**Q1. Is it permissible to use several import statements to import the same module? What would the goal be? Can you think of a situation where it would be beneficial?**

o get a list of the modules that have already been imported, you can look up sys.modules.keys() (note that urllibhere imports a *lot* of other modules):

>>> import sys

>>> print len(sys.modules.keys())

44

>>> print sys.modules.keys()

['copy\_reg', 'sre\_compile', '\_sre', 'encodings', 'site', '\_\_builtin\_\_', 'sysconfig', '\_\_main\_\_', 'encodings.encodings', 'abc', 'posixpath', '\_weakrefset', 'errno', 'encodings.codecs', 'sre\_constants', 're', '\_abcoll', 'types', '\_codecs', 'encodings.\_\_builtin\_\_', '\_warnings', 'genericpath', 'stat', 'zipimport', '\_sysconfigdata', 'warnings', 'UserDict', 'encodings.utf\_8', 'sys', 'virtualenvwrapper', '\_osx\_support', 'codecs', 'readline', 'os.path', 'sitecustomize', 'signal', 'traceback', 'linecache', 'posix', 'encodings.aliases', 'exceptions', 'sre\_parse', 'os', '\_weakref']

>>> import urllib

>>> print len(sys.modules.keys())

70

>>> print sys.modules.keys()

['cStringIO', 'heapq', 'base64', 'copy\_reg', 'sre\_compile', '\_collections', '\_sre', 'functools', 'encodings', 'site', '\_\_builtin\_\_', 'sysconfig', 'thread', '\_ssl', '\_\_main\_\_', 'operator', 'encodings.encodings', '\_heapq', 'abc', 'posixpath', '\_weakrefset', 'errno', '\_socket', 'binascii', 'encodings.codecs', 'urllib', 'sre\_constants', 're', '\_abcoll', 'collections', 'types', '\_codecs', 'encodings.\_\_builtin\_\_', '\_struct', '\_warnings', '\_scproxy', 'genericpath', 'stat', 'zipimport', '\_sysconfigdata', 'string', 'warnings', 'UserDict', 'struct', 'encodings.utf\_8', 'textwrap', 'sys', 'ssl', 'virtualenvwrapper', '\_osx\_support', 'codecs', 'readline', 'os.path', 'strop', '\_functools', 'sitecustomize', 'socket', 'keyword', 'signal', 'traceback', 'urlparse', 'linecache', 'itertools', 'posix', 'encodings.aliases', 'time', 'exceptions', 'sre\_parse', 'os', '\_weakref']

>>> import urllib #again!

>>> print len(sys.modules.keys()) #has not loaded any additional modules

70

Let's give it a whirl:

import sys

>>> sys.modules["foo"] = "bar" # Let's pretend we imported a module named "foo", which is a string.

>>> print \_\_import\_\_("foo")

bar # Not a module, that's my string!

In Python, you use the **import** keyword to make code in one **module** available in another. Imports in Python are important for **structuring your code** effectively. Using imports properly will make you more productive, allowing you to reuse code while keeping your projects maintainable.

This tutorial will provide a thorough overview of Python’s import statement and how it works. The import system is powerful, and you’ll learn how to harness this power. While you’ll cover many of the concepts behind Python’s import system, this tutorial is mostly example driven. You’ll learn from several code examples throughout.

**In this tutorial, you’ll learn how to:**

* Use **modules**, **packages**, and **namespace packages**
* Handle **resources** and **data files** inside your packages
* Import modules **dynamically** at runtime
* **Customize** Python’s import system

>>> import math

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', ..., 'math']

>>> dir(math)

['\_\_doc\_\_', ..., 'nan', 'pi', 'pow', ...]

**Q2. What are some of a module's characteristics? (Name at least one.)**

**Characteristics of Modules:**

* Modules contain instructions, processing logic, and data.
* Modules can be separately compiled and stored in a library.
* Modules can be included in a program.
* Module segments can be used by invoking a name and some parameters.
* Module segments can be used by other modules.

**Q3. Circular importing, such as when two modules import each other, can lead to dependencies and bugs that aren't visible. How can you go about creating a program that avoids mutual importing?**

## Python Circular Imports:

Circular importing is a conceptual error that is formed due to circular dependency created with the import statement in our Python program.

