torch.package

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 1)

Unknown directive type "automodule".

```
.. automodule:: torch.package
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 2)

Unknown directive type "py:module".

```
.. py:module:: torch.package.analyze
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 4)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.package
```

`torch.package` adds support for creating hermetic packages containing arbitrary PyTorch code. These packages can be saved, shared, used to load and execute models at a later date or on a different machine, and can even be deployed to production using `torch::deploy`.

This document contains tutorials, how-to guides, explanations, and an API reference that will help you learn more about `torch.package` and how to use it.

Warning

This module depends on the `pickle` module which is is not secure. Only unpackage data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpackage data that could have come from an untrusted source, or that could have been tampered with.

For more information, review the documentation for the `pickle` module.

Tutorials

Packaging your first model

A tutorial that guides you through packaging and unpackaging a simple model is available on Colab. After completing this exercise, you will be familiar with the basic API for creating and using Torch packages.

How do I...

See what is inside a package?

Treat the package like a ZIP archive

The container format for a `torch.package` is ZIP, so any tools that work with standard ZIP files should work for exploring the contents. Some common ways to interact with ZIP files:

- `unzip my_package.pt` will unzip the `torch.package` archive to disk, where you can freely inspect its contents.

```
$ unzip my_package.pt && tree my_package
my_package
├── .data
│   ├── 94304870911616.storage
│   ├── 94304900784016.storage
│   ├── extern_modules
│   └── version
├── models
│   └── model_1.pkl
└── torchvision
    └── models
        ├── resnet.py
        └── utils.py
~ cd my_package && cat torchvision/models/resnet.py
...
```

- The Python `zipfile` module provides a standard way to read and write ZIP archive contents.

```
from zipfile import ZipFile
with ZipFile("my_package.pt") as myzip:
    file_bytes = myzip.read("torchvision/models/resnet.py")
    # edit file_bytes in some way
    myzip.writestr("torchvision/models/resnet.py", new_file_bytes)
```

- vim has the ability to natively read ZIP archives. You can even edit files and :`write` them back into the archive!

```
# add this to your .vimrc to treat `*.pt` files as zip files
au BufReadCmd *.pt call zip#Browse(expand("<amatch>"))

~ vi my_package.pt
```

Use the `file_structure()` API

:class:`PackageImporter` and :class:`PackageExporter` provide a `file_structure()` method, which will return a printable and queryable `Folder` object. The `Folder` object is a simple directory structure that you can use to explore the current contents of a `torch.package`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 96); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 96); *backlink*

Unknown interpreted text role "class".

The `Folder` object itself is directly printable and will print out a file tree representation. To filter what is returned, use the glob-style `include` and `exclude` filtering arguments.

```
with PackageExporter('my_package.pt') as pe:
    pe.save_pickle('models', 'model_1.pkl', mod)
    # can limit printed items with include/exclude args
    print(pe.file_structure(include=["**/utils.py", "**/*.pkl"], exclude="**/*.storages"))

importer = PackageImporter('my_package.pt')
print(importer.file_structure()) # will print out all files
```

Output:

```
# filtered with glob pattern:
```

```
#    include=["**/utils.py", "**/*.pkl"], exclude="**/*.storages"
─── my_package.pt
    ├── models
    │   └── model_1.pkl
    └── torchvision
        └── models
            └── utils.py

# all files
─── my_package.pt
    ├── .data
    │   ├── 94304870911616.storage
    │   ├── 94304900784016.storage
    │   ├── extern_modules
    │   └── version
    ├── models
    │   └── model_1.pkl
    └── torchvision
        └── models
            ├── resnet.py
            └── utils.py
```

You can also query `Folder` objects with the `has_file()` method.

```
exporter_file_structure = exporter.file_structure()
found: bool = exporter_file_structure.has_file("package_a/subpackage.py")
```

See why a given module was included as a dependency?

Say there is a given module `foo`, and you want to know why your :class:`PackageExporter` is pulling in `foo` as a dependency.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 155); *backlink*
>
> Unknown interpreted text role "class".

:meth:`PackageExporter.get_rdeps` will return all modules that directly depend on `foo`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 157); *backlink*
>
> Unknown interpreted text role "meth".

If you would like to see how a given module `src` depends on `foo`, the :meth:`PackageExporter.all_paths` method will return a DOT-formatted graph showing all the dependency paths between `src` and `foo`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 159); *backlink*
>
> Unknown interpreted text role "meth".

If you would just like to see the whole dependency graph of your :class:`PackageExporter`, you can use :meth:`PackageExporter.dependency_graph_string`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 162); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 162); *backlink*
>
> Unknown interpreted text role "meth".

Include arbitrary resources with my package and access them later?

:class:`PackageExporter` exposes three methods, `save_pickle`, `save_text` and `save_binary` that allow you to save Python objects, text, and binary data to a package.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 167); *backlink*

Unknown interpreted text role "class".

```
with torch.PackageExporter("package.pt") as exporter:
    # Pickles the object and saves to `my_resources/tens.pkl` in the archive.
    exporter.save_pickle("my_resources", "tensor.pkl", torch.randn(4))
    exporter.save_text("config_stuff", "words.txt", "a sample string")
    exporter.save_binary("raw_data", "binary", my_bytes)
```

:class:`PackageImporter` exposes complementary methods named `load_pickle`, `load_text` and `load_binary` that allow you to load Python objects, text and binary data from a package.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 180); *backlink*

Unknown interpreted text role "class".

```
importer = torch.PackageImporter("package.pt")
my_tensor = importer.load_pickle("my_resources", "tensor.pkl")
text = importer.load_text("config_stuff", "words.txt")
binary = importer.load_binary("raw_data", "binary")
```

Customize how a class is packaged?

`torch.package` allows for the customization of how classes are packaged. This behavior is accessed through defining the method `__reduce_package__` on a class and by defining a corresponding de-packaging function. This is similar to defining `__reduce__` for Python's normal pickling process.

Steps:

1. Define the method `__reduce_package__(self, exporter: PackageExporter)` on the target class. This method should do the work to save the class instance inside of the package, and should return a tuple of the corresponding de-packaging function with the arguments needed to invoke the de-packaging function. This method is called by the `PackageExporter` when it encounters an instance of the target class.
2. Define a de-packaging function for the class. This de-packaging function should do the work to reconstruct and return an instance of the class. The function signature's first parameter should be a `PackageImporter` instance, and the rest of the parameters are user defined.

```
# foo.py [Example of customizing how class Foo is packaged]
from torch.package import PackageExporter, PackageImporter
import time


class Foo:
    def __init__(self, my_string: str):
        super().__init__()
        self.my_string = my_string
        self.time_imported = 0
        self.time_exported = 0

    def __reduce_package__(self, exporter: PackageExporter):
        """
        Called by ``torch.package.PackageExporter``'s Pickler's ``persistent_id`` when
        saving an instance of this object. This method should do the work to save this
        object inside of the ``torch.package`` archive.

        Returns function w/ arguments to load the object from a
        ``torch.package.PackageImporter``'s Pickler's ``persistent_load`` function.
        """

        # use this pattern to ensure no naming conflicts with normal dependencies,
        # anything saved under this module name shouldn't conflict with other
        # items in the package
        generated_module_name = f"foo-generated._{exporter.get_unique_id()}"
        exporter.save_text(
            generated_module_name,
            "foo.txt",
            self.my_string + ", with exporter modification!",
        )
        time_exported = time.clock_gettime(1)

        # returns de-packaging function w/ arguments to invoke with
        return (unpackage_foo, (generated_module_name, time_exported,))


def unpackage_foo(
    importer: PackageImporter, generated_module_name: str, time_exported: float
) -> Foo:
```

```
        """
        Called by ``torch.package.PackageImporter``'s Pickler's ``persistent_load`` function
        when depickling a Foo object.
        Performs work of loading and returning a Foo instance from a ``torch.package`` archive.
        """
        time_imported = time.clock_gettime(1)
        foo = Foo(importer.load_text(generated_module_name, "foo.txt"))
        foo.time_imported = time_imported
        foo.time_exported = time_exported
        return foo

