Built-In Libraries vs. External Libraries

As we covered in an earlier course, the Python Standard Library comes as part of the Python installation and includes modules for the most common tasks you can do with Python. But there's tons of other things you might want to do in your scripts, and not all of them are in the standard library. This is where external modules come into play. When developers write a Python module that they think others might find useful, they publish it in ***PyPI*** -- also known as the ***Python Package Index*** (https://pypi.org). We can browse this repository of Python modules to find the module we need. It includes thousands of projects, which are classified by different categories, like topic, development status, and intended audience.

In this module, we're going to be ***transforming*** and ***converting*** images. To do that, we'll be using a popular library for image manipulation: the ***Python Imaging Library (PIL)***. The original PIL library hasn't been updated since 2009 and does not support Python 3. Fortunately, there's a current *fork* of PIL called ***Pillow***, that properly supports Python 3 and is kept up-to-date. The Pillow library is packaged with the name **pillow**, but the module name in Python is still **PIL**.

If you try to import the PIL module on a computer that doesn't have pillow (or PIL) installed, you might get an error like this:

```
>>> import PIL
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'PIL'
```

Okay, looks like I don't have that module yet! As we covered in an earlier course, there are several ways to add external modules to your Python environment. PIL is a pretty common library, and on Linux it's usually available as a native package. For example:

```
(...)
Unpacking python3-pil:amd64 (4.3.0-2) ...
Setting up python3-pil:amd64 (4.3.0-2) ...

user@ubuntu:~$ sudo apt install python3-pil
Reading package lists... Done
Building dependency tree
```

For other environments, you should use Python's package installer, ***pip3***. Like this:

```
$ pip3 install pillow
Collecting pillow
  Downloading https://files.pythonhosted.org/packages/85/28/2c72ba965b52884a0bd71e419761fc
162763dc2e5d9bec2f3b1949f7272a/Pillow-6.2.1-cp37-cp37m-macosx_10_6_intel.whl (3.9MB)
     |████████████████████████████████| 3.9MB 1.7MB/s
Installing collected packages: pillow
Successfully installed pillow-6.2.1
```

Once we've done that, we can try to import the module again. And this time it should succeed with no errors:

```
>>> import PIL
```

That's better!

Now, how do you learn to use a library that you've never worked with before? It's time to get familiar with the library's ***Application Programming Interface (API)***!

What is an API?

Application Programming Interfaces (APIs) help different pieces of software talk to each other. When you write a program, you typically use a bunch of existing libraries for the programming language of your choice. These libraries provide APIs in the form of ***external*** or ***public*** functions, classes, and methods that other code can use to get their job done without having to create a lot of repeated code.

And not only that, APIs can also be used by other pieces of software, even if they were written in a completely different programming language. For example, Cloud services use APIs that your programs can communicate with by making web calls. What's special about an API? What makes it different to any other function that you would write in your own code?

If you look at the library's code, you'll find it has many functions that we're not meant to use directly from our code. These ***internal*** or ***private*** functions, classes, and methods do important work, but they're there to support the functions that are published by the library. You probably don't have time to dig in to understand every little bit of how the code works, but you need to know how to interact with the library to do useful work. An API is sort of like a promise. Even if the library's internal code changes, you expect the function to keep accepting the same parameters and returning the same results. That provides a stable ***interface*** to write your code with. That's an API!

Library authors are free to make improvements and changes to the code *behind* the interface, but they shouldn't make changes to the way the functions are called or the results they provide. Because this would break the code that depends on that library. When a library author needs to make a ***breaking change*** to an API, then they need to have a plan in place for communicating that change to their users. That's why breaking changes *should* be saved for major version increments of a library.

When you choose a certain library to use with your code, the first step is to get familiar with its API. You'll need to look at how the functions are called, what inputs they expect, and what outputs they'll return.

How to Make Sense of an API?

How do you learn to use a library or an API that you've never worked with before? It might take you a bit of time to familiarize yourself with how the library operates, but that's okay. It's worth spending some time understanding the way the functions are organized, the inputs and outputs, and the general expectations of the library.

In general, a good API should be descriptive. You should be able to look at a function's name and have a pretty good idea of what it will do. A well-designed API will follow patterns and ***naming conventions***. That means that the functions, classes and methods should have names that help you understand what to expect from them. And when the name isn't enough, you should have access to the documentation for each of the functions that will help you figure out what they do.

For example, when we visit the reference page for the Image object in Pillow, we see this piece of example code:

```
from PIL import Image
im = Image.open("bride.jpg")
im.rotate(45).show()
```

This piece of code is pretty straightforward. Even without having seen this library before, you can probably guess that it opens an image called bride.jpg, rotates it 45 degrees, and then shows it on the screen.

But how can we know for sure? We can look up each of the functions in the documentation and check what they're supposed to do. When dealing with open-source libraries, we can even check out how the function is implemented to see if it matches our expectations. For a web service API or a closed-source library, you might not have access to the underlying code, but you should have access to the documentation that's generated by the code.

For a Python library like PIL, the code is documented using ***docstrings***. If you remember from waaaay back in our first course, docstrings are documentation that lives alongside the code. You've been using them ever since! When you use "help()" to describe a function, or read a description of what a Python function does in your IDE, what you're reading comes from docstrings in the code.

For example, let's take a look at the documentation for PIL:

```
>>> help(PIL)

Help on package PIL:

NAME
    PIL - Pillow (Fork of the Python Imaging Library)

DESCRIPTION
    Pillow is the friendly PIL fork by Alex Clark and Contributors.
        https://github.com/python-pillow/Pillow/

    Pillow is forked from PIL 1.1.7.

    PIL is the Python Imaging Library by Fredrik Lundh and Contributors.
    Copyright (c) 1999 by Secret Labs AB.

    Use PIL.__version__ for this Pillow version.
    PIL.VERSION is the old PIL version and will be removed in the future.

    ;-)

PACKAGE CONTENTS
    BdfFontFile
    BlpImagePlugin
    BmpImagePlugin
    BufrStubImagePlugin
    ContainerIO
    CurImagePlugin
    DcxImagePlugin
    DdsImagePlugin
    EpsImagePlugin
...
```

Lots of Python modules also publish their documentation online. Pillow's full documentation is published here. There, the docstrings have been compiled into a browsable reference, and they've also written a handbook with tutorials for you to get familiar with the library's API. Woohoo!

How to Use PIL for Working With Images

As we've mentioned, for the project in this module, you'll use the Python Imaging Library to process a bunch of images. So, how does that work?

When using PIL, we typically create **Image** objects that hold the data associated with the images that we want to process. On these objects, we operate by calling different methods that either return a new image object or modify the data in the image, and then end up saving the result in a different file.

For example, if we wanted to resize an image and save the new image with a new name, we could do it with:

```
from PIL import Image
im = Image.open("example.jpg")
new_im = im.resize((640,480))
```

```
new_im.save("example_resized.jpg")
```

In this case, we're using the resize method that returns a new image with the new size, and then we save it into a different file. Or, if we want to rotate an image, we can use code like this:

```
from PIL import Image
im = Image.open("example.jpg")
new_im = im.rotate(90)
new_im.save("example_rotated.jpg")
```

This method also returns a new image that we can then use to create the new rotated file. Because the methods return a new object, we can even combine these operations into just one line that rotates, resizes, and saves:

```
from PIL import Image
im = Image.open("example.jpg")
im.rotate(180).resize((640,480)).save("flipped_and_resized.jpg")
```

There's a ton more that you can do with the PIL library. Have a look at the docs and try it on your computer!