University of Zurich[UZH]

Department of Physics

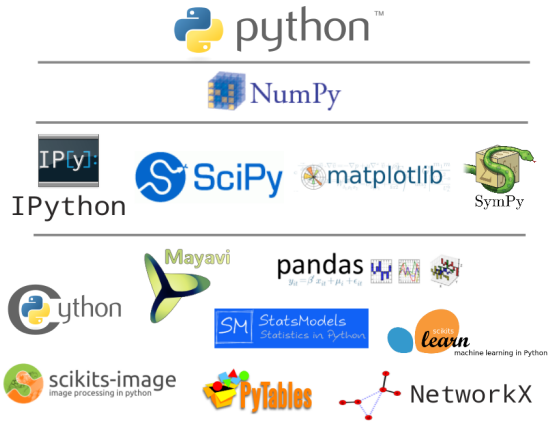
Scientific Programming: Data Structures – NumPy, Pandas & beyond

Scientific Programming with Python

Christian Elsasser

Based partially on a talk by Stéfan van der Walt

The Ecosystem of Homo Python Scientificus



[Ondřej Čertík/LANL]

A Few Technical Remarks

If you want to follow directly the code used in the lecture

- Download the code from the course homepage (Lecture 4)
- Start the virtual environment
  `$ . venv/bin/activate` (from the home directory)
- Create a kernel for the notebook with the virtual environment
  `$ python3 -m ipykernel install --user --name=ve3`
- Unzip the file
  `$ tar zxvf material_Data_lec.tar.gz`
- Enter the created directory
  `$ cd material_Data_lec`
- ... and start the notebook
  `$ ipython3 notebook`

Table of Contents

Numpy – the Fundamental Container for Scientific Computing

`import numpy as np`

NumPy offers memory-efficient data containers for fast numerical operations, *i.e.* in data manipulation and also in typical linear algebra calculations

Standard Python

```
>>> L = list(range(1000))
>>> [i**2 for i in L]
```

NumPy

```
>>> import numpy as np
>>> a = np.arange(1000)
>>> a**2
```

$\Rightarrow$ Speed up by a factor of $\sim$ 100

Creating NumPy Arrays

There are several ways to do so

Creating arrays

```
>>> a = np.array([1,2,4])   # [1,2,4]
>>> b = np.arange(1,15,2)   # [1,3,5,7,9,11,13,15]
>>> c = np.linspace(0,1,6)  # [0.0,0.2,0.4,0.6,0.8,1.0]
>>> d = np.ones((3,3))              # 3x3 array of ones
>>> e = np.zeros((2,5,3))           # 2x5x3 array of zeros
>>> f = np.eye(4)                   # 4x4 unit ‘matrix‘
>>> g = np.diag(np.array([1,2,3,4])) # diagonal ‘matrix‘
>>> h = np.random.rand(4)           # array with [0,1]
>>> i = np.random.randn(4,5)        # 4x5 array (norm. dist)
```

Random seed can be set with `np.random.seed(<integer>)`

Details about NumPy

NumPy's C API

```
ndarray typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyArrayObject ;
```

`np.__version__` indicates version, `np.show_config()` reveals information about LinAlg Calculus

Basic Operations

Many basic functions/operators can be applied on numPy arrays

Examples

```
>>> a = np.random.rand(3,4)
>>> b = np.random.rand(3,4)

>>> a+b
>>> a*b
>>> a/b
>>> a+3.0
>>> np.exp(b)
>>> a>b
```

All element-wise operations including dedicated functions, called universal functions (ufunc)

`math.exp(a)` $\Rightarrow$ failure as it expects scalar

Data Representation

Data type accessible via `dtype` variable

Datatype

```
>>> a = np.array([1,0,-2],dtype=np.int64)   #[1,0,-2]
>>> b = np.array([1,0,-2],dtype=np.float64) #[1.0,0.0,-2.0]
>>> c = np.array([1,0,-2],dtype=np.bool)    #[True,False,True]
>>> c.dtype # dtype('bool')
```

Data Structure

Information via attributes accessible:

`ndim`	number of dimensons
`nbytes`	data size
`shape`	size of the different dimensions (as a tuple)
`size`	total number of entries
`data`	memoryview of the data (`tobytes()` returns the byte representation)
`strides`	number of bytes to jump to in-/decrement index by one (as a tuple)
`flags`	among other things if the memory “belongs” to this array

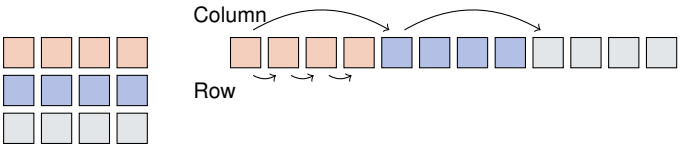
Transpose of arrays can be called by `<array name>.T` $\Rightarrow$ inverts shape and strides (*i.e.* C-contiguous $\leftrightarrow$ F-contiguous)

Be aware that many manipulations do not lead to memory duplications. You can force it by the `copy` method.

