



Igor Vishnevskiy

Java to Python

**Python Learning Guide
for Java Programmers**

Java to Python by Igor Vishnevskiy

Table of Contents

1. A FIRST SIMPLE PROGRAM

2. COMPILING A PROGRAM

3. VARIABLES

4. CONTROL STATEMENTS AND LOOPS

4.1 *if* Statements

4.2 Nested *if* Statements

4.3 *for* Loop Statements

4.4 *loop continue* Statements

4.5 *loop pass* Statements

4.6 *while* Loop Statements

4.7 *do-while* Loop Statements

4.8 *switch* Statements

5. OPERATORS

5.1 Basic Arithmetic Operators

5.2 Relational and Boolean Logical Operators

5.3 Ternary Operator

6. CLASSES

6.1 Class Variables VS Instance Variables

6.2 Local Variables

6.3 Accessing Methods of a Class

6.4 Constructor

6.5 Nested Classes

6.6 Returning from Method

6.7 The *finalize()* Method

6.8 Overloading Methods

6.9 Overloading Constructor

6.10 Using Objects as Parameters

6.11 Recursion

6.12 Access Control

6.13 Static Methods

6.14 Final

6.15 Command-Line Arguments

- 6.16 Variable-Length Arguments**
- 6.17 Inheritance**
- 6.18 Abstract Classes**
- 6.19 Importing Classes into a Workspace**
- 6.20 Exception Handling**
- 6.21 Throwing Custom Exception**
- 6.22 @Decorators are not @Annotations**

7. DATA STRUCTURES

- 7.1 Arrays**
- 7.2 Accessing Values in Arrays**
- 7.3 Updating Values in Arrays**
- 7.4 Getting the Size of an Array**
- 7.5 Sorting Arrays**
- 7.6 Counting Values in an Array**
- 7.7 Inserting a Value under a Certain Index in an Array**
- 7.8 Return the Index of an Element in an Array**
- 7.9 Difference between Appending and Extending to an Array**
- 7.10 Deleting Elements from an Array by Index**
- 7.11 Lists in Python Can Hold Anything**
- 7.12 Multidimensional Arrays**
- 7.13 List Comprehension**
- 7.14 Tuple**
- 7.15 Python's Dictionary Is Java's Map**
- 7.16 Update an Existing Entry in a Dictionary**
- 7.17 Add a New Entry to an Existing Dictionary**
- 7.18 Emptying a Dictionary (Removing All Its Values)**
- 7.19 Delete an Entire Dictionary**
- 7.20 Get the Size of a Dictionary**
- 7.21 Getting Keys and Values of a Dictionary**
- 7.22 Python Dictionary vs. JSON.**
- 7.23 Sets**
- 7.24 Frozen Set**
- 7.25 Stacks**
- 7.26 Queue**
- 7.27 Linked List**
- 7.28 Binary Trees**
- 7.29 Graphs**

8. MULTITHREADING AND MULTIPROCESSING

- 8.1 Multithreading**

- 8.2 Synchronization in Multithreading**
- 8.3 Multiprocessing**
- 8.4 Synchronization in Multiprocessing**

9. I/O

- 9.1 Reading Console Input**
- 9.2 Reading from Files**
- 9.3 How to Avoid Slurping While Reading Content of Large Files**
- 9.4 Writing into Files**
- 9.5 Appending into Files**
- 9.6 Checking Path Existence**
- 9.7 Creating a Path to File**
- 9.8 Reading JSON Files**
- 9.9 Writing JSON Files**
- 9.10 Reading CSV Files**
- 9.11 Writing CSV Files**
- 9.12 Lambda Expressions**

10. STRINGS

- 10.1 String Concatenation with Other Data Types**
- 10.2 Character Extraction**
- 10.3 String Comparison**
- 10.4 *StartsWith()* and *EndsWith()***
- 10.5 Searching Strings**
- 10.6 String Replace**
- 10.7 String *trim()* in Python**
- 10.8 Changing the Case of Characters**
- 10.9 Joining Strings**
- 10.10 String Length**
- 10.11 Reverse a String**

11. SORTING AND SEARCHING ALGORITHMS

- 11.1 Insertion Sort**
- 11.2 Bubble Sort**
- 11.3 Selection Sort**

12. PYTHON'S MODULES, JAVA'S LIBRARIES

- 12.1 Installing Required Modules**
- 12.2 Packaging-Required Modules with Your Project**

13. RUNNING SHELL COMMANDS FROM PYTHON

- 13.1 Running a Single Shell Command**

13.2 Running Multiple Shell Commands as Array

14. QUERYING DATABASES

14.1 SQLite3

14.2 MySQL

14.3 Oracle

15. BUILDING STAND-ALONE APPLICATIONS WITH PYTHON

15.1 PyInstaller

15.2 py2app

15.3 py2exe

16. BUILDING WEBSITES WITH PYTHON

16.1 Django

16.2 Flask

16.3 Pyramid

FROM THE AUTHOR

Python is much like Java and at times even looks simpler. But Python is just as powerful as Java. If Java is the heavy metal of computer programming, then Python is the jazz that opens doors of freedom in software development. Both Java and Python are object-oriented programming languages. Both support Java's famous features such as encapsulation, inheritance and polymorphism. Both can be used to develop desktop and web-based applications. Both are multi-platform and run on all major platforms such as Linux, MS Windows, and Mac OS. Both support graphical user interface development.

Of course, there are also differences between Java and Python. For example, Java programs must be compiled, but in Python you have a choice of compiling your programs into stand-alone applications or running them as interpreted scripts or programs launched by a command from the Command Prompt. There are many other similarities and differences between these two languages, and those similarities make it a lot easier than you might think to learn Python, if you already know Java.

While learning Python myself, I realized how fast and easy it was to understand and pick up Python's syntax when I started converting Java's programming problems into Python. I had already known Java and worked with it professionally for some time, but I found myself having to learn Python fast to advance in my career. It motivated me to find a way to harness my existing knowing to speed up the process of learning a new language. This book is essentially a systematic presentation of the learning process I documented in learning Python using knowledge of Java.

For the engineer who is already proficient in Java, it would be a waste of time to study a

Python textbook that begins with the basic concept of object-oriented programming, since the concept of OOP software development is identical in all languages. The differences from one language to another are in their syntax. Syntax is best learned by using examples of the programming language that the engineer already knows. That's exactly is the learning model of this book.

This book is for those who are already comfortable with developing using Java programming language and therefore assumes knowledge of Java. Designed for Java engineers who want to learn Python, this book walks you through the differences between Java 8 and Python 2.7 syntax using examples from both languages. Specifically, the book will demonstrate how to perform the same procedures in Java and Python. For each procedure, the class names, method names, and variable names are kept consistent between Java and Python examples. This way you can see clearly the differences in syntax between the two languages. Using this approach, you will be up to speed with Python in no time.

1. A FIRST SIMPLE PROGRAM

Right from the first chapter, I will start with an example of a simple program coded in both Java and Python. Throughout this book you will see many similar examples that demonstrate exactly how differently or similarly the procedures are in these two languages.

Take a look at the following simple Java program. It is very straightforward and you know exactly what it does. Then take a look at the Python version of the same program and read the explanation that follows. You will notice that both examples have exactly the same class names and method names. This will be the structure for all examples throughout this book—all examples, in both Java and Python, will have the same class names, method names, and variable names. This is done so you could clearly see the differences in syntax between Java and Python and pick up Python efficiently.

Let's start conquering Python with Example 1.1.

EXAMPLE 1.1

Java:

/*

This is a simple Java program. (File name: *SimpleProgram.java*)

*/

```
class SimpleProgram {  
    public void main(String args[]) {  
        printHelloWorld();  
    }  
  
    public static void printHelloWorld(){  
        System.out.println("Hello World");  
    }  
}
```



```
// End of the Java program
```

In Python, the same program would be like so:

Python:

```
# This is a simple Java program. (File name: pythonAnything.py)
```

```
class SimpleProgram:
    def __init__(self):
        self.printHelloWorld()

    def printHelloWorld(self):
        print "Hello World\n"
```

```
run = SimpleProgram()
```

```
# End of the Python program
```

The example 1.1 is the same program coded in two languages. As you can see right away, the program starts with comments. In Java, comments are created by placing double forward slash in front of the text (such as *//comments*) or slash asterisk asterisk slash that surround the text (such as */*comments*/*). In Python, to place comment texts inside of your code, you will need to place the pound sign before the text instead of forward slashes (*#comments*).

The next difference is that in Java, all code must reside inside of the class, but in Python, the code can reside both inside and outside of the class. As you can see in the above example, my *if* statement is located outside of the class “SimpleProgram.” Similarly, functions can be written outside of the class in Python. You will see how it’s done in later examples.

Now, what I called a “method” is written inside of the class. I call it a method in the same way I refer to methods in Java. In Python, if a function resides inside of the class, it’s called a method. If a function resides outside of a class, it’s called a function. Code could also reside outside of the class and outside of the function. Python executes such code line by line, top to bottom, automatically when the *.py* file is executed. In my case, the *if*

statement is outside of the class and outside of all methods, therefore executed first when Python runs the file *pythonAnything.py*, without calling it explicitly. Inside the *if* statement I create an instance of the class “SimpleProgram” what automatically executes the initializer method that resides inside of the class SimpleProgram. In the initializer method I make a call to the *printHelloWorld()* method, which prints out “Hello World” on the screen.

The initializer method is a method that is executed automatically when an instance of the class is created. It is always written in the same format: *def __init__(self):*. This method is very useful as a setup method that automatically executes some piece of code at the creation of the class’s instance before the user can call and run other methods of that class.

In Java, I explicitly specify that method is public. In contrast, all methods in Python are public by default, thus eliminating the need for specification. To make methods private in Python, I add double underscore in front of the name of the method. So my initializer method *def __init__(self):* is a private method. But my *def printHelloWorld(self):* method is public. If I wanted to turn it into a private member of the class, I would add two underscores in front of its name just like in the following example:

```
def __printHelloWorld(self):
```

That would change the way I call that method from initializer to:

```
def __init__(self):  
    self.__printHelloWorld()
```

Class file names in Java have the *.java* extension, but in Python, file names come with the *.py* extension. One very important aspect to remember is that in Java, the class file name must be the same as a class name, but in Python the class file name can be different from the name of the class that resides inside of that file.

In Java all contents of the classes—methods, loops, *if* statements, switches, etc.—are enclosed into blocks surrounded by curly braces *{}*. In Python curly braces do not exist; instead, all code is denoted by indentation.

In Java:

```
if (x>y){  
    x = y++;
```

```
}
```

For the same procedure in Python, simply remove the curly braces but retain the indentation. Use one tab or four spaces for first-level indentation. For second-level indentation, use two tabs or eight spaces; for third-level, three tabs or 12 spaces, and so on.

In Python:

```
if x>y:  
    x = y + 1
```

Semicolons is used to close a statement in Java, but in Python that is not necessary.

In Java:

```
System.out.print("Hello World");
```

In Python:

```
print "Hello World"
```

However, to print a string on a new line in Python, you need to add a new line character `\n` to the beginning, middle, or end of the string, exactly where you would like to insert a new line break.

In Java:

```
System.out.println("Hello World");
```

In Python:

```
print "HelloWorld\n"
```

Next, Example 1.2 demonstrates the differences between Java and Python in their syntax of class and method structure.

EXAMPLE 1.2

In Java:

```
class AnyClass {  
    public void anyMethod() {  
        System.out.println("Hello World");  
    }  
}
```

In Python:

```
class AnyClass:  
    def anyMehotd():  
        print "Hello World\n"
```

2. COMPILING A PROGRAM

Before you can run a Java program, it first needs to be compiled by a Java compiler and turned into a set of bytecodes that the JVM can understand and execute. In Python, this is not necessary since programs can run as interpreted, although there is a way to compile a Python program into an executable file.

Method *main()* is used in Java applications to define the starting point of the application when Java's compiled *.jar* file is being executed to start a program. In Python, there is no *main()* method since the program is not compiled and I have access to all files of the program. As such, I can start a program by running the *python* command along with the file name passed to it as a first argument. Thus, any one of the files of the application could be my starting point for the application to take off running.

If you have ever taken a look at the code of Python programs, you probably noticed the following line of code:

```
if __name__ == "__main__":
```

Don't let this confuse you. This is not the *main()* as the one that exists in Java. Python's *if __name__ == "__main__":* condition is used to prevent a class from being executed when it is imported by another class, but it will execute when called directly by the *python* command. Examples 2.1 and 2.2 demonstrate the difference in syntax between a Python code with the *if __name__ == "__main__":* condition and one without it.

EXAMPLE 2.1

```
# Start of the program
```

```
class SimpleProgram:
    def __init__(self):
        self.PrintHelloWorld()

    def printHelloWorld(self):
        print "Hello World\n"
```

```
run = SimpleProgram()
```

```
# End of the program
```

EXAMPLE 2.2

```
# Start of the program
```

```
class SimpleProgram:
    def __init__(self):
        self.PrintHelloWorld()

    def printHelloWorld(self):
        print "Hello World\n"

if __name__ == "__main__":
    run = SimpleProgram()
    print "Program will execute."
else:
    print "This class was imported and won't execute."

# End of the program
```

Let's try Example 2.1 or Example 2.2 in action. For that, let's create a new file and name it *pythonAnything.py*. Next, paste code from Example 2.1 into that file. Then place that file into a directory that I will name *python_programs*. Since I am on Mac OS X, I will be running my examples using Terminal. On Windows, you can apply the same methods to run Python programs using the Command Prompt (CMD). Let's open a Terminal window and CD to the *python_programs* directory. Then, type the following command:

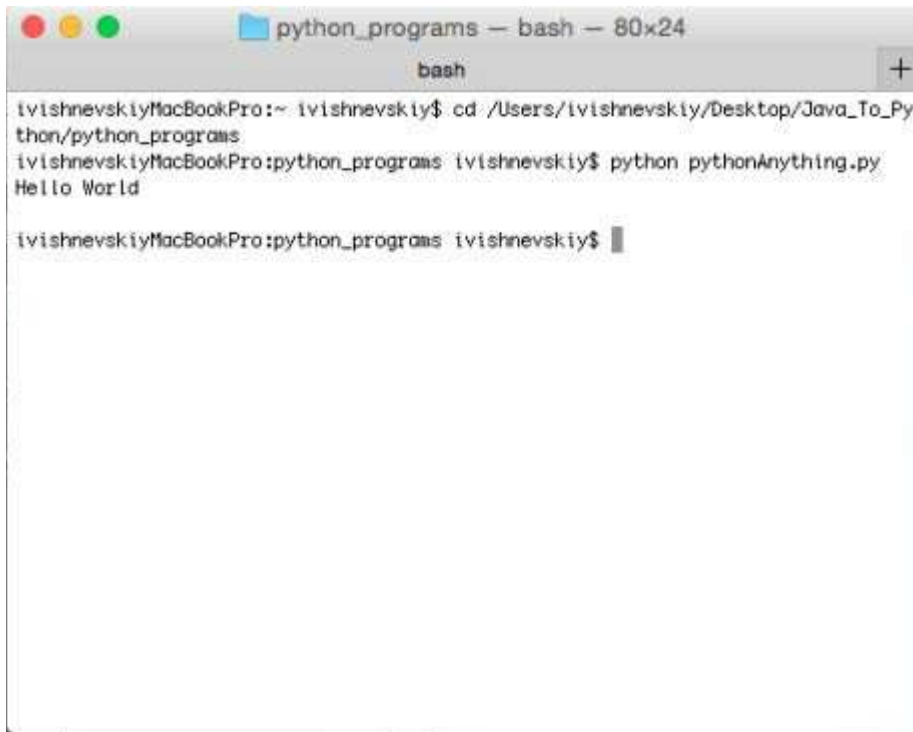
```
python pythonAnything.py
```

What I got is a "Hello World" output and a new line break created by `\n` new line character. Remember to make sure that all your characters in the code are in ASCII format. If you copied code right from the text of this book (its electronic version), the double quotes might paste in an unsupported by Python format. It might cause the following error:

File "pythonAnything.py", line 6

SyntaxError: Non-ASCII character '\xe2' in file pythonAnything.py on line 6, but no encoding declared; see <http://python.org/dev/peps/pep-0263/> for details

To fix it, manually reinsert the double quotes and all other special characters to make sure that all are in the ASCII format. If the program worked correctly, Figure 2.1 is what you can expect to see as an output in your Terminal window.

A screenshot of a macOS Terminal window titled "python_programs — bash — 80x24". The window shows a user navigating to a directory and running a Python script. The output of the script is "Hello World".

```
ivishnevskiyMacBookPro:~ ivishnevskiy$ cd /Users/ivishnevskiy/Desktop/Java_To_Python/python_programs
ivishnevskiyMacBookPro:python_programs ivishnevskiy$ python pythonAnything.py
Hello World
ivishnevskiyMacBookPro:python_programs ivishnevskiy$
```

Figure 2.1

As easy as 1, 2, 3—you just ran your first Python program!

3. VARIABLES

Variables in Python carry the same definition as they do in Java. A variable is the storage location in virtual memory that could be assigned with some value. The value of a variable may be changed during the execution of the program or could be left unchanged as constant. The next program shows how a variable is declared and how it is assigned with a value in Python.

In Python, a variable acts in the same manner as in Java, but, again, simpler. In Java, all variables must be declared with a specific data type, such as integer, string, or boolean. Python again eliminates the need to specify a variable's data type. You simply give a variable its name and assign it a value. Python automatically detects if the assigned value is an integer, string, or any other data type and automatically assigns that variable the correct data type.

When a variable is created with an integer value, it becomes a variable of an integer type. Therefore, if at a later time you wish to convert the variable to a string, you would do so explicitly as such:

str(integerVariable).

Why would you want to convert an integer variable to a string? A great example would be the concatenation of the integer variable to the string. (Concatenation in Python is done in the same way as in Java, by placing a + sign between a string and the variable that is being concatenated to the string.) When concatenation is done in Java, all concatenated variables are automatically converted to a string. Python, however, does not provide the luxury of automatic conversion. To concatenate an integer variable to a string variable, the integer variable has to be explicitly converted to the string as in the following example:

```
strVariable = "String One"
```

```
intVariable = 123
```

```
concatenatedVariable = strVariable + str(intVariable)
```


A similar concept is applicable to the rest of variable data types in Python. In Example 3.1 I will go over every data type and demonstrate how Python's type conversions are done in action.

