VALENCIA COLLEGE

Python:

Programming and Applications

Masood Ejaz

Department of Electrical and Computer Engineering Technology

Valencia College

Contents

6.5. Class and Static Methods

7. Matrix Algebra
   7.1. Arrays and Matrices
   7.2. Special Matrices/Arrays
   7.3. Operations on Matrices
   7.4. User Inputs

8. Plots
   8.1. Single Plots
   8.2. Multiple Plots
   8.3. Other Plotting Functions

9. Symbolic Mathematics
   9.1. Algebraic Equations
   9.2. Limits
   9.3. Derivatives
   9.4. Integral
   9.5. Ordinary Differential Equations
   9.6. Equation Evaluation

10. Numerical Methods
    10.1. Interpolation
    10.2. Curve Fitting
    10.3. Numerical Differentiation
    10.4. Numerical Integration

11. Graphical User Interface (GUI)
    11.1. Widgets
    11.2. Geometry Management
    11.3. Callback Functions
    11.4. Games and Applications

Chapter 1

Basic Functionality

Python is a higher-level programming language which is widely used in academia and industry. It is the highest-ranked language in popularity and usage for 2017 & 2018 by IEEE [1]. Python holds an open-source license which makes the language free to use. Python syntax is more interactive and easier to understand as compared to other popular languages like C/C++ and Java. Many features of Python make its syntax closer to that of MATLAB. Like C++, Python also supports *Object-Oriented Programming (OOP)*

Python was created by Guido van Rossum and first released in 1991. When he began implementing Python, Guido van Rossum was also reading the published scripts from "Monty Python's Flying Circus", a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python [2]. Hence, the name has nothing to do with the snake, *python*.

1.1 Installation

Latest version of Python can be downloaded and installed for free from *Python Software Foundation* website (https://www.python.org/). At the time of writing this text, the latest version of Python is 3.7.0. Once it is downloaded and installed, click on *IDLE (Python's Integrated Development and Learning Environment)* to open *Python Shell*. Python shell is similar to MATLAB command window, where simple Python commands and simple calculations may be carried out. To write a Python program, open a new file from *File* menu, which will open Python editor. One can write python codes and save them with '.py' extension in the editor. To run any program, go to *Run* menu and choose *Run Module*, as shown in *figure 1.1*

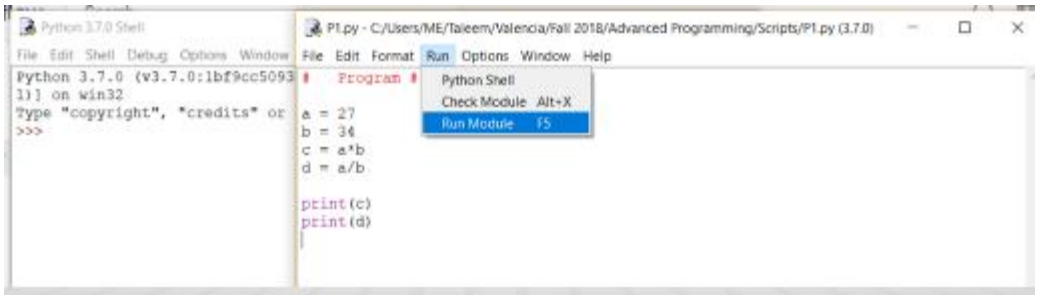


Figure 1.1: Running a program from Python editor

Environment of IDLE is basic and not very interactive. There are other environments written in Python that give advanced editing and interactive execution. One of these environments is *Spyder* (The Scientific Python Development Environment), which is a powerful scientific environment written in Python for scientists, engineers, and data analysts. *Spyder* can be downloaded free from its website (https://www.spyder-ide.org/). Both IDLE and Spyder are used in this text although students are encouraged to use Spyder.

1.2 Modules

Like MATLAB has toolboxes for different categories where different functions under that category are located, Python has different modules. If you are using a specific function from a module, first that module needs to *imported*. One of the most commonly used module is *math*, where most of the mathematical functions are located. There are two ways to use a function from any module:

Method 1: Import the complete module first by using syntax *import module* and then use any function from the module using the format: *module.function()*

For example, *cosine* function from *math* module can be used as follows:

```
>>> import math
>>> math.cos(2)
-0.4161468365471424
```

Method 2: Import only specific functions from the module that are required using format: *from module import function*. Then the imported functions can be used with their names without adding module name with them

For example, *cosine and sins* functions from *math* module can be used as follows:

```
>>> from math import sin, cos
>>> cos(3.4)
-0.9667981925794611
>>> sin(1.2)
0.9320390859672263
>>> cos(sin(3.2))
0.9982967134401437
```

Observe that by default trigonometric functions have their argument in radians. If argument is given in degrees, make sure to convert it into radians before using trigonometric functions.

List of all functions from any module can be checked by importing the module and then using command *dir(module)*. All functions from *math* module are shown in *figure 1.2*

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin
', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e'
, 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgam
ma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder
', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Figure 1.2: All functions from math module

Note that the first five entities under *dir(math)* are *methods* under any class that can be defined from *math* module. Classes and methods are part of object-oriented programming and will be discussed later in this text.

To check the help of any module and its functions, *help(module)* command can be used. *Figure 1.3* shows part of the help for the *math* module

```
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

    atan2(y, x, /)
        Return the arc tangent (measured in radians) of y/x.

        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.
```

Figure 1.3: Help for the 'math' module

Note that for complex numbers, there is a module in Python called *cmath*. Any mathematical function that is expected to produce a complex quantity must be imported from *cmath* instead of *math* module.

There are certain built-in functions that are always available and no module needs to be imported for them. A list of these functions is shown in *figure 1.4*

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

Figure 1.4: Built-in functions that are always available [3]

1.3 Simple Mathematics

Python shell can be used to perform simple mathematics. The standard symbols that are used to carry out basic operations are shown in *Table 1.1*

Table 1.1 – Arithmetic Operations

Operation	*Symbol*	*Explanation*
Addition	+	
Subtraction	-	
Multiplication	*	
Division	/	
Floor division	//	Round off the result to the previous integer
Modulo	%	Remainder after division of one number by another
Power	**	

1.4 Variables

Variables are commonly used to assign values and hold the result of some calculation. Variables are commonly used in expressions and equations. In Python, variable name can be of any length and it can contain letters, digits, and underscore (_), the only special character allowed. Furthermore, name of the variable cannot start with a digit; it can start with a letter or an underscore. Variables are case sensitive; hence, ‘m’ and ‘M’ will be considered as two different variables.

1.5 Output Function

To show or print value of some variable or result of some calculation, *print()* function is used. *print* is a versatile function that not only displays value or result but it can also be used to embed the result in a sentence, much easier than how it is done in MATLAB.

Example 1.1

Write a python program with two values assigned to two variables. Calculate the product of the two variables and divide one variable by the other. Assign the result of each operation to two different variables. Print out the result for each operation and also output each result embedded into a message in such a way that it can clearly explain it.

Code

```
#  Example 1.1

a = 27
b = 34
c = a*b
d = a/b

print(c)
print(d)

print('The result of the expression c = a*b for a = ', a, 'and b = ', b, 'is ', c)
print('The result of the expression d = a/b for a = ', a, 'and b = ', b, 'is ' ,d)
```

Output

```
918
0.7941176470588235
The result of the expression c = a*b for a =  27 and b =  34 is  918
The result of the expression d = a/b for a =  27 and b =  34 is  0.7941176470588235
>>>
```

Observations

- In Python, comments are written using a hashtag (#)
- When embedding a result in a sentence using *print* function, sentence can be written either within apostrophes or inverted commas; both work.
- Results that need to be embedded are written outside the apostrophes or inverted commas separated by commas

Example 1.2

Evaluate the following expression for x = 2.5 radians and y = 3 degrees:

$$f(x,y) = \frac{2\sin(x\pi)\cos(3y)}{4xe^{-y}}$$

Code

```
#   Example 1.2

x = 2.5     #   x is in radians
y = 3       #   y is in degrees

from math import sin, cos, pi, exp       #   importing pi, exponential and trig functions from math module

f = (2*sin(pi*x)*cos(3*y*pi/180))/(4*x*exp(-y))

print("The result for the expression (2sin(pi*x)cos(3y))/(4x*exp(-y)) for x = ", x, "radians and y = ", y, "degrees is ", f)
```

Output

```
The result for the expression (2sin(pi*x)cos(3y))/(4x*exp(-y)) for x =  2.5 radians
and y =  3 degrees is  3.967650126725119
>>>
```

To format a number to show specific precision, *format()* function can be used within *print()* function as shown in the following example

Example 1.3

Repeat *Example 1.2* with x printed with three decimal places and result printed with 4 decimal places

Code:

```
#   Example 1.2

x = 2.5     #   x is in radians
y = 3       #   y is in degrees

from math import sin, cos, pi, exp       #   importing pi, exponential and trig functions from math module

f = (2*sin(pi*x)*cos(3*y*pi/180))/(4*x*exp(-y))

print("The result for the expression (2sin(pi*x)cos(3y))/(4x*exp(-y)) for x = {:0.3f} radians and y = {:d} degrees is {:0.4f}".format(x,y,f))
```

Output

```
The result for the expression (2sin(pi*x)cos(3y))/(4x*exp(-y)) for x = 2.500 radians
and y = 3 degrees is 3.9677
```

1.6 Basic Data Types

There are three basic data types in Python; *Numbers*, *Strings*, and *Boolean*. There are also important collective data types or arrays including *Lists, Dictionaries, Tuples, and Sets* that will be discussed in a later chapter.

Numbers: There are three categories of numbers; *integer*, *floating point* or *decimal numbers*, and *complex*. All three categories are self-explanatory. *type()* function is used to determine the class of data types.

```
>>> q = 2
>>> w = 4.5
>>> e = 3+6.7j
>>> type(q)
<class 'int'>
>>> type(w)
<class 'float'>
>>> type(e)
<class 'complex'>
>>> |
```

It is possible to change the class of different numbers and to make a complex number out of two numbers. Functions that are used to do these jobs are *int()*, *float()*, and *complex()*

```
>>> float(q)
2.0
>>> int(w)
4
>>> complex(q,w)
(2+4.5j)
>>> |
```

Strings: A set of characters is called a *string*. It is defined by set of characters between apostrophes or inverted commas.

```
>>> qw = "Masood"
>>> er = 'Ejaz'
>>> type(qw)
<class 'str'>
>>> type(er)
<class 'str'>
```

Length of a string is represented by its number of characters and can be determined by the function *len()*

```
>>> len(qw)
6
>>> len(er)
4
```

Two strings can be concatenated either using '+' symbol or using *print* function.

```
>>> name = qw+er
>>> print(name)
MasoodEjaz
>>> print(qw, er)
Masood Ejaz
```

To add a space between two strings when using '+' to concatenate, simply add a space using " " between the two strings

```
>>> qw = "Masood"
>>> er = "Ejaz"
>>> print(qw + " " + er)
Masood Ejaz
```

Any character or set of characters from a string can be fetched. To fetch a character or set of characters, string name followed by the *index* number of the character (or characters) in square brackets is used. Note that indexing starts at '0'; hence, the index of the first character in a string is '0'

```
>>> #   First five characters of string qw
>>> qw[:5]
'Masoo'
>>> #   Last five characters of string qw
>>> qw[-5:]
'asood'
>>> #   Elements 1 through 4 of string qw
>>> qw[1:5]
'asoo'
```

Booleans: Boolean class returns two values; *True* or *False*. Any variable that is taking upon result of any comparative or Boolean operation is a Boolean variable. *bool()* function can be used to compare two values and assign result to a Boolean variable.

```
>>> a = bool(1>2)
>>> type(a)
<class 'bool'>
>>> print(a)
False
```

Table 1.2 shows relational operators that return a Boolean value

Table 1.2: Relational Operators

Operator	Description
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Logical operators also yield Boolean results. The three basic logical operators are *and*, *or*, and *not*.

```
>>> b = True
>>> a = False
>>> c = a and b
>>> print(c)
False
>>> d = a or b
>>> print(d)
True
>>> not a
True
```

Some special operators that also produce Boolean results are shown in *Table 1.3*

Table 1.3: Special Operators

Operator	Explanation	Example
is	True if the values are same	x is y
is not	True if values are different	x is not y
in	True if value is in a variable	"w" in string
in not	True if value is not in a variable	"w" in not string

```
>>> qw
'Masood'
>>> 'a' in qw
True
>>> er
'Ejaz'
>>> qw is er
False
>>> 'f' not in er
True
```

1.7 Input Function

input() is used to get an input from user as a string. If an integer or floating point input is required, the string will need to be converted into the desired data type with appropriate functions, *int()* or *float()*

Example 1.4

Ask user to input two numbers for which the following expression will be evaluated. Print out your result in a proper sentence that explains expression, inputs and result.

$$f(x,y) = \frac{2\cos(x)\sin(y)}{4x-3y}$$

Code

```
#   Example 1.4

x = float(input("Enter a number x: "))  #   float() is going to convert the input from a string to a floating point
y = float(input("Enter a number y: "))

from math import sin, cos

f = (2*cos(x)*sin(y))/(4*x - 3*y)

print("The expression (2cosxsin(y))/(4x-3y) is calculated for x = ",x,"and y = ",y,"and result is ",f)
```

Output

```
Enter a number x: 1.2
Enter a number y: 3.4
The expression (2cosxsin(y))/(4x-3y) is calculated for x =  1.2 and y =  3.4 and result is
 0.03429529626182856
>>>
```

Exercises

1.1 Write a Python program to calculate the result of the following expression. Print out your result in a proper sentence that explains expression, inputs and result

$$f(x,y)=\frac{2xy+x^{y}}{3x-4y};x=1.3;y=0.5$$

1.2 Write a Python program to calculate the result of the following expression. Print out your result in a proper sentence that explains expression, inputs and result. Both *x* and *y* are in radians.

$$f(x,y)=\frac{2\cos(5\pi x)\sin(6y^{x})}{2\sqrt{\cos(\sin(3xy))}+4\ln(\cos(\mathrm{y}))};x=2.4,y=0.4$$

1.3 Calculate the roots of the following quadratic equation. Print out your result in a proper sentence that explains quadratic equation and results.

$$x^{2}+2x+3=0$$

1.4 Given that string 1 is "The course number is" and string 2 is "CET3464C, CET4370C", create a new string and print it out with the following characters: "The course number is CET4370C"

1.5 Ask user to enter coefficients of a quadratic equation, *A*, ***B,*** and *C*. Calculate the roots of the quadratic equation $Ax^{2}+Bx+C=0$ and print out your results in a proper sentence that explains correct quadratic equation and its results.

1.6 Write a program to calculate monthly mortgage payment from three parameters; length of mortgage in years, annual interest rate (in percentage), and loan amount. The formula to calculate monthly mortgage payment is as follows:

$$P = \frac{RL}{1-(1+R)^{-N}}$$

where R is the monthly interest rate in decimal, which is calculated as annual interest rate (in decimal) divided by 12 (number of months in a year), L is the initial loan amount, and N is number of months for the mortgage.

Ask user to enter R, L, and N from which calculate and print out the monthly payments in a proper sentence.

Chapter 2

Control Statements

Control statements or *transfer of control* statements are ones that produce a jump in the sequence of execution of statements based on some condition. These statements are present in all programming languages; lower- or higher-level. The most common transfer of control statements are *for*, *while*, and *if-else*. These and some more statements will be discussed in this chapter.

2.1 For Loops

For loops are used when some statements or operations need to be executed for a specific number of times or for specific values. The syntax of a *for* loop in Python is as follows:

for variable in sequence:

statements to be repeated

where *variable* is a variable or argument to hold the iteration count or specific values for the *for* loop and *sequence* is an integer value that represent either iteration from zero to *N*-1 or any specific integers represented as a *list*

When *for* argument is holding an iteration value from zero to *N*-1, the syntax for the *for* statement is as follows:

for k in range(N)

where *k* is the argument or variable of the *for* loop that will change from zero to *N*-1 executing everything inside the *for* loop for *N* times. Note that all statements that are indented under the *for* statement are considered to be inside the *for* loop and are executed the number of times *for* loop is going to run.

Example 2.1

Evaluate a quadratic equation $x^2 + 3x + 16$ for $x = 0$ to 10

Code:

```
#   Example 2.1
#   This is an example to use For loop in Python
#
#   Using for loop, evaluate a quadratic equation, x^2 + 3x +16, for x = 0 to 10 (only integers)

for x in range(11):
    print(x**2+3*x+16)
```

Output:

```
16
20
26
34
44
56
70
86
104
124
146
```

If instead of *for* argument assuming the iteration values of zero to *N*-1, a generic iteration sequence is required with *starting value*, *step-size*, and *end value*, it can be done using the following syntax:

for k in range(start value, end value + 1, step-size)

Example 2.2

Evaluate a quadratic equation $x^2 + 3x + 16$ for x = -10 to 10 with a step-size of 2

Code:

```
#   Example 2.2
#   This is an example to use For loop in Python
#
#   Using for loop, evaluate a quadratic equation, x^2 + 3x +16, for x = -10 to 10 with an increment of 2

for x in range(-10,11,2):
    print(x**2+3*x+16)
```

Output:

```
86
56
34
20
14
16
26
44
70
104
146
```

Since *range()*can only take integers, if an expression needs to be evaluated for a sequence with floating points, first a variable needs to be mapped from the argument values to the intended values and then this variable is used to evaluate the expression for the required values. This is one way to carry out this operation using *for* loop. Same operation will be done differently in later chapters.

Example 2.3

Evaluate a quadratic equation $x^2 + 3x + 16$ for x = -1 to 1 with a step-size of 0.2

Code:

```
#   Example 2.3
#   This is an example to use For loop in Python
#
#   Using for loop, evaluate a quadratic equation, x^2 + 3x +16, for x = -1 to 1 with an increment of 0.2

a = 2/0.2 + 1 #   Total number of points, which is abs(final value-initial value)/increment + 1
for x in range(int(a)): #   int(a) will change the floating value of 'a' into integers since for loop arguments can only be integers
    y = x*0.2 - 1   #   y is a variable that is mapped from the loop iteration value to the required sequence
    f = y**2+3*y+16
    print(f)
```

Output:

```
14.0
14.24
14.56
14.96
15.44
16.0
16.64
17.36
18.16
19.04
20.0
```

When loop has to run for argument values that do not have any sequence, *list* is used. *list* in Python represents an array of values. It will be discussed in detail in a later chapter. Note that list values can be integers or floating points.

Example 2.4

Evaluate a quadratic equation $x^2 + 3x + 16$ for x = [-2, 4.5, 6, 8.1, -10.05, -12, 20]

Code:

```
#   Example 2.4
#   This is an example to use For loop in Python
#
#   Using for loop, evaluate a quadratic equation, x^2 + 3x +16
#
#   This is the first example where "list" will be used

x_values = [-2, 4.5, 6, 8.1, -10.05, -12, 20]  #   list is represented by square-brackets
for x in x_values:
    f = x**2+3*x+16
    print(f)
```

Output:

```
14
49.75
70
105.91
86.8525
124
476
```

Nested For Loops

Nested for loops are statements where there is a *for* loop inside another *for* loop. *Nested for loops* are used when a set of operations need to be executed when values for more than one argument are changing.

Example 2.5

Evaluate an equation, $f(x, y) = 2y\sin(x) + 4\cos(y)$ for x = -1 to 2 with an increment of 0.3 and y = [3, 8, 10, -12]

Code:

```
#   Example 2.5
#   This is an example of nested for loop
#
#   Evaluate an equation, f(x,y) = 2*y*sin(x) + 4*cos(y) for  x = -1 to 2 with increment of 0.3 and y = [3, 8, 10, -12]
#
y_values = [3, 8, 10, -12]
x_range = 3/0.3 + 1 #   Total number of points for x

from math import sin, cos

print("\n\t  x \ty \tf")  #   \n is for new line and \t is for tab
for y in y_values:
    for x in range(int(x_range)):
        x_values = -1 + x*0.3    #   x_values go from -1 to 2 with an increment of 0.3 when x goes from 0 to 10
        f = 2*y*sin(x_values) + 4*cos(y)
        print("\n\t{:.2f} \t{} \t{:.2f}".format(x_values,y,f))
```

Output:

Partial output is shown here

x	y	f
-1.00	3	-9.01
-0.70	3	-7.83
-0.40	3	-6.30
-0.10	3	-4.56
0.20	3	-2.77
0.50	3	-1.08
0.80	3	0.34
1.10	3	1.39
1.40	3	1.95

Series Summation

For loops are frequently used to yield sum of a sequence or series.

Example 2.6

Write a program that calculates the following series, which calculates the value of *ln*(2) [4]

$$\ln(2) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots$$

Code:

```
#   Example 2_6
#
#   This program calculates the natural-log of 2 through the infinite series 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + ......
#

N = int(input("Up to which number the summation of the infinite series for ln(2) is required? "))
ln2 = 0;    #   A variable that will be updated to hold the value of ln2

for n in range (1, N+1): #   This will run the loop for n = 1 to N
    a = (-1)**(n+1)/n
    ln2+=a #    This statement will perform the following job: ln2 = ln2 + a;

print("Adding the infinite series for ln(2) up to {} terms gives {}".format(n,ln2 ))
```

Output:

```
Up to which number the summation of the infinite series for ln(2) is required? 1000
Adding the infinite series for ln(2) up to 1000 terms gives 0.6926474305598223
```

Observe, in *Example 2.6*, the second line inside *for* loop reads: *ln2+=a*. When a mathematical operation is performed to update the value of a variable that is calculated through the same operation, the corresponding code is generally written as:

A = A (operation) B

For example, if value of variable *A* is updated by adding value of variable *B* in the existing value of *A*, the corresponding code will be, *A* = *A* + *B*. In Python, this may be written as *A*+=*B*, although writing *A* = *A* + *B* will also work.

2.2 While Loops

A *For* loop is used when a set of instructions is executed for a known number of iterations. In contrast, the number of iterations for a *while* loop is based on the validity of some condition; as long as the condition is valid, statements written under the *while* loop will keep on executing. Hence, the number of iterations for a *while* loop is unknown. This is perhaps the most important difference between *for* and *while* loops.

In Python, the syntax for a *while* loop is as follows:

while (condition):

statements

Example 2.7

Ask user to enter a number. Add the number back into it to produce another number. Keep repeating this until the final result becomes greater or equal to 1000. Print out the final result and number of iterations it took to get to that result.

Code:

```
#   Example 2_7
#
x = float(input('Enter a number: '))
n = 0; #    number of iterations
while (x <= 1000):
    x+=x #  x = x+x; adding the number back into it and producing the result which will be saved in the same variable
    n+=1 #  updating number of iterations

print('Final number is {:0.2f} and it took {} iterations to yield this result'.format(x,n))
```

Output:

```
Enter a number: 12.4
Final number is 1587.20 and it took 7 iterations to yield this result
```

2.3 If-Else

If-else routine is used where some statements need to be executed if a condition is met else some other statements need to be executed. Python syntax for *if-else* routine is:

if (condition to be checked):

statements

else:

statements

Example 2.8

Ask user to enter a number *x*. If number is zero or positive, evaluate the following expression:

$$f(x) = 2\cos(x)\sin(x) + \sqrt{x\,|\cos(\sin(x))|}$$

If number is negative, evaluate the following expression:

$$f(x) = 2\cos(2x)\sin(3x) + \sqrt{|\cos(\sin(5x))|}$$

Print out your result with the information about the evaluated expression

Code:

```
#   Example 2_8

from math import sin, cos, sqrt
x = float(input('Enter a number to evaluate one of the two expressions: '))

if (x >= 0):
    f = 2*cos(x)*sin(x)+sqrt(x*abs(cos(sin(x))))
    print("Since input number is {:0.2f}, expression 2cos(x)sin(x) + sqrt(x|cos(sin(x))|) is evaluated and the result is {:0.2f}".format(x,f))
else:
    f = 2*cos(2*x)*sin(3*x)+sqrt(abs(cos(sin(5*x))))
    print("Since input number is {:0.2f}, expression 2cos(2x)sin(3x) + sqrt(|cos(sin(5x))|) is evaluated and the result is {:0.2f}".format(x,f))
```

Output:

```
Enter a number to evaluate one of the two expressions: 3.76
Since input number is 3.76, expression 2cos(x)sin(x) + sqrt(x|cos(sin(x))|) is evaluated and the result is 2.72
```

```
Enter a number to evaluate one of the two expressions: -0.09
Since input number is -0.09, expression 2cos(2x)sin(3x) + sqrt(|cos(sin(5x))|) is evaluated and the result is 0.43
```

If-else-if:

When there are multiple sets of statements to be executed for different conditions, *if-else-if* structure is used. In Python, the syntax is as follows:

if (condition 1):

statements

elif (condition 2):

statements

elif (condition 3):

statements

: : :

: : :

else:

statements

Example 2.9

Write a program that asks user to enter a number. If the number is negative, evaluate:

$$f(x) = 2\cos(x)\sin(x) + \sqrt{|x\cos(\sin(x))|}$$

If it is between zero and less than 100, evaluate:

$$f(x) = 2\cos(2x)\sin(3x) + \sqrt{|\cos(\sin(5x))|}$$

and, if it is equal or greater than 100, evaluate:

$$f(x) = 2\cos(\sin(2x))\sin(\cos(3x)) + \sqrt{|\sin(\cos(5x))|}$$

Print out your result with proper explanation of the expression that is evaluated

Code:

```
#   Example 2_9
#   If-Else-If Routine

from math import sin, cos, sqrt
x = float(input('Enter a number to evaluate one of the three expressions: '))

if (x < 0):
    f = 2*cos(x)*sin(x)+sqrt(abs(x*cos(sin(x))))
    print("Since input number is {:0.2f}, which is less than zero, expression 2cos(x)sin(x) + sqrt(|xcos(sin(x))|) is evaluated and the result is {:0.2f}".format(x,f))
elif (x >= 0) and (x < 100):
    f = 2*cos(2*x)*sin(3*x)+sqrt(abs(cos(sin(5*x))))
    print("Since input number is {:0.2f}, which is between zero and 100, expression 2cos(2x)sin(3x) + sqrt(|cos(sin(5x))|) is evaluated and the result is {:0.2f}".format
else:
    f = 2*cos(sin(2*x))*sin(cos(3*x))+sqrt(abs(sin(cos(5*x))))
    print("Since input number is {:0.2f}, which is greater than or equal to 100, expression 2cos(sin(2x))sin(cos(3x)) + sqrt(|sin(cos(5x))|) is evaluated and the result
```

Output:

```
Enter a number to evaluate one of the three expressions: -12.3
Since input number is -12.30, which is less than zero, expression 2cos(x)sin(x) + sqrt(|xcos(sin(x))|) is evaluated and the result is 3.95

In [4]: runfile('C:/Users/ME/Taleem/Valencia/Fall 2018/Advanced Programming/Scripts/Example2_9.py', wdir='C:/Users/ME/Taleem/Valencia/Fall 2018/Advanced Programming/Scripts')

Enter a number to evaluate one of the three expressions: 14.6
Since input number is 14.60, which is between zero and 100, expression 2cos(2x)sin(3x) + sqrt(|cos(sin(5x))|) is evaluated and the result is 1.10

In [5]: runfile('C:/Users/ME/Taleem/Valencia/Fall 2018/Advanced Programming/Scripts/Example2_9.py', wdir='C:/Users/ME/Taleem/Valencia/Fall 2018/Advanced Programming/Scripts')

Enter a number to evaluate one of the three expressions: 200
Since input number is 200.00, which is greater than or equal to 100, expression 2cos(sin(2x))sin(cos(3x)) + sqrt(|sin(cos(5x))|) is evaluated and the result is -0.38
```

2.4 Continue-Break

Continue-Break statements are used with *for* and *while* loops to either continue with a set of statements or break out of the loop at any point.

