```
  Downloading ...
Installing collected packages: pytz
Successfully installed pytz-2018.9
```

pip is best used together with the built-in module venv to consistently track sets of packages to install for your projects (see Item 83: "Use Virtual Environments for Isolated and Reproducible Dependencies"). You can also create your own PyPI packages to share with the Python community or host your own private package repositories for use with pip.

Each module in the PyPI has its own software license. Most of the packages, especially the popular ones, have free or open source licenses (see https://opensource.org for details). In most cases, these licenses allow you to include a copy of the module with your program; when in doubt, talk to a lawyer.

Things to Remember

- The Python Package Index (PyPI) contains a wealth of common packages that are built and maintained by the Python community.
- pip is the command-line tool you can use to install packages from PyPI.
- The majority of PyPI modules are free and open source software.

Item 83: Use Virtual Environments for Isolated and Reproducible Dependencies

Building larger and more complex programs often leads you to rely on various packages from the Python community (see Item 82: "Know Where to Find Community-Built Modules"). You'll find yourself running the python3 -m pip command-line tool to install packages like pytz, numpy, and many others.

The problem is that, by default, pip installs new packages in a global location. That causes all Python programs on your system to be affected by these installed modules. In theory, this shouldn't be an issue. If you install a package and never import it, how could it affect your programs?

The trouble comes from transitive dependencies: the packages that the packages you install depend on. For example, you can see what the Sphinx package depends on after installing it by asking pip:

```
$ python3 -m pip show Sphinx
Name: Sphinx
```

```
Version: 2.1.2
Summary: Python documentation generator
Location: /usr/local/lib/python3.8/site-packages
Requires: alabaster, imagesize, requests,
➥sphinxcontrib-applehelp, sphinxcontrib-qthelp,
➥Jinja2, setuptools, sphinxcontrib-jsmath,
➥sphinxcontrib-serializinghtml, Pygments, snowballstemmer,
➥packaging, sphinxcontrib-devhelp, sphinxcontrib-htmlhelp,
➥babel, docutils
Required-by:
```

If you install another package like `flask`, you can see that it, too, depends on the `Jinja2` package:

```
$ python3 -m pip show flask
Name: Flask
Version: 1.0.3
Summary: A simple framework for building complex web applications.
Location: /usr/local/lib/python3.8/site-packages
Requires: itsdangerous, click, Jinja2, Werkzeug
Required-by:
```

A dependency conflict can arise as `Sphinx` and `flask` diverge over time. Perhaps right now they both require the same version of `Jinja2`, and everything is fine. But six months or a year from now, `Jinja2` may release a new version that makes breaking changes to users of the library. If you update your global version of `Jinja2` with `python3 -m pip install --upgrade Jinja2`, you may find that `Sphinx` breaks, while `flask` keeps working.

The cause of such breakage is that Python can have only a single global version of a module installed at a time. If one of your installed packages must use the new version and another package must use the old version, your system isn't going to work properly; this situation is often called *dependency hell*.

Such breakage can even happen when package maintainers try their best to preserve API compatibility between releases (see Item 85: "Use Packages to Organize Modules and Provide Stable APIs"). New versions of a library can subtly change behaviors that API-consuming code relies on. Users on a system may upgrade one package to a new version but not others, which could break dependencies. If you're not careful there's a constant risk of the ground moving beneath your feet.

These difficulties are magnified when you collaborate with other developers who do their work on separate computers. It's best to assume the worst: that the versions of Python and global packages

that they have installed on their machines will be slightly different from yours. This can cause frustrating situations such as a codebase working perfectly on one programmer's machine and being completely broken on another's.

The solution to all of these problems is using a tool called `venv`, which provides *virtual environments*. Since Python 3.4, `pip` and the `venv` module have been available by default along with the Python installation (accessible with `python -m venv`).

`venv` allows you to create isolated versions of the Python environment. Using `venv`, you can have many different versions of the same package installed on the same system at the same time without conflicts. This means you can work on many different projects and use many different tools on the same computer. `venv` does this by installing explicit versions of packages and their dependencies into completely separate directory structures. This makes it possible to reproduce a Python environment that you know will work with your code. It's a reliable way to avoid surprising breakages.

