Handling missing data

- https://www.digitalocean.com/community/tutorials/how-to-convert-data-types-in-python-3
- https://colab.research.google.com/github/hossainlab/pandas/blob/master/book/notebooks/09-Handling%20Missing%20Values.ipynb#scrollTo=lhqz0w6y7Y04

Visualizing Data

- Box plots
- Histograms
- Pie charts
- Bar charts
- X-Y plots
- Heatmaps

Box Plots

The **box plot** is an excellent tool to visually represent descriptive statistics of a given dataset. It can show the range, interquartile range, median, mode, outliers, and all quartiles. First, create some data to represent with a box plot:

Python >>>

```
>>> np.random.seed(seed=0)
>>> x = np.random.randn(1000)
>>> y = np.random.randn(100)
>>> z = np.random.randn(10)
```

The first statement sets the seed of the NumPy random number generator with `seed()`, so you can get the same results each time you run the code. You don't have to set the seed, but if you don't specify this value, then you'll get different results each time.

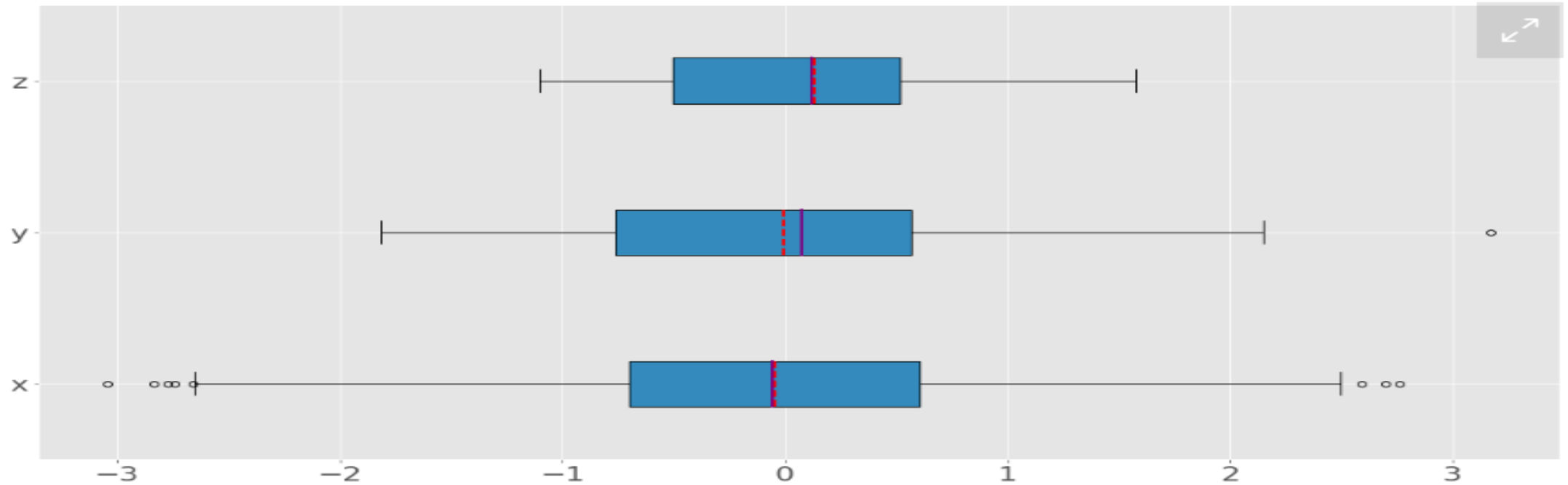
Python

```
fig, ax = plt.subplots()
ax.boxplot((x, y, z), vert=False, showmeans=True, meanline=True,
           labels=('x', 'y', 'z'), patch_artist=True,
           medianprops={'linewidth': 2, 'color': 'purple'},
           meanprops={'linewidth': 2, 'color': 'red'})
plt.show()
```

The parameters of `.boxplot()` define the following:

- **`x`** is your data.
- **`vert`** sets the plot orientation to horizontal when `False`. The default orientation is vertical.
- **`showmeans`** shows the mean of your data when `True`.
- **`meanline`** represents the mean as a line when `True`. The default representation is a point.
- **`labels:`** the labels of your data.
- **`patch_artist`** determines how to draw the graph.
- **`medianprops`** denotes the properties of the line representing the median.
- **`meanprops`** indicates the properties of the line or dot representing the mean.

There are other parameters, but their analysis is beyond the scope of this tutorial.



You can see three box plots. Each of them corresponds to a single dataset (x, y, or z) and show the following:

- **The mean** is the red dashed line.
- **The median** is the purple line.
- **The first quartile** is the left edge of the blue rectangle.
- **The third quartile** is the right edge of the blue rectangle.
- **The interquartile range** is the length of the blue rectangle.
- **The range** contains everything from left to right.
- **The outliers** are the dots to the left and right.

Python >>>

```
>>> hist, bin_edges = np.histogram(x, bins=10)
>>> hist
array([  9,  20,  70, 146, 217, 239, 160,  86,  38,  15])
>>> bin_edges
array([-3.04614305, -2.46559324, -1.88504342, -1.3044936 , -0.72394379,
       -0.14339397,  0.43715585,  1.01770566,  1.59825548,  2.1788053 ,
        2.75935511])
```

It takes the array with your data and the number (or edges) of bins and returns two NumPy arrays:

1. **`hist`** contains the frequency or the number of items corresponding to each bin.
2. **`bin_edges`** contains the edges or bounds of the bin.

What `histogram()` calculates, `.hist()` can show graphically:

Python

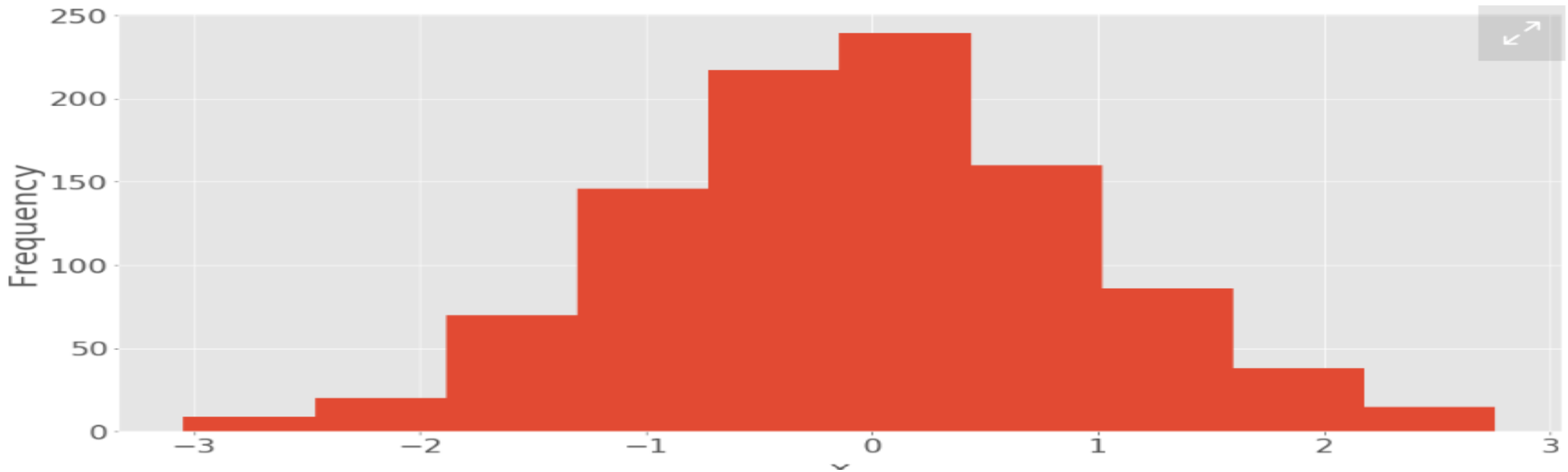
```
fig, ax = plt.subplots()
ax.hist(x, bin_edges, cumulative=False)
ax.set_xlabel('x')
ax.set_ylabel('Frequency')
plt.show()
```

What `histogram()` calculates, `.hist()` can show graphically:

Python

```
fig, ax = plt.subplots()
ax.hist(x, bin_edges, cumulative=False)
ax.set_xlabel('x')
ax.set_ylabel('Frequency')
plt.show()
```

The first argument of `.hist()` is the sequence with your data. The second argument defines the edges of the bins. The third disables the option to create a histogram with cumulative values. The code above produces a figure like this:

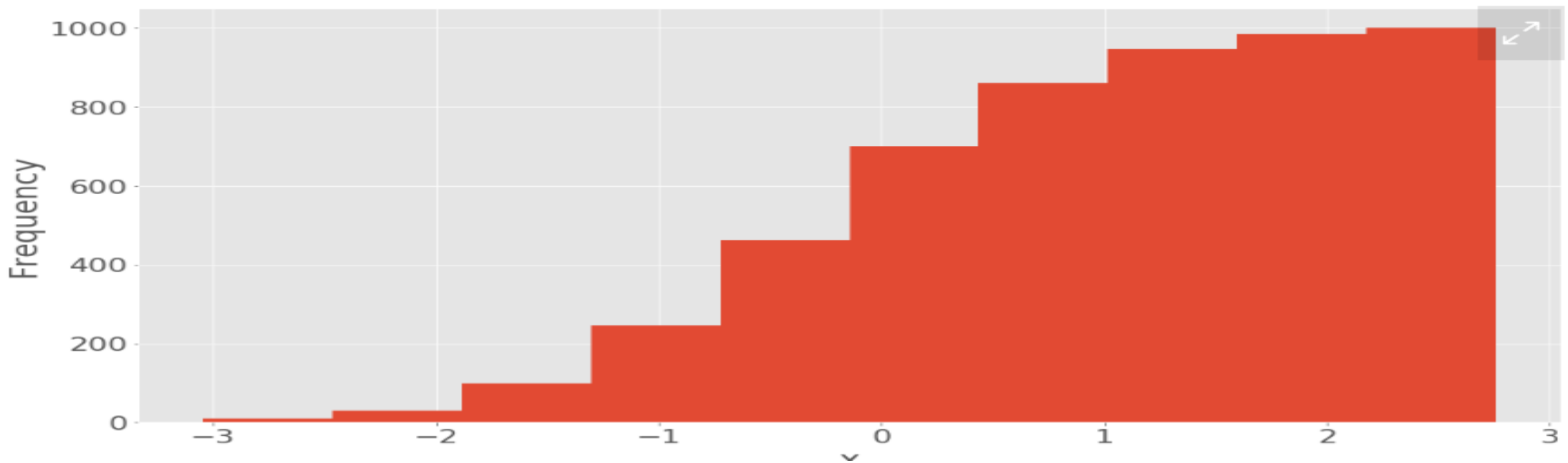


It's possible to get the histogram with the cumulative numbers of items if you provide the argument `cumulative=True` to `.hist()`:

Python

```
fig, ax = plt.subplots()
ax.hist(x, bin_edges, cumulative=True)
ax.set_xlabel('x')
ax.set_ylabel('Frequency')
plt.show()
```

This code yields the following figure:



Pie charts represent data with a small number of labels and given relative frequencies. They work well even with the labels that can't be ordered (like nominal data). A pie chart is a circle divided into multiple slices. Each slice corresponds to a single distinct label from the dataset and has an area proportional to the relative frequency associated with that label.

Let's define data associated to three labels:

Python >>>

```
>>> x, y, z = 128, 256, 1024
```

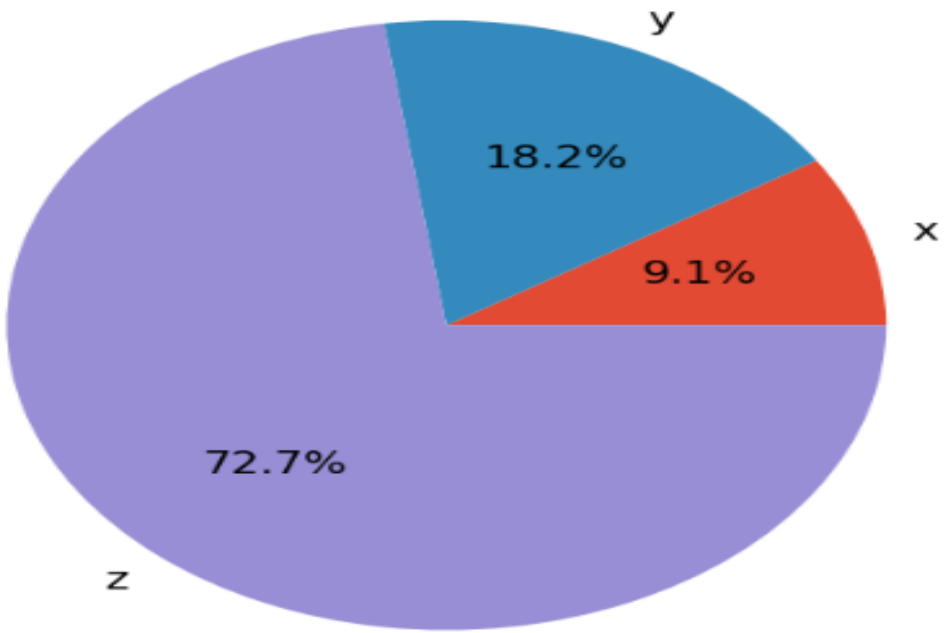
Now, create a pie chart with `.pie()`:

Python

```
fig, ax = plt.subplots()
ax.pie((x, y, z), labels=('x', 'y', 'z'), autopct='%1.1f%%')
plt.show()
```

The first argument of `.pie()` is your data, and the second is the sequence of the corresponding labels. `autopct` defines the format of the relative frequencies shown on the figure. You'll get a figure that looks like this:

The first argument of `.pie()` is your data, and the second is the sequence of the corresponding labels. `autopct` defines the format of the relative frequencies shown on the figure. You'll get a figure that looks like this:



The pie chart shows `x` as the smallest part of the circle, `y` as the next largest, and then `z` as the largest part. The percentages denote the relative size of each value compared to their sum.

Bar Charts

Bar charts also illustrate data that correspond to given labels or discrete numeric values. They can show the pairs of data from two datasets. Items of one set are the **labels**, while the corresponding items of the other are their **frequencies**. Optionally, they can show the errors related to the frequencies, as well.

The bar chart shows parallel rectangles called **bars**. Each bar corresponds to a single label and has a height proportional to the frequency or relative frequency of its label. Let's generate three datasets, each with 21 items:

Python >>>

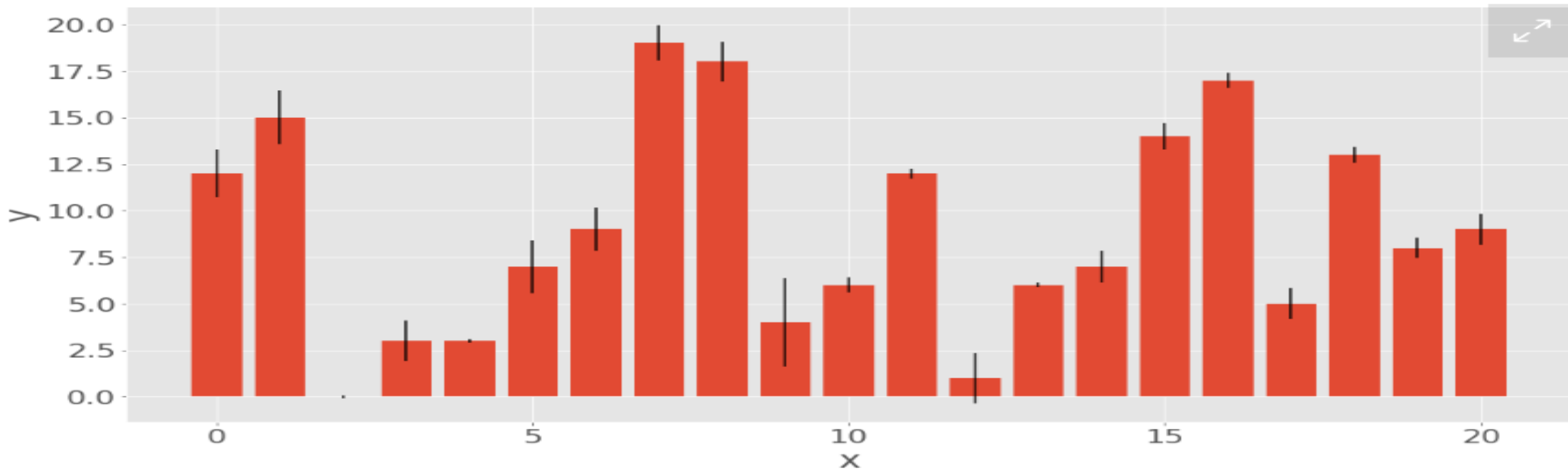
```
>>> x = np.arange(21)
>>> y = np.random.randint(21, size=21)
>>> err = np.random.randn(21)
```

You use `np.arange()` to get `x`, or the array of consecutive integers from `0` to `20`. You'll use this to represent the labels. `y` is an array of uniformly distributed random integers, also between `0` and `20`. This array will represent the frequencies. `err` contains normally distributed floating-point numbers, which are the errors. These values are optional.

You can create a bar chart with `.bar()` if you want vertical bars or `.barh()` if you'd like horizontal bars:

```
fig, ax = plt.subplots())
ax.bar(x, y, yerr=err)
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```

This code should produce the following figure:



The heights of the red bars correspond to the frequencies `y`, while the lengths of the black lines show the errors `err`. If you don't want to include the errors, then omit the parameter `yerr` of `.bar()`.

X-Y Plot

The **x-y plot** or **scatter plot** represents the pairs of data from two datasets. The horizontal x-axis shows the values from the set `x`, while the vertical y-axis shows the corresponding values from the set `y`. You can optionally include the regression line and the correlation coefficient. Let's generate two datasets and perform linear regression with `scipy.stats.linregress()`:

Python >>>

