
VELDI
KOMPETENS
VELDI

Python 3A
Part 1
-
Welcome to the course!

VELDI

What you will learn in this course

- Learn how to setup a new project in a structured way
- Learn about test driven development (TDD)
- Learn about unit- and integration testing
- Learn how to write good and robust code
- Learn practices for how to write clean code

The project in this course

- **Problem:**
  - Looking up something in a huge file (~100 MB) is time consuming since we first need to load the file in memory
- **Solution:**
  - Create a server application which can load the file instead!
  - Looking up something from the client side can now be done by making a request to the server instead
- **Example use case at Volvo:**
  - Getting the interpretation of a signal by using a huge signal database file

How we will work in this course

- We will develop a server application for interpreting a signal using a signal database file
- Each participant will create a repository on GitHub where the code will be stored
- We will use Postman (or similar) as the client
- We will practice Test driven development throughout the course
- We will follow clean code principles

Focus will be on the practices so this particular project is really just a simple example

VELDI

Technical details

- **Prerequisites**
  - **Python 3.6 or newer installed**
  - **Git installed**
  - **Python IDE installed (e.g. PyCharm, Visual studio code)**

- **The server will be developed using *Flask* (a popular lightweight web application framework)**

- **Due to the time limitation, the signal database in this course will be extremely small and simple but the program can easily be scaled to a larger signal database file**

VELDI

Course outline

- Week 1
  - Virtual environment
  - Repository structure
  - Setting up a Flask server
  - How to use Postman
- Week 2
  - Test driven development
  - Unit testing
  - Mocking underlying functions
- Week 3
  - Style check
  - Lint check
  - Code coverage check
  - Achieving full code coverage
- Week 4
  - Integration testing
  - Error handling
  - Logging
- Week 5
  - Design patterns
  - Factory method
  - Clean code

VELDI

Schedule

- Five weeks starting today
- Live sessions via Zoom on Mondays 09:00-12:00
- Optional Q/A sessions on Wednesdays 09:00-11:00
- Optional Q/A sessions on Thursdays 09:00-11:00

VELDI

Course information

- The course will contain learning material, exercises and mandatory assignments
- The deadlines for each assignment are on Tuesdays at 22:00
- The lectures will be recorded and available afterwards
- Estimated own work time 10 hours/week incl. sessions

VELDI

Python 3A
Part 2
-
Virtual environment

VELDI

Virtual environment… what is that?

- Running code on different computers can be tedious since every computer has a different environment
- To make sure that the code will work on any computer, we create a *virtual environment*
- There are many different ways of creating a virtual environment (e.g. docker, virtualenv, pipenv, conda)
- In this course we will learn how to use *pipenv* but you can use whichever virtual environment you want

VELDI

Getting started with pipenv

- Open a terminal from your project location
- Install pipenv by running *pip install --user pipenv*

```
$ pip install --user pipenv
```

- Run *pipenv shell* to activate it

```
$ pipenv shell
```

```
$ pip --version
pip 21.2.4 from C:\Users\EmilW\.virtualenvs\EmilWall_Python2b-
93ZBJKxT\lib\site-packages\pip (python 3.9)
```

- When you're done you can exit the virtual environment

```
$ exit
```

VELDI

The Pipfile

- Running *pipenv shell* will create a file called *Pipfile*
- The Pipfile has a good overview of what packages your virtual environment is containing

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]

[requires]
python_version = "3"
```

VELDI

Installing packages

- **Installing a new package is easily done by running:**

  pipenv install PACKAGE_NAME

```
(.venv) C:\my-python-project>pipenv install flask
```

- **Packages that are only required for development (such as unit tests etc.) should be installed with the --dev flag, i.e. by running:**

  pipenv install PACKAGE_NAME --dev

