Python (Semester 2 2024)

Data Exploration

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

September 9, 2024

Motivation

We have seen how to write simple Python commands.

To put those commands together into Python scripts, we will see how to use **control flow statements**, and also how to access more complex algorithms from **libraries**.

With now knowing about control flow statements, the usage of libraries and data I/O, you are now well equipped to put all of this together to write **more complex code** for data exploration.

Data Exploration

Data exploration is a **critical step** in any data analysis project. It helps us understand the dataset, identify patterns, and gain insights.

Specifically, we will learn how to create summary statistics, check for missing values, and investigate for possible correlations among each column in the dataset. We will be using **libraries** like Pandas, NumPy, and Matplotlib to accomplish these tasks to better understand our data.

Data exploration is a key aspect of data analysis and model building. Without spending significant time on understanding the data and its patterns one cannot expect to build efficient predictive models. Data exploration takes a major chunk of time in a data science project comprising of data cleaning and preprocessing.

Data Exploration

Knowing your data isn't necessarily the most difficult thing in data science, but it is extremely important and it can be time-consuming. Therefore, often this part is overlooked.

In this lecture, we will see the various steps involved in data exploration through general concepts and Python code snippets.

Data Exploration

Knowing your data isn't necessarily the most difficult thing in data science, but it is extremely important and it can be time-consuming. Therefore, often this part is overlooked.

In this lecture, we will see the various steps involved in data exploration through general concepts and Python code snippets.

The **key steps** involved in data exploration are:

- Load data
- Identify variables
- Variable analysis
- Handling missing values
- Handling outliers
- Feature engineering

Loading Data

The first step in data exploration is to load data and identify variables:

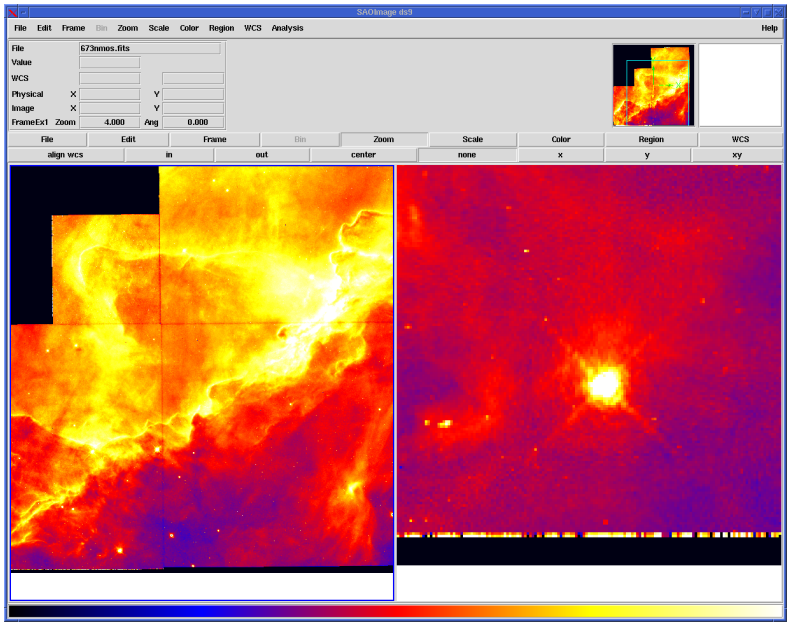
Data sources can vary: e.g. text files, databases, websites and remote services.

We have seen how to load tables in text format (`*.txt`, `*.csv`).
We will see now how to load files in another format that is commonly used in astronomy: **FITS files**.

Astronomical Fits Files

FITS (*Flexible Image Transport System*) is the standard astronomical data format endorsed by both NASA and the IAU. It is widely used in the astronomy community to store images and tables.
FITS can store scientific data sets consisting of multi- dimensional arrays (1-D spectra, 2-D tables, 2-D images or 3-D data cubes).

There exists a range of FITS file viewers; a list of such can be found here:
`https://fits.gsfc.nasa.gov/fits_viewer.html`



Astronomical Fits Files

In addition to FITS viewers, there exits software packages allowing for image viewing, data analysis, and format conversion.

For Python, such packages allow for seemlessly integrating FITS files into scientific code. It should be noted that FITS is a very general data format that is used for many different types of astronomical data sets, so these packages are not

We are here demonstrating the functionality of the `astropy.io.fits` package which provides access to FITS files.

Loading Astronomical Fits Files in Python

To open fits files, we use `astropy.io.fits.open()`:

```
# Example: open an existing FITS file
from astropy.io import fits
fits_image_filename = 'my_fitsfile.fits'
hdul = fits.open(fits_image_filename)
```

The `open()` function takes either the relative or absolute file path.
In addition, the function has several optional arguments which will be discussed later. The `open()` function returns an object called an HDUList which is a list-like collection of HDU objects. An HDU (Header Data Unit) is the highest level component of the FITS file structure, consisting of a header and (typically) a data array or table.
`hdul[0]` is the primary HDU, `hdul[1]` is the first extension HDU, etc. (if there are any extensions), and so on. It should be noted that astropy uses zero-based indexing when referring to HDUs and header cards, whereas the FITS standard uses one-based indexing.

Loading Astronomical Fits Files in Python

The HDUList has a useful method `HDUList.info()`, which summarizes the content of the opened FITS file:

```
hdul.info()
Filename: ...test0.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     138   ()
  1  SCI           1 ImageHDU        61   (40, 40)   int16
  2  SCI           2 ImageHDU        61   (40, 40)   int16
  3  SCI           3 ImageHDU        61   (40, 40)   int16
  4  SCI           4 ImageHDU        61   (40, 40)   int16
```

Loading Astronomical Fits Files in Python

After you are done with the opened file, close it with the `HDUList.close()` method:

```
hdul.close()
```

Loading Astronomical Fits Files in Python

After you are done with the opened file, close it with the `HDUList.close()` method:

```
hdul.close()
```

You can avoid closing the file manually by using `open()` as context manager, just like we saw it previously for text files:

```
with fits.open(fits_image_filename) as hdul:
```

After exiting the with scope the file will be closed automatically. That is (generally) the preferred way to open a file in Python, because it will close the file even if an exception happens.

Loading Astronomical Fits Files in Python

So far we have accessed FITS files locally available on a computer. In addition to that, we can access `remote and cloud-hosted files`.

