and read data in list

Create a new file 'datamine.py' with following contents. These contents are same as Listing 8.1, except it returns the list. Also, a check is applied for input 'mode' value.

```
# datamine.py

import csv

def read_file(filename, mode='warn'):
    ''' read csv file and save data in the list '''
```

(continues on next page)

(continued from previous page)

```
    # check for correct mode
    if mode not in ['warn', 'silent', 'stop']:
        raise ValueError("possible modes are 'warn', 'silent', 'stop'")

    ring_data = [] # create empty list to save data

    with open (filename, 'r') as f:
        rows =  csv.reader(f)
        header = next(rows) # skip the header

        # change the types of the columns
        for row in rows:
            try:
                row[2] = float(row[2]) # radius
                row[3] = float(row[3]) # price
                row[4] = int(row[4]) # quantity
            except ValueError as err: # process value error only
                if mode == 'warn':
                    print("Invalid data, row is skipped")
                    print('Row: {}, Reason : {}'.format(row_num, err))
                elif mode == 'silent':
                    pass # do nothing
                elif mode == 'stop':
                    raise # raise the exception
                continue

            # append data in list in the form of tuple
            ring_data.append(tuple(row))

    return ring_data

def main():
    ring_data = read_file('price.csv')

    # total rows in the file
    print("Total rows: ", len(ring_data))

    # total price calculation
    total_price = 0
    for row in ring_data:
        total_price += row[3] * row[4]
    print("Total price: {:10.2f}".format(total_price))

if __name__ == '__main__':
    main()
```

Run the above code and we will get the following results,

```
$ python datamine.py
Total rows:  6
Total price:    1328.38
```

Now, open the Python shell and run the below code. See the difference, "Previously we returned total_price from the function, therefore we could perform no more operation on the data. But, now we have the data in the form of List, therefore we can perform operation on the data.

```
>>> from datamine import read_file
>>> ring_data = read_file('price.csv')
>>> len(ring_data)
6
>>> ring_data[0]
```

(continues on next page)

(continued from previous page)

```
   ('2016-06-12', 'Gold', 5.5, 80.99, 1)

   >>> for data in ring_data: # print metal with radius > 9
   ...     if data[2] > 9:
   ...             print("Metal: {0}, Radius: {1}".format(data[1], data[2]))
   ...
   Metal: Silver, Radius: 40.3
   Metal: Iron, Radius: 9.2


.. `data_in_dict`:
```

9.1.2 Save and read data in Dictionary

In the previous section, the list is read and data is printed (i.e. name of metal when radius > 9). It worked fine there, but when we have a large number of columns in the list, then it is very difficult to locate the elements using positions e.g. 'data[2]'. For easy referencing, a dictionary can be used as shown below.

Note that, at line 58, the elements are located by the name, i.e. row['price'], which is easier to handle than using index e.g. row[3].

```
# datamine.py

import csv

def read_file(filename, mode='warn'):
    ''' read csv file and save data in the list '''

    # check for correct mode
    if mode not in ['warn', 'silent', 'stop']:
        raise ValueError("possible modes are 'warn', 'silent', 'stop'")

    ring_data = [] # create empty list to save data

    with open (filename, 'r') as f:
        rows =  csv.reader(f)
        header = next(rows) # skip the header

        # change the types of the columns
        for row in rows:
            try:
                row[2] = float(row[2]) # radius
                row[3] = float(row[3]) # price
                row[4] = int(row[4]) # quantity
            except ValueError as err: # process value error only
                if mode == 'warn':
                    print("Invalid data, row is skipped")
                    print('Row: {}, Reason : {}'.format(row_num, err))
                elif mode == 'silent':
                    pass # do nothing
                elif mode == 'stop':
                    raise # raise the exception
                continue

            # ring_data.append(tuple(row))

            # append data in list in the form of tuple
            row_dict = {
                    'date' : row[0],
```

(continues on next page)

(continued from previous page)

```
                    'metal' : row[1],
                    'radius' : row[2],
                    'price' : row[3],
                    'quantity' : row[4]
                }

            ring_data.append(row_dict)

    return ring_data

def main():
    ring_data = read_file('price.csv')

    # total rows in the file
    print("Total rows: ", len(ring_data))

    # total price calculation
    total_price = 0
    for row in ring_data:
        total_price += row['price'] * row['quantity']
    print("Total price: {:10.2f}".format(total_price))

if __name__ == '__main__':
    main()
```

Following is the output of above code,

```
$ python datamine.py
Total rows:  6
Total price:    1328.38
```

9.2 List comprehension

In previous section, we read the data from the file and stored in the list/dictionary to perform further operations.

In this section, we will extract a specific type of data and store them in a new list. Let's do it in the Python shell as below,

9.2.1 Basic method for extraction

In the below code, if-statement along with the 'loop' is used to extract the desired data, i.e. radius < 5.

```
>>> from datamine import read_file
>>> ring_data = read_file('price.csv')
>>> small_ring = []
>>> for ring in ring_data:
...     if ring['radius'] < 5: # store radius < 5
...         small_ring.append((ring['metal'], ring['radius'],  ring['price']))
...
>>> for ring in small_ring: # display content of small_ring
...     print(ring)
...
('Copper', 4.1, 70.25)
('Iron', 3.25, 10.99)
```

9.2.2 List comprehension for extraction

Operation in above section, i.e. if statement with loop, is very common, therefore Python provide a way to do it in one line, which is known as 'list comprehension', as shown below,

```
>>> from datamine import read_file
>>> ring_data = read_file('price.csv')
>>> small_ring = []
>>> small_ring = [(ring['metal'], ring['radius'], ring['price'])
...     for ring in ring_data if ring['radius'] < 5 ]
>>>
>>> for ring in small_ring:
...     print(ring)
...
('Copper', 4.1, 70.25)
('Iron', 3.25, 10.99)
```

9.2.3 Lambda operator

In numref:*data_in_dict*, the data was saved in the dictionary and then a specific type of data is extraced in above setion. In this section, we will sort the data store in the dictionary.

9.2.4 Basis method for sorting

Let's do it in the Python shell as shown below. First see the content of the dictionary again,

```
>>> from datamine import read_file
>>> ring_data = read_file('price.csv')
>>> for data in ring_data:
...     print(data)
...
{'date': '2016-06-12', 'metal': 'Gold', 'radius': 5.5,
    'price': 80.99, 'quantity': 1}

{'date': '2015-07-13', 'metal': 'Silver', 'radius': 40.3,
    'price': 5.5, 'quantity': 3}

{'date': '2016-01-21', 'metal': 'Iron', 'radius': 9.2,
    'price': 14.29, 'quantity': 8}

{'date': '2014-03-23', 'metal': 'Gold', 'radius': 8.0,
    'price': 120.3, 'quantity': 2}

{'date': '2017-09-11', 'metal': 'Copper', 'radius': 4.1,
    'price': 70.25, 'quantity': 12}

{'date': '2011-01-20', 'metal': 'Iron', 'radius': 3.25,
    'price': 10.99, 'quantity': 3}
```

Note that, unlike list, we can not perform the sort() operation on dictionary. We will have following error,

```
>>> ring_data.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'dict' and 'dict'
```

To sort the data in dictionary, we need to provide a 'key' to the sort() method. The key can be extracted from the dictionary using a function. Then this function should be called using sort() method as shown below,

Note: sort() method has the 'key' argument i.e. it is not user-defined name.

```
>>> def sort_name(all_data):
...     return all_data['metal']
...
>>>
>>> ring_data.sort(key=sort_name)  # sort the data
>>> for data in ring_data:
...     print(data)
...
{'date': '2017-09-11', 'metal': 'Copper', 'radius': 4.1,
    'price': 70.25, 'quantity': 12}

{'date': '2016-06-12', 'metal': 'Gold', 'radius': 5.5,
    'price': 80.99, 'quantity': 1}

{'date': '2014-03-23', 'metal': 'Gold', 'radius': 8.0,
    'price': 120.3, 'quantity': 2}

{'date': '2016-01-21', 'metal': 'Iron', 'radius': 9.2,
    'price': 14.29, 'quantity': 8}

{'date': '2011-01-20', 'metal': 'Iron', 'radius': 3.25,
    'price': 10.99, 'quantity': 3}

{'date': '2015-07-13', 'metal': 'Silver', 'radius': 40.3,
    'price': 5.5, 'quantity': 3}
```

9.2.5 Lambda operator

The problem with above method is that we need to define a one line function for each key e.g. 'metal', 'price' and 'radius' etc. (if we want to sort on every key), which is not desirable coding-style.

Lambda operators are the one operators which can be used to replace the one line function. Let's see some example of Lambda operator first,

```
>>> sq = lambda x : x**2  # one line function for x**2
>>> sq(3)
9

>>> sum2Num = lambda x, y : x + y # lambda operator with two variable
>>> sum2Num(3, 4)
7
```

Note: In 'sq = lambda x : x**2', the 'x' is the input argument to function and the value after ':', i.e. x**2, is the return value. And 'sq' is the name of the function i.e. the statement is equivalent to below code,

```
def sq(x):
    return x**2
```

Now, we will use the sort() method using lambda operator as shown below,

```
>>> ring_data.sort(key=lambda all_data : all_data['metal'])
>>> for data in ring_data:
...     print(data)
...
```

(continues on next page)

(continued from previous page)

```
{'date': '2017-09-11', 'metal': 'Copper', 'radius': 4.1,
    'price': 70.25, 'quantity': 12}

{'date': '2016-06-12', 'metal': 'Gold', 'radius': 5.5,
    'price': 80.99, 'quantity': 1}

{'date': '2014-03-23', 'metal': 'Gold', 'radius': 8.0,
    'price': 120.3, 'quantity': 2}

{'date': '2016-01-21', 'metal': 'Iron', 'radius': 9.2,
    'price': 14.29, 'quantity': 8}

{'date': '2011-01-20', 'metal': 'Iron', 'radius': 3.25,
    'price': 10.99, 'quantity': 3}

{'date': '2015-07-13', 'metal': 'Silver', 'radius': 40.3,
    'price': 5.5, 'quantity': 3}
```

9.3 Find and arrange Gold rings

Let's add List comprehension and Lambda operator in the file 'datamine.py'. In the below code, the Gold rings are extracted first; and then the rings are arranged in decreasing order according to radius.

Listing 9.1: Find and arrange Gold rings

```
# datamine.py

import csv

def read_file(filename, mode='warn'):
    ''' read csv file and save data in the list '''

    # check for correct mode
    if mode not in ['warn', 'silent', 'stop']:
        raise ValueError("possible modes are 'warn', 'silent', 'stop'")

    ring_data = [] # create empty list to save data

    with open (filename, 'r') as f:
        rows =  csv.reader(f)
        header = next(rows) # skip the header

        # change the types of the columns
        for row in rows:
            try:
                row[2] = float(row[2]) # radius
                row[3] = float(row[3]) # price
                row[4] = int(row[4]) # quantity
            except ValueError as err: # process value error only
                if mode == 'warn':
                    print("Invalid data, row is skipped")
                    print('Row: {}, Reason : {}'.format(row_num, err))
                elif mode == 'silent':
                    pass # do nothing
                elif mode == 'stop':
                    raise # raise the exception
                continue
```

(continues on next page)

(continued from previous page)

```
            # ring_data.append(tuple(row))

            # append data in list in the form of tuple
            row_dict = {
                    'date' : row[0],
                    'metal' : row[1],
                    'radius' : row[2],
                    'price' : row[3],
                    'quantity' : row[4]
                }

            ring_data.append(row_dict)

    return ring_data

def main():
    ring_data = read_file('price.csv')

    # # total rows in the file
    # print("Total rows: ", len(ring_data))

    # # total price calculation
    # total_price = 0
    # for row in ring_data:
        # total_price += row['price'] * row['quantity']
    # print("Total price: {:10.2f}".format(total_price))

    # extract Gold-ring : using List comprehension
    gold_ring = [ring for ring in ring_data if ring['metal'] == 'Gold']
    for ring in gold_ring: # print metal and radius
        print("Metal: {0}, Radius: {1}".format(ring['metal'], ring['radius']))

    # reverse-sort the data in gold_ring : using Lambda operators
    gold_ring.sort(key=lambda data : data['radius'], reverse=True)
    print("\nRadius in descending order:")
    for ring in gold_ring: # print metal and radius
        print("Metal: {0}, Radius: {1}".format(ring['metal'], ring['radius']))

if __name__ == '__main__':
    main()
```

Now execute the file and we will get the following results,

```
$ python datamine.py
Metal: Gold, Radius: 5.5
Metal: Gold, Radius: 8.0

Radius in descending order:
Metal: Gold, Radius: 8.0
Metal: Gold, Radius: 5.5
```

9.4 Conclusion

In this chapter, we store the data in the list and dictionary. Then we perform the extraction operation on the dictionary using 'list comprehension'. Lastly we used the 'lambda operator' for sorting the data in the dictionary.

Chapter 10

Object oriented programming

10.1 Pythonic style

In this chapter, we will see the basic differences between the OOPs methods in Python and other programming languages e.g. C++ and Java etc. This understanding is essential to use the Python language effectively; and to write the code in Pythonic ways, i.e. not converting a C++/Java code into a Python code.

Here we will see the actual logic behind various pieces of Python language e.g. instances, variables, method and @property etc. Also, we will see the combine usage of these pieces to complete a design with Agile methodology.

10.2 Simple data structure to Classes

In this section, we will convert our previous codes into classes.

10.2.1 Instance variable

First create a class, which can contain various columns of the csv file as shown in the below code. Please note the following points,

- __init__ is the initializer (often called the constructor), which initializes the instance variables.
- self.radius, self.date and self.metal are the instance variables which are created by the __init__.

```
# pythonic.py

import math

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    def __init__(self, date, metal, radius, price, quantity):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.
```

(continues on next page)

(continued from previous page)

```
        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity

    def cost(self):
        return self.price * self.quantity

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

10.2.2 Object of class

Let's create an object 'r' of the class 'Ring', to verify the operation of the class, as shown below,

```
>>> from pythonic import Ring
>>> r = Ring("2017=08-10", "Gold", 5.5, 10.5, 10)
>>> r.metal
'Gold'
>>> r.cost()
105.0
>>> r.area()
95.03317777109125
```

10.2.3 Class variable and initialization

Lets, add some class variables to the above class. And define some initial values to the arguments in the __init__ function as shown below,

---

Note: Class variable should be used to create the shared variables. It is always good to put the shared variables on the class level rather than instance level.

```
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
                 price=5.0, quantity=5):
```

(continues on next page)

(continued from previous page)

```
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity

    def cost(self):
        return self.price * self.quantity

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

Lets check the functionality of the above code,

```
$ python -i pythonic.py
```

Note that we used '-i' option, therefore there is no need to import the class 'Ring' as shown in below code,

```
>>> r = Ring() # no paratmer pass therefore default values will be used
>>> r.date # instance variable
'2017-10-27'
>>> r.radius # instance variable
5.0
>>> r.center # class variable
0.0
>>> r.cost() # class method
25.0
>>> r.area() # class method
78.53981633974483
```

Also, we can modify the instance and class variables values as shown below,

```
>>> r.price # current value
5.0
>>> r.quantity
5
>>> r.cost()
25.0
>>> r.quantity=10 # modify instance variable
>>> r.cost() # price is changed to 50
50.0
>>> r.center = 10 # modify class variable
>>> r.center
10
```

Note: The value of the variables are stored in the dictionary, whose contents can be seen using '__dict__', i.e.,

```
>>> r.__dict__
{'date': '2017-10-27', 'metal': 'Copper', 'radius': 5.0,
'price': 5.0, 'quantity': 10, 'center': 10}
```

10.3 Shipping product and Agile methodology

In previous section, we create the object of the class in the Python shell. Now, we will create object of the class in the file itself, as show below,

Important: Note that, at this moment, we did not add to many features to class in Listing 10.1. We added only two methods here i.e. 'area' and 'cost'. Now, we are ready to ship this product to our customers.

We will add more features according to feedback provided by the costumers as shown in subsequent sections. This is called the 'Agile' methodology. This will give us a chance to understand the psychology behind the several elements of the Python language.

Listing 10.1: Shipping product is ready

```
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
                 price=5.0, quantity=5):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity

    def cost(self):
        return self.price * self.quantity
```

(continues on next page)

(continued from previous page)

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

def main():
    print("Center of the Ring is at:", Ring.center) # modify class variable
    r = Ring(price=8) # modify only price
    print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))

if __name__ == '__main__':
    main()
```

Following are the results for above code,

```
$ python pythonic.py
Center of the Ring is at: 0.0
Radius:5.0, Cost:40
```

10.4 Attribute access

Before moving further, let us understand the attribute access (i.e. variable in the class).

10.4.1 get, set and del

In python, everything is an object. And there are only three operations which can be applied to the objects i.e.,

- get
- set
- delete

These operations are used as below,

```
>>> from pythonic import Ring
>>> r = Ring()
>>>
>>> r.metal # get operation
'Copper'
>>>
>>> r.metal = "Gold" # set operation
>>> r.metal
'Gold'
>>>
>>> del r.metal, r.date, r.price  # delete operation
>>> r.metal
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Ring' object has no attribute 'metal'
>>>
>>> r.__dict__  # only radius and quantity left
{'radius': 5.0, 'quantity': 5}
```

Warning: If attribute does not exist in the dictionary, then it will be created, e.g. in the below code, we used h.metals (plural metals) for the metal name instead of h.metal, therefore a new attribute will be created. Hence, be careful while setting the attributes of the class.

