Important Shortcut Keys

- A -> To **create** cell **above**
- B -> To **create** Cell **below**
- D D -> For **deleting** the cell
- M -> To **markdown** the Cell
- Y -> For **code** the cell
- Z -> To **undo** the deleted cell

1. Import Python Libraries: pandas

- **Pandas** is an open source Python library for data analysis. It gives Python the ability to work with spreadsheet-like data for fast data loading, manipulating, aligning, and merging, among other functions.
- To give Python these enhanced features, Pandas introduces two new data types to Python: **Series and DataFrame**. The DataFrame represents your entire spreadsheet or rectangular data, whereas the Series is a single column of the DataFrame. A Pandas DataFrame can also be thought of as a dictionary or collection of Series objects.
- When given a data set, we first load it and begin looking at its structure and contents. The simplest way of looking at a data set is to examine and subset specific rows and columns. We can see which type of information is stored in each column, and can start looking for patterns by aggregating descriptive statistics.
- Since Pandas is not part of the Python standard library, we have to first tell Python to load (import) the library.

```
# Import the libraries pandas and numpy

import pandas as pd
pd.__version__
```

```
'1.4.1'
```

2. Read and then Print the Data File in Python

```
# Load (Read) the csv data file (Caution: Use backslash in writing location of the file) & st

df = pd.read_csv('gapeveryfiveyears.csv')


# Show the Loaded CSV data

print(df)
type(type(df))
```

```
           country continent  year  lifeExp       pop   gdpPercap
0      Afghanistan      Asia  1952   28.801   8425333  779.445314
1      Afghanistan      Asia  1957   30.332   9240934  820.853030
2      Afghanistan      Asia  1962   31.997  10267083  853.100710
3      Afghanistan      Asia  1967   34.020  11537966  836.197138
4      Afghanistan      Asia  1972   36.088  13079460  739.981106
...            ...       ...   ...      ...       ...         ...
1699      Zimbabwe    Africa  1987   62.351   9216418  706.157306
1700      Zimbabwe    Africa  1992   60.377  10704340  693.420786
1701      Zimbabwe    Africa  1997   46.809  11404948  792.449960
1702      Zimbabwe    Africa  2002   39.989  11926563  672.038623
1703      Zimbabwe    Africa  2007   43.487  12311143  469.709298

[1704 rows x 6 columns]
type
```

Note:

The above data shows:

For various countries: **life expectancy** (lifeExp), **population** (pop) and **GDP per Capita** (gdpPercap) in every 5 years.

3. Get the Data Frame Information

- `shape` : Get the number of rows and columns of the data frame
- `columns` : Get the Columns names
- `dtypes` : Get the data type of each column
- `info` : Get the more information about the data types and missing values information
- `head()`, `tail()` : Get the first and last five obseravtions of the data frame, respectively.

```
# Find the number of rows and columns in the data frame
```

```
print(df.shape)
print(type(df.shape))
```

```
(1704, 6)
```

```
# Print (Get) the columns labels (names) (heading of each column)

print(df.columns)
print(type(df.columns))
df.columns?
```

```
Index(['country', 'continent', 'year', 'lifeExp', 'pop', 'gdpPercap'], dtype='object')
<class 'pandas.core.indexes.base.Index'>
Type:        Index
String form: Index(['country', 'continent', 'year', 'lifeExp', 'pop', 'gdpPercap'], dtyp
Length:      6
```

```
# Print (Get) the data types of each columns of the data frame

print(df.dtypes)
print(type(df))
df.dtypes?
```

```
country       object
continent     object
year           int64
lifeExp      float64
pop            int64
gdpPercap    float64
dtype: object
<class 'pandas.core.frame.DataFrame'>
Type:        property
String form: <property object at 0x7fb964c00680>
Docstring:
Return the dtypes in the DataFrame.

This returns a Series with the data type of each column.
The result's index is the original DataFrame's columns. Columns
with mixed types are stored with the ``object`` dtype. See
:ref:`the User Guide <basics.dtypes>` for more.

Returns
-------
pandas.Series
    The data type of each column.

Examples
--------
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

