Intro to deep learning

Dr. Janoś Gabler, University of Bonn

Lecture 2: Python basics, numpy, pandas, Markdown

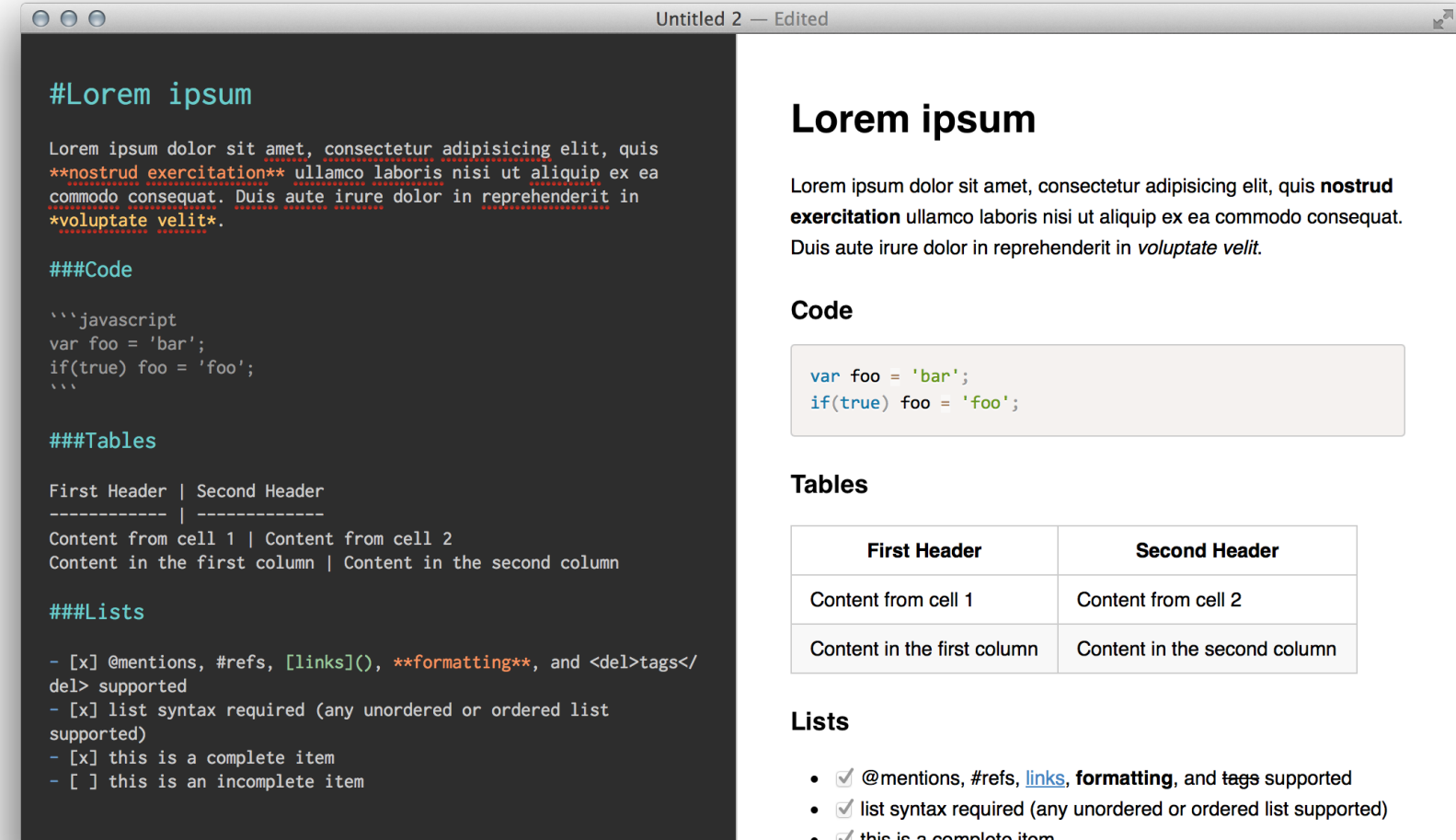
Housekeeping

- I updated the windows environment

Topics

- Questions (15 min)
- Markdown (15 min)
- List and dict comprehensions (15 min)
- Numpy (60 min)
- File Paths (15 min)
- Pandas (45 min)
- Errors, Trackebacks and how to ask for help (15 min)

