The data class

Pietro Berkes & Verjinia Metodieva

Things one thinks about when thinking about data

Processing	Storage	Reproducibility and collaboration
• Efficient processing (no for-loops!) • Organizing data so that analyses are easy	• Size • Access ease • Access time	• Versioning • Lineage tracing (which script / other data was used to generate this?) • Ease of sharing

Things one thinks about when thinking about data

Processing

- Efficient processing (no for-loops!)
- Organizing data so that analyses are easy

Storage

- Size
- Access ease
- Access time

Reproducibility and collaboration

- Versioning
- Lineage tracing (which script / other data was used to generate this?)
- Ease of sharing

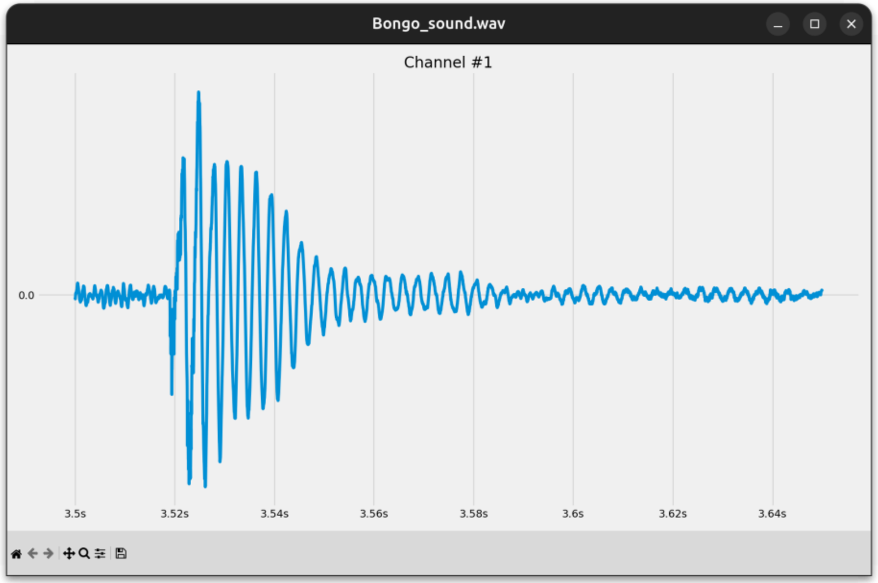
Hands-on

What data structure would you use to represent…

Hands-on

What data structure would you use to represent…

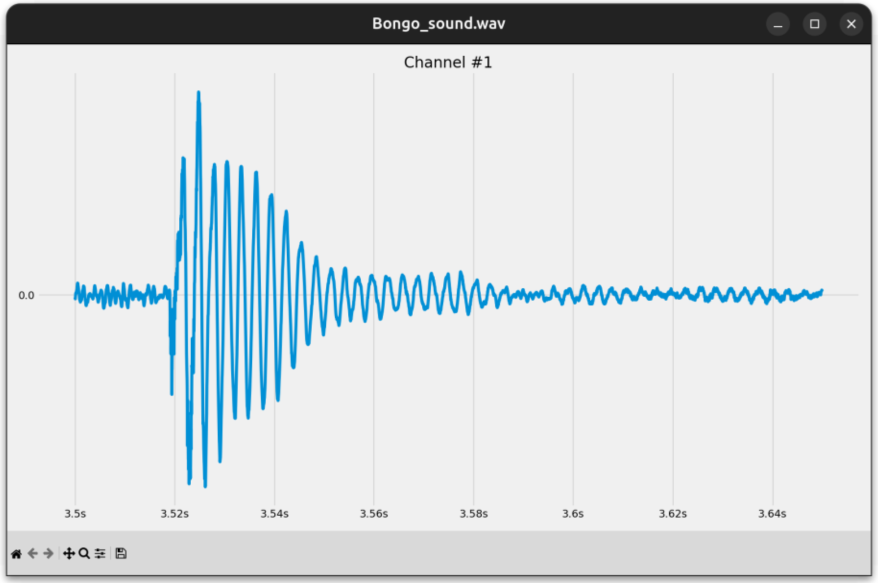
A sound wave?



Hands-on

What data structure would you use to represent...

A sound wave?



NumPy array

```
In [6]: sound_data
Out[6]: array([0.66709183, 0.55973494, 0.95416669, 0.60810949, 0.05188879,
       0.58619063, 0.25555136, 0.72451477, 0.2646681 , 0.08694215,
       0.75592186, 0.67261696, 0.62847452, 0.06232598, 0.20549438,
       0.11718457, 0.25184725, 0.48625729, 0.8103058 , 0.18100915,
       0.81113341, 0.62055231, 0.9046905 , 0.56664205, 0.73235338,
       0.74382869, 0.64856368, 0.80644398, 0.46199345, 0.78516632,
       0.91298397, 0.48290914, 0.20847714, 0.99162659, 0.26374781,
       0.3602381 , 0.07173351, 0.8584085 , 0.32248766, 0.39167573,
       0.67944923, 0.00930429, 0.21714217, 0.58810089, 0.17668711,
       0.57444803, 0.25760187, 0.43785728, 0.39119371, 0.68268063,
       0.95954499, 0.45934239, 0.03616905, 0.23896063, 0.61872801,
       0.76332531, 0.96272817, 0.57169277, 0.50225193, 0.01361629,
       0.15357459, 0.8057233 , 0.0642748 , 0.95013941, 0.38712684,
       0.97231498, 0.20261775, 0.74184693, 0.26629893, 0.84672705,
       0.67662718, 0.96055977, 0.64942314, 0.66487937, 0.86867536,
       0.40815661, 0.1139344 , 0.95638066, 0.87436447, 0.18407227,
       0.64457074, 0.19233097, 0.24012179, 0.90399279, 0.39093908,
       0.26389161, 0.97537645, 0.14209784, 0.75261696, 0.10078122,
       0.87468408, 0.77990102, 0.92983283, 0.45841805, 0.61470669,
       0.87939755, 0.09266009, 0.41177209, 0.46973971, 0.43152144])
```

Hands-on

What data structure would you use to represent…

Phone book entries?

Hands-on

What data structure would you use to represent…

Phone book entries?

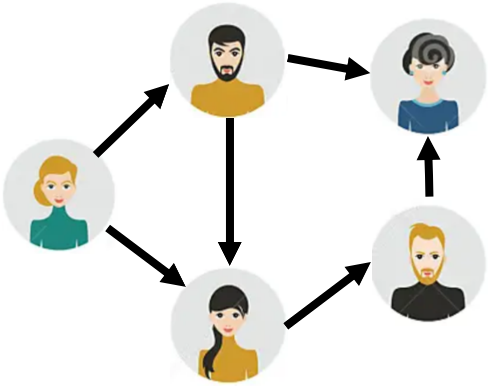
Pandas DataFrame

first_name	last_name	phone_nr	address	ZIP	city
John	Doe	555-1234	123 Maple St	12345	Springfield
Jane	Smith	555-5678	456 Oak St	67890	Rivertown
Alice	Johnson	555-8765	789 Pine St	54321	Lakeside
Bob	Brown	555-4321	321 Birch St	09876	Hilltop
Emma	Davis	555-7890	654 Elm St	11223	Greendale

Hands-on

What data structure would you use to represent...

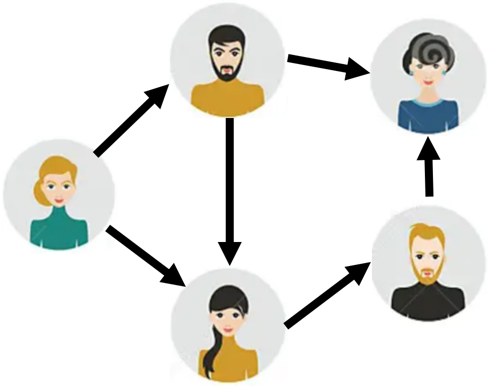
Friendship relations?



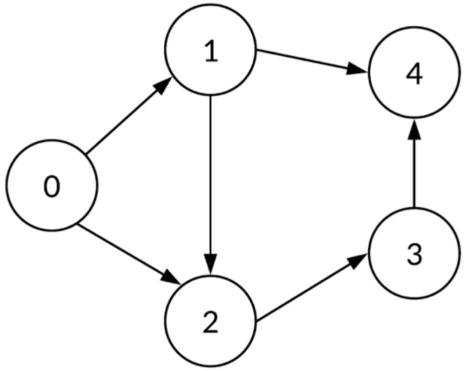
Hands-on

What data structure would you use to represent...

Friendship relations?



Graph



Implemented as

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

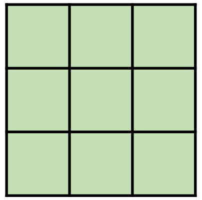
Adjacency matrix (array)

```
A_dict = {
    '0':[1,2],
    '1':[2],
    '2':[3],
    '3':[4],
    '4':[]
}
```

Dictionary

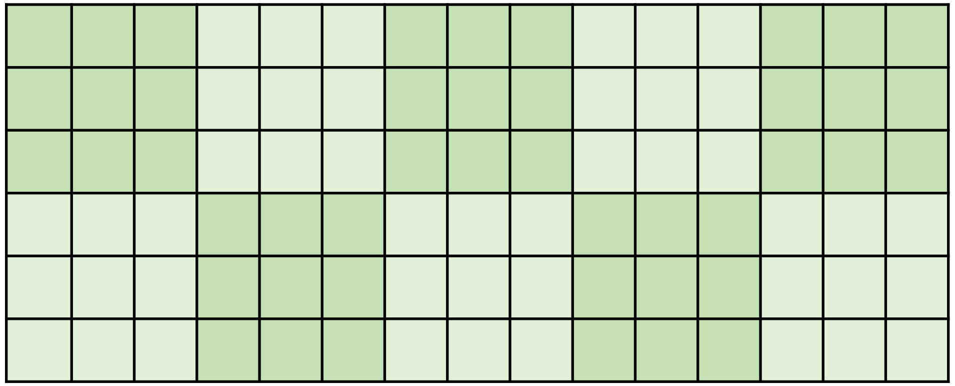
You develop your code on a small data set, how is it going to scale to the complete data set?

Development data



N data points,
Processing time T

Real data

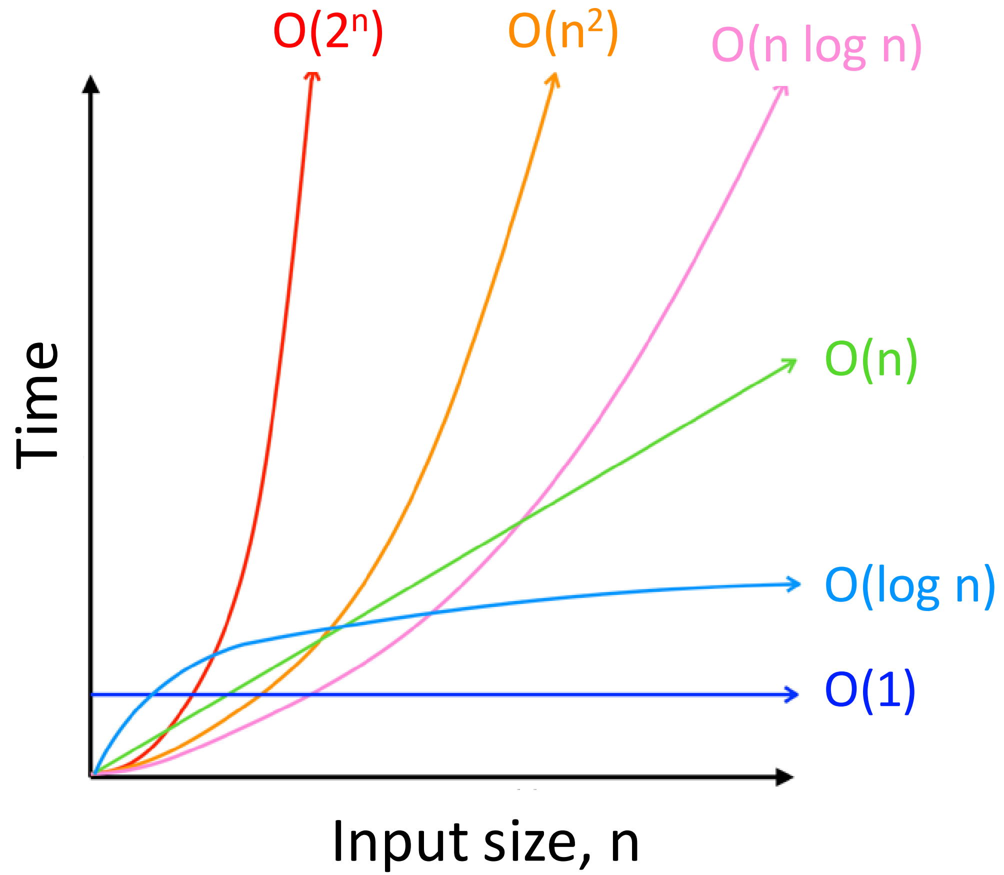


10x N data points
Processing time -> ?

