Python (Semester 2 2024)

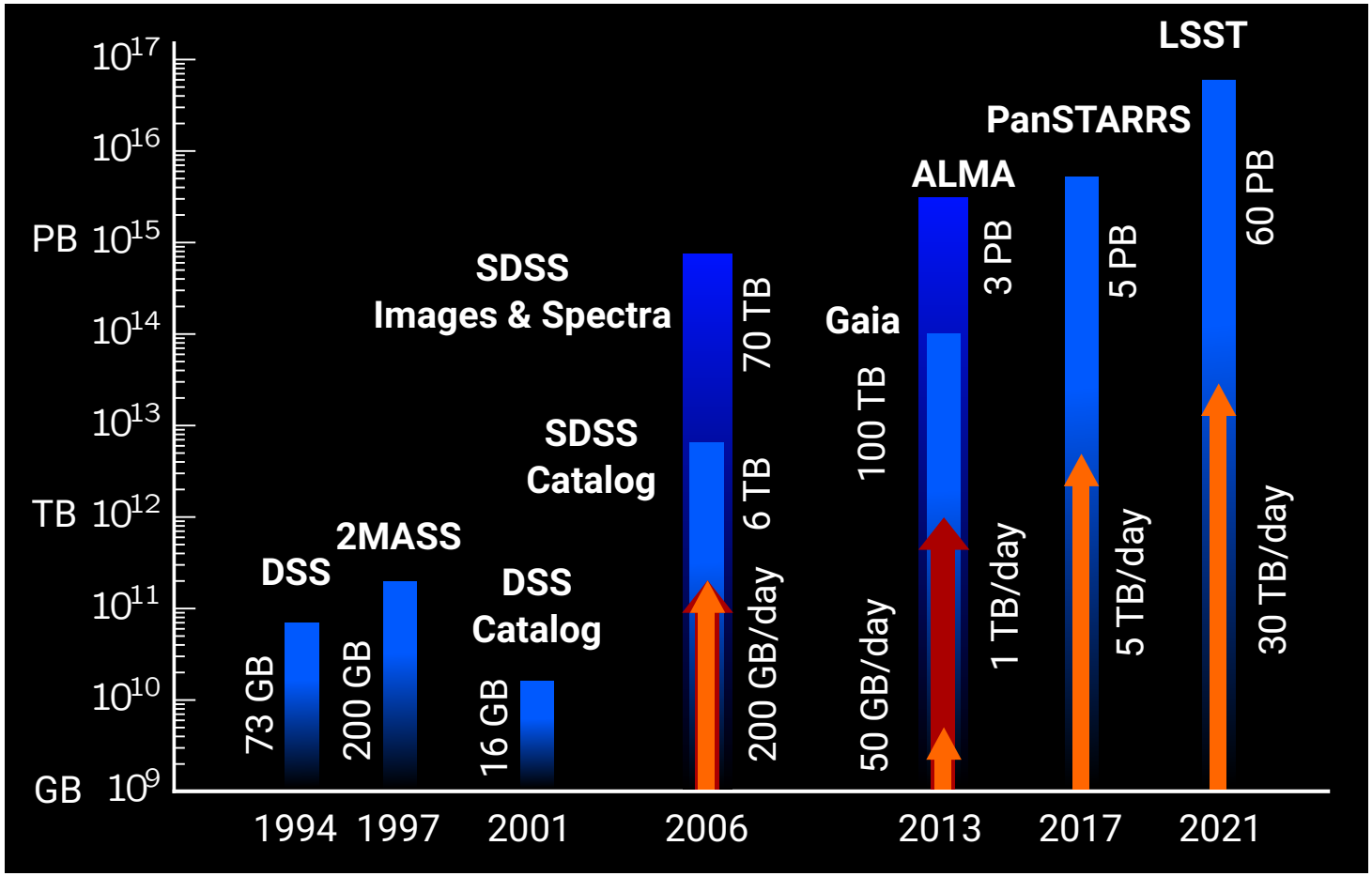
Statistical Packages

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

October 7, 2024

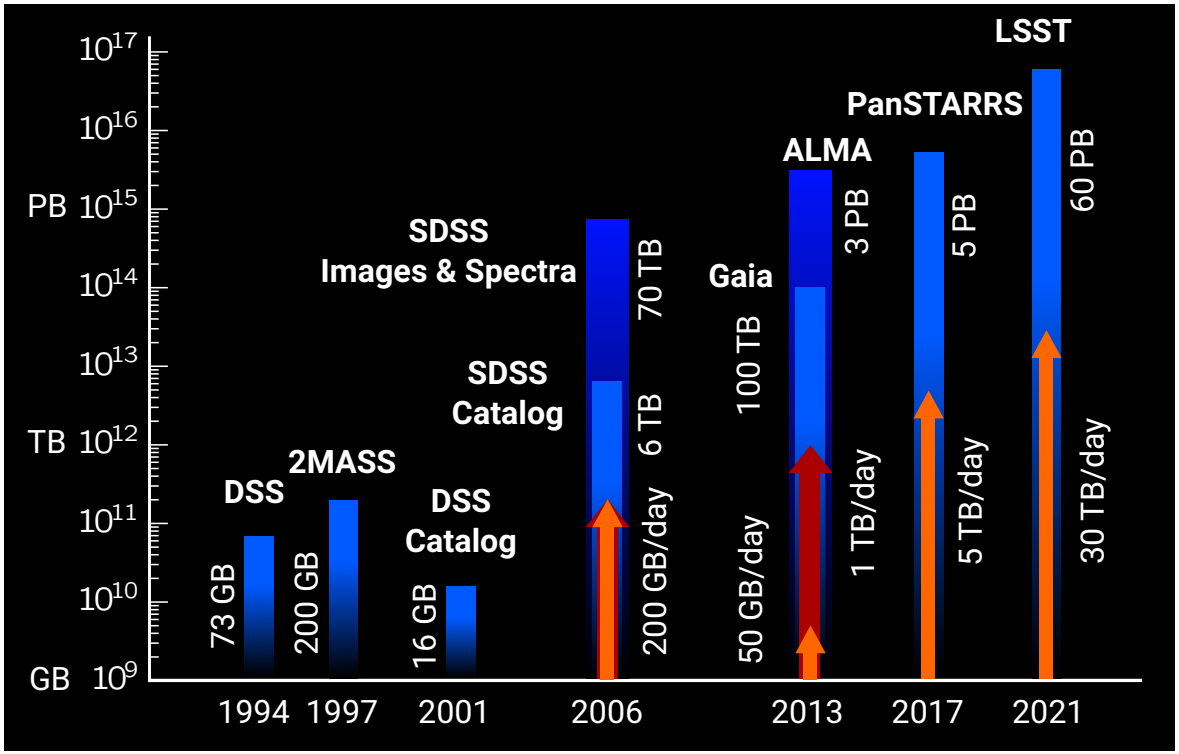
Motivation

Modern all-sky surveys produce an increasing data volume:



Motivation

Modern all-sky surveys produce an increasing data volume:



statistical analysis becomes increasingly important as **tools for discoveries**

General statistics packages

There are a lot of Python packages providing **general statistical algorithms**:

NumPy is a fundamental package for working with numerical arrays in Python. It includes a powerful N-dimensional array object, as well as a set of tools for working with these arrays. NumPy can be used for a variety of basic statistical analyses, such as mean, median, standard deviation, variance, correlation, and covariance. These functions can also be applied along different axes of multidimensional arrays to get statistics for rows, columns, or slices.

