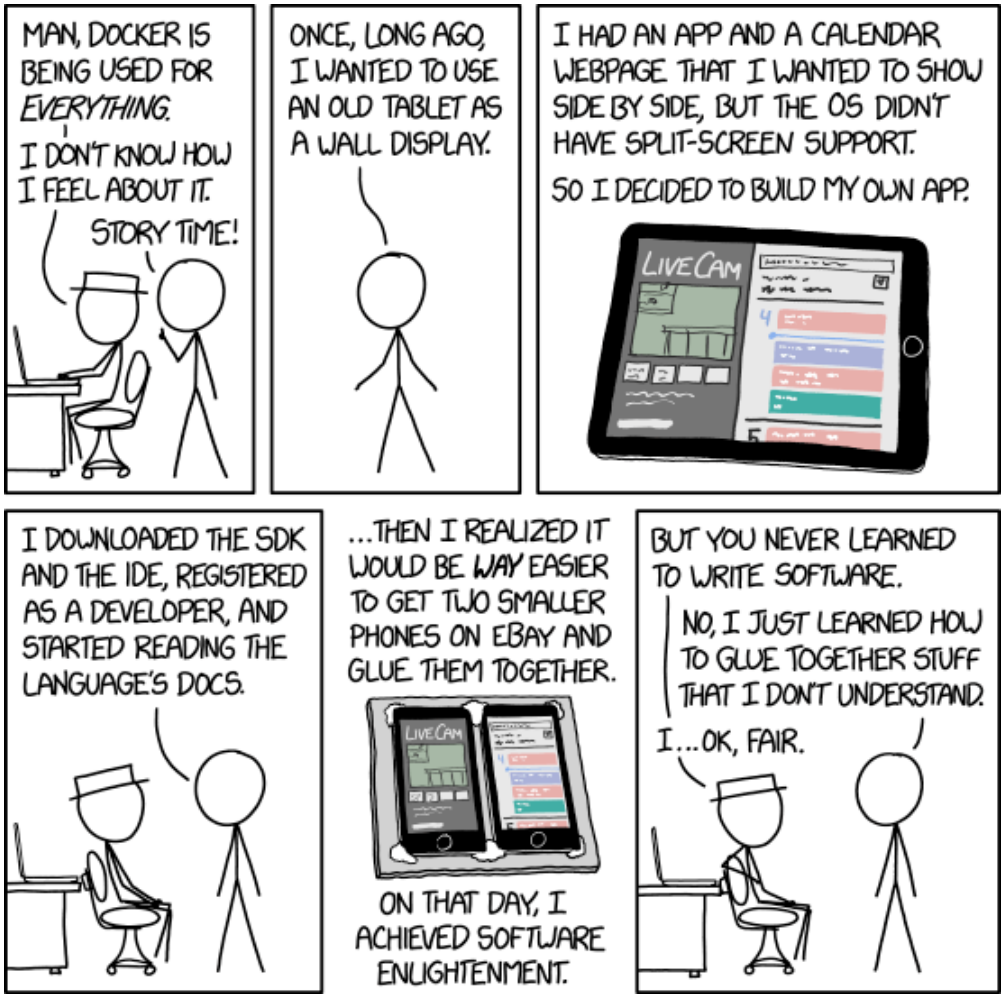
Software Carpentry



xkcd – there may be a lot of this...

Matt Hilton

Astrophysics & Cosmology Research Unit, UKZN

Software carpentry?

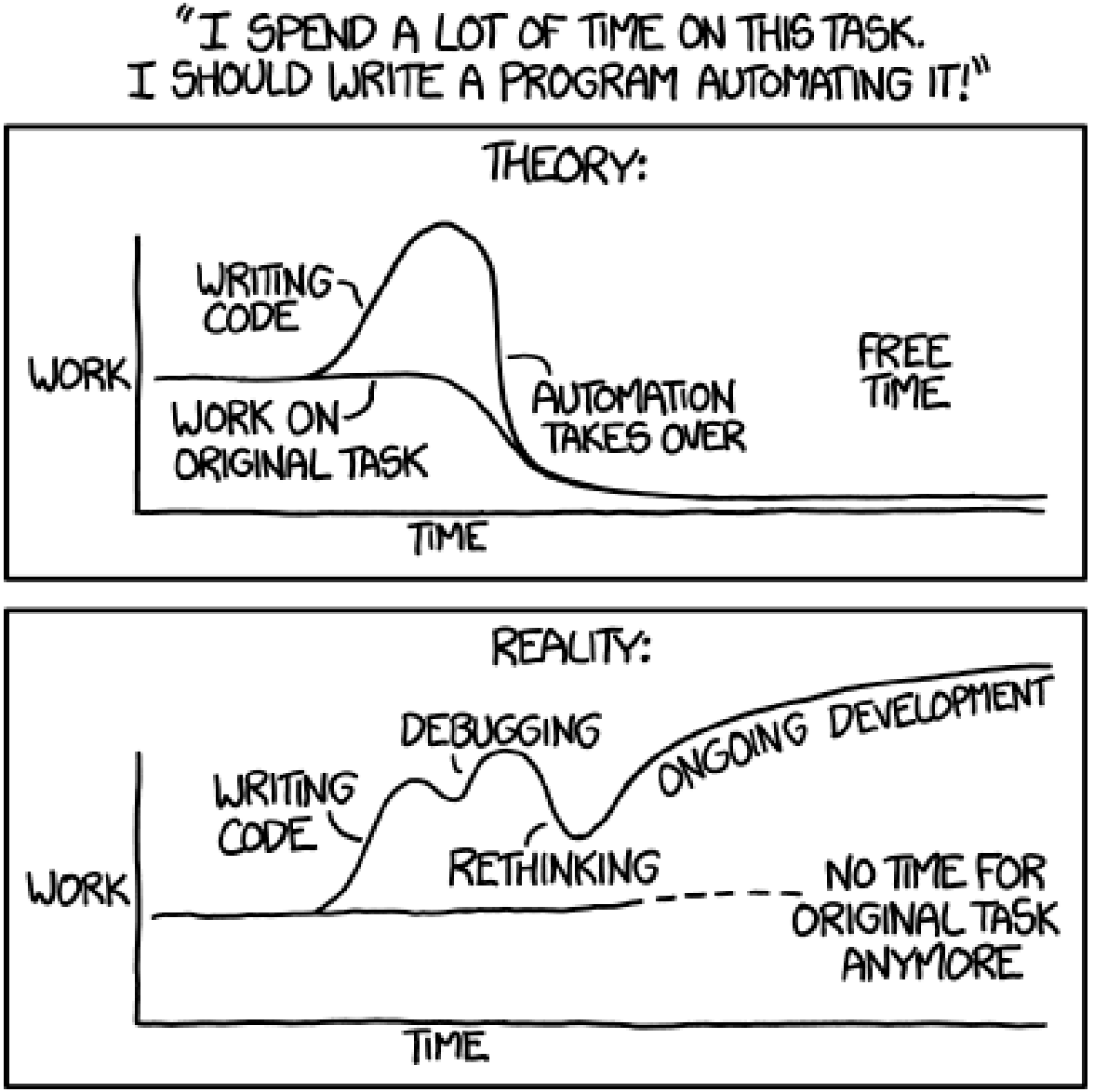
- … as opposed to software engineering???
- An organisation that has been teaching computer skills to scientists since 1998: https://software-carpentry.org/

