Financial Software Engineering
Lecture 3

Co-Pierre Georg
AIFMRM

July 13, 2018

Today

1. Modules and packages
2. Error and exception handling
3. Unit testing

Modules and packages

Moving from scripts to modules and packages

- Up to this point we've worked in a single script
- This becomes increasingly unfeasible as your code base increases in size
- Having one script with 10,000 lines of code is both bad practice and makes your code difficult for others to understand
- OOP makes it easy for us to move away from this
- We can write classes as their own scripts and import these classes and their accompanying methods when we need them

Modules and packages

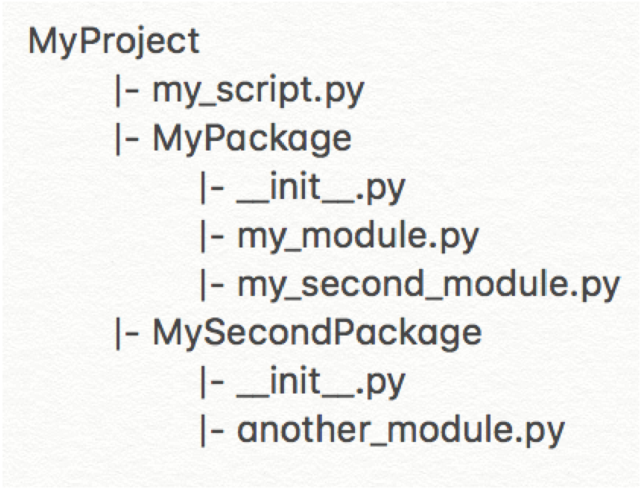
- Just like before, when we've use packages like pandas by using `import pandas` we can import other scripts in the same way
- Within this framework, we'll speak about **modules** and **packages**
- Modules $\rightarrow$ `.py` scripts that can be called by other `.py` scripts
- Packages $\rightarrow$ a collection of modules
- Practically, packages will represent folders in our workspace with many individual modules

Package folder structure

- When we point Python to these folders, we need to specify that these folders should be read as packages as opposed to ordinary folders or directories
- We do this by creating an empty Python script within the package directory, named `__init__.py`
- The `__init__.py` script tells Python that the folder represents a package

Modules and packages

- A typical folder structure will look something like this



```
MyProject
        |- my_script.py
        |- MyPackage
                |- __init__.py
                |- my_module.py
                |- my_second_module.py
        |- MySecondPackage
                |- __init__.py
                |- another_module.py
```

Importing a module vs running a script

- Using this structure, we can design our own packages and modules and import them into our programs / scripts
- Before $\rightarrow$ `.py` scripts were always called via the terminal or command line
- Now $\rightarrow$ `.py` scripts can be called via the terminal *or* via other scripts using the package/module structure
- As a result, a `.py` file has dual behavior - it is a script and a module
- In some cases, we may want to distinguish between these types of behavior

Importing a module vs running a script

- By default, we cannot make this distinction given Python treats scripts and module identically
- In other words, when Python executes (imports) a script (module), it always executes all code at indentation 0

```
def divisors(n):
    result = []
        start = 1

    while start < n:
        if n % start == 0:
            result = result + [start]
        start += 1

    return result

def main():
    n = input("Enter n: ")
    n = int(n)

    print(divisors(n))
```

Introducing __name__

- If this script was imported we would be able to use the methods `divisors()` and `main()`
- If we however called this script directly, nothing would happen
- Why? The script simply creates two classes, but does nothing else
- Perhaps, we'd want this script to execute the `main()` method when it is run directly
- We do however have a convenient way to make this distinction using the global variable `__name__`
- Every time a script is called/imported, Python defines a few special global variables automatically, of which the variable `__name__` is one

Introducing __name__

- Specifically
- → if a script is imported by another script as a module, `__name__` equals the name of the module being imported
- → if a script is called directly from the command line, `__name__` takes on a specific value, namely `"__main__"`

__name__ == "__main__"

- Using this global variable `__name__`, we can now distinguish between script and module functionality using an if statement

```
if __name__ == "__main__":
```

- Any code that we house in the if statement will only execute if the script is called directly from the command line
- Nearly every Python module you'll come across will have this convention implemented at the end of the script

Using __name__ for script-specific functionality

```
def divisors(n):
    result = []
        start = 1

    while start < n:
        if n % start == 0:
            result = result + [start]
        start += 1

    return result

def main():
    n = input("Enter n: ")
    n = int(n)

    print(divisors(n))

if __name__ == "__main__":
    main()
```

- If we import this module, everything is unchanged
- When we call this script directly however, it executes `main()`

Error and exception handling

Errors and exceptions

- Much of programming involves anticipating errors and exceptions
- This is especially true when users interact with your code
- If a user inputs a string value where you've asked for an integer value, it's important your code doesn't break
- Instead, errors like this can be anticipated
- This process of anticipating errors and exceptions and writing code to address them is called **error and exception handling**

