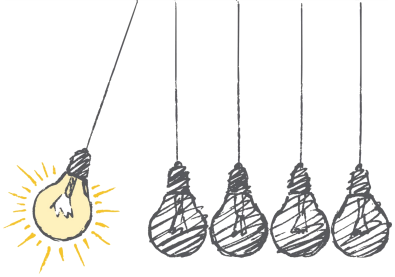
Data Processing in Python Using Pandas



DIME Analytics
Presented by Luis Eduardo San Martin

Development Impact Evaluation (DIME)
The World Bank

WORLD BANK GROUP

i2i DIME
TRANSFORM DEVELOPMENT

Overview

- Introduction
- Pandas
- Importing and exploring data
- Indexing and filtering
- Creating new columns
- Group by
- Merge and append
- Replacing column values
- Descriptive statistics
- Exporting to csv
- Looking ahead

Introduction

Introduction

- This session will introduce you to Pandas
- Pandas is the most popular way to store and process data in Python
- We'll discuss:
  - Pandas dataframes
  - Data processing operations
- We'll compare some Pandas' commands to commands in Stata

Introduction

Why Pandas if I already know Stata?

- **General Python data work:** Almost all Python data work libraries builds on Pandas
- **ML:** If you ever want to implement machine learning using Python, you'll likely need Pandas
- **Cloud platforms:** Cloud platforms assume Python much more often than Stata, and Pandas is the library assumed for data processing
- **Big data:** though Pandas is not suitable for big data, the most popular Python big data tools expect you to know it

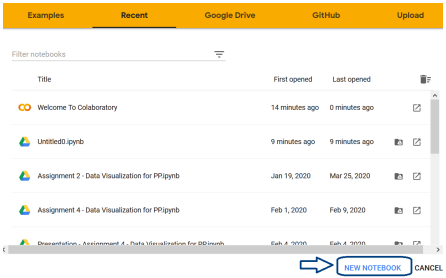
Introduction

Getting started

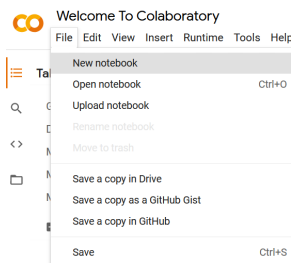
- We'll use Google Colab today
- It is similar to a Google Doc but for coding, and runs Python by default

Getting started

- Go to https://colab.research.google.com
- Click on `NEW NOTEBOOK` if you're already logged in, or go to `File > New notebook` if you're not



Do this if you're already logged in



Do this if you're not – you'll be prompted to log in

Pandas

The Pandas library

- Pandas is an external library for Python, it isn't part of Python's base installation
- In Google colab it is already installed. To use it in your local computer Python installation, you will have to install it using `pip install pandas` in the command line
- Use the following command to import Pandas to your notebook:

```
import pandas as pd
```

Dataframes

- A dataframe is a two-dimensional data structure
- The Stata equivalent to dataframes are datasets
- A dataframe is a data structure where each row represents a unit of observation and each column represents an observation's attribute

	farmerid	crop	Price	Quantity
0	1	Maize	471	5
1	1	Onion	260	18
2	1	Sorghum	469	4
3	1	Spinach	338	4
4	2	Maize	489	9
...	...	...	...	...
10	3	Tomato	96	5
11	3	Wheat	173	1
12	4	Maize	272	13
13	4	Soy	63	15
14	4	Watermelon	269	4

15 rows × 4 columns

Creating a dataframe

There are several ways to create a dataframe from scratch. One of the easiest is:

1. Define a list of strings with the column names

```
[1]  column_names = ['crop', 'quantity']
```

2. Define a list for **each observation**

```
[2]  obs1 = ['Maize', 10]
     obs2 = ['Onion', 8]
```

3. Wrap all of the observation lists in another list

```
[3]  data = [obs1, obs2]
```

Creating a dataframe

4. Use the lists `data` and `column_names` as inputs in the `pd.DataFrame()` command

```
[6]  df = pd.DataFrame(data=data, columns=column_names)
```