# example of saving instances of class Foo

import torch
from torch.package import PackageImporter, PackageExporter
import foo

foo_1 = foo.Foo("foo_1 initial string")
foo_2 = foo.Foo("foo_2 initial string")
with PackageExporter('foo_package.pt') as pe:
    # save as normal, no extra work necessary
    pe.save_pickle('foo_collection', 'foo1.pkl', foo_1)
    pe.save_pickle('foo_collection', 'foo2.pkl', foo_2)
    print(pe.file_structure())

pi = PackageImporter('foo_package.pt')
imported_foo = pi.load_pickle('foo_collection', 'foo1.pkl')
print(f"foo_1 string: '{imported_foo.my_string}'")
print(f"foo_1 export time: {imported_foo.time_exported}")
print(f"foo_1 import time: {imported_foo.time_imported}")

# output of running above script
─── foo_package
    ├── foo-generated
    │   ├── _0
    │   │   └── foo.txt
    │   └── _1
    │       └── foo.txt
    ├── foo_collection
    │   ├── foo1.pkl
    │   └── foo2.pkl
    └── foo.py

foo_1 string: 'foo_1 initial string, with reduction modification!'
foo_1 export time: 9857706.650140837
foo_1 import time: 9857706.652698385
```

Test in my source code whether or not it is executing inside a package?

A :class:`PackageImporter` will add the attribute `__torch_package__` to every module that it initializes. Your code can check for the presence of this attribute to determine whether it is executing in a packaged context or not.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, **line 302**); ***backlink***

Unknown interpreted text role "class".

```
# In foo/bar.py:

if "__torch_package__" in dir():  # true if the code is being loaded from a package
    def is_in_package():
        return True

    UserException = Exception
else:
    def is_in_package():
        return False

    UserException = UnpackageableException
```

Now, the code will behave differently depending on whether it's imported normally through your Python environment or imported from a `torch.package`.

```
from foo.bar import is_in_package

print(is_in_package())  # False

loaded_module = PackageImporter(my_pacakge).import_module("foo.bar")
loaded_module.is_in_package()  # True
```

Warning: in general, it's bad practice to have code that behaves differently depending on whether it's packaged or not. This can lead to hard-to-debug issues that are sensitive to how you imported your code. If your package is intended to be heavily used, consider restructuring your code so that it behaves the same way no matter how it was loaded.

Patch code into a package?

:class:`PackageExporter` offers a `save_source_string()` method that allows one to save arbitrary Python source code to a module of your choosing.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 343); *backlink*
>
> Unknown interpreted text role "class".

```
with PackageExporter(f) as exporter:
    # Save the my_module.foo available in your current Python environment.
    exporter.save_module("my_module.foo")

    # This saves the provided string to my_module/foo.py in the package archive.
    # It will override the my_module.foo that was previously saved.
    exporter.save_source_string("my_module.foo", textwrap.dedent(
        """\
        def my_function():
            print('hello world')
        """
    ))

    # If you want to treat my_module.bar as a package
    # (e.g. save to `my_module/bar/__init__.py` instead of `my_module/bar.py)
    # pass is_package=True,
    exporter.save_source_string("my_module.bar",
                                "def foo(): print('hello')\n",
                                is_package=True)

importer = PackageImporter(f)
importer.import_module("my_module.foo").my_function()  # prints 'hello world'
```

Access package contents from packaged code?

:class:`PackageImporter` implements the importlib.resources API for accessing resources from inside a package.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 374); *backlink*
>
> Unknown interpreted text role "class".

```
with PackageExporter(f) as exporter:
    # saves text to one/a.txt in the archive
    exporter.save_text("my_resource", "a.txt", "hello world!")
    # saves the tensor to my_pickle/obj.pkl
    exporter.save_pickle("my_pickle", "obj.pkl", torch.ones(2, 2))

    # see below for module contents
    exporter.save_module("foo")
    exporter.save_module("bar")
```

The `importlib.resources` API allows access to resources from within packaged code.

