Python Modularization and Dependency Management for the Java Programmer

## Abstract

This article introduces Python dependency management practices for the Java programmer working with Python and PySpark for Data Engineering. This article references another article that details setting up a development environment that permits creating python libraries, and includes a sample project created with the *PyCharm* IDE, as well a sample *PyBuilder* project (Python Maven equivalent). Please see my DevOps article at <https://www.linkedin.com/pulse/setting-up-python-project-virtual-environment-pycharm-donald-trummell> for project creation details. Documentation resources are included in the reference sections at the end of both articles.

Our goal is to cover these Python modularization concepts:

* Dependency Management – Modules.
* Dependency Management – Virtual Environments.
* Deployable artifacts – library formats
* Sample Python and Java versions of a simple “library”.

We will compare a simple Java and Python implementation to illustrate concepts.

We Java programmers benefit from a mature dependency management framework for the Java eco-system, and we are often disappointed with the state of dependency management in Python. There are many Python “packaging” systems, and at least two “standard” packaging mechanisms in Python, while Java benefits from having one standard library mechanism and a deployment mechanism defined by Java EE.

Python tackles modularization in a flexible manner, and is much more like its C language parent in that it is file based (the *module*) rather than class based. A Python file (*module*) located in a directory structure is the primary name-space and containment mechanism for Python. This differs from Java’s use of the class file as the sole name-space and containment mechanism.

A Python package, based on a directory containing zero or more modules, optionally uses a possibly empty file named “***\_\_init\_\_.py*** “ in the module parent directory. This ***init*** file performs operations at the time of import into a dependent module, and is similar to the Java ***static*** block in purpose.

We will illustrate a comparison between Python and Java modularization and dependency management with a small application written in both Python and Java. We first discuss the modularization and deployable artifact concepts of both languages.

## Python Modularity Mechanisms

A Python module is a file consisting of Python code. A module can define functions, classes, variables, and can also include runnable code. Python source modules are compiled to “***.pyc***” byte-code files by the Python interpreter. Python module dependencies are “imported” into the source code module by referencing the dependency with an “import” statement (see <https://www.tutorialspoint.com/python/python_modules.htm> for a quick overview and explanation.)

## Virtual Environments

Python modules, compiled or source, are managed using a Python virtual environment. A virtual environment allows module dependency installation that is specific to the application, and will not “pollute” the global (system) Python environment. A virtual environment for an application is basically a set of directories containing a copy of the python interpreter and related applications; these file are then accessed by a class path defined by the virtual environment *activate* command.

The virtual environment approach solves the problem of creating two different applications depending on different versions of a common library. Java accomplishes this by using a different class path for each application, where the class path includes the correct JAR file for each application. What is not solved by Python is depending on two different versions of the *same* library within an application. Well, actually, Java 9 and beyond accomplishes this with the Java Module system.

## Python Deployable Artifacts

Python allows source code module deployment, as well as deployment of other language source code (e.g., C++). Python also allows the deployment the equivalent of Java byte code; the Python compiled byte-code in a “***.pyc”*** file.

The Python byte-code files are, in general, not portable. They depend on the OS and Python version used to compile them. Java is both portable and backward compatible. For example, a Java version 8 byte-code file will execute on all versions of Java eight or higher, but Java 9 or higher byte-codes will not run on Java 8. The safest way to deploy Python is to deploy source to a system known to have a compatible Python interpreter.

## Appendix 1: PyCharm Community Edition Project Setup

These instructions are for PyCharm 2019.2.3. You can download PyCharm from <https://www.jetbrains.com/pycharm/download/>. We will also setup a sample package structure compatible with ***PyBuilder***.

1. Launch and create new project ***MyAppPy***.
2. On my system, this creates a ***venv*** environment using the current system interpreter. Lets change the interpreter and create a Virtualenv for the selected interpreter: *File -> settings -> Project: MyAppPy -> Project interpreter.*
3. In gear to the right of the *Project interpreter* drop-down, select *add*.
4. The project interpreter dialog allows you to select Virtual Environment, and optionally select globals.
5. Click OK and continue to apply the results. At a high-level, the empty project structure will be:

D:\TEMP2\DEPENDENCIES\MYAPPPY

├───.idea

│ ├───inspectionProfiles

│ └───libraries

└───venv

├───Include

├───Lib

│ └───site-packages

│ └───pip-19.0.3-py3.7.egg

│ ├───EGG-INFO

│ └───pip

│ ├───\_internal

│ └───\_vendor

│

└───Scripts

The ***Virtualenv*** is activated within the PyCharm environment when you open the project, as viewed in the terminal:

Microsoft Windows [Version 10.0.18362.418]

(c) 2019 Microsoft Corporation. All rights reserved.

