SKA data challenge workflow + container setup

- Using containers is all about reproducibility
- That means your code and workflow also need to be reproducible
- Shipping a container with code that isn’t documented and can’t be run or understood isn’t helpful
- I’ll touch on a few things to think about whilst using containers in your workflows

SKA data challenge workflow + container setup

- Last week we saw how to run a full science workflow from inside a container

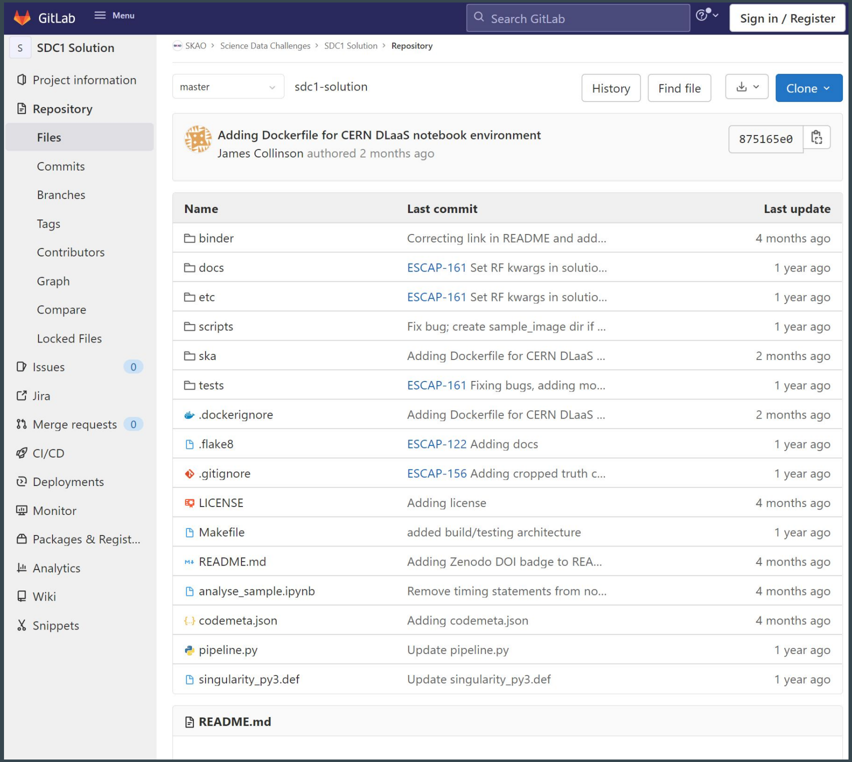
  Link in the program: https://www.youtube.com/watch?v=pcNzLk7Ohj8

  - ➢ This involved primary beam correction, source-finding, training a machine learning model, source classification and scoring results.
  - ➢ Hopefully you've had a play with that - the best way to learn about containers is to practise using them

```python
if __name__ == "__main__":
    """
    Run through a simple analysis workflow to solve SDC1

    1) Preprocess images (correct PB) and crop out the training area for
        building ML model
    2) Find sources in the PB-corrected training images
    3) Train a classifier for each band to predict the class of each source
    4) Find sources in the full PB-corrected image
    5) Predict the class of each source
    6) Calculate the score for each image band, and write out a short report
    """
    time_0 = time()
    # 1) Create in-memory representation of image and preprocess
    print("\nStep 1: Preprocessing; elapsed: {:.2f}s".format(time() - time_0))
    sdc1_image_list = []
    for freq in FREQS:
        new_image = Sdc1Image(freq, image_path(freq), pb_path(freq))
        new_image.preprocess()
        sdc1_image_list.append(new_image)

    # In data/images, we now have PB-corrected and training images for each band

    # 2) Source finding (training):
    print("\nStep 2: Source finding (train); elapsed: {:.2f}s".format(time() - time_0))
    sources_training = {}
    for sdc1_image in sdc1_image_list:
        source_finder = SourceFinder(sdc1_image.train)
        sl_df = source_finder.run()
        sources_training[sdc1_image.freq] = sl_df
```

