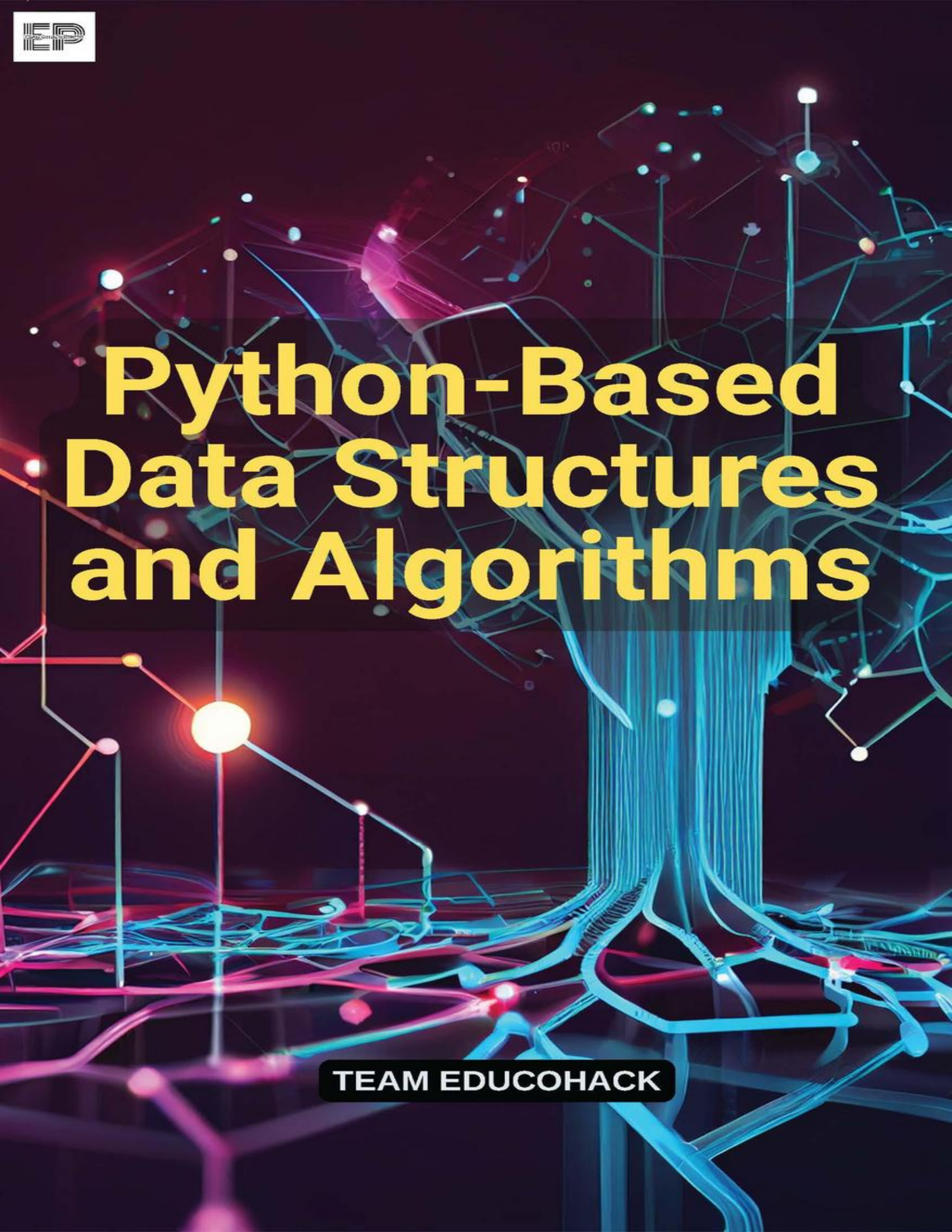
Python-Based
Data Structures
and Algorithms
TEAM EDUCOHACK

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 com- puter. 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 Environ- ment (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

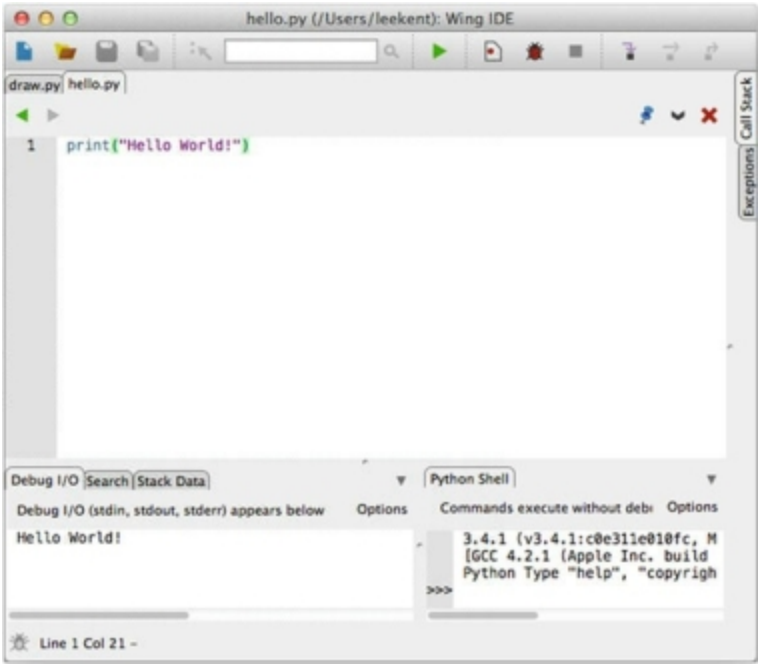

Fig. 1.1 The Wing IDE

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.

1 Python Programming 101 3

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

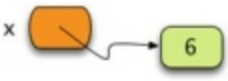

Fig. 1.2 A Reference and Object

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 deci- mal 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 appear- ing 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.

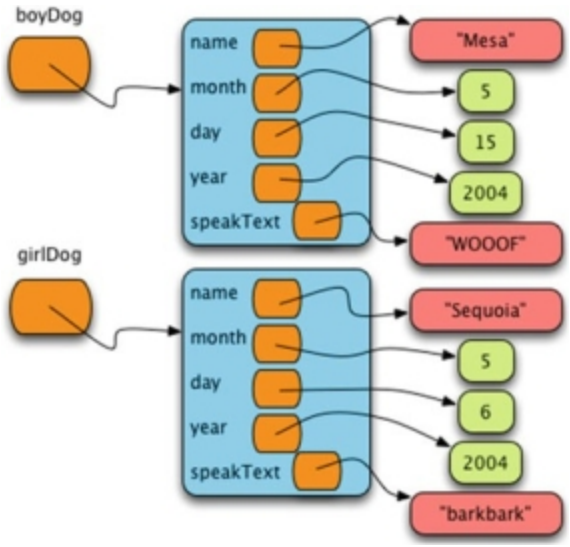

Fig. 1.3 A Couple of Dog Objects

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

```
1 class Dog:
2 # This is the constructor for the class. It is called whenever a Dog
3 # object is created. The reference called "self" is created by Python
4 # and made to point to the space for the newly created object. Python
5 # does this automatically for us but we have to have "self" as the first
6 # parameter to the  init  method (i.e. the constructor).
7 def  init (self, name, month, day, year, speakText):
8 self.name = name
9 self.month = month
10 self.day = day
11 self.year = year
12 self.speakText = speakText
13
14 # This is an accessor method that returns the speakText stored in the
15 # object. Notice that "self" is a parameter. Every method has "self" as its
16 # first parameter. The "self" parameter is a reference to the current
17 # object. The current object appears on the left hand side of the dot (i.e.
18 # the .) when the method is called.
19 def speak(self):
20 return self.speakText
21
22 # Here is an accessor method to get the name
```

```
23 def getName(self):
```

```
24 return self.name
25
26 # This is another accessor method that uses the birthday information to
27 # return a string representing the date.
28 def birthDate(self):
29 return str(self.month) + "/" + str(self.day) + "/" + str(self.year)
30
31 # This is a mutator method that changes the speakText of the Dog object.
32 def changeBark(self,bark):
33 self.speakText = bark
```

1.5 OPERATOR OVERLOADING

Python provides operator overloading, which is a nice feature of programming lan- guages 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

1 **class Dog**:

2 *# This is the constructor for the class. It is called whenever a Dog*

3 *# object is created. The reference called "self" is created by Python*

4 *# and made to point to the space for the newly created object. Python*

5 *# does this automatically for us but we have to have "self" as the first*

6 *# parameter to the init method (i.e. the constructor).*

1. Operator Overloading 9

---

7 **def** init (self, name, month, day, year, speakText):

8 self.name = name

9 self.month = month

10 self.day = day

11 self.year = year

12 self.speakText = speakText

13

14 *# This is an accessor method that returns the speakText stored in the*

15 *# object. Notice that "self" is a parameter. Every method has "self" as its*

16 *# first parameter. The "self" parameter is a reference to the current*

17 *# object. The current object appears on the left hand side of the dot (i.e.*

18 *# the .) when the method is called.*

19 **def** speak(self):

20 **return** self.speakText

21

22 *# Here is an accessor method to get the name*

23 **def** getName(self):

24 **return** self.name

25

26 *# This is another accessor method that uses the birthday information to*

27 *# return a string representing the date.*

```
28 def birthDate(self):
29 return str(self.month) + "/" + str(self.day) + "/" + str(self.year)
30
31 # This is a mutator method that changes the speakText of the Dog object.
32 def changeBark(self,bark):
33 self.speakText = bark
34
35 # When creating the new puppy we don't know it's birthday. Pick the
36 # first dog's birthday plus one year. The speakText will be the
37 # concatenation of both dog's text. The dog on the left side of the +
38 # operator is the object referenced by the "self" parameter. The
39 # "otherDog" parameter is the dog on the right side of the + operator.
40 def  add (self,otherDog):
41 return Dog("Puppy of " + self.name + " and " + otherDog.name, \
42 self.month, self.day, self.year + 1, \
43 self.speakText + otherDog.speakText)
 44
 45 def main():
 46      boyDog = Dog("Mesa", 5, 15, 2004, "WOOOOF")
 47      girlDog = Dog("Sequoia", 5, 6, 2004, "barkbark")
 48      print(boyDog.speak())
 49      print(girlDog.speak())
 50      print(boyDog.birthDate())
 51      print(girlDog.birthDate())
```

```
52      boyDog.changeBark("woofywoofy")
53      print(boyDog.speak())
54      puppy = boyDog + girlDog
55      print(puppy.speak())
56      print(puppy.getName())
57      print(puppy.birthDate())
58
59 if  name  == " main ":
60 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*.
	x * y	Returns the product of *x* and *y*.

__mul__ (self,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 interac- tion.
__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 oper- ators 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 "*tur- tle*.", 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.

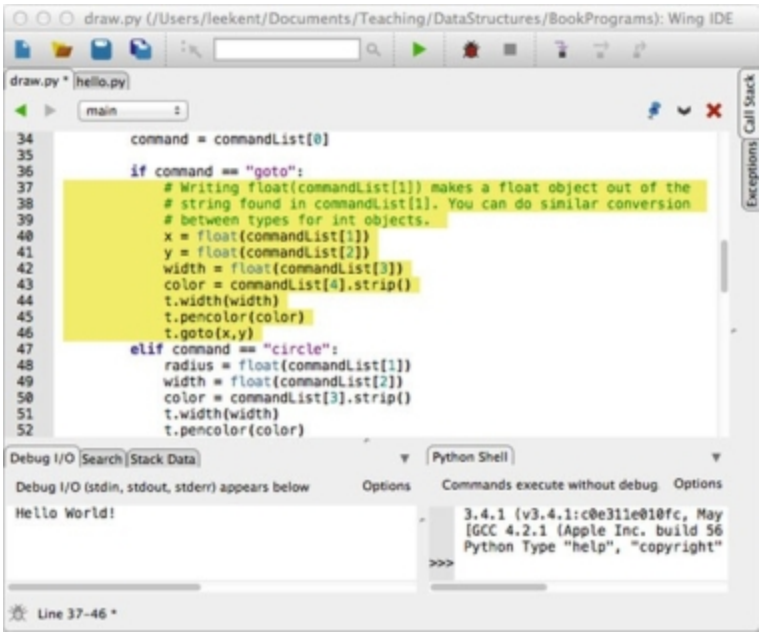

Fig. 1.5 Adjusting Indentation in Wing IDE 101

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

```
1 # Imports at the top.
2 import turtle
3
4 # other function definitions followed by the main function definition
5 def main():
6 # The main code of the program goes here
7 t = turtle.Turtle()
8
9 # this code calls the main function to get everything started. The condition in this
10 # if statement evaluates to True when the module is executed by the interpreter, but
11 # not when it is imported into another module.
12 if name == " main ":
13 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 func- tion 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

1 beginfill, black

2 circle, 20, 1, black

3 endfill

4 penup

5 goto, 120, 0, 1, black

6 pendown

7 beginfill, black

8 circle, 20, 1, black

9 endfill

10 penup

11 goto, 150, 40, 1, black

12 pendown

13 beginfill, yellow

14 goto, -30, 40, 1, black

15 goto, -30, 70, 1, black

16 goto, 60, 70, 1, black

17 goto, 60, 100, 1, black

18 goto, 90, 100, 1, black

19 goto, 115, 70, 1, black

20 goto, 150, 70, 1, black

```
21 goto, 150, 40, 1, black

22 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

```
1 # This imports the turtle graphics module.
2 import turtle
3
4 # The main function is where the main code of the program is written.
5 def main():
6 # This line reads a line of input from the user.
7 filename = input("Please enter drawing filename: ")
8
9 # Create a Turtle Graphics window to draw in.
10 t = turtle.Turtle()
11 # The screen is used at the end of the program.
12 screen = t.getscreen()
13
14 # The next line opens the file for "r" or reading. "w" would open it for
15 # writing, and "a" would open the file to append to it (i.e. add to the
16 # end). In this program we are only interested in reading the file.
17 file = open(filename, "r")
18
19 # The following for loop reads the lines of the file, one at a time
20 # and executes the body of the loop once for each line of the file.
21 for line in file:
```

```
22
23 # The strip method strips off the newline character at the end of the line
24 # and any blanks that might be at the beginning or end of the line.
25 text = line.strip()
26
27 # The following line splits the text variable into its pieces.
28 # For instance, if text contained "goto, 10, 20, 1, black" then
29 # commandList will be equal to ["goto", "10", "20", "1", "black"] after
30 # splitting text.
31 commandList = text.split(",")
32
33 # get the drawing command
34 command = commandList[0]
35
36 if command == "goto":
37 # Writing float(commandList[1]) makes a float object out of the
38 # string found in commandList[1]. You can do similar conversion
39 # between types for int objects.
40 x = float(commandList[1])
41 y = float(commandList[2])
42 width = float(commandList[3])
43 color = commandList[4].strip()
44 t.width(width)
45 t.pencolor(color)
```

1. Reading from a File 15

```
46 t.goto(x,y)
47 elif command == "circle":
48 radius = float(commandList[1])
49 width = float(commandList[2])
50 color = commandList[3].strip()
51 t.width(width)
52 t.pencolor(color)
53 t.circle(radius)
54 elif command == "beginfill":
55 color = commandList[1].strip()
56 t.fillcolor(color)
57 t.begin_fill()
58 elif command == "endfill":
59 t.end_fill()
60 elif command == "penup":
61 t.penup()
62 elif command == "pendown":
63 t.pendown()
64 else:
65 print("Unknown command found in file:",command)
66
```

```
67 #close the file
68 file.close()
69
70 #hide the turtle that we used to draw the picture.
71 t.ht()
72
73 # This causes the program to hold the turtle graphics window open
74 # until the mouse is clicked.
75 screen.exitonclick()
76 print("Program Execution Completed.")
77
78
79 # This code calls the main function to get everything started.
80 if  name  == " main ":
81 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

```
1 # First the file must be opened.
2 file = open(filename,"r")
3
4 # The body of the for loop is executed once for each line in the file.
5 for line in file:
6 # Process each record of the file. Each record must be exactly one line of the
7 # input file. What processing a record means will be determined by the
8 # program you are writing.
9 print(line)
10
11 # Closing the file is always a good idea, but it will be closed when your program
```

12 *# terminates if you do not close it explicitly.*

13 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

1 beginfill

2 black

3 circle

4 20

5 1

6 black

7 endfill

8 penup

9 goto

10 120

11 0

12 1

13 black

14 pendown

15 beginfill

16 black

17 circle

18 20

19 1

20 black

21 endfill

22 penup

23 goto

24 150

25 40

26 1

27 black

28 pendown

29 beginfill

30 yellow

31 goto

32 -30

33 40

34 1

35 black

36 goto

37 -30

38 70

39 1

40 black

41 goto

42 60

43 70

44 1

45 black

46 goto

47 60

48 100

49 1

50 black

51 goto

52 90

53 100

54 1

55 black

56 goto

57 115

58 70

59 1

60 black

61 goto

62 150

63 70

64 1

65 black

66 goto

67 150

68 40

69 1

70 black

71 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

```
1 import turtle
2
3 def main():
4 filename = input("Please enter drawing filename: ")
5
6 t = turtle.Turtle()
7 screen = t.getscreen()
8
9 file = open(filename, "r")
10
11 # Here we have the half a loop to get things started. Reading our first
12 # graphics command here lets us determine if the file is empty or not.
13 command = file.readline().strip()
14
15 # If the command is empty, then there are no more commands left in the file.
16 while command != "":
17
18 # Now we must read the rest of the record and then process it. Because
19 # records are variable length, we'll use an if-elif to determine which
20 # type of record it is and then we'll read and process the record.
21
```

```
22 if command == "goto":
23 x = float(file.readline())
24 y = float(file.readline())
25 width = float(file.readline())
26 color = file.readline().strip()
27 t.width(width)
28 t.pencolor(color)
29 t.goto(x,y)
30 elif command == "circle":
31 radius = float(file.readline())
32 width = float(file.readline())
33 color = file.readline().strip()
34 t.width(width)
35 t.pencolor(color)
36 t.circle(radius)
37 elif command == "beginfill":
38 color = file.readline().strip()
39 t.fillcolor(color)
40 t.begin_fill()
41 elif command == "endfill":
42 t.end_fill()
43 elif command == "penup":
44 t.penup()
45 elif command == "pendown":
46 t.pendown()
```

```
47 else:
48 print("Unknown command found in file:",command)
49
50 # This is still inside the while loop. We must (attempt to) read
51 # the next command from the file. If the read succeeds, then command
52 # will not be the empty string and the loop will be repeated. If
53 # command is empty it is because there were no more commands in the
54 # file and the while loop will terminate.
55 command = file.readline().strip()
56
57
```

```
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

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

# Read the first line of the first record in the file. Of course, firstLine should be
# called something that makes sense in your program.
firstLine = file.readline().strip()

while firstLine != "":
# Read the rest of the record
secondLine = file.readline().strip()
thirdLine = file.readline().strip()
# ...

# Then process the record. This will be determined by the program you are
# writing.
print(firstLine, secondLine, thirdLine)

# Finally, finish the loop by reading the first line of the next record to
# set up for the next iteration of the loop.
firstLine = file.readline().strip()

