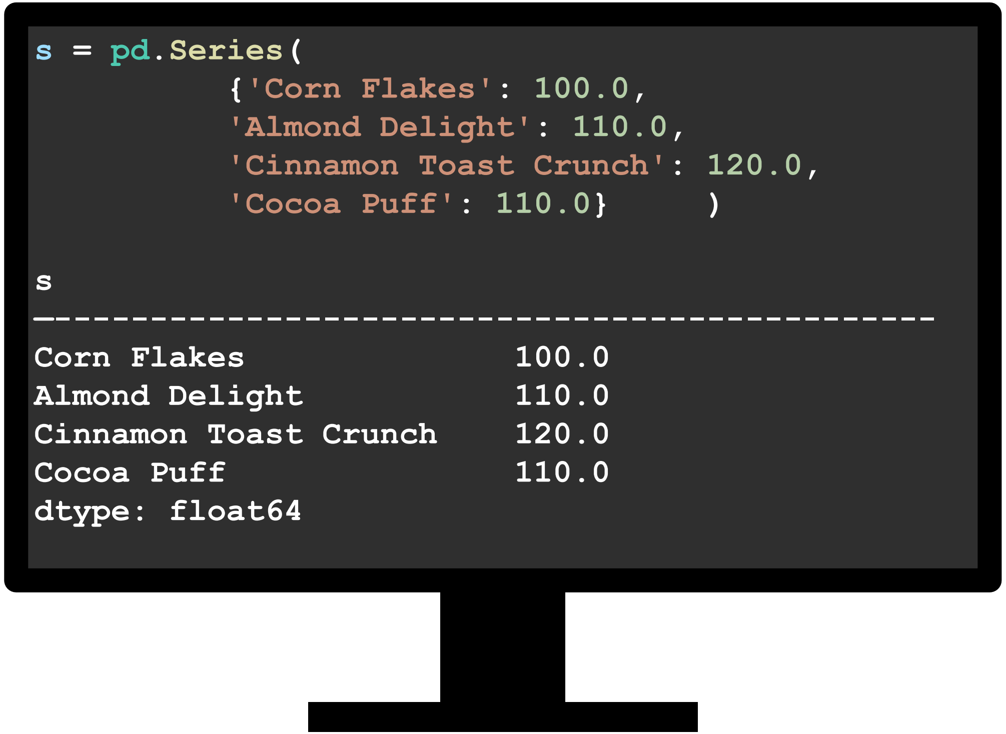
pandas

Pandas

- **Pandas** is a python package containing data structures to make working with labeled data intuitive and easy
  - The two primary data structures are ***Series*** and ***DataFrames***
- The **Pandas** package also contains several useful functions for
  - **Plotting data**
  - **Manipulating data structures**
  - **Data manipulation**

Series

- ➔ Series is one of the primary data structures offered in Pandas
  - ○ They are a one dimensional labeled array
  - ○ They can hold any data type (e.x. string, int, python objects, etc)
- ➔ They can be created from ***array-like objects***, ***iterable objects***, ***dictionaries***, ***scalar values***



```
s = pd.Series(
          {'Corn Flakes': 100.0,
          'Almond Delight': 110.0,
          'Cinnamon Toast Crunch': 120.0,
          'Cocoa Puff': 110.0}     )


s
-----------------------------------------------
Corn Flakes                 100.0
Almond Delight              110.0
Cinnamon Toast Crunch       120.0
Cocoa Puff                  110.0
dtype: float64
```

Constructor

pandas.Series(*data=None, index=None, dtype=None, name=None*)