  (Disclaimer: I'm not from them… nor am I a software engineer…)
- Topics covered on the UK DISCnet course:
  - Automating tasks with the Unix shell
  - Python basics: building programs with Python
  - Writing robust code and unit testing with Python
  - Version control with git
- We have only today… so:
  - This talk – some edited highlights / discussion / tips
  - Later – a tutorial on Git and setting up a Python application to find extrasolar planets in Kepler space telescope data

- What is it?
  - An interpreted, cross-platform high-level programming language
- Why use it?
  - Open Source (https://opensource.org/)
  - Not as fast as compiled languages like C etc., but much faster to write/debug (higher productivity)
  - Nice, friendly syntax (if you like indenting)
  - The standard library is amazing (with great documentation)…
  - … numpy / scipy / matplotlib etc. are amazing too
- What are the alternatives?
  - R? (https://www.r-project.org/)
  - Perl? (https://www.perl.org/)
  - Octave? (https://www.gnu.org/software/octave/)
  - Julia? (looks interesting; https://julialang.org/)

Automation

- More xkcd



Design

- What do we mean by “well designed software”?
- A good design ought to break your code up into the smallest units that are practical (i.e., functions should be short, modular, reusable), and should be organised in a sensible way (e.g., put related functions in the same module; use classes where it makes sense)
- If you are just doing some data analysis task on your own, you perhaps don’t need to worry about design much…
- On the other hand, you DO need to worry about design if you are writing code that is going to be used by others. You will need to think more deeply about the data structures, the tools needed, and especially the interfaces between them.
- You don’t want to have to keep revising interfaces (this is annoying and breaks other people’s code) – if you design it right the first time, you won’t have to.
- If your code has a good design, it should be easy to maintain, and for others to understand

Coding style

- Consistency, and readability, is key – particularly within a package or module
- You can find the style guide for Python and its Standard Library here:

  https://www.python.org/dev/peps/pep-0008/

  (certainly read this if you haven't done much coding before)
- Never use tab to indent in Python – use spaces instead
- Use comments and docstrings (but keep them up-to-date)
- A couple of naming conventions:
  - Module names, function names, variables: `lowercase_with_underscores`
  - Class names: `CapitalizedWords or CamelCase`
- You'll see in the tutorial code that I use `mixedCase` a lot out of habit (I don't like all the underscores)… this is fairly common elsewhere (see, e.g., the Qt Framework)

Debugging

- The nice thing about Python is that it always tells you exactly what went wrong, and where…

  … if this is a problem in someone else's code: remember to give them the entire error message ('it crashed', 'there is a bug' contains no useful information)

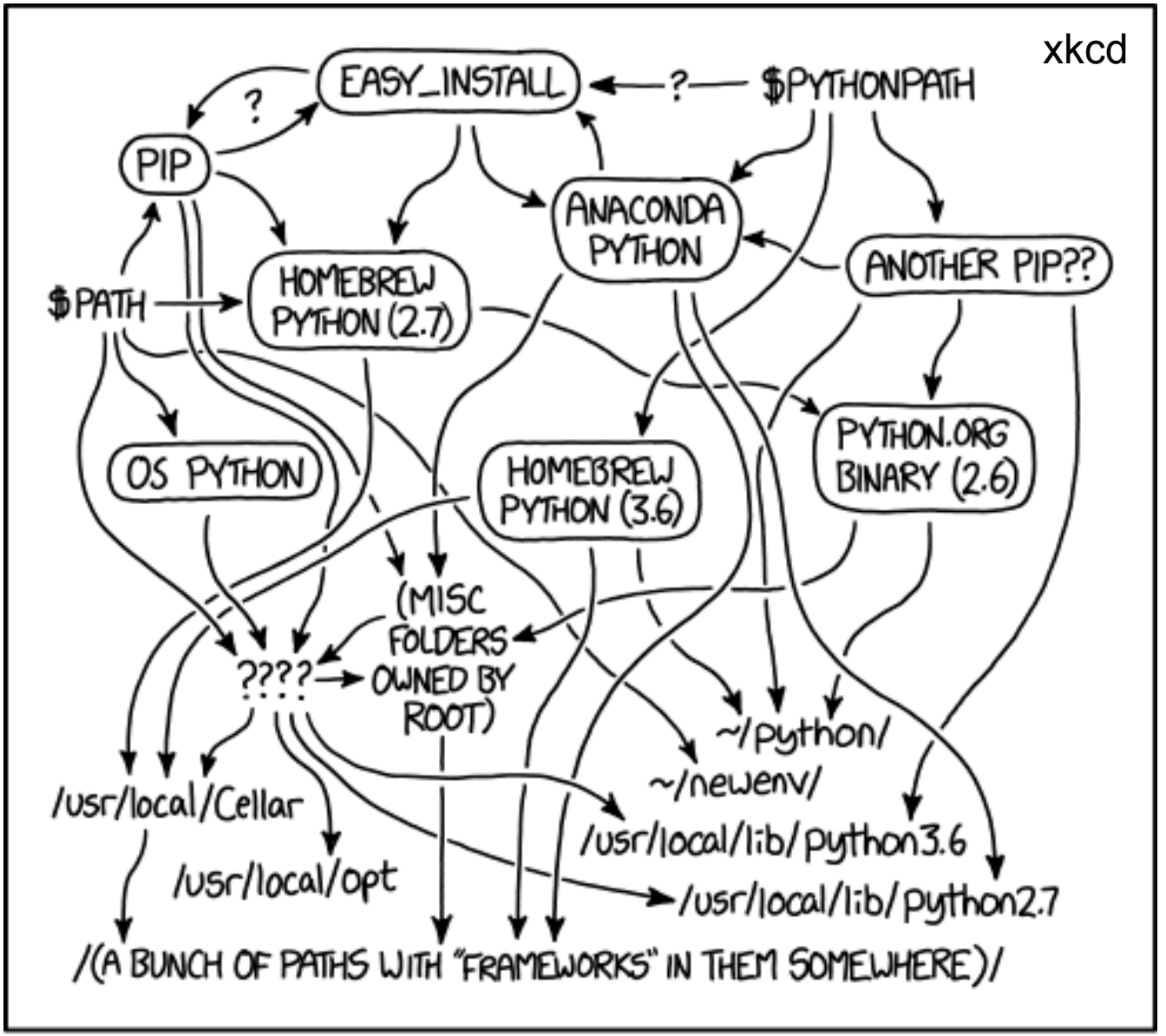
```
Ignoring a divide by zero error ...
Traceback (most recent call last):
  File "debuggingWithIPython.py", line 34, in <module>
    c=divide(a, b)
  File "debuggingWithIPython.py", line 17, in divide
    return a/b
TypeError: unsupported operand type(s) for /: 'float' and 'str'
```

Debugging

```
"""

Simple example of using IPython for debugging

"""

import sys
import IPython

#-----------------------------------------------------------------------
def divide(a, b):
    """Divide two numbers

    Returns a float

    """
    return a/b

#-----------------------------------------------------------------------
# Main

a=5.0
b=2.0

c=divide(a, b)

b=0.0
try:
    c=divide(a, b)
except ZeroDivisionError:
    print("Ignoring a divide by zero error ...")

b="a"
try:
    c=divide(a, b)
except:
    print("This would have triggered a TypeError exception")
    IPython.embed()
    sys.exit()