```
(.venv) C:\my-python-project>pipenv install pytest --dev
```

- **You can either specify a specific version of a package or use the latest**

VELDI

Example of a Pipfile

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]
pytest = "*"          # use latest version of pytest
mock = ">=4.0.0"      # use v4.0.0 or later

[packages]
flask = "==1.1.2"     # use v1.1.2 only

[requires]
python_version = "3"
```

VELDI

Installing the packages

- You can also install packages the other way around, by first specifying the packages in the Pipfile and then run *pipenv install --dev*

```
C:\my-python-project>pipenv install --dev
```

- Installing a package will generate a *Pipfile.lock*-file which contains all dependencies and sub- dependencies
- Run *pipenv sync* for recreating an exact environment on a different computer
- Keep both the Pipfile and Pipfile.lock files version controlled in Git

Where will the virtual environment be?

- The virtual environments will be located in C:\Users\USER_NAME\.virtualenvs by default
- Sometimes it is more convenient to have the virtual environment located in your project folder instead
- To do that, set the environment variable *PIPENV_VENV_IN_PROJECT = True* before executing any pipenv-commands
- Do not forget to restart your applications after setting the environment variable

Python 3A
Part 3
-
Repository structure

VELDI

What a repository should contain

Mandatory files:

- Source folder (where your code is)
- Test folder (where your tests are)
- .gitignore (which files to be ignored by Git)
- README file (introduction and explanation of the project)
- LICENSE file (legal stuff)
- Pipfile or requirements.txt (which packages the project depends on)

Optional files:

- Configuration files (e.g. settings files etc)
- Data files (e.g. static files)
- Documentation files (more documentation apart from README)
- setup.py (making the project installable via pip)

VELDI

Folder structure example (main)

```
my-python-project/                ← (root folder)
    - cfg
    - data
    - docs
    - my_python_project/          ← (source folder)
        - main.py
    - tests/
        - integration/
        - unit/
    - .gitignore
    - LICENSE
    - Pipfile
    - Pipfile.lock
    - README.md
    - setup.py
```

VELDI

Useful links

- How to create a good README file
  - https://www.makeareadme.com/
- How to create a .gitignore file
  - https://riptutorial.com/git/example/885/ignoring-files-and-directories-with-a--gitignore-file
- How to create a LICENSE file:
  - https://choosealicense.com/
- How to organize files and use namespaces
  - https://dev.to/codemouse92/dead-simple-python-project-structure-and-imports-38c6
- How to create setup.py (out of scope for this course)
  - https://packaging.python.org/tutorials/packaging-projects/

VELDI

How to import modules correctly

- Importing modules in Python can cause a lot of headache
- The difference between a *script* and a *module* is that modules are Python files <u>intended to be imported</u> by other Python files
- To avoid problems, import classes and functions using the full module path
  - E.g. from source_folder.file_name import function

```
# Recommended way
from my_python_project.my_file import my_function, MyClass
```

```
# Not recommended
import my_function, MyClass
import my_file
```

VELDI

Making importing modules work

- If you intend to import something from another file of your project, you need to create an empty file called *__init__.py* in that folder
- The *__init__.py*-file converts the folder into a module
- Create an *__init__.py* in the source-folder and one in each underlying folder if you have any
- Later on, when you have test cases, you need add an *__init__.py* file in the tests-folder and its underlying folders as well

However, do not place an __init__.py file in the root-folder of your project!

VELDI

Example structure (source and tests)

```
my-python-project/                  ← (root folder)
    - my_python_project/            ← (source folder)
        - __init__.py
        - file1.py
        - main.py
    - tests/                        ← (tests folder)
        - integration/
            - __init__.py
            - test_integration.py
        - unit/
            - __init__.py
            - test_file1.py
            - test_main.py
        - __init__.py
```

VELDI

Running from command line

- When using the terminal of an IDE (e.g. PyCharm) the program will usually run fine with this command

```
$ py my_python_project\main.py
```

- However, sometimes this does not work in other terminals and then you have to run the following:

```
$ py -m my_python_project.main
```

- Try to run the program from the root-folder instead of the source-folder, like in the examples above