```
>>> r.metals = "Iron" # add item to dict
>>> r.__dict__
{'radius': 5.0, 'quantity': 5, 'metals': 'Iron'}
```

Note: The class method has two layers of the get-set-del operations as shown below,

```
>>> r.area()
78.53981633974483
>>>
>>> (r.area)() # (layer1)(layer2)
78.53981633974483
>>>
>>> a = r.area # layer 1
>>> a
<bound method Ring.area of <pythonic.Ring object at 0xb7132f2c>>
>>> a() # layer 2
78.53981633974483
```

10.4.2 getattr, setattr and delattr

getattr, setattr and delattr are invoked when we use the get, set and delete operations respectively. The knowledge of these methods can be very useful in writing the general purpose codes as shown in Section 10.5.2.

Below an example of getattr and setattr, where the columns are printed using 'getattr'. Note that the code is generalized here, as we can print all the columns using one statement only i.e. Lines 10-11.

```
1  >>> from pythonic import Ring
2  >>> r = Ring()
3  >>> getattr(r, 'metal')  # get
4  'Copper'
5  >>> setattr(r, 'metal', 'Gold')  # set
6  >>> r.metal
7  'Gold'
8  >>>
9  >>> out_col = ['metal', 'radius', 'quantity']
10 >>> for col in out_col: # print columns and value using getattr
11 ...     print("{0} : {1}".format(col, getattr(r, col)))
12 ...
13 metal : Gold
14 radius : 5.0
15 quantity : 5
16 >>>
17 >>> delattr(r, 'metal') # delete 'metal'
18 >>> r.__dict__ # 'metal' is removed from dictionary
19 {'date': '2017-10-27', 'radius': 5.0, 'price': 5.0, 'quantity': 5}
```

10.5 Users

Lets assume that we have two types of users for our product. One of them is 'mathematician' who is using our product for the mathematical analysis of the data; whereas the other is a 'contributer', who is implementing additional features to the product. Both of them will provide some feedbacks according to their need and we will modify our class to meet their requirement.

10.5.1 Average area

The first user, i.e. mathematician, uses our design to calculate the average area of the ring, as shown below,

Listing 10.2: Average area

```
# mathematician.py
# user-1: using Ring.py for mathematical analysis

from random import random, seed
from pythonic import Ring

def calc_avg_area(n=5, seed_value=3):
    # seed for random number generator
    seed(seed_value)

    # random radius
    rings = [Ring(radius=random()) for i in range(n)]

    total = 0
    for r in rings:
        total += r.area()
        # # print values for each iteration
        print("%0.2f, %0.2f, %0.2f" % (r.radius, r.area(), total))
    avg_area = sum([r.area() for r in rings])/n

    return avg_area

def main():
    # generate 'n' rings
    n = 10
    avg_area = calc_avg_area(n=10)
    print("\nAverage area for n={0} is n/{0} = {1:.2f}".format(n, avg_area))

if __name__ == '__main__':
    main()
```

Following is the result of above code,

```
$ python mathematician.py

0.24, 0.18, 0.18
0.54, 0.93, 1.11
0.37, 0.43, 1.54
0.60, 1.15, 2.68
0.63, 1.23, 3.91
0.07, 0.01, 3.93
0.01, 0.00, 3.93
0.84, 2.20, 6.13
0.26, 0.21, 6.34
0.23, 0.17, 6.52

Average area for n=10 is n/10 = 0.65
```

10.5.2 Table formatting

The second user, i.e. contributor, has added the table-formatting functionality for class 'Ring'.

Note: The code is generalized format as compared to Listing 9.1. Following are the changes made here,

- At line 31, the conversion is generalized using 'list comprehension'.
- The function 'print_table (Lines 67-76)' is generalized using 'getattr' method. Here, we can print desired columns of the table by passing a list of 'column names'.

Listing 10.3: Table formatter

```
# contributor.py
# user-2: additional features will be added to pythonic.py by the contributor

import csv

from pythonic import Ring

# this code is copied from datamine.py file and modified slightly
# to include the type of the data
def read_file(filename, types, mode='warn'):
    ''' read csv file and save data in the list '''

    # check for correct mode
    if mode not in ['warn', 'silent', 'stop']:
        raise ValueError("possible modes are 'warn', 'silent', 'stop'")

    ring_data = [] # create empty list to save data

    with open (filename, 'r') as f:
        rows =  csv.reader(f)
        header = next(rows) # skip the header

        # change the types of the columns
        for row in rows:
            try:
                # row[2] = float(row[2]) # radius
                # row[3] = float(row[3]) # price
                # row[4] = int(row[4]) # quantity

                # generalized conversion
                row = [d_type(val) for d_type, val in zip(types, row)]
            except ValueError as err: # process value error only
                if mode == 'warn':
                    print("Invalid data, row is skipped")
                    print('Row: {}, Reason : {}'.format(row_num, err))
                elif mode == 'silent':
                    pass # do nothing
                elif mode == 'stop':
                    raise # raise the exception
                continue

            # ring_data.append(tuple(row))

            # append data in list in the form of tuple
            # row_dict = {
                    # 'date' : row[0],
                    # 'metal' : row[1],
                    # 'radius' : row[2],
                    # 'price' : row[3],
                    # 'quantity' : row[4]
                # }

            # row_dict = Ring(row[0], row[1], row[2], row[3], row[4])
            # # use below or above line
            row_dict = Ring(
                    date = row[0],
                    metal = row[1],
                    radius = row[2],
                    price = row[3],
```

(continues on next page)

(continued from previous page)

```
                    quantity = row[4]
                )

            ring_data.append(row_dict)

    return ring_data

# table formatter
def print_table(list_name, col_name=['metal', 'radius']):
    """ print the formatted output """

    for c in col_name: # print header
        print("{:>7s}".format(c), end=' ')
    print() # print empty line
    for l in list_name: # print values
        for c in col_name:
            print("{:>7s}".format(str(getattr(l, c))), end=' ')
        print()

def main():
    # list correct types of columns in csv file
    types =[str, str, float, float, int]

    # read file and save data in list
    list_data = read_file('price.csv', types)

    # formatted output
    print_table(list_data)
    print()
    print_table(list_data, ['metal', 'radius', 'price'])

if __name__ == '__main__':
    main()
```

Following are the output for the above code,

```
$ python contributor.py

metal  radius
  Gold     5.5
Silver    40.3
  Iron     9.2
  Gold     8.0
Copper     4.1
  Iron    3.25

 metal  radius   price
  Gold     5.5   80.99
Silver    40.3     5.5
  Iron     9.2   14.29
  Gold     8.0   120.3
Copper     4.1   70.25
  Iron    3.25   10.99
```

10.6 Requirement : perimeter method

In Listing 10.2, the mathematician calculated the average area. Suppose, mathematician want to do some analysis on the perimeter of the ring as well, therefore he asked us to add a method to calculate the perimeter as well.

This is quite easy task, and can be implemented as below,

```
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
                 price=5.0, quantity=5):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity

    def cost(self):
        return self.price * self.quantity

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

    def perimeter(self):
        return 2 * math.pi * self.radius

def main():
    print("Center of the Ring is at:", Ring.center) # modify class variable
    r = Ring(price=8) # modify only price
    print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))
    print("Radius:{0}, Perimeter:{1:0.2f}".format(r.radius, r.perimeter()))

if __name__ == '__main__':
    main()
```

10.7 No data hiding in Python

In previous section, we added the method 'perimeter' in the design. Note that, in Python all the attributes can be directly accessed by the clients, and there is no concept of data hiding in the Python. Let's understand this by an example.

Important: In the below code, the mathematician uses the attribute 'radius' directly by changing it to 'r.radius = expansion(r.radius)' at Line 39. Then the new radius is used to calculate the perimeter at Line 40. In the other word, value of radius is changed in the dictionary itself, and then the method perimeter() is used for calculation.

Such access to attributes is available in Python only; whereas other languages e.g. Java and C++ etc. uses the concept of data hiding (using private and protected variable) and the attributes can not be directly accessed. These languages provide some get and set method to access and modify the attributes, i.e. we can make a local copy of the variable for further calculation; whereas in Python, the value is changed in the dictionary itself.

Data hiding is required in C++ and Java etc. as direct access can be a serious problem there and can not be resolved. In Python, data is not hidden from user and we have various methods to handle all kind of situations as shown in this chapter.

Listing 10.4: Accessing the class attribute directly

```
1  # mathematician.py
2  # user-1: using Ring.py for mathematical analysis
3
4  from random import random, seed
5  from pythonic import Ring
6
7  def calc_avg_area(n=5, seed_value=3):
8      # seed for random number generator
9      seed(seed_value)
10
11     # random radius
12     rings = [Ring(radius=random()) for i in range(n)]
13
14     total = 0
15     for r in rings:
16         total += r.area()
17         # # print values for each iteration
18         print("%0.2f, %0.2f, %0.2f" % (r.radius, r.area(), total))
19     avg_area = sum([r.area() for r in rings])/n
20
21     return avg_area
22
23 def expansion(radius=1.0, expansion=2.0):
24         radius *= expansion  # 2.0 times radius expansion due to heat
25         return radius
26
27 def main():
28     # # generate 'n' rings
29     # n = 10
30     # avg_area = calc_avg_area(n=10)
31     # print("\nAverage area for n={0} is n/{0} = {1:.2f}".format(n, avg_area))
32
33     radii = [1, 3, 5] # list of radius
34     rings = [Ring(radius=r) for r in radii] # create object of different radius
35     for r in rings:
36         print("Radius:", r.radius)
37         print("Perimeter at room temperature: %0.2f" % r.perimeter())
38         # radius after expansion
39         r.radius = expansion(r.radius) # modifying the attribute of the class
40         print("Perimeter after heating:, %0.2f" % r.perimeter())
41
42 if __name__ == '__main__':
43     main()
```

Following is the output of above code,

```
$ python mathematician.py

Radius: 1
Perimeter at room temperature: 6.28
Perimeter after heating:, 12.57

Radius: 3
Perimeter at room temperature: 18.85
Perimeter after heating:, 37.70

Radius: 5
Perimeter at room temperature: 31.42
Perimeter after heating:, 62.83
```

10.8 Inheritance overview

Lets review the inheritance in Python quickly. Then we will see some good usage of inheritance in this chapter.

Note: No prior knowledge of the 'super()' and 'multiple inheritance' is required. In this chapter, we will see various examples to understand these topics.

- First create a parent class "Animal" as below,

```
>>> class Animal(object):
...     def __init__(self, name):
...             self.name = name
...
...     def sound(self):
...             print("Loud or soft sound")
...
...     def wild(self):
...             print("Wild or pet animail")
>>>
>>> a = Animal("Tiger")
>>> a.sound()
Loud or soft sound
```

- Now, create child class "PetAnimal" which inherit the class "Animal",

```
>>> class PetAnimal(Animal):
...     def pet_size(self):
...             print("small or big")
...
>>> p  = PetAnimal("cat")
>>> p.pet_size()
small or big
>>> p.sound()  # inherit from Animal
Loud or soft sound
```

- Next, overide the method of class "Animal" in the child class "Dog",

```
>>> class Dog(Animal):
...     def sound(self):
...             print("Dog barks")
...
>>> d = Dog("Tommy")
>>> d.sound() # override the 'sound' of class Animal
Dog barks
```

- We can use both parent and child class method with same name. This can be done using 'super()' as below,

```
>>> class Tiger(Animal):
...     def sound(self):
...             print("Tiger roars")
...             super().sound()  # call the parent class 'sound'
>>> t = Tiger("Tigger")
>>> t.sound() # invoke 'sound' of both child and parent
Tiger roars
Loud or soft sound
```

- Multiple inheritance is possible in Python, which can be used to handle complex situations. Below is an example of multiple inheritance,

```
>>> class Cat(Tiger, Animal):
...     pass
...
>>> c = Cat("Kitty")
>>> c.sound()
Tiger roars
Loud or soft sound
>>> c.wild()
Wild or pet animail
```

- Note that, Python creates a MRO (method resolution order) for multiple inheritance, and if it can not be created then error will be reported. The MRO for class 'Cat' is shown below,

```
>>> help(cat)
    class Cat(Tiger, Animal)
     |  Method resolution order:
     |      Cat
     |      Tiger
     |      Animal
```

- If we inherit Animal first and next Tiger, then below error will occur; because Python uses 'child first' approach and this inheritance will call the Parent first i.e. MRO will be "Cat->Animal->Tiger". And it will report error as it has 'parent first' i.e. Animal comes before Tiger.

```
>>> class Rat(Animal, Tiger):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases Animal, Tiger
```

10.9 New user : making boxes

Suppose, we get another user for our class Ring (i.e. file pythonic.py), who is creating boxes for the ring. For he needs slightly greater perimeter of the box than the ring.

Note that this user is inheriting the class 'Ring', and overriding the method 'perimeter()' at Line 11 as shown below,

Listing 10.5: Box company modifying the class method 'perimeter()'

```
1 # box.py
2 # user-3 : creating boxes for the ring
```

(continues on next page)

(continued from previous page)

```
3
4  from pythonic import Ring
5
6  class Box(Ring):
7      """ Modified perimeter for creating the box """
8
9      def perimeter(self): # override the method 'perimeter'
10         # perimeter is increased 2.0 times
11         return Ring.perimeter(self) * 2.0
12
13 def main():
14     b = Box(radius=8) # pass radius = 8
15     print("Radius:", b.radius)
16     print("Modified perimeter: %0.2f" % b.perimeter()) # (2*pi*radius) * 2
17
18 if __name__ == '__main__':
19     main()
```

Following is the output of above code,

```
$ python box.py
Radius: 8
Modified perimeter: 100.53
```

Note: Now, we have two users who are modifying the class attributes. First is the mathematician (mathematician.py), who is modifying the 'radius' in Listing 10.4. And other is the box company, who is modifying the method 'perimeter' in Listing 10.5. Lastly, we have a contributor who is creating a 'table formatter' in Listing 10.3 for displaying the output nicely.

10.10 Rewrite table formatter using Inheritance

10.10.1 Abstract class

The contributor decided to add more print-formats for the table. For this, he decided to use inheritance to rewrite the code Listing 10.3 using inheritance.

Also, the contributor used the abstract class at Line 11 with two abstract methods. The abstract methods are compulsory to be implemented in the child class.

Important:

- The abstract methods are compulsory to be implemented in the child class. In the other words, the abstract class is the way to force the child class to implement certain method, which are decorated with 'abstract-method'
- Here the aim is to rewrite the code in Listing 10.3 such that we need not to change anything in the function 'main()'. In the other words, if the main() function works fine as it is, then it ensures that the code will not break for the users, who are using Listing 10.3 for printing the table.

Below is the modified Listing 10.3,

Listing 10.6: Print-format using inheritance

```
1  # contributor.py
2  # user-2: additional features will be added to pythonic.py by the contributor
```

(continues on next page)

(continued from previous page)

```
3
4  from abc import ABC, abstractmethod
5  import csv
6
7  from pythonic import Ring
8
9
10 # Abstract class for table-format
11 class TableFormat(object):
12     """ Abastract class """
13
14     @abstractmethod
15     def heading(self, header): # must be implemented in child class
16         pass
17
18     @abstractmethod
19     def row(self, row_data): # must be implemented in child class
20         pass
21
22
23 # text format for table
24 class TextFormat(TableFormat):
25     """ Text format for table """
26
27     def heading(self, header): # print headers
28         for h in header:
29             print("{:>7s}".format(h), end=' ')
30         print()
31
32     def row(self, row_data): # print rows
33         for r in row_data:
34             print("{:>7s}".format(r), end=' ')
35         print()
36
37
38 # this code is copied from datamine.py file and modified slightly
39 # to include the type of the data
40 def read_file(filename, types, mode='warn'):
41     ''' read csv file and save data in the list '''
42
43     # check for correct mode
44     if mode not in ['warn', 'silent', 'stop']:
45         raise ValueError("possible modes are 'warn', 'silent', 'stop'")
46
47     ring_data = [] # create empty list to save data
48
49     with open (filename, 'r') as f:
50         rows =  csv.reader(f)
51         header = next(rows) # skip the header
52
53         # change the types of the columns
54         for row in rows:
55             try:
56                 # row[2] = float(row[2]) # radius
57                 # row[3] = float(row[3]) # price
58                 # row[4] = int(row[4]) # quantity
59
60                 # generalized conversion
61                 row = [d_type(val) for d_type, val in zip(types, row)]
62             except ValueError as err: # process value error only
63                 if mode == 'warn':
```

(continues on next page)

(continued from previous page)

```
                    print("Invalid data, row is skipped")
                    print('Row: {}, Reason : {}'.format(row_num, err))
                elif mode == 'silent':
                    pass # do nothing
                elif mode == 'stop':
                    raise # raise the exception
                continue

            # ring_data.append(tuple(row))

            # append data in list in the form of tuple
            # row_dict = {
                    # 'date' : row[0],
                    # 'metal' : row[1],
                    # 'radius' : row[2],
                    # 'price' : row[3],
                    # 'quantity' : row[4]
                # }

            # row_dict = Ring(row[0], row[1], row[2], row[3], row[4])
            # # use below or above line
            row_dict = Ring(
                    date = row[0],
                    metal = row[1],
                    radius = row[2],
                    price = row[3],
                    quantity = row[4]
                )

            ring_data.append(row_dict)

    return ring_data


# table formatter
def print_table(list_name, col_name=['metal', 'radius'],
        out_format=TextFormat()): # note that class is passed here
    """ print the formatted output """

    # for c in col_name: # print header
        # print("{:>7s}".format(c), end=' ')
    # print() # print empty line
    # for l in list_name: # print values
        # for c in col_name:
            # print("{:>7s}".format(str(getattr(l, c))), end=' ')
        # print()

    # invoke class-method for printing heading
    out_format.heading(col_name) # class is passed to out_format
    for l in list_name:
        # store row in a list
        row_data = [str(getattr(l, c)) for c in col_name]
        out_format.row(row_data) # pass rows to class-method row()


def main():
    # list correct types of columns in csv file
    types =[str, str, float, float, int]

