```
type(7)
```

Always remember to end with the Enter key. After the Shell responds, you should see something like

```
>>> type(7)
<class 'int'>
>>>
```

In the result, `int` is short for integer. The work *class* is basically a synonym for type in Python. At the end you see a further prompt where you can enter your next line....

For the rest of this section, at the `>>>` prompt in the Python *Shell*, individually enter each line below that is set off in `typewriter` font. So next enter

```
type(1.25)
```

Note the name in the last result is `float`, not real or decimal, coming from the term "floating point", for reasons that will be explained later, in Section 1.14.1. Enter

```
type('hello')
```

In your last result you see another abbreviation: `str` rather than string. Enter

```
type([1, 2, 3])
```

Strings and lists are both sequences of parts (characters or elements). We can find the length of that sequence with another function with the abbreviated name `len`. Try both of the following, separately, in the *Shell*:

```
len([2, 4, 6])
len('abcd')
```

Some functions have no parameters, so nothing goes between the parentheses. For example, some types serve as no-parameter functions to create a simple value of their type. Try

```
list()
```

You see the way an empty list is displayed.

Functions may also take more than one parameter. Try

```
max(5, 11, 2)
```

Above, max is short for maximum.

Some of the names of types serve as conversion functions (where there is an obvious meaning for the conversion). Try each of the following, one at a time, in the *Shell*:

```
str(23)
int('125')
```

An often handy Shell feature: an earlier Shell line may to copied and edited by clicking anywhere in the previously displayed line and then pressing ENTER. For instance you should have entered several lines starting with `len`. click on any one, press ENTER, and edit the line for a different test.

1.4. Integer Arithmetic

1.4.1. Addition and Subtraction. We start with the integers and integer arithmetic, not because arithmetic is exciting, but because the symbolism should be mostly familiar. Of course arithmetic is important in many cases, but Python is probably more often used to manipulate text and other sorts of data, as in the sample program in Section 1.2.2.

Python understands numbers and standard arithmetic. *For the whole section on integer arithmetic*, where you see a set-off line in `typewriter` font, type individual lines at the >>> prompt in the Python *Shell*. Press Enter after each line to get Python to respond:

```
77
2 + 3
5 - 7
```

Python should evaluate and print back the value of each expression. Of course the first one does not require any calculation. It appears the shell just echoes back what you printed. Do note that the line with the value *produced* by the shell does not start with >>> and appears at the left margin. Hence you can distinguish what you type (after the ">>>" prompt) from what the computer responds.

The Python Shell is an interactive interpreter. As you can see, after you press Enter, it is evaluating the expression you typed in, and then printing the result automatically. This is a very handy environment to check out simple Python syntax and get instant feedback. For more elaborate programs that you want to save, we will switch to an Editor Window later.

1.4.2. Multiplication, Parentheses, and Precedence. Try in the *Shell*:

```
2 x 3
```

You should get your first *syntax* error. The 'x' should have become highlighted, indicating the location where the Python interpreter discovered that it cannot understand you: Python does not use x for multiplication as you may have done in grade school. The x can be confused with the use of x as a variable (more on that later). Instead the symbol for multiplication is an asterisk '*'. Enter each of the following. You may include spaces or not. The Python interpreter can figure out what you mean either way. Try in the *Shell*:

```
2*5
2 + 3 * 4
```

If you expected the last answer to be 20, think again: Python uses the normal *precedence* of arithmetic operations: Multiplications and divisions are done before addition and subtraction, unless there are parentheses. Try

```
(2+3)*4
2 * (4 - 1)
```

Now try the following in the *Shell*, exactly as written, followed by Enter, with *no* closing parenthesis:

```
5 * (2 + 3
```

Look carefully. There is no answer given at the left margin of the next line and no prompt >>> to start a *new* expression. If you are using Idle, the cursor has gone to the next line and has only *indented* slightly. Python is waiting for you to finish your expression. It is smart enough to know that opening parentheses are always followed by the same number of closing parentheses. The cursor is on a *continuation* line. Type just the matching close-parenthesis and Enter,

```
)
```

and you should finally see the expression evaluated. (In some versions of the Python interpreter, the interpreter puts '...' at the beginning of a continuation line, rather than just indenting.)

Negation also works. Try in the *Shell*:

```
-(2 + 3)
```

1.4.3. Division and Remainders. If you think about it, you learned several ways to do division. Eventually you learned how to do division resulting is a decimal. Try in the *Shell*:

```
5/2
14/4
```

As you saw in the previous section, numbers with decimal points in them are of type float in Python. They are discussed more in Section 1.14.1.

In the earliest grades you would say "14 divided by 4 is 3 with a remainder of 2". The problem here is that the answer is in two parts, the integer quotient 3 and the remainder 2, and neither of these results is the same as the decimal result. Python has separate operations to generate each part. Python uses the doubled division symbol `//` for the operation that produces just the integer quotient, and introduces the symbol `%` for the operation of finding the remainder. Try each in the *Shell*

```
14/4
14//4
14%4
```

Now predict and then try each of

```
23//5
23%5
20%5
6//8
6%8
```