pandas

https://pandas.pydata.org

Cheat Sheet

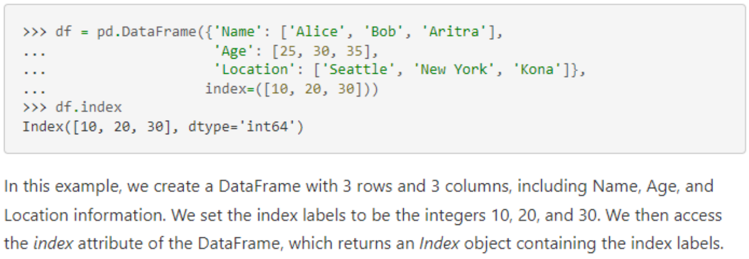
Pandas General Info

- Enhanced versions of NumPy arrays, but rows and columns can have labels rather than simple integer indices
  - Each column in a pandas dataframe can have a label name (i.e. header name such as months) and can contain a different type of data from its neighboring columns (e.g. column_1 with numeric values and column_2 with text strings).
  - By default, each row has an index within a range of values beginning at [0]. However, the row index in pandas dataframes can also be set as labels (e.g. a location name, date).

Pandas Indexing vs. Index Object

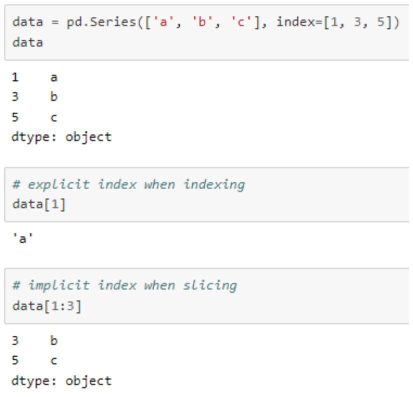
- All cells in a pandas dataframe have both a row index and a column index (i.e. two-dimensional table structure), even if there is only one cell (i.e. value) in the pandas dataframe.
- In addition to selecting cells through location-based indexing (e.g. cell at row 1, column 1), you can also query for data within pandas dataframes based on specific values (e.g. querying for specific text strings or numeric values).
  - DataFrame.loc to get info based on row/column name
  - DataFrame.iloc to get info based on row/column index

- Index Object: The index of a dataframe is an array-like object; by default, is a numeric sequence beginning at 0
  - You can set the index to whatever you want



```
>>> df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Aritra'],
...                    'Age': [25, 30, 35],
...                    'Location': ['Seattle', 'New York', 'Kona']},
...                   index=([10, 20, 30]))
>>> df.index
Index([10, 20, 30], dtype='int64')
```

In this example, we create a DataFrame with 3 rows and 3 columns, including Name, Age, and Location information. We set the index labels to be the integers 10, 20, and 30. We then access the *index* attribute of the DataFrame, which returns an *Index* object containing the index labels.



```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

```
1    a
3    b
5    c
dtype: object
```

```
# explicit index when indexing
data[1]
```

```
'a'
```

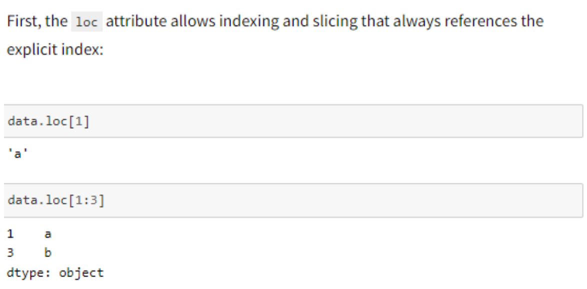
```
# implicit index when slicing
data[1:3]
```

