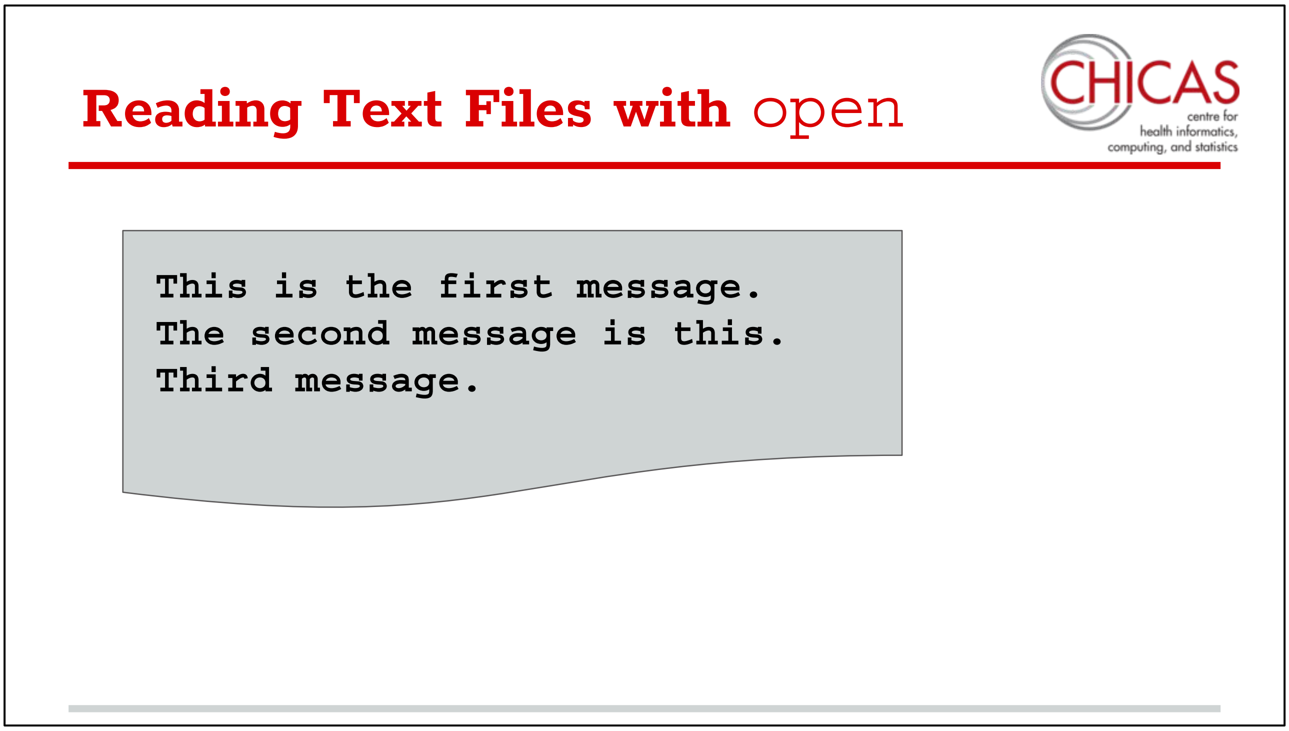
CHICAS centre for health informatics, computing, and statistics

Python for Data Data for Python

Barry Rowlingson

Lancaster University Medical School

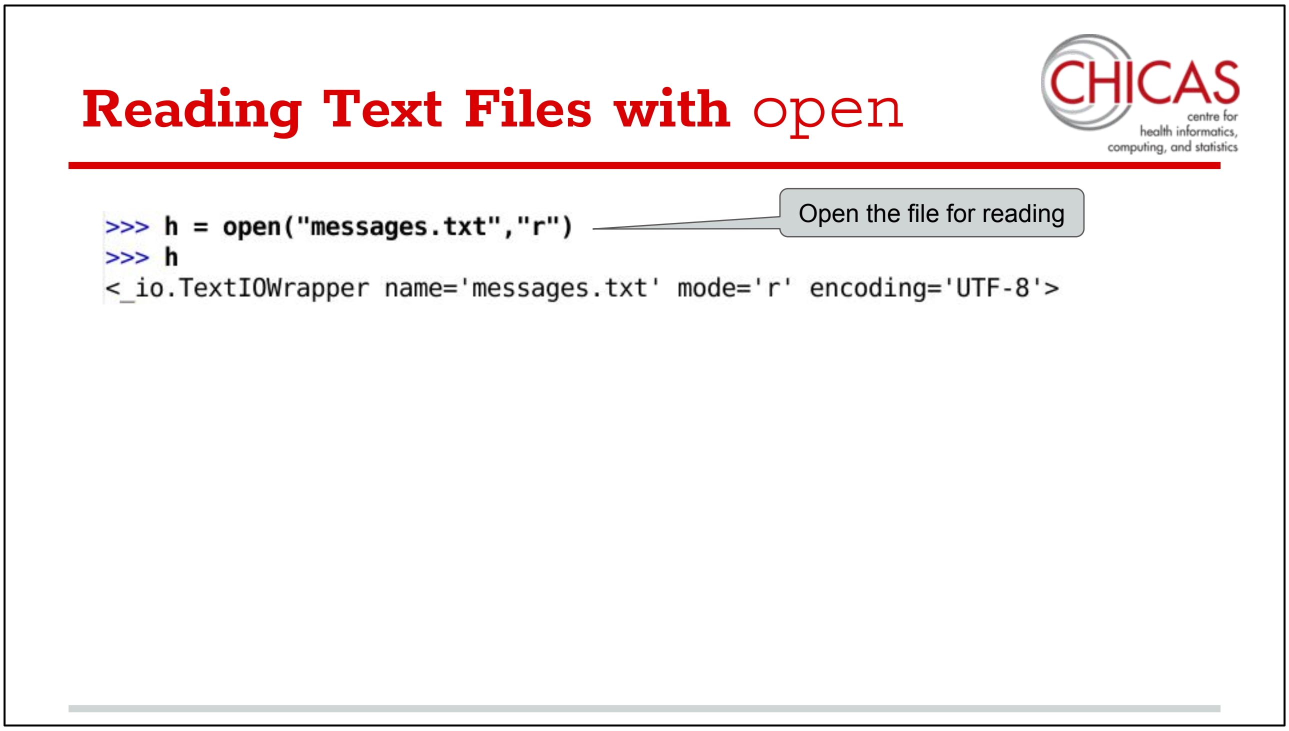
This is all about reading and writing data in Python…



Reading Text Files with `open`

```
This is the first message.
The second message is this.
Third message.
```

The simplest reading we ever need to do is to read some lines from a plain text file. Here’s the file.



Reading Text Files with `open`

CHICAS centre for health informatics, computing, and statistics

```
>>> h = open("messages.txt","r")
>>> h
<_io.TextIOWrapper name='messages.txt' mode='r' encoding='UTF-8'>
```

Open the file for reading

Because reading a file is asking the computer for some resources, you can’t dive straight in and get the data. Its good manners to tell the operating system you are interested in this data file, so most modern languages require you to “open” the file for reading before you start with it. If the file doesn’t exist or there’s some other problem, the OS will let Python know and there will be an error at this point.

Reading Text Files with `open`

```
>>> h = open("messages.txt","r")
>>> h
<_io.TextIOWrapper name='messages.txt' mode='r' encoding='UTF-8'>
```

`h` is… one of these.
A ***connection*** or a ***handle***

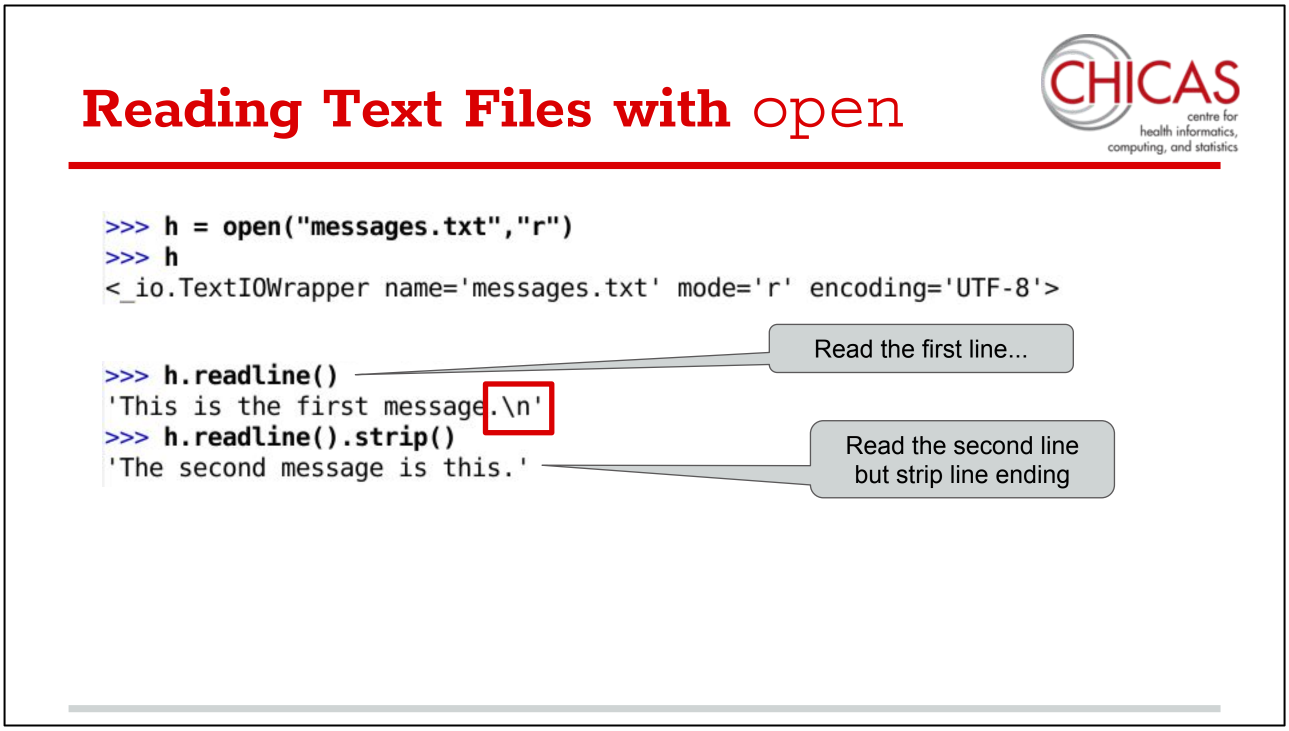
In Python once you've opened the file you get back one of these objects, referred to as a "handle" or a "connection" or something simply a "file" object.

Reading Text Files with `open`

```
>>> h = open("messages.txt","r")
>>> h
<_io.TextIOWrapper name='messages.txt' mode='r' encoding='UTF-8'>

>>> h.readline()
'This is the first message.\n'
```

Read the first line...

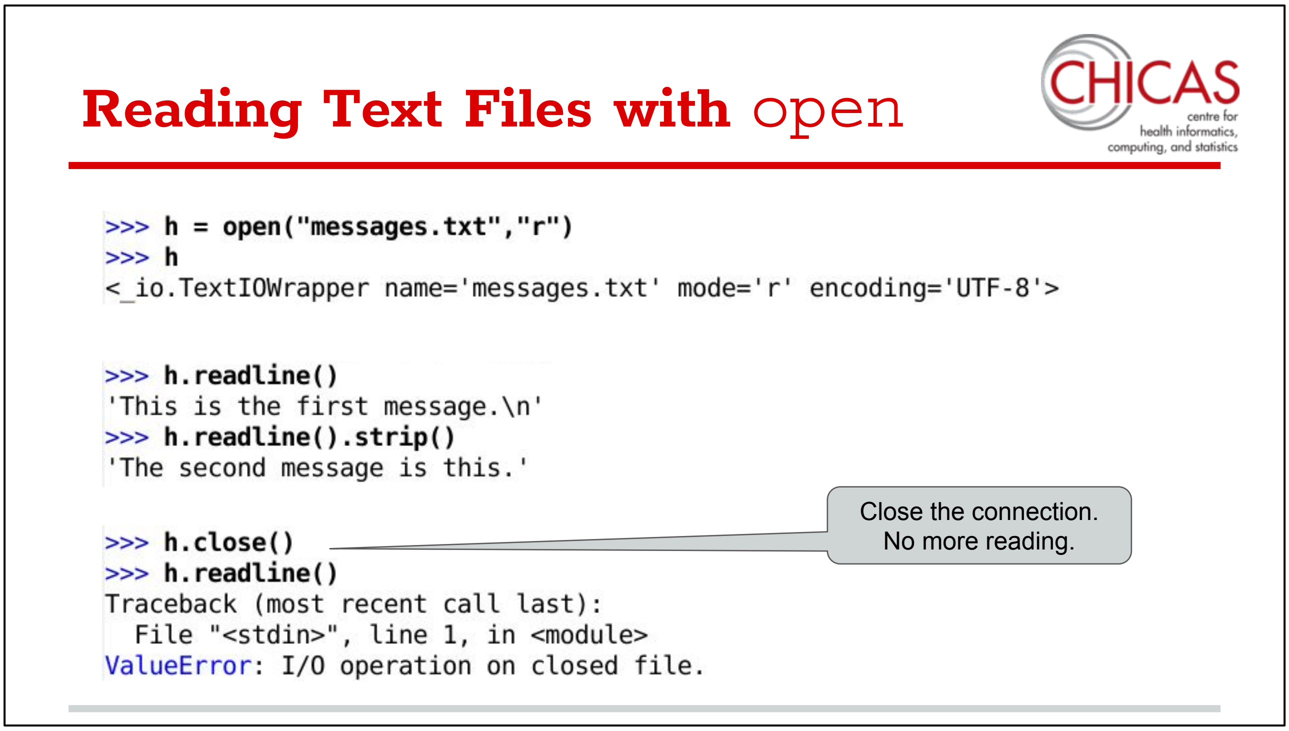
Now you can read a line with the “readline” method, and it is returned.



Reading Text Files with `open`

```
>>> h = open("messages.txt","r")
>>> h
<_io.TextIOWrapper name='messages.txt' mode='r' encoding='UTF-8'>

>>> h.readline()
'This is the first message.\n'
>>> h.readline().strip()
'The second message is this.'
```

Notice the backslash-N on the string? That's the end-of-line marker. If you don't want that (and you usually dont) you can use the "strip" method of text strings to remove it. Notice that when I do a "readline" the second time, its now reading the second line.

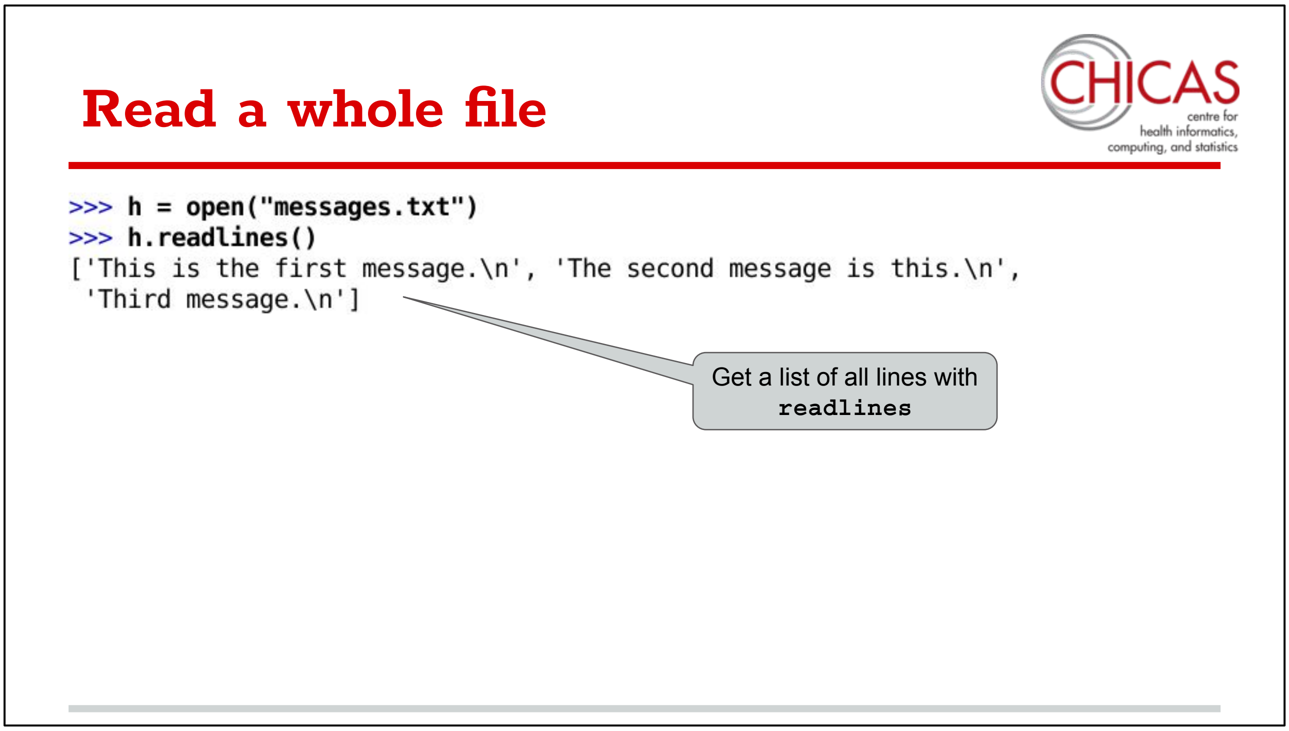


Reading Text Files with `open`

```
>>> h = open("messages.txt","r")
>>> h
<_io.TextIOWrapper name='messages.txt' mode='r' encoding='UTF-8'>


>>> h.readline()
'This is the first message.\n'
>>> h.readline().strip()
'The second message is this.'


>>> h.close()
>>> h.readline()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Close the connection.
No more reading.

Once you've finished reading, its good practice to "close" the connection. Most operating systems have a limit on the number of open files per process. Python will close them automatically when an open goes out of scope, but if you should always remember this is happening. Operations on a closed file handle will fail.

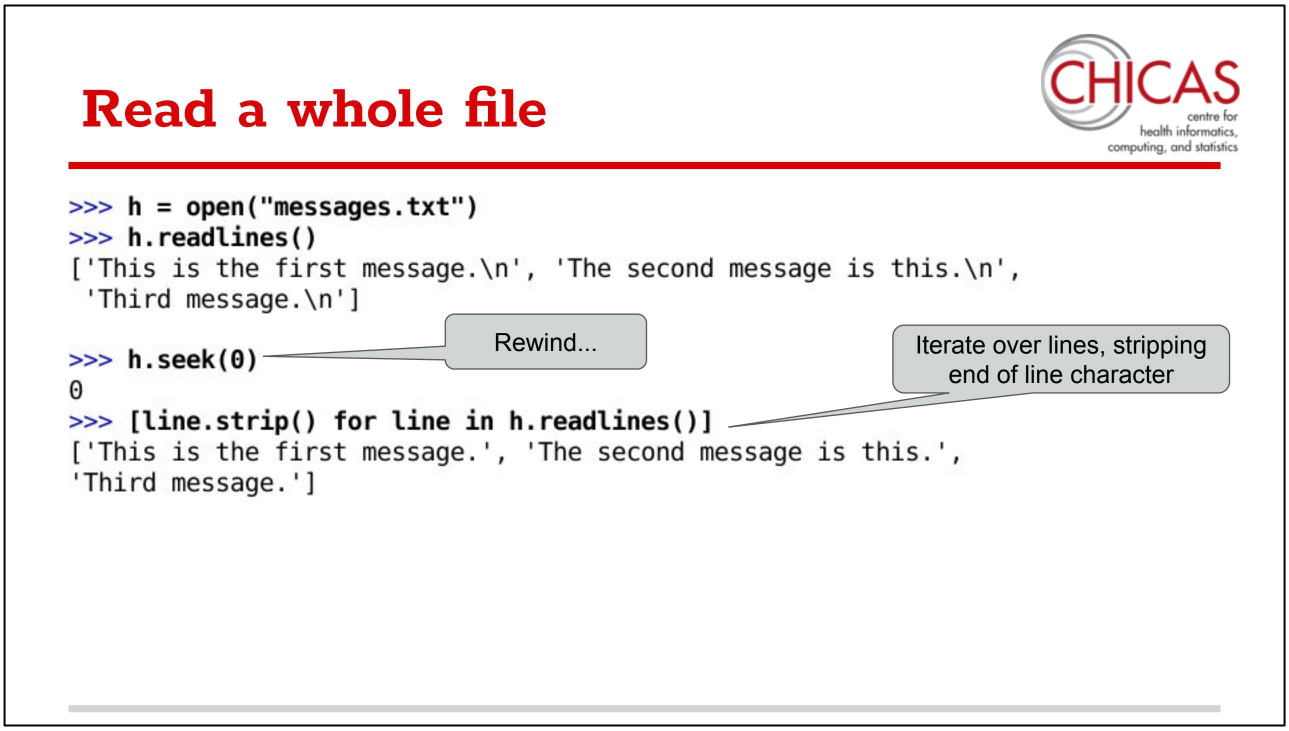


Read a whole file