We're interested in orders of magnitude

How performance scales: big-O

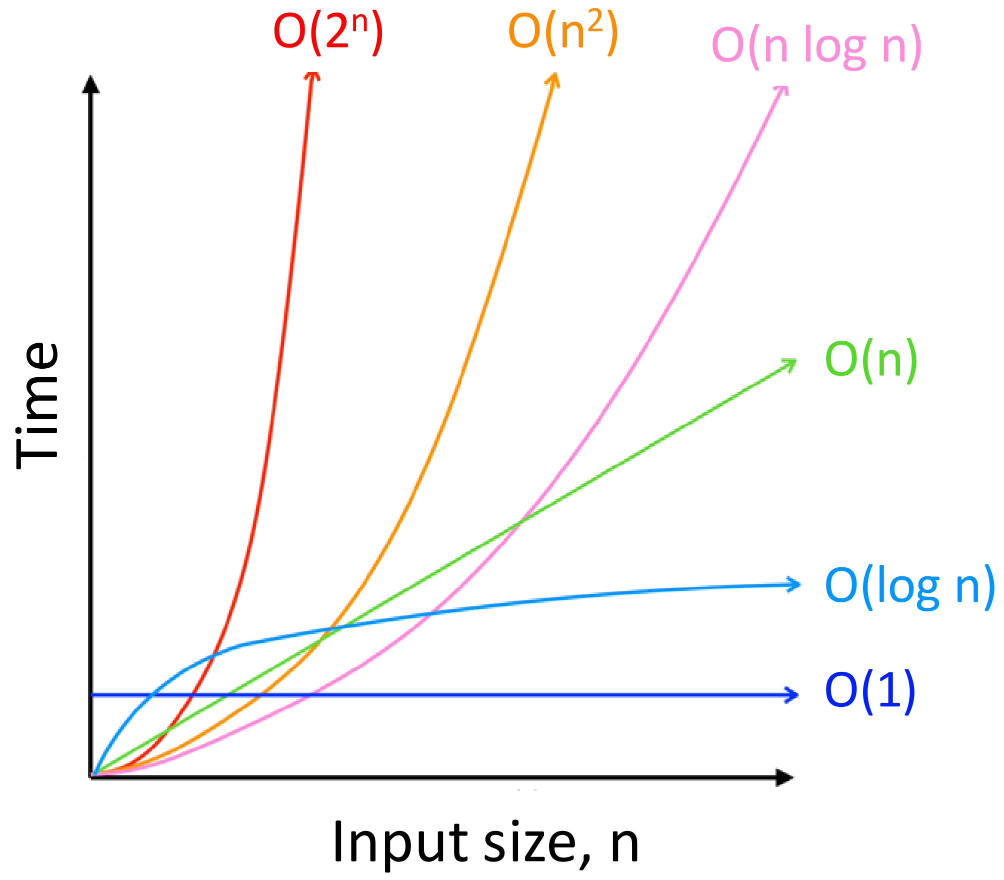
Big-O class	What we call it	Time increase, when data increases 10x
O(1)	constant	1x time
O(n)	linear	10x time
O(n^2)	quadratic	100x time
O(n * log n)	linearithmic	~10-20x time
O(log n)	logarithmic	~1-2x time



Hands-on: Operations on lists

Big-O class	What we call it	Time increase, when data increases 10x
O(1)	constant	1x time
O(n)	linear	10x time
O(n²)	quadratic	100x time
O(n * log n)	linearithmic	~10-20x time
O(log n)	logarithmic	~1-2x time

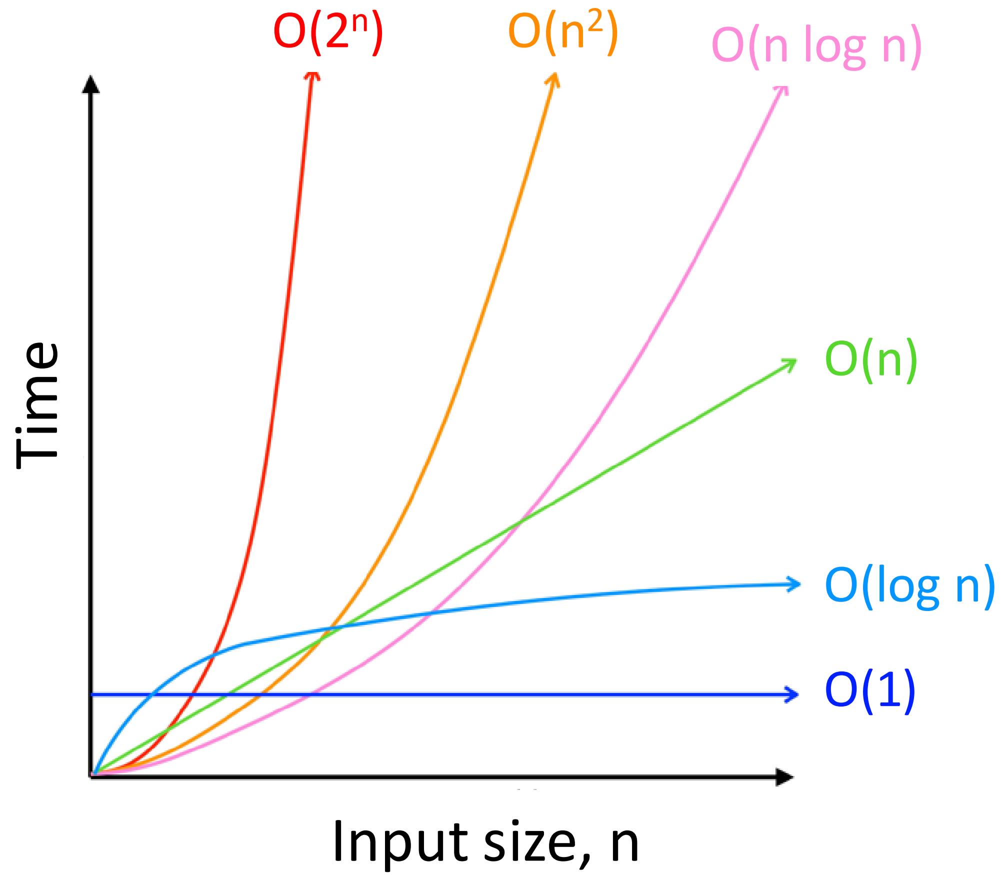
Big-O class	Operation on lists that scales this way
O(1)	
O(n)	
O(n²)	
O(n * log n)	
O(log n)	



Hands-on: Operations on lists

Big-O class	What we call it	Time increase, when data increases 10x
O(1)	constant	1x time
O(n)	linear	10x time
$O(n^2)$	quadratic	100x time
O(n * log n)	linearithmic	~10-20x time
O(log n)	logarithmic	~1-2x time

Big-O class	Operation on lists that scales this way
O(1)	Getting an element by its index
O(n)	Summing elements in list
$O(n^2)$	Computing distance between all pairs of elements in the list
O(n * log n)	Sorting the list
O(log n)	Searching an element in a sorted list



Example: Find common words

Given two lists of words, extract all the words that are in common

```
words1 = ['apple', 'orange', 'banana', 'melon', 'peach']
words2 = ['orange', 'kiwi', 'avocado', 'apple', 'banana']
```

Expected result: `['apple', 'orange', 'banana']`

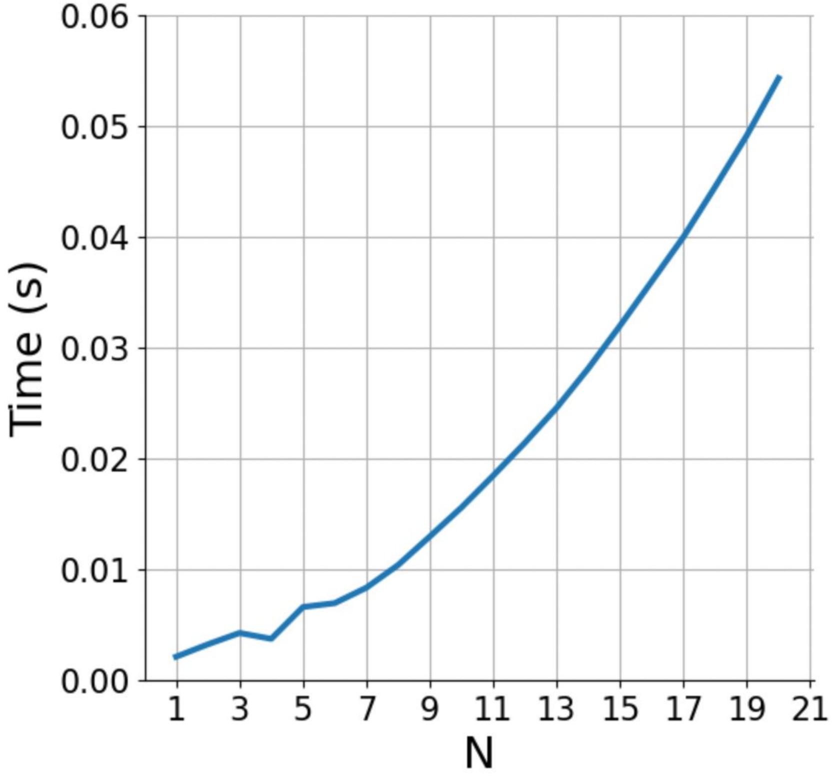
Implementation with two for-loops

```python
words1 = ['apple', 'orange', 'banana', 'melon', 'peach']
words2 = ['orange', 'kiwi', 'avocado', 'apple', 'banana']

common = []
for w in words1:
    if w in words2:
        common.append(w)
```

What is the big-O complexity of this implementation?

Implementation with two for-loops

```python
words1 = ['apple', 'orange', 'banana', 'melon', 'peach']
words2 = ['orange', 'kiwi', 'avocado', 'apple', 'banana']

common = []
for w in words1:            # O(N)
    if w in words2:           # O(N)
        common.append(w)  # O(1)
```



What is the big-O complexity of this implementation?

N * N ~ **O(N^2)**

Implementation with sorted lists

```
words1 = ['apple', 'orange', 'banana', 'melon', 'peach']
words2 = ['orange', 'kiwi', 'avocado', 'apple', 'banana']

words1 = sorted(words1) # ['apple', 'banana', 'melon', 'orange', 'peach']
words2 = sorted(words2) # ['apple', 'avocado', 'banana', 'kiwi', 'orange']

common = []
idx2 = 0
for w in words1:
    while idx2 < len(words2) and words2[idx2] < w:
        idx2 += 1

    if idx2 >= len(words2):
        break

    if words2[idx2] == w:
        common.append(w)
```

What is the big-O complexity of this implementation?

Implementation with sorted lists

```python
words1 = ['apple', 'orange', 'banana', 'melon', 'peach']
words2 = ['orange', 'kiwi', 'avocado', 'apple', 'banana']

words1 = sorted(words1)     # O(N * log(N))
words2 = sorted(words2)     # O(N * log(N))

common = []
idx2 = 0
for w in words1:                                      # O(N)
    while idx2 < len(words2) and words2[idx2] < w:   # O(N) in total
        idx2 += 1

    if idx2 >= len(words2): # O(1)
        break

    if words2[idx2] == w:   # O(1)
        common.append(w)
```

What is the big-O complexity of this implementation?

2 * (N * log(N)) + 2 * N ~ **O(N log N)**

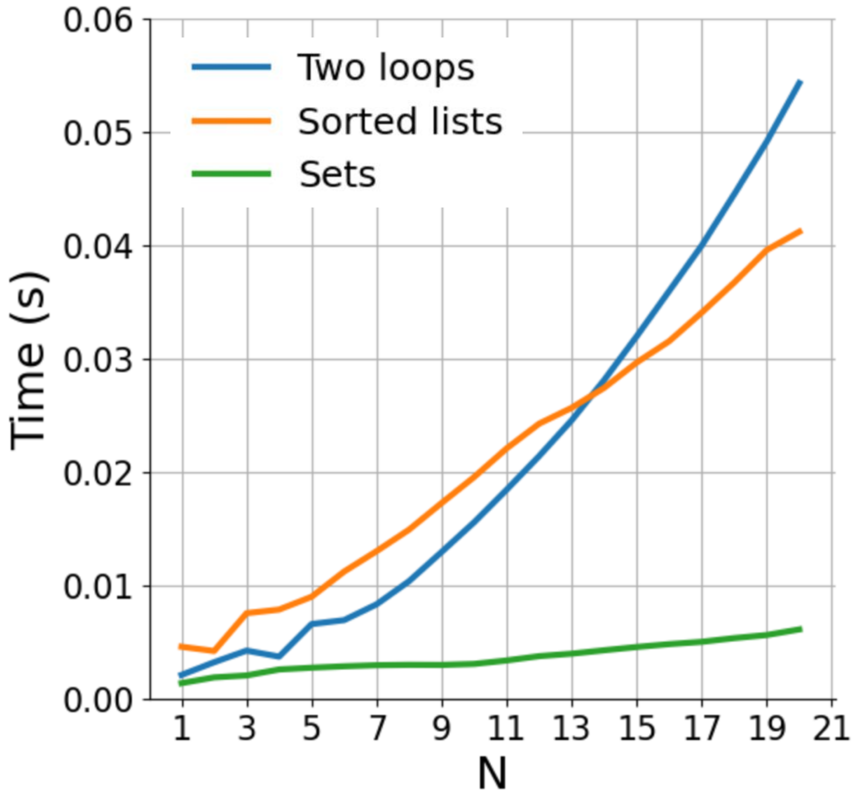
Implementation with sets

```
words1 = ['apple', 'orange', 'banana', 'melon', 'peach']
words2 = ['orange', 'kiwi', 'avocado', 'apple', 'banana']

words2 = set(words2)

common = []
for w in words1:
    if w in words2:
        common.append(w)
```

What is the big-O complexity of this implementation?

Implementation with sets

```python
words1 = ['apple', 'orange', 'banana', 'melon', 'peach']
words2 = ['orange', 'kiwi', 'avocado', 'apple', 'banana']

words2 = set(words2)       # O(N)

common = []
for w in words1:           # O(N)
    if w in words2:            # O(1)
        common.append(w)       # O(1)
```



What is the big-O complexity of this implementation?

N + N ~ **O(N)**

Basic reference sheet about Python data structures

Lists: collection of ordered, arbitrary data

Getting an element by index	O(1)
Appending	O(1)
Inserting an element at index	O(n)
Sorting	O(n log n)
Finding an element (e.g., “if element in my_list: …”)	O(n)

Dictionaries (“hashmaps”)

Inserting	O(1)
Finding a value by key (e.g., “if element in my_dict: …”)	O(1)

Sets: it’s dictionaries without values

Inserting	O(1)
Finding a value by key (e.g., “if element in my_set: …”)	O(1)

See also: https://wiki.python.org/moin/TimeComplexity

Hands-on

Exercise
`exercises/match_tarots`

- Open the notebook `match_tarots`, and follow the instructions!

