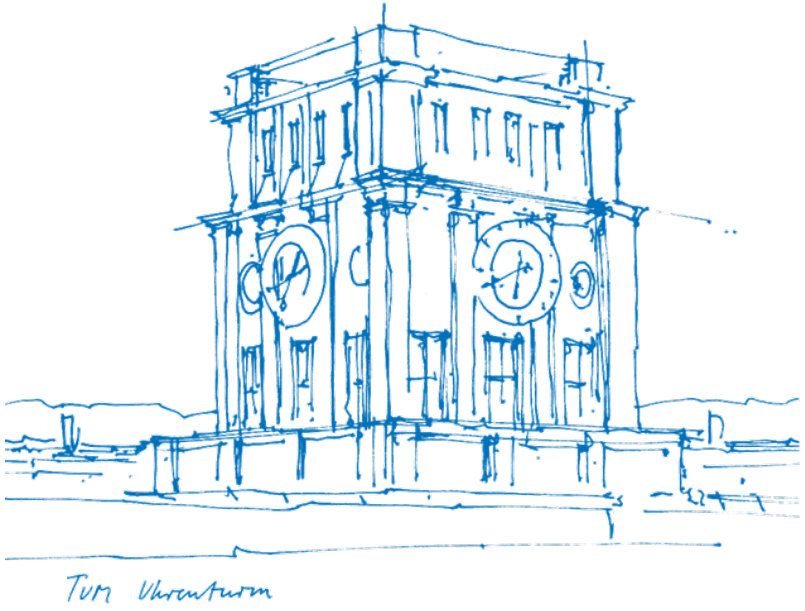
Introduction to Python for MLcomm

Fabian Steiner

Machine Learning for Communications – TUM LNT

October 15th, 2019



Overview

Introduction

Python Basics

Libraries for Numerical Computations
- NumPy
- SciPy
- Pandas

Outlook

Practice problems

What is Python?

- Python is an interpreted, non-statically typed[1] language.
- It supports different programming paradigms (functional, object-oriented, imperative, etc.).
- It supports all major operating systems and comes with a huge standard library.
- Python as a language has different implementations:
  - CPython – standard, reference implementation.
  - PyPy – based on a just-in-time (JIT) compiler. Major speedups compared to CPython.
  - Cython – compiles Python to C.
- Python is open source.

[1] Since Python 3.5 type hints are possible and should be used extensively. See also PEP483, PEP484.

Why Python for ML?

- Python has established a good reputation in the data science field.
- It is a language that is easy to start with.
- It is freely available (compare[2]: Matlab 2000 Euro + Statistics and Machine Learning toolbox: 1000 Euro).

[2] https://de.mathworks.com/pricing-licensing.html
[3] https://www.tensorflow.org/
[4] https://keras.io/
[5] http://scikit-learn.org/

Why Python for ML?

- Python has established a good reputation in the data science field.
- It is a language that is easy to start with.
- It is freely available (compare[2]: Matlab 2000 Euro + Statistics and Machine Learning toolbox: 1000 Euro).
- Three prominent machine learning libraries are developed in Python: Tensorflow[3], Keras[4], scikit-learn[5].

[2] https://de.mathworks.com/pricing-licensing.html
[3] https://www.tensorflow.org/
[4] https://keras.io/
[5] http://scikit-learn.org/

Why Python for ML?

- Python has established a good reputation in the data science field.
- It is a language that is easy to start with.
- It is freely available (compare[2]: Matlab 2000 Euro + Statistics and Machine Learning toolbox: 1000 Euro).
- Three prominent machine learning libraries are developed in Python: Tensorflow[3], Keras[4], scikit-learn[5].

- Extend your horizon: There's a world beyond Matlab.

[2] https://de.mathworks.com/pricing-licensing.html
[3] https://www.tensorflow.org/
[4] https://keras.io/
[5] http://scikit-learn.org/

Programming Envrionment: Jupyter Notebooks

- Following good practice in the ML community, we use Jupyter Notebooks[6].

- Provides an interactive console, where you can type in commands which facilitates trial & error.
- Can be installed locally on Windows, Linux and Mac: Use the Anaconda distribution[7].