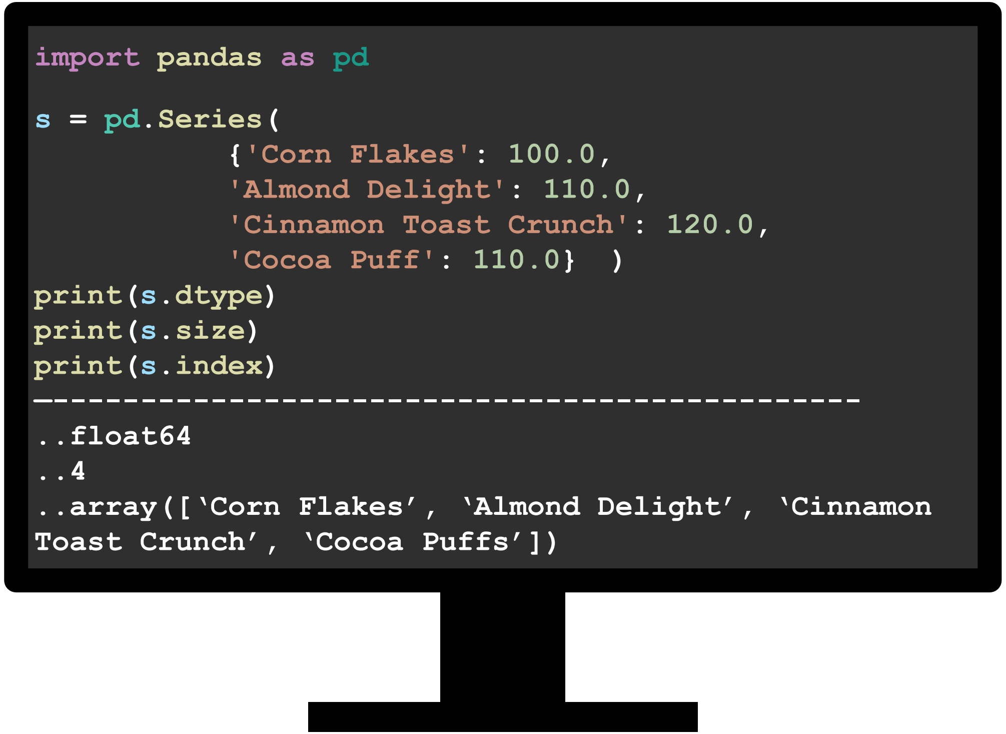
Parameter	Expected value
data	`array-like, Iterable, dict, or scalar value`
index	`array-like or Index (1d)`
dtype	`str, numpy.dtype, or ExtensionDtype,`
name	`str`

Available dtypes

Pandas dtype	Python Type	NumPy type
Object	Str or mixed values	String_, unicode_, mixed types
int64	int	int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64
float64	float	float_, float16, float32, float64
bool	bool	bool_
datetime64	N/A	datetime64[ns]
timedelta[ns]	N/A	N/A
category	N/A	N/A

Series Attributes

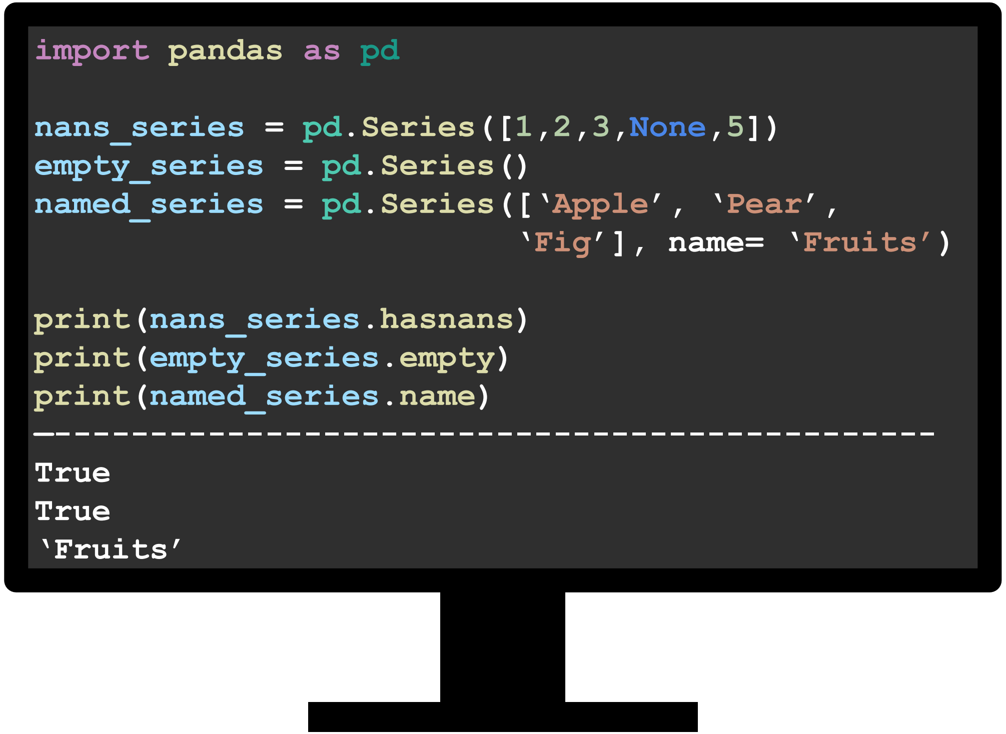
- ➔ **Series.dtype**
  - ○ Returns the overall data type of the underlying object
- ➔ **Series.size**
  - ○ Returns the number of elements in the underlying data
- ➔ **Series.index**
  - ○ Return an array containing the index



```
import pandas as pd

s = pd.Series(
            {'Corn Flakes': 100.0,
             'Almond Delight': 110.0,
             'Cinnamon Toast Crunch': 120.0,
             'Cocoa Puff': 110.0}  )
print(s.dtype)
print(s.size)
print(s.index)
-----------------------------------------------
..float64
..4
..array([‘Corn Flakes’, ‘Almond Delight’, ‘Cinnamon
Toast Crunch’, ‘Cocoa Puffs’])
```