    # read file and save data in list
    list_data = read_file('price.csv', types)
```

(continues on next page)

(continued from previous page)

```
    # formatted output
    print_table(list_data)
    print()
    print_table(list_data, ['metal', 'radius', 'price'])

if __name__ == '__main__':
    main()
```

Now run the code to see whether it is working as previously or not. The below output is same as for Listing 10.3,

```
$ python contributor.py

  metal  radius
   Gold     5.5
 Silver    40.3
   Iron     9.2
   Gold     8.0
 Copper     4.1
   Iron    3.25

  metal  radius   price
   Gold     5.5   80.99
 Silver    40.3     5.5
   Iron     9.2   14.29
   Gold     8.0   120.3
 Copper     4.1   70.25
   Iron    3.25   10.99
```

10.10.2 csv format

Since the above code is working fine, therefore the contributor can add as many format as possible, just by adding a new class. In the below code, he added a 'csv format' for the printing,

Listing 10.7: csv format is added

```
# contributor.py
# user-2: additional features will be added to pythonic.py by the contributor

from abc import ABC, abstractmethod
import csv

from pythonic import Ring


# Abstract class for table-format
class TableFormat(object):
    """ Abastract class """

    @abstractmethod
    def heading(self, header): # must be implemented in child class
        pass

    @abstractmethod
    def row(self, row_data): # must be implemented in child class
        pass


# text format for table
```

(continues on next page)

(continued from previous page)

```
class TextFormat(TableFormat):
    """ Text format for table """

    def heading(self, header): # print headers
        for h in header:
            print("{:>7s}".format(h), end=' ')
        print()

    def row(self, row_data): # print rows
        for r in row_data:
            print("{:>7s}".format(r), end=' ')
        print()


# csv format for table
class CSVFormat(TableFormat):
    """ Text format for table """

    def heading(self, header): # print headers
        print(','.join(header))

    def row(self, row_data): # print rows
        print(",".join(row_data))

# this code is copied from datamine.py file and modified slightly
# to include the type of the data
def read_file(filename, types, mode='warn'):
    ''' read csv file and save data in the list '''

    # check for correct mode
    if mode not in ['warn', 'silent', 'stop']:
        raise ValueError("possible modes are 'warn', 'silent', 'stop'")

    ring_data = [] # create empty list to save data

    with open (filename, 'r') as f:
        rows =  csv.reader(f)
        header = next(rows) # skip the header

        # change the types of the columns
        for row in rows:
            try:
                # row[2] = float(row[2]) # radius
                # row[3] = float(row[3]) # price
                # row[4] = int(row[4]) # quantity

                # generalized conversion
                row = [d_type(val) for d_type, val in zip(types, row)]
            except ValueError as err: # process value error only
                if mode == 'warn':
                    print("Invalid data, row is skipped")
                    print('Row: {}, Reason : {}'.format(row_num, err))
                elif mode == 'silent':
                    pass # do nothing
                elif mode == 'stop':
                    raise # raise the exception
                continue

            # ring_data.append(tuple(row))

            # append data in list in the form of tuple
```

(continues on next page)

(continued from previous page)

```
            # row_dict = {
                    # 'date' : row[0],
                    # 'metal' : row[1],
                    # 'radius' : row[2],
                    # 'price' : row[3],
                    # 'quantity' : row[4]
                # }

            # row_dict = Ring(row[0], row[1], row[2], row[3], row[4])
            # # use below or above line
            row_dict = Ring(
                    date = row[0],
                    metal = row[1],
                    radius = row[2],
                    price = row[3],
                    quantity = row[4]
                )

            ring_data.append(row_dict)

    return ring_data


# table formatter
def print_table(list_name, col_name=['metal', 'radius'],
        out_format=TextFormat()): # note that class is passed here
    """ print the formatted output """

    # for c in col_name: # print header
        # print("{:>7s}".format(c), end=' ')
    # print() # print empty line
    # for l in list_name: # print values
        # for c in col_name:
            # print("{:>7s}".format(str(getattr(l, c))), end=' ')
        # print()

    # invoke class-method for printing heading
    out_format.heading(col_name) # class is passed to out_format
    for l in list_name:
        # store row in a list
        row_data = [str(getattr(l, c)) for c in col_name]
        out_format.row(row_data) # pass rows to class-method row()


def main():
    # list correct types of columns in csv file
    types =[str, str, float, float, int]

    # read file and save data in list
    list_data = read_file('price.csv', types)

    # # formatted output
    # print_table(list_data)
    # print()
    print_table(list_data, ['metal', 'radius', 'price'], out_format=TextFormat())

    print()
    print_table(list_data, ['metal', 'radius', 'price'], out_format=CSVFormat())


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
146     main()
```

Following is the output of above listing,

```
$ python contributor.py
  metal  radius   price
   Gold     5.5   80.99
 Silver    40.3     5.5
   Iron     9.2   14.29
   Gold     8.0   120.3
 Copper     4.1   70.25
   Iron    3.25   10.99

metal,radius,price
Gold,5.5,80.99
Silver,40.3,5.5
Iron,9.2,14.29
Gold,8.0,120.3
Copper,4.1,70.25
Iron,3.25,10.99
```

10.11 Advance inheritance

In this section, __init__ function of parent class is inherited by the child class. Also, a good usage of 'multiple inheritance' and 'class with one method' is shown.

10.11.1 Printing outputs to files

In Listing 10.7, the output can be printed on the screen only. There we requested the contributor, that it will be better if we can save the output in the files as well.

To add this feature, the contributor decided to add an __init__ function to add the print functionality. It is a good approach otherwise we need to add the 'output-file' logic in each format-class.

Below is the code for saving the data in file. Following are the functionality added to this design,

- Outputs can be saved in the files.
- CSVFormat gets this ability through inheriting the parent class __init__.
- TextFormat override the parent class __init__ to increase/decrease the width. Also, it uses 'super()' to inherit the parent class __init__ so that the output can be saved in the file.

Listing 10.8: Print data in file and screen

```
1  # contributor.py
2  # user-2: additional features will be added to pythonic.py by the contributor
3
4  import sys
5  import csv
6  from abc import ABC, abstractmethod
7
8  from pythonic import Ring
9
10
11 # Abstract class for table-format
12 class TableFormat(object):
13     """ Abastract class """
14
15     # print data to file or screen
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, out_file=None):
        if out_file == None:
            # stdout is the location where python prints the output
            out_file = sys.stdout
        self.out_file = out_file

    @abstractmethod
    def heading(self, header): # must be implemented in child class
        pass

    @abstractmethod
    def row(self, row_data): # must be implemented in child class
        pass


# text format for table
class TextFormat(TableFormat):
    """ Text format for table """

    # option for modifying width
    def __init__(self, width=7, out_file=None): # override init
        # inherit parent init as well to save data in file
        super().__init__(out_file)
        self.width = width

    def heading(self, header): # print headers
        for h in header:
            print("{0:>{1}s}".format(h, self.width),
                  end=' ', file=self.out_file)
        print(file=self.out_file)

    def row(self, row_data): # print rows
        for r in row_data:
            print("{0:>{1}s}".format(r, self.width),
                  end=' ', file=self.out_file)
        print(file=self.out_file)


# csv format for table
class CSVFormat(TableFormat):
    """ Text format for table """

    # init will be inherited from parent to save data in file

    def heading(self, header): # print headers
        print(','.join(header), file=self.out_file)

    def row(self, row_data): # print rows
        print(",".join(row_data), file=self.out_file)

# this code is copied from datamine.py file and modified slightly
# to include the type of the data
def read_file(filename, types, mode='warn'):
    ''' read csv file and save data in the list '''

    # check for correct mode
    if mode not in ['warn', 'silent', 'stop']:
        raise ValueError("possible modes are 'warn', 'silent', 'stop'")

    ring_data = [] # create empty list to save data

```

(continues on next page)

(continued from previous page)

```
    with open (filename, 'r') as f:
        rows =  csv.reader(f)
        header = next(rows) # skip the header

        # change the types of the columns
        for row in rows:
            try:
                # row[2] = float(row[2]) # radius
                # row[3] = float(row[3]) # price
                # row[4] = int(row[4]) # quantity

                # generalized conversion
                row = [d_type(val) for d_type, val in zip(types, row)]
            except ValueError as err: # process value error only
                if mode == 'warn':
                    print("Invalid data, row is skipped")
                    print('Row: {}, Reason : {}'.format(row_num, err))
                elif mode == 'silent':
                    pass # do nothing
                elif mode == 'stop':
                    raise # raise the exception
                continue

            # ring_data.append(tuple(row))

            # append data in list in the form of tuple
            # row_dict = {
                    # 'date' : row[0],
                    # 'metal' : row[1],
                    # 'radius' : row[2],
                    # 'price' : row[3],
                    # 'quantity' : row[4]
                # }

            # row_dict = Ring(row[0], row[1], row[2], row[3], row[4])
            # # use below or above line
            row_dict = Ring(
                    date = row[0],
                    metal = row[1],
                    radius = row[2],
                    price = row[3],
                    quantity = row[4]
                )

            ring_data.append(row_dict)

    return ring_data


# table formatter
def print_table(list_name, col_name=['metal', 'radius'],
        out_format=TextFormat(out_file=None)): # note that class is passed here
    """ print the formatted output """

    # for c in col_name: # print header
        # print("{:>7s}".format(c), end=' ')
    # print() # print empty line
    # for l in list_name: # print values
        # for c in col_name:
            # print("{:>7s}".format(str(getattr(l, c))), end=' ')
        # print()
```

(continues on next page)

(continued from previous page)

```
138
139     # invoke class-method for printing heading
140     out_format.heading(col_name) # class is passed to out_format
141     for l in list_name:
142         # store row in a list
143         row_data = [str(getattr(l, c)) for c in col_name]
144         out_format.row(row_data) # pass rows to class-method row()
145
146
147 def main():
148     # list correct types of columns in csv file
149     types =[str, str, float, float, int]
150
151     # read file and save data in list
152     list_data = read_file('price.csv', types)
153
154     # # formatted output
155     # print_table(list_data)
156     # print()
157     print_table(list_data, ['metal', 'radius', 'price'],
158             out_format=TextFormat())
159
160     # print in file
161     out_file = open("text_format.txt", "w") # open file in write mode
162     print_table(list_data, ['metal', 'radius', 'price'],
163             out_format=TextFormat(out_file=out_file))
164     out_file.close()
165
166     # print in file with width = 15
167     out_file = open("wide_text_format.txt", "w") # open file in write mode
168     print_table(list_data, ['metal', 'radius', 'price'],
169             out_format=TextFormat(width=15, out_file=out_file))
170     out_file.close()
171
172     print()
173     print_table(list_data, ['metal', 'radius', 'price'],
174             out_format=CSVFormat())
175     # print in file
176     out_file = open("csv_format.csv", "w") # open file in write mode
177     print_table(list_data, ['metal', 'radius', 'price'],
178             out_format=CSVFormat(out_file=out_file))
179     out_file.close()
180
181
182 if __name__ == '__main__':
183     main()
```

Following are the outputs of the above listing.

Note: The previous main function is also working fine. This check is very important to confirm that the codes of the exitsting user will not bread due to updating of the design.

```
$ python contributor.py

  metal  radius   price
   Gold     5.5   80.99
 Silver    40.3     5.5
   Iron     9.2   14.29
   Gold     8.0   120.3
```

(continues on next page)

(continued from previous page)

```
 Copper     4.1   70.25
   Iron    3.25   10.99

metal,radius,price
Gold,5.5,80.99
Silver,40.3,5.5
Iron,9.2,14.29
Gold,8.0,120.3
Copper,4.1,70.25
Iron,3.25,10.99
```

Below are the contents of the file generated by above listing,

```
$ cat text_format.txt

 metal  radius   price
  Gold     5.5   80.99
Silver    40.3     5.5
  Iron     9.2   14.29
  Gold     8.0   120.3
Copper     4.1   70.25
  Iron    3.25   10.99
```

```
$ cat wide_text_format.txt

 metal          radius          price
  Gold             5.5          80.99
Silver            40.3            5.5
  Iron             9.2          14.29
  Gold             8.0          120.3
Copper             4.1          70.25
  Iron            3.25          10.99
```

```
$ cat csv_format.csv
metal,radius,price
Gold,5.5,80.99
Silver,40.3,5.5
Iron,9.2,14.29
Gold,8.0,120.3
Copper,4.1,70.25
Iron,3.25,10.99
```

10.11.2 Mixin : multiple inheritance

10.11.3 Put quotes around data

Suppose, we want to put quotes around all the data in printed output. This can be accomplished as below,

```
>>> from contributor import *
>>> class QuoteData(TextFormat):
...     def row(self, row_data):
...             # put quotes around data
...             quoted_data = ['"{0}"'.format(r) for r in row_data]
...             super().row(quoted_data)
...
>>> types =[str, str, float, float, int]
>>> list_data = read_file('price.csv', types)
>>> print_table(list_data, ['metal', 'radius', 'price'], out_format=QuoteData())
```

(continues on next page)

(continued from previous page)

```
  metal  radius   price
 "Gold"   "5.5" "80.99"
"Silver"  "40.3"   "5.5"
 "Iron"   "9.2" "14.29"
 "Gold"   "8.0" "120.3"
"Copper"   "4.1" "70.25"
 "Iron"  "3.25" "10.99"
```

Note: The problem with above method is that it is applicable to class 'TextFormat' only (not to CSVFormat). This can be generalized using Mixin as shown below,

10.11.4 Put quotes using Mixin

We can use mix two class as shown below,

Important: Please note following points in the below code,

- In Python, inheritance follows two rule
  - Child is checked before Parent.
  - Parents are checked in order.
- The 'super()' is used in he class "QuoteData", which will call the __init__ function. But this class does not inherit from any other class (or inherits from Python-object class).
- The __init__ function function for the class "QuoteData" will be decided by the child class, i.e. the class MixinQuoteCSV is inheriting "QuoteData" and "CSVFormat".
- Since, the parents are checked in order, therefore "QuoteData" will be check first.
- Also, "Child is checked before Parent" i.e. child will decide the super() function for the parent. For example, super() function of "QuoteData" will call the __init__ function of "parent of child class (not it's own parent)", hence __init__ of CSVFormat will be invoked by the "super()" of QuoteData.
- The correct order of inheritance can be checked using 'help' as below. If we have super() in all classes. Then super() of MixinQuoteText will call the "QuoteData"; then super() of QuoteData will call the TextFormat and so on.

```
>>> from contributor import *
>>> help(MixinQuoteText)
class MixinQuoteText(QuoteData, TextFormat)
 |  Text format for table
 |
 |  Method resolution order:
 |      MixinQuoteText
 |      QuoteData
 |      TextFormat
 |      TableFormat
 |      builtins.object
```

Warning: It is not a good idea to define a class with one method. But this feature can be quite powerful in the case of inheritance, as shown in below example.

Listing 10.9: Adding quotes around printed data using Mixin

```
1 # contributor.py
2 # user-2: additional features will be added to pythonic.py by the contributor
3
```

(continues on next page)

(continued from previous page)

```
import sys
import csv
from abc import ABC, abstractmethod

from pythonic import Ring


# Abstract class for table-format
class TableFormat(object):
    """ Abastract class """

    # print data to file or screen
    def __init__(self, out_file=None):
        if out_file == None:
            # stdout is the location where python prints the output
            out_file = sys.stdout
        self.out_file = out_file

    @abstractmethod
    def heading(self, header): # must be implemented in child class
        pass

    @abstractmethod
    def row(self, row_data): # must be implemented in child class
        pass


# text format for table
class TextFormat(TableFormat):
    """ Text format for table """

    # option for modifying width
    def __init__(self, width=7, out_file=None): # override init
        # inherit parent init as well to save data in file
        super().__init__(out_file)
        self.width = width

    def heading(self, header): # print headers
        for h in header:
            print("{0:>{1}s}".format(h, self.width),
                    end=' ', file=self.out_file)
        print(file=self.out_file)

    def row(self, row_data): # print rows
        for r in row_data:
            print("{0:>{1}s}".format(r, self.width),
                    end=' ', file=self.out_file)
        print(file=self.out_file)


# csv format for table
class CSVFormat(TableFormat):
    """ Text format for table """

    # init will be inherited from parent to save data in file

    def heading(self, header): # print headers
        print(','.join(header), file=self.out_file)

    def row(self, row_data): # print rows
        print(",".join(row_data), file=self.out_file)
```

(continues on next page)

(continued from previous page)

```
65
66
67  # Put quotes around data : This class will be used with other class in Mixin
68  class QuoteData(object):
69      def row(self, row_data):
70          quoted_data = ['"{0}"'.format(r) for r in row_data]
71          super().row(quoted_data) # nature of super() is decided by child class
72
73  # Mixin : QuoteData and CSVFormat
74  # this will decide the nature of super() in QuoteData
75  class MixinQuoteCSV(QuoteData, CSVFormat):
76      pass
77
78  # Mixin : QuoteData and TextFormat
79  class MixinQuoteText(QuoteData, TextFormat):
80      pass
81
82
83  # this code is copied from datamine.py file and modified slightly
84  # to include the type of the data
85  def read_file(filename, types, mode='warn'):
86      ''' read csv file and save data in the list '''
87
88      # check for correct mode
89      if mode not in ['warn', 'silent', 'stop']:
90          raise ValueError("possible modes are 'warn', 'silent', 'stop'")
91
92      ring_data = [] # create empty list to save data
93
94      with open (filename, 'r') as f:
95          rows =  csv.reader(f)
96          header = next(rows) # skip the header
97
98          # change the types of the columns
99          for row in rows:
100             try:
101                 # row[2] = float(row[2]) # radius
102                 # row[3] = float(row[3]) # price
103                 # row[4] = int(row[4]) # quantity
104
105                 # generalized conversion
106                 row = [d_type(val) for d_type, val in zip(types, row)]
107             except ValueError as err: # process value error only
108                 if mode == 'warn':
109                     print("Invalid data, row is skipped")
110                     print('Row: {}, Reason : {}'.format(row_num, err))
111                 elif mode == 'silent':
112                     pass # do nothing
113                 elif mode == 'stop':
114                     raise # raise the exception
115                 continue
116
117             # ring_data.append(tuple(row))
118
119             # append data in list in the form of tuple
120             # row_dict = {
121                     # 'date' : row[0],
122                     # 'metal' : row[1],
123                     # 'radius' : row[2],
124                     # 'price' : row[3],
125                     # 'quantity' : row[4]
```

(continues on next page)

(continued from previous page)

