(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:
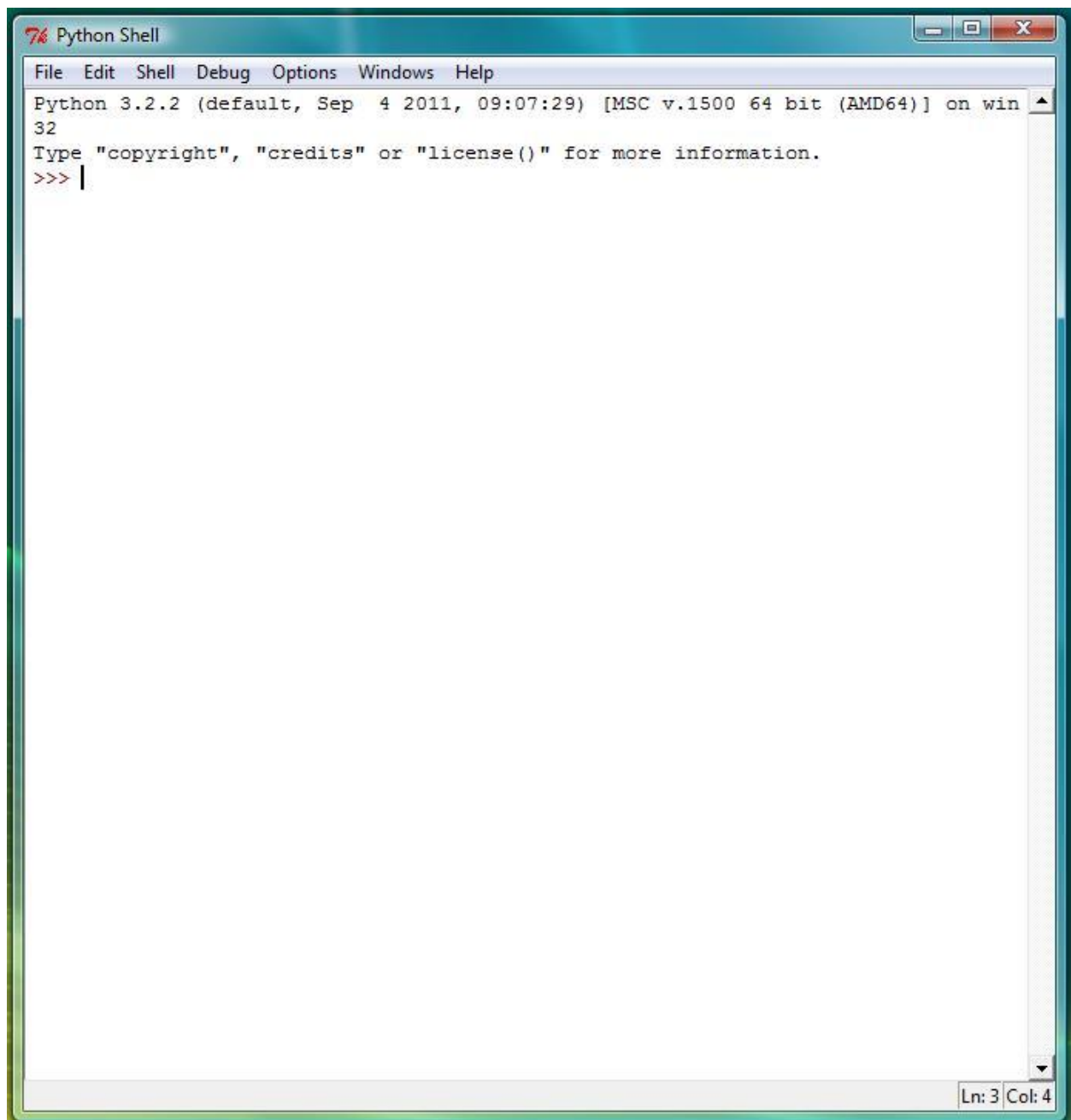
1) Compiled languages
2) Interpreted languages

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:

```
76 Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
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.
>>> |
                                                                    Ln: 3 Col: 4
```

1.2 Output - print statement

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.

1.3 Arithmetic Expressions - A First Look

*Standard operators (+, -, *, /)*

One of the major operations all computer programs have built in are arithmetic computations. These are used as parts of full statements, but it's important to understand the rules of arithmetic expressions in general, so that we can determine how the python interpreter calculates each expression. We can easily see the value of any arithmetic expression by typing it into the interpreter:

```
>>> 3+4
7
>>> 17-6
11
>>> 2 + 3*4
14
>>> (2 + 3)*4
20
>>> 3 + 11/4
5.75
```

Note that we'd never use any of these expressions as a whole line in a python program. The examples above are simply for learning purposes. We'll soon learn how to incorporate arithmetic expressions in python programs.