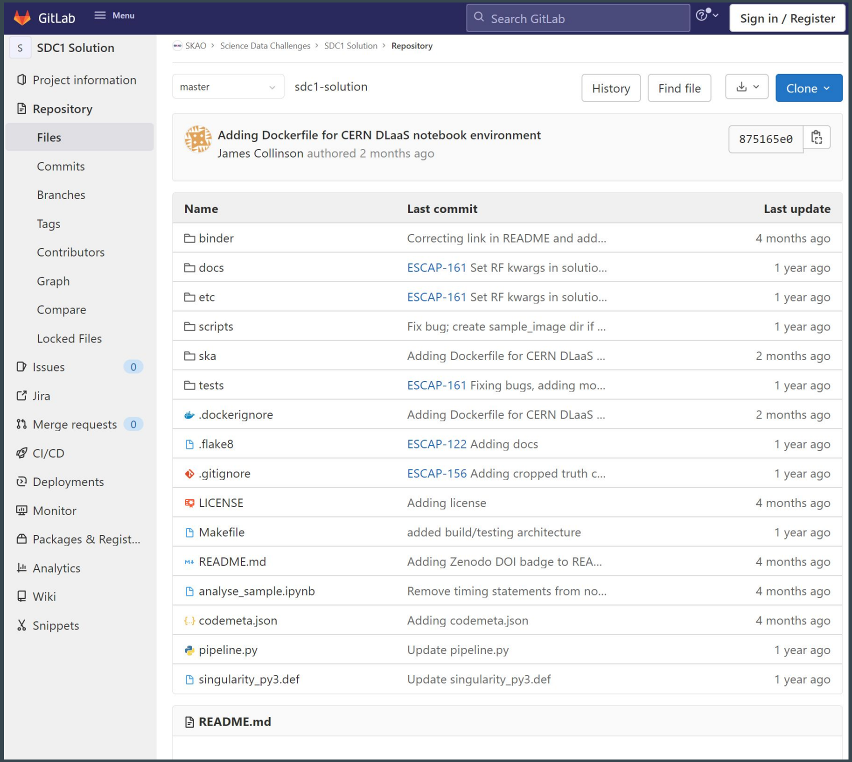
SKA data challenge workflow + container setup

- Best practises in code structures
- Best practises in Gitlab repo
- Using make files
- Using aliases
- Using .dockerignore
- Using unit tests



