NumPy, Matplotlib,and Pandas

Juan F. Imbet *Ph.D.*

Community developed libraries

NumPy

Numpy

Numpy is a library for scientific computing. It is useful for working with arrays and matrices. Most of its code is written in C, so it is very fast. Numpy is used in many scientific computing applications, including machine learning and deep learning. Numpy is imported using the `import` keyword. Numpy is usually imported using the alias `np`.

```
import numpy as np
```

Numpy arrays

Numpy arrays are used to store multiple items in a single variable. They can be created using the `np.array` function. Numpy arrays are similar to lists, but they are faster and more efficient. Numpy arrays can be created from lists, tuples, and other arrays.

```
x = np.array([1, 2, 3])
```

```
array([1, 2, 3])
```

Numpy arrays are faster than lists

```
x = list(range(1000000))
y = np.array(x)
%timeit sum(x)
%timeit np.sum(y)
```

```
4.92 ms ± 9.35 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
130 µs ± 75.3 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Why are numpy arrays faster than lists?

NumPy arrays are faster than lists because they are stored efficiently in memory, allowing for faster data access and manipulation. They also support vectorized operations, which are optimized and implemented in lower-level languages like C, making them much faster than equivalent Python loops. Additionally, NumPy takes advantage of CPU caching, optimized algorithms, and specialized functions, resulting in faster computations compared to pure Python list operations.

Use numpy for linear algebra

Most common functions

```
X = np.array([[1, 2, 3], [4, 5, 6]])
Y = np.array([[1, 2], [3, 4], [5, 6]])
x = np.array([1, 2, 3])

X.T # transpose
X@Y # matrix multiplication
X*X # element-wise multiplication
np.dot(X, Y) # matrix multiplication
np.matmul(X, Y) # matrix multiplication
np.vdot(x, x) # dot product
np.inner(x, x) # dot product
np.outer(x, x) # outer product
np.linalg.norm(x) # norm
np.linalg.det(X@Y) # determinant
np.linalg.inv(X@Y) # inverse
np.linalg.eig(X@Y) # eigenvalues and eigenvectors
np.linalg.svd(X@Y) # singular value decomposition
```

Numpy types

Numpy allows to store data in different types. This is useful when you want to save memory, or when you want to perform operations on different types of data. Data types include int32, int64, float32, float64, bool, and object. The default data type is float64.

```
x = np.array([1, 2, 3], dtype=np.int8)
x.dtype # dtype('int8')
```

matplotlib

Matplotlib

Matplotlib is a library for plotting data. It is useful for visualizing data. Matplotlib is imported using the `import` keyword. Matplotlib is usually imported using the alias `plt`.

```
import matplotlib.pyplot as plt
# or
from matplotlib import pyplot as plt
```

Matplotlib - Most common plots

- Line plot - `plt.plot`
- Scatter plot - `plt.scatter`
- Bar plot - `plt.bar`
- Histogram - `plt.hist`
- Box plot - `plt.boxplot`
- Pie chart - `plt.pie`

Matplotlib is a very powerful library. Graphs can be customized in many ways, and a complete guide would be too long.

Customizing plots

- `plt.title` - Set the title of the plot
- `plt.xlabel` - Set the label for the x-axis
- `plt.ylabel` - Set the label for the y-axis
- `plt.rotate` - Rotate the x-axis labels
- `plt.legend` - Show the legend
- `plt.show` - Show the plot

Check out the examples here

pandas

Pandas

Pandas is a library for data analysis. Probably the most used library in data science. Pandas is imported using the `import` keyword. Pandas is usually imported using the alias `pd`.

```
import pandas as pd
```

Dataframes are the main data structure in Pandas. They are used to store tabular data. They can be created using the `pd.DataFrame` function. Dataframes can be created from lists, tuples, dictionaries, and other dataframes. It is common to name dataframes `df`.

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
```

```
   a  b
0  1  4
1  2  5
2  3  6
```

Pandas - Load data from a file

Pandas can load data from CSV files, JSON files, Excel files, parquet files, and SQL databases.

```
df = pd.read_csv('file.csv')
df = pd.read_json('file.json')
df = pd.read_excel('file.xlsx')
df = pd.read_sql('SELECT * FROM table', connection)
df = pd.read_parquet('file.parquet')
```

Pandas - miscellaneous

Pandas can load data from a URL.

```
df = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv')
```

Pandas and NumPy are closely related. Pandas is built on top of NumPy, and it uses NumPy arrays to store data. Pandas can convert dataframes to NumPy arrays using the `to_numpy` method.

```
df.to_numpy()
```

Dataframe indexing

Dataframes can be indexed using the `iloc` and `loc` methods. The `iloc` method is used to index dataframes by position. The `loc` method is used to index dataframes by label. Dataframes can also be indexed using the `[]` operator. The `[]` operator is used to index dataframes by label.

