Exercise 06 - Unit Tests and Packaging

Deadline

December 07, 2020 - 12:00/noon (UTC+1:00)

Submission Guidelines

1. Clone your repository to your local workstation/computer if you have not done so
2. Structure and work on your submission.
   - Create a directory (folder) for the exercise sheet, e.g. name the directory ”exercise06” for Exercise 06.
   - Place all relevant files/scripts in this newly made directory
   - Do not include the `.git` folder!
3. Commit your work to the git repository.
4. Create a git tag.
   - Tag name must be equivalent to ”SubmissionExerciseSheet06”.
   - Tag can be made via the command line or using the GitLab GUI

Grading (10 pts):

Task	**1**	**2**	**3**	**4**
Points	4	1	5	(+1)

Using Previous Homework

This exercise requires you to build on the work done for Exercise05. If you did not do or finish Exercise05, or do not wish to use your work, you may use the provided API NetworkX script in the sample_files folder. This script will be made available on December 02, 2020.

Tasks

Let's Make a Package

You will now bundle all of your scripts and modules together into a package which can be installed on any user's system. You will have to properly configure specific settings to ensure everything is working correctly, but by the end of this task, you should be able to execute your CLI commands from the command line without having to call specific files.

Task 1 - *Creating the Package Folder* (4 pts)

First you will need to create a folder that has a typical package layout. There are many ways one can do this, I personally use `cookiecutter`, a python package that generates pre-made folder layouts for you with all of the needed files. For this task, I have already designed a basic package folder for you to use, but if you would like more information on how to use `cookiecutter` in the future, you can check out the cookiecutter handout.

1. From the `templates/` folder, copy the `plab2_template` folder (and its contents) to your `exercise06/` folder that will use for submission (0.5 pts)
    - Excluding any files you might have already copied over, your repository's `exercise06/` should look like this:

```
exercise06/
└── plab2_template
    ├── plab2
    │   └── __init__.py
    ├── README.md
    ├── setup.py
    └── tests
        ├── __init__.py
        └── test_plab2.py
```

    - Copy all of your scripts/modules to `exercise06/plab2_template/plab2/`
2. Configure the `exercise06/plab2_template/setup.py` file you copied over (1 pt)
    - Add your name and email to appropriate sections (`author` and `author_email` in the `setup()` method parameters)
    - Verify that the `entry_points` defined in the `setup()` function are correctly configured. These should point to the primary method you call to execute your CLI in your `cli.py` module. For many this may be called `main` which means you do not have to change anything, but if your CLI entry point is named differently, change "main" in the `entry_points` definition to what you named it
3. Fill in the `requirements` list on line 7 of the `exercise06/plab2_template/setup.py` file (1 pt)
    - This list should be comprised of the dependencies (other packages) required to run yours. Ideally, this would not include the dependencies of the dependencies (e.g. pandas requires numpy, but you can simply put pandas in your requirements list and it will install numpy automatically without you having to also include it in the list)
    - *Hint* - If you took the time to setup your virtual environments from the previous lesson, then using `pip list` will show only the packages you have been working with inside your homework virtual environment
4. Modify your imports in all of your files so they will work in a package setting (0.5 pts)
    - e.g. In your `cli.py` file, you may import a method from your `utils.py` module by writing a line such as this:

```
from utils import download_file
```

    - This will fail and your package will not run. You need to tell Python *exactly* from where to import it. You have two options:
        a. *Relative Import* Place a "." in front of `utils` so that way it knows to look in the same directory as the `cli.py` file

b. *Absolute Import* Since we are naming this package "plab2", Python will now look in the plab2's site-package directory for the utils module

5. Verify you can install your package and your CLI is executable without having to call your `cli.py` file directly (1 pt)

   - If everything is configured correctly, navigate to your `exercise06/` directory and execute: `pip install plab2_template/`
   - Once it is installed, you should be able to execute all of your CLI commands using `plab2`
     - e.g. `plab2 compile /path/to/ppis.csv nodes.tsv edges.tsv --enrich`
   - If you are having trouble getting this to work in Windows, try executing it from either PyCharm, a Conda Prompt, Git Bash Terminal, or Powershell

Relative Import

```
from .utils import download_file
```

Absolute Import

```
from plab2.utils import download_file
```

Test Everything

At this point, your codebase has grown by a substantial amount and it is becoming difficult to make sure all of your conditions for all of your methods are still working as intended. This is why developers typically write *unit tests* to check whether parts of their code are running and behaving as intended. In these tasks, you will write a series of unit tests that will check various sections of your methods and set up your configuration so they can all be run at once.