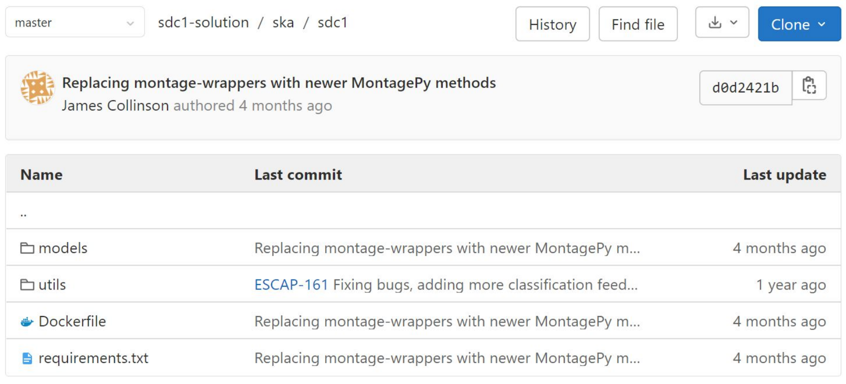
Folder structures

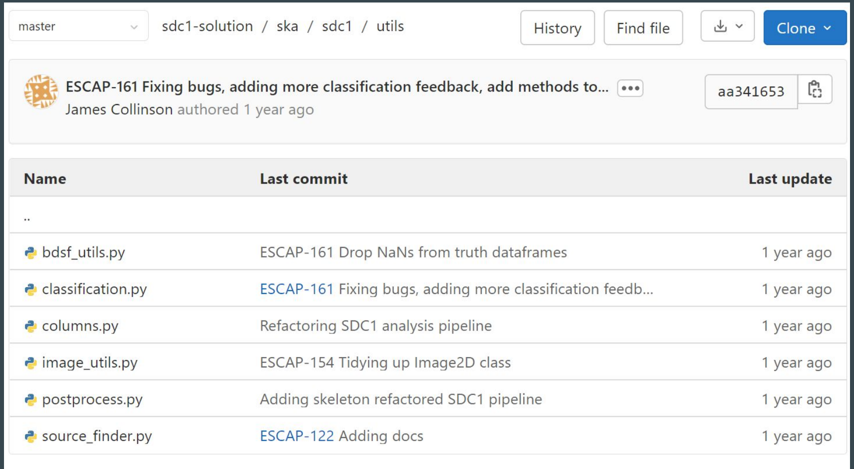
- Your repo is probably capable of doing a variety of things, so organise folders is important
- How you organise things is subjective, but hopefully intuitive
- /docs is self explanatory. We implement automatic documentation using the python package Sphinx. This means docs are updated automatically as you update your code
- /ska contains the actual code and the docker build files required to make the container
- /scripts contains ready to run scripts that could do a variety of things. They are lightweight and easy to interpret, since all the functions are in the /ska folder. Users will look at these rather than the code in /ska
- /tests is used to run unit-tests. These are lightweight functions that ensure your code is working, without having to do any heavy lifting such as accessing large data sets
- /etc contains additional code that is not part of your software, but is needed for additional functionality and ease of use, such as aliases



Main code folder - /ska/sdc1

- All about splitting your code up into functions related to specific tasks
- Users don’t need to come in here and look at things unless they are adapting or adding functionality
- We include the Dockerfile in here
- We also have a version that runs in a Jupyter notebook: /ska/notebook





Dockerfile

- As you've seen in previous talks, this is the file that is used to build your container
- We use a requirements.txt file which has all the Python packages in it

master sdc1-solution / ska / sdc1 / **Dockerfile**

Find file | Blame | History | Permalink

Replacing montage-wrappers with newer MontagePy methods
James Collinson authored 4 months ago