```
>>> h = open("messages.txt")
>>> h.readlines()
['This is the first message.\n', 'The second message is this.\n',
 'Third message.\n']
```

Get a list of all lines with **`readlines`**

If you want to slurp in every line of a file, then there’s the “readlines” method. You get back a list of all the lines, complete with the line ending characters.



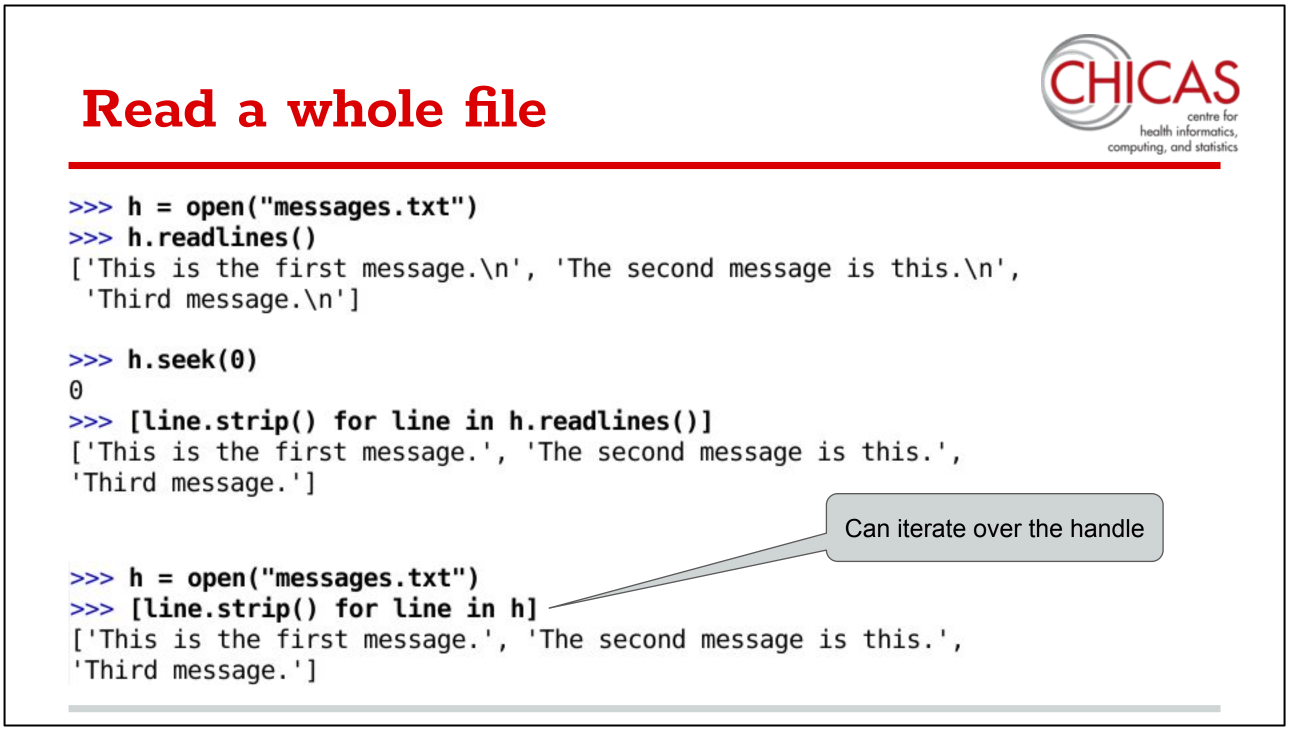
Read a whole file

```
>>> h = open("messages.txt")
>>> h.readlines()
['This is the first message.\n', 'The second message is this.\n',
 'Third message.\n']

>>> h.seek(0)
0
>>> [line.strip() for line in h.readlines()]
['This is the first message.', 'The second message is this.',
'Third message.']
```

Rewind...

Iterate over lines, stripping end of line character

Let’s strip those off. First I use the “seek” method to rewind the file back to the start, and then I use a list comprehension that calls “strip” on every line it reads. Now I get a list of lines without end-of-line characters.


Read a whole file

```
>>> h = open("messages.txt")
>>> h.readlines()
['This is the first message.\n', 'The second message is this.\n',
 'Third message.\n']

>>> h.seek(0)
0
>>> [line.strip() for line in h.readlines()]
['This is the first message.', 'The second message is this.',
'Third message.']


>>> h = open("messages.txt")
>>> [line.strip() for line in h]
['This is the first message.', 'The second message is this.',
'Third message.']
```

Can iterate over the handle

I can do that even quicker by iterating over the handle - Python understands that an iterative operation over a file handle is likely to be iterating over reading lines, so there's a little shortcut that lets you do this.

Or even shorter...

The list function iterates:

```
>>> list(h)
['This is the first message.\n', 'The second message is this.\n', 'Third messag
```

One-liner:

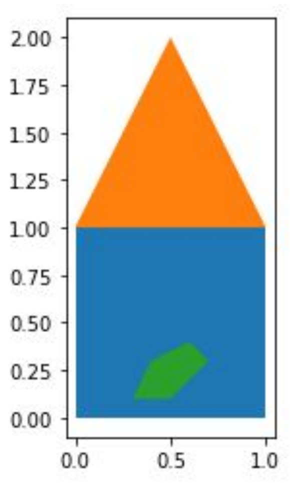
```
>>> L = list(open("messages.txt"))
>>> L
['This is the first message.\n', 'The second message is this.\n', 'Third messag
```

...brevity is not always best practice!

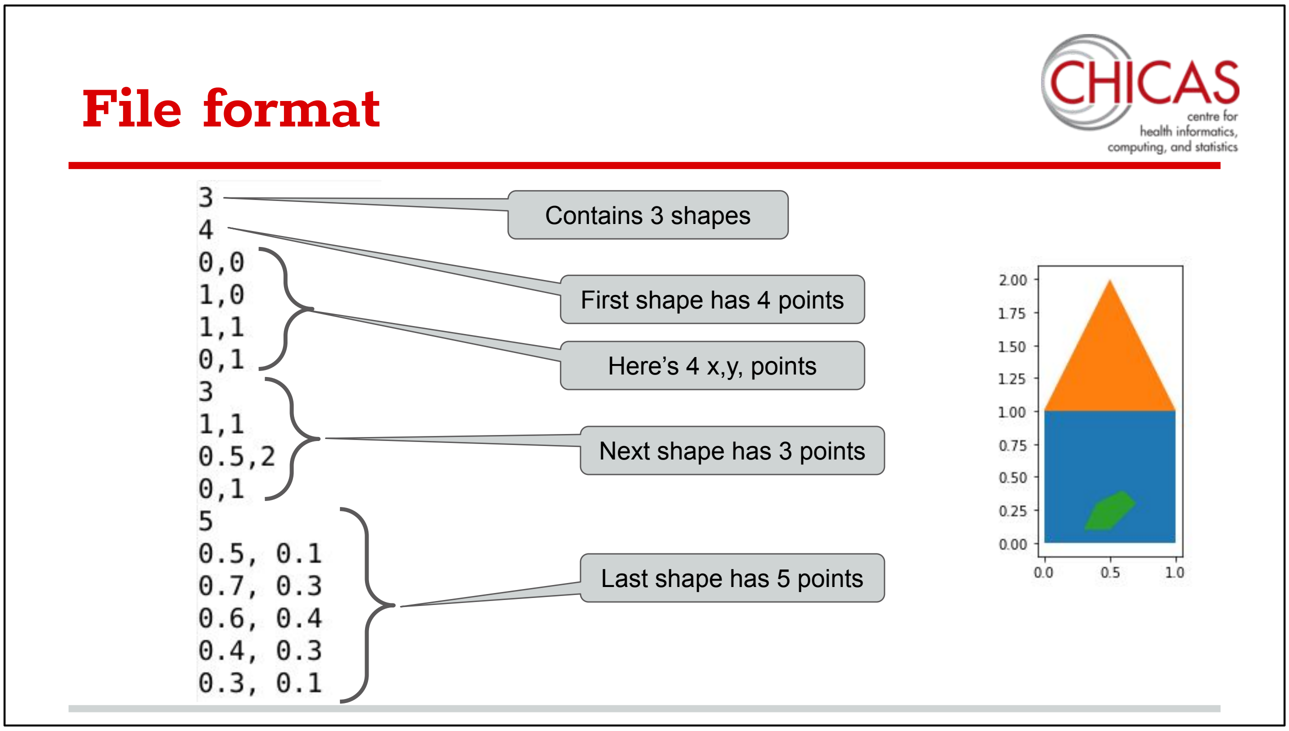
You can make this shorter (without the strip method) by noticing that the “list” function iterates over its argument, and so a simple call to “list” on the file handle will return all the lines. You can then read all the lines with a one-liner, although its not always best practice to cut the number of lines to a minimum.

Why would you read lines?

```
3
4
0,0
1,0
1,1
0,1
3
1,1
0.5,2
0,1
5
0.5, 0.1
0.7, 0.3
0.6, 0.4
0.4, 0.3
0.3, 0.1
```



Why would you want to read a line at a time? Here’s a file format that describes a set of shapes.



File format

```
3
4
0,0
1,0
1,1
0,1
3
1,1
0.5,2
0,1
5
0.5, 0.1
0.7, 0.3
0.6, 0.4
0.4, 0.3
0.3, 0.1
```

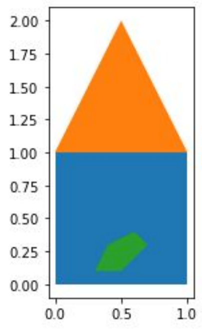
Contains 3 shapes

First shape has 4 points

Here's 4 x,y, points

Next shape has 3 points

Last shape has 5 points

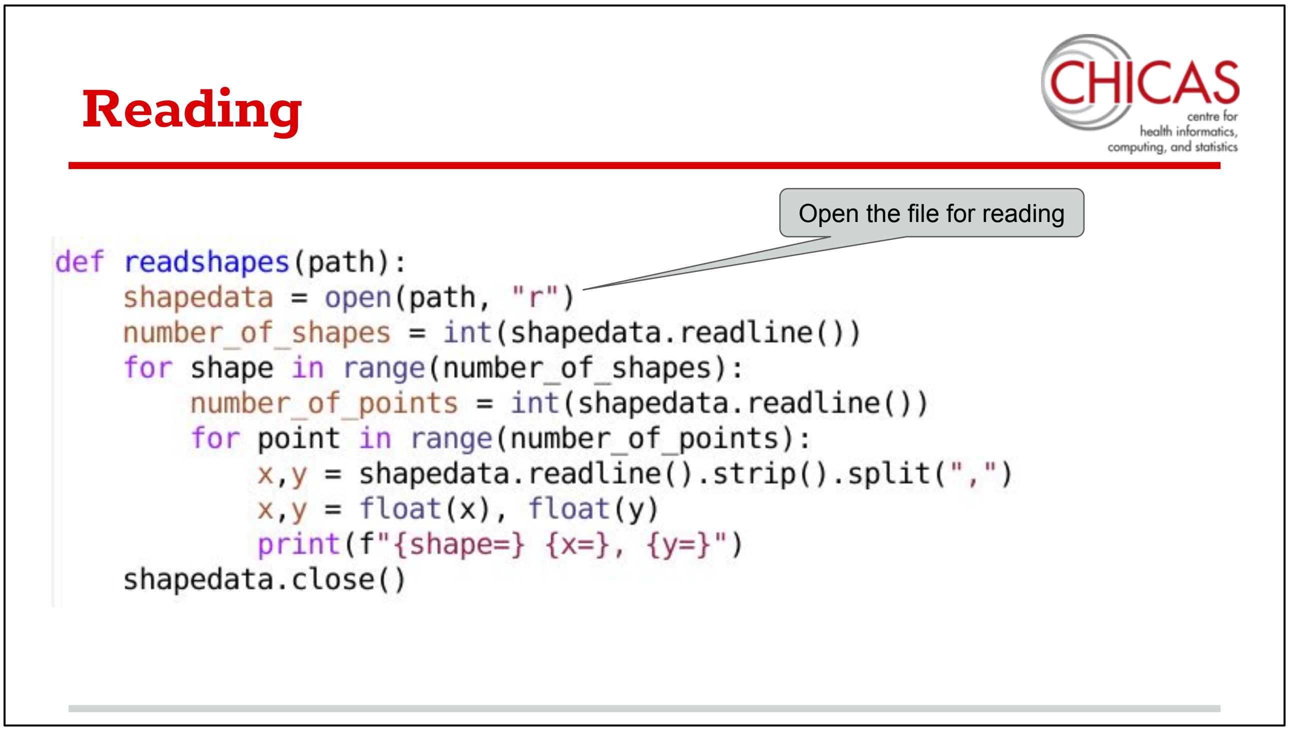


The first line is the number of shapes. Then for each shape there's the number of points in the shape, followed by that number of lines with the xy coordinates of the shape. Then the next shape starts.

Reading

```
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

Here’s a function that reads in and prints out that shape data.

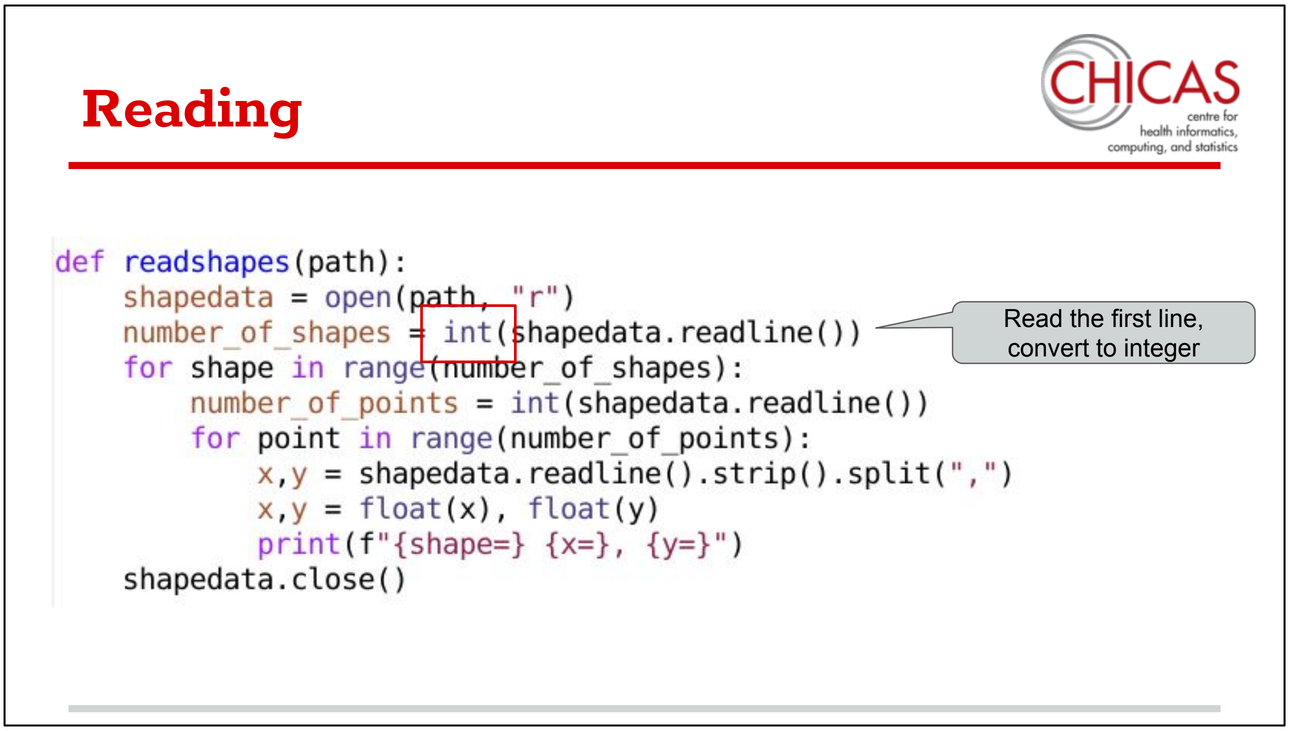


Reading

Open the file for reading

```python
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

First we open the file for reading…



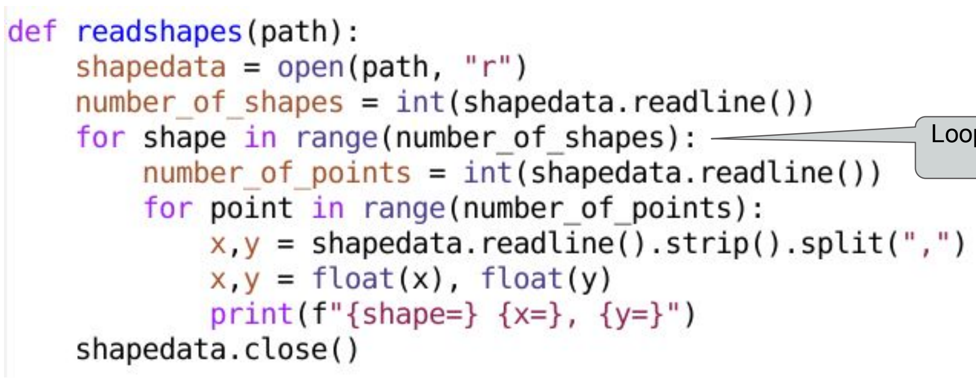
Reading

```python
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

Read the first line, convert to integer

Then the next line is the number of shapes, we convert that to an integer.

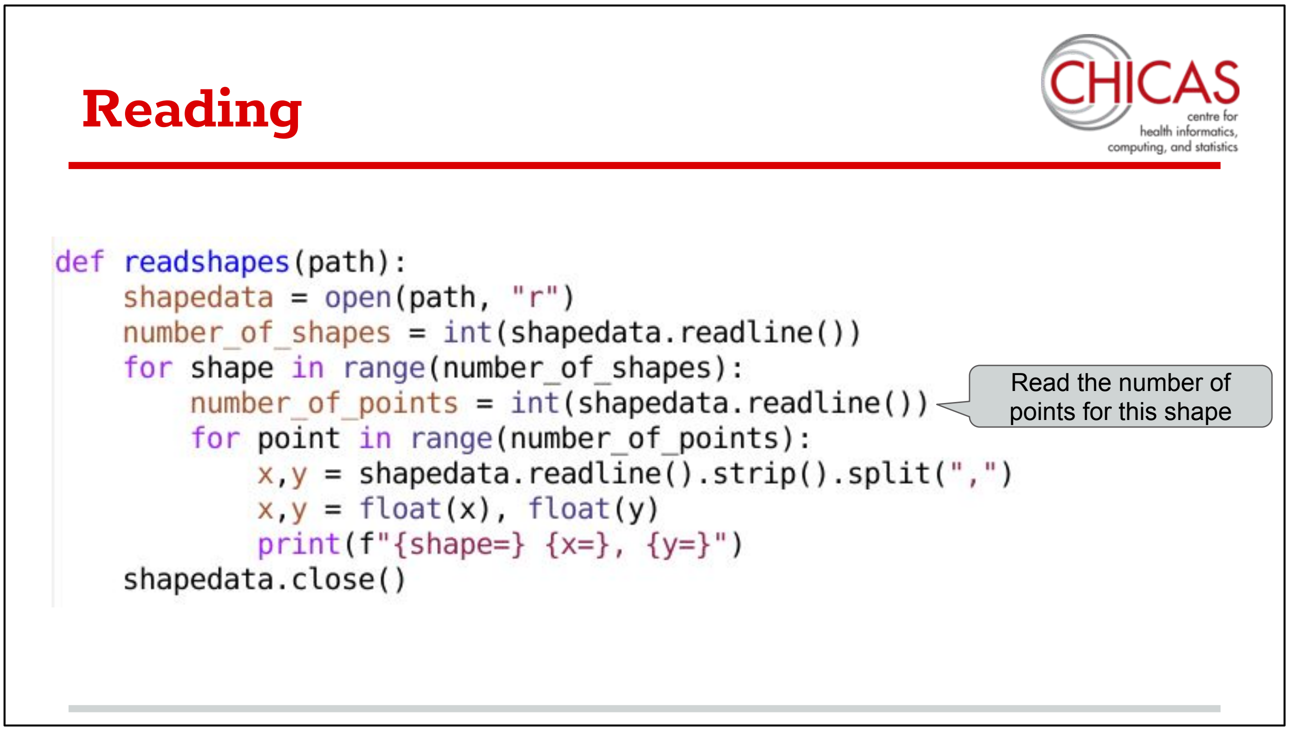
Reading



```
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

Loop over the number of shapes

Next we loop over the number of shapes….

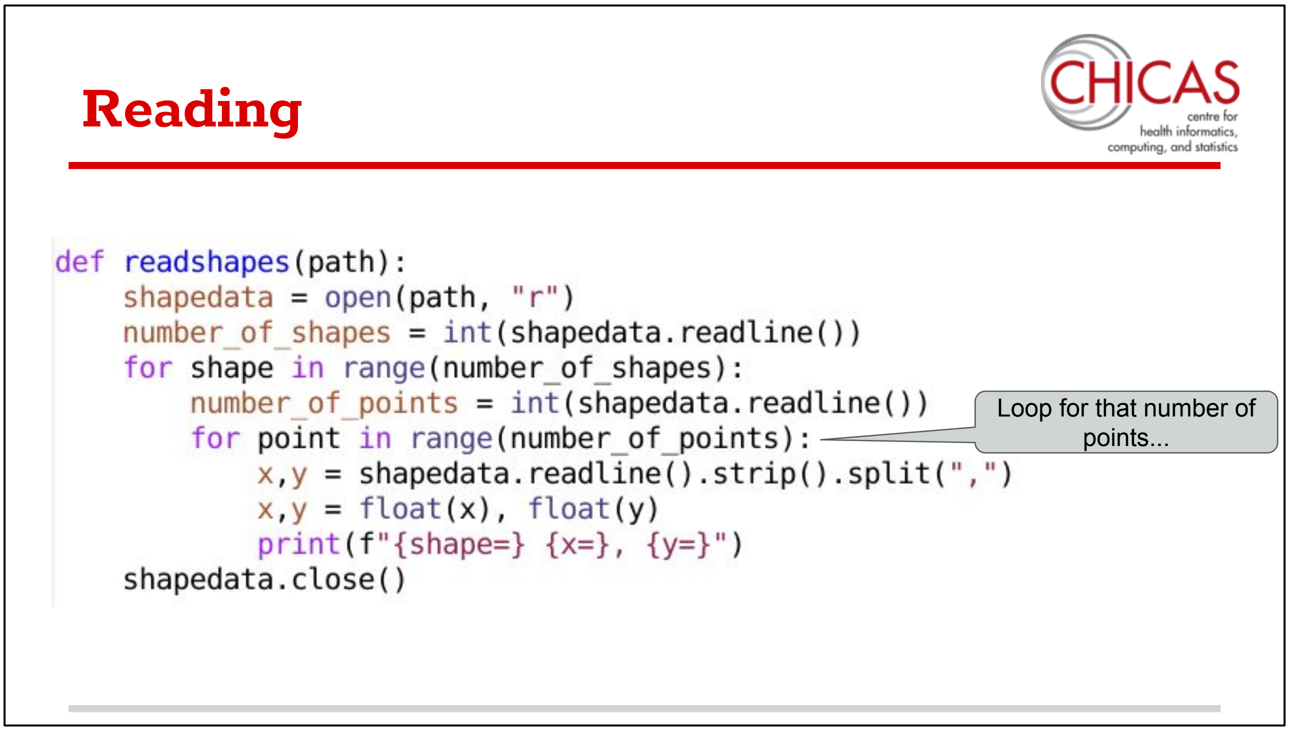


Reading

```
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

Read the number of points for this shape

The next line is the number of points for this shape. Again we convert to integer.