[6] http://jupyter.org/

[7] https://www.anaconda.com/download/

Programming Envrionment: Jupyter Notebooks

- Following good practice in the ML community, we use Jupyter Notebooks[6].

- Provides an interactive console, where you can type in commands which facilitates trial & error.
- Can be installed locally on Windows, Linux and Mac: Use the Anaconda distribution[7].

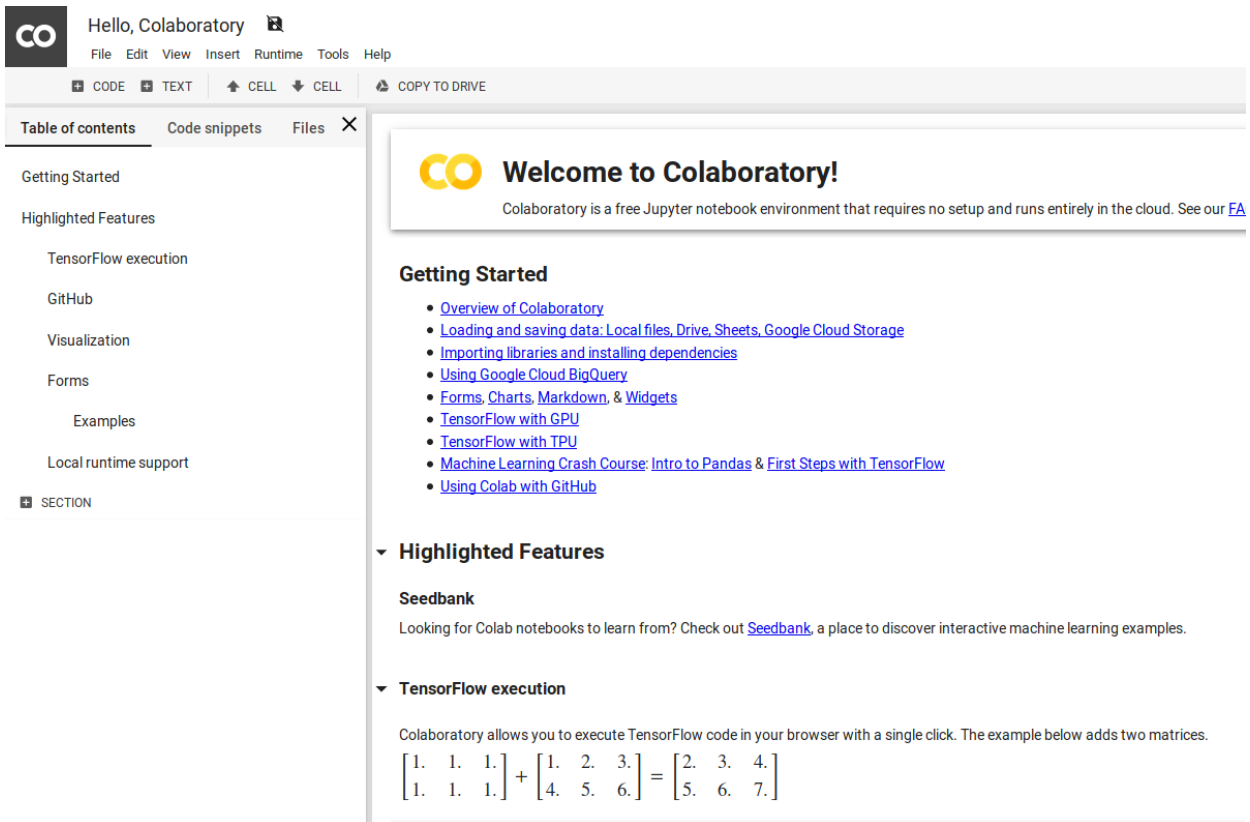
But there is also an easier way . . ..

[6] http://jupyter.org/

[7] https://www.anaconda.com/download/

Programming Environment: Google Colab (I)

- Google provides hosted Jupyter Notebooks.



Programming Environment: Google Colab (II)

- You don’t need to install anything and have access to GPUs and TPUs! We will cover this soon!

