Python3 Simulation with OOP

Meher Krishna Patel

Created on : August, 2015
Last updated : July, 2017

Table of Contents

Believe nothing, no matter where you read it, or who said it, no matter if I have said it, unless it agrees with your own reason and your own common sense.

*–**Buddha***

Chapter 1

Python for simulation

1.1 Aim

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 tutorial, we will do some basic coding using Python, which is essential for mathematical modeling and simulation purposes .

There are various libraries available for Python, which can be used for different applications. In this tutorial, **our aim is to introduce the 'features' and 'libraries' which can be used for mathematical simulations**. Further, Object Oriented Programming (OOP) features of Python are discussed from Chapter 6. The classes in the OOP concepts can make codes more reusable and manageable than the simple procedural methods. Procedural methods or functions are discussed in Section 1.7; whereas OOPs features along with the libraries (i.e. Numpy, Scipy, Cython and Matplotlib) are discussed in later chapters. Note that, the system must have C-compiler for executing Cython codes. In Linux, it can be installed using command '**sudo apt-get install g++**'; and for Windows, please download and install the C-compiler (with default settings) from website 'Visual C++ Build Tools'.

All the codes can be downloaded from the website. First line of each listing in the tutorial, is the name of the python file in the downloaded zip-folder.

1.2 Download and Installation

We 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. For executing codes in the tutorial, we need to install Python along with following libraries,

1. Python 3
2. NumPy
3. SciPy
4. Matplotlib
5. SPYDER (**S**cientific **Py**thon **D**evelopment **E**nvi**r**onment)

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 all the required python libraries for this tutorial.

1.3 Writing first code

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 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.

Listing 1.1: Hello World

```
1  C:\>python
2  >>> print("Hello World")
3  Hello World
4  >>>print("Hello %s, simulation can be done using %s." % ("World", "Python"))
5  Hello World, simulation can be done using Python.
6  >>>
```

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

```
1  # hello.py: prints "Hello World"
2  # save this file to any location e.g. C:\>PythonCodes
3  print("Hello World")
4
5  ## following line will be error because of extra space in the beginning
6  #   print("Hello World with spaces is error") # error
```

Listing 1.3: Run hello.py

```
C:\>cd PythonCodes
C:\PythonCodes>python hello.py
Hello World
C:\PythonCodes>
```

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.

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()

Table 1.1: Built-in object types

1.4 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

```
 1  #variableEx.py: a and b are the variables
 2  # which stores 3 and 5 respectively
 3  a=3
 4  b=6
 5
 6  # following line will add value of a i.e. 3 with 6
 7  print(a+6) # 9
 8
 9  # following line will perform (3+5)/2
10  print((a+b)/2) # 4.5
11
12  # following line will perform (3+5)/2
13  # and then display the interger value only
14  print(int((a+b)/2)) # 4
```

1.5 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.

1.5.1 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

```

```
c = 0o17 # octal: '0o' is used before number i.e. 17
# print command shows the integer value
print(c) # 15
#to see octal form use `oct'
print(oct(c)) #0o17

e = 3+2j # imagenary
print(e) #(3+2j)
print(abs(e)) #3.6055512754639896
# round above value upto 2 decimal
r=round(abs(e),2)
print(r) #3.61
```

1.5.2 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
```

1.5.3 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]
```

1.5.4 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

```python
#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
```

1.5.5 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

```python
#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'}
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'}
```

1.5.6 File

File can be very useful to store the results obtained by simulations. For example, if we want to compare and plot the results of two different simulations, then it is necessary to store the results in some files. Then store results can be used for plotting the data. Files are not discussed in the tutorial, as code can be very complicated (see Appendix A) to store various types of data obtained by the simulations, e.g. integer, complex numbers and arrays etc. Further, in section 1.9, we discuss the SPYDER environment to store data in the file.

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

Table 1.2: Symbols for conditions

1.6 Control structure

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

1.6.1 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*.

1.6.1.1 If statement

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

Example 1.1 (Single If statement) 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,
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.*

▲

Example 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.* ▲

1.6.1.2 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.

Example 1.3 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.* ▲

1.6.1.3 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.

Example 1.4 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 '''
22 output 1:
23 Enter the number: 12
24 Number is divisible by 2 and 3
25 Thank you
26
27 output 2:
28 Enter the number: 8
29 Number is divisible by 2 only
30 x%3=  2
31 Thank you
32
33 output 3:
34 Enter the number: 7
35 Number is not divisible by 2 and 3
36 x%2=  1
37 x%3=  1
38 Thank you
39
40 output 4:
41 Enter the number: 15
42 Number is divisible by 3 only
43 Thank you
44 '''
```

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 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

```python
#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")
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

```python
#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")
```

1.6.2 While loop

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

Example 1.5 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

```python
 1  #WhileExample1.py: Print numbers upto 5
 2
 3  n=1 #initial value of number
 4  print("Numbers upto 5: ")
 5  while n<6:
 6      print(n, end=" "), #end=" ": to stop row change after print
 7      n=n+1
 8  print("\nCode ended at n =  %s" % n)
 9
10  '''
11  output:
12  Numbers upto 5:
13  1 2 3 4 5
```

```
14  Code ended at n =  6
15  '''
```

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. ▲

1.6.3 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 function
2
3   >>> range(5)
4   [0, 1, 2, 3, 4]
5
6   >>> range(1,4)
7   [1, 2, 3]
8
9   >>> range(11, 19, 2)
10  [11, 13, 15, 17]
11
12  >>> range(15, 7, -2)
13  [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 "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. ▲

Example 1.6 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:
15  Numbers in forward order
16  1 2 3 4 5
17  Final value of i is:  4
18
19  Numbers in reverse order
20  5 4 3 2 1
21  Final value of i is:  1
22  '''
23
24  fruits=["apple", "banana", "orange"]
25  print("List of fruits are shown below:")
26  for i in fruits:
27      print(i)
28  '''
29  List of fruits are shown below:
30  apple
31  banana
32  orange
33  '''
```

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. ▲

1.7 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.

Example 1.7 In Listing 1.19, the function 'addTwoNum' adds two numbers.

Listing 1.19: Function

```
 1  #funcEx.py
 2  def addTwoNum(a, b):
 3    sum = a+b
 4    return(sum)
 5
 6  result = addTwoNum(3,4)
 7  print("sum of numbers is %s" % result)
 8
 9  '''
10  sum of numbers is 7
11  '''
```

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. ▲

Example 1.8 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.* ▲

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.

1.8 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.

Example 1.9 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

```
#numpyMatplot.py
import numpy as np
import matplotlib.pyplot as plt
# np.linspace: devide line from 0 to 4*pi into 100 equidistant points
x = np.linspace(0, 4*np.pi, 100)
sinx = np.sin(x) #find sin(x) for above 100 points
plt.plot(x,sinx) # plot (x, sin(x))
plt.xlabel("Time") # label for x axis
plt.ylabel("Amplitude") # label for y axis
plt.title('Sine wave') # title
```

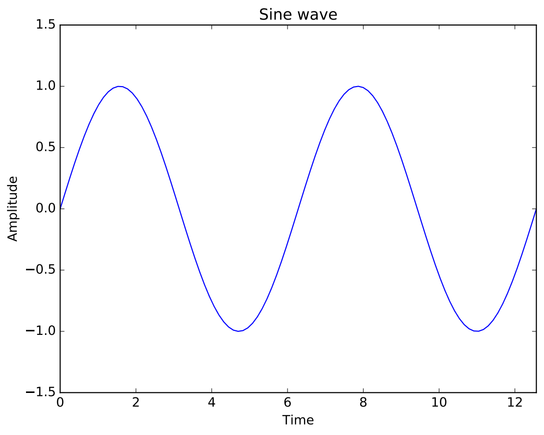


Figure 1.1: Sine wave using Numpy and Matplotlib, Listing 1.21

```
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.* ▲

1.8.1 Arrays

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

1.8.1.1 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]
```

1.8.1.2 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

```

```python
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])]]
'''
```

1.8.2 Matrix

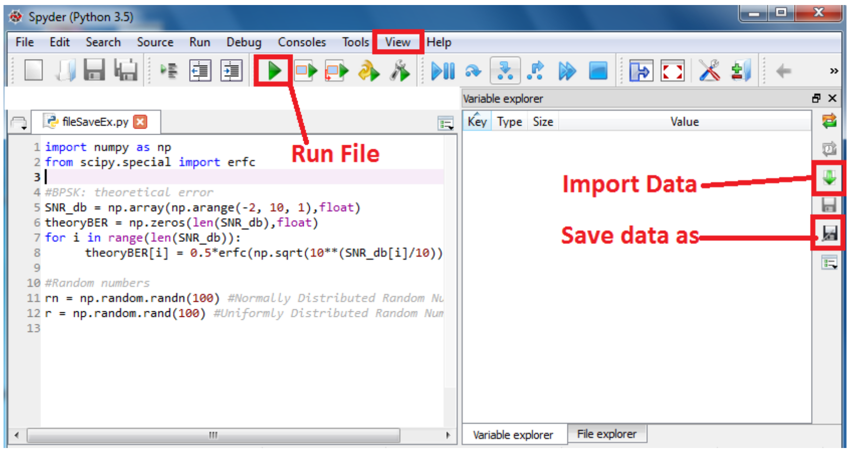
Similar to array, we can define matrix using 'mat' function of numpy library as shown in Listing 1.24. Also, LU factorization of the matrix is shown in Listing 1.25 using Scipy library. There are differences in results of mathematical operations on the matrices defined by 'array' and 'mat', as shown 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

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

```python
#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)))
```



```
import numpy as np
from scipy.special import erfc

#BPSK: theoretical error
SNR_db = np.array(np.arange(-2, 10, 1),float)
theoryBER = np.zeros(len(SNR_db),float)
for i in range(len(SNR_db)):
      theoryBER[i] = 0.5*erfc(np.sqrt(10**(SNR_db[i]/10))

#Random numbers
rn = np.random.randn(100) #Normally Distributed Random Nu
r = np.random.rand(100) #Uniformly Distributed Random Num
```

Figure 1.2: SPYDER Environment

```
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.  ]
 [ 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.]]
'''
```

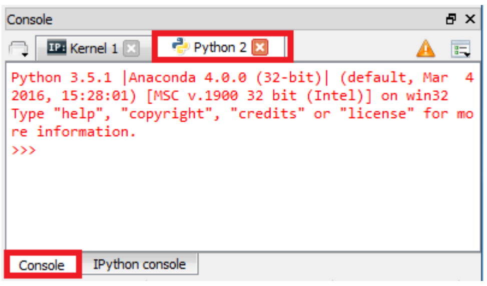
1.9 SPYDER

Saving simulation results using python code can be quite complicated. Further, reading data from those files is even more complicated as shown in Appendix A.

This problem can be easily solved by using SPYDER environment. SPYDER can save the data in various formats, but '.spydata' is the preferred format. Further, the results can be saved in '.mat' format as well, which can be used by MATLAB software.

Fig. 1.2 shows the SPYDER window. On the left side of the window, actual code is shown. From the view option (at the top), you can add more toolbars and panes to your window. 'Import Data' and 'Save data as' buttons (on the right side) are used to save and read the data as shown next.

We can write/run all the codes in previous sections using SPYDER environment. Further, it has auto-completion option as well; e.g. if we write 'import numpy as np', then from next line, after writing 'np.', it will display all the possible commands available in the library.



```
Python 3.5.1 |Anaconda 4.0.0 (32-bit)| (default, Mar  4
2016, 15:28:01) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for mo
re information.
>>>
```

Figure 1.3: Python Console

1.9.1 Python/IPython Console

Note that, if you want to publish the results in research journals, then editors may ask for editable figure files e.g. .ps or .eps formats; often, they do not accept .jpg or .png formats. Python console, by default, generates figure in separate window outside the console; and then figure can be saved in various formats e.g. PDF, .eps and .ps etc. Also, we can change the plot style before saving the plots. Whereas, if you try to plot the figure using IPython console, it will generate the figure inside the console which can be saved in .png format only. We can change this behavior from **Tools→Preferences→IPython Console→Graphics backends→QT**, and restarting the console. After this setting, figure will be displayed in separate window.

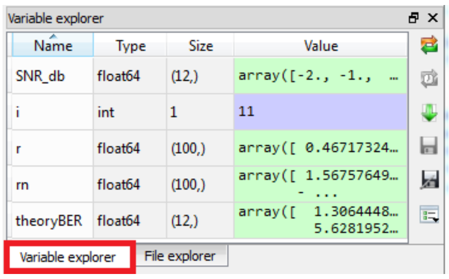
To open a ‘Python console’, click on ‘Consoles → Open a Python console’ at the top of SPYPER window. It will open the console as shown in Fig. 1.3 where console is open with name ‘Python 2’. You may get different number there.

Note that, console window may contain various Python/IPython console simultaneously; but the console which appear on that window is the active console e.g. in Fig. 1.3, ‘Python 2’ is active console.

1.9.2 Save Data

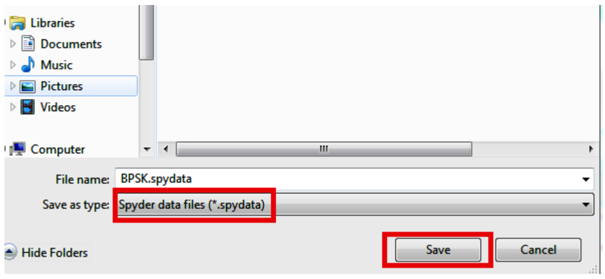
First open the code in Listing 1.26 in SPYDER environment as shown in Fig. 1.2; and then press the ‘Run’ button. The code will execute and all the variables will be displayed in the ‘variable explorer’ e.g. SNR_db etc. as shown in Fig. 1.4a. Now click on the ‘save data as’ button to save it as .spydata file with name ‘BPSK’, as shown in Fig. 1.4b.

Although, SPYDER can save data in ‘.mat’ file as well, but it creates problems while reading it. Therefore, it is better to save data in ‘.spydata’ file.



Name	Type	Size	Value
SNR_db	float64	(12,)	array([-2., -1., ...
i	int	1	11
r	float64	(100,)	array([0.46717324...
rn	float64	(100,)	array([1.56757649... - ...
theoryBER	float64	(12,)	array([1.3064448... 5.6281952...

(a) Data generated after running code



File name: BPSK.spydata
Save as type: Spyder data files (*.spydata)

(b) Save data in .spydata file

Figure 1.4: Run code and save data in .spydata file

Listing 1.26: BER of BPSK system and Random Numbers

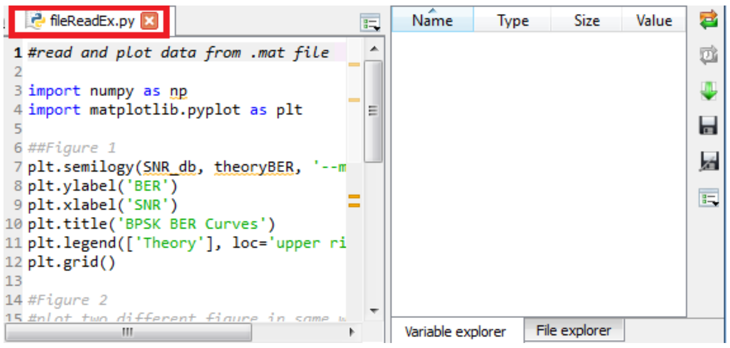
```
1  #fileSaveEx.py
2  import numpy as np
3  from scipy.special import erfc
4  #BPSK: theoretical error
5  SNR_db = np.array(np.arange(-2, 10, 1)) #SNR in db
6  SNR = 10**(SNR_db/10) # SNR_db to SNR conversion
7
8  theoryBER = np.zeros(len(SNR_db),float) # zero vector of size SNR_db
9
10 for i in range(len(SNR_db)):
11   theoryBER[i] = 0.5*erfc(np.sqrt(SNR[i]))
12
13 #Random numbers
14 rn = np.random.randn(100) #Normally Distributed Random Numbers
15 r = np.random.rand(100) #Uniformly Distributed Random Numbers
```

Explanation (Listing 1.26). *Line 11 in Listing 1.26, is the implementation of following equation for different SNR values,*

$$theoryBER = \frac{1}{2}erfc\left(\sqrt{SNR}\right) \tag{1.1}$$

Further, line 14 and 15 generates random variables with different distribution. ▲

1.9.3 Read Data



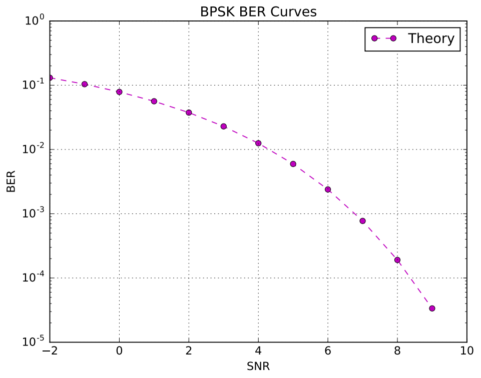
```
#read and plot data from .mat file

import numpy as np
import matplotlib.pyplot as plt

##Figure 1
plt.semilogy(SNR_db, theoryBER, '--m
plt.ylabel('BER')
plt.xlabel('SNR')
plt.title('BPSK BER Curves')
plt.legend(['Theory'], loc='upper ri
plt.grid()

#Figure 2
#plot two different figure in same w
```

Figure 1.5: Load the BPSK.spyderdata file before running this code

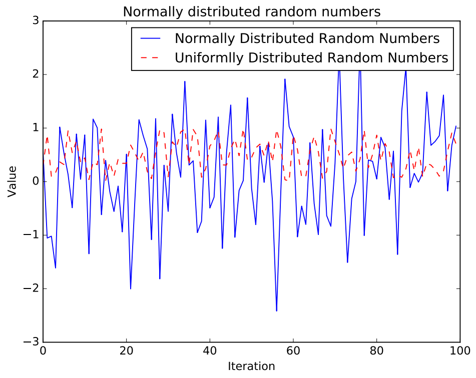
First open the code in Listing 1.27 as shown in Fig. 1.5. After that click on the 'import data' button on variable-explorer and open the 'BPSK.spydata' file, which we saved in previous section. Variable-explorer will again display the stored data as in Fig 1.4a. Now press the 'Run' button again, and Listing 1.27 will execute and plot the data stored in .spydata file as shown in Fig. 1.6.

Listing 1.27: Code for plotting the saved data

```
#fileReadEx.py: read and plot data from .mat file
import numpy as np
import matplotlib.pyplot as plt

##Figure 1
# SNR_db on x-axis, theoryBER on y-axis
#--mo: plot in magenta color with 'o' marker
plt.semilogy(SNR_db, theoryBER, '--mo')
plt.ylabel('BER')
plt.xlabel('SNR')
plt.title('BPSK BER Curves')
plt.legend(['Theory'], loc='upper right')
plt.grid()

#Figure 2
#plot two different figure in same window,
#i.e. NormalrandomNumber and RandomNumber
plt.figure() # open new figure window
```



(a) theoryBER: BPSK



(b) rn, r: Random numbers

Figure 1.6: Plots from the data stored in BPSK.spydata

```
plt.plot(rn)
plt.plot(r, '--r') # '--r': plot in red color with broken line
plt.ylabel('Value')
plt.xlabel('Iteration')
plt.title('Normally distributed random numbers')
plt.legend(
  ['Normally Distributed Random Numbers',
  'Uniformlly Distributed Random Numbers'],
  loc='upper right'
  )

#Display all figures
plt.show()
```

1.10 Conclusion

In this chapter, we saw various features of Python 3 language. Further, we used three libraries i.e. Numpy, Scipy and Matplotlib. These libraries are very useful for performing mathematical simulations. Also, we used SPYDER environment to save the data and then saved data is plotted using Matplotlib library.

May be you are searching among the branches, for what only appears in the roots.

*–**Rumi***

Chapter 2

Plotting Data with Matplotlib

2.1 Aim

In this tutorial, Matplotlib library is discussed in detail, which is used for plotting the data. Our aim is to introduce the commonly used 'plot styles' and 'features' of the Matplotlib library, which are required for plotting the results obtained by the simulations.

2.2 Data generation with Numpy

In this tutorial, Numpy library is used to generate the data for plotting. For this purpose, only 5 functions of numpy library are used, which are shown in Listing 2.1,

Listing 2.1: Data generation using Numpy

```
#numpyDataEx.py
# import numpy library with short name `np'
# outputs are shown as comments
import numpy as np

# 1. linspace(a, b, total_points)
x = np.linspace(2, 8, 4)
print(x) # [ 2.  4.  6.  8.]

# 2. sin(x)
sinx = np.sin(x) # x is consider as radian
print(sinx) # [ 0.90929743 -0.7568025  -0.2794155   0.98935825]

# 3. cos(x)
cosx = np.cos(x)
print(cosx) #[-0.41614684 -0.65364362  0.96017029 -0.14550003]

# 4. rand: uniform random variables
ur = np.random.rand(4) # result below will be different as it is random
print(ur) # [ 0.99791448  0.95621806  0.48124676  0.20909043]