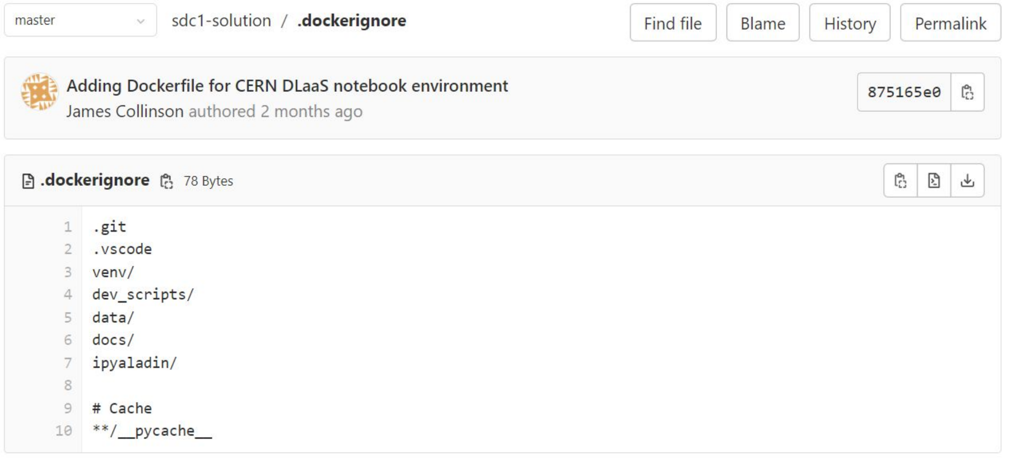
d0d2421b

Dockerfile 981 Bytes

```
FROM ubuntu:18.04

ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update \
    && apt-get --yes install --no-install-recommends \
    bison \
    build-essential \
    cmake \
    eog \
    flex \
    g++ \
    gcc \
    gettext-base \
    gfortran \
    git \
    libarmadillo-dev \
    libblas-dev \
    libcfitsio-dev \
    libfftw3-dev \
    libgsl-dev \
    libgtkmm-3.0-dev \
    libhdf5-serial-dev \
    liblapacke-dev \
    liblog4cplus-1.1-9 \
    liblog4cplus-dev \
    libncurses5-dev \
    libpng-dev \
    libpython3-dev \
    libreadline-dev \
    libxml2-dev \
    openssh-server \
    python3.6 \
    python3-pip \
    python3-tk \
    python3-setuptools \
    subversion \
    vim \
    wcslib-dev \
    wget \
    screen

# python3 requirements
COPY ./requirements.txt /tmp/
RUN python3.6 -m pip install --upgrade pip
RUN python3.6 -m pip install -r /tmp/requirements.txt

ENV LD_LIBRARY_PATH=${LD_LIBRARY_PATH}/usr/local/lib

WORKDIR /opt/

ENTRYPOINT /bin/bash
```

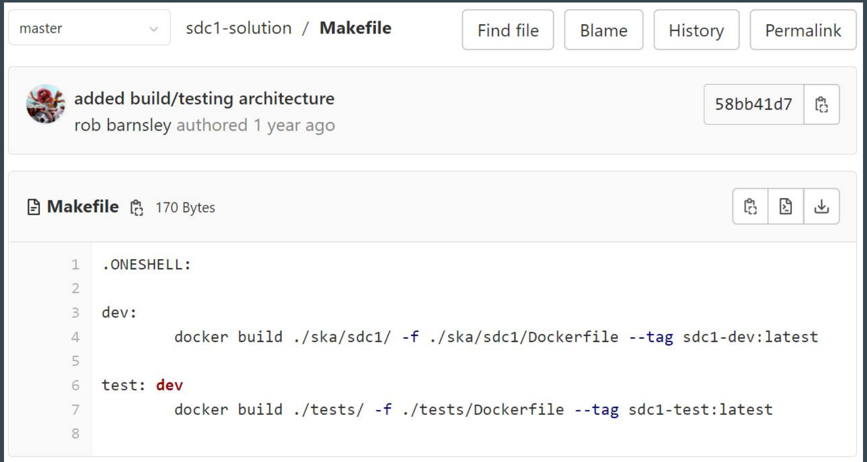
Dockerignore file

- This lets you specify parts of your repository that don’t need to be included when you build your container
- Helps keep your container lightweight



Makefile

- The makefile is used to make long Docker build commands simple to implement
- A user just types “make dev” - as stated in the README, to build the container
- Can be used to build the test container for running unit-tests, or other versions for more complex software



Aliases

- A friendly addition to make your repositories easy to use for people who are less familiar with Docker commands
- They make complex commands easy to execute, avoiding human errors which lead to reproducibility issues
- Use them for commands that don’t need to be changed, such as building, starting, stopping and executing containers
- Aliases and associated Docker scripts are neatly tucked away in the /etc folder, not associated with the code (/ska & /scripts) of your workflow

