Data Handling in Python

Thijs Smolders
Department of Chemistry - Ångström

Data handling 101

- Data handling refers to the process of managing, organizing, and manipulating data in a structured manner. It plays a fundamental role in various fields, ranging from business analytics to scientific research. Data handling is crucial because it allows us to make sense of the vast amount of information available to us. By effectively managing data, we can extract meaningful insights, identify patterns, detect trends, and make informed decisions.
- Data handling enables us to perform tasks such as data cleaning, preprocessing, aggregation, analysis, and visualization. It empowers us to uncover hidden relationships, discover valuable knowledge, and derive actionable outcomes. Without proper data handling techniques, data can be disorganized, inconsistent, and difficult to interpret, hindering our ability to draw accurate conclusions and gain meaningful insights. Therefore, mastering data handling skills is essential for anyone working with data, as it forms the foundation for successful data analysis and informed decision-making.

Data handling in Python

Python plays a crucial role in data handling due to its rich ecosystem of powerful libraries and tools specifically designed for data manipulation and analysis. Python offers a versatile and user-friendly environment for data handling tasks, making it a preferred choice for data professionals and researchers. One of the most valuable libraries for data handling in Python is **Pandas**, which provides efficient data structures and functions for working with structured data, such as tabular data in the form of DataFrames. Another essential library is **NumPy**, which enables efficient numerical operations and provides support for multidimensional arrays. For handling data visualization, **Matplotlib** and **Seaborn** are widely used libraries that offer a comprehensive range of plotting capabilities. Additionally, the **SciPy** library provides a collection of modules for scientific computing and statistical analysis. Python's standard library includes modules such as **csv** and **json**, making it straightforward to read and write data in common file formats. Furthermore, **scikit-learn** is a popular library for machine learning tasks, and **TensorFlow** and **PyTorch** are widely used for deep learning applications. The combination of these powerful libraries and Python's readability and ease of use make it an ideal language for data handling, analysis, and machine learning tasks.

Short intro to NumPy

NumPy

NumPy is a fundamental library that most of the widely used Python data processing libraries are built upon (pandas, OpenCV), inspired by (PyTorch), or can efficiently share data with (TensorFlow, Keras, etc). Understanding how NumPy works gives a boost to your skills in those libraries as well. It is also possible to run NumPy code with no or minimal changes on GPU.

The central concept of NumPy is an n-dimensional array. The beauty of it is that most operations look just the same, no matter how many dimensions an array has.

All (excellent) visuals in the NumPy section are taken from;
https://betterprogramming.pub/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d#c6af

NumPy

At first glance, NumPy arrays are similar to Python lists. They both serve as containers with fast item getting and setting and somewhat slower inserts and removals of elements.

The hands-down simplest example when NumPy arrays beat lists is arithmetic:

Python list

```
In [3]: a = [1, 2, 3]
        [q*2 for q in a]
Out[3]: [2, 4, 6]
```

NumPy array

```
In [4]: a = np.array([1, 2, 3])
        a * 2
Out[4]: array([2, 4, 6])
```

```
In [1]: a = [1, 2, 3]
        b = [4, 5, 6]
        [q+r for q, r in zip(a, b)]
Out[1]: [5, 7, 9]
```

```
In [2]: a = np.array([1, 2, 3])
        b = np.array([4, 5, 6])
        a + b
Out[2]: array([5, 7, 9])
```

UPPSALA UNIVERSITET

np.array

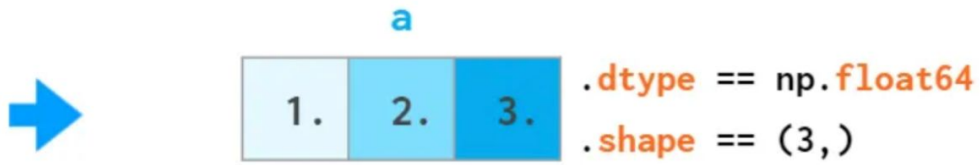
Other than that, NumPy arrays are:

- more compact, especially when there's more than one dimension
- faster than lists when the operation can be vectorized
- slower than lists when you append elements to the end
- usually homogeneous: can only work fast with elements of one type

np.array in 1D

One way to create a NumPy array is to convert a Python list. The type will be auto-deduced from the list element types:

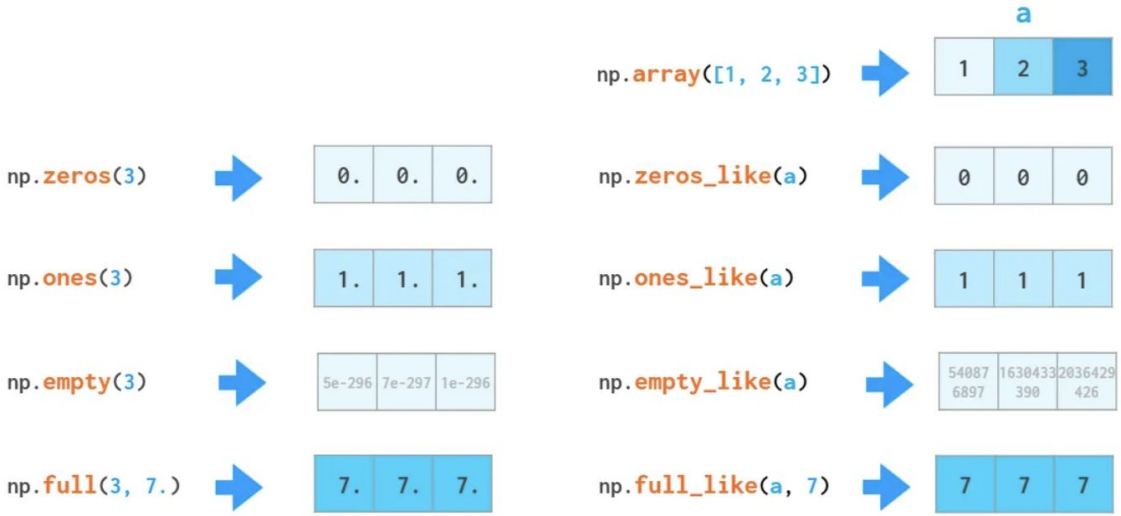
```
a = np.array([1., 2., 3.])
```



Be sure to feed in a homogeneous list, otherwise you'll end up with `dtype='object'`, which annihilates the speed and only leaves the syntactic sugar contained in NumPy.