```
                # }

            # row_dict = Ring(row[0], row[1], row[2], row[3], row[4])
            # # use below or above line
            row_dict = Ring(
                    date = row[0],
                    metal = row[1],
                    radius = row[2],
                    price = row[3],
                    quantity = row[4]
                )

            ring_data.append(row_dict)

    return ring_data


# table formatter
def print_table(list_name, col_name=['metal', 'radius'],
        out_format=TextFormat(out_file=None)): # note that class is passed here
    """ print the formatted output """

    # for c in col_name: # print header
        # print("{:>7s}".format(c), end=' ')
    # print() # print empty line
    # for l in list_name: # print values
        # for c in col_name:
            # print("{:>7s}".format(str(getattr(l, c))), end=' ')
        # print()

    # invoke class-method for printing heading
    out_format.heading(col_name) # class is passed to out_format
    for l in list_name:
        # store row in a list
        row_data = [str(getattr(l, c)) for c in col_name]
        out_format.row(row_data) # pass rows to class-method row()


def main():
    # list correct types of columns in csv file
    types =[str, str, float, float, int]

    # read file and save data in list
    list_data = read_file('price.csv', types)

    # # formatted output
    # print_table(list_data)
    # print()
    # print_table(list_data, ['metal', 'radius', 'price'],
            # out_format=TextFormat())

    # print in file
    # out_file = open("text_format.txt", "w") # open file in write mode
    # print_table(list_data, ['metal', 'radius', 'price'],
            # out_format=TextFormat(out_file=out_file))
    # out_file.close()

    # print in file with width = 15
    # out_file = open("wide_text_format.txt", "w") # open file in write mode
    # print_table(list_data, ['metal', 'radius', 'price'],
            # out_format=TextFormat(width=15, out_file=out_file))
```

(continues on next page)

(continued from previous page)

```
    # out_file.close()

    # print()
    # print_table(list_data, ['metal', 'radius', 'price'],
            # out_format=CSVFormat())
    # # print in file
    # out_file = open("csv_format.csv", "w") # open file in write mode
    # print_table(list_data, ['metal', 'radius', 'price'],
            # out_format=CSVFormat(out_file=out_file))
    # out_file.close()


    print_table(list_data, ['metal', 'radius', 'price'],
            out_format=MixinQuoteText())

    print()
    print_table(list_data, ['metal', 'radius', 'price'],
            out_format=MixinQuoteCSV())

if __name__ == '__main__':
    main()
```

10.12 Conclusion

In this chapter, we saw some of the OOPs feature of Python especially Inheritance. More examples of Inheritance are included in Chapter 13. We discussed some of the differences between Python and other programming language. In next chapter, we will discuss '@property' and 'descriptor' etc. Also, in the next chapter, we will discuss some more differences between Python and other programming language.

Chapter 11

Methods and @property

11.1 Introduction

In this chapter, we will understand the usage of 'methods', '@property' and 'descriptor'.

11.2 Methods, staticmethod and classmethod

In previous chapters, we saw the examples of methods, e.g. 'area' and 'cost' in Listing 10.1, inside the class without any decorator with them. The 'decorators' adds the additional functionality to the methods, and will be discussed in details in Chapter 12.

In this section, we will quickly review the different types of methods with an example. Then these methods will be used with our previous examples.

In the below code, two decorators are used with the methods inside the class i.e. 'staticmethod' and 'classmethod'. Please see the comments and notice: how do the different methods use different values of x for adding two numbers,

---

Note: We can observe the following differences in these three methods from the below code,

- **method** : it uses the instance variable (self.x) for addition, which is set by __init__ function.
- **classmethod** : it uses class variable for addition.
- **staticmethod** : it uses the value of x which is defined in main program (i.e. outside the class). If x = 20 is not defined, then NameError will be generated.

```
# mehodsEx.py

# below x will be used by static method
# if we do not define it, the staticmethod will generate error.
x = 20

class Add(object):
    x = 9  # class variable

    def __init__(self, x):
        self.x = x  # instance variable

    def addMethod(self, y):
        print("method:", self.x + y)

    @classmethod
    # as convention, cls must be used for classmethod, instead of self
```

(continues on next page)

(continued from previous page)

```
    def addClass(self, y):
        print("classmethod:", self.x + y)

    @staticmethod
    def addStatic(y):
        print("staticmethod:", x + y)



def main():
    # method
    m  = Add(x=4) # or m = Add(4)
    # for method, above x = 4, will be used for addition
    m.addMethod(10)  # method :  14

    # classmethod
    c = Add(4)
    # for class method, class variable x = 9, will be used for addition
    c.addClass(10)  # clasmethod : 19

    # for static method, x=20 (at the top of file), will be used for addition
    s = Add(4)
    s.addStatic(10)  # staticmethod : 30

if __name__ == '__main__':
    main()
```

Below is the output for above code,

```
$ python methodEx.py

method: 14
classmethod: 19
staticmethod: 30
```

11.3 Research organization

In previous chapter, we had two users who are modifying the class attributes. First was the mathematician (mathematician.py), who is modifying the 'radius' in Listing 10.4. And other is the box company, who were modifying the method 'perimeter' in Listing 10.5. Lastly, we had a contributor who is creating a 'table formatter' in Listing 10.9 for displaying the output nicely.

> Now, we have another research organization, who were doing their research analysis based on the diameters (not using radius), therefore they want to initialize the class-objects based on the diameter directly.

Note: If we change the the 'radius' to 'diameter' in the __init__ function of class Ring, then we need to modify the 'area' and 'cost' methods as well. Also, the codes of other users (mathematician and box company) will break. But we can solve this problem using 'classmethod' as shown next.

11.3.1 Multiple constructor using 'classmethod'

The 'classmethod' is a very useful tools to create the multiple constructor as shown in the below listing.

```python
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
                 price=5.0, quantity=5):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity


    # Multiple constructor
    # below constructor is added for the 'research organization' who are
    # doing their work based on diameters,
    @classmethod
    def diameter_init(cls, diameter):
        radius = diameter/2;  # change diameter to radius
        return cls(radius) # return radius

    def cost(self):
        return self.price * self.quantity

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

    def perimeter(self):
        return 2 * math.pi * self.radius

def main():
    print("Center of the Ring is at:", Ring.center) # modify class variable
    r = Ring(price=8) # modify only price
    print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))
    print("Radius:{0}, Perimeter:{1:0.2f}".format(r.radius, r.perimeter()))

    print() # check the new constructor 'diameter_init'
    d = Ring.diameter_init(diameter=10)
```

(continues on next page)

(continued from previous page)

```
    print("Radius:{0}, Perimeter:{1:0.2f}".format(d.radius, d.perimeter()))


if __name__ == '__main__':
    main()
```

Following are the outputs for above code,

```
$ python pythonic.py

Center of the Ring is at: 0.0
Radius:5.0, Cost:40
Radius:5.0, Perimeter:31.42

Radius:5.0, Perimeter:31.42
```

11.3.2 Additional methods using 'staticmethod'

Now, the research organization wants one more features in the class, i.e. Meter to Centimeter conversion. Note that this feature has no relation with the other methods in the class. In the other words, the output of Meter to Centimeter conversion will not be used anywhere in the class, but it's handy for the organization to have this feature in the package. In such cases, we can use static methods.

Note: There is no point to create a 'meter_cm' method as 'meter_cm(self, meter)' and store it in the dictionary, as we are not going to use it anywhere in the class. We added this to meet the client's requirement only, hence it should be added as simple definition (not as methods), which can be done using @staticmethod. In the other words, static methods are used to attached functions to classes.

```
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
                 price=5.0, quantity=5):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """
```

(continues on next page)

(continued from previous page)

```
30
31          self.date = date
32          self.metal = metal
33          self.radius = radius
34          self.price = price
35          self.quantity = quantity
36
37
38      # Multiple constructor
39      # below constructor is added for the 'research organization' who are
40      # doing their work based on diameters,
41      @classmethod
42      def diameter_init(cls, diameter):
43          radius = diameter/2;  # change diameter to radius
44          return cls(radius) # return radius
45
46      @staticmethod  # meter to centimeter conversion
47      def meter_cm(meter):
48          return(100*meter)
49
50
51      def cost(self):
52          return self.price * self.quantity
53
54      def area(self):
55          return math.pi * self.radius**2
56
57      def perimeter(self):
58          return 2 * math.pi * self.radius
59
60  def main():
61      print("Center of the Ring is at:", Ring.center) # modify class variable
62      r = Ring(price=8) # modify only price
63      print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))
64      print("Radius:{0}, Perimeter:{1:0.2f}".format(r.radius, r.perimeter()))
65
66      print() # check the new constructor 'diameter_init'
67      d = Ring.diameter_init(diameter=10)
68      print("Radius:{0}, Perimeter:{1:0.2f}".format(d.radius, d.perimeter()))
69      m = 10 # 10 meter
70      print("{0} meter = {1} centimeter".format(m, d.meter_cm(m)))
71
72  if __name__ == '__main__':
73      main()
```

Following is the output of above code,

```
$ python pythonic.py
Center of the Ring is at: 0.0
Radius:5.0, Cost:40
Radius:5.0, Perimeter:31.42

Radius:5.0, Perimeter:31.42
10 meter = 1000 centimeter
```

11.4 Micro-managing

Micro-managing is the method, where client tells us the ways in which the logic should be implemented. Lets understand it by an example.

Note: Now the research organization wants that the 'area' should be calculated based on the 'perimeter'. More specifically, they will provide the 'diameter', then we need to calculate the 'perimeter'; then based on the 'perimeter', calculate the 'radius'. And finally calculate the 'area' based on this new radius. This may look silly, but if we look closely, the radius will change slightly during this conversion process; as there will be some difference between (diameter/2) and (math.pi*diameter)/(2*math.pi). The reason for difference is: on computer we do not cancel the math.pi at numerator with math.pi at denominator, but solve the numerator first and then divide by the denominator, and hence get some rounding errors.

11.4.1 Wrong Solution

The solution to this problem may look pretty simple, i.e. modify the 'area' method in the following way,

```
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
                 price=5.0, quantity=5):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity


    # Multiple constructor
    # below constructor is added for the 'research organization' who are
    # doing their work based on diameters,
    @classmethod
    def diameter_init(cls, diameter):
        radius = diameter/2;  # change diameter to radius
        return cls(radius) # return radius

    @staticmethod  # meter to centimeter conversion
```

(continues on next page)

(continued from previous page)

```
47      def meter_cm(meter):
48          return(100*meter)
49
50      def cost(self):
51          return self.price * self.quantity
52
53      def area(self):
54          # return math.pi * self.radius**2
55          p = self.perimeter() # calculate perimeter
56          r = p / ( 2 * math.pi)
57          return math.pi * r**2
58
59      def perimeter(self):
60          return 2 * math.pi * self.radius
61
62  def main():
63      # print("Center of the Ring is at:", Ring.center) # modify class variable
64      # r = Ring(price=8) # modify only price
65      # print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))
66      # print("Radius:{0}, Perimeter:{1:0.2f}".format(r.radius, r.perimeter()))
67
68      print() # check the new constructor 'diameter_init'
69      d = Ring.diameter_init(diameter=10)
70      print("Radius:{0}, Perimeter:{1:0.2f}, Area:{2:0.2f}".format(
71          d.radius, d.perimeter(), d.area()))
72      # print("Radius:{0}, Perimeter:{1:0.2f}".format(d.radius, d.perimeter()))
73      # m = 10 # 10 meter
74      # print("{0} meter = {1} centimeter".format(m, d.meter_cm(m)))
75
76  if __name__ == '__main__':
77      main()
```

- If we run the above code, we will have the following results, which is exactly right.

```
$ python pythonic.py

Radius:5.0, Perimeter:31.42, Area:78.54
```

Error: Above solution looks pretty simple, but it will break the code in 'box.py in Listing 10.5', as it is modifying the perimeter by a factor of 2.0 by overriding the perimeter method. Therefore, for calculating the area in box.py file, python interpretor will use the child class method 'perimeter' for getting the value of perimeter. Hence, the area will be calculated based on the 'modified parameter', not based on actual parameter.

- In box.py, the area will be wrongly calculated, because it is modifying the perimeter for their usage. Now, radius will be calculated by new perimeter (as they override the perimeter method) and finally area will be modified due to change in radius values, as shown below,

```
>>> from box import *
>>> b = Box.diameter_init(10)
>>> b.radius
5.0
>>> b.perimeter()  # doubled value of perimeter (desired)
62.83185307179586
>>> b.area()  # wrong value of area, see next for correct result
314.1592653589793
>>>
>>> import math
>>> math.pi * b.radius**2  # desired radius
78.53981633974483
>>> math.pi * (b.perimeter()/(2*math.pi))**2  # calculated radius
314.1592653589793
```

11.4.2 Correct solution

Note: The above problem can be resolved by creating a local copy of perimeter in the class. In this way, the child class method can not override the local-copy of parent class method. For this, we need to modify the code as below,

```
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
                 price=5.0, quantity=5):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity


    # Multiple constructor
    # below constructor is added for the 'research organization' who are
    # doing their work based on diameters,
    @classmethod
    def diameter_init(cls, diameter):
        radius = diameter/2;  # change diameter to radius
        return cls(radius) # return radius

    @staticmethod  # meter to centimeter conversion
    def meter_cm(meter):
        return(100*meter)

    def cost(self):
```

(continues on next page)

(continued from previous page)

```
        return self.price * self.quantity

    def area(self):
        # return math.pi * self.radius**2
        # p = self.perimeter() # wrong way to calculate perimeter
        p = self._perimeter()  # use local copy of perimeter()
        r = p / ( 2 * math.pi)
        return math.pi * r**2

    def perimeter(self):
        return 2 * math.pi * self.radius

    # local copy can be created in the lines after the actual method
    _perimeter = perimeter # make a local copy of perimeter

def main():
    # print("Center of the Ring is at:", Ring.center) # modify class variable
    # r = Ring(price=8) # modify only price
    # print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))
    # print("Radius:{0}, Perimeter:{1:0.2f}".format(r.radius, r.perimeter()))

    print() # check the new constructor 'diameter_init'
    d = Ring.diameter_init(diameter=10)
    print("Radius:{0}, Perimeter:{1:0.2f}, Area:{2:0.2f}".format(
        d.radius, d.perimeter(), d.area()))
    # print("Radius:{0}, Perimeter:{1:0.2f}".format(d.radius, d.perimeter()))
    # m = 10 # 10 meter
    # print("{0} meter = {1} centimeter".format(m, d.meter_cm(m)))

if __name__ == '__main__':
    main()
```

Below is the result for above listing,

```
$ python pythonic.py

Radius:5.0, Perimeter:31.42, Area:78.54
```

Now, we will get the correct results for the 'box.py' as well, as shown below,

```
>>> from box import *
>>> b = Box.diameter_init(10)
>>> b.radius
5.0
>>> b.perimeter()  # doubled value of perimeter (desired)
62.83185307179586
>>> b.area()  # desired area
78.53981633974483
```

11.5 Private attributes are not for privatizing the attributes

In Section 10.7, we saw that there is no concept of data-hiding in Python; and the attributes can be directly accessed by the the clients.

Note: There is a misconception that the double underscore ('__') are used for data hiding in Python. In this section, we will see the correct usage of '__' in Python.

11.5.1 Same local copy in child and parent class

In previous section, we made a local copy of method 'perimeter' as '_perimeter'. In this way, we resolve the problem of overriding the parent class method.

But, if the child class (i.e. box.py) makes a local copy the perimeter as well with the same name i.e. _perimeter, then we will again have the same problem, as shown below,

```
# box.py
# user-3 : creating boxes for the ring

from pythonic import Ring

class Box(Ring):
    """ Modified perimeter for creating the box """

    def perimeter(self): # override the method 'perimeter'
        # perimeter is increased 2.0 times
        return Ring.perimeter(self) * 2.0

    _perimeter = perimeter

def main():
    b = Box(radius=8) # pass radius = 8
    print("Radius:", b.radius)
    print("Modified perimeter: %0.2f" % b.perimeter()) # (2*pi*radius) * 2

if __name__ == '__main__':
    main()
```

Now, we will have same problem as before,

```
>>> from box import *
>>> b = Box.diameter_init(10)
>>> b.area()  # wrong value of area again,
314.1592653589793
```

11.5.2 Use '__perimeter' instead of '_perimeter' to solve the problem

The above problem can be solved using double underscore before the perimeter instead of one underscore, as shown below,

- First replace the _perimeter with __ perimeter in the 'pythonic.py' as below,

```
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
```

(continues on next page)

(continued from previous page)

```
                price=5.0, quantity=5):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity


    # Multiple constructor
    # below constructor is added for the 'research organization' who are
    # doing their work based on diameters,
    @classmethod
    def diameter_init(cls, diameter):
        radius = diameter/2;  # change diameter to radius
        return cls(radius) # return radius