```
# Show the first 5 observations of the data frame

print(df.head(10))
print(type(df.head(10)))
```

```
       country continent  year  lifeExp       pop   gdpPercap
```

```
0  Afghanistan      Asia  1952   28.801   8425333  779.445314
1  Afghanistan      Asia  1957   30.332   9240934  820.853030
2  Afghanistan      Asia  1962   31.997  10267083  853.100710
3  Afghanistan      Asia  1967   34.020  11537966  836.197138
4  Afghanistan      Asia  1972   36.088  13079460  739.981106
5  Afghanistan      Asia  1977   38.438  14880372  786.113360
6  Afghanistan      Asia  1982   39.854  12881816  978.011439
7  Afghanistan      Asia  1987   40.822  13867957  852.395945
8  Afghanistan      Asia  1992   41.674  16317921  649.341395
9  Afghanistan      Asia  1997   41.763  22227415  635.341351
<class 'pandas.core.frame.DataFrame'>
```

```
# Show th last 5 observations of the data frame

print(df.tail(10))
```

```
         country continent  year  lifeExp       pop   gdpPercap
1694  Zimbabwe    Africa  1962   52.358   4277736  527.272182
1695  Zimbabwe    Africa  1967   53.995   4995432  569.795071
1696  Zimbabwe    Africa  1972   55.635   5861135  799.362176
1697  Zimbabwe    Africa  1977   57.674   6642107  685.587682
1698  Zimbabwe    Africa  1982   60.363   7636524  788.855041
1699  Zimbabwe    Africa  1987   62.351   9216418  706.157306
1700  Zimbabwe    Africa  1992   60.377  10704340  693.420786
1701  Zimbabwe    Africa  1997   46.809  11404948  792.449960
1702  Zimbabwe    Africa  2002   39.989  11926563  672.038623
1703  Zimbabwe    Africa  2007   43.487  12311143  469.709298
```

Read the following table to know more: Pandas data types Vs Python data types

pandas Type	Python Type	Description
object	string	most common data type
int64	int	whole number
float64	float	numbers with decimal
datetime64	datetime	datetime is found in the Python standard library which is not loaded by default

```
type(df)
```

```
# Print (Get) the more information of the data types of each columns

print(df.info())
print(type(df.info()))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   country    1704 non-null   object
 1   continent  1704 non-null   object
```

```
 2   year       1704 non-null   int64
 3   lifeExp    1704 non-null   float64
 4   pop        1704 non-null   int64
 5   gdpPercap  1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
None
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   country    1704 non-null   object
 1   continent  1704 non-null   object
 2   year       1704 non-null   int64
 3   lifeExp    1704 non-null   float64
 4   pop        1704 non-null   int64
 5   gdpPercap  1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
<class 'NoneType'>
```

Observe non-null in the above output

non-null in any particular column means there is no missing data in that particular cloumn.

4. Looking into the Rows, Columns and Cells

- Different Methods of Indexing Rows (or Columns)

Subset Method	Description
`loc`	Subset based on *index label* (row (or column) name)
`iloc`	Subset based on *row (or column) index* (row (or column) number)

4.1 Subsetting the Rows

- by `loc`
- by `iloc`
- by `head(n)` and `tail(n)` : shows the first n rows data and last n rows data, respectively. For example, If we give n=1 then head(n=1) will show first row and tail(n=1) will show last row.

```
# Single row data
# Show the first row of the data frame [Caution: Python counts from 0]
# using loc command
```

```
print(df.loc[-1])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/opt/JupyterLab/resources/jlab_server/lib/python3.8/site-
packages/pandas/core/indexes/range.py in get_loc(self, key, method, tolerance)
    384                 try:
--> 385                     return self._range.index(new_key)
    386                 except ValueError as err:

ValueError: -1 is not in range

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

KeyError                                  Traceback (most recent call last)
```

5 frames

```
/opt/JupyterLab/resources/jlab_server/lib/python3.8/site-
packages/pandas/core/indexes/range.py in get_loc(self, key, method, tolerance)
    385                     return self._range.index(new_key)
    386                 except ValueError as err:
--> 387                     raise KeyError(key) from err
    388             raise KeyError(key)
    389         return super().get_loc(key, method=method, tolerance=tolerance)

KeyError: -1
```

```
# Single row data
# Show the first row of the data frame [Caution: Python counts from 0]
# using iloc command

print(df.iloc[1])
```

```
country      Afghanistan
continent           Asia
year                1957
lifeExp           30.332
pop              9240934
gdpPercap      820.85303
Name: 1, dtype: object
```

```
# Single row data
# Show the first row of the data frame [Caution: Python counts from 0]
# using head function

print(df.head(1))    # or print(df.head(n= 1))
type(df.head(1))
```

```
      country continent  year  lifeExp      pop   gdpPercap
0  Afghanistan      Asia  1952   28.801  8425333  779.445314
pandas.core.frame.DataFrame
```

```
# Single row data
# Show the 15th row of the data frame [Caution: Python counts from 0]
# using loc command

print(df.loc[14])
```

```
country          Albania
continent         Europe
year                1962
lifeExp            64.82
pop              1728137
gdpPercap    2312.888958
Name: 14, dtype: object
```

```
# Single row data
# Save the 15th row data into its own variable row15_df & also show the 15th row of the data
# using iloc command

print(df.iloc[14])
```

```
# Single row data
# Show the last row of the data frame
# using tail function

print(df.tail(n = 1))
```

```
      country continent  year  lifeExp       pop   gdpPercap
1703  Zimbabwe    Africa  2007   43.487  12311143  469.709298
```

```
# Single row data
# Show the last row of the data frame
# using iloc command

print(df.iloc[-1])    # Compare with previous code using tail fuction
df
```

Note: difference between iloc and loc

With iloc, we can pass -1 to get the last row --- something we could not do with loc.

That is in previous code if you write `print(df.loc[-1])`, it will show error. Try and understand difference between label vs index.

Exercise 1: Show the last row of the data frame (using loc command).

Hint: You need to write some extra lines of code to do the task.

Your Solution Code (write in the cell given below):

[ ] ↳ *3 cells hidden*

4.2 Subsetting the Columns

by Name

- `dataframevariable['column_name']` : Get only one column data.
- `dataframevariable[['ith_column_name', 'jth_column_name', ..., 'kth_column_name']]` : Get multiple columns data.

by loc and iloc command

- by `loc`
- by `iloc`

by Range

- You can use the built-in `range(start, stop, step)` function to create a range of values in Python. This way you can specify beginning and end values, and Python will automatically create a range of values in between. By default, every value between the beginning and the end (inclusive left, exclusive right) will be created, unless you specify a step.
- In Python 3, the range function returns a generator. For example, when range(5) is called, five integers are returned: 0 – 4.
- We will see that we subset columns using a list of integers (in iloc method). Since `range` returns a generator, we have to convert the generator to a list first.
- We use range method for multilple columns data.

by Slicing

- Python's slicing syntax, :, is similar to the range syntax. Instead of a function that specifies start, stop, and step values delimited by a comma, we separate the values with the colon.
- Slicing can be seen as a shorthand means to the same thing as range.
- The colon syntax for slicing only has meaning when slicing and subsetting values, and has no inherent meaning on its own.

```
# Single column data
# Get first column (namely country) data and save it to its own variable (country_df)
# using by name

country_df = df['country']



# Single column data
# Show the first 5 observations of country column

print(country_df.head())


# Single column data
# Show the last 5 observations of country column

print(country_df.tail())


# Multiple columns data

# Question: Show the last 5 observations of first ('country') column, third ('year') column a
# using by name

# Answer:
# first save the given three coulmns data in a new variable
subset1 = df[['country', 'year', 'pop']]  # Note the two square braces

# Show the last 5 observation data of the variable subset1
print(subset1.tail())


# Single column data
# Show the first column of the data frame
# using loc cammand

print(df.loc[:, ['country']])


# Single column data
# Show the first column of the data frame [Caution: Python counts from 0]
# using iloc command

print(df.iloc[:, [0]])


# Multiple columns data
# Show the first ('country') column, third ('year') column and fifth ('pop') column data.
# using loc command

print(df.loc[:, ['country', 'year', 'pop']])
```

```
# Multiple columns data
# Question: save first ('country'), third ('year') and fifth ('pop') columns data in a variab
# using iloc function

# Answer:
# first save the given three coulmns data in a new variable (subset2)
subset2 = df.iloc[:, [0, 2, 4]]  # Note the two square braces

# Show the first 6 data of the variable (subset2)
print(subset2.head(6))



# Multiple columns data
# Question: get the first 4 columns data
# using Range method

# Answer:
# create a range of integers from 0 to 3 inclusive
small_range1 = list(range(4))
print(small_range1)

# subset the dataframe with the range
print(df.iloc[:, small_range1])


# Multiple columns data
# Question: get the last 3 columns data
# using Range method

# Answer:
# create a range of integers from 3 to 5 inclusive
small_range2 = list(range(3,6))
print(small_range2)

# subset the dataframe with the range
print(df.iloc[:, small_range2])


# Multiple columns data
# Question: get the data of first, third and fifth column data
# using Range method
# Hint: (first column - python index 0, third column - python index 2, fifth column - python

