# Hello, World!

In today's lecture we are going to use the state-of-the-art tools for data and text mining in the Python ecosystem. The goal of the lecture is not to become proficient at it (one never becomes proficient at it 😉) but to get an idea of the basics (and not so basics) features.

Today we are going to go through three main topics.

1. Conda and Python environments
2. Plain Python vs Jupyter (Notebooks vs Lab) (Spoiler: check out what I am using)
3. Data Preparation using Numpy, Pandas, and SciPy
4. Creating Recommender Systems from scratch
5. ~~Neural Networks, Deep Learning, BERT, RoBERTa...~~ (please, no. DL is not the solution to all of our problems 😒)

Remember, it's not only important to listen to the lecture, but to practice. Take a topic you'd like, try out these libraries, the algorithms and techniques of the course, and become proficient at it (or begin working on your thesis project and you'll learn it by force 😅).

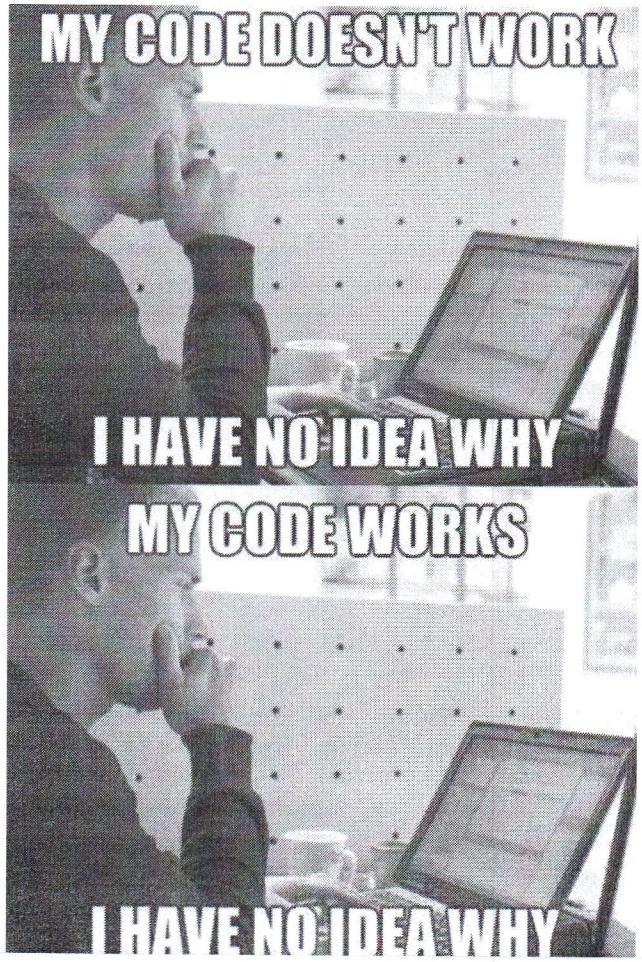
## whoami

I'm Fernando Pérez, a Ph.D. candidate at Polimi. I'm currently researching on Recommender Systems and that has worked "a bit" with Python, numpy, and Pandas. I also like programming memes and bad jokes 🤖

## Last reminders

1. The documentation of all the libraries are your friends. They just need care and attention.
2. If you missed something (or want to check these amazing notebooks again) I've got you covered. All these notebooks are available on GitHub (https://github.com).

Last, but not least, if this happens to you.



# Data Preparation

In this notebook we will have a look to the classic data science stack for Python to prepare our data for recommender

Specifically, we will have a look to the following libraries

1. NumPy
2. Pandas
3. SciPy

In order to show the capabilities of each library, we will use the same dataset across the examples and notebooks. You will see differences.

The first and last positions of the dataset are (this dataset is located on `data/mk-100k/u.data`):

Dataset structure:

| User Id | Item Id | Rating | Timestamp |
|---|---|---|---|
| 196 | 242 | 3 | 881250949 |
| 186 | 302 | 3 | 891717742 |
| 22 | 377 | 1 | 878887116 |
| 244 | 51 | 2 | 880606923 |
| 166 | 346 | 1 | 886397596 |

integer that says the date and time the rating has been made

## NumPy

NumPy is **THE** foundation of all the data science stack in Python. Most of the libraries are built on top of NumPy data structures. Make sure you take your time to understand them.

The basic data structure of NumPy is an `n-array`, i.e., an array of n dimensions with **LOTS** of methods to work with these arrays.

NumPy is really important in the field because it is **FAST**, most of the routines are implemented in C and you can get orders of magnitude faster when compared with their Python counterparts.

### Basic Concepts

In [1]:
```
import numpy as np # Convention, just memorize that np.<something> means a numpy method
```

In [2]:
```
one_dim_array = np.array([1,2,3,4])
one_dim_array
```

Out[2]:
```
array([1, 2, 3, 4])
```

In [3]:
```
two_dim_array = np.array([[1,2,3,4], [1000, 1001, 1002, 1004]])
two_dim_array
```

= matrices

Out[3]:
```
array([[   1,    2,    3,    4],
       [1000, 1001, 1002, 1004]])
```

In [4]:
```
three_dim_array = np.array([[[1,2,3,4], [1000, 1001, 1002, 1004]],
                            [[-1,-2,-3,-4], [-1000, -1001, -1002, -1003]],
                            [[1,-2,3,-4], [1000, -1001, 1002, -1003]],
                            [[-1,2,-3,4], [-1000, 1001, -1002, 1003]]])
three_dim_array
```

it's like an array of matrices: we have 3 dimensions: 2 for the matrices and 1 for the array (# matrices)

Out[4]:
```
array([[[   1,    2,    3,    4],
        [1000, 1001, 1002, 1004]],

       [[  -1,   -2,   -3,   -4],
        [-1000, -1001, -1002, -1003]],

       [[   1,   -2,    3,   -4],
        [1000, -1001, 1002, -1003]],

       [[  -1,    2,   -3,    4],
        [-1000, 1001, -1002, 1003]]])
```

### Load Data

In [9]:
```
data = np.loadtxt("data/ml-100k/u.data",
                  dtype=[('user_id', np.int32),
                         ('item_id', np.int32),
                         ('rating', np.int32),
                         ('timestamp', np.int64)],
                  delimiter="\t")
data
```

← this says that variables are divided with a tab

Out[9]:
```
array([(196,  242, 3, 881250949), (186,  302, 3, 891717742),
       ( 22,  377, 1, 878887116), ..., (276, 1090, 1, 874795795),
       ( 13,  225, 2, 882399156), ( 12,  203, 3, 879959583)],
      dtype=[('user_id', '<i4'), ('item_id', '<i4'), ('rating', '<i4'), ('timestamp', '<i8')])
```

We can see that the data itself is loaded as an array of tuples, which is pretty inconvenient 😒. I would like to have each column in a different array

In [12]:
```
user_ids, item_ids, ratings, timestamps = np.loadtxt("data/ml-100k/u.data",
                                                     dtype=[('user_id', np.int32), ('item_id', np.int32), ('rating', np.int32), ('timestamp', np.int64)],
                                                     delimiter="\t",
                                                     unpack=True)

print(f"{user_ids=}")
print(f"{item_ids=}")
print(f"{ratings=}")
```

← this leads to create arrays of columns

in this way we're creating 4 different arrays

```
print(f"{timestamps=}")
```

```
user_ids=array([196, 186,  22, ..., 276,  13,  12], dtype=int32)
item_ids=array([ 242,  302,  377, ..., 1090,  225,  203], dtype=int32)
ratings=array([3, 3, 1, ..., 1, 2, 3], dtype=int32)
timestamps=array([881250949, 891717742, 878887116, ..., 874795795, 882399156,
       879959583])
```

As you can see, we loaded the whole dataset with just one line of code (using `np.loadtxt`). I want to remark some things.