The four operators listed, addition (+), subtraction (-) and multiplication (*), work exactly as they were taught in grade school. As the examples above illustrate, multiplication and division have higher precedence than addition or subtraction and parentheses can be used to dictate which order to do operations.

## 1.4 Variables in Python

Idea of a Variable

Part of the reason computer programs are powerful is that they can make calculations with different numbers, using the same set of instructions. The way in which this is done is through the use of variables. Rather than calculating 5*5, if we could calculate side*side, for any value of side, then we have the ability to calculate the area of any square instead of the area of a square of side 5.

Python makes variables very easy to use. Any time you want to use a variable, you can put the name of the variable in your code. The only caveat is that when you first create a variable, it does not have a well-defined, so you can't use that variable in a context where it needs a value.

The most simple way in which a variable can be introduced is through an assignment statement as follows:

```
>>> side = 5
```

The name of the variable created is side, and the statement above sets side to the value 5. A picture of what memory looks like at this point in time is as follows:

side [5]
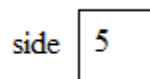
If we follow this statement with

```
>>> area = side*side
```

Then our picture in memory is as follows:

side [5]   area [25]

Let's analyze what's happening here. Any statement with a variable on the left of a single equal sign and an expression on the right of that equal side is called an assignment statement. The goal of an assignment statement is to assign a variable to a value. It works in the following two step process:

1) Figure out the current value of the expression on the right, using the current values of the variables.

2) Change the value of the variable on the left to equal this value.

Thus, in the statement above, at the time it was executed, side was equal to 5. Thus, side*side evaluated to 25. Then, the box for area was replaced with the value 25.

Printing out the value of a variable

Of course, when we run these two statements in IDLE, we don't SEE any evidence that the variables are these two values. In order to do this, we need to learn how to print out the value of a variable in python. The most simple way to do so is as follows:

```
>>> print(area)
25
>>> print(side)
5
```

Notice that when we execute these prints, we DON'T include double quotes. Had we done the following:

```
>>> print("area")
area
>>> print("side")
side
```

the words in question would have printed instead of the values of those corresponding variables. What we see here is the previous rules we learned, that anything in between double quotes gets printed as is, except for escape sequences, does not change. Rather, a new construct (one without double quotes) must be used to print out the value of a variable.

Another natural question that arises is,"What if we want to print out both the value of a variable and some text in the same print?" Python allows us to do this by separating each item we would like to print with commas, as is shown below:

```
>>> print("The area of a square with side",side,"is",area)
The area of a square with side 5 is 25
```

If you carefully examine the text above, you'll see that between each item specified in the print (there are 4 items), python naturally inserted a space in the output, even though we didn't explicitly place one. This is python's default setting and in many cases is quite desirable. What if we wanted to place a period right after the 25 in the statement above? If we put a comma after area and place the string ".", we would find that a space would be inserted in between the 5 and the period:

```
>>> print("The area of a square with side",side,"is",area,".")
The area of a square with side 5 is 25 .
```

To avoid this, we can specify the separator between print items as follows:

```
>>> print("The area of a square with side ",side," is ",area,".",sep="")
The area of a square with side 5 is 25.
```

Notice that when we change the separator from a single space to no space, we then have to manually add spaces after the word "side" and before and after the word "is."

Not that the use of a different separator is typical, but any specifier can be used as the following example illustrates[1]:

```
>>> print(side,side,area,sep=" }:-) ")
5 }:-) 5 }:-) 25
```

Increment statement

Consider following the original two statements in the previous section with the somewhat confusing statement:

```
>>> side = side + 1
```

In mathematics, it's impossible for a variable to equal itself plus one. In programming however, this statement isn't a paradox. By following the rules, we see that at the current time of evaluation, side is equal to 5. It follows that the right hand side of the assignment statement is equal to 5 + 1, or 6. The following step is to *change* the variable on the left (which happens to be side) to this value, 6. The corresponding picture is as follows:

To prove that this is in fact what has happened, execute the following statement:

```
>>> print("area =",area,"side =",side)
area = 25 side = 6
```

---

[1] Wikipedia claims that this emoticon means "devilish." I promise that I, in no way, promote "devilish" behavior however!

One key observation to make here is that area is STILL 25. It has not magically changed to 36 (the new value of side*side) after side was changed. Python only executes the commands it is given. Thus, if we wanted to reevaluate the area of a square with side 6, we'd have to recalculate the area as well.

After we change side, if we run the following lines of code again:

```
>>> area = side*side
>>> print(area)
36
```

We see that NOW, area has changed to 36, because we have specifically reevaluated side*side and stored this new value back into the variable area.