# 5. randn: normal random variables
nr = np.random.randn(4) # result below will be different as it is random
print(nr) # [-0.37188868 -0.5680135  -0.21731407 -0.69523557]
```

2.3 Basic Plots

In this section, basic elements of the plot e.g. Title, axis names and grid etc. are discussed.

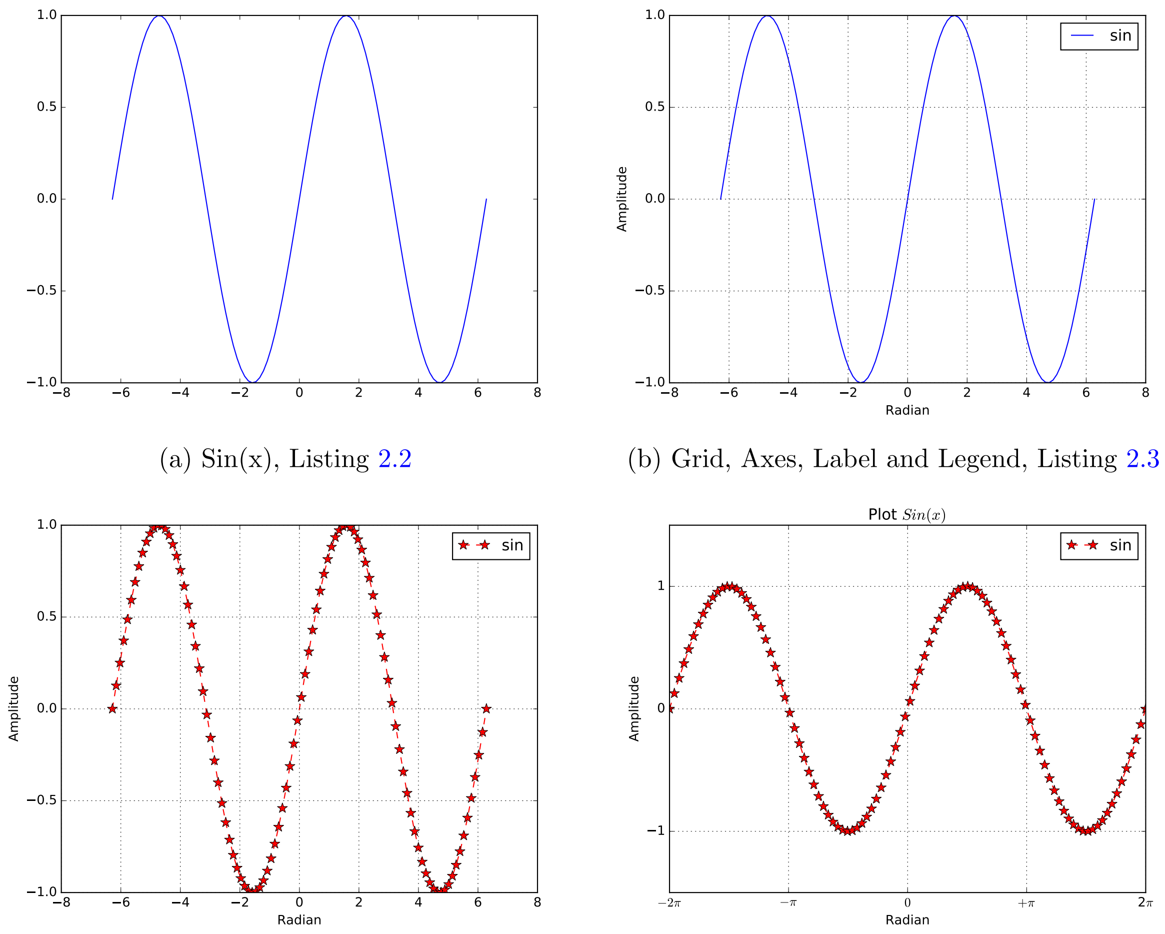
2.3.1 First Plot

Listing 2.2 plots the sin(x) as shown in Fig. 2.1a.

Listing 2.2: Sin(x), Fig. 2.1a

```
1   #firstPlot.py
2   import numpy as np
3   import matplotlib.pyplot as plt
4
5   # close all the figures, if open from previous commands
6   plt.close('all')
7
8   x=np.linspace(-2*np.pi, 2*np.pi, 100)
9   sinx=np.sin(x) # calculate sin(x)
10  plt.plot(x,sinx) #plot x on x-axis and sin_x on y-axis
11  plt.show() #display the plot
```

Explanation (Listing 2.2). *Here, line 8 generates 100 equidistant points in the range [-2π, 2π]. Then line 9 calculates the sine values for those points. Line 10 plots the figure, which is displayed on the screen using line 11.* ▲



(a) Sin(x), Listing 2.2

(b) Grid, Axes, Label and Legend, Listing 2.3

(c) Line-style and Line-marker, Listing 2.4

(d) Axis-limit and Axis-marker, Listing 2.5

Figure 2.1: Sin(x) Plots

2.3.2 Label, Legend and Grid

Here Listing 2.2 is modified as shown in Listing 2.3, to add labels, legend and grid to the plot.

Listing 2.3: Grid, Label and Legend, Fig. 2.1b

```
#firstPlotGL.py
import numpy as np
import matplotlib.pyplot as plt
```

```
 4
 5  # close all the figures, if open from previous commands
 6  plt.close('all')
 7
 8  ############ Sin(x) ####################
 9  x=np.linspace(-2*np.pi, 2*np.pi, 100)
10  sinx=np.sin(x) # calculate sin(x)
11  plt.plot(x,sinx, label='sin')
12
13  ############ Legend ####################
14  # label in plt.plot are displayed by legend command
15  plt.legend(loc="best") #show legend
16
17  #### Lable and Grid ####################
18  plt.xlabel("Radian") # x label
19  plt.ylabel("Amplitude") # y label
20
21  plt.grid() # show grid
22
23  plt.show() #display the plot
```

Explanation (Listing 2.3). *In line 11, "label='sin' " is added which is displayed by 'legend' command in line 15. "loc=best" is optional parameter in line 15. This parameter find the best place for the legend i.e. the place where it does not touch the plotted curve. Line 18 and 19 add x and y label to curves. Finally, line 21 adds the grid-lines to the plot.*

For changing the location of legend, replace 'best' in line 15 with 'center', 'center left', 'center right', 'lower center', 'lower left', 'lower right', 'right', 'upper center', 'upper left' or 'upper right'. ▲

2.3.3 Line style and Marker

It is good to change the line styles and add markers to the plots with multiple graphs. It can be done as shown in Listing 2.4.

Listing 2.4: Line Style and Marker, Fig. 2.1c

```
 1  #firstPlotLineMarker.py
 2  import numpy as np
 3  import matplotlib.pyplot as plt
 4
 5  # close all the figures, if open from previous commands
 6  plt.close('all')
 7
 8  ############ Sin(x) ####################
 9  x=np.linspace(-2*np.pi, 2*np.pi, 100)
10  sinx=np.sin(x) # calculate sin(x)
11
12  ########### Line style and Marker ###################
13  plt.plot(x,sinx, '*--r', markersize=10, label='sin')
14
15  ############ Legend ####################
16  plt.legend(loc="best") #show legend
17
18  #### Lable and Grid ####################
19  plt.xlabel("Radian") # x label
20  plt.ylabel("Amplitude") # y label
21  plt.grid() # show grid
22
23  plt.show() #display the plot
```

Explanation (Listing 2.4). *In line 13, '*- -r' is the combination of three separate parameters i.e. '*', '- -' and 'r', which represents 'marker', 'line style' and 'color' respectively. We can change the order of these combinations e.g. 'r- -*' and '- -r*' etc. Also, combination of two parameters (e.g. 'r- -') or single parameter (e.g. '- -') are valid.*

Table 2.1, 2.2 and 2.3 show some more abbreviations for 'line style', 'marker style' and 'color' respectively. Note that, only one element can be chosen from each style to make the combination.

Further, line 13 contains 'markersize' parameter for changing the size of the marker. Table 2.4 shows the complete list of additional parameters to change the plot style. Lastly, line 13 can be rewritten using complete features of the plot as follows,

Line	Abbreviation
dash-dot line	-.
dashed line	--
dotted line	:
solid line	-

Table 2.1: Line Style

Marker	Abbreviation
Circle marker	o
Cross (x) marker	x
Diamond marker	D
Hexagon marker	h
Pentagon marker	p
Plus marker	+
Point marker	.
Square marker	s
Star marker	*
Triangle down marker	v
Triangle left marker	<
Triangle right	>
Triangle up marker	^

Table 2.2: Marker Style

Color	Abbreviation
black	k
blue	b
cyan	c
green	g
magenta	m
red	r
white	w
yellow	y

Table 2.3: Color Style

Keyword	Description
color	Set the color of the line
linestyle	Sets the line style
linewidth	Sets the line width
marker	Sets the line marker style
markeredgecolor	Sets the marker edge color
markeredgewidth	Sets the marker edge width
markerfacecolor	Sets the marker face color
markersize	Sets the marker size

Table 2.4: Plot Style

plt.plot(x, sinx, color='m', linestyle='-.', linewidth=4, marker='o', markerfacecolor='k', markeredgecolor='g', markeredgewidth=3, markersize=5, label='sin') ▲

2.3.4 Axis and Title

Listing 2.5 adds axis and title in previous figures.

Listing 2.5: Title and Axis, Fig. 2.1d

```
 1  #completeBasicEx.py
 2  import numpy as np
 3  import matplotlib.pyplot as plt
 4
 5  # close all the figures, if open from previous commands
 6  plt.close('all')
 7
 8  ############ Sin(x) ####################
 9  x=np.linspace(-2*np.pi, 2*np.pi, 100)
10  sinx=np.sin(x) # calculate sin(x)
11
12  ########### Line style and Marker ###################
13  plt.plot(x,sinx, '*--r', markersize=10, label='sin')
14
15  ############ Legend ####################
16  plt.legend(loc="best") #show legend
17
18  #### Lable and Grid ####################
19  plt.xlabel("Radian") # x label
20  plt.ylabel("Amplitude") # y label
21  plt.grid() # show grid
22
23  ############ Axis Limit and Marker ##################
24  # x-axis and y-axis limits
25  plt.xlim([-2*np.pi, 2*np.pi]) # x-axis display range
26  plt.ylim([-1.5, 1.5]) # y-axis display range
27
28  # ticks on the axis
29  # display ticks in pi format rather than 3.14 format
30  plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
31            [r'$-2\pi$', r'$-\pi$', r'$0$', r'$+\pi$', r'2$\pi$'])
32  plt.yticks([-1, 0, +1])
33
34  ################## Title #########################
35  plt.title("Plot $Sin(x)$")
36
37  plt.show()
```

Explanation (Listing 2.5). *Lines 25 and 26 in the listing add display range for x and y axis respectively in the plot as shown in Fig. 2.1d. Line 30 and 32 add the ticks on these axis. Further, line 31 adds various 'display-name' for the ticks. It changes the display of ticks e.g. 'np.pi' is normally displayed as '3.14', but "r' + \pi" will display it as* $+\pi$. *Note that '\pi' is the "latex notation" for* π, *and matplotlib supports latex notation as shown in various other examples as well in the tutorial.* ▲

2.4 Multiple Plots

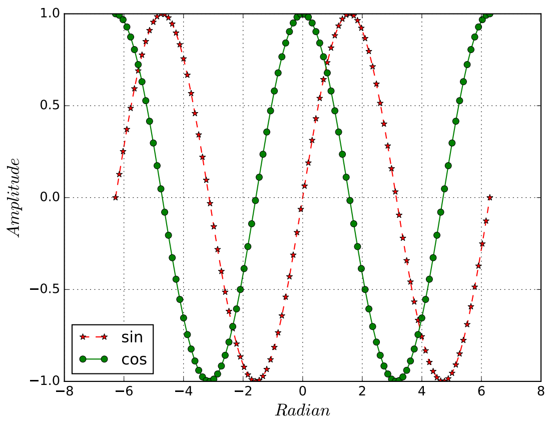
In this section various mutliplots are discussed e.g. plots on the same figure window and subplots etc.

2.4.1 Mutliplots in same window

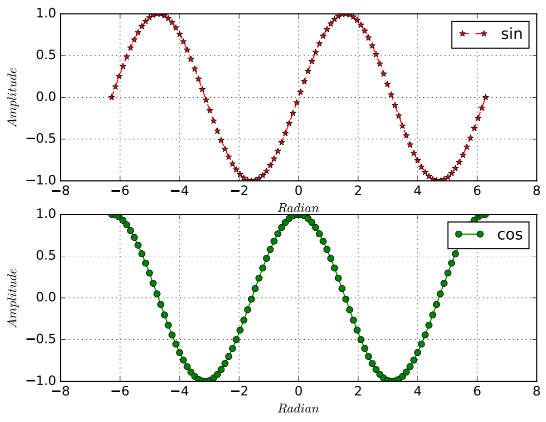
By default, matplotlib plots all the graphs in the same window by overlapping the figures. In Listing 2.6, line 14 and 15 generate two plots, which are displayed on the same figure window as shown in Fig. 2.2a.

Listing 2.6: Mutliplots in same window, Fig. 2.2a

```
#multiplot.py
import numpy as np
import matplotlib.pyplot as plt

# close all the figures, if open from previous commands
plt.close('all')

############ Sin(x) ####################
x=np.linspace(-2*np.pi, 2*np.pi, 100)
sinx=np.sin(x) # calculate sin(x)
cosx=np.cos(x) # calculate cos(x)

########### Line style and Marker ###################
plt.plot(x,sinx, '*--r', label='sin')
plt.plot(x,cosx, 'o-g', label='cos')

############ Legend ####################
plt.legend(loc="best") #show legend

#### Lable and Grid ####################
plt.xlabel(r'$Radian$').set_fontsize(16) # x label
plt.ylabel(r'$Amplitude$').set_fontsize(16) # y label
plt.grid() # show grid

plt.show()
```



(a) Multiplots, Listing 2.6



(b) Subplots, Listing 2.7

Figure 2.2: Multiple plots

2.4.2 Subplots

In Fig. 2.2a, two plots are displayed in the same window. Further, we can divide the plot window in multiple sections for displaying each figure in different section as shown in Fig. 2.2b.

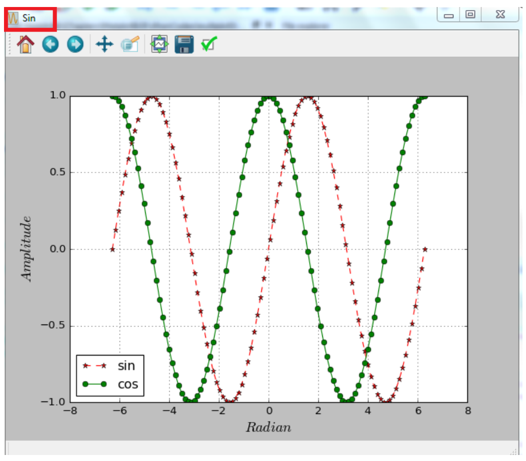
Listing 2.7: Subplots, Fig. 2.2b

```
#subplotEx.py
import numpy as np
import matplotlib.pyplot as plt

# close all the figures, if open from previous commands
plt.close('all')

############ Sin(x) ####################
x=np.linspace(-2*np.pi, 2*np.pi, 100)
sinx=np.sin(x) # calculate sin(x)
cosx=np.cos(x) # calculate cos(x)

########### Subplot ###################
plt.subplot(2,1,1)
```

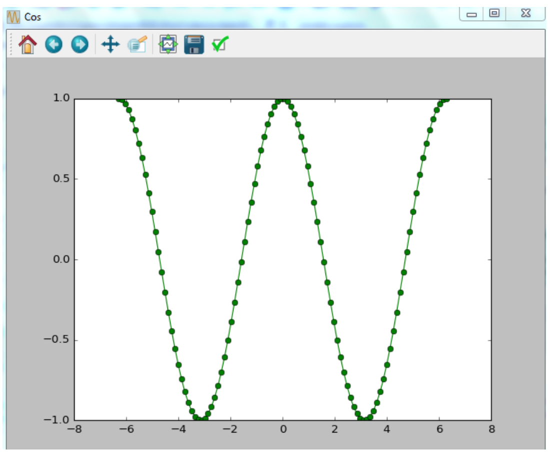
```
15  plt.plot(x,sinx, '*--r', label='sin')
16  plt.grid() # show grid
17  plt.legend() #show legend
18  plt.xlabel(r'$Radian$')# x label
19  plt.ylabel(r'$Amplitude$') # y label
20
21  plt.subplot(2,1,2)
22  plt.plot(x,cosx, 'o-g', label='cos')
23  plt.grid() # show grid
24  plt.legend() #show legend
25  plt.xlabel(r'$Radian$') # x label
26  plt.ylabel(r'$Amplitude$') # y label
27  ############ Legend ####################
28
29  #### Lable and Grid ####################
30  plt.xlabel(r'$Radian$') # x label
31  plt.ylabel(r'$Amplitude$') # y label
32
33
34  plt.show()
```

Explanation (Listing 2.7). *Subplot command takes three parameters i.e. number of rows, numbers of columns and location of the plot. For example in line 14, subplot(2,1,1), divides the figure in 2 rows and 1 column, and uses location 1 (i.e. top) to display the figure. Similarly, in line 21, subplot(2,1,2) uses the location 2 (i.e. bottom) to plot the graph. Further, Listing 2.10 divides the figure window in 4 parts and then location 1 (top-left), 2 (top-right), 3 (bottom-left) and 4 (bottom-right) are used to display the graphs.*

Also, all the plot commands between line 14 and 21, e.g. line 15, will be displayed on the top location. Further, plots defined below line 21 will be displayed by bottom plot window. ▲



(a) Figure window with name "Sin"



(b) Figure window with name "Cos"

Figure 2.3: Open figures in separate windows, Listing 2.8

2.4.3 Mutliplots in different windows

'figure()' command is used to plot the graphs in different windows, as shown in line 17 of Listing 2.8.

Listing 2.8: Mutliplots in different windows, Fig. 2.3

```
1  #multiplotDifferentWindow.py
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # close all the figures, if open from previous commands
6  plt.close('all')
7
8  ############ Sin(x) ####################
9  x=np.linspace(-2*np.pi, 2*np.pi, 100)
```

```
10 sinx=np.sin(x) # calculate sin(x)
11 cosx=np.cos(x) # calculate cos(x)
12
13 ########### Open figure in seperate window ###################
14 #"Sin" is the name of figure window
15 # if not give i.e. plt.figure(), then 1 will be assigned to figure
16 # and for second plt.figure(), 2 will be assigned...
17 plt.figure("Sin")
18 plt.plot(x,sinx, '*--r', label='sin')
19 plt.figure("Cos")
20 plt.plot(x,cosx, 'o-g', label='cos')
21 plt.figure("Sin")
22 plt.plot(x,cosx, 'o-g', label='cos')
23
24 ############ Legend ####################
25 plt.legend(loc="best") #show legend
26
27 #### Lable and Grid ####################
28 plt.xlabel(r'$Radian$').set_fontsize(16) # x label
29 plt.ylabel(r'$Amplitude$').set_fontsize(16) # y label
30 plt.grid() # show grid
31
32 plt.show()
```

Explanation (Listing 2.8). *Here optional name "Sin" is given to the plot which is displayed on the top in Fig. 2.3. Then line 19 opens a new plot window with name "Cos". Finally in line 21, the name "Sin" is used again, which selects the previously open "Sin" plot window and plot the figure there. Hence, we get two plots in this window (i.e. from lines 18 and 22) as show in Fig. 2.3.* ▲

2.5 Semilog Plot

Semilog plots are the plots which have y-axis as log-scale and x-axis as linear scale as shown in Fig. 2.4b. Listing 2.9 plots both the semilog and linear plot of the function e^x.

Listing 2.9: Semilog plot Fig. 2.4

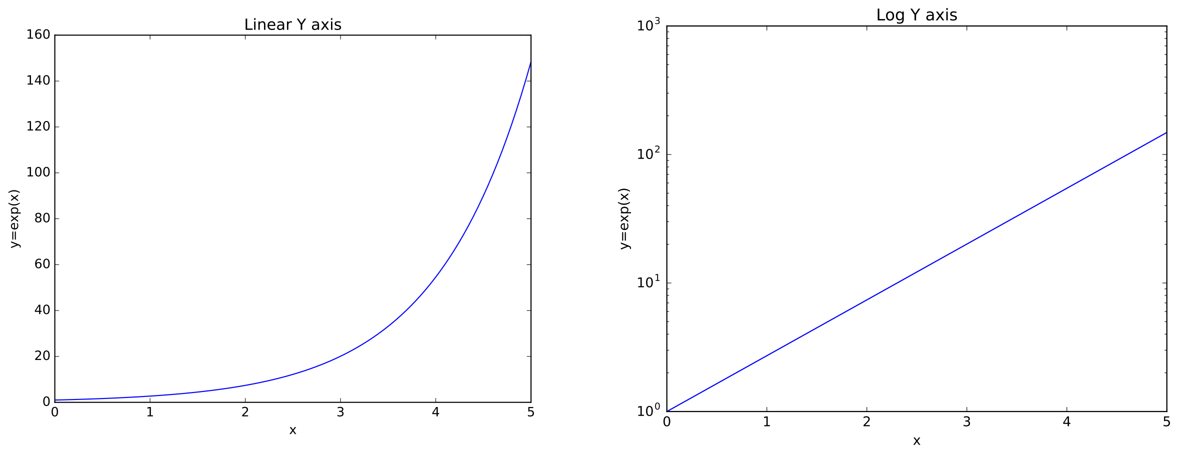
```
#semilogEx.py
import numpy as np
import matplotlib.pyplot as plt

# close all the figures, if open from previous commands
plt.close('all')

x=np.linspace(0.01, 5, 100)
e=np.exp(x)

#linear plot
plt.plot(x,e)
plt.xlabel("x")
plt.ylabel("y=exp(x)")
plt.title("Linear Y axis")

