workflow

exercise 5 .

If you want to follow this document's changes, click on the speech bubble next to the **Share** button on the top right, select **Notifications**, and click **All**.

This exercise has 2 parts:

1. [Exercises about modules and packages](#_zs343f6f01l);
2. [Revising our system as a full-fledged project with tests, documentation, and even continuous integration and deployment](#_p948uvfqhvp0)!

To begin:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** clone git@github.com:advanced-system-design/exercise-5.git  $ **cd** exercise-5/  $ ./scripts/install.sh  $ **source** .env/bin/activate | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** remote remove origin  $ **git** remote add origin \ git@github.com:advanced-system-design/exercise-5-\*.git | |  |
|  |

And when you're done:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** add .  $ **git** commit -m 'Submitting exercise 5.'  $ **git** push origin master | |  |
|  |

# exercises .

1. Implement **e1.py** so that it prints **I'm a script!** when its run as a script, and print **I'm a module!** when imported as a module.
2. Create a package **p**, with the subpackages **x** and **y**.  
   The **x** subpackage must implement the **A** class in the **a** submodule; the **B** class, which subclasses **A**, in the **b** submodule; and export both classes.  
   The y subpackage must implement the **C** class, which also subclasses **A**, in the **c** submodule; the **D** class, which subclasses both **B** and **C**, in the **d** submodule; and export both classes.  
   The package **p** must export all four classes: **A**, **B**, **C** and **D**.  
   It must also be executable: it should accept a single argument — **a**, **b**, **c** or **d** — create an instance of the appropriate class, and print **created *instance***. If no arguments are provided, it should print a usage message.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **python** -m p a  created <p.x.a.A object at 0x105760f40>  $ **python** -m p b  created <p.x.b.B object at 0x105760f40> | |  |
|  |

1. In **e3.py**, implement **load** so that it receives a path to a module, and imports it dynamically, by creating a **types.ModuleType** object and **exec**uting the module's code in its **\_\_dict\_\_**.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **open**('/tmp/m.py', 'w').write('x = 1')  >>> m = load('/tmp/m.py')  >>> m.x  1  >>> m.\_\_name\_\_  'm' | |  |
|  |

1. In **e4.py**, implement the **in\_directory** context manager so that when its context is entered, the current working directory changes to the specified path, and when it's exitted, it restores to its original value.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> os.getcwd()  '/home/user'  >>> **with** in\_directory(**'/tmp'**): ... **print**(os.getcwd())  '/tmp'  >>> os.getcwd()  '/home/user' | |  |
|  |

Pay attention to the question of importability: within the context of some directory, we'd like its contents to be importable, as if we were really executing within it.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **open**('/tmp/m.py', 'w').write('x = 1')  >>> **import** m  ModuleNotFoundError: No module named 'error'  >>> **with** in\_directory('/tmp'): ... **import** m  >>> m.x  1 | |  |
|  |

1. Implement **e5.py** as a magic module that, once imported, can be invoked as a function, accessed as a dictionary, and has the representation **magic module**.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **import** e5  >>> e5  magic module  >>> e5(1)  1  >>> e5[1]  1 | |  |
|  |

1. In **e6.py**, implement the following tests of the **simple** module:

* **test\_mul** should check that **mul** performs multiplication well on positive integers.
* **test\_mul\_zero** should check that **mul** performs multiplication well on zero and other numbers.
* **test\_mul\_negative** should check that **mul** performs multiplication well on a negative number and other numbers, and on two negative numbers.
* **test\_mul\_fractions** should check that **mul** performs multiplication well on a fraction and other numbers.
* **test\_div** should check that **div** performs division well on positive integers.
* **test\_div\_negative** should check that **div** performs division well on a negative number and other numbers, and on two negative numbers.
* **test\_div\_fractions** should check that **div** performs division well on a fraction and other numbers.
* **test\_div\_error** should check that div raises a **ZeroDivisionError** when dividing by zero.
* **test\_greet** should check that **greet(name)** prints a hello message to the name provided to it.
* **test\_greet\_stranger** should check that **greet()** prints a hello message to a stranger if no name is provided to it.
* **test\_read\_value\_int** should check that **read\_value** can successfully read the integers **1** and **2** from a file (use **write\_value** to write it there).
* **test\_read\_value\_str** should check that **read\_value** can successfully read the strings **foo** and **bar** from a file (use **write\_value** to write it there).
* **test\_read\_value\_list** should check that **read\_value** can successfully read the lists **[1, 2, 3]** and [] from a file (use **write\_value** to write it there).

