In [1]:

```python
import pandas as pd
url = (
    "https://archive.ics.uci.edu/ml/machine-learning-databases"
    "/abalone/abalone.data"
)
abalone = pd.read_csv(url, header=None)
```

In [2]:

```python
abalone.head()
```

Out[2]:

	0	1	2	3	4	5	6	7	8
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

In [3]:

```python
abalone.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4177 entries, 0 to 4176
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       4177 non-null   object 
 1   1       4177 non-null   float64
 2   2       4177 non-null   float64
 3   3       4177 non-null   float64
 4   4       4177 non-null   float64
 5   5       4177 non-null   float64
 6   6       4177 non-null   float64
 7   7       4177 non-null   float64
 8   8       4177 non-null   int64  
dtypes: float64(7), int64(1), object(1)
memory usage: 293.8+ KB
```

In [4]:

```python
abalone.shape
```

Out[4]:

```
(4177, 9)
```

You can see that the column names are still missing.

You can find those names in the abalone.names file on the UCI machine learning repository.

You can add them to your DataFrame as follows:

In [5]:

```
abalone.columns = [
    "Sex",
    "Length",
    "Diameter",
    "Height",
    "Whole weight",
    "Shucked weight",
    "Viscera weight",
    "Shell weight",
    "Rings",
]
```

In [6]:

```
abalone.head()
```

Out[6]:

	Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

The imported data should now be more understandable.

But there’s one other thing that you should do: You should remove the Sex column.

The goal of the current exercise is to use physical measurements to predict the age of the abalone.

Since sex is not a purely physical measure, you should remove it from the dataset.

You can delete the Sex column using .drop:

In [7]:

```
abalone = abalone.drop("Sex", axis=1)
```

In [8]:

```
abalone.head()
```

Out[8]:

	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
0	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

Descriptive Statistics From the Abalone Dataset

When working on machine learning, you need to have an idea of the data you're working with.

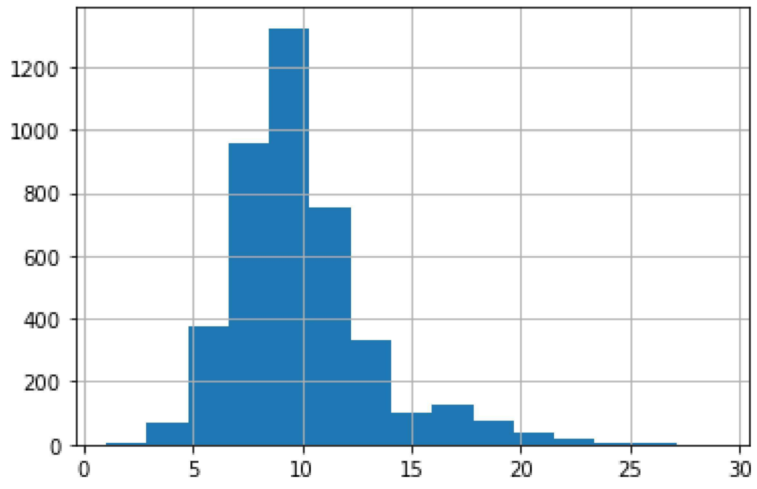
Without going into too much depth, here's a look at some exploratory statistics and graphs.

The target variable of this exercise is Rings, so you can start with that.

A histogram will give you a quick and useful overview of the age ranges that you can expect:

In [9]:

```
import matplotlib.pyplot as plt
abalone["Rings"].hist(bins=15)
plt.show()
```



The code below uses the pandas plotting functionality to generate a histogram with fifteen bins.

The decision to use fifteen bins is based on a few trials.

When defining the number of bins, you generally try to have neither too many observations per bin nor too few.

Too few bins can hide certain patterns, while too many bins can make the histogram lack smoothness.

