PHYS/SPCE 345

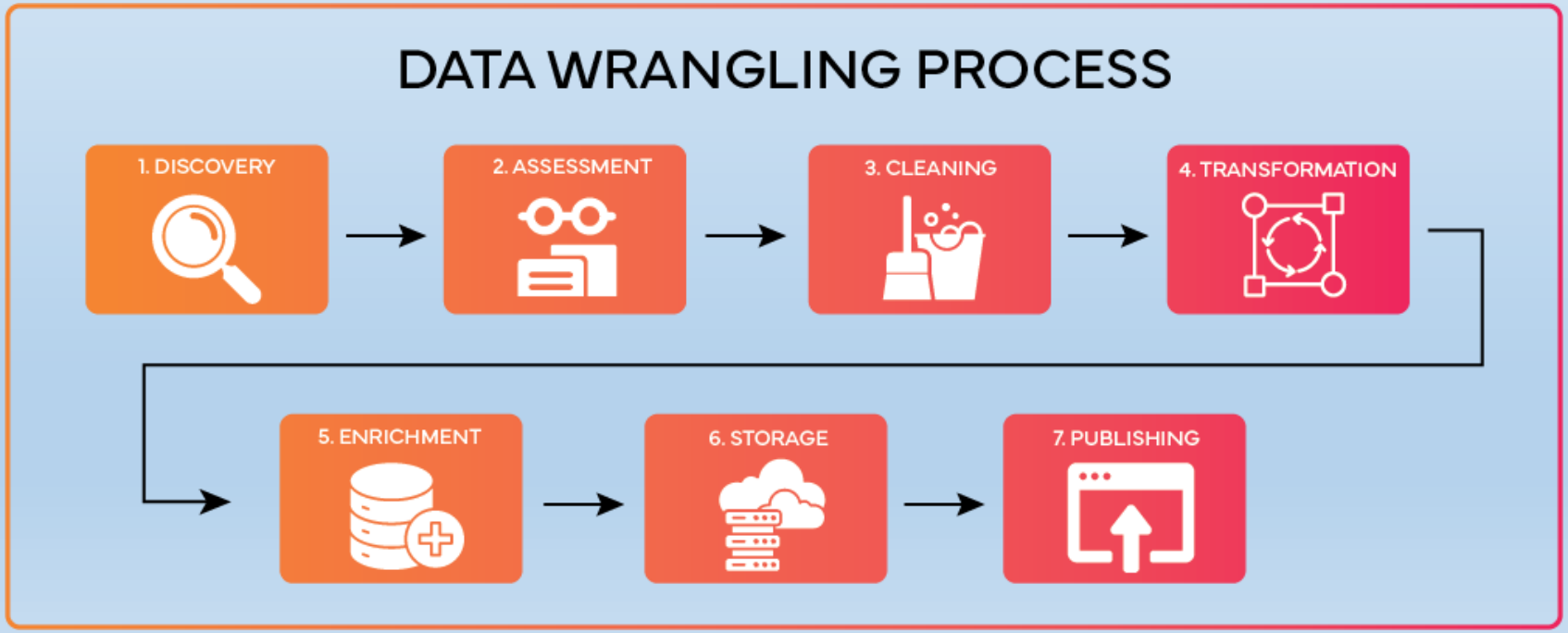
Data Wrangling in Python

Steve Curran, Laby 501
Stephen.Curran@vuw.ac.nz

Data Wrangling

In this section, we will learn to 'wrangle' raw data. Why do we need to do this?

- **To transform the data into a format that is more usable and easier to analyse**
- **To clean and remove errors, e.g. entries which are just plain wrong and/or can cause a crash**
- **To allow the combination with other data**
- **To visualize the data**



The Tools

- **import** pandas **as** pd

  For manipulation of the data. Much more flexible than dealing with arrays directly (and output resembles Excel), although arrays are much faster with very large datasets

  https://www.w3schools.com/python/pandas/

- **import** numpy **as** np

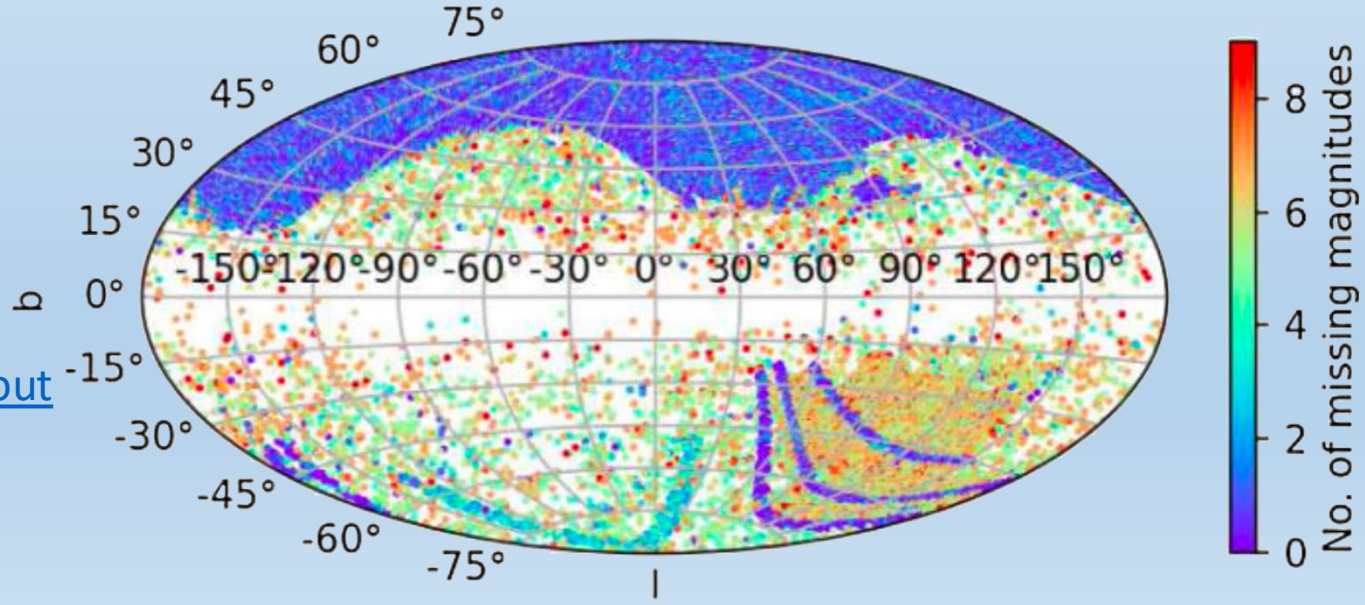
  The alternative if working with arrays. Also *numerical python* contains a lot of useful stats functions (e.g. mean, max, min) https://numpy.org/doc/stable/user/absolute_beginners.html

- **import** matplotlib.pyplot **as** plt

  For plotting and visualisation

  https://matplotlib.org

  https://matplotlib.org/cheatsheets/handout-beginner.pdf



Pandas

First thing we have to do is read the data into a dataframe (df, but name it whatever you want):

- **file.csv -** df = pd.read_csv('file.csv')

  The most favoured format, as the commas allow empty fields (which can play havoc otherwise)

- **file.dat -** df = pd.read_csv('file.dat', delim_whitespace=True) **or** sep='\s+',

  For space/tab separated ascii/txt files

- **file.xlsx -** df1 = pd.read_excel('file.xlsx', sheet_name = 0)

  Have to do each Excel sheet separately – you can loop this if a large number. I just export the ones I need to CSVs, as I find read_excel **a bit temperamental**

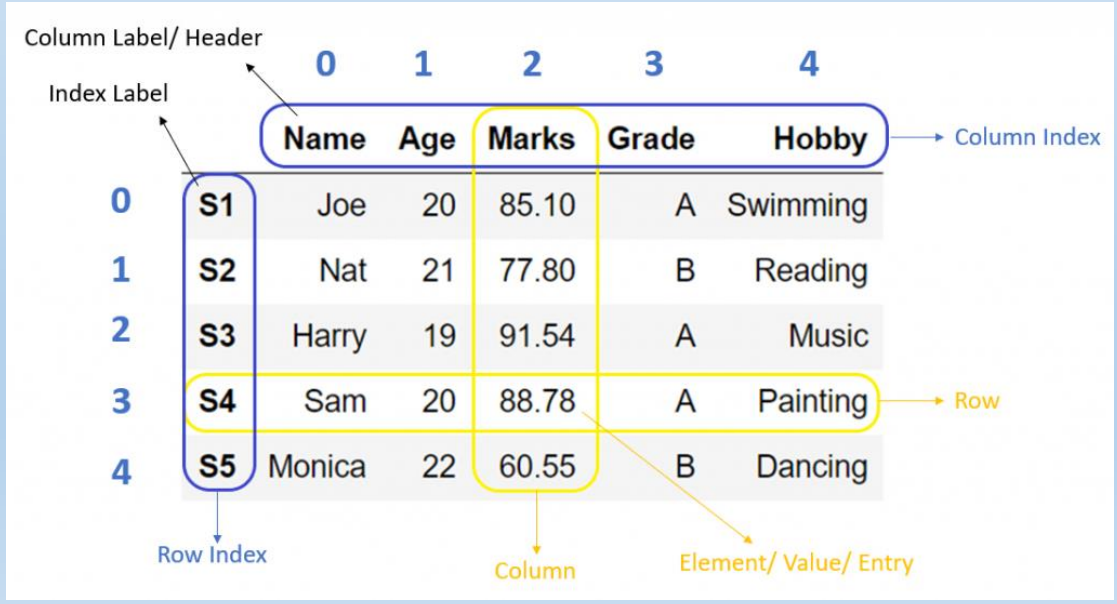
There's also a couple I've never tried:

