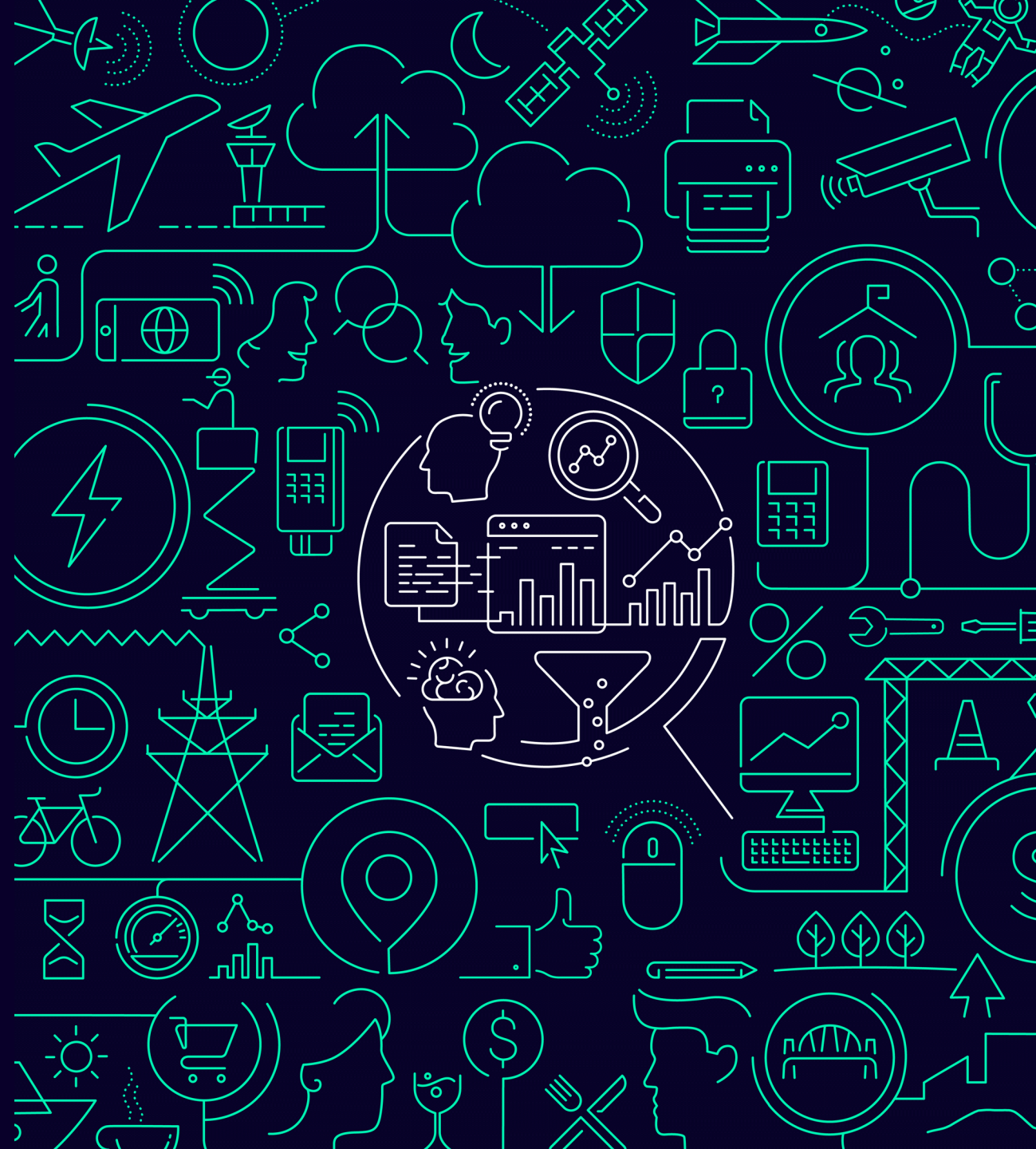


Qrious

Pandas for "Dummies"*

* The title is clearly a lie – Pandas is quite tricky



Why Pandas?



- You need a specialised toolkit for data analysis
- It's a Python library so it's easy to integrate other Python goodness
 - custom functions
 - other Python libraries
 - plotting etc

Why **not** Pandas?



- Pure Python might be better - Pandas might be **overkill** in your case
- SQL might be better in your case
 - SQL is **sufficient** for the analysis you are conducting
 - You might be more **familiar with SQL**
 - Other people in **your team** are more familiar with SQL



Is Pandas easy
to learn?



Is Pandas easy to learn?

Short answer: No



Is Pandas easy to learn?

The long answer is the rest
of the presentation ;-)



"Pandas is
powerful but
difficult to use"

Ted Petrou <https://medium.com/dunder-data/minimally-sufficient-pandas>

"[In Python] There should be one-- and preferably only one --obvious way to do it"

Tim Peters – The [Zen of Python](#)



But in Pandas there are often many ways of achieving the same result – typically only obvious to experienced Pandas practitioners

"I find that the Pandas library disobeys this [obviousness] guidance more than any other library I have encountered"

Ted Petrous – [Minimally Sufficient Pandas](#)



"Pandas is
powerful but
difficult to use"

Ted Petrou <https://medium.com/dunder-data/minimally-sufficient-pandas>

Pandas may or may not be **intuitive** to you



- Numpy or matrices experience? Pandas is built on Numpy so Numpy is a good starting point
- SQL experience? Very different but some great resources*
- R data frame experience? Helps and hurts.
- Python experience? Helps and hurts.

* A very different approach but there is good help for the transition. See:
<https://medium.com/jbennetcodes/how-to-rewrite-your-sql-queries-in-pandas-and-more-149d341fc53e>



"Pandas is
powerful but
difficult to use"

Ted Petrou <https://medium.com/dunder-data/minimally-sufficient-pandas>

80/20 rule



- Pandas has a very rich and expressive syntax
- But it is like **learning a completely new language**
 - even if you already know Python
 - sometimes contradicts Python conventions
- The **80/20 rule** applies – you can get 80% done knowing only 20%
- But what a 20% it is! The minimum amount of understanding required to be comfortable with Pandas is reasonably high
- Dangerous being a “Stack Overflow” Pandas coder



stackoverflow

Pandas Coder



Moto - "Hmmm - I wonder if this will work?"

Symptoms

- Never sure **why** code actually works
- Actually, not even sure **if** code does work
- Will try random changes (adding brackets, removing brackets, axis=0 becomes 1 and back again etc) in **hope** it will fix things
- Unable to **safely** change Pandas code without lots of checking
- Code **riddled with non-obviousness** resulting in **bewildering bugs**



stackoverflow

Pandas Coder



- Possibly safe when:
 - Checking results in Jupyter Notebook as you go
 - Small datasets where errors are easy to see
 - Nobody else has to work on the code later
- But not safe for production code

Key Pandas concepts are your **pitons**



- Don't **hope** your code is doing the right thing
- **Know** your code is correct through confident **reasoning** from solid **knowledge**
- Pandas has **gotchas** which make this challenging



8

aspects of Pandas
it pays to **properly**
understand**



** i.e. have such a rock-solid understanding of the concepts that
you can reason about what your code will do with confidence



Key insights

- You won't remember most of this



- Focus on general concepts / terms so you can look up details later and refresh your memory



DataFrames are tables on steroids



Columns

	fname	lname	age
0	Sam	Smith	23
1	Jo	Singh	12
...			

Rows

DataFrames are tables on steroids



The diagram illustrates a DataFrame as a table with four columns and three rows. The columns are labeled 'fname', 'lname', and 'age'. The rows are indexed 0, 1, and ... (representing more rows). Callouts point to the index column, the column headers, the column labels, and the rows.

	fname	lname	age
0	Sam	Smith	23
1	Jo	Singh	12
...			

DataFrames are tables on steroids



The diagram shows a DataFrame table with callouts for its components: 'Indexes' pointing to the row index column, 'Columns' pointing to the column headers, 'Column labels' pointing to the header cells, 'Built-in functionality' pointing to the table area, and 'Rows' pointing to the data rows. Two muscular arms are shown flexing on either side of the table.

	fname	lname	age
0	Sam	Smith	23
1	Jo	Singh	12
...			

Using DataFrames



Lots of handy, built-in methods e.g.

`.groupby()`

`.to_csv()`

`.multiply()`

`.fillna()`

`.transpose()`

`.pivot()`

Tricky operation made easy and semantic in Pandas – you don't have to work through the code to understand what is happening

Creating DataFrames



Multiple ways to create data frames e.g.

- List of row tuples

```
df = pd.DataFrame([(1,2), (3,4)], columns=['a', 'b'])
```

See docs on `pd.DataFrame.from_records()`

- Dict of columns

```
df = pd.DataFrame({'a': (1,2), 'b': (3,4)})
```

See docs on `pd.DataFrame.from_dict()`

- Read a CSV file

```
df = pd.read_csv('data.csv')
```

DataFrames are a collection of labelled Series



Each column is a labelled Series

df = (

name	age	city
Jo	23	Auckland
Sam	45	Christchurch
Raj	12	Dunedin
Moana	67	Wellington

)

df.age =

23	45	12	67
----	----	----	----

A Series is like a list on steroids





Your **assumptions** about
row indexes & col labels
are probably **false**

No guarantee of uniqueness



Column labels not guaranteed to be unique

	country	region	country
2	NZ	Auckland	NZ
1	UK	Essex	UK
2	Canada	Victoria	Canada
15	USA	Maine	USA
0	USA	Maine	USA
0	Canada	Ontario	Canada

Row indexes not guaranteed to be unique

No guarantee of order



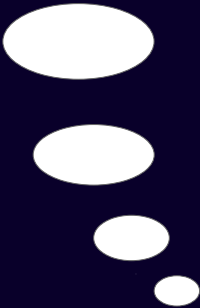
Legitimate, but unordered index

	country	region	country
2	NZ	Auckland	NZ
1	UK	Essex	UK
2	Canada	Victoria	Canada
15	USA	Maine	USA
0	USA	Maine	USA
0	Canada	Ontario	Canada

No guarantee of **order**



Indexes 2, 1, 2, 15, 0, 0!
df.loc[0] isn't the first row?! And it's actually two rows?!
I thought Pandas was like Python! Come back SQL – all is forgiven!



	country	region	country
2	NZ	Auckland	NZ
1	UK	Essex	UK
2	Canada	Victoria	Canada
15	USA	Maine	USA
0	USA	Maine	USA
0	Canada	Ontario	Canada

Default behaviour **misleading**



Default indexes and column labels are zero-based indexes e.g.

```
df = pd.DataFrame([('a', 'b', 'c'), ('d', 'e', 'f')])
```

Index and column labels automatically added

	0	1	2
0	a	b	c
1	d	e	f

Not explicitly defining an index or column labels

But once defined they are arbitrary labels and can be put out of order, repeated etc. E.g. `df = df.append([('x', 'y', 'z'),])`

0 again!