1. We defined our expected datatypes: **THIS IS IMPORTANT** NumPy can figure out what data types there are in the data (but if it just feels lazy it will try to load everything as a `np.float32`). Part of the huge speeds that we can obtain with NumPy and similar libraries is that we define with precision the data types that we're going to work with.
2. We defined the delimiter: Again, NumPy will try to figure out what delimiter we're using, but if we can help it, we'll get results faster (and with less probability of errors 😉).
3. We provided the unpack attribute: This tells NumPy to return each column in a separate array.

Now let's see what is `data`, how it looks like, and more.

shape tells how many items we have in each dimension

In [13]:
```
print(f"{one_dim_array.shape=}")
print(f"{two_dim_array.shape=}")
print(f"{three_dim_array.shape=}")
```

```
one_dim_array.shape=(4,)
two_dim_array.shape=(2, 4)
three_dim_array.shape=(4, 2, 4)
```

ndim tells the number of dimensions

In [16]:
```
print(f"{one_dim_array.ndim=}")
print(f"{two_dim_array.ndim=}")
print(f"{three_dim_array.ndim=}")
```

```
one_dim_array.ndim=1
two_dim_array.ndim=2
three_dim_array.ndim=3
```

In [14]:
```
# First, let's see the shape of the array, i.e., the size of each dimension
print(f"{data.shape=}")
print(f"{user_ids.shape=}")
print(f"{item_ids.shape=}")
print(f"{ratings.shape=}")
print(f"{timestamps.shape=}")
```

```
data.shape=(100000,)
user_ids.shape=(100000,)
item_ids.shape=(100000,)
ratings.shape=(100000,)
timestamps.shape=(100000,)
```

In [15]:
```
# Let's see the number of dimensions
print(f"{data.ndim=}")
print(f"{user_ids.ndim=}")
print(f"{item_ids.ndim=}")
print(f"{ratings.ndim=}")
print(f"{timestamps.ndim=}")
```

```
data.ndim=1
user_ids.ndim=1
item_ids.ndim=1
ratings.ndim=1
timestamps.ndim=1
```

In [18]:
```
# We can access positions on the array just as with Python
print(f"{data[1]=}")
print(f"{user_ids[0]=}")
print(f"{item_ids[0]=}")
print(f"{ratings[0]=}")
print(f"{timestamps[0]=}")
```

```
data[1]=(186, 302, 3, 891717742)
user_ids[0]=196
item_ids[0]=242
ratings[0]=3
timestamps[0]=881250949
```

In [19]:
```
# We can access slices of the array too.
print(f"{data[0:10]=}")
print(f"{user_ids[0:10]=}")
print(f"{item_ids[0:10]=}")
print(f"{ratings[0:10]=}")
print(f"{timestamps[0:10]=}")
```

```
data[0:10]=array([(196, 242, 3, 881250949), (186, 302, 3, 891717742),
       ( 22, 377, 1, 878887116), (244,  51, 2, 880606923),
       (166, 346, 1, 886397596), (298, 474, 4, 884182806),
       (115, 265, 2, 881171488), (253, 465, 5, 891628467),
       (305, 451, 3, 886324817), (  6,  86, 3, 883603013)],
      dtype=[('user_id', '<i4'), ('item_id', '<i4'), ('rating', '<i4'), ('timestamp', '<i8')])
user_ids[0:10]=array([196, 186,  22, 244, 166, 298, 115, 253, 305,   6], dtype=int32)
item_ids[0:10]=array([242, 302, 377,  51, 346, 474, 265, 465, 451,  86], dtype=int32)
ratings[0:10]=array([3, 3, 1, 2, 1, 4, 2, 5, 3, 3], dtype=int32)
timestamps[0:10]=array([881250949, 891717742, 878887116, 880606923, 886397596, 884182806,
       881171488, 891628467, 886324817, 883603013])
```

Until here, we've seen similarities with Python, by indexing and slicing these arrays. We've not seen anything so interesting, yet.

## Arithmetic Operations

One good thing about NumPy is that it integrates very well with Python. Using operators overload, you can have arithmetic operators for arrays that are intuitive and fast.

```
In [20]: print(f"{timestamps = }")

         # Adding 1 second to the timestamps (they're in ms).
         print(f"{timestamps + 1000 = }")

         # Substracting 2 minutes to the timestamps.
         print(f"{timestamps - 1000 * 60 * 2 = }")
```

it applies the operation to each cell of the array

```
timestamps = array([881250949, 891717742, 878887116, ..., 874795795, 882399156,
       879959583])
timestamps + 1000 = array([881251949, 891718742, 878888116, ..., 874796795, 882400156,
       879960583])
timestamps - 1000 * 60 * 2 = array([881130949, 891597742, 878767116, ..., 874675795, 882279156,
       879839583])
```

```
In [21]: # Normalizing ratings
         min_ratings = np.min(ratings)
         max_ratings = np.max(ratings)
         norm_ratings = (ratings - min_ratings) / (max_ratings - min_ratings)
         norm_ratings

         # Standardizing ratings
         mean_ratings = np.mean(ratings)
         std_ratings = np.std(ratings)
         standard_ratings = (ratings - mean_ratings) / std_ratings

         print(f"{norm_ratings = }")
         print(f"{standard_ratings = }")
```

minimum
maximum
mean
standard deviation

```
norm_ratings = array([0.5 , 0.5 , 0.  , ..., 0.  , 0.25, 0.5 ])
standard_ratings = array([-0.47070718, -0.47070718, -2.24743003, ..., -2.24743003,
       -1.35906861, -0.47070718])
```

## Boolean Indexing

What happens if I want to filter out those ratings lower than 4? Well, If you've used SQL, you know that a query for that will be something like this.

```
SELECT *
FROM data AS d
WHERE d.rating >= 4
```

select everything from data as long as rating ≥ 4

How can we do something like that in NumPy? Well, we can use boolean indexing. The basic notion is that we create a boolean array, and we will 'slice' the array using this boolean array.

**IMPORTANT: The boolean array must have the same shape than the original array.**

First we need to create the boolean array (the `WHERE` in SQL)

```
In [22]: where_clause = ratings >= 4
         where_clause
```

```
Out[22]: array([False, False, False, ..., False, False, False])
```

```
In [23]: where_clause.shape
```

```
Out[23]: (100000,)
```

Now that we have our boolean array, we just execute the `FROM` clause

```
In [24]: user_ids[where_clause]
```

if we put an array of boolean indeces it will put out only the elements corresponding to "True"

```
Out[24]: array([298, 253, 286, ..., 806, 676, 716], dtype=int32)
```

```
In [25]: user_ids[where_clause].shape
```

```
Out[25]: (55375,)
```

we removed almost half of the dataset! This is useful for cleaning data

What happened here?

Our boolean array went cell by cell of the `ratings` array asking if the cell had a value `greater or equal` than 4. If the condition held, then that cell in `where_clause` was set to `True`. Else, the cell was set to `False`.

Here we saw two important features of NumPy (and subsequent libraries). The power of filtering and selecting data based on boolean conditions, and that writing these boolean conditions is really easy.

What happens if I want those users with ids greater than 1000 and ratings higher than 4 OR item ids lower than 500 and ratings of 5?