`https://numpy.org/`

General statistics packages

SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. SciPy contains modules for statistics, optimization, linear algebra, integration, interpolation, special functions, Fourier transforms (FT/FFT), signal and image processing, and other tasks common in science and engineering.

`Scipy.stats` provides methods to work with following statistical concepts:

- Random variables
- Probability distributions
- One sample and two sample analysis including comparisons
- Kernel density estimation
- Quasi-Monte Carlo

`https://scipy.org/`

General statistics packages

Statsmodels is a Python package that provides a set of tools for statistical analysis and econometric modeling. It includes tools for performing various statistical tests, as well as linear regression and time series analysis. Statsmodels can be used for both exploratory data analysis and formal hypothesis testing.

`Statsmodels` provides methods to work with following:

- Regression and linear models
- Time series analysis
- Statistical tools such as probability distributions, contingency table, etc.

`https://statsmodels.org/`

Descriptive Statistics

Descriptive statistics is about describing and summarizing data. It uses two main approaches:

The **quantitative approach** describes and summarizes data numerically.

The **visual approach** illustrates data with charts, plots, histograms, and other graphs.

Descriptive Statistics

Descriptive statistics is about describing and summarizing data. It uses two main approaches:

The **quantitative approach** describes and summarizes data numerically.

The **visual approach** illustrates data with charts, plots, histograms, and other graphs.

We have already seen the visual approach in *Lecture 4, Data Visualization*. We will focus here thus on the quantitative approach.

Descriptive statistics can be applied to one or many datasets or variables. When you describe and summarize a single variable, you are performing **univariate analysis**. When you search for statistical relationships among a pair of variables, you're doing **bivariate analysis**. Similarly, this can be generalized to **multivariate analysis**.

Descriptive Statistics

In the following, you will see how we can calculate a variety of **measures in descriptive statistics** in Python:

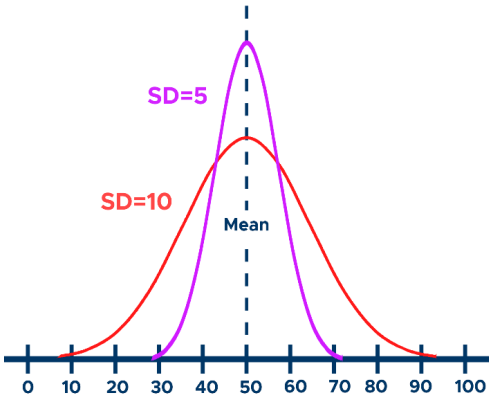
Descriptive Statistics

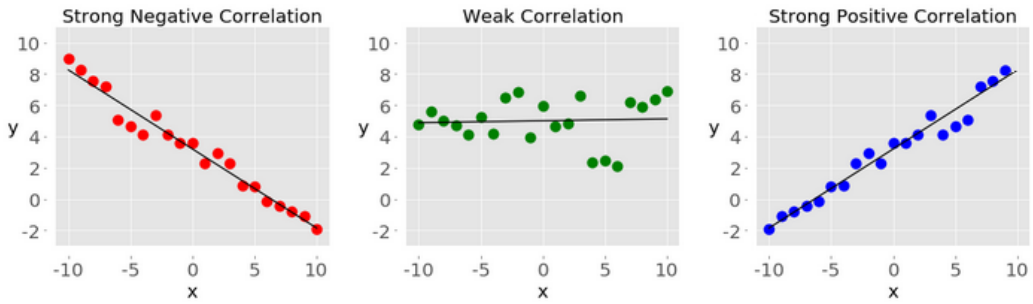
In the following, you will see how we can calculate a variety of **measures in descriptive statistics** in Python:

Central tendency tells you about the centers of the data. Useful measures include the mean, median, and mode.

Variability tells you about the spread of the data. Useful measures include variance and standard deviation.

Correlation (or joint variability) tells you about the relation between a pair of variables in a dataset. Useful measures include covariance and the correlation coefficient.

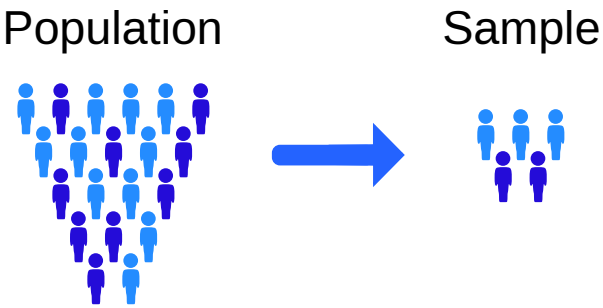




Population and Samples

In statistics, the **population** is a set of all elements or items you are interested in. Populations are often vast, which makes them inappropriate for collecting and analyzing data. Thus statisticians often try to make conclusions about a population based on a **representative subset of that population**, the **sample**.

Ideally, the sample should preserve the essential statistical features of the population to a satisfactory extent.



quantity = N
mean = μ
variance = σ^2
standard deviation = σ

quantity = n
mean = $\bar{x}$
variance = s^2
standard deviation = s

Outliers

An **outlier** is a data point that differs significantly from the majority of data from a sample or population. There are many possible causes of outliers:

- Natural variation in data
- Change in the behavior of the observed system
- Errors in data collection

Outliers

An **outlier** is a data point that differs significantly from the majority of data from a sample or population. There are many possible causes of outliers:

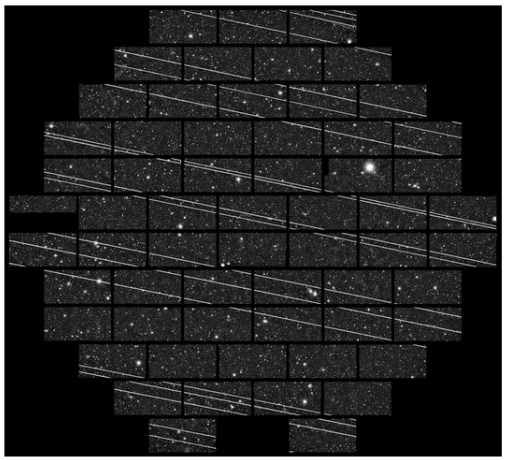
- Natural variation in data
- Change in the behavior of the observed system
- Errors in data collection