Programming Environment: Google Colab (II)

- You don't need to install anything and have access to GPUs and TPUs! We will cover this soon!
  - GPU (graphics processing unit): computation on graphic cards (e.g., NVIDA GeForce 1800Ti) to speed up various computing tasks like matrix vector multiplications.
  - TPU (tensor processing unit): Google's proprietary ASIC for AI tasks, improves upon GPUs, used in the AlphaGo matches.

Programming Environment: Google Colab (II)

- You don't need to install anything and have access to GPUs and TPUs! We will cover this soon!
  - GPU (graphics processing unit): computation on graphic cards (e.g., NVIDA GeForce 1800Ti) to speed up various computing tasks like matrix vector multiplications.
  - TPU (tensor processing unit): Google's proprietary ASIC for AI tasks, improves upon GPUs, used in the AlphaGo matches.
  - Might not be available in the evening, when all the west coast kiddies get up and start doing their homework.

Programming Environment: Google Colab (II)

- You don't need to install anything and have access to GPUs and TPUs! We will cover this soon!
  - GPU (graphics processing unit): computation on graphic cards (e.g., NVIDA GeForce 1800Ti) to speed up various computing tasks like matrix vector multiplications.
  - TPU (tensor processing unit): Google's proprietary ASIC for AI tasks, improves upon GPUs, used in the AlphaGo matches.
  - Might not be available in the evening, when all the west coast kiddies get up and start doing their homework.
- Drawback: You need a Google account for this. Sorry!

Python Basics

Basics: Data Types

Python 3 supports the following data types:

- Integers (`int`)
- Floats (`float`)
- Booleans (`bool`)
- Strings (`str`)
- Lists (`list`)
- Tuples (`tuple`)
- Sets (`set`)
- Dictionaries (`dict`)

Basics: Data Types

Python 3 supports the following data types:

- Integers (`int`)
- Floats (`float`)
- Booleans (`bool`)
- Strings (`str`)
- Lists (`list`)
- Tuples (`tuple`)
- Sets (`set`)
- Dictionaries (`dict`)

The methods of each data type can be inspected via `help`, e.g., `help(int)`.

Basics: Data Types (Integer)

- Integer numbers (`int`):

```
>>> a = 5; type(a)
<type 'int'>
```

- Important method/property: `.real`, `.imag`, `.conjugate()`

Basics: Data Types (Floats)

- Floating point numbers (`float`):

```
>>> a = 5.0; type(a)
<type 'float'>
```

- Important methods: `.real`, `.imag`, `.conjugate()`

Basics: Data Types (Booleans)

- Booleans (`bool`):

```
>>> a = True; type(a)
<type 'bool'>
```

Basics: Data Types (Strings)

- Strings (`str`):

```
>>> a = 'test'; type(a)
<type 'str'>
```

- Important methods: `.format()`, `.find()`, `.join()`, `.split()`.

Basics: Data Types (List)

- Lists (`list`):

```
>>> a = [1, 2, 3]; type(a)
<type 'list'>
>>> a[0]
1
```

- Sequence of arbitrary Python objects that can be modified (mutable) after it has been created.
- Builtin function `len()` returns the number of objects in the tuple.
- Important methods: `.append()`, `.extend()`, `.insert()`, `.remove()`.

Basics: Data Types (Tuples)

- Tuples (`tuple`):

```
>>> a = (1, 2, 3); type(a)
<type 'tuple'>
>>> a[0]
1
```

- Sequence of arbitrary Python objects that can not be modified (immutable) after it has been created.
- Builtin function `len()` returns the number of objects in the tuple.

Basics: Data Types (Sets)

- Sets (`set`):

```
>>> a = set((1,2,3,4)); type(a)
<type 'set'>
```

- A set object is an unordered, mutable collection of distinct Python objects, i.e., `set((1,2,3)) == set((3,2,1,1))`.
- Represents the mathematical concept of a set.
- Supports the associated mathematical operations `.intersection()`, `.union()`, `.difference()`.
- Builtin function `len()` returns the number of objects in the tuple.

Basics: Data Types (Dictionaries)

- Dictionary (`dict`):

```
>>> alphabet = {'a': 1, 'b': 2, 'c': 3}; type(alphabet)
<type 'dict'>
>>> alphabet['b']
2
```

- Build dictionary from list of keys and values:

```
d = dict(zip(mykeys, myvals))
```

- Important methods: `.keys()`, `.values()`, `.items()`.