An SQL query would look like this:

```
SELECT *
FROM data AS d
WHERE (d.rating >= 4 AND d.user_id >= 1000) OR (d.item_id < 500 AND d.rating = 5)
```

we can complicate the conditions

```
In [26]: where_clause = ((ratings >= 4) & (user_ids >= 1000)) | ((item_ids < 500) & (ratings == 5))
         where_clause
```

```
Out[26]: array([False, False, False, ..., False, False, False])
```

```
In [27]: where_clause.shape
```

```
Out[27]: (100000,)
```

```
In [28]: user_ids[where_clause]
```

```
Out[28]: array([253, 200, 122, ..., 650, 429, 716], dtype=int32)
```

Now that we have our boolean array, we just execute the `FROM` clause

```
In [22]: user_ids[where_clause].shape
Out[22]: (15903,)
```

We can also inline the condition. Depending on the length of the condition, it could be better to inline it or not

```
In [18]: user_ids[((ratings >= 4) & (user_ids >= 1000)) | ((item_ids < 500) & (ratings == 5))].shape
Out[18]: (15903,)
```

We introduced NumPy to be faster than normal python. How faster is it? Is it always faster?

## Important! You have to think in vectors

Most of the operations are highly optimized to be done in a vectorized way (e.g. no `for` loops).

Let's see an example of this. First we will prepare four arrays. Two arrays will be implemented as Python `lists` (`huge_array_1` & `huge_array_2`) and two arrays will be NumPy `arrays` (`numpy_huge_array_1` & `numpy_huge_array_2`). We will see the difference in speed by measuring two things:

1. Data structure efficiency
2. Algorithm "efficiency"

We're not going to do something uterly complicated here. We are just going to sum both arrays 😅

```
In [32]: huge_array_1 = [x for x in range(10000000)]
         huge_array_2 = [x*x for x in range(10000000)]

         numpy_huge_array_1 = np.array(huge_array_1)
         numpy_huge_array_2 = np.array(huge_array_2)
```

```
In [33]: def sum_naive_python_loop():
             a = []
             for arr1, arr2 in zip(huge_array_1, huge_array_2):
                 a.append(arr1 + arr2)

         def sum_naive_python_list_comprehension():
             a = [arr1 + arr2 for arr1, arr2 in zip(huge_array_1, huge_array_2)]

         def sum_naive_numpy_loop():
             a = []
             for arr1, arr2 in zip(numpy_huge_array_1, numpy_huge_array_2):
                 a.append(arr1 + arr2)

         def sum_naive_numpy_list_comprehension():
             a = [arr1 + arr2 for arr1, arr2 in zip(numpy_huge_array_1, numpy_huge_array_2)]

         def sum_numpy_approved():
             a = numpy_huge_array_1 + numpy_huge_array_2
```

Show time!

```
In [34]: %timeit sum_naive_python_loop()
         %timeit sum_naive_python_list_comprehension()
         %timeit sum_naive_numpy_loop()
         %timeit sum_naive_numpy_list_comprehension()
         %timeit sum_numpy_approved()
```

```
1.26 s ± 47 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
983 ms ± 3.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
3.24 s ± 42 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
3.08 s ± 132 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
23.9 ms ± 144 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

← slowest (numpy with python loop)

← fastest

## Pandas

Numpy is amazing, nobody will say the contrary. However, in our case it's like working on Assembly when we can work with C or Rust.

As you saw, the data is inside a CSV, stored in four columns: user_ids, item_ids, ratings, and timestamps (We will call these columns `variables`). In each row of the file we have an `observation`, a point in time that describes the four variables. When the data is shaped in a columnar way like this, it is advisable to use a Pandas DataFrame to handle the data.

We're going to do the same operations that we did with Numpy but with Pandas. Trust me, it will be for the better.

```
In [35]: import pandas as pd # Convention
```

```
In [36]: data_df = pd.read_csv("data/ml-100k/u.data",
                               names=["user_id", "item_id", "rating", "timestamp"],
                               dtype=[('user_id', np.int32), ('item_id', np.int32), ('rating', np.int32), ('timestamp', np.int64
         )],
                           delimiter="\t")
         data_df
```

we're adding names because the dataset doesn't have them

Out[36]:

| | user_id | item_id | rating | timestamp |
|---|---|---|---|---|
| **0** | 196 | 242 | 3 | 881250949 |
| **1** | 186 | 302 | 3 | 891717742 |
| **2** | 22 | 377 | 1 | 878887116 |
| **3** | 244 | 51 | 2 | 880606923 |
| **4** | 166 | 346 | 1 | 886397596 |
| **...** | ... | ... | ... | ... |

| | | | | |
|---|---|---|---|---|
| **99995** | 880 | 476 | 3 | 880175444 |
| **99996** | 716 | 204 | 5 | 879795543 |
| **99997** | 276 | 1090 | 1 | 874795795 |
| **99998** | 13 | 225 | 2 | 882399156 |
| **99999** | 12 | 203 | 3 | 879959583 |

100000 rows × 4 columns

What a difference! Apart from the case that Jupyter Notebooks have native support for Pandas DataFrames, we see that with a single function we can load the whole data (just as before) with a single function.

Similarly as with NumPy, we had to take into account some considerations:

1. We defined our expected datatypes: **THIS IS IMPORTANT** ~~NumPy~~ Pandas can figure out what data types there are in the data (but if it just feels lazy it will try to load everything as a `np.float32`). Part of the huge speeds that we can obtain with ~~NumPy~~ Pandas and similar libraries is that we define with precision the data types that we're going to work with.
2. We defined the delimiter: Again, ~~NumPy~~ Pandas will try to figure out what delimiter we're using, but if we can help it, we'll get results faster (and with less probability of errors 😨).

## Series

Here we are in front of the concept of `Series`. Pandas stores columns in what they call `Series` which are extensions of NumPy arrays (the underlying data structure is still a Numpy Array). We can access each `Series` using either an `.attribute` notation (only with valid Python variable names) or by using the `[dictionary_key]` notation (valid with any column name).

As you can imagine, DataFrames are just collections of Series that are somehow connected together.

```
In [37]: data_df.user_id
```

← we're accessing the column "user_id" in the dataset "data_df"

Note: if it was "user id" without underscore then we wouldn't be able to use this method, only the other (↓). We can use this method only with "valid" names.

```
Out[37]: 0        196
         1        186
         2         22
         3        244
         4        166
                 ...
         99995    880
         99996    716
         99997    276
         99998     13
         99999     12
         Name: user_id, Length: 100000, dtype: int32
```

```
In [38]: data_df["timestamp"]
```

← another way to access column

```
Out[38]: 0        881250949
         1        891717742
         2        878887116
         3        880606923
         4        886397596
                    ...
         99995    880175444
         99996    879795543
         99997    874795795
         99998    882399156
         99999    879959583
         Name: timestamp, Length: 100000, dtype: int64
```

## Basic operations

We still have basic operations and methods.

```
In [39]: # First, Let's see the shape of the array, i.e., the size of each dimension
         print(f"{data_df.shape=}")

         # Let's see the number of dimensions
         print(f"{data_df.ndim=}")
```

```
data_df.shape=(100000, 4)
data_df.ndim=2
```

## Indexing on Pandas

Pandas diverges in the way you access the rows or columns with respect to NumPy. For each DataFrame or Series object, we have an `index`. This `index` object is what we use to get rows under certain circumstances.

For instance, `data_df[0]` doesn't return what you'd expect.

