Python

Matt Waismann

Sep 20th, 2021

1 Structuring Python Projects

```
spam
├── ham
└── eggs
    ├── more spam
    └── dead parrots
```

1.1 Some Context

In the simplest world, we would write Python programs directly in the Python interpreter and run the program there. However, if you quit from the interpreter and enter it again, the definitions you have made (functions, classes, variables etc.) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This file prepared in the text editor is known as a **script**. A script is generally a directly executable piece of code, run by itself.

As the world becomes less simple, programs get larger, and holding all of your program on one file becomes infeasible, you may want to split that script into many files. Note that these individual files may not necessarily be scripts themselves. Instead, they might simply contain defintions and functionality intended to be used by some other file. In this event, instead of redudantly copying and pasting that functionalityy into the file that wants it, Python has a way to use functionality-containing files in a script. Such a file is called a **Module**. A module is a file containing Python definitions and statements. The file name is the module name with the suffix '.py' appended. Defintions from a module can be *imported* into other modules or into the *main* module.

Python makes no distinction between scripts and modules. These terms exist to draw a line between code that runs on its own and code that is used to help something else run. However, this line is not clearly defined, and a module can contain executable statements as well as definitions. Within a module (or script), the module's name (everything that precedes the '.py' suffix in the file name) (as a string) is available as the value of the global variable '__name__'.

1.2 Small Projects

For small projects a good structure can be:

- constants.py - This where all the constants will live
- helper.py - This is where all the functions/classes live
- run.py - This is where the script lives and is run. Enclose the script in a function called run() and then call the script in main. This should look like:

```
if __name__ == '__main__':
    run()
```

With projects organized in this way, the run.py file would need to import the constants and helper modules to access the functions, classes, and constants. So long as the helper and constant modules are in the same directory as the module importing them, they can be imported using *import*. There are two approaches that can be taken here. The first approach involves importing the modules without the use of a *. This requires we refer to the modules name or an assigned alias when calling classes or functions:

```
import constants
import helper

print(helper.check_if_prime(constants.EXAMPLE_PRIME_NUMBER))
```

The other approach invovles the use of the *. If we use the * when importing, then all the objects in the imported module will be automatically loaded into the programs namespace:

```
from constants import *
from helper import *

print(check_if_prime(EXAMPLE_PRIME_NUMBER))
```

While importing the objects into the namespace may be problematic for larger projects importing many modules, if careful, this approach should be perfectly fine. has many subdirectories which typically have their own specific functionality. Each of these subdirectories will need a __init__.py file (more on this later).

1.3 Large Projects

For large projects, a single helper and constants module is no longer practical. Instead, we actually should design something that mimics the structure of a Python package that you might find on PyPi. To allow for more organization, a Python package holds all its source code in various subdirectories. Then modules inside these subdirectories can be imported like:

```
from my_models import lstm_model # OR
import my_models.lstm_model
```

In addition, let's suppose that within the my_models subdirectory and lstm_model module there exists a class (or any other object) called TrainModel. Then we could just import that object with:

```
from my_models.lstm_model import TrainModel # OR
import my_models.lstm_model.TrainModel
```

In order for these subdirectories to be properly recognized, you must include an empty *__init__.py* file in the subdirectory. This way, Python recognizes that these subdirectories contain modules that can be imported.

In reality, large PyPi packages are much more complex. Let's take a look at the structure of the *scikit-learn* package on PyPi.

2 Logging

Without logging, most people check the program behaved correctly by embedding print statements at various points in the script. The better practice is to use *logging*.

There are five standard logging levels. The levels are ordered by severity:

- # DEBUG: Detailed information, typically of interest only when diagnosing problems.
- # INFO: Confirmation that things are working as expected.
- # WARNING: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
- # ERROR: Due to a more serious problem, the software has not been able to perform some function.
- # CRITICAL: A serious error, indicating that the program itself may be unable to continue running.

Logging functionality can be accessed through the standard library 'logging'

```
import logging
```

The default level for logging is set to 'WARNING'. This means it will capture everything that is everything at level 'WARNING' and above. Therefore, it includes 'WARNING', 'ERROR', and 'CRITICAL'. The default logging behaviour is to just log results to the console. If we want to log levels below WARNING we need to modify the default behavior. Here's a logging example:

```
add_result = add(num_1, num_2)
logging.warning('Add: {} + {} = {}'.format(num_1, num_2, add_result))
```

Output in the console will look like:

```
WARNING:root:Add: 10 + 5 = 15
```

Of course, this isn't a warning so lets instead change the default logging level

```
logging.basicConfig(level = logging.DEBUG)
```