- **pd.read_table() - reads general delimited files with any delimiter**
- **pd.read_fwf() - reads fixed width files. You can specify fields either by their widths or their position**

Useful options

infile = ‘file.csv’

- pd.read_csv(infile,sep=’|’) **- if the separator is a pipe, or some other character (can also be spaces, e.g.** sep = ‘ ’, sep='\s+’ **)**
- pd.read_csv(infile, usecols=['column_name1', 'column_name2’]) – **if your infile has lots of columns which are not of interest, you can select the ones you want by name**
- pd.read_csv(infile', usecols=[0, 2, 4]) - **as above but selecting columns by the index number**
- pd.read_csv(infile, comment=‘#’) – **if you want to ignore certain lines in your input file you can add a symbol (e.g. #) at the start of the line to specify this**



Useful options

- pd.read_csv(infile,header=None) – **by default the first line of the data are treated as the header, so use this if the header is missing**

  The header can be added when reading in the data or later on, e.g.

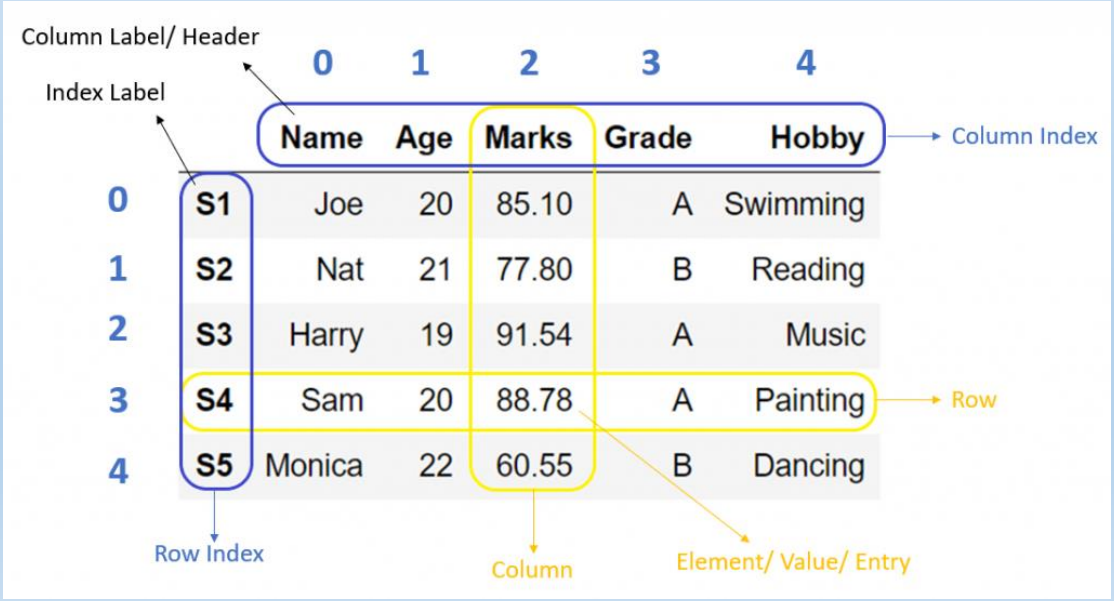
  pd.read_csv(infile header=None, names=['col1', 'col2', 'col3'])

  df.columns = ['col1', 'col2', 'col3']

- pd.read_csv(infile,skiprows=5) – **when you have rows (e.g. 5 rows in this case) at the top of the file you don't want confused for the header**

Note the options can be used in conjunction, e.g.

pd.read_csv(infile,header=None,comment='#')

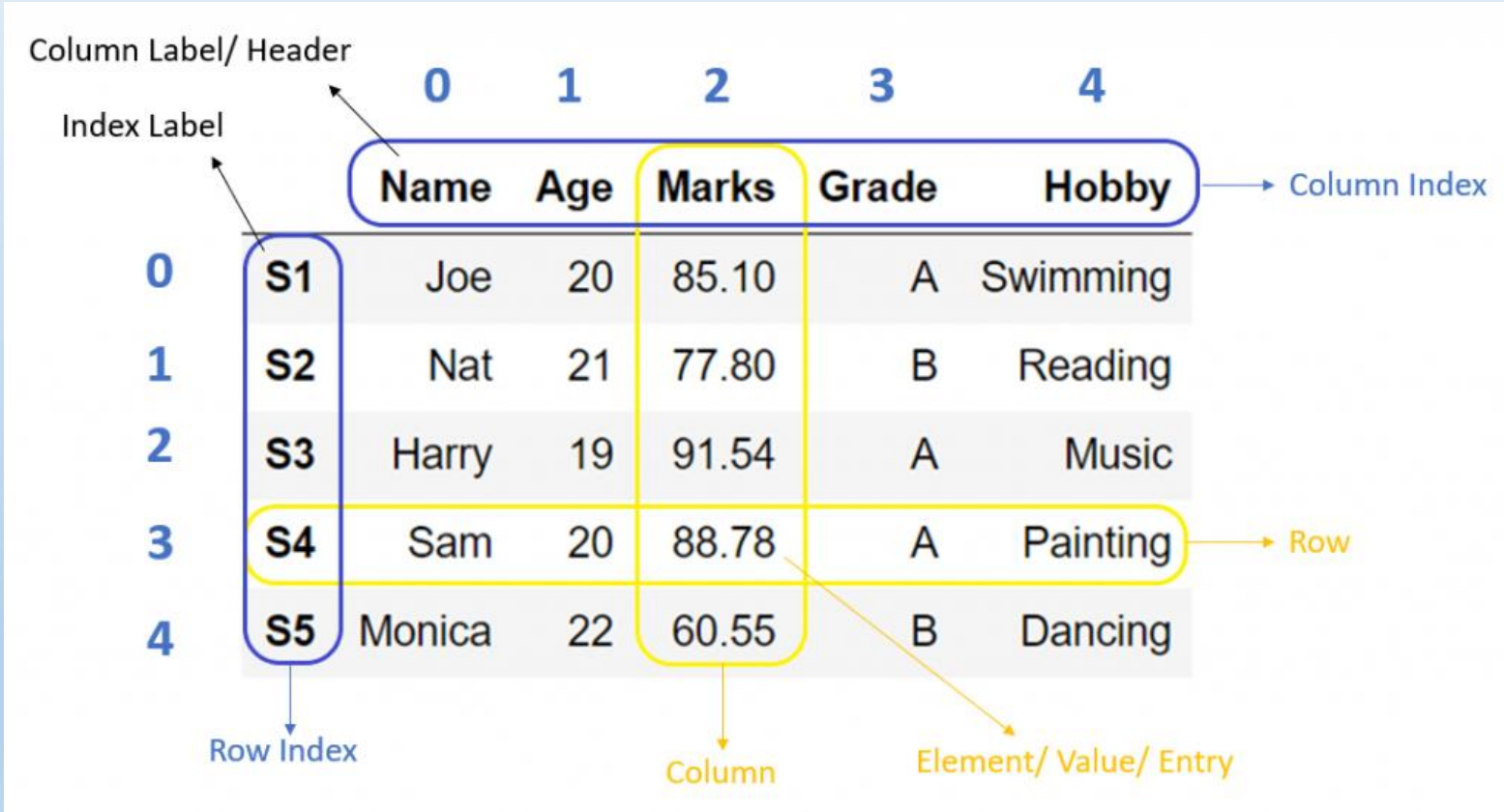


The dataframe

This presents the data in a clear format and is open to serious manipulation, e.g. performing maths on the entries, vectorised operations, transposing and pivoting the data, filtering, matching and grouping (cf. SQL)…

In a good dataframe:

- **Each variable must have its own column**
- **Each observation must have its own row**
- **Each value must have its own cell**



Have a look at https://pandas.pydata.org/docs/getting_started/intro_tutorials

Example, remember this from PHYS/SPCE245?