```

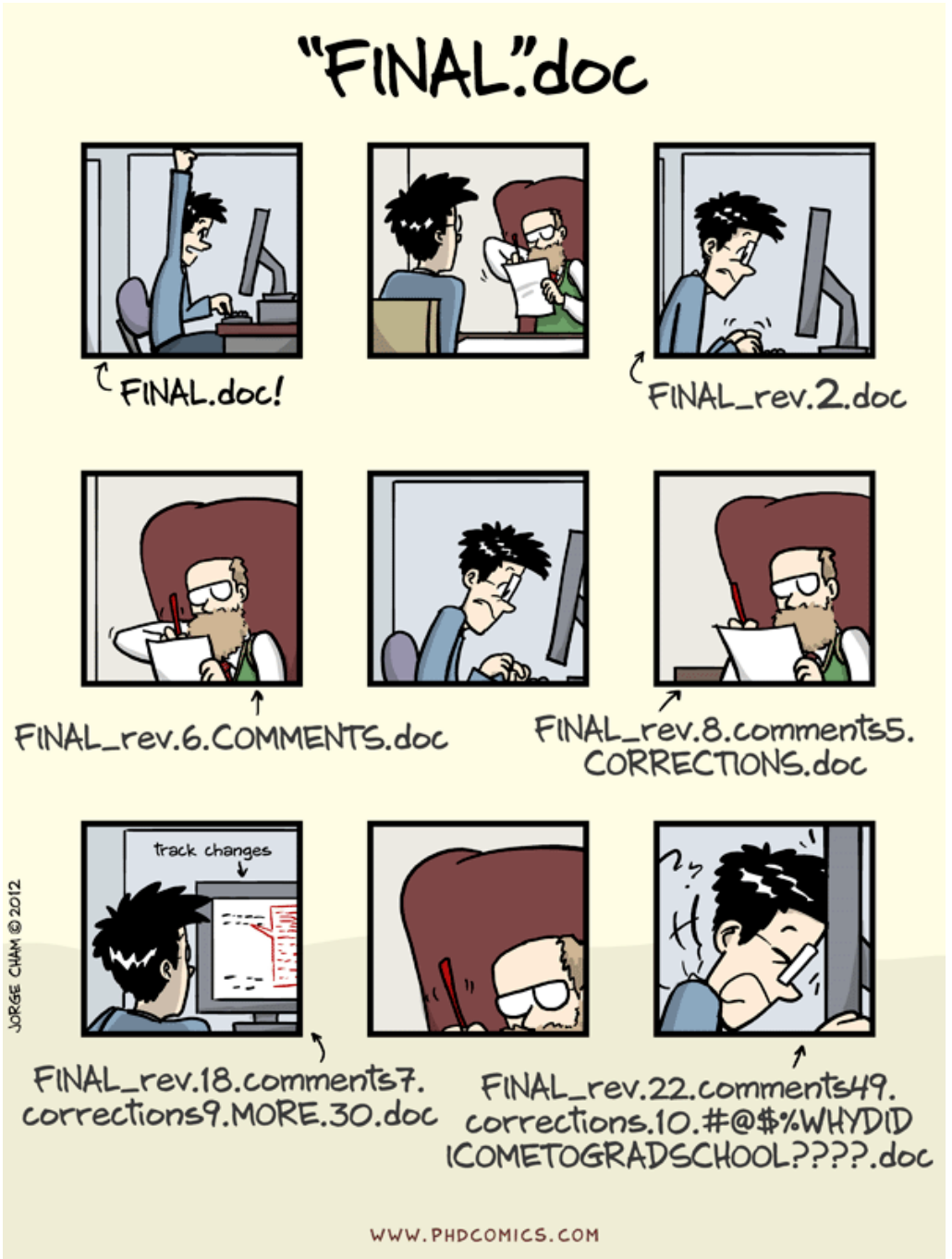
- Because Python is an interpreted language, you can use IPython to “drop in” to a program at any time, inspect the values and types of variables, and figure out why it fell over

Python proliferation



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Version control



- There are many choices…

  … but just use git

- Don't use git to manage a repository with lots of large data files though (keep data and code separate wherever possible)

- Version control helps to keep things running smoothly if you're working in a big collaboration

- Tagging versions / releases of code used in particular papers is a good idea…

Testing, testing, testing...

- You should test your codes frequently
- Some widely-used frameworks exist for Python:
  - The unittest module in the Standard Library:

    https://docs.python.org/3/library/unittest.html
  - The Nose framework:

    https://nose.readthedocs.io/en/latest/
- The general idea is to embed test cases for each individual routine / class etc. into your source code, so that you can automate the testing process on “units” that are as small as practical
- Confession:

  I’ve not used either… I just have a bunch of standard things that I run to check that I’m getting the output I expect for a given package (and I can also use version control to check what changed, if things are different)

Refactoring

- You need to maintain your code… from time to time, you might want to re-write / restructure some of it to make it easier to maintain (refactoring)
- In general: don't break interfaces when you do this – this annoys people who use your code (which may include yourself!)
- Sometimes you may think it would be nice to completely re-write something from scratch…

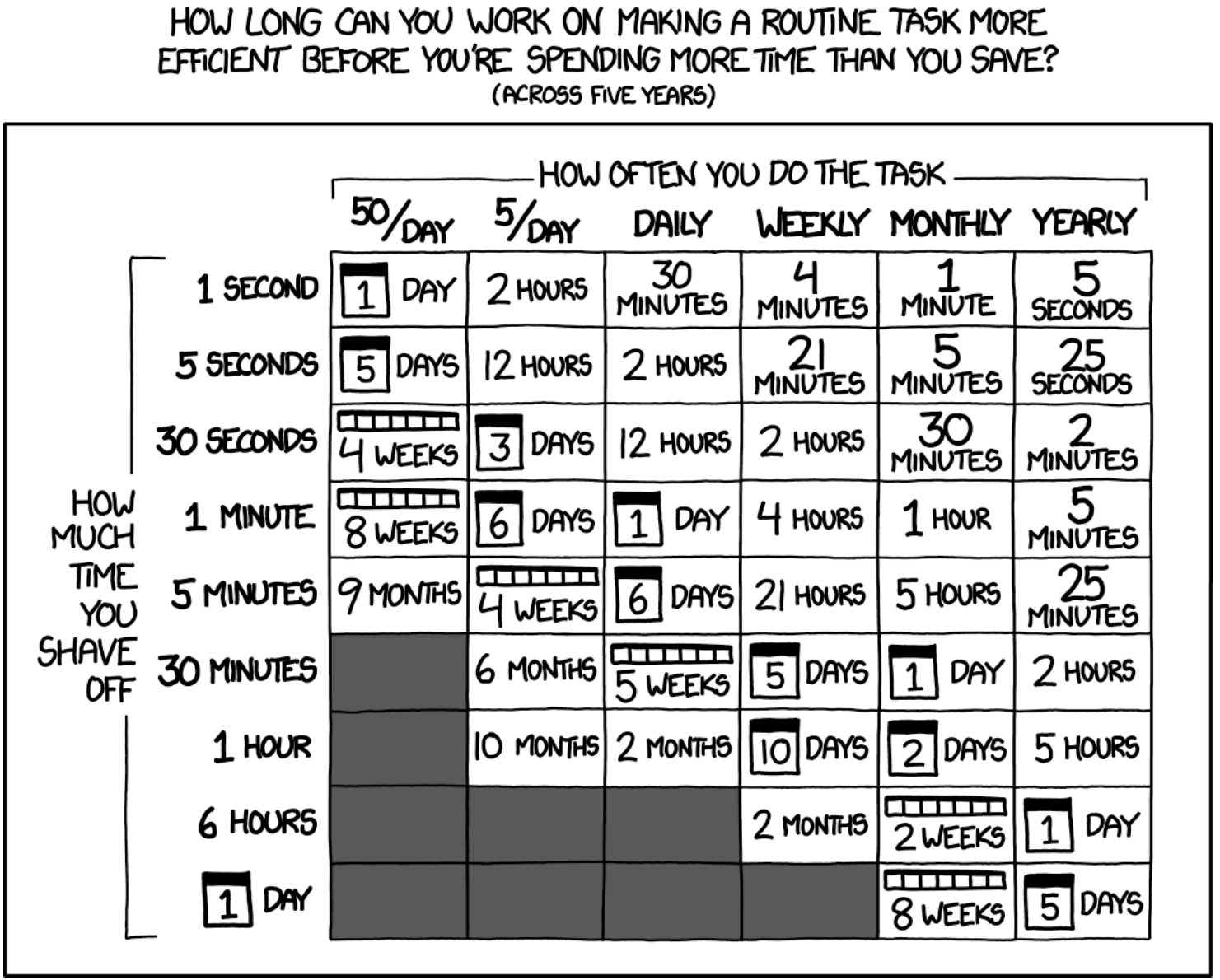
  … but ask yourself: is it worth it?