sdc1-solution / etc / **aliases**

aliases 435 Bytes

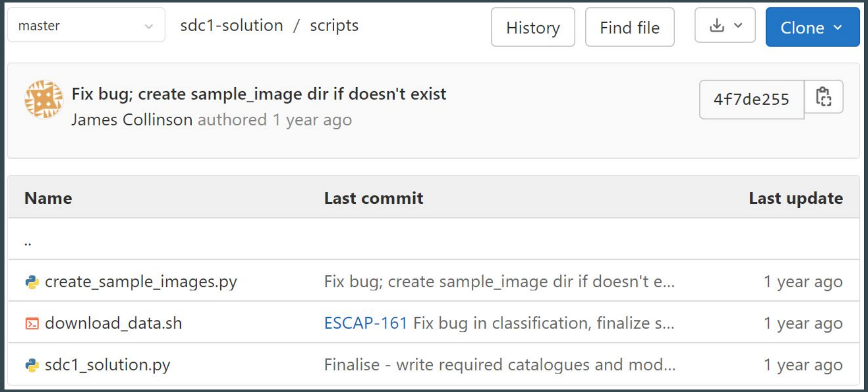
```
#!/bin/bash

alias sdc1-run-unittests="/bin/bash $SDC1_SOLUTION_ROOT/etc/run_unittests.sh"
alias sdc1-start-dev="/bin/bash $SDC1_SOLUTION_ROOT/etc/init_dev.sh"
alias sdc1-exec-dev="docker exec -it sdc1-dev /bin/bash"
alias sdc1-stop-dev="docker stop sdc1-dev"
alias sdc1-start-test="/bin/bash $SDC1_SOLUTION_ROOT/etc/init_test.sh"
alias sdc1-exec-test="docker exec -it sdc1-test /bin/bash"
alias sdc1-stop-test="docker stop sdc1-test"
```

init_dev.sh 391 Bytes

```
#!/bin/bash

docker container stop sdc1-dev >> /dev/null 2>&1
docker container rm sdc1-dev >> /dev/null 2>&1

docker run --name sdc1-dev -d -t \
    -v $SDC1_SOLUTION_ROOT/data:/opt/data \
    -v $SDC1_SOLUTION_ROOT/docs:/opt/docs \
    -v $SDC1_SOLUTION_ROOT/scripts:/opt/scripts \
    -v $SDC1_SOLUTION_ROOT/ska:/opt/ska \
    -v $SDC1_SOLUTION_ROOT/tests:/opt/tests \
    sdc1-dev:latest
```

/scripts

- They are lightweight and easy to interpret, since all the functions are in the /ska folder
- You’ve seen and interacted with these in my demo video from last week
- They can be easily tweaked by the users to get desired outcomes for their specific needs
- Users don’t need to go into /ska to access the bulk of the code, they can interact with these easier



/scripts/sdc1-solution.py

- This runs the whole workflow, and is broken down into steps
- Only ever references functions and data, no “core code” in this Python script, that is all in /ska
- Designed to be an easy-to-read code, where variables can be changes, or additional steps can be implemented