```
In [40]: data_df[0]
```

✗ ← This is how we access the columns!

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/indexes/base.
py in get_loc(self, key, method, tolerance)
   2888             try:
-> 2889                 return self._engine.get_loc(casted_key)
   2890             except KeyError as err:

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()
```

```
pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 0

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
<ipython-input-40-8535285ae2b3> in <module>
----> 1 data_df[0]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/frame.py in _
_getitem__(self, key)
   2900             if self.columns.nlevels > 1:
   2901                 return self._getitem_multilevel(key)
-> 2902             indexer = self.columns.get_loc(key)
   2903             if is_integer(indexer):
   2904                 indexer = [indexer]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/indexes/base.
py in get_loc(self, key, method, tolerance)
   2889                 return self._engine.get_loc(casted_key)
   2890             except KeyError as err:
-> 2891                 raise KeyError(key) from err
   2892
   2893         if tolerance is not None:

KeyError: 0
```

Instead, you'll need to use the `iloc` attribute.

```
In [41]: data_df.iloc[0]
```

← This returns the element in the position in the squares.

```
Out[41]: user_id            196
         item_id            242
         rating               3
         timestamp    881250949
         Name: 0, dtype: int64
```

Do you remember how we retrieved the `timestamp` column before? This is what the `[]` operator works on Pandas, it returns a Series if the key matches a column name.

```
In [42]: data_df["invalid_column"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/indexes/base.
py in get_loc(self, key, method, tolerance)
   2888             try:
-> 2889                 return self._engine.get_loc(casted_key)
   2890             except KeyError as err:

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'invalid_column'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
<ipython-input-42-e97480887c95> in <module>
----> 1 data_df["invalid_column"]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/frame.py in _
_getitem__(self, key)
   2900             if self.columns.nlevels > 1:
   2901                 return self._getitem_multilevel(key)
-> 2902             indexer = self.columns.get_loc(key)
   2903             if is_integer(indexer):
   2904                 indexer = [indexer]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/indexes/base.
py in get_loc(self, key, method, tolerance)
   2889                 return self._engine.get_loc(casted_key)
   2890             except KeyError as err:
-> 2891                 raise KeyError(key) from err
   2892
   2893         if tolerance is not None:

KeyError: 'invalid_column'
```

```
In [43]: data_df["rating"]
```

```
Out[43]: 0        3
         1        3
         2        1
         3        2
         4        1
                 ..
         99995    3
         99996    5
         99997    1
         99998    2
         99999    3
         Name: rating, Length: 100000, dtype: int32
```

## Slicing on Pandas

Oh gosh, if I separated this into another section it's because it's a different topic, and you're not wrong.

Slicing works more or less the same as with Python and NumPy. But only for rows.

In [45]:
```
data_df[0:5]
```

Out[45]:

| | user_id | item_id | rating | timestamp |
|---|---|---|---|---|
| 0 | 196 | 242 | 3 | 881250949 |
| 1 | 186 | 302 | 3 | 891717742 |
| 2 | 22 | 377 | 1 | 878887116 |
| 3 | 244 | 51 | 2 | 880606923 |
| 4 | 166 | 346 | 1 | 886397596 |

In [46]:
```
data_df[0:5, 0:2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-52fddca02dc0> in <module>
----> 1 data_df[0:5, 0:2]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/frame.py in _
_getitem__(self, key)
   2900             if self.columns.nlevels > 1:
   2901                 return self._getitem_multilevel(key)
-> 2902             indexer = self.columns.get_loc(key)
   2903             if is_integer(indexer):
   2904                 indexer = [indexer]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/indexes/base.
py in get_loc(self, key, method, tolerance)
   2887             casted_key = self._maybe_cast_indexer(key)
   2888             try:
-> 2889                 return self._engine.get_loc(casted_key)
   2890             except KeyError as err:
   2891                 raise KeyError(key) from err

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice(0, 5, None), slice(0, 2, None))' is an invalid key
```

In [47]:
```
data_df[0:5, 2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-730341c9a6b6> in <module>
----> 1 data_df[0:5, 2]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/frame.py in _
_getitem__(self, key)
   2900             if self.columns.nlevels > 1:
   2901                 return self._getitem_multilevel(key)
-> 2902             indexer = self.columns.get_loc(key)
   2903             if is_integer(indexer):
   2904                 indexer = [indexer]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/indexes/base.
py in get_loc(self, key, method, tolerance)
   2887             casted_key = self._maybe_cast_indexer(key)
   2888             try:
-> 2889                 return self._engine.get_loc(casted_key)
   2890             except KeyError as err:
   2891                 raise KeyError(key) from err

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice(0, 5, None), 2)' is an invalid key
```

In [48]:
```
data_df[0:5, "rating"]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-35e494211832> in <module>
----> 1 data_df[0:5, "rating"]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/frame.py in _
_getitem__(self, key)
   2900             if self.columns.nlevels > 1:
   2901                 return self._getitem_multilevel(key)
-> 2902             indexer = self.columns.get_loc(key)
   2903             if is_integer(indexer):
   2904                 indexer = [indexer]

/usr/local/Caskroom/miniconda/base/envs/data-mining-and-text-mining/lib/python3.8/site-packages/pandas/core/indexes/base.
py in get_loc(self, key, method, tolerance)
   2887             casted_key = self._maybe_cast_indexer(key)
   2888             try:
-> 2889                 return self._engine.get_loc(casted_key)
   2890             except KeyError as err:
   2891                 raise KeyError(key) from err

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

```
pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice(0, 5, None), 'rating')' is an invalid key
```

```
In [50]: data_df[0:5]["rating"]
```

```
Out[50]: 0    3
         1    3
         2    1
         3    2
         4    1
         Name: rating, dtype: int32
```

## Operations on Columns

Remember when we added and substracted ratings, and normalized and standardized ratings? We're going to do the same.

BUT. We're going to add those values as columns in our DataFrame ~~and maybe make a mistake too in order to delete it~~

```
In [52]: data_df["timestamps_plus_one_sec"] = data_df["timestamp"] + 1000
         data_df["timestamps_minus_two_mins"] = data_df["timestamp"] + 1000 * 60 * 2

         data_df
```

← this adds a new column which is the column "timestamp" + (elementwise) 1000. The new column is called "timestamps_plus_one_sec"

Out[52]:

| | user_id | item_id | rating | timestamp | timestamps_plus_one_sec | timestamps_minus_two_mins |
|---|---|---|---|---|---|---|
| **0** | 196 | 242 | 3 | 881250949 | 881251949 | 881370949 |
| **1** | 186 | 302 | 3 | 891717742 | 891718742 | 891837742 |
| **2** | 22 | 377 | 1 | 878887116 | 878888116 | 879007116 |
| **3** | 244 | 51 | 2 | 880606923 | 880607923 | 880726923 |
| **4** | 166 | 346 | 1 | 886397596 | 886398596 | 886517596 |
| **...** | ... | ... | ... | ... | ... | ... |
| **99995** | 880 | 476 | 3 | 880175444 | 880176444 | 880295444 |
| **99996** | 716 | 204 | 5 | 879795543 | 879796543 | 879915543 |
| **99997** | 276 | 1090 | 1 | 874795795 | 874796795 | 874915795 |
| **99998** | 13 | 225 | 2 | 882399156 | 882400156 | 882519156 |
| **99999** | 12 | 203 | 3 | 879959583 | 879960583 | 880079583 |

100000 rows × 6 columns

```
In [53]: # Normalizing ratings
         min_ratings = data_df.rating.min()
         max_ratings = data_df.rating.max()
         data_df["norm_rating"] = (ratings - min_ratings) / (max_ratings - min_ratings)