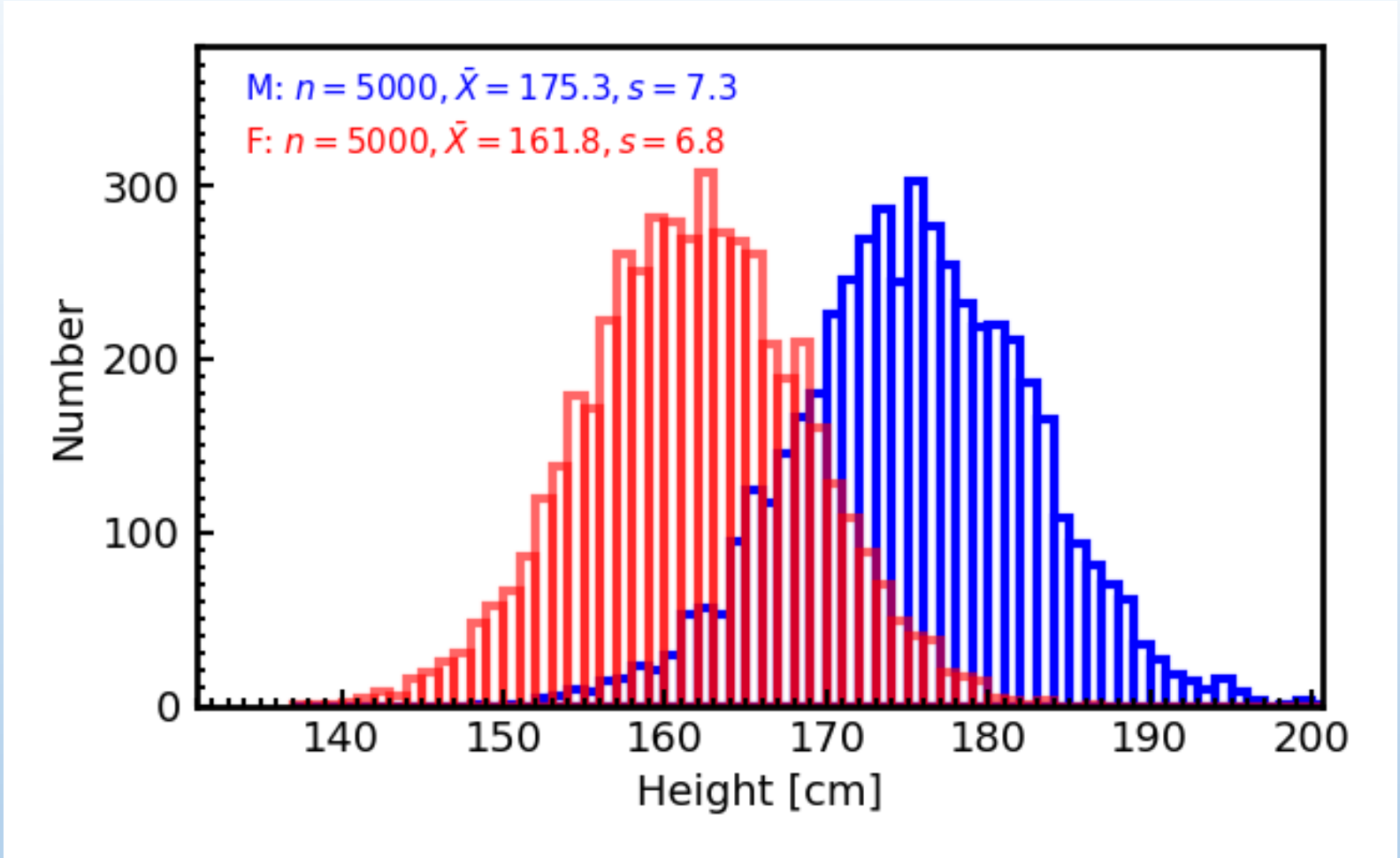
Let's look at height.csv

Ex1.ipynb

```
h = pd.read_csv('height.csv');h
```

	Person ID	Gender	Height
0	8872	Female	61.864667
1	7456	Female	62.819752
2	4841	Male	70.695001
3	17	Male	63.974326
4	7334	Female	66.138172
...	...	...	...
9995	398	Male	67.266363
9996	785	Male	69.297292
9997	9487	Female	63.628666
9998	8875	Female	66.837536
9999	7202	Female	63.993617

10000 rows × 3 columns



Default is to show the first and last 5 lines of data, but this can easily be changed

First – inspect the data

```
h.describe()
```

	Person ID	Height
count	10000.00000	10000.000000
mean	4999.50000	66.367560
std	2886.89568	3.847528
min	0.00000	54.263133
25%	2499.75000	63.505620
50%	4999.50000	66.318070
75%	7499.25000	69.174262
max	9999.00000	78.998742

Describes the numerical values in the data – note that, although numerical, Person ID **is a categorical value (like your student number)**

Number of entries (none appear to be missing here)

Mean value

Standard deviation

Minimum value

1st quartile

2nd quartile (the median)

3rd quartile

Maximum value

Data good – nothing missing, no zero nor negative heights and no obvious outliers (mean ≈ median).

However, look the units! Remember from 245 this was American data, so not in sensible units. Probably inches (can't be feet!). Let's see – 2.54 cm in an inch

```
h['Height'] = 2.54*h['Height']
```

```
h.describe()
```

	Person ID	Height
count	10000.00000	10000.000000
mean	4999.50000	168.573602
std	2886.89568	9.772721
min	0.00000	137.828359
25%	2499.75000	161.304276
50%	4999.50000	168.447898
75%	7499.25000	175.702625
max	9999.00000	200.656806

Note we have a mix of genders, but this may not always be obvious (e.g. if the first and last 5 rows all male)

```
h = pd.read_csv('height.csv');h
```

	Person ID	Gender	Height
0	8872	Female	61.864667
1	7456	Female	62.819752
2	4841	Male	70.695001
3	17	Male	63.974326
4	7334	Female	66.138172
...	...	...	...
9995	398	Male	67.266363
9996	785	Male	69.297292
9997	9487	Female	63.628666
9998	8875	Female	66.837536
9999	7202	Female	63.993617

10000 rows × 3 columns

```
h['Gender'].unique()
```

```
array(['Female', 'Male'], dtype=object)
```

Just the two (data is American and old)

How do we filter this? By selecting a column of the dataframe we select a series, not a single value, so

```
m = h['Gender'] == "Male"; m
```

```
0       False
1       False
2        True
3        True
4       False
```

To filter use the syntax

```
m = h[h['Gender'] == "Male"];
```

	Person ID	Gender	Height
2	4841	Male	179.565303
3	17	Male	162.494787
7	1004	Male	173.326413
9	1127	Male	179.237802
10	11	Male	181.967645

And we can, of course, examine the new dataframes

```
m.describe()
```

	Person ID	Height
count	5000.000000	5000.000000
mean	2499.500000	175.326919
std	1443.520003	7.272940
min	0.000000	148.353539
25%	1249.750000	170.623685
50%	2499.500000	175.330380
75%	3749.250000	180.311409
max	4999.000000	200.656806

We can also work with more than one dataframe at a time and combine these, say after w = pd.read_csv('weight.csv')

```
h_w = pd.merge(h, w, on = ['Person ID'])
del h_w['Gender_y']
h_w = h_w.rename(columns={'Gender_x': 'Gender'})
h_w['Weight'] = h_w['Weight']/2.20462
h_w
```

	Person ID	Gender	Height	Weight
0	8872	Female	157.136254	56.247422
1	7456	Female	159.562170	56.899228
2	4841	Male	179.565303	87.101535
3	17	Male	162.494787	78.418716
4	7334	Female	167.990958	68.983826

Check what you have makes sense!

h_w = pd.merge(h, w, on = ['Person ID'])
h_w['Gender_y']
h_w = h_w.rename(columns={'Gender_x': 'Gender'})
h_w['Weight'] = h_w['Weight']/2.20462

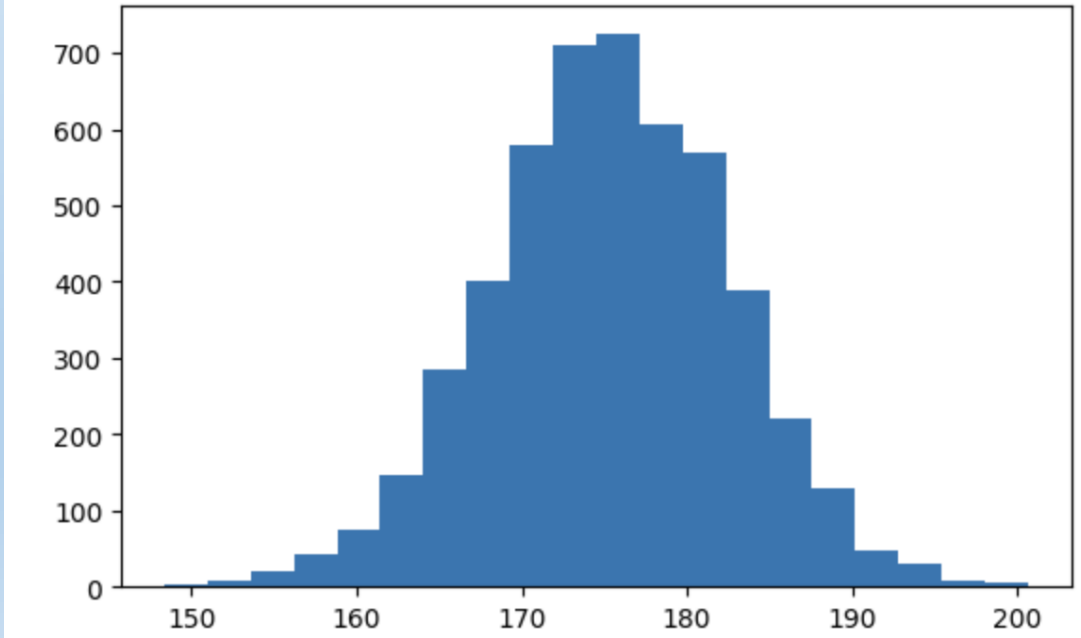
will merge the data, but because each dataframe has a column with the same name, we can remove one of these and rename the remaining
this is not necessary, but is tidier
you'll note the weights appear high, but, again, these are in imperial units (so convert from lb to kg)