sdc1_solution.py 7.09 KB

```python
import os
from pathlib import Path
from time import time

import numpy as np
from ska_sdc import Sdc1Scorer
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

from ska.sdc1.models.sdc1_image import Sdc1Image
from ska.sdc1.utils.bdsf_utils import cat_df_from_srl_df, load_truth_df
from ska.sdc1.utils.classification import SKLClassification
from ska.sdc1.utils.source_finder import SourceFinder

# Challenge frequency bands
#
FREQS = [560, 1400, 9200]


# Input data paths; assumes defaults from download_data.sh
#
def image_path(freq):
    return os.path.join("data", "images", "{}mhz_1000h.fits".format(freq))


def pb_path(freq):
    return os.path.join("data", "images", "{}mhz_pb.fits".format(freq))


def train_truth_path(freq):
    return os.path.join("data", "truth", "{}mhz_truth_train.txt".format(freq))


def full_truth_path(freq):
    return os.path.join("data", "truth", "{}mhz_truth_full.txt".format(freq))


# Output data paths
#
def train_source_df_path(freq):
    return os.path.join("data", "sources", "{}mhz_sources_train.csv".format(freq))


def full_source_df_path(freq):
    return os.path.join("data", "sources", "{}mhz_sources_full.csv".format(freq))


def submission_df_path(freq):
    return os.path.join("data", "sources", "{}mhz_submission.csv".format(freq))
```

```python
if __name__ == "__main__":
    """
    Run through a simple analysis workflow to solve SDC1

    1) Preprocess images (correct PB) and crop out the training area for
        building ML model
    2) Find sources in the PB-corrected training images
    3) Train a classifier for each band to predict the class of each source
    4) Find sources in the full PB-corrected image
    5) Predict the class of each source
    6) Calculate the score for each image band, and write out a short report
    """
    time_0 = time()
    # 1) Create in-memory representation of image and preprocess
    print("\nStep 1: Preprocessing; elapsed: {:.2f}s".format(time() - time_0))
    sdc1_image_list = []
    for freq in FREQS:
        new_image = Sdc1Image(freq, image_path(freq), pb_path(freq))
        new_image.preprocess()
        sdc1_image_list.append(new_image)

    # In data/images, we now have PB-corrected and training images for each band

    # 2) Source finding (training):
    print("\nStep 2: Source finding (train); elapsed: {:.2f}s".format(time() - time_0))
    sources_training = {}
    for sdc1_image in sdc1_image_list:
        source_finder = SourceFinder(sdc1_image.train)
        sl_df = source_finder.run()
        sources_training[sdc1_image.freq] = sl_df

        # (Optional) Write source list DataFrame to disk
        write_df_to_disk(sl_df, train_source_df_path(sdc1_image.freq))

        # Remove temp files:
        source_finder.reset()

    # <Additional feature engineering of the source DataFrames can be performed here>

    # 3) Train classifiers for each frequency's source DataFrame:
    print("\nStep 3: Training classifiers; elapsed: {:.2f}s".format(time() - time_0))
    classifiers = {}
    for freq, source_train_df in sources_training.items():
        # Load truth catalogue for the training area into memory
        train_truth_cat_df = load_truth_df(train_truth_path(freq), skiprows=18)

        # Construct and train classifier
        classifier = SKLClassification(
            algorithm=RandomForestClassifier,
            classifier_kwargs={"n_estimators": 100, "class_weight": "balanced"},
        )
        srl_df = classifier.train(
            source_train_df, train_truth_cat_df, regressand_col="class_t", freq=freq
        )

        # Store model for prediction later
```

Unit tests

- /tests is used to run unit-tests. These are lightweight functions that ensure your code is working, without having to do any heavy lifting such as accessing large data sets.
- For this purpose, very small cutout images (less than 1 MB) are in this folder, and there is a separate Docker-build to run these.
- Unit-tests should always be shipped as fully working bug-free, so users can test if they have their container set-up properly in minutes before having to execute a workflow that could take days.
- What happens? The expected outputs are tested if they exist, and if so, unit-tests are reported as passed.

  In our case, it tests if the primary beam corrected images are created, if source-finding catalogues are created, if machine learning models are created, and if output scoring files are created.

test_sdc1_image.py 2.81 KB

```python
import os

from ska.sdc1.models.sdc1_image import Sdc1Image


class TestSdc1Image:
    def test_preprocess_simple_pb(
        self,
        images_dir,
        test_sdc1_image_image_small_name,
        test_sdc1_image_pb_image_name,
    ):
        """
        Test preprocess with a small test image, with a simple PB correction
        """
        train_file_expected = test_sdc1_image_image_small_name[:-5] + "_train.fits"
        pbcor_file_expected = test_sdc1_image_image_small_name[:-5] + "_pbcor.fits"
        test_image_path = os.path.join(images_dir, test_sdc1_image_image_small_name)
        pb_image_path = os.path.join(images_dir, test_sdc1_image_pb_image_name)

        # Before running preprocess, the segment and train files shouldn't exist
        for expected_file in [train_file_expected, pbcor_file_expected]:
            assert os.path.isfile(os.path.join(images_dir, expected_file)) is False
        sdc1_image = Sdc1Image(560, test_image_path, pb_image_path)
        sdc1_image.preprocess()

        # Check files have been created
        for expected_file in [train_file_expected, pbcor_file_expected]:
            assert os.path.isfile(os.path.join(images_dir, expected_file))

        # Delete them again
        sdc1_image._delete_train()
        sdc1_image._delete_pb_corr()

        # Verify
        for expected_file in [train_file_expected, pbcor_file_expected]:
            assert os.path.isfile(os.path.join(images_dir, expected_file)) is False
```