```
# foo.py:
import importlib.resources
import my_resource

# returns "hello world!"
def get_my_resource():
    return importlib.resources.read_text(my_resource, "a.txt")
```

Using `importlib.resources` is the recommended way to access package contents from within packaged code, since it complies with the Python standard. However, it is also possible to access the parent :class:`PackageImporter` instance itself from within packaged code.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 406); *backlink*
>
> Unknown interpreted text role "class".

```
# bar.py:
import torch_package_importer # this is the PackageImporter that imported this module.

# Prints "hello world!", equivalient to importlib.resources.read_text
def get_my_resource():
    return torch_package_importer.load_text("my_resource", "a.txt")

# You also do things that the importlib.resources API does not support, like loading
# a pickled object from the package.
def get_my_pickle():
    return torch_package_importer.load_pickle("my_pickle", "obj.pkl")
```

Distinguish between packaged code and non-packaged code?

To tell if an object's code is from a `torch.package`, use the `torch.package.is_from_package()` function. Note: if an object is from a package but its definition is from a module marked `extern` or from `stdlib`, this check will return `False`.

```
importer = PackageImporter(f)
mod = importer.import_module('foo')
obj = importer.load_pickle('model', 'model.pkl')
txt = importer.load_text('text', 'my_test.txt')

assert is_from_package(mod)
assert is_from_package(obj)
assert not is_from_package(txt) # str is from stdlib, so this will return False
```

Re-export an imported object?

To re-export an object that was previously imported by a :class:`PackageImporter`, you must make the new :class:`PackageExporter` aware of the original :class:`PackageImporter` so that it can find source code for your object's dependencies.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 447); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 447); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 447); *backlink*
>
> Unknown interpreted text role "class".

```
importer = PackageImporter(f)
obj = importer.load_pickle("model", "model.pkl")

# re-export obj in a new package
with PackageExporter(f2, importer=(importer, sys_importer)) as exporter:
    exporter.save_pickle("model", "model.pkl", obj)
```

Package a TorchScript module?

To package a TorchScript model, use the same `save_pickle` and `load_pickle` APIs as you would with any other object. Saving TorchScript objects that are attributes or submodules is supported as well with no extra work.

```
# save TorchScript just like any other object
with PackageExporter(file_name) as e:
    e.save_pickle("res", "script_model.pkl", scripted_model)
    e.save_pickle("res", "mixed_model.pkl", python_model_with_scripted_submodule)
# load as normal
importer = PackageImporter(file_name)
loaded_script = importer.load_pickle("res", "script_model.pkl")
loaded_mixed = importer.load_pickle("res", "mixed_model.pkl"
```

Explanation

`torch.package` Format Overview

A `torch.package` file is a ZIP archive which conventionally uses the `.pt` extension. Inside the ZIP archive, there are two kinds of files:

- Framework files, which are placed in the `.data/`.
- User files, which is everything else.

As an example, this is what a fully packaged ResNet model from `torchvision` looks like:

```
resnet
├── .data  # All framework-specific data is stored here.
│   │      # It's named to avoid conflicts with user-serialized code.
│   ├── 94286146172688.storage  # tensor data
│   ├── 94286146172784.storage
│   ├── extern_modules  # text file with names of extern modules (e.g. 'torch')
│   ├── version         # version metadata
│   ├── ...
├── model  # the pickled model
│   └── model.pkl
└── torchvision  # all code dependencies are captured as source files
    └── models
        ├── resnet.py
        └── utils.py
```

Framework files

The `.data/` directory is owned by torch.package, and its contents are considered to be a private implementation detail. The `torch.package` format makes no guarantees about the contents of `.data/`, but any changes made will be backward compatible (that is, newer version of PyTorch will always be able to load older `torch.packages`).

Currently, the `.data/` directory contains the following items:

- `version`: a version number for the serialized format, so that the `torch.package` import infrastructures knows how to load this package.
- `extern_modules`: a list of modules that are considered `extern:class:`PackageImporter`. ``extern` modules will be imported using the loading environment's system importer.
- `*.storage`: serialized tensor data.