Let's create a new file *pythonAnythingTwo.py* and put in the code from Example 3.1:

EXAMPLE 3.1

Start of the program

class SimpleProgram:

def variablesInPython(**self**):

 strVariableOne = "String One"

 strVariableTwo = "9876"

 strVariableThree = "9999.99"

 intVariable = 0

 booleanVariable = True

 longVariable = long(9223372036854775807)

 floatVariable = 123.123

#Using Python's function type(), I can see what

#data type is assigned to each variable by Python.

print "Types are:"

print type(strVariableOne)

print type(strVariableTwo)

print type(strVariableThree)

print type(intVariable)

print type(booleanVariable)

print type(longVariable)

print type(floatVariable)

print "\nConverting strVariableTwo to an Integer"

 convertedVariable = int(strVariableTwo)

print convertedVariable

print type(convertedVariable)

print "\nConverting strVariableThree to an Integer"

```
convertedVariable = int(float(strVariableThree))
print convertedVariable
print type(convertedVariable)
```

```
print "\nConverting strVariableTwo to a Float"
convertedVariable = float(strVariableTwo)
print convertedVariable
print type(convertedVariable)
```

```
print "\nConverting strVariableTwo to a Float"
convertedVariable = float(strVariableThree)
print convertedVariable
print type(convertedVariable)
```

```
print "\nConverting intValue to a Long"
convertedVariable = long(intVariable)
print convertedVariable
print type(convertedVariable)
```

```
print "\nConverting longVariable to an Int"
convertedVariable = int(longVariable)
print convertedVariable
print type(convertedVariable)
```

```
print "\nConverting floatValue to a String"
convertedVariable = str(floatVariable)
print convertedVariable
print type(convertedVariable)
```

```
print "\nConverting booleanVariable to a String"
convertedVariable = str(booleanVariable)
print convertedVariable
print type(convertedVariable)
```

```
print "\nConverting booleanVariable to an Integer"
convertedVariable = int(booleanVariable)
print convertedVariable
print type(convertedVariable)
```

```
print "\nUsing intValue as a Boolean"
```

```

if intVariable:
    print "True"
else:
    print "False"

print "\nConcatenating all data types into one string variable"
concatenatedVariable = strVariableOne + \
    str(intVariable) + \
    str(booleanVariable) + \
    str(longVariable) + \
    str(floatVariable)
print concatenatedVariable

```

```

run = SimpleProgram()
run.variablesInPython()

# End of the program

```

From Example 3.1, it is clear that to convert a variable to an integer, I used the Python function *int()*.

To convert to a string, I used Python's function *str()*.

To convert to a float, or double as many call it in Java, I used Python's function *float()*.

To convert to a long, I used Python's function *long()*.

In the section "Converting strVariableThree to an Integer" in Example 3.1, I first converted the string variable to float and then I converted the float variable to integer. This is the way it works in Python. If the string variable has a floating point between digits, it cannot be directly converted to the integer. But with a little trick, I have still achieved my desired result.

```

convertedVariable = int(float(strVariableThree))

```

In the section "Using intVariable as a Boolean" in Example 3.1, the output that I got was "False." That's because my *integer* variable equaled zero. In Python, 0 will always be False and 1 will always be True. Therefore, in your programs instead of explicit True or

False you can return 0 or 1, which can then be used in a Boolean context.

I used Python's function *type()* to find out what type was assigned by Python to my variables. It is a very helpful function, since in many cases, especially when debugging, you won't know the data type of a variable by simply looking at the code. Python 3.x, however, makes it possible to predefine the data type of variables, but even in Python 3.x, it is not necessary to predefine data types. Python 2.7 does not support declaring data types for variables; it does so automatically.

In the section "Concatenating all data types into one string variable," I explicitly casted all variables to a string using Python's function *str()*. As I have already mentioned, concatenation in Python does not automatically convert other data types to string as in Java. Therefore, all non-string variables need to be explicitly converted to a string before they could be concatenated to the string.

Another great example of casting in Python is this:

```
a = "5"  
b = 2  
print(int(a) + b)
```

Output will be: 7

The result of my *pythonAnythingTwo.py* program should look like Figure 3.1.

```
python_programs — bash — 80x53
bash
ivishnevskiyMacBookPro:python_programs ivishnevskiy$ python pythonAnythingTwo.py

Types are:
<type 'str'>
<type 'str'>
<type 'str'>
<type 'int'>
<type 'bool'>
<type 'long'>
<type 'float'>

Converting strVariableTwo to an Integer
9876
<type 'int'>

Converting strVariableThree to an Integer
9999
<type 'int'>

Converting strVariableTwo to a Float
9876.0
<type 'float'>

Converting strVariableTwo to a Float
9999.99
<type 'float'>

Converting intValue to a Long
0
<type 'long'>

Converting longVariable to an Int
9223372036854775807
<type 'int'>

Converting floatValue to a String
123.123
<type 'str'>

Converting booleanVariable to a String
True
<type 'str'>

Converting booleanVariable to an Integer
1
<type 'int'>

Using intValue as a Boolean
False

Concatenating all data types into one string variable
String One0True9223372036854775807123.123
ivishnevskiyMacBookPro:python_programs ivishnevskiy$
```

Figure 3.1

4. CONTROL STATEMENTS AND LOOPS

4.1 *if* Statements

In Python, the concept of *if* statements is the same as in Java. The only slight difference is syntax. Example 4.1 is a clear demonstration of this difference.

EXAMPLE 4.1

Java:

```
if (x > y) {  
    System.out.println("X is larger than Y");  
} else if (x == y) {  
    System.out.println("X equals to Y");  
} else if (x < y) {  
    System.out.println("X is smaller than Y");  
} else {  
    System.out.println("None of the above were true");  
}
```

Python:

```
if x > y:  
    print "X is larger than Y"  
elif x == y:  
    print "X equals to Y"  
elif x < y:  
    print "X is smaller than Y"  
else:  
    print "None of the above are true"
```

4.2 Nested *if* Statements

EXAMPLE 4.2

Java:

```
if (x > y) {  
    if (b < y){  
        System.out.println("B is smaller than Y and X is larger than Y");  
    }  
}
```

Python:

```
if x > y:  
    if b < y:  
        print "B is smaller than y and X is larger than Y"
```

This is all I will cover on *if* statements, since you as a Java developer are familiar with the concept of conditional statements. Example 4.2 should be a sufficient illustration of the difference in syntax between Java and Python. Everything else about *if* statements is the same between Java and Python. In the previous program *pythonAnythingTwo.py* (Example 3.1), the section “Using *intVariable* as a Boolean” uses *if* statements to check if “*intVariable*” contained 0 (False) or 1 (True).

4.3 *for* Loop Statements

For loops in Python are written in a way similar to how enhanced *for* loops are written in Java. Please take a look at the Examples 4.3.1, 4.3.2, and 4.3.3, which are three different situations where *for* loops could be used.

EXAMPLE 4.3.1

Java:

```
for(int x=0; x<5; x++){  
    System.out.println(x);  
}
```

Python:

```
for x in range(0, 5):  
    print x
```

EXAMPLE 4.3.2

Java:

```
String[] array = {"one", "two", "three"};  
for(String value : array){  
    System.out.println(value);  
}
```

Python:

```
array = ["one", "two", "three"];  
for value in array:  
    print value
```

EXAMPLE 4.3.3

Java:

```
String word = "anyWord";  
for (int x = 0; x < word.length(); x++) {  
    System.out.println(word.charAt(x));  
}
```

Python:

```
for letter in "anyWord":  
    print letter
```


In Example 4.3.1, I have printed out the value of the `x` variable in each iteration of a given *for* loop. In Python, the function `range(0, 5)` basically creates a list of integers from 0 to 5, and by using a format similar to Java's enhanced loop, I print out every value of that list and assign them to variable `x`.

In Example 4.3.2, I created an automatically initialized array containing three string values. Using the example from Java, I used enhanced loop format to print out every value of my array. Again, I see that Python's *for* loop format is very similar to Java's enhanced *for* loop format. Since Python doesn't care about data type when running through the *for* loop, the need to specify the data type of each value disappears. The colon sign between the value and the array is replaced with *in*. Instead of a code block defined by a pair of opening and closing curly braces as in Java, I add a colon sign at the end of the header of a *for* loop, *for value in array*:, and with one extra tab of indent following it I specify what should be executed inside of the given *for* loop. In my case, it is *print value*.

Note that the syntax for the automatically initialized array in Python also changed slightly. The values are surrounded by curly braces in Java, but by square braces in Python. I will go over the details of Python's data structures in chapter 7, which will include all you need to know about Python's array lists.

In Example 4.3.3, I used *for* loop to print out every letter of the given string. Python sees all strings as arrays of characters, therefore to iterate through a string you use the same method as in iterating through an array:

EXAMPLE 4.3.4

```
for letter in "anyWord":  
    print letter
```

Example 4.3.4 demonstrates the same as if I would have manually broken up a string *anyWord* into an array of letters and then specified the variable referencing my `letterArray` in the *for* loop as in Example 4.3.5:

EXAMPLE 4.3.5

```
letterArray = ["a", "n", "y", "W", "o", "r", "d"]
for letter in letterArray:
    print letter
```

4.4 *loop continue* Statements

Now, *loop continue* statements is where it gets a little tricky in Python. In Java, you are allowed to label *for* loops and then use *continue* statements to jump from one *for* loop to another within the hierarchy of nested *for* loops. In Python, that's not possible to even simulate in theory. *Continue* statement will only take you to the top of the loop where it resides. Example 4.4 demonstrates what is possible with *continue* statements in Python, but no more than that.

EXAMPLE 4.4

Java:

```
int b = 10;
curentLoop: for (int number = 0; number < b; number++) {
    if (number == 8) {
        continue curentLoop;
    }
    System.out.println(number);
}
```

Python:

```
b = 10
for number in b:
    if number == 8:
        continue
    print number
```

To summarize, Example 4.4 will print 0 to 9, but it will skip digit 8. It is because the *continue* statement makes the program go back to the beginning of the loop without

executing the code located below the *continue* statement.

4.5 *loop pass* Statements

Python has an additional statement, the *pass* statement, that does not exist in Java, but it is very easy to understand because it speaks for itself. In Java, you can simply create an empty method that you would like to come back to and finish later. You can just leave it there knowing that it won't hurt your program. In Python, however, the program won't start if there is an empty method in the class. For that reason the *pass* statement was introduced. It lets you create an empty method—an empty *if* statement or empty loop—inside your program. Example 4.5 demonstrates how this is done.

EXAMPLE 4.5

Python:

```
class MyPythonClass:
    def methodOne(self):
        pass

    def methodTwo(self):
        b = 10
        for number in b:
            if number == 8:
                pass
            print number
        while b == 10:
            pass
```

4.6 *while* Loop Statements

The concept of *while* loops is another point of similarity between Java and Python. The

while loop has a Boolean condition. While that condition is true, the code that resides inside the *while* loop is executed. When the boolean condition becomes false, the *while* loop stops and exits to the next step of the program.

EXAMPLE 4.6.1

Java:

```
int a = 1;  
int b = 10;  
while (a < b){  
    System.out.println("a is smaller than b");  
}
```

Python:

```
a = 1  
b = 10  
while a < b:  
    print "a is smaller than b"
```

In case you need to create an infinite loop, Example 4.6.2 shows exactly how it is done in Python.

EXAMPLE 4.6.2

Java:

```
while (true){  
    System.out.println("running infinitely");  
}
```

Python:

```
while True:  
    print "running infinitely"
```

4.7 *do-while* Loop Statements

The *do-while* loop unfortunately does not exist in Python. However, since I am exploring Python step by step and discover how procedures in Java can also be done in Python, I can simulate a *do-while* loop concept in Python with a bit of savvy thinking. Take a look at Example 4.7. Just as in Java, my Python loop will execute at least once and exit when the condition has been met.

EXAMPLE 4.7

Java:

```
int a = 1;
int b = 10;
do {
    System.out.println("a is smaller than b");
    a++;
} while (a < b);
```

Python:

```
a = 1
b = 10
while True:
    print "a is smaller than b"
    a += 1
    if a > b:
        break
```

Example 4.7 also demonstrated the use of a *break* statement in Python to exit a loop when the condition has been met.

4.8 *switch* Statements

Python, unlike many other programming languages, does not have a Switch statement. But again, I thought outside the box and found a way to use Python's dictionary data structure in a similar way as *switch* statements in Java.

Let's see how can I create my own switches in Python in the following examples (4.8.1 and 4.8.2).

EXAMPLE 4.8.1

Java:

```
private String stringSwitch(String argument) {  
    String valueOut = "";  
    switch (argument) {  
        case "OneIn": valueOut = "OneOut";  
            break;  
        case "TwoIn": valueOut = "TwoOut";  
            break;  
        case "ThreeIn": valueOut = "ThreeOut ";  
            break;  
        case "FourIn": valueOut = "FourOut";  
            break;  
        default: valueOut = "Invalid Value";  
            break;  
    }  
    return valueOut  
}
```

Python:

```
def __stringSwitch(self, argument):  
    return {  
        "OneIn": "OneOut",  
        "TwoIn": "TwoOut",  
        "ThreeIn": "ThreeOut",  
        "FourIn": "FourOut",  
    }
```

```
}[argument]
```

EXAMPLE 4.8.2

Java:

```
private String integerSwitch(int argument) {  
    String valueOut = "";  
    switch (argument) {  
        case 1: valueOut = "One";  
        break;  
        case 2: valueOut = "Two";  
        break;  
        case 3: valueOut = "Three";  
        break;  
        case 4: valueOut = "Four";  
        break;  
        default: valueOut = "Invalid Value";  
        break;  
    }  
    return valueOut;  
}
```

Python:

```
def __integerSwitch(self, argument):  
    return {  
        1 : "One",  
        2 : "Two",  
        3 : "Three",  
        4 : "Four",  
    }[argument]
```

That is as close as I could get to Java's Case/Switch concept in Python. I have incorporated Java's *switch* examples into separate private methods because in Python you would have to build a switch in its own separate method, which would return a value of the dictionary of the specified key. Remember key and value pair in Java? You probably guessed correctly. Dictionary in Python is the same as Map in Java. In chapter 7 I will go

over all details, similarities, and differences between the data structures of Java and Python.

For now, let's see the previous examples in action. Let's add one more *.py* file and call it *pythonAnythingThree.py*. Save the code from Example 4.9 in that file and execute it in Terminal, just as I executed the first two of my Python programs. Please note how I called the first two methods that were public and then last two methods that were private since they had two underscore characters in front of their names. I called all four example methods from within the *def functionThree(self):* method.

EXAMPLE 4.9

Start of the program

class SimpleProgram:

```
def functionThree(self):
    self.ourIfStatementExamples()
    self.ourListExamples()
    print self.__stringSwitch("TwoIn")
    print self.__integerSwitch(3)

def ourIfStatementExamples(self):
    x = 3
    y = 23
    if x > y:
        print "X is larger than Y"
    elif x == y:
        print "X equals to Y"
    elif x < y:
        print "X is smaller than Y"
    else:
        print "None of the above are true"

def ourListExamples(self):
    for x in range(0, 5):
        print x
```



```
array = ["one", "two", "three"];  
for value in array:  
    print value
```

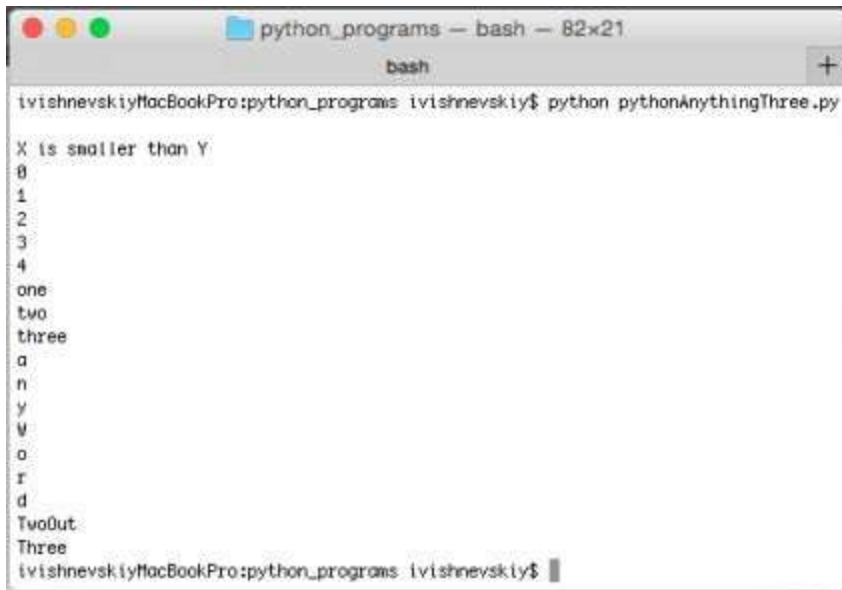
```
for letter in "anyWord":  
    print letter
```

```
def __stringSwitch(self, argument):  
    return {  
        "OneIn": "OneOut",  
        "TwoIn": "TwoOut",  
        "ThreeIn": "ThreeOut",  
        "FourIn": "FourOut",  
    }[argument]
```

```
def __integerSwitch(self, argument):  
    return {  
        1 : "One",  
        2 : "Two",  
        3 : "Three",  
        4 : "Four",  
    }[argument]
```

```
run = SimpleProgram()  
run.functionThree()  
  
# End of the program
```

The output of the *pythonAnythingThree.py* program should be the same as shown in Figure 4.1 below.

A screenshot of a macOS terminal window titled 'python_programs — bash — 82x21'. The window shows a command prompt where the user has run 'python pythonAnythingThree.py'. The output of the script is displayed on the following lines: 'X is smaller than Y', '0', '1', '2', '3', '4', 'one', 'two', 'three', 'a', 'n', 'y', 'W', 'o', 'r', 'd', 'TwoOut', 'Three', and a final command prompt.

```
ivishnevskiyMacBookPro:python_programs ivishnevskiy$ python pythonAnythingThree.py
X is smaller than Y
0
1
2
3
4
one
two
three
a
n
y
W
o
r
d
TwoOut
Three
ivishnevskiyMacBookPro:python_programs ivishnevskiy$
```

Figure 4.1

In the chapters to follow I will not show the screenshots of the coding results. But I encourage you to run each example and see with your own eyes how things work. Try it on your own, just like I have done thus far, and you will see how fast you can pick up Python.

5. OPERATORS

5.1 Basic Arithmetic Operators

The basic arithmetic operators in Java and Python are for the most part the same. I will try to keep this chapter short and only describe a handful of differences.

TABLE 5.1.1

Operator	Result
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Modulus
+=	Addition assignment
−=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

Python does not support the following formats of increment and decrement as in Java:

x++

++x

x--

--x

Instead, in Python you could use the following for increment and decrement:

x += 1

x -= 1

5.2 Relational and Boolean Logical Operators

TABLE 5.2.1

Operator	Result
==	Equal to
is	Equal to
!=	Not equal to
is not	Not equal to
in	Contains
not in	Does not contain
not	Boolean NOT
and	Boolean AND
or	Boolean OR
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Example 5.2.1 demonstrates the use of some of the relational operators that are different in Python compared to Java.

