# AutPy Project

November 30, 2021

## 1 Project

### 1.1 Introduction

For the final block in the AutPy course, you will need to write a small command-line tool. In contrary to the previous labs, this will be something you need to do on your local machine, as the notebooks aren't a good fit to develop reusable "real-world" projects.

The main part of the project will explore topics you learned about in the labs: Flow control, data structures, creating a command-line application and using APIs. Additionally, we have various different **additional topics** which we didn't cover so far. You will need to pick and implement **two** of those.

### 1.2 Grading

- The deadline for the project is the same for all groups: **Thursday, December 23th, 23:59**.
- The project is intended as **individual work**, i.e. is **not a partner/group project**. We will run automated checks to detect plagiarism, and reserve the right to take appropriate measures according to the OST Studien- und Prüfungsreglement should we detect that multiple students submit the same code. Of course helping each other and e.g. sharing a snippet of some lines is fine, but copying entire functions or files is not.
- You will submit your project by uploading a zip file to Jupyter **and running a `!submit` command** at the bottom of this lab. See the instructions at the end for details.
- You can submit multiple times, as usual. No automated tests will run for the project.
- Each of the four previous blocks (with several labs each) makes up 1/6 of the final grade. This project **results in the remaining 2/6 of the final grade**.
- To ensure consistent grading, we are unfortunately unable to accomodate for custom project ideas, despite this being planned initially. Sorry!

We will grade various aspects of your work, such as functionality, adherence to best practices, possible bugs/issues, completeness, etc.

### 1.3 Setup

To get started, you will need to have two things set up on your machine, which we both explain in the following sections:

1. Python itself, at least version 3.6
2. A suitable development environment (editor/IDE)

### 1.3.1 Setting up Python

**Windows** For Windows, the recommended way to set up Python is to download Python from Python.org and run the installer. The 3.10.0 release is very new at the time of writing (the labs currently use 3.9), but it's still the recommended version to download, as it contains various improvements to Python's exception messages, which will make development easier.

Alternative ways to install Python (untested):

- Install it via the Windows Store (if you run `python` in a `cmd` / Powershell window, you should get a prompt)
- Use the Chocolatey terminal package manager
- Use WSL (Windows Subsystem for Linux) and install Python there

Check if Python was successfully installed: After Python is installed, open a `cmd` or Powershell window, and run `py` in it. With the alternative install methods, you might need to run `python` instead. You should get a prompt to enter Python code (starting with >>>). Make sure the displayed version is 3.6 or newer. Exit again with `exit()` and you are all set.

**macOS** If you already use Homebrew, use `brew install python`. If you don't, you're on your own: Either set up Homebrew, or use one of the untested alternative ways:

- Download the installer from Python.org
- Use pyenv

Check if Python was successfully installed: After the installation has finished, open a terminal and run `python3`. You should get a prompt to enter Python code (starting with >>>). Make sure the displayed version is 3.6 or newer. Exit again with `exit()` and you are all set.

**Linux** On Linux, chances are that you already have a suitable version of Python installed. Run `python3` in a terminal and make sure the displayed version is 3.6 or newer. Exit again with `exit()`. If Python isn't installed yet or is only available in an older version, consider the following:

- Check your distribution's packages for a newer version
- On Ubuntu, use the deadsnakes PPA
- Use pyenv

After installation finished, verify again that you are running 3.6 or newer.

### 1.3.2 Setting up a development environment

If you already have a favourite general-purpose text editor you use for programming, feel free to use that (e.g. Vim, Emacs, Sublime Text, Notepad++, Atom, etc.). If you do not have a favourite setup yet, we have these suggestions:

**VS Code** Visual Studio Code is an excellent choice for Python programming, but also great to use for other languages. After installing it, make sure to install the official Python extension. Then make sure you can create a Python file (can be as simple as `print("Hello World!")`) and run it.

**PyCharm** If you like JetBrains products such as IntelliJ or WebStorm, PyCharm is a great choice for Python development too. During your studies you can get a free educational license for the professional edition.

**Eclipse + PyDev** If you use Eclipse already, the PyDev extension might be worth a try. Note that we don't have any personal experience with that setup, however, and the above options might work better.

## 1.4 Project: "CLI mit Wuff und Wau"

The aim of the project is to write a command-line tool which does various operations based on the registered dogs in the city of Zurich. This information is made available as "Open Data" and is thus freely accessible.

Your CLI should have three major features, which are detailed in the sections further down below:

- Search for dogs using a given name
- Perform data analysis based on the entire data set
- Make up new dogs

You are free on how to structure these different functionalities. Some possible approaches:

- Use argparse sub-commands to have e.g. a `wuff.py find Luna`, `wuff.py stats` and `wuff.py create`.
- Write multiple scripts, e.g. a `wuff-find.py`, `wuff-stats.py` and `wuff-create.py`. Make sure you don't copy-paste code between them.
- Implement a single command-line interface which does different things based on how it is invoked, similarly to what we did with the drink generator CLI. For example, you could use `wuff.py` to show statistics, `wuff.py Luna` to search for a dog and `wuff.py --create` to create a new dog.