         # Standardizing ratings
         mean_ratings = data_df.rating.mean()
         std_ratings = data_df.rating.std()
         data_df["standard_ratings"] = (ratings - mean_ratings) / std_ratings

         data_df
```

Out[53]:

| | user_id | item_id | rating | timestamp | timestamps_plus_one_sec | timestamps_minus_two_mins | norm_rating | standard_ratings |
|---|---|---|---|---|---|---|---|---|
| **0** | 196 | 242 | 3 | 881250949 | 881251949 | 881370949 | 0.50 | -0.470705 |
| **1** | 186 | 302 | 3 | 891717742 | 891718742 | 891837742 | 0.50 | -0.470705 |
| **2** | 22 | 377 | 1 | 878887116 | 878888116 | 879007116 | 0.00 | -2.247419 |
| **3** | 244 | 51 | 2 | 880606923 | 880607923 | 880726923 | 0.25 | -1.359062 |
| **4** | 166 | 346 | 1 | 886397596 | 886398596 | 886517596 | 0.00 | -2.247419 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **99995** | 880 | 476 | 3 | 880175444 | 880176444 | 880295444 | 0.50 | -0.470705 |
| **99996** | 716 | 204 | 5 | 879795543 | 879796543 | 879915543 | 1.00 | 1.306009 |
| **99997** | 276 | 1090 | 1 | 874795795 | 874796795 | 874915795 | 0.00 | -2.247419 |
| **99998** | 13 | 225 | 2 | 882399156 | 882400156 | 882519156 | 0.25 | -1.359062 |
| **99999** | 12 | 203 | 3 | 879959583 | 879960583 | 880079583 | 0.50 | -0.470705 |

100000 rows × 8 columns

Changes can also be applied to a column that already exists

```
In [54]: data_df["timestamps_minus_two_mins"] = 5
         data_df
```

← this replace all the elements of the column with a 5

Out[54]:

| | user_id | item_id | rating | timestamp | timestamps_plus_one_sec | timestamps_minus_two_mins | norm_rating | standard_ratings |
|---|---|---|---|---|---|---|---|---|
| **0** | 196 | 242 | 3 | 881250949 | 881251949 | 5 | 0.50 | -0.470705 |
| **1** | 186 | 302 | 3 | 891717742 | 891718742 | 5 | 0.50 | -0.470705 |
| **2** | 22 | 377 | 1 | 878887116 | 878888116 | 5 | 0.00 | -2.247419 |
| **3** | 244 | 51 | 2 | 880606923 | 880607923 | 5 | 0.25 | -1.359062 |
| **4** | 166 | 346 | 1 | 886397596 | 886398596 | 5 | 0.00 | -2.247419 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **99995** | 880 | 476 | 3 | 880175444 | 880176444 | 5 | 0.50 | -0.470705 |
| **99996** | 716 | 204 | 5 | 879795543 | 879796543 | 5 | 1.00 | 1.306009 |
| **99997** | 276 | 1090 | 1 | 874795795 | 874796795 | 5 | 0.00 | -2.247419 |
| **99998** | 13 | 225 | 2 | 882399156 | 882400156 | 5 | 0.25 | -1.359062 |
| **99999** | 12 | 203 | 3 | 879959583 | 879960583 | 5 | 0.50 | -0.470705 |

100000 rows × 8 columns

As I made a mistake, we should delete that column

In [55]:
```
del data_df["timestamps_minus_two_mins"]
data_df
```
← this is how we delete a column

Out[55]:

| | user_id | item_id | rating | timestamp | timestamps_plus_one_sec | norm_rating | standard_ratings |
|---|---|---|---|---|---|---|---|
| **0** | 196 | 242 | 3 | 881250949 | 881251949 | 0.50 | -0.470705 |
| **1** | 186 | 302 | 3 | 891717742 | 891718742 | 0.50 | -0.470705 |
| **2** | 22 | 377 | 1 | 878887116 | 878888116 | 0.00 | -2.247419 |
| **3** | 244 | 51 | 2 | 880606923 | 880607923 | 0.25 | -1.359062 |
| **4** | 166 | 346 | 1 | 886397596 | 886398596 | 0.00 | -2.247419 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **99995** | 880 | 476 | 3 | 880175444 | 880176444 | 0.50 | -0.470705 |
| **99996** | 716 | 204 | 5 | 879795543 | 879796543 | 1.00 | 1.306009 |
| **99997** | 276 | 1090 | 1 | 874795795 | 874796795 | 0.00 | -2.247419 |
| **99998** | 13 | 225 | 2 | 882399156 | 882400156 | 0.25 | -1.359062 |
| **99999** | 12 | 203 | 3 | 879959583 | 879960583 | 0.50 | -0.470705 |

100000 rows × 7 columns

## Boolean Indexing

As you can imagine, this also can be done on Pandas. At least to me, they look more like SQL than before, which I really enjoy

```
SELECT *
FROM data AS d
WHERE d.rating >= 4
```

**IMPORTANT: As before, the boolean array must have the same shape than the dataframe or series in terms of rows.**

First we need to create the boolean array (the `WHERE` in SQL)

In [56]:
```
where_clause = data_df.rating >= 4
where_clause
```

Out[56]:
```
0        False
1        False
2        False
3        False
4        False
         ...
99995    False
99996     True
99997    False
99998    False
99999    False
Name: rating, Length: 100000, dtype: bool
```

In [57]:
```
where_clause.shape
```

Out[57]:
```
(100000,)
```

Now that we have our boolean array, we just execute the `FROM` clause

In [58]:
```
data_df[where_clause]
```
← we can use it for the whole dataframe! (≠ numpy)

Out[58]:

| | user_id | item_id | rating | timestamp | timestamps_plus_one_sec | norm_rating | standard_ratings |
|---|---|---|---|---|---|---|---|
| **5** | 298 | 474 | 4 | 884182806 | 884183806 | 0.75 | 0.417652 |
| **7** | 253 | 465 | 5 | 891628467 | 891629467 | 1.00 | 1.306009 |
| **11** | 286 | 1014 | 5 | 879781125 | 879782125 | 1.00 | 1.306009 |
| **12** | 200 | 222 | 5 | 876042340 | 876043340 | 1.00 | 1.306009 |
| **16** | 122 | 387 | 5 | 879270459 | 879271459 | 1.00 | 1.306009 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **99988** | 421 | 498 | 4 | 892241344 | 892242344 | 0.75 | 0.417652 |
| **99989** | 495 | 1091 | 4 | 888637503 | 888638503 | 0.75 | 0.417652 |
| **99990** | 806 | 421 | 4 | 882388897 | 882389897 | 0.75 | 0.417652 |
| **99991** | 676 | 538 | 4 | 892685437 | 892686437 | 0.75 | 0.417652 |

the original numbers of rows remain

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **99996** | 716 | 204 | 5 | 879795543 | 879796543 | 1.00 | 1.306009 |

55375 rows × 7 columns

What happened here?

Our boolean array went row by row of the `data_df.rating` series asking if the cell had a value `greater or equal` than 4. If the condition held, then that cell in `where_clause` was set to `True`. Else, the cell was set to `False`.

Here we saw two important features of ~~NumPy~~ Pandas (and subsequent libraries). The power of filtering and selecting data based on boolean conditions, and that writing these boolean conditions is really easy.

What happens if I want those users with ids greater than 1000 and ratings higher than 4 OR item ids lower than 500 and ratings of 5?

An SQL query would look like this:

```
SELECT *
FROM data AS d
WHERE (d.rating >= 4 AND d.user_id >= 1000) OR (d.item_id > 5000 AND d.rating = 5)
```

more complex condition

each one has "data_df."

