

So far, we have been using Scratch as the programming language we use to create and implement algorithms programmatically. While Scratch is easy to code in and allows us to come up with some pretty cool programs without requiring too much background in programming, the time has come for us to transition to a more general purpose programming language.

General purpose programming languages are more robust, and can (and are) used in more situations than educational programming languages like Scratch. Think of it like this: using a programming language like Scratch is like building a Lego house only using 2x4 Lego pieces. While it is possible to do so, there is a limitation on what kinds of houses you can build. Conversely, using more general purpose programming languages is like building a house with any kind of Lego piece you can think up in your mind. There are fewer limitations, and the kinds of houses you can build are limitless. In this course, we will use Python as the general purpose programming language.

**Why Python?**

You may have heard about other general purpose programming languages: Java, C, C++, C#, Visual Basic, and so on. So why use Python instead of, say, Java? In the end, it amounts to the simple idea that, unlike all of the other general purpose programming languages listed above, Python allows us to create powerful programs with limited knowledge about syntax, therefore allowing us to focus on problem solving instead. In a sense, Python is logical. That is, nothing must be initially taken on faith (that will ostensibly be explained at a later time). There isn't any excess baggage that's required in order to begin to write even simple Python programs.

Recall how, in geometry, the formula for calculating the volume of a cone was given. At that time, it was simply inexplicable. That is, you were most likely told to memorize it. It is not until a calculus course that this formula is actually derived, and how it came to be is fully explained. Why? Well, it is simply because it requires calculus in order to do so. Most students taking a geometry course have not yet had calculus; however, formulas for calculating the volume of various objects (including a cone) are typical in such a course. The problem, of course, is that we are told to take it on faith that it, in fact, works as described. We are told that, how it works and how it was derived, will be explained at a later time. The problem with this is that it forces memorization of important material as opposed to a deep understanding of it (which, in the end, is the goal).

A similar thing actually occurs in a lot of programming languages. Often, we must memorize syntax that will be explained later. Python is unique in that it does a pretty good job of taking all of that out by just being simple. Programming in Python is immediately logical and explicable.

Take the following simple example of a program that displays the text, "Programming rules, man!" in various general purpose programming languages:

In Java:

```
public class SimpleProgram
{
    public static void main(String[] args)
    {
        System.out.println("Programming rules, man!");
    }
}
```

In C:

```
#include <stdio.h>

int main()
{
    printf("Programming rules, man!\n");
}
```

In C++:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Programming rules, man!" << endl;
}
```

In C#:

```
public class SimpleProgram
{
    public static void Main()
    {
        System.Console.WriteLine("Programming rules, man!");
    }
}
```

In Visual Basic:

```
Module Hello
    Sub Main()
        MsgBox("Programming rules, man!")
    End Sub
End Module
```

And in Python:

```
print "Programming rules, man!"
```

In all of these examples, compiling and running the programs (or interpreting them) produces a single line of output text: “Programming rules, man!” Did you notice that, in all of the examples (except for Python), there seems to be a good bit of seemingly extra stuff for such a simple program? There are a

lot of words that you may not be familiar with or immediately understand: `class`, `public`, `static`, `void`, `main/Main`, `#include`, `printf`, `cout`, `namespace`, `String[]`, `endl`, `Module`, `Sub`, `MsgBox`, and so on. In fact, the only readable version to a beginner is usually the one written in Python. It is pretty evident that the statement `print "Programming rules, man!"` means to display that string of characters to the screen (or console).

Python is extremely readable because it has very simple and consistent syntax. This makes it perfect for beginner programmers. It also forces good coding practices and style, something that is very important for beginners (especially when it comes to debugging and/or maintaining programs). Python has a large set of libraries that provide powerful functionality to do just about anything. Libraries allow Python programmers to use all kinds of things that others have created (i.e., we don't have to reinvent the wheel). A huge benefit of Python is that it is platform independent. It doesn't matter what operating system you use, it is supported with minimal setup and configuration, and there is no need to deal with dependencies (i.e., other things that are required in order to just begin to code in Python).

Don't think that, because of its simplicity, Python is therefore not a powerful language (or perhaps that it doesn't compete with Java or C++). Python is indeed powerful, and can do everything that other programming languages can do (e.g., it does support the object-oriented paradigm).

#### Did you know?

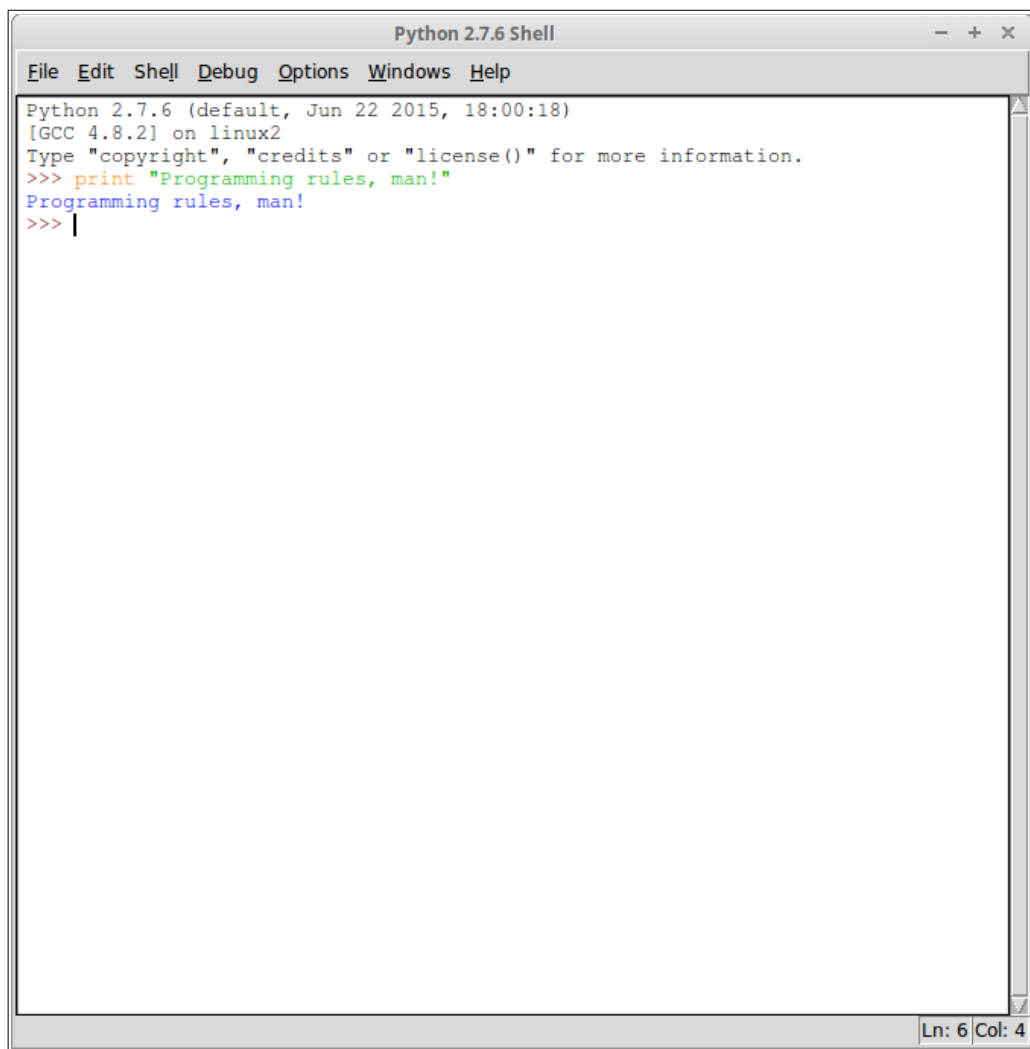
The name of the Python programming language is taken from a television series called Monty Python's Flying Circus (and not from the snake).

#### Integrated development environment

Many programmers write their programs using some sort of text editor (usually a simplistic one, albeit with useful characteristics such as syntax highlighting). In fact, some write programs at the command line (in the terminal) using nothing but a text-based text editor (i.e., without graphical characteristics). Most programmers, however, use an IDE (Integrated Development Environment).

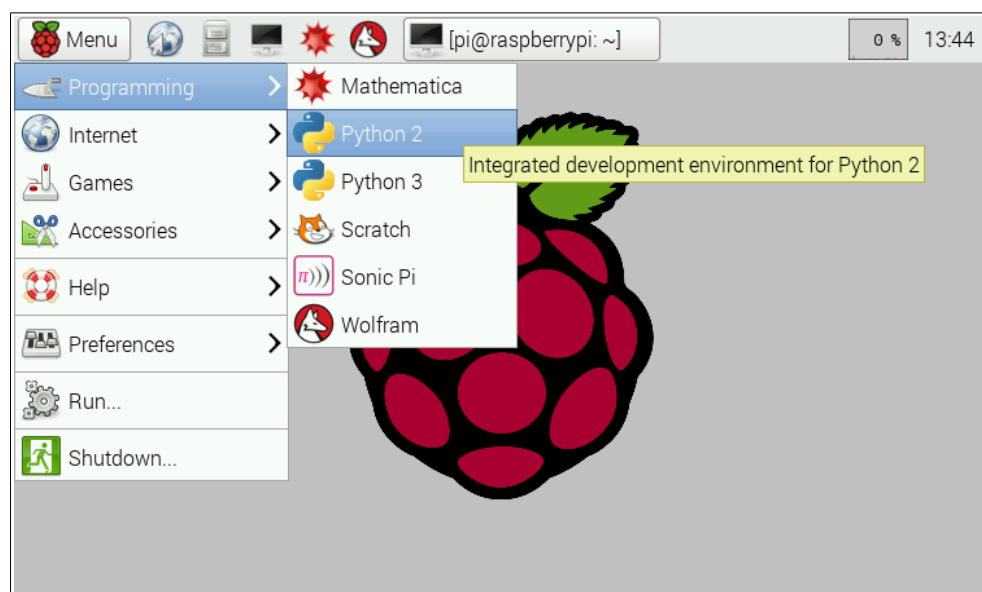
**Definition:** *An **Integrated Development Environment (IDE)** is a piece of software that allows computer programmers to design, execute, and debug computer programs in an integrated and flexible manner.*

On the Raspberry Pi, the IDE used to design Python programs is called IDLE (which stands for Python's Integrated Development Environment). Other IDEs exist for pretty much all of the most used general purpose programming languages: Eclipse, Visual Studio, Code::Blocks, NetBeans, Dev-C++, Xcode, and so on. In fact, many of these IDEs support more than one language (some natively, others by installing additional plug-ins or modules)! Here's an image of IDLE with the program shown earlier implemented (and executed):

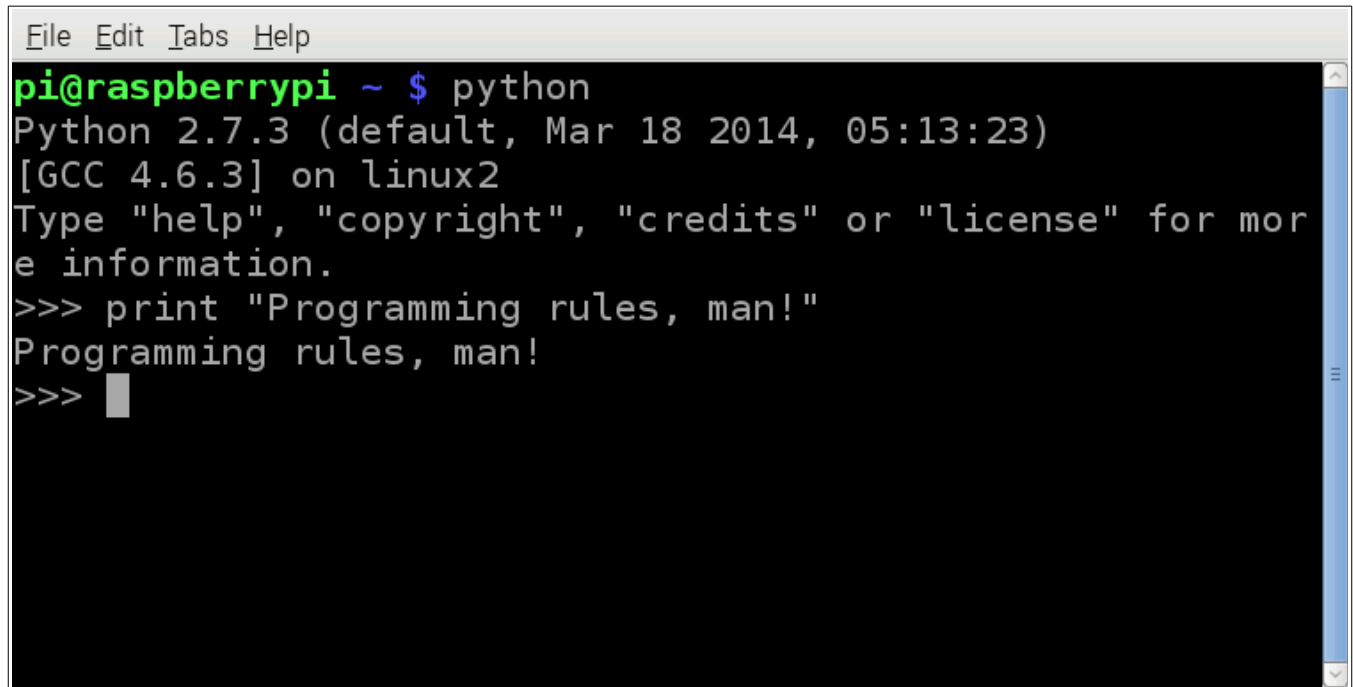
A screenshot of a terminal window titled "Python 2.7.6 Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The terminal text shows the Python version (2.7.6), GCC version (4.8.2), and the OS (linux2). It displays the prompt "Type 'copyright', 'credits' or 'license()' for more information." followed by a command to print "Programming rules, man!". The output is "Programming rules, man!". The cursor is on the next line. The status bar at the bottom right shows "Ln: 6 Col: 4".

```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> print "Programming rules, man!"
Programming rules, man!
>>> |
```

On the Raspberry Pi, IDLE can be launched as follows:



Python programs can also be created and executed at the command line (or terminal). We do so by launching a terminal and typing **python**, which brings up the Python shell:

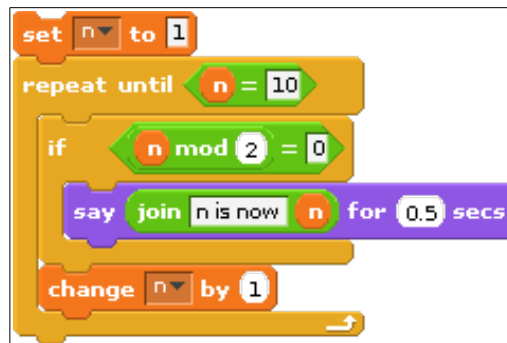


```
File Edit Tabs Help
pi@raspberrypi ~ $ python
Python 2.7.3 (default, Mar 18 2014, 05:13:23)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print "Programming rules, man!"
Programming rules, man!
>>> █
```

### Scratch vs Python

In a previous lesson, you learned that programs written in a programming language are either compiled (to machine language so that a computer can execute them directly) or interpreted, statement-by-statement (in a sense, you could say that programs written in interpreted languages are compiled, line-by-line, in real time). Python is an interpreted language that implements the imperative paradigm. That is, programs are designed as a sequence of instructions (called statements) that can be followed to complete a task.

Let's take a look at a simple program in Scratch and see how it compares to the same thing in Python:



What does this program do? Simply put, it displays the numbers 2, 4, 6, and 8. Take a look at the script above. The variable  $n$  is initially set to 1. A *repeat-until* loop is executed so long as  $n$  is less than 10 (i.e., 1 through 9). Each time the body of the loop is executed, the string “ $n$  is now (plus the value of  $n$ )” is displayed if  $n$  is evenly divisible by 2. For example, if  $n$  is 4, then the string  **$n$  is now 4** is displayed.

Recall that the **mod** operator returns the remainder of a division. Therefore, when  **$n \bmod 2 = 0$**  is true, it means that the remainder of  $n$  divided by 2 is zero – so  $n$  must be even! At the end of the body of the loop, the variable  $n$  is incremented (ensuring that  $n$  will eventually reach the value 10, and we will break out of the *repeat-until* loop).

Here's how this can be similarly done in Python:

```
n = 1
while n < 10:
    if n % 2 == 0:
        print "n is now " + str(n)
    n = n + 1
```

At this point, it is fine if you don't understand everything that's going on syntactically. The idea is simply to illustrate how Scratch and Python differ (and are similar!). But let's try to explain. The block, **set  $n$  to 1**, in Scratch is implemented in Python as, `n = 1`. Pretty similar! Python has no *repeat-until* repetition construct. Instead, we can use a *while* construct with a modified condition. Repeating a task until a variable (in this case,  $n$ ) is 10 is the same thing as repeating it while the variable is less than 10. *If-statements* are similar; however, the **mod** and **equality** operators differ. In Python, we check for equality using the double-equal (`==`) operator. The mod operator is a percent sign (`%`). So the block, **if  $n \bmod 2 = 0$** , in Scratch can be implemented in Python as, `if n % 2 == 0`. Generating the output, “ $n$  is now 4,” for example, can be implemented in Scratch using the familiar **print** statement: `print "n is now 4"`. Of course, we don't always want to display that  $n$  is 4. So we concatenate (or join) the value of  $n$  to the string “ $n$  is now ” just as we did in Scratch. However, since  $n$  is not a string of characters (i.e., it is a number – an integer to be precise), then it must first be converted to a string before being concatenated to another string. This is what `str(n)` does. Finally, the value of  $n$  is incremented by 1 with the statement `n = n + 1`.

In Scratch, it is easy to see the blocks that belong in the body of a repetition construct. The puzzle pieces intrinsically capture this (i.e., they are quite literally visible inside the *repeat-until* block in the script above). In Python, we denote statement hierarchy (i.e., if statements belong in the body of a construct such as a *while* loop) by using indentation. Note how it is quite clear which statements belong in the body of the *while* loop above: the *if-statement* and the statement that increments the variable  $n$  by 1. Note that the *print* statement is inside the true part of the *if-statement* (this is evident by how it is directly beneath the *if-statement* and indented further to the right). Again, at this point it is fine to have a minimal grasp of Python's syntax.

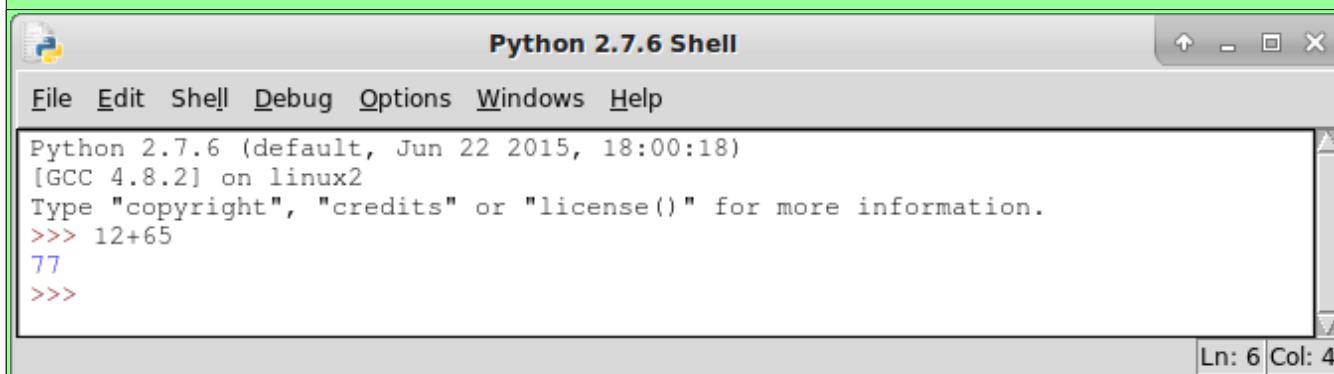
### Activity 1: Python Primer

In this activity, you (or the prof) will experiment with the Python IDE in order to get a basic understanding of and experience with Python.

First, bring up the Python IDE (IDLE). Formally, we call this the Python shell, an active Python interpreter environment. It's quite useful as it can be used to evaluate expressions in real time (i.e., without saving a program to a file first).

## Simple arithmetic expressions

Let's first begin with some simple arithmetic expressions to see how the Python shell can evaluate them and provide real time results. Here's an example of Python evaluating a simple expression ( $12 + 65$ ) and providing the result in IDLE:



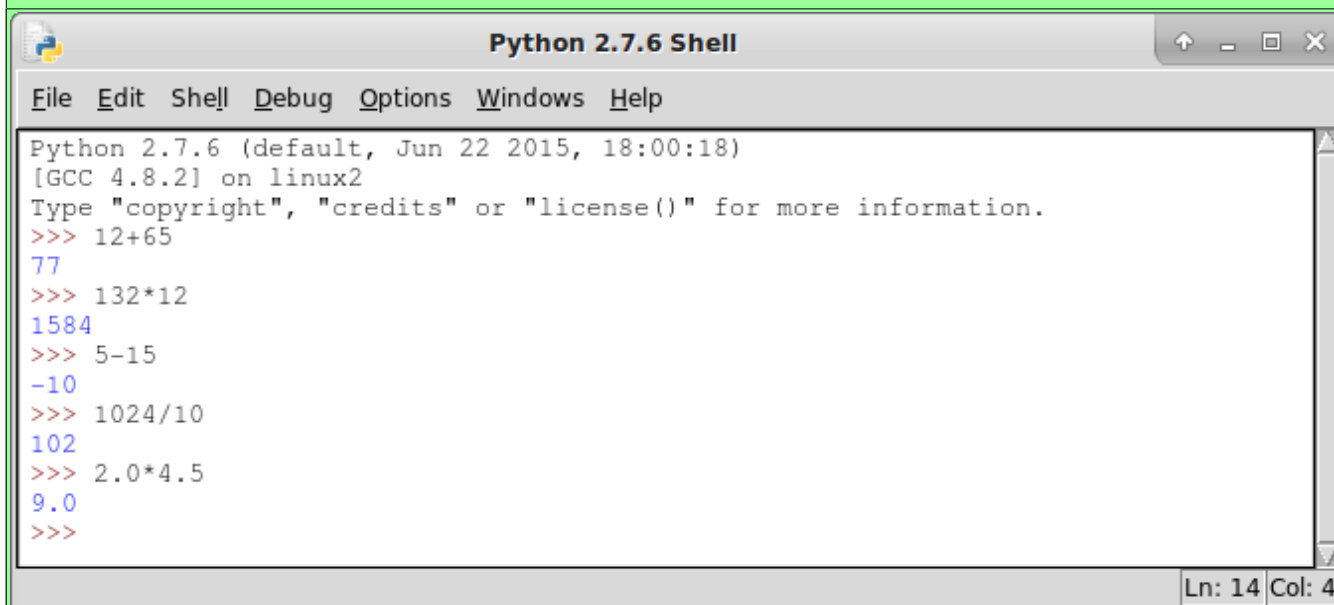
The screenshot shows the Python 2.7.6 Shell window. The title bar is "Python 2.7.6 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main text area contains the following text:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 12+65
77
>>>
```

The status bar at the bottom right shows "Ln: 6 Col: 4".

Note that the size of the IDLE window has been reduced in this document.

Python evaluates the expression,  $12 + 65$ , and provides the result in the Python shell. Here are more examples:



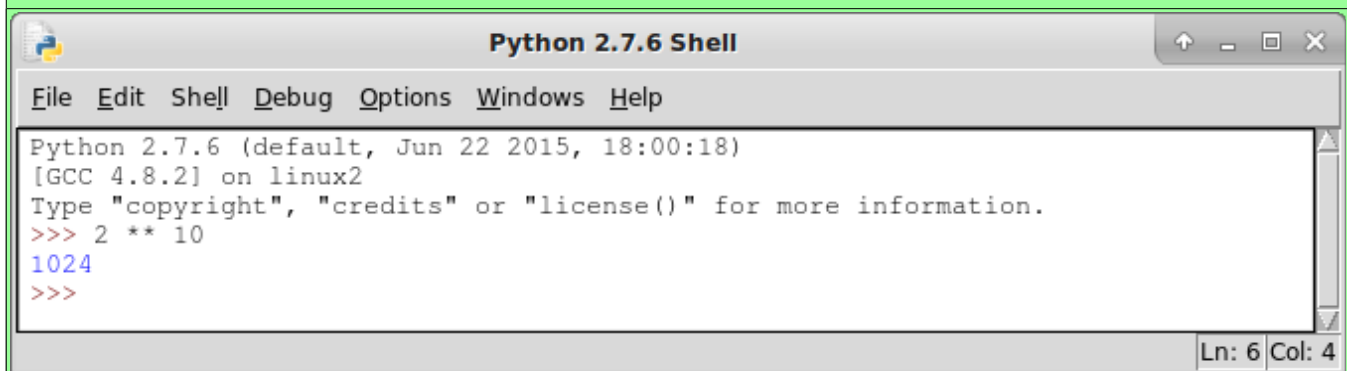
The screenshot shows the Python 2.7.6 Shell window. The title bar is "Python 2.7.6 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main text area contains the following text:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 12+65
77
>>> 132*12
1584
>>> 5-15
-10
>>> 1024/10
102
>>> 2.0*4.5
9.0
>>>
```

The status bar at the bottom right shows "Ln: 14 Col: 4".

Verify that the expressions are indeed correct (e.g.,  $5 - 15 = -10$ ,  $2.0 * 4.5 = 9.0$ , etc).

So far, you have seen the four main arithmetic operators (i.e., +, -, \*, and /). Take a look at this expression:

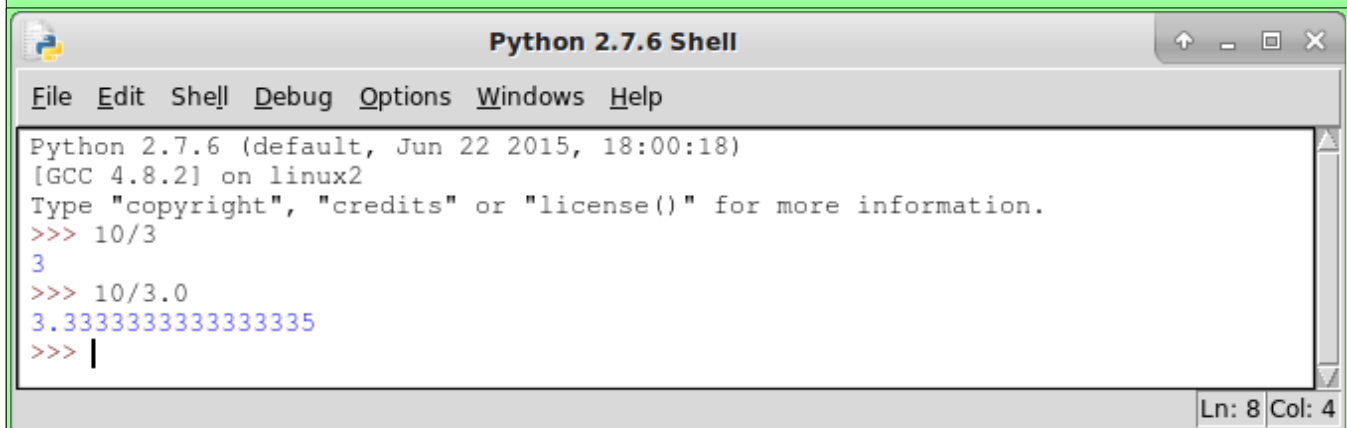


```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 2 ** 10
1024
>>>
```

Ln: 6 Col: 4

What does the \*\* operator do? It performs exponential (power) calculation on the two operands. The expression `2 ** 10` implies two raised to the tenth power (or  $2^{10}$ ), which is indeed 1024.

Now, take a look at the following expressions:



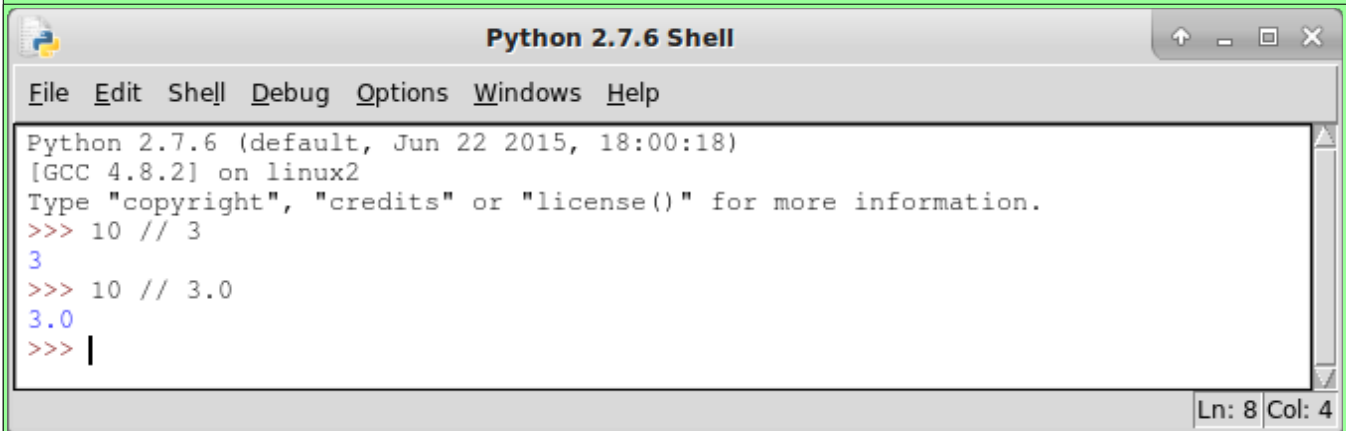
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 10/3
3
>>> 10/3.0
3.3333333333333335
>>> |
```

Ln: 8 Col: 4

Note how the expression `10 / 3` results in 3. The decimal portion of the result (which we calculate to precisely be 3.33333...) seems to be truncated. In fact, the / operator in Python returns integer division if the two operands the operator is being applied to are both integers. That is, it returns the integer portion only (i.e., the quotient) of a division of two integers. To perform floating point division, at least one of the operands must then be floating point. This is shown in the second example above.



