Lab #9: Project sandboxing: virtual environment

Introduction to Linux (NSWI177)

Table of contents

This lab is devoted to the basic principles of reproducible and isolated development. You will see how you can ensure that working on your project – that may require installation of many dependencies – can be setup without modifying anything system-wide on your machine.

Setup

Examples for this lab are available in repository **teaching/nswi177/2021-summer/common/examples**. Please, clone this repository first (forking it will not be needed).

Motivation

During last lab, we showed that the preferred way of installing applications (and libraries and data files) on Linux is via package manager. It installs the application for all users, it allows system-wide upgrades and it generally keeps your system in much cleaner state.

However, system-wide installation may not be always suitable. One typical example are project-specific dependencies. These you do not want to install system wide, mainly for the following reason:

- You do not want to remember to uninstall them when you stop working on the project.
- You want to control when you upgrade them (i.e., system-upgrade shall not affect your project).
- Package manager uses different version.
- Or they may not be packaged at all.

For the above reasons, it is much better to create a project-specific installation that is better *isolated* from the system. Note that installing the dependency per-user (i.e., somewhere into `$HOME`) may not provide the isolation you wish to achieve.

Such approach is supported by most reasonable programming languages and can be usually found under names such as *virtual environment*, *local repository*, *sandbox* or similar (note that the concepts do not map 1:1 across languages and tools but the general idea remains the same).

With such virtual environment, your dependencies are usually installed into a specific directory inside your project (i.e. a subdirectory in your clone that you do not version, though) and you only point your compiler/interpreter to this location.

The directory-local installation then keeps your system clean, allows working on multiple projects with incompatible dependencies because they are completely isolated.

While you rarely commit this installation-directory to your Git repository, you instead commit a configuration file that specifies how to prepare it. Each developer can then recreate it without polluting the main repository with distribution-specific or even OS-dependent files. Yet such configuration file ensures that all developers will be working within the same environment (i.e., same versions of all the dependencies).

It also means that new members of software teams do not have to browse knowledge bases or similar resources to setup their environment. They recreate it easily with the provided configuration file.

Typical workflow

Steps needed to initialize such projects are usually following (note that these are described in language-agnostic matter and specific tools may use different naming or may merge some of the steps automatically).

1. Clone the project (e.g., from a Git repository).
2. Initialize the virtual environment.
3. Activate the virtual environment.
4. Install the dependencies into the virtual environment.
5. Develop (compile, test & run) the project inside the virtual environment.

Developer may decide to remove the whole virtual environment if he needs to start from scratch.

Note that activation of the virtual environment typically removes access to already-installed libraries. That is, inside the virtual environment, the developer starts with a fresh and clean environment with bare compiler. That is actually a very sane decision as it ensures that system-wide installation does not affect the project-specific environment.

In other words, it improves on reproducibility of the whole setup. It also means that the developer needs to specify *every* dependency into the configuration file even if the dependency can be considered as one of those that are *usually* present everywhere.

Dependency installation

Inside the virtual environment, the project usually do not use generic package managers (such as DNF) but instead depends on language-specific package managers.

These are usually cross-platform and use their own software repository. Such repository then hosts only libraries for that particular language. Again, there can be multiple such repositories and it is up to the developer how he configures the project.

Note that there is a certain dichotomy because language-specific package managers can also install the packages system wide, competing with distribution-specific package managers. It is up to the administrator to handle this reasonably. Ideal option is to use the language-specific ones inside the distribution-specific one to ensure that only single entity is responsible for the management of the whole system. Technically, it is a bit more complex than that but that is out of scope now.

In our scenario, the language-specific managers would install only into the virtual environment directory without ever touching the system itself.

Installing Python-specific packages

The rest of the text will focus mostly on Python tools supporting the above-mentioned principles. Similar tools are available for other languages, we believe that demonstrating them on Python is sufficient to understand the principles in practice.

Python has a repository called **Python Package Index (PyPI)** where anyone can publish their Python programs and/or libraries.

The repository can be used through a web browser but also through a command-line client, called `pip`.

`pip` behaves rather similar to DNF. You can use it to install, upgrade or uninstall Python modules.

