Introduction to Computer Programming Pillar: Computer Programming

Machine language

Definition: ***Machine language*** *is a set of specific instructions that can be executed directly by a computer. This language is typically made up of binary digits (1s and 0s).*

A computer can only understand machine language. Machine language instructions are entirely made up of binary digits and can be directly executed by the CPU. Since we have a particularly hard time understanding 1s and 0s, early programmers assigned a set of mnemonics to represent machine code instructions so that they would be a bit more readable. This mapping became known as **assembly language**. People still write code in assembly language, but it is not typically used to create large scale applications.

Programming language

The kinds of languages that are widely used today are known as programming languages. Programming languages allow us to represent algorithms in a way that is similar to English but is more structured and much less ambiguous.

Definition: *A* ***programming language*** *is a precisely constructed language that is specifically used to communicate instructions to a computer.*

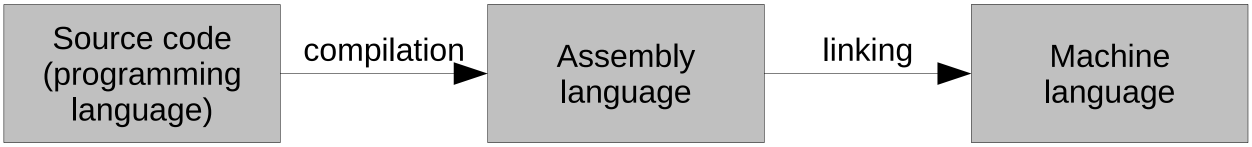
English is a spoken language. As such, it was spoken first, rules were later defined and written down. Therefore, there are many exceptions (i.e., words and phrases that are grammatically correct but don't conform to the general grammar/spelling rules). Spoken languages are sometimes ambiguous and open to interpretation. This means that a single statement can have multiple meanings. For example, the statement “I made the robot fast” can mean several different things. Does it mean that the robot was built quickly? Or does it mean that the robot was modified so that it would move around more quickly than it did before? Perhaps it means that the robot is named Fast. Or maybe that we managed to make the robot stop eating nuts and bolts.

Humans rely on external factors like context and body language to understand the true meaning of a statement in a spoken language. And even then mistakes in interpretation still happen. With computers however, we need a language that is so structured and unambiguous that every computer can understand and interpret a given statement in the exact same way. For example, we don't want two different computers giving us two completely different answers to the arithmetic expression `1 + 1`.
In contrast to spoken languages, programming languages are first defined with rules. The language itself is then derived from those rules. Programming languages are therefore quite structured and not ambiguous. They are very precise and logical.

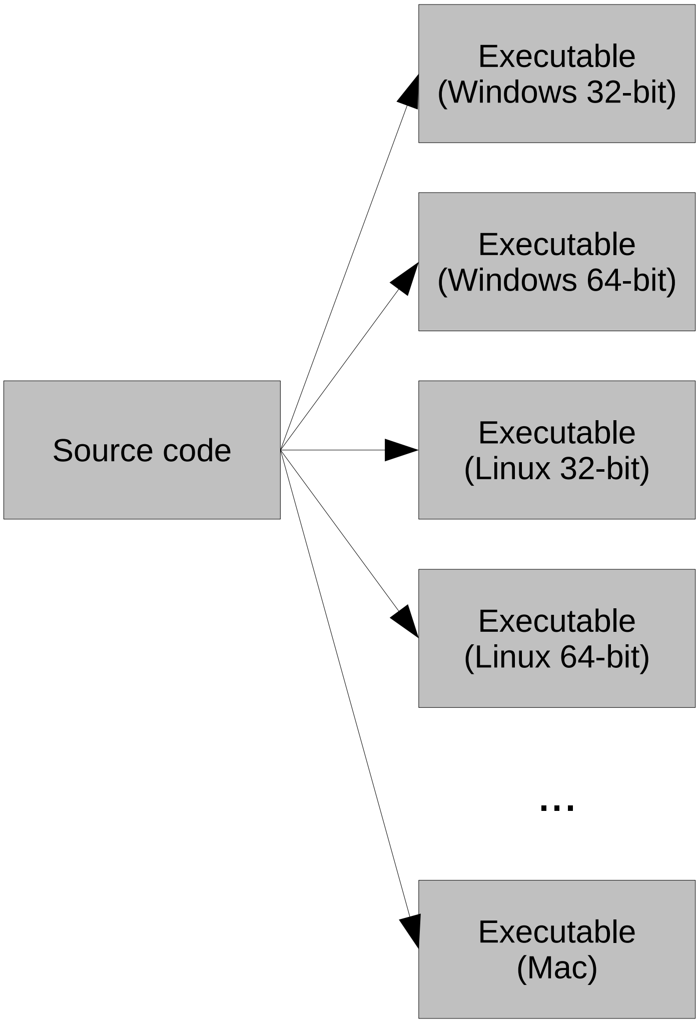
There are many different programming languages that can be used to describe an algorithm. One of them, for example, is called Python, and it is what we will be using for the majority of the Living *with* Cyber curriculum. It is the duty of the programmer to write down the tasks that he/she wants done in a given programming language. Since computers can only understand machine language, we utilize an application known as a **compiler** that translates this programming language into machine language.

Definition: *A **compiler** is a tool used to translate an algorithm expressed in a programming language to machine language. The process by which this conversion from programming language to machine language is done is called **compilation**.*

The compilation process takes an algorithm written in a programming language and translates it to assembly language. From there, a process known as *linking* converts the assembly language to machine language. This is illustrated by the figure below:



Once machine language is generated to match a program, the computer can then directly execute the program and implement the algorithm. A fully compiled language is only executable by a CPU with the same characteristics and operating system (often, including version) as that which it was compiled for. A programmer who wants wide distribution of his software will need to compile source code to the various destination computing architectures and operating systems that are the most likely to be used by the target audience for the application. Of course, the programmer could simply distribute source code and let users compile that themselves. Often, however, programmers do not wish to distribute source code for a variety of reasons (e.g., intellectual property). The figure below shows how a program would need to be compiled numerous times to cover a range of target computing architectures and operating systems:


Executable (Windows 32-bit)
Executable (Windows 64-bit)
Source code
Executable (Linux 32-bit)
Executable (Linux 64-bit)
...
Executable (Mac)

Not all programming languages are compiled to machine language. Some are never compiled and are executed, one instruction at a time, by an **interpreter**.

Definition: *An **interpreter** is a tool used to evaluate instructions, written in a programming language, as the program is executed.*

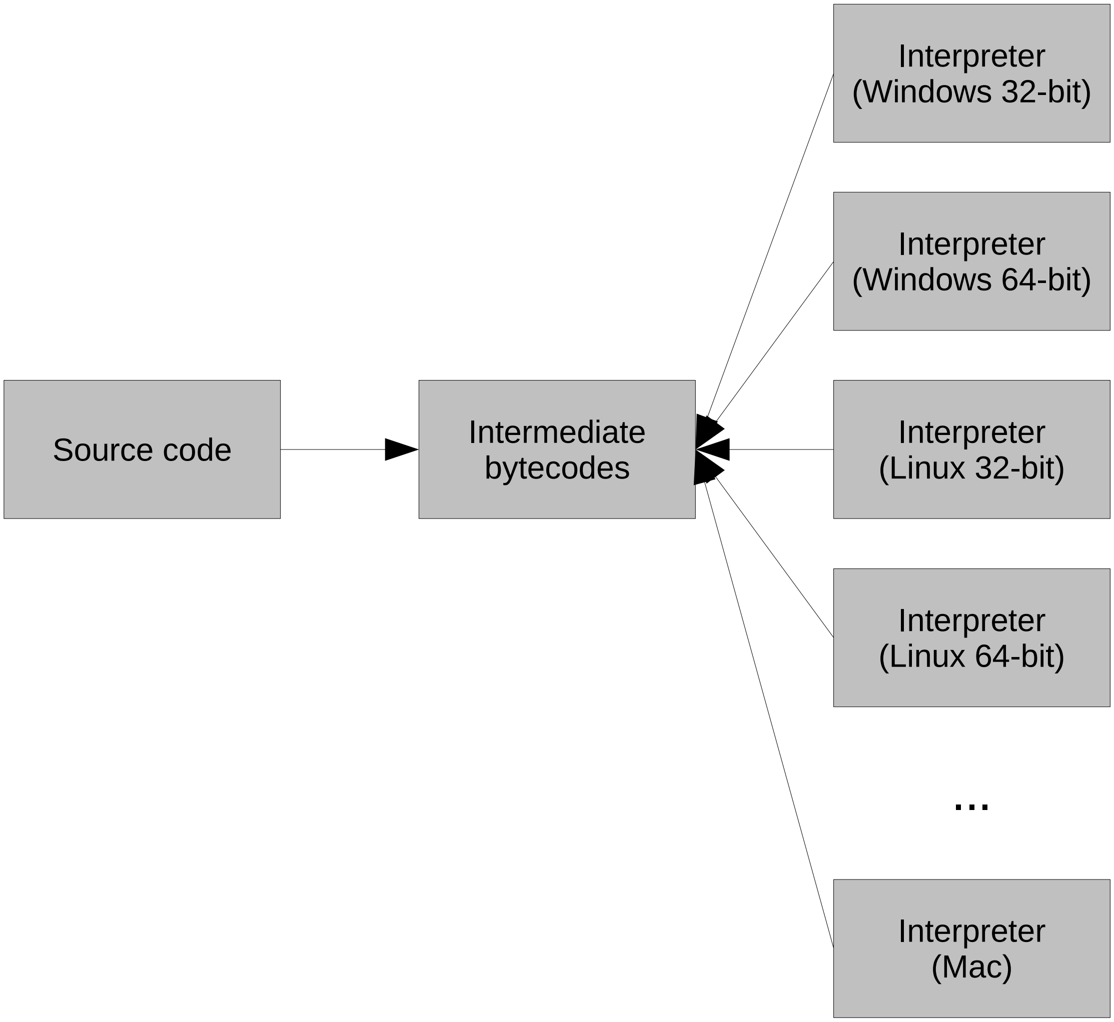
An interpreter can be thought of as a real time compiler that executes high level programming language instructions, one at a time. Interpreted languages are much slower to execute than compiled languages. Examples of interpreted languages are Python, PHP, JavaScript, and Perl. To execute a program written in an interpreted language, you must have an appropriate interpreter installed on your computing system.

Interpreted languages also require programmers to distribute their source code, and users to have an appropriate interpreter installed on their system. Maintaining code privacy is therefore not possible with interpreted languages.

Partially compiled and interpreted languages combine the convenience of interpreted languages (i.e., not having to compile source code to a large number of target machine language executables) and the privacy and speed of compiled languages (i.e., not having to distribute source code).

Definition: *A **hybrid** language is a programming language that is compiled down to an intermediate language, and then interpreted (while the program is executed) from there.*

Examples of partially compiled languages are Java, Python, and Lisp. Note that Python can be strictly interpreted or partially compiled depending on the programmer's preferences. The intermediate language is distributed and subsequently executed on any computing platform that has an interpreter for the intermediate language. For example, Java source code is typically expressed in a `.java` file and partially compiled to Java bytecodes (in a `.class` file) that can then be distributed. A Java Virtual Machine (JVM) executes the bytecodes by interpreting each instruction, one at a time. The benefit of this method is that a programmer can distribute a single file to everyone, regardless of CPU architecture and operating system. Anyone wanting to execute the file simply needs to have a version of the JVM for their computing system. This is illustrated in the figure below:


Source code
Intermediate bytecodes
Interpreter (Windows 32-bit)
Interpreter (Windows 64-bit)
Interpreter (Linux 32-bit)
Interpreter (Linux 64-bit)
...
Interpreter (Mac)

Programming paradigms

Over the past forty years or so, three general classes, or paradigms, of programming languages have emerged. These paradigms include the imperative paradigm, the functional paradigm, and the logical paradigm. In addition, during the past decade or so these paradigms have been extended to include object-oriented features. A language is classified as belonging to a particular paradigm based on the programming features it supports.

Object-oriented imperative languages are, by far, the most popular type of programming language. Both Java and C++ (two of the most used programming languages in industry) are object-oriented imperative languages. Scratch and Python are imperative languages – although Python does contains object-oriented attributes, Scratch does not.

