Lecture 1: Introduction to Pandas

Contents

DSCI 511

Python Programming for Data Science

Lecture learning objectives

- Create a DataFrame from a text file using Pandas `pd.read_csv()` and `pd.read_excel()`
- Examine a DataFrame with `.head()`, `.tail()`, `.describe()` and `.shape`

- Access values from a DataFrame using `[]`, `.loc[]`, and `.iloc[]`
- Apply mathematical functions to columns in a DataFrame
- Create new columns in a DataFrame by performing operations on existing columns
- Remove columns from a DataFrame
- Sort a DataFrame using `.sort_values()`
- Write the contents of a DataFrame to file using `.to_csv()`

Introduction to Pandas

- The most popular Python library for tabular data structures
- You can think of Pandas as an extremely powerful version of Excel (but free and with a lot more features!)
- The only tool you'll need for many (most?) data wrangling tasks

Source: giphy.com

To install Pandas, run this code. You only need to do this once. Ideally this is done in the terminal, not a notebook.

```
$ conda install pandas
```

To use Pandas in your code, you must *import* it. It's common to import and rename Pandas to simply "pd".

```
import pandas as pd
```

When you run this code, Python does a lot of stuff behind the scenes, and we won't discuss details here. What's important is that you can now type `pd.` followed by the name of something from the Pandas library to use it. Pandas is a very large library, with too many functions to discuss in this class. We will focus on just a few parts of Pandas in this course, that we think are going to be most relevant/useful in a data science career. The official documentation for Pandas explains everything that you can do with it, and you should bookmark it now: https://pandas.pydata.org/docs/index.html

The first thing we'll learn to do with Pandas is open a file. We'll start with data from the Internet Movie Database (IMDB). We use the `.read_csv()` function to open files. Run the next cell to open the file `imdb.csv`, and save it as a variable called `imdb`.

```
imdb = pd.read_csv('data/imdb.csv')
```

CSV stands for 'comma-separated values, and it represents a table in a text file. Each row in the file is a row in a table, and columns in each row are separated by commas. Pandas can read from a variety of different file types, but CSV is one of the most common formats in data science and machine learning. Try opening this file in text editor to see what it looks like.

Notice that the name of the file is written bewteen quotation marks. This is how we tell the difference between code and text in Python (and virtually all other programming languages). In technical terms, text is referred to as a *string*, which is one of the fundamental types of information used in programming.

Pandas transforms the information from the file into a type of object called a `DataFrame`, which you can think of like an Excel spreadsheet. DataFrames are your best friend in this course, and you will use them for practically everything.

The concepts of *objects* and *types* are fundamental to programming, and each type has different uses. During this course you'll deal with a variety of types representing different formats (numbers, text, sequences, tables, mappings, etc.). We will discuss types as they come up. If you have a background in programming, you can read about Python's built-in types here: https://docs.python.org/3/reference/index.html

The DataFrame type has many useful *methods* or *functions*. You can think of a function as one or more lines of code which perform some computation(s), and then return a result to you. Most functions allow you to supply *arguments*, which are like options you can set, to modify how the function operates. DataFrames have a very large number of methods and we will only cover some of the more commonly used ones in this class. There is a full list available in the official documentation.

The next few cells have code for quickly inspecting your data:

```
imdb.head(8) #Show the first 8 rows, 8 is an argument
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont
1	The Godfather	1972	A	175 min	Crime, Drama	9.2	100.0	Francis Ford Coppola
2	The Dark Knight	2008	UA	152 min	Action, Crime, Drama	9.0	84.0	Christopher Nolan
3	The Godfather: Part II	1974	A	202 min	Crime, Drama	9.0	90.0	Francis Ford Coppola
4	12 Angry Men	1957	U	96 min	Crime, Drama	9.0	96.0	Sidney Lumet
5	The Lord of the Rings: The Return of the King	2003	U	201 min	Action, Adventure, Drama	8.9	94.0	Peter Jackson
6	Pulp Fiction	1994	A	154 min	Crime, Drama	8.9	94.0	Quentin Tarantino
7	Schindler's List	1993	A	195 min	Biography, Drama, History	8.9	94.0	Steven Spielberg

```
imdb.tail(5) #Show the last 5 rows, 5 is an argument
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director
995	Breakfast at Tiffany's	1961	A	115 min	Comedy, Drama, Romance	7.6	76.0	Blake Edwards
996	Giant	1956	G	201 min	Drama, Western	7.6	84.0	George Stevens
997	From Here to Eternity	1953	Passed	118 min	Drama, Romance, War	7.6	85.0	Fred Zinnemann
998	Lifeboat	1944	NaN	97 min	Drama, War	7.6	78.0	Alfred Hitchcock
999	The 39 Steps	1935	NaN	86 min	Crime, Mystery, Thriller	7.6	93.0	Alfred Hitchcock

```
imdb.describe() #Get basic stats, only works with numeric columns, there are no arguments
```

	Released_Year	IMDB_Rating	Meta_score	No_of_Votes	Gross
count	1000.000000	1000.000000	843.000000	1.000000e+03	8.310000e+02
mean	1992.221000	7.949300	77.971530	2.736929e+05	6.803475e+07
std	39.746924	0.275491	12.376099	3.273727e+05	1.097500e+08
min	1920.000000	7.600000	28.000000	2.508800e+04	1.305000e+03
25%	1976.000000	7.700000	70.000000	5.552625e+04	3.253559e+06
50%	1999.000000	7.900000	79.000000	1.385485e+05	2.353089e+07
75%	2009.000000	8.100000	87.000000	3.741612e+05	8.075089e+07
max	3010.000000	9.300000	100.000000	2.343110e+06	9.366622e+08

Notice that the numbers above do not have quotation marks. This indicates that they should be treated as actual numbers, and not as strings of text. In Python, the number `8` and the string `'8'` are not equivalent. Whole numbers are another fundamental type in programming, technically referred to as *integers* or just *ints*. Decimals numbers e.g. (`10.4`) are treated as a different type, called a *float*.

In addition to functions, DataFrames also have *attributes*, which are just values that you can look up, and which don't require any special computation. For example, `.shape` tells you the number of rows and columns. Attributes are not written with parenthesis.

```
imdb.shape #No brackets!
```

```
(1000, 14)
```

In addition to comma separated files, it's also common to store tables as tab separated files (TSV). The extra space makes this format a little easier for people to read. Sometimes it is necessary to use tabs because you are storing data that

contains commas, such as sentences of natural language text. In this case, you don't want the commas treated as column separators.