Using `venv` on the Command Line

Here's a quick tutorial on how to use `venv` effectively. Before using the tool, it's important to note the meaning of the `python3` command line on your system. On my computer, `python3` is located in the `/usr/local/bin` directory and evaluates to version 3.8.0 (see Item 1: "Know Which Version of Python You're Using"):

```
$ which python3
/usr/local/bin/python3
$ python3 --version
Python 3.8.0
```

To demonstrate the setup of my environment, I can test that running a command to import the `pytz` module doesn't cause an error. This works because I already have the `pytz` package installed as a global module:

```
$ python3 -c 'import pytz'
$
```

Now, I use `venv` to create a new virtual environment called `myproject`. Each virtual environment must live in its own unique directory. The result of the command is a tree of directories and files that are used to manage the virtual environment:

```
$ python3 -m venv myproject
$ cd myproject
```

```
$ ls
bin     include     lib     pyvenv.cfg
```

To start using the virtual environment, I use the `source` command from my shell on the `bin/activate` script. `activate` modifies all of my environment variables to match the virtual environment. It also updates my command-line prompt to include the virtual environment name ("myproject") to make it extremely clear what I'm working on:

```
$ source bin/activate
(myproject)$
```

On Windows the same script is available as:

```
C:\> myproject\Scripts\activate.bat
(myproject) C:>
```

Or with PowerShell as:

```
PS C:\> myproject\Scripts\activate.ps1
(myproject) PS C:>
```

After activation, the path to the `python3` command-line tool has moved to within the virtual environment directory:

```
(myproject)$ which python3
/tmp/myproject/bin/python3
(myproject)$ ls -l /tmp/myproject/bin/python3
... -> /usr/local/bin/python3.8
```

This ensures that changes to the outside system will not affect the virtual environment. Even if the outer system upgrades its default `python3` to version 3.9, my virtual environment will still explicitly point to version 3.8.

The virtual environment I created with `venv` starts with no packages installed except for `pip` and `setuptools`. Trying to use the `pytz` package that was installed as a global module in the outside system will fail because it's unknown to the virtual environment:

```
(myproject)$ python3 -c 'import pytz'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'pytz'
```

I can use the `pip` command-line tool to install the `pytz` module into my virtual environment:

```
(myproject)$ python3 -m pip install pytz
Collecting pytz
```

```
  Downloading ...
Installing collected packages: pytz
Successfully installed pytz-2019.1
```

Once it's installed, I can verify that it's working by using the same test `import` command:

```
(myproject)$ python3 -c 'import pytz'
(myproject)$
```

When I'm done with a virtual environment and want to go back to my default system, I use the `deactivate` command. This restores my environment to the system defaults, including the location of the `python3` command-line tool:

```
(myproject)$ which python3
/tmp/myproject/bin/python3
(myproject)$ deactivate
$ which python3
/usr/local/bin/python3
```

If I ever want to work in the `myproject` environment again, I can just run `source bin/activate` in the directory as before.

Reproducing Dependencies

Once you are in a virtual environment, you can continue installing packages in it with `pip` as you need them. Eventually, you might want to copy your environment somewhere else. For example, say that I want to reproduce the development environment from my workstation on a server in a datacenter. Or maybe I want to clone someone else's environment on my own machine so I can help debug their code.

`venv` makes such tasks easy. I can use the `python3 -m pip freeze` command to save all of my explicit package dependencies into a file (which, by convention, is named `requirements.txt`):

```
(myproject)$ python3 -m pip freeze > requirements.txt
(myproject)$ cat requirements.txt
certifi==2019.3.9
chardet==3.0.4
idna==2.8
numpy==1.16.2
pytz==2018.9
requests==2.21.0
urllib3==1.24.1
```

Now, imagine that I'd like to have another virtual environment that matches the `myproject` environment. I can create a new directory as before by using `venv` and `activate` it:

```
$ python3 -m venv otherproject
$ cd otherproject
$ source bin/activate
(otherproject)$
```

The new environment will have no extra packages installed:

```
(otherproject)$ python3 -m pip list
Package    Version
---------- -------
pip        10.0.1
setuptools 39.0.1
```

I can install all of the packages from the first environment by running `python3 -m pip install` on the `requirements.txt` that I generated with the `python3 -m pip freeze` command:

```
(otherproject)$ python3 -m pip install -r /tmp/myproject/
➥requirements.txt
```