```

```
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

```
1 # Each of the command classes below hold information for one of the
2 # types of commands found in a graphics file. For each command there must
3 # be a draw method that is given a turtle and uses the turtle to draw
4 # the object. By having a draw method for each class, we can
5 # polymorphically call the right draw method when traversing a sequence of
6 # these commands. Polymorphism occurs when the "right" draw method gets
7 # called without having to know which graphics command it is being called on.
8 class GoToCommand:
9 # Here the constructor is defined with default values for width and color.
10 # This means we can construct a GoToCommand objects as GoToCommand(10,20),
11 # or GoToCommand(10,20,5), or GoToCommand(10,20,5,"yellow").
 12 def init (self,x,y,width=1,color="black"):
 13     self.x = x
 14     self.y = y
 15     self.color = color
 16     self.width = width
 17
 18 def draw(self,turtle):
 19     turtle.width(self.width)
 20     turtle.pencolor(self.color)
 21     turtle.goto(self.x,self.y)
```

```
22

23 class CircleCommand:

24 def  init (self,radius, width=1,color="black"):

25 self.radius = radius

26 self.width = width

27 self.color = color

28

29 def draw(self,turtle):

30 turtle.width(self.width)

31 turtle.pencolor(self.color)

32 turtle.circle(self.radius)

33

34 class BeginFillCommand:

35 def  init (self,color):

36 self.color = color

37

38 def draw(self,turtle):

39 turtle.fillcolor(self.color)

40 turtle.begin_fill()

41

42 class EndFillCommand:

43 def  init (self):

44 # pass is a statement placeholder and does nothing. We have nothing

45 # to initialize in this class because all we want is the polymorphic

46 # behavior of the draw method.
```

```
47 pass
48
49 def draw(self,turtle):
50 turtle.end_fill()
51
52 class PenUpCommand:
53 def  init (self):
54 pass
55
56 def draw(self,turtle):
57 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
```

```
6 # This is our PyList class. It holds a list of our graphics
7 # commands.
8
9 class PyList:
10 def  init (self):
11 self.items = []
12
13 def append(self,item):
14 self.items = self.items + [item]
15
16 # if we want to iterate over this sequence, we define the special method
17 # called  iter (self). Without this we'll get "builtins.TypeError:
18 # 'PyList' object is not iterable" if we try to write
19 # for cmd in seq:
20 # where seq is one of these sequences. The yield below will yield an
21 # element of the sequence and will suspend the execution of the for
22 # loop in the method below until the next element is needed. The ability
23 # to yield each element of the sequence as needed is called "lazy" evaluation
24 # and is very powerful. It means that we only need to provide access to as
25 # many of elements of the sequence as are necessary and no more.
```

```
26      def  iter (self):
27      for c in self.items:
28      yield c
29
30 def main():
31      filename = input("Please enter drawing filename: ")
32
33      t = turtle.Turtle()
34      screen = t.getscreen()
35      file = open(filename, "r")
36
37      # Create a PyList to hold the graphics commands that are
38      # read from the file.
39      graphicsCommands = PyList()
40
41      command = file.readline().strip()
42
43      while command != "":
44
45 # Now we must read the rest of the record and then process it. Because
46 # records are variable length, we'll use an if-elif to determine which
47 # type of record it is and then we'll read and process the record.
```

```
48 # In this program, processing the record means creating a command object
49 # using one of the classes above and then adding that object to our
50 # graphicsCommands PyList object.
51
52 if command == "goto":
53 x = float(file.readline())
54 y = float(file.readline())
55 width = float(file.readline())
56 color = file.readline().strip()
57 cmd = GoToCommand(x,y,width,color)
58
59 elif command == "circle":
60 radius = float(file.readline())
61 width = float(file.readline())
62 color = file.readline().strip()
63 cmd = CircleCommand(radius,width,color)
64
65 elif command == "beginfill":
66 color = file.readline().strip()
```

```
67 cmd = BeginFillCommand(color)
68
69 elif command == "endfill":
70 cmd = EndFillCommand()
71
72 elif command == "penup":
73 cmd = PenUpCommand()
74
75 elif command == "pendown":
76 cmd = PenDownCommand()
77 else:
78 # raising an exception will terminate the program immediately
79 # which is what we want to happen if we encounter an unknown
80 # command. The RuntimeError exception is a common exception
81 # to raise. The string will be printed when the exception is
82 # printed.
83 raise RuntimeError("Unknown Command: " + command)
84
85 # Finish processing the record by adding the command to the sequence.
86 graphicsCommands.append(cmd)
87
88 # Read one more line to set up for the next time through the loop.
89 command = file.readline().strip()
90
91 # This code iterates through the commands to do the drawing and
```

```
92 # demonstrates the use of the  iter(self)  method in the
93 # PyList class above.
94 for cmd in graphicsCommands:
95 cmd.draw(t)
96
97 file.close()
98 t.ht()
99 screen.exitonclick()
100 print("Program Execution Completed.")
101
102 if  name  == " main ":
103 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.

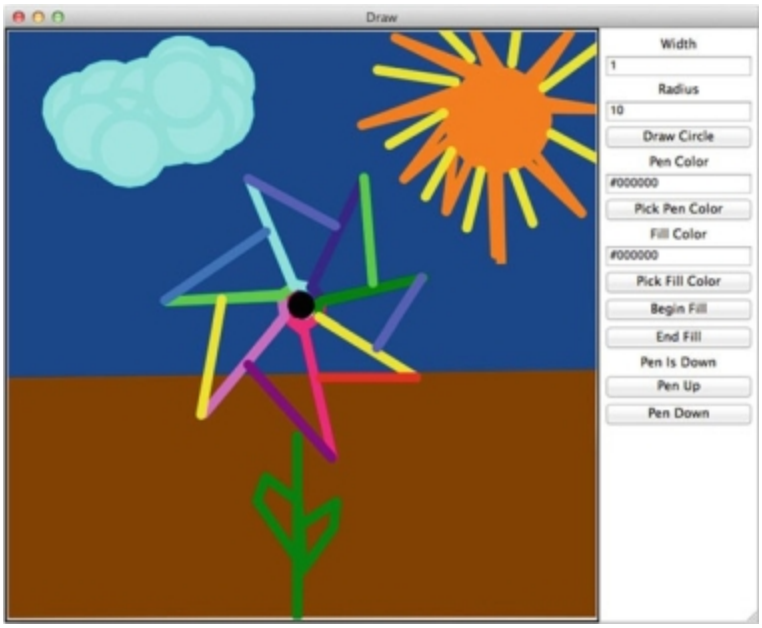

Fig. 1.6 The Draw Program

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.

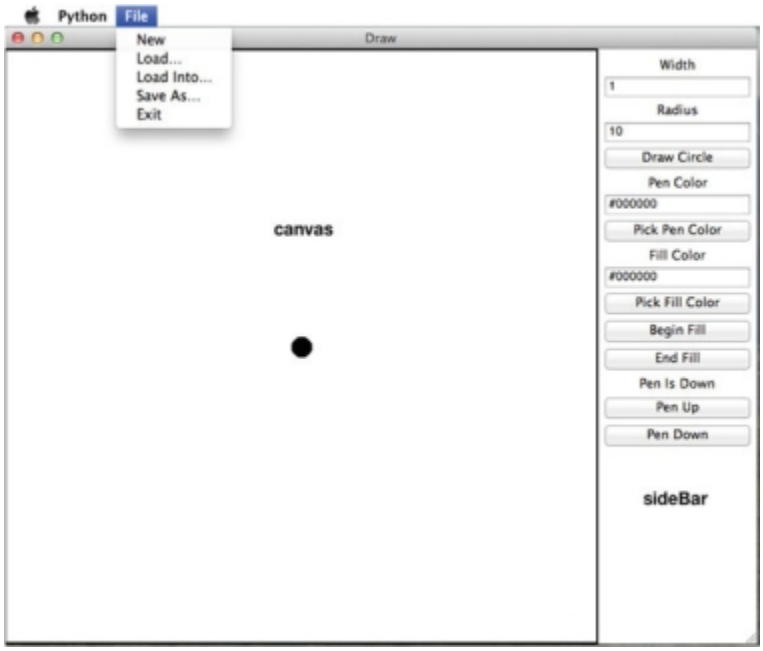

Fig. 1.7 The Draw Program Layout

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 han- dlers. 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()

11
# 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):

15

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

18

# 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)

23

# 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":
```

```
59 x = float(attr["x"].value)
60 y = float(attr["y"].value)
61 width = float(attr["width"].value)
62 color = attr["color"].value.strip()
63 cmd = GoToCommand(x,y,width,color)
64
65 elif command == "Circle":
66 radius = float(attr["radius"].value)
67 width = float(attr["width"].value)
68 color = attr["color"].value.strip()
69 cmd = CircleCommand(radius,width,color)
70
71 elif command == "BeginFill":
72 color = attr["color"].value.strip()
73 cmd = BeginFillCommand(color)
74
75 elif command == "EndFill":
76 cmd = EndFillCommand()
77
78 elif command == "PenUp":
79 cmd = PenUpCommand()
80
81 elif command == "PenDown":
82 cmd = PenDownCommand()
83 else:
```

```
84 raise RuntimeError("Unknown Command: " + command)
85
86 self.graphicsCommands.append(cmd)
87
88 def loadFile():
89
90 filename = tkinter.filedialog.askopenfilename(title="Select a Graphics File")
91
92 newWindow()
93
94 # This re-initializes the sequence for the new picture.
95 self.graphicsCommands = PyList()
96
97 # calling parse will read the graphics commands from the file.
98 parse(filename)
99
100 for cmd in self.graphicsCommands:
101 cmd.draw(theTurtle)
102
103 # This line is necessary to update the window after the picture is drawn.
104 screen.update()
105
106
107 fileMenu.add_command(label="Load...",command=loadFile)
108
```

```
109 def addToFile():
110 filename = tkinter.filedialog.askopenfilename(title="Select a Graphics File")
111
112 theTurtle.penup()
113 theTurtle.goto(0,0)
114 theTurtle.pendown()
115 theTurtle.pencolor("#000000")
116 theTurtle.fillcolor("#000000")
117 cmd = PenUpCommand()
118 self.graphicsCommands.append(cmd)
119 cmd = GoToCommand(0,0,1,"#000000")
120 self.graphicsCommands.append(cmd)
121 cmd = PenDownCommand()
122 self.graphicsCommands.append(cmd)
123 screen.update()
124 parse(filename)
125
126 for cmd in self.graphicsCommands:
127 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

1. *# This tells the root window to display the newly created menu bar.*
2. self.master.config(menu=bar)

158

1. *# Here several widgets are created. The canvas is the drawing area on*
2. *# the left side of the window.*
3. canvas = tkinter.Canvas(self,width=600,height=600)
4. canvas.pack(side=tkinter.LEFT)

163

1. *# By creating a RawTurtle, we can have the turtle draw on this canvas.*
2. *# Otherwise, a RawTurtle and a Turtle are exactly the same.*
3. theTurtle = turtle.RawTurtle(canvas)

167

1. *# This makes the shape of the turtle a circle.*
2. theTurtle.shape("circle")
3. screen = theTurtle.getscreen()

171

1. *# This causes the application to not update the screen unless*
2. *# screen.update() is called. This is necessary for the ondrag event*
3. *# handler below. Without it, the program bombs after dragging the*
4. *# turtle around for a while.*
5. screen.tracer(0)

177

1. *# This is the area on the right side of the window where all the*
2. *# buttons, labels, and entry boxes are located. The pad creates some empty*
3. *# space around the side. The side puts the sideBar on the right side of the*
4. *# this frame. The fill tells it to fill in all space available on the right*
5. *# side.*
6. sideBar = tkinter.Frame(self,padx=5,pady=5)
7. sideBar.pack(side=tkinter.RIGHT, fill=tkinter.BOTH)

185

```
# This is a label widget. Packing it puts it at the top of the sidebar.
pointLabel = tkinter.Label(sideBar,text="Width")
pointLabel.pack()
```

189

```
# This entry widget allows the user to pick a width for their lines.
# With the widthSize variable below you can write widthSize.get() to get
# the contents of the entry widget and widthSize.set(val) to set the value
# of the entry widget to val. Initially the widthSize is set to 1. str(1) is
# needed because the entry widget must be given a string.
widthSize = tkinter.StringVar()
widthEntry = tkinter.Entry(sideBar,textvariable=widthSize)
```

```
1. widthEntry.pack()
2. widthSize.set(str(1))
```

199

```
1. radiusLabel = tkinter.Label(sideBar,text="Radius")
2. radiusLabel.pack()
3. radiusSize = tkinter.StringVar()
4. radiusEntry = tkinter.Entry(sideBar,textvariable=radiusSize)
5. radiusSize.set(str(10))
6. radiusEntry.pack()
```

206

```
1. # A button widget calls an event handler when it is pressed. The circleHandler
2. # function below is the event handler when the Draw Circle button is pressed.
3. def circleHandler():
4. # When drawing, a command is created and then the command is drawn by calling
5. # the draw method. Adding the command to the graphicsCommands sequence means the
6. # application will remember the picture.
7. cmd = CircleCommand(float(radiusSize.get()), float(widthSize.get()), penColor.get())
8. cmd.draw(theTurtle)
9. 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

```
1. # This creates the button widget in the sideBar. The fill=tkinter.BOTH causes the button
2. # to expand to fill the entire width of the sideBar.
3. circleButton = tkinter.Button(sideBar, text = "Draw Circle", command=circleHandler)
4. circleButton.pack(fill=tkinter.BOTH)
```

227

```
1. # The color mode 255 below allows colors to be specified in RGB form (i.e. Red/
2. # Green/Blue). The mode allows the Red value to be set by a two digit hexadecimal
3. # number ranging from 00-FF. The same applies for Blue and Green values. The
4. # color choosers below return a string representing the selected color and a slice
5. # is taken to extract the #RRGGBB hexadecimal string that the color choosers return.
6. screen.colormode(255)
7. penLabel = tkinter.Label(sideBar,text="Pen Color")
8. penLabel.pack()
9. penColor = tkinter.StringVar()
10. penEntry = tkinter.Entry(sideBar,textvariable=penColor)
11. penEntry.pack()
12. # This is the color black.
13. penColor.set("#000000")
```

241

```
1. def getPenColor():
2. color = tkinter.colorchooser.askcolor()
3. if color != None:
4. penColor.set(str(color)[-9:-2])
```

246

```
1. penColorButton = tkinter.Button(sideBar, text = "Pick Pen Color", command=getPenColor)
2. penColorButton.pack(fill=tkinter.BOTH)
```

249

```
1. fillLabel = tkinter.Label(sideBar,text="Fill Color")
2. fillLabel.pack()
3. fillColor = tkinter.StringVar()
4. fillEntry = tkinter.Entry(sideBar,textvariable=fillColor)
5. fillEntry.pack()
6. fillColor.set("#000000")
```

256

```
1. def getFillColor():
2. color = tkinter.colorchooser.askcolor()
3. if color != None:
4. fillColor.set(str(color)[-9:-2])
```

261

```
1. fillColorButton = \
```

2. tkinter.Button(sideBar, text = "Pick Fill Color", command=getFillColor)
3. fillColorButton.pack(fill=tkinter.BOTH)

```
266

267 def beginFillHandler():

268 cmd = BeginFillCommand(fillColor.get())

269 cmd.draw(theTurtle)

270 self.graphicsCommands.append(cmd)

271
```

1. beginFillButton = tkinter.Button(sideBar, text = "Begin Fill", command=beginFillHandler)
2. beginFillButton.pack(fill=tkinter.BOTH)

```
274
```

1. def endFillHandler():
2. cmd = EndFillCommand()
3. cmd.draw(theTurtle)
4. self.graphicsCommands.append(cmd)

```
279
```

1. endFillButton = tkinter.Button(sideBar, text = "End Fill", command=endFillHandler)
2. endFillButton.pack(fill=tkinter.BOTH)

```
282
```

1. penLabel = tkinter.Label(sideBar,text="Pen Is Down")
2. penLabel.pack()

```
285

286 def penUpHandler():

287 cmd = PenUpCommand()
```

```
cmd.draw(theTurtle)

penLabel.configure(text="Pen Is Up")

self.graphicsCommands.append(cmd)

penUpButton = tkinter.Button(sideBar, text = "Pen Up", command=penUpHandler)
penUpButton.pack(fill=tkinter.BOTH)

def penDownHandler():
cmd = PenDownCommand()
cmd.draw(theTurtle)
penLabel.configure(text="Pen Is Down")
self.graphicsCommands.append(cmd)

penDownButton = tkinter.Button(sideBar, text = "Pen Down", command=penDownHandler)
penDownButton.pack(fill=tkinter.BOTH)

# Here is another event handler. This one handles mouse clicks on the screen.
def clickHandler(x,y):
# When a mouse click occurs, get the widthSize entry value and set the width of the
# pen to the widthSize value. The float(widthSize.get()) is needed because
# the width is a float, but the entry widget stores it as a string.
cmd = GoToCommand(x,y,float(widthSize.get()),penColor.get())
cmd.draw(theTurtle)
self.graphicsCommands.append(cmd)
screen.update()
screen.listen()

# Here is how we tie the clickHandler to mouse clicks.
screen.onclick(clickHandler)
```

317

```
def dragHandler(x,y):
cmd = GoToCommand(x,y,float(widthSize.get()),penColor.get())
cmd.draw(theTurtle)
self.graphicsCommands.append(cmd)
screen.update()
screen.listen()
```

324

325 theTurtle.ondrag(dragHandler)

326

```
# the undoHandler undoes the last command by removing it from the
# sequence and then redrawing the entire picture.
def undoHandler():
if len(self.graphicsCommands) > 0:
self.graphicsCommands.removeLast()
theTurtle.clear()
theTurtle.penup()
theTurtle.goto(0,0)
```

```
1. theTurtle.pendown()
2. for cmd in self.graphicsCommands:
3. cmd.draw(theTurtle)
4. screen.update()
5. screen.listen()
```

340