Markdown


Untitled 2 — Edited
#Lorem ipsum
Lorem ipsum dolor sit amet, consectetur adipisicing elit, quis **nostrud exercitation** ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in *voluptate velit*.
###Code
```javascript
var foo = 'bar';
if(true) foo = 'foo';
```
###Tables
First Header	Second Header
Content from cell 1	Content from cell 2
Content in the first column	Content in the second column
###Lists
- [x] @mentions, #refs, [links](), **formatting**, and ~~tags~~ supported
- [x] list syntax required (any unordered or ordered list supported)
- [x] this is a complete item
- [ ] this is an incomplete item
Lorem ipsum
Lorem ipsum dolor sit amet, consectetur adipisicing elit, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit.
Code
var foo = 'bar';
if(true) foo = 'foo';
Tables
First Header
Second Header
Content from cell 1
Content from cell 2
Content in the first column
Content in the second column
Lists
@mentions, #refs, links, formatting, and tags supported
list syntax required (any unordered or ordered list supported)
this is a complete item

Where will we use it

- To write well formatted zulip messages
- For GitHub Issues
- In the `README.md` file

Syntax (Headings)

```
# The largest heading
## The second largest heading
###### The smallest heading
```

The Title

The second largest heading

THE SMALLEST HEADING

Styling text

```
**bold**
```

bold

```
*italic*
```

italic

```
~~strikethrough~~
```

~~strikethrough~~

```
***bold and italic***
```

bold and italic

Lists

```
- This is
- a
- list
```

- This is
- a
- list

```
1. This is
2. an
3. enumeration
```

1. This is
2. an
3. enumeration

Links and Quotes

```
[This will be a link](www.google.com)

> This will be a
> multi-line quote
```

This will be a link

> This will be a multi-line quote

Code Snippets

This is the most important one for communication on zulip!!!

````
`x = 10`
````

Single line/inline code:

`x = 10`

Multi line code

```python
def f(x):
   return np.ones_like(x)
```

````
```python
def f(x):
   return np.ones_like(x)
```
````

Task (5 min)

Write a nicely formatted zulip message to one of your classmates. Include a link, a multi-line code snippet, a bold word and other elements.

Comprehensions

Common loop patterns

Apply calculations to lists

```
>>> squares = []
>>> for i in [1, 2, 3, 4, 5]:
...     squares.append(i ** 2)
>>> squares
[1, 4, 9, 16, 25]
```

Filter lists

```
>>> even = []
>>> for i in range(10):
...   if i % 2 == 0:
...       even.append(i)
[0, 2, 4, 6, 8]
```

- Both initialize and append
- At least three lines
- Can lead to deep indentation

List comprehensions

Math notation

$$\{x^2 | x \in \{1, 2, 3, 4, 5\}\}$$
$$\{x | x \in \{1, ..., 10\}, x \mod 2 = 0\}$$

List comprehension

```
>>> [i ** 2 for i in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]
```

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

- Could include an else statement
- Can call arbitrary functions instead of squaring
- More readable than loops as long as it fits on one line!
- Not much faster than loops

Dict comprehension

```
>>> {i: i ** 2 for i in [1, 2, 3, 4, 5]}
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
>>> skills = {
...     "Janos": 8,
...     "Tobi": 10,
...     "Mariam": 9,
... }
>>> {k: v for k, v in skills.items() if v >= 9}
{"Tobi": 10, "Mariam": 9}
```