When run with superuser privileges, it is able to install packages system-wide. Do not use it like that unless you know what you are doing and you understand the consequences.

To try it safely, we will first setup a virtual environment for it.

Typosquatting

In your distributions upstream package repository, all packages typically has to be reviewed by someone from the distribution security team. This is sadly not true for the PyPI or similar repositories. This said, you as a developer must be more cautious when installing from such sources.

The least what you can do is pay attention to spelling of package names to install. If you do a typo in the package name, you might become one of the victims of the so called Typosquatting attacks.

Since installation of the package involves execution of arbitrary scripts from the package developer, your computer might be exploited very easy and very fast.

You might read more for example **in this blogpost** but searching the web will yield more results.

Virtual environment for Python (a.k.a. `virtualenv` or `venv`)

Python has built-in support for creating a virtual environment.

We will demonstrate this on example in `09/simple/` from **this repository**.

Switch to the above mentioned directory and investigate the contents of `timestamp2iso.py`. Note that `dateparser.parse()` is able to parse various time specification into the native Python date format. The time specification can be even text such as `three days ago`.

Make sure you understand the whole program before continuing.

Try running the `timestamp2iso.py` program.

Unless you have already installed the `python3-dateparser` package system-wide, it should fail with `ModuleNotFoundError: No module named 'dateparser'`. The chances are that you do not have that module installed.

If you have installed the `python3-dateparser`, uninstall it now and try again (just for this demo). But double-check that you would not remove some other program that may require it.

We could now install the `python3-dateparser` with DNF but we already described why that is a bad idea. We could also install it with `pip` globally but that is not the best course of action either.

Instead, we will create a new virtual environment for it.

```
python -m venv my-venv
```

The above command creates a new directory `my-venv` that contains a bare installation of Python. Feel free to investigate the contents of this directory.

We now need to activate the environment.

```
source my-venv/bin/activate
```

Your prompt should have changed: it is prefixed by `(my-venv)` now.

Running `timestamp2iso.py` will still terminate with `ModuleNotFoundError`.

We will now install the dependency:

```
pip install dateparser
```

This will take some time as Python will also download dependencies of this library (and their dependencies etc.). Once the installation finishes, run `timestamp2iso.py` again.

This time, it should work.

```
./timestamp2iso.py three days ago
```

Once we are finished with the development, we can deactivate the environment by calling `deactivate` (this time, without sourcing anything).

Running `timestamp2iso.py` outside the environment shall again terminate with `ModuleNotFoundError`.

`requirements.txt`

The required dependencies are usually stored in a file called `requirements.txt`. It contains simply the list of all dependencies, as passed to the `pip install` command.

The following command then installs the dependencies from the given file.

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

Optionally, the dependencies can be stored together with version specification. Without version, the latest one is assumed. With ==, a specific version can be installed. >= is used to specify minimal required version.

```
pkgname        # latest version
pkgname==4.2   # specific version
pkgname>=4.1   # minimal version
```

This assumes that the packages are using **semantic versioning**.

Note that it is possible to use `pip freeze` to list currently installed dependencies, thus creating a `requirements.txt` that sets specific versions, ensuring the same environment for every developer.

Exercise

Switch to `09/rest` and create a virtual environment there. Look at the code and install the required dependency. Try running the program.

Create corresponding `requirements.txt` too.

Solution.

`pip` or `python -m pip`?

Note that some tutorials will use the following pairs of commands interchangeably. Although there are subtle technical differences in the invocation, we will treat them as being completely equivalent.

Creating virtual environment is possible with either of these commands.

```
python -m venv PARAMS
virtualenv PARAMS
```

`pip` can be invoked in the following two ways.

```
pip PARAMS
python -m pip PARAMS
```

How it works?

Python virtual environment uses two tricks in its implementation.

First of all, the `activate` script extends `$PATH` to also contain path to the `my-venv/bin` directory. That means that calling `python` will prefer the application from the virtualenv's directory (e.g. `my-venv/bin/python`).

Try this yourself: print `$PATH` before and after you activate a virtualenv.

