Intro to Python

CSV Files

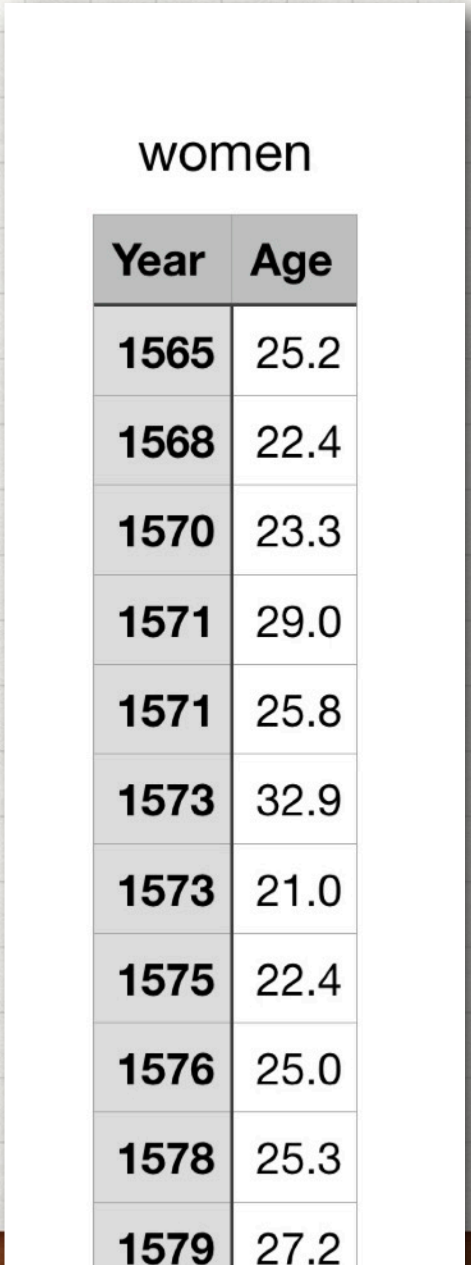
Special Files

Comma-Separated Values (CSV)

SPREADSHEET DATA AS CSV

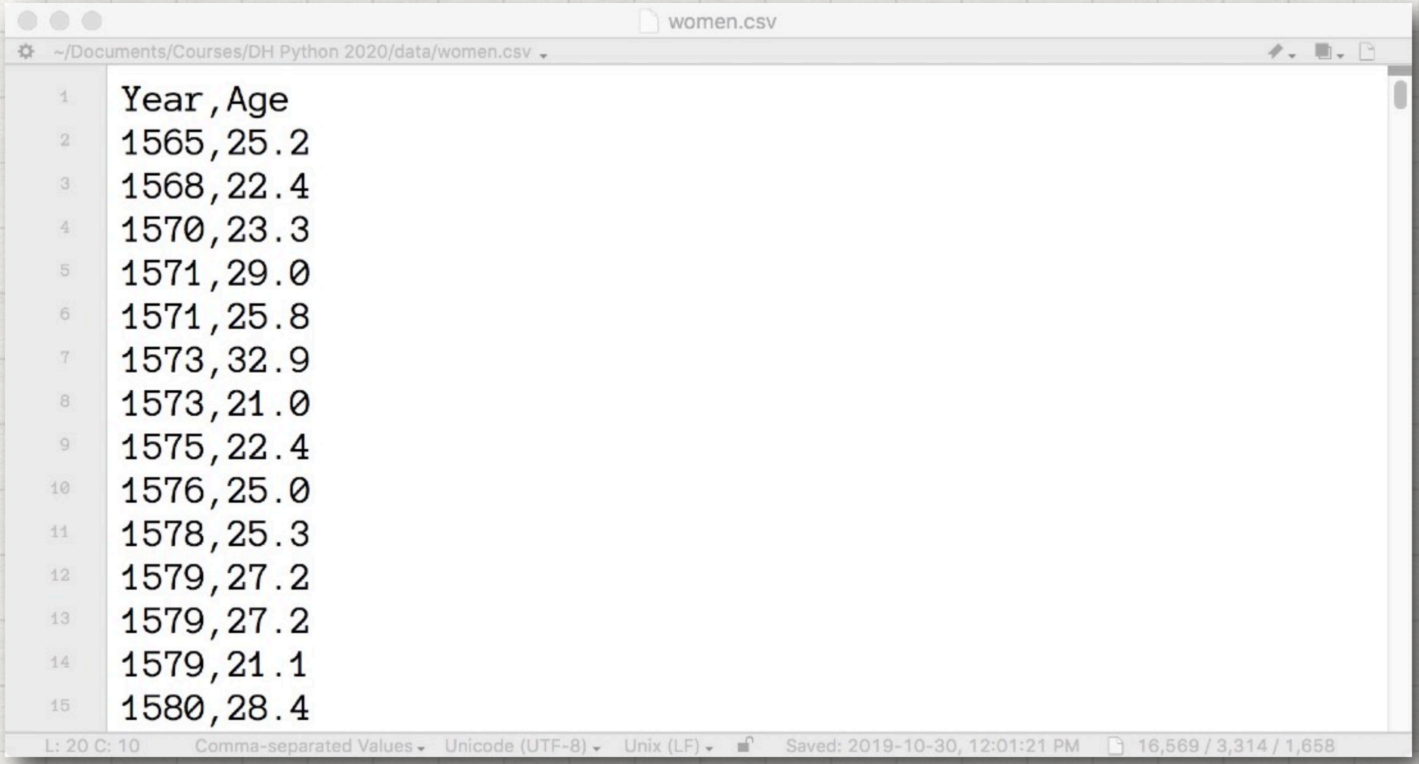
Perhaps the most common data-collection tool is the ordinary spreadsheet. Although spreadsheets often have proprietary (that is, secret) internal data structures and usually contain annoying formatting information (fonts, line spacings, etc.), all spreadsheets can export to a plain-text format called `comma-separated values` ("CSV") files.

