```
  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():
    ...
```

The problem is that the app module that contains the prefs object also imports the dialog class in order to show the same dialog on program start:

```
# app.py
import dialog

class Prefs:
    ...
    def get(self, name):
        ...

prefs = Prefs()
dialog.show()
```

It's a circular dependency. If I try to import the app module from my main program like this:

```
# main.py
import app
```

I get an exception:

```
>>>
$ python3 main.py
Traceback (most recent call last):
  File ".../main.py", line 17, in <module>
    import app
  File ".../app.py", line 17, in <module>
    import dialog
  File ".../dialog.py", line 23, in <module>
    save_dialog = Dialog(app.prefs.get('save_dir'))
AttributeError: partially initialized module 'app' has no
➥attribute 'prefs' (most likely due to a circular import)
```

To understand what's happening here, you need to know how Python's import machinery works in general (see the importlib built-in package for the full details). When a module is imported, here's what Python actually does, in depth-first order:

1. Searches for a module in locations from sys.path
2. Loads the code from the module and ensures that it compiles
3. Creates a corresponding empty module object
4. Inserts the module into sys.modules
5. Runs the code in the module object to define its contents

The problem with a circular dependency is that the attributes of a module aren't defined until the code for those attributes has executed (after step 5). But the module can be loaded with the `import` statement immediately after it's inserted into `sys.modules` (after step 4).

In the example above, the `app` module imports `dialog` before defining anything. Then, the `dialog` module imports `app`. Since `app` still hasn't finished running—it's currently importing `dialog`—the `app` module is empty (from step 4). The `AttributeError` is raised (during step 5 for `dialog`) because the code that defines `prefs` hasn't run yet (step 5 for `app` isn't complete).

The best solution to this problem is to refactor the code so that the `prefs` data structure is at the bottom of the dependency tree. Then, both `app` and `dialog` can import the same utility module and avoid any circular dependencies. But such a clear division isn't always possible or could require too much refactoring to be worth the effort.

There are three other ways to break circular dependencies.

Reordering Imports

The first approach is to change the order of imports. For example, if I import the `dialog` module toward the bottom of the `app` module, after the `app` module's other contents have run, the `AttributeError` goes away:

```
# app.py
class Prefs:
    ...

prefs = Prefs()

import dialog  # Moved
dialog.show()
```

This works because, when the `dialog` module is loaded late, its recursive import of `app` finds that `app.prefs` has already been defined (step 5 is mostly done for `app`).

Although this avoids the `AttributeError`, it goes against the PEP 8 style guide (see Item 2: "Follow the PEP 8 Style Guide"). The style guide suggests that you always put imports at the top of your Python files. This makes your module's dependencies clear to new readers of the code. It also ensures that any module you depend on is in scope and available to all the code in your module.

Having imports later in a file can be brittle and can cause small changes in the ordering of your code to break the module entirely. I suggest not using import reordering to solve your circular dependency issues.

Import, Configure, Run

A second solution to the circular imports problem is to have modules minimize side effects at import time. I can have my modules only define functions, classes, and constants. I avoid actually running any functions at import time. Then, I have each module provide a `configure` function that I call once all other modules have finished importing. The purpose of `configure` is to prepare each module's state by accessing the attributes of other modules. I run `configure` after all modules have been imported (step 5 is complete), so all attributes must be defined.

Here, I redefine the `dialog` module to only access the `prefs` object when `configure` is called:

```
# dialog.py
import app

class Dialog:
    ...

save_dialog = Dialog()

def show():
    ...

def configure():
    save_dialog.save_dir = app.prefs.get('save_dir')
```

I also redefine the `app` module to not run activities on import:

```
# app.py
import dialog

class Prefs:
    ...

prefs = Prefs()

def configure():
    ...
```

Finally, the `main` module has three distinct phases of execution—import everything, `configure` everything, and run the first activity:

```
# main.py
import app
import dialog

app.configure()
dialog.configure()

dialog.show()
```

This works well in many situations and enables patterns like *dependency injection*. But sometimes it can be difficult to structure your code so that an explicit `configure` step is possible. Having two distinct phases within a module can also make your code harder to read because it separates the definition of objects from their configuration.