Data Structure

Strides

Problem of one-dimensional memory to store multi-dimensional arrays:

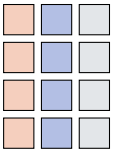


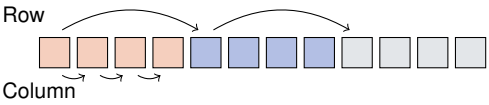
Strides describe the logical alignment of the data within the memory

Data Structure

Strides

Problem of one-dimensional memory to store multi-dimensional arrays:





Transposing the array means to interchange the strides of the different dimensions.

Strides describe the logical alignment of the data within the memory

Data Structure

Information via attributes accessible:

`ndim`	number of dimensons
`nbytes`	data size
`shape`	size of the different dimensions (as a tuple)
`size`	total number of entries
`data`	memoryview of the data (`tobytes()` returns the byte representation)
`strides`	number of bytes to jump to in-/decrement index by one (as a tuple)
`flags`	among other things if the memory “belongs” to this array

Transpose of arrays can be called by `<array name>.T` $\Rightarrow$ inverts shape and strides (*i.e.* C-contiguous $\leftrightarrow$ F-contiguous)

Be aware that many manipulations do not lead to memory duplications. You can force it by the `copy` method.

Get the Data

Reading data from txt/csv/etc. files can be sometimes very painful, especially with complicated/mixed data structure

NumPy offers with the function `loadtxt` an easy way to read in data from text files

`delimiter` defines the columns separation, `comments` the string indicating comments in the text files

Binary files as well as text files are also readable via the function `fromfile`

Get the Data

Complicated data structure are manageable by defining the data type, *e.g.*

Solar.txt (Solar system on June 21, 2014)

```
Sun     332946 2.13E-03 -1.60E-03 -1.20E-04  5.01E-06 ...
Mercury 0.0552 1.62E-01  2.64E-01  6.94E-03 -2.97E-02 ...
Venus   0.8149 3.02E-01  6.54E-01 -8.44E-03 -1.85E-02 ...
Earth   1.00   5.66E-01 -8.46E-01 -9.12E-05  1.40E-02 ...
...
```

Datatype

```
>>> dt = np.dtype([('name','|S7'),('mass',np.float),
 ('position',[('x',np.float),('y',np.float),('z',np.float)]),
 ('velocity',[('x',np.float),('y',np.float),('z',np.float)])])

>>> data = np.loadtxt('Solar.txt',dtype=dt)
```

String in Arrays

String in arrays are in principle not a problem (as seen before), but two things have to be kept in mind

1. Speed reduction due to a different common base type of the objects stored in the array (*i.e.* PyObject)
2. Memory spoiling since the entry size is defined by the maximal length of the stored strings

$\Rightarrow$ if possible, better work with *e.g.* lookup tables

In general you can mix different data type in an array

Mixed datatype

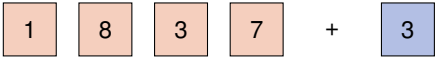
```
>>> na = np.array([2,True,"Hello"],dtype=object)
```

without `dtype=object` the elements would be treated as strings

Broadcasting – Leveraging Vectorisation

Memory-friendly way of combining arrays with different shapes in mathematical operations

Example:



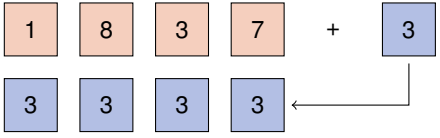
Arrays are alignable if the number of elements in the dimensions match (*i.e.* they are equal or there is only one element)

Details can be found in docstrings `np.doc.broadcasting`

Broadcasting – Leveraging Vectorisation

Memory-friendly way of combining arrays with different shapes in mathematical operations

Example:



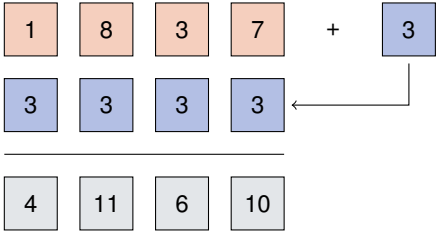
Arrays are alignable if the number of elements in the dimensions match (*i.e.* they are equal or there is only one element)

Details can be found in docstrings `np.doc.broadcasting`

Broadcasting – Leveraging Vectorisation

Memory-friendly way of combining arrays with different shapes in mathematical operations

Example:

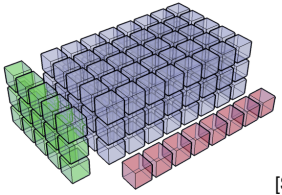


Arrays are alignable if the number of elements in the dimensions match (*i.e.* they are equal or there is only one element)

Details can be found in docstrings `np.doc.broadcasting`

Broadcasting – More complex

Multiplication of a 3×5-array and a 8-element array



[S. v. d. Walt]

Broadcasting

```
>>> a = np.random.rand(3,5)
>>> b = np.random.rand(8)
>>> c = a[...,np.newaxis]*b
>>> c.shape # (3,5,8)
```

`np.newaxis` allows to align the dimensions of arrays so that they can be broadcasted, but be careful and make sure the arrays are aligned as you want them.

Broadcasting – Matching Rules

This principle can be expanded to multi-dimensional arrays, *e.g.* a 3×4-array and a 1D 4-elements array ⇒ adding/multiplying/etc. to each of the three rows the 1D array

Rule: Compare dimensions, starting from the last one. Match when either dimension is one or None, or if dimensions are equal.

`(3,4)`	`(4,1,6)`	`(3,4,1)`	`(3,2,5)`	`(4,2,3)`	`(4,1,3)`
`(4)`	`(1,3,6)`	`(8)`	`(6)`	`(4,3)`	`(4,3)`
`(3,4)`	`(4,3,6)`	`(3,4,8)`	`not OK`	`not OK`	`(4,4,3)`

Arrays can be extended to further dimensions by
`<array name>[...,np.newaxis]`, *e.g.*

`a.shape` → `(3,2)`
⇒ `a[...,np.newaxis,np.newaxis].shape` → `(3,2,1,1)`

Explicit Broadcasting

NumPy has the method `broadcast_arrays` to align two or more arrays

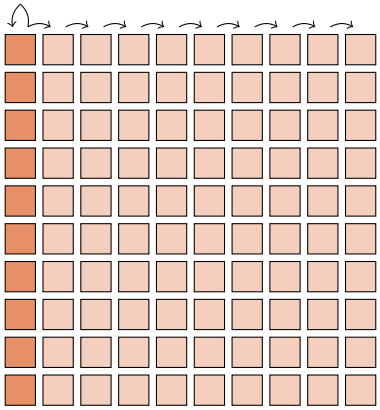
Explicit Broadcasting

```
>>> d = np.random.rand(1,10)
>>> e = np.random.rand(10,1)
>>> dd,ee = np.broadcast_arrays(d,e)
```

`dd` and `ee` are now 10 $\times$ 10-arrays, but without own data

Broadcasted arrays have a stride of zero $\Rightarrow$ pointer stays while index moves

This concept is a generalisation of the `meshgrid` function in MATLAB



Simple Indexing

NumPy allows to easily select subsets in the array, *e.g.*

Simple indexing

```
>>> a = np.arange(100).reshape(10,10)
>>> a[4:9]          # rows 4 to 8
>>> a[:,3:8]        # columns 3 to 7
>>> a[:,-1]         # the last column
>>> a[-2::-3,1:6:2] # 2nd-to-last row every 3rd and every odd column from 1 to 5
```

Also repetition of rows or columns are possible, *e.g.*

Simple indexing (continued)

```
>>> a[:,[1,3,1]]
```

All these operations do not create additional memory entries!

