(Python) Chapter 1: Introduction to Programming in Python

1.1 Compiled vs. Interpreted Languages

Computers only understand 0s and 1s, their native machine language. All of the executable programs on your computer are a collection of these 0s and 1s that tell your computer exactly what to execute. However, humans do a rather poor job of communicating and 0s and 1s. If we had to always write our instructions to computers in this manner, things would go very, very slowly and we'd have quite a few unhappy computer programmers, to say the least. Luckily, there are two common solutions employed so that programmers don't have to write their instructions to a computer in 0s and 1s:

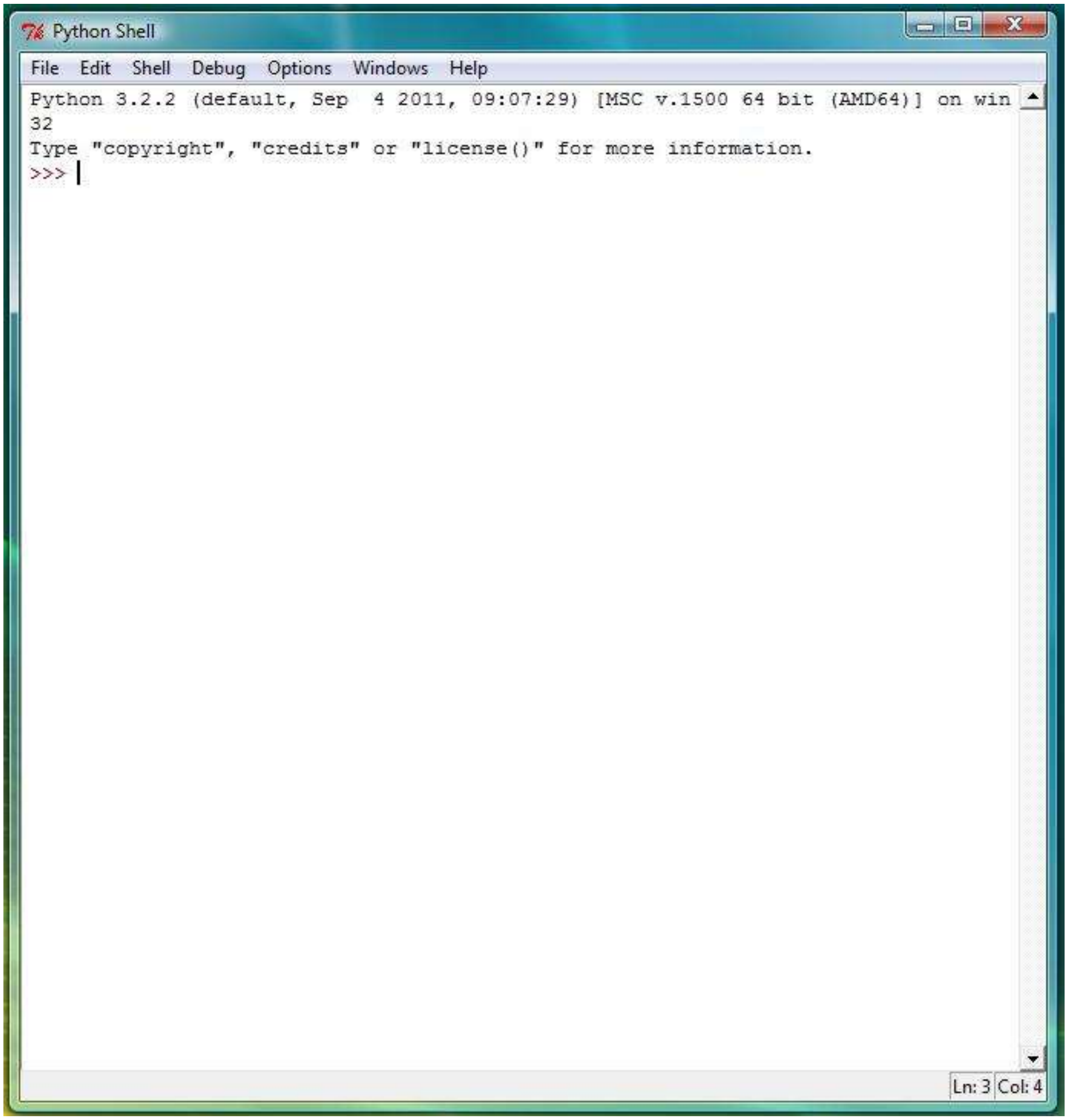
1) Compiled languages
2) Interpreted languages

In a compiled language, the programmer writes a program in a programming language that is human readable. Then, a program called a compiler translates this program into a set of 0s and 1s known as an executable file that the computer will understand. It's this executable file that the computer runs. If one wants to make changes to how their program runs, they must first make a change to their program, then recompile it (retranslate it) to create an updated executable file that the computer understands.