The `.read_csv()` method, despite its name, can also be used to read TSV files by adding a `sep` argument. The next cells opens a file with information about villagers in the video game Stardew Valley. Each villager has likes and dislikes, written out as a comma-separated list, making the CSV format impractical.

```
stardew = pd.read_csv('data/likes_dislikes.tsv', sep='\t') #Note the comma between arguments!
stardew
```

	Villager	Likes	Dislikes
0	Abigail	Amethyst, Pufferfish, Pumpkin, Chocolate Cake	Clay, Wild Horseradish
1	Sebastian	Frozen Tear, Sashimi, Obsidian, Pumpkin Soup	Egg, Mayonnaise, Pickles, Corn
2	Penny	Poppy, Melon, Poppyseed Muffin, Sandfish	Beer, Hops, Void Egg
3	Emily	Cloth, Aquamarine, Ruby, Survival Burger	Fish, Copper Bar, Maki Roll
4	Shane	Beer, Pizza, Hot Pepper, Pepper Poppers	Pickles, Parsnip, Hops

The sequence `\t` is a special symbol representing a tab. In fact, the `sep` argument can take any value, but it's rare to find files that use other separators.

What happens if you forget to add this argument, and ask Pandas to open a TSV file? Let's find out...

```
oops = pd.read_csv('data/likes_dislikes.tsv')
oops
```

```
---------------------------------------------------------------------------
ParserError                               Traceback (most recent call last)
Cell In[8], line 1
----> 1 oops = pd.read_csv('data/likes_dislikes.tsv')
      2 oops

File ~/miniconda3/lib/python3.11/site-packages/pandas/io/parsers/readers.py:1026, in read_csv(filepath_
   1013 kwds_defaults = _refine_defaults_read(
   1014     dialect,
   1015     delimiter,
   (...)
   1022     dtype_backend=dtype_backend,
   1023 )
   1024 kwds.update(kwds_defaults)
-> 1026 return _read(filepath_or_buffer, kwds)

File ~/miniconda3/lib/python3.11/site-packages/pandas/io/parsers/readers.py:626, in _read(filepath_or_b
    623     return parser
    625 with parser:
--> 626     return parser.read(nrows)

File ~/miniconda3/lib/python3.11/site-packages/pandas/io/parsers/readers.py:1923, in TextFileReader.rea
   1916 nrows = validate_integer("nrows", nrows)
   1917 try:
   1918     # error: "ParserBase" has no attribute "read"
   1919     (
   1920         index,
   1921         columns,
   1922         col_dict,
-> 1923     ) = self._engine.read(  # type: ignore[attr-defined]
   1924         nrows
   1925     )
   1926 except Exception:
   1927     self.close()

File ~/miniconda3/lib/python3.11/site-packages/pandas/io/parsers/c_parser_wrapper.py:234, in CParserWra
    232 try:
    233     if self.low_memory:
--> 234         chunks = self._reader.read_low_memory(nrows)
    235         # destructive to chunks
    236         data = _concatenate_chunks(chunks)
```

```
File parsers.pyx:838, in pandas._libs.parsers.TextReader.read_low_memory()

File parsers.pyx:905, in pandas._libs.parsers.TextReader._read_rows()

File parsers.pyx:874, in pandas._libs.parsers.TextReader._tokenize_rows()

File parsers.pyx:891, in pandas._libs.parsers.TextReader._check_tokenize_status()

File parsers.pyx:2061, in pandas._libs.parsers.raise_parser_error()

ParserError: Error tokenizing data. C error: Expected 5 fields in line 3, saw 7
```

This raises an error, sometimes called an exception. The error message contains a 'traceback', which describes where the code stopped working, and sometimes why. The traceback can look overwhelming at first, especially when it is very long. Normally, you start by scrolling to the very bottom of the traceback, where the specific type of error will be mentioned and (sometimes) there is also a message explaining the error. There are many types of errors, just like there are many types of data structures, and you will learn to recognize them through practice.

Don't worry if your code raises errors. It's completely normal part of learning, and there's no harm. Try your best to interpret the error message, and re-examine your code for potential mistakes. If you don't know what the error means, ask your instructor.

Practice

Open the file `data/villains.txt` in a text editor. Determine what the delimiter is, then write code below to open the file with pandas.

```
#PRACTICE CELL
```

The first line in a csv/tsv file is typically a "header", which is a list of column names/labels. It doesn't contain data. Pandas knows this, and by default assumes your file contains a header, as seen in all previous examples. When the DataFrame is created, header names are stored in an atrribute called `columns`:

```
stardew.columns
```

```
Index(['Villager', 'Likes', 'Dislikes'], dtype='object')
```

For a slightly more readable output, you can add `.to_list()`

```
stardew.columns.to_list()
```

```
['Villager', 'Likes', 'Dislikes']
```

Sometimes files lack headers, and every line consists of data. This can occurs when the data is intended to be fed directly into a computer program, where the code is written to locate information by its ordinal position (first column, second column, etc.) and it doesn't need to know anything about the names of the columns.

In such a situation, you don't want Pandas to treat the first line as column names. To ensure these files are read correctly, use the argument `headers=None` Note that `None` doesn't take quotes, and starts with a capital letter. This is a special type in Python that indicates an absence of any value. You don't need the details of how this works, but you should be aware of `None` as it comes up from time to time.

The `measurements.tsv` file contains some (randomly genreated) measurement data without headers, as an example.

```
```

```
measurements = pd.read_csv('data/measurements.tsv', sep='\t', header=None)
measurements.head()
```

	0	1	2	3
0	101	35	12.99	North
1	102	40	15.99	East
2	103	55	9.99	West
3	104	22	19.99	South
4	105	60	5.99	North

Notice that Pandas added column headers for you in this case, although they are simply numbers.

Important: *Python starts counting from 0!* The "first" column in the table has the label 0. This is true of most, but not all, programming languages. The R language is a noteable exception, which starts from 1, and if you're also taking the R class this may trip you up at first.

If you want to add column names, you can do this by adding an argument called `names` like this:

```
measurements = pd.read_csv('data/measurements.tsv', sep='\t', header=None, names=['Experiment ID', 'Tem
measurements.head()
```

	Experiment ID	Temperature	Speed	Direction
0	101	35	12.99	North
1	102	40	15.99	East
2	103	55	9.99	West
3	104	22	19.99	South
4	105	60	5.99	North

Note how the value for `names` is written inside square brackets. This is another basic type in Python called a *list*. In this example, the list contains a set of four strings because that's most appropriate for column labels, but you can store information of any type inside of a list.