Rules for Naming a Variable

Clearly, a variable can't be named anything. Instead, python has rules for which names are valid names for variables and which aren't. In particular, the only characters allowed ina variable name are letters, digits, and the underscore('_') character. Furthermore, variables names can't start with a digit. (This rule is so they aren't confused with numbers.)

In general, while it's not required, it's considered good programming style to give variables names that are connected to the function the variable is serving. In the previous example, both side and area represent the type of information stored in those respective variables. If the variables were named a and b, respectively, someone else reading the code would have much more difficulty figuring out what the code was doing. Many beginning programmers, due to laziness or other factors, get into the habit of creating short variable names not connected to the function of the variable. For small programs these programmers don't run into much difficulty, but in larger programs, it's very difficult to track down mistakes if the function of a variable isn't immediately apparent.

Two Program Examples

Using the set of statements above, we can create a stand alone program by typing the following in a separate window and saving it as a python program:

```
# Arup Guha
# 6/1/2012
# Python Program to calculate the area of a square.

side = 5
area = side*side
print("The area of a square with side",side,"is",area)
```

Executing this program leads to the following output:

```
The area of a square with side 5 is 25
```

Comments

Large pieces of code are difficult for others to read. To aid others, programmers typically put some comments in their code. A comment is piece of a program that is ignored by the interpreter, but can been seen by anyone reading the code. It gives the reader some basic information. A header comment is included at the top of each different program. It identifies the author(s) of the file, the date the file was created/edited as well as the program's purpose. To denote a comment in python, just use the pound symbol (#). All text following the pound symbol on a line is treated as a comment by the interpreter. Although it's not shown above, a comment can start in the middle of a line:

```
area = side*side # Setting the area.
```

However, it's customary just to comment on a line by itself in most instances as follows:

```
# Setting the area.
area = side*side
```

Now let's look at a second program that calculates the total cost of an item with tax:

```
# Arup Guha
# 6/1/2012
# Python Program to cost of an item with tax.

item_price = 10.99
tax_rate = 6.5
total_price = item_price*(1+tax_rate/100)
print("Your total cost is $",total_price,".",sep="")
```

The output of running this statement is as follows:

```
Your total cost is $11.70435.
```

For now, let's not worry about outputting our result to two decimal places. We'll get to that later.

1.5 Arithmetic Expressions in Python

In the two examples in the previous section, we used arithmetic expressions on the right-hand side of the assignment statement (equal sign). Python has its set of rules about how these expressions are to be evaluated, so that there is no ambiguity. So, far, we've simply illustrated that multiplication and division have higher precedence than addition and subtraction, as is frequently taught in grade school mathematics. Furthermore, parentheses have the highest precedence and can be used to "force" the order in which operations are evaluated as is illustrated in the this line of code that was previously shown:

```
total_price = item_price*(1+tax_rate/100)
```

In this expression, we first evaluate the contents of the parentheses before multiplying. In evaluating the contents of the parentheses, we first perform the division, since that has higher precedence than addition. Thus, for example, if tax_rate is 7, then we first take 7/100 to get .07 and then add that to 1 to get 1.07. Then, the current value of item_price is multiplied by 1.07, which is then assigned to total_price.

Python also provides three more operators for us to use:

1. **, for exponentiation.
2. //, for integer division
3. %, for modulus

The following sections describe exactly how each of these operators work, and the order of operations for each of them.

Exponentiation (**)

Many students first learn to associate the caret symbol (^) with exponentiation because that key is typically linked to exponentiation on most calculators that students use in grade school. However, in most programming languages, the caret symbol is either not defined or means something very different than exponentiation.

Some programming languages don't define exponentiation through an operator at all, but python does. The operator is defined to work for real numbers. Here are several examples of its use:

```
>>> 2 ** 3
8
>>> 3 ** 5
```

```
243
>>> 4 ** 10
1048576
>>> 25 ** .5
5.0
>>> 2 ** -3
0.125
>>> 9999 ** 0
1
>>> -5 ** -3
-0.008
>>> -5 ** 3
-125
>>> 1.6743 ** 2.3233
3.311554089370817
```

If both operands to an exponentiation operation are integers and the answer is an integer, then it will be expressed as one. If both operands are integers but the answer isn't, then the answer will be expressed as a real number with decimals. As the examples above illustrate, if an exponent b is negative, then a^b is defined as $1/a^{-b}$. (Technically, this rule is true for all exponents, not just negative ones.)