EXAMPLE 5.2.1

Java:

```
String stringOne = "Hello World";  
String stringTwo = "Goodbye World";
```

```

boolean returnValue;
if (stringOne.contains("World")) {
    System.out.println("string contains word World");
} else if (!stringOne.contains("World")) {
    System.out.println("string does not contain word World");
}

if (stringOne == "Hello World") {
    System.out.println("String is Hello World");
}

if (stringOne != "Hello World") {
    System.out.println("String is not Hello World");
}

if (stringOne.contains("Hello") & stringTwo.contains("Goodbye")) {
    returnValue = true;
}

if (stringOne.contains("Hello") || stringTwo.contains("Goodbye")) {
    returnValue = true;
}

```

Python:

```

stringOne = "Hello World"
stringTwo = "Goodbye World"
if "World" in stringOne:
    print "string contains word World"
elif "World" not in stringOne:
    print "string does not contain word World"

if stringOne is "Hello World":
    print "String is Hello World"

if stringOne is not "Hello World":
    print "String is not Hello World"

if "Hello" in stringOne and "Goodbye" in stringTwo:
    returnValue = True

```

```
if "Hello" in stringOne or "Goodbye" in stringTwo:  
    returnValue = True
```

5.3 Ternary Operator

Ternary (three-way) operator `?` does not exist in Python as it does in Java, but the ternary condition can be achieved with a slightly different syntax starting from Python's version 2.5. The concept is the same as in Java. See Example 5.3.1 below.

EXAMPLE 5.3.1

Java:

```
int b = 8;  
int x = b > 8 ? 4 : 9;
```

Python:

```
b = 8  
x = 4 if b > 8 else 9
```

6. CLASSES

Just as in Java, classes in Python also define a new data type. Once referenced, a new data type can be used to create an object of it. Classes in Python are also templates of an object with their functions encapsulated within the form of methods. Once created, objects are also called instances of their classes.

Example 6.0 demonstrates the simple version of the class and method inside of it, in Java and Python:

EXAMPLE 6.0

Java:

```
public class NameOfTheClass {  
    public void nameOfTheMethod() {  
        System.out.println("Hello World");  
    }  
}
```

Python:

```
class NameOfTheClass():  
    def nameOfTheMethod(self):  
        print "Hello World"
```

6.1 Class Variables VS Instance Variables

Class variables and instance variables are both declared outside of the methods, but inside of the class right under the header of the class. Initially, both class and instance variables are instantiated as class variables when an instance of the class is created. However, if

accessed through the instance of the class rather than through the name of the class, the class variable can be converted into an instance variable. By changing the class variable's initial value through the instance of the class, you create a new instance of that variable that will only belong to that particular instance of that class.

However, there is one more way of dealing with it without creating a new instance of the class variable. If the class variable is accessed through the class name and its value is modified, that value will change throughout all instances of that class, even if multiple instances exist.

Let's see some examples.

I have a class:

```
class NameOfTheClass():  
    def nameOfTheMethod(self):  
        print "Hello World"
```

To make an instance of the class *NameOfTheClass()*, I need to reference it and assign that reference a variable name, just like in Java, but simpler. Example 6.1.1 demonstrates the differences between referencing class in Java and Python. Class, method, and variable names are kept the same between Java and Python examples so you could clearly see the differences in syntax between the two languages.

EXAMPLE 6.1.1

Java:

```
class NameOfTheClass {  
    public String instanceVar = "Hello Universe";  
  
    public void nameOfTheMethod() {  
        System.out.println("Hello World");  
    }  
}  
  
class SomeOtherClass {
```



```

public void someOtherMethod() {
    NameOfTheClass instanceOfTheClass = new NameOfTheClass();
    instanceOfTheClass.nameOfTheMethod();
}
}

```

Python:

```

class NameOfTheClass():
    classVar = "Hello Universe"

```

```

def nameOfTheMethod(self):
    print "Hello World"

```

```

class SomeOtherClass():
    def someOtherMethod(self):
        instanceOfClassOne = NameOfTheClass()
        instanceOfClassTwo = NameOfTheClass()

```

*#By changing value of the class variable through the class name,
#value for both, the instanceOfClassOne.classVar and
#instanceOfClassTwo.classVar changed to a new value.*

```

NameOfTheClass.classVar = "Changed Value of Class Variable"

```

*#By changing value of the class var through the instance name,
#you create new instance of that variable that will only belong
#to that particular instance of the class. Trying to change it's
#value through the name of the class will not work anymore.
#The class variable became an instance variable.*

```

instanceOfClassOne.classVar = "Created New Instance Of classVar With New Value"

```

If you may have noticed, in Python, the variable declared outside of the methods, but inside of the class right under the class header, is by default a class variable, but it can be changed into an instance variable using the method specified in the second comment in Example 6.1.1 above. In contrast, in Java the same variable is by default an instance variable. If declared static, the instance variable becomes a class variable.

6.2 Local Variables

In Java, local variables can only be accessed inside the method where they were declared. Making them accessible in other methods is the job of the instance variable.

In Python, local variables are also the variables that are declared inside the methods. Local variables can work as they do in Java and are only accessible inside the method that they were declared in, or they can be made to accessible among all methods inside a given class by adding *self* in front of any local variable's name.

To access such local variables, you always have to reference them with *self.variableName*. Example 6.2.1 demonstrates how this is done.

EXAMPLE 6.2.1

Java:

```
class NameOfTheClass {  
  
    public String instanceVar = "Hello Universe";  
  
    public void nameOfTheMethodOne() {  
        System.out.println(instanceVar);  
    }  
  
    public void nameOfTheMethodTwo() {  
        System.out.println(instanceVar);  
    }  
}
```

Python:

```
class NameOfTheClass():  
  
    def __init__(self):  
        self.localVar = "Hello Universe"
```

```
def nameOfTheMethodOne(self):  
    print self.localVar
```

```
def nameOfTheMethodTwo(self):  
    print self.localVar
```

Definitely, if *.self* is removed from the variable name, the local variable becomes truly local to the method it was declared in, just like in Java. Example 6.2.2 demonstrates just that.

EXAMPLE 6.2.2

Java:

```
class NameOfTheClass {  
  
    public void nameOfTheMethodOne() {  
        String trulyLocalVar = "Hello Universe";  
        System.out.println(trulyLocalVar);  
    }  
}
```

Python:

```
class NameOfTheClass():  
  
    def nameOfTheMethodOne(self):  
        trulyLocalVar = "Hello Universe"  
        print trulyLocalVar
```

6.3 Accessing Methods of a Class

Methods can be accessed from outside of a class as well as from inside the class where

they reside.

If you have noticed, in Examples 6.2.1 and 6.2.2, Python's methods have *self* specified as an argument. That is there for purpose. To make a given method a member of a class, you have to pass *self* as a first argument for that method. To access any method from within other methods inside that same class where they reside, you have to specify *self* before the name of that method. Python's *self*. acts much like *this*. in Java.

EXAMPLE 6.3.1

Java:

```
class NameOfTheClass {  
  
    public void nameOfTheMethodOne() {  
        String trulyLocalVar = "Hello Universe";  
        System.out.println(trulyLocalVar);  
    }  
  
    public void nameOfTheMethodTwo() {  
        nameOfTheMethodOne();  
    }  
}
```

Python:

```
class NameOfTheClass():  
  
    def nameOfTheMethodOne(self):  
        trulyLocalVar = "Hello Universe"  
        print trulyLocalVar  
  
    def nameOfTheMethodTwo(self):  
        self.nameOfTheMethodOne()
```

Methods can also be accessed through an instance of the class from within another class, similar to how it is done in Java. Example 6.3.2 demonstrates those similarities between

Java and Python:

EXAMPLE 6.3.2

Java:

```
class NameOfTheClass {  
    public void nameOfTheMethod() {  
        System.out.println("Hello World");  
    }  
}  
  
class SomeOtherClass {  
    public void someOtherMethod() {  
        NameOfTheClass instanceOfTheClass = new NameOfTheClass();  
        instanceOfTheClass.nameOfTheMethod();  
    }  
}
```

Python:

```
class NameOfTheClass():  
    def nameOfTheMethod(self):  
        print "Hello World"  
  
class SomeOtherClass():  
    def someOtherMethod(self):  
        instanceOfClassOne = NameOfTheClass()  
        instanceOfClassOne.nameOfTheMethod()
```

6.4 Constructor

Sometimes a program needs some values to be passed to the class at the creation of its object. In Java, constructor is in place for that purpose. Python doesn't have the same constructor concept as Java, but there is a way to mimic it using Python's initializer.

The initializer method in Python will execute logic that is enclosed inside it at the initialization of the object of its class, hence its name, or *shortly* `__init__()`. The initializer is basically another method of the class (`def __init__(self):`). It can contain any logic and/or variables that any other method can have in any class. The initializer, just as any other method, has the ability to accept parameters. And because it automatically executes at the moment when reference to that class is made, before you make a call to any method of that class, the parameters specified for `__init__(parameterOne, parameterTwo)` become mandatory. That's when the initializer starts to play a role of the constructor in Python. Example 6.4.1 demonstrates the implementation of a constructor in Java as well as in Python.

EXAMPLE 6.4.1

Java:

```
class NameOfTheClass {
    String parameterOne;
    int parameterTwo;

    public NameOfTheClass(String parameterOne, int parameterTwo) {
        this.parameterOne = parameterOne;
        this.parameterTwo = parameterTwo;
    }
    public void nameOfTheMethod() {
        System.out.println(parameterOne);
    }
}

class SomeOtherClass {
    public void someOtherMethod() {
        String argOne = "Hello World";
        int argTwo = 1;
        NameOfTheClass instanceOfTheClass = new NameOfTheClass(argOne, argTwo);
        instanceOfTheClass.nameOfTheMethod();
    }
}
```

Python:

```
class NameOfTheClass():  
    def __init__(self, parameterOne, parameterTwo):  
        self.parameterOne = parameterOne  
        self.parameterTwo = parameterTwo  
  
    def nameOfTheMethod(self):  
        print self.parameterOne  
  
class SomeOtherClass():  
    def someOtherMethod(self):  
        argOne = "Hello World"  
        argTwo = 1  
        instanceOfClassOne = NameOfTheClass(argOne, argTwo)  
        instanceOfClassOne.nameOfTheMethod()
```

From Example 6.4.1 you need to remember two things.

1. The `__init__()` method can simply serve the purpose of a setup method executioner of a given sequence of steps before proceeding to execute any method that was called from the class where it resides.
2. If parameters are specified for `__init__(parameterOne, parameterTwo)` method, it starts playing the role of the constructor as well as the executioner of a given sequence of steps before proceeding to execute any method that was called from the class where it resides.

6.5 Nested Classes

Both Java and Python support nested classes. With just some slight differences in the syntax, the same that can be achieved in Java can just as easily be achieved in Python. Example 6.5.1 demonstrate nested classes in both Java and Python.

EXAMPLE 6.5.1

Java:

```
class OuterClass {
    String message = "Our message";

    public class InnerClass {
        public String printMessage() {
            return message;
        }
    }
}

class SomeClass {
    public static void main(String args[]) {
        OuterClass outerInstance = new OuterClass();
        OuterClass.InnerClass innnerInstance = outerInstance.new InnerClass();
        System.out.println(innnerInstance.printMessage());
    }
}
```

Python:

```
class OuterClass():
    message = "Our message"

    def __init__(self):
        self.innerInstance = self.InnerClass()

    class InnerClass():
        def printMessage(self):
            return OuterClass.message

outerInstance = OuterClass()
print outerInstance.innerInstance.printMessage()
```

6.6 Returning from Method

Returning from method is done much the same way in Java and Python. The major difference is that in Java, you have to specify what data type method will return at the header of the method. In Python, any method can return any data type without the need to first declare it anywhere.

EXAMPLE 6.6.1

Java:

```
public class SomeOtherClass {  
    public String someOtherMethod() {  
        String argOne = "Hello World";  
        return argOne;  
    }  
}
```

Python:

```
class SomeOtherClass():  
    def someOtherMethod(self):  
        argOne = "Hello World"  
        return argOne
```

6.7 The *finalize()* Method

The *finalize()* method exists in Java to perform some actions before an object gets destroyed. Often it is used to unlock locked by the object elements or to complete cleanup before object gets destroyed by the garbage collector. In Python, the *finalize()* method is called differently but serves exactly the same purpose.

Python's alternative to Java's *finalize()* is `__exit__()`.

Example 6.7.1 demonstrates the implementation of *finalize()* in Java and `__exit__()` in

Python.

EXAMPLE 6.7.1

Java:

```
public class NewClass {  
    public String someMethod() {  
        String argOne = "Hello World";  
        return argOne;  
    }  
  
    @Override  
    protected void finalize() throws Throwable {  
        try {  
            //clean up logic  
        } finally {  
            super.finalize();  
        }  
    }  
}
```

Python:

```
class SomeClass():  
    def someMethod(self):  
        argOne = "Hello World"  
        return argOne  
  
    def __enter__(self):  
        return self  
  
    def __exit__(self, exc_type, exc_val, exc_tb):  
        #clean up logic  
        return
```

6.8 Overloading Methods

Overloading of methods is done somewhat differently in Python and Java. Java would select which method to run based on the list of arguments that given version of the method requests and/or the data type of those arguments. In Python, if you create multiple methods with the same name and try calling one of them, the other methods will throw an exception stating that the number of arguments provided doesn't match the number of parameters specified for those methods. Basically, Python would try to run all of the methods with the same name every time. But there is still a way to overload methods. When parameters are specified for the method, Python lets you set the default value for the parameters. Once the default value is set, then you do not need to provide value for that parameter when calling that method—it will simply pick up the default value. Inside the method itself, using conditional statements, you can choose what step sequence the method should execute based on what values are set for its parameters. Example 6.8.1 demonstrates how Java-like overloading can be implemented in Python.

EXAMPLE 6.8.1

Java:

```
class SomeClass {  
    public void someMethod() {  
        System.out.println("Running sequence one");  
    }  
    public void someMethod(String valueOne) {  
        System.out.println("Running sequence two");  
    }  
  
    public void someMethod(int valueTwo, Boolean valueTree) {  
        System.out.println("Running sequence three");  
    }  
}
```

Python:

```
class SomeClass():  
    def someMethod(self, valueOne=None, valueTwo=None, valueThree=False):  
        if valueOne != None:
```

```
    print "Running sequence two"
elif valueTwo != None and valueThree:
    print "Running sequence three"
else:
    print "Running sequence one"
```

From the above example, calling *someMethod()* without any arguments, or with one, or with three arguments will all work. Example 6.8.2 demonstrates the variations of how *someMethod()* could be called with a different set of parameters passed to it.

EXAMPLE 6.8.2

Python:

```
run = SomeClass()
run.someMethod()
run.someMethod("valueOneStr")
run.someMethod(None, "valueTwoStr", True)
```

6.9 Overloading Constructor

Very similarly to how I overloaded methods, I can overload Python's constructor, which I mentioned before is called "the initializer" in Python. Remember `__init__()` method? If not, please refer to section "6.4 Constructor."

In Examples 6.8.1 and 6.8.2 I set default values to the parameters of the method, I then overloaded. Similarly, `__init__()` is also a method, therefore the same concept can be applied to it. Inside the initializer then, using conditional statements, I choose which route to take for the program. Let's take a look at Example 6.9.1, which shows how to overload the initializer in Python.

EXAMPLE 6.9.1

Python:

```

class SomeClass():

    def __init__(self, valueOne=None, valueTwo=None, valueThree=False):
        if valueOne != None:
            self.methodTwo()
        elif valueTwo != None and valueThree:
            self.methodThree()
        else:
            self.methodOne()

    def methodOne(self):
        print "Running sequence one"

    def methodTwo(self):
        print "Running sequence two"

    def methodThree(self):
        print "Running sequence three"

```

```

runOne = SomeClass()
runTwo = SomeClass("valueOneStr")
runThree = SomeClass(None, "valueTwoStr", True)

```

6.10 Using Objects as Parameters

Since in Python I do not need to specify data types for the arguments that are expected to be passed to the methods, I can pass any objects of any data types as arguments to those methods. (Notice that in most of the previous examples, I did not specify the object's data type, whatever it might be.)

The example below demonstrates how an object of the class *NameOfTheClass()* can be passed as an argument from the method *someOtherMethod(self)*: located in the class *SomeOtherClass()* to the method *someMethodTwo(self, objectAsParameter)*, where using the passed object I call method *nameOfTheMethod()* from class *NameOfTheClass()*.

In a similar manner, objects can be returned by any method, just as the *someMethodTwo(self, objectAsParameter)*: method returns the object *objectAsParameter*.

EXAMPLE 6.10.1

Python:

```
class NameOfTheClass():  
    def __init__(self, parameterOne, parameterTwo):  
        self.parameterOne = parameterOne  
        self.parameterTwo = parameterTwo  
  
    def nameOfTheMethod(self):  
        print self.parameterOne  
  
class SomeOtherClass():  
    def someOtherMethod(self):  
        argOne = "Hello World"  
        argTwo = 1  
        instanceOfClassOne = NameOfTheClass(argOne, argTwo)  
        self.someMethodTwo(instanceOfClassOne)  
  
    def someMethodTwo(self, objectAsParameter):  
        objectAsParameter.nameOfTheMethod()  
        return objectAsParameter
```

6.11 Recursion

Recursion in Java and Python has the same meaning—method is calling itself to rerun its step sequence. The code from Example 6.11.1, if ran, will run infinitely. It is therefore not the best case when recursion would be usefull in the real life, but it shows the similar technique in Java and Python for a method to call itself.

EXAMPLE 6.11.1

Java:

```
public class SomeClass {  
    public void someMethod() {  
        someMethod();  
    }  
}
```

Python:

```
class SomeClass():  
    def someMethod(self):  
        self.someMethod()
```

6.12 Access Control

Public getters and setters access private methods and/or variables of their class, remember? Python also supports the same concept, allowing me to make some vulnerable methods as private, accessible only by other methods of the class where they reside. In Java you would explicitly type *private* at the header of the method when declaring the private method, or *public* when declaring a public method. In Python, by default all methods are public. To make a method private, you need to add two underscore characters before the name of the method, as shown below in Example 6.12.1.

EXAMPLE 6.12.1

Java:

```
public class SomeClass {  
    public void firstMethod() {  
        System.out.println("Hello public method");  
    }  
    private void secondMethod() {  
        System.out.println("Hello private method");  
    }  
    public void thirdMethod() {
```

```

    //Calling public method:
    firstMethod();
    //Calling private method:
    secondMethod();
}
}

```

Python:

```

class SomeClass():
    def firstMethod(self):
        print "Hello public method"

    def __secondMethod(self):
        print "Hello private method"

    def thirdMethod(self):
        #Calling public method:
        self.firstMethod()
        #Calling private method:
        self.__secondMethod()

```

6.13 Static Methods

Static methods play the same role in Python as they do in Java. To define a static method in Java, you would use the keyword *static*. In Python, static methods are defined using the decorator *@staticmethod*. Python's decorators in Java are called annotations and are placed at exactly the same location, right above the header of the method that you would like to declare static. Please note that in Example 6.13.1, the static method does not have a *self* parameter anymore. For a static method, the *self* parameter does not need to be specified and cannot be specified. All static methods are accessed from other methods using the name of the class where they reside, much the same as in Java. Example 6.13.1 demonstrates the use of static methods in Java and Python.

