Intro to Python — SMM692

Getting Started with Python

Simone Santoni

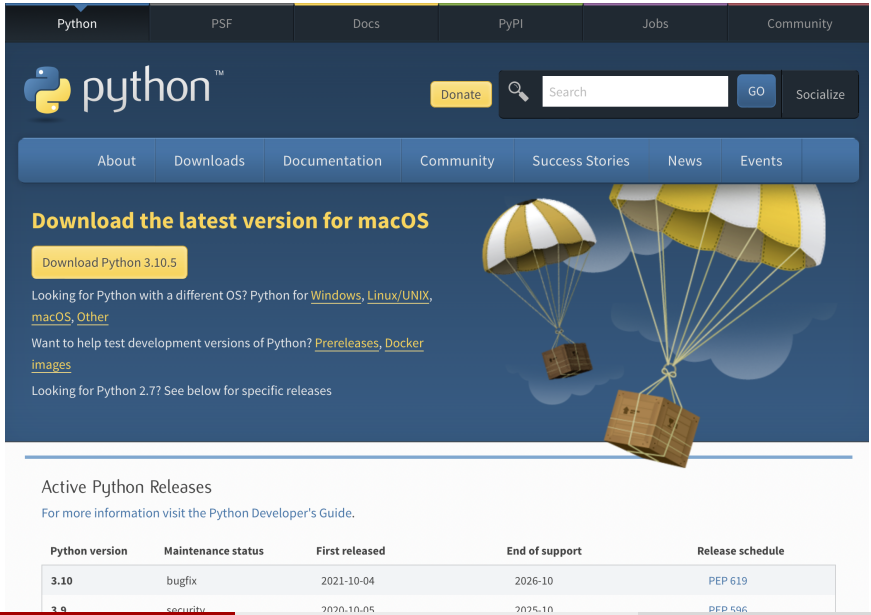
Bayes Business School

MSc Pre-Course Series

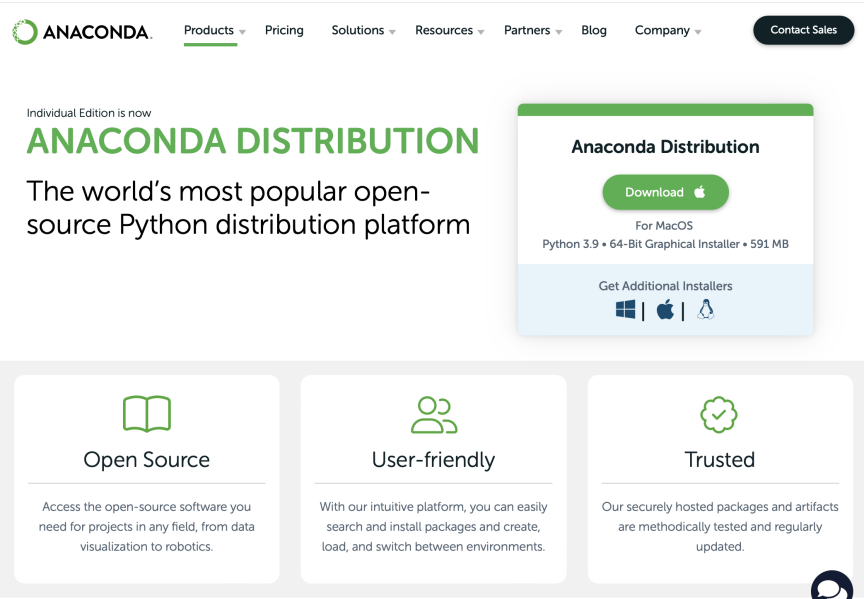
Outline

Outline

Option 1: Official Installer



Option 2: Anaconda Distro (Preffered Way)



Outline

Steps

1. Downaload the installer for your operating system (unless you have a very old machine running Win, go for the 64-Bit version)
2. Run the installer
   - For Linux: navigate to the folder where you have downloaded the installer as per step 1, open a shell session, then run `$ bash ./Anaconda3-XXXX.XX-Linux-x86-64.sh`
   - For Win and Mac OS: just run the graphical installer downloaded in step 1
3. Accept the terms proposed by the Anaconda people to use their software, comprising Python, the `conda` package manager, and a bundle of modules for data science
4. That's it!
   - For Linux users: if you accepted the default installation options, an environmental variable has been created either in your `.bashrc` or `.zshrc`. That means you can access the various pieces of software included in the Anaconda installation (e.g., Anaconda Navigator) from a shell session
   - For Win and Mac OS users: the various pieces of software included in the Anaconda installation are available from the menu of your system