as a spreadsheet



women

Year	Age
1565	25.2
1568	22.4
1570	23.3
1571	29.0
1571	25.8
1573	32.9
1573	21.0
1575	22.4
1576	25.0
1578	25.3
1579	27.2



women.csv

~/Documents/Courses/DH Python 2020/data/women.csv

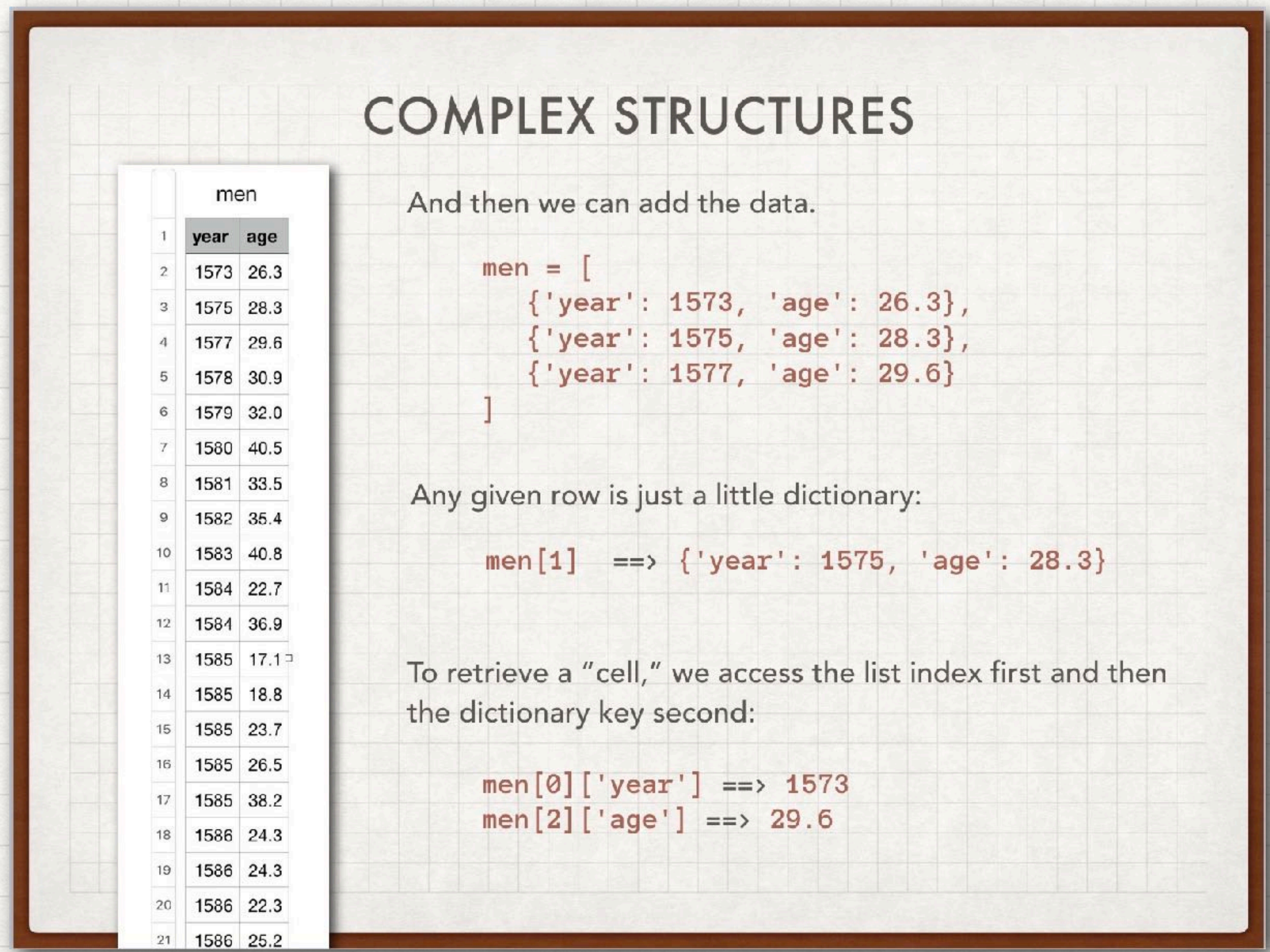
```
Year,Age
1565,25.2
1568,22.4
1570,23.3
1571,29.0
1571,25.8
1573,32.9
1573,21.0
1575,22.4
1576,25.0
1578,25.3
1579,27.2
1579,27.2
1579,21.1
1580,28.4
```

