# Python

# The Logo Way

by

DR. MOHAMMAD SAEED

# *Python*

# The *Logo* Way

by

Dr. Mohammad Saeed

This book is made available under the Creative Commons Attribution 4.0 International Public License. This means you are free to share and adapt this work as long as you give appropriate credit and indicate if changes were made. This work will be available without any restrictions.
https://creativecommons.org/licenses/by/4.0/legalcode

*To*

*My children & all the children of the world*

# Contents


**Preface** ........ 8

**I. Basics** ........ 14

**1. Starting Python** ........ 15

**2. Squares and Circles** ........ 16

Your first program ........ 16

Saving your program ........ 18

Turtle commands ........ 19

Playing with Colors ........ 20

*Program:* Pakistan Flag ........ 22

**3. Q & A** ........ 23

Let's talk to our Python programs ........ 23

Writing ........ 25

Function ........ 26

Choice ........ 28

*Program:* Basic Calculator ........ 30

**4. Cycling** ........ 31

Whiling away ........ 34

Randomization ........ 34

*Program:* Polygons ........ 36

*Program:* Polygons - 2 ........ 38

# II. Gaming .......... 40

**5. Game Board** .......... 41

Shaping up your Turtle .......... 42

*Program:* Ping-Pong .......... 43

**6. Moving Objects** .......... 44

Key to Move .......... 44

Auto-Move .......... 45

Play time .......... 49

Score Board .......... 49

Winner .......... 50

**7. Sights and Sounds** .......... 51

Musica .......... 51

# III. GUI .......... 55

**8. Tkinter** .......... 56

Buttons .......... 57

Display .......... 58

Fonts .......... 62

Functioning Buttons .......... 63

Changing Icon .......... 67

**9. Art Toolkit** .......... 69

Calculation to Drawing .......... 69

Button labels .......... 71

**IV. Files** ........ 73

**10. Reading & Writing** ........ 74

Let’s Read ........ 74

Writing ........ 76

**11. Tables** ........ 77

Dictionary and List ........ 78

Creating an HTML file ........ 83

Hyperlinks ........ 89

**12. Graphs** ........ 90

Matplotlib ........ 90

Graph modifications ........ 93

Labeling the data points ........ 95

Bar Graph ........ 96

Pie Chart ........ 97

Fractal Graph ........ 99

Saving Images ........ 103

*Program:* Fractal Genes ........ 103

**Epilogue** ........ 106

**About the Author** ........ 110

# Preface

Learning is a beautiful journey and it is made by making lots of mistakes. Those who are afraid of making mistakes learn little. The best learners are children. They are not afraid. They explore and experiment. This is my journey to this book and I dedicate it to my children. All children of the world are like my children - beautiful, curious to learn and asking big questions like ‘How’ and ‘Why’. This book is for all the children to spark their curiosity.

I was introduced to computers and programming in Grade 6. This was the 1980s. St. Patrick’s High School in Karachi under the able leadership of Bishop Lobo was a far thinking institution. The Cold War was coming to an end and though the Afghan War with USSR raged next door, I felt safe at school in Pakistan. I had just gained admission to the ‘Cambridge’ section of the school. There were several educational systems in Pakistan (and still are) and the most modern curriculum with the best teachers were found in the ‘Cambridge’ system. The Section was led by Mrs. Yolande Henderson – one of the most influential people of my life. Mrs. Henderson was a just leader, a highly competent Teacher, and influenced everyone with her moral courage and steadfast character. Bishop Lobo and Mrs. Henderson introduced Computer studies into our curriculum at a time when computers were rare and expensive. They required special air conditioned rooms away from the sun and computer teachers were even rarer.

Mr. Patrick was our Computer Teacher. We addressed male teachers as, ‘Sir’. So ‘Sir Patrick’ was the one who led me into this beautiful journey of computer programming. He was tall, slim young man who knew how to command young boys. One day as seventh graders our class rushed in to the computer room and everyone tried to grab a computer for themselves (there were maybe 8 to 10 computer stations). The noise level was high and Mr. Patrick stood by the white board with a marker. He ordered the class to be quiet. Then asked, ‘How many want to work on the computer and how many want to learn on the board?’ Who would have chosen the board! We all said we want the computers. His next question was a bit stunning, ‘What will you do on the computers?’ Of course, we had no idea. So he wrote on the white board four large letters, ‘GIGO’. He said, ‘Garbage In, Garbage Out’. All the boys quietly positioned themselves towards the board and after he had taught the concepts he wished to, he allowed us to

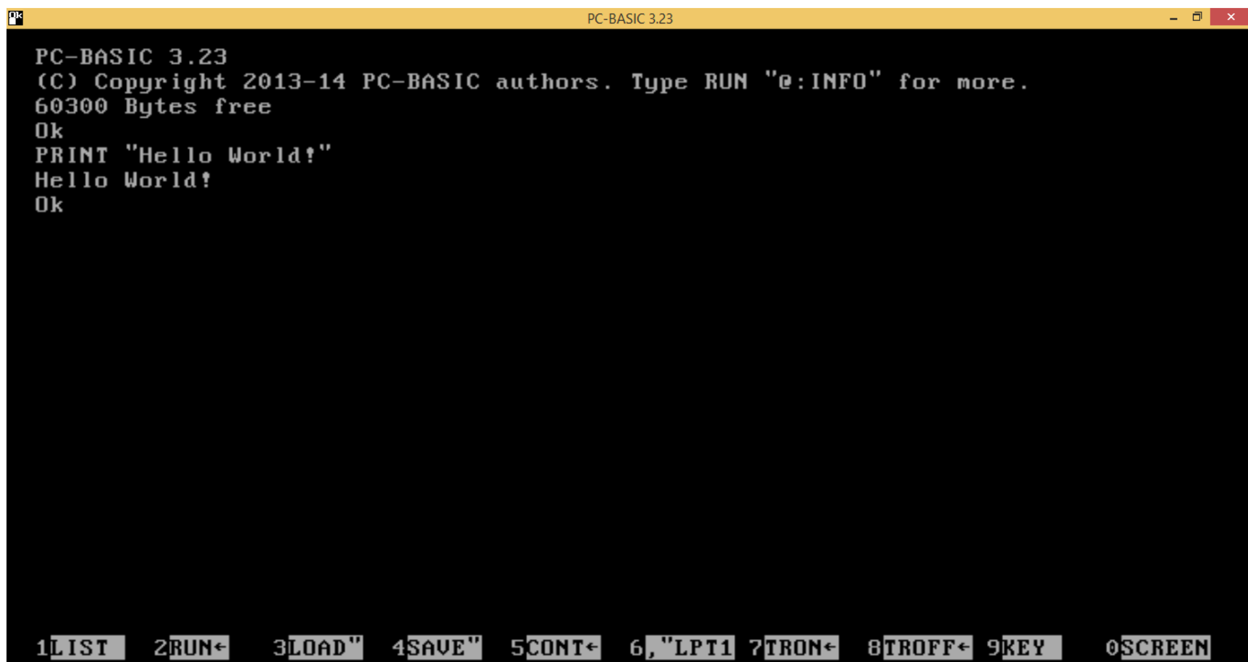
practice them on the computers. The class left disciplined, dignified and nothing like the unruliness with which they had entered. GIGO is a lesson that has had a lasting impact on my life.

Computers were so fascinating, I was usually found in the Computer lab after school with just a handful of boys. For about an hour of extra time after school I would get the dedicated teaching of Sir Patrick and unrestricted access to the computers. We started learning 'Logo' as 6$^{th}$ graders and by 8$^{th}$ grade I represented my school in a National competition in programming. Three of us made a 'clock', with its minute, hour and seconds' hands, ticking like a real clock and one could set any time one wanted. We made it in Logo (1-5). Of course much of the work was done after school and I had already met that selection criterion.

The 9$^{th}$ graders were put on a different project. One day, I saw them place a tracing paper with the map of Pakistan on the screen. Then they started using the arrow keys to draw the map on the screen. It fascinated me. They were not using Logo and it was interesting how they could program the computer to draw. My curiosity drove me to find out a bit more. All I could, was that they used a language called BASIC. It was a while later that I dived into GW-BASIC and even merged my suite of Logo programs with BASIC. I finally was able to make a matching program, which I called the 'Artliner'. It allowed me to draw on the screen, as well as save my images and use those in my game called , 'Skiing' – similar to an Atari game in which the skier moved at the bottom of the screen and trees moved in random order towards him. If he avoided a tree the score went up. If he bumped into the tree a life was lost (max 3 lives allowed). By grade 11, I programmed, Artliner, to write GW-BASIC code for whatever I drew on the screen. This way I taught the computer to program itself.

My romance with GW-BASIC continued well into my PostDoc at Northwestern University, Chicago. I wouldn't have been able to do what I did in my PostDoc, had I not known GW-BASIC. I had to pipette DNA from four 96-well plates into a 384-well plate using a multi-pipetter with 8 channels. They would fit into alternate wells in the 384-well plate. A multi-pipette with adjustable channels would cost the lab $8000. In order to save that money, I would use the existing multi-pipette to pipette into alternate wells in columns (vertically). The second problem arose with the scanning of the 384-well plate that was done by the ABI-7900 machine in rows (horizontally). My database was built in the order of the 96-well plates. So

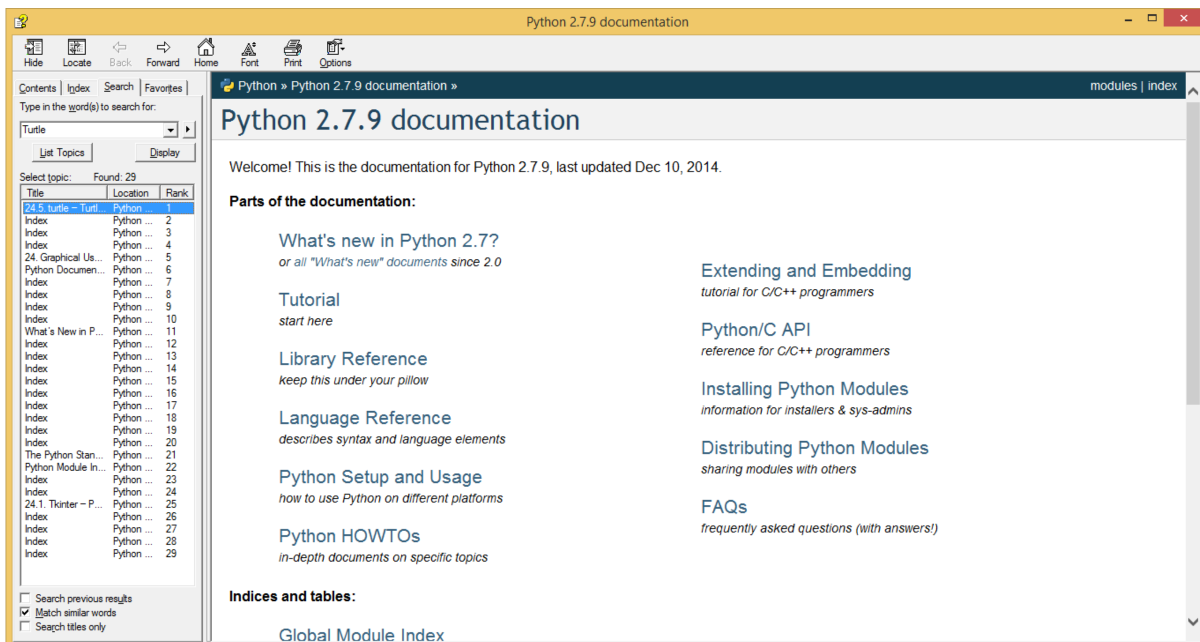
the task was to build a program that would convert the horizontally read data points (genotypes) into the format of the 96-well plates. I achieved it with a modest sized code in GW-BASIC, saved $8000 for the lab and got my results for ~ 2000 DNA samples in a single day along with its statistical analysis. As soon as I would get the genotypes in the database order, my analysis suites were all ready to crunch the data and spit out the genetic association results.




BASIC, however, started to be severely limited with Window upgrades (6). It was nearly impossible to run the small 64kb file that had run so well on an 8088 processor all the way through 286, 386, 486, Pentium and finally on Duo-Core, on iCore5 and beyond. Special setup had to be made, such as setting up a Windows XP virtual machine. It ran, but things were not the same. I had to graduate out of GW-BASIC which was my creative machine for over 2 decades. It was painful, I tell you. Such a loss. I carried my little programs on to the latest machines as well, but running them effectively was nearly impossible. I needed a new language that would allow me to take my Genetics' research into a fresh realm.

I kept searching for a new version of BASIC; something that would allow me to keep the coding formats, essentially the language, but be accessible to me on

current laptops. I got lucky in Dec 2014 and came across ‘PC-BASIC’. It fulfilled all my requirements. Interestingly, it was made in a language called, ‘Python’. This fascinated me even more and I looked deeper into Python. By that time Python was ranked as number 1 language in Data Science. I found its formats not very different from BASIC / QBASIC (QuickBASIC an interim version to Visual BASIC). Python is a freely available language with lot of online coding support. Its documentation in the HELP section is very supportive indeed.




I benefitted greatly from a COURSERA course from RICE University taught by Professors Joe Warren and Scott Rixner. They used ‘Codeskulptor’ as their teaching tool for Python. Another great help was Professor Charles Severance’s free online resources especially his book, ‘Python for Everybody’ (8). Then, I found the ‘Turtle’ module embedded in Python 2.7.9 which brought me full circle back to Logo (9): “Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig, Cynthia Solomon and Seymour Papert in 1966” (1-5).

Here, I present Python to children in the fun-way I learnt with Sir Patrick in school, using the Turtle module of Python. I use the Python 2 version, which I find closer to GW-BASIC format and once learnt can easily be upgraded to Python 3. I have

used Dr. Chuck's PY4E book and his 'Python for Informatics' (2009) (10), as a guide to teach the coding concepts and mimic the color coding format of the GW-BASIC manual (11) that I found easy to read and apply. I use the Logo approach (way) with Python's 'Turtle' module to make programming visually understandable especially for children. This is how coding became fun and a highly creative activity for me and I hope it will be the same for the readers of this book.

I feel that just like the English language became the entry point for higher education, machine language will be in the very-near future. I pointed this out on a Facebook post in 2015, soon after learning Python and encouraged its teaching in medical colleges and schools. Now, that vision is coming to fruition with several schools and a few medical colleges in Pakistan, starting to teach Python.

ImmunoCure
10 July 2015 at 00:35

Machine language is the English of the near Future.

I encourage the teaching of a simple programming language such as Python in schools and Medical Colleges

http://pythonlearn.com/

Freely available video lectures and book on Python programming by Prof Charles Severance.

To be able to communicate with machines is the skill that soon will determine career success in every profession including Medicine - due to the advancement of Genomics and the near advent of personalised medicine. Python is a freely available, simple to use yet powerful programming language. Tomorrow's labs will be more in silico (dry) than wet bench and we'll be working with Big Data.

PYTHONLEARN.COM
**PythonLearn - Exploring Data**
The goal of this site is to provide a set of materials in support of my Python for Informatics: Exploring Information book to allow you to learn Python on your own. This page serves as an outline of the materials to support the textbook.

3 Likes

In the past 3-years, I had introduced Python to my children and they have enjoyed learning it. Now is the time to bring Python at the finger-tips of every child so that they can express themselves creatively in the programming domain and enjoy it thoroughly as well.

Best wishes for a creative journey

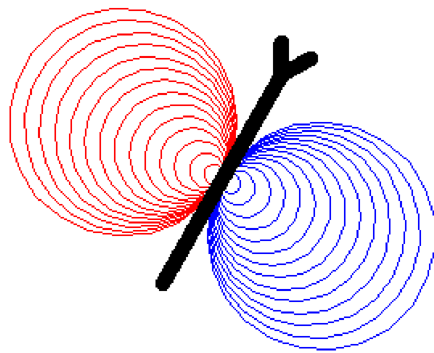
Dr. Mohammad Saeed

December 2020, Karachi, Pakistan
Website: www.immunocure.pk
Twitter: @DrMSaeed_pk
Email: msaeed@immunocure.pk

*References:*


1. https://compform.net/turtles/
2. https://www.media.mit.edu/posts/the-seeds-that-seymour-sowed/
3. https://mindstorms.media.mit.edu/
4. https://el.media.mit.edu/logo-foundation/index.html
5. https://news.mit.edu/2016/seymour-papert-pioneer-of-constructionist-learning-dies-0801
6. Challenge to scientists: does your ten-year-old code still run? Nature. TECHNOLOGY FEATURE. 24 AUGUST 2020. https://www.nature.com/articles/d41586-020-02462-7
7. https://www.dr-chuck.com/
8. https://www.py4e.com/html3/01-intro
9. https://docs.python.org/2/library/turtle.html
10. http://www.py4inf.com/
11. https://hwiegman.home.xs4all.nl/gw-man/index.html#google_vignette

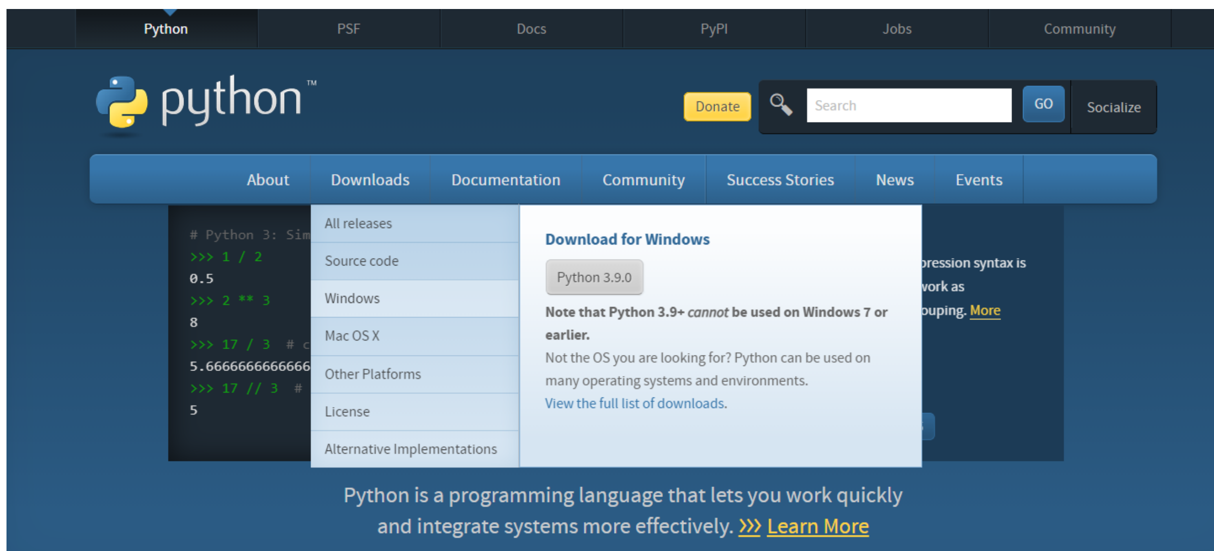
# I. Basics



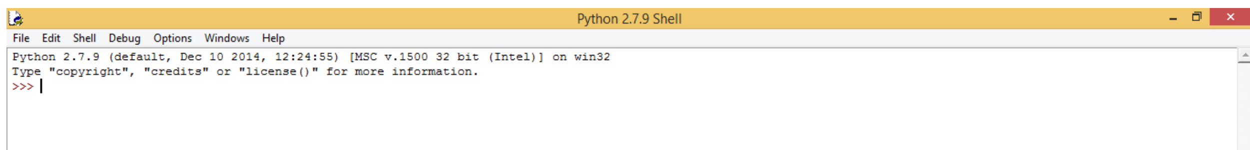
# 1. Starting Python

Welcome to Python. I assume you are running PC / Laptop with a Windows environment and have internet access. I encourage you to use Google Chrome as your internet Browser.

Go to www.python.org, click **Downloads** and select **Windows** from the dropdown menu. Although, Python 3 is the newer version I encourage you to use Python 2 as it has more online help available and I find the format easier to learn.




I use the Python 2.7.9 version (2014), however you may want to use the latest version of Python 2. For download I suggest you choose, **Windows x86 MSI installer** as many modules are not available in the other version such as the '**64 MSI installer**'.




When Python has downloaded, follow the instructions to install it on your computer and open **Python IDLE** (Integrated DeveLopment Environment) (1-3). You will see the screen above. Congratulations you are ready to program.