```
.data
├── 94286146172688.storage
├── 94286146172784.storage
├── extern_modules
├── version
├── ...
```

User files

All other files in the archive were put there by a user. The layout is identical to a Python regular package. For a deeper dive in how Python packaging works, please consult this essay (it's slightly out of date, so double-check implementation details with the Python reference documentation).

```
<package root>
├── model  # the pickled model
│   └── model.pkl
├── another_package
│   ├── __init__.py
│   ├── foo.txt         # a resource file , see importlib.resources
│   └── ...
└── torchvision
    └── models
        ├── resnet.py   # torchvision.models.resnet
        └── utils.py    # torchvision.models.utils
```

How `torch.package` finds your code's dependencies

Analyzing an object's dependencies

When you issue a `save_pickle(obj, ...)` call, :class:`PackageExporter` will pickle the object normally. Then, it uses the `pickletools` standard library module to parse the pickle bytecode.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 559); *backlink***
>
> Unknown interpreted text role "class".

In a pickle, an object is saved along with a `GLOBAL` opcode that describes where to find the implementation of the object's type, like:

```
GLOBAL 'torchvision.models.resnet Resnet`
```

The dependency resolver will gather up all `GLOBAL` ops and mark them as dependencies of your pickled object. For more information about pickling and the pickle format, please consult the Python docs.

Analyzing a module's dependencies

When a Python module is identified as a dependency, `torch.package` walks the module's python AST representation and looks for import statements with full support for the standard forms: `from x import y`, `import z`, `from w import v as u`, etc. When one of these import statements are encountered, `torch.package` registers the imported modules as dependencies that are then themselves parsed in the same AST walking way.

Note: AST parsing has limited support for the `__import__(...)` syntax and does not support `importlib.import_module` calls. In general, you should not expect dynamic imports to be detected by `torch.package`.

Dependency Management

`torch.package` automatically finds the Python modules that your code and objects depend on. This process is called dependency resolution. For each module that the dependency resolver finds, you must specify an *action* to take.

The allowed actions are:

- `intern`: put this module into the package.
- `extern`: declare this module as an external dependency of the package.
- `mock`: stub out this module.
- `deny`: depending on this module will raise an error during package export.

Finally, there is one more important action that is not technically part of `torch.package`:

- Refactoring: remove or change the dependencies in your code.

Note that actions are only defined on entire Python modules. There is no way to package "just" a function or class from module and leave the rest out. This is by design. Python does not offer clean boundaries between objects defined in a module. The only defined unit of dependency organization is a module, so that's what `torch.package` uses.

Actions are applied to modules using patterns. Patterns can either be module names (`"foo.bar"`) or globs (like `"foo.**"`). You associate a pattern with an action using methods on :class:`PackageImporter`, e.g.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 603); *backlink*

Unknown interpreted text role "class".