Basics: Printing and Formatting (I)

- Printing is done via the `print()` function.
- Each string has a corresponding `format()` method.

```
>>> print('Hello {}!'.format('Fabian'))
Hello Fabian!
>>> print('Hello {} {}!'.format('Fabian', 'Steiner'))
Hello Fabian Steiner!
>>> print('Hello {firstname} {lastname}!'.
  format(firstname='Fabian', lastname='Steiner'))
Hello Fabian Steiner!
```

Basics: Printing and Formatting (II)

- Similar to C's `printf()`, several format specifiers are supported.
- Syntax of format specifiers: `:<field_width>.<precision><data_type>`

```
>>> print('Num: {:d}'.format(2))
Num: 2
>>> print('Num: {:10d}'.format(2))
Num:          2
>>> print('Num: {:10d}'.format(223))
Num:        223
>>> print('Num: {:.3f}'.format(3.14159))
Num: 3.142
```

Basics: Conditions and If Statements

- Conditions:
  equals: `a == b`, not equals: `a != b`, less then or equal to: `a <= b`, ...
- If Statements

```
if a == b:
    print('a and b equal!')
elif a < b:
    print('a smaller than b!')
else:
    print('none of the two conditions is true!')
```

- Indentation is important: use four spaces

→ This is how Python identifies blocks of code (cf. curly brackets in C, `end` in matlab)

Basics: Loops

- For Loops

```
for i in range(1,5):
    print(i)
```

```
for i in [1,2,3,5,7]:
    print(i)
```

- While Loops

```
i = 1
while i <= 5
    print(i)
    i += 1
```

- Indentation is important: use four spaces

Basics: List comprehension

- Create lists from existing lists or an iterable object.

```
y = [x**2 for x in range(1, 10)]
```

- This can be combined with conditions.

```
y = [x**2 for x in mylist if x % 2 == 0]
```

- A n-dimensional extension is possible.

```
z = [x*y for x in mylist1 for y in mylist2]
```

Basics: Generators (I)

In many cases, a new list should not be generated explicitely, because the individual list members are not needed. Hence, memory can be saved.

- Traditional

```
res = sum( [x**2 for x in range(1,10)] )
```

- Generator

```
res = sum( (x**2 for x in range(1,10)) )
```

Basics: Generators (II)

More elaborate example:

```
def gen_combs(set1, set2):
  l = []
  for s1 in set1:
    for s2 in set2:
      l.append((s1,s2))
  return l
```

```
def gen_combs(set1, set2):
    for s1 in set1:
        for s2 in set2:
            yield (s1, s2)
```

```
>>> for item in gen_combs((1,2,3),(4,5,6)):
    print(item)
```

Basics: Defining functions

```
def mysum(arg1, arg2):
    result = arg1 + arg2
    return result
```

- Indentation is important: Use four spaces.
- Automatic cleanup tool: `autopep8`.
- Functions help to structure your program and to write reusable code.

Basics: Functions with Type Annotations

```
def mysum(arg1: int, arg2: int): -> int
    result: int = arg1 + arg2
    return result
```

- As mentioned before, Python is dynamically linked.
- Since Python 3.5, type annotations in the code itself are possible.
- Very helpful for collaborations in teams and a large code base.
- Type annotations are not enforced by the interpreter.
- Complex or composite types are also possible, use `typing` module.
- Check out MyPy (`http://mypy-lang.org/`) for static type checking.

Basics: `with` Statement

- Many times you will encounter the following pattern:
  - Open a file/datebase connection.
  - Retrieve and work with the data.
  - Close the connection.
- For convenience, Python takes care of the opening and closing:

```
>>> with open('file.txt') as f:
        data = f.read()
        print(item)
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Basics: Organizing your code (I)

- To avoid clogging your namespace, put your code into separate files and import them if required.
- Example: File `ml_tools.py` with function `entr()`.