```
>>> x = np.arange(21)
>>> y = 5 + 2 * x + 2 * np.random.randn(21)
>>> slope, intercept, r, *__ = scipy.stats.linregress(x, y)
>>> line = f'Regression line: y={intercept:.2f}+{slope:.2f}x, r={r:.2f}'
```

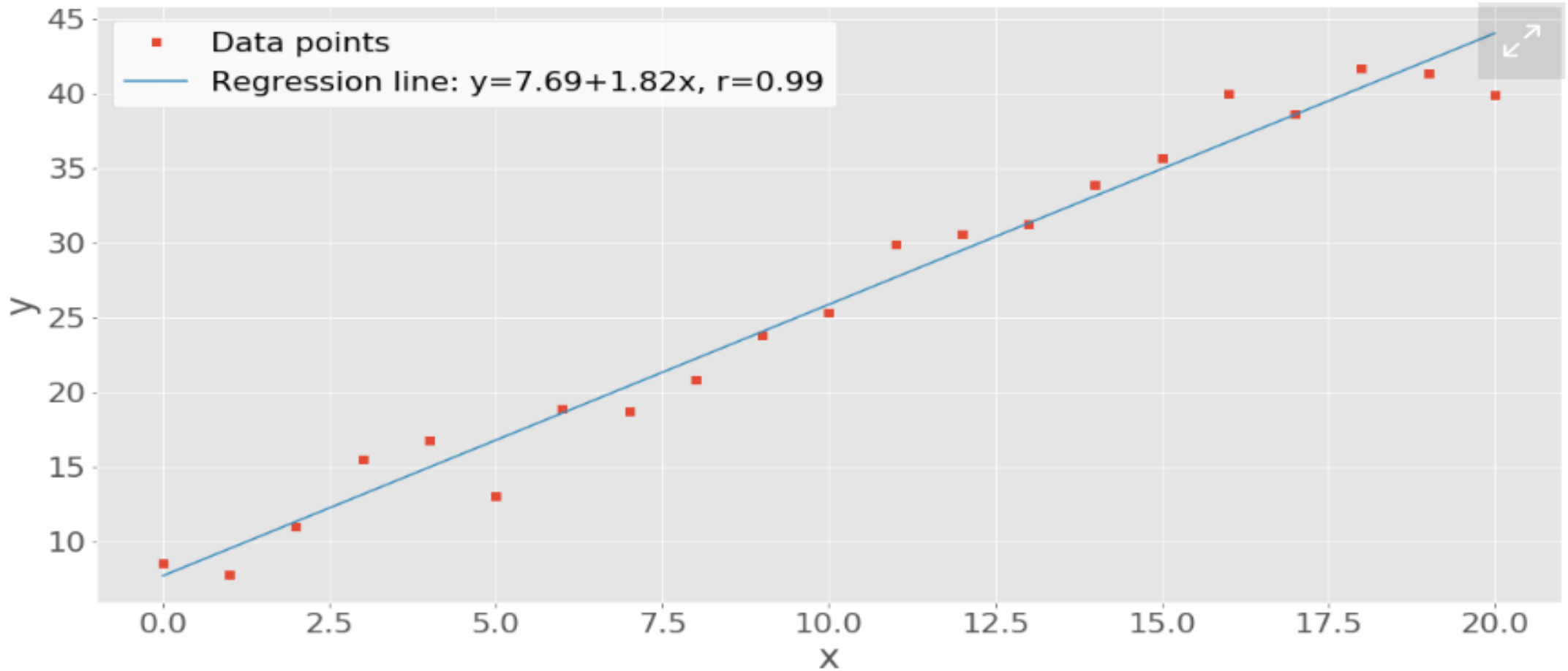
The dataset `x` is again the array with the integers from 0 to 20. `y` is calculated as a linear function of `x` distorted with some random noise.

`linregress` returns several values. You'll need the `slope` and `intercept` of the regression line, as well as the correlation coefficient `r`. Then you can apply `.plot()` to get the x-y plot:

Python

```
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()
```

The result of the code above is this figure:



You can see the data points (x-y pairs) as red squares, as well as the blue regression line.

Heatmaps

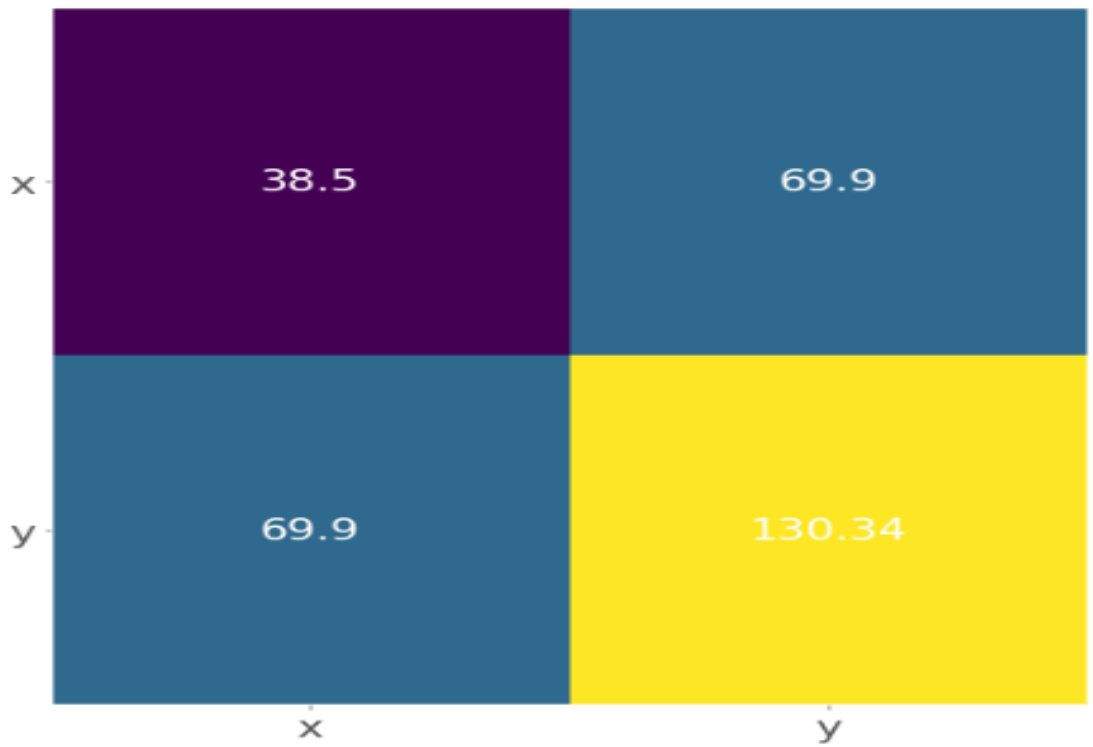
A **heatmap** can be used to visually show a matrix. The colors represent the numbers or elements of the matrix. Heatmaps are particularly useful for illustrating the covariance and correlation matrices. You can create the heatmap for a covariance matrix with `.imshow()`:

Python