```
1. screen.onkeypress(undoHandler, "u")
2. screen.listen()
```

343

```
1. # The main function in our GUI program is very simple. It creates the
2. # root window. Then it creates the DrawingApplication frame which creates
3. # all the widgets and has the logic for the event handlers. Calling mainloop
4. # on the frames makes it start listening for events. The mainloop function will
5. # return when the application is exited.
6. def main():
7. root = tkinter.Tk()
8. drawingApp = DrawingApplication(root)
```

352

```
1. drawingApp.mainloop()
2. print("Program Execution Completed.")
```

355

```
1. _____ if name == " main ":
2. main()
```

1.15 XML FILES

Reading a standard text file, like the graphics commands file we read using the loop and a half pattern in Sect. 1.13.2, is a common task in computer programs. The only problem is that the program must be written to read the specific format of the input file. If we later wish to change the format of the input file to include, for example, a new option like fill color for a circle, then we are stuck updating the program and updating all the files it once read. The input file format and the program must always be synchronized. This means that all old formatted input files must be converted to the new format or they must be thrown away. That is simply not acceptable to most businesses because data is valuable.

To deal with this problem, computer programmers designed a language for describing data input files called XML which stands for eXtensible Markup Lan- guage. XML is a meta-language for data description. A meta-language is a language for describing other languages. The XML meta-language is universally accepted. In fact, the XML format is governed by a standards committee, which means that we can count on the XML format remaining very stable and backwards compatible forever. Any additions to XML will have to be compatible with what has already been defined.

An XML document begins with a special line to identify it as an XML file. This line looks like this.

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

The rest of an XML file consists of elements or nodes. Each node is identified by a tag or a pair of beginning and ending tags. Each tag is delimited (i.e. surrounded) by angle brackets. For instance, here is one such tag.

<GraphicsCommands>

Most XML elements are delimited by a opening tag and a closing tag. The tag above is an opening tag. Its matching closing tag looks like this.

</GraphicsCommands>

The slash just before the tag name means that it is a closing tag. An opening and closing tag may have text or other XML elements in between the two tags so XML documents may contain XML elements nested as deeply as necessary depending on the data you are trying to encode.

Each XML element may have attributes associated with it. For instance, consider an XML element that encapsulates the information needed to do a *GoTo* graphics command. To complete a *GoTo* command we need the *x* and *y* coordinates, the *width* of the line, and the pen *color*. Here is an example of encoding that information in XML format.

<**Command** x="1.0" y="1.0" width="1.0" color="#000000">GoTo</**Command**>

In this example the attributes are *x*, *y*, *width*, and *color*. Each attribute is mapped to its value as shown above. The *GoTo* text is the text that appears between the opening and closing tags. That text is sometimes called the child data.

By encoding an entire graphics commands input file in XML format we eliminate some of the dependence between the Draw *program* and its *data*. Except for the XML format (i.e. the grammar) the contents of the XML file are completely up to the programmer or programmers using the data. The file in Sect. 1.15.1 is an example of the truck picture's XML input file.

1.15.1 THE TRUCK XML FILE

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2 <GraphicsCommands>
3 <Command color="black">BeginFill</Command>
4 <Command radius="20.0" width="1" color="black">Circle</Command>
5 <Command>EndFill</Command>
6 <Command>PenUp</Command>
7 <Command x="120.0" y="0.0" width="1.0" color="black">GoTo</Command>
8 <Command>PenDown</Command>
9 <Command color="black">BeginFill</Command>
10 <Command radius="20.0" width="1" color="black">Circle</Command>
11 <Command>EndFill</Command>
12 <Command>PenUp</Command>
13 <Command x="150.0" y="40.0" width="1.0" color="black">GoTo</Command>
14 <Command>PenDown</Command>
15 <Command color="yellow">BeginFill</Command>
16 <Command x="-30.0" y="40.0" width="1.0" color="black">GoTo</Command>
17 <Command x="-30.0" y="70.0" width="1.0" color="black">GoTo</Command>
18 <Command x="60.0" y="70.0" width="1.0" color="black">GoTo</Command>
19 <Command x="60.0" y="100.0" width="1.0" color="black">GoTo</Command>
20 <Command x="90.0" y="100.0" width="1.0" color="black">GoTo</Command>
```

```
21 <Command x="115.0" y="70.0" width="1.0" color="black">GoTo</Command>
22 <Command x="150.0" y="70.0" width="1.0" color="black">GoTo</Command>
23 <Command x="150.0" y="40.0" width="1.0" color="black">GoTo</Command>
24 <Command>EndFill</Command>
25 </GraphicsCommands>
```

XML files are text files. They just contain extra XML formatted data to help stan- dardize how XML files are read. Writing an XML file is as simple as writing a text file. While indentation is not necessary in XML files, it is often used to highlight the format of the file. In Sect. 1.15.1 the *GraphicsCommands* element contains one *Command* element for each drawing command in the picture. Each drawing com- mand contains the command type as its text. The command types are *GoTo*, *Circle*, *BeginFill*, *EndFill*, *PenUp*, and *PenDown*. The attributes of a command are data like *x*, *y*, *width*, *radius*, and *color* that are used by the various types of commands.

To write the commands to a file, each of the Command classes can be modified to produce an XML element when converted to a string using the special *str* method. For instance, Sect. 1.15.2 contains the modified GoToCommand class sup- porting the creation of an XML element.

1.15.2 THE GOTOCOMMAND WITH XML CREATION CODE

```
# The following classes define the different commands that
# are supported by the drawing application.
class GoToCommand:
    def init (self,x,y,width=1,color="black"):
        self.x = x
        self.y = y
        self.width = width
        self.color = color

        # The draw method for each command draws the command
        # using the given turtle
    def draw(self,turtle):
        turtle.width(self.width)
        turtle.pencolor(self.color)
        turtle.goto(self.x,self.y)
```

```
16
17 ____# The str method is a special method that is called
18         # when a command is converted to a string. The string
19         # version of the command is how it appears in the graphics
20         # file format.
21 ____def str (self):
22 return '<Command x="' + str(self.x) + '" y="' + str(self.y) + '" width="' + \
23 str(self.width) + '" color="' + self.color + '">GoTo</Command>'
```

By returning a string like this from each of the command objects, the code to write the draw program's data to a file is very simple. All that is needed is some code that writes the *xml* line as the first line, followed by the <*GraphicsCommands*> tag and the command elements. Finally, the <*GraphicsCommands*> tag must be written. The code in Sect. 1.15.3 accomplishes this.

1.15.3 WRITING GRAPHICS COMMANDS TO AN XML FILE

```
1 file = open(filename, "w")

2 file.write('<?xml version="1.0" encoding="UTF-8" standalone="no" ?>\n')

3 file.write('<GraphicsCommands>\n')

4 for cmd in self.graphicsCommands:

5 file.write(' '+str(cmd)+"\n")

6

7 file.write('</GraphicsCommands>\n')

8 file.close()
```

Writing an XML file is like writing any text file except that the text file must conform to the XML grammar specification. There are certainly ways to create XML files that differ from how it was presented in Sect. 1.15.3. In the next section we'll learn about XML parsers and a very simple way to read XML documents. It turns out there are at least some XML frameworks that make writing an XML document just as simple.

1.16 READING XML FILES

XML files would be difficult to read if we had to read them like we read a regular text file. This is especially true because XML files are not line-oriented. They conform to the XML grammar, but the grammar does not specify anything about the lines in the file. Instead of reading an XML file by reading lines of the file, we use a special tool called a *parser*. A *parser* is written according to the rules of a *grammar*, in this case the XML grammar. There are many XML parsers that have been written and different parsers have different features. The one we will use in this text is one of the simpler parsers called *minidom*. The *minidom* parser reads an entire XML file by calling the *parse* method on it. It places the entire contents of an XML file into an sequence of *Element* objects. An *Element* object contains the child data and attributes of an XML element along with any other elements that might be defined inside this element.

To use the minidom parser, you must first import the module where the minidom parser is defined.

```
import xml.dom.minidom
```

Then, you can read an entire XML file by calling the *parse* method on an XML document as follows.

```
xmldoc = xml.dom.minidom.parse(filename)
```

Once you have done that, you can read a specific type of element from the XML file by calling the method *getElementsByTagName* on it. For instance, to get the *GraphicsCommands* element from the graphics commands XML file, you would write this.

```
graphicsCommands = xmldoc.getElementsByTagName("GraphicsCommands")[0]
```

The XML document contains the GraphicsCommands element. Calling *getEle- mentsByTagName* on *GraphicsCommands* returns a list of all elements that match this tag name. Since we know there is only one of these tags in the file, we can write [0] to get the first element from the list. Then, the *graphicsCommands* element contains just the one element from the file and all the *Command* elements of the file are located within it. If we want to go through all these elements we can use a for loop as in the code in Sect. 1.16.1.

1.16.1 USING AN XML PARSER

```
1 for commandElement in graphicsCommands:
2 print(type(commandElement))
3 command = commandElement.firstChild.data.strip()
4 attr = commandElement.attributes
5 if command == "GoTo":
6 x = float(attr["x"].value)
7 y = float(attr["y"].value)
8 width = float(attr["width"].value)
9 color = attr["color"].value.strip()
10 cmd = GoToCommand(x,y,width,color)
11
12 elif command == "Circle":
13 radius = float(attr["radius"].value)
14 width = float(attr["width"].value)
15 color = attr["color"].value.strip()
16 cmd = CircleCommand(radius,width,color)
17
18 elif command == "BeginFill":
19 color = attr["color"].value.strip()
20 cmd = BeginFillCommand(color)
21
22 elif command == "EndFill":
```

```
23 cmd = EndFillCommand()

24

25 elif command == "PenUp":

26 cmd = PenUpCommand()

27

28 elif command == "PenDown":

29 cmd = PenDownCommand()

30 else:

31 raise RuntimeError("Unknown Command: " + command)

32

33 self.append(cmd)
```

In the code in Sect. 1.16.1 the *attr* variable is a dictionary mapping the attribute *names* (i.e. keys) to their associated *values*. The child data of a *Command* node can be found by looking at the *firstChild.data* for the node. The *strip* method is used to strip away any unwanted blanks, tabs, or newline characters that might appear in the string.

1.17 CHAPTER SUMMARY

In this first chapter we have covered a large amount of material which should be mostly review but probably covered some things that are new to you as well. Don't be too overwhelmed by it all. The purpose of this chapter is to get you asking questions about the things you don't understand. If you don't understand something, you should ask your teacher or someone who knows more about programming in Python. They can likely help you. Asking questions is a great way to learn and Computer Science is all about a lifetime of learning.

Here is a list of the important concepts you should have learned in this chapter.

You should:

- know how to create an *object*, both from a literal value and by calling the object's constructor explicitly.

- understand the concept of a *reference* pointing at a value (i.e. an object) in Python.

- know how to call a *method* on an object.

- know how to *import* a module.

- understand the importance of *indentation* in Python programs.

- know why you write a *main* function in Python programs and how to call the main function.

- know how to read records from a file whether they be multi-line, single line, fixed

length, or variable length records.

- know how to define a *container* class like PyList defined in this chapter.

- understand the concept of *polymorphism* and how that means an object will do the right thing when a method is called.

- understand the *Accumulator* pattern and how to use it in a program.

- know how to implement a simple GUI using Tkinter in Python. Entry boxes, labels, buttons, frames, and event handlers should all be concepts that are understood and can be programmed by looking back at the examples in this chapter.

- and finally you should know how to read and write XML files in your programs.

There is a lot of example code in this chapter and the final version of the *Draw* program is provided on the text's website or in Sect. 20.1. While it is doubtful you will be able to memorize each line of the code you found in this chapter, you should make sure you know how things work when you look at it and you should remember that you can use this chapter as a resource. Come back to it often when you need to see how to do something in later chapters. Using this example code as a reference will help to answer a lot of your questions in future chapters.

1.18 REVIEW QUESTIONS

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. What does IDE stand for and why is it a good idea to use an IDE?

2. What code would you write to create a string containing the words *Happy Birth- day!*? Write some code to point a reference called *text* at that newly created object.

3. What code would you write to take the string you created in the last question and split it into a list containing two strings? Point the reference *lst* at this newly created list.

4. What code would you write to upper-case the string created in the second ques- tion. Point the reference named *bDayWish* at this upper-cased string.

5. If you were to execute the code you wrote for answering the last three questions, what would the string referenced by *text* contain after executing these three lines of code?

6. How would you create a dictionary that maps “Kent” to “Denise” and maps “Steve” to “Lindy”? In these two cases “Kent” and “Steve” are the keys and “Denise” and “Lindy” are the values.

7. Consult Chap. 17. How would you map a key to a value as in the previous prob- lem when the dictionary was first created as an empty dictionary? HINT: This would be called setting an item in the documentation in the appendix. Write a short piece of code to create an empty dictionary and then map “Kent” to “Denise” and “Steve” to “Lindy”.

8. What method is called when $x < y$ is written? In which class is the method a member? In other words, if you were presented with $x < y$ in a program, how

would you figure out which class you needed to examine to understand exactly what $x < y$ meant?

9. What method is called when $x \ll y$ is written?

10. What is the loop and a half problem and how is it solved?

11. Do you need to use the solution to the loop and a half problem to read an XML file? Why or why not?

12. Polymorphism and Operator Overloading are closely related concepts. Can you briefly explain how the two concepts are similar and how Python supports them? HINT: No is not a valid answer.

13. What would you write so that a program asks the user to enter an integer and then adds up all the even integers from 2 to the integer entered by the user. HINT: You might want to review how to use the *range* function to accomplish this and decide on what pattern of programming you might use.

14. How do you create a window using Tkinter?

15. What is the purpose of a Frame object in a Tkinter program?

16. What are three types of widgets in the Tkinter framework?

17. When reading an XML file, how many lines of code does it take to read the file?

18. How do you get a single element from an XML document? What line(s) of code do you have to write? Provide an example.

19. When traversing an XML document, how do you get a list of elements from it? What line(s) of code do you have to write? Provide an example.

20. What is an attribute in an XML document and how do you access an attribute's value? Provide an example from the text or from another example you find online.

1.19 PROGRAMMING PROBLEMS

1. Starting with the version of the *Draw* program that reads an input file with variable length records, add a new graphics command of your choice to the program. Consider how it would be written to a file, create a test file, write your code, and test it. You must design two files: a sample test file, and the program itself. Some examples might be a graphics command to draw a star with some number of points, a rectangle with a height and width, etc.
2. Starting with the *Draw* program provided in Sect. 20.1, extend the program to include a new button to draw a new shape for the *Draw* program. For instance, have the draw program draw a star on the screen or a smiley face or something of your choosing. HINT: If you use the forward and back methods to draw your shape, you can scale it by multiplying each forward and back amount by a scale value. Then, you can let the user pick a scale for it (or use the radius amount as your scale) and draw your shape in whatever size you like. To complete this exercise you must extend your XML format to include a new graphics command to store the relevant information for drawing your new shape. You must also define a new graphicsCommand class for your new shape.
3. Add the ability to draw a text string on a *Draw* picture. You'll need to let the user pick a point size. For a real challenge, let the user pick the font type from a drop-down list of font types. Draw a string that you have the user enter in an entry box.
4. Find an XML document of your choice on the internet, write code to parse through the data and plot something from that data whether it be some value over time or something else. Use turtle graphics to plot the data that you find.
5. Add a new button to the drawing program presented in Sect. 20.1 that draws a rainbow centered above the current location of the turtle. This can be done quite easily by using *sin* and *cos* (i.e. sine and cosine). The *sin* and *cos* functions take radians as a parameter. To draw a rainbow,

the radians would range from 0 to *math.pi* from the math module. You must import the *math* module to get access to *math.cos* and *math.sin* as well as *math.pi*. To draw values in an arc, you can use a *for loop* and let a variable, *i*, range from 0 to 100. Then *radius * math.cos*(*i/100.0 * math.pi*), *radius * math.sin*(*i/100.0 * math.pi*) is the next *x,y* coordinate of the rainbow's arc. By varying the radius you will get several stripes for your rainbow.

Each stripe should have a different color. To vary the color, you might convert a 24-bit number to hex. To convert a number to hexadecimal in Python you can use the *hex* function. You must make sure that your color string is 6 digits long and starts with a pound sign (i.e. #) for it to be a valid color string in Python.

COMPUTATIONAL COMPLEXITY 2

In the last chapter we developed a drawing program. To hold the drawing commands we built the *PyList* container class which is a lot like the built-in Python list class, but helps illustrate our first data structure. When we added a drawing command to the sequence we called the append method. It turns out that this method is called a lot. In fact, the flower picture in the first chapter took around 700 commands to draw. You can imagine that a complex picture with lots of free-hand drawing could contain thousands of drawing commands. When creating a free-hand drawing we want to append the next drawing command to the sequence quickly because there are so many commands being appended. How long does it take to append a drawing command to the sequence? Can we make a guess? Should we care about the exact amount of time?

In this chapter you'll learn how to answer these questions and you'll learn what questions are important for you as a computer programmer. First you'll read about some principles of computer architecture to understand something about how long it takes a computer to do some simple operations. With that knowledge you'll have the tools you'll need to make informed decisions about how much time it might take to execute some code you have written.

2.1 CHAPTER GOALS

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

- What are some of the primitive operations that a computer can perform?
- How much time does it take to perform these primitive operations?
- What does the term *computational complexity* mean?
- Why do we care about *computational complexity*?
- When do we need to be concerned about the complexity of a piece of code?
- What can we do to improve the efficiency of a piece of code?
- What is the definition of Big-Oh notation?

- What is the definition of Theta notation?

- What is *amortized complexity* and what is its importance?

- How can we apply what we learned to make the *PyList* container class better?

2.2 COMPUTER ARCHITECTURE

A computer consists of a *Central Processing Unit* (i.e. the *CPU*) that interacts with *Input/Output* (i.e. *I/O*) devices like a keyboard, mouse, display, and network interface. When you run a program it is first read from a storage device like a hard drive into the *Random Access Memory*, or RAM, of the computer. RAM loses its contents when the power is shut off, so copies of programs are only stored in RAM while they are running. The permanent copy of a program is stored on the hard drive or some other permanent storage device.

The RAM of a computer holds a program as it is executing and also holds data that the program is manipulating. While a program is running, the CPU reads input from the input devices and stores data values in the RAM. The CPU also contains a very limited amount of memory, usually called *registers*. When an operation is performed by the CPU, such as adding two numbers together, the operands must be in registers in the CPU. Typical operations that are performed by the CPU are addition, subtraction, multiplication, division, and storing and retrieving values from the RAM.