Example 2.10

Write a program that checks number from a list between 0 and 9 and print out the number outside the list

Code:

```
#   Example 2.10
#   Example of continue statement

a = [2, 5, 6, 8] #  a list with numbers between 1 and 10
for x in range(10):
    if x in a:
        continue
    print(x)
```

Output:

```
0
1
3
4
7
9
```

Example 2.11

Ask user to enter a number. Use *while* loop to start adding 1 to the number entered by the user recursively until it will hit 100 plus the number entered by the user. Print the final number.

Code:

```
#   Example 2.11
#   An example to show 'break' statement

a = float(input("Enter a number: "))
x = 0;
while True:  #   This will always run unless you break it
    x+=a
    if (x == a+100):
        break
print("Final Number: {:0.2f}".format(x))
```

Output:

```
Enter a number: 2
Final Number: 102.00
```

Exercises

2.1 Evaluate the value of the following equation for x = 4 to 6 with a step-size of 0.2. Print your result.

$$f(x) = 2e^{x}\sin(\cos(x)) + 3$$

2.2 Evaluate the value of the following equation for x = *-1* to *2* with a step-size of 0.1, and y = [2, 5, 8, 9]. Print out your results in a table as shown below.

$$f(x, y) = 6y\sin(x) + \sqrt{|\cos(x)|^{y}}$$

Note: you have to evaluate *f(x,y)* for each combination of the (*x*,*y*) pair. Under *print* function, *n* is used to go to the next line and *t* is used to produce a tab.

Sample Output:

```
  x     y   f(x,y)
-1.0    2   -9.56
-1.0    5   -25.03
-1.0    8   -40.31
-1.0    9   -45.38
-0.8    2   -7.91
-0.8    5   -21.12
-0.8    8   -34.20
-0.8    9   -38.54
-0.6    2   -5.95
```

2.3 *Multiplication Table:* Write a program to print out the multiplication table of an integer. Ask user for two inputs; a number (*integer)* for which multiplication table is produced and another number (*integer)* up to which it will be calculated. Once you get both the values, calculate and print out the multiplication table.

Sample Output

```
Enter a number for which multiplication table is required: 3

Enter a number up to which multiplication table is required: 14
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
3 x 11 = 33
3 x 12 = 36
3 x 13 = 39
3 x 14 = 42
```

2.4 *Fourier Series:* According to *Fourier*, any periodic function is a combination of three quantities: average value of the function, a sinusoid with the same frequency as the original periodic function, called *fundamental* component, and an infinite series of sinusoids, each with frequency to be a multiple of the fundamental frequency, called *harmonics*. This is called *Fourier series* of the periodic function.

Fourier series of a sawtooth waveform, as shown in *figure 2.1*, may be calculated from the following equation:

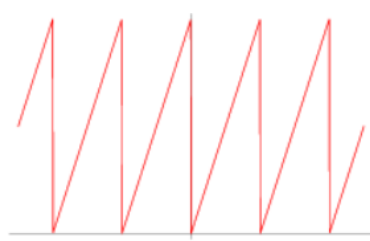


Figure 2.1: Sawtooth waveform

$$v(t) = \frac{V}{2} - \sum_{n=1,2,3,4,,\ldots} \frac{V}{n\pi} \sin(2\pi n f t)$$

where V is the peak value of the waveform, n is the harmonic number (n = 1 is the fundamental component), f is the frequency of the waveform (in Hertz), and t is the time range over which the Fourier series is evaluated (range of x-axis)

Write a program to calculate the value of the Fourier series of the sawtooth waveform at a single value of time. Ask user to enter *V*, *n*, *f*, and *t*. Evaluate the series up to *n*-th harmonic at time *t*.

Sample Output

```
Enter the amplitude of the waveform: 10

Enter the last harmonic of the waveform up to which Fourier series is
required: 23

Enter the frequency of the waveform: 15

Enter the time instant at which the value of the Fourier series is
required: 0.43

The Fourier series value for the sawtooth waveform with amplitude 10.0
at time 0.43 up to harmonic number 23 is 4.439
```

2.5 Randomly generate an integer between 1 and 10. Ask user to guess the number. Keep asking until user guesses the correct number. Print out the number of iterations it took to guess the correct number.

(Note: Use *randint* from *random* module to generate random integers)

2.6 Write a program which picks a pair of dice. If the sum of the rolled numbers is 10, print "You WON!", else print "You lost! Better luck next time!"

2.7 Write a program that requests the age of the user. If the entered age is less than 5, print "No school yet". If it is between 6 and 10, print "Elementary school". If it is between 11 and 13, print "Middle school". If it is between 14 and 17, print "High school". If it is between 18 and 22, print "University time". If entered age is greater than 22, print "You are ready for life!"

2.8 Write a program that guides the user to guess a number between 1 and 100 picked randomly by the computer. The program should guide the user like "Go Down" or "Go Up" based on user's guess. Once user successfully guesses the number, show the number of trials it took by the user.

2.9 Evaluate the value of the following equation for x = [2.3, 8.9, -19.8, 6.7, 3], and y = [2, 5, 8, 9]

$$f(x, y) = 6y\sin(x) + \sqrt{|\cos(x)|^y}$$

Chapter 3

Collective Data Types

Some of the core data types in Python are collective data types or arrays. Some of these include *strings*, *lists*, *dictionaries*, and *tuples*. *Strings* and *lists* have already been introduced earlier but they will be discussed in more detail in this chapter. *Dictionaries* and *tuples* will be introduced in this chapter.

3.1 Strings

Strings and some of the string related functions were discussed in *chapter 1*. Just to reiterate, a string is a set of characters defined between double quotes, “ ”, or single quotes, ‘ ‘. Length of a string can be found using *len()* function and it is the number of characters in a string. Any specific character or number of characters can be accessed using the index of characters in the square brackets, *string_name[]*. Remember that index number starts at zero.

Some of the methods that can be carried out on a string are given in *Table 3.1* [4], [5], [6], [7]. The syntax to use these methods on any string is *string.method()*

Table 3.1 – Common functions used with strings

Method	*Explanation*
split()	Splits the words in a string to create a list
index()	Finds the index of a character written inside the parentheses from a string. If the character appears multiple times in the string, it will only find the index of its first instance.
rindex()	Same as index() but finds the index of the last instance of a character
count()	Counts the number of instances a character written inside the parentheses appears in a string
upper()	Converts all letters to upper-case from a string
lower()	Converts all letters to lower-case from a string
find()	Find the lowest index of the substring given inside the parentheses
rfind()	Find the highest index of the substring given inside the parentheses
strip()	Leading characters given inside parentheses are removed from a string
lstrip()	Similar to strip
rstrip()	Trailing characters given inside parentheses are removed from a string
swapcase()	Swap cases of letters within a string

ljust(width,optional char)	Left justify a string with the width given by 'width'. If length of the string is less than the width, rest will be filled out by the characters given by 'char'. If 'char' is not given, spaces will be used.
rjust(width,optional char)	Right justify a string with the width given by 'width'. If length of the string is less than the width, rest will be filled out by the characters given by 'char'
center(width, optional char)	Center justify a string with the width given by 'width'. If length of the string is less than the width, rest will be filled out by the characters given by 'char'
zfill(width)	Pad a string with zeros to complete length of the string as given by 'width'
replace(old, new, optional number)	Replaces the 'old' substring by the 'new' string. Optional number is the number of old substrings replaced by the new string
capitalize()	First character of each word will be capitalized and rest will be converted to lowercase. If first letter is already capitalized, it will be changed to lowercase
endswith()	Checks if a string ends with specified characters
startswith()	Checks if a string starts with specified characters

Example 3.1

In this example, methods from *table 3.1* will be examined

Code:

```
#   Example 3.1
#   This example shows use of different string functions
#

print("\n\n")
print("Sample string is:\n")
str = "EET4370C - Advanced Programming Applications"    #   Sample string
print(str)
print("\n") #   "\n" is the new line specifier

print("split")
print("-----")
A = str.split()
print(A,"\n")   #   "\n" is the new line specifier

print("index for 'a'")
print("-----")
B = str.index("a")
print(B, "\n")

print("count 'a'")
print("-----")
C = str.count("a")
print(C, "\n")

print("upper")
print("-----")
D = str.upper()
print(D, "\n")

print("lower")
print("-----")
E = str.lower()
print(E, "\n")

print("find")
print("-----")
F = str.find("a")
print(F, "\n")

print("rfind")
print("-----")
G = str.rfind("a")
print(G, "\n")
```

Output:

```
Sample string is:

EET4370C - Advanced Programming Applications


split
-----
['EET4370C', '-', 'Advanced', 'Programming', 'Applications

index for 'a'
-----
14

count 'a'
-----
3

upper
-----
EET4370C - ADVANCED PROGRAMMING APPLICATIONS

lower
-----
eet4370c - advanced programming applications

find
-----
14


rfind
-----
38
```

```python
print("strip")
print("-----")
H = str.strip("EET")
print(H, "\n")

print("lstrip")
print("-----")
I = str.lstrip("EET")
print(I, "\n")

print("rstrip")
print("-----")
J = str.rstrip("erions")
print(J, "\n")

print("swapcase")
print("-----")
K = str.swapcase()
print(K, "\n")

print("ljust")
print("-----")
L = str.ljust(50, "*")
print(L, "\n")

print("rjust")
print("-----")
M = str.rjust(50, "*")
print(M, "\n")

print("center")
print("-----")
N = str.center(50, "*")
print(N, "\n")

print("zfill")
print("-----")
O = str.zfill(50)
print(O, "\n")
```

```
strip
-----
4370C - Advanced Programming Applications

lstrip
-----
4370C - Advanced Programming Applications

rstrip
-----
EET4370C - Advanced Programming Applicat

swapcase
-----
eet4370c - aDVANCED pROGRAMMING aPPLICATIONS

ljust
-----
EET4370C - Advanced Programming Applications******

rjust
-----
******EET4370C - Advanced Programming Applications

center
-----
***EET4370C - Advanced Programming Applications***

zfill
-----
000000EET4370C - Advanced Programming Applications
```

```python
print("replace")
print("-----")
P = str.replace("a", "A",2)
print(P, "\n")

print("capitalize")
print("-----")
Q = str.capitalize()
print(Q, "\n")

print("endswith")
print("-----")
R = str.endswith("ing")
print(R, "\n")

print("endswith")
print("-----")
S = str.endswith("ions")
print(S, "\n")

print("startswith")
print("-----")
T = str.startswith("Eet")
print(T, "\n")

print("startswith")
print("-----")
U = str.startswith("EET")
print(U, "\n")
```

```
replace
-----
EET4370C - AdvAnced ProgrAmming Applications

capitalize
-----
Eet4370c - advanced programming applications

endswith
-----
False

endswith
-----
True

startswith
-----
False

startswith
-----
True
```

String Formatting:

A string can be formatted in two common ways; either using '%' specifier, like in MATLAB, or using *string.format()* method. These syntax are already used in different examples earlier. They will be examined in detail in this sub-section [8], [9].

Formatting strings using '%' specifier is an old method, which is quite similar to MATLAB formatting. The syntax is as follows:

"characters %f characters %d characters %s" %(variable1, variable2, variable3)

In this syntax, *characters* are any characters and *%f, %d, and %s* are specifiers that will be replaced by *variable1*, which should be a floating point, *variable2*, which should be an integer, and *variable3*, which should be a string, respectively.

```
In [14]: "a = %d plus b = %f will yield %f" %(a, b, a+b)
Out[14]: 'a = 4 plus b = 4.350000 will yield 8.350000'
```

Some other common specifiers are *%e* for exponential numbers and *%g* for general numbers, which formats number either in the regular fixed-point or exponential is they are very large or very small.

Specifiers can also be adjusted by *%<integer_part.decimal_part>f* to represent a floating point by a specific number of integer and decimal places. Same can be done for integers and strings but there is no decimal part for them in the specifier.

Example 3.2

Ask user to enter an integer *x* and a floating-point *y*. Evaluate and print out the result for the following expression using % specifiers:

$$f(x,y) = \frac{2\sin(xy)}{x+y}$$

Code:

```
#   Example 3.2
#   This code evluates f(x,y) = 2sin(xy)/(x+y)

x = int(input('Enter an integer x: '))
y = float(input('Enter a number y: '))
s = "2sin(xy)/(x+y)"

from math import sin

fxy = 2*sin(x*y)/(x+y)

print("\nThe result for the expression %s for x = %d and y = %3.2f is %3.2f" %(s,x,y,fxy))
```

Output:

```
Enter an integer x: 2

Enter a number y: 3.4

The result for the expression 2sin(xy)/(x+y) for x = 2 and y = 3.40 is 0.18
```

The new way to perform formatting of a string is by *string.format()* method.

"characters {} characters {}".format(variable1, variable2)

In this syntax, first set of brackets will be replaced by *variable1* and second set will be replaced by *variable2*:

```
In [11]: 'a = {} and b = {}'.format(a,b)
Out[11]: 'a = 4 and b = 4.35'
```

In curly brackets, specific formatting types can be given that correspond to the list of variables in parentheses. A list of common formatting types that can be used in formatting a string is given in *Table 3.2* [10]

Table 3.2: List of Formatting Types

Type	Meaning
d	Decimal integer
c	Corresponding Unicode character
b	Binary format
o	Octal format
x	Hexadecimal format (lower case)
X	Hexadecimal format (upper case)
n	Same as 'd'. Except it uses current locale setting for number separator
e	Exponential notation. (lowercase e)
E	Exponential notation (uppercase E)
f	Displays fixed point number (Default: 6)
F	Same as 'f'. Except displays 'inf' as 'INF' and 'nan' as 'NAN'
g	General format. Rounds number to p significant digits. (Default precision: 6)
G	Same as 'g'. Except switches to 'E' if the number is large.
%	Percentage. Multiples by 100 and puts % at the end.

The difference between the old type and new type of formatting is to replace % by *{:}*. *%d, %f, %s, %e, and %g* are written as *{:d}, {:f}, {:s}, {:e},* and *{:g}*. To convert a number in different bases, the syntax is *{:b}, {:o}, and {:x}* for binary, octal, and hexadecimal conversion.

```
In [17]: '42 in binary is {:b}'.format(42)
Out[17]: '42 in binary is 101010'
```

Example 3.3

Repeat *Example 3.2* and print out your results with new formatting

Code:

```
#   Example 3.3
#   This code evluates f(x,y) = 2sin(xy)/(x+y) and prints result out using new formatting

x = int(input('Enter an integer x: '))
y = float(input('Enter a number y: '))
s = "2sin(xy)/(x+y)"

from math import sin

fxy = 2*sin(x*y)/(x+y)

print("\nThe result for the expression {:s} for x = {:d} and y = {:3.2f} is {:3.2f}".format(s,x,y,fxy))
```

Output:

```
Enter an integer x: 4

Enter a number y: 4.5

The result for the expression 2sin(xy)/(x+y) for x = 4 and y = 4.50 is -0.18
```

Example 3.4

Ask user to enter a number. Convert the number in binary, octal, and hexadecimal, and print them out

Code:

```
#   Example 3.4
#   This program converts a number entered by user in different bases

x = int(input("Enter a number: "))

print("Input number is {}, which is {:b} in binary. {:o} in octal, and {:x} in hexadecimal".format(x,x,x,x))
```

Output:

```
Enter a number: 132
Input number is 132, which is 10000100 in binary. 204 in octal, and 84 in hexadecimal
```

Alignment of a string can also be done using the new format. *Table 3.3* shows the alignment options [5]

Table 3.3: String Alignment

Option	Meaning
'<'	Forces the field to be left-aligned within the available space (this is the default for most objects).
'>'	Forces the field to be right-aligned within the available space (this is the default for numbers).
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types. It becomes the default when '0' immediately precedes the field width.
'^'	Forces the field to be centered within the available space.

There is also a *sign* option available in formatting, as shown in *Table 3.4* [5]

Table 3.4: Sign Options in Formatting

Option	Meaning
'+'	indicates that a sign should be used for both positive as well as negative numbers.
'-'	indicates that a sign should be used only for negative numbers (this is the default behavior).
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

Example 3.5

Evaluate an equation, $f(x, y) = 2y\sin(x) + 4\cos(y)$ for x = -1 to 2 with an increment of 0.3 and y = [3, 8, 10, -12]. Present your results in a proper tabular form

Code:

```
#   Example 3.5
#   Evaluate an equation, f(x,y) = 2*y*sin(x) + 4*cos(y) for  x = -1 to 2 with increment of 0.3 and y = [3, 8, 10, -12]
#   This example is similar to Example 2.5 except output is printed in the proper tabular form
y_values = [3, 8, 10, -12]
x_range = 3/0.3 + 1 #   Total number of points for x

from math import sin, cos

#   printing 'x' and 'y' center aligned within their space, and 'f' right aligned
print("\n{:^18} {:^20} {:>8}\n".format('x', 'y', 'f')) #   {:^20s} will be the same
for y in y_values:
    for x in range(int(x_range)):
        x_values = -1 + x*0.3    #   x_values go from -1 to 2 with an increment of 0.3 when x goes from 0 to 10
        f = 2*y*sin(x_values) + 4*cos(y)
        print("{:^18.2f} {:>10d} {:>20.2f}".format(x_values,y,f))
```

Output:

Part of the output is shown

```
      x                   y                   f

    -1.00                 3               -9.01
    -0.70                 3               -7.83
    -0.40                 3               -6.30
    -0.10                 3               -4.56
     0.20                 3               -2.77
     0.50                 3               -1.08
     0.80                 3                0.34
     1.10                 3                1.39
     1.40                 3                1.95
     1.70                 3                1.99
     2.00                 3                1.50
    -1.00                 8              -14.05
    -0.70                 8              -10.89
    -0.40                 8               -6.81
    -0.10                 8               -2.18
     0.20                 8                2.60
     0.50                 8                7.09
     0.80                 8               10.90
     1.10                 8               13.68
     1.40                 8               15.19
     1.70                 8               15.28
     2.00                 8               13.97
    -1.00                10              -20.19
    -0.70                10              -16.24
    -0.40                10              -11.14
```

Universal Character Set or Unicode characters can also be printed through a string by using *\u* followed by its code point in hexadecimal [11]. A list of Unicode characters and their code points can be found at Wikipedia [12]

```
In [12]: print("Omega: \u03A9; Copyright: \u00A9; Pi: \u03C0")
Omega: Ω; Copyright: ©; Pi: π
```

If a string is multiplied by a number, it will be repeated by that many times;

```
In [11]: string = 'eet4370C'

In [12]: print(2*string)
eet4370Ceet4370C
```

3.2 Lists

Lists are arrays of numbers and/or characters defined inside square brackets. Lists can be single-dimensional or multi-dimensional.

```
In [9]: #single-dimensional list

In [10]: A = [1, 8, 7, 8.9, 10]

In [11]: type(A)
Out[11]: list
```

```
In [12]: #Multi-dimensional list

In [13]: B = [[1,2],[4.5,89],[3.4,9.8]]

In [14]: C = [[2,8],
    ...:      [5,9],
    ...:      [10,87]
    ...:     ]
    ...:
```

Length of a list can be found out by *len()* function. Note that for multidimensional list, *len()* will take each multidimensional entity to be a single entity and generate length of the list based on the number of multidimensional entities.

List can also be comprised of a mix of numbers and strings;

```
In [1]: list = [1, 4.5, 67, 'eet4370c', 98]

In [2]: list[0]
Out[2]: 1

In [3]: list[3]
Out[3]: 'eet4370c'

In [4]: type(list[3])
Out[4]: str

In [5]: type(list[2])
Out[5]: int

In [6]: type(list[1])
Out[6]: float

In [7]: len(list)
Out[7]: 5

In [8]: len(list[3])
Out[8]: 8
```

Observe that the list is comprised of integers, a floating point, and a string. *type()* of each component of list can yield its data type. Also, the length of the list is five but length of each component of list can be different. List can also be created with single-dimensional and multi-dimensional entities

```
In [9]: list = [1.2, [3.4, 6.5], [4.5, 3, 2], "cs"]

In [10]: len(list)
Out[10]: 4
```

To remove an entity from a list, *del()* function can be used.

```
In [37]: B = [2, 3, 4, 5, 6, 7, 8]

In [38]: del(B[2])

In [39]: B
Out[39]: [2, 3, 5, 6, 7, 8]

In [40]: del(B[1:3])

In [41]: B
Out[41]: [2, 6, 7, 8]
```

To remove a variable altogether, *del(variable)* can be used

To combine two lists, a '+' sign can be used

```
In [22]: W = [12, 34, 56, 78]

In [23]: Q = [56, 78]

In [24]: W+Q
Out[24]: [12, 34, 56, 78, 56, 78]
```

To repeat a list, it can be multiplied by any integer to repeat it that many times

```
In [25]: 3*W
Out[25]: [12, 34, 56, 78, 12, 34, 56, 78, 12, 34, 56, 78]
```

List Comprehensions

List comprehensions are a useful way to create lists whose contents obey some rule [6].

Example 3.6

Create a list of results obtained by evaluating expression $2x^2$ for odd values of x from 1 to 19

Code:

```
#   Example 3.6
#   List Comprehension example # 1

TwoX_2 = [2*x**2 for x in range(1,20,2)]
print(TwoX_2)
```

Output:

```
[2, 18, 50, 98, 162, 242, 338, 450, 578, 722]
```

Example 3.7

Create a list for *2sin(x)cos(x)* for *x* = -2 to 2 with a step-size of 0.1 using list comprehension. Print out your results in a proper tabular form

Code:

```
#   Example 3.7
#   List comprehensions example 2

#   Make a list from the result of 2sin(x)cos(x) for x = -2 to 2 with step size of 0.01
from math import sin, cos
Total_points = int(4/0.01 + 1)  #   total number of points to be evaluated
x = [q*0.01-2 for q in range(Total_points)] #   x is the list of numbers from -2 to 2 with interval of 0.01
y = [2*sin(x[i])*cos(x[i]) for i in range(Total_points)]    #   y is the list comprised of 2sin(x)cos(x)

print("\n{:^15s} {:^20s}".format("x","2sin(x)cos(x)\n"))
for i in range(Total_points):
    print("{: >10.2f} {:>15.2f}".format(x[i],y[i]))
```

Output:

Part of the output is shown

```
     x           2sin(x)cos(x)

   -2.00            0.76
   -1.99            0.74
   -1.98            0.73
   -1.97            0.72
   -1.96            0.70
   -1.95            0.69
   -1.94            0.67
   -1.93            0.66
   -1.92            0.64
   -1.91            0.63
   -1.90            0.61
   -1.89            0.60
   -1.88            0.58
   -1.87            0.56
   -1.86            0.55
   -1.85            0.53
   -1.84            0.51
   -1.83            0.50
   -1.82            0.48
   -1.81            0.46
   -1.80            0.44
   -1.79            0.42
   -1.78            0.41
   -1.77            0.39
   -1.76            0.37
   -1.75            0.35
   -1.74            0.33
   -1.73            0.31
   -1.72            0.29
   -1.71            0.27
   -1.70            0.26
   -1.69            0.24
   -1.68            0.22
   -1.67            0.20
```

List Input:

Using list comprehension, a list of numbers can be obtained from a user with the following syntax:

A = [float(B) for B in input().split(',')]

When this piece of code is used, user will enter input numbers separated by commas, which will create a list of floating points. If it is desired to ask user to enter numbers with space between them to create a list, *split()* function will represent space instead of comma in the code. If instead of floating points, integers are required, *float()* will be replaced by *int()*

Example 3.8

Ask user to enter a list of numbers to evaluate the following expression:

$$f(x) = 5x^2 + 3x + 8 + 2\cos(x)$$

Code:

```
#   Example 3.8
#   This program will ask user to enter input that will be saved as a list
#   This input list will be used to evaluate 5x^2+3x+8+2cos(x)

from math import cos
x = [float(B) for B in input("Enter values for 'x' seperated by commas to evaluate the expression: ").split(',')]
y = [5*x**2+3*x+8+2*cos(x) for x in x]

print("\n{:^15s} {:^20s}".format("x","5x^2+3x+8+2cos(x)\n"))
for i in range(len(x)):
    print("{: >10.2f} {:>15.2f}".format(x[i],y[i]))
```

Output:

```
Enter values for 'x' seperated by commas to evaluate the expression: 12.3, 4.5, 6.7, -10.2,
87.6

       x         5x^2+3x+8+2cos(x)

     12.30          803.28
      4.50          122.33
      6.70          254.38
    -10.20          496.17
     87.60        38641.47
```

Methods for Lists

Some of the common methods that can be used for list object are shown in *figure 3.1* [10]

`list.` **`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.` **`extend`**(*iterable*)
Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

`list.` **`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.` **`remove`**(*x*)
Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.` **`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.` **`clear`**()
Remove all items from the list. Equivalent to `del a[:]`.