(venv) D:\Temp2\Dependencies\MyAppPy>

When working at the system command prompt, the virtual environment must be activated and deactivated, independent of PyCharm. Here is an example:

D:\Temp2\Dependencies\MyAppPy>python

Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> quit()

D:\Temp2\Dependencies\MyAppPy>venv\Scripts\***activate.bat***

(venv) D:\Temp2\Dependencies\MyAppPy>python

Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> quit()

(venv) D:\Temp2\Dependencies\MyAppPy>venv\Scripts\***deactivate.bat***

D:\Temp2\Dependencies\MyAppPy>python

Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> quit()

D:\Temp2\Dependencies\MyAppPy>

## Appendix 2: PyBuilder Sample Project Setup

These instructions are executed after setting up your PyCharm project. You can install PyBuilder from PiPy (<https://pypi.org/>).

Sdfdfdfdsf sdfgdsf

# References - Resources

1. Virtualenv – used to create a controlled Python runtime environment: <https://pypi.python.org/pypi/virtualenv>).
2. Additional Virtualenv documentation: <https://virtualenv.pypa.io/en/latest/>).
3. Venv background: <https://realpython.com/python-virtual-environments-a-primer/>.
4. PYC files not portable: <https://github.com/conan-io/conan/issues/5404>, and <https://sft.its.cern.ch/jira/browse/CORALCOOL-2358?workflowName=CORAL+and+COOL+%28edit+closed%29&stepId=6>.
5. Explanation of Python Import: <https://chrisyeh96.github.io/2017/08/08/definitive-guide-python-imports.html>.
6. Official documentation of the import system: <https://docs.python.org/3/reference/import.html#regular-packages>.
7. Dan Bader’s complete course on dependency management: <https://dbader.org/products/managing-python-dependencies/>.
8. Official Python Packaging Authority Users Guide: <https://packaging.python.org>.
9. Need for Python static type checking in large code bases: <https://blogs.dropbox.com/tech/2019/09/our-journey-to-type-checking-4-million-lines-of-python/?utm_medium=email&utm_source=topic+optin&utm_campaign=awareness&utm_content=20190918+data+nl&mkt_tok=eyJpIjoiT0RjME56VmlaV1JoWkdRNSIsInQiOiJkelhxOHhtNmhQRkFrQmJMTlExRmcxV1RCeTV4Z1ZMNkxCalJxcEVmbGhZRkJvQ2U3NVwvd2p4NHhjcW1qUDJqMHVoK1wvWlRBNzZvN3NpNUdOa2NHXC91c3Q5YUVVUCtHaGlBWDcwdVRpQ0VBMmd3M2dcL010dUd2OUJTXC9hRzlzczllIn0%3D>.

## PyBuilder Documentation

1. PyBuilder Documentation Home: <http://pybuilder.github.io/>.
2. PyBuilder GitHub repository: <https://github.com/pybuilder/pybuilder>.
3. PyBuilder master ***build.py*** link in GitHub: <https://github.com/pybuilder/pybuilder/blob/master/build.py>.
4. PyBuilder tutorial (top level): <https://pybuilder.readthedocs.io/en/latest/walkthrough-new.html>.
5. PyBuilder tutorials: <http://pybuilder.github.io/documentation/tutorial.html#.XaJXGkZKiUk>.
6. PyBuilder PDF: <https://buildmedia.readthedocs.org/media/pdf/pybuilder/stable/pybuilder.pdf>.

## Databricks Documentation

1. Databricks library documentation: <https://docs.databricks.com/libraries.html>.

# Bibliography

1. Expert Python Programming, Third Edition © 2019 (ISBN 978-1-78980-889-6).
2. Mastering Python, © 2016 (ISBN 978-1-78528-972-9).
3. Mastering PyCharm, © 2015 (ISBN 978-1-78355-131-6) [interpreter paths, 86].