Python Programming 101

1

This Computer Science text further develops the skills you learned in your first CS text or course and adds to your bag of tricks by teaching you how to use efficient algorithms for dealing with large amounts of data. Without the proper understanding of efficiency, it is possible to bring even the fastest computers to a grinding halt when working with large data sets. This has happened before, and soon you will understand just how easy it can occur. But first, we'll review some patterns for programming and look at the Python programming language to make sure you understand the basic structure and syntax of the language.

To begin writing programs using Python you need to install Python on your computer. The examples in this text use Python 3. Python 2 is not compatible with Python 3 so you'll want to be sure you have Python 3 or later installed on your computer. When writing programs in any language a good Integrated Development Environment (IDE) is a valuable tool so you'll want to install an IDE, too. Examples within this text will use Wing IDE 101 as pictured in Fig. 1.1, although other acceptable IDEs are available as well. The Wing IDE is well maintained, simple to use, and has a nice debugger which will be useful as you write Python programs. If you want to get Wing IDE 101 then go to http://wingware.com. The website http://cs.luther.edu/~leekent/CS1 has directions for installing both Python 3 and Wing IDE 101. Wing IDE 101 is the free version of Wing for educational use.

There are some general concepts about Python that you should know when reading the text. Python is an interpreted language. That means that you don't have to go through any extra steps after writing Python code before you can run it. You can simply press the debug button in the Wing IDE (it looks like an insect) and it will ask you to save your program if you haven't already done so at least once. Then it will run your program. Python is also dynamically typed. This means that you will not get any type errors before you run your program as you would with some programming languages. It is especially important for you to understand the types of data you are using in your program. More on this in just a bit. Finally, your Python programs are interpreted by the Python interpreter. The shell is another name for the Python interpreter and Wing IDE 101 gives you access to a shell within the IDE

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,

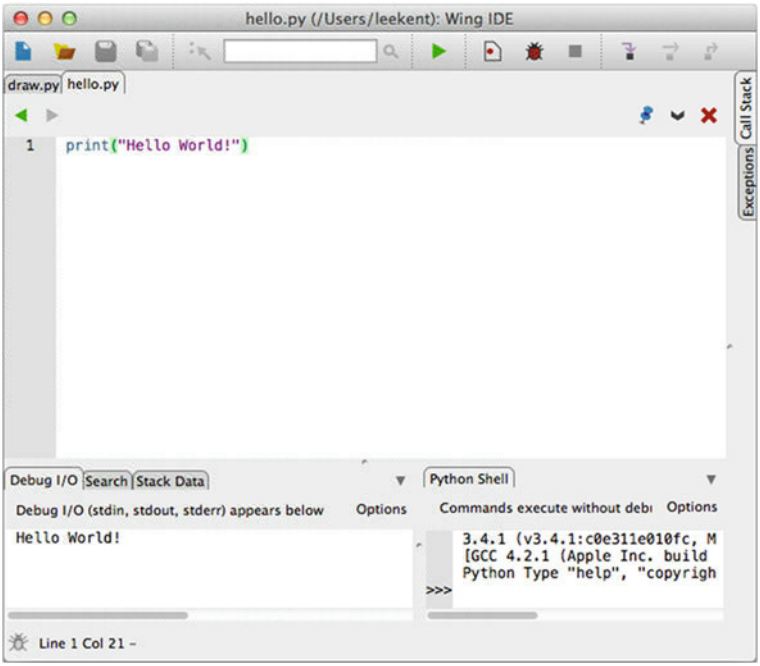

Fig. 1.1 The Wing IDE

itself. You can type Python statements and expressions into the window pane that says *Python Shell* to quickly try out a snippet of code before you put it in a program.

Like most programming languages, there are a couple kinds of errors you can get in your Python programs. Syntax errors are found before your program runs. These are things like missing a colon or forgetting to indent something. An IDE like Wing IDE 101 will highlight these syntax errors so you can correct them. Run-time errors are found when your program runs. Run-time errors come from things like variables with unexpected values and operations on these values. To find a run-time error you can look at the *Stack Data* tab as it appears in Fig. 1.1. When a run-time error occurs the program will stop executing and the *Stack Data* tab will let you examine the run-time stack where you can see the program variables.

In the event that you still don't understand a problem, the Wing IDE 101 (and most other IDEs) lets you step through your code so you can watch as an error is reproduced. The three icons in the upper right corner of Fig. 1.1 let you *Step Into* a function, *Step Over* code, and *Step Out Of* a function, respectively. Stepping over or into your code can be valuable when trying to understand a run-time error and how it occurred.

One other less than obvious tool is provided by the Wing IDE. By clicking on the line number on the left side of the IDE it is possible to set a breakpoint. A breakpoint causes the program to stop execution just before the breakpoint. From there it is possible to begin stepping over your code to determine how an error occurred.

To motivate learning or reviewing Python in this chapter, the text will develop a simple drawing application using turtle graphics and a Graphical User Interface (GUI) framework called Tkinter. Along the way, you'll discover some patterns for programming including the accumulator pattern and the loop and a half pattern for reading records from a file. You'll also see functions in Python and begin to learn how to implement your own datatypes by designing and writing a class definition.

1.1 Chapter Goals

By the end of this chapter, you should be able to answer these questions.

- What two parts are needed for the accumulator pattern?
- When do you need to use the loop and a half pattern for reading from a file?
- What is the purpose of a class definition?
- What is an object and how do we create one?
- What is a mutator method?
- What is an accessor method?
- What is a widget and how does one use widgets in GUI programming?

1.2 Creating Objects

Python is an object-oriented language. All data items in Python are objects. In Python, data items that could be thought of as similar are named by a type or class. The term *type* and *class* in Python are synonymous: they are two names for the same thing. So when you read about *types* in Python you can think of *classes* or vice versa.

There are several built-in types of data in Python including *int*, *float*, *str*, *list*, and *dict* which is short for dictionary. These types of data and their associated operations are included in the appendices at the end of the text so you have a quick reference if you need to refer to it while programming. You can also get help for any type by typing *help(typename)* in the Python shell, where *typename* is a type or class in Python. A very good language reference can be found at http://python.org/doc, the official Python documentation website.

1.2.1 Literal Values

There are two ways to create objects in Python. In a few cases, you can use a literal value to create an object. Literal values are used when we want to set some variable to a specific value within our program. For example, the literal 6 denotes any object with the integer value of 6.

```
x = 6
```

This creates an *int* object containing the value 6. It also points the reference called *x* at this object as pictured in Fig. 1.2. All assignments in Python point references

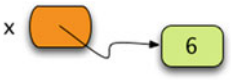

Fig. 1.2 A Reference and Object

at objects. Any time you see an assignment statement, you should remember that the thing on the left side of the equals sign is a reference and the thing on the right side is either another reference or a newly created object. In this case, writing $x = 6$ makes a new object and then points x at this object.

Other literal values may be written in Python as well. Here are some literal values that are possible in Python.

- *int* literals: 6, 3, 10, −2, etc.
- *float* literals: 6.0, −3.2, 4.5E10
- *str* literals: 'hi there', "how are you"
- *list* literals: [], [6, 'hi there']
- *dict* literals: {}, {'hi there':6, 'how are you':4}

Python lets you specify *float* literals with an exponent.

So, *4.5E10* represents the *float* 45000000000.0. Any number written with a decimal point is a *float*, whether there is a 0 or some other value after the decimal point. If you write a number using the *E* or exponent notation, it is a float as well. Any number without a decimal point is an *int*, unless it is written in *E* notation. String literals are surrounded by either single or double quotes. List literals are surrounded by [and]. The [] literal represents the empty list. The {} literal is the empty dictionary.