`list.` **`index`**(*x*[, *start*[, *end*]])
Return zero-based index in the list of the first item whose value is equal to *x*. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

`list.` **`count`**(*x*)
Return the number of times *x* appears in the list.

`list.` **`sort`**(*key=None*, *reverse=False*)
Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).

`list.` **`reverse`**()
Reverse the elements of the list in place.

`list.` **`copy`**()
Return a shallow copy of the list. Equivalent to `a[:]`.

Figure 3.1: Common methods applied to list object

Example 3.9

An example for list methods will be shown here

Code:

```
#   Example 3.9
#   This example shows different methods for list object

A = [1, 3.4, 5, 8, 9, "apples", 6.7] #   This list will be used to demostrate different methods
print('Sample list: ', A)

#   list.append
print('\nlist.append("bananas")')
A.append("bananas")
print(A)

#   list.insert
print('\nlist.insert(5, "oranges")')
A.insert(5, "oranges")
print(A)

#   list.remove
print('\nlist.remove("bananas")')
A.remove("bananas")
print(A)

#   list.pop
print('\nlist.pop(4)')
print(A.pop(4))
print(A)

#   list.index
print('\nlist.index("apples")')
print(A.index("apples"))

#   list.count
print('\nlist.count(3.4)')
print(A.count(3.4))

#   list.sort
B = [9.8, -1, 56, 23, 9]
print('\nlist.sort()')
print("\nNumber string to test sort: ",B)
B.sort()
print("sorted string: ",B)

#   list.revers
print('\nlist.reverse()')
A.reverse()
print(A)

#   list.extend
print('\nlist.extend(another list)')
A.extend(B)
print(A)
```

Output:

```
Sample list:  [1, 3.4, 5, 8, 9, 'apples', 6.7]

list.append("bananas")
[1, 3.4, 5, 8, 9, 'apples', 6.7, 'bananas']

list.insert(5, "oranges")
[1, 3.4, 5, 8, 9, 'oranges', 'apples', 6.7, 'bananas']

list.remove("bananas")
[1, 3.4, 5, 8, 9, 'oranges', 'apples', 6.7]

list.pop(4)
9
[1, 3.4, 5, 8, 'oranges', 'apples', 6.7]

list.index("apples")
5

list.count(3.4)
1

list.sort()

Number string to test sort:  [9.8, -1, 56, 23, 9]
sorted string:  [-1, 9, 9.8, 23, 56]

list.reverse()
[6.7, 'apples', 'oranges', 8, 5, 3.4, 1]

list.extend(another list)
[6.7, 'apples', 'oranges', 8, 5, 3.4, 1, -1, 9, 9.8, 23, 56]
```

Some other techniques to print Lists

In addition to *for* loops as discussed earlier, *lists* can also be printed without a *for* loop, by converting a list into a string, or by using *map* function [34]. * operator can be used to print a list with different styles without a *for* loop, as shown in the following example.

Example 3.10:

Use * operator to print a list without commas, with commas, and in a column instead of a row.

Code:

```
# Example 3.10

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

# printing the list using * operator separated by space
print("\nPrinting lists separated by space")
print(*a)

# printing the list using * and sep operator
print("\nPrinting lists separated by commas")
print(*a, sep = ", ")

# print in new line as a column vector
print("\nPrinting lists in new line (as a column vector)")
print(*a, sep = "\n")
```

Output:

```
Printing lists separated by space
1 2 3 4 5

Printing lists separated by commas
1, 2, 3, 4, 5

Printing lists in new line (as a column vector)
1
2
3
4
5
```

3.3 Dictionaries

Dictionaries are data structures used to assign *keys* to values. Each element in a dictionary is represented by a *key:value* pair. Lists can be considered as dictionaries where each element is assigned an integer key; its index. However, key (index) for a dictionary can be any number or string.

```
In [1]: dict = {12:56, 34:98, 2.3:109.4}

In [2]: len(dict)
Out[2]: 3

In [3]: dict[2.3]
Out[3]: 109.4

In [4]: dict[34]
Out[4]: 98
```

```
In [6]: My_Dictionary = {"Course":"EET4370C", "Time":9, "Day":"Tuesday", "Week":3}

In [7]: len(My_Dictionary)
Out[7]: 4

In [8]: My_Dictionary["Day"]
Out[8]: 'Tuesday'

In [9]: My_Dictionary["Week"]
Out[9]: 3
```

Any item in dictionary can be changed by assigning a new value to its key

```
In [10]: My_Dictionary["Week"] = 8

In [11]: My_Dictionary
Out[11]: {'Course': 'EET4370C', 'Time': 9, 'Day': 'Tuesday', 'Week': 8}
```

Any item can be added to a dictionary by assigning value to its key

```
In [13]: My_Dictionary[19]=234

In [14]: My_Dictionary
Out[14]: {'Course': 'EET4370C', 'Time': 9, 'Day': 'Tuesday', 'Week': 8, 19: 234}
```

To find out if a key is in a dictionary, *in* and *not in* can be used

```
In [14]: My_Dictionary
Out[14]: {'Course': 'EET4370C', 'Time': 9, 'Day': 'Tuesday', 'Week': 8, 19: 234}

In [15]: print("Course" in My_Dictionary)
True

In [16]: print(20 not in My_Dictionary)
True

In [17]: print("Time" not in My_Dictionary)
False

In [18]: print("time" not in My_Dictionary)
True
```

Some of the common dictionary functions are:

list(dictionary) function lists all keys of the dictionary [10]

```
In [2]: d = {"Course":"CET4370C", 12:567, "A":[1, 2, 3, 4]}

In [3]: list(d)
Out[3]: ['Course', 12, 'A']
```

del dictionary[key] deletes the key and value from a dictionary

```
In [4]: del d["Course"]

In [5]: d
Out[5]: {12: 567, 'A': [1, 2, 3, 4]}
```

sorted(dictionary) function can list the keys in an increasing order if keys are numbers

```
In [12]: A
Out[12]: {12: 345, 43: 'we', -12: 'car', 12.3: 45}

In [13]: sorted(A)
Out[13]: [-12, 12, 12.3, 43]
```

dict() function can build a dictionary directly from key-value pairs

```
In [14]: dict([(1,2), ('sep', 13), (24.3,"tree")])
Out[14]: {1: 2, 'sep': 13, 24.3: 'tree'}
```

```
In [16]: dict(a=12, b=16, c=90, course ="EEt4370c")
Out[16]: {'a': 12, 'b': 16, 'c': 90, 'course': 'EEt4370c'}
```

Dictionary Comprehensions

Like list comprehensions, *dictionary comprehension* can create a dictionary from evaluated values of a function for the values at which the function is evaluated. The syntax for dictionary comprehension is:

dict = {x: f(x) for x in (values)<optional condition>} or

dict = {x: f(x) for x in [values] <optional condition>}

Note that *<optional condition>* can be any condition used on *x* values (e.g. *if x > 0*)

Example 3.11

Create a dictionary when an expression $f(x) = \dfrac{12x^2 + 3x + 5\cos(4x)}{\sqrt{|\sin(x)|}}$ is evaluated for *x* = (12, 3, 89, -23, 14, 89)

Code:

```
#   Example 3.10
#   Dictionary Comprehension

from math import cos, sin, sqrt

fx = {x: (12*x**2+3*x+5*cos(4*x))/sqrt(abs(sin(x))) for x in (12, 3, 89, -23, 14, 89.5)}
print(fx)
```

Output:

```
{12: 2403.784204124345, 3: 322.68395202508384, 89: 102778.06087285372, -23:
6822.319255751679, 14: 2409.6090991570227, 89.5: 96426.64506899116}
```

Methods for Dictionaries

There are different methods that are used with dictionary object. Some of the common methods are given in *Table 3.5*. [13]

Table 3.5: Common Dictionary Methods

Method	*Explanation*
dictionary.clear()	Removes all Items from a dictionary
dictionary.copy()	Returns a shallow copy of a dictionary
dictionary.fromkeys()	Creates a dictionary from a given sequence
dictionary.get()	Returns value of key
dictionary.items()	Returns a view object that displays a list of dictionary's (key, value) tuple pairs.
dictionary.keys()	Returns view object of all keys
dictionary.popitem()	Returns and removes an arbitrary element (key, value) pair from the dictionary.
dictionary.setdefault()	Returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.
dictionary.pop()	Removes and returns an element from a dictionary having the given key.
dictionary.values()	Returns view of all values in dictionary
dictionary.update()	Updates the dictionary with the elements from the another dictionary object or from an iterable of key/value pairs.

Example 3.12

Examine all methods given in *Table 3.5*

```
In [1]: A = {23:768.54, 9:45, 45.6:98.6, "course":'EET4370C'}

In [2]: B = {}

In [3]: A.clear()

In [4]: A
Out[4]: {}

In [5]: A = {23:768.54, 9:45, 45.6:98.6, "course":'EET4370C'}

In [6]: B.fromkeys({45, 87, 90, 'q'}, 100)
Out[6]: {90: 100, 45: 100, 'q': 100, 87: 100}

In [7]: A.get(9)
Out[7]: 45

In [8]: A.items()
Out[8]: dict_items([(23, 768.54), (9, 45), (45.6, 98.6), ('course', 'EET4370C')])

In [9]: A.keys()
Out[9]: dict_keys([23, 9, 45.6, 'course'])

In [10]: A.popitem()
Out[10]: ('course', 'EET4370C')
```

```
In [11]: A
Out[11]: {23: 768.54, 9: 45, 45.6: 98.6}

In [12]: A.setdefault(100,987)
Out[12]: 987

In [13]: A
Out[13]: {23: 768.54, 9: 45, 45.6: 98.6, 100: 987}

In [14]: A.pop(23)
Out[14]: 768.54

In [15]: A
Out[15]: {9: 45, 45.6: 98.6, 100: 987}

In [16]: A.values()
Out[16]: dict_values([45, 98.6, 987])

In [17]: A.update({100:1000})

In [18]: A
Out[18]: {9: 45, 45.6: 98.6, 100: 1000}

In [19]: A.update({1000:'d'})

In [20]: A
Out[20]: {9: 45, 45.6: 98.6, 100: 1000, 1000: 'd'}
```

3.4 Tuples

Tuples are very similar to lists, except that they are immutable (they cannot be changed). Also, they are created using *parentheses*, rather than square brackets.

```
In [40]: A = (2, 3.4, 7.8, 10.9, -34.5)

In [42]: B = (1.2, 'cat', 78, 1e6, [1, 2, 3])

In [43]: len(B)
Out[43]: 5

In [45]: B[3]
Out[45]: 1000000.0
```

Assigning a value to a tuple index generates an error as tuples are immutable

```
In [46]: B[3] = 5
Traceback (most recent call last):

  File "<ipython-input-46-a8bb482bf24d>", line 1, in <module>
    B[3] = 5

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

Tuples can also be created by assigning different data types without parentheses.

```
In [47]: T = 1, 9, 8.7, 1.2e2, 'core'

In [48]: T
Out[48]: (1, 9, 8.7, 120.0, 'core')
```

An empty tuple can be created with an empty parentheses pair: *tuple()*

3.5 Sets

Sets are data structures, similar to lists or dictionaries. They are created using curly braces, or the *set*() function [6]

```
In [56]: set1 = {1, 2, 3, 4.5, 6.7, 'door'}

In [57]: set1
Out[57]: {1, 2, 3, 4.5, 6.7, 'door'}

In [58]: set2 = set([1, 5, 7, 2, 5.4, 9.8])

In [59]: set2
Out[59]: {1, 2, 5, 5.4, 7, 9.8}

In [62]: len(set1)
Out[62]: 6
```

Sets differ from lists in several ways, but share several list operations. They are unordered, which means that they can't be indexed. They cannot contain duplicate elements. Due to the way they're stored, it's faster to check whether an item is part of a set, rather than part of a list.

To check if an entity is part of a set, *in* and *not in* can be used.

```
In [61]: print(2 in set2)
True
```

When a set is assigned duplicate elements, these elements are counted only once when it is created.

```
In [65]: set4 = {12.3, 3.4, 8.7, 2, -1, 8, -1, -56, -2}

In [66]: set4
Out[66]: {-56, -2, -1, 2, 3.4, 8, 8.7, 12.3}
```

Observe that duplicate elements are removed from *set4* and it has been organized in increasing order of numbers.

Table 3.6 shows some of the commonly used operations (methods) on a set [14], [15]

Table 3.6: Operations on Sets

Operation	*Equivalent*	*Result*
s.issubset(t)	s <= t	test whether every element in *s* is in *t*
s.issuperset(t)	s >= t	test whether every element in *t* is in *s*
s.union(t)	s | t	new set with elements from both *s* and *t*
s.intersection(t)	s & t	new set with elements common to *s* and *t*
s.difference(t)	s – t	new set with elements in *s* but not in *t*
s.symmetric_difference(t)	s ^ t	new set with elements in either *s* or *t* but not both
s.update(t)	s |= t	return set *s* with elements added from *t*
s.intersection_update(t)	s &= t	return set *s* keeping only elements also found in *t*
s.difference_update(t)	s -= t	return set *s* after removing elements found in *t*
s.symmetric_difference_update(t)	s ^= t	return set *s* with elements from *s* or *t* but not both
s.add(x)		add element *x* to set *s*
s.remove(x)		remove *x* from set *s*; raises 'KeyError' if not present
s.discard(x)		removes *x* from set *s* if present
s.pop()		remove and return an arbitrary element from *s*; raises 'KeyError' if empty
s.clear()		remove all elements from set *s*
s.copy()		new set with a shallow copy of *s*

Example 3.13

Ask user to enter to different sequence of numbers to create two sets. Evaluate the following expression for the values that are common in both input sets.

$$f(x) = 2x^2 + 5x + 6$$

Code:

```
#   Example 3.12
#   Ask user to enter two number sets. Evaluate an expression for the values which are
#   common in both sets and create a dictionary with values as keys and results as values

s1 = [float(a) for a in input('Enter some numbers seperated by commas to create set 1: ').split(',')]
s2 = [float(a) for a in input('Enter some numbers seperated by commas to create set 1: ').split(',')]

s1 = set(s1)    #   creating set
s2 = set(s2)    #   creating set

s3 = s1 & s2    #   A new set with elements common in both s1 and s2

y = {x: 2*x**2 + 5*x + 6 for x in s3}   # dictionary comprehension
print('\n',y)
```

Output:

```
Enter some numbers to create set 1: 23, 98, 89, -19

Enter some numbers to create set 1: -19, 23, 67, 89, 24, 65

 {89.0: 16293.0, -19.0: 633.0, 23.0: 1179.0}
```

Exercises

3.1 Ask user to enter names of three classes; *Advanced Programming Applications*, *Microcontroller Devices*, and *Data Communications*. If user enters the first one, print out that the class is in the morning, for second one, print out that it is in the evening, and for the third one, print out that it is in the afternoon. Make sure that user can enter the class name with any letter uppercase or lowercase.

3.2 Evaluate the equation $f(x) = \dfrac{2\cos(x) + 5\sin(\cos(5x))}{\sqrt{|\sin(4x)|}}$ for 100 points divided between –*pi* to *pi* . Print out values of *x* and *f(x)* in a proper tabular form.

3.3 Ask user to enter some numbers as a list. Evaluate the expression given in *problem 3.2* for the numbers in the list and print out results in a proper tabular form.

3.4 Ask user to enter some resistance values as a *list*. Also, ask if resistors are connected in series or parallel. First check if any value in the list is negative. If so, print "Resistors can't be negative". If all values are positive then based on if they are connected in series or parallel, calculate the equivalent resistance and print it out with an appropriate message.

$$R_{series} = R_1 + R_2 + R_3 + \cdots R_n$$

$$R_{parallel} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \cdots \frac{1}{R_n}}$$

[Note: You can use *any()* to check out specific values based on some condition in a list:

*any(*condition *for* condition_variable *in* list*)*]

3.5 Redo *problem 3.4* and check for negative values using *list.sort()* method

3.6 Ask user to enter some values for which equation from *exercise 3.2* can be evaluated. Evaluate the equation and create a dictionary from values and results.

3.7 Create a dictionary for *problem 3.6* for only positive values

3.8 Ask user to enter two sets of numbers. Create another set with only numbers that are not common between the two sets. Use this third set to evaluate the following expression:

$$f(x) = 5x\cos(6x) + x^2\sin(x)$$

Save your results in a set and print it

Chapter 4

User-Defined Functions

Like in other procedural programming languages, you can also write your own functions in Python. *Functions* are programs that meant to be run several times or they are called from different programs. Hence, they are frequently re-used. For example, *print*, *input*, *sin*, *cos*, *sqrt* etc., are all functions that are regularly used in different programs or from python console. Whenever a function is called, it goes to the program written for that function, executes it, compile results, and goes back to the program from where it was called with results.

4.1 Defining a Function

In Python, syntax of a function is as follows:

def function_name(optional variables):

code

If a function needs to return some values, *return <values >* syntax is used in the code

Example 4.1

Write a function to evaluate a quadratic equation $2x^2 + 3x + 4$ for an input argument x

Code:

```
#   Example 4.1
#
#   This function will evaluate a quadratic equation 2x^2 + 3x + 4 for 'x'

def Ex4_1(x):
    y = 2*x**2 + 3*x + 4
    return y


a = Ex4_1(2.3)
print(a)

b = Ex4_1(5)
print(b)
```

Output:

```
21.479999999999997
69
```

If a function is to be called from another program, the program and the function should be saved in the same folder. The program that is calling the function needs to import it first before calling it:

from function_file import function

Hence, the function file is treated as a module where function is located.

Example 4.2

Call the function from another program to evaluate the quadratic equation for an input

Code:

```
#   Example 4.2
#   Calling a function from another program
#   This program will call the function created in Example 4.1

from Example4_1 import Ex4_1

print('\n The result of the quadratic equation 2x\u00B2 + 3x + 4 for x = 3 is {}'.format(Ex4_1(3)))
```

Output:

```
The result of the quadratic equation 2x² + 3x + 4 for x = 3 is 31
```

Observations:

Observe the use of Unicode to print $2x^2$. Also, *Example 4.1* code is modified to have only function *Ex4_1()*. Rest of the code that was calling the function is removed.

If a function module is required to be accessed from any folder (system-wide), it has to be saved somewhere on the *PYTHONPATH*. Usually, saving it to `……\Python37-32\Lib\site-packages` should work (assuming it is running on Windows) [16].

A function can also take multiple input variables of different data types and give out multiple outputs of different data types.

Example 4.3

Create a function to evaluate the following expression:

$$f(x,\alpha) = 2\cos(\alpha x) + 5\sin(3\alpha x)$$

Input arguments are *x*, which can be a floating-point, integer, list, tuple or set, and α, which is an integer or floating-point. Output of the function should be a floating-point, integer, or a list, same length as input *x*.

Code:

```
#   Example 4.3

def Ex4_3(z,a):

    from math import sin, cos

    if type(z) == list:
        f = [2*cos(a*x) + 5*sin(3*a*x) for x in z]
    else:
        f = 2*cos(a*z) + 5*sin(3*a*z)

    return f
```

Output:

```
In [16]: from Example4_3 import Ex4_3

In [17]: print(Ex4_3(2.3,2))
4.4941732933504115

In [22]: print(Ex4_3([2.3, 4, 5.6, 7],8))
[-3.075873273097626, 6.586385448184745, 4.544634529457648, -3.2794262284252316]
```

Multiple functions can also be defined within the same program and called individually

Example 4.4:

Write a program to calculate *neper frequency*, *resonant frequency*, and *roots* of the characteristic equation for series and parallel *RLC* transient circuits. Define one function for series and the other for parallel circuits.

Code:

```
#   Example 4.4
#   This example will house two functions; one to calculate parameters for series
#   RLC transient circuits and the other for parallel circuits

def series(R, L, C):
    import math, cmath
    Neper = 1/(2*R*C)
    Resonant = 1/math.sqrt(L*C)
    Root1 = -Neper + cmath.sqrt(Neper**2 - Resonant**2)
    Root2 = -Neper - cmath.sqrt(Neper**2 - Resonant**2)
    return(Neper, Resonant, Root1, Root2)

def parallel(R, L, C):
    import math, cmath
    Neper = R/(2*L)
    Resonant = 1/math.sqrt(L*C)
    Root1 = -Neper + cmath.sqrt(Neper**2 - Resonant**2)
    Root2 = -Neper - cmath.sqrt(Neper**2 - Resonant**2)
    return(Neper, Resonant, Root1, Root2)
```

Output:

```
In [39]: from Example4_4 import series, parallel

In [40]: R = 100; L = 2e-3; C = 40e-6

In [41]: Series_Transient = series(R, L, C)

In [42]: Parallel_Transient = parallel(R, L, C)

In [43]: print(Series_Transient)
(125.0, 3535.5339059327375, (-125+3533.3235062756426j), (-125-3533.3235062756426j))

In [44]: print(Parallel_Transient)
(25000.0, 3535.5339059327375, (-251.26265847083778+0j), (-49748.737341529166+0j))
```

Observations:

(i) When you are assigning values to different variables in the same line, separate them by semicolon

(ii) Output from the functions is returned as a *tuple*. Individual values from the output can be obtained by accessing the specific index of the tuple. For example, *neper frequency* for the series transient circuit can be assigned to a variable as *Series_Transient[0]*

So far, all examples discussed have input and output arguments to be numbers. Let's look at an example where input argument is a string and function doesn't use *return* statement.

Example 4.5

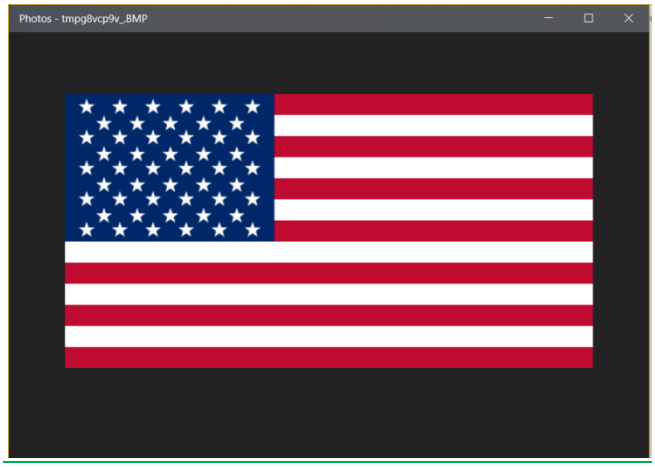
Write a function that takes URL of an image and displays it [4]

Code:

```
#   Example 4.5
#   This function will take URL of an image as a string and then display the image

def Ex4_5(URL):

    #    urllib is a module where url submodules and functions are located
    from urllib.request import urlretrieve

    #   PIL (Python Image Library) is a module that deals with images
    from PIL import Image

    image_name = 'image1'    #  assigning a name to the image that will be retrieved from the URL

    #   image will be downloaded from URL and saved in the variable image_name with name image1
    urlretrieve(URL, image_name)

    Image.open(image_name).show()
```

Output:

```
In [71]: from Example4_5 import Ex4_5

In [72]: Ex4_5('https://cdn.britannica.com/33/4833-004-297297B9.jpg')
```



```
In [74]: Ex4_5('https://www.lamborghini.com/sites/it-en/files/DAM/it/models_gateway/blocks/huracan.png')
```

Note that once you return a value from a function, it immediately stops being executed. Any code after the *return* statement will never happen.

4.2 Functions as Objects

Although they are created differently from normal variables, *functions* are just like any other kind of value. They can be assigned and reassigned to variables, and later referenced by those names [6].

Example 4.6

Create a function that takes two values and produce sum and product of those values

Code:

```
# Example 4.6

def This_is_a_really_long_function_name(x,y):
    x_plus_y = x+y
    x_times_y = x*y
    return(x_plus_y, x_times_y)
```

Output:

```
In [79]: from Example4_6 import This_is_a_really_long_function_name

In [80]: a = This_is_a_really_long_function_name

In [81]: print(a(2,3))
(5, 6)
```

Observation:

Function name is assigned to a variable *a* on *line 80* and then function is called by the new name *a*

Functions can also be used as arguments of other functions [6].

Example 4.7

Create a function that calls another function as its argument

Code:

```
1 #   Example 4.7
2 #   This function will call another function as its input argument
3
4 def Ex4_7(func, x):
5     #   this will evaluate the function given by func at value 'x' and
6     #   then evaluate the result again for func
7     a = func(func(x))
8     return a
```

Output:

```
In [82]: import Example4_1, Example4_7

In [83]: b = Example4_7.Ex4_7(Example4_1.Ex4_1,3)

In [84]: print(b)
2019
```

Observation:

When function *Ex4_7* is called with function *Ex4_1* as its argument, first the quadratic equation defined by *Ex4_1* is evaluated at $x = 3$ to get a result of 31, and then it is evaluated again for 31 to get the final result of 2019.

4.3 Recursive Functions

Recursive function is the one that uses itself inside its body, i.e. it calls itself from its body.

Example 4.8

Create a function to calculate the factorial of a number *x*, *x!*

$x! = x \times (x-1) \times (x-2) \times (x-3) \times \cdots \times 1$, which can also be represented as,

$x! = x \times (x-1)!$, with 0! equals to zero

Code:

```
#   Example 4.8

#   A function to calculate factorial of its argument

def Ex4_8(x):
    if x == 1:
        return 1
    else:
        return x*Ex4_8(x-1)
```

Output:

```
In [3]: from Example4_8 import Ex4_8

In [4]: print(Ex4_8(10))
3628800
```

For every recursive function, there is a *base* condition that determines the last time function is going to call itself. In *example 4_8*, this base condition is the *if* condition, i.e. when *x* is 1, returns 1. If a *base* condition is not there, recursive functions will keep on calling themselves and will get stuck in an infinite loop.

Recursion can also be indirect. One function can call a second, which calls the first, which calls the second, and so on. This can occur with any number of functions.

Example 4.9

Create a function that calculates result of an expression, x^2+2x+3. If result is less than 100, call another function with the result as input argument to evaluate another expression, $5x+4$. If result is less than 100, it will call the first function again with result as its input argument and so on.

Code:

```
#   Example 4.9

#   Two functions will call each other until it reaches to the base condition

def f1(x):
    if x >= 100:
        return x
    else:
        x = x**2+2*x+3
        return f2(x)

def f2(x):
    if x >= 100:
        return x
    else:
        x = 5*x+4
        return f1(x)
```

Output:

```
In [5]: from Example4_9 import f1

In [6]: print(f1(2))
3602
```

Output Explained:

```
In [8]: x = 2

In [9]: x = x**2+2*x+3

In [10]: print(x)
11

In [11]: x = 5*x+4

In [12]: print(x)
59

In [13]: x = x**2+2*x+3

In [14]: print(x)
3602
```

4.4 Lambdas

Lambdas are *anonymous* functions. They are not defined in a regular fashion and they don't have any name. They are defined on the fly and generally used as an input argument of another function. Generally, they are defined by expressions to be evaluated at a value. They are defined in a single line with the following syntax:

lambda x:f(x)

If this function is called for an argument *x*, it can be done as follows:

f = (lambda x:f(x))(x)

where result of the function will be stored in the variable *f*.

