yt Cheat Sheet

General Info

For everything yt please see `http://yt-project.org`. Documentation `http://yt-project.org/doc/index.html`. Need help? Start here `http://yt-project.org/doc/help/` and then try the IRC chat room `http://yt-project.org/irc.html`, or the mailing list `http://lists.spacepope.org/listinfo.cgi/yt-users-spacepope.org`.

Installing yt

The easiest way to install yt is to use the installation script found on the yt homepage or the docs linked above. If you already have python set up with `numpy`, `scipy`, `matplotlib`, `h5py`, and `cython`, you can also use `pip install yt`

Command Line yt

yt, and its convenience functions, are launched from a command line prompt. Many commands have flags to control behavior. Commands can be followed by **--help** (e.g. **yt render --help**) for detailed help for that command including a list of the available flags.
`iyt`— Load yt and IPython.
`yt load` *dataset* — Load a single dataset.
`yt help` — Print yt help information.
`yt stats` *dataset* — Print stats of a dataset.
`yt update` — Update yt to most recent version.
`yt update --all` — Update yt and dependencies to most recent version.
`yt version` — yt installation information.
`yt notebook` — Run the IPython notebook server.
`yt upload_image` *image.png* — Upload PNG image to imgur.com.
`yt upload_notebook` *notebook.nb* — Upload IPython notebook to hub.yt-project.org.
`yt plot` *dataset* — Create a set of images.
`yt render` *dataset* — Create a simple volume rendering.
`yt mapserver` *dataset* — View a plot/projection in a Gmaps-like interface.
`yt pastebin` *text.out* — Post text to the pastebin at paste.yt-project.org.
`yt pastebin_grab` *identifier* — Print content of pastebin to STDOUT.
`yt bugreport` — Report a yt bug.
`yt hop` *dataset* — Run hop on a dataset.

yt Imports

In order to use yt, Python must load the relevant yt modules into memory. The import commands are entered in the Python/IPython shell or used as part of a script.
`import yt` — Load yt.
`from yt.config import ytcfg` — Used to set yt configuration options. If used, must be called before importing any other module.
`from yt.analysis_modules.`*halo_finding*`.api import *` — Load halo finding modules. Other modules are loaded in a similar way by swapping the *emphasized* text. See the **Analysis Modules** section for a listing and short descriptions of each.

YTArray

Simulation data in yt is returned as a YTArray. YTArray is a numpy array that has unit data attached to it and can automatically handle unit conversions and detect unit errors. Just like a numpy array, YTArray provides a wealth of built-in functions to calculate properties of the data in the array. Here is a very brief list of some useful ones.
`v = a.in_cgs()` — Return the array in CGS units
`v = a.in_units('Msun/pc**3')` — Return the array in solar masses per cubic parsec
`v = a.max(), a.min()` — Return maximum, minimum of `a`.
`index = a.argmax(), a.argmin()` — Return index of max, min value of `a`.
`v = a[`*index*`]` — Select a single value from `a` at location *index*.
`b = a[`*i:j*`]` — Select the slice of values from `a` between locations *i* to *j-1* saved to a new Numpy array `b` with length *j-i*.
`sel = (a > const)` — Create a new boolean Numpy array `sel`, of the same shape as `a`, that marks which values of `a > const`. Other operators (e.g. <, !=, %) work as well.
`b = a[sel]` — Create a new Numpy array `b` made up of elements from `a` that correspond to elements of `sel` that are *True*. In the above example `b` would be all elements of `a` that are greater than `const`.
`a.write_hdf5(`*filename.h5*`)` — Save `a` to the hdf5 file *filename.h5*.

IPython Tips

These tips work if IPython has been loaded, typically either by invoking `iyt` or `yt load` on the command line, or using the IPython notebook (`yt notebook`). `Tab complete` — IPython will attempt to auto-complete a variable or function name when the `Tab` key is pressed, e.g. *HaloFi*–`Tab` would auto-complete to *HaloFinder*. This also works with imports, e.g. *from numpy.random.*–`Tab` would give you a list of random functions (note the trailing period before hitting `Tab`).
`?, ??` — Appending one or two question marks at the end of any object gives you detailed information about it, e.g. *variable_name?*.
Below a few IPython "magics" are listed, which are IPython-specific shortcut commands.
`%paste` — Paste content from the system clipboard into the IPython shell.
`%hist` — Print recent command history.
`%quickref` — Print IPython quick reference.
`%pdb` — Automatically enter the Python debugger at an exception.
`%debug` — Drop into a debugger at the location of the last unhandled exception.
`%time, %timeit` — Find running time of expressions for benchmarking.
`%lsmagic` — List all available IPython magics. Hint: `?` works with magics.
Please see `http://ipython.org/documentation.html` for the full IPython documentation.