Errors and exceptions

- Error and exception handling has 3 building blocks
- `try` → attempt to run this block of code which potentially returns an error
- `except` → if there is an error in the **try** block, execute this code
- `finally` → this code always executes, irrespective of any errors
- While any code in `except` will execute if an error or exception is thrown, it is also possible to catch specific exceptions
- `ValueError`, `TypeError`, `RuntimeError` etc
- See the Python documentation for more

Errors and exceptions

- In the following block of code, if a user inputs a string, we'll receive an error

```
val = int(input("Please enter an integer: "))
```

- In this case however, the error is a predictable one
- As a result, our program doesn't need to terminate; rather we can prompt the user until they submit an input of the correct format
- We'll do this using `try`, `except` and `finally`

Errors and exceptions

```python
try:
    val = int(input("Please enter an integer: "))
except:
    print("Looks like you did not enter an integer!")
    val = int(input("Try again-Please enter an integer: "))
finally:
    print("Finally, I executed!")
```

- Now, our code will try to execute and if it fails, it prompts the user with an appropriate message

Errors and exceptions

- Note that our `except` will catch any error
- In some cases though, we may have different responses for different types of errors
- Consider the example

```
a = 5
b = 0
c = a/b
```

- This case will throw an exception since we cannot divide by zero
- However, by setting `b = "h"`, we'll get a different error since we cannot divide and int by a string

Errors and exceptions

- We can control for various types of exceptions by specifying them in the except statements

```
a = 5
b = 0

try:
    c = a/b
except ZeroDivisionError:
    print("Can't divide by Zero!")
except TypeError:
    print("You provided something that is not an int")
finally:
    print('All Done!')
```

- Now we get an appropriate exception message for the appropriate error

Errors and exceptions

- As you build systems which users interact with they will invariably make mistakes
- Many of these mistakes will break your code if not controlled for
- Fortunately, many of these errors and exceptions are predictable
- Using `try`, `except` and `finally`, we can effectively control for these predictable errors and exceptions

Coding standards and conventions

PEP 8

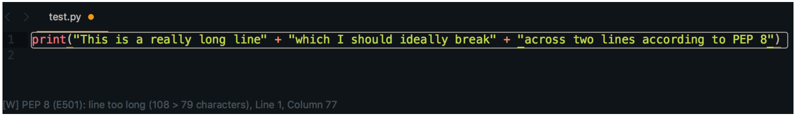
- As we've seen so far, indentation and structure matter in Python
- This need for structure extends beyond the language itself; the Python community emphasizes the need for code to be well structured
- This has been formalized in the development of the Python Enchancement Proposals, or **PEP8**
- $\rightarrow$ the de facto style guide for Python
- $\rightarrow$ set of coding and style conventions for Python to promote readability and structural consistency of code

PEP 8

- PEP8 contains a range of Python conventions. Some of the most standard include
- Line length - convention is maximum of 79 characters
  - If you choose another convention, state so in the preamble of your code
- Naming conventions
  - Modules / packages / functions / variable names: `lowercase` or `lowercase_with_underscores`
  - Classes: `CapitalizedWords`
  - Constants: `UPPERCASE`
- These conventions ensure that Python code is readable and that the structure is consistent
- Read up more here

PEP 8

- Most text editors and IDEs make it easy to adhere to PEP 8, by including a **linter**, either by default or via a plug-in
- → a tool, or set of tools, that analyze source code and flag programming and stylized errors, bugs etc.
- In this case, Python specific linters will also flag violations of PEP 8
- Importantly, these appear as comments on the screen as to throwing an error



```
print("This is a really long line" + "which I should ideally break" + "across two lines according to PEP 8")
```

```
[W] PEP 8 (E501): line too long (108 > 79 characters), Line 1, Column 77
```

Unit testing

Testing

- Testing refers to the practice of writing code, independent of your application, with the specific purpose of testing your application
- Importantly, testing does not imply checking that your code is correct, rather, whether your code performs as expected under a given set of conditions

Testing

- While linters will typically pick up syntax errors, and error and exception handling picks up errors which could your code to break, testing looks at whether your code returns the desired output
- This is important, since a function which takes a variable, `a`, and attempts to square it by using `a**3`, will still execute while clearly being incorrect

Why do we need to test code?