#semilog plot
#log(exp(x))=x therefore straight line will be displayed
plt.figure()
plt.semilogy(x,e) #semilogy: semilog y-axis
plt.xlabel("x")
plt.ylabel("y=exp(x)")
plt.title("Log Y axis")

plt.show() #display the plot
```

2.6 Plot types

In this section, various plot types are discussed.

2.6.1 Histogram

Histogram can be generated using hist() command as illustrated in line 11 in Listing 2.10. By default it generates 10 bins, which can be increased by providing the number of bins as shown



(a) Linear Plot

(b) Semilog Plot

Figure 2.4: Semilog Plot vs Linear Plot 2.9

in line 15. Further from Fig. 2.5, we can see that 'rand' generates the random number in the range [0,1] with uniform density, whereas 'randn' generates the random number in the range [-1,1] with Gaussian (Normal) density.

Listing 2.10: Histogram, Fig. 2.5

```
1   #histogramEx.py
2   import numpy as np
3   import matplotlib.pyplot as plt
4
5   plt.close("all")
6
7   ur = np.random.rand(10000)
8   nr = np.random.randn(10000)
9
10  plt.subplot(2,2,1)
11  plt.hist(ur)
12  plt.xlabel("Uniform Random Number, Default 10 Bin")
13
14  plt.subplot(2,2,2)
15  plt.hist(ur, 20) # display 20 bins
16  plt.xlabel("Uniform Random Number, 20 Bin")
17
18  plt.subplot(2,2,3)
19  plt.hist(nr)
20  plt.xlabel("Normal Random Number, Default 10 Bin")
21
22  plt.subplot(2,2,4)
23  plt.hist(nr, 20) # display 20 bins
24  plt.xlabel("Normal Random Number, 20 Bin")
25
26  plt.show()
```

2.6.2 Scatter plot

Scatter plots are similar to simple plots and often use to show the correlation between two variables. Listing 2.11 generates two scatter plots (line 14 and 19) for different noise conditions, as shown in Fig. 2.6. Here, the distortion in the sine wave with increase in the noise level, is illustrated with the help of scatter plot.

Listing 2.11: Scatter plot, Fig. 2.6

```
1   #scatterEx.py
2   import numpy as np
3   import matplotlib.pyplot as plt
4
5   plt.close("all")
```

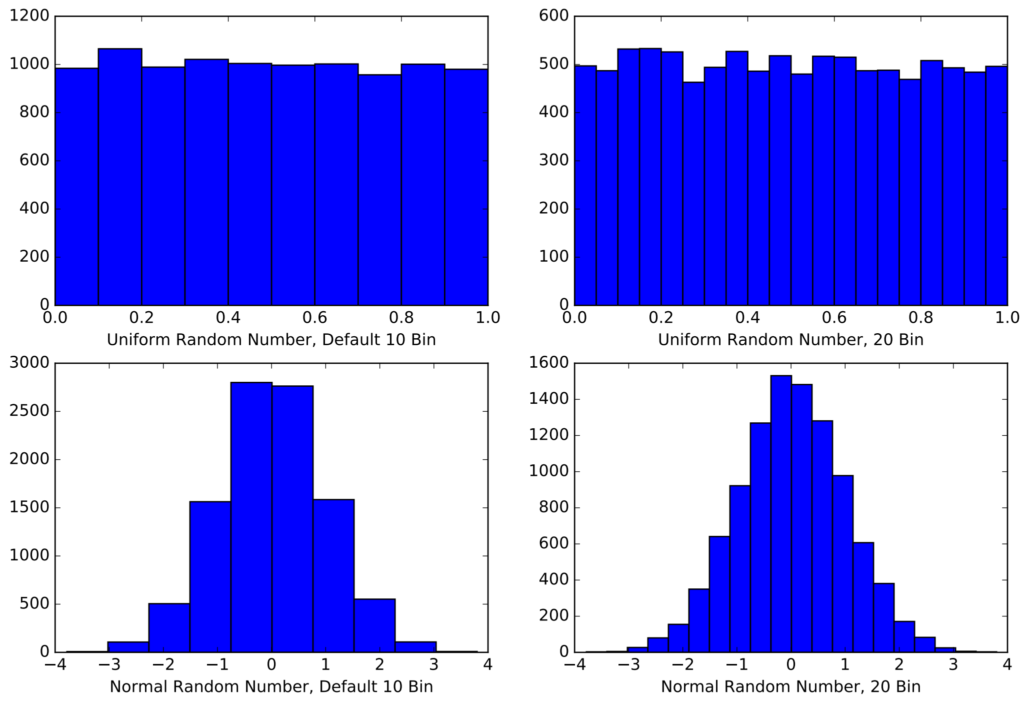


Figure 2.5: Histograms, Listing 2.10

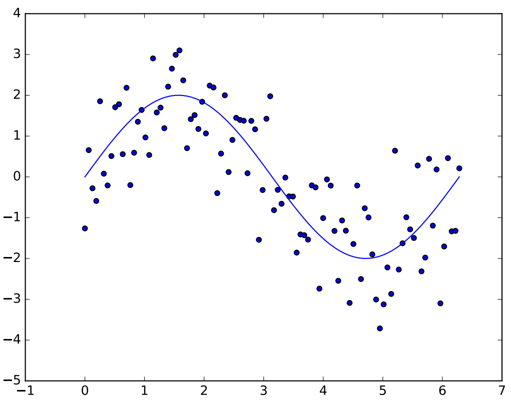
```

N=100
x = np.linspace(0, 2*np.pi, N)
noise = np.random.randn(N)
signal = 2*np.sin(x)

y = signal + noise
plt.plot(x, signal) # signal + noise
plt.scatter(x, y) #scatter plot

plt.figure()
y = signal + 0.2*noise # singal + 0.2*noise i.e. low noise
plt.plot(x, signal)
plt.scatter(x, y) #scatter plot

plt.show()
```

2.6.3 Pie chart

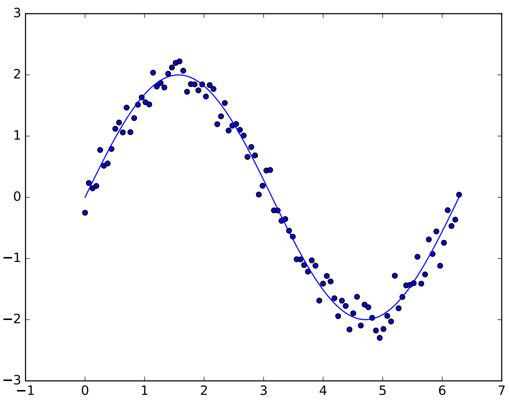
Here, pie charts are generated in two formats. Listing 2.12 generates a simple Pie chart with data names as show in Fig. 2.7a. Also in line 9, figsize=(5,5) command is used here, to resize the output figure window. Further, Listing 2.13 adds additional features (line 13) to it i.e. explode and auto-percentage as shown in Fig. 2.7b.

Listing 2.12: Pie chart, Fig. 2.7a

```
1   #pieEx.py
2   import matplotlib.pyplot as plt
3
4   plt.close("all")
5
6   x = [30, 20, 15, 25, 10]
7   dataName = ['data1', 'data2', 'data3', 'data4', 'data5']
8
9   plt.figure(figsize=(5,5)) #figsize: output figure window size
10  plt.pie(x, labels=dataName)
11
12  plt.show()
```

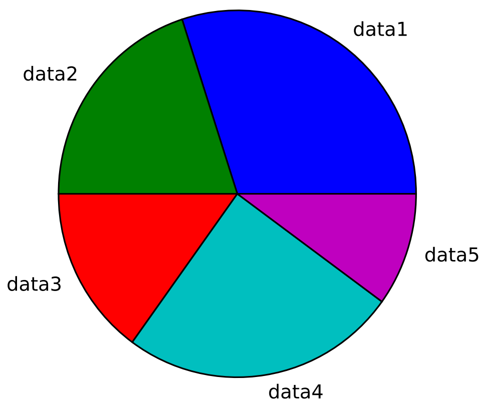


(a) High level of Noise

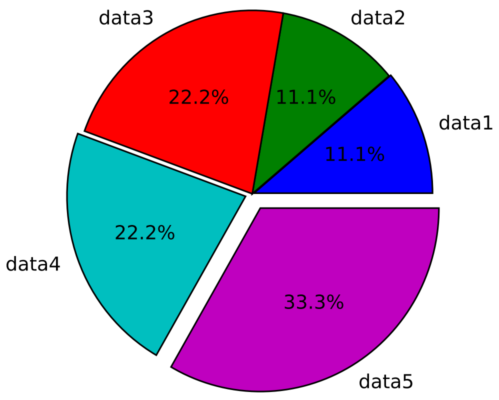


(b) Low level of Noise

Figure 2.6: Scatter Plot, Listing 2.11



(a) Pie Chart with labels, Listing 2.12

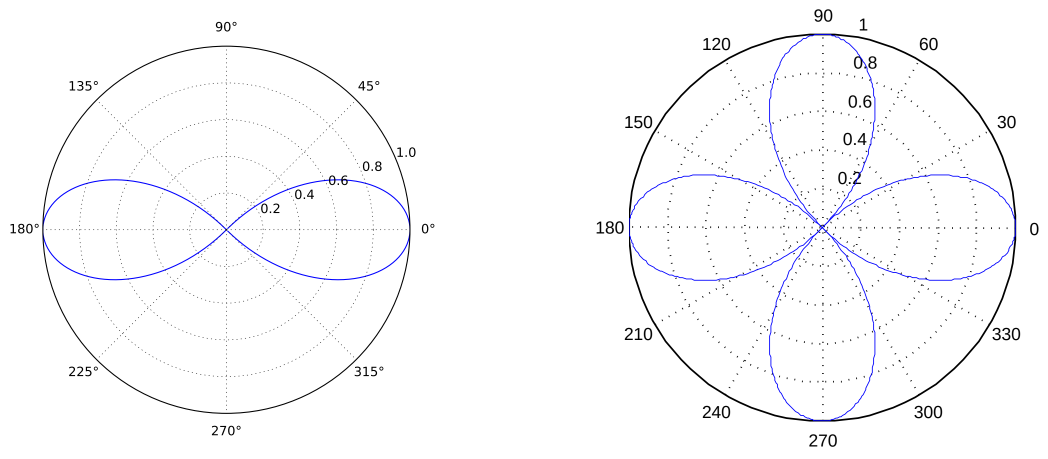


(b) Explode and Auto Percentage, Listing 2.13

Figure 2.7: Pie Chart

Listing 2.13: Pie chart: explode and auto-percentage, Fig. 2.7a

```
#pieEx2.py
import matplotlib.pyplot as plt

plt.close("all")

x = [10, 10, 20, 20, 30]
dataName = ['data1', 'data2', 'data3', 'data4', 'data5']
explode = [0.01, 0, 0, 0.04, 0.09]

plt.figure(figsize=(5,5))
# autopct='%.2f %%': %.2f display value upto 2 decimal,
# %% is used for displaying % at the end
plt.pie(x, explode=explode, labels=dataName, autopct='%.2f%%')
plt.show()
```



(a) Matplotlib: polar plot of Cos(2x) (b) Actual: polar plot of Cos(2x)

Figure 2.8: Polar Plot, Listing 2.14

2.6.4 Polar plot

In matplotlib, polar plots are based on clipping of the curve so that $r \geq 0$. For example, in Fig. 2.8a (generated by line 12 in Listing 2.14), only two lobes of $cos(2x)$ are generated instead of four. Other two lobes have negative value of 'r', therefore these are clipped by the matplotlib. The actual four lobes are shown in Fig. 2.8b, which can not be generated by matplotlib.

Listing 2.14: Polar plot, Fig. 2.8a

```
1  #polarplotEx.py
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  plt.close("all")
6
7  x=np.linspace(0,2*np.pi, 1000)
8
9  # polar axes is based on clipping so that r >= 0.
10 # therefore only 2 lobes are shown as oppose to 4 lobes.
11 y = np.cos(2*x)
12 plt.polar(x, y)
13
14 plt.show()
```

2.6.5 Bar chart

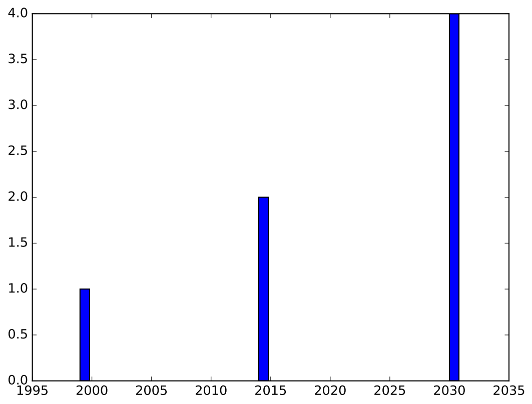
In this section, two types of bar charts are discussed as shown in Fig. 2.9 . Listing 2.15 plots a simple bar chart for 'years vs x'; whereas Listing 2.16 plots multiple data.

Listing 2.15: Bar Chart, Fig. 2.9a

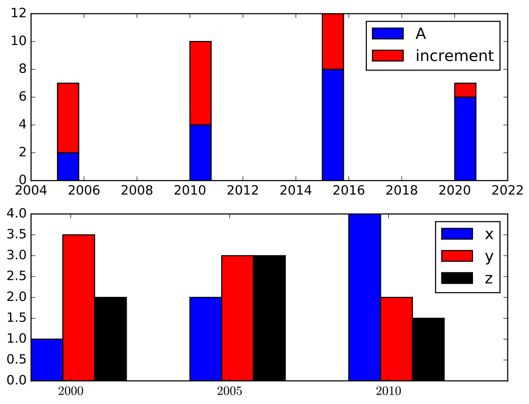
```
#barchartEx.py
import matplotlib.pyplot as plt

plt.close("all")

x = [1, 2, 4]
years = [1999, 2014, 2030]

plt.bar(years, x)

plt.show()
```

Listing 2.16: Bar Chart with multiple data, Fig. 2.9b

```
#barchartCombineEx.py
```



(a) Bar Chart with single data, Listing 2.15



(b) Bar chart with multiple data, Listing 2.16

Figure 2.9: Bar Charts

```
2   import numpy as np
3   import matplotlib.pyplot as plt
4
5   plt.close("all")
6
7   #### subplot 1
8   A = [2, 4, 8, 6]
9   increment = [5, 6, 4, 1]
10  years = [2005, 2010, 2015, 2020]
11
12  plt.subplot(2,1,1)
13  plt.bar(years, A, color = 'b', label='A')
14  plt.bar(years, increment, color = 'r', bottom = A, label='increment')
15  plt.legend()
16
17  plt.show()
18
19  ##### subplot 2
20  x = [1, 2, 4]
21  y = [3.5, 3, 2]
22  z = [2, 3, 1.5]
23
24  width = 0.2
25  locs = np.arange(1, len(x)+1)
26  plt.subplot(2,1,2)
27  plt.bar(locs, x, width=width, label='x')
28  plt.bar(locs+width, y, width=width, color="red", label='y')
29  plt.bar(locs+2*width, z, width=width, color="black", label='z')
30  plt.legend()
31
32  plt.xticks([1.25, 2.25, 3.25],
33             [r'$2000$', r'$2005$', r'$2010$'])
34
35  plt.show()
```

Explanation (Listing 2.16). *In Fig. 2.9b, the data 'increment' is plotted above the data 'A' using 'bottom' parameter in line 14.*

*Further, in the lower subplot, bar charts are plotted side by side using combination of 'locs' and 'width' variable in line 24 and 25. 'width' parameter set the width of the bar; which is set to 0.2 in line 24. Line 27 plots the data 'x' first; then, line 28 plots next data set i.e. 'y', but location is shifted by the 'width' due to command 'locs+width' in the line. Hence, bar chart is plotted beside the bars of the line 27. After that, line 29 shifted the plot of data 'z' by '2*width'. Finally line 32 add ticks for the x-axis and we get the final plot as shown in Fig. 2.9b* ▲

2.7 Annotation

Annotation can be used to make graph more readable as show in Fig. 2.10. Text is added to graph using 'annotate()' command with two different methods as shown in line 25 and 40 in

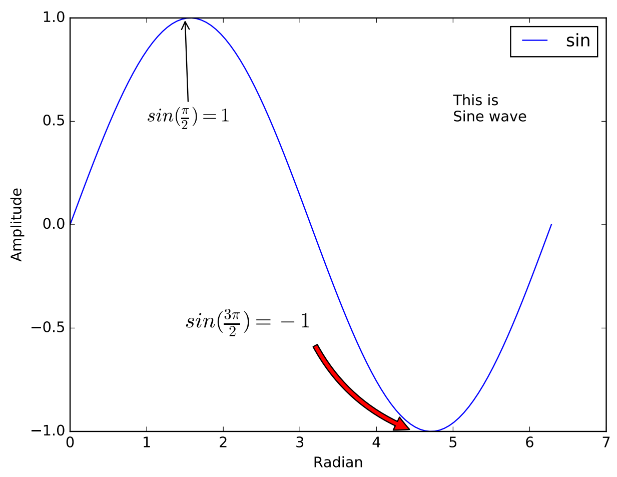


Figure 2.10: Annotation, Listing 2.17

Listing 2.17. Also, 'text()' command (at line 49) is used to add text to the figure.

Listing 2.17: Annotation, Fig. 2.10

```
 1  #annotateEx.py
 2  import numpy as np
 3  import matplotlib.pyplot as plt
 4
 5  # close all the figures, if open from previous commands
 6  plt.close('all')
 7
 8  ############ Sin(x) ####################
 9  x=np.linspace(0, 2*np.pi, 100)
10  sinx=np.sin(x) # calculate sin(x)
11  plt.plot(x,sinx, label='sin') # legend
12
13  ############ Legend ####################
14  plt.legend(loc="best") #show legend
15
16  #### Lable and Grid ####################
17  plt.xlabel("Radian") # x label
18  plt.ylabel("Amplitude") # y label
19
20  ############# Annotate ######################
21  #1.
22  # other arrow style
23  # '-', '->', '-[', '<-', '<->', 'fancy', 'simple', 'wedge'
24  #sin(pi/2)=1
25  plt.annotate(r'$sin(\frac{\pi}{2})=1$',
26               fontsize=16,
27               xy=(1.5, 1), xytext=(1 , 0.5),
28               arrowprops=dict(arrowstyle="->"),
29          )
30
31  # 2.
32  #==============================================================
33  # width: The width of the arrow in points
34  # frac: The fraction of the arrow length occupied by the head
35  # headwidth: The width of the base of the arrow head in points
36  # shrink: Moves the tip and the base of the arrow some percent
37  # away from the annotated point and text,
38  #==============================================================
39  #sin(3*pi/2)=-1
40  plt.annotate(r'$sin(\frac{3\pi}{2})=-1$',
41               fontsize=18,
42               xy=(4.5, -1), xytext=(1.5 , -0.5),
43               arrowprops=dict(facecolor='red', shrink = 0.04,
44                               connectionstyle="arc3,rad=.2"),
45          )
46
47  # 3.
48  #################### Add text to plot ###########
49  plt.text(5, 0.5, 'This is \nSine wave');
50
51  plt.show() #display the plot
```

2.8 Sharing Axis

Listing 2.18, 2.19 and 2.20 create the instances of figure() and subplot() functions of matplotlib to generate various plots.

2.8.1 Common axis for two plots

Here x axis is common for two different plots. Further, in one plot log y-axis is used.

Listing 2.18: Sharing Axis, Fig. 2.11a

```
 1  #shareaxisEx.py
 2  import numpy as np
 3  import matplotlib.pyplot as plt
 4
 5  N=100
 6  x=np.linspace(0.1,4, N)
 7  e1 = np.exp(x)
 8  e2 = np.exp(-x)
 9
10  # create an object 'fig1'  of figure()
11  fig1 = plt.figure()
12
13  # create two instances of fig1 at location 1,1,1
14  subfig1 = fig1.add_subplot(1,1,1)
15  subfig2 = fig1.add_subplot(1,1,1)
16
17  # share the x-axis for both plot
18  subfig2 = subfig1.twinx()
19
20  # plot subfig1
21  # semilogy for log 'y' axis, for log x use semilogx
22  subfig1.semilogy(x, e1)
23  subfig1.set_ylabel("log scale: exp(x)")
24  subfig1.set_xlabel("x-->")
25
26  # plot subfig2
27  subfig2.plot(x, e2, 'r')
28  subfig2.set_ylabel("simple scale: exp(-x)")
29
30  plt.show()
```

Explanation (Listing 2.18). *Line 11 creates an instance 'fig1' of figure() function. Then subfig1 and subfig2 instances of 'fig1' are created in line 14 and 15. 'twinx()' command in line 18, shares the x-axis for both the plots.*

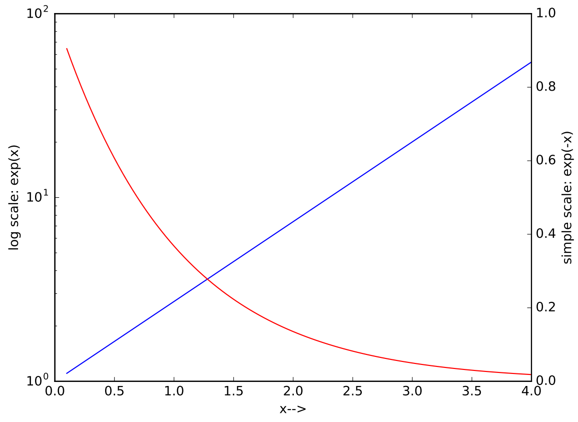
Line 22-24 set the various parameter for subfig1; also note that 'set_' is used for x and y labels. Then line 27-28 plots the second figure. Finally line 30 displays both the plots as shown in Fig. 2.11a. ▲

2.8.2 Sharing Axis-ticks

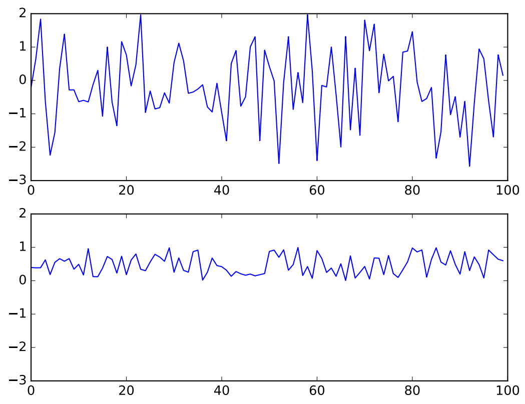
Here, same y-axis ticks (i.e. [-3, 2]) are used for two subplots as illustrated in Fig. 2.11b using Listing 2.19. In the listing, line 15 and 16 create two subplots. Further, line 15 contains 'sharey' parameter which sets ticks in the y-axis of subfig2 equal to subfig1.

Listing 2.19: Sharing Y Axis ticks, Fig. 2.11b