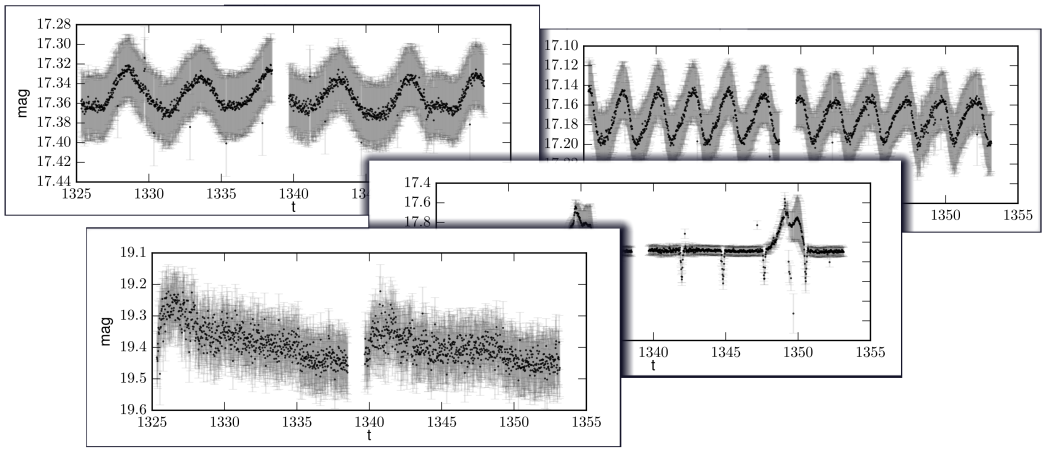
Data collection errors are a particularly prominent cause of outliers: Instrumental limitations, limitations of data processing procedures can mean that the correct data is simply not obtainable. Other errors can be caused by miscalculations, data contamination, human error, and more. Outliers, however, can also deliver valuable information about (astronomical) objects behaving unusual.

There isn't a precise mathematical definition of outliers. You have to rely on experience, **knowledge about the subject of interest**, and common sense to determine if a data point is an outlier and how to handle it.

Outliers

some examples of outliers in astronomical data:





left: An image from the Cerro Tololo Inter-American Observatory shows streaks left by Starlink satellites. (Image credit: CTIO/AURA/DELVE)

right: Light curves from TESS are influenced by systematics due to the satellite heating up.

Statistical Analysis Use Cases

In the following, we will illustrate the **capabilities of various statistical packages** on specific use cases.

Statistical Analysis Use Cases

In the following, we will illustrate the **capabilities of various statistical packages** on specific use cases.

We will work especially with `scipy.stats`.

In addition to providing basic functionality to calculate such as means, averages, standard deviations, `scipy.stats` contains many functions for performing statistical tests and calculations.
For example, you can use `scipy.stats.ttest_ind()` to perform a t-test for comparing the means of two independent samples, `scipy.stats.chisquare()` to perform a chi-square test for testing the independence of two categorical variables, or `scipy.stats.linregress()` to perform a linear regression analysis.
You can also use `scipy.stats.norm()` to create a normal distribution object, `scipy.stats.binom()` to create a binomial distribution object, and use their methods to calculate probabilities, percentiles, or random samples.

Statistical Analysis Use Cases

We start by importing all the packages we need:

```
import math
import statistics
import numpy as np
import scipy.stats
import pandas as pd
```

Statistical Analysis Use Cases

We start by importing all the packages we need:

```
import math
import statistics
import numpy as np
import scipy.stats
import pandas as pd
```

Let's create Python lists containing some arbitrary numeric data:

```
x = [8.0, 1, 2.5, 4, 28.0]
x_with_nan = [8.0, 1, 2.5, math.nan, 4, 28.0]
print(x)
print(x_with_nan)
```

We get:

Statistical Analysis Use Cases

We start by importing all the packages we need:

```
import math
import statistics
import numpy as np
import scipy.stats
import pandas as pd
```

Let's create Python lists containing some arbitrary numeric data:

```
x = [8.0, 1, 2.5, 4, 28.0]
x_with_nan = [8.0, 1, 2.5, math.nan, 4, 28.0]
print(x)
print(x_with_nan)
```

We get:

```
[8.0, 1, 2.5, 4, 28.0]
[8.0, 1, 2.5, nan, 4, 28.0]
```

Statistical Analysis Use Cases

In data science, missing values are common, and ared often replaced with NaN. It is important to understand the behavior of the Python statistics routines when they come across a not-a-number value (NaN).
In Python, you can use any of the following:

```
float('nan')
math.nan
np.nan
```

Statistical Analysis Use Cases

In data science, missing values are common, and ared often replaced with NaN. It is important to understand the behavior of the Python statistics routines when they come across a not-a-number value (NaN).
In Python, you can use any of the following:

```
float('nan')
math.nan
np.nan
```

You can check for NaN:

```
print(math.isnan(np.nan), np.isnan(math.nan))
```

Statistical Analysis Use Cases

In data science, missing values are common, and ared often replaced with NaN. It is important to understand the behavior of the Python statistics routines when they come across a not-a-number value (NaN).
In Python, you can use any of the following:

```
float('nan')
math.nan
np.nan
```

You can check for NaN:

```
print(math.isnan(np.nan), np.isnan(math.nan))
```

```
(True, True)
```

Statistical Analysis Use Cases

In data science, missing values are common, and ared often replaced with NaN. It is important to understand the behavior of the Python statistics routines when they come across a not-a-number value (NaN).
In Python, you can use any of the following:

```
float('nan')
math.nan
np.nan
```

You can check for NaN:

```
print(math.isnan(np.nan), np.isnan(math.nan))
```

```
(True, True)
```

Please keep in mind that comparing two `nan` values for equality returns `False`. In other words, `math.nan == math.nan` is `False`.

Statistical Analysis Use Cases

Now, create `np.ndarray` and `pd.Series` objects that correspond to `x` and `x_with_nan`:

```
y, y_with_nan = np.array(x), np.array(x_with_nan)
z, z_with_nan = pd.Series(x), pd.Series(x_with_nan)
print(y)
print(y_with_nan)
print(z)
print(z_with_nan)
```

We get:

Statistical Analysis Use Cases


```
array([ 8. ,  1. ,  2.5, 4. , 28. ])

array([ 8. ,  1. ,  2.5,  nan,  4. , 28. ])

0     8.0
1     1.0
2     2.5
3     4.0
4    28.0
dtype: float64

0     8.0
1     1.0
2     2.5
3     NaN
4     4.0
5    28.0
dtype: float64
```

We now have two NumPy arrays (`y` and `y_with_nan`) and two Pandas series (`z` and `z_with_nan`). All of these are 1D sequences of values.

Calculating Descriptive Statistics

Based on these data, we now continue with **calculating descriptive statistics**.

Calculating Descriptive Statistics

Based on these data, we now continue with **calculating descriptive statistics**.

The **Measures of Central Tendency** show the central or middle values of datasets. There are several definitions of what's considered to be the center of a dataset:

Calculating Descriptive Statistics

Based on these data, we now continue with **calculating descriptive statistics**.

