Working with Python

Python is available on all Sterrewacht and Lorentz Institute GNU/Linux workstations. In most cases, both python v2 and python v3 are available. Please note that python v2 has reached its *end-of-life* on 01-01-2020 and therefore is no longer actively supported.

Python packages

Many common python packages, such as numpy, scipy, and astropy are available to any users regardless of the workstation/server. These are installed either locally to the workstation – usually via the OS package manager – or remotely and exposed to the users by means of the `module` command.

For local python installations, you can list all installed packages via

```
python3 -m pip list # or python2 -m pip list
```

For remote python installations – installations on our software disk – you must first load a python module and then list all packages in that module

```
module load Python/3.6.6-foss-2018b
which python
/easybuild/easybuild/fc31/software/Python/3.6.6-foss-2018b/bin/python
python -m pip list
```

If the package you would like to use is not installed at all you have two options:

- Request the installation to the system administrators.
- Install it yourself, to one of the directories to which you have writing access.

The two options are described in detail in the sections below.

Request a Python Package Installation

If you believe that the required package could be useful to other researchers in the Observatory or Lorentz Institute, then you can request its installation via our helpdesk https://helpdesk.strw.leidenuniv.nl/ (STRW) or https://helpdesk.lorentz.leidenuniv.nl/ (Lorentz) giving motivations and detailed instructions on where to find the requested package and its license information. We will notify you when the installation is complete.

Install a Python Package yourself

There are instances in which you would like to install a python package that is not useful to other researchers in your department and/or you are a developer who wants to try and modify development versions of installed packages or new packages. In other words, if

1. The package is not of interest to the majority of users.
2. You want a custom stash of packages, that is not visible to other users.
3. You want to isolate a set of packages to a specific python application, usually to minimize the possibility of version conflicts.

we advise you follow one of the methods below to install the package yourself. Rest assured though that we can always assist you during the process.


The following methods are valid either you use a local (to your workstation/server) python installation or a python installation provided via the `module` command. If you choose the latter, remember to load first an appropriate python module.

METHOD 1: pip with the `--user' option

Python 2.6 introduced the possibility of package installations via a “user scheme”. According to this scheme, Python distributions support an alternative install location that is specific to a user. Python provides this functionality via the `site` module which commands where you, as a user, will be installing python packages

```
python -m site --user-base # value of site.USER_BASE
python -m site --user-site # path to your site-packages directory
```

If the values returned by the command above satisfy you, you can then proceed to install packages in your user-space

```
pip install --user SomePackage
```

In the STRW and IL environments, site.USER_BASE defaults to `$HOME/.local`. This path can be customised/updated by modifying the environment variable PYTHONUSERBASE

```
export PYTHONUSERBASE=/somewhere/I/can/write/to # alternative location
pip install --user SomePackage
```

will install `SomePackage' in `/somewhere/I/can/write/to/lib/python*/site-packages`.

When using the `user' scheme to install packages, it is important to note

- When globally installed packages are on the python path, and they conflict with the installation requirements, they are ignored, and not uninstalled.
- When globally installed packages are on the python path, and they satisfy the installation requirements, pip does nothing, and reports that requirement is satisfied.
- pip will not perform a `–user` install in a virtualenv unless the virtualenv was created specifying `–system-site-packages`. Nonetheless, pip will never install a package that conflicts with a package in the virtualenv site-packages.

Method 1 - subsection Incompatible versions

Unfortunately, python's 'user' directory is independent of the operating system version, but most of

the compute nodes including VDESK, LOFAR cluster and other compute nodes, run RedHat Enterprise Linux, which is sufficiently different to cause packages installed on the desktop not to work all the time.

In cases like this, it might be necessary to create a separate python user directory structure for those machines:

Add to your .bashrc something like this:

```
if [ ! -f /etc/fedora-release ]; then
  export PYTHONUSERBASE=$HOME/.local-rhel9
fi
```

For users of the `tcsh` shell, add this to your .tcshrc in stead:

```
if (! -f /etc/fedora-release) then
  setenv PYTHONUSERBASE $HOME/.local-rhel9
endif
```

And make sure to create that directory ~/.local-rhel9. Now the pip –user commands on RHEL9 machines will install into that newly created directory in stead of the default one used by the desktop systems.

METHOD 2: venv

venv is a tool that creates isolated Python environments; it replaces the obsolete `virtualenv` that provided similar functionality for python 2.x. A python environment is essentially a folder which contains copies of all necessary files needed for a Python project to run. In addition each virtual environment will contain a copy of the utility pip to manage packages. For example, let us suppose you would like to install `pymatlab` which is not installed on the departmental workstations, then you could do

```
$ mkdir /data2/username/venvs
$ python3 -m venv /data2/username/venvs/pymatlab
```

to create a virtual environment (folder) called pymatlab (note that this example explicitly creates this in a directory on your local `/data2` disk, in order to avoid running out of disk quota in your home directory, which can easily happen since venvs can become rather big).