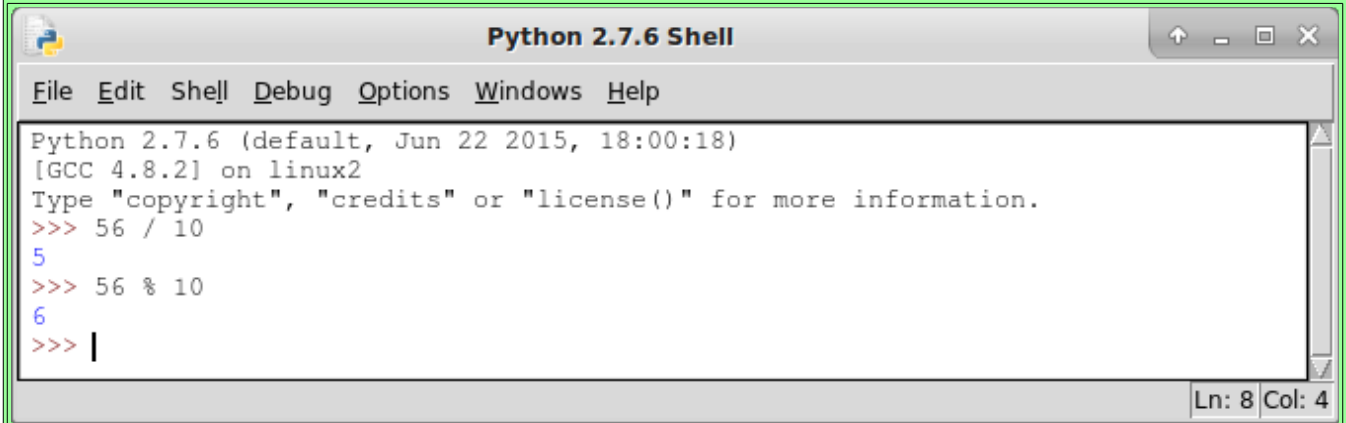
We can force integer division regardless of the type of operands with the `//` operator as follows:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 10 // 3
3
>>> 10 // 3.0
3.0
>>> |
```

Ln: 8 Col: 4

There is one more arithmetic operator in Python: the `%` operator. This operator returns the remainder of a division. Take a look at the following example:



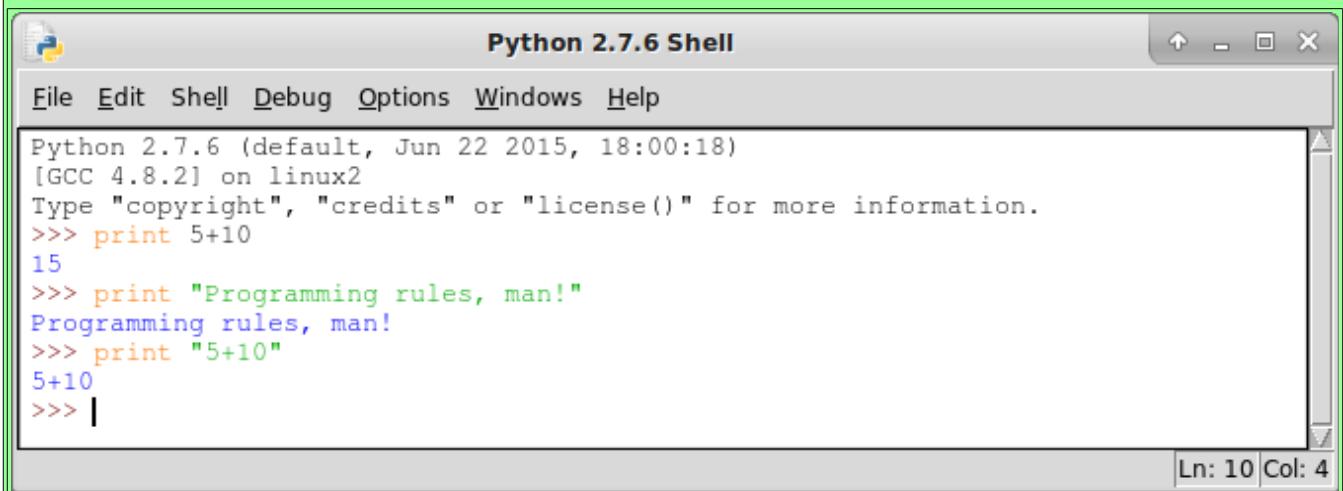
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> 56 / 10
5
>>> 56 % 10
6
>>> |
```

Ln: 8 Col: 4

The expression `56 / 10` is indeed 5 (the `/` operator produces an integer since both of the operands, 56 and 10, are integers). The remainder that results from the expression `56 / 10` is 6 (since  $56 / 10 = 5$  remainder of 6). Therefore, `56 % 10 = 6`.

## Output

As seen earlier, Python allows output via a **print** statement. Here are some simple examples:

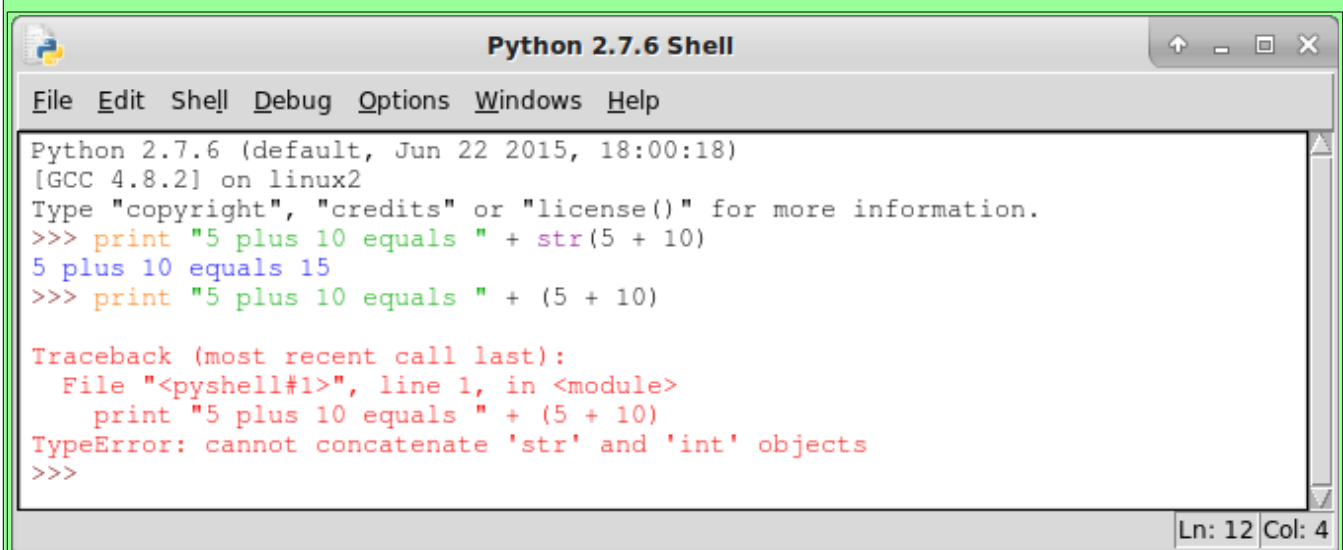


```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> print 5+10
15
>>> print "Programming rules, man!"
Programming rules, man!
>>> print "5+10"
5+10
>>> |
```

Ln: 10 Col: 4

The statement `print 5 + 10` instructs Python to display the result of the expression `5 + 10` (which is 15). The statement `print "Programming rules, man!"` does just what it did when shown earlier. Take a look at the last statement: `print "5+10"`. It looks suspiciously like the first statement, except that the expression `5 + 10` is enclosed in quotes. This lets the Python interpreter know that the characters `"5 + 10"` are to be interpreted as a string (characters strung together) as opposed to an arithmetic expression consisting of the `+` operator and the two operands, 5 and 10. This is why the output of this statement is, quite literally, `5 + 10`.

Suppose that you would like to print the following string of characters: 5 plus 10 equals 15 (and that you would like for 15 to be calculated as the result of the expression `5 + 10`). This can be accomplished as follows:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> print "5 plus 10 equals " + str(5 + 10)
5 plus 10 equals 15
>>> print "5 plus 10 equals " + (5 + 10)

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print "5 plus 10 equals " + (5 + 10)
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Ln: 12 Col: 4

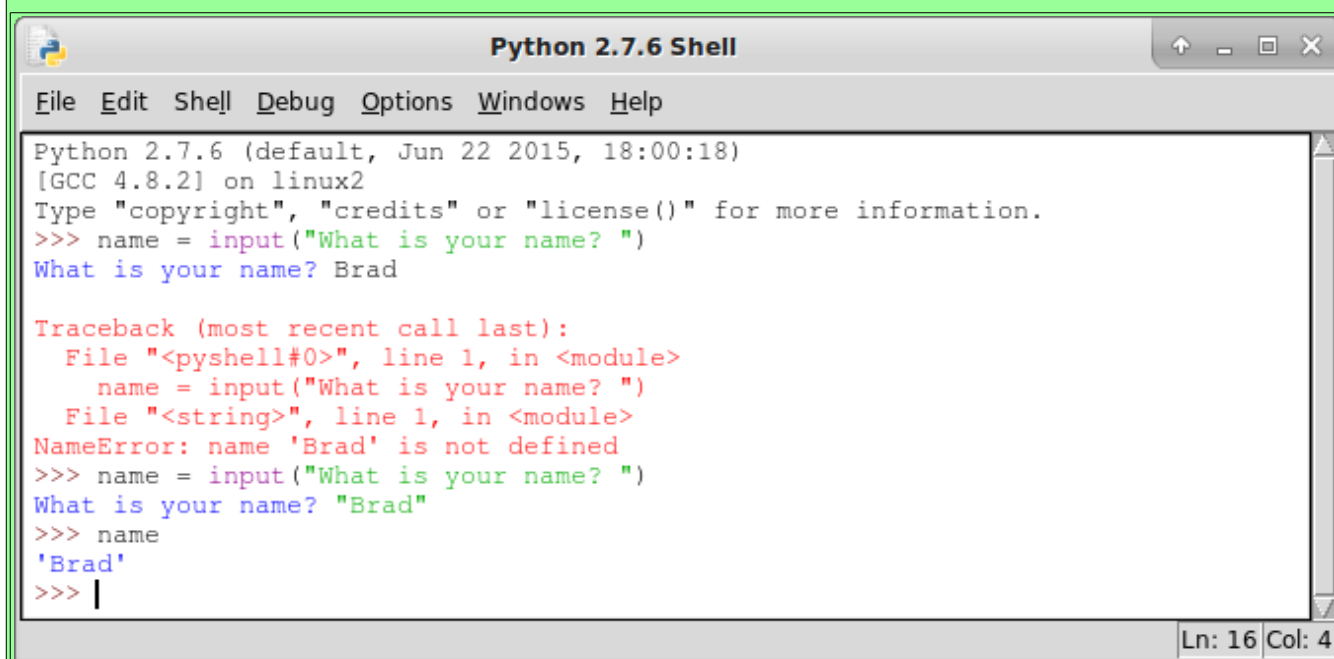
First, note the error produced by the following statement: `print "5 plus 10 equals " + (5 + 10)`. The error occurs because Python does not know how to “add” or concatenate the string `"5 plus`

10 equals " to the integer that results from the expression (5 + 10). To instruct the Python interpreter to concatenate the result of this expression (as characters) to the first part of the string, the result (15) must be converted to a string. In Python, this is accomplished with the **str()** function. Python converts anything within the parentheses (we call this the parameters of the function) to a string of characters. Therefore, the expression `str(5 + 10)` instructs the Python interpreter to first add 5 and 10 (to produce the result 15), and then convert 15 to the string "15". It is then valid to concatenate the string "5 plus 10 equals " to the string "15".

You may also have noticed that, for most of the examples, operands and operators were separated by a space. For example, the expression `5+10` was written in the Python shell as `5 + 10` (with spaces). This is an example of good coding style that increases the readability of our programs.

## Input

Python also supports statements that allow users to input information via the **input()** function. This information is typically stored in variables. Take a look at the following example:

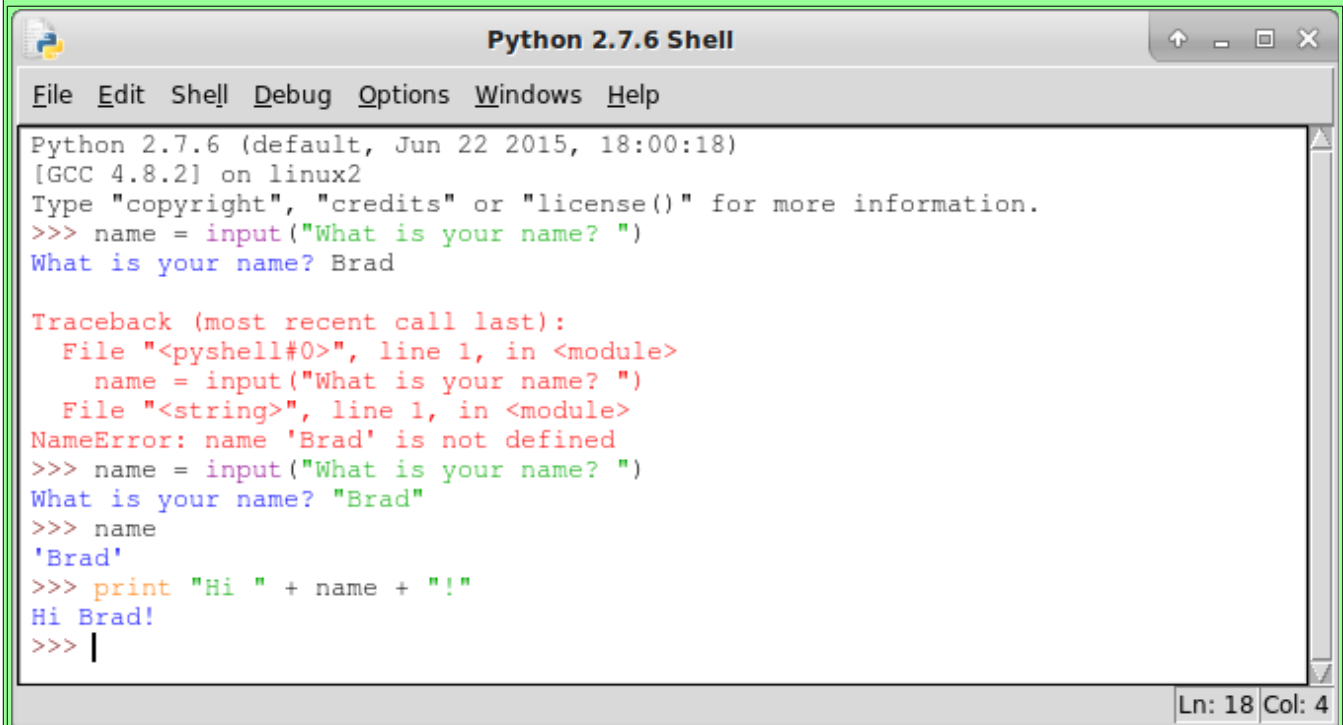
A screenshot of a Python 2.7.6 Shell window. The window title is "Python 2.7.6 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The shell output shows the Python version and GCC version, followed by a prompt to type "copyright", "credits", or "license()". The user enters a command to assign the input of "What is your name? " to a variable named 'name'. The user then enters "Brad". This causes a NameError because the variable 'Brad' is not defined. The error message is displayed in red. The user then enters the command to assign the input of "What is your name? " to a variable named 'name' again, but this time with quotes around the input string. The user then enters the command to display the value of 'name', which outputs 'Brad'. The status bar at the bottom right shows "Ln: 16 Col: 4".

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    name = input("What is your name? ")
  File "<string>", line 1, in <module>
NameError: name 'Brad' is not defined
>>> name = input("What is your name? ")
What is your name? "Brad"
>>> name
'Brad'
>>> |
```

The statement, `name = input("What is your name? ")`, prompts the user for a name. It stores the result in the variable *name*. Note that it expects any type. We could have very well entered in 23 as the response, and the variable *name* would have stored the integer 23. But, of course, we wanted to enter an actual name. Since a name is a string of characters, then we must make sure to enclose the response with quotes. What happens if we forget to do so? Well, we get the error shown above. In the end, we can display the contents of the variable *name* simply by typing its name (i.e., *name*) in the Python shell. this is also shown above.

We can now use the variable *name* as follows:



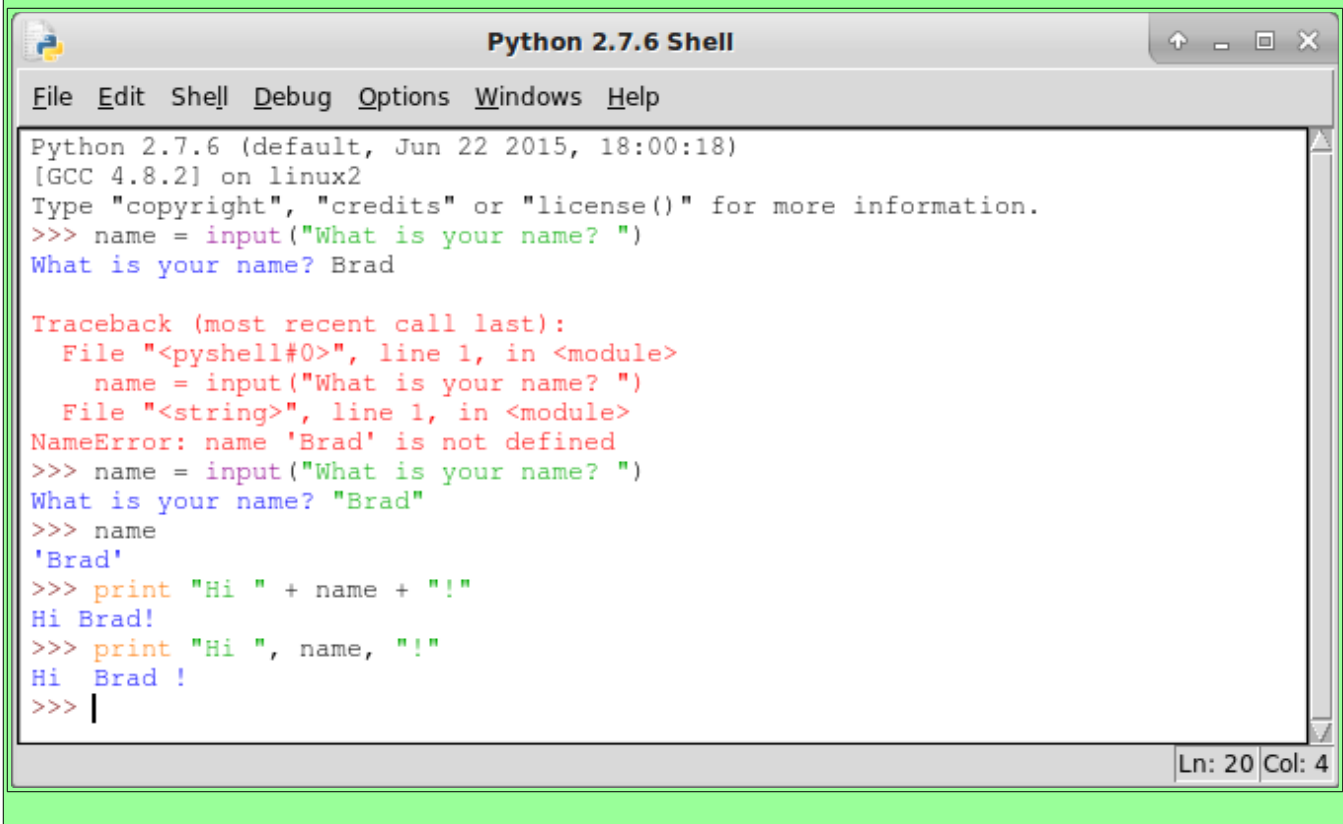
A screenshot of a Python 2.7.6 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The main text area shows the following code and output:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    name = input("What is your name? ")
  File "<string>", line 1, in <module>
NameError: name 'Brad' is not defined
>>> name = input("What is your name? ")
What is your name? "Brad"
>>> name
'Brad'
>>> print "Hi " + name + "!"
Hi Brad!
>>> |
```

The status bar at the bottom right indicates 'Ln: 18 Col: 4'.

That's another example of string concatenation. The **print** statement has another format:



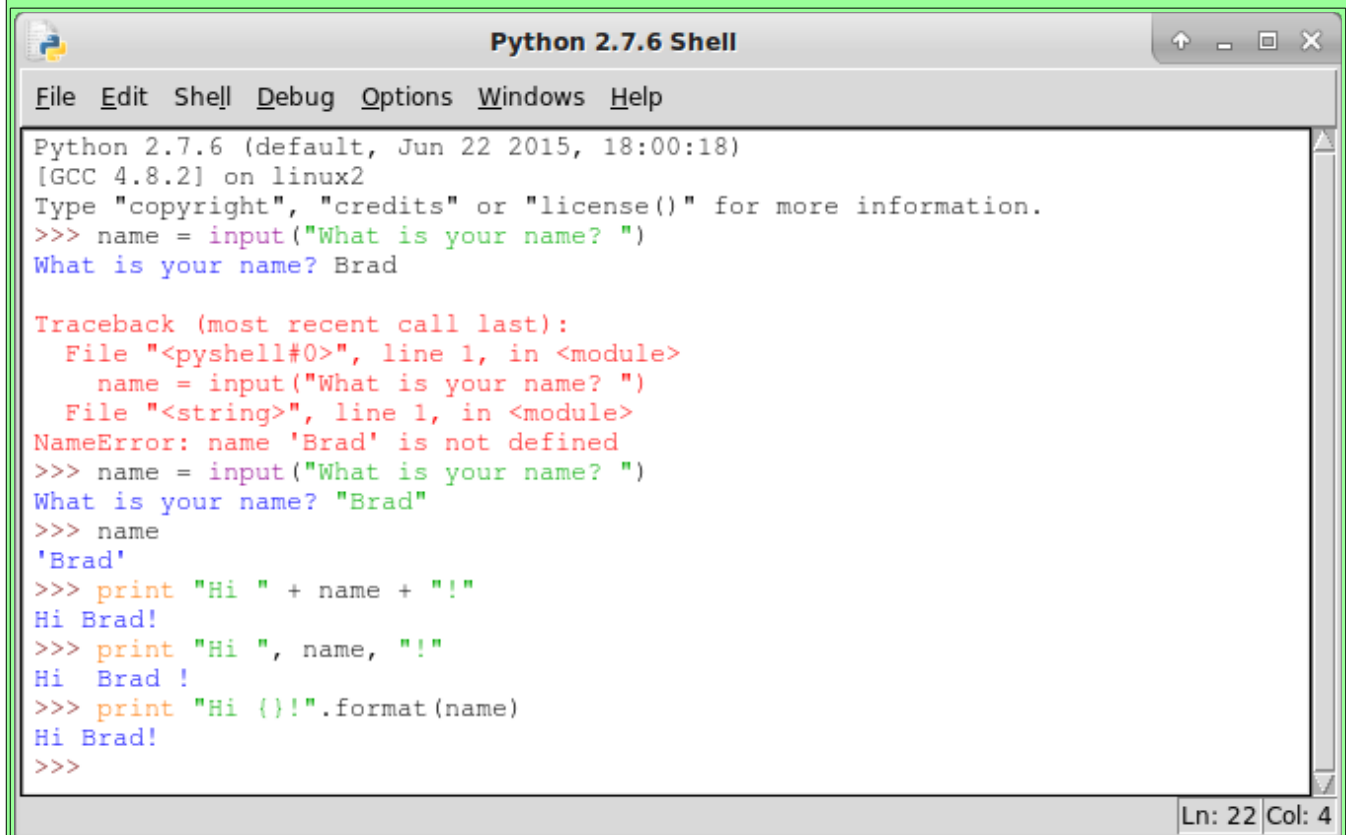
A screenshot of a Python 2.7.6 Shell window, similar to the one above. The main text area shows the following code and output:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    name = input("What is your name? ")
  File "<string>", line 1, in <module>
NameError: name 'Brad' is not defined
>>> name = input("What is your name? ")
What is your name? "Brad"
>>> name
'Brad'
>>> print "Hi " + name + "!"
Hi Brad!
>>> print "Hi ", name, "!"
Hi Brad !
>>> |
```

The status bar at the bottom right indicates 'Ln: 20 Col: 4'.

Note how we can separate the components of what we want output with commas. But note the difference! Apparently, Python inserts a space in between each component when the **print** statement is formatted in this manner. If this is not desired, we can modify the statement as follows:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad

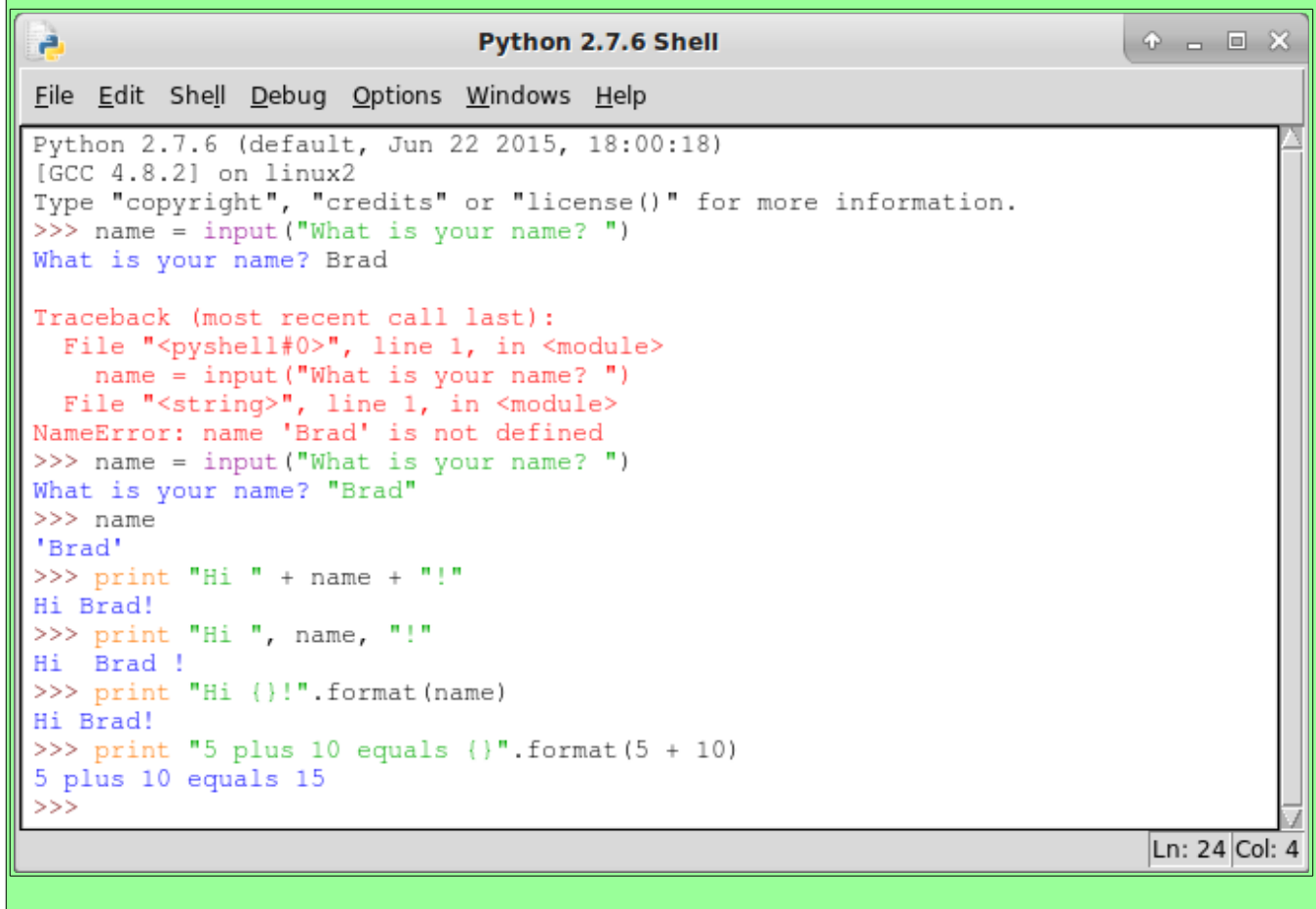
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    name = input("What is your name? ")
  File "<string>", line 1, in <module>
NameError: name 'Brad' is not defined
>>> name = input("What is your name? ")
What is your name? "Brad"
>>> name
'Brad'
>>> print "Hi " + name + "!"
Hi Brad!
>>> print "Hi ", name, "!"
Hi Brad !
>>> print "Hi {}".format(name)
Hi Brad!
>>>
```

Ln: 22 Col: 4

The braces ({} ) within a string are known as format fields. They are intended to note that something belongs there (that will be specified at a later time). To specify the contents to replace the braces with, we execute the **format** method on the string with the format field. We provide the values (which, in the example above, is just the contents of the variable *name*) that will be formatted to a string and replace the format fields. In the example above, the contents of the variable *name* is converted to a string (if necessary) and inserted in the string over the braces. this results in the output, “Hi Brad!”