1. Python Circular Imports is a type of Circular dependency. It occurs in python when two or more models import each other and it repeats the importing connection into an infinite circular call.
2. With Circular Imports, the python script gives an error. To run the python script it has to be removed and it is very difficult to find and remove the script manually.
3. Circular imports are created because of the bad coding design and implementation-related logical anomalies.
4. import module2
5. def function1():
6. module2.function2()
7. def function3():
8. print('Hasta la vista, Gaurav!')
9. # module 2
10. import module1
11. def function2():
12. print('Hey, Gaurav')
13. module1.function3()
14. # \_\_init.py
15. import module1
16. module1.function1()

**Q4. Why is \_ \_all\_ \_ in Python?**

\_\_all\_\_ is **used to document the public API of a Python module**.

The public names defined by a module are determined by checking the module’s namespace for a variable named \_\_all\_\_; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in \_\_all\_\_ are all considered public and are required to exist. If \_\_all\_\_ is not defined, the set of public names includes all names found in the module’s namespace which do not begin with an underscore character ('\_'). \_\_all\_\_ should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

**Q5. In what situation is it useful to refer to the \_ \_name\_ \_ attribute or the string '\_ \_main\_ \_'?**

Conclusion. There is a really nice use case for the \_\_name\_\_ variable, whether you want a file that can be run as the main program or imported by other modules. We can use an if \_\_name\_\_ == "\_\_main\_\_" block to allow or prevent parts of code from being run when the modules are imported.

To run one of these functions modify the if \_\_name\_\_ == "\_\_main\_\_" part of file\_one to look like this:

if \_\_name\_\_ == "\_\_main\_\_":

print("File one executed when ran directly")

function\_two()

else:

print("File one executed when imported")

**Q6. What are some of the benefits of attaching a program counter to the RPN interpreter application, which interprets an RPN script line by line?**

## What Does Reverse Polish Notation (RPN) Mean?

Reverse Polish notation (RPN) is a method for conveying mathematical expressions without the use of separators such as brackets and parentheses. In this notation, the operators follow their operands, hence removing the need for brackets to define evaluation priority. The operation is read from left to right but execution is done every time an operator is reached, and always using the last two numbers as the operands. This notation is suited for computers and calculators since there are fewer characters to track and fewer operations to execute.

Reverse Polish notation is also known as postfix notation.

Techopedia Explains Reverse Polish Notation (RPN)

Reverse Polish notation was proposed by Burks, Warren and Wright in 1954 and so named because it was simply the reverse of Polish notation (prefix notation), invented by the Polish logician Jan Lukasiewicz, which puts the operator before the operands. In the 1960s, it was then independently reinvented by E.W. Dijkstra and F.L. Bauer for reducing the number of times computer memory is accessed and increasing performance. It made use of the computer’s stack to store its operands before executing the operator.

RPN leads to faster calculations for a couple of reasons. One is that there is less information to store. Therefore, instead of needing to store nine characters for the expression ((5 – 3) \* 2), computers using RPN only need to store five characters with the expression 5 3 – 2 \*. And because there are fewer characters to process, execution becomes faster.

So in a computer using RPN, the evaluation of the expression 5 1 – 3 \* is as follows:

1. Push 5 into the stack. This is the first value.
2. Push 1 into the stack. This is the second value and is on the position above the 5.
3. Apply the subtraction operation by taking two operands from the stack (1 and 5). The top value (1) is subtracted from the value below it (5), and the result (4) is stored back to the stack. 4 is now the only value in the stack and is in the bottom.
4. Push 3 into the stack. This value is in the position above 4 in the stack.
5. Apply the multiplication operation by taking the last two numbers off the stack and multiplying them. The result is then placed back into the stack. After this operation, the stack now only contains the number 12.