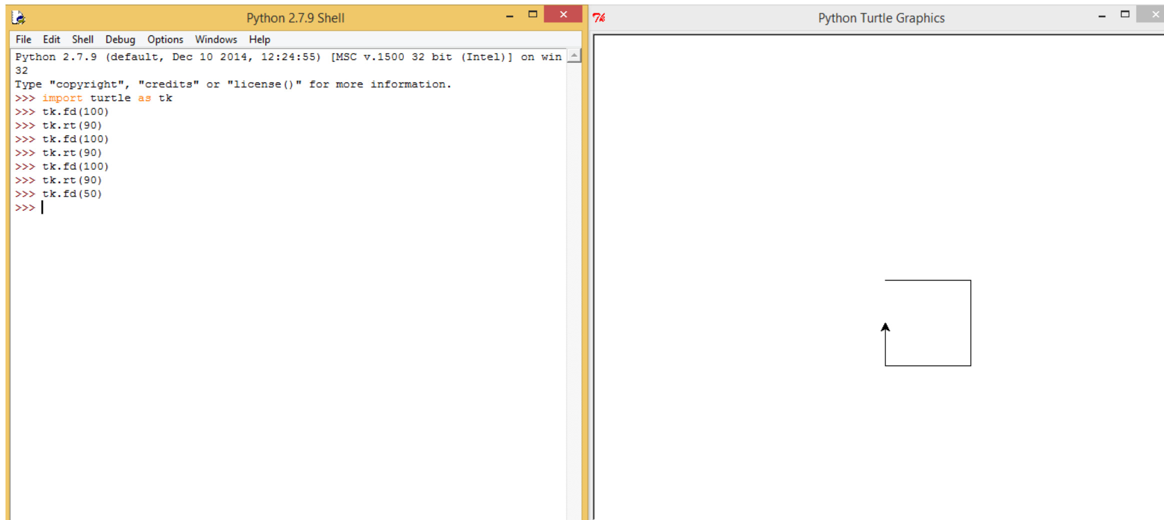
# 2. Squares and Circles

## Your first program

It is easy. The triple arrows (**>>>**) indicate that Python is waiting for your command. Since we will learn Python, The Logo Way, we'll ask Python to make us enter into the Logo environment. This happens with the magic command that opens the door in to the Logo world:

**Import turtle as tk**

(You can simply write: import turtle but then each command you give to turtle will have to be preceded by the entire word, turtle. The above command allows us to shorten our code by using an arbitrary set of letters, tk).




Now, we are ready. The Turtle module allows us to draw on the screen.

The first command to learn is to tell Turtle to move forward. The command is also called **forward** or **fd** for short. The format is as follows:

**tk.forward(100)**

or

**tk.fd(100)**

100 is the distance you want the turtle to move and is always written within brackets

This will make Turtle move 100 points forward

Similar to forward there are commands called **right (rt)**, **left (lt)** and **backward (bk)**. The right and left commands turn the angle of the turtle, whereas backward is the opposite of forward.

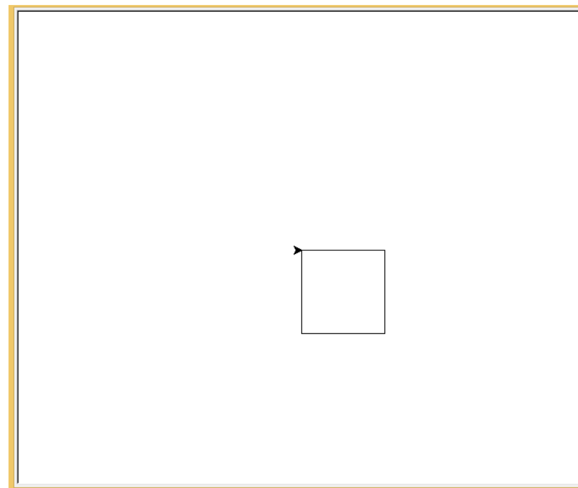
Note:

Turtle points to the East in the standard Python mode. However, in the Logo mode the turtle always points North. But we won't bother about this right now.

We are trying to make a Square. So we'll command the turtle to turn right 90', then move forward 100 more points. If we do this four times we will complete our Square.

Try the complete program below:

```
>>> import turtle as tk
>>> tk.fd(100)
>>> tk.rt(90)
>>> tk.fd(100)
>>> tk.rt(90)
>>> tk.fd(100)
>>> tk.rt(90)
>>> tk.fd(100)
>>> tk.rt(90)
>>>
```

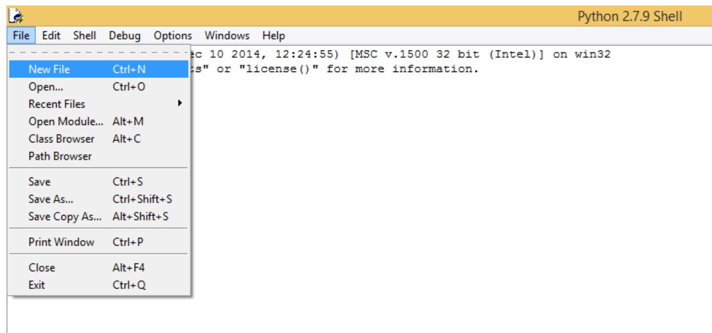


Congratulations! You have completed your first program.

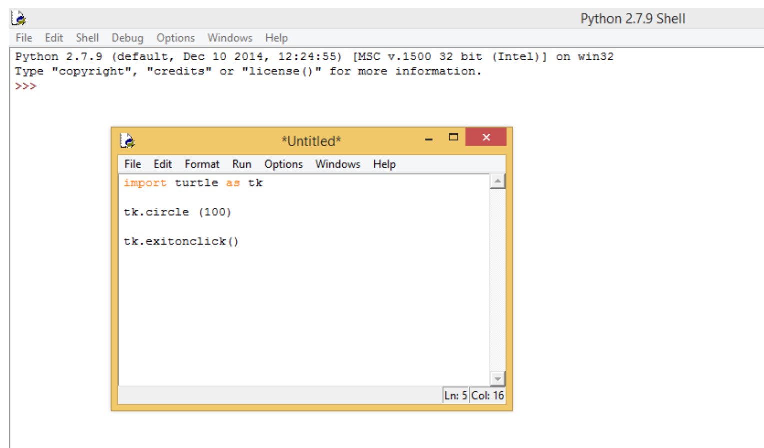
*Exercise 1.* Try making a Square using the commands **lt(90)** and **bk(100)**

## Saving your program

So far the program you have made is in the environment called **Python Shell**. This is a temporary area for your programs, mostly used to test a few commands. Real programming is done by creating a '**New File**' as shown below:




Here we are going to make a program which will command Turtle to draw a circle with radius 100 points. You would have noticed when drawing the square that the Turtle window would stall and you would have had to close it forcefully. The reason is that you need to use a Turtle command **exitonclick** to prevent Turtle from stalling.

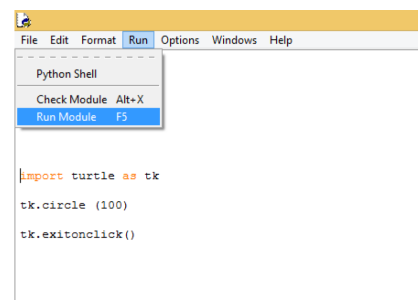



Your Circle program will have the following code (notice there are no arrows **>>>**):

**import turtle as tk**

**tk.circle (100)**

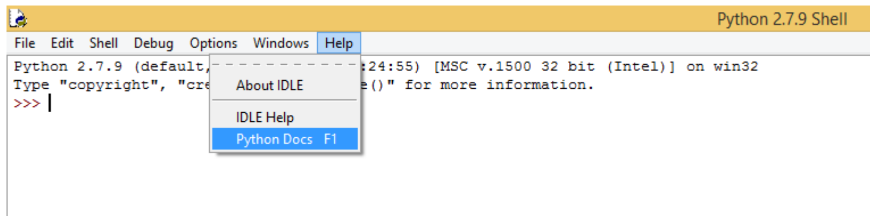
**tk.exitonclick()**




You will need to save your program before you can run it as shown here (→). In fact as you try to Run your program, Python will ask you to save it automatically.

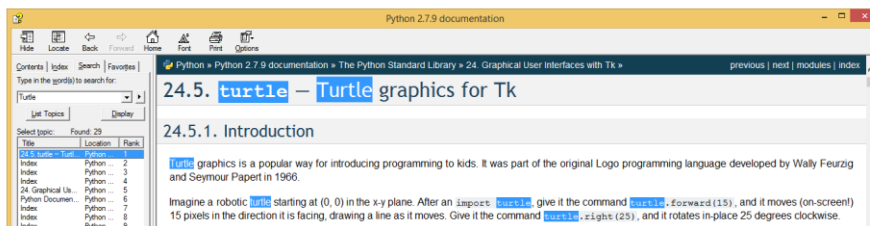
## Turtle commands

It's time to learn a few more ways to command your Turtle. The easiest way to do so is to open IDLE, click on HELP and select Python Docs or Press F1 directly in IDLE.

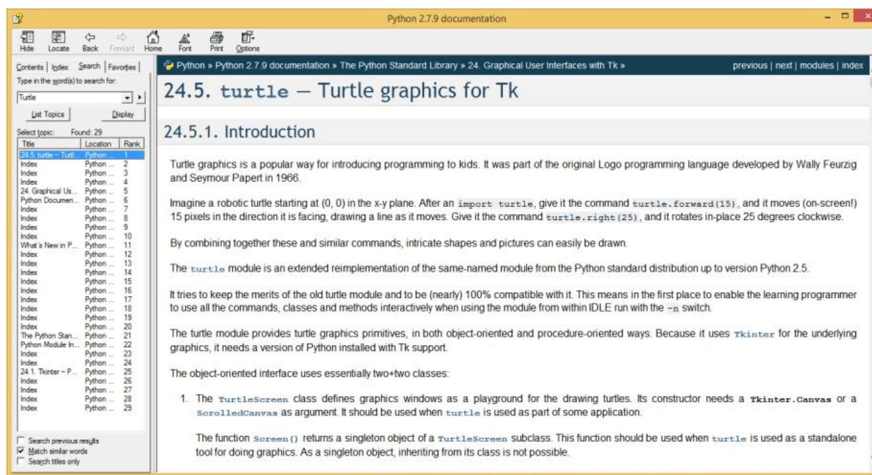



Type Turtle in the Search box and click List Topics or press Enter.

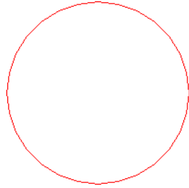
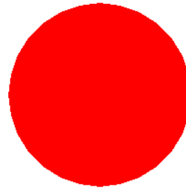
Click on Turtle in the listed topics.




To remove the highlights, click Back button and then Forward button on the top of the screen. Scroll down to see the commands.



**Playing with Colors**

Let's change the color of our circle to Red. Simply command Turtle to choose the color red: **tk.color('red')** as shown below. You may choose a different color such as 'green' or 'blue' but the format requires that you place the name of color in simple quotation marks. To paint the circle inside, bracket the circle command between **begin_fill()** and **end_fill()** commands as shown below. Don't forget to end your program with **tk.exitonclick()**. Hide the Turtle with the command **tk.ht()**.

You will not need to write tk before every Turtle command if you import the Turtle library this way:

```
from turtle import *
```

However, I prefer to keep the **tk.** format as when several libraries are loaded Turtle commands can be separately identified.

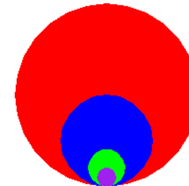
```
import turtle as tk

tk.ht()
tk.color('red')

tk.begin_fill()
tk.circle (100)
tk.end_fill()

tk.exitonclick()
```

Now you've made the Flag of Japan. Well done!

*Exercise 2.* Try making Circles of different sizes and different colors.

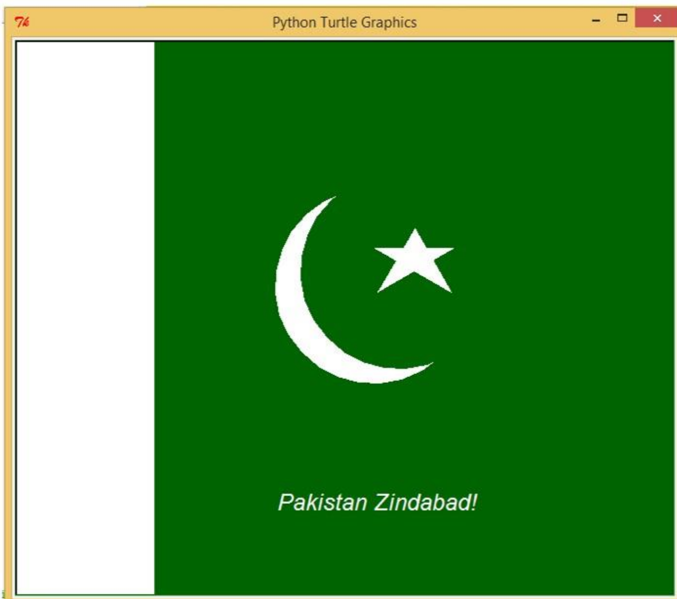
*Exercise 3.* Now paint these circles with various colors.

```
import turtle as tk

tk.ht()

tk.color('red')
tk.begin_fill()
tk.circle (100)
tk.end_fill()

tk.color('blue')
tk.begin_fill()
tk.circle (50)
tk.end_fill()

tk.color('green')
tk.begin_fill()
tk.circle (20)
tk.end_fill()

tk.color('purple')
tk.begin_fill()
tk.circle (10)
tk.end_fill()

tk.exitonclick()
```



From now on, always remember to import turtle module: **import turtle as tk**

*Exercise 4.* Let's make the Flag of Pakistan now.




```
import turtle as tk

tk.reset()
tk.hideturtle()

tk.pu()
tk.setpos(-350, 0)
tk.pd()
tk.bgcolor("dark green")
tk.color("white")

tk.begin_fill()
tk.seth(90)
tk.fd(300)
tk.rt(90)
tk.fd(150)
tk.rt(90)
tk.fd(600)
tk.rt(90)
tk.fd(150)
tk.rt(90)
tk.fd(300)
tk.end_fill()

tk.pu()
tk.home()
tk.fd(75)
tk.rt(90)
tk.fd(50)
tk.lt(90)

tk.lt(30)
tk.pd()
tk.color("white")
tk.begin_fill()
tk.circle(100)
tk.end_fill()

tk.pu()
tk.fd(30)

tk.pd()
tk.color("dark green")
tk.begin_fill()
tk.circle(100)
tk.end_fill()

tk.pu()
tk.home()
tk.color('white')
tk.setpos(30, 30)
tk.lt(60)
tk.pd()

tk.begin_fill()
tk.fd(75)
tk.rt(120)
tk.fd(75)
tk.rt(150)
tk.fd(90)
tk.rt(150)
tk.fd(80)
tk.rt(150)
tk.fd(90)
tk.end_fill()

tk.pu()
tk.setpos(30, -200)
tk.write("Pakistan Zindabad!", False,
align=("center"), font=('Arial', 18, 'italic'))


tk.exitonclick()
```

Python Program by:
Dr. Mohammad Saeed and kids
14th August 2018

*Program:* **Pakistan Flag**

This program will make the Pakistan flag using the turtle module of Python 2

**Column 1**

```
import turtle as tk

tk.reset()
tk.hideturtle()

tk.pu()
tk.setpos(-350, 0)
tk.pd()
tk.bgcolor("dark green")
tk.color("white")

tk.begin_fill()
tk.seth(90)
tk.fd(300)
tk.rt(90)
tk.fd(150)
tk.rt(90)
tk.fd(600)
tk.rt(90)
tk.fd(150)
tk.rt(90)
tk.fd(300)
tk.end_fill()

tk.pu()
tk.home()
tk.fd(75)
tk.rt(90)
tk.fd(50)
tk.lt(90)

tk.lt(30)
tk.pd()
tk.color("white")
tk.begin_fill()
tk.circle(100)
tk.end_fill()
```

**Column 2**

```
tk.pu()
tk.fd(30)

tk.pd()
tk.color("dark green")
tk.begin_fill()
tk.circle(100)
tk.end_fill()

tk.pu()
tk.home()
tk.color('white')
tk.setpos(30, 30)
tk.lt(60)
tk.pd()

tk.begin_fill()
tk.fd(75)
tk.rt(120)
tk.fd(75)
tk.rt(150)
tk.fd(90)
tk.rt(150)
tk.fd(80)
tk.rt(150)
tk.fd(90)
tk.end_fill()

tk.pu()
tk.setpos(30, -200)
tk.write("Pakistan Zindabad!", False,
align=("center"), font=('Arial', 18, 'italic'))


tk.exitonclick()
```

# 3. Q & A

**Let's talk to our Python programs**. Our programs will ask us questions and act on our answers. One of the ways Python programs interact with the users is by the **raw_input** command that allows users to enter **string values** to questions asked by the **program**.

Python processes all user input coming in through the **raw_input** command as text (or string values), even if the values entered by the user are numbers. Therefore, the number values have to undergo special processing prior to being used as numbers in the program.

Numbers are processed in two major ways:

a. Integers (**int**)
b. Decimals (**float**)

We store values (numbers or strings) in **variables** by assigning them to Python commands.

So if in Python Shell:

```
a = 10
b = 'flower'
```

We have stored the number **10** in the variable **a**, and the word **'flower'** in variable **b**. Note that if the word flower is assigned to **b** without quotations Python will assume it to be the name of another variable and generate an error. Therefore you can name your variables anything you want except as designated Python commands. E.g. you can name your variable **a** or **flower** or **a1_sig** but you cannot name a variable **raw_input**, as this is a Python command.

In the following program, we'll draw a circle, however, its **size** and **color** will be determined by the **user**. Note that size (radius) is a number (**integer**) whereas, color is text (**string**). So, we will ask the user to enter their choices and store them in variables. Then use the values stored in the variables to give commands to Turtle.

The format for **raw_input** is that you can place your question in quotation marks 'radius' or "What is the radius of your Circle: " within the brackets as below:

```
raw_input ('radius please ? ')
```

or

```
raw_input ("What is the radius of your Circle: ")
```

However, since the answer by the user is not assigned to a variable, your program will ask the question but not be able to use the answer.

Therefore we will assign the **raw_input** statement to a variable naming it **radius**: (remember variables can be given any name except Python command names)

```
radius = raw_input ('Please enter radius of Circle: ')
```

The user will enter the radius of the circle that Turtle should draw, but this is a number. Your program will recognize the value stored in **radius** as a number only if you tell your program that it is an **integer**. Otherwise Python will consider the number entered by the user as a text and generate an error when processing it.

We make sure our program recognizes the user value for the size of the circle as a number by assigning our variable **radius** as an integer using the **int()** command and assigning it to another variable **rad**:

```
rad = int(radius)
```

We will use the variable rad instead of the variable radius in the Turtle command for circle:

```
tk.circle(rad)
```

Note: Try using the variable radius instead, to see how Python generates an error:

```
tk.circle(radius)
```

Since, color is a string (text) and not a number, it can be processed directly by Python after being assigned to a variable called col_line: **tk.color(col_line)**

## Writing

If we want our program to reply to the user in words, we will ask it to **print** a response. The **print** command is one of the most commonly, versatile and easily used commands of Python.

In Python 2 you can write your statement to print within brackets or without but the statement has to be within quotation marks.

To print to the Turtle screen (canvas) instead, you will need to use the command:

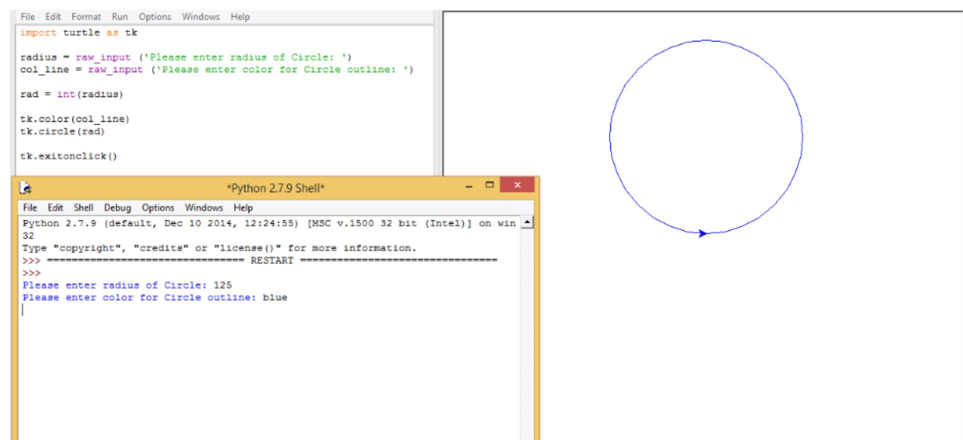
**tk.write ("Circle Drawn")**

Our first conversation with Python as a user:

**radius = raw_input ('Please enter radius of Circle: ')**
**col_line = raw_input ('Please enter color for Circle outline: ')**

**rad = int(radius)**

**tk.color(col_line)**
**tk.circle(rad)**




**print ('The Circle has been drawn. Please click to exit. Thank you')**

**tk.exitonclick()**

Summary:

Assigning Variables, Input (**raw_input** command), Numbers as integers (**int()** command), Processing number and string variables in Turtle commands, Output i.e. Printing text to screen (**print** command).

**Function**

What if we want to teach Python to perform a function, then use it subsequently multiple times?

In this section we'll teach Python to draw a circle and store the code as a **Function** and use it to draw multiple circles. Just like we store **values** in **variables**, we store **code** in **functions**.

