PROGRAMS

In this course, we are using Python as a tool for generating visualizations. However, Python is a general purpose programming language that can be used to write software to control the computer or another device (such as a robot) through the computer. Thus, by learning to produce visualizations you are also learning how to program computers.

The basic structure of Python program is as follows:

```
def main():
    <do something>
    <do something>
    ...
```

Creating a program is essentially the same as defining a function. We are just adopting a convention that all of our programs will be called **`main`**. In general, the structure of a program is as follows (we have provided line numbers so that we can refer to them:

```
Line 1: from pylab import *

Line 2: any other imports

Line 3: function definitions

Line 4: def main():
Line 5:     <do something>
Line 6:     <do something>
Line 7:     ...

Line 8: main()
```

Every visualization program will begin with the same line (`Line 1`). This, as you have seen, imports the Pylab library. If you are using any other libraries or modules they should be imported next (this is shown in `Line 2`). Import statements are then followed by the definitions of any functions (`Line 3`) and the definition of the function `main`. The last line (`Line 8)` is an invocation of the function `main`. This is placed so that when you load the program into the Python Shell, the program will immediately start executing. In order to illustrate this, let us write a function that uses the `plotDrugData` function from the previous chapter to generate several of plots:

```
# File: MutliPlot.py
# Purpose: To generate plots of all drug use versus time

# First import Pylab
from pylab import *

# Then import the data
from DrugData import *
```

```
# Define new functions

def plotDrugData(data, name):
    plot(Year, data)
    show()
    xlabel('Year')
    ylabel('Percentage')
    title(name + ' Drug use by High School Seniors 1979-2007')

# define the main program
def main():
    figure(1); plotDrugData(Marijuana, 'Marijuana')
    figure(2); plotDrugData(Cocaine, 'Cocaine')
    figure(3); plotDrugData(Heroin, 'Heroin')
    figure(4); plotDrugData(Hallucinogens, 'Hallucinogens')
    figure(5); plotDrugData(Alcohol, 'Alcohol')
    figure(6); plotDrugData(Cigarettes, 'Cigarettes')

# invoke main
main()
```

We have used a new command in the definition of the main function: **`figure(<#>)`**. Recall that figures are the windows in which Pylab generates a plot. Each figure is numbered (the first one opened is 1, the second 2, etc.). We can explicitly tell Pylab to open a new figure using the `figure` function, where the parameter is the number we wish to assign to the figure.

Do This: To run this program, start IDLE, create a new file, enter the program, and save it as `MultiPlot.py`. Then select `Run Module` from the `Run` menu in the text editor. Or, you could enter the following command in the Python shell:

```
>>> from MultiPlot.py import *
```

This statement is essentially equivalent to the Run Module option. When you run the program, notice that the program immediately generates all six graphics (comparisons of Drug use versus time).

SPEAKING PYTHON

We have jumped into writing fairly sophisticated programs without giving you any formal introduction to the nuts and bolts of Python. Here, we will provide a few more details about the language and help to demystify some of the gibberish you have entered or received back from the Python Shell.

What you mainly know about Python so far are actually a set of commands used to generate and format visualizations. These visualization commands are integrated into the main body of the Python environment through the use of the Pylab library. Python comes with many other libraries (or modules) that we will be using throughout the course. If you need to access commands in any library, all you need to do is import them.

Libraries are mainly made up of sets of functions, which provide the basic building blocks for any program. In most programming languages, functions are created from a pre-defined set of functions and mechanism (or syntax) for defining additional functions. In the case of Python, this is the `def` construct. By now, the creation of functions should be familiar to you. When using the `def` construct to define a new function, you must provided Python with two pieces of information: the commands that define the task the function is to accomplish, and the **name** of the function. Names are a critical part of programming and Python has rules about what is an acceptable name.

Names

A name is a label that is used to identify a user-defined element of a program so that it can be manipulated. A name in Python must begin with either an alphabetic letter (a-z or A-Z) or an underscore (i.e., _) and can be followed by any sequence of numbers, digits, or underscores. No spaces. No symbols other than the underscore. Python names are *case sensitive*, meaning that `myPlot` and `myplot` and `Myplot` are distinct names as far as Python is concerned. Once you create a name, you must consistently use the same spelling. So, now that we know how to create a name, what sorts of things should we name?

