```
>>> ================================ RESTART ========
>>>
Hello world!
>>>
```

You could also have typed this single printing line directly in the Shell in response to a Shell prompt. When you see `>>>`, you could enter the print function and get the exchange between you and the Shell:

```
>>> print('Hello world')
Hello world!
>>>
```

The three lines above are *not* a program you could save in a file and run. This is just an exchange in the *Shell*, with its `>>>` prompts, individual line to execute and the response. Again, just the single line, with no `>>>`,

```
print('Hello world!')
```

entered into the *Edit* window forms a program you can save and run. We will shortly get to more interesting many-statement programs, where it is much more convenient to use the Edit window than the Shell!

1.9.4. Program Documentation String. The program above is self evident, and shows how short and direct a program can be (unlike other languages like Java). Still, right away, get used to documenting a program. Python has a special feature: If the beginning of a program is just a quoted string, that string is taken to be the program's *documentation string*. Open the example file `hello2.py` in the Edit window:

```
'''A very simple program,
showing how short a Python program can be!
Authors: ___, ___
'''

print('Hello world!') #This is a stupid comment after the # mark
```

Most commonly, the initial documentation goes on for several lines, so a multi-line string delimiter is used (the triple quotes). Just for completeness of illustration in this program, another form of comment is also shown, a comment that starts with the symbol # and extends to the end of the line. The Python interpreter completely ignores this form of comment. Such a comment should only be included for better human understanding. Avoid making comments that do not really aid human understanding. (Do what I say, not what I did above.) Good introductory comment strings and appropriate names for the parts of your programs make fewer # symbol comments needed.

Run the program and see the documentation and comment make no difference in the result.

1.9.5. Screen Layout. Of course you can arrange the windows on your computer screen any way that you like. A suggestion as you start to use the combination of the editor to write, the shell to run, and the tutorial to follow along: Make all three mostly visible your computer screen at once. Drag the editor window to the upper left. Place the Shell window to the lower left, and perhaps reduce its height a bit so there is not much overlap. If you are looking at the web version of the tutorial on the screen, make it go top to bottom on the right, but not overlap the Idle windows too much. The web page rendering should generally adapt to the width pretty well. You can always temporarily maximize the window. Before resizing the browser window, it is good to look for an unusual phrase on your page, and search for it after resizing, since resizing can totally mess up your location in the web page.

There is an alternative to maximization for the Idle editor window: It you want it to go top to bottom of the screen but not widen, you can toggle that state with Alt-2. Play with all this.

1.10. Input and Output

1.10.1. The `input` Function. The hello program of Section 1.9.3 always does the same thing. This is not very interesting. Programs are only going to be reused if they can act on a variety of data. One way to get data is directly from the user. Modify the hello.py program as follows in the editor, and save it from the File menu with Save As...., using the name `hello_you.py`.

```
person = input('Enter your name: ')
print('Hello', person)
```

Run the program. In the Shell you should see

```
Enter your name:
```

Follow the instruction (and press Enter). Make sure the typing cursor is in the Shell window, at the end of this line. After you type your response, you can see that the program has taken in the line you typed. That is what the built-in function `input` does: First it prints the string you give as a parameter (in this case `'Enter your name:  '`), and then it waits for a line to be typed in, and returns the string of characters you typed. In the `hello_you.py` program this value is assigned to the variable person, for use later.

The parameter inside the parentheses after `input` is important. It is a *prompt*, prompting you that keyboard input is expected at that point, and hopefully indicating what is being requested. Without the prompt, the user would not know what was happening, and the computer would just sit there waiting!

Open the example program, `interview.py`. Before running it (with any made-up data), see if you can figure out what it will do:

```
'''Illustrate input and print.'''

applicant = input("Enter the applicant's name: ")
interviewer = input("Enter the interviewer's name: ")
time = input("Enter the appointment time: ")
print(interviewer, "will interview", applicant, "at", time)
```

The statements are executed in the order they appear in the text of the program: *sequentially*. This is the simplest way for the execution of the program to flow. You will see instructions later that alter that natural flow.

If we want to reload and modify the `hello_you.py` program to put an exclamation point at the end, you could try:

```
person = input('Enter your name: ')
print('Hello', person, '!')
```

Run it and you see that it is not spaced right. There should be no space after the person's name, but the default behavior of the print function is to have each field printed separated by a space. There are several ways to fix this. You should know one. Think about it before going on to the next section. Hint: [4]

1.10.2. Print with Keyword Parameter sep. One way to put punctuation but no space after the person in hello_you.py is to use the plus operator, +. Another approach is to change the default separatpr between fields in the print function. This will introduce a new syntax feature, *keyword parameters*. The print function has a keyword parameter named sep. If you leave it out of a call to print, as we have so far, it is set equal to a space by default. If you add a final field, sep=", in the print function in hello_you.py, you get the following example file, `hello_you2.py`:

```
person = input('Enter your name: ')
print('Hello ', person, '!', sep='')
```

Try the program.

Keyword paramaters must be listed at the end of the parameter list.

1.10.3. Numbers and Strings of Digits. Consider the following problem: Prompt the user for two numbers, and then print out a sentence stating the sum. For instance if the user entered 2 and 3, you would print "The sum of 2 and 3 is 5."

You might imagine a solution like the example file `addition1.py`, shown below. There is a problem. Can you figure it out before you try it? Hint: [5] End up running it in any case.

```
x = input("Enter an integer: ")
y = input("Enter another integer: ")
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='')  # error!
```

[4]The + operation on strings adds no extra space.