The `open()` function supports a `use_fsspec` argument which allows file paths to be opened using `fsspec`. The `fsspec` package supports a range of remote and distributed storage backends such as Amazon and Google Cloud Storage. For example, you can access a Hubble Space Telescope image located in the Hubble's public Amazon S3 bucket as follows:

```
# Location of a large Hubble archive image in Amazon S3 (213 MB)
url = "s3://stpubdata/hst/public/j8pu/j8pu0y010/j8pu0y010_drc.fits"

# Extract a 10-by-20 pixel cutout image
with fits.open(url, use_fsspec=True, fsspec_kwargs={"anon": True})
as hdul:

    cutout = hdul[1].section[10:20, 30:50]
```

Loading Astronomical Fits Files in Python


```
# Location of a large Hubble archive image in Amazon S3 (213 MB)
url = "s3://stpubdata/hst/public/j8pu/j8pu0y010/j8pu0y010_drc.fits"

# Extract a 10-by-20 pixel cutout image
with fits.open(url, use_fsspec=True, fsspec_kwargs={"anon": True})
as hdul:

    cutout = hdul[1].section[10:20, 30:50]
```

Note that the example obtains a cutout image using the ImageHDU.section attribute rather than the traditional ImageHDU.data attribute. The use of `.section` ensures that only the necessary parts of the FITS image are transferred from the server, rather than downloading the entire data array. This trick can significantly speed up your code if you require small subsets of large FITS files located on slow (remote) storage systems.

Loading Astronomical Fits Files in Python

We have seen how to open FITS files. This is the more typical use case you will encounter.
Packages such as `astropy` also allow for creating FITS files.
The typical use case for creating FITS files is distributing and publishing astronomical catalogs.

A **complete documentation** of astropy's FITS functionality, including on how to **create FITS files**, can be found online:

`https://docs.astropy.org/en/stable/io/fits/`

Functional Programming

We have already used `functions`:

When we import a library like `numpy` and then call e.g. `np.sin()`, we are using a function someone else wrote and included within the `numpy` library.
We will now see how we can **write** these functions ourselves, thus creatin **user-defined functions**.

Functional Programming

There are two main reasons to use functions:

Code Readability: Functions break up code into individually testable, easily-debuggable pieces.

Code Reusability: Functions enable us to reuse code in a convenient way.

Functional Programming

There are two main reasons to use functions:

Code Readability: Functions break up code into individually testable, easily-debuggable pieces.

Code Reusability: Functions enable us to reuse code in a convenient way.

An **example** that illustrates this:

Imagine a **data extraction pipeline** for producing a 1D spectrum of galaxies from 2D CCD image data stored in multiple of FITS files.
One would want to write a function to read fits files into your code and perform the data reduction (such as sky subtractions), then a function to identify the spectral orders in your images and one to extract (collapse) the 2D data into a single dimension.

Writing Functions

In Python, the **function declaration** syntax is as follows:

```
def function_name(arguments):
    # function body

    return
```

Here, we have te following elements:

- `def` keyword used to declare a function
- `function_name` any name given to the function
- `arguments` any value passed to function
- `return` optional, returns value from a function

A Python function may or may not **return a value**.
Note: The return statement also denotes that the function has ended. Any code after the return statement is not executed.

Writing Functions

Let's now see an example.

```
def load_img(fname):
    with fits.open(fname) as hdulist:
    img = hdulist[0].data
    return img
```

This function to load a file assumes the following:

- the file we load is a FITS file
- the library was already imported with `import astropy.io.fits as fits`
- within the FITS file, the science image is located in the first extension.

Writing Functions

Let's now see an example.

```
def load_img(fname):
    with fits.open(fname) as hdulist:
    img = hdulist[0].data
    return img
```

This function to load a file assumes the following:

- the file we load is a FITS file
- the library was already imported with `import astropy.io.fits as fits`
- within the FITS file, the science image is located in the first extension.

This code isn't very flexible but might make sense given the above criteria hold.

Writing Functions


```
def load_img(fname):
    with fits.open(fname) as hdulist:
      img = hdulist[0].data
      return img
```

Using this example, we can go through the elements of a function declaration introduced above in more detail.

The function declaration starts with a `def` keyword, followed by a space and then the function name. Next we have a parenthesis set, containing the names of the arguments for using the function. The argument names are our choice as long as we are consistent about their use within the function. These names only matter within the function and are discarded after (more on this later on). Finally, we have a colon, and an indented block comprising the code we want to run when the function is called, followed by a return statement which specifies what the function returns to our global code when it's done.

Writing Functions

We can add more flexibility to the above function:

```
def load_img(fname,extension=0):
  with fits.open(fname) as hdulist:
    img = hdulist[extension].data
    return img
```

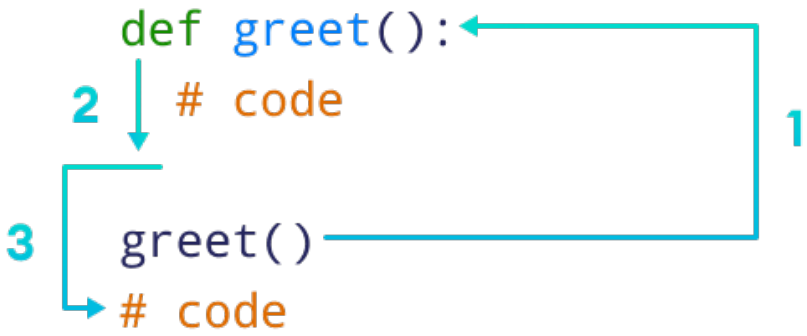
Now in addition to a filename `fname`, an optional extension `extension` can be given. The default value of `extension` is 0.

Now, to use this function, we need to call it.
Here's how we can call the `load_img()` function in Python:

```
load_img('imagefile.fits',1)
load_img('survey.fits',0)
```

Writing Functions

When a function is called, the control of the program goes to the function definition. All code inside the function is executed. The control of the program jumps to the next statement after the function call.



An important aspect of functions is that variables defined and used within a function are what is known as **local in scope**: those variables are created when the function is called and destroyed once the output is returned, so those values are not retrievable outside the function.

In contrast, **global variables** (defined outside of functions) are accessible from within the function, regardless of them being an argument or not. However, relying on this is considered **bad programming style**. Instead, a function should only depend on the variables listed as arguments, so the function could be moved to any other code without breaking it.

Variable Scope in Functions

As an example, we examine the following block of code:

```
constant = 5

def load_img(fname,extension=0):
     with fits.open(fname) as hdulist:
          img = hdulist[extension].data + constant
          return img
```

What would happen if we ran the above block?

Variable Scope in Functions

As an example, we examine the following block of code:

```
constant = 5

def load_img(fname,extension=0):
     with fits.open(fname) as hdulist:
          img = hdulist[extension].data + constant
          return img
```

What would happen if we ran the above block?