Well, we know that names can be used to represent functions. In other words, what the computer does each time you use a function name (like `plotDrugData`) is specified in the definition of the function. Names can also be used to represent other elements of a program. For example, you may want to represent a quantity by a name, such as an average value or the label for an axes. You did this when you defined the function `plotDrugData`, shown below:

```
def plotDrugData(data, name):
    plot(Year, data)
    show()
    xlabel('Year')
    ylabel('Percentage')
    title(name + ' Drug use by High School Seniors 1979-2007')
```

Functions take parameters (such as data and name) to help customize what they do. By parameterizing a function you are able to produce similar, but varying outcomes. In many ways, this is idea is similar to that of mathematical functions: sine(x), for example, computes the sine of whatever value you provide for x. However, for this to work, there has to be a way to define a function such that it is independent of specific parameter values. To do this, we use names to represent and designate specific values in a Python program. Names that represent values are called **variables**. What name you use is up to you, but in general, it is good programming practice to pick names that are easy to 1) read, 2) type, and 3) appropriately reflect the entity they represent.

Values

The ability to designate values by names is probably one of the most important features of programming. It is this facility that allows us to easily import, update, and otherwise manipulate information. Python provides a simple mechanism for designating values with names:

```
myFavoriteFood = 'curry'
x = 5
DowIndex = 12548.30.
```

Values can be **numbers** or **strings**. The above are examples of assignment statements in Python. The exact syntax of an assignment statement is:

```
<variable name> = <expression>
```

It reads as: *Let the variable named by* `<variable name>` *be assigned the value that is the result of calculating the expression* `<expression>`. So...what is an expression? The following are examples of expressions:

```
5
>>> 5 + 3
8
>>> 3 * 4
12
>>> 3.2 + 4.7
7.9
>>> 10 / 2
5
```

An ***expression*** is any Python command that returns a result. The simplest expression you can type is a number (as shown above). A number evaluates to itself. That is, a 5 is a 5. This special type of expression is called a ***literal***. When you enter an expression, Python evaluates it and returns a result (e.g., 5 + 3 → 8). Also, addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`) can be used on numbers to form expressions that involve numbers.

You may have also noticed that numbers can be written as whole numbers (3, 5, 10, 1655673, etc) or with decimal points (3.2, 0.5, etc) in them. Python (and most computer languages) distinguishes between these two ***types*** of numbers. Whole numbers are called ***integers*** and those with decimal points in them are called ***floating point*** numbers. While the arithmetic operations are defined on both kinds of numbers, there are some differences you should be aware of. For example:

```
>>> 10.0/3.0
3.3333333333333335
>>> 10/3
3
>>> 1/2
0
>>> 1.0/2
0.5
```

When you divide a floating point number by another floating point number, you get a floating point result. However, when you divide an integer by another integer, you get an integer result. Thus, in the examples above, you get the result 3.3333333333333335 when you divide 10.0 by 3.0, but you get 3 when you divide 10 by 3. Knowing this, the result of dividing 1 by 2 (see above) is zero (0) should not be surprising. That is, while the division operation looks the same (`/`), it treats integers differently than floating point values. However, if at least one of the numbers in an arithmetic operation is a floating point number, Python will give you a floating point result (see last example above).

There are many other types of values built into Python, including strings. Strings are text made up of sequences of characters. Python requires strings to be written enclosed in quotes, which can be single (`'I am a string.'`), double (`"I am a string, also."`), or even triple (`'''Me too!!!'''`). We will be talking about strings and related types in a later chapter.

Objects

Integers, floats, and strings are all data types that are an integral part of Python. Each type represents a certain set of values and has a set of associated operations that can be used to manipulate those values. In other words, data are *passive* entries manipulated and combined by *active* operations. This is the traditional approach to computation. However, to do more complex computation, it sometimes helps to view data in a different way.

Most modern programs are built using an **object-oriented (OO) approach**. Object-oriented programming is a very complex topic that involves many levels of abstraction. In this course, you will not be required to fully understand the OO approach, but you will have to use objects and so we need to spend some time discussing them.

The basic idea of object-oriented programming is to view a complex system as an interaction of ***objects***. For us, objects are a special sort of active data type that combines data and operations. They *know stuff* (they contain data) and they *do stuff* (they have operations). Operations that belong to objects are called **methods**.

We have already used several objects in our Python programming. In fact every element of the Pylab interface is an object: any plot, figure, and label that you have generated is an object!

