

Introduction to Python

IPTA 2018 Student Workshop, Socorro NM

Adam Brazier and Nate Garver-Daniels

How is this going to proceed?

- Some talking, but mostly practical examples through which you work
 - These will include useful tools like numpy, astropy, etc. We will be walking around to help people with advice and technical support
- The practical examples will be in Jupyter (formerly iPython) notebooks. You will be working through those.
- There will be a 45m-1h section at the end where we return to the slides, to cover some best practices, gotchas and general advice
 - This will include virtualenv, code formatting, portability, general software engineering
- Stop us at any time to ask questions!

How do we tend to learn, as astronomers

In general, astronomers learn programming like this:

1. Reading other people's code

Much of the work in astronomy is produced with code. Ideally, you can get to read it, to see what they did. Even without coding proficiency, you can get the shape of what it's doing. Ideally, the code is somewhat clear, and has comments and maybe even some documentation.

Sometimes it's not clear, has few useful comments, and is not documented.

How do we tend to learn, as astronomers

In general, astronomers learn programming like this:

1. Reading other people's code
2. Hacking other people's code

A good starting point for what you want to do is often what other people have already done, that worked. After some of this, you start to feel like you can do it yourself, but! Fear level: high

How do we tend to learn, as astronomers

In general, astronomers learn programming like this:

1. Reading other people's code
2. Hacking other people's code
3. Writing your own code

Now you need to worry about the rules more than before. You also need to design your code first (or, at least, you should). At this point, you are responsible for the results! Fear level: may still be high

How do we tend to learn, as astronomers

In general, astronomers learn programming like this:

1. Reading other people's code
2. Hacking other people's code
3. Writing your own code
4. **NOW I AM THE MASTER OF THE UNIVERSE! NO FEAR**

How do we tend to learn, as astronomers

In general, astronomers learn programming like this:

1. Reading other people's code
2. Hacking other people's code
3. Writing your own code
4. NOW I AM THE MASTER OF THE UNIVERSE! NO FEAR

This method actually pretty much works, with pitfalls with which you may already be familiar. Also the basic approach we will take in this workshop, with guidelines as to how to proceed to universe-mastery, hopefully reducing the fear level.

What is the scope and target audience?

- We have a very wide variety of Python expertise in any room of astronomers
- There should be something to interest most of you
 - We don't *assume* any pre-existing expertise
 - The examples cover a range of difficulty
- We can't cover everything, but we aim to help you find the rest
- We will proceed by *doing*, rather than a lot of instruction from the front

What is Python? Why Python?

Python is an interpreted language and runs on about any platform. In most cases, you will be sending your programs to the Python interpreter which then runs them.

Characteristics of Python include:

- Indentation to define code blocks
- Duck-typing (types are *checked at run-time*)
- Cross-platform portability (but libraries you call may not be cross-platform)

We use it because Python is very popular in astronomy; it allows rapid application development (RAD) and has an extensive set of libraries which are useful to us.

Which Python

There are two major versions of Python, Python 2 (current and final minor version is 2.7, will be retired in 2020) and Python3 (current version Python 3.6 and 3.7)

You may still need to use Python2.7 for some applications, but we are using Python3 here. Of differences, there is some discussion in this notebook:

```
Python2_and_Python3_Some__future__.ipynb
```