  … also remember: the computer does not care if your code is beautiful or not, if it works

  "Perfect is the enemy of good enough"

Optimisation

- Yet more xkcd



Optimisation

- Speed tips: always use numpy array operations where possible, and avoid huge `for` loops
- There are some fancy ways to do profiling, but in my experience, using `time.time()` is enough to help you find the slow parts of your code
- Once found, concentrate on those parts (e.g., if a single routine takes 100 sec, and another takes 0.1 sec, which will you focus on? It depends if you're calling the 0.1 sec routine a million times and the 100 sec routine once)
- For calculations that run once and not often (e.g., in part of a pipeline) cache the result somewhere and load it from disk for subsequent re-runs
- If you really need to do something element-by-element on a huge array, here are a couple of possible options
  - Cython (http://cython.org/) - I've used this several times
  - Numba (http://numba.pydata.org/)

```
"""

Example of using time.time() for checking which parts of the code
take the longest to run.

"""

import time
import numpy as np

t0=time.time()
A=np.random.uniform(0, 1, [1000, 2000])
B=np.random.uniform(0, 1, [A.shape[1], 300])
t1=time.time()
print("Making matrices: %.3f sec" % (t1-t0))

C=np.dot(A, B)
t2=time.time()
print("Matrix multiplication: %.3f sec" % (t2-t1))

pInv=np.linalg.pinv(A)
t3=time.time()
print("Pseudo-inverse: %.3f sec" % (t3-t2))

u, s, vh=np.linalg.svd(A)
t4=time.time()
print("SVD: %.3f sec" % (t4-t3))

print("All: %.3f sec" % (t4-t0))

# Output when running this code on my laptop:
#
#   Making matrices: 0.054 sec
#   Matrix multiplication: 0.094 sec
#   Pseudo-inverse: 1.642 sec
#   SVD: 1.931 sec
#   All: 3.721 sec
#
# These are not super-accurate benchmarks, but the idea is to look at the
# order-of-magnitude, if you are deciding which parts of the code could use
# a speed-up (if possible)
#
# For individual routines like this, you could use %timeit in IPython instead
# (which will give acccurate benchmarks)

```

Parallelisation

- Do you need it? Optimise your code first
- There is the multiprocessing module in the Standard Library:

  https://docs.python.org/3.3/library/multiprocessing.html

  … but I find to use it generally requires major surgery on my code
- MPI4Py:

  http://mpi4py.readthedocs.io/en/stable/

  for me is the easiest / quickest way – and even if you're running on a multicore laptop and not a cluster, you can still use it (and MPI is very easy to install on Ubuntu)

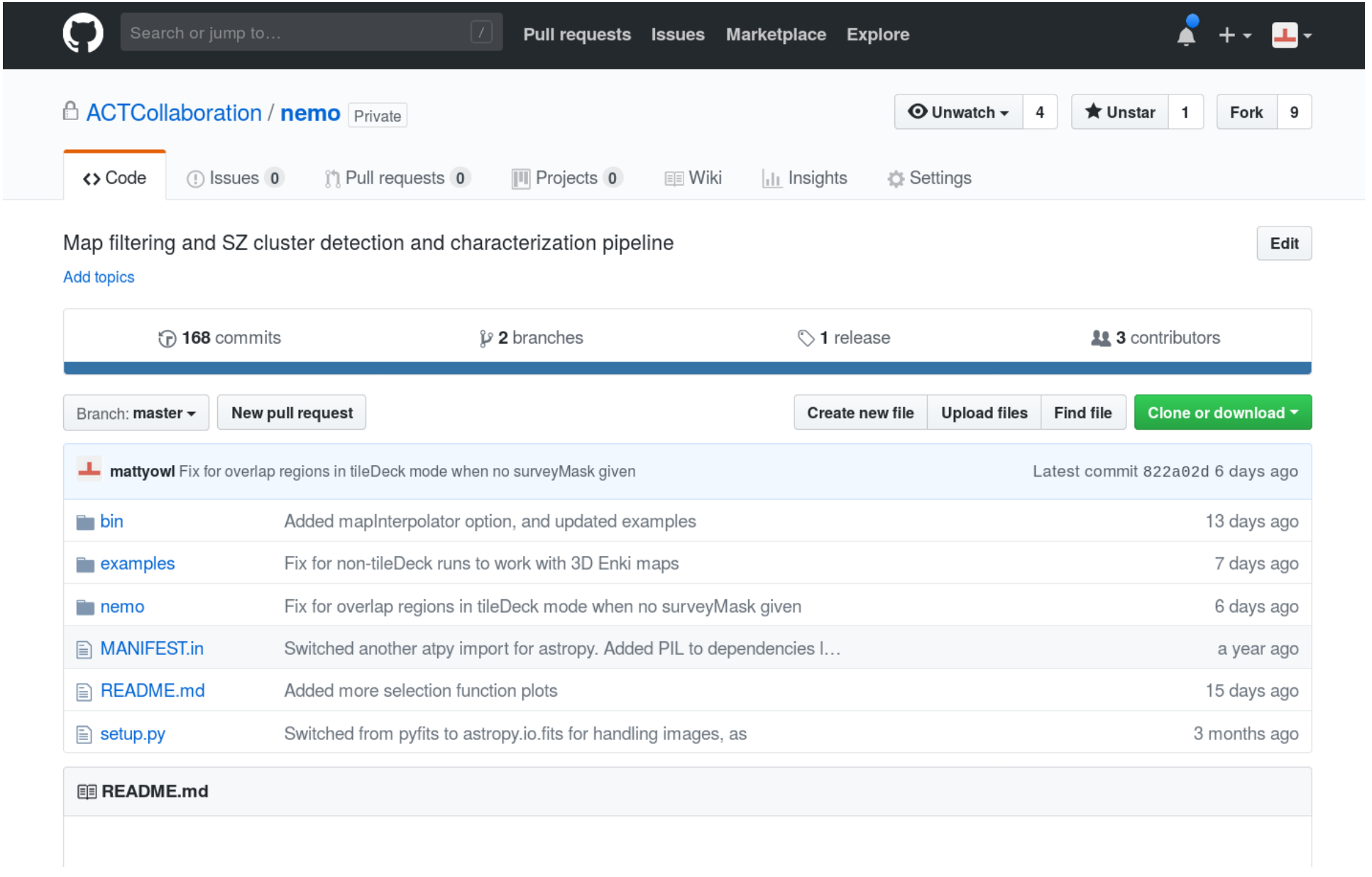
  e.g., mpirun --np 4 your_python_program
- It is also possible to use threads, rather than processes (advantage: less memory usage; disadvantage: more complicated than MPI, depending on what you are trying to do)

Some tools for collaboration

- Git:
  - Distributed version control: allows many users to work on the same code, and has tools to merge / fork / compare code pretty easily; Open Source
- Github (https://github.com/):
  - A web service that hosts Git source-code repositories, with a nice, no-nonsense interface. Free for public, Open Source repositories
  - Alternative services are available (e.g., Sourceforge, Gitlab) that can also host Git repositories
- Slack (https://slack.com/):
  - Seems to have become the leading real-time messaging app for code development
  - Free (if you only keep ~10,000 messages) and cross-platform but proprietary software

Github (or Sourceforge or...)