To understand objects let's try looking at more detail in one of the objects built-in to the Pylab interface: **figures**. As discussed earlier, figures are the windows in which Pylab produces its visualizations. A figure is also an object, defined in the Pylab library and is thus an active data type that contains information and has operations. The information contained in the figure object relates to its size, position, color, and other aspects of its format. It also contains information that connects it to a plot that may be visualized within it. The methods of the object allow you to create figures and manipulate any of the data they contain.

Do This: In the Python Shell, create a figure as follows:

```
>>> figure()
```

Invoking this command produces a new figure (and actually produces it as well). A function that creates a new object when invoked is called a **constructor**. When we use a constructor to generate an object, we are **creating an instance of that object**. An constructor is a Python command that can be evaluated (i.e., it produces and output); thus, constructors are expressions and their output (instances) can be named using variables. For example, the following:

```
>>> fig = figure()
```

assigns the instance of a figure created by the constructor to the name `fig`. This ability to name objects gives Python and other object-oriented programming languages an amazing amount of flexibility. First of all, it allows us to have multiple instances of the same object. For example:

```
>>> fig1 = figure(1)
>>> fig2 = figure(2)
>>> myfig = figure(57)
```

We can pass a parameter to the figure() command which identifies the number of the figure (i.e., order in the list of opened figures) , thus creating multiple open figures at the same time. Each one can be then named by a variable.

Secondly, naming an object allows us to we can manipulate it and all the data that comprises that object with very little effort. As stated earlier, all objects know stuff (have data) and do stuff (have operations). Giving a name to an instance of an object allows us to do stuff to the object and its data. To perform operations on objects, we need to send the objects a message. Messages (or operations) which an object responds are its **methods**. We can send objects messages using **dot-notation**. For example, the following:

```
>>> fig.savefig('myfig.pdf')
```

Saves the figure `fig` to the pdf file `myfig.pdf`. There are many other messages we can send to the our figure. For example, the following will make the face color of the figure green:

```
>>> fig.set_facecolor('g')
```

A list of all the methods for a given Pylab object is listed in the Matplotlib class library, which is linked from the course website.

GRAPHICS

Because objects are very tenable in nature (i.e., they can be held on to and manipulated) , the easiest way to get familiar with them is often through the manipulation of graphical elements (e.g., points, lines, rectangles, circles, and text). For us, in our study of visualization, playing around with graphical elements is also important so that we can appreciate how much work goes into the programming of a **Graphical User Interface** (GUI) and a sophisticated plotting library such as Pylab.

Python is not a graphical language. In other words, the ability to produce graphics is not built-in to Python the way it is in other languages (such as Visual Basic). Thus, to produce graphical elements, we need to use a library. There are many such libraries, but we are just going to use a simple one, called **`graphics`** that is downloadable from the course website (in the links section). A link to full documentation for this library is also provided.

ALERT: To get this graphics library to work correctly, you need to open IDLE by double-clicking on your IDLE short-cut. If you choose to open IDLE by editing a .py file in IDLE, your graphics commands will freeze-up the system. Kind of the opposite bug as seen in Pylab!!!

To use the graphics library (or module) you need to download the file graphics.py and save it in your Python folder. Then, like any other module, you need to import it to access all of its features:

```
>>> from graphics import *
```

There are two types of objects defined in the graphics library: windows and objects to be drawn. The `GraphWin` class defines a window in which drawing can be done, and `GraphObjects` are provided that can be drawn in a `GraphWin`. Windows created by `GraphWin` are actually fairly sophisticated: they can even accept input from the mouse!

Do This: In a new file, create the following graphics program. This will later serve as a template for all graphics programs you will write in this course.