Integer division (//)

Python provides a second division operator, //, which performs integer division. In particular, the answer of an integer division operation is always an integer. In particular, a//b is defined as the largest number of whole times b divides into a. Here are a few examples of evaluating expressions with integer division:

```
>>> 25//4
6
>>> 13//6
2
>>> 100//4
25
>>> 99//100
0
>>> -17//18
-1
>>> -1//10000000
-1
```

```
>>> -99999999//99999999
-1
>>> -25//4
-7
>>> -13//6
-3
>>> -12//6
-2
>>> -11//6
-2
>>> 0//5
0
```

It's important to note that python deals with this operation differently than many other programming languages and different than most people's intuitive notion of integer division. In particular, when most people see -13//6, they tend to think that this is pretty close to -2, so that the answer should be -2. But, if we look at the technical definition of integer division in python, we see that -2 is greater than -13/6, which is roughly -2.166667, and the largest integer less than or equal to this value is really -3.

Incidentally, the definition of integer division in Python does not require the two numbers that are being divided to be integers. Thus, integer division operations are permissible on numbers that aren't integers. Consider the following examples:

```
>>> 6.4 // 3.1
2.0
>>> 3.4 // 3.5
0.0
>>> 9.9 // .03
330.0
>>> -45.3 // -11.2
4.0
>>> 45.3 // -11.2
-5.0
```

This feature of Python is used relatively rarely, so no further details will be given at this point.

Modulus Operator (%)

For those who have never programmed, it's likely that the modulus operator, denoted by the percent sign(%), is not familiar. In mathematics, modulus is typically defined for integers only. However in python, the modulus operator is defined for both integers and real numbers. If both operands are integers, then the answer will be an integer, otherwise, it will be a real number.

Intuitively, the modulus operator calculates the remainder in a division, while integer division calculates the quotient. Another way of thinking about modulus is that is simply calculates the leftover when dividing two numbers. This intuitive definition is correct when dealing with positive numbers in Python. Negative numbers are another story, however.

The formal definition of a % b is as follows:

a % b evaluates to a - (a // b)*b.

a // b represents the whole number of times b divides into a. Thus, we are looking for the total number of times b goes into a, and subtracting out of a, that many multiples of b, leaving the "leftover."

Here are some conventional examples of mod, using only non-negative numbers:

```
>>> 17 % 3
2
>>> 37 % 4
1
>>> 17 % 9
8
>>> 0 % 17
0
>>> 98 % 7
0
>>> 199 % 200
199
```

We see in these examples, if the first value is ever less than the second value, then the answer of the operation is simply the first value, since the second value divides into it 0 times. In the rest of the examples, we see that if we simply subtract out the appropriate number of multiples of the second number, we arrive at the answer.

With negative integers however, one must simply plug into the formal defintion instead of attempting to use intuition. Consider the following examples:

```
>>> 25 % -6
-5
>>> -25 % -6
-1
>>> -25 % 6
5
>>> -48 % 8
0
>>> -48 % -6
0
>>> -47 % 6
1
>>> -47 % 8
1
>>> -47 % -8
-7
```

The key issue that explains the first two results is that there is a different answer for the two integer divisions 25//-6 and -25//-6. The first evaluates to -5 while the second evaluates to 4. Thus, for the first, we calculate -6 * -5 = 30, and need to subtract 5 to obtain 25. For the second, we calculate -6 * 4 = -24, and need to subtract 1 to obtain -25.

See if you can properly apply the definition given to explain each of the other answers shown above.

In Python, the modulus operator is also defined for real numbers, using the same definition as the one shown. Here are a few examples of its use:

```
>>> 6.4 % 3.1
0.20000000000000018
>>> 3.4 % 3.5
3.4
>>> 9.9 % .03
7.216449660063518e-16
>>> -45.3 % -11.2
-0.5
>>> 45.3 % -11.2
-10.7
```

If we take a look at the first and third examples, we see the answer that python gives is slightly different than we expect. We expect the first to be 0.2 exactly and the third to be 0. Unfortunately, many real numbers are not stored perfectly in the computer. (This is true in all computer languages.) Thus, in calculations involving real numbers, occasionally there are some slight round-off errors. In this case, the digits 1 and 8 way at the right end of the number represent a round-off error. In the third case, the last part of it, e-16, simply means to multiply the previously shown number by 10^{-16}, so the entire part printed represents the round-off error, which is still tiny, since it's less than 10^{-15}. As long as we don't need extreme precision in our calculations, we can live with the slight errors produced by real number calculations on a typical computer. The more complicated calculations are, the greater the possible error could cascade to be. But, for our purposes, we'll simply assume that our real number answers are "close enough" for our purposes. (There are a variety of advanced techniques that help programmers contend with round off error of real number calculations.)