This command cranks along for a little while as it retrieves and installs all of the packages required to reproduce the first environment. When it's done, I can list the set of installed packages in the second virtual environment and should see the same list of dependencies found in the first virtual environment:

```
(otherproject)$ python3 -m pip list
Package    Version
---------- --------
certifi    2019.3.9
chardet    3.0.4
idna       2.8
numpy      1.16.2
pip        10.0.1
pytz       2018.9
requests   2.21.0
setuptools 39.0.1
urllib3    1.24.1
```

Using a `requirements.txt` file is ideal for collaborating with others through a revision control system. You can commit changes to your code at the same time you update your list of package dependencies, ensuring that they move in lockstep. However, it's important to note that the specific version of Python you're using is *not* included in the `requirements.txt` file, so that must be managed separately.

The gotcha with virtual environments is that moving them breaks everything because all of the paths, like the `python3` command-line tool, are hard-coded to the environment's install directory. But ultimately this limitation doesn't matter. The whole purpose of virtual environments is to make it easy to reproduce a setup. Instead of moving a virtual environment directory, just use `python3 -m pip freeze` on the old one, create a new virtual environment somewhere else, and reinstall everything from the `requirements.txt` file.

Things to Remember

- Virtual environments allow you to use `pip` to install many different versions of the same package on the same machine without conflicts.
- Virtual environments are created with `python -m venv`, enabled with `source bin/activate`, and disabled with `deactivate`.
- You can dump all of the requirements of an environment with `python3 -m pip freeze`. You can reproduce an environment by running `python3 -m pip install -r requirements.txt`.

Item 84: Write Docstrings for Every Function, Class, and Module

Documentation in Python is extremely important because of the dynamic nature of the language. Python provides built-in support for attaching documentation to blocks of code. Unlike with many other languages, the documentation from a program's source code is directly accessible as the program runs.

For example, you can add documentation by providing a *docstring* immediately after the `def` statement of a function:

```
def palindrome(word):
    """Return True if the given word is a palindrome."""
    return word == word[::-1]

assert palindrome('tacocat')
assert not palindrome('banana')
```

You can retrieve the docstring from within the Python program by accessing the function's `__doc__` special attribute:

```
print(repr(palindrome.__doc__))
```

```
>>>
'Return True if the given word is a palindrome.'
```

You can also use the built-in `pydoc` module from the command line to run a local web server that hosts all of the Python documentation that's accessible to your interpreter, including modules that you've written:

```
$ python3 -m pydoc -p 1234
Server ready at http://localhost:1234/
Server commands: [b]rowser, [q]uit
server> b
```

Docstrings can be attached to functions, classes, and modules. This connection is part of the process of compiling and running a Python program. Support for docstrings and the `__doc__` attribute has three consequences:

- The accessibility of documentation makes interactive development easier. You can inspect functions, classes, and modules to see their documentation by using the `help` built-in function. This makes the Python interactive interpreter (the Python "shell") and tools like IPython Notebook (https://ipython.org) a joy to use while you're developing algorithms, testing APIs, and writing code snippets.
- A standard way of defining documentation makes it easy to build tools that convert the text into more appealing formats (like HTML). This has led to excellent documentation-generation tools for the Python community, such as Sphinx (https://www.sphinx-doc.org). It has also enabled community-funded sites like Read the Docs (https://readthedocs.org) that provide free hosting of beautiful-looking documentation for open source Python projects.
- Python's first-class, accessible, and good-looking documentation encourages people to write more documentation. The members of the Python community have a strong belief in the importance of documentation. There's an assumption that "good code" also means well-documented code. This means that you can expect most open source Python libraries to have decent documentation.

To participate in this excellent culture of documentation, you need to follow a few guidelines when you write docstrings. The full details are discussed online in PEP 257 (https://www.python.org/dev/peps/pep-0257/). There are a few best practices you should be sure to follow.

Documenting Modules

Each module should have a top-level docstring—a string literal that is the first statement in a source file. It should use three double quotes (`"""`). The goal of this docstring is to introduce the module and its contents.

The first line of the docstring should be a single sentence describing the module's purpose. The paragraphs that follow should contain the details that all users of the module should know about its operation. The module docstring is also a jumping-off point where you can highlight important classes and functions found in the module.