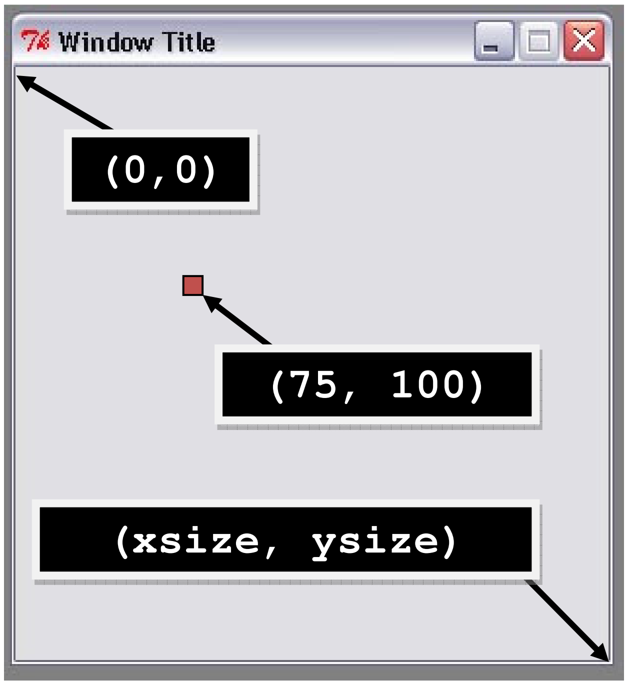
```
# import the graphics library
from graphics import *

# main program
def main():
    win = GraphWin('Window Title', 300, 300) # create a graphics window
    win.getMouse(); # wait for the mouse to be pressed in the window
    win.close() # close the graphics window

main()
```

The first step in any graphics program is to create a window. The command `GraphWin()` is the constructor for a graphics window object. It takes three parameters: the title and dimensions of the window. We then name the window created by assigning it to the variable **`win`**. This object instance can then be manipulated using the dot-notation and object methods. Run the module. The last two commands of the program provide away for us to close the window and still have time to view the window by telling Python to not do anything until the mouse is clicked in the window.

Try This: Comment-out the `win.getMouse()`command and re-run the program. What happens?

As it is, our window is not very interesting. So let's add some stuff. The graphics module provides defines additional objects that are representations of drawing elements. These include: **`Point`**, **`Line`**, **`Circle`**, **`Oval`**, **`Rectangle`**, **`Polygon`**, **`Text`**, and **`Image`**. Various attributes of graphical objects can be set, such as outline-color, fill-color, and line-width, in pretty much the same way you set line formats in Pylab.

Before we begin to draw objects, however, we first need to say a few things about our window. In particular, when we draw an object we need to be able to tell Python where in a window we want something drawn! So, how do we know where is where? Graphics windows use **coordinate systems** (just like mathematical coordinate systems) to identify locations in the windows. There are many different coordinate systems, but the default system used by your graphics window is as indicated below:



In this coordinate system, the ***upper left corner*** is the **origin** (0,0). The width and height of the coordinate system are defined by passing parameters to the constructor GraphWin(<title>, <xdimension>, <ydimension>). Any point in the system is defined by an x and y coordinate that measures how far it is away from the origin in each direction.

Do This: Now that we know how we can determine the position of an object in our window, we can begin to draw. Let's start by drawing a Point.

```
# import the graphics library
from graphics import *

# main program
def main():
    win = GraphWin('Window Title', 300, 300) # create a graphics window

    # draw a point
    p = Point(75, 100) # create the point
    p.draw(win) # draw the point

    win.getMouse(); # wait for the mouse to be pressed in the window
    win.close() # close the graphics window

main()
```

Run the module. Congratulations! You've just drawn the Point at location x = 75, y = 100 on the screen. A boring black Point, but a Point nonetheless. This was accomplished in two steps. First, we used a constructor to create a Point instance whose coordinates were (75, 100). We then used a Point method to draw the Point on the graphics window, `win`. In general the syntax for creating and drawing a Point is:

```
<point-name> = Point(<x-coordinate>, <y-coordinate>)
<point-name>.draw(<window-name>)
```

Once an instance of a Point object is created, we can also modify any of its properties through the use of methods that manipulate its features. For example, all objects are initially created with an unfilled black line. To make our point red, we could add the following command to our program:

```
p.setOutline('red')
```

Remember that the graphics library is different from Pylab and so manipulating properties and identifying colors is done a bit differently. All of the properties and available settings are listed in the ***graphics.py reference manual*** that is linked from the course website.

Do This: Now it is time to move on to something a little more complicated. Let's draw a line:

```
# import the graphics library
from graphics import *

# main program
def main():
    win = GraphWin('Window Title', 300, 300) # create a graphics window

    myLine = Line(Point(0,0), Point(200, 200)) # create a line
    myLine.draw(win) # draw the line in the window

    win.getMouse(); # wait for the mouse to be pressed in the window
    win.close() # close the graphics window

main()
```

Note how drawing a line is much the same as drawing a point; however, instead of providing one coordinate, we provide two: the start and end of the line. The graphics library also provides some interesting functions just for formatting lines. For example, the `setArrow(<type>)` method allows you to add an arrow to the end of the line.

