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.

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 instinfo` — yt installation information.
`yt notebook` — Run the IPython notebook server.
`yt serve` (*dataset*) — Run yt-specific web GUI (*dataset* is optional).
`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 hub_register` — Register with hub.yt-project.org.
`yt hub_submit` — Submit hg repo to hub.yt-project.org.
`yt bootstrap_dev` — Bootstrap a yt development environment.
`yt bugreport` — Report a yt bug.
`yt hop` *dataset* — Run hop on a dataset.
`yt rpdb` — Connect to running rpd session.

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.

`from yt.mods import *` — Load base yt modules.
`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.api import *` — Load all yt analysis modules.
`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.

Numpy Arrays

Simulation data in yt is returned in Numpy arrays. The Numpy package provides a wealth of built-in functions that operate on Numpy arrays. Here is a very brief list of some useful ones. Please see http://docs.scipy.org/doc/numpy/reference/ for the full numpy documentation.

`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.dump(`*filename.dat*`)` — Save `a` to the binary file *filename.dat*.
`a = np.load(`*filename.dat*`)` — Load the contents of *filename.dat* into `a`.

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.
`%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. `pf = load(`*dataset*`)` — Reference a single snapshot.
`dd = pf.h.all_data()` — Select the entire volume.
`a = dd[`*field_name*`]` — Saves the contents of *field* into the numpy array `a`. Similarly for other data containers.
`pf.h.field_list` — A list of available fields in the snapshot.
`pf.h.derived_field_list` — A list of available derived fields in the snapshot.
`val, loc = pf.h.find_max("Density")` — Find the `val`ue of the maximum of the field `Density` and its `loc`ation.
`sp = pf.h.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 = pf.h.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 = pf.h.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*.
`bl = pf.h.boolean(`*constructor*`)` — Create a boolean data container. *constructor* is a list of pre-defined non-boolean data containers with nested boolean logic using the "AND", "NOT", or "OR" operators. E.g. *constructor= [sp, "NOT", (di, "OR", re)]* gives a volume defined by *sp* minus the patches covered by *di* and *re*.
`pf.h.save_object(sp, `*"sp_for_later"*`)` — Save an object (`sp`) for later use.
`sp = pf.h.load_object(`*"sp_for_later"*`)` — Recover a saved object.

Defining New Fields & Quantities

`yt` expects on-disk fields, fields generated on-demand and in-memory. Quantities reduce a field (e.g. "Density") defined over an object (e.g. "sphere") to get a single value (e.g. "Mass").

```
def _MetalMassMsun(field,data)
    return
data["Metallicity"]*data["CellMassMsun"]
add_field("MetalMassMsun",function=_MetalMassMsun)
```

Define a new quantity; note the first function operates on grids and data objects and the second on the results of the first.

```
def _TotalMass(data):
    baryon_mass = data["CellMassMsun"].sum()
```

```
    particle_mass = data["ParticleMassMsun"].sum()
    return baryon_mass, particle_mass
def _combTotalMass(data, baryon_mass,
particle_mass):
    return baryon_mass.sum() + particle_mass.sum()
add_quantity("TotalMass", function=_TotalMass,
    combine_function=_combTotalMass, n_ret = 2)
```

Slices and Projections

`slc = SlicePlot(pf,` *axis*, *field*, *center=*, *width=*, *weight_field=*, *additional parameters*`)` — Make a slice plot perpendicular to *axis* of *field* weighted by *weight_field* at (code-units) *center* with *width* in code units or a (value, unit) tuple. Hint: try *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 = ProjectionPlot(pf,` *axis*, *field*, *addit. params*`)` — Make a projection.
`prj = OffAxisSlicePlot(pf,` *normal*, *fields*, *center=*, *width=*, *depth=*, *north_vector=*, *weight_field=*`)` —Make an off-axis slice. Note this takes an array of fields.
`prj = OffAxisProjectionPlot(pf,` *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 *contourlim*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`).
`Coordinate Transformation` — (`coordinate_transformation`).
`Halo Finding` — Locate halos of dark matter particles (`halo_finding`).
`Halo Mass Function` — Find halo mass functions from data and from theory (`halo_mass_function`).
`Halo Profiling` — Profile and project multiple halos (`halo_profiler`).
`Halo Merger Tree` — Create a database of halo mergers (`halo_merger_tree`).
`Light Cone Generator` — Stitch datasets together to perform analysis over cosmological volumes.
`Light Ray Generator` — Analyze the path of light rays.
`Radial Column Density` — Calculate column densities around a point (`radial_column_density`).
`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`).
`Two Point Functions` — Two point correlations (`two_point_functions`).

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 –parallel
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.
`from yt.pmods import *` — Load yt faster when in parallel. This replaces the usual `from yt.mods import *`.
`parallel_objects()` — A way to parallelize analysis over objects (such as halos or clumps).

Pre-Installed Versions

yt is pre-installed on several supercomputer systems.
NICS Kraken — *module load yt*

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

`pf.field_info['field'].take_log = False` — When plotting `field`, do not take log. Must enter `pf.h` before this command.