The **imperative paradigm** is based on the idea that a program is a sequence of commands or instructions (usually called **statements**) that the computer is to follow to complete a task. The imperative style of programming is the oldest, and now with object-oriented extensions, continues to be far and away the most popular style of programming.

The Living *with* Cyber curriculum first (and very briefly) utilizes Scratch as the programming language. This is quickly followed by Python. Scratch is not intended to be used to create applications designed for production systems. That is, it is not a *general purpose* programming language. Instead, it is a teaching tool aimed at simplifying the process of learning to program. Scratch purposefully omits many features available in other popular programming languages in order to keep the language from becoming overly complex. This allows you to focus on the *big picture* rather than get bogged down in the complexities inherent in *real* programming languages and their development environments.

One way of thinking about writing Scratch programs is to compare it to programming in a *production* programming language with training wheels on. Complex and useful programs can be written in Scratch; however, there are many things that programmers are allowed to do in production languages that are not possible (at least not straightforward) in Scratch. For example, Scratch does not support functions and function calls directly, nor does it support recursion directly. These terms may not be familiar right now; however, these restrictions are designed to help beginning programmers avoid making common mistakes.

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 that you can build are limitless. From this point, we will use Python as the general purpose programming language in the course.

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). It is based on a few profound ideas (collectively known as **The Zen of Python** written by Tim Peters):

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one--and preferably only one--obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
```

```
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Did you know?

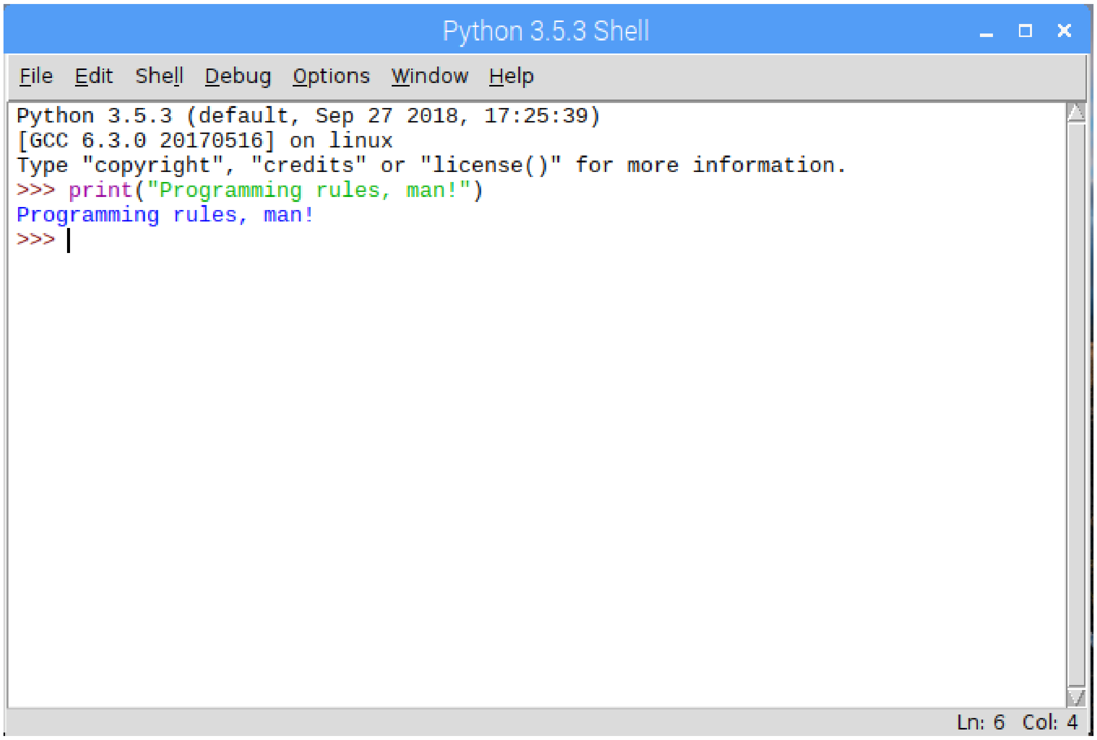
The name of the Python programming language is taken from a television series called Monty Python's Flying Circus (and not from the snake).

Integrated development environment

Many programmers write their programs in a general purpose programming language using nothing but a text-based 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).

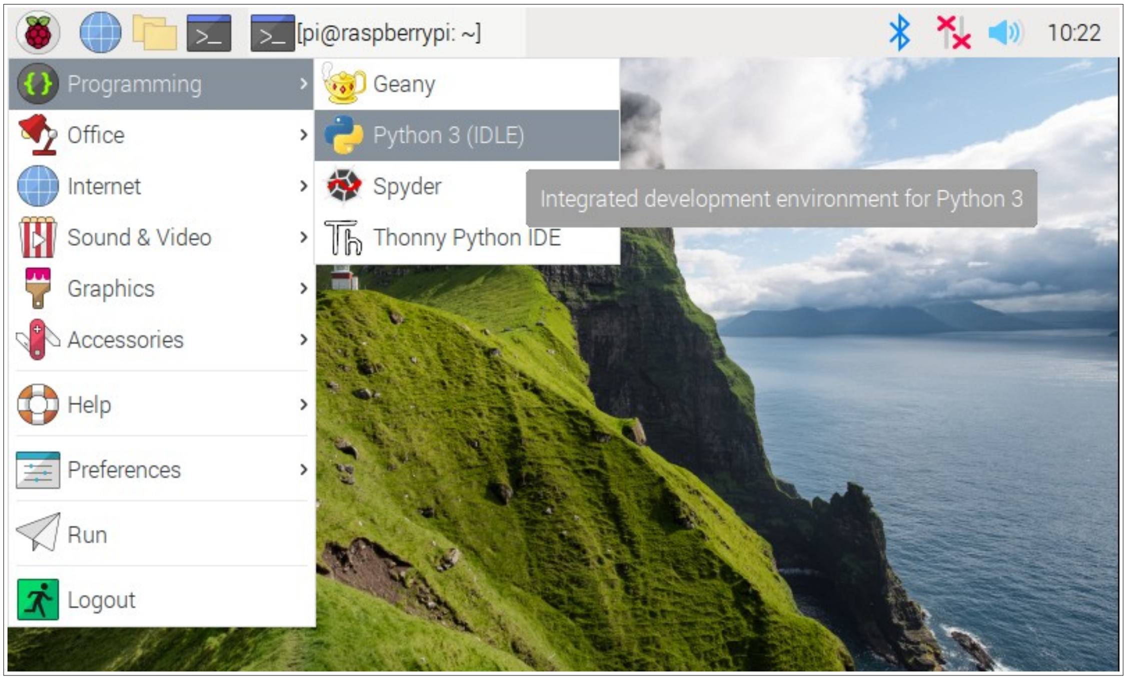
Definition: *An **Integrated Development Environment (IDE)** is a piece of software that allows computer programmers to design, execute, and debug computer programs in an integrated and flexible manner.*

On the Raspberry Pi, the IDE used to design Python programs is called IDLE (which stands for Python's **I**ntegrated **D**eve**L**opment **E**nvironment). 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):

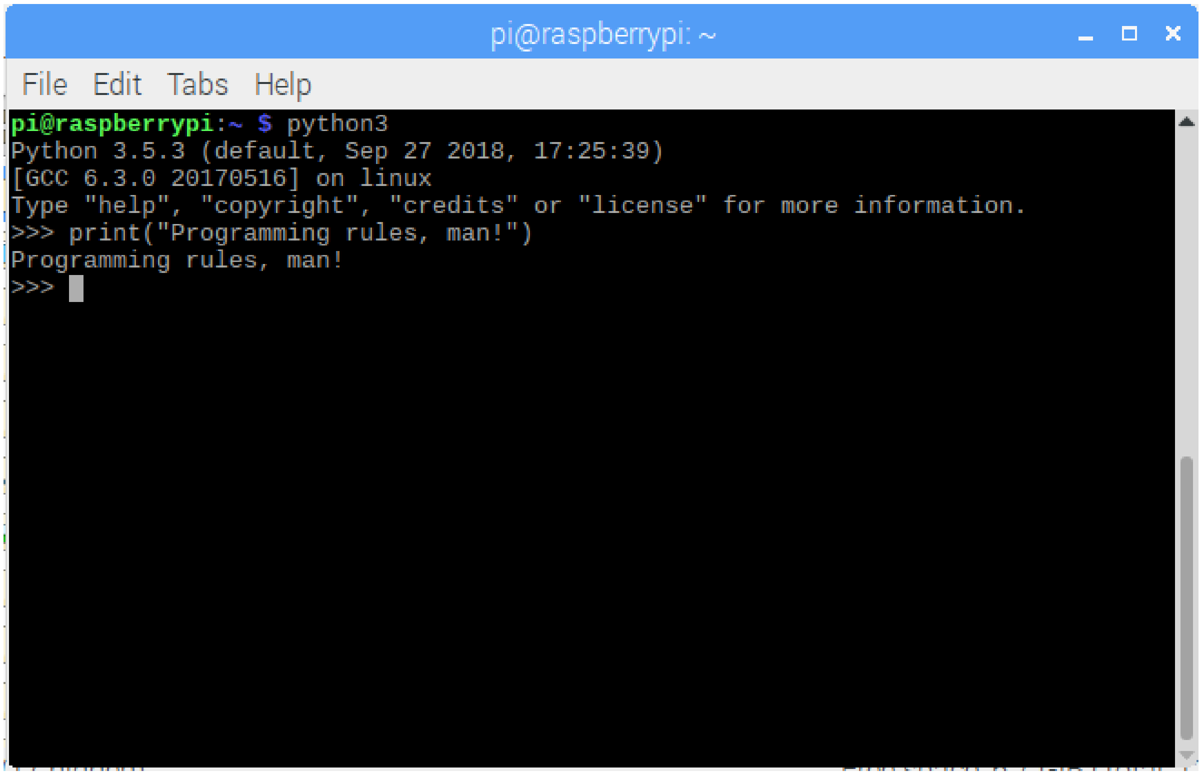


```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> print("Programming rules, man!")
Programming rules, man!
>>> 
```

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:

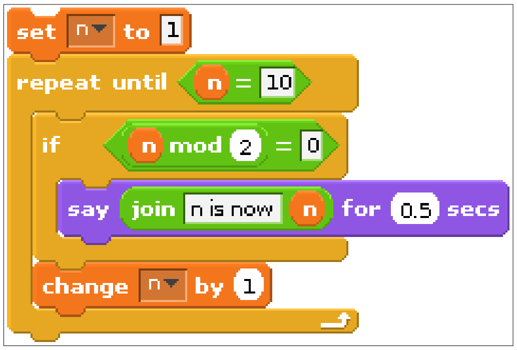


```
pi@raspberrypi:~ $ python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Programming rules, man!")
Programming rules, man!
>>> 
```

Scratch vs Python

Earlier in this 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:



```
set n to 1
repeat until n = 10
    if n mod 2 = 0
        say join n is now n for 0.5 secs
    change n by 1
```