The function would have no problem adding the value of `constant` to our image, since it is defined globally in our code (outside the function). But if I copied and pasted this function into a different script, which didn't have a `constant` variable defined, I'd get an error. Even scarier, if I did have one defined but for a completely different reason in a different part of the code, I'd never know what happens.

Documentation

One important way to ensure we are doing all this correctly, as well as making things easier on ourselves later and on anyone else who may use our function is **good documentation**.

This is slightly different than commenting our code to describe what's going on (though we should do this).

Documentation is a **built-in feature** of the way Python does functions. The way to set documentation for a function is to place it inside triple quotes right at the top of our function definition, as follows:

Documentation


```
def load_img(fname,extension=0):
'''A function to quickly extract the data extension of a fits file
INPUTs:
fname (string): path/file name of the fits file to be loaded
extension (int) [default: 0]: extension to index
RETURNS:
img (array_like): data attribute queried'''

with fits.open(fname) as hdulist:
     img = hdulist[extension].data
     return img
```

Once having set documentation this way, you can run the command

```
help(load_img)
```

from Python/iPython to show the function's documentation.

Function Arguments

Setting optional arguments, args, and kwargs Optional Arguments
Functions can take **optional arguments**. They must come after the non-default variables and they must have a default value.
We can set arguments to be optional by setting the arguments of the `def` call equal to some default values. An example:

```
def somefunction(var1,var2,var3=1,var4='cat'):
     output = str(var1+var2+var3) + var4
     return output
```

Function Arguments

So far the number of arguments, despite some might be optional, was fixed: We specified a numer of inputs. We can use the `*args` and `**kwargs` (keyword arguments) commands to create variable numbers of arguments of a function. An example:

```
def test_function(farg,*args):
     print 'formal argument: ', farg
     print ' length of args: ', len(args)
     for arg in args:
          print 'new arg:', arg
```

Function Arguments

So far the number of arguments, despite some might be optional, was fixed: We specified a numer of inputs. We can use the `*args` and `**kwargs` (keyword arguments) commands to create variable numbers of arguments of a function. An example:

```
def test_function(farg,*args):
     print 'formal argument: ', farg
     print ' length of args: ', len(args)
     for arg in args:
          print 'new arg:', arg
```

What happens here is the following:
The formal argument `farg` is read in like a normal argument. We could have any number of these. But we've specified the last argument as `*args`, which tells Python to expect some uknown number of inputs after this. Within the function, one then can iterate through the list of extra inputs to do things with them individually.

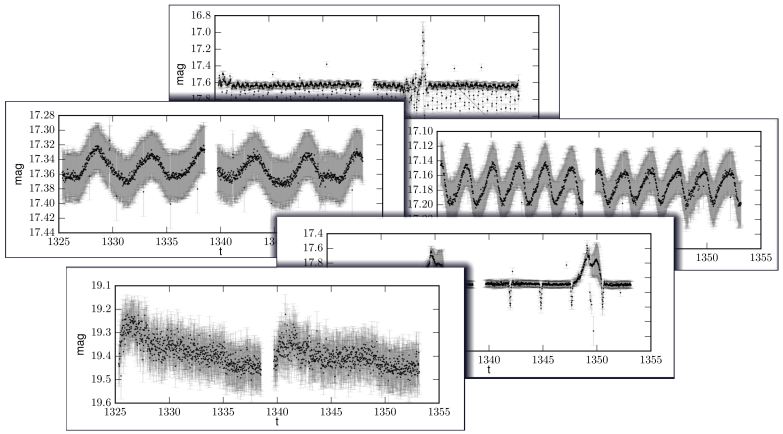
Function Arguments

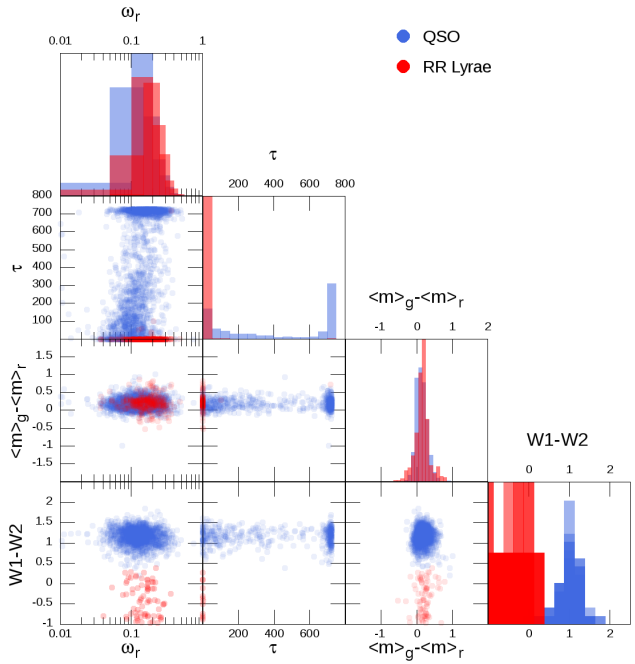
`**kwargs` (keyword arguments) are very similar to `*args` in the way that they let you pass a variable number of extra variables to the function. The difference is, when you feed those extra arguments into the function, you individually give each a new keyword by setting it equal to it in the function call. Then, instead of putting all the extra arguments into a list, they are put into a dictionary where each value is linked to the key and can be accessed via dictionary style slicing. An example:

```
def kwarg_examplefunction(farg, **kwargs):
     print 'formal argument: ', farg
     for key in kwargs:
          print 'argument: ', kwargs[key]
```

Plotting

We have already briefly seen how to use the `matplotlib` library. We will go into more detail here in the context of data exploration.
Generating plots of various kind is an essential step in data exploration: We can plot the raw data (just after reading it from a file) to familiarize ourselves with typical outliers, we can make plots to investigate possible cross-correlations, and more.