General requirements:

- Download the data from the opendata.swiss website using the `requests` library. You don't need to store the data in a file.
- Don't hardcode the URLs to the data, use the provided API to get the data for a given year (more details below). By default, use the data for the lastest available year.
- Handle errors in an appropriate way, e.g. by showing a useful error message to the user. Make sure you follow best practices while catching exceptions (also see The Most Diabolical Python Antipattern – Real Python). As a rule of thumb, consider invalid values or other possible issues for data coming either from the user, or from the internet.
- Make sure it's clear to a user how to use your tool without having to read the code. For example, make sure the scripts have sensible names and/or show an appropriate `--help` output (`argparse` or the other CLI toolkits will help you with that). If you choose to write multiple scripts, you can also provide a "readme" file that explains how to invoke them.
- If you choose to write multiple scripts, make sure you don't copy-paste code between them.

Additional hints:

- **Consider getting the basic functionality to work first** (e.g. having no error handling and hardcoding the 2021 URL), and then revisiting the requirements.

- `requests` will give you a string with the downloaded data - use `.splitlines()` to turn them into a list of strings, which you can then pass to the CSV reader.
- Consider using a `csv.DictReader` for easy access to the data. You should not specify `fieldnames`, as they are contained in the first row of the file.

### 1.4.1 Search for dogs

When your tool gets called with the name of a dog, it shows the birth year and sex (male/female) of the dog (or dogs, if there are multiple with the same name).

Sample output (your invocation and output may vary):

```
$ wuff find Luna
Luna 2014 w
Luna 2016 w
# ... more results omitted ...
```

Use the data for the last available publication year.

### 1.4.2 Perform data analysis

Your tool finds and prints various interesting aspects about the given data. Those are:

- Which is the longest dog name? (if there is a tie, it's okay to print only one, or all of them, whatever you prefer)
- Which is the shortest one? (ditto)
- Which are the top 10 most common names? (hint: a collections.Counter makes this easier. You don't need to consider similar names, i.e. "Luna" and "Luna-Verena (Mona Lisa)" are two different dogs.)
- How many dogs are male vs. female?

Your tool should print all this information in a single run. How the output looks exactly is up to you, as long as all this information is visible.

The data has been published every year from 2015 to 2021. The command should have a `--year` argument which allows to specify which year to get the data for. Use the "Metadatenzugriff: API (JSON)" button on the main website to get a machine-readable format of all the available download URLs. Write your code in a way that it will continue to work when new data is published in 2022, and always uses the newest year if no such argument is given.

**Hint: Consider getting the basic functionality to work first** (hard-coding the 2021 download URL), and then revisiting the `--year` flag.

Sample output (your invocation and output may vary):

```
$ wuff stats
Shortest Name: Bo
Longest Name:  Zar-Lorcan vom Franzosenkeller
# ... more output omitted ...
```

### 1.4.3 Make up new dogs

The third functionality in your tool is to make up a new dog from the existing data. To do so, it should collect the following attributes:

- A random dog name from the data
- A random birth year from the data
- Random sex (`m` or `f`)
- A random dog picture/video from https://random.dog/ (hint: use either the JSON API or plain text URL to find the download URL)

Finally, download the picture to a file follwing the pattern `dogname_birthyear.extension` (for example `Bello_2010.jpg`), based on the data collected above. Downloading binary files via `requests` is a bit harder than downloading text, check this StackOverflow answer for some guidance.

When all required data has been gathered, print the properties of the new dog (name, birth year, sex, filename).

Additionally, the command should have an `-o` / `--output-dir` argument which allows specifying a directory where the file should be put. By default, the file should end up in the current directory.

Sample output (your invocation and output may vary):

```
$ wuff create -o ~/Downloads
Here's your new dog!
Name: Noizy
Birh year: 2013
Sex: m
The image of the new dog can be found here: /home/alice/Downloads/Noizy_2013.png
```

## 1.5 Additional topics

As outlined above, you are required to implement **two** additional topics. They use third-party libraries or tools you haven't used so far.

**Note:** Depending on your setup and/or operating system, you will not be able to run Python tools (like `pip`, etc.) in your terminal. The easiest way around this is to prefix the command with `py -m` (Windows) or `python3 -m` (Linux/macOS), e.g. `py -m pip install ...`. In case you still run into trouble, please ask us for help!