```
matrix = np.cov(x, y).round(decimals=2)
fig, ax = plt.subplots()
ax.imshow(matrix)
ax.grid(False)
ax.xaxis.set(ticks=(0, 1), ticklabels=('x', 'y'))
ax.yaxis.set(ticks=(0, 1), ticklabels=('x', 'y'))
ax.set_ylim(1.5, -0.5)
for i in range(2):
    for j in range(2):
        ax.text(j, i, matrix[i, j], ha='center', va='center', color='w')
plt.show()
```

Here, the heatmap contains the labels `'x'` and `'y'` as well as the numbers from the covariance matrix. You'll get a figure like this:

Here, the heatmap contains the labels `'x'` and `'y'` as well as the numbers from the covariance matrix. You'll get a figure like this:



The yellow field represents the largest element from the matrix `130.34`, while the purple one corresponds to the smallest element `38.5`. The blue squares in between are associated with the value `69.9`.

You can obtain the heatmap for the correlation coefficient matrix following the same logic:

You can obtain the heatmap for the correlation coefficient matrix following the same logic:

Python

```
matrix = np.corrcoef(x, y).round(decimals=2)
fig, ax = plt.subplots()
ax.imshow(matrix)
ax.grid(False)
ax.xaxis.set(ticks=(0, 1), ticklabels=('x', 'y'))
ax.yaxis.set(ticks=(0, 1), ticklabels=('x', 'y'))
ax.set_ylim(1.5, -0.5)
for i in range(2):
    for j in range(2):
        ax.text(j, i, matrix[i, j], ha='center', va='center', color='w')
plt.show()
```

The result is the figure below:

The yellow color represents the value `1.0`, and the purple color shows `0.99`.

- Python Statistics Fundamentals: How to Describe Your Data – Real Python

Objective of statistics of dataset

- What **numerical quantities** you can use to describe and summarize your datasets
- How to **calculate** descriptive statistics in pure Python
- How to get **descriptive statistics** with available Python libraries
- How to **visualize** your datasets

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.

You can apply descriptive statistics to one or many datasets or variables. When you describe and summarize a single variable, you're performing univariate analysis. When you search for statistical relationships among a pair of variables, you're doing a bivariate analysis. Similarly, a multivariate analysis is concerned with multiple variables at once.

Types of Measures

- **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.

- In statistics, the **population** is a set of all elements or items that you're interested in. Populations are often vast, which makes them inappropriate for collecting and analyzing data. That's why statisticians usually try to make some conclusions about a population by choosing and examining a representative subset of that population.
- This subset of a population is called a **sample**. Ideally, the sample should preserve the essential statistical features of the population to a satisfactory extent. That way, you'll be able to use the sample to glean conclusions about the population.

Outliers

- An **outlier** is a data point that differs significantly from the majority of the data taken from a sample or population. There are many possible causes of outliers, but here are a few to start you off:
- **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. For example, the limitations of measurement instruments or procedures can mean that the correct data is simply not obtainable. Other errors can be caused by miscalculations, data contamination, human error, and more.

Python Statistics Libraries

- There are many Python statistics libraries out there for you to work with, but in this tutorial, you'll be learning about some of the most popular and widely used ones:
- **Python's statistics** is a built-in Python library for descriptive statistics. You can use it if your datasets are not too large or if you can't rely on importing other libraries.
- **NumPy** is a third-party library for numerical computing, optimized for working with single- and multi-dimensional arrays. Its primary type is the array type called ndarray. This library contains many routines for statistical analysis.

- **SciPy** is a third-party library for scientific computing based on NumPy. It offers additional functionality compared to NumPy, including scipy.stats for statistical analysis.
- **Pandas** is a third-party library for numerical computing based on NumPy. It excels in handling labeled one-dimensional (1D) data with Series objects and two-dimensional (2D) data with DataFrame objects.
- **Matplotlib** is a third-party library for data visualization. It works well in combination with NumPy, SciPy, and Pandas.

Calculating Descriptive Statistics

Python

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

Python

```
>>> x = [8.0, 1, 2.5, 4, 28.0]
>>> x_with_nan = [8.0, 1, 2.5, math.nan, 4, 28.0]
>>> x
[8.0, 1, 2.5, 4, 28.0]
>>> x_with_nan
[8.0, 1, 2.5, nan, 4, 28.0]
```

```
>>> math.isnan(np.nan), np.isnan(math.nan)
(True, True)
>>> math.isnan(y_with_nan[3]), np.isnan(y_with_nan[3])
(True, True)
```

Measures of Central Tendency

- 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. In this tutorial, you'll learn how to identify and calculate these measures of central tendency:
- Mean
- Weighted mean
- Geometric mean
- Harmonic mean
- Median
- Mode

```
>>> mean_ = statistics.mean(x)
>>> mean_
8.7
>>> mean_ = statistics.fmean(x)
>>> mean_
8.7
```

```
>>> mean_ = statistics.mean(x_with_nan)
>>> mean_
nan
>>> mean_ = statistics.fmean(x_with_nan)
>>> mean_
nan
```

If you use NumPy, then you can get the mean with `np.mean()`:

```
Python >>>
>>> mean_ = np.mean(y)
>>> mean_
8.7
```

In the example above, `mean()` is a function, but you can use the corresponding method `.mean()` as well:

```
Python >>>
>>> mean_ = y.mean()
>>> mean_
8.7
```

The function `mean()` and method `.mean()` from NumPy return the same result as `statistics.mean()`. This is also the case when there are `nan` values among your data:

```
Python >>>
>>> np.mean(y_with_nan)
nan
>>> y_with_nan.mean()
nan
```

You often don't need to get a `nan` value as a result. If you prefer to ignore `nan` values, then you can use `np.nanmean()`:

Python >>>

```
>>> np.nanmean(y_with_nan)
8.7
```

`nanmean()` simply ignores all `nan` values. It returns the same value as `mean()` if you were to apply it to the dataset without the `nan` values.

`pd.Series` objects also have the method `.mean()`:

Python >>>

```
>>> mean_ = z.mean()
>>> mean_
8.7
```

As you can see, it's used similarly as in the case of NumPy. However, `.mean()` from Pandas ignores `nan` values by default:

Python >>>

```
>>> z_with_nan.mean()
8.7
```

This behavior is the result of the default value of the optional parameter `skipna`. You can change this parameter to modify the behavior.

Weighted Mean

The **weighted mean**, also called the **weighted arithmetic mean** or **weighted average**, is a generalization of the arithmetic mean that enables you to define the relative contribution of each data point to the result.

You define one **weight w_i** for each data point x_i of the dataset x, where $i = 1, 2, \ldots, n$ and n is the number of items in x. Then, you multiply each data point with the corresponding weight, sum all the products, and divide the obtained sum with the sum of weights: $\sum_i(w_i x_i) / \sum_i w_i$.

> **Note:** It's convenient (and usually the case) that all weights are **nonnegative**, $w_i \geq 0$, and that their sum is equal to one, or $\sum_i w_i = 1$.

The weighted mean is very handy when you need the mean of a dataset containing items that occur with given relative frequencies. For example, say that you have a set in which 20% of all items are equal to 2, 50% of the items are equal to 4, and the remaining 30% of the items are equal to 8. You can calculate the mean of such a set like this:

Python >>>