Reproducibility

- Reproducibility is more than just presenting your code inside a container
- Think about how a non-expert user sees your Gitlab, how they can reproduce your result, and how they can make use of your workflow for other purposes
- Hopefully you’ve seen how containers make life a lot easier for you and your users

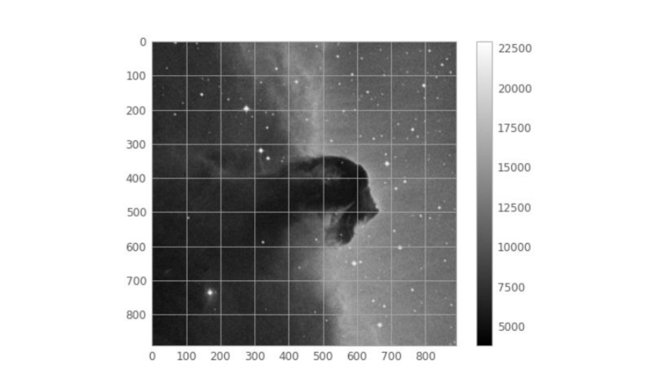
Homework option 1

- I’m sure you all have code + workflows out there that can be implemented inside a container
- Start simple, this could be creating a figure for a paper using some data
- Write your Dockerfile to include the dependencies and build the container
- Try out the code inside, how are you going to get your data to plot? Do you get the expected output?
- Make a Git repository for the workflow with the mindset of a non-expert user coming to reproduce your plot/result
- Add some documentation (at least a README) on how a user can build, run, exec the container, and run code inside
- Can you create makefiles or aliases to make these commands easier to type?
- If you get stuck or want feedback, let us know and we can have a look and try running your code

Homework option 2

- If you would rather test out a really simple workflow, try implementing this plot from the Astropy docs inside a container

https://docs.astropy.org/en/stable/generated/examples/io/plot_fits-image.html



Read and plot an image from a FITS file

This example opens an image stored in a FITS file and displays it to the screen.

This example uses **astropy.utils.data** to download the file, **astropy.io.fits** to open the file, and **matplotlib.pyplot** to display the image.

By: Lia R. Corrales, Adrian Price-Whelan, Kelle Cruz

License: BSD

Set up matplotlib and use a nicer set of plot parameters

```
import matplotlib.pyplot as plt
from astropy.visualization import astropy_mpl_style
plt.style.use(astropy_mpl_style)
```

Download the example FITS files used by this example:

```
from astropy.utils.data import get_pkg_data_filename
from astropy.io import fits

image_file = get_pkg_data_filename('tutorials/FITS-images/HorseHead.fits')
```

Use **astropy.io.fits.info()** to display the structure of the file:

```
fits.info(image_file)
```

Out:

```
Filename: /home/docs/.astropy/cache/download/url/ff6e0b93871033c68022ca026a956d87/contents
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     161   (891, 893)   int16
  1  er.mask       1 TableHDU        25   1600R x 4C   [F6.2, F6.2, F6.2, F6.2]
```

Generally the image information is located in the Primary HDU, also known as extension 0. Here, we use **astropy.io.fits.getdata()** to read the image data from this first extension using the keyword argument `ext=0`:

```
image_data = fits.getdata(image_file, ext=0)
```

The data is now stored as a 2D numpy array. Print the dimensions using the shape attribute:

```
print(image_data.shape)
```

Out:

```
(893, 891)
```

Display the image data:

```
plt.figure()
plt.imshow(image_data, cmap='gray')
plt.colorbar()
```

Good luck have fun

- Thanks for following, we hope you feel more confident with using Containers

- We also hope you feel more inspired about why they are useful and important

Thanks for listening