- Inside a dict comprehension you can loop over lists or dicts
- More readable than loops if it fits on one line
- We will use dictionaries and dictionary comprehensions a lot when work with multiple models and want to compare them

Task 1 (5 min)

Imports and Libraries

Third party libraries

- Python is a general purpose programming language
- The base language you have seen so far is extended by labraries
  - Standard library (e.g. pathlib, functools)
  - Third party libraries (e.g. numpy, pandas, pytorch)
- Libraries need to be imported to use them
- Third party libraries need to be installed

Different ways to import

Import one function / object

```
from numpy import array
```

Import an entire library

```
import numpy
```

Import entire library and rename it

```
import numpy as np
```

Import everything from a library

```
from numpy import *
```

- Use single import if you need one thing
- Use library import if you need many functions
- Use shorthand if there is a convention, e.g. numpy (np), pandas (pd), seaborn (sns)
- Never ever use `` `import *` ``

ModuleNotFoundError

```
>>> from numpai import array
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[32], line 1
----> 1 from nampai import array

ModuleNotFoundError: No module named 'nampai'
```

- Meaning: The library you asked for is not found
- Do you have a typo in the library name?
- Is the library installed in your environment?
- **Is the correct environment activated?**

ImportError

```
from numpy import arrrrray
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[33], line 1
----> 1 from numpy import arrrrray

ImportError: cannot import name 'arrrrray' from 'numpy' (/home/janos/miniconda3/
envs/dl_intro/lib/python3.11/site-packages/numpy/__init__.py)
```

- Something went wrong during import
- Do you have typos in what you want to import?
- Is the correct version of the library installed?

numpy

What is numpy

- Library providing:
  - Multidimensional arrays
  - Fast elementwise calculations
  - Fast linear algebra
- The reason we use Python
- Knowing numpy helps you to learn everything else (jax, pytorch, ...)

Review Article | Open Access | Published: 16 September 2020

Array programming with NumPy

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, ... Travis E. Oliphant + Show authors

Nature **585**, 357–362 (2020) | Cite this article

Abstract

Array programming provides a powerful, compact and expressive syntax for accessing, manipulating and operating on data in vectors, matrices and higher-dimensional arrays. NumPy is the primary array programming library for the Python language. It has an essential role in research analysis pipelines in fields as diverse as physics, chemistry, astronomy, geoscience, biology, psychology, materials science, engineering, finance and economics. For example, in astronomy, NumPy was an important part of the software stack used in the discovery of gravitational waves[1] and in the first imaging of a black hole[2]. Here we review how a few fundamental array concepts lead to a simple and powerful programming paradigm for organizing, exploring and analysing scientific data. NumPy is the foundation upon which the scientific Python ecosystem is constructed. It is so pervasive that several projects, targeting audiences with specialized needs, have developed their own NumPy-like interfaces and array objects. Owing to its central position in the ecosystem, NumPy increasingly acts as an interoperability layer between such array computation libraries and, together with its application programming interface (API), provides a flexible framework to support the next decade of scientific and industrial

Download PDF

Sections | Figures | References

Abstract
Main
NumPy arrays
Scientific Python ecosystem
Array proliferation and interoperability
Discussion
References
Acknowledgements
Author information
Ethics declarations
Additional information
Supplementary information
Rights and permissions
About this article
This article is cited by
Comments

Creating arrays from lists

Flat list

```
>>> np.array([1, 2, 3, 4])
array([1, 2, 3, 4])
```

Nested list

```
>>> np.array([[1, 2], [3, 4])
array([[1. , 2. ],
       [3, 4. ]])
```

Mixed dtypes in list

```
>>> np.array([[1, 2], [3.1, 4])
array([[1. , 2. ],
       [3.1, 4. ]])
```

- Can use flat or (deeply) nested lists
- Lists length cannot be rugged
- List might have mixed dtypes, arrays will not!

