Contents

Astro 330 Course Book

As we make our way through the course, the Lectures and Lab Assignments will be added to this book as chapters (in their respective parts). You will be able to git-pull to download the relevant notebooks, as well as view the assignments locally. They will also be online.

> **Note**
>
> We recommend you spawn up your own notebooks for assignments, pushing them to the relevant github directory via github classroom. If you work directly in the pulled assignent document, it may be overwritten or have merge conflicts when you next try to pull. You will be able to copy the downloaded files, so you can make a copy of the Lab if you wish to work directly on it.

1. Astronomical Imaging & Functional Programming

Programming Topics: Functions, optional arguments, args and kwargs, documentation

Science Topics: filters/bands, multiband data, combinations, and continuum subtraction

Specialized Tools: `Astropy` tools for working with image data (`SkyCoord`, `units`, `WCS`, and `Cutout2D`)

Functional Programming

Everyone here is familiar with the idea of functions in Python. We import them (e.g., `np.array()`) and use them. We can also define our own functions. The standard function in python looks like

```
def func(args):
    '''
    Description of A function
    '''
    # Do something in the function
    return #something
```

Why write functional code? What concrete benefits arise from writing functions in our code?

My Answers

- Prevents repetition of code that is run often
- Allows for manual or automatic unit testing (ensuring each definable task in the code is working as expected)
- Allows for double documentation. Functions themselves encourage docstrings, which helps users (you included) understand your function's purpose. It also means your "main" code is composed fewer lines, due to calling functions instead. This makes the primary code easier to read, especially if the functions are well-named.
- Functions are modular: you can import functions from one python file to another, the basic structure of package creation. (also works for classes)
- Functions encourage simple outputs
- Functions create regions of local scope, which helps prevent mucking up your global namespace

How to Write a Function

Besides the formatting, how do we actually construct good functions? How do we do so efficiently?

- **Playtest** code in the **global namespace** (i.e., your script or notebook). When it's working, **move into a function**.
- Functions should be **< 50 lines**, and **ideally < 20**. They should handle **simple, repetitive operations** that don't have many steps
- A function **wrapping** several smaller functions is better than one long function
- Anytime you find yourself **copying and pasting code**, you could probably use a function
- **Don't** write **detailed docstrings** until the function is **"mature"**. #Comments throughout are sufficient.
- Because you won't have to type them out often, variable names in functions can be **highly descriptive.**

- It's better to let the **code** guide **when to make** functions, rather than forcing all code into functions *a priori*.

Documentation

Documentation is one of the **most important things** we create as programmers. It is the **number 1** descriminator in whether a codebase remains useful over any extended period of time.

I consider all of the following documentation

- variable names
- function and class names
- #inline comments
- docstrings for classes, methods, and functions

Docstrings are the most formal form of documentation. These are descriptions of scripts, methods, classes, and functions which are typically written in one of several standard methods.

The docstring styling is important, because there are automated programs such as Sphinx and mkdocs which can analyze your code base and construct documentation websites for you.

We'll be using the numpy/scipy in this class. It is a standard styling which is compatible with the tools listed above and is easy to write and read. It looks like this:

```
def func(arg1,arg2,arg3=0,**kwargs):
    '''
    A function which does something with three optional arguments and some keyword
arguments

    Parameters
    ----------
    arg1: float
        description of arg1, which has been indicated to be a float
    arg2: array_like
        description of arg2, which has been indicated to be an array
    arg3: int, optional
        description of arg3, an optional input. Default: 0
    **kwargs: dict_like, optional
        Any additional keyword arguments will be stored in the functions kwargs
dictionary.

    Returns
    -------
    output1, output2: int, float
        This function returns 2 outputs, one an int and one a float, apparently.

    Notes
    -----
    Requires numpy version >1.15.5
    Notes go here

    See Also
    --------
    Any links to other functions can go here
    '''
    pass
```

I definitely don't write that level of documentation for most of my functions (at least not as I'm writing them). My docs, when I'm playtesting code, usually look more like

```
def func(a,b,c):
    # Does Calculation X using a,b,c, which should be floats
    pass
```

Then, once the function has been fleshed out, and I know for (mostly) sure that the inputs and outputs I want are stable, I'll write up the more official docstring from above. In writing code for myself or colleagues (i.e., not distributed), notes and see also is generally overkill. But the basic docs laying out parameters and returns is worth doing. Here's a website built using automatic docstrings.

*Args and **kwargs

The simplest version of a Python function includes several **ordered** inputs, defined in the `def` line of the function definition. One step higher in complexity, after the ordered arguments, one can add **optional** or **keyword** arguments, which have set defaults but can be overwritten by the user by entering an argument with the appropriate keyword, via `kw=something`.

These methods are likely familiar. Less familiar may be the use of `*args` and `**kwargs` as inputs to your functions.

In the simplest sense, these arguments are **packing** and **unpacking** variables. As a reminder about unpacking and packing, let's look at an example:

```
theta = (1,5,3,6,7,3)
def func(theta):
    a,b,c,d,e,f = theta
```

In the code above, we "packed" theta into an iterable containing subparts, and read this container into `func`. We then unpacked it by setting 6 variables all equal to `theta`. This works only if the number of variables equals the length of the input.

Tuples are the default for packing things, but any simple iterable (like a list) work. Here's another example:

```
def func():
    a = 2
    b = 5
    return a,b
```

By returning `a,b`, we are actually telling python to pack the outputs into a tuple. If we then run out function by setting a single variable, via `out = func()`, then `out` will be a tuple containing two items. But we can also run our function via `c,d = func()`, and it will unpack that tuple directly into the two variables specified.

As a random side note, one of our favorite functions, `np.where()`, returns a `tuple` containing the indices of interest and then, usually, nothing. Hence, you'll often see people run where as

```
w, = np.where(somthing>something_else)
```

That comma tells python to unpack the tuple, and the lack of a second variable tells it to simply dump whatever the second output is.

Packing and unpacking are key to our use of args and kwargs in functions. But first, why use them at all?

- You may want to allow any number of inputs to your function
- You may want to pass arguments from your function to another.

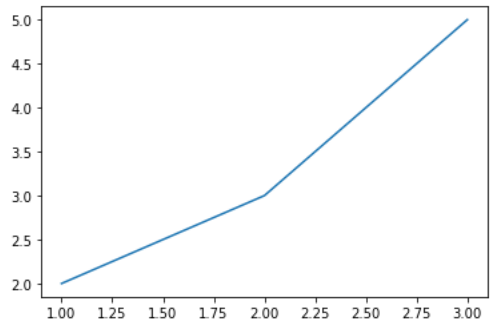
Matplotlib is a classic example of a library with many `**kwarg` enabled functions. When we use wrappers that conveniently allow a complex plot to be made in one line, we still want to be able to put in parameters that alter or set more granular settings.

By adding `**kwargs` as an optional input to our functions, the user can supply any number of extra keyword/arg pairs. These sit harmlessly in a dictionary called `kwargs`. If then inside our function we call another function which has `**kwargs` enabled, we can then feed our `kwargs` dict into that function, and unpack it into keyword argument pairs again with the `**`.

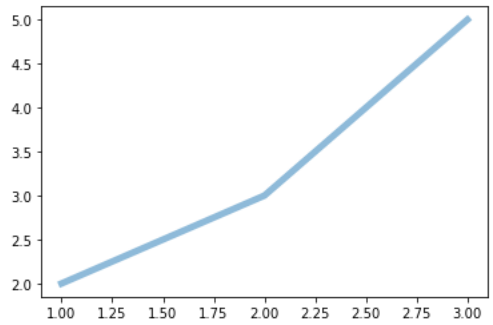
Example

```
import matplotlib.pyplot as plt
def plot_wrapper(x,y,**kwargs):
    fig, ax = plt.subplots()
    ax.plot(x,y,color='C0',**kwargs)
```

```
x = [1,2,3]
y = [2,3,5]
plot_wrapper(x,y)
```




```
x = [1,2,3]
y = [2,3,5]
plot_wrapper(x,y,lw=5,alpha=0.5)
```



Notice that when I run `plot_wrapper`, I'm passing in arguments accepted by `ax.plot`, and since I feed `**kwargs` into `ax.plot` inside my function, it passes through to the plot command.

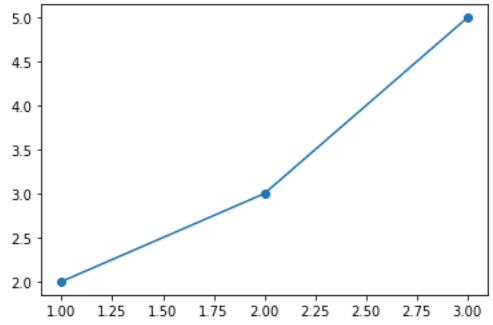
Checking kwargs

In the above example, we blindly fed all kwargs of `plot_wrapper` into `ax.plot`. So If I were to add a kwarg to `plot_wrapper` that is **not** a kwarg of `ax.plot()`, I'd get an error.

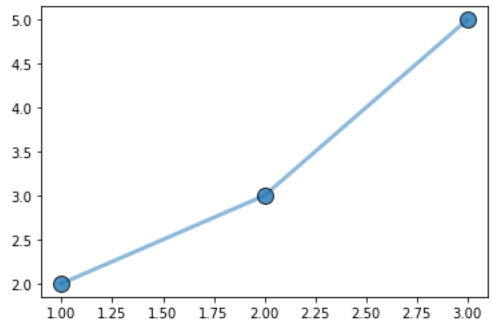
We can be more careful by either demanding a dictionary input for `matplotlib` commands specifically, or we can check the keys of `kwargs` manually and ensure that we only pass the relevant ones along. Since we can't be sure which of the *many* inputs a user would use, in this case, it makes most sense to demand a `dict`. We still use the magic of unpacking to take that dictionary and pass it into the plotting command as key-value pairs, though.

```
def plot_wrapper(x,y,line_args={},symb_args={}):
    fig, ax = plt.subplots()
    ax.plot(x,y,color='C0',**line_args)
    ax.plot(x,y,'o',color='C0',**symb_args)
```

```
plot_wrapper(x,y)
```




```
plot_wrapper(x,y,line_args={'lw':3,'alpha':0.5},
                 symb_args={'ms':12,'mec':'k','alpha':0.8})
```



Notice that normally, one can't feed a dictionary of key-values into a function. `ax.plot()` expects the keywords to look like `(...,lw=3,alpha=0.5)`. But because we used `**` on our dict, we *unpacked* it into the function.

So, to summarize: `**` packs up a key=value pairs when used in a function definition, storing in a dict called `kwargs`, and then using `**kwargs` *again* in the *call* (rather than definition) of another function will unpack that dictionary and pass the inputs to the function as desired.

Working with Images

Now that we're caught up on our function definitions, documentation, and fancy use, let's dive into the science part: images!

What's in an image

Astronomical images differ from standard images taken with a phone or DSLR camera primarily in that they both use older CCD technology and are typically "black and white", recording only intensity of light and not color.

Knowledge about color instead comes from the placement of **filters** in front of the detectors which have a certain **bandpass**, allowing through light within a certain wavelength range and blocking light outside it.

Thus, we only ever get one color at a time, unless the telescope splits incoming light by color using, for example, a **dichroic**, and then sends the two streams to two detectors with different filters simultaneously.

This means that in some sense, single-band (i.e., one narrow-ish wavelength range) images are simple programmatically: They are 2D arrays in which each pixel (position in array) contains a single number: counts (in some units).

Usually, these units are counts in ADU (analog-to-digital units), which are roughly related to the actual counts of photons hitting the detector, via properties such as the gain of the system.

Generally, we are not concerned with the exact units/gain/etc., because we back out the flux in photons (or energy, rather) via empirical calibrations with objects in the sky of known flux.

Narrowband Data and Continuum Subtraction

Many astronomical surveys are carried out with **broadband** filters, such as *U,B,V,R,I* or *Sloan-u,g,r,i,z*. These wide filters let in a lot of light, making for quicker observations.

Narrowband filters, meanwhile, are generally used to target **emission lines** from gas. This includes the famous $H\alpha$ line, along with other ionization lines.

Narrowband filters attempt to block all light except that just surrounding the emission line. But the emission seen (at some wavelength targeting a line) is the superposition of the emission line and the stellar continuum (i.e., the star light also coming from the galaxy/region).

Thus, to use narrowband data properly, it must be **continuum subtracted**. Generally, this is done by scaling a broadband image also encompassing the line (so for λ6563 $H\alpha$. R-band) and then subtracting it from the narrowband image.

Breakout Question: Discuss with your partner an obvious problem with recovering emission line fluxes using this method.

Why bring this up?

In Lab 2, we'll be working with a set of *HST* images in different bands, including $H\alpha$, and you'll have to perform a continuum subtraction.

When working directly with images, the primary mode of operation is the measurement of flux in different regions on the sky – in both broadband, and narrowband data.

http://www.astrofoto.ca/serge/DSO/M33_Ha-RC.htm

Image Handling with Astropy

Lastly, we need to know how to work with astronomical images in Python. As "simple" 2D arrays, simply opening and plotting an image is easy. But keeping track of, say, the celestial coordinates associated with a pointing on the sky, is not.

Luckily, the open-source `astropy` library has many tools that facilitate the streamlined handling of astronomical imaging.

Opening the Image

Opening images using astropy is made easy using the `fits` file handler. We'll use a context manager to read in an image:

```
from astropy.io import fits

with fits.open('antenna_Rband.fits') as hdu:
    header = hdu[0].header
    image = hdu[0].data
```

We can confirm we've loaded a 2D array:

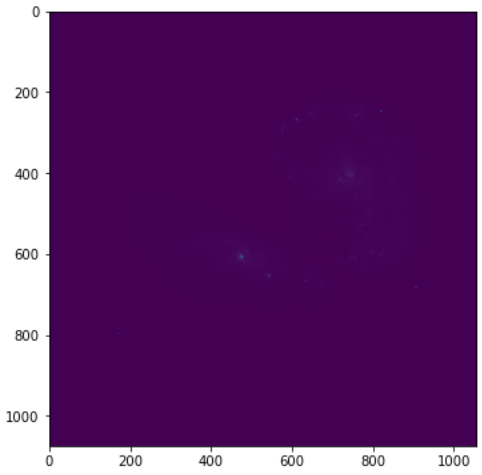
```
image
```

```
array([[ 4.8691406,  5.432617 ,  5.713867 , ..., 11.48584  , 11.48584  ,
        11.48584  ],
       [ 5.573242 ,  5.713867 ,  5.291504 , ..., 11.48584  , 11.48584  ,
        11.48584  ],
       [ 6.4179688,  5.573242 ,  5.573242 , ..., 11.48584  , 11.48584  ,
        11.48584  ],
       ...,
       [11.48584  , 11.48584  , 11.48584  , ...,  6.2768555,  5.432617 ,
         5.9956055],
       [11.48584  , 11.48584  , 11.48584  , ...,  5.291504 ,  5.432617 ,
         5.573242 ],
       [11.48584  , 11.48584  , 11.48584  , ...,  5.291504 ,  4.8691406,
         3.4614258]], dtype=float32)
```

We can make a basic plot showing it:

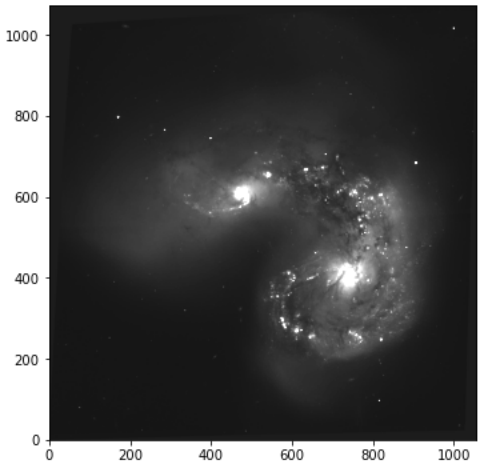
```
fig, ax = plt.subplots(figsize=(6,6))
ax.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x7fcfd9379bb0>
```



That may not have looked like much, but it did at least confirm we have a 2D image. Let's tweak some settings.

```
import numpy as np
fig, ax = plt.subplots(figsize=(6,6))
m = np.mean(image)
s = np.std(image)
ax.imshow(image,origin='lower',vmin=m-s,vmax=m+5*s,cmap='gray');
```



Next, we need to plot with our x and y axes representing sky coordinate (RA/DEC) rather than pixel position. For this, we'll need to use `astropy`'s `World Coordinate System` module. There are several defined coordinate systems on the sky, but for this class, we'll primarily be using RA and DEC. (i.e., equitorial coordinates measured in hour angles and degrees).

```
from astropy.wcs import WCS

wcs_im = WCS(header)
```

```
INFO:
                Inconsistent SIP distortion information is present in the FITS header
and the WCS object:
                SIP coefficients were detected, but CTYPE is missing a "-SIP" suffix.
                astropy.wcs is using the SIP distortion coefficients,
                therefore the coordinates calculated here might be incorrect.

                If you do not want to apply the SIP distortion coefficients,
                please remove the SIP coefficients from the FITS header or the
                WCS object.  As an example, if the image is already distortion-
corrected
                (e.g., drizzled) then distortion components should not apply and the
SIP
                coefficients should be removed.

                While the SIP distortion coefficients are being applied here, if that
was indeed the intent,
                for consistency please append "-SIP" to the CTYPE in the FITS header
or the WCS object.

                 [astropy.wcs.wcs]
```

```
WARNING: FITSFixedWarning: 'datfix' made the change 'Set MJD-OBS to 53207.000000 from
DATE-OBS'. [astropy.wcs.wcs]
```

The wcs for a given image is constructed from the image header, where some information about the pixel scale and image pointing are stored. Notice here we got a warning. This happens sometimes with *HST* images that are mosaics of pointings that have already been drizzled (an image alignment/combination process). We need to remove the mentioned coefficients.

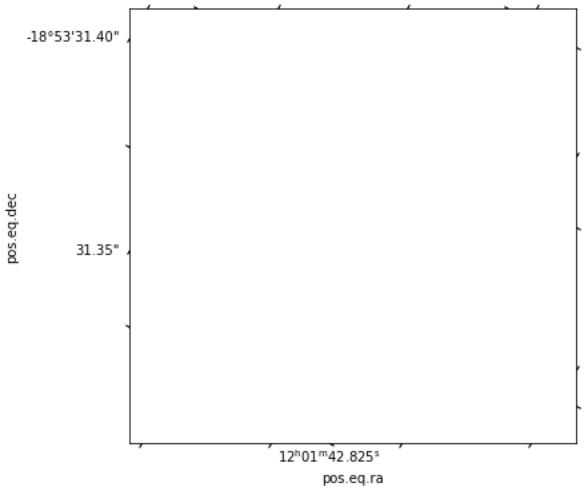
The following will be provided for lab:

```
def strip_SIP(header):
    A_prefixes = [i for i in header.keys() if i.startswith('A_')]
    B_prefixes = [i for i in header.keys() if i.startswith('B_')]
    for a,b in zip(A_prefixes,B_prefixes):
        del header[a]
        del header[b]
    return header
```

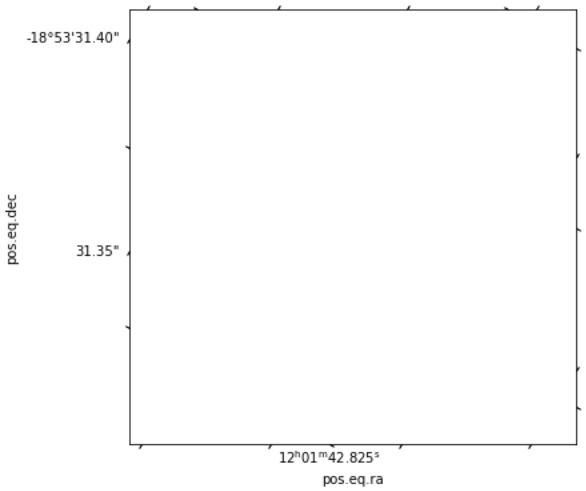
```
header_fixed = strip_SIP(header)
wcs = WCS(header_fixed)
```

We are now ready to plot our image in celestial coordinates. Our game plan is to use `matplotlib`'s `projection`. You may be familiar with this as the way to make an axis, e.g., in polar coordinates. `Astropy WCS` objects can be treated as projections in `matplotlib`. Below I show two ways to initialize an axis with our new wcs projection.

```
fig, ax = plt.subplots(figsize=(6,6),subplot_kw={'projection':wcs})
```



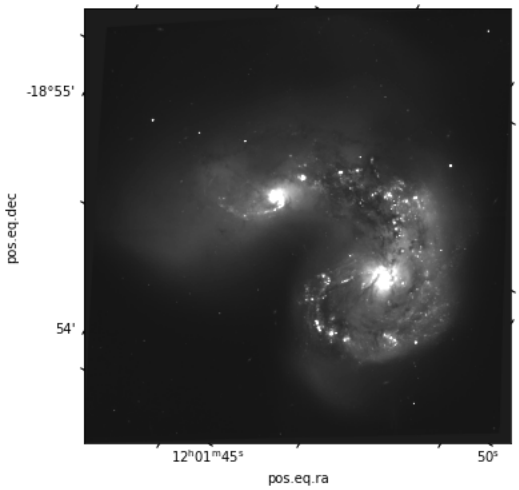

```
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(projection=wcs)
```



The two methods work equally well. The former is slightly preferred, as it makes it easy to make multiple panels with a wcs projection.

Now, all we have to do, is plop our image into it:

```
fig, ax = plt.subplots(figsize=(6,6),subplot_kw={'projection':wcs})
ax.imshow(image,origin='lower',vmin=m-s,vmax=m+5*s,cmap='gray');
```




```
ax
```

```
<WCSAxesSubplot:xlabel='pos.eq.ra', ylabel='pos.eq.dec'>
```

Lab Preview

In the lab, you'll be working with the same image of the Antenna galaxy, in several bands. In addition to the basic reading in and use of header/wcs, you'll be learning about `SkyCoord`, the `astropy` handler for coordinates, and `units` which is a module we'll be discussing throughout class. You'll use `Cutout2D` to make zoom in plots of the image while retaining coordinate information. And you'll use the `photutils` package and `sep` to find sources and perform some aperture photometry.

Lab 1: Overview, Review, and Environments

Objectives

In this lab, we'll

- Review the computational infrastructure around our data science environments,

- Go through the process of ensuring that we have a Python environment set up for this class with the proper installed packages
- Within our environment, we'll review the basic data science operations in Python, and introduce some tips and tricks.

> **Take a Deep Breath**
>
> Don't freak out if stuff presented here is brand new to you! Ask a friend, google around (esp. stack overflow) and find a solution. All of these examples can be done in a few lines of code.

Part I: Computational Ecosystem

In the space below, or in your own assignment, answer the following:

Question A

Describe the following terms, and point out the differences between them. Feel free to look things up.

- python:
- the terminal:
- the file system:
- jupyter:
- an IDE:
- a text editor:
- git:
- PATH:

I am writing this lab in a notebook. We'll be discussing the pros and cons of notebooks in class. Below, I'm going to check which python installation on my computer is being pointed to within my PATH. As a reminder, you can see your full path by echoing it from the terminal. Within a notebook, that looks like this:

```
!echo $PATH
```

```
/Users/ipasha/anaconda3/bin:/Users/ipasha/anaconda3/condabin:/anaconda3/bin:/usr/loca
l/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Users/ipasha/anaconda3/bin:.
```

> **Note**
>
> The "!" in my notebook allows me to run terminal commands from a notebook; you don't need this symbol when running commands in an actual terminal.

I can check my python as follows:

```
!which python
```

```
/Users/ipasha/anaconda3/bin/python
```

We can see that calls to `python` are triggering the python installed in `anaconda3`, which is what we want (see the installation video for more details). If your call to `which python` in the terminal returns something like `usr/bin/python`, then something has likely gone wrong with your installation. There are some troubleshooting steps suggested in the installation video.

Setting up an Environment.

We can think of packages as programs installed on our computer. But what if one of my projects needs Photoshop 14.0, and another needs a feature that was only available in Photoshop 12.5.2? An environment is the system on which your code is being executed. When you fire up a terminal, this is usually your *base* environment, the default one for your *user*

on a given computer system. But rather than always installing programs in this base installation, we can create custom environments for each of our projects. We can then install the exact dependencies for those projects within our environments, and we'll know they won't mess with each other.

For this class, we're going to be doing a lot of package installations. To ensure we are all on the same page and are working with the same tools, we're going to use a `conda environment` to maintain versioning. Note: there are multiple environment creation tools/methods. Conda environments are the predominant standard in astronomy, hence their use here.

In your terminal, type the following:

> **Note**
>
> If you are on WINDOWS, you NEED to use the ANACONDA TERMINAL. NOT YOUR WINDOWS POWER SHELL/COMMAND PROMPT. Search your pc for anaconda and you'll see an anaconda launcher (if you followed the installation video correctly). From there, you should be able to find an anaconda terminal/prompt. That's where you should do anything whenever I say "from your terminal".

```
conda create -n a330 python=3.8
```

Once you run this, answer "y" to the prompts, and your new environment will be installed.

> **Note**
>
> The above command may take several minutes to execute.

Next, we want to activate this environment (still in our terminal). We do this as follows:

```
conda activate a330
```

When you do so, you should see the left hand edge of your prompt switch from (base) to (a330).

Next, let's make an alias so that getting into our a330 environment is a snap. We're going to access a file called `.bash_profile`, which allows us to set aliases and environment variables. This file is located in your home directory, so I can print mine here:

```
!more ~/.bash_profile
```

```
# >>> conda init >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$(CONDA_REPORT_ERRORS=false '/anaconda3/bin/conda' shell.bash hook 2>
/dev/null)"
if [ $? -eq 0 ]; then
    \eval "$__conda_setup"
else
    if [ -f "/anaconda3/etc/profile.d/conda.sh" ]; then
# . "/anaconda3/etc/profile.d/conda.sh"  # commented out by conda initialize
        CONDA_CHANGEPS1=false conda activate base
    else
        \export PATH="/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda init <<<
export BASH_SILENCE_DEPRECATION_WARNING=1
export PATH=$PATH:/Users/ipasha/anaconda3/bin
export PYTHONPATH=/Users/ipasha/anaconda3/bin/python3.8
PATH=$PATH:.
alias python='python3'


[K[?1l>/ipasha/.bash_profile[m[K
```

Notice above I use the `~` which is a shorthand for home directory. On my computer, the default home directory for my user is `/Users/ipasha/`.

This file has some conda stuff in it at the top, as well as some path and python path exports, as well as an alias.

Yours should also have the conda init stuff, if you installed anaconda properly. Using your text editor of choice, add a line to this file that reads `alias a330='conda activate a330'`.

Now, from your terminal, source your profile by typing `source ~/.bash_profile`. You're good to go! Test that you can activate your environment by typing `a330` and hitting enter.

> **Note**
>
> To deactivate, just type `conda deactivate`.

Using Vi/vim

Vi/vim is a built-in terminal program that allows for the editing of files. It is helpful to learn, especially when working on remote servers. We'll go into it more later, but here is a step by step for performing the above step with vim.

- First: from the terminal, type `vim ~/.bash_profile` and hit enter. This will open the editor. If 'vim' isn't recognized, try 'vi'.
- Next: Press the "I" key to open insert mode. Move your cursor with the arrow keys to the desired line, then type in the alias command shown to left.
- Finally: Press `esc` to get out of insert mode, then type `:wq` and hit enter in order to "write" then "quit".

Adding Jupyter

You'll be using notebooks during this class, and we need to make sure that we can access our new environment from within Jupyter notebook. To ensure this, we're going to do the following:

First, make sure your environment is activated.

Then, type:

```
conda install -c anaconda ipykernel
```

This ensures we can select different kernels inside jupyter. A kernel is basically "the thing that is python", the root thing being run on your system when you use python. By creating environments, we're creating different unique kernels, and we can now get to them within our notebooks.

Now, run the following:

```
python -m ipykernel install --user --name=a330
```

Once you've done this, you should have the ability to access your new environment from within Jupyter. We can test this as follows:

- First, open a new terminal window, and activate your environment (if you made the alias, this means typing `a330` in your terminal.
- Next, type `jupyter lab` to open jupyter lab. If for some reason you don't have jupyter lab yet, you can install it now with `conda install -c conda-forge jupyterlab`.
- Once you have lab open, there should be a 'launcher' page, with one option being to create a new notebook using python – you *should* see your environment listed there.
- If you don't hit refresh on the webpage just in case.
- You can also click on the option to open a python3 notebook. Inside, in the top right corner, it should say your current environment (probably Python 3). Clicking that, it should give you the option to choose a different environment, and your environment should be listed there.

> **Note**
>
> If you already had a lab open, you'll have to hit refresh to get it to show up.

Installing Packages

Now that we have our environment, we're going to install the set of packages we need for this class. We may need more of them as the semester goes on, but for now, do the following (in your terminal, within your environment).

```
conda install -n a330 numpy scipy astropy matplotlib
```

(again, hitting "y" when prompted). Again, this step might take a minute or so to run.

Congrats, you now have an environment set up for this class, and can jump in and out of it at will, either in your terminal, or within a Jupyter notebook.

> **Hot Tip**
>
> It's highly recommended you do these steps anytime you start a new research project. Up front, you may not know all the dependencies that will arise, but as you go along, if you keep your work to that environment, you'll be able to carefully control which versions of which packages you're accessing at all times.

Part II: Python Review

In this section, I'll ask you to perform some pythonic operations to get back into the swing of things if it has been a little while.

For this assignment, please carry out your work in a Jupyter notebook, with the questions labeled and your output shown. You'll submit this notebook via Github, but we will discuss how to perform this step in class.

Question 1

Create a 2D array of dimensions 1000 x 1000, in which the values in each pixel are random-gaussian distributed about a mean of 10, with a sigma of 2, and then use matplotlib to display this image. Make sure (0,0) is in the lower lefthand corner.

```
# Your Code
```

Question 2

The distribution of pixels in your above image should not have many outliers beyond 3-sigma from the mean, but there will be some. Find the location of any 3-sigma outliers in the image, and highlight them by circling their location. Confirm that the fraction of these out of the total number of pixels agrees with the expectation for a normal distribution.

```
# Your Code
```

Question 3

When dealing with astronomical data, it is sometimes advisable to not include outliers in a calculation being performed on a set of data (in this example, an image). We know, of course, that the data we're plotting ARE coming from a gaussian distribution, so there's no reason to exclude, e.g., 3-sigma outliers, but for this example, let's assume we want to.

Create a numpy masked array in which all pixels that are > 3σ from the image mean are masked. Then, calculate the mean and sigma of the new masked array.

```
# Your Code
```

Clipping the outliers of this distribution should not affect the mean in any strong way, but should noticably decrease σ.

Question 4:

Using Array indexing, re-plot the same array from above, but zoom in on the inner 20% of the image, such that the full width is 20% of the total. Note: try not to hard code your indexing. You should be able to flexibly change the percentage. For this one, use a white-to-black color map.

```
# Your Code
```

Your image should now be 200 by 200 pixels across. Note that your new image has its own indexing. A common "gotcha" when working with arrays like this is to index in, but then try to use indices found (e.g., via `where()`) in the larger array on the cropped in version, which can lead to errors.

Question 5

Often, we have an expression to calculate of the form

$$\sum_i \sum_j a_i b_j$$

Your natural impulse for coding this double sum might look like this:

```
total = 0
for i in a:
    for j in b:
         total+= i*j
```

which, mathematically, makes sense! But as it turns out, there's a way we can do this without any loops at all — and when $\vec{a}$ and $\vec{b}$ get long, this becomes hugely important in our code.

The trick we're going to use here is called array broadcasting, which you can read about at the link if you're not already familar. I'm going to give you $\vec{a}$ and $\vec{b}$ below. For this exercise, calculate the double sum indicated above without the use of a for-loop. Check that your code worked by using the slow double-loop method.

Hint

The command `np.newaxis` will be useful here, or for a slightly longer solution, try `np.repeat` and `reshape()`.

```
a = np.array([1,5,10,20])
b = np.array([1,2,4,16])
# Your Code
```

Tip

If you're familiar with the jupyter magic command `%%timeit`, try timing your loop vs non-loop solutions with a longer list (say, 5000 random numbers in $\vec{a}$ and $\vec{b}$). How much faster is the non-loop?

Question 6

Often in astronomy we need to work with grids of values. For example, let's say we have a model that describes some data, and the model has 2 parameters, a and b.

We might choose different combinations of a and b, and determine a metric for how well models of such combinations fit our data (e.g., χ^2).

We may then want to plot this χ^2 value for each point on our grid – that is, at each grid position corresponding to some a_i and b_j.

Below, I provide a function, `chi2`, which returns a single number given some singular inputs `a` and `b`.

Create some arrays of `a` and `b` to test that range between 1 and 25, and have 10 entries evenly spaced between those values. Then, loop over them and find the χ^2 using my function.

Note

We can't get around the double loop in this case, because we are operating under the assumption that the calculation of some single χ^2 using a unique combination of a_i and b_j cannot be vectorized. If it could, we wouldn't need to do this activity. But often, we can't, because the creation of a model given some inputs is nontrivial.

Once you've stored the χ^2 values for each combination of a and b, create a plot with a and b as the axes and show using colored circles the χ^2 value at each location. Add a colorbar to see the values being plotted.

To create this grid, use the `np.meshgrid()` function. For your plot, make sure the marker size is big enough to see the colors well.

```
def chi2(a,b):
    return ((15-a)**2+(12-b)**2)**0.2 #note, this is nonsense, but should return a
different value for each input a,b

# Your Code
```

Question 7

Re-show your final plot above, making the following changes:

- label your colorbar as χ^2 using latex notation, with a fontsize>13
- Make your ticks point inward and be longer
- Make your ticks appear on the top and right hand axes of the plot as well
- If you didn't already, label the x and y axes appropriately and with a font size > 13
- Make sure the numbers along the axes have fontsizes > 13

```
# Your Code
```

Question 8

Some quick list comprehensions! For any unfamilar, **comprehensions** are pythonic statements that allow you to compress a for-loop (generally) into a single line, and usually runs faster than a full loop (but not by a ton).

Take the for-loop below and write it as a list comprehension.

```
visited_cities = ['San Diego', 'Boston', 'New York City','Atlanta']
all_cities = ['San Diego', 'Denver', 'Boston', 'Portland', 'New York City', 'San
Francisco', 'Atlanta']

not_visited = []
for city in all_cities:
    if city not in visited_cities:
        not_visited.append(city)

print(not_visited)
```

```
['Denver', 'Portland', 'San Francisco']
```

```
# Your Code
```

Next, create an array of integers including 1 through 30, inclusive. Using a comprehension, create a numpy array containing the squared value of only the odd numbers in your original array. (*Hint, remember the modulo operator*)

```
# Your Code
```

In the next example, you have a list of first names and a list of last names. Use a list comprehension to create an array that is a list of full names (with a space between first and last names).

```
first_names = ['Bob','Samantha','John','Renee']
last_names = ['Smith','Bee','Oliver','Carpenter']

# Your Code
```

Challenge Problem (worth Extra Credit)

I've created new lists that contain strings of the names in the format Lastname,Firstname, with random leading/trailing spaces and terrible capitalizations. Use a list comprehension to make our nice, "Firstname Lastname" list again.

```
all_names = ['sMitH,BoB   ', '  bee,samanthA',' oLIVER,JOHN ','  caRPENTer,reneE  ']

# Your Code
```

> **Note**
>
> Note that with this last example, we're entering a degree of single-line length and complexity that it almost doesn't make sense to use a comprehension anymore. Just because something CAN be done in one line doesn't mean is has to be, or should be.

You may be wondering what use case this type of coding has in astronomy – turns out, quite a lot. Take this example: you read in a data table and the columns have names like "FLUX HA", "FLUX ERR", etc.

If you're trying to make a `pandas DataFrame` of this table, it is advantageous to rename these columns something like `flux_ha` and `flux_err`. This way, commands like `df.flux_ha` can be used.

Being able to iterate over the string list of column names and turn caps into lower case, spaces into underscores, etc., is a useful skill that will come in handy when wrangling data.

In the solutions, I will show how I myself would do the above example in production code:

By making the string cleaning steps a function, I could take the time to explain what is going on within the function, as well as control for additional possibilities (like the names being in First,Last formatting). I could make this function more robust and complex, and my final comprehension stays readable and simple, as I loop over the names and run each through my handy functions (with some settings tweaked, potentially).

Question 9

Take the arrays `XX`, `YY`, and `ZZ` below and create one multidimensional array in which they are the columns. Print to confirm this worked.

```
XX = np.array([1,2,3,4,5,6,7,8,9])
YY = np.array([5,6,7,8,9,10,11,12,13])
ZZ = np.array([10,11,12,13,14,15,16,17,18])

# Your Code
```

Question 10

Units, units, units. The bane of every scientists' existence... except theorists that set every constant equal to 1.

In the real world, we measure fluxes or magnitudes in astronomical images, infer temperatures and densities from data and simulations, and ultimately have to deal with units one way or another.

Thankfully, our friends at `astropy` know this, and they've come to save the day. This next question serves as an introduction to the `units` module in astropy, which can be both a live saver and a pain in the ass, but at the end of the day is absolutely worth learning.

```
import astropy.units as u
```

The standard import for this library is `u`, so be careful not to name any variables that letter.

