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.