Series Attributes

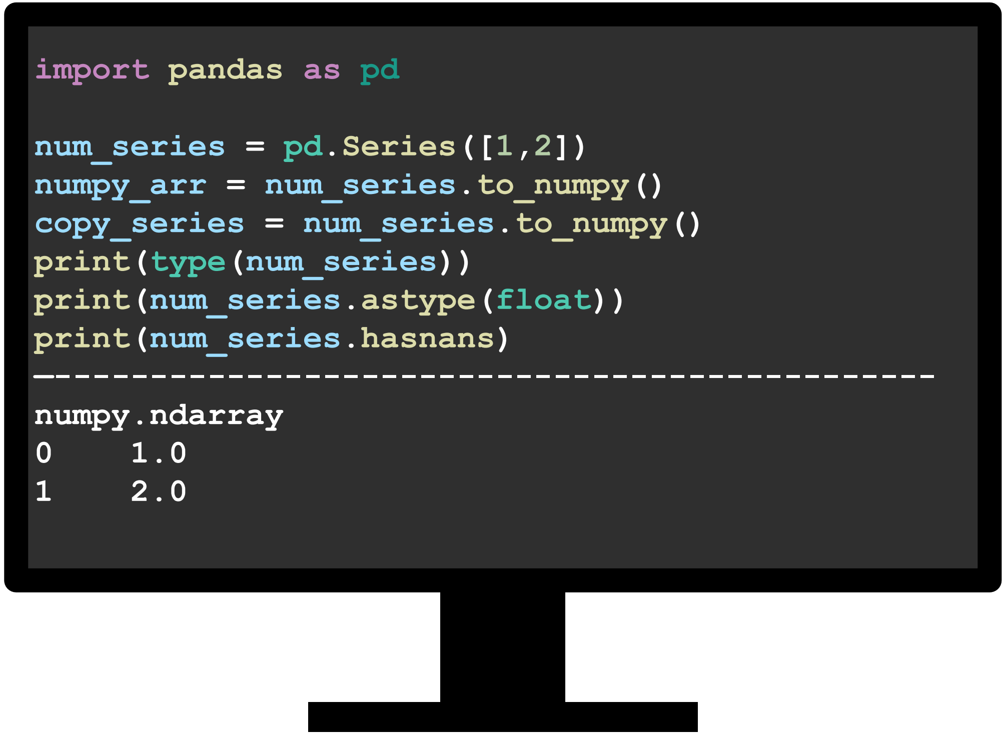
- ➔ **Series.hasnans**
  - ○ Returns True if there are nans in the series
- ➔ **Series.empty**
  - ○ Returns true if the Series is empty
- ➔ **Series.name**
  - ○ Returns the name of the Series (the column name)



```
import pandas as pd

nans_series = pd.Series([1,2,3,None,5])
empty_series = pd.Series()
named_series = pd.Series([‘Apple’, ‘Pear’,
                          ‘Fig’], name= ‘Fruits’)

print(nans_series.hasnans)
print(empty_series.empty)
print(named_series.name)
----------------------------------------------
True
True
‘Fruits’
```

