How To - PYTHON ONBOARDING

1. Python Project folder structure

```
Project/
    bin/
    CHANGES.txt
    docs/
    LICENSE.txt
    MANIFEST.in
    README.txt
    setup.py
    requirements.txt
    project/
        __init__.py
        location.py
        utils.py
        test/
            __init__.py
            test_location.py
            test_utils.py
```

1. Put into **`bin`** any scripts you've written that use your package and which you think would be useful for your users. If you don't have any, then remove the **`bin`** directory.
2. Put into **CHANGES.txt ,** release history of your project. Example,

```
v0.1.1, 12/03/2013 -- Complete refactory of code to isolate NMAP objects.
v0.1.0, 11/03/2013 -- First developement release packaged for Project Ninaval
```

3. The `docs` **dir** should contain any design docs, implementation notes, a FAQ, or any other docs you've written.
4. The **`LICENSE.txt`** file is often just a copy/paste of your license of choice. We recommend going with a commonly-used license, such as the GPL, BSD, or MIT.
5. A **MANIFEST.in** file can be added in a project to define the list of files to include in the distribution built by the sdist command. When sdist is run, it will look for the **MANIFEST.in** file and interpret it to generate the **MANIFEST** file that contains the list of files that will be included in the package.
6. The `README.txt` file should be written in reST so that the PyPI can use it to generate your project's PyPI page. Example,

```
======================================================================
Python training package, Version 2.7-3.5, July 2016.
This is README.txt, the class material's top-level user guide.
Author/Instructor: Narinder, http://learning-python.com.
======================================================================

This package contains the class workbook in HTML format, Python
installers, and extra examples.  It is available both on CD or
USB stick, and on the Web at http://learning-python.com/class.
The web version is identical to that on CDs or USB stick, but
has no Python installers (get these from python.org/downloads)
or ebook copies (get these in your CD/USB distribution).
```

7. **Setup.py** file is used for package and distribution management. Example,

```
from distutils.core import setup

setup(
    name='TowelStuff',
    version='0.1.0',
    author='J. Random Hacker',
    author_email='jrh@example.com',
    packages=['towelstuff', 'towelstuff.test'],
    scripts=['bin/stowe-towels.py','bin/wash-towels.py'],
    url='http://pypi.python.org/pypi/TowelStuff/',
    license='LICENSE.txt',
    description='Useful towel-related stuff.',
    long_description=open('README.txt').read(),
    install_requires=[
        "Django >= 1.1.1",
        "caldav == 0.1.4",
    ],
)
```

8. The actual module will reside under the project directory i.e. project/ . If your module consists of only a single file, you can place it directly in the root of your repository.
9. The __init__.py files are required to make Python treat the directories as containing packages. It can be empty.
10. The **test/** directory contains the test written for your package. It can contain package integration and unit tests. A small test suite will often exist in a single file at the root directory. Once a test suite grows, you can move your tests to a directory **test/.**