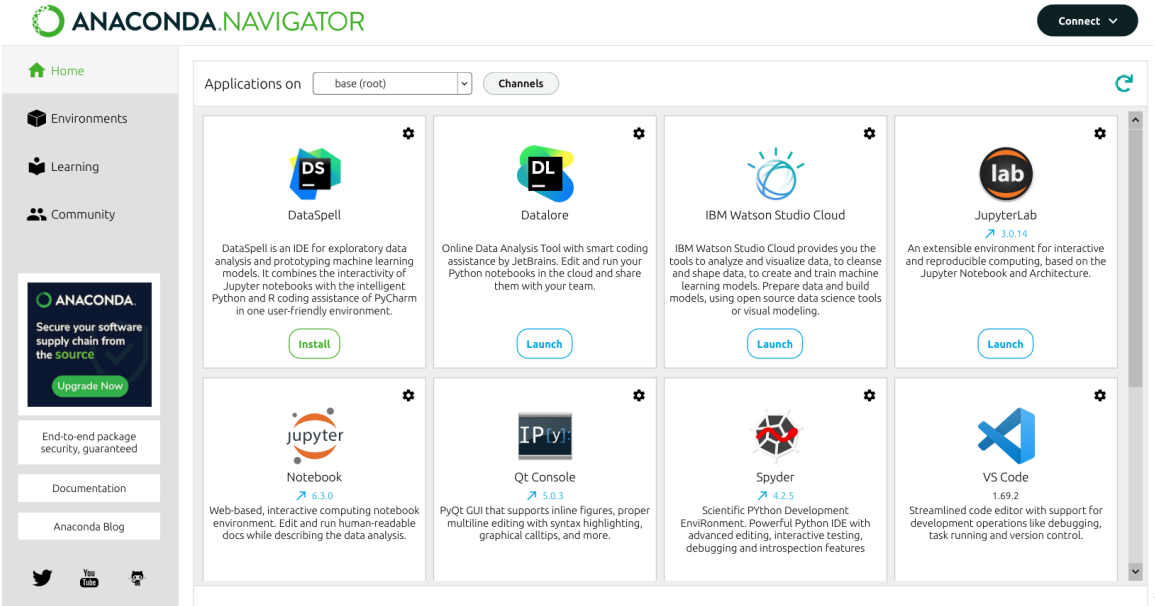
Outline

How is the Anaconda Distro Different?

- Anaconda is 'battery-included' — it comes with a humongous number of modules for data science
  - If you use the official Python installation, then you have to install the modules you need on your own!
- Anaconda is a bundle of various pieces of software:
  - `conda` is the Swiss army knife to manage Python modules and environments
  - `anaconda-navigator` is the graphical interface from within to access Python IDEs and related desktop/web applications

Outline

What Software Can I Access from Anaconda Navigator?



Python as a Language and Interpreter

- Typically, we refer to Python as a programming language
- However, Python is also a software package called an interpreter
  - An interpreter is a kind of program that executes other programs
  - When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains
  - In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine

Python Execution Model

- When you instruct Python to run your script (e.g., 'script.py'), there are a few steps the interpreter carries out before your code actually starts crunching away
- First, the code is compiled to something called 'byte code'
  - This step happens behind the scenes (there is nothing to do for the programmer!)
  - A file called 'script.pyc', automatically generated, contains the translation of your code into lower-level code instructions
- Then, the compiled code is routed to something called a 'Python Virtual Machine' (PVM)
  - This step is hidden to the programmer like the previous one
  - Mainly, it iterates through your byte code instructions, one by one, to carry out their operations

Step	Entity
Source	script.py
↓	
Byte code	script.pyc
↓	
Runtime	PVM

Outline

Script Preparation → Script Run

Step 1: script preparation

The below displayed Python code achieves two things:i) it prints the string object "Bazinga!", and ii) it prints the resul of the algebraic operation $4 + 2$. Note that all lines starting with # are not considered Python code — instead, they are comments that illustrate/explain the logic of the script.

```python
# Print a string object
print("Bazinga")
# Print the result of an algebraic operation
print(2 + 4)
```