Dynamic Import

The third—and often simplest—solution to the circular imports problem is to use an `import` statement within a function or method. This is called a *dynamic import* because the module import happens while the program is running, not while the program is first starting up and initializing its modules.

Here, I redefine the `dialog` module to use a dynamic import. The `dialog.show` function imports the `app` module at runtime instead of the `dialog` module importing `app` at initialization time:

```
# dialog.py
class Dialog:
    ...

save_dialog = Dialog()

def show():
    import app  # Dynamic import
    save_dialog.save_dir = app.prefs.get('save_dir')
    ...
```

The `app` module can now be the same as it was in the original example. It imports `dialog` at the top and calls `dialog.show` at the bottom:

```
# app.py
import dialog
```

```
class Prefs:
    ...

prefs = Prefs()
dialog.show()
```

This approach has a similar effect to the import, configure, and run steps from before. The difference is that it requires no structural changes to the way the modules are defined and imported. I'm simply delaying the circular import until the moment I must access the other module. At that point, I can be pretty sure that all other modules have already been initialized (step 5 is complete for everything).

In general, it's good to avoid dynamic imports like this. The cost of the `import` statement is not negligible and can be especially bad in tight loops. By delaying execution, dynamic imports also set you up for surprising failures at runtime, such as `SyntaxError` exceptions long after your program has started running (see Item 76: "Verify Related Behaviors in `TestCase` Subclasses" for how to avoid that). However, these downsides are often better than the alternative of restructuring your entire program.

Things to Remember

- Circular dependencies happen when two modules must call into each other at import time. They can cause your program to crash at startup.
- The best way to break a circular dependency is by refactoring mutual dependencies into a separate module at the bottom of the dependency tree.
- Dynamic imports are the simplest solution for breaking a circular dependency between modules while minimizing refactoring and complexity.

Item 89: Consider `warnings` to Refactor and Migrate Usage

It's natural for APIs to change in order to satisfy new requirements that meet formerly unanticipated needs. When an API is small and has few upstream or downstream dependencies, making such changes is straightforward. One programmer can often update a small API and all of its callers in a single commit.

However, as a codebase grows, the number of callers of an API can be so large or fragmented across source repositories that it's infeasible or impractical to make API changes in lockstep with updating callers to match. Instead, you need a way to notify and encourage the people that you collaborate with to refactor their code and migrate their API usage to the latest forms.

For example, say that I want to provide a module for calculating how far a car will travel at a given average speed and duration. Here, I define such a function and assume that speed is in miles per hour and duration is in hours:

```
def print_distance(speed, duration):
    distance = speed * duration
    print(f'{distance} miles')

print_distance(5, 2.5)
```

```
>>>
12.5 miles
```

Imagine that this works so well that I quickly gather a large number of dependencies on this function. Other programmers that I collaborate with need to calculate and print distances like this all across our shared codebase.

Despite its success, this implementation is error prone because the units for the arguments are implicit. For example, if I wanted to see how far a bullet travels in 3 seconds at 1000 meters per second, I would get the wrong result:

```
print_distance(1000, 3)
```

```
>>>
3000 miles
```

I can address this problem by expanding the API of `print_distance` to include optional keyword arguments (see Item 23: "Provide Optional Behavior with Keyword Arguments" and Item 25: "Enforce Clarity with Keyword-Only and Positional-Only Arguments") for the units of `speed`, `duration`, and the computed distance to print out:

```
CONVERSIONS = {
    'mph': 1.60934 / 3600 * 1000,    # m/s
    'hours': 3600,                   # seconds
    'miles': 1.60934 * 1000,         # m
    'meters': 1,                     # m
    'm/s': 1,                        # m
    'seconds': 1,                    # s
}
```

```
def convert(value, units):
    rate = CONVERSIONS[units]
    return rate * value

def localize(value, units):
    rate = CONVERSIONS[units]
    return value / rate

def print_distance(speed, duration, *,
                   speed_units='mph',
                   time_units='hours',
                   distance_units='miles'):
    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')
```