Plotting

Let's start with the basics. Say we have an **independent variable** (like time), and a **measured (dependent) variable** (like position). This type of data could easily be read in from a 2-column text file and then plotted against each other, plotting e.g. a **light curve**.

```
# example: Plotting x vs. y
import matplotlib.pyplot as plt
import numpy as np

file_name = '/research/timeseries.txt'
data = np.loadtxt(file_name)
times = data[:,0]
positions = data[:,1]

# Now let's plot the data
plt.plot(times,positions)
plt.show()
```

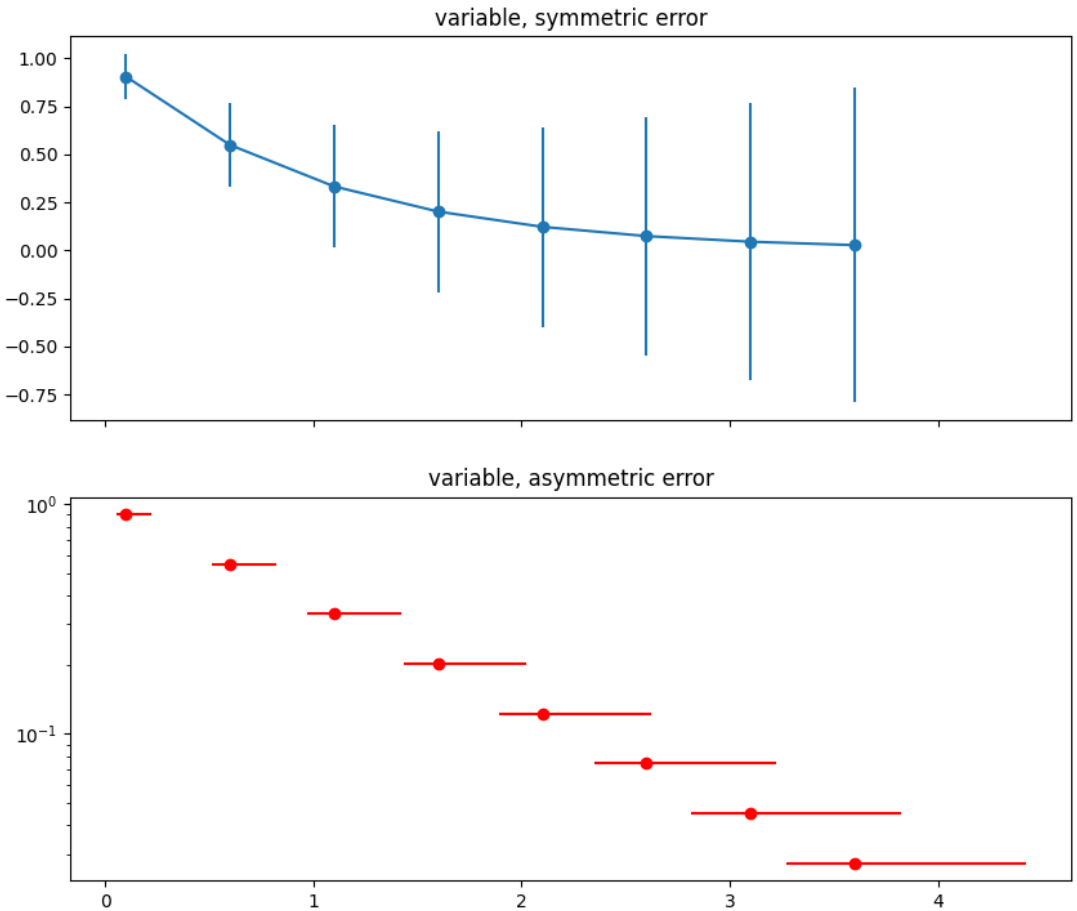
Plotting

Note that in Python when plotting, the first argument is an array of x values and the second value is an array of y values, and the number of elements in the two arrays must match.
In addition, we can specify a third column as error column, e.g. a magnitude error.
When executing the code, you'll notice that the default way Python plots is by plotting the positions against the times and connecting them with blue lines. We however want to plot the individual data points.
In the following we will see an example that plots measurements along with their errors, and plots everything in individual data points:

Plotting


```
# example: Plotting x vs. y with error bars
import matplotlib.pyplot as plt
import numpy as np

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)
# example error bar values that vary with x-position
error = 0.1 + 0.2 * x

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True)
ax0.errorbar(x, y, yerr=error, fmt='-o')
ax0.set_title('variable, symmetric error')

# error bar values w/ different -/+ errors
lower_error = 0.4 * error
upper_error = error
asymmetric_error = [lower_error, upper_error]

ax1.errorbar(x, y, xerr=asymmetric_error, fmt='ro')
ax1.set_title('variable, asymmetric error')
ax1.set_yscale('log')
plt.show()
```

Plotting



Plotting

When using `plt.plot` and similar such as `errorbar` or `scatter`, there are optional settings you can specify, mainly color and line/ point style.
If you choose a discrete symbol (like `'o'` for circles or `'+'` for plusses), then Python won't connect them automatically.
You can use a matplotlib shortcut to simultaniously choose a color and linestyle as follows:

```
plt.plot(times, positions, 'ro')
```

This would plot the discrete points as red circles.
You can also specify the size of the symbol by including the argument `ms`, e.g. `ms=10`

More on this can be found in the `matplotlib` documentation.

Plotting

So far we have displayed plots by using simply a

```
plt.show()
```

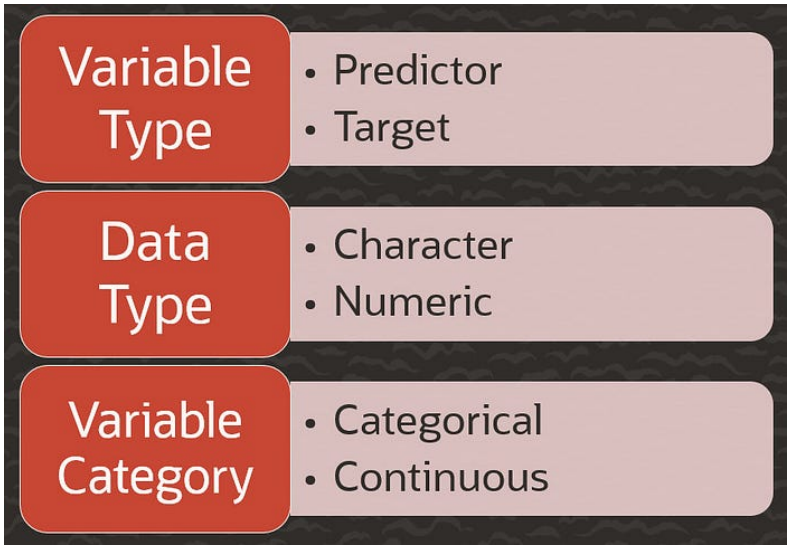
at the end. In many cases it is better to save plots, and doing so as a vectorized `*.pdf` file.
For this purpose, the plotting code needs to be modified as follows:

```
import matplotlib
matplotlib.use('PDF')
import matplotlib.pyplot as plt

# here comes your code to create the plot

plt.savefig('plot.pdf)
```

Properties of Data Sets

Identifying the **predictor and target variable(s)** is one of the key steps in model building.