2.2.1 RUNNING A PROGRAM

When a user runs a program on a computer, the following actions occur:

1. The program is read from the disk or other storage device into RAM.
2. The operating system (typically Mac OS X, Microsoft Windows, or Linux) sets up two more areas of RAM called the run-time stack and the heap for use by the program.
3. The operating system starts the program executing by telling the CPU to start executing the first instruction of the computer.
4. The program reads data from the keyboard, mouse, disk, and other input sources.
5. Each instruction of the program retrieves small pieces of data from RAM, acts on them, and writes new data back to RAM.
6. Once the data is processed the result is provided as output on the screen or some other output device.

Because there is so little memory in the CPU, the normal mode of operation is to store values in the RAM until they are needed for a CPU operation. The RAM is a much bigger storage space than the CPU. But, because it is bigger, it is also slower than the CPU. Storing a value in RAM or retrieving a value from RAM can take as much time as several CPU operations. When needed, the values are copied from the

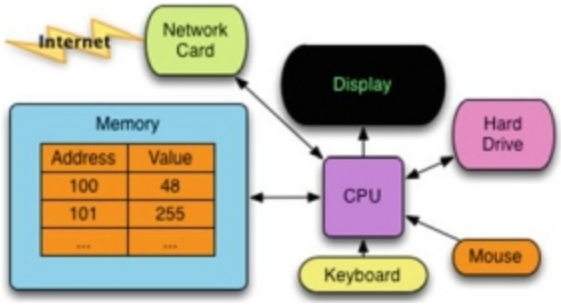

Fig. 2.1 Conceptual View of a Computer

RAM into the CPU, the operation is performed, and the result is then typically written back into the RAM. The RAM of a computer is accessed frequently as a program runs, so it is important that we understand what happens when it is accessed (Fig. 2.1). One analogy that is often used is that of a post office. The RAM of a computer is like a collection of post office boxes. Each box has an address and can hold a value. The values you can put in RAM are called bytes (i.e. eight bits grouped together). With eight bits, 256 different values can be stored. Usually bytes are interpreted as integers, so a byte can hold values from 0 to 255. If we want to store bigger values, we can group bytes together into words. The word size of a computer is either 32 bits (i.e. four bytes) or 64 bits, depending on the architecture of the computer's hardware. All modern computer hardware is capable of retrieving or storing a word at a time.

The post office box analogy helps us to visualize how the RAM of a computer is organized, but the analogy does not serve well to show us how the RAM of a computer *behaves*. If we were going to get something from a post office box, or store something in a post office box, there would have to

be some kind of search done to find the post office box first. Then the letter or letters could be placed in it or taken from it. The more post office boxes in the post office, the longer that search would take. This helps us understand the fundamental problem we study in this text. As the size of a problem space grows, how does a program or algorithm behave? In terms of this analogy, as the number of post office boxes grows, how much longer does it take to store or retrieve a value?

The RAM of a computer does not *behave* like a post office. The computer does not need to find the right RAM location before the it can retrieve or store a value. A much better analogy is a group of people, each person representing a memory location within the RAM of the computer. Each person is assigned an address or name. To store a value in a location, you call out the name of the person and then tell them what value to remember. It does not take any time to find the right person because all the people are listening, just in case their name is called. To retrieve a value, you call the name of the person and they tell you the value they were told to remember. In this way it takes exactly the same amount of time to retrieve any value from any memory location. This is how the RAM of a computer works. It takes exactly the same amount of time to store a value in any location within the RAM. Likewise, retrieving a value takes the same amount of time whether it is in the first RAM location or the last.

2.3 ACCESSING ELEMENTS IN A PYTHON LIST

With experimentation we can verify that all locations within the RAM of a computer can be accessed in the same amount of time. A Python list is a collection of contiguous memory locations. The word *contiguous* means that the memory locations of a list are grouped together consecutively in RAM. If we want to verify that the RAM of a computer behaves like a group of people all remembering their names and their values, we can run some tests with Python lists of different sizes to find the average time to retrieve from or store a value into a random element of the list.

To test the behavior of Python lists we can write a program that randomly stores and retrieves values in a list. We can test two different theories in this program.

1. The size of a list does not affect the average access time in the list.
2. The average access time at any location within a list is the same, regardless of its location within the list.

To test these two theories, we'll need to time retrieval and storing of values within a list. Thankfully, Python includes a datetime module that can be used to record the current time. By subtracting two datetime objects we can compute the number of microseconds (i.e. millionths of a second) for any operation within a program. The program in Sect. 2.3.1 was written to test list access and record the access time for retrieving values and storing values in a Python list.

2.3.1 LIST ACCESS TIMING

```
1 import datetime
2 import random
3 import time
4
5 def main():
6
7 # Write an XML file with the results
8 file = open("ListAccessTiming.xml","w")
9
10 file.write('<?xml version="1.0" encoding="UTF-8" standalone="no" ?>\n')
11
12 file.write('<Plot title="Average List Element Access Time">\n')
13
```

1. # Test lists of size 1000 to 200000.
2. xmin = 1000
3. xmax = 200000

```
17
18 # Record the list sizes in xList and the average access time within
19 # a list that size in yList for 1000 retrievals.
20 xList = []
21 yList = []
22
23 for x in range(xmin, xmax+1, 1000):
```

```
24
25 xList.append(x)
26
27 prod = 0
28
```

```
    # Creates a list of size x with all 0's
    lst = [0] * x

31

    # let any garbage collection/memory allocation complete or at least
    # settle down
    time.sleep(1)

35

    # Time before the 1000 test retrievals
    starttime = datetime.datetime.now()

38

    for v in range(1000):
    # Find a random location within the list
    # and retrieve a value. Do a dummy operation
    # with that value to ensure it is really retrieved.
    index = random.randint(0,x-1)
    val = lst[index]
    prod = prod * val
    # Time after the 1000 test retrievals
    endtime = datetime.datetime.now()

48

    # The difference in time between start and end.
    deltaT = endtime - starttime

51

52 # Divide by 1000 for the average access time

53 # But also multiply by 1000000 for microseconds.

54 accessTime = deltaT.total_seconds() * 1000

55

56 yList.append(accessTime)

57
```

```
58 file.write(’ <Axes>\n’)
59 file.write(’ <XAxis min="’+str(xmin)+’" max="’+str(xmax)+’">List Size</XAxis>\n’)
60 file.write(’ <YAxis min="’+str(min(yList))+’" max="’+str(60)+’">Microseconds</YAxis>\n’)
61 file.write(’ </Axes>\n’)
62
63 file.write(’ <Sequence title="Average Access Time vs List Size" color="red">\n’)
64
65 for i in range(len(xList)):
66 file.write(’ <DataPoint x="’+str(xList[i])+’" y="’+str(yList[i])+’"/>\n’)
67
68 file.write(’ </Sequence>\n’)
69
```

1. *# This part of the program tests access at 100 random locations within a list*
2. *# of 200,000 elements to see that all the locations can be accessed in*
3. *# about the same amount of time.*
4. xList = lst
5. yList = [0] * 200000

```
75
76 time.sleep(2)
77
```

1. **for** i **in** range(100):
2. starttime = datetime.datetime.now()
3. index = random.randint(0,200000-1)
4. xList[index] = xList[index] + 1
5. endtime = datetime.datetime.now()
6. deltaT = endtime - starttime
7. yList[index] = yList[index] + deltaT.total_seconds() * 1000000

```
85
86 file.write(’ <Sequence title="Access Time Distribution" color="blue">\n’)
```

87

```
1. for i in range(len(xList)):
2. if xList[i] > 0:
3. file.write(’ <DataPoint x="’+str(i)+’" y="’+str(yList[i]/xList[i])+’"/>\n’)
```

91

```
1. file.write(’ </Sequence>\n’)
2. file.write(’</Plot>\n’)
3. file.close()
```

```
1. if  name  == " main ":
2. main()
```

When running a program like this the times that you get will depend not only on the actual operations being performed, but the times will also depend on what other activity is occurring on the computer where the test is being run. All modern operating systems, like Mac OS X, Linux, or Microsoft Windows, are multi-tasking. This means the operating system can switch between tasks so that we can get email while writing a computer program, for instance. When we time something we will not only see the effects of our own program running, but all programs that are currently running on the computer. It is nearly impossible to completely isolate one program in a multi-tasking system. However, most of the time a short program will run without too much interruption.

The program in Sect. 2.3.1 writes an XML file with its results. The XML file format supports the description of experimentally collected data for a two dimensional plot of one or more sequences of data. One sample of the data that this program generates looks like Sect. 2.3.2. The data is abbreviated, but the format is as shown in Sect. 2.3.2.

2.3.2 A PLOT XML SAMPLE

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2 <Plot title="Average List Element Access Time">
3 <Axes>
4 <XAxis min="1000" max="200000">List Size</XAxis>
5 <YAxis min="20.244" max="60">Microseconds</YAxis>
6 </Axes>
7 <Sequence title="Average Access Time vs List Size" color="red">
8 <DataPoint x="1000" y="33.069"/>
9 <DataPoint x="2000" y="27.842"/>
10 <DataPoint x="3000" y="23.908"/>
11 <DataPoint x="4000" y="26.349"/>
12 <DataPoint x="5000" y="23.212"/>
13 <DataPoint x="6000" y="23.765"/>
14 <DataPoint x="7000" y="21.251"/>
15 <DataPoint x="8000" y="21.321"/>
16 <DataPoint x="9000" y="23.197"/>
17 <DataPoint x="10000" y="21.527"/>
18 <DataPoint x="11000" y="35.799"/>
19 <DataPoint x="12000" y="22.173"/>
20 ...
21 <DataPoint x="197000" y="26.245"/>
22 <DataPoint x="198000" y="30.013"/>
```

23 <**DataPoint** x="199000" y="25.888"/>

24 <**DataPoint** x="200000" y="23.578"/>

25 </**Sequence**>

26 <**Sequence** title="Access Time Distribution" color="blue">

27 <**DataPoint** x="219" y="41.0"/>

28 <**DataPoint** x="2839" y="38.0"/>

29 <**DataPoint** x="5902" y="38.0"/>

30 <**DataPoint** x="8531" y="58.0"/>

31 <**DataPoint** x="11491" y="38.0"/>

32 <**DataPoint** x="15415" y="38.0"/>

33 <**DataPoint** x="17645" y="31.0"/>

34 <**DataPoint** x="18658" y="38.0"/>

35 <**DataPoint** x="20266" y="40.0"/>

36 <**DataPoint** x="21854" y="31.0"/>

```
37 ...
38 <DataPoint x="197159" y="37.0"/>
39 <DataPoint x="199601" y="40.0"/>
40 </Sequence>
41 </Plot>
```

Since we'll be taking a look at quite a bit of experimental data in this text, we have written a Tkinter program that will read an XML file with the format given in Sect. 2.3.2 and plot the sequences to the screen. The PlotData.py program is given in Chap. 20.4.

If we use the program to plot the data gathered by the list access experiment, we see a graph like the one in Fig. 2.2. This graph provides the experimental data to back up the two statements we made earlier about lists in Python. The red line shows the average element access time of 1,000 element accesses on a list of the given size. The average access time (computed from a sample of 1,000 random list accesses) is no longer on a list of 10,000 than it is on a list of 160,000. While the exact values are not printed in the graph, the exact values are not important. What we would be interested in seeing is any trend toward longer or shorter average access times. Clearly the only trend is that the size of the list does not affect the average access time. There are some ups and downs in the experimental data, but this is caused by the system being

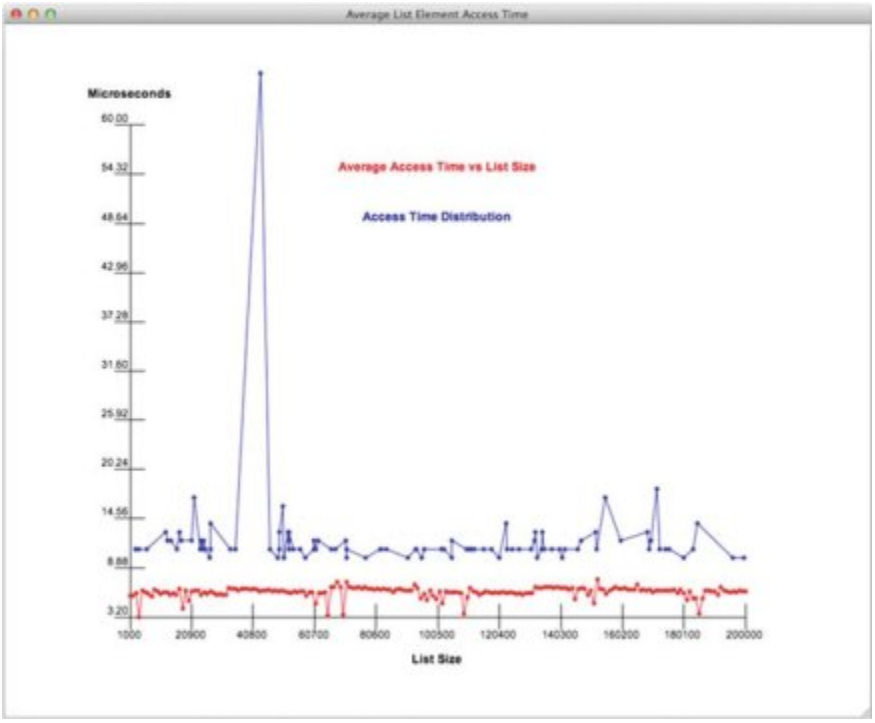

Fig. 2.2 Access Times in a Python List

a multi-tasking system. Another factor is likely the caching of memory locations. A cache is a way of speeding up access to memory in some situations and it is likely that the really low access times benefited from the existence of a cache for the RAM of the computer. The experimental data backs up the claim that *the size of a list does not affect the average access time in the list*.

The blue line in the plot is the result of doing 100 list retrieval and store operations on one list of 200,000 elements. The reason the blue line is higher than the red line is likely the result of doing both a retrieval from and a store operation into the element of the list. In addition, the further apart the values in memory, the less likely a cache will help reduce the access time. Whatever the reason for the blue line being higher the important thing to notice is that accessing the element at index 0 takes no more time than accessing any other element of the sequence. All locations within the list are treated equally. This backs up the claim that *the average access time at any location within a list is the same, regardless of its location within the list*.

2.4 BIG-OH NOTATION

Whichever line we look at in the experimental data, the access time never exceeds 100 μs for any of the memory accesses, even with the other things the computer might be doing. We are safe concluding that accessing memory takes less than 100 μs. In fact, 100 μs is much more time than is needed to access or store a value in a memory location. Our experimental data backs up the two claims we made earlier. However, technically, it does not prove our claim that accessing memory takes a constant amount of time. The architecture of the RAM in a computer could be examined to prove that accessing any memory location takes a constant amount of time. Accessing memory is just like calling out a name in a group of people and having that person respond with the value they were assigned. It doesn't matter which person's name is called out. The response time will be the same, or nearly the same. The actual time to access the RAM of a computer may vary a little bit if a cache is available, but at least we can say that there is an upper bound to how much time accessing a memory location will take.

This idea of an upper bound can be stated more formally. The formal statement of an upper bound is called Big-Oh notation. The Big-Oh refers to the Greek letter Omicron which is typically used when talking about upper bounds. As computer programmers, our number one concern is how our programs will perform when we have large amounts of data. In terms of the memory of a computer, we wanted to know how our program would perform if we have a very large list of elements. We found that all elements of a list are accessed in the same amount of time independent of how big this list is. Let's represent the size of the list by a variable called *n*. Let the average access time for accessing an element of a list of size *n* be given by *f* (*n*). Now we can state the following.

$$O(g(n)) = \{ f \mid \exists d > 0, n_0 \in Z^+ \; 3 \; 0 \leq f(n) \leq d\, g(n), \forall n \geq n_0 \}$$

In English this reads as follows: The class of functions designated by $O(g(n))$ consists of all functions f, where there exists a d greater than 0 and an n_0 (a positive integer) such that 0 is less than or equal to $f(n)$ is less than or equal to d times $g(n)$ for all n greater than or equal to n_0.

If f is an element of $O(g(n))$, we say that $f(n)$ is $O(g(n))$. The function g is called an asymptotic upper bound for f in this case. You may not be comfortable with the mathematical description above. Stated in English the set named $O(g(n))$ consists of the set of all functions, $f(n)$, that have an upper bound of $d^* g(n)$, as n approaches infinity. This is the meaning of the word *asymptotic*. The idea of an asymptotic bound means that for some small values of n the value of $f(n)$ might be bigger than the value of $d^* g(n)$, but once n gets big enough (i.e. bigger than n_0), then for all bigger n it will always be true that $f(n)$ is less than $d^* g(n)$. This idea of an asymptotic upper bound is pictured in Fig. 2.3. For some smaller values the function's performance, shown in green, may be worse than the blue upper bound line, but eventually the upper bound is bigger for all larger values of n.

We have seen that the average time to access an element in a list is constant and does not depend on the list size. In the example in Fig. 2.2, the list size is the n in the definition and the average time to access an element in a list of size n is the $f(n)$.

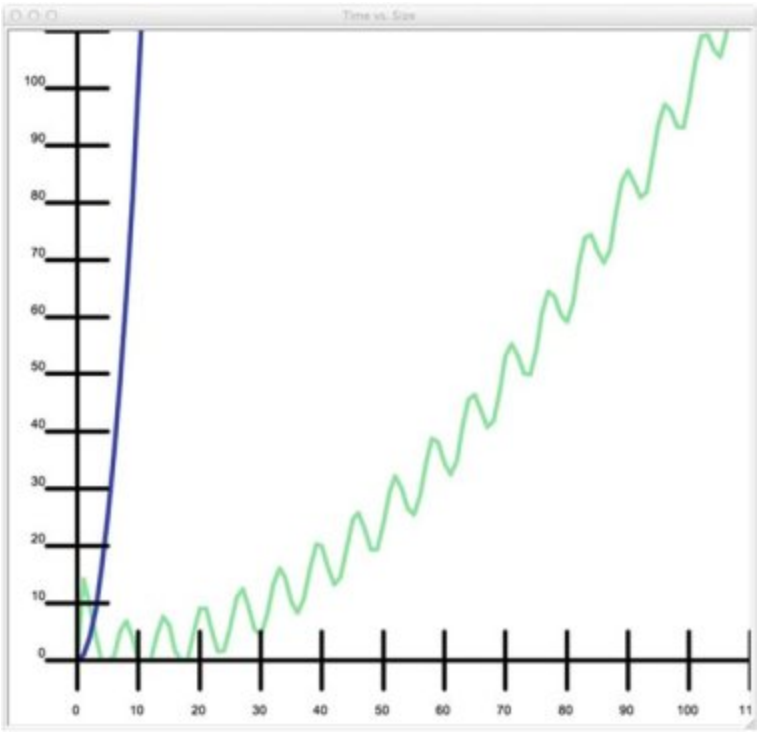

Fig. 2.3 An Upper Bound

Because the time to access an element does not depend on *n*, we can pick *g*(n) = 1. So, we say that the average time to access an element in a list of size *n* is O(1). If we assume it never takes longer than 100 µs to access an element of a list in Python, then a good choice for *d* would be 100. According to the definition above then it must be the case that f(n) is less than or equal to 100 once *n* gets big enough.

The choice of *g*(n) = 1 is arbitrary in computing the complexity of accessing an element of a list. We could have chosen *g*(n) = 2. If *g*(n) = 2 were chosen, *d* might be chosen to be 50 instead of 100. But, since we are only concerned with the overall

growth in the function *g*, the choice of 1 or 2 is irrelevant and the simplest function is chosen, in this case O(1). In English, when an operation or program is O(1), we say it is a *constant time* operation or program. This means the operation does not depend on the size of *n*.

It turns out that most operations that a computer can perform are O(1). For instance, adding two numbers together is a O(1) operation. So is multiplication of two numbers. While both operations require several cycles in a computer, the total number of cycles does not depend on the size of the integers or floating point numbers being added or multiplied. A cycle is simply a unit of time in a computer. Comparing two values is also a constant time operation. When computing complexity, any arithmetic calculation or comparison can be considered a constant time operation.

This idea of computational complexity is especially important when the com- plexity of a piece of code depends on *n*. In the next section we'll see some code that depends on the size of the list it is working with and how important it is that we understand the implications of how we write even a small piece of code.

2.5 THE PYLIST APPEND OPERATION

We have established that accessing a memory location or storing a value in a memory location is a O(1), or constant time, operation. The same goes for accessing an element of a list or storing a value in a list. The size of the list does not change the time needed to access or store an element and there is a fixed upper bound for the amount of time needed to access or store a value in memory or in a list.

With this knowledge, let's look at the drawing program again and specifically at the piece of code that appends graphics commands to the PyList. This code is used a lot in the program. Every time a new graphics command is created, it is appended to the sequence. When the user is doing some free-hand drawing, hundreds of graphics commands are getting appended every minute or so. Since free-hand drawing is somewhat compute intensive, we want this code to be as efficient as possible.

2.5.1 INEFFICIENT APPEND