- The most popular site for hosting code repositories in the world is Github (Microsoft bought it the other week)



Releasing your code

- Where?
- What license?
- Is it easy to use?
- Did you write some nice documentation?

Main Page Download Installation Documentation Bugs and Feature Requests

astLib Python astronomy modules

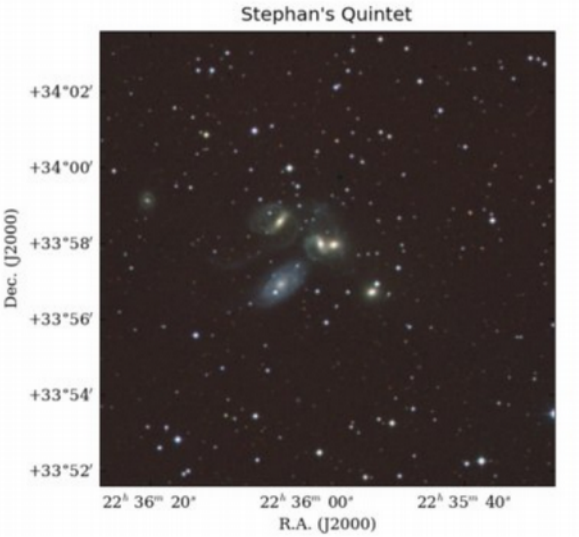
astLib is a set of Python modules that provides some tools for research astronomers. It can be used for astronomical plots, some statistics, common calculations, coordinate conversions, and manipulating FITS images with World Coordinate System (WCS) information through PyWCSTools - a simple wrapping of WCSTools by Jessica Mink. PyWCSTools is distributed (and developed) as part of astLib.

Version 0.10.0 released

Mon, 03/05/2018 - 07:49 — Matt Hilton

Version 0.10.0 of astLib has been released, and is available from both SourceForge and PyPI.

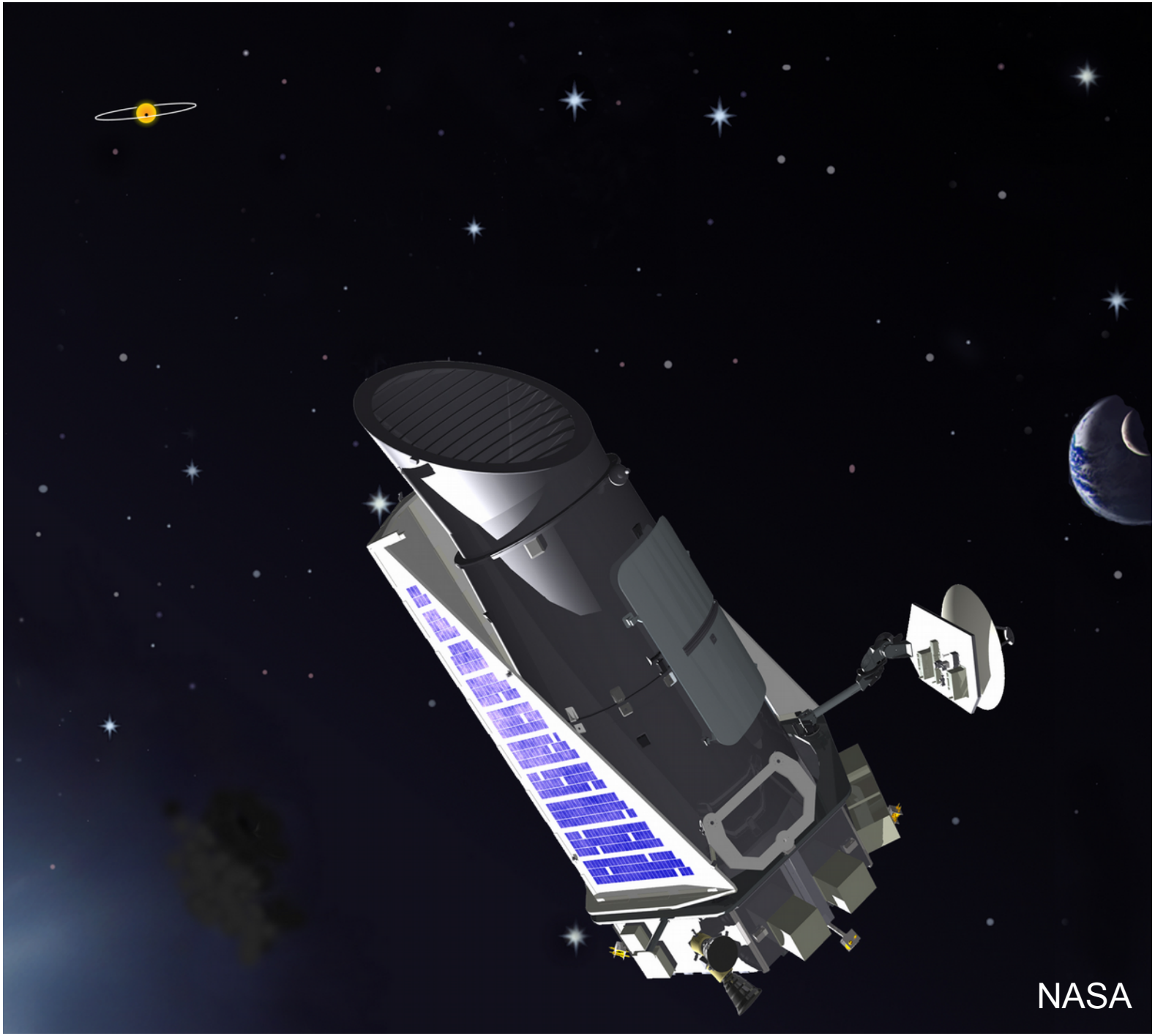
This release breaks compatibility with pyfits (which is depreciated by STScI), and now requires astropy. If you still require a version that works wiith pyfits, please continue to use release 0.9.3.