```
my_exporter.intern("torchvision.**")
my_exporter.extern("numpy")
```

If a module matches a pattern, the corresponding action is applied to it. For a given module, patterns will be checked in the order that they were defined, and the first action will be taken.

`intern`

If a module is `intern`-ed, it will be placed into the package.

This action is your model code, or any related code you want to package. For example, if you are trying to package a ResNet from `torchvision`, you will need to `intern` the module torchvision.models.resnet.

On package import, when your packaged code tries to import an `intern`-ed module, PackageImporter will look inside your package for that module. If it can't find that module, an error will be raised. This ensures that each :class:`PackageImporter` is isolated from the loading environment—even if you have `my_interned_module` available in both your package and the loading environment, :class:`PackageImporter` will only use the version in your package.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 624); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 624); *backlink*

Unknown interpreted text role "class".

Note: Only Python source modules can be `intern`-ed. Other kinds of modules, like C extension modules and bytecode modules, will raise an error if you attempt to `intern` them. These kinds of modules need to be `mock`-ed or `extern`-ed.

`extern`

If a module is `extern`-ed, it will not be packaged. Instead, it will be added to a list of external dependencies for this package. You can find this list on `package_exporter.extern_modules`.

On package import, when time packaged code tries to import an `extern`-ed module, :class:`PackageImporter` will use the default Python importer to find that module, as if you did `importlib.import_module("my_externed_module")`. If it can't find that module, an error will be raised.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 638); *backlink*
>
> Unknown interpreted text role "class".

In this way, you can depend on third-party libraries like `numpy` and `scipy` from within your package without having to package them too.

Warning: If any external library changes in a backwards-incompatible way, your package may fail to load. If you need long-term reproducibility for your package, try to limit your use of `extern`.

`mock`

If a module is `mock`-ed, it will not be packaged. Instead a stub module will be packaged in its place. The stub module will allow you to retrieve objects from it (so that `from my_mocked_module import foo` will not error), but any use of that object will raise a `NotImplementedError`.

`mock` should be used for code that you "know" will not be needed in the loaded package, but you still want to available for use in non-packaged contents. For example, initialization/configuration code, or code only used for debugging/training.

Warning: In general, `mock` should be used as a last resort. It introduces behavioral differences between packaged code and non-packaged code, which may lead to later confusion. Prefer instead to refactor your code to remove unwanted dependencies.

Refactoring

The best way to manage dependencies is to not have dependencies at all! Often, code can be refactored to remove unnecessary dependencies. Here are some guidelines for writing code with clean dependencies (which are also generally good practices!):

Include only what you use. Do not leave unused imports in our code. The dependency resolver is not smart enough to tell that they are indeed unused, and will try to process them.

Qualify your imports. For example, instead of writing import foo and later using `foo.bar.baz`, prefer to write `from foo.bar import baz`. This more precisely specifies your real dependency (`foo.bar`) and lets the dependency resolver know you don't need all of `foo`.

Split up large files with unrelated functionality into smaller ones. If your `utils` module contains a hodge-podge of unrelated functionality, any module that depends on `utils` will need to pull in lots of unrelated dependencies, even if you only needed a small part of it. Prefer instead to define single-purpose modules that can be packaged independently of one another.

Patterns

Patterns allow you to specify groups of modules with a convenient syntax. The syntax and behavior of patterns follows the Bazel/Buck glob().

A module that we are trying to match against a pattern is called a candidate. A candidate is composed of a list of segments separated by a separator string, e.g. `foo.bar.baz`.

A pattern contains one or more segments. Segments can be:

- A literal string (e.g. `foo`), which matches exactly.
- A string containing a wildcard (e.g. `torch`, or `foo*baz*`). The wildcard matches any string, including the empty string.
- A double wildcard (`**`). This matches against zero or more complete segments.

Examples:

- `torch.**`: matches `torch` and all its submodules, e.g. `torch.nn` and `torch.nn.functional`.
- `torch.*`: matches `torch.nn` or `torch.functional`, but not `torch.nn.functional` or `torch`
- `torch*.**`: matches `torch`, `torchvision`, and all of their submodules

When specifying actions, you can pass multiple patterns, e.g.

```
exporter.intern(["torchvision.models.**", "torchvision.utils.**"])
```

A module will match against this action if it matches any of the patterns.

You can also specify patterns to exclude, e.g.

```
exporter.mock("**", exclude=["torchvision.**"])
```

A module will not match against this action if it matches any of the exclude patterns. In this example, we are mocking all modules except torchvision and its submodules.

When a module could potentially match against multiple actions, the first action defined will be taken.

torch.package sharp edges

Avoid global state in your modules

Python makes it really easy to bind objects and run code at module-level scope. This is generally fine—after all, functions and classes are bound to names this way. However, things become more complicated when you define an object at module scope with the intention of mutating it, introducing mutable global state.

Mutable global state is quite useful—it can reduce boilerplate, allow for open registration into tables, etc. But unless employed very carefully, it can cause complications when used with torch.package.

Every :class:`PackageImporter` creates an independent environment for its contents. This is nice because it means we load multiple packages and ensure they are isolated from each other, but when modules are written in a way that assumes shared mutable global state, this behavior can create hard-to-debug errors.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst, line 730); *backlink*
>
> Unknown interpreted text role "class".

Types are not shared between packages and the loading environment

Any class that you import from a :class:`PackageImporter` will be a version of the class specific to that importer. For example:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst, line 736); *backlink*
>
> Unknown interpreted text role "class".