Matplotlib

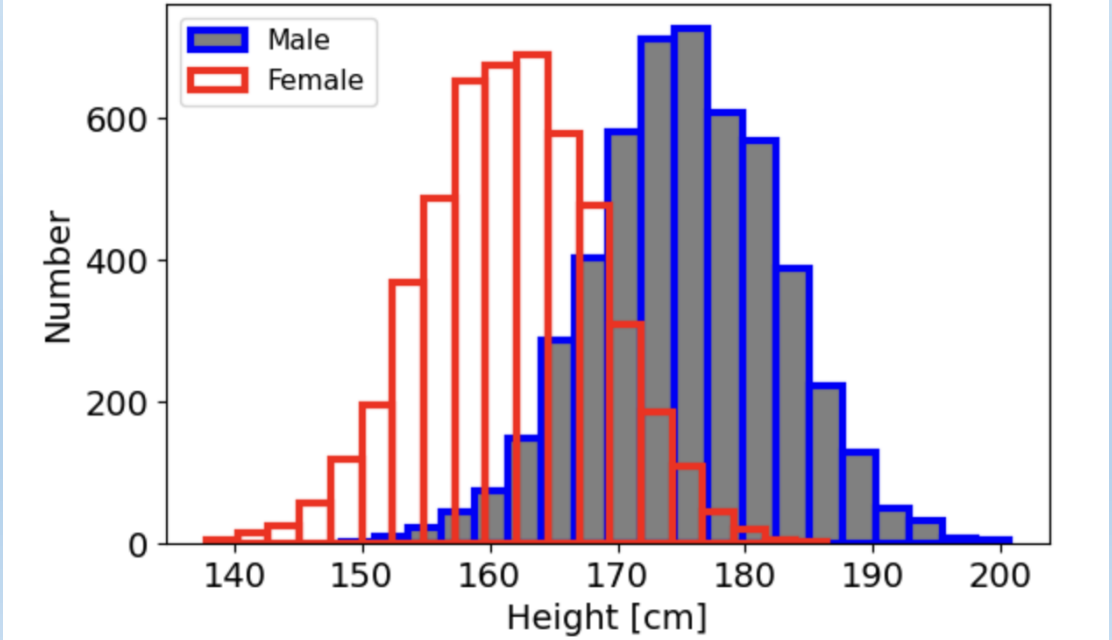
For plotting the data – by no means are you limited to this package (I've heard of pgplot being used and this is way older than most you. I'm guilty of using this up to 2023, but with C).

As you might have seen in 245, the default settings give pretty basic plots, but these can easily be snazzed up

```
plt.figure(figsize=(6.5,4))
plt.hist(m["Height"], bins=20)
plt.show()
```



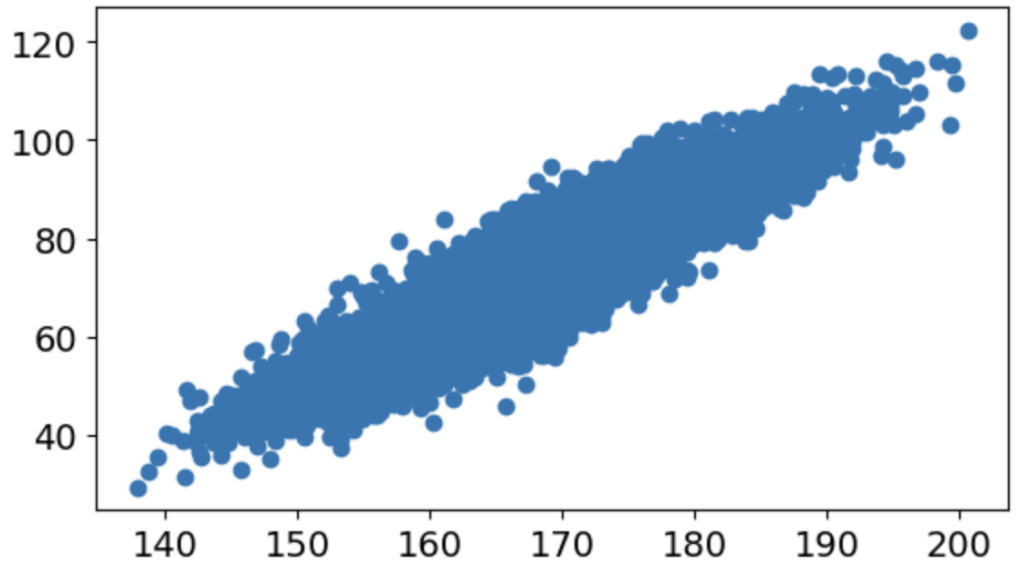

```
font = 14
plt.rcParams.update({'font.size': font})

plt.figure(figsize=(6.5,4))
plt.hist(m["Height"],bins=20,facecolor="grey", edgecolor='b',
         linewidth=3, label="Male")
plt.hist(f["Height"],bins=20,facecolor='none', edgecolor='r',
         linewidth=3, label="Female")
plt.xlabel('Height [cm]')
plt.ylabel('Number')
plt.legend(loc = 'upper left', fontsize = 0.8*font)
plt.show()
```



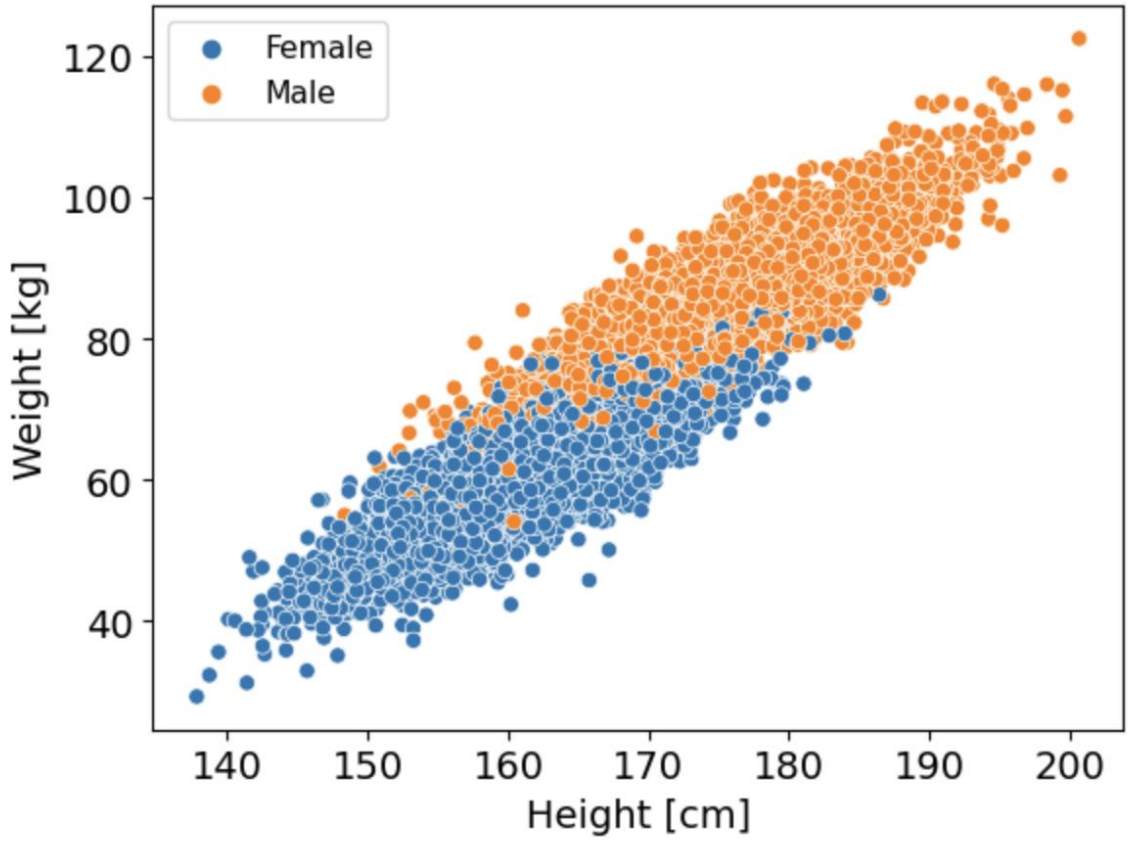
With the data combined we can plot the different parameters against each other

