Chapter 10 - Collaboration

May 8, 2021

0.1 Item 83: Use Virtual Enviorments for Isolated and Reproducible Dependencies

```
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:>


To leave type
(myproject) deactivate
```

- to test an import you can type
  - `python -c 'import flask`
- to see which python you are using (virtual or not)
  - `which python3`

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

- you can add documentation by providing a `docstring` immediately after the `def` statement of a function

[1]:
```
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

[2]:
```
print(repr(palindrome.__doc__))
```

'Return True if the given word is a palindrome'

- you can also run 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`
- support for docstrings and the `__doc__` attribute has three consequences:
  - the accessibility of documentation makes interative development easier. you can inspect functions/classes and modules to see their documentation by using the `help` built-in function
  - a standard way of defining documentation makes it easy to build tools tht convert text into more appealing formats. this has led to excellent documentation-generation tools such as `Sphinx`
  - Python's first-class, accessible and good-looking documentation encourages peopel to write more documentation

https://www.python.org/dev/peps/pep-0257/

0.2.1 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 qoutes
- the goal of the docstring is to introduce the module and its contents
- thge first line of the docstring should be a single sentence describing the modules purpose
- the paragraph that follows 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 high-light important classes and fucntions found in the module

[3]:
```
# 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
"""
```

[3]:
```
"Library for finding linguistic patterns in words.\n\nTesting how words relate
to each other can be tricky sometimes!\nThis module provides easy ways to
determine when words you've\nfound have special properties\n\nAvailable
functions:\n- palindrome: Determine if a word is a palindrome\n- check_anagram:
Determine if two words are anagrams\n"
```

0.2.2 Documenting Classes

- each class should have a class-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 provide guidance to subclasses on how to properly interact with protected attributes

[4]:
```
class Player:
    """Represents a player of the game

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

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

0.2.3 Documenting Functions

- each public function and method should have a docstring
- the first line is a single-sentence description of what the function does
- the paragraph that folows 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 interface should be explained

[5]:
```
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
    """
```

Special Cases in Writing Docstrings for functions: - 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 "return

None" - if a function's interface includes raising exceptions, its docstring should describe each exception that's raised and when it's raised - if you dont expect a function to raise an exception during normal operation, don't mention that fact - if a function accepts a variable number of arguments or 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 - if a function i s a generator, its docstring should describe what the generator yeilds when its iterated - if a function is an asynchronous coroutine, its docstring should explain when it will stop execution

0.2.4 Using Docstring and Type Annotations

- if you are using `type` annotations, then there are no reason to specify that in the `doc-string`

[9]:
```
from typing import Container, List

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
    """
```

0.2.5 Things to Remember

- write documentation for every module, class, method and function using docstrings
- 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/methods`: document every argument, returned value, raised exception, and other behaviors in the docstring following the `def` statement
- if your're using `type` annotations, omit the information that already present in the type annotations from docstrings since it would be redundant to have it in both places

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

- at some point in the refactoring, you'll find youself 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 the `__init__.py` is present, any other Python file in that directory will be available for import, using a path relative to the directory

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

- to import `utils` module, we use the absolute module name that includes the package directory's name:

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

- the pattern continues when we have package directories present within other packages
  - `mypackage.foo.bar`

0.3.1 Namespaces

- the first use of a package is to help divide your modules into seprate namespaces
- they enable you to have many modules with the same filename but different absolute paths that are unique
- 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 breakes when the functions, classes or submodules defined in the package have the same names
- say that we want to use the `inspect` function from both the `analysis.utils` and the `frontend.utils` modules
- importing the attribute 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!')
```

- another approach for avoiding imported name conflicts is to always access names by their higest unique module name
- using the example above, that would mean we would use the basic `import` statement 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!')
```

0.3.2 Stable APIs

- the second use of packages in Python is to provide strict, stable APIs for external consumers
- by hiding your internal code oganization form external users, 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 imprted from `foo`
- if `__all__` isn't present in foo, then only public attributes- those without a leading underscore- are imported

- say that we want to provide a package for calculating collisions between moving projectiles
- we define the `models` module of `mypackage` to contain the representation of projectiles

[13]:
```
# models.py
__all__ = ['Projectile']

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

- we also define a `utils` module in `mypackage` to perform operations on the `Projectile` instance, such as simulating collisions between them

[20]:
```
# utils.py
# from . models import Projectile

__all__ = ['simulate_collision']
```

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

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

- now we would 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 form `mypackage` instead of importing from `packgage.modules` 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 modift the `__init__.py` file in the `mypackage` directory
- this file is what actually becomes the contents of the `mypackage` module when its improted
- thus you can specify an explicit API for `mpypackage` by limiting what you imported into `__init__.py`
- since all of my internal modules already specify `__all__`, we can expose the public interface of `mypackage` by simply importing everything from the internal modules and updating `__all__` accordingly

[21]:
```
"""
# init__.py