Fancy Indexing

NumPy also allows to select subsets via arrays of indices, *e.g.*

Fancy indexing

```
>>> a = np.arange(100).reshape(10,10)
>>> i0 = np.random.randint(0,10,(8,1,8))
>>> i1 = np.random.randint(0,10,(2,8))
>>> a[i0,i1] # creates a 8×2×8 array
```

- First broadcasting of the two index arrays `i0` and `i1`
- Then selecting the elements in `a` according to the broadcasted arrays

Caution: Mixing of indexing types (*e.g.* `b[5:10,i0,:,i1]`) can lead to unpredictable output shapes (and to barely readable code)

Pandas

`pandas` – Never use excel again!

https://pandas.pydata.org

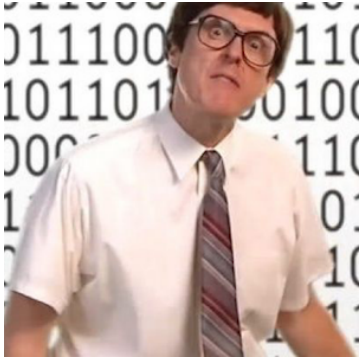
- Offering data containers plus corresponding functionality
- *e.g.* Time series `pd.Series` and their notorious functions (*i.e.* rolling-“whatever”-you-want function)
- Many SQL-like data operations (group, merge)
- Interface to a large variety of file formats (Nobody has heard about all of them!)
- Data interface/API to many data repositories (Yahoo Finance, FRED)

Excel on steroids!

. . . but particularly helpful tool to transform data (clean-up, aggregation, . . .)

numpy *vs.* pandas

Numpy



fast and good with numbers

Pandas

a bit slow and cool with everything

Some Functionalities and Pitfalls

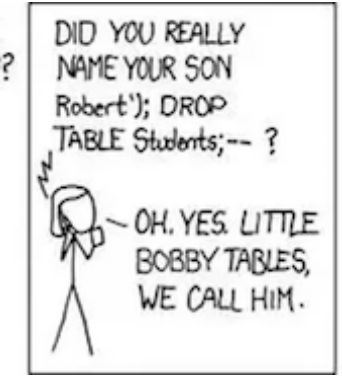
Functionalities

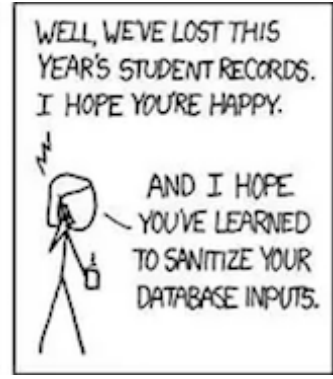
- Fill missing (NA) values according to different principles
- `merge`, `join` and `append`
- Derive new features via `map` (from Series) or `apply` (from Dataframe)
- `groupby` and `resample` (time series) to aggregate data

Pitfalls

- Pandas tries to be smart!!!
- It accepts data as long as it can derive the lowest common ancestor (which is almost always the case although ending up with `object`)
- ...so you should check the data types `dtypes` since your processing code (*e.g.* `groupby`) will work, but not as expected

Other Data Storing Options have also Their Justification . . .





Pickle and JSON – Brothers from Other Mothers

Pickle

- Python proprietary
- . . . thus also Python objects storable
  → class instances
  → numpy arrays
- Binary files

JSON (javascript online notation)

- Interface to other/web applications
- Similar structures
  Python: array → JSON: array
  Python: dict → JSON: object
- Some format issues need to be cleared

Pickle

```
>>> a = dict(...)
>>> with open(<filename>,'wb') as f_o:
>>>    pickle.dump(a,f_o)
>>> with open(<filename>,'rb') as f_i:
>>>    b = pickle.load(f_i)
```

JSON

```
>>> a = dict(...)
>>> with open(<filename>,'w') as f_o:
>>>    json.dump(a,f_o)
>>> with open(<filename>,'r') as f_i:
>>>    b = json.load(f_i)
```

. . . also string-wise possible (dumps/loads)

Connection to SQL Databases - Example with `sqlite3`

https://www.sqlite.org

What is SQLite?

- SQLite is a light-weight (= server-less) SQL-type (= spreadsheet-based) database system.
- The database processes are direclty embedded in the front-end program.
- Due to the outsourced write-interlock handling write intensive programs will suffer.
- Some SQL commands are not valid (removing column, rename column).

`SQLAlchemy` (http://www.sqlalchemy.org) is for a database-type independent approach probably the most suitable package with connections to:

- MySQL
- Microsoft Access
- SQLite

A Few Typical (SQL) Commands

https://www.sqlite.org

Purpose	Command
Retrieve all the data from a table	`SELECT * FROM <table>`
Retrieve some columns (`c1,c2`) from a table `t` with a condition	`SELECT c1,c2 FROM t WHERE <cond>`
Group entries according to values	`SELECT SUM(c1),AVG(c2) FROM t GROUP BY c3,c4`
Add new entry	`INSERT INTO t (c1,c2) values (v1,v2)`
Delete an entry	`DELETE FROM t WHERE c1=v1 AND c2=v2`

sqlite3

https://docs.python.org/3.6/library/sqlite3.html

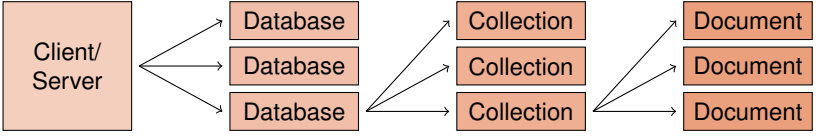
- Database operations on sqlite3 databases
- `sqlite3.conntect` to get a handler on the database
- Default output of (part of) a row is a list → possibility to change the behaviour via the `row_factory` variable of the database
- Use `?` as placeholder insted of concatenating the SQL command by Python string operations
- Use `executemany()` to run same SQL command with several parameter sets
- All executed commands need to be commited before closing the connection (`<dbvariable>.commit()`)

NoSQL Databases and the Flexibility of Data Formats

- General problem of matching datastructures to a spreadsheet
- . . . on the other hand nice ways to store individual “rows” → JSON

MongoDB as most common NoSQL database http://www.mongodb.org

- It requires a corresponding database server.
- `$ mongod` in the console for start-up



- Individual documents as special JSON objects (BSON = binary JSON)

pymongo as Python Interface to MongoDB

Basic interface

How to select the collection

```
>>> from pymongo import MongoClient
>>> client = MongoClient(<url>,<port>)
>>> db = client[<database name>]
>>> col = db[<collection name>]
```

Handle to the collection can be sued to insert, get, alter or delete entries

Actions

How to read/write/modify

```
>>> col.insert_one(<dict>)
>>> col.insert_many(<list of dict>)
>>> col.delete_one/_many(<query>)
>>> col.find_one/find(<query>)
```

Queries are formulated as dictionnaries → `{<variable> : <sub-query>}` with sub-query as `{<operator> : <value(s)>}` or `{"$and/$or" : <list of sub-queries>}`.

- `find` method returns iterable cursor
- MongoDB requires unique identifier `_id` (if not specified given via a hash as `ObjectID`)

Summary

- Python offers various options to handle data suitable for different purposes
  - NumPy is a very powerful tool for numerical computations and data manipulations
  - Pandas offers functionalities of the combination of spreadsheet and databases processing.
  - Various other options to store data – different formats for different purposes
- Further leverage with analytics tool (`scipy`) $\rightarrow$ *cf.* Thursday morning lecture
- Very handy tool for data management ...
- ... but for certain particular tasks other, more suitable options (*e.g.* large image databases that can be heavily compressed)
- Try it out, try it out, try it out!

References

1. Stéfan van der Walt, *Diving into NumPy*, Advanced Scientific Programming in Python, 2013 (Zurich)
2. Bartosz Teleńczuk, *Introduction to data visualization*, Advanced Scientific Programming in Python, 2013 (Zurich)
3. Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux, *The NumPy array: a structure for efficient numerical computation*, Computing in Science and Engineering (IEEE)
4. http://www.numpy.org
5. http://pandas.pydata.org
6. http://www.mongodb.com

Backup

Data Structure (Advanced)

Further information via the `flags` variable accessible:

C_CONTIGUOUS	dimension ordering C-like
F_CONTIGUOUS	dimension ordering Fortran-like
OWNDATA	responsibility of memory handling
WRITEABLE	data changable
ALIGNED	appropriate hardware alignment
UPDATEIFCOPY	update of base array

C-contiguous:
$a[0,0], a[0,1], \ldots, a[0,n], a[1,0], \ldots, a[m,n]$

F-contiguous:
$a[0,0], a[1,0], \ldots, a[m,0], a[0,1], \ldots, a[m,n]$

Broadcasting (Dimensional)

This principle can be expanded to multi-dimensional arrays, *e.g.* a 3×4-array and a 1D 4-elements array ⇒ adding/multiplying/etc. to each of the three rows the 1D array

Rule: Compare dimensions, starting from the last one. Match when either dimension is one or None, or if dimensions are equal.

```
(3,4)        (4,1,6)      (3,4,1)      (3,2,5)
  (4)        (1,3,6)         (8)          (6)
-------------------------------------------------
(3,4)        (4,3,6)      (3,4,8)      not OK
```

Arrays can be extended to further dimensions by
`<array name>[...,np.newaxis]`, *e.g.*

`a.shape` → `(3,2)`
⇒ `a[...,np.newaxis,np.newaxis].shape` → `(3,2,1,1)`