```
3    b
5    c
dtype: object
```

Slicing with Pandas

- Location based:
  - Using iloc (think index location – locating things by the index #)
  - Same rules as for numpy indexing start:stop:step
  - Mydataframe.iloc[Rowselection, columnselection]
- Label-based:
  - To grab out specific rows: use loc and the names of the labels rather than index number
- Grabbing out individual columns
  - To select a column just do: dataframe["columnname"] – returns a 1D pandas series
  - To select a column as a dataframe or to select more than 1 column: dataframe[['columnname1", "columnname2"]]
- Grabbing out ranges of rows:
  - Mydataframe[rowstart:rowstop]
- Filtering values:
  - Finding values that meet certain criteria: Mydataframe[dataframe["column"] == value]

- loc versus iloc example:



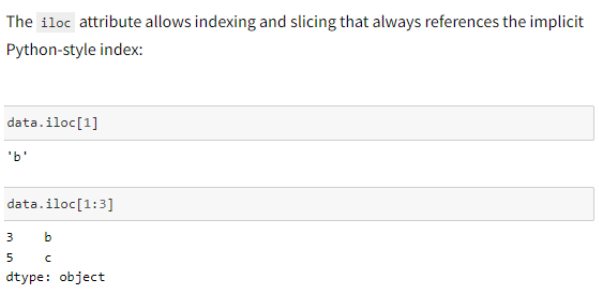
First, the `loc` attribute allows indexing and slicing that always references the explicit index:

```
data.loc[1]
```

```
'a'
```

```
data.loc[1:3]
```

```
1    a
3    b
dtype: object
```



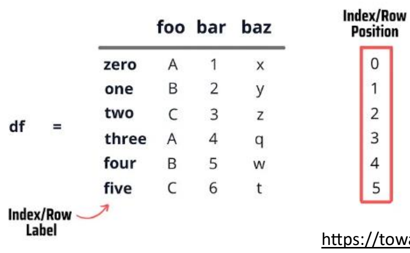
The `iloc` attribute allows indexing and slicing that always references the implicit Python-style index:

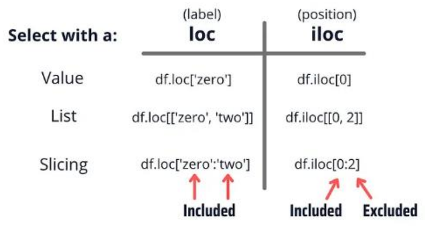
```
data.iloc[1]
```

```
'b'
```

```
data.iloc[1:3]
```

```
3    b
5    c
dtype: object
```





Select with a:	(label) loc	(position) iloc
Value	df.loc['zero']	df.iloc[0]
List	df.loc[['zero', 'two']]	df.iloc[[0, 2]]
Slicing	df.loc['zero':'two']	df.iloc[0:2]

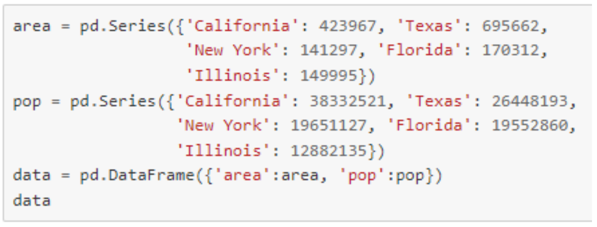
Included
Included
Excluded

https://towardsdatascience.com/

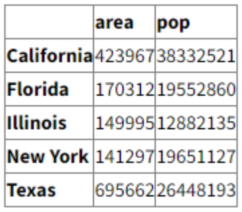
pandas Cheat Sheet

https://pandas.pydata.org

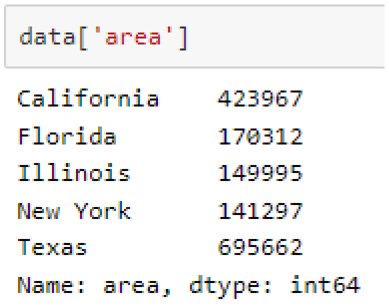
- Dataframe using a dictionary, and index only one column by column name:



```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```



	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193



```
data['area']
```

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Read in a File

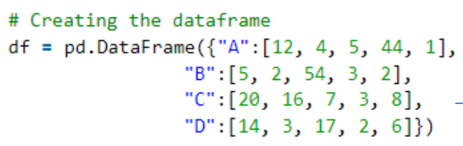
- pandas provides the read_csv() function to read data stored as a csv file into a pandas DataFrame. pandas supports many different file formats or data sources out of the box (csv, excel, sql, json, parquet, …), each of them with the prefix read_*.
  - titanic = pd.read_csv("data/titanic.csv")
  - titanic = pd.read_txt(“data/titanic.txt”)

