Using libraries in Python

Programming would be incredibly time consuming if you couldn't reuse code. You've seen the basic form of code reuse already—the function, which allows you to lock up a transformation from some data to some other data, give it a name, and then apply it repeatedly, and to arbitrary data.

Well, it would also be super time-consuming if you couldn't use other people's code. There are many *libraries* (also called "packages" or "modules," but note that these terms can be a little ambiguous) in Python that represent useful code written by other people. Many of them are built into the standard distribution of Python—indeed, a lot of the reason that Python is such a popular programming language is that it comes with a lot of libraries, it's what's called a "batteries included" language. But you can get lots of libraries that aren't included with Python as well, from the Python Package Index.

When you use a library, you have access to the variables it provides (and the stuff those variables point to). For example, a library might provide specialized kinds of data, like a string containing information about the version of Python you're using. It might provide functions that you can call. It might even provide classes (we'll talk about those later).

Using Libraries

In order to use a library, you must `import` it.

The most straightforward way to import a library is to just type `import` then its name. For example, suppose you wanted to find out what the current working directory is? (The current working directory is the directory on your hard drive that Python thinks you're "in," for example, where it will save files generated by your code if you don't specify another location.). Well, if you the documentation for that library, you will see that the built-in `os` library provides a function called `getcwd()` which gives you the name of the current working directory. So you'd run the following code:

```
import os
print(os.getcwd())
```

As you may have guessed, when you import a library, it creates what's known as a *namespace*—what that means is that you can't just refer what it provides by the name of the variable (or "name") on its own. You have to preface it by putting the name of the library and a period. This is a good thing. Otherwise, you could accidentally overwrite names that you've used somewhere else, or names that Python provides in the standard library.

For example, the `json` module provides a function called `load()` that allows you to load files in JSON format and turn them into Python data that your

code can use. But, as you might imagine, "load" is a pretty common name, and other libraries might use it as well. If it wasn't for the namespacing functionality, if you imported the `json` library you couldn't use any other library providing a function with the name "load," because when you imported `json` it would overwrite whatever any other library had assigned to that name. This would be pretty bad.

What if you don't want to refer to a function (or string, or whatever) from within its namespace? Well, then you can import it into a global namespace like this:

```
from os import getcwd
print(getcwd())
```

The `from [library] import [name]` form just imports the specified names from a module directly into the global namespace. What this means is that `getcwd()` will become available to you without prefacing it with the `os` namespace... but no other names from the `os` library will be available, just the specific one you chose to import.

(You could import all the names in the library into the global namespace with `from os import *` but that's usually a terrible idea that will lead to all sorts of bugs in your code.)

You can also rename libraries on import. For example, you might do this if a library has a really long name that's hard to type or you want to rename it to something you'll remember more easily. For example:

```
import os as library_with_the_cwd_function
print(library_with_the_cwd_function.getcwd())
```

This is something that people in the data science world do a lot for some reason. People import the `numpy` library as `np` by convention, for example, and `pandas` gets imported as `pd`. I don't know why this custom started, but it's pretty much universal among data types who use Python. (And Numpy and Pandas are both super important, so you'll be seeing this a lot.)

If you really want, you can also rename specific names, as in:

```
from os import getcwd as working_directory
print(working_directory())
```

Further reading on importing: this Digital Ocean tutorial is pretty good.

Installing Libraries

You shouldn't have to install libraries in this course if you've followed the instructions I've given. Both Azure Notebooks and the Anaconda distribution contain all the libraries that we will be using.

However, if you do want to dig into Python programming further and need to install libraries down the road, here are some tips.

- First, the Anaconda distribution comes with a command line program called "conda." You can install lots of libraries from the special Anaconda package repository with `conda install [libraryname]` at the package.
- The standard program to install things is called "pip." Again, `pip install [libraryname]` at the command line will do you.
- A common and very obnoxious problem that happens with libraries is having incompatible versions of multiple libraries that need to work together. Usually, to avoid this, Python programmers create what's known as virtual environments, collections of libraries that live together and don't talk to other environments on their machines. This is way beyond the scope of the course (and the ecosystem of tools to do it changes rapidly), but if you get into using a bunch of packages, you should really look into doing this, or you'll get yourself into a mess sooner or later.
- You should usually be careful installing libraries that you don't have confidence in (aren't widely known, etc.). They might execute malicious code. This is a security problem that a lot of people are worrying about, and it's actually been a problem a few times—most recently in a library for the JavaScript programming language, which got millions of downloads and then got caught stealing bitcoin. So be careful.