Constructors

Ones in different shapes

```
>>> np.ones(3)
array([1., 1., 1.])
```

```
>>> np.ones((2, 2))
array([[1., 1.],
       [1., 1.]])
```

Ranges

```
>>> np.arange(3)
array([0, 1, 2])
```

Identity matrix

```
>>> np.eye(2)
array([[1., 0.],
       [0., 1.]])
```

- `np.ones_like`
- `np.zeros`
- `np.empty`
- `np.linspace`
- `np.full`
- There are many array creation routines (docs)
- Learn them like vocabulary!

Reshaping

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> b = a.reshape((2, 3))
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> b.shape
(2, 3)
```

- Reshape can transform arrays from one shape to another
- Shapes are denoted as in matrices, i.e. (`n_rows`, `n_cols`)
- Number of elements must not change
- Elements aranged in row-major order

Task 2

(8 min)

Indexing 1d

```
>>> a = np.array([0, 1, 2, 3, 4])
>>> a[2]
2
```

```
>>> a[-1]
4
```

```
>>> a[1: 3]
array([1, 2])
```

```
>>> a[2:]
array([2, 3, 4])
```

```
>>> a[:2]
array([0, 1])
```

```
>>> a[[0, 3]]
array([0, 3])
```

```
a[[True, False, True, False, True]]
array([0, 2, 4])
```

- Indexing with square brackets
- Indexing starts at 0
- Single elements are not returned as array
- Slices include lower bound and exclude upper bound
- Lower or upper bound can be left out
- Last example: boolean indexing

Indexing 2d

select a row

```
>>> b = np.arange(12).reshape(4, 3)
>>> b[2]
array([6, 7, 8])
```

select a column

```
>>> b[:, 2]
array([ 2,  5,  8, 11])
```

select an element

```
>>> b[1, 2]
5
```

select a slice

```
>>> b[:2, :2]
array([[0, 1],
       [3, 4]])
```

- Indexing in multiple dimensions is just the same as in one!
- Separate the indexing for the dimensions by commas
- Leave out the `:` for later dimensions if you do not want to restrict them

Task 3

(8 min)

Two types of multiplication

```
>>> a = np.arange(4).reshape(2, 2)
>>> b = np.linspace(0.1, 0.4, 4).reshape(2, 2)
>>> b
array([[0.1, 0.2],
       [0.3, 0.4]])
```

```
>>> a * b
array([[0. , 0.2],
       [0.6, 1.2]])
```

```
>>> a @ b
array([[0.3, 0.4],
       [1.1, 1.6]])
```

```
>>> a.dot(b)
array([[0.3, 0.4],
       [1.1, 1.6]])
```

- `*` means elementwise multiplication
- `@` and `.dot` mean matrix multiplication
- Both generalize to high dimensions
- Both have different requirements on array shapes

Other operations between arrays

```
>>> a = np.arange(3)
>>> b = np.ones(3)
>>> a + b
array([1., 2., 3.])
```

```
>>> a - b
array([-1.,  0.,  1.])
```

```
>>> a / b
array([0., 1., 2.])
```

```
>>> b / a
array([inf, 1. , 0.5])
```

```
>>> a ** 2
array([0, 1, 4])
```

- Addition, subtraction and division work as expected
- Exponentiation uses `**`
- Division by zero does not raise an error

Mathematical functions

```
>>> a = np.array([1, 1.5, 2])
>>> np.exp(a)
array([2.71828183, 4.48168907, 7.3890561 ])
```

```
>>> np.log(a)
array([0.        , 0.40546511, 0.69314718])
```

```
>>> np.sqrt(a)
array([1.        , 1.22474487, 1.41421356])
```

```
>>> np.sin(a)
array([0.84147098, 0.99749499, 0.90929743])
```

- Numpy functions usually apply elementwise
- Faster and more readable than looping
- For more functions see the docs

Reductions

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a.mean()
2.5
```

```
>>> a.prod()
24
```

```
>>> a.sum()
10
```

```
>>> a.std()
1.118033988749895
```

```
>>> a.sum(axis=1)
array([3, 7])
```