Here's an example of a module docstring:

```
# words.py
#!/usr/bin/env python3
"""Library for finding linguistic patterns in words.

Testing how words relate to each other can be tricky sometimes!
This module provides easy ways to determine when words you've
found have special properties.

Available functions:
- palindrome: Determine if a word is a palindrome.
- check_anagram: Determine if two words are anagrams.
...
"""

...
```

If the module is a command-line utility, the module docstring is also a great place to put usage information for running the tool.

Documenting Classes

Each class should have a class-level docstring. This largely follows the same pattern as the module-level docstring. The first line is the single-sentence purpose of the class. Paragraphs that follow discuss important details of the class's operation.

Important public attributes and methods of the class should be highlighted in the class-level docstring. It should also provide guidance to subclasses on how to properly interact with protected attributes (see Item 42: "Prefer Public Attributes Over Private Ones") and the superclass's methods.

Here's an example of a class docstring:

```
class Player:
    """Represents a player of the game.

    Subclasses may override the 'tick' method to provide
    custom animations for the player's movement depending
    on their power level, etc.
```

```
    Public attributes:
    - power: Unused power-ups (float between 0 and 1).
    - coins: Coins found during the level (integer).
    """
    ...
```

Documenting Functions

Each public function and method should have a docstring. This follows the same pattern as the docstrings for modules and classes. The first line is a single-sentence description of what the function does. The paragraphs that follow should describe any specific behaviors and the arguments for the function. Any return values should be mentioned. Any exceptions that callers must handle as part of the function's interface should be explained (see Item 20: "Prefer Raising Exceptions to Returning `None`" for how to document raised exceptions).

Here's an example of a function docstring:

```
def find_anagrams(word, dictionary):
    """Find all anagrams for a word.

    This function only runs as fast as the test for
    membership in the 'dictionary' container.

    Args:
        word: String of the target word.
        dictionary: collections.abc.Container with all
            strings that are known to be actual words.

    Returns:
        List of anagrams that were found. Empty if
        none were found.
    """
    ...
```

There are also some special cases in writing docstrings for functions that are important to know:

- If a function has no arguments and a simple return value, a single-sentence description is probably good enough.
- If a function doesn't return anything, it's better to leave out any mention of the return value instead of saying "returns None."
- If a function's interface includes raising exceptions (see Item 20: "Prefer Raising Exceptions to Returning `None`" for an example), its docstring should describe each exception that's raised and when it's raised.

- If you don't expect a function to raise an exception during normal operation, don't mention that fact.
- If a function accepts a variable number of arguments (see Item 22: "Reduce Visual Noise with Variable Positional Arguments") or keyword arguments (see Item 23: "Provide Optional Behavior with Keyword Arguments"), use `*args` and `**kwargs` in the documented list of arguments to describe their purpose.
- If a function has arguments with default values, those defaults should be mentioned (see Item 24: "Use `None` and Docstrings to Specify Dynamic Default Arguments").
- If a function is a generator (see Item 30: "Consider Generators Instead of Returning Lists"), its docstring should describe what the generator yields when it's iterated.
- If a function is an asynchronous coroutine (see Item 60: "Achieve Highly Concurrent I/O with Coroutines"), its docstring should explain when it will stop execution.

Using Docstrings and Type Annotations

Python now supports type annotations for a variety of purposes (see Item 90: "Consider Static Analysis via `typing` to Obviate Bugs" for how to use them). The information they contain may be redundant with typical docstrings. For example, here is the function signature for `find_anagrams` with type annotations applied:

```
from typing import Container, List

def find_anagrams(word: str,
                  dictionary: Container[str]) -> List[str]:
    ...
```

There is no longer a need to specify in the docstring that the `word` argument is a string, since the type annotation has that information. The same goes for the `dictionary` argument being a `collections.abc.Container`. There's no reason to mention that the return type will be a `list`, since this fact is clearly annotated. And when no anagrams are found, the return value still must be a `list`, so it's implied that it will be empty; that doesn't need to be noted in the docstring. Here, I write the same function signature from above along with the docstring that has been shortened accordingly:

```
def find_anagrams(word: str,
                  dictionary: Container[str]) -> List[str]:
    """Find all anagrams for a word.
```

```
    This function only runs as fast as the test for
    membership in the 'dictionary' container.

    Args:
        word: Target word.
        dictionary: All known actual words.

    Returns:
        Anagrams that were found.
    """
    ...
```

The redundancy between type annotations and docstrings should be similarly avoided for instance fields, class attributes, and methods. It's best to have type information in only one place so there's less risk that it will skew from the actual implementation.