**Q7. What are the minimum expressions or statements (or both) that you'd need to render a basic programming language like RPN primitive but complete— that is, capable of carrying out any computerised task theoretically possible?**

## Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value None). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is

**expression\_stmt** ::= [starred\_expression](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-starred_expression)

## Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

assignment\_stmt ::= ([target\_list](https://docs.python.org/3/reference/simple_stmts.html" \l "grammar-token-python-grammar-target_list) "=")+ ([starred\_expression](https://docs.python.org/3/reference/expressions.html" \l "grammar-token-python-grammar-starred_expression) | [yield\_expression](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-yield_expression))

target\_list ::= [target](https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-target) ("," [target](https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-target))\* [","]

target ::= [identifier](https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-python-grammar-identifier)

| "(" [[target\_list](https://docs.python.org/3/reference/simple_stmts.html" \l "grammar-token-python-grammar-target_list)] ")"

| "[" [[target\_list](https://docs.python.org/3/reference/simple_stmts.html" \l "grammar-token-python-grammar-target_list)] "]"

| [attributeref](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-attributeref)

| [subscription](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-subscription)

| [slicing](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-slicing)

| "\*" [target](https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-target)

## The assert statement

Assert statements are a convenient way to insert debugging assertions into a program:

**assert\_stmt** ::= "assert" [expression](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-expression) ["," [expression](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-expression)]

The simple form, assert expression, is equivalent to

**if** **not** **if** \_\_debug\_\_:

expression: **raise** **AssertionError**

The extended form, assert expression1, expression2, is equivalent to

**if** \_\_debug\_\_:

**if** **not** expression1: **raise** **AssertionError**(expression2)

These equivalences assume that [\_\_debug\_\_](https://docs.python.org/3/library/constants.html#debug__) and [AssertionError](https://docs.python.org/3/library/exceptions.html" \l "AssertionError" \o "AssertionError) refer to the built-in variables with those names. In the current implementation, the built-in variable [\_\_debug\_\_](https://docs.python.org/3/library/constants.html#debug__) is True under normal circumstances, False when optimization is requested (command line option [-O](https://docs.python.org/3/using/cmdline.html#cmdoption-O)). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to [\_\_debug\_\_](https://docs.python.org/3/library/constants.html#debug__) are illegal. The value for the built-in variable is determined when the interpreter starts.

## 7.4. The pass statement

**pass\_stmt** ::= "pass"

[pass](https://docs.python.org/3/reference/simple_stmts.html#pass) is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

**def** f(arg): **pass** *# a function that does nothing (yet)*

**class** **C**: **pass** *# a class with no methods (yet)*

## 7.5. The del statement

**del\_stmt** ::= "del" [target\_list](https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-target_list)

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a [global](https://docs.python.org/3/reference/simple_stmts.html#global) statement in the same code block. If the name is unbound, a [NameError](https://docs.python.org/3/library/exceptions.html" \l "NameError" \o "NameError) exception will be raised.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

*Changed in version 3.2:*Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

## 7.6. The return statement

**return\_stmt** ::= "return" [[expression\_list](https://docs.python.org/3/reference/expressions.html" \l "grammar-token-python-grammar-expression_list)]

[return](https://docs.python.org/3/reference/simple_stmts.html#return) may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else None is substituted.

[return](https://docs.python.org/3/reference/simple_stmts.html#return) leaves the current function call with the expression list (or None) as return value.

When [return](https://docs.python.org/3/reference/simple_stmts.html#return) passes control out of a [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement with a [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) clause, that finally clause is executed before really leaving the function.

In a generator function, the [return](https://docs.python.org/3/reference/simple_stmts.html#return) statement indicates that the generator is done and will cause [StopIteration](https://docs.python.org/3/library/exceptions.html" \l "StopIteration" \o "StopIteration) to be raised. The returned value (if any) is used as an argument to construct [StopIteration](https://docs.python.org/3/library/exceptions.html" \l "StopIteration" \o "StopIteration) and becomes the StopIteration.value attribute.

In an asynchronous generator function, an empty [return](https://docs.python.org/3/reference/simple_stmts.html#return) statement indicates that the asynchronous generator is done and will cause [StopAsyncIteration](https://docs.python.org/3/library/exceptions.html" \l "StopAsyncIteration" \o "StopAsyncIteration) to be raised. A non-empty return statement is a syntax error in an asynchronous generator function.