```
In [46]: where_clause = (((data_df.rating >= 4) & (data_df.user_id >= 1000))
                       | ((data_df.item_id < 500) & (data_df.rating == 5)))
         where_clause
```

```
Out[46]: 0        False
         1        False
         2        False
         3        False
         4        False
                  ...
         99995    False
         99996     True
         99997    False
         99998    False
         99999    False
         Length: 100000, dtype: bool
```

```
In [47]: where_clause.shape
```

```
Out[47]: (100000,)
```

```
In [45]: data_df[where_clause]
```

Out[45]:

| | user_id | item_id | rating | timestamp | timestamps_plus_one_sec | norm_rating | standard_ratings |
|---|---|---|---|---|---|---|---|
| **7** | 253 | 465 | 5 | 891628467 | 891629467 | 1.0 | 1.306009 |
| **12** | 200 | 222 | 5 | 876042340 | 876043340 | 1.0 | 1.306009 |
| **16** | 122 | 387 | 5 | 879270459 | 879271459 | 1.0 | 1.306009 |
| **26** | 38 | 95 | 5 | 892430094 | 892431094 | 1.0 | 1.306009 |
| **29** | 160 | 234 | 5 | 876861185 | 876862185 | 1.0 | 1.306009 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **99957** | 833 | 474 | 5 | 875122675 | 875123675 | 1.0 | 1.306009 |
| **99961** | 766 | 91 | 5 | 891310125 | 891311125 | 1.0 | 1.306009 |
| **99962** | 650 | 479 | 5 | 891372339 | 891373339 | 1.0 | 1.306009 |
| **99963** | 429 | 199 | 5 | 882386006 | 882387006 | 1.0 | 1.306009 |
| **99996** | 716 | 204 | 5 | 879795543 | 879796543 | 1.0 | 1.306009 |

15903 rows × 7 columns

Now that we have our boolean array, we just execute the `FROM` clause

We can also inline the condition. Depending on the length of the condition, it could be better to inline it or not

```
In [46]: data_df[((data_df.rating >= 4) & (data_df.user_id >= 1000)) | ((data_df.item_id < 500) & (data_df.rating == 5))]
```

Out[46]:

| | user_id | item_id | rating | timestamp | timestamps_plus_one_sec | norm_rating | standard_ratings |
|---|---|---|---|---|---|---|---|
| **7** | 253 | 465 | 5 | 891628467 | 891629467 | 1.0 | 1.306009 |
| **12** | 200 | 222 | 5 | 876042340 | 876043340 | 1.0 | 1.306009 |
| **16** | 122 | 387 | 5 | 879270459 | 879271459 | 1.0 | 1.306009 |
| **26** | 38 | 95 | 5 | 892430094 | 892431094 | 1.0 | 1.306009 |
| **29** | 160 | 234 | 5 | 876861185 | 876862185 | 1.0 | 1.306009 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **99957** | 833 | 474 | 5 | 875122675 | 875123675 | 1.0 | 1.306009 |
| **99961** | 766 | 91 | 5 | 891310125 | 891311125 | 1.0 | 1.306009 |
| **99962** | 650 | 479 | 5 | 891372339 | 891373339 | 1.0 | 1.306009 |
| **99963** | 429 | 199 | 5 | 882386006 | 882387006 | 1.0 | 1.306009 |
| **99996** | 716 | 204 | 5 | 879795543 | 879796543 | 1.0 | 1.306009 |

15903 rows × 7 columns

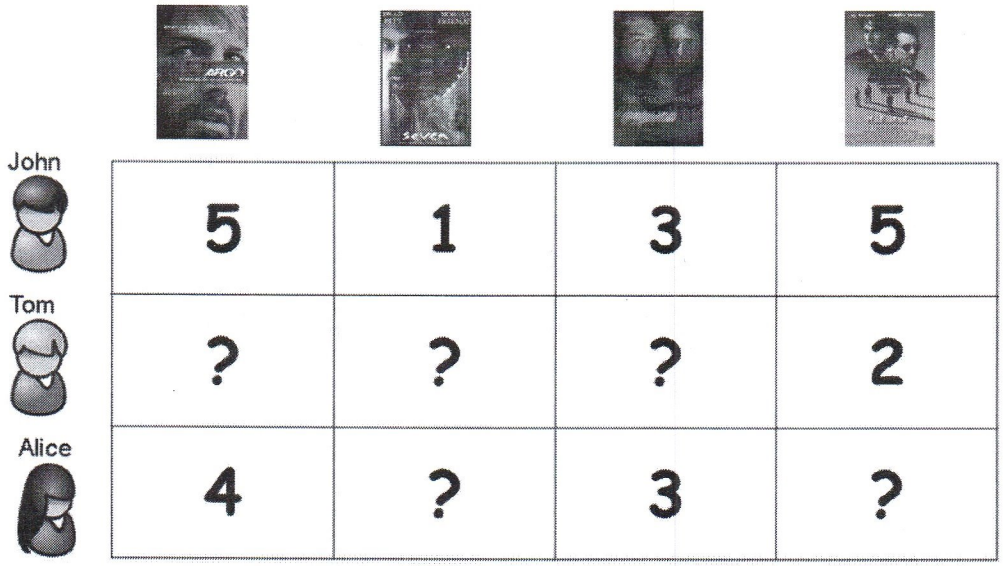
## Scipy

Scipy is another **core** library for data science. It provides functions and methods for several domains and calculations using NumPy arrays as base.

Previously, we only took each column in a separate array. However, this data can be organized in a matrix in the following way.

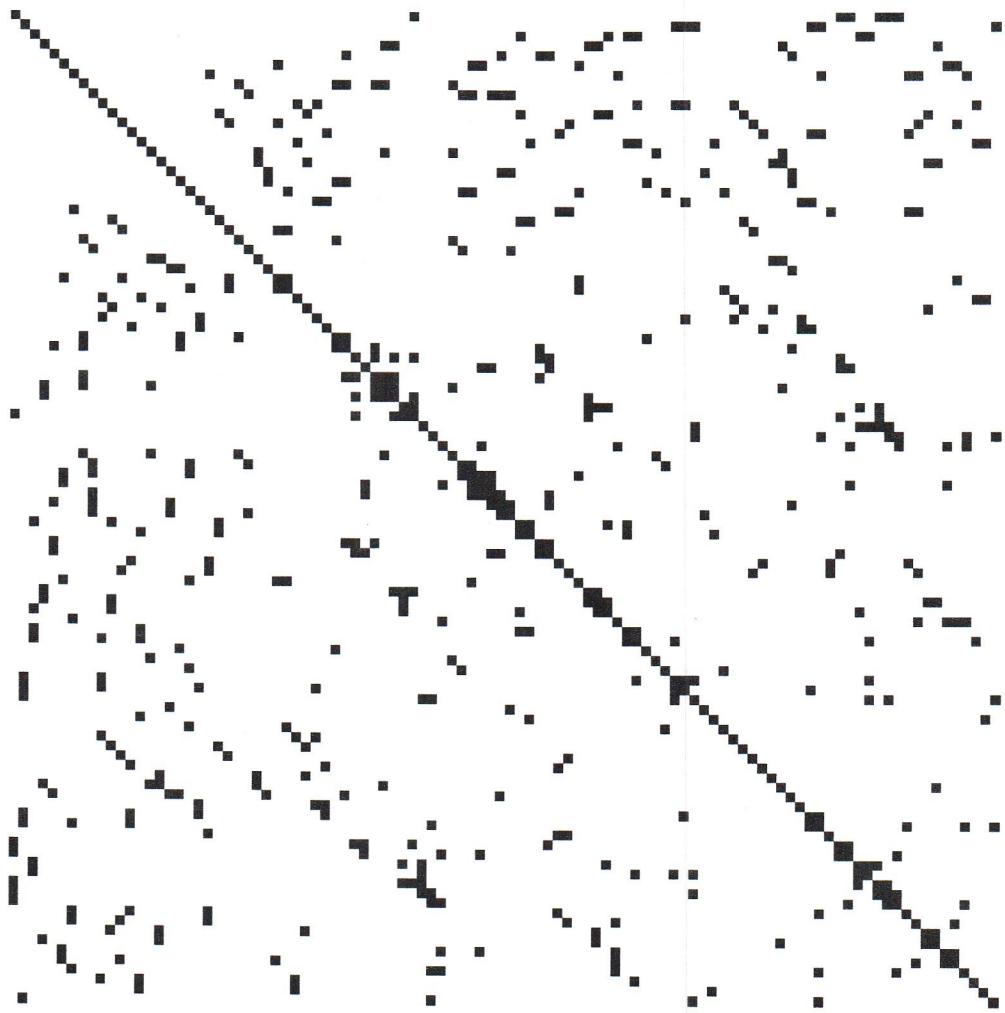
First, let's call the matrix $URM$. This matrix will have $nu$ rows and $ni$ columns. $nu$ is the number of users, and $ni$ is the number of items. For a user id $u$ and an item id $i$, the matrix cell $URM_{u,\ i}$ is equal to the rating that $u$ gave to $i$ ($r_{u,\ i}$).

Visually, the matrix looks as follows.




Not all users have rated all the items. Similarly, not all items have received a rating from every user. In fact, this matrix is **really sparse**. Which means that the number of empty spaces is **REALLY** high.

Visually,



Every black dot represents a rating, you can see that matrix is almost empty.

The good thing is that scipy contains a module for sparse matrices 😉. We will also calculate important features of the matrix.

In [59]:
```
from scipy import sparse
```

Before going into sparse matrices. We have a little problem. These are highly optimized data structures that are needed for **certain** operations. For instance, if we're going to do operations on the rows, then we use a **row** optimized sparse matrix.

In [60]:
```
# Row optimized sparse matrix
urm_csr = sparse.csr_matrix((ratings, (user_ids, item_ids)))