In the example, we use `python3` as the python for this environment; if there are multiple python versions on the system, and you want to base your venv on a specific version, use that version to create the venv, e.g. `python3.12 -m venv /data2/username/venvs/pymatlab`.

The last step before starting to use the newly generated environment is to activate it, that is to prepend its `/bin` folder to your $PATH environment variable. This is done by issuing

```
source /data2/username/pymatlab/bin/activate # bash shells
source /data2/username/pymatlab/bin/activate.csh # c shells
```

To acknowledge the activation of pymatlab, the terminal prompt will be changed to

```
(pymatlab)username@hostname:~/python_virt_envs/pymatlab$
```

to emphasize that you are operating in a virtual environment. To install pymatlab (or any other package) locally (in your virtual environment) run pip within that environment

```
pip install  pymatlab
```

Your virtual environment now should have the same core python packages defined globally for all the Observatory or Lorentz Institute users plus any packages installed in the virtual environment. Note that you do NOT use –`user` on the pip command in this case, since that would install in your $PYTHONUSERBASE directory (see above) instead of the venv!!

In any cases, it is advisable you keep a backup of your virtual environment configuration by creating a list of installed packages

```
pip freeze > packages.dat
```

This can help collaborators and fellow developers to reproduce your environment with

```
pip install -r packages.dat
```

When you are done working in a virtual environment deactivate it running

```
deactivate
```

At any time, any virtual environment can be destroyed by removing the corresponding folder from the file system so do not panic if things do not work, just delete your virtual environment and start all over again.

Note: System administrators will not be responsible and/or manage users virtual environments. You are strongly advised you consult the documentation.

METHOD 2: OBSOLETE: virtualenv (python 2.x)

This guide refers to virtualenv version 12.0.7.

virtualenv is a tool that creates isolated Python environments. A python environment is essentially a folder which contains copies of all necessary files needed for a Python project to run. In addition each virtual environment will contain a copy of the utility pip to manage packages. For example, let us suppose you would like to install `pymatlab` which is not installed on the departmental workstations, then you could do

```
$ mkdir python_virt_envs && cd python_virt_envs
$ virtualenv --system-site-packages pymatlab
```

to create a virtual environment (folder) called pymatlab.

Python virtual environments containing specific versions of python can be created using the `-p` option as in `virtualenv -p /usr/bin/python3.6`.

The last step before starting to use the newly generated environment is to activate it, that is to prepend its /bin folder to your $PATH environment variable. This is done by issuing

```
source pymatlab/bin/activate # bash shells
source pymatlab/bin/activate.csh # c shells
```

To acknowledge the activation of pymatlab, virtualenv will change the terminal prompt $PS1 to

```
(pymatlab)username@hostname:~/python_virt_envs/pymatlab$
```

to emphasize that you are operating in a virtual environment. To install pymatlab (or any other package) locally (in your virtual environment) run pip within that environment

```
pip install  pymatlab
```

Your virtual environment now should have the same core python packages defined globally for all the Observatory or Lorentz Institute users plus any packages installed in the virtual environment.

In any cases, it is advisable you keep a backup of your virtual environment configuration by creating a list of installed packages

```
pip freeze > packages.dat
```

This can help collaborators and fellow developers to reproduce your environment with

```
pip install -r packages.dat
```

When you are done working in a virtual environment deactivate it running

```
deactivate
```

At any time, any virtual environment can be destroyed by removing the corresponding folder from the file system by executing

```
rm -rf ~/python_virt_envs/pymatlab
```

so do not panic if things do not work, just delete your virtual environment and start all over again.

Note: System administrators will not be responsible and/or manage users virtual environments. You are strongly advised you consult the documentation

```
virtualenv --help
```

METHOD 3: easy_install with the `--user' option

Easy Install is a python module (easy_install) that lets you automatically download, build, install, and manage Python packages. By default, easy_install installs python packages into Python's main site-packages directory, and manages them using a custom .pth file in that same directory. Very often though, a user or developer wants easy_install to install and manage python packages in an

alternative location. This possible via the –user option in a similar fashion to pip's

```
easy_install -N --user pymatlab
```

This will install pymatlab in ${HOME}/.local/ ready to be imported in your next python session. If you want to install your package in a different location than $HOME/.local, then set the environment variable $PYTHONUSERBASE to a custom location, e.g,

```
export PYTHONUSERBASE=/home/user/some/where/I/can/write # alternative location
```

Please consult the docs to know more:

```
python -m easy_install --help
```

Migrating packages between python versions

Another issue when using personal installs may arrise on operating system upgrades, when a newer version of python is made the default (eg, moving from python 3.7 to python 3.9). Notes copied from the Fedora release notes:

1. Make a list of installed packages in the old python version:

```
python3.7 -m pip freeze > installed.txt
```

1. Reinstall for the current python version:

```
python3.9 -m pip install --user -r installed.txt
```

1. Optionally, uninstall the packages from the old python version and/or remove the obsolete directory under $HOME/.local/lib/python3.7

Example: how to let python search arbitrary library paths

For instance for python v2.7 installations, create or edit

`$HOME/.local/lib/python2.7/site-packages/my-super-library.pth`

by appending the path of your choice

```
echo "/my/home/sweet/home/library" >> $HOME/.local/lib/python2.7/site-packages/my-super-library.pth
```

All `.pth` files will be sourced by python provided they are in the right location.

Example: how to create your own python environment module

Please read here.

Example: numpy with openBLAS

In this example we create a python2 virtual environment in which we will install the latest version of numpy that will use the openBLAS library.


The procedure and paths below will work on any maris node.

```
virtualenv py2_numpy_openBLAS
source py2_numpy_openBLAS/bin/activate
cd py2_numpy_openBLAS
mkdir numpy
pip install -d numpy numpy && cd numpy
tar xzf numpy-X.Y.z.tar.gz
cd numpy-X.Y.Z/
cp site.cfg.example site.cfg
```

Edit `site.cfg` with your favorite editor such that

```
[openblas]
libraries = openblas
library_dirs = /usr/lib64
include_dirs = /usr/include/openblas/
runtime_library_dirs = /usr/lib64
```

then install numpy

```
python setup.py install
```

If the installation is going smoothly you should see

```
....

openblas_info:
  FOUND:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/lib64']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
    runtime_library_dirs = ['/usr/lib64']

....
```

```
Installed /some/where/py2_numpy_openBLAS/lib/python2.7/site-packages/numpy-
X.Y.Z-py2.7-linux-x86_64.egg
```

Now that numpy is installed you could also install scipy, for instance

```
pip install scipy
```

openBLAS will automatically use multithreading on the basis of the computer resources and the executable. If you wanted more control on multithreading you could either build openBLAS from source by specifying the number of threads or specify the number of threads in your application. If none of the above methods satisfies you, then it is possible to set the environment variable OPENBLAS_NUM_THREADS.

 Be careful! Choose the number of threads with care or your application will run slower than a single-threaded one!

 If your application is parallelized please build OpenBLAS with USE_OPENMP=1.

 If your application is already multi-threaded, it will conflict with OpenBLAS multi-threading. You must

- export OPENBLAS_NUM_THREADS=1 in the environment variables. Or
- Call openblas_set_num_threads(1) in the application on runtime. Or
- Build OpenBLAS single thread version, e.g. make USE_THREAD=0

In any cases, please READ the docs.

Bypassing the existing python environment

Occasionally, something in the systemwide directories (e.g `/software/local/lib64/python2.7/site-packages`) interferes with your python application. Perhaps you have a code that requires a specific, older, version of numpy or matplotlib. Just installing that version is not always sufficient. The trick is, to set the PYTHONPATH to point first to a directory where you place a private `sitecustomize.py` which then overrides the one we have placed in /usr/lib64/python2.7/site-packages (which is where we add the /software directories to the path for everyone). Here is how:

```
  mkdir /some/location/python_custom_dir
  setenv PYTHONPATH
/some/location/python_custom_dir:/usr/lib64/python2.7/site-packages
```

The `sitecustomize.py` could be something like this:

```
  import sys
```

```
  import site
  mypath='/usr/lib64/python%s/site-packages' % sys.version[:3]
  # We want this directory at the start of the path, to enforce the original
defaults
  sys.path.insert(1,mypath)
  # In order to find also eggs and subdirectories, addsitedir seems
necessary:
  site.addsitedir(mypath, known_paths=None)
```

Anaconda/Miniconda

Another way of using a private python install (separate versions etc), is to install and use Anaconda/Miniconda. Since these environments can encompass much more than just python, they deserve their own page (especially since they come with their own share of pitfalls).

Jupyter Notebooks

Depending on your operating system (Fedora or RedHat) you might get a different python kernel version as the standard kernel. If you get `python2` as the default kernel and only option, but wish the use the `python3` kernel you need to add this kernel to you local environment. This can be done by executing:

```
  python3 -m ipykernel install --user
```

Once this command has run successfully, it will have installed python3 as a jupyter kernel.

After starting `jupyter notebook` you can select `python3` as kernel.

If you need to work with several python setups (e.g. the system python3, but also from loaded environment modules), it is easy to assign a name to the generated kernel, e.g:

```
  python3 -m ipykernel install --user --name system-python3
```