You may not have previously used dictionaries. A dictionary is a mapping of keys to values. In the dictionary literal, the key 'hi there' is mapped to the value 6, and the key 'how are you' is mapped to 4. Dictionaries will be covered in some detail in Chap. 5.

1.2.2 Non-literal Object Creation

Most of the time, when an object is created, it is not created from a literal value. Of course, we need literal values in programming languages, but most of the time we have an object already and want to create another object by using one or more existing objects. For instance, if we have a string in Python, like '6' and want to create an *int* object from that string, we can do the following.

```
y = '6'
x = int(y)
print(x)
```

In this short piece of code, *y* is a reference to the *str* object created from the string literal. The variable *x* is a reference to an object that is created by using the object that *y* refers to. In general, when we want to create an object based on other object values we write the following:

```
variable = type(other_object_values)
```

The *type* is any type or class name in Python, like *int*, *float*, *str* or any other type. The *other_object_values* is a comma-separated sequence of references to other objects that are needed by the class or type to create an instance (i.e. an object) of that type. Here are some examples of creating objects from non-literal values.

```
z = float('6.3')
w = str(z)
u = list(w) # this results in the list ['6', '.', '3']
```

1.3 Calling Methods on Objects

Objects are useful because they allow us to collect related information and group them with behavior that act on this data. These behaviors are called *methods* in Python. There are two kinds of methods in any object-oriented language: *mutator* and *accessor* methods. *Accessor* methods access the current state of an object but don't change the object. *Accessor* methods return new object references when called.

```
x = 'how are you'
y = x.upper()
print(y)
```

Here, the method *upper* is called on the object that *x* refers to. The *upper* accessor method returns a new object, a *str* object, that is an upper-cased version of the original string. Note that *x* is not changed by calling the *upper* method on it. The *upper* method is an accessor method. There are many accessor methods available on the *str* type which you can learn about in the appendices.

Some methods are mutator methods. These methods actually change the existing object. One good example of this is the *reverse* method on the *list* type.

```
myList = [1, 2, 3]
myList.reverse()
print(myList) # This prints [3, 2, 1] to the screen
```

The *reverse* method mutates the existing object, in this case the list that *myList* refers to. Once called, a mutator method can't be undone. The change or mutation is permanent until mutated again by some other mutator method.

All classes contain accessor methods. Without accessor methods, the class would be pretty uninteresting. We use accessor methods to retrieve a value that is stored in an object or to retrieve a value that depends on the value stored in an object.

If a class had no accessor methods we could put values in the object but we could never retrieve them.

Some classes have mutator methods and some don't. For instance, the *list* class has mutator methods, including the *reverse* method. There are some classes that don't have any mutator methods. For instance, the *str* class does not have any mutator methods. When a class does not contain any mutator methods, we say that the class is *immutable*. We can form new values from the data in an *immutable* class, but once an immutable object is created, it cannot be changed. Other immutable classes include *int* and *float*.

1.4 Implementing a Class

Programming in an object-oriented language usually means implementing classes that describe objects which hold information that is needed by the program you are writing. Objects contain data and methods operate on that data. A *class* is the definition of the *data* and *methods* for a specific type of *object*.

Every class contains one special method called a constructor. The constructor's job is to create an instance of an object by placing references to data within the object itself. For example, consider a class called Dog. A dog has a name, a birthday, and a sound it makes when it barks. When we create a Dog object, we write code like that appearing in Sect. 1.4.1.

1.4.1 Creating Objects and Calling Methods

```
1 boyDog = Dog("Mesa", 5, 15, 2004, "WOOOF")
2 girlDog = Dog("Sequoia", 5, 6, 2004, "barkbark")
3 print(boyDog.speak())
4 print(girlDog.speak())
5 print(boyDog.birthDate())
6 print(girlDog.birthDate())
7 boyDog.changeBark("woofywoofy")
8 print(boyDog.speak())
```

Once created in the memory of the computer, dog objects looks like those appearing in Fig. 1.3. Each object is referenced by the variable reference assigned to it, either *girlDog* or *boyDog* in this case. The objects themselves are a collection of references that point to the information that is stored in the object. Each object has name, month, day, year, and speakText references that point to the associated data that make up a Dog object.

To be able to create *Dog* objects like these two objects we need a *Dog* class to define these objects. In addition, we'll need to define *speak*, *birthDate*, and *changeBark* methods. We can do this by writing a class as shown in Sect. 1.4.2. Comments about each part of the class appear in the code. The special variable *self* always points at the current object and must be the first parameter to each method in the class.

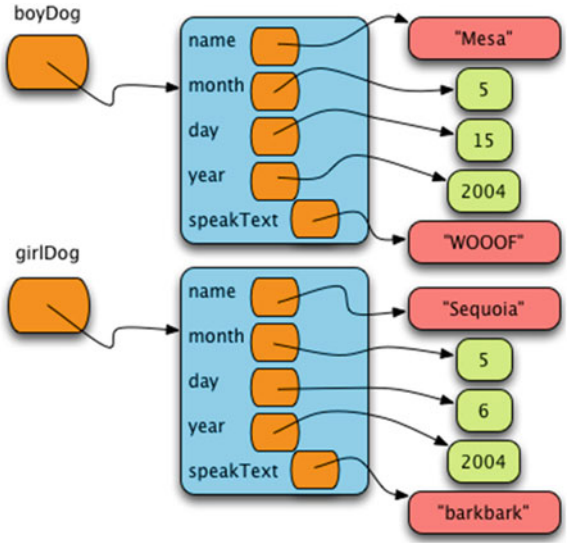

Fig. 1.3 A Couple of Dog Objects

Python takes care of passing the *self* argument to the methods. The other arguments are passed by the programmer when the method is called (see the example of calling each method in Sect. 1.4.1).

1.4.2 The Dog Class

```
class Dog:
    # This is the constructor for the class. It is called whenever a Dog
    # object is created. The reference called "self" is created by Python
    # and made to point to the space for the newly created object. Python
    # does this automatically for us but we have to have "self" as the first
    # parameter to the __init__ method (i.e. the constructor).
    def __init__(self, name, month, day, year, speakText):
        self.name = name
        self.month = month
        self.day = day
        self.year = year
        self.speakText = speakText

    # This is an accessor method that returns the speakText stored in the
    # object. Notice that "self" is a parameter. Every method has "self" as its
    # first parameter. The "self" parameter is a reference to the current
    # object. The current object appears on the left hand side of the dot (i.e.
    # the .) when the method is called.
    def speak(self):
        return self.speakText

    # Here is an accessor method to get the name
    def getName(self):
```

```
        return self.name

    # This is another accessor method that uses the birthday information to
    # return a string representing the date.
    def birthDate(self):
        return str(self.month) + "/" + str(self.day) + "/" + str(self.year)

    # This is a mutator method that changes the speakText of the Dog object.
    def changeBark(self,bark):
        self.speakText = bark
```

1.5 Operator Overloading

Python provides operator overloading, which is a nice feature of programming languages because it makes it possible for the programmer to interact with objects in a very natural way. Operator overloading is already implemented for a variety of the built-in classes or types in Python. For instance, integers (i.e. the *int* type) understand how they can be added together to form a new integer object. Addition is implemented by a special method in Python called the *__add__* method. When two integers are added together, this method is called to create a new integer object. If you look in the appendices, you'll see examples of these special methods and how they are called. For example, in Chap. 13 the *__add__* method is called by writing $x + y$ where x is an integer. The methods that begin and end with two underscores are methods that Python associates with a corresponding operator.