0	x	y	z
---	---	---	---

Training often correct but misleading



From a Jupyter Notebook in an on-line tutorial on Pandas

```
In [8]: df['Hired']
```

```
Out[8]: 0    Y
        1    Y
        2    N
        3    Y
        4    N
        5    Y
        6    Y
        7    Y
        8    Y
        9    N
       10    N
       11    Y
       12    Y
```

1) slicing example

You can also extract a given range of rows from a named column, like so:

```
In [9]: df['Hired'][:5]
```

```
Out[9]: 0    Y
        1    Y
        2    N
        3    Y
        4    N
```

2) item selection example

Or even extract a single value from a specified column / row combination:

```
In [10]: df['Hired'][5]
```

```
Out[10]: 'Y'
```

Training often correct but misleading



```
In [8]: df['Hired']
```

```
Out[8]: 0    Y
        1    Y
        2    N
        3    Y
        4    Y
        5    Y
        6    Y
        7    Y
        8    Y
        9    N
       10    N
       11    Y
       12    Y
        Name: Hired, dtype: object
```

To show the risks I've made versions of the df which have duplicate indexes – let's see what happens

Hired

```
0
5
2
3
4
5
6
7
8
9
10
11
5
Y
Y
Y
Y
Y
Y
Y
N
N
Y
Y
```

Version 1 – some duplicate 5s

Hired

```
a
a
a
a
a
a
a
a
a
a
a
a
a
a
Y
v
N
Y
N
Y
Y
N
N
Y
Y
```

Version 2 – all "a"s

Training often correct but misleading



```
In [8]: df['Hired']  
Out[8]: 0    Y  
        1    Y  
        2    N  
        3    Y  
        4    N  
        5    Y  
        6    Y  
        7    Y  
        8    Y  
        9    N  
       10    N  
       11    Y  
       12    Y  
Name: Hired, dtype: object
```

You can also extract a

```
In [9]: df['Hired']  
Out[9]: 0    Y  
        1    Y  
        2    N  
        3    Y  
        4    N
```

2) item selection example

Or even extract a single value from a specified column / row combination:

```
In [10]: df['Hired'][5]  
Out[10]: 'Y'
```

```
Hired  
0    Y  
5    Y  
2    N  
3    Y  
4    N  
5    Y  
6    Y  
7    Y  
8    Y  
9    N  
10   N  
11   Y  
12   Y  
Name: Hired, dtype: object  
>>> df['Hired'][5]  
5    Y  
5    Y  
5    Y  
Name: Hired, dtype: object
```

```
Hired  
a    Y  
a    Y  
a    N  
a    Y  
a    N  
a    Y  
a    Y  
a    Y  
a    Y  
a    N  
a    N  
a    Y  
a    Y  
a    Y  
a    Y  
Name: Hired, dtype: object  
>>> df['Hired'][5]  
IndexError: 5
```

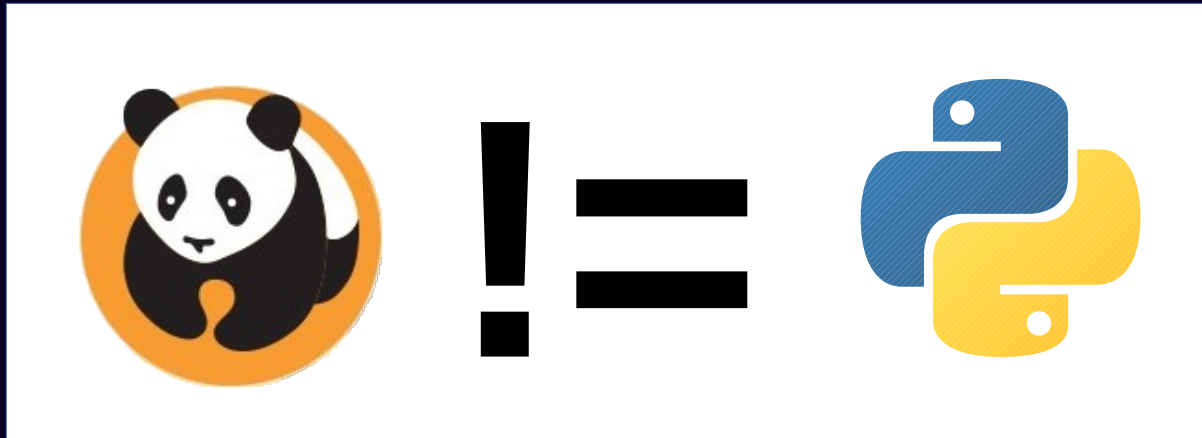
BUSTED!
Correct in example but misleading.
Pandas collects by index label (not position). Can contain duplicates or fail

More details later on how to avoid confusion

Indexing **not** like Python



- Totally unlike sequences in Python where 0 ALWAYS refers to the first item
- Thinking in Python can actually mislead you in Pandas



What to do?



`df.reset_index()` will set new zero-based index

Or when concatenating or appending, use `ignore_index=True`

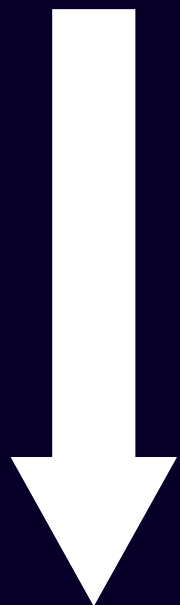
But most importantly,

always remember: there are
no guarantees about
uniqueness or order

Axis 0 is downwards; Axis 1 is across



0



	fname	lname	age
0	Sam	Smith	23
1	Jo	Singh	12
...			



1

E.g. axis & .sum()



df.sum(axis=0)

Sum values
downwards through rows
i.e. sum the columns

0 ↓

	a	b	c
0	10	15	23
1	25	3	12

35 , 18 , 35

Returns a Series

E.g. axis & .sum()

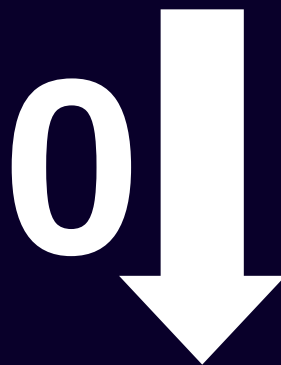


df.sum(axis=0)

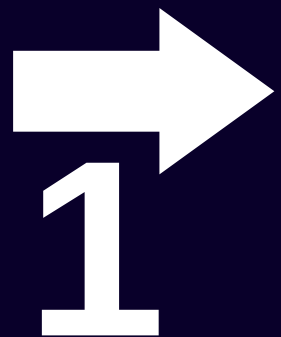
Sum values downwards through rows
i.e. sum the columns

Sum values across cols
i.e. sum rows

df.sum(axis=1)



	a	b	c	
0	10	15	23	
1	25	3	12	
	35	18	35	



	a	b	c	
0	10	15	23	48
1	25	3	12	40

Same with axis & .apply()



`df.apply(max, axis=0)`

Apply function to values going downwards i.e. per column

0 ↓

	a	b	c
0	10	15	23
1	25	3	12

25 15 23

The max of each col

`df.apply(max, axis=1)`

Apply function to values across cols i.e. values in a row

→
1

	a	b	c	
0	10	15	23	23
1	25	3	12	25



0 ↓ downwards

1 → across

Operations can apply to multiple values at once



Can make a change to all elements in a table at once without looping through rows and columns

E.g. multiplying everything by 10 all at once

2.5	3.6
1.2	11.09

 * 10 =

25	36
12	110.9



Key insights



- Multi-element operations are one of the coolest things about Pandas
- But with **great power** comes **great pain** ;-)
- Code can be hard to interpret
- Fortunately, a clear understanding of key concepts really helps



Terminology



- Pandas has **vector** operations
 - about super-efficient, all-at-once operations
 - about what happens under the hood – not just a synonym for operations which propagate across axes or elements
- Changes **propagate** (whether across an axis or individual elements in an axis / series)
- We **broadcast** changes across an axis (specifically about operations between Series and Data Frames in Pandas)
- **Element-wise** operations process every indiv value

Element-wise functions



- Element-wise functions operate on every individual element
e.g. Numpy's absolute value function:

```
np.abs(

|    |      |
|----|------|
| 2  | 111  |
| -1 | -100 |

)
```

```
= 

|            |              |
|------------|--------------|
| np.abs(2)  | np.abs(111)  |
| np.abs(-1) | np.abs(-100) |


```

```
= 

|   |     |
|---|-----|
| 2 | 111 |
| 1 | 100 |


```

Like running the function on each value separately but optimised for efficiency

- Pandas also has a syntax for applying functions to entire columns or rows

Standard row or col functions



- `.sum()`, for example, automatically sums per column

`(`

20	404
30	101

`)`.`sum()`

Axis 0 is the default

= `sum(`

20
30

`)` `sum(`

404
101

`)`

Summing all values downwards by column

=

50	505
----	-----

A Series – only one axis – like a list

- `.sum()` also totals rows if `axis = 1` e.g. `.sum(axis=1)`

Pandas can combine individual values & aggregates through vectorisation



- `df / df.sum()` does something interesting and useful
- Every column in `df` is divided by the sum of that column
- And because division is vectorised, it is like dividing every value in the column by the sum of that column
- Nice – we have a column `% :-)`

Pandas can combine individual values & aggregates through vectorisation



- `df / df.sum()` does something interesting and useful
- Every column in `df` is divided by the sum of that column
- And because division is vectorised, it is like dividing every value in the column by the sum of that column
- Nice – we have a column `% :-)`
- Confused? Once again with pictures!



Example combining indiv values & aggregates



1) `df / df.sum()`
(same as `df / df.sum(axis=0)`)

20	404
30	101

/

20	404
30	101

`.sum()`

Example combining indiv values & aggregates



1) `df / df.sum()`
(same as `df / df.sum(axis=0)`)

20	404
30	101

/

20	404
30	101

`.sum()`

2) `.sum()` applied downwards
in `df` i.e. to each column

20	404
30	101

/

$\begin{pmatrix} 20 \\ 30 \end{pmatrix} .sum()$	$\begin{pmatrix} 404 \\ 101 \end{pmatrix} .sum()$
---	---

Tackle numerator later

Resolving denominator first

Example combining indiv values & aggregates



1) `df / df.sum()`
(same as `df / df.sum(axis=0)`)

20	404
30	101

/

20	404
30	101

`.sum()`

2) `.sum()` applied downwards
in `df` i.e. to each column

20	404
30	101

/

$\begin{pmatrix} 20 \\ 30 \end{pmatrix} .sum()$	$\begin{pmatrix} 404 \\ 101 \end{pmatrix} .sum()$
---	---

3) Series returned

20	404
30	101

/

50	505
----	-----

Example combining indiv values & aggregates



1) `df / df.sum()`
(same as `df / df.sum(axis=0)`)

20	404
30	101

 /

20	404
30	101

`.sum()`

2) `.sum()` applied downwards
in df i.e. to each column

20	404
30	101

 /

(20)	(404)
30	101

`.sum()`

(404)	(101)
30	101

`.sum()`

3) Series returned

20	404
30	101

 /

50	505
----	-----

4) Division of each col by
matching sum val in series

(20) / 50	(404) / 505
30	101

We broadcast the Series [50, 505]
across the df columns.
More on broadcasting later.

Example combining indiv values & aggregates



1) `df / df.sum()`
(same as `df / df.sum(axis=0)`)

20	404	/	20	404	<code>.sum()</code>
30	101		30	101	

2) `.sum()` applied downwards
in `df` i.e. to each column

20	404	/	<code>(20) .sum()</code>	<code>(404) .sum()</code>
30	101		<code>(30) .sum()</code>	<code>(101) .sum()</code>

3) Series returned

20	404	/	50	505
30	101			

4) Division of each col by
matching sum val in series

<code>(20) / 50</code>	<code>(404) / 505</code>
<code>(30) / 50</code>	<code>(101) / 505</code>

5) Same as div of each val in col
by matching sum val in series

20 / 50	404 / 505
30 / 50	101 / 505

Example combining indiv values & aggregates



1) `df / df.sum()`
(same as `df / df.sum(axis=0)`)

20	404
30	101

 /

20	404
30	101

`.sum()`

2) `.sum()` applied downwards
in `df` i.e. to each column

20	404
30	101

 /

$(\begin{matrix} 20 \\ 30 \end{matrix})$	$(\begin{matrix} 404 \\ 101 \end{matrix})$
--	--

`.sum()` `.sum()`

3) Series returned

20	404
30	101

 /

50	505
----	-----

4) Division of each col by
matching sum val in series

$(\begin{matrix} 20 \\ 30 \end{matrix}) / 50$	$(\begin{matrix} 404 \\ 101 \end{matrix}) / 505$
---	--

5) Same as div of each val in col
by matching sum val in series

20 / 50	404 / 505
30 / 50	101 / 505

6) SUCCESS!!