To "assign" units to a variable, we multiply by the desired unit as follows. Note that generally the module knows several aliases/common abrreviations for a unit, if it is uniquely identifiable.

```
star_temp = 5000*u.K
star_radius = 0.89 * u.Rsun
star_mass = 0.6 * u.Msun
```

We can perform trivial conversions using the `.to()` method.

```
star_radius.to(u.km)
```

$$619173 \text{ km}$$

Once we attach units to something, it is now a `Quantity` object. Quantity objects are great, above, we saw they have built-in methods to facilitate conversion. They can also be annoying – sometimes another function we've written needs just the raw value or array back out. To get this, we use the `.value` attribute of a quantity object:

```
star_mass.to(u.kg).value
```

```
1.1930459224188305e+30
```

This now strips away all `Quantity` stuff and gives us an array or value to use elsewhere in our code.

Units are great because they help us combine quantities while tracking units and dimensional analysis. A common operation in astronomy is converting a flux to a luminosity given a distance, using

$$F = \frac{L}{4\pi D^2}$$

where L is the luminosity and D is the distance to the source.

What if I've made a flux measurement in astronomical units such as erg/s/cm^2, and I want to know the luminosity in solar luminosities, and my distance happens to be in Mpc? Regardless of my input units, I can easily do this:

```
L = 4 * np.pi * (3.6*u.Mpc)**2 * (7.5e-14 * u.erg/u.s/u.cm**2)
L.to(u.Lsun)
```

$30381.226\ \mathrm{L}_\odot$

This conversion worked because the units worked out. If my units of flux weren't correct, I'd get an error:

```
L = 4 * np.pi * (3.6*u.Mpc)**2 * (7.5e-14 * u.erg/u.s/u.cm**2/u.AA)
L.to(u.Lsun)
```

```
---------------------------------------------------------------------------
UnitConversionError                       Traceback (most recent call last)
<ipython-input-142-1a5aea91b170> in <module>
      1 L = 4 * np.pi * (3.6*u.Mpc)**2 * (7.5e-14 * u.erg/u.s/u.cm**2/u.AA)
----> 2 L.to(u.Lsun)

~/anaconda3/lib/python3.8/site-packages/astropy/units/quantity.py in to(self, unit,
equivalencies)
    687         # and don't want to slow down this method (esp. the scalar case).
    688         unit = Unit(unit)
--> 689         return self._new_view(self._to_value(unit, equivalencies), unit)
    690
    691     def to_value(self, unit=None, equivalencies=[]):

~/anaconda3/lib/python3.8/site-packages/astropy/units/quantity.py in _to_value(self,
unit, equivalencies)
    658         if equivalencies == []:
    659             equivalencies = self._equivalencies
--> 660         return self.unit.to(unit, self.view(np.ndarray),
    661                             equivalencies=equivalencies)
    662

~/anaconda3/lib/python3.8/site-packages/astropy/units/core.py in to(self, other,
value, equivalencies)
    985             return UNITY
    986         else:
--> 987             return self._get_converter(other, equivalencies=equivalencies)
(value)
    988
    989     def in_units(self, other, value=1.0, equivalencies=[]):

~/anaconda3/lib/python3.8/site-packages/astropy/units/core.py in _get_converter(self,
other, equivalencies)
    916                             pass
    917
--> 918             raise exc
    919
    920     def _to(self, other):

~/anaconda3/lib/python3.8/site-packages/astropy/units/core.py in _get_converter(self,
other, equivalencies)
    901         # if that doesn't work, maybe we can do it with equivalencies?
    902         try:
--> 903             return self._apply_equivalencies(
    904                 self, other, self._normalize_equivalencies(equivalencies))
    905         except UnitsError as exc:

~/anaconda3/lib/python3.8/site-packages/astropy/units/core.py in
_apply_equivalencies(self, unit, other, equivalencies)
    884         other_str = get_err_str(other)
    885
--> 886         raise UnitConversionError(
    887             "{} and {} are not convertible".format(
    888                 unit_str, other_str))

UnitConversionError: 'erg Mpc2 / (Angstrom cm2 s)' and 'solLum' (power) are not
convertible
```

Here, `units` realized that I was putting in units of flux density, but wanted a luminosity out, and ultimately those units don't resolve out. Thus, it can be a great way to catch errors in your inputs.

Note: just be careful that sometimes, you throw a constant into an equation but the constant has some units. If you're going to use the unit module to do a calculation, ALL inputs that HAVE units must be assigned them correctly as above for it to work.

For your exercise, consider the following:

The virial temperature of a galaxy halo is given roughly by

$$T_{\rm vir} \simeq 5.6 \times 10^4 \ {\rm K} \left(\frac{\mu}{0.59}\right) \left(\frac{M_{\rm halo}}{10^{10} \ M_\odot}\right)^{2/3} \left(\frac{1+z}{4}\right)$$

where here, we can assume μ is 0.59.

Write a function that takes as an input a halo mass, redshift, and optionally μ (default 0.59), and returns the virial temperature in Kelvin. Your function should take in an astropy quantity with mass units, but should allow for the mass to be input with any appropriate units.

```
# Your Code
```

Lab 2: Astronomical Imaging I

Gather round, for I shall tell a tale old as time. Long, long ago, astronomers gazed at the heavens, and with naught but their eyes, recorded the positions of the stars they saw there. Then, with the telescope, long, heavy contraptions which reached for the skies like an outstretched arm, (thanks, f/20 focal lengths) they gazed through eyepieces in freezing domes, falling, on occasion, to their deaths from cages suspended at prime foci.

Next came glass – emulsion plates – which upon extended exposure revealed upon their ghostly frames the splotches of stars and soon, galaxies and nebulae. Many a grad student, of course, spent their nights twiddling thumbs over micrometer dials to keep these stars in place. Manual guiding... this story teller shudders to imagine it. And yet, even then, with permanent record, no measure could be made that weren't 'tween the eyes of an observer, peering at the glass through magnifying eyepiece, assigning grades of brightness and, amazingly, pretty much nailing the spectral classification of stars by their absorption features.

And after this painstaking work, came the earliest CCDs, fractured by detector failures, riddled with readout noise, consumed by cosmic ray hits, laid low by low quantum efficiencies.

Now... we use Python.

Astronomical images are one of the basic building blocks of astronomical research. While we now obtain data in myriad ways, from gravitational waves to neutrinos, from spectra to polarimetry, the basic tenant of astronomy (and it's most recongizable public impact) is the images we make of the sky.
That is why the first science topic we'll be tackling is imaging. And along the way, we'll learn how `astropy` makes our lives *so* much easier when dealing with images, and how to use robust functions to perform the analyses one might want to carry out on images (after we're done admiring their beauty).

If the glove `FITS`

Many of you are probably familiar with the `FITS` standard. It stands for `Flexible Image Transport System`, and for all intents and purposes, it acts as a container (much like a zip file). Within it, one can store several kinds of information: images (2d arrays of values), as the name implies, headers (dictionary like objects with metadata), and sometimes tabular data as well. A `FITS` file can contain any number of *extensions*, which just refer to "slots" where stuff is stored. Every "slot" has a `header` attribute and a `data` attribute. Thus, you could store a hundred astronomical images in 1 file, each with a different extension and a different header describing the image in that slot.

> **Tip**
>
> Contextual clues are important when dealing with `FITS` files. Files that are explicitly single images almost always have the image stored in the 0th extension, while files that are explicitly table data tend to have the table stored in the 1st extension (with the 0th empty).

The `FITS` standard is pretty old, and may be retired soon, but almost all ground based telescopes still use it for their native image output, so it behooves us to know how to get image data out of `FITS` files.

Problem 1: Loading a FITS file

Write a function which takes as its argument a `string` filepath to a `FITS` file, and should have an optional argument to set the extension (default 0). It should then load the given extension of that `fits` file using a context manager, and return a tuple containing the header and data of that extension.

The function should have documentation that describes the inputs and outputs. Documentation is **incredibly important** when writing code, both in-line and (# comments) and at the top of functions, methods, and classes.

In this class, we'll be using Sphinx-compatible documentation written in the Numpy/Scipy style. There are several reasons to do this.

1. It is a user-friendly and write-friendly, readable documentation format that is easy to add to your functions.
2. It can be rendered by Sphinx, the most popular automatic documentation renderer. If you've ever read the online documentation pages for functions in, e.g., numpy and scipy, those pages were rendered automatically based on the docstrings of the functions in question. This is possible with tools like Sphinx, but the documentation must be formatted correctly for this to work.

In the cell below, I've provided a dummy function which takes in any number of inputs (mininum 3) and chooses a random input to return. The formatting of the documentation is shown there (as well as in the link above).

```
import numpy as np

def random_return(a,b,c,*args):
    '''
    A function which requires three inputs which are floats, accepts any number of
additional inputs (of any kind), and returns one randomly chosen input.

    Parameters
    ----------
    a: int
        description of this integer input
    b: int
        description of this integer input
    c: int
        description of this integer input
    *args: tuple
        any additional arguments get stored here

    Returns
    -------
    choice
        The randomly selected input (type not specified)
    '''
    full_input_list = [a,b,c] + list(args)
    choice = np.random.choice(full_input_list)
    return choice
```

```
random_return(1,5,4,6,4,21,6)
```

```
6
```

When our function has been imported in some code, we can use the `help` command to see the documentation at any time:

```
help(random_return)
```

```
Help on function random_return in module __main__:

random_return(a, b, c, *args)
    A function which requires three inputs which are floats, accepts any number of
additional inputs (of any kind), and returns one randomly chosen input.

    Parameters
    ----------
    a: int
        description of this integer input
    b: int
        description of this integer input
    c: int
        description of this integer input
    *args: tuple
        any additional arguments get stored here

    Returns
    -------
    choice
        The randomly selected input (type not specified)
```

You will also notice my use of `*args` in this function. This allows me to enter additional function arguments. Similar is `**kwargs`, which allows additional arguments tied to an input keyword. The former gets stored in a tuple, while the latter gets stored in a dictionary. We'll be using these a lot in class — you can refresh or learn the basics in Section 2.8 of the chapter on functional programming here.

So as a brief overview of documentation, it contains

1. A brief summary of the function
2. The word Parameters with the next line having underlines of the same length
3. arguments, which are followed by a colon and the data type(s). On the next line, indented, descriptions of the arguments
4. The word Returns, with the same underline scheme
5. The returned objects, labeled the same way as the input.

Also above, we saw how to format when no data type is specified. There, additional inputs could've been *any* data type, so we can't be sure what the output will be. The main thing we didn't cover is optional arguments. Those are set like this:

```
a: int, optional
    Description of the thing. (default 5)
```

So we mark it as optional, and then give the default for it.

With that, you're ready to write and document your code below. **All functions you write in this class should have documentation**.

```
from astropy.io import fits
import os

# your code
def load_fits(...):
    '''

    '''
    pass
```

Problem 2: Data I/O

Problem 2.1

Using the function you created above, read in the header and data of the file `antenna_Rband.fits` which came with the lab assignment.

> **Note**
>
> While this may seem small, the fact that your read-in is now one line instead of ~5 does improve your efficiency! But only if you use your function enough times to overcome the initial time spent writing it... We also have the flexibility to take our function and give it more features over time, which we will do in this class.

Problem 2.2

Next, we need to plot the image data. There are several operations that we almost always perform when plotting astronomical data, as well as several user-preferences for how we "by default" plot images before we begin tweaking things. If you spend any time as an astronomer, you will plot quite literally *thousands* of images — why set all these settings every time, when we can write a handy function to do it for us?

Write a function which takes in as input arguments

- an image (2D array or masked array)

as well as the following optional arguments (so set a default)

- figsize (default (15,13))
- cmap (default 'gray_r')
- scale (default 0.5)
- **kwargs (see here)

Inside the function, create figure and axes objects using `plt.subplots()`. When working in notebooks, it is often useful to set the `figsize` argument of subplots to a nice large-ish value, such as `(15,13)`, which will make the image fill most of the notebook. Since *your* function has set figsize as an argument, you can feed `figsize` directly into the `subplots` call, so that a user of the function can leave the default or set their own.

Next, use `ax.imshow()` to actually plot the image. You'll want to save the output of this, e.g., `im = ax.imshow(...)`. In this plotting call, set `imshow`'s argument `origin='lower'`. We *always* want to do this when dealing with imaging, as we want (0,0) to be a coordinate.

> **Note**
>
> By default, matplotlib uses a "matrix-style" plotting, where 0 of the y axis is in the *top* left corner, and 0 of the x axis is in the *bottom* left corner.

Also within the call to `imshow()`, feed in the cmap from your function (i.e., `cmap=cmap`). The other critical `imshow()` arguments are `vmin` and `vmax`, which sets the saturation points (and thus the contrast) of the image.