5. Now your dataframe is defined in the variable `df`. You can use that name to refer to or see a representation of it, as in:

```
[7]  df
```

	crop	quantity
0	Maize	10
1	Onion	8

Creating a dataframe

Another way to create a dataframe from zero is to define an empty dataframe and then create its columns individually.

```
[8]  df = pd.DataFrame()                  # empty df
     df['crop'] = ['Maize', 'Onion']      # crop col
     df['quantity'] = [10, 8]             # quantity col
     df
```

	crop	quantity
0	Maize	10
1	Onion	8

Importing and exploring data

Importing data to a dataframe from a file

- In our work, we don't usually need to define a dataframe from scratch
- More often, we load pre-existing data files
- To load a `.csv` file into a dataframe, we use the command `pd.read_csv()`

Reading and exploring data

Importing data to a dataframe from a file

`pd.read_csv()`

```
[9]  data_location = 'https://osf.io/925cv/download'
     crops = pd.read_csv(data_location)
```

Importing data to a dataframe from a file

`pd.read_csv()`

```
[9]  data_location = 'https://osf.io/925cv/download'
     crops = pd.read_csv(data_location)
```

- `data_location` is a string with the location of our file
- It can be a URL or a path in your local disk – though a path in your local disk won't work directly with Colab
- `pd.read_csv()` is the Pandas function to read `.csv` files into dataframes. It takes the file location string as input

Reading and exploring data

Importing data to a dataframe from a file

```
[10] crops
```

	farmerid	crop	Price	Quantity
0	1	Maize	471	5
1	1	Onion	260	18
2	1	Sorghum	469	4
3	1	Spinach	338	4
4	2	Maize	489	9
...	...	...	...	...
10	3	Tomato	96	5
11	3	Wheat	173	1
12	4	Maize	272	13
13	4	Soy	63	15
14	4	Watermelon	269	4

15 rows × 4 columns

Importing data to a dataframe from a file

Pandas can also read other type of files:

- `.dta` files with the command `pd.read_stata()`
- `.sav` files with the command `pd.read_spss()`

Though we won't provide examples for them today

Exploring a dataframe

- Running the dataframe name as a command will show a representation of it. If the dataframe has too many rows or columns, Python will print only the first and last rows and columns.

- Though not exactly the same, this is the closest Python has to Stata's `browse` command

```
[10] crops
```

	farmerid	crop	Price	Quantity
0	1	Maize	471	5
1	1	Onion	260	18
2	1	Sorghum	469	4
3	1	Spinach	338	4
4	2	Maize	489	9
...	...	...	...	...
10	3	Tomato	96	5
11	3	Wheat	173	1
12	4	Maize	272	13
13	4	Soy	63	15
14	4	Watermelon	269	4

15 rows × 4 columns

Exploring a dataframe

We can also use the `.head()` and `.tail()` attributes to return the first and last observations of a dataframe

```
[9] crops.head()
```

	farmerid	crop	Price	Quantity
0	1	Maize	471	5
1	1	Onion	260	18
2	1	Sorghum	469	4
3	1	Spinach	338	4
4	2	Maize	489	9

```
[10] crops.tail()
```

	farmerid	crop	Price	Quantity
10	3	Tomato	96	5
11	3	Wheat	173	1
12	4	Maize	272	13
13	4	Soy	63	15
14	4	Watermelon	269	4

Exploring a dataframe

To see how many rows and columns a dataframe has, we use the `.shape` attribute:

```
[11] crops.shape

     (15, 4)
```

The result is a tuple (an immutable list) whose elements are the number of rows and columns

Exploring a dataframe

We can also get the number of rows with the `len()` function:

```
[12] len(crops)

     15
```

The result of `len()` is an integer.

Exploring a dataframe

To check the column names, we use the `.columns` attribute:

```
[13] crops.columns

     Index(['farmerid', 'crop', 'Price', 'Quantity'], dtype='object')
```

Indexing and filtering

Indexing

- Every row and column of a dataframe has a **label**
- Row labels are represented by the index
- Column labels are represented by the column names

index

column names

	farmerid	crop	Price	Quantity
0	1	Maize	471	5
1	1	Onion	260	18
2	1	Sorghum	469	4
3	1	Spinach	338	4
4	2	Maize	489	9
...	...	...	...	...
10	3	Tomato	96	5
11	3	Wheat	173	1
12	4	Maize	272	13
13	4	Soy	63	15
14	4	Watermelon	269	4

15 rows × 4 columns

Column indexing

We can subset a single column of a dataframe using two methods:

- `df.column_name`
- `df["column_name"]`

Remember that in Python we can use both double or single quotes interchangeably most of the times.