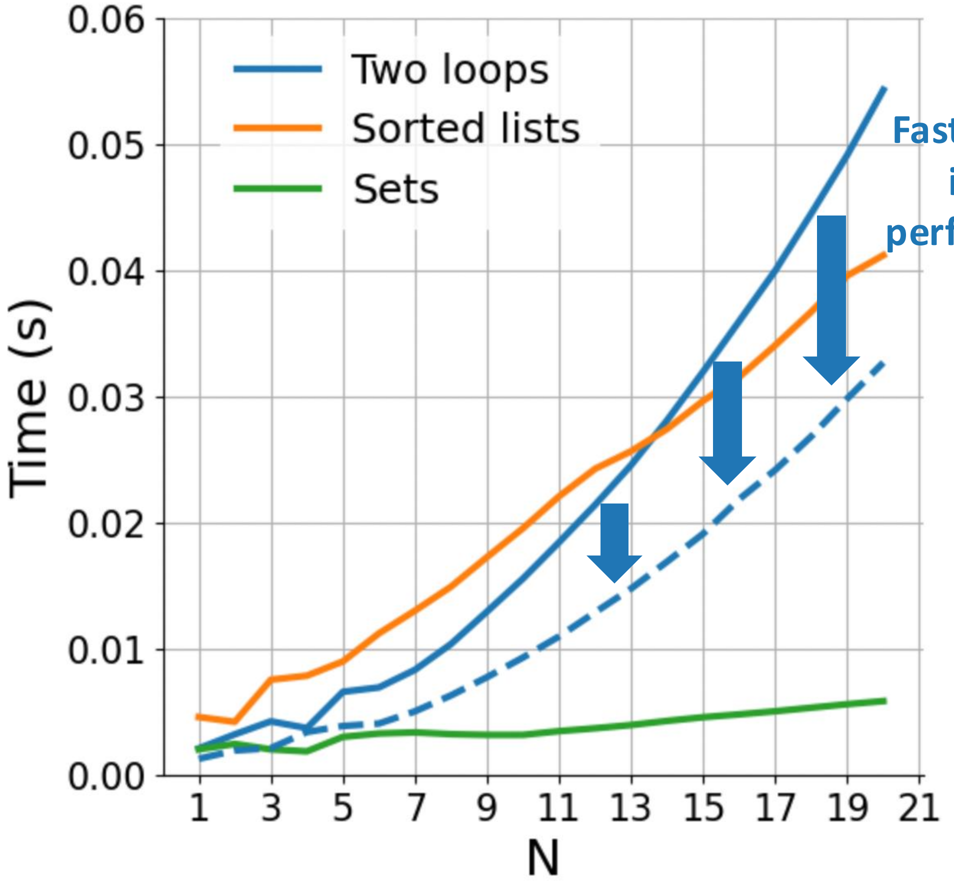
- Submit a PR for Issue #7 on GitHub

How does rewriting in C change the performance?

(rewriting in C, parallelization; same algorithm)

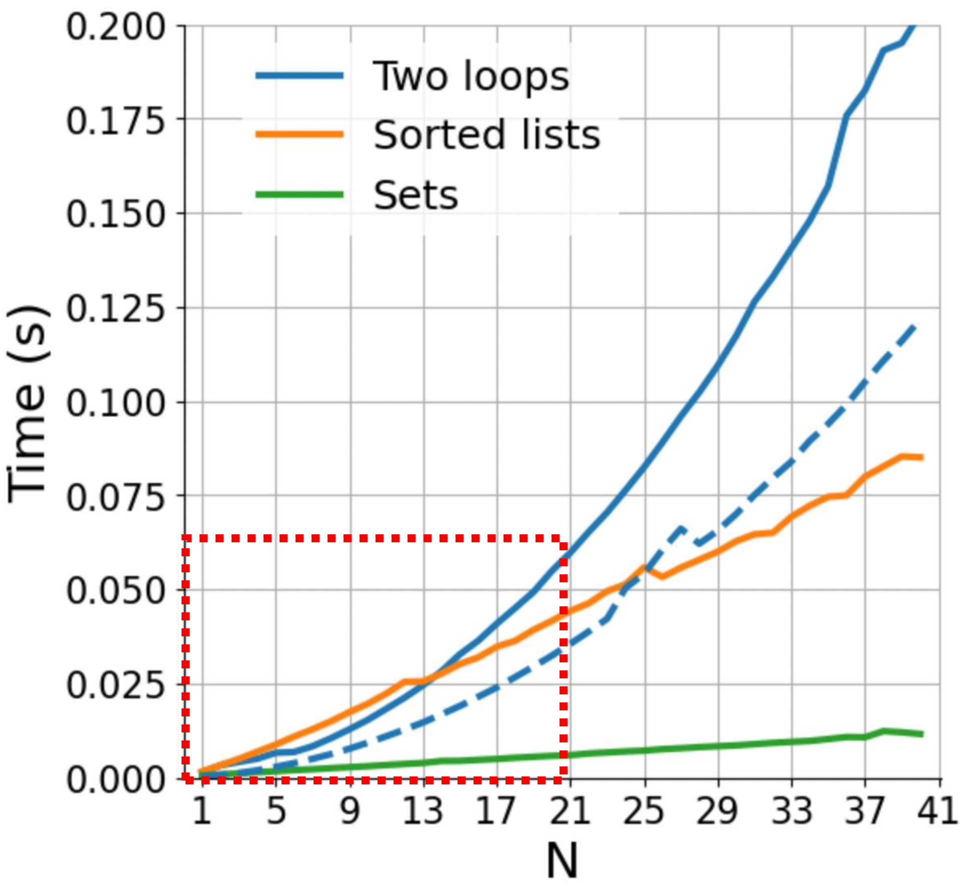
How does rewriting in C change the performance?

(rewriting in C, parallelization; same algorithm)

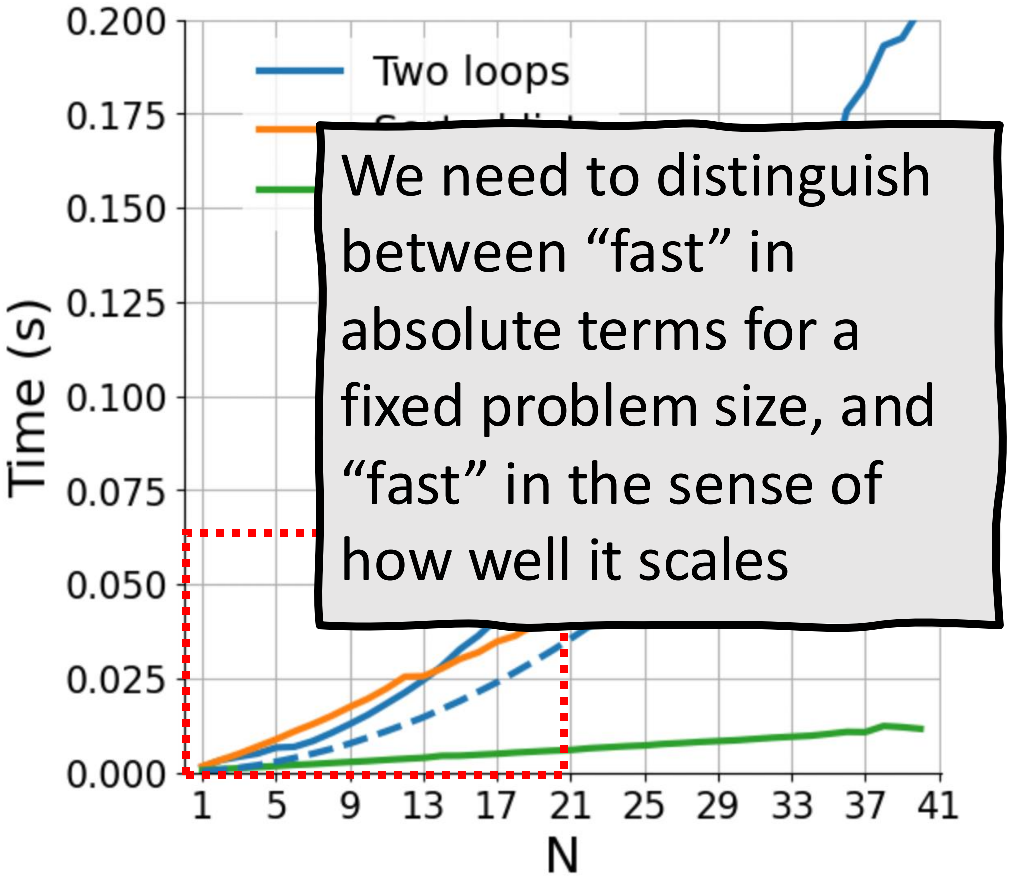


Faster language increases performance for fixed N

How does rewriting in C change the performance?
(rewriting in C, parallelization; same algorithm)



How does rewriting in C change the performance?
(rewriting in C, parallelization; same algorithm)



We need to distinguish between "fast" in absolute terms for a fixed problem size, and "fast" in the sense of how well it scales

COMING UP NEXT:
NumPy and the array data structure

NumPy

NumPy – huh, yeah – what's it good for?

- Introduces new data structure:
  the array

An array is a regular, N-dimensional grid of data of the same type, typically numerical data

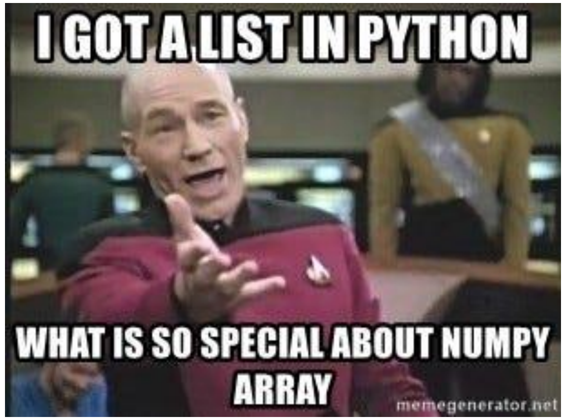
NumPy – huh, yeah – what’s it good for?

- Introduces new data structure:
  the array

An array is a regular, N-dimensional grid of data of the same type, typically numerical data

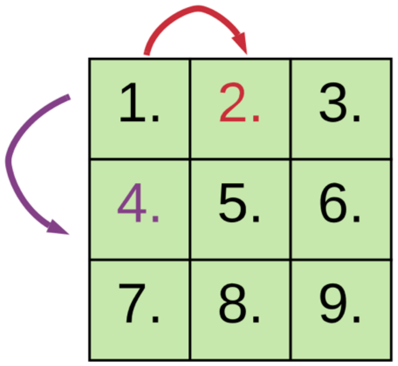
- An array could be represented as a list-of-lists
- Why are NumPy arrays better than a list-of-lists?

Computer architecture class

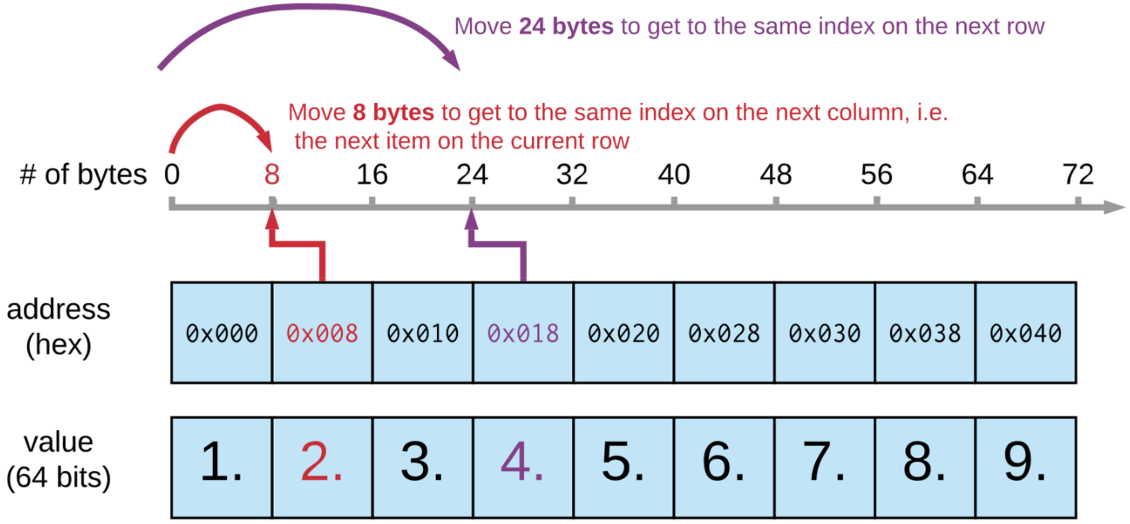


Efficiency of NumPy

1) **Memory**:
   - data occupies the minimum amount of memory required
   - some operations can be done without touching the memory at all!

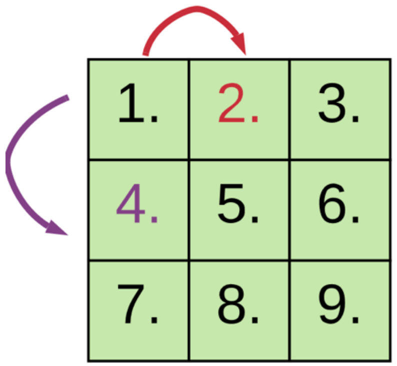



```
x = np.array([[1., 2., 3.],
              [4., 5., 6.],
              [7., 8., 9.]])
x.dtype ➔ dtype('float64') = 64 bits = 8 bytes
x.shape ➔ (3, 3)           x.itemsize ➔ 8
x.strides ➔ (24, 8)        x.nbytes ➔ 72
```



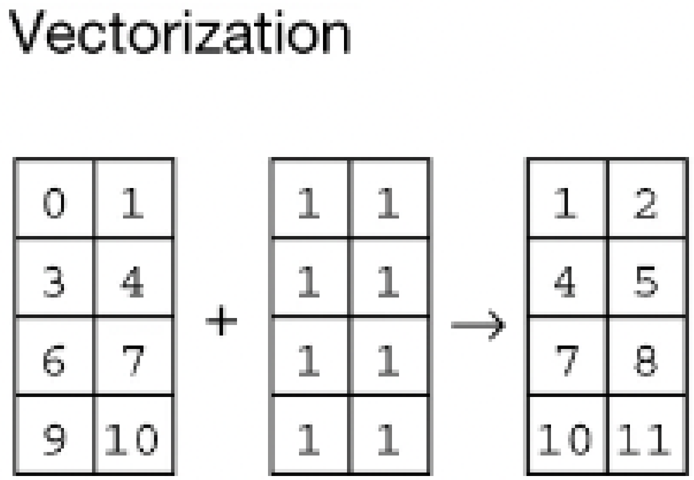
Efficiency of NumPy

1) **Memory**:
    - data occupies the minimum amount of memory required
    - some operations can be done without touching the memory at all!



