Shape Measurement Pipeline Template

Author: Samuel Farrens
Email: samuel.farrens@cea.fr
Year: 2017

Contents

Introduction

This template has been set up to reduce the amount of time needed to integrate new software into the shape measurement pipeline (ShapePipe). The template has been built from existing pipeline elements and therefore is currently subject to several hard-wired limitations that may be circumvented in the future.

Please report any bugs or oversights discovered while implementing a new pipeline package using this template in order to improve it for future users.

Note that this template cannot take into account everything that you may wish to implement in the pipeline but will hopefully get you started.

Package Contents

Below is a list of the template package contents:

- `README.txt`: Is generic readme file for any pipeline package. You can modify this if you want to improve the documentation of your package.
- `config`: Is a directory containing the package configuration file and launch script. You will need to modify both.
- `find_package_name.sh`: Is a script set up to help you find references to `package_name`.
- `license.txt`: Is a generic GNU public software license.
- `package_name`: Is a directory with all of the template code, some of which you will need to modify for your package.
- `setup.py`: Is a script used for installing Python modules. You only need to update your package name.
- `template.{html, md, pdf}`: These instructions in html, markdown and PDF format.

Implementation

This section describes the steps you should follow to implement a new pipeline package. By following the steps listed below you should be able to get your code working without having to touch any other part of the pipeline, but only if

more complicated functionality is not required.

1. Create a new branch of the repository.
2. Copy the template and update the package name.
3. Set up the package configuration file (`package_config_smp.cfg`).
4. Set up the input/output files and options for the code to be implemented (`_set_filenames` in `execute.py`).
5. Define the execution line to be run (`_set_exec_line` in `execute.py`).
6. Commit and push your updates to the remote repository.

New Package Branch

The first thing you should do to create your new pipeline package is to clone the ShapePipe GitLab repository to your local machine.

```
git clone https://drf-gitlab.cea.fr/cosmostat/ShapePipe
```

Then create a new branch to work on your package.

```
git checkout -b NEW_BRANCH_NAME
```

Before you continue, make sure you are working on your new branch and **not on the master branch!** You can easy check this as follows:

```
git status
```

The asterisk (`*`) should be next to the name of your new branch.

Update Package Name

Once you have set up your new branch you should make a copy of this template package and give it an appropriate name.

Inside your template copy (**don't modify the orginal template!**) you should replace every reference to `package_name` with the name of your new package. To help you find these references you can run the `find_package_name.sh` script.

```
./find_package_name.sh
```

A simple search and replace in each file will suffice.

Configuration File

The default package configuration file is `package_config_smp.cfg` , which can be found in the `config` directory. This file specifies all of the values required to run the pipeline element (*i.e.* your code). At present some of the parameters listed in this file are hard-wired arguments required by either `mpfg` or `mpfx` . This may change in the future, for now the values you should provide are for the following arguments:

- `BASE_INPUT_DIR` : Specifies the path to the code configuration file (**not the package configuration file!**).
- `BASE_OUTPUT_DIR` : Specifies the path where the code outputs should be saved.
- `BASE_DIR` : Specifies the path where the code input files are located.
- `EXEC_PATH` : Specifies the code executable (with full path if not in system PATH).
- `DEFAULT_FILENAME` : Specifies the name of the code configuration file if needed.
- `INPUT_FILENAME_FORMATS` : Specifies the file name conventions for the input files.
- `OUTPUT_CATALOG_FILE_ENDING:` Specifies the file ending expected for the output files.

Additional arguments can be added to the `[CODE]` section if necessary. For example:

- `EXTRA_CODE_OPTION` : Specifies an additional command line option for running the code.

The default arguments are set up to run the Unix command `diff` . Note that the line `INPUT_FILENAME_FORMATS = ['file1.txt', 'file2.txt']` specifies that for every run of `diff` the pipeline will expect two input files with the naming conventions `file1.txt` and `file2.txt` . `mpfx` retains a hard-wired requirement that all input names use a file naming system with pattern `xxx-x` , this may be relaxed in the future. Therefore, for the default example, the code input directory (*i.e.* `BASE_DIR`) should contain files such as:

- `file1-000-0.txt` , `file2-000-0.txt` ,
- `file1-001-0.txt` , `file2-001-0.txt` ,
- `file1-002-0.txt` , `file2-002-0.txt`
- *etc.*

Set File Names

In order to implement your code in the pipeline you will need to specify the input and output file names. Additionally, if your code uses a configuration file it should also be specified here.