Now, I can modify the speeding bullet call to produce an accurate result with a unit conversion to miles:

```
print_distance(1000, 3,
               speed_units='meters',
               time_units='seconds')
```

```
>>>
1.8641182099494205 miles
```

It seems like requiring units to be specified for this function is a much better way to go. Making them explicit reduces the likelihood of errors and is easier for new readers of the code to understand. But how can I migrate all callers of the API over to always specifying units? How do I minimize breakage of any code that's dependent on `print_distance` while also encouraging callers to adopt the new units arguments as soon as possible?

For this purpose, Python provides the built-in `warnings` module. Using warnings is a programmatic way to inform other programmers that their code needs to be modified due to a change to an underlying library that they depend on. While exceptions are primarily for automated error handling by machines (see Item 87: "Define a Root `Exception` to Insulate Callers from APIs"), warnings are all about communication between humans about what to expect in their collaboration with each other.

I can modify print_distance to issue warnings when the optional keyword arguments for specifying units are not supplied. This way, the arguments can continue being optional temporarily (see Item 24: "Use None and Docstrings to Specify Dynamic Default Arguments" for background), while providing an explicit notice to people running dependent programs that they should expect breakage in the future if they fail to take action:

```
import warnings

def print_distance(speed, duration, *,
                   speed_units=None,
                   time_units=None,
                   distance_units=None):
    if speed_units is None:
        warnings.warn(
            'speed_units required', DeprecationWarning)
        speed_units = 'mph'

    if time_units is None:
        warnings.warn(
            'time_units required', DeprecationWarning)
        time_units = 'hours'

    if distance_units is None:
        warnings.warn(
            'distance_units required', DeprecationWarning)
        distance_units = 'miles'

    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')
```

I can verify that this code issues a warning by calling the function with the same arguments as before and capturing the sys.stderr output from the warnings module:

```
import contextlib
import io

fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
```

```
    print_distance(1000, 3,
                   speed_units='meters',
                   time_units='seconds')

print(fake_stderr.getvalue())
```

```
>>>
1.8641182099494205 miles
.../example.py:97: DeprecationWarning: distance_units required
  warnings.warn(
```

Adding warnings to this function required quite a lot of repetitive boilerplate that's hard to read and maintain. Also, the warning message indicates the line where `warning.warn` was called, but what I really want to point out is where the call to `print_distance` was made *without* soon-to-be-required keyword arguments.

Luckily, the `warnings.warn` function supports the `stacklevel` parameter, which makes it possible to report the correct place in the stack as the cause of the warning. `stacklevel` also makes it easy to write functions that can issue warnings on behalf of other code, reducing boilerplate. Here, I define a helper function that warns if an optional argument wasn't supplied and then provides a default value for it:

```
def require(name, value, default):
    if value is not None:
        return value
    warnings.warn(
        f'{name} will be required soon, update your code',
        DeprecationWarning,
        stacklevel=3)
    return default

def print_distance(speed, duration, *,
                   speed_units=None,
                   time_units=None,
                   distance_units=None):
    speed_units = require('speed_units', speed_units, 'mph')
    time_units = require('time_units', time_units, 'hours')
    distance_units = require(
        'distance_units', distance_units, 'miles')

    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')
```

I can verify that this propagates the proper offending line by inspecting the captured output:

```
import contextlib
import io

fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    print_distance(1000, 3,
                   speed_units='meters',
                   time_units='seconds')

print(fake_stderr.getvalue())
```

```
>>>
1.8641182099494205 miles
.../example.py:174: DeprecationWarning: distance_units will be
➥required soon, update your code
  print_distance(1000, 3,
```