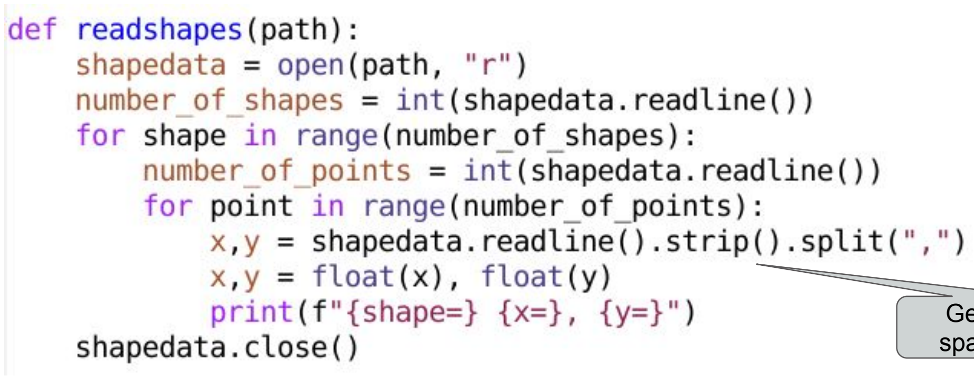
Reading

```python
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

Loop for that number of points...

Now we have an inner loop for the number of points…

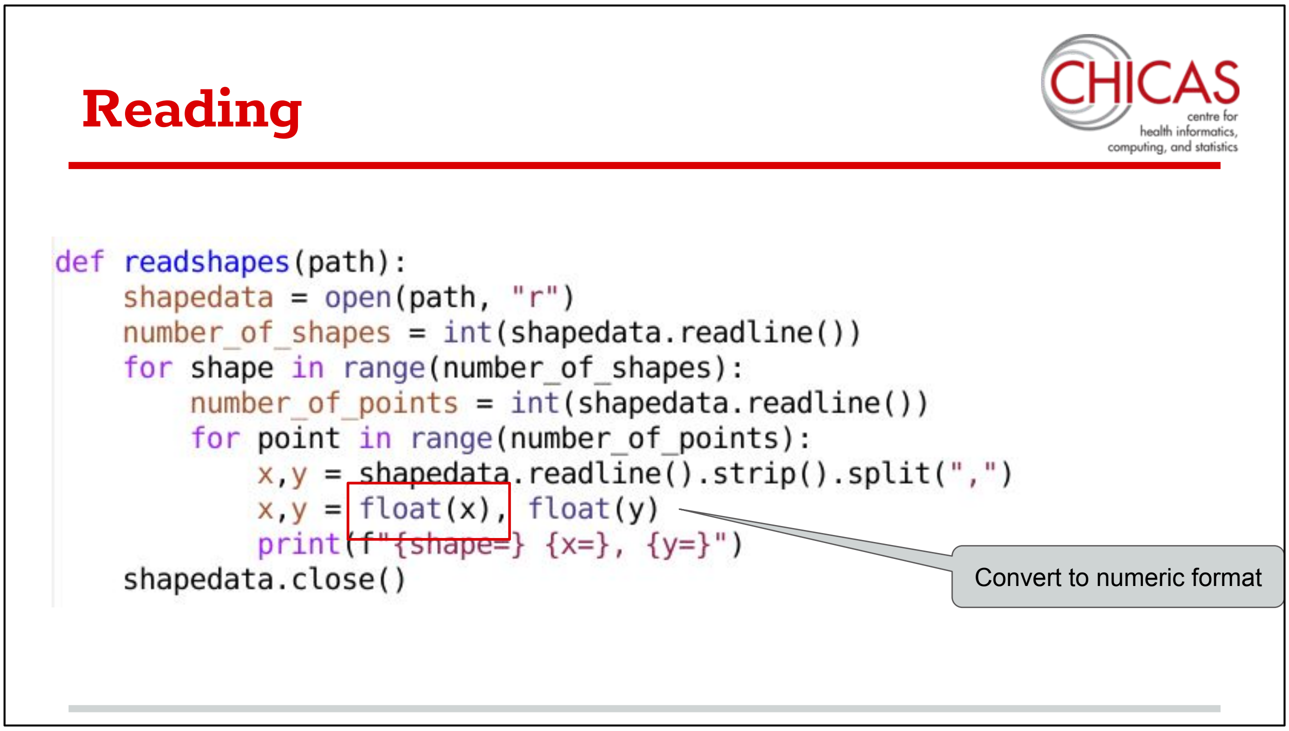
Reading



```
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

Get the line, strip any space, split on comma

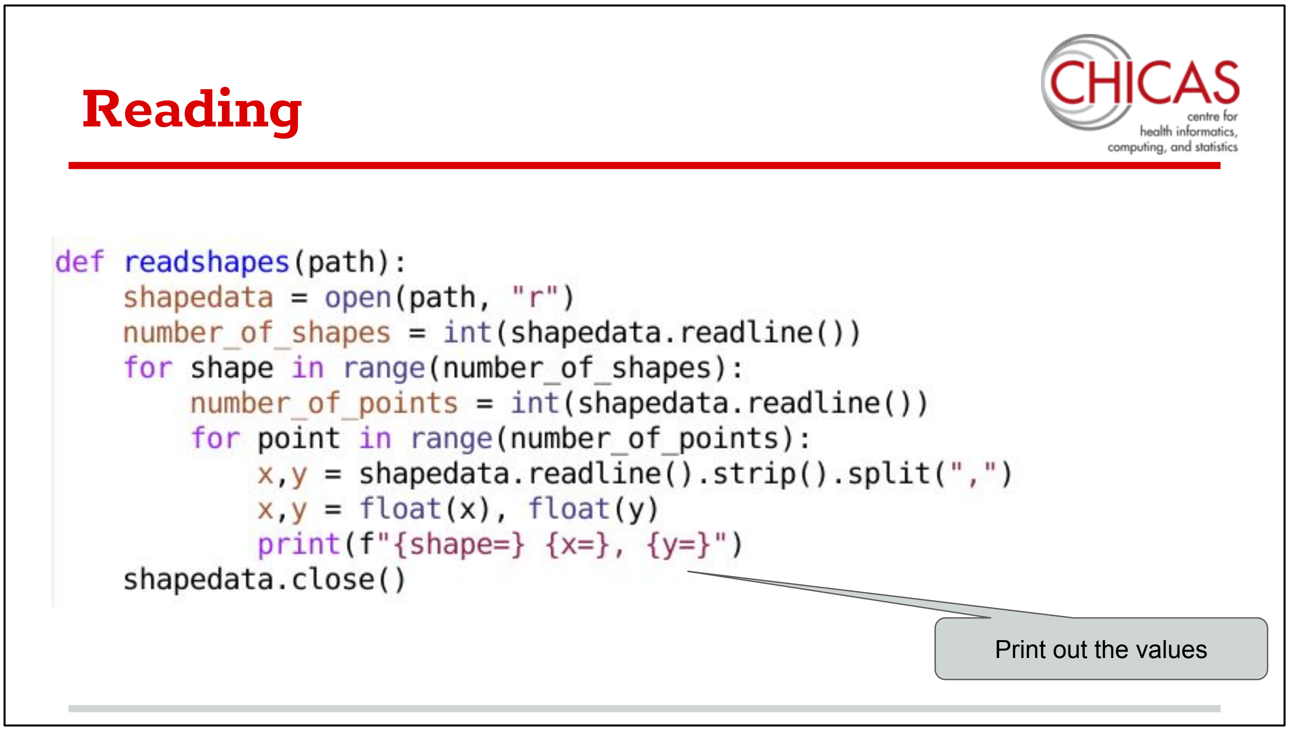
Read the line, strip the line-end marker, split the text by comma. This gets back two text values into x and y



Reading

```
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

These need to be decimal numbers so we conver them with “float”.

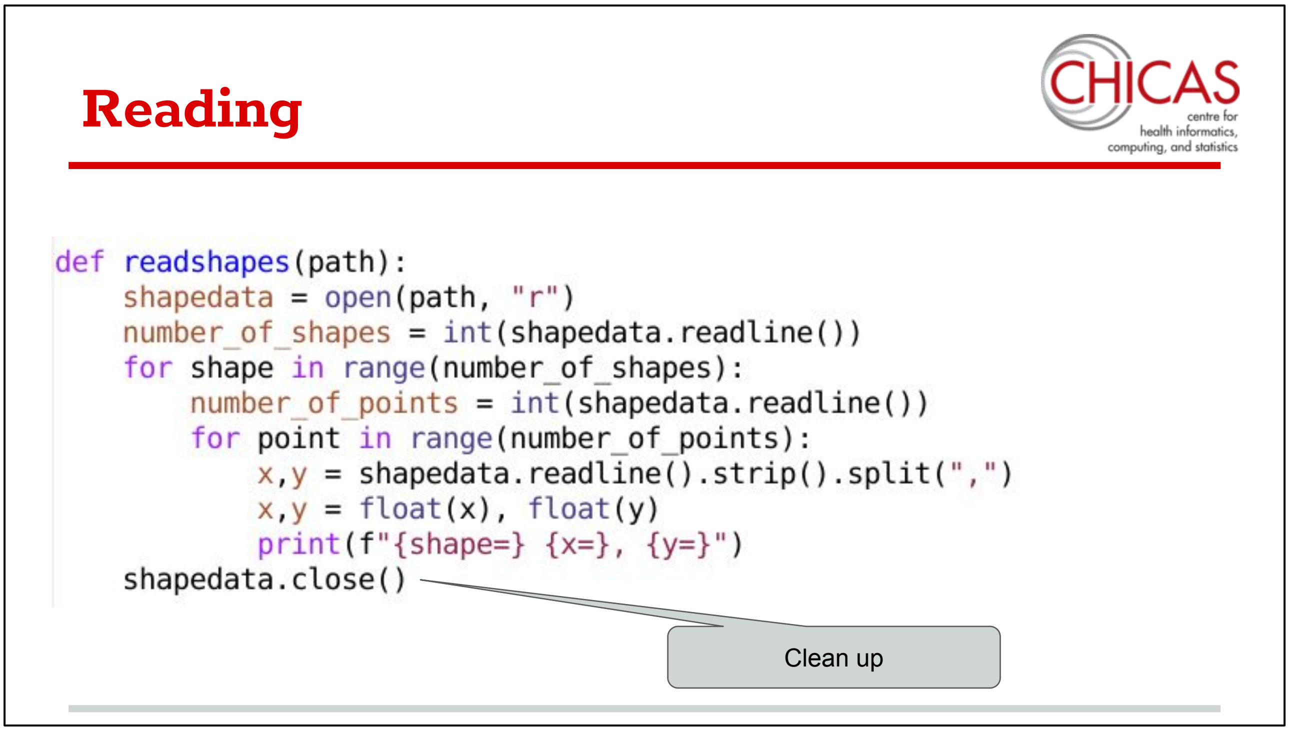


Reading

```python
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

Print out the values

And then we can print them out, here using the very handy f-string notation.

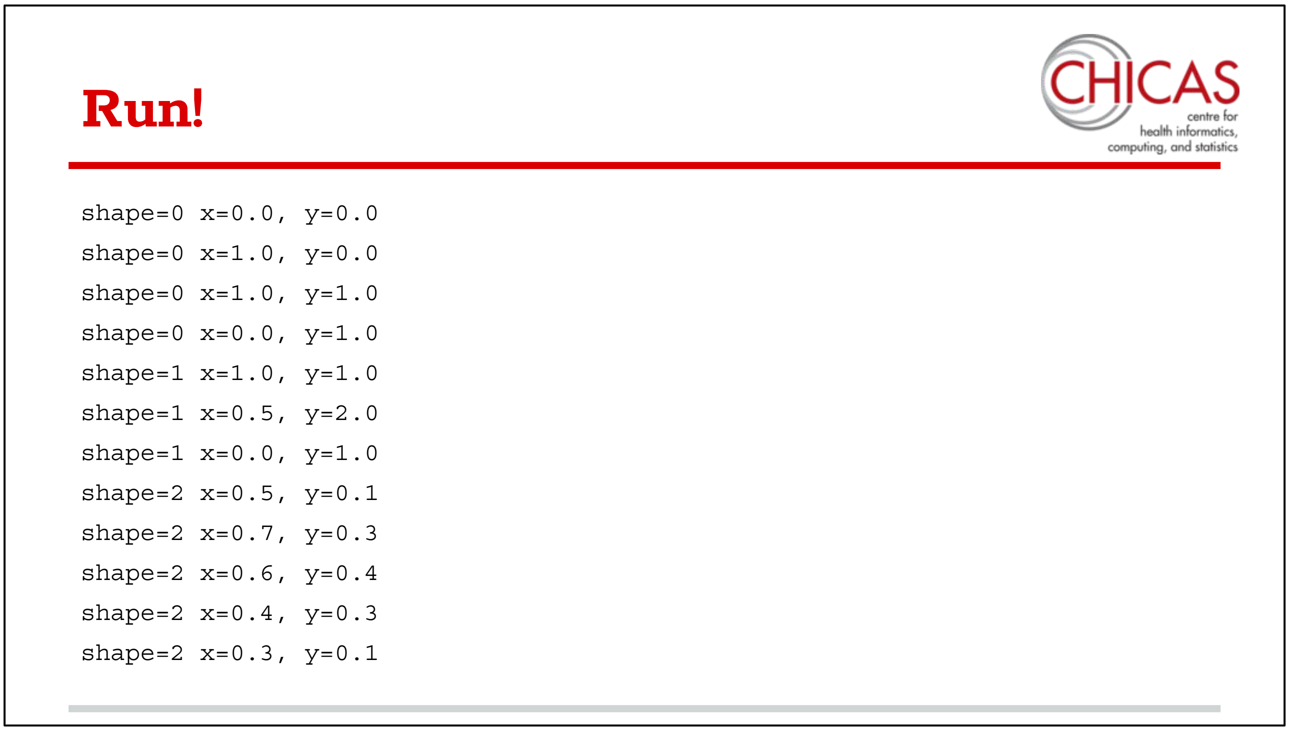


Reading

```python
def readshapes(path):
    shapedata = open(path, "r")
    number_of_shapes = int(shapedata.readline())
    for shape in range(number_of_shapes):
        number_of_points = int(shapedata.readline())
        for point in range(number_of_points):
            x,y = shapedata.readline().strip().split(",")
            x,y = float(x), float(y)
            print(f"{shape=} {x=}, {y=}")
    shapedata.close()
```

Clean up

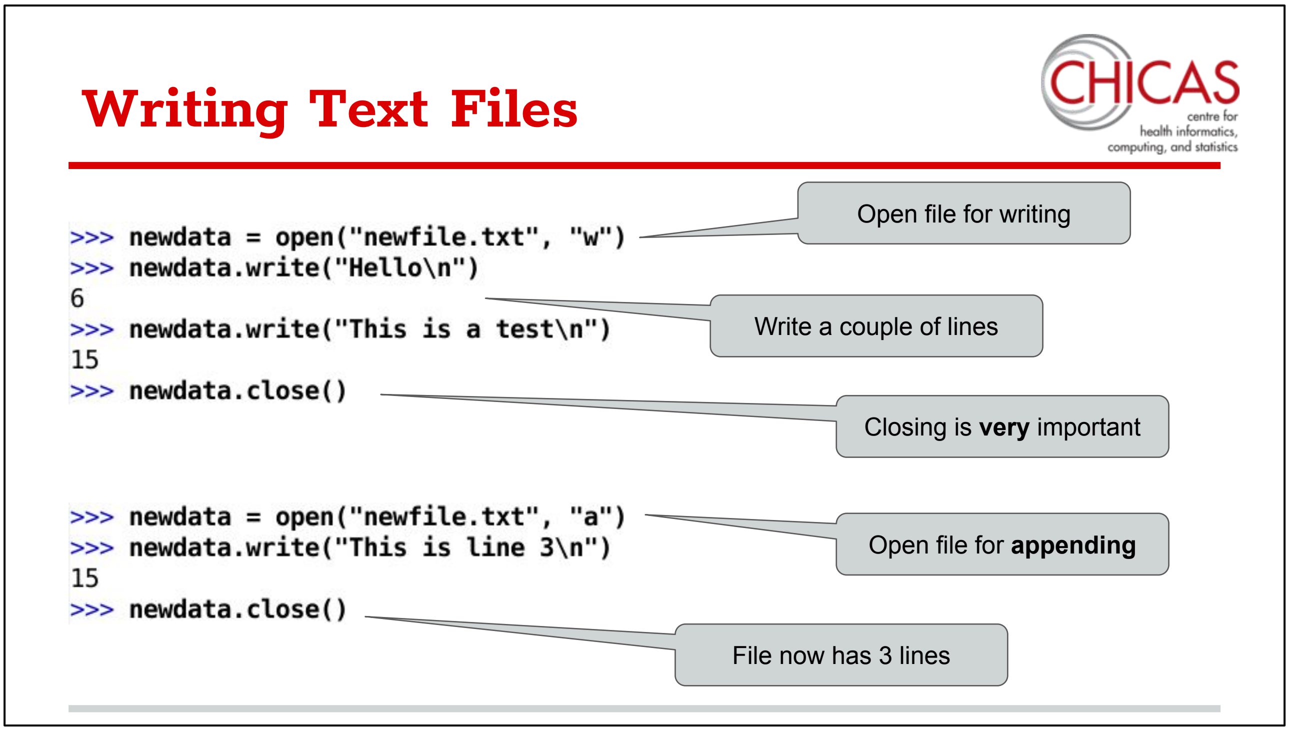
And finally close the connection.



Run!

```
shape=0 x=0.0, y=0.0
shape=0 x=1.0, y=0.0
shape=0 x=1.0, y=1.0
shape=0 x=0.0, y=1.0
shape=1 x=1.0, y=1.0
shape=1 x=0.5, y=2.0
shape=1 x=0.0, y=1.0
shape=2 x=0.5, y=0.1
shape=2 x=0.7, y=0.3
shape=2 x=0.6, y=0.4
shape=2 x=0.4, y=0.3
shape=2 x=0.3, y=0.1
```

Run that, and this gets printed. How would you return these values instead of printing them out?