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 mod 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
```

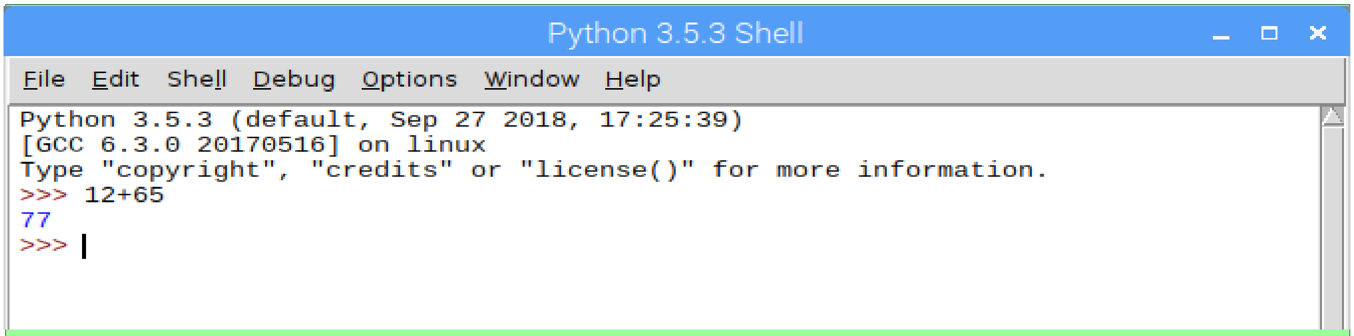
At this point, it is fine if you don't understand everything that's going on syntactically. The idea is simply to illustrate how Scratch and Python differ (and are similar!). But let's try to explain. The block, **set *n* to 1**, in Scratch is implemented in Python as, `n = 1`. Pretty similar! Python 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* mod 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 (and/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).

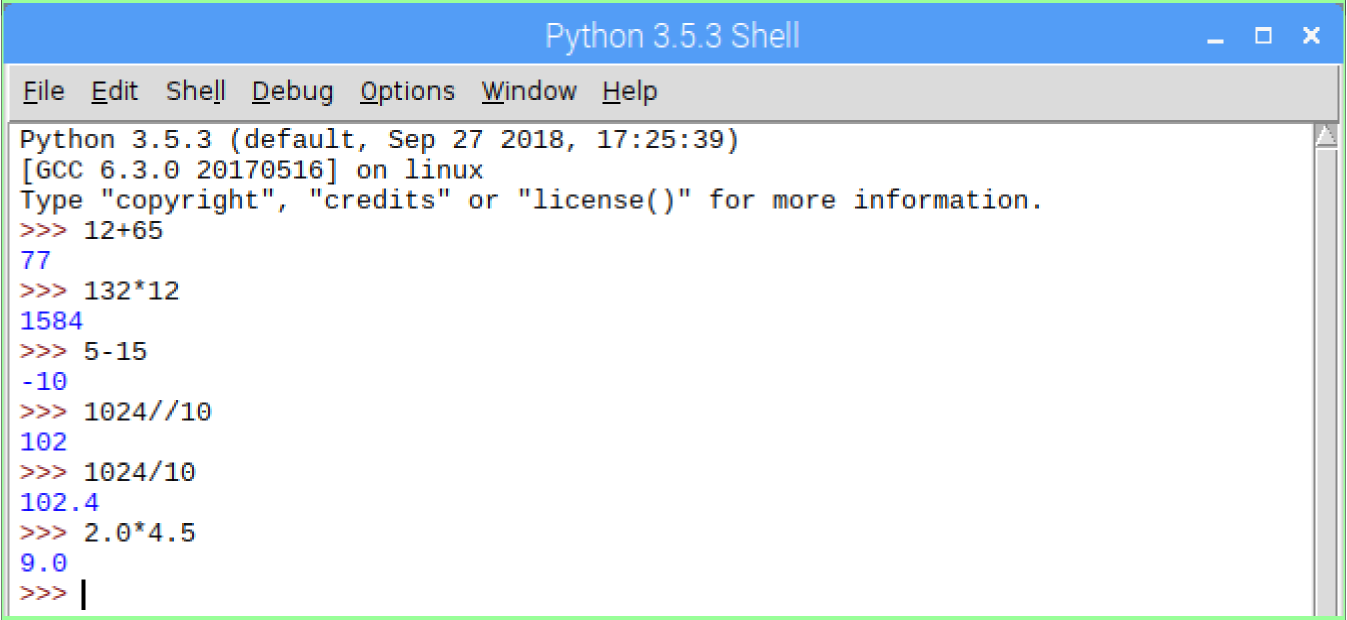
Simple arithmetic expressions
Let's first begin with some simple arithmetic expressions to see how the Python shell can evaluate them and provide real time results. Here's an example of Python evaluating a simple expression (12 + 65) and providing the result in IDLE:



```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 12+65
77
>>> 
```

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

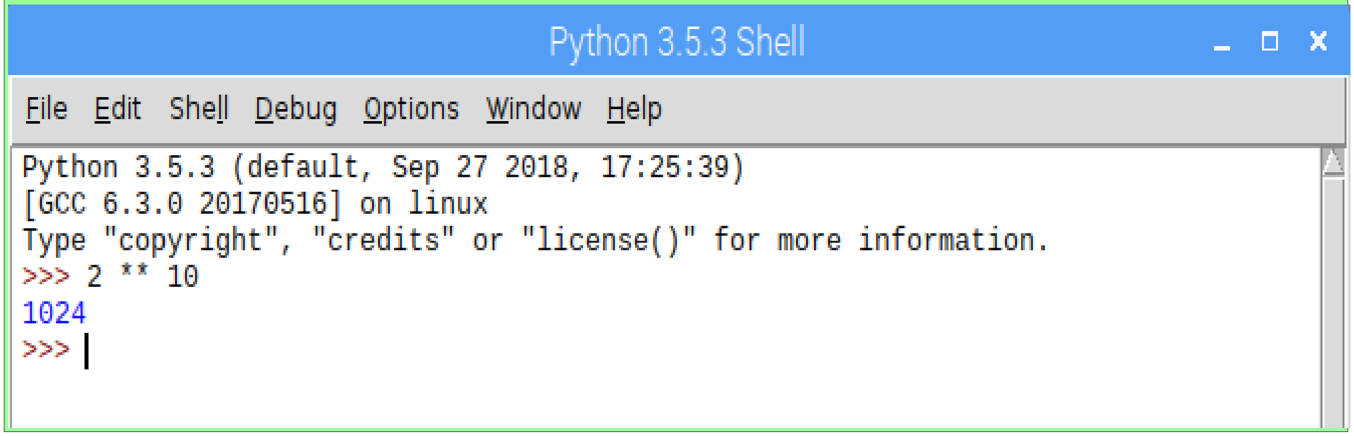
Python evaluates the expression, 12 + 65, and provides the result in the Python shell. Here are more examples:



```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 12+65
77
>>> 132*12
1584
>>> 5-15
-10
>>> 1024//10
102
>>> 1024/10
102.4
>>> 2.0*4.5
9.0
>>> 
```

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

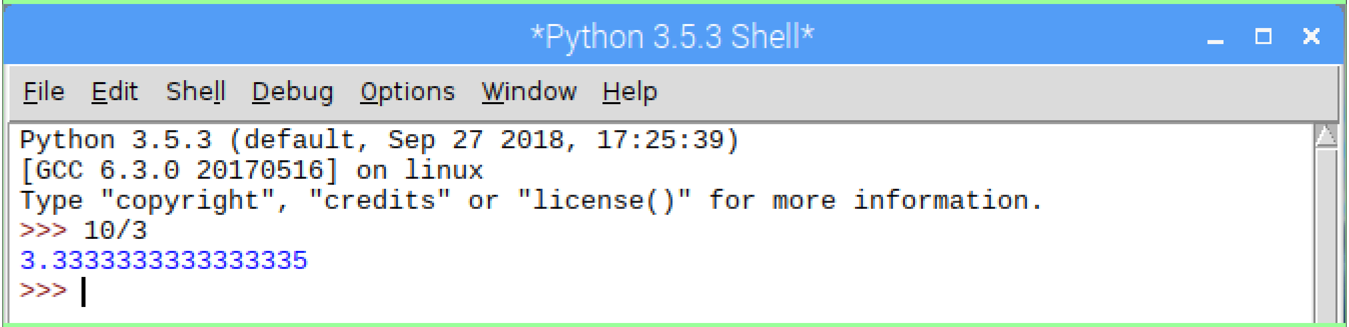
So far, you have seen the four main arithmetic operators (i.e., +, -, *, and /). Take a look at this expression:



```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 2 ** 10
1024
>>> 
```

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.

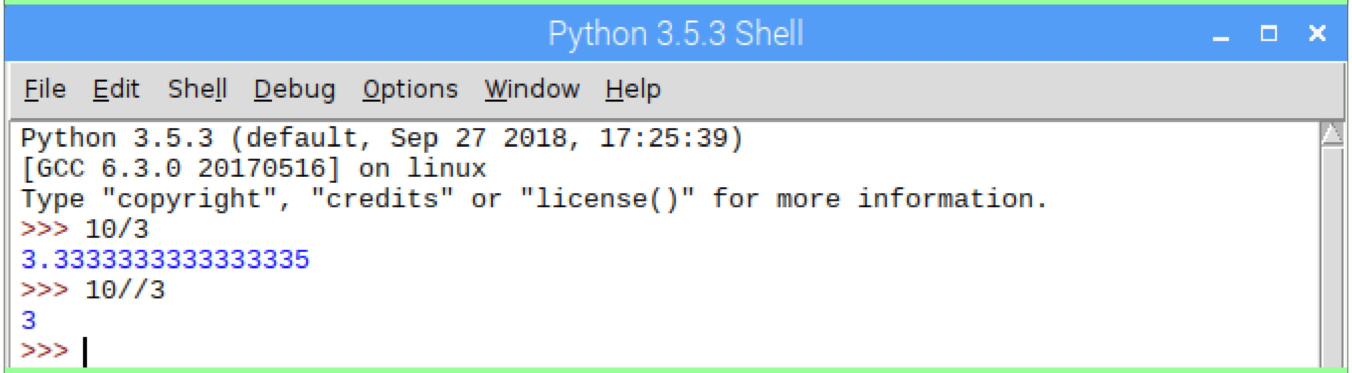
Now, take a look at the following expressions:



```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 10/3
3.3333333333333335
>>> 
```

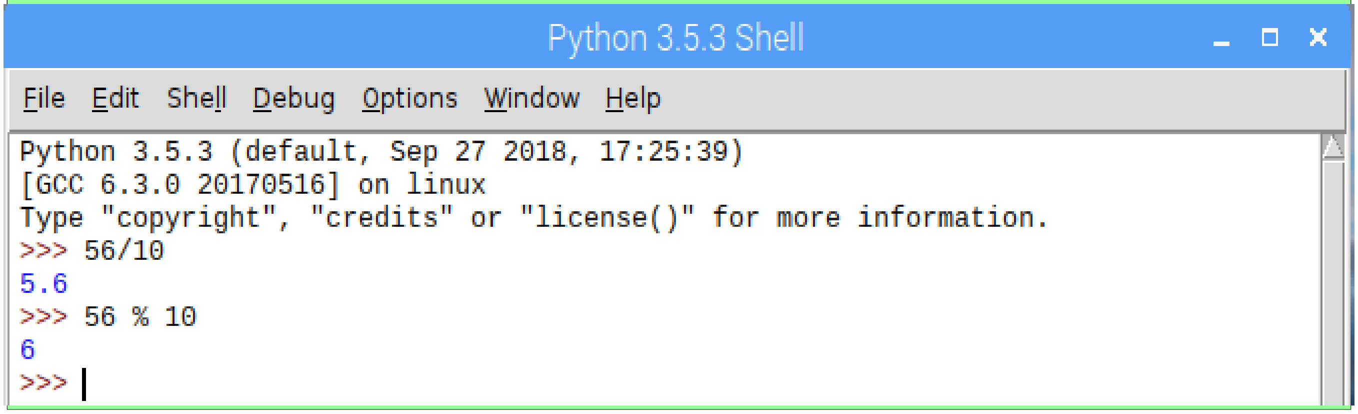
Note how the expression 10 / 3 results in 3. The decimal portion of the result (which we calculate to precisely be 3.33333...) seems to be truncated. In fact, the / operator in Python 3.x returns integer division if the two operands the operator is being applied to are both integers. That is, it returns 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 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 10/3
3.3333333333333335
>>> 10//3
3
>>> 
```

There is one more arithmetic operator in Python: the % operator. This operator returns the remainder of a division (similar to the **mod** operator in Scratch). Take a look at the following example:

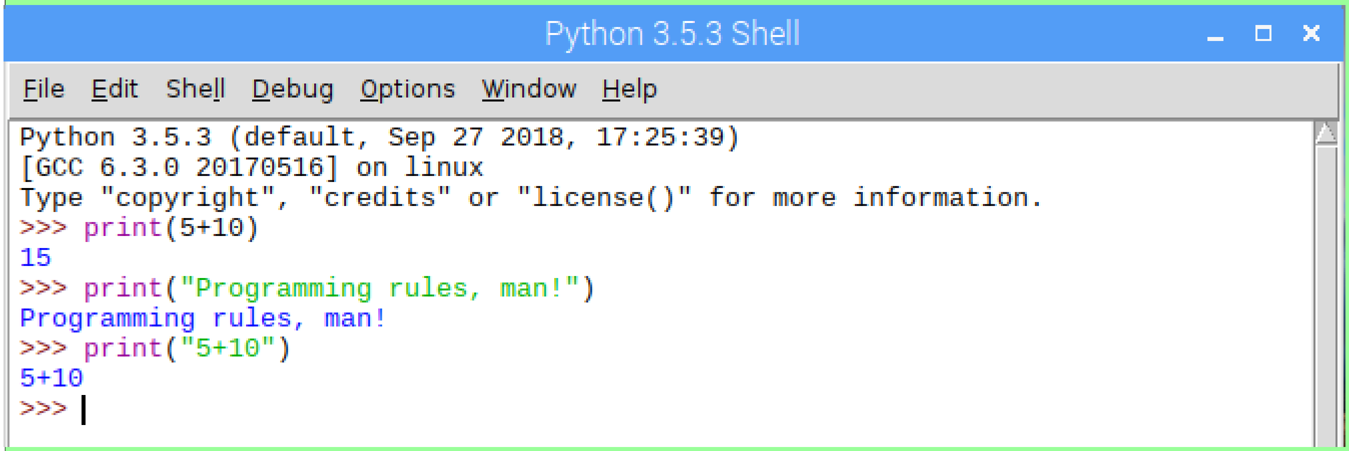


```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 56/10
5.6
>>> 56 % 10
6
>>> 
```

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

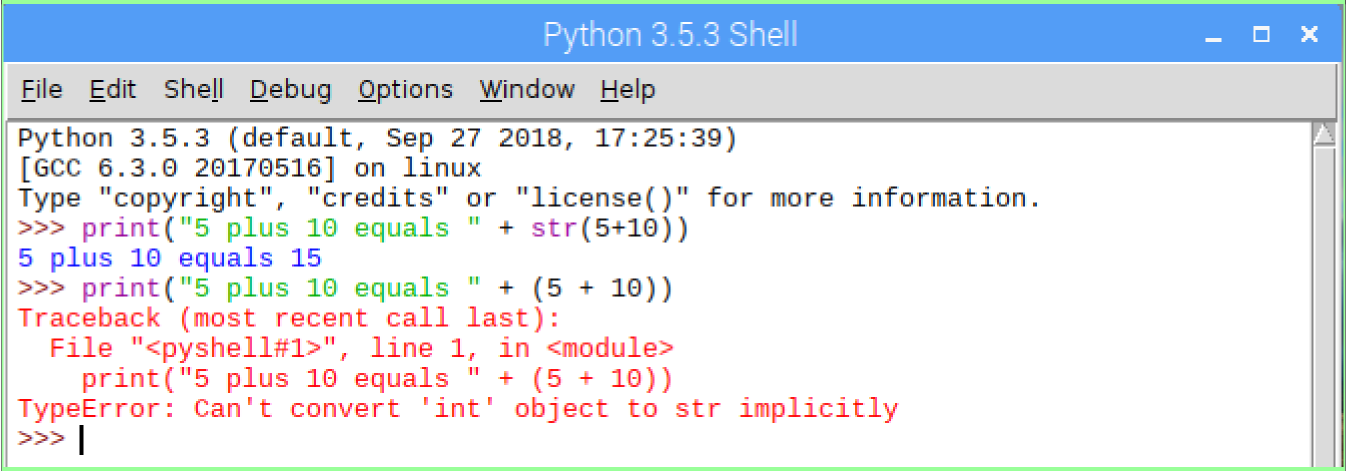
As seen earlier, Python allows output via a **print** statement. Here are some simple examples:



```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> print(5+10)
15
>>> print("Programming rules, man!")
Programming rules, man!
>>> print("5+10")
5+10
>>> 
```

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:



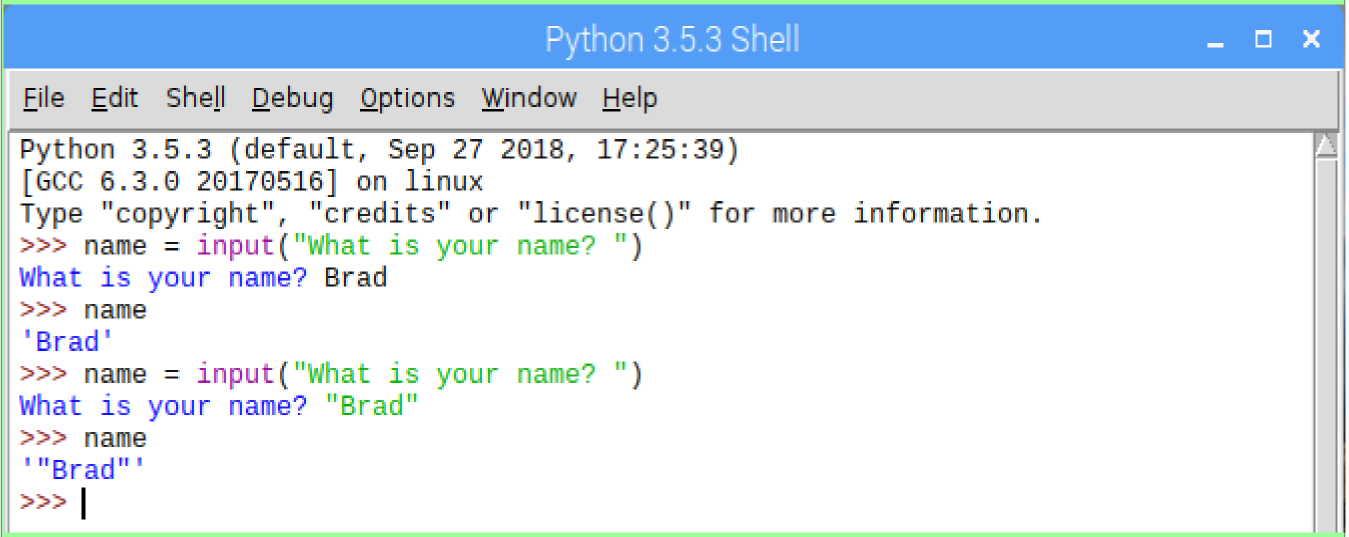
```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> print("5 plus 10 equals " + str(5+10))
5 plus 10 equals 15
>>> print("5 plus 10 equals " + (5 + 10))
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print("5 plus 10 equals " + (5 + 10))
TypeError: Can't convert 'int' object to str implicitly
>>> 
```

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.

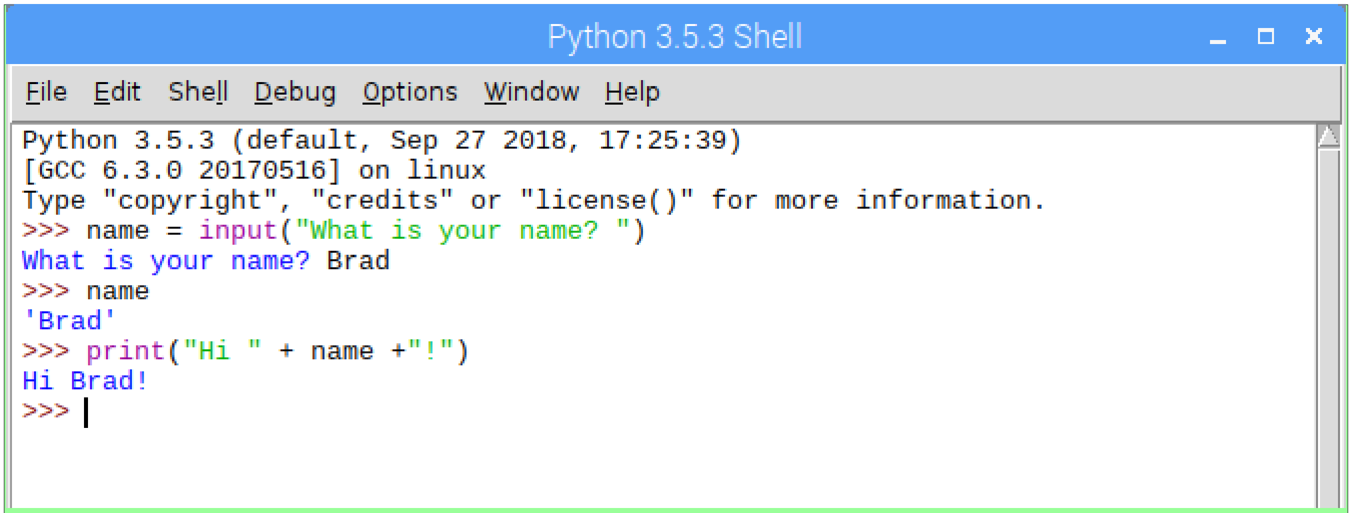
Input

Python also supports statements that allow users to input information via the **input()** function. This information is typically stored in variables. In Python 3.x, the **input()** function returns **string** type only. Take a look at the following example:



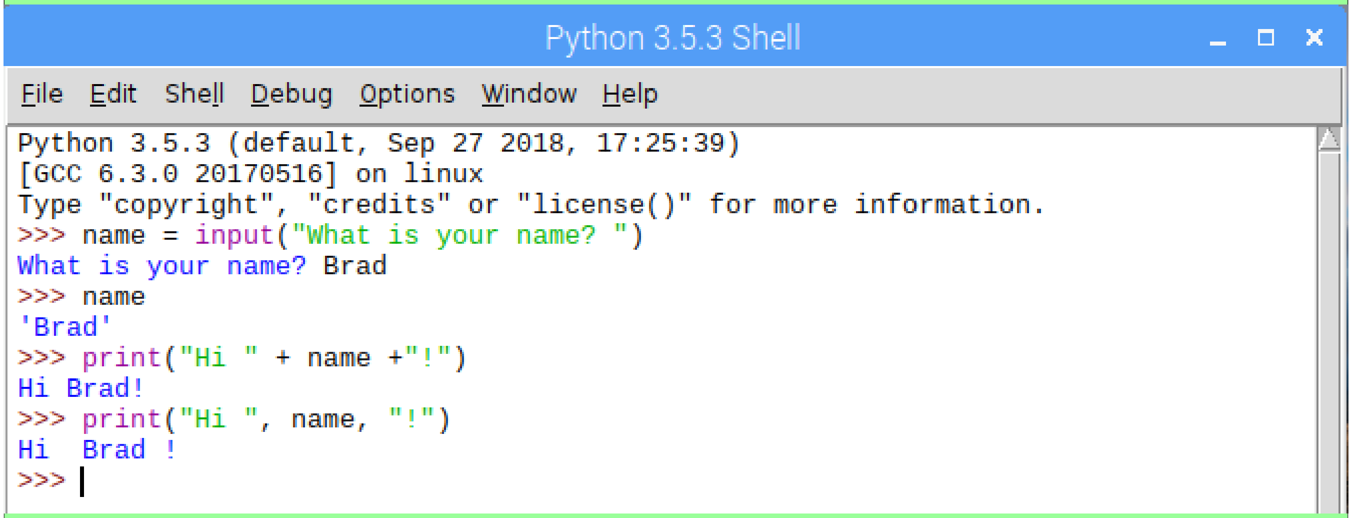
```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad
>>> name
'Brad'
>>> name = input("What is your name? ")
What is your name? "Brad"
>>> name
'"Brad"'
>>> 
```

The statement, `name = input("What is your name? ")`, prompts the user for a name. It stores the result in the variable *name*. We could have very well entered in 23 as the response, and the variable name would have stored as string `"23"`. But, of course, we wanted to enter an actual name. 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:



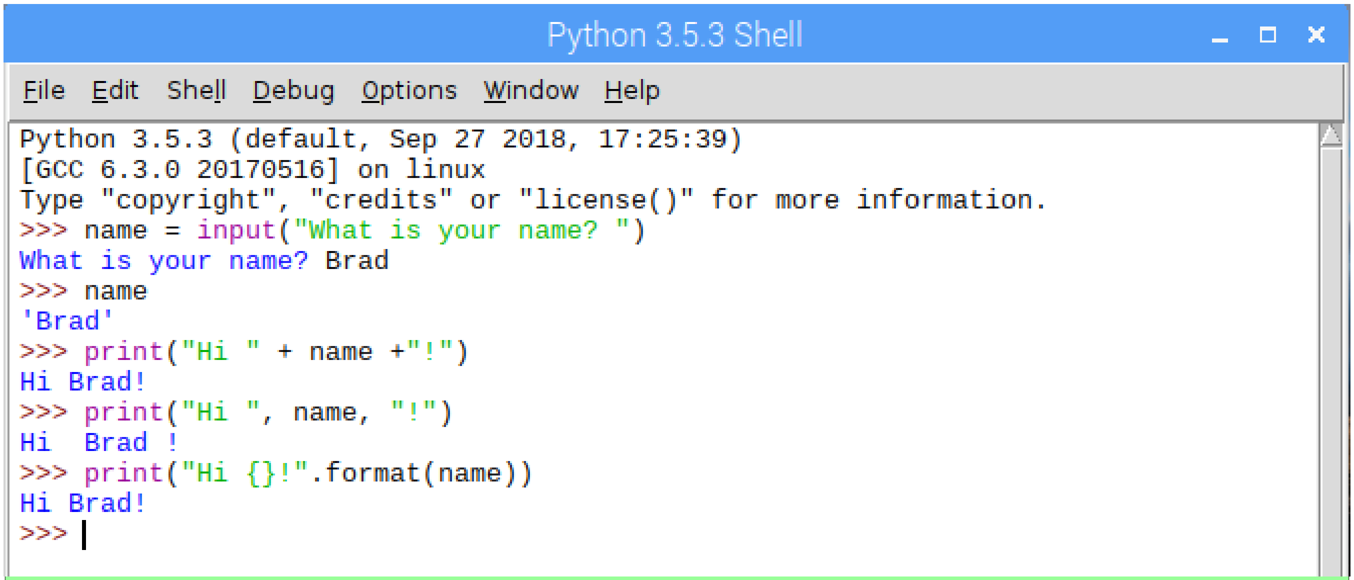
```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad
>>> name
'Brad'
>>> print("Hi " + name +"!")
Hi Brad!
>>> 
```

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



```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad
>>> name
'Brad'
>>> print("Hi " + name +"!")
Hi Brad!
>>> print("Hi ", name, "!")
Hi  Brad !
>>> 
```

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:

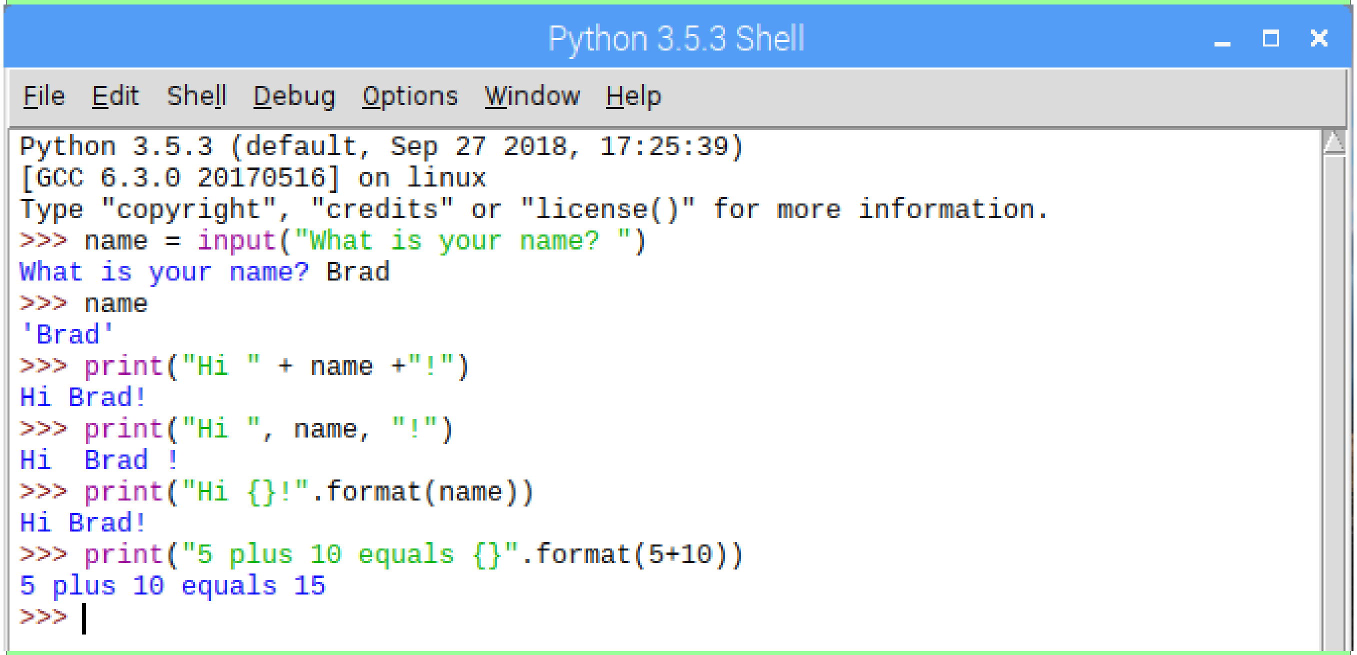


```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name = input("What is your name? ")
What is your name? Brad
>>> name
'Brad'
>>> print("Hi " + name +"!")
Hi Brad!
>>> print("Hi ", name, "!")
Hi  Brad !
>>> print("Hi {}!".format(name))
Hi Brad!
>>> 
```

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!"

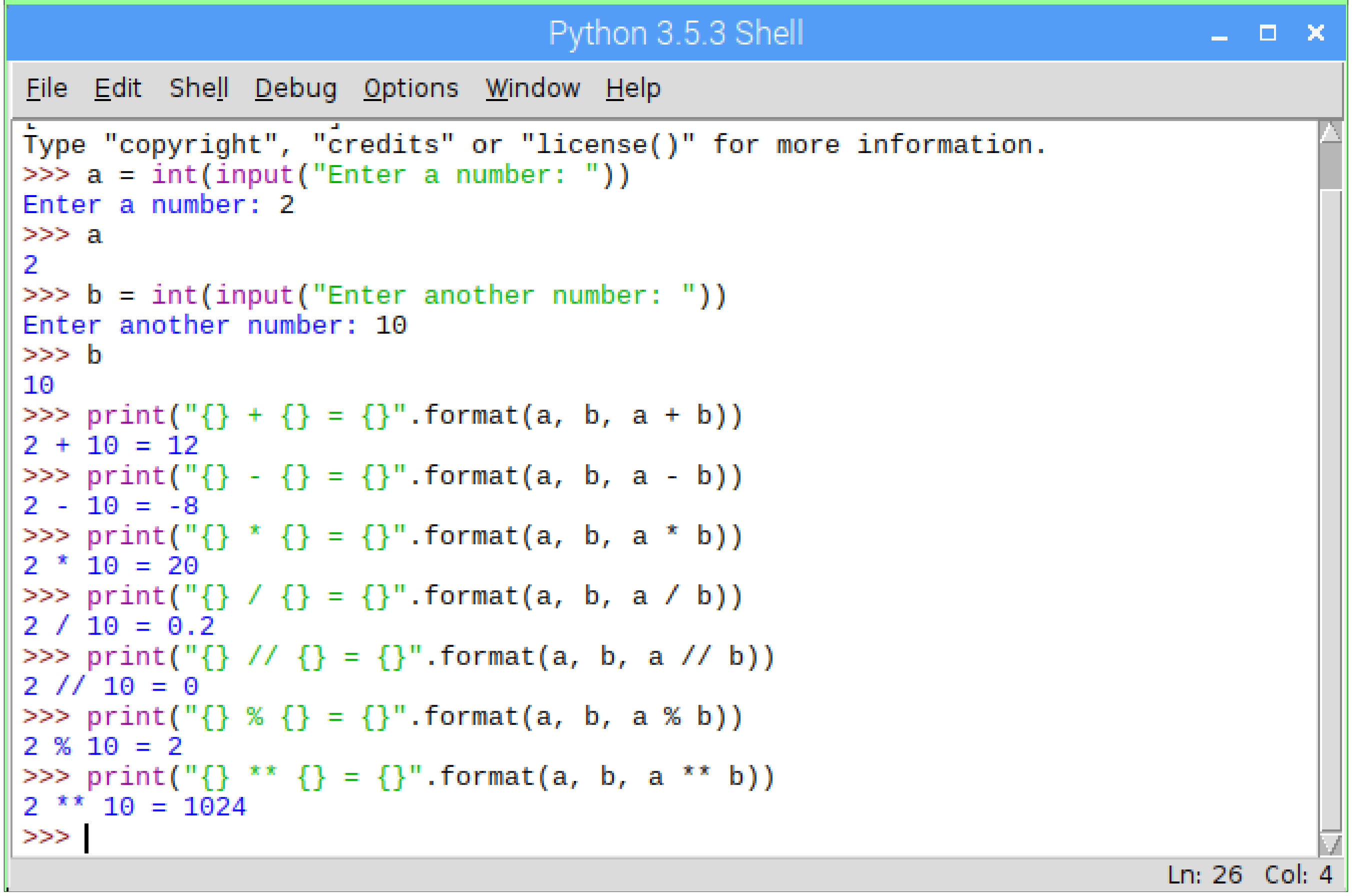
Here's another example of this with the following statement:

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



```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 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
>>> 
```

Here are more examples of this with arithmetic expressions:



```
Type "copyright", "credits" or "license()" for more information.
>>> a = int(input("Enter a number: "))
Enter a number: 2
>>> a
2
>>> b = int(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 = 0.2
>>> 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
>>> 
```

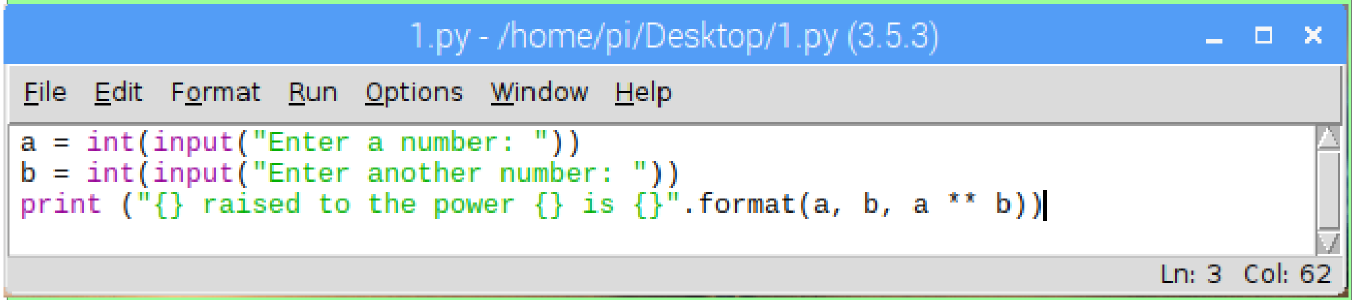
Creating programs and saving files

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:

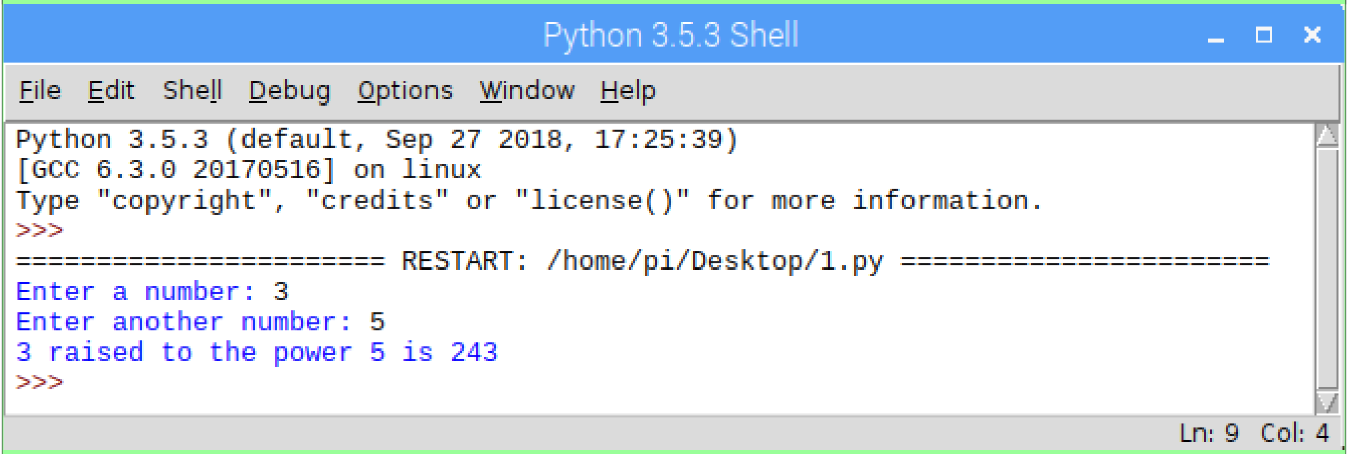
```
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print("{} raised to the power {} is {}".format(a, b, a ** b))
```

This is what you should see at this point:


1.py - /home/pi/Desktop/1.py (3.5.3)
File Edit Format Run Options Window Help
Ln: 3 Col: 62

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

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 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
======================= RESTART: /home/pi/Desktop/1.py =======================
Enter a number: 3
Enter another number: 5
3 raised to the power 5 is 243
>>>
```

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.
```

A possible solution is:

```
a = int(input("Enter a number: "))
b = int(input("Enter another number: "))
print("The quotient of {} divided by {} is {} with a remainder
      of {}.".format(a, b, a / b, a % b))
```

Data types, constants, and variables

Definition: *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.