The **Measures of Central Tendency** show the central or middle values of datasets. There are several definitions of what's considered to be the center of a dataset:

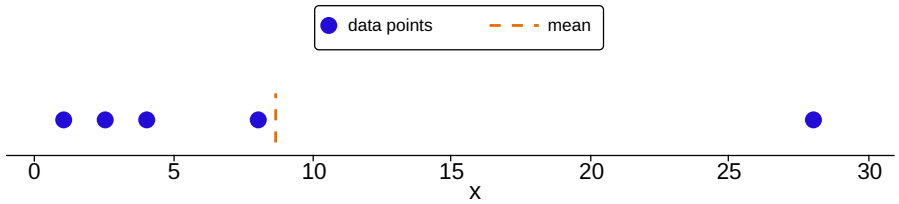
- Mean
- Weighted mean
- Geometric mean
- Harmonic mean
- Median
- Mode

Calculating Descriptive Statistics

Mean

The sample mean, also called the sample arithmetic mean or simply the average, is the arithmetic average of all the items in a dataset. The mean of a dataset x is mathematically expressed as $\sum_i x_i/n$ where $i = 1, ..., n$.

This figure illustrates the mean of a sample with five data points:
The blue dots represent the data points from our array `y`. The orange dashed line is their mean.



Calculating Descriptive Statistics

Mean

You can calculate the mean with pure Python, without importing libraries:

```
mean = sum(x) / len(x)
```

You can also apply built-in Python statistics functions:

```
meanval = statistics.mean(x)
meanfval = statistics.fmean(x)
```

`fmean()` is introduced in Python 3.8 as a faster alternative to `mean()`. It always returns a floating-point number.
In case there are `nan` values among your data, then `statistics.mean()` and `statistics.fmean()` will return `nan` as the output.
If you use NumPy, then you can get the mean with `np.mean()`.

Weighted Mean

The weighted mean, also called the weighted arithmetic mean or weighted average, generalizes the arithmetic mean with defining the relative contribution of each data point to the result.
For each data point x_i of the dataset x, a weight w_i is defined. The weighted mean is then computed using the following equation:

$$\frac{\sum_i (w_i x_i)}{\sum_i (w_i)}$$

Note: It is convenient to keep all the weights positive, $w_i \leq 0$, with their sum equal to one, $\sum_i w_i = 1$.

The weighted mean is very handy for calculating the mean of a dataset containing items that occur with given relative frequencies: Knowing the total number of items is not required to calculate the mean.

Weighted Mean

The weighted mean, also called the weighted arithmetic mean or weighted average, generalizes the arithmetic mean with defining the relative contribution of each data point to the result.
For each data point x_i of the dataset x, a weight w_i is defined. The weighted mean is then computed using the following equation:

$$\frac{\sum_i (w_i x_i)}{\sum_i (w_i)}$$

Note: It is convenient to keep all the weights positive, $w_i \leq 0$, with their sum equal to one, $\sum_i w_i = 1$.

The weighted mean is very handy for calculating the mean of a dataset containing items that occur with given relative frequencies: Knowing the total number of items is not required to calculate the mean.

In NumPy, you can use `np.average()` to get the weighted mean of NumPy arrays.

Harmonic Mean

The harmonic mean is the reciprocal of the mean of the reciprocals of all items in the dataset:

$$\frac{n}{\sum_i (1/x_i)}$$

where $i = 1, \ldots, n$.

Harmonic Mean

The harmonic mean is the reciprocal of the mean of the reciprocals of all items in the dataset:

$$\frac{n}{\sum_i (1/x_i)}$$

where $i = 1, ..., n$.

A pure Python implementation of the harmonic mean is the following:

```
hmean = len(x) / sum(1 / item for item in x)
print(hmean)
```

```
2.7613412228796843
```

The result is quite different from the value of the arithmetic mean for the same data `x`, which we calculated to be 8.7.

Harmonic Mean

You can also calculate this measure with `statistics.harmonic_mean()`:

```
hmean = statistics.harmonic_mean(x)
```

Again, if you have a `nan` value in a dataset, then it'll return `nan`. If there's at least one 0, then it'll return 0. If you provide at least one negative number, then you'll get `statistics.StatisticsError`.

Another way to calculate the harmonic mean is to use `scipy.stats.hmean()`.

If your dataset contains nan, 0, a negative number, or anything but positive numbers, then you'll get a `ValueError`.

Calculating Descriptive Statistics

Geometric Mean

The geometric mean is the n-th root of the product of all elements in a dataset x:

$$\sqrt[n]{\prod_i x_i}$$

where $i = 1, ..., n$.

Geometric Mean

The geometric mean is the n-th root of the product of all elements in a dataset x:

$$\sqrt[n]{\prod_i x_i}$$

where $i = 1, ..., n$.

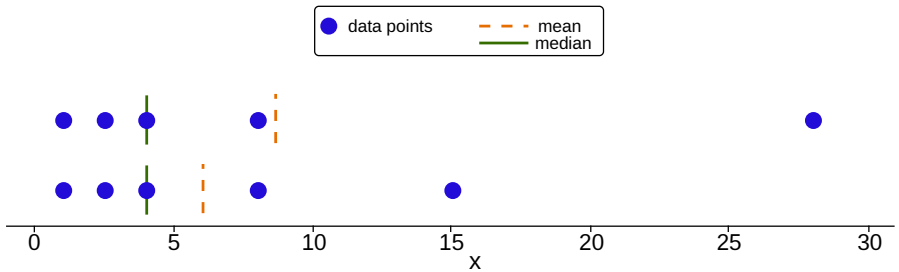
Python 3.8 introduced `statistics.geometric_mean()`, which converts all values to floating-point numbers and returns their geometric mean:
If you pass data with `nan` values, then `statistics.geometric_mean()` will behave like most similar functions and return `nan`. If there's a zero or negative number among your data, then `statistics.geometric_mean()` will raise the `statistics.StatisticsError`.

You can also get the geometric mean with `scipy.stats.gmean()`.
If you have *nan* values in a dataset, then `gmean()` will return `nan`. If there is at least one 0, then it will return 0.0 and give a warning. If you provide at least one negative number, then you'll get `nan` and a warning.

Median

The sample median is the middle element of a sorted (increasing or decreasing order) dataset with n elements. For n being odd, the median is the value at the middle position: $0.5(n+1)$. For an even nn, the median is the arithmetic mean of the values at the positions $0.5n$ and $0.5n+1$.

The main difference between the behavior of the mean and median is related to **dataset outliers or extremes**. The mean is heavily affected by outliers, but the median only depends on outliers either slightly or not at all. Consider the following figure:



The upper dataset has the items 1, 2.5, 4, 8, and 28. Its mean is 8.7, the median is 5. The lower dataset shows what happens if we change the dataset's maximum value: the mean changes but the median does not.

Median

You can compare the mean and median as one way to **detect outliers** and asymmetry in your data. Whether the mean value or the median value is more useful to you depends on the context of your particular problem.

Median

Here is one of many possible pure Python implementations of the median:

You can get the median with `statistics.median()`. For our array `x`, the sorted version is `[1, 2.5, 4, 8.0, 28.0]`, so the element in the middle is 4.