Load and Access Data

The first step in using yt is to reference a simulation snapshot. After that, simulation data is generally accessed in yt using *Data Containers* which are Python objects that define a region of simulation space from which data should be selected. `ds = yt.load(`*dataset*`)` — Reference a single snapshot.
`dd = ds.all_data()` — Select the entire volume.
`a = dd[`*field_name*`]` — Copies the contents of *field* into the YTArray `a`. Similarly for other data containers.
`ds.field_list` — A list of available fields in the snapshot.
`ds.derived_field_list` — A list of available derived fields in the snapshot.
`val, loc = ds.find_max("Density")` — Find the `value` of the maximum of the field `Density` and its `location`.
`sp = ds.sphere(`*cen*`,`*radius*`)` — Create a spherical data container. *cen* may be a coordinate, or "max" which centers on the max density point. *radius* may be a float in code units or a tuple of (*length, unit*).
`re = ds.region(`*cen*`, `*left edge*`, `*right edge*`)` — Create a rectilinear data container. *cen* is required but not used. *left* and *right edge* are coordinate values that define the region.
`di = ds.disk(`*cen*`, `*normal*`, `*radius*`, `*height*`)` — Create a cylindrical data container centered at *cen* along the direction set by *normal*,with total length 2×*height* and with radius *radius*.
`ds.save_object(sp, `*"sp_for_later"*`)` — Save an object (`sp`) for later use.
`sp = ds.load_object(`*"sp_for_later"*`)` — Recover a saved object.

Defining New Fields

`yt` expects on-disk fields, fields generated on-demand and in-memory. Field can either be created before a dataset is loaded using `add_field`:

```
def _metal_mass(field,data)
   return data["metallicity"]*data["cell_mass"]
add_field("metal_mass", units='g',
function=_metal_mass)
```

Or added to an existing dataset using `ds.add_field`:

```
ds.add_field("metal_mass", units='g',
function=_metal_mass)
```

Slices and Projections

`slc = yt.SlicePlot(ds, `*axis or normal vector*`, `*field*`, `*center=*`, `*width=*`, `*weight_field=*`, `*additional parameters*`)` — Make a slice plot perpendicular to *axis* (specified via 'x', 'y', or 'z') or a normal vector for an off-axis slice of *field* weighted by *weight_field* at (code-units) *center* with *width* in code units or a (value, unit) tuple. Hint: try *yt.SlicePlot?* in IPython to see additional parameters.

`slc.save(`*file_prefix*`)` — Save the slice to a png with name prefix *file_prefix*. `.save()` works similarly for the commands below.
`prj = yt.ProjectionPlot(ds,` *axis*, *field*, *addit. params*`)` — Make a projection.
`prj = yt.OffAxisProjectionPlot(ds,` *normal*, *fields*, *center=*, *width=*, *depth=*, *north_vector=*, *weight_field=*`)` —Make an off axis projection. Note this takes an array of fields.

