Exercise 05 - Logging and APIs

Deadline

November 30, 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 ”exercise05” for Exercise 05.
   - 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 ”SubmissionExerciseSheet05”.
   - Tag can be made via the command line or using the GitLab GUI

Grading (10 pts):

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

Using Previous Homework

This exercise requires you to build on the work done for Exercise04. If you did not do or finish Exercise04, or do not wish to use your work, you may use the provided OOP NetworkX script in the sample_files folder. This script will be made available on November 25, 2020.

Code Design

In the previous homework, you modified your code to use an OOP schema. From this point on, you may choose how to structure your code when no direct instructions are given. Though it may be tempting to revert back to using a purely functional programming style, keep in mind the reasons why one may choose to use OOP instead during this assignment.

Tasks

Task 1 - *Cache It* (1 pt)

For this assignment, we will be generating and storing several files that we would like to keep for future reference and to avoid having to constantly query a remote server. For that, we need to create folders for storing these files on the user's computer. Conventionally, cache directories and configuration files are stored in the user's home directory in a "hidden" state, meaning they start with a period ".". Folders and files that start with a period are hidden on Unix systems (e.g. Linux or OS X), but not on Windows. Here we will create a hidden cache directory to store our data and log files.

- Create a new file called `startup.py` and write a script that will:
  - Identify the user's home directory
    - This needs to work regardless of the user's operating system!
  - Generate the following folder structure:
    - Create a folder in the user's home directory called `.ex05/`
    - Create a subdirectory inside `.ex05/` labeled with your GitLab handle
    - Generate 2 subdirectories within `.ex05/{gitlab-handle}/` called `logs/` and `data/`

        - Make sure the **absolute paths** of the `logs/` and `data/` are saved as variables so they can be easily imported by other modules
    - Example layout of the folder structure in the home directory:

```
.ex05/
└── bschultz/
    ├── data/
    └── logs/
```

- *Hint* Make sure your script does not override existing folders or crash if the folders are already there. Save a file inside these folders and run the script multiple times to make sure it only generates new folders if none are there
- Make sure this script is executed whenever someone uses the CLI

Task 2 - *Log It* (2 pts)

When creating and executing programs, it is useful to incorporate and use logging files to track the progress of executed code or a running system. Being able to know how far a process was able to get to can help in debugging the issue in the code, but logging also allows one to handle and log specific, non-system breaking errors, format messages however desired, and assign the level of severity of the logged message. Here, you will introduce a `logger` into your code and track when specific tasks are executed. For more information on the `logging` library, read the *Logging* section in the information handout.

1. In your `startup.py` file, add configuration settings to your root logger (0.5 pt)
    - Set up your root logger's *basic configuration* with the following information:
        - Severity level = `DEBUG`
        - Everything is logged to a file in the `logs/` folder created in Task 1 (name of the file does not matter, but is should end in `.log`)
        - Log messages are formatted using the following string `'%(asctime)s - %(name)s - %(levelname)s - %(message)s'`
2. Create unique loggers for every module at the top of each of the files (0.5 pts)
    - Use the `logger = logging.getLogger(__name__)` create the logger with the name of file (remember special variables?)
3. Add logging to points in your code where the following is executed and make sure it is logged in your log file using the indicated severity level (1 pt):

Point in Code	Logging Message Information	Logging Level
Whether a PPI file was passed or Node/Edge lists	Which file(s) accepted as input (PPI or node/edge lists)	`INFO`
New node/edge files are generated	New node/edge files were made and their locations	`INFO`
Graph image generated	New graph image was generated and its location	`INFO`
If no paths are found between two given protein nodes	"No paths found between {symbol1} and {symbol2}"	`WARNING`
Wrong file format is used for output image	Use your error message created for Exercise03 Task5	`ERROR`

Task 3 - *Download It* (2 pts)

Though our network contains a lot of useful interaction information, HGNC gene symbols are not the optimal manner of identifying what is in the network. HGNC symbols change over time, therefore it is a wise idea to include standardized identifiers

for each of our nodes whenever possible. Since we do not have this information readily available, we must use an API service to get what we need.

1. Using the `requests` library, write code designed to download information using the HGNC REST API and write it to a file in your `data/` cache folder created in Task 1 (1.5 pts)
    - Your methods should be able to accept an HGNC symbol and download the information from the HGNC server using the `http://rest.genenames.org/fetch/symbol/:symbol` API request
        - `:symbol` should be replaced when using this
    - Include code that checks the status code of the response
        - If the response status code is acceptable (e.g. 200), an **info** log entry is made
        - If the response status code is unacceptable (e.g. not 200), then an **error** log entry is made
    - Downloaded data should be saved using the HGNC symbol as the name of the file, e.g. information on TNF should be saved to `tnf.json`
        - Two formats are available from HGNC: JSON and XML. You may download and store using whichever one you wish, just be consistent and label the saved file with the proper extension (e.g. `.json` and `.xml` respectively)
        - *Note:* As noted above, HGNC symbols change over time so naming our saved data using the symbol is not good practice, however, it will be okay for the purposes of this exercise
    - Make sure you **check** whether the information you downloaded is useful i.e. if a result was found. If no results for a HGNC symbol were found, have your logger create a **warning** stating that fact.
2. Create a file called `utils.py` and move all of your methods related to downloading and storing information to this module. For more information on utility methods, read the *Utility Methods* section in the information handout (0.5 pts)

Task 4 - *Extract It* (2 pts)

1. Create a new method in your `utils.py` module to extract identifier information for a given HGNC symbol using the information obtained from the HGNC REST API (1 pt)
    - This method should return a dictionary with the following information:
        - HGNC ID
        - Ensembl Gene ID
        - UniProt IDs
    - Downloading data over and over again is very inefficient. Make sure your code is designed to first check whether the information is already in the `data/` folder before querying the HGNC REST API
2. Add a CLI command called `info` that takes an HGNC symbol as an input parameter and: (1 pt)
    - prints these identifiers to STDOUT
    - provides the link to the HGNC gene symbol report page for that particular symbol if one exists

Task 5 - *Enrich It* (3 pts [+1 Bonus])

Now that you have created the tools for gathering additional information on the nodes in your graph, it is time to integrate these features into your `Network` class methods. These new features should be executed when the `enrich` option is executed i.e. if the user wishes to add RNA/gene nodes to their network

1. Modify your methods related to the `enrich` option to also gather identifier information for every HGNC symbol in the network (1 pt)
    - This should involve using the method(s) you created in Task 4
    - Make sure that you use proper etiquette when querying the HGNC API
        - You **will** be deducted points if you do not adhere to HGNC's guidelines
        - The `time.sleep()` method may be useful here…
    - *BONUS* - Use the `tqdm` library to track the progress (+1 pt)
2. Adapt your code so that the following information is added to the generated node TSV file (1 pt):
    - UniProt identifiers are included in the protein nodes' metadata
    - HGNC IDs and Ensembl Gene IDs are included in the gene nodes' metadata
3. Add additional options to your `create` CLI command (which may involve modifying your methods) so that users can choose to have identifiers displayed in their generated network image instead of HGNC symbols (1 pt)
    - Protein nodes should have their UniProt ID displayed
        - In the case of multiple UniProt IDs, only show the first one
    - Gene nodes should have their HGNC ID displayed

- If a node does not have an ID, then the original symbol is kept