1.6 Reading User Input in Python

input statement

Python makes reading input from the user very easy. In particular, Python makes sure that there is always a prompt (a print) for the user to enter some information. Consider the following example entered

```
>>> name = input("What is your name?\n")
What is your name?
Simone
>>> print("Please to meet you ", name, ".", sep="")
Please to meet you Simone.
```

In this way, instead of the print always printing out the same name, it will print out whatever name the user entered. The key is that the input statement read in whatever the user entered, and then the assignment statement, with the equal sign, assigned this value to the variable name. Then, we were free to use name as we pleased, knowing that it stored the value the user entered.

Our previous programs that calculated the price of an item with tax and the area of a square, were limited because they always calculated the same price and area. Our program would be much more powerful if we allowed the user to enter the appropriate values so that our program could calculate the information THEY are interested in (as opposed to the same value every time.)

Before we look at the necessary edits to make these programs take in user input, one other note must be made about the input function. It always returns whatever it reads in from the user as a string. This means that if the user enters a number, such as 79, the input statement returns it as "79" instead. (This is a string that literally is the character '7' followed by the character '9', instead of the number 79.) Thus, we need some method to convert the string "79" to the number 79. The way this is done is through a function that changes its input to a different type. To turn a string into an integer, use the int function:

```
>>> age = int(input("How old are you, "+name+"?\n"))

How old are you, Simone?
22
>>> print("Your name is ",name,". You are ",age," years old.", sep="")
Your name is Simone. You are 22 years old.
```

In this example, we had to use the int function (on the first line) to convert the string the input function returned into an integer. Then, this was assigned to the variable age. Thus, age stores an integer and not a string.

In prompting Simone, you'll notice that plus signs, denoting string concatenation were used instead of commas, which we first used when we learned the print statement. The reason for this is that while the print statement takes in multiple items separated by commas (these are called parameters), the input statement only takes in a single string. Thus, we were forced to create a single string by using string concatenation. Since the variable name is a string, we were able to concatenate it with the rest of the message. Here is the error message we would have gotten had we attempted to pass the input function separate items separated by commas:

```
>>> age = int(input("How old are you, ",name,"?\n"))
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    age = int(input("How old are you, ",name,"?\n"))
TypeError: input expected at most 1 arguments, got 3
```

The last line of IDLE's output tells us what occurred. The input function expects 1 argument, or piece of information, but we gave it three pieces of information, since the commas are what separate pieces of information (arguments) given to a function.

In the ensuing print, we can give the print function multiple pieces of information separated by commas, and we see that age has indeed been stored as 22.

1.7 Examples of Programs Using the input() statement

Making More General Programs

In all of our examples, we've only been able to write programs that made very specific calculations. For example, we only found the area of one specific square, or the price of one specific item with tax. This isn't terribly helpful because not all of the items we buy will be the same price.

It would be nice if the same program can answer any price question or any square area question. This is where user input helps us. Instead of being forced to set a variable to one specific value, we can simply ask the user to enter a value, and then set a variable to that value, allowing the user to dictate the specific calculation that occurs. Here is the program that calculates the area of a square edited to allow for user input:

```
# Arup Guha
# 6/22/2012
# Python Program to calculate the area of a square - using user input.

side = int(input("Please enter the side of your square.\n"))
area = side*side
print("The area of a square with side",side,"is",area)
```

In converting our item price program to incorporate user input, we find that the type of information we are reading in is different than an integer. It's a real number, which is called a "float" in python. In order to convert a string to a float, the float function must be used. Thus, our edited program is as follows:

```
# Arup Guha
# 6/22/2012
# Python Program to cost of an item with tax.
# Edited to take in user input. Uses Orange County's sales tax rate.

tax_rate = 6.5

item_price = float(input("Please enter the price of your item.\n"))
total_price = item_price*(1+tax_rate/100)
print("Your total cost is $",total_price,".",sep="")
```

Temperature Conversion Example

While Americans use the Fahrenheit scale to measure temperature, many others use the Celsius scale. In this program, we'll ask the user to enter a temperature in Celsius and our program will convert that temperature to Fahrenheit. The formula for conversion is as follows:

$$F = 1.8C + 32$$

Here is the program:

```
# Arup Guha
# 6/22/2012
# Program to convert Celsius to Fahrenheit

temp_cel = float(input("Please enter the temperature in Celsius.\n"))
temp_fahr = 1.8*temp_cel + 32;
print(temp_cel,"degrees Celsius =",temp_fahr,"degrees Fahrenheit.")
```

Here is a sample of the program running:

```
>>>
Please enter the temperature in Celsius.
37
37.0 degrees Celsius = 98.60000000000001 degrees Fahrenheit.
>>>
```

Fuel Efficiency Example

Consider the following problem:

You are taking a road trip. When you fill up your gas tank (you know how many gallons your tank is), you notice the reading on your odometer. Later in the drive, you see exactly how much gas is left and the reading on the odometer. Given all of this information, we want to calculate how many more miles we can drive before having to stop for gas again. In real life, we would want to include a margin for error and stop several miles short of when our fuel would run out. But for purposes of simplicity, in this program, we'll simply calculate when we expect to run out of fuel, assuming that we drive with a constant fuel efficiency.

