Lecture VII - NumPy and Pandas for Scientific Computing

Programming with Python

Dr. Tobias Vlćek

Quick Recap of the last Lecture

Modules

- Modules are `.py` files containing Python code
- They are used to organize and reuse code
- They can define functions, classes, and variables
- Can be imported into other scripts

. . .

> **Tip**
>
> We can import entire modules or individual functions, classes or variables.

Standard Libraries

- Python includes many built-in modules like:
  - `random` provides functions for random numbers
  - `os` allows interaction with the operating system
  - `csv` is used for reading and writing CSV files
  - `re` is used for working with regular expressions

Packages

- Packages are collections of modules
- Often available from the Python Package Index (PyPI)
- Install using `pip install <package_name>`
- Virtual environments help manage dependencies

. . .

> **Tip**
>
> Virtual environments are not that important for you right now, as they are mostly used if you work on several projects with different dependecies at once.

NumPy Module

What is NumPy?

- **NumPy** is a package for scientific computing in Python
- Provides large, multi-dimensional arrays and matrices
- Wide range of functions to operate on these
- Python lists can be slow - Numpy arrays are much faster

. . .

> **Note**
>
> The name of the package comes from Numerical Python.

Why is NumPy so fast?

- Arrays are stored in a contiguous block of memory
- This allows for efficient memory access patterns
- Operations are implemented in the languages `C` and `C++`

. . .

Question: Have you heard of C and C++?

How to get started

1. Install NumPy using `pip install numpy`
2. In Thonny, `Tools -> Manage Packages...`
3. Import NumPy in a script using `import numpy as np`

. . .

```python
import numpy as np
x = np.array([1, 2, 3, 4, 5]); type(x)
```

```
numpy.ndarray
```

. . .

> **Note**
>
> You don't have to use `as np`. But it is a common practice to do so.

Creating Arrays

- The backbone of Numpy is the so called `ndarray`
- Can be initialized from different data structures:

```
import numpy as np

array_from_list = np.array([1, 1, 1, 1])
print(array_from_list)
```

```
[1 1 1 1]
```

```
import numpy as np

array_from_tuple = np.array((2, 2, 2, 2))
print(array_from_tuple)
```

```
[2 2 2 2]
```

Hetergenous Data Types

- It is possible to store different data types in a `ndarray`

```
import numpy as np

array_different_types = np.array(["s", 2, 2.0, "i"])
print(array_different_types)
```

```
['s' '2' '2.0' 'i']
```

. . .

> **Note**
>
> But it is mostly not recommended, as it can lead to performance issues. If possible, try to **keep the types homogenous**.

Prefilled Arrays

Improve performance by **allocating memory upfront**

- `np.zeros(shape)`: to create an array of zeros
- `np.random.rand(shape)`: array of random values
- `np.arange(start, stop, step)`: evenly spaced
- `np.linspace(start, stop, num)`: evenly spaced

. . .

> **Note**
>
> The shape refers to the size of the array. It can have one or multiple dimensions.

Dimensions

- The shape is specified as tuple in these arrays
- `(2)` or `2` creates a 1-dimensional array (vetor)
- `(2,2)` creates a 2-dimensional array (matrix)
- `(2,2,2)` 3-dimensional array (3rd order tensor)
- `(2,2,2,2)` 4-dimensional array (4th order tensor)
- …

Computations

- We can apply operations to the entire array at once
- This is much faster than applying them element-wise

. . .

```python
import numpy as np
x = np.array([1, 2, 3, 4, 5])
x + 1
```

```
array([2, 3, 4, 5, 6])
```

Arrays in Action

Task: Practice working with Numpy:

```python
# TODO: Create a 3-dimensional tensor with filled with zeros
# Choose the shape of the tensor, but it should have 200 elements
# Add the number 5 to all values of the tensor

# Your code here
assert sum(tensor) == 1000

# TODO: Print the shape of the tensor using the method shape()
# TODO: Print the dtype of the tensor using the method dtype()
# TODO: Print the size of the tensor using the method size()
```

Indexing and Slicing

- Accessing and slicing `ndarray` works as before
- Higher dimension element access with multiple indices

. . .

Question: What do you expect will be printed?

```python
import numpy as np
x = np.random.randint(0, 10, size=(3, 3))
print(x); print("---")
print(x[0:2,0:2])
```

```
[[5 2 4]
 [9 8 8]
```

```
 [0 5 4]]
---
[[5 2]
 [9 8]]
```

Data Types

- Numpy provides data types as characters
- `i`: integer
- `b`: boolean
- `f`: float
- `S`: string
- `U`: unicode

. . .

```python
string_array = np.array(["Hello", "World"]); string_array.dtype
```

```
dtype('<U5')
```

Enforcing Data Types

- We can also **provide** the type when creating arrays

. . .

```python
x = np.array([1, 2, 3, 4, 5],  dtype = 'f'); print(x.dtype)
```

```
float32
```

. . .