2) **Speed**:
    - Many operations can be done very efficiently in C. For this to be useful, we need to avoid Python for-loops at all costs!
    - operating on entire arrays rather than their individual elements

→ "vectorize" the code



NumPy’s memory efficiency



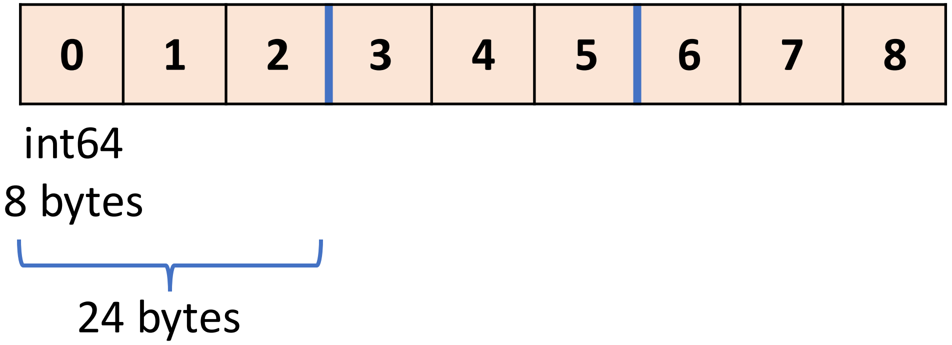

Memory block

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

int64

The array data is stored in a contiguous memory block, using native data types

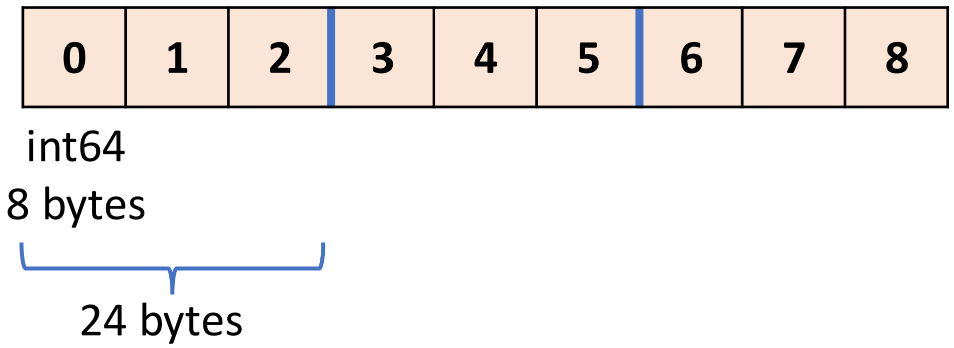
Memory block



NumPy array metadata

`dtype`	`int64`
`ndim`	`2`
`shape`	`(3, 3)`
`strides`	`(24, 8)`

Memory block



NumPy array metadata

dtype	int64
ndim	2
shape	(3, 3)
strides	(24, 8)

Metadata tells NumPy how to interpret the memory block

Memory block

0	1	2	3	4	5	6	7	8

int64
8 bytes

24 bytes

Metadata tells NumPy how to interpret the memory block

NumPy array metadata

dtype	int64
ndim	2
shape	(3, 3)
strides	(24, 8)

Memory block

0	1	2	3	4	5	6	7	8

int64
8 bytes

24 bytes

Metadata tells NumPy how to interpret the memory block

NumPy array metadata

dtype	int64
ndim	2
shape	(3, 3)
strides	(24, 8)

NumPy view

0	1	2
3	4	5
6	7	8

Memory block

0	1	2	3	4	5	6	7	8

The same memory block can be interpreted in many ways

NumPy operation

NumPy view

```
x
```

0	1	2
3	4	5
6	7	8

```
x.ravel()
```

0	1	2	3	4	5	6	7	8

```
x.T
```

0	3	6
1	4	7
2	5	8

```
x[::2, ::2]
```

0	2
6	8

Memory block

0	1	2	3	4	5	6	7	8

The same memory block can be interpreted in many ways

NumPy operation — **NumPy view**

```
x
```

0	1	2
3	4	5
6	7	8

```
x.ravel()
```

0	1	2	3	4	5	6	7	8

```
x.T
```

0	3	6
1	4	7
2	5	8

```
x[::2, ::2]
```

0	2
6	8

Memory block

0	1	2	3	4	5	6	7	8

The same memory block can be interpreted in many ways

NumPy operation	NumPy array metadata	NumPy view
`x`	dtype: int64; ndim: 2; shape: (3, 3); strides: (24, 8)	[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
`x.ravel()`	dtype: int64; ndim: 1; shape: (9,); strides: (8,)	[0, 1, 2, 3, 4, 5, 6, 7, 8]
`x.T`	dtype: int64; ndim: 2; shape: (3, 3); strides: (8, 24)	[[0, 3, 6], [1, 4, 7], [2, 5, 8]]
`x[::2, ::2]`	dtype: int64; ndim: 2; shape: (2, 2); strides: (48, 16)	[[0, 2], [6, 8]]

Memory block

0	1	2	3	4	5	6	7	8

There are NumPy operations that can be performed just by changing the metadata

NumPy operation | **NumPy array metadata** | **NumPy view**

very efficient --> **O(1)**

```
x
```

dtype	int64
ndim	2
shape	(3, 3)
strides	(24, 8)

0	1	2
3	4	5
6	7	8

```
x.ravel()
```

dtype	int64
ndim	1
shape	(9,)
strides	(8,)

0	1	2	3	4	5	6	7	8

```
x.T
```

dtype	int64
ndim	2
shape	(3, 3)
strides	(8, 24)

0	3	6
1	4	7
2	5	8

```
x[::2, ::2]
```

dtype	int64
ndim	2
shape	(2, 2)
strides	(48, 16)

0	2
6	8

Memory block

0	1	2	3	4	5	6	7	8

The same memory block can be interpreted in many ways

NumPy operation | **NumPy array metadata** | **NumPy view**

```
x
```

`dtype`	`int64`
`ndim`	`2`
`shape`	`(3, 3)`
`strides`	`(24, 8)`

0	1	2
3	4	5
6	7	8

How does the metadata look in this case?

```
x[ [0, 1, 2], [1, 0, 1] ]
```

`dtype`	
`ndim`	
`shape`	
`strides`	

1	3	7

Memory block

0	1	2	3	4	5	6	7	8

The same memory block can be interpreted in many ways

NumPy operation

```
x
```

NumPy array metadata

dtype	int64
ndim	2
shape	(3, 3)
strides	(24, 8)

NumPy view

0	1	2
3	4	5
6	7	8

In this case new memory needs to be allocated

Another memory block

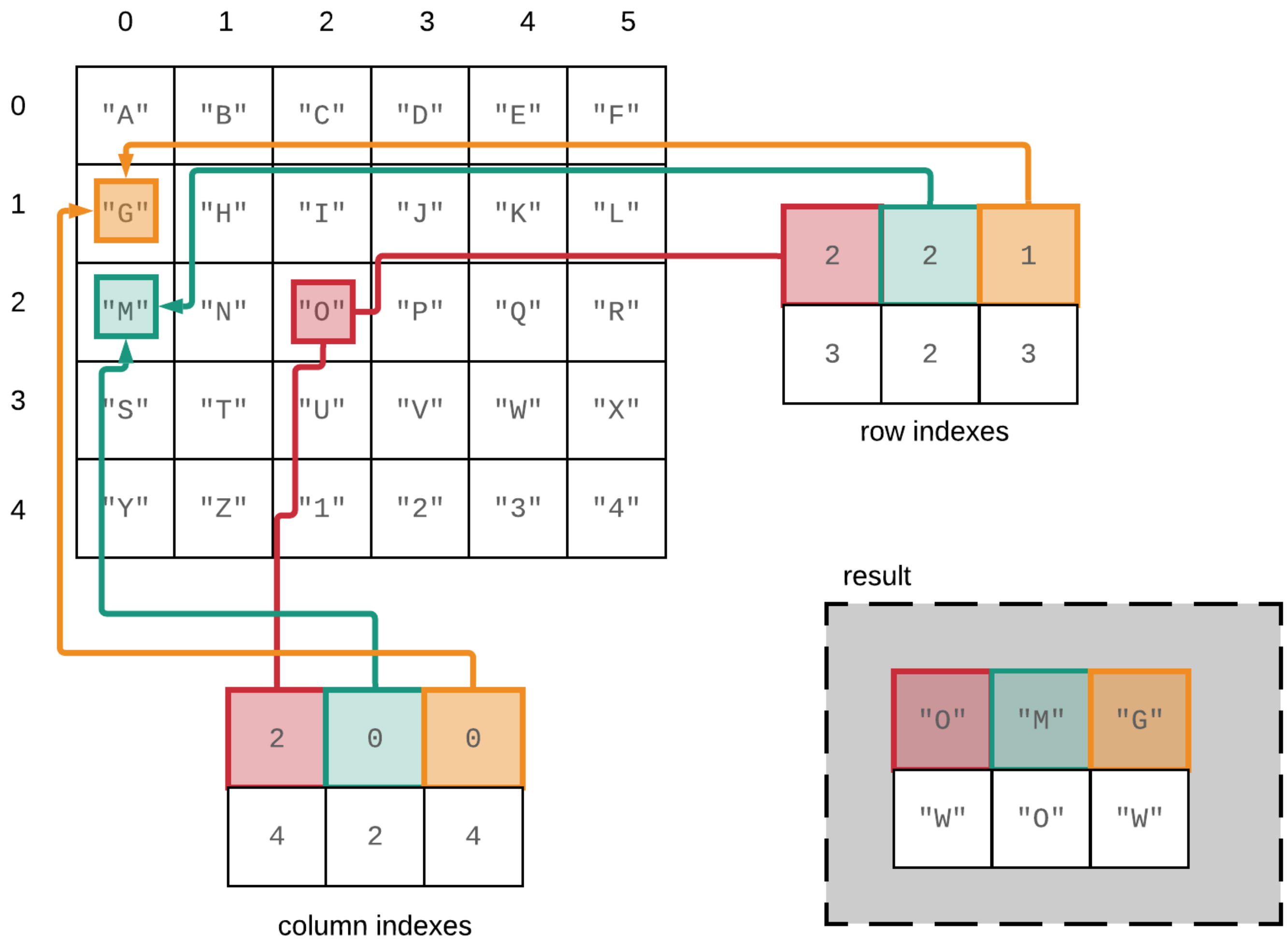
1	3	7

```
x[ [0, 1, 2], [1, 0, 1] ]
```

dtype	
ndim	
shape	
strides	

1	3	7

Fancy indexing in NumPy – reference slide



Operations that only change the metadata return a "**view " of the original memory block**, otherwise a new memory block needs to be allocated, returning a **"copy"**

Live Coding

notebooks/NumPy/**NumPy_views_and_copies.ipynb**

NumPy views and copies

Live Coding

notebooks/NumPy/**NumPy_views_and_copies.ipynb**

View

- accessing the array without changing the memory block
- slicing gives views
- in-place operations modify the memory block and all of its views

Copy

- when a copy of an array needs to be created, it allocates a separate memory block and associates it with a new metadata
- fancy indexing always gives copies
- a copy can be forced by method .copy()

NumPy views and copies

View

- accessing the array without changing the memory block
- slicing gives views
- in-place operations modify the memory block and all of its views

Copy

- when a copy of an array needs to be created, it allocates a separate memory block and associates it with a new metadata
- fancy indexing always gives copies
- a copy can be forced by method .copy()

Exercise

`exercises/view_or_copy/view_or_copy.ipynb`

A special kind of view: broadcasting operations

Memory block

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

NumPy array metadata

dtype	int64
ndim	2
shape	(4, 9)
strides	(0, 8)

The shape says we have 4 rows and 9 columns

A stride of 0 means that for each new row, we don't move in memory

A special kind of view: broadcasting operations

Memory block

0	1	2	3	4	5	6	7	8

NumPy array metadata

`dtype`	`int64`
`ndim`	`2`
`shape`	`(4, 9)`
`strides`	`(0, 8)`

The shape says we have 4 rows and 9 columns

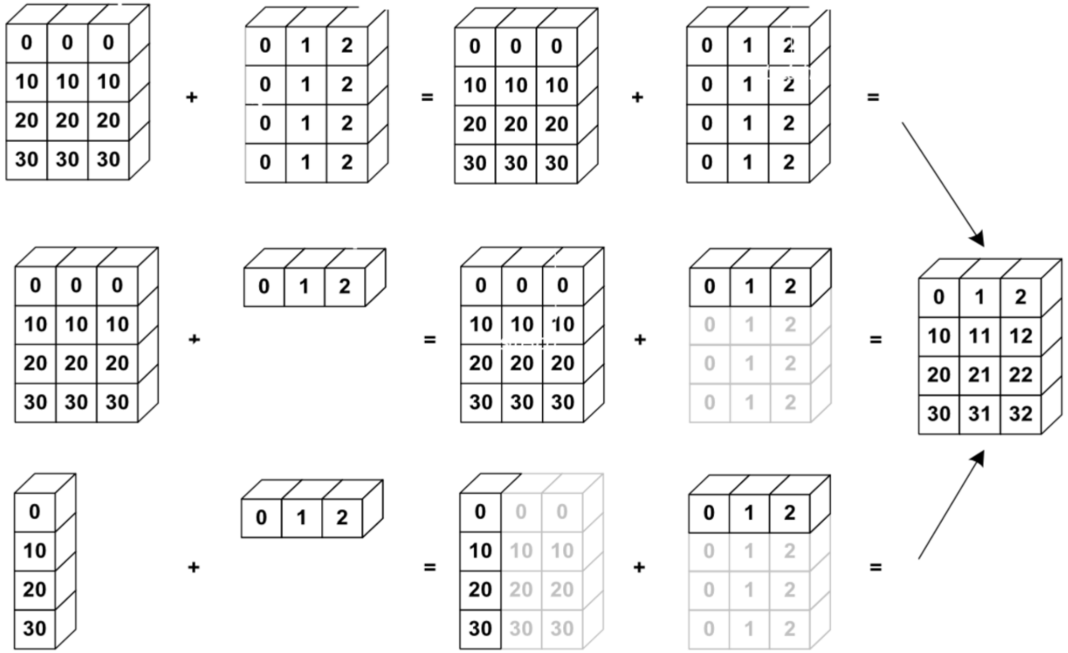
A stride of 0 means that for each new row, we don't move in memory