We will define a function, using the command **def**, that we will call circ. It will draw a circle with Turtle and use to input variables, r (radius) and c (color).

```
def circ (r, c):

   tk.color(c)

   tk.begin_fill()
   tk.circle(r)
   tk.end_fill()
```

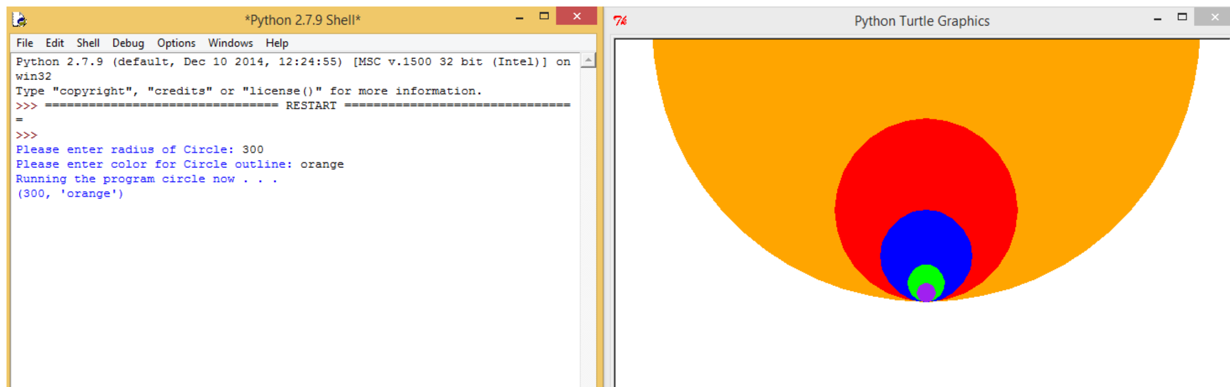
We can now call this function multiple times in our program:

```
circ (100, 'red')

circ (50, 'blue')
```

A function can accept variables such as in the case of circ function that we just defined or it may not. If it is not required to accept variables then the format will be as follows:

```
def sq():
   tk.fd(100)
```

It is important to note the **format** and the **indentations**. When **def** is used it is followed by a function name (can be any text) and ends with **()** and a **:** as above. The code that defines the function is indented. The function ends with de-indentation of the code.




```
import turtle as tk

tk.ht()

radius = raw_input ('Please enter radius of Circle: ')
col_line = raw_input ('Please enter color for Circle outline: ')

rad = int(radius)

def circ (r, c):
   tk.color(c)
   tk.begin_fill()
   tk.circle(r)
   tk.end_fill()

print ('Running the program circle now . . . ')

circ (rad, col_line)

circ (100, 'red')
circ (50, 'blue')
circ (20, 'green')
circ (10, 'purple')

tk.exitonclick()
```

**Choice**

Allowing the program to make a choice is an important programming component. This is made possible using the **if** command.

Suppose we are making a traffic signal at a crossing that responds to a switch controlled by a policeman (the user). If a car is approaching it needs to turn red for the pedestrians and cyclists. If a cycle is approaching then it will turn green. Any other input (e.g. ped) will make it orange. This is a user dependent choice program.

Let's make our signal with two lights. If off they are black. If on, they can be red, orange or green. Hence, the color of the lights are **variables**.

We define our signal as a function of two circles fixed at certain positions in the screen using the command **setpos**.

```
import turtle as tk

tk.ht()

def signal(c, o):
   tk.pu()
   tk.setpos(0, 0)
   tk.pd()

   tk.color(c)
   tk.begin_fill()
   tk.circle(25)
   tk.end_fill()

   tk.pu()
   tk.setpos(50, 0)
   tk.pd()

   tk.color(o)
   tk.begin_fill()
   tk.circle(25)
   tk.end_fill()

c = 'black'
o = 'orange'
signal(c, o)

cross = raw_input ("What's coming, car or
cycle? ")

if cross == 'car':
   o = 'red'
   c = 'black'

elif cross == 'cycle':
   c = 'green'
   o = 'black'

else:
   c = 'orange'
   o = 'orange'

signal(c, o)

tk.exitonclick()
```

Note the indentations. When the signal function ends the indentation ends – last line of **def signal ():** being **tk.end_fill()**.

However, the **if** command has similar indentation. It means that when the condition defined by if is satisfied the code following the **:** that is indented needs to be executed. In the code below, if the user answers car to the question:

```
"What's coming, car or cycle? "
```

```
if cross == 'car':
    o = 'red'
    c = 'black'
```

Then (indicated by the **:** ) the color of the first circle will be chosen as black and the second circle will be red.

Else (**else**) is used when there are more than two choices and is always kept as the last statement. In between choices are made using the statement **elif** (a combination of else and if).

```
else:
    c = 'orange'
    o = 'orange'
```

*Exercise 5.* Run the program above several times to see how the signal changes.

*Exercise 6.* Let's make a Calculator using the commands in this chapter.

Note: Division is Python 2 is deficient when it comes to decimals (**float**). Therefore, you need to import it from Python 3 with the command:

```
from __future__ import division
```

(This must be the first line in your code)

*Program:* Basic Calculator

This program performs addition, subtraction, multiplication and division on user input data like a calculator

**Column 1**

```
print "Calculator"
print ''
print 'Add, Subtract, Multiply, Divide'
print ''

def Add():
   aa = raw_input ('Please enter first
number for addition: ')

   bb = raw_input ('Please enter second
number for addition: ')

   a = int(aa)
   b = int(bb)

   add = a+b

   print aa, '+', bb, '=', add

def Subtract():
   ss = raw_input('Please enter first
number for subtraction: ')

   dd= raw_input('Please enter second
     number to subtract: ')

   s=int(ss)
   d= int(dd)

   sub = s-d

   print s, '-', d, '=', sub

def Multiply():
   mm = raw_input ('Please enter first
number to multiply ')
```

**Column 2**

```
   nn = raw_input ('Please enter second
number to multiply ')

   m= int(mm)
   n= int(nn)

   mult = m*n

   print m, 'x', n, '=', mult

def Divide():
   yy = raw_input ('Please enter number to
divide ')

   xx = raw_input ('Please enter number to
divide by ')

   y = float(yy)
   x = float(xx)

   div = y/x
   print y, '/', x, '=', div

# Main calculate
q = raw_input('a) Add b) Subtract c)
Multiply d) Divide: ')
print ''
if q == 'a':
   Add()
if q == 'b':
   Subtract()
if q == 'c':
   Multiply()
if q == 'd':
   Divide()
print ''
```

# 4. Cycling

Often we want our programs to perform repetitive tasks. In fact that is one of the most useful aspects of programming. These cyclical processes are called **loops**.

To make a loop, the **for** statement is used. Instead of repeatedly running our circle program (circ) or individually calling our circ function as below, we can run a **for** loop.

```
circ (100, 'red')
circ (50, 'blue')
circ (20, 'green')
```

How many times should the loop be run? This is determined by a process called **iteration**. Iterations can be done in a number of ways, however, one of the easiest is to use the command, **range**.

In Python Shell, execute the command **range** as below:

```
>>> range (5)
[0, 1, 2, 3, 4]
>>>
```

This creates a **list** of numbers 0 to 4, shown within square brackets. This has led us to a very important concept called a **list**. It is a list, contained with [ ], that is iterable. We can iterate the list created by range using the **for** command:

```
for i in range (3):
```

The colon **:** represents *then* – i.e. run the following indented code **3** times. We have called **i** the variable (can be any text) in which the value of the items in the list are stored. The Python command **in** allows the **for** loop to access the list.

```
for i in range (3):
   circle(rad, col_line)
```

We can understand the iteration of a list better with the following code executed in Python Shell:

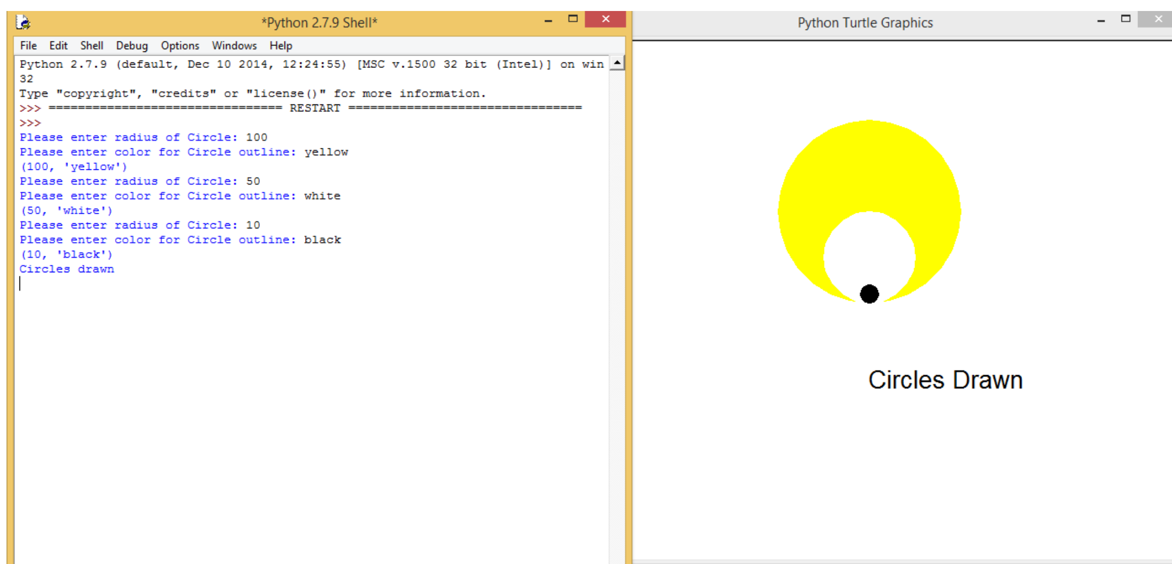
First, we create a list using the command **range**. This list called **i** contains **3** items. If we print the list **i** it shows its contents.

```
>>> i = range(3)
>>> print i
[0, 1, 2]
```

Then we run the **for** loop to extract each item individually and **print** each item (value).

```
>>> for value in i:
        print value

0
1
2
>>>
```

For our circ program we want that the user enters three values of radius and color. These values are then passed on to our circ function which then draws the required circle. In the end, the program **print**s 'Circles drawn' in the Python Shell area as well as **write**s it on the Turtle canvas.



Here’s the entire code for our program above:

```
import turtle as tk

tk.ht()

def circ (r, c):
   tk.color(c)
   tk.begin_fill()
   tk.circle(r)
   tk.end_fill()


for i in range (3):
   radius = raw_input ('Please enter radius of Circle: ')
   col_line = raw_input ('Please enter color for Circle outline: ')

   rad = int(radius)

   circ (rad, col_line)

   print (rad, col_line)

print ('Circles drawn ')

tk.pu()
tk.setpos(0, -100)

tk.write ("Circles Drawn", font=("Arial", "20"))

tk.exitonclick()
```

### Whiling away

Another way of running a loop is by using the **while** command:

In the example below we define a variable **a** and assign it a value **10**. Then we create a loop using **while** and subtract 1 from it until the value becomes **0**. The while loop will run as long as **a > 0** (per our code).

```
>>> a = 10
>>> while a>0:
        print a, 'continue to run'
        a = a – 1
```

(Note: a = a – 1 can also be written as a -= 1)

```
10 continue to run
9 continue to run
8 continue to run
7 continue to run
6 continue to run
5 continue to run
4 continue to run
3 continue to run
2 continue to run
1 continue to run
>>>
```

### Randomization

If we want our programs to make random choices we will need to **import** the **random** module.

```
>>> import random as rnd
```

If we want to choose a random number for a list of numbers from 0 to 100, we can do so with the following command:

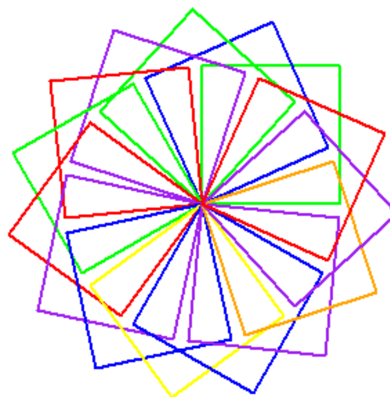
```
>>> i = range (100)

>>> rnd.choice(i)
```

To iterate this function to select 10 numbers from this list of 100 randomly, we can execute the following code in Python Shell:

```
>>> for no in range(10):
        a = rnd.choice(i)
        print a

95
37
60
41
24
22
17
70
58
32
>>>
```

Summary: After the Polygon programs below, you will be familiar with several important Python commands including:

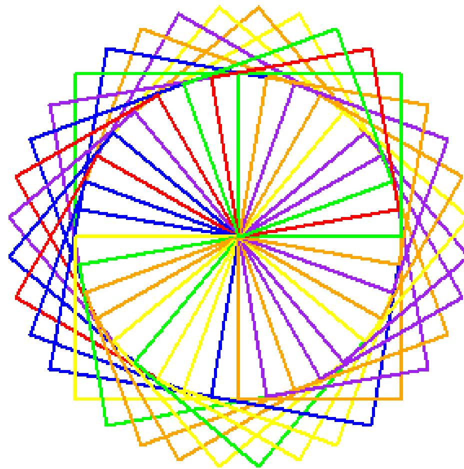
**for, if, else, elif, range, in, while, random, return, global, from, import, def. You have also learnt the important concepts of functions and list.**



*Program:* Polygons

This program will make a square of randomly selected colors (from a list) and rotate it to make a polygonal design

```
from __future__ import division

import turtle as tk
import random as rnd

sq_no = raw_input ('How many squares do you want? ')
sqn = int(sq_no)

ang = 360 / sqn

cl = ['blue', 'green', 'red', 'yellow', 'orange', 'purple']

tk.pensize(2)
tk.hideturtle()

def sq():
    for side in range(4):
        tk.fd(100)
        tk.lt(90)

for i in range(sqn):

    co = rnd.choice(cl)
    tk.color(co)

    sq()
    tk.left(ang)


tk.exitonclick()
```



In the program, Polygons the **random.choice()** command iterates over the color list cl and chooses a color randomly. However, it can still choose the same color in the next iteration, resulting in two or more consecutive squares being of the same color.

To prevent the program from using the same color consecutively we will use two new commands **global** and **return**. They are not directly involved in allowing the program to choose but they make unique loops possible.

We make a function to choose the color randomly, **rnd_color()**. First, we assign global variables current pen color (**pco**) and chosen color (**co**). These variables are listed in the main program and imported into the **functions** using the command **global**. They are processed in the function and their new values are reported back to the main program using the command **return**.

Explanation of code as # after the command:

```
pco = '' # global variable current pen color
co = ''   # global variable chosen pen color

def rnd_color():   # define the function to choose a color randomly
  global pco, co   # import the global variables in to the function

  co = rnd.choice(cl)  # make a random color choice

  if co != pco:      # if the color chosen is different from previous color then ...
    pco = co         # choose this color for the next square and store it as pco for the next round

  else:              # if the color chosen is the same as previous square then...
    for i in range (5):      # create a loop of 5 iterations
      c = rnd.choice(cl)  # i.e. repeat the random choice 5 times
      if c != pco:           # if a color is different from pco then ...
        co = c               # choose this color

  pco = co                   # make this new chosen color as the new stored value for pco
  return co, pco             # update the color choices in the main program
```

*Program:* Polygons - 2

```
from __future__ import division       # always the first line in code

import turtle as tk
import random as rnd

sq_no = raw_input ('How many squares do you want? ')
sqn = int(sq_no)

ang = 360 / sqn

cl = ['blue', 'green', 'red', 'yellow', 'orange', 'purple']

tk.pensize(2)
tk.hideturtle()

pco = ''
co = ''

def rnd_color():
    global pco, co

    co = rnd.choice(cl)

    if co != pco:
        pco = co

    else:
        print '=', pco, co
        for i in range (5):
            c = rnd.choice(cl)
            if c != pco:
                co = c

    pco = co

    return co, pco
```

```
def sq():
    for side in range(4):
        tk.fd(100)
        tk.lt(90)


def polygon():
    global co

    for i in range(sqn):
        rnd_color()
        tk.color(co)
        sq()
        tk.left(ang)

polygon()

tk.exitonclick()
```

Further programming help is easily available online (1-3).

# References

1. Python Help (F1) docs
2. https://stackoverflow.com/questions
3. https://www.codegrepper.com/code-examples/python

# II. Gaming

# 5. Game Board

Making a game in Python is fun and a great way to practice your programming skills. In this chapter, we will go over the basics of making a game: how we can transform Turtle from drawing mode to moving objects across the screen.

Principle 1. Objects moving on the screen are all Turtles

We will first make three turtles on the screen. This is done using the command: **tk.Turtle()**

Note: Turtle here is spelled with capital **T** whereas when the turtle module is imported it is spelled with simple **t**.

Position the 1st turtle on the right, the 2nd on the left and leave the 3rd in the center.

```
tk1 = tk.Turtle() # 1st turtle
tk1.pu()
tk1.fd(450)

tk2 = tk.Turtle() # 2nd turtle
tk2.pu()
tk2.fd(-450)

tk3 = tk.Turtle() # 3rd turtle
```

Now, we'll set up the game board called the **Screen()** and assign it to a variable win (for window). This is done prior to setting up the **Turtles()**.

```
win = tk.Screen()
win.setup(width=500, height=300) # The game window will be 500 x 300 pixels
```

**Shaping up your Turtle**

You can change the shape of your turtles using the command: tk.shape(). Following shapes are already available as part of turtle module: "arrow", "turtle", "circle", "square", "triangle", "classic".

Hence, if you want to change the shape of your turtle to a circle, you'll type:

**tk.shape('circle')**

In the current program where we have just made three turtles (tk1, tk2 and tk3), we'll change the shape of tk1 and tk2 to square and tk3, in the center, to a circle.

```
tk1.shape('square')
tk2.shape('square')
tk3.shape('circle')
```

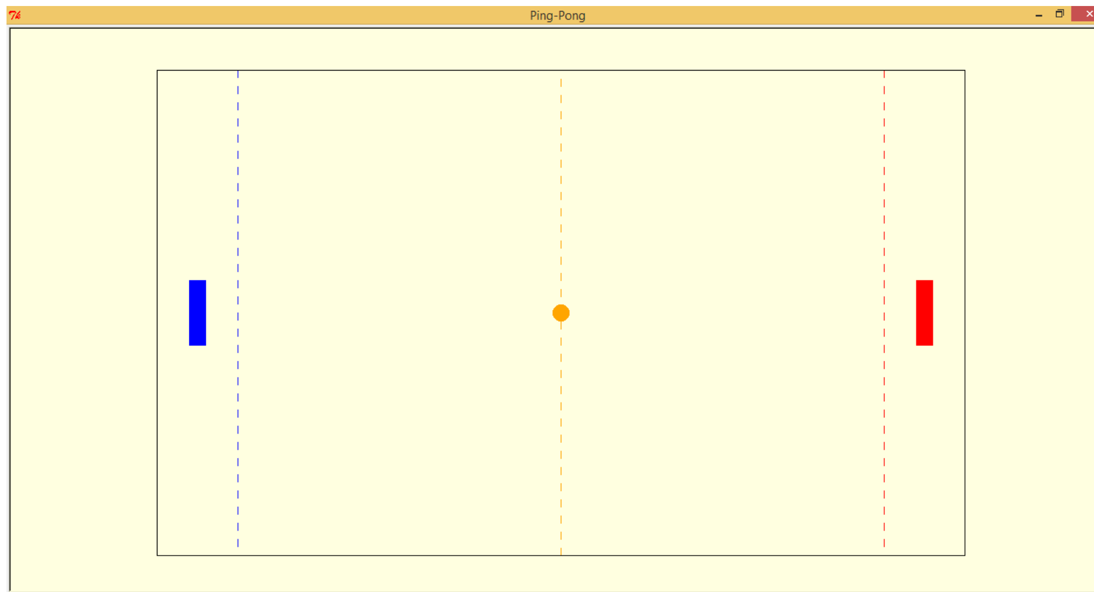
Now, we'll convert tk1 and tk2 into Ping-Pong bats by stretching the squares into rectangles, using the **shapesize()** command:

```
tk1.shapesize(4, 1, 1) # it takes 3 values: height, width and outline
```

Here's the complete code of the classic 'Pong' game board:

```
import turtle as tk

win = tk.Screen()
win.setup(width=500, height=300)

tk1 = tk.Turtle()
tk1.pu()
tk1.fd(450)
tk1.shape('square')
tk1.shapesize(4, 1, 1)

tk2 = tk.Turtle()
tk2.pu()
tk2.fd(-450)
tk2.shape('square')
tk2.shapesize(4, 1, 1)

tk3 = tk.Turtle()
tk3.shape('circle')

tk.exitonclick()
```

*Program:* Ping-Pong




```
import turtle as tk

# Setting the play screen
win = tk.Screen()
win.title('Ping-Pong')
win.setup(width=500,
height=300)
win.bgcolor('light yellow')

# Right Bat
tk1 = tk.Turtle()
tk1.color('red')
tk1.pu()
tk1.fd(450)
tk1.shape('square')
tk1.shapesize(1, 4, 1)

# Right Border
tk1.pu()
tk1.goto(400, 0)
tk1.setheading(90)
tk1.fd(300)

for i in range(30):
    tk1.pd()
    tk1.bk(10)
    tk1.pu()
    tk1.bk(10)

tk1.pu()
tk1.goto(450, 0)

# Left Bat
tk2 = tk.Turtle()
tk2.color('blue')
tk2.pu()
tk2.fd(-450)
tk2.shape('square')
tk2.shapesize(1, 4, 1)

# Left Border
tk2.pu()
tk2.goto(-400, 0)
tk2.setheading(90)
tk2.fd(300)

for i in range(30):
    tk2.pd()
    tk2.bk(10)
    tk2.pu()
    tk2.bk(10)

tk2.pu()
tk2.goto(-450, 0)

# Ball
tk3 = tk.Turtle()
tk3.color('orange')
tk3.shape('circle')

# Center line
tk3.pu()
tk3.rt(90)
tk3.fd(300)

for i in range(30):
    tk3.pd()
    tk3.bk(10)
    tk3.pu()
    tk3.bk(10)

tk3.pu()
tk3.home()

# Completing the Table
tk3.pu()
tk3.color('black')
tk3.goto(-500, 300)
tk3.pd()
tk3.fd(1000)
tk3.rt(90)
tk3.fd(600)
tk3.rt(90)
tk3.fd(1000)
tk3.rt(90)
tk3.fd(600)

tk3.pu()
tk3.color('orange')
tk3.home()

tk.exitonclick()
```

# 6. Moving Objects

It’s fascinating to see an object move on screen. In the Ping-Pong game that we are trying to create we’ll first try and move the bats when a key is pressed.