Plot Annotations

Plot callbacks are functions itemized in a registry that is attached to every plot object. They can be accessed and then called like `prj.annotate_velocity(factor=16, normalize=False)`. Most callbacks also accept a *plot_args* dict that is fed to matplotlib annotator.
`velocity(`*factor=*, *scale=*, *scale_units=*, *normalize=*`)` — Uses field "x-velocity" to draw quivers
`magnetic_field(`*factor=*, *scale=*, *scale_units=*, *normalize=*`)` — Uses field "Bx" to draw quivers
`quiver(`*field_x*, *field_y*, *factor=*, *scale=*, *scale_units=*, *normalize=*`)`
`contour(`*field=*, *ncont=*, *factor=*, *clim=*, *take_log=*, *additional parameters*`)` —Plots a number of contours *ncont* to interpolate *field* optionally using *take_log*, upper and lower contour*lim*its and *factor* number of points in the interpolation.
`grids(`*alpha=*, *draw_ids=*, *periodic=*, *min_level=*, *max_level=*`)` —Add grid boundaries.
`streamlines(`*field_x*, *field_y*, *factor=*, *density=*`)`
`clumps(`*clumplist*`)` — Generate *clumplist* using the clump finder and plot.
`arrow(`*pos*, *code_size*`)` Add an arrow at a *pos*ition.
`point(`*pos*, *text*`)` — Add text at a *pos*ition.
`marker(`*pos*, *marker=*`)` — Add a matplotlib-defined marker at a *pos*ition.
`sphere(`*center*, *radius*, *text=*`)` — Draw a circle and append *text*.
`hop_circles(`*hop_output*, *max_number=*, *annotate=*, *min_size=*, *max_size=*, *font_size=*, *print_halo_size=*, *fixed_radius=*, *min_mass=*, *print_halo_mass=*, *width=*`)` — Draw a halo, printing it's ID, mass, clipping halos depending on number of particles (*size*) and optionally fixing the drawn circle radius to be constant for all halos.
`hop_particles(`*hop_output*, *max_number=*, *p_size=*, *min_size*, *alpha=*`)` — Draw particle positions for member halos with a certain number of pixels per particle.
`particles(`*width*, *p_size=*, *col=*, *marker=*, *stride=*, *ptype=*, *stars_only=*, *dm_only=*, *minimum_mass=*, *alpha=*`)` — Draw particles of *p_size* pixels in a slab of *width* with *col*or using a matplotlib *marker* plotting only every *stride* number of particles.
`title(`*text*`)`

The ~/.yt/ Directory

yt will automatically check for configuration files in a special directory (`$HOME/.yt/`) in the user's home directory.

The `config` file — Settings that control runtime behavior.
The `my_plugins.py` file — Add functions, derived fields, constants, or other commonly-used Python code to yt.

Analysis Modules

The import name for each module is listed at the end of each description (see **yt Imports**).

`Absorption Spectrum` — (`absorption_spectrum`).
`Clump Finder` — Find clumps defined by density thresholds (`level_sets`).
`Halo Finding` — Locate halos of dark matter particles (`halo_finding`).
`Light Cone Generator` — Stitch datasets together to perform analysis over cosmological volumes.
`Light Ray Generator` — Analyze the path of light rays.
`Rockstar Halo Finding` — Locate halos of dark matter using the Rockstar halo finder (`halo_finding.rockstar`).
`Star Particle Analysis` — Analyze star formation history and assemble spectra (`star_analysis`).
`Sunrise Exporter` — Export data to the sunrise visualization format (`sunrise_export`).

Parallel Analysis

Nearly all of yt is parallelized using MPI. The *mpi4py* package must be installed for parallelism in yt. To install *pip install mpi4py* on the command line usually works. Execute python in parallel similar to this:
mpirun -n 12 python script.py
The file `script.py` must call the `yt.enable_parallelism()` to turn on yt's parallelism. If this doesn't happen, all cores will execute the same serial yt script. This command may differ for each system on which you use yt; please consult the system documentation for details on how to run parallel applications.
`parallel_objects()` — A way to parallelize analysis over objects (such as halos or clumps).

Mercurial

Please see `http://mercurial.selenic.com/` for the full Mercurial documentation.
`hg clone https://bitbucket.org/yt_analysis/yt` — Clone a copy of yt.
`hg status` — Files changed in working directory.
`hg diff` — Print diff of all changed files in working directory.
`hg diff -r`*RevX* `-r`*RevY* — Print diff of all changes between revision *RevX* and *RevY*.
`hg log` — History of changes.
`hg cat -r`*RevX file* — Print the contents of *file* from revision *RevX*.
`hg heads` — Print all the current heads.
`hg revert -r`*RevX file* — Revert *file* to revision *RevX*. On-disk changed version is moved to *file.orig*.
`hg commit` — Commit changes to repository.
`hg push` — Push changes to default remote repository.
`hg pull` — Pull changes from default remote repository.
`hg serve` — Launch a webserver on the local machine to examine the repository in a web browser.

FAQ

`slc.set_log('field', False)` — When plotting `field`, use linear scaling instead of log scaling.