Things to Remember

- Write documentation for every module, class, method, and function using docstrings. Keep them up-to-date as your code changes.
- For modules: Introduce the contents of a module and any important classes or functions that all users should know about.
- For classes: Document behavior, important attributes, and subclass behavior in the docstring following the `class` statement.
- For functions and methods: Document every argument, returned value, raised exception, and other behaviors in the docstring following the `def` statement.
- If you're using type annotations, omit the information that's already present in type annotations from docstrings since it would be redundant to have it in both places.

Item 85: Use Packages to Organize Modules and Provide Stable APIs

As the size of a program's codebase grows, it's natural for you to reorganize its structure. You'll split larger functions into smaller functions. You'll refactor data structures into helper classes (see Item 37: "Compose Classes Instead of Nesting Many Levels of Built-in Types" for an example). You'll separate functionality into various modules that depend on each other.

At some point, you'll find yourself with so many modules that you need another layer in your program to make it understandable. For

this purpose, Python provides *packages*. Packages are modules that contain other modules.

In most cases, packages are defined by putting an empty file named `__init__.py` into a directory. Once `__init__.py` is present, any other Python files in that directory will be available for import, using a path relative to the directory. For example, imagine that I have the following directory structure in my program:

```
main.py
mypackage/__init__.py
mypackage/models.py
mypackage/utils.py
```

To import the `utils` module, I use the absolute module name that includes the package directory's name:

```
# main.py
from mypackage import utils
```

This pattern continues when I have package directories present within other packages (like `mypackage.foo.bar`).

The functionality provided by packages has two primary purposes in Python programs.

Namespaces

The first use of packages is to help divide your modules into separate namespaces. They enable you to have many modules with the same filename but different absolute paths that are unique. For example, here's a program that imports attributes from two modules with the same filename, `utils.py`:

```
# main.py
from analysis.utils import log_base2_bucket
from frontend.utils import stringify

bucket = stringify(log_base2_bucket(33))
```

This approach breaks when the functions, classes, or submodules defined in packages have the same names. For example, say that I want to use the `inspect` function from both the `analysis.utils` and the `frontend.utils` modules. Importing the attributes directly won't work because the second `import` statement will overwrite the value of `inspect` in the current scope:

```
# main2.py
from analysis.utils import inspect
from frontend.utils import inspect  # Overwrites!
```

The solution is to use the `as` clause of the `import` statement to rename whatever I've imported for the current scope:

```
# main3.py
from analysis.utils import inspect as analysis_inspect
from frontend.utils import inspect as frontend_inspect

value = 33
if analysis_inspect(value) == frontend_inspect(value):
    print('Inspection equal!')
```

The `as` clause can be used to rename anything retrieved with the `import` statement, including entire modules. This facilitates accessing namespaced code and makes its identity clear when you use it.

Another approach for avoiding imported name conflicts is to always access names by their highest unique module name. For the example above, this means I'd use basic `import` statements instead of `import from`:

```
# main4.py
import analysis.utils
import frontend.utils

value = 33
if (analysis.utils.inspect(value) ==
    frontend.utils.inspect(value)):
    print('Inspection equal!')
```

This approach allows you to avoid the `as` clause altogether. It also makes it abundantly clear to new readers of the code where each of the similarly named functions is defined.

Stable APIs

The second use of packages in Python is to provide strict, stable APIs for external consumers.

When you're writing an API for wider consumption, such as an open source package (see Item 82: "Know Where to Find Community-Built Modules" for examples), you'll want to provide stable functionality that doesn't change between releases. To ensure that happens, it's important to hide your internal code organization from external users. This way, you can refactor and improve your package's internal modules without breaking existing users.

Python can limit the surface area exposed to API consumers by using the `__all__` special attribute of a module or package. The value of `__all__` is a `list` of every name to export from the module as part of its public API. When consuming code executes `from foo import *`,

only the attributes in `foo.__all__` will be imported from `foo`. If `__all__` isn't present in `foo`, then only public attributes—those without a leading underscore—are imported (see Item 42: "Prefer Public Attributes Over Private Ones" for details about that convention).