When we say that Python supports operator *overloading* we mean that if you define a method for your class with a name that is operator overloaded, your class will support that operator as well. Python figures out which method to call based on the types of the operands involved. For instance, writing $x + y$ calls the *int* class *__add__* method when x is an integer, but it calls the *float* type's *__add__* method when x is a *float*. This is because in the case of the *__add__* method, the object on the left hand side of the + operator corresponds to the object on the left hand side of the dot (i.e. the period) in the equivalent method call $x.\_\_add\_\_(y)$. The object on the left side of the dot determines which add method is called. The + operator is overloaded.

If we wanted to define addition for our *Dog* class, we would include an *__add__* method in the class definition. It might be natural to write *boyDog* + *girlDog* to create a new puppy object. If we wished to do that we would extend our Dog class as shown in Sect. 1.5.1.

1.5.1 The Dog Class with Overloaded Addition

```
class Dog:
    # This is the constructor for the class. It is called whenever a Dog
    # object is created. The reference called "self" is created by Python
    # and made to point to the space for the newly created object. Python
    # does this automatically for us but we have to have "self" as the first
    # parameter to the __init__ method (i.e. the constructor).
```

```
    def __init__(self, name, month, day, year, speakText):
        self.name = name
        self.month = month
        self.day = day
        self.year = year
        self.speakText = speakText

    # This is an accessor method that returns the speakText stored in the
    # object. Notice that "self" is a parameter. Every method has "self" as its
    # first parameter. The "self" parameter is a reference to the current
    # object. The current object appears on the left hand side of the dot (i.e.
    # the .) when the method is called.
    def speak(self):
        return self.speakText

    # Here is an accessor method to get the name
    def getName(self):
        return self.name

    # This is another accessor method that uses the birthday information to
    # return a string representing the date.
    def birthDate(self):
        return str(self.month) + "/" + str(self.day) + "/" + str(self.year)

    # This is a mutator method that changes the speakText of the Dog object.
    def changeBark(self,bark):
        self.speakText = bark

    # When creating the new puppy we don't know it's birthday. Pick the
    # first dog's birthday plus one year. The speakText will be the
    # concatenation of both dog's text. The dog on the left side of the +
    # operator is the object referenced by the "self" parameter. The
    # "otherDog" parameter is the dog on the right side of the + operator.
    def __add__(self,otherDog):
        return Dog("Puppy of " + self.name + " and " + otherDog.name, \
                   self.month, self.day, self.year + 1, \
                   self.speakText + otherDog.speakText)

def main():
    boyDog = Dog("Mesa", 5, 15, 2004, "WOOOOF")
    girlDog = Dog("Sequoia", 5, 6, 2004, "barkbark")
    print(boyDog.speak())
    print(girlDog.speak())
    print(boyDog.birthDate())
    print(girlDog.birthDate())
    boyDog.changeBark("woofywoofy")
    print(boyDog.speak())
    puppy = boyDog + girlDog
    print(puppy.speak())
    print(puppy.getName())
    print(puppy.birthDate())

if __name__ == "__main__":
    main()
```

This text uses operator overloading fairly extensively. There are many operators that are defined in Python. Python programmers often call these operators *Magic Methods* because a method automatically gets called when an operator is used in an expression. Many of the common operators are given in the table in Fig. 1.4 for your

Method Defintion	Operator	Description
__add__(self,y)	x + y	The addition of two objects. The type of *x* determines which add operator is called.
__contains__(self,y)	y in x	When *x* is a collection you can test to see if *y* is in it.
__eq__(self,y)	x == y	Returns *True* or *False* depending on the values of *x* and *y*.
__ge__(self,y)	x >= y	Returns *True* or *False* depending on the values of *x* and *y*.
__getitem__(self,y)	x[y]	Returns the item at the y^{th} position in x.
__gt__(self,y)	x > y	Returns *True* or *False* depending on the values of *x* and *y*.
__hash__(self)	hash(x)	Returns an integral value for *x*.
__int__(self)	int(x)	Returns an integer representation of *x*.
__iter__(self)	for v in x	Returns an iterator object for the sequence *x*.
__le__(self,y)	x <= y	Returns *True* or *False* depending on the values of *x* and *y*.
__len__(self)	len(x)	Returns the size of *x* where *x* has some length attribute.
__lt__(self,y)	x < y	Returns *True* or *False* depending on the values of *x* and *y*.
__mod__(self,y)	x % y	Returns the value of *x* modulo *y*. This is the remainder of *x/y*.
__mul__(self,y)	x * y	Returns the product of *x* and *y*.
__ne__(self,y)	x != y	Returns *True* or *False* depending on the values of *x* and *y*.
__neg__(self)	-x	Returns the unary negation of *x*.
__repr__(self)	repr(x)	Returns a string version of x suitable to be evaluated by the *eval* function.
__setitem__(self,i,y)	x[i] = y	Sets the item at the i^{th} position in *x* to *y*.
__str__(self)	str(x)	Return a string representation of *x* suitable for user-level interaction.
__sub__(self,y)	x - y	The difference of two objects.

Fig. 1.4 Python Operator Magic Methods

convenience. For each operator the magic method is given, how to call the operator is given, and a short description of it as well. In the table, *self* and *x* refer to the same object. The type of *x* determines which operator method is called in each case in the table.

The *repr*(*x*) and the *str*(*x*) operators deserve a little more explanation. Both operators return a string representation of *x*. The difference is that the *str* operator should return a string that is suitable for human interaction while the *repr* operator is called when a string representation is needed that can be evaluated. For instance, if we wanted to define these two operators on the *Dog* class, the *repr* method would return the string "Dog('Mesa', 5,15,2004, 'WOOOF')" while the *str* operator might return just the dog's name. The *repr* operator, when called, will treat the string as an expression that could later be evaluated by the *eval* function in Python whereas the *str* operator simply returns a string for an object.

1.6 Importing Modules

In Python, programs can be broken up into modules. Typically, when you write a program in Python you are going to use code that someone else wrote. Code that others wrote is usually provided in a module. To use a module, you import it. There are two ways to import a module. For the drawing program we are developing in this chapter, we want to use turtle graphics. Turtle graphics was first developed a long time ago for a programming language called Logo. Logo was created around 1967 so the basis for turtle graphics is pretty ancient in terms of Computer Science. It still

remains a useful way of thinking about Computer Graphics. The idea is that a turtle is wandering a beach and as it walks around it drags its tail in the sand leaving a trail behind it. All that you can do with a turtle is discussed in the Chap. 18.

There are two ways to import a module in Python: the *convenient* way and the *safe* way. Which way you choose to import code may be a personal preference, but there are some implications about using the *convenient* method of importing code. The convenient way to import the turtle module would be to write the following.

```
from turtle import *
t = Turtle()
```

This is convenient, because whenever you want to use the *Turtle* class, you can just write *Turtle* which is convenient, but not completely safe because you then have to make sure you never use the identifier *Turtle* for anything else in your code. In fact, there may be other identifiers that the turtle module defines that you are unaware of that would also be identifiers you should not use in your code. The safe way to import the turtle module would be as follows.

```
import turtle
t = turtle.Turtle()
```

While this is not quite as *convenient*, because you must precede *Turtle* with "*turtle.*", it is *safe* because the *namespace* of your module and the turtle module are kept separate. All identifiers in the turtle module are in the *turtle namespace*, while the local identifiers are in the *local namespace*. This idea of *namespaces* is an important feature of most programming languages. It helps programmers keep from stepping on each others' toes. The rest of this text will stick to using the safe method of importing modules.