We haven't set `vmin` and `vmax` as arguments of our outer function, but because of `kwargs`, we can still create a default here that can be overriden from outside.

As a default, within your function, calculate the mean and standard deviation of the image. Set some temporary variables with the quantities `mu - scale*sigma` and `mu + scale*sigma` (where here `mu` is the calculated mean and `sigma` is the calculated std dev, and `scale` was the optional input). Next, check the kwargs dictionary (which will exist in your function because we added the packing argument `**kwargs` to our function. IF `vmin` and `vmax` are in this dictionary, plug those into your imshow command. Otherwise, use the values determined by the calculation above. Bonus point for accomodating either no vmin/vmax entered, just vmin or vmax, or both (using the calculated values when things aren't provided).

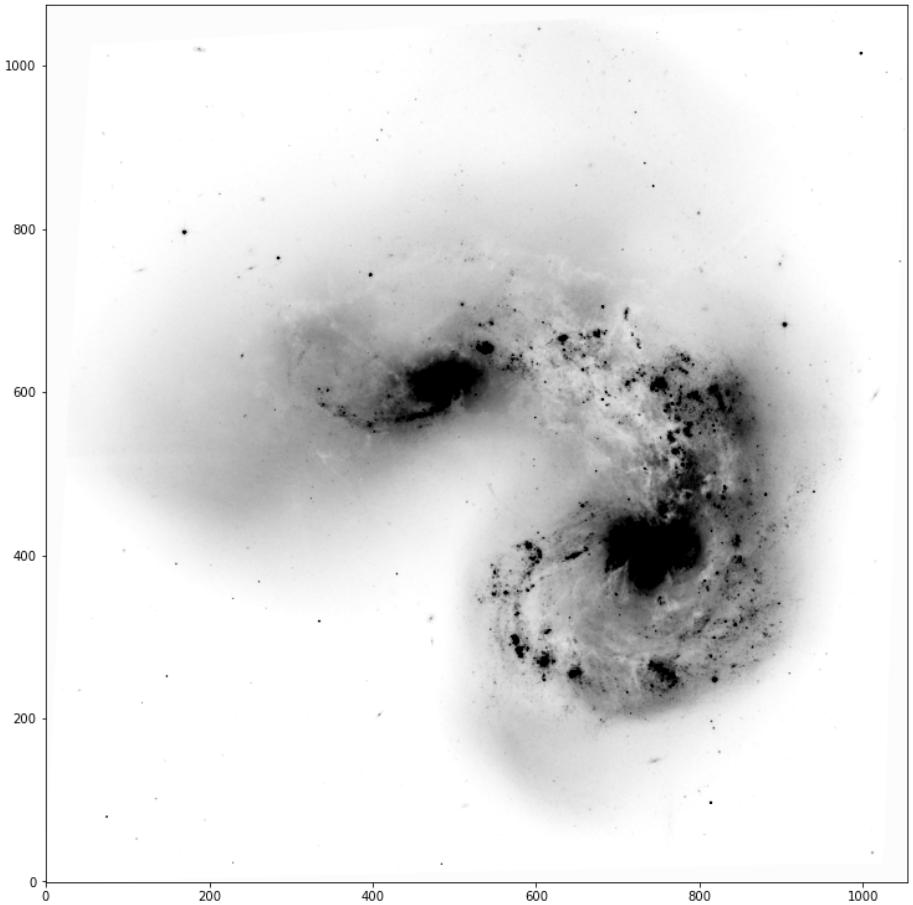
Your function should **return** the created `fig` and `ax` objects so the user can continue to tweak them.

Run your function and test its outputs. Once you're satisfied it's working, use it to plot the provided data. Find either a vmin/vmax pair, or a choice of `scale` which makes the image look pretty!

```
import matplotlib.pyplot as plt
import numpy as np

# Your code
def implot(...):
    '''
    WRITE DOCSTRING HERE
    '''
    pass #replace with your code
```

```
# I've included my output below for your comparison. Overwrite it with your own!
```



Problem 2.3

Copy your function down, we're going to keep messing with it.

So far, we've made it so that with a simple function call and, potentially, with just the input of an image array, we get out a nice plot with a scaling, colormap, and origin selection. In this section, we are going to allow (optionally) for a colorbar to be added to the figure. We're also going to add in the ability for the figure to be plotted in celestial coordinates (i.e., RA and DEC) instead of pixel units, if information about the image (via the world coordinate system, WCS) exists in the image header.

> **Note**
>
> Generally, WCS information is present in the headers of *published* images like the one here, but *not* present in raw data gathered at the telescope. This is because images need to undergo a process known as *plate solving* to determine both the direction (coordinates) toward the image, as well as the pixel scale (i.e., how many arcseconds per pixel in the image).

Add three new optional arguments to your function.

- colorbar = False
- header = None
- wcs = None

Let's start with the colorbar. At the end of your plotting commands, check if `colorbar=True`, and if so, create a colorbar via `plt.colorbar()`, setting the `mappable` argument to whatever you saved the output of `ax.imshow()` into above. Also set the `ax` argument to be your ax; this will tell `matplotlib` to steal a bit of space from that axis to make room for the colorbar.

```
# Your code
```

> **Tip**
>
> When I do this, the colorbar is matching the figure height, rather than the axis height. If that bugs you like it bugs me, check out this solution from StackOverflow, which you can adapt within your function to make the cbar match the axis height.

In order to plot in RA and DEC coordinates, we need to first have an `astropy` `WCS` object associated with the image in question. You can import `WCS` from `astropy.wcs`. WCS objects are created from the headers of plate-solved fits files. In our function, we allow the user to input either a header or a WCS object directly. More on WCS can be found in the lecture notes, or here.

Within your function, check if a wcs is input – if it is, we're good to go and can safely ignore `header` (even if it is provided). If instead only `header` is provided, use the `WCS()` function to create a new wcs object from that header.

> **Tip**
>
> You'll want to do this at the very top of your function.

We now need to move our `fig, ax = ....` creation line into an if-statement. If we are using WCS "stuff", you'll need to set a `projection` for your plot that uses the wcs. This is accomplished as follows:

```
fig, ax = plt.subplots(...,subplot_kw={'projection':wcs})
```

where `wcs` is whatever you've named the output of `WCS(header)` or is the WCS input directly into the function.

```
from astropy.wcs import WCS

# your code
```

Warning

In this case, we will get an error from our function that happens because of some distortion coefficient nonsense between astropy and drizzled HST images. Since it's not pertinent to our lab, I'm providing below a snippet of code you should use to fix your header before running `WCS(header)`.

```
def strip_SIP(header):
    A_prefixes = [i for i in header.keys() if i.startswith('A_')]
    B_prefixes = [i for i in header.keys() if i.startswith('B_')]
    for a,b in zip(A_prefixes,B_prefixes):
        del header[a]
        del header[b]
    return header
```

If this worked correctly, when you add the header you read in from our image, you should now see the axes of your plot change from pixels to `pos.eq.ra` and `pos.eq.dec`. We're now looking at actual on-sky coordinates! Yay!

Problem 2.4

Within the if-blocks of your function that sets the `ax` to be a wcs projection, set the x and y labels to read "Right Ascension [hms]" and "Declination [degrees]" in fontsize 15.

Lastly, to polish things off, use `ax.tick_params()` to set inward, larger ticks, and increase the axis tick label size to 15.

Note

You'll notice (especially with larger ticks) that the are not perpendicular to the axes spines. This is because this particular image has been rotated with respect to the standard celestial coordinate axes. This can be seen more clearly if you add the following to your function: `ax.coords.grid(color='gray', alpha=0.5, linestyle='solid')`. Try doing that, adding an optional keyword to your function called 'grid' and enabling this command if it is set.

Tip

To check if a condition is true (e.g, `if condition == True:`), you can just type `if condition:`

It's taken us some time, but this image could now be placed in a scientific publication. And, since we have a handy function for it, we can make images that look this nice on the fly with a short one line command, yet still have a lot of flexibility over many important inputs. And of course, the figure and axes objects are returned, so one could run this function and then continue tweaking the plot after the fact.

Note

As a final note, I want to draw your attention to the fact that once you use the `wcs` projection on some plot, it's no longer a normal ax object, it's a wcsax object. This makes changing certain elements of those axes a little more involved than for a standard one. I use this page and the others linked there when doing so.

Problem 3: Cutouts and Aperture Photometry

When working with astronomical images, like the one we've been using in this lab, it is often advantageous to be working with a cutout – a chunk of the image centered on a certain coordinate, and of a certain size. For example, if there is an HII region or star cluster of interest in the above image, we may like to zoom in on it to examine it more closely.

Now that we've switched over to using celestial coordinates instead of pixels as our projection frame, zooming in on a region is not as simple as slicing our array, e.g., `image[500:700,200:550]`. On the plus side, the framework we'll learn here is very robust, and will allow for all sorts of useful measurements.

To make a cutout, we'll need the `Cutout2D` module within `astropy`, which I'll import below. To provide the position of the cutout, we need to be able to feed in astronomical coordinates. For this, we'll use `SkyCoord`, a module in `astropy.coordinates`. Finally, we'll need to integrate the `units` module in `astropy` to successfully create coordinate objects.

```
from astropy.nddata import Cutout2D
from astropy.coordinates import SkyCoord
import astropy.units as u
```

Let's start with a `SkyCoord` object. There are several coordinate systems used in astronomy, e.g., Galactic coordinates (b, l), Equatorial (RA, DEC). The most common (especially in any extragalactic setting) is RA and DEC (as you can see in the image you've plotted already).

The documentation for `SkyCoord` is solid, and worth reading.

The general way we create these objects is, e.g.,

```
coord = SkyCoord('12:01:53.6 -18:53:11',unit=(u.hourangle,u.deg))
```

Tip

You can input various types of strings and formattings for the coordinates, just be sure to specify the units as shown above. The documentation shows the various methods.

Problem 3.1

In this case, the coordinates I set above are for NGC 4039, which is the smaller of the two galaxies in the image we're using.

Note

If at any point you're trying to make a coordinate object for a well known galaxy/object, try, e.g., `coord = SkyCoord.from_name('NGC 4038')`, and ususally that will work!

In the cell below, use the coordinate we've created, plus a size (use 1x1 arcminutes), and the wcs object for our image, to create a `Cutout2D` object.

```
#cutout = # your code here
```

Now, use your fancy new function to plot the new cutout.

Note

Cutout objects contain the image and their own wcs, accessible via, e.g., `cutout.data` and `cutout.wcs`.

```
# plot it
```

Problem 3.2

We're now going to do some aperture photometry.

Definition

Aperture Photometry is the process of defining a region on an image (generally, but not always circular), and measuring the sum pixel value within that region. The region is known as an aperture, and the "collapsing" of the 2D spatial information about counts in each pixel into a single number is known as photometry.

Below, I provide a new coordinate, this time centered on the region between the two galaxies. Make a new cutout of that region (again, 1x1 arcmin), and plot it.

```
new_coord = SkyCoord('12:01:55.0 -18:52:45',unit=(u.hourangle,u.deg))
```

```
# your code
```

```
#plot it
```

In this region, there are a lot of blobby looking roughly circular sources — Some of the larger ones are HII star forming regions, the smaller ones are likely stars. Later in this lab, we'll use multi-wavelength data to try to suss out what is what.

Often, for calibration purposes, we'd need to create apertures around all those sources in the image. We definitely don't want to do that by hand! Instead, we're going to use the `sep` package.

Note

Simply `pip install sep` inside your `a330` environment terminal to get it installed, you should then be able to import it.

```
import sep
```

There are three steps to performing aperture photometry with `sep`, which are detailed in it's documentation.

- Estimate the background
- Find Sources
- Perform Aperture Photometry

Using the instructions presented in the documentation linked, measure the background of the cutout image, and run the source extractor on it. Don't forget to subtract the background before running the extractor!

To do this, write a function that takes as input the data (in this case, a `cutout.data` object and a threshold scale (to be multiplied by the `globalrms`, and performs these steps, returning the `objects` array. Don't forget to document it!

