reference manual and user guide for the slacx image data workflow manager

L. A. Pellouchoud

October 3, 2016

0.1 Introduction

The `slacx` software package provides a fast and lean platform for processing image-like data. It is being developed to perform analysis of x-ray diffraction patterns for ongoing projects at SLAC/SSRL. At the core of `slacx` is a non-graphical workflow engine meant for batch processing based on instructions from a text input file. On top of that engine is a graphical user interface for users to develop and debug their workflows. Other features include:

- A plugin module for using slacx and its components in the `xi-cam` software package
- An operation development interface for users with little or no programming experience to write their own processing routines
- A network client for directing remote (distributed) computations and communicating with remote filesystems to store and retrieve data

Some long-term goals of `slacx` are:

- to streamline data analysis and standardize storage to make data easily accessible for future reference
- eliminate the development of redundant analysis routines, reducing the potential for bugs and moving towards highly refined and efficient analysis routines
- to interface with experimental equipment to enable model-driven feedback targeting specific engineering objectives

The `slacx` developers would love to hear from you if you have wisdom, thoughts, haikus, bugs, artwork, suggestions, or limericks. Get in touch with us at `slacx-developers@slac.stanford.edu`.

Chapter 1

installation

Successful building of `slacx` depends on a few trusty, stable packages. Developers should take care to favor a stable and cross-platform utility over a fancier option, in order to keep installation of `slacx` fast and easy.

1.1 system dependencies

The packages described in this section may already be prepared on your operating system. If you have already used `pip` to install packages before, you can probably skip this section entirely and move on to section 1.1.4.

1.1.1 pip, setuptools, and wheel

`pip` is a Python package manager (acronym: `pip` Installs Python). There are other ways to install packages, but for now this documentation will cover the usage of `pip`. `setuptools` is a Python library that supports many of the operations performed in this section. `wheel` is a package distribution and installation library supporting the `wheel` distribution standard. Users will typically want to install these together.

`pip`, `setuptools`, and `wheel` are installed by default on many Linux distributions and OSX, but should be upgraded. Enterprise Linux (e.g. RHEL, CentOS, SL, OL) are the exception. Enterprise Linux users should skip ahead to section 1.1.3. For other Linux distributions (Fedora, Ubuntu, etc.) it is suggested that the user employ the system's package manager (skip ahead to section 1.1.2), though the user may choose instead to use `pip` to upgrade itself, as shown below.

Perform the upgrade by typing:

```
pip install -U pip setuptools
```

1.1.2 pip, wheel, and setuptools for non-Enterprise Linux

The instructions differ for the various distributions. The commands listed here are copied from the web at `https://packaging.python.org/`. Enter the commands in a terminal, as listed for your particular distribution. If your distribution is not covered here, please consider adding its instructions to the docs and submitting a pull request.

- Fedora 21:

```
sudo yum upgrade python-setuptools
sudo yum install python-pip python-wheel
```

- Fedora 22:

```
sudo dnf upgrade python-setuptools
sudo dnf install python-pip python-wheel
```

- Debian/Ubuntu:

```
sudo apt-get install python-setuptools python-pip python-wheel
```

1.1.3 pip, wheel, and setuptools for Enterprise Linux

tested on RHEL 6.8 (Santiago)

adding the EPEL repository

NOTE: You may already have connection to the `EPEL` repository. You can tell if you have `EPEL` connected by entering:

```
sudo yum repolist all
```

Look through the list, under the `repo name` heading, for a repo named `EPEL`. If you have it, skip ahead to section 1.1.3.

The easiest way to maintain `pip`, `setuptools`, and `wheel` is to enable the `EPEL` repository (Extra Packages for Enterprise Linux) for your particular distribution. The `EPEL` SIG (Special Interest Group) maintains `rpm` installers that add `EPEL` to RHEL's `yum` repository list. Begin by downloading this installer and running it with `rpm`.

These instructions assume you are using RHEL6, where the `EPEL` installer will be `epel-release-latest-6.noarch.rpm`. For other Enterprise Linux distributions, check the web at `https://fedoraproject.org/wiki/EPEL` for the correct `EPEL` installer file, and replace `epel-release-latest-6.noarch.rpm` in the following instructions with the appropriate filename.