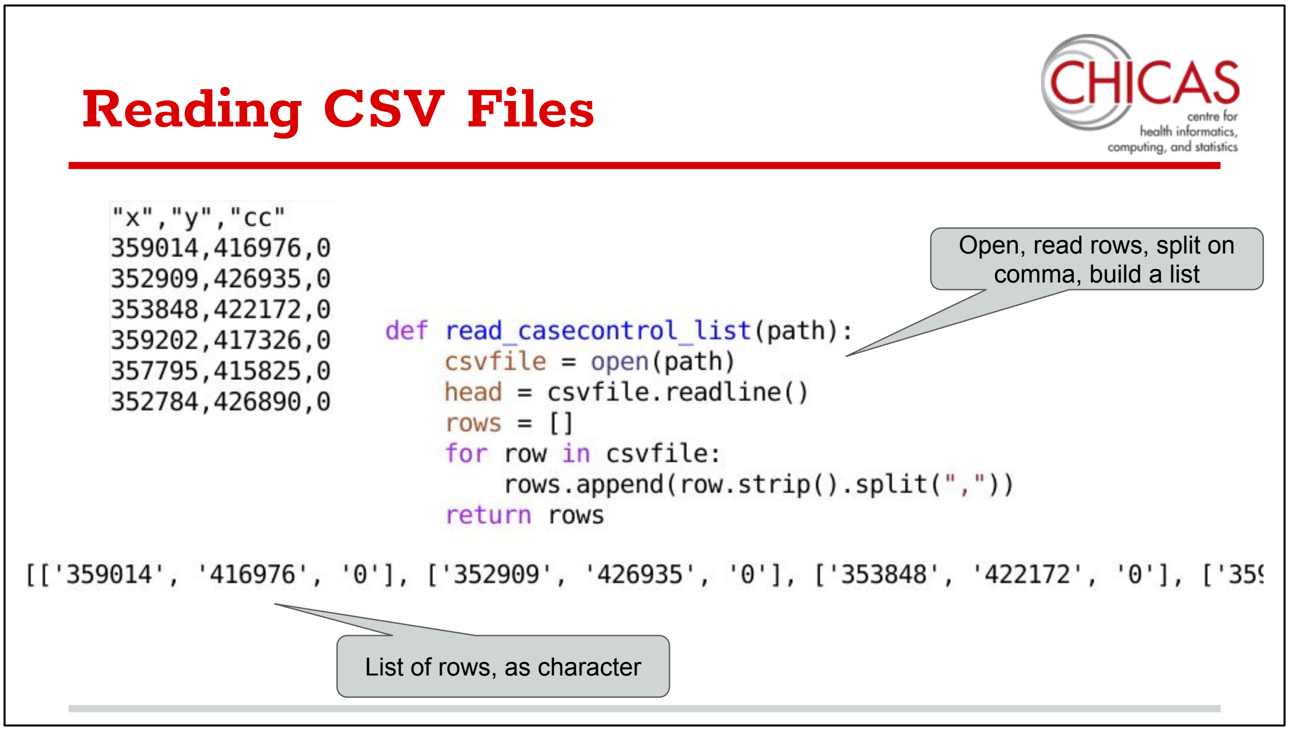


Writing Text Files

```
>>> newdata = open("newfile.txt", "w")
>>> newdata.write("Hello\n")
6
>>> newdata.write("This is a test\n")
15
>>> newdata.close()
```

```
>>> newdata = open("newfile.txt", "a")
>>> newdata.write("This is line 3\n")
15
>>> newdata.close()
```

Writing text files is similar to reading them except you use the “write” method instead of “read” ones. Note that closing the file is very important here since the operating system might not have actually written the data to disk until the “close” operation is complete. You can also open a file for appending lines if you want to add text to a file.



Reading CSV Files

```
"x","y","cc"
359014,416976,0
352909,426935,0
353848,422172,0
359202,417326,0
357795,415825,0
352784,426890,0
```

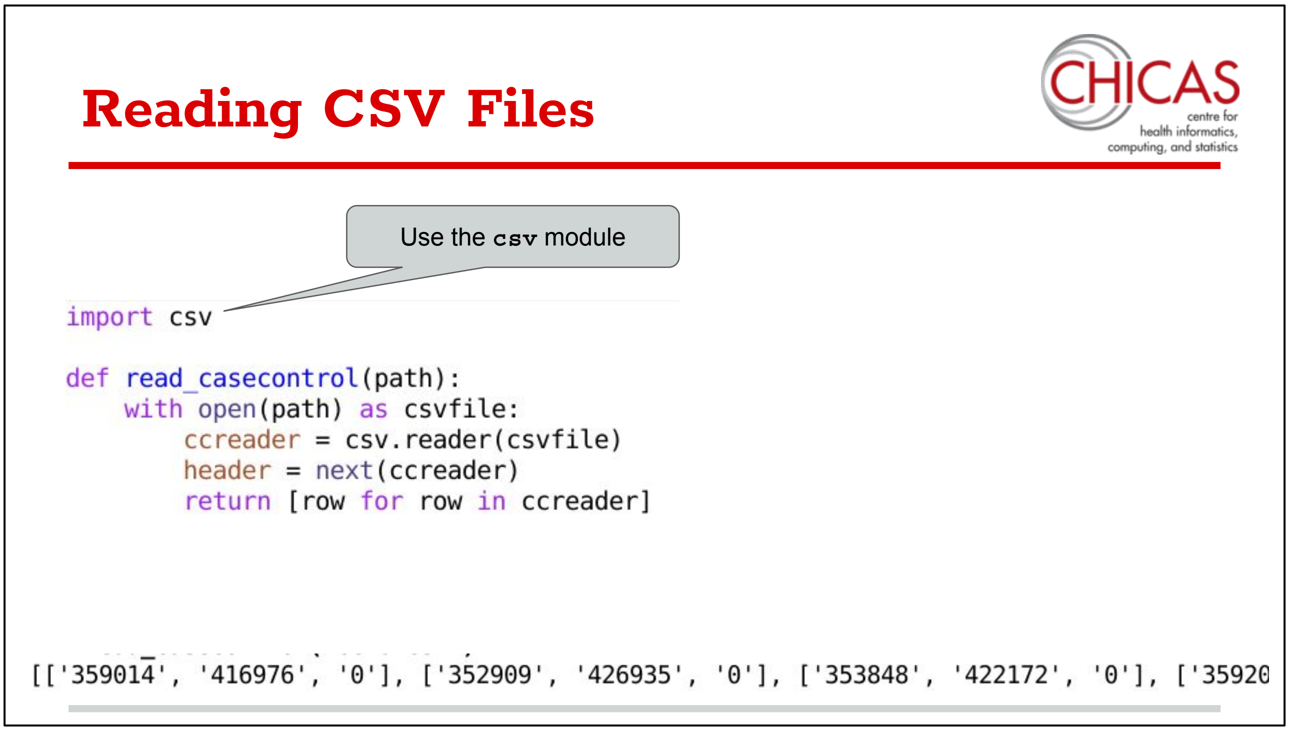
```python
def read_casecontrol_list(path):
    csvfile = open(path)
    head = csvfile.readline()
    rows = []
    for row in csvfile:
        rows.append(row.strip().split(","))
    return rows
```

Open, read rows, split on comma, build a list

```
[['359014', '416976', '0'], ['352909', '426935', '0'], ['353848', '422172', '0'], ['35
```

List of rows, as character

Commonly our data is a bit more structured, often as a tabular structure in a CSV file. You could open the file and read the lines in using a loop, splitting on commas to get text values for each row.



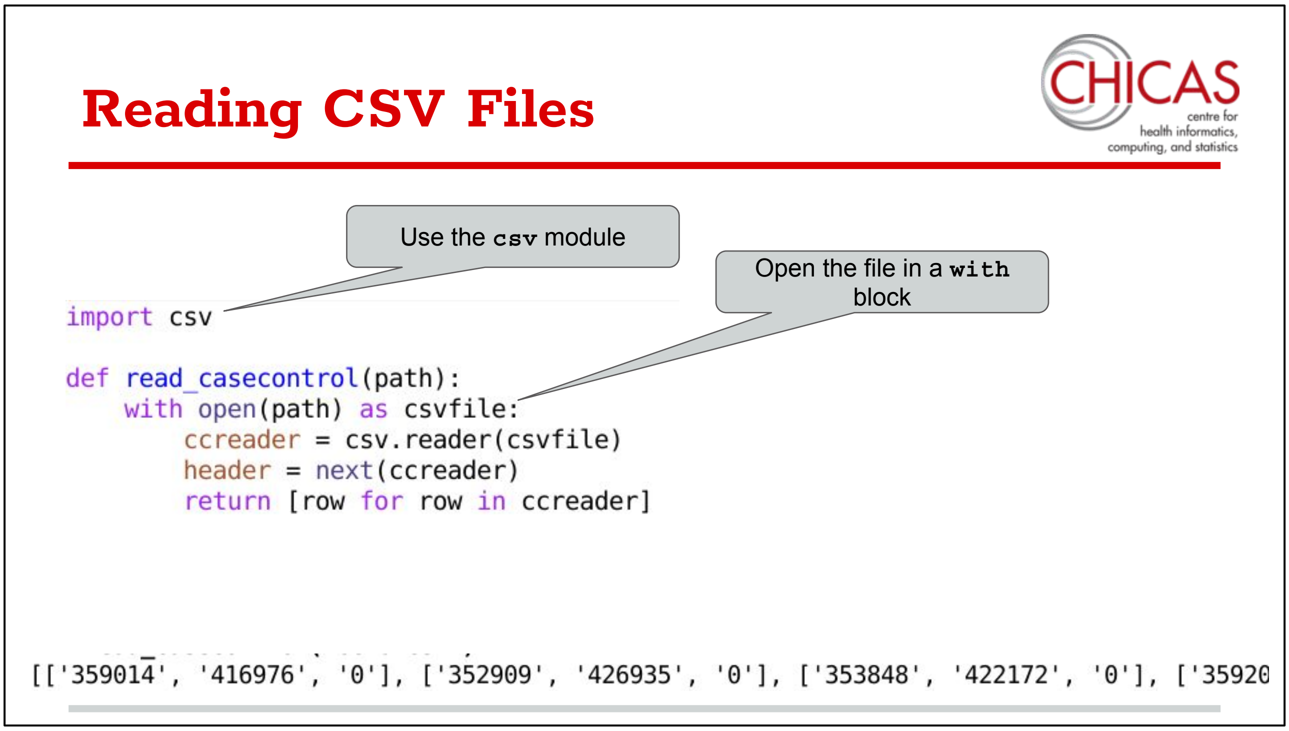
Reading CSV Files

Use the `csv` module

```
import csv

def read_casecontrol(path):
    with open(path) as csvfile:
        ccreader = csv.reader(csvfile)
        header = next(ccreader)
        return [row for row in ccreader]
```

```
[['359014', '416976', '0'], ['352909', '426935', '0'], ['353848', '422172', '0'], ['35920
```

But this is such a common thing that there is a built-in python module for reading CSV files. Import it…



Reading CSV Files

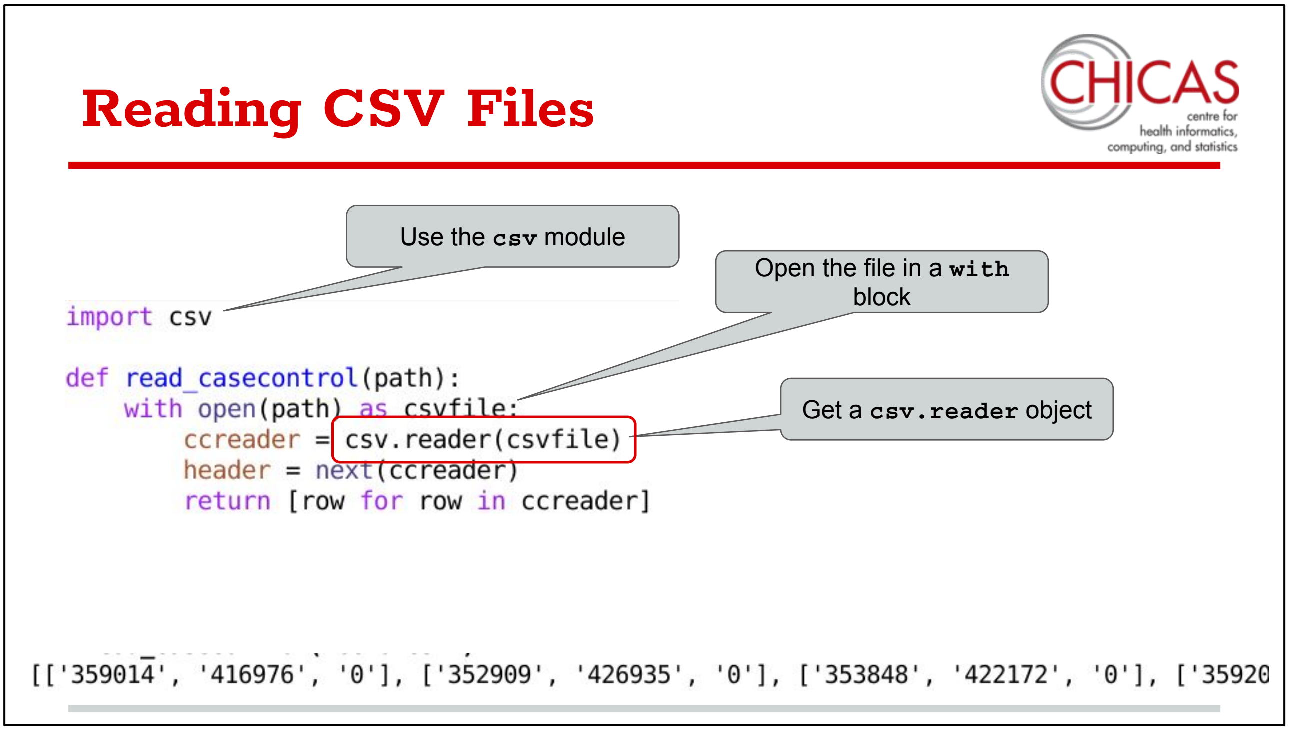
Use the **csv** module

Open the file in a **with** block

```
import csv

def read_casecontrol(path):
    with open(path) as csvfile:
        ccreader = csv.reader(csvfile)
        header = next(ccreader)
        return [row for row in ccreader]
```

```
[['359014', '416976', '0'], ['352909', '426935', '0'], ['353848', '422172', '0'], ['35920
```

Then open the file in a “with” block. This will ensure the file is closed at the end of the code.



Reading CSV Files

Use the `csv` module

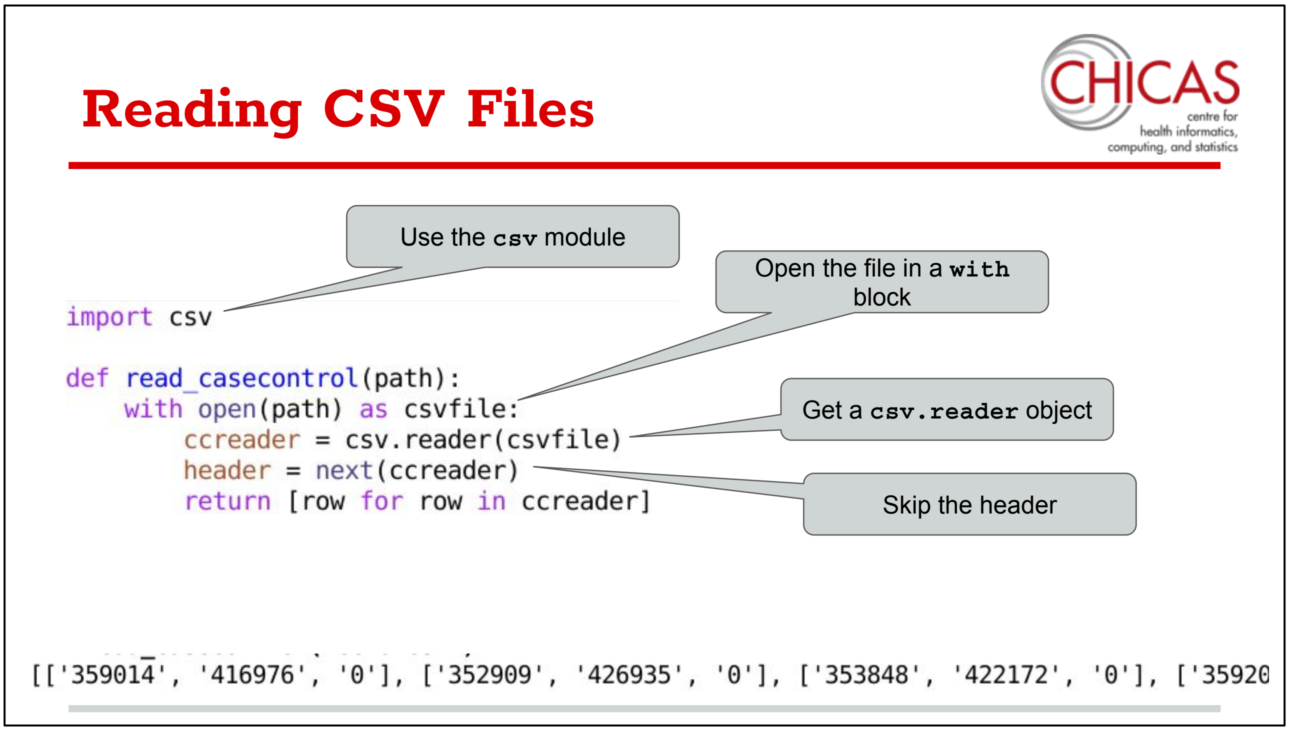
Open the file in a `with` block

Get a `csv.reader` object

```
import csv

def read_casecontrol(path):
    with open(path) as csvfile:
        ccreader = csv.reader(csvfile)
        header = next(ccreader)
        return [row for row in ccreader]
```

```
[['359014', '416976', '0'], ['352909', '426935', '0'], ['353848', '422172', '0'], ['35920
```

Then we get something like the ordinary file handle, but it knows its reading a CSV…



Reading CSV Files

CHICAS centre for health informatics, computing, and statistics

Use the **`csv`** module

Open the file in a **`with`** block

Get a **`csv.reader`** object

Skip the header

```
import csv

def read_casecontrol(path):
    with open(path) as csvfile:
        ccreader = csv.reader(csvfile)
        header = next(ccreader)
        return [row for row in ccreader]
```

```
[['359014', '416976', '0'], ['352909', '426935', '0'], ['353848', '422172', '0'], ['35920
```

That object is “iterable”, so we can call “next” on it to get the next line. This gets us the headers.

Reading CSV Files

CHICAS centre for health informatics, computing, and statistics

Use the **csv** module

Open the file in a **with** block

Get a **csv.reader** object

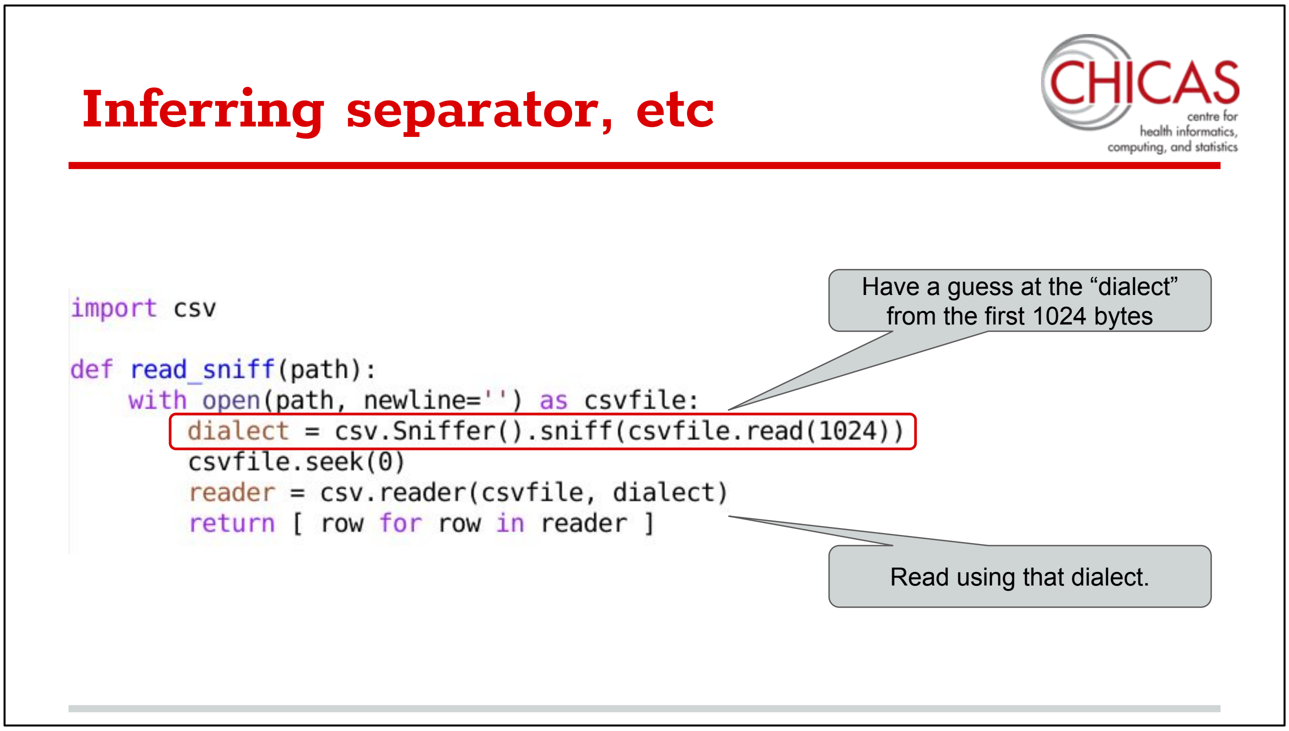
Skip the header

Loop over the rest

```
import csv

def read_casecontrol(path):
    with open(path) as csvfile:
        ccreader = csv.reader(csvfile)
        header = next(ccreader)
        return [row for row in ccreader]
```

```
[['359014', '416976', '0'], ['352909', '426935', '0'], ['353848', '422172', '0'], ['35920
```

Then we can iterate to get the rest of the lines, and you can see them as a list of lists of character-based items.