# Columns optimized
urm_csc = sparse.csc_matrix((ratings, (user_ids, item_ids)))
```

Operations on these type of matrices must be done with care, the time they take will increase **REALLY** a lot.

In [61]:
```
%timeit urm_csr + urm_csr
%timeit urm_csr + urm_csc
%timeit urm_csr[:, :150]
%timeit urm_csr[:150, :]
%timeit urm_csr.tocsc()
%timeit urm_csr.tocoo()
%timeit urm_csr.tolil()
```

```
379 µs ± 9.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
928 µs ± 22.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
232 µs ± 2.76 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
123 µs ± 3.47 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
554 µs ± 7.45 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
311 µs ± 5.68 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
4.55 ms ± 76.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [62]:
```
%timeit urm_csc + urm_csc
%timeit urm_csc + urm_csr
%timeit urm_csc[:, :150]
%timeit urm_csc[:150, :]
%timeit urm_csc.tocsr()
%timeit urm_csc.tocoo()
%timeit urm_csc.tolil()
```

```
405 µs ± 9.84 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
991 µs ± 32.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
140 µs ± 855 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
251 µs ± 7.15 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
516 µs ± 20.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
325 µs ± 10.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
5.3 ms ± 56 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

## Save the data for further use

In [63]:
```
sparse.save_npz("data/urm_csr.npz", matrix=urm_csr, compressed=True)
sparse.save_npz("data/urm_csc.npz", matrix=urm_csc, compressed=True)
```

# Fun Time! Application on Recommender Systems

We've seen basic operations for loading, filtering, and processing datasets using NumPy, Pandas, and Scipy.

Now we've arrived to the fun part. We will be implementing recommender systems using these libraries.

We will take into account recommendations at length $n$, i.e., each algorithm will return a list of $n$ items.

In [2]:
```
import numpy as np
import pandas as pd
from lenskit.algorithms import Recommender, item_knn, user_knn
from scipy import sparse
```

## Dataset Loading

In [3]:
```
urm_df = data_df = pd.read_csv("data/ml-100k/u.data",
                               names=["user", "item", "rating", "timestamp"],
                               dtype=[('user', np.int32), ('item', np.int32), ('rating', np.int32), ('timestamp', np.int64)],
                       delimiter="\t")
urm_csr = sparse.load_npz("data/urm_csr.npz",)
urm_csc = sparse.load_npz("data/urm_csc.npz",)
```

## Constants

In [5]:
```
recommendation_length = 10
(num_users, num_items), num_interactions = urm_csr.shape, urm_csr.nnz

rng = np.random.default_rng()

print(recommendation_length, num_users, num_items)
```

```
10 944 1683
```

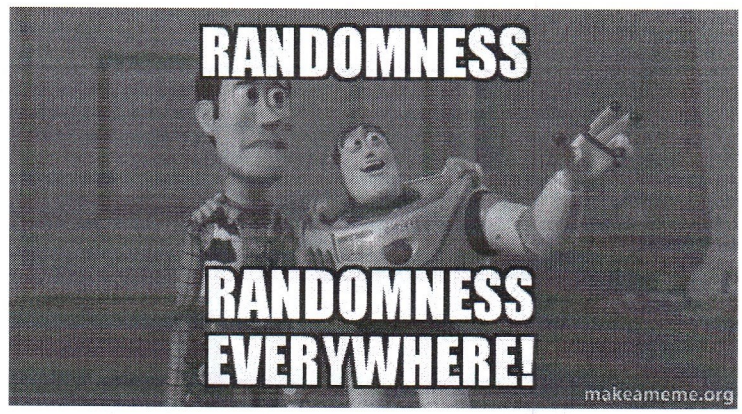
we recommend 10 items

In [7]:
```
density = urm_csr.nnz / (num_users * num_items)
density
```

Out[7]: 0.06294248567429025

(density of the matrix) that's why we're using tools for sparse matrices

## Recommender: Random




The name it's more or less self explanatory. But just to make it clear, it recommends $n$ random items of the catalog.

In [8]:
```
def random_item_recommender():
    return rng.permutation(np.arange(1,num_items + 1))[:recommendation_length]
```

we want 10 random elements without repetitions
⇒ we create a list of items ID's: item_1, item_2, ..
This is what **np.arange**
Then we do a permutation of it, so the elements position is changed.
Lastly, we take the first 10 elements of the permuted list.

## Recommender: Top Popular


This is one of the most basic recommender out there. It just recommend the most popular items to all users.

: we count how many ratings a movie has: more ratings → more popular

The interesting part is to count the number of times each item has been interacted.

When we have a CSC Sparse matrix, we can get the number of elements stored in each column using the `indptr` attribute Reference (https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csc_matrix.html#scipy.sparse.csc_matrix) and the `np.ediff1d` function.

More specifically, for a matrix like this:

```
[[1, 0, 4],
 [0, 0, 5],
 [2, 3, 6]]
```

We have that `indptr = np.array([0, 2, 3, 6])` (will always have one more column than the original number of columns), `indices = np.array([0, 2, 2, 0, 1, 2])`, and `data = np.array([1, 2, 3, 4, 5, 6])`. If we want to know where in the matrix we have a value, we do the following:

In [9]:
```
example_matrix = sparse.csc_matrix(np.array([[1, 0, 4], [0, 0, 5], [2, 3, 6]]))

for column_idx in range(3):
    row_ranges = example_matrix.indices[example_matrix.indptr[column_idx]:example_matrix.indptr[column_idx+1]]
    values = example_matrix.data[example_matrix.indptr[column_idx]:example_matrix.indptr[column_idx+1]]
    print(f"{column_idx = } - {row_ranges = } - {values = }")
```