If you would remove an element by slicing `x[:-1]`, which is `x` without the last item 28.0, now there are two middle elements, 2.5 and 4. Their average is 3.25.

Unlike most other functions from the Python statistics library, `median()` doesn't return `nan` when there are `nan` values among the data points

You can also get the median with `np.median()`.

Measures of Variability

The measures of central tendency, which we saw so far, alone aren't sufficient to describe data. We also need the measures of variability that quantify the spread of data points. In the following we will learn how to identify and calculate the following variability measures:

- Variance
- Standard deviation
- Skewness
- Percentiles
- Ranges

Calculating Descriptive Statistics

Variance

The sample variance quantifies the spread of the data points from the mean. The sample variance of the dataset x with n elements is

$$s^2 = \frac{\sum_i (x_i - \text{mean}(x))^2}{(n-1)}$$

where $i = 1, ..., n$ and $\text{mean}(x)$ is the sample mean of x.

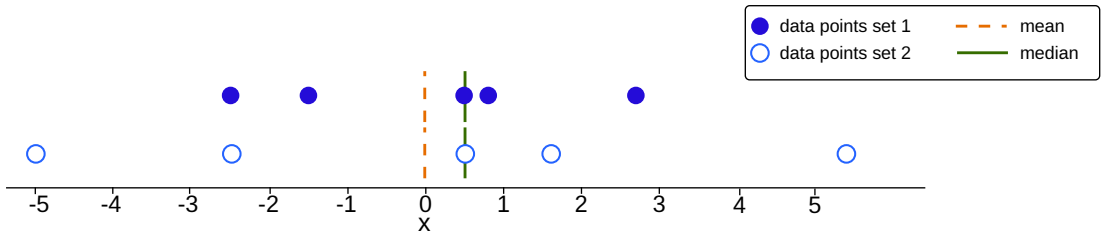
Variance

The sample variance quantifies the spread of the data points from the mean. The sample variance of the dataset x with n elements is

$$s^2 = \frac{\sum_i (x_i - \text{mean}(x))^2}{(n-1)}$$

where $i = 1, ..., n$ and $\text{mean}(x)$ is the sample mean of x.

The figure shows the importance of the variance when describing datasets:



Blue dots: This dataset has a smaller variance. It also has a smaller range or a smaller difference between the largest and smallest item.
White dots: This dataset has a larger variance. It also has a bigger range or a bigger difference between the largest and smallest item.

Calculating Descriptive Statistics

Variance

Here's how you can calculate the sample variance with pure Python:

```
n = len(x)
mean = sum(x) / n
var = sum((item - mean)**2 for item in x) / (n - 1)
```

This approach is sufficient and calculates the sample variance well. However, the shorter and more elegant solution is to call the existing function `statistics.variance()`:

```
var = statistics.variance(x)
```

If you have `nan` values among your data, then `statistics.variance()` will return `nan`.

Standard Deviation

The sample standard deviation is another measure of data spread. The standard deviation, s, is the positive square root of the sample variance. The standard deviation is often more convenient than the variance because it has the same unit as the data points. Once you get the variance, you can calculate the standard deviation with pure Python:

```
std = var ** 0.5
```

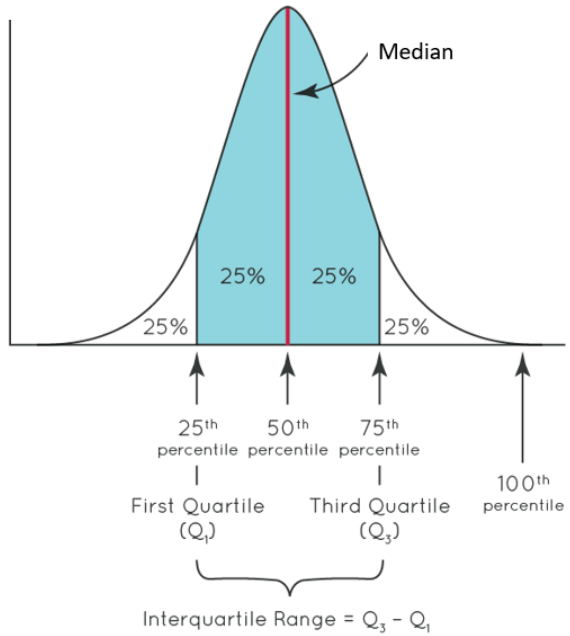
Although this solution works, you can also use `statistics.stdev()`:

```
std = statistics.stdev(x)
```

Percentiles

The sample "p percentile" is the element in the dataset such that p% of the elements in the dataset are less than or equal to that value.
Also, (100 - p)% of the elements are greater than or equal to that value. If there are two such elements in the dataset, then the sample "p percentile" is their arithmetic mean. Each dataset has three **quartiles**, which are the percentiles that divide the dataset into four parts:

- The first quartile is the sample 25th percentile. It divides roughly 25% of the smallest items from the rest of the dataset.
- The second quartile is the sample 50th percentile or the median. Approx. 25% of the items lie between the first and second quartiles and another 25% between the second and third quartiles.
- The third quartile is the sample 75th percentile. It divides roughly 25% of the largest items from the rest of the dataset.



Calculating Descriptive Statistics

Percentiles

If you want to divide your data into several intervals, then you can use `statistics.quantiles()`:

```
x = [-5.0, -1.1, 0.1, 2.0, 8.0, 12.8, 21.0, 25.8, 41.0]
print(statistics.quantiles(x, n=2))

print(statistics.quantiles(x, n=4, method='inclusive'))
```

Calculating Descriptive Statistics

Percentiles

If you want to divide your data into several intervals, then you can use `statistics.quantiles()`:

```
x = [-5.0, -1.1, 0.1, 2.0, 8.0, 12.8, 21.0, 25.8, 41.0]
print(statistics.quantiles(x, n=2))

print(statistics.quantiles(x, n=4, method='inclusive'))
```

The result is:

```
[8.0]
[0.1, 8.0, 21.0]
```

Calculating Descriptive Statistics

Ranges

The range of data is the difference between the maximum and minimum element in the dataset. You can get it with the the NumPy function `np.ptp()`.

This function returns `nan` if there are `nan` values in your NumPy array.

Calculating Descriptive Statistics

Printing Summary of Descriptive Statistics

SciPy and Pandas offer useful routines to get descriptive statistics with a single function or method call. You can use `scipy.stats.describe()`:

```
statssummary = scipy.stats.describe(y, ddof=1, bias=False)
print(statssummary)
```

Printing Summary of Descriptive Statistics

SciPy and Pandas offer useful routines to get descriptive statistics with a single function or method call. You can use `scipy.stats.describe()`:

```
statssummary = scipy.stats.describe(y, ddof=1, bias=False)
print(statssummary)
```

The result is:

```
DescribeResult(nobs=9, minmax=(-5.0, 41.0), mean=11.622222222222222,
variance=228.75194444444446, skewness=0.9249043136685094,
kurtosis=0.14770623629658886)
```

You have to provide the dataset as the first argument. The argument can be a NumPy array, list, tuple, or similar data structure. You can omit `ddof=1` since it's the default and only matters when you are calculating the variance. You can pass `bias=False` to force correcting the skewness and kurtosis for statistical bias.

