Installing Python interpreters

Apache Beam supports multiple Python versions. You might be able to iterate on the Beam code using one Python version provided by your OS, assuming this version is also supported by Beam. However, you will need to have interpreters for all supported versions to be able to run test suites locally using Gradle, and to work on Beam releases. Therefore, we recommend installing a Python interpreter for each supported version or launching a docker-based development environment that should have these interpreters preinstalled using: `start-build-env.sh`.

There are several ways how you might install multiple Python versions.

- You can download, build and install CPython from sources.
- If you are an Ubuntu user, you could add a third-party repository 'Deadsnakes' and install the missing versions via apt. If you install from Deadsnakes, make sure to also install `python#.#-dev`, `python#.#-venv` and `python#.#-distutils` packages.
- You can use PyEnv to download and install Python versions (Recommended).

Installation steps may look as follows:
1. Follow the steps below in How to setup pyenv.
2. Install Python interpreter for each supported Python minor version. For example:

```bash
pyenv install 3.6.13
pyenv install 3.7.10
pyenv install 3.8.9
pyenv install 3.9.4
```

For major.minor.patch versions currently used by Jenkins cluster, see Current Installations.
3. Make installed interpreters available in your shell by running

```bash
pyenv global 3.8.9 3.6.13 3.7.10 3.9.4
```

After these steps, all `python3.x` interpreters should be available in your shell. The first version in the list passed to `pyenv global` will be used as default `python / python3` interpreter if the minor version is not specified.

Please contribute to these instructions if they didn’t work for you.

Developing the Python SDK

- Beam Gradle tooling can build and test Python SDK, and Jenkins jobs use it, so it needs to be maintained.
- You can directly use the Python toolchain instead of having Gradle orchestrate it, which may be faster for you, but it is your preference. If you want to use Python tools directly, we recommend setting up a virtual environment before testing your code.
- The commands below assume that you’re in the SDKs/python directory.

Virtual Environment Setup

Setting up a virtual environment is required for running tests directly, such as via `pytest` or an IDE like PyCharm. To create an environment, install Python SDK from the sources with `test` and GCP dependencies, follow the below instructions:
On macOS/Linux

1. Use the following code:

```bash
# Initialize virtual environment called "env" in ~/.virtualenvs or any other directory. (Consider using pyenv, to manage the python version as well as installed packages in your virtual environment)
$ python3 -m venv ~/.virtualenvs/env

# Activate virtual environment.
$ . ~/.virtualenvs/env/bin/activate

# Upgrade other tools. (Optional)
pip install --upgrade pip
pip install --upgrade setuptools

# Install setup.py requirements.
(env) $ pip install -r build-requirements.txt

# Install Apache Beam package in editable mode.
(env) $ pip install -e .[gcp,test]
```

On Windows

1. Use the following code:

```bash
> c:\Python37\python.exe -m venv c:\path\to\env
> c:\path\to\env\Scripts\activate.bat
# Powershell users should run instead:
> c:\path\to\env\Scripts\activate.ps1

(env) > pip install -e .[gcp,test]
```

2. You can deactivate the virtualenv when done.

```bash
(env) $ deactivate
```

Virtual Environments with pyenv

- A more advanced option, pyenv allows you to download, build, and install locally any version of Python, regardless of which versions your distribution supports.
- pyenv also has a virtualenv plugin, which manages the creation and activation of virtualenvs.
- The caveat is that you'll have to take care of any build dependencies, and those are probably still constrained by your distribution.
- These instructions were made on a Linux/Debian system.

How to setup pyenv (with pyenv-virtualenv plugin)

1. Install prerequisites for your distribution.
2. curl https://pyenv.run | bash
3. Add the required lines to ~/.bashrc (as returned by the script).
4. Note (12/10/2021): You may have to manually modify .bashrc as described here: https://github.com/pyenv/pyenv-installer/issues/112#issuecomment-971964711. Remove this note if no longer applicable.
5. Open a new shell. If pyenv command is still not available in PATH, you may need to restart the login session.

Example on Ubuntu:
Example: How to Run Unit Tests with PyCharm Using Python 3.7.9 in a virtualenv

1. Install Python 3.7.9 and create a virtualenv
   - pyenv install 3.7.9
   - pyenv virtualenv 3.7.9 ENV_NAME
   - pyenv activate ENV_NAME
2. Upgrade packages (recommended)
   - pip install --upgrade pip setuptools
3. Set up PyCharm
   a. Start by adding a new project interpreter (from the bottom right or in Settings).
   b. Select Existing environment and the interpreter, which should be under ~/pyenv/versions/3.7.9/envs/ENV_NAME/bin/python or ~/.pyenv/versions/ENV_NAME/bin/python.
   c. Switch interpreters at the bottom right.