In an interpreted language, rather than doing all the translation at once, the compiler translates some of the code written (maybe a line or two) in a human readable language to an intermediate form, and then this form gets "interpreted" to 0s and 1s that the machine understands and immediately executes. Thus, the translation and execution are going on simultaneously.

Python is an interpreted programming language. One standard environment in which students often write python programs is IDLE (Integrated Distributed Learning Environment). This environment offers students two separate ways to write and run python programs. Since the language is interpreted, there exists an option for students to write a single line of python code and immediately see the results. Alternatively, students can open a separate window, put all of their commands in this window first, and then interpret their program. The first method lets students see, in real time, the results of their statements. The second follows the more traditional model of composing an entire program first before compiling and seeing the results. For learning purposes, the first technique is very useful. Ultimately however, students must develop their programs utilizing the second method.

When you open IDLE (Version 3.2.2) for the first time, you're presented with the following window:



```
Python 3.2.2 (default, Sep  4 2011, 09:07:29) [MSC v.1500 64 bit (AMD64)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> 
```

<u>1.2 Output - print statement</u>

print statement - basic form

The prompt (>>>) awaits the user to enter a line of python to be interpreted. The most simple line of code that shows a result is a print statement. Try the following example:

```
>>> print("Hello World!")
```

When you hit enter, you get the following output from the IDLE editor:

```
Hello World!
```

Now, consider typing in the following:

```
>>> print(Hello World)
```

Unfortunately, IDLE responds with the following error message:

```
SyntaxError: invalid syntax
```

Programming languages have very strict syntax rules. Whereas in English, if a grammar rule is improperly used, most people still understand the gist of the message, in a programming language, even if the most tiny rule is broken, the interpreter can NOT compensate by fixing the error. Rather, the interpreter gives an error message alerting the programmer about the error. In this case the message itself isn't terribly useful, since it's not specific at all. In other cases, the error messages are more specific. In this case it's clear that the only difference between the statement that worked and the one that didn't is that the latter is missing a pair of double quotes. This is the syntax error committed above.

Now, we can formally present the proper syntax of the print statement in python:

```
print(<string expression>)
```

First, we use the keyword "print," followed by a pair of enclosing parentheses (). Inside those parentheses we must provide a valid string expression.

The first type of string expression we'll learn is a string literal. In common English, the word literal means, "in accordance with, involving, or being the primary or strict meaning of the word or words; not figurative or metaphorical." In programming, literal simply means "constant." A literal expression is one that can not change value. In python, and in many other programming languages, string literals are designated by matching double quotes. Everything inside of the double quotes is treated as a string, or sequence of characters, exactly as they've been typed, with a few exceptions.

Thus, the meaning of

```
print("Hello World!")
```

in python is to simply print out what appears inside the double quotes exactly.

Before moving on, try printing out several messages of your own composition.

print statement - escape sequences

After experimenting with the print statement, you might find some limitations. For example, try printing a message that will be printed on multiple lines using a single print statement such as the following:

```
Python
is
fun!
```

One idea might be to physically hit the enter key after typing in "Python" in the middle of the print statement. Unfortunately, doing this yields the error:

```
SyntaxError: EOL while scanning string literal
```

EOL stands for "end of line." The meaning of the error is that the interpreter was waiting to read in the end of the string literal, denoted by the second double quote, before the end of the line, since all python statements must fit on a single line. When the interpreter encountered the end of the line, which meant the end of the statement as well, it realized that the string literal had not been finished.

In order to "fix" this issue, we need some way to denote to the interpreter that we wish to advance to the next line, without having to literally type in the enter key. python, as many other languages do, provides escape sequences to deal with issues like this one. An escape sequence is a code for a character not to be taken literally. For example, the escape sequence for the new line character is \n. When these two characters are encountered in sequence in a string literal, the interpreter knows not to print out a backslash followed by an n. Rather, it knows that these two characters put together are the code for a new line character. Thus, to print out

```
Python
is
fun!
```

to the screen with a single print, we can do the following:

```
print("Python\nis\nfun!")
```

Here is a list of commonly used escape sequences:

Character	Escape Sequence
tab	\t
double quote	\"
single quote	\'
backslash	\\

The rest can be found in python's online documentation.