```
 1  #subplotEx2.py
 2  import numpy as np
 3  import matplotlib.pyplot as plt
 4
 5  N=100
 6  x=np.arange(N)
 7  rn = np.random.randn(N)
 8  r = np.random.rand(N)
 9
10  # create an object 'fig1'  of figure()
11  fig1 = plt.figure()
12
13  # create two instances of fig1
14  subfig1 = fig1.add_subplot(2,1,1)
```



(a) Shared x-axis by two figures, Listing 2.18



(b) Same Y axis values, Listing 2.19

Figure 2.11: Shared Axis

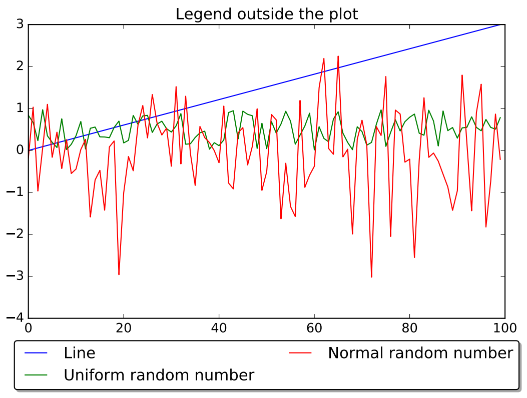


Figure 2.12: Legend outside the plot, Listing 2.20

```
subfig2 = fig1.add_subplot(2,1,2, sharey = subfig1) #share y axis

# plot figures
subfig1.plot(x, rn)
subfig2.plot(x, r)
plt.show()
```

2.9 Add legend outside the plot

In Listing 2.20, legends are placed outside the figure as shown in Fig. 2.12. It can be quite useful, when we have large number of figures in a single plot. Note that, in line 12, instance of subplot is created directly; whereas in Listing 2.11b, subplot are created using instances of figure(), which require '*add_subplot*' command as shown in line 14 and 15 there.

Listing 2.20: Legend outside the plot, Fig. 2.12

```
1  #legendPosEx.py
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  # close all the figures, if open from previous commands
6  plt.close('all')
7
8  ur = np.random.rand(100)
```

```
nr = np.random.randn(100)
x=np.linspace(0,3, 100)

ax = plt.subplot(1,1,1)

ax.plot(x, label="Line")
ax.plot(ur, label="Uniform random number")
ax.plot(nr, label="Normal random number")
ax.set_title("Legend outside the plot")

# Plot position and shriking to create space for legends
box = ax.get_position()
ax.set_position([box.x0, box.y0 + box.height * 0.2,
                 box.width, box.height * 0.8])

# Add legend below the axis
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),
          fancybox=True, shadow=True, ncol=2)

plt.show()
```

If you first fortify yourself with the true knowledge of the Universal Self, and then live in the midst of wealth and worldliness, surely they will in no way affect you.

–Ramakrishna Paramahansa

Chapter 3

Speed up the simulations

3.1 Introduction

Python is designed such that beginners and experienced programmers can learn it quickly. Also, Python provides various useful built-in structures and libraries for different applications. Numpy, Scipy and Matplotlib libraries along with simple coding style make python a preferred language for scientific computing as compared to other languages e.g C, C++ and Fortran etc.

Although Python is extremely easy to learn as compare to C, C++ and Fortran, but on the cost of speed of simulation. Python is dynamic typed language, whereas the C/C++ are static typed languages. In the other words, in C/C++, we define types of the variables while declaring it; whereas we do not define the types of variables in Python. Python interpreter checks the data type while executing the script, which results in longer simulation time.

Cython is a programming language which has the advantage of both the languages i.e. Python and C/C++. Cython provides the static binding for Python. Also, it converts the python code into highly optimized C/C++ code which increases the simulation speed. Further, Numba compiler is another good option to speed up the simulation. It increases the simulation speed by converting the python code to optimized machine code.

Note that, the system must have C-compiler for executing Cython codes. In Linux, it can be installed using command 'sudo apt-get install g++'; and for Windows, please download and install the C-compiler (with default settings) from website 'Visual C++ Build Tools'.

3.2 Saving functions in separate file

In this section, a simple code is written in which both definitions and main program are in the same file. Then in next section, functions and main programs are separated in two different files, which is easy to maintain as compared to previous method. The 'import' command is used to call definitions to the main program. See Appendix B, to learn more the 'import' command.

3.2.1 Function and code in same file

In Listing 3.1, functions (line 2 and 5) and main program (line 8-12) are written in the same file. But, for lengthy programs, such methods can be too complicated to manage. Therefore, it is better to write all the functions in separate files.

Listing 3.1: Add function in same file

```
1  #funcEx.py
2  def add2Num(a, b):
3      return(a+b)
```

```

def diff2Num(a,b):
    return(a-b)

x = add2Num(2,3)
print(x) # 5

y = diff2Num(2, 3)
print(y) # -1
```

3.2.2 Function and code in different files

Saving definitions to another file is very easy. To do this, cut and paste the defintion from Listing 3.1 to another file, e.g. 'funcFile.py' as shown in Listing 3.2.

Listing 3.2: Save definitions in separate file

```
#funcFile.py: define you function here

def add2Num(a, b):
    return(a+b)

def diff2Num(a,b):
    return(a-b)
```

Next task is to import functions in Listing 3.2 to Listing 3.3 which contains the main program. At line 4 of Listing 3.3, add2Num and diff2Num functions are imported to the code using 'import' command. Note that, 'funcFile' is the python file where definitions are saved as shown in Listing 3.2. We can have multiple function-files with different names; and all these files can be imported in the same way. Further, if we want to import all the functions of the file then '*' can be used as shown in Line 7 of Listing 3.3.

Listing 3.3: Call definition to the main code

```
1  #funcMain.py
2
3  #call functions from files
4  from funcFile import add2Num, diff2Num
5
6  ##to call everything from file, use below lien.
7  #from funcFile import *
8
9  x = add2Num(2,3)
10 print(x) # 5
11
12 y = diff2Num(2, 3)
13 print(y) # -1
```

3.3 Cython

Cython is a programming language which translate the Python code to optimized C/C++ code. Due to optimized code, there is an increase in the simulation speed, as shown in Table 3.1. This table compares the performances of various codes, which are used in this tutorial.

3.3.1 First Cython Code (.pyx file)

Cython files are saved as '.pyx' files. Further, Cython supports all the python syntax, therefore simply by changing the filetype (i.e. .py to .pyx) we can achieved faster speed of simulation, as compared in Row 1 and 2 of Table 3.1. Following are the two steps to convert the python codes in Listing 3.2 and 3.3 to cython codes,

1. **Change File type**:
   Save Listing 3.2 as '.pyx' file with some other name, e.g. 'funcCfile.pyx' as shown in Listing 3.4. Here, different file names are chosen to avoid confusion with file names.

Table 3.1: Speed comparison of various codes described in section 3.4

S. No.	Function Name (Code Type)	Time (sec)	xSpeed	Description
1	**purePyloop (Python)**	19.52	1	Pure python
2	**pyloop (Cython)**	9.69	2	Pure python in .pyx file
3	**cyloop(Cython)**	5.28	3.7	Argumnent type is defined
4	**cypyloop (Cython)**	5.25	3.7	Argument and return types are defined
5	**cypyloop2 (Cython)**	0.055	354.9	Argument, return and variable types are defined
6	**numbaLoop (Numba)**	0.19	102.7	Numba compiler

Listing 3.4: Save definitions in separate file

```
#funcCfile.pyx:

def add2Num(a, b):
    return(a+b)

def diff2Num(a,b):
    return(a-b)
```

2. **Import '.pyx' file to main file**:
   The 'import; command in python does not search for .pyx file. We need to add two lines, (i.e. lines 4-5 in in Listing 3.5), to tell python interpreter to find files in '.pyx' format as well.

   Above two steps convert the python-code into cython-code. Now, we can run the cython code using "python funcCmain.py" command to see the outputs, as shown in Listing 3.5.

Listing 3.5: Call definition to the main code

```
 1  #funcCmain.py
 2
 3  #to import pyx file: following two lines are needed
 4  import pyximport
 5  pyximport.install()
 6
 7  #call functions from files
 8  from funcCfile import add2Num, diff2Num
 9
10  ##to call everything from file, use below lien.
11  #from funcFile import *
12
13  x = add2Num(2,3)
14  print(x) # 5
15
16  y = diff2Num(2, 3)
17  print(y) # -1
```

There are three methods available to import the cython code in the file.

(a) One method is discussed in this section, which is sufficient as long as we are writing our code in Python and Cython language.

(b) Second method is used in Jupyter Notebook, which is described on the website. Also, you can download the copy of Jupyter notebook for this tutorial from the homepage.

(c) Third method needs a setup.py file, which is required when we are using 'C/C++' codes along with the python codes. Since, we are interested only in Python-Codes for simulations, therefore it is not discussed in the tutorial.

3.3.2 def, cdef and cpdef

Cython definition can be created in three ways i.g. using ‘def’, ‘cdef’ and ‘cpdef’ keywords. These three keywords are discussed in this section,

3.3.2.1 def

Cython ‘def’ can be defined in two ways as shown as shown at lines 4 and 9 in Listing 3.6. In first method i.e. line 4, normal python definition is written but saved in ‘.pyx’ file, as discussed in section 3.3.1. In the second method, we define argument type in ‘def’ (see ‘int x’ at line 9 in Listing 3.6). Note that we can not define the return type for the function using ‘def’. For this we use ‘cdef’ and ‘cpdef’.

3.3.2.2 cdef

We can define the return type of the function using ‘cdef’ as shown at line 15 in Listing 3.6. Here first ‘int’ is the return type of the definition.

‘cdef’ can be called from the same file only e.g. in Listing 3.6, line 25 is calling ‘cdef’ function (i.e. cfunc) at line 15. But this function can not be called from other files e.g. line 15 in Listing 3.7 will generate error.

3.3.2.3 cpdef

Cython function with return type, can be called from outside file if it is define as ‘cpdef’. ‘cpdef’ adds the python wrapper to the definition, therefore it can be called from outside files e.g. line 18 in Listing 3.7 is calling ‘cpdef’ function at line 20 in Listing 3.6. Further, line 21 in Listing 3.7 is calling ‘cpdef’ function at line 24 in Listing 3.6; then line 24 is calling ‘cdef’ definition at line 15 (which can not be called directly from outside files).

Listing 3.6: Cython Definitions

```
 1 #cythonDef.pyx
 2 
 3 #pure python function run by cython
 4 def pyfunc(x):
 5     return (x+1)
 6 
 7 # cython function: as input type for argument 'x' is defined
 8 #def is used to define it.
 9 def cyfunc(int x):
10     return (x+1)
11 
12 # cdef: return type 'int' is defined, cdef/cpdef is required for this.
13 # cdef function can not be called from other file
14 # it can be called within same file
15 cdef int cfunc(int x):
16     return(x+1)
17 
18 # cpdef: it adds the python wrapper
19 # hence, it can be called from outside file
20 cpdef int cypyfunc(int x):
21     return(x+1)
22 
23 # cpdef:
24 cpdef int cypyfunc2(int x):
25     y = cfunc(x)
26     return(y+1)
```

Listing 3.7: Main function to call Cython Definitions

```
1 #cythonDefMain.py
2 
3 #to import pyx file: following two lines are needed
4 import pyximport
5 pyximport.install()
6 
7 #call functions from files
```

```
from cythonDef import *

print("pyfunc ", pyfunc(2)) # 3

print("cyfunc ", cyfunc(2)) #3

##error: cdef can not be called from outside
#print(cfunc(int x))

#cpdef can be called from outside the function file
print("cypyfunc ", cypyfunc(2)) # 3

#cfunc (cdef) is called through cypyfunc2 (cpdef)
print("cypyfunc2 ", cypyfunc2(2)) # 4
```

Note that, adding 'argument types', 'return types' and 'variable types (see Line 28-29 in Listing 3.10)' increase the speed of the code significantly as shown in Table 3.1. To convert a python code to cython code, look for the variables, which do not change their type during execution and define their types as show in Listing 3.10.

3.4 Timing Comparison: Python, Cython, Numba

In this section, simulation times are compared for different codes, to execute the simple loops. Further, Numba compiler is discussed which can increase the speed of the code significantly by adding decorator to each definition.

3.4.1 Python

In Listing 3.8, two 'for' loops are defined and value of variable 'x' is increasing at each iteration. Execution time for this simple loop is calculated for different codes. Further, 'time' function is used to measure the simulation time as shown in line 3 of Listing 3.9.

Listing 3.8: Pure python loop

```
#purePythonLoop.py

N=10000

#pure python function
def purePyloop(x):
    for i in range(N):
        for j in range(N):
            x = x+1
    return(x)
```

Listing 3.9: Main function to call purePythonLoop.py

```
1  #purePythonLoopMain.py
2
3  import time
4  from purePythonLoop import purePyloop
5
6  start_time = time.time()
7  print(purePyloop(2))
8  print("--- %s seconds ---" % (time.time() - start_time)) #19.52 sec
```

3.4.2 Cython

In Listing 3.10, cython functions are defined in four ways as shown below,

1. pyloop: this is simple python loop saved in '.pyx' file
2. cyloop: in this function, python function (def) with 'argument type' is defined.
3. cypyloop: Here 'return type' and 'argument type' are defined using 'cpdef'.
4. cypyloop2: In this function 'return type', 'argument type' and 'variable type' are defined using 'cpdef'.

In 'cypyloop2' function (line 28 in Listing 3.10), data types of i and j are defined; which increases the simulation speed by 350 times as shown in Table 3.1. Run Listing 3.11 to compare the simulation times of various codes.

Listing 3.10: Main function to call Cython Definitions

```
1  #cythonFunc.pyx
2
3  N=10000
4  #pure python function
5  def pyloop(x):
6      for i in range(N):
7          for j in range(N):
8              x = x+1
9      return(x)
10
11 # cython function: as input type for argument 'x' is defined
12 def cyloop(int x):
13     for i in range(N):
14         for j in range(N):
15             x = x+1
16     return (x)
17
18 # cpdef: it can be called from outside file
19 # adds python wrapper to code
20 cpdef int cypyloop(int x):
21     for i in range(N):
22         for j in range(N):
23             x = x+1
24     return(x)
25
26 # cpdef: it can be called from outside file
27 cpdef int cypyloop2(int x):
28     cdef: #define all variable types inside cdef
29         int i, j
30     for i in range(N):
31         for j in range(N):
32             x = x+1
33     return(x)
```

Listing 3.11: Main function to call cythonFunc.py

```
#cythonMain.py

#to import pyx file: following two lines are needed
import pyximport
pyximport.install()

import time
from cythonFunc import *

start_time = time.time()
print(pyloop(2))
print("--- %s seconds ---" % (time.time() - start_time)) #9.69 sec

start_time = time.time()
print(cyloop(2))
print("--- %s seconds ---" % (time.time() - start_time)) #5.28 sec

start_time = time.time()
print(cypyloop(2))
print("--- %s seconds ---" % (time.time() - start_time)) #5.25 sec

start_time = time.time()
print(cypyloop2(2))
print("--- %s seconds ---" % (time.time() - start_time)) #0.055 sec
```

3.4.3 Numba

Numba can increase the speed of the python code just by adding one line before each definition as shown in Listing 3.12. In line 3 of the listing, 'autojit' is imported from Numba library, then a decorator is placed before the definition at line 7. After these changes, run the code in Listing 3.13; this time code will run 100 times faster than the original python code as shown in Table 3.1. Lastly, there are various other options available for Numba-decorators, but 'autojit' works fine for most of the cases.

Listing 3.12: Add Numba to python definitions

```python
#numbaLoop.py

from numba import autojit
N=10000

#pure python function
@autojit
def numbaLoop(x):
    for i in range(N):
        for j in range(N):
            x = x+1
    return(x)
```

Listing 3.13: Main function to call numbaLoop.py

```python
#numbaMain.py

import time
from numbaLoop import numbaLoop

start_time = time.time()
print(numbaLoop(2))
print("--- %s seconds ---" % (time.time() - start_time)) #0.19 sec
```

3.5 Conclusion

In practice, simulations code can be quite lengthy. Also, most of the simulations are based on Monte-Carlo approach, where we need to iterate the codes for large number of times. Sometimes these codes may take few days to complete the simulation. In this tutorial, we saw the various methods to reduce the simulation time of the codes, which can be quite helpful for the simulators based on Monte-Carlo approach.

Happiness is your nature. It is not wrong to desire it. What is wrong is seeking it outside when it is inside.

–Ramana Maharshi

Chapter 4

Confirm theory with simulations

4.1 Introduction

Simulations are often used to confirm the theoretical results or to analyze the outcome of the newly proposed systems. If the results obtained by the simulations are satisfactory, then the systems are implemented on the hardware. This is done because direct implementation and debugging on the hardware can be quite costly and time consuming process.

Numpy and scipy libraries provide various mathematical and statistical functions for numerical computation. Numpy (Numeric Python) provides various routines for fast operations on arrays e.g. sorting, selecting, discrete Fourier transforms and basic linear algebra etc. Whereas scipy (Scientific Python) contains algorithms related to specialized applications e.g. eigenvalues, optimization and interpolation etc. Hence, these two libraries are quite useful for simulation purposes as discussed in this tutorial. Further, the plots of the results are quite useful in understanding, analyzing and comparing the results; which can be generated with matplotlib library.

In this tutorial, our aim is to show that how simulations are used to confirm the theoretical results. For this purpose, the block diagram of a wireless communication system is illustrated in section 4.3, along with the 'theoretical error expression' in the system. Then, theoretical expression is plotted using numpy, scipy and matplotlib libraries. Finally, various blocks of the block-diagram are simulated and the simulation plot is compared with the theoretical plot, to validate the theoretical expression obtained by the mathematical calculation.

4.2 Numpy

In this section, some of the numpy features are discussed, which are essential to understand the simulator used in this tutorial.

4.2.1 N-Dimensional Array (ndarray)

In this section, arrays are defined using 'array' function; and then certain elements are chosen from the array, which is known as slicing.

4.2.1.1 Defining Array

We can define N-dimensional array using 'array' command as shown Listing 4.1. In the listing, one dimensional (1D) and 2D arrays are defined at lines 6 and 10 respectively.

Listing 4.1: Defining Arrays

```
1  # defineArray.py
```

```

import numpy as np

# one dimensional array
a = np.array([1, 2, 3])
print (a) # [1 2 3]

# two dimensional array
b = np.array([
        [1, 2, 3],
        [4, 5, 6]
    ])
## or define as follows in one line: less readable
# b = np.array([[1, 2, 3],[4, 5, 6]])
print(b)
'''
[1 2 3]
[[1 2 3]
 [4 5 6]]
'''
```

4.2.1.2 Slicing of Array

Similar to python, indexing in numpy start with zero. We can access and change the values of elements with indexing, as shown in Listing 4.2.

Note that, at line 6 of Listing 4.2, b is defined with all the elements as integer. Therefore, array b is by default set to integer type. If we try to change the element with float value, numpy will save it as integer, as shown in line 20. We can change this behavior by defining the type while defining arrays as shown in line 27.

Listing 4.2: Accessing elements of array

```
1  # accessArray.py
2  
3  import numpy as np
4  
5  # two dimensional array
6  b = np.array([
7          [1, 2, 3],
8          [4, 5, 6]
9      ])
10 
11 b[0,1] # 0th row and 1st column = 2
12 
13 b[1, 2] = 10 # change 6 to 10
14 print(b)
15 '''
16 [[ 1  2  3]
17  [ 4  5 10]]
18 '''
19 
20 b[1, 1] = 15.2 #change 5 to 15.2
21 print(b) # note 15 is saved instead of 15.2
22 '''
23 [[ 1  2  3]
24  [ 4 15 10]]
25 '''
26 
27 c = np.array([[9, 8, 7], [3, 5, 1]], dtype=float)
28 c[1, 1] = 10.5 # change 5 to 10.5
29 print(c)
30 '''
31 [[  9.    8.    7. ]
32  [  3.   10.5   1. ]]
33 '''
```

We can also change multiple values of arrays using slicing as shown in Listing 4.3.

Listing 4.3: Slicing of Array

```
# sliceArray.py

import numpy as np

# note that in 'arange command'
# last number i.e. 10 is not included in the list
```

```
s = np.array([np.arange(1,10)])
print(s) # [[1 2 3 4 5 6 7 8 9]]