Have a play and try different things

```
plt.figure(figsize=(6.5,3.6))
plt.scatter(x = h_w["Height"], y = h_w["Weight"])
plt.show()
```




```
import seaborn as sns
sns.scatterplot(x = h_w["Height"], y = h_w["Weight"], hue = h_w['Gender'])
plt.xlabel('Height [cm]')
plt.ylabel('Weight [kg]')
plt.legend(loc = 'upper left', fontsize = 0.8*font)
plt.show()
```



Note that seaborn will plot the labels and legend without being asked, but the former are based on the column names (so don't have the units), plus I don't like the legend at full size

Data Cleaning

Doing in *Ex2.ipynb*

Load a new dataset on the number of fires in the Amazon rainforest

df = pd.read_csv(url, encoding = "ISO-8859-1")

	ano	mes	estado	numero	encontro
0	1998	Janeiro	Acre	0 Fires	1/1/1998
1	1999	Janeiro	Acre	0 Fires	1/1/1999
2	2000	Janeiro	Acre	0 Fires	1/1/2000
3	2001	Janeiro	Acre	0 Fires	1/1/2001
4	2002	Janeiro	Acre	0 Fires	1/1/2002
...	...	...	...	...	...
6449	2012	Dezembro	Tocantins	128	1/1/2012
6450	2013	Dezembro	Tocantins	85	1/1/2013
6451	2014	Dezembro	Tocantins	223	1/1/2014
6452	2015	Dezembro	Tocantins	373	1/1/2015
6453	2016	Dezembro	Tocantins	119	1/1/2016

6454 rows × 5 columns

ISO/IEC 8859-1 encodes what it refers to as "Latin alphabet no. 1", consisting of 191 characters from the Latin script. This character-encoding scheme is used throughout the Americas, Western Europe, Oceania, and much of Africa.

Looks like names are in Spanish or Portuguese, which we could of course work with but let's change the column names to see how this is done. The syntax is

df = df.rename(columns={'old1': 'new1', 'old2': 'new2'})

but we can do this directly by using a *dictionary*

A dictionary is a built-in data structure used to store collections of data in key-value pairs. It is mutable, meaning its contents can be changed after creation.

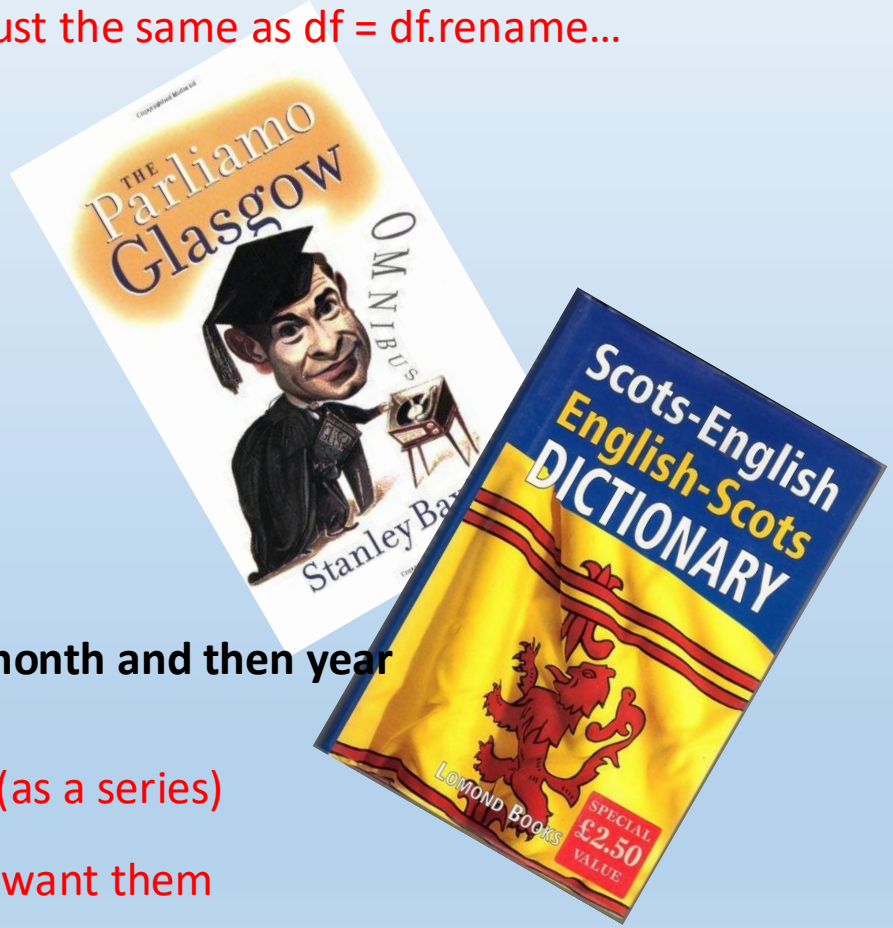
Renaming Columns

```
new_columns = {'ano' : 'year','estado': 'state','mes': 'month',
               'numero': 'number_of_fires','encontro': 'date'}
df.rename(columns = new_columns, inplace=True)
df
```

	year	month	state	number_of_fires	date
0	1998	Janeiro	Acre	0 Fires	1/1/1998
1	1999	Janeiro	Acre	0 Fires	1/1/1999
2	2000	Janeiro	Acre	0 Fires	1/1/2000
3	2001	Janeiro	Acre	0 Fires	1/1/2001
4	2002	Janeiro	Acre	0 Fires	1/1/2002

dictionary – {key:value}

note the inplace = True, which is just the same as df = df.rename…



Next, we'll rearrange the columns so the date is in the first field, followed by month and then year

df.columns

df = df[['date','month','year','state', 'number_of_fires']]

list the columns (as a series)

in the order you want them

```
df.describe()
```

	year
count	6454.000000
mean	2007.461729
std	5.746654
min	1998.000000
25%	2002.000000
50%	2007.000000
75%	2012.000000
max	2017.000000

number_of_fires **is not included under** describe **– so it's not numerical.**

The fact that some of these include Fires **means this parameter is treated as a string.**

	date	month	year	state	number_of_fires
0	1/1/1998	Janeiro	1998	Acre	0 Fires
1	1/1/1999	Janeiro	1999	Acre	0 Fires
2	1/1/2000	Janeiro	2000	Acre	0 Fires
3	1/1/2001	Janeiro	2001	Acre	0 Fires
4	1/1/2002	Janeiro	2002	Acre	0 Fires
...	...	...	...	...	...
6449	1/1/2012	Dezembro	2012	Tocantins	128
6450	1/1/2013	Dezembro	2013	Tocantins	85
6451	1/1/2014	Dezembro	2014	Tocantins	223
6452	1/1/2015	Dezembro	2015	Tocantins	373
6453	1/1/2016	Dezembro	2016	Tocantins	119

6454 rows × 5 columns

To remove unnecessary text from use str.strip

```
df['number_of_fires'] = df['number_of_fires'].str.strip(" Fires")
```

But we also have to tell pandas that this is not a string anymore

```
df['number_of_fires'] = df['number_of_fires'].astype(int)
```

but crashes – still have some non-numeric values in column?

Missing Data

```
df.isnull().sum()
```

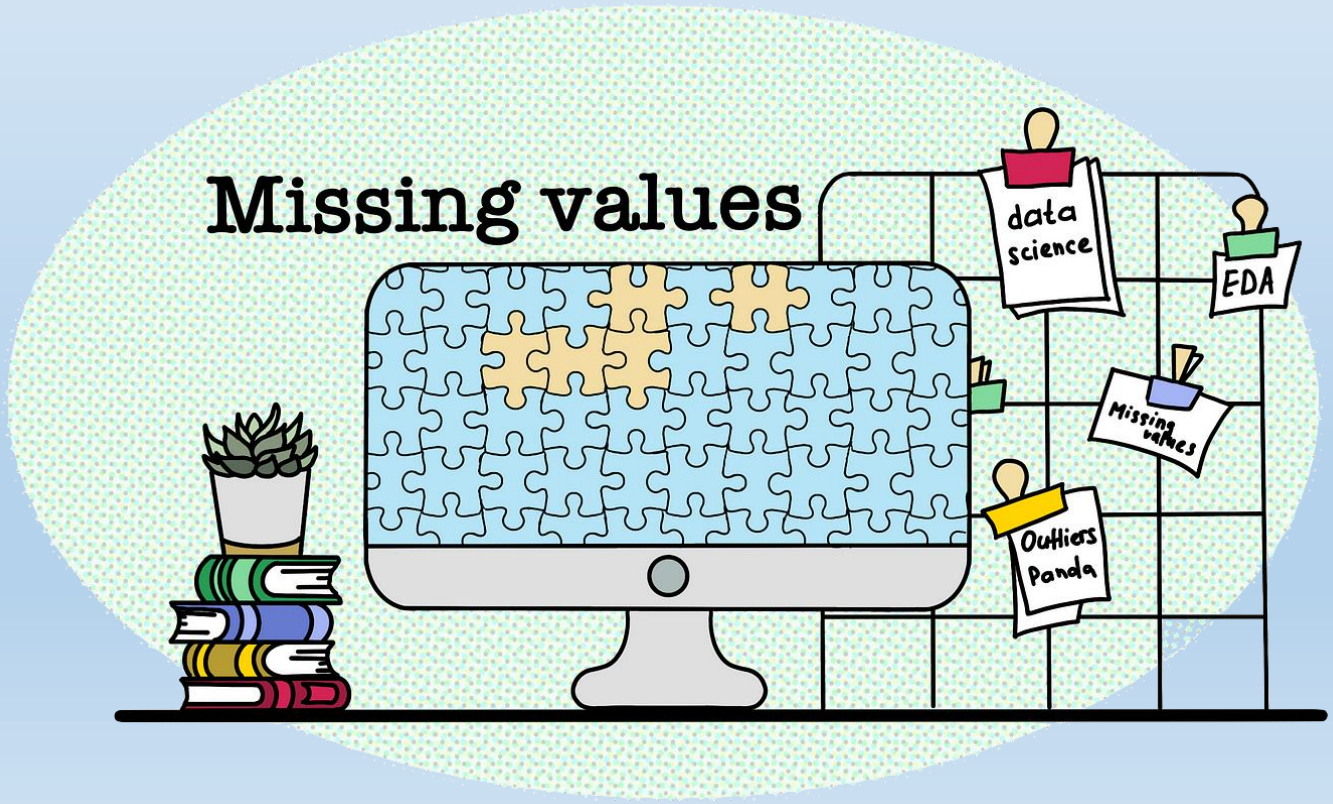
```
date                0
month               0
year                0
state               0
number_of_fires     132
dtype: int64
```