```
1 class PyList:
2 def  init (self):
3 self.items = []
4
5 # The append method is used to add commands to the sequence.
6 def append(self,item):
7 self.items = self.items + [item]
8
9 ...
```

The code in Sect. 2.5.1 appends a new item to the list as follows:

1. The item is made into a list by putting [and] around it. We should be careful about how we say this. The item itself is not changed. A new list is constructed from the item.

2. The two lists are concatenated together using the + operator. The + operator is an accessor method that does not change either original list. The concatenation

creates a new list from the elements in the two lists.

3. The assignment of *self.items* to this new list updates the PyList object so it now refers to the new list.

The question we want to ask is, how does this append method perform as the size of the PyList grows? Let's consider the first time that the append method is called. How many elements are in the list that is referenced by

self.items? Zero, right? And there is always one element in [item]. So the append method must access one element of a list to form the new list, which also has one element in it.

What happens the second time the append method is called? This time, there is one element in the list referenced by *self.items* and again one element in [*item*]. Now, two elements must be accessed to form the new list. The next time append is called three elements must be accessed to form the new list. Of course, this pattern continues for each new element that is appended to the PyList. When the nth element is appended to the sequence there will have to be *n* elements copied to form the new list. Overall, how many elements must be accessed to append *n* elements?

2.6 A PROOF BY INDUCTION

We have already established that accessing each element of a list takes a constant amount of time. So, if we want to calculate the amount of time it takes to append *n* elements to the PyList we would have to add up all the list accesses and multiply by the amount of time it takes to access a list element plus the time it takes to store a list element. To count the total number of access and store operations we must start with the number of access and store operations for copying the list the first time an element is appended. That’s one element copied. The second append requires two

copy operations. The third append requires three copy operations. So, we have the following number of list elements being copied.

$$1 + 2 + 3 + 4 + \cdots + n = \sum_{i=1}^{n} i$$

In mathematics we can express this sum with a summation symbol (i.e. $\sum$). This is the mathematical way of expressing the sum of the first n integers. But, what is this equal to? It turns out with a little work, we can find that the following is true.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

We can prove this is true using a proof technique from Mathematics called mathe- matical induction. There are a couple of variations of mathematical induction. We'll use what is called weak induction to prove this. When proving something using induction you are really constructing a meta-proof. A meta-proof is a set of steps that you can repeat over and over again to find your desired result. The power of induction is that once we have constructed the meta-proof, we have proved that the result is true for all possible values of n.

We want to prove that the formula given above is valid for all *n*. To do this we first show it is true for a simple value of *n*. In our case we'll pick 1 as our value of

n. In that case we have the following.

$$\sum_{i=1}^{1} 1 = \frac{(1+1)}{2}$$

This is surely true. This step is called the *base case* of the inductive proof. Every

proof by induction must have a base case and it is usually trivial.

The next step is to create the meta-proof. This meta-proof is called the *inductive case*. When forming the inductive case we get to assume that the formula holds for all values, *m*, where *m* is less than *n*. This is called *strong induction*. In *weak induction* we get to assume that the formula is valid for *n* −1 and we want to show that it is valid for

n. We'll use weak induction in this problem to finish our proof. Again, this step helps

us form a set of steps that we can apply over and over again to get from our base case to whatever value of *n* we need to find. To begin we will make note of the following.

$$i =$$

$$i =1$$

$$\mathbb{N}^{-1}$$

$$i + n$$

$$i =1$$

This is true by the definition of summation. But now we have a sum that goes to $n - 1$ and weak induction says that we know the equation is valid for $n - 1$. This is called the *inductive hypothesis*. Since it holds for $n - 1$ we know the following is true. We get this by substituting $n - 1$ everyplace that we see an n in the original formula.

$$\underline{n} \quad \underline{1}\,\underline{n}$$

$$n-1$$

$$i = (\underline{\quad} \;)$$

$$2$$

$$i =1$$

Now we can use this fact in proving the equality of our original formula. Here we go!

$$\sum_{i=1}^{n-1} i =$$

$$\sum_{i=1}^{n} i =$$

$$+ n =$$

$$\frac{n(n-1)}{2} + n =$$

$$\frac{n(n-1) + 2n}{2} =$$

$$\frac{n^2 - n + 2n}{2} =$$

$$\frac{n2 + n}{2} =$$

$$\frac{n(n+1)}{2}$$

If you look at the left side and all the way over at the right side of this formula you can see the two things that we set out to prove were equal are indeed equal. This concludes our proof by induction. The meta-proof is in the formula above. It is a template that we could use to prove that the equality holds for $n = 2$. To prove the

equality holds for $n = 2$ we needed to use the fact that the equality holds for $n = 1$.

This was our base case. Once we have proved that it holds for $n = 2$ we could use that same formula to prove that the equality holds for $n = 3$. Mathematical induction doesn't require us to go through all the steps. As long as we've created this meta-proof

we have proved that the equality holds for all n. That's the power of induction.

2.7 MAKING THE PYLIST APPEND EFFICIENT

Now, going back to our original problem, we wanted to find out how much time it takes to append *n* items to a PyList. It turns out, using the append method in Sect. 2.5.1, it will perform in $O(n^2)$ time. This is because the first time we called append we had to copy one element of the list. The second time we needed to copy two elements. The

third time append was called we needed to copy three elements. Our proof in Sect. 2.6 is that $1 + 2 + 3 + \cdots + n$ equals $n*(n + 1)/2$. The highest powered term in this formula is the n^2 term. Therefore, the append method in Sect. 2.5.1 exhibits $O(n^2)$ complexity. This is not really a good result. The red curve in the graph of Fig. 2.4 shows the actual results of how much time it takes to append 200,000 ele-

ments to a *PyList*. The line looks somewhat like the graph of $f(n) = n^2$. What this tells us is that if we were to draw a complex program with say 100,000 graphics com-

mands in it, to add one more command to the sequence it would take around 27 s. This is unacceptable! We may never draw anything that complex, but a computer should be able to add one more graphic command quicker than that!

In terms of big-Oh notation we say that the *append* method is $O(n^2)$. When *n* gets large, programs or functions with $O(n^2)$ complexity are not very good. You typically want to stay away from writing code that has this kind of computational complex- ity associated with it unless you are absolutely sure it will never be called on large data sizes.

One real-world example of this occurred a few years ago. A tester was testing some code and placed a CD in a CD drive. On this computer all the directories and file names on the CD were read into memory and sorted alphabetically. The sorting algorithm that was used in that case had $O(n^2)$ complexity. This was OK because most CDs put in this computer had a relatively small number of directories and files on them. However, along came one CD with literally hundreds of thousands of files

on it. The computer did nothing but sort those file names alphabetically for around 12 h. When this was discovered, the programmer rewrote the sorting code to be more efficient and reduced the sorting time to around 15 s. That's a BIG difference! It also illustrates just how important this idea of computational complexity is.

If we take another look at our PyList append method we might be able to make it more efficient if we didn't have to access each element of the first list when concatenating the two lists. The use of the + operator is what causes Python to

access each element of that first list. When + is used a new list is created with space

for one more element. Then all the elements from the old list must be copied to the

new list and the new element is added at the end of this list.

Using the *append* method on lists changes the code to use a mutator method to alter the list by adding just one more element. It turns out that adding one more element to an already existing list is very efficient in Python. In fact, appending an item to a list is a O(1) operation as we'll see later in this chapter. This means to append *n* items to a list we have gone from $O(n^2)$ to O(n). Later in this chapter we'll learn just how Python can insure that we get O(1) complexity for the append operation. The blue line in Fig. 2.4 shows how the PyList append method works when the

+ operator is replaced by calling the list append method instead. At 100,000 elements

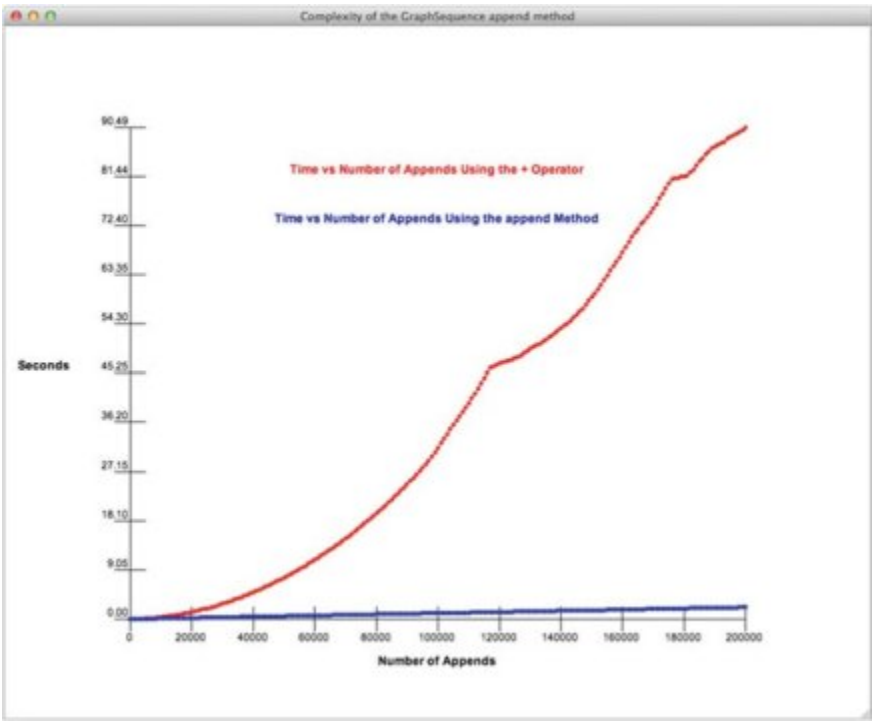

Fig. 2.4 The Complexity of Appending to a Pylist

in the PyList we go from 27 s to add another element to maybe a second, but probably less than that. That's a nice speedup in our program. After making this change, the PyList append method is given in Sect. 2.7.1.

2.7.1 EFFICIENT APPEND

```
1 class PyList:
2 def  init (self):
3 self.items = []
4
5 # The append method is used to add commands to the sequence.
6 def append(self,item):
7 self.items.append(item)
8
9 ...
```

2.8 COMMONLY OCCURRING COMPUTATIONAL COMPLEXITIES

The algorithms we will study in this text will be of one of the complexities of O(1), O(log n), O(n log n), O(n^2), or O(c^n). A graph of the shapes of these functions appears in Fig. 2.5. Most algorithms have one of these complexities corresponding to some factor of *n*. Constant values added or multiplied to the terms in a formula for measur- ing the time needed to complete a computation do not affect the overall complexity of that operation. Computational complexity is only affected by the highest power term of the equation. The complexities graphed in Fig. 2.5 are of some power *n* or the log of *n*, except for the really awful exponential complexity of O(c^n), where c is some constant value.

As you are reading the text and encounter algorithms with differing complexities, they will be one of the complexities shown in Fig. 2.5. As always, the variable *n* represents the size of the data provided as input to the algorithm. The time taken to process that data is the vertical axis in the graph. While we don't care about the exact numbers in this graph, we do care about the overall shape of these functions. The flatter the line, the lower the slope, the better the algorithm performs. Clearly an algorithm that has exponential complexity (i.e. O(c^n)) or n-squared complexity (i.e. O(n^2)) complexity will not perform very well except for very small values of *n*. If you know your algorithm will never be called for large values of *n* then an inefficient algorithm might be acceptable, but you would have to be really sure that you knew that your data size would always be small. Typically we want to design algorithms that are as efficient as possible.

In subsequent chapters you will encounter sorting algorithms that are O(n^2) and then you'll learn that we can do better and achieve O(n log n) complexity. You'll see search algorithms that are O(n) and then learn how to achieve O(log n) com- plexity. You'll also learn a technique called

hashing that will search in O(1) time. The techniques you learn will help you deal with large amounts of data as efficiently

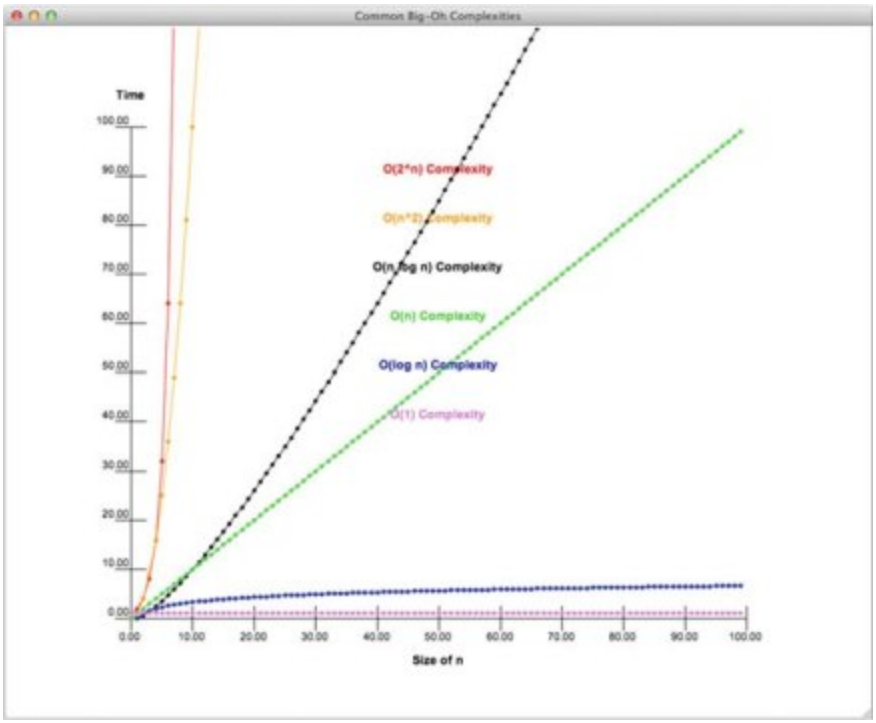

Fig. 2.5 Common Big-Oh Complexities

as possible. As each of these techniques are explored, you'll also have the opportu- nity to write some fun programs and you'll learn a good deal about object-oriented programming.

2.9 MORE ASYMPTOTIC NOTATION

Earlier in this chapter we developed Big-Oh notation for describing an upper bound on the complexity of an algorithm. There we began with an intuitive understanding of the idea of efficiency saying that a function exhibits a complexity if it is bounded above by a function of *n* where *n* represents the size of the data given to the algo- rithm. In this section we further develop these concepts to bound the efficiency of an algorithm from both above and below.

We begin with an in-depth discussion of efficiency and the measurement of it in Computer Science. When concerning ourselves with algorithm efficiency there are two issues that must be considered.

- The amount of time an algorithm takes to run
- and, related to that, the amount of space an algorithm uses while running.

Typically, computer scientists will talk about a space/time tradeoff in algorithms. Sometimes we can achieve a faster running time by using more memory. But, if we use too much memory we can slow down the computer and other running programs. The *space* that is referred to is the amount of *RAM* needed to solve a problem. The *time* we are concerned with is a measure of how the number of operations grow as the size of the data grows.

Consider a function $T(n)$ that is a description of the running time of an algorithm, where n is the size of the data given to the algorithm. As computer scientists we want to study the *asymptotic behavior* of this function. In other words, we want to study how $T(n)$ increases as $n \rightarrow \infty$. The value of n is a Natural number representing possible sizes of input data. The natural numbers are the set of non-negative integers. The

definition in Sect. 2.9.1 is a re-statement of the Big-Oh notation definition presented earlier in this chapter.

2.9.1 BIG-OH ASYMPTOTIC UPPER BOUND

$$O(g(n)) = \{ f(n) \mid \exists d > 0 \text{ and } n_0 > 0 \; 3 \; 0 \leq f(n) \leq dg(n) \; \forall n \geq n_0 \}$$

We write that

$$f(n) \text{ is } O(g(n)) \Leftrightarrow f \in O((g(n)))$$

and we say that *f* is big-oh *g* of *n*. The definition of Big-Oh says that we can find an upper bound for the time it will take for an algorithm to run. Consider the plot of time versus data size given in Fig. 2.3. Data size, or *n* is the x axis, while time is the *y* axis. Imagine that the green line represents the observed behavior of some algorithm.

The blue line clearly is an upper bound to the green line after about n = 4. This is what the definition of big-Oh means. For a while, the upper bounding function may not be an upper bound, but eventually it becomes an upper bound and stays that way all the way to the limit as *n* approaches infinity.

But, does the blue line represent a tight bound on the complexity of the algorithm whose running time is depicted by the green line? We'd like to know that when we describe the complexity of an algorithm it is truly representational of the actual running time. Saying that the algorithm runs in $O(n^2)$ is accurate even if the algorithm runs in time proportional to *n* because Big-Oh notation only describes an upper bound. If we truly want to say what the algorithm's running time is proportional to, then we need a little more power. This leads us to our next definition in Sect. 2.9.2.

2.9.2 ASYMPTOTIC LOWER BOUND

$Q(g(n)) = \{ f(n) \mid \exists c > 0 \text{ and } n_0 > 0\ 3\ 0 \leq cg(n) \leq f(n)\ \forall n \geq n_0 \}$

Omega notation serves as a way to describe a lower bound of a function. In this case the lower bound definition says for a while it might be greater, but eventually there is some n_0 where $T(n)$ dominates $g(n)$ for all bigger values of n. In that case, we can write that the algorithm is $Q(g(n))$. Considering our graph once again, we see that

the purple line is dominated by the observed behavior sometime after $n = 75$. As with the upper bound, for a while the lower bound may be greater than the observed

behavior, but after a while, the lower bound stays below the observed behavior for all bigger values of n.

With both a lower bound and and upper bound definition, we now have the notation to define an asymptotically tight bound. This is called *Theta* notation.

2.9.3 THETA ASYMPTOTIC TIGHT BOUND

$0(g(n)) = \{ f(n) | \exists\ c > 0, d > 0\ and\ n_0 > 0\ 3\ 0 \leq cg(n) \leq f(n) \leq dg(n)\ \forall n \geq n_0\}$

If we can find such a function *g*, then we can declare that $0(g(n))$ is an asymptotically tight bound for $T(n)$, the observed behavior of an algorithm. In Fig. 2.6 the upper

bound blue line is $g(n) = n^2$ and the lower bound purple line is a plot of $g(n)/110$. If we let $c = 1$ and $d = 1/110$, we have the asymptotically tight bound of $T(n)$ at $0(n^2)$. Now, instead of saying that n-squared is an upper bound on the algorithm's behavior,

we can proclaim that the algorithm truly runs in time proportional to n-squared. The behavior is bounded above and below by functions of n-squared proving the claim that the algorithm is an n-squared algorithm.

2.10 AMORTIZED COMPLEXITY

Sometimes it is not possible to find a tight upper bound on an algorithm. For instance, most operations may be bounded by some function *c*g*(n) but every once in a while there may be an operation that takes longer. In these cases it may be helpful to employ something called *Amortized Complexity*. Amortization is a term used by accountants when spreading the cost of some business transaction over a number of years rather than applying the whole expense to the books in one fiscal year. This same idea is employed in Computer Science when the cost of an operation is averaged. The key idea behind all amortization methods is to get as tight an upper bound as we can for the worst case running time of any sequence of *n* operations on a data structure (which usually starts out empty). By dividing by *n* we get the average or *amortized* running time of each operation in the sequence.

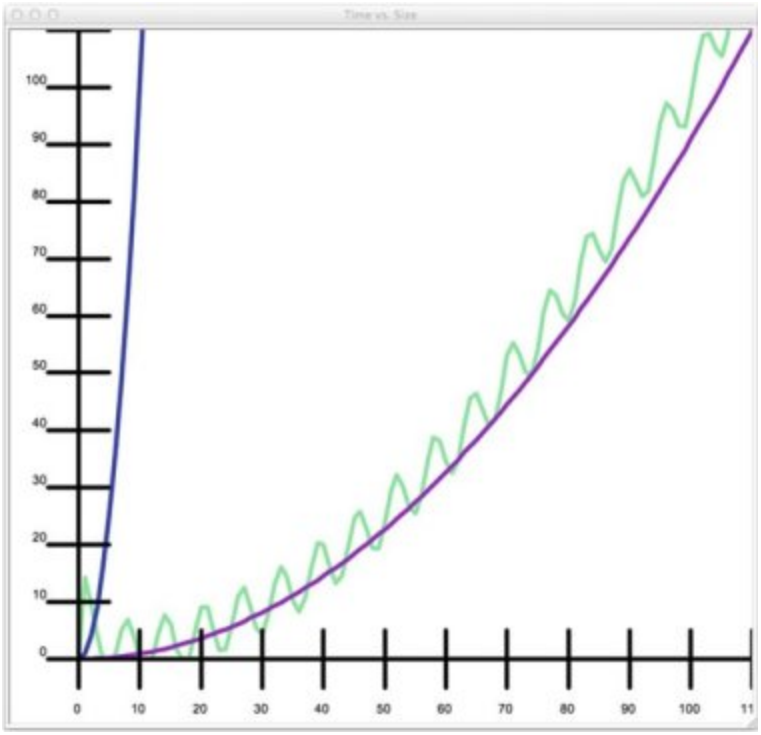

Fig. 2.6 A Lower and Upper Bound

Consider the PyList append operation discussed earlier in this chapter. The latest version of the PyList append method simply calls the Python append operation on lists. Python is implemented in C. It turns out that while Python supports an append operation for lists, lists are implemented as arrays in C and it is not possible to add to an array in C. An array can be allocated with a fixed size, but cannot have its size increased once created.

Pretend for a moment that Python lists, like C arrays, did not support the *append* method on lists and that the only way to create a list was to write something like *[None]*n* where *n* was a fixed value. Writing *[None]*n* creates a fixed size list of *n* elements each referencing the value *None*. This is the way C and C++ arrays are allo-

cated. In our example, since we are pretending that Python does not support append

we must implement our PyList append method differently. We can’t use the *append*

method and earlier in this chapter we saw that that adding on item at a time with the

+ operator was a bad idea. We’ll do something a little different. Our PyList append operation, when it runs out of space in the fixed size list, will double the size of the list copying all items from the old list to the new list as shown in the code in Sect. 2.10.1.

2.10.1 A PYLIST CLASS