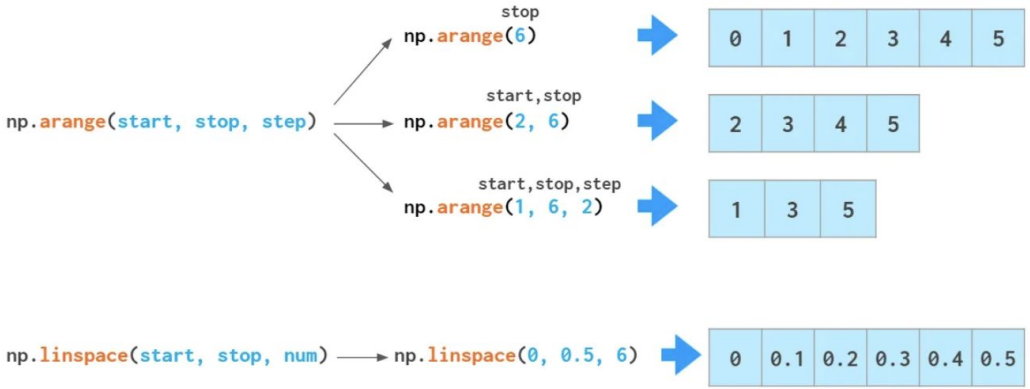
np.array in 1D

NumPy arrays cannot grow the way a Python list does: No space is reserved at the end of the array to facilitate quick appends. So it is a common practice to either grow a Python list and convert it to a NumPy array when it is ready or to preallocate the necessary space with np.zeros or np.empty. It is often necessary to create an empty array which matches the existing one by shape and elements type.



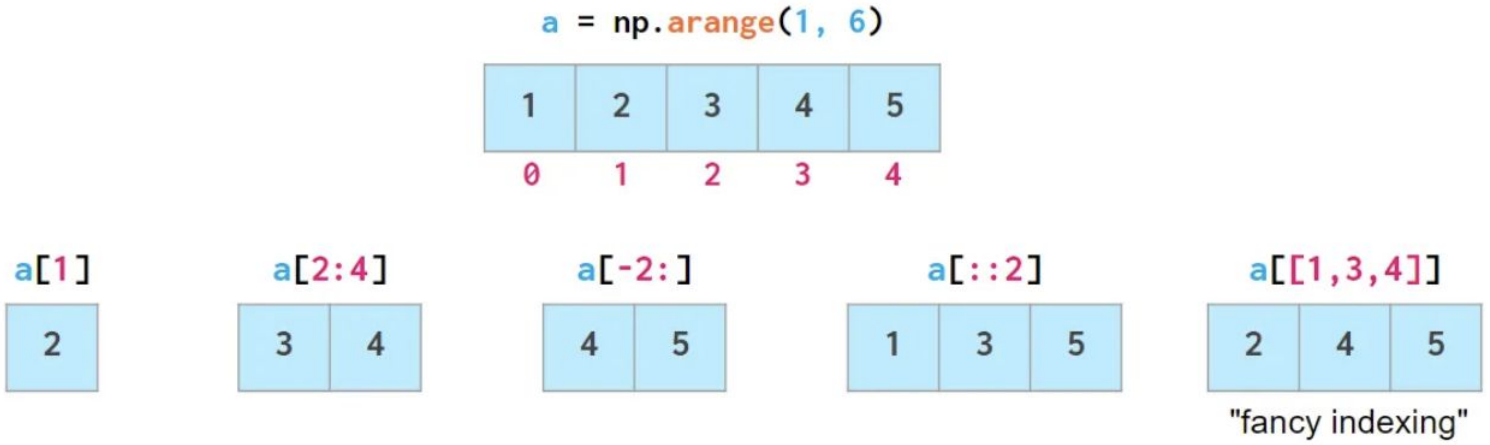
np.array in 1D; array initialization

There are as many as two functions for array initialization with a monotonic sequence in NumPy:



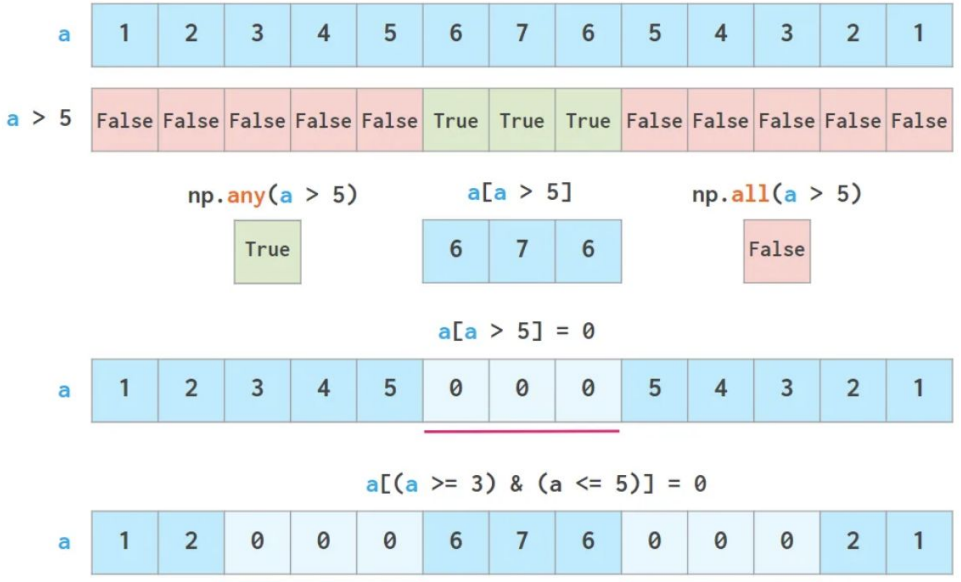
np.array in 1D; vector indexing

Once you have your data in the array, NumPy is brilliant at providing easy ways of giving it back:



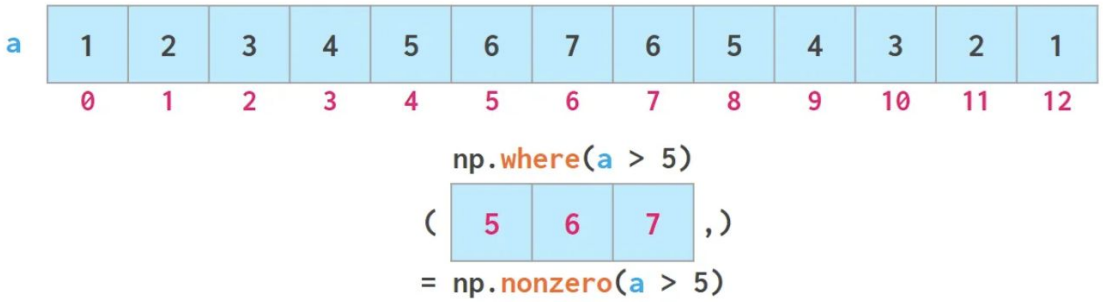
np.array in 1D; boolean indexing

Another super-useful way of getting data from NumPy arrays is boolean indexing, which allows using all kinds of logical operators.



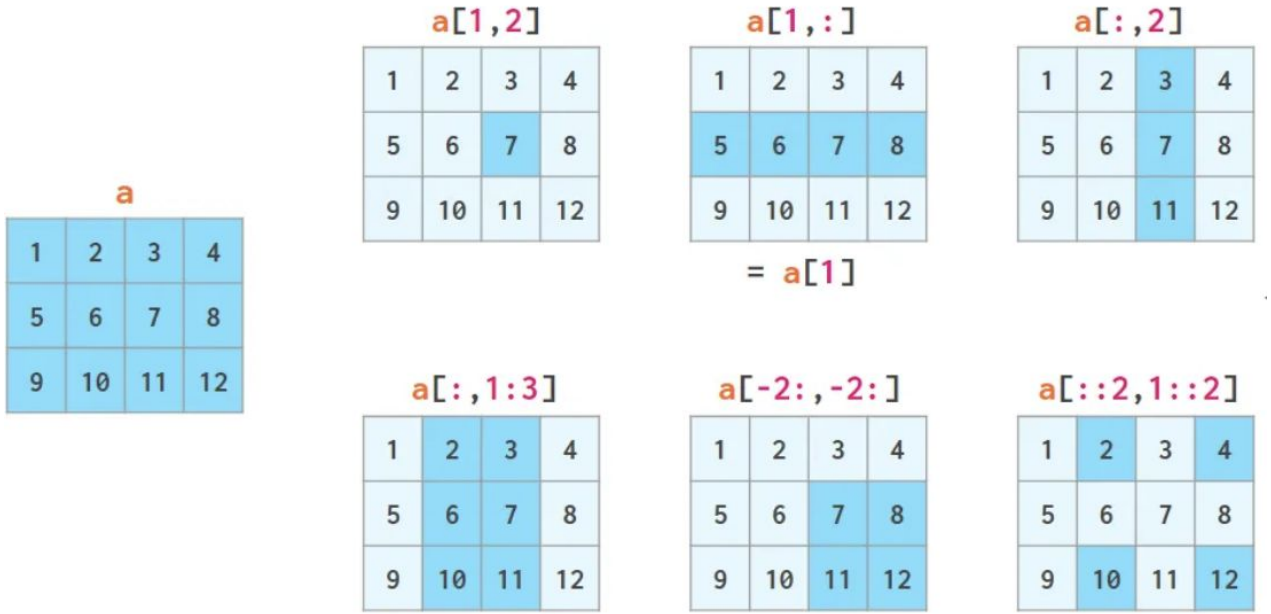
np.array in 1D; np.where

np.where; Return elements chosen from *x* or *y* depending on *condition*.