EXAMPLE 6.13.1

Java:

```
class SomeClass {  
    public static void someStaticMethod() {  
        System.out.println("Hello static method");  
    }  
}  
  
public class SomeOtherClass {  
    public void someOtherMethod() {  
        SomeClass.someStaticMethod();  
    }  
}
```

Python:

```
class SomeClass():  
    @staticmethod  
    def someStaticMethod():  
        print "Hello static method"  
  
class SomeOtherClass():  
    def someOtherMethod(self):  
        SomeClass.someStaticMethod()
```

6.14 Final

In Java, declaring a variable as *final* prevents its contents from being modified. In Python, *final* does not exist, but there are immutable data structures. For example, while you can edit values of the *list*, you cannot edit the values of the *tuple* because it is immutable. Section 7.14 will explain the implementation of tuples in greater detail.

The same concept applies to set (in Java) versus a frozen set (in Python). A frozen set is also immutable, while set is mutable. Section 7.24 discusses the implementation of the frozen set in Python.

6.15 Command-Line Arguments

In Java, by default the *main()* method of a program expects arguments to be passed when the *java -jar filename.jar* command is executed. In Python, the process is different. To pass arguments when executing a *.py* file using the *python filename.py* command in the command prompts, that *.py* file should import module *sys*, then through the reference to the module *sys* you can extract the arguments into the program and use them from then on inside or outside of the class enclosed in the *.py* file that is being called. Example 6.15.1 demonstrates exactly how command-line arguments are passed to a Python program.

EXAMPLE 6.15.1

Java:

Command:

“java -jar filename.jar argumentOne argumentTwo argumentThree”

```
public class CommandLineArgsExample {  
    public static void main(String args[]) {  
        System.out.println("Argument 1: " + args[0]);  
        System.out.println("Argument 1: " + args[1]);  
        System.out.println("Argument 1: " + args[2]);  
    }  
}
```

Python:

Command:

“python filename.py argumentOne argumentTwo argumentThree”

```

import sys
class CommandLineArgsExample():
    def __init__(self):
        print "Argument 1: " + str(sys.argv[1])
        print "Argument 2: " + str(sys.argv[2])
        print "Argument 3: " + str(sys.argv[3])

```

Note that in the Python example, the index of the first argument is 1, not 0 as it is in Java. This is because Python places the name of the file that is being executed under index 0, then under index 1 and so on. It appends arguments that follow the filename in the same order, from left to right. The rest of the process is very similar to the way command-line arguments work in Java.

6.16 Variable-Length Arguments

Just like in Java, Python lets you pass dynamic numbers of arguments to the method. While concept and purpose stays the same, syntax changes. In Java, you would use three dots before the variable name to turn that variable into a *VarArg*. In Python, to achieve the same, you would need to place an asterisk in front of the variable name for the same result. Example 6.16.1 demonstrates the differences.

EXAMPLE 6.16.1

Java:

```

class VarArgsExample {
    public void methodName(String ... varName) {
        for (int counter = 0; counter < varName.length; counter++)
            System.out.println("Argument" + counter + ": " + varName[counter]);
    }
}

```

Python:

```

class VarArgsExample():
    def methodName(*varName):
        for counter, element in enumerate(varName):
            print "Argument" + counter + ": " + element

```

6.17 Inheritance

Just as in Java, Python lets classes inherit each other's properties. One major difference is that in Java, one class can extend only one other class at a time, whereas Python removes that limitation and allows one class to extend multiple other classes, thus inheriting properties of multiple classes in one shot. Example 6.17.1 below illustrates this difference.

EXAMPLE 6.17.1

Java:

```

class MainClass {
    public static void main(String args[]) {
        SomeClass scRef = SomeClass();
        scRef.someMethod();
    }
}

class SomeClass {
    public void someMethod() {
        System.out.println("Inside SomeClass");
    }
}

class SomeOtherClass extends SomeClass {
    public void someOtherMethod() {
        System.out.println("Inside SomeOtherClass");
    }
}

```

Python:

```
class MainClass():  
    def __init__(self):  
        scRef = SomeClass()  
        scRef.someMethod()  
        scRef.methodOfThirdClass()  
  
class SomeClass():  
    def someMethod(self):  
        print "Inside SomeClass"  
  
class ThirdClass():  
    def methodOfThirdClass(self):  
        print "Extended this one too"  
  
class SomeOtherClass(SomeClass, ThirdClass):  
    def someOtherMethod(self):  
        print "Inside SomeOtherClass"
```

As you can see from Example 6.17.1, in Python, class *SomeOtherClass* has extended class *SomeClass*. Although this is the same as inheritance in Java, in this example for Python I added one more class, *ThirdClass*, which was also extended by *SomeOtherClass*. This demonstrates how multiple classes can be extended in Python, which is impossible in Java.

Similarly, as in Java, only the public methods can be inherited. To declare a method as public in Java, I explicitly specify that method as public. In Python, by default all methods are public, therefore you don't need to specify that at all. But to make method private, you add double underscore in front of the name of the method. Therefore, to make following method private:

```
def printHelloWorld(self):
```

all that needs to be done is add two underscore characters before the name of the method, as shown below.

```
def __printHelloWorld(self):
```

6.18 Abstract Classes

Abstract classes serve a purpose of a blueprint that dictates what methods must be implemented in the classes that extend them. Abstract classes exist in both Java and Python. You are familiar with how it's done in Java, which is straightforward, but in Python the syntax gets a little more confusing.

In Python, abstract classes are declared using the module *abc*. This module must first be imported into the *.py* file, where abstract class will reside. Then, below the header of the class, the following needs to be specified for that class to become *abstract*: `__metaclass__ = abc.ABCMeta`. Methods are declared abstract with the help of the decorator `@abc.abstractmethod`, which is placed right above the header of each abstract method. Python's decorators are used in a way similar to how annotations are used in Java. Take a look at Example 6.19.1.

EXAMPLE 6.19.1

Java:

```
abstract class AbstractClass {  
    abstract String mustHaveMethod();  
}  
  
public class SomeOtherClass extends AbstractClass {  
    public String mustHaveMethod() {  
        System.out.println("mustHaveMethod has been implemented");  
        return "String";  
    }  
}
```

Python:

```
import abc  
  
class AbstractClass(object):
```

```
__metaclass__ = abc.ABCMeta
```

```
@abc.abstractmethod  
def mustHaveMethod(self):  
    return
```

```
class SomeOtherClass(ABCClass):  
    def mustHaveMethod(self):  
        print "mustHaveMethod has been implemented"
```

6.19 Importing Classes into a Workspace

1) Let's say you have one *.py* file that contains multiple classes inside and you need to import all of them at once. All you need to import is a file that contains all of the classes that you would like to use in a separate workspace of the program.

Similar to Java, in Python different classes could reside in different *.py* files. In Java, those would be *.java* files. To import a class from one *.java* file to another, you need to specify its classpath by separating directory names with a dot. In Python, the same can be done in Python 3.x, but in Python 2.7, separating directory names with a dot in a class path would not work. Examples in this section demonstrates how to import classes from other files in Python 2.7.

In Python 2.7, if a file with the class to be imported is located in a different directory than the file to which import needs to be made, the module *sys* is used to insert a path into a memory. Python will then be looking for the file from which it is to import the needed class in the provided path. Please take a look at Example 6.19.1.

EXAMPLE 6.19.1

Python:

```
import sys
```

```
sys.path.insert("/path/to/file/")
```

```
import fileName
```

If the whole file was imported, as in Example 6.19.1, then all classes of *fileName.py* will be accessible through the file name itself. Example 6.19.2 demonstrates this.

EXAMPLE 6.19.2

Python:

```
import sys
```

```
sys.path.insert("/path/to/file/")
```

```
import fileName
```

```
class SomeClass():
```

```
    def someMethod(self):
```

```
        referenceToClass = fileName.ClassName()
```

```
        referenceToClass.methodName()
```

2) Let's say you have one *.py* file that contains multiple classes, but you would like to import only one class. Example 6.19.3 shows how you would limit the import to only one class from the given *.py* file, even if that file contains multiple classes.

EXAMPLE 6.19.3

Python:

```
import sys
```

```
sys.path.insert("/path/to/file/")
```

```
from fileName import ClassName
```

```
class SomeClass():
```

```
    def someMethod(self):
```

```
        classObj = ClassName()
```

```
        classObj.methodName()
```


3) What if you would like to import a class and assign it a distinctive variable name at the import? That is also possible in Python. Example 6.19.4 demonstrates how.

EXAMPLE 6.19.4

Python:

```
import sys
sys.path.insert("/path/to/file/")

import fileName as myLibrary

class SomeClass():
    def someMethod(self):
        myClass = myLibrary.ClassName()
        myClass.methodName()
```

In Example 6.19.4 note that reference to class *ClassName* will have to be made through variable *myLibrary*.

4) What if the *.py* file is located two directories back and there is no way to specify a path to them going forward. It could be achieved by telling your working *.py* file to start looking for the class path two directories back. Again I call for help from the same module, *sys*, which has to first be imported into the working code base. Then by calling: *sys.path.append("../..")*, I can specify that I would like this particular *.py* file to start looking for classes to import two directories back from where my working *.py* file is located.

EXAMPLE 6.19.5

Python:

```
import sys
sys.path.insert("../..")

import fileName
```

6.20 Exception Handling

Exceptions are caught and handled much the same way in Java and Python. A small difference in syntax is easy to understand with the following example (6.20.1). One of the obvious differences is that what is called “*catch*” in Java is referred to as “*except*” in Python. In Java you would use try/catch and in Python you would use try/except. For the rest of exception handling, the two languages are similar in action.

EXAMPLE 6.20.1

Java:

```
class NameOfTheClass {  
    public void nameOfTheMethod() {  
        System.out.println("Hello World");  
    }  
}  
  
class SomeOtherClass {  
    public void someOtherMethod() {  
        NameOfTheClass instanceOfTheClass = new NameOfTheClass();  
  
        try {  
            instanceOfTheClass.nameOfTheMethod();  
        } catch (Exception e) {  
            System.out.println("Exception caught");  
            e.printStackTrace();  
        }  
    }  
}
```

Python:

```
class NameOfTheClass():  
    def nameOfTheMethod(self):  
        print "Hello World"
```

```

class SomeOtherClass():
    def someOtherMethod(self):
        instanceOfClassOne = NameOfTheClass()

        try:
            instanceOfClassOne.nameOfTheMethod()
        except Exception, e:
            print "Exception caught"
            print e

```

6.21 Throwing Custom Exception

A very common practice in programming is to generate custom exception for exceptional cases related only to your case. This way you can catch only your specific exception and handle it, while leaving the rest of possible exceptions unhandled. This can be done both in Java and Python. Example 6.21.1 demonstrates how custom exception can be created and thrown in both languages.

EXAMPLE 6.21.1

Java:

```

class YourCustomException extends Exception {
    public YourCustomException(String errorMessage) {
        super(errorMessage);
    }
}

class SomeOtherClass {
    public static void main(String args[]) {
        throw new YourCustomException("Your custom error message.");
    }
}

```

Python:

```
class MyException(Exception):
    def __init__(self, message):
        super(MyException, self).__init__(message)

class SomeOtherClass():
    def __init__(self):
        raise MyException("Your custom error message.")

SomeOtherClass()
```

6.22 @Decorators are not @Annotations

Python has a property very similar in appearance to Java's annotations. Annotations in Java are added to replace XML-based meta data files that were used to preserve meta data and access it at the runtime of the Java program. In Python, what looks like annotations are called decorators. Decorators are also specified above the header of the method to which they belong and also have an @ sign in front to distinguish them. Their purpose is slightly different from annotations in Java. In brief, Python lets you pass functions as arguments to other functions. Just like you pass a variable as an argument to any method/function, you can also pass a function itself. A decorator basically tells the decorated function to be passed as an argument to the decorator function.

In Example 6.22.1, I named functions as *decoratorFunctionAnyName()* and *decoratedFunctionAnyName()* to distinguish the two. The first, *decoratorFunctionAnyName()*, has a nested function inside called *innerFunctionAnyName()*. Don't ask me why; it's the way the syntax is. It grabs the value that *decoratedFunctionAnyName()* returns, which in my case is 3000, adds 1 to it, and returns the updated result. The *decoratorFunctionAnyName()* then returns the value that *innerFunctionAnyName()* returns. Note that in the return statement of *decoratorFunctionAnyName()* function, I used only the name of *innerFunctionAnyName*, without trailing parentheses *()*. Including the trailing parentheses will cause exceptions. Next, the *decoratedFunctionAnyName()* function picks up the value returned by

decoratorFunctionAnyName() and returns it on the place of *x* variable. Later, the *print* statement outputs that updated value into a console.

Also note that name next to the @ sign is exactly the same name as the name of the decorator function: *decoratorFunctionAnyName() = @decoratorFunctionAnyName*

Take a look at how decorators are implemented in Python in Example 6.22.1.

EXAMPLE 6.22.1

Python:

```
def decoratorFunctionAnyName(decoratedFunctionAsArg):
```

```
    def innerFunction():
```

```
        result = decoratedFunctionPassedAsArgument() + 1
```

```
        return result
```

```
    return innerFunction
```

```
@decoratorFunctionAnyName
```

```
def decoratedFunctionAnyName(x=3000):
```

```
    return x
```

```
print decoratedFunctionAnyName()
```

Note that in Example 6.23.1, I have two functions running outside of the class. Now let's see Example 6.23.2, which shows how decorators can be implemented and used if my code was enclosed into a class.

EXAMPLE 6.22.2

Python:

```
class SomeClass():
```

```
def decoratorFunctionAnyName(decoratedFunctionAsArg):  
    def innerFunction(self):  
        result = decoratedFunctionPassedAsArgument(self) + 1  
        return result  
    return innerFunction
```

```
@decoratorFunctionAnyName  
def decoratedFunctionAnyName(self, x=3000):  
    return x
```

```
run = SomeClass()  
print run.decoratedFunctionAnyName()
```

7. DATA STRUCTURES

7.1 Arrays

In Java, arrays are of a fixed length, but in Python, an array that is called a “list” acts more like ArrayList in Java. It can increase or decrease in size dynamically. Another difference is that in Java, an array has to be initialized with a set data type, but in Python it can hold one or a mix of any data types and even sub lists or any other collection types, which I will talk about later in section 7.11. Examples 7.1.1 and 7.1.2 compare Java’s array data structure to Python’s list data structure.

EXAMPLE 7.1.1

Java:

```
int[] array = new int[5];  
array[0] = 123;  
array[1] = 321;  
array[2] = 999;  
array[3] = 764;  
array[4] = 987;
```

Python:

```
array = []  
array.append(123)  
array.append(321)  
array.append(999)  
array.append(764)  
array.append(987)
```

EXAMPLE 7.1.2

Java:

```
int array[] = { 123, 321, 999, 764, 987 };
```

Python:

```
array = [123, 321, 999, 764, 987]
```

7.2 Accessing Values in Arrays

The way values are accessed in arrays is the same in Java and Python, even in syntax.

EXAMPLE 7.2.1

Java:

```
int array[] = { 123, 321, 999, 764, 987 };  
System.out.print(array[3]);
```

Python:

```
array = [123, 321, 999, 764, 987]  
print array[3]
```

7.3 Updating Values in Arrays

Updating values in arrays is also done almost the same way in Java and Python.

EXAMPLE 7.3.1

Java:

```
int array[] = { 123, 321, 999, 764, 987 };  
array[3] = 444;
```


Python:

```
array = [123, 321, 999, 764, 987]  
array[3] = 444
```

7.4 Getting the Size of an Array

The size of an array in Python is retrieved by function *len()*. This function could also be used to get the length of all collection types in Python and to calculate the number of characters in a given string.

EXAMPLE 7.4.1

Java:

```
int array[] = { 123, 321, 999, 764, 987 };  
System.out.print(array.length);
```

Python:

```
array = [123, 321, 999, 764, 987]  
print len(array)
```

7.5 Sorting Arrays

The process of sorting arrays in Python is similar to in Java. With the exception of a few slight changes in syntax, the process for the most part is almost identical. Example 7.5.1 demonstrates how to sort array in Java and Python in ascending as well as descending orders.

EXAMPLE 7.5.1

Java:

```
int array[] = { 123, 321, 999, 764, 987 };  
//Sort in Ascending Order  
Arrays.sort(array);  
//Sort in Descending Order  
Arrays.sort(array, Collections.reverseOrder());
```

Python:

```
array = [123, 321, 999, 764, 987]  
#Sort in Ascending Order  
array.sort()  
#Sort in Descending Order  
array.sort(reverse=True)
```

7.6 Counting Values in an Array

Counting appearances of elements in a given array in Python is simpler than ever. In Java, the simplest way to count the appearance of a given value in a given array is done through the loop and the conditional comparison of each element to the value being counted. Python, on the other hand, has a special function just for that: *count()*. Let's see the *count()* function in action using the next in Example 7.6.1.

EXAMPLE 7.6.1

Java:

```
int array[] = { 123, 321, 999, 764, 987 };  
int count = 0;  
int value = 999;  
for (int i = 0; i < array.length; i++) {  
    if (array[i] == value) {
```

```

        count++;
    }
}

```

Python:

```

array = [123, 321, 999, 764, 987]
count = array.count(999)

```

7.7 Inserting a Value under a Certain Index in an Array

Let's say I have an array of five elements. I would like to insert a value under index 3. If I just wanted to update the value under index 3, I would use the method described in section 7.3 (Updating values in Arrays). However, since I already have a value under index 3 and would like to preserve it, Example 7.7.1 demonstrates how I could add a new value under index 3 and shift all existing values in the array one index to the right.

EXAMPLE 7.7.1

Java:

```

Integer array[] = { 123, 321, 999, 764, 987 };
ArrayList<Integer> list = new ArrayList<Integer>(Arrays.asList(array));
list.add(3, 555);

```

Python:

```

array = [123, 321, 999, 764, 987]
array.insert(3, 555)

```

7.8 Return the Index of an Element in an Array

Returning the index of a desired value in a given Python's list is very easy. Example 7.8.1 is self-explanatory and demonstrates just that.

EXAMPLE 7.8.1

Java:

```
int array[] = { 123, 321, 999, 764, 987 };  
Arrays.asList(array).indexOf(321);
```

Python:

```
array = [123, 321, 999, 764, 987]  
array.index(321)
```

7.9 Difference between Appending and Extending to an Array

Here I have something new that doesn't exist for arrays in Java. As I have mentioned earlier, in Python, arrays can hold any data type and even lists. Let's call the initial array the "parent list" and the list element that I want to add to it the "child list." The child list could be appended and/or extended to the parent list. If appended, the child list element will be added as its own list element, or, if you will, a "child sub list" of a parent list. But if extended, the child list will be broken into its values and each of those values will then be separately appended to the parent list, thus making each value the separate element of the parent list. This is illustrated in Example 7.9.1. Since this functionality is specific to Python, the example will only contain code sample from Python.