**Key to Move**

Coding keys to move objects e.g. Ping-Pong bats on the screen.

```
# Setting the position of right and left bats at 0

ry, ly = 0, 0


# Defining the move function for the right bat

def r_bat_up():
   global Ry
   ry = tk1.ycor()
   ry += 20
   tk1.sety(ry)
   return ry


# Listen for key presses

win.listen()
win.onkey(r_bat_up, 'Up')
```

*Exercise 7.* Write the code for moving the left bat up. Hint: Assign a separate set of keys to the left bat e.g. 'e' and ‘x’ such as: `win.onkey(l_bat_up, 'e')`

*Exercise 8.* Write the code for moving the right and left bats down. Hint: If moving the bat up requires addition of 20 units, moving it down will require subtraction of 20 units from its current position: `ry -= 20`

Now that you have made both the Ping-Pong bats move up and down as you press the respective keys, let’s make the ball move.

**Auto-Move**

Coding objects to move on the screen automatically.

This is another level of challenge. It involves randomizing the position of moving object and setting its direction and speed. Finally, moving objects on the screen have to interact with each other to allow the game to be played. What happens when the ball collides with a bat? And what happens when it doesn't?

So, let's build the narrative of the movement of the ball on the Game Board.

1. Ball starts at the center (position 0, 0)
2. It moves in a direction with a set speed
3. Ball collides with the top / bottom border of the Board as is reflected
4. Bat misses ball, the ball goes off the Board and restarts at center
5. Ball collides with bat and is returned to the opposite direction

```
# Setting the position of ball at 0, 0

bx, by = 0, 0


# Setting the initial direction and speed of ball

dx, dy = -5, 5  # moves 5 points up and to the left


# Defining the function for the movement of the ball

def ball_move():
   global bx, by, dx, dy
   bx = bx + dx  # key code to calculate the movement of the ball horizontally
   by = by + dy  # key code to calculate the next ball position vertically

   tk3.goto(bx, by) # actually moves the ball in the calculated direction

   win.ontimer(ball_move, 5) # updates the position of the ball every 5ms
   return bx, by    # updates the calculations of the ball position
```

# Note the key statements above. The main code that runs the game is win.ontimer [win = turtle.Screen()]. The function that has to be continuously updated has to be called by the win.ontimer. Frequency of update is the second variable in milliseconds. However, it must be noted that when a function is called within the ontimer the **()** are **not** included.

So far the code has solved the first two points of the ball movement narrative. Let’s add code to allow the ball to reflect (travel in opposite direction with same speed) when it hits the top or bottom borders of the Game Board.

The height of our Game Board is 300 (and its depth from center is -300). Given the ball has a radius of 10 points it should reflect when its center reaches position +290 or -290. Adding the following code to our function **ball_move()** should do the trick.

```
if by >= 290:
    dy *= -1

elif by <= -290:
    dy *= -1
```

Essentially, the ‘speed’ in the y-direction is multiplied by -1 to reflect the ball vertically. What should be done to reflect the ball off the bat in the horizontal direction?

The obvious answer is:

```
dx *= -1
```

However, it is easier to code for the ball being missed by the bat and going off the Game Board than to code a hit. So, we will write the code for this move first.

If the ball goes beyond the thickness of the bat then it should restart.

```
if bx >= 460 or bx <= -460:
    win.tracer(0) # makes the turtle (ball) disappear

    bx, by = 0, 0 # resets turtle (ball) position to center

    tk3.goto(bx, by) # send the turtle (ball) to center
    win.tracer(True) # makes the turtle (ball) re-appear
```

The bat hitting the ball is a bit more complicated. Dimensions of the bat are distinctly different from the Game Board borders. It has a thickness and has a length and is a distance away from the border. Moreover, its position changes in the vertical direction.

The ball must strike the bat within its vertical borders to be considered 'hit' and therefore reflected. The y-position of the center of the ball (**by**) has to be between the top and bottom borders of the bat. Since the bat is 4 times the size of the ball a height / depth of 40 should work.

```
if by >= ry - 40 and by <= ry + 40:
    dx *= -1
    bx = 430


elif by >= ly - 40 and by <= ly + 40:
    dx *= -1
    bx = -430
```

The horizontal reflection off the bat is the most challenging to handle. This code requires a bit of experimentation due to the thickness (x-dimensions) of the bat and its distance from the Board border. If not done properly the ball will bounce repeatedly within the bat thickness or get repeatedly reflected behind the bat.

```
if (bx > 430 and bx <= 455) and (by >= ry - 40 and by <= ry + 40):
    dx *= -1
    bx = 430


elif (bx <-430 and bx >= -455) and (by >= ly - 40 and by <= ly + 40):
    dx *= -1
    bx = -430
```

Hence, our final code for the movement of the ball should look like:

```
# Setting the position of ball at 0, 0
bx, by = 0, 0

# Setting the initial direction and speed of ball
dx, dy = -5, 5  # moves 5 points up and to the left
```

```
def ball_move():

   global bx, by, dx, dy, ry, ly
   bx = bx + dx
   by = by + dy

# Ball hits bat

   if (bx > 430 and bx <= 455) and (by >= ry - 40 and by <= ry + 40):
      dx *= -1
      bx = 430


   elif (bx <-430 and bx >= -455) and (by >= ly - 40 and by <= ly + 40):
      dx *= -1
      bx = -430

# Ball is missed and goes off the Game Board

   elif bx >= 460 or bx <= -460:
      win.tracer(0)
      bx, by = 0, 0
      tk3.goto(bx, by)
      win.tracer(True)

# Ball hits the top or bottom of the Game Board

   if by >= 290:
      dy *= -1
   elif by <= -290:
      dy *= -1


# Keep the ball moving

   tk3.goto(bx, by)

   win.ontimer(ball_move, 5)


# Update position of the ball

   return bx, by, ry, ly
```

### Play time

It's time to play Ping_Pong. Call the function you wrote, **ball_move()** to start the game. Make sure you call the function **exitonclick()** to end the game.

```
# Play

ball_move()


# End Game

tk.exitonclick()
```

### Score Board

What the fun without a score board! The opposite player gets a point when a bat misses the ball.

```
# Setting up the Score board

# Initialize the score
left_score = 0
right_score = 0
```

Since the score is another changing element on the Game Board, it has to be a Turtle. Writing on screen is done using the **turtle.write()** command.

```
# Displays the score

sketch = tk.Turtle()
sketch.speed(0)
sketch.color("orange") # We choose to write the score in orange color
sketch.penup()
sketch.hideturtle()
sketch.goto(0, 260) # Position the score board on the screen

# Now write the initial score on the screen

sketch.write("Left Player : 0         Right Player: 0", align="center", font=("Calibri", 14, "bold"))
```

Scores are updated inside the **ball_move()** function under the section ‘Ball is missed and goes off the Game Board’.

```
# Ball is missed and goes off the Game Board

  elif bx >= 460 or bx <= -460:
    win.tracer(0)

    # Update scores
    if bx >=460:
      left_score += 1
    elif bx <= -460:
      right_score += 1

    sketch.clear()
    sketch.write("Left Player : {}        Right Player : {}".format(left_score, right_score),
align="center", font=("Calibri", 14, "bold"))

    bx, by = 0, 0
    tk3.goto(bx, by)
    win.tracer(True)
```

### Winner

The score decides the winner of the game. This requires an **if** and an **else** statement. If the score reaches, say 10, then end the game [**tk.clickonexit()**] else continue updating the ball position.

```
# Keep the ball moving --- unless a player reaches the maximum allowed score

  if left_score == 10 or right_score == 10:
    sketch.color("black")
    sketch.penup()
    sketch.hideturtle()
    sketch.goto(0, 0)
    sketch.write("GAME OVER", align="center", font=("Calibri", 24, "bold"))
    tk.exitonclick()

  else:
    tk3.goto(bx, by)
    win.ontimer(ball_move, 5)
```

# 7. Sights and Sounds

Graphics and sounds make any game more interesting. Once you have made the game adding graphics and sounds is fun and easy.

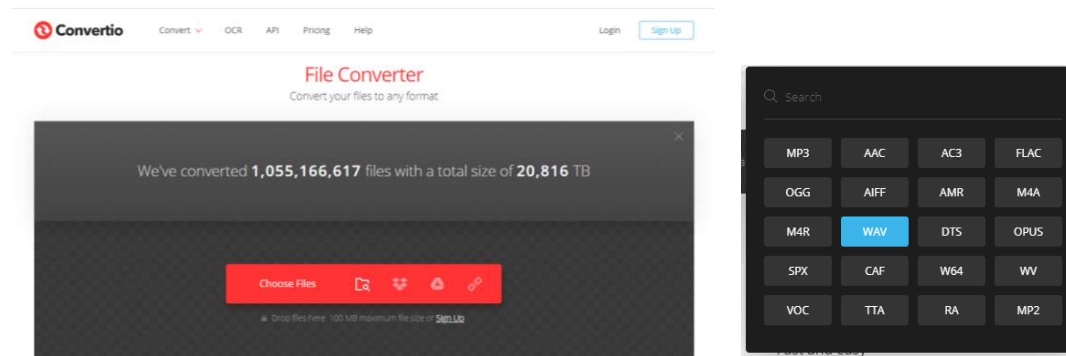
### Musica

The **winsound** module incorporated in Python can provide a loads of music to your game.

**import winsound as snd**

We will add background music to the game so that while the game is being played the music keeps the players entertained.

The **Playsound()** command allows a music file to be played in the background. The music file has to be in the WAV format. You can record any music using the in-built Windows Sound Recorder. However, this will generate a Windows Media Audio file (.wma). Python needs .wav files. So, you will have to convert the .wma file into .wav file. This can easily be done online using the website: https://convertio.co/




When you have uploaded your recorded .wma file you'll be given the option of choosing the format to convert to. Choose WAV (highlighted blue above).

I recorded the Ertugrul theme music and play it in the game using the code:

**snd.PlaySound('Ertugrul.wav', snd.SND_LOOP+snd.SND_ASYNC)**

**SND.LOOP** plays the sound file repeatedly. The **SND_ASYNC** flag must also be added to avoid blocking. Please note the format of the command above. Add this command in the **# Play** section above **ball_move()**.

Background art is added using the bg.pic command. This adds .gif files only. If you cannot find a .gif file use any picture .jpg or .png and convert it .gif using https://convertio.co/

```
# Add background to the section
## Setting the play screen
win.bgpic('sky.gif')
```

Change the shape of the Turtles acting as 'bats' to birds. Download two bird gif files and assign one to each bat function

```
win.register_shape('bird.gif') # remember win is not a command. win = turtle.Screen()
tk1.shape('bird.gif')
```

```
# turn off these two commands by commenting out
#tk1.shape('square')
#tk1.shapesize(1, 4, 1)
```

Therefore the Right bat definition will be as follows:

```
# Right Bat
win.register_shape('bird.gif')
tk1 = tk.Turtle()
tk1.color('red') # you may comment out this line also
tk1.pu()
tk1.fd(450)
tk1.shape('bird.gif')
#tk1.shape('square')
#tk1.shapesize(1, 4, 1)
```

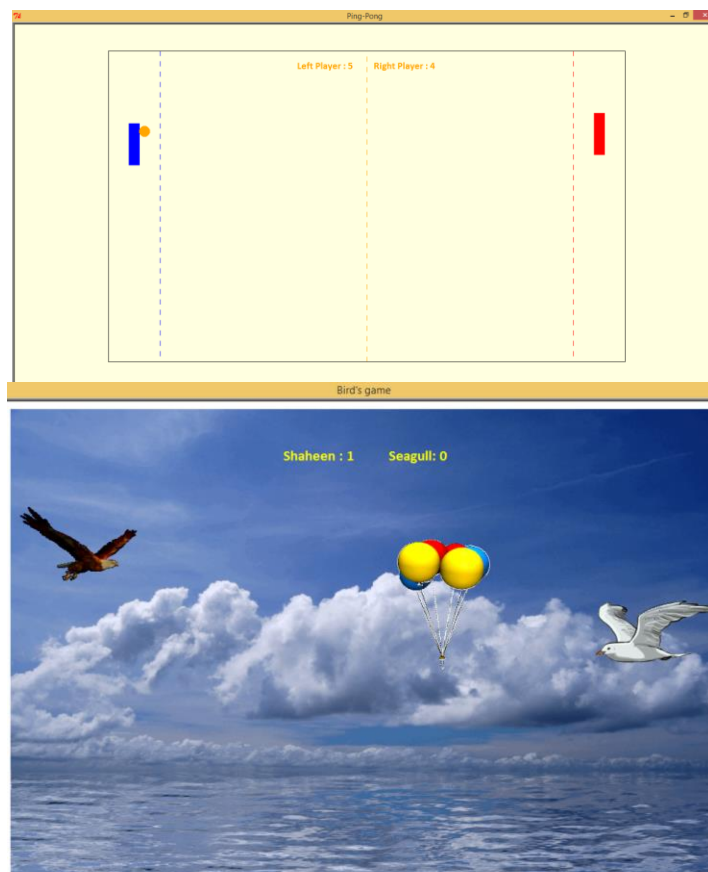
*Exercise 9.* Write the code for changing the Left bat into a bird such as an eagle. Hint: Remember to download a .gif file, register the shape and then assign it to tk2.

*Exercise 10.* Write the code to change the ball into balloons

*Exercise 11.* Comment out the correct lines in the Game Board to eliminate the Ping-Pong Table. Hint: see code in Chapter 5.

*Exercise 12.* To make the score more meaningful, the bird that 'pops' the balloons (Hint: the bat that hits the ball) should get a point. Change to the code to allow this to happen. Hint: Move the scoring lines to where the bat hits the ball instead of where the bat misses the ball. See also Object oriented programming (1).

Now, your Ping-Pong game will morph into a bird fight in the sky with background music. Further help in making games is available online (2, 3).

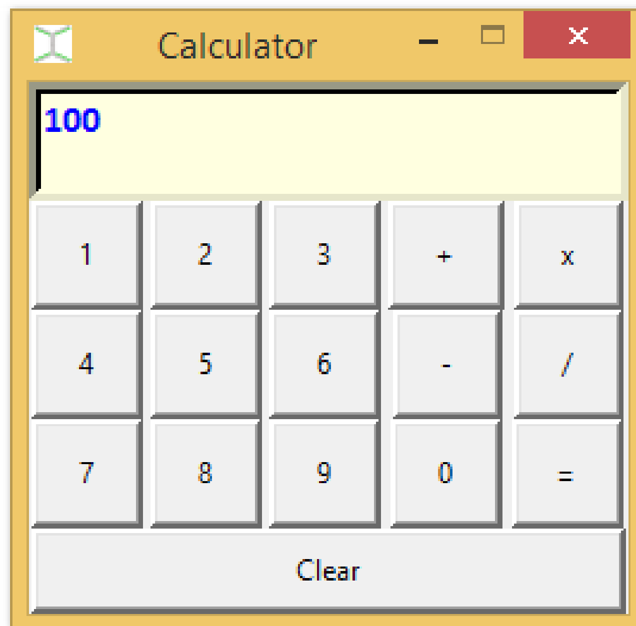


## References


1. OOP: https://www.youtube.com/watch?v=JeznW_7DlB0
2. https://www.youtube.com/watch?v=XGf2GcyHPhc
3. https://www.youtube.com/watch?v=jO6qQDNa2UY

# III. GUI

## Graphic User Interface



# 8. Tkinter

One of the most interesting developments in computing was the introduction of Macintosh by Apple. It was a point and click operation for desktop computing, revolutionizing its access to the common person. Prior to that, one had to learn a lot of commands to do work on the computer *(which I actually did and find it still useful)*. Since 1990s Windows by Microsoft led the **Graphics User Interface (GUI)** market starting with its blockbuster, Windows 3.0 *(and this caused me to be left behind in the skyrocketing programming world until I discovered Python. This however was not the only reason. I guess the main reason was my medical college prep and entry – at that time Medicine was quite aloof from computing; not so now)*. The module that allows GUI development in Python is called Tkinter (often pronounced *kinter*, with T silent). In this section, we will develop a GUI based Calculator using Tkinter and then see how we can use Tkinter (1, 2) and Turtle together.

Just like the Turtle module had to be imported we have to import Tkinter. At this point I'd caution you that there are certain critical differences in the module between Python 2 and Python 3 versions. The Python 3 compatible versions are more advanced and useful however we'll stick to Python 2 for now.

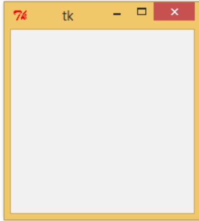
```
from Tkinter import *
```

Then comes the two most important lines of code between which the entire GUI program works.

```
root = Tk()  # line 1
 . . . . Entire GUI program will be here
root.mainloop() # last line
```

Essentially, what these two lines of code do is define a GUI window called 'root' and then the mainloop() command keep the window running, waiting for any buttons to be pressed, any text to be written etc.

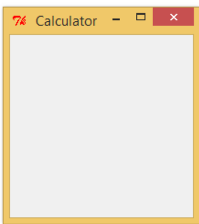
If only these two commands are given the result will be the following:




So let’s first given our program a title. Remember all lines of code will be written between the first and the last lines.

```
root.title('Calculator')
```

So the code and the result will be as follows:




```
from Tkinter import *

root = Tk()

root.title('Calculator')

root.mainloop()
```

### Buttons

Now, let’s make our first button. The ***Button*** widget consists of at least 4 components: a) which window to place the button in, b) the text written on the button, c) its dimensions and d) the command that dictates its operations. We will create a button for the number 1 and call it *N1* and define its command as *N1_press*. Next, we will place the button in its place on the Calculator using the ***grid()*** system. Define the N1_press command first without any function.

```
def N1_press():
   return
N1 = Button (root, text = '1', padx = 15, pady = 10, borderwidth = 3, command = N1_press)
```

The padding of the button determines its size. You can play with padx and pady to see how the button changes in dimensions.

Now, let's place the button on the Calculator in row 2 and 1st column (note the first column is called zero).

```
N1.grid (row = 2, column = 0)
```

Though the button was placed in row 2 by the grid system it appears in row 1. The reason is that there is nothing in row 1 and the grid system is a relative system. It places things not on an x- / y- coordinate system but in rows and columns relative to other items in the grid. This is what makes it challenging to use. You will notice this in setting the dimensions of the buttons and text boxes.

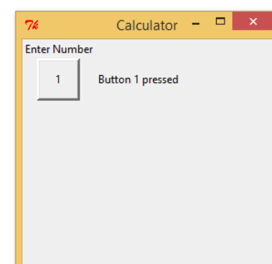
## Display

To display text in the root window several ways can be used. We will go over ***Labels***, ***Entry*** and ***Text*** widgets. You'll see the differences between them.

```
# Label (*Note the Capital L)

label1 = Label (root, text = 'Enter Number')
label1.grid (row = 0, column = 0)
```

We can code our button for number 1 to 'print' something in the Calculator window using the widget, Label.

