## DevOps and Application Construction

We did not cover DevOps in our meeting. That is, constructing an application using a CI/CD methodology. Nor did we deal with Application Modularity. Modularity is used to rapidly customize and divide an application into services. I have to do this for both Java and Python applications. These considerations inform application architecture as well.

**CI/CD**

Gap, and many other enterprises, will not distribute Python code via PIP and PyPI. Standard PIP and **setup.py** can’t be used for application deployment in these circumstances. As a result, creating a deployment for a Python application in Azure Functions, or AWS Lambda, involve many tedious steps that are generally scripted:

1. Create a zip file with your Python code appropriately structured.
2. Copy the Zip to an s3 bucket (or a Blob or GitHub on Azure)
3. Setup an appropriate instance and add a Python interpreter
4. Setup a virtual environment
5. Run a script to PIP install library dependencies

Azure does a little better then AWS in terms of tooling support, but similar steps apply. Naturally, Java has better support because it is more mature in this space.

**Modularity**

Application functionality is broken into units often referred to as “modules”. Model elements (functions and classes) should be closely related to the task they perform (high cohesion.) Conversely, modules in your application should be usable outside your application with minimal dependence on other application modules (loose coupling.)

Modules follow these tenants:

* Strong encapsulation (execution code not visible.)
* Specified interface (only public interface is accessible to a client.)
* Explicit dependencies (module lists it’s dependencies)

Python does not directly support the three tenants, but with programmer discipline, it can mimic them. The biggest module management problem is dependencies. Transitive dependencies are discovered as modules are readied for execution. This problem often surfaces as multiple references to different versions of libraries. Java 9 attempts to remedy this with a language modification; but Python has no “automated” answer.

*Dynamic Invocation*

The “strategy pattern” is heavily used in application architecture, and is supported by modularization. Common strategy examples include choice of JDBC driver, or the resource representation (JSON/XML) of a service response.

One generally creates an instance of a function (or class), which is “polymorphic” in terms of the request, but performs the unique action required by the strategy. Here is a Python example:

hdfs\_readers = {AVRO\_TYPE: \_create\_avro\_input\_df, ORC\_TYPE: \_create\_orc\_input\_df,

PARQUET\_TYPE: \_create\_parquet\_input\_df}

. . .

# Create input dataframe

input\_df = hdfs\_readers[file\_type](file\_type, hdfs\_file, hive\_ctx)

This approach works well when the code is known at compile time. Java frameworks extend this strategy driven approach. They allow an application to use strategies where the code is only supplied at runtime. Simulating this in Python is possible, but tedious and again, requires programmer discipline. Python requires dynamic importation of code (e.g., \_\_import\_\_ et. Al.), and must have the PYTHONPATH appropriately adjusted.

**Python Importing references:**

* Complete, current, and understandable: <https://dev.to/0xcrypto/dynamic-importing-stuff-in-python--1805>
* Older, but still useful: <https://www.blog.pythonlibrary.org/2012/07/31/advanced-python-how-to-dynamically-load-modules-or-classes/>
* Azure note: Functions integrates with BitBucket, Dropbox, GitHub, and Azure DevOps (see <https://docs.microsoft.com/en-us/azure/azure-functions/functions-continuous-deployment>.)