2. Compile/Build/Test/Publsih using Jenkins File

1. Create a **jenkins** file at the root level of your repository.
2. The **jenkins** file can contain **compile**,**build**,**test** and **package** stages. Example,

```
pipeline {
    agent {label 'python-slave'}
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') {
            steps {
                sh 'python -m py_compile sources/add2vals.py
sources/calc.py'
            }
        }
        stage('Test') {
            steps {
                sh 'python test/test.py'
            }
            post {
                always {
                    junit 'test-reports/*.xml'
                }
            }
        }
        stage('Package') {

            steps {
```

```
                sh 'python setup.py bdist'
            }

        }
  stage("Publsih to Nexus"){

   steps {

    nexusArtifactUploader artifacts: [[artifactId: 'com.python.NV',
classifier: '', file:
'/home/jenkins/workspace/POC/Python-pipeline/dist/py-jenkins-0.1.0.
linux-x86_64.tar.gz',
    type: 'tar.gz']], credentialsId: 'nexus', groupId:
'com.python.NV', nexusUrl: 'novartis.devops.altimetrik.io:8082',
nexusVersion: 'nexus3', protocol:
    'http', repository: 'Python-Project', version:
'{$BUILD_NUMBER}'

   }
        }
  stage("Publsih to Jfrog"){

   steps {

    rtUpload (
       serverId: 'jfrog',
       spec: '''{
          "files": [
         {
          "pattern":
"/home/jenkins/workspace/POC/Python-pipeline/dist/py-jenkins-0.1.0.
linux-x86_64.tar.gz",
          "target": "python/"
         }
           ]
         }'''
      )

   }
        }
  stage('Install Sonar-Scanner') {

            steps {

                sh '''wget -q
https://binaries.sonarsource.com/Distribution/sonar-scanner-cli/son
ar-scanner-cli-4.2.0.1873-linux.zip
          echo "Download completed."
       echo "Unziping downloaded file..."
       unzip sonar-scanner-cli-4.2.0.1873-linux.zip
```

```
        echo "Unzip completed."
        rm sonar-scanner-cli-4.2.0.1873-linux.zip
        echo "Installation completed successfully."

        '''

            }

         }
  stage('Run Coverage Test') {

            steps {

                sh 'nosetests -sv --with-xcoverage
--xcoverage-file=/home/jenkins/workspace/POC/Python-pipeline/covera
ge.xml'

            }

         }

  stage('Run Sonar Analysis') {
   steps {
    withSonarQubeEnv('Sonar') {
     sh "sonar-scanner-4.2.0.1873-linux/bin/sonar-scanner"
   }

            }
  }
```

```
    }
}
```

3. Go back to Jenkins, log in again if necessary and click **create new jobs** under **Welcome to Jenkins!**
   Note: If you don't see this, click **New Item** at the top left.
4. In the **Enter an item name** field, specify the name for your new Pipeline project.
5. Scroll down and click **Pipeline**, then click **OK** at the end of the page.
6. (*Optional*) On the next page, specify a brief description for your Pipeline in the **Description** field
7. Click the **Pipeline** tab at the top of the page to scroll down to the **Pipeline** section.
8. From the **Definition** field, choose the **Pipeline script from SCM** option. This option instructs Jenkins to obtain your Pipeline from Source Control Management (SCM).
9. From the **SCM** field, choose **Git**.
10. In the **Repository URL** field, specify the URL for your python project.
11. Click **Save** to save your new Pipeline project.

Note : We are using a docker slave for execution of our pipeline. The DockerFIle for the slave is as follows,

DockerFile

```
FROM jenkinsci/slave:3.35-3
USER root
RUN apt-get update && \
   apt-get install --no-install-recommends -y \
   python-pip \
   python-dev \
   pylint \
   && pip install --upgrade setuptools \
   && pip install wheel \
   && pip install xmlrunner \
   && pip install pyinstaller \
   && pip install nose coverage nosexcover \
   && rm -rf /tmp/downloaded_packages/ /tmp/*.rds \
   && rm -rf /var/lib/apt/lists/*
USER jenkins
```

3. Testing

1. We are using python in-built unittest library for performing unit test on our code.
2. The `unittest` module provides a rich set of tools for constructing and running tests.

Unit-test Example

```
import unittest
import sys
sys.path.append('/home/jenkins/workspace/POC/Python-pipeline/source
s')
import calc
```

```
import xmlrunner

class TestCalc(unittest.TestCase):
    """
    Test the add function from the calc library
    """

    def test_add_integers(self):
        """
        Test that the addition of two integers returns the correct
total
        """
        result = calc.add2(1, 2)
        self.assertEqual(result, 3)

    def test_add_floats(self):
        """
        Test that the addition of two floats returns the correct
result
        """
        result = calc.add2('10.5', 2)
        self.assertEqual(result, 12.5)

    def test_add_strings(self):
        """
        Test the addition of two strings returns the two strings as
one
        concatenated string
        """
        result = calc.add2('abc', 'def')
        self.assertEqual(result, 'abcdef')

    def test_add_string_and_integer(self):
        """
        Test the addition of a string and an integer returns them
as one
        concatenated string (in which the integer is converted to a
string)
        """
        result = calc.add2('abc', 3)
        self.assertEqual(result, 'abc3')

    def test_add_string_and_number(self):
        """
        Test the addition of a string and a float returns them as
one
        concatenated string (in which the float is converted to a
string)
        """
        result = calc.add2('abc', '5.5')
```

```
        self.assertEqual(result, 'abc5.5')

if __name__ == '__main__':
    runner = xmlrunner.XMLTestRunner(output='test-reports')
```

```
    unittest.main(testRunner=runner)
```