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-6.noarch.rpm
rpm -i epel-release-latest-6.noarch.rpm
```

These commands should enable `EPEL`.

installing pip from EPEL

`pip` is one (important) package that can be obtained from `EPEL`. In many cases, `pip` can be obtained simply by running:

```
sudo yum install python-pip
```

In some cases, the `python-pip` package will not be found. One possibility is that the package will have a different name in the user's specific set of repositories. The next thing to try is:

```
sudo yum install python27-python-pip
```

If this doesn't work, the user must find a package that provides `pip`. Try the following:

```
sudo yum whatprovides *python-pip*
```

This may return a package name like
`python27-python-pip-1.5.6-5.el6.noarch`. Try directly installing that package:

```
sudo yum install python27-python-pip-1.5.6-5.el6.noarch
```

Once `pip` has been installed, it should be used to install or upgrade `wheel` and `setuptools`. With a friendly configuration, the following will work:

```
sudo pip install --upgrade setuptools
sudo pip install --upgrade wheel
```

If these throw errors, for example,

```
error while loading shared libraries: libpython2.7.so.1.0: cannot open shared
     object file: No such file or directory
```

the user will have to locate that shared library and add its location to their `LD_LIBRARY_PATH` environment variable. One way to locate the library is to ask `yum` to find it:

```
yum whatprovides *libpython2.7*
```

may produce something like:

```
python27-python-libs-ver.rel.arch : Runtime libraries for Python
```

```
Repo       : installed
Matched from:
Filename   : /path/to/libpython2.7.so.1.0
```

The search will also probably bring up some development and debugging libraries:

```
python27-python-devel-2.7.8-3.el6.x86_64 : The libraries and header files needed
     for Python development
Repo       : <repo-name>
Matched from:
Filename   : /opt/rh/python27/root/usr/lib64/libpython2.7.so
Filename   : /opt/rh/python27/root/usr/lib64/python2.7/config/libpython2.7.so
```

and:

```
python27-python-debug-2.7.8-3.el6.x86_64 : Debug version of the Python runtime
Repo       : <repo-name>
Matched from:
Filename   : /opt/rh/python27/root/usr/lib64/libpython2.7_d.so
Filename   : /opt/rh/python27/root/usr/lib64/libpython2.7_d.so.1.0
Filename   :
     /opt/rh/python27/root/usr/share/systemtap/tapset/libpython2.7-debug-64.stp
Other      : libpython2.7_d.so.1.0()(64bit)
```

The package that needs to be installed in each case (to get the required library) is named in the top line of each block. You might as well install any of these that are not already installed (note that `python27-python-libs-ver.rel.arch` is already marked as installed in this case). Install the debug and development libraries if necessary:

```
sudo yum install python27-python-debug-2.7.8-3.el6.x86_64
sudo yum install python27-python-devel-2.7.8-3.el6.x86_64
```

The path of the shared object is the line following the `Filename:` label. Take that line and place it in your `LD_LIBRARY_PATH` by adding to your shell configuration file, e.g. for `~/.bashrc`:

```
echo ‘export LD_LIBRARY_PATH=/dir/with/libpython/:$LD_LIBRARY_PATH’ >> ~/.bashrc
source ~/.bashrc
```

With this done, try again to install/upgrade `setuptools` and `wheel`:

```
sudo pip install --upgrade setuptools
sudo pip install --upgrade wheel
```

If this doesn't work, try setting up `sudo` to preserve your `LD_LIBRARY_PATH`:

```
echo "alias ldsudo=‘sudo LD_LIBRARY_PATH=$LD_LIBRARY_PATH’" >> ~/.bashrc
export ~/.bashrc
```

Now, try again in install/upgrade `setuptools` and `wheel`, but this time use `ldsudo`. If it doesn't work, ask your sysadmin for help. Once these are done, some additional steps are needed to prepare the system for `Qt` applications. Read on, section 1.1.3

preparing PySide Qt on Linux

Since `slacx` is built on the `Qt` platform (via `PySide` bindings), a few more system libraries are needed. `Qt` itself is written in C++ and uses C extensions, so C++ and C compilers will be needed. These compilers will be used with the cross-platform `cmake` utility, and `Qt` makes extensive use of the `qmake` tool. This section guides you through the installation of these libraries.

1. Check for existing C and C++ compilers:

   ```
   which cc
   which c++
   ```

   If the system has no `cc` or `c++`, find them with

   ```
   yum whatprovides *bin/cc
   yum whatprovides *bin/c++
   ```

   A list of compiler and developer suites, for example various `devtoolset` versions, will come up. Take a minute to find the latest and most general version. Take note of the package name and the directory containing the binaries. Install the package and add the directory for the compiler binaries to your path.

   ```
   sudo yum install devtoolset-<latest-versoin>-gcc.x86_64
   echo 'export PATH=/opt/rh/devtoolset-4/root/usr/bin/:$PATH' >> ~/.bashrc
   ```

2. Repeat this process for `qmake`, which will come with other `Qt` utilities.

   ```
   yum whatprovides *bin/qmake
   sudo yum install qt-devel-4.6.2-28.el6_5.x86_64
   echo 'export PATH=/usr/lib64/qt4/bin/:$PATH' >> ~/.bashrc
   ```

3. Repeat this process for `cmake`. For `cmake`, no `$PATH` modifications should be needed.

   ```
   yum whatprovides *bin/cmake
   sudo yum install cmake-2.8.12.2-4.el6.x86_64
   ```

1.1.4 setting up a virtual environment with virtualenv

At this point, it is assumed the user has working installations of `pip`, `wheel`, and `setuptools`. This will be the default case for many systems.

Users familiar with `virtualenv` may choose to skip this section. Building software in a virtual environment is suggested as a way of sandboxing libraries that are used by other system components. After setting up `pip`, `setuptools`, and `wheel` (above), the installation and setup of a virtual environment is easy.

Begin by installing `virtualenv`:

```
sudo pip install virtualenv
```

Now, assuming you want to store the `slacx` virtual environment in directory `<dir>`, execute the following in a terminal

```
mkdir <dir>
virtualenv <dir>
source <dir>/bin/activate
# To terminate virtualenv session:
deactivate
```

With the virtual environment active, Python packages can be installed without `sudo` and will only affect the package space of the virtual environment. Any packages installed in the virtual environment are effectively invisible once the virtual environment is turned off by calling `deactivate`.

1.1.5 installing python dependencies

After installing the system packages and starting a virtual environment (above), the installation of the necessary Python packages should be easy. Ensure the virtual environment is active (you should see the name of the virtual environment in parentheses to the left of the command prompt), then install each of these packages by `pip install <packagename>`. Some of these may take a few minutes to install (`scipy` for example). Be patient, make some coffee, `pip` will probably eventually finish.

NOTE: If you are installing `slacx` from a wheel-compatible distribution, the dependencies will be handled automatically, so there is no need to do this.

- `numpy`: Python numerical computation library
- `scipy`: Python scientific math library
- `PySide`: Python `Qt` bindings library: Linux distributions may first need to see section 1.1.3 above.
- `fabio`: Fable I/O library, parses image files from common CCD array detectors devices
- `PyQtGraph`: Scientific visualization library built on PyQt4/PySide

- `matplotlib`: Highly versatile plotting library
- `lxml`: XML and HTML parsing library
- `QDarkStyle`: A dark stylesheet for the Qt GUI- your eyes will be happier
- `pillow`: A replacement for the outdated Python Imaging Library (PIL)

Chapter 2

development

This chapter covers the best practices for contributing to `slacx` via git (section 2.1) and instructions for developing operations for use in the `slacx` workflow engine (section 2.2). These instructions are intended for people who are familiar with the 'slacx' development team and have received an invitation to work in their private repository. Public forking of the repository is not currently allowed.

2.1 contributing to slacx

The `slacx` repository is currently private. Request access by getting in touch with the `slacx` developers. Try `slacx-developers@slac.stanford.edu`. Code development is currently under the "shared repository model", meaning developers have write access to the main repository. In this model, you must be careful with your edits. In particular, **all coding should be done in feature branches**. Never push changes directly to the 'dev' or 'master' branches. If you do, we will roll back to before the changes took place and initiate the process outlined below for pulling the new feature into 'dev'.

The repository has a 'master' branch that is intended to be the most stable version, and a 'dev' branch that is intended for development. The rest of the branches are feature branches. Once you have access, clone the repository, check out the 'dev' branch, and create your own development or feature branch. When you are satisfied, pull in the latest from 'dev' and merge your feature branch with it, and submit a pull request. Here are step-by-step instructions for using 'git' from the command line:

- Change to the directory where you want to do your development work: 'cd mydevdir'
- Clone the repository: 'git clone https://lensonp@bitbucket.org/smashml/slacx.git'. This should automatically set the cloned repository as the remote 'origin'. To make sure, you can use 'git remote -v'.

- Create your own feature branch: ‘git branch my-feature-branch‘

  — **start here every time you work on your feature** —

- Check out the ‘dev‘ branch: ‘git checkout dev‘
- Download the latest from origin: ‘git fetch origin‘
- Merge the latest from the ‘dev‘ branch: ‘git merge origin/dev‘
- Check out your feature branch: ‘git checkout my-feature-branch‘
- Rebase on the ‘dev‘ branch: ‘git rebase dev‘ If another developer made changes that affect your code, you will be able to roll them in to your code base more easily if this is done every time you work on your feature.
- After coding, stage your edits: ‘git add .‘
- After staging your edits, commit them with a description: ‘git commit -m ’description of edits’‘
- Push your commits to origin: ‘git push origin my-feature-branch‘

  — **start here when you are done with your feature** —

- Rebase on the ‘dev‘ branch: ‘git rebase dev‘
- Submit a pull request via the online interface. Request to pull ‘my-feature-branch‘ into ‘dev‘. The ‘slacx‘ development team will discuss, edit, and ultimately accept your feature.

2.2 developing operations for slacx

Developing operations for use in the `slacx` workflow engine is meant to be as painless as possible. A continuing effort will be made to streamline this process.

The most general description of an operation is to treat it as a black box that takes some inputs (data and parameters), performs some processing with those inputs, and produces some output (data and other features). Users can choose to build their operations in two ways.

One option is to write the function into a python class that defines names for its inputs and outputs and then stores the input and output data within instances (objects) of that class. This process is most easily understood by following the template and instructions provided in section 2.2.1 below.

Another option (not yet implemented) is to write the function into a python module that contains formatted metadata about how the function should be called, with input and output data stored in the workflow engine itself.

In either case, the user/developer writes their operation into a module `somefile.py` and then places this module in the `slacx/core/operations/` directory. If the implementation obeys the specified format, the operation will automatically be made available to the `slacx` workflow manager the next time `slacx` is started.

If the user/developer is not careful about implementing their operation, `slacx` will probably raise exceptions and exit. This could happen at startup (when the operation is read), during workflow management (when the operations is loaded with inputs), or during execution (when the operation is called upon to compute things), depending on where (if any) errors were made.

2.2.1 operations as python classes

The following code block gives a minimal, commented template for developing operations as python classes. Users/developers with some programming background may find all the information they need in this template alone.