Target is the dependent variable and predictor is the independent variable based on which the prediction is made. Categorical or discrete variables are those that cannot be mathematically manipulated. It is made up of fixed values such as 0 and 1. On the other hand, continuous variables can be interpreted using mathematical functions like the average or sum.



Properties of Data Sets

To get a first overview of some data we loaded, we are here using the pandas library.
It provides functionality for loading files, as well as for data manipulation such as working with missing data, easily selecting columns and so on.

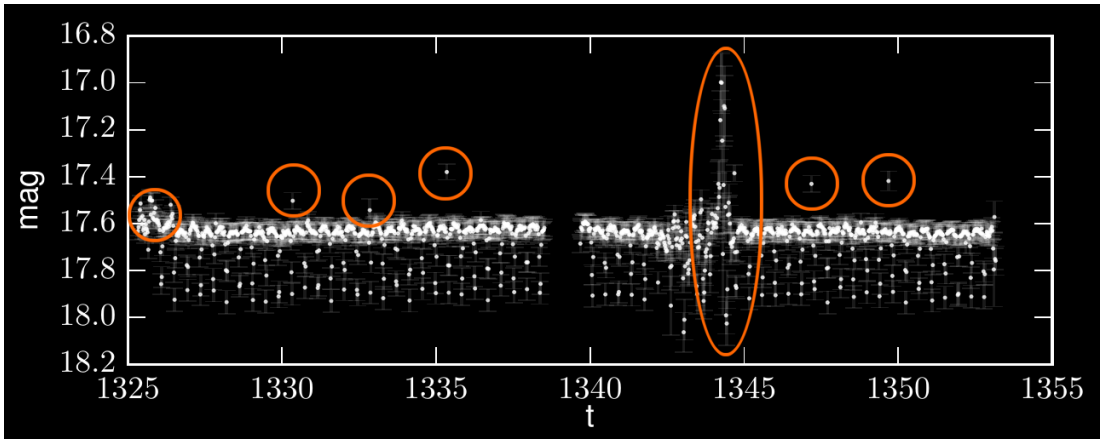
```
# example: Loading a file and explore its content

#Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns#Load the data

#get an overview of the data
lightcurve=pd.read_csv("lightcurve.csv")
lightcurve.head()
lightcurve.tail()
lightcurve.sample(10)

#identify variable type
lightcurve.dtypes
lightcurve.info()
lightcurve.describe()
```

Outliers and Missing Data

Data sets, especially in astronomy, can contain **outliers and missing data**.

Missing data is often caused by observing conditions, when observations didn't take place or didn't fulfil the quality criteria for certain analysis.

Outliers are also often the result of bad observing conditions, or issues with a software like a photometric pipeline. Other outliers might be caused by the way an instrument works that can be tolerated in certain cases, but might not be satisfying the criteria we set on our data for other kinds of analysis. See e.g. the following stellar light curve from the TESS survey:



Outliers and Missing Data

Important questions when thinking about missing data:

- How prevalent is the missing data?
- Is missing data random or does it have a pattern?

Outliers and Missing Data

Important questions when thinking about missing data:

- How prevalent is the missing data?
- Is missing data random or does it have a pattern?

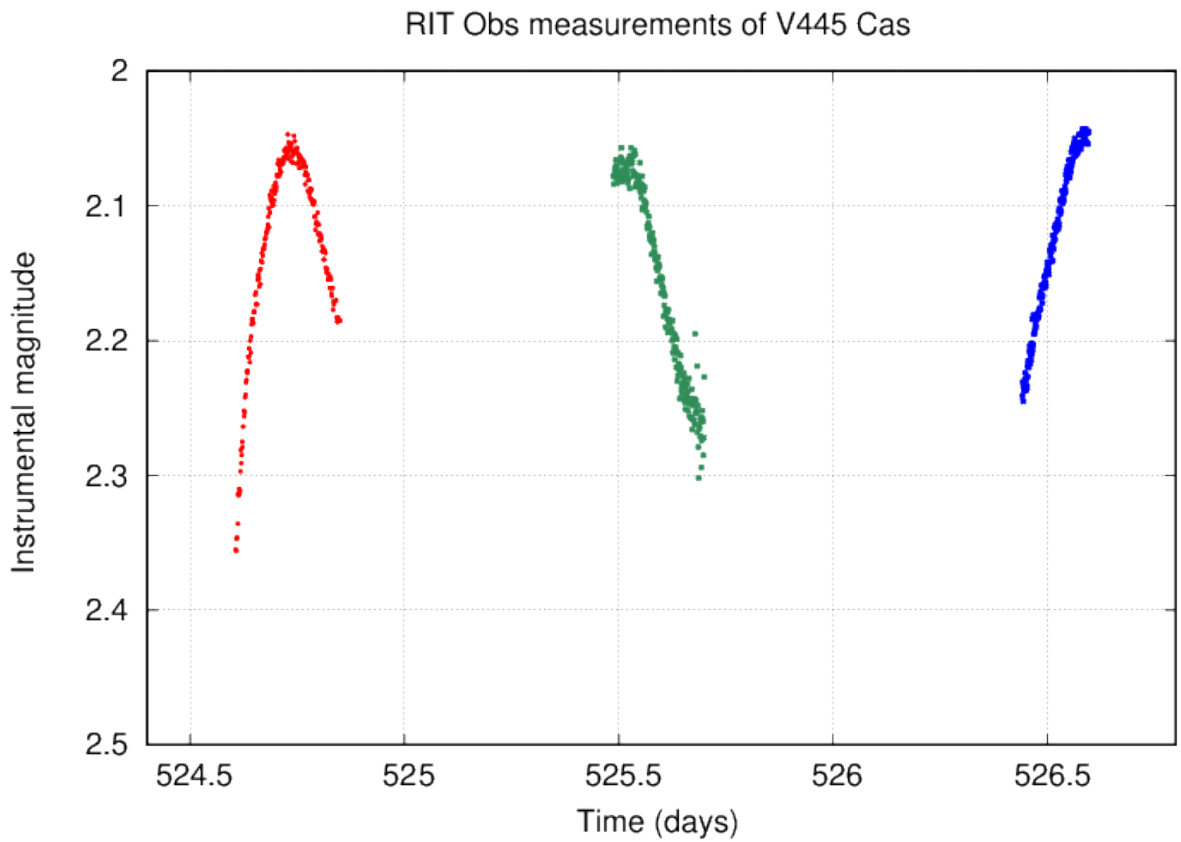
Answering these questions is important as missing data can imply

- errors in subsequent data analysis
- a (drastically) reduced resulting sample size

Both can lead to **wrong results** as well as can prevent us from proceeding with the analysis. Moreover, from a **substantive perspective**, we need to ensure that the missing data process is not biased and hidding an inconvenient truth, also known as **"window effect"**.