Headers are generally contained in the first line of a file, but in some rare cases they might appear on a different line. In this case, instead of using `header=None` you can supply a number that corresponds to the header line. For example, in the file `data/sales.csv` the first two lines includes some general statistics followed by a blank line, then the headers appear on the 4th line. Since Python starts counting at zero, that should be line 3:

```
sales = pd.read_csv('data/sales.csv', header=3)
sales
```

	001	150	2024-08-01
0	2	200	2024-08-03
1	3	250	2024-08-07

Except that doesn't look right! Those aren't normal column headers. It turns out that Pandas ignores blank lines in the file and so they don't count. We actually have to set the header to 2

```
sales = pd.read_csv('data/sales.csv', header=2)
sales
```

	CustomerID	PurchaseAmount	PurchaseDate
0	1	150	2024-08-01
1	2	200	2024-08-03
2	3	250	2024-08-07

Opening files on the internet

The `read_csv()` function can be used for reading csv/tsv files stored on the internet. This works exactly the same way as loading a file from your computer, except you specify the web address. A very famous machine-learning dataset called the "Iris Dataset" is available online as a CSV file for example. Note this file lacks headers.

```
iris = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=N
iris
```

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

In addition to plain text csv/tsv files, Pandas also supports opening spreadsheets in the Microsoft Excel. Spreadsheets are stored in a different format beacuse they also have to record information about font types and sizes, cell colours and borders, formulas, etc. In addition, a single spreadsheet file can save multiple 'sheets' of data at once.

To open spreadsheets, use the `.read_excel()` function. This method has the same `header` and `names` arguments you learned for `.read_csv()`, but it doesn't need a `sep` argument (that informatoin is stored as part of the spreadsheet, so Pandas can look it up for you).

Some spreadsheet file contain multiple sheet, but you can only load one at a time. andas assumes you want the first one. If you need to specify another one, you can add a `sheet_name` argument. You can pass an integer to this argument (e.g.

`sheet_name=4` loads the 3rd sheet) or you can pass a string representing the name of the sheet (e.g. `sheet_name='Quarterly Earnings'`)

Practice

The file `data/World_Development_Indicators.xlsx` contains some randomly selected data from the World Bank's World Development Indicators (https://databank.worldbank.org/source/world-development-indicators#). It is saved as a spreadsheet with two sheets. The first sheet is metadata, the second sheet contains the actual Indicator data. Try writing some code below to open this second sheet with Pandas.

Note: You may get an `ImportError` when you use `pd.read_excel()`, depending on how your Python is set up. If this happens, uncomment the following code and run it to install another package (just like you installed Pandas earlier). Delete the cell, since you don't need to install the package twice, then re-run your `.open_excel()` code.

```
#Run this if necessary
#!pip install openpyxl
```

```
#PRACTICE CELL
```

Lastly, it's worth mentioning that Pandas has over a dozen different `.read_something()` methods, for various files types, such as `pd.read_html()`, `pd.read_json()`, and `pd.read_sql()`. You can consult the official documentation for more details on these: https://pandas.pydata.org/docs/search.html?q=read_

- There are several ways to select data from a DataFrame:
    1. `[ ]` and `[[ ]]`
    2. `.iloc[]`
    3. `.loc[]`

Indexing with [ ] and [[ ]]

You can access a single column by placing the name in quotes between square brackets.

```
imdb['Series_Title']
```

```
0      The Shawshank Redemption
1                 The Godfather
2               The Dark Knight
3         The Godfather: Part II
4                   12 Angry Men
                  ...
995      Breakfast at Tiffany's
996                       Giant
997       From Here to Eternity
998                    Lifeboat
999                The 39 Steps
Name: Series_Title, Length: 1000, dtype: object
```

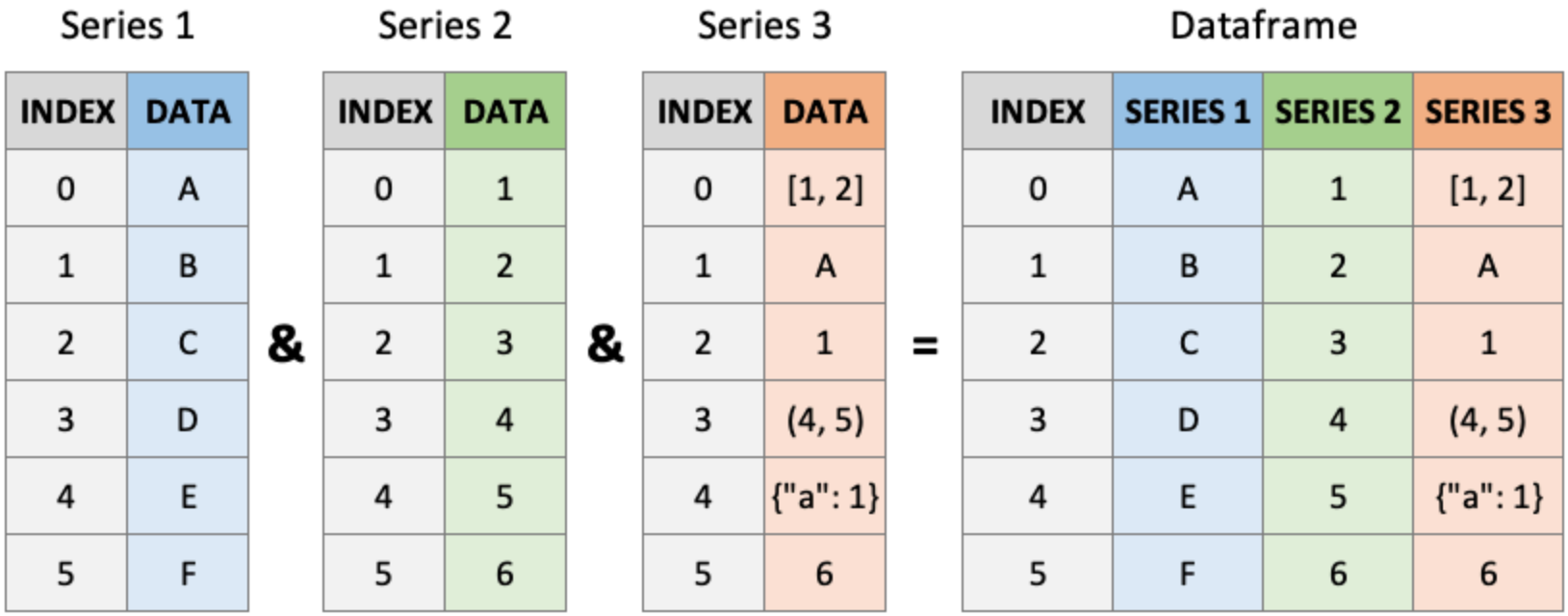
To get data from multiple columns, list all column names inside double square brackets.

```
imdb[['Series_Title', 'Genre']]
```