Printing Summary of Descriptive Statistics

Particular values can be accessed with dot notation:

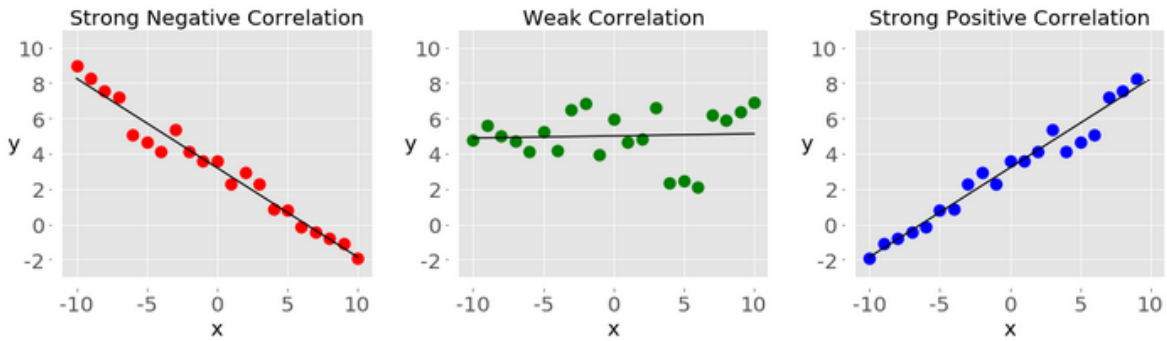
```
nobs = statssummary.nobs
min = statssummary.minmax[0]
max = statssummary.minmax[1]
print(statssummary)
```

Correlations

It is often necessary to examine the **relationship between two variables** in a dataset. This is done by calculating **measures of correlation** between pairs of data.

- **Positive correlation** exists when larger values of x correspond to larger values of y and vice versa.
- **Negative correlation** exists when larger values of x correspond to smaller values of y and vice versa.
- **Weak or no correlation** exists if there is no such apparent relationship.

The following figure illustrates negative, weak, and positive correlation:



Correlations

Important: Correlation is not a measure or indicator of causation, but only of association.

Correlations

The **sample covariance** is a measure that quantifies the strength and direction of a relationship between a pair of variables:

The covariance of the variables x and y is mathematically defined as

$$s_{xy} = \sum_i (x_i - \text{mean}(x))(y_i - \text{mean}(y))/(n-1),$$

where $i = 1, .., n$.

It follows that the covariance of two identical variables is actually the variance:

$$s_{xx} = \sum_i (x_i - \text{mean}(x))^2/(n-1) = (s_x)^2.$$

Correlations

In the following, we define some example data to work with these measures. We create two Python lists and use them to get corresponding NumPy arrays and Pandas series:

```
x = list(range(-10, 11))
y = [0, 2, 2, 2, 2, 3, 3, 6, 7, 4, 7, 6, 6, 9, 4, 5, 5, 10, 11, 12, 14]
x_, y_ = np.array(x), np.array(y)
x__, y__ = pd.Series(x_), pd.Series(y_)
```

You can calculate the covariance in pure Python using the above equation:

```
n = len(x)
mean_x, mean_y = sum(x) / n, sum(y) / n
 cov_xy = (sum((x[k] - mean_x) * (y[k] - mean_y) for k in range(n))
           / (n - 1))
```

The result will be 19.95.

Correlations

NumPy has the function `cov()` that returns the covariance matrix:

```
cov_matrix = np.cov(x_, y_)
print(cov_matrix)
```

```
array([[38.5       , 19.95      ],
       [19.95      , 13.91428571]])
```

Note that `cov()` has the optional parameters `bias` (which is by default `False`), and `ddof` (which is by default `None`). Their default values are suitable for getting the **sample covariance matrix**. The upper-left element of the covariance matrix is the covariance of x and x, or the variance of x. Similarly, the lower-right element is the covariance of y and y, or the variance of y.
The other two elements of the covariance matrix are equal and represent the actual covariance between x and y.

Statistical Modeling

In statistical modeling, regression analysis is a set of statistical processes for estimating the relationships between a dependent variable (often called the 'outcome' or 'response' variable, or a 'label' in machine learning parlance) and one or more independent variables (often called 'predictors', 'covariates', 'explanatory variables' or 'features'). The most common form of regression analysis is linear regression, in which one finds the line (or a more complex linear combination) that most closely fits the data according to a specific mathematical criterion.

Statistical Modeling

In statistical modeling, regression analysis is a set of statistical processes for estimating the relationships between a dependent variable (often called the 'outcome' or 'response' variable, or a 'label' in machine learning parlance) and one or more independent variables (often called 'predictors', 'covariates', 'explanatory variables' or 'features'). The most common form of regression analysis is linear regression, in which one finds the line (or a more complex linear combination) that most closely fits the data according to a specific mathematical criterion.

Regression can be used to **fit a function** to data.

Most regression models propose that Y_i is a function (**regression function**) of X_i and β, with e_i representing an additive error term that may stand in for un-modeled determinants of Y_i or random statistical noise:

$$Y_i = f(X_i, \beta) + e_i$$

The **goal** is to estimate the function $f(X_i, \beta)$ that most closely fits the data.

Regression

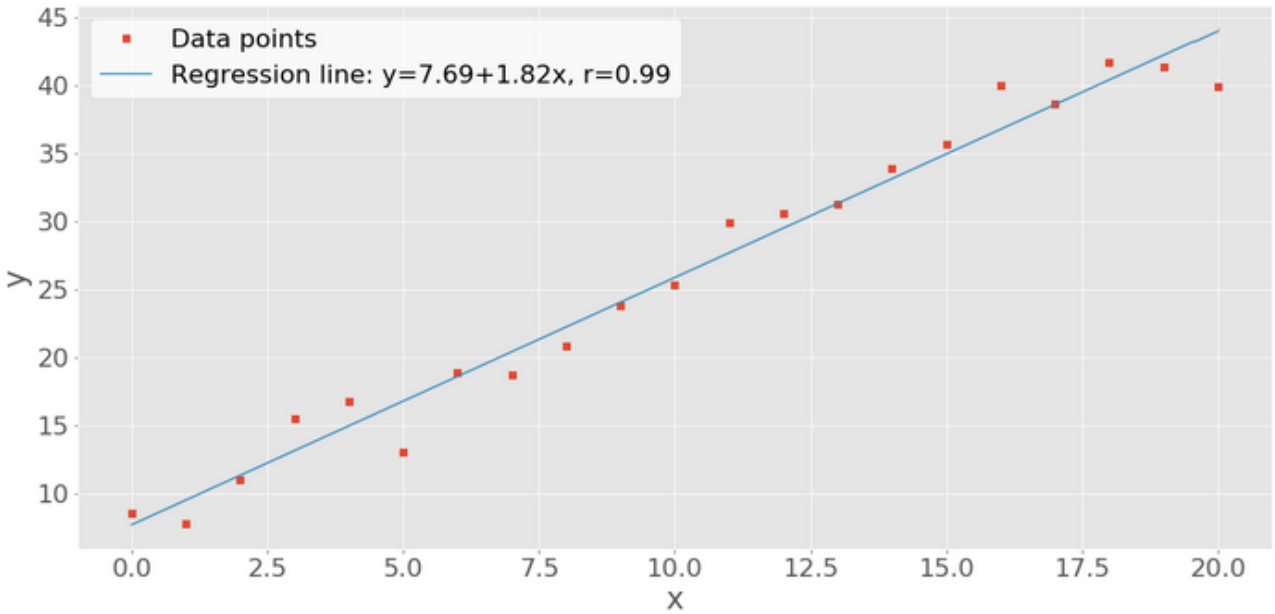
We generate two datasets and perform linear regression with `scipy.stats.linregress()`:

```
# generate data
x = np.arange(21)
y = 5 + 2 * x + 2 * np.random.randn(21)

# linear regression
slope, intercept, r, *__ = scipy.stats.linregress(x, y)
line = f'Regression line: y={intercept:.2f}+{slope:.2f}x, r={r:.2f}'

#plot
fig, ax = plt.subplots()
ax.plot(x, y, linewidth=0, marker='s', label='Data points')
ax.plot(x, intercept + slope * x, label=line)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend(facecolor='white')
plt.show()
```