```
In [22]: x = (lambda x: 2*x+4) (5)

In [23]: print(x)
14
```

Although, *lambda* functions are anonymous but they can be assigned to a variable and then that variable can be treated as a function that can be evaluated at any argument

```
In [24]: f = lambda x: 2*x+4

In [25]: print(f(10))
24
```

Example 4.10

Create a function that takes an expression as its input argument and evaluates that and square of that expression at a number, also entered as an input argument. The function should return both results, i.e. result of evaluation of the expression and its square.

Code:

```
#   Example 4.10
#

def Ex4_10(f,x):
    return (f(x), (f(x))**2)
```

Output:

```
In [26]: from Example4_10 import Ex4_10

In [27]: q = lambda w: 2*w**2 + 5*w + 6

In [28]: Ex4_10(q, 3)
Out[28]: (39, 1521)
```

Lambdas can also be used with multiple variables

```
In [48]: (lambda x,y,z: 2*x**y + x*y*z)(1,2,3)
Out[48]: 8
```

4.5 ***eval()* Function**

eval() function can be used to evaluate the result of a mathematical expression in Python. The mathematical expression should be entered as a string in the *eval* function.

```
In [56]: eval('2+2')
Out[56]: 4

In [57]: x = 23

In [58]: eval('x + 3')
Out[58]: 26


In [60]: x = 2; y = 3; z = 4

In [61]: from math import sin, cos, sqrt


In [64]: eval('2*x*y + cos(z*y) + sqrt(2*x*sin(y))')
Out[64]: 13.595172820627672
```

Exercises

4.1 Write a function that takes four numbers as input arguments: *A*, *B*, *C*, and *x*. The function should evaluate a quadratic equation Ax^2+Bx+C and returns the result. Check your function from within the program as shown in *example 4_1*.

4.2 *Second-Order Control Systems:* Write a function to evaluate the response of a second-order control system as given by the following equation:

$$y(t,\varsigma) = 1 - \frac{1}{\sqrt{1-\zeta^2}} e^{-\zeta t} \sin(\sqrt{1-\zeta^2}\, t + \cos^{-1}(\zeta))$$

Input argument *t* is a list with time values and damping coefficient ζ is a floating-point, value of which is between 0 and 1; $0 < \zeta < 1$. Return your result *y*, which will be a list. Check your function from within the program.

4.3 Create a program with two functions, one to calculate the series equivalent of resistors and the other for parallel equivalent. Input argument of the functions is an array of resistor values entered either a list, tuple, or a set. Each function should return the equivalent of resistors as an integer or floating point. Check your function from within the program.

4.4 Create a function that takes an integer as its argument and adds all the previous integers until zero into the argument value. For example, *f(4)* = 4+3+2+1+0. Use recursive function concepts to write this function. Check your function from within the program.

4.5 Create a function that takes a mathematical expression and a list to evaluate the expression, as input arguments, and returns the evaluated values of the expression. Check your function from within the program

4.6 Write a function to generate Fibonacci numbers up to an input number *x*. Fibonacci numbers is a series of numbers that starts at 1 and the next number is the sum of two previous numbers. Check your function from within the program. Fibonacci numbers: 1, 1, 2, 3, 5, 8, …..

Chapter 5

Files

Like in any other programming language, files can be read, written and appended from a python code.

5.1 File Input and Output

The syntax to open a file is as follows:

f = open(file name as a string) or

f = open(file name as a string, operation to be done on file)

The default operation when a file is opened is the file *read* operation. *Table 5.1* shows different operations that can be carried out with file opening, and their codes [4].

Table 5.1 – File Opening Options

Option	*Explanation*
r	Opens a file for reading only (default option)
r+	Opens a file for both reading and writing
w	Opens a file for writing only; overwrites if file exists, creates one if it doesn't.
w+	Opens a file for both writing and reading
a	Opens a file for appending

Hence, to open a file to read and then write, the syntax will be:

f = open(file name as a string, 'r+')

A *'b'* appended to the option makes the file opened for reading, writing or appending for binary data. For example, if an image file is opened to be read that is saved in binary, the syntax will be:

f = open(file name as a string, 'rb')

Once operations on a file is done, it needs to be closed using the following syntax:

f.close()

Example 5.1

Open a text file and read and print the text written into it

Code:

```
#   Example 5.1 - Reading a file

f = open('Ex5_1.txt')
f_contents = f.read()   #   read method is used to read contents of file
print('\n File contents are: ', f_contents)
print('\n Number of characters in the Opened file are {}'.format(len(f_contents)))
f.close()
```

Output:

```
File contents are:  CET 4370C - Advanced Programming Applications

Number of characters in the Opened file are 45
```

Make sure that file is saved in the same folder where the program is saved that is calling the file. If file is another folder, full path to the file is required to open it:

```
f = open('C:/Users/mejaz/Teaching/Valencia/Fall 2018/Ex5_1.txt')
```

Example 5.2

Open the text file *Ex5_2*, read its contents and print them out. Then skip the first 20 characters and read the next 10 characters.

Code:

```
#   Example 5.2

f = open('Ex5_2.txt','r')
f_contents = f.read()   #   Reading the contents of file
print('\nContents of the file are: \n\n', f_contents)

#   skipping first 20 characters
print('\n\nSkipping first 20 characters and reading the next 10 characters of the file:\n')
print(f_contents[20:31])

#   seek method can be used to skip file characters as well
print('\n\nSkipping first 20 characters using seek method and reading the next 10 characters of the file:\n')
f.seek(20)  #    go 'n' charcaters forward with seek method
print(f.read(11))   #   read and print out the next 10 characters

f.close()
```

Output:

```
Contents of the file are:

 CET 4370C - Advanced Programming Applications
 Tuesdays - 9:00AM - 10:45AM
 West Campus Room 11-243


Skipping first 20 characters and reading the next 10 characters of the file:

 Programmin


Skipping first 20 characters using seek method and reading the next 10 characters of the file:

 Programmin
```

Example 5.3:

Open text file *Ex5_2* and append it with 'Valencia College' in a new line

Code:

```
#   Example 5.3 - Append

f = open('Ex5_2.txt','a')
Text = '\n Valencia College'
f.write(Text)   #   writing text to the file
f.close()

f = open('Ex5_2.txt','r')
print('\n New Contents of the file are: \n',f.read())
f.close()
```

Output:

```
 New Contents of the file are:
 CET 4370C - Advanced Programming Applications
 Tuesdays - 9:00AM - 10:45AM
 West Campus Room 11-243
 Valencia College
```

If a line needs to be inserted somewhere in the text file, first the file needs to be read as a list with *f.readlines()* method, then text is inserted using *insert* method for lists, and then it needs to be written back as a list with *f.writelines()* method.

Example 5.4

Insert 'Department of Electrical and Computer Engineering Technology' before 'Valencia College' in the file *Ex5_2.txt*

Code:

```
#   Example 5.4 - Inserting text in a file

f = open('Ex5_2.txt','r+')
Text = ' Department of Electrical and Computer Engineering Technology\n'
f_contents = f.readlines()   #   reading contents of file per line as a list; each line refers to one item in the list

f_contents.insert(3, Text)   #   inserting text to the file on the fourth line; inserting an element on list[3]

f.seek(0)   #   beginning of file
f.writelines(f_contents) #   writing new contents to the file with text inserted as a list

f.close()

f = open('Ex5_2.txt','r')
print('\nThe new contents of the file are: \n',f.read())
f.close()
```

Output:

```
The new contents of the file are:
 CET 4370C - Advanced Programming Applications
 Tuesdays - 9:00AM - 10:45AM
 West Campus Room 11-243
 Department of Electrical and Computer Engineering Technology
 Valencia College
```

When a file is read with *readlines()* as a list, other list functions and methods can be applied to modify the contents of the file and then *writelines()* can be used to write the modified file back

Example 5.5

Remove the text inserted in the file *Ex5_2.txt* in *Example 5.4* and save it back into the same file

Code:

```
#   Example 5.5 - Removing text from a file

f = open('Ex5_2.txt','r')
#Text = ' Department of Electrical and Computer Engineering Technology\n'
f_contents = f.readlines()   #   reading contents of file per line as a list

del f_contents[3]   #   deleting fourth line of text (fourth element of the list)
f.close()

f = open('Ex5_2.txt', 'w')
f.writelines(f_contents) #   writing new contents to the file; it will erase the existing contents
f.close()

f = open('Ex5_2.txt','r')
print('\nThe new contents of the file are: \n',f.read())
f.close()
```

Output:

```
The new contents of the file are:
 CET 4370C - Advanced Programming Applications
 Tuesdays - 9:00AM - 10:45AM
 West Campus Room 11-243
 Valencia College
```

There are multiple ways to utilize numbers from a file. One of the ways is shown in *Example 5.6*

Example 5.6

Evaluate an expression to create a number list. Save this list in a file. Open the file to read numbers and then use them to evaluate an expression

Code:

```
#   Example 5.6
#   Writing data points in a file and then using them to evaluate some expression

A = [x**2 + 4.6*x + 6.8 for x in range(10)] #   Evalauting expression to generate result as a list
f = open('Ex5_6.txt', 'w')  #   opening a file to write
for k in A:
    f.write('%s\n' %k)  #   writing each floating number in a new line as a string
f.close


q = open('Ex5_6.txt','r')   #   opening file to read numbers
B = q.readlines()   #   converting each number as a string in a list
T = [B[0].rstrip('\n')] #   removing '\n' character from the first string in the list using 'rstrip' function
for k in range(1,len(B)):
    T.append(B[k].rstrip('\n')) #   removing rest of '\n' characters from rest of the strings and creating a li

#   Converting all strings in the list to floating points
V = [float(x) for x in T]
#   Evalauting expression
U = [2*x**3 + 5*x for x in V]
print(U)
q.close()
```

Output:

```
[662.8639999999999, 3875.2479999999987, 16100.0,
52016.67199999999, 140075.05599999995, 329407.18399999995,
698179.3279999997, 1363383.9999999993, 2492071.9519999996,
4314024.176000002]
```

Exercises

5.1 Create a file with the following text through your program:

CET 4370 – Advanced Programming Applications

Fall 2018

ECET Department

Valencia College

Orlando, Florida

Open the file and print its contents.

Now, insert *West Campus* between *Valencia College* and *Orlando, Florida*. Print out the new contents.

Now, append the file with *University Center, Building 11*. Print out the new contents

Now, delete *Fall 2018* from the file. Print out the new contents.

5.2 Open the provided data file *Exer5_2*.txt and evaluate $2x^2+5x+8cos(sin(x))$. Write your results in another file named *Your First Name_Last Name_Ex5_2*.

Chapter 6

Object-Oriented Programming

Although, we have used terms *objects*, *classes*, and *methods* in conjunction with different variables and functions, the programming approach that has been used so far is *Procedure-Oriented Programming (POP)*. In procedure oriented programming, *procedures*, also called *routines*, subroutines or *functions*, simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

While the focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, *Object-Oriented Programming* (OOP) breaks down a programming task into *objects* that expose behavior (*methods*) and data (*members or attributes*) using interfaces. The most important distinction is that while procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together, so an "object", which is an *instance* of a *class*, operates on its "own" data structure [17].

Classes and *objects* are the most important concepts in OOP. As mentioned earlier, an object is a special instantiation of a class. After a class is defined with *methods* (which are usually user-defined function) and the *variables* in it, then an object or multiple objects can be created referring to the same class [4].

6.1 Creating Classes

The syntax to create a class is,

class Name of Class:

body of class

Body of class contains methods, objects, and variables

Example 6.1

Define a class *Car* and define a method with attributes (variables) to identify *make*, *model*, and *color* of the car. Then define few objects in the class

Code:

```
#   Example 6.1 - Object-Oriented Programming
#   A class 'Car' is defined in this example with methods make, model, and color of the car
#   Some objects will be defined in the class

class Car:
    def __init__(self, make, model, color):         #   function to define method in class Car
                                                    #   with attributes make, model, and color
        self.mk = make                              #   class instance variable 'mk'
        self.md = model                             #   class instance variable 'md'
        self.co = color                             #   class instance variable 'co'

car1 = Car('Lamborghini', 'Aventador', 'Green')     #   Object # 1 in the class Car
car2 = Car('Mercedes', 'GT', 'Black')               #   Object # 2 in the class Car
car3 = Car('Volkswagen', 'Beetle', 'Yellow')        #   Object # 3 in the class Car

#   ----------------------------------------------------------------------
#     printing object in the class with different method attributes
print(car1.mk)
print(car2.md)
print(car3.co)
```

Output:

```
Lamborghini
GT
Yellow
```

The *__init__* method is the most important method in a class. It is called class *constructor*. This method is called when an instance (object) of the class is created, using the class name as a function. All methods must have *self* as their first parameter, although it isn't explicitly passed. Python adds the *self* argument to the list when a method is called. Within a method definition, *self* refers to the instance calling the method [6].

Instances of a class have *attributes*, which are pieces of data associated with them. In this example, *Car* instance has attributes *make*, *model*, and *color*. These can be accessed by putting a *dot*, and the variable name after an instance.

Note that in *Example 6.1*, instance variable name is different from attribute name but it can be defined to be the same. For example, *self.model = model*. This is a more common practice to keep instance variable name to be the same as attribute name.

If some specific object is to be used from a class, the object needs to be imported first from the class *module*.

```
In [6]: from Example6_1 import car1, car2, car3

In [7]: print(car2.md)
GT

In [8]: print(car1.mk)
Lamborghini

In [9]: print(car3.md)
Beetle
```

A class may have other methods defined to add more functionality. An example is shown as follows.

Example 6.2

Add a method *year* to the class *Car* from *Example 6.1* that determines model year of the car

Code:

```
#   Example 6.2 - Object-Oriented Programming
#   A class 'Car' is defined in this example with methods make, model, and color of the car
#   Some objects will be defined in the class

class Car:
    def __init__(self, make, model, color):         #   function to define method in class Car
                                                    #   with attributes make, model, and color
        self.mk = make                              #   class instance variable 'mk'
        self.md = model                             #   class instance variable 'md'
        self.co = color                             #   class instance variable 'co'

    def year(self):                                 #   Defining another method 'year' in the class car
        if self.mk == 'Lamborghini':
            print('2012')
        elif self.mk == 'Mercedes':
            print('2000')
        elif self.mk == 'Volkswagen':
            print('2003')



car1 = Car('Lamborghini', 'Aventador', 'Green')     #   car1 is the object or function in the class Car
car2 = Car('Mercedes', 'GT', 'Black')               #   car2 is the object or function in the class Car
car3 = Car('Volkswagen', 'Beetle', 'Yellow')        #   car3 is the object or function in the class Car

#   ---------------------------------------------------------------------
#     printing object in the class with different method attributes
print(car1.mk)
print(car2.md)
print(car3.co)

#   Calling method year for different cars
car1.year()
car2.year()
car3.year()
```

Output:

```
Lamborghini
GT
Yellow
2012
2000
2003
```

Observe that the other method (function) defined in the above example is without __init__.

Example 6.3

Define a class *Card*, which is about picking up a card from a deck of 52 cards. Name of the user should be passed to the __init__ function of the class. The code should print out name of the player and the card that is randomly drawn for three different players [4]

Code:

```
#   Example 6_3

class Card:

    SUITS = {1: 'Clubs', 2: 'Hearts', 3: 'Spades', 4: 'Diamonds'}   #   defining a dictionary with each shape in the pack
    VALUES = {1:'Ace', 2: '2', 3: '3', 4: '4', 5: '5', 6: '6', 7: '7', 8: '8', 9: '9', 10: '10', 11: 'Jack', 12: 'Queen', 13: 'King'}

    def __init__(self, name):
        from random import randint
        self.v = randint(1,13)  #   generating a random integer from 1 to 13
        self.s = randint(1,4)

        print('{}, you have {} of {}'.format(name, Card.VALUES[self.v], Card.SUITS[self.s]))

card1 = Card('Player 1')
card2 = Card('Player 2')
card3 = Card('Player 3')
```

Output:

```
Player 1, you have 9 of Hearts
Player 2, you have 4 of Diamonds
Player 3, you have 5 of Spades
```

Example 6.4

Write a program that has a class named *IsEven*. A number should be passed to the class, and the code should print 'YES' or 'NO' based on if the input number is even or not.

Code:

```
#   Exercise 6.3
#   A class that prints Yes if the passed number is prime, else print No

class IsEven:
    def __init__(self,x):#   x is the number passed to the class
        self.x = x
        if x%2 == 0:   #   checking remainder with modulo function
            print('Number is Even')
        else:
            print('Number is Odd')

num1 = IsEven(3)
num2 = IsEven(4)
```

Output:

```
Number is Odd
Number is Even
```

6.2 Class Inheritance

Inheritance provides a way to share functionality between classes. If there are some similarities between different classes, a *superclass* can be created that can share its properties with other classes. To inherit a class from another class, the *superclass* name is place in parentheses after the class name. A class that inherits from another class is called *subclass* [6], [18].

Example 6.5

Define a subclass on the class from *Example 1*, Car

Code:

```
#   Example 6.5 - Object-Oriented Programming - Inheritance
#   A class 'Car' is defined in this example with methods make, model, and color of the car
#   Car is a superclass
#   A subclass will be defined that will inherit its attributes from Car

class Car:
    def __init__(self, make, model, color):         #   function to define method in class Car
                                                    #   with attributes make, model, and color
        self.mk = make                              #   class instance variable 'mk'
        self.md = model                             #   class instance variable 'md'
        self.co = color                             #   class instance variable 'co'

class Wheels(Car):  #   Defining a subclass of Car; How many Wheels for the Car
    def Three(self):    #   A function to define cas with three wheels
        print('{} has a car called {} with three wheels'.format(self.mk, self.md))

Morgan = Wheels('Morgan', '3-Wheeler', 'Black')
Morgan.Three()
```

Output:

```
Morgan has a car called 3-Wheeler with three wheels
```

Class inheritance can also be *indirect*, where one class can inherit from another and that class can inherit from a third class. This is also called *multilevel* inheritance.

Example 6.6

Create a superclass *Vehicle* that defines type of a vehicle. Define a subclass *Car* under Vehicle that defines make, model, and color of the car. Finally, define a subclass under Car, *Mechanics*, which defines the engine and transmission characteristics of the car.

Code:

```
'''   Example 6.6 - Object-Oriented Programming - Indirect or multilevel Inheritance
A class 'Vehicle' is defined with method vehicle type
A class 'Car' is inherited from class Vehicle with methods make, model, and color of the car
A class 'Mechanics' will be defined that will inherit its attributes from Car'''

class Vehicle:
    def __init__(self, vtype):                          #   function to define method in class Vehicle
        self.vtype = vtype
        print('\n Type of the vehicle is {}'.format(vtype))    #   This will be printed out when last subclass will be called

class Car(Vehicle):
    def __init__(self, vtype, make, model, color):   #   function to define method in class Car
                                                     #   with attributes make, model, and color
        Vehicle.__init__(self,vtype)
        self.mk = make                               #   class instance variable 'mk'
        self.md = model                              #   class instance variable 'md'
        self.co = color                              #   class instance variable 'co'
        print(' Make of the car is {}\n Model of the car is {}\n Color of the car is {}'.format(make,model,color))

class Mechanics(Car):                                #   Defining a subclass of Car; mechanical aspects of car
    def __init__(self, vtype, make, model, color, engine, transmission):          #   A function to define engine and transmission of the car

        Car.__init__(self, vtype, make, model, color)
        self.engine = engine
        self.transmission = transmission

    def specs(self):
        print(' Car has {} engine mated with {} transmission'.format(self.engine, self.transmission))

    def pic(self,URL):                                       #   function to print out the picture of the car
        self.URL = URL
        from Example4_5 import Ex4_5                 #   Calling function to print out picture of the vehicle from a URL
        Ex4_5(URL)

Car1 = Mechanics('5-door SUV','Lamborghini', 'Urus', 'yellow','4.0 L FSI twin-turbocharged V8','8-speed ZF 8HP automatic')
Car1.specs()
Car1.pic('https://cdn.motor1.com/images/mgl/GRBZE/s1/2018-lamborghini-urus.jpg')
```

Output:

```
 Type of the vehicle is 5-door SUV
 Make of the car is Lamborghini
 Model of the car is Urus
 Color of the car is yellow
 Car has 4.0 L FSI twin-turbocharged V8 engine mated with 8-
speed ZF 8HP automatic transmission
```

Observations:

Observe that *Car1* is an object defined in the subclass *Mechanics*. Two methods are defined under this class; *specs*, to print out mechanical specifications of the car, and *pic*, to print out picture of the car from its URL, which is calling function *Ex4_5*, as discussed in *chapter 4*. When the two methods are executed from this class, first the superclass *Car* is called, which in turns calls the superclass *Vehicle*. Output of the program is the execution of superclass *Vehicle*, followed by subclass (superclass for *Mechanics) Car*, followed by subclass *Mechanics*.

The function *super* is a useful inheritance-related function that refers to the parent class. It can be used to find the method with a certain name in an object's superclass. It can also be used to call a superclass constructor from a subclass constructor, since class constructor is also a method.

In *example 6.6*, superclass constructor from subclass can also be written as:

```
super().__init__(vtype)

super().__init__(vtype, make, model, color)
```

Observe that the class name is replaced by *super()* and *self* is eliminated from the list of variables.

Any function that is defined in the superclass can be accessed from its subclass by defining that function as a method on the object defined on the subclass. For example, if there is a function *A(self)* defined in the superclass, and there is a subclass *subclass_B*, then an object *C* defined on the subclass can access the function *A* as follows [21]:

```
#   Object assigment
C = subclass_B(instance variables)
#   Method A defined on object C
C.A()
```

6.3 Magic Methods

Magic methods are special methods which have *double underscores* at the beginning and end of their names. They are also known as *dunders*. So far, the only one we have encountered is *__init__*, but there are several others. They are used to create functionality that can't be represented as a normal method [6]. A list of different magic methods is given in *figure 6.1* [20].

Binary Operators	
Operator	**Method**
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Extended Assignments

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

Unary Operators

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Comparison Operators

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Figure 6.1: Overview of Magic Methods

One common use of magic methods is *operator overloading*. This means defining operators for custom classes that allow operators such as + and * to be used on them. An example magic method is __add__ for +, as mentioned in *figure 6.1*. The mechanism of operator overloading works like this: If we have an expression "*x* + *y*" and *x* is an instance of class *K*, then Python will check the class definition of *K*. If *K* has a method __add__, it will be called with x.__add__(y), otherwise it will return an error message.

Example 6.7

Create a class to perform array-wise multiplication of two three dimensional vectors

Code:

```
#   Example 6.7 - Magic Methods

class Mul_3D_Vectors:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def __mul__(self, other):   #   magic method __mul__ is used to perform multiplication
        return Mul_3D_Vectors(self.x*other.x, self.y*other.y, self.z*other.z)

#   Defining objects on class
Vector1 = Mul_3D_Vectors(2, 5, 6)
Vector2 = Mul_3D_Vectors(8, 3, 2)

#   Test
Vector1_times_Vector2 = Vector1*Vector2
print(Vector1_times_Vector2.x)
print(Vector1_times_Vector2.y)
print(Vector1_times_Vector2.z)