0.4	0.8
0.6	0.2

Example combining indiv values & aggregates



```
df / df.sum()
```

Note - there is a safer and more readable way of writing this I'll demonstrate soon

You'll find lots of code like this in Stack Overflow so you might as well be a little bit familiar with it



Broadcasting safety message

Broadcasting step by step



Refresher – broadcasting is applying an operation (e.g. addition) between a Series and a Data Frame

1	2	+	10	100	=	11	102
3	4					13	104
5	6					15	106
7	8					17	108
9	10					19	110
11	12					21	112

Broadcasting step by step



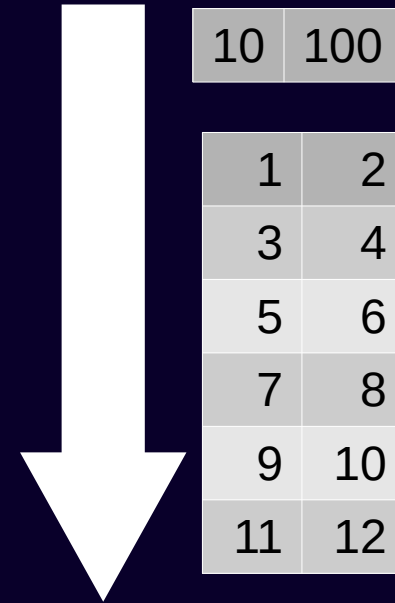
1	2
3	4
5	6
7	8
9	10
11	12

 +

10	100
----	-----

 =

11	102
13	104
15	106
17	108
19	110
21	112



Matches axis 1
and broadcasts
downwards along axis 0

1	2
3	4
5	6
7	8
9	10
11	12

result

Broadcasting step by step



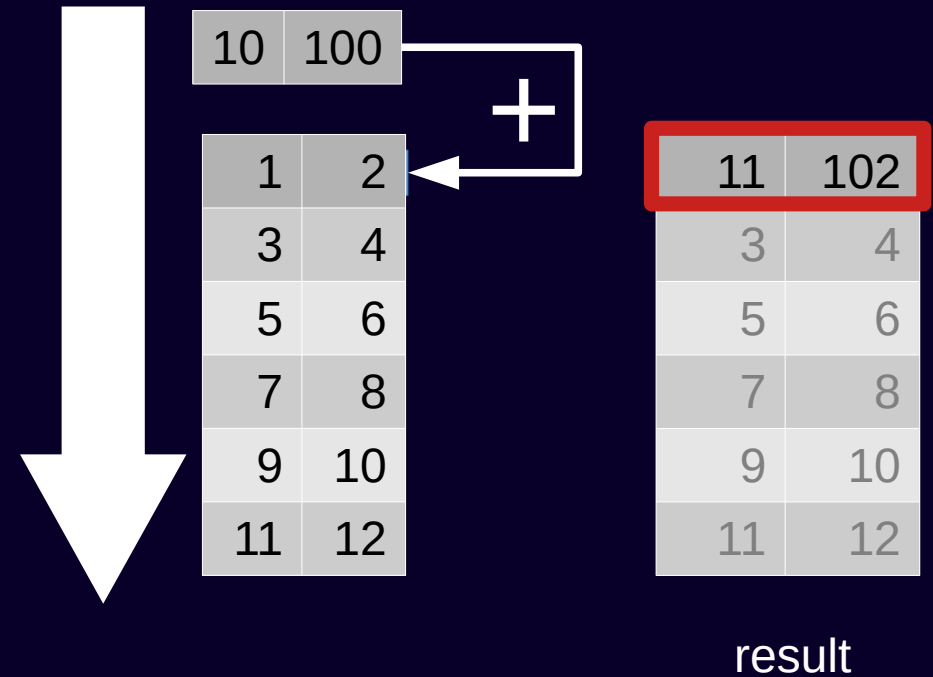
1	2
3	4
5	6
7	8
9	10
11	12

 +

10	100
----	-----

 =

11	102
13	104
15	106
17	108
19	110
21	112



Broadcasting step by step



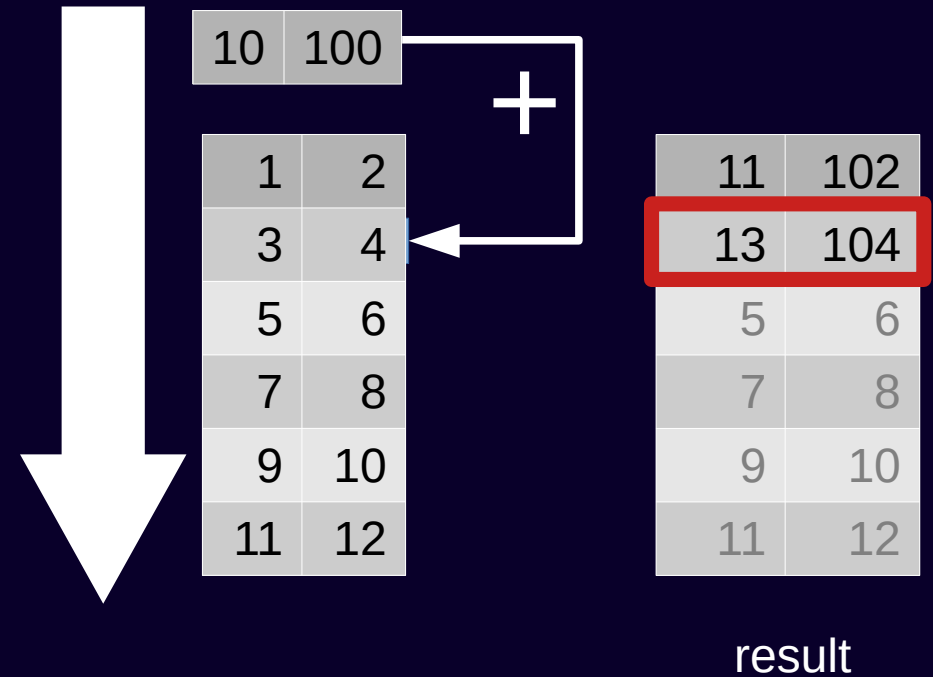
1	2
3	4
5	6
7	8
9	10
11	12

 +

10	100
----	-----

 =

11	102
13	104
15	106
17	108
19	110
21	112



Broadcasting step by step



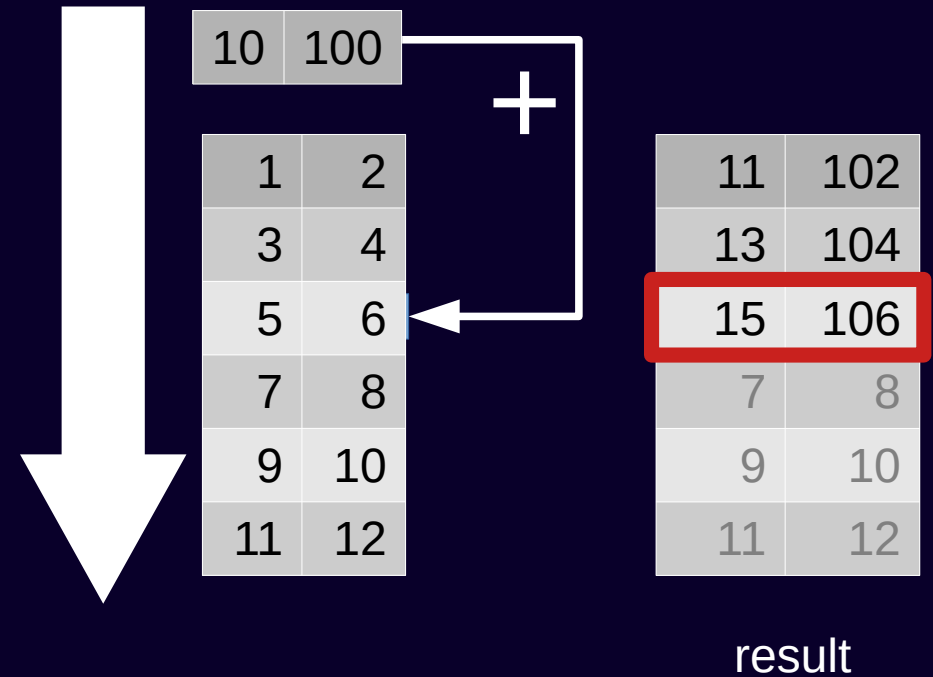
1	2
3	4
5	6
7	8
9	10
11	12

 +

10	100
----	-----

 =

11	102
13	104
15	106
17	108
19	110
21	112



Broadcasting step by step



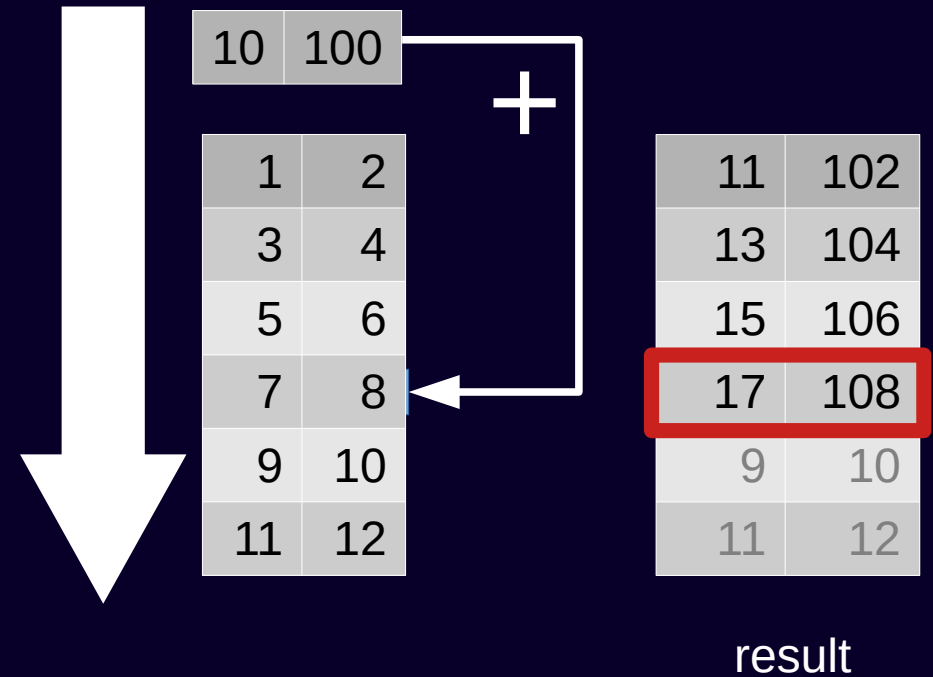
1	2
3	4
5	6
7	8
9	10
11	12

 +

10	100
----	-----

 =

11	102
13	104
15	106
17	108
19	110
21	112



Broadcasting step by step



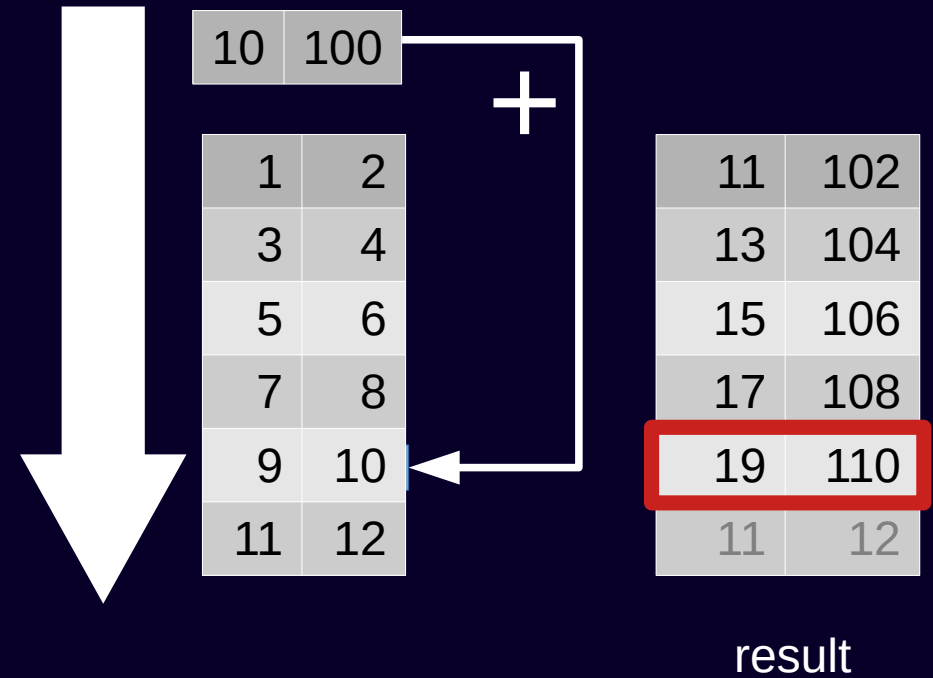
1	2
3	4
5	6
7	8
9	10
11	12

 +

10	100
----	-----

 =

11	102
13	104
15	106
17	108
19	110
21	112



Broadcasting step by step



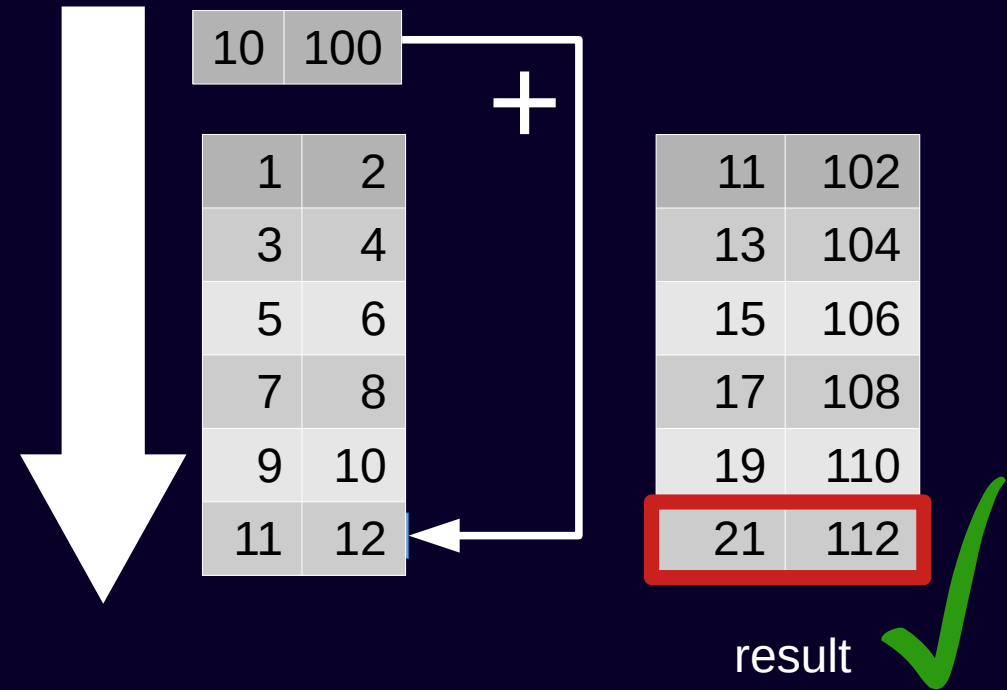
1	2
3	4
5	6
7	8
9	10
11	12

 +

10	100
----	-----

 =

11	102
13	104
15	106
17	108
19	110
21	112



result



Explicit syntax available



To specify matching axis explicitly use a different syntax

1	2
3	4
5	6
7	8
9	10
11	12

`.add(

10	100
----	-----

 , axis=1) =`

11	102
13	104
15	106
17	108
19	110
21	112

Same result as earlier but
matching axis is more explicit

Can now broadcast across cols



Broadcasting is downwards by default – need to set **matching axis** to 0 to broadcast across cols

1	2
3	4
5	6

.add(

10	100	1000
----	-----	------

, axis=0) =

11	12
103	104
1005	1006

Can now broadcast across cols



Broadcasting is downwards by default – need to set **matching axis** to 0 to broadcast across cols

1	2
3	4
5	6

`.add(`

10	100	1000
----	-----	------

`, axis=0)` =

11	12
103	104
1005	1006

10
100
1000

1	2
3	4
5	6

1	2
3	4
5	6

result

Can now broadcast across cols