L: 20 C: 10 Comma-separated Values Unicode (UTF-8) Unix (LF) Saved: 2019-10-30, 12:01:21 PM 16,569 / 3,314 / 1,658

as a CSV

2-D DATA STRUCTURES

Just as a reminder, my CSV data will probably become the basis for a more complex data structure than simple lists or even simple dictionaries. Here’s what we saw in an earlier lecture:



COMPLEX STRUCTURES

	men	
1	**year**	**age**
2	1573	26.3
3	1575	28.3
4	1577	29.6
5	1578	30.9
6	1579	32.0
7	1580	40.5
8	1581	33.5
9	1582	35.4
10	1583	40.8
11	1584	22.7
12	1584	36.9
13	1585	17.1
14	1585	18.8
15	1585	23.7
16	1585	26.5
17	1585	38.2
18	1586	24.3
19	1586	24.3
20	1586	22.3
21	1586	25.2

And then we can add the data.

```
men = [
   {'year': 1573, 'age': 26.3},
   {'year': 1575, 'age': 28.3},
   {'year': 1577, 'age': 29.6}
]
```

Any given row is just a little dictionary:

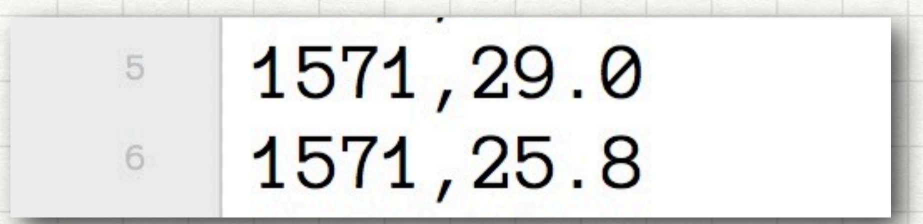
```
men[1]  ==> {'year': 1575, 'age': 28.3}
```

To retrieve a “cell,” we access the list index first and then the dictionary key second:

```
men[0]['year'] ==> 1573
men[2]['age'] ==> 29.6
```

I’M HAVING ISSUES, PART 1

Given what we’ve learned about files, it would be logical to assume that each line will come in as one string (true!)! So we should simply be able to use Python’s `split()` method as a way to separate the columns. Makes sense at first.