1.7 Indentation in Python Programs

Indentation plays an important role in Python programs. An indented line belongs to the line it is indented under. The *body* of a function is indented under its function definition line. The *then* part of an *if* statement is indented under the *if*. A *while* loop's body is indented under it. The methods of a class are all indented under the class definition line. All statements that are indented the same amount and grouped together are called a *block*. It is important that all statements within a *block* are indented exactly the same amount. If they are not, then Python will complain about inconsistent indentation.

Because indentation is so important to Python, the Wing IDE 101 lets you select a series of lines and adjust their indentation as a group, as shown in Fig. 1.5. You first select the lines of the block and then press the *tab* key to increase their indentation. To decrease the indentation of a block you select the lines of the block and press *Shift-tab*. As you write Python code this is a common chore and being able to adjust the indentation of a whole block at a time is a real timesaver.

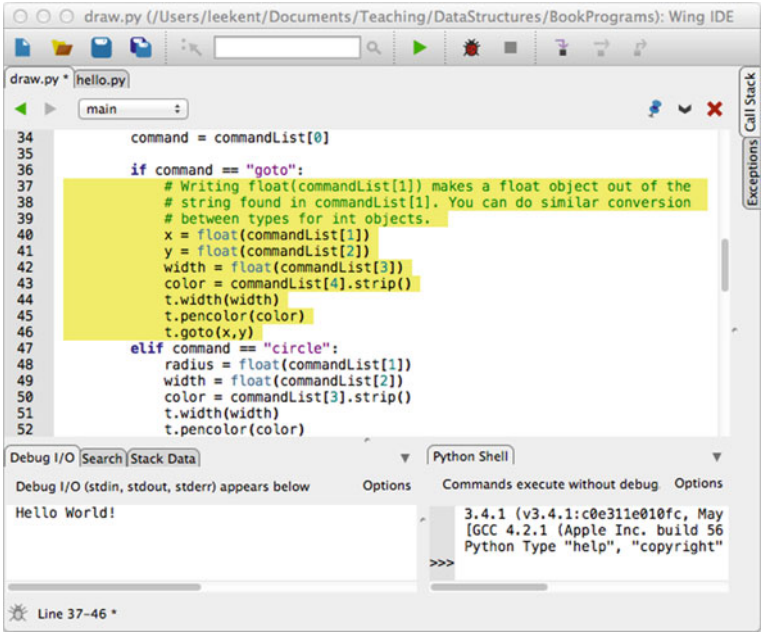

Fig. 1.5 Adjusting Indentation in Wing IDE 101

1.8 The *Main* Function

Programs are typically written with many function definitions and function calls. One function definition is written by convention in Python, usually called the *main* function. This function contains code the program typically executes when it is first started. The general outline of a Python program is given in Sect. 1.8.1.

1.8.1 Python Program Structure

```
# Imports at the top.
import turtle

# other function definitions followed by the main function definition
def main():
    # The main code of the program goes here
    t = turtle.Turtle()

# this code calls the main function to get everything started. The condition in this
# if statement evaluates to True when the module is executed by the interpreter, but
# not when it is imported into another module.
if __name__ == "__main__":
    main()
```

The *if* statement at the end of the code in Sect. 1.8.1 is the first code executed after the import statements. The *if* statement's condition evaluates to *True* when the

program is run as a stand-alone program. Sometimes we write modules that we may want to import into another module. Writing this *if* statement to call the main function makes the module execute its own main function when it is run as a stand-alone program. When the module is imported into another module it will not execute its main function. Later you will have the opportunity to write a module to be imported into another module so it is a good habit to form to always call the *main* function in this way.

1.9 Reading from a File

To begin our drawing program, let's assume that a picture is stored in a file and we wish to read this file when the program is started. We'll assume that each line of the file contains a drawing command and its associated data. We'll keep it simple and stick to drawing commands that look like this in the input file:

- goto, x, y, width, color
- circle, radius, width, color
- beginfill, color
- endfill
- penup
- pendown

Each line of the file will contain a record with the needed information. We can draw a picture by providing a file with the right sequence of these commands. The file in Sect. 1.9.1 contains records that describe a pickup truck.

1.9.1 A Text File with Single Line Records

```
beginfill, black
circle, 20, 1, black
endfill
penup
goto, 120, 0, 1, black
pendown
beginfill, black
circle, 20, 1, black
endfill
penup
goto, 150, 40, 1, black
pendown
beginfill, yellow
goto, -30, 40, 1, black
goto, -30, 70, 1, black
goto, 60, 70, 1, black
goto, 60, 100, 1, black
goto, 90, 100, 1, black
goto, 115, 70, 1, black
goto, 150, 70, 1, black
```

```
goto, 150, 40, 1, black
endfill
```

To process the records in the file in Sect. 1.9.1, we can write a Python program that reads the lines of this file and does the appropriate turtle graphics commands for each record in the file. Since each record (i.e. drawing command) is on its own line in the file format described in Sect. 1.9.1, we can read the file by using a *for* loop to read the lines of the file. The code of Sect. 1.9.2 is a program that reads these commands and processes each record in the file, drawing the picture that it contains.

1.9.2 Reading and Processing Single Line Records

```
# This imports the turtle graphics module.
import turtle

# The main function is where the main code of the program is written.
def main():
    # This line reads a line of input from the user.
    filename = input("Please enter drawing filename: ")

    # Create a Turtle Graphics window to draw in.
    t = turtle.Turtle()
    # The screen is used at the end of the program.
    screen = t.getscreen()

    # The next line opens the file for "r" or reading. "w" would open it for
    # writing, and "a" would open the file to append to it (i.e. add to the
    # end). In this program we are only interested in reading the file.
    file = open(filename, "r")

    # The following for loop reads the lines of the file, one at a time
    # and executes the body of the loop once for each line of the file.
    for line in file:

        # The strip method strips off the newline character at the end of the line
        # and any blanks that might be at the beginning or end of the line.
        text = line.strip()

        # The following line splits the text variable into its pieces.
        # For instance, if text contained "goto, 10, 20, 1, black" then
        # commandList will be equal to ["goto", "10", "20", "1", "black"] after
        # splitting text.
        commandList = text.split(",")

        # get the drawing command
        command = commandList[0]

        if command == "goto":
            # Writing float(commandList[1]) makes a float object out of the
            # string found in commandList[1]. You can do similar conversion
            # between types for int objects.
            x = float(commandList[1])
            y = float(commandList[2])
            width = float(commandList[3])
            color = commandList[4].strip()
            t.width(width)
            t.pencolor(color)
```

```
            t.goto(x,y)
        elif command == "circle":
            radius = float(commandList[1])
            width = float(commandList[2])
            color = commandList[3].strip()
            t.width(width)
            t.pencolor(color)
            t.circle(radius)
        elif command == "beginfill":
            color = commandList[1].strip()
            t.fillcolor(color)
            t.begin_fill()
        elif command == "endfill":
            t.end_fill()
        elif command == "penup":
            t.penup()
        elif command == "pendown":
            t.pendown()
        else:
            print("Unknown command found in file:",command)

    #close the file
    file.close()

    #hide the turtle that we used to draw the picture.
    t.ht()

    # This causes the program to hold the turtle graphics window open
    # until the mouse is clicked.
    screen.exitonclick()
    print("Program Execution Completed.")


# This code calls the main function to get everything started.
if __name__ == "__main__":
    main()
```

When you have a data file where each line of the file is its own separate record, you can process those records as we did in Sect. 1.9.2. The general pattern is to open the file, use a for loop to iterate through the file, and have the body of the for loop process each record. The pseudo-code in Sect. 1.9.3 is the abstract pattern for reading one-line records from a file.