Conversions

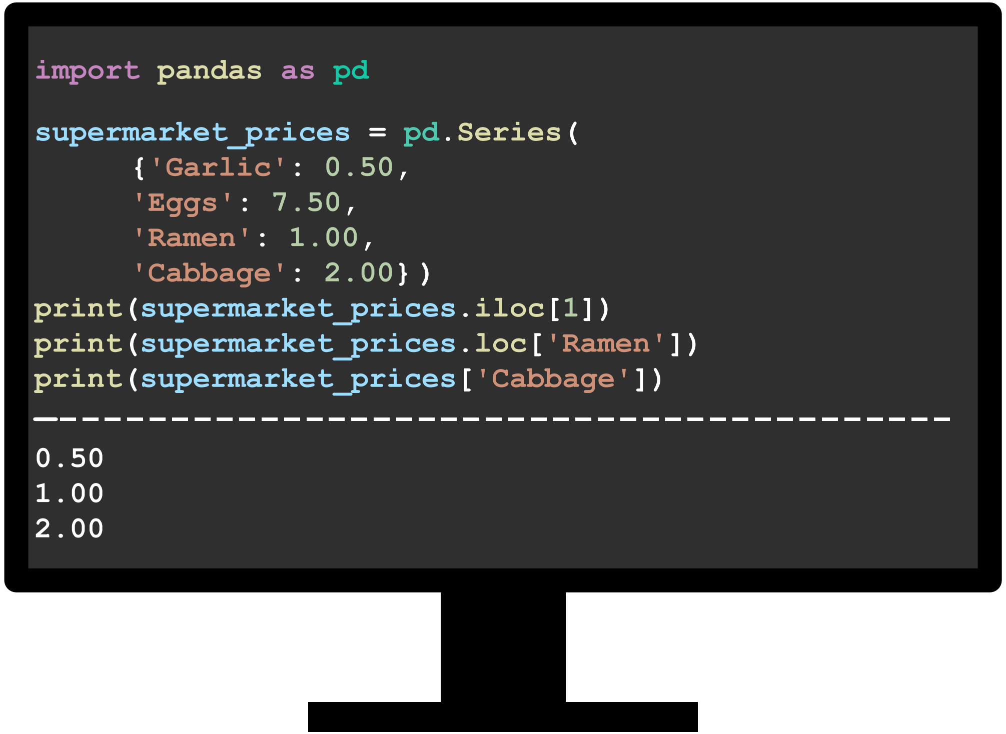
- ➔ **Series.to_numpy()**
  - ○ Will return an array containing the Series data
- ➔ **Series.astype(*dtype, copy=True, errors= ‘raise’*)**
  - ○ Returns a Series with data type cast to specified dtype
- ➔ **Series.copy()**
  - ○ Returns a deep copy of a Series



```
import pandas as pd

num_series = pd.Series([1,2])
numpy_arr = num_series.to_numpy()
copy_series = num_series.to_numpy()
print(type(num_series))
print(num_series.astype(float))
print(num_series.hasnans)
----------------------------------------------
numpy.ndarray
0     1.0
1     2.0
```

Viewing Series

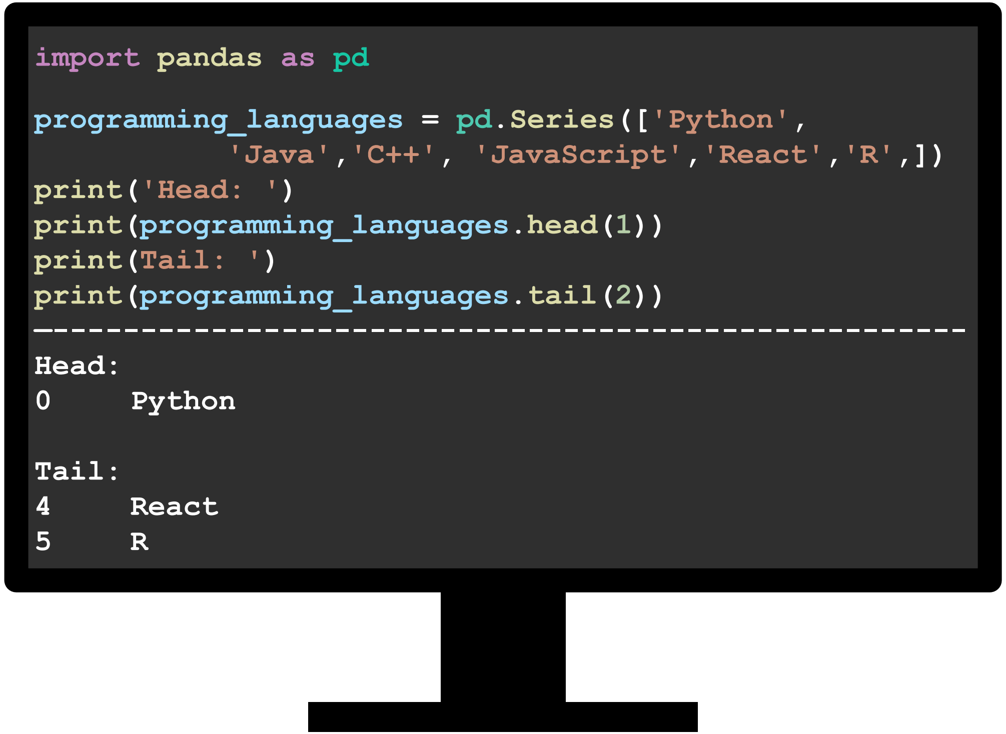
- **Series.iloc[*<index>*]**
  - Returns data at the specified index position
- **Series.loc[*<label>*]**
  - Returns data at the specified label. Uses index if there's no label
  - Is also aliased to **[*<label>*]**
- The data in a pandas series is mutable



```
import pandas as pd

supermarket_prices = pd.Series(
      {'Garlic': 0.50,
      'Eggs': 7.50,
      'Ramen': 1.00,
      'Cabbage': 2.00})
print(supermarket_prices.iloc[1])
print(supermarket_prices.loc['Ramen'])
print(supermarket_prices['Cabbage'])
----------------------------------------
0.50
1.00
2.00
```