This also explains why we should always specify `/usr/bin/env python` in the shebang instead of `/usr/bin/python`.

You can also open the `activate` script and see how this is implemented. Note that `deactivate` is actually a function.

Why is the `activate` script not executable? **Hint.**

The second trick is that Python searches for modules (i.e., for files implementing an import module) relative from the path to the `python` binary. Hence, when the `python` is inside `my-venv/bin`, Python will look for the modules into `my-venv/lib`. And that is the location where your locally installed files will be placed.

You can check this by executing the following one-liner that prints Python search directories (again, before and after activation).

```
python -c 'import sys; print(sys.path)'
```

Note that Python 3.3 added an extra configuration option via `pyvenv.cfg` that allows the virtual environment to actually copy very few files while still ensuring an isolated environment.

Note that the principle above is actually very simple and requires a minimal support from the interpreter itself. And it also demonstrates how `$0` (or `argv[0]`) can be practically used.

Furthermore, the interpreter does not need to be configured to know whether it is running in the virtual environment or not – there is simply no difference from the implementation point of view. Something to keep in mind when designing anything similar.

A bit more verbose explanation with more examples can be found for example **here**.

Setup tools

This section shows how a Python program shall be prepared for further distribution. And how virtualenv helps us in testing it.

Please, open the directory `09/install` and setup a virtual environment there.

For proper (re)distribution, we have moved the source code into a separate module. Investigate the `matfyz/` subdirectories and refresh why `__init__.py` is needed (even when it is empty).

To launch the program now, we need to execute it in its modular form, that is via the following call.

```
python -m matfyz.nswi177.timestamp2iso 1 day ago
```

Feel free to create a shell wrapper for this program to simplify the launch.

Open `setup.py` and try to explain what it contains.

Now run the following commands and after they finish, look into <virtual-env>/bin.

```
./setup.py build && ./setup.py install
```

You should have noticed that the `bin` subdirectory now contains `timestamp2iso`. This is automatically created wrapper for your program. Notice that it does not have any filename extension and (when installed system-wide) looks like a normal program.

Exercise

Return to the `rest` example and add `setup.py` there.

Note that it is really not needed to remember how exactly `setup.py` looks. Only know where you can find a suitable template and update it to your needs.

Creating your own packages (e.g. for DNF)

While the work for creating `setup.py` may seem to complicate things a lot, it actually saves time in the long run.

Virtually any Python developer would be now able to install your program and have a clear starting point when investigating other details.

Note that if you have installed some program via DNF system-wide and that program was written in Python, somewhere inside it was `setup.py` that looked very similar to the one you have just seen. Only instead of installing the script into your virtual environment, it was installed globally.

There is really no other magic behind it.

Note that for example Ranger is written in Python and **this script** describes its installation (it is a script for creating packages for DNF). Note that the `%py3_install` is a macro that actually calls `setup.py install`.

Higher-level tools

We can think of the `pip` and `virtualenv` as low-level tools. However, there are also tools that combine both of them and bring more comfort for package management. In Python there are at least two favorite choices, namely **Poetry** and **Pipenv**.

Internally, these tools use `pip` and `virtualenv`, so you are still able to have independent working spaces, as well as, possibility to install the specific package from **Python Package Index (PyPI)**.

The complete introduction of these tools is out of the scope for this course. Generally, they use the same principles but they add some extra functions that are nice to have. Briefly, the major differences are:

- Abandon the `requirements.txt` in advantage of `pyproject.toml` and `poetry.lock` for `poetry` (alternatively, `Pipfile` and `Pipfile.lock` for `pipenv`).
- Eliminates the potential problems with forgotten `pip freeze` after updating dependencies.
- Enables to remove package together with its dependencies.
- Adds support for development-specific dependencies.
- Easier project initialization.

Note that other languages have their own tools:

- Ruby has **bundler**.
- Julia has **Pkg**.
- Rust has **cargo**.
- JavaScript has **npm**.
- ...

Graded tasks

`09/tapsum2json` (40 points)

Write a program that produces summary of TAP results in a JSON format.