Here's another example of this with the following statement:

```
print "5 plus 10 equals {}".format(5 + 10):
```



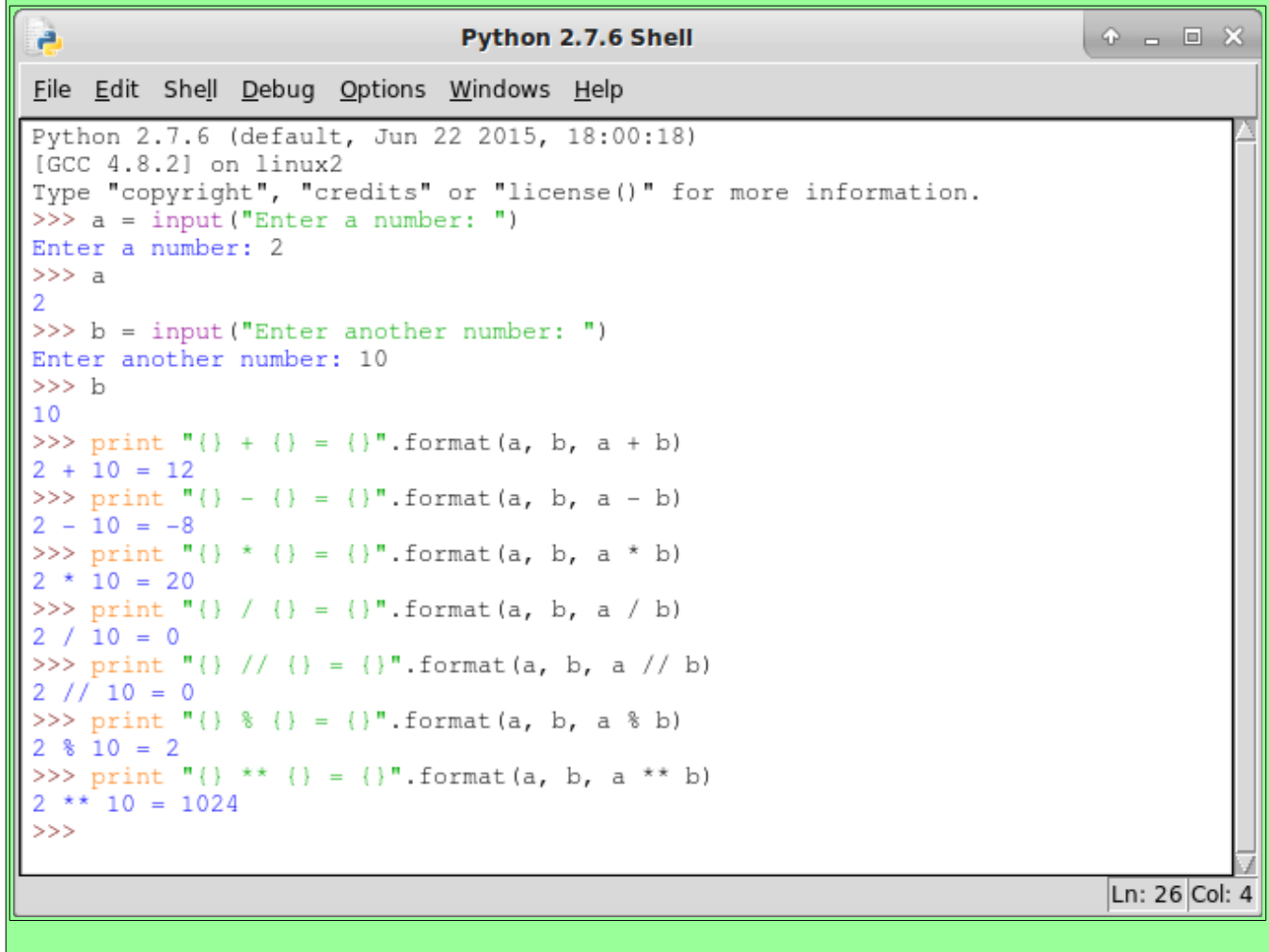
The screenshot shows a terminal window titled "Python 2.7.6 Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The terminal output is as follows:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    name = input("What is your name? ")
  File "<string>", line 1, in <module>
NameError: name 'Brad' is not defined
>>> name = input("What is your name? ")
What is your name? "Brad"
>>> name
'Brad'
>>> print "Hi " + name + "!"
Hi Brad!
>>> print "Hi ", name, "!"
Hi Brad !
>>> print "Hi {}".format(name)
Hi Brad!
>>> print "5 plus 10 equals {}".format(5 + 10)
5 plus 10 equals 15
>>>
```

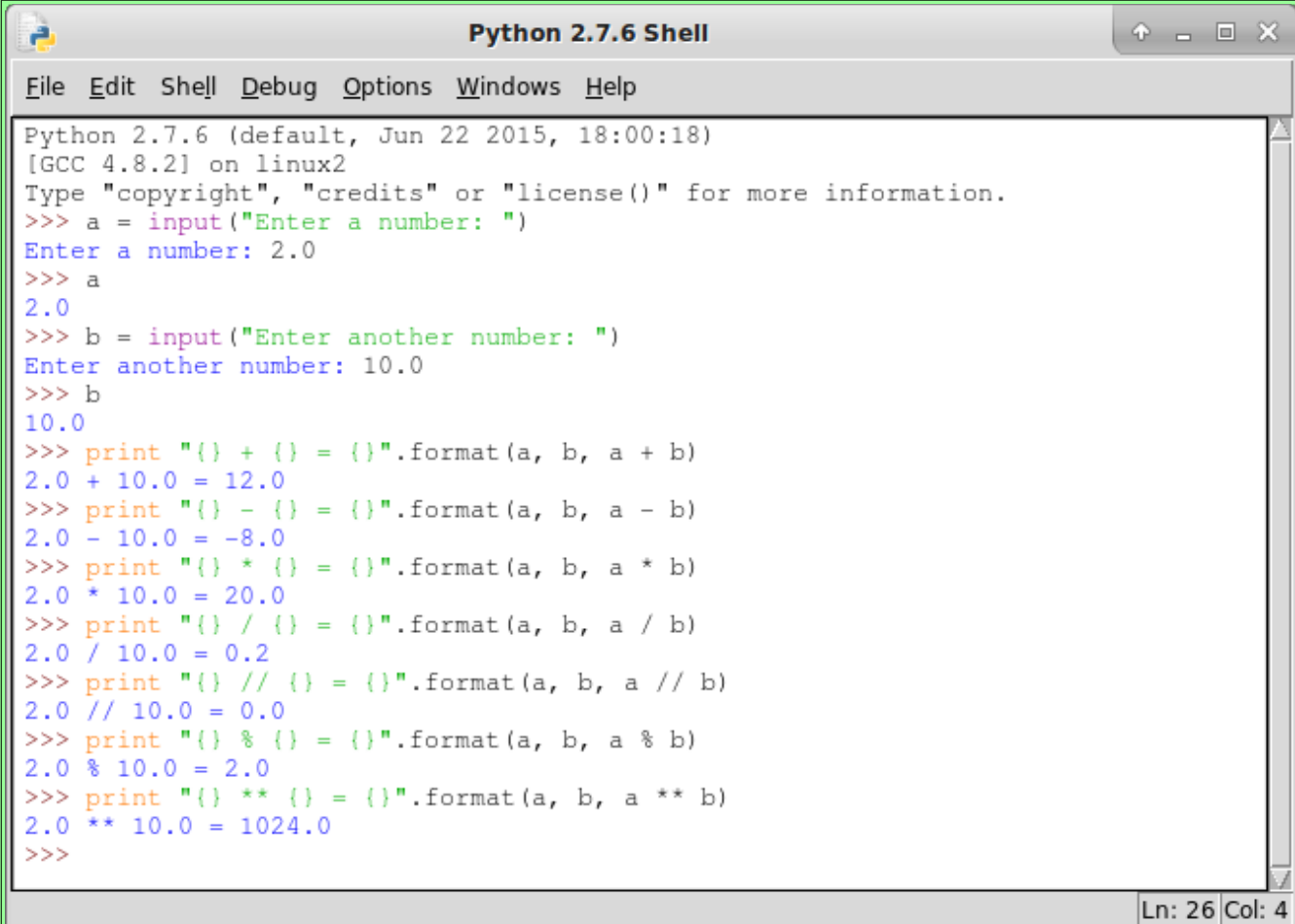
The status bar at the bottom right of the window shows "Ln: 24 Col: 4".

Here are more examples of this with arithmetic expressions:

A screenshot of a Python 2.7.6 Shell window. The window has a title bar with the text "Python 2.7.6 Shell" and standard window controls (up arrow, minus, square, X). Below the title bar is a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main area of the window contains a Python interpreter session. It starts with the version and environment information: "Python 2.7.6 (default, Jun 22 2015, 18:00:18) [GCC 4.8.2] on linux2". Then it shows the user entering a number 2 for variable 'a' and another number 10 for variable 'b'. Finally, it shows a series of print statements that output arithmetic expressions using the variables a and b, such as "2 + 10 = 12", "2 - 10 = -8", "2 \* 10 = 20", "2 / 10 = 0", "2 // 10 = 0", "2 % 10 = 2", and "2 \*\* 10 = 1024". The bottom right corner of the window shows "Ln: 26 Col: 4".

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a = input("Enter a number: ")
Enter a number: 2
>>> a
2
>>> b = input("Enter another number: ")
Enter another number: 10
>>> b
10
>>> print "{} + {} = {}".format(a, b, a + b)
2 + 10 = 12
>>> print "{} - {} = {}".format(a, b, a - b)
2 - 10 = -8
>>> print "{} * {} = {}".format(a, b, a * b)
2 * 10 = 20
>>> print "{} / {} = {}".format(a, b, a / b)
2 / 10 = 0
>>> print "{} // {} = {}".format(a, b, a // b)
2 // 10 = 0
>>> print "{} % {} = {}".format(a, b, a % b)
2 % 10 = 2
>>> print "{} ** {} = {}".format(a, b, a ** b)
2 ** 10 = 1024
>>>
```

Try to do the same thing as in the example above, except set  $a=2.0$  and  $b=10.0$ . Notice any differences?



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a = input("Enter a number: ")
Enter a number: 2.0
>>> a
2.0
>>> b = input("Enter another number: ")
Enter another number: 10.0
>>> b
10.0
>>> print "{} + {} = {}".format(a, b, a + b)
2.0 + 10.0 = 12.0
>>> print "{} - {} = {}".format(a, b, a - b)
2.0 - 10.0 = -8.0
>>> print "{} * {} = {}".format(a, b, a * b)
2.0 * 10.0 = 20.0
>>> print "{} / {} = {}".format(a, b, a / b)
2.0 / 10.0 = 0.2
>>> print "{} // {} = {}".format(a, b, a // b)
2.0 // 10.0 = 0.0
>>> print "{} % {} = {}".format(a, b, a % b)
2.0 % 10.0 = 2.0
>>> print "{} ** {} = {}".format(a, b, a ** b)
2.0 ** 10.0 = 1024.0
>>>
```

Ln: 26 Col: 4

Other than the variables and the results being expressed as decimals (i.e., floating point numbers), the statement, `print "{} / {} = {}".format(a, b, a / b)`, results in the exact result, 0.2, since at least one of the operands is a floating point value.

### Creating programs and saving files

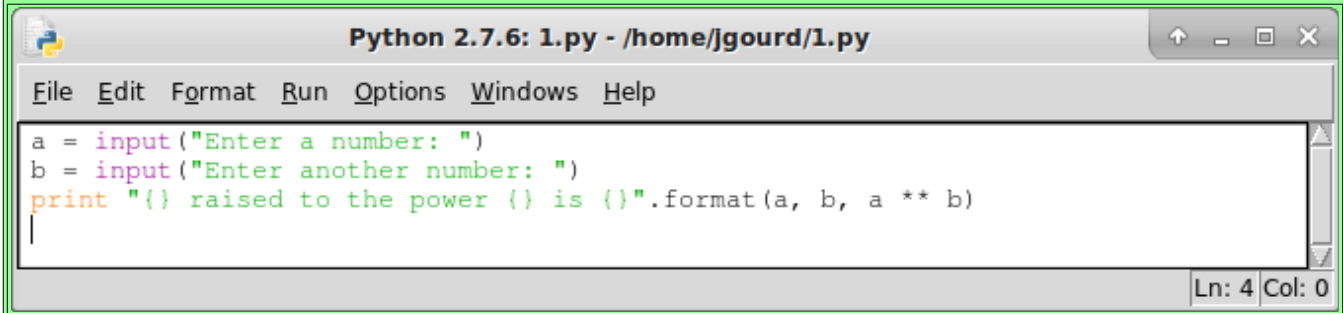
So far, we have been entering statements in the Python shell. These statements have been interpreted, one at a time. If we were to close the Python shell, everything that we entered would be lost. In order to save Python programs, we must type them in a separate editor outside of the Python shell, save them in a file. Once this has been done, we can then execute them in the Python shell.

To create a new Python program, click on **File | New File** (or press **Ctrl+N**) in the Python shell. This brings up a new window (an editor that is a part of IDLE) in which we can type our program. Type the following program into this new window:

```
a = input("Enter a number: ")
b = input("Enter another number: ")
print "{} raised to the power {} is {}".format(a, b, a ** b)
```

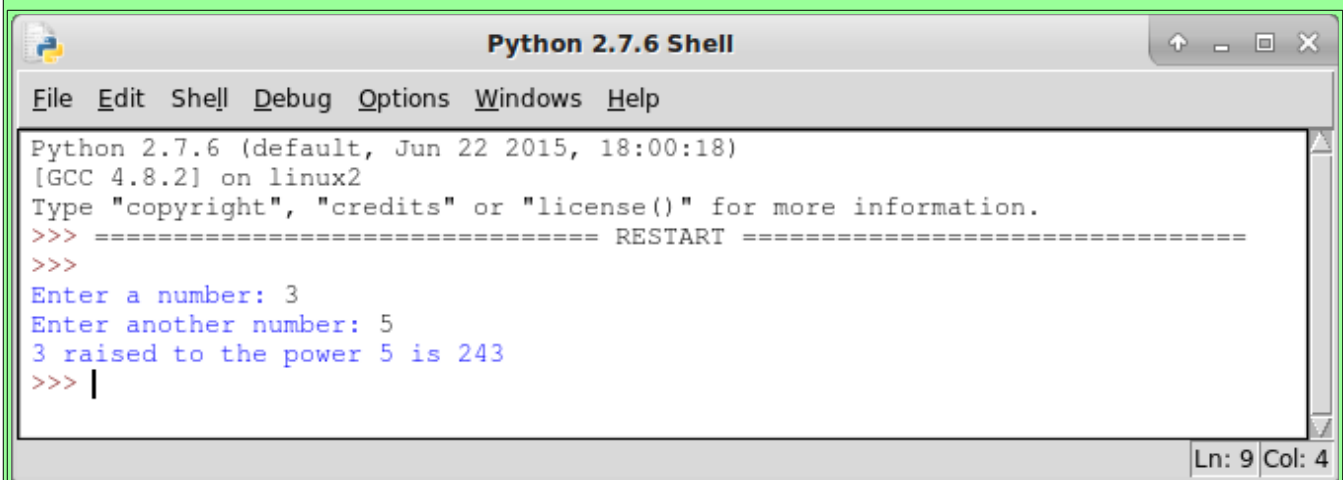


This is what you should see at this point:



```
Python 2.7.6: 1.py - /home/jgourd/1.py
File Edit Format Run Options Windows Help
a = input("Enter a number: ")
b = input("Enter another number: ")
print "{} raised to the power {} is {}".format(a, b, a ** b)
Ln: 4 Col: 0
```

Before we can run this program, it must be saved. Do so by clicking on **File | Save** (or press **Ctrl+S**). Give it an appropriate name, and save it to an appropriate location. Now it can be executed by clicking on **Run | Run Module** (or by pressing **F5**). This executes the program in the Python shell:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter a number: 3
Enter another number: 5
3 raised to the power 5 is 243
>>> |
Ln: 9 Col: 4
```

Provide values for the two requested numbers (3 and 5 were provided in the example above).

Lastly, to exit IDLE, click on **File | Exit** (or press **Ctrl+Q**).

### Reloading a saved file

To load a saved Python program, simply double-click on the saved file. This should bring up the IDLE editor with your file loaded in it. Sometimes, double-clicking on the file just opens it up in a notepad-like editor by default. To force it to open in IDLE, right-click the file instead, and select **Open with IDLE**. This should load it in the IDLE editor. The program can then be executed as before, by clicking on **Run | Run Module** (or by pressing **F5**). This will automatically open a Python shell and execute the program.

### Your turn

Write a Python program that prompts the user for two numbers (let's use 14 and 3 for this example) and subsequently displays the following string:

The quotient of 14 divided by 3 is 4 with a remainder of 2.

Since you are familiar with writing programs in Scratch, let's use that familiarity to compare the various programming constructs and see how they differ syntactically in Python.

### Data types, constants, and variables

As mentioned in a previous lesson, the kinds of values that can be expressed in a programming language are known as its **data types**. Recall that Scratch supports only two data types: text and numbers. Since Python is a general purpose programming language, it supports many more data types. Actually, it can support virtually any type that you can think of! That is, Python allows you to define your own type for use in whatever way you wish. Since this is user-defined, let's focus on what are called primitive types for now. The **primitive types** of a programming language are those data types that are built-in (or standard) to the language and typically considered as basic building blocks (i.e., more complex types can be created from these primitive types).

Python's standard types can be grouped into several classes: numeric types, sequences, sets, and mappings. Although there are actually others, we will focus on these for now.

Numeric types include whole numbers, floating point numbers, and complex numbers. Python has two whole number types: `int` and `float`. The `int` data type is a 64-bit integer (in Python 2.x) or an integer of unlimited length (in Python 3.x). Actually, an integer of unlimited length also exists (only) in Python 2.x as the `long` data type. These integer types can represent negative or positive whole numbers. The `float` type is a 64-bit floating point (decimal) number. Lastly, the `complex` type represents complex numbers (i.e., numbers with real and imaginary parts). Most of our programs will require only `int` and `float`.

So what does this all mean? We create variables that contain data of some data type. Knowing the data type of a variable is like knowing the superpowers of a person you can control. In this analogy, the superpowers of a data type are the methods and properties that can be leveraged for use in whatever program you are writing at the time. For example, one of the superpowers of the numeric data types is raising them to a power. To do that, we can use the function of the form `pow(x, y)`. In this example, `x` and `y` are variables that are of type `int` or `float`. The **pow** function returns the value of the computation involving raising the value in `x` to the power of the value in `y` (i.e.,  $x^y$ ). This function would not typically be able to work for variables that aren't numeric data types. You may recall that the same functionality can be implemented in Python as: `x ** y`. This effectively performs the same thing.

A **constant** is defined as a value of a particular type that does not change over time. In Python (just as in Scratch), both numbers and text may be expressed as constants. **Numeric constants** are composed of the digits 0 through 9 and, optionally, a negative sign (for negative numbers), and a decimal point (for floating point numbers). For example, the number -3.14159 is a numeric constant in Python.

Just as in Scratch, a **text constant** consists of a sequence of characters (also known as a string of characters – or just a **string**). The following are examples of valid string constants:

“A man, a plan, a canal, Panama.”

“Was it Eliot's toilet I saw?”

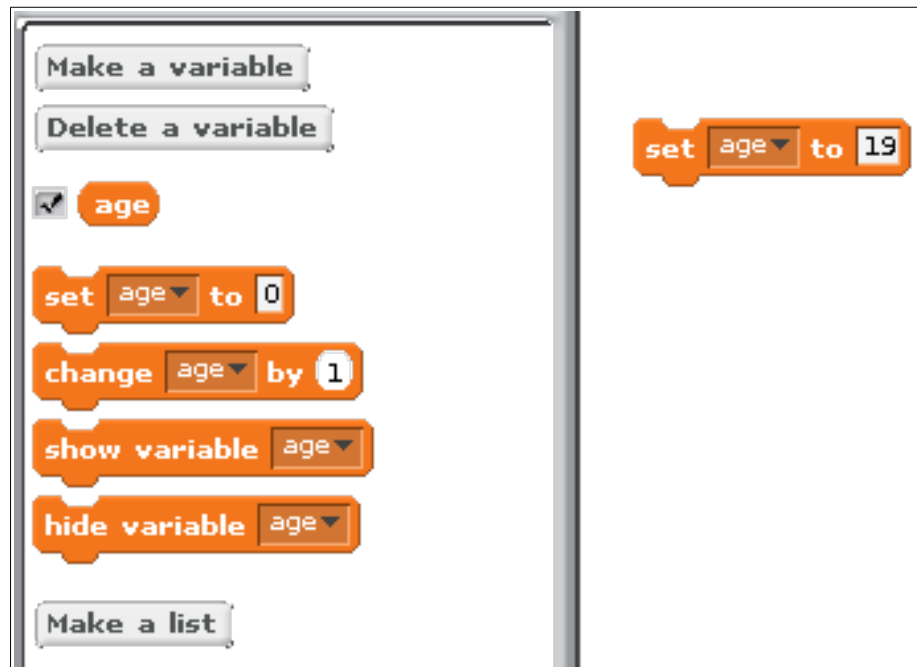
“There are 10 kinds of people in this world. Those who know binary, those who don't, and those who didn't know it was in base 3!”

Note that, unlike Scratch, Python requires the quotes surrounding text constants.

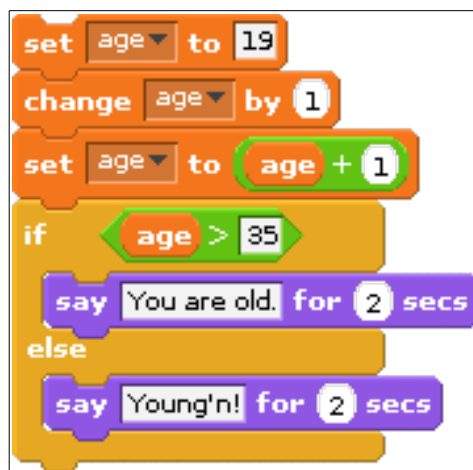
Recall that a **variable** is defined to be a named object that can store a value of a particular type. Before a variable can be used, its name must be declared. Here is an example of declaring and initializing a variable in Python:

```
age = 19
```

In Scratch, the variable had to first be declared in the *variables* blocks group. A **set var to n** block was then used to initialize the variable:



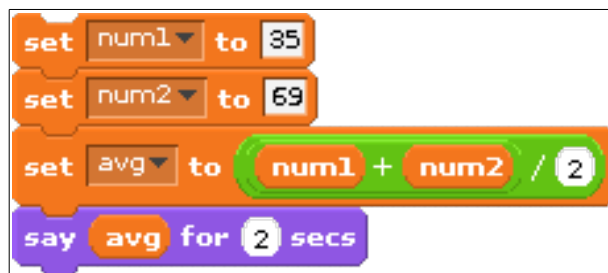
Here are some examples that deal with variables and how they compare in Scratch and Python:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> age = 19
>>> age = age + 1
>>> age = age + 1
>>> if age > 35:
    print "You are old."
else:
    print "Young'n!"

Young'n!
>>> age
21
>>> |
```

Another example:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> num1 = 35
>>> num2 = 69
>>> avg = (num1 + num2) / 2.0
>>> print avg
52.0
>>> num1
35
>>> num2
69
>>> avg
52.0
>>> |
```

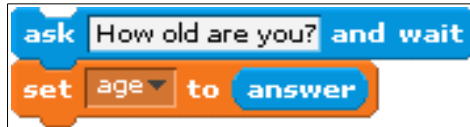
In short, to declare variables in Python, we simply write a statement that assigns a value to a variable. Note that, just as in Scratch, we can assign a value of a different type to a variable. For example:

```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> var = 5
>>> var
5
>>> var = 3.14159
>>> var
3.14159
>>> var = "Pi"
>>> var
'Pi'
>>> |
```

## Input and output

As illustrated earlier, input and output statements in Python are relatively straightforward (and pretty much reflect their implementations in Scratch). The output statement in Scratch is implemented through a **say** block. As you know, the output is then reflected on the stage (usually through a text bubble above a sprite). In Python, output is implemented as a print statement: `print "This is some output!"`

In Scratch, input is obtained through the **ask ... and wait** block. Subsequently, the answer to the question asked can be assigned to a variable using the **set n to answer** block:



In Python, we use the **input** statement to ask a question and obtain user input. In the same statement, we can assign the result of this to a variable:

```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> age = input("How old are you? ")
How old are you? 41
>>> age
41
>>>
```

Of course, we need to take care to properly specify whether the input is numeric or text (i.e., with quotes).

## Expressions and assignment

You've seen how to assign values to variables above using a simple assignment statement. For example:

```
name = "Shonda Lear"
age = 19
grade = 91.76
letter_grade = "A"
```

These are all examples of assignment statements. In this configuration, the equal sign (=) functions as the assignment operator. Later, you will see how it can also be used to compare values or expressions.

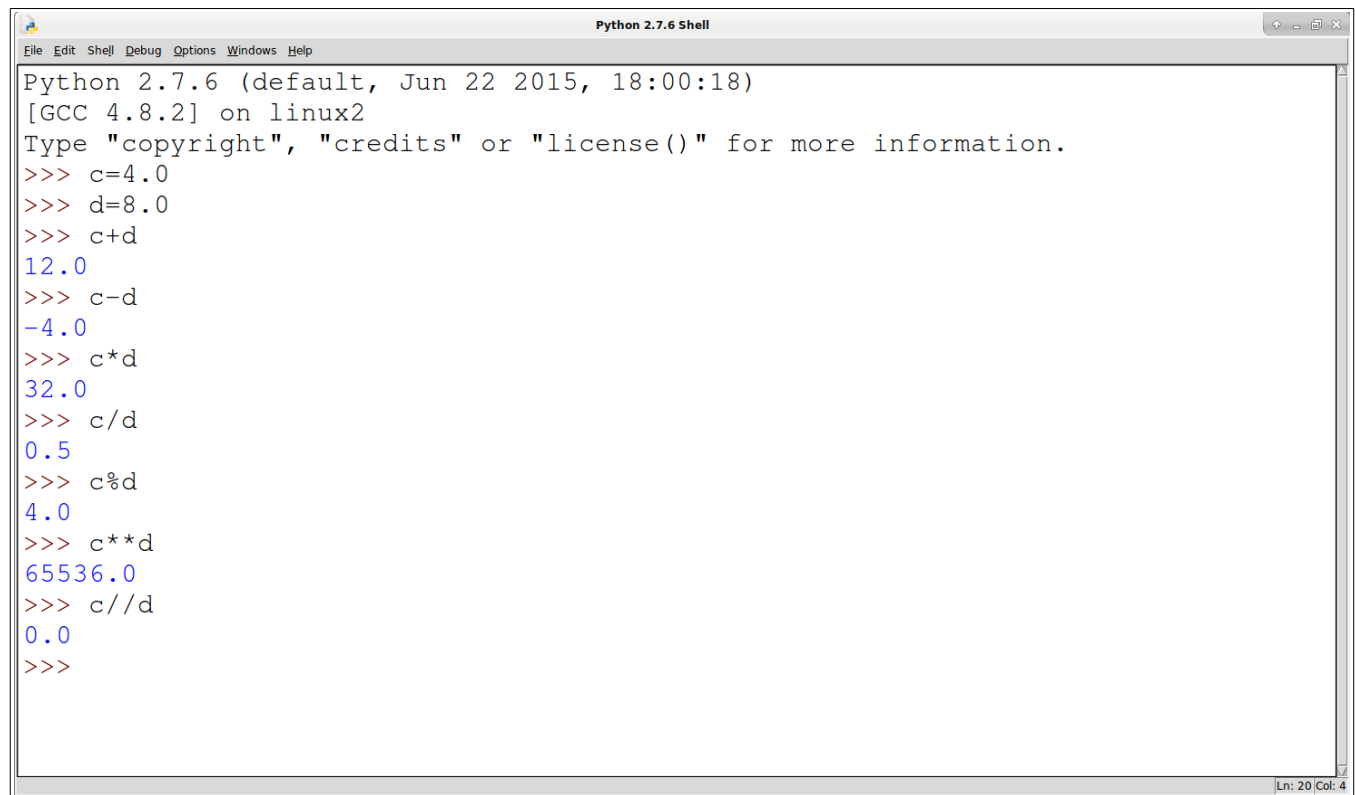
An **expression** in a programming language is some combination of values (e.g., constants and variables) that are evaluated to produce some new value. For example, a simple expression in Python is `1 + 2`. The result of this expression is, of course, 3! Expressions usually take on the form of *operand operator operand*. In the previous example, the operator was + and the operands were 1 and 2. The operator +

has a very well defined behavior on operands of numeric types: it simply adds them. On string types, it concatenates.