	Series_Title	Genre
0	The Shawshank Redemption	Drama
1	The Godfather	Crime, Drama
2	The Dark Knight	Action, Crime, Drama
3	The Godfather: Part II	Crime, Drama
4	12 Angry Men	Crime, Drama
...	...	...
995	Breakfast at Tiffany's	Comedy, Drama, Romance
996	Giant	Drama, Western
997	From Here to Eternity	Drama, Romance, War
998	Lifeboat	Drama, War
999	The 39 Steps	Crime, Mystery, Thriller

1000 rows × 2 columns

Note: The double-bracket method creates a new DataFrame, containing only the columns you specified. The single-bracket method actually creates a different type of object called a *Series*, which represents a single column, not a table. A DataFrame is actually made up of multiple Series objects. You can think of their relationship like this:



Once you have a column (Series), `.value_counts()` is a useful way to quickly inspect the data.

```
directors = imdb['Director']
directors.value_counts()
```

```
Director
Alfred Hitchcock    14
Steven Spielberg    13
Hayao Miyazaki      11
Akira Kurosawa      10
Martin Scorsese     10
                    ..
Kinji Fukasaku       1
Eric Bress           1
Thomas Kail          1
Irvin Kershner       1
Lana Wachowski       1
Name: count, Length: 548, dtype: int64
```

A related function is `.nunique()` which counts the number of unique values in your data:

```
directors.nunique()
```

```
548
```

This function actually returns

Practice

- Create a DataFrame with columns for the 4 starring actors
- Create a Series with just the names of the directors
- Create a DataFrame that only contains the run time of the movies
- Find all the different genres with `.value_counts()`

```
#PRACTICE CELL
```

Indexing with `.iloc`

First we'll try out `.iloc[]` which accepts *integers* as references to rows/columns. Integers are another type of Python object, which represent whole numbers.

The code in the following block returns the first row of the IMDB table:

(**Remember:** *Python starts counting from 0, not 1!*)

```
imdb.iloc[0]
```

```
Series_Title     The Shawshank Redemption
Released_Year                        1994
Certificate                             A
Runtime                           142 min
Genre                               Drama
IMDB_Rating                           9.3
Meta_score                           80.0
Director                    Frank Darabont
Star1                          Tim Robbins
Star2                       Morgan Freeman
Star3                           Bob Gunton
Star4                       William Sadler
No_of_Votes                        2343110
Gross                             28341469.0
Name: 0, dtype: object
```

Just like before, using a single bracket will get you a Series (a column of data) and using double brackets gets you a DataFrame (a table of data).

```
imdb.iloc[[0]]
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director	Star
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont	Tim Robbins

To count 'backwards' you can use negative numbers, starting from -1

```
imdb.iloc[[-1]]  # Returns a DataFrame with only the last row of the table
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director	S
999	The 39 Steps	1935	NaN	86 min	Crime, Mystery, Thriller	7.6	93.0	Alfred Hitchcock	R D

If you want multiple rows, you can list them all in double brackets

```
imdb.iloc[[0, 5, 99]]
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director	
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont	R
5	The Lord of the Rings: The Return of the King	2003	U	201 min	Action, Adventure, Drama	8.9	94.0	Peter Jackson	
99	Good Will Hunting	1997	U	126 min	Drama, Romance	8.3	70.0	Gus Van Sant	W

You can select both a row and column like this:

```
imdb.iloc[6, 0] #7th row, 1st column
```

```
'Pulp Fiction'
```

To select multiple rows and columns, you can list them all individually in square brackets:

```
imdb.iloc[ [3,5,10], [1,2] ] #4th, 6th, and 10th rows, 2nd and 3rd columns
```

	Released_Year	Certificate
3	1974	A
5	2003	U
10	2001	U

Or you can use "slice" notation to select range of rows or columns. Slices are written as `x:y` which reads as "starting from position x, going up to *but not including* position y". You can omit the first number, and Python will assume you want to start on the first position (position 0). You can omit the last number and Python will assume you want up to and including the last row.

```
imdb.iloc[0:5] #get the first five rows, i.e. from row 0 up to but not including row 5
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director	S
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont	Rob
1	The Godfather	1972	A	175 min	Crime, Drama	9.2	100.0	Francis Ford Coppola	Ma Bra
2	The Dark Knight	2008	UA	152 min	Action, Crime, Drama	9.0	84.0	Christopher Nolan	Chri
3	The Godfather: Part II	1974	A	202 min	Crime, Drama	9.0	90.0	Francis Ford Coppola	Pa
4	12 Angry Men	1957	U	96 min	Crime, Drama	9.0	96.0	Sidney Lumet	H Fo

```
imdb.iloc[:10, 4:] #get the first 10 row, and from the 5th column onward
```

	Genre	IMDB_Rating	Meta_score	Director	Star1	Star2	Star3	Star4	No_of_Vo
0	Drama	9.3	80.0	Frank Darabont	Tim Robbins	Morgan Freeman	Bob Gunton	William Sadler	2343
1	Crime, Drama	9.2	100.0	Francis Ford Coppola	Marlon Brando	Al Pacino	James Caan	Diane Keaton	1620
2	Action, Crime, Drama	9.0	84.0	Christopher Nolan	Christian Bale	Heath Ledger	Aaron Eckhart	Michael Caine	2303
3	Crime, Drama	9.0	90.0	Francis Ford Coppola	Al Pacino	Robert De Niro	Robert Duvall	Diane Keaton	1129
4	Crime, Drama	9.0	96.0	Sidney Lumet	Henry Fonda	Lee J. Cobb	Martin Balsam	John Fiedler	689
5	Action, Adventure, Drama	8.9	94.0	Peter Jackson	Elijah Wood	Viggo Mortensen	Ian McKellen	Orlando Bloom	1642
6	Crime, Drama	8.9	94.0	Quentin Tarantino	John Travolta	Uma Thurman	Samuel L. Jackson	Bruce Willis	1826
7	Biography, Drama, History	8.9	94.0	Steven Spielberg	Liam Neeson	Ralph Fiennes	Ben Kingsley	Caroline Goodall	1213
8	Action, Adventure, Sci-Fi	8.8	74.0	Christopher Nolan	Leonardo DiCaprio	Joseph Gordon-Levitt	Elliot Page	Ken Watanabe	2067
9	Drama	8.8	66.0	David Fincher	Brad Pitt	Edward Norton	Meat Loaf	Zach Grenier	1854

```
imdb.iloc[:-200, :] #What does this do?
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont
1	The Godfather	1972	A	175 min	Crime, Drama	9.2	100.0	Francis Ford Coppola
2	The Dark Knight	2008	UA	152 min	Action, Crime, Drama	9.0	84.0	Christopher Nolan
3	The Godfather: Part II	1974	A	202 min	Crime, Drama	9.0	90.0	Francis Ford Coppola
4	12 Angry Men	1957	U	96 min	Crime, Drama	9.0	96.0	Sidney Lumet
...	...	...	...	...	...	...	...	...
795	Ocean's Eleven	2001	UA	116 min	Crime, Thriller	7.7	74.0	Steven Soderbergh
796	Vampire Hunter D: Bloodlust	2000	U	103 min	Animation, Action, Fantasy	7.7	62.0	Yoshiaki Kawajiri
797	O Brother, Where Art Thou?	2000	U	107 min	Adventure, Comedy, Crime	7.7	69.0	Joel Coen
798	Interstate 60: Episodes of the Road	2002	R	116 min	Adventure, Comedy, Drama	7.7	NaN	Bob Gale
799	South Park: Bigger,	1999	A	81 min	Animation, Comedy,	7.7	73.0	Trey Parker

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director
	Longer & Uncut				Fantasy			

800 rows × 14 columns

Practice

Use `.iloc` to do the following:

- Get the 7th, 8th, and 27th rows of the table. Does the order of the integers have to match the order of the table?
- Get the first row and second-to-last column
- Get the IMDB Rating column for row 95
- Get rows 20 through 25 and the second, third, and seventh columns