Indexing and filtering

Column indexing

```
[11] price = crops.Price
     price
```

```
0     471
1     260
2     469
3     338
4     489
     ...
10     96
11    173
12    272
13     63
14    269
Name: Price, Length: 15, dtype: int64
```

```
[12] price = crops['Price']
     price
```

```
0     471
1     260
2     469
3     338
4     489
     ...
10     96
11    173
12    272
13     63
14    269
Name: Price, Length: 15, dtype: int64
```

Column indexing

- One difference between the two methods is that the first method doesn't allow column name references, while the second does
- The second method also allows to index column names with spaces in the middle

```
[16] col_name = 'Price'
     price = df[col_name]
     price
```

```
0     471
1     260
2     469
3     338
4     489
     ...
10     96
11    173
12    272
13     63
14    269
Name: Price, Length: 15, dtype: int16
```

Multi-column indexing

- We can use a syntax similar to the second method to index more than one column at the same time
- Instead of including a string with one column name inside the brackets, we include a list of strings with the column names to index

```
df[["col_name1", "col_name2", "col_name3", ...]]
```

- Note that inside the outer brackets we have a list of strings

Multi-column indexing

```
[19] price_quantity = crops[['Price', 'Quantity']]
     price_quantity
```

	Price	Quantity
0	471	5
1	260	18
2	469	4
3	338	4
4	489	9
...	...	...
10	96	5
11	173	1
12	272	13
13	63	15
14	269	4

15 rows × 2 columns

```
[18] columns_to_index = ['Price', 'Quantity']
     price_quantity = crops[columns_to_index]
     price_quantity
```

	Price	Quantity
0	471	5
1	260	18
2	469	4
3	338	4
4	489	9
...	...	...
10	96	5
11	173	1
12	272	13
13	63	15
14	269	4

15 rows × 2 columns

Multi-column indexing

A one-column index operation returns a Pandas series. A multicolumn index returns another dataframe.

```
[22] price
```

```
0     471
1     260
2     469
3     338
4     489
     ...
10     96
11    173
12    272
13     63
14    269
Name: Price, Length: 15, dtype: int16
```

```
[23] price_quantity
```

	Price	Quantity
0	471	5
1	260	18
2	469	4
3	338	4
4	489	9
...	...	...
10	96	5
11	173	1
12	272	13
13	63	15
14	269	4

15 rows × 2 columns

Row indexing

There are basically two methods to index rows in Pandas. The simplest is `.iloc[]`, which is used to index the i-th row or rows:

- Indexing a single row: `df.iloc[i]`
- Indexing a range of continuous rows from i until (j-1): `df.iloc[i:j]`
- Indexing the i-th and j-th non-continuous rows: `df.iloc[[i, j]]`

Very important: In Python, every numeric index starts at zero, not at one

Indexing and filtering

Row indexing

- Indexing a single row: `df.iloc[i]`

	farmerid	crop	Price	Quantity
0	1	Maize	471	5
1	1	Onion	260	18
2	1	Sorghum	469	4
3	1	Spinach	338	4
4	2	Maize	489	9
...	...	...	...	...
10	3	Tomato	96	5
11	3	Wheat	173	1
12	4	Maize	272	13
13	4	Soy	63	15
14	4	Watermelon	269	4

15 rows × 4 columns

```
[12] # Indexing the tenth row:
     tenth_row = crops.iloc[10]
     tenth_row
```

```
farmerid          3
crop         Tomato
Price            96
Quantity          5
Name: 10, dtype: object
```

Note that the outcome of indexing a single row is a Pandas series, not a dataframe

Row indexing

- Indexing a range of continuous rows from i until (j-1): `df.iloc[i:j]`

```
[15] # Indexing from rows 5 to 9:
     rows_5_9 = crops[5:10]
     rows_5_9
```

	farmerid	crop	Price	Quantity
5	2	Onion	190	20
6	2	Soy	182	8
7	2	Tomato	252	13
8	2	Watermelon	428	13
9	3	Sorghum	12	11

Note that the index of the resulting dataframe doesn't start with zero anymore. We'll explain more about this in a while.