We can remove Null or NaN (not a number) values by dropping rows with NaN values

```
df = df.dropna()
```

```
df = df.reset_index()
```

resets row indices in case any rows were dropped



Note that the count has dropped from 6454

```
df.describe()
```

	year	number_of_fires
count	6322.000000	6322.000000
mean	2007.462670	110.667972
std	5.747903	192.248217
min	1998.000000	0.000000
25%	2002.000000	3.000000
50%	2007.000000	26.000000
75%	2012.000000	117.000000
max	2017.000000	998.000000

What do we do with missing data?

- **Remove the rows, as above**
- **Replace them (impute) with some arbitrary number (e.g. 0 or the average)**

```
df= df.fillna(0)
df = df.fillna(df.mean())
```

- **Forward Fill (**ffill**) or Back Fill (**bfill**).**

  ffill propagates the last observed non-null value forward until…

  bfill propagates the first observed non-null value backward until...

  another non-null value is met.

- **Impute the missing data using machine learning to estimate what these should be**

But bear in mind that the values are probably wrong. These are handy if the other columns contain useful data and you don't want to discard the whole row – each of these methods will leave us with all 6454 entries

Modifying the contents of the dataframe

As well as mathematical operations and converting strings to numbers, we can, for example…

Convert month names from Portuguese to English

```
: import warnings
warnings.filterwarnings("ignore")
warnings.simplefilter(action='ignore', category=FutureWarning)

months = {'Janeiro': 'January','Fevereiro': 'February','MarÃČÂğo': 'March',
          'Abril': 'April','Maio': 'May','Junho': 'June','Julho': 'July',
          'Agosto':'August','Setembro': 'September','Outubro': 'October',
          'Novembro': 'November','Dezembro':'December'}
months
df["month"] = df["month"].map(months)
df
```

:

	date	month	year	state	number_of_fires
0	1/1/1998	January	1998	Acre	0
1	1/1/1999	January	1999	Acre	0
2	1/1/2000	January	2000	Acre	0
3	1/1/2001	January	2001	Acre	0
4	1/1/2002	January	2002	Acre	0

If you don't want the annoying, and not very helpful, warnings

Defining a dictionary

Changing the values in the column by mapping the dictionary

This can be very useful for other tasks, such as conditional statements – instead of

if something == the_other:

Grouping Data

```
number = df.groupby('state').count()
number
```

state	date	month	year	number_of_fires
Acre	234	214	234	234
Amapa	231	212	231	231
Amazonas	234	216	234	234
Bahia	233	213	233	233

```
mon = df.groupby('month').agg({'number_of_fires': ['sum']})
mon
```

month	number_of_fires sum
April	00100121000321197301042006206121185317182516 63...
August	130631363967281.9766454.1988399604451941.68242...
December	70001172000104136863212230954371163033345545 50...
February	00001030050200000251020164153373114287432263 34...

Using groupby **– note that this is counting the number of entries (rows for each state), not the number of times a fire has been recorded, which we can do with..**

Other aggregation functions are min ,max , mean, std

Summing the number of fires per month highlights a problem – the number_of_fires **column is considered to comprise strings (a hangover from** 0 Fires**) and so pandas is summing them accordingly.**

```
#CHANGE TO INTEGER
#df['number_of_fires'] = df['number_of_fires'].astype(int)
# CRASHES
df['number_of_fires'] = df['number_of_fires'].astype(float)
mon = df.groupby('month').agg({'number_of_fires': ['sum','min',
                                                   'max','mean','std']})
mon
```

However, this crashes when using astype(int)

- I found that the number_of_fires **is not an integer, but a float.**

	number_of_fires				
	sum	min	max	mean	std
month					
April	28550.770	0.0	947.0	54.073428	108.719126
August	91619.442	0.0	995.0	174.846263	289.002947
December	57753.459	0.0	956.0	114.137271	162.087125
February	30920.050	0.0	871.0	57.794486	101.503046
January	47043.844	0.0	960.0	88.428278	131.316791
July	89931.720	0.0	989.0	170.972852	250.162517

```
total = df.groupby('state')['number_of_fires'].sum()
total
```

```
state
Acre                17971.030
Amapa               20108.576
Amazonas            29890.129
Bahia               43411.951
Ceara               30395.042
Distrito Federal     3501.000
Espirito Santo      37002.276
```

Note also that the months are in alphabetical order, which we'll change next

Sort month name

```
order = ['January', 'February', 'March', 'April', 'May',
         'June', 'July', 'August', 'September', 'October',
         'November', 'December']
mons= mon.reindex(order, axis=0)
mons
```

	number_of_fires				
	sum	min	max	mean	std
month					
January	47043.844	0.0	960.0	88.428278	131.316791
February	30920.050	0.0	871.0	57.794486	101.503046
March	NaN	NaN	NaN	NaN	NaN
April	28550.770	0.0	947.0	54.073428	108.719126
May	34154.363	0.0	942.0	64.564013	134.416394
June	55301.675	0.0	979.0	103.755488	175.849554
July	89931.720	0.0	989.0	170.972852	250.162517
August	91619.442	0.0	995.0	174.846263	289.002947
September	60071.119	0.0	998.0	113.771059	201.861751
October	85841.257	0.0	964.0	162.886636	242.427983
November	87063.812	0.0	995.0	165.520555	218.601142
December	57753.459	0.0	956.0	114.137271	162.087125

This wasn't obvious in the unsorted version

Which state has the most fires on average?

```
ave = df.groupby('state').mean()
ave.sort_values(by=['number_of_fires'],
                    ascending=False)
```

	year	number_of_fires
state		
Sao Paulo	2007.440678	211.390669
Mato Grosso	2007.508475	202.472405
Bahia	2007.463519	186.317386
Piau	2007.493562	161.127258
Minas Gerais	2007.467532	160.183013

See what this does

```
df.groupby(['state','month']).agg({'number_of_fires':
                    ['sum','min', 'max','mean','std']})
```

Look at the distributions

Minimum Working Barplot

```
import matplotlib.pyplot as plt
#plt.figure(figsize=(6.5,4))
#plt.bar(mons['month'],mons['mean'])
#plt.show()
mons.columns
```

```
MultiIndex([('number_of_fires',  'sum'),
            ('number_of_fires',  'min'),
            ('number_of_fires',  'max'),
            ('number_of_fires', 'mean'),
            ('number_of_fires',  'std')],
           )
```

~~number_of_fires~~				
sum	**min**	**max**	**mean**	**std**

Bar plot crashes because of the MultiIndex **format generated by** groupby **means it can't find** month **or** mean

```
mons2 = mons.droplevel(level=0,axis=1)
mons2.insert(0, 'month', mons.index)
mons.index.names = ['index']
mons2
```

index	month	sum	min	max	mean	std
January	January	47043.844	0.0	960.0	88.428278	131.316791
February	February	30920.050	0.0	871.0	57.794486	101.503046
March	March	NaN	NaN	NaN	NaN	NaN
April	April	28550.770	0.0	947.0	54.073428	108.719126