Cleaning up environments

To delete all environments created with pyenv, run:

```
pyenv virtualenvs --bare --skip-aliases | xargs -n 1 pyenv virtualenv-delete -f
```

Troubleshooting

If you have issues, find troubleshooting at pyenv common build problems.

Error: No module named distutils. (23/07/2021)

As of 23/07/2021, users of some versions of Debian are currently experiencing the error "ModuleNotFoundError: No module named 'distutils.util'" when using the Python Beam SDK. This is typically because a desired version of Python interpreter is no longer available in the distribution. This can be fixed by installing Python via pyenv, rather than relying on the packages included with the Debian distribution. See Installing Python interpreters above.

The error may also manifest in environments created with virtualenv tool even with Python interpreters installed via pyenv. The workaround can be to downgrade to virtualenv==16.1 or use pyenv or venv to create virtual environments. You will also likely need to clean the previously created environment: `rm -rf /path/to/beam/build/gradlenv`

Running Tests

If you update any of the cythonized files in Python SDK, you must first install the cython package, and run the following command before testing your changes

```
python setup.py build_ext --inplace
```

Running Tests Using pytest

If you've set up a virtualenv above, you can now run tests directly using pytest.
Running Integration Tests Using `pytest`

To run an integration test you may need to specify additional parameters for the runner.

Unless you are using Direct runner, you must build the Beam SDK tarball. To do so, run the following commands from the root of the git repository.

```
cd sdks/python/
python setup.py sdist
```

We will use the tarball built by this command in the `--sdk_location` parameter.

It is helpful to emit the test logs to console immediately when they occur. You can do so with the `--log-cli=True` option. You could additionally customize the logging level with the `log_level` option.

Sample invocation:

```
python -m pytest -o log-cli=True -o log-level=Info apache_beam/ml/gcp/cloud_dlp_it_test.py::CloudDLPIT::test_inspection --test-pipeline-options='--runner=TestDataflowRunner --project=<project> --temp-location=gs://<bucket>/tmp --sdk-location=dist/apache-beam-2.35.0.dev0.tar.gz --region=us-central1'
```

Timeouts in Integration Tests

While integration tests running on Jenkins have timeouts that are set with an adequate buffer (4500 secs), tests that are invoked locally via `python -m pytest ...` may encounter timeout failures. This is because the `timeout` property defined in our `pytest.ini` file is set to 600 secs, which may not be enough time for a particular integration test. To get around this, either change the value of `timeout` to a higher number, or add a `pytest` timeout decorator above the function(s) inside your test class.

Example:

```
class PubSubIntegrationTest(unittest.TestCase):
    @pytest.mark.timeout(1200)
    def test_streaming_with_attributes(self):
        # test logic here
```

For more information about timeouts in `pytest`, go to this site.

Running Unit Tests Using `tox`

Here are some tips for running tests using tox:

- Tox does not require a virtualenv with Beam + dependencies installed. It creates its own.
- It also runs tests faster, utilizing multiple processes (via `pytest-xdist`).
- For a list of environments, run `tox -l`.
- `tox` also supports passing arguments after double dashes to `pytest`.

Execute the following code for running tests using tox:

```
(env) $ pip install tox
(env) $ tox -e py38-cloud # all tests
(env) $ tox -e py38 -- -k test_progress
```
Running Tests Using gradle

Integration tests suites on Jenkins are configured in groovy files that launch certain gradle tasks (example). You could launch test suites locally by executing the gradle targets directly (for example: ./gradlew :sdks:python:test-suites:dataflow:py37:postCommitPy37). This option may only be available to committers, as by default the test suites are configured to use apache-beam-testing project.

To run only a subset of tests using this approach, you could adjust the test label in the test (such as it_postcommit) and the selector where the test suite is defined.

Lint and Formatting Checks

Beam codebase enforces consistency of the code style and formatting rules using linters and an autoformatting tool yapf.

- To run all consistency checks, run the following commands:
  ```
  pip install tox
  .././gradlew lint # Runs several linter checks
  tox -e py3-yapf-check # Runs code formatting checks
  ```

- To auto-format the code in place, run:
  ```
  tox -e py3-yapf
  ```

Running lint and yapf Automatically on Each Commit with pre-commit Hooks

The pre-commit tool allows you to run pre-configured checks automatically when you commit changes with `git commit`.

- To enable pre-commit, run:
  ```
  pip install pre-commit
  pre-commit install
  ```

⚠️ When the pre-commit hook for yapf applies formatting changes in place, the check fails with an error if files were modified by this hook, and you have to re-run `git commit`.