    @staticmethod  # meter to centimeter conversion
    def meter_cm(meter):
        return(100*meter)

    def cost(self):
        return self.price * self.quantity

    def area(self):
        # return math.pi * self.radius**2
        # p = self.perimeter() # wrong way to calculate perimeter
        p = self.__perimeter()  # use local copy of perimeter()
        r = p / ( 2 * math.pi)
        return math.pi * r**2

    def perimeter(self):
        return 2 * math.pi * self.radius

    # local copy can be created in the lines after the actual method
    __perimeter = perimeter # make a local copy of perimeter

def main():
    # print("Center of the Ring is at:", Ring.center) # modify class variable
    # r = Ring(price=8) # modify only price
    # print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))
    # print("Radius:{0}, Perimeter:{1:0.2f}".format(r.radius, r.perimeter()))

    print() # check the new constructor 'diameter_init'
    d = Ring.diameter_init(diameter=10)
    print("Radius:{0}, Perimeter:{1:0.2f}, Area:{2:0.2f}".format(
        d.radius, d.perimeter(), d.area()))
    # print("Radius:{0}, Perimeter:{1:0.2f}".format(d.radius, d.perimeter()))
    # m = 10 # 10 meter
    # print("{0} meter = {1} centimeter".format(m, d.meter_cm(m)))
```

(continues on next page)

(continued from previous page)

```
79
80 if __name__ == '__main__':
81     main()
```

- Next replace the _perimeter with __ perimeter in the 'box.py' as below,

```
1  # box.py
2  # user-3 : creating boxes for the ring
3
4  from pythonic import Ring
5
6  class Box(Ring):
7      """ Modified perimeter for creating the box """
8
9      def perimeter(self): # override the method 'perimeter'
10         # perimeter is increased 2.0 times
11         return Ring.perimeter(self) * 2.0
12
13     __perimeter = perimeter
14
15 def main():
16     b = Box(radius=8) # pass radius = 8
17     print("Radius:", b.radius)
18     print("Modified perimeter: %0.2f" % b.perimeter()) # (2*pi*radius) * 2
19
20 if __name__ == '__main__':
21     main()
```

Important: _ _ is not designed for making the attribute private, but for renaming the attribute with class name to avoid conflicts due to same name as we see in above example, where using same name i.e. _perimeter in both parent and child class resulted in the wrong answer.

If we use _ _perimeter instead of _perimeter, then _ _perimeter will be renamed as _ClassName__perimeter. Therefore, even parent and child class uses the same name with two underscore, the actual name will be differed because python interpretor will add the ClassName before those names, which will make it different from each other.

- Now, we have same name in both the files i.e. __perimeter, but problem will no longer occur as the 'class-names' will be added by the Python before __perimeter. Below is the result for both the python files,

```
$ python pythonic.py

Radius:5.0, Perimeter:31.42, Area:78.54
```

```
>>> from box import *
>>> b = Box.diameter_init(10)
>>> b.area()
78.53981633974483
```

Note: There is no notion of private attributes in Python. All the objects are exposed to users. * But, single underscore before the method or variable indicates that the attribute should not be access directly. * _ _ is designed for freedom not for privacy as we saw in this section.

11.6 @property

Now, we will see the usage of @property in Python.

11.6.1 Managing attributes

Currently, we are not checking the correct types of inputs in our design. Let's see our 'pythonic.py' file again.

```
>>> from pythonic import *
>>> r = Ring()
>>> r.radius
5.0
>>> r.radius = "Meher"
>>> 2 * r.radius
'MeherMeher'
```

Note that in the above code, the string is saved in the radius; and the multiplication is performed on the string. We do not want this behavior, therefore we need to perform some checks before saving data in the dictionary, which can be done using @property, as shown below,

---

Important:

- @property decorator allows . operator to call the function. Here, self.radius will call the method radius, which returns the self._radius. Hence, _radius is stored in the dictionary instead of dict.
- If we use 'return self.radius' instead of 'return self._radius', then the @property will result in infinite loop as self.property will call the property method, which will return self.property, which results in calling the @property again.
- Further, we can use @property for type checking, validation or performing various operations by writing codes in the setter method e.g. change the date to todays date etc.

```
1  # pythonic.py
2
3  import math
4  import time
5
6  class Ring(object):
7      """ Here we will see the actual logic behind various pieces of Python
8      language e.g. instances, variables, method and @property etc.
9      Also, we will see the combine usage of these pieces to complete a
10     design with Agile methodology.
11     """
12
13     # class variables
14     date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
15     center = 0.0 # center of the ring
16
17     def __init__(self, date=date, metal="Copper", radius=5.0,
18                  price=5.0, quantity=5):
19         """ init is not the constructor, but the initializer which
20         initialize the instance variable
21
22         self : is the instance
23
24         __init__ takes the instance 'self' and populates it with the radius,
25         metal, date etc. and store in a dictionary.
26
27         self.radius, self.metal etc. are the instance variable which
28         must be unique.
```

(continues on next page)

(continued from previous page)

```
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity

    @property
    # method-name should be same as attribute i.e. 'radius' here
    def radius(self):
        return self._radius # _radius can be changed with other name

    @radius.setter
    def radius(self, val):
        # 'val' should be float or int
        if not isinstance(val, (float, int)):
            raise TypeError("Expected: float or int")
        self._radius = float(val)

    # Multiple constructor
    # below constructor is added for the 'research organization' who are
    # doing their work based on diameters,
    @classmethod
    def diameter_init(cls, diameter):
        radius = diameter/2;  # change diameter to radius
        return cls(radius) # return radius

    @staticmethod  # meter to centimeter conversion
    def meter_cm(meter):
        return(100*meter)

    def cost(self):
        return self.price * self.quantity

    def area(self):
        # return math.pi * self.radius**2
        # p = self.perimeter() # wrong way to calculate perimeter
        p = self.__perimeter()  # use local copy of perimeter()
        r = p / ( 2 * math.pi)
        return math.pi * r**2

    def perimeter(self):
        return 2 * math.pi * self.radius

    # local copy can be created in the lines after the actual method
    __perimeter = perimeter # make a local copy of perimeter

def main():
    # print("Center of the Ring is at:", Ring.center) # modify class variable
    # r = Ring(price=8) # modify only price
    # print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))
    # print("Radius:{0}, Perimeter:{1:0.2f}".format(r.radius, r.perimeter()))

    print() # check the new constructor 'diameter_init'
    d = Ring.diameter_init(diameter=10)
    print("Radius:{0}, Perimeter:{1:0.2f}, Area:{2:0.2f}".format(
        d.radius, d.perimeter(), d.area()))
    # print("Radius:{0}, Perimeter:{1:0.2f}".format(d.radius, d.perimeter()))
    # m = 10 # 10 meter
    # print("{0} meter = {1} centimeter".format(m, d.meter_cm(m)))
```

(continues on next page)

(continued from previous page)

```
90
91  if __name__ == '__main__':
92      main()
```

- Below is the results for above code,

```
>>> from pythonic import *
>>> r = Ring()
>>> r.radius
5.0
>>> r.radius = 1
>>> r.radius
1.0
>>> r.radius = 3.0
>>> r.radius
3.0
>>> r.radius = "Meher"
Traceback (most recent call last): [...]
TypeError: Expected: float or int
```

- Below is the dictionary of the object 'r' of code. Note that, the 'radius' does not exist there anymore, but this will not break the codes of other users, due to following reason.

Note: @property is used to convert the attribute access to method access.

In the other words, the radius will be removed from instance variable list after defining the 'property' as in above code. Now, it can not be used anymore. But, @property decorator will convert the attribute access to method access, i.e. the dot operator will check for methods with @property as well. In this way, the code of other user will not break.

```
>>> r.__dict__
{'date': '2017-11-01', 'metal': 'Copper', '_radius': 3.0, 'price': 5.0, 'quantity': 5}
```

11.6.2 Calling method as attribute

If a method is decorated with @property then it can be called as 'attribute' using operator, but then it can not be called as attribute, as shown in below example,

```
>>> class PropertEx(object):
...     def mes_1(self):
...             print("hello msg_1")
...
...     @property
...     def mes_2(self):  # can be called as attribute only
...             print("hello msg_2")
...
>>> p = PropertEx()
>>> p.mes_1()
hello msg_1
>>> p.mes_2
hello msg_2
>>> p.mes_2()  # can not be called as method
hello msg_2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

11.6.3 Requirement from research organization

Now, we get another rule from the research organization as below,

- we do not want to store radius as instance variable,
- instead convert radius into diameter and save it as instance variable in the dictionary.

Note: This condition will raise following two problems,

- Main problem here is that, other users already started using this class and have access to attribute 'radius'. Now, if we replace the key 'radius' with 'diameter', then their code will break immediately.
- Also, we need to update all our code as everything is calculated based on radius; and we are going to remove these attribute from diameter.

This is the main reason for hiding attributes in java and c++, as these languages have no easy solution for this. Hence, these languages use getter and setter method.

But in python this problem can be solved using @property. @property is used to convert the attribute access to method access.

In the other words, as radius will be removed from instance variable list after meeting the need of the organization, therefore it can not be used anymore. But, @property decorator will convert the attribute access to method access, i.e. the dot operator will check for methods with @property as well. In this way, the codes of other users will not break, as shown next.

```
# pythonic.py

import math
import time

class Ring(object):
    """ Here we will see the actual logic behind various pieces of Python
    language e.g. instances, variables, method and @property etc.
    Also, we will see the combine usage of these pieces to complete a
    design with Agile methodology.
    """

    # class variables
    date = time.strftime("%Y-%m-%d", time.gmtime()) # today's date "YYYY-mm-dd"
    center = 0.0 # center of the ring

    def __init__(self, date=date, metal="Copper", radius=5.0,
                 price=5.0, quantity=5):
        """ init is not the constructor, but the initializer which
        initialize the instance variable

        self : is the instance

        __init__ takes the instance 'self' and populates it with the radius,
        metal, date etc. and store in a dictionary.

        self.radius, self.metal etc. are the instance variable which
        must be unique.
        """

        self.date = date
        self.metal = metal
        self.radius = radius
        self.price = price
        self.quantity = quantity

```

(continues on next page)

(continued from previous page)

```
    @property
    # method-name should be same as attribute i.e. 'radius' here
    def radius(self):
        # return self._radius # _radius can be changed with other name
        return self.diameter/2 # _radius can be changed with other name

    @radius.setter
    def radius(self, val):
        # 'val' should be float or int
        if not isinstance(val, (float, int)):
            raise TypeError("Expected: float or int")
        # self._radius = float(val)
        self.diameter = 2 * float(val)

    # Multiple constructor
    # below constructor is added for the 'research organization' who are
    # doing their work based on diameters,
    @classmethod
    def diameter_init(cls, diameter):
        radius = diameter/2;  # change diameter to radius
        return cls(radius) # return radius

    @staticmethod  # meter to centimeter conversion
    def meter_cm(meter):
        return(100*meter)

    def cost(self):
        return self.price * self.quantity

    def area(self):
        # return math.pi * self.radius**2
        # p = self.perimeter() # wrong way to calculate perimeter
        p = self.__perimeter()  # use local copy of perimeter()
        r = p / ( 2 * math.pi)
        return math.pi * r**2

    def perimeter(self):
        return 2 * math.pi * self.radius

    # local copy can be created in the lines after the actual method
    __perimeter = perimeter # make a local copy of perimeter

def main():
    # print("Center of the Ring is at:", Ring.center) # modify class variable
    # r = Ring(price=8) # modify only price
    # print("Radius:{0}, Cost:{1}".format(r.radius, r.cost()))
    # print("Radius:{0}, Perimeter:{1:0.2f}".format(r.radius, r.perimeter()))

    print() # check the new constructor 'diameter_init'
    d = Ring.diameter_init(diameter=10)
    print("Radius:{0}, Perimeter:{1:0.2f}, Area:{2:0.2f}".format(
        d.radius, d.perimeter(), d.area()))
    # print("Radius:{0}, Perimeter:{1:0.2f}".format(d.radius, d.perimeter()))
    # m = 10 # 10 meter
    # print("{0} meter = {1} centimeter".format(m, d.meter_cm(m)))

if __name__ == '__main__':
    main()
```

- Lets check the output of this file first, which is working fine as below. Note that the dictionary, now contains the 'diameter' instead of 'radius'

```
>>> from pythonic import *
>>> r = Ring()
>>> r.__dict__  # dictionary contains 'diameter' not 'radius'
{'date': '2017-11-01', 'metal': 'Copper',
'diameter': 10.0, 'price': 5.0, 'quantity': 5}
>>>
>>> r.radius  # radius is still accessbile
5.0
>>> r.area()  # area is working fine
78.53981633974483
>>> r.diameter # diameter is accessbile
10.0
```

- Next verify the output for the 'box.py' file again.

```
>>> from box import *
>>> b = Box.diameter_init(10)
>>> b.area()  # area is working fine
78.53981633974483
>>> b.__dict__
{'date': 5.0, 'metal': 'Copper', 'diameter': 10.0, 'price': 5.0, 'quantity': 5}
>>>
```

Chapter 12

Decorator and Descriptors

12.1 Decorators

Decorator is a function that creates a wrapper around another function. This wrapper adds some additional functionality to existing code. In this tutorial, various types of decorators are discussed.

12.1.1 Function inside the function and Decorator

Following is the example of function inside the function.

```
1  # funcEx.py
2
3  def addOne(myFunc):
4      def addOneInside(x):
5          print("adding One")
6          return myFunc(x) + 1
7      return addOneInside
8
9  def subThree(x):
10     return x - 3
11
12 result = addOne(subThree)
13
14 print(subThree(5))
15 print(result(5))
16
17 # outputs
18 # 2
19 # adding One
20 # 3
```

Above code works as follows,

- Function 'subThree' is defined at lines 9-10, which subtract the given number with 3.
- Function 'addOne' (Line 3) has one argument i.e. myFunc, which indicates that 'addOne' takes the function as input. Since, subThree function has only one input argument, therefore one argument is set in the function 'addOneInside' (Line 4); which is used in return statement (Line 6). Also, "adding One" is printed before returning the value (Line 5).
- In line 12, return value of addOne (i.e. function 'addOneInside') is stored in 'result'. Hence, 'result' is a function which takes one input.
- Lastly, values are printed at line 13 and 14. Note that "adding One" is printed by the result(5) and value is incremented by one i.e. 2 to 3.

Another nice way of writing above code is shown below. Here (*args and **kwargs) are used, which takes all the arguments and keyword arguments of the function.

```
1  # funcEx.py
2
3  def addOne(myFunc):
4      def addOneInside(*args, **kwargs):
5          print("adding One")
6          return myFunc(*args, **kwargs) + 1
7      return addOneInside
8
9  def subThree(x):
10     return x - 3
11
12 result = addOne(subThree)
13
14 print(subThree(5))
15 print(result(5))
16
17 # outputs
18 # 2
19 # adding One
20 # 3
```

Now, in the below code, the return value of addOne is stored in the 'subThree' function itself (Line 12),

```
1  # funcEx.py
2
3  def addOne(myFunc):
4      def addOneInside(*args, **kwargs):
5          print("adding One")
6          return myFunc(*args, **kwargs) + 1
7      return addOneInside
8
9  def subThree(x):
10     return x - 3
11
12 subThree = addOne(subThree)
13
14 print(subThree(5))
15 # outputs
16 # adding One
17 # 3
```

Lastly, in Python, the line 12 in above code, can be replaced by using decorator, as shown in Line 9 of below code,

```
1  # funcEx.py
2
3  def addOne(myFunc):
4      def addOneInside(*args, **kwargs):
5          print("adding One")
6          return myFunc(*args, **kwargs) + 1
7      return addOneInside
8
9  @addOne
10 def subThree(x):
11     return x - 3
12
13 print(subThree(5))
14 # outputs
15 # adding One
16 # 3
```

In this section, we saw the basics of the decorator, which we will be used in this tutorial.

12.1.2 Decorator without arguments

In following code, Decorator takes a function as the input and print the name of the function and return the function.

```
# debugEx.py

def printName(func):
    # func is the function to be wrapped
    def pn(*args, **kwargs):
        print(func.__name__)
        return func(*args, **kwargs)
    return pn
```

Next, put the printName function as decorator in the mathEx.py file as below,

```
# mathEx.py

from debugEx import printName

@printName
def add2Num(x, y):
    # add two numbers
    # print("add")
    return(x+y)

print(add2Num(2, 4))
help(add2Num)
```

Finally, execute the code and the name of each function will be printed before calculation as shown below,

```
$ python mathEx.py
add
6

Help on function pn in module debugEx:

pn(*args, **kwargs)
    # func is the function to be wrapped
```

Important: Decorator brings all the debugging code at one places. Now we can add more debugging features to 'debugEx.py' file and all the changes will be applied immediately to all the functions.

Warning: Decorators remove the help features of the function along with name etc. Therefore, we need to fix it using functools as shown next.

Rewrite the decorator using wraps function in functools as below,

```
# debugEx.py

from functools import wraps

def printName(func):
    # func is the function to be wrapped
```

(continues on next page)

(continued from previous page)

```
    # wrap is used to exchange metadata between functions
    @wraps(func)
    def pn(*args, **kwargs):
        print(func.__name__)
        return func(*args, **kwargs)
    return pn
```

If we execute the mathEx.py again, it will show the help features again.

Note: @wraps exchanges the metadata between the functions as shown in above example.

