Import: Modules and Packages

Our tour through the essentials of Python and NumPy required us to regularly make use of `import` statements. This allowed us to access the functions and objects that are provided by the standard library and by NumPy.

```
# accessing `defaultdict` from the standard library's `collections` package
from collections import defaultdict

# import the entire numpy package, giving it the alias 'np'
import numpy as np
```

Despite our regular use of the `import` statement, we have thus far swept its details under the rug. Here, we will finally pay our due diligence and discuss Python's import system, which entails understanding the way that code can be organized into modules and packages. Modules are individual `.py` files from which we can import functions and objects, and packages are collections of such modules. Detailing this packaging system will not only provide us with insight into the organization of the standard library and other collections of Python code, but it will permit us to create our own packages of code.

To conclude this section, we will demonstrate the process of installing a Python package on your system; supposing that you have written your own Python package, installing it enables you to import it anywhere on your system. A brief overview will be provided of the two most popular venues for hosting Python packages to the world at large: the Python Package Index (PyPI) and Anaconda.org.

The official Python tutorial provides a nice overview of this material and dives into details that we will not concern ourselves with here.

Auto-reload:

As you follow along with this section in a Jupyter notebook include the following code at the top of you notebook:

```
%load_ext autoreload
%autoreload 2
```

Executing these "magic commands" will inform your notebook to actively reload all imported modules and packages as they are modified. If you do not execute these commands, your notebook will not "see" any changes that you have made to a module that has already been imported, unless you restart the kernel for that notebook.

Modules

A Python 'module' refers to a single `.py` file that contains function definitions and variable-assignment statements. Importing a module will execute these statements, rendering the resulting objects available via the imported module.

Let's create our own module and import it into an interactive Python session. Open a Jupyter notebook or IPython console in a known directory on your computer. Now, with an IDE or simple text editor (not software like Microsoft Word!) create a text file named `my_module.py` in the same directory as you Python session. The contents of `my_module.py` should be:

```
"""
Our first Python module. This initial string is a module-level documentation string.
It is not a necessary component of the module. It is a useful way to describe the
purpose of your module.
"""

print("I am being executed!")

some_list = ["a", 1, None]

def square(x):
    return x ** 2

def cube(x):
    return x ** 3
```

Returning to our interactive Python session we can import this module into our session. Python is able to "find" this module because it is in the present directory - more on this later. Importing `my_module.py` will execute all of its code in order from top to bottom, and will produce a Python object named `my_module`; this is an instance of the built-in `module` type. Note that we do not include the `.py` suffix in our import statement.

```
# importing my_module in our interactive session
>>> import my_module
I am being executed!

# produced is a object that is an instance of the module-type
>>> my_module
<module 'my_module' from 'usr/my_dir/my_module.py'>

>>> type(my_module)
module
```

As expected, importing our module causes our `print` statement to be executed, which explains why `'I am being executed!'` is printed to the console. Next, the objects `some_list`, `square`, and `cube` are defined as the remaining code is executed. *These are made available as attributes of our module object*.

```
# all of the variables assigned in the module are made
# available as attributes of the module object
>>> my_module.some_list
['a', 1, None]

>>> my_module.square
<function my_module.square(x)>

>>> my_module.square(2)
4

>>> my_module.cube(2)
8
```

It is critical to understand that this is the means by which the contents of a module is made available to the environment in which it was imported. In this vein, a good way to get to know the contents of a module is to make use of the auto-completion feature provided by IDEs and interactive consoles to list all of the attributes of a module object. The built-in `help` function can also be used to summarize a module's contents:

```
>>> help(my_module)
Help on module my_module:

NAME
    my_module

DESCRIPTION
    Our first Python module. This initial string is a module-level documentation string.
    It is not a necessary component of the module.

FUNCTIONS
    cube(x)

    square(x)

DATA
    some_list = ['a', 1, None]

FILE
    c:\users\ryan soklaski\desktop\learning_python\python\module5_oddsandends\my_module.py
```

Takeaway:

A module is simply a text file named with a .py suffix, whose contents consist of Python code. A module can be imported into an interactive console environment (e.g. a Jupyter notebook) or into another module. Importing a module executes that module's code and produces a module-type object instance. Any variables that were assigned during the import are bound as attributes to that object.

Reading Comprehension: Creating a simple module

Create a simple math module named `basic_math.py`. It should make available the irrational numbers π and e, and the function `deg_to_rad`, which converts an angle from degrees to radians.