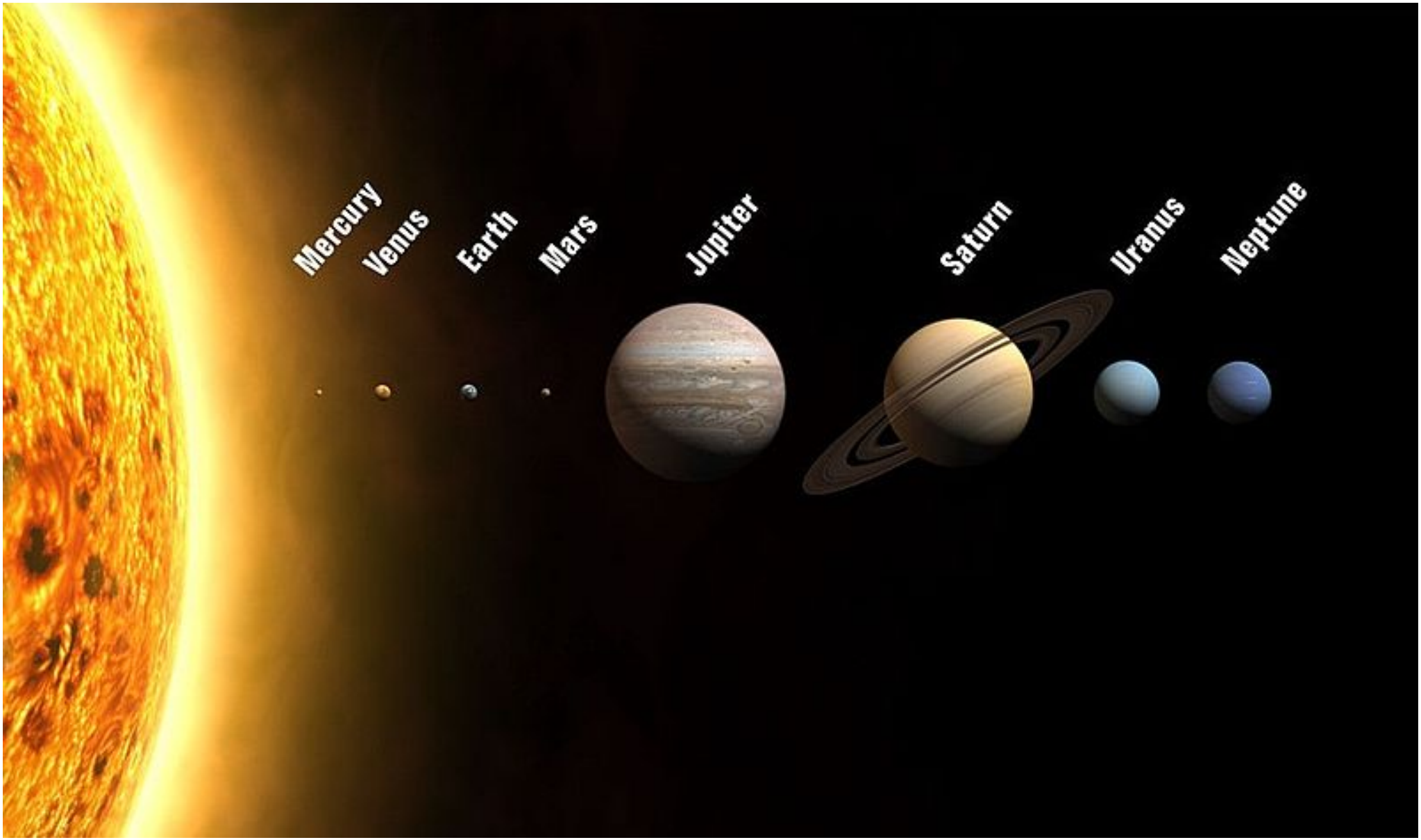
Example output from the astPlots.ImagePlot class in version 0.3.1. Image data taken from the Digitized Sky Survey, using the SkyView interface.

http://astlib.sourceforge.net/ - more than 10 years old…

… and now for some science …



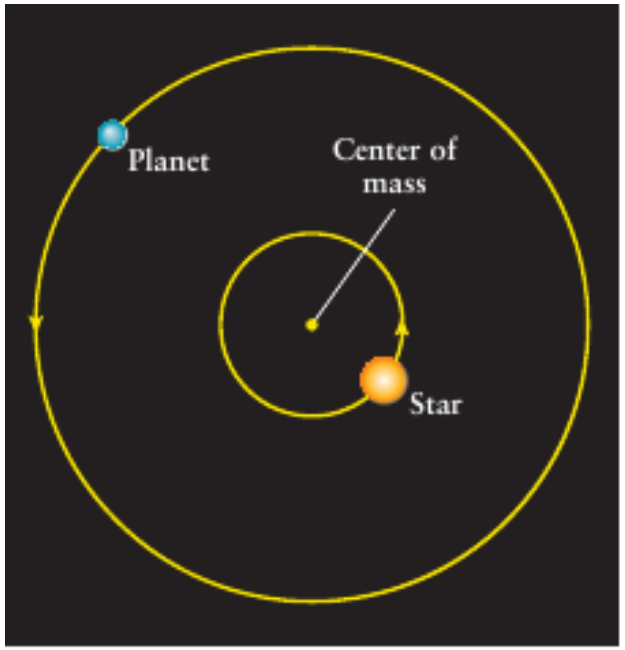
Our Solar System



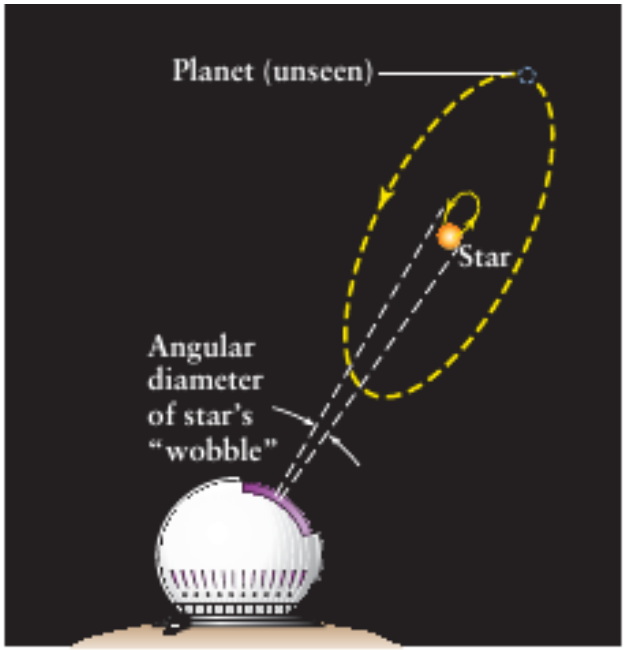
Sizes of planets are to scale – the distances between them are not!

Extrasolar planets

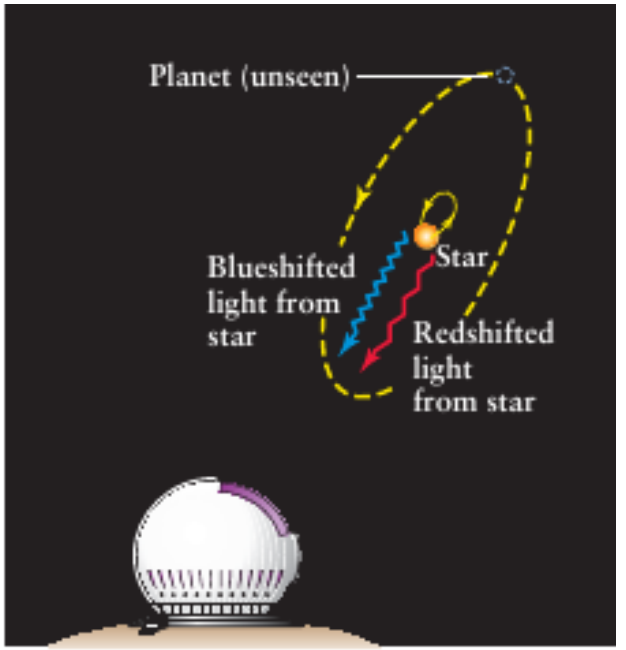
- Since 1995, 3594 planets have been discovered around stars other than the Sun (exoplanet.eu)
- This is very challenging observationally, because stars are ~1 billion times brighter than planets
- One way to find extrasolar planets is to look for the wobble of their parent star – this is the **radial velocity method**