# below line is known as slicing
x = s[0,3:6] #get element 3 to 5 from line 0
print(x) # [4 5 6]

s[0, 5:9] = 2 #replace number from position 5 to 9 by 2.
print(s) # [[1 2 3 4 5 2 2 2 2]]
```

4.2.2 Linear Algebra

In the previous section, arrays are used to define matrix. In this section, various types of matrices, e.g. identity matrix and zero matrix etc., are created using Numpy library. Also, various mathematical operations are performed on matrices.

4.2.2.1 Special matrix

Zero matrix, diagonal matrix and identity matrix etc. are frequently used in the simulations, which can be defined as shown in Listing 4.4.

Listing 4.4: Special matrix

```
# specialMatrix.py

import numpy as np

I = np.identity(3) # identity matrix
print(I)
'''
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
'''

d = np.diag([2, 3, 4]) # diagonal matrix
print(d)
'''
[[2 0 0]
 [0 3 0]
 [0 0 4]]
'''

z = np.zeros(3) #zero matrix
print(z) # [ 0.  0.  0.]

z32 = np.ones((3,2), dtype=float) # one matrix of size (3,2)
print(z32)
'''
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
'''

zI = np.zeros_like(I) # define zero matrix of size 'I' at line 5
print(zI) # zero matrix of order (3,3) as I define above

E = np.eye(3) # identity matrix of size (3,3)
print(E)

dE = np.eye(5, k=2) # identity matrix along k-th diagonal
print(dE)
'''
[[ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
'''
```

4.2.2.2 ‘mat’ command to create matrix

Similar to ‘array’, the ‘mat’ command can be used to define matrices as shown in Listing 4.5.

Listing 4.5: ‘mat’ command

```
# matMatrix.py

import numpy as np

m = np.mat('2, 3, 4; 9, 8, 7; 1, 6, 5')
print(m)
'''
[[2 3 4]
 [9 8 7]
 [1 6 5]]
 '''
```

4.2.2.3 ‘mat’ vs ‘ndarray’

There are some differences in mathematical operations on the matrices, which are created using ’mat’ and ’array’ commands, as shown in Listing 4.6. Further, array to matrix conversion is also shown at line 40 of the listing.

Listing 4.6: ‘mat’ vs ‘ndarray’ command

```
 1  # arrayMatDiff.py
 2
 3  import numpy as np
 4
 5  # type difference
 6  m = np.mat('1, 1, 1; 2, 4, 2; 3, 3, 5')
 7  print(type(m)) # numpy.matrixlib.defmatrix.matrix
 8
 9  n = np.array([[1, 1, 1], [2, 4, 2 ], [3, 3, 5]])
10  print(type(n)) # numpy.ndarray
11
12  # multiplication
13  print(m*m*m) # matrix multiplication
14  '''
15  [[ 46  62  62]
16   [124 172 164]
17   [186 246 254]]
18  '''
19
20  print(n*n*n) # point to point multiplication
21  '''
22  [[  1   1   1]
23   [  8  64   8]
24   [ 27  27 125]]
25   '''
26
27  #To perform matric multiplicaiton using array,
28  # we need to use 'dot' operator,
29  print(n.dot(n).dot(n)) # matrix multiplication using array
30  '''
31  [[ 46  62  62]
32   [124 172 164]
33   [186 246 254]]
34   '''
35
36  # ndarray and matrix matlitpication
37  print(n.dot(n)*m) #same as above
38
39  # convert array to matrix using 'mat'
40  print(np.mat(n)**2) # convert array to matrix
41  '''
42  [[ 6  8  8]
43   [16 24 20]
44   [24 30 34]]
45  '''
```

4.2.3 Random Number Generator

In this section various types of random numbers are discussed, which are very useful in simulations to generate the random data. Further, Fig. 4.1 compares the random data generated by various functions in the listing.

Listing 4.7: Random Number Generator

```
# randomNumberEx.py

import numpy.random as nr
```

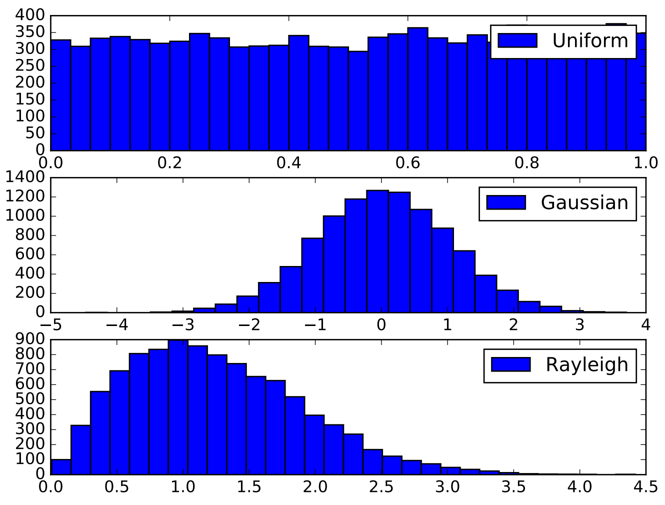


Figure 4.1: Random Number Generator, Listing 4.7

```python
import matplotlib.pyplot as plt

# generate sequence of 0 and 1
rInt = nr.randint(0, 2, 10)
print(rInt) # [0 1 0 1 1 1 1 1 1 0]

# generate sequence of +/- 1
rInt2 = [2*i + 1 for i in nr.randint(-1, 1, 10)]
print(rInt2) # [1, -1, 1, 1, 1, -1, 1, -1, -1, -1]

plt.close("all")

# uniform random variable
plt.subplot(3,1,1)
uniformRandom = nr.rand(10000)
plt.hist(uniformRandom, 30, label="Uniform")
plt.legend()

# normal random variable
plt.subplot(3,1,2)
normalRandom = nr.randn(10000)
plt.hist(normalRandom, 25, label="Gaussian")
plt.legend()

# Rayleigh distributed random number
plt.subplot(3,1,3)
rayleighRandom = nr.rayleigh(1, 10000)
plt.hist(rayleighRandom, 30, label="Rayleigh")
plt.legend()

plt.show()
```

4.3 System model

Fig. 4.2 shows the block diagram of the proposed system. In this system, there are four blocks i.e. transmitter, noisy environment, receiver and decision blocks.

Explanation (Fig. 4.2)**.** *In this system, the transmitter (on the left side) transmits the series of random data with values* ± 1*. Then noise is added by the environment, which is modeled as Gaussian distributed data. Hence, received signal is not same as the transmitted signal. Next, we need to make the decision about the transmitted signals (based on received signals, whose values are no longer* ± 1*, due to addition of noise) as shown in the decision block; i.e. if the received signal is greater than zero then transmitted signal is 1 otherwise it is zero. Based on*

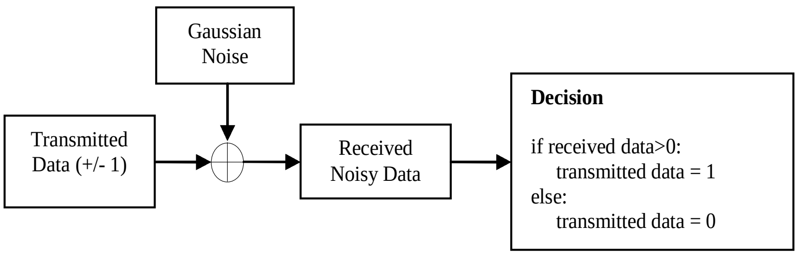


Figure 4.2: Block diagram of the proposed system

this process, the final theoretical bit error rate (BER) equation is given by,

$$BER = \frac{1}{2} erfc\left(\sqrt{SNR}\right) \tag{4.1}$$

where, 'SNR' and '$erfc$' are the signal to noise ratio and complementary error function respectively. ▲

Theories are verified using simulations with following steps,

1. **First, we need to plot the theoretical results, i.e. BER equation which is given in equation (4.1) in this example.**
2. **Next, we need to simulate the block diagram and plot the results, with procedure given in explanation.**
3. **If above two graphs match each other, then we can say that the theoretical result is correct for the proposed system in Fig. 4.2.**

4.4 Theoretical BER

In this section, theoretical BER of BPSK system is plotted which is given by equation 4.1. Listing 4.8 is the python code for plotting the BER equation.

Explanation (Listing 4.8). *Equation 4.1 is written at line 16 of the listing. BER values are calculated for 10 different SNR values i.e. '-2dB to 9dB' as shown in line 10 of the code. In line 11, dB values are converted into normal power scale. Then, these values are used at line 16 to calculate the BER values. Note that, lines 28-29 tell python-interpretor that main() function should be executed first.* ▲

Listing 4.8: Theoretical BER of BPSK system in Gaussian Noise, Fig. 4.3

```
 1  #theoryBPSK.py: plot theoretical BER of BPSK
 2
 3  import matplotlib.pyplot as plt
 4  import numpy as np
 5  from scipy.special import erfc
 6
 7
 8  def main():
 9    #BPSK: theoretical error
10    SNRdB = np.array(np.arange(-2, 10, 1)) #SNR in db
11    SNR = 10**(SNRdB/10) # SNRdB to SNR conversion
12
13    theoryBER = np.zeros(len(SNRdB),float) # zero vector of size SNRdB
14
15    for i in range(len(SNRdB)):
16        theoryBER[i] = 0.5*erfc(np.sqrt(SNR[i]))
17
18    #plot results
19    plt.semilogy(SNRdB, theoryBER, '--mo')
20    plt.ylabel('BER')
21    plt.xlabel('SNR')
22    plt.title('BPSK BER Curves')
23    plt.legend(['Theory'])
```

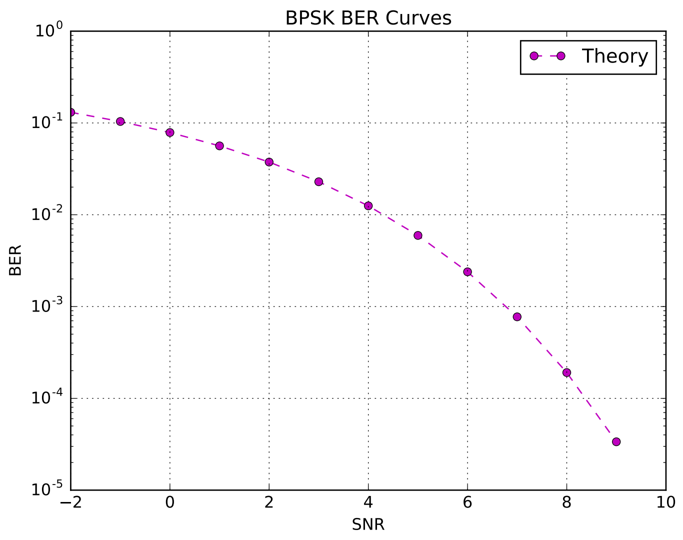


Figure 4.3: Theoretical BER of BPSK system in Gaussian Noise, Listing 4.8

```
    plt.grid()
    plt.show()

# Standard boilerplate to call the main() function.
if __name__ == '__main__':
    main()
```

4.5 Confirm theory with simulation

In previous section, only theoretical result was plotted using python. In this section, block diagram of the system i.e. Fig. 4.2 is also simulated as shown in Listing 4.9 and 4.10. Further, these listing are converted into cython, numba and matlab codes as well, for increasing the speed of the simulations. These codes are given in the Appendix C. Simulation times for python, cython, numba and matlab are compared in Table 4.1.

S. No.	Code Type	Time (sec)	xSpeed
1	**Python**	87.96	1
2	**Numba**	8.93	9.8
3	**Cython**	7.43	11.8
4	**Matlab**	7.44	11.8

Table 4.1: Simulation time comparison: Python, Numba and Cython

Explanation (Listing 4.9 and 4.10). *In the Listing 4.9, time function in line 3-4 are used to measure the time of simulation. Further in line 11, first 'errorCalculation' is the name of the python file, in which second 'errorCalculation' is defined as function, as shown in Listing 4.10.*

The simulator is iterated 30 times as shown in line 15 of Listing 4.9, then average of these 30 values are taken in line 35. The function 'errorCalculation' (line 11) calculates the BER values for each iteration, which are stored in 'error' matrix (line 32).

SNR is defined as ratio of signal and noise power i.e. $SNR = \frac{E_b}{N_0}$, *where* E_b *is the signal power and* $\frac{N_0}{2}$ *is the noise power. Since, the power of the signal is considered as 1, therefore* $SNR = \frac{1}{N_0}$. *Line 25 converts the noise powers into noise voltages i.e.* $\sqrt{\frac{N_0}{2}} = \sqrt{\frac{1}{2*SNR}}$, *as we need to add the noise to the signal as shown in Fig. 4.2. These values of noise are send to Listing 4.10 through line 31.*

Next, Listing 4.10 uses above calculated noise values as noiseAmp (line 5). In line 18, Gaussian noise is generated using 'randn' function (i.e. normal random distribution function). Note that, Normal distribution is nothing but the Gaussian distribution with zero mean and unit variance. Also, transmitted bits (i.e. 0 and 1) are generated randomly in line 7, which are converted into ±1 in lines 11-15. Next, different noise levels are added to signal in line 21. Line 24-28 make decision about the transmitted bit according to decision rule shown in Fig. 4.2. Line 30 compares the decision with actual transmitted bit and set the value to 1 if decision is wrong, otherwise set it to zero. Finally, all these 1's are added (line 31) to calculate the total number of wrong decisions and the value is return to Listing 4.9.

This process is repeated 30 time (line 15) in Listing 4.9 and average to these errors are taken in line 36 [1]*, which is the plotted by lines 47-55. Finally, we find that plot of theoretical BER (line 39-41) matches with the simulation as shown in Fig. 4.4. Therefore we can say that the theoretical result is correct, as it is verified by the simulator (which is created according to proposed system in Fig. 4.2)*

▲

Listing 4.9: Simulation BER of BPSK system in Gaussian Noise, Fig. 4.4

```
 1 # simBPSK.py
 2
 3 import time #to measure the simulation time
 4 startTime=time.time()
 5
 6 import numpy as np
 7 from scipy.special import erfc
 8 import matplotlib.pyplot as plt
 9
10 #errorCalculation: calculate the error in decision
11 from errorCalculation import errorCalculation
12
13 def main():
14   bitLength  = 100000 #number of transmitted bit
15   iterLen=30 #iterate 30 time for better average
16
17   SNRdB = np.array(np.arange(-2, 10, 1),float)
18   SNR = 10**(SNRdB/10) # SNR_db to SNR conversion
19   noise = np.zeros(len(SNRdB), float)
20
21   error =np.zeros((iterLen, len(SNRdB)), float)
22
23   for iL in range(iterLen):
24     #Generate various Noise-voltage (N0/2) Level
25     for i in range (len(noise)):
26       noise[i]= 1/np.sqrt(2*SNR[i]) # N0/2 = 1/sqrt(2*noisePower)
27
28     #calculate error
29     errorMatrix =np.zeros(len(SNRdB), float)
30     for i in range(len(noise)):
31       errorMatrix[i] = errorCalculation(bitLength, noise[i])
32     #save error at each iteration
33     error[iL]=errorMatrix
34
35   #average error
36   BER = error.sum(axis=0)/(iterLen*bitLength)
37
38   #theoritical error
39   theoryBER = np.zeros(len(SNRdB),float)
40   for i in range(len(SNRdB)):
41     theoryBER[i] = 0.5*erfc(np.sqrt(SNR[i]))
42
43   #print simulation time
44   print(time.time()-startTime) # 87.96 sec
45
46   #plot the graph
47   plt.semilogy(SNRdB, BER,'--')
48   plt.semilogy(SNRdB, theoryBER, 'mo')
49   plt.ylabel('BER')
50   plt.xlabel('SNR')
51   plt.title('BPSK BER Curves')
52   plt.legend(['Simulation', 'Theory'], loc='upper right')
53
54   plt.grid()
55   plt.show()
```

[1] The process of repeating the simulation various times and then taking the average at the end is known as Monte-Carlo simulation.

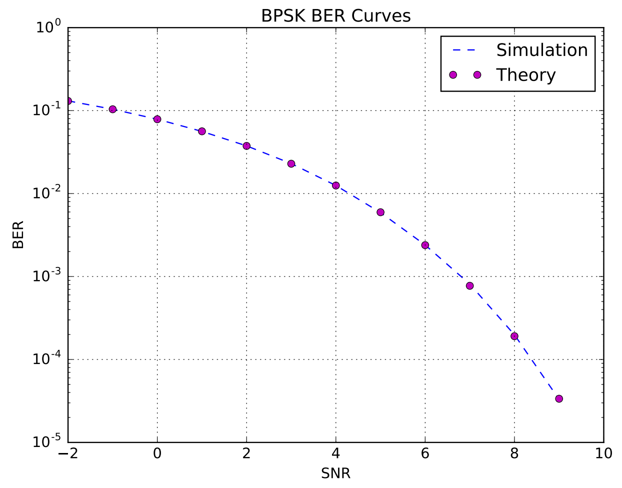


Figure 4.4: Theory and Simulation BER are overlapped, Listing 4.9

```

# Standard boilerplate to call the main() function.
if __name__ == '__main__':
  main()
```

Listing 4.10: Calculate error for each iteration

```
# errorCalculation.py

import numpy as np

def errorCalculation(bitLength, noiseAmp):
  #uniformly distributed sequence to generate singal
  b = np.random.uniform(-1, 1, bitLength)

  #binary signal generated from 'b'
  signal = np.zeros((bitLength),float)
  for i in range(len(b)):
    if b[i] < 0:
      signal[i]=-1
    else:
      signal[i]=1

  #Normal Distribution: Gaussian Noise with unit variance
  noise = np.random.randn(bitLength)

  #recevied signal
  rxSignal = signal + noiseAmp*noise
  # decision:
  detectedSignal = np.zeros((bitLength),float)
  for i in range(len(b)):
    if rxSignal[i] < 0:
      detectedSignal[i]=-1
    else:
      detectedSignal[i]=1
  #error matrix: save 1 if error else 0
  errorMatrix = abs((detectedSignal - signal)/2)
  error=errorMatrix.sum() #add all 1's for total error
  return error
```

4.6 Conclusion

In this tutorial, we have seen the basics of Numpy. Numpy, Scipy and Matplotlab libraries are used to compare the theoretical and simulation results for the BPSK system in the white Gaussian noise condition. Lastly, the graph of simulation result overlapped with the theoretical results, which validates the theoretical results. Cython, numba and matlab codes are shown in appendix and simulation time is compared for python code, cython code and numba code in Table 4.1.

Love does not grown on trees or brought in the market, but if one wants to be "Loved" one must first know how to give unconditional Love.

–Kabeer

Chapter 5

Iterating over the list

5.1 Introduction

Since the Lists and dictionaries can be modified, therefore these are used for data manipulation operations such as retrieving or editing the data. In this chapter, various iteration options are shown for list and dictionaries.

5.2 Lists

Items in the list can be iterated in various ways. In Listing 5.3.2, all the numbers in the list are squared using 'for loop',

Listing 5.1: For loop to iterate the elements in the list

```python
#squareList.py
num = [1, 2, 3, 4, 5]
square_num = []
for n in num:
  square_num.append(n ** 2)
print(square_num) #[1, 4, 9, 16, 25]
```

But there are other useful features available in Python, which can be used to perform such operations with less coding. In this section, following methods are discussed to iterate through all the items in the list,

1. List comprehension
2. Lambda function
3. Map
4. Filter
5. Reduce

5.2.1 List comprehension

List comprehension is the short and easy way to apply some operations on a list and save it in new list. Listing 5.2 squares the elements in a list using list comprehension (see Line 5 and 9).

Listing 5.2: List comprehension

```python
 1  #listComprehension.py
 2  num = [1, 2, 3, 4, 5]
 3
 4  #square all the numbers in list 'num'
 5  square_num = [n**2 for n in num]
 6  print(square_num) #[1, 4, 9, 16, 25]
 7
 8  #squre only even number
 9  square_even_num = [n**2 for n in num if n%2==0]
10  print(square_even_num)#[4, 16]
```

5.2.2 Lambda function

Python supports the functions those have no names, which are known as Lambda functions. Listing 5.3 is the example of Lambda function. Here two lambda functions are defined at Lines 2 and 6 using keyword 'lambda'. In line two 'lambda : x, y' represents that 'x' and 'y' are the two parameters used by the function 'sum_num'. Then Line 3 calls the function 'sum_num' and the output is printed by line 4.

Listing 5.3: Lambda function

```
1  #LambdaEx.py
2  sum_num = lambda x, y: x+y; #'sum_num' function should be defined before calling it
3  y = sum_num(3,2) # calling 'sum_num' function
4  print(y) # 5
5  
6  square_num = lambda n: n**2
7  z = square_num(3)
8  print(z) # 9
```

Lambda function can be used with 'map', 'filter' and 'reduce' to perform various operations on the lists as shown in next sections.

5.2.3 Map and filter

Map and filter can be used to perform above operations on list using lambda function as shown in Listing 5.4. Map can be used to store the results obtained by the operations on the elements of the list (Line 5). Further, 'filter' can be used to perform operations on certain elements from the list e.g. even numbers only, and the 'map' command can applied to selected elements (Line 14).

Listing 5.4: Map and Filter