This problem is slightly more involved than the previous ones. Rather than immediately typing into the IDLE window, we need to sit back and think about the problem, sketching out what variables we want to use and how we will solve the problem.

After reading the problem statement, we see that we must read in the following variables from the user:

1) Initial Odometer Reading (in miles)
2) Gas Tank Size (in gallons)
3) Odometer Reading at Intermediate Point (in miles)
4) How Much Gas is Left at Intermediate Point (in gallons)

The difference in variables 3 and 1 represents the distance traveled while the difference of variables 2 and 4 represents the amount of gas used in the interim described. The division of the former by the latter will yield our fuel efficiency in miles/gallon. Since we know how many gallons of gas are left, we can multiply this by our fuel efficiency to see how much longer we can drive.

Let's take a look at the program:

```
# Arup Guha
# 6/22/2012
# Calculates the number of miles before having to refuel.

# Get all the user input.
start_odometer = int(input("What is the initial odometer reading?\n"))
gas_tank = float(input("How many gallons of gas does your tank hold\n"))
mid_odometer = int(input("What was your second odometer reading?\n"))
gas_left = float(input("How many gallons were left then?\n"))

# Calculate miles driven and gas used.
miles_driven = mid_odometer - start_odometer
gas_used = gas_tank - gas_left

# Calculate fuel efficiency and distance left to travel.
mpg = miles_driven/gas_used
distance_left = gas_left*mpg

print("You can go",distance_left,"miles before needing to refuel.")
```

Koolaid Example

You are running a Koolaid stand during the summer and want to be able to calculate how many glasses of Koolaid you need to sell to reach a profit goal for the day. There is a fixed daily rent for the stand itself and the other items you use. In addition, you have to buy materials (Koolaid mix, sugar). You know the cost of your materials per glass of Koolaid. Finally, you have a desired profit goal. The goal of this program is to calculate the minimum number of glasses of Koolaid that need to be sold to reach your profit goal. (Note: This answer has to be an integer.)

Once again, let's plan a bit before writing our program. First, let's idenfity the pieces of information we must get from the user:

1) Daily Rent (in dollars)
2) Cost of Material per glass (in cents)
3) Price you Charge Customers per glass (in cents)
4) Profit Goal (in dollars)

First, we can calculate the profit per glass of Koolaid by subtracting variable 2 from variable 3. We can then convert the profit goal PLUS the daily rent into cents, since this is our true revenue goal. (Due to the types of information involved, this will be a bit more difficult than multiplying by 100.) Finally, we can divide the two using integer division. Note: We find that this division will sometimes give us the incorrect answer. Consider the case where we need to earn 300 cents and we make a profit of 40 cents per glass. The integer division gives us $300//40 = 7$, but if we were to only sell 7 glasses, we'd only make up 280 cents, instead of 300. In particular, we find that this division is always one glass too low, except for with the two values divide evenly, such as 280 and 40. Thus, we want every profit value from 281 through 320 divided by 40 to round up to 8, while 280 divided by 40 stays at 7. We can simply do this by adding an offset of 39 to the division. (Note: 39 represents one less than the profit per cup of Koolaid.)

Here's the whole program:

```python
# Arup Guha
# 6/22/2012
# Koolaid Example

# Get the user input.
rent = int(input("How many dollars is the rent for your stand?\n"))
cost = int(input("How many cents do the materials cost, per glass?\n"))
price = int(input("How many cents do you charge per glass?\n"))
goal = int(input("What is your profit goal, in dollars?\n"))

# Calculate the profit per cup, in cents.
profit_per_cup = price - cost

# The total revenue we need to hit.
target = 100*(goal + rent)

# Calculate the final result, taking care of the off by one case.
num_cups = (target + profit_per_cup - 1) // profit_per_cup

# Print the final result.
print("You must sell at least",num_cups,"glasses to reach your goal.")
```

This example shows that care must be taken in correctly making calculations and that tools such as integer division and some creativity can be helpful in solving problems. (Note: An if statement can be used to make this problem easier. This will be covered in Chapter 2.)

1.8 Math Class

Several values and functions that are commonly associated with mathematics are useful in writing computer programs. In python, these are included in the math library. It's common in python and nearly all other programming languages to have many extra libraries that contain functions to help the programmer. In python, in order to use a library, an import statement must be made at the beginning of the python file. To import the math library, we simply add the line:

```python
import math
```

at the beginning of our python file.