1.9.3 Pattern for Reading Single Line Records from a File

```
# First the file must be opened.
file = open(filename,"r")

# The body of the for loop is executed once for each line in the file.
for line in file:
    # Process each record of the file. Each record must be exactly one line of the
    # input file. What processing a record means will be determined by the
      # program you are writing.
    print(line)

# Closing the file is always a good idea, but it will be closed when your program
```

```
# terminates if you do not close it explicitly.
file.close()
```

1.10 Reading Multi-line Records from a File

Sometimes records of a file are not one per line. Records of a file may cross multiple lines. In that case, you can't use a *for* loop to read the file. You need a *while* loop instead. When you use a while loop, you need to be able to check a condition to see if you are done reading the file. But, to check the condition you must first try to read at least a little of a record. This is a kind of chicken and egg problem. Which came first, the chicken or the egg? Computer programmers have a name for this problem as it relates to reading from files. It is called the *Loop and a Half Pattern*. To use a while loop to read from a file, we need a loop and a half. The half comes before the while loop.

Consider the program we are writing in this chapter. Let's assume that the records of the file cross multiple lines. In fact, let's assume that we have variable length records. That is, the records of our file consist of one to five lines. The drawing commands will be exactly as they were before. But, instead of all the data for a record appearing on one line, we'll put each piece of data on its own separate line as shown in Sect. 1.10.1.

1.10.1 A Text File with Multiple Line Records

```
beginfill
black
circle
20
1
black
endfill
penup
goto
120
0
1
black
pendown
beginfill
black
circle
20
1
black
endfill
penup
goto
150
40
```

```
1
black
pendown
beginfill
yellow
goto
-30
40
1
black
goto
-30
70
1
black
goto
60
70
1
black
goto
60
100
1
black
goto
90
100
1
black
goto
115
70
1
black
goto
150
70
1
black
goto
150
40
1
black
endfill
```

To read a file as shown in Sect. 1.10.1 we write our loop and a half to read the first line of each record and then check that line (i.e. the graphics command) so we know how many more lines to read. The code in Sect. 1.10.2 uses a while loop to read these variable length records.

1.10.2 Reading and Processing Multi-line Records

```
import turtle

def main():
    filename = input("Please enter drawing filename: ")

    t = turtle.Turtle()
    screen = t.getscreen()

    file = open(filename, "r")

    # Here we have the half a loop to get things started. Reading our first
    # graphics command here lets us determine if the file is empty or not.
    command = file.readline().strip()

    # If the command is empty, then there are no more commands left in the file.
    while command != "":

        # Now we must read the rest of the record and then process it. Because
        # records are variable length, we'll use an if-elif to determine which
        # type of record it is and then we'll read and process the record.

        if command == "goto":
            x = float(file.readline())
            y = float(file.readline())
            width = float(file.readline())
            color = file.readline().strip()
            t.width(width)
            t.pencolor(color)
            t.goto(x,y)
        elif command == "circle":
            radius = float(file.readline())
            width = float(file.readline())
            color = file.readline().strip()
            t.width(width)
            t.pencolor(color)
            t.circle(radius)
        elif command == "beginfill":
            color = file.readline().strip()
            t.fillcolor(color)
            t.begin_fill()
        elif command == "endfill":
            t.end_fill()
        elif command == "penup":
            t.penup()
        elif command == "pendown":
            t.pendown()
        else:
            print("Unknown command found in file:",command)

        # This is still inside the while loop. We must (attempt to) read
        # the next command from the file. If the read succeeds, then command
        # will not be the empty string and the loop will be repeated. If
        # command is empty it is because there were no more commands in the
        # file and the while loop will terminate.
        command = file.readline().strip()


```

```
58      # close the file
59      file.close()
60
61      t.ht()
62      screen.exitonclick()
63      print("Program Execution Completed.")
64
65  if __name__ == "__main__":
66      main()
```

When reading a file with multi-line records, a while loop is needed. Notice that on line 13 the first line of the first record is read prior to the while loop. For the body of the while loop to execute, the condition must be tested prior to executing the loop. Reading a line prior to the while loop is necessary so we can check to see if the file is empty or not. The first line of every other record is read at the end of the while loop on line 55. This is the loop and a half pattern. The first line of the first record is read before the while loop while the first line of every other record is read inside the while loop just before the end. When the condition becomes false, the while loop terminates.

The abstract pattern for reading multi-line records from a file is shown in Sect. 1.10.3. There are certainly other forms of this pattern that can be used, but memorizing this pattern is worth-while since the pattern will work using pretty much any programming language.

1.10.3 Pattern for Reading Multi-line Records from a File

```
1   # First the file must be opened
2   file = open(filename, "r")
3
4   # Read the first line of the first record in the file. Of course, firstLine should be
5   # called something that makes sense in your program.
6   firstLine = file.readline().strip()
7
8   while firstLine != "":
9       # Read the rest of the record
10      secondLine = file.readline().strip()
11      thirdLine = file.readline().strip()
12      # ...
13
14      # Then process the record. This will be determined by the program you are
15      # writing.
16      print(firstLine, secondLine, thirdLine)
17
18      # Finally, finish the loop by reading the first line of the next record to
19      # set up for the next iteration of the loop.
20      firstLine = file.readline().strip()
21
22  # It's a good idea to close the file, but it will be automatically closed when your
23  # program terminates.
24  file.close()
```

1.11 A Container Class

To further enhance our drawing program we will first create a data structure to hold all of our drawing commands. This is our first example of defining our own class in this text so we'll go slow and provide a lot of detail about what is happening and why. To begin let's figure out what we want to do with this container class.

Our program will begin by creating an empty container. To do this, we'll write a line like this.

```
graphicsCommands = PyList()
```

Then, we will want to add graphics commands to our list using an append method like this.

```
command = GotoCommand(x, y, width, color)
graphicsCommands.append(command)
```

We would also like to be able to iterate over the commands in our list.

```
for command in graphicsCommands:
    # draw each command on the screen using the turtle called t.
    command.draw(t)
```

At this point, our container class looks a lot like a list. We are defining our own list class to illustrate a first data structure and to motivate discussion of how lists can be implemented efficiently in this and the next chapter.

1.12 Polymorphism

One important concept in Object-Oriented Programming is called polymorphism. The word *polymorphic* literally means *many forms*. As this concept is applied to computer programming, the idea is that there can be many ways that a particular behavior might be implemented. In relationship to our PyList container class that we are building, the idea is that each type of graphics command will know how to draw itself correctly. For instance, one type of graphics command is the *GoToCommand*. When a *GoToCommand* is drawn it draws a line on the screen from the current point to some new (*x,y*) coordinate. But, when a *CircleCommand* is drawn, it draws a circle on the screen with a particular radius. This *polymorphic* behavior can be defined by creating a class and draw method for each different type of behavior. The code in Sect. 1.12.1 is a collection of classes that define the polymorphic behavior of the different graphics *draw* methods. There is one class for each drawing command that will be processed by the program.