Hint

Warning

When I ran this, I got an error that my "array was not C-contiguous." I found the solution to this issue in this stackoverflow post. Loosely, `sep` uses C-bindings to actually run the heavy lifting code in C rather than Python (it's faster). This means input arrays must be arranged in physical memory the way C is used to. This particular array was not, but it is easy to order it this way.

```
# Your code

def run_sep(...):
    '''
    DOCSTRING HERE
    '''
    pass
```

Run your function and store the output in a variable called `objects`. You should now have an object array containing many detected sources.

```
# run func
```

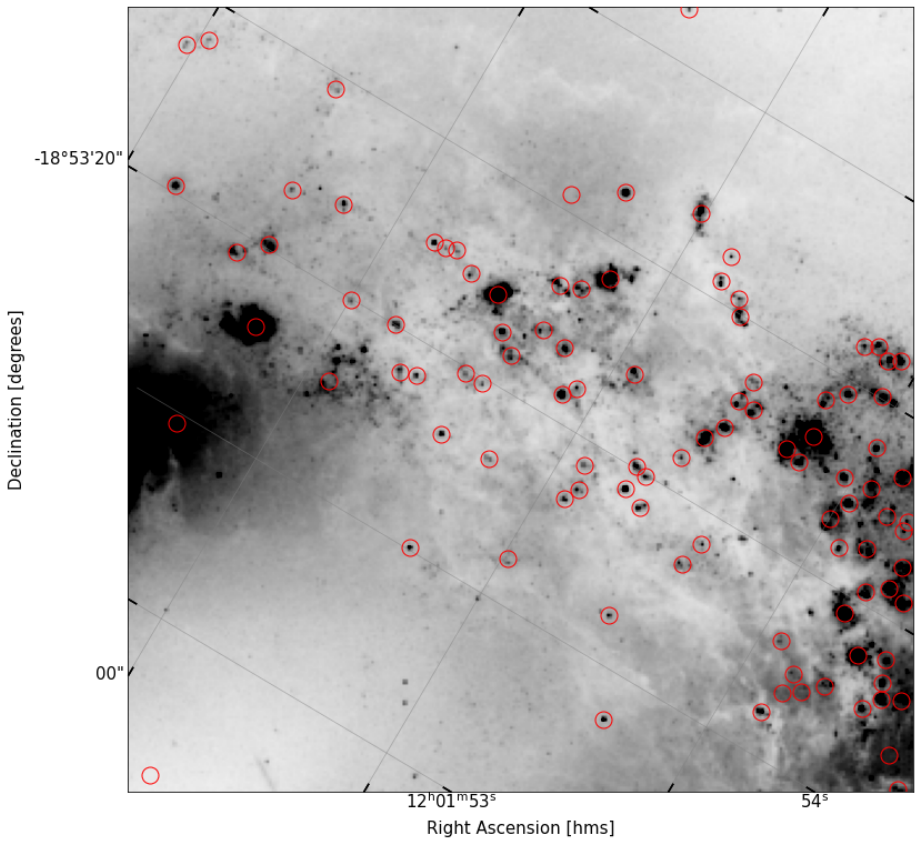
The positions of the determined sources are stored in the output structured array and can be indexed with, e.g., `objects['x']` and `objects['y']`. Replot your image of the cutout, but now circle the positions of all your detected sources. Do they line up with actual point sources in the image?

Hint

```
# Your code
```

```
# Here's my image, for comparison
```

```
[<matplotlib.lines.Line2D at 0x7f8231254520>]
```



It looks like we've done a pretty adequate job finding the sources in this field – there'a a few that got missed, and a few we might want to remove, but overall, this is pretty solid.

We need to start working with these objects we've found. While `sep` can perform aperture photometry itself (reading the docs, you can see it is simple to feed in the objects and a pixel radius), we're going to be a bit more careful about things. To better visualize and work with this data, I'd like it to be a `pandas DataFrame`. We're going to be using those a lot in this course. Converting a `numpy` structured array to a dataframe is simple: I provide the code below. Run it, and then simply type `df` in a new cell to see a nicely formatted pandas table of our objects.

```
import pandas as pd
df = pd.DataFrame(objects)
```

```
df #run this
```

With `sep` providing the first pass, we can cull a few objets from our sample that very clearly are not star forming regions. Using your DataFrame, plot the flux of each detected object using the `flux` column. You should have at least a few that are huge outliers, with dramatically more flux the rest.

```
# plot
```

Write a function called `remove_outliers` that reads in a dataframe and a flux-min and flux-max. It should filter the dataframe to only include fluxes between the input values, and return the new dataframe. Then use this function on your data, choosing an appropriate cutoff.

```
# Your Code

def remove_outliers(...):
    '''
    '''
    pass
```

```
# run the function
```

```
# plot again
```

Re-plot the set of sources you have now, over the data.

```
# plot
```

You should see that some of the circles which were over bright parts of the galaxy are no longer here.

> **Note**
>
> You may find that there are some visible sources which, despite tinkering, don't get caught by `sep`. That's okay – if we really wanted them, we could just go in and put down apertures by hand. Generally when performing a step like this in the field, we have a pre-determined set of points to use, because we have measured fluxes for them, and wish to flux calibrate our data.

Problem 4: Continuum Subtraction

At this point, we have the tools necessary to make flux measurements in apertures (at least, via `sep` – we will also learn how to do this with the `photutils` package). However, R band photometry of HII regions is not exceedingly interesting. More interesting is the flux in $H\alpha$, an emission line caused by Hydrogen recombination. This line (at 6563 Angstrom) is located *within* the R band, and is imaged using a narrow filter compared ot R.

The flux measured by an $H\alpha$ filter contains both the flux from the emission line as well as the underlying continuum — essentially, the starlight from the galaxy. If we can get a clean measure of the flux in $H\alpha$ (sans this continuum), we can make a direct estimate of the star formation rate of the system.

Problem 4.1

To do this, we need to access an $H\alpha$ image, and then subtract off the continuum present (which we'll infer from our R band image). Located in the lab directory is a file called `antenna_Haband.fits`. Use your fits loader function from above to read in this new image, create an equivalent sized and centered cutout to the R band data, and plot it, with the same sources found in the R band circled.

```
# Your Code
```

There's a few interesting things to note right away here. Looking just north of the center of the image, there's now a large amount of flux in $H\alpha$ coming from some blobs which do not appear in the R band. This is likely due to the fact that by zero-ing in on the ionized gas, we can see the large envelopes of gas being lit up by the star formation in this region.

> **Note**
>
> The active merger scenario between these two galaxies is triggering a lot of star formation.

Problem 4.2

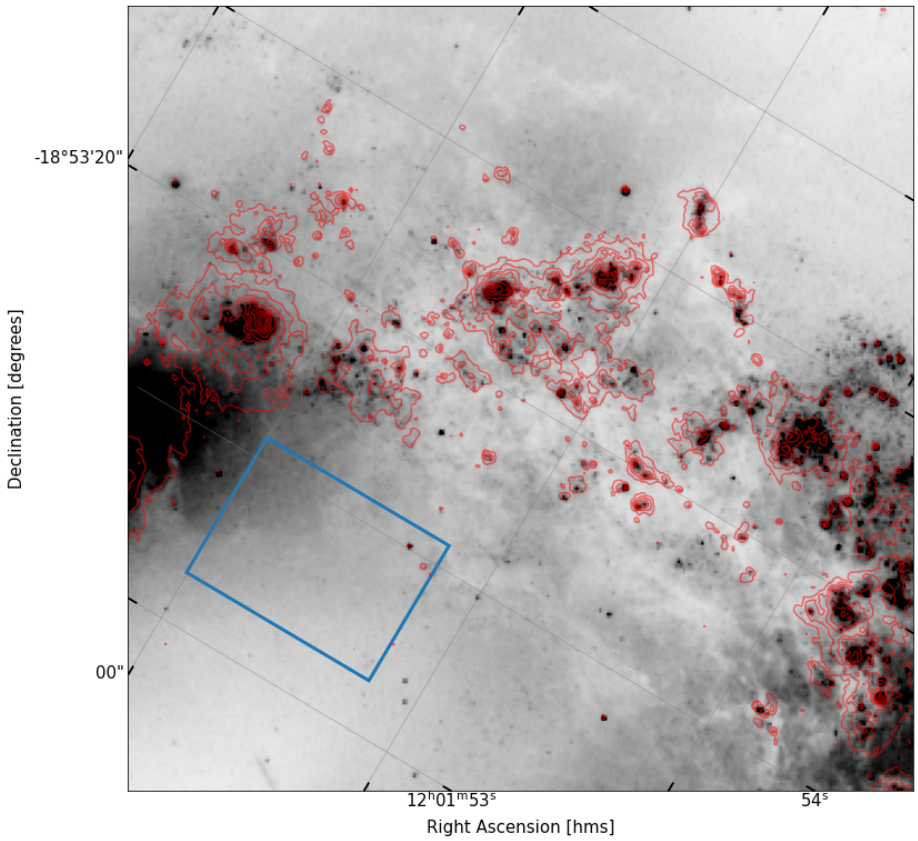
To get a better idea of how the $H\alpha$ flux compares to the R band distribution, use the `plt.contour()` tool to measure contours of $H\alpha$, drawing them over the $H\alpha$ image and tweaking the levels until you think you are well tracing the distribution. Then, plot those contours over the R band data instead.

> **Hint**
>
> It is often beneficial to use `np.logspace()` when defining contour levels, as it allows you to cover large dynamic range with fewer contours. You may also find it helpful to set your contour `alpha` to something < 1, to better see both the image underneath and the contours.

```
# Your Code
```

To help guide you, I've shown what I got for my plot below — you need not emulate it exactly. The blue box shown in the image will be useful to you in the next part of the problem.

```
# My solution
```



We can now see things a lot more clearly! It is obvious that the $H\alpha$ contours trace some of the sources we were seeing in the R band, but that the gas is more extended around these clusters of sources, as we might expect.

Problem 4.3

We must now attempt a continuum subtraction of the data. We can't simply subtract the R band from the $H\alpha$ band, because the R band filter is much wider, and thus for similar exposure times, collects many more photons than the narrower $H\alpha$ filter. Ideally, one could use a set of foreground stars (which are pure continuum sources in both images) to measure fluxes in both, and find a scaling constant. Here, all of the sources we see in this image are most likely actually HII regions. This means if we use them to scale between our images, we will likely oversubtract true flux.

Instead, what I'm going to do here (which may be slightly sketchy), is pick a "blank" region of continuum emission from the R band which has no $H\alpha$ contours, and assert that this patch must have the same flux in both images. In the picture above, I indicated a blue rectangular patch on the sky. This is what we'll be using to measure our continuum-to-narrowband ratio.

To make this patch, we're going to use `SkyRectangularAperture`, from `photutils`. You can pip install `photutils` in your `a330` environment if you don't have it. Then do the following imports:

```
from photutils.aperture import SkyRectangularAperture
from photutils.aperture import aperture_photometry
```

As shown in it's documentation, we need to feed in a `SkyCoord` position, as well as a width and a height. I've provided the new coordinate to use below.

Create a rectangle of your own, measuring $0.27''$ by $0.2''$. You can now use the `aperture_photometry()` function to measure the flux in your aperture, applied to a certain image. Using the documentation as needed, find the flux in this box for both the R band and $H\alpha$ band data, and then determine the ratio between those values. Don't forget about the `wcs`!

```
patch_cent = SkyCoord('12:01:53.7 -18:52:52',unit=(u.hourangle,u.deg))

# Your code
```

We can now use this ratio to perform our subtraction.

Problem 4.4

Now, fill in the function below, which should read in your rectangular patch, the R band image, and the $H\alpha$ image. Within the function, copy in the code that determines the ratio, and then use the ratio you found to scale your R band image, then subtract it from the $H\alpha$ image. The function should return this new image array.

Plot up your continuum subtracted image using your `implot()` function, adding back in the apertures we found earlier and the contours we made from the full $H\alpha$ image.

```
# your code
def continuum_subtract(...):
    '''
    DOCSTRING HERE
    '''
    pass
```

What do you notice about the distribution of apertures with respect to the distribution of $H\alpha$ gas? Is there a strong alignment of the R band sources and the gas? What does this imply about most of the R band sources?

Answer here

Problem 4.5

Lastly, let's load up the B band image. As you probably know, the B band traces bluer light, and thus will more preferentially see young, hot stars (whereas the R band traces the main sequence and turn off stellar distribution).

Load up the `antenna_Bband.fits` image, make a cutout, and plot it below. Use your `load_fits()` function and your `implot()` function. Make a new set of countours from your *continuum subtracted* $H\alpha$ data, and overplot that onto the B band data. What do you see?

```
# Your code
```

In my own image, there are some clusters of B-band flux that align well with the $H\alpha$ contours, and some B band sources out on their own. It is unsurprising that there is a correspondance between them; excess in B light implies the prescence of UV radiation as well, from young O/B stars. It is this radiation responsible for ionizing the gas that is shining in $H\alpha$, so the $H\alpha$-emitting gas should be loosely clustered around sources bright in the B band (the experiment would be even cleaner if we used, say, *GALEX* FUV data).