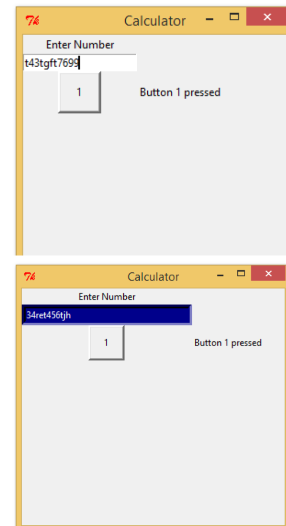
```
def N1_press():
   Label (root, text = 'Button 1 pressed').grid (row=2, column=1)
   Return
```



```
# Entry (*Note the Capital E)

En1 = Entry (root).grid (row = 1, column = 0)
```

So far the code and its results is as follows:

```
from Tkinter import *
root = Tk()
root.title('Calculator')

def N1_press():
   Label (root, text = 'Button 1 pressed').grid(row=2, column=1)
   return

label1 = Label (root, text = 'Enter Number')
label1.grid (row = 0, column = 0)

En1 = Entry (root).grid (row = 1, column = 0)

N1 = Button (root, text = '1', padx = 15, pady = 10, borderwidth =
3, command = N1_press)
N1.grid (row = 2, column = 0)
root.mainloop()
```

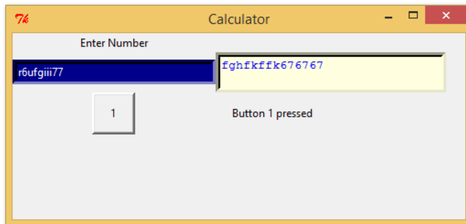



Anything can be typed directly from the keyboard into the Entry widget. It is free text. To get the blue Entry screen modify the code as:

```
En1 = Entry (root, width = 35, bg = 'dark blue', fg = 'white', borderwidth = 5).grid (row = 1,
column = 0)
```

Background color is bg, foreground color is fg. You may observe the 3D effects of the Entry widget by increasing the borderwidth = 25.

```
# Text (*Note the Capital T)

Tx1 = Text (root, width=30, height = 2, bg = 'light yellow', fg = 'blue', borderwidth =
5).grid(row = 1, column = 1, columnspan = 2)
```




'columnspan' allows the widget to extend over multiple columns. Its effects will be truly noticed when multiple widgets are present in the root window.

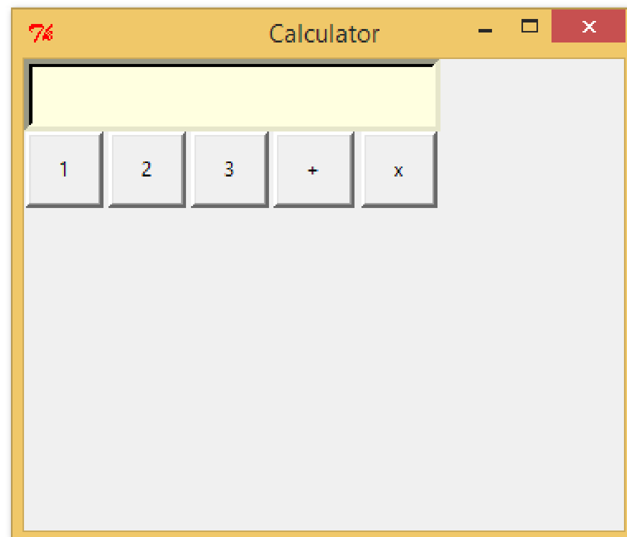
The ***Text*** widget is similar to ***Entry*** in that, free text can be written. However, the Text widget accepts multiple lines followed by an Enter whereas the Entry widget accepts only a single line. Also the ***Text*** widget can be used for <u>both input and output</u> whereas the Entry widget is only for input. Hence, we will use the Text widget as our Calculator screen, call it totals (instead of Tx1) and columnspan = 5 (there will be 5 buttons in each row i.e. 5 columns).

We will comment out the Label and Entry widget commands (for now and delete them later).

So far the code and its results is as follows:

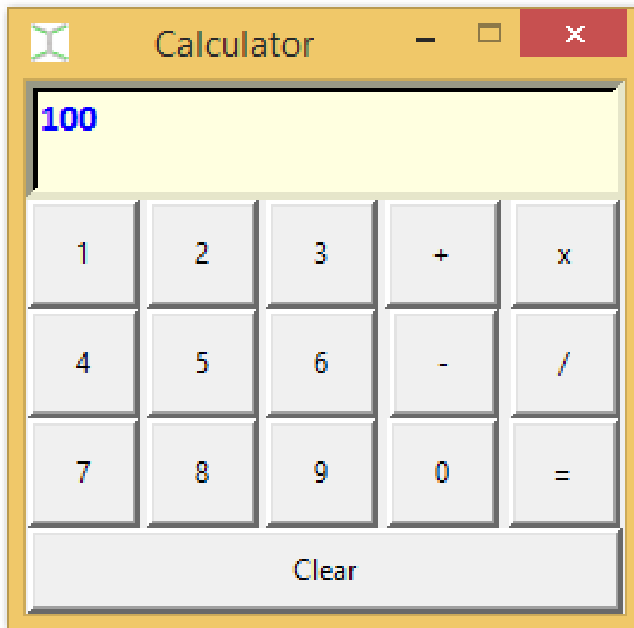
```
from Tkinter import *
root = Tk()
root.title('Calculator')

def N1_press():
   return

##label1 = Label (root, text = 'Enter Number').grid (row = 0, column = 0)
##En1 = Entry (root, width = 35, bg = 'dark blue', fg = 'white', borderwidth = 5).grid (row = 1,
column = 0)

totals = Text (root, width=30, height = 2, bg = 'light yellow', fg = 'blue', borderwidth =
5).grid(row =0, column = 0, columnspan = 5)

N1 = Button (root, text = '1', padx = 15, pady = 10, borderwidth = 3, command = N1_press)
N1.grid (row = 2, column = 0)
root.mainloop()
```

Let's press ahead with rapidly coding our calculator and making all the buttons:




```
from Tkinter import *

root = Tk()

root.title('Calculator')

def N1_press():
   return

def N2_press():
   return

def N3_press():
   return

def Nadd_press():
   return

def Nmul_press():
   return

totals = Text (root, width=30, height = 2, bg = 'light yellow', fg = 'blue', borderwidth =
5).grid(row =0, column = 0, columnspan = 5)

N1 = Button (root, text = '1', padx = 15, pady = 10, borderwidth = 3, command = N1_press)
N1.grid (row = 2, column = 0)

N2 = Button(root, text = '2', padx = 15, pady = 10, borderwidth = 3, command = N2_press)
N2.grid(row = 2, column = 1)

N3 = Button(root, text = '3', padx = 15, pady = 10, borderwidth = 3, command = N3_press)
N3.grid(row = 2, column = 2)

Nadd = Button(root, text = '+', padx = 15, pady = 10, borderwidth = 3, command =
Nadd_press)
Nadd.grid(row = 2, column = 3)

Nmul = Button(root, text = 'x', padx = 15, pady = 10, borderwidth = 3, command =
Nmul_press)
Nmul.grid(row = 2, column = 4)

root.mainloop()
```

*Exercise 13.* Write the code for the rest of the Calculator buttons (4 to 9, 0, -, /, = and clear) as shown in the picture below:




To make the Clear button increase *padx* and use *columnspan* argument:

```
Clr = Button (root, text = 'Clear', padx = 106, pady = 5, borderwidth = 3)
Clr.grid(row = 5, column = 0, columnspan = 5)
```

**Fonts**

Configure the Text widget, totals using a tuple. A ***Tuple*** is used to store multiple items in a single variable in an ordered manner that is unchangeable (unlike a List) – see Chapter 4. Cycling for basic description of List.

```
font_tuple = ("Calibri", 12, "bold")
totals.configure (font = font_tuple)
```

**Functioning Buttons**

The Calculator essentially takes two numbers and applies a function to them. E.g. Add 9 and 3. Then the calculator displays the answer of this function i.e. 12.

This requires us to store two input numbers in memory, apply a function and display a result number. We also need to store the number of the button pressed, e.g. if 7 is pressed, it should be displayed on the ***Text*** screen. We can achieve this using ***global*** variables.

```
# Globals

num = 0            # button pressed
func = ''          # function
fst_num = 0        # first number
scd_num = 0        # second number
result = 0
```

Now, let's define the # Button functions. Display on the Text screen is achieved using the command **.insert** and since we want the number to be added to the left side we will code it to be added from the **END** of the Text screen. This will allow more numbers to be added sequentially from the left e.g. 123.

Our Text widget is coded as totals. Code: **totals.insert (END, num)**

```
def N1_press():
    global num, func
    num = 1
    totals.insert(END, num)
    return num, func
```

Similarly, when the button 3 is pressed the function will be as follows:

```
def N3_press():
   global num, func
   num = 3
   totals.insert(END, num)
   return num, func
```

*Exercise 14.* Write the code for the rest of the Calculator number buttons.

**Define Calculator functions: Add, Subtract, Multiple, Divide, Clear and Equal-to**

We read the series of buttons pressed and displayed on the *Text* screen using the **.get** command and clear the Text screen using the **.delete** command. Note the format of the commands below:

```
totals.get ('1.0', END) # Read from 1st row and character 0 to End

totals.delete ('1.0', END) # Clear the screen for entry of the second number
```

```
# Functions
```

**Addition function**

```
def Nadd_press():
   global fst_num, func
   fst_num = totals.get ('1.0', END) # Read from 1st row and character 0 to End
   func = 'Add'
   totals.delete('1.0', END)
   return fst_num, func
```

**Multiplication function**

```
def Nmul_press():
   global fst_num, func
   fst_num = totals.get('1.0', END)
   func = 'Mul'
   totals.delete('1.0', END)
   return fst_num, func
```

*Exercise 15.* Define the subtraction and division functions of the Calculator.

**Clear Function**

This function deletes all calculations and restores initial global values. Another GUI function called **messagebox** can be *imported at the top of the program* to show a pop up as a warning, just to make things interesting and for you to learn this new tool as well.

```
def Clr_press():

   global num, fst_num, snd_num, func, result

   num, fst_num, snd_num, func, result = 0, '', '', '', ''

   messagebox.showwarning ('Clear', 'All Calculations cleared')

   totals.delete('1.0', END)

   return num, fst_num, snd_num, func, result
```

There are three major types of message boxes (3): information, warning and question.

See Python docs Help for details:
https://docs.python.org/3/library/tkinter.messagebox.html

We will import the message box immediately after importing Tkinter as below:

```
from Tkinter import *

import tkMessageBox as messagebox
```

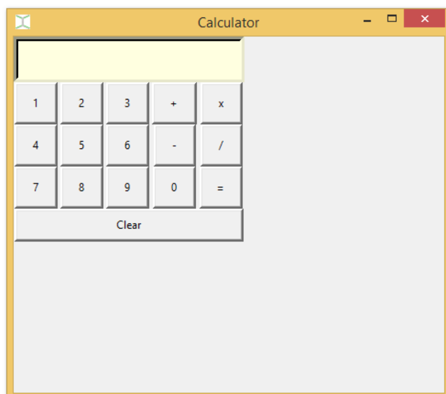
Note the format is different for Python 2 and 3 versions. In Python 2 the module is called **tkMessageBox** whereas in Python 3 it is called **messagebox**. Therefore to harmonize the code we will import the module *as* messagebox (you already know any name is possible and is user assigned).

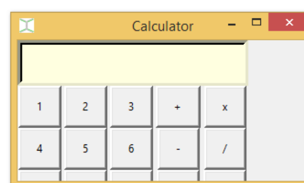
Now our code will work:

```
def Clr_press():

   ...

   messagebox.showwarning ('Clear', 'All Calculations cleared')
```

We have only one major function left and that will allow all calculations i.e Equal to. Let’s take a short break and introduce two fun GUI functions.

You would notice you can still drag the corners of the Calculator and change its look:



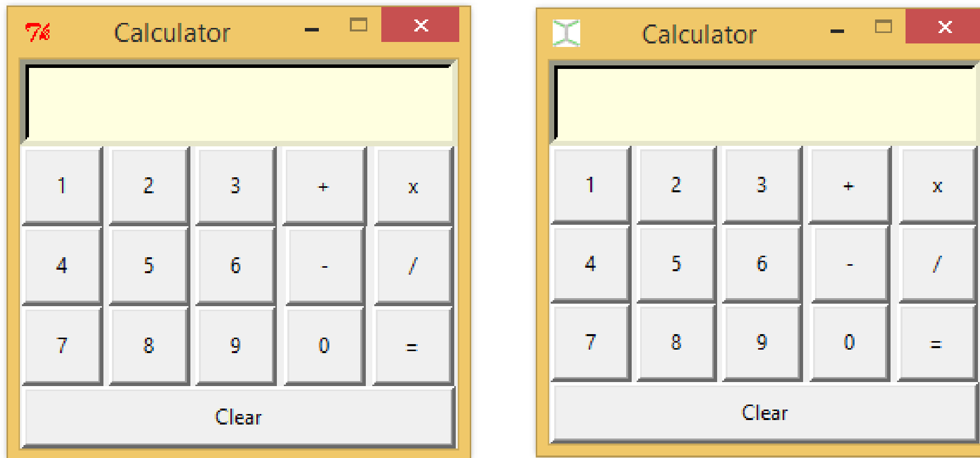




This issue can be fixed with a single command at the end, just *before* the **mainloop** command

```
root.resizable (False, False)
root.mainloop()
```

## Changing Icon

There is an icon on the left hand corner of the GUI. It's set at ***Tk*** for Tkinter. You can customize it.




You can use any image and convert it to an **.ico** file then give its path to the command **.iconbitmap**

```
# Convert to .ico file
# https://image.online-convert.com/
# Note: use / for sub-directory not \ as usual

root.iconbitmap('C:/Users/Desktop/I.ico')
```

**Back to the Calculator function: Equal-to**

It will pull out the first number from the memory, clear the Text screen to get the second number and process these two numbers based on the function (+, -, x, /) pressed by the user.

The first few lines of code are simple. Use **global** to get the first number, **.delete** to clear the Text screen and **.get** to obtain the second number.

To determine the function that the user pressed use a **global** variable (func) and the **if** command. The 'func' variable was updated by button presses (+, -, x, /) (see Button code above).

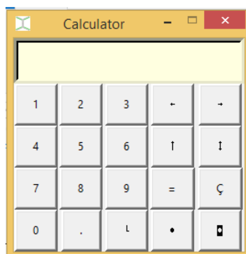
Now the calculation becomes simple. If 'Add' is pressed then add the first and second numbers and store the result in a new variable that is displayed on the Text screen. The process can be repeated for all calculation functions.

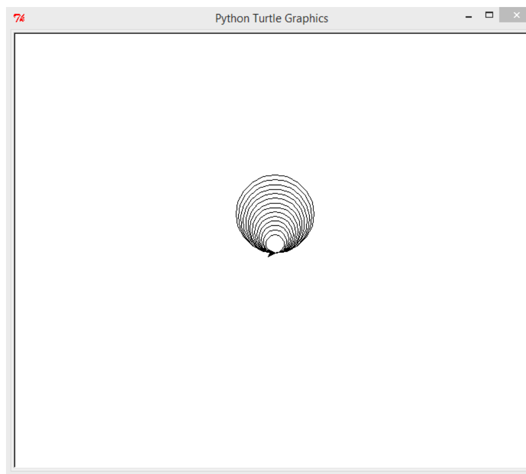
```
def Neql_press():

  global fst_num, snd_num, func, result

  scd_num = totals.get ('1.0', END)
  totals.delete ('1.0', END)

  if func == 'Add':
    add_tot = float(fst_num)+float(scd_num)
    result = str(add_tot)
    totals.insert('1.1', result)
    fst_num, snd_num, func = add_tot, 0, 0

  if func == 'Sub':
    sub_tot = float(fst_num)-float(scd_num)
    result = str(sub_tot)
    totals.insert('1.1', result)
    fst_num, snd_num, func = sub_tot, 0, 0

  if func == 'Mul':
    mul_tot = float(fst_num)*float(scd_num)
    result = str(mul_tot)
    totals.insert('1.1', result)
    fst_num, snd_num, func = mul_tot, 0, 0

  if func == 'Div':
    div_tot = float(fst_num)/float(scd_num)
    result = str(div_tot)
    totals.insert('1.1', result)
    fst_num, snd_num, func = div_tot, 0, 0

  return fst_num, snd_num
```

# 9. Art Toolkit

With a little out of the box thinking we can now convert our little calculator into an Art toolkit. Instead of performing calculations our calculator can morph into a console for drawing on a Turtle canvas. It's simple. We need to write a few drawing functions, assign them to buttons and change the button labels to represent these functions.

### Calculation to Drawing




The first lines of code would obviously be:

```
from Tkinter import *

import turtle as tk
```

We can leave all the number buttons and functions unchanged as we will need to tell turtle how much to move forward or back, the radius of a circle, the angle to change etc. What we do need to change are the calculator functions such as addition, subtraction etc. Instead, we will have functions as forward, backward right and left movements or circle and square. Let's write the Up function

**Moving turtle upwards**

```
def Up_press():
   global fst_num, func
   fst_num = totals.get('1.0', END)
   tk.seth(90)
   tk.fd(float(fst_num))
   totals.delete('1.0', END)
   return fst_num, func
```

Note: the angle of turtle movement is set within the function (90') and the distance to move is set by the user using the calculator number buttons.

*Exercise 16.* Define the Down, Right and Left functions of the Art toolkit.

Let’s define a few other art functions.

**Change the direction (angle) of the turtle:**

```
def Ang_press():
    global fst_num, func
    fst_num = totals.get('1.0', END)
    tk.seth(float(fst_num))
    totals.delete('1.0', END)
    return fst_num, func
```

**Draw a Circle:**

```
def Circ_press():
    global fst_num, func
    fst_num = totals.get('1.0', END)
    tk.circle(float(fst_num))
    totals.delete('1.0', END)
    return fst_num, func
```

**Draw a Square:**

```
def Sq_press():
    global fst_num, func
    fst_num = totals.get('1.0', END)
    for side in range(4):
        tk.fd(float(fst_num))
        tk.rt(90)
    totals.delete('1.0', END)
    return fst_num, func
```

### Button labels

What's now left is to re-label the buttons to indicate the functions they code. Here, we can make of the ASCII codes that include certain symbols. The ASCII code contains a list of 255 characters that can be called in using the command chr() as below:

```
Circ = Button(root, text = chr(7), padx = 15, pady = 10, borderwidth = 3, command = Circ_press)

Circ.grid(row = 5, column = 3)


Sq = Button(root, text = chr(8), padx = 15, pady = 10, borderwidth = 3, command = Sq_press)

Sq.grid(row = 5, column = 4)
```

To view all 255 ASCII characters we can write a simple **for** loop in **Python Shell** and select the character we like for the button label:

```
>>> for i in range(255):
        print i, chr(i)
```

Of course, do not forget the last lines of code to run your GUI:

```
# Run

root.resizable(False, False)

root.mainloop()
```

Now, your Calculator is morphing in to an Art toolkit, made possible with the fusion of **Turtle** and **Tkinter** modules. Transform it more! Happy coding.

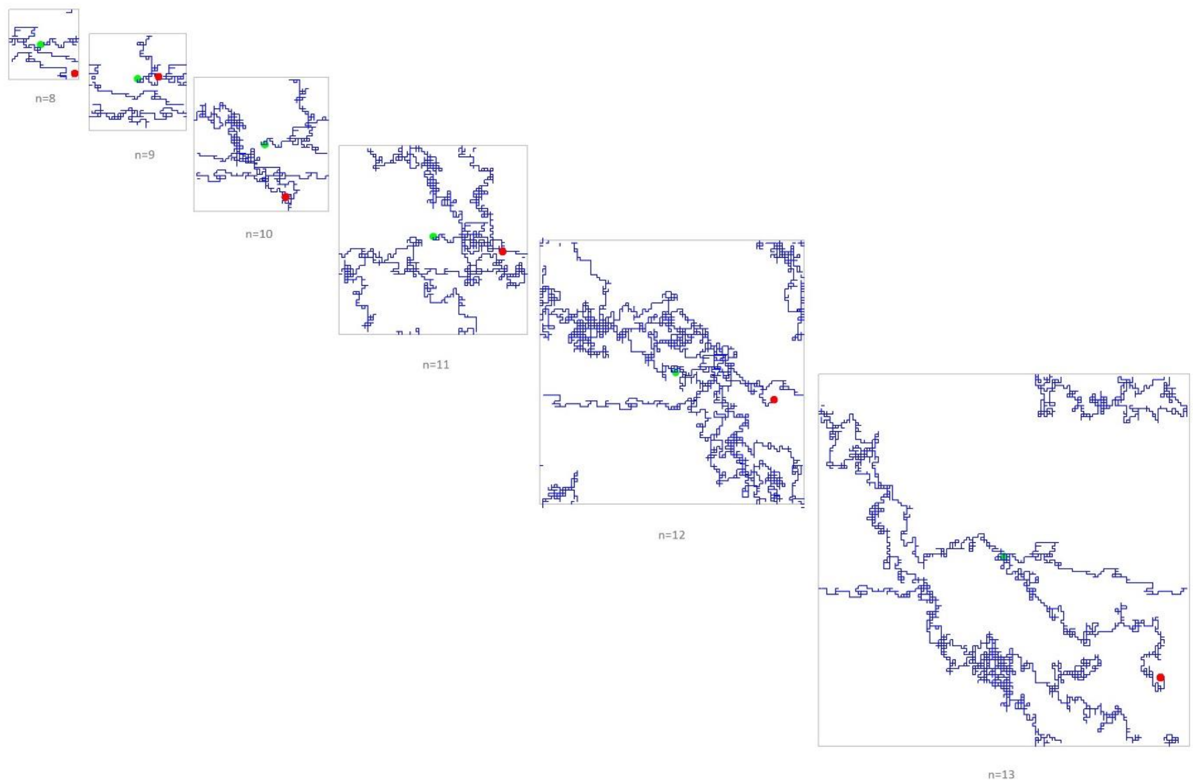
# References


1. GUI 5-Hour Codemy Course:
   https://www.youtube.com/watch?v=YXPyB4XeYLA.
   https://github.com/flatplanet/Intro-To-TKinter-Youtube-Course

2. Codemy.com: https://codemy.com/ and Channel:
   https://www.youtube.com/channel/UCFB0dxMudkws1q8w5NJEAmw

3. https://docs.python.org/3/library/tkinter.messagebox.html

# IV. Files



# 10. Reading & Writing

## Let's Read

It is critical to store information as text to a disk and to be able to retrieve it using our Python programs. This will include saving output of lengthy calculations, re-formatted data etc. Size of the output data may vary. In the case of genetic data it may reach terra-bytes. It will therefore be important to read such files in bite size to understand their structure for further processing by our Python code.