Broadcasting is downwards by default – need to set **matching axis** to 0 to broadcast across cols

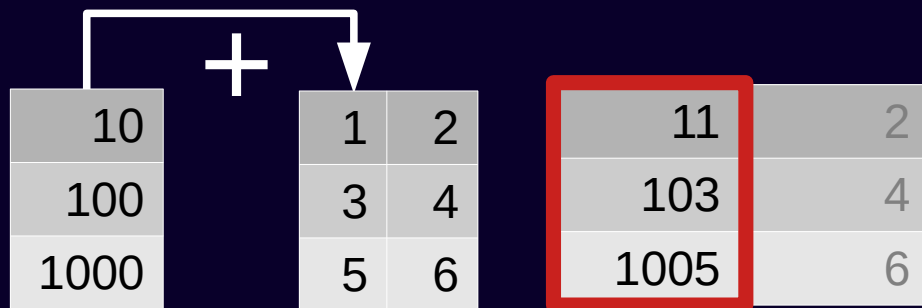
1	2
3	4
5	6

`.add(`

10	100	1000
----	-----	------

`, axis=0)` =

11	12
103	104
1005	1006

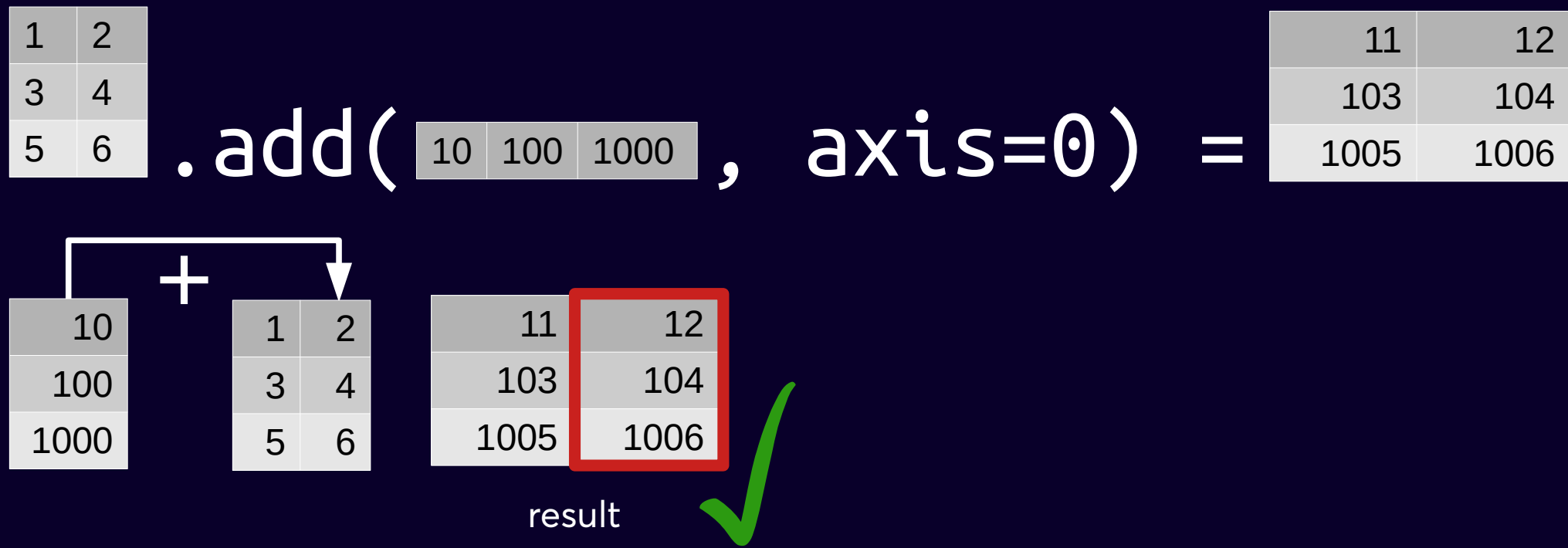


result

Can now broadcast across cols



Broadcasting is downwards by default – need to set **matching axis** to 0 to broadcast across cols



Explicit is better than implicit



If you don't specify matching axis explicitly you might successfully broadcast down the wrong axis if the shape has the same number of rows and cols.

So ... in the interests of readable, maintainable code, always use the explicit syntax in code you are keeping / maintaining

And remember – you are specifying the **matching axis**, not what it is broadcasting over

Note – the default matching axis is 1 not 0

```
.add( , axis=0)
```



Improving code from an earlier slide



`df / df.sum()`

Elegant but
opaque

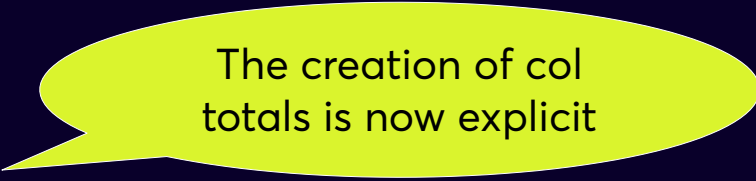
Improving code from an earlier slide



```
df / df.sum()
```

is equivalent to

```
df / df.sum(axis=0)
```



The creation of col
totals is now explicit

A yellow speech bubble callout pointing to the `axis=0` parameter in the code below. The text inside the bubble explains that this parameter makes the creation of column totals explicit.

Improving code from an earlier slide



```
df / df.sum()
```

is equivalent to

```
df / df.sum(axis=0)
```

is equivalent to

```
df.div(df.sum(axis=0), axis=1)
```

Everything explicit but mix of axis
0 and 1 potentially confusing

Improving code from an earlier slide



```
df / df.sum()
```

is equivalent to

```
df / df.sum(axis=0)
```

is equivalent to

```
df.div(df.sum(axis=0), axis=1)
```

is equivalent to

```
s_col_tots = df.sum(axis=0)
```

```
df.div(s_col_tots, axis=1)
```


Improving code from an earlier slide



Separate responsibilities
– easier to understand

This part is responsible for collecting col
totals (summing downwards along axis 0).
Nice and explicit.

```
s_col_tots = df.sum(axis=0)
```

```
df.div(s_col_tots, axis=1)
```

Improving code from an earlier slide



This part is responsible for dividing each value by its column total.

It does this by broadcasting division of the col totals. The col totals match on the columns (axis 1) and broadcast downwards across axis 0. Success!

```
s_col_tots = df.sum(axis=0)
```

```
df.div(s_col_tots, axis=1)
```

We can tell there is broadcasting – we are combining a DataFrame and a Series in the same (division) operation. Recognising broadcasting when you see it really helps.

