Python Circular Import

What is a Circular Dependency

Circular dependency occurs when two or more modules depend on each other. This is due to the fact that each module is defined in terms of the other (figure 1).

For example:

Function1()

    Calls function2()

And

Function2()

    Calls function1()

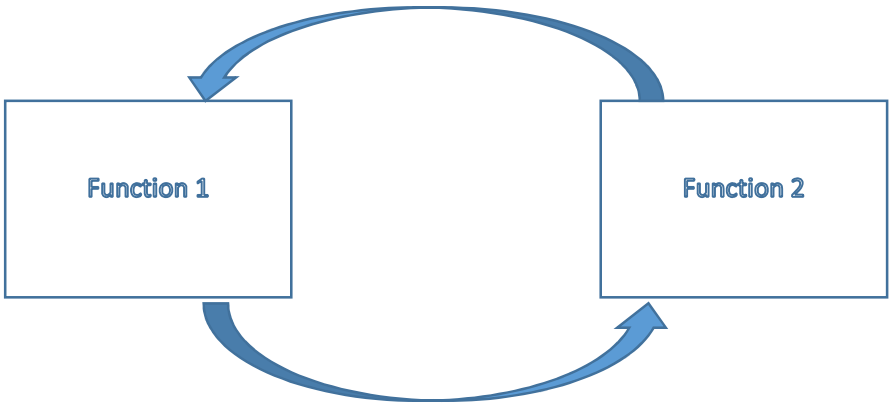

Figure 1

Problems of Circular Dependency

Circular dependency can cause programming difficulties. For example, it may generate tight coupling, and as a consequence, reduced code reusability. This fact also makes the code more difficult to maintain.

In addition, circular dependency can be the source of potential failures, such as infinite recursions, memory leaks, and cascade effects.

What is a Circular Import

Circular import is a form of circular dependency created with the import statement.

For example, let's analyze the following code:

```
1        # module 1
2        import module2
3        def function1():
4              print (“Hello World”)
```

```
1       #module 2
2       import module1
3       module1.function1()
```

Or, similarly:

```
1       #module 2
2       from module1 import funtion1
3       function1()
```

When Python imports a module, it checks the module registry to see if the module was already imported. If the module was already registered, Python uses that object. The module registry is a table of modules that have been initialized, indexed by module name. This table is accessed through sys.modules.

If it was not registered, Python finds the module, initializes it if necessary, and executes it in the new module’s namespace.

In our example, when Python reaches import module2, it loads and executes it. However, module 2 calls for module 1; which in turn defines function1().

The problem is then that function1() is never reached as the program remains in the import statement of module 2, generating a circular dependency.

Best Practices: How to Avoid this Problem

In general, circular imports are the result of bad designs. A deeper analysis of the program could have concluded that both modules could have been merged into one. The resulting code will be something like:

```
1   # module 1&2
2   def function1():
3         print (“Hello World”)
4   # some code
5   function1()
```

However, the merged module may have some unrelated functions (tight coupling) and a very large size.

Another solution could have been to defer the import of module 2 until it is needed. This can be done by placing the import of module 2 after the definition of function1():

```
1        # module 1
2        def function1():
```

```
3           print (“Hello World”)
4       import module2
```

In this case, Python will execute function1(), and then load module2.

This approach doesn’t contradict Python syntax, as the Python documentation says: “It is customary *but not required* to place all import statements at the beginning of a module (or script, for that matter)” [1].

The Python documentation also says that it is advisable to use import X, instead of other statements, such as from module import *, or from module import a,b,c.

Wrapping up

Circular imports are a specific case of circular references. Generally, they can be resolved with better code design. However, sometimes, the resulting design can contain a large amount of code, or mix unrelated functionalities (tight coupling).

References

[1] Python Software Foundation. *Modules.* Available at https://docs.python.org/2/tutorial/modules.html