Probably best to start projects in Python3 unless you have legacy 2.7 code

Jupyter Notebooks

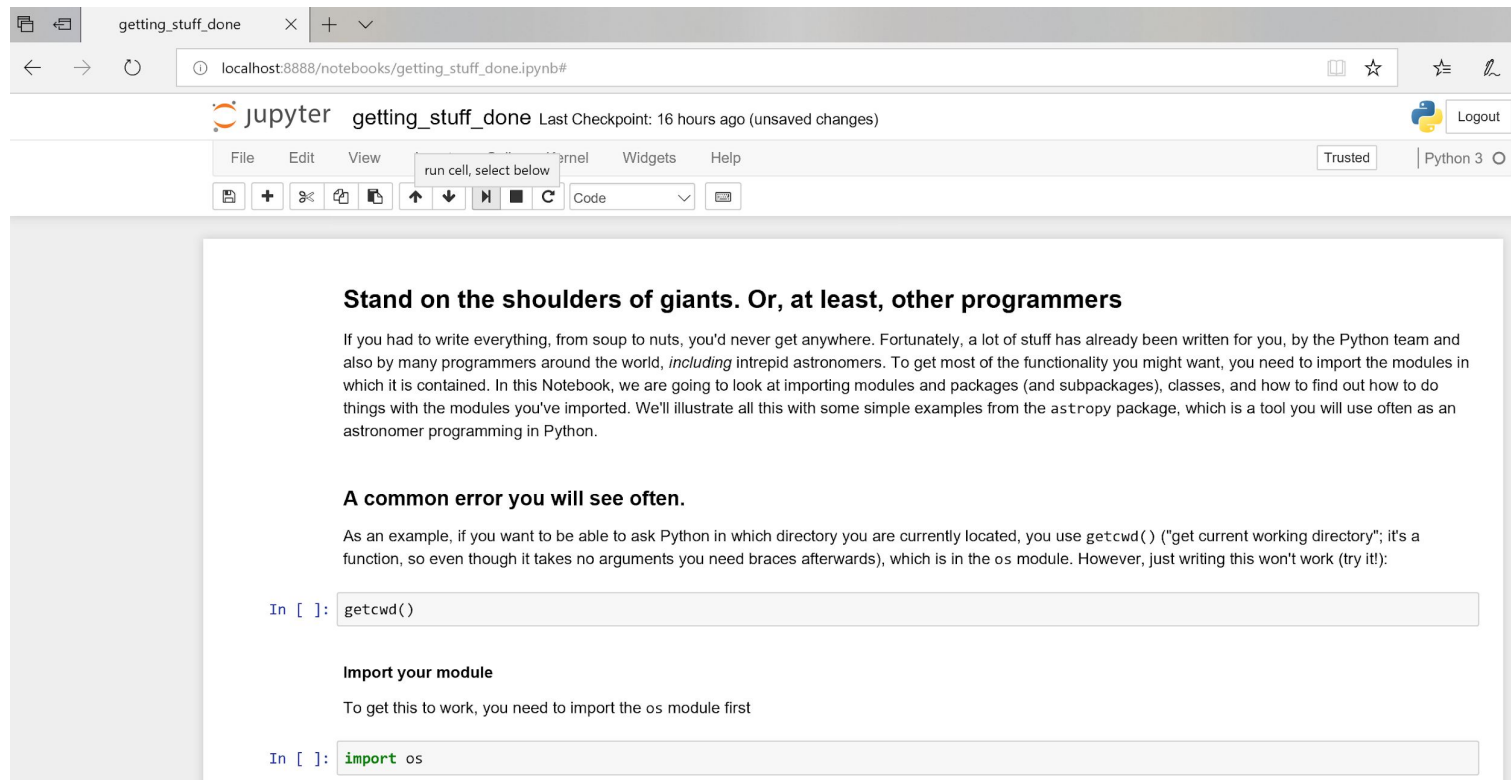
Most Python code is in plaintext files which are sent to the interpreter. You can also type directly in an interactive environment.

Jupyter Notebooks are halfway between a script file and fully interactive; an excellent tool, particularly for demonstration and education, allowing you to enter code in your browser and have it sent to Jupyter's Python interpreter in batches

As described in the software guidance on the meeting website, start up your Docker container, find your Jupyter Notebook URL and enter it into your browser. You will see an index of available files, will be selecting one and getting started!

WARNING: Opening a notebook in more than one tab? Nearly always a bad idea!

A notebook page



getting_stuff_done

localhost:8888/notebooks/getting_stuff_done.ipynb#

jupyter getting_stuff_done Last Checkpoint: 16 hours ago (unsaved changes) Logout

File Edit View run cell, select below Kernel Widgets Help Trusted Python 3

Stand on the shoulders of giants. Or, at least, other programmers

If you had to write everything, from soup to nuts, you'd never get anywhere. Fortunately, a lot of stuff has already been written for you, by the Python team and also by many programmers around the world, *including* intrepid astronomers. To get most of the functionality you might want, you need to import the modules in which it is contained. In this Notebook, we are going to look at importing modules and packages (and subpackages), classes, and how to find out how to do things with the modules you've imported. We'll illustrate all this with some simple examples from the *astropy* package, which is a tool you will use often as an astronomer programming in Python.