Improving code from an earlier slide



Still have to correctly interpret the code but at least now you have a fighting chance, step by step.

```
s_col_tots = df.sum(axis=0)
```

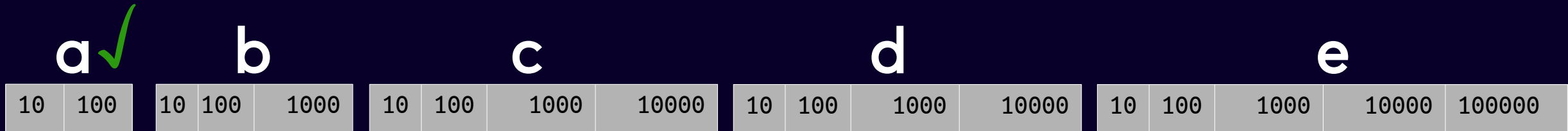
```
df.div(s_col_tots, axis=1)
```



Matching axis



When broadcasting, the Series must have the same length as the matching axis for the DataFrame



Series 'a' is the right length (2) to match df axis 1 and broadcast downwards across axis 0. So `df.add(a, axis=1)` will work

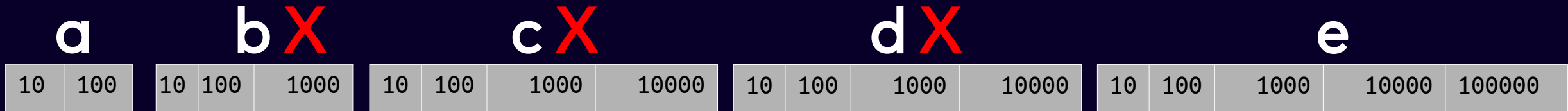
df

1	2
3	4
5	6
7	8
9	10

Matching axis



When broadcasting, the Series must have the same length as the matching axis for the DataFrame



Don't match either axis of df

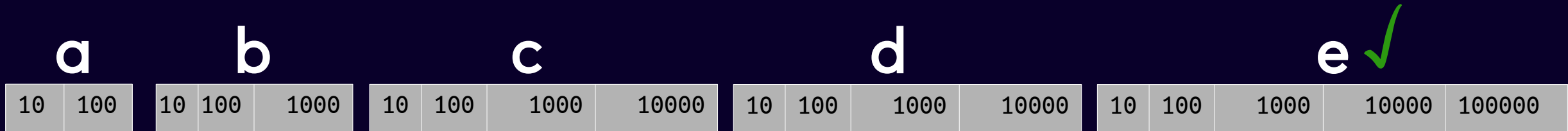
df

1	2
3	4
5	6
7	8
9	10

Matching axis



When broadcasting, the Series must have the same length as the matching axis for the DataFrame



df

1	2
3	4
5	6
7	8
9	10

Series 'e' is the right length (5) to match df axis 0 and broadcast across axis 1. So

`df.add(e, axis=0)`
will work.



.applymap(), .apply(), & functions

.applymap()



- Applies a function to every element in a data frame
- Don't use applymap if there is a vectorised alternative. The vectorised version is easier to read, less typing, and much faster e.g. squaring every value

`df ** 2` ✓ `df.applymap(lambda x: x**2)` ✗

- Lambda (anonymous) functions are often used with `.applymap()` e.g. `df.applymap(lambda x: x**2)`
- Must use `.apply()` instead for Series (in which case no axis needed or allowed)

.apply()



- .apply() also expects a function
- The function always processes either cols or rows
- ... so there needs to be an axis (0 is the default i.e. per column operations)

e.g. `df.apply(sum_the_row, axis=1)`

.apply() and Series



- Don't accidentally run `.apply(..., axis=...)` on a Series – will get a possibly confusing error like "'axis' is an invalid keyword argument"
- Sometimes `.apply()` is overkill

```
df.age.apply(lambda x: x < 10)
```

```
df.age < 10
```

Preferred –
easier to read

are the same – both return a Series where `age < 10`

.apply() is flexible



The function supplied can work with:

- the row or col as a whole e.g.
`col.sum()`
- or values identified by index (if in col)
or col label (if in row) e.g.
`row['Sat'] + row['Sun']`
`row.Sat + row.Sun`
- or a combination e.g.
`row.Sat / row.sum()`



.apply() and propagating functions



Functions can (in effect) operate on individual values if they can be propagated across the row or col they operate on. E.g. division

- return `x / x.sum()`
return `col / col.sum()`

x is commonly used but it is more explicit and readable to use col (or row) as appropriate

- all individual values in the row or col if the function vectorises
e.g. multiplying each element by 4

```
return col * 4
```

But ... if the function is vectorised then why use `.apply()`? Might be better to directly apply the function to the data frame! Faster, less typing, easier to read. Note – `.apply()` makes it easier to work with cols if you don't want to transpose data – `df / df.sum()` etc apply division row-by-row.

Why .apply() often works element-wise



If a function propagates it will effectively work element-wise
e.g. `def square(item): return item ** 2`

`(`

2	5
-1	-6

`)`.`apply(square)`

Axis 0 (per column) is the default

`=` `square(`

2
-1

`)` `square(`

5
-6

`)`

`=`

<code>square(2)</code>	<code>square(5)</code>
<code>square(-1)</code>	<code>square(-6)</code>

`=`

4	25
1	36

Functions with named values



Functions can reference labels in the row or column

```
df = pd.DataFrame([(12.5, 23), (17, 25)], columns=['Sat', 'Sun'])
```

	Sat	Sun
0	12.5	23
1	17.0	25

```
def get_weekend_tot(row):  
    return row['Sat'] + row['Sun']
```

```
df['weekend'] = df.apply(get_weekend_tot, axis=1)
```

	Sat	Sun	weekend
0	12.5	23	35.5
1	17.0	25	42.0

Functions that aggregate



Functions can aggregate the row or col or use aggregations

```
df = pd.DataFrame([(12.5, 23), (17, 25)], columns=['Sat', 'Sun'])
```

```
   Sat  Sun
0  12.5  23
1  17.0  25
```

```
def col_pct(col):
    return (100 * col) / col.sum()
```

```
df[['Sat col pct', 'Sun col pct']] = df.apply(
    col_pct, axis=0).round()
```

```
   Sat  Sun  Sat col pct  Sun col pct
0  12.5  23         42.0         48.0
1  17.0  25         58.0         52.0
```



`.loc[]` is good for
safely filtering
rows & columns

Filtering can be by **content or label**



- Filtering by **content**

e.g. all rows where fruit is "banana"

- Filtering by **label**

e.g. all columns ending in "_monthly"

e.g. rows with index between 100 and 200

Filtering options compared



`df[]`

`df.loc[]`

`df.iloc[]`

`.filter()`

`df.query()`

Filtering options compared



~ `df[]`

Filtering on data frame directly

`df.loc[]`

`df.iloc[]`

`.filter()`

`df.query()`

Only safe for column filtering when there are clear labels e.g. `df['region']`. Otherwise too many gotchas to be safe in production code

Filtering options compared



~ df[]

✓ df.loc[]

Flexible and explicit –
recommended

df.iloc[]

.filter()

df.query()

Filtering options compared



~ `df[]`

✓ `df.loc[]`

✗ `df.iloc[]`

`.filter()`

`df.query()`

Not as useful as `df.loc[]` especially given availability of `df.head()` and `df.tail()`.
Probably more valuable in a focused mathematical / engineering context

Filtering options compared



~ df[]

✓ df.loc[]

✗ df.iloc[]

✓ .filter()

df.query()

Most semantic choice for
label-based (col labels or
row indexes) filtering

A yellow speech bubble with a black outline, pointing towards the .filter() method. It contains the text: "Most semantic choice for label-based (col labels or row indexes) filtering".

Filtering options compared



- ~ `df[]`
- ✓ `df.loc[]`
- ✗ `df.iloc[]`
- ✓ `.filter()`
- ✓ `df.query()`

Newer addition to
Pandas – has its own
mini-language

Filtering options compared



`df[]`

`df.loc[]`

`df.iloc[]`

`.filter()`

`df.query()`



General-purpose so
the main focus here

A yellow speech bubble with a black outline, pointing towards the `df.loc[]` text. It contains the text "General-purpose so the main focus here".

Using .loc to filter rows and cols



```
df.loc[[100, 300], ['b', 'c']]
```

Row filtering

Column filtering

Note – double square brackets

Using .loc to filter rows and cols



```
df.loc[[100, 300], ['b', 'c']]
```

	a	b	c	d
0				
100				
200				
300				

The rows we want
[100, 300]

Using .loc to filter rows and cols



```
df.loc[[100, 300], ['b', 'c']]
```

	a	b	c	d
0				
100				
200				
300				

	a	b	c	d
0				
100				
200				
300				

The columns we want
['b', 'c']

Using .loc to filter rows and cols



```
df.loc[[100, 300], ['b', 'c']]
```

	a	b	c	d
0				
100				
200				
300				

	a	b	c	d
0				
100				
200				
300				

	a	b	c	d
0				
100				
200				
300				

The intersections of row and column filtering are selected

Powerful but strange syntax



- Numpy-flavoured syntax for filtering
 - whether you find that intuitive or not depends on your previous experience
 - Pandas filtering can be very different from Numpy filtering**
- Gotchas
 - some design decisions for convenience at expense of consistency
 - consistency with Numpy trumps consistency with Python

** Make a Numpy array and a Pandas DataFrame from the same data: `data = [(1,2), (3,4), (5,6)]`
When a Numpy array, `np_data[1] = array([3, 4])` because 1 refers to the row index
When a Pandas DataFrame, `df_data[1] = Pandas Series [2, 4, 6]` because 1 refers to column label
Very different :-)

.loc[] has square brackets



- Square brackets instead of parentheses
 - Why? A Numpy-derived short-hand
e.g. instead of `my_array[2][0]` we write `my_array[2, 0]`
- Standard parentheses with named parameters such as `rows` and `cols` would have been more Pythonic i.e. readable

.loc[] is label-based

Unless boolean filtering
– more on that later



- Selection is based on row indexes and/or column labels
- So if we have a data frame with 1, 3, 3 as index labels we can't select `df.loc[2]` (or there will be a `KeyError`)

Available
labels

	fruit	num	date	weight	name
1	apple	1	2019	36.1	Jo
3	banana	2	2018	27.5	Moana
3	cherry	3	2019	19.7	Sam

- And if an index or col label is repeated, filtering will include everything that matches e.g. `df.loc[3]` will return two rows
- Slicing is by label so we cannot use `[:-1]` etc as in normal Python – it is actually from one label to another

Always rows first (then columns)

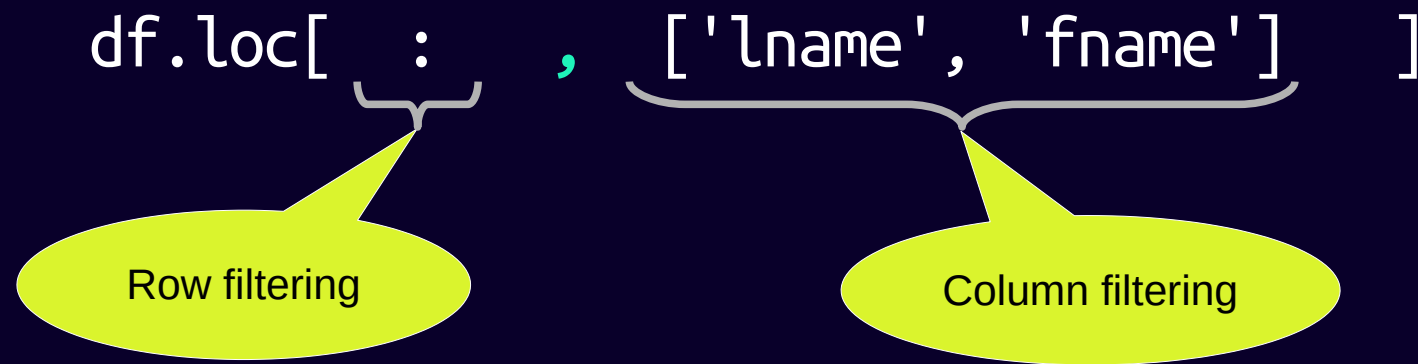


- This **guarantee** is what makes `.loc[]` the safe choice for production
- Not as **explicit** as keyword parameters in standard Python functions