```
>>> import ml_tools
>>> H = ml_tools.entr([0.3, 0.7])
```

Basics: Organizing your code (I)

- To avoid clogging your namespace, put your code into separate files and import them if required.
- Example: File `ml_tools.py` with function `entr()`.

```
>>> import ml_tools
>>> H = ml_tools.entr([0.3, 0.7])
```

- If the import name is too long, it can be abbreviated by `import longname as ln`.
- For larger code bases, modules are more appropriate.
- For this, we first create the module folder `mlcomm`, mark it as a module for Python by placing an empty `__init__.py` file and then add the corresponding files.

Basics: Organizing your code (II)

An example module may look like:

```
mlcomm
├── __init__.py
├── tools
│   ├── __init__.py
│   └── it.py
└── em
    ├── __init__.py
    └── em.py
```

The individual parts can be imported as (inspect your namespace with `dir()` afterwards):

```
>>> from mlcomm import em
>>> import mlcomm.tools.it
```

Basics: LBYL vs. EAFP / Exception handling

LBYL

Look before you leap.

EAFP

Easier to ask for forgiveness than permission.

Basics: LBYL vs. EAFP / Exception handling

LBYL

Look before you leap.

EAFP

Easier to ask for forgiveness than permission.

Python's paradigm follows the EAFP style:

```
>>> d = {'name': ['Peter', 'George'], 'age': [20, 30]}
>>> try:
...     places = d['places']
... except KeyError:
...     print('No key named places')
...     places = None
No key named places
```

Coding Style

- You do not write code for yourself, but for others to read it.
- Additionally, you read your own code more often than you will actually write it.
- It is essential to stick to a good practice and common conventions.

[8] `https://www.python.org/dev/peps/pep-0008/`

Coding Style

- You do not write code for yourself, but for others to read it.
- Additionally, you read your own code more often than you will actually write it.
- It is essential to stick to a good practice and common conventions.
- For Python, these rules are summarized in PEP8[8].
  - Naming conventions for variables, functions, classes.
  - Identitation style.
  - Comment style.
- Stick to it. Every (reasonable) company/project leader will enforce this as well!

[8] `https://www.python.org/dev/peps/pep-0008/`

Libraries for Numerical Computations

NumPy

- NumPy is the fundamental package for numerical computing with Python.
- It provides
  - functions for dealing with n-dimensional arrays,
  - various mathematical functions,
  - a Matlab-like interface.
- NumPy uses 0-based indexing.
- NumPy assigns by reference.
- Import NumPy into your code as

```
>>> import numpy as np
```

NumPy: Arrays (I)

- Create matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$:

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

- Index single element

```
>>> a[0,1]
```

- Index first row:

```
>>> a[0,:]
```

- Index first column:

```
>>> a[:,0]
```

NumPy: Arrays (II)

- Get size

```
>>> a.shape
```

- Get number of elements

```
>>> a.size
```

- Vertically concatenate the arrays `a` and `b`:

```
>>> c = vstack((a,b))
```

- Horizontally concatenate the arrays `a` and `b`:

```
>>> c = hstack((a,b))
```

NumPy: Arrays (III)

- Serialize array

```
>>> a.flatten()
```

- Create zero 3×3 matrix

```
>>> a = np.zeros((3,3))
```

- Create 3×3 all ones matrix

```
>>> a = np.ones((3,3))
```

- Create 3×3 identity matrix

```
>>> a = np.eye(3)
```

NumPy: Arrays (IV)

- Create list of values ranging from 1.0 to 4.9 in step sizes of 0.1.

```
>>> a = np.arange(1.0,5.0,0.1)
```

- Transpose.

```
>>> a.T
```

- Conjugate transpose, i.e., Hermitian.

```
>>> a.conj().T
```

NumPy: Linear Algebra

- Innner product of two vectors (1D arrays) `a` and `b`.

```
>>> a.dot(b)
```

- Matrix-vector product of matrix `A` and vector `b`.

```
>>> A.dot(b)
```

- Matrix-matrix product of matrix `A` and matrix `B`.

```
>>> A.dot(B)
```

- Componentwise product of two matrices `A` and `B`.