A common error you will see often.

As an example, if you want to be able to ask Python in which directory you are currently located, you use `getcwd()` ("get current working directory"; it's a function, so even though it takes no arguments you need braces afterwards), which is in the `os` module. However, just writing this won't work (try it!):

```
In [ ]: getcwd()
```

Import your module

To get this to work, you need to import the `os` module first

```
In [ ]: import os
```

Suggested order:

1. `Basic_basics.ipynb` (the basic stuff)
2. `getting_stuff_done.ipynb` (imports, etc, astropy example)
3. `Loops_ifs_files.ipynb` (control-of-flow and file access)
4. `Functions.ipynb` (also contains Classes)
5. `Python2_and_Python3_Some_future.ipynb` (2 vs. 3 advice)
6. `Numpy_and_you.ipynb` (useful stuff; includes matplotlib)
7. `Faking_it_at_the_command_line.ipynb`
8. `More_activities.ipynb` (some fun activities)
9. `Psrcat_fun.ipynb` (more fun activities)

Part II: Some additional topics

Here we cover some practices, particularly relating to:

- Your design
- How you produce your software
- How you run your software
- Some good practices

Often your immediate aim is “something that works quickly and I’ll fix it up later”.

Part II: Some additional topics

Here we cover some practices, particularly relating to:

- Your design
- How you produce your software
- How you run your software
- Some good practices

Often your immediate aim is “something that works quickly and I’ll fix it up later”.

Often you end up using the code for years as it becomes a Frankensteinian mess

Software *engineering* practices help resolve this. This isn’t a complete list of them!

Part II: Some additional topics

Here we cover some practices, particularly relating to:

- Your design
- How you produce your software
- How you run your software
- Some good practices

This is an aside, but I primarily use Windows for my personal machines, so if you want advice about working like that (Bash on Windows, conemu, Docker on Windows, etc, I'm happy to talk about it). My code mostly runs on Linux.

Running your code as a script

In many cases, your production code will be called from the command line, like:

```
Python myscript.py --param1=arg1 --param2=arg2
```

`myscript.py` will be the master script that calls what is necessary, parsing the passed arguments and running the show

You would like `myscript.py` to be:

Extensible, easy to read, redistributable, robust, maintainable, etc. Apple pie and roses, all the good things.

What follows are some pointers to achieving it, some of which are Python-specific

Days of coding save hours of thinking*!

Software *requirements* are the things the software *has* to achieve; if it does not, the software has failed. Conventionally written as:

The X shall Y

Typically they are functional:

The calendar shall fill in the dates between start and end and pass to Z

Or they are nonfunctional/design constraints:

The system shall initiate in less than 10 seconds

But wait, that will take forever!

Requirements should be appropriate:

Number and depth of requirements requires judgement; often you can write them down at a high level very quickly, and sometimes that is enough

Requirements should be complete

If you *need* something else from the software, and it's not there, add it.

Requirements are like a contract:

If you don't achieve them, you failed in your design or coding. If you achieve them and the software isn't what you needed, your requirements were at fault.

This is my guide

Requirements should be appropriate:

Number and depth of requirements requires judgement; often you can write them down at a high level very quickly, and sometimes that is enough

Requirements should be complete

If you *need* something else from the software, and it's not there, add it.

Requirements are like a contract:

If you don't achieve them, you failed in your design or coding. If you achieve them and the software isn't what you needed, your requirements were at fault.

Meanwhile, over in industry...

There are many approaches to requirements. Some approaches are exhaustive, some are much lighter (agile, even!)

In an exhaustive approach, they may be identified by *Use Cases*, before design

In a more Agile approach, embodied in User Stories during development

In practice, the two approaches may be mixed. I like this sort of approach, using requirements to retire big risks and inform large designs, but leaving other requirements to implementation. The complexity may scale with team size.

Design and architecture

In most cases, I wouldn't worry about the difference between the two, and think of it all as design (strictly design deals with functional requirements and architecture with non-functional requirements)

Just design the code first!