The easiest way to learn is by practicing an example. So let's say we want to know the sequence of a gene. The genome of many organisms including humans (homo sapiens) has been sequenced and available at the click of a button. In your internet browser (e.g. Chrome) go to the website:
https://www.ncbi.nlm.nih.gov/gene/

Type in the name of a gene, e.g. SOD1. You will see the following information and click on the SOD1 homo sapiens gene displayed on top (Gene ID 6647).

As you scroll down the next page you will see FASTA link. Click it and it will show you the entire sequence of the human SOD1 gene (Nucleotide FASTA report).

The DNA sequence of the SOD1 gene is shown. Remember from your basic biology class that DNA is made of 4 letters, A, T, G and C. The genetic sequence will be made up of these 4 alphabets throughout. You'll notice it is quite long. In fact the human SOD1 gene has precisely 9,238 alphabets! This is a modest size gene only. You can save the entire sequence as a text file by clicking on "Send to" drop down as shown, choosing File and saving in FASTA format. The file that will be in your 'Downloads' folder will be **sequence.fasta**

You can rename the file as gene.txt by going to the command prompt through the File option at the top of your Downloads folder or typing cmd in the windows search option. Typing the following command at the prompt in the command-console will rename the sequence.fasta file: **ren seq*.* gene.txt**

You can now write a short code to read only the first 4 lines of the gene.txt file.

```
db = open ('gene.txt', 'r') # Main command that opens a file for reading

count = 1
print ('')
while count<5:              ## Loop for Testing Header (25 line header) - change accordingly
    line = db.readline()    ## readline() is a function that reads a single line of the file
    line = line.rstrip() ## rstrip() is a function that eliminates the 'enters' at the end
    print (line)            ## line is just a variable name (can be any name or letter)
    count += 1

db.close()
```

Note: Your Python program (name it Reader.py) must be saved in the same directory as the gene.txt file. It is not necessary to change the name of your file. You can read the sequence.fasta file with just as much ease if you change the first line of code to:

```
db = open ('sequence.fasta', 'r')
```

**Writing**

We can now open a file for writing data and saving it to a disk.

```
fname = open ('abc.txt', 'w') # Main command that opens a file for reading
```

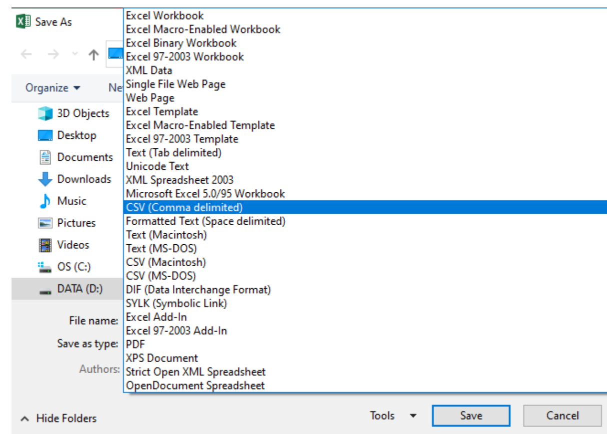
We'll write a short code and save it to the Desktop. It will create a text (.txt) file called abc.txt containing the text: testing writing in file

```
from __future__ import print_function # allows easier print functions

fname = open('abc.txt', 'w')

print ('testing writing in file', file=fname)     # file = fname writes the text to the file

fname.close()
```

# 11. Tables

Think of at least ten flowers and their colors. Make a table in Excel as shown below and save the file as **.csv** (Comma limited) by changing 'Save as type'. Keep the File name as Flowers:

| Serial | Flower | Color |
|---|---|---|
| 1 | Rose | Red |
| 2 | Hibiscus | Red |
| 3 | Rose | Pink |
| 4 | Tulip | Orange |
| 5 | Sunflower | Yellow |
| 6 | Motia | White |
| 7 | Campa | Yellow-white |
| 8 | SadaBahar | Purple |
| 9 | Cambaily | White |
| 10 | Campa | Pink |




Now if you open the Flowers.csv file with Notepad you see the file as follows:

Flowers - Notepad

File Edit Format View Help

```
Serial,Flower,Color
1,Rose,Red
2,Hibiscus,Red
3,Rose,Pink
4,Tulip,Orange
5,Sunflower,Yellow
6,Motia,White
7,Campa,Yellow-white
8,SadaBahar,Purple
9,Cambaily,White
10,Campa,Pink
```

Ln 100% Windows (CRLF) UTF-8

This is what a comma limited file looks like. Let's use Python to read each column separately and make some use of this data on flower color.

We can read this tabulated file line by line. However we also want to **split** the columns (comma separated) so that we can read each variable. The entire data in a row is stored as a list v:

['10', 'Campa', 'Pink']

```
f = open ('flowers.csv', 'r')

for i in f:
   i = i.rstrip()
   v = i.split(',')## split the data in rows into separate columns and stores it in a list v
   print v[0], v[1], v[2]          ## v forms a list and each column can be separately processed
                                   ## variable number e.g. v[2] is flower color written in column 3
f.close()                ## closes the file (important: 'if you open something you must close it')
```

The print function can be improved by using the code (1st line of your program):

```
from __future__ import print_function
```

When the **print_function** is imported the **print** statement changes to:

```
print (v[0], v[1], v[2])
```

### Dictionary and List

These are two interesting data storing tools in Python. They have discrete differences as we shall observe here.

```
from __future__ import print_function

flo_col = []             # make a list of flower colors
flo_dict = {}            # make a dictionary

f = open ('flowers.csv', 'r')
for i in f:
   i = i.rstrip()
   v = i.split(',')
   print (v[0], v[1], v[2])

   flo_col.append(v[2])        # use the append command to add flower color (v[2]) to list
   flo_dict[v[1]] = v[2]# add key, value pairs to dictionary flower [v[1]] and color v[2]

f.close()
```

The format for creating and accessing a list is different from that of a dictionary. This must be carefully noted as in the above code. In a dictionary there is a **key** and **values**. There may be multiple values assigned to a single key but a key cannot appear in a dictionary more than once (no duplication of keys). The command **.append** only works for lists and not dictionaries.

Notice an empty list has square brackets and dictionary has curvy brackets.

```
flo_col = []            # make a list of flower colors
flo_dict = {}           # make a dictionary
```

Also note the brackets at the time of assigning variables to a list as compared to a dictionary (see code above).

When we look at a list in detail we realize it is an exact copy of the color column of the table in the order that the column exists. Typing the name of the list in Python Shell shows the list and its order.

```
>>>flo_col
['Color', 'Red', 'Red', 'Pink', 'Orange', 'Yellow', 'White', 'Yellow-white', 'Purple', 'White', 'Pink']
```

The length (**len**) of the list is the same as the number of rows in the table.

```
>>>len(flo_col)
11
```

However, the dictionary is distinctly different from the list though it was created simultaneously from the same table. It has key:value pairs but not in the order of the table and duplicate names of flowers (keys) are replaced by later occurrences.

```
>>> flo_dict
{'Hibiscus': 'Red', 'Flower': 'Color', 'Cambaily': 'White', 'Sunflower': 'Yellow', 'Rose': 'Pink', 'Tulip': 'Orange', 'Motia': 'White', 'SadaBahar': 'Purple', 'Campa': 'Pink'}
```

```
>>> len(flo_dict)
9
```

Dictionary does not allow duplicate keys, reducing the length to 9 from 11.
To read the key:value pairs in the dictionary we can iterate it just as if it were a list using the command **.items()**.

```
for k, v in flo_dict.items():     # k and v are just variable names signifying key and value
   print (k, v)
```

```
Hibiscus Red
Flower Color
Cambaily White
Sunflower Yellow
Rose Pink
Tulip Orange
Motia White
SadaBahar Purple
Campa Pink
```

The dictionary is however, processed much faster than lists. The advantage in speed allows rapid searching and hence processing of data.

If we were to find out the names of flowers (keys) which were white (values), the search could be easily done using the following code:

```
for k, v in flo_dict.items():
   if v == 'Red':
      print (k, v)
```

```
Hibiscus Red
```

NB: The 'Rose, Red' has been eliminated from the dictionary because of the later 'Rose, Pink'. Rose is a duplicate key and only the later key:value pair is retained.

If we wanted to iterate the dictionary only for the keys, we can use the **.keys()** command:

```
for k in flo_dict.keys():
   print (k)
```

If we only wanted to iterate the values then we can use the **.values()** command:

```
for v in flo_dict.values():
   print (v)
```

Sometimes it is necessary to eliminate duplicates from the data and sometimes it is detrimental. To prevent a dictionary from eliminating duplicates the keys will have to be unique such as a serial number.

If we change our dictionary assignment code to:

```
flo_dict[v[0]] = v[1], v[2]
```

NB: v[0] being serial number, v[1] flower name and v[2] color, then the dictionary will be as complete as the table itself. It contains all the row, column pairs as in the Excel table except that the order will be different.

{'10': ('Campa', 'Pink'), '1': ('Rose', 'Red'), '3': ('Rose', 'Pink'), '2': ('Hibiscus', 'Red'), '5': ('Sunflower', 'Yellow'), '4': ('Tulip', 'Orange'), '7': ('Campa', 'Yellow-white'), '6': ('Motia', 'White'), '9': ('Cambaily', 'White'), '8': ('SadaBahar', 'Purple'), 'Serial': ('Flower', 'Color')}

The length of the dictionary now will be the same as the list and all the rows in the table. A list can only hold data from a column in a table but not the whole table content as separate variables. This is possible in a dictionary as shown here. Now if we search the dictionary for the second value being 'Red' we get the complete answer of 'Rose, Red' and 'Hisbiscus, Red'.

```
for k, v in flo_dict.items():
    if v[1] == 'Red':
        print (k, v)
```

```
1 ('Rose', 'Red')
2 ('Hibiscus', 'Red')
```

A dictionary command, **sorted,** orders the dictionary according to the **keys**.

```
for k, v in sorted(flo_dict.items()):
    print (k, v[0], v[1])
```

We can label the variables with names as well and develop the dictionary in a more readable manner

**Serial = v[0]**
**Flower = v[1]**
**Color = v[2]**

#flo_dict[v[0]] = v[1], v[2] # instead of code we can use variable names that are more easily understandable as below

**flo_dict[Serial] = Flower, Color**

## Yet our dictionary is not ordered according to Serial number:

1 Rose Red
10 Campa Pink
2 Hibiscus Red
3 Rose Pink
4 Tulip Orange
5 Sunflower Yellow
6 Motia White
7 Campa Yellow-white
8 SadaBahar Purple
9 Cambaily White
Serial Flower Color

We will edit our input file flowers.csv and delete the header:

Serial Flower Color

*Exercise 17.* Is there another way the header can be eliminated?

Answer: Yes. Insert the reading code (in the previous chapter) between opening the file and reading it with the **for** loop.

```
f = open ('flowers.csv', 'r') # open the tablulated data file

count = 0
print ('')
while count<1: ## Loop for eliminate Header (1 line header)
    line = f.readline()
    line = line.rstrip()
    print (line)
    count += 1

for i in f:
```

Now the serial only has numbers and can be converted to an integer as below:

```
Serial = int(v[0])
```

Now if we run our code our dictionary will be properly organized in serial order:

```
1 Rose Red
2 Hibiscus Red
3 Rose Pink
4 Tulip Orange
5 Sunflower Yellow
6 Motia White
7 Campa Yellow-white
8 SadaBahar Purple
9 Cambaily White
10 Campa Pink
```

## Creating an HTML file

We can write our dictionary to an HTML file. Of course you will need to learn a bit of HTML as well (which is quite easy). Here, we will go over some HTML code as well that will make it easy to learn.

```
fhtml = open ('Phool.html', 'w') # open the tabulated HTML file
```

Note: Flower is called ‘Phool’ in Urdu (Latinized spelling).

You can directly print the dictionary to your HTML file however it will be printed as a single line.

```
for k, v in sorted(flo_dict.items()):
        print (k, v[0], v[1], file=fhtml)
```

1 Rose Red 2 Hibiscus Red 3 Rose Pink 4 Tulip Orange 5 Sunflower Yellow 6 Motia White 7 Campa Yellow-white 8 SadaBahar Purple 9 Cambaily White 10 Campa Pink

Therefore, we first need to create a Table in our HTML file

```
print ('<table><align="center">', file=fhtml) # this will tabulate your print data
```

| | | |
|---|---|---|
| 1 | Rose | Red |
| 2 | Hibiscus | Red |
| 3 | Rose | Pink |
| 4 | Tulip | Orange |
| 5 | Sunflower | Yellow |
| 6 | Motia | White |
| 7 | Campa | Yellow-white |
| 8 | SadaBahar | Purple |
| 9 | Cambaily | White |
| 10 | Campa | Pink |

Let’s make our table neat

| | | |
|---|---|---|
| 1 | Rose | Red |
| 2 | Hibiscus | Red |
| 3 | Rose | Pink |
| 4 | Tulip | Orange |
| 5 | Sunflower | Yellow |
| 6 | Motia | White |
| 7 | Campa | Yellow-white |
| 8 | SadaBahar | Purple |
| 9 | Cambaily | White |
| 10 | Campa | Pink |

```
print ('<TABLE BORDER="3" CELLSPACING="1" CELLPADDING="5">', file=fhtml)
```

Let’s add some color to our table and create a title.

| Flowers | | |
|---|---|---|
| 1 | Rose | Red |
| 2 | Hibiscus | Red |
| 3 | Rose | Pink |
| 4 | Tulip | Orange |
| 5 | Sunflower | Yellow |
| 6 | Motia | White |
| 7 | Campa | Yellow-white |
| 8 | SadaBahar | Purple |
| 9 | Cambaily | White |
| 10 | Campa | Pink |

```
print ('<tr bgcolor="#006400"><td align="center"
colspan="18"><B><H1><p
style="color:#FFFFFF";>Flowers</B></H1></td></tr>',
file=fhtml)
```

Let’s add some more color to our table and create a header.

Center align the data and in light green color write the column titles, Serial, Flower and Color.

Note: Each value is contained between and open bracket signified by <td> and a close bracket, </td>. Whatever the value e.g. Serial is in inverted commas and preceded and followed by a + sign (indicating insert this value in the code).

```
print ('<tr align="center"
bgcolor="#9ACD32"><td>'+'Serial'+'</td><td>'+'Flower'+'</td><td>'+'Color'+'</td></tr>', file
= fhtml)
```

The code should look like this in Python:

```
print ('<tr align="center"
bgcolor="#9ACD32"><td>'+'Serial'+'</td><td>'+'Flower'+'</td><td>'+'Color'+'</td></tr>', file
= fhtml)
```

Now let’s print our sorted dictionary (flo_dict) to our html file as a table. However, remember HTML does not know numbers. It only prints in strings. So we have to convert our numbers to string using the function **str**.

E.g. **str(k)** and **str(v[0])**

```
for k, v in sorted(flo_dict.items()):
      print ('<tr bgcolor = "White"> <td>'+str(k)+'</td><td>'+str(v[0])+'</td>
<td>'+str(v[1])+'</td></tr>', file = fhtml)
```

If you want to center align the data in each row add **align="center"** after **<tr**

| Flowers | | |
|---|---|---|
| Serial | Flower | Color |
| 1 | Rose | Red |
| 2 | Hibiscus | Red |
| 3 | Rose | Pink |
| 4 | Tulip | Orange |
| 5 | Sunflower | Yellow |
| 6 | Motia | White |
| 7 | Campa | Yellow-white |
| 8 | SadaBahar | Purple |
| 9 | Cambaily | White |
| 10 | Campa | Pink |

```
print ('<tr align="center"
bgcolor="White"><td>'+str(k)+'</td><td>'+str(v[0])+'</td><td>'+str(v[1])+'</td></tr>', file =
fhtml)
```

Now, your beautiful table in an HTML format is ready. You can open it in any internet browser such as Google Chrome.

If the HTML file (Phool.html) is opened in Notepad it will show the HTML code that was written by our Python program:

```
<table><align="center">
<TABLE BORDER="3" CELLSPACING="1" CELLPADDING="5">
<tr bgcolor="#006400"><td align="center" colspan="18"><B><H1><p
style="color:#FFFFFF";>Flowers</B></H1></td></tr>
<tr align="center" bgcolor="#9ACD32"><td>Serial</td><td>Flower</td><td>Color</td></tr>
<tr align="center" bgcolor="White"><td>1</td><td>Rose</td><td>Red</td></tr>
<tr align="center" bgcolor="White"><td>2</td><td>Hibiscus</td><td>Red</td></tr>
<tr align="center" bgcolor="White"><td>3</td><td>Rose</td><td>Pink</td></tr>
<tr align="center" bgcolor="White"><td>4</td><td>Tulip</td><td>Orange</td></tr>
<tr align="center" bgcolor="White"><td>5</td><td>Sunflower</td><td>Yellow</td></tr>
<tr align="center" bgcolor="White"><td>6</td><td>Motia</td><td>White</td></tr>
<tr align="center" bgcolor="White"><td>7</td><td>Campa</td><td>Yellow-white</td></tr>
<tr align="center" bgcolor="White"><td>8</td><td>SadaBahar</td><td>Purple</td></tr>
<tr align="center" bgcolor="White"><td>9</td><td>Cambaily</td><td>White</td></tr>
<tr align="center" bgcolor="White"><td>10</td><td>Campa</td><td>Pink</td></tr>
```

Enjoy merging HTML with Python!

Let’s process our dictionary values further and make our HTML file a bit interactive by creating hyperlinks.

First, we want variables in the dictionary to be ‘read and understood’. This is easily done using the **if** command in the **for** loop. So if we want our code to recognize the flower Motia (Jasmine) then we will check to see if flower (variable v[0] in our code) is == ‘Motia’:

```
if v[0] == 'Motia':
```

If that is the case then we want to color the row as the same color as the flower. In this case Motia is white so we will assign a light yellow color to all the other rows so that when Motia is selected the row color (white) looks highlighted.

We therefore create a new variable and place it above and outside the **for** loop:

```
highlight = '#FFF8DC''  # hexa code for light yellow
```

Similarly, we want to select the pink rose. However, in our dictionary we have a red rose as well. So we have to make our **if** statement recognize both the flower (v[0]) and its color (v[1])

```
if v[0] == 'Rose' and v[1] == 'Pink':
```

We want to see the picture of the flower when the hyperlink is clicked. There are two ways to go about it. We can find the picture on the internet and save the link to the page (e.g. https://pakistan.desertcart.com/products/77902556-seedlings-india-motia-live-plant). We download photo of the flower and create link to the saved photo file on our computer. The saved photo on our computer is accessible through the file path such as: 'C:\Python27\Rose_Pink.jpg'. We will therefore define a new variable called link to insert the relevant link in the HTML file. Remember all variables have to be strings (not numeric) in HTML code.

```
highlight = '#FFF8DC'
link = ''

for k, v in sorted(flo_dict.items()):

    if v[0] == 'Rose' and v[1] == 'Pink':

        highlight = "Pink"   # color the row pink instead of light yellow
        link = 'C:\Python27\Rose_Pink.jpg'  # path to the pink rose photo

    elif v[0] == 'Motia':  # we use elif here so that both pink rose and motia are highlighted

        highlight = "White"
        link = 'https://pakistan.desertcart.com/products/77902556-seedlings-india-motia-
live-plant'

    else:   # we need the else statement to return the row color and links to default values
        highlight = '#FFF8DC'
        link = ''

    print ('<tr align="center" bgcolor='+highlight+'><td>'+str(k)+'</td><td>'+'<a
href='+link+'>'+str(v[0])+'</a></td><td>'+str(v[1])+'</td></tr>', file = fhtml)
```

**Hyperlinks**

They require the initiating code: **<a href='+insert link to file here+'>'**
What needs to be hyperlinked follows and the <a> loop is closed. Here we want to hyperlink the name of the flower that is coded by the variable v[0]. Remember, HTML only processes string variables. So we code the flower name as str(v[0]). If we want we can assign it to a new variable, flower or flower_name, inside the **for** loop.