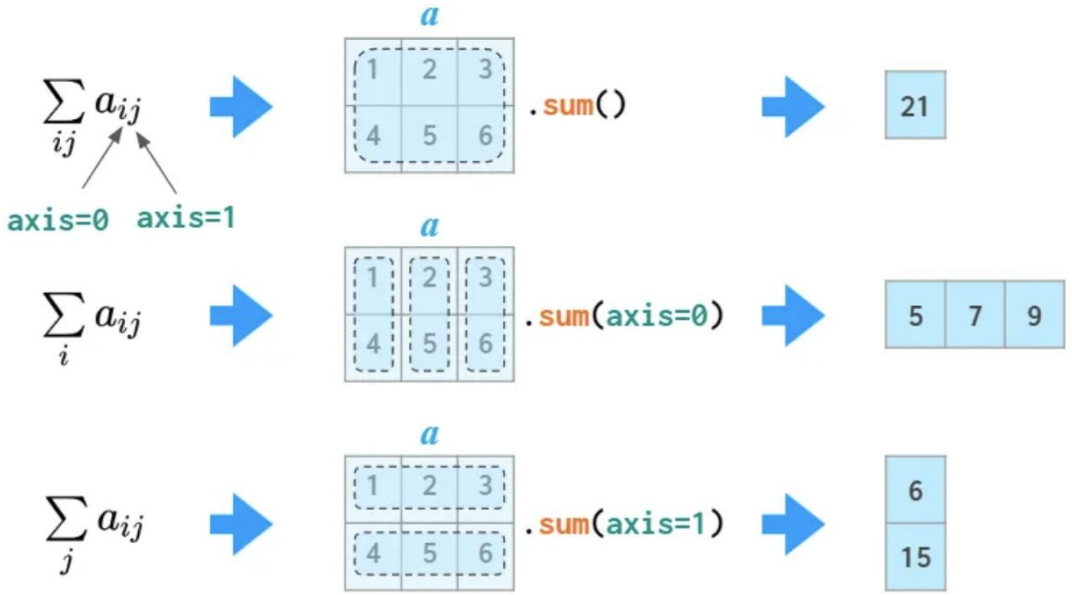
np.array in 2D

np.array; 2D indexing



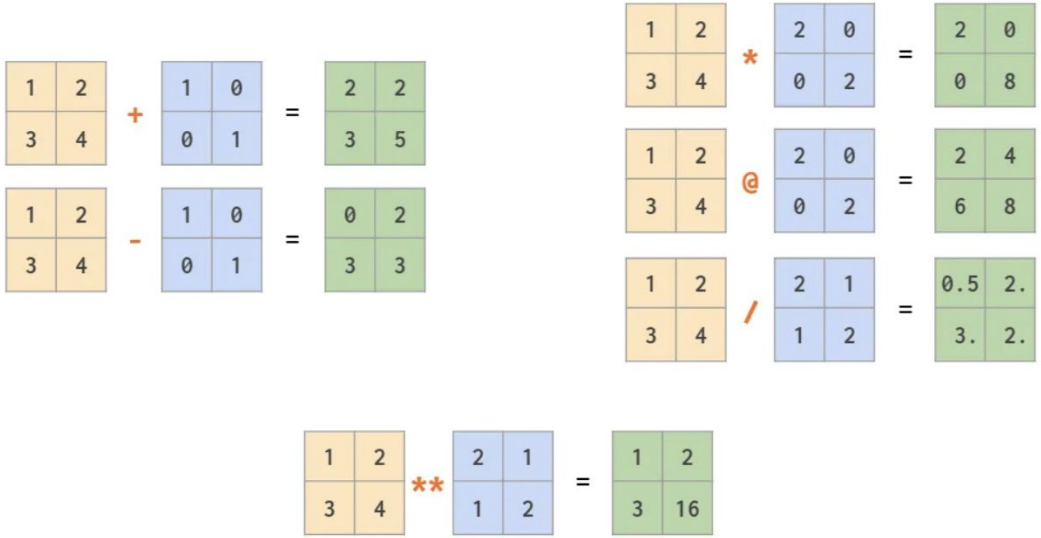
np.array; 2D indexing

In many operations (e.g., sum) you need to tell NumPy if you want to operate across rows or columns. To have a universal notation that works for an arbitrary number of dimensions, NumPy introduces a notion of axis: The value of the axis argument is, as a matter of fact, the number of the index in question: The first index is axis=0, the second one is axis=1, and so on. So in 2D axis=0 is column-wise and axis=1 means row-wise.



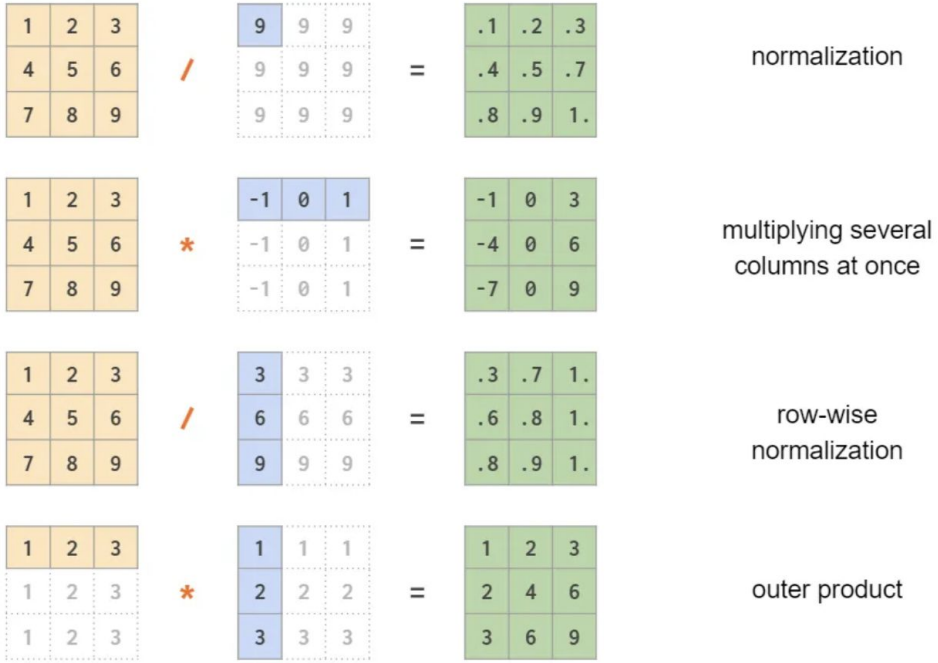
np.array; matrix arithmetic

In addition to ordinary operators (like +,-,*,/,// and **) which work element-wise, there's a @ operator that calculates a matrix product:



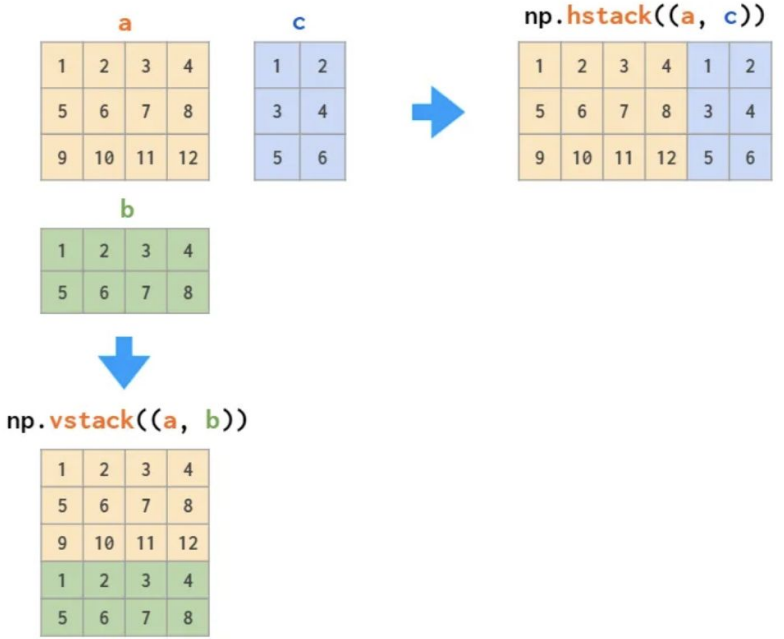
np.array; matrix arithmetic

As a generalization of broadcasting from scalar that we've seen already in the first part, NumPy allows mixed operations between a vector and a matrix, and even between two vectors