Row indexing

- Indexing the i-th and j-th non-continuous rows: `df.iloc[[i, j]]`

```
[14] # Indexing three non-consecutive rows:
     rows_2_0_9 = crops.iloc[[2, 0, 9]]
     rows_2_0_9
```

	farmerid	crop	Price	Quantity
2	1	Sorghum	469	4
0	1	Maize	471	5
9	3	Sorghum	12	11

The index of the resulting dataframe is not sorted, it keeps the order in which we selected the rows to subset

Row indexing

The second method to index rows in Pandas is `.loc[]`. It subsets the rows whose index values coincide with the input inside the brackets.

- Until now, the dataframes we've worked with had an index which coincided with the row number
- That's not always the case, as we'll soon see
- The command to index the row whose index value is `i` is this: `df.loc[i]`

We'll show more on the diference between the `.loc[]` and `iloc[]` methods in the next slide

Indexing and filtering

Row indexing

```
[19] rows_2_0_9
```

	farmerid	crop	Price	Quantity
2	1	Sorghum	469	4
0	1	Maize	471	5
9	3	Sorghum	12	11

```
[18] # Getting the 0-th row from rows_2_0_9:
     rows_2_0_9.iloc[0]

     farmerid          1
     crop         Sorghum
     Price            469
     Quantity           4
     Name: 2, dtype: object
```

```
[17] # Getting the row with index value "0"
     # from rows_2_0_9:
     rows_2_0_9.loc[0]

     farmerid          1
     crop          Maize
     Price           471
     Quantity          5
     Name: 0, dtype: object
```

Single-value indexing

To index a single value of a dataframe, we need to index a column and the row-index value `.loc[]` or position `.iloc[]`

```
[14] crops.iloc[4]['crop']

     'Maize'
```

```
[15] crops['crop'].iloc[4]

     'Maize'
```

We can specify the row first and the column later, or viceversa

Filtering

- In Stata, we use the command `keep if` *column_condition* to filter observations
- Pandas' syntax to filter is heavier, as we shall see soon

Filtering

- To filter values of a dataframe, we use brackets and include a list or Pandas series with boolean values inside them:

  ```
  df[list_with_booleans]
  ```

- The observations filtered-in are the ones that have a value of `True` in their cooresponding position

Filtering

Note that the list or Pandas series with booleans needs to have the same length as the dataframe

```
[19] boolean_list = [True] * 5 + [False] * 10
     boolean_list
```

```
[True,
 True,
 True,
 True,
 True,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False]
```

```
[20] crops[boolean_list]
```

	farmerid	crop	Price	Quantity
0	1	Maize	471	5
1	1	Onion	260	18
2	1	Sorghum	469	4
3	1	Spinach	338	4
4	2	Maize	489	9

Filtering

- Other than a list with booleans, we can use a Pandas series with booleans
- The advantage of this is that we can generate them very easily when operating a dataframe column with a logical condition

```
[22] crops['Quantity'] < 6
```

```
0      True
1     False
2      True
3      True
4     False
      ...
10     True
11     True
12    False
13    False
14     True
Name: Quantity, Length: 15, dtype: bool
```

Filtering

```
[24] quantity_less_6 = crops['Quantity'] < 6
     fewest_crops = crops[quantity_less_6]
     fewest_crops
```

	farmerid	crop	Price	Quantity
0	1	Maize	471	5
2	1	Sorghum	469	4
3	1	Spinach	338	4
10	3	Tomato	96	5
11	3	Wheat	173	1
14	4	Watermelon	269	4

Filtering

We can also use more than one condition at the same time:

```
[27] condition = (crops['Price'] < 200) & (crops['crop'] == 'Tomato')
     crops_subset = crops[condition]
     crops_subset
```

	farmerid	crop	Price	Quantity
10	3	Tomato	96	5

Important: When using more than one condition, each of them must be enclosed in parentheses.