__all__ = []
from .models import *

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

[21]:
```
'\n# init__.py\n\n__all__ = []\nfrom .models import *\n\n__all__ +=
models.__all__\nfrom .utils import *\n__all__ += utils.__all\n'
```

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

[22]:
```
# 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`

`improt *` statement
- the internal-only names are effectively hidden

- the approach is great when its important to provide an explicit stable API
- if you are howerver 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

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

- every program has at least one development environment: the `production environment`
- the goal of writing a program in the first place is to put it to work in the production environment
- sometimes you need a `development environment` to be able to test because the `production environment` may not be reproducable

[3]:
```
TESTING = True

class TestingDatabase:
    ...
class RealDatabase:
    ...

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

- you can use an `if` statement at the module level to decide how the module will define names
- for complex configurations use a dedicated `configuration` file and tools like `configparser` built-in module

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

- 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 types
- for `APIs` its much more powerful to define a root `Exception` in my module and having all other exceptions raised by that module inherit from the root exception

[4]:
```
class Error(Exception):
    """Base-class for all exceptions raised by this module"""

class InvalidDensityError(Error):
    """There was a problem with 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, a consumer of my API makes a function call with a `try/except` statement that catches my root exception

- 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

0.5.1 the `insulation` has three helpful effects:

- first, the root exception lets 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 are deliberately raised
- if they don't handle such exceptions, it will propagate all the way to the insulating `except` block that catches my module's root exception

- the second advantage of using root exceptions is that they can help find bugs in an API module's code
- if you have not raised the exception yourself, then its a bug
- using the `try/except` statement above will not insulate the API consumers from bugs in my API modules 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 need to be fixed
- the output for the example below includes botht he `logging.exception` message and the default interpreter output for the exception since it was re-raised

[5]:
```
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
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-cc7fcf1eec70> in <module>
      1 try:
----> 2     weight = my_module.determine_weight(0, 1)
      3 except my_module.InvalidDensityError:

NameError: name 'my_module' is not defined

During handling of the above exception, another exception occurred:

NameError                                 Traceback (most recent call last)
<ipython-input-5-cc7fcf1eec70> in <module>
      1 try:
      2     weight = my_module.determine_weight(0, 1)
----> 3 except my_module.InvalidDensityError:
      4     weight = 0
      5 except my_module.Error:

NameError: name 'my_module' is not defined
```

- the third impack of using root exception is future-proofing an API
- over time we might want to expand an API to provide more specific exceptions in certain situations
- for example, we could add `Exception` subclass that indicates the error condition of supplying negative densities
- the calling code will continue to work exactly as before it already catches `InvalidDensityError` exceptions
- in the future, the caller could decide to special-case the new type of exception and change the handling behavior accordingly

[6]:
```
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
```

```
---------------------------------------------------------------------------
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-6-082b42a7e20f> in <module>
      1 try:
----> 2     weight = my_module.determine_weight(1, -1)
      3 except my_module.NegativeDensityError:

NameError: name 'my_module' is not defined

During handling of the above exception, another exception occurred:

NameError                                 Traceback (most recent call last)
<ipython-input-6-082b42a7e20f> in <module>
      1 try:
      2     weight = my_module.determine_weight(1, -1)
----> 3 except my_module.NegativeDensityError:
      4     raise ValueError('Must supply non-negative density')
      5 except my_module.InvalidDensityError:

NameError: name 'my_module' is not defined
```

- we can make `API` future-proofing better by providing a broader set of exceptions directly below the root exception
- for example, we can make it so that one set of errors is related to calculating `weights`, another related to calculating `volume` and a third related to calculating density

[8]:
```
# 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 kind of root exception
- this makes it easier to insulate layers of calling code from `API` code based on broad functionality

0.6 Item 88: Know How to Break Circular Dependencies

- when a module is imported, here is what Python actually does, in a `DFS` manor:

Pythons Import Machinery: 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

obejct 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 arent defined untill the code for those attributes has executed
- but the module can be loaded with the `import` statement immediately after its inserted into the `sys.modules` (aftert step 4)
- 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

0.6.1 Reording Imports

- the first approach is to change the orfer of imports
- put the `module` causing the trouble towards the bottom

0.6.2 Import, Configure, Run

- a second solution is to have modules minimize side effects at import time
- we can have modules only define functions, classes, and constants
- we can avoid actually running any functions at import time
- then we can have each module provide a `configure` function that we can call once all the other modules have finished importing
- the purpose of `configure` is to prepare each module's state by accessing the attributes of other modules
- we run `configure` after all modules have been imported (step 5 complete), so all attributes must be defined
- the approach works in many situations and enables patterns like `dependency injections` but make code hard to read because it separates the definition of objects from their configuration