#   creating a vector from multiplication result
Vector1_times_Vector2 = [Vector1_times_Vector2.x, Vector1_times_Vector2.y, Vector1_times_Vector2.z]
print(Vector1_times_Vector2)
```

Output:

```
16
15
12
[16, 15, 12]
```

6.4 Hidden Methods and Variables

Hidden methods are defined with two underscores "__" at the beginning of the method name. Hidden methods cannot be defined directly on an object defined on the class:

```
#   Object defined on ClassB
A = classB(variables)

#   Method allowed from classB to be defined on object A
A.AllowedMethod()

#   Method not allowed from classB to be defined on an object A
A.__NotAllowedMethod()
```

Likewise, variable names that start with double underscores can only be modified from within the class. These are hidden variables and cannot be modified from outside the class where they are defined.

Example 6.8

Define a class with both hidden and unhidden class instances.

Code:

```
1 #   Example 6.8 - Data hiding
2
3 class A:
4     def __init__(self, x, y):
5         self.x = x
6         self.y = y
7
8     def x_power_y(self):     #   Unhidden function
9         return self.x**self.y
10
11    def __x_power_y(self):  #   Hidden function
12        return self.x**self.y
13
14 #   Object definition
15
16 B = A(2,3)
17 print(B.x_power_y())
18 #print(B.__x_power_y())
```

Output:

Defining unhidden method on object B, as shown in line 17

```
In [33]: runfile('C:/Users/ME/Taleem/Valencia/Fall 2018/Advanced Programming/Scripts/
Example6_8.py', wdir='C:/Users/ME/Taleem/Valencia/Fall 2018/Advanced Programming/Scripts')
8
```

Removing # from line 18 to define hidden method on object B

```
  File "C:/Users/ME/Taleem/Valencia/Fall 2018/Advanced Programming/Scripts/Example6_8.py", line 18, in
<module>
    print(B.__x_power_y())

AttributeError: 'A' object has no attribute '__x_power_y'
```

6.5 Class and Static Methods

Methods of objects discussed so far are called by an instance of a class, which is then passed to the *self* parameter of the method. *Class methods* are different; they are called by a class, which is passed to the *cls* parameter of the method. A common use of class methods are factory methods, which instantiate an instance of a class, using different parameters than those usually passed to the class constructor. Class methods are marked with a *classmethod decorator*, @classmethod [6].

Example 6.9

Create a class to measure the area of a rectangle based on the dimension of its sides. Now, define a class method to calculate the area if all sides of the rectangle have same dimensions, i.e. a square

Code:

```
#   Example 6.9 - Example of using class methods

class Rectangle:
    def __init__(self, x, y):   #   defining instantiation method of the class
        self.x = x
        self.y = y

    def area(self):             #   defining an area method that takes on values from the class instatiation
        return self.x*self.y

    @classmethod                #   defining a class method
    def square(cls,side):       #   observe that the instance variable is different
        return cls(side,side)   #   class instance will be returned with both x and y to be same for a square

#   defining an object on the class
rect1 = Rectangle(4,5)
print(rect1.area())             #   Printing method area on object rect1

#   defining class method square on the object rect2
rect2 = Rectangle.square(9)
print(rect2.area())              #   Printing method area on object rect2
```

Output:

```
20
81
```

A static method is marked with a decorator *@staticmethod*. It is different from an instance method or a class method in the sense that it does not have any class constructor (cls) or instance constructor (self). Rather, it is just any function defined within a class. A static method can't modify or access class state. They are generally used to construct utility functions.

Example 6.10

Insert a static method in *example 6.9* to calculate area of a square

Code:

```
#   Example 6.10 - Example of using Static methods

class Rectangle:
    def __init__(self, x, y):   #   defining instantiation method of the class
        self.x = x
        self.y = y

    def area(self):             #   defining an area method that takes on values from the cl
        return self.x*self.y

    @classmethod                #   defining a class method
    def square(cls,side):       #   observe that the instance variable is different
        return cls(side,side)   #   class instance will be returned with both x and y to be

    @staticmethod               #   Defining a static method
    def area_of_square(x):
        return x**2

#   defining an object on the class
rect1 = Rectangle(4,5)
print(rect1.area())             #   Printing method area on object rect1

#   defining class method square on the object rect2
rect2 = Rectangle.square(9)
print(rect2.area())              #   Printing method area on object rect2

rect3 = Rectangle.area_of_square(5)
print(rect3)
```

Output:

```
20
81
25
```

Exercises

6.1 Create a class *Student_Schedule* with method attributes to be *student name*, *semester*, and *course*. Define three objects in the class for three different students.

6.2 Write a program that has a class named *Lottery*. A name should be passed to the class where the class should randomly select six numbers (integers from 0 to 9) and print the name and picked numbers.

6.3 Write a program that has a class named *IsPrime*. A number should be passed to the class, and the code should print 'YES' or 'NO' based on if the input number is prime or not.

[*The easiest logic, albeit very inefficient, is to divide the given number from 2 through one less than the number. If any number will divide it completely without any remainder, the number is prime. There are other more efficient logics as well. I will leave it upon you as which one you would like to use.*]

6.4 Create a class *'Area*' to calculate area of a geometric figure from its two dimensions. Now create two subclasses of *Area*, one *Triangle* and the other *Rectangle* that will call the superclass to calculate areas of the respective geometrical figures from their dimensions.

Chapter 7

Matrix Algebra

In this chapter, a new module *Numpy* will be used, which stands for *Numerical Python*. This module is very commonly used to perform numerical and scientific computing together with *Scipy* (*Scientific Python*) module [4].

7.1 Arrays and Matrices

Numpy module has array and matrix classes. *Arrays* are indexed lists that can be single dimensional or multidimensional. The basic application of arrays is in scientific applications. *Matrices* are arrangement of numbers in two dimensions and generally used in linear algebra. Matrices are subset of arrays and can only be two-dimensional, as mentioned above, whereas arrays can be *N*-dimensional. Two important functions in *numpy* to create *n*-dimensional arrays are *linspace()* and *arange()*.

```
In [22]: # 4 equally spaced numbers between 0 and 12

In [23]: X = linspace(0, 12, 4)

In [24]: X
Out[24]: array([ 0.,  4.,  8., 12.])

In [25]: # 10 equally spaced numbers from 1 to 10

In [26]: X = linspace(1, 10, 10)

In [27]: X
Out[27]: array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

```
In [44]: # numbers from 0 to 10 with interval of 2

In [45]: X = arange(0, 11, 2)

In [46]: X
Out[46]: array([ 0,  2,  4,  6,  8, 10])

In [47]: # numbers from 10 to 11 with interval of 0.1

In [48]: X = arange(10, 11.1, 0.1)

In [49]: X
Out[49]: array([10. , 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9, 11. ])
```

Any scientific or mathematical function can be operated on arrays but make sure to import functions from *numpy* instead of *math* or other modules;

```
In [67]: from numpy import sin, cos, pi, sqrt

In [68]: print(sin(X))
[-0.54402111 -0.62507065 -0.69987469 -0.76768581 -0.82782647 -0.87969576
 -0.92277542 -0.95663502 -0.98093623 -0.99543625 -0.99999021]

In [69]: print(2*X**2 + 5*X + 8)
[258.   262.52 267.08 271.68 276.32 281.   285.72 290.48 295.28 300.12
 305.  ]
```

numpy also has *array* and *matrix* functions to define *N*-dimensional arrays and matrices;

```
In [70]: from numpy import array, matrix

In [71]: A = array([[1,2,3],[5,6,7],[3,8,9.2],[23,6,7]])

In [72]: A
Out[72]:
array([[ 1. ,  2. ,  3. ],
       [ 5. ,  6. ,  7. ],
       [ 3. ,  8. ,  9.2],
       [23. ,  6. ,  7. ]])
```

len() function can be used to find out the largest dimension of an array. To find total number of elements in an array, *size* method can be used. Also, to find dimensions of an array, *shape* method can be used.

```
In [76]: print(A.shape)
(4, 3)

In [77]: print(A.size)
12
```

Any specific element in an array can be accessed from its indices. Remember, indices start at zero;

```
In [84]: A[3][2]
Out[84]: 7.0

In [85]: A[0][0]
Out[85]: 1.0
```

To define matrices, *matrix* or *mat* function can be used;

```
In [86]: B = matrix("[1, 5, 6;8, 9, 10;5 7 8]")

In [87]: print(B)
[[ 1  5  6]
 [ 8  9 10]
 [ 5  7  8]]

In [88]: print(B.size)
9

In [89]: print(B.shape)
(3, 3)


In [95]: from numpy import mat

In [96]: D = mat('1 2 3;4 5 6')

In [97]: print(D)
[[1 2 3]
 [4 5 6]]
```

Similar to arrays, all mathematical and scientific operations can be carried over matrices as well.

An array with dimensions more than two can also be created easily;

```
In [117]: W = array([[[1,2],[4,5]],[[7,8],[9,6]], [[4,5],[9,2]]])

In [118]: print(W.shape)
(3, 2, 2)

In [119]: W
Out[119]:
array([[[1, 2],
        [4, 5]],

       [[7, 8],
        [9, 6]],

       [[4, 5],
        [9, 2]]])

In [120]: print(W[1][1][0])
9

In [121]: print(W[2][1][1])
2
```

To insert a new column or row in a matrix or array, *insert()* function can be used [23];

```
In [55]: B = array([[1, 4, 5], [23, 98, 12], [20, 7, 62]])

In [56]: B = insert(B, 3, [[2,41,62]], 1)

In [57]: B
Out[57]:
array([[ 1,  4,  5,  2],
       [23, 98, 12, 41],
       [20,  7, 62, 62]])
```

```
In [36]: C
Out[36]:
matrix([[ 1,  7,  8],
        [ 2,  3,  7],
        [12, 65,  3],
        [12, 45, 63]])

In [37]: C = insert(C, 3, [2, 41, 62], 0)

In [38]: print(C)
[[ 1  7  8]
 [ 2  3  7]
 [12 65  3]
 [ 2 41 62]
 [12 45 63]]
```

Note that the last parameter in the *insert* function is ‘1’ or ‘0’. ‘0’ represents row insert and ‘1’ represents column insert.

If a matrix or an array needs to be appended, *append()* function can be used from *numpy*.

```
In [2]: A = matrix('[1 6 5;9 8 7;3 4 7;2 4 7]')

In [3]: A
Out[3]:
matrix([[1, 6, 5],
        [9, 8, 7],
        [3, 4, 7],
        [2, 4, 7]])
```

```
In [7]: A = append(A,[[23, 45, 67]],0)

In [8]: A
Out[8]:
matrix([[ 1,  6,  5],
        [ 9,  8,  7],
        [ 3,  4,  7],
        [ 2,  4,  7],
        [23, 45, 67]])
```

To delete a row or column from a matrix, *delete()* function can be used from *numpy*.

```
In [8]: A
Out[8]:
matrix([[ 1,  6,  5],
        [ 9,  8,  7],
        [ 3,  4,  7],
        [ 2,  4,  7],
        [23, 45, 67]])

In [9]: from numpy import delete

In [10]: A = delete(A, [1], 0)

In [11]: A
Out[11]:
matrix([[ 1,  6,  5],
        [ 3,  4,  7],
        [ 2,  4,  7],
        [23, 45, 67]])

In [12]: A = delete(A, [1], 1)

In [13]: A
Out[13]:
matrix([[ 1,  5],
        [ 3,  7],
        [ 2,  7],
        [23, 67]])
```

7.2 Special Matrices/Arrays

There are some built-in functions or methods available in *numpy* module to generate some special matrices. These matrices include *zero*, *unity*, *identity*, *diagonal*, and *empty* matrices. *Table 7.1* shows description and corresponding functions for these matrices.

Table 7.1: Special Matrices

Function	*Description*
zeros((a,b))	Creates an array of zeros with size *a* x *b*. Use *zeros(a,b, dtype = complex)* if array is required to hold complex values
ones((a,b))	Creates an array of ones with size *a* x *b*
eye(a)	Creates an identity square array of *a* x *b*
diag([a, b, c])	Creates a diagonal array with elements *a*, *b*, and *c*
empty((a,b))	Creates an empty array of *a* x *b*

7.3 Operations on Matrices

Addition and Subtraction: Addition and subtraction of matrices is done in simple way using '+' and '-' operations. Matrices being added or subtracted must have same dimensions.

```
In [4]: A = mat('1, 6, 9;4, 5, 7')

In [5]: B = mat('2.3 6.7 8.9; 6, 7, 8')

In [6]: print(A+B)
[[ 3.3 12.7 17.9]
 [10.  12.  15. ]]

In [7]: print(A-B)
[[-1.3 -0.7  0.1]
 [-2.  -2.  -1. ]]
```

Multiplication: If multiplication is carried out between two arrays, it will produce an array-wise or element-wise multiplication. If it is carried out between two matrices, it will result in matrix multiplication. Hence, number of columns of matrix *A* should be same as number of rows of matrix *B*, if *A***B* is carried out, and the resultant matrix will have same number of rows as matrix *A* and same number of columns as matrix *B*. If number of columns of matrix *A* is not equal to the number of rows of matrix *B*, their multiplication will result in an error message.

Matrix multiplication:

```
In [10]: C = mat('1,3;5.6,8;3,2')

In [11]: A
Out[11]:
matrix([[1, 6, 9],
        [4, 5, 7]])

In [12]: B
Out[12]:
matrix([[2.3, 6.7, 8.9],
        [6. , 7. , 8. ]])

In [13]: C
Out[13]:
matrix([[1. , 3. ],
        [5.6, 8. ],
        [3. , 2. ]])

In [14]: print(A*C)
[[61.6 69. ]
 [53.  66. ]]
```

```
In [15]: print(A*B)
Traceback (most recent call last):

  File "<ipython-input-15-02d234b7ecbd>", line 1, in <module>
    print(A*B)

  File "C:\Users\mejaz\AppData\Local\Continuum\anaconda3\lib\site-packages\numpy\matrixlib
\defmatrix.py", line 309, in __mul__
    return N.dot(self, asmatrix(other))

ValueError: shapes (2,3) and (2,3) not aligned: 3 (dim 1) != 2 (dim 0)
```

Array multiplication:

```
In [19]: D = np.array([[1, 2, 5], [5, 6, 9]])

In [20]: D
Out[20]:
array([[1, 2, 5],
       [5, 6, 9]])

In [21]: E = np.array([[6, 8, 3], [2, 4, 1]])

In [22]: E
Out[22]:
array([[6, 8, 3],
       [2, 4, 1]])

In [23]: print(D*E)
[[ 6 16 15]
 [10 24  9]]
```

If matrix multiplication is required with arrays, *dot()* function can be used. This function will be further explored after *division*.

Division: Both arrays and matrices perform element-wise division.

```
In [19]: A = array([[2, 4, 5], [7, 8, 9], [3, 8, 6]])

In [20]: C = array([[10, 14, 15], [7, 4, 18], [3, 18, 16]])

In [21]: B = matrix('[4, 9, 2; 8, 7, 4; 2, 4, 1]')

In [22]: D = matrix('[42, 91, 22; 82, 72, 42; 22, 42, 12]')

In [23]: A/C
Out[23]:
array([[0.2       , 0.28571429, 0.33333333],
       [1.        , 2.        , 0.5       ],
       [1.        , 0.44444444, 0.375     ]])

In [24]: B/D
Out[24]:
matrix([[0.0952381 , 0.0989011 , 0.09090909],
        [0.09756098, 0.09722222, 0.0952381 ],
        [0.09090909, 0.0952381 , 0.08333333]])
```

Dot Product: Dot product, also called *inner* product is the sum of the product of element-wise multiplication between the components of two vectors. In general, dot product of two vectors is a scalar product that yields a quantity without any direction. If **A** and **B** are two vectors, the dot product between them can be given by:

$$\mathbf{A} \bullet \mathbf{B} = |\mathbf{A}||\mathbf{B}|\cos\theta \tag{7.1}$$

where θ is the angle between the two vectors. As mentioned earlier, dot product between vectors (arrays) in Python is carried out using *dot* function from *numpy*.

```
In [54]: A = array([1, 5, 9])

In [55]: B = array([2, 3, 4])

In [56]: print(dot(A,B))
53

In [57]: print(dot(B,A))
53
```

Cross Product: Cross product between two vectors results in another vector which is perpendicular to both the vectors on which cross product is applied. Mathematically,

$$\mathbf{A} \times \mathbf{B} = |\mathbf{A}||\mathbf{B}|\sin\theta\ \hat{\mathbf{r}} \tag{7.2}$$

where $\hat{\mathbf{r}}$ is a unit vector perpendicular to both **A** and **B** vectors. In Python, function *cross* can be used from *numpy* to carry out cross product.

```
In [59]: from numpy import cross

In [60]: print(cross(A,B))
[-7 14 -7]

In [61]: print(cross(B,A))
[  7 -14   7]
```

Observe that unlike dot product, cross product is not a commutative operation.

Transpose: Matrix transpose is taken by the method 'T' applied on the array object.

```
In [72]: C = array([[1,5,9],[2,3,4]])

In [73]: C.T
Out[73]:
array([[1, 2],
       [5, 3],
       [9, 4]])


In [80]: C.T.shape
Out[80]: (3, 2)

In [81]: C.shape
Out[81]: (2, 3)

In [82]: D = array([[1],[2],[3]])

In [83]: D
Out[83]:
array([[1],
       [2],
       [3]])
```

There is also *transpose()* method from *numpy* module that can be used to take transpose of a matrix:

```
In [7]: A
Out[7]: array([[1, 2, 3]])

In [8]: print(A.transpose())
[[1]
 [2]
 [3]]
```

Inverse: Inverse of a matrix is calculated using 'I' method, if the object is defined on matrix class;

```
In [11]: from numpy import mat

In [12]: C = mat('1, 7, 8;2, 3, 7;12, 65, 3')

In [13]: type(C)
Out[13]: numpy.matrixlib.defmatrix.matrix

In [14]: print(C.I)
[[-0.52347418  0.58568075  0.02934272]
 [ 0.0915493  -0.10915493  0.01056338]
 [ 0.11032864  0.02230047 -0.0129108 ]]
```

```
In [15]: from numpy import matrix

In [16]: D = matrix('[12, 32, 46;76,23,87;2,13,98]')

In [17]: type(D)
Out[17]: numpy.matrixlib.defmatrix.matrix

In [18]: print(D.I)
[[-0.00638213  0.01442373 -0.00980905]
 [ 0.04133894 -0.00616049 -0.01393499]
 [-0.00535349  0.00052285  0.01225278]]
```

If object is defined on array or matrix class, inverse can also be taken using *numpy.linalg.inv()* method;

```
In [23]: B = array([[1, 4, 5], [23, 98, 12], [20, 7, 62]])

In [24]: print(numpy.linalg.inv(B))
[[-0.77346069  0.02749451  0.05705434]
 [ 0.15309152  0.00490512 -0.01329547]
 [ 0.23221892 -0.009423   -0.00077449]]
```

Determinant: Determinant of a matrix or array can be taken using *numpy.linalg.det()* method;

```
In [25]: C
Out[25]:
matrix([[ 1,  7,  8],
        [ 2,  3,  7],
        [12, 65,  3]])

In [26]: print(numpy.linalg.det(C))
852.0000000000005

In [27]: B = array([[1, 4, 5], [23, 98, 12], [20, 7, 62]])

In [28]: print(numpy.linalg.det(B))
-7747.000000000007

In [29]: print(numpy.linalg.inv(C))
[[-0.52347418  0.58568075  0.02934272]
 [ 0.0915493  -0.10915493  0.01056338]
 [ 0.11032864  0.02230047 -0.0129108 ]]
```

7.4 User Inputs

One of the ways to get user defined matrix and array is as follows:

```
a = matrix([[float(input()) for x in range (c)] for y in range(r)])

d = array([[float(input()) for x in range (c)] for y in range(r)])
```

where ‘c’ is the number of columns and ‘r’ is the number of rows. Using this syntax, each element of a matrix or an array has to be entered row-wise individually.

The above method to enter matrix elements is cumbersome as it requires the user to enter each element one by one. The easier method is as follows:

```
In [39]: A = matrix(input('Enter the matrix A: '))

Enter the matrix A: [1, 2; 3, 4]

In [40]: A
Out[40]:
matrix([[1, 2],
        [3, 4]])
```

Exercises

7.1 Evaluate $y = 2\sin(3\cos(x)) + 5x^2$ for $x = 0$ to 5π divided into 200 points. Print your results in two columns with x in the first column and the corresponding value of y in the second.

7.2 Evaluate equation from *exercise 7.1* for $x = 0$ to 5π with an interval of 0.1. Print your results in two columns with x in the first column and the corresponding value of y in the second.

7.3 Write a program to calculate angle between the two vectors in radians and in degrees.

[*Hint: Use equation (7.1)*]

7.4 Solve the following set of linear equations using matrix inversion method:

$$2x + 3y + 7z = 10$$
$$5x + 2y + 10z = 21$$
$$x + 8y + 9z = 36$$

[Note: There is also a *solve()* function in *numpy.linalg* package to solve linear equations. You can confirm your result with that method]

7.5 Solve a set of linear equations entered by the user using **matrix inversion** method. Ask user to enter the number of unknown variables from which determine the dimensions of the coefficient matrix and the column vector. Then, ask user to enter the coefficients matrix, and then ask to enter constant vector.

7.6 Solve a set of linear equations entered by the user using **Cramer's rule**. Ask user to enter the number of unknown variables from which determine the dimensions of the coefficient matrix and the column vector. Then, ask user to enter the coefficients matrix, and then ask to enter constant vector.

Chapter 8

Plots

In python, two-dimensional graphs can be plotted using *matplotlib.pyplot* and *numpy* modules. Using these modules, plots can be created with similar features as MATLAB.

8.1 Single Plots

plot() function from *matplotlib.pyplot* can be used to create two-dimensional plots. *x* and *y* values of the plot are entered as arrays, matrices or a list. *show()* method is required to display the plot. Some useful function that you can import from *matplotlib.pyplot* to create plots are *plot, xlabel, ylabel, grid, title, show, legend* etc.

Example 8.1

From a certain laboratory experiment, data points for current are obtained for specific values of voltages from a circuit. These data are given as follows:

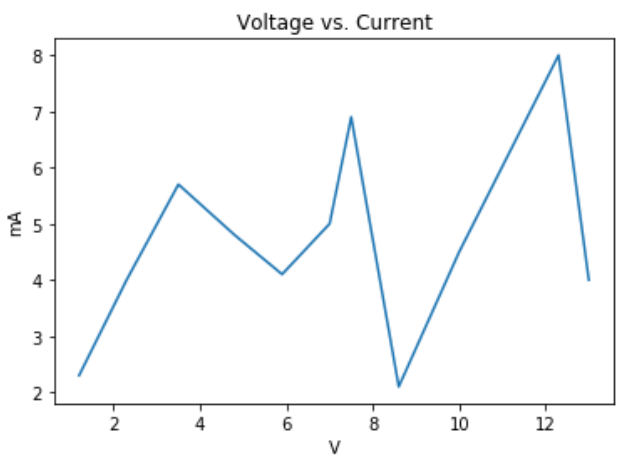
Voltage (V)	*Current (mA)*
1.2	2.3
2.3	4.0
3.5	5.7
4.8	4.8
5.9	4.1
7.0	5.0
7.5	6.9
8.6	2.1
10.0	4.5
12.3	8.0
13.0	4.0

Draw a plot between voltage and current. Properly label your axes and give a suitable title to your plot.

Code:

```
#   Example 8.1 - Single Plots from input vectors

#from numpy import array

voltage = [1.2, 2.3, 3.5, 4.8, 5.9, 7.0, 7.5, 8.6, 10, 12.3, 13]
current = [2.3, 4.0, 5.7, 4.8, 4.1, 5.0, 6.9, 2.1, 4.5, 8, 4]

import matplotlib.pyplot as plt

plt.plot(voltage,current)   #   plot between x and y
plt.xlabel('V')             #   x-label
plt.ylabel('mA')             #   y-label
plt.title('Voltage vs. Current')
plt.show()                  # show will display the plot
```

Output:



Example 8.2

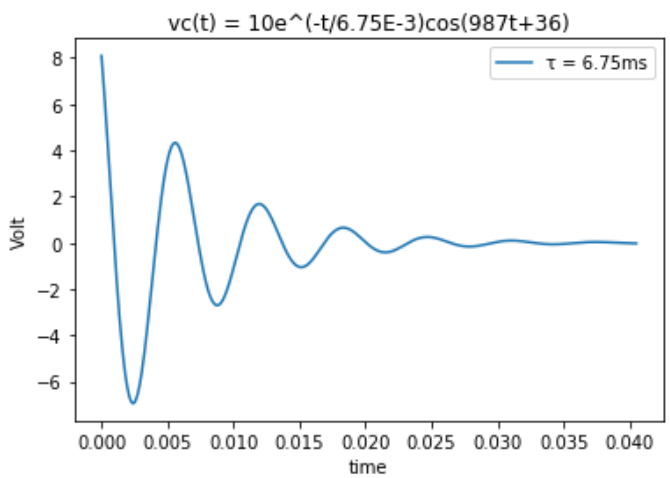
Plot a graph for the voltage across a capacitor in a parallel *RLC* circuit, as given by the following expression:

$$vc(t) = 10e^{-\frac{t}{6.75\times10^{-3}}}\cos(987t + 36.7^{\circ})\,\text{V}; \qquad t \geq 0$$

Code:

```
#   Example 8.2 - Plot of capacitor voltage in a parallel RLC circuit

from numpy import linspace, cos, exp, pi
from matplotlib.pyplot import plot, xlabel, ylabel, title, legend, show

t = linspace(0, 6*6.75e-3, 600)     #   creating an array of 600 points between zero and six times the time constant
vc = 10*exp(-t/6.75e-3)*cos(987*t+36*pi/180)    #   Evalauting voltage for the defined time array

plot(t,vc)   #   plot between x and y
xlabel('time')              #   x-label
ylabel('Volt')              #   y-label
plot_title = 'vc(t) = 10e^(-t/6.75E-3)cos(987t+36)'
title(plot_title)
legend(['\u03C4 = 6.75ms']) #   to put legend on the graph
show()                  # show will display the plot
```

Output:



Like in MATLAB, plots can be created with different colors and styles. Axes' range can also be defined, and a grid can also be turned on for the plot [4].