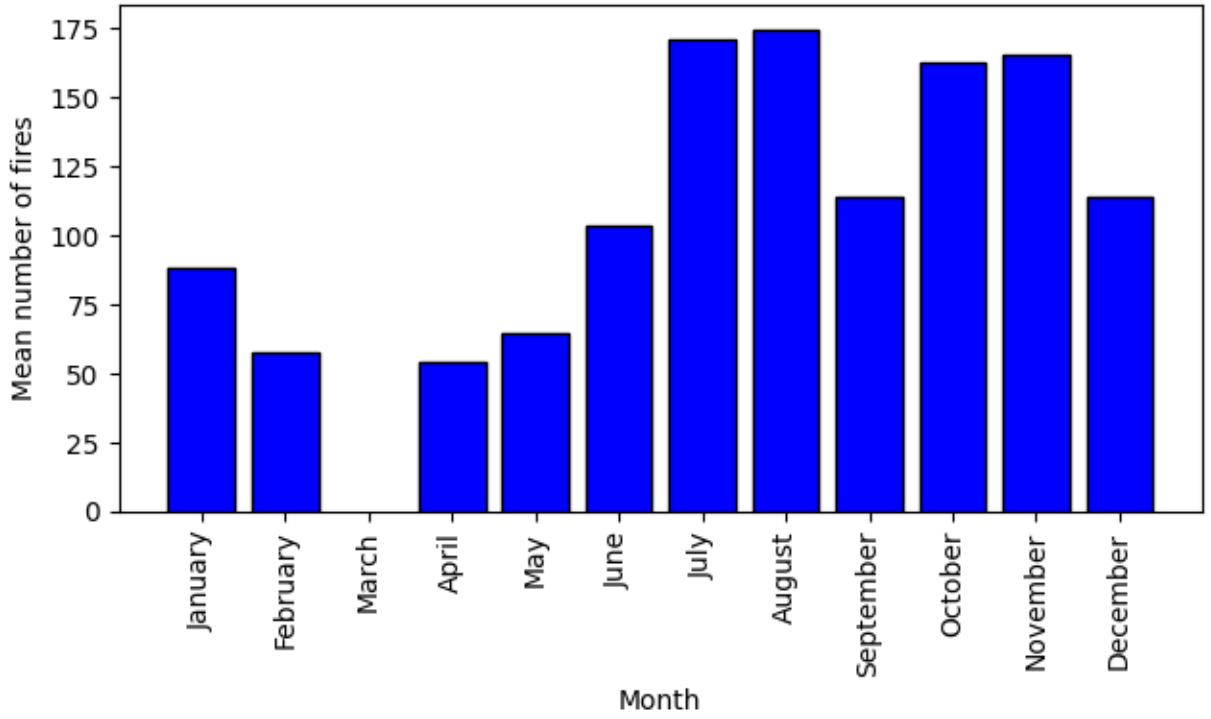
mons2 = mons.droplevel(level=0,axis=1) **drops the first level heading (**axis = 1, **horizontally along the columns)**

mons2.insert(0, 'month', mons.index) **uses the index column from the DF as the first column '**month'

mons.index.names = ['index'] **renames the index – although not recognised, it'll still complain**

Works now but let's do as function so it can be re-used

```
def barplot(data,xpara,ypara,xlabel,ylabel,rot):
    plt.figure(figsize=(6.5,4))
    plt.bar(data[xpara],data[ypara], facecolor = 'b', edgecolor ='k')
    plt.xticks(rotation=rot)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.tight_layout(pad=0.1)
    plt.savefig("Fires-%s_%s.png" %(xpara,ypara)) # HARD COPY
    plt.show()
barplot(mons2,'month','mean','Month','Mean number of fires',90)
```



E.g. for total number of fires binned by month

barplot(mons2,'month','sum','Month','Total number of fires',90)

for total number of fires binned by state, year, etc

barplot(state,'month','sum','Month','Total number of fires',90)

But will have to fix the columns as on previous slide – could put the whole thing in a function

Time Series

Convert to datetime object

```
df['Date'] = pd.to_datetime(df['date'], format='%d/%m/%Y')
df # BIG Y FOR FULL YEAR, y IF E.G. 1/2/12
```

	date	month	year	state	number_of_fires	Date
0	1/1/1998	January	1998	Acre	0.0	1998-01-01
1	1/1/1999	January	1999	Acre	0.0	1999-01-01
2	1/1/2000	January	2000	Acre	0.0	2000-01-01
3	1/1/2001	January	2001	Acre	0.0	2001-01-01
4	1/1/2002	January	2002	Acre	0.0	2002-01-01
...	...	...	...	...	...	...
6449	1/1/2012	December	2012	Tocantins	128.0	2012-01-01
6450	1/1/2013	December	2013	Tocantins	85.0	2013-01-01
6451	1/1/2014	December	2014	Tocantins	223.0	2014-01-01
6452	1/1/2015	December	2015	Tocantins	373.0	2015-01-01
6453	1/1/2016	December	2016	Tocantins	119.0	2016-01-01

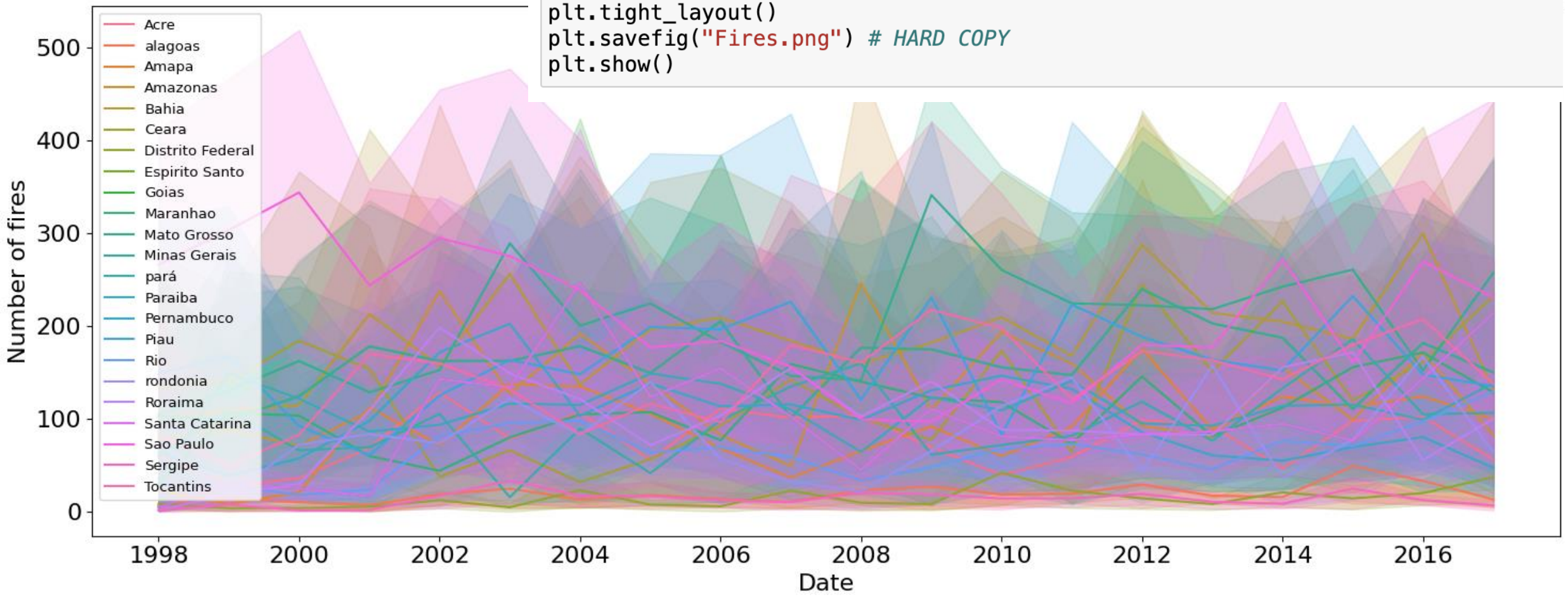
6322 rows × 6 columns

Also in the datetime **module.**

- **date: date (year, month, and day).**
- **time: time (hour, minute, second, microsecond).**
- **datetime: a specific moment in time, combining date and time.**
- **timedelta: the difference between two dates, time or datetime instances.**
- **timezone: time zone with a fixed offset from UTC.**

A bit busy, but you get the idea

```
import seaborn as sns
font = 16
plt.rcParams.update({'font.size': font})
plt.figure(figsize=(15,6))
sns.lineplot(x = df["Date"], y = df["number_of_fires"], hue = df['state'])
plt.xlabel('Date')
plt.ylabel('Number of fires')
plt.legend(loc = 'upper left', fontsize = 0.6*font)
plt.tight_layout()
plt.savefig("Fires.png") # HARD COPY
plt.show()
```



But, from summing all of the data it appears that the date **column doesn't match the** month

```
# Less dates than expected, check
df1['date'].unique()
len(df1['date'].unique()) # YEP, ONLY 20 - 1 JAN, SO WHERE'S THE OTHER?
```

```
20
```

```
#CHECK - YEP date COLUMN IS UNRELIABLE
df1[df1['month'] == 'May']
```

	date	month	year	state	number_of_fires
80	1/1/1998	May	1998	Acre	0.0
81	1/1/1999	May	1999	Acre	0.0
82	1/1/2000	May	2000	Acre	1.0
83	1/1/2001	May	2001	Acre	0.0
84	1/1/2002	May	2002	Acre	0.0
...	...	...	...	...	...
6310	1/1/2013	May	2013	Tocantins	373.0
6311	1/1/2014	May	2014	Tocantins	618.0
6312	1/1/2015	May	2015	Tocantins	512.0
6313	1/1/2016	May	2016	Tocantins	778.0
6314	1/1/2017	May	2017	Tocantins	576.0