Viewing Series

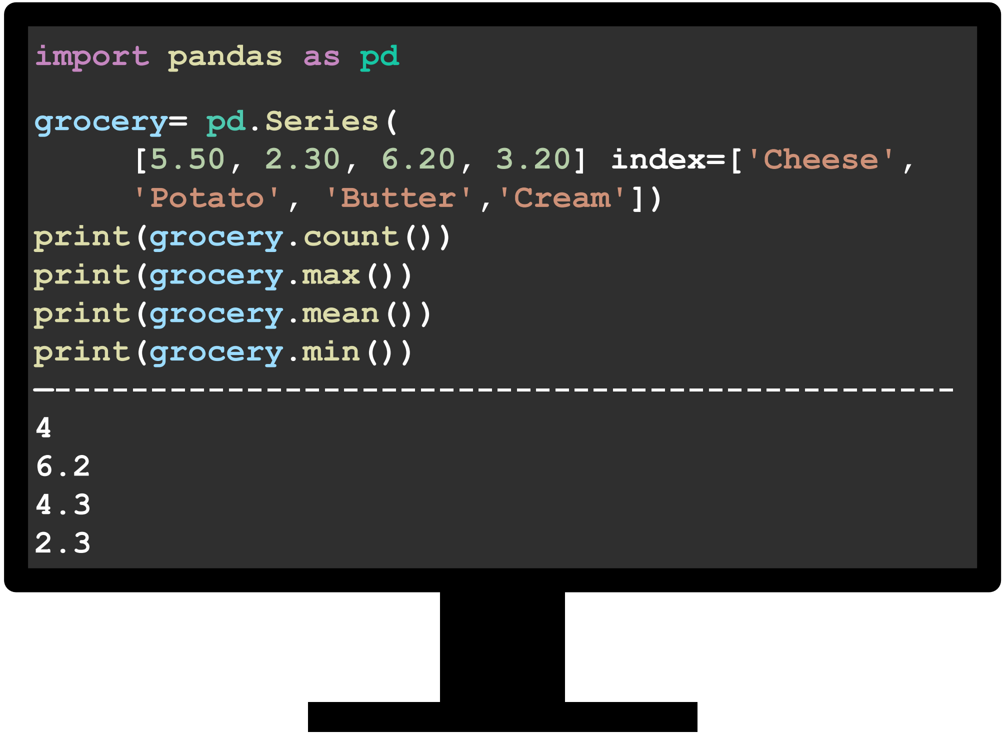
- ➔ **Series.head(*n*)**
  - Shows the first n rows
- ➔ **Series.tail(*n*)**
  - Shows the last n rows
- ➔ **Series[*series == condition*]**
  - We can also search a series based on a condition
  - Returns series whose elements fulfill condition



```
import pandas as pd

programming_languages = pd.Series(['Python',
               'Java','C++', 'JavaScript','React','R',])
print('Head: ')
print(programming_languages.head(1))
print(Tail: ')
print(programming_languages.tail(2))
--------------------------------------------------------
Head:
0      Python

Tail:
4      React
5      R
```