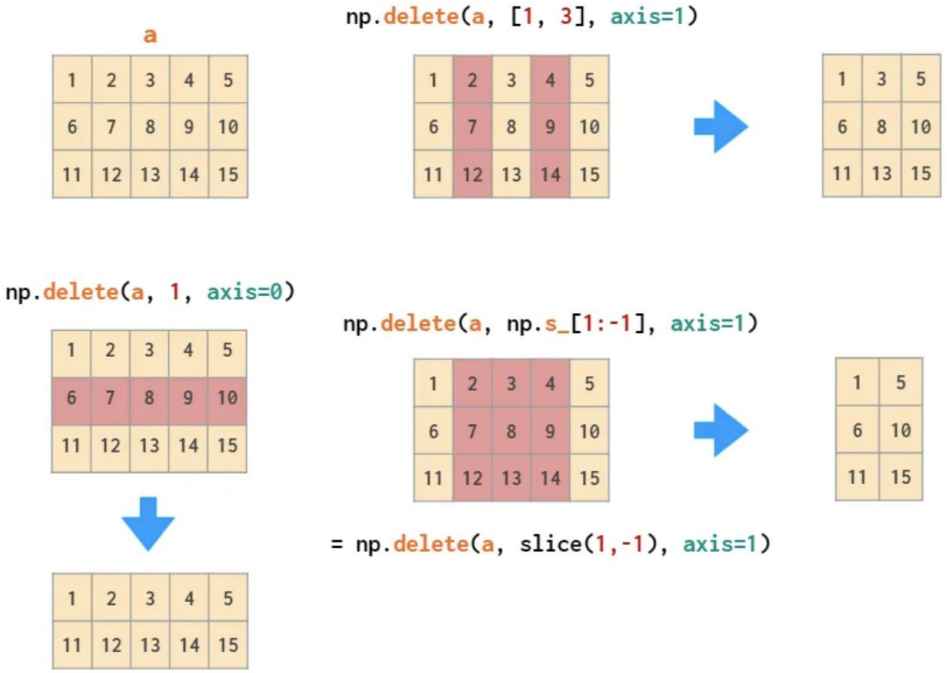
np.array; matrix manipulation

There are two main functions for joining the arrays:



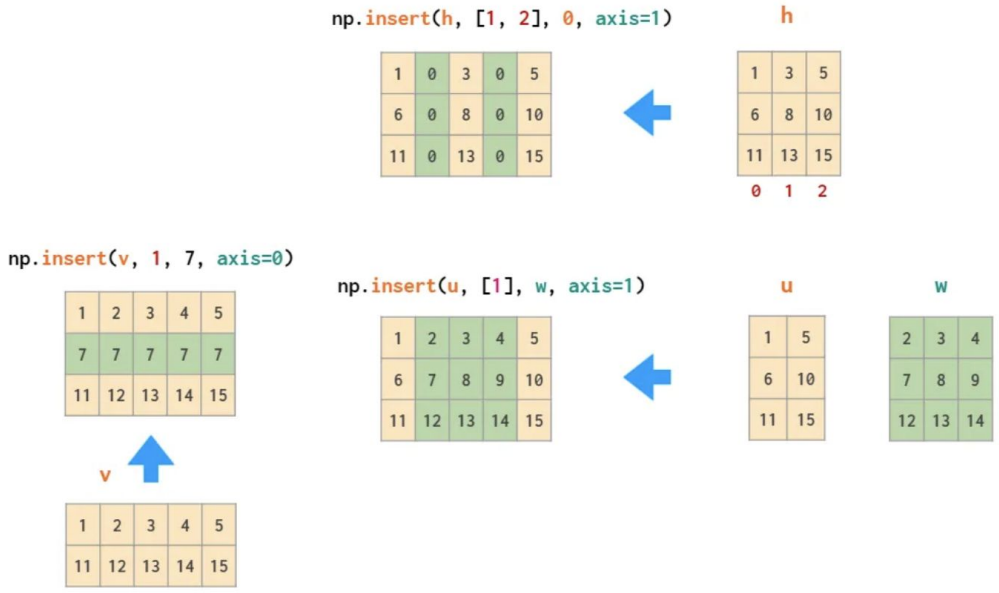
np.array; matrix manipulation

Specific columns and rows can be deleted like:



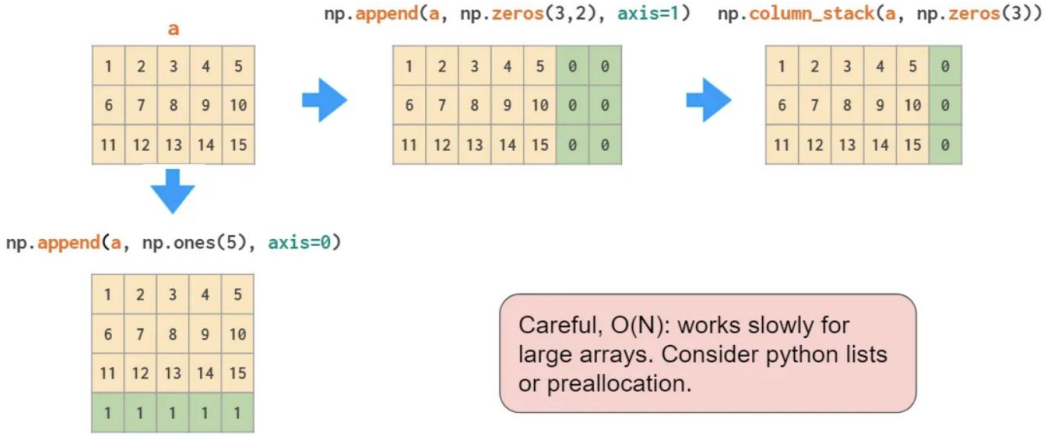
np.array; matrix manipulation

Specific columns and rows can be inserted like:



np.array; matrix manipulation

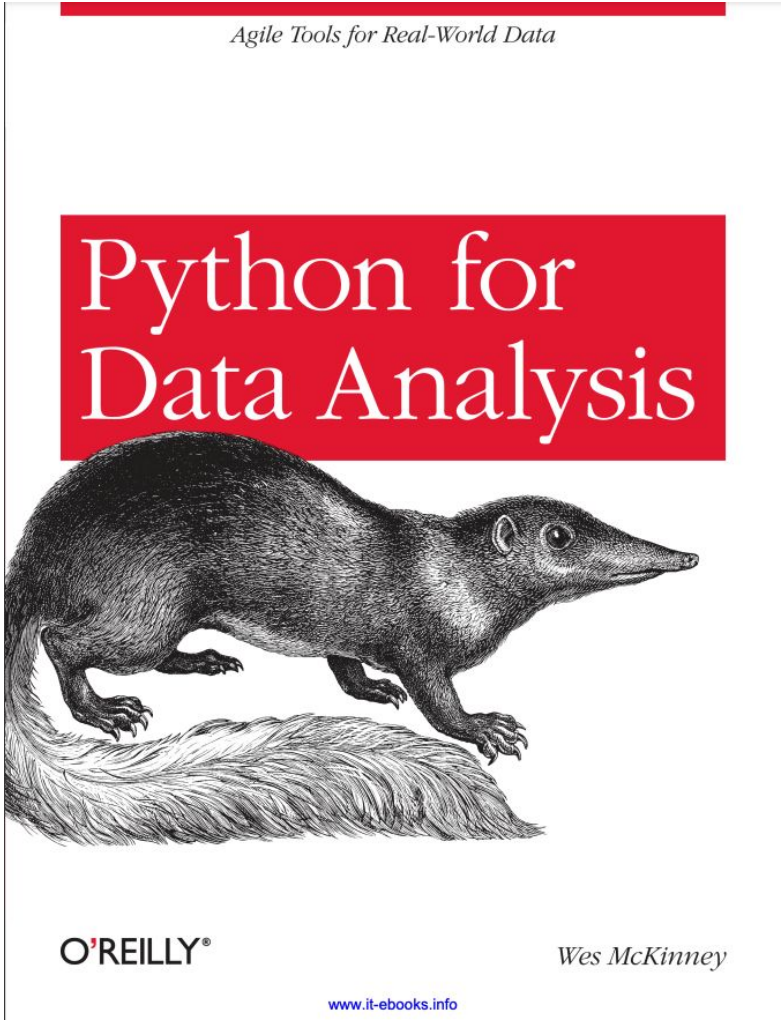
Specific columns and rows can be appended like:



Pandas to the rescue

- Very useful resource!
- This book is concerned with the nuts and bolts of manipulating, processing, cleaning, and crunching data in Python.
- It is also a practical, modern introduction to scientific computing in Python, tailored for data-intensive applications.
- This is a book about the parts of the Python language and libraries you'll need to effectively solve a broad set of data analysis problems.

https://bedford-computing.co.uk/learning/wp-content/uploads/2015/10/Python-for-Data-Analysis.pdf
(link is also available in the Workshop_3/README.md)



What is data in this context?

- When I say “data”, what am I referring to exactly? The primary focus is on **structured data**, a deliberately vague term that encompasses many different common forms of data, such as
  - Multidimensional arrays (matrices)
  - Tabular or spreadsheet-like data in which each column may be a different type (string, numeric, date, or otherwise). This includes most kinds of data commonly stored in relational databases or tab- or comma-delimited text files
  - Multiple tables of data interrelated by key columns (what would be primary or foreign keys for a SQL user)
  - Evenly or unevenly spaced time series

Handling data using pd.Series and pd.DataFrame

pandas.Series: A pandas Series is a one-dimensional labeled array that can hold data of any type (integer, float, string, etc.). It consists of two main components: the index and the data. The index labels each element of the Series, allowing for easy and efficient data access. The data can be a NumPy array, a Python list, or a scalar value. With a Series, you can perform various operations such as indexing, slicing, arithmetic computations, and statistical analysis. It is a versatile data structure commonly used for representing a single column or row of data.

pandas.DataFrame: A pandas DataFrame is a two-dimensional labeled data structure that can hold data of different types (similar to a table or spreadsheet). It consists of rows and columns, where each column can have a different data type. DataFrames provide a powerful and flexible way to manipulate and analyze structured data. You can think of a DataFrame as a collection of Series objects, where each Series represents a column of data. DataFrames offer a wide range of functionalities, including data alignment, merging and joining, data filtering, grouping and aggregating, and handling missing values.

pd.Series

pd.Series

pd.Series; A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data.

The string representation of a Series displayed interactively shows the index on the left and the values on the right. If we do not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created.

pd.Series

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index. You can get the array representation and index object of the Series via its values and index attributes, respectively.

```
obj = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

print(obj.values)
print(obj.index)

>>>>>>>>>>>>>
[ 4  7 -5  3]
Index(['d', 'b', 'a', 'c'], dtype='object')
```

UPPSALA UNIVERSITET

pd.Series

Values can be obtained by index:

```
obj = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
print(obj[['c', 'a', 'd']])
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>
c    3
a   -5
d    4
dtype: int64
```

UPPSALA UNIVERSITET

pd.Series

Note that slicing is inclusive, in contrast to Numpy arrays or standard Python lists!

```
obj = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
print(obj['b':'c']])
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>
b    7
a   -5
c    3
dtype: int64
```

UPPSALA UNIVERSITET

pd.Series

Mathematical operations can be applied directly;

```
print(obj * 2)
```

```
d     8
b    14
a   -10
c     6
dtype: int64
```

pd.Series

Should you have data contained in a Python dict, you can create a Series by passing the dict:

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = Series(sdata)
```

Desired indices can be specified, values that are missing are inserted as NaN

```
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = Series(sdata, index=states)
```

UPPSALA UNIVERSITET

pd.Series

A critical Series feature for many applications is that it automatically aligns differently indexed data in arithmetic operations:

```
obj3

Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

```
obj4

Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

```
obj3 + obj4

California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

UPPSALA UNIVERSITET

pd.DataFrame

pandas.DataFrame

pandas provides rich data structures and functions designed to make working with structured data fast, easy, and expressive. It is, as you will see, one of the critical ingredients enabling Python to be a powerful and productive data analysis environment. The primary object in pandas that will be used in the book is the **DataFrame**, a twodimensional tabular, column-oriented data structure with both row and column labels:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

UPPSALA UNIVERSITET

pd.DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series (one for all sharing the same index). Compared with other such DataFrame-like structures you may have used before (like R's data.frame), row-oriented and column-oriented operations in DataFrame are treated roughly symmetrically. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays.