For example, say that I want to provide a package for calculating collisions between moving projectiles. Here, I define the `models` module of `mypackage` to contain the representation of projectiles:

```
# models.py
__all__ = ['Projectile']

class Projectile:
    def __init__(self, mass, velocity):
        self.mass = mass
        self.velocity = velocity
```

I also define a `utils` module in `mypackage` to perform operations on the `Projectile` instances, such as simulating collisions between them:

```
# utils.py
from . models import Projectile

__all__ = ['simulate_collision']

def _dot_product(a, b):
    ...

def simulate_collision(a, b):
    ...
```

Now, I'd like to provide all of the public parts of this API as a set of attributes that are available on the `mypackage` module. This will allow downstream consumers to always import directly from `mypackage` instead of importing from `mypackage.models` or `mypackage.utils`. This ensures that the API consumer's code will continue to work even if the internal organization of `mypackage` changes (e.g., `models.py` is deleted).

To do this with Python packages, you need to modify the `__init__.py` file in the `mypackage` directory. This file is what actually becomes the contents of the `mypackage` module when it's imported. Thus, you can specify an explicit API for `mypackage` by limiting what you import into `__init__.py`. Since all of my internal modules already specify `__all__`, I can expose the public interface of `mypackage` by simply importing everything from the internal modules and updating `__all__` accordingly:

```
# __init__.py
__all__ = []
from . models import *
```

```
__all__ += models.__all__
from . utils import *
__all__ += utils.__all__
```

Here's a consumer of the API that directly imports from `mypackage` instead of accessing the inner modules:

```
# api_consumer.py
from mypackage import *

a = Projectile(1.5, 3)
b = Projectile(4, 1.7)
after_a, after_b = simulate_collision(a, b)
```

Notably, internal-only functions like `mypackage.utils._dot_product` will not be available to the API consumer on `mypackage` because they weren't present in `__all__`. Being omitted from `__all__` also means that they weren't imported by the `from mypackage import *` statement. The internal-only names are effectively hidden.

This whole approach works great when it's important to provide an explicit, stable API. However, if you're building an API for use between your own modules, the functionality of `__all__` is probably unnecessary and should be avoided. The namespacing provided by packages is usually enough for a team of programmers to collaborate on large amounts of code they control while maintaining reasonable interface boundaries.

Beware of `import *`

Import statements like `from x import y` are clear because the source of `y` is explicitly the `x` package or module. Wildcard imports like `from foo import *` can also be useful, especially in interactive Python sessions. However, wildcards make code more difficult to understand:

- `from foo import *` hides the source of names from new readers of the code. If a module has multiple `import *` statements, you'll need to check all of the referenced modules to figure out where a name was defined.
- Names from `import *` statements will overwrite any conflicting names within the containing module. This can lead to strange bugs caused by accidental interactions between your code and overlapping names from multiple `import *` statements.

The safest approach is to avoid `import *` in your code and explicitly import names with the `from x import y` style.

Things to Remember

- Packages in Python are modules that contain other modules. Packages allow you to organize your code into separate, non-conflicting namespaces with unique absolute module names.
- Simple packages are defined by adding an `__init__.py` file to a directory that contains other source files. These files become the child modules of the directory's package. Package directories may also contain other packages.
- You can provide an explicit API for a module by listing its publicly visible names in its `__all__` special attribute.
- You can hide a package's internal implementation by only importing public names in the package's `__init__.py` file or by naming internal-only members with a leading underscore.
- When collaborating within a single team or on a single codebase, using `__all__` for explicit APIs is probably unnecessary.

Item 86: Consider Module-Scoped Code to Configure Deployment Environments

A deployment environment is a configuration in which a program runs. Every program has at least one deployment environment: the *production environment*. The goal of writing a program in the first place is to put it to work in the production environment and achieve some kind of outcome.

Writing or modifying a program requires being able to run it on the computer you use for developing. The configuration of your *development environment* may be very different from that of your production environment. For example, you may be using a tiny single-board computer to develop a program that's meant to run on enormous supercomputers.

Tools like `venv` (see Item 83: "Use Virtual Environments for Isolated and Reproducible Dependencies") make it easy to ensure that all environments have the same Python packages installed. The trouble is that production environments often require many external assumptions that are hard to reproduce in development environments.