Next, import this module and compute $e^{i\pi}$ and compute 45 degrees in radians.

Import Statements

Python provides a flexible framework for importing modules and specific members of a module. We have already seen in our work with NumPy that we can specify an *alias* in our import statement; this can be especially convenient for import modules with long names:

```
# the module object for numpy is returned with the variable name `np`
>>> import numpy as np
>>> np.array([2., 3.])
array([2., 3.])
```

In general we can perform an alias for an import via: `import <module_name> as <alias_name>`.

Next, the pattern `from <module_name> import <thing1>, <thing2>, ...` permits us to import specific objects from the module instead of the importing the entire module as a whole. Let's import `square` and `some_list` from basic_module.py:

```
>>> from my_module import square, some_list
I am being executed!

>>> some_list
['a', 1, None]

>>> square(2)
4
```

Note that the module is still being executed in full, however instead of producing the module-instance `my_module`, this imort statement instead only returns the specified objects that were defined in the module.

Lastly, you can specify `*` to refer to all of the module's attributes.

```
# import all of the contents of `my_module`
>>> from my_module import *
```

You can include in your module a list named `__all__`, which stores attribute names as strings, to restrict those attributes that are referred to by `*`. That is, if we included `__all__ = ["cube", "some_list"]` within `my_module.py`, then `from my_module import *` will only import `cube` and `some_list`, and not `square`.

Lastly, we can also make use of aliasing for this style of import:

```
>>> from my_module import cube as my_cube
>>> my_cube(2)
8
```

Packages

It is not unusual, when working on larger scale projects, to want to organize one's code into several modules. For example, suppose that we are writing software for performing facial recognition. We might want to have a camera module for taking pictures, a face-detection module for storing a model-class capable of detecting faces, and a database module that is used to store and update "seen" faces. These modules can be stored in a collective *package*.

A Python package is a directory containing a file with the name `__init__.py`, along with other Python modules and subpackages (i.e. subdirectories, each with its own `__init__.py` file and associated modules). The `__init__.py` file is of special importance - it serves as the indicator that its housing directory is to be treated as a package. For example, let's build a bare-bones package with the following directory structure:

```
- your current Jupyter notebook / console session
- a_dir/
    |--__init__.py
```

Note that your interactive Python session (e.g. Jupyter notebook) should be in active in the same directory as `a_dir/`. The name of the directory containing the `__init__.py` file is the name of the package, thus the name of this package is `a_dir`. Suppose the contents of `__init__.py` is as follows:

```
def sum_func(x, y):
    return x + y

def divide_func(x, y):
    return x / y
```

As with a module, importing this package will execute the contents of `__init__.py` and make available `sum_func` and `divide_func` as attributes of the resulting module object:

```
# importing a python package
>>> import a_dir
>>> a_dir.divide_func(1, 2)
0.5
```

Let's graduate to a more fleshed out package, which contains modules and subpackages. The following package, `face_detection`, possesses the modules `utils`, `database`, and `model`. It also contains the subpackage `camera`, which contains the `config` module and the `calibration` module.

```
- face_detection/
    |-- __init__.py
    |-- utils.py
    |-- database.py
    |-- model.py
    |-- camera/
        |-- __init__.py
        |-- calibration.py
        |-- config.py
```

We can access the contents of these modules via `<package>.<module>`, `<package>.<subpackage>.<module>` and so on. Consider the following examples:

```
# importing a function from the `database` module
>>> from face_detection.database import load_database

# importing the entire `model` module
>>> from face_detection import model

# importing a function from the `camera` subpackage
>>> from face_detection.camera.config import restore_default
```

See that the `.` syntax allows us to drill progressively deeper down into modules and subpackages, relative to our top-level package.

Reading Comprehension: Packages

Suppose that we are working with a package named `mail`. The `mail/` directory contains an `__init__.py` module whose contents are:

```
def send_mail(x):
    return x

phrase_of_the_day = "get that package delievered!"
```

It also contains the module `delivery.py` with the contents:

```
def get_zip():
    """just a dummy function"""
    return 871092
```

1. Create this package.
2. Import the send_mail function
3. Import the delivery module and then execute it's get-zip function

Intra-Module Imports

Modules within a package can import from one another; suppose, for instance, that both `face_detection.database` and `face_detection.camera.calibration` both want to leverage the `face_detection.utils` module. Recall the layout of this package:

```
- face_detection/
    |-- __init__.py
    |-- utils.py
    |-- database.py
    |-- model.py
    |-- camera/
        |-- __init__.py
        |-- calibration.py
        |-- config.py
```

There are two import styles that can be used to facilitate these inter-module imports: absolute imports and relative imports.