Inferring separator, etc

```
import csv

def read_sniff(path):
    with open(path, newline='') as csvfile:
        dialect = csv.Sniffer().sniff(csvfile.read(1024))
        csvfile.seek(0)
        reader = csv.reader(csvfile, dialect)
        return [ row for row in reader ]
```

Have a guess at the "dialect" from the first 1024 bytes

Read using that dialect.

CSV files can have different separators (commas, semicolons, spaces etc) or other oddities that we can think of as different "dialects" of CSV. The csv package has a mechanism for having a guess at the dialect using a "sniffer". This looks at the first 1024 characters in the file and if it sees commas, it sets the separator to comma, and so on. Then you can seek back to the start of the file and read the rows.

Side Note: Incomprehension

Note that:

```
return [ row for row in reader ]
```

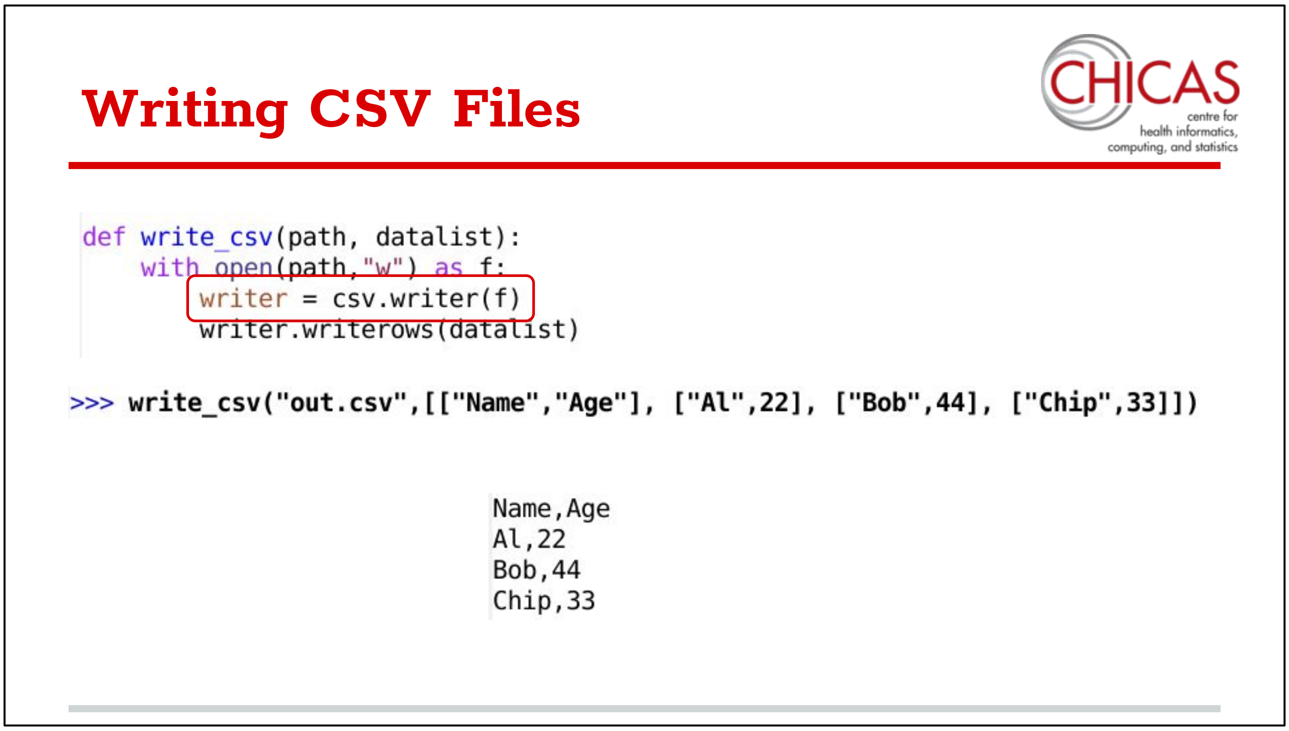
Is nearly always better expressed as:

```
return list(reader)
```

Use that structure if modifying (or filtering)the elements:

```
[line.strip() for line in h]
```

Note I've done a non-pythonic thing in the last couple of slides. This list comprehension is really a sign that I've forgotten that I can just iterate over an iterable with the "list" function. I only really need a comprehension if I want to modify or filter the elements in a list.



Writing CSV Files

```
def write_csv(path, datalist):
    with open(path,"w") as f:
        writer = csv.writer(f)
        writer.writerows(datalist)
```

```
>>> write_csv("out.csv",[["Name","Age"], ["Al",22], ["Bob",44], ["Chip",33]])
```

```
Name,Age
Al,22
Bob,44
Chip,33
```

Writing data to a CSV is similar. You can take a list of lists and send them to a file with a csv writer object.

Data Frames with Pandas

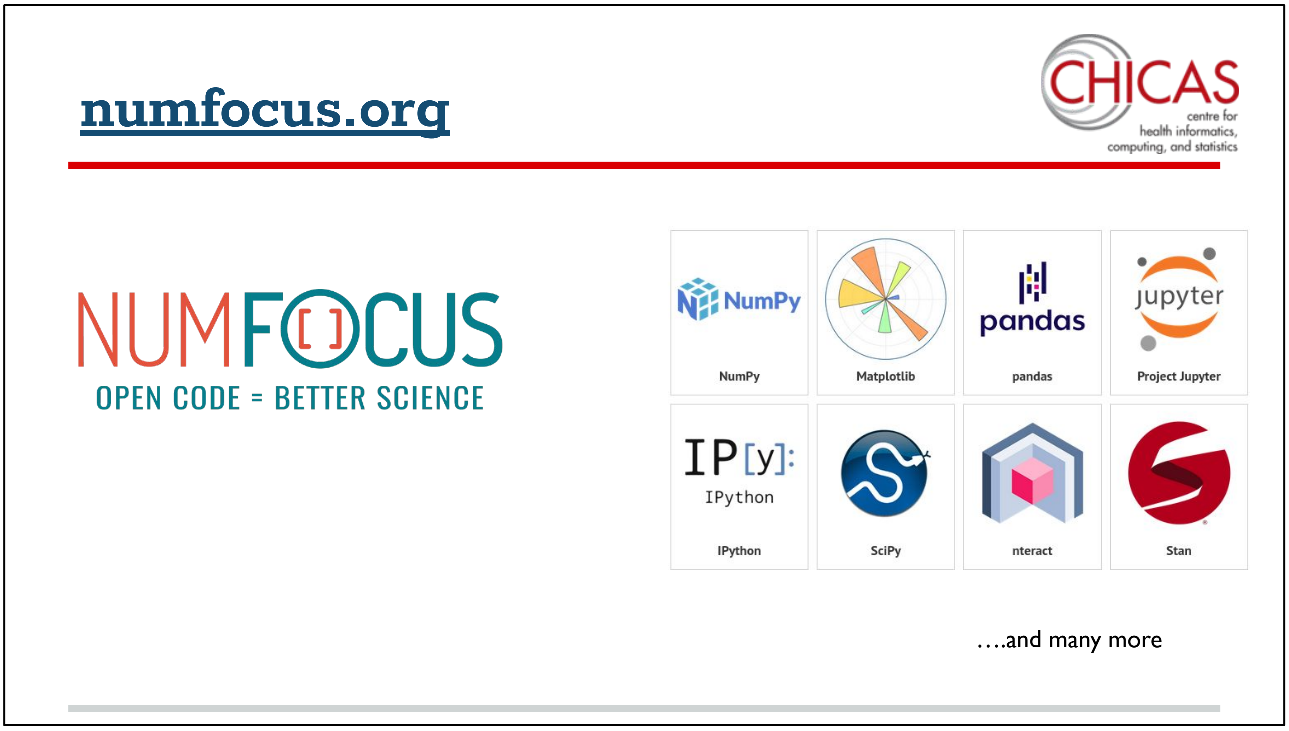
CHICAS centre for health informatics, computing, and statistics

https://pandas.pydata.org/

- Data Series
- Data Frame

Part of “PyData”

Python’s lists and dictionaries are okay for storing simple data, but for more complex situations we need a more capable data structure. And for tabular data we have the “pandas” project, which gives us data series and data frames, as well as some other data types. Its part of the PyData project.



numfocus.org

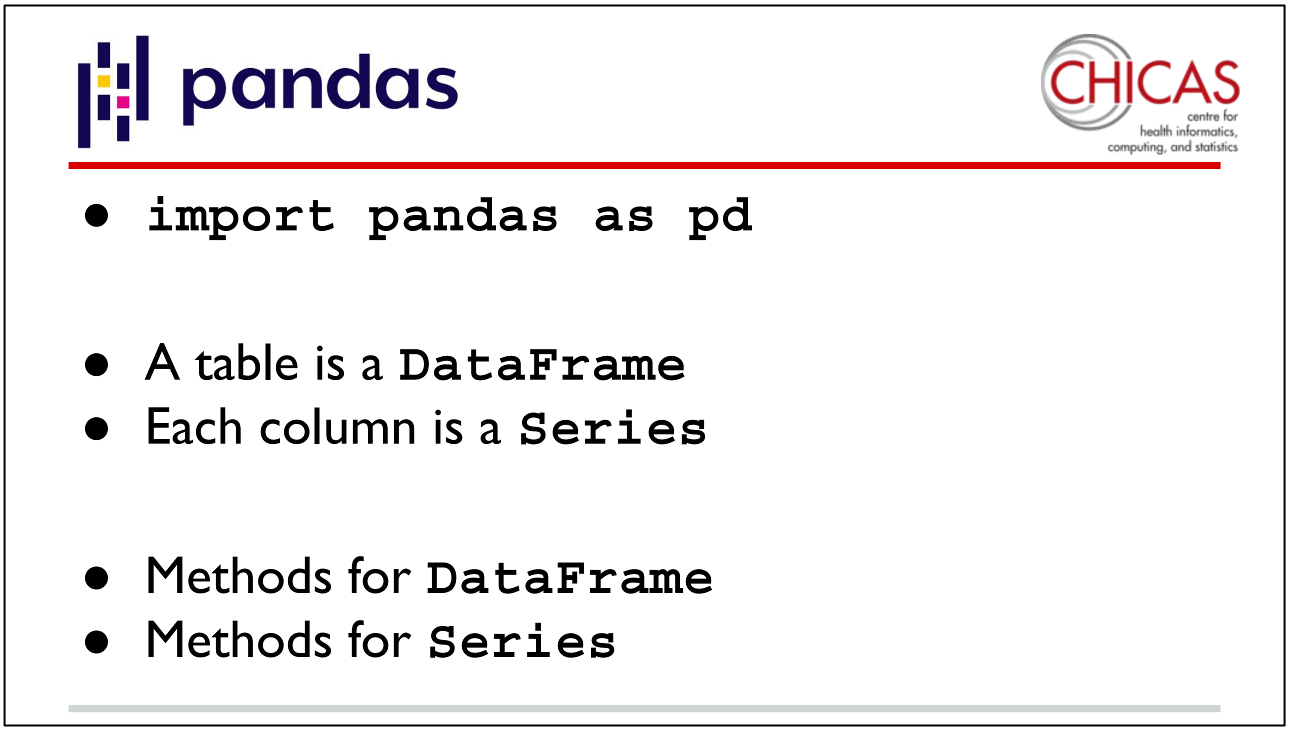
CHICAS centre for health informatics, computing, and statistics

NUMFOCUS
OPEN CODE = BETTER SCIENCE

NumPy	Matplotlib	pandas	Project Jupyter
IPython	SciPy	nteract	Stan

….and many more

PyData is a project of the NumFocus organisation, a US charity that sponsors various open source projects in the data science arena. Some we've used already.



pandas

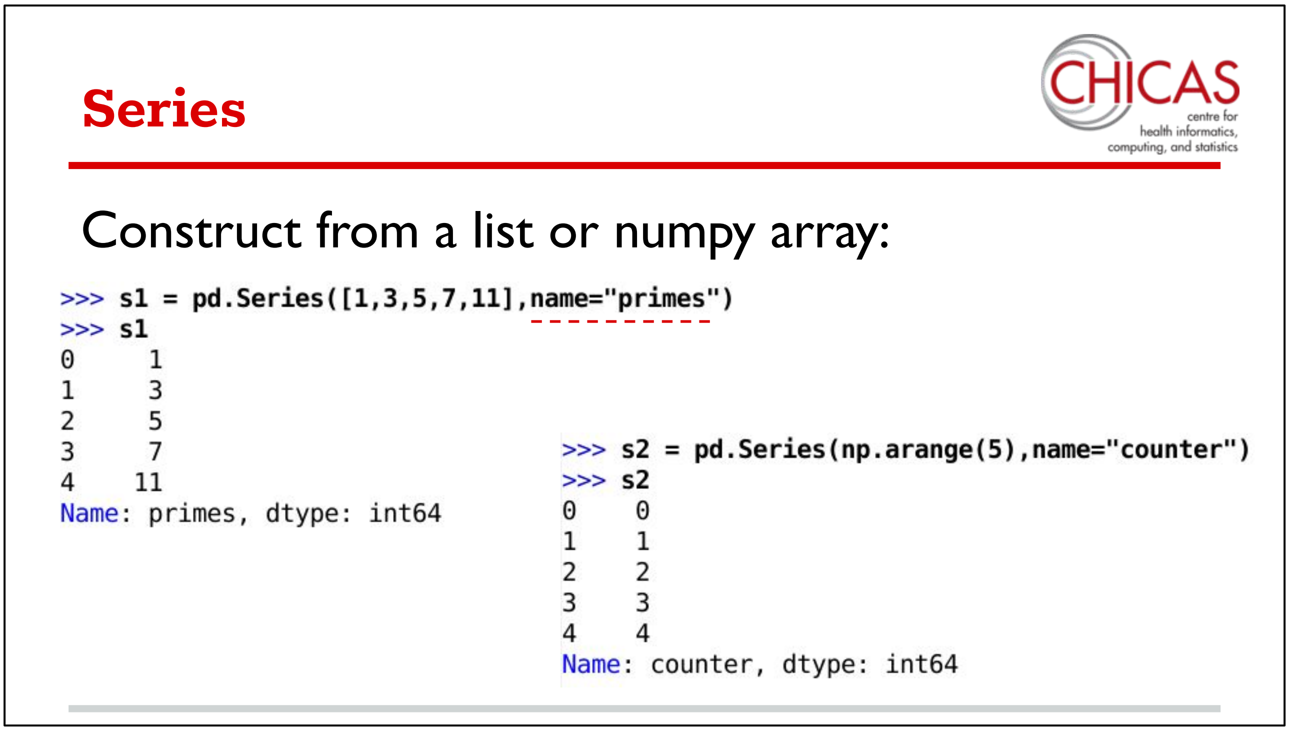
CHICAS
centre for
health informatics,
computing, and statistics

- `import pandas as pd`

- A table is a `DataFrame`
- Each column is a `Series`

- Methods for `DataFrame`
- Methods for `Series`

So pandas. Its conventionally imported as “pd” for short.



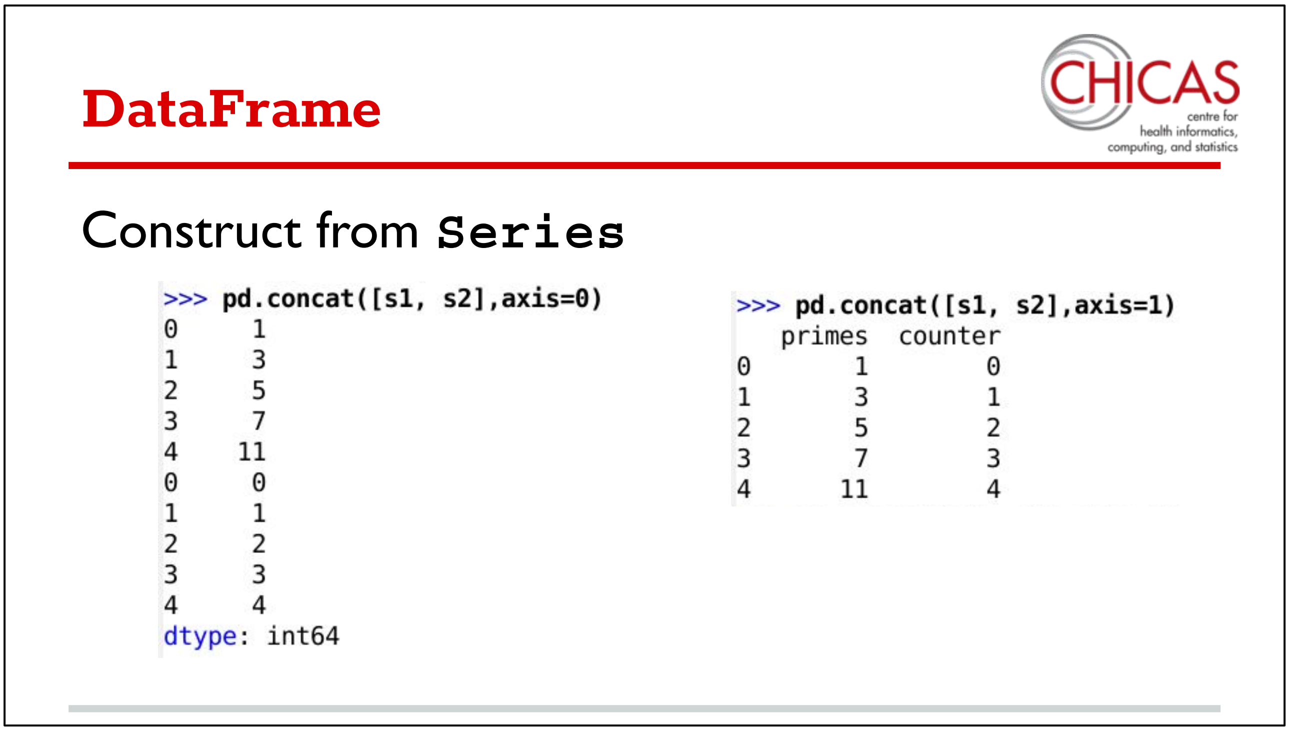
Series

Construct from a list or numpy array:

```
>>> s1 = pd.Series([1,3,5,7,11],name="primes")
>>> s1
0     1
1     3
2     5
3     7
4    11
Name: primes, dtype: int64
```

```
>>> s2 = pd.Series(np.arange(5),name="counter")
>>> s2
0    0
1    1
2    2
3    3
4    4
Name: counter, dtype: int64
```

A Series is like a vector in R, but it has a name attached to it (that’s separate from the name of the object). You can construct them from lists. They also have a data type. These here are two numeric integer series.



