In [ ]:

```
Sometimes when you are importing from a module, you would like to know whether
a modules function is being used as an import, or if you are using the original
.py file of that module. In this case we can use the:

      if __name__ == "__main__":

line to determine this. For example:

When your script is run by passing it as a command to the Python interpreter:

    python myscript.py

all of the code that is at indentation level 0 gets executed. Functions and
classes that are defined are, well, defined, but none of their code gets ran.
Unlike other languages, there's no main() function that gets run automatically
- the main() function is implicitly all the code at the top level.

In this case, the top-level code is an if block.  __name__ is a built-in variable
 which evaluate to the name of the current module. However, if a module is being
 run directly (as in myscript.py above), then __name__ instead is set to the
 string "__main__". Thus, you can test whether your script is being run directly
  or being imported by something else by testing

    if __name__ == "__main__":
        ...

If that code is being imported into another module, the various function and
class definitions will be imported, but the main() code won't get run. As a
basic example, consider the following two scripts:

    # file one.py
    def func():
        print("func() in one.py")

    print("top-level in one.py")

    if __name__ == "__main__":
        print("one.py is being run directly")
    else:
        print("one.py is being imported into another module")

and then:

    # file two.py
    import one

    print("top-level in two.py")
    one.func()

    if __name__ == "__main__":
        print("two.py is being run directly")
    else:
        print("two.py is being imported into another module")

Now, if you invoke the interpreter as

    python one.py

The output will be

    top-level in one.py
```

```
one.py is being run directly
If you run two.py instead:

    python two.py

You get

  top-level in one.py
  one.py is being imported into another module
  top-level in two.py
  func() in one.py
  two.py is being run directly

Thus, when module one gets loaded, its __name__ equals "one" instead of __main__.
```