Pandas Operations

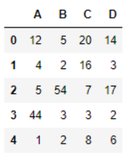
- Because of the tabular structure, you can work with cells in pandas dataframes:
  - across an entire row
  - across an entire column (or series, a one-dimensional array in pandas)
  - by selecting cells based on location or specific values
- Due to its inherent tabular structure, pandas dataframes also allow for cells to have null values (i.e., no data value such as blank space, NaN, -999, etc).

Common Functions and Methods (*Note: **kwargs stands for keyword arguments)*

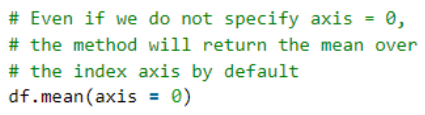
- .head() gives you the first 5 rows
- .tail() gives you the last 5 rows
- .info() gives you information about the dataframe
- .describe() gives you summary statistics on the dataframe (e.g. mean, std, min, max, etc.)
- .sort_values() will sort based on whatever argument you give it, like .sort_values(by=”precip”, asending=False) to sort by a column called “precip” in descending order
- Concatenate: pd.concat([series1,series2]) or pd.concat([df1,df2])
- Insert a column at any position: DataFrameName.insert(loc, column, value, allow_duplicates = False)
- Return the sum of the values for the requested axis:
  - DataFrame.sum(axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwargs)
- Set the index: DataFrame.set_index(keys, drop=True, append=False, inplace=False, verify_integrity=False)
- Reset the index: DataFrame.reset_index(level=None, drop=False, inplace=False, col_level=0, col_fill=”)
- Fill in missing values: DataFrame.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, **kwargs)
- Return the mean of a specified axis: DataFrame.mean(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
- Return the standard deviation of an axis: Series.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)
- Return the max of an axis: DataFrame.max()
- Return the min of an axis: DataFrame.min()



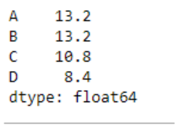
```
# Creating the dataframe
df = pd.DataFrame({"A":[12, 4, 5, 44, 1],
                   "B":[5, 2, 54, 3, 2],
                   "C":[20, 16, 7, 3, 8],
                   "D":[14, 3, 17, 2, 6]})
```



	A	B	C	D
0	12	5	20	14
1	4	2	16	3
2	5	54	7	17
3	44	3	3	2
4	1	2	8	6

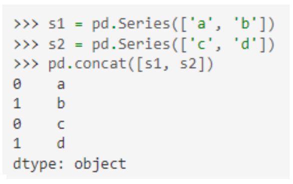


```
# Even if we do not specify axis = 0,
# the method will return the mean over
# the index axis by default
df.mean(axis = 0)
```

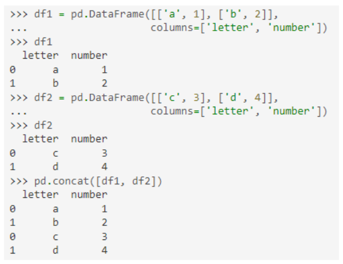


```
A    13.2
B    13.2
C    10.8
D     8.4
dtype: float64
```

<u>Concatenate Examples</u>



```
>>> s1 = pd.Series(['a', 'b'])
>>> s2 = pd.Series(['c', 'd'])
>>> pd.concat([s1, s2])
0    a
1    b
0    c
1    d
dtype: object
```



```
>>> df1 = pd.DataFrame([['a', 1], ['b', 2]],
...                    columns=['letter', 'number'])
>>> df1
  letter  number
0      a       1
1      b       2
>>> df2 = pd.DataFrame([['c', 3], ['d', 4]],
...                    columns=['letter', 'number'])
>>> df2
  letter  number
0      c       3
1      d       4
>>> pd.concat([df1, df2])
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4
```