.loc[] always needs rows



Can't just select columns using .loc[] - must use slice : to refer to all rows e.g.



will return a data frame for the lname and fname columns only but for all rows

.loc[] is **safer** than df[]



Don't use df[] in **production code** for filtering by content. Use .loc[] or .query()



- Narrows down the range of quirks to be understood – more than enough to **master** in .loc[]
- Mixing different approaches can be confusing – similar syntaxes yield completely different results

Gotchas** where similar syntax but very different results

** Identifying gotchas is not necessarily a criticism – sometimes design trade-offs have to be made and there are contradictory principles to achieve consistency with. But they are still gotchas ;-)

[] vs .loc[]



df =

	0	1
0	apple	banana
1	cherry	date

- df[0] vs df.loc[0]
 - df[0] is the column with the label 0 i.e. the Series ['apple', 'cherry']
 - df.loc[0] is the row with the label 0 i.e. the Series ['apple', 'banana']

df.loc[0]

	0	1
0	apple	banana
1	cherry	date



.loc[] vs .iloc[]



df =

	0	1
0	apple	banana
1	cherry	date

- `df.loc[0:1]` vs `df.iloc[0:1]`
 - `df.loc[0:1]` includes both ends because it is label-based not index-based so is `[('apple', 'banana'), ('cherry', 'date')]`
 - `df.iloc[0:1]` excludes the final index value like typical Python so is `[('apple', 'banana')]` only

`df.loc[0:1]` {

	0	1
0	apple	banana
1	cherry	date

 } `df.iloc[0:1]`

Slicing confusion



df =

	0	1	2
0	apple	banana	cherry
1	date	elderberry	fig

- `df[0] = ['apple', 'date']`
- `df[0:1] = ['apple', 'banana', 'cherry']` **

df[0]

	0	1	2
0	apple	banana	cherry
1	date	elderberry	fig

** Slicing inside `df[]` slices rows. The official justification - "This is provided largely as a convenience since it is such a common operation" https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html. The Pandas creator notes the tension in "Python for Data Analysis" - "This might seem inconsistent for some readers , but this syntax arose out of practicality and nothing more" (p.127)



More powerful filtering

More powerful filtering



- Pandas allows very flexible filtering using `.loc[]`
- e.g.

```
df.loc[  
    (df.year.isin([2017, 2018])  
     &  
     (df.age == 'Senior'))]
```
- More on that after discussion of boolean filtering
- Consider `.query()` as an alternative

.query()



- A mini-language e.g. backticks for column names with gaps, @ for variables in scope etc
- Added more recently than .loc[] so using .query() doesn't mean you can ignore .loc[] - it appears in examples, existing code etc
- Good documentation:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html>

<http://jose-coto.com/query-method-pandas>

.query() examples



- Example of .loc[] vs .query():

```
grades_df.loc[grades_df.Test_3.isin([98, 99, 100])]
```

```
grades_df.query("Test_3 in [98, 99, 100]")
```

- Can use "and" and "or" in query and usual operator precedence applies so no need to add extra parentheses (as in .loc[])

```
df.query("year == 2019 and suburb in ('Mt Albert', 'Mt Eden')")
```

Boolean filtering is important



False



True

Boolean filtering illustrated



F False **T** True

	T	F	F	T	F
apple	1	2019	36.1	Jo	
banana	2	2018	27.5	Moana	
cherry	3	2019	19.7	Sam	

Boolean array

Data frame

Boolean filtering illustrated



F False **T** True

Applying boolean array to columns

	T	F	F	T	F
apple	1	2019	36.1	Jo	
banana	2	2018	27.5	Moana	
cherry	3	2019	19.7	Sam	



apple	36.1
banana	27.5
cherry	19.7

"False" columns will be removed

Boolean filtering illustrated



F False **T** True

T	F	F	T	F
apple	1	2019	36.1	Jo
banana	2	2018	27.5	Moana
cherry	3	2019	19.7	Sam



apple	36.1
banana	27.5
cherry	19.7

Boolean array applied to rows

T	apple	1	2019	36.1	Jo
F	banana	2	2018	27.5	Moana
T	cherry	3	2019	19.7	Sam



"False" rows will be removed

apple	1	2019	36.1	Jo
cherry	3	2019	19.7	Sam

Must match dimensions of data in axis



Obviously the boolean array needs to have the same number of values as the axis being filtered

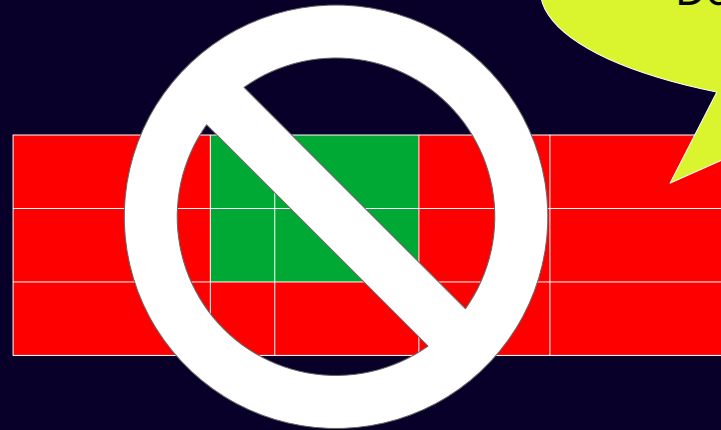
The diagram illustrates the concept of boolean array filtering on a 2D array. It shows a 4x5 grid of colored cells (green and red) with a corresponding 4x5 grid of boolean values (green and red) above it. The boolean values are used to filter the rows of the data grid. The first two rows are filtered out (marked with red X), and the last two rows are kept (marked with green checkmarks).

apple	1	2019	36.1	Jo
banana	2	2018	27.5	Moana
cherry	3	2019	19.7	Sam

Boolean filtering restrictions



fruit	num	date	weight	name
apple	1	2019	36.1	Jo
banana	2	2018	27.5	Moana
cherry	3	2019	19.7	Sam



Can't apply this DataFrame as a boolean filter, for example

- Any single boolean filter is for rows or columns only (not both at same time)
- Can't apply a boolean matrix to filter the data frame the same way you might apply a data frame matrix to another data frame matrix
- But can use two boolean filters in `.loc[]` - one for rows and one for cols

Conditional filtering is boolean filtering



fruit	num	date	weight	name
apple	1	2019	36.1	Jo
banana	2	2018	27.5	Moana
cherry	3	2019	19.7	Sam

- You can have one boolean filter for rows in the first part of `loc[]` and another by cols in the second
- e.g. filtering rows by `fruit == 'cherry'`

apple
banana
cherry

and cols by `columns.str.startswith('w')`

fruit	num	date	weight	name
-------	-----	------	--------	------

Passing in boolean indexes



The following are equivalent:

- `df.loc[df.fruit == 'cherry', df.columns.str.startswith('w')]`

- `df.loc[

apple
banana
cherry

,

fruit	num	date	weight	name
-------	-----	------	--------	------

]`

	fruit	num	date	weight	name
apple	apple	1	2019	36.1	Jo
banana	banana	2	2018	27.5	Moana
cherry	cherry	3	2019	19.7	Sam

weight
0 19.7

Multiple conditions



- Use parentheses
- Use parentheses (yes – seriously!)
- Don't use ~~and~~ or ~~or~~ – instead use & and |
- e.g. `df.loc[(df.year == 2019) & (df.age == 'Senior')]`

How a DataFrame is filtered affects what is returned



If filtering for one column only the default is to return a Series

```
data = [  
    (1, 2),  
    (3, 4),  
]
```

```
df = DataFrame(data, columns=['a', 'b'])
```

```
df.loc[:, 'b']
```

```
Series [ 2, 4]
```

How to guarantee a DataFrame



```
Data = [  
    (1, 2),  
    (3, 4),  
]
```

Same source
DataFrame

```
df = DataFrame(data, columns=['a', 'b'])
```

- To get back a DataFrame pass in a list of columns (albeit with only one column inside)

```
df.loc[:, ['b']]
```

DataFrame [[2], [4]] (vs Series [2, 4])

- Sometimes `.to_frame()` will be useful to turn a series into a DataFrame

Check if changes are **persisting**



- In Pandas it is not always clear if changes will persist or not

- `df`

	0	1
0	a	b
1	c	d

- `df.append([('e', 'f')]. ignore_index=True)`

- `df?`

	0	1		0	1	
0	a	b	OR	0	a	b
1	c	d		1	c	d
				2	e	f

Check if changes are **persisting**



- In Pandas it is not always clear if changes are persisting or not

- `df`

	0	1
0	a	b
1	c	d

- `df.append([('e', 'f')]. ignore_index=True)`

- `df?`

	0	1
0	a	b
1	c	d



OR

	0	1
0	a	b
1	c	d
2	e	f



Filtering data frames returns views



- Applying an operation on a filtered data frame is operating on the same parts of the original data frame



i.e. what you do matters! If you want to operate on a copy, use `.copy()`



If you're not sure ...

CHECK!

Eight key concepts

- 1) DataFrames are enhanced tables with rows and columns (like spreadsheets)
- 2) Row indexes and column labels are not guaranteed unique or in order
- 3) Axis 0 is downwards through rows; axis 1 is across columns
- 4) Operations can be applied to multiple elements / rows/ cols without looping

8

Eight key concepts

- 5) Prefer `.loc[]` for filtering
- 6) Become comfortable with Boolean filtering
- 7) Know when you're getting back a Series or a Data Frame
- 8) Ensure you know if changes are persisting or not

8

Practical tasks



CSVs – reading and writing



- `pd.read_csv()`
- `df.to_csv(..., index=False)` ## `index=False` stops the index being added as the first column
- `'sep'` and `'delimiter'` are synonymous parameters

Demo DataFrame for next slides



Source dataframe:

df

	year	city	suburb	club	age	freq	fees
0	2017	Auckland	Mt Albert	MABC	Senior	33.0	2000
1	2017	Auckland	Mt Albert	MABC	Junior	70.0	2300
2	2018	Auckland	Mt Albert	MABC	Senior	39.0	2100
3	2018	Auckland	Mt Albert	MABC	Junior	NaN	2450
4	2019	Auckland	Mt Albert	MABC	Senior	40.0	2200
5	2019	Auckland	Mt Albert	MABC	Junior	70.0	2750
6	2016	Auckland	Mt Albert	MABC	Senior	33.0	2000
7	2015	Auckland	Mt Albert	MABC	Junior	70.0	2300
8	2018	Auckland	Mt Eden	Gillies Ave BC	Senior	120.0	3000
9	2018	Auckland	Mt Eden	Gillies Ave BC	Junior	234.0	5000
10	2019	Auckland	Mt Eden	Gillies Ave BC	Senior	124.0	3100
11	2019	Auckland	Mt Eden	Gillies Ave BC	Junior	265.0	5575
12	2018	Wellington	Mirimar	MBC	Senior	67.0	1100
13	2018	Wellington	Mirimar	MBC	Junior	183.0	2200
14	2019	Wellington	Mirimar	MBC	Senior	66.0	1000
15	2019	Wellington	Mirimar	MBC	Junior	187.0	2350

Making new calculated columns



```
df_extra = df.copy()
```

```
df_extra['tot_fees'] = (df_extra.fees * df_extra.freq).round()
```

Note – can't use dot notation for new field's name

```
df_extra.head(5)
```

	year	city	suburb	club	age	freq	fees	tot_fees
0	2017	Auckland	Mt Albert	MABC	Senior	33.0	2000	66000.0
1	2017	Auckland	Mt Albert	MABC	Junior	70.0	2300	161000.0
2	2018	Auckland	Mt Albert	MABC	Senior	39.0	2100	81900.0
3	2018	Auckland	Mt Albert	MABC	Junior	NaN	2450	NaN
4	2019	Auckland	Mt Albert	MABC	Senior	40.0	2200	88000.0