```
>>> A*B
```

NumPy: Broadcasting

- We want to apply a certain operation to all columns or rows of a matrix.
- Example: Add the vector $\begin{pmatrix} 10 & 10 \end{pmatrix}$ to all rows of the matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$:

NumPy: Broadcasting

- We want to apply a certain operation to all columns or rows of a matrix.
- Example: Add the vector $\begin{pmatrix} 10 & 10 \end{pmatrix}$ to all rows of the matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$:

```
>>> A = np.array([[1,2], [3, 4]])
>>> B = A + np.array([10, 10])
>>> B
array([[11, 12],
       [13, 14]])
```

- This operation is called broadcasting in NumPy. It's a powerful tool!

NumPy: Random Numbers

- Create vector of n normally distributed random numbers:

```
>>> N = np.random.randn(n)
```

- Create vector of n uniformly distributed, integer random numbers between `lb` and `ub`:

```
>>> N = np.random.randint(lb, ub + 1)
```

- For more, see `help(np.random)`.

NumPy: Passing by reference (I)

```
>>> a = np.array([[1,2], [3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> b = a[:,0]
>>> b
array([1, 3])
>>> b[:] = 8
>>> a
array([[8, 2],
       [8, 4]])
```

NumPy: Passing by reference (II)

- If real copies are needed:

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

NumPy: Importing Data

- Read simple text files:

```
>>> data = np.loadtxt('filename.txt')
```

- Save simple text files:

```
>>> np.savetxt('filename.txt', data)
```

- Detailed reference of all parameters can be found online[9].

[9] https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.loadtxt.html

NumPy: Importing Data

- Read simple text files:

```
>>> data = np.loadtxt('filename.txt')
```

- Save simple text files:

```
>>> np.savetxt('filename.txt', data)
```

- Detailed reference of all parameters can be found online[9].
- Read Matlab files:

```
>>> data = scipy.io.loadmat('filename.mat')
```

[9] https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.loadtxt.html

NumPy: Plotting (I)

- Import the necessary functionality:

```
>>> import matplotlib.pyplot as plt
```

- Generate data and plot:

```
>>> x = np.linspace(1,10,10)
>>> y = 2*x
>>> plt.plot(x, y)
>>> plt.show()
```

- Result can be saved with

```
>>> plt.savefig('fig.png')
```

NumPy: Plotting (II)

- Exposed interface is similar to the Matlab plotting functionality.
- If logarithmic plots are desired:
  - `plt.semilogx(x,y)`
  - `plt.semilogy(x,y)`
  - `plt.loglog(x,y)`
- The axis can be modified via
  - `plt.xlabel('X-Label')`
  - `plt.ylabel('Y-Label')`
  - `plt.xlim((0, 10))`
  - `plt.ylim((0, 10))`

NumPy: Outlook

- Full NumPy reference[10].
- Guide for users transitioning from Matlab[11].
- Use `timeit` module for benchmarking[12] small snippets of your code.
- Further information on improving NumPy performance[13].

[10] `https://docs.scipy.org/doc/numpy/reference/`

[11] `https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html`

[12] `https://docs.python.org/2/library/timeit.html`

[13] `http://ipython-books.github.io/featured-01/`

SciPy

What's the relation[14] of SciPy and NumPy?

"In an ideal world, NumPy would contain nothing but the array data type and the most basic operations: indexing, sorting, reshaping, basic elementwise functions, et cetera. All numerical code would reside in SciPy. However, one of NumPy's important goals is compatibility, so NumPy tries to retain all features supported by either of its predecessors. Thus NumPy contains some linear algebra functions, even though these more properly belong in SciPy. [...]"

[14] https://www.scipy.org/scipylib/faq.html#id16

SciPy

- The SciPy module therefore contains the actual numerical algorithms.
- Import module as

```
>>> import scipy as sc
```

  - `sc.integrate`: Numerical integration, quadrature rules.
  - `sc.optimize`: Constrained/unconstrained optimization algorithms, root finding.
  - `sc.linalg`: Supersedes `np.linalg`.
  - `sc.stats`: Implements various distributions, their PDFs, CDFs and moments.

SciPy

- The SciPy module therefore contains the actual numerical algorithms.
- Import module as