Thus, one way to print out the following

```
Sam says, "Goodbye!"
```

is as follows:

```
print("Sam says, \"Goodbye!\"")
```

Second Way to Denote a String Literal in Python

One way in which python differs from other langauges is that it provides two ways to specify string literals. In particular, instead of using double quotes to begin and end a string literal, one can use single quotes as well. Either is fine. Thus, the message above can be printed out more easily as follows:

```
print('Sam says, "Goodbye!"')
```

From the beginning of the statement, the python interpreter knows that the programmer is using single quotes to denote the start and end of the string literal, and can therefore treat the double quote it encounters as a double quote, instead of the end of the string.

Automatic newlines between prints

Normally when we run IDLE, we are forced to see the results of a single line of code immediately. Most real computer programs however involve planning a sequence of instructions in advance, and then seeing the results of all of those instructions running, without having to type in each new instruction, one at a time, while the program is running.

This will be useful for us so we can see the effect of running two consecutive print statements in a row.

In order to do this, when you are in IDLE's main window, simply click on the "File" menu and select the first choice, "New Window." After this selection, a new empty window will pop up. From here type the following into the window:

```
print("Hello ")
print("World!")
```

Then, go to the "File" menu in the new window and click on the choice, "Save As." Click to the directory to which you want to save this file and give a name in the box labeled "File Name." Something like hello.py will suffice. Make sure to add the .py ending even though the file type is already showing below. This will ensure that the IDLE editor highlighting will appear. Once you've saved the file, you are ready to run/interpret it. Go to the "Run" menu and select, "Run Module." Once you do this, you'll see the following output:

```
Hello
World!
```

What has happened is that by default, python inserts a newline character between each print statement. While this is desireable often times, there will be cases where the programmer does NOT want to automatically advance to the next line. To turn off this automatic feature, add the following to the print statement:

```
print("Hello ", end = "")
print("World!")
```

When we add a comma after the string literal, we are telling the print statement that we have more information for it. In particular, we are specifying that instead of ending our print with the default newline character, we'd like to end it with nothing. Note that we can put any string inside of the double quotes after the equal sign and whatever we specify will be printed at the end of that particular print statement. In this case, we have not made the same specification for the second print, so that the newline character is printed after the exclamation point.

While there are some other nuances to basic printing, this is good enough for a start. Other rules for printing will be introduced as needed.

*String operators (+, *)*

Python also offers two operators for strings: string concatenation (+), and repeated string concatenation(*). The concatenation of two strings is simply the result of placing one string next to another. For example, the concatenation of the strings "apple " and "pie" is "apple pie". The repeated concatenation of the same string is simply repeating the same string a certain number of times. For example, in python, multiplying "ahh" by 4 yields "ahhahhahhahh".

Note that these operators also work for numbers and are defined differently for numbers. In a programming language, whenever the same item has two different definitions, the term given to that practice is "overloading." Thus, in python (and in some other programming languages), the + sign is overloaded to have two separate meanings. (This is common in English. For example, the verb "to sign" can either mean to write one's signature, or to communicate an idea in sign language.) The computer determines which of the two meanings to use by looking at the two items being "added." If both items are strings, python does string concatenation. If both are numbers, python adds. If one item is a string and the other is a number, python gives an error.
Alternatively, for repeated string concatenation, exactly one of the two items being multiplied must be a string and the other must be a non-negative integer. If both items are numbers, regular multiplication occurs and if both items are strings, an error occurs. The following examples clarify these rules:

```
print("Happy "+"Birthday!")
print(3 + 4)
print("3 + 4")
print("3"+"4")
print(3*4)
print(3*"4")
print("3"*4)
print("I will not talk in class.\n"*3)
```

If we save this segment in a .py file and run it, the output is as follows:

```
Happy Birthday!
7
3 + 4
34
12
444
3333
I will not talk in class.
I will not talk in class.
```

```
I will not talk in class.
```

The following statements each cause an error:

```
print(3+"4")
print("3"+4)
print("me"*"you")
```

The errors are as follows:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
TypeError: Can't convert 'int' object to str implicitly
TypeError: can't multiply sequence by non-int of type 'str'
```

In each case, the interpreter points out that a type error has occured. It was expecting a number in the first statement, a string in the second statement and a number in the third statement for the second item.