```
#PRACTICE CELL
```

Indexing with `.loc`

- Now let's look at `.loc` which accepts *labels* as references to rows/columns. Column labels are also called "headers", and row labels are also called "indexes".
- Labels are often represented as text (especially column headers), which is known as a *string* type in Python. Strings are always written between quotation marks.
- If your CSV file doesn't have any labels, then Pandas will assign it integers as labels by default. For example, the IMDB table doesn't have row labels included, so the rows are labelled as `0`, `1`, `2`, `3`, etc. This can be very confusing at first, since `.iloc[]` and `.loc[]` seem to return the same values:

```
imdb.loc[[0]] #Get first row using label
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director	Star
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont	Tim Robbins

```
imdb.iloc[[0]] #Get first row using integer - looks the same as above!
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director	Star
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont	Tim Robbins

```
imdb.loc[0, 'Genre'] # Get the row labelled '0' and the column labelled 'Genre'
```

```
'Drama'
```

```
imdb.iloc[0, 'Genre'] #this will raise an error, because iloc only accepts integers
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:966, in _LocationIndexer._valida
    965 try:
--> 966     self._validate_key(k, i)
    967 except ValueError as err:

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1614, in _iLocIndexer._validate_
   1613 else:
-> 1614     raise ValueError(f"Can only index by location with a [{self._valid_types}]")

ValueError: Can only index by location with a [integer, integer slice (START point is INCLUDED, END poi

The above exception was the direct cause of the following exception:

ValueError                                Traceback (most recent call last)
Cell In[37], line 1
----> 1 imdb.iloc[0, 'Genre'] #this will raise an error, because iloc only accepts integers

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1184, in _LocationIndexer.__geti
   1182     if self._is_scalar_access(key):
   1183         return self.obj._get_value(*key, takeable=self._takeable)
-> 1184     return self._getitem_tuple(key)
   1185 else:
   1186     # we by definition only have the 0th axis
   1187     axis = self.axis or 0

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1690, in _iLocIndexer._getitem_t
   1689 def _getitem_tuple(self, tup: tuple):
-> 1690     tup = self._validate_tuple_indexer(tup)
   1691     with suppress(IndexingError):
   1692         return self._getitem_lowerdim(tup)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:968, in _LocationIndexer._valida
    966         self._validate_key(k, i)
    967     except ValueError as err:
--> 968         raise ValueError(
    969             "Location based indexing can only have "
    970             f"[{self._valid_types}] types"
    971         ) from err
    972 return key
```

```
ValueError: Location based indexing can only have [integer, integer slice (START point is INCLUDED, END
```

In the IMBD data, numbers work perfectly well as row labels, but this isn't always the case. Let's open up another table with data about languages around the world, and set up more appropriate row labels.

Practice

- Use the `.read_csv()` method from the beginning of the lecture to load the dataset called "WACL.csv" from the "data" folder
- Assign it to a variable called 'languages'
- Use the `head()` function to inspect the first 8 rows.

```
#PRACTICE CELL

languages = pd.read_csv('data/WACL.csv')
languages.head()
```

	iso_code	language_name	longitude	latitude	area	continent	status	family	
0	aiw	Aari	36.5721	5.95034	Africa	Africa	not endangered	South Omotic	
1	kbt	Abadi	146.992	-9.03389	Papunesia	Pacific	not endangered	Austronesian	
2	mij	Mungbam	10.2267	6.5805	Africa	Africa	shifting	Atlantic-Congo	
3	aau	Abau	141.324	-3.97222	Papunesia	Pacific	shifting	Sepik	
4	abq	Abaza	42.7273	41.1214	Eurasia	Europe	threatened	Abkhaz-Adyge	keteva

Note on the left there is a series of integers, representing the default row index. All of the languages have a unique 3 letter code in the `iso_column`, and that would make a better index label. We can convert a column to an index with the `set_index()` function like this:

```
languages.set_index('iso_code')
```

iso_code	language_name	longitude	latitude	area	continent	status	family	
aiw	Aari	36.5721	5.95034	Africa	Africa	not endangered	South Omotic	c
kbt	Abadi	146.992	-9.03389	Papunesia	Pacific	not endangered	Austronesian	
mij	Mungbam	10.2267	6.5805	Africa	Africa	shifting	Atlantic-Congo	
aau	Abau	141.324	-3.97222	Papunesia	Pacific	shifting	Sepik	
abq	Abaza	42.7273	41.1214	Eurasia	Europe	threatened	Abkhaz-Adyge	ketevan_lc
...	...	...	...	...	...	...	...	
zom	Zou	93.9253	24.0649	Eurasia	Asia	threatened	Sino-Tibetan	
gnd	Zulgo-Gemzek	14.0578	10.827	Africa	Africa	not endangered	Afro-Asiatic	
zul	Zulu	31.3512	-25.3305	Africa	Africa	not endangered	Atlantic-Congo	
zun	Zuni	-108.782	35.0056	North America	Americas	shifting	?	
zzj	Zuojiang Zhuang	107.3622	21.83753	Eurasia	Asia	not endangered	Tai-Kadai	

3338 rows × 8 columns

Note the new set of codes on the left, which replace the integer indexes from before. The `iso_code` column is also gone from the table. Now try using `.loc` to access the first language like this:

```
languages.loc['aiw']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[40], line 1
----> 1 languages.loc['aiw']

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1191, in _LocationIndexer.__geti
   1189 maybe_callable = com.apply_if_callable(key, self.obj)
   1190 maybe_callable = self._check_deprecated_callable_usage(key, maybe_callable)
-> 1191 return self._getitem_axis(maybe_callable, axis=axis)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1431, in _LocIndexer._getitem_ax
   1429 # fall thru to straight lookup
   1430 self._validate_key(key, axis)
-> 1431 return self._get_label(key, axis=axis)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1381, in _LocIndexer._get_label(
   1379 def _get_label(self, label, axis: AxisInt):
   1380     # GH#5567 this will fail if the label is not present in the axis.
-> 1381     return self.obj.xs(label, axis=axis)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/generic.py:4301, in NDFrame.xs(self, key, ax
   4299             new_index = index[loc]
   4300 else:
-> 4301     loc = index.get_loc(key)
   4303     if isinstance(loc, np.ndarray):
   4304         if loc.dtype == np.bool_:

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexes/range.py:417, in RangeIndex.get_loc(
    415         raise KeyError(key) from err
    416 if isinstance(key, Hashable):
--> 417     raise KeyError(key)
    418 self._check_indexing_error(key)
    419 raise KeyError(key)

KeyError: 'aiw'
```

Ooops! That raises a KeyError, which means that the label 'aiw' still doesn't exist! Why not?

This is because `set_index()`, like many functions in Pandas, returns a *copy* of your DataFrame. It does not modify the original. After setting the index, you need re-assign the new DataFrame to the old variable. Get comfortable with this pattern of coding.

```
languages = languages.set_index('iso_code')
```

```
languages.loc['aiw'] #This returns a Series (a single column), use double brackets [['aiw']] to get a D
```

```
language_name                        Aari
longitude                         36.5721
latitude                          5.95034
area                               Africa
continent                          Africa
status                     not endangered
family                       South Omotic
source           daniel_aberra_aberra_1994
Name: aiw, dtype: object
```

You can also tell Pandas about the index column immediately when you create the DataFrame, instead of setting it later:

```
languages = pd.read_csv('data/WACL.csv', index_col='iso_code')
languages
```

iso_code	language_name	longitude	latitude	area	continent	status	family	
aiw	Aari	36.5721	5.95034	Africa	Africa	not endangered	South Omotic	c
kbt	Abadi	146.992	-9.03389	Papunesia	Pacific	not endangered	Austronesian	
mij	Mungbam	10.2267	6.5805	Africa	Africa	shifting	Atlantic-Congo	
aau	Abau	141.324	-3.97222	Papunesia	Pacific	shifting	Sepik	
abq	Abaza	42.7273	41.1214	Eurasia	Europe	threatened	Abkhaz-Adyge	ketevan_lc
...	...	...	...	...	...	...	...	
zom	Zou	93.9253	24.0649	Eurasia	Asia	threatened	Sino-Tibetan	
gnd	Zulgo-Gemzek	14.0578	10.827	Africa	Africa	not endangered	Afro-Asiatic	
zul	Zulu	31.3512	-25.3305	Africa	Africa	not endangered	Atlantic-Congo	
zun	Zuni	-108.782	35.0056	North America	Americas	shifting	?	
zzj	Zuojiang Zhuang	107.3622	21.83753	Eurasia	Asia	not endangered	Tai-Kadai	

3338 rows × 8 columns

Using `.loc`, you can specify both a row and column label:

```
languages.loc['mnk', 'status']
```

```
'not endangered'
```

Lastly, you can select many rows and many columns by putting the labels in square brackets:

```
languages.loc[ ['dmg','klv','ute'], ['latitude', 'longitude'] ]
```

iso_code	latitude	longitude
dmg	5.31911	116.901
klv	-16.5073	167.818
ute	40.0965	-110.305

In some cases, you might want to combine an integer with a label. If you have a row label and a column number you can combine them using `.loc` like this:

```
languages.loc['aiw', languages.columns[0]] # Get the row labelled 'aiw', and the first column
```

```
'Aari'
```

And if you have a row number and a column label, then use this pattern:

```
languages.loc[languages.index[-1], 'continent'] #Get the last row, and the column labelled the 'Contine
```

```
'Asia'
```

Practice

- Use .loc to return a DataFrame for the language with the code ‘aiw’
- Use .loc to find the continent for the Zuni language (‘zun’)
- Use .loc to find the language family and endangerment status of Hausa (‘hau’), Japanese (‘jpn’), and Warlpiri (‘wbp’)
- Use .loc to get the longitude column for the 7th row
- Use .loc to get the language family for the 3rd row from the bottom

```
#PRACTICE CELL
```

Indexing cheatsheet

Method	Syntax	Output
Select column	`df[col_label]`	Series
Select row/column by label	`df.loc[row_label, col_label]`	Object for single selection, Series for one row/column, otherwise DataFrame
Select row/column by integer	`df.iloc[row_int, col_int]`	Object for single selection, Series for one row/column, otherwise DataFrame
Select by row integer & column label	`df.loc[df.index[row_int], col_label]`	Object for single selection, Series for one row/column, otherwise DataFrame
Select by row label & column integer	`df.loc[row_label, df.columns[col_int]]`	Object for single selection, Series for one row/column, otherwise DataFrame

One powerful feature of Pandas is the ability to perform operations on entire columns of data at once (technically called 'vectorization'). This is often useful when dealing with numbers, so let's load another dataset with student scores across a six different high school subjects. Open the file `data/student_scores.csv` using the `pd.read_csv()` function and save it in a variable called `scores`.

Practice

- Set the row index to the student ID
- What did student #41169 score in History?
- Get only the Geography and Art scores.

- How did student #52230 score on the 10th question?
- How did the first three students score in English?

```
#PRACTICE CELL
```

It's common to want the highest, lowest, and average scores for a student, or for a particular subject. This which can be done using the functions `.max()`, `.idxmax()`, `.min()`, `.idxmin()`, or `.mean()`.