- Or we can **change** them for existing arrays

```python
x = np.array([1, 2, 3, 4, 5],  dtype = 'f'); print(x.astype('i').dtype)
```

```
int32
```

. . .

> **Note**
>
> Note, how the types are specified as `int32` and `float32`.

Sidenote: Bits

Question: Do you have an idea what `32` stands for?

. . .

- It's the number of bits used to represent a number
  - `int16` is a 16-bit integer
  - `float32` is a 32-bit floating point number
  - `int64` is a 64-bit integer
  - `float128` is a 128-bit floating point number

Why do Bits Matter?

- They matter, because they can affect:
  - the performance of your code
  - the precision of your results

. . .

- That's why numbers can have a limited precision!
  - An `int8` has to be in the range of -128 to 127
  - An `int16` has to be in the range of -32768 to 32767

. . .

Question: Size difference between `int16` and `int64`?

Joining Arrays

- You can use `concatenate` two **join arrays**
- With `axis` you can specify the dimension
- In 2-dimensions `hstack()` and `vstack()` are easier

. . .

Question: What do you expect will be printed?

```
import numpy as np
ones = np.array((1,1,1,1))
twos = np.array((1,1,1,1)) *2
print(np.vstack((ones,twos))); print(np.hstack((ones,twos)))
```

```
[[1 1 1 1]
 [2 2 2 2]]
[1 1 1 1 2 2 2 2]
```

Common Methods

- `sort()`: sort the array from low to high
- `reshape()`: reshape the array into a new shape
- `flatten()`: flatten the array into a 1D array
- `squeeze()`: squeeze the array to remove 1D entries
- `transpose()`: transpose the array

. . .

> Tip
>
> Try experiment with these methods, they can make your work much easier.

Speed Differences in Action

Task: Complete the following task to practice with Numpy:

```python
# TODO: Create a 2-dimensional matrix with filled with ones of size 1000 x 1000.
# Afterward, flatten the matrix to a vector and loop over the vector.
# In each loop iteration, add a random number between 1 and 10000.
# TODO: Now, do the same with a list of the same size and fill it with random numbers.
# Then, sort the list as you have done with the Numpy vector before.
# You can use the `time` module to compare the runtime of both approaches.
import time
start = time.time()
# Your code here
end = time.time()
print(end - start) # time in seconds
```

```
9.059906005859375e-06
```

Pandas Module

What is Pandas?

- Pandas is a data manipulation and analysis library
- It provides data structures like **DataFrames and Series**
- Tools for data cleaning, analysis, and visualization
- It can also be used to work with Excel files!

How to install Pandas

- In the last lecture, we have installed it with `pip install pandas` or with Thonny
- Now, import the package `import pandas as pd`

. . .

> **Note**
>
> You can also use a different abbreviation, but `pd` is the most common one.

Creating DataFrames

- DataFrames behave quite similar to Numpy arrays
- But they have row and column labels

. . .

```python
import pandas as pd
df = pd.DataFrame({ # DataFrame is created from a dictionary
    "Name": ["Tobias", "Robin", "Nils", "Nikolai"],
    "Kids": [2, 1, 0, 0],
    "City": ["Oststeinbek", "Oststeinbek", "Hamburg", "Lübeck"],
    "Salary": [3000, 3200, 4000, 2500]}); print(df)
```

```
      Name  Kids         City  Salary
0   Tobias     2  Oststeinbek    3000
1    Robin     1  Oststeinbek    3200
2     Nils     0      Hamburg    4000
3  Nikolai     0       Lübeck    2500
```