Using `iloc`

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
df.iloc[0] # First row
df.iloc[0, 0] # First element of first row
df.iloc[:, 0] # First column
df.iloc[0:2] # First two rows
df.iloc[0:2, 0:2] # First two rows and columns
df.iloc[[0, 2]] # First and third row
df.iloc[[0, 2], [0, 2]] # First and third row and column
df.iloc[lambda x: x.index % 2 == 0] # Even rows
df.iloc[lambda x: x.index % 2 == 0, lambda x: x.columns % 2 == 0] # Even rows and columns
```

Dataframe indexing (2)

Using `loc`

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
df.loc[0] # First row
df.loc[0, 'a'] # First element of first row
df.loc[:, 'a'] # First column
df.loc[0:2] # First two rows
df.loc[0:2, 'a':'b'] # First two rows and columns
df.loc[[0, 2]] # First and third row
df.loc[[0, 2], ['a', 'b']] # First and third row and column
df.loc[lambda x: x.index % 2 == 0] # Even rows
df.loc[lambda x: x.index % 2 == 0, lambda x: x.columns % 2 == 0] # Even rows and columns
```

Data operations

Dataframes can be modified using the `assign` method. The `assign` method is used to modify dataframes by adding new columns. Dataframes can also be modified using the `[]` operator. The `[]` operator is used to modify dataframes by adding new columns.

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
df['c'] = df.a + df.b # Add a new column
df['c'] = df['a'] + df['b'] # Add a new column
```

Columns can be accessed using the `[]` operator. The `[]` operator is used to access columns by label. Columns can also be accessed using the `.` operator. The `.` operator is used to access columns by label. The `.` operator cannot be used to describe a column that does not exist, and cannot be used to describe a column with a reserved name. E.g. `df.dtype` is not a valid column.

Inplace vs not inplace

Dataframes can be modified inplace or not inplace. The `inplace` parameter is used to specify whether or not the dataframe should be modified inplace. The `inplace` parameter has two possible values: `True` and `False`. For efficiency reasons, it is recommended to not modify dataframes inplace.

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
df.rename(columns={'a': 'A'}, inplace=True) # Modify inplace
df = df.rename(columns={'a': 'A'}) # Modify not inplace
```

The `groupby` operation produces a `DataFrameGroupBy` object

```
df = pd.DataFrame({'group': ['A', 'A', 'B', 'B'], 'value': [1, 2, 3, 4]})
df.groupby('group')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f8e1c0b5d90>
```

`apply` and custom functions

The `apply` method is used to apply a function to a dataframe. The `apply` method has two arguments: a function, and an axis. The function is applied to each row or column of the dataframe. The axis is used to specify whether the function should be applied to each row or column. The axis can be `0` for rows, or `1` for columns. The default axis is `0`.

```
df = pd.DataFrame({'group': ['A', 'A', 'B', 'B'], 'value': [1, 2, 3, 4]})
df.groupby('group').apply(lambda x: x.value.sum())
# or
df.groupby('group')['value'].apply(lambda x: x.sum())
# or
df.groupby('group').value.sum()
```

```
group
A    3
B    7
```

Homework (Optional, do not use GPT)

In this homework you will process and analyze a dataset of the Titanic passengers. The dataset is available here. The dataset contains the following columns:

- `survived` - Whether the passenger survived (0 = No, 1 = Yes)
- `pclass` - The class of the passenger (1 = 1st, 2 = 2nd, 3 = 3rd)
- `sex` - The
- `age` - The age of the passenger
- `sibsp` - The number of siblings or spouses the passenger had aboard the Titanic

- `parch` - The number of parents or children the passenger had aboard the Titanic
- `fare` - The fare the passenger paid
- `embarked` - The port the passenger embarked from (C = Cherbourg, Q = Queenstown, S = Southampton)
- `class` - The class of the passenger
- `who` - (man, woman, child).
- `adult_male` - Whether the passenger is an adult male.
- `deck` - The deck the passenger was on.
- `embark_town` - The town the passenger embarked from.
- `alive` - Whether the passenger survived (yes, no).
- `alone` - Whether the passenger was alone.

Questions

- Estimate the probability of survival for a passenger in each class. (1, 2, 3). The probability of survival is the number of survivors divided by the total number of passengers in each class.
- How does this probability change if the passenger was a man, a woman, a male child, or a female child?
- Determine if the event of *not* surviving given that the passenger was an adult male is independent of the class of the passenger. Recall that two events are independent if the probability of one event occurring does not affect the probability of the other event occurring.

$$P(A|B) = P(A)$$

Questions (2)

- Estimate the average age of the passengers who survived and the passengers who did not survive.
- Estimate the average fare paid by the passengers who survived and the passengers who did not survive.
- Based on the documentation of pandas and matplotlib, create a plot that shows the relatiin between the fare paid and the age of the passengers.
- How does this relationship change across the different classes?