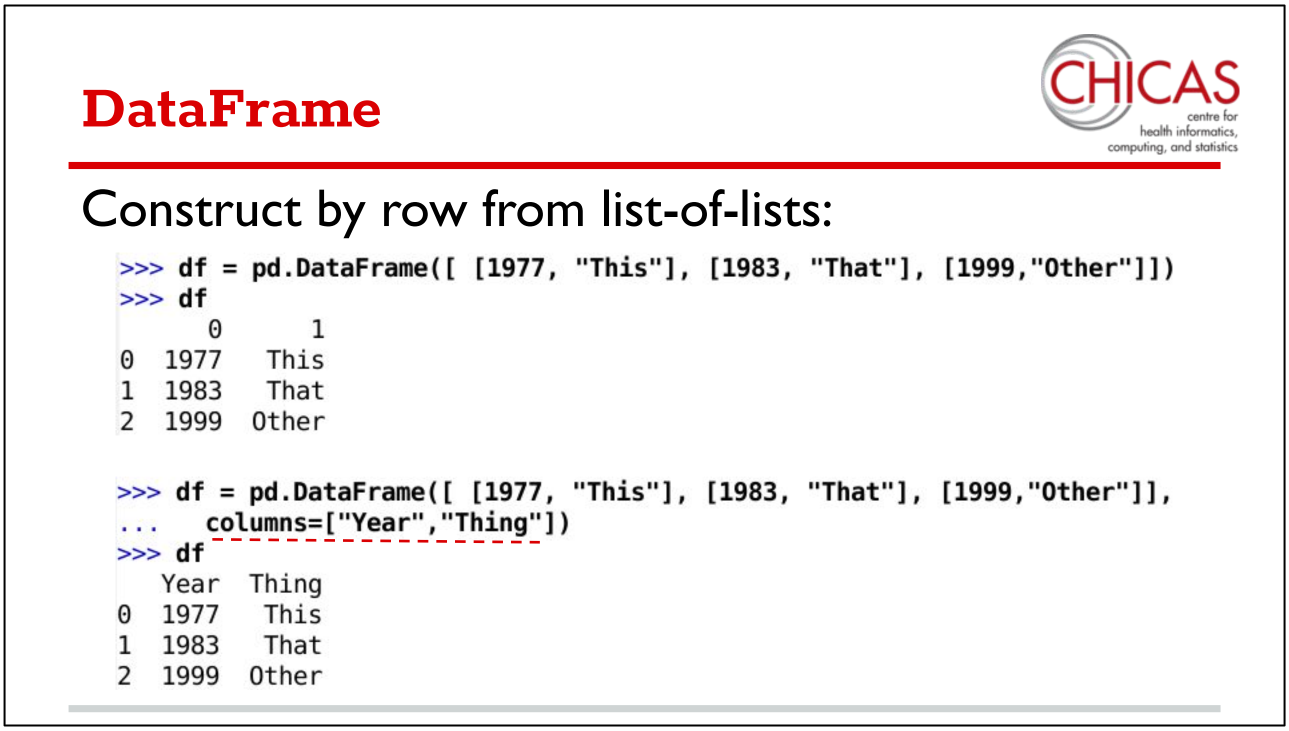
DataFrame

Construct from `Series`

```
>>> pd.concat([s1, s2],axis=0)
0     1
1     3
2     5
3     7
4    11
0     0
1     1
2     2
3     3
4     4
dtype: int64
```

```
>>> pd.concat([s1, s2],axis=1)
   primes  counter
0       1        0
1       3        1
2       5        2
3       7        3
4      11        4
```

You can make a Data Frame from Series objects, for example by “concat”, but make sure you get the axis right.



DataFrame

CHICAS centre for health informatics, computing, and statistics

Construct by row from list-of-lists:

```
>>> df = pd.DataFrame([ [1977, "This"], [1983, "That"], [1999,"Other"]])
>>> df
      0      1
0  1977   This
1  1983   That
2  1999  Other

>>> df = pd.DataFrame([ [1977, "This"], [1983, "That"], [1999,"Other"]],
...    columns=["Year","Thing"])
>>> df
   Year  Thing
0  1977   This
1  1983   That
2  1999  Other
```

You can also construct a data frame by rows from the elements in a list. You can name the columns with a “columns” argument.

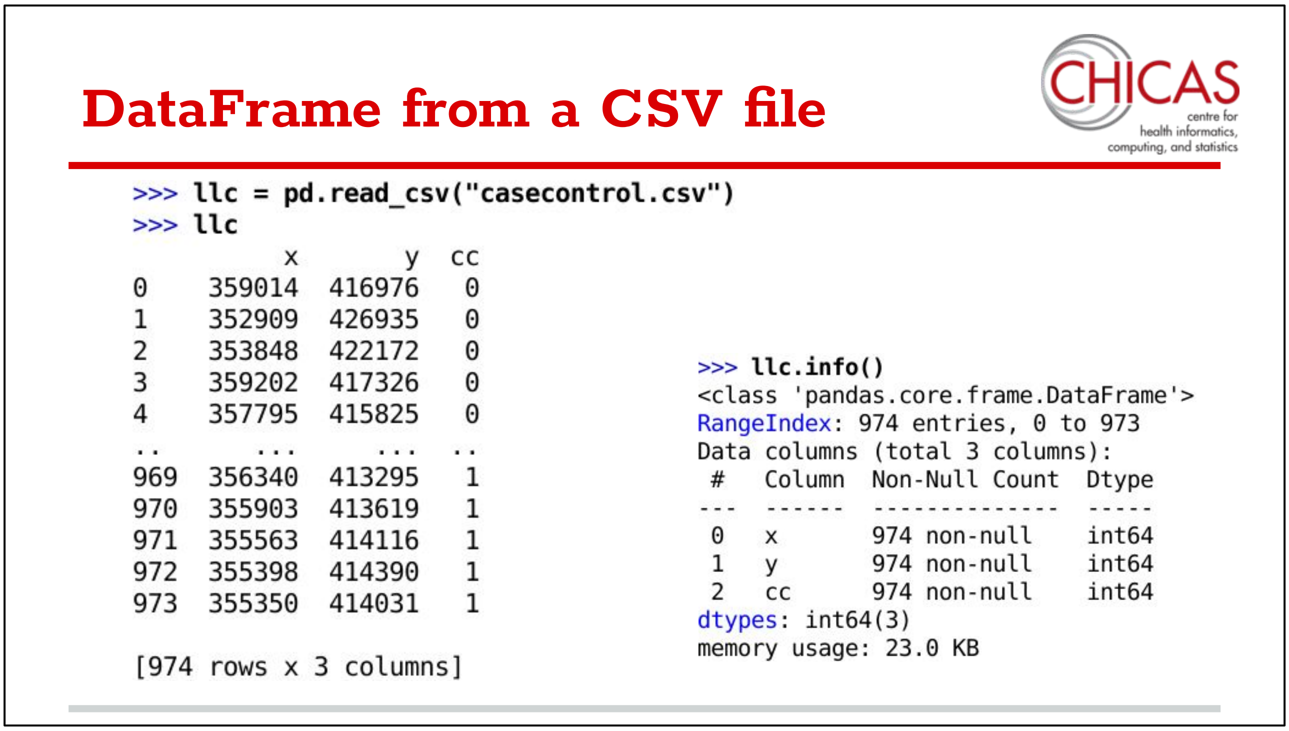
DataFrame by columns from dict

```
>>> data_dict = dict(Year=[1977, 1983, 1999], Thing=["This","That","Other"])
```

```
>>> data_dict
{'Year': [1977, 1983, 1999], 'Thing': ['This', 'That', 'Other']}
```

```
>>> pd.DataFrame(data_dict)
   Year  Thing
0  1977   This
1  1983   That
2  1999  Other
```

If you want to create by columns, the easiest way is probably with a dictionary, where it gets the names from the keys of the dictionary.



DataFrame from a CSV file

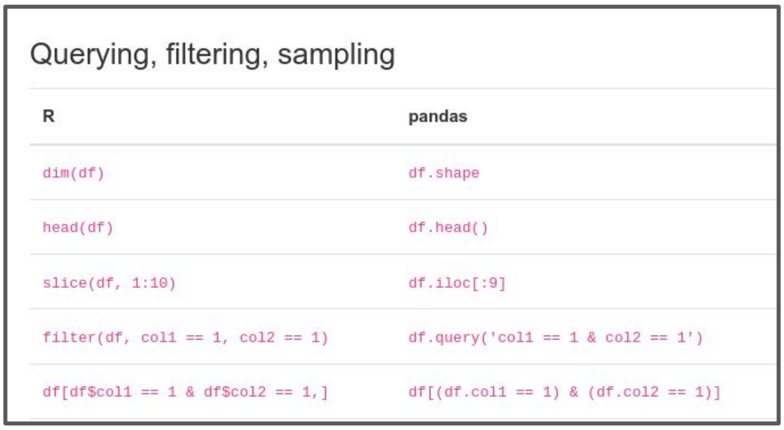
```
>>> llc = pd.read_csv("casecontrol.csv")
>>> llc
          x       y  cc
0    359014  416976   0
1    352909  426935   0
2    353848  422172   0
3    359202  417326   0
4    357795  415825   0
..      ...     ...  ..
969  356340  413295   1
970  355903  413619   1
971  355563  414116   1
972  355398  414390   1
973  355350  414031   1

[974 rows x 3 columns]
```

```
>>> llc.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 974 entries, 0 to 973
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   x       974 non-null    int64
 1   y       974 non-null    int64
 2   cc      974 non-null    int64
dtypes: int64(3)
memory usage: 23.0 KB
```

Pandas has a read_csv method for reading directly from a CSV file. You can also change the dialect with some other arguments.
Once you've got a pandas data frame, you can get a quick summary of the columns with the "info" method.

DataFrame methods

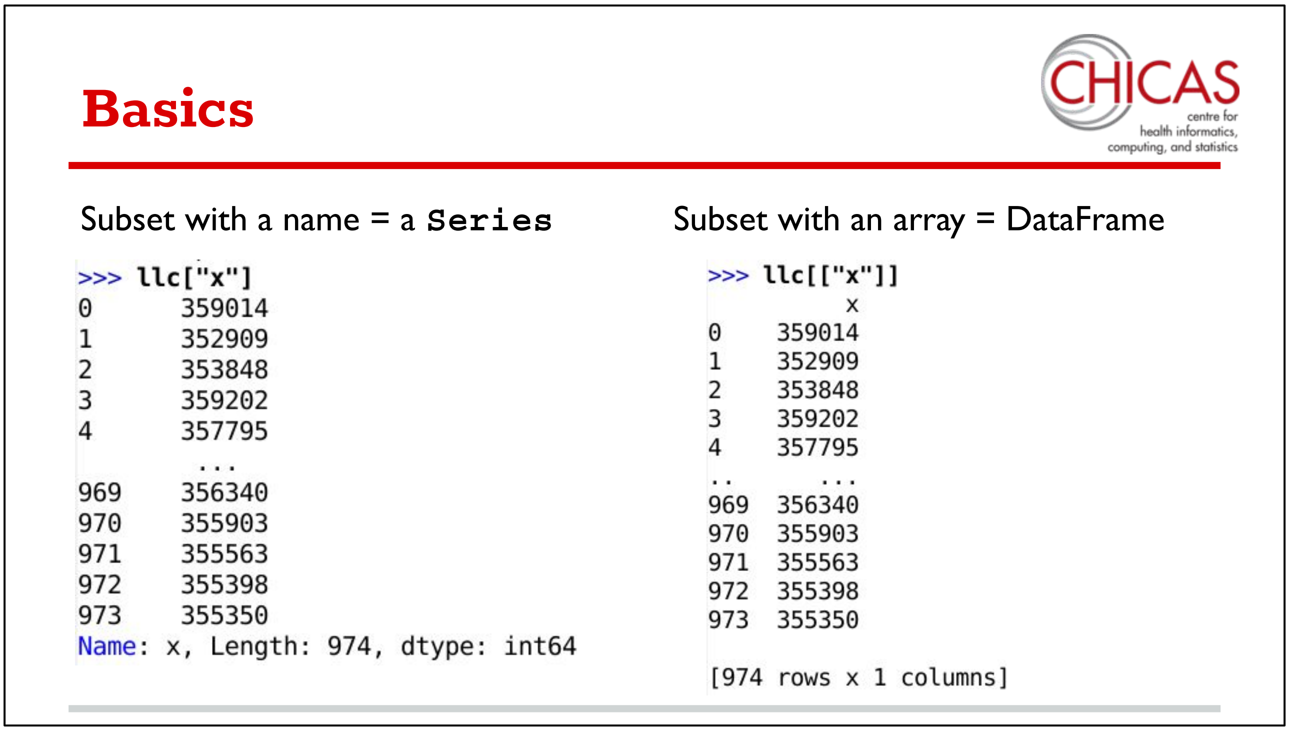
https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_r.html



Querying, filtering, sampling

R	pandas
`dim(df)`	`df.shape`
`head(df)`	`df.head()`
`slice(df, 1:10)`	`df.iloc[:9]`
`filter(df, col1 == 1, col2 == 1)`	`df.query('col1 == 1 & col2 == 1')`
`df[df$col1 == 1 & df$col2 == 1,]`	`df[(df.col1 == 1) & (df.col2 == 1)]`

There’s a guide to Pandas for R users online that goes through all the differences and shows corresponding code for pandas and R data frames.



Basics

CHICAS centre for health informatics, computing, and statistics

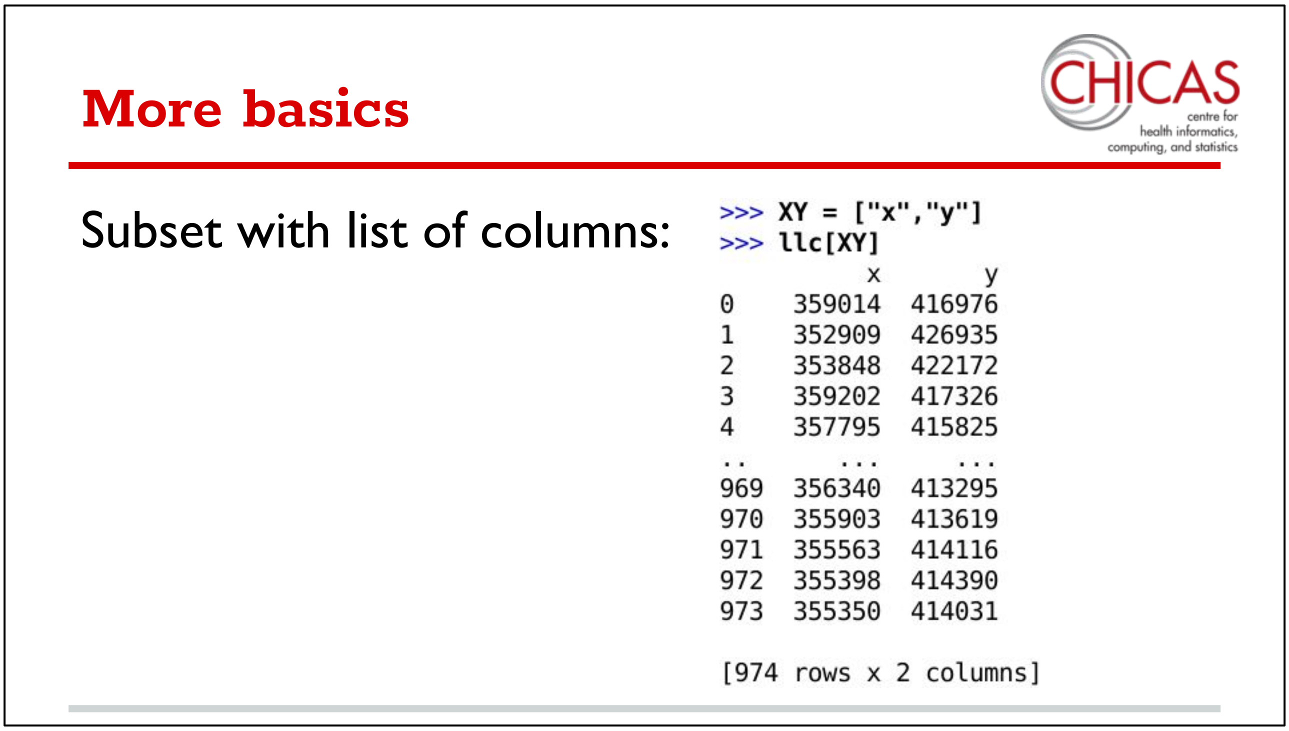
Subset with a name = a `Series`

```
>>> llc["x"]
0      359014
1      352909
2      353848
3      359202
4      357795
        ...
969    356340
970    355903
971    355563
972    355398
973    355350
Name: x, Length: 974, dtype: int64
```

Subset with an array = DataFrame

```
>>> llc[["x"]]
          x
0    359014
1    352909
2    353848
3    359202
4    357795
..      ...
969  356340
970  355903
971  355563
972  355398
973  355350

[974 rows x 1 columns]
```

For example, if you subset with a name you get the column with that name as a Series. If you subset with a list you get a data frame. Note that although this looks like R’s double-square-bracket method for getting list elements, what you are seeing is a list, constructed by the inner square brackets, inside the square brackets subsetting the data frame.



More basics

Subset with list of columns:

```
>>> XY = ["x","y"]
>>> llc[XY]
          x       y
0    359014  416976
1    352909  426935
2    353848  422172
3    359202  417326
4    357795  415825
..      ...     ...
969  356340  413295
970  355903  413619
971  355563  414116
972  355398  414390
973  355350  414031

[974 rows x 2 columns]
```

Here's that in two stages. XY is a list with two names, and when I subset I get the data frame with those columns.

Basics

Get **`Series`** with **`.column`**

```
>>> llc.x
0      359014
1      352909
2      353848
3      359202
4      357795
        ...
969    356340
970    355903
971    355563
972    355398
973    355350
Name: x, Length: 974, dtype: int64
```

Create **`Series`** with index

```
>>> llc["x0"] = llc.x - 355000
```

You can also extract a series using dotted notation. You can't create a new series like this though, you have to do it via a named subset.

Index with a slice

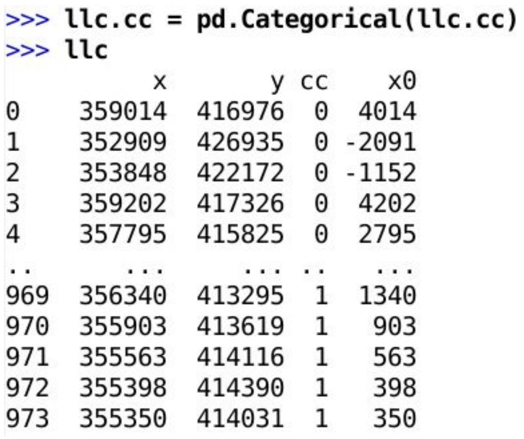
Get rows:

```
>>> llc[:3]
        x       y  cc
0  359014  416976   0
1  352909  426935   0
2  353848  422172   0
```

If you subset with a slice, you get rows.

Categorical Series

Change existing column to categorical:

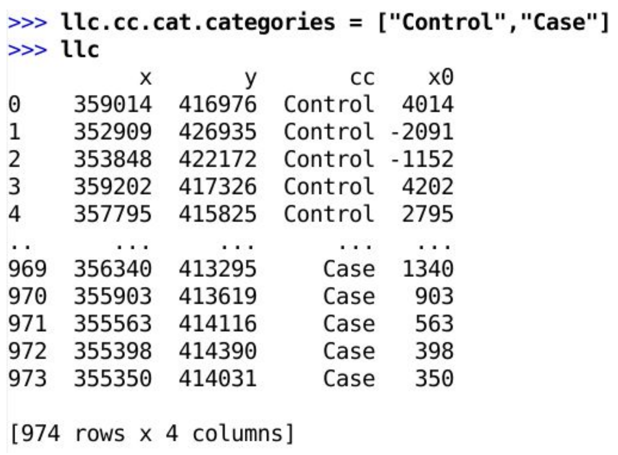


```
>>> llc.cc = pd.Categorical(llc.cc)
>>> llc
          x       y cc    x0
0    359014  416976  0  4014
1    352909  426935  0 -2091
2    353848  422172  0 -1152
3    359202  417326  0  4202
4    357795  415825  0  2795
..      ...     ... ..   ...
969  356340  413295  1  1340
970  355903  413619  1   903
971  355563  414116  1   563
972  355398  414390  1   398
973  355350  414031  1   350
```

Pandas' equivalent to R's factors is the Categorical data type. You can create a categorical series from any other series, and it ses the unique values.

Change Category Names

Modify the `.cat` attribute of the `Series`



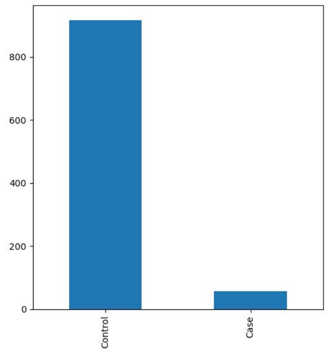
```
>>> llc.cc.cat.categories = ["Control","Case"]
>>> llc
          x       y       cc    x0
0    359014  416976  Control  4014
1    352909  426935  Control -2091
2    353848  422172  Control -1152
3    359202  417326  Control  4202
4    357795  415825  Control  2795
..      ...     ...      ...   ...
969  356340  413295     Case  1340
970  355903  413619     Case   903
971  355563  414116     Case   563
972  355398  414390     Case   398
973  355350  414031     Case   350

[974 rows x 4 columns]
```

The labels of a categorical series can be changed, this is like changing the levels of a factor in R.