Hence, our HTML code will look like this:

```
<td><a href='+link+'>'+str(v[0])+'</a></td>
```

*Exercise 18.* Can the hyperlink be a constant word such as photo instead of a variable such as flower_name?

Answer: Yes. In this case we need to create another column with the word 'photo' in each row. So our HTML code will look like:

```
print ('<tr align="center" bgcolor='+highlight+'><td>'+str(k)+'</td><td>'+str(v[0])+'</td><td>'+str(v[1])+'</td><td><a href='+link+'>'+'photo'+'</a></td></tr>', file = fhtml)
```

| Flowers | | | |
|---|---|---|---|
| Serial | Flower | Color | Picture |
| 1 | Rose | Red | photo |
| 2 | Hibiscus | Red | photo |
| 3 | Rose | Pink | photo |
| 4 | Tulip | Orange | photo |
| 5 | Sunflower | Yellow | photo |
| 6 | Motia | White | photo |
| 7 | Campa | Yellow-white | photo |
| 8 | SadaBahar | Purple | photo |
| 9 | Cambaily | White | photo |
| 10 | Campa | Pink | photo |

Here, we moved the hyperlink to the photo to a separate column after the flower color.

The header of the table has to be upgraded as well to include a 4th column called 'Picture':

```
print ('<tr align="center" bgcolor="#9ACD32"><td>'+'Serial'+'</td><td>'+'Flower'+'</td><td>'+'Color'+'</td><td>'+'Picture'+'</td></tr>', file = fhtml)
```

# 12. Graphs

Making graphs is a fantastic way of visualizing data for analysis. Python has excellent capabilities for making graphs. However, this requires setting up certain **packages** in Python which is a bit challenging. So let's get going with it immediately. For everyone's convenience I have detailed the process on **stackoverflow** – a resource you will need to develop your Python skills further.

### Matplotlib

This is the main **library** needed for making graphs in Python. however it will only work after you install all the packages detailed below:

Download these libraries: **numpy, matplotlib, six, dateutil, pyparsing, ptz**

My solution to running **Matplotlib** is available at the following link (1):
http://stackoverflow.com/users/5179477/mohammad-saeed

Then click on the sub-link: Can't install Matplotlib for Python

**numPy:** (Download from the link below) (2)

**http://sourceforge.net/projects/numpy/files/NumPy/1.9.0/numpy-1.9.0-win32-superpack-python2.7.exe/download**

NB: numpy is currently available for Windows only in 32-bit format

Checking your numpy download in Python Shell:

```
>>> import numpy as np

>>> a = np.arange(10)

>>> a
```

Installing **dateutil and six:**

**cmd** (to reach windows command prompt type **cmd** in windows search box)

Type the following at the command prompt in C: drive

**C:\> cd**

**C:\> cd python27\scripts**

**C:\ python27\scripts> pip2.7 install python-dateutil**

This should install **six** as well

Installing **pyparsing and pytz:**

**C:\ python27\scripts> pip2.7 install pyparsing**

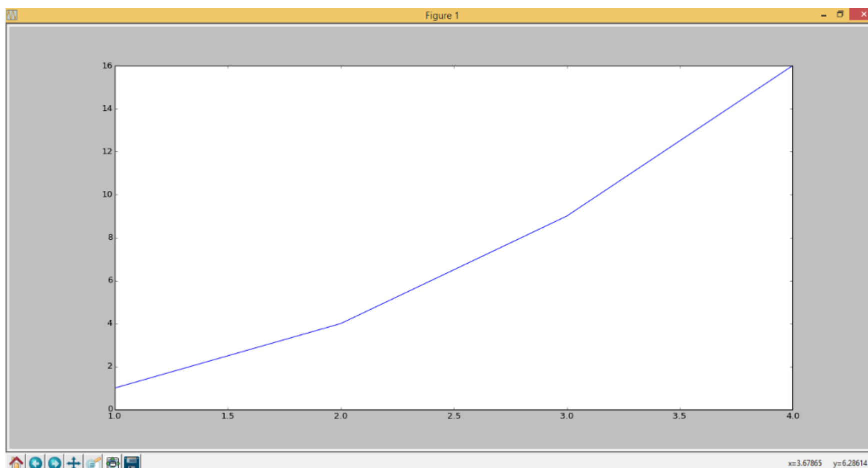
**C:\ python27\scripts>pip2.7 install pytz**

**matplotlib:** (Download from the link below)

**http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.4.3/windows/matplotlib-1.4.3.win32-py2.7.exe/download?use_mirror=liquidtelecom**

Checking your matplotlib:

```
>>> from matplotlib import pyplot
>>> pyplot.plot([1, 2, 3, 4], [1, 4, 9, 16])
>>> pyplot.show()
```




This graph will be shown.

If we change our table (by editing our file flowers.csv) to include the number of particular flowers in the garden then our table will look like:

| Flowers | | | |
|---|---|---|---|
| Serial | Flower | Color | Number |
| 1 | Rose | Red | 10 |
| 2 | Hibiscus | Red | 15 |
| 3 | Rose | Pink | 12 |
| 4 | Tulip | Orange | 5 |
| 5 | Sunflower | Yellow | 7 |
| 6 | Motia | White | 65 |
| 7 | Campa | Yellow-white | 30 |
| 8 | SadaBahar | Purple | 26 |
| 9 | Cambaily | White | 55 |
| 10 | Campa | Pink | 16 |

*Exercise 19.*

a) How can you code the dictionary flo_dict to generate this table?

b) What changes will you make to the html code to obtain the result shown in this table?

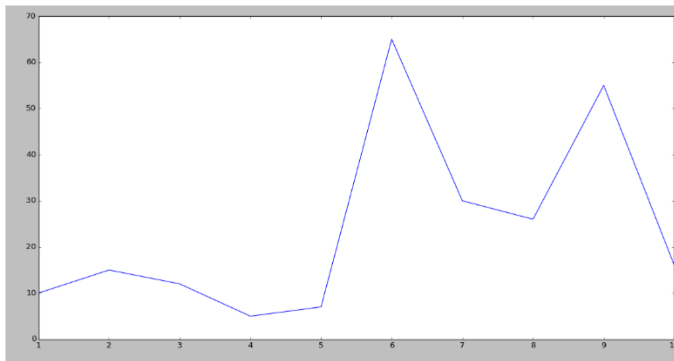
To check whether our matplotlib was working or not we typed the following in Python Shell:

**pyplot.plot([1, 2, 3, 4], [1, 4, 9, 16])**

Notice that we are plotting two **numeric** lists. List x = [1, 2, 3, 4] and list y = [1, 4, 9, 16].

Pyplot takes these lists and plots the second (y) against the first (x).

Pyplot needs only numeric lists and cannot have string lists. So we cannot plot the flower numbers (numeric on y-axis) against flower names or colors (string on x-axis). In our table 'Flowers' we have only two numeric columns – serial and numbers. So we will plot flower numbers (flw_num) against serial (ser).

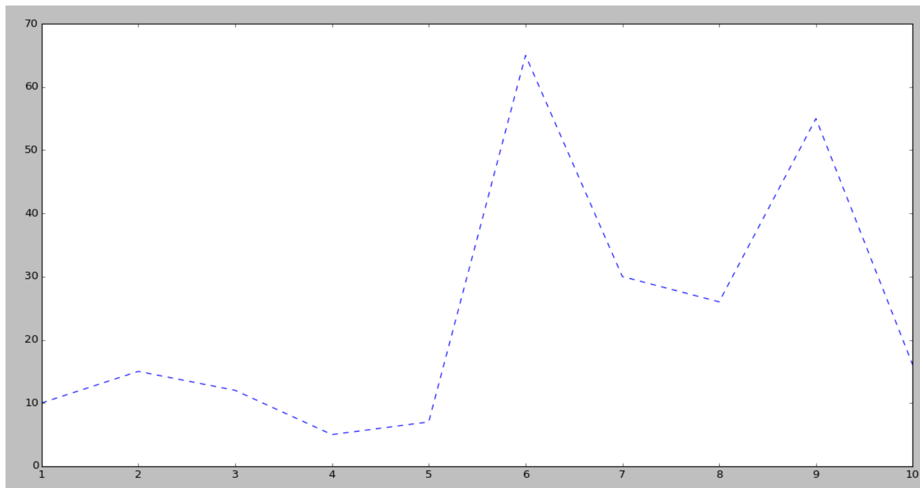


**plt.plot(ser, flw_num)**

**plt.show()**

This graph will be shown (default color is blue)

## Graph modifications

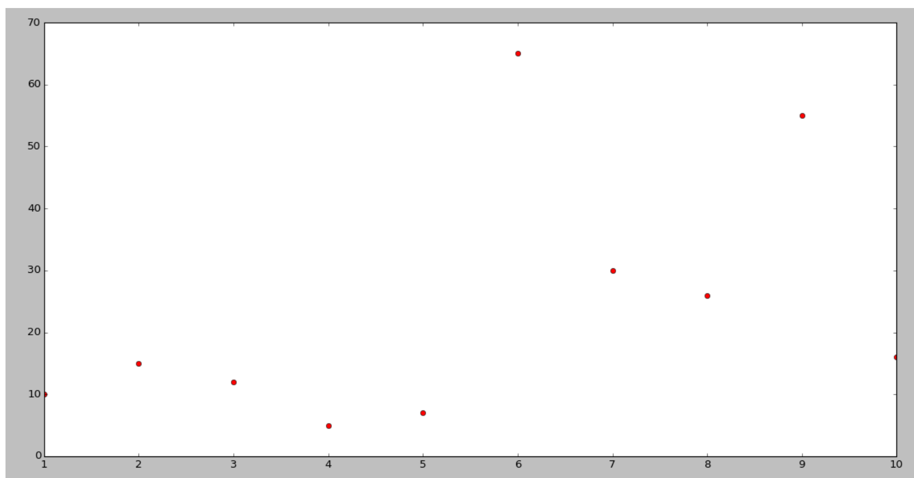



We can make a dashed line graph in blue by simply adding 'b--' to the code.

```
plt.plot(ser, flw_num, 'b--')

plt.show()
```

We can make a red dot graph by changing it to ‘ro’ (r = red color, o = dot (simple letter o, not 0).

```
plt.plot(ser, flw_num, 'ro')

plt.show()
```



Now, let's give our graph a **title**, each axis a **label**, set the **axis** according to our data, visualize the **grid**, and draw a cut off line across.

**plt.plot(ser, flw_num, 'r-')** # single dash will produce a solid line graph

**plt.title('Flowers in the Garden')** # graph title

**plt.xlabel('Flower Type')** # label for x-axis

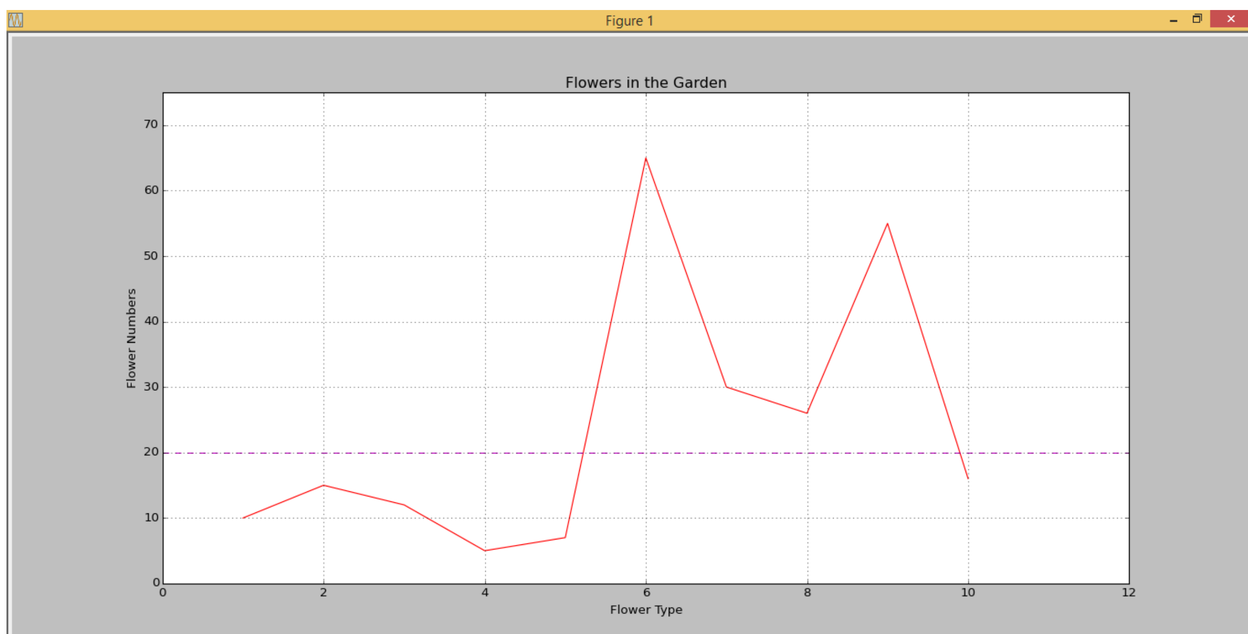
**plt.ylabel('Flower Numbers')** # label for y-axis

**plt.axis([0, max(ser)+2, 0, max(flw_num)+10])** # select graph axis from 0 to the maximum serial number, max(ser), and add 2 so as to completely include all data on x-axis. Similarly, find the maximum number of flowers, max(flw_num), and add 10 to the y-axis.

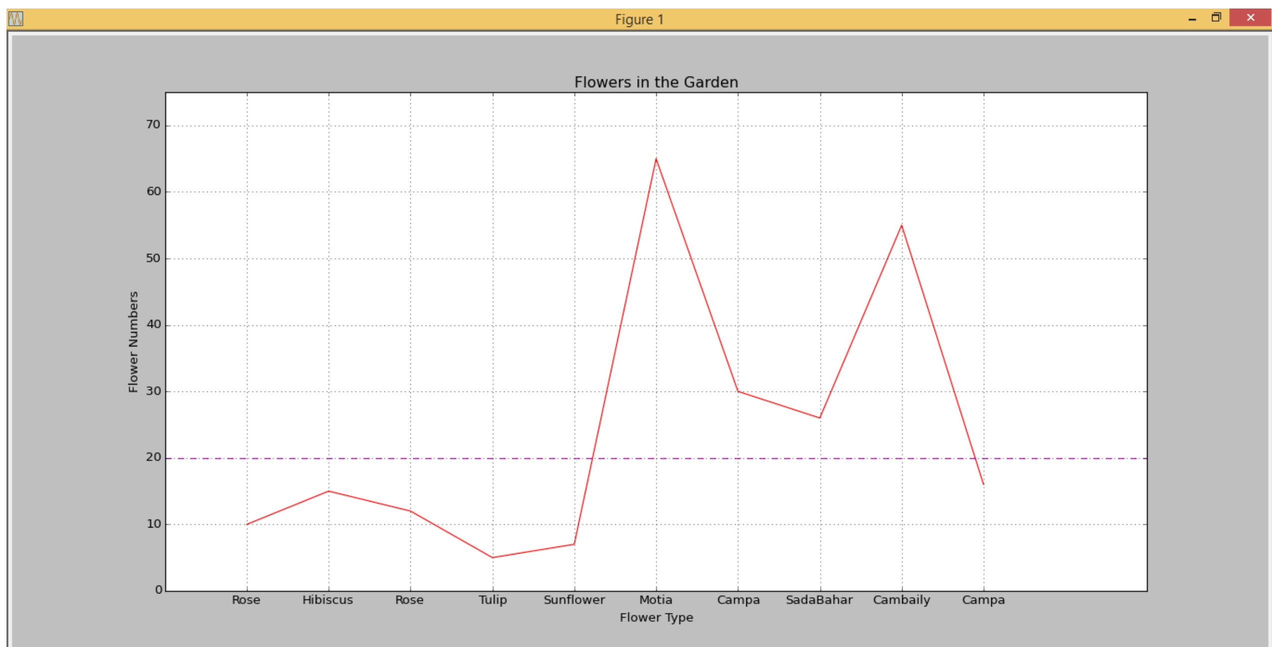
**plt.grid(True)** # ensure we see the grid on the graph

**plt.plot([0, 12], [20, 20], 'm--')** # draw a cut off dashed line in magenta at y=20. This is a line plot, so two lists are required. The initial coordinate for the line is 0, 20 and the final coordinate is 12, 20.

**plt.show()**



### Labeling the data points




It requires a special modification to label the x-axis tick marks with the names of the flowers (list flw). We have to create a **subplot** to access the **ticks** then change the **labels** of the ticks to the **list** flw (flower names).

**fig = plt.figure()** # first we have to assign the figure to a variable (e.g. fig)

```
plt.plot(ser, flw_num, 'r-')
plt.title('Flowers in the Garden')
plt.xlabel('Flower Type')
plt.ylabel('Flower Numbers')
plt.axis([0, max(ser)+2, 0, max(flw_num)+10])
plt.grid(True)
plt.plot([0, 12], [20, 20], 'm--')
```

**ax = fig.add_subplot(111)** # Add a subplot to the figure. Remember ax is only a variable name. The main command here is add_subplot.

111 represent the number and position of the graphs. Here there is 1 graph, located at (x, y) = (1, 1).

Now suppose, we had to display two graphs together, one above the other, the subplot would be coded as 211 for the first graph and 212 for the second.

If we had to display two graphs side by side, it would mean that the display is divided into 4 areas. In that case the code would be subplot (221) for graph 1 and (222) for graph 2. Similarly, for 3 graphs one on top of another the subplot would be coded as 311, 312 and 313. Understanding this system of displaying graphs greatly helps our creativity and enhances our presentation (See reference 3).

**ax.set_xticks(ser)** # this will set the ticks on the x-axis, bringing balance to the graph. Try commenting out this line (above) to see how the graph becomes unbalanced.

**ax.set_xticklabels(flw)** finally, this command will set the tick labels on the x-axis according to the list (flw) i.e. flower names.

plt.show() # command to display the graph created above.

**Bar Graph**

Instead of a line graph as above we can create a bar graph simply by changing **plot** to **bar**:

Replace: plt.plot(ser, flw_num, 'r-')

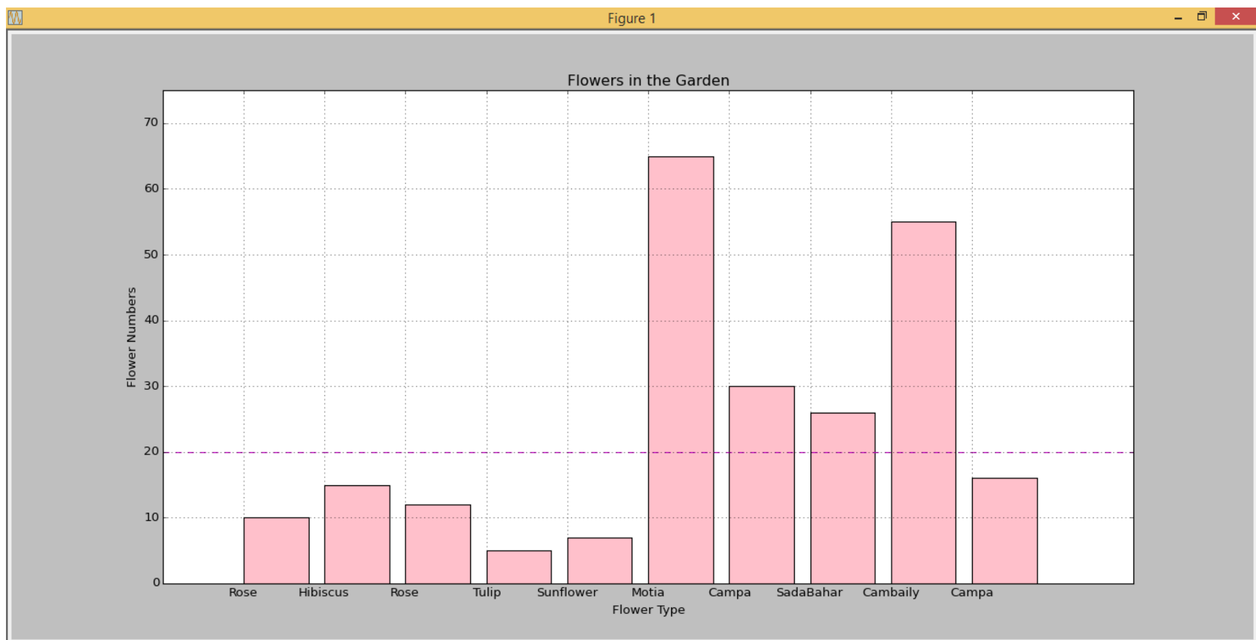
By: **plt.bar(ser, flw_num, color = ‘pink’)**

NB: Color of the bar graph is chosen by using the command color =, with the color of your choice within inverted commas

Labeling a bar graph requires an additional step compared to labeling the data points on a line graph. The reason is that on a line graph the tick is a single point, whereas on a bar graph the bar has a width and is not a single point. Therefore, the bars need to be centered on the tick for current display of labels.