Pandas combines the high performance array-computing features of NumPy with the flexible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data. pandas is the primary tool that is used in the book.

Generating a pd.DataFrame

There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays.

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
 'year': [2000, 2001, 2002, 2001, 2002],
 'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

Generating a pd.DataFrame

As with Series, if you pass a column that isn't contained in data, it will appear with NA values in the result:

```
frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
    index=['one', 'two', 'three', 'four', 'five'])
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

Generating a pd.DataFrame

Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

```
frame2['debt'] = np.arange(5.)
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0

pd.DataFrame data retrieval

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
print(frame2['state'])
OR
print(frame2.state)
>>>>>>>>>>>>>
one         Ohio
two         Ohio
three       Ohio
four      Nevada
five      Nevada
Name: state, dtype: object
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0

UPPSALA UNIVERSITET

pd.DataFrame data retrieval using .loc and .iloc

loc is label-based, which means that you have to specify rows and columns based on their row and column labels.

iloc is integer position-based, so you have to specify rows and columns by their integer position values

```
print(frame2.loc['two', 'state']) >>>> ???

print(frame2.iloc[1,2]) >>>> ???
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0

pd.DataFrame data retrieval using .loc and .iloc

loc is label-based, which means that you have to specify rows and columns based on their row and column labels.

iloc is integer position-based, so you have to specify rows and columns by their integer position values

```
print(frame2.loc['two', 'state']) >>>> Ohio

print(frame2.iloc[1,2]) >>>> ???
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0

UPPSALA UNIVERSITET

pd.DataFrame data retrieval using .loc and .iloc

loc is label-based, which means that you have to specify rows and columns based on their row and column labels.

iloc is integer position-based, so you have to specify rows and columns by their integer position values

```
print(frame2.loc['two', 'state']) >>>> Ohio

print(frame2.iloc[1,2]) >>>> 1.7
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0

pd.DataFrame; selecting by slicing or boolean array

```
frame2[:2]
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0

```
frame2[frame2['pop']>2]
```

	year	state	pop	debt
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0

Some examples on larger datasets!

pandas.DataFrame; titanic data

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
886	0	2	male	27.0	0	0	13.0000	S	Second	man	True	NaN	Southampton	no	True
887	1	1	female	19.0	0	0	30.0000	S	First	woman	False	B	Southampton	yes	True
888	0	3	female	NaN	1	2	23.4500	S	Third	woman	False	NaN	Southampton	no	False
889	1	1	male	26.0	0	0	30.0000	C	First	man	True	C	Cherbourg	yes	True
890	0	3	male	32.0	0	0	7.7500	Q	Third	man	True	NaN	Queenstown	no	True

pandas.DataFrame; titanic data

```
titanic.age.mean()    >>>>> 29.70

titanic['fare'].sum() >>>>> 28693.9493

for class_nr in titanic['class'].unique().sort_values():
    survivors = titanic[titanic['class']==class_nr].survived.sum()
    passengers = len(titanic[titanic['class']==class_nr])
    survival_rate = survivors/passengers*100
    print('Survival rate in {} class of {:.1f}%'.format(class_nr, survival_rate))
>>>>>>>>>>>>>>
Survival rate in First class of 63.0%
Survival rate in Second class of 47.3%
Survival rate in Third class of 24.2%
```

UPPSALA UNIVERSITET

Women and children first?

```
len(titanic[titanic.adult_male==True])             >>>>> 537

titanic[titanic.adult_male==True].survived.sum()   >>>>> ???

len(titanic[titanic.adult_male==False])           >>>>> 354

titanic[titanic.adult_male==False].survived.sum() >>>>> ???
```

Women and children first!

```
len(titanic[titanic.adult_male==True])             >>>>> 537

titanic[titanic.adult_male==True].survived.sum()   >>>>> 88

len(titanic[titanic.adult_male==False])           >>>>> 354

titanic[titanic.adult_male==False].survived.sum() >>>>> 254
```

Health expectancy

What is the relation between spending and life expectancy?

Which country had the highest life expectancy in 2000?

Which country has the highest average life expectancy over the last decade?

	Year	Country	Spending_USD	Life_Expectancy
0	1970	Germany	252.311	70.6
1	1970	France	192.143	72.2
2	1970	Great Britain	123.993	71.9
3	1970	Japan	150.437	72.0
4	1970	USA	326.961	70.9
...	...	...	...	...
269	2020	Germany	6938.983	81.1
270	2020	France	5468.418	82.3
271	2020	Great Britain	5018.700	80.4
272	2020	Japan	4665.641	84.7
273	2020	USA	11859.179	77.0

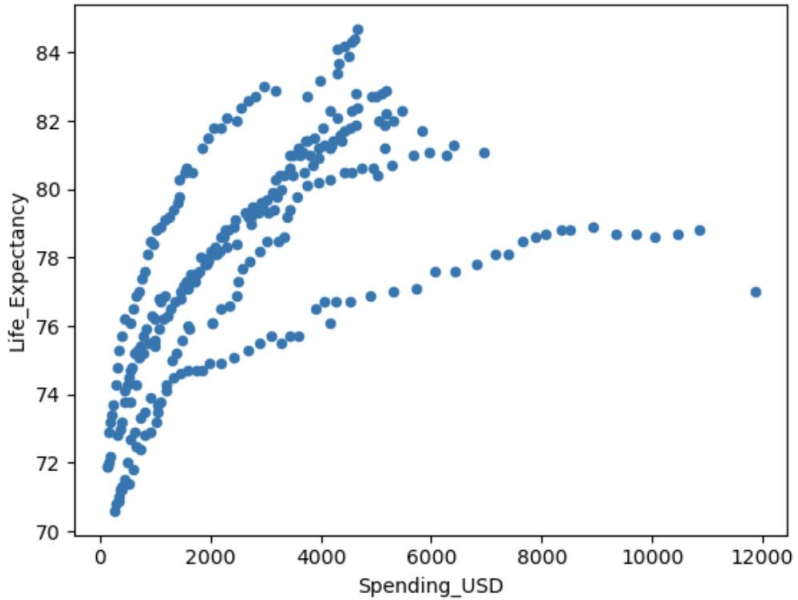
UPPSALA UNIVERSITET

Relationships between columns

What is the relation between spending and life expectancy?