3. A testcase is created by subclassing `unittest.TestCase`.
4. The five individual tests are defined with methods whose names start with the letters `test`. This naming convention informs the test runner about which methods represent tests.
5. The crux of each test is a call to `assertEqual()` to check for an expected result.
6. We can also use `assertTrue()` or `assertFalse()` to verify a condition; or `assertRaises()` to verify that a specific exception gets raised.
7. The final block shows a simple way to run the tests. `unittest.main()` provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
----------------------------------------------------------------------
---
Ran 5 tests in 0.000s

OK
```

8. The `TestCase` class provides several assert methods to check for and report failures. The following table lists the most commonly used methods

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	2.7
`assertIsNot(a, b)`	`a is not b`	2.7
`assertIsNone(x)`	`x is None`	2.7
`assertIsNotNone(x)`	`x is not None`	2.7
`assertIn(a, b)`	`a in b`	2.7
`assertNotIn(a, b)`	`a not in b`	2.7
`assertIsInstance(a, b)`	`isinstance(a, b)`	2.7
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	2.7

4. Packaging

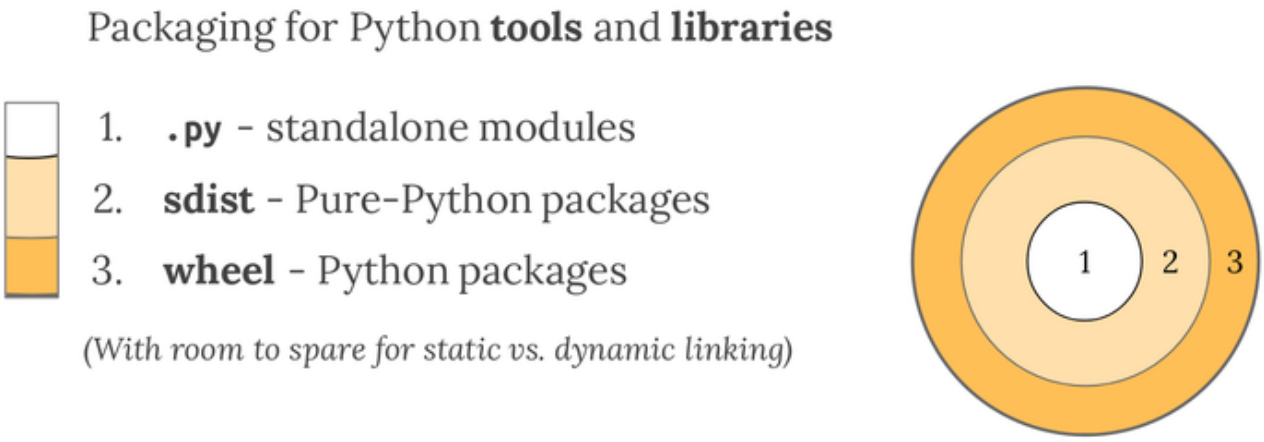
1. You may have heard about PyPI, `setup.py`, and `wheel` files. These are just a few of the tools Python's ecosystem provides for distributing Python code to developers.
2. A Python file, provided it only relies on the standard library, can be redistributed and reused.
3. However, this pattern won't scale for projects that consist of multiple files, need additional libraries, or need a specific version of Python.

Python source distributions

1. As your code contains nothing but pure Python code, and you know your deployment environment supports your version of Python, then you can use Python's native packaging tools to create a *source* **distribution package**, or *sdist* for short.
2. Python's *sdists* are compressed archives (`.tar.gz` files) containing one or more packages or modules.

Python binary distributions

1. So much of Python's practical power comes from its ability to integrate with the software ecosystem, in particular libraries written in C, C++, Fortran, Rust, and other languages.
2. Not all developers have the right tools or experiences to build these components written in these compiled languages, so Python created the wheel, a package format designed to ship libraries with compiled artifacts. In fact, Python's package installer, `pip`, always prefers wheels because installation is always faster, so even pure-Python packages work better with wheels.



Requirements for packaging

1. The most important file is `setup.py` which exists at the root of your project directory.It's the file where various aspects of your project are configured. The primary feature of `setup.py` is that it contains a global `setup()` function. The keyword arguments to this function are how specific details of your project are defined.It's the command line interface for running various commands that relate to packaging tasks. To get a listing of available commands, run