Note – we can't `.astype('int')` `tot_fees` because of NaN in field

https://pandas.pydata.org/pandas-docs/stable/user_guide/gotchas.html#support-for-integer-na

Controlling columns produced



- `.drop()` can drop individual columns or a list of columns e.g.
`df.drop(columns='year')`
`df.drop(columns=['year', 'age'])`
- The columns dropped are only actually removed from the data frame if we set `inplace=True` - otherwise it is only on what is returned
- When you calculate a new field you want to be able to give it a useful name. You may also want to override the names supplied in the original inputs. `.rename()` is useful e.g.
`df.rename(columns={'year': 'year_of_play'})`
- If you want to reset all the column names it might be easiest to set the columns attribute directly e.g. `df.columns = ['year_of_play', etc]`

Grouping



- See <https://www.shanelynn.ie/summarising-aggregation-and-grouping-data-in-python-pandas/>
- `.groupby()` returns a special `DataFrameGroupBy` object which you can't really "see" e.g. by printing it. But it lets you get all sorts of interesting results.
- The easiest is by using the `.describe()` method on it.
 - Note - you can't filter describe to only display results for selected column labels only data types. To specify individual fields to use do filtering earlier
- Examples
 - `df.groupby('year').describe()`
 - `df.loc[:, 'freq'].describe()`

Grouping – selecting columns



Filtered columns before passing through to `.groupby()` and `.describe()`

```
df.loc[:, ['year', 'freq']].groupby('year').describe()
```

	freq count	mean	std	min	25%	50%	75%	max
year								
2015	1.0	70.000000	NaN	70.0	70.00	70.0	70.00	70.0
2016	1.0	33.000000	NaN	33.0	33.00	33.0	33.00	33.0
2017	2.0	51.500000	26.162951	33.0	42.25	51.5	60.75	70.0
2018	5.0	128.600000	80.568604	39.0	67.00	120.0	183.00	234.0
2019	6.0	125.333333	86.226833	40.0	67.00	97.0	171.25	265.0

.count()



.count() counts all **NON-missing** values whereas .size() counts ALL values including missing

```
df.groupby('year').count()
```

	city	suburb	club	age	freq	fees
year						
2015	1	1	1	1	1	1
2016	1	1	1	1	1	1
2017	2	2	2	2	2	2
2018	6	6	6	6	5	6
2019	6	6	6	6	6	6

There is one missing value in the freq column

.size()



.size() is for the df as a whole, not for each column

```
df.groupby('year').size()
```

```
year
2015    1
2016    1
2017    2
2018    6
2019    6
```

.first() and .last()



These methods use the order of the data frame they are based on. You may need to apply `sort_values()` beforehand

```
df.groupby('suburb').first()
```

	year	city	club	age	freq	fees
suburb						
Mirimar	2018	Wellington	MBC	Senior	67.0	1100
Mt Albert	2017	Auckland	MABC	Senior	33.0	2000
Mt Eden	2018	Auckland	Gillies Ave BC	Senior	120.0	3000

.min() and .max()



Note that this method works on strings as well as numbers

```
df.groupby('suburb').min()
```

suburb	year	city	club	age	freq	fees
Mirimar	2018	Wellington	MBC	Junior	66.0	1000
Mt Albert	2015	Auckland	MABC	Junior	33.0	2000
Mt Eden	2018	Auckland	Gillies Ave BC	Junior	120.0	3000

```
df.groupby('suburb').max()
```

suburb	year	city	club	age	freq	fees
Mirimar	2019	Wellington	MBC	Senior	187.0	2350
Mt Albert	2019	Auckland	MABC	Senior	70.0	2750
Mt Eden	2019	Auckland	Gillies Ave BC	Senior	265.0	5575

.groupby fields



Note that this method works on strings as well as numbers

```
df.groupby('city').sum()[['freq']].add_prefix('Sum_of_')
```

	Sum_of_freq
city	
Auckland	1098.0
Wellington	503.0

Flexible aggregation



We can get different aggregate types for different fields e.g. min for one and max for another. In SQL it is easy. How do we do it in pandas? We need to pass in a dictionary.

```
df.groupby('city').agg({'fees': 'sum', 'freq': 'max'})
```

	fees	freq
city		
Auckland	34775	265.0
Wellington	6650	187.0

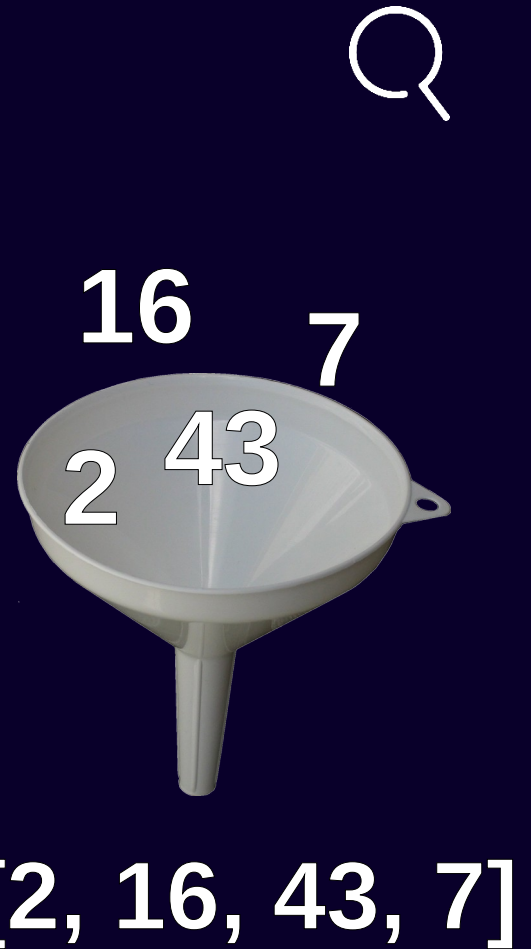
The result will have correct aggregate results but the labels will need fixing. Time to use `.rename()` again.

Custom aggregation

Aggregation functions take in **multiple** items and return a **single** item e.g. mean handles the numbers in the aggregated field which are in the group and returns a single arithmetic mean

list deserves special mention – it gathers up the results into a single list

You can also use your own functions (named or anonymous lambdas) to do anything you want to the gathered values – as long as a single value is returned



Updating



```
df3 = df.copy()
```

```
df3.loc[3:4, ['fees', 'freq']] = df3.loc[3:4, ['fees', 'freq']].multiply(100)
```

```
df3.head(7)
```

	year	city	suburb	club	age	freq	fees
0	2017	Auckland	Mt Albert	MABC	Senior	33.0	2000
1	2017	Auckland	Mt Albert	MABC	Junior	70.0	2300
2	2018	Auckland	Mt Albert	MABC	Senior	39.0	2100
3	2018	Auckland	Mt Albert	MABC	Junior	NaN	245000
4	2019	Auckland	Mt Albert	MABC	Senior	4000.0	220000
5	2019	Auckland	Mt Albert	MABC	Junior	70.0	2750
6	2016	Auckland	Mt Albert	MABC	Senior	33.0	2000

Union joining (appending)



- `.concat([df1, df2, ...])`
- `axis=0` (rows) for appending (the default), `axis = 1` (columns) for putting alongside
- https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html
- Need same column names if trying to append (unlike SQL UNION) – otherwise all NaNs in the non-aligned cells

Appending without compatible cols



Demo DataFrame to append:

```
df2 = pd.DataFrame(  
    [[2020, 'Christchurch', 'Yaldhurst', 'YBC', 'Junior', 199, 2_700], ])
```

Note different column labels compared
with df we're concatenating with

```
df2  
      0      1      2      3      4      5      6  
0  2020  Christchurch  Yaldhurst  YBC  Junior  199  2700
```

Faulty concatenated data



Oops! We have all the columns from both data frames and NaNs filling in all the mismatched areas :-)

```
df3 = pd.concat([df2, df])
```

```
df3.head(3)
```

	0	1	2	3	...	fees	freq	suburb	year
0	2020.0	Christchurch	Yaldhurst	YBC	...	NaN	NaN	NaN	NaN
0	NaN	NaN	NaN	NaN	...	2000.0	33.0	Mt Albert	2017.0
1	NaN	NaN	NaN	NaN	...	2300.0	70.0	Mt Albert	2017.0

Correctly concatenated data



Ensuring compatible columns:

```
df2 = pd.DataFrame(  
    [[2020, 'Christchurch', 'Yaldhurst', 'YBC', 'Junior', 199, 2_700], ],  
    columns=df.columns)
```

df2

	year	city	suburb	club	age	freq	fees
0	2020	Christchurch	Yaldhurst	YBC	Junior	199	2700

```
df3 = pd.concat([df2, df])
```

Success because of aligned column names:

```
df3.head(3)
```

	year	city	suburb	club	age	freq	fees
0	2020	Christchurch	Yaldhurst	YBC	Junior	199.0	2700
0	2017	Auckland	Mt Albert	MABC	Senior	33.0	2000
1	2017	Auckland	Mt Albert	MABC	Junior	70.0	2300



Labels matter, not order



Compatible columns but inconsistent order (swapped year and city)

```
df2 = pd.DataFrame(  
    [['Christchurch', 2020, 'Yaldhurst', 'YBC', 'Junior', 199, 2_700], ],  
    columns=['city', 'year', 'suburb', 'club', 'age', 'freq', 'fees'])
```

df2

	city	year	suburb	club	age	freq	fees
0	Christchurch	2020	Yaldhurst	YBC	Junior	199	2700

```
df3 = pd.concat([df2, df], sort=False)
```

Succeeded even though columns out of order:

```
df3.head(3)
```

	city	year	suburb	club	age	freq	fees
0	Christchurch	2020	Yaldhurst	YBC	Junior	199.0	2700
0	Auckland	2017	Mt Albert	MABC	Senior	33.0	2000
1	Auckland	2017	Mt Albert	MABC	Junior	70.0	2300



Joining on a key

- Like an inner join in SQL
- merge is what is usually needed unless joining on index (`.join()` will do)
- See https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

Example join



```
df2 = pd.DataFrame([('Auckland', 'Fuel tax'), ], columns=['city', 'Tax'])
```

```
df2
```

```
   city      Tax
0  Auckland Fuel tax
```

```
pd.merge(left=df, right=df2, how='left', on='city')
```

```
   year      city      suburb      club      age      freq      fees      Tax
0  2017  Auckland  Mt Albert      MABC  Senior    33.0    2000  Fuel tax
1  2017  Auckland  Mt Albert      MABC  Junior    70.0    2300  Fuel tax
...
6  2016  Auckland  Mt Albert      MABC  Senior    33.0    2000  Fuel tax
7  2015  Auckland  Mt Albert      MABC  Junior    70.0    2300  Fuel tax
8  2018  Auckland      Mt Eden  Gillies Ave BC  Senior   120.0    3000  Fuel tax
...
11 2019  Auckland      Mt Eden  Gillies Ave BC  Junior   265.0    5575  Fuel tax
12 2018  Wellington  Mirimar      MBC      Senior    67.0    1100      NaN
...
```

Column percentages



Strategy – make a data frame of calculated percentages and set it as a new column

```
df['pct_freq'] = (  
    df.freq  
    .apply(lambda col: (100 * col) / col.sum())  
    .round(2)  
)
```

Because df.freq is a Series .apply() will not accept an axis argument

```
df['pct_freq'] = (  
    df.freq  
    .multiply(100)  
    .div(  
        df.freq.sum()  
    ).round(2)  
)
```