```
# Users and developers should remove all comments from this template.
# All text outside comments that is meant to be removed or replaced
# is <written within angle brackets>.

# Operations implemented as python classes
# have a common interface for communicating
# with the slacx workflow manager.
# That common interface is ensured by inheriting it
# from an abstract class called 'Operation'.
from core.operations.slacxop import Operation

# Name the operation, specify inheritance (Operation)
class <OperationName>(Operation):
    # Give a brief description of the operation
    # bracketed by """triple-double-quotes"""
    """<Description of Operation>"""

    # Write an __init__() function for the Operation.
    def __init__(self):
        # Name the input and output data/parameters for your operation.
        # Format names as 'single_quotes_without_spaces'.
        input_names = ['<input_name_1>','<input_name_2>',<...>]
        output_names = ['<output_name_1>','<output_name_2>',<...>]
        # Call the __init__ method of the Operation abstract (super)class.
        # This instantiates {key:value} dictionaries of inputs and outputs,
        # which have keys generated from input_names and output_names.
        # All values in the dictionary are initialized as None.
        super(Identity,self).__init__(input_names,output_names)
        # Write a free-form documentation string describing each item
        # that was named in input_names and output_names.
        self.input_doc['<input_name_1>'] = '<expectations for input 1>'
        self.input_doc['<input_name_2>'] = '<etc>'
        self.output_doc['<output_name_1>'] = '<form of output 1>'
        self.output_doc['<output_name_2>'] = '<etc>'
        # Categorize the operation. Multiple categories are acceptable.
        # Indicate subcategories with a '.' character.
        self.categories = ['<CAT1>','<CAT2>.<SUBCAT1>',<...>]

    # Write a run() function for this Operation.
    def run(self):
        # Optional- create references in the local namespace for cleaner code.
        <inp1> = self.inputs['<input_name_1>']
        <inp2> = self.inputs['<input_name_2>']
```

```
<etc>
# Perform the computation
< ... >
# Save the outputs
self.outputs['<output_name_1>'] = <computed_value_1>
self.outputs['<output_name_2>'] = <etc>
```

2.2.2 operations as functions in a module

This functionality is not yet implemented.