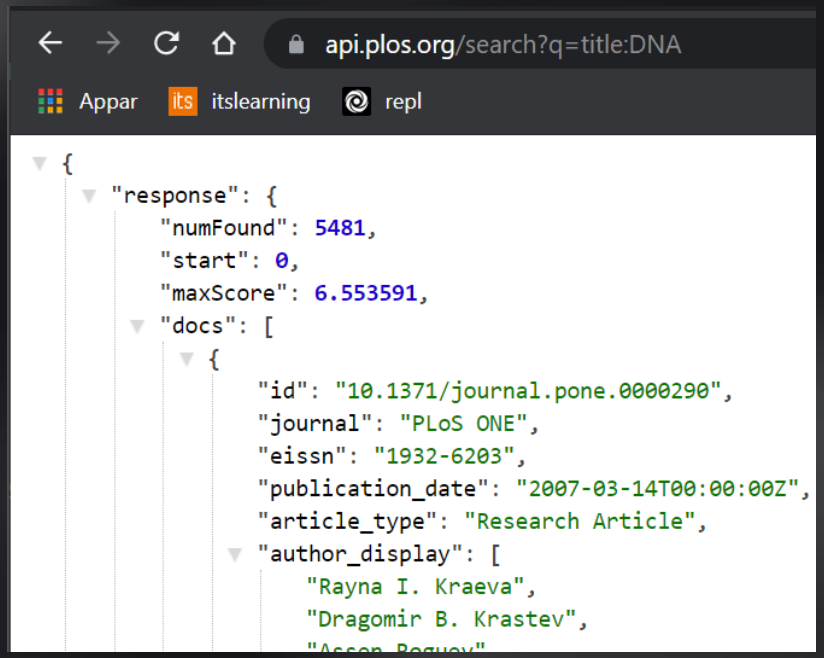
VELDI

Python 3A
Part 4
-
Setting up a Flask server

VELDI

Web API

- Web pages often fetch content from what is commonly referred to as web API:s
- A web API has a domain adress (URL like google.com)
- It also has *routes* which decide what content to fetch
- In Python you can create an API with *flask*



Running your first Flask application

- Start by running *pipenv install flask*
- Create a file in your source folder called *routes.py* where you write the following:

```
# routes.py
from flask import Flask

my_app = Flask(__name__)

@my_app.route("/", methods=["GET"])
def hello():
    return "Hello world!"

my_app.run()
```

- Open http://127.0.0.1:5000/ in your browser to see the result!

VELDI

Some words about RESTful APIs

- REST is a very popular software architecture used in web services
- Here are some of the most common HTTP methods:

GET - get something from the server
POST - send data to the server and let the server process it
PUT - modify something in the server
DELETE - delete something from the server

VELDI

Python 3A
Part 5
-
How to use Postman

VELDI

How to use Postman

- Go to https://www.postman.com/downloads/ and download the Postman app (sign up with Google if needed)