```
from foo import MyClass

my_class_instance = MyClass()

with PackageExporter(f) as exporter:
    exporter.save_module("foo")

importer = PackageImporter(f)
imported_MyClass = importer.import_module("foo").MyClass

assert isinstance(my_class_instance, MyClass)  # works
assert isinstance(my_class_instance, imported_MyClass)  # ERROR!
```

In this example, MyClass and import_MyClass are *not the same type*. In this specific example, MyClass and import_MyClass have exactly the same implementation, so you might thing it's okay to consider them the same class. But consider the situation where import_MyClass is coming from an older package with an entirely different implementation of MyClass — in that case, it's unsafe to consider them the same class.

Under the hood, each importer has a prefix that allows it to uniquely identify classes:

```
print(MyClass.__name__)  # prints "foo.MyClass"
print(imported_MyClass.__name__)  # prints <torch_package_0>.foo.MyClass
```

That means you should not expect isinstance checks to work when one of the arguments if from a package and the other is not. If you need this functionality, consider the following options:

- Doing duck typing (just using the class instead of explicitly checking that it is of a given type).
- Make the typing relationship an explicit part of the class contract. For example, you can add an attribute tag self.handler = "handle_me_this_way" and have client code check for the value of handler instead of checking the type directly.

How torch.package keeps packages isolated from each other

Each :class:`PackageImporter` instance creates an independent, isolated environment for its modules and objects. Modules in a package can only import other packaged modules, or modules marked extern. If you use multiple :class:`PackageImporter` instances to load a single package, you will get multiple independent environments that do not interact.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst, line 777); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 777); *backlink*
>
> Unknown interpreted text role "class".

This is achieved by extending Python's import infrastructure with a custom importer. :class:`PackageImporter` provides the same core API as the `importlib` importer; namely, it implements the `import_module` and `__import__` methods.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 781); *backlink*
>
> Unknown interpreted text role "class".

When you invoke :meth:`PackageImporter.import_module`, :class:`PackageImporter` will construct and return a new module, much as the system importer does. However, :class:`PackageImporter` patches the returned module to use `self` (i.e. that :class:`PackageImporter` instance) to fulfill future import requests by looking in the package rather than searching the user's Python environment.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 784); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 784); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 784); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 784); *backlink*
>
> Unknown interpreted text role "class".

Mangling

To avoid confusion ("is this `foo.bar` object the one from my package, or the one from my Python environment?"), :class:`PackageImporter` mangles the `__name__` and `__file__` of all imported modules, by adding a *mangle prefix* to them.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 790); *backlink*
>
> Unknown interpreted text role "class".

For `__name__`, a name like `torchvision.models.resnet18` becomes `<torch_package_0>.torchvision.models.resnet18`.

For `__file__`, a name like `torchvision/models/resnet18.py` becomes `<torch_package_0>.torchvision/modules/resnet18.py`.

Name mangling helps avoid inadvertent punning of module names between different packages, and helps you debug by making stack traces and print statements more clearly show whether they are referring to packaged code or not. For developer-facing details about mangling, consult `mangling.md` in `torch/package/`.

API Reference

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 804)
>
> Unknown directive type "autoclass".
>
> ```
> .. autoclass:: torch.package.PackagingError
> ```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 806)

Unknown directive type "autoclass".

```
.. autoclass:: torch.package.EmptyMatchError
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 808)

Unknown directive type "autoclass".

```
.. autoclass:: torch.package.PackageExporter
  :members:

  .. automethod:: __init__
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 813)

Unknown directive type "autoclass".

```
.. autoclass:: torch.package.PackageImporter
  :members:

  .. automethod:: __init__
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]package.rst`, line 818)

Unknown directive type "autoclass".

```
.. autoclass:: torch.package.Directory
  :members:
```