Let's look at some of the functions and constants available to us from the math library in python:

Note: A complete list is located at http://docs.python.org/library/math.html.

math.pi - an approximation of the ratio of the circumference to the diameter of a circle.
math.e - an approximation of the base of the natural logarithm.

math.ceil(x) - Returns the smallest integer greater than or equal to it, as a float.
math.fabs(x) - Returns the absolute value of x.
math.factorial(x) - Returns x factorial, which is 1 * 2 * 3 * ... *x.
math.floor(x) - Returns the greatest integer less than or equal to x as a float.

math.exp(x) - Returns e^x.
math.log(x) - Returns ln x.
math.log(x, base) - Returns $\log_{base}x$.
math.sqrt(x) - Returns the square root of x.

math.cos(x) - Returns the cosine of x radians.
math.sin(x) - Returns the sine of x radians.
math.tan(x) - Returns the tangent of x radians.

As is indicated in this list above, in order to call these functions, we must type "math." before the name of each function. Namely, we must specify from which library the function is from, so that it doesn't get confused with other functions with the same name.

Let's look at a couple examples of python programs that make calls to the math library.

Circle Area and Circumference Example

A standard formula taught to all geometry students is the area of a circle ($A = \pi r^2$). In this program, we'll ask the user to enter the radius of a circle and print out the corresponding area and circumference ($C = 2\pi r$). We'll use the math library's value for pi in our calculations.

```
# Arup Guha
# 6/23/2012
# Calculates the area and circumference of a circle, given its radius.

import math

radius = int(input("What is the radius of your circle?\n"))

area = math.pi*(radius**2)
circumference = 2*math.pi*radius

print("The area of your circle is ",area,".",sep="")
print("The circumference of your circle is ",circumference,".",sep="")
```

Here is one sample of this program running:

```
>>>
What is the radius of your circle?
5
The area of your circle is 78.53981633974483.
The circumference of your circle is 31.41592653589793.
>>>
```

Koolaid Example Revisited

One of the most difficult issues we faced in the Koolaid program was correctly calculating the number of cups we had to sell to meet our profit goal. We avoided an "off by one error" with the following line of code:

```
num_cups = (target + profit_per_cup - 1) // profit_per_cup
```

If you carefully think about it, what we really wanted to do was a regular division between the variables target and profit_per_cup, but we just wanted to take the ceiling of that value! Namely, if we needed to sell 7.25 cups to hit our goal, we really couldn't sell .25 of a cup, we just have to sell the whole cup for a total of 8 cups! Now, with knowledge of the math library, we can rewrite this line of code to be much easier to understand, as follows:

```
num_cups = int(math.ceil(target/profit_per_cup))
```

Finally, note that the ceil function in the math library returns its answer as a float. Since we want to print out an integer, we can simply call the int function on the result we get from the ceil function.

Number of Possible Meals

Many restaurants boast that they offer many, many meal choices. Typically, this simply means that for the "meal", they can choose a certain number of items out of a whole batch. (For example, you may be allowed to choose 3 distinct side items out of 15 possible ones.) Mathematically, the number of ways to choose k items out of n (without repeats) is $\binom{n}{k}$, which is read as "n choose k". (Also, some books use the notation $_nC_k$ to denote a combination.) The formula for calculating a combination is as follows: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

For our program, let's assume that a combo meal includes a number of choices of appetizers out of a total AND a number of choices of entrees out of a total. We'll ask the user to enter the following four values:

1) Total possible number of appetizers
2) Number of appetizers allowed for the combo
3) Total possible number of entrees
4) Number of entrees allowed for the combo

We can use the factorial function to calculate both of the necessary combinations. After calculating both combinations, we can just multiply them to get our final answer, because for each possible choice of appetizers, we can match it with each possible choice of entrees. (You can visualize this with a 2 dimensional grid, with the rows labeled by all appetizers and the columns labeled by all entrees. Each square in the grid corresponds to a unique meal, and clearly the number of meals listed is the product of the number of rows and columns of the grid.)

In this example, we'll introduce one final modification to our program. In most programming languages, code starts by executing from a function main. While python doesn't require this, it's good practice to get into the habit of defining a function main anyway. It will help when transitioning to other languages and once the python programs you write exceed a few lines, organizationally it will be advantageous to have a main function. In order to incorporate this into your program, simply write the following line right before your program instructions:

```
def main():
```

Python requires consistent indenting, thus every statement inside of main must be indented. A standard indentation is 4 spaces or a tab. Once you've finished writing your code in main, you must call the function main, since what you've done is just define it. But, defining a function doesn't mean it will get used. It only gets used if it gets called. Here is how to call the function main:

```
main()
```