- Reductions take an array of numbers and reduce it to fewer numbers
- Again, faster and more readable than loops
- All reductions support axis arguments

Task 4

(8 min)

File Paths

File path rules

- There are many ways to work with file paths in Python
- Some are not portable
- Rules for this class:
  1. Always use pathlib Paths instead of strings
  2. Do not hardcode Paths outside of shared folders
  3. Always concatenate paths wtih `/`
- If you copy paste a path from your windows file explorer, all three rules are violated!

Pathlib

```
dl_student/
  exercises/
    exercise_1.ipynb
    exercise_1_solution.ipynb
    exercise_2.ipynb
    exercise_2_solution.ipynb
    data.csv
```

- `dl_student` and `exercises` are directories, rest are files
- Assume you are in `exercise_2.ipynb`

```
>>> from pathlib import Path
>>> EXERCISE_DIR = Path().resolve()
>>> EXERCISE_DIR
PosixPath('/home/janos/dl_student/exercises')
```

```
>>> DATA_PATH = EXERCISE_DIR / "data.csv"
>>> DATA_PATH
PosixPath('/home/janos/dl_student/exercises/data.csv')
```

```
>>> ROOT = EXERCISE_DIR.parent
>>> ROOT
PosixPath('/home/janos/dl_student/)
```

```
>>> DATA_PATH.exists()
True
```

What can you assume about paths

- If you do not collaborate, whatever you want
- If you collaborate with someone, there is a notion of a "shared folder"
  - Dropbox folder
  - Git repository
  - ...
- You can make assumptions on the structure inside this folder
- You cannot assume, where this folder is put

Task 5

5 min

Pandas

What is pandas

- Library that implements
  - DataFrames: A dataset with multiple columns and rows
  - Series: Single columns of a DataFrame
  - Functions to manipulate Series and DataFrames
- Used in academia and industry and required for many jobs
- DataFrames are not only useful for empirical datasets
- There can be multiple datasets defined at the same time (for the stata users)

Create data from scratch

```
>>> df = pd.DataFrame(
>>>   data=np.arange(6).reshape(3, 2),
>>>   columns=["a", "b"],
>>> )
>>> print(df)
   a  b
0  0  1
1  2  3
2  4  5
```

```
>>> sr = pd.Series([11, 12, 10], index=[1, 2, 0])
>>> sr
1    11
2    12
0    10
dtype: int64
```

```
>>> df["c"] = sr
>>> print(df)
   a  b   c
0  0  1  10
1  2  3  11
2  4  5  12
```

- DataFrames have:
  - data
  - columns
  - index
- Can hold any type of data
- The index is aligned when assigning a new column

Loading and saving data

```
>>> df.to_csv("data.csv")
>>> pd.read_csv("data.csv")
>>> df.to_stata("data.dta")
>>> pd.read_stata("data.dta")
>>> df.to_pickle("data.pkl")
>>> pd.read_pickle("data.pkl")
>>> df.to_parquet("data.parquet")
>>> pd.read_parquet("data.parquet")
```

csv

- human readable
- compatible with any language
- brittle

dta

- can be exchanged with stata

pkl

- Good for intermediate results
- Lossless and fast

parquet

- Good dataset for long term storage
- Compatible with many languages

Task 6

8 minutes

Accessing columns

List the columns

```
>>> df.columns
Index(['a', 'b', 'c'], dtype='object')
```

Select a column

```
>>> df["a"]
0    0
1    2
2    4
Name: a, dtype: int64
```

Loop over columns:

```
>>> for col in df.columns:
...     print(col)
a
b
c
```

- This, together with setting a new column gives you everything you need to define variables in a loop
- Looping over columns is not slow (if the DataFrame is long)
- I deliberately do not show how to loop over rows because that is painfully slow!

