### 5.4.1 Loaders

Module loaders provide the critical function of loading: module execution. The import machinery calls the `importlib.abc.Loader.exec_module()` method with a single argument, the module object to execute. Any value returned from `exec_module()` is ignored.

Loaders must satisfy the following requirements:

- If the module is a Python module (as opposed to a built-in module or a dynamically loaded extension), the loader should execute the module's code in the module's global name space (`module.__dict__`).
- If the loader cannot execute the module, it should raise an `ImportError`, although any other exception raised during `exec_module()` will be propagated.

In many cases, the finder and loader can be the same object; in such cases the `find_spec()` method would just return a spec with the loader set to `self`.

Module loaders may opt in to creating the module object during loading by implementing a `create_module()` method. It takes one argument, the module spec, and returns the new module object to use during loading. `create_module()` does not need to set any attributes on the module object. If the method returns `None`, the import machinery will create the new module itself.

New in version 3.4: The `create_module()` method of loaders.

Changed in version 3.4: The `load_module()` method was replaced by `exec_module()` and the import machinery assumed all the boilerplate responsibilities of loading.

For compatibility with existing loaders, the import machinery will use the `load_module()` method of loaders if it exists and the loader does not also implement `exec_module()`. However, `load_module()` has been deprecated and loaders should implement `exec_module()` instead.

The `load_module()` method must implement all the boilerplate loading functionality described above in addition to executing the module. All the same constraints apply, with some additional clarification:

- If there is an existing module object with the given name in `sys.modules`, the loader must use that existing module. (Otherwise, `importlib.reload()` will not work correctly.) If the named module does not exist in `sys.modules`, the loader must create a new module object and add it to `sys.modules`.
- The module *must* exist in `sys.modules` before the loader executes the module code, to prevent unbounded recursion or multiple loading.
- If loading fails, the loader must remove any modules it has inserted into `sys.modules`, but it must remove **only** the failing module(s), and only if the loader itself has loaded the module(s) explicitly.

Changed in version 3.5: A `DeprecationWarning` is raised when `exec_module()` is defined but `create_module()` is not.

Changed in version 3.6: An `ImportError` is raised when `exec_module()` is defined but `create_module()` is not.

### 5.4.2 Submodules

When a submodule is loaded using any mechanism (e.g. `importlib` APIs, the `import` or `import-from` statements, or built-in `__import__()`) a binding is placed in the parent module's namespace to the submodule object. For example, if package `spam` has a submodule `foo`, after importing `spam.foo`, `spam` will have an attribute `foo` which is bound to the submodule. Let's say you have the following directory structure:

```
spam/
    __init__.py
    foo.py
    bar.py
```

and `spam/__init__.py` has the following lines in it: