

Good coding practices (#GCP)

Tips to make your code better for you and others

Vanessa Moss

ASTRON

Introduction

- **My title:** observatory astronomer, based at ASTRON in the Netherlands
- Previously **affiliated** with University of Sydney, CSIRO, Sydney Observatory in Australia
- **NCSS Challenge** during 2nd year uni (Python), previously Matlab or more basic computing
- Astronomers writing code vs. programmers developing for astronomy (**different skills**)
- **Code-savvy astronomers** help bridge the gap!

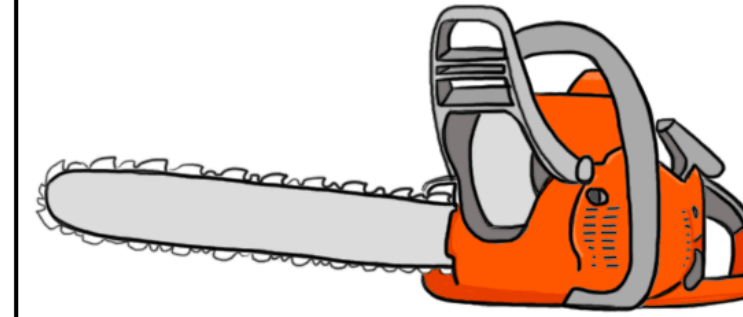


This talk

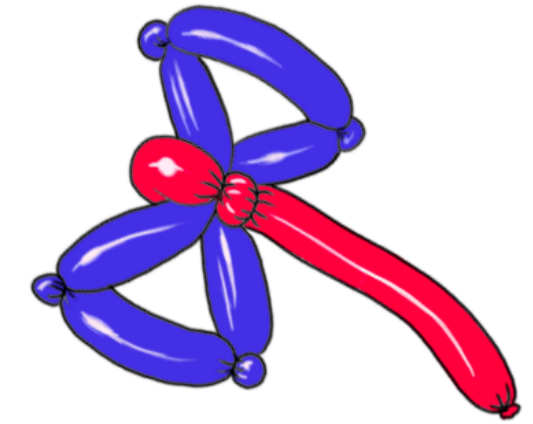
- **Goal:** to introduce you to some guiding principles that will help make your code better
- First half from a talk by **Zheng Meyer-Zhao**, given at last year's school, second half is new
- The last 60 min of this talk will be spent on an **exercise** getting you to put some good coding practices into use
- There is no ***one*** way to code things*, but there are definitely good ways and bad ways to approach programming!

CODE PROGRESSION

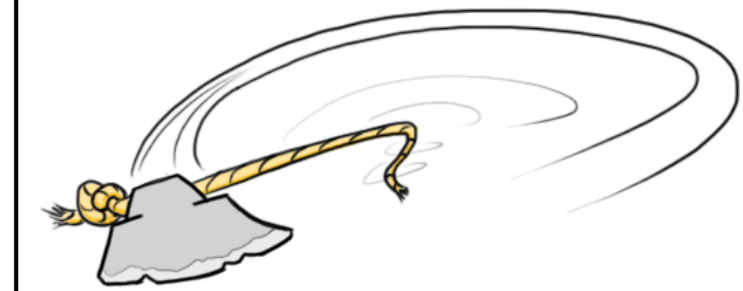
ARCHITECTURE



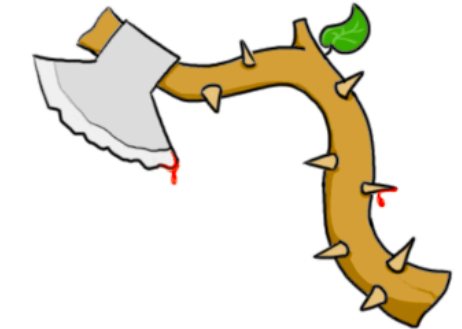
PROTOTYPE



PILOT



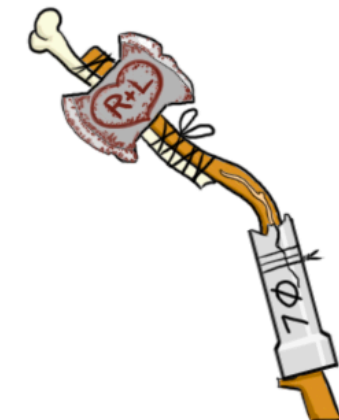
BETA



RELEASE



LEGACY



DOCUMENTATION



MONKEYUSER.COM

Overall guidelines

With credit to Zheng's 2018 talk

ASTERICS-OBELICS 2019 (V. Moss, [@cosmicpudding](#))



1. Write programs for people

- *Write programs for people, not computers*
- A program shouldn't need people to hold many facts in their head at once to understand it
- *good clear comments help with this!*
- **Variable names** should be consistent, distinct and meaningful to aid with reading the code
- Keep a **consistent** code style/formatting (there are various styles, but consistency is key!)

```
import os
import sys
from astropy.io import ascii
import matplotlib.pyplot as plt

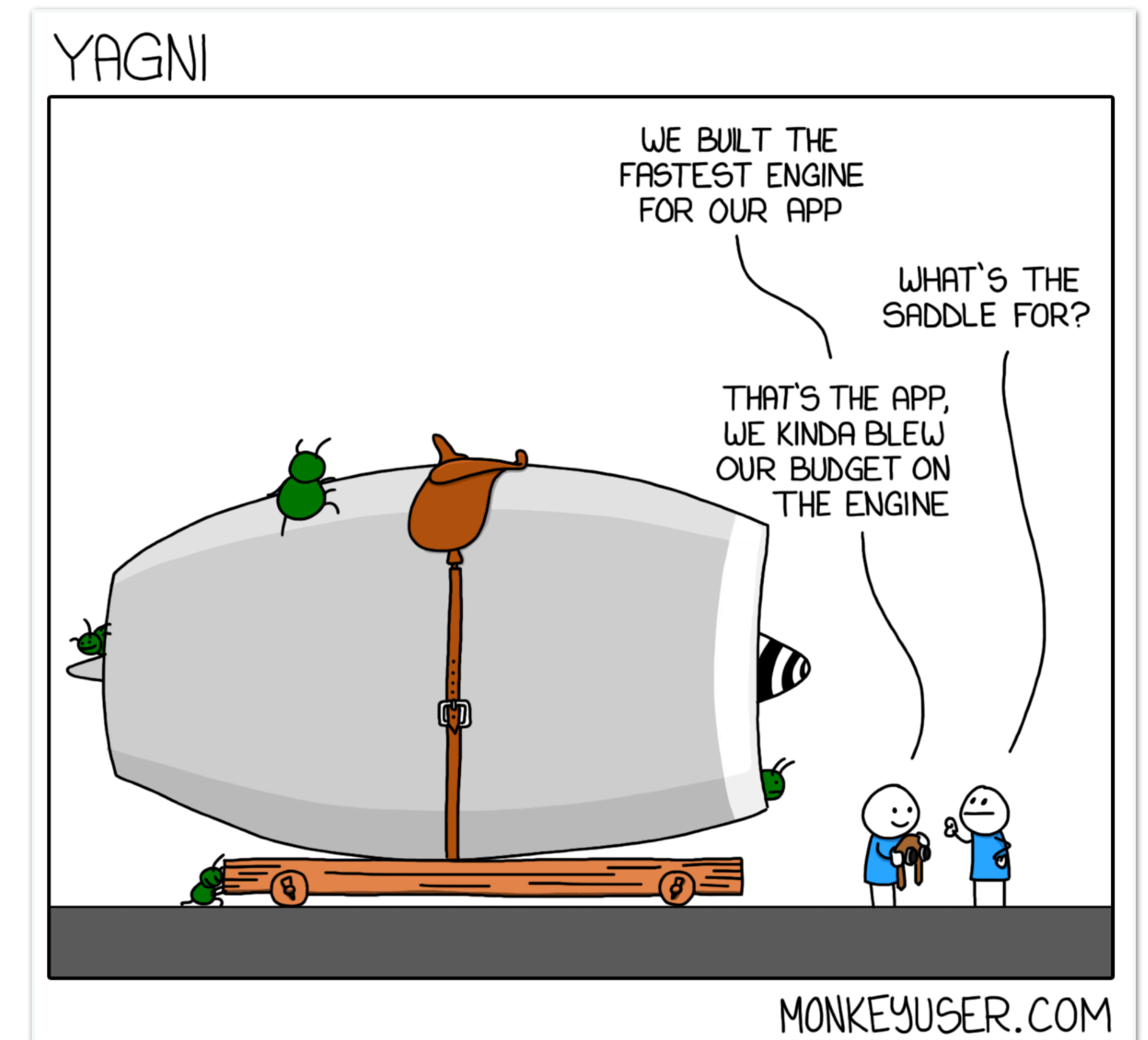
# Read RA/dec values from table
table_data = ascii.read('table.txt')
ra_list = table_data['ra']
dec_list = table_data['dec']

# Plot the results as a scatter plot
fig, ax = plt.subplots(figsize=(10,6))
ax.scatter(ra_list, dec_list)

# Save the results in a PNG
plt.savefig("radec_plot.png", dpi=200,
bbox_inches='tight')
```

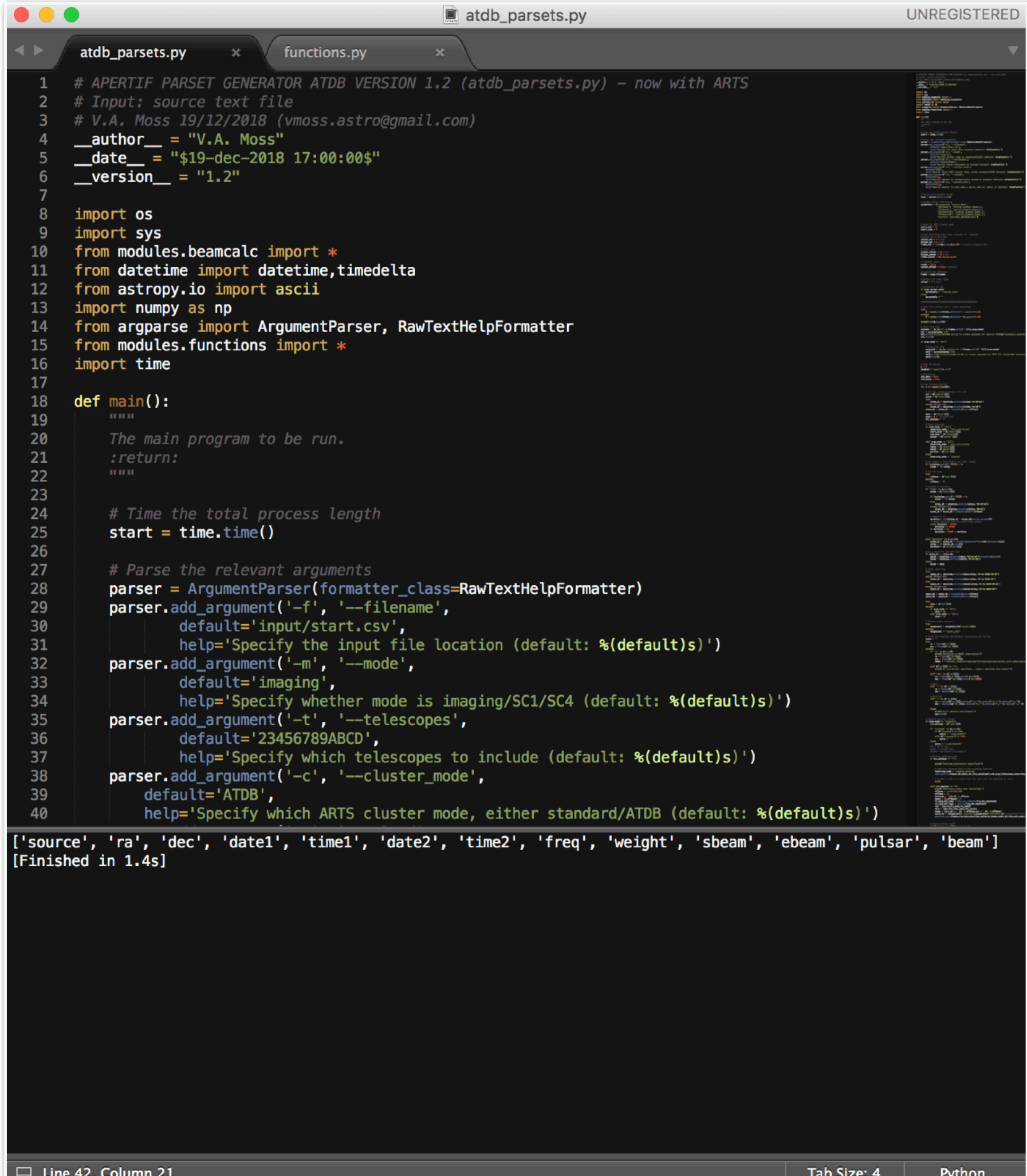

2. Develop pragmatically

- *Approach development of a new script or package in a pragmatic way*
- The amount of effort should be **proportional** to the scope of the code, and its legacy value
- **Fast** code, **good** code, or **cheap** code: pick **two**
- Whatever time you think the code will take to develop, **double it** (at least)
- Best learnt over time, but invest the **right amount of effort** for different parts of the code



3. Let the computer do the work

- *Make use of the tools that exist to be as efficient as you can*
- Make the **computer** repeat tasks (e.g. cron job)
- Use **IDE/build tool** to make workflow efficient
- "If you've only got a hammer, everything looks like a nail" = use the **right tool** for the job
- **Pycharm/Sublime**, etc are good for quick runs of code, **iPython** for interactive commands



```
# APERTIF PARSET GENERATOR ATDB VERSION 1.2 (atdb_parsets.py) - now with ARTS
# Input: source text file
# V.A. Moss 19/12/2018 (vmoss.astro@gmail.com)
__author__ = "V.A. Moss"
__date__ = "$19-dec-2018 17:00:00$"
__version__ = "1.2"

import os
import sys
from modules.beamcalc import *
from datetime import datetime, timedelta
from astropy.io import ascii
import numpy as np
from argparse import ArgumentParser, RawTextHelpFormatter
from modules.functions import *
import time

def main():
    """
    The main program to be run.
    :return:
    """

    # Time the total process length
    start = time.time()

    # Parse the relevant arguments
    parser = ArgumentParser(formatter_class=RawTextHelpFormatter)
    parser.add_argument('-f', '--filename',
                        default='input/start.csv',
                        help='Specify the input file location (default: %(default)s)')
    parser.add_argument('-m', '--mode',
                        default='imaging',
                        help='Specify whether mode is imaging/SC1/SC4 (default: %(default)s)')
    parser.add_argument('-t', '--telescopes',
                        default='23456789ABCD',
                        help='Specify which telescopes to include (default: %(default)s)')
    parser.add_argument('-c', '--cluster_mode',
                        default='ATDB',
                        help='Specify which ARTS cluster mode, either standard/ATDB (default: %(default)s)')

    ['source', 'ra', 'dec', 'date1', 'time1', 'date2', 'time2', 'freq', 'weight', 'sbeam', 'ebeam', 'pulsar', 'beam']
    [Finished in 1.4s]
```

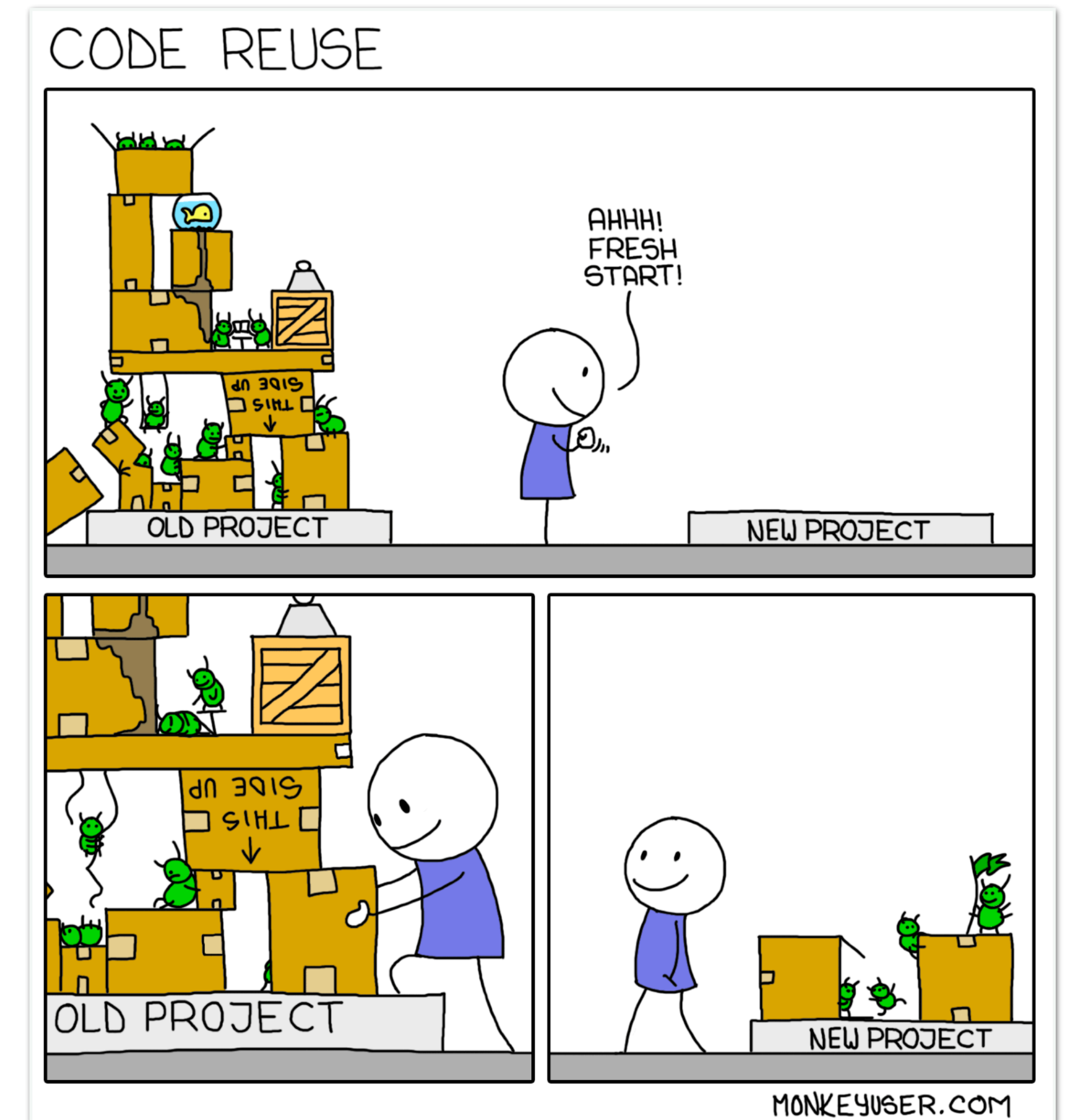

4. Make incremental changes

- *Small constant changes are the best way to prevent destabilising your code*
- Work in **small steps**, and if your code is in version control, commit these changes often
- Which also means: **use version control!**
- Version control is **critical** for code shared with others, but is also incredibly useful for your own code for history, backups, deploys, etc...

Wrong variable name from dict	5f38af6	<>
Use new function to identify calibrators from a previous day	3cec775	<>
Add previous day functionality	cdd7276	<>
Add archived status check to calibrators	df89202	<>
Commits on Mar 26, 2019		
Add print of return to verify contents	38b8a1c	<>
Commits on Mar 25, 2019		
Cannot know success for errored	d4aefc8	<>
bracket typo	01e83a4	<>
Also for the one case	db36daf	<>
Accept new pipeline response, add slack id functionality	ca0e415	<>
Test tagging people in Slack messages +1	19a095a	<>
Test tagging people in Slack messages	8002ed8	<>
Commits on Mar 22, 2019		
Deal with case when one calibrator is a None type	0b57878	<>
Commits on Mar 20, 2019		
Set any other cal to none if only length one, when others are several	244946f	<>
Missing a break for calibrators before target	c2533d3	<>
Remove run one limitation, only consider recent datasets, and not cha...	8d81d5d	<>

5. Don't repeat yourself

- *Repetitive code is usually redundant, and is really difficult to both read and debug*
- If you find yourself **copying and pasting code**, then this usually means it would make a good function or module
- **Re-use** code instead of rewriting it: identify code that you need often, and make that into easily-accessible functions (when modular)
- That doesn't mean you shouldn't look for ways to **improve** the re-used code though!



6. Plan for mistakes

- *If it works the first time, be suspicious*
- No matter how carefully you program, there will always be **bugs**, often in unusual places
- For debugging, **print statements*** are extremely useful to check if the code does what you think it should be doing (*better: logging)
- Make use of **existing libraries** for testing (e.g. off the shelf libraries)
- Use bugs as good tests (e.g. **unit tests**)

A simple unit test

```
1 # test_port1.py
2
3 import unittest
4 from portfolio1 import Portfolio
5
6 class PortfolioTest(unittest.TestCase):
7     def test_buy_one_stock(self):
8         p = Portfolio()
9         p.buy("IBM", 100, 176.48)
10        assert p.cost() == 17648.0
```

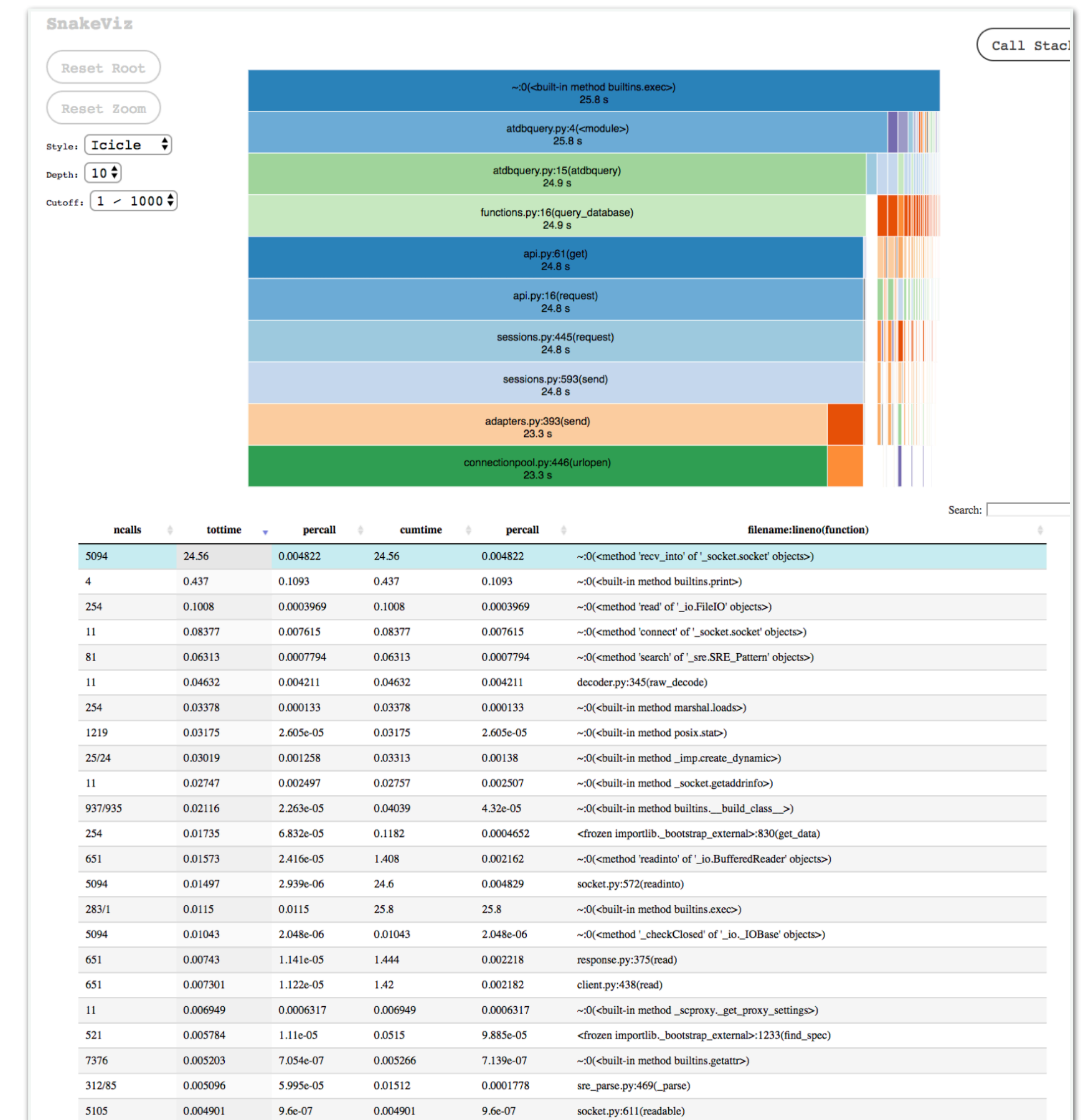
```
1 $ python -m unittest test_port1
2 .
3 -----
4 Ran 1 test in 0.000s
5
6 OK
```

bit.ly/pytest0

@nedbat

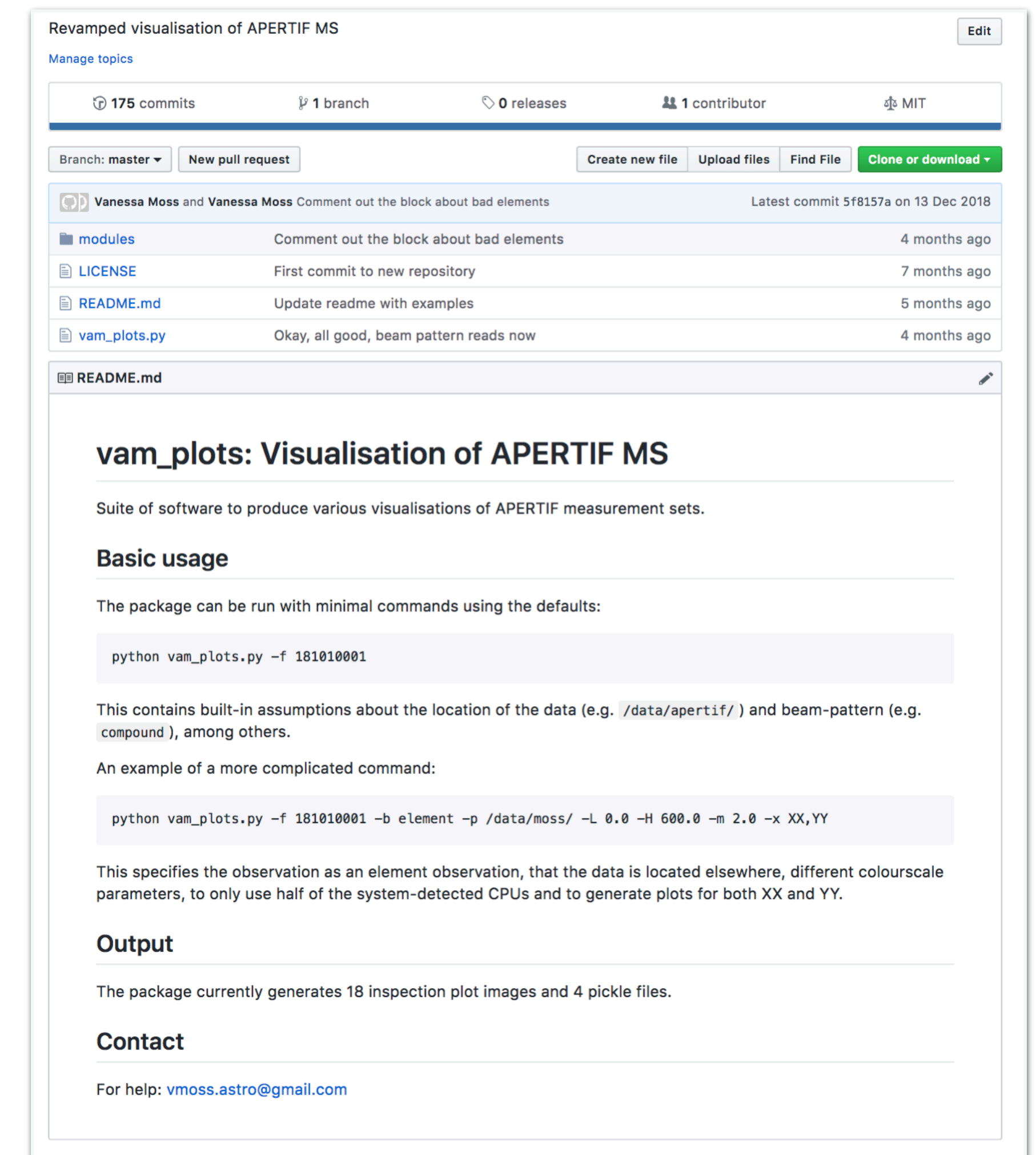
7. Make it work, then optimise

- *You should optimise software after it works correctly, based on the use case required*
- The first goal should always be to get the code **working** - it is rarely the case that the design from the beginning limits optimisation
- Use profilers to identify bottlenecks e.g. **SnakeViz** (<https://jiffyclub.github.io/snakeviz>)
- Write code in a **high-level** language, and avoid re-inventing the wheel where possible
- But also optimise **as you go**, where you can!



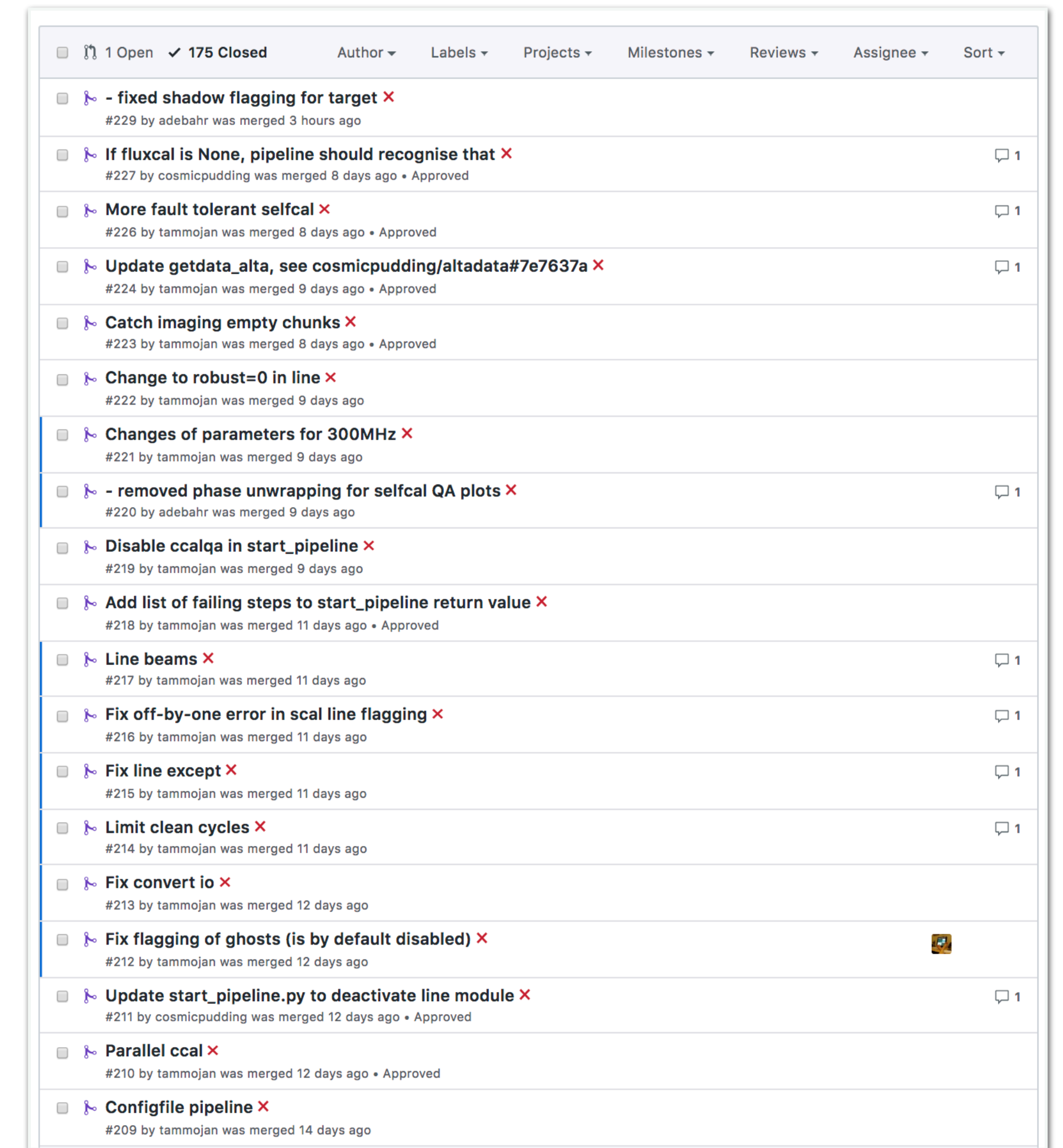
8. Document what you've done

- *Don't trust your future self to remember what you coded and why*
- Documentation should favour design + purpose over mechanics, but **all are important**
- **Block comments** above code can be helpful for keeping track of what sections do (*docstrings*)
- **Revisit** code even after it works (plan this!), and **refactor** to make the flow clearer/better
- Keep documentation **close** to the software



9. Collaborate

- ***Your code will always find improvements when others try to understand or use it***
- Use pre-merge code reviews (e.g. pull requests) for a **double-check** on what you have changed
- **Pair-programming** can help a lot when trying to implement new or complicated code
- Use **issue-tracking tools** to keep track of progress (e.g. JIRA, Redmine, Github issues)
- Ask people to try **using your code**, in order to find out what you should change or improve



Specific coding tips

These will be important in the exercise!

ASTERICS-OBELICS 2019 (V. Moss, [@cosmicpudding](#))



Argparse for argument handling

- Python allows you to pass variables as arguments after the code, e.g. **sys.argv[1]**
- Argparse is a **better way** to handle input variables into your scripts, giving you more control over their types and format: <https://docs.python.org/2/howto/argparse.html>
- It also enables users of the script to get an **overview** of what each parameter means: e.g. `python vam_plots.py --help`
- As part of the **exercise**, you will add a (long) list of arguments to an argparse formatter :)

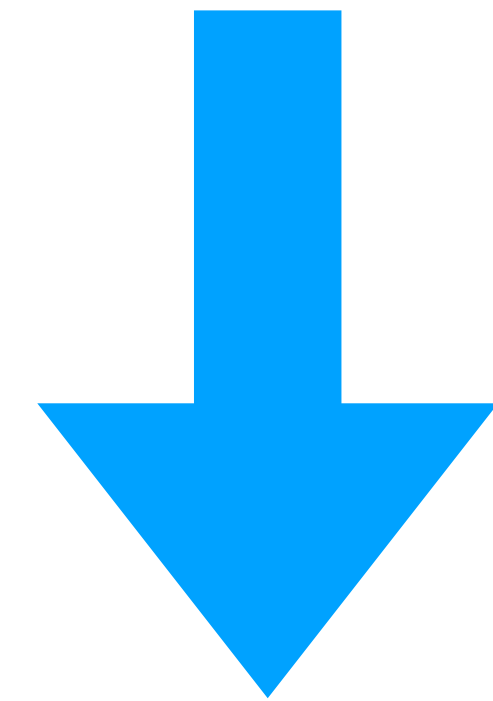
```
parser = ArgumentParser(formatter_class=RawTextHelpFormatter)
parser.add_argument('-p', '--path',
                    default='/data/apertif/',
                    type=str,
                    help='Specify path location of target folder (default: %(default)s)')
parser.add_argument('-f', '--folder',
                    default='181130004',
                    type=str,
                    help='Specify path location of target folder (default: %(default)s)')
parser.add_argument('-L', '--low',
                    default=200.,
                    type=float,
                    help='Specify amp colorscale low-value (default: %(default)s)')
parser.add_argument('-H', '--high',
                    default=1000.,
                    type=float,
                    help='Specify amp colorscale high-value (default: %(default)s)')
parser.add_argument('-PL', '--pmin',
                    default=-90.,
                    type=float,
                    help='Specify phase colorscale low-value (default: %(default)s)')
parser.add_argument('-PH', '--pmax',
                    default=+90.,
                    type=float,
                    help='Specify phase colorscale high-value (default: %(default)s)')
parser.add_argument('-r', '--rfflag',
                    default='< 1000MHz',
                    type=str,
                    help='Specify properties of RFI-flagging (default: %(default)s)')
parser.add_argument('-b', '--beam_pattern',
                    default=None,
                    type=str,
                    help='Specify whether element or compound beams (default: %(default)s)')
parser.add_argument('-m', '--multi_factor',
                    default=1.0,
                    type=float,
                    help='Specify what factor to divide available CPUs by (default: %(default)s)')
parser.add_argument('-x', '--pol',
                    default='XX',
                    type=str,
                    help='Specify which polarisation/s to image (default: %(default)s)')
parser.add_argument('-t', '--file_type',
                    default='png',
                    type=str,
                    help='Specify what file-type save images as (default: %(default)s)')
parser.add_argument('-s', '--slices',
                    default=False,
                    action='store_true',
                    help='Specify whether to use the time slices method (default: %(default)s)')

# Parse the arguments above
args = parser.parse_args()
```


Make the code importable

- **Scripts** are useful, and by default what we tend to write for uses in astronomy (e.g. a linear flow to do a specific task, such as make a plot)
- If you make the bulk of your code **a function** (**skyviewbot()** for example), you can use it as a function as well as running it with default settings as normal Python code
- This means the functionality can then be called by another script, as well as being used on its own - so you've made your code **importable**
- This is done **by default** in the exercise later

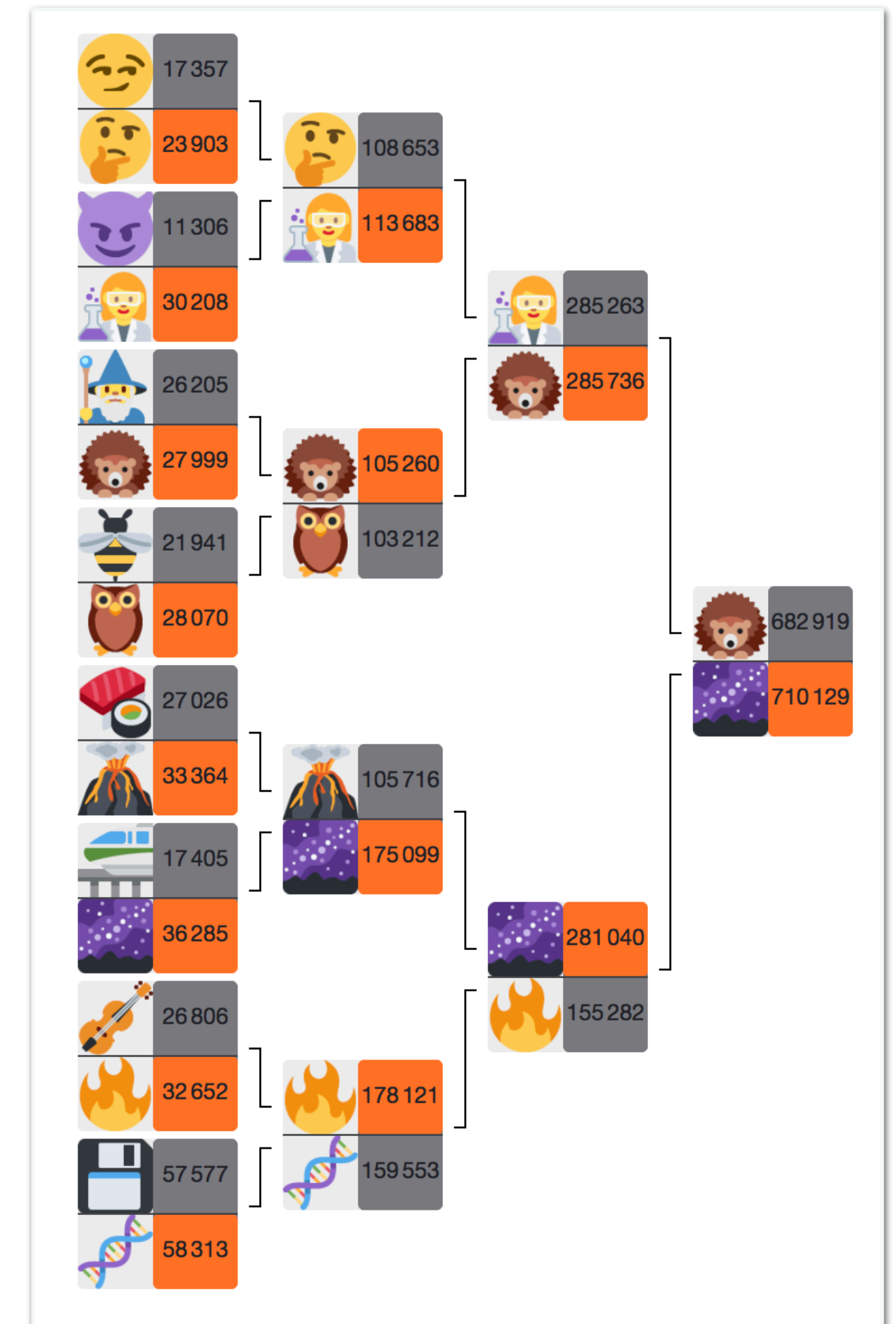
```
if __name__ == '__main__':  
    skyviewbot()
```



```
def skyviewbot():  
    ...
```


Modularisation and functions

- Whenever you use some calculation or more complicated code **more than once**, you should make it into a function
- Functions enable **much more flexibility** because you break down the "function" of that command into its input parameters
- Often you can put the functions at the top of the script, but when there are many, it is neater to store them in a dedicated **functions.py** or so
- You'll turn code into a function in the **exercise**



Docstrings

- Inline comments and block comments are nice for someone reading the code, but they don't help you get an **overview** of the whole code from the outside
- **Docstrings** are a good way to include an overview of functionality, arguments, and return variables in a structured way
- For the **exercise**, we recommend using the **Google docstring conventions**: https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html



```
def call_skyview(field, survey, pos, fov, coord, proj='Car', pix=500):
    """Call Skyview to download data from a survey based on input parameters

    Args:
        field (str): name of the field, used in naming the output file
        survey (str): name of survey, from https://skyview.gsfc.nasa.gov/current/cgi/survey.pl
        pos (float,float): position coordinates as a tuple
        fov (float): FOV in degrees
        coord (str): coordinate system (e.g. Galactic, J2000, B1950)
        proj (str): projection of image. (e.g. Car, Sin)
        pix (int): pixel dimensions of image (e.g. 500)

    Returns:
        str: name of resulting fits file

    Examples:
        >>> call_skyview('pks1657-298', 'dss', (255.291,-29.911), 5, 'J2000')
        'skyview_pks1657-298_dss.fits'
        >>> call_skyview('B0329+54', 'nvss', (144.99497,-01.22029), 0.5, 'Gal')
        'skyview_B0329+54_nvss.fits'
    """
```

Exercise: SkyviewBot

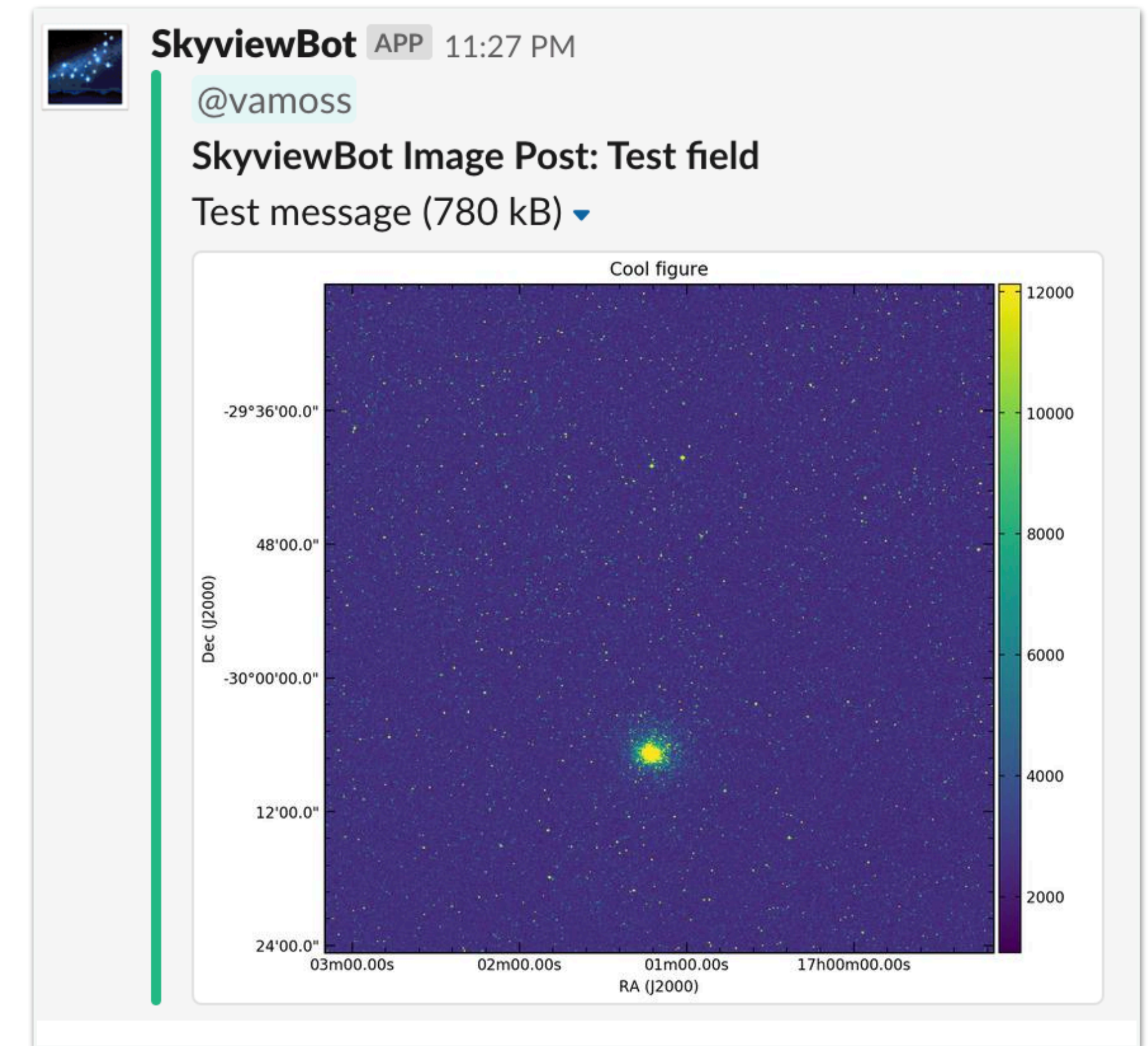
Let's make some code better using #GCP

ASTERICS-OBELICS 2019 (V. Moss, [@cosmicpudding](#))



Exercise: SkyviewBot

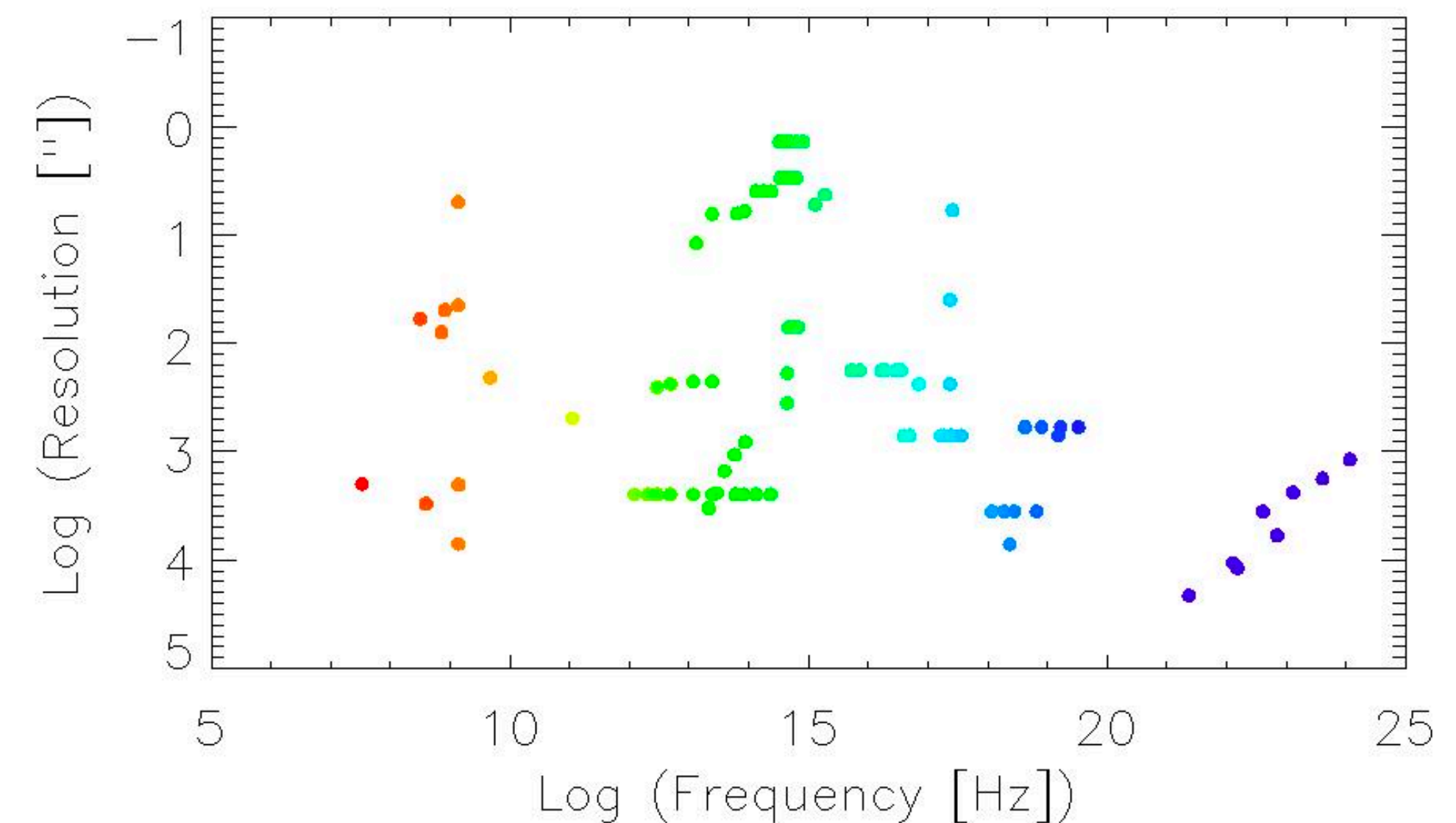
- **Goal:** to put together a Python script (using good coding practices!) that will fetch a survey region of your choice from NASA **Skyview**, image the FITS file using **APLpy** and then send your masterpiece to Slack using **Webhooks**
- **Dependencies:** APLpy, PyDrive, Java for Skyview (1.8.0_151, or use the included FITS)
- There is a base repository on Github: <https://github.com/cosmicpudding/skyviewbot>
- **Tomorrow:** package the resulting scripts as a proper pip-installable (T. Dijkema)



Have people run `git pull origin master` in School2019, then `conda env update -f environment.yml` to get the latest requirements (in particular aplpy and pydrive)

NASA Skyview

- **Skyview** is a VO tool that provides easy access to 100+ multi-wavelength surveys
- Easily accessible from Python using a thin wrapper around the **skyview.jar** file (provided you can get Java working!)
- The **astroquery** package also provides alternative access to Skyview (try this if you get stuck later with the .jar wrapper)
- See their documentation for more info: <https://skyview.gsfc.nasa.gov/current/docs/jar.html>



Sidequest: Java



- Downloading the [skyview.jar](#) file might just "work"... here are my settings:

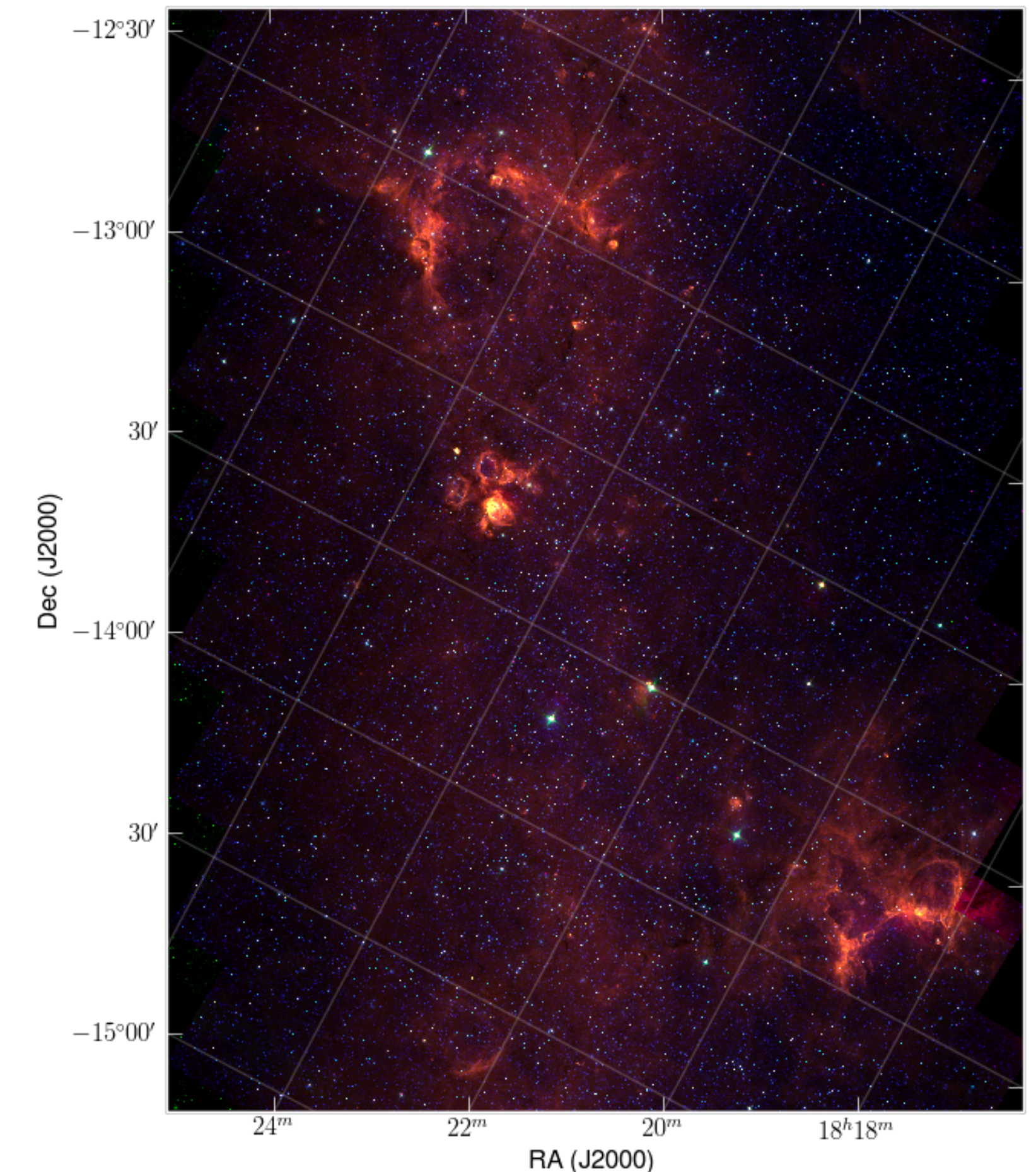
```
>> which java
/usr/bin/java
>> /Library/Internet\ Plug-Ins/JavaAppletPlugin.plugin/Contents/Home/bin/java -version
java version "1.8.0_151"
Java(TM) SE Runtime Environment (build 1.8.0_151-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.151-b12, mixed mode)
```