Step 2: script run

Outline

Running a Python Shell in the Terminal

```
(base) ➜  ~ python
Python 3.9.7 (default, Sep 16 2021, 08:50:36)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Bazinga")
Bazinga
>>> print(2 + 4)
6
>>>
```

Running an IPython Shell in the Terminal

```
(base) ➜  ~ ipython
Python 3.9.7 (default, Sep 16 2021, 08:50:36)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.29.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print("Bazinga")
Bazinga

In [2]: print(2 + 4)
6

In [3]:
```

Running a Python IDE

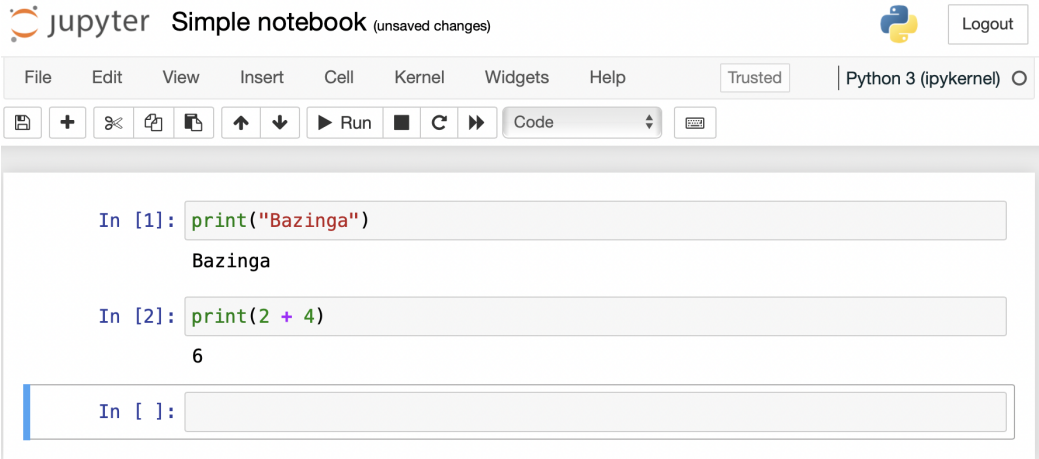
There are plenty of Python IDEs in the market, including:

- Colab (online)
- IDLE
- Datalore (online)
- Jupyter/Jupyterlab
- PyCharm
- Qt Console
- Spyder
- Thonny
- Wing

By installing a couple of plugins, the following (advanced) text editors can turn into Python IDEs:

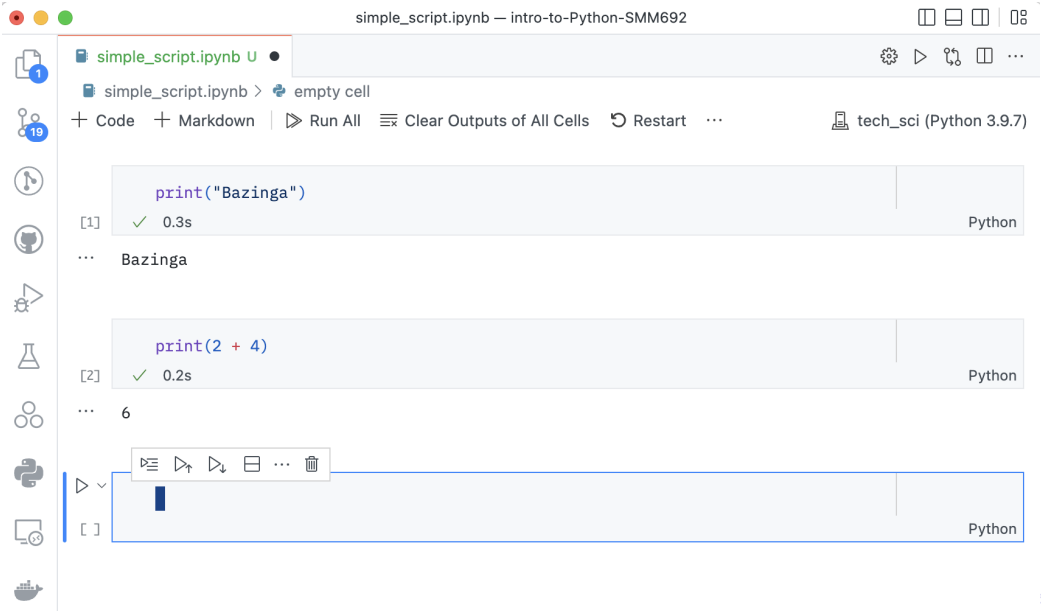
- Emacs
- Vim/Neovim
- VSCode

Interactive Python Coding with Jupyter



Interactive Python Coding with VSCode

Interactive Python Coding in Jupyter with VSCode



Outline

Why Do We Need a Python Environment?