# Answer:
# create a range of integers from 0 to 5 exclusive, every other data
small_range3 = list(range(0,5,2))
print(small_range3)

# subset the dataframe with the range
print(df.iloc[:, small_range3])
```

```
# Multiple columns data
# Question: get the last 3 columns data
# using Slicing method

# Answer:
# subset the dataframe with the slicing the last 3 columns (3 to 5 inclusive)
print(df.iloc[:, 3:6])  # or print(df.iloc[:, 3:])



# Multiple columns data
# Question: get the first 3 columns data
# using Slicing method

# Answer:
# subset the dataframe with the slicing the first 3 columns (0 to 2 inclusive)
print(df.iloc[:, 0:3])  # or print(df.iloc[:, :3])


# Multiple columns data
# Question: get the data of third, fourth and fifth column data
# using Slicing method

# Answer:
# subset the dataframe with the slicing the columns 2 to 4 inclusive
print(df.iloc[:, 2:5])


# Multiple columns data
# Question: get the every other first 5 columns
# using Slicing method

# Answer: every other first 5 columns are first, third and fifth columns
# subset the dataframe with the slicing the columns 0 to 5 inclusive with step 2
print(df.iloc[:, 0:6:2])
```

Exercise 2: What is the result in each of the following cases? Verify and Justify.

- `df.iloc[:, 0:6:]`
- `df.iloc[:, 0::2]`
- `df.iloc[:, :6:2]`
- `df.iloc[:, ::2]`
- `df.iloc[:, ::]`

4.3 Subsetting the Cell (Rows and Columns both)

- by `loc`

- by `iloc`

```
# Specific row and specific column data
# Get the 43rd country name in our data frame (df)
# using loc command

print(df.loc[42, 'country'])


# Specific row and specific column data
# Get the 43rd country name in our data frame (df)
# using iloc command

print(df.iloc[42, 0])


# Specific row and multiple columns data
# Get the 43rd country name and its population in our data frame (df)
# using loc command

print(df.loc[42, ['country','pop']])


# Specific row and multiple columns data
# Get the 43rd country name and its population in our data frame (df)
# using iloc command

print(df.iloc[42, [0,4]])  # country is 1st column and population is 5th column


# Multiple rows and specific column data
# Get the 43rd and 54th country names in our data frame (df)
# using loc command

print(df.loc[[42,53], 'country'])


# Multiple rows and specific column data
# Get the 43rd and 54th country names in our data frame (df)
# using iloc command

print(df.iloc[[42,53], 0])


# Multiple rows and multiple columns data
# Get the 1st, 100th and 1000th rows data
# Get the corresponding data of columns 'country', 'lifeExp' and 'gdpPercap'
# using loc command

print(df.loc[[0, 99, 999], ['country','lifeExp', 'gdpPercap' ]])


# Multiple rows and multiple columns data
# Get the 1st, 100th and 1000th rows data
```

```
# Get the 1st, 100th and 1000th rows data
# Get the corresponding data of columns 'country', 'lifeExp' and 'gdpPercap'
# using iloc command

print(df.iloc[[0, 99, 999], [0, 3, 5]])
df.iloc?
```

```
         country  lifeExp    gdpPercap
0    Afghanistan   28.801   779.445314
99    Bangladesh   43.453   721.186086
999     Mongolia   51.253  1226.041130
Type:        property
String form: <property object at 0x7fc03353fdb0>
Docstring:
Purely integer-location based indexing for selection by position.

``.iloc[]`` is primarily integer position based (from ``0`` to
``length-1`` of the axis), but may also be used with a boolean
array.

Allowed inputs are:

- An integer, e.g. ``5``.
- A list or array of integers, e.g. ``[4, 3, 0]``.
- A slice object with ints, e.g. ``1:7``.
- A boolean array.
- A ``callable`` function with one argument (the calling Series or
  DataFrame) and that returns valid output for indexing (one of the above).
  This is useful in method chains, when you don't have a reference to the
  calling object, but would like to base your selection on some value.

``.iloc`` will raise ``IndexError`` if a requested indexer is
out-of-bounds, except *slice* indexers which allow out-of-bounds
indexing (this conforms with python/numpy *slice* semantics).

See more at :ref:`Selection by Position <indexing.integer>`.

See Also
--------
DataFrame.iat : Fast integer location scalar accessor.
DataFrame.loc : Purely label-location based indexer for selection by label.
Series.iloc : Purely integer-location based indexing for
               selection by position.

Examples
--------
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000

**Indexing just the rows**

With a scalar integer.

>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
```