EXAMPLE 7.9.1

Python:

```
parentList = [123, 321, 999, 764, 987]  
childList = [2, 4, 6, 2, 8]  
parentList.append(childList)  
#parentList will look this way: [123, 321, 999, 764, 987, [2, 4, 6, 2, 8]]
```

```
parentList = [123, 321, 999, 764, 987]
childList = [2, 4, 6, 2, 8]
parentList.extend(childList)
#parentList will look this way: [123, 321, 999, 764, 987, 2, 4, 6, 2, 8]
```

7.10 Deleting Elements from an Array by Index

In Java, for complex array manipulations you would not even use a regular array. Instead, you would use ArrayList. In Python, however, there is one list that acts as everything that you would need in an array. In Example 7.10.1, I first convert a regular integer array in Java to ArrayList and then delete an item from within ArrayList by index. In Python the function that removes the element by index is named the same way as in Java, *remove()*.

EXAMPLE 7.10.1

Java:

```
int array[] = { 123, 321, 999, 764, 987 };
ArrayList<Integer> arraylist = new ArrayList<Integer>(Arrays.asList(array));
Arraylist.remove(2);
```

Python:

```
array = [123, 321, 999, 764, 987]
array.remove(2)
```

Python also provides one more way of deleting values by index from an array, as shown in Example 7.10.2.

EXAMPLE 7.10.2

Python:

```
array = [123, 321, 999, 764, 987]
del array[2]
```

Both methods will do the job. Using *del* or *remove()* will delete the specified element from the array, which in Python is called the list.

7.11 Lists in Python Can Hold Anything

In Java, all arrays are limited to hold elements of only one specific data type. For example, string arrays can hold only strings. Integer arrays can hold only integers. In Python, however, arrays are free to hold any combination of supported data types and even lists, tuples and, dictionaries. Example 7.11.1 demonstrates a working Python list that holds a variety of data types as values.

EXAMPLE 7.11.1

Python:

```
array = [123, [2, 4, 6, 2, 8], "testString", {"key" : "value", "key" : "value"},
("testStringTwo", 123)]
```

In the above example, the first element of my array is an integer, the second a list of integers, the third a string, the fourth a dictionary, and the fifth a tuple. This array is absolutely valid in Python.

7.12 Multidimensional Arrays

Multidimensional arrays are also supported in Python, although working with them in Python can be more confusing than in Java. It is impossible to simply create an empty

multidimensional array in Python as you could with an empty regular Python list. In Python, you need to first create a multidimensional array first using list comprehension, which I will get to in the next paragraph of this chapter (7.13). When an array is created with default ZERO integer values filled into each cell of each row and column combination, then I can replace the ZERO (or any value chosen to be default) values using nested *for* loops with those that were meant to be in that multidimensional array. Example 7.12.1 converts a two-dimensional array in Java into a two-dimensional list in Python.

EXAMPLE 7.12.1

Java:

```
String[][] twoDimensionalArray = new String[6][5];
for (int counter = 0; counter < 6; counter++) {
    for (int innerCounter = 0; innerCounter < 5; innerCounter++) {
        twoDimensionalArray[counter][innerCounter] = "value";
    }
}
```

Python:

```
twoDimensionalArray = [[0 for innerCounter in range(5)] for counter in range(6)]
for counter in range(6):
    for innerCounter in range(5):
        twoDimensionalArray[counter][innerCounter] = "value"
```

7.13 List Comprehension

List comprehension is a cool feature in Python's data structures and doesn't exist in Java. The concept is very simple to understand. It is basically a way to build an array of values in Python using only one line of code. It might be confusing at first, but with practice, you will realize that building arrays using list comprehension in Python is more convenient than using the conventional method, although the latter certainly works in Python. The choice of which approach to use is entirely yours as a Python software developer. Example 7.13.1 compares the conventional way of building an array using *for* loop and using list

comprehension.

EXAMPLE 7.13.1

Python:

```
list = [x**2 for x in range(10)]
```

is the same as:

```
list = []  
for x in range(10):  
    list[x] = x**2
```

7.14 Tuple

What is a tuple? It is basically the same thing as a list in Python, but a tuple is immutable, while a list is mutable. A tuple has a fixed size in nature, but a list—as I have mentioned before—can grow and decrease in size dynamically. In Python, you cannot dynamically add elements to tuples, and neither do they support *append* or *extend* as lists do. Tuples also do not support *remove* or *pop* as lists do.

With all their limitations, tuples have benefits that make them useful in some situations. Tuples are faster than lists, so if you are building a list of constants that you would only need to iterate through, then use a tuple instead of a list. Second, tuples protect your data from being changed since elements in tuple cannot be changed, making your code safer.

The difference between initializing a tuple and a list is in the parentheses used to surround the elements. Example 7.14.1 demonstrates a tuple. Just take a Python list from any of the previous examples and change the square parentheses to the rounded parentheses. That's all—you just created a tuple from the list.

EXAMPLE 7.14.1

Python:

```
tuple = (123, 321, 999, 764, 987)
```

A tuple can be converted into a list, and vice versa, as shown in Example 7.14.2.

EXAMPLE 7.14.2

Python:

```
tupleOne = (123, 321, 999, 764, 987)
listOne = [123, 321, 999, 764, 987]
```

```
listTwo = list(tupleOne)
tupleTwo = tuple(listOne)
```

7.15 Python's Dictionary Is Java's Map

Dictionaries in Python serve the same purpose as maps do in Java. Keys and values pair with each other. Keys must be unique, but if by any chance you have specified a duplicate key, don't worry, the program will still run. When duplicate keys are encountered during assignment, the last assignment overwrites all previous assignments. Keys can also only be of immutable types such as integers, strings, or tuples. As you have learned previously, tuples are immutable and can therefore be used as keys in Python's dictionaries. In Java's maps, keys can only be of one data type within the same map, for example integer. But in Python, even within the same dictionary, keys can be of different data types. One key could be an integer, another a string, and a third a tuple, as long as they are immutable. Values could be any mutable or immutable of any data types, even lists, tuples, and/or nested dictionaries. Examples 7.15.1 and 7.15.2 demonstrate how Java's map would look if converted to Python's dictionary.

EXAMPLE 7.15.1

Java:

```
HashMap<Integer, String> dictionary = new HashMap<Integer, String>();  
dictionary.put(1, "One");  
dictionary.put(2, "Two");  
dictionary.put(3, "Three");  
System.out.println(dictionary.get(1));  
System.out.println(dictionary.get(3));
```

Python:

```
dictionary = {1 : "One", 2 : "Two", 3 : "Three"}  
print dictionary[1]  
print dictionary[3]
```

EXAMPLE 7.15.2

Python:

```
dictionaryTwo = {1 : "ValueOne", "KeyTwo" : [276, "ten"], (1, 2, 3) : "valueThree"}  
print dictionaryTwo[1]  
print dictionaryTwo["KeyTwo"]  
print dictionaryTwo[(1, 2, 3)]
```

7.16 Update an Existing Entry in a Dictionary

Updating values in a Python dictionary is done similarly to updating values in a Python list, but instead of an index you would provide an existing dictionary key specifying which value to update. Please take a look at Example 7.16.1.

EXAMPLE 7.16.1

Java:

```
HashMap<Integer, String> dictionary = new HashMap<Integer, String>();  
dictionary.put(1, "One");  
System.out.println(dictionary.get(1));  
dictionary.put(1, "UpdatedOne");  
System.out.println(dictionary.get(1));
```

Python:

```
dictionary = {1 : "One", 2 : "Two", 3 : "Three"}  
print dictionary[1]  
dictionary[1] = "UpdatedOne"  
print dictionary[1]
```

7.17 Add a New Entry to an Existing Dictionary

Adding new key/value pairs in Python's dictionary is also similar to updating values in Python's list, but instead of an index, you would provide a new key and then assign it a new value. Please take a look at Example 7.17.1.

EXAMPLE 7.17.1

Java:

```
HashMap<Integer, String> dictionary = new HashMap<Integer, String>();  
dictionary.put(1, "One");  
dictionary.put(2, "Two");  
dictionary.put(3, "Three");  
System.out.println(dictionary.get(1));  
dictionary.put(4, "Four");  
System.out.println(dictionary.get(4));
```

Python:

```
dictionary = {1 : "One", 2 : "Two", 3 : "Three"}
```

```
print dictionary[1]
dictionary[4] = "Four"
print dictionary[4]
```

7.18 Emptying a Dictionary (Removing All Its Values)

Emptying a Java map and a Python dictionary uses exactly the same procedure. Both languages provide the method, *clear()*, that will delete all key/value pairs from a given dictionary data structure. Example 7.18.1 demonstrates how it is done in both languages.

EXAMPLE 7.18.1

Java:

```
HashMap<Integer, String> dictionary = new HashMap<Integer, String>();
dictionary.put(1, "One");
dictionary.put(2, "Two");
dictionary.put(3, "Three");
dictionary.clear();
```

Python:

```
dictionary = {1 : "One", 2 : "Two", 3 : "Three"}
dictionary.clear()
```

7.19 Delete an Entire Dictionary

Deleting an entire dictionary in Python requires a different process compared to in Java. Whereas in Java you would normally give the given map object a null value and garbage collector will pick it up and get rid of it, in Python the *del* function will delete whole dictionary object. Please take a look at Example 7.19.1.

EXAMPLE 7.19.1

Java:

```
HashMap<Integer, String> dictionary = new HashMap<Integer, String>();  
dictionary = null;
```

Python:

```
dictionary = {1 : "One", 2 : "Two", 3 : "Three"}  
del dictionary
```

7.20 Get the Size of a Dictionary

The size of a dictionary is returned using exactly the same Python function as in getting the size of an array (see 7.4 Getting Size of the Array): *len()*. Example 7.20.1 demonstrates how to calculate the size of a Python dictionary.

EXAMPLE 7.20.1

Java:

```
HashMap<Integer, String> dictionary = new HashMap<Integer, String>();  
dictionary.put(1, "One");  
dictionary.put(2, "Two");  
dictionary.put(3, "Three");  
dictionary.size();
```

Python:

```
dictionary = {1 : "One", 2 : "Two", 3 : "Three"}  
len(dictionary)
```

7.21 Getting Keys and Values of a Dictionary

There are three ways to get the list of all keys and values, separately or together, that reside in a given dictionary.

1. `dictionary.items()` returns the list of *(key, value)* pairs of the given dictionary.
2. `dictionary.keys()` returns the list of all keys of the given dictionary.
3. `dictionary.values()` returns the list of all values of the given dictionary.

Example 7.21.1 demonstrates how all three methods look in practice in both Java and Python.

EXAMPLE 7.21.1

Java:

```
HashMap<Integer, String> dictionary = new HashMap<Integer, String>();  
dictionary.put(1, "One");  
dictionary.put(2, "Two");  
dictionary.put(3, "Three");  
dictionary.entrySet();  
dictionary.keySet();  
dictionary.values();
```

Python:

```
dictionary = {1 : "One", 2 : "Two", 3 : "Three"}  
dictionary.items()  
dictionary.keys()  
dictionary.values()
```

7.22 Python Dictionary vs. JSON.

You may have noticed by now that the format of a Python dictionary is very similar to the JSON format. You are right. In the following example, I created a dictionary. I then converted it to a string, and then if you save that string to a text file named *filename.json*,

you will basically get the JSON file. Now, there is a JSON library in Python that is designed for working with data in JSON format, but Example 7.22.1 demonstrates one additional trick that could be just as useful.

Although Python's dictionary is similar to JSON in format, the two aren't completely identical. One of the differences, for example, is that JSON does not support Python values such as *None* but supports values that Python's dictionary does not support, such as *null*. Example 7.22.1 will help you form associations of Python's dictionary to something that you are likely already familiar with from working in Java. In a later section, 9.8 Writing JSON, I will discuss how to correctly write JSON file in Python.

EXAMPLE 7.22.1

Python:

```
dictionary = {1 : "ValueOne", "KeyTwo" : [276, "ten"], 3 : "valueThree", 4 : {1 :  
"One", 2 : "Two", 3 : "Three"}}  
jsonString = str(dictionary)  
print jsonString
```

7.23 Sets

Just as a *set* data structure in Java, a Python *set* is also basically an array that doesn't and cannot have any duplicate values. The way sets are created in Python is akin to the conversion of one type of Python's list into another. An array can be converted into a set what in part will remove all duplicate values from it, leaving only one of each values. A tuple can also be converted into a set, and, just as with arrays, all duplicates will be removed, leaving only one of each value. Sets can also be created by replacing the square parentheses of the array with curly parentheses, as those used to create Python's dictionary. Sets are as free as arrays in Python. They can hold any combination of values of any data types, but without duplicates.

EXAMPLE 7.23.1

Java:

```
Set<Integer> set = new HashSet<Integer>();
```

```
set.add(1);  
set.add(2);  
set.add(3);
```

Python:

Python's Set created out of the array:

```
setOne = set([1, 2, 3])
```

Python's Set created out of a **tuple**:

```
setTwo = set((1, 2, 3))
```

Python's Set created **with** curly parenthesis:

```
setThree = {1, 2, 3}
```

Modification to a set—such as adding, removing, or editing elements—is done in the same way as for a Python list. Therefore, I will skip these details and move on to some cool things that can be done to a set in Python but cannot be done in Java.

The Python function *difference()* returns the difference of two or more sets as a new set. Please take a look at Example 7.23.2.

EXAMPLE 7.23.2

Python:

```
setOne = set([1, 2, 3])  
setTwo = set((2, 3, 4))  
setThree = {1, 3, 5}
```

#Will return values that exist in setOne but not setTwo.

```
whatNotInSetTwo = setOne.difference(setTwo)
```

#Will return list of values that exist in setThree but not in setTwo or setOne. If, for example, value exists in one of the pairs of setTwo or setOne, that value will not be returned.


```
setThree.difference(setTwo).difference(setOne)
```

The function *difference_update()* will remove all elements of another set from the first set. Basically, *setOne.difference_update(setTwo)* is identical to *setOne = setOne - setTwo*.

EXAMPLE 7.23.3

Python:

```
setOne = set([1, 2, 3])
setTwo = set((2, 3, 4))
setThree = {1, 3, 5}
```

```
#Will remove values 2 and 3 from setOne.
setOne.difference_update(setTwo)
```

The next function, *intersection()*, will return a set of elements that match between two sets.

EXAMPLE 7.23.4

Python:

```
setOne = set([1, 2, 3])
setTwo = set((2, 3, 4))
setThree = {1, 3, 5}
```

```
#Will return a set with values 2 and 3.
setFour = setOne.intersection(setTwo)
```

7.24 Frozen Set

Frozen set is an immutable version of Python's set data structure. While elements can be modified in a set, they can not be modified in a frozen set. Frozen set is also iterable, just as a set. Please take a look at Example 7.24.1.

EXAMPLE 7.24.1

Python:

```
array = ["wordOne", "wordTwo", "wordTree", "wordOne"]  
frozenSet = frozenset(array)
```

7.25 Stacks

Stacks can be implemented just as well in Python as in Java. In the following example, Example 7.25.1, you can see my implementation of stack using Python. And the rule of the stack? Last in, first out. I didn't include the implementation of stack using Java because that would be familiar to you, a seasoned Java programmer.

EXAMPLE 7.25.1

Python:

```
class Stack():  
    def __init__(self):  
        self.__stack = []  
  
    def isEmpty(self):  
        return len(self.__stack) == 0  
  
    def push(self, object):  
        self.__stack.append(object)  
  
    def pop(self):  
        if self.isEmpty():
```

```

        return 0
    else:
        objectToReturn = self.__stack[len(self.__stack) - 1]
        self.__stack.pop(len(self.__stack) - 1)
        return objectToReturn

def peek(self):
    return self.__stack[len(self.__stack) - 1]

def search(self, object):
    return self.__stack.index(object)

def size(self):
    return len(self.__stack)

```

7.26 Queue

Queue is a very useful data structure not only in Java, but also in Python. I use it often and include my implementation of the queue data structure below in Example 7.26.1. The rule of the queue? First in, first out.

EXAMPLE 7.26.1

Python:

```

class Queue():
    def __init__(self):
        self.__queue = []

    def isEmpty(self):
        return len(self.__queue) == 0

    def push(self, object):
        self.__queue.append(object)

```

```

def pop(self):
    if self.isEmpty():
        return 0
    else:
        objectToReturn = self.__queue[0]
        self.__queue.pop(0)
        return objectToReturn

def peek(self):
    return self.__queue[0]

def search(self, object):
    return self.__queue.index(object)

def size(self):
    return len(self.__queue)

```

7.27 Linked List

There is a little more functionality to the linked list compared to the stack or queue data structures. Linked lists come in two forms, Singly Linked List and Doubly Linked List. In Example 7.27.1 below, I show my implementation of the Singly Linked List using Python, which stretches across five pages of this material. Example covers most of the important features of the Singly Linked List.

EXAMPLE 7.27.1

Python:

#Beginning of Linked List Implementation.

```

class Node():

```

```
def __init__(self, data, next):  
    self.data = data  
    self.next = next
```

```
def getData(self):  
    return self.data
```

```
def setData(self, value):  
    self.data = value
```

```
def getNext(self):  
    return self.next
```

```
def setNext(self, newNext):  
    self.next = newNext
```

```
class SinglyLinkedList():
```

```
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.currentSize = 0
```

```
    def add(self, value):  
        if(self.isEmpty()):  
            self.head = Node(value, None)  
            self.tail = self.head  
        else:  
            self.tail.setNext(Node(value, None))  
            self.tail = self.tail.getNext()  
        self.currentSize += 1  
        return True
```

```
    def isEmpty(self):  
        return self.size() == 0  
    def size(self):
```

```
return self.currentSize
```

```
def get(self, index):  
    if index < 0 or index >= self.currentSize:  
        raise IndexError  
  
    if index < self.currentSize - 1:  
        current = self.head  
        for i in xrange(index):  
            current = current.getNext()  
        return current.getData()  
    return self.tail.getData()
```

```
def clear(self):  
    self.head = None  
    self.tail = None  
    self.currentSize = 0
```

```
def removeByIndex(self, index):  
    if index < 0 or index >= self.currentSize:  
        raise IndexError  
  
    current = self.head  
    for i in xrange(0, self.size()-1):  
        last = current  
        current = current.getNext()  
        if i >= index and i < self.size():  
            last.setData(current.getData())  
    self.tail = last  
    self.currentSize -= 1  
    return True
```

*# Will delete element if part of it
contains the provided value.*

```
def removeValueContains(self, value):  
    current = self.head
```

```

for i in xrange(0, self.size()-1):
    last = current
    current = current.getNext()
    if str(last.getData()).lower().__contains__(str(value).lower()):
        for j in xrange(i, self.size()-1):
            last.setData(current.getData())
            last = current
            current = current.getNext()
        self.tail = last
        self.currentSize -= 1
        return True
raise ValueError
return False

```

*# Will delete element if it equals
to the provided value. Exact match.*

```

def removeValueEquals(self, value):
    current = self.head
    for i in xrange(0, self.size()-1):
        last = current
        current = current.getNext()
        if last.getData() == value:
            for j in xrange(i, self.size()-1):
                last.setData(current.getData())
                last = current
                current = current.getNext()
            self.tail = last
            self.currentSize -= 1
            return True
    raise ValueError
return False

```

*# Will return True if element in the Linked List
equals to the provided value. Exact match.*

```

def containsExactMatch(self, value):
    current = self.head
    for i in xrange(0, self.size()-1):
        last = current

```

```
    current = current.getNext()
    if last.getData() == value:
        return True
    return False
```

*# Will return True if part of some element in the Linked List
contains the provided value.*

```
def containsWithin(self, value):
    current = self.head
    for i in xrange(0, self.size()-1):
        last = current
        current = current.getNext()
        if str(last.getData()).lower().__contains__(str(value).lower()):
            return True
    return False
```

```
def toArray(self):
    list = []
    current = self.head
    for i in xrange(self.size()):
        last = current
        current = current.getNext()
        list.append(last.getData())
    return list
```

*# To preserve the Pythonic way to iterate through collections,
iterate basically converts linked list to array, which you
can iterate through using Python's for loop.*

Example:
"for item in myList.iterate():
print item"

```
def iterate(self):
    return self.toArray()
```

```
def indexOf(self, value):
    current = self.head
```



```

for i in xrange(self.size()):
    last = current
    current = current.getNext()
    if last.getData() == value:
        return i
raise ValueError

```

```

def lastIndexOf(self, value):
    current = self.head
    toReturn = -1
    for i in xrange(self.size()):
        last = current
        current = current.getNext()
        if last.getData() == value:
            toReturn = i
    if toReturn >= 0:
        return toReturn
    else:
        raise ValueError

```

```

def set(self, index, value):
    if index < 0 or index > self.currentSize:
        raise IndexError
    if index == self.currentSize:
        self.add(value)
        return
    current = self.head
    for i in xrange(self.size()):
        last = current
        current = current.getNext()
        if i == index:
            last.setData(value)

```

```

class ValueError(Exception):
    pass

```

#End of Linked List Implementation.