Like Scratch, Python has a variety of operators, broken down into several classes: arithmetic operators, relational (comparison) operators, assignment operators, logical operators, bitwise operators, membership operators, and identity operators. Let's first take a look at the arithmetic operators since they relate directly to assignment. The **arithmetic operators** allow us to perform arithmetic operations on two operands. In the following table, assume that  $a = 23$ ,  $b = 17$ ,  $c = 4.0$ , and  $d = 8.0$ :

Python Arithmetic Operators and Examples			
+	addition	$a + b = 40$	$c + d = 12.0$
-	subtraction	$a - b = 6$	$c - d = -4.0$
*	multiplication	$a * b = 391$	$c * d = 32.0$
/	division	$a / b = 1$	$c / d = 0.5$
%	modulus	$a \% b = 6$	$c \% d = 4.0$
**	exponentiation	$a ** b = 141050039560662968926103L$	$c ** d = 65536.0$
//	floor division	$a // b = 1$	$c // d = 0.0$

Note the L at the end of the result of the expression  $a ** b$ . In Python 2.x, 64-bit integers are of type `int`, and unlimited length integers are of type `long`. An L at the end of a number implies that it is of the `long` type. Here is output of the examples in the previous table using the variables `c` and `d` in IDLE:

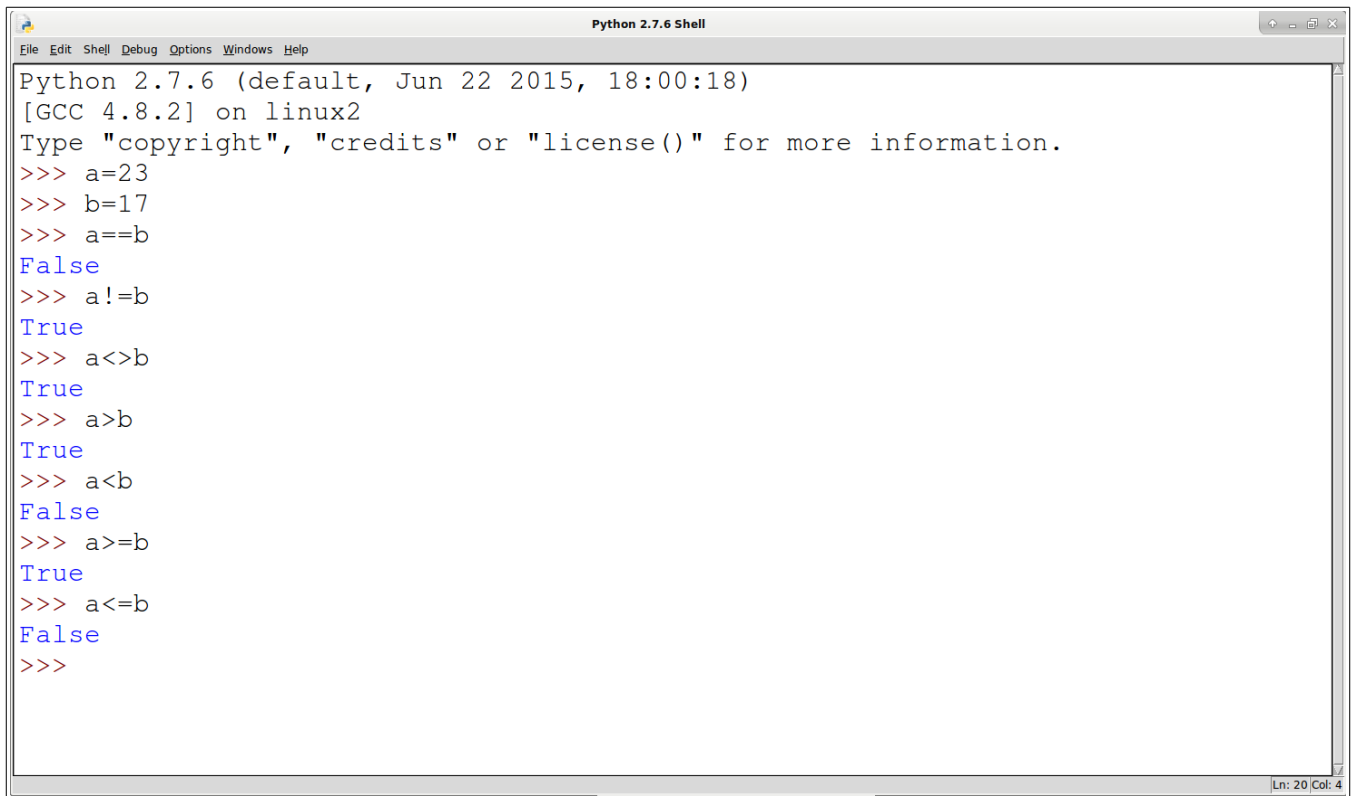


```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> c=4.0
>>> d=8.0
>>> c+d
12.0
>>> c-d
-4.0
>>> c*d
32.0
>>> c/d
0.5
>>> c%d
4.0
>>> c**d
65536.0
>>> c//d
0.0
>>>
```

The **relational operators** allow us to compare the values of two operands. The result is the relation among the operands. In the following table, assume that  $a = 23$  and  $b = 17$ :

Python Relational Operators and Examples		
<code>==</code>	equality	<code>a == b</code> is False
<code>!=</code>	inequality	<code>a != b</code> is True
<code>&lt;&gt;</code>	inequality	<code>a &lt;&gt; b</code> is True
<code>&gt;</code>	greater than	<code>a &gt; b</code> is True
<code>&lt;</code>	less than	<code>a &lt; b</code> is False
<code>&gt;=</code>	greater than or equal to	<code>a &gt;= b</code> is True
<code>&lt;=</code>	less than or equal to	<code>a &lt;= b</code> is False

Note that the capitalization of `True` and `False` is intentional. In Python, the boolean value *true* is expressed as `True` and *false* as `False`. Here is output of the examples in the previous table in IDLE:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a=23
>>> b=17
>>> a==b
False
>>> a!=b
True
>>> a<>b
True
>>> a>b
True
>>> a<b
False
>>> a>=b
True
>>> a<=b
False
>>>
```

In Python, relational operators are typically used in if-statements, where branching is often desired. This will be illustrated in more detail later.



The **assignment operators** allow us to assign values to variables. You have already seen the most basic example of this using the equal assignment operator (as in the statement: `age = 19`). In the following table, assume that `a = 23.0` and `b = 17`:

Python Assignment Operators and Examples		
<code>=</code>	<code>a = b</code> assigns <code>b</code> to <code>a</code>	<code>a = 17</code>
<code>+=</code>	<code>a += b</code> increments <code>a</code> by <code>b</code> (same as <code>a = a + b</code> )	<code>a = 40.0</code>
<code>-=</code>	<code>a -= b</code> decrements <code>a</code> by <code>b</code> (same as <code>a = a - b</code> )	<code>a = 6.0</code>
<code>*=</code>	<code>a *= b</code> multiplies <code>a</code> by <code>b</code> and stores the result in <code>a</code> (same as <code>a = a * b</code> )	<code>a = 391.0</code>
<code>/=</code>	<code>a /= b</code> divides <code>a</code> by <code>b</code> and stores the result in <code>a</code> (same as <code>a = a / b</code> )	<code>a = 1.3529411764705883</code>
<code>%=</code>	<code>a %= b</code> divides <code>a</code> by <code>b</code> and stores the remainder in <code>a</code> (same as <code>a = a % b</code> )	<code>a = 6.0</code>
<code>**=</code>	<code>a **= b</code> raises <code>a</code> to the power <code>b</code> and stores the result in <code>a</code> (same as <code>a = a ** b</code> )	<code>a = 1.4105003956066297e+23</code>
<code>//=</code>	<code>a //= b</code> divides <code>a</code> by <code>b</code> and stores the floor of the result to <code>a</code> (same as <code>a = a // b</code> )	<code>a = 1.0</code>

Here is output of the examples in the previous table in IDLE:

```

Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a=23.0
>>> b=17
>>> a=b
>>> a
17
>>> a=23.0
>>> a+=b
>>> a
40.0
>>> a=23.0
>>> a-=b
>>> a
6.0
>>> a=23.0
>>> a*=b
>>> a
391.0
>>> ,

```

```

Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
>>> a=23.0
>>> a/=b
>>> a
1.3529411764705883
>>> a=23.0
>>> a%=b
>>> a
6.0
>>> a=23.0
>>> a**=b
>>> a
1.4105003956066297e+23
>>> a=23.0
>>> a//=b
>>> a
1.0
>>> |

```

The **bitwise operators** work on bits and perform bit-by-bit operations. Think back to binary addition or to the primitive logic gates. Each of these concepts operated on bits and produced bits. In the following table, assume that a = 60 (or 00111100 in binary) and b = 13 (or 00001101 in binary):

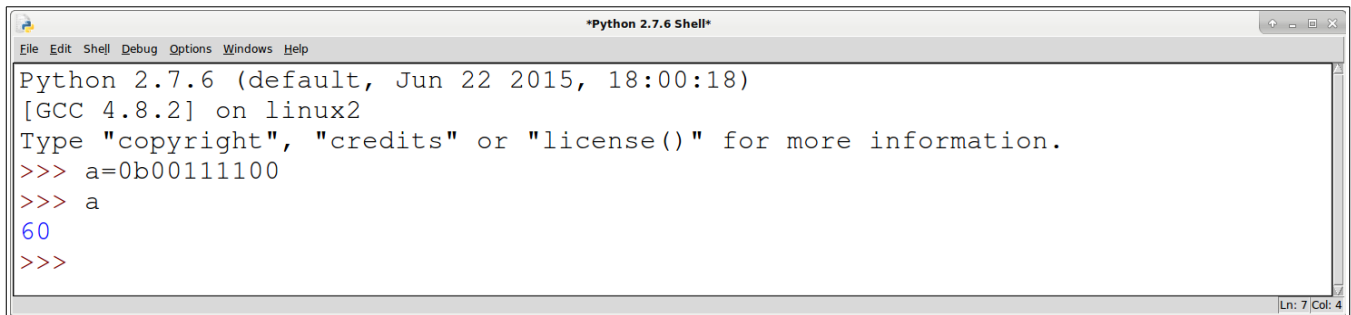
Python Bitwise Operators and Examples		
&	bitwise <i>and</i>	a & b = 00001100 (or 12 in decimal)
	bitwise <i>or</i>	a   b = 00111101 (or 61 in decimal)
^	bitwise <i>xor</i>	a ^ b = 00110001 (or 49 in decimal)
~	bitwise <i>not</i>	~a = 11000011 (or -61 in decimal; we will explain this one later)
<<	left shift	a << 2 = 11110000 (or 240 in decimal)
>>	right shift	a >> 2 = 1111 (or 15 in decimal)

The bitwise not has the effect of inverting the bits. Why 11000011 in binary is equal to -61 in decimal will be explained in a later lesson. Here is output of the examples in the previous table in IDLE:

```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a=60
>>> b=13
>>> bin(a)
'0b111100'
>>> bin(b)
'0b1101'
>>> a&b
12
>>> bin(a&b)
'0b1100'
>>> a|b
61
>>> bin(a|b)
'0b111101'
>>> a^b
49
>>> bin(a^b)
'0b110001'
>>> ~a
-61
Ln: 26 Col: 4
```

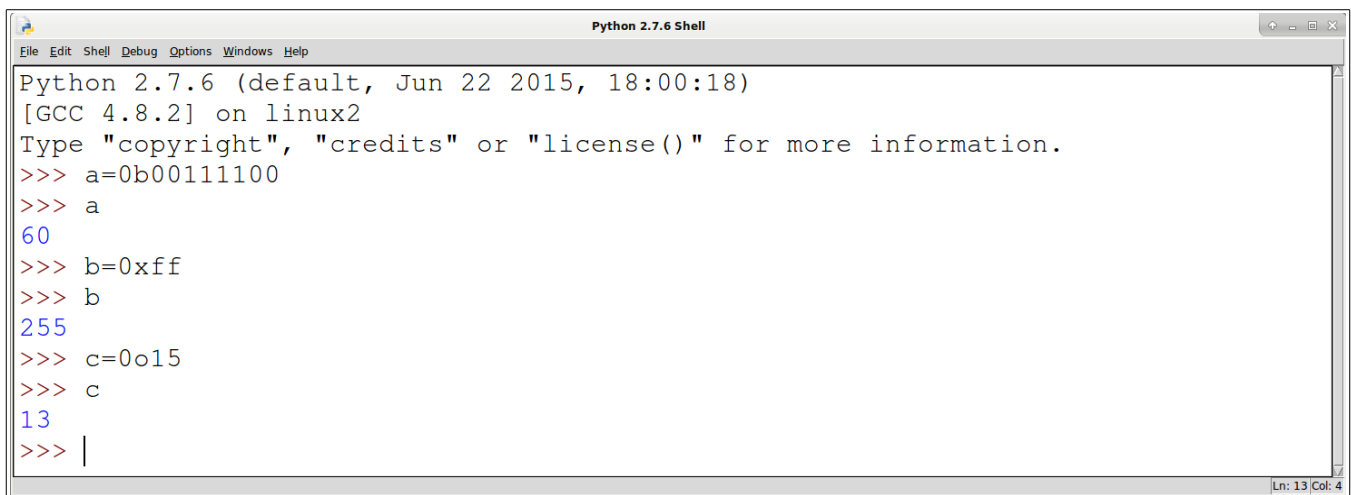
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a=60
>>> b=13
>>> a<<2
240
>>> bin(a<<2)
'0b11110000'
>>> a>>2
15
>>> bin(a>>2)
'0b1111'
>>> |
Ln: 14 Col: 4
```

Note the use of the **bin** function. It returns the binary representation of a value. If  $a = 60$ , the statement `bin(a)` returns `0b111100` (which is 60 in binary). The prefix `0b` implies binary. In fact, you can assign values to variables in binary form using this prefix:



```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a=0b00111100
>>> a
60
>>>
```

This can be done in other bases as well. For example, in hexadecimal (with the prefix `0x`) or in octal (with the prefix `0o`):



```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a=0b00111100
>>> a
60
>>> b=0xff
>>> b
255
>>> c=0o15
>>> c
13
>>> |
```

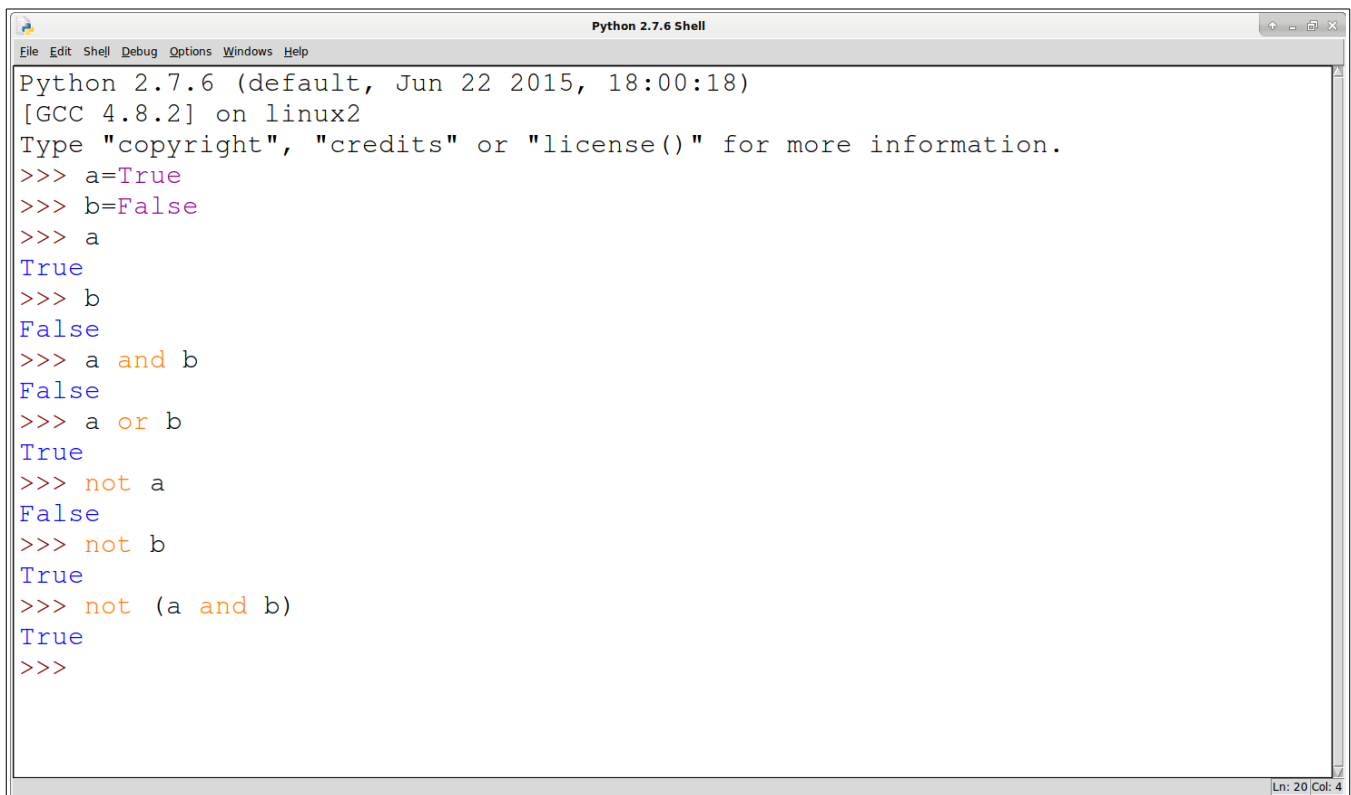
The logical operators evaluate two operands and return the logical result (i.e., True or False). Again, think back to the primitive logic gates and how they were effectively mapped to conditions in if-statements. Logical operators operate on conditions and provide the overall logical result. In the following table, assume that  $a = \text{True}$  and  $b = \text{False}$ :

Python Logical Operators and Examples		
and	logical <i>and</i>	a and b is False
or	logical <i>or</i>	a or b is True
not	logical <i>not</i>	not a is False; not b is True

Note that this is equivalent to the primitive logic gates, where 0 is substituted for False and 1 for True. Here is the truth table for the and gate shown in this manner:

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

When a is True and b is False, the result of a and b is False. Here is output of the examples in the previous table in IDLE:

A screenshot of a Python 2.7.6 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The title bar says 'Python 2.7.6 Shell'. The main text area shows the following output:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a=True
>>> b=False
>>> a
True
>>> b
False
>>> a and b
False
>>> a or b
True
>>> not a
False
>>> not b
True
>>> not (a and b)
True
>>>
```

The status bar at the bottom right shows 'Ln: 20 Col: 4'.

The logical operators do work when the inputs (i.e., *a* and *b* in the previous examples) aren't necessarily equal to True and False. That is, they also work when they are numeric values. Take, for example, the following:

```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> a=20
>>> b=10
>>> a
20
>>> b
10
>>> a and b
10
>>> a or b
20
>>> not a
False
>>> not b
False
>>> a=0
>>> not a
True
>>> ,
```

The results can be a bit confusing. This can be explained by the following table, where the variables *a* and *b* have numeric values:

Python Logical Operators and Examples		
and	logical <i>and</i>	returns a if a is false, b otherwise
or	logical <i>or</i>	returns b if a is false, a otherwise
not	logical <i>not</i>	returns True if a is True, False otherwise

Note that, in Python, 0 is False and 1 is True. However, in if-statements, any non-zero result evaluates to True. Formally, in the context of Boolean expressions, the following values are interpreted as false: False, None, numeric zero of all types, and empty strings and containers. All other values are interpreted as true. Convince yourself that the effect is indeed the same when evaluating logical conditions whether they are numeric or Boolean.

Did you know?

The *and* and *or* logical operators are **short circuit** operators. That is, to evaluate a True or False result, the minimum number of inputs required to produce such an output is evaluated. For example, suppose that *a* = False and *b* = True. The expression *a and b* is only True if both *a* and *b* are True. Since *a* is False, then there is no need to evaluate (or test) the value of *b*. This would be useless and waste CPU

cycles. Similarly, if  $a = \text{True}$  and  $b = \text{True}$ , the evaluation of the expression  $a \text{ or } b$  only requires checking that  $a$  is  $\text{True}$  for the entire expression to evaluate to  $\text{True}$  (i.e., there is no need to evaluate/test the value of  $b$ ).

### Primary control constructs

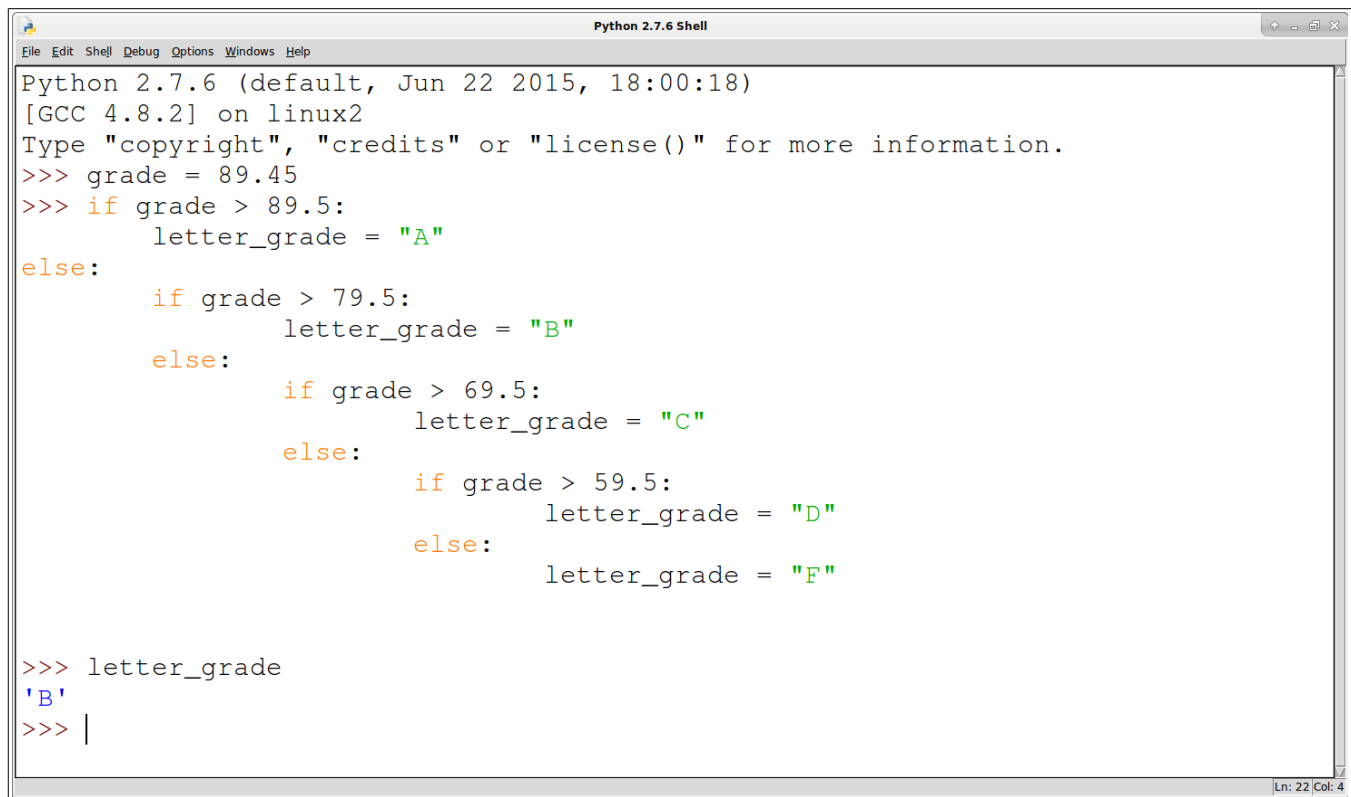
Recall the three primary control constructs: sequence, selection, and repetition. To show how they are implemented in Python, we'll start with examples in Scratch and show their Python equivalents.

Sequence implies one statement after another. In Scratch, we simply attach the blocks in the order that we wish them to be executed. In Python, the idea is the same. We simply place statements in the order that we wish them to be executed. Since you have seen this already in some of the examples above, we'll move on to selection.

Recall that **selection** constructs contain one or more blocks of statements and specify the conditions under which the blocks should be executed. Here's an example in Scratch:



First, convince yourself that the script correctly assigns a letter grade based on a numeric grade. Now here is an equivalent snippet of code in Python:

A screenshot of a Python 2.7.6 Shell window. The window title is "Python 2.7.6 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The shell output shows the Python version and GCC version, followed by a prompt to type "copyright", "credits", or "license()". The user enters a grade of 89.45. Then, an if-else statement is executed to assign a letter grade. The output shows the letter grade 'B'.

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> grade = 89.45
>>> if grade > 89.5:
    letter_grade = "A"
else:
    if grade > 79.5:
        letter_grade = "B"
    else:
        if grade > 69.5:
            letter_grade = "C"
        else:
            if grade > 59.5:
                letter_grade = "D"
            else:
                letter_grade = "F"

>>> letter_grade
'B'
>>> |
```

The structure of an if-statement in Python is:

```
if condition:
    if_body (true part)
```

The structure of an if-else statement in Python is:

```
if condition:
    if_body (true part)
else:
    else_body (false part)
```

Note in the grade/letter\_grade example above that there are a few nested if-else statements. Python provides a more elegant way to do the same thing using the **elif** clause (which stands for **else if**):

```
if grade > 89.5:
    letter_grade = "A"
elif grade > 79.5:
    letter_grade = "B"
elif grade > 69.5:
    letter_grade = "C"
elif grade > 59.5:
    letter_grade = "D"
else:
    letter_grade = "F"
```



This is indeed a bit more readable.

Note that the condition can be enclosed in parentheses (usually for readability); however, this is optional:

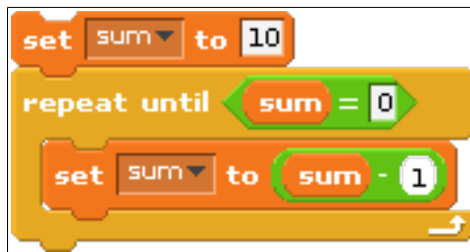
```
if (condition):  
    if_body (true part)
```

For example:

```
if (age > 40):  
    print "You are old!"
```

## Repetition

Python provides several constructs for **repetition**. The **while** loop is the most general one, and allows for both event- and count-control. Comparing this to Scratch, the while loop is similar to the **repeat-until** and **repeat-n** blocks. Here is a simple example in Scratch:



This simple script initializes a variable, *sum*, to 10. It then repeatedly decrements it by one until it is 0. This can be accomplished in Python using a while loop. The structure of a while loop in Python is:

```
while condition:  
    loop_body
```

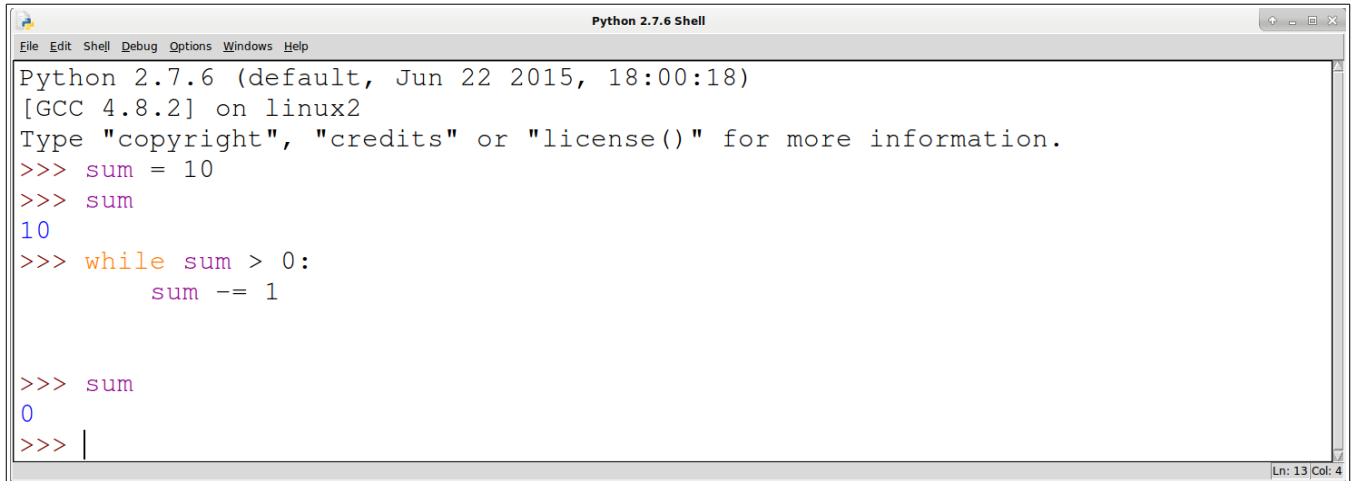
The condition may be enclosed in parentheses (as with the condition in an if-statement):

```
while (condition):  
    loop_body
```

Here is one way to accomplish the same task described in the Scratch script above in Python using a while loop:

```
sum = 10  
while sum > 0:  
    sum -= 1
```

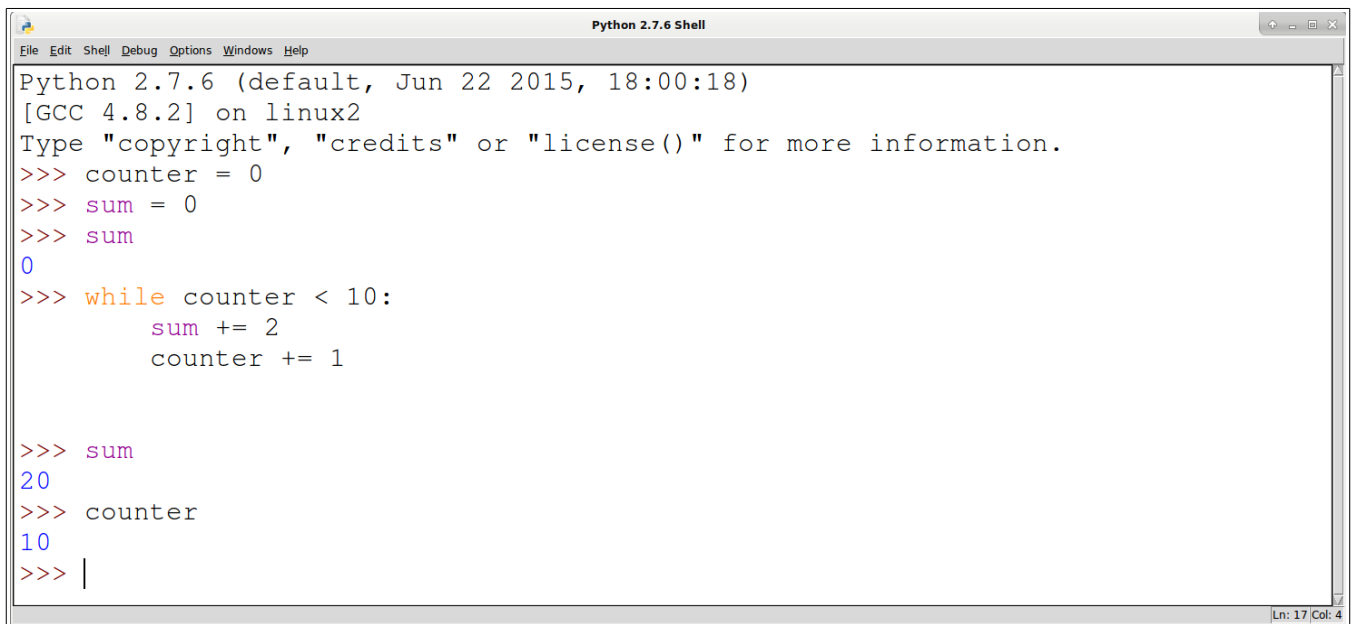
Here's this program shown in IDLE:



```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> sum = 10
>>> sum
10
>>> while sum > 0:
    sum -= 1

>>> sum
0
>>> |
```

To implement Scratch's **repeat-n** loop in Python with a while loop, we need to create a counter:



```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> counter = 0
>>> sum = 0
>>> sum
0
>>> while counter < 10:
    sum += 2
    counter += 1

>>> sum
20
>>> counter
10
>>> |
```

Python also has another repetition construct (a **for** loop) that will be covered at a later time.

## Comments

It is often useful to provide informative text in our programs. This text is not interpreted or converted to some sort of executable format as typical source code may be. It simply exists to provide information to developers, coders, or users working on or inspecting source code. This kind of text is called a **comment**. We often comment parts of programs to describe what something does, why a choice in construct was chosen, and so on. Typically, a header at the top of our programs is also inserted to provide information such as who authored the program, when it was last updated, and what it does.

In Python, there are two kinds of comments: single- and multi-line comments. Although there are several ways of commenting, we will only discuss the more widely used methods.

Single-line comments span a single line (or a part of a line). A single line comments begins with the *pound* or *hash* (for the Twitter crowd) sign: #. A single-line comments can take up the entire line (i.e., the line begins with #), or it can follow a valid Python statement (i.e., only the latter part of a line is commented). Here are sample single-line comments:

```
# get the user's name
name = input("What is your name? ")
name = "Dr. " + name      # prepend the Dr. title

# say hello!
print "Hello {}".format(name)
```

In the snippet above, there are three single-line comments. Two each take up an entire line. The third takes up only part of the line. The text that comes before it is valid Python syntax that is interpreted. Note that, once a comment has been started on a line, the rest of the line must be a comment.

