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
```

Finding remainders will prove more useful than you might think in the future!

1.5. Strings, Part I

Enough with numbers for a while. Strings of characters are another important type in Python.

1.5.1. String Delimiters, Part I. A string in Python is a sequence of characters. For Python to recognize a sequence of characters, like `hello`, as a string, it must be enclosed in quotes to delimit the string.

For this whole section on strings, continue trying each set-off line of code in the *Shell*. Try

```
"hello"
```

Note that the interpreter gives back the string with *single* quotes. Python does not care what system you use. Try

```
'Hi!'
```

Having the choice of delimiters can be handy.

EXERCISE 1.5.1.1. * Figure out how to give Python the string containing the text: `I'm happy.` Try it. If you got an error, try it with another type of quotes, and figure out why that one works and not the first.

There are many variations on delimiting strings and embedding special symbols. We will consider more ways later in Section 1.8.

A string can have any number of characters in it, including 0. The empty string is `''` (two quote characters with nothing between them).

Strings are a new Python type. Try

```
type('dog')
type('7')
type(7)
```

The last two lines show how easily you can get confused! Strings can include any characters, *including* digits. Quotes turn even digits into strings. This will have consequences in the next section....

1.5.2. String Concatenation. Strings also have operation symbols. Try in the *Shell* (noting the *space* after `very`):

```
'very ' + 'hot'
```

The plus operation with strings means *concatenate* the strings. Python looks at the type of operands before deciding what operation is associated with the +.

Think of the relation of addition and multiplication of integers, and then guess the meaning of

```
3*'very ' + 'hot'
```

Were you right? The ability to repeat yourself easily can be handy.

EXERCISE 1.5.2.1. * Figure out a *compact* way to get Python to make the string, "YesYesYesYesYes", and try it. How about "MaybeMaybeMaybeYesYesYesYesYes" ? Hint: [2]

Predict the following and then test. Remember the last section on types:

```
7+2
'7'+'2'
```

Python checks the types and interprets the plus symbol based on the type. Try

```
'7'+2
```

With mixed string and int types, Python sees an ambiguous expression, and does not guess which you want – it just gives an error! [3]

[2] Hint for the second one: use two *'s and a +.

[3] Be careful if you are a Java programmer! This is unlike Java, where the 2 would be automatically converted to '2' so the concatenation would make sense.

1.6. Variables and Assignment

Each set-off line in this section should be tried in the Shell.

Try

```
width = 10
```

Nothing is displayed by the interpreter after this entry, so it is not clear anything happened. Something has happened. This is an *assignment statement*, with a *variable*, `width`, on the left. A variable is a name for a value. An assignment statement associates a variable name on the left of the equal sign with the value of an expression calculated from the right of the equal sign. Enter

```
width
```

Once a variable is assigned a value, the variable can be used in place of that value. The response to the expression `width` is the same as if its value had been entered.

The interpreter does not print a value after an assignment statement because the value of the expression on the right is not lost. It can be recovered if you like, by entering the variable name and we did above.

Try each of the following lines:

```
height = 12
area = width * height
area
```

The equal sign is an unfortunate choice of symbol for assignment, since Python's usage is not the mathematical usage of the equal sign. If the symbol ←had appeared on keyboards in the early 1990's, it would probably have been used for assignment instead of =, emphasizing the asymmetry of assignment. In mathematics an equation is an *assertion* that *both* sides of the equal sign *are already, in fact, equal*. A Python assignment statement *forces* the variable on the left hand side *to become* associated with the value of the expression on the right side. The difference from the mathematical usage can be illustrated. Try:

```
10 = width
```

so this is not equivalent in Python to `width = 10`. The *left hand* side must be a variable, to which the assignment is made. Try

```
width = width + 5
```

This is, of course, nonsensical as mathematics, but it makes perfectly good sense as an assignment, with the right-hand side calculated first. Can you figure out the value that is now associated with width? Check by entering

```
width
```

In the assignment statement, the expression on the right is evaluated *first*. At that point `width` was associated with its original value 10, so `width + 5` had the value of 10 + 5 which is 15. That value was then assigned to the variable on the left (`width` again) to give it a *new* value. We will modify the value of variables in a similar way routinely.

Assignment and variables work equally well with strings. Try:

```
first = 'Sue'
last = 'Wong'
name = first + ' ' + last
name
```

Try entering:

```
first = fred
```

Note the different form of the error message. The earlier errors in these tutorials were *syntax* errors: errors in translation of the instruction. In this last case the syntax was legal, so the interpreter went on to execute the instruction. Only *then* did it find the error described. There are no quotes around `fred`, so the interpreter assumed `fred` was an identifier, but the name fred was not defined at the time the line was executed.

It is easy to forget quotes where you need them and put them around a variable name that should not have them!

Try in the *Shell*:

```
fred = 'Frederick'
first = fred
first
```

Now `fred`, without the quotes, makes sense.

There are more subtleties to assignment and the idea of a variable being a "name for" a value, but we will worry about them later, in Section 2.4.6. They do not come up if our variables are just numbers and strings.

Autocompletion: A handy short cut. Python remembers all the variables you have defined at any moment. This is handy when editing. Without pressing Enter, type into the Shell just

f

Then *hold down* the Alt key and press the '/' key. This key combination is abbreviated Alt-/. You should see `f` autocompleted to be `first`. This is particularly useful if you have long identifiers! You can press Alt-/ several times if more than one identifier starts with the initial sequence of characters you typed. If you press Alt-/ again you should see `fred`. Backspace and edit so you have `fi`, and then and press Alt-/ again. You should not see fred this time, since it does not start with `fi`.

1.6.1. Literals and Identifiers. Expressions like `27` or `'hello'` are called *literals*, coming from the fact that they *literally* mean exactly what they say. They are distinguished from variables, who value is *not* directly determined by their name.

The sequence of characters used to form a variable name (and names for other Python entities later) is called an *identifier*. It identifies a Python variable or other entity.

There are some restrictions on the character sequence that make up an identifier:

- The characters must all be letters, digits, or underscores '_', and must start with a letter. In particular, punctuation and blanks are not allowed.
- There are some words that are *reserved* for special use in Python. You may not use these words as your own identifiers. They are easy to recognize in Idle, because they are automatically colored orange. For the curious, you may *read* the full list:

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

There are also identifiers that are automatically defined in Python, and that you could redefine, but you probably should not unless you really know what you are doing! When you start the editor, we will see how Idle uses color to help you know what identifies are predefined.

Python is case sensitive: The identifiers `last`, `LAST`, and `LaSt` are all different. Be sure to be consistent. Using the Alt-/ auto-completion shortcut in Idle helps ensure you are consistent.

What is legal is distinct from what is conventional or good practice or recommended. Meaningful names for variables are important for the humans who are looking at programs, understanding them, and revising them. That sometimes means you would like to use a name that is more than one word long, like `price at opening`, but blanks are illegal! One poor option is just leaving out the blanks, like `priceatopening`. Then it may be hard to figure out where words split. Two practical options are

- underscore separated: putting underscores (which are legal) in place of the blanks, like `price_at_opening`.
- using *camelcase*: omitting spaces and using all lowercase, except capitalizing all words after the first, like `priceAtOpening`

Use the choice that fits your taste (or the taste or convention of the people you are working with).

1.7. Print Function, Part I

In interactive use of the Python interpreter, you can type an expression and immediately see the result of its evaluation. This is fine to test out syntax and maybe do simple calculator calculations. In a program

run from a file like the first sample program, Python does not display expressions this way. If you want your program to display something, you can give explicit instructions with the print function. Try in the *Shell*:

```
x = 3
y = 5
print('The sum of', x, 'plus', y, 'is', x+y)
```

The print function will prints as strings everything in a comma-separated sequence of expressions, and it will separate the results with single blanks by default. Note that you can mix types: anything that is not already a string is automatically converted to its string representation.

You can also use it with no parameters:

```
print()
```

to just advance to the next line.

1.8. Strings Part II

1.8.1. Triple Quoted String Literals. Strings delimited by one quote character are required to lie within a single Python line. It is sometimes convenient to have a multi-line string, which can be delimited with triple quotes: Try typing the following. You will get continuation lines until the closing triple quotes. Try in the *Shell*:

```
sillyTest = '''Say,
"I'm in!"
This is line 3'''
print(sillyTest)
```

The line structure is preserved in a multi-line string. As you can see, this also allows you to embed both single and double quote characters!

1.8.2. Escape Codes. Continuing in the *Shell* with `sillyTest`, enter just

```
sillyTest
```

The answer looks strange! It indicates an alternate way to encode the string internally in Python using *escape codes*. Escape codes are embedded inside string literals and start with a backslash character (\). They are used to embed characters that are either unprintable or have a special syntactic meaning to Python that you want to suppress. In this example you see the most common ones:

Escape code	Meaning
\'	'
\n	newline
\\	\

The newline character indicates further text will appear on a new line when *printed*. When you use a print function, you get the actual printed meaning of the escaped coded character.

Predict the result, and try in the *Shell*:

```
print('a\nb\n\nc')
```

Did you guess the right number of lines splitting in the right places?

1.9. The Idle Editor and Execution

1.9.1. Loading a Program in the Idle Editor, and Running It. It is time to put longer collections of instructions together. That is most easily done by creating a text file and running the Python interpreter on the file. Idle simplifies that process.

First you can put an existing file into an Idle Edit Window. Click on the Idle File menu and select Open. (Or as you see, you can use the shortcut Ctrl+O. That means holding down the Ctrl key, and pressing the letter O for Open.) You should get a file selection dialog. You should have the sample program madlib.py displayed in the list. Select it and open it. (If you do not see the program, then you either failed to download the example programs, Section 1.2.1, or you did not start Idle in the proper folder, Section 1.2.4.)

You will see the source code again. Now run this program from inside of Idle: Go to the Run menu of that Edit window, and select Run Module. Notice the shortcut (F5).