```
>>> 0.2 * 2 + 0.5 * 4 + 0.3 * 8
4.8
```

Here, you take the frequencies into account with the weights. With this method, you don't need to know the total number of items.

You can implement the weighted mean in pure Python by combining `sum()` with either

Weighted Mean

The **weighted mean**, also called the **weighted arithmetic mean** or **weighted average**, is a generalization of the arithmetic mean that enables you to define the relative contribution of each data point to the result.

You define one **weight w_i** for each data point x_i of the dataset x, where $i = 1, 2, \ldots, n$ and n is the number of items in x. Then, you multiply each data point with the corresponding weight, sum all the products, and divide the obtained sum with the sum of weights: $\sum_i(w_i x_i) / \sum_i w_i$.

> **Note:** It's convenient (and usually the case) that all weights are **nonnegative**, $w_i \geq 0$, and that their sum is equal to one, or $\sum_i w_i = 1$.

The weighted mean is very handy when you need the mean of a dataset containing items that occur with given relative frequencies. For example, say that you have a set in which 20% of all items are equal to 2, 50% of the items are equal to 4, and the remaining 30% of the items are equal to 8. You can calculate the mean of such a set like this:

Python >>>

```
>>> 0.2 * 2 + 0.5 * 4 + 0.3 * 8
4.8
```

Here, you take the frequencies into account with the weights. With this method, you don't need to know the total number of items.

You can implement the weighted mean in pure Python by combining `sum()` with either

The result is the same as in the case of the pure Python implementation. You can also use this method on ordinary lists and tuples.

Another solution is to use the element-wise product `w * y` with `np.sum()` or `.sum()`:

Python >>>

```
>>> (w * y).sum() / w.sum()
6.95
```

That's it! You've calculated the weighted mean.

However, be careful if your dataset contains `nan` values:

Python >>>

```
>>> w = np.array([0.1, 0.2, 0.3, 0.0, 0.2, 0.1])
>>> (w * y_with_nan).sum() / w.sum()
nan
>>> np.average(y_with_nan, weights=w)
nan
>>> np.average(z_with_nan, weights=w)
nan
```

In this case, `average()` returns `nan`, which is consistent with `np.mean()`.

Harmonic Mean

The **harmonic mean** is the reciprocal of the mean of the reciprocals of all items in the dataset: $n / \Sigma_i(1/x_i)$, where $i = 1, 2, \ldots, n$ and n is the number of items in the dataset x. One variant of the pure Python implementation of the harmonic mean is this:

Python >>>

```
>>> hmean = len(x) / sum(1 / item for item in x)
>>> hmean
2.7613412228796843
```

It's quite different from the value of the arithmetic mean for the same data `x`, which you calculated to be 8.7.

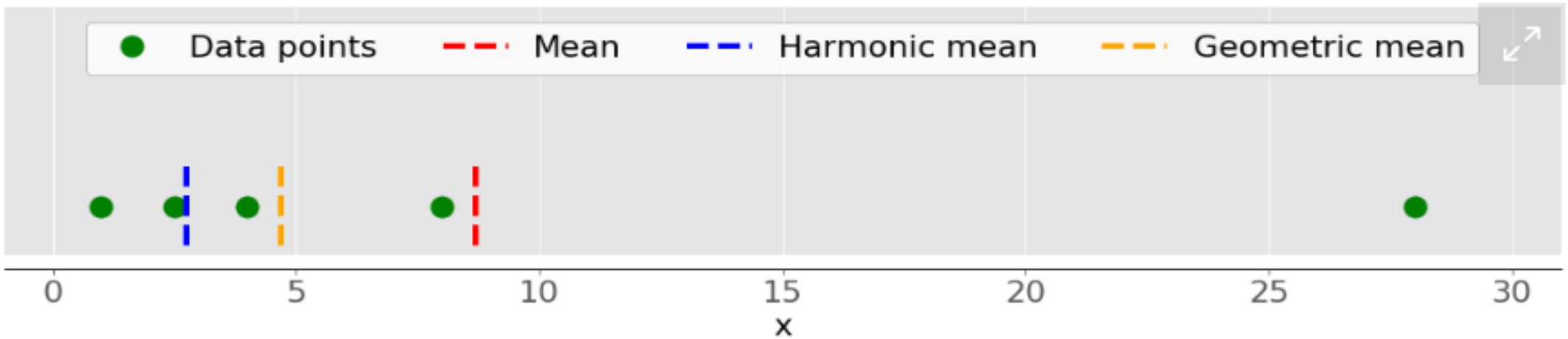
You can also calculate this measure with `statistics.harmonic_mean()`:

Python >>>

```
>>> hmean = statistics.harmonic_mean(x)
>>> hmean
2.7613412228796843
```

Geometric Mean

The **geometric mean** is the n-th root of the product of all n elements x_i in a dataset x: $\sqrt[n]{(\prod_i x_i)}$, where $i = 1, 2, \ldots, n$. The following figure illustrates the arithmetic, harmonic, and geometric means of a dataset:



Again, the green dots represent the data points 1, 2.5, 4, 8, and 28. The red dashed line is the mean. The blue dashed line is the harmonic mean, and the yellow dashed line is the geometric mean.

Again, the green dots represent the data points 1, 2.5, 4, 8, and 28. The red dashed line is the mean. The blue dashed line is the harmonic mean, and the yellow dashed line is the geometric mean.

You can implement the geometric mean in pure Python like this:

Python >>>

```
>>> gmean = 1
>>> for item in x:
...     gmean *= item
...
>>> gmean **= 1 / len(x)
>>> gmean
4.677885674856041
```

As you can see, the value of the geometric mean, in this case, differs significantly from the values of the arithmetic (8.7) and harmonic (2.76) means for the same dataset `x`.

Python 3.8 introduced `statistics.geometric_mean()`, which converts all values to floating-point numbers and returns their geometric mean:

Python >>>

