2_Test_Driven_Development

February 28, 2024

Programming Techniques in Computational Linguistics II – FS24

1 Lecture 2: Test Driven Development

1.1 Topics

- Virtual Environments
- Logging
- Testing

1.2 Learning Objectives

You are able to …

- explain the purpose of **virtual environments** and typical scenarios when they're used
- create, replicate and share **virtual environments**
- log information using the `logging` module
- write simple **unit test** using `pytest`
- name two advantages of **Test Driven Development**

2 Virtual Environments

2.1 Common Problems in Package Management

- Will upgrading a package break some process on my system?
- Will installing a new package require the downgrade of another package in my system?
- Does installing a new package require a different python version?

→ Do not install on the system level, but in a virtual environment

2.2 Advantages of Virtual Environments

- Allow different Python versions and dependencies for different projects
- Projects are easier to install (more portable)
- Project can be tested for multiple versions of dependencies

2.2.1 `venv` module

- Part of the standard library of Python3
- Will create an environment based on the system level python version

```
# Creating a virtual environment:
    $ python3 -m venv myenv
# Activating an environment:
    $ source myenv/bin/activate
# Installing a dependency inside the environment:
    (myenv) $ pip install <dependency>
# Deactivating the environment:
    (myenv) $ deactivate
```

For Windows users: Link

2.2.2 Useful commands

```
$ which python
```

Prints the path of the active Python binary. The virtual environment has been successfully activated if the output is not `/usr/bin/python` but `myenv/bin/python`

```
$ python --version
```

Prints the version of the active Python binary

2.2.3 Alternative Tools

- `virtualenv` can install any Python version (Python 2.7+ and Python 3.3+)
- `Conda` is part of the Anaconda distribution of Python and is Anaconda's default package manager

Use either if you need a specific Python version, otherwise, `venv` is sufficient in most situations.

2.2.4 Describing Dependencies

How do I tell users or collaborators of my project which Python packages it depends on?

```
$ pip freeze
```

Output installed packages in requirements format.

Example

```
bleach==3.1.0
Django==3.0.2
jsonlines==1.2.0
langdetect==1.0.7
matplotlib==3.1.2
nltk==3.4.5
numpy==1.18.1
pandas==0.25.3
requests==2.22.0
...
```

Requirements File Creating the file from an existing virtual environment:

```
$ pip freeze > requirements.txt
```

Collaborators can install the dependencies into their own virtual environment using:

```
$ pip install -r requirements.txt
```

Demonstration

1. Download folder `exercises.zip` from OLAT and unzip it. (`Materials/Lectures/02_Test_Driven_Development`)
2. Create a new virtual environment and activate it.
3. Use pip to install the packages in `requirements.txt`
4. Run `plot.py` to confirm the installation works.

Commands

```
python -m venv testvenv

source testvenv/bin/activate

pip install -r requirements.txt

python plot.py

# uncomment lines in plot.py

pip install matplotlib

pip freeze > requirements.txt
```

Syntax of `requirements.txt`

```
# Install the latest version:
mypkg

# Install a specific version (version pinning):
mypkg==1.0

# Install the latest 1.x version:
mypkg<2

# Install a package directly from a Git repository:
git+https://gitlab.uzh.ch/mypkg@sometag
```

Anecdote from final exam 2023 One task required converting a list of strings to `datetime` objects.

The requirements file specified `pandas==2.0.2` but some students used `pandas==1.x`. With the version change, the behaviour of the `pandas.to_datetime` function was changed as well.

pandas version 1.x:

```
pandas.to_datetime(["12.24.2012", "24.12.2020", "1999-05-25"])
>> DatetimeIndex(['2012-12-24', '2020-12-24', '1999-05-25'], dtype='datetime64[ns]')
```

pandas version 2.0.2:

```
[6]: import pandas
     pandas.to_datetime(["24.12.2020", "12.24.2012",  "13.13.13"], errors="coerce")
```

```
/var/folders/j2/0s11d_v57t74gy_39kl316_c0000gn/T/ipykernel_66012/3744611769.py:2
: UserWarning: Parsing dates in %d.%m.%Y format when dayfirst=False (the
default) was specified. Pass `dayfirst=True` or specify a format to silence this
warning.
  pandas.to_datetime(["24.12.2020", "12.24.2012",  "13.13.13"], errors="coerce")
```

```
[6]: DatetimeIndex(['2020-12-24', 'NaT', 'NaT'], dtype='datetime64[ns]', freq=None)
```

In pandas version 1.X, `format='mixed'` is true by default. This has changed in version 2.X.

Many students used the flag `errors='coerce'` to ignore invalid dates.

As a result, the majority of dates could not be parsed.

TL;DR: Using the wrong package version could cost you points in the exam.

3 Logging

Diagnostic logging Records events related to the application's operation. If a user calls in to report an error, the logs can be searched for context.

Audit logging Records events for analysis. A user's transactions (such as a clickstream) can be extracted and combined with other user details (such as eventual purchases) for reports.

3.0.1 Why not just use print?

- Logging can offer more information (filepath, linenumber, timestamp …)
- Logging is done on different levels on verbosity (info, warning, error, …) and can be selectively activated or silenced
- But in many cases, using print is easier and faster

```
[7]: import logging
```

```
[10]: logging.debug('The program has now reached this line.')
      logging.info('Everything is working as expected.')
      logging.warning('Now something unexpected has happened.')
      logging.error('The program has been unable to do something.')
      logging.critical('The program is unable to continue running!')
```

```
DEBUG:root:The program has now reached this line.
INFO:root:Everything is working as expected.
```

```
WARNING:root:Now something unexpected has happened.
ERROR:root:The program has been unable to do something.
CRITICAL:root:The program is unable to continue running!
```

[9]:
```
logging.getLogger().setLevel(logging.DEBUG)
```

4 Test Driven Development

4.1 Question

Alice has the task to implement a system which converts German singulars to plurals (e.g. Baum → Bäume).