There are plenty of tools for this, like UML, but a sketch of packages, modules and classes with appropriate names and communication flow is a good start.

Interfaces are important! They allow modular development, and collaborative programming; you need to decide what elements do, but also what they take in and what they put out.

More on interfaces

Specifying and agreeing interfaces is a big part of multi-programmer design

If the behaviours of module A and module B are known, with parameters and return values also specified, you can develop modules A and B independently

Even when developing solo, this is helpful

This also enables meaningful unit tests

A nice basic design will make the interfaces easier to write out

Managing per-run arguments in Python

You have three main options for getting per-run settings into your code:

- User input
- Command-line arguments
- Configuration file

We will not mention re-writing “magic values” into your code every time you want different behaviour. Other than for very basic prototyping and debugging, DON'T.

Managing per-run arguments in Python

You have three main options for getting per-run settings into your code:

- User input
- Command-line arguments
- Configuration file

We will not mention re-writing “magic values” into your code every time you want different behaviour. Other than for very basic prototyping and debugging, DON'T

I mean it! Don't do that!

User input

This is not all that common, but you can have variables filled by the user at run-time using Python's `input` (formerly, in Python 2, `raw_input`)

```
dms = input("How many dms? ")
```

```
find_pulsars(dms)
```

I haven't had much use for it, but one can imagine cases where user confirmation configuration at keyboard might be helpful (eg, "do you want to proceed").

You are trusting the user to be sensible. You should test that their input makes sense (and don't try to run the input as literal Python with `exec`, it's dangerous)

Command-line arguments: `sys.argv`

Python can take arguments at the command-line, like:

```
python my_script.py --preserve my_file.txt
```

For simple command-line arguments, you can use `sys.argv`, which is a Python list of the things passed at the command line (doesn't care about `--preserve` vs. `my_file.txt`, just their positions in the list)

```
import sys
```

```
myFile = sys.argv[2]
```

Why 2, not 1? `sys.argv[0]` is the script name.

Command-line arguments with pizazz

Relying argument order is tricky, particularly when not all arguments required

Python has better options:

- `getopt.getopt(args, shortopts, longopts=[])` ← C-style parsing
- `getopt.gnu_getopt(args, shortopts, longopts=[])` ← more flexible
- `argparse` ← **strongly recommended**

It is worth taking the time to learn `argparse`; it might seem like a hassle at first, but it pays off in operational simplicity, and it gives more extensibility to your code

Managing arguments: config files

In some sense, command-line arguments tell the code *what to do*. In some cases, you want to specify a bunch of other parameters better-described as *how to do it*.

Instead of 10-20 command-line args, set these in a separate configuration file.

Options for this include:

- A .py file containing a dictionary (harder to edit, but parses with `import`)
- YAML (like Linux config file) or other simple configuration file format
- JSON or XML document (commonly used for holding many key-value pairs)

Python has libraries for parsing YAML, JSON, XML and other formats

But I thought you said no magic values?

Editing a config file isn't magic values as it's not in the executable code, but it's not something you want to edit *every* run (unless runs are both long and complete)

You can also put in values which are specific to a given deployment of the code:

Local filepaths, URLs, databases, etc

Things which you don't change often, but *might* change (so, don't hardcode)

You can also put in authentication information, but: mustn't be included in any versioning repository, can't be readable by anyone but the application identity. This is sometimes necessary, but needs to be done carefully.

Modularity: one simple trick!

It can make debugging easier if every module can also be run, with some default settings, from the command line. You can do this by adding:

```
if __name__ == '__main__':  
    entry_function(some_values)
```

When you run `python myscript.py` from the command-line, the `__name__` attribute is set to `__main__`, so whatever code you write after the `if . . :` will execute; you'll probably then call the main entry function in the module, or perhaps also write in some argument parsing into the `if` block to decide what to do.

Code standards

Many organisations have coding standards (such as Google's)

The default Python style guide is PEP8 (and PEP257 for docstrings)*

PEP8 highlights include naming conventions, layout rules, programming advice

Whatever style you do use, consistency really enhances readability

Future you will also be thankful!