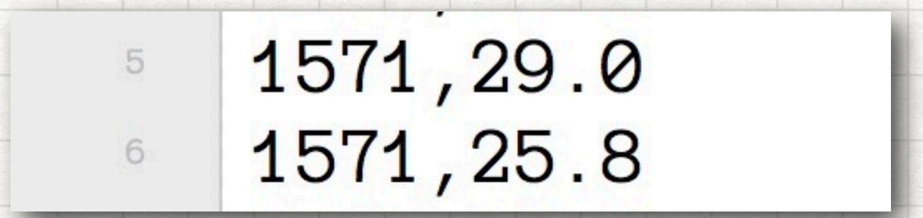


```
1571,29.0
1571,25.8
```

```
line.split(',')
```

I'M HAVING ISSUES, PART 1

Given what we've learned about files, it would be logical to assume that each line will come in as one string (true!)! So we should simply be able to use Python's `split()` method as a way to separate the columns. Makes sense at first.



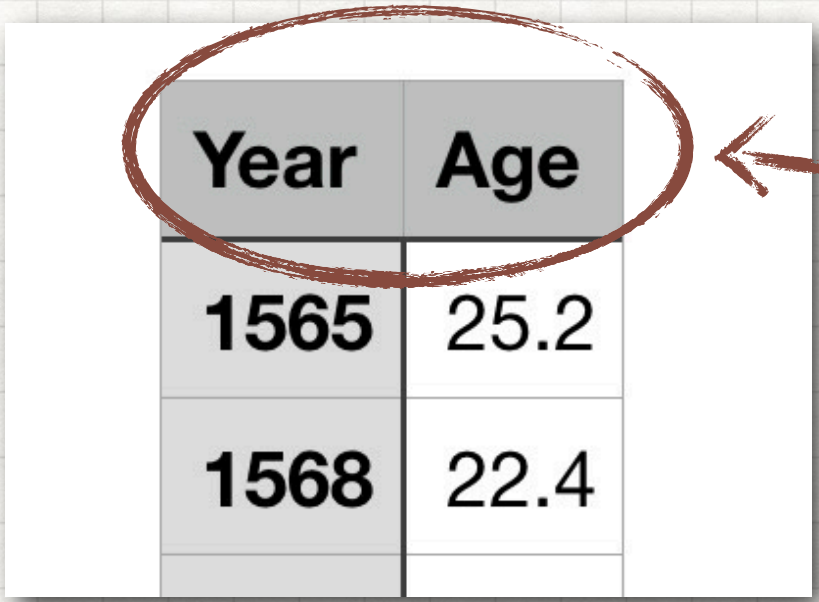
```
1571,29.0
1571,25.8
```

```
line = '1571,29.0'
print( line.split(',') )
['1571', '29.0']
```

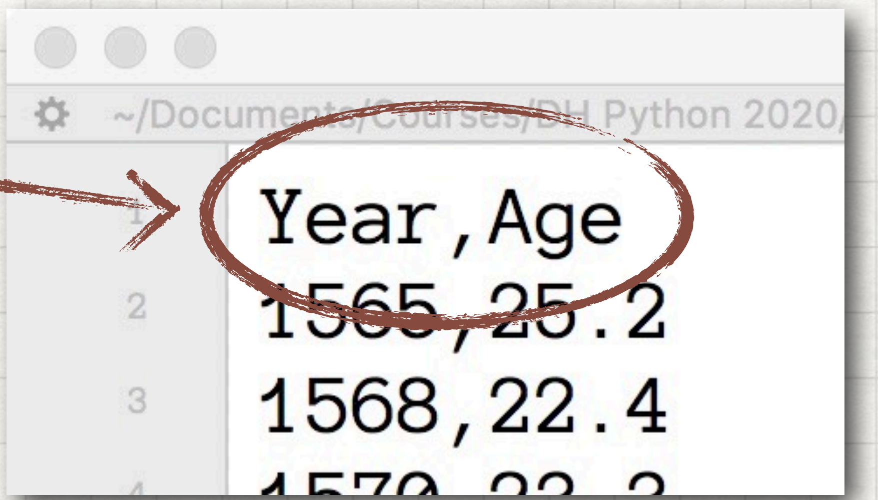
Umm, why are these strings?!

I’M HAVING ISSUES, PART 2

There’s also a pesky header here.