- You **may** have to add to your `.bash_profile` if the above isn't working:

```
# Java
export PATH="/Library/Internet Plug-Ins/JavaAppletPlugin.plugin/Contents/Home/bin:$PATH"
```


APLpy

- **Partner package** of astropy, developed by T. Robitaille and E. Bressert (v2 = Python 3.5+)
- Allows quick **visualisation** of astronomical FITS images, with coordinates and wrappers around standard Matplotlib functions
- APLpy website: <https://aplpy.readthedocs.io> or <http://aplpy.github.io>
- APLpy examples: http://aplpy.sourceforge.net/static_gallery_mirror (cloned? 🙋)
- Code for making a figure included in **exercise**



API: Google upload

- **Various APIs** exist to assist with automating otherwise manual actions such as uploading an image to Google Drive, sending batch emails or tweeting from Python (e.g. [@zootopialicense](#))
- Once you've made your image using APLpy, you can use **PyDrive** to automatically upload it via the Google Drive API to make it **web-accessible**
- Google Dev: <https://console.cloud.google.com>
- PyDrive: <https://pypi.org/project/PyDrive>
- We may have fun times **authorising**... TBD!

The screenshot shows the Google Cloud Platform console interface for configuring a 'Client ID for Web application'. The header includes the Google Cloud Platform logo, the project name 'SkyviewBot', and a search icon. Below the header, there are links for 'Client ID for Web application', 'DOWNLOAD JSON', and 'RESET SECRET'. The main content area displays the following information:

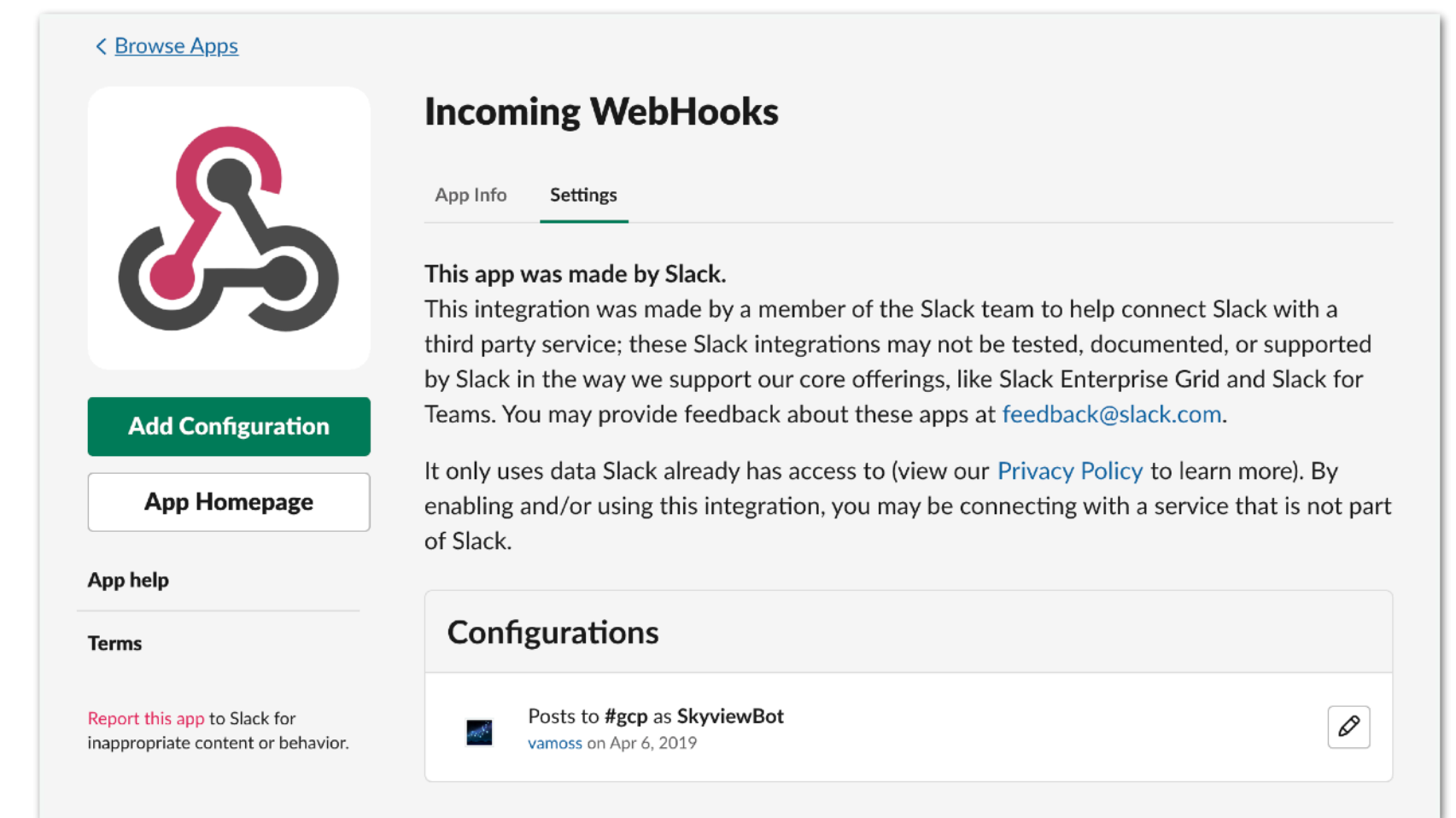
Client ID	1072399314963-jjc9nm6oquru4d534lvikl68i08gr9no.apps.googleusercontent.com
Client secret	uoQL3fZz9Kb1VxmSOx9jdUQp
Creation date	Apr 6, 2019, 9:44:16 AM

Below the table, there is a section for 'Name' with a dropdown menu showing 'SkyviewBotClient'. The 'Restrictions' section includes a link to 'Learn More' and a note that 'Origins and redirect domains must be added to the list of Authorized Domains in the OAuth consent settings.' The 'Authorized JavaScript origins' section has a text input field containing 'https://www.example.com' and a note that 'Type in the domain and press Enter to add it'. The 'Authorized redirect URIs' section has a list of four URIs: 'http://localhost:8080/oauth2callback', 'http://localhost:8090/oauth2callback', 'http://localhost:8080/', and 'http://localhost:8090/'. Each URI has a trash icon to its right. Below the list, there is a text input field containing 'https://www.example.com' and a note that 'Type in the domain and press Enter to add it'. At the bottom, there are 'Save' and 'Cancel' buttons.