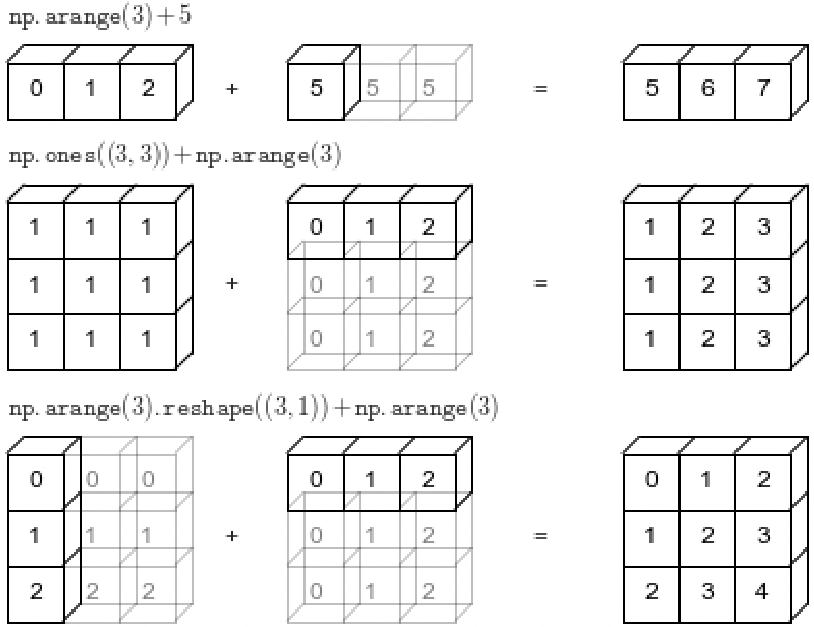
As a result, we obtain a view with duplicated rows, without using extra memory!

NumPy view

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

NumPy uses broadcasting to perform operation on arrays of different shape without having to allocate extra memory





Broadcasting notebook summary

Dive in Broadcastig

`notebooks/NumPy/broadcasting.ipynb`

- how NumPy treats arrays with different shapes during arithmetic operations

- Rules of broadcasting
  - **1:** If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
  - **2:** If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
  - **3:** If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

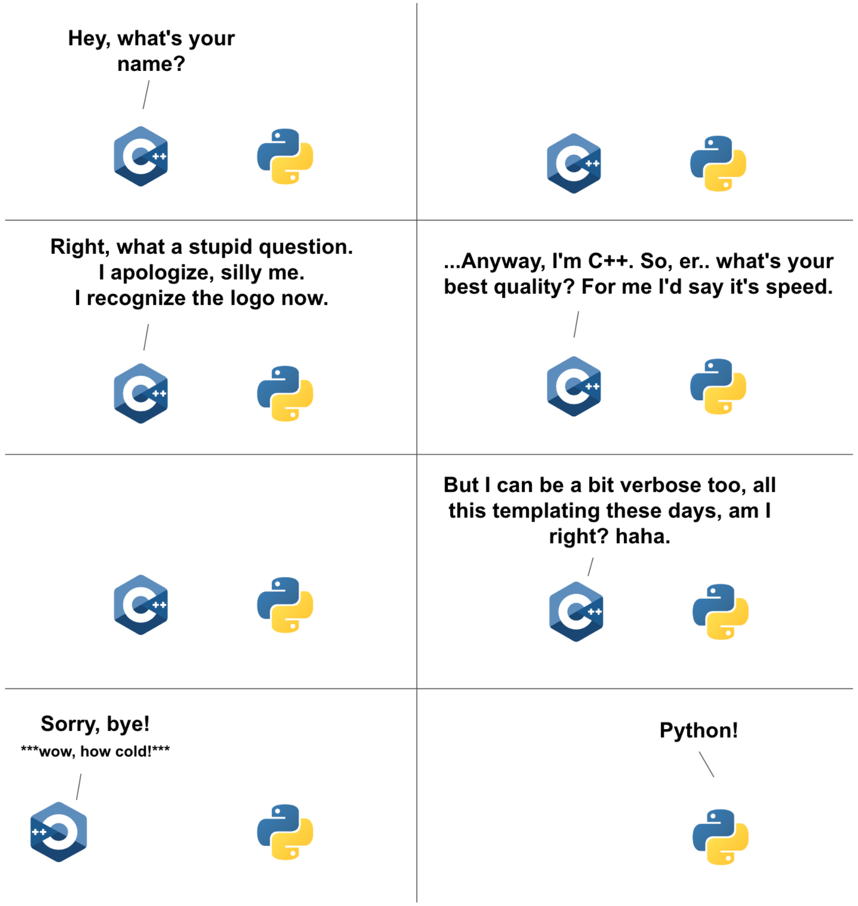
NumPy’s speed efficiency

For-loops in Python vs in C

- Data is of a C numerical type → regular layout in memory
  - A C loop can jump from one memory location to the next by moving by "strides" bytes and accumulating the result

- To get that performance, one needs to vectorize! it's important to <u>avoid for-loops at all costs</u>

(with NumPy in Python)



Vectorization

operations performed on entire arrays **at once**

→Faster computation

→no looping through each element individually

Vectorization

operations performed on entire arrays **at once**

→Faster computation

→no looping through each element individually

Aggregation functions

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Basic operators

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

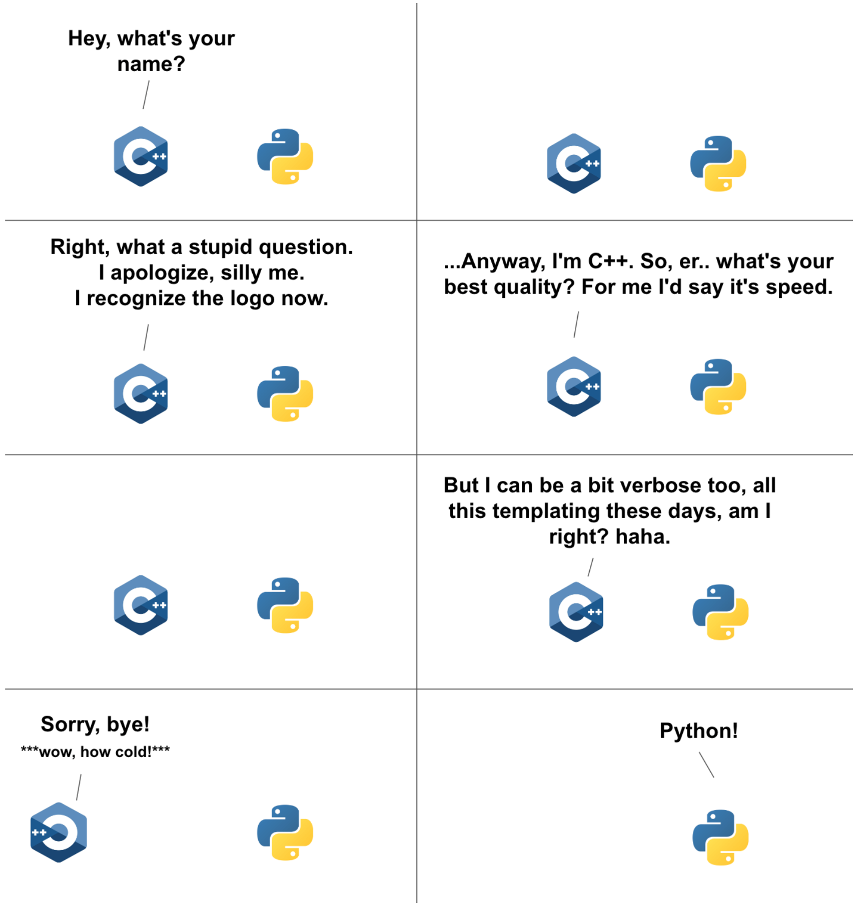
For-loops in Python vs in C

- Data is of a C numerical type → regular layout in memory
  - A C loop can jump from one memory location to the next by moving by "strides" bytes and accumulating the result

- To get that performance, one needs to vectorize! it's important to <u>avoid for-loops at all costs</u>

(with NumPy in Python)

How is efficiency of Python vs C in the Big-O sense?



Exercise: vectorize the code

Exercise

exercises/NumPy_vectorize / NumPy_vectorize.ipynb

Tabular data

Spreadsheets and databases rule the world!

Ariel Fischman holds the Guinness World Record for owning the most spreadsheet software (over 500!)

https://techcommunity.microsoft.com/t5/excel-blog/guinness-world-records-the-largest-collection-of-spreadsheet/ba-p/216592

What is tabular data?

Unlike arrays, each column can represent another type of value, with different data types

Date (index)	Wind speed	Wind direction	Rain fall (mm)	Hours of sun
7.3.2024	7.1	N	0.0	10
8.3.2024	0.3	NW	2.1	2
9.3.2024	1.1	SE	0.3	5

Subject ID (index)	Condition ID	Presentation nr	Response time (ms)	Response
VM	732	2	28	LEFT
VM	732	3	41	RIGHT
PB	665	1	73	LEFT

What is tabular data?

Column and rows have meaningful labels (indices) that are attached to the data for each operation

Date (index)	Wind speed	Wind direction	Rain fall (mm)	Hours of sun
7.3.2024	7.1	N	0.0	10
8.3.2024	0.3	NW	2.1	2
9.3.2024	1.1	SE	0.3	5

Subject ID (index)	Condition ID	Presentation nr	Response time (ms)	Response
VM	732	2	28	LEFT
VM	732	3	41	RIGHT
PB	665	1	73	LEFT

Many tools to handle tabular data

- Python tools
  - pandas: in-memory tabular data
  - dask: on-disk tabular data
- SQL databases
  - Optimized for retrieving rows (tree data structure for index)
  - Transactional: groups of operations are either all executed, or none
- Columnar DBs, Spark, Hadoop
  - Optimized for operations on columns
  - Ideal for data science tasks
  - Operations can be automatically distributed over multiple machines

Tabular data ideas and operations are universal for all tabular data tools

Pandas

```
df.groupby('condition_id')['response_time'].mean()
```

dask

```
df.groupby('condition_id')['response_time'].mean()
```

PySpark

```
df.groupby('condition_id').avg('response_time')
```

SQL

```
SELECT condition_id,
       AVG(response_time) AS avg_response_time
FROM df
GROUP BY condition_id;
```

Introduction to Pandas

Live Coding

notebooks/030_tabular_data/
010_pandas_introduction.ipynb

Introduction to Pandas

Live Coding
`notebooks/030_tabular_data/010_pandas_introduction.ipynb`

Main points:

- A DataFrame is a tabular data structure
- DataFrames have labeled columns and rows (“indices”)
- Columns can be of different C-native dtypes
- Operations are on columns by default
- NaNs are interpreted as missing data and ignored in most operations
- Strings (and dates) have a special accessor to perform vectorized string (or date) operations

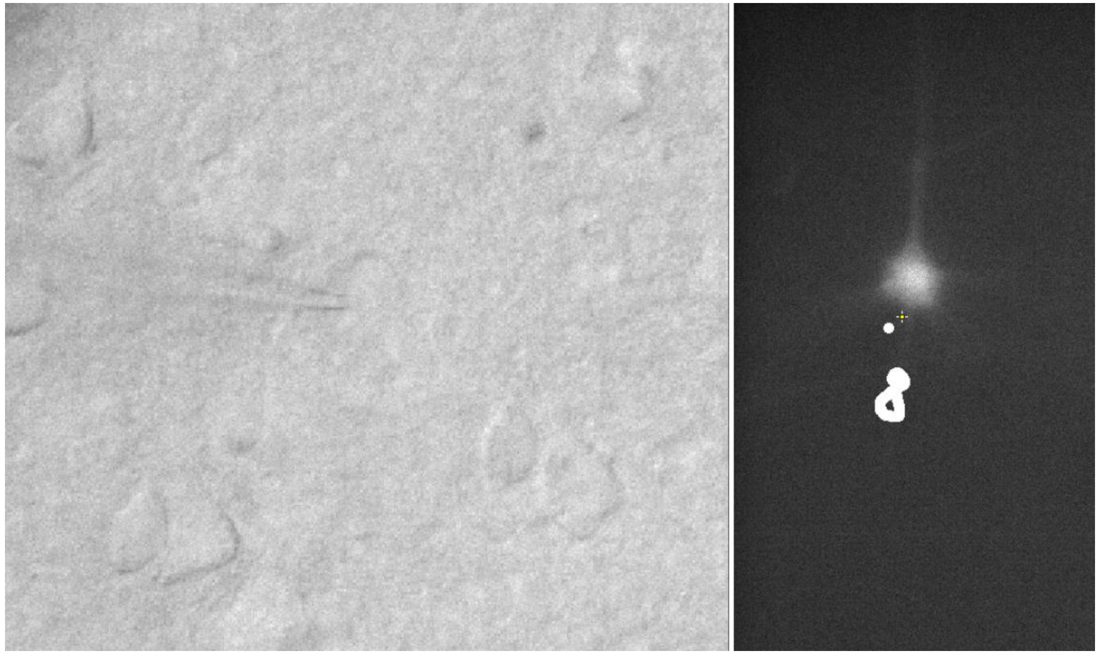
Basic Pandas reference slide

- Looking at data
  - `df.head()` : show the first 5 rows
  - `df.tail()` : show the last 5 rows
  - `df.sample(n)` : show n random rows
- Attributes
  - `df.shape` : size of the table
  - `df.dtypes` : print dtype of cols
  - `df.columns` : column index
  - `df.index` : rows index
- Indexing
  - `df[‘age’]` : get column ‘age’
  - `df[[‘age’, ‘name’]]` : multiple columns
  - `df.iloc[0, 2]` : one element, by position
- Exploration
  - `df[‘name’].unique()` : unique values
  - `df[‘age’].describe()` : summary stats
  - `df[‘age’].value_counts(dropna=False)` : number of rows per unique value in column
- Adding a column
  - `df[‘new’] = df[‘age’] * 3.1` : add new column
- Filtering
  - `df[df['age'] > 30]` : select rows where condition is True
- Operations
  - `df.min(), .max(), .mean(), .std(), etc.` : column-wise operations
  - `df.count()` : count of non-NaN elements in columns
  - `df.sort_values(‘name’)` : reorder rows by values of column ‘name’
  - `df.sort_index()` : reorder rows by the index values
- String operations
  - `df[‘name’].str` : accessor for operations on the strings in a col
  - `df['name'].str[2:4]` : slice the strings in a col
  - `df['name'].str.count(‘a’)` : count the letter ‘a’ in the string in a col