```
>>> gmean = statistics.geometric_mean(x)
>>> gmean
4.67788567485604
```

Python >>>

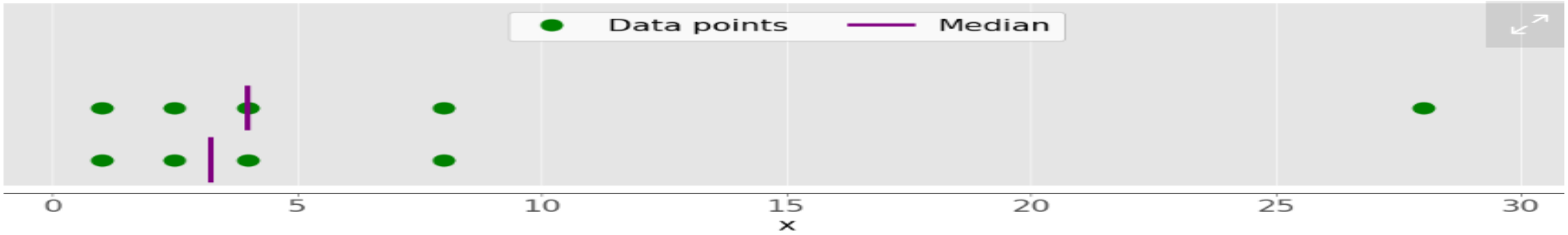
```
>>> scipy.stats.hmean(y)
2.7613412228796843
>>> scipy.stats.hmean(z)
2.7613412228796843
```

Again, this is a pretty straightforward implementation. However, if your dataset contains `nan`, `0`, a negative number, or anything but positive numbers, then you'll get a `ValueError`!

Median

The **sample median** is the middle element of a sorted dataset. The dataset can be sorted in increasing or decreasing order. If the number of elements n of the dataset is odd, then the median is the value at the middle position: $0.5(n + 1)$. If n is even, then the median is the arithmetic mean of the two values in the middle, that is, the items at the positions $0.5n$ and $0.5n + 1$.

For example, if you have the data points 2, 4, 1, 8, and 9, then the median value is 4, which is in the middle of the sorted dataset (1, 2, 4, 8, 9). If the data points are 2, 4, 1, and 8, then the median is 3, which is the average of the two middle elements of the sorted sequence (2 and 4). The following figure illustrates this:



The data points are the green dots, and the purple lines show the median for each dataset. The median value for the upper dataset (1, 2.5, 4, 8, and 28) is 4. If you remove the outlier 28 from the lower dataset, then the median becomes the arithmetic average between 2.5 and 4, which is 3.25.

PCA

https://colab.research.google.com/drive/1VOpZTHDJjna2DU6PvVrPWaayQh9hii6t

https://colab.research.google.com/github/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/05.09-Principal-Component-Analysis.ipynb#scrollTo=Jtl5ZErQ4a2Z

https://www.section.io/engineering-education/kernel-pca-in-python/

https://andreask.cs.illinois.edu/cs357-s15/public/demos/05-orthgonality/Orthogonal%20projection.html

Linear Discriminant Analysis.

- A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.
- The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix.
- The fitted model can also be used to reduce the dimensionality of the input by projecting it to the most discriminative directions, using the transform method.

- *class* sklearn.discriminant_analysis.**LinearDiscriminantAnalysis**(*solver='svd', shrinkage=None, priors=None, n_components=None, store_covariance=False, tol=0.0001, covariance_estimator=None*)

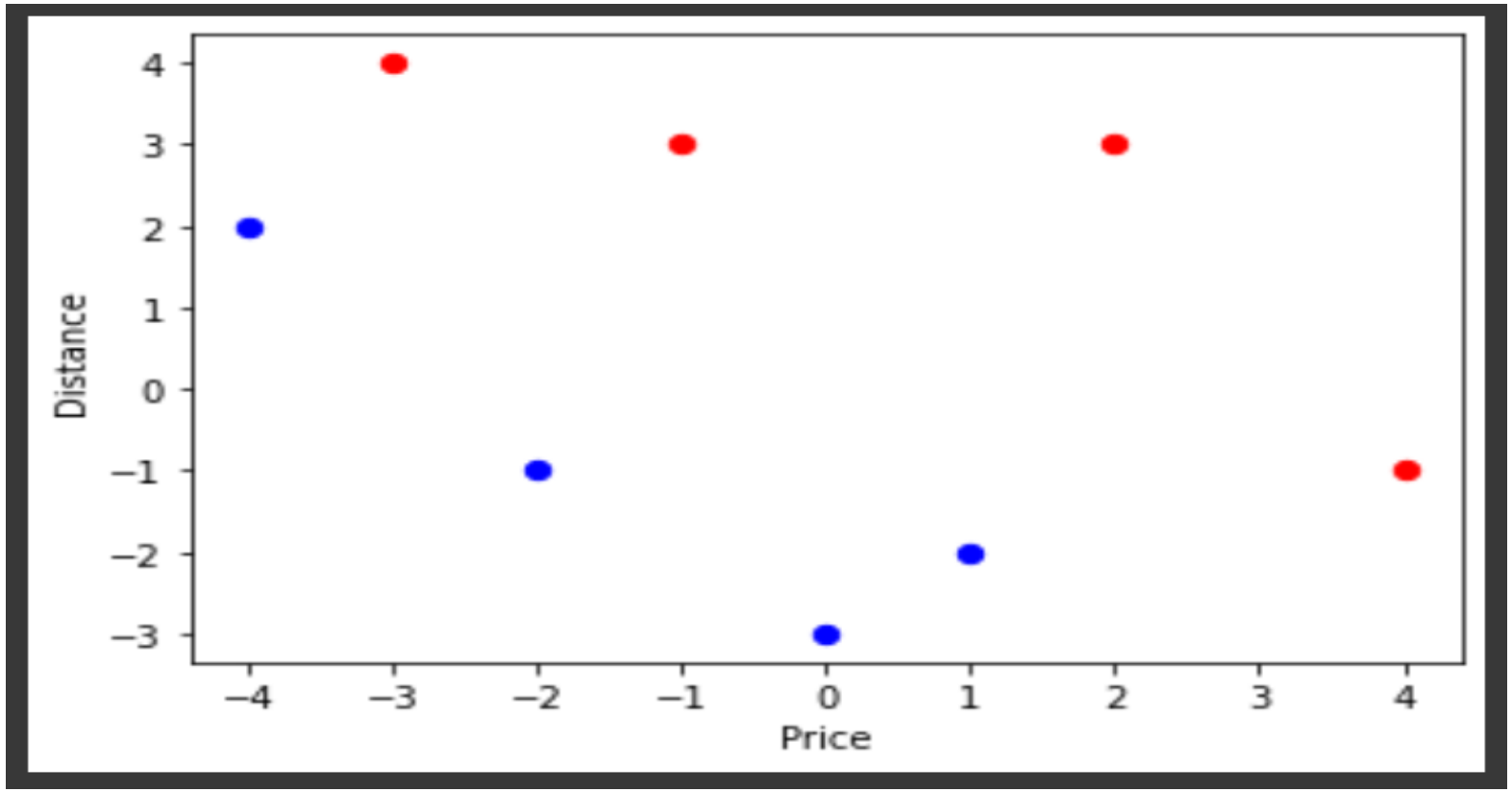
- Solver to use, possible values:‘svd’: Singular value decomposition (default). Does not compute the covariance matrix, therefore this solver is recommended for data with a large number of features.
- ‘lsqr’: Least squares solution. Can be combined with shrinkage or custom covariance estimator.
- ‘eigen’: Eigenvalue decomposition. Can be combined with shrinkage or custom covariance estimator.
- Linear Discriminant Analysis.ipynb - Colaboratory (google.com)