Math

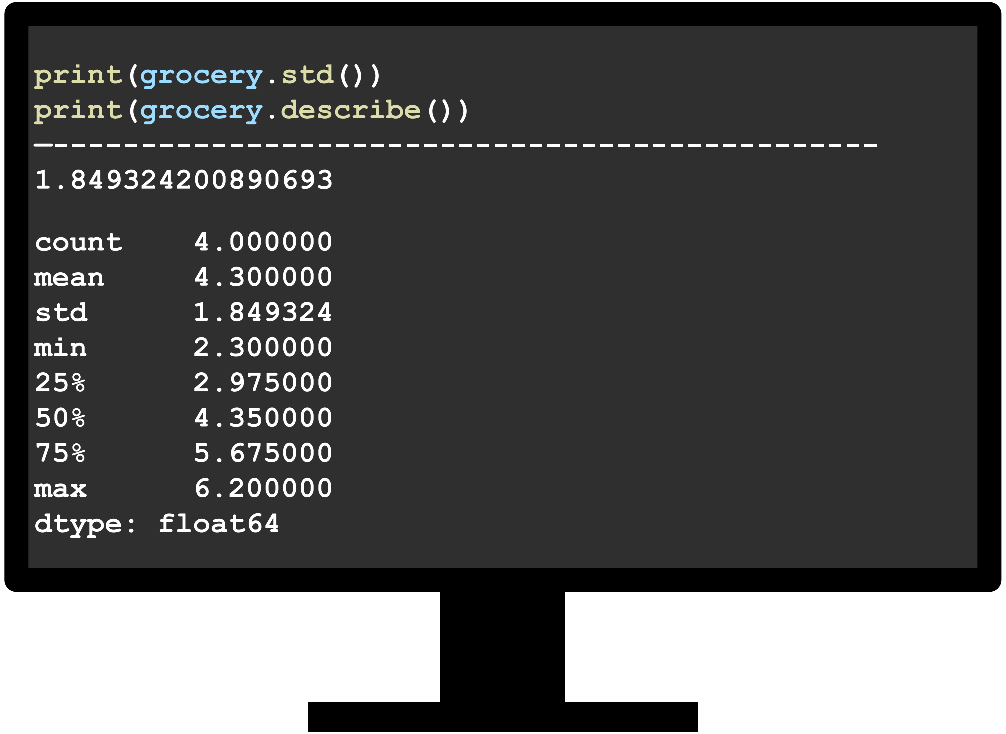
- ➔ **Series.count()**
  - Returns a count of non NA values
- ➔ **Series.max()**
  - Returns the largest value
- ➔ **Series.mean()**
  - Returns the average of the series
- ➔ **Series.min()**
  - Returns the minimum value



```
import pandas as pd

grocery= pd.Series(
     [5.50, 2.30, 6.20, 3.20] index=['Cheese',
     'Potato', 'Butter','Cream'])
print(grocery.count())
print(grocery.max())
print(grocery.mean())
print(grocery.min())
-----------------------------------------------
4
6.2
4.3
2.3
```

Math

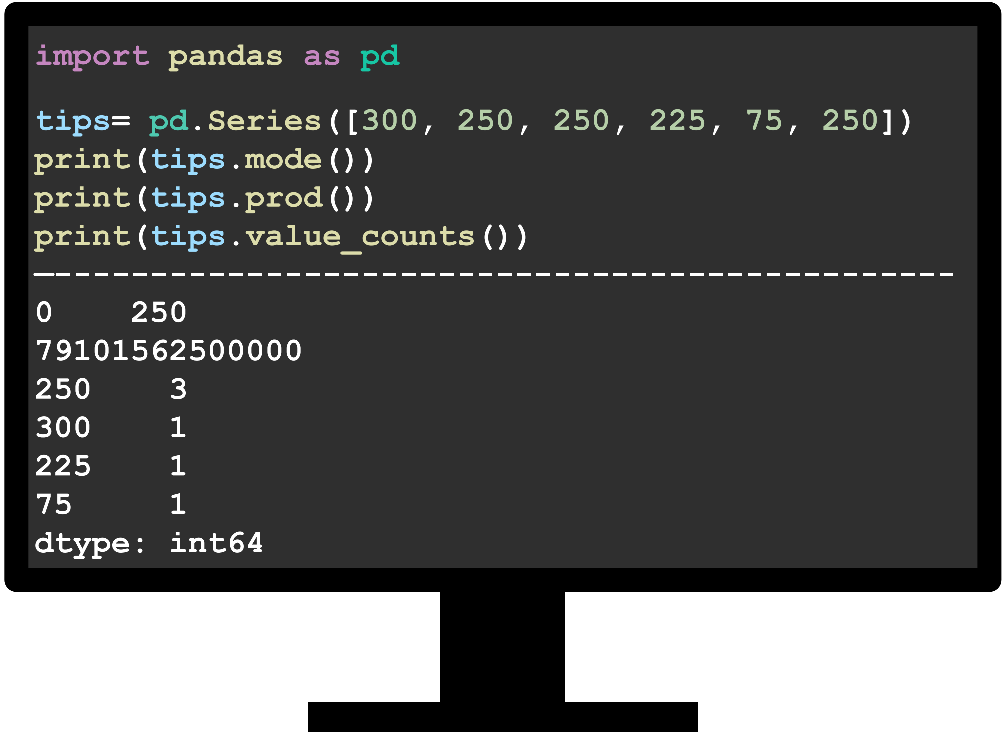
- **Series.std()**
  - Returns the mean of the series
- **Series.describe()**
  - Returns the max, min, count, mean and std in a series
- **Series.abs()**
  - Returns the absolute value of a series



```
print(grocery.std())
print(grocery.describe())
-----------------------------------------------
1.849324200890693

count     4.000000
mean      4.300000
std       1.849324
min       2.300000
25%       2.975000
50%       4.350000
75%       5.675000
max       6.200000
dtype: float64
```

