Episode 2
Variables, Expressions, and Statements

Computers work by changes to elements held in memory that can be in one of two states. Because one state ("off") can be represented as 0 and the other ("on") by 1, computers work in base two; these units are called binary digits (*bits*). In all modern computers these bits are organized into groups of 8, called *bytes*. Numbers written in base 10 must be converted, from base 10 to base 2 for input, or from base 2 to base 10 for output.

Memory is further grouped into *words*. The number of bits in a word is the basis for calling a computer "32 bits" or "64 bits." All computers since the early 2000s have been 64-bit computers, meaning that each word consists of 64 bits or 8 bytes.

Computer programs are founded around *variables*, but unlike mathematical variables they are just "nicknames" for locations in memory. Furthermore, in nearly all programming languages a single equal sign (=) means assignment, not equality. Thus

```
j=15
```

tells the computer to take the value `15`, as represented in binary, and place it into the memory location represented by the variable named `j`. This means that a mathematically nonsensical statement like

```
a=a+b
```

makes perfect sense to the interpreter; it adds the value of `b` to the value of `a` and stores the result back into the memory location represented by `a`.

Programmers should try to create variable names that are meaningful to humans reading the program; the variable name should in some way connect to what it represents. For example, a variable representing the radius of a circle would be better named `radius` than `x`, whereas the names of variables representing points in space might follow the mathematical convention and be called `x`, `y`, and `z`. In Python variable names must begin with a letter of the alphabet, either upper or lower case, or they may begin with an underscore. However, variable names beginning with underscores have special meaning to the interpreter so we shall ignore them for now. The rest of the variable name may consist of letters, numbers, or underscores. Python makes a distinction between upper and lower case, so `myVariable` and `MyVariable` are different. One character that may not be present in a variable name is a blank space. Because of this, variable names that combine words to be descriptive must link the two words. Python's usual conventional style is to use an underscore, for example is_open, but programmers may also use "camel case", for example isOpen.

Because each computer has a finite number of words and each word has a finite number of bits, the things we can represent have limits. There are rules for different ways to interpret each sequence of bits. For programming we often deal with only a few types, specifically numerical types (integer and floating point) and characters.

Integers, like their mathematical counterparts, are whole numbers with no fractional part. Unlike mathematical integers, only a finite range can be represented in the hardware. The default integer in Python is 32 bits, with one bit used for the sign. However, Python has a special type, the *long integer*, which can have arbitrary size. It is not represented in hardware, however, but in software, so it can be slow. Long integers can be represented with the letter L following the numerals.

Floating-point numbers have a sign, an exponent, and a mantissa. They are similar to numbers written in base-10 scientific notation, but using base 2 instead. In Python the default floating-point number is 64 bits long, which is called *double* or *double precision* in other languages. These numbers can range from about 10^{-308} to 10^{308}. This is a large range but we still can see that the subset of mathematical numbers we can represent is very finite.

A character is a type that represents a letter, a digit, a special symbol like `$`, and some special characters used for formatting, such as tabs. Depending on the language and writing system, a character may occupy one, two, or up to four bytes. The bit sequence is interpreted according to some encoding standard. Python uses a standard called utf-8 (Unicode Transformation Format-8 bit) as its default. This standard can represent nearly all writing systems and is the dominant standard for the World Wide Web. However, Python commands and variables must use only the Roman alphabet.

Each type has *operators* defined on it. Numerical types have the usual mathematical operators of addition (`+`), subtraction (`−`), multiplication (`*`), and division (`/`). Other types have different operators, which may use similar symbols to mathematical operators but do different things.

Watch out in Python 2.7 for division of one integer by another. In Python 2.7, `3/4` is `0`! This is because it is an integer operator that returns an integer result, the whole-number part of the division. To get the remainder use `%`, so `3%4` is `3`. In Python 3 versions, `3/4` is `0.75` as you might expect. In Python 3 you can obtain only the integer part with `//`, i.e. `3//4` is `0` in both versions of Python.

Operators always have a precedence. Higher-precedence operations will be performed before lower-precedence operations. For arithmetic operations, ** (exponentiation) has the highest precedence. Next are * and / with equal precedence, and finally + and - are equal to one another. Parentheses can be used to change this as needed for a program. For example, `4.**2./4.` is `4.0`, but `4.**(2./4.)` is `2.0` (and `4.**(2/4)` is `1.0` in Python 2.7)

We can see the precedence rules working by typing into our iPython console the following lines:

```
11.+3.*12./4.

(11.+3)*12./4.

(11.+3*12)/4.

11.+3/4.*12.
```

```
11.+3./(4.*12.)
```

Python supports operation assignment operators for **, *, /, +, and -. These operators are written **=, *=, /=, +-, and -=. They first perform the indicated operation and then assign the result back to the variable. For example,

```
a=a+b
```

Is equivalent to

```
a+=b
```

Expressions are combinations of variables, operators, and other commands we have not yet learned. It must be possible for the interpreter to arrive at a unique value for the expression, given values for all variables. It will follow its own set of internal rules to do so, so be sure to use parentheses to make things clear if they are needed.

It's important for all variables and other constructs to be defined so that an expression can be evaluated. If we type into the Editor pane

```
R = b**2. - 4.*a*c

print R
```

we'll see the yellow warning triangle telling us that `a`, `b`, and `c` are undefined. If we go ahead and run that selection we'll get an error:

```
NameError: name 'b' is not defined
```

It could no longer go on as soon as it determined that `b` did not have a value. We can add to our script

```
a = 2.

b = 3.

c = 1.
```

and now it runs correctly.

We must remember that we cannot print expressions without a print statement in a script we must run. If we type directly at the console we can rely on expression evaluation

```
In  [5]: 4.*a*c
Out [5]: 8.0
```

However, a line consisting only of 4.*a*c in the script does nothing; it neither prints the value nor stores the result anywhere.

A *statement* is a complete "sentence" of the language. It represents a command the interpreter can execute. One of the most common is an assignment statement, such as

```
area=math.pi*radius**2
```

Another example is the print statement

```
print area
```

If you are writing a script file, typing

```
math.pi*radius**2
```
Without an assignment or a print this will not generate a warning, but it will accomplish nothing. At a direct interpreter prompt, however, it will print the value of the expression.

In Python 2.7 print is a statement, whereas in Python 3, print() is a function. One major difference is that the print function can be used in a more advanced concept called lambda functions. In Python 3 this is valid.
```
output=lambda x: print(x)
output("Hello")
Hello
```
You can use the print function, rather than the print statement, in Python 2.7 if you import it:
```
from __future__ import print_function
```
This statement has two underscores on each side of `future`.

Finally, we learned about comments. Python has two types; for ordinary comments anything from a # to the end of the line is ignored. Everything between triple double quotes is ignored, including line breaks.

```
#This is my comment.
j-=2  #skip j-1

"""
    Everything these triple double quotes is
    treated as a comment.
"""
```

Triple double quotes are most frequently used for documentation strings or *docstrings*. A docstring contains information about the purpose, functionality, and usually author of a script, module, function, or class. Spyder automatically inserts a dummy docstring for each file, which you should fill out with more details about the code.

```
"""
This program computes the day of the week given a date in the
Gregorian calendar.  The user inputs the day, month, and year as
integers.
Author:     A. Programmer
Changelog:  Initial version 2013-05-20
            Bug in months list fixed 2013-05-22
"""
```

So far we have studied simple scalar variables. The next topic will be compound types, which will enable us to develop more sophisticated ways of representing data.