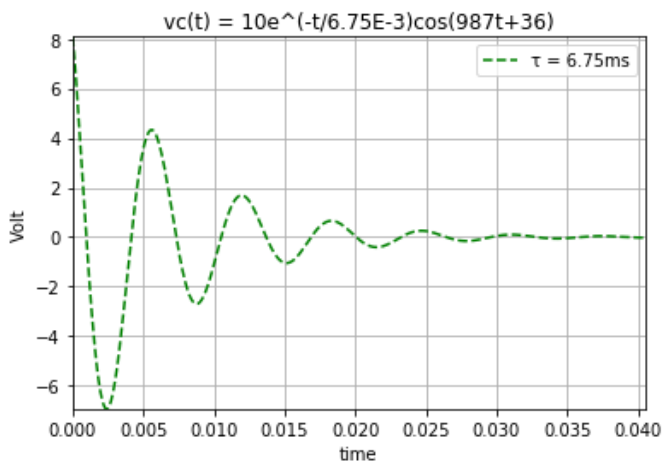
Example 8.3

Repeat *Example 8.2* and plot the function with green color and dashed line. Fix axes range properly according to the input time range and function values. Turn grid on for the graph as well.

Code:

```
#   Example 8.3 - Plot of capacitor voltage in a parallel RLC circuit

from numpy import linspace, cos, exp, pi

t = linspace(0, 6*6.75e-3, 600)     #   creating an array of 600 points between zero and six
vc = 10*exp(-t/6.75e-3)*cos(987*t+36*pi/180)    #   Evalauting voltage for the defined time

import matplotlib.pyplot as plt

plt.plot(t,vc, 'g--')   #   plot between x and y with blue color and dashed line
plt.xlabel('time')              #   x-label
plt.ylabel('Volt')              #   y-label
plot_title = 'vc(t) = 10e^(-t/6.75E-3)cos(987t+36)'
plt.title(plot_title)
plt.axis([0, 6*6.75e-3, min(vc), max(vc)])  #   axes setting
plt.legend(['\u03C4 = 6.75ms']) #   to put legend on the graph
plt.grid(True)
plt.show()                  # show will display the plot
```

Output:



Different styles and colors that are acceptable in plotting are shown in *Table 8.1* and *Table 8.2* respectively [24].

Table 8.1: Acceptable Styles in Plotting

character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker
'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

Table 8.2: Acceptable Colors in Plotting

character	color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

8.2 Multiple Plots

Multiple plots in the same figure can be done exactly as it is done in MATLAB, i.e. by creating pairs of *x* and *y* for different plots inside the *plot()* function;

matplotlib.pyplot.plot(x1, y1, x2, y2, x3, y3, …., xn, yn)

Example 8.4

Plot the following two expressions in two different figures. Also, plot both of them in the same figure. Ask user to enter data points to evaluate each expression.

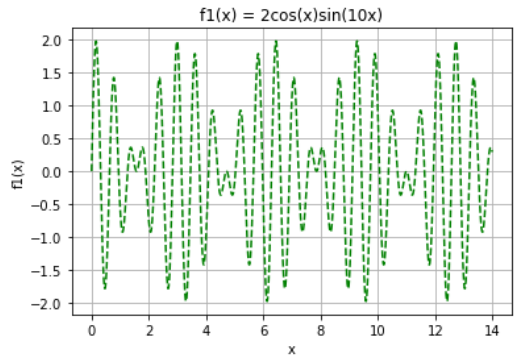
$$f_1(x) = 2\cos(x)\sin(10x)$$
$$f_2(x) = 5\sin(x) + \sqrt{|x\cos(4x)|}$$

Code:

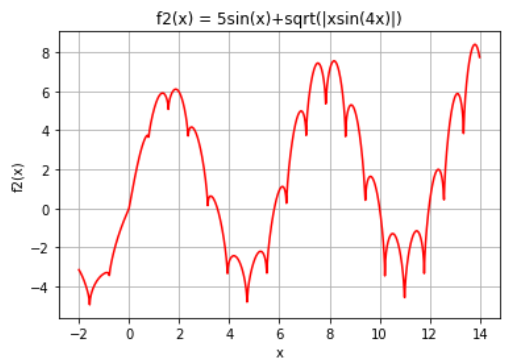
```
#   Example 8.4 - Plots in different windows
from numpy import arange, cos, sin, sqrt
import matplotlib.pyplot as plt

x_init1 = float(input('Enter the initial value of x for the first plot: '))
x_final1 = float(input('Enter the Final value of x for the first plot: '))
x_step1 = float(input('Enter the step-size for the x values for the first plot: '))

x1 = arange(x_init1, x_final1, x_step1)
f1 = 2*cos(x1)*sin(10*x1)

#   Plot # 1
plt.plot(x1,f1, 'g--')   #   plot between x and y
plt.xlabel('x')              #   x-label
plt.ylabel('f1(x)')              #   y-label
plot_title = 'f1(x) = 2cos(x)sin(10x)'
plt.title(plot_title)
plt.grid(True)
plt.show()                   # show will display the plot


x_init2 = float(input('Enter the initial value of x for the second plot: '))
x_final2 = float(input('Enter the Final value of x for the second plot: '))
x_step2 = float(input('Enter the step-size for the x values for the second plot: '))

x2 = arange(x_init2, x_final2, x_step2)
f2 = 5*sin(x2) + sqrt(abs(x2*sin(4*x2)))

#   Plot # 2
plt.plot(x2,f2, 'r')   #   plot between x and y
plt.xlabel('x')              #   x-label
plt.ylabel('f2(x)')              #   y-label
plot_title = 'f2(x) = 5sin(x)+sqrt(|xsin(4x)|)'
plt.title(plot_title)
plt.grid(True)
plt.show()                   # show will display the plot

#   Plot # 3 - Both plots in the same figure
plt.plot(x1,f1,x2,f2)   #   plot between x1, f1 and x2, y2 on the same figure
plt.xlabel('x')              #   x-label
plt.ylabel('f1(x) & f2(x)')              #   y-label
plot_title = 'f1(x) = 2cos(x)sin(10x) & f2(x) = 5sin(x)+sqrt(|xsin(4x)|)'
plt.title(plot_title)
plt.grid(True)
plt.legend(['f1(x)', 'f2(x)'])
plt.show()                   # show will display the plot
```

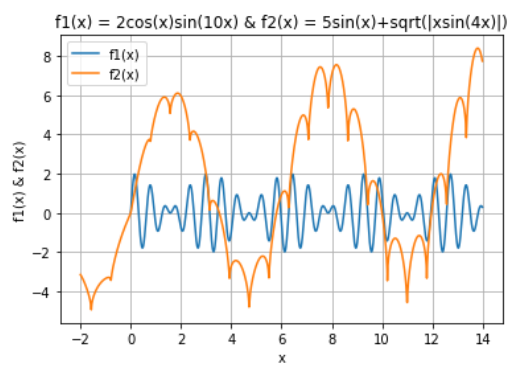
Output:

```
Enter the initial value of x for the first plot: 0

Enter the Final value of x for the first plot: 14

Enter the step-size for the x values for the first plot: 0.01
```




```
Enter the initial value of x for the second plot: -2

Enter the Final value of x for the second plot: 14

Enter the step-size for the x values for the second plot: 0.01
```





Multiple plots can also be plotted in different windows in the same figure using *subplot()* function (similar to MATLAB). *subplot(rcm)* means that the window will be divided into *r* rows and *c* columns, and *m*-th window will be chosen to plot the graph.

Example 8.5

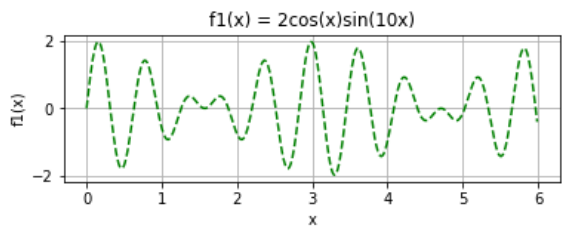
Plot the two functions from *example 8.4* in two subplots

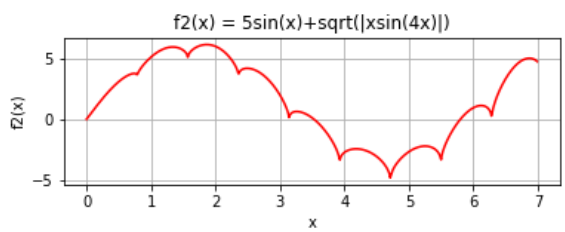
Code:

```
#   Example 8.5 - Subplots

from numpy import arange, cos, sin, sqrt
import matplotlib.pyplot as plt

x_init1 = float(input('Enter the initial value of x for the first plot: '))
x_final1 = float(input('Enter the Final value of x for the first plot: '))
x_step1 = float(input('Enter the step-size for the x values for the first plot: '))

x1 = arange(x_init1, x_final1, x_step1)
f1 = 2*cos(x1)*sin(10*x1)

x_init2 = float(input('Enter the initial value of x for the second plot: '))
x_final2 = float(input('Enter the Final value of x for the second plot: '))
x_step2 = float(input('Enter the step-size for the x values for the second plot: '))

x2 = arange(x_init2, x_final2, x_step2)
f2 = 5*sin(x2) + sqrt(abs(x2*sin(4*x2)))

#   Plot # 1
plt.subplot(211)        #   Divide figure into 2 rows and 1 column and plot the graph in the first window
plt.plot(x1,f1, 'g--')   #   plot between x and y
plt.xlabel('x')             #   x-label
plt.ylabel('f1(x)')             #   y-label
plot_title = 'f1(x) = 2cos(x)sin(10x)'
plt.title(plot_title)
plt.grid(True)
plt.show()                  # show will display the plot


#   Plot # 2
plt.subplot(212)    #   Plot graph in the second window of the subplot
plt.plot(x2,f2, 'r')   #   plot between x and y
plt.xlabel('x')             #   x-label
plt.ylabel('f2(x)')             #   y-label
plot_title = 'f2(x) = 5sin(x)+sqrt(|xsin(4x)|)'
plt.title(plot_title)
plt.grid(True)
plt.show()                  # show will display the plot
```

Output:

```
Enter the initial value of x for the first plot: 0

Enter the Final value of x for the first plot: 6

Enter the step-size for the x values for the first plot: 0.01

Enter the initial value of x for the second plot: 0

Enter the Final value of x for the second plot: 7

Enter the step-size for the x values for the second plot: 0.01
```





8.3 Other Plotting Function

Some of the other plotting functions available in *matplotlib.pyplot* are shown in *Table 8.3* [4].

Table 8.3: Some of the available plotting functions in Python

Function	Description
bar	*Bar graph*
contour	*Contours*
hist	*Histogram*
loglog	*Plot with log scaling*
pie	*Pie chart*
polar	*Polar plot*
stem	*Stem plot (discrete plot)*
step	*Staircase plot*

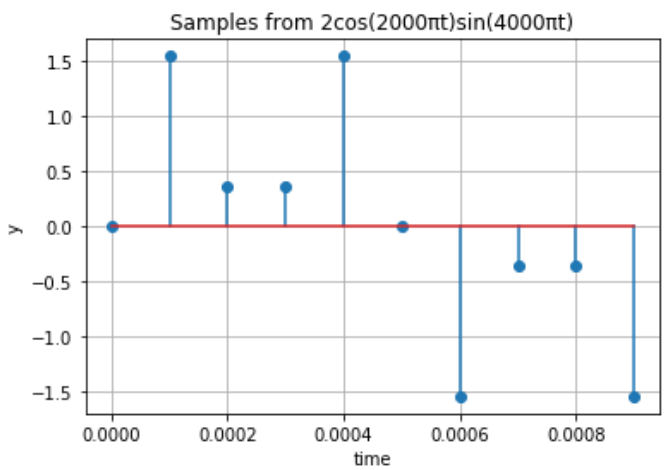
Example 8.6

Create a discrete plot for the signal *2cos(2000πt)sin(4000πt)* sampled with the sampling time of 100µs.

Code:

```
#   Example 8.6 - Stem plot

from numpy import arange, pi, cos, sin
import matplotlib.pyplot as plt

to = 100e-6   #   sampling time
t = arange(0,1e-3,to)   #   generating a time vector with sampling times

y = 2*cos(2000*pi*t)*sin(4000*pi*t)

plt.stem(t,y)   #   discrete plot between x and y
plt.xlabel('time')              #   x-label
plt.ylabel('y')             #   y-label
plot_title = "Samples from 2cos(2000\u03C0t)sin(4000\u03C0t)"
plt.title(plot_title)
plt.grid(True)
plt.show()                  # show will display the plot
```

Output:



Note that if different types of plots need to be plotted in the same window, *figure()* option can be used to determine which figure should be used for plotting;

matplotlib.pyplot.figure(x)

Exercises

8.1 Draw a plot for the following polynomial for *x* from -10 to 10 with step size of 0.1;

$$f(x) = x^5 + 3x^4 + 2x^3 + 10x^2 + 6x + 10$$

8.2 *Second-Order Control Systems:* Plot the response of a second-order control system as given by the following equation:

$$y(t,\varsigma) = 1 - \frac{1}{\sqrt{1-\zeta^2}} e^{-\zeta t} \sin(\sqrt{1-\zeta^2}\, t + \cos^{-1}(\zeta))$$

Ask user to enter the initial and final values of time and step-size. Also, ask to enter the damping coefficient value (ζ). Properly label your axes and put equation as title of your plot. Also, put a legend on the graph to show the value of the damping coefficient.

8.3 Create a function that takes an *expression* and *data points* to evaluate that expression as two input arguments, and plot the expression through your function. Turn the grid on as well. Test your function from your program.

8.4 Repeat *exercise 8.2* to plot response of the second-order control systems for multiple values of damping coefficient. Ask user to enter the initial and final value of the time, as well as step-size. Also, ask user to enter different values of damping coefficients as a list. Plot all responses in a single plot. Place a legend to show the corresponding value of damping coefficient for each plot.

8.5 *Fourier Series (revisited):* According to *Fourier*, any periodic function is a combination of three quantities: average value of the function, a sinusoid with the same frequency as the original periodic function, called *fundamental* component, and an infinite series of sinusoids, each with frequency to be a multiple of the fundamental frequency, called *harmonics*. This is called *Fourier series* of the periodic function.

Fourier series of a sawtooth waveform, as shown in *figure 8.1*, may be calculated from the following Fourier series:

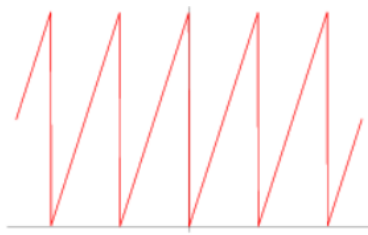


Figure 8.1: Sawtooth waveform

$$v(t) = \frac{V}{2} - \sum_{n=1,2,3,4,,\ldots} \frac{V}{n\pi} \sin(2\pi n f t)$$

where V is the peak value of the waveform, n is the harmonic number (n = 1 is the fundamental component), f is the frequency of the waveform (in Hertz), and t is the time range over which the Fourier series is evaluated (range of x-axis)

Write a program to calculate and plot the Fourier series of the sawtooth waveform for a range of time. Ask user to enter V, n, and f. Evaluate the series up to n-th harmonic for three time periods of the waveform. Take step size to be one-hundredth of the time period

8.6 *Quantization:* Quantization is a step in the conversion of an analog signal into its digital equivalent through an analog-to-digital converter (ADC) [26]. Create a function with input arguments to be a data vector, maximum reference voltage for the ADC (*Vref+)*, minimum reference voltage for the ADC (*Vref-*), and the number for bits for the ADC *(b*) (4 input arguments). The function should yield two plots in the same window:

(i) Input function vs. time (*f(t)* vs. *n*) using *plot* function
(ii) Quantized values of input function vs *n* using a step plot

Note that n goes from 0 to N-1. Following are the steps to calculate quantized value x_q for any input value x:

(i) Calculate the step-size for the ADC: $\Delta = \dfrac{(Vref+) - (Vref-)}{2^b}$

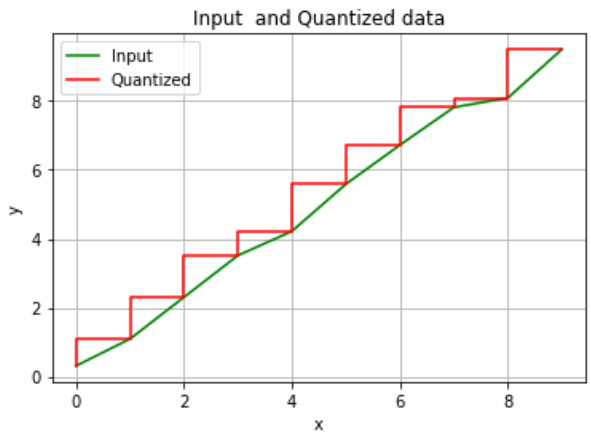
(ii) Calculate the value of step corresponding to the input value: $i = round((x - (Vref-)) / \Delta)$

(iii) Calculate the quantized voltage: $x_q = i\Delta + (Vref-)$

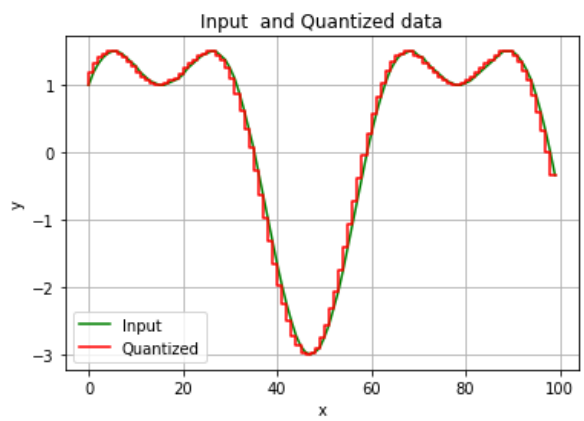
Check your function from your program

Sample Outputs

```
In [4]: from Exercise8_6 import quan

In [5]: A = [0.3, 1.1, 2.3, 3.5, 4.2, 5.6, 6.7, 7.8, 8.1, 9.5]

In [6]: quan(A, 10, 0, 8)
```




```
In [10]: from math import cos, sin

In [11]: x = [q*0.1 for q in range(100)]

In [12]: y = [2*sin(i) + cos(2*i) for i in x]

In [13]: quan(y,4,-4,8)
```



Chapter 9

Symbolic Mathematics

In this chapter, evaluation of equations, expressions, and mathematical operations will be explained using symbolic mathematics. The *sympy* (Symbolic Python) module is used for this purpose, which contains various functions to carry out symbolic calculations.

9.1 Algebraic Equations

A number of useful functions are available in *sympy* to carry out different algebraic operations and manipulations. Some of the most commonly used functions are shown in *Table 9.1* [4].

Table 9.1: Commonly used Algebraic Functions in 'sympy' Module

Function	*Explanation*
apart	Applies partial fraction decomposition on rational functions
cancel	Write rational functions with no common factors
collect	Collects like terms
expand	Writes expressions in an expanded form
factor	Generate factors of entered polynomials
simplify	Simplifies the expression
solve	Solve algebraic equation to find its roots
solve_linear_system	Solves a system of linear equations

It will be helpful to check the *help* for each function before using it to get a better understanding (*help*(*sympy.function_name*)).

Example 9.1

Create partial fractions of the following rational function:

$$\frac{20x^2+3}{x^6+3x^5-75x^4-155x^3+1194x^2+1272x-2240}$$

Code:

```
#   Example 9-1 - Partial Fraction in sympy

from sympy import *      #  This will import all functions from sympy

x = symbols('x')         #  defining that x will be used in the expression as 'x'

expr1 = (20*x**2 + 3)/(x**6+3*x**5-75*x**4-155*x**3+1194*x**2+1272*x-2240)   # Fractional expression
print('\n')
print(expr1)          #    Checking teh expression

A = apart(expr1)    #   Partial fraction
print('\n')
print(A)
```

Output:

```
(20*x**2 + 3)/(x**6 + 3*x**5 - 75*x**4 - 155*x**3 + 1194*x**2 + 1272*x - 2240)


-1283/(29160*(x + 8)) + 503/(5832*(x + 5)) - 83/(2916*(x + 2)) + 23/(2916*(x - 1)) - 323/(5832*(x - 4)) + 983/(29160*(x - 7))
```

Example 9.2

Expand and find roots of algebraic expressions

Code:

```
#   Example 9.2 - expand() and solve()

from sympy import *

x, y= symbols('x, y')

expr1 = 2*x*(y-1) + 2*x*(x+y)
print('Expansion')
print(expr1.expand(basic=True))

expr2 = x**4 - 15*x**2 + 10*x +24
print('\nRoots of', str(expr2))
print(solve(expr2))
```

Output:

```
Expansion
2*x**2 + 4*x*y - 2*x

Roots of x**4 - 15*x**2 + 10*x + 24
[-4, -1, 2, 3]
```

9.2 Limits

There is a *limit()* function in *sympy* to calculate limits of expressions [4]. The syntax is:

limit(expression, limit variable, limit value)

Example 9.3

Calculate limit of the following functions:

(i) $\lim_{x \to 0} \frac{\sin(5x)}{6x}$

(ii) $\lim_{x \to 0} \frac{3x^2 + 7x + 8}{6x}$

Code:

```
#   Example 9.3 - Limit

from sympy import *

x = symbols('x')

y1 = limit(sin(5*x)/(6*x), x, 0)
y2 = limit((3*x**2 + 7*x + 8)/(6*x), x, 0)

print(y1)
print(y2)
```

Output:

```
5/6
oo
```

9.3 Derivatives

sympy has a function *diff()* to calculate derivative of any function [4]. The syntax is:

diff(expression, variable of derivative, order of derivative)

Example 9.4

Calculate the first and second derivative of the following expression:

$$f(x) = 2\sin(x)\cos(6x)$$

Code:

```
#   Example 9.4 - Derivatives

from sympy import *

x = symbols('x')

y1 = diff(2*sin(x)*cos(6*x), x)
y2 = diff(2*sin(x)*cos(6*x), x, 2)

print(y1)
print(y2)
```

Output:

```
-12*sin(x)*sin(6*x) + 2*cos(x)*cos(6*x)
-2*(37*sin(x)*cos(6*x) + 12*sin(6*x)*cos(x))
```

9.4 Integrals

sympy has a function *integrate()* to calculate integral of any function [4]. The syntax is:

integrate(expression, (variable of integration, limits(optional)))

If integration is done over multiple variables:

integrate(expression, (first variable of integration, limits(optional)), (second variable, limits(optional))

Example 9.5

Integrate the following two functions:

$$f_1(x) = \int (2x^2 + \cos(x))dx$$

$$f_2(x, y) = \int_{y=0}^{2} \int_{x=0}^{3} (2xy + 4\sin(x))dxdy$$

Code:

```
#   Example 9.5 - Integrals

from sympy import symbols, cos, sin, integrate

x,y = symbols('x,y')

f1 = integrate(2*x**2 + cos(x), x)
f2 = integrate(2*x*y + 4*sin(x), (x, 0,3),(y,0,2))

print(f1)
print(f2)
```

Output:

```
2*x**3/3 + sin(x)
-8*cos(3) + 26
```

Note that if a closed-form solution of an integral does not exist, symbolic integration will just return the same input:

```
In [1]: from sympy import symbols, sin, cos, integrate

In [2]: x,y = symbols('x,y')

In [3]: f2 = integrate(2*x*y + cos(sin(x)), (x, -3, 3), (y, 0,2))

In [4]: print(f2)
Integral(2*x*y + cos(sin(x)), (x, -3, 3), (y, 0, 2))
```

9.5 Ordinary Differential Equations

sympy has a function *dsolve()* to ordinary differential equations [4]. Apart from *dsolve*, when solving for a solution for ordinary differential equations, some other functions from *sympy* are also required to be imported, which include *Eq*, *Function*, and *Derivative*.

Example 9.6

Determine the solution of following differential equations:

(i) $\dfrac{dy}{dx} + 3y = 3$

(ii) $\dfrac{d^2y}{dx^2} + y = 2$

Code:

```
#   Example 9.6 - Ordinary Differential Equations

from sympy import symbols, Eq, Derivative, dsolve, Function

x,y = symbols('x,y')

f = Function('f')   #   y is f
f1 = dsolve(Eq(Derivative(f(x),x) + 3*f(x),3), f(x))
f2 = dsolve(Eq(Derivative(f(x),x,x) + f(x), 2), f(x))

print(f1)
print(f2)
```

Output:

```
Eq(f(x), C1*exp(-3*x) + 1)
Eq(f(x), C1*sin(x) + C2*cos(x) + 2)
```

Explanation:

The solution of the first differential equation is, $y = Ce^{-3x} + 1$

The solution of the second differential equation is, $y = C_1 \sin(x) + C_2 \cos(x) + 2$

9.6 Equation Evaluation

Although it is easy to evaluate the value of an equation or an expression at any given value(s) of its variable(s), *sympy* also contains a function *evalf()* to do the same [4].

Example 9.7

Evaluate the values of the following equations at the required points.

