Infrastructure as Code- Walkthrough

Modules and Packages

Contents

A **walkthrough** is intended to bring you through a technical exercise. A walkthrough shows you how I completed a task in a particular context, on specific systems, at a point in time. The document is a good basic guide, but you should always confirm that the information is current and if a more recent best practice exists, you should make yourself aware of it.

Introduction

This walkthrough is provided to demonstrate how to get started coding with modules and packages in Python. In Python we have the standard libraries which come default. There are millions of lines of code available in 3rd party libraries that have been written over the years. And we can write our own libraries of functions. If I do work in a single thematic area once, as I properly structure and save the code, I can easily reuse this code in any later project.

Read https://realpython.com/python-modules-packages/ to get some background.

Since Python 3 was released, we have had a technique called virtual environments. I am not going to cover them here, but do some background reading to understand what they are.

Deliverables

You will be required to keep a list of commands with notes as to why you are using them as well as results of any testing you have done. You may be asked to present these notes as evidence of work completed or as part of a lab book. Check with your lecturer!

Prerequisites

1. I will carry out my exercises using a Windows 11 desktop. You can carry out this work on almost any modern platform.
2. You should have already installed Python as per my notes.
3. You should have already installed VSC as per my notes.
4. You have created a directory to keep example files in and you are ready to code.
    a. This could be on your OneDrive, in these examples, I am using **OneDrive\Python\Exercises_06**

Modules

Modules are just .py scripts that can be called from another .py script.

I'm going to create two programs

- **project.py** is the code for a project I might write
- **reusable.py** is the code for the utilities that I could reuse across multiple projects.

```
"""
project.py
"""
import reusable
print("Running code from the project")
print(reusable.my_square(4))
```

```
"""
reusable.py
"""
def my_square(a: int)->int:
    print("Running code from the module")
    return a*a
```

If I run project.py I get the following output.

```
Running code from the project
Running code from the module
16
```

The Python Standard Library

There are libraries of standard functions available, take a few minutes and review the documentation [1]. Python code in one module gains access to the code in another module by the process of importing it.

1. As a simple example, if I wanted to do mathematics, I could write functions to derive trigonometry values from first principles. Instead of each individual programmer doing this, we can call the **math** library from the Python standard library. Create a file called **library1.py** with the following content, run it to test.

```
import math
print("Input lengths of the two short triangle sides:")
a = float(input("a: "))
b = float(input("b: "))
c = math.sqrt(a**2 + b**2)
print("The length of the hypotenuse to four decimal places is: {hypo:1.4f}".format(hypo=c))
```

The import statement does not make the contents of a module directly accessible. Instead, it creates a namespace, making the contents available. In this case the function **sqrt** becomes available as part of the **math** package.

2. Create a file called **library2.py** with the following content, run it to test.

```
from math import sqrt
print("Input lengths of the two short triangle sides:")
a = float(input("a: "))
b = float(input("b: "))
c = sqrt(a**2 + b**2)
print("The length of the hypotenuse to four decimal places is: {hypo:1.4f}".format(hypo=c))
```

It is better coding practice to import the specific function you require from a module, allowing the function to be directly accessed within your code. Do not use separate lines, for example, to import trigonometry functions use

from math import sin, cos, radians

Packages

To help organize modules and provide a naming hierarchy, Python has the concept of packages.

Packages are just collections of modules, and we normally keep them in a separate directory. We add an empty file with the filename **__init__.py** to signal to Python that this is a package.

Under the folder Exercises_06, I create a folder called **mylib**. Under this folder, I create an empty file called **__init__.py** marking this as a package. I add the following line to **__init__.py**

copyright = "© JOR 2022"

This is now a global variable accessible throughout.

There are much more useful things to put into **__init__.py** but I'll leave you to figure that out as your coding skills develop.

1. Under the folder **Exercises_06**, I create and test a file called **project1.py** as shown.

```
import mylib
print(mylib.copyright)
```

2. Under the folder **Exercises_06\mylib**, I create a file called **square.py** with the following content. I run it to test.

```
def square(x):
   return x*x

print(square(2))
```

3. Under the folder **Exercises_06\mylib**, I create a file called **cube.py** with the following content. I run it to test.

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

print(cube(2))
```

4. Back up to the folder **Exercises_06**, I create a file called **project2.py** with the following content. I run it to test.