```
scores = pd.read_csv('data/student_scores.csv')
scores = scores.set_index('Student_ID')
```

```
#Find the highest score for a particular student
scores.loc[46410].max()
```

```
np.int64(99)
```

```
#Find the row index (=student IDs) of the students with the highest score in each subject
scores.idxmax()
```

```
Biology      56645
Chemistry    69204
Physics      69204
English      46410
Drama        79018
Art          69346
dtype: int64
```

```

```

```
#For each student, find their highest scoring subject
scores.idxmax(axis=1)
```

```
Student_ID
23583    Biology
69204    Physics
74763    Biology
79080    English
61824      Drama
49915      Drama
16055        Art
47192    English
48641    Physics
65083    Physics
56645    Biology
62012      Drama
92865        Art
13367    Biology
69346        Art
41169      Drama
68390    English
98353    English
58887    Physics
69618      Drama
51213    Biology
79018      Drama
46410    English
90278    English
34549    Biology
52230      Drama
13918    Biology
14379    English
54578    Biology
79496    Biology
dtype: object
```

```
#Find the lowest score in a particular subject
scores['English'].min()
```

```
np.int64(53)
```

```
#Find the row index (=student ID) of the students with the lowest scores in each subject
scores.idxmin()
```

```
Biology      41169
Chemistry    79080
Physics      92865
English      69204
Drama        48641
Art          13367
dtype: int64
```

```
#For each row (student), find their lowest scoring subject
scores.idxmin(axis=1)
```

```
Student_ID
23583          Art
69204      English
74763      English
79080    Chemistry
61824    Chemistry
49915    Chemistry
16055      Physics
47192          Art
48641        Drama
65083    Chemistry
56645    Chemistry
62012    Chemistry
92865    Chemistry
13367    Chemistry
69346    Chemistry
41169      English
68390        Drama
98353    Chemistry
58887      English
69618    Chemistry
51213    Chemistry
79018          Art
46410    Chemistry
90278    Chemistry
34549        Drama
52230          Art
13918    Chemistry
14379      Physics
54578    Chemistry
79496      Physics
dtype: object
```

```
#Find the average for every column (subject) at once
scores.mean()
```

```
Biology      78.966667
Chemistry    53.200000
Physics      71.933333
English      76.000000
Drama        78.100000
Art          70.433333
dtype: float64
```

```
#Find the average for every row (student) at once
scores.mean(axis=1)
```

```
Student_ID
23583    70.500000
69204    79.333333
74763    66.166667
79080    76.666667
61824    82.166667
49915    67.500000
16055    73.833333
47192    72.333333
48641    73.333333
65083    74.333333
56645    72.500000
62012    79.500000
92865    62.166667
13367    70.666667
69346    67.833333
41169    68.000000
68390    77.500000
98353    63.166667
58887    69.666667
69618    75.500000
51213    61.166667
79018    71.166667
46410    68.166667
90278    83.000000
34549    74.833333
52230    65.833333
13918    76.333333
14379    68.166667
54578    67.666667
79496    64.166667
dtype: float64
```

The results of these operations can be assigned to new columns in your DataFrame.

```
scores['Student Mean'] = scores.mean(axis=1)
scores
```

Student_ID	Biology	Chemistry	Physics	English	Drama	Art	Student Mean
23583	97	71	67	62	72	54	70.500000
69204	88	74	97	53	67	97	79.333333
74763	74	64	68	57	70	64	66.166667
79080	81	25	81	94	91	88	76.666667
61824	71	69	83	97	98	75	82.166667
49915	58	54	54	78	87	74	67.500000
16055	63	72	58	67	91	92	73.833333
47192	70	54	70	94	94	52	72.333333
48641	70	66	96	91	53	64	73.333333
65083	90	27	93	76	72	88	74.333333
56645	99	37	66	76	86	71	72.500000
62012	92	59	86	63	96	81	79.500000
92865	67	29	50	74	75	78	62.166667
13367	99	44	79	90	61	51	70.666667
69346	58	42	73	60	76	98	67.833333
41169	56	69	75	55	85	68	68.000000
68390	79	62	94	96	55	79	77.500000
98353	68	38	70	84	68	51	63.166667
58887	82	54	92	53	81	56	69.666667

	Biology	Chemistry	Physics	English	Drama	Art	Student Mean
Student_ID							
69618	94	32	88	81	96	62	75.500000
51213	87	34	50	69	75	52	61.166667
79018	96	64	55	59	99	54	71.166667
46410	56	30	77	99	92	55	68.166667
90278	74	73	82	98	89	82	83.000000
34549	93	65	61	79	59	92	74.833333
52230	58	70	59	55	99	54	65.833333
13918	98	51	77	76	75	81	76.333333
14379	77	56	55	87	57	77	68.166667
54578	94	45	51	88	58	70	67.666667
79496	80	66	51	69	66	53	64.166667

A highly useful feature of Pandas is the ability to perform math on entire columns at once. For example, suppose the Chemistry exam was too difficult, and we need to scale everyone's grade up by a small amount.

```
scores['Scaled Chemistry'] = scores['Chemistry'] * 1.03
scores
```

Student_ID	Biology	Chemistry	Physics	English	Drama	Art	Student Mean	Scaled Chemistry
23583	97	71	67	62	72	54	70.500000	73.13
69204	88	74	97	53	67	97	79.333333	76.22
74763	74	64	68	57	70	64	66.166667	65.92
79080	81	25	81	94	91	88	76.666667	25.75
61824	71	69	83	97	98	75	82.166667	71.07
49915	58	54	54	78	87	74	67.500000	55.62
16055	63	72	58	67	91	92	73.833333	74.16
47192	70	54	70	94	94	52	72.333333	55.62
48641	70	66	96	91	53	64	73.333333	67.98
65083	90	27	93	76	72	88	74.333333	27.81
56645	99	37	66	76	86	71	72.500000	38.11
62012	92	59	86	63	96	81	79.500000	60.77
92865	67	29	50	74	75	78	62.166667	29.87
13367	99	44	79	90	61	51	70.666667	45.32
69346	58	42	73	60	76	98	67.833333	43.26
41169	56	69	75	55	85	68	68.000000	71.07
68390	79	62	94	96	55	79	77.500000	63.86
98353	68	38	70	84	68	51	63.166667	39.14
58887	82	54	92	53	81	56	69.666667	55.62

Student_ID	Biology	Chemistry	Physics	English	Drama	Art	Student Mean	Scaled Chemistry
69618	94	32	88	81	96	62	75.500000	32.96
51213	87	34	50	69	75	52	61.166667	35.02
79018	96	64	55	59	99	54	71.166667	65.92
46410	56	30	77	99	92	55	68.166667	30.90
90278	74	73	82	98	89	82	83.000000	75.19
34549	93	65	61	79	59	92	74.833333	66.95
52230	58	70	59	55	99	54	65.833333	72.10
13918	98	51	77	76	75	81	76.333333	52.53
14379	77	56	55	87	57	77	68.166667	57.68
54578	94	45	51	88	58	70	67.666667	46.35
79496	80	66	51	69	66	53	64.166667	67.98

You can also perform the operations between columns. For example, to find the mean of just the sciences:

```
scores['Science Mean'] = (scores['Biology'] + scores['Chemistry'] + scores['Physics']) / 3
```

Practice

- Add a new column called “Lowest grade” that contains the lowest grade for each student
- Add a new column called “Best subject” that contains the subject where the student got the highest score

- Suppose a TA made a grading error. Increase everyone's English score by 1%

```
#PRACTICE CELL
```

You can remove rows with the `.drop()` function by specifying a row label

```
#drop() returns a copy, so don't forget to save it back to a variable!
scores = scores.drop(79496) #drops the student with id 79496
scores = scores.drop([14379, 54578]) #drops multiple students, note the square brackets
```

You can remove columns with `.drop()` by adding `axis=1`

```
scores = scores.drop('Biology', axis=1) #drops the Biology column
scores = scores.drop(['Art', 'Drama'], axis=1) #drops multiple columns, note the square brackets
```

You can also created new DataFrames by selecting only certain rows from an old one, which effectively 'drops' them

```
scores = pd.read_csv('data/student_scores.csv', index_col='Student_ID') #deleted too many things earlie
science_scores = scores[['Biology', 'Chemistry', 'Physics']] #note the double-brackets!
science_scores
```

	Biology	Chemistry	Physics
Student_ID			
23583	97	71	67
69204	88	74	97
74763	74	64	68
79080	81	25	81
61824	71	69	83
49915	58	54	54
16055	63	72	58
47192	70	54	70
48641	70	66	96
65083	90	27	93
56645	99	37	66
62012	92	59	86
92865	67	29	50
13367	99	44	79
69346	58	42	73
41169	56	69	75
68390	79	62	94
98353	68	38	70
58887	82	54	92

	Biology	Chemistry	Physics
Student_ID			
69618	94	32	88
51213	87	34	50
79018	96	64	55
46410	56	30	77
90278	74	73	82
34549	93	65	61
52230	58	70	59
13918	98	51	77
14379	77	56	55
54578	94	45	51
79496	80	66	51

DataFrames can be sorted according to column values with the `sort_values` function. Let's return to the Internet Movie Database.

```
imdb = pd.read_csv('data/imdb.csv')
```

```
imdb.head(5) #remind yourself what this looks like
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director	S
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont	Rob
1	The Godfather	1972	A	175 min	Crime, Drama	9.2	100.0	Francis Ford Coppola	Ma Bra
2	The Dark Knight	2008	UA	152 min	Action, Crime, Drama	9.0	84.0	Christopher Nolan	Chri
3	The Godfather: Part II	1974	A	202 min	Crime, Drama	9.0	90.0	Francis Ford Coppola	Pa
4	12 Angry Men	1957	U	96 min	Crime, Drama	9.0	96.0	Sidney Lumet	H Fo

```
#By default sorts from lowest to highest
imdb.sort_values(by='Gross')
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director
630	Adams æbler	2005	R	94 min	Comedy, Crime, Drama	7.8	51.0	Anders Thomas Jensen
390	Knockin' on Heaven's Door	1997	NaN	87 min	Action, Crime, Comedy	8.0	NaN	Thomas Jahn
624	Mr. Nobody	2009	R	141 min	Drama, Fantasy, Romance	7.8	63.0	Jaco Van Dormael
926	Dead Man's Shoes	2004	NaN	90 min	Crime, Drama, Thriller	7.6	52.0	Shane Meadows
605	Ajeossi	2010	R	119 min	Action, Crime, Drama	7.8	NaN	Jeong-beom Lee
...	...	...	...	...	...	...	...	...
993	Blowup	1966	A	111 min	Drama, Mystery, Thriller	7.6	82.0	Michelangelo Antonioni
995	Breakfast at Tiffany's	1961	A	115 min	Comedy, Drama, Romance	7.6	76.0	Blake Edwards
996	Giant	1956	G	201 min	Drama, Western	7.6	84.0	George Stevens
998	Lifeboat	1944	NaN	97 min	Drama, War	7.6	78.0	Alfred Hitchcock
999	The 39 Steps	1935	NaN	86 min	Crime, Mystery, Thriller	7.6	93.0	Alfred Hitchcock

1000 rows × 14 columns

```
#If you want to sort highest-to-lowest, set the ascending argument to False
imdb.sort_values(by='Released_Year', ascending=False)
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director
8	Inception	3010	UA	148 min	Action, Adventure, Sci-Fi	8.8	74.0	Christopher Nolan
464	Dil Bechara	2020	UA	101 min	Comedy, Drama, Romance	7.9	NaN	Mukesh Chhabra
613	Druk	2020	NaN	117 min	Comedy, Drama	7.8	81.0	Thomas Vinterberg
205	Soul	2020	U	100 min	Animation, Adventure, Comedy	8.1	83.0	Pete Docter
18	Hamilton	2020	PG-13	160 min	Biography, Drama, History	8.6	90.0	Thomas Kail
...	...	...	...	...	...	...	...	...
193	The Gold Rush	1925	Passed	95 min	Adventure, Comedy, Drama	8.2	NaN	Charles Chaplin
194	Sherlock Jr.	1924	Passed	45 min	Action, Comedy, Romance	8.2	NaN	Buster Keaton
568	Nosferatu	1922	NaN	94 min	Fantasy, Horror	7.9	NaN	F.W. Murnau
127	The Kid	1921	Passed	68 min	Comedy, Drama, Family	8.3	NaN	Charles Chaplin
321	Das Cabinet des Dr. Caligari	1920	NaN	76 min	Fantasy, Horror, Mystery	8.1	NaN	Robert Wiene

1000 rows × 14 columns

After you've created a dataset in Pandas and made some changes, you may want to save those changes back to a file. This can be done easily with `.to_csv()`:

```
imdb.to_csv('my_imdb.csv')
```

Practice

- Load the IMBD data
- Make a new DataFrame that contains only these columns: Series_Title, Runtime, IMDB_Rating, Director
- Set the Series_Title as the row index
- Sort the data in reverse alphabetical order, by director name
- Convert the IMDB rating into a score out of 100
- Assume the Gross value is in American dollars. Convert it to Canadian dollars (1 USD = 1.37 CAD) then remove the original Gross column.
- Write the DataFrame to a csv file and open it in Excel. Note what happens to your row labels in this file!