Multi-line comments begin and end with three single our double quotes in succession. They are typically used in source code headers, to comment out blocks of code for reasons such as debugging, and so on. Here are sample multi-line comments:

```
"""
Author: Manny McFarlane
Last updated: 2016-03-01
Description: This program is nothing but fluff.
"""

'''
And
here
is
another
multi-line
comment!
'''

""" This is also a valid
    multi-line comment """

''' And so is this! '''
```

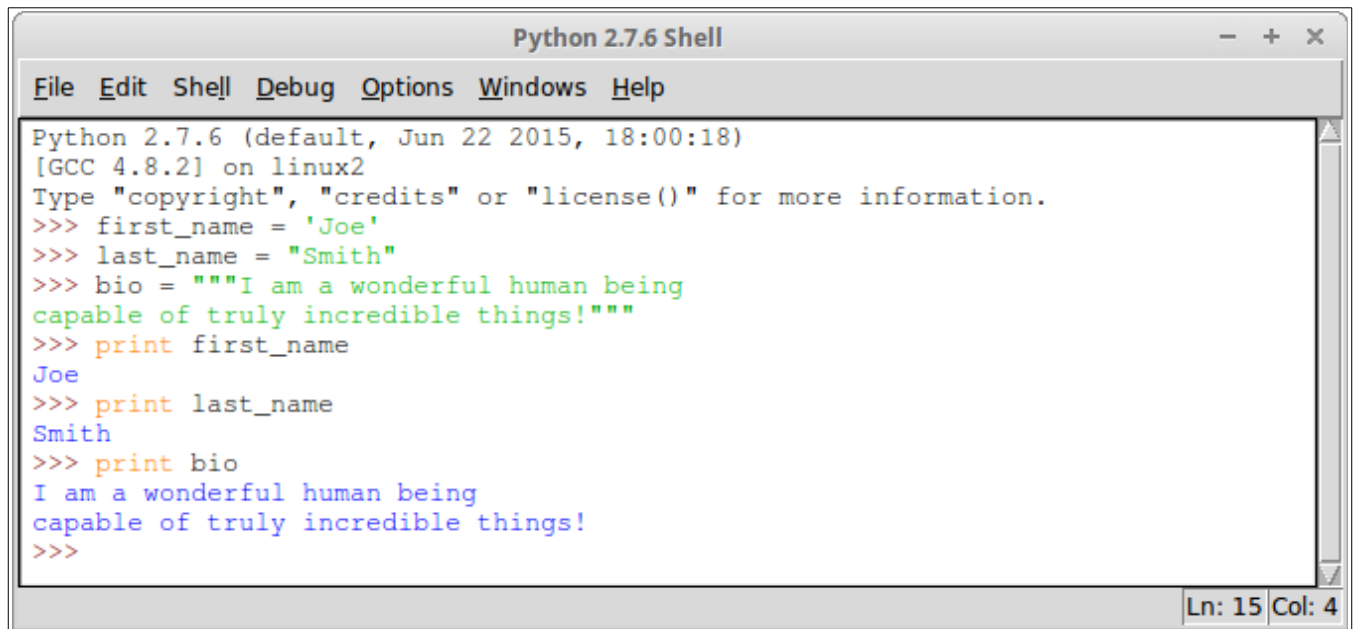
Note that single and double quotes cannot be mixed in multi-line comments. That is, a multi-line comment cannot start with three single quotes and end with three double quotes. Many Python programmers prefer to implement multi-line comments as a sequence of single-line comments; for example:

```
# Author: Manny McFarlane
# Last updated: 2016-03-01
# Description: This program is nothing but fluff.
```

This often stems from the fact that, in Python, strings can be enclosed in single quotes ('), double quotes ("), or three successive single or double quotes. The latter allows strings to span multiple lines. For example:

```
first_name = 'Joe'
last_name = "Smith"
bio = """I am a wonderful human being
capable of truly incredible things!"""
```

Here is the output of this code snippet:



```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> first_name = 'Joe'
>>> last_name = "Smith"
>>> bio = """I am a wonderful human being
capable of truly incredible things!"""
>>> print first_name
Joe
>>> print last_name
Smith
>>> print bio
I am a wonderful human being
capable of truly incredible things!
>>>
```

Specifically regarding program headers, many programmers choose to implement them as follows to make them readable and easily identifiable:

```
#####
# Author: Manny McFarlane
# Last updated: 2016-03-01
# Description: This program is nothing but fluff.
#####
```

### Identifiers and reserved words

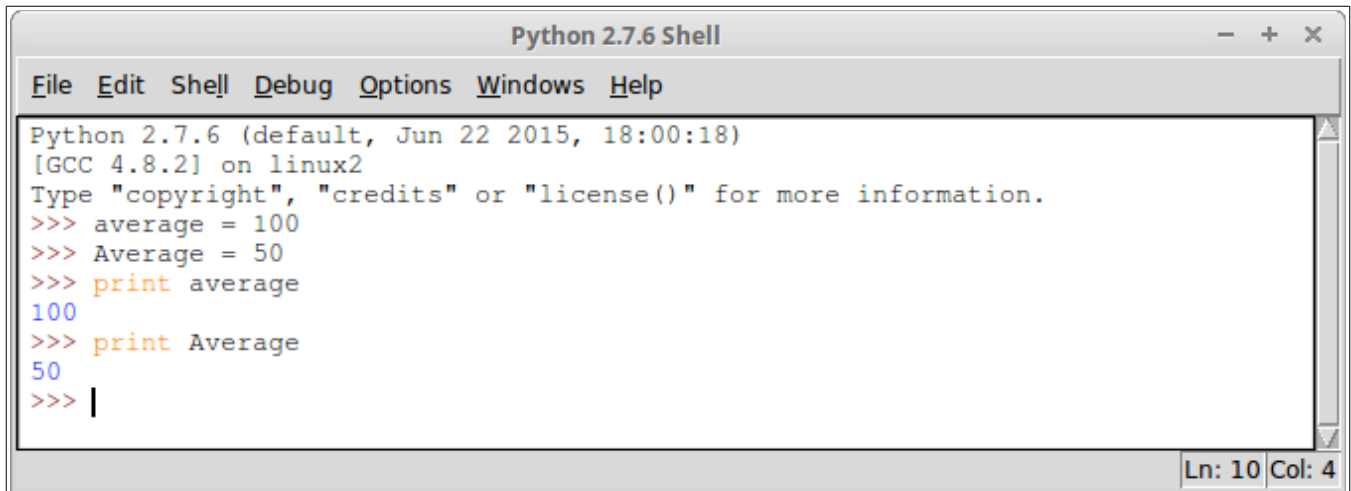
An **identifier** is a name used to identify a variable, function, or other object (that will be discussed later). Variable names (such as `age` and `average`, for example) or function names (such as `midPoint` and `distance`, for example) are all valid identifiers.

In Python, identifiers must begin with a letter (either lowercase *a* to *z* or uppercase *A* to *Z*) or an underscore (`_`) followed by zero or more letters, underscores, and digits (0 through 9). Here are examples of valid identifiers:

```
average
Average
average_grade
averageScore
```

```
_mustard
_7a69_
_1b2c3X7Y9Z0
```

Note that Python is a case-sensitive language. For example, the identifier `average` is not the same as the identifier `Average`. Take a look at this example:



The screenshot shows a terminal window titled "Python 2.7.6 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The terminal output shows the Python version (2.7.6), GCC version (4.8.2), and the operating system (linux2). It then displays the execution of several Python commands: `>>> average = 100`, `>>> Average = 50`, `>>> print average` (output: 100), and `>>> print Average` (output: 50). The cursor is at the prompt `>>> |`. The status bar at the bottom right indicates "Ln: 10 Col: 4".

**Reserved words** (sometimes called **keywords**) in a programming language are words that are meaningful to the language and cannot be used as identifiers. Most programming languages have quite a few reserved words. Python 2.7.6, for example, has the following reserved words:

<code>and</code>	<code>as</code>	<code>assert</code>	<code>break</code>	<code>class</code>
<code>continue</code>	<code>def</code>	<code>del</code>	<code>elif</code>	<code>else</code>
<code>except</code>	<code>exec</code>	<code>finally</code>	<code>for</code>	<code>from</code>
<code>global</code>	<code>if</code>	<code>import</code>	<code>in</code>	<code>is</code>
<code>lambda</code>	<code>not</code>	<code>or</code>	<code>pass</code>	<code>print</code>
<code>raise</code>	<code>return</code>	<code>try</code>	<code>while</code>	<code>with</code>
<code>yield</code>				

You are already familiar with some of these: `and`, `elif`, `else`, `if`, `not`, `or`, `print`, and `while`. Many of these reserved words will be discussed later.

### Subprograms

A **subprogram** is a block or segment of organized, reusable, and related statements that perform some action. Subprograms are useful because they allow programmers to define reusable code that can be executed repeatedly in a single program. Recall that very few *real* programs are written as one long piece of code. Instead, traditional imperative programs generally consist of large numbers of relatively simple subprograms that work together to accomplish some complex task. While it is theoretically possible to write large programs without the use of subprograms, as a practical matter any significant program must be decomposed into manageable pieces if humans are to write and maintain it.

Recall that when a subprogram is invoked, or **called**, from within a program, the *calling* part of the program pauses temporarily so that the called subprogram can carry out its actions. That is, flow of control is temporarily transferred to the subprogram. Eventually, the called subprogram will complete its task and control will once again **return** to the caller. When this occurs, the calling program resumes its execution from the point it was at when the call took place.

Recall that subprograms can call other subprograms (including copies of themselves as we observed with recursion). These subprograms can, in turn, call other subprograms. This chain of subprogram invocations can extend to an arbitrary depth as long as the *bottom* of the chain is eventually reached. It is necessary that infinite calling sequences be avoided, since each subprogram in the chain of subprogram invocations must eventually complete its task and return control to the program that called it.

Subprograms are broken down into two types: methods and functions. Generally, a **method** is a subprogram that performs an action and returns flow of control to the point at which it was called. A **function** is similar; however, it returns some sort of value before flow of control is transferred back to the point at which it was called. For example, a method may simply display some useful information about a program to the user (e.g., a program's help menu), while a function may compute some numeric value and return it to the user. Subprograms in Python are generally just referred to as functions, regardless of whether or not they return a value. For the remainder of this lesson, we will refer to subprograms as functions.

In Python, functions must formally be declared prior to their use. That is, the body or content of a function must be specified in a program before it can be called. The syntax for declaring a function is as follows:

```
def function_name(optional_parameters):  
    function_body
```

The keyword **def** is a reserved word in Python and is used to declare functions. A function name can be any valid identifier. The function name must be followed by a set of parentheses containing optional parameters. **Parameters** allow for values (constants, variables, expressions, and so on) to be passed in to a function. For example, a function may accept two values, calculate their average, and return the result to the caller. The function definition is terminated with a colon (:). The body of a function (i.e., its enclosed statements) is indented.

Here is an example of a simple function that displays a line of text:

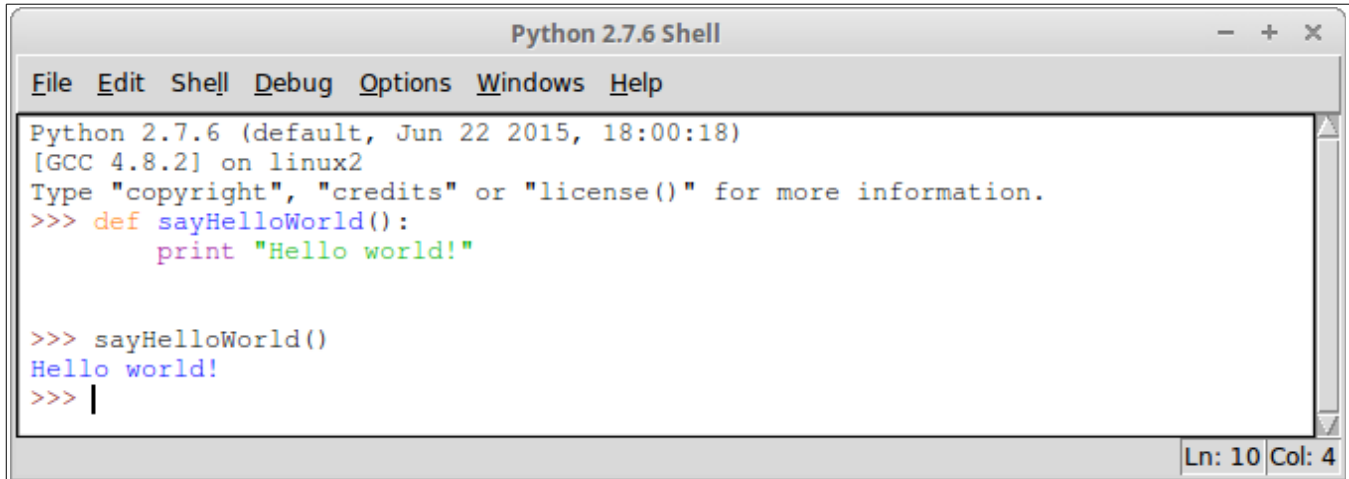
```
def sayHelloWorld():  
    print "Hello world!"
```

This function is called `sayHelloWorld` and takes no parameters. It simply displays the text, “Hello world!”

To call this function, we simply need to specify its name and the values of its parameters (if any) as follows:

```
sayHelloWorld()
```

Here is sample output of calling this simple function:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> def sayHelloWorld():
    print "Hello world!"

>>> sayHelloWorld()
Hello world!
>>> |
```

Ln: 10 Col: 4

Formally, functions have a **header** and a **body**. The header is the statement that defines the function (i.e., with the **def** keyword). The header of a function is often called its **signature**, and provides its name and any parameters. Function parameters help a function complete its task by providing input values. In fact, each call to a function possibly means a new set of parameters. Some functions compute and return a result, called the **return value**, that is returned via the **return** keyword.

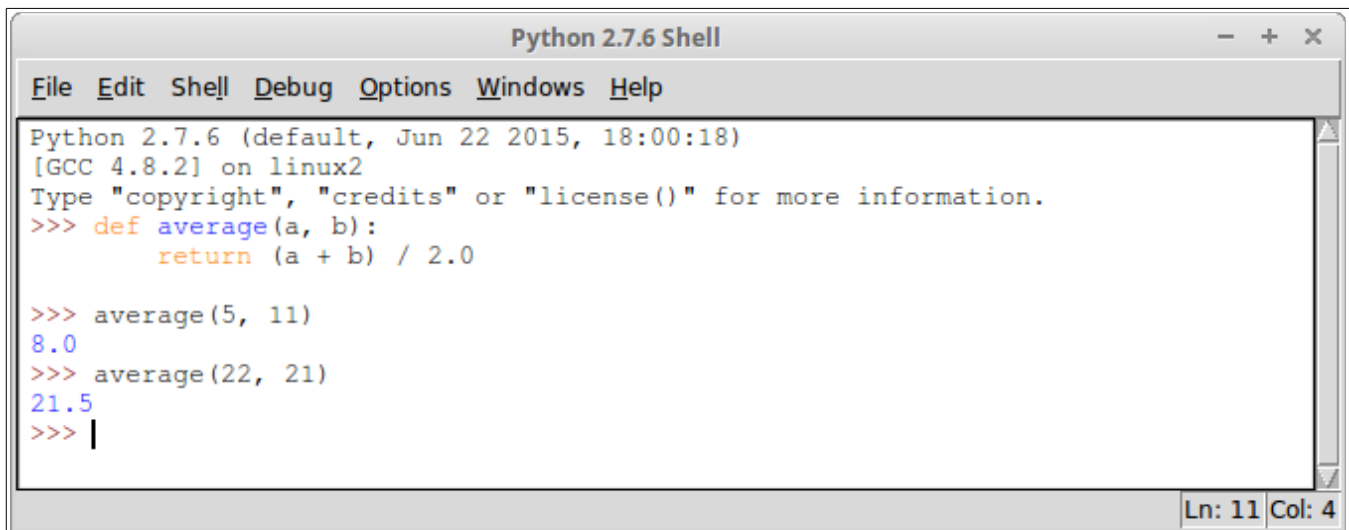
Here's a function that accepts two parameters and calculates (and returns) the average of the two:

```
def average(a, b):
    return (a + b) / 2.0
```

And here's how it could be called:

```
average(5, 11)
```

The output of this (and another example) is shown below:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> def average(a, b):
    return (a + b) / 2.0

>>> average(5, 11)
8.0
>>> average(22, 21)
21.5
>>> |
```

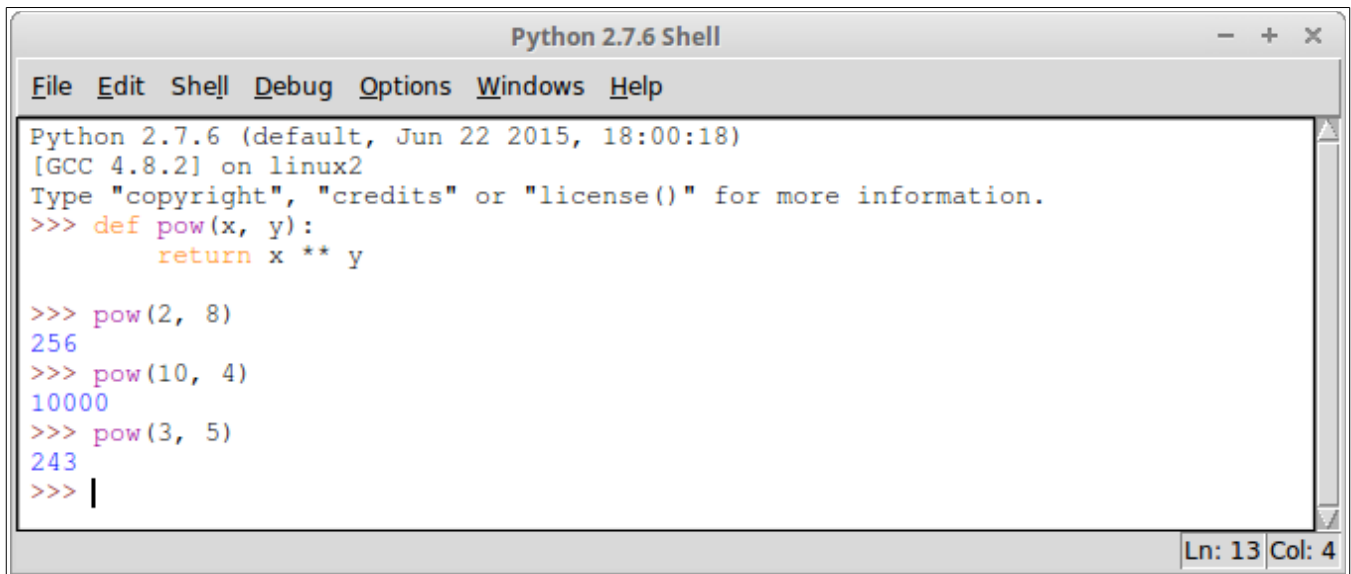
Ln: 11 Col: 4

Note the **return** keyword. Its purpose is to return whatever expression comes after it. The statement `return (a + b) / 2.0` returns the result of the expression `(a + b) / 2.0` to the caller (which happened at the statement `average(5, 11)`).

Here's a `pow` function that returns the exponentiation of one parameter by another:

```
def pow(x, y):  
    return x ** y
```

And here's sample output of this function with various parameters:



```
Python 2.7.6 Shell  
File Edit Shell Debug Options Windows Help  
Python 2.7.6 (default, Jun 22 2015, 18:00:18)  
[GCC 4.8.2] on linux2  
Type "copyright", "credits" or "license()" for more information.  
>>> def pow(x, y):  
    return x ** y  
  
>>> pow(2, 8)  
256  
>>> pow(10, 4)  
10000  
>>> pow(3, 5)  
243  
>>> |  
Ln: 13 Col: 4
```

### Formal vs actual parameters

You have seen that a function can have parameters. These parameters are formally defined when the function is defined; for example:

```
def average(a, b):  
    return (a + b) / 2.0
```

Here, the variables *a* and *b* are formally defined as parameters that must be passed in to the function `average` when it is called. In this context, the variables *a* and *b* are called **formal parameters**. It is where they are defined (in a formal manner).

Now consider a point in the source code where this function is called; for example:

```
avg = average(11, 67)
```

Here, the result of a call to the function `average` with the supplied values (or parameters) 11 and 67 is stored in the variable `avg`. These values, 11 and 67, are considered **actual parameters** in this context. That is, they are the *actual* values that will be passed in as parameters to the function `average`. In fact, they are mapped to the formally defined parameters (i.e., formal parameters) *a* and *b* in the function `average`. That function will use these values to make calculations and return the average of the two. The value returned replaces the function call. Think of this replacement as follows:

```
avg = average(11, 67)  
      39.0
```



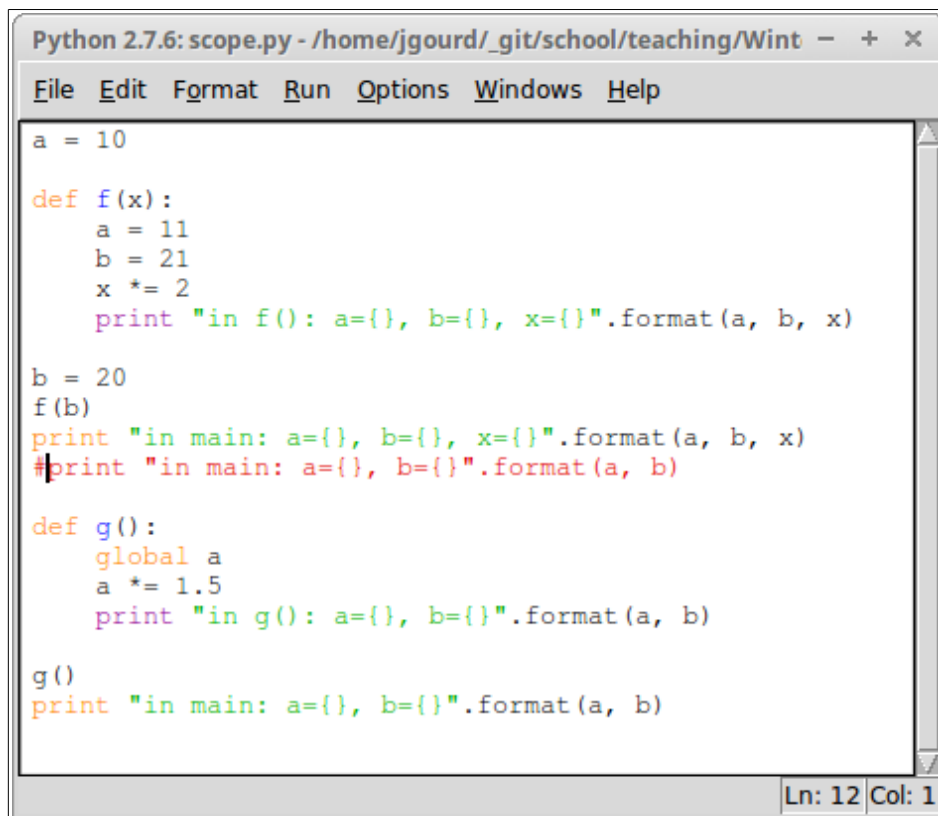
Therefore, the variable *avg* is assigned the value 39.0 after the call to the function *average* is complete. Consider this call to the same function:

```
x = 11
y = 67
avg = average(x, y)
```

Here, the result is still the same. The average of the two variables, *x* and *y* (with the values 11 and 67 respectively), is stored in the variable *avg*. Here, *x* and *y* are also actual parameters (even if they are variables themselves) because they represent the actual values supplied to the function *average*.

## Variable scope

Consider the following Python program snippet:



```
Python 2.7.6: scope.py - /home/jgourd/_git/school/teaching/Wint - + x
File Edit Format Run Options Windows Help
a = 10

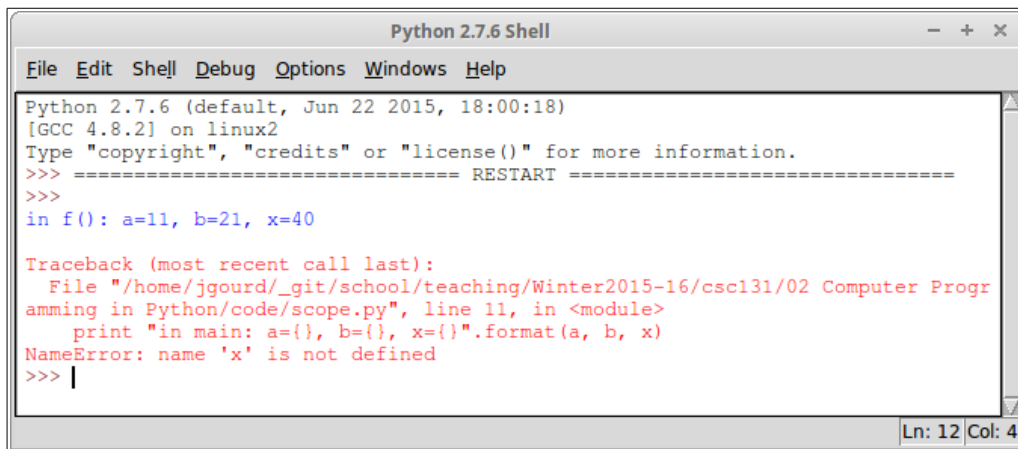
def f(x):
    a = 11
    b = 21
    x *= 2
    print "in f(): a={}, b={}, x={}".format(a, b, x)

b = 20
f(b)
print "in main: a={}, b={}, x={}".format(a, b, x)
#print "in main: a={}, b={}".format(a, b)

def g():
    global a
    a *= 1.5
    print "in g(): a={}, b={}".format(a, b)

g()
print "in main: a={}, b={}".format(a, b)
Ln: 12 Col: 1
```

The variables *a* and *b* are considered **global variables**. That is, they are accessible throughout the entire program because they are defined outside of any block context (e.g., a loop construct, a function, etc). Global variables can be accessed anywhere. Their **scope** is global (i.e., throughout the entire program). Take a look at the output of the program above:

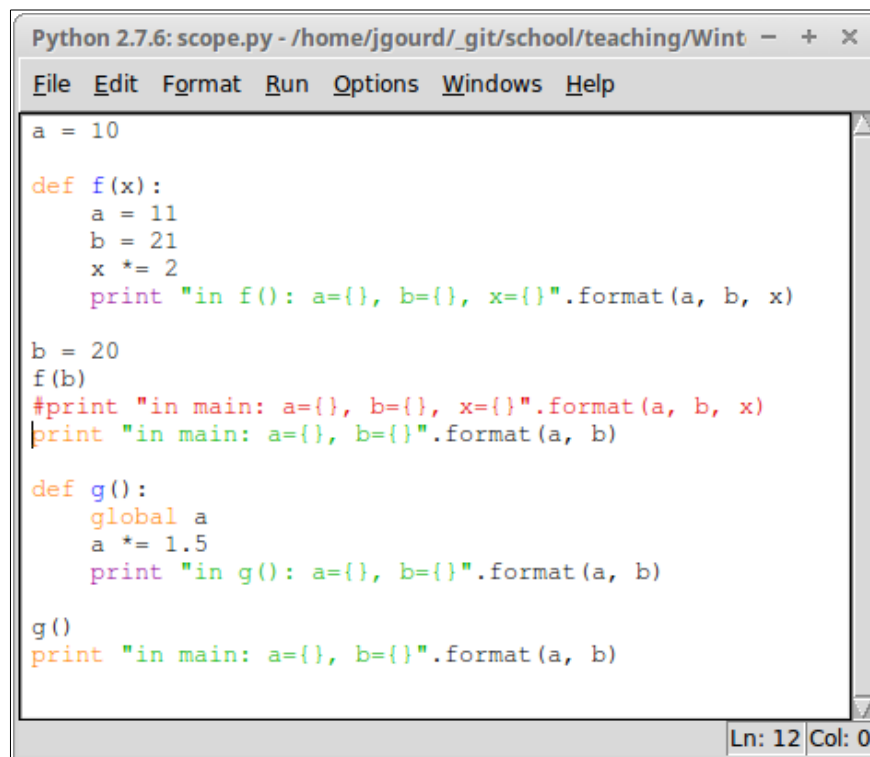


```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
in f(): a=11, b=21, x=40

Traceback (most recent call last):
  File "/home/jgourd/_git/school/teaching/Winter2015-16/csc131/02 Computer Progr
    amming in Python/code/scope.py", line 11, in <module>
      print "in main: a={}, b={}, x={}".format(a, b, x)
NameError: name 'x' is not defined
>>> |
```

Initially, the variable  $a$  is assigned the value 10. The next segment of code defines the function  $f$ . This is only a definition (i.e., the statements are not actually interpreted or executed at this point). Then, the variable  $b$  is assigned the value 20. What follows is a call to the function  $f$ , passing the variable  $b$  as an **actual parameter**. Control is then transferred to the function  $f$ , whose statements are now executed. Note that, to the function  $f$ , the variable  $x$  is the formal parameter that takes on the value passed in (from the variable  $b$ ). So the variable  $x$  is now equal to the value of the variable  $b$  that was passed in at the point of the call to  $f$ . Note that the variable  $x$  is **local** to the function  $f$ . That is, it is defined in  $f$  and only accessible in  $f$ . Once  $f$  completes and control is transferred back to the point at which function  $f$  was called, the variable  $x$  is no longer accessible! Therefore,  $x$  is considered a **local variable**. Its **scope** is valid only in the function  $f$ . And this is why the output shown above produced an error.