You can select from the following topics:

### 1.5.1 Version control with git

Use the `git` project to track changes while you are working on your project. Git allows you to create a so-called "repository" from your project folder, and then track changes by creating a "commit" for every change. Later, it's easy to go back to older versions of your code or view the changes between different versions - no more `my_script_v3_final_REALLYFINAL.py`!

**Note:** Git is a very powerful but also complex tool. Mastering it will be very useful for your studies (and your job!), but for this project you will only need a very basic understanding. You will learn more advanced features of git in the "Software Engineering" classes.

Also note: Git can be used via the command-line, but in the beginning a graphical tool can help understanding git concepts. There are several free git GUIs available, one of those is for example SmartGit (free for non-commercial use).

Requirements:

- Create a git repository for your project and submit the `.git` folder as part of your submission. Note that the folder might be hidden, so take care to include it.
- You do not need to push your repository to GitHub/GitLab. If you do so, **make sure the repository is set to private**.
- You do not need to use branches.
- Make multiple commits while working on your project and write descriptive commit messages for your changes.

Resources:

- Real Python has a good first Introduction to Git and GitHub for Python Developers
- The GitHub Git Cheat Sheet provides a quick overview of the most important commands (but you can also use git integrations from your IDE, e.g. the VS Code Git integration)
- The official Git Book provides a more in-depth look at various topics. Note that you will only need the first and parts of the second chapter (until around 2.4) for this course.

### 1.5.2 Type annotations

You might already have noticed that in many other programming languages you need to specify the types of arguments, return values or variables (so called "static typing"), while in Python, you don't need to do so ("dynamic typing").

However, Python still provides syntax for optional type annotations (or "type hints"). The language itself ignores those annotations, but separate tools can read them and warn the developer if the code doesn't conform to the annotations.

Tools like VS Code or PyCharm also benefit from those annotations, as they will be able to provide better auto-completion and other guidance based on them. The by far most common tool used to verify the annotations is mypy. For this topic, you set up mypy and annotate your code with type annotations.

Requirements:

- Run mypy on your code and make sure that type checking succeeds.
- Make sure all your functions are fully type annotated. You can annotate types with `Any` where really needed, but don't leave them unannotated. Hint: mypy provides options to turn unannotated or partially annotated functions into errors.
- You can disable warnings from mypy with appropriate comment directives, if needed, but make sure you explain why you did so.

Resources:

- mypy documentation
- Python reference for the `typing` module
- Python Type Checking (Guide) – Real Python

- PEP 484 – Type Hints, the PEP (Python Enhancement Proposal) originally introducing this concept.

### 1.5.3 Click or Typer instead of argparse

Instead of using `argparse` to declare the command-line interface of your application *imperatively* (i.e. telling the `argparse` parser which arguments to add), there are libraries which take a more *declarative* approach (i.e. you "describing" the end result and the library doing the rest).

The Click library is a very popular package for writing command line tools: You write a Python function and then add some additional code on top (a so-called "decorator", some new syntax you didn't use so far) to turn it into a command-line.

If you chose to add type annotations above, you could also use Typer instead, which follows a similar philosophy, but is based on type annotations rather than decorators.

Requirements:

- Use either Click or Typer rather than argparse

Resources:

- Click documentation
- Typer documentation
- Primer on Python Decorators – Real Python (also tells you how to write your own custom decorators, which is not something you need to understand in detail)

### 1.5.4 Using black and flake8 for code formatting and linting

Developers love discussing the correct way to format code. Or like Nadia Eghbal says in her excellent book "Working in Public: The Making and Maintenance of Open Source Software": "if there's one thing I've learned, it's that developers have *opinions*".

Python has an (almost) universally accepted style guide, called PEP 8. There is even a song about it! However, arguably formatting code consistently is something machines are better at than humans. That's how black was born: An auto-formatter for Python code, automatically reformatting your code according to fixed rules.

For this topic, additionally to black, you will also run flake8 on your code. Flake8 is a so-called "linter", a tool which is designed to uncover additional issues in your code - both style as well as functional issues.

Finally, if you chose to create a git repository for your project, also consider configuring pre-commit, which lets you run tools like `black` and `flake8` over a git repository easily. This step is optional, however.

Requirements:

- Set up `black` and `flake8` for your project.
- Make sure that `flake8` is configured so that it's compatible with black (hint: Check black's documentation).
- Ensure that the code you submit is formatted by black and does not raise any warnings from flake8.

- You can disable warnings from flake8 with appropriate comment directives, if needed, but make sure you explain why you did so.

Resources:

- flake8 documentation
- black documentation
- pre-commit documentation

### 1.5.5 Using rich for beautiful output

The rich library provides beautiful formatting for terminal output. It provides you with a Python API to easily output things like emoji, tables, etc. to the terminal.