I *strongly* recommend that you read PEP8 (even if you don't always implement it)

PEP means "Python Enhancement Proposals". There are many, but some are canonical

Comments and docstrings

Comments should always be *useful* (ie, not stating the obvious, or wrong)

Sometimes that will mean removing them after some time, as code evolves

Comments that say *why* are often particularly useful

Functions intended for external use (via `import`) should have a *docstring*

You indicate, but enforce, non-external functions with a leading underscore in the function name, like `_my_function`

Docstrings describe functions' behaviour, discovered with `help`. Read PEP257!

Errors

Python has some decent error-handling including based on:

- `try` (delineates the scope of the code where an error may occur)
- `except` (like “catch” in other languages)
- `else` (if the exception isn’t caught by the `excepts`)
- `finally` (happens when exiting the `try` block)

Put the most specific errors first in the error-catching stack. You are using `except` to catch things you know might happen, and handle them; avoid just saying `pass` in an `except` just to keep your code running!

```
try, except, (else), finally
```

Like this (say a file has been read into a dictionary):

```
try:
```

```
    var = my_dict[varname]
```

```
except KeyError:
```

```
    print (str(varname) + "is not available!")
```

```
finally:
```

```
    file.close()
```

Exceptions and Pythonicness

If you want to know a file exists, you could:

```
if os.path.isfile(file_path):  
    do_something
```

However (for various reasons), this is the preferred “Pythonic” way:

```
try:  
    file.open(file_path, 'r'):  
  
except IOError as e:  
    handle_this_situation()
```

Installing Python software with `setup.py`

Python code will generally come with a `setup.py` installation script. Invoke it like:

```
Python setup.py install/develop*
```

You should write your own `setup.py` for your own code. See the Python documentation on “Writing the setup script”

`setup.py` will put your code into `site-packages`, so you can import it

Your `setup.py` may get fairly complex, but often it'll be simple.

*`develop` will install the software as `.py` files, not byte-compiled `.pyc` files

Using `pip`

`pip` is the python package manager; you probably got it with your Python distro

`pip` is very useful and can also be used to *uninstall* the code

`pip` installs from `setup.py`, or a wheel file (`.whl`); run it in the same directory

`pip` can also install packages from the Python Package Index, `PyPy`:

```
pip install simplejson
```

This will find the package on `PyPy` and install it for you (will also do any required compilation, as with `numpy`)

conda. What is it? Why use it?

conda is a free and complete Python distro, useful for scientific computing

We are using conda in our Docker container

conda has its own package manager, called (surprise!) conda

You don't *need* to use conda--I typically don't, on my own machines--but it means that you can avoid a lot of build issues for some packages (eg, `scipy`)

There's a smaller conda called `miniconda`. You can just add what you want

To install packages built for conda, you can just `conda install mypackage`

`pip` **VS** `conda`

If `conda` is your default Python (or else you have activated it), they'll both work pretty much the same, but `conda` will get packages from the conda repository and `pip` will get them from PyPi

`conda` often does better handling cases where there are non-Python components and dependencies (Pip can be used to install, say, `scipy`, but `conda` has one ready for you; this is particularly important on MS Windows)

You can use `conda` and `pip` side-by-side; they both install Python to the same place. `conda` is both a distribution and an installer with access to that `conda` distribution; `pip` is an installer with access to the PyPi archive.

Virtualisation with `virtualenv`

A `virtualenv` is a walled garden in which your code lives. You make them like:

```
virtualenv env (or whatever you want to call it; env is standard)
```

This will create a folder `env`, which contains a Python interpreter, some scripts in a folder called `bin`, and a `site-packages` folder (initially empty). You activate it via the appropriate `bin/activate` script. On Linux and OSX:

```
source bin/activate (On windows run the .ps script)
```

Your prompt now shows you you've activated the `virtualenv`. Deactivate like:

```
deactivate
```

Using virtualenv

When you type `python` at the command-line, it'll use the `virtualenv` Python and `pip` and will look first in the `virtualenv`'s version of `site-packages`

This lets you have Python libraries that aren't the same version as the host Python

You can also get that Python and `pip` without activating by referring to the full path of your `virtualenv` versions inside `bin`

```
path/to/env/bin/python
```