Creating new columns

Creating new columns

- To create a new column in a dataframe, we define it using the brackets as in:

```
df[new_col_name] = value
```

- Other than a value, we can use columns operations to define new columns

Creating new columns

```
[31] crops['Revenue'] = crops['Price'] * crops['Quantity']
     crops
```

	farmerid	crop	Price	Quantity	Revenue
0	1	Maize	471	5	2355
1	1	Onion	260	18	4680
2	1	Sorghum	469	4	1876
3	1	Spinach	338	4	1352
4	2	Maize	489	9	4401
...	...	...	...	...	...
10	3	Tomato	96	5	480
11	3	Wheat	173	1	173
12	4	Maize	272	13	3536
13	4	Soy	63	15	945
14	4	Watermelon	269	4	1076

15 rows × 5 columns

Group by

Group by

- The syntax to group a dataframe is:

```
df.groupby(by = "col_name").sum()
```

- This will return a grouped dataframe by `col_name`, where every other column contains a sum of its previous values by `col_name`
- Other possible operations are: `.mean()`, `.std()`, `.quantile()`
- We can also group by more than one column, by replacing "`col_name`" with a list of strings containing the column names to group by

Group by

```
[7] crops
```

	farmerid	crop	Price	Quantity	Revenue
0	1	Maize	471	5	2355
1	1	Onion	260	18	4680
2	1	Sorghum	469	4	1876
3	1	Spinach	338	4	1352
4	2	Maize	489	9	4401
...	...	...	...	...	...
10	3	Tomato	96	5	480
11	3	Wheat	173	1	173
12	4	Maize	272	13	3536
13	4	Soy	63	15	945
14	4	Watermelon	269	4	1076

15 rows × 5 columns

```
[6] farmer_revenue = crops[['farmerid', 'Revenue']].\
                     groupby(by='farmerid').sum()
    farmer_revenue
```

	Revenue
farmerid	
1	10263
2	18497
3	785
4	5557

Group by

- After grouping, the resulting dataframe has the group column as index
- This means that `farmer_revenue` has a **meaningful index**, an index that has information itself and is different than the row number
- Meaningful indices can be useful in some cases, but that's out of the topics we'll cover today

```
[31] farmer_revenue
```

farmerid	Revenue
1	10263
2	18497
3	785
4	5557

Group by

To move `farmerid` back to the columns, use the attribute `.reset_index()`. We could have also used the argument `as_index=False` in `.groupby()` in the first place for this.

```
[22] farmer_revenue = farmer_revenue.reset_index()
     farmer_revenue
```

	farmerid	Revenue
0	1	10263
1	2	18497
2	3	785
3	4	5557

```
[23] farmer_revenue = crops[['farmerid','Revenue']].\
                      groupby(by='farmerid', as_index=False).sum()
     farmer_revenue
```

	farmerid	Revenue
0	1	10263
1	2	18497
2	3	785
3	4	5557

Merge and append

Merge and append

Merge

The basic syntax to merge two dataframes is:

```
pd.merge(left=left_df,
         right=right_df,
         left_on=col_name,
         right_on=col_name,
         how=merge_type)
```

Merge and append

Merge

The type of merge can be one of these values:

- **”left”:** keep only observations from left df, similar to Stata’s `keep(master)` option
- **”right”:** keep only observations from right df, similar to Stata’s `keep(using)`
- **”inner”:** keep all matched observations, similar to `keep(match)`
- **”outer”:** keep all observations, similar to not using Stata’s `keep()` option

Merge and append

Merge

To show how a merge is done, we first read a second dataframe:

```
[31] hh_file = 'https://osf.io/gu5kn/Download'
     hh = pd.read_csv(hh_file)
     hh
```

	hhid	hhmembers	head_age
0	1	9	56
1	2	6	59
2	3	3	40
3	4	9	78
4	5	8	61
...	...	...	...
15	16	7	25
16	17	6	38
17	18	4	46
18	19	3	21
19	20	2	26

20 rows × 3 columns

Merge and append

Merge

```
[34] hh_revenue_outer = pd.merge(left=hh,
                                 right=farmer_revenue,
                                 how='outer',
                                 left_on='hhid',
                                 right_on='farmerid')
     hh_revenue_outer
```

	hhid	hhmembers	head_age	farmerid	Revenue
0	1	9	56	1.0	10263.0
1	2	6	59	2.0	18497.0
2	3	3	40	3.0	785.0
3	4	9	78	4.0	5557.0
4	5	8	61	NaN	NaN
...	...	...	...	...	...
15	16	7	25	NaN	NaN
16	17	6	38	NaN	NaN
17	18	4	46	NaN	NaN
18	19	3	21	NaN	NaN
19	20	2	26	NaN	NaN