Requirements:

- Use `rich` to beautify your output.
- Use colors and emojis (e.g. dogs?).
- For the "list dogs by name" functionality, output the data as a table.

Resources:

- rich documentation
- rich README in Swiss German ;-)

### 1.5.6 Using Pillow to edit images

The Pillow library (originally PIL, "Python Imaging library") allows you to create and edit image files via a relatively simple Python API.

Requirements:

- The download URL that delivers dog media not only delivers pictures, but also videos. Adjust your code that makes up dogs so that it only considers still images (`.jpg`, `.JPG`, `.jpeg` or `.JPEG` extensions). If you get a video (`.gif`, `.mp4`, etc.), ask for another download URL.
- Use Pillow to stitch two dog images together side-by-side. Fill up the remaining space with a white background. Save the resulting file (now containing two dog photos) with the name as per the project description.

Resources:

- Pillow documentation

### 1.5.7 Poetry to create an installable project

We have already used `pip install` to install various libraries. For this topic, your goal is to make your own project installable via pip.

Traditionally, this is done by writing a setup.py file. While it's good to know about how those work, we are going to use a more modern alternative, called Poetry.

Requirements:

- Use `poetry new` to create your project structure

- Use `poetry add` to add additonal dependencies required by your project
- Make sure you add `pyproject.toml` and `poetry.lock` to your submission

Resources:

- Poetry documentation

### 1.5.8 Writing tests with pytest

**Note: This is a more advanced topic which requires a solid understanding of the topics we have covered.**

With pytest, you can run automated tests against your code, to ensure it works correctly. For this topic, you are expected to write a couple of tests to ensure that your tool works properly.

Requirements:

- Make sure that those parts of your code which interact with the network are properly isolated from the parts which process the data, and that those parts are again separated from the output.
- Write unit tests for the code that analyzes the data, where you feed the code with dummy data and check the return values.
- Write unit tests for the code that downloads the dog pictures. Make sure they end up in a temporary directory via pytest's `tmp_path` fixture.
- Optional: Use the responses library to "fake" network answers, so that you can also write tests for the parts accessing an API, without needing internet access.

Resources:

- pytest: helps you write better programs — pytest documentation
- Temporary directories and files — pytest documentation
- pytest Tech-Webinar, 18.08.2020

## 2 Feedback form

We'd like to get some feedback for this lab! To give us feedback, double-click the cells below and edit it in the appropriate places:

- Replace `[ ]` by `[x]` to cross checkboxes, they should look like this once you finish editing:
  - ☐ uncrossed
  - ☒ crossed
- Add additional text where indicated (optional)

**Difficulty:**

The difficulty of the materials for the project was:

- ☐ Much too difficult
- ☐ A little too difficult
- ☐ Just right
- ☐ A little too easy
- ☐ Much too easy

**Time:**

The project spans across two blocks, so we expect you to spend a total of about 16h on it. Do you think you spent:

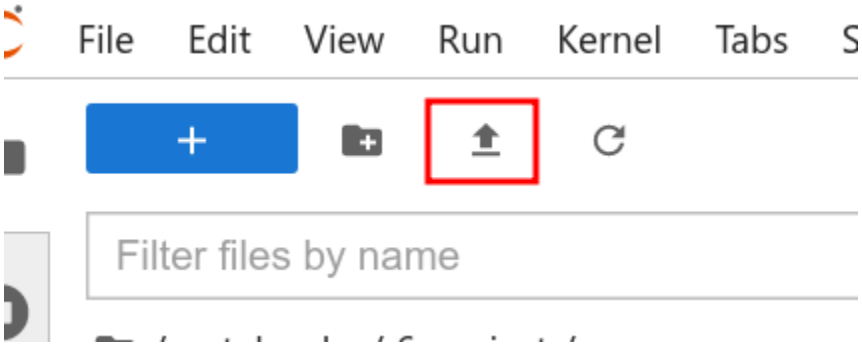
- ☐ Much more time
- ☐ A little more time
- ☐ About the scheduled amount of time
- ☐ A little less time
- ☐ Much less time

**Any topics you found especially enjoyable or difficult in the project?**

**Anything else you'd like to tell us?**

## 3 Submit

First, create a .zip file from your project folder, then **upload the zip via the upload button**:




Then, run the cell below to submit your work. Adjust the filename as needed. You can submit as often as you like.

In case of technical problems: - *Don't panic* - If you're in a course, show the error to your instructor. If not: * Take a screenshot of the error * Send an email with it and the notebook attached to your instructor (`florian.bruhin@ost.ch`, `meline.sieber@ost.ch` or `urs.baumann@ost.ch`) * Please add `florian.bruhin@ost.ch` and `urs.baumann@ost.ch` to the Cc

[ ]:
```
!submit --project project.zip
```