12.1.3 Decorators with arguments

Suppose, we want to pass some argument to the decorator as shown below,

```
# mathEx.py

from debugEx import printName

@printName('**')
def add2Num(x, y):
    '''add two numbers'''
    return(x+y)

print(add2Num(2, 4))
# help(add2Num)
```

Note: To pass the argument to the decorator, all we need to write a outer function which takes the input arguments and then write the normal decorator inside that function as shown below,

```
# debugEx.py

from functools import wraps

def printName(prefix=""):
    def addPrefix(func):
        msg = prefix + func.__name__
        # func is the function to be wrapped

        # wrap is used to exchange metadata between functions
        @wraps(func)
        def pn(*args, **kwargs):
            print(msg)
            return func(*args, **kwargs)
        return pn
    return addPrefix
```

Now, run the above code,

```
$ python mathEx.py
**add2Num
6
```

Error

- But, above code will generate error if we do not pass the argument to the decorator as shown below,

```
# mathEx.py

from debugEx import printName

@printName
def add2Num(x, y):
    '''add two numbers'''
    return(x+y)

print(add2Num(2, 4))
# help(add2Num)
```

Following error will be generate after running the code,

```
$ python mathEx.py
Traceback (most recent call last):
  File "mathEx.py", line 10, in <module>
    print(add2Num(2, 4))
TypeError: addPrefix() takes 1 positional argument but 2 were given
```

- One solution is to write the two different codes e.g. 'printName' and 'printNameArg'; then use these decorators as required. **But this will make code repetitive** as shown below,

```
# debugEx.py

from functools import wraps

def printName(func):
    # func is the function to be wrapped

    # wrap is used to exchange metadata between functions
    @wraps(func)
    def pn(*args, **kwargs):
        print(func.__name__)
        return func(*args, **kwargs)
    return pn


def printNameArg(prefix=""):
    def printName(func):
        # func is the function to be wrapped

        # wrap is used to exchange metadata between functions
        @wraps(func)
        def pn(*args, **kwargs):
            print(prefix + func.__name__)
            return func(*args, **kwargs)
        return pn
    return printName
```

Now, modify the math.py as below,

```
# mathEx.py

from debugEx import printName, printNameArg

@printNameArg('**')
def add2Num(x, y):
```

(continues on next page)

(continued from previous page)

```
    '''add two numbers'''
    return(x+y)

@printName
def diff2Num(x, y):
    '''subtract two integers only'''
    return(x-y)

print(add2Num(2, 4))
print(diff2Num(2, 4))
# help(add2Num)
```

Next execute the code,

```
$ python mathEx.py
**add2Num
6
diff2Num
-2
```

12.1.4 DRY decorator with arguments

In previous code, we repeated the same code two time for creating the decorator with and without arguments. But, there is a better way to combine both the functionality in one decorator using partial function as shown below,

```
# debugEx.py

from functools import wraps, partial

def printName(func=None, *, prefix=""):
    if func is None:
        return partial(printName, prefix=prefix)
    # wrap is used to exchange metadata between functions
    @wraps(func)
    def pn(*args, **kwargs):
        print(prefix + func.__name__)
        return func(*args, **kwargs)
    return pn
```

Now, modify the mathEx.py i.e. remove printNameArg decorator from the code, as below,

```
# mathEx.py

from debugEx import printName

@printName(prefix='**')
def add2Num(x, y):
    '''add two numbers'''
    return(x+y)

@printName
def diff2Num(x, y):
    '''subtract two integers only'''
    return(x-y)

print(add2Num(2, 4))
print(diff2Num(2, 4))
# help(add2Num)
```

Next, run the code and it will display following results,

```
$ python mathEx.py
**add2Num
6
diff2Num
-2
```

Partial function is required because, when we pass argument to the decorator i.e. @printName(prifix='**'), then decorator will not find any function argument at first place, hence return func(*arg, **kwargs) will generate error as there is no 'func'.

To solve this problem, partial is used which returns the an new function, with modified parameters i.e. newFunc(func = printName, prefix = prefix).

12.1.5 Decorators inside the class

In previous sections, decorators are defined as functions. In this section, decorators will be defined as class methods.

class method and instance method decorator

In the following code, two types of decorators are defined inside the class i.e. using class method and instance method,

```
# clsDecorator.py

from datetime import datetime

class DateDecorator(object):
    # instance method decorator
    def instantMethodDecorator(self, func):
        def printDate(*args, **kwargs):
            print("Instance method decorator at time : \n", datetime.today())
            return func(*args, **kwargs)
        return printDate

    # class method decorator
    @classmethod
    def classMethodDecorator(self, func):
        def printDate(*args, **kwargs):
            print("Class method decorator at time : \n", datetime.today())
            return func(*args, **kwargs)
        return printDate


# decorator: instance method
a = DateDecorator()
@a.instantMethodDecorator
def add(a, b):
    return a+b

# decorator: class method
@DateDecorator.classMethodDecorator
def sub(a, b):
    return a-b

sum = add(2, 3)
# Instance method decorator at time :
#  2017-02-04 13:31:27.742283

diff = sub(2, 3)
```

(continues on next page)

(continued from previous page)

```
38 # Class method decorator at time :
39 #  2017-02-04 13:31:27.742435
```

Note that, we need to instantiate the instance mehtod decorator before using it as shown at line 23; whereas class decorator can be used as ClassName.DecoratorName.

12.1.6 Conclusion

In this section, we saw the relation between 'function inside the function' and decorator. Then, decorator with and without arguments are discussed. Lastly, class decorators are shown with examples.

12.2 Descriptors

Descriptor gives us the fine control over the attribute access. It allows us to write reusable code that can be shared between the classes as shown in this tutorial.

12.2.1 Problems with @property

Before beginning the descriptors, let's look at the @property decorator and it's usage along with the problem. Then we will see the descriptors in next section.

```
# square.py

class Square(object):
    def __init__(self, side):
        self.side = side

    def aget(self):
        return self.side ** 2

    def aset(self, value):
        print("can not set area")

    def adel(self):
        print("Can not delete area")

    area = property(aget, aset, adel, doc="Area of sqare")


s = Square(10)
print(s.area)  # 100

s.area = 10  # can not set area

del s.area  # can not delete area
```

Note that in above code, no bracket is used for calculating the area i.e. 's.area' is used, instead of s.area(); because area is defined as property, not as a method.

Above code can be rewritten using @property decorator as below,

```
# square.py

class Square(object):
    """ A square using property with decorator"""
    def __init__(self, side):
```

(continues on next page)

(continued from previous page)

```
        self.side = side

    @property
    def area(self):
        """Calculate the area of the square"""
        return self.side ** 2

    # name for setter and deleter (i.e. @area) must be same
    # as the method for which @property is used i.e. area here
    @area.setter
    def area(self, value):
        """ Do not allow set area directly"""
        print("Can not set area")

    @area.deleter
    def area(self):
        """Do not allow deleting"""
        print("Can not delete area")


s = Square(10)
print(s.area)  # 100

s.area = 10  # can not set area

del s.area  # can not delete area
```

Note that, @area.setter and @area.deleter are optional here. We can stop writing the code after defining the @property. In that, case setter and deleter option will generate standard exception i.e. attribute error here. If we want to perform some operations, then setter and deleter options are required.

Further, code is repetitive because @property, setter and deleter are the part of same method here i.e. area.

We can merge all three methods inside one method as shown next. But before looking at that example, let's understand two python features first i.e. **kwargs and locals().

**kwargs

**kwargs converts the keyword arguments into dictionary as shown below,

```
def func(**kwargs):
    print(kwargs)

func(a=1, b=2)  # {'a': 1, 'b': 2}
```

locals()

Locals return the dictionary of local variables, as shown below,

```
def mathEx(a, b):
    add = a + b
    diff = a - b
    print(locals())

mathEx(3, 2)  # {'a': 3, 'add': 5, 'b': 2, 'diff': 1}
```

Now, we can implement all the get, set and del method inside one method as shown below,

```
# square.py

def nested_property(func):
    """ Nest getter, setter and deleter"""
    names = func()

    names['doc'] = func.__doc__
    return property(**names)


class Square(object):
    """ A square using property with decorator"""
    def __init__(self, side):
        self.side = side

    @nested_property
    def area():
        """Calculate the area of the square"""

        def fget(self):
            """ Calculate area """
            return self.side ** 2

        def fset(self, value):
            """ Do not allow set area directly"""
            print("Can not set area")

        def fdel(self):
            """Do not allow deleting"""
            print("Can not delete area")

        return locals()

s = Square(10)
print(s.area)  # 100

s.area = 10  # can not set area

del s.area  # can not delete area
```

Note: @propery is good for performing certain operations before get, set and delete operation. But, we need to implement it for all the functions separately and code becomes repetitive for larger number of methods. In such cases, descriptors can be useful.

12.2.2 Data Descriptors

```
# data_descriptor.py

class DataDescriptor(object):
    """ descriptor example """
    def __init__(self):
        self.value = 0

    def __get__(self, instance, cls):
        print("data descriptor __get__")
        return self.value

    def __set__(self, instance, value):
```

(continues on next page)

(continued from previous page)

```
        print("data descriptor __set__")
        try:
            self.value = value.upper()
        except AttributeError:
            self.value = value

    def __delete__(self, instance):
        print("Can not delete")

class A(object):
    attr =  DataDescriptor()



d = DataDescriptor()
print(d.value)  # 0

a = A()
print(a.attr)
# data descriptor __get__
# 0

# a.attr is equivalent to below code
print(type(a).__dict__['attr'].__get__(a, type(a)))
# data descriptor __get__
# 0

# set will upper case the string
a.attr = 2  # 2
# lazy loading: above o/p will not display if
# below line is uncommented
a.attr = 'tiger' # TIGER
print(a.__dict__) # {}

# Following are the outputs of above three commands
# data descriptor __set__
# data descriptor __set__
# {}
# data descriptor __get__
# data descriptor __get__
# TIGER
```

Note:

- Note that object 'd' does not print the 'data descriptor __get__' but object of other class i.e. A prints the message. In the other words, descriptor can not use there methods by its' own. **Other's class-attributes** can use descriptor's methods as shown in above example.

Also, see the outputs of last three commands. We will notice that,

Note:

- The set values are not store in the instance dictionary i.e. print(a.__dict__) results in empty dictionary.
- Further, a.attr = 2 and a.attr 'tiger' performs the set operation immediately (see the __set__ message at outputs), but __get__ operations are performed at the end of the code, i.e. first print(a.__dict__) outputs are shown and then get operations is performed.
- Lastly, set operation stores only last executed value, i.e. only TIGER is printed at the end, but not 2.

12.2.3 non-data descriptor

non-data descriptor stores the assigned values in the dictionary as shown below,

```
# non_data_descriptor.py

class NonDataDescriptor(object):
    """ descriptor example """
    def __init__(self):
        self.value = 0

    def __get__(self, instance, cls):
        print("non-data descriptor __get__")
        return self.value + 10

class A(object):
    attr =  NonDataDescriptor()

a = A()
print(a.attr)
# non-data descriptor __get__
# 10

a.attr = 3
a.attr = 3
print(a.__dict__) # {'attr': 4}
```

Important:

- In Non-data descriptor, the assigned values are stored in instance dictionary (and only last assigned value is stored in dictionary); whereas data descriptor assigned values are stored in descriptor dictionary because the set method of descriptor is invoked.

12.2.4 __getattribute__ breaks the descriptor usage

In below code, __getattribute__ method is overridden in class Overriden. Then, instance of class Overriden is created and finally the descriptor is called at Line 18. In this case, __getattribute__ method is invoked first and does not give access to descriptor.

```
# non_data_descriptor.py

class NonDataDescriptor(object):
    """ descriptor example """
    def __init__(self):
        self.value = 0

    def __get__(self, instance, cls):
        print("non-data descriptor __get__")
        return self.value + 10

class Overriden(object):
    attr =  NonDataDescriptor()
    def __getattribute__(self, name):
        print("Sorry, No way to reach to descriptor!")

o = Overriden()
o.attr  # Sorry, No way to reach to descriptor!
```

Note: Descriptors can not be invoked if __getattribute__ method is used in the class as shown in above example. We need to find some other ways in such cases.

12.2.5 Use more than one instance for testing

Following is the good example, which shows that test must be performed on more than one object of a classes. As following code, will work fine for one object, but error can be caught with two or more objects only.

```
# Examples.py

class DescriptorClassStorage(object):
    """ descriptor example """
    def __init__(self, default = None):
        self.value = default

    def __get__(self, instance, cls):
        return self.value
    def __set__(self, instance, value):
        self.value = value

class StoreClass(object):
    attr =  DescriptorClassStorage(10)


store1 =  StoreClass()
store2 =  StoreClass()

print(store1.attr, store2.attr)  # 10, 10

store1.attr = 30

print(store1.attr, store2.attr) # 30, 30
```

In above code, only store1.attr is set to 30, but value of store2.attr is also changes. This is happening because, in data-descriptors values are stored in descriptors only (not in instance dictionary as mentioned in previous section).

12.2.6 Examples

12.2.6.1 Write a descriptor which allows only positive values

Following is the code to test the positive number using descriptor,

```
# positiveValueDescriptor.py

class PositiveValueOnly(object):
    """ Allows only positive values """

    def __init__(self):
        self.value = 0

    def __get__(self, instance, cls):
        return self.value

    def __set__(self, instance, value):
        if value < 0 :
            raise ValueError ('Only positive values can be used')
        else:
```

(continues on next page)

(continued from previous page)

```
            self.value = value

class Number(object):
    """ sample class that uses PositiveValueOnly descriptor"""

    value = PositiveValueOnly()


test1 = Number()
print(test1.value)  #  0

test1.value = 10
print(test1.value)  #  0

test1.value = -1
# [...]
# ValueError: Only positive values can be used
```

12.2.6.2 Passing arguments to decorator

IN previous codes, no arguments were passed in the decorator. In this example, taxrate is passed in the decorator and total price is calculated based on tax rate.

```
# taxrate.py

class Total(object):
    """ Calculate total values """

    def __init__(self, taxrate = 1.20):
        self.rate = taxrate

    def __get__(self, instance, cls):
        # net_price * rate
        return instance.net * self.rate

    # override __set__, so that there will be no way to set the value
    def __set__(self, instance, value):
        raise NoImplementationError("Can not change value")

class PriceNZ(object):
    total = Total(1.5)

    def __init__(self, net, comment=""):
        self.net = net
        self.comment = comment

class PriceAustralia(object):
    total = Total(1.3)

    def __init__(self, net):
        self.net = net


priceNZ = PriceNZ(100, "NZD")
print(priceNZ.total)  # 150.0

priceAustralia = PriceAustralia(100)
print(priceAustralia.total)  # 130.0
```

Note: In above example, look for the PriceNZ class, where init function takes two arguments and one of which is used by descriptor using 'instance.net' command. Further, init function in class Total need one argument i.e. taxrate, which is passed by individual class which creating the object of the descriptor.

12.2.7 Conclusion

In this section, we discussed data-descriptors and non-data-descriptors. Also, we saw the way values are stored in these two types of descriptors. Further, we saw that __getattribute__ method breaks the descriptor calls.

Chapter 13

More examples

This chapter contains several examples of different topics which we learned in previous chapters,

13.1 Generalized attribute validation

In this chapter 'Functions', '@property', 'Decorators' and 'Descriptors' are described. Also, these techniques are used together in final example for attribute validation. Attribute validation is defined in Section Section 13.1.1.1.

13.1.1 Function

Various features are provided in Python3 for better usage of the function. In this section, 'help feature' and 'argument feature' are added to functions.

13.1.1.1 Help feature

In Listing 13.1, anything which is written between 3 quotation marks after the function declaration (i.e. line 6-9), will be displayed as the output of 'help' command as shown in line 13.

Note: Our aim is to write the function which adds the integers only, but currently it is generating the output for the 'strings' as well as shown in line 21. Therefore, we need 'attribute validation' so that inputs will be verified before performing the operations on them.

Listing 13.1: Help feature

```
1  # addIntEx.py
2
3  # line 6-9 will be displayed,
4  # when help command is used for addIntEx as shown in line 13.
5  def addIntEx(x,y):
6      '''add two variables (x, y):
7          x: integer
8          y: integer
9          returnType: integer
10     '''
11     return (x+y)
12
13 help(addIntEx)  # line 6-9 will be displayed as output
14
15 #adding numbers: desired result
```

(continues on next page)

(continued from previous page)

```
16 intAdd = addIntEx(2,3)
17 print("intAdd =", intAdd) # 5
18
19 # adding strings: undesired result
20 # attribute validation is used for avoiding such errors
21 strAdd = addIntEx("Meher ", "Krishna")
22 print("strAdd =", strAdd) # Meher Krishna
```

13.1.1.2 Keyword argument

In Listing 13.2, 'addKeywordArg(x, *, y)' is a Python feature; in which all the arguments after '*' are considered as positional argument. Hence, 'x' and 'y' are the 'positional' and 'keyword' argument respectively. Keyword arguments must be defined using the variable name e.g 'y=3' as shown in Lines 9 and 12. If name of the variable is not explicitly used, then Python will generate error as shown in Line 16. Further, keyword argument must be defined after all the positional arguments, otherwise error will be generated as shown in Line 19.

Lastly, in Line 2, the definition 'addKeywordArg(x: **int**, *, y: **int**) -> **int**' is presenting that inputs (x and y) and return values are of integer types. These help features can be viewed using metaclass command, i.e. '.__annotations__', as shown in Lines 23 and 24. Note that, this listing is not validating input types. In next section, input validation is applied for the functions.