20 rows × 5 columns

```
[35] hh_revenue_inner = pd.merge(left=hh,
                                 right=farmer_revenue,
                                 how='inner',
                                 left_on='hhid',
                                 right_on='farmerid')
     hh_revenue_inner
```

	hhid	hhmembers	head_age	farmerid	Revenue
0	1	9	56	1	10263
1	2	6	59	2	18497
2	3	3	40	3	785
3	4	9	78	4	5557

Merge and append

Merge

- An inner merge will only keep the matched observations
- An outer merge will include all obervations
- In Pandas we don’t specify if it’s a merge from one to many or one to one

Merge

- If a column is repeated in both dataframes, they will be added with the suffixes "`_x`" and "`_y`" to differentiate them. To avoid this, it's better to index only the columns we'll need to use before merging
- If we want to generate a column with the source of each row, we need to add the argument `indicator=True`. This is similar to the variable `_merge` that Stata creates by default

Merge and append

Append

Appending two dataframes in Pandas:

```
df.append(other_df)
```

This appends `other_df` to the end of `df`.

Merge and append

Append

```
[38] location = 'https://osf.io/azvmf/Download'
     more_crops = pd.read_csv(location)
     more_crops
```

	farmerid	crop	Price	Quantity
0	5	Maize	429	7
1	5	Onion	237	20
2	5	Sorghum	427	6
3	5	Spinach	308	6
4	6	Maize	445	11
...	...	...	...	...
10	7	Tomato	87	7
11	7	Wheat	157	3
12	8	Maize	248	15
13	8	Soy	57	17
14	8	Watermelon	245	6

15 rows × 4 columns

```
[39] crops_total = crops.append(more_crops)
     crops_total
```

	farmerid	crop	Price	Quantity	Revenue
0	1	Maize	471	5	2355.0
1	1	Onion	260	18	4680.0
2	1	Sorghum	469	4	1876.0
3	1	Spinach	338	4	1352.0
4	2	Maize	489	9	4401.0
...	...	...	...	...	...
10	7	Tomato	87	7	NaN
11	7	Wheat	157	3	NaN
12	8	Maize	248	15	NaN
13	8	Soy	57	17	NaN
14	8	Watermelon	245	6	NaN

30 rows × 5 columns

Merge and append

Append

- The resulting dataframe has 30 rows but the last value we see in the index is 14
- This is because append operations keep the index of the original dataframes intact. The index is now composed of two consecutive counts from 0-14, duplicated
- To reset the index, we can use the attribute `.reset_index(drop=True)` or the argument `ignore_index=True` inside `.append()`

Important: Don't think of the dataframe index as a row unique identifier. It can have duplicated values

Merge and append

Append

```
[40] crops_total = crops_total.reset_index(drop=True)
     crops_total
```

	farmerid	crop	Price	Quantity	Revenue
0	1	Maize	471	5	2355.0
1	1	Onion	260	18	4680.0
2	1	Sorghum	469	4	1876.0
3	1	Spinach	338	4	1352.0
4	2	Maize	489	9	4401.0
...	...	...	...	...	...
25	7	Tomato	87	7	NaN
26	7	Wheat	157	3	NaN
27	8	Maize	248	15	NaN
28	8	Soy	57	17	NaN
29	8	Watermelon	245	6	NaN

30 rows × 5 columns

```
[41] crops_total = crops.append(more_crops, ignore_index=True)
     crops_total
```

	farmerid	crop	Price	Quantity	Revenue
0	1	Maize	471	5	2355.0
1	1	Onion	260	18	4680.0
2	1	Sorghum	469	4	1876.0
3	1	Spinach	338	4	1352.0
4	2	Maize	489	9	4401.0
...	...	...	...	...	...
25	7	Tomato	87	7	NaN
26	7	Wheat	157	3	NaN
27	8	Maize	248	15	NaN
28	8	Soy	57	17	NaN
29	8	Watermelon	245	6	NaN

30 rows × 5 columns

Replacing column values

Replacing column values

- To replace an entire column with a single value, we just overwrite the column with: `df[col_name] = new_value`
- To replace certain values based on a condition, we use `df.loc[]` again

Replacing column values

The syntax to replace values based on conditions is:

```
df.loc[row_indexer, col_indexer] = new_value
```

