# reference manual and user guide for paws: the Platform for Automated Workflows by SSRL

Lenson A. Pellouchoud

February 14, 2017

## 0.1 Introduction

The `paws` software package aims to provide a fast and lean platform for building and executiong workflows for data processing. It was originally developed to perform analysis of diffraction images for ongoing projects at SLAC/SSRL. At the core of `paws` is a workflow engine that uses a library of simple operations to perform data analysis, for single inputs, batches of inputs, or inputs generated in real time.

`paws` is currently written in Python, making use of the PySide Qt library bindings for GUI elements as well as data structures and threading facilities. Internally, `paws` keeps track of data in Qt-based tree structures, and much of the graphical part of the program involves interacting with these trees through the Qt model-view framework.

`paws` also provides its functionality to `xi-cam`, a general-purpose synchrotron x-ray diffraction data analysis package with lots of fast and user-friendly data analysis tools. TODO: Cite and link to xi-cam here.

Some the core goals of `paws`:

- Eliminate redundant development efforts
- Streamline and standardize routine data analysis
- Simplify data storage and provide large-scale analysis
- Perform data analysis in real time for results-driven feedback

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

## 0.2 Quick start

Minimal and usually-effective installation instructions.

## 0.3 Usage instructions

How to use `paws`.

### 0.3.1 Operations

The library of operations built into `paws` was designed with specific workflows in mind, and in some cases an effort was made to provide more general functionality. Some operations have been used a lot and are known to be stable, while others may introduce bugs if used in ways other than intended. In order to keep the core requirements of `paws` to a minimum, `paws` defaults to a state where all operations are unavailable to the user. The user must choose which

operations will be used for their workflow, and the selected operations will attempt to import as the user enables them. The list of enabled operations is saved in a configuration file so that they do not need to be enabled every time `paws` is run.

This section gives an overview of the `paws` operation manager, including how to enable and disable operations and best practices for developing new operations.

**Enabling and Disabling Operations**

**Operation Development**

### 0.3.2 Workflow

**Building a Workflow**

**Execution**

**Batch and Realtime Execution**

### 0.3.3 API

**Python API**

**Command line API**

# Chapter 1

# installation

Instructions will go here for installing `paws` using the Python package installer `pip`, as soon as the `paws` package is prepared for that. Currently, installation from `pip` is not implemented. Users looking to install `paws` from source should look to appendix section 3.1.

# Chapter 2

# development

This chapter covers some best practices for contributing to `paws` via git (section 2.1) and instructions for developing operations for use in the `paws` workflow engine (section 2.2). These instructions are intended for people who are familiar with the `paws` development team and have received an invitation to work in their private repository.

## 2.1 contributing to paws

The `paws` repository is currently private. Request access by getting in touch with the `paws` developers at `paws-developers@slac.stanford.edu`. You can also join the paws-developers listserv. Send an email to

```
listserv@listserv.slac.stanford.edu
```

with no subject and the following text in the body:

```
SUB PAWS-DEVELOPERS
```

If you want to unsubscribe, use:

```
UNSUB PAWS-DEVELOPERS
```

If you need to configure your list settings, look up the latest ListServ documentation.

Code development is currently following the "shared repository model", meaning developers have write access to the main repository. All contributed code should be written in feature branches. The `master` and `dev` branches are protected, such that they can only be contributed to via pull requests or by contacting the development team.

The `master` branch is intended to be the most stable tested version, and the `dev` branch is for staging features that have been developed. The rest

of the branches should contain features being actively developed. Once you have access, clone the repository and create your own development or feature branch. When you are satisfied, make sure your feature branch includes the latest changes to `dev` (either merge `dev` into your branch, or rebase your branch on `dev`), and then submit a pull request to initiate a review that will ultimately merge your feature branch with `dev`.

## 2.2 developing operations for paws

Developing operations for use in the `paws` 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, parameters, references to plugin objects), performs some processing with those inputs, and produces some output (data and other features).

`paws` Operations are implemented by subclassing a Python abstract base class. An Operation defines names for its inputs and outputs, documentation for each input and output, and a run() method that creates outputs from the specified inputs. This process is most easily understood by following the template in section 2.2.1 below.

