*Interaction*. Assuming your Python is configured properly, you should participate in an interaction that looks something like the following. You can run this any way you like: in IDLE, from a shell prompt, and so on.

Note: lines that start with a “**%**” denote shell-prompt command lines; don’t run these lines at Python’s “**>>>**” prompt, and don’t type either the “**%**” or “**>>>**” characters yourself—enter just the code after the these prompts.  Run “**%**” shell-prompt commands in a Command Prompt window on Windows, and run “**>>>**” commands in Python (IDLE’s shell window, etc.).  You don’t need to run the first command that follows if you’re working only in IDLE, and you may need to use a full “C:\Python35\python” instead of just “python” if Python isn’t on your system’s PATH setting:

% **python**# "%" means your shell prompt (e.g., "C:\code>")

*...copyright information lines...*

>>> **"Hello World!"**# ">>>" is Python’s prompt: Python code goes here

'Hello World!'

>>>                       # Ctrl-D, Ctrl-Z, or window close to exit

2.      *Programs*. Here’s what your code (i.e., module) file and shell interactions should look like; again, feel free to run this other ways—by clicking its icon, by IDLE’s Edit/RunScript menu option, and so on:

Note: in this section a “**File: xxx.py**” in italics gives the name of the file in which code following it is to be stored, and be sure to always use the parenthesized call form “**print(xxx)**” if you’re using Python 3.X (see the [statements unit](https://learning-python.com/class/Workbook/unit04.htm) for more details).

*# File: module1.py*# enter this code in a new file

print 'Hello module world!'      # 3.X: use the form print('…')

% **python module1.py**# run this command line at a system prompt

Hello module world!

3.      *Modules*. The following interaction listing illustrates running a module file by importing it. Remember that you need to reload it to run again without stopping and restarting the interpreter. The bit about moving the file to a different directory and importing it again is a trick question: if Python generates a module1.pyc file in the original direc­tory, it uses that when you import the module, even if the source code file (.py) has been moved to a directory not on Python’s search path. The .pyc file is written auto­matically if Python has access to the source file’s directory and contains the com­piled bytecode version of a module. We look at how this works again in the modules unit.

% **python**

>>> **import module1**

Hello module world!

>>>

4.      *Scripts*. Assuming your platform supports the #! trick, your solution will look like the fol­lowing (though your #! line may need to list another path on your machine):

*File: module1.py*

#!/usr/local/bin/python          (or #!/usr/bin/env python)

print 'Hello module world!'

% **chmod +x module1.py**

% **module1.py**

Hello module world!

5.      *Errors*. The interaction below demonstrates the sort of error messages you get if you com­plete this exercise. Really, you’re triggering Python exceptions; the default excep­tion handling behavior terminates the running Python program and prints an error message and stack trace on the screen. The stack trace shows where you were at in a program when the exception occurred (it’s not very interesting here, since the excep­tions occur at the top level of the interactive prompt; no function calls were in progress). In the exceptions unit, you will see that you can catch exceptions using “try” statements and process them arbitrarily; you’ll also see that Python includes a full-blown source-code debugger for special error detection requirements. For now, notice that Python gives meaningful messages when programming errors occur (instead of crashing silently):

% **python**

>>> **1 / 0**

Traceback (innermost last):

  File "<stdin>", line 1, in ?

ZeroDivisionError: integer division or modulo

>>>

>>> **x**

Traceback (innermost last):

  File "<stdin>", line 1, in ?

NameError: x

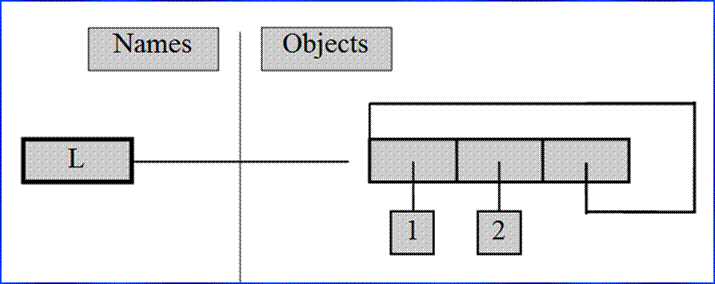
6.      *Breaks*. When you type this code:

L = [1, 2]

L.append(L)

you create a cyclic data-structure in Python. In Python releases before Version 1.5.1, the Python printer wasn’t smart enough to detect cycles in objects, and it would print an unending stream of [1, 2, [1, 2, [1, 2, [1, 2, and so on, until you hit the break key combination on your machine (which, technically, raises a keyboard-interrupt excep­tion that prints a default message at the top level unless you intercept it in a program). Beginning with Python Version 1.5.1, the printer is clever enough to detect cycles and prints [[...]] instead to let you know.

The reason for the cycle is subtle and requires information you’ll gain in the next unit. But in short, assignment in Python always generates refer­ences to objects (which you can think of as implicitly followed pointers). When you run the first assignment above, the name L becomes a named reference to a two-item list object. Now, Python lists are really arrays of object references, with an append method that changes the array in-place by tacking on another object reference. Here, the append call adds a reference to the front of L at the end of L, which leads to the cycle illustrated in the figure below. Believe it or not, cyclic data structures can sometimes be useful (but maybe not when printed!). Today, Python can also reclaim (garbage collect) such objects cyclic automatically.



*A cyclic list*