A more elegant alternative



Column percentages by group



Strategy – make df with totals by group and join it to df using group as key. Then set new column to results of simple calculation of percentages.

```
df_yearly_freq_sum = (  
    df.groupby(['year'])  
    .agg({'freq': 'sum'})  
    .add_prefix('tot_')  
)
```

```
df_yearly_freq_sum  
   tot_freq  
year  
2015      70.0  
2016      33.0  
...
```

Using an easily understood** join ready for simple percentage calculation

A yellow speech bubble with a black outline and a tail pointing towards the bottom left. The text inside is black and centered.

```
df3 = pd.merge(df, df_yearly_freq_sum, on='year')
```

** Especially by people with SQL experience

Column percentages by group ...



```
df3['annual_freq_pct'] = (100*(df3.freq / df3.tot_freq)).round()
```

```
df3.loc[:, ['year', 'city', 'suburb', 'club', 'age', 'freq', 'annual_freq_pct']]  
    .sort_values(['year', 'city', 'suburb', 'club', 'age'])  
    .reset_index(drop=True).head()
```

Tip - `.reset_index()` is also a great way of turning a multi-index into columns (e.g. after a groupby operation)

Data frame already has percentages calculated – just selecting columns to display and resetting index after sorting

	year	city	suburb	club	age	freq	annual_freq_pct
0	2015	Auckland	Mt Albert	MABC	Junior	70.0	100.0
1	2016	Auckland	Mt Albert	MABC	Senior	33.0	100.0
2	2017	Auckland	Mt Albert	MABC	Junior	70.0	68.0
3	2017	Auckland	Mt Albert	MABC	Senior	33.0	32.0
4	2018	Auckland	Mt Albert	MABC	Junior	NaN	NaN

Row percentages



```
df_hrs = pd.DataFrame(  
    [('Jo', 36, 6, 6, 6.5, 7, 5), ('Sam', 24, 4, 7, 7, 0, 4)],  
    columns=['worker', 'age', 'mon', 'tue', 'wed', 'thur',  
            'fri'])
```

```
df_hrs  
  worker  age  mon  tue  wed  thur  fri  
0     Jo   36    6    6  6.5    7    5  
1     Sam  24    4    7  7.0    0    4
```

And make a spare one for a comparison later

```
df_hrs2 = df_hrs.copy()
```

Row percentages ...



.sum() will total **all** numeric fields so you may need to filter the columns first. In this example we have added another numeric field we do not want included in total (age) but we have failed to filter it out from the data being summed.

```
df_hrs['tot_hrs'] = df_hrs.sum(axis=1)
```

df_hrs

	worker	age	mon	tue	wed	thur	fri	tot_hrs	
0	Jo	36	!	6	6	6.5	7	5	66.5
1	Sam	24	4	7	7.0	0	4	46.0	

Oops! Let's do it again with age removed from the data being summed

Row percentages ...



```
df_hrs2['tot_hrs'] = df_hrs2.drop('age', axis=1).sum(axis=1)
```

df_hrs2

	worker	age	mon	tue	wed	thur	fri	tot_hrs
0	Jo	36	6	6	6.5	7	5	30.5
1	Sam	24	4	7	7.0	0	4	22.0

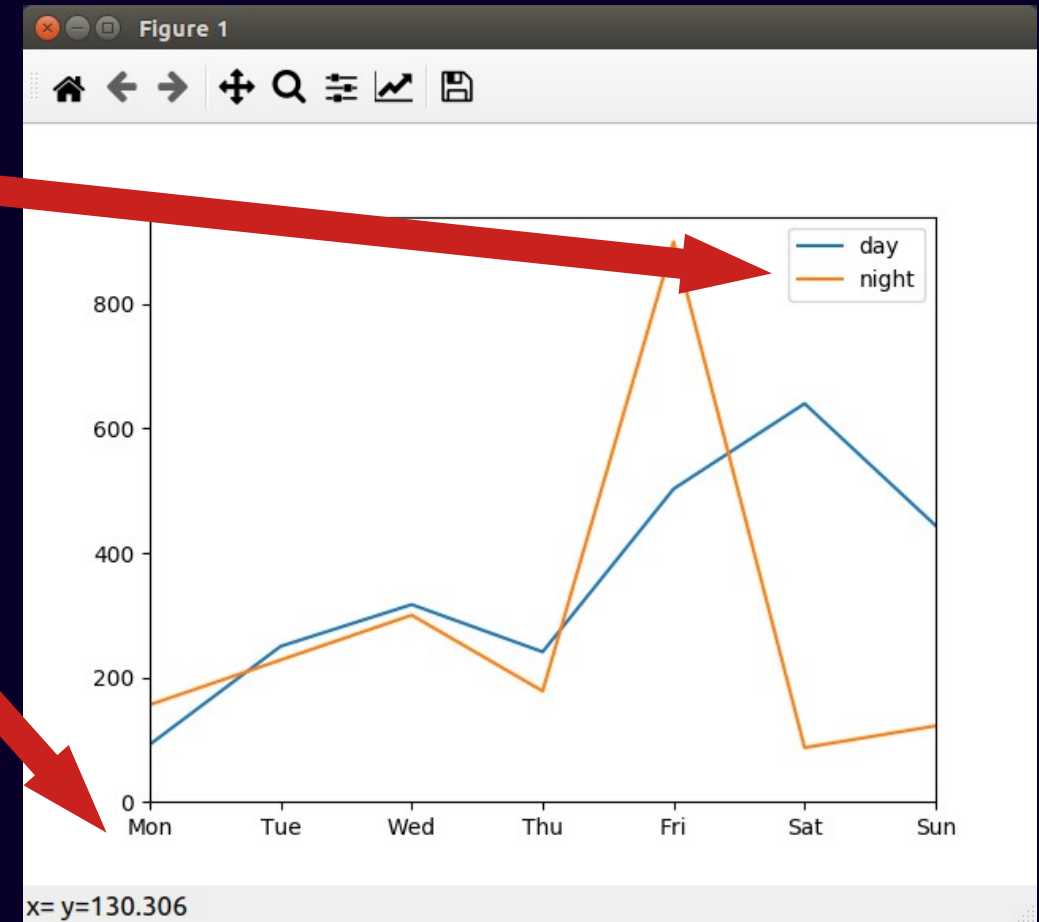


Plotting two columns



```
from matplotlib import pyplot as plt
```

```
df = pd.DataFrame(  
    {'day': [92, 250, 317,  
            241, 503, 640, 444],  
     'night': [156, 228, 300,  
              178, 900, 87, 122]},  
    index=['Mon', 'Tue',  
          'Wed', 'Thu', 'Fri',  
          'Sat', 'Sun'])  
df.plot()  
plt.ylim(0)  
plt.show()
```



Plotting result of aggregation



```
from matplotlib import pyplot as plt
df = pd.DataFrame([('Snooker', 'South', 4), ('Snooker', 'South', 5), ('Badminton', 'South', 6),
                  ('Snooker', 'North', 14), ('Snooker', 'North', 12), ('Badminton', 'North', 18), ('Badminton', 'North', 28),
                  ('Football', 'South', 6), ('Football', 'South', 7),
                  ('Frisbee Golf', 'South', 3),
                  ('Football', 'North', 11),
                  ('Frisbee Golf', 'North', 11),
                  ('Football', 'North', 12), ('Football', 'North', 20),
                  ], columns=['Sport', 'Area', 'Score'])
```

```
df2 = (
    df.groupby(['Sport', 'Area'])
      ['Score']
      .mean())
```

```
df2.unstack(level=1)
```

```
df2.plot()
```

```
plt.title('Average Sport Score')
```

```
plt.ylim(0)
```

```
plt.show()
```

Level 0 of multi-index is 'Sport' and 1 is 'Area' so unstack 'Area' part into columns



Tuple results unpacked into cols



Strategy - make a data frame and set new columns to it – simple

```
df = pd.DataFrame({'nums': [1,2,3,4]})
def powers(num):
    return num ** 2, num ** 3
data = df.nums.apply(powers).to_list()
df_square_cube = pd.DataFrame(data, index=df.index)
df[['square', 'cube']] = df_square_cube
```

Makes a list of one-item tuples from a list of items

	nums	square	cube
0	1	1	1
1	2	4	8
2	3	9	27
3	4	16	64

Delete rows with missing values



```
>>> data [('Jo', 1, 5), ('Sam', None, None), ('Avi', None, 3),
          ('Noor', 4, None), ('Cat', 6, 6)]
>>> df = pd.DataFrame(data, columns=['name', 'start', 'end'])
>>> df
   name  start  end
0    Jo    1.0  5.0
1   Sam   NaN  NaN
2   Avi   NaN  3.0
3  Noor    4.0  NaN
4   Cat    6.0  6.0
>>> df.dropna(subset=['start', 'end'])
   name  start  end
0    Jo    1.0  5.0
1   Sam   NaN  NaN
2   Avi   NaN  3.0
3  Noor    4.0  NaN
4   Cat    6.0  6.0
```

Delete rows with missing values



```
>>> data [('Jo', 1, 5), ('Sam', None, None), ('Avi', None, 3),  
         ('Noor', 4, None), ('Cat', 6, 6)]
```

```
>>> df = pd.DataFrame(data, columns=['name', 'start', 'end'])
```

```
>>> df
```

	name	start	end
0	Jo	1.0	5.0
1	Sam	NaN	NaN
2	Avi	NaN	3.0
3	Noor	4.0	NaN
4	Cat	6.0	6.0

```
>>> df.dropna(subset=['start', 'end'])
```

	name	start	end
0	Jo	1.0	5.0
1	Sam	NaN	NaN
2	Avi	NaN	3.0
3	Noor	4.0	NaN
4	Cat	6.0	6.0

Delete rows with missing values



```
>>> data [('Jo', 1, 5), ('Sam', None, None), ('Avi', None, 3),  
         ('Noor', 4, None), ('Cat', 6, 6)]
```

```
>>> df = pd.DataFrame(data, columns=['name', 'start', 'end'])
```

```
>>> df
```

	name	start	end
0	Jo	1.0	5.0
1	Sam	NaN	NaN
2	Avi	NaN	3.0
3	Noor	4.0	NaN
4	Cat	6.0	6.0

```
>>> df.dropna(subset=['start', 'end'])
```

	name	start	end
0	Jo	1.0	5.0
1	Sam	NaN	NaN
2	Avi	NaN	3.0
3	Noor	4.0	NaN
4	Cat	6.0	6.0

Delete rows with missing values



```
>>> data [('Jo', 1, 5), ('Sam', None, None), ('Avi', None, 3),
          ('Noor', 4, None), ('Cat', 6, 6)]
>>> df = pd.DataFrame(data, columns=['name', 'start', 'end'])
>>> df
   name  start  end
0    Jo    1.0  5.0
1   Sam   NaN  NaN
2   Avi   NaN  3.0
3  Noor    4.0  NaN
4   Cat    6.0  6.0
>>> df.dropna(subset=['start', 'end'])
   name  start  end
0    Jo    1.0  5.0

4   Cat    6.0  6.0
```

Delete rows with missing values



```
>>> data [('Jo', 1, 5), ('Sam', None, None), ('Avi', None, 3),
          ('Noor', 4, None), ('Cat', 6, 6)]
>>> df = pd.DataFrame(data, columns=['name', 'start', 'end'])
>>> df
   name  start  end
0    Jo    1.0  5.0
1   Sam   NaN  NaN
2   Avi   NaN  3.0
3  Noor    4.0  NaN
4   Cat    6.0  6.0
>>> df.dropna(subset=['start', 'end'])
   name  start  end
0    Jo    1.0  5.0
4   Cat    6.0  6.0
```

Misc



Pandas has a rich syntax and some very useful functionality

We've covered a lot of useful techniques but Pandas has much more to offer.

Good luck exploring!

Pandas – powerful and flexible



Thanks Wes McKinney!

