Distribution vs. Package Manager vs. Environment Manager vs. Development Environment

The Python ecosystem is fairly large. In it you'll find many development tools and it can sometimes be difficult to know what the difference is. For example, what's the difference between Conda and Anaconda (distributions)? How about pip and Conda (package managers)? This Python Pointer was designed to help bring some clarity to the litany of terms used to refer to the various parts of a "Python environment". Below we'll give you a brief overview, along with some optional, supplemental reading, to help you understand the difference between a Python distribution, a Python package installer, and a Python virtual environment. Though they're not officially supported for this course, you're welcome to experiment with any of the tools below (or other tools of interest not listed here) to complete this course.

What is Python distribution?

A "distribution" (also referred to as a "package") is a collection of files required to install an application. For example, our Python install instructions mention you can download the official Python distribution here for local installation. The installer you download and run installs the Python distribution provided by python.org and contains a number of different libraries that contain code to perform specific tasks: math, file I/O, logging, etc. This is not the only popular Python distribution available however, so I'd like to start by introducing you to a few popular Python distributions (each of which contain the Python interpreter so, once installed, you'll be able to execute Python code):

Python Distributions

Anaconda: a Python distribution that contains most of the core Python libraries but also includes many data science-specific tools including: Plotly and Bokeh (visualization), TensorFlow & PyTorch (deep learning), and NumPy & DataFrames (data analysis, both of which we'll be looking at this semester).

ActivePython: a customizable Python distribution that allows you to select your OS, and any packages you'd like to include, then sorts out and installs all of the necessary dependencies to get your Python runtime environment functioning. The value that ActivePython provides is a consisten, reproducible environment that can be customized to run on various different platforms.

Miniconda: a very lightweight Python distribution that contains the Python interpreter (just like you will have/could have downloaded from python.org along with a package manager named "Conda" that allows you to install Conda-formatted packages. We'll discuss package managers in the next section.

It's worth noting that there's absolutely nothing keeping you from creating your own Python distribution. If you wanted to create a distribution that, for example, contains tools for image processing along with a few specific packages for performing complex mathematical analysis, you can certainly do that.

Python Package Managers

Package managers are used to add/remove packages to/from your Python distribution. For example, if you want to install a Python library for doing web development (Flask, for example), you'll need a way to install the Flask package so that your Python distribution is aware of it and knows where to find the files to make them available for use within your program. Here are some of the most popular package managers for Python:

pip ("Package installer for Python"): pip is the most popular package manager and is very easy-to-use (we'll have a chance to use it for installing some image processing and data science libraries in the second half of the course). It is also the official package manager adopted by the Python community. pip is only capable of installing Python packages and does not support installing packages in other libraries Like C or Java. As of the end of 2020, pip has built-in support for resolving package dependencies. So if you want to install package A, but package A requires package B, pip will automatically download and install package B before installing package A. Previous versions of pip would tell you a dependency needed to be resolved, but you as the developer were responsible for manually installing the dependency yourself.

Conda: not to be confused with Anaconda, Conda is a multi-language package manager meaning it can be used to install not only Python packages, but packages written for a large number of other languages as well. It is also included in the Anaconda and Miniconda distributions mentioned above. Like pip, it can be used to install a variety of packages as well as resolve dependencies at install-time, but it has a few advantages. Perhaps the biggest advantage of Conda over pip is that Conda also serves as an "environment manager" which allows you to quickly switch between different Python environments on the same machine. We'll look at environment management in the next section.

Environment Managers

Imagine a scenario where you want to be able to test your code on different versions of Python; let's say 3.6 and 3.9, for example. It's easy enough to install both distributions, but how, then, do you go about running your code on a specific version? Furthermore, what if you only want to use a set of database libraries for one program in one version of Python, but you want to test another application that uses natural language processing libraries for a different version of Python, both on the same machine? Manually, this is very tedious and arduous to do, so the Python community created "virtual environments". Please note that these virtual environments are NOT the same thing as virtual machines, but there is definitely some conceptual overlap between the two. A virtual environment in Python is an isolated environment that only references a specific version of the Python interpreter as well as a specific set of libraries. These virtual environments allow you to create custom, segregated environments that contain specific settings that do not impact your system's Python environment overall. Using these virtual environments, it becomes easy to install Python 3.6 and some database libraries into one isolated environment while installing Python 3.9 and a few natural language processing into a separate one. It then becomes easy to switch back-and-forth between these environments and allows you to keep your custom environments separate, but on the same machine. Here are the most popular environment managers within the Python ecosystem:

1. venv: the default environment manager that comes with the official Python distribution downloaded from python.org. It is a command-line tool that allows you to pretty easily create, customize, manage, and switch between virtual environments.
2. virtualenv: similar to venv, but with more functionality. In practice I tend to see virtualenv used more than venv, but both are pretty common. Because virtualenv is not the default environment manager in the official Python distribution, you must install it. How you as? Well, since virtualenv is a Python package, it can simply be installed using the pip package manager we mentioned earlier. Something like: **`pip install virtualenv`** should do the trick!
3. pipenv: is built using venv, but adds built-in functionality for being able to use pip to install packages within a virtual environment. Previously, a developer would have to install a virtual machine then separately use pip to install packages inside of it. With pipenv, pip is integrated into your virtual environment automatically which simplifies the management of packages within your virtual environments.

Development Environments

If you've already setup your development environment for the course, you're probably using VS Code, PyCharm, the course VM, or REPL. If not, you may want to take a look at the "Installing Python" section before reading this section. In addition to the officially supported options mentioned above, I wanted to introduce you to a few popular "integrated development environments" (IDE's), applications where you can write, test, execute, and debug your Python code:

1. Atom: an IDE that provides ease-of-use for beginners as well as advanced functionality for more experienced developers. Atom also has a fair number of downloadable themes that look pretty sharp in my opinion too. You can even customize your IDE's style sheet using CSS yourself!
2. Jupyter: provides "Notebooks" that have become extremely popular for a wide variety of use cases, especially data science. Jupyter Notebooks are accessed through a web applications but there are a number of free, hosted solutions such as Google Colab and MyBinder (you can try both for free using the provided links). Jupyter Notebooks allow you to mix code with rich content (text, images, etc.) and are a great way to annotate and share your applications. We'll introduce Jupyter Notebooks when we begin our discussion on Data Science later in the course, but they're definitely a tool we recommend taking a look at before then if you're interested.
3. Honorable Mention: Eclipse, IntelliJ, Spyder, and IDLE
4. (Dis)Honorable Mention: vi, vim (vi improved), Emacs: somewhat joking here, but if you're a glutton for punishment (like me) or are forced into an environment without a graphical interface (which is actually far more common than it probably sounds), vi(m) and Emacs are two of the pre-eminent command-line text editors for writing code.