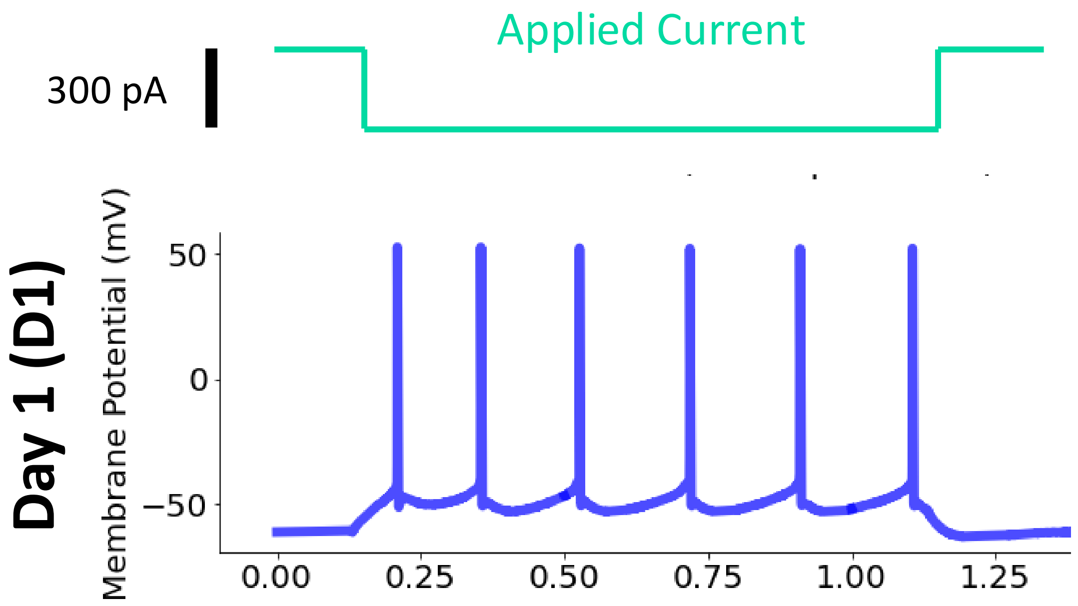
Tabular data example from the lab

- Research question: Does neuronal activity change over time? Does this depend on the overall activity level of the neuronal network?
  The mainstream theory suggests that neural activity is self-regulating to maintain a baseline level ("homeostatic plasticity")

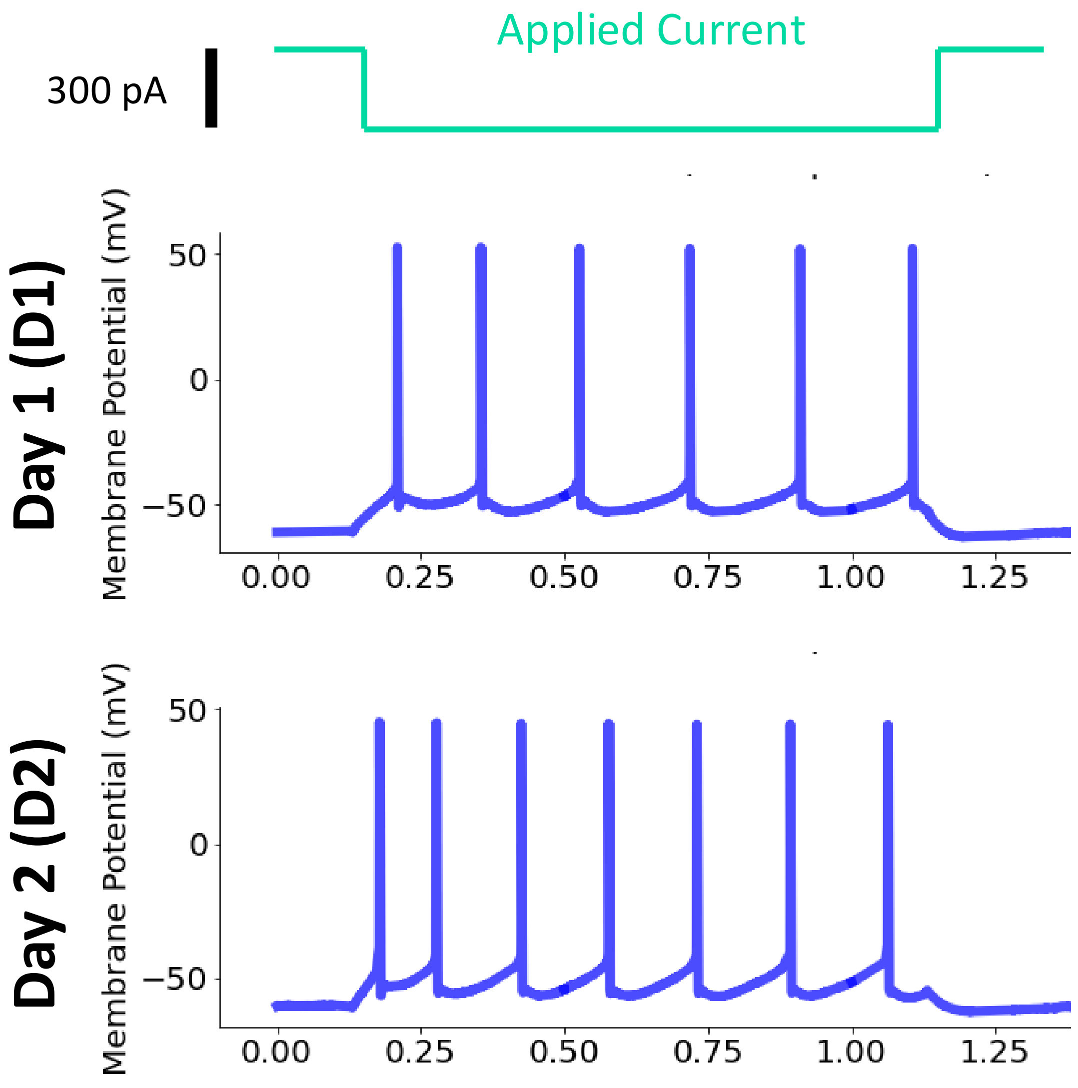
- Exp design: patch clamp recordings from the same cells (or different cells/ same slices) before and after prolonged incubation in high potassium (K)
- Potassium stimulates and TTX silences the entire network, allowing us to control the overall activity



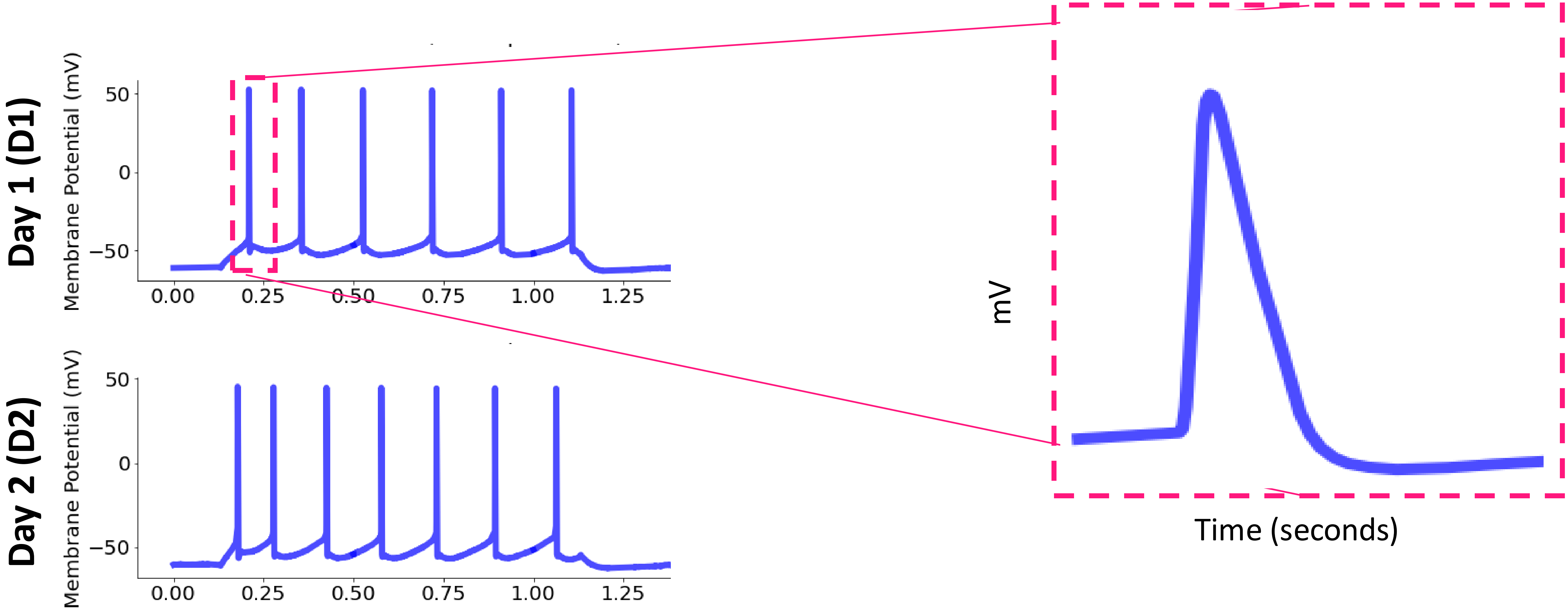
Variables


Applied Current
300 pA
Day 1 (D1)
Membrane Potential (mV)
50
0
−50
0.00
0.25
0.50
0.75
1.00
1.25

Variables

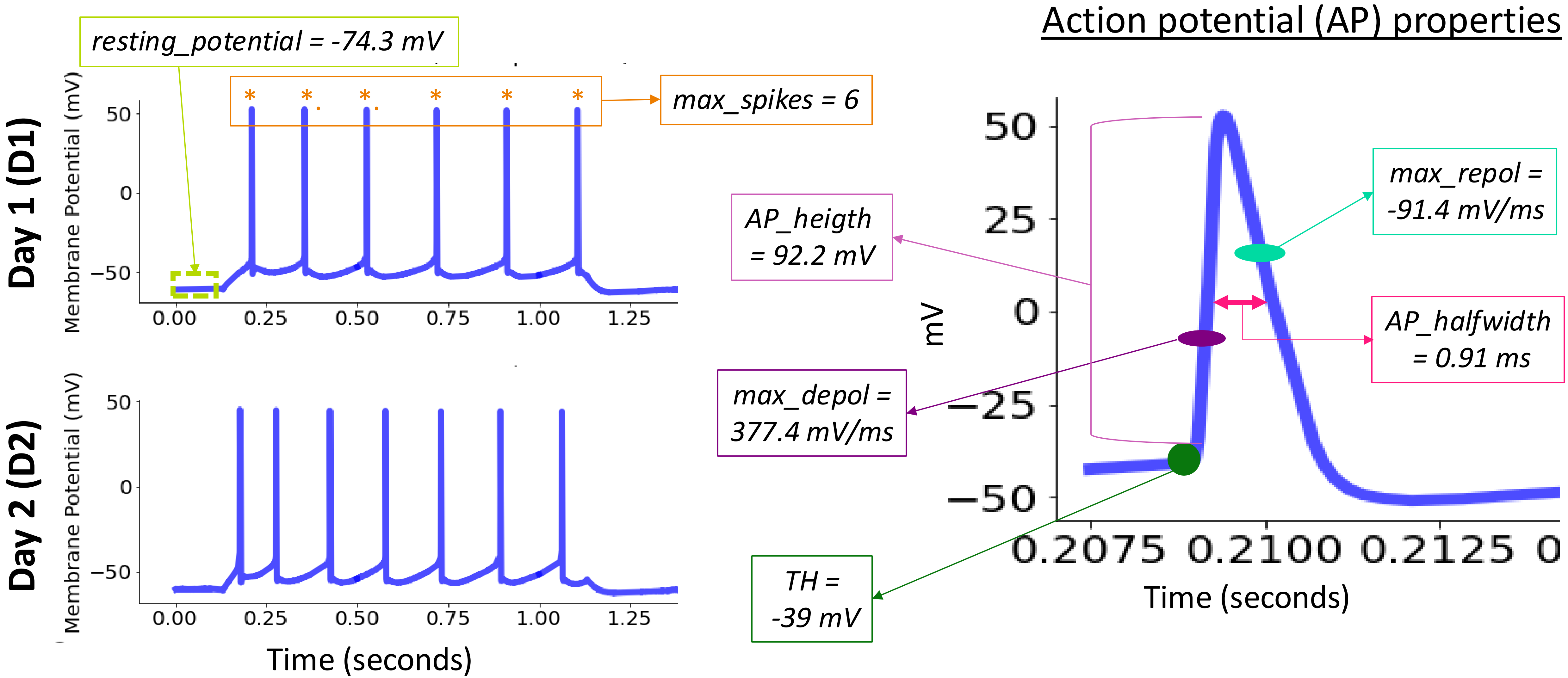

Applied Current
300 pA
Day 1 (D1)
Membrane Potential (mV)
50
0
−50
0.00
0.25
0.50
0.75
1.00
1.25
Day 2 (D2)
Membrane Potential (mV)
50
0
−50
0.00
0.25
0.50
0.75
1.00
1.25