This is important sometimes, such as when calling `python` from `cron` (also, environment variables need to be set in the `cron` job itself, not in `activate`)

Redistribution

You should use the `virtualenv`'s `pip` to install Python libraries. When done, make a file list of the dependencies:

```
pip freeze > requirements.txt
```

Now you can distribute your code (and its `setup.py`!) and the requirements file, and then users can grab the dependencies with:

```
pip install -r requirements.txt
```

If the requirements are on PyPi, users can make their `virtualenv` and install everything into it. NB: exclude `env` from your repository, eg, with `.gitignore`

Virtualisation with conda

You can do the same sorts of things with conda itself:

```
conda create --name <envname> python=<version>
```

From here on in, it's a similar story to using `virtualenv` itself. The Docker container we distributed has a conda Python2 environment you get with:

```
source activate python2
```

Now you get the strengths of conda from before. NB: instead of `pip freeze`:

```
conda list --export > package-list.txt
```

Briefly, testing

Quality Assurance is a big topic, but testing is a critical part. Tests let you avoid a bunch of errors, particularly as your code is undergoing further development.

Python's basic test functionality in the `unittest` module; `pytest` more complete

If you don't write tests as you go--even starting with them--you'll never get to it!

Tests are important if you want to incorporate your code into a bigger project, to keep developing your code or to redistribute it widely. People like to see tests!

Basically, tests check whether code does what it's supposed to. They're only as good as your selection of the tests.

Python for high performance

By default (because of the Global Interpreter Lock of CPython) Python is single-threaded. Even if you `import threading`, it still runs on one core

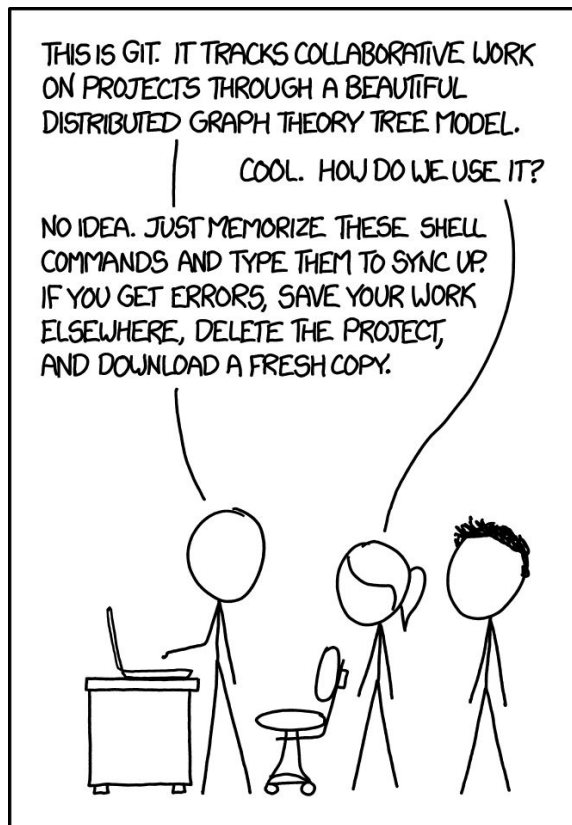
You *can* use `import multiprocessing` to get multi-core usage. Be careful!

Often, high speeds are obtained by *wrapping* compiled code, as `numpy` does. You can write your own Python wrappers (using `Cython`, `swig`, `PyCUDA`, `f2py`, etc)

MPI functionality exists in `mpi4py`

You can time your code with `timeit` (for small run-times) or `time`

Finally, use version control (at last! A picture!)



Version control saves much heartache

The most common system in astronomy is `git`, but `mercurial`, `subversion`, `cvs` (old!) also used

Github has a pretty good web interface to a free-to-use (for public or educational use) repo host

Commit fairly often. You generally don't need to push so often.

Make `README`, `LICENSE` and `.gitignore` files

In conclusion

Python is huge--keep learning it! Also, be nice to people who don't know as much.

Good Software engineering is habitual, even if you don't get the habits all at once

A little extra time spent doing it right is rewarded down the road, many times over, but your judgement of what effort “doing it right” is worth will evolve