```
1  #mapFilterEx.py
2  num = [1, 2, 3, 4, 5]
3  
4  #square all the number in list 'num' using map
5  m = map(lambda n: n**2, num)
6  print(m) # <map object at 0x000000000AE77EF0>
7  square_num = list(m) #convert 'm' into list
8  print(square_num) #[1, 4, 9, 16, 25]
9  
10 n = list(map(lambda n: n**3, num))  #list used with map in one line
11 print(n) #[1, 8, 27, 64, 125]
12 
13 #square only even number in list 'num' using map and filter
14 e = map(lambda n: n**2, filter(lambda n: n % 2 == 0, num))
15 square_even_num = list(e)
16 print(square_even_num) #[4, 16]
```

5.2.4 Reduce

Reduce can be used when we want to perform certain operations on the list e.g. adding all the elements in the list etc. Listing 5.5 adds all the numbers in the list.

Listing 5.5: Sum of the list using 'reduce'

```
#reduceEx.py
from functools import reduce
num = [1, 2, 3, 4, 5]
sumNum = reduce((lambda x, y: x + y), num)
print(sumNum) #15
```

5.3 Other iteration methods

In this section, various other iteration options are shown on lists and dictionaries.

5.3.1 Manual Iteration

__iter__ and __next__ can be used for manual iterations (iteration on demand) as shown in Listing 5.6. In the listing, 'iter' command at Line 4 starts the iteration and the 'next' command can be used to iterate the elements as shown in Lines 8-10. Since, fruits-list (Line 2) contains only 3 items, there fourth next-option (Line 13) will generate error as shown below.

Listing 5.6: Manual iteration

```
 1 #manualIterEx.py
 2 fruits = ["Apple", "Banana", "Pear"]
 3
 4 fruit = iter(fruits)  #start iteration
 5 ## above line can be written as follows,
 6 ## fruit = fruits.__iter__()
 7
 8 print(next(fruit)) # Apple
 9 # above line can be written as follows,
10 print(fruit.__next__())# Banana
11 print(next(fruit)) # Pear
12
13 ##error because list contain only 3 fruits
14 #print(next(fruit))
```

5.3.2 Implicit Iteration

Whenever there is an iteration operation on the object, then __iter__ is invoked implicitly as shown in Listing 5.7. In the listing, __iter__ method is defined at line 6 which iterate over the list using 'iter' command as discussed in Section 5.3.1.

Since, iteration operation is applied to instance of 'Fruit' class, therefore __iter__ method of the class is invoked implicitly. Note that '$*$ operations is used at Line 10, which has the same meaning as Line 12. In the other words, $*$ passes the elements of the list individually (see comments for more details). Further, if we comment, the __iter__ method in Fruit class, then following error will be generated.

Listing 5.7: Implicit iteration

```
 1 #implicitIterEx.py
 2 class Fruit:
 3   def __init__(self, *fruits):
 4     self.fruits = fruits
 5
 6   def __iter__(self):
 7     return iter(self.fruits)
 8
 9 f = ["Apple", "Banana", "Pear"]
10 fruits = Fruit(*f)  # *f send list items as 3-arguments.
11 # above line can be written as below as well.
12 # fruits = Fruit("Apple", "Banana", "Pear")
13
14 #this line is different from above two methods
15 #fruits =Fruit(f) #it will send fruites as 1-list
16
17 i=1
18 #iterating over instance of Fruit class.
19 for fruit in fruits: # __iter__ will be invoked, as iteration is applied to instance
20   print("%s: %s" % (i, fruit))
21   i+=1
22 #1: Apple
23 #2: Banana
24 #3: Pear
```

5.3.3 Generator

Functions can be converted into generators using "yield". In other words, yield command allows function to iterate as shown in Listing 5.8. Here we can see that, Line 12 calls the method 'fruitList' of 'Fruit' class at each iteration. Therefore, fruitList() cab be iterated with the help of "yield" command. Further, if we replace yield with print statement then error will be generated with message **'TypeError: Fruit object is not iterable'**.

Listing 5.8: Generator

```
#generateEx.py
class Fruit:
  def __init__(self, *fruits):
    self.fruits = fruits

  def fruitList(self):
    for fruit in self.fruits:
      yield(fruit)

fruits = Fruit("Apple", "Banana", "Pear")

for fruit in fruits.fruitList():
  print(fruit)
#Apple
#Banana
#Pear
```

5.3.4 Enumerate

In Listing 5.7, we define an incrementing variable 'i' (at Line 17) for indexing purpose. In such cases, 'Enumerate' option can be used, which iterates the index-value pair. By default, index in enumerate(self.fruits) starts from zero. enumerate(self.fruits,1) is used for starting index as 1 as shown in Line 8.

Listing 5.9: Enumerate

```
 1  #enumerateEx.py
 2  class Fruit:
 3    def __init__(self, *fruits):
 4      self.fruits = fruits
 5
 6    def fruitList(self):
 7      # 1 is used for starting index as 1
 8      for i, fruit in enumerate(self.fruits,1):
 9        yield(i, fruit)
10
11  fruits = Fruit("Apple", "Banana", "Pear")
12
13  for fruit in fruits.fruitList():
14    print(fruit)
15  #(1, 'Apple')
16  #(2, 'Banana')
17  #(3, 'Pear')
```

5.4 Zip

Zip is good for iterating over multiple sequences.

5.4.1 Iteration

If we have more than one list then it can be combined using 'zip' command. Zip checks for the shortest lists before combining them as shown in Line 7 of Listing 5.10. Here, 'list' is used to convert the output of the 'zip' command in to list. Line 10 iterates over the list and values are prited using Line 11.

If we want to combine the list based on longest list, then 'zip_longest' must be used (LIne 18) which can be imported from 'itertools'(Line 2). Missing elements in the sort lists will be filled by 'None' as shown in outputs at Lines 23 and 24. Further, desire name can be given to these missing elements using 'fillvalue' option of zip command (see Line 28).

Listing 5.10: Zip iteration

```
1  #zipIteration.py
2  from itertools import zip_longest
3
4  fruits = ["Apple", "Banana", "Pear"]
5  prices = [1, 2, 3, 4, 5]
```

```

print(list(zip(fruits, prices)))#[('Apple', 1), ('Banana', 2), ('Pear', 3)]

#zip includes shortest list and add other list to it
for fruit, price in zip(fruits, prices):
  print("%s: $%s" % (fruit, price))
#Apple: $1
#Banana: $2
#Pear: $3

#to add longest list use zip_longest
#by default is store None for undefined items
for fruit, price in zip_longest(fruits, prices):
  print("%s: $%s" % (fruit, price))
#Apple: $1
#Banana: $2
#Pear: $3
#None: $4
#None: $5

#use fillvalue to give desire name to unavailble items e.g. 'Free' in below line
for fruit, price in zip_longest(fruits, prices, fillvalue="Free"):
  print("%s: $%s" % (fruit, price))
#Apple: $1
#Banana: $2
#Pear: $3
#Free: $4
#Free: $5
```

5.4.2 Creation and operations dictionary

Output of 'zip' command can be can be covered into dictionary using 'dict' keyword as shown in Line 5 of Listing 5.11. Then various operations can be performed on the dictionary using 'zip' e.g. sorting, finding minimum and maximum etc. as shown in Listing 5.12.

Listing 5.11: Creating dictionary using zip

```
1  #zipDict.py
2  fruits = ["Apple", "Banana", "Pear"]
3  ids = [1, 2, 3]
4
5  fruitDict = dict(zip(ids, fruits))
6  print(fruitDict) #{1: 'Apple', 2: 'Banana', 3: 'Pear'}
```

Listing 5.12: Operations on dictionaries with zip

```
#operationDict.py
fruits = ["Apple", "Banana", "Pear"]
prices = [10, 12, 9]

fruitDict = dict(zip(fruits, prices))

#sort by keys as fruitDict.keys() is used first
# if fruitDict.values() is used first, then it will be sorted by price
sort_fruit_byName = sorted(zip(fruitDict.keys(), fruitDict.values()))
print("sorted alphabatically with price:", sort_fruit_byName)
#sorted alphabatically with price: [('Apple', 10), ('Banana', 12), ('Pear', 9)]

# min: for finding minimum value as fruitDict.values() is used first
min_price = min(zip(fruitDict.values(), fruitDict.keys()))
print("Cheapest Fruit: ", min_price)
#Cheapest Fruit:  (9, 'Pear')

#see below code as well
max_price = max(fruitDict)
print(max_price)  #Pear i.e. according to letter

max_price = max(fruitDict.values())
print(max_price) #12 i.e. maximum price
```

5.5 Conclusion

In this chapter, various iterations operations on list, i.e. generator, enumerate, map and filter etc. are discussed. Finally, zip command is used to convert the list into dictionary and then minimum, maximum and sort etc. operations are performed on dictionaries.

Forgiveness is the best charity. It is easy to give the poor money and goods when one has plenty, but to forgive is hard; but it is the best thing if one can do it.

–***Meher Baba***

Chapter 6

Object Oriented Programming

6.1 Introduction

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.

6.2 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.

6.2.1 Create class and object

Class is created using keyword 'class' as shown in Line 4 of Listing 6.1, 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 6.1: 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>
```

6.2.2 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 6.2.3.

Listing 6.2: 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
```

6.2.3 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 6.4)**.** *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 6.3: 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
```

6.2.4 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 6.4. This boiler-plate is discussed in Section 4.4, which tells the python-interpretor that the ‘main’ is the starting point of the code.

Listing 6.4: Constructor with default values

```python
#ex4.py

#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)

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

    # if no name is passed, the default value i.e. Unknown will be used
    k = Jungle()
    k.welcomeMessage() # Hello Unknown, Welcome to the Jungle

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

6.2.5 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 6.5; whereas save the ‘main()’ in ‘main.py’ file as shown in Listing 6.6. 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 6.6.

Listing 6.5: Save classes in separate file

```python
#jungleBook.py

#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)
```

Listing 6.6: Import class to main.py

```python
 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()
```

6.3 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 6.7; 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 6.8, then this object 'r' will have the access to 'visitorName' as well (which is in the parent class). Note that, inheritance is discussed in detail in Chapter 8.

Listing 6.7: 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):
10         print("Hello %s, Welcome to the Jungle" % self.visitorName)
11
12 class RateJungle(Jungle):
13     def __init__(self, name, feedback):
14         # feedback (1-10) :  1 is the best.
15         self.feedback = feedback # Public Attribute
16
17         # inheriting the constructor of the class
18         super().__init__(name)
19
20     # using parent class attribute i.e. visitorName
21     def printRating(self):
22         print("Thanks %s for your feedback" % self.visitorName)
```

Listing 6.8: Usage of parent-class method and attributes in child-class

```
1  #main.py
2
3  ## import class 'Jungle' and 'RateJungle' from jungleBook.py
4  from jungleBook import Jungle, RateJungle
5
6  def main():
7      r = RateJungle("Meher", 3)
8
9      r.printRating() # Thanks Meher for your feedback
10
11     # calling parent class method
12     r.welcomeMessage() # Hello Meher, Welcome to the Jungle
13
14 # standard boilerplate to set 'main' as starting function
15 if __name__=='__main__':
16     main()
```

6.4 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 6.9, 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 6.10. 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 6.9: Polymorphism example with function 'move'

```
1  #scarySound.py
2
3  class Animal:
4      def scarySound(self):
5          print("Animals are running away due to scary sound.")
6
7  class Bird:
8      def scarySound(self):
9          print("Birds are flying away due to scary sound.")
10
```

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

Listing 6.10: 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()
```

6.5 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 6.11. Since, 'scarySound' is defined as abstractmethod at Line 15-17, therefore it is compulsory to implement it in all the subclasses. **Look at the class 'Insect' in Listing 6.9, 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 6.12.**

Listing 6.11: Abstract class and method

```
1   #jungleBook.py
2
3   from abc import ABCMeta, abstractmethod
4
5   #Abstract class and abstract method declaration
6   class Jungle(metaclass=ABCMeta):
7       #constructor with default values
8       def __init__(self, name="Unknown"):
9           self.visitorName = name
10
11      def welcomeMessage(self):
12          print("Hello %s, Welcome to the Jungle" % self.visitorName)
13
14      # abstract method is compulsory to defined in child-class
15      @abstractmethod
16      def scarySound(self):
17          pass
```

Listing 6.12: Abstract methods are compulsory to define in child-class

```
1   #scarySound.py
2
3   from jungleBook import Jungle
4
5   class Animal(Jungle):
6       def scarySound(self):
7           print("Animals are running away due to scary sound.")
8
9   class Bird(Jungle):
10      def scarySound(self):
11          print("Birds are flying away due to scary sound.")
12
13  # 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 6.13: 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()
```

6.6 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 6.14 and 6.15 show the example of attributes along with the methods to access them. Please read the comments to understand these codes.

Listing 6.14: Public and private attribute

```
#jungleBook.py

#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
```

```python
        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 6.15: Accessing public and private attributes

```python
#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

    ## access to private attribute is not allowed
    ## uncomment following line to see error
    # print("Jungle Guide rating : ", r.__jungleGuideRating)

    ## private attribute can still be accessed as below,
    ## objectName._className.__attributeName
    print ("Guide rating : ",  r._RateJungle__jungleGuideRating) # Guide rating :  110

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

6.7 Class Attribute

Class attribute is the variable of the class (not of method) as shown in Line 7 of Listing 6.17. 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 6.16: Class attributes and it's access

```python
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.
```

```
        self.feedback = feedback # Public Attribute

        # add new feedback value to sum_of_feedback
        Jungle.sum_of_feedback += self.feedback
        # increase total number of feedback by 1
        RateJungle.total_num_feedback += 1

        # 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)

class Test:
    def __init__(self):
        # inheritance is not required for accessing class attribute
        print("sum_of_feedback (Jungle class attribute) : ", Jungle.sum_of_feedback)
        print("total_num_feedback (RateJungle class attribute) : ", RateJungle.total_num_feedback)
```

Listing 6.17: Main program

```
#main.py

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

def main():
    r = RateJungle("Meher", 3)
    s = RateJungle("Krishna", 2)

    r.averageFeedback() # Average feedback :  2.5


    # Test class is using other class attributes without inheritance
    w = Test()
    ''' sum_of_feedback (Jungle class attribute) :  5.0
        total_num_feedback (RateJungle class attribute) :  2
    '''

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

6.8 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.

6.8.1 __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 6.18, 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 6.18: __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
11      def __del__(self):
12          Student.totalStudent -= 1
13          print("Total Students (del) : ", self.totalStudent)
14
```

```python
def main():
    s1 = Student("Meher") # Total Students (init) :  1
    s2 = Student("Krishna") # Total Students (init) :  2
    s3 = Student("Patel") # Total Students (init) :  3

    ## delete object s1
    del s1 # Total Students (del) :  2

    # print(s1.name) # error because s1 object is deleted
    print(s2.name) # Krishna
    print(s3.name) # Patel

    ## since there is no further statements, therefore
    ## 'del' will be executed for all the objects and
    ## following results will be displayed

    # Total Students (del) :  1
    # Total Students (del) :  0

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

6.8.2 __str__

When __str__ is defined in the class, then 'print' statement for object (e.g. print(j) at Line 11 of Listing 6.19), will execute the __str__ statement, instead of printing the address of object, as happened in Listing 6.1. This statement is very useful for providing the useful information about the class using print statement.

Listing 6.19: __str__ method is executed when 'print' statement is used for object

```python
 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()
```

6.8.3 __call__

The __call__ method is executed, when object is used as function, as shown in Line 20 of Listing 6.20; where object 'd' is used as function i.e. d(300).

Listing 6.20: __call__ method is executed when object is used as function

```python
 1  # callEx.py
 2
 3  #class declaration
 4  class CalculatePrice:
 5      # discount in %
 6      def __init__(self, discount):
 7          self.discount = discount
 8
 9      def __call__(self, price):
10          discountPrice = price - price*self.discount/100
11          return (price, discountPrice)
12
13  def main():
14      # create object of class CalculatePrice with 10% discount
15      d = CalculatePrice(10)
16
17      # using object as function i.e. d(300)
18      # since two variables are return by call fuction, therefore
19      # unpack the return values in two variables
20      price, priceAfterDiscount =  d(300)
21      print("Original Price: %s,  Price after discount : %s "
22                      % (price, priceAfterDiscount))
23
```

```
    ## 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()
```

6.8.4 __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 6.21: __dict__ and __doc__

```
 1 #dictEx.py
 2 
 3 #class declaration
 4 class Jungle:
 5     """ List of animal and pet information
 6         animal = string
 7         isPet = string
 8     """
 9     def __init__(self, animal="Elephant", isPet="yes"):
10         self.animal =  animal
11         self.isPet = isPet
12 
13 def main():
14     # create object of class Jungle
15     j1 = Jungle()
16     print(j1.__dict__) # {'isPet': 'yes', 'animal': 'Elephant'}
17 
18     j2 = Jungle("Lion", "No")
19     print(j2.__dict__) # {'isPet': 'No', 'animal': 'Lion'}
20 
21     print(Jungle.__dict__)
22     """ {'__doc__': '__doc__': ' List of animal and pet information \n
23                 animal = string\n      isPet = string\n    ',
24         '__weakref__': <attribute '__weakref__' of 'Jungle' objects>,
25         '__module__': '__main__',
26         '__dict__': <attribute '__dict__' of 'Jungle' objects>,
27         '__init__': <function Jungle.__init__ at 0x00466738>}
28     """
29 
30     print(Jungle.__doc__)
31     """List of animal and pet information
32         animal = string
33         isPet = string
34     """
35 
36 # standard boilerplate to set 'main' as starting function
37 if __name__=='__main__':
38     main()
```

6.8.5 __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 6.22. Please read the comments of Listing 6.22 for better understanding. Similarly, __getattr__ is invoked whenever we try to access the value of an attribute, **which is not in the dictionary**.

Listing 6.22: __setattr__ and __getattr__

```
 1 # setAttr.py
 2 class StudentID:
 3     def __init__(self, id, name, age = "30"):
 4         self.id = id
 5         self.firstName = name
 6         self.age = age
 7 
 8     # all the init parameters need to be specified in 'setattr'
 9     def __setattr__(self, name, value):
10         if(name == "id"): # setting id
11             if isinstance(value, int) and value > 0 :
```

```python
                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'
    # 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()
```

6.9 Conclusion

In this chapter, we learn various features of object oriented programming in Python. 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.

Slowly slowly O mind, Everything in own pace happens. Gardner may water a hundred buckets, but fruit arrives only in its season.

–Kabeer

Chapter 7

Simulation using OOP

7.1 Introduction

In this chapter, the BSPK simulator is implemented using OOP method. The theory is explained in in Section 4.5; also the logic in the codes are same, but the OOP method is used for implementation.

7.2 BPSK system using OOP

In Listing 7.1, the class BPSK is created at Line 6, which contains various methods e.g. signal_generator (Line 11) and noise_generator (Line 24) etc. for the BPSK signals. Then these functions are used in 'main' function as shown in Listing 7.2. In main function, Lines 27-32 uses the methods in class BPSK. It can be seen that it is very easy for the users to use these methods to generate various kinds of signal; but at the same time the complexity of the code is increased.

Listing 7.1: BPSK class for various signal generation

```
 1  # BPSK.py
 2
 3  import numpy as np
 4  from scipy.special import erfc
 5
 6  class BPSK(object):
 7    def __init__(self, bit_length):
 8      self.bit_length = bit_length
 9
10    #binary signal generator
11    def signal_generator(self):
12      #uniformly distributed sequence
13      b = np.random.uniform(-1, 1, self.bit_length)
14
15      #convert to binary and save in signal
16      signal = np.zeros((self.bit_length),float)
17      for i in range(self.bit_length):
18        if b[i] < 0:
19          signal[i]=-1
20        else:
21          signal[i]=1
22      return signal
23
24    def noise_generator(self):
25      #Gaussian Noise
26      noise = np.random.randn(self.bit_length)
27      return noise
28
29    def recieved_signal(self, signal, SNR, noise):
30      recieved_signal = signal + SNR*noise
31      return recieved_signal
32
33    def detected_signal(self, recieved_signal):
34      detected_signal = np.zeros((self.bit_length),float)
35      for i in range(self.bit_length):
36        if recieved_signal[i] < 0:
37          detected_signal[i]=-1
```

```
      else:
        detected_signal[i]=1
    return detected_signal

  def error(self, signal, detected_signal):
    error_matrix = abs((detected_signal - signal)/2)
    error=error_matrix.sum()
    # print error
    return error

  def theoryBerBPSK(self, SNR_db):
    #calculate theoritical BER
    theoryBER = np.zeros(len(SNR_db),float)
    for i in range(len(SNR_db)):
      theoryBER[i] = 0.5*erfc(np.sqrt(10**(SNR_db[i]/10)))
    return theoryBER
```

Listing 7.2: Main function to generate BPSK BER curve

```
# main.py
import matplotlib.pyplot as plt
import numpy as np

from BPSK import BPSK

def main():
  bit_length  = 10000
  iter_len=30

  #noise calculations
  #SNR db range: -2 to 10
  SNR_db = np.array(np.arange(-2, 10, 1),float)
  SNR = np.zeros(len(SNR_db), float)
  for i in range (len(SNR)):
    SNR[i]= 1/np.sqrt(2)*10**(-SNR_db[i]/20)

  #instance of BPSK class
  bpsk = BPSK(bit_length)

  #accumulate bit error rate for various iterations
  instantaneous_error =np.zeros((iter_len, len(SNR_db)), float)
  for iter in range(iter_len):
    error_matrix =np.zeros(len(SNR_db), float)

    for i in range(len(SNR)):
      signal =  bpsk.signal_generator()
      noise = bpsk.noise_generator()
      recieved_signal = bpsk.recieved_signal(signal, SNR[i], noise)
      detected_signal = bpsk.detected_signal(recieved_signal)
      error = bpsk.error(signal, detected_signal)
      error_matrix[i] = error

    instantaneous_error[iter]=error_matrix

  #Average BER
  BerBPSK = instantaneous_error.sum(axis=0)/(iter_len*bit_length)

  #calculate theoritical BER
  theoryBerBPSK = bpsk.theoryBerBPSK(SNR_db)

  #plot data (not from the .csv file)
  plt.semilogy(SNR_db, BerBPSK,'--')
  plt.semilogy(SNR_db, theoryBerBPSK, 'mo')
  plt.ylabel('BER')
  plt.xlabel('SNR')
  plt.title('BPSK BER Curves')
  plt.legend(['Simulation', 'Theory'], loc='upper right')
  plt.grid()
  plt.show()

# Standard boilerplate to call the main() function.
if __name__ == '__main__':
  main()
```

7.3 Conclusion

In this chapter, we reimplemented the BPSK code of Section 4.5 using OOP method.