Math

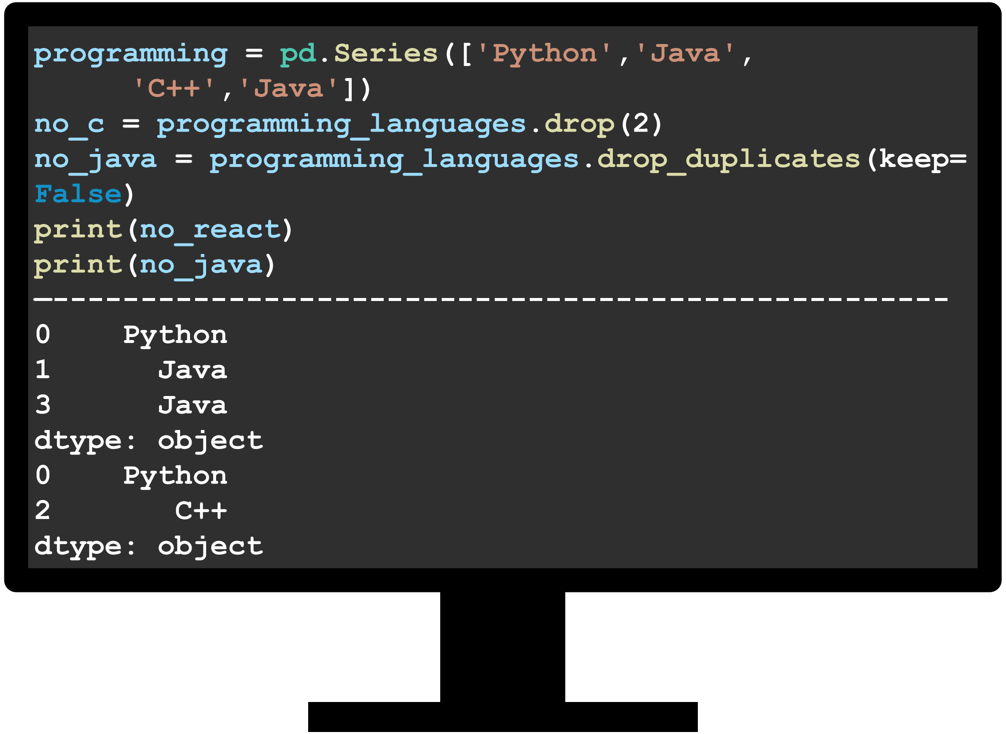
- **Series.mode()**
  - Returns a Series containing the mode(s)
- **Series.prod()**
  - Returns the product of the Series
- **Series.value_counts()**
  - Returns a Series containing a count of each unique value



```
import pandas as pd

tips= pd.Series([300, 250, 250, 225, 75, 250])
print(tips.mode())
print(tips.prod())
print(tips.value_counts())
------------------------------------------------
0    250
79101562500000
250    3
300    1
225    1
75     1
dtype: int64
```

Manipulation

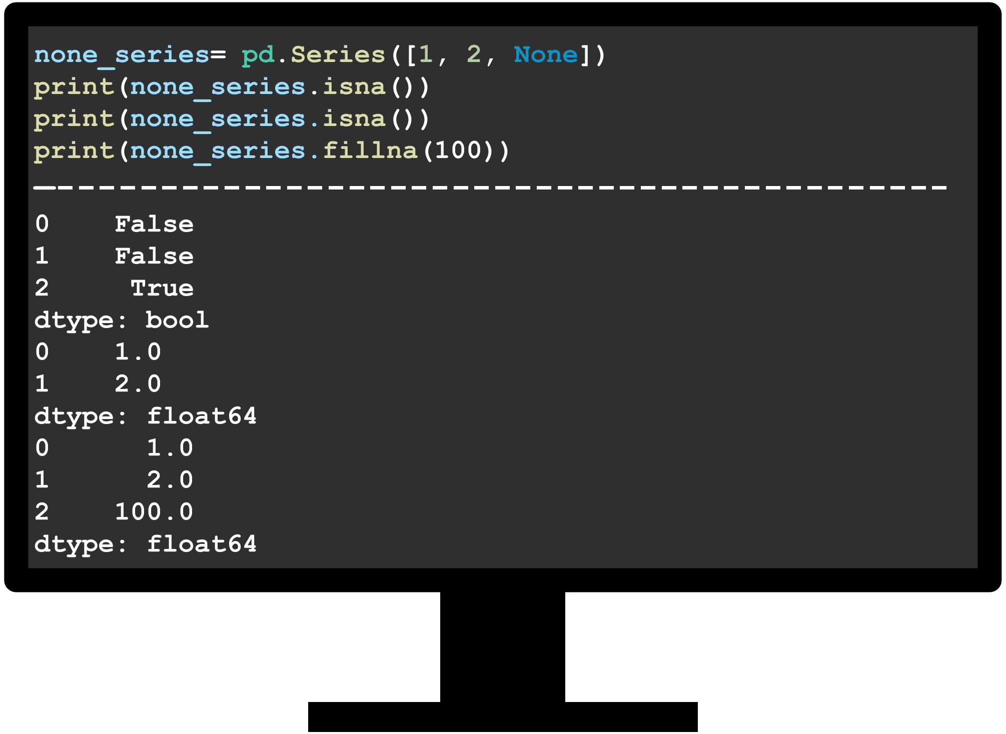
- ➔ Series.drop(*labels=None, inplace= False*)
  - ○ Returns a Series with specified index labels removed
- ➔ Series.drop_duplicates(*keep= ’first’, inplace= False*)
  - ○ Returns Series with duplicate values removed
  - ○ Choices for keep parameter are first, last, or False (removes all duplicates)



```
programming = pd.Series(['Python','Java',
      'C++','Java'])
no_c = programming_languages.drop(2)
no_java = programming_languages.drop_duplicates(keep=
False)
print(no_react)
print(no_java)
-----------------------------------------------------
0     Python
1       Java
3       Java
dtype: object
0     Python
2        C++
dtype: object
```