1. In **e7.py**, implement the **path** fixture so that the test passes.

# back to our system

OK, it's time to turn the academic exercise we've been developing so far into an actual, full-fledged software project, with a repository, tests, documentation, and CI/CD pipeline of its own!

This part of the homework will not be directly graded at this point — instead, this ongoing project will be evaluated at the end of the semester, and form 20% of your grade based on its overall quality, consistency, software design, etc.

First, open a public GitHub repository for your project.  
The reason this repository has to be public, is that Travis CI, ReadTheDocs and similar CI/CD services are only free for open-source projects.  
Think of any name you'd like, and organize your project accordingly; for example, for my [Foobar](https://github.com/dan-gittik/foobar) project, I have:

* A **README.md** file with installation instructions and basic usage.
* A **setup.py** with my project's settings.
* A **requirements.txt** with my project's requirements.
* A **scripts/** directory with an **install.sh** script that creates a virtual environment, installs the requirements, and drops a path file to make the project importable from anywhere.
* The **foobar** package, with the **Foo** and **Bar** classes, and a command-line interface (based on the [**click**](https://click.palletsprojects.com/en/7.x/) library) to operate them both.
* A **tests/** folder with some tests.
* A **.gitignore** file to avoid including the virtual environment (or pytest's cache, or the documentation's build folder).

Please make sure your project includes all of the above (you can write your own tests, or borrow the ones I've provided in previous exercises for now).

The package API should be exposed as follows:

* The **run\_server**, **upload\_thought**, and **run\_webserver** functions, as well as the **Thought** class, should be available from the root of the package.
* The **Connection** and **Listener** classes should be available from a **utils** subpackage.
* A CLI should be available to the server, the client and then webserver from the package level (e.g. as **python -m *yourproject* [COMMAND] [ARGS]**).  
  You can stick with the **cli.py** we've developed, but I'd recommend ditching it in favor of [**click**](https://click.palletsprojects.com/en/7.x/) (especially since we'll be adding to that CLI in the future).
* You can keep the **website.py** we've developed, but again I'd recommend ditching it in favor of [**flask**](https://www.fullstackpython.com/flask.html) (again, this will make future development easier).

Furthermore, please make sure all the code in your project conforms to [flake8](http://flake8.pycqa.org/en/latest/).

Now, link your GitHub account with [Travis CI](https://travis-ci.org/), and create a [Travisfile](https://docs.travis-ci.com/user/languages/python/) that will cause pytest to run on your tests directory (we're only working with python 3.8).

Every time you push something to your repository, it should trigger Travis CI to run all your tests — you can then add a [build status badge](https://docs.travis-ci.com/user/status-images/) to your README.

Now, link your GitHub account with [CodeCov](https://codecov.io/), and make sure you're seeing a coverage report for your project whenever you push an update. You can even add a coverage status badge!

Finally, use [Sphinx](http://www.sphinx-doc.org/en/master/) to generate a documentation project in the **docs/** directory, write a bit of documentation, make sure it compiles and looks presentable, and link your account with [ReadTheDocs](https://readthedocs.org/).

Note that in order for the documentation to be rebuilt whenever you push a new commit, you'd have to set up some [webhooks](https://docs.readthedocs.io/en/stable/webhooks.html), so please do.