Selfness for all brings about undisturbed harmony without loss of discrimination, and unshakable peace without indifference to the surroundings.

–***Meher Baba***

Chapter 8

Inheritance: super()

8.1 Introduction

In this chapter, 'super' method is discussed for inheritance in Python. Further, multiple inheritance with different parameters in the 'init' function, is also discussed.

8.2 Inherit '__init__' from parent class

If we want to inherit the '__init__()' function of the parent class, then 'super' function is required. In Listing 8.1, the class 'rectArea' is inheriting the __init__ function of the class 'reacLen'. After inheriting the value of length, the area is calculated.

Explanation (Listing 8.1). *In the listing, two classes are defined at Lines 2 and 6. The class 'rectLen' (Line 2) contains the 'init' function which initialize the length variable. The other class i.e. 'rectArea' inherits the 'rectLen' class at line 6. Then in Line 7, the 'rectArea' class inherits the 'init' function using 'super' command, which initializes the value of 'length'. Next Line 8, initializes the value of 'width' and then area is calculated at Line 9. Note that, 'return' function can not be used inside the 'init' method i.e. 'init' can not return any value. If we uncomment the line 11, Python interpretor will generate the error.* ▲

Listing 8.1: Inherit '__init__' function from parent class

```
 1  #initSuper.py
 2  class rectLen:
 3    def __init__(self, length):
 4      self.length = length
 5
 6  class rectArea(rectLen):
 7    def __init__(self, length, width):
 8      super().__init__(length) # inheriting `init' fucntion from 'rectLen'
 9      self.width = width
10      print("Area: ", self.length * self.width)  # Area: 12
11      # return(self.length, self.width) #wrong, as init returns None
12
13  # create object of the class 'rectArea'
14  r = rectArea(3,4)
15  print("Length = %s, Width = %s" % (r.length, r.width)); #Length = 3, Width = 4
```

8.3 Inherit 'methods' from parent class

If we have same method names in 'parent' and 'child' classes, then Python by default uses the 'child class' method. In such cases, 'super' function is required to use the 'parent class' method in 'child class', as shown in Listing 8.2.

Explanation (Listing 8.2)**.** *In the listing, the parent class 'A' and the child class 'B' have the same method name i.e. 'printClass' at Lines 3 and 7. Line 15 creates a instance of class B, and then 'printName' method is called at Line 16. The 'printName' class at Line 10, first calls the 'printClass' method of 'class B' at Line 11; and then in Line 12, it uses the 'super' function to call the 'printClass' method of parent class.* ▲

Listing 8.2: Inherit 'method' from parent class for same method names

```
 1  #methodSuper.py
 2  class A:
 3    def printClass(self):
 4      print("A")
 5
 6  class B(A):
 7    def printClass(self):
 8      print("B")
 9
10    def printName(self):
11      B.printClass(self); # B
12      super().printClass(); # A
13
14  #instant of B
15  b = B()
16  b.printName() # calling printClass
```

8.4 Inherit multiple classes

In this section, multiple classes are inherited by the child class. Careful coding is required for such cases, because for smaller codes, we may get correct results using wrong and easy methods. But, in complex codes, such wrong practices may generate errors. In following sections, some of the problems are discuss, along with the final correct solution of the problems.

8.4.1 Problem 1: super() calls only one class __init__

Note that, super() function can call the 'init' function of only one class at a time. If Listing 8.3, class C (Line 11) is inheriting class A and B. But the super() command only inherit the __init__ function of class A because it appeared first in Line 11. Therefore, output of Line 13, will be 'A'. Similarly, if we replace the 'C(A,B)' with 'C(B,A)' at line 11, then output of Line 13 will be 'B'.

Listing 8.3: 'super' inherit only one 'init' function

```
 1  #superInheritOneClass.py
 2  class A:
 3    def __init__(self):
 4      print("A")
 5
 6  class B:
 7    def __init__(self):
 8      print("B")
 9
10  # A is written before B
11  class C(A,B): #therefore super will inherit A
12    def __init__(self):
13      super().__init__() # A
14
15  #instance of C
16  c=C()
```

8.4.2 Wrong solution to problem 1

To Inherit both the __init__ function, we can call each __init__ function explicitly as shown in Listing 8.4. In this Listing, two 'super' functions are used to call the 'init' function of two classes. This solution looks correct, but may create problems as shown in Section 8.5.

Listing 8.4: ‘super’ inherit only one ‘init’ function

```python
#multipleSuper.py
class A:
  def __init__(self):
    print("A")

class B:
  def __init__(self):
    print("B")

class C(A,B):
  def __init__(self):
    A.__init__(self) # A
    B.__init__(self) # B

c=C()
```

8.4.3 Correct solution

We need to define ‘super’ function in each parent classes, which is the correct solution to the problem of multiple inheritance. The ‘super’ function is defined in the parent classes at Lines 5 and 11 of Listing 8.5.

Explanation (Order of outputs in Listing 8.5)**.** *In the listing, the class ‘C’ (Line 20) inherits threes classes i.e. ‘B’, ‘A’ and ‘D’. When, class ‘C’ is instantiated at Line 25, then the ‘init’ functions are inherited by Line 22. Line 22, first inherits the ‘init’ function of ‘class B’, because ‘B’ appears first at Line 20. Hence, line 10 (i.e. ‘reached B’) is printed first. Next, Line 11 uses the ‘super().__init__’, therefore Python interpretor goes to ‘init’ function of next class i.e. A and prints ‘reached A’ from Line 4. Line 5, again uses the ‘super’ function, therefore interpretor goes to next class i.e. ‘D’ and prints the ‘reached D’ from Line 16. Then again, ‘super’ function is used, but this time there is no other class to inherit, therefore interpretor goes to the line 18 and prints ‘D’ and returns to the ‘super’ of previous class (i.e. Line 5) and prints the next Line i.e. ‘A’. Next, it goes to ‘super’ of class ‘B’ (i.e. Line 12) and finally it prints the Line 13 i.e. ‘B’.*

Remember that, the statements which appear after the ‘super’ functions, the print orders are opposite to the order of inheritances (see outputs at Lines 29-31); whereas the statements which appear before the super, have the same sequences as the inheritance sequences (see Line 26-28).

▲

Listing 8.5: ‘super’ in parent class for multiple inheritance

```python
1  #superParentClass.py
2  class A:
3    def __init__(self):
4      print("reached A")
5      super().__init__()
6      print("A")
7
8  class B:
9    def __init__(self):
10     print("reached B")
11     super().__init__()
12     print("B")
13
14 class D:
15   def __init__(self):
16     print("reached D")
17     super().__init__()
18     print("D")
19
20 class C(B,A,D):
21   def __init__(self):
22     super().__init__()
23
24 # instantiate the class C
```

```
c=C() # see the order of outputs
# reached B
# reached A
# reached D
# D
# A
# B
```

8.5 Math problem

This section summarizes the above examples using a mathematical problem. Here we want to calculate $x * 2 + 5$ where $x = 3$. In case 1 and case 2, __init__ function is invoked directly by calling it using class name, which is the wrong way to solve the problem as discussed in Section 8.4.

8.5.1 Case 1: Wrong method

Listing 8.6 gives the correct answer to the problem, but the answer depends on the order of invoking methods (i.e. Line 13 and 14) rather that order of inheritance (i.e. Line 10). e.g. if we interchange the line 14 and 13, the answer will be changed as shown in next section.

Listing 8.6: Answer depends on order of 'invoking' the class rather the inheritance

```
1  #mathWrong1.py
2  class plus5:
3    def __init__(self):
4      self.value += 5
5
6  class multiply2:
7    def __init__(self):
8      self.value *=  2
9
10 class C(multiply2, plus5):
11   def __init__(self, value):
12     self.value = value
13     multiply2.__init__(self) # B
14     plus5.__init__(self) # A
15
16 c=C(3)
17 print (c.value) #11 i.e. 3*2+6
```

8.5.2 Case 2: Problem with case 1

In Listing 8.7, line 13 and 14 of Listing 8.6 are interchanged, therefore the solution is also changed as shown in Line 17. Since, lines are interchanged, therefore 'plus' operations is performed first, and then multiplication is performed.

Listing 8.7: Problem with Listing 8.6

```
1  #mathWrong2.py
2  class plus5:
3    def __init__(self):
4      self.value += 5
5
6  class multiply2:
7    def __init__(self):
8      self.value *=  2
9
10 class C(multiply2, plus5):
11   def __init__(self, value):
12     self.value = value
13     plus5.__init__(self)
14     multiply2.__init__(self)
15
16 c=C(3)
17 print (c.value) #16 i.e. (3+5)*2
```

8.5.3 Case 3: Correct method

Here, 'super' method is used as discussed in Section 8.4.3. Now, solution depends on the order of inheritance, which is easier to manage than 'Case 1'.

Listing 8.8: Answer depends on the order of inheritance

```python
#mathCorrect.py
class plus5:
  def __init__(self):
    self.value += 5
    super().__init__()

class multiply2:
  def __init__(self):
    self.value *=  2
    super().__init__()


class C(multiply2, plus5):
  def __init__(self, value):
    self.value = value
    super().__init__()

c=C(3)
print (c.value) #11 i.e. 3*2+5
```

8.6 Different parameters in __init__

In all the above examples, __init__ has only one parameter i.e. "self". If classes have different parameters in __init__ functions, then above examples will not work. For different parameters in inheriting classes, we need to modify our code so that multiple inheritance work properly. In Listing 8.9, **kwargs is used to solve the problem. Note that, **kwargs is used in all the '__init__' methods i.e. Lines 3, 9 and 1, because the 'init' functions have different parameters at these lines. Also, **kwargs is used with super() functions i.e. at Lines 6, 12 and 17. In this way, problem of multiple parameters in 'init' functions can be solved.

Listing 8.9: Different parameters in __init__

```python
 1  #multipleInitParameter.py
 2  class plus5:
 3    def __init__(self, price="", **kwargs):
 4      self.value += 5
 5      print("price", price) # price 8
 6      super().__init__(**kwargs)
 7
 8  class multiply2:
 9    def __init__(self, quantity="", **kwargs):
10      self.value *=  2
11      print("quantity", quantity) # quantity 6
12      super().__init__(**kwargs)
13
14  class C(multiply2, plus5):
15    def __init__(self, value, **kwargs):
16      self.value = value
17      super().__init__(**kwargs)
18
19  #instantiate C
20  c=C(3, quantity=6, price=8)
21  print ("Value: =", c.value) # 11 i.e. 3*2+5
```

8.7 Conclusion

In this chapter, we saw the use of 'super' function for various types of inheritance situations. Also, we discuss the multiple inheritance with different parameters in 'init' functions.

Mental suffering is worse than physical suffering. Physical suffering sometimes comes as a blessing because it serves the purpose of easing mental suffering by weaning away mans attention from the mental suffering.

–Meher Baba

Chapter 9

Function, decorator, @property and descriptor

9.1 Introduction

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 9.2.1.

9.2 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.

9.2.1 Help feature

In Listing 9.1, 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 that, 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 9.1: 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
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
```

9.2.2 Keyword argument

In Listing 9.2, ‘$addKeywordArg(x, *, y)$’ 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 9.2: 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)
```

9.2.3 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 9.3, 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 9.3: 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'>}
```

9.3 Decorators

Decorators are used to add additional functionalities to functions. In Section 9.2.3, '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 9.6 to write the generalized validation which can validate any kind of input.

9.3.1 Add decorators and problems

Listing 9.4 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 9.5.

Listing 9.4: Decorator which prints the name of function

```
# funcNameDecorator.py
def funcNameDecorator(func): # function as input
  def printFuncName(*args, **kwargs): #take all arguments of function as input
    print("Function Name:", func.__name__) # print function name
    return func(*args, **kwargs) # return function with all arguments
  return printFuncName
```

In Listing 9.5, 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.

Problem: 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 9.5: 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 that, it is recommonded to define the decorators in the separate files e.g. 'funcNameDecorator.py' file is used here. Its not good practice to define decorator in the same file, it may give some undesired results.

9.3.2 Remove problems using functools

The above problem can be removed by using two additional lines in Listing 9.4. The listing is saved as Listing 9.6 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 9.7 is same as Listing 9.5 except new decorator which is defined in Listing 9.6 is called at line 4.

Listing 9.6: 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 9.7: Help features are visible again

```
1   #addIntDecoratorFunctool.py
2   from funcNameDecoratorFunctool import funcNameDecoratorFunctool
3
4   @funcNameDecoratorFunctool
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  intAdd=addIntDecorator(2, y=3) # Function Name: addIntDecorator
14  print("intAdd =", intAdd) # 5
15
16  help(addIntDecorator) # lines 6-9 will be displaed
17
18  print(addIntDecorator.__annotations__)
19  ##{'return': <class 'int'>, 'y': <class 'int'>, 'x': <class 'int'>}
```

9.4 @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 9.8). *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.

***Problem:** 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 9.8: Attribute validation using @property

```
 1  #rectProperty.py
 2  class Rectangle:
 3    '''
 4    -Calculate Area and Perimeter of Rectangle
 5    -getter and setter are used to displaying and setting the length value.
 6    -width is set by init function
 7    '''
 8
 9    # self.length is used in below lines,
10    # but length is not initialized by __init__,
11    # initialization is done by .setter at lines 34 due to @property at line 27,
12    # also value is displayed by getter (@propety) at line 28-32
13    # whereas `width' is get and set as simple python code and without validation
14    def __init__(self, length, width):
15      #if self._length is used, then it will not validate through setter.
16      self.length = length
17      self.width = width
18
19    @property
20    def area(self):
21      '''Calculates Area: length*width'''
22      return self.length * self.width
23
24    def perimeter(self):
25      '''Calculates Perimeter: 2*(length+width)'''
26      return 2 * (self.length + self.width)
27
28    @property
29    def length(self):
30      '''displaying length'''
31      print("getting length value through getter")
32      return self._length
33
34    @length.setter
35    def length(self, value):
36      '''setting length'''
37      print("saving value through setter", value)
38      if not isinstance(value, int): # validating length as integer
39        raise TypeError("Only integers are allowed")
40      self._length = value
41
42  r = Rectangle(3,2) # following output will be displayed
43  ## saving value through setter 3
44  print(r.length) # following output will be displayed
45  ## getting length value through getter
46  ## 3
47
48  ## @property is used for area,
49  ## therefore it can be accessed directly to display the area
50  print(r.area) # following output will be displayed
51  ## getting length value through getter
52  ## 6
53
54  #r=Rectangle(4.3, 4) # following error will be generated
55  ## [...]
56  ## TypeError: Only integers are allowed
57
58  # print perimeter of rectangle
59  print(Rectangle.perimeter(r))
60  ## getting length value through getter
61  ## 10
```

9.5 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 9.9. In this section, validation is applied using Descriptor to remove the problem with @property.

Explanation (Listing 9.9). *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 9.9: 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
```

9.6 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 that, various types i.e. '@typeAssert(author=str, length=int, width=float)' will be defined by class Author for validation.

Explanation (Listing 9.10). *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 9.10: Generalized attribute validation

```
#rectGeneralized.py
class TypeCheck:
  def __init__(self, parameter, expected_type):
    self.parameter = parameter
    self.expected_type = expected_type

  def __get__(self, instance, cls):
    if instance is None: # required if descriptor is
      return self # used as class variable
    else: # in this code, only following line is required
      return instance.__dict__[self.parameter]

  def __set__(self, instance, value):
    print("setting %s to %s" % (self.parameter, value))
    if not isinstance(value, self.expected_type):
      raise TypeError("%s value is expected" % self.expected_type)
    instance.__dict__[self.parameter] = value

def typeAssert(**kwargs):
  def decorate(cls):
    for parameter, expected_type in kwargs.items():
      setattr(cls, parameter, TypeCheck(parameter, expected_type))
    return cls
  return decorate

 # define attribute types here in the decorator
@typeAssert(author=str, length = int, width = float)
class Rect:
  def __init__(self, *,  length, width, author = ""): #require kwargs
    self.length = length
    self.width = width * 1.0 # to accept integer as well
    self.author = author

r = Rect (length=3, width=3.1, author = "Meher")
## setting length to 3
## setting width to 3.1
## setting author to Meher

#r = Rect (length="len", width=3.1, author = "Meher") # error shown below
## File "rectProperty.py", line 42,
## [ ... ]
## TypeError: <class 'int'> value is expected
```

9.7 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.

True happiness begins when a man learns the art of right adjustment to other persons, and right adjustment involves self-forgetfulness and love.

–Meher Baba

Chapter 10

More on Methods

10.1 Introduction

In this chapter, differences between three types of methods, i.e. 'methods', 'class methods' and 'static methods', are discussed. Further, decorators are defined inside the class as 'instance method' and 'class methods'.

10.2 Method, @classmethod and @staticmethod

Listing 10.1 adds two variables i.e. 'x' and 'y' using different types of method. The differences between the outputs are used to illustrate the functioning of these method.

Explanation (Listing 10.1)**.** *In the listing, the value of 'x' is set to 5 in class 'Add' at Line 3. Then, the value of 'x = 4' is passed while creating the different instances of the class i.e. at Lines 21, 25 and 37. We will discuss all these instances along with the outputs in the following paragraphs.*

First instance of the class is created at Line 21, which initialize the value of x=4. Then Line 23 calls the method 'addFunc' at line 22 with the value of 'y' as 3. Since x=4 and y=3, therefore the output value is returned as 7 by Line 9.

Again, second instance is created at Line 25 with the value of x = 4. But this time class method is called which is defined at Lines 11-14. Note that '@classmethod' decorator is used to declare the method as 'class method'. When a method is declared as 'class method', then it will use that value of variable which is define inside the class (not the initialized value); i.e. x= 5 will be used by class method (not initialized value i.e. x =4). Since, y=3 is passed by Line 28, therefore output of the this method will be '5+3=8'. Further, class methods can be directly called by the main program as shown in Line 31. Lastly, if x = 5 is not defined at Line 3, then 'self.x' (at Line 14) will generate the attribute error i.e. "AttributeError: type object 'Add' has no attribute x ".

Lastly, static method is defined at Lines 16-18 using '@staticmethod' keyword. Note that, 'self' is not used with static methods i.e. Line 17 does not contain 'self.x' in the code. Also value of x is defined as 12 in main code at Line 35. Then, the instance of class 'Add' is created at line 37, where x is initialized as 4. Note that, static method does not use the initialized values; it will use that value of x which is define in main program i.e. at Line 35. If we do not define it, the code will generate error. Since the value of x is 12 and Line 38 passes y=3, therefore sum will be 15 in this case. Similar to class methods, static method can be called directly from the main function as shown in Line 41. ▲

Listing 10.1: Different types of methods

```python
#methodEx.py
class Add:
  x = 5
  def __init__(self, x):
    self.x = x

  #simple method
  def addFunc(self, y):
    print ("method:", self.x+y)

  @classmethod
  #as convention, cls must be in place of self
  def addClass(self, y): # i.e. def addClass(cls, y): is the correct way
    print ("class:", self.x+y)

  @staticmethod
  def addStatic(y): #no self is used here
    print("static:", x+y)

#METHOD
aFunc = Add(4)
aFunc.addFunc(3) # method: 7

#CLASS METHOD
aClass = Add(4)
#below line will not use init function's self.x = 4
#it will use class variable value i.e. x = 5
aClass.addClass(3) # class: 8

#Directly calling @classmethod
Add.addClass(3) # class: 8

#STATIC METHOD
# x is defined here for static method, it will get it from the class
x=12

aStatic = Add(4)
aStatic.addStatic(3) #static: 15

#Directly calling @staticmethod
Add.addStatic(3) #static: 15
```

Following conclusions can be drawn from the Listing 10.1,

1. 'Method' uses the initialized values of variables (i.e. self.x) for operations.
2. 'classMethod' uses that value of variable which is define inside the class (not the initialized value).
3. 'staticMethod' uses that value of variable which is define in the main function; i.e. it neither uses the initialized value nor that value which is defined inside the class.
4. Both static method and class method can be called directly from the main function.

10.3 Decorator inside the class

In Section 9.3, the decorators are defined as functions. In this section, we will define the decorators as the methods inside the class. In Listing 10.2, decorators are defined as "instance method" and "class method".