Bonus Question (up to 3 points)

If you want to take your analysis farther, try measuring some fluxes off of your continuum subtracted $H\alpha$ data, placing several manual apertures down over the regions of highest $H\alpha$ concentration (which also align with B band concentrations).

The measures you get from this will be in counts on the detector. We need to convert these counts into flux units (e.g., erg s^{-1} cm^{-2}). To do this,

- pull the 'PHOTFLAM' keyword from the header of your $H\alpha$ data
- also pull the 'EXPTIME' value from the header.

Start by taking your fluxes in counts, multiplying by the `PHOTFLAM` value, and dividing by the `EXPTIME` value. This puts you in erg s^{-1} cm^{-2} *per Angstrom*. Thus, what we have is technically a *flux density*. To convert to a true flux, we'll simply integrate over the bandpass... but for this example, let's assume a constant flux across the bandpass, which for the ACS 658N filter ($H\alpha$), is 136.27 angstroms.

Once you have fluxes, use the distance to NGC 4038/9 to determine the luminosity in erg/s of your collective set of sources, and then use the $SFR(H\alpha)$ calibration of Kennicut & Bell to convert to an SFR. How does your answer compare to, e.g.,

- The SFR of the Milky Way?
- The SFR of the Orion Nebula?
- The SFR of the Tarantula Nebula?

Lab 3: Building a Photometric Pipeline

In this lab, we'll be using classes and functions to build a pipeline which will automatically extract the fluxes of stars in an image. We're all familiar with aperture photometry, but in this case, we're going to take the additional step of convolving our image with a PSF.

Our pipeline will be split into several steps.

1. Reading in an image
2. Finding the local peaks in the image (the stars)
3. Calculating the centroid of each peak region
4. Convolving with the PSF and extracting flux.

Problem 1

Read in the image `2020-04-15-0001.fits` and plot it, using your handy functions from Lab 2. This image was taken of the M81/M82 field using the Dragonfly Telephoto Array (in a narrowband configuration).

```
# Your solution
```

Problem 2

Our goal is to estimate the PSF of the above image, then measure fluxes of the stars and galaxies here accounting for the PSF.

To start this process, we need to locate the stars in this image. We saw how to segment an image using `sep` last time, but in this lab we are going to carry out this step ourselves, using two methods.

Problem 2.1

Before we do this, we want to take the step of masking out several regions of the image which may register as peaks but which are not nicely isolated stars. In particular, the two galaxies, and the two optical artifacts (actually, detector amp glow) in the bottom center and bottom left of the image.

By querying the image dimensions, construct a masked array which masks out rough patches over these regions we want to avoid. Plot your masked array to demonstrate the appropriate regions have been masked out.

```
# Solution
mask = ...
```

Problem 2.2

Now that we have the appropriate image regions masked, we can move on to the peak finder.

The "fast" or "efficient" method of doing this involves some scipy filtering operations. But for our purposes, the "slow way" (iterating over the image pixels) takes ~few seconds to run, and is worth doing to build intuition.

Complete the function below to find peaks in an image by looping over each pixel and checking its neighbors, with a "peak" being defined as a region of higher flux than all adjacent pixels (i.e., the 8 surrounding pixels). In order to not pick up random noise pixels, also take an input called `threshold`. Within your algorithm, don't return any pixels which are "peaks" but for which the pixel value is below this threshold.

```
# Solution
def find_peaks(image, threshold):
    '''
    Algorithm for finding peaks (above a threshold) in an image
    '''
    #Your code here
    return
```

The looping solution is slow, and will not scale well if we have to run on many images, but for one image is okay.

There are several solutions which generally involve either **filtering** the image or **cross correlating** the image with a template. Here's one such solution — you'll note that it runs much faster than the standard looping method you used above.

```
from scipy.ndimage import maximum_filter

def findpeaks_maxfilter(image, threshold):
    '''
    Algorithm for finding peaks (above a threshold) in an image

    Parameters
    ----------
    image: array_like
        2D array containing the image of interest.
    threshold: float
        minimum pixel value for inclusion in search

    Returns
    -------
    peaks: array_like
        array containing the x and y coordinates of peak regions.
    '''
    neighborhood = np.ones((3,3),dtype=bool) # just 3x3 True, defining the neighborhood
over which to filter
    # find local maximum for each pixel
    amax = maximum_filter(image, footprint=neighborhood) #max filter will set each 9-
square region in the image to the max in that region.

    peaks = np.where((image == amax) & (image >= threshold)) #find the pixels
unaffected by the max filter.
    peaks = np.array([peaks[0],peaks[1]]).T
    return peaks
```

Let's take a moment to understand how this algorithm works. The key is in the `maximum_filter()` step. Filtering a 2D image is a process carried out in fourier space, which is what allows scipy to carry it out quickly. But what is maximum filtering?

> **Definition**
>
> Maximum Filtering is the process by which all pixels in local neighborhoods within an array are raised to the maximum value of any pixel in that neighborhood

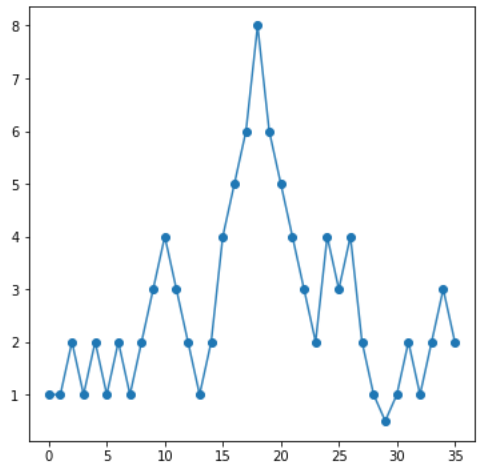
Let's look at a 1D case. Below, I define a 1D array that has some peaks in it.

```
array_1dpeaks =
np.array([1,1,2,1,2,1,2,1,2,3,4,3,2,1,2,4,5,6,8,6,5,4,3,2,4,3,4,2,1,0.5,1,2,1,2,3,2])
```

Our data looks like this:

```
fig, ax = plt.subplots(figsize=(6,6))
ax.plot(array_1dpeaks,lw=1.5,alpha=0.9)
ax.plot(array_1dpeaks,'o',color='C0')
```

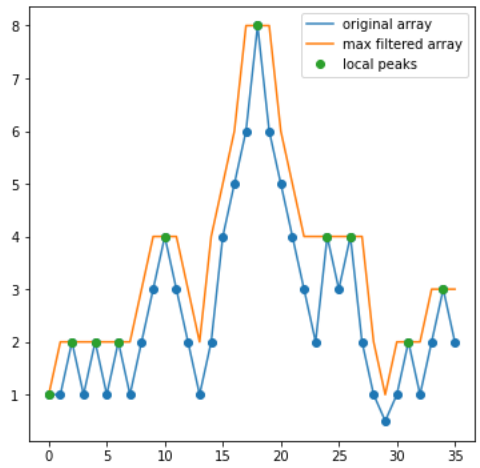
```
[<matplotlib.lines.Line2D at 0x7fc8ecf17ee0>]
```



Let's now run the maximum filter on this data and plot it's output. I'm going to pick a neighborhood of 3, which means +/- 1 pixel around each location.

```
mf = maximum_filter(array_1dpeaks, footprint=np.ones(3,dtype=bool))
```

```
fig, ax = plt.subplots(figsize=(6,6))
ax.plot(array_1dpeaks,lw=1.5,alpha=0.9,label='original array')
ax.plot(array_1dpeaks,'o',color='C0')
ax.plot(mf,label='max filtered array')
eq, = np.where(array_1dpeaks==mf)
ax.plot(np.arange(len(mf))[eq],mf[eq],'o',label='local peaks')
ax.legend();
```



What the filtering has done is for every 3 pixel neighborhood across this array, it's raised the value of all three pixels to the maximum value across the three. So we see that anywhere the three pixels were, e.g., (1,2,1), they are all now 2. What you should notice looking at this plot is that the max filtering has also identified true peaks in our data! Notice that the only spots where the orange curve (the max filtered version of the data) is equal to the original array is exactly at locations that are local maxima. This is because when applying max filtering to an array, the only values *unchanged* by the filtering are those that *are* local maxima (in the neighborhood defined).

And thus, we have our peaks! All we need to do is find out `where()` the max filtered array equals the original array (pixel by pixel). Of course, we can also put in a threshold (like above, maybe 2.5) to ensure low level noise doesn't enter in. This is why the `findpeaks_maxfilter()` function has a threshold option as well.

Note

You may notice that the first index in the array is marked as a peak, despite not being one. Edges are always troublesome with these algorithms, and they normally have multiple options for how edges are handled.

Problem 2.3

For the remainder of this assignment, you can use either your peak finder, or the faster one provided. But before you decide, as a quick check, use the `%%timeit` magic command in your notebook to test how fast the two algorithms are respectively. If you were working with a sample of 1000 images, how long would the looping algorithm take compared to the max-filtering case?

```
%%timeit
# your code here
```

```
%%timeit
# your code here
```

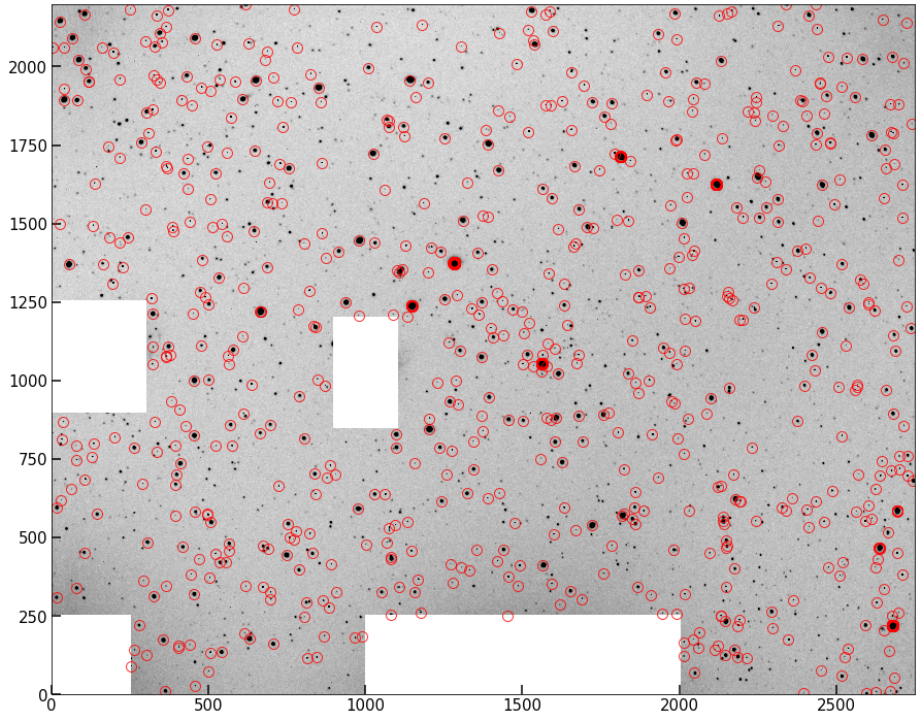
```
# how long for 1000 images with each function?
```

Problem 2.4

Run your peakfinder on your masked image, and assemble the list of peaks you'll use moving forward. I suggest using 3σ above the mean image value as a solid threshold.

Plot up the original image, but use our "source circling" technique from last lab to circle all the peaks found in the masked image. I show mine below.

```
#My peaks below
```



Problem 2.5

This should look pretty good – most of what's circled above is clearly a star/point source in the image. However, one problem with this method is that single hot pixels are going to be registered as peaks, even via the more clever algorithm. We need a way to eliminate these from our sample before moving on.

Copy down your peak-finder and add in something that checks that not only is there a true peak, but that at least 4 of the pixels around the peak are also elevated in flux (I used 0.5 times the peak flux). The easiest way is to loop over the peaks after they're found and institute the check — there are far fewer peaks than pixels, so this doesn't significantly affect the runtime. But feel free to find a cleverer solution!

> ⚠ **Warning**
>
> Be careful with transpositions in this problem. when you plot coordinates, you plot(x,y), but when you index the image, you index image[y,x]. Be tracking which is which! As a note, the original output of `findpeaks_maxfilter()`, peaks, was of the form [array([y1,y2,y3,...]),array([x1,x2,x3,...])].

```
def findpeaks_maxfilter(image, threshold):
    '''
    Algorithm for finding peaks (above a threshold) in an image
    '''
    # add in check against single pixel peaks
    return ...
```

Re-find your peaks using your newer, better algorithm, and plot them below as before.