- In addition to checking the validity of your code, testing has 3 major advantages

1. Ensures that changes or additions to your code, don't break existing code
2. Forces you to think carefully about conditions that would break your code
3. Helps you clearly specify the conditions under which your code executes

- These factors are even more important in a collaborative or team based environment, where multiple people are using, adding to and editing a single code base

Unit testing

- With OOP languages, where multiple functions, methods and classes are all being used together it is very often difficult finding errors
- It is even more difficult finding the source of the error, since many errors can be occurring simultaneously
- In this course, we'll focus on one specific framework of testing, **unit testing**, aimed at making it easier to test our code for these errors

Unit testing

- Unit testing involves testing individual pieces or “units” of code
- These units could be functions, classes, modules etc.
- Unit tests make use of a sequence of **assertions** which test if some statement is true
- What makes unit testing special is the convention that units are isolated from other units of code

Unit testing

- In other words, these units are self-contained and can be executed and tested irrespective of the rest of the code base
- By breaking up your code base into smaller, testable chunks of code, unit testing makes the identification of errors easier
- As a result, while unit tests involve the testing of code during-or-post development, it also dictates how code is structured during development
- By implementing unit tests in the development process, unit testing encourages the development of modular code
- In Python we'll use the built-in `unittest` package to write programs to test our code

Unit testing

- Let's create a script which simply capitalizes a given input and call this `cap.py`

```
def cap_text(text):
    return text.capitalize()
```

- Let's now write a test script using `unittest` and call it `test_cap.py`
- test scripts should always have the prefix "test_"
- Modules that implement unit testing look through your subdirectories for files with this prefix

Unit testing

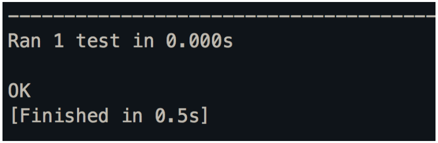
- To implement a unit test we write a test class which inherits from the `unittest` package
- This gives us access to a range of `assert` methods, which we'll use to test the output from our code with the desired output
- See the `unittest` documentation for a full list of `assert` methods

Unit testing

```
import unittest
import cap

class TestCap(unittest.TestCase):

    def test_one_word(self):
        text = 'python'
        result = cap.cap_text(text)
        self.assertEqual(result, 'Python')

if __name__ == '__main__':
    unittest.main()
```

Unit testing

- In `test_cap.py` we've written a method which passes a string to the method `cap_text()` and tests in the function returns the correct result
- We make this comparison using the `assertEqual()` method
- Running `test_cap.py` we receive the following output



```
-----------------------------------
Ran 1 test in 0.000s

OK
[Finished in 0.5s]
```

- `unittest` tells us our script has passed the test

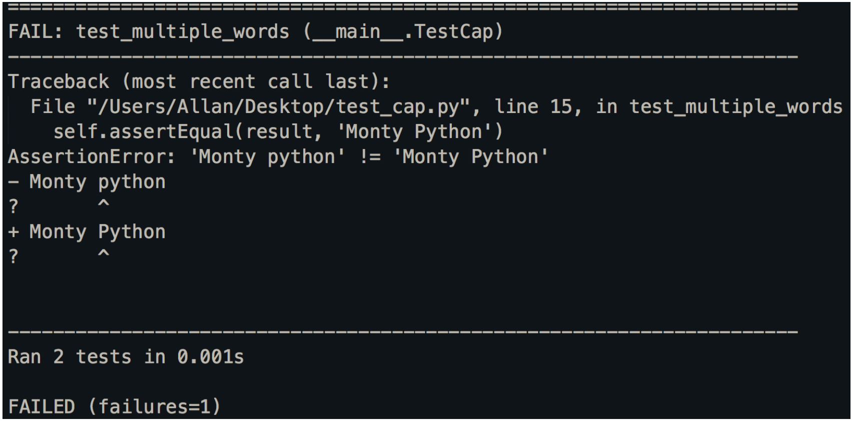
Unit testing

- We now add one more method to our test script to test how our function performs when we pass a string with multiple words

```python
import unittest
import cap

class TestCap(unittest.TestCase):

    def test_one_word(self):
        text = 'python'
        result = cap.cap_text(text)
        self.assertEqual(result, 'Python')

    def test_multiple_words(self):
        text = 'monty python'
        result = cap.cap_text(text)
        self.assertEqual(result, 'Monty Python')

if __name__ == '__main__':
    unittest.main()
```

Unit testing

- Now, running our test script we see the following output



```
======================================================================
FAIL: test_multiple_words (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Allan/Desktop/test_cap.py", line 15, in test_multiple_words
    self.assertEqual(result, 'Monty Python')
AssertionError: 'Monty python' != 'Monty Python'
- Monty python
?       ^
+ Monty Python
?       ^



----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Unit testing

- `unittest` now tells is us that 1 test failed, shows us what condition was not met and in this case, it also shows us what characters were responsible for the test failing
- In this case, our function only capitalized the first letter of the first word

Unit testing

- We'll use unit tests throughout this course $\rightarrow$ a good convention is to write your test as soon as you create any new classes
- Before running your main script, you will run your test script to check that your methods all pass their respective tests
- It is also good practice to run your test script at the end of every development session
- See the `unittest` documentation for more information
- While we'll use `unittest`, feel free to explore the other packages that offer similar (improved) functionality, most notably nose and pytest