```
column_idx = 0 - row_ranges = array([0, 2], dtype=int32) - values = array([1, 2])
column_idx = 1 - row_ranges = array([2], dtype=int32) - values = array([3])
column_idx = 2 - row_ranges = array([0, 1, 2], dtype=int32) - values = array([4, 5, 6])
```

Thanks to the way indptr is constructed, we can deduct that the number of non zero elements on each column $i$ is solely the difference between `indptr[i+1] - indptr[i]`. Moreover, after we've calculated the number of nnz elements in each column, we sort the indices with `np.argsort` and get the latest $n$ indices (argsort sorts in ascending order)

We add 1 at the end because ids begin at 1.

In [11]:
```
def top_popular_item_recommender():
    return np.argsort(np.ediff1d(urm_csc.indptr))[num_items - 1: num_items - 1 - recommendation_length:-1] + 1
```

## What are we recommending?

With this dataset we have the information of items, such as genres, directors, movie title, and more. This is the structure of the data.

| movie id | movie title | release date | video release date | IMDb URL | unknown | Action | Adventure | Animation | Children's | Comedy | Crime | Documentary | Drama | Fantasy | Film-Noir | Horror | Music |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

First, let's load that information into a DataFrame

In [12]:
```
item_features = pd.read_csv("data/ml-100k/u.item",
                            names=["movie_id", "movie_title", "release_date", "video_release_date",
                                   "IMDB URL", "unknown", "Action", "Adventure", "Animation", "Children's",
                                   "Comedy", "Crime", "Documentary", "Drama", "Fantasy",
                                   "Film-Noir", "Horror", "Musical", "Mystery", "Romance", "Sci-Fi",
                                   "Thriller", "War", "Western"],
                            delimiter="|",
                            parse_dates=True,
                            dtype={"movie_id": np.int32,
                                   "movie_title": "object",
                                   "video_release_date": "object",
                                   "IMDB URL": "object",
                                   "unknown": "object",
                                   "Action": np.bool,
                                   "Adventure": np.bool,
                                   "Animation": np.bool,
                                   "Children's": np.bool,
                                   "Comedy": np.bool,
                                   "Crime": np.bool,
                                   "Documentary": np.bool,
                                   "Drama": np.bool,
                                   "Fantasy": np.bool,
                                   "Film-Noir": np.bool,
                                   "Horror": np.bool,
                                   "Musical": np.bool,
                                   "Mystery": np.bool,
                                   "Romance": np.bool,
                                   "Sci-Fi": np.bool,
                                   "Thriller": np.bool,
                                   "War": np.bool,
                                   "Western": np.bool},
                            encoding='latin-1')
item_features
```

Out[12]:

| | movie_id | movie_title | release_date | video_release_date | IMDB URL | unknown | Action | Adventure | Animation | Children' |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | Toy Story (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Toy%20Story%2... | 0 | False | False | True | True |
| 1 | 2 | GoldenEye (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?GoldenEye%20(... | 0 | True | True | False | False |
| 2 | 3 | Four Rooms (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Four%20Rooms%... | 0 | False | False | False | False |
| 3 | 4 | Get Shorty (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Get%20Shorty%... | 0 | True | False | False | False |
| 4 | 5 | Copycat | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact? | 0 | False | False | False | False |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | (1995) | | | Copycat%20(1995) | | | | | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **1677** | 1678 | Mat' i syn (1997) | 06-Feb-1998 | NaN | http://us.imdb.com/M/title-exact?Mat%27+i+syn+... | 0 | False | False | False | False |
| **1678** | 1679 | B. Monkey (1998) | 06-Feb-1998 | NaN | http://us.imdb.com/M/title-exact?B%2E+Monkey+(... | 0 | False | False | False | False |
| **1679** | 1680 | Sliding Doors (1998) | 01-Jan-1998 | NaN | http://us.imdb.com/Title?Sliding+Doors+(1998) | 0 | False | False | False | False |
| **1680** | 1681 | You So Crazy (1994) | 01-Jan-1994 | NaN | http://us.imdb.com/M/title-exact?You%20So%20Cr... | 0 | False | False | False | False |
| **1681** | 1682 | Scream of Stone (Schrei aus Stein) (1991) | 08-Mar-1996 | NaN | http://us.imdb.com/M/title-exact?Schrei%20aus%... | 0 | False | False | False | False |

1682 rows × 24 columns

And now, we create a function that takes a recommendation list as input and returns the rows in the dataframe that match with those ids.

In [13]:
```
def get_item_features(item_ids):
    return item_features[item_features.movie_id.isin(item_ids)]
```

In [16]:
```
get_item_features(random_item_recommender())
```

Out[16]:

| | movie_id | movie_title | release_date | video_release_date | IMDB URL | unknown | Action | Adventure | Animation | Children' |
|---|---|---|---|---|---|---|---|---|---|---|
| **39** | 40 | To Wong Foo, Thanks for Everything! Julie Newm... | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?To%20Wong%20F... | 0 | False | False | False | False |
| **169** | 170 | Cinema Paradiso (1988) | 01-Jan-1988 | NaN | http://us.imdb.com/M/title-exact?Nuovo%20cinem... | 0 | False | False | False | False |
| **249** | 250 | Fifth Element, The (1997) | 09-May-1997 | NaN | http://us.imdb.com/M/title-exact?Fifth%20Eleme... | 0 | True | False | False | False |
| **281** | 282 | Time to Kill, A (1996) | 13-Jul-1996 | NaN | http://us.imdb.com/M/title-exact?Time%20to%20K... | 0 | False | False | False | False |
| **575** | 576 | Cliffhanger (1993) | 01-Jan-1993 | NaN | http://us.imdb.com/M/title-exact?Cliffhanger%2... | 0 | True | True | False | False |
| **659** | 660 | Fried Green Tomatoes (1991) | 01-Jan-1991 | NaN | http://us.imdb.com/M/title-exact?Fried%20Green... | 0 | False | False | False | False |
| **756** | 757 | Across the Sea of Time (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?Across%20The%... | 0 | False | False | False | False |
| **850** | 851 | Two or Three Things I Know About Her (1966) | 01-Jan-1966 | NaN | http://us.imdb.com/M/title-exact?Deux%20ou%20t... | 0 | False | False | False | False |
| **1271** | 1272 | Talking About Sex (1994) | 01-Jan-1994 | NaN | http://us.imdb.com/M/title-exact?Talking%20Abo... | 0 | False | False | False | False |
| **1526** | 1527 | Senseless (1998) | 09-Jan-1998 | NaN | http://us.imdb.com/M/title-exact?imdb-title-12... | 0 | False | False | False | False |

10 rows × 24 columns

In [17]:
```
get_item_features(top_popular_item_recommender())
```

Out[17]:

| | movie_id | movie_title | release_date | video_release_date | IMDB URL | unknown | Action | Adventure | Animation | Children's |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 2 | GoldenEye (1995) | 01-Jan-1995 | NaN | http://us.imdb.com/M/title-exact?GoldenEye%20(... | 0 | True | True | False | False |
| **50** | 51 | Legends of the Fall (1994) | 01-Jan-1994 | NaN | http://us.imdb.com/M/title-exact?Legends%20of%... | 0 | False | False | False | False |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **100** | 101 | Heavy Metal (1981) | 08-Mar-1981 | NaN | http://us.imdb.com/M/title-exact?Heavy%20Metal... | 0 | True | True | True | False |
| **121** | 122 | Cable Guy, The (1996) | 14-Jun-1996 | NaN | http://us.imdb.com/M/title-exact?Cable%20Guy,%... | 0 | False | False | False | False |
| **181** | 182 | GoodFellas (1990) | 01-Jan-1990 | NaN | http://us.imdb.com/M/title-exact?GoodFellas%20... | 0 | False | False | False | False |
| **258** | 259 | George of the Jungle (1997) | 01-Jan-1997 | NaN | http://us.imdb.com/M/title-exact?George+of+the... | 0 | False | False | False | True |
| **286** | 287 | Marvin's Room (1996) | 18-Dec-1996 | NaN | http://us.imdb.com/M/title-exact?Marvin's%20Ro... | 0 | False | False | False | False |
| **288** | 289 | Evita (1996) | 25-Dec-1996 | NaN | http://us.imdb.com/M/title-exact?Evita%20(1996) | 0 | False | False | False | False |
| **294** | 295 | Breakdown (1997) | 02-May-1997 | NaN | http://us.imdb.com/M/title-exact?Breakdown%20% | 0 | True | False | False | False |