Variables



Day 1 (D1)
Membrane Potential (mV)
50
0
−50
0.00
0.25
0.50
0.75
1.00
1.25
Day 2 (D2)
Membrane Potential (mV)
50
0
−50
0.00
0.25
0.50
0.75
1.00
1.25
mV
Time (seconds)

Variables


resting_potential = -74.3 mV
max_spikes = 6
Day 1 (D1)
Day 2 (D2)
Membrane Potential (mV)
50
0
−50
0.00
0.25
0.50
0.75
1.00
1.25
Time (seconds)
Action potential (AP) properties
AP_heigth = 92.2 mV
max_repol = -91.4 mV/ms
AP_halfwidth = 0.91 ms
max_depol = 377.4 mV/ms
TH = -39 mV
mV
25
−25
0.2075
0.2100
0.2125
Time (seconds)

Hands-on

Let's have a look at the neural data

Exercise

`exercises/pandas_intro`

- Use Pandas to explore the neural data
- Submit a PR for Issue #2 on GitHub

TABULAR DATA OPERATIONS

Common operations on tabular data

- Tabular data has additional needs compared to arrays. Understanding how to vectorize these operations is critical for handling them

- Combine information across tables (**join, anti-join**)
  - **Join**: e.g., combine table with experiments results with table with experiments metadata (date, location, experimenter, free-form notes, …)
  - **Anti-join**: e.g. student compiles list of outliers, exclude them from the table of experiments to analyze
- Summary tables (**split-apply-combine**)
  - E.g., compute average measurement and standard deviation by experimental condition and treatment dosage
- Window functions to vectorize complex computations over groups
  - E.g., compute the time distance between experiments by lab technician

Joins

Join operations: combining informations from multiple tables

subject_id	condition_id	response_time	response
312	A1	0.12	LEFT
312	A2	0.37	LEFT
312	C2	0.68	LEFT
313	A1	0.07	RIGHT
313	B1	0.08	RIGHT
314	A2	0.29	LEFT
314	B1	0.14	RIGHT
314	C2	0.73	RIGHT

+

	orientation	duration	surround	stimulus_type
A1	0	0.1	FULL	LINES
A2	0	0.01	NONE	DOTS
B1	45	0.1	NONE	PLAID
B2	45	0.01	FULL	PLAID
C1	90	0.2	FULL	WIGGLES

=

subject_id	condition_id	response_time	response	orientation	duration	surround	stimulus_type
312	A1	0.12	LEFT	0.0	0.10	FULL	LINES
312	A2	0.37	LEFT	0.0	0.01	NONE	DOTS
312	C2	0.68	LEFT	NaN	NaN	NaN	NaN
313	A1	0.07	RIGHT	0.0	0.10	FULL	LINES
313	B1	0.08	RIGHT	45.0	0.10	NONE	PLAID
314	A2	0.29	LEFT	0.0	0.01	NONE	DOTS
314	B1	0.14	RIGHT	45.0	0.10	NONE	PLAID
314	C2	0.73	RIGHT	NaN	NaN	NaN	NaN

Join operations

Live Coding

`notebooks/tabular_data/020_join_operations.ipynb`

Join operations

Live Coding

`notebooks/tabular_data/020_join_operations.ipynb`

Main points:

- Join operations can be used to combine two tables using the values of one or more columns
- Different types of join:
  - left/right: keep all the column values that are present in the first/second table
  - inner: keep all the column values that are present in both tables
  - outer: keep all the column values that are present in one or the other tables
- Anti-joins can be used to exclude the values that are present in one, but not the other table (filtering based on arbitrary criteria)

Hands-on

Exercise

exercises/tabular_join

- Use joins to add experiment information to the neural data
- Use anti-joins to remove outliers
- Submit a PR for Issue #3 on GitHub

Split-apply-combine

The basic structure of most numerical analyses

split | **apply** | **combine**

	subject_id	condition_id	response_time	response
0	312	A1	0.12	LEFT
1	312	A2	0.37	LEFT
2	312	C2	0.68	LEFT
3	313	A1	0.07	RIGHT
4	313	B1	0.08	RIGHT
5	314	A2	0.29	LEFT
6	314	B1	0.14	RIGHT
7	314	C2	0.73	RIGHT
8	711	A1	4.01	RIGHT
9	712	A2	3.29	LEFT
10	713	B1	5.74	LEFT
11	714	B2	3.32	RIGHT

A1

	subject_id	condition_id	response_time	response
0	312	A1	0.12	LEFT
3	313	A1	0.07	RIGHT
8	711	A1	4.01	RIGHT

A2

	subject_id	condition_id	response_time	response
1	312	A2	0.37	LEFT
5	314	A2	0.29	LEFT
9	712	A2	3.29	LEFT

B1

	subject_id	condition_id	response_time	response
4	313	B1	0.08	RIGHT
6	314	B1	0.14	RIGHT
10	713	B1	5.74	LEFT

B2

	subject_id	condition_id	response_time	response
11	714	B2	3.32	RIGHT

C2

	subject_id	condition_id	response_time	response
2	312	C2	0.68	LEFT
7	314	C2	0.73	RIGHT

mean

	response_time
A1	1.4

	response_time
A2	1.316667

	response_time
B1	1.986667

	response_time
B2	3.32

	response_time
C2	0.705

	response_time
condition_id	
A1	1.400000
A2	1.316667
B1	1.986667
B2	3.320000
C2	0.705000

Split-apply-combine operations

Live Coding

`notebooks/030_tabular_data/030_split-apply-combine.ipynb`

Split-apply-combine operations

Live Coding
notebooks/030_tabular_data/
030_split-apply-combine.ipynb

Main points:

- Tabular data tools have a way to vectorize the standard split-apply-combine operations, using a "group-by" command
- In addition, Pandas has got a "pivot-table" command that can be used to simplify the creation of more complex summary tables

```
df.groupby('condition_id')['response_time'].mean()
```

split

	subject_id	condition_id	response_time	response
0	312	A1	0.12	LEFT
1	312	A2	0.37	LEFT
2	312	C2	0.68	LEFT
3	313	A1	0.07	RIGHT
4	313	B1	0.08	RIGHT
5	314	A2	0.29	LEFT
6	314	B1	0.14	RIGHT
7	314	C2	0.73	RIGHT
8	711	A1	4.01	RIGHT
9	712	A2	3.29	LEFT
10	713	B1	5.74	LEFT
11	714	B2	3.32	RIGHT

A1

	subject_id	condition_id	response_time	response
0	312	A1	0.12	LEFT
3	313	A1	0.07	RIGHT
8	711	A1	4.01	RIGHT

A2

	subject_id	condition_id	response_time	response
1	312	A2	0.37	LEFT
5	314	A2	0.29	LEFT
9	712	A2	3.29	LEFT

B1

	subject_id	condition_id	response_time	response
4	313	B1	0.08	RIGHT
6	314	B1	0.14	RIGHT
10	713	B1	5.74	LEFT

B2

	subject_id	condition_id	response_time	response
11	714	B2	3.32	RIGHT

C2

	subject_id	condition_id	response_time	response
2	312	C2	0.68	LEFT
7	314	C2	0.73	RIGHT

apply

mean

	response_time
A1	1.4

	response_time
A2	1.316667

	response_time
B1	1.986667

	response_time
B2	3.32

	response_time
C2	0.705

combine

	response_time
condition_id	
A1	1.400000
A2	1.316667
B1	1.986667
B2	3.320000
C2	0.705000

```
data.pivot_table(
      index='condition_id', columns='response',
      values='response_time', aggfunc='mean',
)
```

split

apply

combine

	subject_id	condition_id	response_time	response
0	312	A1	0.12	LEFT
1	312	A2	0.37	LEFT
2	312	C2	0.68	LEFT
3	313	A1	0.07	RIGHT
4	313	B1	0.08	RIGHT
5	314	A2	0.29	LEFT
6	314	B1	0.14	RIGHT
7	314	C2	0.73	RIGHT
8	711	A1	4.01	RIGHT
9	712	A2	3.29	LEFT
10	713	B1	5.74	LEFT
11	714	B2	3.32	RIGHT

response	LEFT	RIGHT
condition_id		
A1	0.12	2.04
A2	1.32	NaN
B1	5.74	0.11
B2	NaN	3.32
C2	0.68	0.73

Hands-on

Exercise

exercises/
tabular_split_apply_combine

- Compute summary statistics for the neural data
- Submit a PR for Issue #4 on GitHub

Hands-on

Exercise

`exercises/tuberculosis`

- Compute some summary tables for the WHO tuberculosis data

Males → sp_m ; 15-24 years → 1524

rownames	country	year	sp_m_014	sp_m_1524	sp_m_2534	...	sp_f_2534	sp_f_3544	sp_f_4554	sp_f_5564	sp_f_65
5551	San Marino	2009	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
642	Belarus	2009	0.0	66.0	173.0	...	52.0	52.0	41.0	25.0	68.0
7234	Zimbabwe	2007	138.0	500.0	3693.0	...	3311.0	0.0	553.0	213.0	90.0
3471	Kuwait	2008	0.0	18.0	90.0	...	47.0	27.0	7.0	5.0	6.0
3336	Jordan	2009	1.0	5.0	15.0	...	14.0	8.0	3.0	7.0	12.0
2689	Grenada	2008	NaN	1.0	NaN	...	NaN	NaN	NaN	NaN	NaN
634	Belarus	2001	2.0	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN

Tidy Data

Same data, different organization
Which one is best for data analysis?

	John Smith	Jane Doe	Mary Johnson
treatmenta	—	16	3
treatmentb	2	11	1

	treatmenta	treatmentb
John Smith	—	2
Jane Doe	16	11
Mary Johnson	3	1

name	trt	result
John Smith	a	—
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

Same data, different organization
Which one is best for data analysis?

	John Smit
treatmenta	—
treatmentb	2

What do we want?

We want data to be in a natural format, such that data analysis is easy

name	trt	result
John Smith	a	—
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

	tmenta	treatmentb
	—	2
	16	11
Mary Johnson	3	1

Tidy data

In tidy data:
1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

Variables (or features, attributes)

Observations (or samples)

Subject ID	Condition ID	Trial nr	Response time (ms)	Response
VM	732	2	28	LEFT
VM	732	3	41	RIGHT
PB	665	1	73	LEFT

Variables increase when new types of measurements are introduced

Observations increase when new units (dates, subjects, …) are measured

Hands-on

Identify variables, observations, and values.

What would a tidy version look like?

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

Table 11: Original weather dataset. There is a column for each possible day in the month. Columns `d9` to `d31` have been omitted to conserve space.

Hands-on

Identify variables, observations, and values.

What would a tidy version look like?

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

Table 11: Original weather dataset. There is a column for each possible day in the month. Columns `d9` to `d31` have been omitted to conserve space.

id	date	tmax	tmin
MX17004	2010-01-30	27.8	14.5
MX17004	2010-02-02	27.3	14.4
MX17004	2010-02-03	24.1	14.4
MX17004	2010-02-11	29.7	13.4
MX17004	2010-02-23	29.9	10.7
MX17004	2010-03-05	32.1	14.2
MX17004	2010-03-10	34.5	16.8
MX17004	2010-03-16	31.1	17.6
MX17004	2010-04-27	36.3	16.7
MX17004	2010-05-27	33.2	18.2

(b) Tidy data

Messy data

"Tidy datasets are all alike but every messy dataset is messy in its own way"
- Hadley Wickham

Variables are stored in both rows and columns 😱

city	type	date	temperature
Bilbao	tmax	2024-07-03	34
Bilbao	tmin	2024-07-03	25
Bordeaux	tmax	2024-03-21	29
Bordeaux	tmin	2024-03-21	23
Berlin	tmax	2021-08-16	21
Berlin	tmin	2021-08-16	14
Heraklion	tmax	2021-09-01	30
Heraklion	tmin	2021-09-01	23

Column headers are values, not variable names 😱

subject	date	A	B
PB	2024-07-03	0.12	0.19
VM	2024-03-21	0.37	0.41
TZ	2021-08-16	0.68	0.73
LS	2021-09-01	0.07	0.08
ZS	2023-11-11	0.08	0.16

Some variables are stored in the file names 😱

```
2024-01_prices_DE.csv
2024-01_prices_FR.csv
2024-02_prices_DE.csv
2024-02_prices_FR.csv
```

Multiple variables are stored in one column 😱

country	year	variable	cases
Angola	2000	sp_m_014	186.0
Angola	2001	sp_m_014	230.0
Angola	2002	sp_m_014	435.0
Angola	2003	sp_m_014	409.0
Angola	2004	sp_m_014	554.0
Angola	2005	sp_m_014	520.0
Angola	2006	sp_m_014	540.0

The life-changing magic of tidying up data

Pivoting – we know this one

city	type	date	temperature
Bilbao	tmax	2024-07-03	34
Bilbao	tmin	2024-07-03	25
Bordeaux	tmax	2024-03-21	29
Bordeaux	tmin	2024-03-21	23
Berlin	tmax	2021-08-16	21
Berlin	tmin	2021-08-16	14
Heraklion	tmax	2021-09-01	30
Heraklion	tmin	2021-09-01	23

The “type” column is storing variable names

pivot

	type	tmax	tmin
city	date		
Berlin	**2021-08-16**	21	14
Bilbao	**2024-07-03**	34	25
Bordeaux	**2024-03-21**	29	23
Heraklion	**2021-09-01**	30	23

```
df.pivot_table(
        index=['city', 'date'], columns='type’,
        values='temperature', aggfunc='max’,
)
```

The life-changing magic of tidying up data

Melting – it's kind of the opposite of pivoting

The treatment values are stored as a column name 😱

subject	date	A	B
PB	2024-07-03	0.12	0.19
VM	2024-03-21	0.37	0.41
TZ	2021-08-16	0.68	0.73
LS	2021-09-01	0.07	0.08
ZS	2023-11-11	0.08	0.16

melt →

subject	date	variable	response_time
PB	2024-07-03	A	0.12
VM	2024-03-21	A	0.37
TZ	2021-08-16	A	0.68
LS	2021-09-01	A	0.07
ZS	2023-11-11	A	0.08
PB	2024-07-03	B	0.19
VM	2024-03-21	B	0.41
TZ	2021-08-16	B	0.73
LS	2021-09-01	B	0.08
ZS	2023-11-11	B	0.16

```
pd.melt(data, id_vars=['subject', 'date'], value_name='response_time')
```

Split the columns in (A) "id_vars" and (B) non-"id_vars". The column names in (B) are used as new values in a new column "variable". The values in columns (B) go into a new column, "response_time".