(i) $\quad f_1(x) = 2x^2 + \cos(\sin(3x)); \quad x = 2.34$

(ii) $\quad f_2(x, y) = \dfrac{6x\cosh(x) + 3y\sin(x)}{2x + 3y}; \quad x = 2.0; y = 1.2$

Code:

```
#   Example 9.7 - Equation Evaluation

from sympy import symbols, evalf, sin, cos, cosh

x,y = symbols('x,y')

f1 = 2*x**2 + cos(sin(x))
f2 = (6*x*cosh(x) + 3*y*sin(x))/(2*x+3*y)

Result1 = f1.evalf(subs = {x:2.34})
Result2 = f2.evalf(subs = {x:2.0, y:1.2})

print('Function f1(x) evaluated at x = 2.34 gives', Result1)
print('Function f2(x) evaluated at x = 2.0 and y = 1.2 gives', Result2)
```

Output:

```
Function f1(x) evaluated at x = 2.34 gives 11.7040171347114
Function f2(x) evaluated at x = 2.0 and y = 1.2 gives 6.37102881968106
```

Exercises

9.1 Calculate the factors of the following polynomials:

(i) $x^5 + 2x^4 - 31x^3 - 8x^2 + 180x - 144$

(ii) $x^2 + 9$

9.2 Calculate the roots of the following polynomials:

(i) $x^5 + 2x^4 - 31x^3 - 8x^2 + 180x - 144$

(ii) $x^2 + 9$

9.3 Evaluate the following expressions at the points mentioned:

(i) $2x\cos(\sin(x)) + 3\sqrt{|x\sin(5x + 63^\circ)|}; \quad x = 5.8$

(ii) $2xy\tan(3.4x + y); \qquad x = 4, y = 0.3$

9.4 Current through a capacitor is related to the voltage across it as shown below,

$$i_c = C\frac{dv}{dt}$$

If voltage across capacitor is given by $2t\cos(5t)$, calculate the value of current at t = 2.2 sec. Assume C = 2F.

9.5 If current through a 100mF capacitor is given by 4sin(100t) mA, determine the expression for the capacitor voltage.

9.6 Write a program that asks user to enter values of resistor, inductor, and capacitor as a list for a parallel *RLC* source-free circuit. Solve for the expression of the voltage across capacitor from the following system's equation:

$$\frac{d^2v}{dt^2} + \frac{1}{RC}\frac{dv}{dt} + \frac{1}{LC}v = 0$$

9.7 Write a program that asks user to enter values of resistor, inductor, and capacitor as a list for a series *RLC* source-free circuit. Solve for the expression of the current through inductor from the following system's equation:

$$\frac{d^2i}{dt^2} + \frac{R}{L}\frac{di}{dt} + \frac{1}{LC}i = 0$$

Chapter 10

Numerical Methods

In this chapter, some of the numerical techniques used for interpolation, extrapolation, integration, differentiation, curve fitting, and ordinary differential equations are explained. Numerical techniques are generally used when a closed-form solution of a mathematical problem is hard to evaluate or achieve. Hence, using numerical methods, the solution is evaluated in numerical form, not in a closed-form. However, for some problems, a closed-form solution is achieved from given numerical points.

10.1 Interpolation

Interpolation is a technique to find out a new data point within a set of different data points. For example, if there are multiple (x, y) pairs and if a value of y for a specific value of x is required that lie within the range of given x points, interpolation can be used. There are many methods for interpolating the unknown value of y based on the given value of x. Some of these methods are *piecewise constant interpolation*, also called *nearest neighbor interpolation*, *linear interpolation*, *polynomial interpolation*, *spline interpolation* etc [26]. These methods differ from each other in the fact that how the given points are assumed to be connected with each other. For example, *linear interpolation*, connects all points through straight lines, whereas *polynomial interpolation* uses higher-order polynomials to connect the given points, as shown in *figure 10.1* [26].

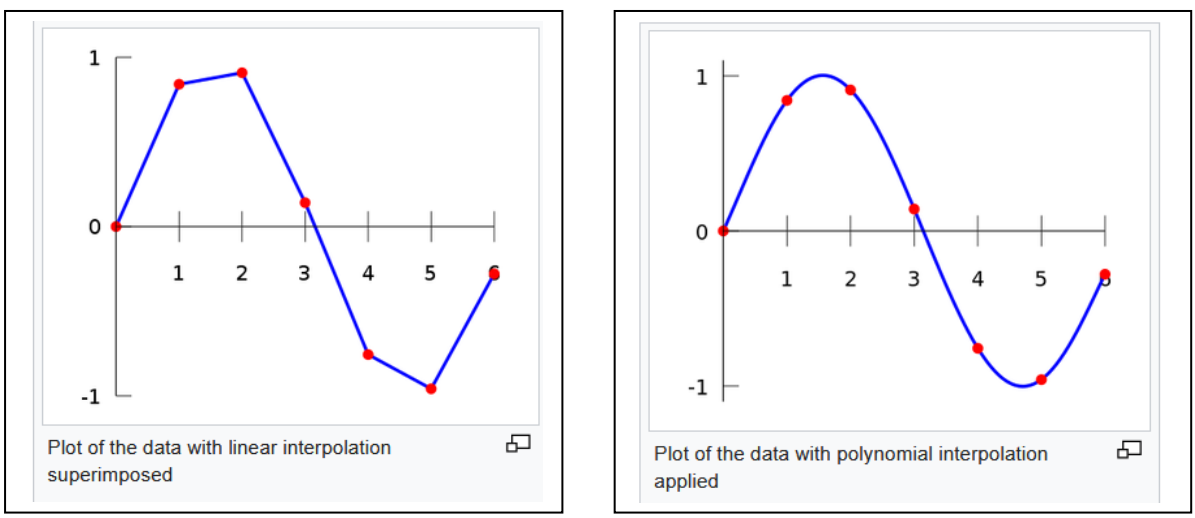


Figure 10.1: Linear and Polynomial Interpolation

In Python, the most commonly used interpolation function is *interp* from *numpy*. This function uses linear interpolation technique to determine the unknown value of *y* for a given value of *x*.

Example 10.1

Interpolate the values of *y* for *x* = 2.0, 3.0, and 4.5 from the given data points.

Data Points: (0.2, 1.1), (1.2, 3.4), (2.4, 5.6), (3.9, 2.9), (5.0, 1.2)

Code:

```
#   Exercise 10.1 - Interpolation

from numpy import interp

x = [0.2, 1.2, 2.4, 3.9, 5.0]
y = [1.1, 3.4, 5.6, 2.9, 1.2]

xi = [2, 3, 4.5]    #   points at which interpolation is reqd
yi = interp(xi, x, y)   #   performing interpolation at the required points

print('The interpolated values for x = ', xi, ' are', yi)
```

Output:

```
The interpolated values for x =  [2, 3, 4.5]  are [4.86666667 4.52       1.97272727]
```

10.2 Curve Fitting

Curve fitting is a numerical technique that yields a closed-form solution from numerical points. The idea is to create a polynomial that passes through the given numerical points. Note that a polynomial that will pass through all of the given numerical points is not always possible; hence, the resultant polynomial fits best for all points with the least possible error. Also, the higher the degree of polynomial, the smaller the error there will be between the actual values and evaluated values.

In Python, *polyfit* is the function to generate the coefficients of polynomial that will fit through the given points. Also, *poly1d* is the function to create the polynomial function from the coefficients. Both functions are part of *numpy* module [4].

Example 10.2

Write a code to find the third order polynomial for the following points: (0, 0), (1, 0.4), (1.6, 1.2), (2.2, 2.4), (3, 1), (3.5, -0.5), (4.0, -3.2), (4.5, -1.2), (5, 0.3), (5.5, 1.3). From the generated coefficient of polynomial, create the polynomial equation and evaluate its value at $x = 10$

Code:

```
#   Example 10.2 - Curve Fitting

from numpy import polyfit, poly1d

x = [0, 1, 1.6, 2.2, 3, 3.5, 4, 4.5, 5, 5.5]
y = [0, 0.4, 1.2, 2.4, 1, -0.5, -3.2, -1.2, 0.3, 1.3]

z = polyfit(x,y,deg = 3)    #   creating a polynomial of third order using curve fitting
fx = poly1d(z)  #   creating polynomial equation from coefficients z

print('\n Coefficients of the polynomial that will fit through the input points are: ', z)
print('\n Value of the polynomial at 10 is: ', fx(10))
```

Output:

```
 Coefficients of the polynomial that will fit through the input points are:
[ 0.25074148 -2.05512666  4.04620425 -0.53456975]

 Value of the polynomial at 10 is:  85.15628238602947
```

Note:

The polynomial generated by the program is:

$$0.25074148x^3 - 2.05512666x^2 + 4.04620425x - 0.53456975$$

10.3 Numerical Differentiation

Derivative of a function yields slope of the line tangent to the point on the function where the derivative is evaluated. It determines rate of change of the function with respect to the variable of differentiation. To calculate the derivative of a function at a given point numerically, *derivative()* function from *scipy.misc* module can be used. The *derivative* function employs central difference formula, which is based on Taylor series expansion of the input function close to the point at which the derivative is required to be evaluated. The numerical approximation of the first-order derivative of a function at a point x using central difference theorem is given by,

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{10.1}$$

where h is a distance very close to x, sometime referred to as dx. Higher-order numerical approximations of derivative using central difference theorem can also be derived.

Example 10.3

Evaluate the numerical approximation of the first- and second-order derivatives of the following function at x = 1.23. Assume spacing/mesh size to be 1×10^{-6}:

$$f(x) = 2\cos(x)\sin(3x)(x^3 + 2x^2 + 4)$$

Code:

```
#   Example 10.3 - Numerical Differentiation

from scipy.misc import derivative

def f(x):
    from numpy import sin, cos
    return 2*cos(x)*sin(x)*(x**3+2*x**2+4)  #   defining input function

D1 = derivative(f, 1.23, dx = 1e-6)  #   First-order derivative of the function at 1.23
D2 = derivative(f, 1.23, dx = 1e-6, n = 2)  #   Second-order derivative of the function at 1.23

print('\n First-Order Derivative: ',D1)
print('\n Second-Order Derivative: ',D2)
```

Output:

```
First-Order Derivative:  -7.842972303162554

Second-Order Derivative:  -44.60520841575999
```

Note that for a third or higher order derivative, use the *order* argument in the function with a higher odd number value assigned to the *order* argument. In addition to that, do not use a very small mesh size.

Example 10.4:

Evaluate the third-order derivative of $2x^5\cos(\sin(6x))$ at $x = 2$

Code:

```
#   Example 10.4 - Fall 2020

#   Evaluate the third-order derivative of 2x^5*cos(sin(6x)) at x = 2

from scipy.misc import derivative


def f(x):
    from numpy import cos, sin
    return 2*x**5 + cos(sin(6*x))

D3 = derivative(f, 2, dx = 0.001, n = 3, order = 5)

print(D3)
```

Output:

68.32036092419003

10.4 Numerical Integration

There are quite a few numerical integration methods that calculate the area under the given function curve between two limits (definite integrals). A big class of numerical integration methods is called *Quadrature* methods. This includes *Gaussian*, *Clenshaw-Curtis*, *Newton-Cotes*, *Gauss-Kronrod*, *Gauss-Hermite*, *Gauss-Lagurre*, and *tinh-sinh* quadrature methods. These methods are generally based on interpolation or extrapolation rules. In addition to these methods, there are some other methods including *Romberg's*, *Riemann*, and *Runge-Kutta* methods [27].

In Python, several numerical integration functions are present under *scipy.integrate* sub-package. Two functions are examined here, *quad* and *romberg* [4].

quad function has three input arguments, *function*, *lower limit*, and *upper limit*, and two output arguments, *integral value* and *error* between the actual integral value and numerical integral value. *romberg* function has same input arguments as *quad* and has one output argument, *integral value*.

Example 10.5

Write a code to perform numerical integration using *quad* and *romberg* functions for the function $2x\cos(x)+\sin(4x)$ between $2 \le x \le 3$.

Code:

```
#   Example 10.4 - Numerical Integration

from scipy import sin, cos
from scipy.integrate import quad, romberg

y = lambda x: 2*x*cos(x)+sin(4*x)

int1, err1 = quad(y, 2, 3)
int2 = romberg(y, 2, 3)

print('Integral of 2xcos(x)+sin(4x) from x = 2 to 3 using quad method is ', int1, 'and error is ', err1)
print('\nIntegral of 2xcos(x)+sin(4x) from x = 2 to 3 using romberg method is ', int2)
```

Output:

```
Integral of 2xcos(x)+sin(4x) from x = 2 to 3 using quad method is  -4.185499477185406 and error is
4.6468378891277926e-14

Integral of 2xcos(x)+sin(4x) from x = 2 to 3 using romberg method is  -4.185499477189357
```

Exercises

10.1 Generate 10 values from *10cos(x)sin(2x)* for x = 0:10. Now interpolate 10 values between each of the (*x*,*y*) pairs. Hence, you will have a total of 100 values. Print out the original *x*-*y* with 10 values and another *x*-*y* plot with the interpolated 110 values.

10.2 *Extrapolation:* In mathematics, *extrapolation* is the process of estimating, beyond the original observation range, the value of a variable on the basis of its relationship with another variable. Like interpolation, there are several methods for extrapolating values of *y* for given values of *x*. In this exercise you have to write a function that will perform *linear extrapolation*.

Linear extrapolation is carried out by extending the line from the last point to the point at which value of *y* is extrapolated. Linear extrapolation will only provide good results when used to extend the graph of an approximately linear function or not too far beyond the known data. If (x_1, y_1) and (x_2, y_2) are the last two points, respectively, beyond which another point (x_3, y_3) is extrapolated, the following relationship can be used:

$$y_3 = y_1 + \frac{x_3 - x_1}{x_2 - x_1}(y_2 - y_1)$$

Your function should have three input arguments; last two pairs of data points, and *x* value of the extrapolated *y*. The function should return extrapolated value of *y*. Check your function from your program.

10.3 Create polynomials with order from 4 to 9 for the points given in *Example 10.2*. For each polynomial, evaluate *y* for each given *x* value. Print the following outputs with eight properly organized columns: (i) actual *x* (ii) actual *y* (iii-viii) *y* for polynomials with order four to nine for each value of *x*.

10.4 Evaluate first-, second-, and third-order derivatives of $x^3 + 2x^2 + 9\cos(5x)$ at $x = 2.8$. Keep spacing to be 0.001.

10.5 Calculate the voltage across a 10μF capacitor for 1ms ≤ t ≤ 2ms if a current 2cos(x)sin(2x) passes through it. Remember, the relationship between the capacitor voltage and current is given by,

$$i_c = C\frac{dv}{dt}$$

Chapter 11

Graphical User Interface (GUI)

Graphical User Interface (GUI) is an important aspect of most of the programs and applications these days. Python possesses several toolkits to build GUIs, These toolkits include, but not limited to, *tkinter*, *wxPython*, *PyQt*, *PyGTK* etc. In this chapter *tkinter* toolkit will be used to create basic GUIs.

11.1 Widgets

In GUIs, widgets include *Button*, *Canvas*, *Checkbutton*, *Entry*, *Frame*, *OptionMenu*, *Scrollbar*, *Scale*, *Text*, and many more from *tkinter* module [4]. A list of all the available widgets can be found by importing *tkinter* module and using *dir()* function [28].

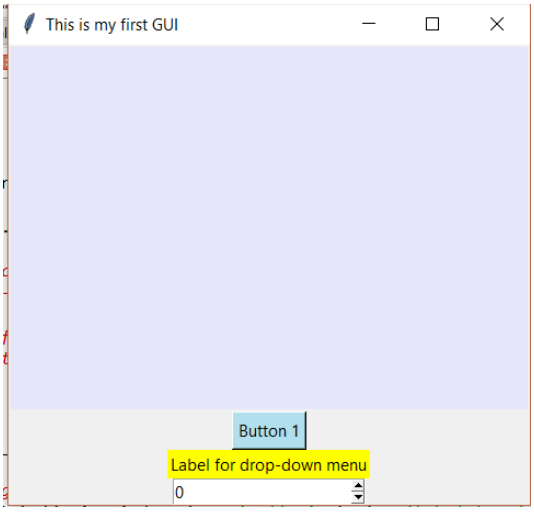
Exercise 11.1

Create a window with some widgets [4].

Code:

```
#   Example 11_1 - GUIs Example # 1
#
#   In this example, a main window will be created with a button, a spin box

from tkinter import *    #   Import all functions from tkinter
main_window = Tk()       #   This creates an object on class Tk which defines main window of the GUI
main_window.title("This is my first GUI")    #   Defining the method 'title' on the object

#   Widget creation inside the main window
#   --------------------------------------

#   Button function from tkinter will create a button in the main window with background color blue
#   and text 'Button 1' with teh following syntax
My_Button = Button(main_window, bg = 'LightBlue2', text = 'Button 1')

#   Spinbox function from tkinter will create a drop-down menu in teh main window
My_dropdown_menu = Spinbox(main_window, from_=0, to=5)

#   Creating a label for the drop down menu (spinbox)
My_label = Label(main_window, bg = 'yellow', text = 'Label for drop-down menu')

#   Creating a canvas inside the main window
My_Canvas = Canvas(main_window, bg = 'lavender'  )

#   Placing created widgets in the main window with PACKING geometry manager

My_Canvas.pack()        #   canvas is placed in teh main_window
My_Button.pack()        #   button is placed in the main window
My_label.pack()         #   label is placed in the main window
My_dropdown_menu.pack() #   dropdown menu is placed in teh main window

mainloop()  #   This will hold everything in the main wondow intact until the object is closed
```

Output:



Note that each function from *tkinter* module can be customized in many ways. For example, many different background and foreground colors can be used, location of canvas can be adjusted, location of buttons can be adjusted etc. These options can be checked using *help(tkinter.function_name)* or using one of the online references [28]. A pallet of colors that can be used with different widgets in *tkinter* module is shown in *figure 11.1* [29].

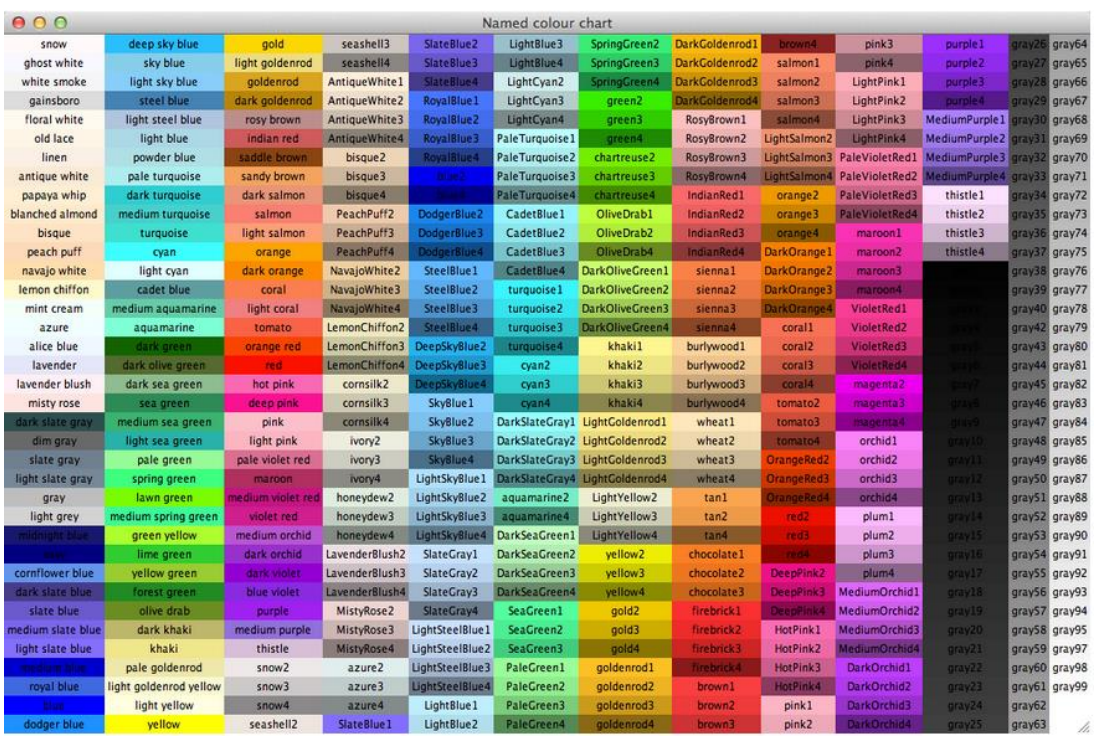


Figure 11.1: Color Pallet for Background Colors for Widgets from tkinter

Let's explore different options available for a textbox through *Example 11.2* [29].

Example 11.2

Write a code to create a *listbox* with following options:

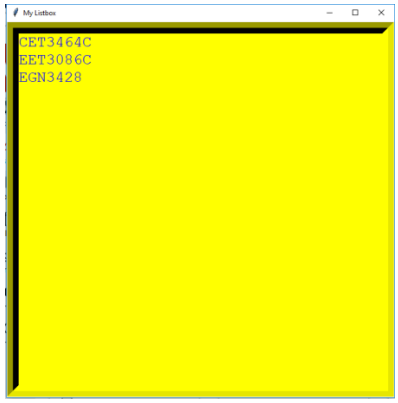
- Foreground color is blue
- Background color is yellow
- Width is 40
- Height is 20
- Border width is 20
- Font is Courier
- Font size is 20

The *listbox* should list different course options

Code:

```
#   Example 11.2
#   Listbox

from tkinter import *
main_window = Tk()       #   This creates an object on class Tk which defines main window of the GUI
main_window.title("My Listbox")    #   Defining the method 'title' on the object

My_Listbox = Listbox(main_window, fg = 'blue', bg = 'yellow', width = 40, height = 20, borderwidth = 20,
                     font = ('Courier', 20))
My_Listbox.pack()   #   Listbox is placed in the main window

Classes = ['CET3464C', 'EET3086C', 'EGN3428']   #   list of classes to appear in the listbox

for k in Classes:
    My_Listbox.insert(END, k)   #   END is required to insert items in a listbox

mainloop()
```

Output:



11.2 Geometry Management

To arrange and display widgets in the main window, three main approaches are used; *pack*, *place*, and *grid*. *pack* method has already been used in the previous examples. The *place* method requires more attention to make windows work within a pattern and usually not used for ordinary windows and dialog layouts [4][30]. *grid* method is a better approach, which will be discussed in detail.

The *grid* manager allows to put widgets in the GUI considering the position of rows and columns. It lines up a widget like a table. Some of the important options that can be used within *grid* method are *row*, *column sticky*, *padx*, and *pady*, along with some other options. A description of *grid* options are summarized in *Table 11.1* [31].

Table 11.1: Grid Options in tkinter()

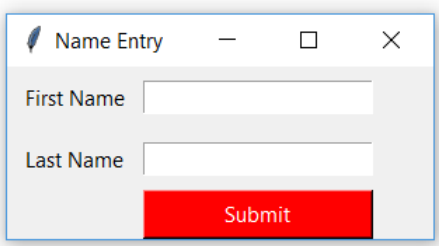
grid option	Explanation
column	Insert the widget at this column. Column numbers start with 0. If omitted, defaults to 0.
columnspan	If given, indicates that the widget cell should span multiple columns. The default is 1.
*in, in_*	Place widget inside a given widget. You can only place a widget inside its parent, or in any descendant of its parent. If this option is not given, it defaults to the parent. Note that **in** is a reserved word in Python. To use it as a keyword option, append an underscore (**in_**).
ipadx	Optional horizontal internal padding. Works like **padx**, but the padding is added *inside* the widget borders. Default is 0.
ipady	Optional vertical internal padding. Works like **pady**, but the padding is added *inside* the widget borders. Default is 0.
padx	Optional horizontal padding to place around the widget in a cell. Default is 0.
pady	Optional vertical padding to place around the widget in a cell. Default is 0.
row	Insert the widget at this row. Row numbers start with 0. If omitted, defaults to the first empty row in the grid.
rowspan	If given, indicates that the widget cell should span multiple rows. Default is 1.
sticky	Defines how to expand the widget if the resulting cell is larger than the widget itself. This can be any combination of the constants **S**, **N**, **E**, and **W**, or **NW**, **NE**, **SW**, and **SE**. For example, **W** (west) means that the widget should be aligned to the left cell border. **W+E** means that the widget should be stretched horizontally to fill the whole cell. **W+E+N+S** means that the widget should be expanded in both directions. Default is to center the widget in the cell.

Example 11.3

Write a code to enter first name and last name in an entry. At the bottom, a submit button should be located [4].

Code and Output:

```
#   Example 11.3
#
#   Write a code to enter first name and last name in an entry.
#   At the bottom, a submit button should be located

from tkinter import *
main_window = Tk()        #   This creates an object on class Tk which defines
                         #   main window of the GUI
main_window.title("Name Entry")    #   Defining the method 'title' on the object

label1 = Label(main_window, text = "First Name")
label1.grid(row = 0, column = 0, sticky = W, padx = 10)  #   aligned to the left
entry1 = Entry(main_window)
entry1.grid(row = 0, column = 1, sticky = E, pady = 10)  #   aligned to the right

label2 = Label(main_window, text = "Last Name")
label2.grid(row = 1, column = 0, sticky = W, padx = 10)  #   aligned to the left
entry2 = Entry(main_window)
entry2.grid(row = 1, column = 1, sticky = E, pady = 10)  #   aligned to the right

Button1 = Button(main_window, text = "Submit", bg = "red", fg = "white")
Button1.grid(row = 4, column = 1, sticky = W+E)     #   stretch the submit button in column 1
                                                    #   from one side to the other
mainloop()
```



11.3 Callback Functions

Callback functions create functionality between entry from a widget and resultant actions that are supposed to be taken. For example, if a button is pressed in a widget, what should be the result?