The exact syntax for creating a line is:

```
<line-name> = Line(<point 1>, <point 2>)
<line-name>.draw(<window-name>)
```

In other words, you can define a line using two points that you have already created.

Creating two dimensional graphical elements is done in the exact same way that the one dimensional elements are created. For example, to plot a circle in this program, you would add the following statements:

```
c = Circle(Point(100,100), 50)
c.draw(win)
```

The syntax for creating a circle is:

```
<circle-name> = Circle(<center-point>, radius)
<circle-name>.draw(<window-name>)
```

Likewise, for creating a Rectangle you would use the following syntax:

```
<rectangle-name> = Rectangle(<point 1>, <point 2>)
<rectangle-name>.draw(<window-name>)
```

where the points define opposite corners of the rectangle. An Oval is defined in the same way.

And again, for each object you can modify its properties: outline, fill, width, etc.

Read through the reference manual (~5 pages) to familiarize yourself with the rest of the graphics objects and methods.

PYTHON REVIEW

In this chapter we learned a lot about Python syntax, including the following:

- The basic structure of a python program is as follows:

```
# import modules
# define functions

def main(): # define main function
    <do something>
    <do something>
     ...

main() # invoke main function
```

- **Names** are labels used to identify user-defined elements of a program. Names must start with an alphabetic letter (A-Z, a-z) or an underscore and can be followed by any sequence of letters, digits, or undercores. No spaces. No symbols.

- Any command that returns a value is an **expression**.

- **Values** can be assigned to names using assignment statements:
  `<variable-name> = expression`

PYLAB REVIEW

In this chapter we learned several Pylab commands. These included the following:

- The graphics window in which Pylab produces in plots is called a figure. Figures are objects that can be manipulated by the following commands:
  - **`figure(<number>)`**: creates a new figure, titled: `Figure <number>`. If number is not provided, the figure is titled `Figure 1`. The command is a **constructor** that returns a pointer to an instance of a figure and can thus be assigned a name.

  - **`<figure-name>.savefig('myfig.pdf')`**: saves the figure `<figure-name>` the file `myfig.pdf`.

  - **`<figure-name>.set_facecolor(<color>)`**: sets the background color of the figure

GRAPHICS LIBRARY REVIEW

In this chapter we were introduced to basics of creating and drawing in a Graphics window using the **`graphics.py`** module. We learned the following:

- Drawing is done in Graphics Window Objects, which can be manipulated with the following commands:

  - `<window-name> = GraphWin('Window Title', <xdimension>, <ydimension>)`: creates an instance of a graphic window object of dimension <xdimension> and <ydimension> (in pixels), titles it and assigns to the variable <window-name>.
  - `<window-name>.getMouse()`: waits for the mouse button to be clicked in the window
  - `<window-name>.close()`: closes the window

- Once a window is created, Graphics Objects can be drawn in the window using the following commands:
  - Points:
    - `<point-name> = Point(<x-coordinate>, <y-coordinate`
    - `<point-name>.draw(<window-name>)`
  - Lines:
    - `<line-name> = Line(<point1>, <point2>)`
    - `<line-name>.draw(<window-name>)`
  - Rectangles:
    - `<rectangle-name> = Rectangle(<point1>, <point2>)`
    - `<rectangle-name>.draw(<window-name>)`
  - Circles:
    - `<circle-name> = Circle(<center-point>, radius)`
    - `<circle-name>.draw(<window-name>)`

EXERCISES

NOTE: You will not be using Pylab for these Exercises!

1. There are 2.54 centimeters in every inch. Write a function that takes a measurement in feet and converts it to a measure in meters. The function should return the calculated result. (10 pts)

2. Develop a top-down design to help solve Exercise #4. (10 pts)

3. Use the graphing library to write a program that generates a line and scatter plot of five points of data: (100, 50), (72, 10), (30, 27), (50, 13), (90, 43). The program should graph the points relative to a set of axes (i.e., a vertical and horizontal line) and label each axes. You do not need to worry about tick marks. Extra credit will be given if you visualize each data point with a Circle or Rectangle or other Polygon (i.e. diamond or triangle) instead of a Point object. Submit a screen snap-shot or printout of your figure along with your code. (30 pts)

4. Develop a top-down design to help solve Exercise #6. (10 pts)

5. Use the graphing library to write a program that draws a self-portrait. Submit a print-out or screen-snapshot of your portrait along with your code. (30 pts)