Regression

The dataset `x` is the array with the integers from 0 to 20. `y` is calculated as a linear function of `x` distorted with some random noise.
linregress returns several values. We need the slope and intercept of the regression line, as well as the correlation coefficient `r`.

The result of the code before is this figure:

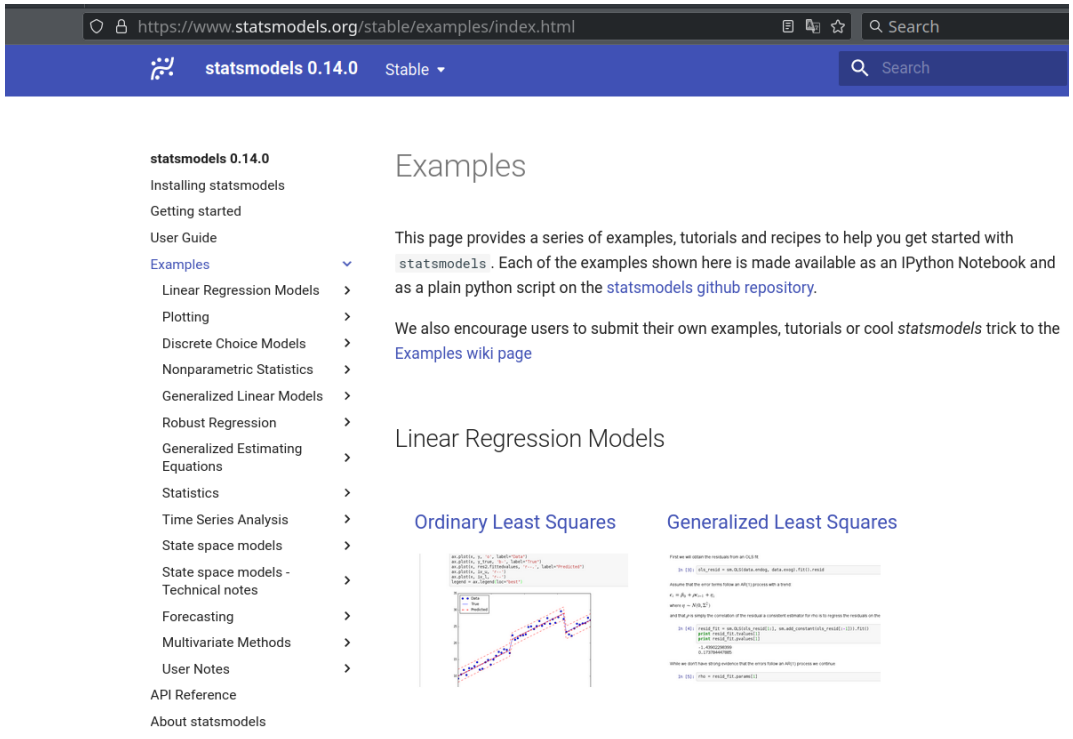


Statistical Modeling - More Use Cases

examples can be found here:

`https://www.statsmodels.org/stable/examples/index.html`

The methods shown there are especially useful for processing time series data and spectra.



Astronomical functionality with `astropy.stats`

The `astropy.stats` package holds statistical functions or algorithms used in astronomy. While the `scipy.stats` and `statsmodels` packages contains a wide range of statistical tools, they are general-purpose packages and are missing some tools that are particularly useful or specific to astronomy. This package is intended to provide such functionality.

See the documentation: `https://docs.astropy.org/en/stable/stats/`

Astronomical functionality with `astropy.stats`

A number of different tools are contained in the stats package. Please see the documentation for a complete list of tools and their different usages.
For example, sigma clipping, which is a common way to **estimate the background of an astronomical image**, can be performed with the `sigma_clip()` function. By default, the function returns a masked array, a type of Numpy array used for handling missing or invalid entries.
Masked arrays retain the original data but also store another boolean array of the same shape where `True` indicates that the value is masked.

Astronomical functionality with `astropy.stats`

A number of different tools are contained in the stats package. Please see the documentation for a complete list of tools and their different usages. For example, sigma clipping, which is a common way to **estimate the background of an astronomical image**, can be performed with the `sigma_clip()` function. By default, the function returns a masked array, a type of Numpy array used for handling missing or invalid entries.
Masked arrays retain the original data but also store another boolean array of the same shape where `True` indicates that the value is masked.

We get started by importing `astropy.stats`:

```
from astropy import stats
```

Astronomical functionality with `astropy.stats`

In the following, we see example code on how to apply sigma clipping:

```
import numpy as np
from astropy.stats import sigma_clip
x = np.array([1, 0, 0, 1, 99, 0, 0, 1, 0])
```

The mean is skewed by the outlier. Sigma-clipping (3 sigma by default) returns a masked array, and so functions like mean will ignore the outlier:

```
clipped = sigma_clip(x)
print(clipped)
```

Astronomical functionality with `astropy.stats`

In the following, we see example code on how to apply sigma clipping:

```
import numpy as np
from astropy.stats import sigma_clip
x = np.array([1, 0, 0, 1, 99, 0, 0, 1, 0])
```

The mean is skewed by the outlier. Sigma-clipping (3 sigma by default) returns a masked array, and so functions like mean will ignore the outlier:

```
clipped = sigma_clip(x)
print(clipped)
```

```
masked_array(data=[1, 0, 0, 1, --, 0, 0, 1, 0],
             mask=[False, False, False, False,  True, False,
                   False, False,
                   False],
       fill_value=999999)
```