```
python setup.py --help-commands
```

2. All projects should contain a readme file that covers the goal of the project.
3. A `MANIFEST.in` is needed when you need to package additional files that are not automatically included in a source distribution.
4. Every package should include a license file detailing the terms of distribution. In many jurisdictions, packages without an explicit license can not be legally used or distributed by anyone other than the copyright holder.
5. Although it's not required, the most common practice is to include your Python modules and packages under a single top-level package that has the same name as your project, or something very close.

Choosing a Versioning Scheme

1. Different Python projects may use different versioning schemes based on the needs of that particular project, but all of them are required to comply with the flexible **public version scheme** specified in **PEP 440** in order to be supported in tools and libraries like `pip` and `setuptools`.
2. Here are some examples of compliant version numbers:

```
1.2.0.dev1  # Development release
1.2.0a1     # Alpha Release
1.2.0b1     # Beta Release
1.2.0rc1    # Release Candidate
1.2.0       # Final Release
1.2.0.post1 # Post Release
15.10       # Date based release
23          # Serial release
```

3. **Semantic**: For new projects, the recommended versioning scheme is based on Semantic Versioning, but adopts a different approach to handling pre-releases and build metadata.The essence of semantic versioning is a 3-part MAJOR.MINOR.MAINTENANCE numbering scheme, where the project author increments:
    a. MAJOR version when they make incompatible API changes,
    b. MINOR version when they add functionality in a backwards-compatible manner, and
    c. MAINTENANCE version when they make backwards-compatible bug fixes.
4. **Date Based**: Semantic versioning is not a suitable choice for all projects, such as those with a regular time based release cadence and a deprecation process that provides warnings for a number of releases prior to removal of a feature.A key advantage of date based versioning is that it is straightforward to tell how old the base feature set of a particular release is given just the version number.Version numbers for date based projects typically take the form of YEAR.MONTH (for example, `12.04`, `15.10`).
5. **Serial**: This is the simplest possible versioning scheme, and consists of a single number which is incremented every release.While serial versioning is very easy to manage as a developer, it is the hardest to track as an end user, as serial version numbers convey little or no information regarding API backwards compatibility.
6. **Hybrid:** Combinations of the above schemes are possible. For example, a project may combine date based versioning with serial versioning to create a YEAR.SERIAL numbering scheme that readily conveys the approximate age of a release, but doesn't otherwise commit to a particular release cadence within the year.

Packaging Your Project

1. To have your project installable from a Package Index like PyPI, you'll need to create a Distribution (aka "Package") for your project.
2. Minimally, you should create a Source Distribution

```
python setup.py sdist
```

   A "source distribution" is unbuilt (i.e. it's not a Built Distribution), and requires a build step when installed by pip. Even if the distribution is pure Python (i.e. contains no extensions), it still involves a build step to build out the installation metadata from `setup.py`.
3. You should also create a wheel for your project. A wheel is a built package that can be installed without needing to go through the "build" process. Installing wheels is substantially faster for the end user than installing from a source distribution.
4. If your project is pure Python (i.e. contains no compiled extensions) and natively supports both Python 2 and 3, then you'll be creating what's called a **Universal Wheel.**