```
1 class PyList:
2 # The size below is an initial number of locations for the list object. The
3 # numItems instance variable keeps track of how many elements are currently stored
4 # in the list since self.items may have empty locations at the end.
    5 def init (self,size=1):
6 self.items = [None] * size
7 self.numItems = 0
8
9 def append(self,item):
10 if self.numItems == len(self.items):
11 # We must make the list bigger by allocating a new list and copying
12 # all the elements over to the new list.
13 newlst = [None] * self.numItems * 2
14 for k in range(len(self.items)):
15 newlst[k] = self.items[k]
16
17 self.items = newlst
18
19 self.items[self.numItems] = item
20 self.numItems += 1
21
22 def main():
```

```
23 p = PyList()
24
25 for k in range(100):
26 p.append(k)
27
28 print(p.items)
29 print(p.numItems)
30 print(len(p.items))
31
32 if name == " main ":
33 main()
```

The claim is that, using this new PyList append method, a sequence of *n* append operations on a PyList object, starting with an empty list, takes O(n) time meaning that individual operations must not take longer than O(1) time. How can this be true? Whenever the list runs out of space a new list is allocated and all the old elements are copied to the new list. Clearly, copying *n* elements from one list to another takes longer than O(1) time. Understanding how append could exhibit O(1) complexity relies on computing the *amortized complexity* of the *append* operation. Technically, when the list size is doubled the complexity of *append* is O(n). But how often does that happen? The answer is *not that often*.

2.10.2 PROOF OF APPEND COMPLEXITY

The proof that the append method has O(1) complexity uses what is called the accounting method to find the amortized complexity of append. The accounting method stores up cyber dollars to pay for expensive operations later. The idea is that there must be *enough* cyber dollars to pay for any operation that is more expensive than the desired complexity.

Consider a sequence of *n* append operations on an initially empty list. Appending the first element to the list is done in O(1) time since there is space for the first item added to the list because one slot was initially allocated in the list. Storing a value in an already allocated slot takes O(1) time. However, according to the accounting method, we'll claim that the cost of doing the append operation requires an additional two cyber dollars. This is still O(1) complexity. Each time we run out of space we'll double the number of slots in the fixed size list. Allocating a fixed size list is a O(1) operation regardless of the list size. The extra work comes when copying the elements from the old list to the new list.

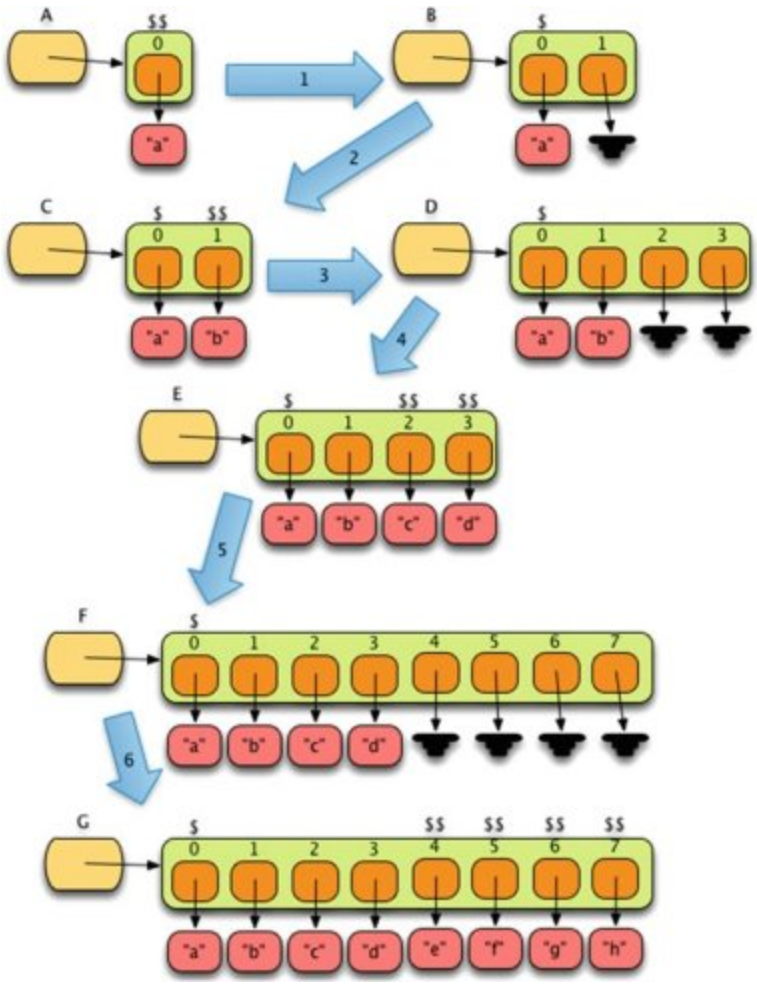

Fig. 2.7 Append Cyber Dollars

The first time we need to double the size is when the second append is called. There are two cyber dollars stored up at this point in time. One of them is needed when copying the one element stored in the old list to the new fixed size list capable of holding two elements. Transition one in Fig. 2.7 shows the two stored cyber dollars and the result after copying to the new list when moving from step A to step B.

When append is called on version B of the list the result is version C. At this point, three cyber dollars are stored to be used when doubling the list size to four locations. The first two are filled with the old contents of the list. Two of the three stored cyber dollars are used while copying these values to the new list. When the list of size four fills, two additional append operations have occurred, storing five cyber dollars. Four of these cyber dollars are used in the copy from step E to step F. Again, when the list of size eight fills in step G there are nine stored cyber dollars to be used in doubling the list size and copying the elements over.

But, what if we didn't double the size of the list each time. If we increased the list size by one half its previous size each time, we could still make this argument work if we stored four cyber dollars for each append operation. In fact, as long as the size of the list grows proportionally to its current size each time it is expanded this argument still works to prove that appending to a list is a O(1) operation when lists must be allocated with a fixed size.

As mentioned earlier, the Python list object is implemented in *C*. While Python provides an append operation, the *C* language can only allocate fixed size lists, called arrays in *C*. And yet, Python list objects can append objects in O(1) time as can be observed by experimentation or by analyzing the *C* code that implements Python list objects. The Python list *append* implementation achieves this by increasing the list size as described in this section when the fixed size array runs out of space to achieve an amortized complexity of O(1).

2.11 CHAPTER SUMMARY

This chapter covered some important topics related to the efficiency of algorithms. Efficiency is an important topic because even the fastest computers will not be able to solve problems in a reasonable amount of time if the programs that are written for them are inefficient. In fact, some problems can’t be solved in a reasonable amount of time no matter how the program is written. Nevertheless, it is important that we understand these issues of efficiency. Finding the complexity of a piece of code is an important skill that you will get better at the more you practice. Here are some of the things you should have learned in this chapter. You should:

- know the complexity of storing or retrieving a value from a list or the memory of the computer.

- know how memory is like a post office.

- know how memory is NOT like a post office.

- know how to use the datetime module to get information about the time it takes to complete an operation in a program.

- know how to write an XML file that can be used by the plotting program to plot information about the performance of an algorithm or piece of code.

- understand the definition of big-Oh notation and how it establishes an upper bound on the performance of a piece of code.

- understand why the list + operation is not as efficient as the *append* operation.

- understand the difference between O(n), $O(n^2)$, and other computational complex- ities and why those differences are important to us as computer programmers.

- Understand Theta notation and what an asymptotically tight bound says about an algorithm.

- Understand Amortized complexity and how to apply it in some simple situations.

2.12 REVIEW QUESTIONS

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. How is a list like a bunch of post office boxes?
2. How is accessing an element of a list NOT like retrieving the contents of a post office box?
3. How can you compute the amount of time it takes to complete an operation in a computer using Python?
4. In terms of computational complexity, which is better, an algorithm that is $O(n^2)$ or an algorithm that is $O(2^n)$?
5. Describe, in English, what it means for an algorithm to be $O(n^2)$.
6. When doing a proof by induction, what two parts are there to the proof?
7. If you had an algorithm with a loop that executed *n* steps the first time through, then $n - 2$ the second time, $n - 4$ the next time, and kept repeating until the last time through the loop it executed 2 steps, what would be the complex- ity measure of this loop? Justify your answer with what you learned in this chapter.
8. Assume you had a data set of size *n* and two algorithms that processed that data set in the same way. Algorithm A took 10 steps to process each item in the data set. Algorithm B processed each item in 100 steps. What would the complexity be of these two algorithms?
9. Explain why the *append* operation on a list is more efficient than the + operator.
10. Describe an algorithm for finding a particular value in a list. Then give the

computational complexity of this algorithm. You may make any assumptions you want, but you should state your assumptions along with your algorithm.

2.13 PROGRAMMING PROBLEMS

1. Devise an experiment to discover the complexity of comparing strings in Python. Does the size of the string affect the efficiency of the string comparison and if so, what is the complexity of the comparison? In this experiment you might want to consider a best case, worst case, and average case complexity. Write a program that produces an XML file with your results in the format specified in this chapter. Then use the PlotData.py program to visualize those results.
2. Conduct an experiment to prove that the product of two numbers does not depend on the size of the two numbers being multiplied. Write a program that plots the results of multiplying numbers of various sizes together. HINT: To get a good reading you may want to do more than one of these multiplications and time them as a group since a multiplication happens pretty quickly in a computer. Verify that it truly is a O(1) operation. Do you see any anomalies? It might be explained by Python's support of large integers. What is the cutoff point for handling multiplications in constant time? Why? Write a program that produces an XML file with your results in the format given in this chapter. Then visualize your results with the PlotData.py program provided in this chapter.
3. Write a program to gather experimental data about comparing integers. Compare integers of different sizes and plot the amount of time it takes to do those com- parisons. Plot your results by writing an XML file in the Ploy.py format. Is the comparison operation always O(1)? If not, can you theorize why? HINT: You may want to read about Python's support for large integers.
4. Write a short function that searches for a particular value in a list and returns the position of that value in the list (i.e. its index). Then write a program that times how long it takes to search for an item in lists of different sizes. The size of the list is your *n*. Gather results from this experiment and write them to an XML file in the PlotData.py format. What is the complexity of this algorithm? Answer this question in a

comment in your program and verify that the experimental results match your prediction. Then, compare this with the *index* method on a list. Which is more efficient in terms of computational complexity? HINT: You need to be careful to consider the average case for this problem, not just a trivial case.

5. Write a short function that given a list, adds together all the values in the list and returns the sum. Write your program so it does this operation with varying sizes of lists. Record the time it takes to find the sum for various list sizes. Record this information in an XML file in the PlotData.py format. What complexity is this algorithm? Answer this in a comment at the top of your program and verify it with your experimental data. Compare this data with the built-in *sum* function in Python that does the same thing. Which is more efficient in terms of computational complexity? HINT: You need to be careful to consider the average case for this problem, not just a trivial case.
6. Assume that you have a datatype called the Clearable type. This data type has a fixed size list inside it when it is created. So Clearable(10) would create a clearable list of size 10. Objects of the Clearable type should support an append operation and a lookup operation. The lookup operation is called getitem (item). If *cl* is

a Clearable list, then writing *cl*[*item*] will return the item if it is in the list and return *None* otherwise. Writing *cl*[*item*] results in a method call of cl. getitem (item). Unlike the append operation described in Sect. 2.10.1, when the Clearable object fills up the list is automatically cleared or emptied on the next call to append by setting all elements of the list back to *None*. The Clearable object should always keep track of the number of values currently stored in the object. Form a theory about the complexity of the append operation on this datatype. Then write a test program to test the Clearable object on different initial sizes and numbers of append operations. Create one sequence for each different initial size of the Clearable datatype and write your results in the plot format described in this chapter. Then comment on how your theory holds up or does not hold up given your experimentation results.

RECURSION 3

Don’t think too hard! That’s one of the central themes of this chapter. It’s not often that you tell computer programmers not to think too hard, but this is one time when it is appropriate. You need to read this chapter if you have not written recursive functions before. Most computer science students start by learning to program in a style called *imperative* programming. This simply means that you are likely used to thinking about creating variables, storing values, and updating those values as a program proceeds. In this chapter you are going to begin learning a different style of programming called *functional* programming. When you program in the func- tional style, you think much more about the definition of *what* you are programming than *how* you are going to program it. Some say that writing recursive functions is a *declarative* approach rather than an *imperative* approach. You’ll start to learn what that means for you very soon. When you start to get good at writing recursive functions you’ll be surprised how easy it can be!

Python programs are executed by an interpreter. An interpreter is a program that reads another program as its input and does what it says. The Python interpreter, usually called *python*, was written in a language called C. That C program reads a Python program and does what the Python program says to do in its statements. An interpreter interprets a program by running or executing what is written within it. The interpreter interacts with the operating system of the computer to use the network, the keyboard, the mouse, the monitor, the hard drive, and any other I/O device that it needs to complete the work that is described in the program it is interpreting. The picture in Fig. 3.1 shows you how all these pieces fit together.

In this chapter we'll introduce you to scope, the run-time stack, and the heap so you understand how the interpreter calls functions and where local variables are stored. Then we'll provide several examples of recursive functions so you can begin to see how they are written. There will be a number of recursive functions for you to practice writing and we'll apply recursion to drawing pictures as well.

One thing you will not do in the homework for this chapter is write code that uses a *for* loop or a *while* loop. If you find yourself trying to write code that uses either kind of loop you are trying to write a function imperatively rather than functionally.

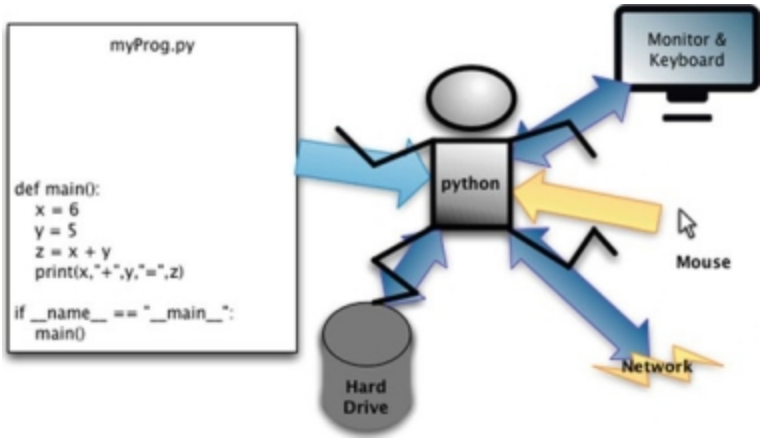

Fig. 3.1 The Python Interpreter

Recursion is the way we will repeat code in this chapter. A recursive function has no need for a *for* or *while* loop.

3.1 CHAPTER GOALS

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

- How does Python determine the meaning of an identifier in a program?
- What happens to the run-time stack when a function is called?
- What happens to the run-time stack when a function returns from a call?
- What are the two important parts to a recursive function and which part comes first?
- Exactly what happens when a return statement is executed?
- Why should we write recursive functions?
- What are the computational complexities of various recursive functions?

You should also be able to write some simple recursive functions yourself without thinking too hard about how they work. In addition, you should be able to use a debugger to examine the contents of the run-time stack for a recursive function.

3.2 SCOPE

To form a complete mental picture of how your programs work we should further explore just how the Python interpreter executes a Python program. In the first chapter we explored how references are the things which we name and that references point to objects, which are unnamed. However, we sometimes call an object by the name of the reference that is pointing at it. For instance, if we write:

```
x = 6
```

it means that *x* is a reference that points to an object with a *6* inside it. But sometimes we are careless and just say that *x equals 6*. It is important that you understand that even when we say things like *x equals 6* what we really mean is that *x is a reference that points to an object that contains 6*. You can see why we are careless sometimes. It takes too many words to say what we really mean and as long as everyone understands that references have names and objects are pointed to by references, then we can save the words. The rest of this text will make this assumption at times. When it is really important, we'll make sure we distinguish between references and objects.

Part of our mental picture must include *Scope* in a Python program. Scope refers to a part of a program where a collection of identifiers are visible. Let's look at a simple example program.

3.2.1 LOCAL SCOPE

Consider the code in Fig. 3.2. In this program there are several scopes. Every colored region of the figure delimits one of those scopes. While executing line 23 of the program in Fig. 3.2 the light green region is called the *Local* scope. The local scope is the scope of the function that the computer is currently executing. When your program is executing a line of code, the scope that surrounds that line of code is called the local scope. When you reference an identifier in a statement in your program, Python first examines the local scope to see if the identifier is defined there, within the local scope. An identifier, *id*, is defined under one of three conditions.

- A statement like *id* = ... appears somewhere within the current scope. In this case

id would be a reference to an object in the local scope.

- *id* appears as a parameter name of the function in the current scope. In this case *id* would be a reference to an object that was passed to the current function as an argument.

- *id* appears as a name of a function or class through the use of a function *def* or

class definition within the current scope.

While Python is executing line 23 in Fig. 3.2, the reference *val* is defined within its local scope. If Python finds *id* in the local scope, it looks up the corresponding value and retrieves it. This is what happens when *val* is encountered on line 23. The object that is referenced by *val* is retrieved and returned.

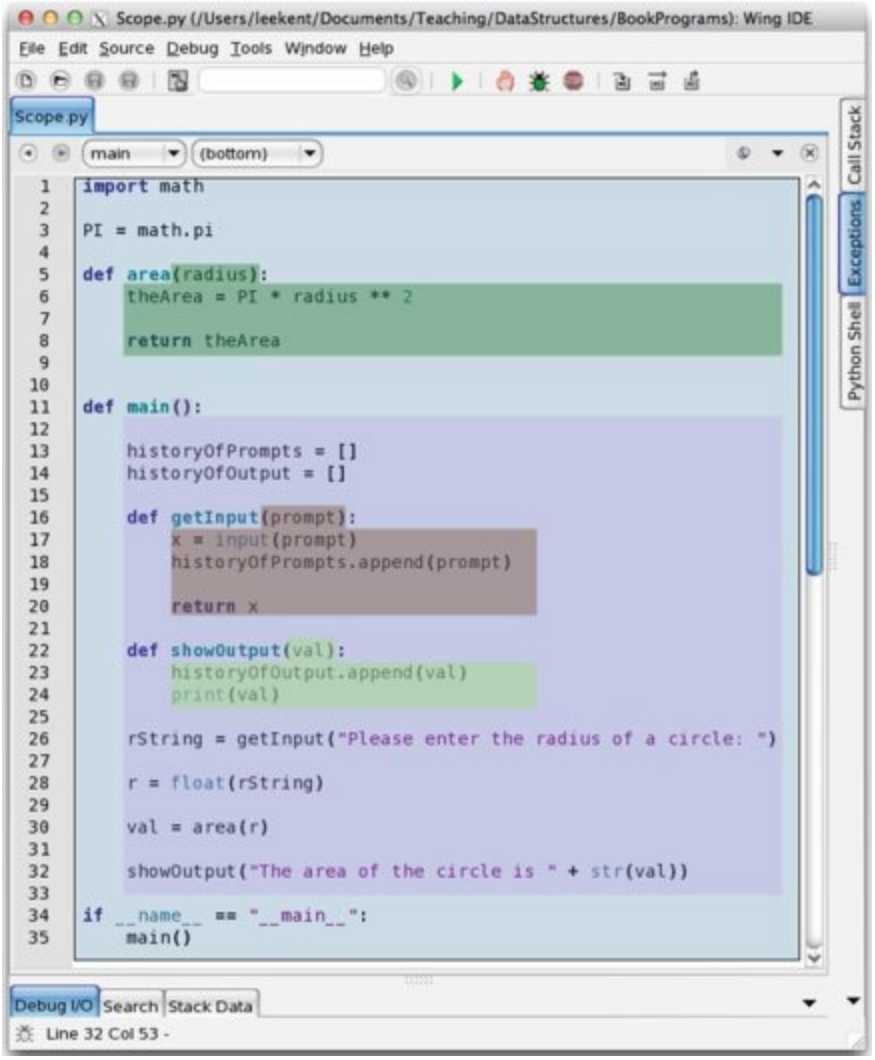

Fig. 3.2 Scopes within a Simple Program

3.2.2 ENCLOSING SCOPE

If Python does not find the reference *id* within the local scope, it will examine the *Enclosing* scope to see if it can find *id* there. In the program in Fig. 3.2, while Python is executing the statement on line 23, the enclosing scope is the purple region of the program. The identifiers defined in this enclosing scope include *historyOfPrompts*, *historyOfOutput*, *rString*, *r*, *val*, *getInput*, and *showInput*. Notice that function names are included as identifiers. Again, Python looks for the identifier using the same

conditions as defined in Sect. 3.2.1 for the local scope. The identifier must be defined using *id* = ..., it must be a parameter to the enclosing function, or it must be an identifier for a class or function definition in the enclosing scope's function. On line 23, when Python encounters the identifier *historyOfOutput* it finds that identifier

defined in the enclosing scope and retrieves it for use in the call to the append method.

Which scope is local depends on where your program is currently executing. When executing line 23, the light green region is the local scope. When executing line 18 the brown region is the local scope. When executing line 14 or line 26 the purple region is the local scope. When executing line 6 the darker green region is the local scope. Finally, when executing line 1 or 3 the blue region is the local scope. The local scope is determined by where your program is currently executing.

Scopes are nested. This means that each scope is nested inside another scope. The final enclosing scope of a module is the module itself. Each module has its own scope. The blue region of Fig. 3.2 corresponds to the module scope. Identifiers that are defined outside of any other functions, but inside the module, are at the module level. The reference *PI* in Fig. 3.2 is defined at the module level. The functions *area* and *main* are also defined at the module level scope.

While executing line 23 of the program in Fig. 3.2 the identifier *val* is defined in the local scope. But, *val* is also defined in the enclosing scope. This is acceptable and often happens in Python programs. Each scope has its own copy of identifiers. The choice of which *val* is visible is made by always selecting the innermost scope that defines the identifier. While executing line 23 of the program in Fig. 3.2 the *val* in the local scope is visible and the *val* in the enclosing scope is hidden. This is why it is important that we choose our variable names and identifiers carefully in our

programs. If we use an identifier that is already defined in an outer scope, we will no longer be able to access it from an inner scope where the same identifier is defined.

It is relatively easy to determine all the nested scopes within a module. Every function definition (including the definition of methods) within a module defines a different scope. The scope never includes the function name itself, but includes its parameters and the body of the function. You can follow this pattern to mentally draw boxes around any scope so you know where it begins and ends in your code.

3.2.3 GLOBAL SCOPE

Using Python it is possible to define variables at the *Global* level. Generally this is a bad programming practice and we will not do this in this text. If interested you can read more about global variables in Python online. But, using too many global variables will generally lead to name conflicts and will likely lead to unwanted side effects. Poor use of global variables contributes to spaghetti code which is named for the big mess you would have trying to untangle it to figure out what it does.

3.2.4 BUILT-IN SCOPE

The final scope in Python is the *Built-In* scope. If an identifier is not found within any of the nested scopes within a module and it is not defined in the global scope, then Python will examine the built-in identifiers to see if it is defined there. For instance, consider the identifier *int*. If you were to write the following:

```
x = int("6")
```

Python would first look in the local scope to see if *int* were defined as a function or variable within that local scope. If *int* is not found within the local scope, Python would look in all the enclosing scopes starting with the next inner-most local scope and working outwards from there. If not found in any of the enclosing scopes, Python would then look in the global scope for the *int* identifier. If not found there, then Python would consult the *Built-In* scope, where it would find the *int* class or type.

With this explanation, it should now be clear why you should not use identifiers that already exist in the built-in scope. If you use *int* as an identifier you will not be able to use the *int* from the built-in scope because Python will find *int* in a local or enclosing scope first.

3.2.5 LEGB

Mark Lutz, in his book *Learning Python* [6], described the rules of scope in Python programs using the LEGB acronym. This acronym, standing for *Local*, *Enclosing*, *Global*, and *Built-In* can help you memorize the rules of scope in Python. The order of the letters in the acronym is important. When the Python interpreter encounters an identifier in a program, it searches the local scope first, followed by all the enclosing scopes from the inside outward, followed by the global scope, and finally the built-in scope.

3.3 THE RUN-TIME STACK AND THE HEAP

As we learned in the last section, the parameters and body of each function define a scope within a Python program. The parameters and variables defined within the local scope of a function must be stored someplace within the RAM of a computer. Python splits the RAM up into two parts called the *Run-time Stack* and the *Heap*.

The run-time stack is like a stack of trays in a cafeteria. Most cafeterias have a device that holds these trays. When the stack of trays gets short enough a spring below the trays pops the trays up so they are at a nice height. As more trays are added to the stack, the spring in this device compresses and the stack pushes down. A *Stack* in Computer Science is similar in many ways to this kind of device. The run-time stack is a stack of *Activation Records*. The Python interpreter *pushes* an activation

record onto the run-time stack when a function is called. When a function returns the Python interpreter *pops* the corresponding activation record off the run-time stack.

Python stores the identifiers defined in the local scope in an activation record. When a function is called, a new scope becomes the local scope. At the same time a new activation record is pushed onto the run-time stack. This new activation record holds all the variables that are defined within the new local scope. When a function returns its corresponding activation record is popped from the run-time stack.

The *Heap* is the area of RAM where all objects are stored. When an object is created it resides in the heap. The run-time stack never contains objects. References to objects are stored within the run-time stack and those references point to objects in the heap.

Consider the program in Fig. 3.2. When the Python interpreter is executing lines 23 and 24 of the program, the run-time stack looks as it does in Fig. 3.3. There are three activation records on the run-time stack. The first activation record pushed onto the run-time stack was for the *module*. When the module first began executing, the Python interpreter went through the module from top to bottom and put any variable definitions in the module scope into the activation record for the module. In this program that consisted of the reference *PI* to the value 3.14159.

Then, at the end of the module the *if* statement called the *main* function. This caused the Python interpreter to push the activation record for the main function. The variables defined within the main function include *historyOfPrompts*, *history- OfOutput*, *rString*, *r*, and *val*. Each of these appear within the activation record for the main function.

As the main function began executing it called the *getInput* function. When that call occurred there was an activation record pushed for the function call. That acti- vation record contained the *prompt* and *x* variables. This activation record does not appear in the figure because by the time we execute line 23 and 24 of the program the Python interpreter has already returned from the *getInput* function. When the interpreter returned from the function call the corresponding activation record was popped from the run-time stack.

Finally, the program calls the *showOutput* function on line 26 and execution of the function begins. An activation record for the *showOutpout* function call was pushed onto the run-time stack when *showOutput* was called. The references local to that scope, which includes just the *val* variable, were stored the activation record for this function call.

You can run this example program using Wing or some other IDE. The code for it appears in Sect. 20.2. When you use the Wing IDE to run this program you can stop the program at any point and examine the run-time stack. For instance, Fig. 3.4 shows Wing in the midst of running this program. A breakpoint has been set on line 24 to stop the program. The tab at the bottom of the Wing IDE window shows the *Stack Data*. This is the run-time stack.

Right below the *Stack Data* tab there is a combination box that currently displays *showOutput*(): *Scope.py, line 24*. This combo box lets you pick from the activation record that is currently being displayed. If you pick a different activation record, its contents will be displayed directly below it in the Wing IDE.

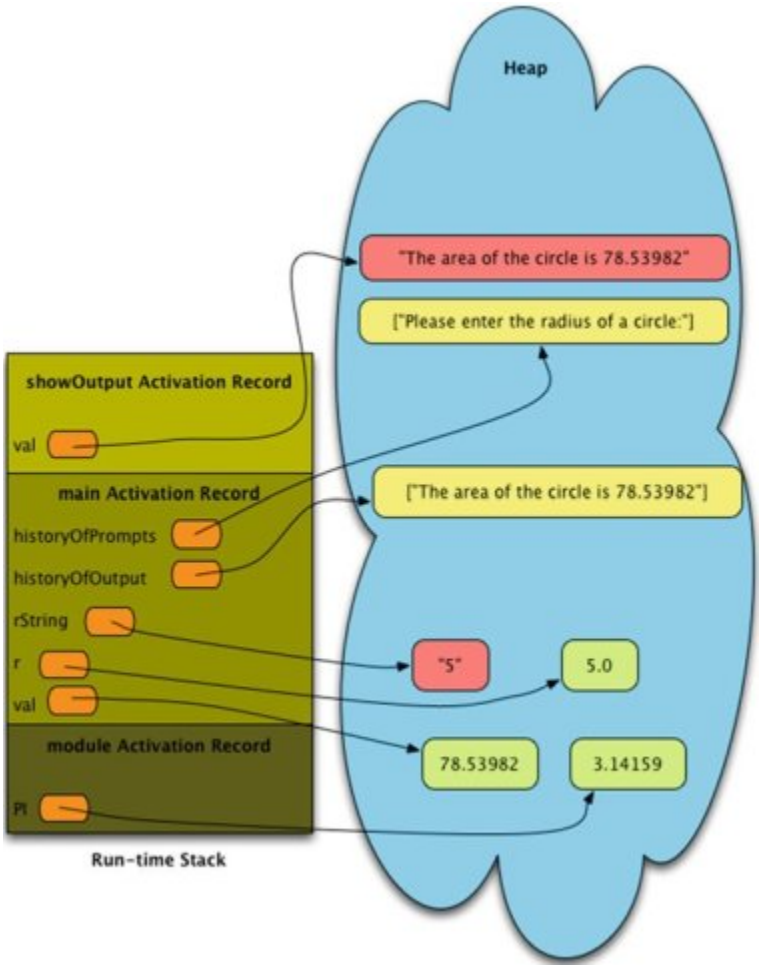

Fig. 3.3 The Run-time Stack and the Heap

One important note should be made here. Figure 3.4 shows *historyOfOutput* as a local variable in the *showOutput* function. This is not really the case, because the *historyOfOutput* reference is not defined within the local scope of the *showOutput* function. However, due to the way Python is implemented the reference for this vari- able shows up in the activation record for *showOutput* because it is being referenced from this scope. But, the reference to *historyOfOutput* in the activation record for *showOutput* and the reference called *historyOfOutput* in the *main* activation record point at the same object so no real harm is done. The important thing to note is

that the Wing IDE is correct in showing the *historyOfOutput* variable as a local variable in this activation record since this is a reflection of Python's implementation and not due to a bug in Wing IDE 101.

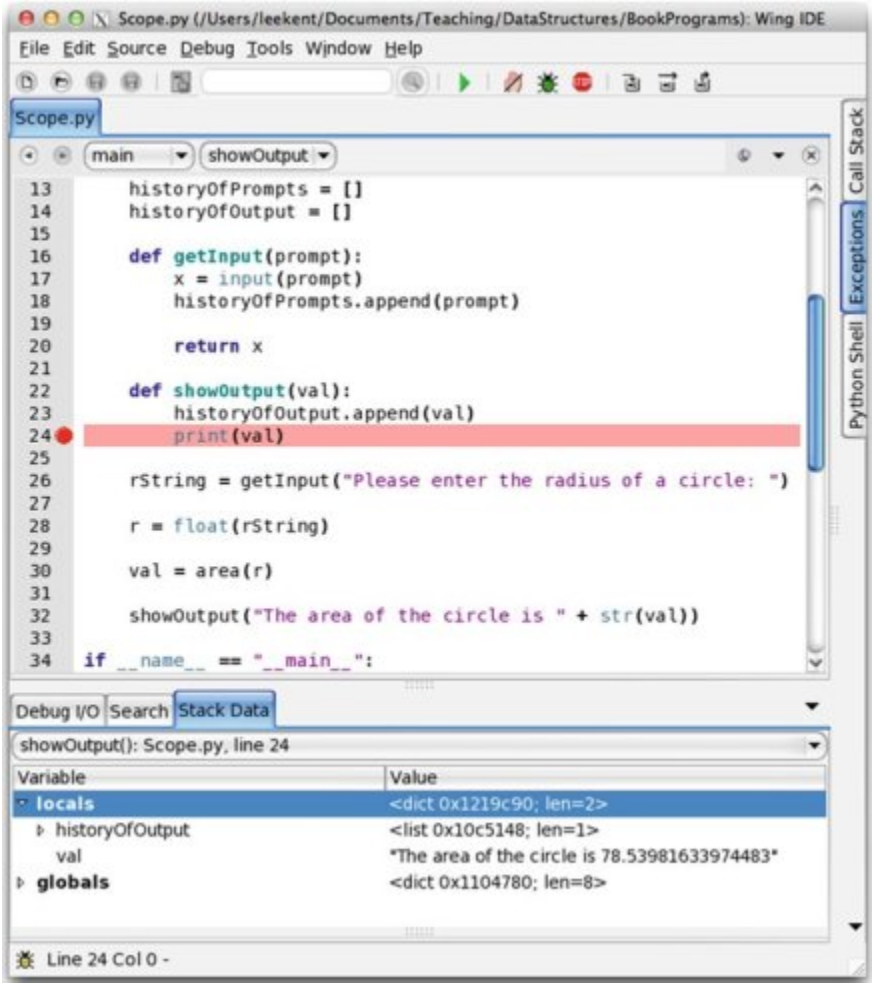

Fig. 3.4 The Wing IDE Showing the Run-time Stack

3.4 WRITING A RECURSIVE FUNCTION

A recursive function is simply a function that calls itself. It’s really very simple to write a recursive function, but of course you want to write recursive functions that actually do something interesting. In addition, if a function just kept calling itself it would never finish. Actually, it would finish when run on a computer because we just learned that every time you call a function, an activation record is pushed on the run-time stack. If a recursive function continues to call itself over and over it will

eventually fill up the run-time stack and you will get a stack overflow error when running such a program.

To prevent a recursive function from running forever, or overflowing the run- time stack, every recursive function must have a base case, just like an inductive proof must have a base case. There are many similarities between inductive proofs and recursive functions. The base case in a recursive function must be written first, before the function is called recursively.

Now, wrapping your head around just how a recursive function works is a little difficult at first. Actually, understanding *how* a recursive function works isn't all that important. When writing recursive functions we want to think more about *what* it does than *how* it works. It doesn't pay to think too hard about *how* recursive functions work, but in fact even that will get much easier with some practice.

When writing a recursive function there are four rules that you adhere to. These rules are not negotiable and will ensure that your recursive function will eventually finish. If you memorize and learn to follow these rules you will be writing recursive functions in no time. The rules are:

1. Decide on the name of your function and the arguments that must be passed to it to complete its work as well as what value the function should return.

2. Write the base case for your recursive function first. The base case is an *if* state- ment that handles a very simple case in the recursive function by returning a value.

3. Finally, you must call the function recursively with an argument or arguments that are smaller in some way than the parameters that were passed to the function when the last call was made. The argument or

arguments that get smaller are the same argument or arguments you examined in your base case.

4. Look at a concrete example. Pick some values to try out with your recursive function. Trust that the recursive call you made in the last step works. Take the result from that recursive call and use it to form the result you want your function to return. Use the concrete example to help you see how to form that result.

We'll do a very simple example to begin with. In the last chapter we proved the following.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

So, if we wanted to compute the sum of the first *n* integers, we could write a Python program as shown in Sect. 3.4.1.

3.4.1 SUM OF INTEGERS