Year	Age
1565	25.2
1568	22.4



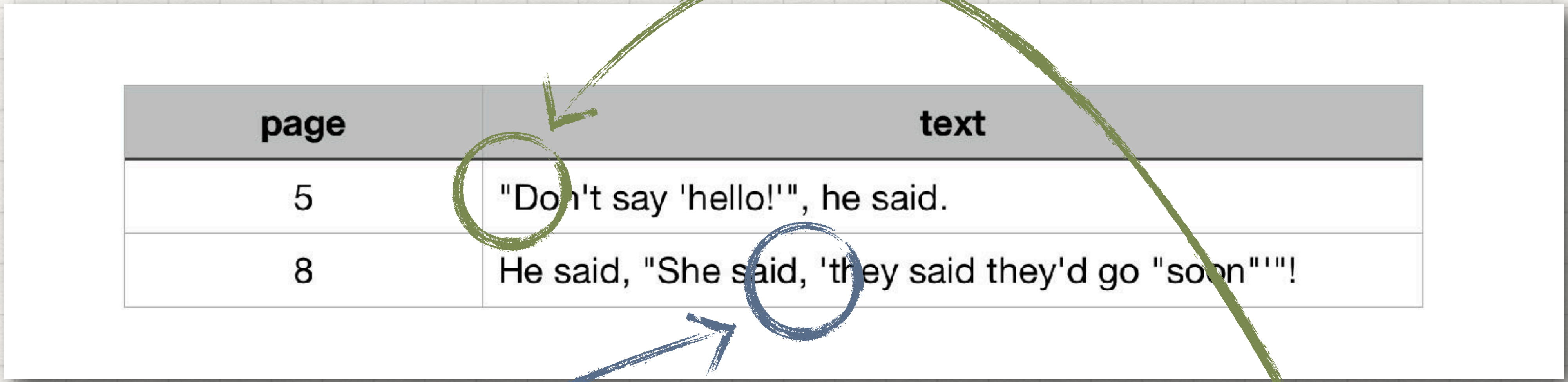
~/Documents/Courses/DH Python 2020/

```
Year,Age
1565,25.2
1568,22.4
```

So it seems we’ll either need to skip the first row — or perhaps each row could become a dictionary. Either might be a good choice. Can I tell Python which I prefer? *(Answer: yes!)*

I'M HAVING ISSUES, PART 3

What happens if my spreadsheet data is complicated? Say, what if it has commas and quotation marks inside it?



page	text
5	"Don't say 'hello!'", he said.
8	He said, "She said, 'they said they'd go "soon"'"!

There are commas inside the cells...

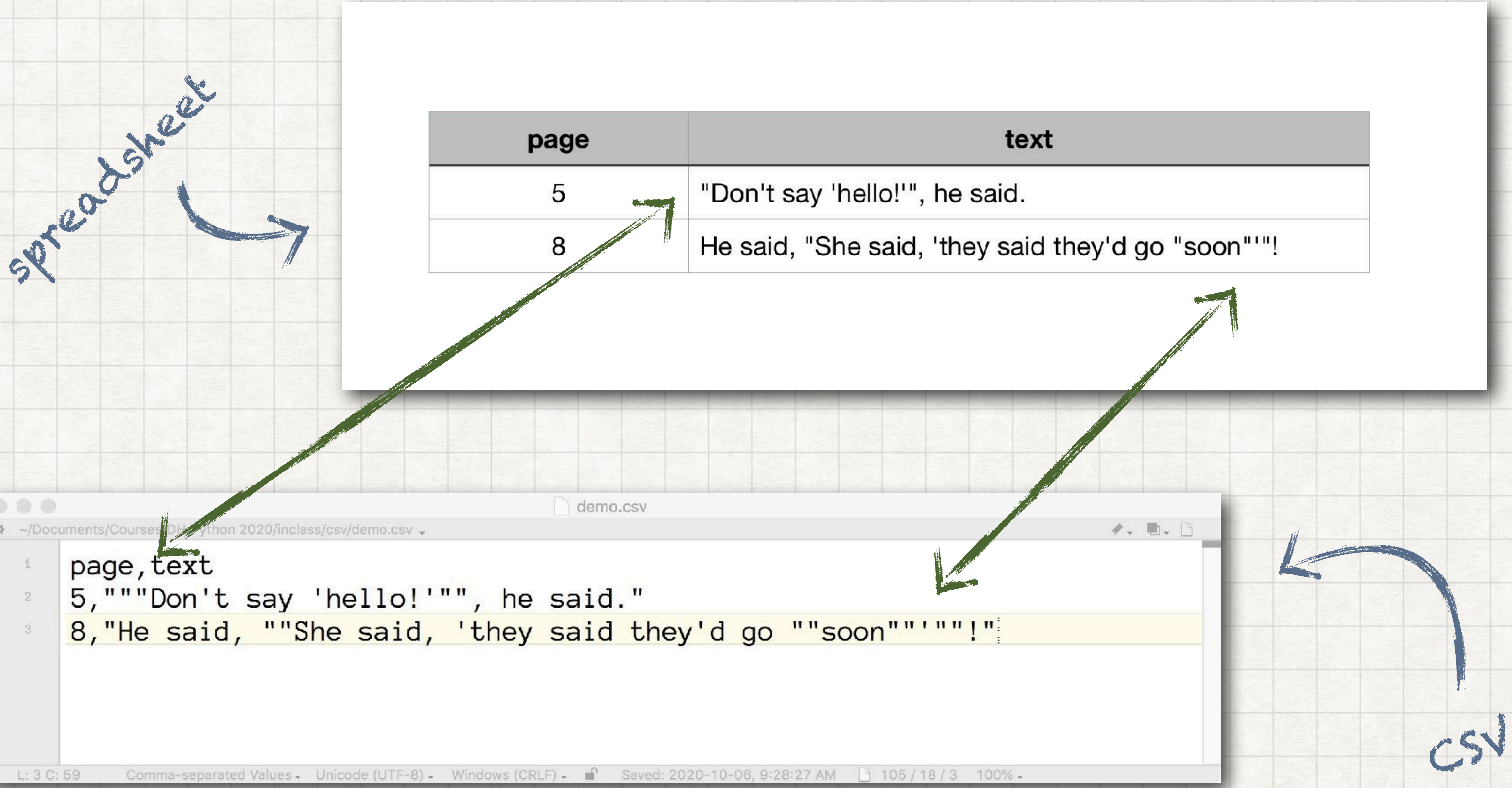
... which means CSV values need to be wrapped in quotes!