1.12.1 Graphics Command Classes

```
# Each of the command classes below hold information for one of the
# types of commands found in a graphics file. For each command there must
# be a draw method that is given a turtle and uses the turtle to draw
# the object. By having a draw method for each class, we can
# polymorphically call the right draw method when traversing a sequence of
# these commands. Polymorphism occurs when the "right" draw method gets
# called without having to know which graphics command it is being called on.
class GoToCommand:
    # Here the constructor is defined with default values for width and color.
    # This means we can construct a GoToCommand objects as GoToCommand(10,20),
    # or GoToCommand(10,20,5), or GoToCommand(10,20,5,"yellow").
    def __init__(self,x,y,width=1,color="black"):
        self.x = x
        self.y = y
        self.color = color
        self.width = width

    def draw(self,turtle):
        turtle.width(self.width)
        turtle.pencolor(self.color)
        turtle.goto(self.x,self.y)

class CircleCommand:
    def __init__(self,radius, width=1,color="black"):
        self.radius = radius
        self.width = width
        self.color = color

    def draw(self,turtle):
        turtle.width(self.width)
        turtle.pencolor(self.color)
        turtle.circle(self.radius)

class BeginFillCommand:
    def __init__(self,color):
        self.color = color

    def draw(self,turtle):
        turtle.fillcolor(self.color)
        turtle.begin_fill()

class EndFillCommand:
    def __init__(self):
        # pass is a statement placeholder and does nothing. We have nothing
        # to initialize in this class because all we want is the polymorphic
        # behavior of the draw method.
        pass

    def draw(self,turtle):
        turtle.end_fill()

class PenUpCommand:
    def __init__(self):
        pass

    def draw(self,turtle):
        turtle.penup()
```

```
58
59 class PenDownCommand:
60     def __init__(self):
61         pass
62
63     def draw(self,turtle):
64         turtle.pendown()
```

1.13 The Accumulator Pattern

To use the different command classes that we have just defined, our program will read the variable length records from the file as it did before using the *loop and a half* pattern that we have already seen. Patterns of programming, sometimes called *idioms*, are important in Computer Science. Once we have learned an idiom we can apply it over and over in our programs. This is useful to us because as we solve problems its nice to say, "Oh, yes, I can solve this problem using that idiom". Having idioms at our fingertips frees our minds to deal with the tougher problems we encounter while programming.

One important pattern in programming is the *Accumulator Pattern*. This pattern is used in nearly every program we write. When using this pattern you initialize an accumulator before a loop and then inside the loop you add to the accumulator. For instance, the code in Sect. 1.13.1 uses the accumulator pattern to construct the list of squares from 1 to 10.

1.13.1 List of Squares

```
1 # initialize the accumulator, in this case a list
2 accumulator = []
3
4 # write some kind of for loop or while loop
5 for i in range(1,11):
6     # add to the accumulator, in this case add to the list
7     accumulator = accumulator + [i ** 2]
```

To complete our graphics program, we'll use the loop and a half pattern to read the records from a file and the accumulator pattern to add a command object to our PyList container for each record we find in the file. The code is given in Sect. 1.13.2.

1.13.2 A Graphics Program

```
1 import turtle
2
3 # Command classes would be inserted here but are left out because they
4 # were defined earlier in the chapter.
5
```

```
# This is our PyList class. It holds a list of our graphics
# commands.

class PyList:
    def __init__(self):
        self.items = []

    def append(self,item):
        self.items = self.items + [item]

    # if we want to iterate over this sequence, we define the special method
    # called __iter__(self). Without this we'll get "builtins.TypeError:
    # 'PyList' object is not iterable" if we try to write
    # for cmd in seq:
    # where seq is one of these sequences. The yield below will yield an
    # element of the sequence and will suspend the execution of the for
    # loop in the method below until the next element is needed. The ability
    # to yield each element of the sequence as needed is called "lazy" evaluation
    # and is very powerful. It means that we only need to provide access to as
    # many of elements of the sequence as are necessary and no more.
    def __iter__(self):
        for c in self.items:
            yield c

def main():
    filename = input("Please enter drawing filename: ")

    t = turtle.Turtle()
    screen = t.getscreen()
    file = open(filename, "r")

    # Create a PyList to hold the graphics commands that are
    # read from the file.
    graphicsCommands = PyList()

    command = file.readline().strip()

    while command != "":

        # Now we must read the rest of the record and then process it. Because
        # records are variable length, we'll use an if-elif to determine which
        # type of record it is and then we'll read and process the record.
        # In this program, processing the record means creating a command object
        # using one of the classes above and then adding that object to our
        # graphicsCommands PyList object.

        if command == "goto":
            x = float(file.readline())
            y = float(file.readline())
            width = float(file.readline())
            color = file.readline().strip()
            cmd = GoToCommand(x,y,width,color)

        elif command == "circle":
            radius = float(file.readline())
            width = float(file.readline())
            color = file.readline().strip()
            cmd = CircleCommand(radius,width,color)

        elif command == "beginfill":
            color = file.readline().strip()
```

```
            cmd = BeginFillCommand(color)

        elif command == "endfill":
            cmd = EndFillCommand()

        elif command == "penup":
            cmd = PenUpCommand()

        elif command == "pendown":
            cmd = PenDownCommand()
        else:
            # raising an exception will terminate the program immediately
            # which is what we want to happen if we encounter an unknown
            # command. The RuntimeError exception is a common exception
            # to raise. The string will be printed when the exception is
            # printed.
            raise RuntimeError("Unknown Command: " + command)

        # Finish processing the record by adding the command to the sequence.
        graphicsCommands.append(cmd)

        # Read one more line to set up for the next time through the loop.
        command = file.readline().strip()

    # This code iterates through the commands to do the drawing and
    # demonstrates the use of the __iter(self)__ method in the
    # PyList class above.
    for cmd in graphicsCommands:
        cmd.draw(t)

    file.close()
    t.ht()
    screen.exitonclick()
    print("Program Execution Completed.")

if __name__ == "__main__":
    main()
```

1.14 Implementing a GUI with Tkinter

The word GUI means Graphical User Interface. Implementing a Graphical User Interface in Python is very easy using a module called Tkinter. The Tcl/Tk language and toolkit was designed as a cross-platform method of creating GUI interfaces. Python provides an interface to this toolkit via the Tkinter module.

A GUI is an event-driven program. This means that you write your code to respond to events that occur in the program. The events occur as a result of mouse clicks, dragging the mouse, button presses, and menu items being selected.

To build a GUI you place widgets in a window. Widgets are any element of a GUI like labels, buttons, entry boxes, and sometimes invisible widgets called frames. A frame is a widget that can hold other widgets. The drawing application you see in Fig. 1.6 is one such GUI built with Tkinter. In this section we'll develop this drawing application so you learn how to create your own GUI applications using Tkinter and to improve or refresh your Python programming skills.

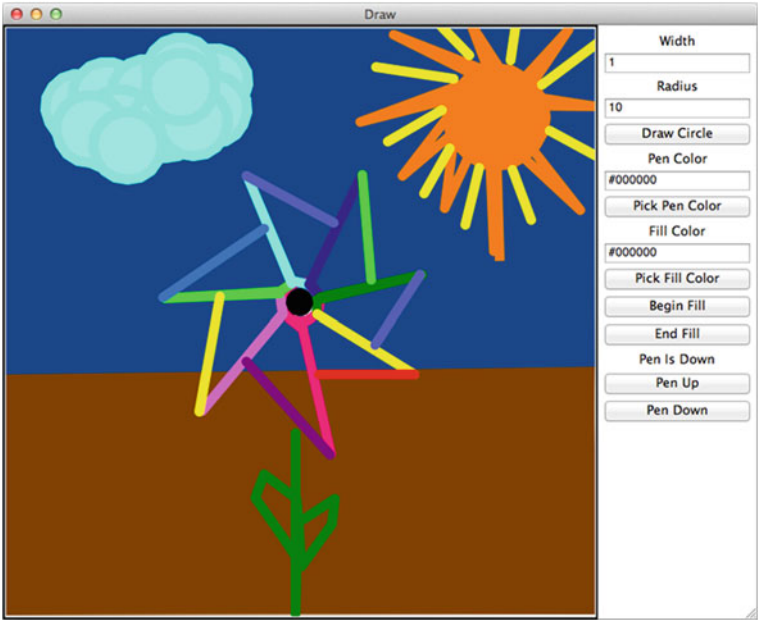

Fig. 1.6 The Draw Program

To construct a GUI you need to create a window. It is really very simple to do this using Tkinter.

```
root = tkinter.Tk()
```

This creates an empty window on the screen, but of course does not put anything in it. We need to place widgets in it so it looks like the window in Fig. 1.6 (without the nice picture that Denise drew for us; thanks Denise!). We also need to create event handlers to handle events in the drawing application.