7.28 Binary Trees

Ah, the almighty binary tree. As complex as this data structure might seem, it is possible to implement it using Python. What you see below is my implementation of a binary tree, which should provide a good example of how to implement binary trees in Python.

Rules of binary tree? The left node should always be smaller than the parent node and the right node should always be larger than the parent node. The very first parent node is the root node. I have also experimented by adding two of my own features: Tree To List, which will return the list of values from the tree, and List To Tree, which will generate a binary tree from the values of the provided list. Example 7.28.1 contains my four-page implementation of the binary tree data structure using Python.

EXAMPLE 7.28.1

Python:

#Beginning of Binary Tree Implementation.

```
class Node():
    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None
        self.parent = None

    def getLeftChild(self):
        return self.leftChild

    def getRightChild(self):
        return self.rightChild

    def getData(self):
        return self.data

    def getParent(self):
```

```
return self.parent
```

```
def setLeftChild(self, newLeftChild):  
    self.leftChild = newLeftChild
```

```
def setRightChild(self, newRightChild):  
    self.rightChild = newRightChild
```

```
def setData(self):  
    return self.data
```

```
def setParent(self, parent):  
    self.parent = parent
```

```
class Tree():
```

```
    def __init__(self):  
        self.root = None  
        self.__containsBool = False  
        self.__findNodeObj = None  
        self.__listObj = []
```

```
    def add(self, data):  
        if self.root == None:  
            self.root = Node(data)  
        else:  
            self.__traverseAndAdd(self.root, Node(data))
```

```
    def __traverseAndAdd(self, node, nodeToAdd):  
        if node.getData() >= nodeToAdd.getData():  
            if node.getLeftChild() == None:  
                node.setLeftChild(nodeToAdd)  
                node.getLeftChild().setParent(node)  
            else:  
                self.__traverseAndAdd(node.getLeftChild(), nodeToAdd)
```

```
        if node.getData() < nodeToAdd.getData():  
            if node.getRightChild() == None:
```

```
node.setRightChild(nodeToAdd)
node.getRightChild().setParent(node)
else:
    self.__traverseAndAdd(node.getRightChild(), nodeToAdd)
```

```
def remove(self, value):
    if self.root != None:
        node = self.findNode(value)
    else:
        return None
    parentNode = node.getParent()
    if node.getLeftChild() != None and node.getRightChild() != None:
        if node.getData() < parentNode.getData():
            parentNode.setLeftChild(node.getRightChild())
            self.__traverseAndAdd(self.root, node.getLeftChild())
        else:
            parentNode.setRightChild(node.getLeftChild())
            self.__traverseAndAdd(self.root, node.getRightChild())
    elif node.getLeftChild() != None and node.getRightChild() == None:
        if node.getData() < parentNode.getData():
            parentNode.setLeftChild(node.getLeftChild())
        else:
            parentNode.setRightChild(node.getLeftChild())
    elif node.getLeftChild() == None and node.getRightChild() != None:
        if node.getData() < parentNode.getData():
            parentNode.setLeftChild(node.getRightChild())
        else:
            parentNode.setRightChild(node.getRightChild())
    elif node.getLeftChild() == None and node.getRightChild() == None:
        if node.getData() < parentNode.getData():
            parentNode.setLeftChild(None)
        else:
            parentNode.setRightChild(None)
```

```
def findNode(self, value):
    if self.root != None:
        self.__getNodeTraversal(self.root, value)
    else:
```

```
    return None
returnNodeObj = self.__findNodeObj
self.__findNodeObj = None
print returnNodeObj.getData()
return returnNodeObj
```

```
def __getNodeTraversal(self, node, value):
    if node != None:
        if node.getData() == value:
            self.__findNodeObj = node
        self.__getNodeTraversal(node.getLeftChild(), value)
        self.__getNodeTraversal(node.getRightChild(), value)
```

```
def contains(self, value):
    if self.root != None:
        self.__containsTraversal(self.root, value)
    returnBool = self.containsBool
    self.__containsBool = False
    print returnBool
    return returnBool
```

```
def __containsTraversal(self, node, value):
    if node != None:
        if node.getData() == value:
            self.containsBool = True
        self.__containsTraversal(node.getLeftChild(), value)
        self.__containsTraversal(node.getRightChild(), value)
```

```
def traverse(self, traversalType):
    if self.root != None:
        if traversalType.replace(" ", "").lower() == "preorder":
            self.__preOrderTraverse(self.root)
        elif traversalType.replace(" ", "").lower() == "inorder":
            self.__inOrderTraverse(self.root)
        elif traversalType.replace(" ", "").lower() == "postorder":
            self.__postOrderTraverse(self.root)
```

```
def __preOrderTraverse(self, node):  
    if node != None:  
        print node.getData()  
        self.__preOrderTraverse(node.getLeftChild())  
        self.__preOrderTraverse(node.getRightChild())
```

```
def __inOrderTraverse(self, node):  
    if node != None:  
        self.__inOrderTraverse(node.getLeftChild())  
        print node.getData()  
        self.__inOrderTraverse(node.getRightChild())
```

```
def __postOrderTraverse(self, node):  
    if node != None:  
        self.__postOrderTraverse(node.getLeftChild())  
        self.__postOrderTraverse(node.getRightChild())  
        print node.getData()
```

```
def toList(self):  
    if self.root != None:  
        self.__toListTraversal(self.root)  
    returnObj = self.__listObj  
    self.__listObj = []  
    return returnObj
```

```
def __toListTraversal(self, node):  
    if node != None:  
        self.__toListTraversal(node.getLeftChild())  
        self.__listObj.append(node.getData())  
        self.__toListTraversal(node.getRightChild())
```

```
def listToTree(self, list):  
    for item in list:
```

```
self.add(item)
```

#End of Binary Tree Implementation.

7.29 Graphs

Graph is the one of the most complex data structures to understand and to implement. Even then, it is very much implementable in Python, a powerful object-oriented programming language. Having worked extensively with Java, I think that out of all newer programming languages out there, only Python can compete with Java and may become a language of choice that would replace Java altogether. My humble opinion. In any case, my implementation of a Java-like graph data structure is shown in Example 7.29.1.

EXAMPLE 7.29.1

Python:

#Beginning of Graph Implementation.

```
class Node():
```

```
    def __init__(self, data):
        self.data = data
        self.next = None
        self.edges = []
```

```
    def setNext(self, next):
        self.next = next
```

```
    def getNext(self):
        return self.next
```

```
    def setData(self, data):
        self.data = data
```

```
    def getData(self):
        return self.data
```

```
    def addEdge(self, edge):
```

```
self.edges.append(edge)
```

```
def getEdges(self):  
    return self.edges
```

```
def getSelf(self):  
    return self
```

```
class Graph():
```

```
    def __init__(self):  
        self.root = None
```

```
    def addNode(self, value):  
        if self.root == None:  
            print "Im insodeeee"  
            self.root = Node(value)  
        else:  
            self.__findLastNode(self.root, value)
```

```
    def __findLastNode(self, node, value):  
        if node.getNext() != None:  
            self.__findLastNode(node.getNext(), value)  
        else:  
            print node.getData()  
            return node.setNext(Node(value))
```

```
    def addEdge(self, valueOfStartNode, valueOfEndNode):  
        startNode = self.__findNode(self.root, valueOfStartNode)  
        endNode = self.__findNode(self.root, valueOfEndNode)  
        self.foundNode = None  
        if startNode != Node and endNode != None:  
            startNode.addEdge(endNode)  
            return True  
        else:  
            return False
```



```

def __findNode(self, node, value):
    self.foundNode = None
    if node.getNext() != None:
        if str(node.getData()).lower() == str(value).lower():
            self.foundNode = node
        else:
            self.parentNode = node
            self.__findNode(node.getNext(), value)
    return self.foundNode

```

```

def printConnectionsOfAllNodes(self):
    if self.root == None:
        print "Nothing to print, first add Root Node"
    else:
        self.__iterateNodes(self.root)

```

```

def __iterateNodes(self, node):
    if node.getNext() != None:
        print str(node.getData()) + str(node.getEdges())
        self.__iterateNodes(node.getNext())

```

```

def isReachable(self, valueOfStartNode, valueOfEndNode):
    self.numOfEdges = 0
    self.isReachableBool = False
    self.lookedAt = []
    startNode = self.__findNode(self.root, valueOfStartNode)
    endNode = self.__findNode(self.root, valueOfEndNode)
    if startNode != None:
        print "\nstartNode: "+startNode.getData()
        print "endNode: "+endNode.getData()
        self.__findTheEdgeConnection(startNode, endNode)
        print "isReachable: "+str(self.isReachableBool)
        if self.isReachableBool is True:
            print "Took "+str(self.numOfEdges) + " edge(s) to reach endNode"
            return self.isReachableBool
    else:
        return self.isReachableBool

```

```

def __findTheEdgeConnection(self, startNode, endNode):
    listOfEdges = startNode.getEdges()
    if endNode in startNode.getEdges():
        self.isReachableBool = True
    else:
        for item in listOfEdges:
            subListOfEdges = item.getEdges()
            if len(item.getEdges()) > 0:
                for subItem in subListOfEdges:
                    self.__findTheEdgeConnection(subItem, endNode)
    self.numOfEdges += 1

```

```

def delete(self, value):
    nodeObj = self.__findNode(self.root, value)
    print nodeObj.getData()
    print self.parentNode.getData()
    self.parentNode.setNext(nodeObj.getNext())

```

```

def __iterateAndDeleteEdges(self, currentNode, nodeToDelete):
    if currentNode.getNext() != None:
        if nodeToDelete in currentNode.getEdges():
            currentNode.getEdges().remove(nodeToDelete)
            self.__iterateAndDeleteEdges(currentNode.getNext(), nodeToDelete)

```

#End of Graph Implementation.

8. MULTITHREADING AND MULTIPROCESSING

Java lets you run separate processes in parallel using Java's Thread Model. Python also gives you that ability, but in two variations unique to each other: "multithreading" and "multiprocessing." The difference between the two lies in how separate processes share virtual memory where they run.

8.1 Multithreading

Multithreading runs within the same memory and shares it based on availability. For example, if one thread took 80% of available memory, then only 20% is left to run other processes until more memory out of the occupied 80% becomes available. Example 8.1.1 demonstrates the implementation of the multithreading in Java and in Python. I kept the class and method names the same between Java and Python examples for you compare how the two languages implement multithreading.

EXAMPLE 8.1.1

Java:

```
class SomeClass {  
    public static void main(String args[]) {  
        new ThreadNode("First");  
        new ThreadNode("Second");  
        new ThreadNode("Third");  
        new ThreadNode("Fourth");  
    }  
}  
  
class ThreadNode implements Runnable {  
    String threadName;  
    Thread threadObj;
```

```

ThreadNode(String threadName) {
    String name = threadName;
    threadObj = new Thread(this, name);
    threadObj.start();
}

public void run() {
    for (int counter = 5; counter > 0; counter--) {
        System.out.println(threadName + ": " + counter);
    }
}

```

Python:

```

import threading

class ThreadNode(threading.Thread):

    def __init__(self, threadName):
        threading.Thread.__init__(self)
        self.threadName = threadName

    def run(self):
        for counter in range(5):
            print self.threadName + ": " + str(counter)

class SomeClass():
    def __init__(self):
        thread1 = ThreadNode("First")
        thread1.start()
        thread2 = ThreadNode("Second")
        thread2.start()
        thread3 = ThreadNode("Third")
        thread3.start()
        thread4 = ThreadNode("Fourth")
        thread4.start()

```

SomeClass()

8.2 Synchronization in Multithreading

Python's *threading* module provides the concept that a *Lock()* can lock out the rest of the processes and line them up in the queue. All consequent threads execute only after the current one has completed its execution. Before execution, the thread needs to *acquire()* its reserved spot in the queue. When the execution is completed, the thread has to *release()* itself, opening a pathway for the next thread in line. Example 8.2.1 demonstrates exactly how that is done in Python.

EXAMPLE 8.2.1

Python:

```
import threading

class ThreadNode(threading.Thread):

    def __init__(self, threadName, lockObj):
        threading.Thread.__init__(self)
        self.threadName = threadName
        self.lockObj = lockObj

    def run(self):
        self.lockObj.acquire()
        for counter in range(5):
            print self.threadName + ": " + str(counter)
        self.lockObj.release()

class SomeClass():
    def __init__(self):
```

```

lockObj = threading.Lock()

thread1 = ThreadNode("First", lockObj)
thread1.start()
thread2 = ThreadNode("Second", lockObj)
thread2.start()
thread3 = ThreadNode("Third", lockObj)
thread3.start()
thread4 = ThreadNode("Fourth", lockObj)
thread4.start()

```

SomeClass()

8.3 Multiprocessing

Multiprocessing splits memory evenly among the processes that are to run in parallel. Each process then runs within its own virtual memory and does not take memory away from other processes. If, for example, you run four processes on a quad core CPU, then each process would run on its own separate core without violating the cores of other processes. In Python, multiprocessing can be implemented using multiple ways. I specified two ways in Examples 8.3.1 and 8.3.2. Both work well and demonstrate the flexibility of Python.

EXAMPLE 8.3.1

Python:

```

from multiprocessing import Process

```

```

class ThreadNode:

```

```

    def __init__(self, threadName):
        self.threadName = threadName
        Process(target=self.run, args=()).start()

```

```

    def run(self):
        for counter in range(5):

```

```
print self.threadName + ": " + str(counter)
```

```
class SomeClass():  
    def __init__(self):  
        ThreadNode("First")  
        ThreadNode("Second")  
        ThreadNode("Third")  
        ThreadNode("Fourth")
```

```
SomeClass()
```

EXAMPLE 8.3.2

Python:

```
from multiprocessing import Process
```

```
class ThreadNode:  
    def run(self, threadName):  
        for counter in range(5):  
            print threadName + ": " + str(counter)
```

```
class SomeClass():  
    def __init__(self):  
        threadObj = ThreadNode()  
        Process(target=threadObj.run, args=("First",)).start()  
        Process(target=threadObj.run, args=("Second",)).start()  
        Process(target=threadObj.run, args=("Third",)).start()  
        Process(target=threadObj.run, args=("Fourth",)).start()  
SomeClass()
```

8.4 Synchronization in Multiprocessing

Python's *multiprocessing* module also provides the concept that *Lock()* works in a similar fashion as the *Lock()* of the *threading* module. *Lock()* can prevent all consequent processes from execution and line them up in the queue for execution only after the current one has completed its execution. Before execution, the process need to *acquire()* its reserved spot in the queue. When that execution is completed, the process then has to *release()* itself, opening a door for the next process in line to enter and start its execution. Example 8.4.1 demonstrates exactly how that is done in Python. Please note that from the *multiprocessing* module, I imported *Process* and *Lock* classes.

EXAMPLE 8.4.1

Java:

```
from multiprocessing import Process, Lock

class SomeClass():
    def __init__(self):
        lockObj = Lock()

        intVariable = 1000
        for num in range(10):
            Process(target=self.ourProcess, args=(lockObj, intVariable)).start()

    def ourProcess(self, lockObj, intVariable):
        lockObj.acquire()
        print 'Printing my Integer: ' + str(intVariable)
        lockObj.release()

SomeClass()
```


9. I/O

9.1 Reading Console Input

Reading console input can be handy sometimes, although there may only be rare occasions for it, from my experience. The examples below demonstrate how to achieve this in both Java and Python. Please note that in the Python example I have converted the input into an integer for validation purpose. That demonstrates how the program can validate provided input and, in my case, make sure that the input is indeed an Integer. Java does that automatically, but in Python I have to do a little extra work to achieve that. Please take a look at Example 9.1.1.

EXAMPLE 9.1.1

Java:

```
import java.util.Scanner;

public class SomeClass {
    public static void main(String [] args) {

        Scanner in = new Scanner(System.in);

        int intVar = in.nextInt();
        String strVar = in.next();
        System.out.println("Printing int: " + intVar);
        System.out.println("Printing string: " + strVar);
    }
}
```

Python:

```
class SomeClass():
    def __init__(self):
```

```

try:
    intVar = int(raw_input('Enter Integer Value:'))
except ValueError:
    print "Not an Integer."

strVar = raw_input('Enter String Value:')

print "Printing int: " + str(intVar)
print "Printing string: " + strVar

```

SomeClass()

9.2 Reading from Files

Example 9.2.1 demonstrates the syntax differences between Java and Python when reading content from files. Both examples will grab each line of a text file's content and output it one by one into a console.

EXAMPLE 9.2.1

Java:

```

import java.io.*;
class ReadFileContent {
    public static void main(String args[]) {
        File fileToRead = new File("/path/to/file/filename.txt");
        BufferedReader reader = new BufferedReader (new FileReader(fileToRead));
        while (reader.readLine() != null) {
            String data = reader.readLine();
            System.out.println(data);
        }
        reader.close();
    }
}

```

Python:

```
class ReadFileContent():  
    def __init__(self):  
        fileToRead = open("/path/to/file/filename.txt", "r")  
        for line in fileToRead.readlines():  
            print line  
        fileToRead.close()
```

ReadFileContent()

9.3 How to Avoid Slurping While Reading Content of Large Files

Let's start with defining what slurping is. Slurping is reading the content of the whole file at once into virtual memory, then working with it out of the virtual memory. It is the fastest way to read the content of a file, but what if the size of this content is larger than the size of your virtual memory? Then slurping will cause an out-of-memory exception.