```
import mylib.cube as mycube
import mylib.square as mysquare

print(mycube.cube_text, mycube.cube(3))
print(mysquare.square_text, mysquare.square(3))
```

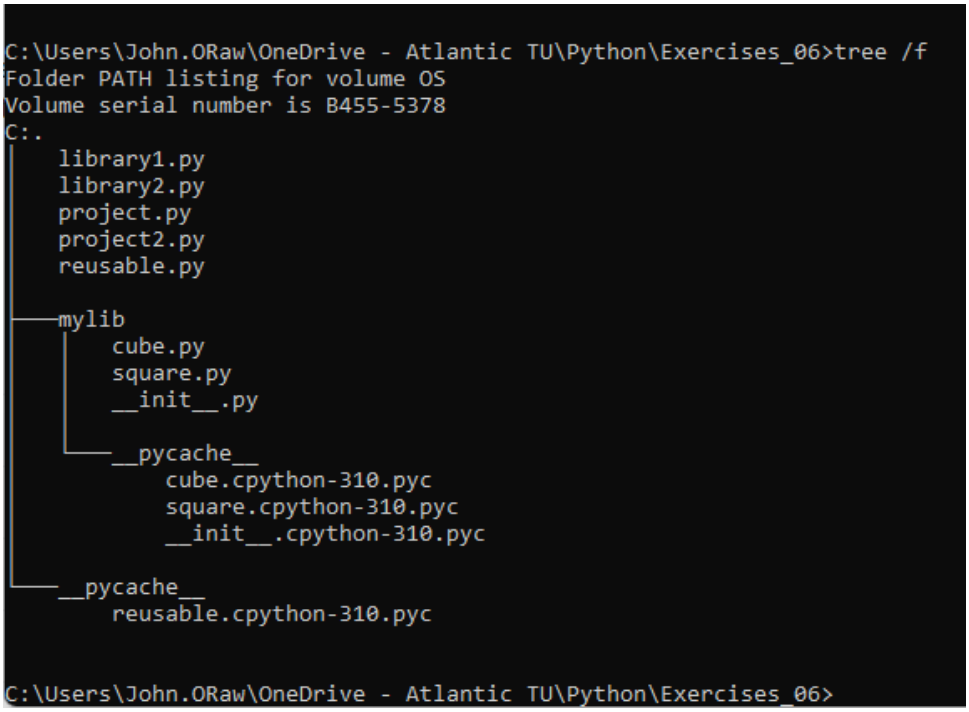
There are quite a few ideas to take away from this.

I can build packages of reusable code, making all my work modular. To reuse this code in a different project, all I need to do is to copy the entire directory. Obviously, I need heavy commenting and comprehensible documentation to make this really work.

In the examples, I have used functions, but I've also created and used variables from within my packages.

I can use the **as** keyword to reference an imported function and give it any name I like to make it meaningful in my code. Note that I can apply this technique to an entire module also. For example, the graphing library **matplotlib** is typically imported as **plt**, making code more economical and readable.

In case you are lost, I have used the tree command to map my directory structure for this exercise so far.



In a large project, I may have several packages which I call, and I need to keep track of all of them.

In a recent project I had network utilities, serial utilities, the decoding of UBlox GPS equipment, the decoding of standard marine instruments, all as separate packages. Imagine being able to pull years of work into a project in a modular way, without having to worry too much about interdependencies, it all just works!

Module or Import?

I edited **cube.py**, I added a line as shown.

```
cube_text = "Yo, time to cube stuff!"

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

print(cube(2))
print(f"This module is called {__name__}")
```

In Python, there are many variables which use a double underscore, like **__main__** and we call these dunder variables. when I run a script, Python sets the dunder variable **__name__** to **__main__**

If I run **cube.py**, I get the response

This module is called __main__

When a .py file is imported as a module, the variable **__name__** is set to the name of the module.

Examine and try running **project2.py** which calls **cube.py** as a module, I now get the response

This module is called mylib.cube

I now edit cube.py to best practice and test it.

```
cube_text = "Yo, time to cube stuff!"

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


if (__name__ == '__main__'):
   print(f"This module is called {__name__} and executes as a standalone script")
else:
   print(f"This module is called {__name__} and is being called by another script")
```

If I run the module by itself, it runs the **cube.py** code under the **if** statement and tells me it is a standalone script.

This module is called __main__ and executes as a standalone script

If I run **project2.py** it runs the **cube.py** code under the **else** statement, and I get the message

This module is called mylib.cube and is being called by another script

In this way, I set a design pattern for simple scripting. I write my functions and my variables, and I can call then from within the script for basic functionality.

If I want to use this script in a different project, as a module, I can include it and only the functions and variables will be accessible when the module is called.

Exercise

Tidy up **cube.py** and **square.py** so they follow this pattern. Include code under the if statement so you can verify that the modules till work.

Public Packages

A programmer can write code for their use, or they take it publicly available for anybody to use. The Package Installer for Python (PIP) [2,3] is the standard tool used to find and install these packages. This is installed when you install Python.

Read the documentation at https://packaging.python.org/en/latest/tutorials/installing-packages/

There is an index that **pip** can access called the Python Package Index or PyPi [4].

At the command prompt in Windows, type **pip -h**

In Linux or Apple, this may be **pip3 -h**

Once you confirm pip is installed, update it by using **pip install --upgrade pip**

One of the most used and useful external packages in Python is **numpy**, however, it is not included in the base libraries.

Open a new file under **Exercises_06** as **public.py** and enter the line **import numpy**

VSC will underline **numpy** as it does not have a package of that name. If you run this, it will generate an error. In the VSC terminal window, I type **pip install numpy**

I restart VSC and numpy is recognized!

You can also remove packages by typing **pip uninstall *package_name***

You can list the packages install by typing **pip list**

You can get information about a package by typing **pip show *package_name***

Work your way through https://pip.pypa.io/en/stable/getting-started/

Finally

As projects get bigger and more complex, so too does the complexity of the underlying modules and packages. In the examples covered here, I created a single package with multiple module files, all of which were trivial. In a large project I might have a package called networking. Under that I might have a separate package for dealing with all things TCP. I might have another separate package for dealing with all things UDP. and just for good measure, multicast we'll also have its own package. Very quickly we end up with a nested directory tree of packages, all of which can be called relative to the root directory the application is in.

In your spare time, play with this concept, but you do need to implement it in any projects for this module.

References

[1] https://docs.python.org/3/library/

[2] https://pypi.org/project/pip/

[3] https://github.com/pypa/pip

[4] https://pypi.org/