The warnings module also lets me configure what should happen when a warning is encountered. One option is to make all warnings become errors, which raises the warning as an exception instead of printing it out to sys.stderr:

```
warnings.simplefilter('error')
try:
    warnings.warn('This usage is deprecated',
                  DeprecationWarning)
except DeprecationWarning:
    pass  # Expected
```

This exception-raising behavior is especially useful for automated tests in order to detect changes in upstream dependencies and fail tests accordingly. Using such test failures is a great way to make it clear to the people you collaborate with that they will need to update their code. You can use the `-W error` command-line argument to the Python interpreter or the `PYTHONWARNINGS` environment variable to apply this policy:

```
$ python -W error example_test.py
Traceback (most recent call last):
  File ".../example_test.py", line 6, in <module>
    warnings.warn('This might raise an exception!')
UserWarning: This might raise an exception!
```

Once the people responsible for code that depends on a deprecated API are aware that they'll need to do a migration, they can tell the warnings module to ignore the error by using the `simplefilter` and `filterwarnings` functions (see https://docs.python.org/3/library/warnings for all the details):

```
warnings.simplefilter('ignore')
warnings.warn('This will not be printed to stderr')
```

After a program is deployed into production, it doesn't make sense for warnings to cause errors because they might crash the program at a critical time. Instead, a better approach is to replicate warnings into the `logging` built-in module. Here, I accomplish this by calling the `logging.captureWarnings` function and configuring the corresponding `'py.warnings'` logger:

```
import logging

fake_stderr = io.StringIO()
handler = logging.StreamHandler(fake_stderr)
formatter = logging.Formatter(
    '%(asctime)-15s WARNING] %(message)s')
handler.setFormatter(formatter)

logging.captureWarnings(True)
logger = logging.getLogger('py.warnings')
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

warnings.resetwarnings()
warnings.simplefilter('default')
warnings.warn('This will go to the logs output')

print(fake_stderr.getvalue())
```

```
>>>
2019-06-11 19:48:19,132 WARNING] .../example.py:227:
➥UserWarning: This will go to the logs output
  warnings.warn('This will go to the logs output')
```

Using logging to capture warnings ensures that any error reporting systems that my program already has in place will also receive notice of important warnings in production. This can be especially useful if my tests don't cover every edge case that I might see when the program is undergoing real usage.

API library maintainers should also write unit tests to verify that warnings are generated under the correct circumstances with clear and actionable messages (see Item 76: "Verify Related Behaviors in TestCase Subclasses"). Here, I use the warnings.catch_warnings function as a context manager (see Item 66: "Consider contextlib and with Statements for Reusable try/finally Behavior" for background) to wrap a call to the require function that I defined above:

```
with warnings.catch_warnings(record=True) as found_warnings:
    found = require('my_arg', None, 'fake units')
    expected = 'fake units'
    assert found == expected
```

Once I've collected the warning messages, I can verify that their number, detail messages, and categories match my expectations:

```
assert len(found_warnings) == 1
single_warning = found_warnings[0]
assert str(single_warning.message) == (
    'my_arg will be required soon, update your code')
assert single_warning.category == DeprecationWarning
```

Things to Remember

- The warnings module can be used to notify callers of your API about deprecated usage. Warning messages encourage such callers to fix their code before later changes break their programs.
- Raise warnings as errors by using the -W error command-line argument to the Python interpreter. This is especially useful in automated tests to catch potential regressions of dependencies.
- In production, you can replicate warnings into the logging module to ensure that your existing error reporting systems will capture warnings at runtime.
- It's useful to write tests for the warnings that your code generates to make sure that they'll be triggered at the right time in any of your downstream dependencies.

Item 90: Consider Static Analysis via typing to Obviate Bugs

Providing documentation is a great way to help users of an API understand how to use it properly (see Item 84: "Write Docstrings for Every Function, Class, and Module"), but often it's not enough, and incorrect usage still causes bugs. Ideally, there would be a programmatic

mechanism to verify that callers are using your APIs the right way, and that you are using your downstream dependencies correctly. Many programming languages address part of this need with compile-time type checking, which can identify and eliminate some categories of bugs.