- **row_indexer:** a list or Pandas series with booleans. Dataframe observations with a value of `True` for their corresponding row order will be replaced
- **col_indexer:** a string with the column name to replace. Can be a list of strings for more than column
- **new_value:** the new value we want to use

Replacing column values

```
[43] crops_total['is_onion'] = False
     crops_total.loc[crops_total['crop'] == 'Onion', 'is_onion'] = True
     crops_total
```

	farmerid	crop	Price	Quantity	Revenue	is_onion
0	1	Maize	471	5	2355.0	False
1	1	Onion	260	18	4680.0	True
2	1	Sorghum	469	4	1876.0	False
3	1	Spinach	338	4	1352.0	False
4	2	Maize	489	9	4401.0	False
...	...	...	...	...	...	...
25	7	Tomato	87	7	NaN	False
26	7	Wheat	157	3	NaN	False
27	8	Maize	248	15	NaN	False
28	8	Soy	57	17	NaN	False
29	8	Watermelon	245	6	NaN	False

30 rows × 6 columns

Descriptive statistics

Descriptive statistics

- The attribute `.describe()` returns a dataframe with descriptive statistics
- By default, it includes only columns with numeric types
- It can also be applied on a single column when indexing

```
[17] desc_stats = crops_total.describe()
     desc_stats
```

	farmerid	Price	Quantity	Revenue
count	30.000000	30.000000	30.000000	15.000000
mean	4.333333	252.400000	10.533333	2340.133333
std	2.309401	142.744189	5.709842	1754.651038
min	1.000000	11.000000	1.000000	132.000000
25%	2.000000	167.750000	6.000000	1010.500000
50%	4.500000	246.500000	10.500000	1876.000000
75%	6.000000	376.250000	15.000000	3668.000000
max	8.000000	489.000000	22.000000	5564.000000

Descriptive statistics

- We can also get statistics for individual columns
- Some attributes for this are: `.mean()`, `.sum()`, `.std()`, `.min()`, `.max()`, `.median()`, `.quantile()`, `.count()`

```
[22] crops_total['Price'].mean()
     252.4

[23] crops_total[crops_total['crop'] == 'Maize']['Quantity'].median()
     10.0

[25] crops_total[crops_total['crop'] == 'Tomato']['Quantity'].quantile(0.75)
     13.5

[26] crops_total['Quantity'].min()
     1
```

Descriptive statistics

To tabulate the values of a column, use the atribute `.value_counts()`

```
[28] crops_total['crop'].value_counts()
```

```
Maize          6
Sorghum        4
Tomato         4
Onion          4
Watermelon     4
Soy            4
Spinach        2
Wheat          2
Name: crop, dtype: int64
```

You can use the argument `dropna=False` to include the counts of `NaN` values

Exporting to csv

Exporting to csv

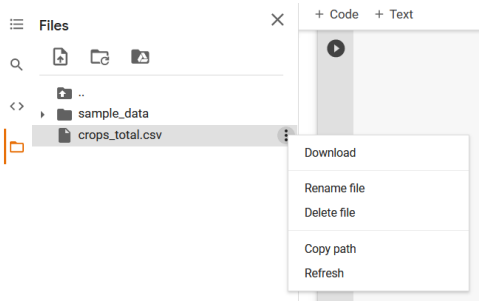
Finally, we can export the dataframe `crops_total` to a csv file using the attribute `.to_csv()`

```
[29] export_file = 'crops_total.csv'
     crops_total.to_csv(export_file, index=False)
```

Pandas by default includes a column with the index when exporting. We use the argument `index=False` to omit it.

Downloading from Colab

- Given that we used Colab for these exercises, the resulting file was exported to Colab's cloud storage
- To download the file to yor computer, click the folder icon to the left, locate the file, click on the vertical ellipsis next to it and click `Download`



Looking ahead

Looking ahead

- Pandas is a huge data processing library
- We've barely skimmed the surface of its features today. It can do any data wrangling operation possible to do in Stata
- Its official documentation is exceptionally clear and detailed, especially for Python standards. Check it out here: https://pandas.pydata.org/docs/
- Some examples of Python data work libraries that use Pandas or build on its syntax:
  - **Data visualization:** altair, seaborn, matplotlib
  - **Machine Learning:** scikit-learn
  - **Big data:** koalas
  - **GIS analysis:** geopandas