Understand Python libraries, packages, and modules

Recently, you learned about Python libraries, packages, and modules. As you've discovered, importing these tools saves data professionals time and enhances their programming. Another benefit of commonly used libraries is that they are constantly scrutinized and updated by talented and knowledgeable programmers. Thus, you can be confident that the underlying code is high quality.

In this reading, you'll learn more about the basic features of libraries, packages, and modules; how they are related; and a selection of basic modules you might use as a data professional.

Libraries, packages, and modules

A **library** is a corpus of reusable code modules and their accompanying documentation. Libraries are bundled into **packages** that you install, which can then be imported into your coding environment as needed. You'll typically encounter the terms "library" and "package" used interchangeably. Generally, this certificate program will refer to both as libraries, but it's important to be acquainted with both terms.

Modules are similar to libraries, in that they are groups of related classes and functions, but they are generally subcomponents of libraries. In other words, a library can have many different modules, and you can choose to import the entire library or just the module you need.

Import statements

Libraries and modules beyond the Python standard library typically must be imported into your working environment on an as-need basis. Additional libraries are installed first and then imported into your working environment as needed.

To import a library or module, use an import statement. Import statements require particular syntax using the **import** keyword. Here are some examples:

Note: The following code block is not interactive.

```
import numpy
```

This import statement imports the NumPy library into your working environment. After running this command, you'll have access to all NumPy classes and functions. For instance, to use the `array()` function on `[2, 4, 6]`, you'd write:

Note: The following code block is not interactive.

```
numpy.array([2, 4, 6])
```

Notice that to access the `array()` function, you must precede it with `numpy`, because this indicates that the function is coming from the NumPy library.

Aliasing

Another time-saver with Python libraries is aliasing. Aliasing helps you avoid typing a library's full name every time you want to access one of its functions. Instead, you'll assign the library an **alias**. An alias is an abbreviated name, which is designated using the `as` keyword:

Note: The following code block is not interactive.

```
import numpy as np
```

In this case, the NumPy library is imported with the `np` alias. You can assign any abbreviation you like as an alias, but commonly used libraries have common aliases. Therefore, straying from those could cause confusion when sharing code with others. Here are some common libraries and their conventional aliases used by data professionals:

Note: The following code block is not interactive.

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

NumPy is used for high-performance vector and matrix computations. Pandas is a library for manipulating and analyzing tabular data. Seaborn and matplotlib are both libraries used to create graphs, charts, and other data visualizations.

After running these imports, whenever you want to use a function from one of these libraries, precede the function with the alias. Returning to the example with NumPy's `array()` function, after aliasing, you'd write:

Note: The following code block is not interactive.

```
np.array([2, 4, 6])
```

Additional import syntax

Importing modules

Recall from the previous example:

Note: The following code block is not interactive.

```
import matplotlib.pyplot as plt
```

You may have noticed that this syntax differs slightly from the other examples. In this case, matplotlib is the library and pyplot is a module inside. The pyplot module is aliased as `plt`, and it's accessed from the matplotlib library using the dot.

Importing functions

Just as you can import libraries and modules, you can also import individual functions from libraries or from modules within libraries using a specific syntax. Here's an example depicting a common import when using the scikit-learn library to build machine learning models:

Note: The following code block is not interactive.

```
from sklearn.metrics import precision_score, recall_score
```

Again, notice the different syntax. The import statement begins with the **from** keyword, followed by `sklearn.metrics`—the scikit-learn library + the `metrics` module. Next is the `import` keyword followed by the desired functions. In this case, there are two: `precision_score` and `recall_score`.

The same syntax can be applied to the example using NumPy's `array()` function. However, note that you typically would not encounter individual functions being imported from NumPy. It's much easier and more common to just import the whole library.

Note: The following code block is not interactive.

```
from numpy import array
```

When a function is imported by name, like in this example, you can use it without any preceding syntax to indicate the library or module that it comes from:

Note: The following code block is not interactive.

```
array([2,4,6])
```

Discouraged syntax

One last syntactical variation that you might encounter is:

Note: The following code block is not interactive.

```
from library.module import *
```

This imports everything from a particular library or module and allows you to use its functions without any preceding syntax. So, for instance, if you wrote `from numpy import *`, you'd be able to use all of NumPy's functions without preceding them with `numpy` or `np`. **This approach is not recommended** because it makes it difficult to track where functions come from. However, it's helpful to be aware of this because you will likely encounter it in your work as a data professional. And, in specific instances, it might be useful.

Commonly used built-in modules

The Python standard library comes with a number of built-in modules relevant to data professional work such as `math`, `datetime`, and `random`. These can be imported without additional installation. In other words, you can import them directly, as long as you have Python installed. For example:

datetime

- Provides many helpful date and time conversions and calculations

Example:

```
import datetime
today = datetime.date.today()           # assign today's date to a variable
print(today)                            # print today's date
print(today.year)                       # print the year today is in

delta = datetime.timedelta(days=30)     # assign a timedelta of 30 days to a
                                        # variable
print(today - delta)                    # print date of 30 days before today
```

Run

Reset

math

- Provides access to mathematical functions

Example:

```
import math
print(math.exp(0))          # e**0
print(math.log(1))          # ln(1)
print(math.factorial(4))    # 4!
print(math.sqrt(100))       # square root of 100
```

Run

Reset

random

- Useful for generating random numbers

Example:

```
import random
print(random.random())          # 0.0 <= X < 1.0
print(random.choice([1, 2, 3])) # choose a random element from a sequence
print(random.randint(1, 10))    # a <= X <= b
```

Run

Reset

Key takeaways

Libraries, packages, and modules are gateways to Python's countless capabilities. Understanding how to leverage them for your own coding needs will unlock new tools and functions that make your work much more efficient. Check out the Python Package Index at the PyPI repository to search for useful libraries. There are packages designed for applications as diverse as chemistry, audio editing, natural language processing, and video games. Whatever it is you're trying to do, chances are someone has developed a suite of tools to help you!

Mark as completed