autoskyview@gmail.com
astericsannecy

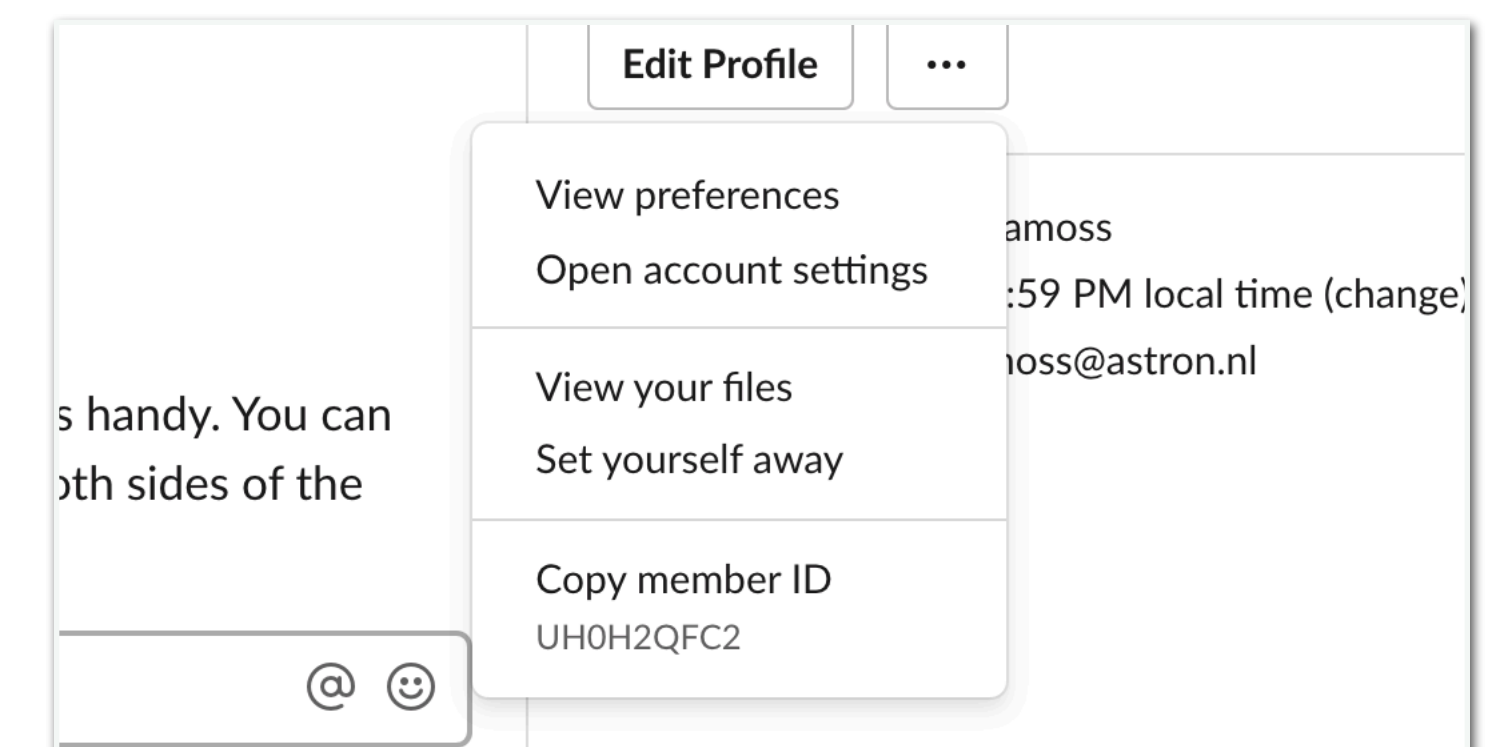
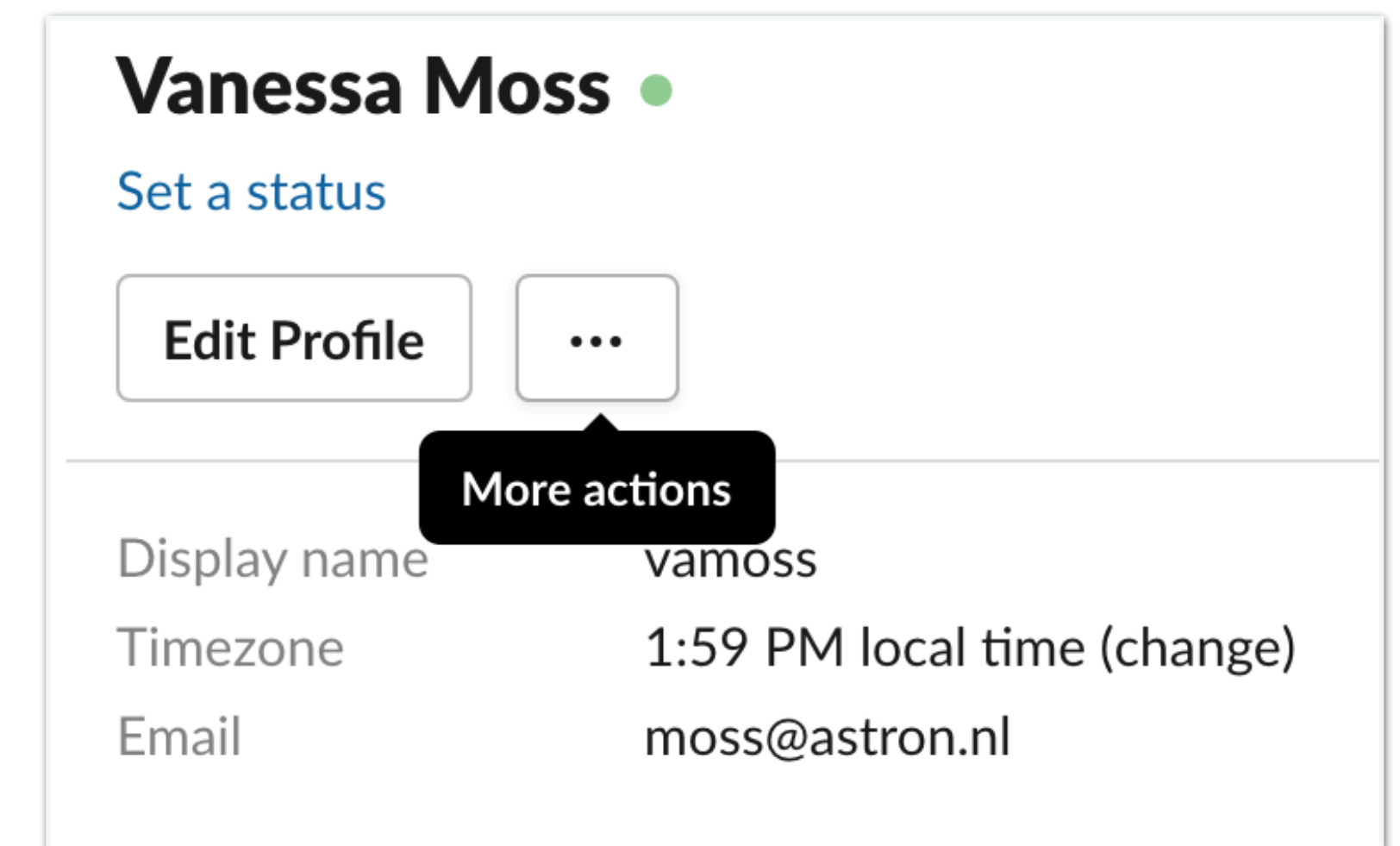
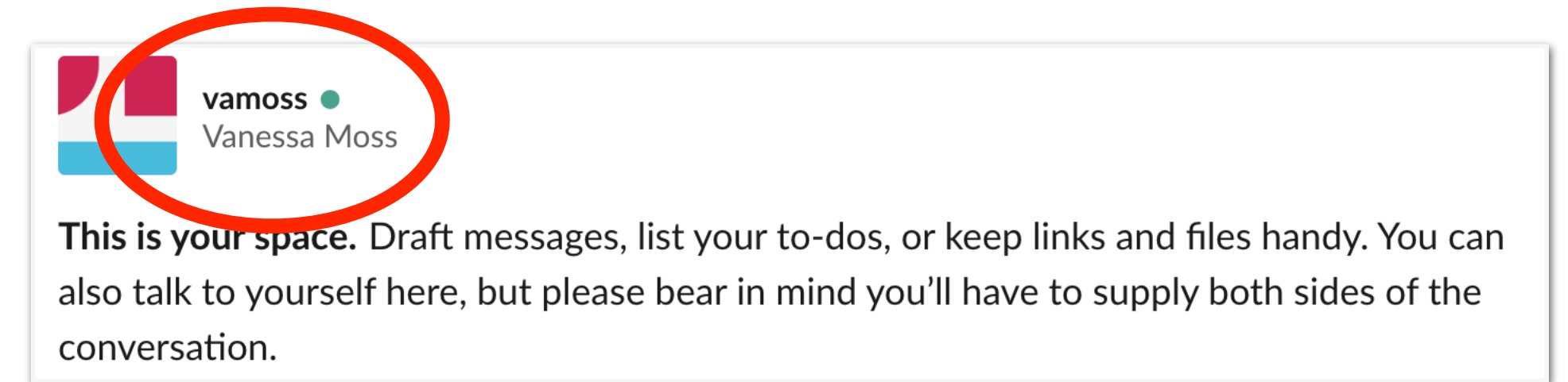
API: Slack web hooks

- **Slack webhooks** allow you to easily send automated messages to a channel on Slack
- Slack webhooks are a **custom integration** that you can add to any Slack, resulting in a URL that hooks into the particular channel
- In this case, we'll send your awesome images to the **ASTERICS Slack** channel for this talk: **#gcp**
- You can also do **outgoing** Slack webhooks
- See documentation here: <https://api.slack.com/docs/message-attachments>



Extra: your Slack ID

- You should tag **your Slack ID** with any posts that you make to the **#gcp** channel
- Go to **ASTERICS OBELICS Slack** (better: **app**): <http://obelics-school.slack.com>
- Click your name under "**Direct Messages**", then click the username in the DM window
- Click "...", and then "**Copy member ID**" - this is your unique Slack ID for the workspace, which can then be used to tag yourself in posts



Let's get started!

- **Github base repo** (see **README** file):
<https://github.com/cosmicpudding/skyviewbot>
- I've left **specific sections** for you to fill in from the skeleton code (e.g. add argparse) - read the comments to find out what to do!
- Feel free to take the code in **whichever direction you want**, and improve it in any way you see fit! e.g. more plot options, more elaborate Slack messages, error handling, etc
- You have **60 minutes** to complete this exercise, and the best image post* will win a **prize!**