Task 2 - *Setting Up the Test Files* (1 pt)

You may have already noticed that there is a `tests/` folder in your package directory. This is where we will store all of our tests.

1. Create a test file for each module in your package. (0.5 pts)

   - All of your test files should have the same naming structure: `test_{module name}.py` where "module name" is the name of file from your package which will be tested e.g. "utils", "cli", "network", etc

   - Give each test file a docstring at the top describing what it is testing. Here is what it looks like at the top of my `test_cli.py` file:

     ```
     """Tests for the CLI."""

     import click
     from plab2 import cli

     # My tests and other imports
     ```

2. Create test classes within each of your test modules (0.5 pts)

   - e.g. In your `test_utils.py` module - make a `TestUtils` class
   - Do the same for the for the others
   - *Note* - A test class allows unit testing to be performed very easily using the `pytest` library
   - *Note* - while one can combine all of their unit tests into one test class for the whole module they are testing, you can (and should) break it up into multiple test classes when it appropriate to do so
     - In your case, you may have multiple classes in your `network.py` module, each of which can have their own test class in `test_network.py`

Task 3 - *Writing the Tests* (5 pts)

Now it is time to implement some basic unit tests for your code. If you need an idea of how a test file should look like before you get started, check out the example test file. If you need help setting up PyCharm for testing or running tests from the command line, see the testing handout.

General Guidelines

- Try to use the built-in `assert` statement to verify that your methods are returning what you expect them to. You can use several assert statements in one method!
- Have each unit test method test an actual method in your code
- All test methods should use the naming structure `test_{some method}`
- Don't forget your docstrings!

1. Write unit tests for the methods in your `utils.py` (or wherever you are storing methods for downloading/parsing data from HGNC) (1.5 pts)
    - It should check whether a file can be successfully downloaded (use a favorite HGNC symbol to check)
    - Verify that a cache file is created if information is successfully downloaded
    - Check whether it can read/parse a particular cache file and return the correct information
        - For example, lookup what the HGNC ID, Ensembl Gene ID, and UniProt IDs should be for "TNF" and "assert" whether your method returns the expected values
2. Write unit tests for the methods in your `network.py` (or wherever you are storing the methods for creating a network and generating images) (3 pts)

- Write unit tests that check the following:
    - Whether a graph can be imported using either a PPI file or node/edge lists
        - Verify both methods are working by checking it against how many **known** edges there should be
    - Can enrich the graph with RNA/DNA nodes as well as the associated `translated` / `transcribed` edges
        - Use `assert` to check the counts. #Protein == #RNA == #DNA == # `translated` == # `transcribed`
    - Can generate an image
        - Have it check whether a file now exists at the specified location
    - Whether it can find the shortest path between a given pair of HGNC symbols
        - Use a defined list (path) of symbols and compare it to what the method produces
    - If it can calculate summary statistics for a graph
        - Check the counts
    - If it can export the summary statistics as CSV, TSV, and JSON

BONUS Task 4 - *Writing Tests for the CLI* (+1 pt)

Of course, testing your CLI is also important to make sure that your software can be used to its fullest extent. However, because of how it is designed, and that we are using Click, testing it properly requires a bit of extra work. It will be up to you to figure out how to write a working unit test for your CLI commands, but it will require using a "runner". To get the bonus point, write a unit test for your `compile` CLI command that does the following:

- Executes the `compile` CLI command
- **Asserts** that it has the proper exit code (should be "0")
- **Asserts** that node/edge lists are generated
- That the correct help message is generated for the `enrich` option when someone calls `plab2 create --help`
- Removes any generated node/edge lists and **asserts** that they are removed