For example, say that I want to run a program in a web server container and give it access to a database. Every time I want to modify my program's code, I need to run a server container, the database schema must be set up properly, and my program needs the password for access. This is a very high cost if all I'm trying to do is verify that a one-line change to my program works correctly.

The best way to work around such issues is to override parts of a program at startup time to provide different functionality depending on the deployment environment. For example, I could have two different `__main__` files—one for production and one for development:

```
# dev_main.py
TESTING = True

import db_connection

db = db_connection.Database()
```

```
# prod_main.py
TESTING = False

import db_connection

db = db_connection.Database()
```

The only difference between the two files is the value of the `TESTING` constant. Other modules in my program can then import the `__main__` module and use the value of `TESTING` to decide how they define their own attributes:

```
# db_connection.py
import __main__

class TestingDatabase:
    ...

class RealDatabase:
    ...

if __main__.TESTING:
    Database = TestingDatabase
else:
    Database = RealDatabase
```

The key behavior to notice here is that code running in module scope—not inside a function or method—is just normal Python code. You can use an `if` statement at the module level to decide how the module will define names. This makes it easy to tailor modules to your various deployment environments. You can avoid having to reproduce costly assumptions like database configurations when they aren't needed. You can inject local or fake implementations that ease interactive development, or you can use mocks for writing tests (see Item 78: "Use Mocks to Test Code with Complex Dependencies").

> Note
> When your deployment environment configuration gets really complicated, you should consider moving it out of Python constants (like TESTING) and into dedicated configuration files. Tools like the `configparser` built-in module let you maintain production configurations separately from code, a distinction that's crucial for collaborating with an operations team.

This approach can be used for more than working around external assumptions. For example, if I know that my program must work differently depending on its host platform, I can inspect the `sys` module before defining top-level constructs in a module:

```
# db_connection.py
import sys

class Win32Database:
    ...

class PosixDatabase:
    ...

if sys.platform.startswith('win32'):
    Database = Win32Database
else:
    Database = PosixDatabase
```

Similarly, I could use environment variables from `os.environ` to guide my module definitions.

Things to Remember

- Programs often need to run in multiple deployment environments that each have unique assumptions and configurations.
- You can tailor a module's contents to different deployment environments by using normal Python statements in module scope.
- Module contents can be the product of any external condition, including host introspection through the `sys` and `os` modules.

Item 87: Define a Root `Exception` to Insulate Callers from APIs

When you're defining a module's API, the exceptions you raise are just as much a part of your interface as the functions and classes you define (see Item 20: "Prefer Raising Exceptions to Returning `None`" for an example).

Python has a built-in hierarchy of exceptions for the language and standard library. There's a draw to using the built-in exception types for reporting errors instead of defining your own new types. For example, I could raise a `ValueError` exception whenever an invalid parameter is passed to a function in one of my modules:

```
# my_module.py
def determine_weight(volume, density):
    if density <= 0:
        raise ValueError('Density must be positive')
    ...
```

In some cases, using `ValueError` makes sense, but for APIs, it's much more powerful to define a new hierarchy of exceptions. I can do this by providing a root `Exception` in my module and having all other exceptions raised by that module inherit from the root exception:

```
# my_module.py
class Error(Exception):
    """Base-class for all exceptions raised by this module."""

class InvalidDensityError(Error):
    """There was a problem with a provided density value."""

class InvalidVolumeError(Error):
    """There was a problem with the provided weight value."""

def determine_weight(volume, density):
    if density < 0:
        raise InvalidDensityError('Density must be positive')
    if volume < 0:
        raise InvalidVolumeError('Volume must be positive')
    if volume == 0:
        density / volume
```

Having a root exception in a module makes it easy for consumers of an API to catch all of the exceptions that were raised deliberately. For example, here a consumer of my API makes a function call with a `try/except` statement that catches my root exception:

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.Error:
    logging.exception('Unexpected error')

>>>
Unexpected error
```

```
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(1, -1)
  File ".../my_module.py", line 10, in determine_weight
    raise InvalidDensityError('Density must be positive')
InvalidDensityError: Density must be positive
```

Here, the `logging.exception` function prints the full stack trace of the caught exception so it's easier to debug in this situation. The `try/except` also prevents my API's exceptions from propagating too far upward and breaking the calling program. It insulates the calling code from my API. This insulation has three helpful effects.

First, root exceptions let callers understand when there's a problem with their usage of an API. If callers are using my API properly, they should catch the various exceptions that I deliberately raise. If they don't handle such an exception, it will propagate all the way up to the insulating `except` block that catches my module's root exception. That block can bring the exception to the attention of the API consumer, providing an opportunity for them to add proper handling of the missed exception type:

```
try:
    weight = my_module.determine_weight(-1, 1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')
```

```
>>>
Bug in the calling code
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(-1, 1)
  File ".../my_module.py", line 12, in determine_weight
    raise InvalidVolumeError('Volume must be positive')
InvalidVolumeError: Volume must be positive
```

The second advantage of using root exceptions is that they can help find bugs in an API module's code. If my code only deliberately raises exceptions that I define within my module's hierarchy, then all other types of exceptions raised by my module must be the ones that I didn't intend to raise. These are bugs in my API's code.

Using the `try/except` statement above will not insulate API consumers from bugs in my API module's code. To do that, the caller needs to add another `except` block that catches Python's base `Exception` class.

This allows the API consumer to detect when there's a bug in the API module's implementation that needs to be fixed. The output for this example includes both the `logging.exception` message and the default interpreter output for the exception since it was re-raised:

```
try:
    weight = my_module.determine_weight(0, 1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')
except Exception:
    logging.exception('Bug in the API code!')
    raise  # Re-raise exception to the caller
```

```
>>>
Bug in the API code!
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(0, 1)
  File ".../my_module.py", line 14, in determine_weight
    density / volume
ZeroDivisionError: division by zero
Traceback ...
ZeroDivisionError: division by zero
```

The third impact of using root exceptions is future-proofing an API. Over time, I might want to expand my API to provide more specific exceptions in certain situations. For example, I could add an `Exception` subclass that indicates the error condition of supplying negative densities:

```
# my_module.py
...

class NegativeDensityError(InvalidDensityError):
    """A provided density value was negative."""

...

def determine_weight(volume, density):
    if density < 0:
        raise NegativeDensityError('Density must be positive')
    ...
```

The calling code will continue to work exactly as before because it already catches `InvalidDensityError` exceptions (the parent class of `NegativeDensityError`). In the future, the caller could decide to special-case the new type of exception and change the handling behavior accordingly:

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.NegativeDensityError:
    raise ValueError('Must supply non-negative density')
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')
except Exception:
    logging.exception('Bug in the API code!')
    raise
```

```
>>>
Traceback ...
NegativeDensityError: Density must be positive

The above exception was the direct cause of the following
➥exception:

Traceback ...
ValueError: Must supply non-negative density
```

I can take API future-proofing further by providing a broader set of exceptions directly below the root exception. For example, imagine that I have one set of errors related to calculating weights, another related to calculating volume, and a third related to calculating density:

```
# my_module.py
class Error(Exception):
    """Base-class for all exceptions raised by this module."""

class WeightError(Error):
    """Base-class for weight calculation errors."""

class VolumeError(Error):
    """Base-class for volume calculation errors."""

class DensityError(Error):
    """Base-class for density calculation errors."""
...
```

Specific exceptions would inherit from these general exceptions. Each intermediate exception acts as its own kind of root exception. This makes it easier to insulate layers of calling code from API code based on broad functionality. This is much better than having all callers catch a long list of very specific `Exception` subclasses.

Things to Remember

- Defining root exceptions for modules allows API consumers to insulate themselves from an API.
- Catching root exceptions can help you find bugs in code that consumes an API.
- Catching the Python `Exception` base class can help you find bugs in API implementations.
- Intermediate root exceptions let you add more specific types of exceptions in the future without breaking your API consumers.

Item 88: Know How to Break Circular Dependencies

Inevitably, while you're collaborating with others, you'll find a mutual interdependence between modules. It can even happen while you work by yourself on the various parts of a single program.

For example, say that I want my GUI application to show a dialog box for choosing where to save a document. The data displayed by the dialog could be specified through arguments to my event handlers. But the dialog also needs to read global state, such as user preferences, to know how to render properly.

Here, I define a dialog that retrieves the default document save location from global preferences:

```
# dialog.py
import app

class Dialog:
    def __init__(self, save_dir):
        self.save_dir = save_dir
    ...

save_dialog = Dialog(app.prefs.get('save_dir'))

def show():
    ...
```