But the cells also have quotes...

... which means even more quotes!

I'M HAVING ISSUES, PART 3

CSV might sound simple, but the rules get really complicated really fast. Just look at what happens to this CSV, for example:



page	text
5	"Don't say 'hello!'", he said.
8	He said, "She said, 'they said they'd go "soon"'"!

```
page,text
5,"""Don't say 'hello!'"", he said."
8,"He said, ""She said, 'they said they'd go ""soon""'""!"
```

```
L: 3 C: 59    Comma-separated Values    Unicode (UTF-8)    Windows (CRLF)    Saved: 2020-10-06, 9:28:27 AM    105 / 18 / 3    100%
```

PYTHON HAS A SOLUTION

Python has a special object that reads and writes CSV files. It knows all the rules so that we don't have to. It's an external module that we need to load before we can use:

```
import csv
```

Next, we create the CSV object and give it a *file object* that has been created with an `open()` call.

```
csv.reader( ordinary_file_object )
csv.writer( ordinary_file_object )
```

The CSV Reader or CSV Writer object becomes your new iterator for any subsequent file operations:

```
with open('women.csv') as file:
    csvreader = csv.reader(file)
    rows = [row for row in csvreader]
```

HEADER ROWS

CSV files commonly have header rows. I have a choice in handling them:

1) **Just skip it with `next()`.** Because the file object is an iterator, I can tell it to advance to the next row by calling **`next()`**:

```
with open('women.csv') as file:
    csvreader = csv.reader(file)
    header = next(csvreader)
    rows = [row for row in csvreader]
```

2) **Use DictReader.** Although we can extract the header and build a list of dictionaries by hand, the **`cvs`** module can do it for us automatically with a DictReader object.

```
with open('women.csv') as file:
    csvreader = csv.DictReader(file)
    rows = [row for row in csvreader]
```

CASTING DATA TYPES

Because we import all the data as a string, splitting isn't always enough. If I need these to be numbers, I can take the time to *cast* the columns to their proper types:

```
with open('women.csv') as file:
    csvreader = csv.reader(file)
    # pull off the first line -- they're the headers.
    #   header[0] => the first column's header
    #   header[1] => the next column's header
    header = next(csvreader)

    # now collect all the other parts:
    data = [{header[0]: int(line[0]), header[1]: float(line[1])} for line in csvreader]
```