1. We do not want to break the system-wise installation of Python!
2. We want to create collections of Python modules to deploy in specific projects/types of projects

Do Not Touch the System-Wise Installation of Python!

- The large majoity of operating systems come with Python installed by default
  - That installation is called 'system Python'
  - 'System Python' runs or manages a large number of essential operating systems processes
- Every time we install a new Python module A, the package manager (e.g, conda) checks the libraries on which A depends
  - If a dependency $D(A)$ is not installed, the package manager will install it for us
- A single module can depend on many libraries
  - The web of dependencies linking Python modules can get complicated
  - If inconsistencies arise within the web of dependencies, we may not be able to install the modules we need
  - ... even worst, some modules previosuly installed may stop to work!
- Now, you do not want to break some essential features of you operating system
- The implication is that we should be bette off not using 'syteme Python' for ou data science work

Create PyEnvs, Ad Hoc Collections of Python Modules!

- Mainly, a Python environment is a Python installation — along with a set of modules — that has nothing to do with 'system Python'
  - In other words, Python environments are not used by default for any processes regardin the operating systems
- Typically, we create a Python environment to carry out a specific project or a class of projects (e.g., Machine Learning)
- The advantage is twofold:
  - Reliability — the web of dependencies is relatively simple insofar as we install only the few modules that are required by the project or project class. Likely as not, we will not come across installation issues!
  - Reproducibility — the environment explicates the modules are necessary to carry out a certain project. Hence, a user who wants to reproduce our project's results knows which modules to install

Outline

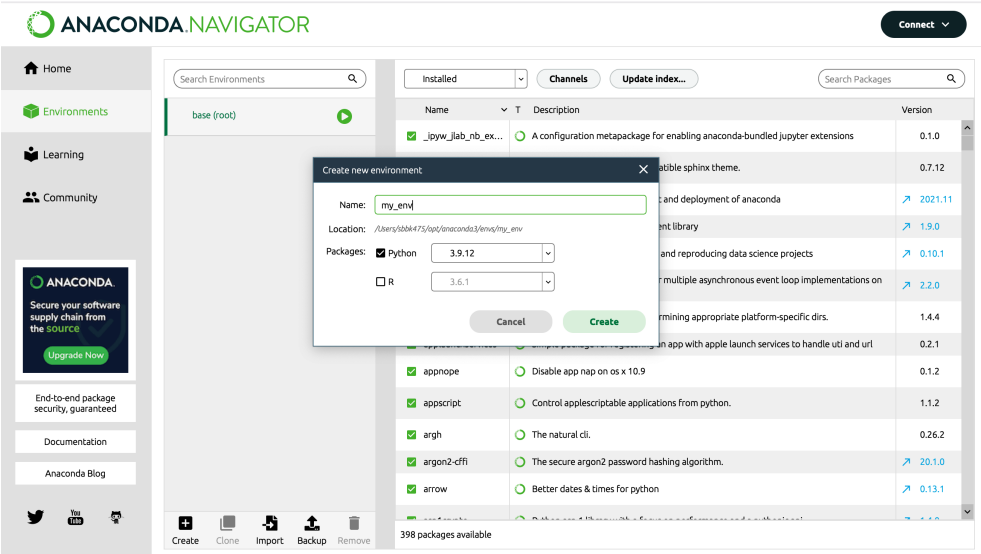
Creating PyEnvs from the Command Line (1 out of 2)

```
(base) ➜  ~ conda create -n my_env
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /Users/sbbk475/opt/anaconda3/envs/my_env



Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate my_env
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) ➜  ~
```

Creating PyEnvs from the Command Line (2 out of 2)

```
(base) ➜  ~ conda activate my_env
(my_env) ➜  ~ conda install ipython jupyter numpy matplotlib
```

Creating PyEnvs from within Anaconda Navigator (1 out of 2)



Creating PyEnvs from within Anaconda Navigator (2 out of 2)