Let's now change the logging output level to something more appropriate like 'DEBUG'

```
add_result = add(num_1, num_2)
logging.warning('Add: {} + {} = {}'.format(num_1, num_2, add_result))
```

As said earlier, default logging only outputs to the console. If you prefer to have a log file, include a file name in the logging.basicConfig mutator

```
logging.basicConfig(filename = 'test.log', level = logging.DEBUG)
```

The default log format is to include logging level is

```
LOGLEVEL:logger:message (e.g. DEBUG:root:Add: 10 + 5 = 15 )
```

To modify the log format to include things like date and time, include a 'format' argument in the logging.basicConfig mutator.

```
logging.basicConfig(filename = 'test.log', level = logging.DEBUG, format =
    '%(asctime)s:%(levelname)s:%(message)s')
```

Now the output in the log file will look like

```
2021-09-20 12:53:08,118: DEBUG: Add: 10 + 5 = 20
```

A sample program might look like this

```
import logging
logging.basicConfig(filename='employee.log', level = logging.INFO,
                    format= '%(levelname)s:%(message)s')
class Employee:
  def __init__(self, first, last):
      self.first = first
      self.last = last

      logging.info('Created Employee: {} - {}'. format(self.fullname, self.email))
  @property
  def email(self):
      return '{}.{}@email.com'.format(self.first, self.last)
  @property
  def fullname(self):
      return '{} {}'.format(self.first, self.last)

emp_1 = Employee('John', 'Smith')
emp_2 = Employee('Jane', 'Doe')
```

The 'employee.log' file should look like

```
INFO:Created Employee: John Smith - John.Smith@gmail.com
INFO:Created Employee: Jane Doe - Jane.Doe@gmail.com
```

3 Error Handling

A common type of error in Python is when a certain expression is not formed in inaccordance with the prescribed usage. Such an error is called a syntax error. Syntax errors are detected before the program is executed.

```
>>> print "hello world"
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("hello")?
```

Here we will be concerned with a different type of error, a runtime error, called an exception. This is type of error is detected by the Python interpreter while the program is executing. When these errors occur, Python writes a Traceback to the console, specificying the path to where the interpreter detected the error and the name of the error (e.g. FileNoteFoundError).

The fundemental building blocks of error handling are

```
try:
    pass
except Exception:
    pass
else:
    pass
finally:
    pass
```

The Python interpreter will execute whatever code is in the try statement. If an exception occurs, instead of the program stopping, we enter the except clause. The generalized exception is 'Exception'. This will capture any exception that occurs in the try block. However, this is not a best practice since the motivation for error handling is to address specific errors that occur. We will only enter the except clause if no exception is specified, the generalized 'Exception' is used, or the correct exception expression is used.

Note that we can include multiple except clauses. The Python interpreter will attempt each except statement until it finds one that allows the exception to enter (the three cases described earlier) then it will not attempt to enter any other except statements. Therefore, it's best practice to include the most specific except statements at the top and the most general exceptions (if you want them) at the bottom.

The *as* clause can be used to assign an alias to an exception name in the except statement. This is one of the three statements in Python which permit the use of *as*. The other two cases are in *import* statements and the *with* statement to name the resource being allocated. Here we see the usage for except statements:

```
try:
  f = open('test_file.txt')
except FileNotFoundError as e:
  print(e)
```

The *else* clause executes code if the *try* clause doesn't raise an exception.

The *finally* clause executes regardless of what happens, exception or not. This has a common use case of closing a connection to a database.

Note that we have the ability to raise exceptions ourself with the use of the *raise* clause. This typically is used inside of an if statement where a certain failure criteria was met. We can use the *raise* clause within a try statement or even outside the context of exception hanlding statements.

Here's an example that summarizes everything we've discussed:

```
try:
  f = open('my_data.txt')
  if f.name == "not_my_data.txt':
      raise Exception("This is the wrong file")
      raise ValueError("This is just to show that you can raise different types of exceptions")
  except FileNoteFoundError as e:
      print(e)
  except Exception:
      print("something else went wrong")
  else:
      print(f.read())
      f.close()
  finally:
      print("The code executed but it may or may not have contained an exception. I am the
          finally statement so I execute regardless.")
```

3.1 Creating Your Own Exception

To create your own exception you must create a custom exception class that inherits from the Exception class. The structure should be as follow:

```
class CoffeeTooHotException(Exception):
    def __init__(self, msg):
        super().__init__(msg)
```

Now you can raise a new exception like any other exception:

```
  if self.__temperature > 85:
      raise CoffeeTooHotException("Coffee too hot")
```