In Python, *file.read()* and *file.readlines()* both slurp the whole file into memory. In cases when slurping a full file is not an option, Python provides a way to read the file line by line very simply, such as in Example 9.3.1.

EXAMPLE 9.3.1

Python:

```
class ReadFileContentWihtoutSlurping():  
    def __init__(self):  
        sourceFile = "/path/to/file/filename.txt"  
        with open(sourceFile, "r") as file:  
            for line in file:  
                print line
```

ReadFileContentWihtoutSlurping()

9.4 Writing into Files

Writing into files using Python is accomplished almost the same way as reading from files. Example 9.4.1 shows how it's done. Note that Example 9.4.1 overwrites existing file content every time it is executed. Also note that if a file doesn't exist, the program will create a file, but only if the specified path to the file exists. If one or more directories do not exist, the following program will throw an exception.

EXAMPLE 9.4.1

Java:

```
import java.io.*;
class WriteFileContent {
    public static void main(String args[]) {
        String fileNameWithPath = "/path/to/file/filename.txt";
        File fileToWrite = new File(fileNameWithPath);
        if (!fileToWrite.exists()) {
            fileToWrite.createNewFile();
        }
        PrintWriter writer = new PrintWriter(fileNameWithPath, "UTF-8");
        writer.println("text to write");
        writer.close();
    }
}
```

Python:

```
class WriteFileContent():
    def __init__(self):
        fileToWrite = open("filename.txt", "w")
        fileToWrite.writelines("text to write")
        fileToWrite.close()
```

WriteFileContent()

9.5 Appending into Files

When writing into files, often you need to append lines of text into the same file while preserving the file's existing content. Using Example 9.4.1 and changing only one character for the second argument of function *open()*, I can start appending into a file instead of overwriting it every time I access it. Please take a look at Example 9.5.1. the second argument of function *open()* is set to *a*, which in this case stands for “append”.

EXAMPLE 9.5.1

Python:

```
class WriteFileContent():
    def __init__(self):
        fileToRead = open("filename.txt", "a")
        fileToRead.writelines("text to write")
        fileToRead.close()
```

WriteFileContent()

9.6 Checking Path Existence

To prevent a program from throwing an exception, if the path to a file that is being read from or written into doesn't exist, it's helpful to first perform a check for the existence of the specified path. To make that possible, first I need to import the *os* module and then use its *path.exists* function, which will return a boolean value, letting me know whether a path exists. Example 9.6.1 demonstrates the implementation of this check.

EXAMPLE 9.6.1

Python:

```
import os

class SomeClass():
    def __init__(self):
        print os.path.exists("/path/to/bla/bla/")

SomeClass()
```

Example 9.6.2 demonstrates how to check for path existence in combination with writing into a file if a path does exist.

EXAMPLE 9.6.2

Python:

```
class WriteFileContent():
    def __init__(self):
        pathToFile = "/path/to/file/"
        fileName = "filename.txt"
        if os.path.exists(pathToFile):
            fileToRead = open(os.path.join(pathToFile, fileName), "w")
            fileToRead.writelines("text to write")
            fileToRead.close()

WriteFileContent()
```

9.7 Creating a Path to File

When you are trying to write into a file but the path doesn't exist, you can simply create it by adding all missing directories and sub directories of the missing path. It can be easily done in Python using the *os* module, as shown in Example 9.7.1. You may notice that I

moved out the code that writes text to file, into its own method and added an *else* to my condition, which checks for the existence of the provided path. If the path doesn't exist, it will create all directories that are missing in the path. Function *os.makedirs(pathToFile)* works perfectly for that and is demonstrated in Example 9.7.1.

EXAMPLE 9.7.1

Python:

```
class WriteFileContent():
    def __init__(self):
        pathToFile = "/path/to/file/"
        fileName = "filename.txt"
        if os.path.exists(pathToFile):
            self.write(pathToFile, fileName)
        else:
            os.makedirs(pathToFile)
            self.write(pathToFile, fileName)

    def write(self, pathToFile, fileName):
        fileToRead = open(os.path.join(pathToFile, fileName), "w")
        fileToRead.writelines("text to write")
        fileToRead.close()
```

WriteFileContent()

9.8 Reading JSON Files

To demonstrate the power and simplicity of Python, let's take a look at how to read from and write into a JSON format file. First I want to turn your attention to the format of Python's dictionary data structure and JSON's format. You are correct—they are almost identical. The only differences are that boolean values in Python start with capital letters (*True* or *False*), but in JSON they are lowercased (*true* or *false*), and JSON *null* value is *None* in Python's dictionary. Besides those two differences, the format is the same and could be iterated through similarly.

However, to convert JSON format data into Python's object you need to use a module called *json*. The name is very simple to remember and self-explanatory, just like everything else in Python. Module *json* first needs to be imported into the workspace where the method that is going to use it resides. In Example 9.8.1, method *getJsonContent()* converts a string in JSON format to a JSON object. JSON string could have been read from a file or passed to the method as an argument. Using *json* module also fixes the differences by converting Python's *None*, *True* and *False* into JSON's *null*, *true* and *false*.

EXAMPLE 9.8.1

Python:

```
import json

class WorkingWithJSON():

    def getJsonContent(self, jsonStr):
        jsonObj = json.loads(jsonStr)
        return jsonObj
```

In my next example, Example, 9.8.2, let's iterate through the keys of the JSON. As you can see, using a method of iteration very similar to how you would iterate through an array in Python, I can extract the list of keys out of the JSON object.

EXAMPLE 9.8.2

Python:

```
import json

class WorkingWithJSON():

    def getJsonContent(self, jsonStr):
        jsonObj = json.loads(jsonStr)
        for key in jsonObj:
            print key
```

And at last, let's extract the values that are attached to the list of keys in my JSON object.

EXAMPLE 9.8.3

Python:

```
import json

class WorkingWithJSON():

    def getJsonContent(self, jsonStr):
        jsonObj = json.loads(jsonStr)
        for key in jsonObj:
            #print outputs value of a given key
            print jsonObj[key]
```

Above, Example 9.8.3 demonstrates a simple way of iterating a JSON object. In Python there are other modules with powerful features designed specifically for working with JSON formats. Let's take a more complex case as an example.

Let's say I have a list where each element is a JSON string with multiple key and value pairs. I would like to sort the list of elements first by the *name* key values, then by *lastName* key values, and lastly by *age* key values. Example 9.8.4 demonstrates how to achieve this with ease in Python.

EXAMPLE 9.8.4

Python:

```
class WorkingWithJSON():

    def sortJsonContent(self):
        ourList = [{'name': 'Alex', 'age': 39, 'lastName': 'Simpson'},
                    {'name': 'Jack', 'age':40, 'lastName': 'Johnson'},
                    {'name': 'Jack', 'age':21, 'lastName': 'Beatles'}]
```

```
sortedList = sorted(ourList, key=itemgetter('name', 'lastName', 'age'))

return sortedList
```

9.9 Writing JSON Files

Writing JSON files is as easy as reading them. If you remember, in the previous chapter I mentioned that JSON format is almost identical to the format of Python's dictionary. The similarity is even more prominent when you are writing JSON files. To do that, first programmatically you need to build a Python dictionary in the format that you would like your JSON to be structured by and fill in values. Then, by simply passing the Python dictionary object to *json.dump*, you can convert the dictionary into a perfectly fine JSON string that is later saved into a *.json* file. In the following example, note that I added one value that is boolean and in Python's format where booleans are all capitalized. After the *.json* file is generated, even the Python Boolean value *True* is converted into the JSON format value, *true*. The same is done for Python's *None*, which is converted to JSON's *null*. Example 9.9.1 demonstrates how to properly write JSON files in Python.

EXAMPLE 9.9.1

Python:

```
import json
```

```
class WorkingWithJSON():
```

```
    def writeContentToJSON(self):
```

```
        jsonFilePath = "/path/to/file/our_csv_file.json"
```

```
        dataDictionary = {"key one": "value one", "key two": [1, 2, 3], "key three":
{"sub key three": ["list", "of", "values", True]}}
```

```
        with open(jsonFilePath, 'w') as jsonWrite:
```

```
            json.dump(dataDictionary, jsonWrite)
```

9.10 Reading CSV Files

CSV format is very popular, especially among big data developers, to use as feed export and injection format. Python has a great library that makes working with CSVs as simple as it can get. In the next example, Example 9.10.1, I demonstrated reading from a CSV file by providing the path to your CSV file and filename, and then return Python's CSV object that contains all content from the given CSV file. Just as with the *json* module, the *csv* module needs to be imported first. Please note that you can specify different delimiters—it could be comma (,), pipe (|), or tab (\t)—depending on what delimiter is used in the CSV file you are trying to read from.

EXAMPLE 9.10.1

Python:

```
import csv

class WorkingWithCSV():

    def getCSVContent(self):
        csvFilePath = "/path/to/file/our_csv_file.csv"
        with open(csvFilePath, 'rb') as csvFile:
            csvContent = csv.reader(csvFile, delimiter=',', quotechar='"')
            return csvContent
```

Example 9.10.2 demonstrates how to extract and read values of the *csvContent* object. As you can see in Example 9.10.2, I added iteration of *csvContent*, which outputs each row of the content, then I print it out into a console. If you run the following example, you will see that each row is an array of values from each row of the CSV file. Since each row is a list, you can extract its values just as you would extract elements from any array list by index.

EXAMPLE 9.10.2

Python:

```

import csv

class WorkingWithCSV():

    def getCSVContent(self):
        csvFilePath = "/path/to/file/our_csv_file.csv"
        with open(csvFilePath, 'rb') as csvFile:
            csvContent = csv.reader(csvFile, delimiter=',', quotechar='"')
            for row in csvContent:
                print row

```

There is more to it. Just as with JSON, in Python there are multiple powerful modules available to work with CSV format. The next example demonstrates module *pandas*, which solves a problem of grabbing the list of values of only one column out of a given CSV file. Let's say I have a CSV file that has column names specified at the header and I need to extract a list of values of only one specific column. The example below shows exactly how to achieve this in Python. Module *pandas* also first needs to be imported before it could be referenced and used in a program. Please take a look at Example 9.10.3.

EXAMPLE 9.10.3

Python:

```

import pandas

class WorkingWithCSV():

    def getCSVContent(self):
        csvFilePath = "/path/to/file/our_csv_file.csv"
        csvContent = pandas.read_csv(csvFilePath)
        column_values_list = csvContent['column_name']
        return column_values_list

```

9.11 Writing CSV Files

Writing into CSV can also be done with ease in Python using the same *csv* module. First, you need to aggregate all values into the array list. Then pass this array list object to the *csv.writer* and it will do the rest. It's as simple as it sounds. Please note that you can specify different delimiters, be it comma (,), pipe (|), or tab (\t). In Example 9.11.1, I specified a comma delimiter for the CSV file that I am trying to generate.

EXAMPLE 9.11.1

Python:

```
class WorkingWithCSV():
    def writeContentToCSV(self):
        csvFilePath = "/path/to/file/our_csv_file.csv"

        values = ["value-one", "value,two", 3, 44:44, "value five"]

        with open(csvFilePath, 'w') as csvWrite:
            csvWriter = csv.writer(csvWrite, delimiter=',')
            csvWriter.writerows(values)
```

9.12 Lambda Expressions

Almighty lambdas, the anonymous functions not bound to the name generated at the runtime. Lambdas are supported in Java starting from Java 8 and also in Python. In both languages the concept of lambda is pretty much the same. Syntax, however, is slightly different, but it's easy to understand if you know what lambdas are and how to use them in Java. Lambdas, by the way, can also be passed as arguments to other functions in both languages. Example 9.12.1 demonstrates the differences in syntax between the same lambda implemented in Java and in Python. It's a very simple example that returns a boolean value after comparing two strings.

EXAMPLE 9.12.1

Java:

boolean comparingStrings = (String strOne, String strTwo)-> strOne.equals(strTwo);

Python:

comparingStrings = **lambda** strOne, strTwo: strOne == strTwo

To run lambda function from Example 9.12.1 after it has been generated, you need to do the following. As part of Example 9.12.2 let's print out the returned result of above lambdas using Java and Python.

EXAMPLE 9.12.2

Java:

boolean comparingStrings = (String strOne, String strTwo)-> strOne.equals(strTwo);
System.**out**.println(comparingStrings("Hello", "World"));

Python:

comparingStrings = **lambda** strOne, strTwo: strOne == strTwo
print comparingStrings ("Hello", "World")

10. STRINGS

10.1 String Concatenation with Other Data Types

I decided to mention string concatenation of strings with other data types because the way this works in Python is completely opposite from how it works in Java. Concatenation itself is done using the same operator (+) in both Java and Python. Now let's move on to the differences.

In Java, concatenating any data type to string converts that value to a string. In Python, however, automatic conversion does not happen. To concatenate any data type to a string, the other data type needs to be first explicitly converted into a string. Please take a look at Example 10.1.1.

EXAMPLE 10.1.1

Java:

```
String strVar = "Hello World" + 9 + true;
```

Python:

```
strVar = "Hello World" + str(9) + str(True)
```

You may have noticed that in the Python version of Example 10.1.1, I used the `str()` function to convert 9 and a boolean value into a string. Without `str()`, Python will throw an exception.

The data type of the variable in Python can be checked using function `type()`. Example 10.1.2 will print data types of variables into a console. Values do have data types in Python, but Python assigns them automatically. In other words, 9 would be assigned the

integer data type, but “*Hello World*” would get string data type assigned.

EXAMPLE 10.1.2

Python:

```
strVar = "Hello World"
intVar = 9
boolVar = True
print type(strVar)
print type(intVar)
print type(boolVar)
```

10.2 Character Extraction

Python sees all strings as arrays of characters. Therefore, to extract a character out of a string you would use the same method as you would to extract an element out of an array. Example 10.2.1 demonstrates how I can get a string character by index.

EXAMPLE 10.2.1

Java:

```
String strVar = "Hello World";
char charVar = strVar.charAt(4);
```

Python:

```
strVar = "Hello World"
charVar = strVar[4]
```

To get a part of a string starting from the character at index 4 to the character at index 9, for example, you need to use Python’s so-called “slice notation.” Slice notation is basically a `:`, a colon character that separates the range of values from start to end. It will work the same way for any array or string in Python. Please take a look at Example 10.2.2.

EXAMPLE 10.2.2

Java:

```
String strVar = "Hello World";  
String subStr = strVar.substring(4, 9);
```

Python:

```
strVar = "Hello World"  
subStr = strVar[4:9]
```

subStr in this case will be assigned the value *o Wor*.

Example 10.2.3 demonstrates how to get all characters of the string after the character under index 4.

EXAMPLE 10.2.3

Java:

```
String strVar = "Hello World";  
String subStr = strVar.substring(4);
```

Python:

```
strVar = "Hello World"  
subStr = strVar[4:]
```

subStr in this case will be assigned the value *o World*.

Example 10.2.4 demonstrates how to subtract two characters from the end of a string.

EXAMPLE 10.2.4

Python:

```
strVar = "Hello World"  
subStr = strVar[:-2]
```

subStr in this case will be assigned the value *Hello Wor*.

Using *for* loop you can iterate through the characters of the string just as with the elements of any array in Python, such as in Example 10.2.5.

EXAMPLE 10.2.5

Python:

```
strVar = "Hello World"  
for charElement in strVar:  
    print charElement
```

10.3 String Comparison

To compare two strings in Python, you would use double equal sign. Methods *equals()* and *equalsIgnoreCase()* do not exist in Python as in Java, but they could easily be replicated. Example 10.3.1 demonstrates how to replicate Java's *equal()* in Python.

EXAMPLE 10.3.1

Java:

```
String strOne = "Hello World";  
String strTwo = "Hello World";  
boolean equalsOrNot = strOne.equals(strTwo);
```

Python:

```
strOne = "Hello World"  
strTwo = "Hello World"  
equalsOrNot = strOne == strTwo
```

Java's *equalsIgnoreCase()* can be mimicked in Python using the method shown in Example 10.3.2.

EXAMPLE 10.3.2

Java:

```
String strOne = "Hello World";  
String strTwo = "hello world";  
boolean equalsOrNot = strOne.equalsIgnoreCase(strTwo);
```

Python:

```
strOne = "Hello World"  
strTwo = "hello world"  
equalsOrNot = strOne.lower() == strTwo.lower()
```

10.4 *StartsWith()* and *EndsWith()*

These two are almost identical in Java and Python. Example 10.4.1 explains it all.

EXAMPLE 10.4.1

Java:

```
String strOne = "Hello World";  
String strTwo = "hello world";  
boolean startsWithOrNot = strOne.startsWith("He");
```

```
boolean endsWithOrNot = strTwo.endsWith("rld");
```

Python:

```
strOne = "Hello World"  
strTwo = "hello world"  
startsWithOrNot = strOne.startsWith("He")  
endsWithOrNot = strTwo.endsWith("rld")
```

10.5 Searching Strings

In Java, to search for the first occurrence of a word or a character inside a string, you would use *indexOf()*, and for the last occurrence you would use *lastIndexOf()*. In Python, the counterparts of these functions are *find()* for the first occurrence and *rfind()* for the last occurrence. Example 10.5.1 demonstrates them both in action.

EXAMPLE 10.5.1

Java:

```
String strOne = "Hello, Today's Wonderful World of People's Happiness.";  
int firstOccurence = strOne.indexOf("'s");  
int lastOccurence = strOne.lastIndexOf("'s");
```

Python:

```
strOne = "Hello, Today's Wonderful World of People's Happiness."  
firstOccurence = strOne.find("'s")  
lastOccurence = strOne.rfind("'s")
```

10.6 String Replace

In Java and Python, *replace()* plays exactly the same role and is even called the same way. Example 10.6.1 demonstrates these similarities.

EXAMPLE 10.6.1

Java:

```
String strOne = "Hello, Today's Wonderful World of People's Happiness.";
String strTwo = strOne.replace("Today", "Yesterday");
```

Python:

```
strOne = "Hello, Today's Wonderful World of People's Happiness."
strTwo = strOne.replace("Today", "Yesterday")
```

10.7 String *trim()* in Python

Java's *trim()* is called *strip()* in Python. Both serve exactly the same purpose of removing preceding and trailing spaces around the string. Example 10.7.1 shows *strip()* in action.

EXAMPLE 10.7.1

Java:

```
String strOne = " Hello, Today's Wonderful World of People's Happiness.";
String strTwo = strOne.trim();
```

Python:

```
strOne = " Hello, Today's Wonderful World of People's Happiness. "
strTwo = strOne.strip()
```

10.8 Changing the Case of Characters

Example 10.8.1 demonstrates how to convert all characters to uppercase or lowercase in Java and in Python.