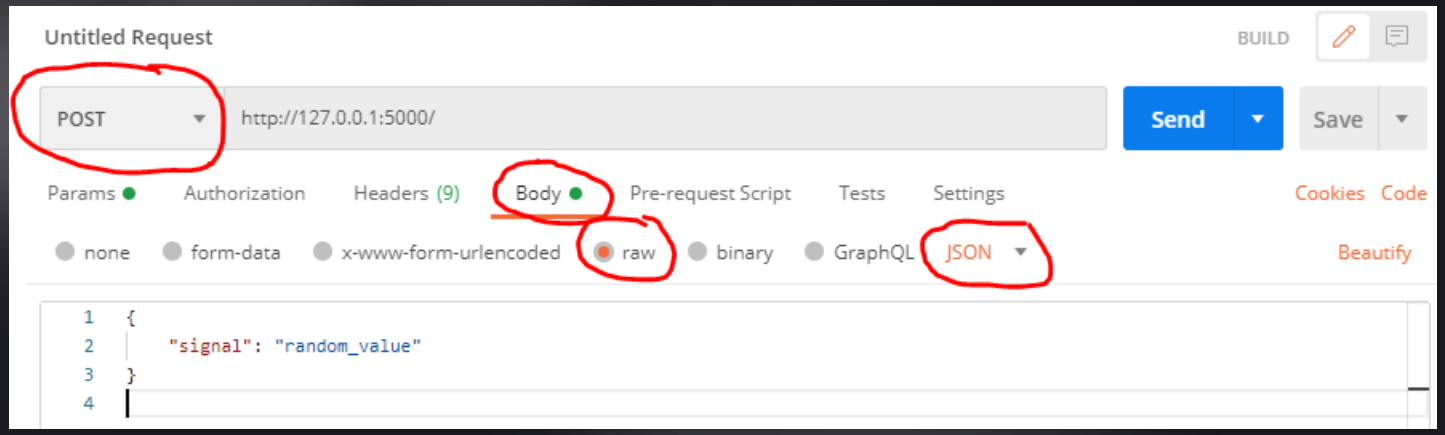
- Open Postman and run your first GET-request to http://127.0.0.1:5000/

- Verify that you got the response: *"Hello world!"*

VELDI

Sending a POST-request

- Select POST from the drop down list
- Go to the *Body*-tab and select *raw* and *JSON* in the drop down list

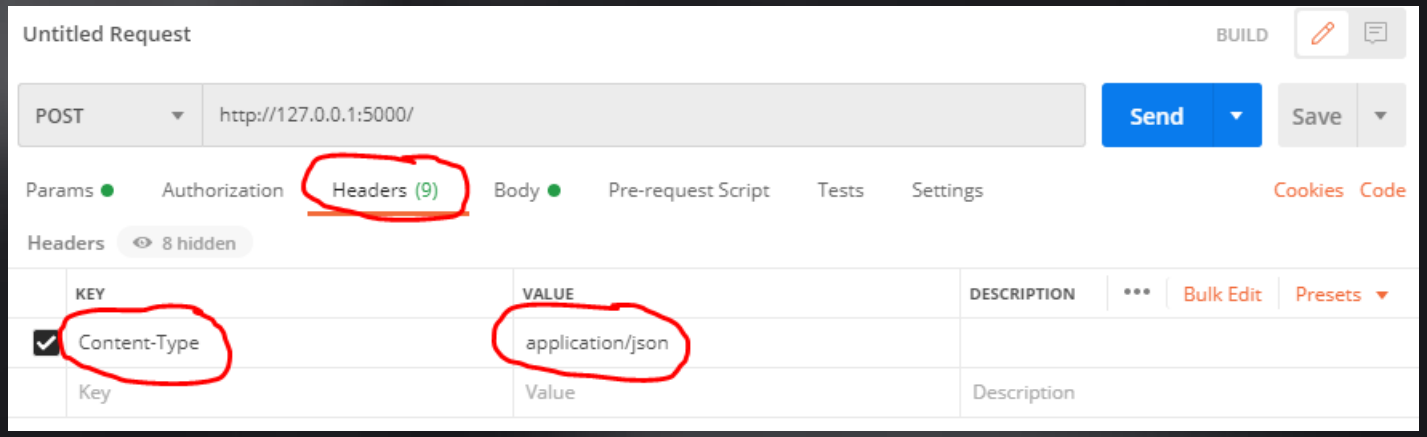


- Add the data (the payload) in JSON-format in the free text field

VELDI

Sending a POST-request

- In order for the server to interpret the request properly, it needs to know which format the request has
- This is done by adding *Content-Type* as key and *application/json* as value



- Now you can click the Send-button!

VELDI

Postman alternative

- **If you find Postman too complicated or run into problems, there is an alternative called *RestMan***

- **Download the chrome extension here: https://chrome.google.com/webstore/search/restman**

VELDI

veldikompetens.se

Thank you!

VELDI KOMPETENS