0.6.3 Dynamic Imports

- the third solution is to use `import` statement within a function or method
- this is called `dynamic import` because the improts happen while the program is running, not while the program is first starting up and initializing its modules

0.6.4 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
- best way to break a circular dependency is by refactoring multiple dependencies into a separate module at the bottom of the dependency tree

0.7 Item 89: Consider `Warnings` to Refactor and Migrate Usage

- for large `APIs` making changes will not be possible
- thus we 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 this purpose python provides the built-in `warnings` module

- using warnings is a programmatic way to inform other programers that thair code needs to be modified due to change to an underlying library that depends on
- `warnings` are all about communication between humans about what to expect in their collaboration with each other

[9]:
```
import warnings

def warn(people):
    if people is None:
        warnings.warn(
         'people is None', DeprecationWarning)

warn(None)
```

```
<ipython-input-9-a79ca2631358>:5: DeprecationWarning: people is None
  warnings.warn(
```

- `warnings.warm` 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 function that can issue warnings on behalf of other code, reducing boilerplate

[11]:
```
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 warn(people):
    if people is None:
        need_people = require('people', 4, 'humpy')

warn(None)
```

- the `warnings` module also lets us 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`

[13]:
```
warnings.simplefilter('error')
try:
    warnings.warn('This usage is deprecated',
                  DeprecationWarning)
except DeprecationWarning:
    pass # Expected
```

- this exception-raising behavior is especially useful for automates tests in order to detect

changes in upstram 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 thier code
- you can use the `-W error` command-line argument to the Python interpreter or the `PYTHONWARNINGS` environment variables to apply this policy

- once the users 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` function

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

- after a program is deployed into production, it does not make sense for warnings to cause errors because they might crash the program at a critical time
- a better approach is to replicate warnings into the `logging` built-in module
- we can accomplish this by calling `logging.captureWarnings` function and configure the corresponding `py.warrnings` logger

[ ]:
```
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())
```

- 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 test don't cover edge cases 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
- below we use the `warnings.catch_warnings` function as a context manager to wrap a call to the `require` function that I defined above

[ ]:
```
with warnings.catch_warnings(record=True) as found-warnings:
    found = require('my_args', None, 'fake units')
    expected = 'fake units'
```

```
    assert found == expected
```

- once we have collected the warning messages, I can verify that their number, detail messages and categories match my expec tations

[ ]:
```
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
```

0.8 Item 90: Consider Static Analysis via `typing` to Obviate Bugs

- 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 codebase can be incrementally updated to specific types as desired
- the benefit of adding type information to Python program is that you can run `static analysis` tools to ingest a programs 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
- the most popular onces are `mypy`, `pytype` and `pyre`
- `python3 -m mypy --strict example.py`
- parameter and variable type annotations are delineated with a colon `name: type`
- return value types are specified with `-> type` following the arugment list
- a common mistake for beginner programers is to mix `bytes` and `str` instances
- type annotations can also be applied to class methods

[ ]:
```
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"
```

- one of the strenghts of Python's dynamism is the ability to write generic functionality that operates on `duck` types
- this allows one implementation to accept a wide range of types, saving a lot of duplicative efforts and simplifying testing
- below we define such generic function for combining values from a `list`

[ ]:
```
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
```

- we can use `typing` modules 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], value: 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] # Opps: Included a complex number
assert result == 10
```

- another extremely common error is to encounter a `None` value when you though you have a valid object

[ ]:
```
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
```

- `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 does not match the `int` type required by 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"
```

- you can also use typing with exceptions
- unlike Java, which has checked exceptions that are enforced at the API boundary of every method, Python's type annotations are more simmilar to `C#`
- 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 refrences
- imagine we have two classes and one holds a refrence 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 we apply type hints to this program and run `mypy` it will say that there are no issues
  - `python3 -m mypy --strict example.py`

[ ]:
```
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)
```

- howerver, if we 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 its 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)
```

- one workaround supported by these static analysis tools is to use a string as the type annotation that contains the forward refrence
- 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
- this instructs the Python interpreter to completely ignore the values supplied in type annotations when the program is being run
- this resolves the forward refrence 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)
```

0.8.1 Type Annotations Best Pratices

- its going to slow you down if you try to use type annotations fom the start when writing a new piece of code. General strategy is to write a first version without annotations, then write tests, and then add type information where its most valuable
- type hints are most important at the boundaries of a codebase, such as an API you provide that many callers depend on
- Type hints complement integration tests and warning to ensure that your API caller 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 arent part of an API. Howerver it may not be worth striving for `100%` coverage in your type annotations because you'll quickly enounter diminishing returns
- if possible you should include static analysis as part of your anomated build and test systems 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, its important to run the type checker as you go. Otherwise, you may nearly finish sprinkling type hints everwhere and then be hit by a huge wall of errors from the type checking tool