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

L. A. Pellouchoud

September 20, 2016

## 0.1 Introduction

The goal of `slacx` is to provide a fast, stable, and flexible platform for the analysis of image-like data. Much of the early development focused on supporting workflows for processing x-ray diffraction patterns. In principle, however, many other kinds of data could be processed by `slacx`. For this reason, a priority in `slacx` development is to maintain a flexible platform for users to develop their own processing routes. With basic programming knowledge, users can develop operations that load automatically into the `slacx` engine and become available for use in workflows. Instructions and a template for developing `slacx` operations are given in chapter 2.

# 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 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 it 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)*

The easiest way to maintain `pip`, `setuptools`, and `wheel` is to enable the `EPEL` repository (Extra Packages for Enterprise Linux) for your particular distribution.

### 1.1.4 setting up a virtual environment with virtualenv

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`:

```
pip install virtualenv
```

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

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

```
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. Install each of these packages by `pip install <packagename>`.

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
- `PyQtGraph`: Scientific visualization library built on PyQt4/PySide
- `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

# operation development

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 in the current implementation of `slacx`.

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 2.1 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.

## 2.1 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. Those requiring a more detailed walkthrough should read on.

```
# 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 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>'

    # 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 operations as functions in a module

etc etc etc