In the file `execute.py` within the class `PackageRunner` there is a private method called `_set_filenames` which defines a dictionary called `self._fnames` . The default keywords for this dictionary are:

- `'input_filepath'` : Specifies the input filename(s) based on the input file types defined in the package configuration file.
- `'config_filepath'` : Specifies the input configuration file name (if needed).
- `'output_filepath'` : Sets the output file name for the code.
- `'output_filepath'` : Sets the expected output filename from the code. This option is only required if your code has multiple outputs.
- `'extra_option'` : Specifies an additional option to be added to the command line.

Note that you can provide as many additional dictionary keys to `self._fnames` as needed for your code.

The default file names in the template are set up to run the Unix command `diff` . Note that the `self._fnames['config_filepath']` option is commented out as this code does not use a configuration file.

```
  def _set_filenames(self):

    # --- Input files to be read
    self._fnames['input_filepath'] = [self._job.get_file_path(file_type)
                                      for file_type in self.file_types]

    input_filename = (os.path.splitext(os.path.split(
                      self._fnames['input_filepath'][0])[1])[0])

    # --- Executable configuration file
    # !!! Uncomment this line if your code uses a config file !!!
    # self._fnames['config_filepath'] = self._get_exec_config_filepath()

    # --- Target directory where to store files
    output_path = os.path.join(self._worker.result_output_dir,
                               self._job.get_branch_tree())

    # --- Outpu file name
    self._fnames['output_filepath'] = os.path.join(output_path,
                                                   input_filename)

    # --- Expected output catalog file name
    output_cat_filename_exp = (self._get_output_catalog_filename(
                               input_filename))
```

```
    # --- Output catalog file path
    self._fnames['output_filepath_exp'] = (os.path.abspath(os.path.join(
                                           output_path,
                                           output_cat_filename_exp)))

    self._log_exp_output(self._fnames['output_filepath_exp'])

    # -- Load an extra code option from config file
    self._fnames['extra_option'] = (self._worker.config.get_as_string(
                                    'EXTRA_CODE_OPTION', 'CODE'))
```

To implement you new code uncomment the `self._fnames['config_filepath']` option if you code uses a configuration file. Then, add/remove keywords to/from `self._fnames` if necessary. Finally, make sure the package configuration file is properly set up.

Set Execution Line

Once your input and output file names have been specified you need only define the execution line used to run your code

In the file `execute.py` within the class `PackageRunner` there is a private method called `_set_exec_line` which defines the execution line.

The default execution line in the template is set up to run the Unix command `diff`.

```
  def _set_exec_line(self):

    # --- Execution line
    exec_path = self._worker.config.get_as_string('EXEC_PATH', 'CODE')

    self._exec_line = ('{0} {1} {2} {3} > {4}_diff.txt').format(
                       exec_path,
                       self._fnames['input_filepath'][0],
                       self._fnames['input_filepath'][1],
                       self._fnames['extra_option'],
                       self._fnames['output_filepath'])

    self._log_exec_line()
```

The `exec_path` string reads full path to the executable defined in the package configuration file, while `self.exec_line` defines a string with the execution line as it would be run in a command line.

In this example, using the defaults provided in the package configuration file, the execution line would be:

```
diff /home/user/template/input/file1-000-0.txt /home/user/template/input/file2-000-0.txt -y > /home/user/template/output/file1-000-0_diff.txt
```

To implement your new code, simply edit the string inside the `()` and provide the necessary keywords to `self._fnames` for the input/output files and code options.

Commit and Push Changes

After creating your new pipeline package you should upload your modifications to the remote repository (*i.e.* GitLab).

```
git add .
git commit -m "message describing your changes"
git push
```

I recommend you make regular commits during the development of your pipeline package to avoid losing work and to

help avoid potential conflicts.

Execution

The pipeline package can be run (locally or on the cluster) using the `launch.cmd` file provided in the `config` directory.

```
# !/bin/bash

# Set path to package module
export PACKAGE_DIR="$HOME/ShapePipe/modules/template_package"

# Run package
python ${PACKAGE_DIR}/package_name/package_name_SMP.py -d ${PACKAGE_DIR}/config -c
package_config_smp.cfg
```

You simply need to change `package_name` to the name of your package and then run:

```
./launch.cmd
```