You can see the histogram in the following graph:

```
The histogram shows that most abalones in the dataset have between five and fifteen rings, but that it’s possible to get up to twenty-five rings.
The older abalones are underrepresented in this dataset.
This seems intuitive, as age distributions are generally skewed like this due to natural processes.

A second relevant exploration is to find out which of the variables, if any, have a strong correlation with the age.
A strong correlation between an independent variable and your goal variable would be a good sign, as this would confirm that physical measurements and age are related.
```

You can observe the complete correlation matrix in correlation_matrix.

The most important correlations are the ones with the target variable Rings.

You can get those correlations like this:

In [10]:

```python
correlation_matrix = abalone.corr()
correlation_matrix["Rings"]
```

Out[10]:

```
Length            0.556720
Diameter          0.574660
Height            0.557467
Whole weight      0.540390
Shucked weight    0.420884
Viscera weight    0.503819
Shell weight      0.627574
Rings             1.000000
Name: Rings, dtype: float64
```

```
Now look at the correlation coefficients for Rings with the other variables.
The closer they are to 1, the more correlation there is.

You can conclude that there’s at least some correlation between physical measurements of adult abalones and their age, yet it’s also not very high.
Very high correlations mean that you can expect a straightforward modeling process.
In this case, you’ll have to try and see what results you can obtain using the kNN algorithm.
```

A Step-by-Step kNN From Scratch in Python

Plain English Walkthrough of the kNN Algorithm

```
The kNN algorithm is a little bit atypical as compared to other machine learning algorithms.
As you saw earlier, each machine learning model has its specific formula that needs to be estimated.
The specificity of the k-Nearest Neighbors algorithm is that this formula is computed not at the moment of fitting but rather at the moment of prediction.
This isn’t the case for most other models.
```

```
When a new data point arrives, the kNN algorithm, as the name indicates, will start by finding the nearest neighbors of this new data point.
Then it takes the values of those neighbors and uses them as a prediction for the new data point.
```

```
As an intuitive example of why this works, think of your neighbors.
Your neighbors are often relatively similar to you. They’re probably in the same socioeconomic class as you.
Maybe they have the same type of work as you, maybe their children go to the same school as yours, and so on.
But for some tasks, this kind of approach is not as useful.
For instance, it wouldn’t make any sense to look at your neighbor’s favorite color to predict yours.
```

```
The kNN algorithm is based on the notion that you can predict the features of a data point based on the features of its neighbors. In some cases, this method of prediction may be successful, while in other cases it may not.
Next, you’ll look at the mathematical description of “nearest” for data points and the methods to combine multiple neighbors into one prediction.
```

Define “Nearest” Using a Mathematical Definition of Distance

To find the data points that are closest to the point that you need to predict, you can use a mathematical definition of distance called Euclidean distance.

To get to this definition, you should first understand what is meant by the difference of two vectors.

You must understand that your data points are actually vectors.

You can then compute the distance between them by computing the norm of the difference vector.

You can compute this in Python using linalg.norm() from NumPy.

```
In this case, you should compute the norm of the difference vector c to obtain the distance between the data points.
```

In [11]:

```python
# Example :
import numpy as np
a = np.array([2, 2])
b = np.array([4, 4])
np.linalg.norm(a - b)
```

Out[11]:

```
2.8284271247461903
```

```
In this code block, you define your data points as vectors.
You then compute norm() on the difference between two data points.
This way, you directly obtain the distance between two multidimensional points.
Even though the points are multidimensional, the distance between them is still a scalar, or a single value.
```

```
In case you want to get more details on the math, you can have a look at the Pythagorean theorem to understand how the Euclidean distance formula is derived.
```

Find the k Nearest Neighbors

Now that you have a way to compute the distance from any point to any point, you can use this to find the nearest neighbors of a point on which you want to make a prediction.

You need to find a number of neighbors, and that number is given by k.

The minimum value of k is 1. This means using only one neighbor for the prediction.

The maximum is the number of data points that you have. This means using all neighbors.

The value of k is something that the user defines.