Absolute Imports

The absolute import style works by specifying all modules in terms of their absolute position in relation to the topmost package. Suppose that we want to import the `utils` module in our `model` module, stated as an absolute import, this would be:

```
import face_detection.utils
```

or, using an alias

```
import face_detection.utils as utils
```

Suppose that we want to import the `utils` module in `face_detection.camera.calibration` as well; the absolute import would look identical since the absolute location of `utils` relative to the top-level package does not change based on where we are using this import statement.

As an additional example, to import the camera configuration module anywhere in the package, the absolute import statement would appear as:

```
import face_detection.camera.config
```

The absolute import syntax supports all of the variations the we enumerated above, such as aliased imports and `from <module> import <object>`.

Relative Imports

Relative imports use dots to indicate the relative position of the imported module, based on the location of the module doing the importing. For example, suppose that we want to import the `utils` module in our `model` module using a relative import statement:

```
from . import utils
```

See that `.` serves to represent "the current package". `..` then refers to "one package above the current one". Thus a relative import of `utils` from `face_detection/camera/calibration.py` would look like:

```
from .. import utils
```

We use `..` because `utils` is not in the same package as `calibration.py`, but is one package above it. We can also import specific contents from a module in this way:

```
from ..utils import some_util_func
```

The relative import style is more restricted than the absolute import style. It is only able to embody the `from <module> import <thing1>, <thing2>` form of import statements. Despite their limited form, relative imports can be nice to use if you are dealing with a project with deeply nested sub-packages.

Installing a Package

PYTHONPATH and Site-Packages

Thus far in our reading of Python packages we have had to take care that all of the modules and packages that we have written reside in the same directory as our interactive Python session. How is it that we are able to import NumPy in any session or module, without any knowledge of where that package is located? This is because we have installed *NumPy*, which means that the package has been placed in our "Python path", which is indicated as `PYTHONPATH`.

The `PYTHONPATH` specifies the directories that the Python interpreter will look in when importing modules. You can check your `PYTHONPATH` using `sys.path`:

```
# looking up `PYTHONPATH`
>>> import sys
>>> sys.path
['',
 '/home/TerranceWasabi/miniconda3/bin',
 '/home/TerranceWasabi/miniconda3/lib/python36.zip',
 '/home/TerranceWasabi/miniconda3/lib/python3.6',
 '/home/TerranceWasabi/miniconda3/lib/python3.6/lib-dynload',
 '/home/TerranceWasabi/miniconda3/lib/python3.6/site-packages'
]
```

Note that the first entry in `PYTHONPATH` is `''`, meaning that the Python interpreter will first look in the current directory when trying to do an import. If it does not find any package or module that satisfies the import statement, then it will proceed to check the next entry in `PYTHONPATH`. This is why we took care to create our modules and packages in the same directory as our active Python session.

Now note the last directory in `PYTHONPATH`: site-packages. *Site-packages is the target directory in which all installed Python packages are placed by default*. We can import NumPy wherever we'd like because it was placed in site-packages when it was installed, and the Python interpreter will always look in site-packages when attempting to fulfill an import statement.

In lieu of printing out your `PYTHONPATH`, you can look up the location of your site-packages directly:

```
# looking up your site-packages
>>> import site
>>> site.getsitepackages()
['/home/TerranceWasabi/miniconda3/lib/python3.6/site-packages']
```

It must be mentioned that we are sweeping some details under the rug here. Installing NumPy does not merely entail copying its various modules and packages wholesale into site-packages. That being said, we will not dive any deeper into the technical details of package installation beyond understanding where packages are installed and thus where our Python interpreter looks to import them.

Installing Your Own Python Package

Suppose that we are happy with the work we have done on our `face_detector` project. We will want to install this package - placing it in our site-packages directory so that we can import it irrespective of our Python interpreter's working directory. Here we will construct a basic setup script that will allow us to accomplish this.

We note outright that the purpose of this section is strictly to provide you with the minimum set of instructions needed to install a package. We will not be diving into what is going on under the hood at all. Please refer to An Introduction to Distutils and Packaging Your Project for a deeper treatment of this topic.