[5]The input function produces values of string type.

We do not want string concatenation, but integer addition. We need integer operands. Briefly mentioned in Section 1.3 was the fact that we can use type names as functions to convert types. One approach would be to do that. Further variable names are also introduced in the example `addition2.py` file below to emphasize the distinctions in types. Read and run:

```
'''Conversion of strings to int before addition'''

xString = input("Enter an integer: ")
x = int(xString)
yString = input("Enter another integer: ")
y = int(yString)
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='')
```

Needing ito convert string input to numbersis a common situation, both with keyboard input and later in web pages. While the extra variables above emphasized the steps, it is more concise to write as in the variation in example file, `addition3.py`, doing the conversons to type `int` immediately:

```
'''Two numeric inputs'''

x = int(input("Enter an integer: "))
y = int(input("Enter another integer: "))
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='')
```

The simple programs so far have followed a basic *programming pattern*: input-calculate-output. Get all the data first, calculate with it second, and output the results last. The pattern sequence would be even clearer if we explicitly create a named result variable in the middle, as in `addition4.py`:

```
x = int(input("Enter an integer: "))
y = int(input("Enter another integer: "))
sum = x + y
print('The sum of ', x, ' and ', y, ' is ', sum, '.', sep='')
```

We will see more complicated patterns, which involve repetition, in the future.

Exercise 1.10.3.1. * Write a version, `add3.py`, that asks for three numbers, and lists all three, and their sum, in similar format to the example above.

Exercise 1.10.3.2. * a. Write a program, `quotient.py`, that prompts the user for two integers, and then prints them out in a sentence with an integer division problem like `"The quotient of 14 and 3 is 4 with a remainder of 2"`. Review Section 1.4.3 if you forget the integer division or remainder operator.

1.10.4. String Format Operation. A common convention is fill-in-the blanks. For instance,

```
Hello _____!
```

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. Python has a similar construction, better called fill-in-the-braces. There is a particular operation on strings called `format`, that makes substitutions into places enclosed in braces. For instance the example file, `hello_you3.py`, creates and prints the same string as in `hello_you2.py` from the previous section:

```
person = input('Enter your name: ')
greeting = 'Hello {}!'.format(person)
print(greeting)
```

There are several new ideas here!.

First *method* calling syntax is used. You will see in this more detail at the beginning of the next chapter. Strings and other objects have a special syntax for functions, called *methods*, associated with the *particular type of object*. In particular `str` objects have a method called `format`. The syntax for methods has the object followed by a period followed by the method name, and further parameters in parentheses.

`object.methodname(paramters)`

In the example above, the object is the string `'Hello {}!'`. The method is named `format`. There is one further parameter, `person`.

The string has a special form, with braces embedded. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the format method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter

In the code above, this new string is assigned to the identifier `greeting`, and then the string is printed. The identifier `greeting` was introduced to break the operations into a clearer sequence of steps. Since the value of `greeting` is only referenced once, it can be eliminated with the more concise version:

```
person = input('Enter your name: ')
print('Hello {}!'.format(person))
```

Consider the interview program. Suppose we want to add a period at the end of the sentence (with no space before it). One approach would be to combine everything with plus signs. Another way is printing with keyword `sep=''`. Another approach is with string formating. Here the idea is to fill in the blanks in

```
_____ will interview _____ at _____.
```

There are multiple places to substiitute, and the format approach can be extended to multiple substitutions: Each place in the format string where there is `'{}'`, the `format` operation will substitute the value of the next parameter in the format parameter list.

Run the example file `interview2.py`, and check that the results from all three methods match.

```
'''Compare different approaches to printing with embedded values.'''

applicant = input("Enter the applicant's name: ")
interviewer = input("Enter the interviewer's name: ")
time = input("Enter the appointment time: ")
print(interviewer + ' will interview ' + applicant + ' at ' + time +'.')
print(interviewer, ' will interview ', applicant, ' at ', time, '.', sep='')
print('{} will interview {} at {}.'.format(interviewer, applicant, time))
```

A technical point: Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final formatted string. The rule is to double the braces: `'{{'` and `'}}'`. The example code `formatBraces.py`, shown below, makes `setStr` refer to the string `'The set is {5, 9}.'`. The initial and final doubled braces in the format string generate literal braces in the formatted string:

```
a = 5
b = 9
formatStr = 'The set is {{{}, {}}}.'
setStr = formatStr.format(a, b)
print(setStr)
```

This kind of format string depends directly on the order of the parameters to the format method. There is another approach with a dictionary, that was used in the first sample program, and will be discussed more in Section 1.12.2 on dictionaries. The dictionary approach is probably the best in many cases, but the count-based approach is an easier start, particularly if the parameters are just used once, in order.

(Optional elaboration) Imagine the format parmaters numbered in order, starting from 0. In this case 0, 1, and 2. The number of the parameter position may be included inside the braces, so an alternative to the last line of `interview2.py` is (added in example file `interview3.py`):

```
print('{0} will interview {1} at {2}.'.format(interviewer, applicant, time))
```

This is more verbose than the previous version, with no obvious advantage. If you desire to use some of the parameters more than once, then the approach with the numerical identification with the parameters is useful. Every place the string includes `'{0}'`, the `format` operation will substitute the value of the initial parameter in the list. Wherever `'{1}'` appears, the next format parameter will be substituted....

Predict the results of the example file `arith.py` shown below,and then check yourself by running it. In this case the numbers referring to the parameter positions are necessary. They are both repeated and used out of order:

```
'''Fancier format string example.'''
```