Putting widgets in a window is called *layout*. Laying out a window relies on a layout manager of some sort. Windowing toolkits support some kind of layout. In Tkinter you either *pack*, *grid*, or *place* widgets within a window. When you *pack* widgets it's like packing a suitcase and each widget is stacked either beside or below the previous widget packed in the GUI. Packing widgets will give you the desired layout in most situations, but at times a *grid* may be useful for laying out a window. The *place* layout manager lets you place widgets at a particular location within a window. We'll use the *pack* layout manager to layout our drawing application.

When packing widgets, to get the proper layout, sometimes you need to create a Frame widget. Frame widgets hold other widgets. In Fig. 1.7 two frame widgets have been created. The DrawingApplication frame is the size of the whole window and holds just two widgets that are placed side by side within it: the canvas and the sideBar frame. A canvas is a widget on which a turtle can draw. The sideBar widget holds all the buttons, entry boxes, and labels.

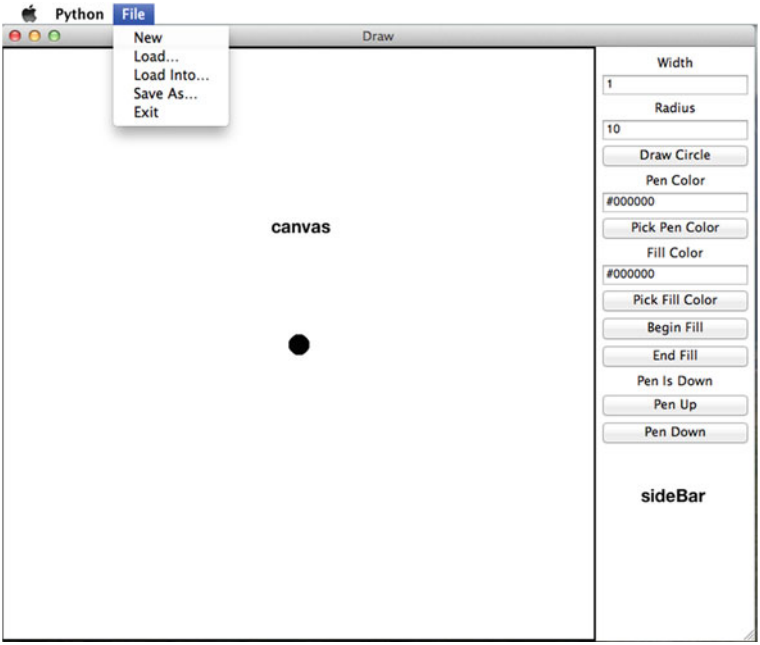

Fig. 1.7 The Draw Program Layout

The DrawingApplication frame *inherits* from Frame. When programming in an object-oriented language, sometimes you want to implement a class, but it is almost like another class. In this case, the *DrawingApplication* is a *Frame*. This means there are two parts to DrawingApplication objects, the Frame part of the DrawingApplication and the rest of it, which in this case is the *PyList* sequence of graphics commands. Our frame will keep track of the graphics commands that are used to draw the picture on the canvas. Portions of the code appear in Sect. 1.14.1. The code in Sect. 1.14.1 shows you all the widgets that are created and how they are packed within the window.

The *canvas* and the *sideBar* widgets are added side by side to the Drawing Application frame. Then all the entry, label, and button widgets are added to the *sideBar* frame.

In addition, there is a menu with the *Draw* application. The menu is another widget that is added to the window (called *self.master* in the code in Sect. 1.14.1). The *fileMenu* is what appears on the menu bar. The menu items "New", "Load...", "Load Into...", "Save As...", and "Exit" are all added to this menu. Each menu item is linked to an event handler that is executed when it is selected.

When *theTurtle* object is created in Sect. 1.14.1, it is created as a *RawTurtle*. A *RawTurtle* is just like a turtle except that a *RawTurtle* can be provided a canvas to draw on. A *Turtle* object creates its own canvas when the first turtle is created. Since we already have a canvas for the turtle, we create a *RawTurtle* object.

In addition to the event handlers for the widgets, there are three other event handlers. The *onclick* event occurs when you click the mouse button on the canvas. The *ondrag* event handler occurs when the turtle is dragged around the canvas. Finally, the *undoHandler* is called when the *u* key is pressed on the keyboard.