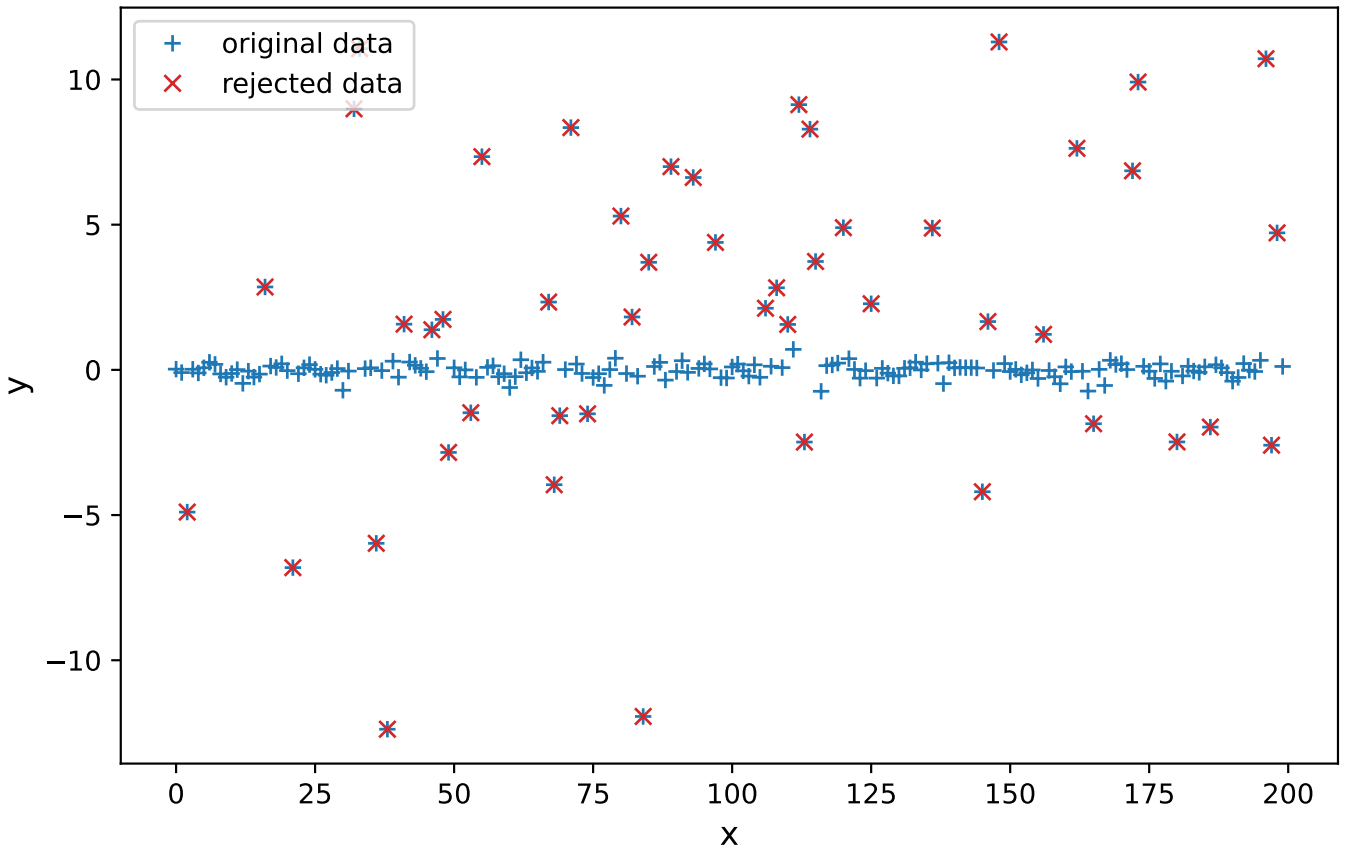
Astronomical functionality with `astropy.stats`

Here is some complete code including a plot to show the sigma clipping:

```
import numpy as np
import scipy.stats as stats
from matplotlib import pyplot as plt
from astropy.stats import sigma_clip, mad_std

# Generate fake data that has a mean of 0 and standard deviation of 0.2 with outliers
rng = np.random.default_rng(0)
x = np.arange(200)
y = np.zeros(200)
c = stats.bernoulli.rvs(0.35, size=x.shape)
y += (rng.normal(0., 0.2, x.shape) + c * rng.normal(3.0, 5.0, x.shape))

filtered_data = sigma_clip(y, sigma=3, maxiters=1, stdfunc=mad_std)

# plot the original and rejected data
plt.figure(figsize=(8,5))
plt.plot(x, y, '+', color='#1f77b4', label="original data")
plt.plot(x[filtered_data.mask], y[filtered_data.mask], 'x',
         color='#d62728', label="rejected data")
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc=2, numpoints=1)
```

Astronomical functionality with `astropy.stats`

we get the following plot:



Machine learning with `scikit-learn`

Scikit-learn is a package for **Machine Learning and Data Mining** that builts on NumPy, SciPy and matplotlib. This package provides many efficient algorithms covering various aspects of machine learning, such as preprocessing, feature extraction, dimensionality reduction, clustering, classification, regression, and model evaluation.

Machine learning with `scikit-learn`

Scikit-learn is a package for **Machine Learning and Data Mining** that builts on NumPy, SciPy and matplotlib. This package provides many efficient algorithms covering various aspects of machine learning, such as preprocessing, feature extraction, dimensionality reduction, clustering, classification, regression, and model evaluation.

The usage of scikit-learn is beyond the scope of this course.

Machine learning with `scikit-learn`

Scikit-learn is a package for **Machine Learning and Data Mining** that builts on NumPy, SciPy and matplotlib. This package provides many efficient algorithms covering various aspects of machine learning, such as preprocessing, feature extraction, dimensionality reduction, clustering, classification, regression, and model evaluation.

The usage of scikit-learn is beyond the scope of this course.

If you are interested in more on this: my courses on machine learning offer an introduction into this topic as well as more advanced applications.

Summary

Descriptive statistics measures are very imporant if dealing with large data sets, and they are the building blocks of more complex algorithms like those of machine learning, fitting algorithms and more.

Summary

Descriptive statistics measures are very imporant if dealing with large data sets, and they are the building blocks of more complex algorithms like those of machine learning, fitting algorithms and more.
We have seen many ways on how to carry out statistics on various levels by making use of Python packages. Each of these packages has its own strengths and weaknesses, so it is important to choose the right tool for the job.

Summary

Descriptive statistics measures are very imporant if dealing with large data sets, and they are the building blocks of more complex algorithms like those of machine learning, fitting algorithms and more.
We have seen many ways on how to carry out statistics on various levels by making use of Python packages. Each of these packages has its own strengths and weaknesses, so it is important to choose the right tool for the job.

- Use Python's statistics for the most important Python statistics functions.
- Use NumPy to handle arrays efficiently.
- Use SciPy for additional Python statistics routines for NumPy arrays.
- Use pandas to work with labeled datasets.
- Use Matplotlib to visualize data with plots, charts, and histograms.

An Outlook: Astronomical Data Sources

Next time we will continue with **how to access astronomical catalogs and other data sources** conveniently with Python.