Reading from CSV Files

```
df = pd.read_csv("employees.csv") # Reads the CSV file
print(df)
```

```
       Name  Age Department        Position  Salary
0     Alice   30         HR         Manager   50000
1       Bob   25         IT       Developer   60000
2   Charlie   28    Finance         Analyst   55000
3     David   35  Marketing       Executive   52000
4       Eve   32      Sales  Representative   48000
5     Frank   29         IT       Developer   61000
6     Grace   31         HR       Assistant   45000
7      Hank   27    Finance         Analyst   53000
8       Ivy   33  Marketing         Manager   58000
9      Jack   26      Sales  Representative   47000
10     Kara   34         IT       Developer   62000
11      Leo   30         HR         Manager   51000
12     Mona   28    Finance         Analyst   54000
13     Nina   35  Marketing       Executive   53000
14    Oscar   32      Sales  Representative   49000
15     Paul   29         IT       Developer   63000
16    Quinn   31         HR       Assistant   46000
17     Rita   27    Finance         Analyst   52000
18      Sam   33  Marketing         Manager   59000
19     Tina   26      Sales  Representative   48000
20      Uma   34         IT       Developer   64000
21    Vince   30         HR         Manager   52000
22     Walt   28    Finance         Analyst   55000
23     Xena   35  Marketing       Executive   54000
24     Yara   32      Sales  Representative   50000
25     Zane   29         IT       Developer   65000
26     Anna   31         HR       Assistant   47000
27      Ben   27    Finance         Analyst   53000
28    Cathy   33  Marketing         Manager   60000
29    Dylan   26      Sales  Representative   49000
30     Ella   34         IT       Developer   66000
31     Finn   30         HR         Manager   53000
32     Gina   28    Finance         Analyst   56000
33     Hugo   35  Marketing       Executive   55000
34     Iris   32      Sales  Representative   51000
35     Jake   29         IT       Developer   67000
36     Kyla   31         HR       Assistant   48000
37     Liam   27    Finance         Analyst   54000
38      Mia   33  Marketing         Manager   61000
39     Noah   26      Sales  Representative   50000
40    Olive   34         IT       Developer   68000
41     Pete   30         HR         Manager   54000
42   Quincy   28    Finance         Analyst   57000
43     Rose   35  Marketing       Executive   56000
44    Steve   32      Sales  Representative   52000
45     Tara   29         IT       Developer   69000
46     Umar   31         HR       Assistant   49000
```

```
47      Vera    27    Finance          Analyst    55000
48      Will    33  Marketing          Manager    62000
49      Zara    26      Sales   Representative    51000
```

Basic Operations

- Use the `df.head()` method to display the first 5 rows
- Use the `df.tail()` method to display the last 5 rows

...

```
df = pd.read_csv("employees.csv")
print(df.tail())
```

```
    Name  Age Department        Position  Salary
45  Tara   29         IT       Developer   69000
46  Umar   31         HR       Assistant   49000
47  Vera   27    Finance         Analyst   55000
48  Will   33  Marketing         Manager   62000
49  Zara   26      Sales  Representative   51000
```

Information about the DataFrame

- Use `df.info()` to display information about a DataFrame

...

```
df = pd.read_csv("employees.csv")
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50 entries, 0 to 49
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Name        50 non-null     object
 1   Age         50 non-null     int64
 2   Department  50 non-null     object
 3   Position    50 non-null     object
 4   Salary      50 non-null     int64
dtypes: int64(2), object(3)
memory usage: 2.1+ KB
None
```

Statistics about a DataFrame

- Use `df.describe()` to display summary statistics
- Use the `df.index` **attribute** to access the **index**

...

```
df = pd.read_csv("employees.csv")
print(df.describe())
```

```
             Age        Salary
count  50.000000     50.000000
mean   30.320000  54980.000000
std     2.958488   6175.957333
min    25.000000  45000.000000
25%    28.000000  50250.000000
50%    30.000000  54000.000000
75%    33.000000  59750.000000
max    35.000000  69000.000000
```

Filtering DataFrames

- Use `df['column_name']` to access a column
- Use the `df[df['column'] > value]` method to filter

...

```
df = pd.read_csv("employees.csv")
df_high_salary = df[df['Salary'] >= 67000]
print(df_high_salary)
print(df_high_salary.iloc[2]["Name"]) #Access the third row and the "Name" column
print(df_high_salary.loc[40]["Name"]) #Access the label 40 and the "Name" column
```

```
     Name  Age Department   Position  Salary
35   Jake   29         IT  Developer   67000
40  Olive   34         IT  Developer   68000
45   Tara   29         IT  Developer   69000
Tara
Olive
```

Filtering in Action

Task: Complete the following task:

```
# TODO: Load the employees.csv located in the git repository into a DataFrame
# First, filter the DataFrame for employees with a manager position
# Then, print the average salary of the remaining employees
# Finally, print the name of the employee with the lowest salary
```

...

> **Note**
>
> Note, that we can use the `mean()` method on the `Salary` column, as it is a numeric column. In addition, we can use the `min()` method on the `Salary` column to find the lowest salary.