TAP – or **Test Anything Protocol** – is a universal format for test results. It is used by BATS and the GitLab pipeline too.

```
1..4
ok 1 One
ok 2 Two
ok 3 Three
not ok 4 Four
#
# -- Report --
# filename:77:26: note: Something is wrong here.
# --
#
```

Your program will accept a list of arguments – filenames – and read them using appropriate consumer. Each of the files would be a standalone TAP result (i.e., what a BATS produces with `-t`). Nonexistent files will be skipped and recorded as having zero tests.

The program will then print summary of the tests in the following format.

```
{
  "summary": [
    {
      "filename": "filename1.tap",
      "total": 12,
      "passed": 8,
      "skipped": 3,
      "failed": 1
    },
    {
      ...
    }
  ]
}
```

We expect you will use a library for reading TAP files: **tap.py** is certainly a good option but feel free to find a better alternative.

Your solution must contain a `requirements.txt` with list of library dependencies that can be passed to `pip install`. Your solution must also be installable via `setup.py` and create a `tapsum2json` executable on the `$PATH`. This is mandatory as we will test your solution like this (see the tests for details).

Save your implementation into `09/tapsum2json` subdirectory.

Note that the automated tests require `json_reformat` utility from the `yajl` DNF package (`sudo dnf install yajl`).

The JSON reformatting is part of the tests only to allow easy visual comparison of the result. We do not require you to format JSON by yourself though passing `indent=True` to `json.dump` certainly helps debugging.

`09/templater` (40 points)

Create a simple templater.

Initial implementation is inside `09/templater`. It uses **Jinja** templates and your task is to add proper parsing of a YAML header and add a filter `arabic2roman` into Jinja for converting Arabic numerals to their Roman form (i.e. convert `52` to `LII`).

The YAML header in each input file contains a dictionary of variables that will be available in the template.

Examples are provided in the `09/templater` directory.

Your solution must contain a `requirements.txt` with list of library dependencies that can be passed to `pip install`. Your solution must also be installable via `setup.py` and create a `templater` executable on the `$PATH`. This is mandatory as we will test your solution like this (see the tests for details).

Save your implementation into `09/templater` subdirectory.

Note that we expect that you will refactor the initial code to use proper modules etc. etc. We provide it as an example on how to work with Jinja.

Note that the existing code has to be updated to properly parse the YAML front-matter and store only the *rest* of the file into `content`.

`09/kramdown` (20 points)

The example repository contains a simple script written in Ruby in `09/kramdown`. Don't panic. The script is extremely simple and your Python knowledge would be enough to finish this task.

Copy this directory to your submission repository and implement the missing conversion from Markdown to HTML using the **Kramdown renderer**. Full description follows.

Note that Ruby uses tool called `bundle` to install so-called Gems (libraries). The Gems are specified inside `Gemfile` (currently, it contains `slop` for parsing command-line arguments).

Following two commands prepare a local installation directory `vendor` and install the gems into it.

```
bundle config set --local path vendor
bundle install
```

To execute the utility `./kramdown_render.rb`, you need to run it via `bundle exec`.

```
bundle exec ./kramdown_render.rb example.md
```

Your task consists of the following two subtasks.

- Add Kramdown based conversion of the input file to HTML, including adding the appropriate gems to `Gemfile`. This is the major part of the task.
- Add support for `--stylesheet` parameter that inserts the following snippet into HTML header (only when `--standalone` is used), replacing `ARG` with actual argument.

```
<link rel="stylesheet" type="text/css" href="ARG" />
```

Deadline: May 17, AoE

Solutions submitted after the deadline will not be accepted.

Note that at the time of the deadline we will download the contents of your project and start the evaluation. Anything uploaded/modified later on will not be taken into account!

Note that we will be looking *only at your master branch* (unless *explicitly specified otherwise*), do not forget to merge from other branches if you are using them.

Changelog

2021-04-29: Clarification about JSON reformatting in `09/tapsum2json`.

2021-04-26: Clarification about the YAML front matter in `09/templater`, fixes in example JSON output in `09/tapsum2json`.

© D3S

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University

Malostranské náměstí 25
118 00 Praha 1
Czech Republic