Let's remove the error by commenting the offending statement and replacing it as follows:



```
Python 2.7.6: scope.py - /home/jgourd/_git/school/teaching/Wint - + x
File Edit Format Run Options Windows Help
a = 10

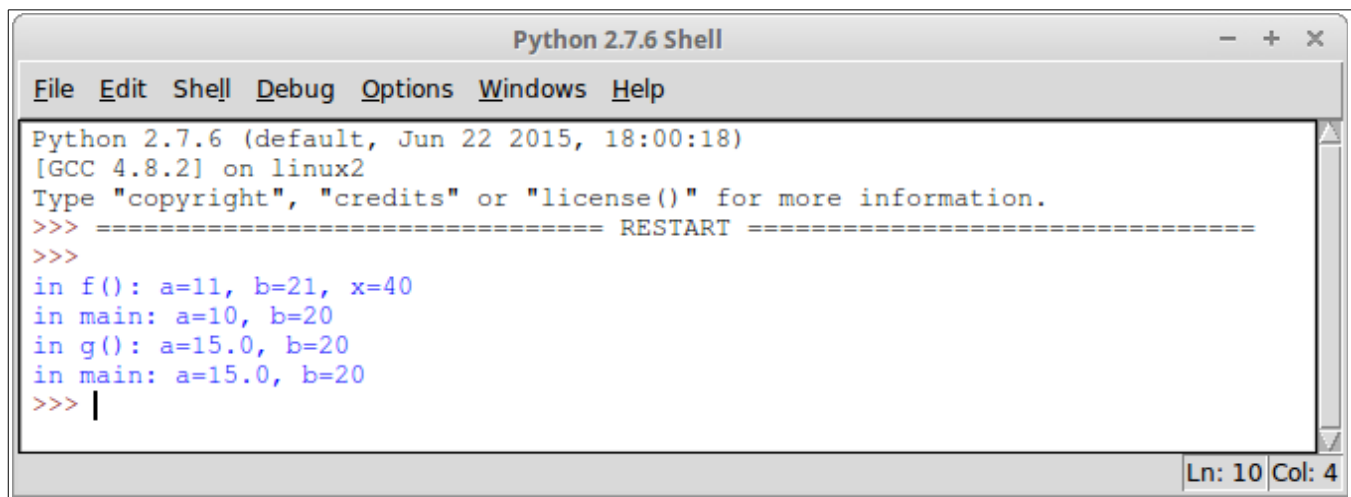
def f(x):
    a = 11
    b = 21
    x *= 2
    print "in f(): a={}, b={}, x={}".format(a, b, x)

b = 20
f(b)
#print "in main: a={}, b={}, x={}".format(a, b, x)
print "in main: a={}, b={}".format(a, b)

def g():
    global a
    a *= 1.5
    print "in g(): a={}, b={}".format(a, b)

g()
print "in main: a={}, b={}".format(a, b)
```

Now take a look at the output:

A screenshot of a Python 2.7.6 Shell window. The window has a title bar "Python 2.7.6 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following output:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
in f(): a=11, b=21, x=40
in main: a=10, b=20
in g(): a=15.0, b=20
in main: a=15.0, b=20
>>> |
```

The status bar at the bottom right shows "Ln: 10 Col: 4".

In particular, let's start with the first two lines of output:

```
in f(): a=11, b=21, x=40
in main: a=10, b=20
```

When the function `f` is called, the value of `b` (20) is passed in (i.e., mapped) to the variable `x`. Since `a` and `b` are global variables, they are also accessible in the function `f`. In `f`, all three variables are changed (`a` is changed to 11, `b` is changed to 21, and `x` is doubled). We expect the values to be 11, 21, and 40 (since `x` is originally 20 and is doubled). These results are clear.

What is not clear is the reason why, when control is transferred back to the point at which `f` was called, the variables `a` and `b` revert to their original values (10 and 20 respectively). It turns out that `a` and `b` in the function `f` only refer to the global variables in read-only statements (i.e., in any statement that does not change their values). For example, in the statement: `print a`. Once an assignment statement is executed (as in the statement: `a = 11`), Python considers the variable to be a new local variable. So this local variable `a` is only accessible in the function `f`. The global variable `a` is not changed. Now you see that we can define many variables of the same name, so long as their scope is mutually exclusive (that is, there are no scope overlaps).

Python does provide a way, however, to change the value of global variables within a function. By using the keyword **global**, it knows to consider the variable that follows it as a reference to one defined globally. This is shown in the program snippet above (in the function `g`). Here, the variable `a` is specified as global (and thus makes a reference to the global variable defined at the top of the program). When `g` changes `a` to 1.5 times its value, the change is persistent in that it makes a change to the global variable. When control is transferred back to the point at which `g` was called, the variable `a` remains changed!

```
in g(): a=15.0, b=20
in main: a=15.0, b=20
```

## Program flow

It is very important to be able to identify the flow of control in any program, particularly to understand what is going on. In Python, function definitions aren't executed in the order that they are written in the source code. Functions are only executed when they are called. This is perhaps best illustrated with an example:

```
1:  def min(a, b):
2:      if (a < b):
3:          return a
4:      else:
5:          return b

6:  def max(a, b):
7:      if (a > b):
8:          return a
9:      else:
10:         return b

11: num1 = input("Enter a number: ")
12: num2 = input("Enter another number: ")
13: print "The smaller is {}".format(min(num1, num2))
14: print "The larger is {}".format(max(num1, num2))
```

Each Python statement is numbered for reference. Lines 1 through 5 represent the definition of the function `min`. This function returns the *minimum* of two values provided as parameters. Lines 6 through 10 represent the definition of the function `max`. This function returns the *maximum* of two values provided as parameters. Lines 11 through 14 represent the main part of the program. Although the Python interpreter does *see* lines 1 through 10, those lines are not actually *executed* until the functions `min` and `max` are actually called. The first line of the program to actually be executed is line 11. In fact, here is the order of the statements executed in this program if `num1 = 34` and `num2 = 55`:

11, 12, 13, 1, 2, 3, 14, 6, 7, 9, 10

Let's explain. Line 11 asks the user to provide some value for the first number (which is stored in the variable `num1`). Line 12 asks the user to provide some value for the second number (which is stored in the variable `num2`). Line 13 displays some text; however, part of the text must be obtained by first calling the function `min`. This transfers control to line 1 (where `min` is defined). The two actual parameters, `num1` and `num2`, are then passed in and mapped to the formal parameters defined in `min`, `a` and `b`. Then, line 2 is executed and performs a comparison of the two numbers. Since `a = 34` and `b = 55`, then the condition in the `if`-statement is true. Therefore, line 3 is executed before control is transferred back to the main program with the value of the smaller number returned (and then control continues on to line 14). Note that lines 4 and 5 are never executed in this case!

Line 14 is then executed and displays some text. Again, part of the text must be obtained by first calling the function `max`. This transfers control to line 6 (where `max` is defined). The variables `a` and `b` take on the values 34 and 55 respectively. Line 7 is then executed, and the result of the comparison is false. Therefore, line 8 is not executed. Control then goes to line 9, and then to line 10 which returns the larger value. The program then ends.

What is the order of execution if  $a = 55$  and  $b = 34$ ?

What if  $a = 100$  and  $b = 100$ ?

Knowing the order in which statements are executed is crucial to debugging programs and ultimately to creating programs that work.

### Sequences

The most basic data structure in Python is the sequence. A **sequence** is composed of (related) elements. Each element in a sequence is assigned an index (or position). A sequence with  $n$  elements has indexes 0 to  $n-1$ . Python has many built-in types of sequences; however, the most popular is called the list.

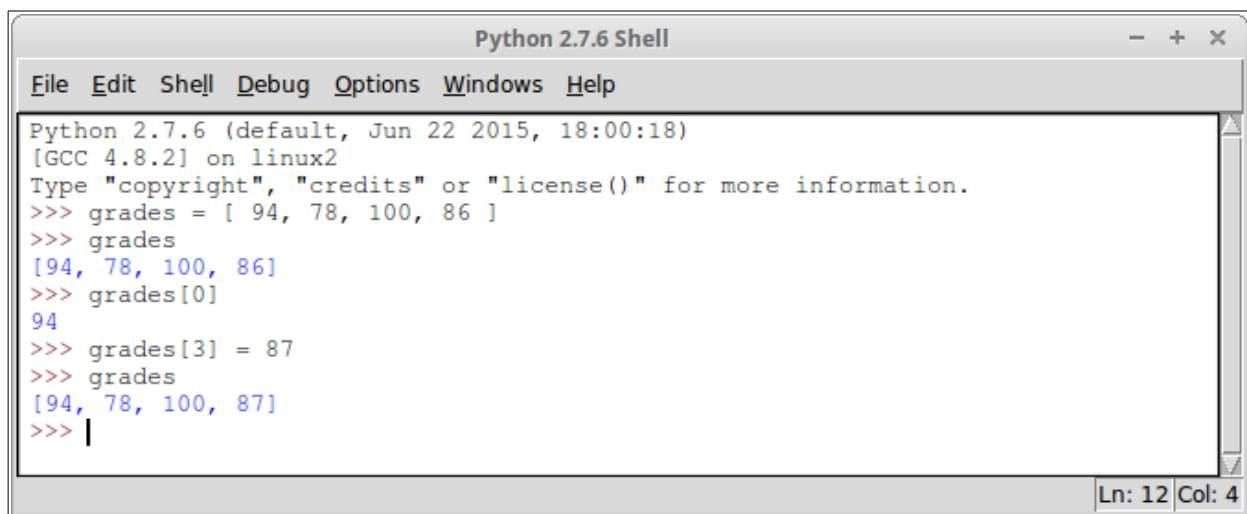
The **list** in Python is quite versatile and is declared using square brackets; for example:

```
grades = [ 94, 78, 100, 86 ]
```

The statement above declares the list `grades` with four integers: 94, 78, 100, and 86. The list can be displayed in its entirety (e.g., with the statement `print grades`); however, we can access each element individually by its index (specified within brackets). Accessing can mean to read a value in the list, or it can mean to change a value in the list; for example:

```
grades[0]
grades[3] = 87
```

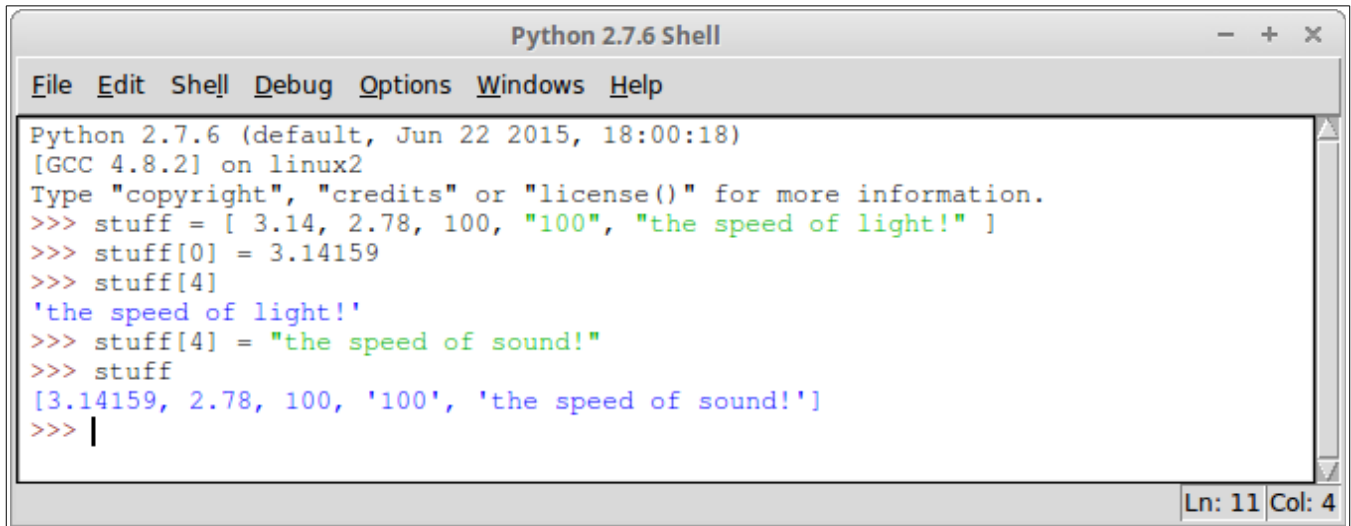
Here's an example of this:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> grades = [ 94, 78, 100, 86 ]
>>> grades
[94, 78, 100, 86]
>>> grades[0]
94
>>> grades[3] = 87
>>> grades
[94, 78, 100, 87]
>>> |
```

Note that, in Python, the values within a list do not need to be of the same data type! This is a bit different than lists in most other general purpose programming languages (usually, those languages call their lists **arrays**) in which all elements must be of the same type. Here's an example of a *heterogeneous* (meaning diverse) list in Python:

```
stuff = [ 3.14, 2.78, 100, "100", "the speed of light!" ]
```

A screenshot of a Python 2.7.6 Shell window. The window has a title bar "Python 2.7.6 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following Python code and output:

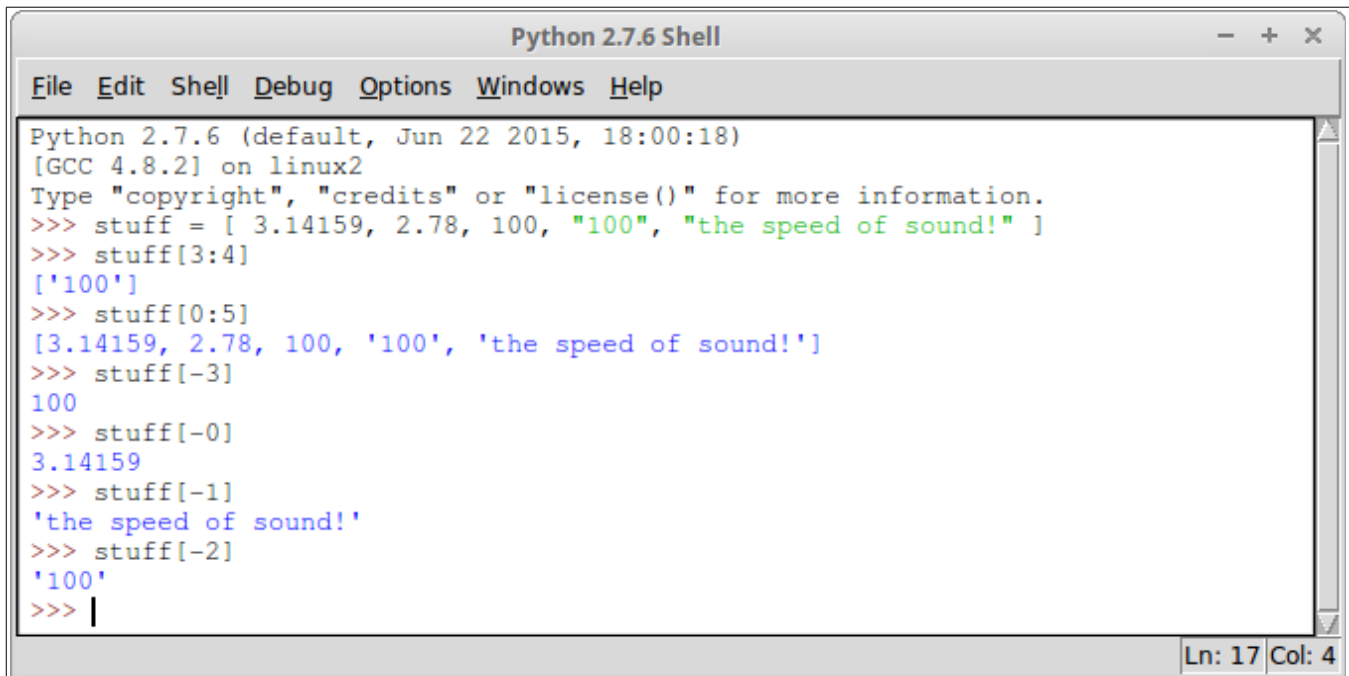
```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> stuff = [ 3.14, 2.78, 100, "100", "the speed of light!" ]
>>> stuff[0] = 3.14159
>>> stuff[4]
'the speed of light!'
>>> stuff[4] = "the speed of sound!"
>>> stuff
[3.14159, 2.78, 100, '100', 'the speed of sound!']
>>> |
```

The status bar at the bottom right shows "Ln: 11 Col: 4".

More than one value in a list can be accessed at a time. We can specify a range (or interval) of indexes in the format `[lower:upper+1]` which means the interval `[lower, upper)` (i.e., closed at lower and open at upper). That is, the lower index in the range is inclusive but the upper is not. For example:

```
stuff[3:4]      # accesses index 3 (the same as stuff[3])
stuff[0:5]      # accesses indexes 0 through 4
stuff[-3]       # accesses the third index from the right
```

Here are some examples:

A screenshot of a Python 2.7.6 Shell window. The window has a title bar 'Python 2.7.6 Shell' and a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The main text area shows the following code and output:

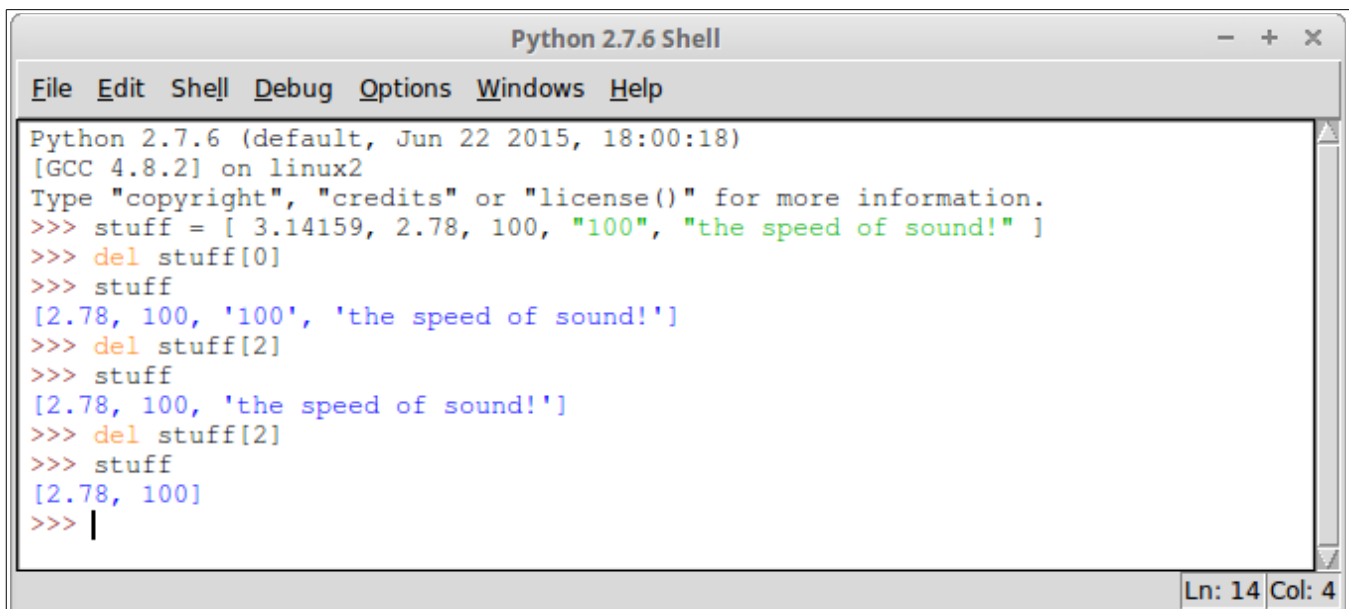
```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> stuff = [ 3.14159, 2.78, 100, "100", "the speed of sound!" ]
>>> stuff[3:4]
['100']
>>> stuff[0:5]
[3.14159, 2.78, 100, '100', 'the speed of sound!']
>>> stuff[-3]
100
>>> stuff[-0]
3.14159
>>> stuff[-1]
'the speed of sound!'
>>> stuff[-2]
'100'
>>> |
```

The status bar at the bottom right shows 'Ln: 17 Col: 4'.

Note the difference between the element 100 (a number) at index 2 and the element '100' (a string) at index 3.

List elements can be deleted with the **del** keyword as follows:

```
del stuff[2]
```

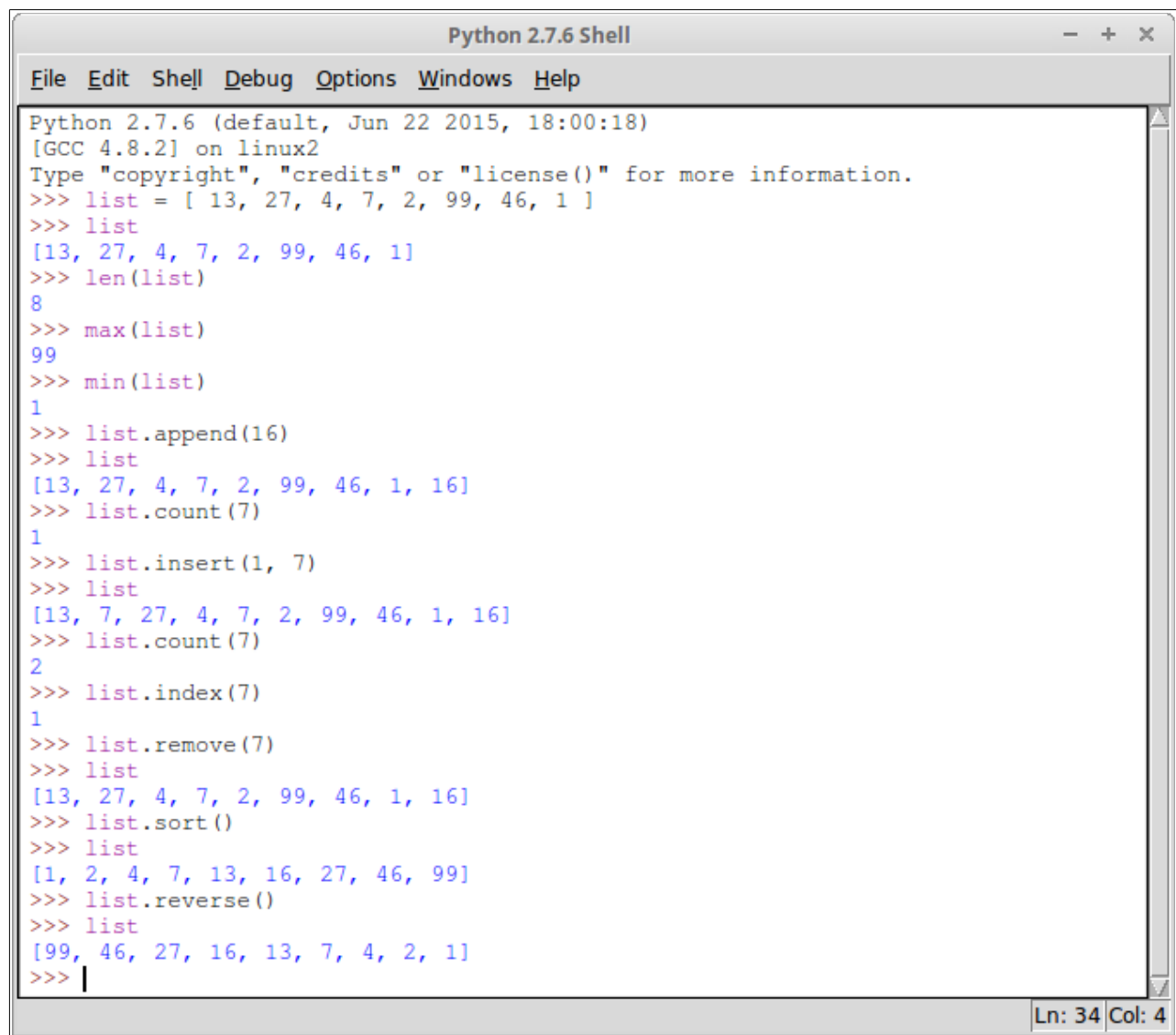
A screenshot of a Python 2.7.6 Shell window, similar to the first one. The main text area shows the following code and output:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> stuff = [ 3.14159, 2.78, 100, "100", "the speed of sound!" ]
>>> del stuff[0]
>>> stuff
[2.78, 100, '100', 'the speed of sound!']
>>> del stuff[2]
>>> stuff
[2.78, 100, 'the speed of sound!']
>>> del stuff[2]
>>> stuff
[2.78, 100]
>>> |
```

The status bar at the bottom right shows 'Ln: 14 Col: 4'.

Python provides several built-in operations that can be performed on lists. Here are many of them:

<code>len(list)</code>	Returns the length of a list
<code>max(list)</code>	Returns the item in the list with the maximum value
<code>min(list)</code>	Returns the item in the list with the minimum value
<code>list.append(item)</code>	Inserts item at the end of the list
<code>list.count(item)</code>	Returns the number of times an item appears in the list
<code>list.index(item)</code>	Returns the index of the first occurrence of item
<code>list.insert(index, item)</code>	Inserts an item at the specified index in the list
<code>list.remove(item)</code>	Removes the first occurrence of item from the list
<code>list.reverse()</code>	Reverses the items in the list
<code>list.sort()</code>	Sorts a list

A screenshot of a Python 2.7.6 Shell window. The window has a title bar "Python 2.7.6 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following code and output:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> list = [ 13, 27, 4, 7, 2, 99, 46, 1 ]
>>> list
[13, 27, 4, 7, 2, 99, 46, 1]
>>> len(list)
8
>>> max(list)
99
>>> min(list)
1
>>> list.append(16)
>>> list
[13, 27, 4, 7, 2, 99, 46, 1, 16]
>>> list.count(7)
1
>>> list.insert(1, 7)
>>> list
[13, 7, 27, 4, 7, 2, 99, 46, 1, 16]
>>> list.count(7)
2
>>> list.index(7)
1
>>> list.remove(7)
>>> list
[13, 27, 4, 7, 2, 99, 46, 1, 16]
>>> list.sort()
>>> list
[1, 2, 4, 7, 13, 16, 27, 46, 99]
>>> list.reverse()
>>> list
[99, 46, 27, 16, 13, 7, 4, 2, 1]
>>> |
```

The status bar at the bottom right shows "Ln: 34 Col: 4".



## Searching and sorting examples

In previous lessons, we designed several searching algorithms (sequential/linear search and binary search) and sorting algorithms (bubble sort, selection sort, and insertion sort). We first specified them in pseudocode, and for some we showed how they could be implemented in Scratch. To help get a better understanding of Python, let's revisit some of these and see how they could be implemented in Python.

First, let's implement a simple sequential/linear search. Recall the pseudocode for the algorithm:

```
n ← value to search for
i ← 1
found ← false
repeat
    if value of item i in the list = n
    then
        found ← true
    else
        increment i
    end
until i > the length of the list or found = true
display found
```

First, we see that the algorithm makes use of a **repeat-until** construct that Python doesn't have. We must convert it to a **while** construct instead. The following condition must be changed so that it works within a while construct:

```
repeat
    ...
until i > the length of the list or found = true
```

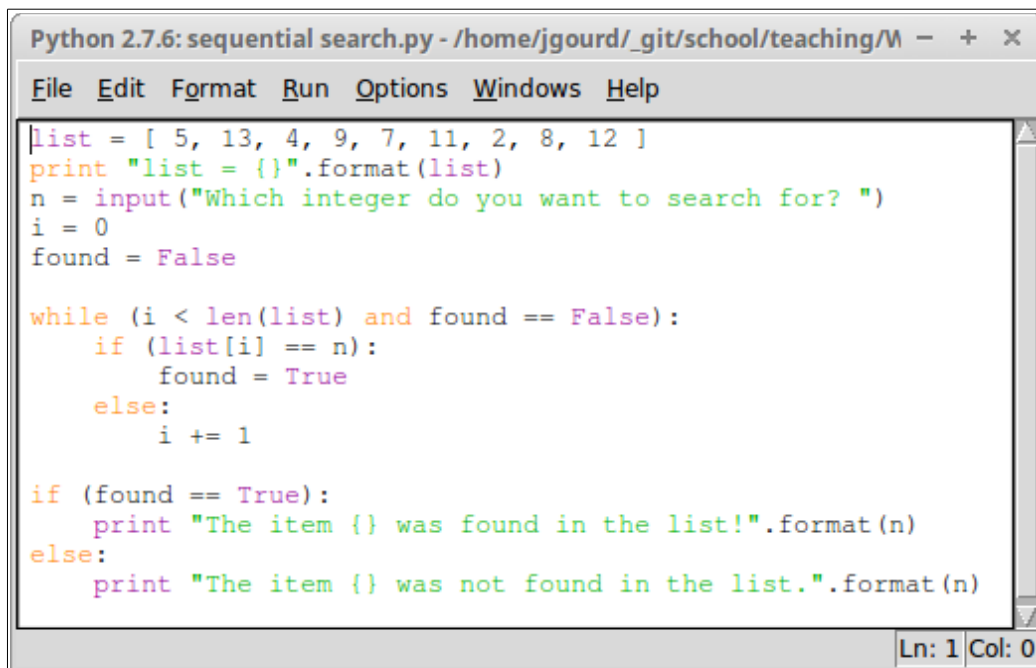
The *until* condition must be converted to an appropriate inverse *while* condition:

```
while i < the length of the list and found != true
    ...
```

Convince yourself that the conditions resulting in execution of the body of the loop are indeed the same.

In addition, we know that Python lists begin at index 0. Therefore, we will have to initialize the variable *i* to 0 to reflect this.

Let's see how this simple algorithm can be implemented in Python (with a few *frills*):

A screenshot of a Python 2.7.6 IDE window titled "Python 2.7.6: sequential search.py - /home/jgourd/\_git/school/teaching/M". The window contains a Python script for a sequential search algorithm. The code defines a list [5, 13, 4, 9, 7, 11, 2, 8, 12], prompts the user for an integer to search for, and uses a while loop to iterate through the list until the item is found or the end is reached. It then prints a message indicating whether the item was found.

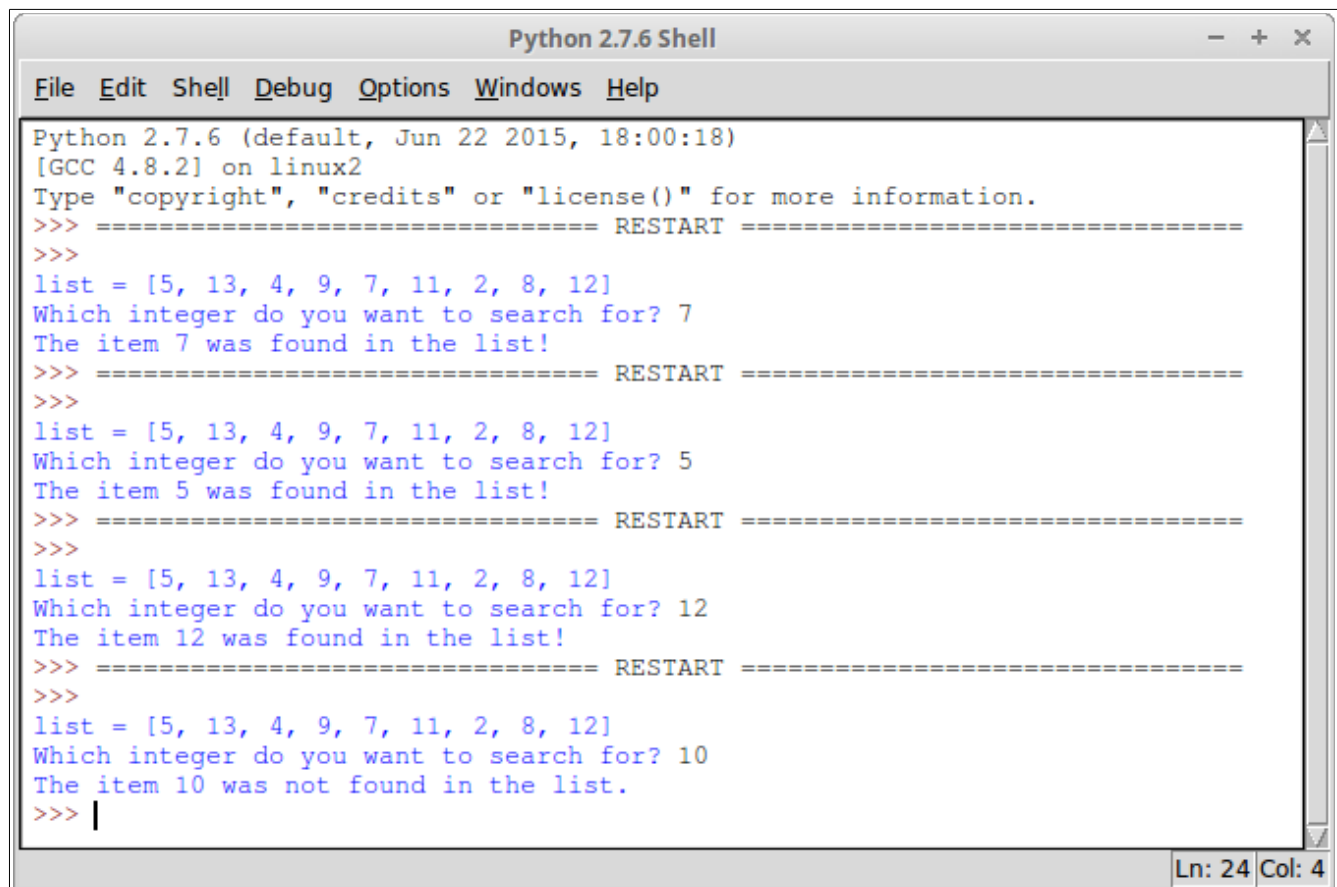
```
list = [ 5, 13, 4, 9, 7, 11, 2, 8, 12 ]
print "list = {}".format(list)
n = input("Which integer do you want to search for? ")
i = 0
found = False

while (i < len(list) and found == False):
    if (list[i] == n):
        found = True
    else:
        i += 1

if (found == True):
    print "The item {} was found in the list!".format(n)
else:
    print "The item {} was not found in the list.".format(n)
```

The status bar at the bottom right shows "Ln: 1 Col: 0".

