A minimal introduction to Python for R users

Kristen Feher
27 November 2024

Introduction

While R is undoubtably the most popular programming language for bioinformaticians, I have decided to write this package in Python. The reason for this is that a few core elements of this pipeline are technically easier to programme in Python than R.

To make this package accessible for an R user with limited or no Python experience, I have designed this package with a very straightforward user interface such that it can be used as a computational workhorse. Furthermore, saving output in an R-friendly format is automated, such that plotting and other downstream bioinformatics tasks can be done in R.

To get started, I have created an extremely minimal guide to setting up and using Python, targeted to the R user. There are better Python guides out there, but my main aim is to highlight some major differences between R and Python that confused me when I started with Python.

```
>Code in this font in entered in a bash terminal.
```

Code in this font is python code in a .py file or .ipynb Jupyter notebook or R code.

Choice of IDE

Visual Studio Code has a very convenient interface for Python.

Begin by downloading and setting up, as explained here (accessed 27-11-2024): https://code.visualstudio.com/docs/introvideos/basics

Environment and installing packages

Next, make sure VS code is set up for Python (accessed 27-11-2024) : https://code.visualstudio.com/docs/python/python-tutorial

The above tutorial walks you through creating an environment, which is simply a hidden folder called .venv/
Packages are downloaded using '`pip install`' in a bash terminal while in your working directory and saved to .venv/lib/
This is in contrast to R, where packages are downloaded once to a central location (*.libPaths()*), and are visible everywhere.

Manual set up of environment

It is possible to open a bash terminal in VS Code. Alternatively, you may be using Python directly from a bash terminal without using VS Code (assuming you have installed Python on your computer). To manually create a new environment in bash (instead of following the tutorial mentioned in the previous section), navigate to your working directory and use the following:

```
>python -m venv .venv/
>source .venv/bin/activate
```

After this is completed, you can install packages (e.g. numpy) using

```
>pip install numpy
```

Importantly, DO NOT navigate into .venv/ to install packages, stay in the top of your working directory. '`pip`' knows where to install.

This is like using *install.packages("an_r_package")* in R, except it needs to be done in the bash terminal. In contrast, you install R packages from within the R console. Installing a package in Python should also install all its dependencies as well.

Installing Python wheels

A 'wheel' is a build of a package. If you have downloaded a wheel and have it on your local computer, you can install the wheel like this:

```
>pip install /path/to/wheel/fancy_wheel-0.1.0.whl
```

If you want to install an updated version:

```
>pip uninstall fancy_wheel
>pip install /path/to/wheel/fancy_wheel-1.1.0.whl
```

and then restart the kernel.

Managing packages using poetry

The classical way to install and manage packages is by using ‘pip’. In order to keep a record of the packages that are currently installed, use the following:

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

However, there are more modern and streamlined methods for managing dependencies. There are a few options out there, but I have enjoyed using ‘poetry’ : https://python-poetry.org/ (accessed 27-11-2024)

Poetry has been especially good for managing dependencies in developing a package, as well as building a ‘wheel’ (a package build).
To get started (https://python-poetry.org/docs/), first install ‘pipx’. Then in a bash terminal (not necessarily in a project directory as these will be visible everywhere):

```
>pipx install poetry
```

To use poetry in a project called ‘my_new_project’, first create a new top level working directory. Then in that working directory:

```
>poetry my_new_project
>cd my_new_project
>python -m venv .venv/
>source .venv/bin/activate
```

Now, instead of using pip to install a package such as ‘numpy’, you can use poetry:

```
>poetry add numpy
```

The advantage of using poetry is that is automatically creates a 'pyproject.toml' file which keeps track of all dependencies, and is necessary for building packages.

Built-in types

As you might expect, Python has built-in types:
https://docs.python.org/3/library/stdtypes.html

and built-in functions:
https://docs.python.org/3/library/functions.html

If you are already familiar with R, it should be self-explanatory.

Using Classes in Python

Classes are very central to the Python experience. Additionally, methods (functions) attached to classes are only visible to the class they belong to.

Once you have set up your python project, go ahead and open a new .py file to store your code. Alternatively, open a jupyter notebook for a more interactive experience (within VS Code you will need to install a jupyter add-on).

Two computational workhorse packages are *numpy* and *pandas*. *Numpy* provides infrastructure for computing over arrays, and *pandas* offers a DataFrame that is similar to an R data.frame.

To use these packages, begin your code with

import numpy as np
import pandas as pd

'*np*' and 'pd' are short for *numpy* and *pandas* respectively. You can use whatever alias you like, but these are the commonly used aliases. This step is like using *library(an_r_package)* in R.

The reason why you need an alias is the things contained in *numpy* and *pandas* are invisible to the local environment, unless you specify the full

path. Having an alias means you don't have to type the full package name every time. For example, to create an array or DataFrame, you must use:

```
new_array = np.array([1, 2])
new_df = pd.DataFrame([1, 2])
```

Let's say I have a package called *my_package* which contains a class called *my_class*. This class has an attribute called *attribute1*. To access *attribute1*, the code looks like this:

```
import my_package as mp
# create an instance of my_class. In this case, constructing the instance
# doesn't require any arguments, but you will often need to supply
# arguments inside the brackets
X = mp.my_class()
# don't use brackets to access attributes
X.attribute1
```

If *my_package* had a subpackage, inventively called *subpackage*, which contained *my_class*, the code would look like this:

```
import my_package as mp
X = mp.subpackage.my_class
X.attribute1
```

An alternative for easier typing:

```
import my_package.subpackage as sub
X = sub.my_class
X.attribute1
```

There is a second subpackage called *another_subpackage* with a class called *another_class*. *another_class* has a method (AKA function) called *another_method*, whereas *my_class* has a method called *my_method*. The methods are only visible to their respective classes and this is how you use them:

X1 = mp.subpackage.my_class()
 # Use brackets to apply the function. In this case, applying the function
 # doesn't require any arguments, but you will often need to supply
 # arguments inside the brackets.
X1.my_function()
X2 = mp.another_subpackage.another_class()
X2 .another_function()

The methods act on the classes to change the instance of the class or produce some output based on the instance of the class. The following would produce an error:

X1.another_function()
X2.my_function()

In summary, the dot "." in python is used to specify a path. When used with *import*, you can import a subpackage. When used to access an attribute of a class, it somewhat similar to using "$" or "@" in R. When used to access a method of a class, it is somewhat similar to using "%>%" in R .

Installing the umap package

A bioinformatician is almost certainly going to want to create a UMAP at some point in time. As of 27-11-2024, it can be a little tricky in python. If you are using my package, I hope it is seamless, but here is a short guide in case you need to do it manually (the situation may have changed by the time you read this). Firstly, there are two packages with 'umap' in their names, you need to use 'umap-learn'. The dependencies don't seem to be managed well and you have to add them manually:

```
>poetry add numpy scipy
>poetry add scikit-learn
>poetry add numpy^1.22
>poetry add numba
>poetry add umap-learn
```

In your python code, import it like this (the underscore isn't a typo):

import umap.umap_ as umap