Carrying on, we will want to create a setup-script, `setup.py`, *in the same directory as our package*. That is, our directory structure should look like:

```
- setup.py
- face_detection/
    |-- __init__.py
    |-- utils.py
    |-- database.py
    |-- model.py
    |-- camera/
        |-- __init__.py
        |-- calibration.py
        |-- config.py
```

The bare bones build script for preparing your package for installation, `setup.py`, is as follows:

```
# contents of setup.py
import setuptools

setuptools.setup(
    name="face_detection",
    version="1.0",
    packages=setuptools.find_packages(),
)
```

If you read through the additional materials linked above, you will see that there are many more fields of optional information that can be provided in this setup script, such as the author name, any installation requirements that the package has, and more.

Armed with this script, we are ready to install our package locally on our machine! In your terminal, navigate to the directory containing this setup script and your package that it being installed. Run

```
python setup.py install
```

and voilà, your package `face_detection` will have been installed to site-packages. You are now free to import this package from any directory on your machine. In order to uninstall this package from your machine execute the following from your terminal:

```
pip uninstall face_detection
```

One final but important detail. The installed version of your package will no longer "see" the source code. That is, if you go on to make any changes to your code, you will have to uninstall and reinstall your package before your will see the effects system-wide. Instead you can install your package in develop mode, such that a symbolic link to your source code is placed in your site-packages. Thus any changes that you make to your code will immediately be reflected in your system-wide installation. Thus, instead of running `python setup.py install`, execute the following to install a package in develop mode:

```
python setup.py develop
```

pip and conda: Package Managers

Python packages can be shared worldwide. There are two widely-used Python package managers, `pip` and `conda`. `pip` downloads and installs packages from The Python Package Index (PyPI), whereas `conda` downloads and installs packages from the Anaconda Cloud. Both `conda` and `pip` are installed as part of the Anaconda distribution.

To install a package via `pip`, you execute

```
pip install <package_name>
```

To install a package via `conda`, you execute

```
conda install <package_name>
```

Both managers will install packages to your site-packages directory.

There are substantial benefits for using `conda` rather than `pip` to install packages. First and foremost, `conda` has a powerful "environment solver", which tracks the inter-dependencies of Python packages. Thus it will attempt to install, upgrade, and downgrade packages as needed to accommodate your installations. Additionally, the default list of packages available via `conda` are curated and maintained by Continuum Analytics, the creators of Anaconda. To elucidate one of the benefits of this, installing NumPy via `pip` will deliver the vanilla version of NumPy to you; `conda` will install an mkl-optimized version of NumPy, which can execute routines substantially faster. Finally, `conda` also serves as an environment manager, which allows you to maintain multiple, non-conflicting environments that can house different configurations of installed Python packages and even different versions of Python itself.

That being said, there are some benefits to using `pip`. PyPi is accessible and easy to upload packages to; this is likely the easiest means for distributing a Python package worldwide. As such, `pip` provides access to a wider range of Python packages. That being said, `conda` can also be directed to install packages from custom channels - providing access to packages outside of the curated Anaconda distribution. This has become a popular method of installation for machine learning libraries like PyTorch and TensorFlow.

You are free to install some packages using `conda` and others with `pip`. Just take care not to accidentally install the same package with both - this can lead to a real mess.

Links to Official Documentation

- Python Tutorial: Modules
- An Introduction to Distutils
- Packaging Your Project
- PyPi

Reading Comprehension Exercise Solutions:

Creating a simple module: Solution

Create a simple math module named `basic_math.py`. It should make available the irrational numbers π and e, and the function `deg_to_rad`, which converts an angle from degrees to radians.

The contents of `basic_math.py` should be:

```
"""Basic math constants and functions"""

pi = 3.141592653589793
e = 2.718281828459045

def deg_to_rad(angle):
    return (pi / 180) * angle
```

```
>>> import basic_math

# Euler's formula: e**(i * pi) = -1
>>> basic_math.e ** (basic_math.pi * complex(0,1))
(-1+1.2246467991473532e-16j)

>>> basic_math.deg_to_rad(45)
0.7853981633974483
```

Packages: Solutions

1. Create this package.

```
mail/
    |-- __init__.py
    |-- delivery.py
```

2. Import the send_mail function

```
>>> from mail import send_mail
```

3. Import the delivery module and then execute it's get-zip function

```
>>> from mail import delivery
>>> delivery.get_zip()
871092
```