EXAMPLE 10.8.1

Java:

```
String strOne = "Hello, Today's Wonderful World of People's Happiness.";
String strAllLowerCase = strOne.toLowerCase();
String strAllUpperCase = strOne.toUpperCase();
```

Python:

```
strOne = "Hello, Today's Wonderful World of People's Happiness."
strAllLowerCase = strOne.lower()
strAllUpperCase = strOne.upper()
```

10.9 Joining Strings

Just as Java, Python also has a *join* function that serves the same purpose and works in a similar fashion. Syntax is slightly different, but exactly that—only slightly. Please take a look at Examples 10.9.1.

EXAMPLE 10.9.1

Java:

```
String[] strArr = {"WordOne", "WordTwo", "WordThree", "WordFour",
"WordFive"};
String joinedStr = String.join("-", strArr);
```

Python:

```
strArr = ["WordOne", "WordTwo", "WordThree", "WordFour", "WordFive"]  
joinedStr = "-".join(strArr)
```

In both cases, variable *joinedStr* will contain the value *WordOne-WordTwo-WordThree-WordFour-WordFive*.

10.10 String Length

As I have already mentioned previously, Python sees all strings as arrays of characters. Therefore, to get the length of a string, you would use exactly the same method as you would to get the size of an array: using Python's function *len()*. Example 10.10.1 demonstrates this.

EXAMPLE 10.10.1

Java:

```
String strOne = "Hello, Today's Wonderful World of People's Happiness."  
int strLength = strOne.length();
```

Python:

```
strOne = "Hello, Today's Wonderful World of People's Happiness."  
strLength = len(strOne)
```

10.11 Reverse a String

There are two ways to reverse a string in Python, shown in Example 10.11.1, and both are

unique to Python. Method One uses a double slice notation, which is a very unique syntax in Python that has no equivalent in Java. Method Two reverses an array then joins that array of characters back to the single string.

EXAMPLE 10.11.1

Java:

```
String strOne = "Hello, Today's Wonderful World of People's Happiness.";
String reversedStr = new StringBuilder(strOne).reverse().toString();
```

Python:

Method One:

```
strOne = "Hello, Today's Wonderful World of People's Happiness."
reversedStr = strOne[::-1]
```

Method Two:

```
strOne = "Hello, Today's Wonderful World of People's Happiness."
reversedStr = ''.join(reversed(strOne))
```

11. SORTING AND SEARCHING ALGORITHMS

11.1 Insertion Sort

```
def insertionSort():  
    array = [6, 8, 2, 5, 6, 10, 25, 12, 6, 4, 4, 6]  
    for index, item in enumerate(array):  
        print item  
        print index  
        subIndex = index  
        while subIndex > 0 and array[subIndex-1] > item:  
            array[subIndex] = array[subIndex - 1]  
            subIndex -= 1  
  
        array[subIndex] = item  
  
    print array
```

11.2 Bubble Sort

```
def bubbleSort():  
    array = [6, 8, 2, 5, 6, 10, 25, 12, 6, 4, 4, 6]  
    for index, item in reversed(list(enumerate(array))):  
        for num in range(index):  
            if array[num] > array[num+1]:  
                temp = array[num]  
                array[num] = array[num+1]  
                array[num+1] = temp  
  
    print array
```

11.3 Selection Sort

```
def selectionSort():
    array = [6, 8, 2, 5, 6, 10, 25, 12, 6, 4, 4, 6]
    for index, item in enumerate(array):
        minimum = index
        for num in range(index, len(array)):
            if array[minimum] > array[num]:
                minimum = num
        temp = array[index]
        array[index] = array[minimum]
        array[minimum] = temp
    print array
```

11.4 Binary Search

```
def binarySearch():
    searchingForValue = 5
    array = [6, 8, 2, 5, 6, 10, 25, 12, 6, 4, 4, 6]
    lowIndex = 0
    highIndex = len(array)-1

    while lowIndex <= highIndex:
        middleIndex = (lowIndex+highIndex) / 2
        if array[middleIndex] < searchingForValue:
            lowIndex = middleIndex + 1
        elif array[middleIndex] > searchingForValue:
            highIndex = middleIndex - 1
        else:
            print "\nFound a Match for "+str(searchingForValue)+" at index "+str(middleIndex)
            lowIndex = highIndex + 1;
```


12. PYTHON'S MODULES, JAVA'S LIBRARIES

12.1 Installing Required Modules

Just like with Java, Python software programming engineers have developed and continue to develop a wealth of libraries that are packaged in the form of modules. In Java, libraries are downloaded and imported in the form of *.jar* files. In Python, modules are installed using package manager *pip*. The command to install a module is very simple: *pip install module_name*. Python's own website provides a complete Python Package Index that lists most, if not all, available modules for development in Python. Python itself comes prepackaged with lots of useful modules, but if you tried to import a certain module and received the error *ImportError: No module named paramiko*, that means module *paramiko* is not installed in your Python environment and needs to be installed. You can do so with the command *pip install paramiko*. I randomly chose the *paramiko* module to use for this example, but it is actually a great SFTP utility with many useful features. You can use the same procedures for other modules that are not installed into your Python environment. After installing them, you'll be able to use them without issues.

12.2 Packaging-Required Modules with Your Project

What makes Java so comfortable to work with is its inclusion of all required libraries with your project in the form of *.jar* files. This way when project is moved from one machine to another, all required libraries move right with it. It could be done manually by downloading and storing *.jar* files with the project or using Maven's *pom.xml*.

In Python, modules can be installed what makes them available for all projects running on that environment. Modules also can be packaged and included with the project, so if a project gets moved to an environment where required modules are not installed, they can still be imported without the need to install them.

To achieve that, a specific sequence of steps needs to be performed to package the module and store it with the project in its workspace.

First, you need to locate your installed module(s). When running the *pip install* command, all modules are installed and their installations are stored somewhere in the environment.

To find out where they are stored, I need to execute Python's command `import` to import the module path to where I would like it to locate in my environment. Normally, you need to do this for the modules that did not come prepackaged with the distribution of Python you are running on. Let's use one such module, called *paramiko*. First, let's install this module by running the following command:

```
>>> pip install paramiko
```

Next, import it by running this command:

```
>>> import paramiko
```

The next command will output the path to the installation of *paramiko* module in my environment.

```
>>> print paramiko.__file__
```

Output in my case would look the following way:

```
"/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-  
packages/paramiko/__init__.pyc"
```

Now, let's zip the directory that contains my installed module. The root directory of *paramiko* module is:

```
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-
```

packages/paramiko/.

I can easily zip it by running the Linux *zip* command, such as this:

```
>>> sudo zip -r zipped_ matplotlib.zip  
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/  
paramiko/
```

Now I have the *paramiko* module zipped and stored on my environment's drive under filename *zipped_paramiko.zip*. This file is my exported module, which I need to store somewhere inside the workspace of my project.

To import this module into my project and be able to use it, I need to use Python's module *sys*, which comes prepackaged as part of all Python distributions. Using *sys.path.insert()*, I can specify a path to the zipped module and then import it just as I would have imported it after installing it into environment. Example 12.2.1 demonstrates how to import a zipped module into a workspace.

EXAMPLE 12.2.1

Python:

```
import sys  
sys.path.insert(0, '/path/to/zipped/module/file/paramiko.zip')  
import paramiko  
  
class SomeClass():  
  
    def __init__(self):  
        port = 22  
        host = ""  
        username = ""  
        password = ""  
        transport = paramiko.Transport((host, port))  
        transport.connect(username = username, password = password)  
        sftp = paramiko.SFTPClient.from_transport(transport)  
        sftp.put("local_file", "sftp_path")
```

```
run = SomeClass()
```


13. RUNNING SHELL COMMANDS FROM PYTHON

13.1 Running a Single Shell Command

Very often you might need to execute a shell command out of your Python program, for example to access HDFS commands. Python provides a module, called *subprocess*, that is designed for that very purpose. The command can be passed as a string and you definitely need to read the logs, and not only *stdout*, but also *stderr*. The method in the next example demonstrates exactly that. It executes the shell command, and then redirects both *stdout* and *stderr* log streams into one stream, which it then captures and returns.

EXAMPLE 13.1.1

Python:

```
def runShellCommand(self, commandAsStr):
    logToReturn = ""
    try:
        popen = subprocess.Popen(commandAsStr, stdout=subprocess.PIPE,
stderr=subprocess.STDOUT, shell=True)
        lines_iterator = iter(popen.stdout.readline, b"")
        for line in lines_iterator:
            print line
            logToReturn = logToReturn + str(line)+"\n"
    except Exception,e:
        print e
    return logToReturn
```

13.2 Running Multiple Shell Commands as Array

At times you might need to run a sequence of shell commands, one that would set environment variables and one that would start a certain process. For that you will need to pass two or more commands in one shot in the form of an array. Example 13.2.1 demonstrates how to achieve this with features from Example 13.1.1.

EXAMPLE 13.2.1

Python:

```
def runShellCommandAsArray(commandAsArray):
    logToReturn = ""
    try:
        popen = subprocess.Popen(commandAsArray, stdout=subprocess.PIPE,
stderr=subprocess.STDOUT)
        lines_iterator = iter(popen.stdout.readline, b"")
        for line in lines_iterator:
            print line
            logToReturn = logToReturn + str(line)+"\\n"
    except Exception,e:
        print e
    return logToReturn
```


14. QUERYING DATABASES

14.1 SQLite3

To connect to and query a SQLite3 database, Python provides a module named exactly the same as the database itself: *sqlite3*. First, the *sqlite3* module needs to be imported into the file where the code base for it will reside. Then, the sequence of steps listed in Example 14.1.1 will connect to it, execute the select statement, and print the first ten selected records.

EXAMPLE 14.1.1

Python:

```
import sqlite3
```

```
class QuerySQLite3:
```

```
    def __init__(self):
        conn = sqlite3.connect('path/to/db_file.db')
        dbConnection = conn.cursor()
        executeQuery = dbConnection.execute('select * from TABLE_NAME limit 10')
        data = executeQuery.fetchall()
        print data
        dbConnection.close()
```

```
QuerySQLite3()
```

14.2 MySQL

Connection to MySQL is very similar to how it is done in the case of SQLite3 database. The module, *MySQLdb*, will need to be installed first using the command *pip install*

MySQLdb. After the module is installed, it will need to be imported into a code base and used just as in Example 14.2.1 that will connect to mySQL database, execute the select statement, and print the first ten selected records. Do not forget to provide host address, username, password, and the name of your database for the module to connect to your database.

EXAMPLE 14.2.1

Python:

```
import MySQLdb
```

```
class QueryMySQL:
```

```
    def __init__(self):
        conn = MySQLdb.connect("host", "username", "password", "database_name" )
        dbConnection = conn.cursor()
        executeQuery = dbConnection.execute('select * from TABLE_NAME limit 10')
        data = executeQuery.fetchall()
        print data
        dbConnection.close()
```

```
QueryMySQL()
```

14.3 Oracle

Working with Oracle database is by far the most complex process out of the three databases. Before Python can connect to an Oracle database, even if one is hosted remotely, you would need to set up Oracle SDK. Then, you would need to install and use Python's module *cx_Oracle* to connect and run commands against the Oracle database.

The *cx_Oracle* module depends on the version of the GLIBC installed on your Linux. The latest version of *cx_Oracle*, as of the time of this writing, required GLIBC 2.14. I brought this up because my distribution of Red Hat Linux ran on GLIBC 2.12 and there was no way

for me to update GLIBC to version 2.14 without negatively affecting other processes that ran on my Linux machine. For another solution, I decided to go with the older version of *cx_Oracle* utility. Below are the steps I performed to set the Oracle environment and install the required module in order to work with Oracle from Python.

Steps:

First, run the following command and see what version of GLIBC runs on your Linux distribution:

```
rpm -qa glibc
```

The response you will see should look something like this:

```
glibc-2.12-1.192.el6.x86_64
```

That tells you that you run on *machine x86_64* and GLIBC version 2.12, as in my case. Remember, the latest version at the time of this writing is *cx_Oracle* version 5.2.1, which requires GLIBC version 2.14 or higher.

Next, follow these steps to set up Oracle SDK and install *cx_Oracle*.

Step 1:

Download and install the following Oracle packages of Version 11.1.0.7.0:

```
http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html
```

1) oracle-instantclient11.1-basic-11.1.0.7.0-1.x86_64.rpm

The command to install is:

```
sudo rpm -ivh oracle-instantclient11.1-basic-11.1.0.7.0-1.x86_64.rpm
```

2) oracle-instantclient11.1-sqlplus-11.1.0.7.0-1.x86_64.rpm

The command to install is:

```
sudo rpm -ivh oracle-instantclient11.1-sqlplus-11.1.0.7.0-1.x86_64.rpm
```

3) oracle-instantclient11.1-devel-11.1.0.7.0-1.x86_64.rpm

The command to install is:

```
sudo rpm -ivh oracle-instantclient11.1-devel-11.1.0.7.0-1.x86_64.rpm
```

Step 2:

Run the following exports:

- 1) export LD_LIBRARY_PATH=/usr/lib/oracle/11.1/client64/lib/ >> ~/.bashrc
- 2) export ORACLE_HOME=/usr/lib/oracle/11.1/client64 >> ~/.bashrc
- 3) export PATH=\$ORACLE_HOME/bin:\$PATH >> ~/.bashrc

Step 3:

Download and install the following version of *cx_Oracle* module:

https://sourceforge.net/projects/cx-oracle/files/5.1.2/cx_Oracle-5.1.2-11g-py27-1.x86_64.rpm/download

The command to install is:

```
sudo rpm -ivh cx_Oracle-5.1.2-11g-py27-1.x86_64.rpm
```

After SDK is setup, all dependencies are installed, and paths are set in the *bash_profile*, you then need to install Python's module, *cx_Oracle*. As explained on its website (<http://cx-oracle.sourceforge.net>), the module can be installed simply using *pip install cx_Oracle*. However, from my experience, I could not make the latest version work and ended up installing an older version. Older versions of the *cx_Oracle* module can be found here <https://sourceforge.net/projects/cx-oracle/files/>.

When the *cx_Oracle* module is installed, please test the installation by importing it into Python environment first and make sure that import works without any errors.

Then try example 14.3.1 to run a select statement and print out the first ten records, as I did in SQLite3 and MySQL examples from the previous two section.

EXAMPLE 14.3.1

Python:

```
import cx_Oracle
```

```
class OracleDB():
```

```
    def getConnected(self):
```

```
        conn = cx_Oracle.connect('username/password@host:1251/database_name')
```

```
        dbConnection = conn.cursor()
```

```
        dbConnection.execute('select * from TABLE_NAME limit 10')
```

```
        data = dbConnection.fetchall()
```

```
        print data
```

```
        dbConnection.close()
```

```
        conn.close()
```


15. BUILDING STAND-ALONE APPLICATIONS WITH PYTHON

Believe it or not, you can also build a fully functional desktop-based application with great user-friendly interface. Just like Java, Python is also a multi-platform language, therefore the application needs to be developed only once and it can run on multiple platforms, such as Mac OS X, MS Windows, and Linux. You probably have read somewhere that Python is an interpreted scripting language. Not for me. For me Python is a fully featured, object-oriented programming language that gives you the ability to run interpreted scripts or programs, as well as the ability to compile programs into a fully featured, stand-alone, multi-platform applications that would run on all major desktop operating systems. As I am writing this book, Python hasn't yet reached the mobile platform world, but who knows, it might just be around the corner too. For now, I list below three resources that will help you build Linux-based, Mac OS-based, or Windows-based applications.

15.1 PyInstaller

According to developers of PyInstaller, it is a program that freezes (packages) Python programs into stand-alone executables, under Windows, Linux, Mac OS X, FreeBSD, Solaris, and AIX. Its main advantages over similar tools are that PyInstaller works with Python 2.7 and 3.3—3.5, it builds smaller executables thanks to transparent compression, it is fully multi-platform, and it uses the OS support to load the dynamic libraries, thus ensuring full compatibility. The main goal of PyInstaller is to be compatible with third-party packages out of the box. This means that, with PyInstaller, all the required tricks to make external packages work are already integrated within PyInstaller itself so that there is no user intervention required. You'll never be required to look for tricks in wikis and apply custom modification to your files or your setup scripts. As an example, libraries like PyQt, Django, or matplotlib are fully supported, without requiring you to handle plugins or external data files manually.

Please visit PyInstaller's website for more details and tutorials:

<http://www.pyinstaller.org/>

15.2 py2app

According to developers of py2app, it is a Python setuptools command, which will allow you to make stand-alone application bundles and plugins from Python scripts for Mac OS X operating system. The website of py2app offers a great tutorial that explains in the tiniest detail how to install the py2app and compile Python programs using it.

Please visit py2app's website for more details and tutorials:

<https://pythonhosted.org/py2app/tutorial.html>

15.3 py2exe

According to developers of py2exe, it is a Python Distutils extension, which converts Python scripts into executable Windows programs, able to run without requiring a Python installation. The website of py2exe also has a great tutorial that goes into great detail in explaining how to install py2exe and compile Python programs using it.

Please visit py2exe's website for more details and tutorials:

<http://www.py2exe.org/index.cgi/Tutorial>

16. BUILDING WEBSITES WITH PYTHON

Python also provides a wealth of platforms for you to develop fully featured, database-driven web applications. Below I list three of the most popular platforms that can be used to build websites with Python.

16.1 Django

Developers of the Django platform promote it as a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of web development, so you can focus on writing your web application without needing to reinvent the wheel. It's free and open source. Django is by far my favorite platform and what I use to develop online. There are extensive free tutorials all over YouTube and Django's own website.

Please visit Django's website for more details and tutorials:

<https://www.djangoproject.com>

16.2 Flask

Flask is also very popular in the world of web development with Python. However, Flask is designed for small-scale websites, as opposed to Django, which is designed for building larger and more complex websites.

Please visit Flask's website for more details and tutorials:

<http://flask.pocoo.org>

16.3 Pyramid

Pyramid is a very general open source Python web framework. As a framework, its primary job is to make it easier for a developer to create an arbitrary web application. The type of application being created isn't really important; it could be a spreadsheet, a corporate intranet, or a social networking platform. Pyramid is general enough that it can be used in a wide variety of circumstances.

Please visit Pyramid's website for more details and tutorials:

<http://www.pylonsproject.org/projects/pyramid/about>

Author's Blog

I hope you have found this book helpful in speeding up your transition from Java to Python. My blog is another resource as you convert to Python, since there is indeed a lot more to Python than is contained here. Python has grown into a very serious competitor for powerful languages such as Java, and it continues to evolve, offering more and more modules that open up doors to the development of more complex systems.

As I deepen my experience as a Python programmer, I take notes in the form of articles and post them to my blog. So on my blog you will find additional tricks that are possible with Python. My online blog is accessible free for all at <http://HowToProgramInPython.com>.

The Contact page on my blog is a communication tool you can use to reach me with your questions, suggestions, and corrections to this book. I hope that in reading this book, you see how much you already know about Python as a Java engineer, and how quickly you can master Python and become an even stronger software developer.