You should notice that we've decreased our total number of peaks. But you should find that now, everything currently circled looks like a bright, "resolved" star. (resolved insofar as the PSF is spreading the light of the star over multiple pixels).

Problem 2.6

In your the image above, you should see that ~8-10 stars look like they are circled by several very closely overlapping circles all targeting the same star. Infer (or investigate and determine) why this has happened, and write your answer below.

answer here

Problem 3

We now have a function that can return the peaks in a given image. Our next step is going to be to estimate the exact center of those peaks (stars) using their **centroid**.

> **Definition**
>
> The centroid is the light-weighted-mean of a set of pixels. It is not always the maximum-valued pixel, and is determined to sub-pixel accuracy.

Many of you have seen the centroid formula, but as a reminder, it looks like this (in 1D):

$$x_{\rm com} = \frac{\sum x_i F_i}{\sum F_i},$$

where x_i are the positions and F_i are the fluxes at those positions.

In 2D, when working with images, the x and y centers of mass are independent, and the 2D centroid is just the location ($x_{\rm com}$, $y_{\rm com}$).

Problem 3.1

Finish the function below, which should read in an image and a peak location returned by your peak finder, then create a window of NxN pixels around it (user-settable), and determine the centroid of this window. The x, y locations of this centroids should be returned.

One subtlety — We want to use a window size greater than 1 pixel on either side of each peak. But because our peak finder is likely to find peaks near the edge of the detector (both because it needs only 1 pixel-thick borders and because it handles edges), if we write a centroid function that, e.g., uses a 10x10 pixel window, we'll end up trying to index over the edge of the original image. Because of this, your function should raise an exception if a peak position is entered whose distance from an edge is less than half the window size.

```
# Solution
def centroid_peak(image,peak_x,peak_y,window=10):
    '''
    Given an image and list of peak positions, determine the centroid in the region of
each peak.
    FINISH DOCSTRING
    '''
    # Your code
    return x_center, y_center
```

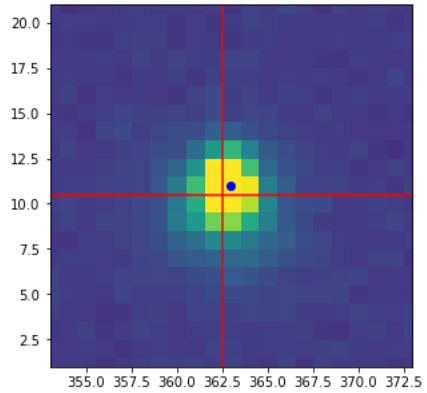
Problem 3.2

Use your `centroid_peak` function to confirm that the algorithm is working by testing it on a few individual peaks from your peak list, and make a plot showing the window region and the determined centroid (along with the location of the input peak). I'm leaving behind a demo of what I mean below. The blue point is the pixel with the peak flux, while the crosshairs center on the determined centroid

> **Note**
>
> It's ok if the centroid is not at what appears to be the center of the light distribution of the star. Often due to seeing and detector effects, along with tricks of the stretch you happen to be using, the centroid doesn't look perfectly centered.

```
# Example below
```

```
(1.0, 21.0)
```



Problem 3.3

Now do this for all peaks in your image. You may need to use `try` and `except` blocks to skip any peaks that raise errors in your centroid code for being too close to the edge. I'll leave it to your discration whether you handle this by dropping that peak from the list and moving on, or by trying a smaller window first. (The second requires a bit more boiler plate but would allow you to retain a larger selection of stars).

```
# centroid all peaks
```

Problem 3.4

If you recall from above, you determined why the peak algorithm occasionally marked a bunch of pixels as peaks within the same star, which shouldn't happen. It should be clear from your answer that these stars will be unusable for the purposes of measuring the PSF of stars in the image. We thus need to remove these ~8-10 stars from our sample.

Write a function which takes in the list of centroids, and identifies these cases. The easiest way to do this is to simply iterate through the list of centroids and compute the distance to all other centroids in the list. Any centroid which has another centroid very nearby (within, say, 5 pixels) should be removed from our sample.

This function should return the final list of centroids for use. Plot them over the data to confirm these "stacked" peak cases have been removed.

```
# clean list of centroids
```

Problem 4

Now that we have well-centered centroids for a large sample of stars in our image, we can go about estimating the PSF. The first step in this process is estimating and subtracting the background. But wait! We've been using a raw telescope image for this lab so far... we need to perform the basic calibration steps of dark subtraction and flatfield correction before we can move on.

Problem 4.1

Write a calibration function that reads in the image, along with the provided `flat` and `dark`, and does the basic calibrations on the image. Confirm that the mean/median image value (a rough estimate of the background) drops significantly. The image should also look flatter, without the strong amp glow features.

```
def calibration(image,dark,flat):
    '''
    Docs
    '''

    return
```

Problem 4.2

We now need to estimate the background. Performing that fit (generally a low order 2D polynomial of some kind) is beyond the scope of this lab, but we encourage you to look into this step more if you are interested.

Instead, we're going to return to the `sep` package and take advantage of its background estimation feature. Recall it reads in the base image array and a separate mask; we'll use the same mask we created earlier in the assignment, which will cut out the galaxies. The amp glow should be largely removed by the dark subtraction, but we'll leave those areas masked as well just in case.

Using the same stucture as in Lab 2, calculate the background using sep.

> **Warning**
>
> Don't forget about the C order switch.

```
import sep

# your code
```

Plot the spatially varying background, using `back = bkg.back()`

We can see two splotches where the galaxies were masked out and the algorithn attempted to interpolate over them. There also appears to be a large gradient in the background (on the order of ~150 counts), likely due to a detecter issue the instrument was having at the time. In any case, we want to subtract this background from our image. Do that below, and `implot` your final subtraction, with your final set of centroid positions circled.

Problem 4.3

Armed with a dark-subtracted, flat-fielded, background-subtracted image, as well as with a list of centroids corresponding to stars in our image, we are ready to estimate the PSF.

There are two main functional forms typically used to fit star profiles: 2D Gaussians, and Moffat profiles (which combines the shapes of a Gaussian and Lorentzian to best match both the inner and outer regions of the PSF).

We're going to use the `Gaussian2D` class from `astropy` to do this:

```
from astropy.modeling.functional_models import Gaussian2D
```

For each star, a Gaussian2D profile (normalized) will be used "as the PSF". The parameters we need to know for this profile are x, y, for which we'll use the centroids we calculated earlier, the amplitude (set by the normalization), and σ_x, σ_y, the standard deviations in the two axes. For this lab, we're going to assume our stars are circular ($\sigma_x = \sigma_y$). This is a strictly incorrect, but not a bad assumption for most cases. All other optional arguments we won't need, primarily due to the assumption of circularity.

> **Note**
>
> We are going to make a point estimate of the "size" of the stars in our image, which constrains us from using a more fancy model for the PSF. An example of a more sophisticated setup would be *fitting* a Gaussian or Moffat profile to every star, and in a Bayesian framework marginalizing over the stars to determine the best-fit PSF (including ellipticity, etc) for the image, or, even fancier, interpolating a PSF model which varies over the detector.

PSF photometry works by multiplying the *data* (say, a cutout around a star) by the *estimated PSF* during the fluxing stage. Instead of picking a radius and performing aperture photometry (which includes fully all pixels within the aperture and throws out all pixels beyond), this method attempts to weight each pixel fractionally by how likely it is to be stellar flux, with the weighting coming from the PSF of the detector. This means further pixels may still be included, but will contribute less than pixels near the center of the star.

The formula for measuring the PSF flux of a star is

$$f_{\mathrm{PSF}} = \frac{\sum \hat{f}_i p_i}{\sum p_i^2},$$

where $\hat{f}_i$ are the fluxes in your image and p_i is your PSF estimate. This formula should be reminiscent of the centroiding formula; it's a similar weighting scheme.

`Gaussian2D` is a class, but we want to interact with it pretty simply, and have simplified inputs. I've made a quick wrapper function below which allows us to enter a single σ and then x, y grids created via `np.meshgrid()`, and creates the Gaussian and evaluates it on our grid.

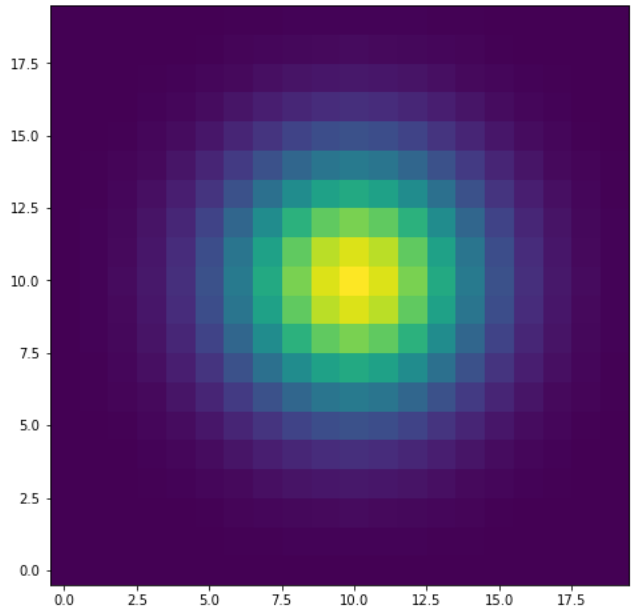
```
def eval_gauss(x_arr,y_arr,sigma,mu_x,mu_y):

    g = Gaussian2D.evaluate(x=x_arr,y=y_arr,amplitude=1,theta=0,x_mean=mu_x,
                    y_mean=mu_y,
                    x_stddev=sigma,
                    y_stddev=sigma)
    g/=np.sum(g)
    return g
```

```
xx, yy = np.meshgrid(np.arange(0,20),
                     np.arange(0,20))
model = eval_gauss(x_arr=xx,y_arr=yy,sigma=3,mu_x=10,mu_y=10)
```

```
fig, ax = plt.subplots(figsize=(8,8))
ax.imshow(model,origin='lower')
```

```
<matplotlib.image.AxesImage at 0x7fdba887a9a0>
```



As we can see, I now have a model for the PSF which I can easily create for given inputs. We're going to do this for cutouts around each star, and instead of a random σ, we're going to estimated it using the second moment (moment of inertia) of the star itself.

The formula for this (from Markevich et al. 1989) is

$$\sigma_x = \left[\frac{\sum x_i^2 \hat{f}_i}{\sum \hat{f}_i} - x_{\rm com}^2 \right]^{1/2}$$

$$\sigma_y = \left[\frac{\sum y_i^2 \hat{f}_i}{\sum \hat{f}_i} - y_{\rm com}^2 \right]^{1/2}$$

In this case, we'll need to use `meshgrid()` directly within our second moment function, as you can see it depends on the difference between the pixels and the centroid.

```
def second_moment(image_cutout,xx,yy,centroid_x,centroid_y):
    '''
    Measure the second moment of the light distribution of star cutouts
    '''
    # Your code
    return sigma_x, sigma_y
```

Within 10%, this calculation, which requires no fitting, tells us σ (under the assumption the distribution is Gaussian). Thus, by running out image cutouts through this function, we can derive for ourselves a good σ to choose in our Gaussian model of the PSF.

Problem 5

Last but not least, it's time to put eeeeeevvvvveeerrrryyyyttthhhiiiinnnggggg together.

Let's sum up what we've done so far. In individual chunks and functions (albeit not in this order), we've

1. Read in telescope imaging, along with dark and flat frames, and perform standard calibrations
2. Used `sep` to estimate and subtract the spatially varying background
3. Written a peak-finder which locates peaks in the image
4. Gone through those peaks and removed hot pixels
5. Centroided the stars of those peaks and removed those which are associated with the same star
6. Wrote code which can estimate the PSF from each star, and generate a gaussian model of it
7. Perform PSF photometry by multiplying the image flux by the PSF model

Right now, all this code is scattered throughout different cells and functions. Our one goal in this problem is to integrate these many steps into an easy to use **class** which automates this process, allowing for user input when necessary.

Your goal is to write a single class, `PSFPhotometry`. It should have methods which handle the steps we've put together in this lab.

One valuable aspect of the class format is that for information that's useful throughout the redution (like the peak and centroid lists), we can set these to be class `attributes`. This makes them accessible from anywhere in the class, and saves us the trouble of having to constantly return and read in these values.

Once you've created your pipeline, run our image through it and output a catalog of positions and fluxes. I've started off the basic structure of this class below.

```
class PSFPhotometry():
    '''
    Docstring
    '''
    def __init__(self,...):
        '''
        Docstring
        '''
        pass
    def calibrate(self,...):
        '''
        Docstring
        '''
        pass
    def ...
```

By Imad Pasha & Marla Geha