### 2.2.1 operation template

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.

```
from operation import Operation
import optools

# Replace <OperationName> with a SHORT operation title
class <OperationName>(Operation):
    # Replace <Description of Operation> with a detailed description
    """<Description of Operation>"""

    def __init__(self):
        """<Description of anything notable performed during __init__()>"""
        # Name the inputs and outputs for your operation.
        input_names = ['<input_name_1>','<input_name_2>',<...>]
        output_names = ['<output_name_1>','<output_name_2>',<...>]
        # Replace <OperationName> with the same name chosen above
        super(<OperationName>,self).__init__(input_names,output_names)
        # Provide a basic description for each of the inputs and outputs
        self.input_doc['<input_name_1>'] = '<expectations for input 1>'
        self.input_doc['<input_name_2>'] = '<etc>'
        self.output_doc['<output_name_1>'] = '<description of output 1>'
        self.output_doc['<output_name_2>'] = '<etc>'
        # OPTIONAL: set default sources, types, values for the inputs.
        # Valid sources:
        #   optools.no_input (default input to None),
```

```
    #   optools.wf_input (take input from another operation in the workflow),
    #   optools.text_input (manual text input)
    #   optools.plugin_input (take input from a PawsPlugin)
    #   optools.batch_input (input provided by a batch/realtime operation)
    self.input_src['<input_name_1>'] = <optools.some_source>
    self.input_src['<input_name_2>'] = <etc>
    # Valid types: optools.none_type (None), optools.str_type (string),
    #   optools.int_type (integer), optools.float_type (float),
    #   optools.bool_type (boolean), optools.ref_type (direct reference),
    #   optools.path_type (a path to something in the filesystem or workflow),
    #   optools.auto_type (default for chosen source, or input set by batch)
    self.input_type['<input_name_1>'] = <optools.some_type>
    self.input_type['<input_name_2>'] = <etc>

# Write a run() function for this Operation.
def run(self):
    """<Description of processing performed by run()>"""
    # 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 output
    self.outputs['<output_name_1>'] = <computed_value_1>
    self.outputs['<output_name_2>'] = <etc>
```

### 2.2.2 batch and realtime execution operations

Some additional work is required to develop custom batch and realtime execution controllers. etc etc.

# Chapter 3

# appendix

## 3.1 Installation from Source

### 3.1.1 pip, setuptools, and wheel

`pip` is a Python package manager (acronym: `pip` Installs Python). `setuptools` is a Python library that supports many of the setup operations that follow. `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. This may not be true for some Enterprise Linux (e.g. RHEL, CentOS, SL, OL) machines. Enterprise Linux users may want to see section 3.1.3. For most mainstream Linux distributions (Fedora, Ubuntu, etc.) it is suggested that the user employ the system's package manager (skip ahead to section 3.1.2).

### 3.1.2 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
```

Once pip is installed, it can be used to upgrade itself by typing:

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

### 3.1.3 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 3.1.3.

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 to 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 3.1.3

### preparing PySide Qt on Linux

Since `paws` 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
```

### 3.1.4 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 `paws` 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`.

### 3.1.5 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>`.

- `PySide`: Python `Qt` bindings library: Some Linux distributions may need to go through section 3.1.3 above before this installs nicely.
- `QDarkStyle`: A dark stylesheet for the Qt GUI.
- `PyYAML`: Python library for YAML serialization language

`paws` is not very useful, though, without the modules that are used to operate on data. You may want to install some of the libraries that are used commonly in `paws` operations:

- `numpy`: Python numerical computation library
- `scipy`: Python scientific math library
- `matplotlib`: Highly versatile plotting library
- `PyQtGraph`: Scientific visualization library built on PyQt4/PySide
- `pillow`: A replacement for the outdated Python Imaging Library (PIL)
- `tifffile`: Library for reading data from .tif image files
- `fabio`: Fable I/O library, parses image files from common CCD array detectors devices
- `pyFAI`: Fast Azimuthal Integration library for manipulating and reducing image data
- `tzlocal`: Tools for interpreting time data relative to the local time zone