```
# Import the libraries we need
import numpy as np
import pandas as pd
from numpy.linalg import matrix_rank
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

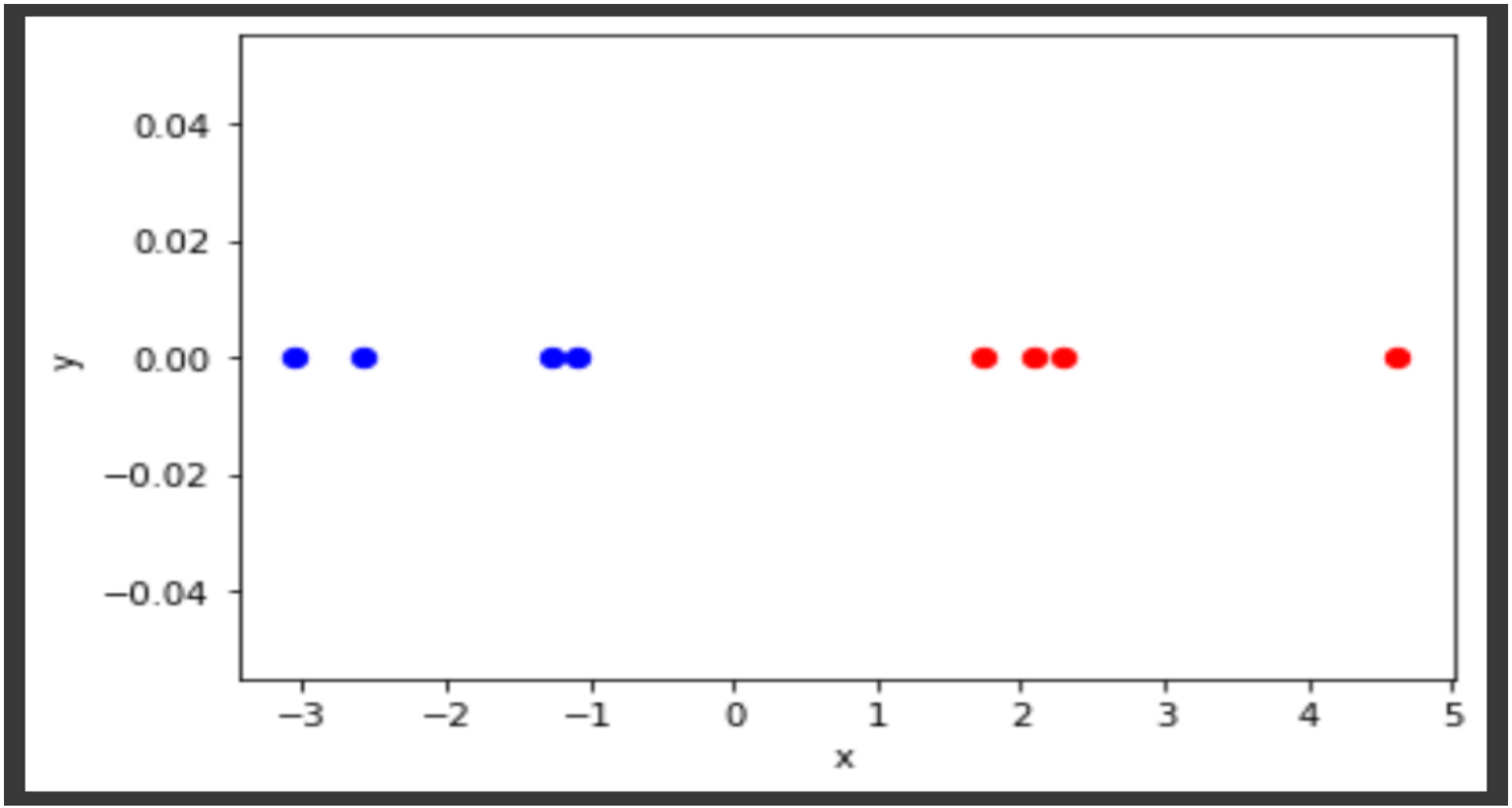
Perform LDA on the 2D case

- First we load the same data

```
data_url = 'https://omarshehata.github.io/lda-explorable/data/sample-2d.csv'
data = pd.read_csv(data_url)

# Flip the Y to match the JavaScript graph
data['Distance'] = -data['Distance']

ax = data[data['class'] == 0].plot.scatter(x='Price', y='Distance', c='red', s=40);
ax = data[data['class'] == 1].plot.scatter(x='Price', y='Distance', c='blue', s=40,ax=ax);
```


Distance
Price
4
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3
4

```
# Initialize the LDA to reduce to 1 dimension
LDA = LinearDiscriminantAnalysis(n_components=1, solver='eigen')
# Separate the data and class
Features = data.loc[:,'Price':'Distance'].values
Classes = data['class']
# Find the best projection
ProjectedData = LDA.fit_transform(Features, Classes)
# Put it in a dataframe and plot it
n = ProjectedData.shape[0]
newData = pd.DataFrame(data={
    'class' : Classes,
    'x' : ProjectedData.reshape(n),
    'y' : np.zeros(n)
})

ax = newData[newData['class'] == 0].plot.scatter(x='x', y='y', c='red', s=40);
ax = newData[newData['class'] == 1].plot.scatter(x='x', y='y', c='blue', s=40,ax=ax);
```


0.04
0.02
0.00
−0.02
−0.04
y
−3
−2
−1
0
1
2
3
4
5
x