Listing 13.2: Keyword argument

```
1  # addKeywordArg.py
2  def addKeywordArg(x:int, *,  y:int) -> int:
3      '''add two numbers:
4           x: integer, postional argument
5           y: integer, keyword argument
6           returnType: integer    '''
7      return (x+y)
8
9  Add1 = addKeywordArg(2, y=3) # x: positional arg and y: keyword arg
10 print(Add1) # 5
11
12 Add2 = addKeywordArg(y=3, x=2) # x and y as keyword argument
13 print(Add2) # 5
14
15 ## it's wrong, because y is not defined as keyword argument
16 #Add3 = addPositionalArg(2, 3) # y should be keyword argument i.e. y=3
17
18 ## keyword arg should come after positional arg
19 #Add4 = addPositionalArg(y=3, 2) # correct (2, y=3)
20
21 help(addKeywordArg) # Lines 3-6 will be displayed as output
22
23 print(addKeywordArg.__annotations__)
24 ## {'return': <class 'int'>, 'x': <class 'int'>, 'y': <class 'int'>}
25
26 ## line 2 is only help (not validation), i.e. string addition will still unchecked
27 strAdd = addKeywordArg("Meher ", y = "Krishna")
28 print("strAdd =", strAdd)
```

13.1.1.3 Input validation

In previous section, help features are added to functions, so that the information about the functions can be viewed by the users. In this section, validation is applied to input arguments, so that any invalid input will not be process by the function and corresponding error be displayed to the user.

In Listing 13.3, Lines 8-9 are used to verify the type of input variable 'x'. Line 8 checks whether the input is integer or not; if it is not integer that error will be raised by line 9, as shown in lines 19-22. Similarly, Lines 11-12 are used to verify the type of variable 'y'.

Listing 13.3: Input Validation

```
1  # addIntValidation.py
2  def addIntValidation(x:int, *,  y:int)->int:
3      '''add two variables (x, y):
4          x: integer, postional argument
5          y: integer, keyword argument
6          returnType: integer
7          '''
8      if type(x) is not int: # validate input 'x' as integer type
9          raise TypeError("Please enter integer value for x")
10
11     if type(y) is not int: # validate input 'y' as integer type
12         raise TypeError("Please enter integer value for y")
13
14     return (x+y)
15
16 intAdd=addIntValidation(y=3, x=2)
17 print("intAdd =", intAdd)
18
19 #strAdd=addIntValidation("Meher ", y = "Krishna")
20 ## Following error will be generated for above command,
21 ## raise TypeError("Please enter integer value for x")
22 ## TypeError: Please enter integer value for x
23
24 help(addIntValidation) # Lines 3-6 will be displayed as output
25 print(addIntValidation.__annotations__)
26 ## {'return': <class 'int'>, 'x': <class 'int'>, 'y': <class 'int'>}
```

13.1.2 Decorators

Decorators are used to add additional functionalities to functions. In Section Section 13.1.1.3, 'x' and 'y' are validated individually; hecce, if there are large number of inputs, then the method will not be efficient. Decorator will be used in Section Section 13.1.5 to write the generalized validation which can validate any kind of input.

13.1.2.1 Add decorators and problems

Listing 13.4 is the decorator, which prints the name of the function i.e. whenever the function is called, the decorator will be executed first and print the name of the function and then actual function will be executed. The decorator defined above the function declaration as shown in line 4 of Listing 13.5.

Listing 13.4: Decorator which prints the name of function

```
1  # funcNameDecorator.py
2  def funcNameDecorator(func): # function as input
3      def printFuncName(*args, **kwargs): #take all arguments of function as input
4          print("Function Name:", func.__name__) # print function name
5          return func(*args, **kwargs) # return function with all arguments
6      return printFuncName
```

In Listing 13.5, first decorator 'funcNameDecorator' is imported to the listing in Line 2. Then, decorator is applied to function 'addIntDecorator' in Line 4. When Line 15 calls the function 'addIntDecorator', the decorator is executed first and name of function is printed, after that print command at Line 16 is executed.

Warning: In the Listing, we can see that Help function is not working properly now as shown in Listing 19. Also, Decorator removes the metaclass features i.e. 'annotation' will not work, as shown in Line 24.

Listing 13.5: Decorator applied to function

```
1  #addIntDecorator.py
2  from funcNameDecorator import funcNameDecorator
3
4  @funcNameDecorator
5  def addIntDecorator(x:int, *,  y:int) -> int:
6      '''add two variables (x, y):
7          x: integer, postional argument
8          y: integer, keyword argument
9          returnType: integer
10     '''
11     return (x+y)
12
13 ## decorator will be executed when function is called,
14 ## and function name will be displayed as output as shown below,
15 intAdd=addIntDecorator(2, y=3) # Function Name: addIntDecorator
16 print("intAdd =", intAdd) # 5
17
18 ##problem with decorator: help features are not displayed as shown below
19 help(addIntDecorator) # following are the outputs of help command
20 ## Help on function wrapper in module funcNameDecorator:
21 ## wrapper(*args, **kwargs)
22
23 ## problem with decorator: no output is displayed
24 print(addIntDecorator.__annotations__) # {}
```

Note: It is recommonded to define the decorators in the separate files e.g. 'funcNameDecorator.py' file is used here. It's not good practice to define decorator in the same file, it may give some undesired results.

13.1.2.2 Remove problems using functools

The above problem can be removed by using two additional lines in Listing 13.4. The listing is saved as Listing 13.6 and Lines 2 and 6 are added, which solves the problems completely. In Line 2, 'wraps' is imported from 'functools' library and then it is applied inside the decorator at line 6. Listing 13.7 is same as Listing 13.5 except new decorator which is defined in Listing 13.6 is called at line 4.

Listing 13.6: Decorator with 'wrap' decorator

```
1  # funcNameDecoratorFunctool.py
2  from functools import wraps
3
4  def funcNameDecoratorFunctool(func): # function as input
5      #func is the function to be wrapped
6      @wraps(func)
7      def printFuncName(*args, **kwargs): #take all arguments of function as input
8          print("Function Name:", func.__name__) # print function name
9          return func(*args, **kwargs) # return function with all arguments
10     return printFuncName
```

Listing 13.7: Help features are visible again

```
1  # addIntDecoratorFunctool.py
```

(continues on next page)

(continued from previous page)

```
from funcNameDecoratorFunctool import funcNameDecoratorFunctool

@funcNameDecoratorFunctool
def addIntDecorator(x:int, *,  y:int) -> int:
    '''add two variables (x, y):
        x: integer, postional argument
        y: integer, keyword argument
        returnType: integer
    '''
    return (x+y)

intAdd=addIntDecorator(2, y=3) # Function Name: addIntDecorator
print("intAdd =", intAdd) # 5

help(addIntDecorator) # lines 6-9 will be displaed

print(addIntDecorator.__annotations__)
##{'return': <class 'int'>, 'y': <class 'int'>, 'x': <class 'int'>}
```

13.1.3 @property

In this section, area and perimeter of the rectangle is calculated and '@property' is used to validate the inputs before calculation. Further, this example is extended in the next sections for adding more functionality for 'attribute validation'.

Explanation Listing 13.8

In line 28 of the listing, @property is used for 'length' attribute of the class Rectangle. Since, @property is used, therefore 'getter' and 'setter' can be used to validate the type of length. Note that, in setter part, i.e. Lines 34-40, self._length (see '_' before length) is used for setting the valid value in 'length' attribute. In the setter part validation is performed at Line 38 using 'isinstance'. n Line 54, the value of length is passed as float, therefore error will be raised as shown in Line 56.

Now, whenever 'length' is accessed by the code, it's value will be return by getter method as shown in Lines 28-32. In the other words, this block will be executed every time we use 'length' value. To demonstrate this, print statement is used in Line 31. For example, Line 44 print the length value, therefore line 31 printed first and then length is printed as shown in Lines 45-46.

Also, @property is used for the method 'area' as well. Therefore, output of this method can be directly obtained as shown in Lines 48-52. Further, for calculating area, the 'length' variable is required therefore line 51 will be printed as output, which is explained in previous paragraph.

Note: In this listing, the type-check applied to 'length' using @property. But, the problem with this method is that we need to write it for each attribute e.g. length and width in this case which is not the efficient way to do the validation. We will remove this problem using Descriptor in next section.

Listing 13.8: Attribute validation using @property

```
# rectProperty.py
class Rectangle:
    '''
    -Calculate Area and Perimeter of Rectangle
    -getter and setter are used to displaying and setting the length value.
    -width is set by init function
    '''

    # self.length is used in below lines,
    # but length is not initialized by __init__,
```

(continues on next page)

(continued from previous page)

```
    # initialization is done by .setter at lines 34 due to @property at line 27,
    # also value is displayed by getter (@propety) at line 28-32
    # whereas `width' is get and set as simple python code and without validation
    def __init__(self, length, width):
        #if self._length is used, then it will not validate through setter.
        self.length = length
        self.width = width

    @property
    def area(self):
        '''Calculates Area: length*width'''
        return self.length * self.width

    def perimeter(self):
        '''Calculates Perimeter: 2*(length+width)'''
        return 2 * (self.length + self.width)

    @property
    def length(self):
        '''displaying length'''
        print("getting length value through getter")
        return self._length

    @length.setter
    def length(self, value):
        '''setting length'''
        print("saving value through setter", value)
        if not isinstance(value, int): # validating length as integer
            raise TypeError("Only integers are allowed")
        self._length = value

r = Rectangle(3,2) # following output will be displayed
## saving value through setter 3
print(r.length) # following output will be displayed
## getting length value through getter
## 3

## @property is used for area,
## therefore it can be accessed directly to display the area
print(r.area) # following output will be displayed
## getting length value through getter
## 6

#r=Rectangle(4.3, 4) # following error will be generated
## [...]
## TypeError: Only integers are allowed

# print perimeter of rectangle
print(Rectangle.perimeter(r))
## getting length value through getter
## 10
```

13.1.4 Descriptors

Descriptor are the classes which implement three core attributes access operation i.e. get, set and del using '__get__', '__set__' and '__del__' as shown in Listing 13.9. In this section, validation is applied using Descriptor to remove the problem with @property.

Explanation Listing 13.9

Here, class integer is used to verify the type of the attributes using '__get__' and '__set__' at Lines

6 and 12 respectively. The class 'Rect' is calling the class 'Integer' at Lines 19 and 20. The name of the attribute is passed in these lines, whose values are set by the Integer class in the form of dictionaries at Line 16. Also, value is get from the dictionary from Line 10. Note that, in this case, only one line is added for each attribute, which removes the problem of '@property' method.

Listing 13.9: Attribute validation using Descriptor

```
1  # rectDescriptor.py
2  class Integer:
3      def __init__(self, parameter):
4          self.parameter = parameter
5
6      def __get__(self, instance, cls):
7          if instance is None: # required if descriptor is
8              return self # used as class variable
9          else: # in this code, only following line is required
10             return instance.__dict__[self.parameter]
11
12     def __set__(self, instance, value):
13         print("setting %s to %s" % (self.parameter, value))
14         if not isinstance(value, int):
15             raise TypeError("Interger value is expected")
16         instance.__dict__[self.parameter] = value
17
18 class Rect:
19     length = Integer('length')
20     width = Integer('width')
21     def __init__(self, length, width):
22         self.length = length
23         self.width = width
24
25     def area(self):
26         '''Calculates Area: length*width'''
27         return self.length * self.width
28
29 r = Rect(3,2)
30 ## setting length to 3
31 ## setting width to 3
32
33 print(r.length) # 3
34
35 print("Area:", Rect.area(r)) # Area:  6
36
37 #r = Rect(3, 1.5)
38 ## TypeError: Interger value is expected
```

13.1.5 Generalized validation

In this section, decorators and descriptors are combined to create a validation, where attribute-types are defined by the individual class authors.

Note: Note that, various types i.e. '@typeAssert(author=str, length=int, width=float)' will be defined by class Author for validation.

Explanation Listing 13.10

In this code, first a decorator 'typeAssert' is applied to class 'Rect' at line 27. The typeAssert contains the name of the attribute along with it's valid type. Then the decorator (Lines 19-24), extracts the 'key-value' pairs i.e. 'parameter-expected]_type' (see Line 21) and pass these to descriptor 'TypeCheck'

through Line 22. If type is not valid, descriptor will raise error, otherwise it will set the values to the variables. Finally, these set values will be used by the class 'Rect' for further operations.

Listing 13.10: Generalized attribute validation

```
1  #rectGeneralized.py
2  class TypeCheck:
3      def __init__(self, parameter, expected_type):
4          self.parameter = parameter
5          self.expected_type = expected_type
6
7      def __get__(self, instance, cls):
8          if instance is None: # required if descriptor is
9              return self # used as class variable
10         else: # in this code, only following line is required
11             return instance.__dict__[self.parameter]
12
13     def __set__(self, instance, value):
14         print("setting %s to %s" % (self.parameter, value))
15         if not isinstance(value, self.expected_type):
16             raise TypeError("%s value is expected" % self.expected_type)
17         instance.__dict__[self.parameter] = value
18
19 def typeAssert(**kwargs):
20     def decorate(cls):
21         for parameter, expected_type in kwargs.items():
22             setattr(cls, parameter, TypeCheck(parameter, expected_type))
23         return cls
24     return decorate
25
26  # define attribute types here in the decorator
27 @typeAssert(author=str, length = int, width = float)
28 class Rect:
29     def __init__(self, *,  length, width, author = ""): #require kwargs
30         self.length = length
31         self.width = width * 1.0 # to accept integer as well
32         self.author = author
33
34 r = Rect (length=3, width=3.1, author = "Meher")
35 ## setting length to 3
36 ## setting width to 3.1
37 ## setting author to Meher
38
39 #r = Rect (length="len", width=3.1, author = "Meher") # error shown below
40 ## File "rectProperty.py", line 42,
41 ## [ ... ]
42 ## TypeError: <class 'int'> value is expected
```

13.1.6 Summary

In this chapter, we learn about functions, @property, decorators and descriptors. We see that @property is useful for customizing the single attribute whereas descriptor is suitable for multiple attributes. Further, we saw that how decorator and descriptor can be used to enhance the functionality of the code with DRY (don't repeat yourself) technique.

13.2 Inheritance with Super

In this section, inheritance is discussed using super command. In most languages, the super method calls the parent class, whereas in python it is slightly different i.e. it consider the child before parent, as shown in this

section.

13.2.1 Super : child before parent

Lets understand super with the help of an example. First, create a class Pizza, which inherits the DoughFactory for getting the dough as below,

```
# pizza.py

class DoughFactory(object):

    def get_dough(self):
        return 'white floor dough'


class Pizza(DoughFactory):

    def order_pizza(self, *toppings):
        print("getting dough")

        # dough = DoughFactory.get_dough()
        ## above line is commented to work with DRY principle
        ## use super as below,
        dough = super().get_dough()
        print("Making pie using '%s'" % dough)

        for topping in toppings:
            print("Adding %s" % topping)

if __name__ == '__main__':
    Pizza().order_pizza("Pepperoni", "Bell Pepper")
```

Run the above code and we will get below output,

```
$ python -i pizza.py

getting dough
Making pie using 'white floor dough'
Adding Pepperoni
Adding Bell Pepper
>>> help(Pizza)
    Help on class Pizza in module __main__:

    class Pizza(DoughFactory)
     |  Method resolution order:
     |      Pizza
     |      DoughFactory
     |      builtins.object
```

Note: The resolution order shows that the way in which python interpretor tries to find the methods i.e. it tries to find get_dough in Pizza class first; if not found there, it will go to DoughFactory.

Now, create the another class in separate python file as below,

```
# wheatDough.py

from pizza import Pizza, DoughFactory
```

(continues on next page)

(continued from previous page)

```
class WheatDoughFactory(DoughFactory):

    def get_dough(self):
        return("wheat floor dough")

class WheatPizza(Pizza, WheatDoughFactory):
    pass

if __name__ == '__main__':
    WheatPizza().order_pizza('Sausage', 'Mushroom')
```

Note: In python, Inheritance chain is not determine by the Parent class, but by the child class.

If we run the wheatDough.py, it will call the super command in class Pizza in pizza.py will not call his parent class i.e. DoughFactory, but the parent class of the child class i.e. WheatDoughFactory. This is called Dependency injection.

Important:

- super consider the children before parents i.e. it looks methods in child class first, then it goes for parent class.
- Next, it calls the parents in the order of inheritance.
- use keyword arguments for cooperative inheritance.

For above reasons, super is super command, as it allows to change the order of Inheritance just by modifying the child class, as shown in above example.

Note: For better understanding of the super command, some more short examples are added here,

---

13.2.2 Inherit __init__

In the following code, class RectArea is inheriting the __init__ function of class RectLen. In the other word, length is set by class RectLen and width is set by class RectArea and finally area is calculated by class RectArea.

```
# rectangle.py

class RectLen(object):
    def __init__(self, length):
        self.length = length


class RectArea(RectLen):
    def __init__(self, length, width):
        self.width = width
        super().__init__(length)
        print("Area : ", self.length * self.width)


RectArea(4, 3)  # Area :  12
```

In the same way, the other functions of parent class can be called. In following code, printClass method of parent class is used by child class.

```
# printClass.py

class A(object):
    def printClassName(self):
        print(self.__class__.__name__)

class B(A):
    def printName(self):
        super().printClassName()

a = A()
a.printClassName()  # A

b = B()
b.printClassName()  # B
```

Note: In above code, print(self.__class__.__name__) is used for printing the class name, instead of print("A"). Hence, when child class will inherit this function, then __class__ will use the name of the child class to print the name of the class, therefore Line 15 prints "B" instead of A.

13.2.3 Inherit __init__ of multiple classes

In this section, various problems are discussed along with the solutions, which usually occurs during multiple inheritance.