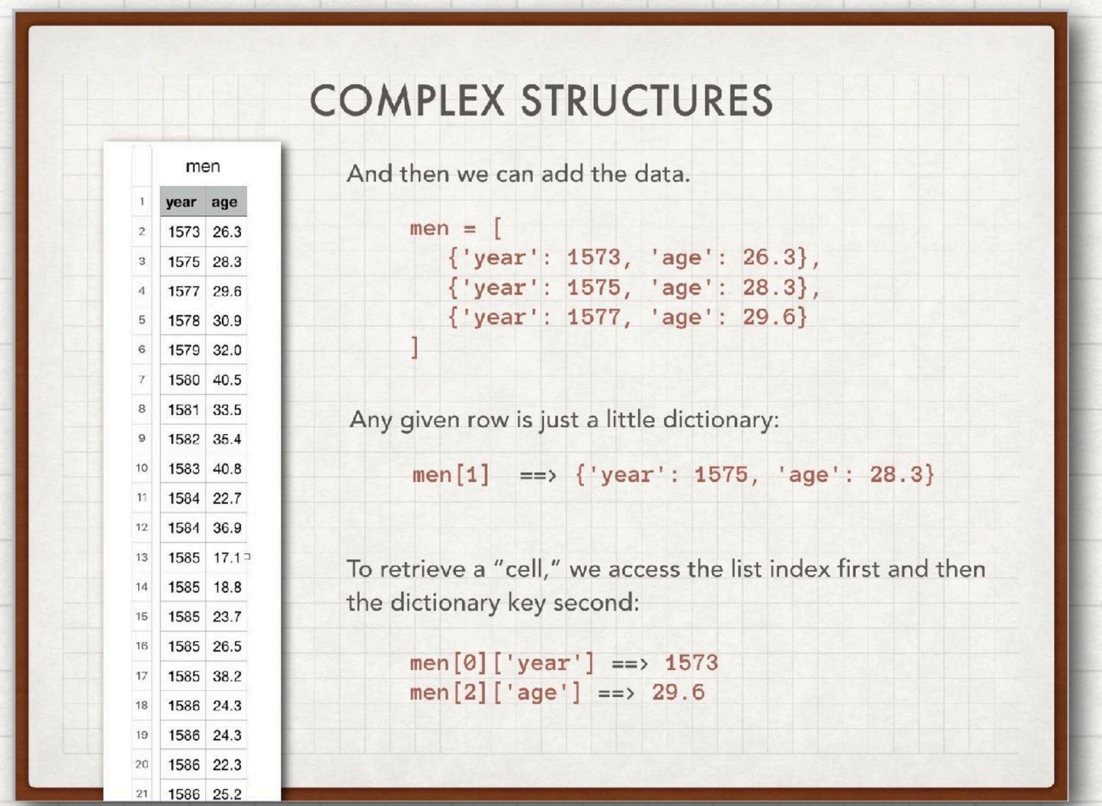
```
['1568', '22.4']
```

```
>>> data[1]
{'Year' 1568, 'Age' 22.4}
```

Look! Numbers!

HEADER ROWS

2) **Use DictReader.** We saw in an earlier slideshow that spreadsheets are two-dimensional data structures. We built one of those by hand, but Python's DictReader can build them for us automatically.



COMPLEX STRUCTURES

men

1	year	age
2	1573	26.3
3	1575	28.3
4	1577	29.6
5	1578	30.9
6	1579	32.0
7	1580	40.5
8	1581	33.5
9	1582	35.4
10	1583	40.8
11	1584	22.7
12	1584	36.9
13	1585	17.1
14	1585	18.8
15	1585	23.7
16	1585	26.5
17	1585	38.2
18	1586	24.3
19	1586	24.3
20	1586	22.3
21	1586	25.2

And then we can add the data.

```
men = [
   {'year': 1573, 'age': 26.3},
   {'year': 1575, 'age': 28.3},
   {'year': 1577, 'age': 29.6}
]
```

Any given row is just a little dictionary:

```
men[1]  ==> {'year': 1575, 'age': 28.3}
```

To retrieve a "cell," we access the list index first and then the dictionary key second:

```
men[0]['year'] ==> 1573
men[2]['age'] ==> 29.6
```

```
with open('women.csv') as file:
    csvreader = csv.DictReader(file)
    rows = [row for row in csvreader]
```

```
rows[1]['Year']
'1568'
```

Note: still a string!

Special Files

Comma-Separated Values (CSV)