(a) A star and its planet

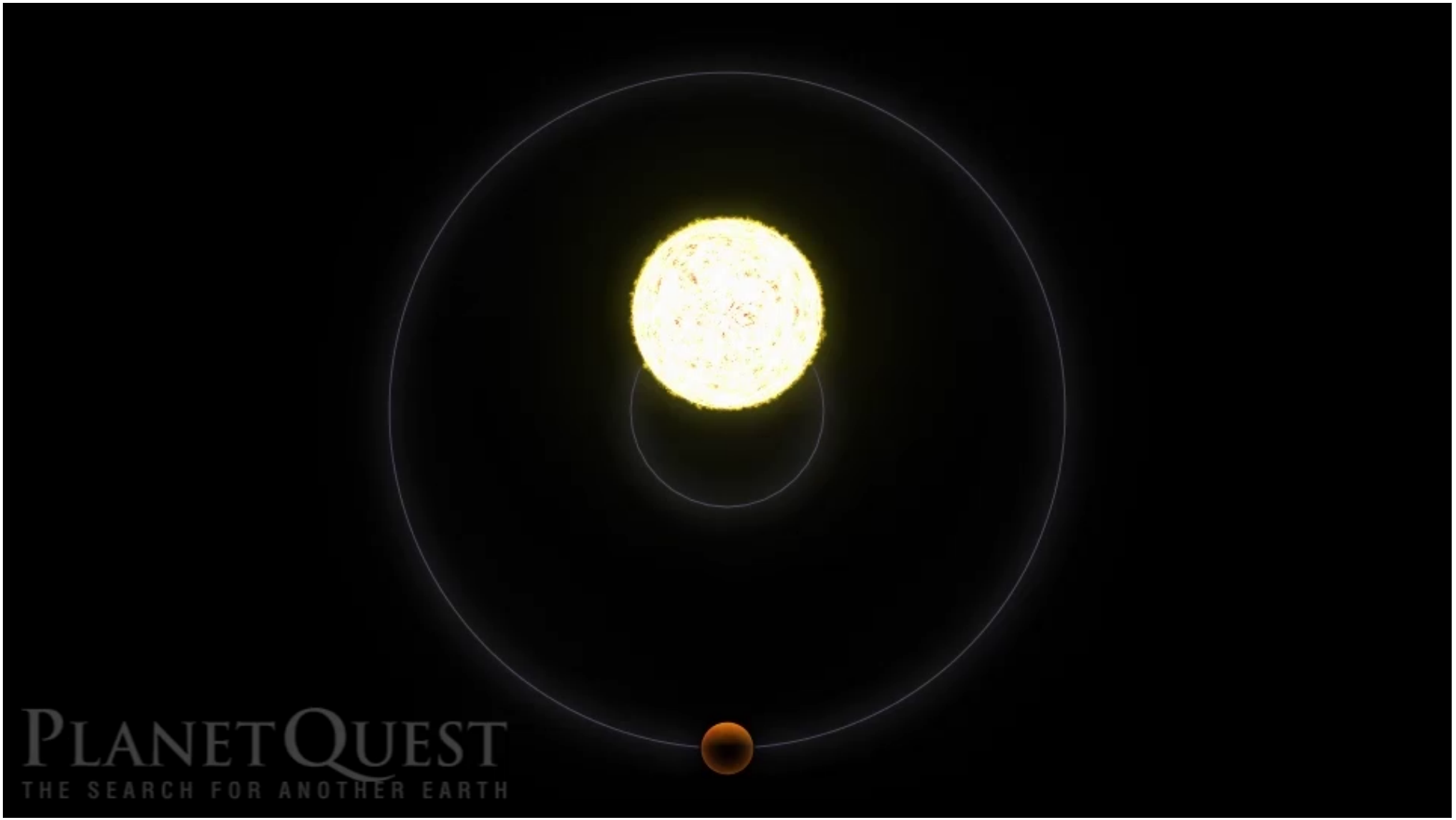


(b) The astrometric method



(c) The radial velocity method

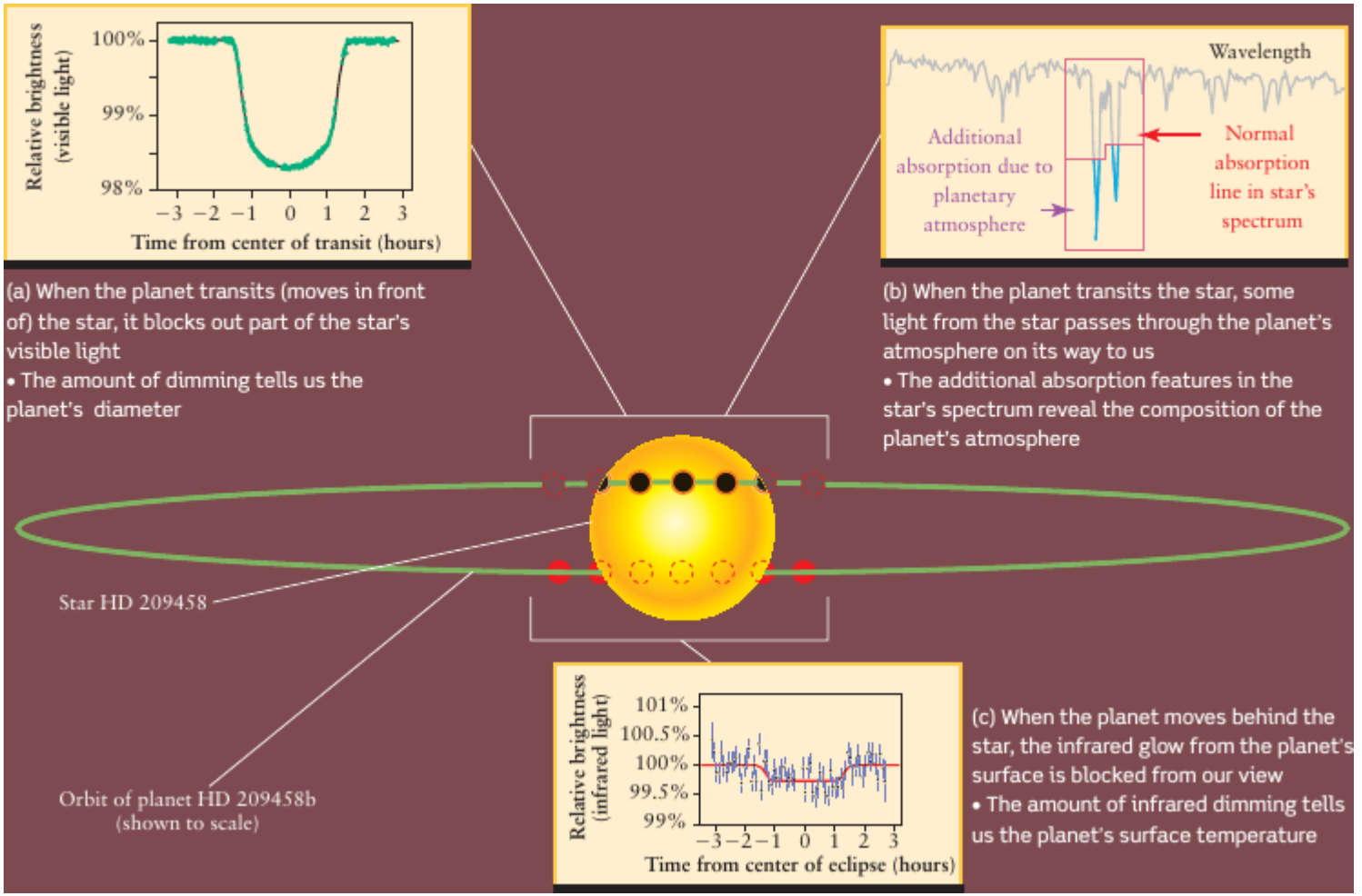
Radial velocity method



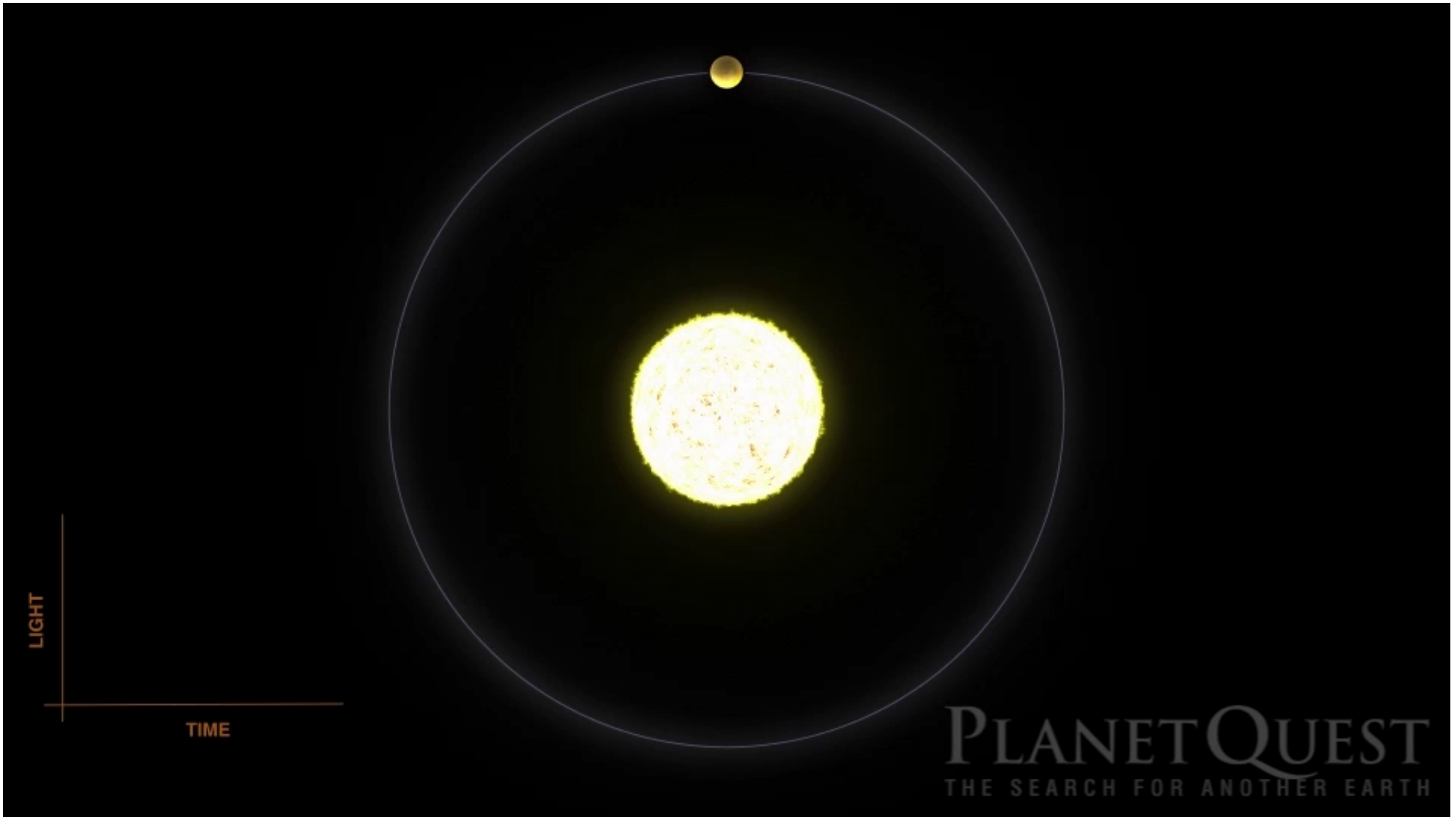
exoplanets.nasa.gov

Transit searches

- Radial velocity searches give an estimate of the mass of an exoplanet, but not its size – for that, **transit searches** can be used
- When an exoplanet transits in front of its parent star, it dims its light for a short period



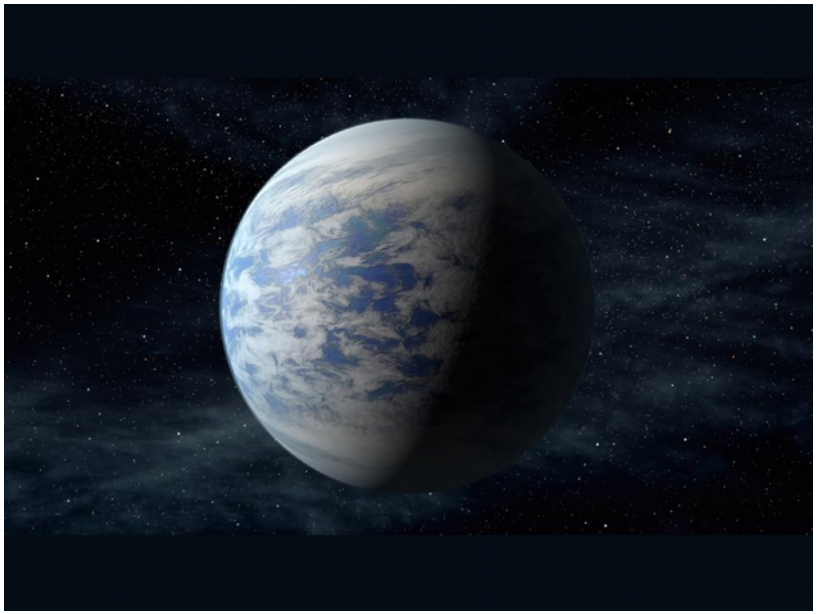
Transit searches



exoplanets.nasa.gov

Properties of exoplanets

- The combination of radial velocity measurements of planets detected using the transit technique gives us both the mass and size of planets, and hence some idea of their average density
- One surprising finding is that many extrasolar planets have masses similar to Jupiter and are orbiting very close to their parent stars
- These objects are called **"hot Jupiters"** – they presumably must have migrated inwards from the edges of their solar systems
- One goal of exoplanet searches is to find planets similar to the Earth to look for signs of **extraterrestrial life**...



Kepler/NASA/JPL

The tutorial...

- We'll go through the process of setting up a Python application to play with Kepler light curve data – this will take you through:
  - Basic Git usage
  - Setting up a pure-Python package (with modules and an executable script)
  - Adding a nice command-line interface
  - Loading and processing Kepler light curve data
- It's open ended, because I leave it to you to figure out how to fit the data…
- You can find it here:

  https://github.com/mattyowl/AIMS-DISCnet-2018