The life-changing magic of tidying up data

`pd.concat` – add together tables with the same variables (columns)

Some variables are stored in the file names

```
2024-01_prices_DE.csv
2024-01_prices_FR.csv
2024-02_prices_DE.csv
2024-02_prices_FR.csv
```

```
tables = []
for filename in filenames:
    # Parse filename
    year_month, _, country = filename[:-4].split('_')
    # Read table and add columns for the variables
    df = pd.read_csv(filename)
    # Add the variables that were in the filename
    df['year_month'] = year_month
    df['country’] = country
    # Store table
    tables.append(df)

# Create complete table
tidy_df = pd.concat(tables)
```

Hands-on

Exercise

`exercises/tabular_tidy_data`

- Tidy up the data set in the tuberculosis exercise and compute the summary stats
- Submit a PR for Issue #5 on GitHub

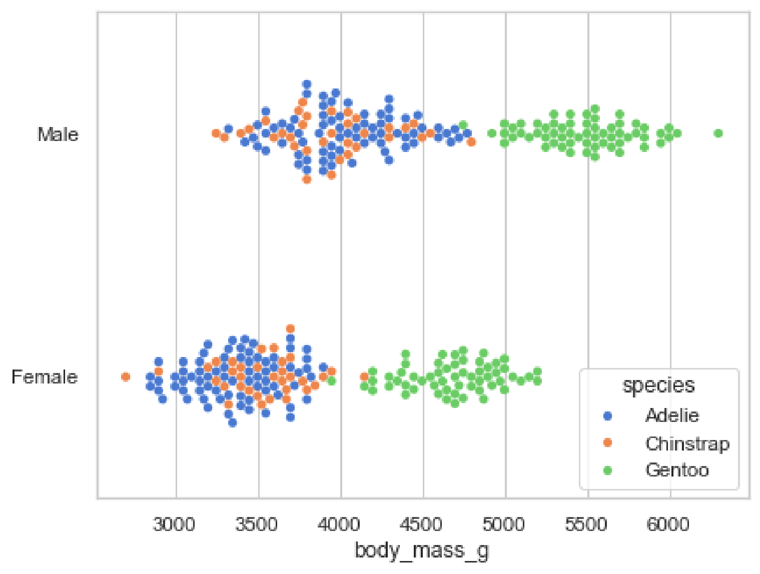
Multiple variables are stored in one column

country	year	variable	cases
Angola	2000	sp_m_014	186.0
Angola	2001	sp_m_014	230.0
Angola	2002	sp_m_014	435.0
Angola	2003	sp_m_014	409.0
Angola	2004	sp_m_014	554.0
Angola	2005	sp_m_014	520.0
Angola	2006	sp_m_014	540.0

Why is tidy data good?

- Many analyses require a simple sequence of steps:
  - Filter by individual variables to discard data that is not needed
  - Group and summarize
  - Re-arrange (e.g. sort)
  - Visualize
- Joining tidy tables is easy!
- One can write generic code that takes tidy data as input.
  For example, **seaborn** relies on tidy data to make complex plots

```
sns.swarmplot(
    data=df,
    x="body_mass_g",
    y="sex",
    hue="species”,
)
```



Window functions

Window functions: grouped row-by-row operations

- “Window functions” are a kind of split-apply-combine operation, but instead of aggregating the data in a group and returning one value per group, they return one value per row
- Examples: ranking all entries in a group; computing the distance between timestamps per group; number the rows by group in chronological order

- In Pandas, most of these operations can be performed with a combination of sorting and grouping-by

Window functions

Live Coding

`notebooks/tabular_data/040_window_functions.ipynb`

Window functions

Live Coding

notebooks/tabular_data/
040_window_functions.ipynb

- Main points:
  - Window functions perform row-by-row operations on grouped data
  - They are an advanced way of avoiding for loops with tabular data
  - In Pandas, they can be achieved with a combo of sorting and grouping-by

Window functions operations

```
df['nr_lefts'] = df.sort_values('time (ms)').groupby('subject_id')['is_left'].cumsum()
```

	subject_id	time (ms)	response	is_left
574	3	540	RIGHT	False
1190	2	552	LEFT	True
1895	2	1036	LEFT	True
53	3	257	RIGHT	False
158	2	743	RIGHT	False
551	3	619	LEFT	True
1602	1	43	RIGHT	False
413	1	471	LEFT	True
785	1	121	LEFT	True
1393	2	903	RIGHT	False
629	2	353	LEFT	True
1829	3	768	RIGHT	False
902	1	1093	LEFT	True
1486	2	3	RIGHT	False

	subject_id	time (ms)	response	is_left
1602	1	43	RIGHT	False
785	1	121	LEFT	True
413	1	471	LEFT	True
902	1	1093	LEFT	True

	subject_id	time (ms)	response	is_left
1486	2	3	RIGHT	False
629	2	353	LEFT	True
1190	2	552	LEFT	True
158	2	743	RIGHT	False
1393	2	903	RIGHT	False
1895	2	1036	LEFT	True

	subject_id	time (ms)	response	is_left
53	3	257	RIGHT	False
574	3	540	RIGHT	False
551	3	619	LEFT	True
1829	3	768	RIGHT	False

	nr_lefts
1602	0
785	1
413	2
902	3

	nr_lefts
1486	0
629	1
1190	2
158	2
1393	2
1895	3

	nr_lefts
53	0
574	0
551	1
1829	1

	subject_id	time (ms)	response	is_left	nr_lefts
1602	1	43	RIGHT	False	0
785	1	121	LEFT	True	1
413	1	471	LEFT	True	2
902	1	1093	LEFT	True	3
1486	2	3	RIGHT	False	0
629	2	353	LEFT	True	1
1190	2	552	LEFT	True	2
158	2	743	RIGHT	False	2
1393	2	903	RIGHT	False	2
1895	2	1036	LEFT	True	3
53	3	257	RIGHT	False	0
574	3	540	RIGHT	False	0
551	3	619	LEFT	True	1
1829	3	768	RIGHT	False	1

Hands-on

Exercise

exercises/
tabular_window_functions

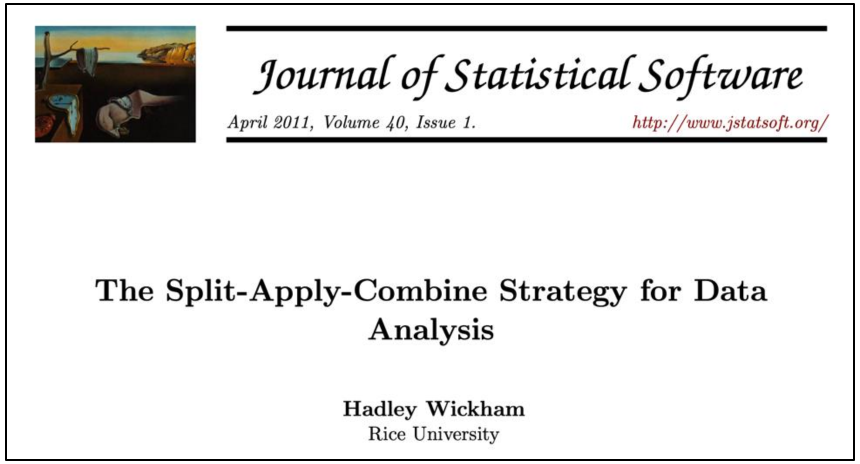
- Compute the average number of days each patcher waited between experiments
- Submit a PR for Issue #6 on GitHub

Global summary

- There are many different data structures, each specialized in efficiently processing one type of data
- Code performance grows differently with data size: Big-O
- NumPy array efficiently store data in a C-native memory block, interpreted as an array using some metadata
- NumPy operations that only need to change the metadata do so, creating a view of the same memory block. These operations are O(1)!
- Tabular data can also be vectorized using joins, anti-joins, split-apply-combine operations, and window functions
- For these operations to be efficient and painless, data should be stored in a tidy data format

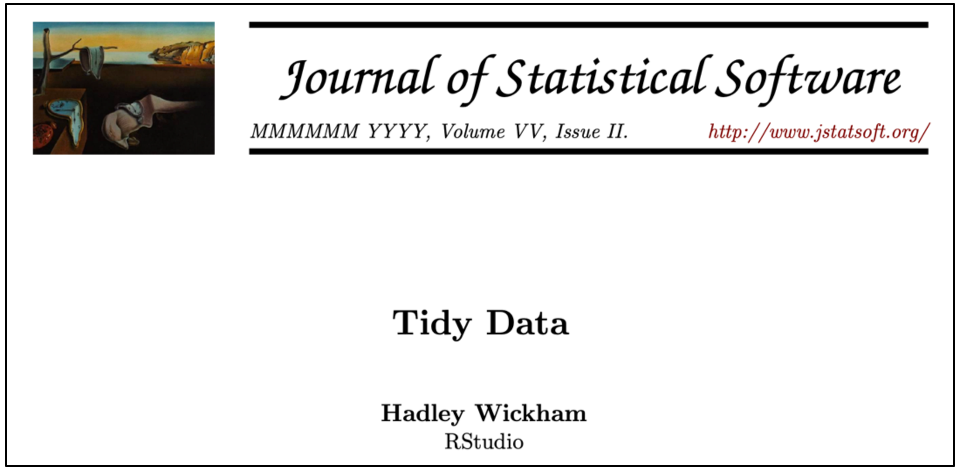
What we didn’t talk about

- Other data structures: graphs, trees, priority queues, …
- Options for working with large data on disk / remotely (instead of in-memory)
- Best practices in data handling: versioning, lineage, sharing
- Organizing a complex data set in multiple tables
- … and a lot more!


Journal of Statistical Software
April 2011, Volume 40, Issue 1. http://www.jstatsoft.org/

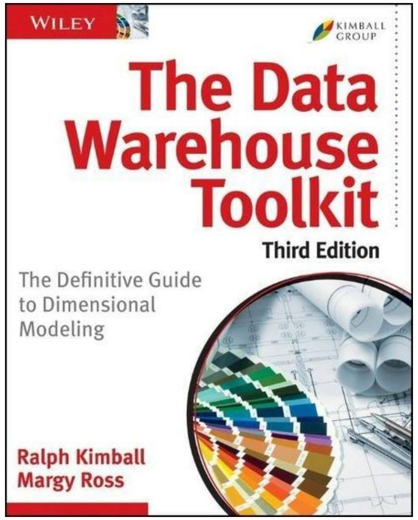
The Split-Apply-Combine Strategy for Data Analysis

Hadley Wickham
Rice University


Journal of Statistical Software
MMMMMM YYYY, Volume VV, Issue II. http://www.jstatsoft.org/

Tidy Data

Hadley Wickham
RStudio



Thank you!

Data organization

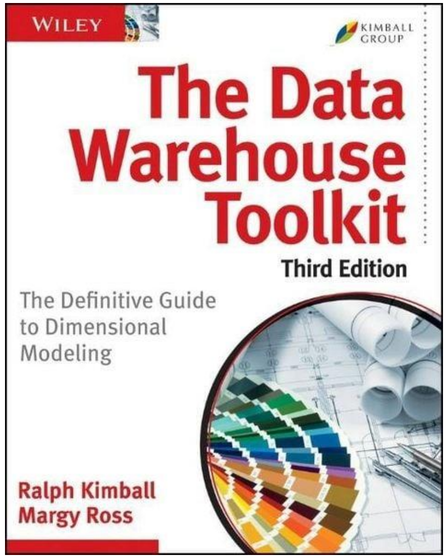
- Data organization concepts:
  - tidy data
  - normalized data (star organization)
  - data science friendly data (denormalized)

Organizing multiple tables

- Dimension vs fact tables
- De-normalization (but for data analys flat tables are more convienent)

id	artist	track	time
1	2 Pac	Baby Don't Cry	4:22
2	2Ge+her	The Hardest Part Of ...	3:15
3	3 Doors Down	Kryptonite	3:53
4	3 Doors Down	Loser	4:24
5	504 Boyz	Wobble Wobble	3:35
6	98^0	Give Me Just One Nig...	3:24
7	A*Teens	Dancing Queen	3:44
8	Aaliyah	I Don't Wanna	4:15
9	Aaliyah	Try Again	4:03
10	Adams, Yolanda	Open My Heart	5:30
11	Adkins, Trace	More	3:05
12	Aguilera, Christina	Come On Over Baby	3:38
13	Aguilera, Christina	I Turn To You	4:00
14	Aguilera, Christina	What A Girl Wants	3:18
15	Alice Deejay	Better Off Alone	6:50

id	date	rank
1	2000-02-26	87
1	2000-03-04	82
1	2000-03-11	72
1	2000-03-18	77
1	2000-03-25	87
1	2000-04-01	94
1	2000-04-08	99
2	2000-09-02	91
2	2000-09-09	87
2	2000-09-16	92
3	2000-04-08	81
3	2000-04-15	70
3	2000-04-22	68
3	2000-04-29	67
3	2000-05-06	66



Dealing with changes in the data

- Recommendations:
  - NEVER overwite a data file. Treat data files as immutable
  - Use versioning for changes in the data file, and load the latest version for new analyses, old versions to reproduce previous results
  - (pond is a library I'm working on to automatize this process)
- Like in computer code:
  - Adding new columns / rows is generally ok
  - Deleting/changing a column is not! Code will break! Add a new column instead