```
1 def sumFirstN(n):
2 return n * (n+1) // 2
3
4 def main():
```

```
5 x = int(input("Please enter a non-negative integer: "))

6

7 s = sumFirstN(x)

8

9 print("The sum of the first", x, "integers is", str(s)+".")

10

11 if  name  == " main ":

12 main()
```

In this case, this would be the best function we could write because the complexity of the *sumFirstN* function is O(1). This means the time it takes to execute this function is not dependent on the size of the data, *n*. However, to illustrate a recursive function, let's go back to the definition of summation. The definition for summation has two parts. First, the base case of the definition.

$$0$$

$$i = 0$$

$$i = 1$$

The recursive part of the definition is as follows. This is what we call a recursive definition because it is defined in terms of itself. Notice that the recursive definition is defined in terms of a smaller *n*, in this case $n - 1$. The summation to $n - 1$ is our recursive call and it will work. If we want to compute the sum of the first 5 integers, then the recursive call computes 1 + 2 + 3 + 4 to give us 10. Adding *n* will give use

15, the result we want.

$$i =$$

$$i = 1$$

$$N - 1$$

$$i + n$$

$$i = 1$$

The two parts of this recursive definition can be translated directly into a recursive function in Python. The recursive definition is given in Sect. 3.4.2.

3.4.2 RECURSIVE SUM OF INTEGERS

```
1 def recSumFirstN(n):
2 if n == 0:
3 return 0
4 else:
5 return recSumFirstN(n-1) + n
6
7 def main():
8 x = int(input("Please enter a non-negative integer: "))
9
10 s = recSumFirstN(x)
11
12 print("The sum of the first", x, "integers is", str(s)+".")
13
14 if  name  == " main ":
15 main()
```

The *recSumFirstN* function in the code of Sect. 3.4.2 is recursive. It calls itself with a smaller value and it has a base case that comes first, so it is well-formed. There is

one thing that we might point out in this recursive function. The *else* is not necessary. When the Python interpreter encounters a **return** statement, the interpreter returns immediately and does not execute the rest of the function. So, in Sect. 3.4.2, if the function returns 0 in the *then* part of the *if* statement, the rest of the function is not executed. If *n* is not zero, then we want to execute the code on the *else* statement. This means we could rewrite this function as shown in Sect. 3.4.3.

3.4.3 NO ELSE NEEDED

```
1 def recSumFirstN(n):
2 if n == 0:
3 return 0
4
5 return recSumFirstN(n-1) + n
```

The format of the code in Sect. 3.4.3 is a common way to write recursive functions. Sometimes a recursive function has more than one base case. Each base case can be handled by an *if* statement with a return in it. The recursive case does not need to be in an else when all base cases result in a return. The recursive case comes last in the recursive function definition.

3.5 TRACING THE EXECUTION OF A RECURSIVE FUNCTION

Early in this chapter you were given the mandate “Don’t think too hard” when writing a recursive function. Understanding exactly *how* a recursive function works may be a bit difficult when you are first learning about them. It may help to follow the execution of a recursive function in an example. Consider the program in the previous section. Let’s assume that the user entered the integer 4 at the keyboard. When this program begins running it will have an activation record on the run-time stack for the *module* and the *main* function.

When the program gets to line 10 in the code of Sect. 3.4.2, where the *recSumFirstN* function is first called, a new activation record will be pushed for the function call, resulting in three activation records on the run-time stack. The Python interpreter then jumps to line 2 with *n* pointing at the number 4 as shown in the picture of Fig. 3.5. Execution of the function proceeds. The value of *n* is not zero, so Python executes line 5 where there is another function call to *recSumFirstN*. This causes the Python interpreter to push another activation record on the run-time stack and the interpreter jumps to line 2 again. This time the value of *n* is 3. But again, this is not zero, so line 5 is executed and another activation record is pushed with a new value of 2 for *n*. This repeats two more times for values of 1 and 0 for *n*.

The important thing to note in this program execution is that there is one copy of the variable *n* for each recursive function call. An activation record holds the local variables and parameters of all variables that are in the local scope of the function.

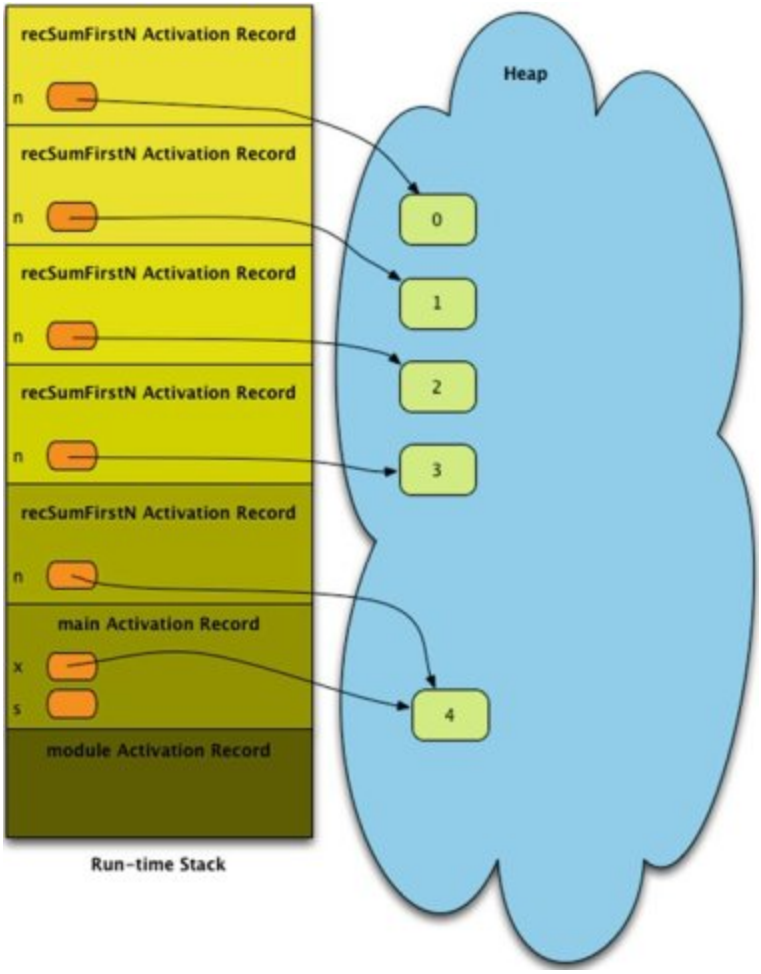

Fig. 3.5 The Run-time Stack of a Recursive Function Call

Each time the function is called a new activation record is pushed and a new copy of the local variables is stored within the activation record. The picture in Fig. 3.5 depicts the run-time stack at its deepest point.

When execution of the function gets to the point when *n* equals 0, the Python interpreter finds that *n* equals 0 on line 2 of the code. It is at this point that the *sumFirstN* function returns its first value. It returns 0 to the previous function call where *n* was 1. The return occurs on line 5 of the code. The activation record for the function call when *n* was 0 is popped

from the run-time stack. This is depicted in Fig. 3.6 by the shading of the activation record in the figure. When the function returns the space for the activation record is reclaimed for use later. The shaded

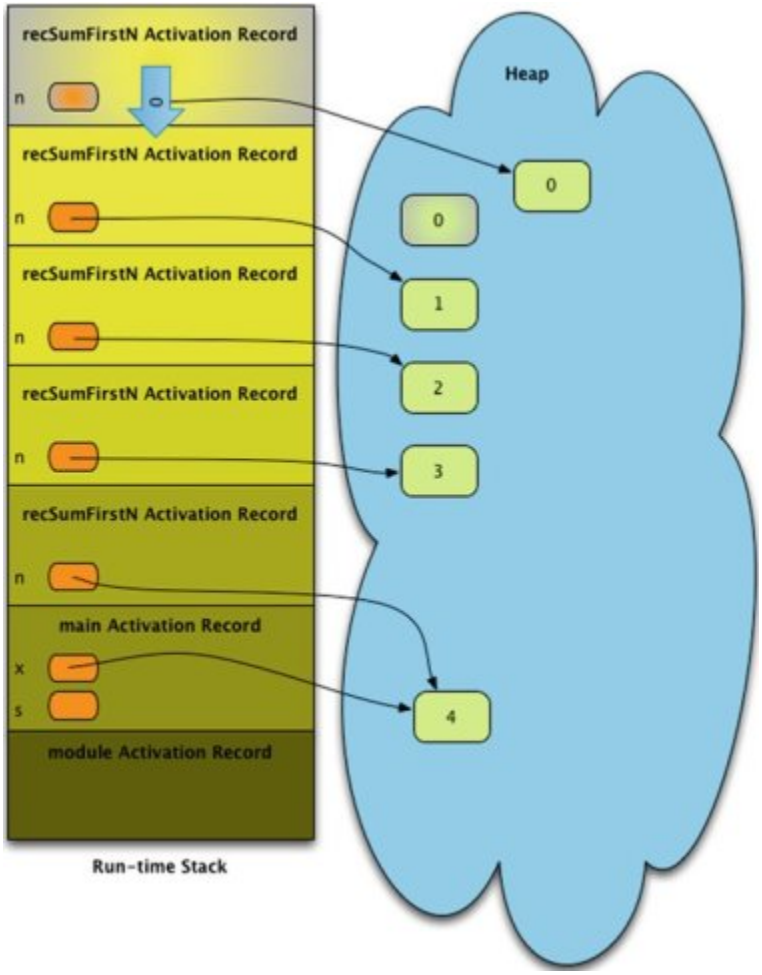

Fig. 3.6 The First Return from recSumFirstN

object containing 0 on the heap is also reclaimed by the garbage collector because there are no references pointing at it anymore.

After the first return of the *RecSumFirstN*, the Python interpreter returns to line 5 in the previous function call. But, this statement contains a return statement as well. So, the function returns again. Again, it returns to line 5, but this time with a value of 1. The function returns again, but with a value of 3 this time. Again, since it returned to line 5, the function returns again with a value of 6. Finally, once again the function returns, this time with a

value of 10. But this time the *recSumFirstN* function returns to line 10 of the main function where *s* is made to point to the value of 10. This is depicted in Fig. 3.7.

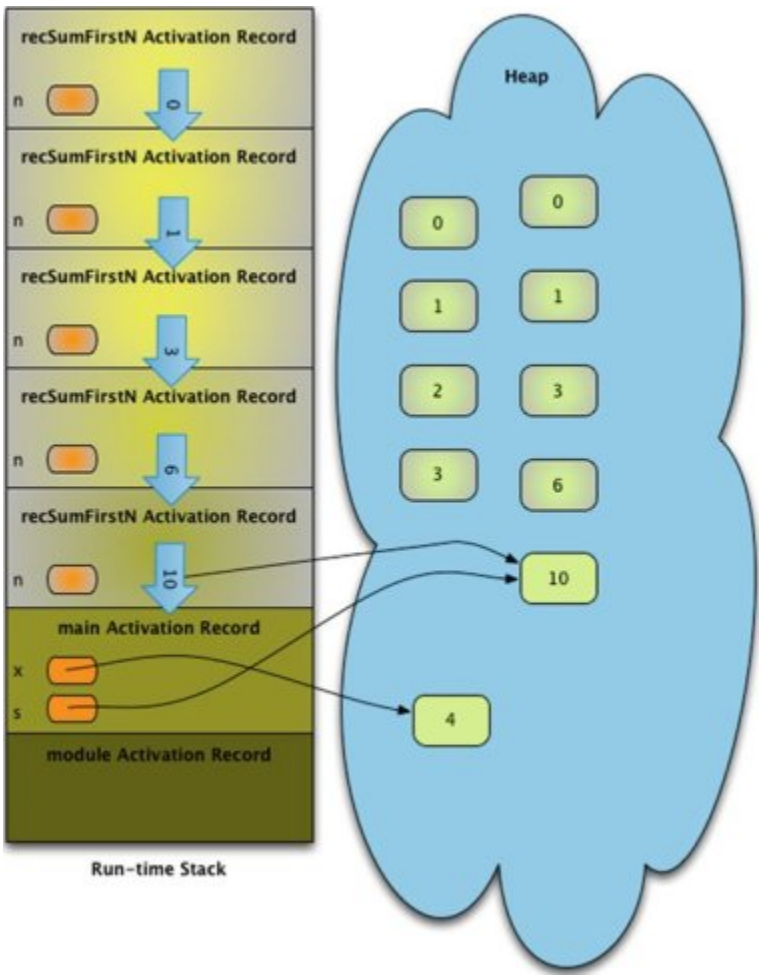

Fig. 3.7 The Last Return from recSumFirstN

The program terminates after printing the 10 to the screen and returning from the *main* function after line 12 and from the *module* after line 15. The importance of this example is to illustrate that each recursive call to *recSumFirstN* has its own copy of the variable *n* because it is local to the scope of the *recSumFirstN* function. Each time the function is called, the local variables and parameters are copied into the corresponding activation record. When a function call returns, the corresponding activation record is popped off the run-time stack. This is how a recursive function is executed.

3.6 RECURSION IN COMPUTER GRAPHICS

Recursion can be applied to lots of different problems including sorting, searching, drawing pictures, etc. The program given in Sect. 3.6.1 draws a spiral on the screen as shown in Fig. 3.8.

3.6.1 RECURSIVE SPIRAL

```
1 import turtle
2
3 def drawSpiral(t, length, color, colorBase):
4     #color is a 24 bit value that is changing a bit
5     #each time for a nice color effect
6     if length == 0:
7         return
8
9     # add 2^10 to the old color modulo 2^24
10    # the modulo 2^24 prevents the color from
11    # getting too big.
12    newcolor = (int(color[1:],16) + 2**10)%(2**24)
13
14    # find the color base integer value
15    base = int(colorBase[1:],16)
16
17    # now if the new color is less than the base
18    # add the base modulo 2^24.
19    if newcolor < base:
```

```
20      newcolor = (newcolor + base)%(2**24)
21
22      # let newcolor be the hex string after conversion.
23      newcolor = hex(newcolor)[2:]
24
25      # add a pound sign and zeroes to the front so it
26      # is 6 characters long plus the pound sign for a
27      # proper color string.
28      newcolor = "#"+("0"*(6-len(newcolor)))+newcolor
29
30      t.color(newcolor)
31      t.forward(length)
32      t.left(90)
33
34      drawSpiral(t, length-1, newcolor, colorBase)
35
36 def main():
37      t = turtle.Turtle()
38      screen = t.getscreen()
39      t.speed(100)
40      t.penup()
41      t.goto(-100,-100)
```

```
42      t.pendown()
43
44      drawSpiral(t, 200, "#000000", "#ff00ff")
45
46      screen.exitonclick()
47
```