Outliers and Missing Data

The following plot shows an observational window effect affecting a variable star light curve:



Distributions

Once we have handled missing data and outliers, we need to get a better idea on the distribution of our data.

According to Hair et al. (2013), four assumptions should be tested:

1. Normality Do the data follow a normal distribution? Knowing that is important because several statistic tests rely on this (e.g. t-statistics). Remember that univariate normality doesn't ensure multivariate normality (which is what we would like to have).

2. Homoscedasticity This term refers to the 'assumption that dependent variable(s) exhibit equal levels of variance across the range of predictor variable(s)'. Homoscedasticity is desirable because we want the error term to be the same across all values of the independent variables.

Distributions

Once we have handled missing data and outliers, we need to get a better idea on the distribution of our data.

According to Hair et al. (2013), four assumptions should be tested:

3. Linearity The most common way to assess linearity is to examine scatter plots and search for linear patterns. If patterns are not linear, it would be worthwhile to explore data transformations (e.g. the logarithm).

4. Absence of correlated errors Correlated errors occur when one error is correlated to another. This happens often in time series, where some patterns are time related. During further analysis, we might want to explain that effect.

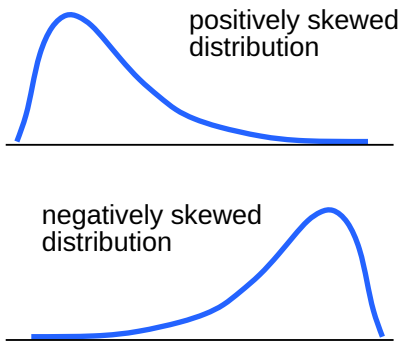
Properties of Data Sets

We can test for **normality** by plotting a histogram and fitting a Gaussian. When analyzing the histogram, we pay attention to two properties:

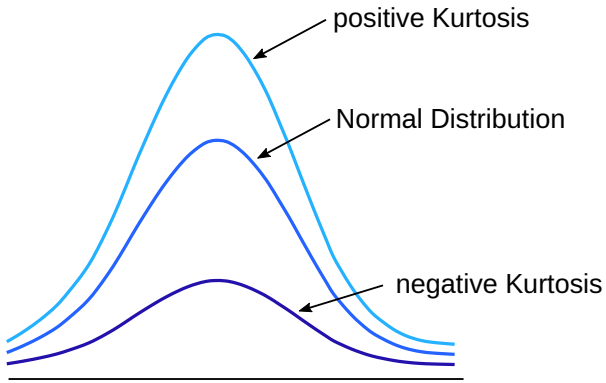
Skewness is a measure of symmetry, or more precisely, the lack of symmetry.

Kurtosis is a measure of whether the data are heavy-tailed or light-tailed relative to a normal distribution. That is, data sets with high kurtosis tend to have heavy tails, or outliers. Data sets with low kurtosis tend to have light tails, or lack of outliers.

Skewness



Kurtosis



Properties of Data Sets

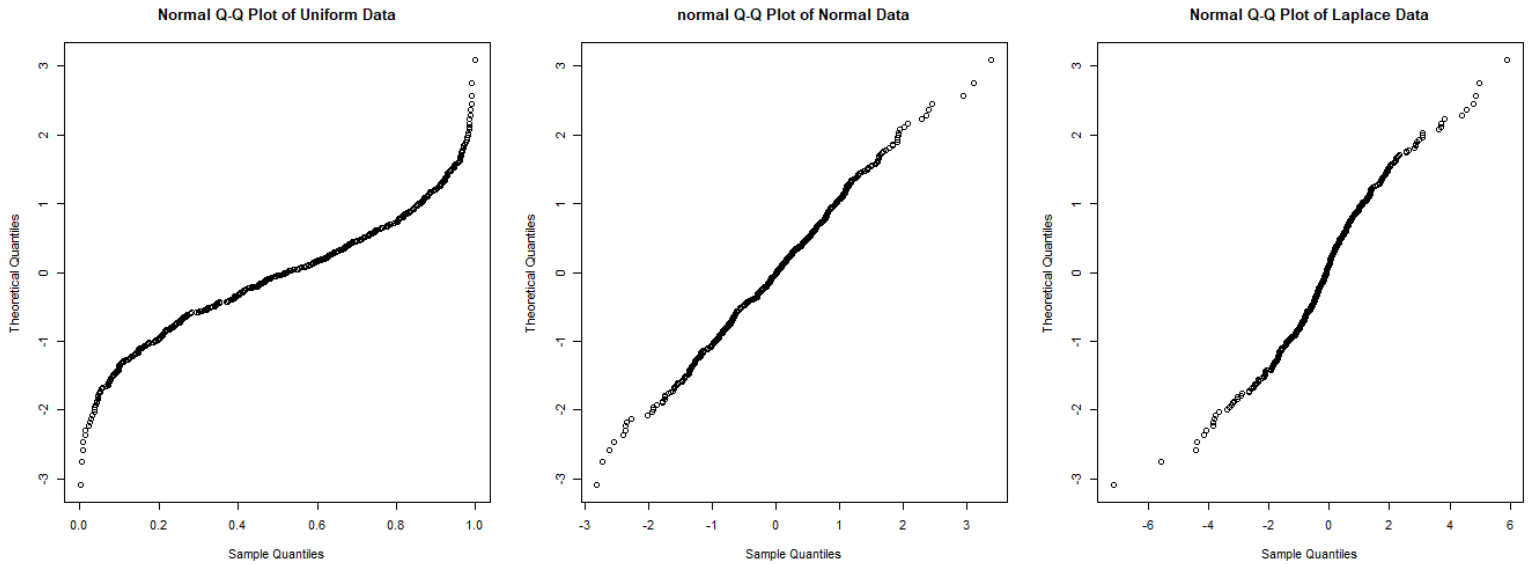
Within `scipy.stats`, functionality to calculate the skewness and kurtosis of a data set can be found.
Also, the calculation is possile within the `seaborn` library:

```
f, axes = plt.subplots(2, 2, figsize=(7, 7), sharex=True)
sns.distplot( data[cols_viz[0]] , color="skyblue", ax=axes[0, 0])
print("Skewness: %f" % data[cols_viz[0]].skew())
print("Kurtosis: %f" % data[cols_viz[0]].kurt())

sns.distplot( data[cols_viz[1]] , color="olive", ax=axes[0, 1])
print("Skewness: %f" % data[cols_viz[1]].skew())
print("Kurtosis: %f" % data[cols_viz[1]].kurt())
sns.distplot( data[cols_viz[2]] , color="gold", ax=axes[1, 0])
sns.distplot( data[cols_viz[3]] , color="teal", ax=axes[1, 1])
plt.show()
```

Properties of Data Sets

Another way to investigate whether a data set follows a normal distribution is the **normal probability plot**.