Historically Python has focused on dynamic features and has not provided compile-time type safety of any kind. However, more recently Python has introduced special syntax and the built-in `typing` module, which allow you to annotate variables, class fields, functions, and methods with type information. These *type hints* allow for *gradual typing*, where a codebase can be incrementally updated to specify types as desired.

The benefit of adding type information to a Python program is that you can run *static analysis* tools to ingest a program's source code and identify where bugs are most likely to occur. The `typing` built-in module doesn't actually implement any of the type checking functionality itself. It merely provides a common library for defining types, including generics, that can be applied to Python code and consumed by separate tools.

Much as there are multiple distinct implementations of the Python interpreter (e.g., CPython, PyPy), there are multiple implementations of static analysis tools for Python that use `typing`. As of the time of this writing, the most popular tools are `mypy` (https://github.com/python/mypy), `pytype` (https://github.com/google/pytype), `pyright` (https://github.com/microsoft/pyright), and `pyre` (https://pyre-check.org). For the `typing` examples in this book, I've used `mypy` with the `--strict` flag, which enables all of the various warnings supported by the tool. Here's an example of what running the command line looks like:

```
$ python3 -m mypy --strict example.py
```

These tools can be used to detect a large number of common errors before a program is ever run, which can provide an added layer of safety in addition to having good unit tests (see Item 76: "Verify Related Behaviors in `TestCase` Subclasses"). For example, can you find the bug in this simple function that causes it to compile fine but throw an exception at runtime?

```
def subtract(a, b):
    return a - b

subtract(10, '5')
```

```
>>>
Traceback ...
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Parameter and variable type annotations are delineated with a colon (such as `name: type`). Return value types are specified with `-> type` following the argument list. Using such type annotations and `mypy`, I can easily spot the bug:

```
def subtract(a: int, b: int) -> int:  # Function annotation
    return a - b

subtract(10, '5')  # Oops: passed string value
```

```
$ python3 -m mypy --strict example.py
.../example.py:4: error: Argument 2 to "subtract" has
incompatible type "str"; expected "int"
```

Another common mistake, especially for programmers who have recently moved from Python 2 to Python 3, is mixing `bytes` and `str` instances together (see Item 3: "Know the Differences Between `bytes` and `str`"). Do you see the problem in this example that causes a run-time error?

```
def concat(a, b):
    return a + b

concat('first', b'second')
```

```
>>>
Traceback ...
TypeError: can only concatenate str (not "bytes") to str
```

Using type hints and `mypy`, this issue can be detected statically before the program runs:

```
def concat(a: str, b: str) -> str:
    return a + b

concat('first', b'second')  # Oops: passed bytes value
```

```
$ python3 -m mypy --strict example.py
.../example.py:4: error: Argument 2 to "concat" has
➥incompatible type "bytes"; expected "str"
```

Type annotations can also be applied to classes. For example, this class has two bugs in it that will raise exceptions when the program is run:

```
class Counter:
    def __init__(self):
        self.value = 0
```

```
    def add(self, offset):
        value += offset

    def get(self) -> int:
        self.value
```

The first one happens when I call the `add` method:

```
counter = Counter()
counter.add(5)
```

```
>>>
Traceback ...
UnboundLocalError: local variable 'value' referenced before
➥assignment
```

The second bug happens when I call `get`:

```
counter = Counter()
found = counter.get()
assert found == 0, found
```

```
>>>
Traceback ...
AssertionError: None
```

Both of these problems are easily found by `mypy`:

```
class Counter:
    def __init__(self) -> None:
        self.value: int = 0  # Field / variable annotation

    def add(self, offset: int) -> None:
        value += offset      # Oops: forgot "self."

    def get(self) -> int:
        self.value           # Oops: forgot "return"

counter = Counter()
counter.add(5)
counter.add(3)
assert counter.get() == 8
```

```
$ python3 -m mypy --strict example.py
.../example.py:6: error: Name 'value' is not defined
.../example.py:8: error: Missing return statement
```

One of the strengths of Python's dynamism is the ability to write generic functionality that operates on duck types (see Item 15: "Be Cautious When Relying on dict Insertion Ordering" and Item 43: "Inherit from collections.abc for Custom Container Types"). This allows one implementation to accept a wide range of types, saving a lot of duplicative effort and simplifying testing. Here, I've defined such a generic function for combining values from a list. Do you understand why the last assertion fails?