Definition: *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 in the Living *with* Cyber curriculum.

Numeric types include whole numbers, floating point numbers, and complex numbers. Python has two whole number types: `int` and `long`. The `int` data type is a 64-bit integer. This means that 64 bits (i.e. a bit can be either 0 or 1) is used to represent a single whole number. For example the following series of bits represent a single integer (in fact it represents the number 846,251,337): 0000 0000 0000 0000 0000 0000 0000 0000 0011 0010 0111 0000 1100 0101 0100 1001. We will explore this in more detail at a later time. The `long` data type is an integer of unlimited length. This means Python will give us enough bits to store any number we want! Note that in Python 3.x, an `int` is an integer of unlimited length (there is no `long` data type). These integer types can represent negative or positive whole numbers. The `float` type is a 64-bit floating point (decimal) number. This means it can hold numbers like 3.14 and -90.3324235. 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.

Definition: *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.

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!"

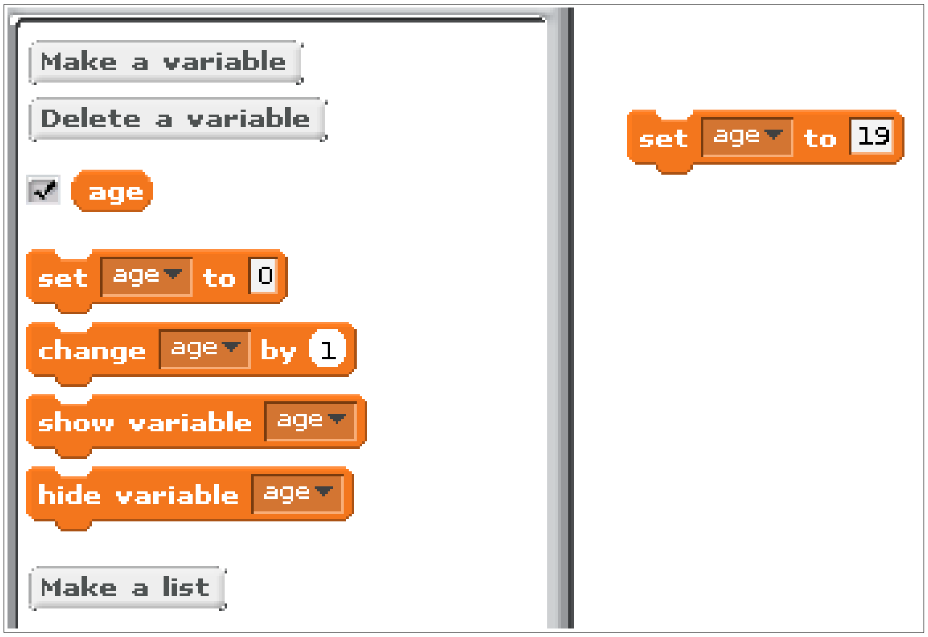
Note that, unlike Scratch, Python requires the quotes surrounding text constants.

Definition: *A **variable** is a named object that can store a value of a particular data type.*

Recall that Scratch supports two types of variables: text variables and numeric variables. Moreover, before a variable can be used, its name must be declared. In many programming languages, both its name and type must be declared; however, both Scratch and Python only require a variable's name to be declared before it is used. Here is an example of declaring and initializing a variable in Python:

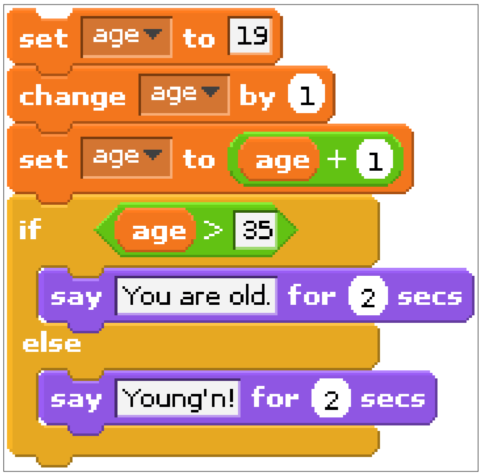
```
age = 19
```

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:

Scratch:



```
set age to 19
change age by 1
set age to (age + 1)
if <age > 35>
    say You are old. for 2 secs
else
    say Young'n! for 2 secs
```

Python (blue text with lack of >>> prompt indicates console output):

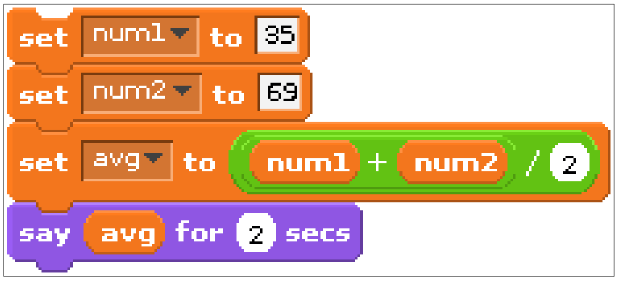
```
>>> age = 19
>>> age = age + 1
>>> age = age + 1
>>> if age > 35:
        print("You are old.")
else:
        print("Young'n!")

Young'n!
>>> age
```

```
    21
```

Another example:

Scratch:



```
set num1 to 35
set num2 to 69
set avg to ((num1 + num2) / 2)
say avg for 2 secs
```

Python:

```
    >>> 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 name. Note that, just as in Scratch, we can assign a value of a different type to a variable. For example:

```
    >>> var = 5
    >>> var
    5
    >>> var = 3.14159
    >>> var
    3.14159
    >>> var = "Pi"
    >>> var
    'Pi'
```

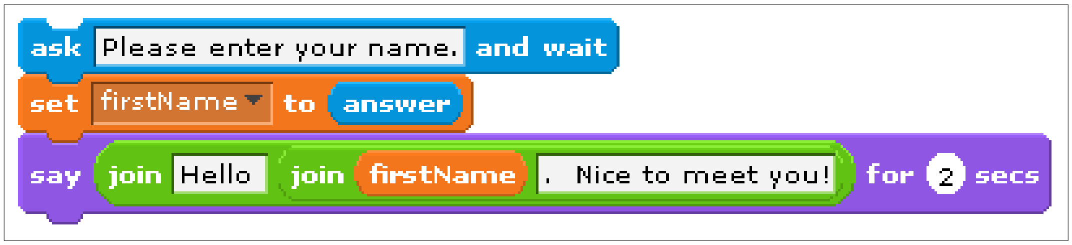
It is important to realize that, while human programmers generally try to give variables names that reflect the use to which they will be put, the variable name itself doesn't mean anything to the computer. For example, the numeric variable `age` can be used to hold any number, not just an age. It is perfectly legal for `age` to hold the number of students in a class or the number of eggs in your refrigerator. The computer couldn't care less. Human programmers, on the other hand, generally care a great deal. They expect a variable's name to accurately reflect its purpose; so while it is possible to do so, it would be considered poor programming practice to use the variable `age` to store anything other than an age.

Input and output

In order for a computer program to perform any useful work, it must be able to communicate with the outside world. The process of communicating with the outside world is known as input/output (or I/O). Most imperative languages include mechanisms for performing other kinds of I/O such as detecting where the mouse is pointing and accessing the contents of a disk drive.

The flexibility and power that input statements give programming languages cannot be overstated. Without them the only way to get a program to change its output would be to modify the program code itself, which is something that a typical user cannot be expected to do.

General-purpose programming languages allow human programmers to construct programs that do amazing things. When attempting to understand what a program does, however, it is vitally important to always keep in mind that the computer does not comprehend the meaning of the character strings it manipulates or the significance of the calculations it performs. Take, for example, the following simple Scratch program:

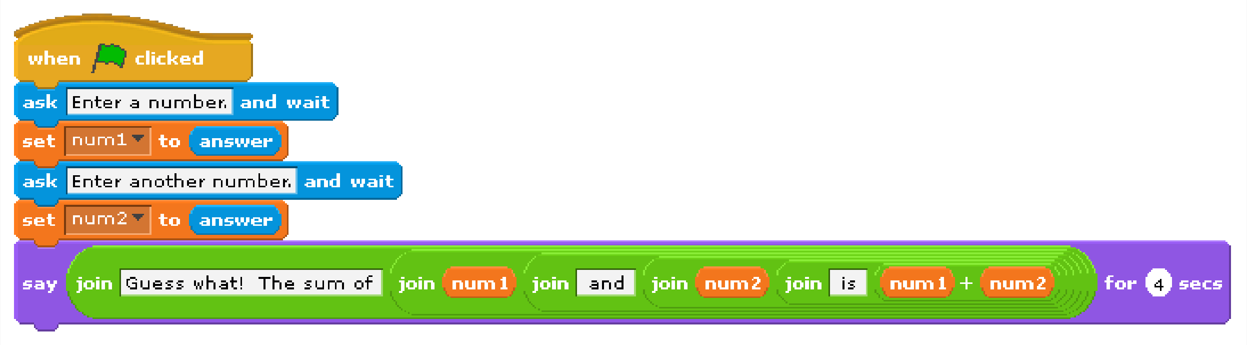


```
ask [Please enter your name: ] and wait
set [firstName] to (answer)
say (join [Hello ] (join (firstName) [, Nice to meet you!])) for (2) secs
```

This program simply displays strings of characters, stores user input, and echoes that input back to the screen along with some additional character strings. The computer has no clue what the text string "Please enter your name: " means. For all it cares, the string could have been "My hovercraft is full of eels." or "qwerty uiop asdf ghjkl;" (or any other text string for that matter). Its only concern is to copy the characters of the text string onto the display screen.

Only in the minds of human beings do the sequence of characters "Please enter your name: " take on meaning. If this seems odd, try to remember that comprehension does not even occur in the minds of all humans, only those who are capable of reading and understanding written English. A four year old, for example, would not know how to respond to this prompt because he or she would be unable to read it. This is so despite the fact that if you were to ask the child his or her name, he or she could immediately respond and perhaps even type it out on the keyboard for you.

Now consider this Scratch program:



```
when green flag clicked
ask [Enter a number: ] and wait
set [num1] to (answer)
ask [Enter another number: ] and wait
set [num2] to (answer)
say (join [Guess what!  The sum of ] (join (num1) (join [ and ] (join (num2) (join [ is ] ((num1) + (num2))))))) for (4) secs
```

Here, the input is numeric instead of text. The program prompts the user for two numbers, which it then computes the sum for and displays to the user. Note that two variables were declared: `num1` and `num2`. The first number is captured and stored in the variable `num1`. The second number is captured and stored in the variable `num2`. What do you think would happen if the user did not provide numeric input and, for example, inputted "Bob" for the first number? In the *real world*, programmers must create robust programs that examine user input in order to verify that it is of the proper type before processing that input. If the input is found to be in error, the program must take appropriate corrective action, such as rejecting the invalid input and requesting the user try again.

In Python, output is implemented as a **print** statement: `print("This is some output!").` 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:

```
>>> age = int(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 (in Python 3.x its a text by default. Use casting to convert into numeric).