The code that is need for the subplot is:

**ax.bar(ser, flw_num, 0.5, align = "center")** # the bars are centered and have size 0.5 width (1.0 will leave no space between bars and make the graph look like a Histogram).

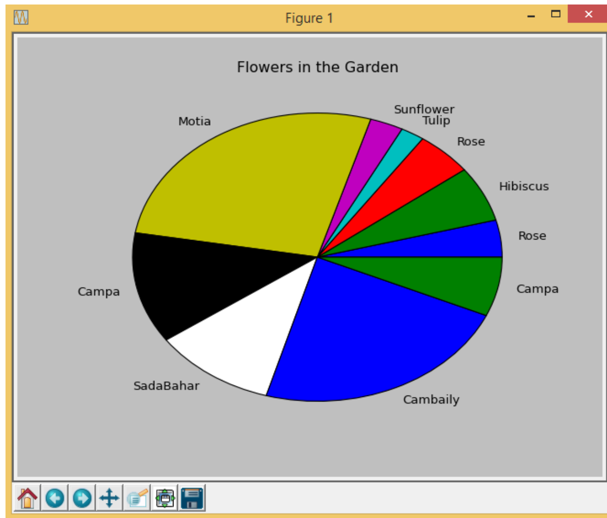



```
fig = plt.figure()
plt.bar(ser, flw_num, color='pink')
… code as above …
plt.plot([0, 12], [20, 20], 'm--')
ax = fig.add_subplot(111)
ax.bar(ser, flw_num, 0.5, align = "center")
ax.set_xticks(ser)
ax.set_xticklabels(flw)
plt.show()
```

## Pie Chart

Another important way data can be graphed is as a **pie** chart. The difference is that in a line or bar graph we had two numeric lists. In a pie chart we have only one numeric list and we can directly apply the labels using s string list.

**plt.pie(flw_num, labels = flw)** # instead of bar we use pie and labels are directly inserted from the string list, flw (flower names).




```
plt.pie(flw_num, labels = flw)
plt.title('Flowers in the Garden')
plt.show()
```

To assign **colors** of our choice to the pie chart, we can make a **list** and use the **colors** command in **pie**:

```
cs=['red', 'blue', 'green', 'pink', 'yellow',
'orange', 'purple', 'white', 'magenta', 'grey']

plt.pie(flw_num, labels = flw, colors=cs)
```

Finally, if we want to display two graphs in a layered fashion, e.g. the line graph and the bar chart together, we can put the two codes together subplots in a single figure.

```
fig = plt.figure()

plt.subplot(211)
plt.plot(ser, flw_num, 'r-')
plt.title('Flowers in the Garden')
plt.xlabel('Flower Type')
plt.ylabel('Flower Numbers')
plt.axis([0, max(ser)+2, 0,
max(flw_num)+10])
plt.grid(True)
plt.plot([0, 12], [20, 20], 'm--')
ax = fig.add_subplot(211)
ax.set_xticks(ser)
ax.set_xticklabels(flw)
```

```
plt.subplot(212)
cs=['red', 'blue', 'pink', 'orange', 'yellow', 'white', 'green', 'magenta', 'grey', 'purple']
plt.title('Flowers in the Garden')
bx = fig.add_subplot(212)
bx.bar(ser, flw_num, 0.5, align = "center", color=cs)
bx.set_xticks(ser)
bx.set_xticklabels(flw)

plt.tight_layout()
plt.show()
```

*Exercise 20.*

A) Display the line graph, the bar graph and the pie chart in a single figure.

B) Add a column titled 'Plants' to the .csv file containing the number of plants of each flower type in the garden. E.g. 3 Red Rose plants, 5 Motia plants etc. Now, rewrite the code to display a bar graph of the flower numbers and a line graph of number of plants for each flower type below the bar graph.

**Fractal Graph**

Fractals are self-similar objects independent of scale possessing infinite detail. Koch's snowflake is an example of a mathematically exact fractals. However, much of nature is full of fractals such as coastlines, trees, mountains and even heart rhythms (4). What was so interesting to discover is that even the sequence of our genes has fractal organization (5, 6). Using a novel graphing method we can see the self-similar repeating patterns in our genes. This method displays the sequence of a gene like a photograph and is therefore sensitive to pattern recognition.

Let's go back to the SOD1 sequence we saved as the file gene.txt and use it to make the fractal graph. To simplify our code, delete the first line of the file:

>NC_000021.9:31659693-31668931 Homo sapiens chromosome 21, GRCh38.p14 Primary Assembly

Next, put an X at the end of the file. This will serve as the end of file mark and make our reading and processing the file much easier.

The first step is to open the file 'gene.txt', read it letter by letter (nucleotide, a DNA letter) and store the letters in a **list** called 'g'. We will use the **while** loop to do this.

After saving the DNA letters (nucleotides) to **list** g (entire gene sequence is converted to individual letters), we can draw the gene sequence using **Turtle**.

```
db = open ('gene.txt', 'r')
g = []

while True:

      nuc = db.read(1) # read one letter at a time
      nuc = nuc.rstrip() # do not include '\n' indicating enters for new lines

      if nuc=='G':
         nuc = 'G'

      elif nuc=='C': # if C is read, save C to list g
         nuc = 'C'

      elif nuc=='A':
         nuc = 'A'

      elif nuc=='T':
         nuc = 'T'

      if nuc == 'X': break # stop the while loop is letter X is encountered (which we deliberately
                             placed to indicate end of file).

      g.append(nuc) # save the nucleotide to the list g

db.close() # close the file

print len(g)   # answer should be 9372
```

Of course, the first line of our program would be to **import** the **turtle** module

```
import turtle as tu
```

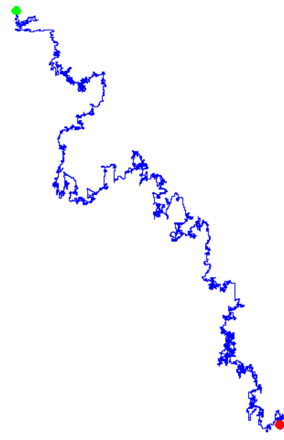
The last line will be **tu.exitonclick()**

Then to speed up the drawing of a large gene such as human SOD1 we will turn the tracer off, hide the turtle and increase speed to 'fastest'.

```
tu.hideturtle()
tu.speed('fastest')
tu.tracer(False)
```

To draw the fractal diagram we will iterate the list g to determine the nucleotide that is present in the gene sequence and draw it on the screen. Since there are 4 DNA letters (A, T, G, C) we will choose 4 different directions (N, S, E and W) to represent them. If G is encountered we will move the turtle 1 unit towards the East. Similarly, in case of C we will move the turtle 1 unit towards the West and so forth.

```
for n in g:

    if n=='G':
        tu.seth(0)
        tu.forward(1)

    if n=='C':
        tu.seth(180)
        tu.forward(1)

    if n=='A':
        tu.seth(90)
        tu.forward(1)

    if n=='T':
        tu.seth(270)
        tu.forward(1)
```




Note: Since the human SOD1 is a relatively long gene and has considerably more 'T' than other nucleotides, the graph predominantly grows downwards and goes beyond the screen. To prevent this we can start our graph higher up by positioning the turtle by **tu.setpos(0, 250)**.

```
tu.pu()
tu.setpos(0, 250)
tu.pd()
```

Also we can indicate the beginning of the graph with a green circle and the end of the graph with a red circle. The start position indicator code would be:

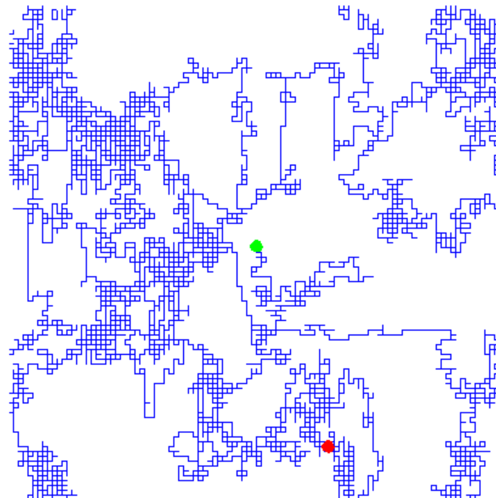
```
tu.color('green')
tu.begin_fill()
tu.circle(5)
tu.end_fill()
```

To generate the fractal graph as a photograph we will define the screen size and set borders so that when the turtle approaches them it wraps around to the opposite side. This will allow the sequence to weave in a defined square space.

```
## Set box size
wni = 200
tu.screensize(wni, wni)

tu.pu()
tu.setpos(0, 0) # center position the turtle so that all gene sequences start in the center
tu.pd()
```

To allow this weaving we need to take note of the turtle position in x and y axis. For ease of visualization we can move the turtle 4 units instead of 1.

```
for n in g:

    tpx = tu.xcor() # record position of turtle on the x-axis
    tpy = tu.ycor() # record position of turtle on the y-axis

    if tpx >=wni: # if turtle exceeds the box dimensions then
      tu.pu()
      tu.setx(-wni) # set the turtle position to the opposite side
      tu.pd()

    elif tpx <=-wni:
      tu.pu()
      tu.setx(wni)
      tu.pd()

    elif tpy >=wni:
      tu.pu()
      tu.sety(-wni)
      tu.pd()

    elif tpy <=-wni:
      tu.pu()
      tu.sety(wni)
      tu.pd()

    if n=='G':
        tu.seth(0) .... The rest of the code is described in the section above on previous page
```



### Saving Images

How can we save this interesting photograph? We first obtain the image of the screen using **getscreen()** command. Then this image is saved using the **getcanvas()** command as a postscript file.

```
ts = tu.getscreen()

ts.getcanvas().postscript(file= 'Gene.eps')
```

The EPS file can be converted to any photo format such as JPEG (7).

Use EPS converter to convert your .eps image to JPEG:
https://www.epsconverter.com/

OR Download: https://epsviewer.org/downloadfinal.aspx")

*Program:* **Fractal Genes**

```
import turtle as tu
db = open ('gene.txt', 'r')
g = []

while True:
    nuc = db.read(1)
    nuc = nuc.rstrip()

    if nuc=='G':
      nuc = 'G'

    elif nuc=='C':
      nuc = 'C'

    elif nuc=='A':
      nuc = 'A'

    elif nuc=='T':
      nuc = 'T'

    if nuc == 'X': break
    g.append(nuc)
db.close()
print len(g)

# Start canvas and draw sequence
stpi = 4 # open diagram = 1
tu.hideturtle()
tu.speed('fastest')
tu.tracer(False)

## Set box size
wni = 200
tu.screensize(wni, wni) # for closed diagram
tu.pu()
tu.setpos(0, 0)
tu.pd()

## Mark start position with green circle
tu.color('green')
tu.begin_fill()
tu.circle(5)
tu.end_fill()
tu.color('blue') # draw diagram in blue
```

```
Main loop for reading gene sequence from
list g and drawing it on canvas with turtle

for n in g:

    # For open diagram comment out here
    tpx = tu.xcor()
    tpy = tu.ycor()

    if tpx >=wni:
      tu.pu()
      tu.setx(-wni)
      tu.pd()

    elif tpx <=-wni:
      tu.pu()
      tu.setx(wni)
      tu.pd()

    elif tpy >=wni:
      tu.pu()
      tu.sety(-wni)
      tu.pd()

    elif tpy <=-wni:
      tu.pu()
      tu.sety(wni)
      tu.pd()
    # Till here
```

```
# Draw the Gene Sequence

    if n=='G':
        tu.seth(0)
        tu.forward(stpi)

    if n=='C':
        tu.seth(180)
        tu.forward(stpi)

    if n=='A':
        tu.seth(90)
        tu.forward(stpi)

    if n=='T':
        tu.seth(270)
        tu.forward(stpi)

# End drawing with red circle

tu.color('red')
tu.begin_fill()
tu.circle(5)
tu.end_fill()
tu.hideturtle()
tu.pu()

# Save image

ts = tu.getscreen()
ts.getcanvas().postscript(file='Gene.eps')

tu.exitonclick()
```

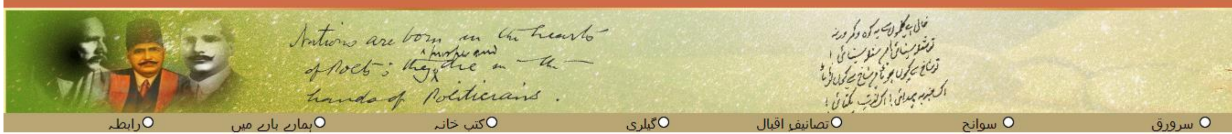
This little **Python** program allows us to study genes and understand how these interesting natural sequences developed (5). I hope this word will inspire you to "boldly go where no one has gone before." This brings us to the epilogue that combines for me the art, science, philosophy and inspiration for this book.

## References


1. https://stackoverflow.com/users/5179477/mohammad-saeed
2. Harris CR et al. Array programming with NumPy. Nature. 2020;585(7825):357-362. doi: 10.1038/s41586-020-2649-2. PMID: 32939066.
3. https://github.com/dr-saeed/OASIS/blob/master/OASIS.py
4. Saeed M. (2005). Fractals analysis of cardiac arrhythmias. TheScientificWorldJournal, 5, 691–701.
5. Saeed M. (2020). Fractal genomics of SOD1 Evolution. Immunogenetics, 72(9-10), 439–445.
6. https://github.com/dr-saeed/GeneFractals/blob/main/GeneFractals.py
7. File conversion from one format to another: https://convertio.co/

# Epilogue

Allama Muhammad Iqbal is the inspiration behind the creation of Pakistan. He did his PhD in Philosophy from Germany and bridged western and eastern thought with special emphasis on Islam (1). His message is of Love in action, symbolized by the Shaheen (falcon), under whose wings is the entire world yet it is unaffected by it. Its purpose is high just like its flight. The Love Iqbal talks about is not lip service but passionate action. It requires sacrifice and courage. Much of his work is in the form of poetry (Urdu and Persian) while his prose works are in English.




## عِلم و عِشق

علم نے مجھ سے کہا عشق ہے دیوانہ پن
عشق نے مجھ سے کہا علم ہے تخمین و ظن
بندۂ تخمین و ظن! کِرمِ کتابی نہ بن
عشق سراپا حضور، علم سراپا حِجاب!

عشق کی گرمی سے ہے معرکۂ کائنات
علم مقامِ صفات، عشق تماشائے ذات
عشق سکُون و ثبات، عشق حیات و ممات
علم ہے پیدا سوال، عشق ہے پنہاں جواب!

عشق کے ہیں معجزات سلطنت و فقر و دیں
عشق کے ادنیٰ غلام صاحبِ تاج و نگیں
عشق مکان و مکیں، عشق زمان و زمیں
عشق سراپا یقیں، اور یقیں فتحِ باب!

شرعِ محبّت میں ہے عشرتِ منزل حرام
شورشِ طُوفاں حلال، لذّتِ ساحل حرام
عشق پہ بجلی حلال، عشق پہ حاصل حرام
علم ہے اِبن الکتاب، عشق ہے اُمّ الکتاب!

His poem (above) is beautiful, a treat that will be missed by those who cannot read Urdu (2). To compensate a bit, the English translation (3) is provided below and processed by Python to determine the frequency of all the words in the

poem. Unfortunately, Python can only process Latinized languages such as English and German, but not other languages such as Arabic, Urdu and Persian that have a different script. Hopefully, in the future, this too might be possible.

**Knowledge and Love**

Knowledge said to me, Love is madness;
Love said to me, Knowledge is calculation
O slave of calculation, do not be a bookworm!
Love is Presence entire, Knowledge nothing but a Veil.
The universe is moved by the warmth of Love;
Knowledge deals with the Attributes, Love is a vision of the Essence;
Love is peace and permanence, Love is Life and Death:
Knowledge is the rising question, Love is the hidden answer.
Kingdom, faith and faqr are all miracles of Love;
The crowned kings and lords are base slaves of Love;
Love is the Space and the Creation, Love is Time and Earth!
Love is conviction entire, and conviction is the key!
The luxury of destination is forbidden in the religion of Love;
Fighting the storms is permitted, but the comfort of the shore is forbidden;
Lightning is permitted to Love, Harvest is forbidden.
Knowledge is the child of the Book, Love is the mother of the Book.

Translated by: K. A. Shafique

NB: Faqr means being ascetic

Let’s do some Python on this poem. It’ll ruin its aesthetic value but it will make the translation more mathematically accessible. Word search is an important function in Literature and frequency data of words help to understand the mind of the author.

The poem above is saved in a text file (Iqbal.txt) and processed to split it into words by separating them by white-spaces.

The simple thing to do is make a dictionary of words in the poem. The words will be the keys and their frequency will be the values.

Code to determine frequency count of the translation of Iqbal's poem, 'Knowledge and Love'.

```
words = {} # make a dictionary to store the words of the poem
poem = open ('Iqbal.txt', 'r') # open the file containing the poem

for line in poem:
   line = line.rstrip()
   word = line.split(' ') # separate words by white-spaces

   for w in word: # run a second for loop to determine if the word is already in the dictionary
      if w not in words:
         words[w] = 1 # the key will be the word and value its count
      else:
         words[w] +=1 # count is increased every time the same word is encountered

poem.close()

print ''
print words # print the dictionary
```

```
{'and': 8, 'the': 16, 'all': 1, 'Veil.': 1, 'Knowledge': 7, 'be': 1, 'calculation': 1, 'me,': 2, 'is': 20, 'child': 1, 'moved': 1, 'permanence,': 1, 'not': 1, 'answer.': 1, 'rising': 1, 'faqr': 1, 'in': 1, 'kings': 1, 'key!': 1, 'Book,': 1, 'permitted,': 1, 'Love;': 4, 'Life': 1, 'said': 2, 'Kingdom,': 1, 'comfort': 1, 'religion': 1, 'peace': 1, 'storms': 1, 'shore': 1, 'Death:': 1, 'permitted': 1, 'to': 3, 'forbidden': 1, 'Attributes,': 1, 'Love,': 1, 'bookworm!': 1, 'Harvest': 1, 'warmth': 1, 'miracles': 1, 'luxury': 1, 'do': 1, 'slave': 1, 'slaves': 1, 'question,': 1, 'universe': 1, 'Creation,': 1, 'but': 2, 'Fighting': 1, 'mother': 1, 'base': 1, 'Space': 1, 'O': 1, 'Time': 1, 'nothing': 1, 'crowned': 1, 'The': 3, 'Lightning': 1, 'with': 1, 'by': 1, 'entire,': 2, 'a': 3, 'faith': 1, 'hidden': 1, 'Love': 12, 'conviction': 2, 'Presence': 1, 'calculation,': 1, 'forbidden.': 1, 'deals': 1, 'destination': 1, 'Earth!': 1, 'lords': 1, 'of': 10, 'forbidden;': 1, 'Essence;': 1, 'madness;': 1, 'Book.': 1, 'vision': 1, 'are': 2}

>>>
```

Here, ends this book. Python can read and process the words of the poem of Iqbal, whether as a translation or in Latinized form, but it cannot **Love** nor can it produce the passionate **action**, Iqbal inspires us with. That, **YOU** have to do!

# References


1. http://www.allamaiqbal.com/biography/en/index.php
2. http://www.allamaiqbal.com/poetry.php?bookbup=23&orderno=398&lang_code=ur&lang=4&conType=ur
3. https://www.allamaiqbal.com/poetry.php?bookbup=23&orderno=398&lang_code=en&lang=2&conType=en

## About the Author

Dr. Mohammad Saeed is a Rheumatologist whose hobby is computer programming. He graduated from Aga Khan University Medical College, Karachi and completed a Postdoctoral Fellowship in Genetics at Northwestern University, Chicago. He completed his clinical training, including Residency in Internal Medicine and Fellowship in Rheumatology from University of Arkansas for Medical Sciences (UAMS). Additionally, he also did a Postdoctoral Fellowship in Immunology from UAMS and a Musculoskeletal Ultrasound Fellowship from the USSONAR program at Boston University. He has established ImmunoCure in Karachi, where he practices. ImmunoCure is a center for inflammatory diseases (Rheumatology) that is involved in providing excellence in clinical care and research.

Contact:

Website: www.immunocure.pk
Twitter: @DrMSaeed_pk, @ImmunoCure_pk
Email: msaeed@immunocure.pk