529 rows × 5 columns

According to date **all of the months under the** date **column are January!**

I'm beginning to suspect that this dataset was set up to test us

That's not right - opportunity to show more cleaning (dates are always 'fun')

Fix Dates

```
df2 = df1.copy() # TO SAVE GOING BACK TO DEFINING df1
month_map = {'January': 1, 'February': 2, 'March': 3, 'April': 4,
    'May': 5, 'June': 6, 'July': 7, 'August': 8, 'September': 9, 'October': 10,
              'November': 11, 'December': 12}

df2['month_no'] = df2['month'].map(month_map); #df2
df2.dropna(inplace = True) # IN CASE ALL THE NaN FOR MARCH CAUSE PROBLEMS
df2['month_no'] = df2['month_no'].astype(int);#df2
df2['month_name'] = df2['month'] # to_datetime NEEDS SPECIFIC FORMAT
df2['month'] = df2['month_no'];#df2
df2['Date'] = pd.to_datetime(df2[['year','month']].assign(day=1))
del df2['date']
df2
```

Mapping month names to month number

Became a float, convert to int

Renaming columns as datetime is fussy

Make the first date day 1

	month	year	state	number_of_fires	month_no	month_name	Date
0	1	1998	Acre	0.000	1	January	1998-01-01
1	1	1999	Acre	0.000	1	January	1999-01-01
2	1	2000	Acre	0.000	1	January	2000-01-01
3	1	2001	Acre	0.000	1	January	2001-01-01

```
df3 = df2.groupby(['Date']).agg({'number_of_fires':
                                 ['sum','mean']})
#len(df3)
```

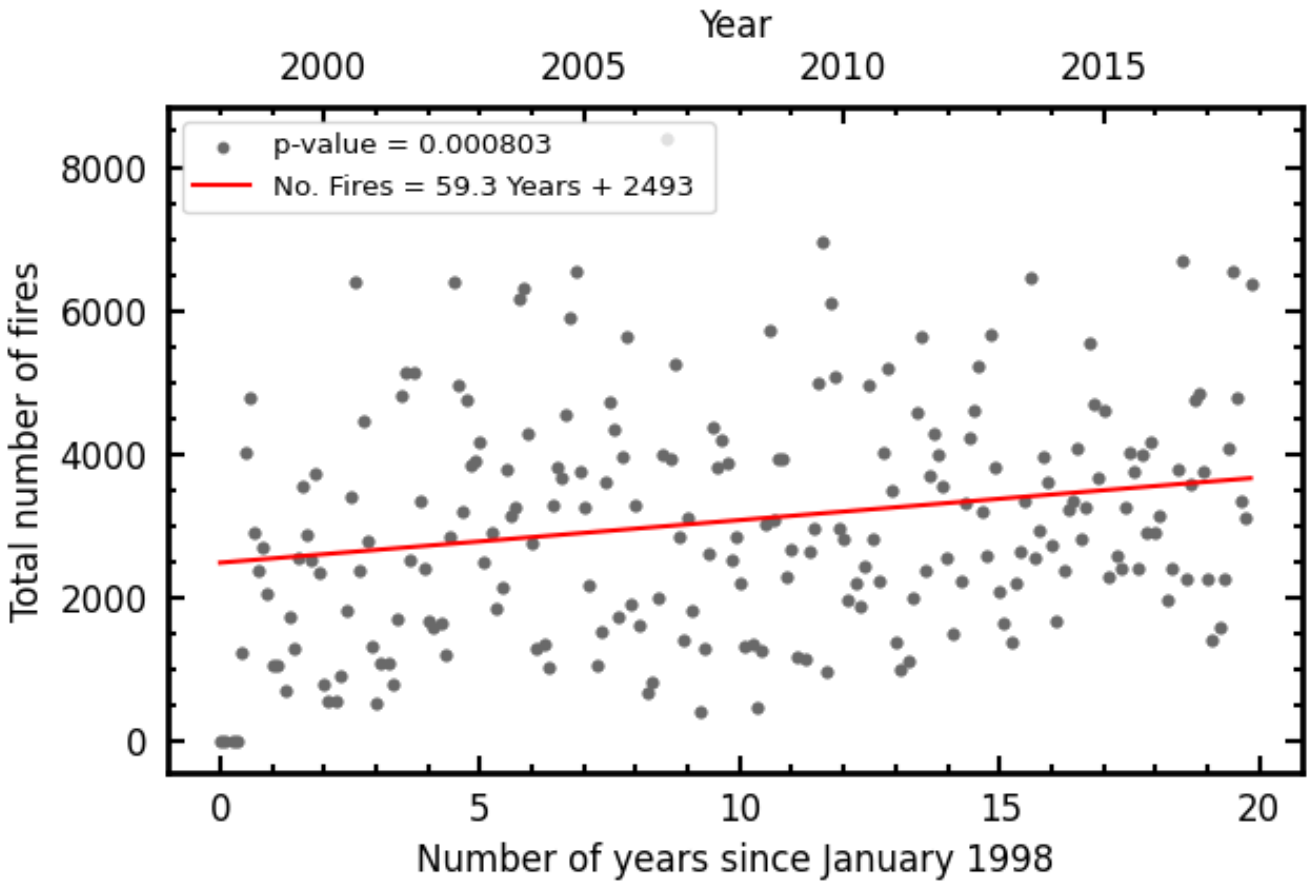
```
# THAT'S MORE LIKE IT, BUT AFTER groupby HAVE TO DEAL WITH COLUMN NAMES
#df3.head()
df4 = df3.droplevel(level=0,axis=1)
df4.insert(0, 'Date', df3.index)
df4.index.names = ['index']
start = pd.to_datetime('1998-01-01') # THis WILL BE USEFUL FOR THE PLOT
df4['Days'] = (df4['Date'] - start).dt.days
df4['Years'] = df4['Days']/365.25
df4.head()
```

index	Date	sum	mean	Days	Years
1998-01-01	1998-01-01	0.000	0.000000	0	0.000000
1998-02-01	1998-02-01	0.000	0.000000	31	0.084873
1998-04-01	1998-04-01	0.000	0.000000	90	0.246407
1998-05-01	1998-05-01	0.000	0.000000	120	0.328542
1998-06-01	1998-06-01	1246.201	47.930808	151	0.413415

Fix the mess left by groupby

dt **giving time difference**

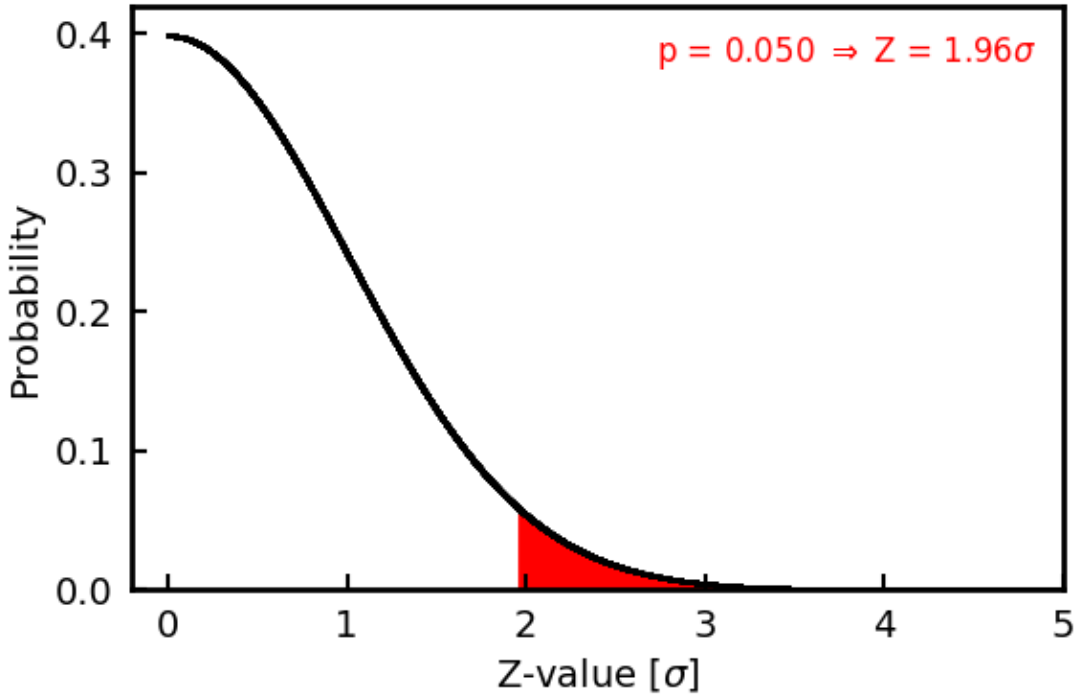
dt **doesn't work with years (or months)**



An increase of 59 fires per year, on average (starting from 2493 in 1998)

This is *statistical evidence*, not *opinion*, that fires are becoming more frequent.

What could be causing that?



The distribution has a probability of $p = 8.0 \times 10^{-4}$ of arising by chance ($Z = 3.35$) – remember for the 'softer sciences' $p < 0.05$ ($Z > 1.96$) is considered significant!