Example 11.4

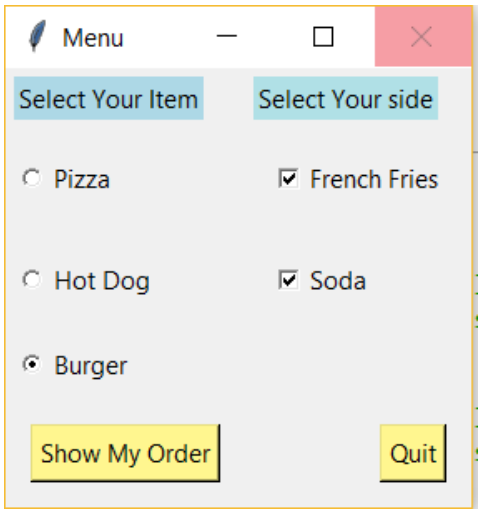
Create a GUI to select food items from a fast food order menu. Once selection is done, print out the selected items.

Code

```
#   Example 11.4
#
#   Create a GUI to select food items from a fast food order menu.
#   Once selection is done, print out the selected items.

from tkinter import *
main_window = Tk()
main_window.title("Menu")

#   Assigning different variable types to be used from tkinter

Btn1 = StringVar()  #   All three variables can take string data
Var1 = IntVar()
Var2 = IntVar()

def MyCall():
    print("You Selected: ", Btn1.get())     #   The value that will be entered
                                            #   for Btn1 will be displayed

    if Var1.get():
        print("French Fries")   #   sides
    if Var2.get():
        print("Soda")

#   Creating Widgets

label1 = Label(main_window, bg = "light blue", text = "Select Your Item")
label1.grid(row = 0, column = 0, sticky = W, padx = 5, pady = 5)

label2 = Label(main_window, bg = "powder blue", text = "Select Your side")
label2.grid(row = 0, column = 1, sticky = W, padx = 5, pady = 5)

#   Items

Rad1 = Radiobutton(main_window, text = "Pizza", variable = Btn1, value = "Pizza")
Rad1.grid(row = 1, column = 0, sticky = W, padx = 5, pady = 5)

Rad2 = Radiobutton(main_window, text = "Hot Dog", variable = Btn1, value = "Hot Dog")
Rad2.grid(row = 2, column = 0, sticky = W, padx = 5, pady = 5)

Rad3 = Radiobutton(main_window, text = "Burger", variable = Btn1, value = "Burger")
Rad3.grid(row = 3, column = 0, sticky = W, padx = 5, pady = 5)

#   Sides

Chek1 = Checkbutton(main_window, text = "French Fries", variable = Var1)
Chek1.grid(row = 1, column = 1, sticky = W, padx = 15, pady = 15)

Chek2 = Checkbutton(main_window, text = "Soda", variable = Var2)
Chek2.grid(row = 2, column = 1, sticky = W, padx = 15, pady = 15)
```

```
#   Execution Buttons
B1 = Button(main_window, text = "Show My Order", bg = "khaki1", command = MyCall)
B1.grid(row = 4, column = 0, sticky = W, padx = 15, pady = 15)

B2 = Button(main_window, text = "Quit", bg = "khaki1", command = main_window.quit)
B2.grid(row = 4, column = 1, sticky = E, padx = 15, pady = 15)

mainloop()
```

Output:




```
You Selected:  Burger
French Fries
Soda
```

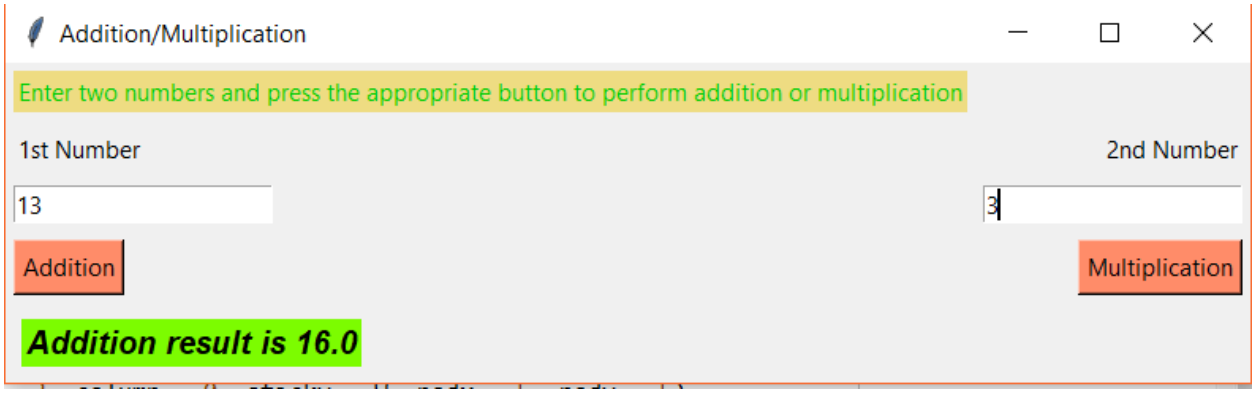
Example 11.5

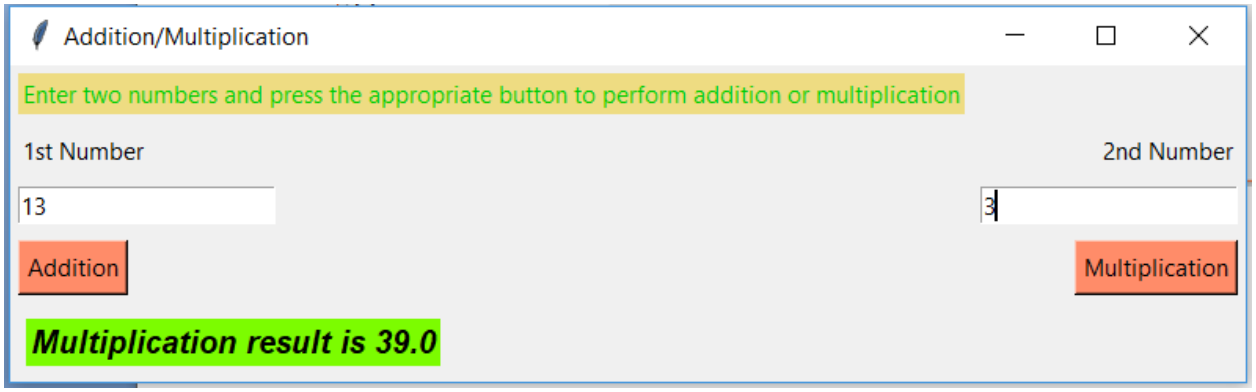
Create a GUI where two numbers are entered and two options are given; addition and multiplication. Based on the selected option, either addition or multiplication will be performed on the two numbers and result will be displayed.

Code:

```
#   Example 11-5
#
#   Create a GUI where two numbers are entered and two options are given;
#   addition and multiplication. Based on the selected option, either
#   addition or multiplication will be performed on the two numbers and
#   result will be displayed.

from tkinter import *
main_window = Tk()
main_window.title("Addition/Multiplication")

def Num_Add():  #   function to perform addition
    result = num1.get()+num2.get()
    Showlabel['text'] = "Addition result is {}".format(result)
def Num_Mul():  #   function to perform multiplication
    result = num1.get()*num2.get()
    Showlabel['text'] = "Multiplication result is {}".format(result)

num1 = DoubleVar()
num2 = DoubleVar()

num1.set = ("0")      #   Set the initial value to the first number to 0
num2.set = ("0")

NameLabel = Label(main_window, bg = "LightGoldenrod2", fg = "green3",
    text = "Enter two numbers and press the appropriate button to perform addition or multiplication")
NameLabel.grid(row = 0, column = 0, padx = 5, pady = 5)

num1Label = Label(main_window, text = "1st Number")     #   Label for the first number
num1Label.grid(row = 2, column = 0, sticky = W, padx = 5, pady = 5)

num1Labe2 = Label(main_window, text = "2nd Number")
num1Labe2.grid(row = 2, column = 1, sticky = E, padx = 5, pady = 5)

#   Number Entry
num1Entry = Entry(main_window, textvariable = num1)
num1Entry.grid(row = 3, column = 0, sticky = W, padx = 5, pady = 5)

num2Entry = Entry(main_window, textvariable = num2)
num2Entry.grid(row = 3, column = 1, sticky = E, padx = 5, pady = 5)

#   Calculation Buttons; command = Num_Add_Mul will call the function once the button is pressed
Add_Button = Button(main_window, text = "Addition", bg = "salmon1", command = Num_Add)
Add_Button.grid(row = 5, column = 0, padx = 5, pady = 5, sticky = W)

Mul_Button = Button(main_window, text = "Multiplication", bg = "salmon1", command = Num_Mul)
Mul_Button.grid(row = 5, column = 1, padx = 5, pady = 5, sticky = E)

Showlabel = Label(main_window, bg = "lawn green", text = "", font=("Helvetica", 12, "bold italic"))
Showlabel.grid(row = 6, column = 0, sticky = W, padx = 10, pady = 10)

mainloop()
```

Output:





In the previous example, *num1* and *num2* are two control variables to hold values of the two numbers. These are defined as *floating point* variables. Including *DoubleVar*, other control variables that can be defined under *tkinter* are shown in *Table 11.2* [35].

Table 11.2: Control Variables in tkinter

Variables	*Function*
StringVar()	Holds a string; default value ""
IntVar()	Holds an integer; default value 0
DoubleVar()	Holds a float; default value 0.0
BooleanVar()	Holds a boolean, returns 0 for False and 1 for True

To read the current value of such a variable, call the method *get()*. The value of such a variable can be changed with the *set()* method. This is shown in the previous example.

The next example shows how to create a plot embedded in a GUI. Values of different parameters of a function can be changed interactively to yield the corresponding graph.

Example 11.6

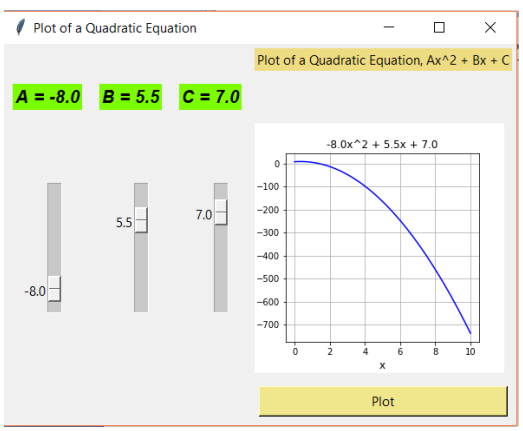
Create a GUI to plot a quadratic equation where the coefficients of the quadratic equation can be changed using three slider buttons.

Code:

```
#   Example 11.6
#
#   Create a GUI to plot a quadratic equation where the coefficients of
#   the quadratic equation can be changed using three slider buttons.

from tkinter import *
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg as FC
import matplotlib.pyplot as plt
from numpy import linspace

main_window = Tk()
main_window.title("Plot of a Quadratic Equation")

NameLabel = Label(main_window, bg = "LightGoldenrod2",
   text = "Plot of a Quadratic Equation, Ax^2 + Bx + C")
NameLabel.grid(row = 0, column = 10, padx = 5, pady = 5)

#   To draw three sliders for A, B, and C

#   Slider A
def Quad_CoeffA(val):
    NameA["text"] = "A = {}".format(val)

NameA = Label(main_window, bg = "lawn green", text = "", font=("Helvetica", 12, "bold italic"))
NameA.grid(row = 1, column = 0, sticky = W, padx = 10, pady = 10)

#   Scale widget will create a sliding scale. Check reference [33] for Scale widget
ScaleA = Scale(main_window, from_=10, to=-10, resolution = 0.5, command = Quad_CoeffA, length = 150)
ScaleA.grid(row = 2, column = 0, sticky = W, padx = 10, pady = 10)

#   Slider B
def Quad_CoeffB(val):
    NameB["text"] = "B = {}".format(val)

NameB = Label(main_window, bg = "lawn green", text = "", font=("Helvetica", 12, "bold italic"))
NameB.grid(row = 1, column = 1, sticky = W, padx = 10, pady = 10)

#   Scale widget will create a sliding scale. Check reference [33] for Scale widget
ScaleB = Scale(main_window, from_=10, to=-10, resolution = 0.5, command = Quad_CoeffB, length = 150)
ScaleB.grid(row = 2, column = 1, sticky = W, padx = 10, pady = 10)

#   Slider C
def Quad_CoeffC(val):
    NameC["text"] = "C = {}".format(val)

NameC = Label(main_window, bg = "lawn green", text = "", font=("Helvetica", 12, "bold italic"))
NameC.grid(row = 1, column = 2, sticky = W, padx = 10, pady = 10)

#   Scale widget will create a sliding scale. Check reference [33] for Scale widget
ScaleC = Scale(main_window, from_=10, to=-10, resolution = 0.5, command = Quad_CoeffC, length = 150)
ScaleC.grid(row = 2, column = 2, sticky = W, padx = 10, pady = 10)
```

```python
#   Plotting the graph
def Quad_Plot():
    x = linspace(0, 10, 100)    #   x-range for the plot

    A = ScaleA.get()
    B = ScaleB.get()
    C = ScaleC.get()

    fx = A*x**2 + B*x + C

    fig = plt.figure(figsize = (4,4))
    plt.plot(x,fx,color = "blue")
    plt.ylabel('f(x)', fontsize = 12)
    plt.xlabel('x', fontsize = 12)
    plt.title(str(A)+"x^2 + "+str(B)+"x + "+str(C))
    plt.grid(True)
    canvas = FC(fig, master = main_window)
    canvas.get_tk_widget().grid(row = 2, column = 10, sticky=SW, padx = 5, pady = 5)
    canvas.draw()

#   Plot Button

Plot_Btn = Button(main_window, text = "Plot", bg = "khaki", command = Quad_Plot)
Plot_Btn.grid(row = 4, column = 10, sticky = E+W, padx = 10, pady = 10)

mainloop()
```

Output:



11.4 Games and Applications

Games and different applications (apps) use GUI. In addition to the GUI layout, each game and application require some logic to be performed between different variables to yield results. Let's look at designing a simple game next.

Example 11.7

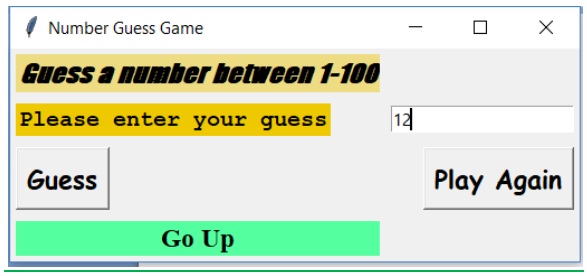
Create a GUI to guess a random number generated between 1 and 100. If user guess is incorrect, guide the user by displaying if guess is lower or higher than the random number. Once user guesses the number correctly, display the number of iterations it took to guess the correct number [4].

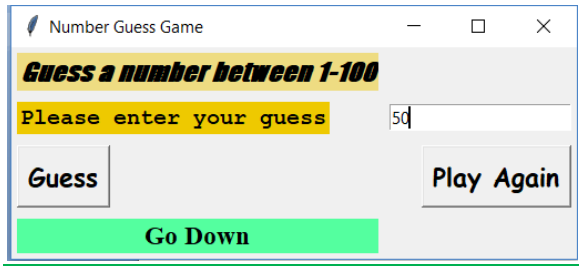
Code:

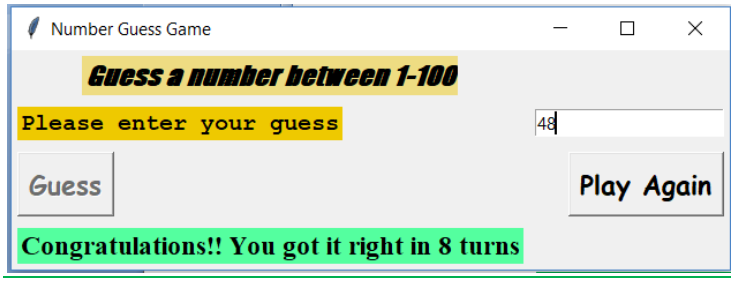
```
#   Example 11.7
#
#   Create a GUI to guess a random number generated between 1 and 100.
#   If user guess is incorrect, guide the user by displaying if guess is lower or higher
#   than the random number. Once user guesses the number correctly,
#   display the number of iterations it took to guess the correct number.

from tkinter import *
from random import randint

main_window = Tk()
main_window.title("Number Guess Game")

NameLabel = Label(main_window, bg = "LightGoldenrod2",
   text = "Guess a number between 1-100", font=("Impact", 14, "bold italic"))
NameLabel.grid(row = 0, column = 0, padx = 5, pady = 5)

Picked_Number = randint(1,100)  #   Random integer between 1 and 100
Counter = 1;    #   initial value of Counter

def Start_Guess():   #   function to start guessing a number
    global Picked_Number, Counter     #  global variable to be used inside and outside the function
    Guess = Guess_Number.get()  #   getting the value of user entered guess
    if (Guess == Picked_Number):
        Message["text"] = "Congratulations!! You got it right in {} turns".format(Counter)
        Guess_Button.configure(state = DISABLED)
        Play_Again_Button.configure(state = NORMAL)
    elif(Guess>Picked_Number):
        Message["text"] = "Go Down"
    else:
        Message["text"] = "Go Up"
    Counter+=1  #   increasing the value of counter

def Play_Again():
    global Picked_Number, Counter     #  global variable to be used inside and outside the function
    Guess_Button.configure(state = NORMAL)
    Play_Again_Button.configure(state = DISABLED)
    Counter = 1 #   resetting the counter
    Picked_Number = randint(1,100)  #   Random integer between 1 and 100
```

```
#   Guess Entry
Guess_Number = IntVar()
Guess_Entry = Entry(main_window, textvariable = Guess_Number)
Guess_Entry.grid(row = 3, column = 2, sticky = W, padx = 5, pady = 5)

#   Guess Label
Guess_Label = Label(main_window, bg = "gold2", text = "Please enter your guess",
                    font=("Courier", 12, "bold"))
Guess_Label.grid(row = 3, column = 0, sticky = W, padx = 5, pady = 5)

#   Guess Button
Guess_Button = Button(main_window, text = "Guess", command = Start_Guess,
                      font=("Comic Sans MS", 14, "bold"))
Guess_Button.grid(row = 4, column = 0, sticky = W, padx = 5, pady = 5)

#   Play Again Button
Play_Again_Button = Button(main_window, text = "Play Again", command = Play_Again,
                           font=("Comic Sans MS", 14, "bold"))
Play_Again_Button.grid(row = 4, column = 2, sticky = E, padx = 5, pady = 5)

#   Message Label
Message = Label(main_window, bg = "SeaGreen1", text = "", font=("Times New Roman", 14, "bold"))
Message.grid(row = 6, column = 0, sticky = W+E, padx = 5, pady = 5)

mainloop()
```

Output:







Exercises

11.1 Create a GUI with three entries; *resistor*, *inductor*, and *capacitor*. There should be two buttons; *Parallel RLC* and *Series RLC*. Based on the entered values of components and the button that is pressed, display the values of *Neper Frequency* (α), *Resonant Frequency* (ω_o), and the two roots of the characteristic equation for the corresponding RLC circuit, s_1 and s_2. Put a proper label to explain the purpose of the GUI as well.

Parallel RLC Circuits:

$$\alpha = \frac{1}{2RC}; \qquad \omega_o = \frac{1}{\sqrt{LC}}$$

$$s_{1,2} = -\alpha \pm \sqrt{\alpha^2 - \omega_o^{\,2}}$$

Series RLC Circuits:

$$\alpha = \frac{R}{2L}; \qquad \omega_o = \frac{1}{\sqrt{LC}}$$

$$s_{1,2} = -\alpha \pm \sqrt{\alpha^2 - \omega_o^{\,2}}$$

11.2 *Second-Order Control Systems:* Create a GUI with an embedded plot of the response of a second-order control system as given by the following equation:

$$y(t,\varsigma) = 1 - \frac{1}{\sqrt{1-\zeta^2}} e^{-\zeta t} \sin(\sqrt{1-\zeta^2}\,t + \cos^{-1}(\zeta))$$

Take time value to from 0 to 15 with step size of 0.01. Assign a slider to the value of the damping coefficient ζ that can be changed from 0 to 0.95 with resolution of 0.05. Properly display the chosen value of ζ as well. Label your axes and put equation as title of your plot.

Bibliography

[1] S. Cass, 'The 2018 Top Programming Languages', IEEE Spectrum, 2018. [Online]. Available: https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages [Accessed: August 26, 2018]

[2] 'General Python F.A.Q', Python Software Foundation. [Online]. Available: https://docs.python.org/2/faq/general.html#why-is-it-called-python [Accessed: August 26, 2018]

[3] 'Built-in Functions', Python Software Foundation. [Online]. Available: https://docs.python.org/2/library/functions.html# [Accessed: August 26, 2018]

[4] Irfan Turk, *Python Programming for Engineers and Scientists*. Columbia, S.C.: CreateSpace Independent Publishing Platform

[5] 'Common String Operations', Python Software Foundation. [Online]. Available: https://docs.python.org/2/library/string.html [Accessed: September 8, 2018]

[6] 'Python Tutorial', Sololearn. [Online]. Available: https://www.sololearn.com/Play/Python/ [Accessed: September 8, 2018]

[7] 'Python Strings Methods', Python by Programiz. [Online]. Available: https://www.programiz.com/python-programming/methods/string [Accessed: September 8, 2018]

[8] 'String Formatting', Learnpython.org. [Online]. Available: https://www.learnpython.org/en/String_Formatting. [Accessed: September 9, 2018]

[9] 'Using *%* and *.format()* for great good!', Pyformat. [Online]. Available: https://pyformat.info/. [Accessed: September 9, 2018]

[10] 'Data Structures', Python Software Foundation. [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html [Accessed: September 10, 2018]

[11] 'Unicode characters for engineers in Python', Python for Undergraduate Engineers. [Online]. Available: https://pythonforundergradengineers.com/unicode-characters-in-python.html [Accessed: September 12, 2018]

[12] 'List of Unicode Characters', Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/List_of_Unicode_characters [Accessed: September 12, 2018]

[13] 'Python Dictionary Methods', Python by Programiz. [Online]. Available: https://www.programiz.com/python-programming/methods/dictionary [Accessed: September 15, 2018]

[14] 'Sets – Unordered collection of unique elements', Python Software Foundation. [Online]. Available: https://docs.python.org/2/library/sets.html [Accessed: September 15, 2018]

[15] 'Python Sets', Python by Programiz. [Online]. Available: https://www.programiz.com/python-programming/set [Accessed: September 15, 2018]

[16] 'Python – How to save functions', Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/20938456/python-how-to-save-functions [Accessed: September 16, 2018]

[17] 'Procedural Programming', Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Procedural_programming [Accessed: September 30, 2018]

[18] 'Inheritance', Python Course. [Online]. Available: https://www.python-course.eu/python3_inheritance.php [Accessed: October 12, 2018]

[19] 'Python Inheritance and polymorphism', The Python Guru. [Online]. Available: https://thepythonguru.com/python-inheritance-and-polymorphism/ [Accessed: October 11, 2018]

[20] 'Magic methods and operator overloading', Python Course. [Online]. Available: https://www.python-course.eu/python3_magic_methods.php [Accessed: October 12, 2018]

[21] 'Subclass of a class', Codesdope. [Online]. Available: https://www.codesdope.com/python-subclass-of-a-class/ [Accessed: October 13, 2018]

[22] 'Matrix manipulation in Python', GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/matrix-manipulation-python/ [Accessed: October 19, 2018]

[23] 'Python Matrix', Python by Programiz. [Online]. Available: https://www.programiz.com/python-programming/matrix [Accessed: October 20, 2018]

[24] 'matplotlib.pyplot.plot', matplotlib. [Online]. Available: https://matplotlib.org/2.1.1/api/_as_gen/matplotlib.pyplot.plot.html [Accessed: October 21, 2018]

[25] Li Tan, *Digital Signal Processing – Fundamentals and Applications*. Purdue University North Central.: Elsevier

[26] 'Interpolation', Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Interpolation [Accessed: November 4, 2018]

[27] 'Numerical Integration', Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Numerical_integration [Accessed: November 5, 2018]

[28] 'Tkinter', Python Tkinter Course. [Online]. Available: https://www.python-course.eu/tkinter_labels.php [Accessed: November 11, 2018]

[29] 'The Tkinter Listbox Widget', An Introduction to Tkinter. [Online]. Available: http://effbot.org/tkinterbook/listbox.htm [Accessed: November 15, 2018]

[30] 'The Tkinter Place Geometry Manager', An Introduction to Tkinter. [Online]. Available: http://effbot.org/tkinterbook/place.htm [Accessed: November 15, 2018]

[31] 'The Tkinter Grid Geometry Manager', An Introduction to Tkinter. [Online]. Available: http://effbot.org/tkinterbook/grid.htm#Tkinter.Grid.grid-method [Accessed: November 15, 2018]

[32] 'The Variable Classes', An Introduction to Tkinter. [Online]. Available: http://effbot.org/tkinterbook/variable.htm [Accessed: November 17, 2018]

[33] 'The Tkinter Scale Widget', An Introduction to Tkinter. [Online]. Available: http://effbot.org/tkinterbook/scale.htm [Accessed: November 18, 2018]

[34] 'Print Lists in Python (4 Different Ways)', Geeks for Geeks [Online]. Available: https://www.geeksforgeeks.org/print-lists-in-python-4-different-ways/ [Accessed: November 12, 2020]

[35] 'Tkinter Variable Classes', Python Tkinter Course [Online]. Available: https://www.python-course.eu/tkinter_variable_classes.php [Accessed: November 30, 2020]