```
def combine(func, values):
    assert len(values) > 0

    result = values[0]
    for next_value in values[1:]:
        result = func(result, next_value)

    return result

def add(x, y):
    return x + y

inputs = [1, 2, 3, 4j]
result = combine(add, inputs)
assert result == 10, result  # Fails
```

```
>>>
Traceback ...
AssertionError: (6+4j)
```

I can use the typing module's support for generics to annotate this function and detect the problem statically:

```
from typing import Callable, List, TypeVar

Value = TypeVar('Value')
Func = Callable[[Value, Value], Value]

def combine(func: Func[Value], values: List[Value]) -> Value:
    assert len(values) > 0

    result = values[0]
    for next_value in values[1:]:
        result = func(result, next_value)

    return result
```

```
Real = TypeVar('Real', int, float)

def add(x: Real, y: Real) -> Real:
    return x + y

inputs = [1, 2, 3, 4j]  # Oops: included a complex number
result = combine(add, inputs)
assert result == 10
```

```
$ python3 -m mypy --strict example.py
.../example.py:21: error: Argument 1 to "combine" has
➥incompatible type "Callable[[Real, Real], Real]"; expected
➥"Callable[[complex, complex], complex]"
```

Another extremely common error is to encounter a `None` value when you thought you'd have a valid object (see Item 20: "Prefer Raising Exceptions to Returning `None`"). This problem can affect seemingly simple code. Do you see the issue here?

```
def get_or_default(value, default):
    if value is not None:
        return value
    return value

found = get_or_default(3, 5)
assert found == 3

found = get_or_default(None, 5)
assert found == 5, found  # Fails
```

```
>>>
Traceback ...
AssertionError: None
```

The `typing` module supports *option types*, which ensure that programs only interact with values after proper null checks have been performed. This allows `mypy` to infer that there's a bug in this code: The type used in the return statement must be `None`, and that doesn't match the `int` type required by the function signature:

```
from typing import Optional

def get_or_default(value: Optional[int],
                   default: int) -> int:
    if value is not None:
        return value
    return value  # Oops: should have returned "default"
```

```
$ python3 -m mypy --strict example.py
.../example.py:7: error: Incompatible return value type (got
➥"None", expected "int")
```

A wide variety of other options are available in the `typing` module. See https://docs.python.org/3.8/library/typing for all of the details. Notably, exceptions are not included. Unlike Java, which has checked exceptions that are enforced at the API boundary of every method, Python's type annotations are more similar to C#'s: Exceptions are not considered part of an interface's definition. Thus, if you want to verify that you're raising and catching exceptions properly, you need to write tests.

One common gotcha in using the `typing` module occurs when you need to deal with forward references (see Item 88: "Know How to Break Circular Dependencies" for a similar problem). For example, imagine that I have two classes and one holds a reference to the other:

```
class FirstClass:
    def __init__(self, value):
        self.value = value

class SecondClass:
    def __init__(self, value):
        self.value = value

second = SecondClass(5)
first = FirstClass(second)
```

If I apply type hints to this program and run `mypy` it will say that there are no issues:

```
class FirstClass:
    def __init__(self, value: SecondClass) -> None:
        self.value = value

class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value

second = SecondClass(5)
first = FirstClass(second)
```

```
$ python3 -m mypy --strict example.py
```

However, if you actually try to run this code, it will fail because SecondClass is referenced by the type annotation in the FirstClass.__init__ method's parameters before it's actually defined:

```
class FirstClass:
    def __init__(self, value: SecondClass) -> None:  # Breaks
        self.value = value

class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value

second = SecondClass(5)
first = FirstClass(second)
```

```
>>>
Traceback ...
NameError: name 'SecondClass' is not defined
```

One workaround supported by these static analysis tools is to use a string as the type annotation that contains the forward reference. The string value is later parsed and evaluated to extract the type information to check:

```
class FirstClass:
    def __init__(self, value: 'SecondClass') -> None:  # OK
        self.value = value

class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value

second = SecondClass(5)
first = FirstClass(second)
```

A better approach is to use `from __future__ import annotations`, which is available in Python 3.7 and will become the default in Python 4. This instructs the Python interpreter to completely ignore the values supplied in type annotations when the program is being run. This resolves the forward reference problem and provides a performance improvement at program start time:

```
from __future__ import annotations

class FirstClass:
    def __init__(self, value: SecondClass) -> None:  # OK
        self.value = value
```

```
class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value

second = SecondClass(5)
first = FirstClass(second)
```

Now that you've seen how to use type hints and their potential benefits, it's important to be thoughtful about when to use them. Here are some of the best practices to keep in mind:

- It's going to slow you down if you try to use type annotations from the start when writing a new piece of code. A general strategy is to write a first version without annotations, then write tests, and then add type information where it's most valuable.
- Type hints are most important at the boundaries of a codebase, such as an API you provide that many callers (and thus other people) depend on. Type hints complement integration tests (see Item 77: "Isolate Tests from Each Other with `setUp`, `tearDown`, `setUpModule`, and `tearDownModule`") and warnings (see Item 89: "Consider `warnings` to Refactor and Migrate Usage") to ensure that your API callers aren't surprised or broken by your changes.
- It can be useful to apply type hints to the most complex and error-prone parts of your codebase that aren't part of an API. However, it may not be worth striving for 100% coverage in your type annotations because you'll quickly encounter diminishing returns.
- If possible, you should include static analysis as part of your automated build and test system to ensure that every commit to your codebase is vetted for errors. In addition, the configuration used for type checking should be maintained in the repository to ensure that all of the people you collaborate with are using the same rules.
- As you add type information to your code, it's important to run the type checker as you go. Otherwise, you may nearly finish sprinkling type hints everywhere and then be hit by a huge wall of errors from the type checking tool, which can be disheartening and make you want to abandon type hints altogether.

Finally, it's important to acknowledge that in many situations, you may not need or want to use type annotations at all. For small programs, ad-hoc code, legacy codebases, and prototypes, type hints may require far more effort than they're worth.

Things to Remember

- Python has special syntax and the `typing` built-in module for annotating variables, fields, functions, and methods with type information.
- Static type checkers can leverage type information to help you avoid many common bugs that would otherwise happen at runtime.
- There are a variety of best practices for adopting types in your programs, using them in APIs, and making sure they don't get in the way of your productivity.

Index

D

Video Course Companion to *Effective Python*

SAVE 60%*
CODE: VIDEOBOB

livelessons
video instruction from technology experts

Effective Python LiveLessons provides visual demonstration of the items presented in the book version so you can see each item in action. Watch Brett Slatkin provide concise and specific guidance on what to do and what to avoid when writing programs using Python.

Watch and learn how to:

- Use expressions and statements more efficiently
- Make better use of comprehensions and generators
- Work with concurrency and parallelism
- Make better use of functions and classes
- Make your programs more robust

Save 60%*—Use coupon code VIDEOBOB

informit.com/slatkin/video

**Discount code VIDEOBOB confers a 60% discount off the list price of featured video when purchased on InformIT. Offer is subject to change.*

Pearson

informIT
the trusted technology learning source

Photo by izusek/gettyimages

Register Your Product at informit.com/register

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

**Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions (informit.com/promotions)
- Sign up for special offers and content newsletter (informit.com/newsletters)
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit informit.com/community

informIT®
the trusted technology learning source

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • Peachpit Press

Pearson