Explanation (Listing 10.2). *In the listing, two decorators are defined inside the class 'Date' i.e. at Lines 6 and 14. The first decorator (i.e. at Line 6) is the simple decorator, but it is inside the class, therefore it is called 'decorator as instance method'. To use this decorator, first we need to create the instance of the class 'Date'. This is done at Line 22. Then, in line 23, this instance is used to apply the decorator on the 'sum' function at line 22. Line 31 calls the 'sum' function and date and time is printed by Line 8 before displaying the result.*

The second decorator (i.e. at Line 15) is called the class decorator, because @classmethod is used before it i.e in Line 14. In previous section, we saw that the class method can be called directly from the main function. Hence, there is no need to create an instance for this decorator and it can be applied directly as shown in Line 27; where decorator is applied to 'sub' method. Line 31 calls the 'sub' function and date and time is printed by Line 17 before displaying the result. ▲

Listing 10.2: Decorator inside the class method

```python
#decoratorMethod.py
from datetime import datetime

class Date:
  # decorator as instance method
  def instanceMethodDecorator(self, func): #day-month-year
    def wrapper(*args, **kwargs):
      print("instanceMethodDecorator called at time:",
        datetime.today())
      return func(*args, **kwargs)
    return wrapper

  # decorator as class method
  @classmethod
  def classMethodDecorator(self, func): #day-month-year
      def wrapper(*args, **kwargs):
        print("classMethodDecorator called at time:",
          datetime.today())
        return func(*args, **kwargs)
      return wrapper

a = Date()
@a.instanceMethodDecorator
def add(a,b): #decorated using instanceMethodDecorator
  print(a+b)

@Date.classMethodDecorator
def sub(a,b): #decorated using classMethodDecorator
  print(a-b)

sum = add(3, 2)
# instanceMethodDecorator called at
# time: 2016-06-02 06:18:14.776555
# 5

sub = sub(3,2)
# classMethodDecorator called at
# time: 2016-06-02 06:18:14.776555
# 1
```

10.4 Multiple Constructor

Multiple constructor can be defined using @classmethod as shown in Listing 10.3. Listing is quite simple, where Date-objects can be initialized in three different format i.e. ymd (year-month-day), dmy and mdy.

Note that, the order of elements in return statements (i.e. Lines 18 and 27) must be exactly same as in the 'init' function in Line 3. It is essential because these return values are sent back to 'init' function, which finally store the values. For better understanding of the code, first uncomment the Lines 13-22 and then execute the code.

Listing 10.3: Multiple Constructor

```python
 1  # multipleConstructor.py
 2  class Date:
 3      def __init__(self, year, month, day): #year-month-day
 4          self.year = year
 5          self.month = month
 6          self.day = day
 7
 8      def __str__(self):
 9          return("Date in dmy format : %s-%s-%s" % (self.day, self.month, self.year))
10
11      @classmethod
12      def dmy(cls, day, month, year): #day-month-year
13          # print("dmy")
14          cls.year = year
15          cls.month = month
16          cls.day = day
17          #order of return should be same as init
18          return cls(cls.year, cls.month, cls.day)
19
20      @classmethod
21      def mdy(cls, month, day, year): #month-day-year
22          # print("mdy")
23          cls.year = year
24          cls.month = month
25          cls.day = day
```

```
        #order of return should be same as init
        return cls(cls.year, cls.month, cls.day)

a=Date(2016, 12, 11)
print(a) # Date in dmy format : 11-12-2016

b=Date.dmy(9, 10, 2015)
print(b) # Date in dmy format : 9-10-2015

c=Date.mdy(7, 8, 2014)
print(c) # Date in dmy format : 8-7-2014
```

10.5 Conclusion

In this chapter, we see the various types of methods i.e. simple method, static method and class method. Also, we learn the methods by which the decorators can be defined inside the class.

Feelings of heat and cold, pleasure and pain, are caused by the contact of the senses with their objects. They come and they go, never lasting long. You must accept them.

–Krishna

Appendix A

Saving results without SPYDER

Listing A.1 and A.2 show the codes for saving and reading the data using Python code. We can see the complexity of the codes for these functionalities. Also, these codes do not save and read complex numbers. The code may become quite complex, if we try to add those functionalities as well.

In mathematical simulations, total number of variables are large. Also, complex numbers occur frequently, which make the python codes too lengthy and complex to store the data. Hence, it is better to use SPYDER environment to save and retrieve the data as shown in section 1.9.

Listing A.1: Save Data to File

```python
#fileSaveEx2.py
import numpy as np
from scipy.special import erfc
from itertools import zip_longest


#BPSK: theoretical error
SNR_db = np.array(np.arange(-2, 10, 1),float)
theoryBER = np.zeros(len(SNR_db),float)
for i in range(len(SNR_db)):
  theoryBER[i] = 0.5*erfc(np.sqrt(10**(SNR_db[i]/10)))

#Random numbers
rn = np.random.randn(100) #Normally Distributed Random Numbers
r = np.random.rand(100) #Uniformly Distributed Random Numbers

#save data in .csv file

#zip_longest is used
#because 'rn' has different length than SNR_db adn theory_BER.
#zip_longest checks the highest length,
#to create the number of rows in .csv file.
#if 'zip' is used instead of zip_longest,
#then lowest number of rows will be created in .csv file.
#and we will loose the remaining lines for 'rn' and 'r' in this example.

#mention all data in following line
data  = zip_longest(theoryBER, SNR_db, rn, r, fillvalue=0)
#add header(title) for each data, to recognize the column in .csv file
np.savetxt(
    'bpsk.csv',
    list(data),
    fmt = '%f, %f, %f, %f', #%f to store in float format
    header='Theory, SNR, Normal Random Numbers, Uniform Random Number'
  )
```

Listing A.2: Read Data from File

```python
#fileReadEx2.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style

```

```
#load and save column data in variables
# usecols=(1,0): load column-1 first and save to theoryBER;
# and column-0 to SNR_db
# you can define usecols in any order e.g. usecols(0,2) etc.
SNR_db, theoryBER = np.loadtxt('bpsk.csv', unpack=True,
    delimiter=',',
    skiprows=1,
    usecols=(1,0,)
  )

# usecols=(2): load column-1 save to randomNumber
#do not skip the command after 2 i.e. usecols=(2,))
NormalrandomNumber = np.loadtxt(
    'bpsk.csv',
    unpack=True,
    delimiter=',',
    skiprows=1,
    usecols=(2,)
  )

RandomNumber = np.loadtxt(
    'bpsk.csv',
    unpack=True,
    delimiter=',',
    skiprows=1,
    usecols=(3,)
  )

##Figure 1
plt.semilogy(SNR_db, theoryBER, '--mo')
plt.ylabel('BER')
plt.xlabel('SNR')
plt.title('BPSK BER Curves')
plt.legend(['Theory'], loc='upper right')
plt.grid()

#Figure 2
#plot two different figure in same window,
#i.e. NormalrandomNumber and RandomNumber
plt.figure() # create new figure window
plt.plot(NormalrandomNumber)
plt.plot(RandomNumber, '--r')
plt.ylabel('Value')
plt.xlabel('Iteration')
plt.title('Normally distributed random numbers')
plt.legend(
  ['Normally Distributed Random Numbers',
  'Uniformlly Distributed Random Numbers'],
  loc='upper right'
  )

#Display all figures
plt.show()
```

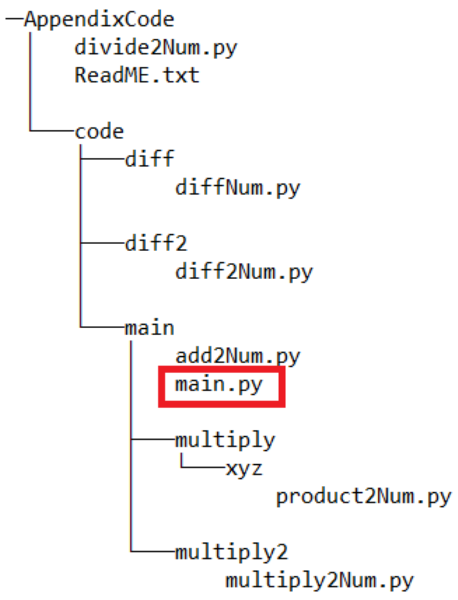
One who abandons all desires and becomes free from longing and the feeling of 'I' and 'my' attains peace.

–Krishna

Appendix B

Relative imports in python 3

In this section, relative imports are discussed for Python 3.3 and later versions. Before python 3.3, Python interpretor used to search for the files in the folders which contains the '__init__.py' file. But, in the later versions, this file is not required.



```
—AppendixCode
 |    divide2Num.py
 |    ReadME.txt
 |
 └───code
     ├───diff
     |       diffNum.py
     |
     ├───diff2
     |       diff2Num.py
     |
     └───main
         |   add2Num.py
         |   main.py
         |
         ├───multiply
         |   └───xyz
         |           product2Num.py
         |
         └───multiply2
                 multiply2Num.py
```

Figure B.1: Structure for importing files

Fig. B.1 shows the directory structure for the relative import example. Here 'main.py' is the main calling function. From that location, we want to import all the '.py' files in the figure.

Listing B.1: Add two number

```python
#add2Num.py

def add2(a,b):
    return(a+b)
```

Listing B.1 is the code which returns the sum of two numbers. All the other codes have these two lines with different function names and return values, e.g. in diff2Num.py, the return value is (a-b) etc. Further, Listing B.2 is the 'main.py' file which imports all the functions from the python files in the 'AppnedixCode' directory. Complete code is available on the website.

Listing B.2: main.py

```python
#main.py
import sys
```

```
 3 import os
 4 
 5 a=5
 6 b=2
 7 
 8 # 1.
 9 from add2Num import add2
10 print(add2(a,b)) # 5
11 
12 # 2.
13 #no need for sys.path for 'multiply folder',
14 #as it is available in the main folder
15 from multiply.xyz.product2Num import product2Num
16 print(product2Num(a,b)) # 10
17 
18 # 3.
19 #add `multiply2' folder in the path
20 sys.path.append(os.path.join(os.path.dirname(__file__), 'multiply2'))
21 from multiply2Num import product2Num
22 print(product2Num(a,b)) # 10
23 
24 # 4.
25 #add all files and folder from back folder
26 sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
27 # 'diff2' folder is available at back folder
28 # folder name 'diff2' is required as it is not explictly imported
29 from diff2.diff2Num import diff2
30 print(diff2(a,b)) # 3
31 
32 # 5.
33 #add only `diff' folder from back folder
34 sys.path.append(os.path.join(os.path.dirname(__file__), '..', 'diff'))
35 # no need to write diff.diff2Num, because folder is already imported
36 from diffNum import diff
37 print(diff(a,b)) # 3
38 
39 
40 # 6.
41 # add all files and folder from back-back folder to code
42 sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
43 # 'divide2Num.py' file is available at back-back folder
44 from divide2Num import divide2
45 print(divide2(a,b)) # 2.5
```

Explanation (Listing B.2). *In this listing, 6 cases are studied for importing the files in the main.py. Note that, we put 'import (line 9 etc.)' and 'sys.path...(line 20 etc.)' functions in the middle of the code for better understanding; ideally these lines must be at the top of the code. All the 6 cases are described below,*

Importing files which are in same folder as main.py

1. *In line 9, 'add2Num' is the python file name, whereas 'add2' is the function name defined in the file, as shown in Listing B.1. Since both main.py and add2Num.py are in the same folder, therefore it can be imported directly.*

Importing files from the folders, which exist in the same folder (or forward folders) with respect to main.py's folder

2. *Now in Fig. B.1, we can see that 'product2Num.py' is inside the 'multiply→xyz' folder, therefore it can not be import as above method. In line 15, 'from multiply.xyz.product2Num import product2Num' is used, which can be read as this, "go to multiply folder first, then inside xyz folder find production.py file and finally import 'product2Num' definition.*
3. *Line 20, adds the 'multiply2' folder in the path, therefore all the files inside this folder will be checked by the python interpretor. Hence, in line 21, 'multiply2Num' can be used instead of 'multiply2.multiply2Num'. Note that, 'multiply2.multiply2Num' is also correct for this case.*

Importing files from the folders, which exist in the previous (back) folders from main.py's folder

4. *In line 26, '..' is use to tell the interpretor to include the previous folder from the main.py location. After that, we can import files as shown in in method 2.*
5. *Line 34 imports only 'diff' folder from the previous folder. Rest of the working is same as method 3.*
6. *Finally in line 42, two '..' are used which tells the python interpreator to add the folder which is two stage backward from the main.py file. Rest of the working is same as method 1.*

▲

For man, mind is the cause of bondage and mind is the cause of liberation. Mind absorbed in sense objects is the cause of bondage, and mind detached from the sense objects is the cause of liberation.

–*Krishna*

Appendix C

Numba, Cython and Matlab Codes

C.1 Numba Code

In Numba code, only '*@autojit*' decorator is imported and added in the code 'errorCalculation.py' as shown in Listing C.2; and rest of the code is same as the original python code.

Listing C.1: Theoretical BER of BPSK system in Gaussian Noise

```python
# Numba/simBPSK.py
import time #to measure the simulation time
startTime=time.time()

import numpy as np
from scipy.special import erfc
import matplotlib.pyplot as plt

#errorCalculation: calculate the error in decision
from errorCalculation import errorCalculation

def main():
  bitLength  = 100000 #number of transmitted bit
  iterLen=30 #iterate 30 time for better average

  SNRdB = np.array(np.arange(-2, 10, 1),float)
  SNR = 10**(SNRdB/10) # SNR_db to SNR conversion
  noise = np.zeros(len(SNRdB), float)

  error =np.zeros((iterLen, len(SNRdB)), float)

  for iL in range(iterLen):
    #Generate various Noise-voltage (N0/2) Level
    for i in range (len(noise)):
      noise[i]= 1/np.sqrt(2*SNR[i]) # N0/2 = 1/sqrt(2*noisePower)

    #calculate error
    errorMatrix =np.zeros(len(SNRdB), float)
    for i in range(len(noise)):
      errorMatrix[i] = errorCalculation(bitLength, noise[i])
    #save error at each iteration
    error[iL]=errorMatrix

  #average error
  BER = error.sum(axis=0)/(iterLen*bitLength)

  #theoritical error
  theoryBER = np.zeros(len(SNRdB),float)
  for i in range(len(SNRdB)):
    theoryBER[i] = 0.5*erfc(np.sqrt(SNR[i]))

  #print simulation time
  print(time.time()-startTime) # 8.93 sec

  #plot the graph
  plt.semilogy(SNRdB, BER,'--')
  plt.semilogy(SNRdB, theoryBER, 'mo')
  plt.ylabel('BER')
  plt.xlabel('SNR')
  plt.title('BPSK BER Curves')
  plt.legend(['Simulation', 'Theory'], loc='upper right')

  plt.grid()
  plt.show()

# Standard boilerplate to call the main() function.
```

```python
if __name__ == '__main__':
  main()
```

Listing C.2: Calculate error for each iteration

```python
# Numba/serrorCalculation.py
import numpy as np
from numba import autojit

@autojit
def errorCalculation(bitLength, noiseAmp):
  #uniformly distributed sequence to generate singal
  b = np.random.uniform(-1, 1, bitLength)

  #binary signal generated from 'b'
  signal = np.zeros((bitLength),float)
  for i in range(len(b)):
    if b[i] < 0:
      signal[i]=-1
    else:
      signal[i]=1

  #Normal Distribution: Gaussian Noise with unit variance
  noise = np.random.randn(bitLength)

  #recevied signal
  rxSignal = signal + noiseAmp*noise
  # decision:
  detectedSignal = np.zeros((bitLength),float)
  for i in range(len(b)):
    if rxSignal[i] < 0:
      detectedSignal[i]=-1
    else:
      detectedSignal[i]=1
  #error matrix: save 1 if error else 0
  errorMatrix = abs((detectedSignal - signal)/2)
  error=errorMatrix.sum() #add all 1's for total error
  return error
```

C.2 Cython Code

In cython code, there are more changes in the original python code as compare to numba, as shown in Listing C.4.

Listing C.3: Theoretical BER of BPSK system in Gaussian Noise

```python
# Cython/simBPSK.py

#import .pyx files
import pyximport
pyximport.install()

import time #to measure the simulation time
startTime=time.time()

import numpy as np
from scipy.special import erfc
import matplotlib.pyplot as plt

#errorCalculation: calculate the error in decision
from errorCalculation import errorCalculation

def main():
  bitLength  = 100000 #number of transmitted bit
  iterLen=30 #iterate 30 time for better average

  SNRdB = np.array(np.arange(-2, 10, 1),float)
  SNR = 10**(SNRdB/10) # SNR_db to SNR conversion
  noise = np.zeros(len(SNRdB), float)

  error =np.zeros((iterLen, len(SNRdB)), float)

  for iL in range(iterLen):
    #Generate various Noise-voltage (N0/2) Level
    for i in range (len(noise)):
      noise[i]= 1/np.sqrt(2*SNR[i]) # N0/2 = 1/sqrt(2*noisePower)

    #calculate error
    errorMatrix =np.zeros(len(SNRdB), float)
    for i in range(len(noise)):
      errorMatrix[i] = errorCalculation(bitLength, noise[i])
    #save error at each iteration
    error[iL]=errorMatrix

  #average error
  BER = error.sum(axis=0)/(iterLen*bitLength)

  #theoritical error
  theoryBER = np.zeros(len(SNRdB),float)
  for i in range(len(SNRdB)):
    theoryBER[i] = 0.5*erfc(np.sqrt(SNR[i]))

  #print simulation time
  print(time.time()-startTime) # 7.43 sec

  #plot the graph
  plt.semilogy(SNRdB, BER,'--')
  plt.semilogy(SNRdB, theoryBER, 'mo')
  plt.ylabel('BER')
  plt.xlabel('SNR')
  plt.title('BPSK BER Curves')
  plt.legend(['Simulation', 'Theory'], loc='upper right')

  plt.grid()
  plt.show()

# Standard boilerplate to call the main() function.
if __name__ == '__main__':
  main()
```

Listing C.4: Calculate error for each iteration

```
# Cython/errorCalculation.pyx

import numpy as np
import cython

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef double errorCalculation(int bitLength, double noiseAmp):
  #uniformly distributed sequence to generate singal
  cdef:
      int  i
      double error =0.0
      double[:] b = np.random.uniform(-1, 1, bitLength)
      double[:] noise = np.random.randn(bitLength)
      double[:] signal = np.zeros(bitLength)
      double[:] rxSignal = np.zeros(bitLength)
      double[:] detectedSignal = np.zeros(bitLength)
      double[:] errorMatrix = np.zeros(bitLength)

  for i in range(bitLength):
    if b[i] < 0:
      signal[i]=-1
    else:
      signal[i]=1

  #recevied signal
  for i in range(bitLength):
      rxSignal[i] = signal[i] + noiseAmp*noise[i]
  #decision
  for i in range(bitLength):
    if rxSignal[i] < 0:
      detectedSignal[i]=-1
    else:
      detectedSignal[i]=1
  #store 1 in errorMatrix if error else 0
  for i in range(bitLength):
      errorMatrix[i] = abs((detectedSignal[i] - signal[i])/2)
  #add all 1's to find total error
  error = np.sum(errorMatrix)
  return error
```

C.3 Matlab Code

BER performance is evaluated using Matlab codes in Listing C.5 and C.6.

Listing C.5: Theoretical BER of BPSK system in Gaussian Noise

```
%SimBPSK.m
```

```matlab
clear all
close all
clc

tic
bitLength  = 100000; %  100000; %number of transmitted bit
iterLen=30; %iterate 30 time for better average

SNRdB = -2:1:10;
SNR = 10.^(SNRdB/10); % SNR_db to SNR conversion
noise = zeros(1,length(SNRdB));

error =zeros(iterLen, length(SNRdB));

for iL = 1:iterLen
  %Generate various Noise-voltage (N0/2) Level
  for i =1:length(noise)
    noise(1,i) = 1/sqrt(2*SNR(i)); % N0/2 = 1/sqrt(2*noisePower)
  end

  %calculate error
  errorMatrix =zeros(1, length(SNRdB));
  for i=1:length(noise)
    errorMatrix(i) = ErrorCalculation(bitLength, noise(i));
  end
  %save error at each iteration
  error(iL,:)=errorMatrix;
end

%average error
BER = sum(error)/(iterLen*bitLength);

%theoritical error
theoryBER = zeros(1, length(SNRdB));
for i= 1:length(SNRdB)
  theoryBER(i) = 0.5*erfc(sqrt(SNR(i)));
end

toc1 = toc;
%print simulation time
fprintf('elapse time: %0.5f sec', toc1) %    sec

%plot the graph
semilogy(SNRdB, BER,'--r')
hold on
semilogy(SNRdB, theoryBER, '*')
legend('Simulation', 'Theory')
```

Listing C.6: Calculate error for each iteration

```matlab
% ErrorCalculation.m
function error = ErrorCalculation(bitLength, noiseAmp)
  % uniformly distributed sequence to generate singal
  ip = rand(1,bitLength)>0.5; % generating 0,1 with equal probability
  b = 2*ip-1; % BPSK modulation : convert 0 to -1; 1 to 1

  %binary signal generated from 'b'
  signal = zeros(1, bitLength);
  for i=1:length(b)
    if b(i) < 0
      signal(i)=-1;
    else
      signal(i)=1;
    end
  end
  %Normal Distribution: Gaussian Noise with unit variance
  noise = randn(1, bitLength);

  %recevied signal
  rxSignal = signal + noiseAmp*noise;

  % decision:
  detectedSignal = zeros(1, (bitLength));
  for i = 1:length(b)
    if rxSignal(i) < 0
      detectedSignal(i)=-1;
    else
      detectedSignal(i)=1;
    end
  end

  %error matrix: save 1 if error else 0
  errorMatrix = abs((detectedSignal - signal)/2);
  error=sum(errorMatrix); %add all 1's for total error
```