- To disable the pre-commit, run:
  ```
  pre-commit uninstall
  ```

Running yapf formatter manually

To run manually:

1. Install yapf.
   ```
   pip install yapf==0.29.0
   ```

⚠️ For consistency, use the current version of yapf in https://github.com/apache/beam/blob/master/sdks/python/tox.ini

2. To format changed files in your branch:
   ```
   # Run from root beam repo dir
   git diff master --name-only | grep "\.py$" | xargs yapf --in-place
   ```
3. To format just a single directory:

```bash
yapf --in-place --parallel --recursive apache_beam/path/to/files
```

4. To format files with uncommitted changes, run:

```bash
git diff --name-only | grep "\.py\$" | xargs yapf --in-place
```

5. If you need to exclude one particular file or pattern from formatting, add it to the `.yapfignore` file (sdks/python/.yapfignore).

---

### Run hello world against modified SDK Harness

To run a hello world against modified SDK Harness, execute the following code:

```bash
# Build the Flink job server (default job server for PortableRunner) that stores the container locally.
./gradlew :runners:flink:1.7:job-server:container:docker

# Build portable SDK Harness, which builds and stores the container locally.
# Build for all python versions
./gradlew :sdks:python:container:buildAll
# Or build for a specific python version, such as py35
./gradlew :sdks:python:container:py35:docker

# Run the pipeline.
python -m apache_beam.examples.wordcount --runner PortableRunner --input <local input file> --output <local output file>
```

---

### Run hello world against modified Dataflow Fn API Runner Harness and SDK Harness

To run a hello world against modified Dataflow Fn API Runner Harness and SDK Harness, execute the following code:

```bash
# Build portable worker
./gradlew :runners:google-cloud-dataflow-java:worker:build -x spotlessJava -x rat -x test
./gradlew :runners:google-cloud-dataflow-java:worker:shadowJar

# Build portable Python SDK harness and publish it to GCP
./gradlew -Pdocker-repository-root=gcr.io/dataflow-build/$USER/beam -p sdks/python/container docker
gcloud docker -- push gcr.io/dataflow-build/$USER/beam/python:latest

# Initialize python
cd sdks/python
virtualenv env
dotenv/bin/activate

# Run pipeline
python -m apache_beam.examples.wordcount --runner DataflowRunner --num_workers 1 --project <gcp_project_name> --output gs://path --temp_location gs://path --worker_harness_container_image gcr.io/dataflow-build/$USER/beam/python:latest --experiment beam_fn_api --sdk_location build/apache-beam-2.12.0.dev0.tar.gz --debug
```

---

### Run Integration Test from IDE (TODO: please revise these instructions now that we migrated to PyTest)
To run an integration test from an IDE in a debug mode, you can create a Nosetests configuration. For example, to run a VR test on Dataflow runner from IntelliJ/PyCharm, you can adjust the configuration as follows:

1. Set Target to Module and point to the test file.
2. Set Additional arguments (sample, adjust as needed):

   ```
   Running a ValidatesRunner test
   ```
   ```
   --test-pipeline-options=--runner=TestDataflowRunner --project=<YOUR PROJECT> --region=us-central1 --temp_location=gs://your_bucket/tmp --sdk_location=.dist/apache-beam--<version>.dev0.tar.gz --requirements_file=./postcommit_requirements.txt --num_workers=1 --sleep_secs=20 --attr=ValidatesRunner1 --nocapture
   ```
3. Set Working directory to /path/to/beam/sdks/python.

Run a screen diff integration Test for Interactive Beam

For Interactive Beam/Notebooks, we need to verify if the visual presentation of executing a notebook is stable. A screen diff integration test that executes a test notebook and compares results with a golden screenshot does the trick. To run a screen diff integration Test for Interactive Beam:

1. Execute the following code for preparation work:

   ```
   Test dependencies
   ```
   ```
   # Install additional Python dependencies if absent, under beam/sdks/python, run:
   pip install -e .[interactive,interactive_test,tests]
   
   The tests use headless chrome to render visual elements, make sure the machine has chrome executable installed.
   If you’re reading this document in a chrome browser, you’re good to go for this step.
   # Otherwise, e.g., on a Linux machine, you might want to do:
   wget --quiet https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb && 
   apt install -y ./google-chrome-stable_current_amd64.deb
   
   As chrome version advances/differs, the chromedriver-binary needs to stay in sync with chrome.
   # As a bash example to dynamically install the correct chromedriver-binary.
   google_chrome_version=$(google-chrome --version)
   chromedriver_lower_version=${google_chrome_version%.*.*.*}
   chromedriver_upper_version=$(($chromedriver_lower_version+1))
   pip install "chromedriver-binary>=${chromedriver_lower_version},<${chromedriver_upper_version}"
   
   # For consistency of screenshots, roboto-mono font should have been installed.
   # You can download the font from https://fonts.google.com/specimen/Roboto-Mono.
   # Otherwise, you can install through CLI, e.g., on Linux:
   ```
2. To run the tests:

   ```
   Running screen diff integration test
   ```
   ```
   # Under beam/sdks/python, run:
   pytest -v apache_beam/runners/interactive/testing/integration/tests
   
   # TL;DR: you can use other test modules, such as nosetests and unittest:
   nosetests apache_beam/runners/interactive/testing/integration/tests
   python -m unittest apache_beam/runners/interactive/testing/integration/tests/init_square_cube_test.py
   
   # To generate new golden screenshots for screen diff comparison:
   nosetests apache_beam/runners/interactive/testing/integration/tests --with-save-baseline
   ```
3. Golden screenshots are temporarily taken and stored by the system platform. The currently supported platforms are Darwin (macOS) and Linux.
4. Each test will generate a stable unique hexadecimal id. The golden screenshots are named after that id.
5. To add new tests, put a new test notebook file (.ipynb) under the apache_beam/runners/interactive/testing/integration /test_notebooks directory.
6. Add a single test under apache_beam/runners/interactive/testing/integration/tests directory. A test is simple as:

```python
from apache_beam.runnersinteractive.testing.integration.screen_diff import BaseTestCase
class SimpleTest(BaseTestCase):

def test_simple_notebook(self):
    self.assert_notebook('simple_notebook')  # Just put the notebook file name here, e.g., 'simple_notebook'.
```

### How to Install an Unreleased Python SDK without Building It

SDK source zip archive and wheels are continuously built after merged commits to https://github.com/apache/beam

1. Click on a recent ‘Build python source distribution and wheels job’ that ran successfully on the github.com/apache/beam master branch from this list.
2. Click on List files on Google Cloud Storage Bucket on the right-side panel.
3. Expand List file on Google Cloud Storage Bucket in the main panel.
4. Locate and Download the ZIP file. For example, apache-beam-2.25.0.dev0.zip from GCS.
   - It’s simplest to download the file using your browser by replacing the prefix “gs://” with “https://storage.googleapis.com/”. For example, https://storage.googleapis.com/beam-wheels-staging/master/02bf081d0e86016395a415cebee2812620aaf4b-207975627/apache-beam-2.25.0.dev0.zip
   - Or follow these instructions to download using the gsutil command-line tool.
5. Install the downloaded zip file. For example:

```bash
pip install apache-beam-2.25.0.dev0.zip
# Or, if you need extra dependencies:
pip install apache-beam-2.25.0.dev0.zip[aws,gcp]
```

6. When you run your Beam pipeline, pass in the `--sdk_location` flag pointed at the same ZIP file.

```bash
SimpleTest
--sdk_location=apache-beam-2.25.0.dev0.zip
```

### How to update dependencies that are installed in Python container images

When we build Python container images for Apache Beam SDK, we install PyPI packages of Apache Beam and some additional PyPI dependencies that will likely benefit users. The complete list of dependencies is specified in `base_image_requirements.txt` files, for each Python minor version. These files are generated from Beam SDK `requirements`, specified in `setup.py`, and a short list of additional dependencies specified in `base_image_requirements_manual.txt`.

We expect all Beam dependencies (including transitive dependencies, and deps for some of the 'extra's, like [gcp]) to be specified with exact versions in the requirements files. Therefore, you may need to regenerate the requirements files when you modify Python SDKs dependencies in `setup.py`.

Regenerate the requirements files by running: `./gradlew :sdks:python:container:generatePythonRequirementsAll` and committing the changes. Execution takes about ~5 min per Python version and is somewhat resource-demanding. You can also regenerate the dependencies individually per version with targets like: `./gradlew :sdks:python:container:py38:generatePythonRequirements`.

To run the command successfully, you will need Python interpreters for all versions supported by Beam. See: Installing Python Interpreters.

### Errors

You may see the following error with a particular python version like Python 3.6.
Task :sdks:python:container:py36:generatePythonRequirements FAILED

+ ENV_PATH=/usr/local/google/home/yeandy/beam/build/python36_requirements_gen
+ rm -rf /usr/local/google/home/yeandy/beam/build/python36_requirements_gen
+ python3.6 -m venv /usr/local/google/home/yeandy/beam/build/python36_requirements_gen

Error: Command 'SPATH=/usr/local/google/home/yeandy/beam/build/python36_requirements_gen/bin/python3.6', '-Im', 'ensurepip', '--upgrade', '--default-pip']' died with <Signals.SIGSEGV: 11>.

FAILURE: Build failed with an exception.

* Where:
Script '/usr/local/google/home/yeandy/beam/sdks/python/container/common.gradle' line: 37

* What went wrong:
Execution failed for task ':sdks:python:container:py36:generatePythonRequirements'.
> Process 'command 'sh'' finished with non-zero exit value 1

You may see that the pip command will lead to segmentation fault as well. If this happens, remove the python version from pyenv, and reinstall the version like this.

CFLAGS="-O2" pyenv install 3.6.12

There have been issues with older Python versions. See here for details.