In this plot, data is plotted against the theoretical normal distribution plot. In case of normality, the data distribution should closely follow the diagonal that represents the normal distribution.



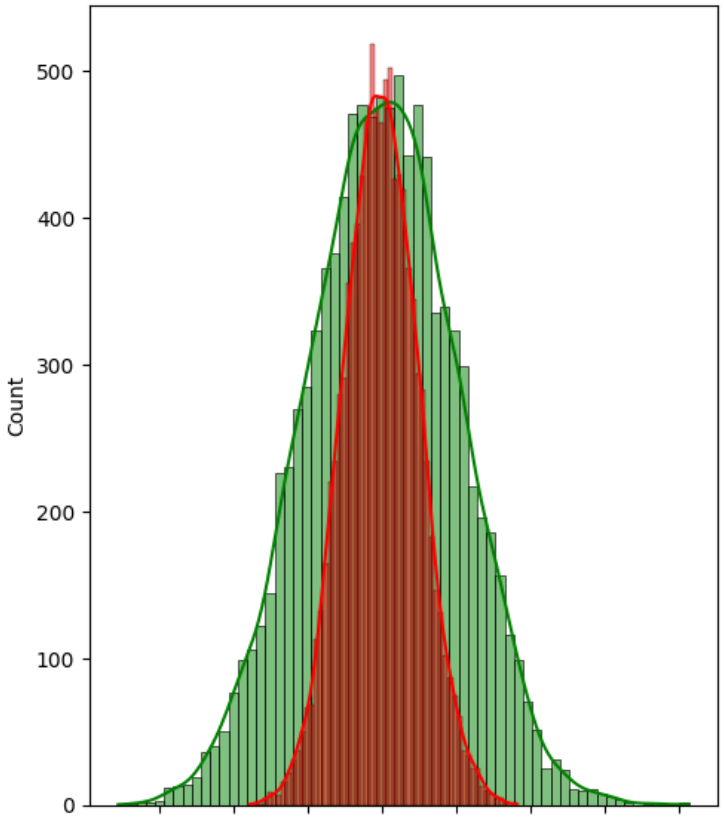
Properties of Data Sets

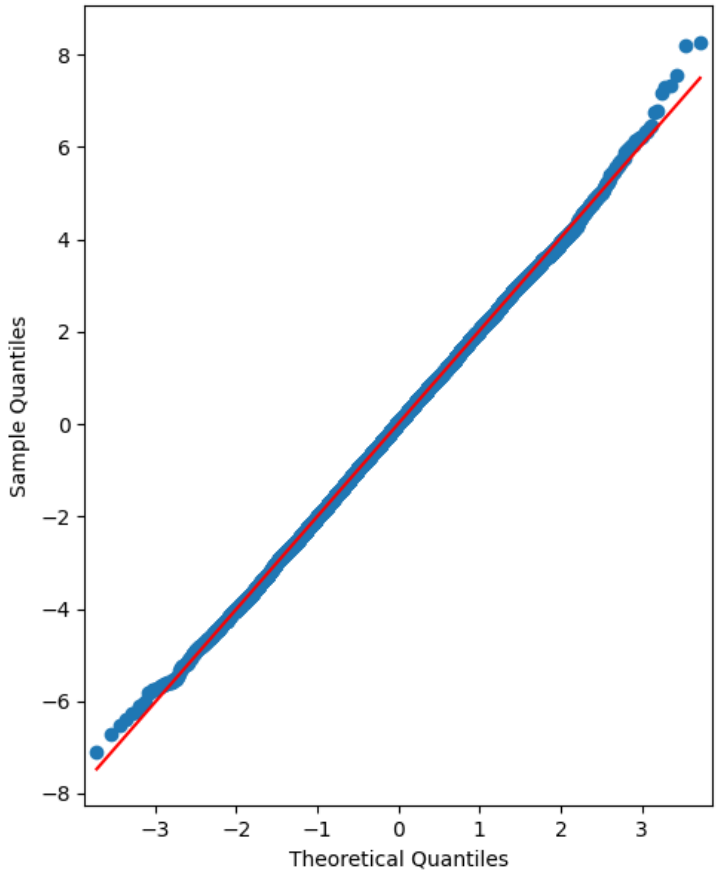

```
# example: normal probability plot

# imports
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as sc
import statsmodels.graphics.gofplots as sm

# define distributions
sample_size = 10000
standard_norm = np.random.normal(size=sample_size)
heavy_tailed_norm = np.random.normal(loc=0, scale=2, size=sample_size)
skewed_norm = sc.skewnorm.rvs(a=5, size=sample_size)
skew_left_norm = sc.skewnorm.rvs(a=-5, size=sample_size)

# plots for standard distribution
fig, ax = plt.subplots(1, 2, figsize=(12, 7))
sns.histplot(standard_norm,kde=True, color ='blue',ax=ax[0])
sm.ProbPlot(standard_norm).qqplot(line='s', ax=ax[1])

# plot for right-tailed distribution
fig, ax = plt.subplots(1, 2, figsize=(12, 7))
sm.ProbPlot(skewed_norm).qqplot(line='s', ax=ax[1]);
sns.histplot(skewed_norm,kde=True, color ='blue',ax=ax[0])
```

Properties of Data Sets


```
#  (continued)

# plot for heavy tailed distribution
fig, ax = plt.subplots(1, 2, figsize=(12, 7))
sm.ProbPlot(heavy_tailed_norm).qqplot(line='s',color ='green', ax=ax[1]);
sns.histplot(heavy_tailed_norm,kde=True, color ='green',ax=ax[0])
sns.histplot(standard_norm,kde=True, color ='red',ax=ax[0])

plt.show()
```