```
python setup.py bdist --universal
```

5. If your project is pure Python but does not natively support both Python 2 and 3, then you'll be creating a **Pure Python Wheel.**

```
python setup.py bdist
```

   bdist will detect that the code is pure Python, and build a wheel that's named such that it's usable on any Python installation with the same major version (Python 2 or Python 3) as the version you used to build the wheel.
6. If your project contains compiled extensions, then you'll be creating what's called a **Platform Wheel.**

```
python setup.py bdist
```

   bdist will detect that the code is not pure Python, and build a wheel that's named such that it's only usable on the platform that it was built on
7. Before you can build wheels for your project, you'll need to install the `wheel`package:

```
pip install wheel
```

5. Publishing Artifacts to JFrog and Nexus

1. For nexus-jenkins integration, please follow How To - Nexus Jenkins Integration
2. For Jfrog-jenkins integration, please follow Jfrog Artifactory- Jenkins Integration
3. To publsih artifacts to nexus, please include the following stage in your jenkins file.

```
stage("Publsih to Nexus"){

   steps {

    nexusArtifactUploader artifacts: [[artifactId: 'com.python.NV',
classifier: '', file:
'/home/jenkins/workspace/POC/Python-pipeline/dist/py-jenkins-0.1.0.
linux-x86_64.tar.gz',
    type: 'tar.gz']], credentialsId: 'nexus', groupId:
'com.python.NV', nexusUrl: 'novartis.devops.altimetrik.io:8082',
nexusVersion: 'nexus3', protocol:
    'http', repository: 'Python-Project', version:
'{$BUILD_NUMBER}'

   }
        }
```

4. To publsih artifacts to Jforg, please include the following stage in your jenkins file.

```
stage("Publsih to Jfrog"){

   steps {

    rtUpload (
       serverId: 'jfrog',
       spec: '''{
          "files": [
         {
          "pattern":
"/home/jenkins/workspace/POC/Python-pipeline/dist/py-jenkins-0.1.0.
linux-x86_64.tar.gz",
          "target": "python/"
         }
           ]
         }'''
      )

   }
        }
```

6. Sonar-Integration for Code Coverage and Code Smell

1. SonarQube-Jenkins integration should be done. For information regarding same, please refer https://altimetrik.atlassian.net/wiki/spaces/DC/pages/210043079/How+To+-Jenkins-SonarQube+Integration .
2. Create a **sonar-project.properties** file with the following content in the root of your project.

Sonar Properties file

```
sonar-project.properties
sonar.projectKey=py123
sonar.projectName=Python-pipeline
sonar.projectVersion=0.1.0
sonar.sources=/home/jenkins/workspace/POC/Python-pipeline/sources
sonar.language=py
sonar.python.xunit.reportPath=/home/jenkins/workspace/POC/Python-pipeline/test-reports/*.xml
sonar.python.coverage.reportPath=/home/jenkins/workspace/POC/Python-pipeline/coverage.xml
```

3. (Optional)Install sonar-scanner utility if using a docker slave, with the following commands

```
wget -q
https://binaries.sonarsource.com/Distribution/sonar-scanner-cli/sonar-scanner-cli-4.2.0.1873-linux.zip
unzip sonar-scanner-cli-4.2.0.1873-linux.zip
rm sonar-scanner-cli-4.2.0.1873-linux.zip
```

4. For Code Coverage we are using nosetest python utility. Example,

```
nosetests -sv --with-xcoverage
--xcoverage-file=/home/jenkins/workspace/POC/Python-pipeline/coverage.xml
```

5. We have passed junit test report and nosetests coverage report to sonar using properties file.
6. Run the sonar-scanner using the following code in your jenkins file.

```
withSonarQubeEnv('Sonar') {
     sh "sonar-scanner-4.2.0.1873-linux/bin/sonar-scanner"
   }
```