```
>>> import scipy as sc
```

  - `sc.integrate`: Numerical integration, quadrature rules.
  - `sc.optimize`: Constrained/unconstrained optimization algorithms, root finding.
  - `sc.linalg`: Supersedes `np.linalg`.
  - `sc.stats`: Implements various distributions, their PDFs, CDFs and moments.
- Instead of re-inventing the wheel (numerical algorithms can be super hard to implement reliably!), use the provided ones.
- But: Make always sure that they actually implement what you would like to have.

Pandas

- Machine learning is closely associated with “big data”.
- Before being able to work with big data, you first have to get it into Python.

Pandas

- Machine learning is closely associated with “big data”.
- Before being able to work with big data, you first have to get it into Python.
- Pandas provides convenient abstraction layers for handling data.
  - Reading and writing spreadsheets.
  - Sorting and viewing data.
  - Database-like access: joins, groups, pivoting.

Outlook

Outlook

- A lot of topics could not be covered today.

Outlook

- A lot of topics could not be covered today.
  - Exception handling.

Outlook

- A lot of topics could not be covered today.
  - Exception handling.
  - Database interaction.

Outlook

- A lot of topics could not be covered today.
  - Exception handling.
  - Database interaction.
  - Filesystem access.

Outlook

- A lot of topics could not be covered today.
  - Exception handling.
  - Database interaction.
  - Filesystem access.
  - Concurrent execution.

Outlook

- A lot of topics could not be covered today.
  - Exception handling.
  - Database interaction.
  - Filesystem access.
  - Concurrent execution.
  - Object oriented programming: concept of objects and classes.

Outlook

- A lot of topics could not be covered today.
  - Exception handling.
  - Database interaction.
  - Filesystem access.
  - Concurrent execution.
  - Object oriented programming: concept of objects and classes.
  - Virtual environments.

Outlook

- A lot of topics could not be covered today.
  - Exception handling.
  - Database interaction.
  - Filesystem access.
  - Concurrent execution.
  - Object oriented programming: concept of objects and classes.
  - Virtual environments.
  - Extensions with own C modules.

Outlook

- A lot of topics could not be covered today.
  - Exception handling.
  - Database interaction.
  - Filesystem access.
  - Concurrent execution.
  - Object oriented programming: concept of objects and classes.
  - Virtual environments.
  - Extensions with own C modules.
- Play around yourself, write code and discuss with your colleagues.

Practice Problems

Practice problems I

The following simple tasks will help you to familiarize yourself with Python and to recap some of the basics that have been introduced.

1. Implement a function with the signature `discrete_entr`

```
def discrete_entr(pX): pass
```

   that calculates the entropy of the provided distribution `pX`. Take care of a proper error checking. The entropy is defined as

$$\sum_{x \in \mathrm{supp}(P_X)} -P_X(x) \log_2(P_X(x)).$$

Practice problems II

2. Implement a function with the signature `discrete_cross_entr`

```
def discrete_cross_entr(pX, pY): pass
```

that calculates the cross-entropy of the distributions `pX` and `pY`. Take care of a proper error checking and write a unit test. The cross entropy is defined as

$$\sum_{x \in \mathrm{supp}(P_X)} -P_X(x) \log_2(P_Y(x)).$$

Practice problems III

3. Implement a function with the signature `discrete_kl_div`

```
def discrete_kl_div(pX, pY): pass
```

that calculates the Kullback-Leibler divergence of the distributions `pX` and `pY`. Take care of a proper error checking and write a unit test. The Kullback-Leibler divergence is defined as

$$\sum_{x \in \mathrm{supp}(P_X)} P_X(x) \log_2 \left(\frac{P_X(x)}{P_Y(x)} \right).$$

Practice problems IV

4. Implement a function with the signature `act_fct`

```
def act_fct(x,type_fct): pass
```

that returns the value of different activation functions evaluated at `x` depending on the `type_fct` parameter:

- Identity (`identity`): $y = f(x) = x$.
- Sigmoid (`sigmoid`): $y = f(x) = \frac{1}{1+\mathrm{e}^{-x}}$.
- Tanh (`tanh`): $y = f(x) = \tanh(x)$.
- Rectified linear unit (`rect_lin_unit`): $y = f(x) = \max(0, x)$.