13.2.3.1 Problem : super() calls __init__ of one class only

In following example, class C is inheriting the class A and B. But, the super function in class C will inherit only one class init function i.e. init function of the class which occurs first in the inheritance order.

```
#multipleInheritance.py

class A(object):
    def __init__(self):
        print("A")

class B(object):
    def __init__(self):
        print("B")

class C(A, B):
    def __init__(self):
        super().__init__()

# init of class B is not inherited
c = C() # A
```

13.2.3.2 Solution 1

Following is the first solution, where __init__ function of classes are invoked explicitly.

```
#multipleInheritance.py

class A(object):
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        print("A")

class B(object):
    def __init__(self):
        print("B")

class C(A, B):
    def __init__(self):
        A.__init__(self)  # self is required
        B.__init__(self)

c = C()
# A
# B
```

13.2.3.3 Correct solution

Following is the another solution of the problem; where super() function is added in both the classes. Note that, the super() is added in class B as well, so that class(B, A) will also work fine.

```
#multipleInheritance.py

class A(object):
    def __init__(self):
        print("reached A")
        super().__init__()
        print("A")

class B(object):
    def __init__(self):
        print("reached B")
        super().__init__()
        print("B")

class C(A, B):
    def __init__(self):
        super().__init__()

c = C()
# reached A
# reached B
# B
# A
```

The solution works fine here because in Python super consider the child before parent, which is discussed in Section *Super : child before parent*. Please see the order of output as well.

13.2.4 Math Problem

This section summarizes the above section using math problem. Here, we want to calculate (x * 2 + 5), where x = 3.

13.2.4.1 Solution 1

This is the first solution, __init__ function of two classes are invoked explicitly. The only problem here is that the solution does not depend on the order of inheritance, but on the order of invocation, i.e. if we exchange the lines 15 and 16, the solution will change.

```
# mathProblem.py

class Plus5(object):
    def __init__(self, value):
        self.value = value + 5

class Multiply2(object):
    def __init__(self, value):
        self.value = value * 2

class Solution(Multiply2, Plus5):
    def __init__(self, value):
        self.value = value

        Multiply2.__init__(self, self.value)
        Plus5.__init__(self, self.value)


s = Solution(3)
print(s.value)  # 11
```

13.2.4.2 problem with super

One of the problem with super is that, the top level super() function does not work if it has some input arguments. If we look the output of following code carefully, then we will find that error is generated after reaching to class Plus5. When class Plus5 uses the super(), then it calls the metaclass's (i.e. object) __init__ function, which does not take any argument. Hence it generates the error 'object.__init__() takes no parameters'.

To solve this problem, we need to create another class as shown in next section.

```
# mathProblem.py

class Plus5(object):
    def __init__(self, value):

        print("Plus 5 reached")
        self.value = value + 5

        super().__init__(self.value)
        print("Bye from Plus 5")

class Multiply2(object):
    def __init__(self, value):

        print("Multiply2 reached")
        self.value = value * 2

        super().__init__(self.value)
        print("Bye from Multiply2")

class Solution(Multiply2, Plus5):
    def __init__(self, value):
        self.value = value
        super().__init__(self.value)


s = Solution(3)
print(s.value)

# Multiply2 reached
```

(continues on next page)

(continued from previous page)

```
# Plus 5 reached
# [...]
# TypeError: object.__init__() takes no parameters
```

13.2.4.3 Solution 2

To solve the above, we need to create another class, and inherit it in classes Plus5 and Multiply2 as below,

In below code, MathClass is created, whose init function takes one argument. Since, MathClass does not use super function, therefore above error will not generate here.

Next, we need to inherit this class in Plus5 and Multiply2 for proper working of the code, as shown below. Further, below code depends on order of inheritance now.

```
# mathProblem.py

class MathClass(object):
    def __init__(self, value):
        print("MathClass reached")
        self.value = value
        print("Bye from MathClass")

class Plus5(MathClass):
    def __init__(self, value):
        print("Plus 5 reached")
        self.value = value + 5
        super().__init__(self.value)
        print("Bye from Plus 5")

class Multiply2(MathClass):
    def __init__(self, value):
        print("Multiply2 reached")
        self.value = value * 2
        super().__init__(self.value)
        print("Bye from Multiply2")

class Solution(Multiply2, Plus5):
    def __init__(self, value):
        self.value = value
        super().__init__(self.value)


s = Solution(3)
print(s.value)  # 11

# Multiply2 reached
# Plus 5 reached
# MathClass reached
# Bye from MathClass
# Bye from Plus 5
# Bye from Multiply2
# 11

## uncomment below to see the Method resolution order
print(help(Solution))
# class Solution(Multiply2, Plus5)
#  |  Method resolution order:
#  |      Solution
#  |      Multiply2
#  |      Plus5
```

(continues on next page)

(continued from previous page)

```
#  |      MathClass
#  |      builtins.object
```

13.2.5 Conclusion

In this section, we saw the functionality of the super() function. It is shown that super() consider the child class first and then parent classes in the order of inheritance. Also, help command is used for observing the 'method resolution order' i.e. hierarchy of the inheritance.

13.3 Generators

Any function that uses the 'yield' statement is the generator. Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator iterator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

13.3.1 Feed iterations

Typically, it is used to feed iterations as below,

```
# generatorEx.py

def countdown(n):
    while (n>0):
        yield n
        n -= 1

for x in countdown(5):
    print(x)  # 5 4 3 2 1

print()
c = countdown(3)
print(next(c))  # 3
print(next(c))  # 2
print(next(c))  # 1
print(next(c))
# Traceback (most recent call last):
#   File "rectangle.py", line 16, in <module>
#     print(next(c))
# StopIteration
```

If the generator exits without yielding another value, a StopIteration exception is raised.

13.3.2 Receive values

'yield' can receive value too. Calling the function creates the generator instance, which needs to be advance to next yield using 'next' command. Then generator is ready to get the inputs, as shown below,

```
# generatorEx.py

def rxMsg():
    while True:
        item = yield
        print("Message : ", item)
```

(continues on next page)

(continued from previous page)

```
msg = rxMsg()
print(msg)  # <generator object rxMsg at 0xb7049f8c>

next(msg)
# send : Resumes the execution and "sends" a value into the generator function
msg.send("Hello")
msg.send("World")
```

13.3.3 Send and receive values

Both send and receive message can be combined together in generator. Also, generator can be closed, and next() operation will generate error if it is used after closing the generator, as shown below,

```
# generatorEx.py

def rxMsg():
    while True:
        item = yield
        yield("Message Ack: " + item)


msg = rxMsg()

next(msg)
m1 = msg.send("Hello")
print(m1)  # Message Ack: Hello

next(msg)
m2 = msg.send("World")
print(m2)  # Message Ack: World

msg.close()  # close the generator

next(msg)
# Traceback (most recent call last):
#   File "rectangle.py", line 21, in <module>
#     next(msg)
# StopIteration
```

13.3.4 Return values in generator

Generator can return values which is displayed with exception,

```
# generatorEx.py

def rxMsg():
    while True:
        item = yield
        yield("Message Ack: " + item)
        return "Thanks"


msg = rxMsg()

next(msg)
m1 = msg.send("Hello")
print(m1)  # Message Ack: Hello
```

(continues on next page)

(continued from previous page)

```
next(msg)
# Traceback (most recent call last):
#   File "rectangle.py", line 16, in <module>
#     next(msg)
# StopIteration: Thanks

m2 = msg.send("World")
print(m2)
```

13.3.5 'yield from' command

When yield from <expr> is used, it treats the supplied expression as a subiterator. All values produced by that subiterator are passed directly to the caller of the current generator's methods,

```
# generatorEx.py

def chain(x, y):
    yield from x
    yield from y

a = [1, 2, 3]
b = [20, 30]

for i in chain(a, b):
    print(i, end=' ')  # 1, 2, 3, 20, 30

print()
for i in chain(chain(a, a), chain(b, a)):
    print(i, end=' ')  # 1 2 3 1 2 3 20 30 1 2 3
```

Chapter 14

Unix commands

14.1 Introduction

In this chapter, we will implement some of the Unix commands using 'Python'.

14.2 'argparse'

The 'argparse' module is a command line interface. We will use 'argparse' module to read the values from the terminal. For more details, please see the online documentation of the module.

Listing 14.1: 'argpase' module

```
# argparse_ex.py

import argparse

def mulNum(a):
    p = 1
    for item in a:
        p *= item
    return p

parser = argparse.ArgumentParser(description="First argparse code")

# read  list of integers
parser.add_argument(
        type=int, # type int
        nargs='+', # read multiple values in a list
        dest='integers', # name of the attribute returned by argparse
        help='read list of integer: 1 2 3')

parser.add_argument(
        '-p', '--product', # name -p or --product
        dest='multiply', # by default 'dest=product'
        action='store_const', # store values
        const=mulNum, # call function 'mulNum'
        help='product of integers'
    )

parser.add_argument(
        '--sum',
        dest='accumulate',
```

(continues on next page)

(continued from previous page)

```
        action='store_const',
        const=sum, # inbuilt method 'sum' to add the values of list
        help='sum of integers'
    )

args = parser.parse_args() # save arguments in args

# if --sum is in command
if args.accumulate:
    sum = args.accumulate(args.integers)
    print("sum =", sum)

# if '-p or --product' is in command
if args.multiply != None:
# if args.multiply is not None:
    # pass arg.integers to arg.multiply, which calls function 'mulNum'
    product = args.multiply(args.integers)
    print("product = ", product)


####### Execution ####################


######## Help ############
# $ python argparse_ex.py -h  (or $ python argparse_ex.py --help)
# usage: argparse_ex.py [-h] [-p] [--sum] integers [integers ...]

# First argparse code

# positional arguments:
  # integers       read list of integer: 1 2 3

# optional arguments:
  # -h, --help     show this help message and exit
  # -p, --product  product of integers
  # --sum          sum of integers

############ Results #########
# $ python argparse_ex.py 2 5 1 --sum
# sum = 8

# $ python argparse_ex.py 2 5 1 -p
# product =  10

# $ python argparse_ex.py 2 5 1 --product
# product =  10

# $ python argparse_ex.py 2 5 1 -p --sum
# sum = 8
# product =  10

# $ python argparse_ex.py -p 2 5 1 --sum
# sum = 8
# product =  10
```

14.3 find

14.3.1 Command details

Create some files and folders inside a directory. Next go to the directory and run following commands. We have following files and folder in the current directory,

```
$ tree .
.
├── box.py
├── contributor.py
├── csv_format.csv
├── datamine.py
├── data.txt
├── expansion.py
├── expansion.txt
├── mathematician.py
├── methodEx.py
├── price2.csv
├── price.csv
├── price_missing.csv
├── price.py
├── pythonic.py
├── ring.py
├── text_format.txt
├── unix_commands
│   ├── argparse_ex.py
│   ├── cat.py
│   ├── file1.txt
│   └── file2.txt
└── wide_text_format.txt
```

```
(show all items which ends with .py)
$ find -name "*.py"
./box.py
./contributor.py
./datamine.py
./expansion.py
./mathematician.py
./methodEx.py
./price.py
./pythonic.py
./ring.py
./unix_commands/argparse_ex.py
./unix_commands/cat.py


(show all items which contains 'x' in it)
$ find -name "*x*"
./box.py
./data.txt
./expansion.py
./expansion.txt
./methodEx.py
./text_format.txt
./unix_commands
./unix_commands/argparse_ex.py
./unix_commands/file1.txt
./unix_commands/file2.txt
./wide_text_format.txt
```

(continues on next page)

(continued from previous page)

```
(show all directories which contains 'x' in it)
$ find -name "*x*" -type d
./unix_commands


(show all files which contains 'x' in it)
$ find -name "*x*" -type f
./box.py
./data.txt
./expansion.py
./expansion.txt
./methodEx.py
./text_format.txt
./unix_commands/argparse_ex.py
./unix_commands/file1.txt
./unix_commands/file2.txt
./wide_text_format.txt


(for more options, see man page)
$ man find
```

14.3.2 Python implementation

- First, we need to write the code, which can traverse through the directories. Following is the current directory structure,

```
$ tree .
.
├── argparse_ex.py
├── cat.py
├── f2
│   └── tiger.txt
├── file1.txt
├── file2.txt
├── find_ex.py
└── folder1
    └── dog.txt
```

- Python code to traverse through each directory is shown below,

Listing 14.2: traverse directories using 'pathlib'

```
1  # find_ex.py
2
3  from pathlib import Path
4
5  # location of directory
6  p = Path('.') # current directory
7
8  print("All files and folders in current directory:")
9  all_item = [x for x in p.iterdir()]
10 for a in all_item:
11     print(a)
12
13 # files in current folder
14 print("\n")
15 print("Files in current folder:")
```

(continues on next page)

(continued from previous page)

```
files = [x for x in p.iterdir() if x.is_file()]
for f in files:
    print(f)

# directories in current folder
print("\n")
directory = [x for x in p.iterdir() if x.is_dir()]
print("Directories in current folder:")
for d in directory:
    print(d)
```

Below is the output for above code,

```
$ python find_ex.py

All files and folders in current directory:
argparse_ex.py
cat.py
f2
file1.txt
file2.txt
find_ex.py
folder1


Files in current folder:
argparse_ex.py
cat.py
file1.txt
file2.txt
find_ex.py


Directories in current folder:
f2
folder1
```

- Below is code to implement the 'find' command,

Listing 14.3: 'find' command using Python

```
# find_ex.py

import argparse
import sys

from pathlib import Path

parser = argparse.ArgumentParser(description="Find command")

# positional argument when defined without --
# nargs='*' or '?' is required for default-positional-arg values
# * : read all command line arguments;  ?: read one command line arg
parser.add_argument('location', type=str, nargs="*", default='.')

# optional argument when defined with --
# no 'nargs' i.e. it's str (not list of str)
parser.add_argument('--name', type=str, default="*")

# possible values are "d", "f" and "all"
parser.add_argument('--type',type=str,default="all",choices=["d", "f", "all"])
```

(continues on next page)

(continued from previous page)

```
21
22  args = parser.parse_args() # save arguments in args
23  loc = Path(args.location[0])
24
25  items=[]
26  for l in loc.rglob(args.name):
27      if args.type == "d" and l.is_dir():
28          items.append(l)
29      elif args.type == "f" and l.is_file():
30          items.append(l)
31      elif args.type == "all":
32          items.append(l)
33
34  # print output
35  for i in items:
36      print(i)
```

```
(show files which starts with 'f')
$ find -name "f*" -type "f"
./file1.txt
./file2.txt
./find_ex.py

$ python find_ex.py --name "f*"  --type "f"
file1.txt
file2.txt
find_ex.py



(show directories which starts with 'f')
$ python find_ex.py --name "f*" --type "d"
f2
folder1

$ python find_ex.py --name "f*" --type "d"
f2
folder1


(show everything which starts with 'f')
$ find -name "f*"
./f2
./file1.txt
./file2.txt
./find_ex.py
./folder1

$ python find_ex.py --name "f*"
f2
file1.txt
file2.txt
find_ex.py
folder1


(show all directories)
$ find -type "d"
.
./f2
./folder1
```

(continues on next page)

(continued from previous page)

```
$ python find_ex.py --type "d"
f2
folder1

(read t* from different location)
$ python find_ex.py ./folder1 --name "t*"
$ python find_ex.py ./f2 --name "t*"
f2/tiger.txt
$ python find_ex.py .. --name "t*"
../text_format.txt
../unix_commands/f2/tiger.txt
```

14.4 grep

- "grep" command is used to find the pattern in the file, e.g. in below code 'Dog' is search in the file 'file2.txt'.

```
$ grep "Dog" file2.txt
Dog nudged cat.
```

- Below is the implementation of 'grep' command using Python

```
# grep_ex.py

import argparse
import re

from pathlib import Path


parser = argparse.ArgumentParser(description="grep command")

# positional argument when defined without --
# nargs='?' is required for default-positional-arg values
parser.add_argument('pattern', type=str, nargs=1)
parser.add_argument('location', type=str, nargs='*')


args = parser.parse_args() # save arguments in args
loc = Path(args.location[0]) # save location
ptrn = args.pattern[0] # save pattern

lines = open(loc).readlines();

for line in lines:
    if re.compile(ptrn).search(line):
        print(line, end="")
```

- The output of 'grep' command are shown below,

```
$ grep "nud" file2.txt
Dog nudged cat.
$ python grep_ex.py "nud" file2.txt
Dog nudged cat.
```

14.5 cat

In this section, we will implement the command 'cat' of unix. We will write a code to read the name of the files from the command line.

- First create few files e.g. 'file1.txt' and 'file2.txt' etc. and add some contents to it.
- Now read these files as shown below, which emulates the functionality of 'unix's cat command',

Listing 14.4: Read files from command line

```
# cat.py

import argparse

parser = argparse.ArgumentParser(description="First argparse code")

# read arguments
parser.add_argument(
        nargs='+', # read multiple values in a list
        dest='files', # name of the attribute returned by argparse
        help='unix "cat" operation')

# line number
parser.add_argument(
        '-n', '--numbers', # default 'dest=numbers'
        action='store_true', help='print line numbers')

args = parser.parse_args() # save arguments in args
# print(f"args: {args.files}") # print argument read by 'dest=files'
# print(f"first element: {args.files[0]}") # print first element of arg.files


# open and read each files
line_no = 0
for file_name in args.files:
    with open(file_name, 'r') as w:
        if args.numbers:  # print data with line numbers
            for data in w.readlines(): # start from 1
                line_no += 1
                print("{0:5d}\t {1}".format(line_no, data), end='')
        else: # print data without line numbers
            data = w.read()
            print(data)
```

Below are the outputs of above code,

- actual 'cat' command output

```
$ cat file1.txt file2.txt -n
    1  This is cat. Cat is pet.
    2  This is cat. Cat is pet.
    [...]
   15  This is cat. Cat is pet.
   16  Dog nudged cat.
   17  Duck is sleeping.
```

- Python code output

```
$ python cat.py file1.txt file2.txt -n
    1   This is cat. Cat is pet.
    2   This is cat. Cat is pet.
    [...]
   15   This is cat. Cat is pet.
```

(continues on next page)

(continued from previous page)

```
16  Dog nudged cat.
17  Duck is sleeping.
```

Note:

- Try both commands without '-n' as well.
- Run below command to see more functionalities of 'cat' command; and try to add some of those to 'cat.py'

```
$ man cat
```

Index