Grouping DataFrames

Grouping

- Grouping is a powerful feature of Pandas
- Groups data by one or more columns
- And then perform operations
- Syntax is `df.groupby('column').method()`

. . .

```
df = pd.read_csv("employees.csv")
df = df.drop(columns=["Name", "Department"])
df.groupby(['Position']).mean() # Mean per position
```

	Age	Salary
Position		
Analyst	27.5	54400.0
Assistant	31.0	47000.0
Developer	30.6	64500.0
Executive	35.0	54000.0
Manager	31.5	56000.0
Representative	29.0	49500.0

Grouping by Multiple Columns

- Group by multiple columns `['column1', 'column2']`
- You can use lists or tuples to specify multiple columns

. . .

```
df = pd.read_csv("employees.csv")
df = df.drop(columns=["Name"])
# Max per position and department
df.groupby(['Position', "Department"]).max()
```

Position	Department
Analyst	Finance
Assistant	HR
Developer	IT
Executive	Marketing

Position	Department
Manager	HR Marketing
Representative	Sales

Grouping with Aggregations

- As seen, we can use aggregation functions:
  - `sum()`: sum of the values
  - `mean()`: mean of the values
  - `max()`: maximum of the values
  - `min()`: minimum of the values
  - `count()`: count of the values

Melting DataFrames

- Use `pd.melt()` to transform from wide to long

...

```
df = pd.read_csv("employees.csv").drop(columns=["Name"])
df = pd.melt(df, id_vars=['Position'])
print(df.head()); print(df.tail())
```

```
         Position variable value
0         Manager      Age    30
1       Developer      Age    25
2         Analyst      Age    28
3       Executive      Age    35
4  Representative      Age    32
           Position variable  value
145       Developer   Salary  69000
146       Assistant   Salary  49000
147         Analyst   Salary  55000
148         Manager   Salary  62000
149  Representative   Salary  51000
```

Pandas in Action

Task: Complete the following task:

```
# TODO: Load the employees.csv again into a DataFrame
# First, group by the "Position" column and count the employees per position
# Then, group by the "Department" column and calculate the sum of all other columns per
↪  department
df = pd.read_csv("employees.csv")
# Your code here
```

...

> **Note**
>
> Do you notice any irregularities while calculating the sum per department?

Concatenating DataFrames

- `pd.concat()` to concatenate along shared columns

```
df1 = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
df2 = pd.DataFrame({"A": [7, 8, 9], "B": [10, 11, 12]})
df = pd.concat([df1, df2])
print(df)
```

```
   A   B
0  1   4
1  2   5
2  3   6
0  7  10
1  8  11
2  9  12
```

Joining DataFrames

- Use `pd.join()` to join DataFrames along columns
- Joining is done on the index by default!

```
df1 = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]}, index=['x', 'y', 'z'])
df2 = pd.DataFrame({"C": [7, 8, 9], "D": [10, 11, 12]}, index=['z', 'y', 'w'])
df = df1.join(df2)
print(df)
```

```
   A  B    C     D
x  1  4  NaN   NaN
y  2  5  8.0  11.0
z  3  6  7.0  10.0
```

Merging DataFrames on Columns

- `pd.merge(df_name, on='column', how='type')`
- merge DataFrames along shared columns
- `how` specifies the type of merge
  - `inner`: rows with matching keys in both DataFrames
  - `outer`: rows from both are kept, missing values are filled
  - `left`: rows from the left are kept, missing values are filled
  - `right`: rows from right are kept, missing values are filled

Outer Merge

```
df3 = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
df4 = pd.DataFrame({"A": [2, 3, 4], "C": [7, 8, 9]})
df_merged = df3.merge(df4, on="A", how="outer")
print(df_merged)
```

```
   A    B    C
0  1  4.0  NaN
1  2  5.0  7.0
2  3  6.0  8.0
3  4  NaN  9.0
```

Working with Excel Files

Working with Excel Files

Reading Excel Files

- Read using the `pd.read_excel(file_path)` function
- Write using the `df.to_excel(file_path)` method

...

```python
import pandas as pd
df = pd.read_csv("employees.csv")
df.to_excel("employees.xlsx", index=False)
```

...

> **Note**
>
> Note, that you likely need to install the `openpyxl` package to be able to write Excel files, as it handles the file format.

Advanced Excel file handling

```python
df = pd.read_excel("employees.xlsx")

# Writes to the Employees sheet and does not include row indices
df.to_excel("employees.xlsx", sheet_name="Employees", index=False)

# Reads from the Employees sheet
df = pd.read_excel("employees.xlsx", sheet_name="Employees")
```

...

> **Note**
>
> **And that's it for todays lecture!**
> You now have the basic knowledge to start working with scientific computing. Don't worry that we haven't applied Excel files yet, we will do so in the upcoming tutorial.

Literature

Interesting Books

- Downey, A. B. (2024). Think Python: How to think like a computer scientist (Third edition). O'Reilly. Link to free online version
- Elter, S. (2021). Schrödinger programmiert Python: Das etwas andere Fachbuch (1. Auflage). Rheinwerk Verlag.

. . .

For more interesting literature to learn more about Python, take a look at the literature list of this course.