Here's how it looks in IDLE with a few sample runs:

A screenshot of a Python 2.7.6 Shell window titled "Python 2.7.6 Shell". The window shows the execution of the sequential search algorithm with four sample runs. Each run starts with the list definition, followed by a prompt for an integer to search for, and then the result message. The runs are separated by "RESTART" lines. The first three runs find the items 7, 5, and 12, while the fourth run does not find the item 10.

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 7
The item 7 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 5
The item 5 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 12
The item 12 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 10
The item 10 was not found in the list.
>>> |
```

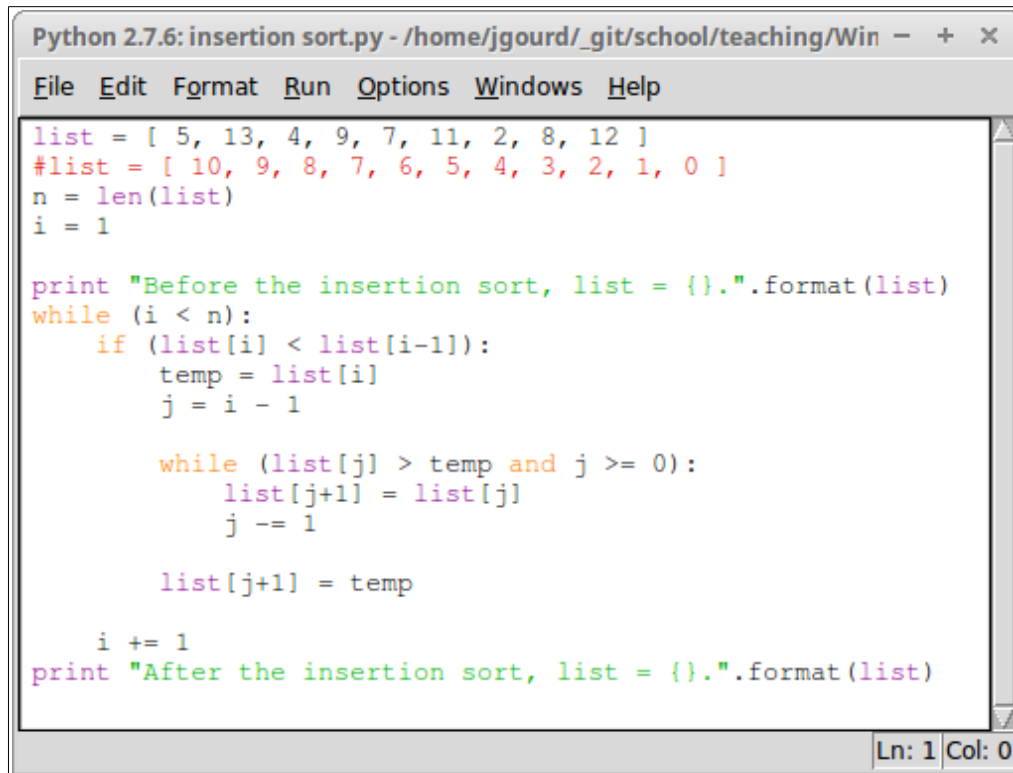
The status bar at the bottom right shows "Ln: 24 Col: 4".

Take a closer look at the sample executions of the sequential search algorithm. Note how the values to search for (i.e., the various values of  $n$ ) were picked to adequately (and fully) test the algorithm. The first value, 7, represents some arbitrary element in the middle of the list (i.e., not at the beginning or the end). The second and third values, 5 and 12, represent the first and last elements in the list respectively. The final value, 10, represents an item not in the list. This tests the boundary conditions (first, last, none), which is an important part of testing any algorithm.

Now let's take a look at the insertion sort algorithm that was implemented in a previous activity:

```
 $n \leftarrow$  length of the list
 $i \leftarrow 2$ 
repeat
  if item  $i$  of list < item  $i-1$  of list
  then
    temp  $\leftarrow$  item  $i$  of list
     $j \leftarrow i - 1$ 
    repeat
      if item  $j$  of list > temp
      then
        replace item  $j+1$  of list with item  $j$  of list
      end
       $j \leftarrow j - 1$ 
    until  $j = 0$  or item  $j$  of list not > temp
    replace item  $j+1$  of list with temp
  end
   $i \leftarrow i + 1$ 
until  $i > n$ 
```

This algorithm was tailored for Scratch. Recall that lists in Scratch begin at index 1 (they begin at index 0 in Python). Also recall that Scratch has a useful **replace item n of list l with x** block. Python has no such convenient *construct*. However, we can modify the algorithm to work in Python as follows:



```
Python 2.7.6: insertion sort.py - /home/jgourd/_git/school/teaching/Win - + x
File Edit Format Run Options Windows Help

list = [ 5, 13, 4, 9, 7, 11, 2, 8, 12 ]
#list = [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 ]
n = len(list)
i = 1

print "Before the insertion sort, list = {}".format(list)
while (i < n):
    if (list[i] < list[i-1]):
        temp = list[i]
        j = i - 1

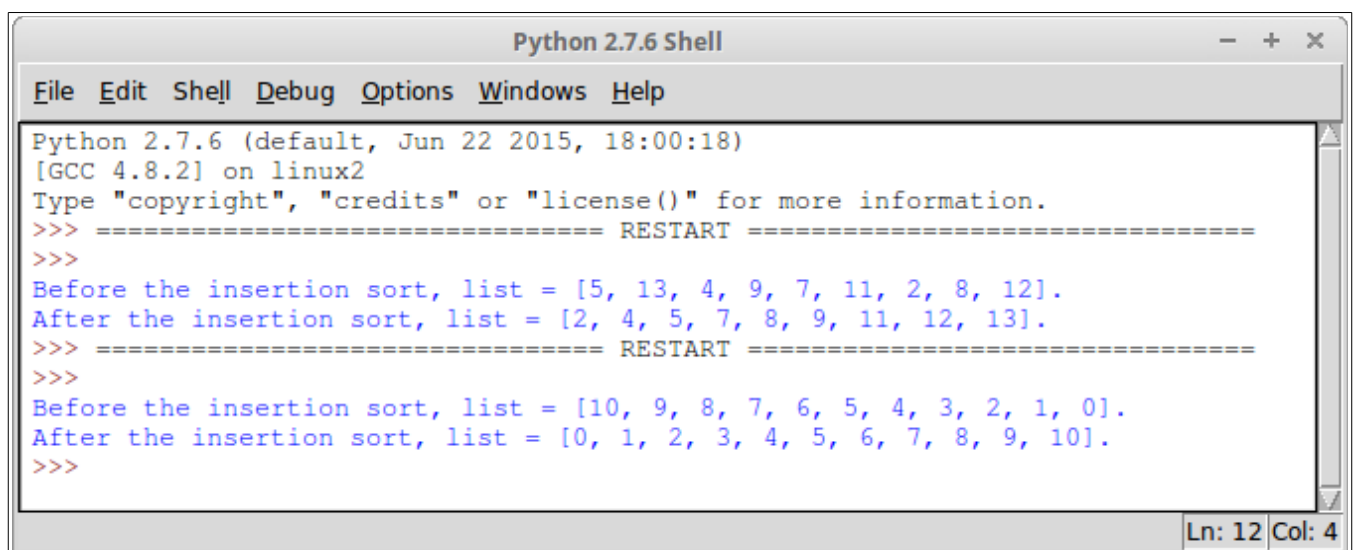
        while (list[j] > temp and j >= 0):
            list[j+1] = list[j]
            j -= 1

        list[j+1] = temp

    i += 1
print "After the insertion sort, list = {}".format(list)

Ln: 1 Col: 0
```

The commented list in the second line makes it simple to test different lists. Here's how it looks in IDLE with a few sample runs (where the initial lists are different):



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help

Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Before the insertion sort, list = [5, 13, 4, 9, 7, 11, 2, 8, 12].
After the insertion sort, list = [2, 4, 5, 7, 8, 9, 11, 12, 13].
>>> ===== RESTART =====
>>>
Before the insertion sort, list = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0].
After the insertion sort, list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
>>>

Ln: 12 Col: 4
```

Ready to try one on your own? Recall the pseudocode for the binary search (an efficient search that only works on sorted lists):

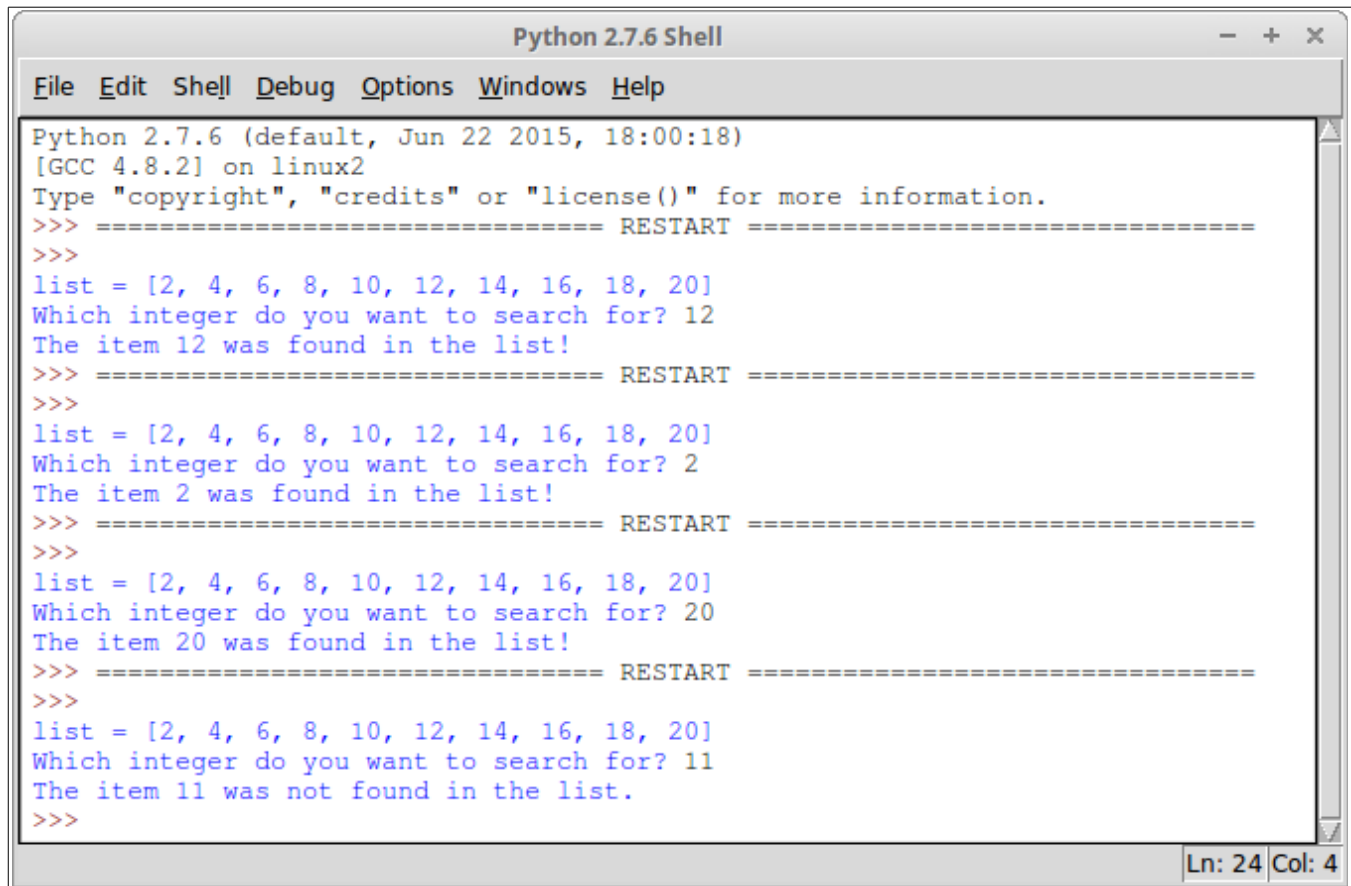
```
num ← number to search for
n ← number of items in the list
repeat
  mid ← floor( $n / 2$ ) + 1
  if num = item at mid of the list
  then
    display "num was found!"
    stop
  else if num > item at mid of the list
  then
    discard items at 1 through mid of the list
  else
    discard items at mid through n of the list
  end
  n ← number of items in the list
until n = 0
display "num was not found!"
```

This algorithm was tailored to work in Scratch. In general purpose programming languages, we typically do not *discard* items from a list. Instead, we leave them there and figure out another way to effectively remove elements from consideration. Here's a modified algorithm for the binary search that implements this idea:

```
num ← number to search for
found ← false
first ← 0
last ← number of items in the list - 1
while first ≤ last and found ≠ true
    mid ← floor((first + last) / 2)
    if num = item at mid of the list
        then
            found ← true
        else if num > item at mid of the list
            then
                first ← mid + 1
        else
            last ← mid - 1
    end
display found
```

Try to implement this algorithm in Python in the space below:

Here are some sample runs in IDLE of a working binary search algorithm:



```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
list = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Which integer do you want to search for? 12
The item 12 was found in the list!
>>> ===== RESTART =====
>>>
list = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Which integer do you want to search for? 2
The item 2 was found in the list!
>>> ===== RESTART =====
>>>
list = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Which integer do you want to search for? 20
The item 20 was found in the list!
>>> ===== RESTART =====
>>>
list = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Which integer do you want to search for? 11
The item 11 was not found in the list.
>>>
```

## Activity 1

See the *Raspberry Pi Activity 2: My Binary Addiction v2* document.

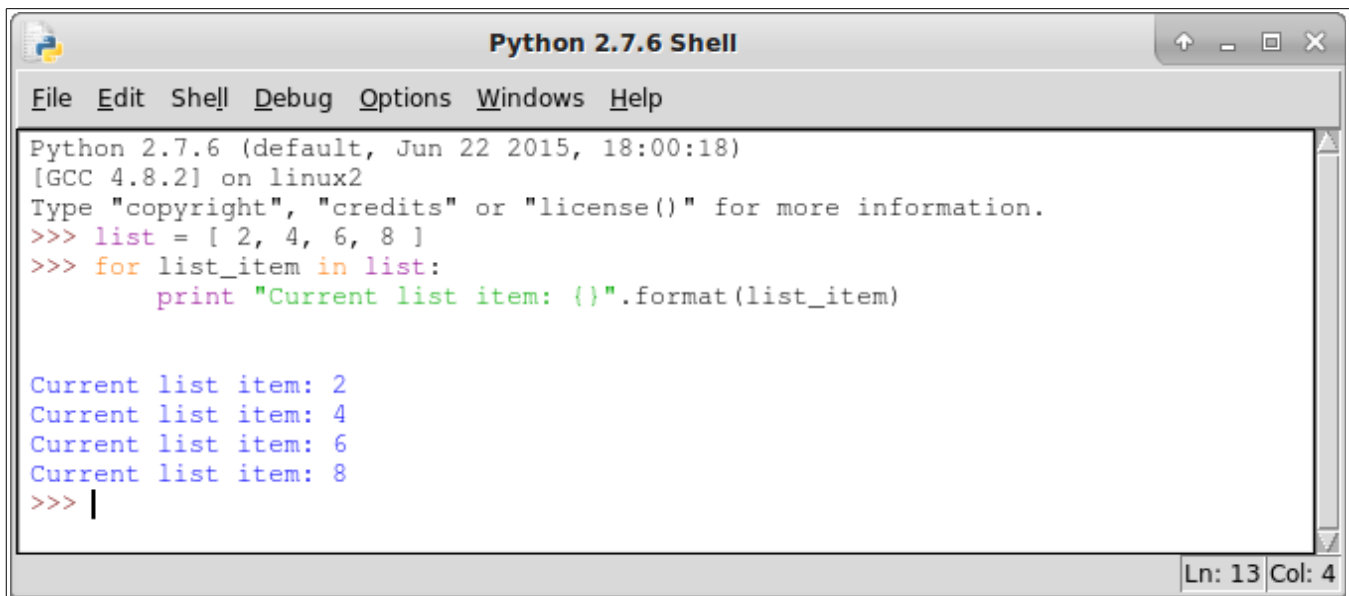
## The for loop

As briefly mentioned earlier in this lesson, Python provides one more repetition construct called the **for** loop. Typically, while loops are considered to be sentinel-driven; that is, they require a condition to either be true or false that indicates execution of the statements in the loop. Recall that Scratch also had a **repeat-n** block that repeated a task a set (or fixed) number of times. This kind of loop can be implemented in Python using the for loop construct.

The structure of a for loop in Python is:

```
for iterating_variable in sequence:
    loop_body
```

So far, the only sequence that has been discussed is the list. Here's an example that uses the for loop on a list:



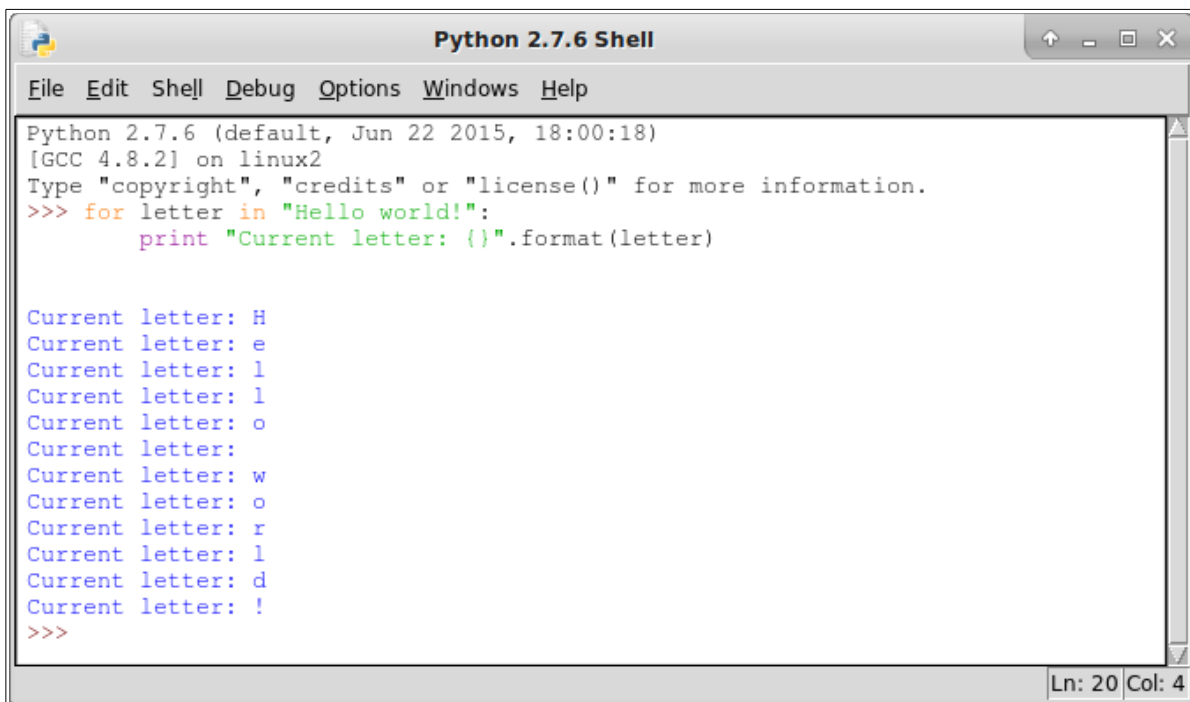
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> list = [ 2, 4, 6, 8 ]
>>> for list_item in list:
    print "Current list item: {}".format(list_item)

Current list item: 2
Current list item: 4
Current list item: 6
Current list item: 8
>>> |
```

Ln: 13 Col: 4

The variable `list_item` is used as an iterator. That is, it takes on the value of each item in the list at each iteration of the for loop. The first time, `list_item` takes on the first item in the list, 2. The second time, it takes on the second item in the list, 4. Eventually, it takes on the last item in the list, 8. In total, the body of the for loop executes once for each item in the list (or four times).

The for loop is actually quite powerful and flexible. It can, for example, iterate through the letters of a string:



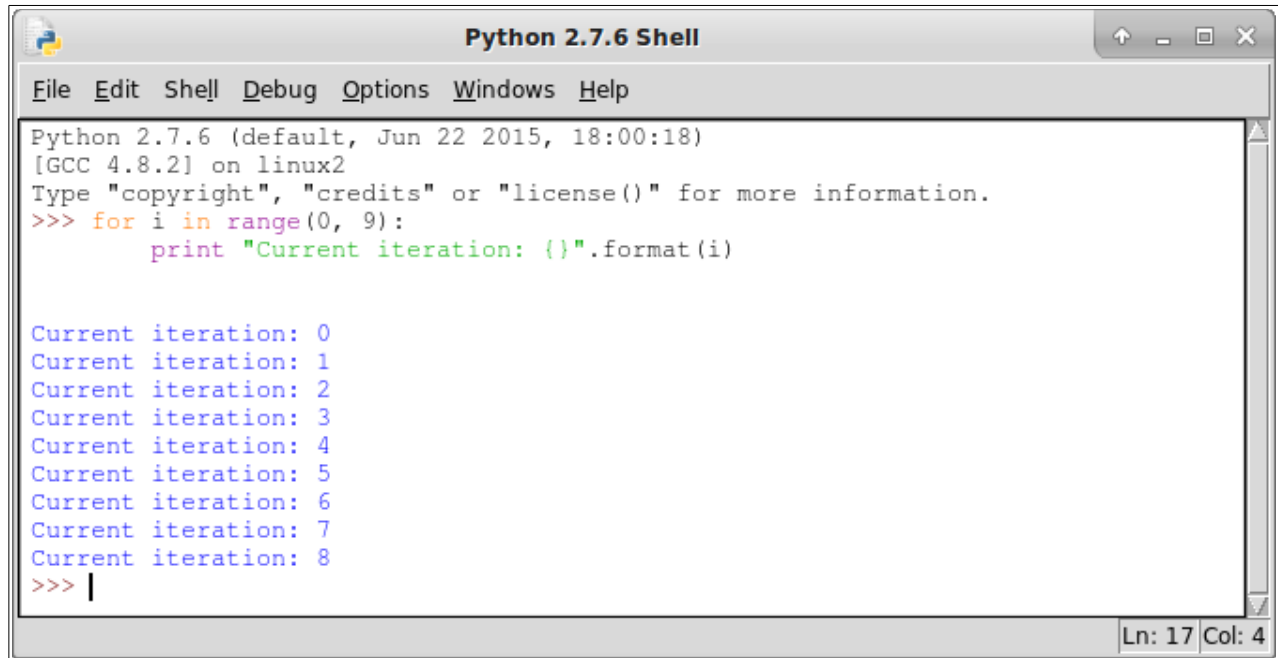
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> for letter in "Hello world!":
    print "Current letter: {}".format(letter)

Current letter: H
Current letter: e
Current letter: l
Current letter: l
Current letter: o
Current letter: 
Current letter: w
Current letter: o
Current letter: r
Current letter: l
Current letter: d
Current letter: !
>>>
```

Ln: 20 Col: 4



Typically, for loops iterate through a counter. The counter can then be used to refer to a variety of things, including the index of the elements of a list. To structure a for loop such that it iterates, say from 0 through 8, we would use the built-in Python function, `range()`:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> for i in range(0, 9):
    print "Current iteration: {}".format(i)

Current iteration: 0
Current iteration: 1
Current iteration: 2
Current iteration: 3
Current iteration: 4
Current iteration: 5
Current iteration: 6
Current iteration: 7
Current iteration: 8
>>> |
```

The `range()` function typically takes two parameters: a start value and a stop value. The start value is included in the range; however, the stop value is not. For example, to iterate from -10 to 10:

```
for i in range(-10, 11)
```

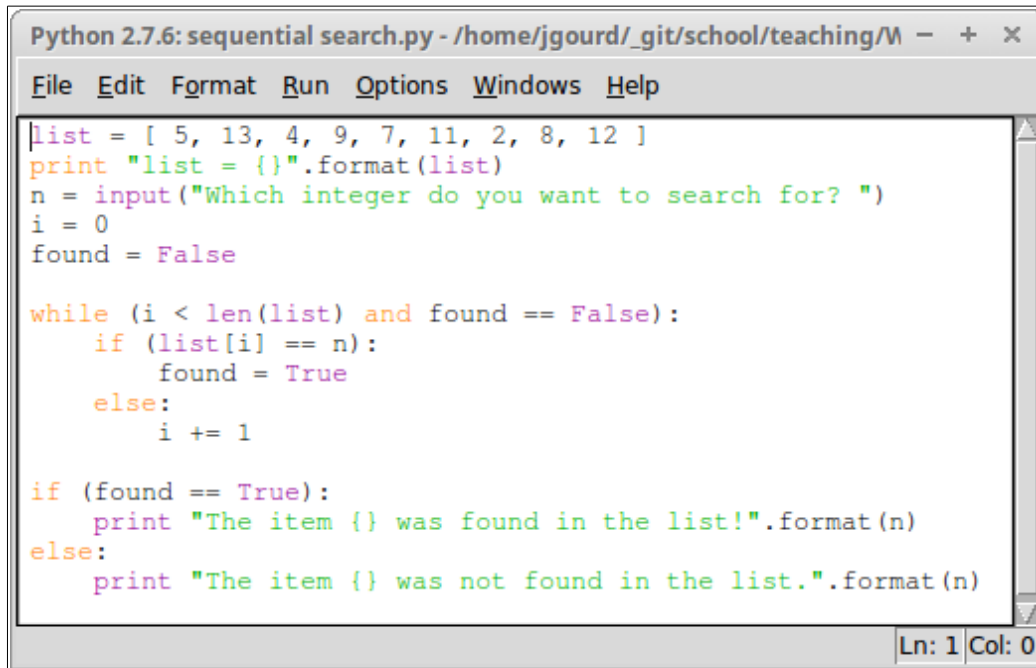
Note that an optional third parameter can be specified to indicate a step value. For example, suppose that you wish to iterate from 0 to 100 in increments of 10 (i.e., 0, 10, 20, ..., 90, 100):

```
for i in range(0, 101, 10)
```

More on the for loop will be discussed later.

## Exiting loops early

Sometimes, we wish to exit a loop before it is completely finished. Take, for example, the following sequential search:



```
Python 2.7.6: sequential search.py - /home/jgourd/_git/school/teaching/W - + x
File Edit Format Run Options Windows Help

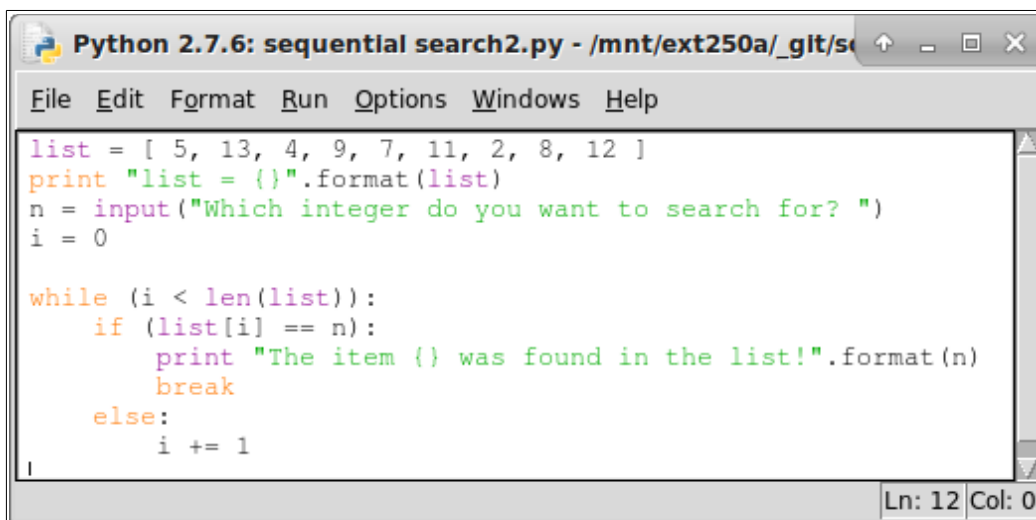
list = [ 5, 13, 4, 9, 7, 11, 2, 8, 12 ]
print "list = {}".format(list)
n = input("Which integer do you want to search for? ")
i = 0
found = False

while (i < len(list) and found == False):
    if (list[i] == n):
        found = True
    else:
        i += 1

if (found == True):
    print "The item {} was found in the list!".format(n)
else:
    print "The item {} was not found in the list.".format(n)

Ln: 1 Col: 0
```

Note that the variable *found* is used to break out of the loop if it is set to true (i.e., the specified value has been found in the list). We could modify this algorithm to not use the variable *found* and, instead, simply exit the loop early as follows:



```
Python 2.7.6: sequential search2.py - /mnt/ext250a/_git/s - + x
File Edit Format Run Options Windows Help