Calculations on Series

```
>>> sr = pd.Series([1, 2, 3])
>>> sr
0    1
1    2
2    3
dtype: int64
```

```
>>> sr ** 2
0    1
1    4
2    9
dtype: int64
```

```
>>> np.exp(sr)
0     2.718282
1     7.389056
2    20.085537
```

```
>>> sr.sum()
6
```

- Addition, multiplication, exponentiation etc. work as expected
- You can use fast elementwise numpy functions
- The usual reductions like `sum`, `mean` and `std` can be used directly as Series methods.

Task 7

10 min

Query

```
>>> print(df.query("a >= 2 & b < 4"))
   a  b   c
1  2  3  11
```

```
>>> df["d"] = ["bla", "bla", "blubb"]
>>> print(df)
   a  b   c      d
0  0  1  10    bla
1  2  3  11    bla
2  4  5  12  blubb
```

```
>>> print(df.query("a >= 2 & d == 'bla'"))
a  b   c    d
1  2  3  11  bla
```

- Queries take a string that describes which rows should be selected
- Use "&" to combine conditions with 'and'
- Use "|" to combine conditions with 'or'
- Column names do not need quotation marks
- Values of string variables need single quotes
- More in the documentation

Task 8

5 min

Errors and Tracebacks

Simple traceback

```
nested_list = [[1, 2, 3], [4, 5]]
arr = np.array(nested_list)
squares = arr ** 2
```

```
---------------------------------------------------------------------------
ValueError (1)                                Traceback (most recent call last)
Cell In[34], line 2
      1 nested_list = [[1, 2, 3], [4, 5]]
----> 2 arr = np.array(nested_list)
(2)   3 squares = arr ** 2

ValueError: setting an array element with a sequence. The requested array has an inhomogeneo
us shape after 1 dimensions. The detected shape was (2,) + inhomogeneous part. (3)
```

1. ValueError: Usually means you made a mistake in a function call
2. The `` `--->` `` tells you the exact line where it happened
3. The message tells you what exactly happened

Go through long traceback in notebook

Common sources of errors

- ValueError: You called a function with something invalid
- KeyError: You have a type in a variable name or a dictionary key
- TypeError: You called the function with something that has the wrong type
- ImportError: You have a typo in an import
- ModuleNotFoundError: You did not activate the environment

How not to ask for help

- "I wanted to do the exercise but it does not work"
- "pandas does not work on my computer"
- "My query does not work, here is a screenshot"

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~/miniconda3/envs/dl_intro/lib/python3.11/site-packages/pandas/core/computation/scope.py:198, in Scope.resolve
(self, key, is_local)
    197 if self.has_resolvers:
--> 198     return self.resolvers[key]
    200 # if we're here that means that we have no locals and we also have
    201 # no resolvers

File ~/miniconda3/envs/dl_intro/lib/python3.11/collections/__init__.py:1004, in ChainMap.__getitem__(self, key)
   1003         pass
-> 1004 return self.__missing__(key)

File ~/miniconda3/envs/dl_intro/lib/python3.11/collections/__init__.py:996, in ChainMap.__missing__(self, key)
    995 def __missing__(self, key):
--> 996     raise KeyError(key)
```

What to keep in mind

- I do not remember what task 3 in exercise 5 is
- I like to see that you tried on your own
- I like to see that you tried to reduce the amount I have to read
- I like well formatted and copy pastable examples

A better way

In the task where we should query the iris dataset (exercise 2, Task 8), the following line gives me a KeyError:

```
iris.query("sepal_lenght <= 5")
```

The message was

```
KeyError: 'sepal_lenght'
```

I tried to run the query with different variables and it works, but with sepal_length it doesn't.

I attach the entire traceback as `` `txt` `` file ...

You can go even further

- Read this excellent blogpost by Matthew Rocklin

Task 9

(10 min)