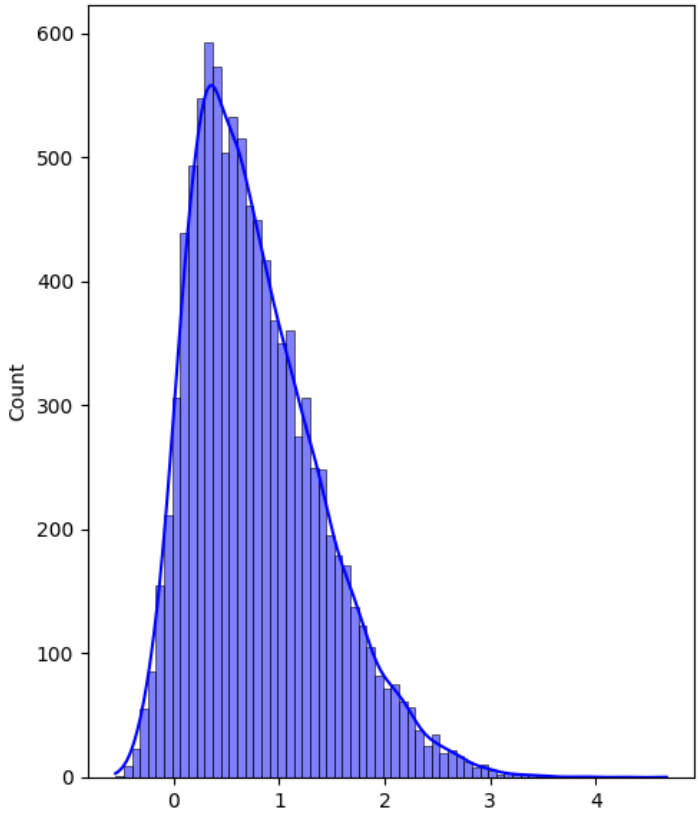
Properties of Data Sets

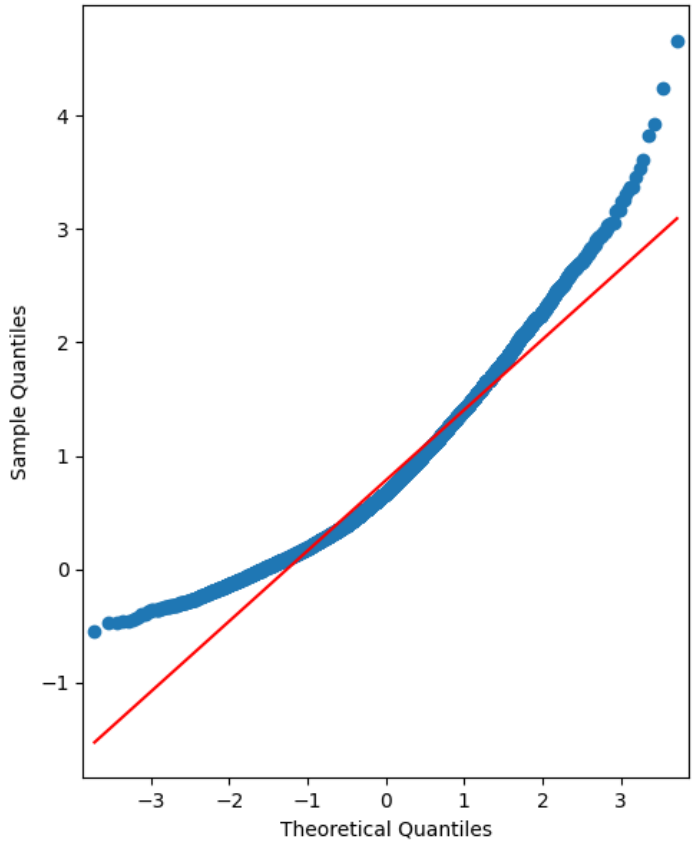




Properties of Data Sets

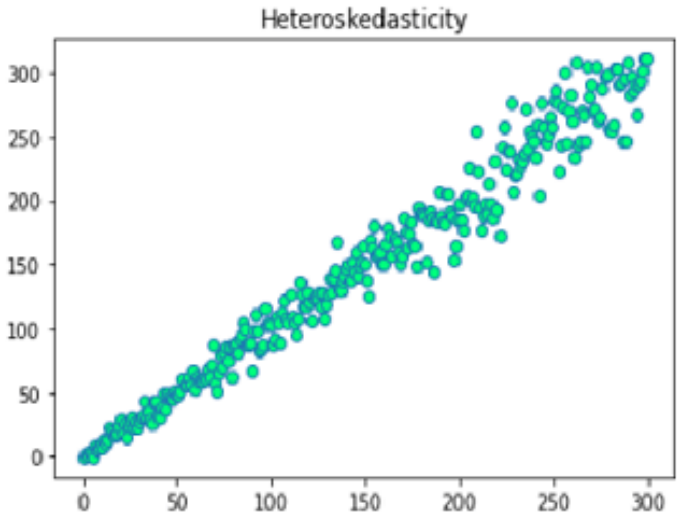
How it also could look like:

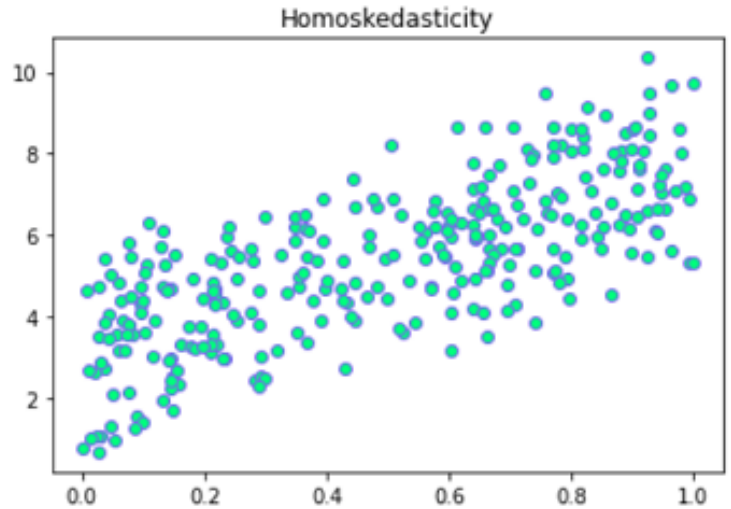




Properties of Data Sets

The best approach to test homoscedasticity for two metric variables is graphically. Departures from an equal dispersion are shown by such shapes as cones (small dispersion at one side of the graph, large dispersion at the opposite side) or diamonds (a large number of points at the center of the distribution).





An Outlook: Data Visualization

Here we have seen how to explore our data, including how to generate plots that qucikly visualize data properties.

In the next lecture, we will see how we can **visualize data to produce high-quality plots and graphs** for publications and presentations.