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.

Definition: *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. What do you think would happen if the operands are of two different types (e.g., numeric and string)? Let's see:

```
>>> print(1 + "one")

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(1 + "one")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> print("one" + 1)
```

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print("one" + 1)
TypeError: cannot concatenate 'str' and 'int' objects
>>> print("one" + str(1))
one1
```

As expected, Python doesn't know what it means to “add” a numeric type to a string type. Therefore, it results in an error: unsupported operand type(s). To “add” a string type to a numeric type, we must convert the numeric type to a string type via `str`. Then, Python understands that “adding” actually means concatenating two strings. It is interesting to note how Python handles “multiplying” a string type and a numeric type like this:

```
print("hello" * 5)
```

What do you suppose the output will be? Would you expect an error? Or would you expect something like this:

```
hellohellohellohellohello
```

It turns out that Python understands how to “multiply” both types by interpreting the `*` operator as concatenating a string type a number of times specified by a numeric type. Therefore, the statement `print(10 * "hello")` concatenates the string type "hello" 10 times!

Python has many different operators that perform a variety of operations on operands. These will be discussed later in this lesson.

Subprograms

Definition: *A **subprogram** is a block or segment of organized, reusable, and related statements that perform some action.*

It is essentially a program within a program. Recall an earlier lesson on representing algorithms as to-do lists. One algorithm represented the steps necessary to *get to class*. One of those steps was *eat breakfast*. We noted how we could zoom in to that step and identify the sub-steps necessary to complete the *eat breakfast* step. Control flow shifted from the main to-do list to the *eat breakfast* to-do list when the *eat breakfast* step was encountered, and then returned to the main to-do list at the point where it left earlier. We can consider the *eat breakfast* to-do list as a subprogram.

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.

Subprograms make the construction of software libraries possible.

Definition: *A **software library** (or just **library**) is a collection of subprograms, or routines as they are sometimes called, for solving common problems that have been written, tested, and debugged.*

Most programming languages come with extensive libraries for performing mathematical and text string operations and for building graphical user interfaces. These languages allow programmers to include library routines in their code. Using subprograms from the library speeds up the software development process and results in a more reliable finished product.

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

Subprograms can call other subprograms (including copies of themselves as we will see later). 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.

Definition: *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. As you will see later, an **identifier** is a name used to identify a variable, function, or other object (note that objects will be discussed later). The function name must be followed by a set of parentheses containing optional parameters.

Definition: ***Parameters** allow for values (constants, variables, expressions, and so on) to be passed in to a function. They form the input needed for the function to execute properly.*

For example, a function may accept two values, calculate their average, and return the result to the caller. The function **definition** (or **header**) 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:

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


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

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. The body of a function describes what the function does. This is the code within the function. 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:

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

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

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
```

Recall that `x ** y` in Python implies exponentiation and means *x* raised to the power *y*. Formally, we call `**` the exponentiation operator. Operators will be discussed in detail in the next section.

Here's sample output of this function with various parameters:

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

>>> pow(2, 8)
256
>>> pow(10, 4)
10000
>>> pow(3, 5)
243
```

Operators

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. Operators allow operations to be performed on operands. Let's first take a look at the arithmetic operators since they relate directly to assignment.

Definition: *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 = 1.3529411764705883	c / d = 0.5
%	modulus	a % b = 6	c % d = 4.0
**	exponentiation	a ** b = 141050039560662968926103	c ** d = 65536.0
//	floor division	a // b = 1	c // d = 0.0

In Python 2.x, 64-bit integers are of type `int`, and unlimited length integers are of type `long`. Integers in Python 3.x are of unlimited size. There is no integer of type `long` in Python 3 anymore. Here is output of the examples in the previous table using the variables c and d in IDLE:

```
>>> 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
```

Definition: *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		
==	equality	a == b is False
!=	inequality	a != b is True
<>	inequality	a <> b is True
>	greater than	a > b is True
<	less than	a < b is False
>=	greater than or equal to	a >= b is True
<=	less than or equal to	a <= b 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`. Also note that the use of <> is mostly discouraged as this is an old style way of checking for inequality. It is therefore recommended to use `!=` instead. Here is output of the examples in the previous table in IDLE:

```
>>> 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.

Definition: *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		
=	a = b assigns b to a	a = 17
+=	a += b increments a by b (same as a = a + b)	a = 40.0
–=	a –= b decrements a by b (same as a = a – b)	a = 6.0
*=	a *= b multiplies a by b and stores the result in a (same as a = a * b)	a = 391.0
/=	a /= b divides a by b and stores the result in a (same as a = a / b)	a = 1.3529411764705883
%=	a %= b divides a by b and stores the remainder in a (same as a = a % b)	a = 6.0
**=	a **= b raises a to the power b and stores the result in a (same as a = a ** b)	a = 1.4105003956066297e+23
//=	a //= b divides a by b and stores the floor of the result to a (same as a = a // b)	a = 1.0

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

```
>>> 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
```

Identifiers and reserved words

Definition: *An **identifier** is a name used to identify a variable, function, or other object (note that objects 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_
a1b2c3X7Y9Z0
```

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:

```
>>> average = 100
>>> Average = 50
>>> print(average)
100
>>> print(Average)
50
```

Definition: ***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 3.x, for example, has the following reserved words:

`and`	`except`	`lambda`	`with`
`as`	`finally`	`nonlocal`	`while`
`assert`	`False`	`None`	`yield`
`break`	`for`	`not`	
`class`	`from`	`or`	
`continue`	`global`	`pass`	
`def`	`if`	`raise`	
`del`	`import`	`return`	
`elif`	`in`	`True`	
`else`	`is`	`try`	

You are already familiar with some of these: `def`, `print,` and `return`. Many of the Python reserved words will be discussed later.

Comments

It is often useful to provide informative text in our programs.

Definition: *A **comment** is text that is not interpreted nor 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.*

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 comment 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 or 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: 2020-09-08
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: 2020-09-08
# 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:

```
>>> 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: 2020-09-08
# Description: This program is nothing but fluff.
#################################################
```

Primary control constructs

What makes computers more than simple calculating devices is their ability to respond in different ways to different situations. In order to create programs capable of solving more complex tasks we need to examine how the basic instructions we have studied can be organized into higher-level constructs. Recall that the vast majority of imperative programming languages support three types of control constructs which are used to group individual statements together and specify the conditions under which they will be executed. Again, these control constructs are: sequence, selection, and repetition.

Recall from a previous RPi activity that **sequence** requires that the individual statements of a program be executed one after another, in the order that they appear in the program. Sequence is defined implicitly by the physical order of the statements. It does not require an explicit program structure. This is related to our previous discussion on **control flow**.

Definition: ***Selection*** *constructs contain one or more blocks of statements and specify the conditions under which the blocks should be executed.*

Basically, selection allows a human programmer to include within a program one or more blocks of *optional* code along with some tests that the program can use to determine which one of the blocks to perform. Selection allows imperative programs to choose which particular set of actions to perform, based on the conditions that exist at the time the construct is encountered during program execution.

Definition: ***Repetition*** *constructs contain exactly one block of statements together with a mechanism for repeating the statements within the block some number of times.*

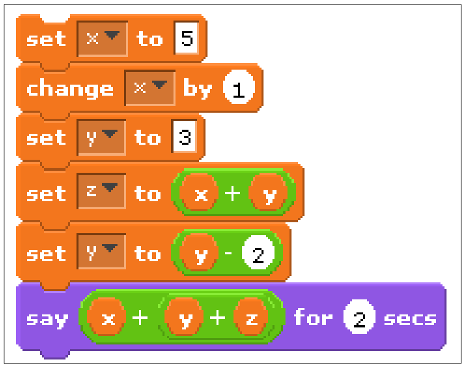
There are two major types of repetition: iteration and recursion. **Iteration**, which is usually implemented directly in a programming language as an explicit program structure, often involves repeating a block of statements either (1) while some condition is true or (2) some fixed number of times. **Recursion** involves a subprogram (e.g., a function) that makes reference to itself. As with sequence, recursion does not normally have an explicit program construct associated with it.

Sequence

Sequence is the most basic control construct. It is the *glue* that holds the individual statements of a program together. Yet, when students who are new to programming try to understand how a particular program works, they often just glance over the various statements making up the program to get a *feel* for what it does rather than methodically tracing through the sequence of actions it performs. One reason such an approach is tempting is because students tend to believe they can figure out what a program is *supposed* to do based on contextual clues such as the meaning of variable names and character strings.

While it is often possible to gain a superficial knowledge of a program simply by reading it, this approach will not give you the kind of detailed understanding that is frequently required to accurately predict a program's output. Being able to carefully trace through a program to determine exactly what it does is an important skill. Failure to carefully follow the sequence of instructions often leads to confusion when trying to understand the behavior of a program.

The following Scratch program illustrates the importance of sequence. It contains a little *do nothing* program that displays the value 16. What makes this program interesting is not so much what its output is, as the way in which that output is computed. Without carefully tracing through the program, one statement at a time, it would be difficult to correctly predict the final output generated by the program.



```
set x to 5
change x by 1
set y to 3
set z to (x + y)
set y to (y - 2)
say (x + (y + z)) for 2 secs
```

Note that the variables *x*, *y*, and *z* were declared in the variables blocks group. The following illustrates the state of the program's memory after executing each line of code. After performing the first declaration, the program knows only about the variable *x*. After the second declaration, it knows of *x* and *y*, and after the third, it knows of *x*, *y*, and *z*. Since these variables have not yet been assigned values, their values are considered to be undefined at this point.

declaring *x*	*x*					
declaring *y*	*x*		*y*			
declaring *z*	*x*		*y*		*z*	
set *x* to 5	*x*	5	*y*		*z*	
change *x* by 1	*x*	6	*y*		*z*	
set *y* to 3	*x*	6	*y*	3	*z*	
set *z* to *x* + *y*	*x*	6	*y*	3	*z*	9
set *y* to *y* − 2	*x*	6	*y*	1	*z*	9
say *x* + *y* + *z* for 2 secs	*x*	6	*y*	1	*z*	9

Program output: `16`

In Python, sequence is implemented in a manner very similar to Scratch: we simply place statements in the order that we wish them to be executed. Here's the program above in Python:

```
x = 5
x += 1
y = 3
z = x + y
y -= 2
print(x + y + z)
```

Formally, sequence can be defined as follows:

Definition: ***Sequence*** *represents the natural order of statement execution. It represents a top-down, sequential order where lines of code are evaluated one after the other.*

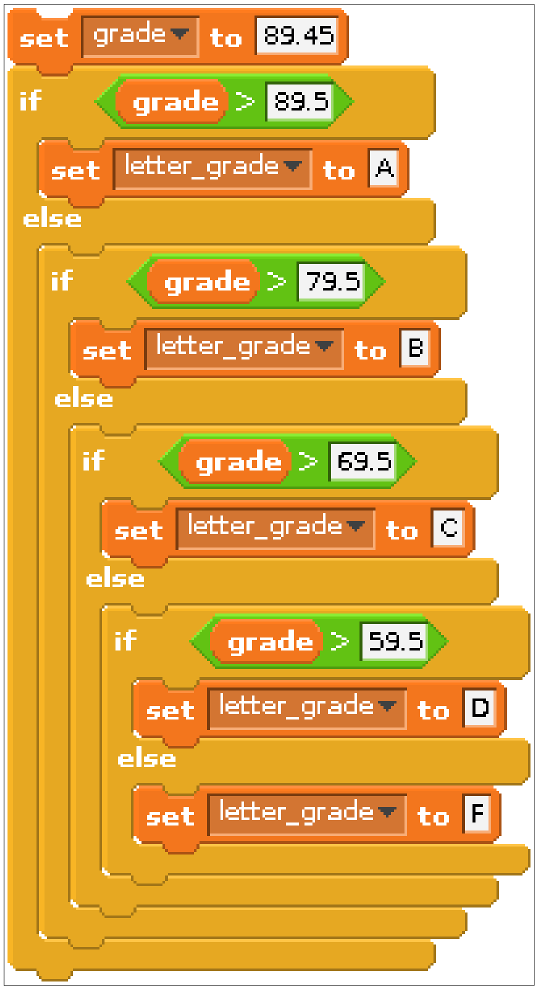
Selection

Selection statements give imperative languages the ability to make choices based on the results of certain condition tests. These condition tests take the form of **Boolean expressions**, which are expressions that evaluate to either *true* or *false*. As discussed earlier, there are various types of Boolean expressions, but most of the time condition tests are based on relational operators. Recall that **relational operators** compare two expressions of like type to determine whether their values satisfy some criterion. Selection statements use the results of Boolean expressions to choose which sequence of actions to perform next. The general form of all Boolean expressions:

expression relational_operator expression

Both Scratch and Python support two different selection statements: ***if-else*** and ***if***. An *if-else* statement allows a program to make a two-way choice based on the result of a Boolean expression. *If-else* statements specify a Boolean expression and two separate blocks of code: one that is to be executed if the expression is *true*, the other to be executed if the expression is *false*. Recall that, in Scratch, **selection** constructs contain one or more blocks of statements and specify the conditions under which the blocks should be executed.

Here's an example:



```
set grade to 89.45
if grade > 89.5
    set letter_grade to A
else
    if grade > 79.5
        set letter_grade to B
    else
        if grade > 69.5
            set letter_grade to C
        else
            if grade > 59.5
                set letter_grade to D
            else
                set letter_grade to F
```

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:

```
>>> 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'
```

Note the structure of an *if-else* statement in Python:

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

Also note the colons after the condition and the **else** reserved word, and the indentation of both the true and false parts. Both are vital in indicating where the *true* and *false* sections of the *if-else* statement begin and end!

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.

The *if* statement is similar to the *if-else* statement except that it does not include an *else* block. That is, it only specifies what to do if the Boolean expression is *true*.

The structure of an *if* statement in Python is:

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

Again, note the colon and indentation. As in the *if-else* statement, both are vital in indicating where the *true* section of the *if* statement begins and ends! *If* statements are generally used by programmers to allow their programs to detect and handle conditions that require *special* or *additional* processing. This is in contrast to *if-else* statements, which can be viewed as selecting between two (or more) equal choices.

Note that the condition of *if* and *if-else* statements can be enclosed in parentheses. This is optional; however, it is recommended as it improves readability:

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

For example:

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

Repetition
Repetition is the name given to the class of control constructs that allow computer programs to repeat a task over and over. This is useful, particularly when considering the idea of solving problems by decomposing them into repeatable steps. There are two primary forms of repetition: *iteration* and *recursion*.

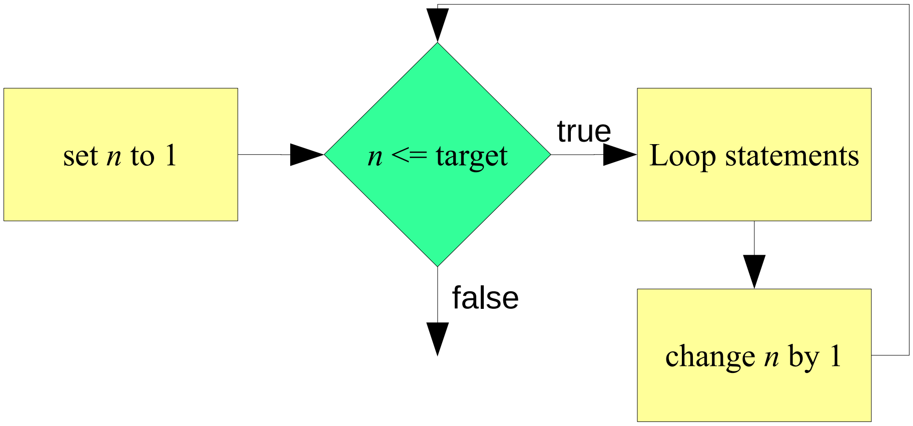
Recall that Scratch supports iteration in two main forms: the *repeat* loop and the *forever* loop. The *repeat* loop has two forms: *repeat-until* and *repeat-n* (where *n* is some fixed or known number of times). The *repeat-until* loop is condition-based; that is, it executes the statements of the loop until a condition becomes true. The *repeat-n* loop is count-based; that is, it executes the statements of the loop *n* times.

In a *repeat-until* loop, the Boolean expression is first evaluated. If it evaluates to false, the loop statements are executed; otherwise, the loop halts. Here's an example in pseudocode:

```
    total ← 0
    repeat
        num ← prompt for a positive number (or -1 to quit)
        if num > 0
        then
            total ← total + num
        end
    until num < 0
    display total
```

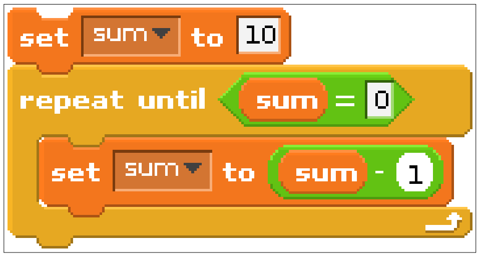
This program asks the user to enter a positive number or -1. If a positive number is entered, it is added to a running total. If -1 is entered, the program displays the total and halts. The *repeat-until* loop is used here to repeat the process of asking the user for input until the value entered is less than 0. It is interesting to note that although the program instructs users to enter -1 to quit, the condition that controls the loop is actually `num < 0` (which will be true for any negative number). Thus, the loop will actually terminate whenever the user enters any number less than zero (e.g., -5).

In Scratch, the *repeat-n* loop executes the loop statements a fixed (or known) number of times. Recall the flowchart for the *repeat-n* loop:



Although the programmer does not have access to a variable that counts the specified number of times (shown as *n* in the figure above), the process works in this manner. A counter is initially set to 1. A Boolean expression is then evaluated that checks to see if that counter is less than or equal to the target value (e.g., 10). If so, the loop statements execute. Once the loop statements have completed, the counter is incremented, and the expression is reevaluated.

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



```
set sum to 10
repeat until sum = 0
    set sum to sum - 1
```

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
```

As in if and if-else statements, the condition may be enclosed in parentheses. It is recommended to do so for readability:

```
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:

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

>>> sum
0
```

There is a slight difference between the condition in Scratch's repeat-until loop and the condition in Python's while loop: the condition in the while loop needs to be true to remain in the loop; the loop is terminated whenever the condition evaluates to false. Conversely, the condition in the repeat-until loop should be false to remain in the loop. The repeat-until loop is terminated whenever the condition evaluates to true.

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

```
>>> counter = 0
>>> sum = 0
>>> sum
0
>>> while (counter < 10):
        sum += 2
        counter += 1

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

Before leaving the topic of iteration, we should say a few words about the idea of *nested* loops. Two loops are nested when one loop appears within the body of another loop. This is quite common, and in fact may be carried out to an arbitrary depth. However, to keep the logic of a program from becoming too hard to follow, programmers try to limit nesting depths to no more than three or four levels deep.

The following Python program displays the multiplication table. This program consists of two nested *repeat-until* loops. The loop variable of the outer loop is `i`, and the loop variable for the inner loop is `j`. Both of these loops count from one to nine:

```
i = 0
while (i < 9):
    i += 1
    j = 0
    while (j < 9):
        j += 1
        print(str(i) + " * " + str(j) + " = " + str(i * j))
```

The program's output is of the form `i * j = x`, where `i` and `j` represent the values of the respective variables, and *x* is their product. Notice that `j` runs through its entire range, from 1 to 9, before `i` is incremented by 1. This behavior is easy to understand when you think about the structure of the program.

Let's look at the outer loop. What does it do? Well, first `i` is initialized to 0, it is then compared to 9, and since it is not equal, the first iteration of the loop commences. The first statement of the loop increments `i` and then initializes `j` to 0. The next statement is another *repeat-until* loop. In order for the first iteration of the outer loop to complete, the program must execute this inner loop to completion.

The process repeats until all 81 entries in the multiplication table, from 1 × 1 to 9 × 9, are computed and displayed.

We conclude this section with the following Python program:

```
bottles = 99
while (bottles > 0):
    print(str(bottles) + " bottles of beer on the wall.")
    print(str(bottles) + " bottles of beer.")
    print("Take one down, pass it around.")
    bottles -= 1
    print(str(bottles) + " bottles of beer on the wall.")
    print()
```

This program displays the lyrics to the song, "99 Bottles of Beer on the Wall." As you most likely know, the song begins as follows:

```
99 bottles of beer on the wall.
99 bottles of beer.
Take one down, pass it around.
98 bottles of beer on the wall.

98 bottles of beer on the wall.
98 bottles of beer.
Take one down, pass it around.
97 bottles of beer on the wall.
```

It continues in this manner, with one less bottle in each verse, until it finally runs out of beer. An interesting feature of the program is that it decrements *bottles* in the middle of the loop rather than at the end. You should trace through the program with a few bottles to convince yourself that it does work properly. One thing you will probably notice as you do so is that when the program gets down to one beer, it reports that as "1 bottles of beer on the wall." While this lack of grammatical correctness might not seem like such a big deal, especially after 98 beers, we can correct it easily with an *if* statement.

Recursion

Definition: ***Recursion*** *is a type of repetition that is implemented when a subprogram calls itself.*

When a recursive call takes place, control is passed to what appears to be a brand new copy of the subprogram. This copy of the subprogram may, in turn, call another copy of the subprogram. That copy may call another copy, and so on. Eventually, these recursive calls must terminate and return control to the original calling program.

Take the "99 Bottles of Beer on the Wall" program above. It illustrated an iterative method of singing the song. Here's an example of the same program in Python, implemented using recursion:

```
def consumeBeers(bottles):
    if (bottles > 0):
        print(str(bottles) + " bottles of beer on the wall.")
        print(str(bottles) + " bottles of beer.")
        print("Take one down, pass it around.")
        print(str(bottles - 1)+ " bottles of beer on the wall.")
        print()
        consumeBeers(bottles - 1)
consumeBeers(99)
```

At first glance you might think that this program is nearly identical to the program shown earlier. There are, however, two major differences between the two. First, instead of a *repeat-until* loop, this program has an *if* statement. Also, just before the end of the *if* statement, the same subprogram (*consumeBeers*) is called. This is the recursive call!

Note that the value of the variable `bottles` is decremented by 1 at the recursive call. When control enters the subprogram, the value is checked to see if it is greater than 0. This ensures that, so long as `bottles` is decremented each time the subprogram executes, it will eventually reach 0. When this occurs, it will cause the Boolean expression in the *if* statement to evaluate to false, thereby not executing its block (and calling itself again) and stopping the recursion.

One way to envision recursion is to think of it as a spiral. Each time a subprogram calls itself, we descend down a level of the spiral until we eventually reach the bottom. At that point, execution begins to *unwind* as the subprogram calls complete and we retrace our path back up through the various levels until finally arriving at the *top* level where execution began.

The recursive program above illustrates what is called *tail recursion*, because the recursive call is the last action taken by the subprogram. In tail recursion, there is no work to be done during the *unwinding* process because it was all done on the way *down* the spiral. In Scratch, this is the only way of implementing recursion. Python permits other forms of recursion, where the recursive call can occur before other statements.

Many students, upon learning how recursion works, worry that programs that employ this form of repetition might be very inefficient in terms of their utilization of machine resources. After all, you have all of those *copies* of the subprogram hanging around. The good news is that recursion is not nearly as expensive as you probably think. For one thing, only one copy of the actual subprogram code is needed. All that is reproduced during each call is the *execution environment*, the variables and whatnot that are used by that *version* of the subprogram. While it is true that recursion generally involves more overhead than iteration, recursive calls are really no more expensive than any other kind of function call. In fact, some optimizing compilers convert tail recursion into iteration so there is often no additional expense in using that form of recursion at all.

Aside from the efficiency issue, you may be wondering why programming languages would support recursion. After all, whenever the need for repetition arises the programmer could always use one of the iteration constructs. The reasons for supporting both recursion and iteration are the same as those, for example, supporting two types of selection statements (*if* and *if-else*): clarity and convenience. Some problems are simply easier to solve using recursion than iteration. For these types of problems, a recursive solution is often more compact and easier to read than an iterative one.

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 = int(input("Enter a number: "))
12:  num2 = int(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 *num1* = 55 and *num2* = 34?

What if *num1* = 100 and *num2* = 100?

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