Table/Plot of categories

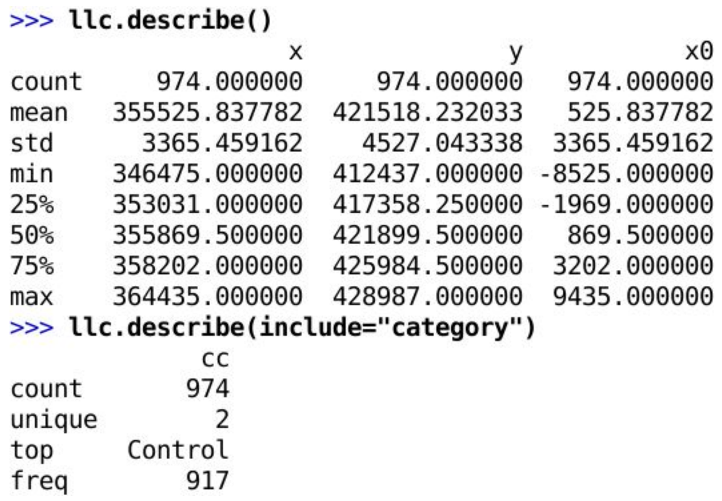
```
>>> llc.cc.value_counts()
Control    917
Case        57
Name: cc, dtype: int64
```

```
>>> llc.cc.value_counts().plot(kind="bar")
<matplotlib.axes._subplots.AxesSubplot object at 0x7ff4d391d1c0>
```



You can get basic statistics of a category and a simple barplot.

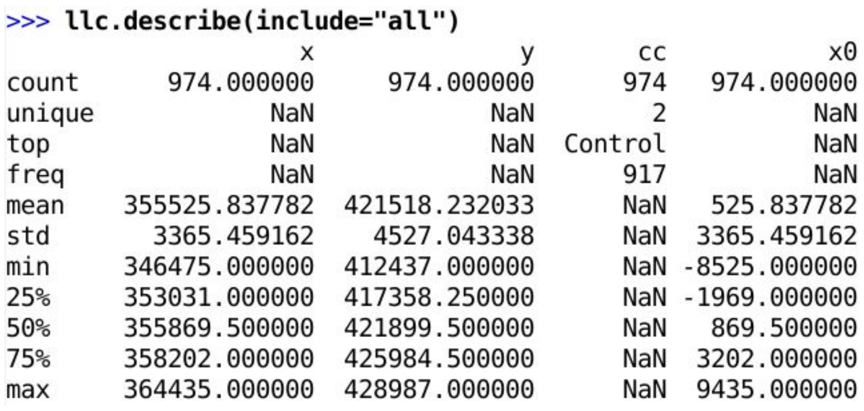
Summaries via `.describe()`



```
>>> llc.describe()
                   x              y           x0
count     974.000000     974.000000   974.000000
mean   355525.837782  421518.232033   525.837782
std      3365.459162    4527.043338  3365.459162
min    346475.000000  412437.000000 -8525.000000
25%    353031.000000  417358.250000 -1969.000000
50%    355869.500000  421899.500000   869.500000
75%    358202.000000  425984.500000  3202.000000
max    364435.000000  428987.000000  9435.000000
>>> llc.describe(include="category")
             cc
count       974
unique        2
top     Control
freq        917
```

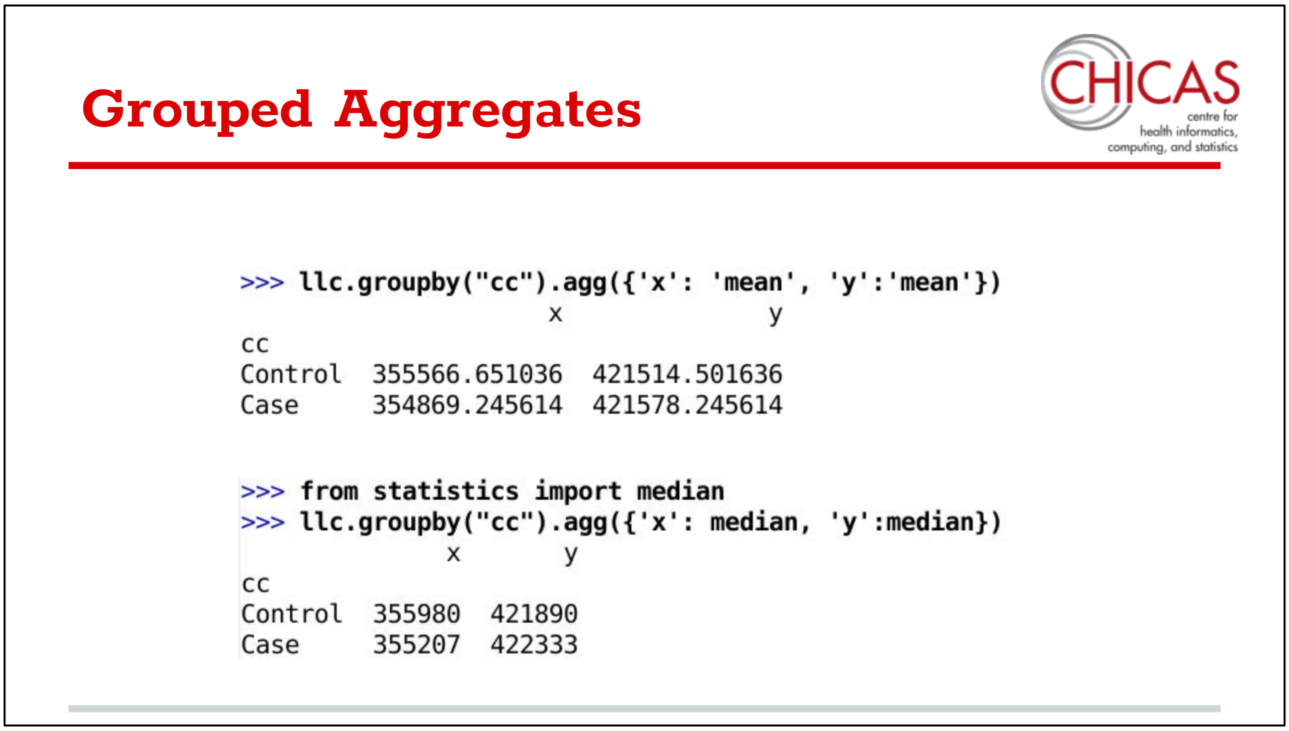
The equivalent of “summary” is the “describe” method which gives summary statistics of each numeric column. If you want info about the categorical columns you can ask for those.

All the things…



```
>>> llc.describe(include="all")
                    x              y       cc           x0
count      974.000000     974.000000      974   974.000000
unique            NaN            NaN        2          NaN
top               NaN            NaN  Control          NaN
freq              NaN            NaN      917          NaN
mean    355525.837782  421518.232033      NaN   525.837782
std       3365.459162    4527.043338      NaN  3365.459162
min     346475.000000  412437.000000      NaN -8525.000000
25%     353031.000000  417358.250000      NaN -1969.000000
50%     355869.500000  421899.500000      NaN   869.500000
75%     358202.000000  425984.500000      NaN  3202.000000
max     364435.000000  428987.000000      NaN  9435.000000
```

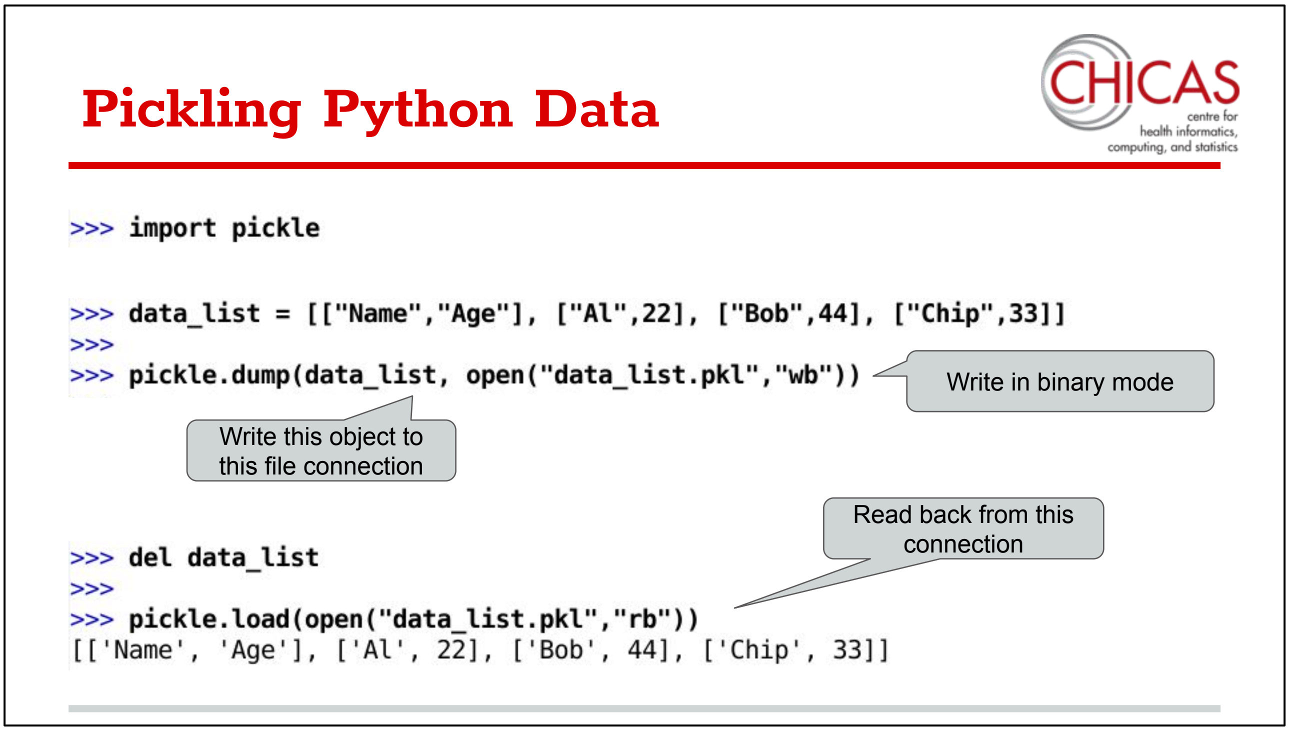
If you ask for everything you get a table with a bunch of missing elements where it doesn’t make sense to measure that for that type of column!



Grouped Aggregates

```
>>> llc.groupby("cc").agg({'x': 'mean', 'y':'mean'})
                     x              y
cc
Control  355566.651036  421514.501636
Case     354869.245614  421578.245614

>>> from statistics import median
>>> llc.groupby("cc").agg({'x': median, 'y':median})
              x       y
cc
Control  355980  421890
Case     355207  422333
```

There are methods for grouping and computing aggregates over groups too. Here I group over the CC column and compute the mean x and y values. I can use any function here, such as “median” imported from Python’s statistics module.



Pickling Python Data

CHICAS centre for health informatics, computing, and statistics

```
>>> import pickle

>>> data_list = [["Name","Age"], ["Al",22], ["Bob",44], ["Chip",33]]
>>>
>>> pickle.dump(data_list, open("data_list.pkl","wb"))
```

Write in binary mode

Write this object to this file connection

Read back from this connection

```
>>> del data_list
>>>
>>> pickle.load(open("data_list.pkl","rb"))
[['Name', 'Age'], ['Al', 22], ['Bob', 44], ['Chip', 33]]
```

Time for some more general data things in Python. Pickling. If you want to save an object in your Python session and its too complex for a CSV file, you can “pickle” it. Use the “pickle” module and dump it to a file. Make sure the file is opened in “wb” mode for writing binary data. Then you can get your object back by loading it.

Structured Data

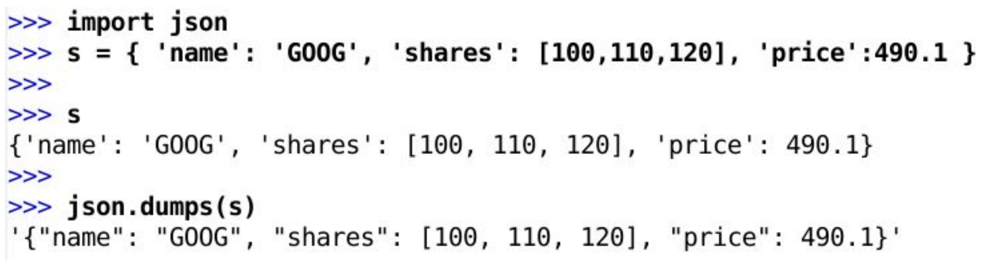
- JSON
- XML
- YAML

The pickle format is specific to Python, so if you want to interchange data with other systems, such as R for example, you need a data interchange format. There are plenty of them, I'll look at three here.

JSON

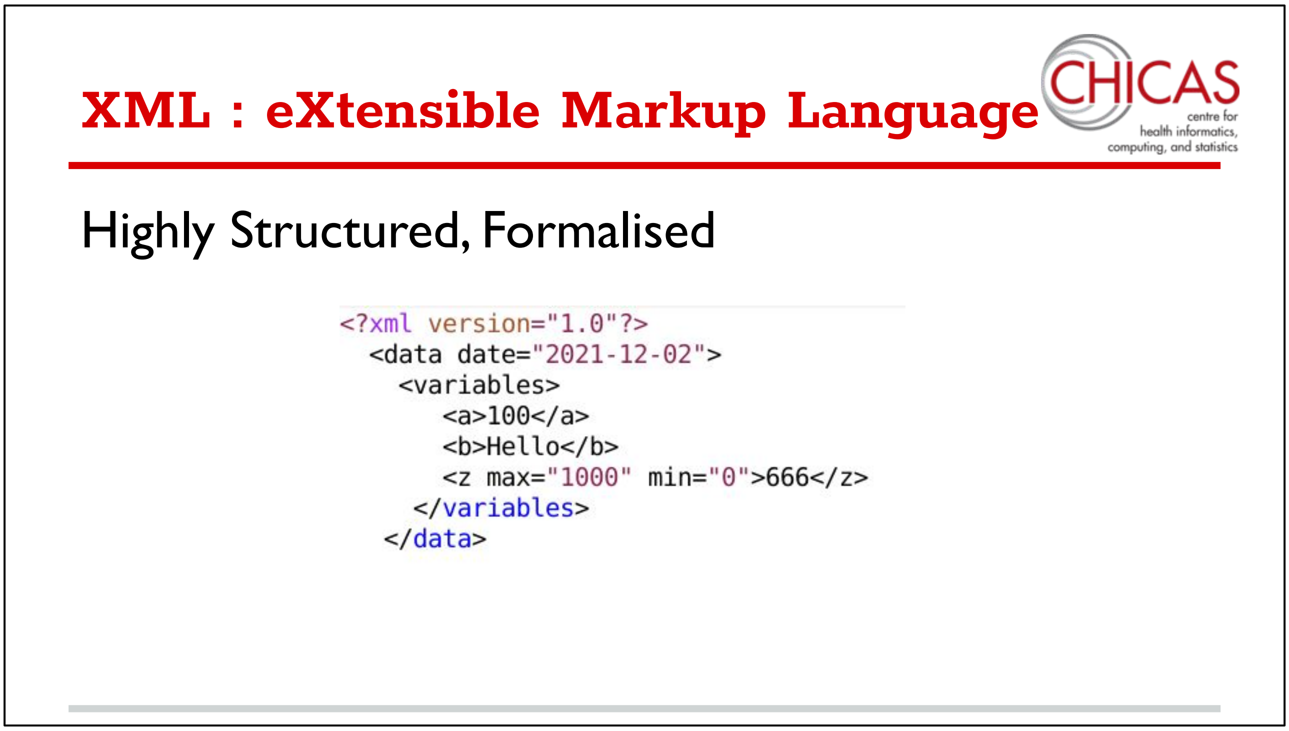
CHICAS centre for health informatics, computing, and statistics

JavaScript Object Notation



```
>>> import json
>>> s = { 'name': 'GOOG', 'shares': [100,110,120], 'price':490.1 }
>>>
>>> s
{'name': 'GOOG', 'shares': [100, 110, 120], 'price': 490.1}
>>>
>>> json.dumps(s)
'{"name": "GOOG", "shares": [100, 110, 120], "price": 490.1}'
```

JSON is a text format for structured data. There’s a built-in module for it in Python. You can get a JSON representation of a Python object as a string using the “dumps” method, which you could then write to a file. Notice how its very similar to - but not quite exactly the same as - the way Python prints its objects out. JSON is very popular for data interchange and there’s packages for most common modern programming languages.



XML : eXtensible Markup Language

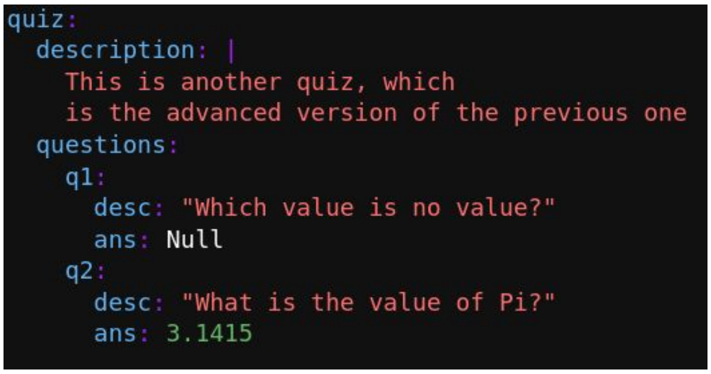
Highly Structured, Formalised

```
<?xml version="1.0"?>
  <data date="2021-12-02">
    <variables>
       <a>100</a>
       <b>Hello</b>
       <z max="1000" min="0">666</z>
     </variables>
   </data>
```

The big beast of data interchange formats is XML. This is data interchange designed by committee, and is often referred to as an “enterprise” format. Here’s a very simple example, but the XML standard specifies all sorts of complexities, including methods for transforming from one XML format to another expressed in XML. It can do anything. The problem is that its not really for human consumption.

YAML, YML

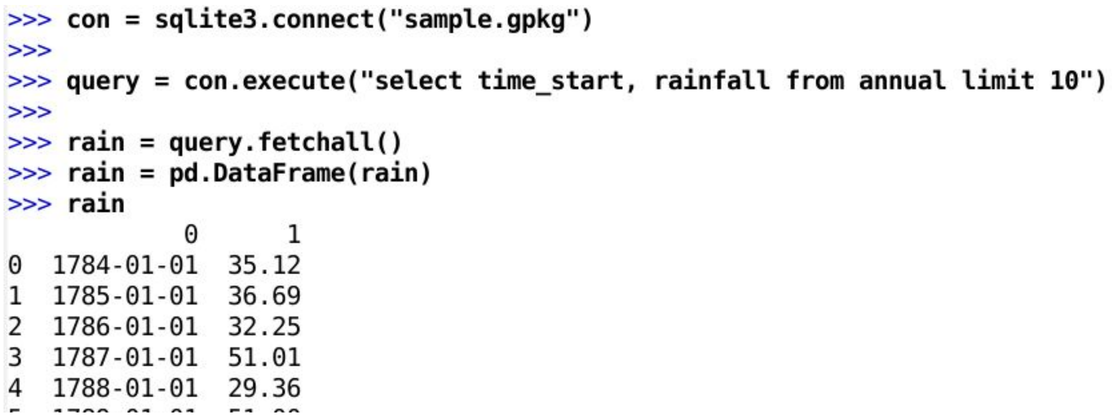
Human and Computer-readable



```yaml
quiz:
  description: |
    This is another quiz, which
    is the advanced version of the previous one
  questions:
    q1:
      desc: "Which value is no value?"
      ans: Null
    q2:
      desc: "What is the value of Pi?"
      ans: 3.1415
```

In response to XMLs complexity, in cases where less complexity is needed, YAML (yet another markup language) was developed. This removes all the angle brackets and tags for a nested structure defined by indenting, and a tag: value system. This is readable enough to be human readable and writable, but also structured enough for computers to read it. And modules exist for reading YAML for most of the common modern languages - for Python the things you need is the pyyaml package.

Databases

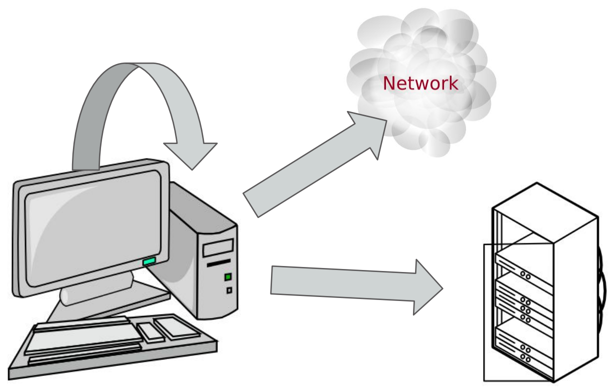


```
>>> con = sqlite3.connect("sample.gpkg")
>>>
>>> query = con.execute("select time_start, rainfall from annual limit 10")
>>>
>>> rain = query.fetchall()
>>> rain = pd.DataFrame(rain)
>>> rain
            0      1
0  1784-01-01  35.12
1  1785-01-01  36.69
2  1786-01-01  32.25
3  1787-01-01  51.01
4  1788-01-01  29.36
```

Lots of data is kept in organised databases. The way to access these is similar to accessing files - you open a connection, do stuff with the connection, and then close the connection to tidy up. Here I'm connecting to a database stored in a file called "sample.gpkg", which is an SQLite3 file. Then I can run access data by querying tables in that database and converting them to pandas data frames.