The pd.DataFrame has a .plot() method!

```
health = sns.load_dataset("healthexp")
health.plot.scatter('Spending_USD', 'Life_Expectancy')
```



UPPSALA UNIVERSITET

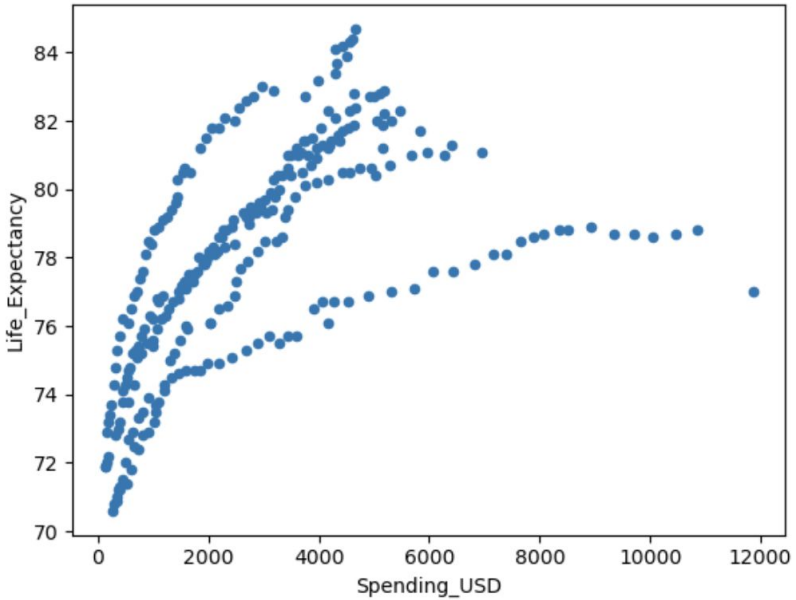
Relationships between columns quantified

What is the relation between spending and life expectancy?

pd.DataFrame allows you to correlate two columns very easily

using the pd.DataFrame.corr() method.

```
health.Spending_USD.corr(health.Life_Expectancy)
>>>>>
0.579
```



UPPSALA UNIVERSITET

Finding a maximum value

Which country had the highest life expectancy in 2000?

	Year	Country	Spending_USD	Life_Expectancy
148	2000	Canada	2450.593	79.1
149	2000	Germany	2895.533	78.2
150	2000	France	2687.530	79.2
151	2000	Great Britain	1897.202	77.9
152	2000	Japan	1847.786	81.2
153	2000	USA	4536.561	76.7

```
health[health['Year']==2000]
health[health['Year']==2000]['Life_Expectancy'].max()    >>>>> 81.2
health[health['Year']==2000]['Life_Expectancy'].argmax() >>>>> 152

health.iloc[152]
>>>>>>>>>>>>>>>>>>  Year                   2000
                    Country                Japan
                    Spending_USD        1847.786
                    Life_Expectancy         81.2
                    Name: 152, dtype: object
```

UPPSALA UNIVERSITET

Average values

Which country has the highest average life expectancy over the last decade?

```
life_exp = health.groupby('Country')['Life_Expectancy'].mean()
>>>>>
Country
Canada           81.77
France           82.48
Germany          80.85
Great Britain    81.13
Japan            83.68
USA              78.73
Name: Life_Expectancy, dtype: float64
```

UPPSALA UNIVERSITET

Reading in data to a pd.DataFrame

pandas features a number of functions for reading tabular data as a DataFrame object

- **read_csv** Load delimited data from a file, URL, or file-like object. Use comma as default delimiter
- **read_table** Load delimited data from a file, URL, or file-like object. Use tab ('\t') as default delimiter
- **read_fwf** Read data in fixed-width column format (that is, no delimiters) read_clipboard Version of
- **read_table** that reads data from the clipboard. Useful for converting tables from web pages

Other ways of storing and reading data

Data files

Data files are a common way to store and exchange data in various formats. In Python, there are several ways to store data in files, each with its own characteristics.

1. JSON (JavaScript Object Notation): JSON is a popular lightweight data interchange format that is easy to read and write for both humans and machines. Python provides the json module, which allows you to work with JSON files. You can use the json.dump() function to write Python objects into a JSON file, and json.load() function to read the data from a JSON file and convert it back into Python objects.
2. CSV (Comma-Separated Values): CSV files are a common format for storing tabular data, where each row represents a record and columns are separated by commas. Python's csv module provides functionality to read from and write to CSV files. You can use the csv.reader() function to read data from a CSV file, and csv.writer() function to write data to a CSV file.

Data files

3. Pickle: Pickle is a Python-specific binary protocol for serializing and deserializing Python objects. It allows you to store complex data structures, including custom classes, in a compact binary format. The pickle module in Python provides methods like pickle.dump() and pickle.load() to write and read objects respectively to and from pickle files.

4. HDF5 (Hierarchical Data Format): HDF5 is a file format commonly used for storing large and complex datasets. The h5py library in Python provides an interface to work with HDF5 files. It allows you to create datasets, groups, and attributes within the file, and perform efficient read and write operations on large datasets.

These are just a few examples of different ways to store data in Python. Depending on the specific requirements and characteristics of your data, you can choose the most appropriate file format and corresponding Python libraries to work with.

Scipy for statistical analysis

NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link.

SciPy SciPy is a collection of packages addressing a number of different standard problem domains in scientific computing. Here is a sampling of the packages included:

- scipy.integrate: numerical integration routines and differential equation solvers
- scipy.linalg: linear algebra routines and matrix decompositions extending beyond those provided in numpy.linalg.
- scipy.optimize: function optimizers (minimizers) and root finding algorithms
- scipy.signal: signal processing tools
- scipy.sparse: sparse matrices and sparse linear system solvers
- scipy.special: wrapper around SPECFUN, a Fortran library implementing many common

Scikit-learn, tensorflow and pytorch

A brief introduction