After having completed the code, she tries a handful of words and all seem to be inflected correctly. Alice is happy and sends the code to her coworker.

Is Alice's way of checking the system efficient? If not, what are the alternatives?

In general, anything that is repeatable can also be automatically tested. For example:

- The integrity of the system (Does it work?)
- The functionality of the system (Are words correctly inflected?)
- The error handling (Can the system recover from an error?)
- The performance (Can the system handle many requests?)
- The installation process (Will the coworker be able to install it?)

4.2 Types of tests

- Unit testing: Isolated tests of functions or classes
- Integration testing: Test how different modules/components interact
- Regression testing: Make sure changes to the code work as intended / don't have side effects

4.2.1 Unit Tests in Python

We use the module `pytest` in this course.

```
$ pip install pytest
```

Run:

```
$ pytest
```

`pytest` will find all files with the name `test_*.py` or `*_test.py` in the current directory and its subdirectories and run all test functions (also starting with `test_*`).

File: *examply.py*

```
def normalize_swiss_german(sentence: str) -> str:
    """Replace German Esszett 'ß' with 'ss' in input sentence
    and return normalized sentence.
    """
    return re.sub("ß", "ss", sentence)
```

File: *test_examply.py*

```
def test_esszett_lower() -> None:
    test_input = "Die Vorlesung findet statt an der Andreasstraße."
    expected = "Die Vorlesung findet statt an der Andreasstrasse."
    assert normalize_swiss_german(test_input) == expected
```

The `assert` Keyword `assert` is used for debugging, and to catch errors early.

```
[13]: i = 1.
```

```
[14]: assert type(i) == float, f'i is type {type(i)}, not float'
```

Assert that exception is raised File: *examply.py*

```
def foo(d: dict) -> None:
    print(d["a"])
```

File: *test_examply.py*

```
def test_key_error() -> None:
    with pytest.raises(KeyError):
        foo(dict(b=2))
```

4.2.2 Interactive Example: Anonymization

"The **Swiss SMS corpus** consists of 25'947 SMS (~650'000 tokens), which were sent in by the Swiss public in 2009/2010."

Stark, Elisabeth; Ueberwasser, Simone; Ruef, Beni (2009-2015). Swiss SMS Corpus. University of Zurich. www.sms4science.ch

Interactive Example: Part 1 "In an effort to remove information about phone numbers, bank accounts etc., all numbers with three and more digits were removed and each digit was replaced with one N.

The phone number `079 987 65 43` would thus become `NNN NNN 65 43`, while `0799876543` would be `NNNNNNNNNN`."

Let's use ChatGPT to generate tests: Prompt:

write a pytest test function for a function that anonymizes phone numbers and fulfills the following criteria:

"In an effort to remove information about phone numbers, bank accounts etc., all numbers with three and more digits were removed and each digit was replaced with one N. The phone number 079 987 65 43 would thus become NNN NNN 65 43, while 0799876543 would be NNNNNNNNNN."

Interactive Example: Part 2 "All email adresses were removed and replaced with `xxx@yyy.ch`, while keeping the number of characters.

`info@uzh.ch` would therefore become `xxxx@yyy.ch`, while `admin@google.com` would become `xxxxx@yyyyyy.com`."

Let's use ChatGPT to implement the function: Prompt:

write the function anonymize_email_address, so that it will pass the provided test cases:

```
EMAIL_TESTCASES = [
    ("info@uzh.ch", "xxxx@yyy.ch"),
    ("simple123@example.ch", "xxxxxxxxx@yyyyyyy.ch"),
    ("very.common@example.org", "xxxxxxxxxxx@yyyyyyy.org"),
    ("other.email-with-hyphen@example.info", "xxxxxxxxxxxxxxxxxxxxxxx@yyyyyyy.info"),
    ("x@example.cl.ch", "x@yyyyyyyyyy.ch"),
]

@pytest.mark.parametrize("test_input,expected", EMAIL_TESTCASES)
def test_email(test_input, expected) -> None:
    assert anonymize_email_address(test_input) == expected
```

4.2.3 Coverage

Code coverage is a measurement of what percentage of the code is being tested by your tests.

You need to install the `pytest-cov` plugin.

```
$ pip install pytest-cov
```

Run `pytest` with coverage:

```
$ pytest --cov

Name                Stmts   Miss  Cover
-------------------------------------
my_module.py           10      1    90%
my_other_module.py   20      0   100%
-------------------------------------
TOTAL                  30      1    97%
```

Display lines without coverage:

```
pytest --cov-report term-missing --cov


Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
my_module.py           10      1    90%   28
my_other_module.py   20      0   100%
-------------------------------------------------
TOTAL                  30      1    97%
```

4.3 Test-Driven Development

Test Driven Development (or TDD) is a software development technique whereby developers write test cases before they write any implementation code.

4.3.1 TDD Cycle

4.4 Take-home messages

- You know how to use virtual environments and why they are important
- You are familiar with the `logging` module and it's core functionalities
- You know the principles of test-driven development and are able to implement unit tests.