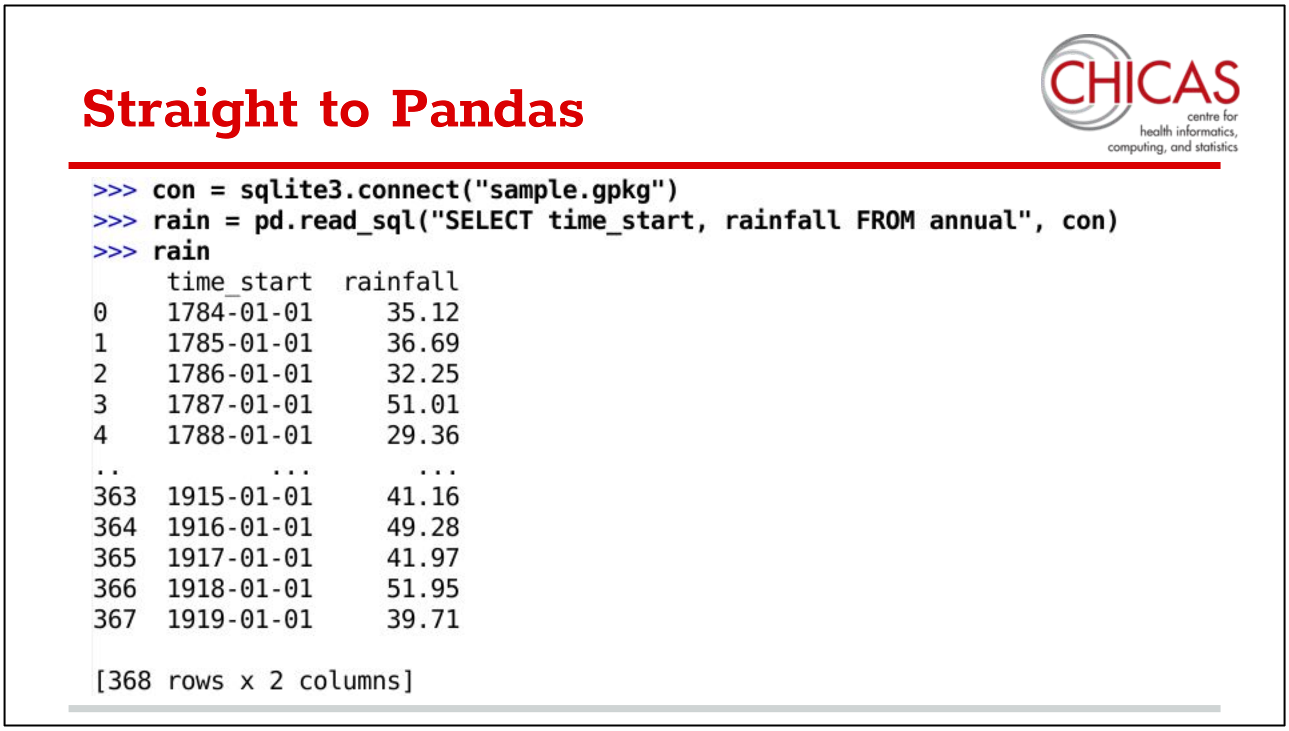
Databases

```
>>> query = con.execute("select time_start, rainfall from annual limit 10")
```



Connection could be to a database on disk, a local server, or a network cloud service.

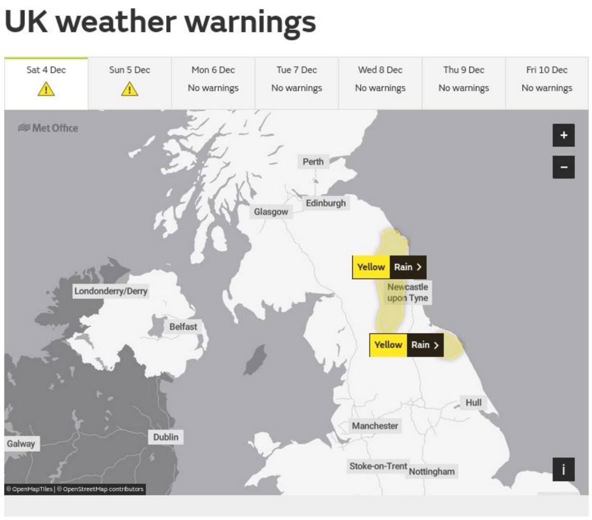
One of the advantages of the database system is that once you have the connection, your code is the same whether you have a connection to a local file, a networked local database, or some cloud data service. As long as there’s a “driver” for your data source type, you can do this.

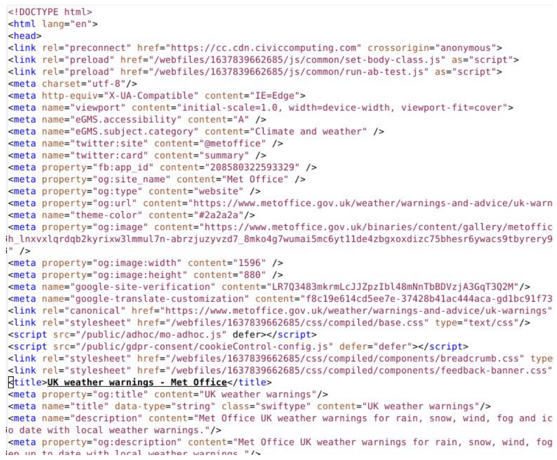

Straight to Pandas

```
>>> con = sqlite3.connect("sample.gpkg")
>>> rain = pd.read_sql("SELECT time_start, rainfall FROM annual", con)
>>> rain
     time_start  rainfall
0    1784-01-01     35.12
1    1785-01-01     36.69
2    1786-01-01     32.25
3    1787-01-01     51.01
4    1788-01-01     29.36
..          ...       ...
363  1915-01-01     41.16
364  1916-01-01     49.28
365  1917-01-01     41.97
366  1918-01-01     51.95
367  1919-01-01     39.71

[368 rows x 2 columns]
```

And you can go straight to pandas data frames with some pandas methods once you have a connection.

Web Scraping



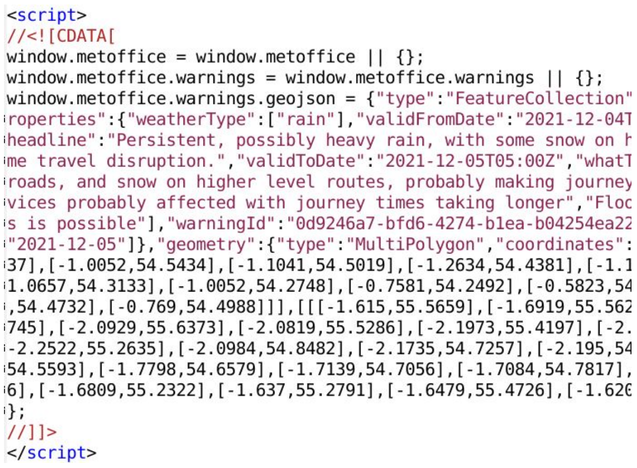


```
<!DOCTYPE html>
<html lang="en">
<head>
<link rel="preconnect" href="https://cc.cdn.civiccomputing.com" crossorigin="anonymous">
<link rel="preload" href="/webfiles/1637839662685/js/common/set-body-class.js" as="script">
<link rel="preload" href="/webfiles/1637839662685/js/common/run-ab-test.js" as="script">
<meta charset="utf-8"/>
<meta http-equiv="X-UA-Compatible" content="IE=Edge">
<meta name="viewport" content="initial-scale=1.0, width=device-width, viewport-fit=cover">
<meta name="eGMS.accessibility" content="A" />
<meta name="eGMS.subject.category" content="Climate and weather" />
<meta name="twitter:site" content="@metoffice" />
<meta name="twitter:card" content="summary" />
<meta property="fb:app_id" content="208580322593329" />
<meta property="og:site_name" content="Met Office" />
<meta property="og:type" content="website" />
<meta property="og:url" content="https://www.metoffice.gov.uk/weather/warnings-and-advice/uk-warn
<meta name="theme-color" content="#2a2a2a"/>
<meta property="og:image" content="https://www.metoffice.gov.uk/binaries/content/gallery/metoffic
h_lnxvxlqrdqb2kyrixw3lmmul7n-abrzjuzyvzd7_8mko4g7wumai5mc6yt11de4zbgxoxdizc75bhesr6ywacs9tbyrery9
" />
<meta property="og:image:width" content="1596" />
<meta property="og:image:height" content="880" />
<meta name="google-site-verification" content="LR7Q3483mkrmLcJJZpzIbl48mNnTbBDVzjA3GqT3Q2M"/>
<meta name="google-translate-customization" content="f8c19e614cd5ee7e-37428b41ac444aca-gd1bc91f73
<link rel="canonical" href="https://www.metoffice.gov.uk/weather/warnings-and-advice/uk-warnings"
<link rel="stylesheet" href="/webfiles/1637839662685/css/compiled/base.css" type="text/css"/>
<script src="/public/adhoc/mo-adhoc.js" defer></script>
<script src="/public/gdpr-consent/cookieControl-config.js" defer="defer"></script>
<link rel="stylesheet" href="/webfiles/1637839662685/css/compiled/components/breadcrumb.css" type
<link rel="stylesheet" href="/webfiles/1637839662685/css/compiled/components/feedback-banner.css"
<title>UK weather warnings - Met Office</title>
<meta property="og:title" content="UK weather warnings"/>
<meta name="title" data-type="string" class="swiftype" content="UK weather warnings"/>
<meta name="description" content="Met Office UK weather warnings for rain, snow, wind, fog and ic
o date with local weather warnings."/>
<meta property="og:description" content="Met Office UK weather warnings for rain, snow, wind, fog
ep up to date with local weather warnings."/>
```

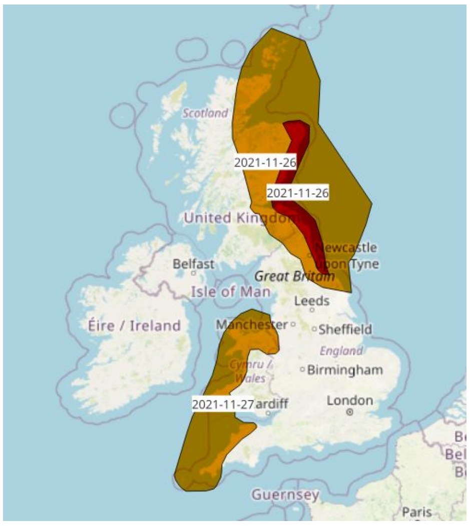
Another common source of data, some of it informal and of possible dubious authenticity or legality, is web pages. Pretty much anything you see on a web page is on your computer (otherwise how would you see it?) and so could potentially be converted into useful data. This process is known as “web scraping”. This is the Met Office UK Weather Warnings page and shows here a couple of rain warnings. I can look at the web page source and I know that somewhere in here must be the data that defines those warnings and the warning text and colour and so on.

Scrape scrape…

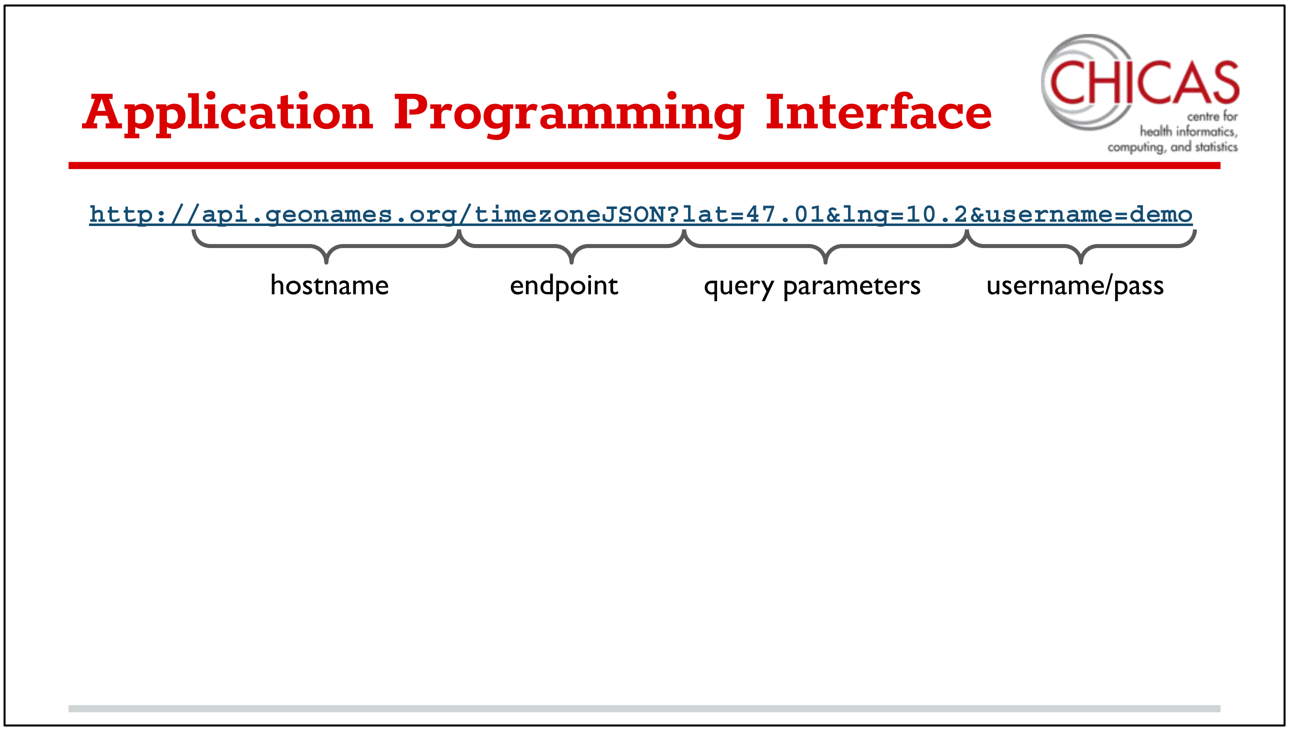
Line 1731



```
<script>
//<![CDATA[
window.metoffice = window.metoffice || {};
window.metoffice.warnings = window.metoffice.warnings || {};
window.metoffice.warnings.geojson = {"type":"FeatureCollection"
roperties":{"weatherType":["rain"],"validFromDate":"2021-12-04T
headline":"Persistent, possibly heavy rain, with some snow on h
me travel disruption.","validToDate":"2021-12-05T05:00Z","whatT
roads, and snow on higher level routes, probably making journey
vices probably affected with journey times taking longer","Floc
s is possible"],"warningId":"0d9246a7-bfd6-4274-b1ea-b04254ea22
"2021-12-05"]},"geometry":{"type":"MultiPolygon","coordinates":
37],[-1.0052,54.5434],[-1.1041,54.5019],[-1.2634,54.4381],[-1.1
1.0657,54.3133],[-1.0052,54.2748],[-0.7581,54.2492],[-0.5823,54
,54.4732],[-0.769,54.4988]]],[[[-1.615,55.5659],[-1.6919,55.562
745],[-2.0929,55.6373],[-2.0819,55.5286],[-2.1973,55.4197],[-2.
-2.2522,55.2635],[-2.0984,54.8482],[-2.1735,54.7257],[-2.195,54
54.5593],[-1.7798,54.6579],[-1.7139,54.7056],[-1.7084,54.7817],
6],[-1.6809,55.2322],[-1.637,55.2791],[-1.6479,55.4726],[-1.62
};
//]]>
</script>
```



And I find it at line 1731 inside a <script> tag as part of a “geoJSON” object. Its then not too hard for me to extract that geoJSON and load it into my favourite geographic information system and map it (this is a different set of warnings from the previous page).

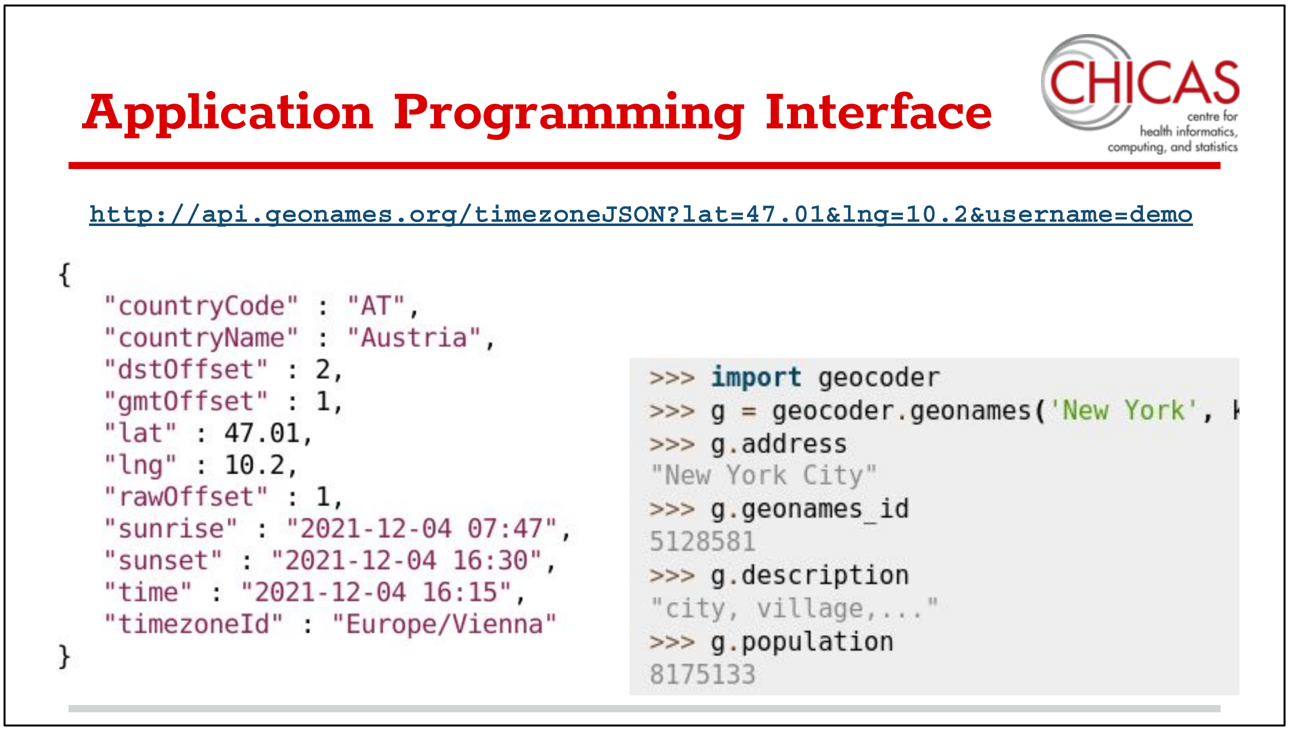


Application Programming Interface

CHICAS centre for health informatics, computing, and statistics

$$\texttt{http://}\underbrace{\texttt{api.geonames.org}}_{\text{hostname}}\underbrace{\texttt{/timezoneJSON}}_{\text{endpoint}}\underbrace{\texttt{?lat=47.01\&lng=10.2}}_{\text{query parameters}}\underbrace{\texttt{\&username=demo}}_{\text{username/pass}}$$

To save people having to scrape data from web pages and encourage a more formal approach to getting useful data, some companies expose an API for their data service. This is a URL that will get you some timezone info for this lat-long coordinate from a service called “Geonames”. If you go to this URL in a browser (and you may need to register a username first and replace it here) you get back some JSON data:



Application Programming Interface

CHICAS centre for health informatics, computing, and statistics

http://api.geonames.org/timezoneJSON?lat=47.01&lng=10.2&username=demo

```
{
   "countryCode" : "AT",
   "countryName" : "Austria",
   "dstOffset" : 2,
   "gmtOffset" : 1,
   "lat" : 47.01,
   "lng" : 10.2,
   "rawOffset" : 1,
   "sunrise" : "2021-12-04 07:47",
   "sunset" : "2021-12-04 16:30",
   "time" : "2021-12-04 16:15",
   "timezoneId" : "Europe/Vienna"
}
```

```
>>> import geocoder
>>> g = geocoder.geonames('New York', k
>>> g.address
"New York City"
>>> g.geonames_id
5128581
>>> g.description
"city, village,..."
>>> g.population
8175133
```

And that data looks like this - the point is in Austria and the timezone and current time is given, together with some other data. Now you could write some Python code to do all this - there's a builtin module for doing web requests in Python - but for many popular services a quick search on PyPI will find that someone has already written a complete module for you. Here's the "geocoder" module with has a method for calling the "geonames" web services and getting info back easily.

Summary

- Data is? Data are?
- Reading/Writing files in Python
- Pandas for Data Frames
- Getting Data from Sources

I've not answered the important question - is "data" singular or plural? But I've outlined basic data read/write operations in Python, and given an outline to Pandas for data frame work, and shown a bit about sources of data that a modern data scientist is going to encounter. Although people will still email you Excel spreadsheets and data in tables in PDFs…