Optimization tools can help you with this, as you'll see in the last part of this tutorial.

```
Now, to find the nearest neighbors in NumPy, go back to the Abalone Dataset.
As you've seen, you need to define distances on the vectors of the independent variables, so you should first get your pandas DataFrame into a NumPy array using the .values attribute:
```

In [12]:

```
X = abalone.drop("Rings", axis=1)
X = X.values
y = abalone["Rings"]
y = y.values
```

```
This code block generates two objects that now contain your data: X and y.
X is the independent variables and y is the dependent variable of your model.
Note that you use a capital letter for X but a lowercase letter for y.
This is often done in machine learning code because mathematical notation generally uses a capital letter for matrices and a lowercase letter for vectors.
```

In [13]:

```
# You can create the NumPy array for this data point as follows:
new_data_point = np.array([
    0.569552,
    0.446407,
    0.154437,
    1.016849,
    0.439051,
    0.222526,
    0.291208,
])
```

In [14]:

```
# The next step is to compute the distances between this new data point and each of the
distances = np.linalg.norm(X - new_data_point, axis=1)
```

In [15]:

```
print(distances)
```

```
[0.59739395 0.9518455  0.40573594 ... 0.20397872 0.14342627 1.10583307]
```

You now have a vector of distances, and you need to find out which are the three closest neighbors.

To do this, you need to find the IDs of the minimum distances.

You can use a method called .argsort() to sort the array from lowest to highest, and you can take the first k elements to obtain the indices of the k nearest neighbors:

In [16]:

```
k = 3
nearest_neighbor_ids = distances.argsort()[:k]
nearest_neighbor_ids
```

Out[16]:

```
array([4045, 1902, 1644], dtype=int64)
```

```
This tells you which three neighbors are closest to your new_data_point.
In the next paragraph, you’ll see how to convert those neighbors in an estimation.
```

Voting or Averaging of Multiple Neighbors

Having identified the indices of the three nearest neighbors of your abalone of unknown age, you now need to combine those neighbors into a prediction for your new data point.

As a first step, you need to find the ground truths for those three neighbors:

In [17]:

```
nearest_neighbor_rings = y[nearest_neighbor_ids]
nearest_neighbor_rings
```

Out[17]:

```
array([ 9, 11, 10], dtype=int64)
```

Now that you have the values for those three neighbors, you’ll combine them into a prediction for your new data point.

Combining the neighbors into a prediction works differently for regression and classification.

Average for Regression

In regression problems, the target variable is numeric.

You combine multiple neighbors into one prediction by taking the average of their values of the target variable.

You can do this as follows:

In [18]:

```python
prediction = nearest_neighbor_rings.mean()
```

```
You'll get a value of 10 for prediction. This means that the 3-Nearest Neighbor prediction for your new data point is 10.
You could do the same for any number of new abalones that you want.
```

Mode for Classification

```
In classification problems, the target variable is categorical.
As discussed before, you can't take averages on categorical variables.
For example, what would be the average of three predicted car brands? That would be impossible to say.
You can't apply an average on class predictions.
```

```
Instead, in the case of classification, you take the mode.
The mode is the value that occurs most often.
This means that you count the classes of all the neighbors, and you retain the most common class.
The prediction is the value that occurs most often among the neighbors.
```

```
If there are multiple modes, there are multiple possible solutions.
You could select a final winner randomly from the winners.
You could also make the final decision based on the distances of the neighbors, in which case the mode of the closest neighbors would be retained.
```

```
You can compute the mode using the SciPy mode() function.
As the abalone example is not a case of classification, the following code shows how you can compute the mode for a toy example:
```

In [19]:

```python
import scipy.stats
class_neighbors = np.array(["A", "B", "B", "C"])
scipy.stats.mode(class_neighbors)
```

Out[19]:

```
ModeResult(mode=array(['B'], dtype='<U1'), count=array([2]))
```

As you can see, the mode in this example is "B" because it's the value that occurs most often in the input data.

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```