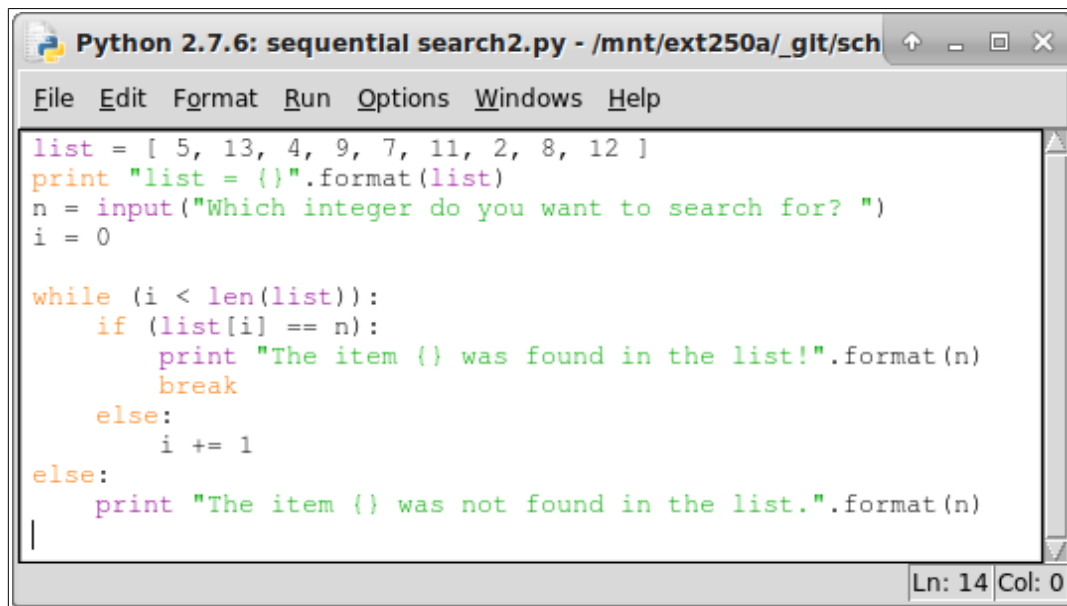
list = [ 5, 13, 4, 9, 7, 11, 2, 8, 12 ]
print "list = {}".format(list)
n = input("Which integer do you want to search for? ")
i = 0

while (i < len(list)):
    if (list[i] == n):
        print "The item {} was found in the list!".format(n)
        break
    else:
        i += 1

Ln: 12 Col: 0
```

Note the reserved word **break**. It is used to break out of a loop construct early (i.e., possibly before all of the iterations have completed). In the case above, if the specified value is found in the list, an appropriate message is displayed, and control breaks from the while loop and is transferred to any statements located beneath the while loop (there are none in the example above).

You may have noticed that the snippet above does not contain a message if the specified value has not been found (unlike the example shown earlier in this lesson). Loop constructs in Python also support an else clause (much like if-statements). For the while loop, the else clause occurs when the condition becomes false. Take, for example, the following modification of the sequential search:



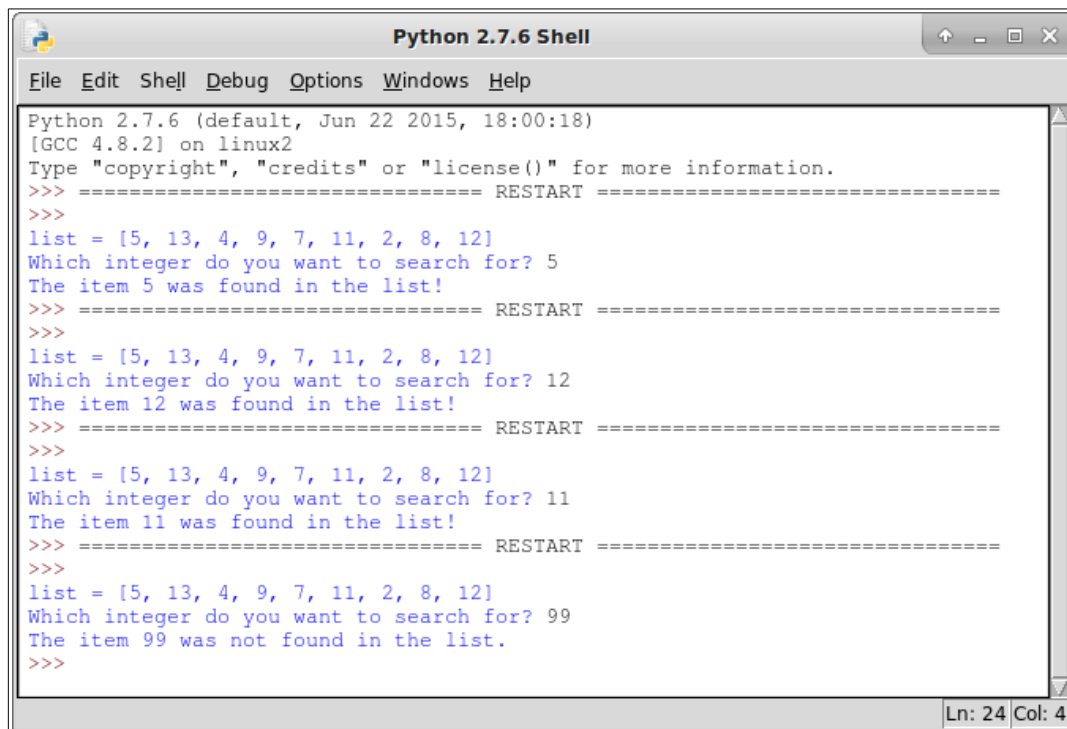
```
Python 2.7.6: sequential search2.py - /mnt/ext250a/_glt/sch
File Edit Format Run Options Windows Help

list = [ 5, 13, 4, 9, 7, 11, 2, 8, 12 ]
print "list = {}".format(list)
n = input("Which integer do you want to search for? ")
i = 0

while (i < len(list)):
    if (list[i] == n):
        print "The item {} was found in the list!".format(n)
        break
    else:
        i += 1
else:
    print "The item {} was not found in the list.".format(n)
```

Ln: 14 Col: 0

This change would inform the user if the specified item was not found in the list (i.e., a search through the entire length of the list has completed). Note that, if the specified item is indeed found in the list, the break statement is reached **which breaks out of the entire while loop (including the else clause)**. For this reason, the *not found* message is not displayed if the specified item is actually found in the list.

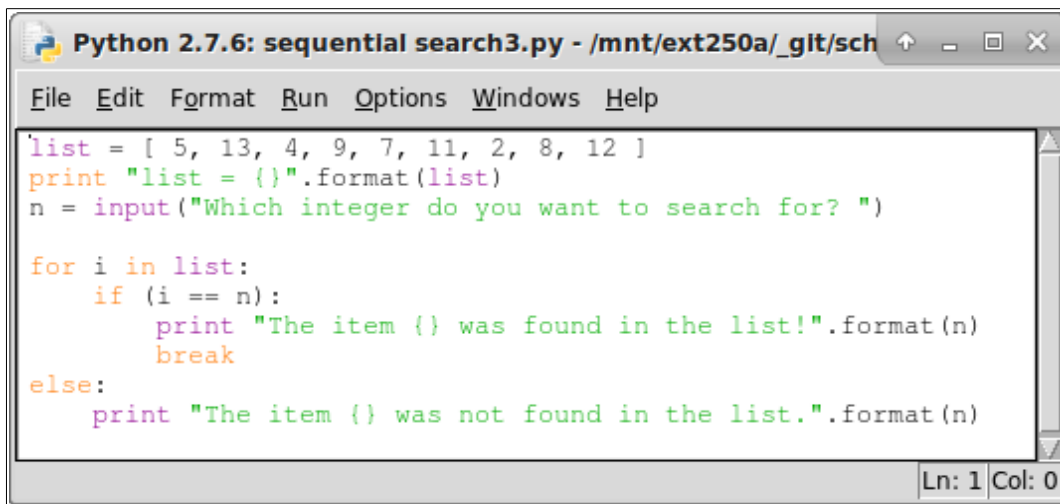


```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help

Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 5
The item 5 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 12
The item 12 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 11
The item 11 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 99
The item 99 was not found in the list.
>>>
```

Ln: 24 Col: 4

The same thing can be done with for loops. That is, a break statement and an else clause can be included if desired. Here's an example of the sequential search using a for loop:

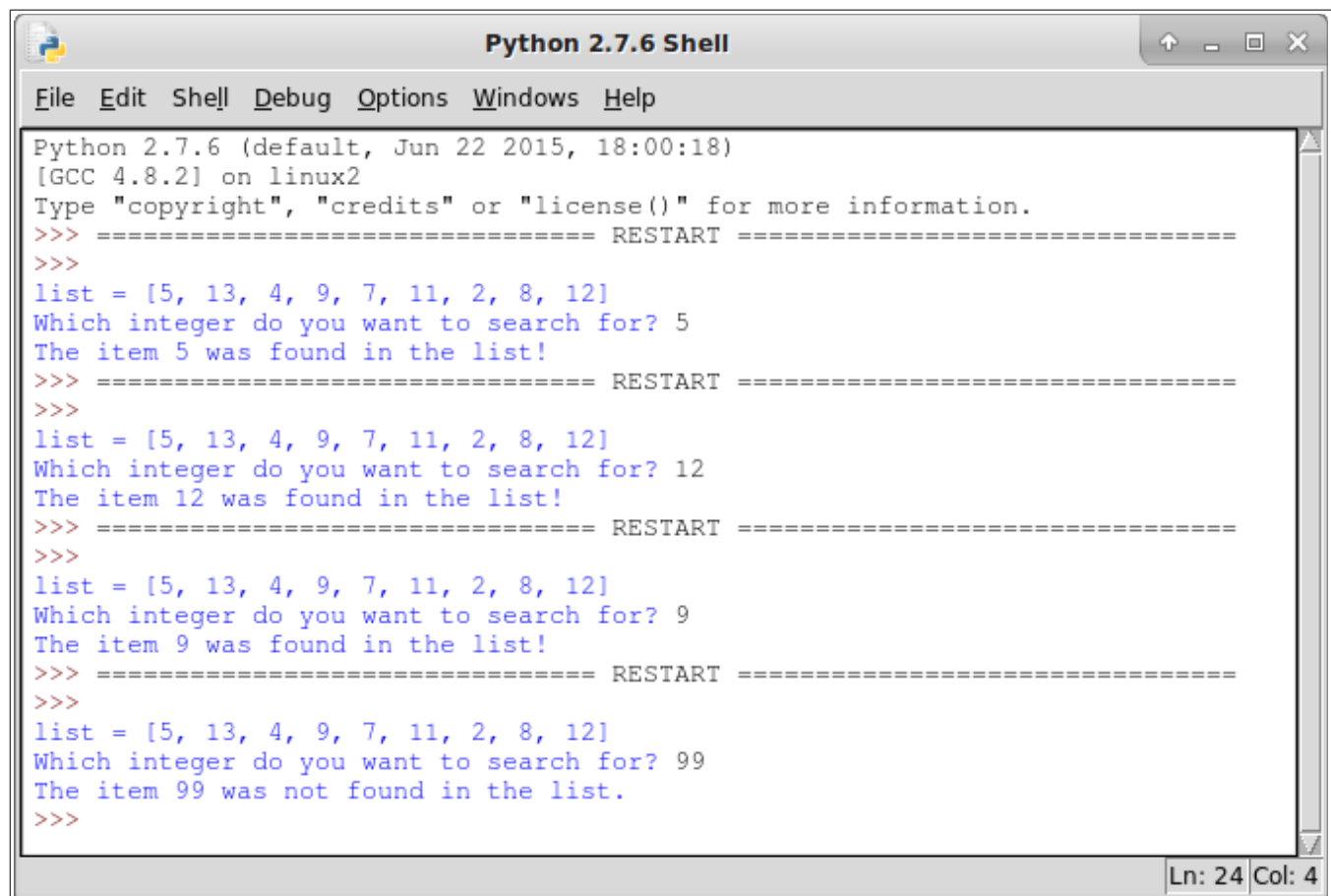


The screenshot shows a Python 2.7.6 IDE window titled "Python 2.7.6: sequential search3.py - /mnt/ext250a/\_git/sch". The menu bar includes File, Edit, Format, Run, Options, Windows, and Help. The code in the editor is as follows:

```
list = [ 5, 13, 4, 9, 7, 11, 2, 8, 12 ]
print "list = {}".format(list)
n = input("Which integer do you want to search for? ")

for i in list:
    if (i == n):
        print "The item {} was found in the list!".format(n)
        break
    else:
        print "The item {} was not found in the list.".format(n)
```

The status bar at the bottom right indicates "Ln: 1 Col: 0".



The screenshot shows a Python 2.7.6 Shell window titled "Python 2.7.6 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The output of the script is as follows:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 5
The item 5 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 12
The item 12 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 9
The item 9 was found in the list!
>>> ===== RESTART =====
>>>
list = [5, 13, 4, 9, 7, 11, 2, 8, 12]
Which integer do you want to search for? 99
The item 99 was not found in the list.
>>>
```

The status bar at the bottom right indicates "Ln: 24 Col: 4".

## Other operators

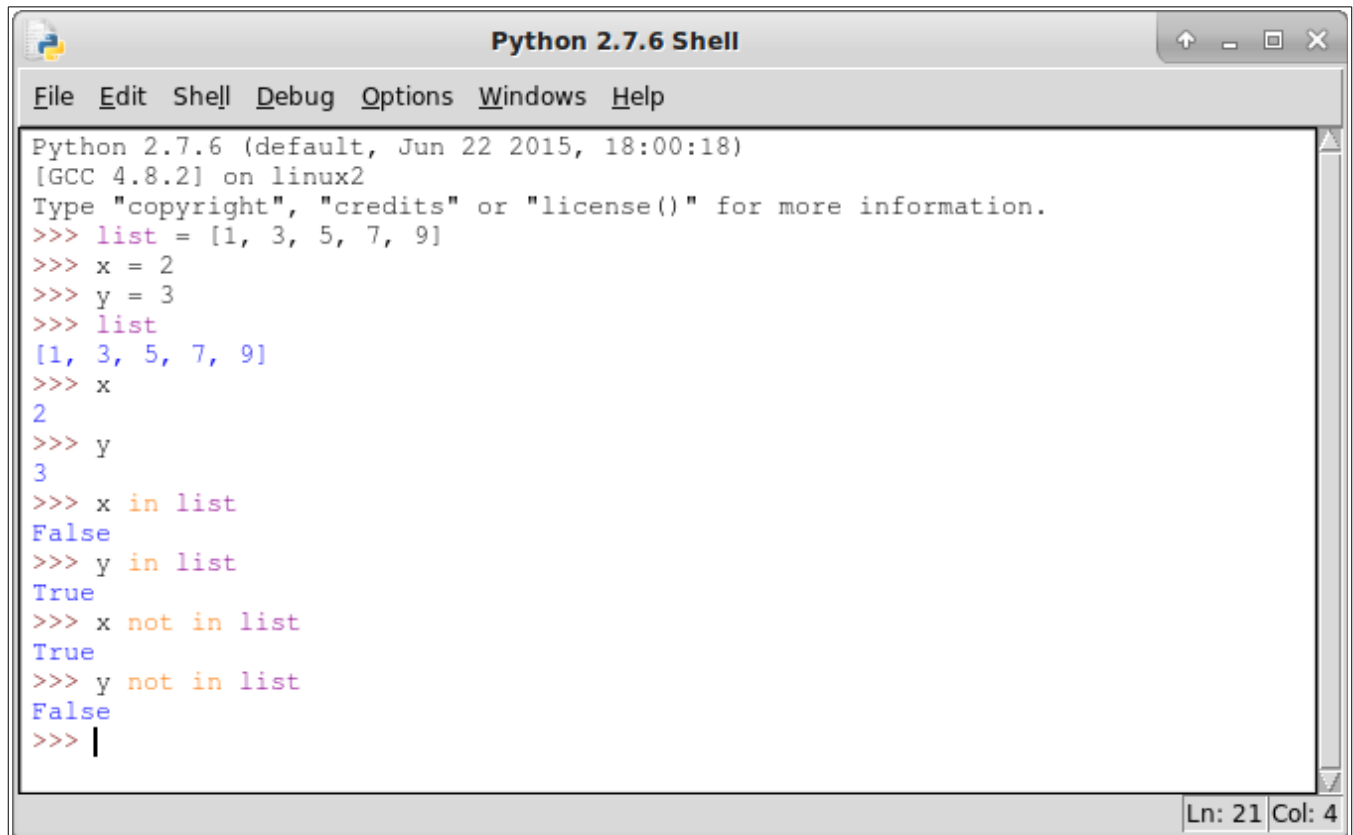
Python provides two more classes of operators: membership operators and identity operators. Although some have been used and introduced in previous examples, they have not yet been formally discussed.

Membership operators test for some value's membership in a sequence (e.g., to test if an element exists in a list, or if a character exists in a string). For the following table, assume that `list = [ 1, 3, 5, 7, 9 ]`, `x = 2` and `y = 3`.

Python Membership Operators and Examples		
<code>in</code>	Returns true if a specified value is in a specified sequence or false otherwise	<code>x in list</code> is False; <code>y in list</code> is True
<code>not in</code>	Returns true if a specified value is not in a specified sequence or true otherwise	<code>x in list</code> is True; <code>y in list</code> is False

You have seen this in previous for loop examples (e.g., `for i in list`).

Here are a few examples in IDLE:

A screenshot of the Python 2.7.6 Shell window. The window title is "Python 2.7.6 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The shell output shows the following code and results:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> list = [1, 3, 5, 7, 9]
>>> x = 2
>>> y = 3
>>> list
[1, 3, 5, 7, 9]
>>> x
2
>>> y
3
>>> x in list
False
>>> y in list
True
>>> x not in list
True
>>> y not in list
False
>>> |
```

The status bar at the bottom right shows "Ln: 21 Col: 4".

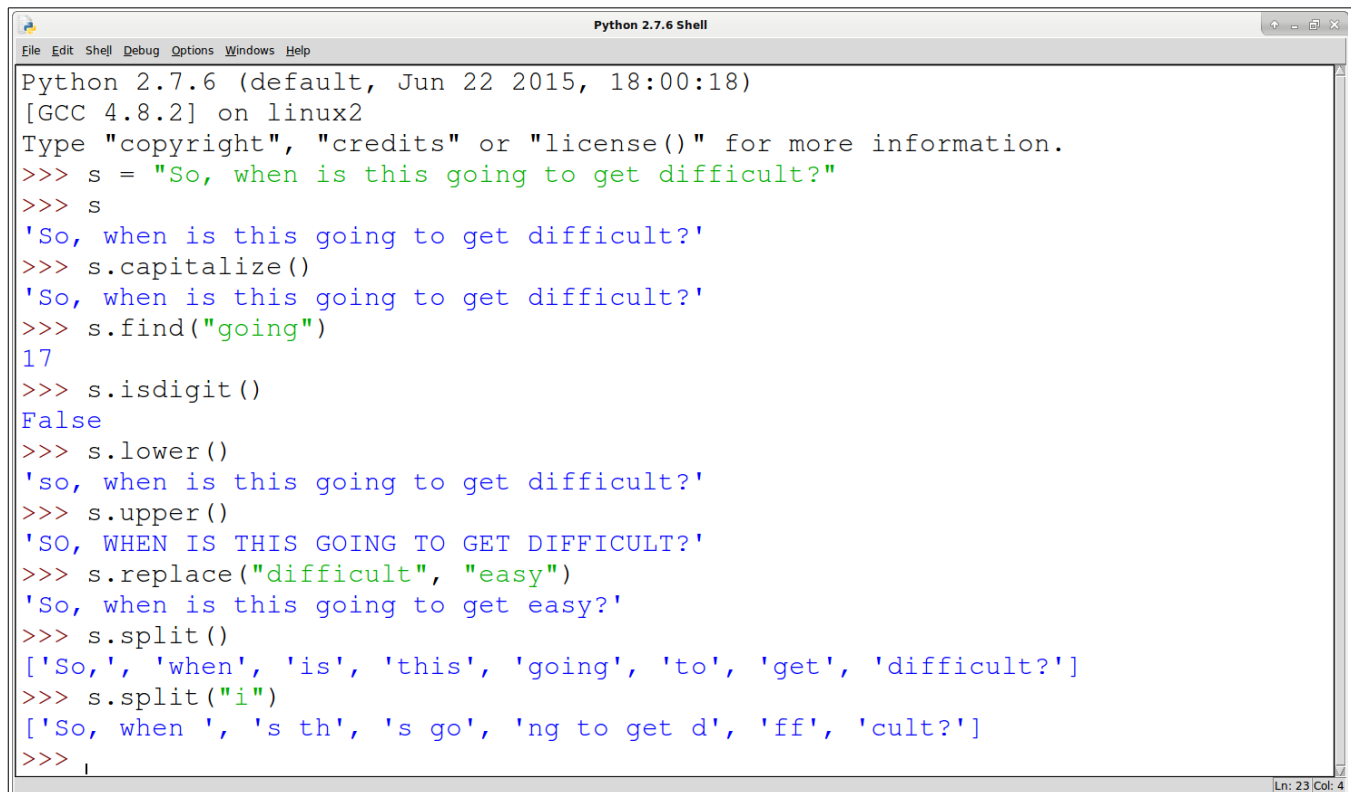
Identity operators are primarily used in the object-oriented paradigm and will be discussed at a later time.

## String methods

Strings are often necessary when writing programs. As such, Python provides a variety of methods that work on strings. You have already seen one such method, **format()**, that formats a string as specified (we did this earlier in one variant of the **print** statement). The following table lists some of the more useful string methods:

Python String Methods/Functions	
<code>str.capitalize()</code>	capitalizes the first character of a string
<code>str.find()</code>	returns the first index of a string within another string
<code>str.format()</code>	formats a string according to a specification
<code>str.isdigit()</code>	determines if a string consists only of numeric characters
<code>str.lower()</code>	converts a string to lowercase
<code>str.replace()</code>	replaces all occurrences of a string (within a string) with another string
<code>str.split()</code>	returns a list of the words in a string
<code>str.upper()</code>	converts a string to uppercase

These string methods are explained in greater detail in a variety of online sources. We suggest that you Google them and try them out. Here are a few examples in IDLE:



```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> s = "So, when is this going to get difficult?"
>>> s
'So, when is this going to get difficult?'
>>> s.capitalize()
'So, when is this going to get difficult?'
>>> s.find("going")
17
>>> s.isdigit()
False
>>> s.lower()
'so, when is this going to get difficult?'
>>> s.upper()
'SO, WHEN IS THIS GOING TO GET DIFFICULT?'
>>> s.replace("difficult", "easy")
'So, when is this going to get easy?'
>>> s.split()
['So,', 'when', 'is', 'this', 'going', 'to', 'get', 'difficult?']
>>> s.split("i")
['So, when ', 's th', 's go', 'ng to get d', 'ff', 'cult?']
>>>
```

Note the execution of the string method **str.find()** above: `s.find("going")`. This string method returns the first index of the string, “going”, within the string, `s`. Why is the result 17? At first glance, it

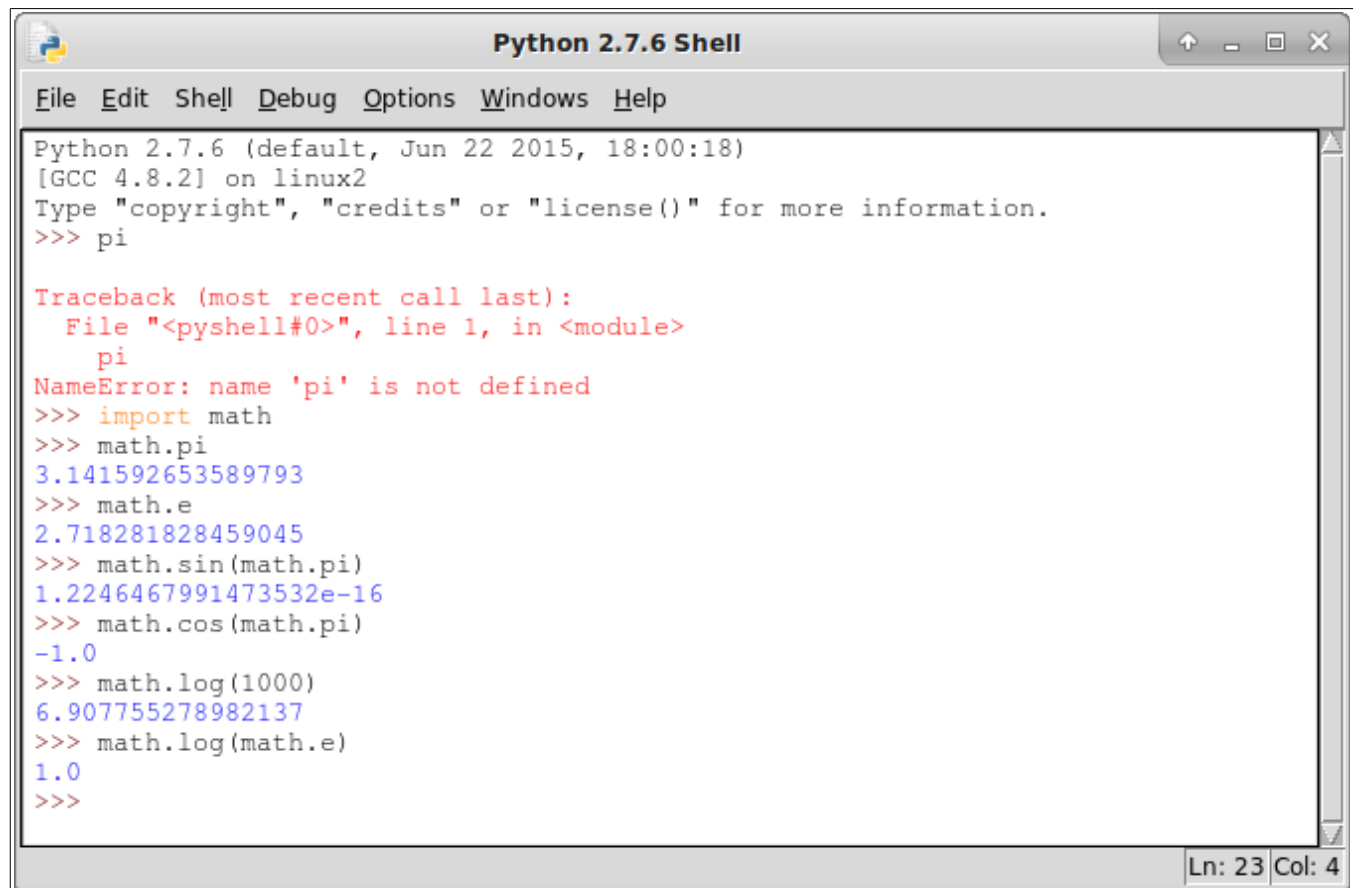
seems that the first character of the string, “going”, is at position 18. However, the characters of a string in Python begin at index 0 (similar to lists).

### Importing external libraries

It is often useful (and necessary) to import external functionality into our programs. Often, others have designed functions and other bits of code that may prove useful. We don't always want to recreate things that already exist. Python supports the importing of such things via the **import** reserved word. For example, many of the programs we create require the use of mathematical functions (beyond simple arithmetic; e.g., sin, cos, tan) or mathematical constants (e.g., pi, e). The structure of an import statement is as follows:

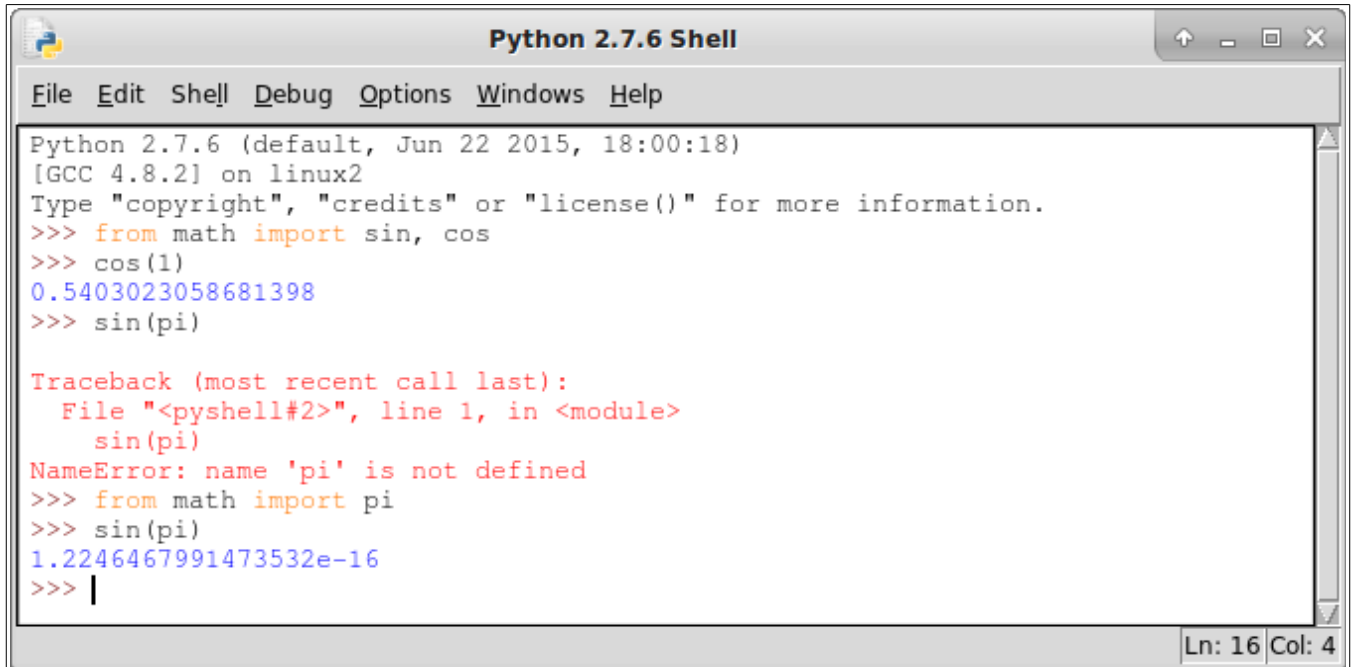
```
import library
```

Pretty simple. Here's an example of the importing and use of the math library:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> pi
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    pi
NameError: name 'pi' is not defined
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.sin(math.pi)
1.2246467991473532e-16
>>> math.cos(math.pi)
-1.0
>>> math.log(1000)
6.907755278982137
>>> math.log(math.e)
1.0
>>>
```

Note in the example, the invalid use of *pi* before importing that math library. In addition, any value or function used in a library must be fully qualified with the name of the library (e.g., we need to specify *math.pi* and not just *pi*). Alternatively, we can itemize what we wish to import from a library. This allows us to use values and functions directly without having to specify the library name. Here is an example:



```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> from math import sin, cos
>>> cos(1)
0.5403023058681398
>>> sin(pi)

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    sin(pi)
NameError: name 'pi' is not defined
>>> from math import pi
>>> sin(pi)
1.2246467991473532e-16
>>> |
```

Ln: 16 Col: 4