1.14.1 A GUI Drawing Application

```
# This class defines the drawing application. The following line says that
# the DrawingApplication class inherits from the Frame class. This means
# that a DrawingApplication is like a Frame object except for the code
# written here which redefines/extends the behavior of a Frame.
class DrawingApplication(tkinter.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.buildWindow()
        self.graphicsCommands = PyList()

    # This method is called to create all the widgets, place them in the GUI,
    # and define the event handlers for the application.
    def buildWindow(self):

        # The master is the root window. The title is set as below.
        self.master.title("Draw")

        # Here is how to create a menu bar. The tearoff=0 means that menus
        # can't be separated from the window which is a feature of tkinter.
        bar = tkinter.Menu(self.master)
        fileMenu = tkinter.Menu(bar,tearoff=0)

        # This code is called by the "New" menu item below when it is selected.
        # The same applies for loadFile, addToFile, and saveFile below. The
        # "Exit" menu item below calls quit on the "master" or root window.
        def newWindow():
            # This sets up the turtle to be ready for a new picture to be
            # drawn. It also sets the sequence back to empty. It is necessary
            # for the graphicsCommands sequence to be in the object (i.e.
            # self.graphicsCommands) because otherwise the statement:
            # graphicsCommands = PyList()
            # would make this variable a local variable in the newWindow
            # method. If it were local, it would not be set anymore once the
            # newWindow method returned.
            theTurtle.clear()
            theTurtle.penup()
            theTurtle.goto(0,0)
            theTurtle.pendown()
            screen.update()
            screen.listen()
            self.graphicsCommands = PyList()

        fileMenu.add_command(label="New",command=newWindow)

        # The parse function adds the contents of an XML file to the sequence.
        def parse(filename):
            xmldoc = xml.dom.minidom.parse(filename)

            graphicsCommandsElement = xmldoc.getElementsByTagName("GraphicsCommands")[0]

            graphicsCommands = graphicsCommandsElement.getElementsByTagName("Command")

            for commandElement in graphicsCommands:
                print(type(commandElement))
                command = commandElement.firstChild.data.strip()
                attr = commandElement.attributes
                if command == "GoTo":
```

```
                    x = float(attr["x"].value)
                    y = float(attr["y"].value)
                    width = float(attr["width"].value)
                    color = attr["color"].value.strip()
                    cmd = GoToCommand(x,y,width,color)

                elif command == "Circle":
                    radius = float(attr["radius"].value)
                    width = float(attr["width"].value)
                    color = attr["color"].value.strip()
                    cmd = CircleCommand(radius,width,color)

                elif command == "BeginFill":
                    color = attr["color"].value.strip()
                    cmd = BeginFillCommand(color)

                elif command == "EndFill":
                    cmd = EndFillCommand()

                elif command == "PenUp":
                    cmd = PenUpCommand()

                elif command == "PenDown":
                    cmd = PenDownCommand()
                else:
                    raise RuntimeError("Unknown Command: " + command)

                self.graphicsCommands.append(cmd)

        def loadFile():

            filename = tkinter.filedialog.askopenfilename(title="Select a Graphics File")

            newWindow()

            # This re-initializes the sequence for the new picture.
            self.graphicsCommands = PyList()

            # calling parse will read the graphics commands from the file.
            parse(filename)

            for cmd in self.graphicsCommands:
                cmd.draw(theTurtle)

            # This line is necessary to update the window after the picture is drawn.
            screen.update()


        fileMenu.add_command(label="Load...",command=loadFile)

        def addToFile():
            filename = tkinter.filedialog.askopenfilename(title="Select a Graphics File")

            theTurtle.penup()
            theTurtle.goto(0,0)
            theTurtle.pendown()
            theTurtle.pencolor("#000000")
            theTurtle.fillcolor("#000000")
            cmd = PenUpCommand()
            self.graphicsCommands.append(cmd)
            cmd = GoToCommand(0,0,1,"#000000")
            self.graphicsCommands.append(cmd)
            cmd = PenDownCommand()
            self.graphicsCommands.append(cmd)
            screen.update()
            parse(filename)

            for cmd in self.graphicsCommands:
                cmd.draw(theTurtle)
```

```
128
129             screen.update()
130
131         fileMenu.add_command(label="Load Into...",command=addToFile)
132
133         # The write function writes an XML file to the given filename
134         def write(filename):
135             file = open(filename, "w")
136             file.write('<?xml version="1.0" encoding="UTF-8" standalone="no" ?>\n')
137             file.write('<GraphicsCommands>\n')
138             for cmd in self.graphicsCommands:
139                 file.write('    '+str(cmd)+"\n")
140
141             file.write('</GraphicsCommands>\n')
142
143             file.close()
144
145         def saveFile():
146             filename = tkinter.filedialog.asksaveasfilename(title="Save Picture As...")
147             write(filename)
148
149         fileMenu.add_command(label="Save As...",command=saveFile)
150
151
152         fileMenu.add_command(label="Exit",command=self.master.quit)
153
154         bar.add_cascade(label="File",menu=fileMenu)
155
156         # This tells the root window to display the newly created menu bar.
157         self.master.config(menu=bar)
158
159         # Here several widgets are created. The canvas is the drawing area on
160         # the left side of the window.
161         canvas = tkinter.Canvas(self,width=600,height=600)
162         canvas.pack(side=tkinter.LEFT)
163
164         # By creating a RawTurtle, we can have the turtle draw on this canvas.
165         # Otherwise, a RawTurtle and a Turtle are exactly the same.
166         theTurtle = turtle.RawTurtle(canvas)
167
168         # This makes the shape of the turtle a circle.
169         theTurtle.shape("circle")
170         screen = theTurtle.getscreen()
171
172         # This causes the application to not update the screen unless
173         # screen.update() is called. This is necessary for the ondrag event
174         # handler below. Without it, the program bombs after dragging the
175         # turtle around for a while.
176         screen.tracer(0)
177
178         # This is the area on the right side of the window where all the
179         # buttons, labels, and entry boxes are located. The pad creates some empty
180         # space around the side. The side puts the sideBar on the right side of the
181         # this frame. The fill tells it to fill in all space available on the right
182         # side.
183         sideBar = tkinter.Frame(self,padx=5,pady=5)
184         sideBar.pack(side=tkinter.RIGHT, fill=tkinter.BOTH)
185
186         # This is a label widget. Packing it puts it at the top of the sidebar.
187         pointLabel = tkinter.Label(sideBar,text="Width")
188         pointLabel.pack()
189
190         # This entry widget allows the user to pick a width for their lines.
191         # With the widthSize variable below you can write widthSize.get() to get
192         # the contents of the entry widget and widthSize.set(val) to set the value
193         # of the entry widget to val. Initially the widthSize is set to 1. str(1) is
194         # needed because the entry widget must be given a string.
195         widthSize = tkinter.StringVar()
196         widthEntry = tkinter.Entry(sideBar,textvariable=widthSize)
```

```
197         widthEntry.pack()
198         widthSize.set(str(1))
199
200         radiusLabel = tkinter.Label(sideBar,text="Radius")
201         radiusLabel.pack()
202         radiusSize = tkinter.StringVar()
203         radiusEntry = tkinter.Entry(sideBar,textvariable=radiusSize)
204         radiusSize.set(str(10))
205         radiusEntry.pack()
206
207         # A button widget calls an event handler when it is pressed. The circleHandler
208         # function below is the event handler when the Draw Circle button is pressed.
209         def circleHandler():
210             # When drawing, a command is created and then the command is drawn by calling
211             # the draw method. Adding the command to the graphicsCommands sequence means the
212             # application will remember the picture.
213             cmd = CircleCommand(float(radiusSize.get()), float(widthSize.get()), penColor.get())
214             cmd.draw(theTurtle)
215             self.graphicsCommands.append(cmd)
216
217             # These two lines are needed to update the screen and to put the focus back
218             # in the drawing canvas. This is necessary because when pressing "u" to undo,
219             # the screen must have focus to receive the key press.
220             screen.update()
221             screen.listen()
222
223         # This creates the button widget in the sideBar. The fill=tkinter.BOTH causes the button
224         # to expand to fill the entire width of the sideBar.
225         circleButton = tkinter.Button(sideBar, text = "Draw Circle", command=circleHandler)
226         circleButton.pack(fill=tkinter.BOTH)
227
228         # The color mode 255 below allows colors to be specified in RGB form (i.e. Red/
229         # Green/Blue). The mode allows the Red value to be set by a two digit hexadecimal
230         # number ranging from 00-FF. The same applies for Blue and Green values. The
231         # color choosers below return a string representing the selected color and a slice
232         # is taken to extract the #RRGGBB hexadecimal string that the color choosers return.
233         screen.colormode(255)
234         penLabel = tkinter.Label(sideBar,text="Pen Color")
235         penLabel.pack()
236         penColor = tkinter.StringVar()
237         penEntry = tkinter.Entry(sideBar,textvariable=penColor)
238         penEntry.pack()
239         # This is the color black.
240         penColor.set("#000000")
241
242         def getPenColor():
243             color = tkinter.colorchooser.askcolor()
244             if color != None:
245                 penColor.set(str(color)[-9:-2])
246
247         penColorButton = tkinter.Button(sideBar, text = "Pick Pen Color", command=getPenColor)
248         penColorButton.pack(fill=tkinter.BOTH)
249
250         fillLabel = tkinter.Label(sideBar,text="Fill Color")
251         fillLabel.pack()
252         fillColor = tkinter.StringVar()
253         fillEntry = tkinter.Entry(sideBar,textvariable=fillColor)
254         fillEntry.pack()
255         fillColor.set("#000000")
256
257         def getFillColor():
258             color = tkinter.colorchooser.askcolor()
259             if color != None:
260                 fillColor.set(str(color)[-9:-2])
261
262         fillColorButton = \
263             tkinter.Button(sideBar, text = "Pick Fill Color", command=getFillColor)
264         fillColorButton.pack(fill=tkinter.BOTH)
265
```