Putting this all together, we have the following program:

```
# Arup Guha
# 6/26/2012
# Calculates the number of possible combo meals.
import math

# Most languages define a function main, which starts execution.
def main():

    # Get the user information.
    numapps = int(input("How many total appetizers are there?\n"))
    yourapps = int(input("How many of those do you get to choose?\n"))
    numentrees = int(input("How many total entrees are there?\n"))
    yourentrees = int(input("How many of those do you get to choose?\n"))

    #Calculate the combinations of appetizers and entrees.
    appcombos = (math.factorial(numapps)/math.factorial(yourapps)
                /math.factorial(numapps-yourapps))
    entreecombos = (math.factorial(numentrees)/math.factorial(yourentrees)
                /math.factorial(numentrees-yourentrees))

    # Output the final answer.
    print("You can order", int(appcombos*entreecombos), "different meals.")

#Call main!
main()
```

One other new feature you'll notice in this program is a single line of code spanning two lines. This occurs with both the assignment statement for appcombos and entreecombos. In order to get python to recognize that the entire expression actually belongs on a single line, an extra set of parentheses are used. Though there are other ways to indicate multiple lines belong to a single line of code, this is the preferred way. The other two ways are splitting the line at a comma, if a comma is already in the expression and adding a single backward slash at the end of each physical line as a separate token. Here is an example of the latter:

```
appcombos = math.factorial(numapps)/math.factorial(yourapps) \
                /math.factorial(numapps-yourapps)
```

Each chapter will contain a set of 10 practice programs to help students reinforce the concepts in the chapter. These problem descriptions will not be as specific as a typical graded assignment for an introductory programming class. Rather, the goal of these problems is to give students simple ideas of programs to write and let the students fill in gaps in the problem description. Furthermore, instead of following the specifications in the problems exactly, one may and should make additional changes and/or additions to the problem description to create desired programs.

1) Write a program that prompts the user for the area of a circle and calculates the radius of that circle and prints it out.

2) Write a program that prompts the user to enter two points on the Cartesian plane with different x coordinates and calculate and output the slope between the two points. You may assume this slope is not undefined.

3) Write a program that asks the user for their hourly pay, the number of hours they work in a week, and the number of weeks they are working in the summer, as well as the cost of a single video game. Output the total number of video games they could buy, if they spent all of their earnings on video games, as well as the leftover money they would have. (The latter value must be strictly less than the cost of a game.) Assume all inputs are integers.

4) Write a program that asks the user for the price of gasoline per gallon, the number of gallons of gas currently in their car, the miles per gallon their car gets, and the length of their road trip in miles and calculates and prints out the amount the user will have to spend on extra gas to complete the road trip. (You may assume that the user will have to buy some gas to complete the trip.)

5) Write a program to guesstimate the total number of jelly beans in a right circular cylinder. In particular, the user must enter both the radius and the height of the cylinder, as well as the radius of the jellybean (we'll assume it's a sphere). For simplicity's sake, assume that the amount of volume a single jellybean takes up is simply the volume of the cube it would fit into. (Thus, if the radius of a jellybean is 2 units, then the total volume it takes up in the jar is 8 cubic units.) You should output a guess as to how many jellybeans are in the jar. Your guess need not be an integer. (If you want to enhance the program you may output the nearest integer to the actual value.)

6) Imagine a two-way straight railway and two trains approaching each other from opposite directions. You need to compute how long (in minutes) would it take for these trains to come side by side. You will also compute how many miles each train travels till then. Write a program to read in the distance (in miles), speeds of the trains (in miles per hour), and output the time to meet (in minutes) and the distance traveled by each train (in miles).

7) Write a program that calculates the cost of buying season football tickets. Ask the user for the number of upper bowl tickets (these are $25 per game) and lower bowl tickets (these are $50 per game) they desire, followed by the number of games in the season and the sales tax, entered as a percentage. Output the total cost of all the tickets, including tax.

8) Write a program that calculates the number of pictures that can be stored on a thumb drive. Ask the user to enter the number of gigabytes of data the thumb drive can store, as well as the length and width of each picture, in pixels. Assume that each pixel takes 3 bytes of storage. (This isn't really the case, since most pictures are stored in a compressed format.) Output your answer as a whole number.

9) Research how electoral votes are calculated, look up the most current census numbers and use this to write an electoral vote calculator. Your calculator should ask the user for the population of their state and output the number of electoral votes that state should receive. Since there are some slight exceptions in the actual process, compared to the straight mathematical theory behind it, your program will occasionally be off by 1 electoral vote compared to the actual allocation.

10) Write a program that prompts the user to enter the coefficients a, b and c from the quadratic equation $ax^2 + bx + c = 0$, and calculates the two real roots of the equation. You may assume that the user enters values of a, b and c that lead to the equation having two distinct real roots, namely, the user will enter values such that $b^2 > 4ac$, and a will not equal 0.