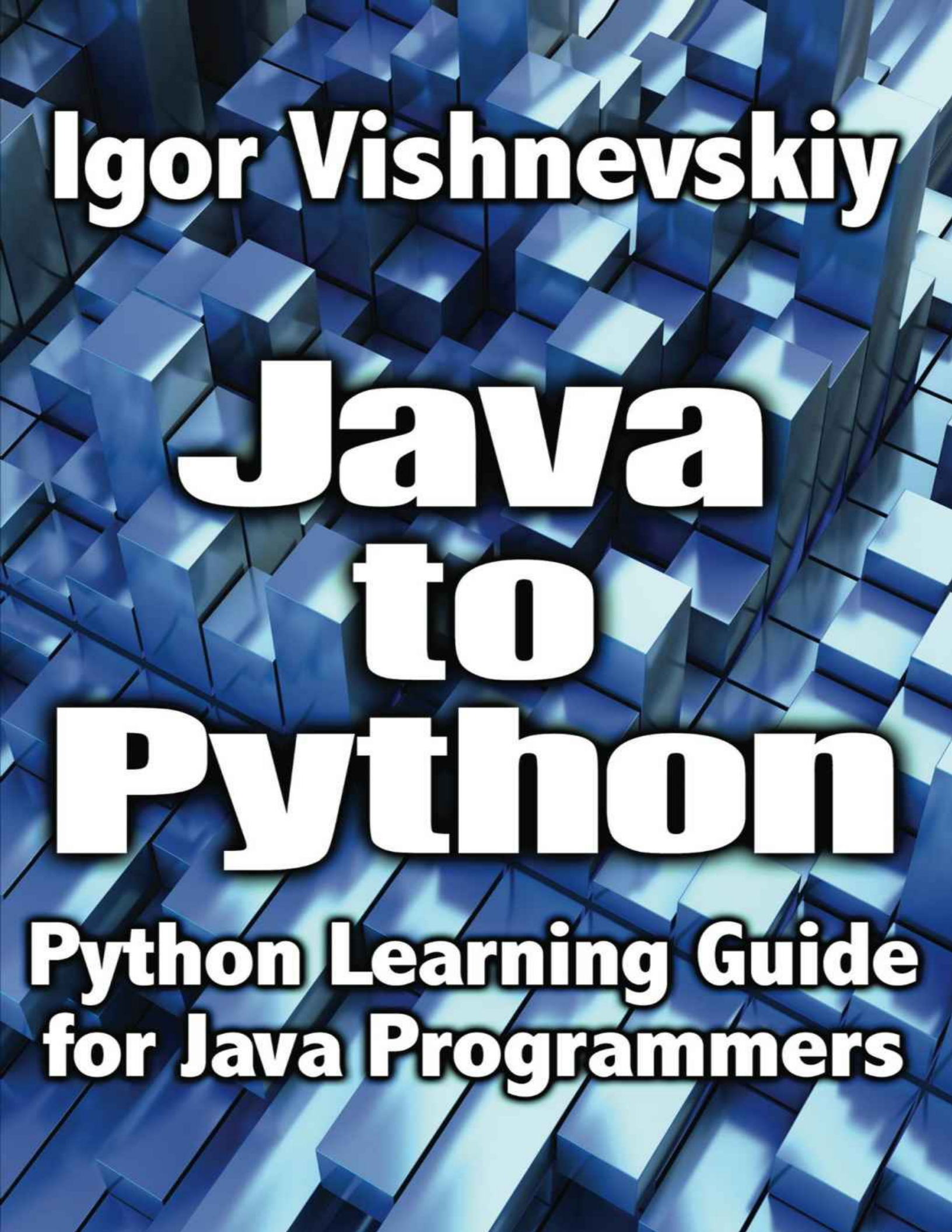
Igor Vishnevskiy
Java to Python
Python Learning Guide for Java Programmers

Java to Python by Igor Vishnevskiy

Table of Contents

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 similaries 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 exmples, 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 explicity. 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). Lets' 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 ASCI 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 ASCI format. If the program worked correctly, Figure 2.1 is what you can expect to see as an output in your Terminal window.

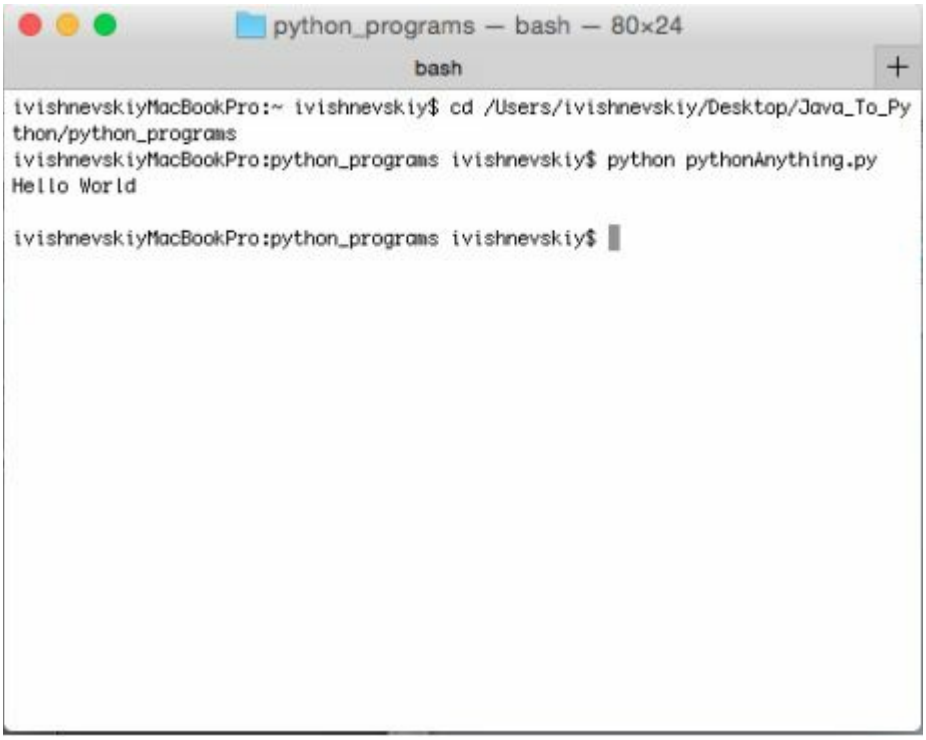

Figure 2.1

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 intVariable 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 floatVariable 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 intVariable 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 covert 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.