Manipulation

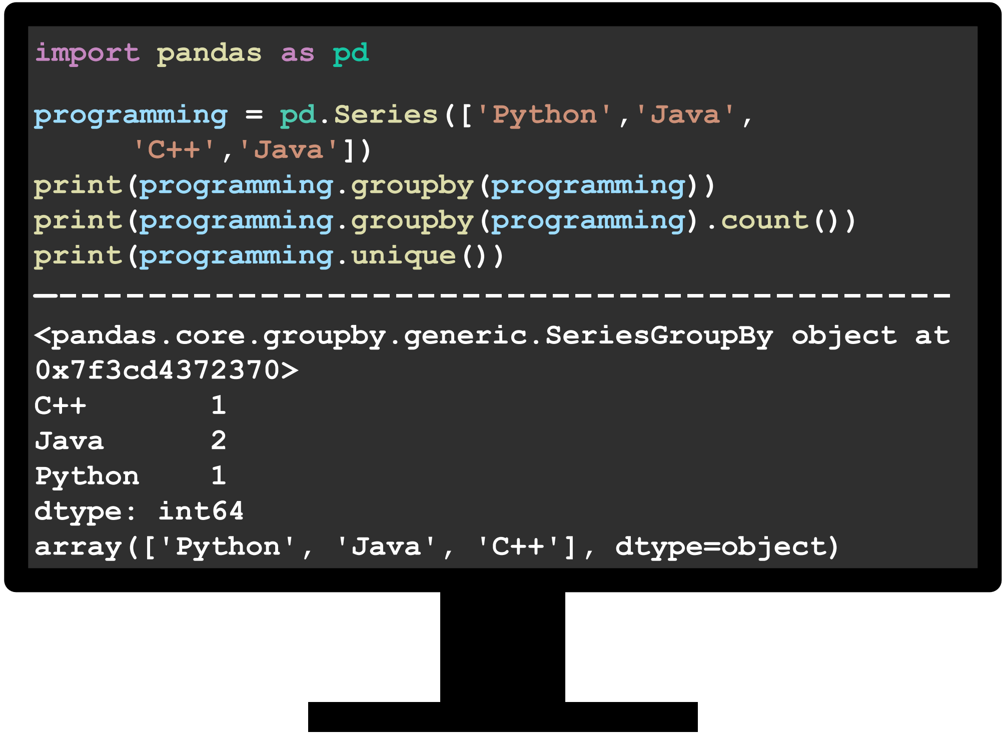
- ➔ Series.isna()
  - ○ Returns a Series of booleans mapping if a value is missing
- ➔ Series.dropna(*inplace= False*)
  - ○ Returns a Series with all missing values removed
- ➔ Series.fillna(*value=None, inplace= False*)
  - ○ Returns a Series with all missing values replaced with ‘value’



```
none_series= pd.Series([1, 2, None])
print(none_series.isna())
print(none_series.isna())
print(none_series.fillna(100))
―――――――――――――――――――――――――――――――――――――――――――――――――――――――
0     False
1     False
2      True
dtype: bool
0     1.0
1     2.0
dtype: float64
0       1.0
1       2.0
2     100.0
dtype: float64
```

Manipulation

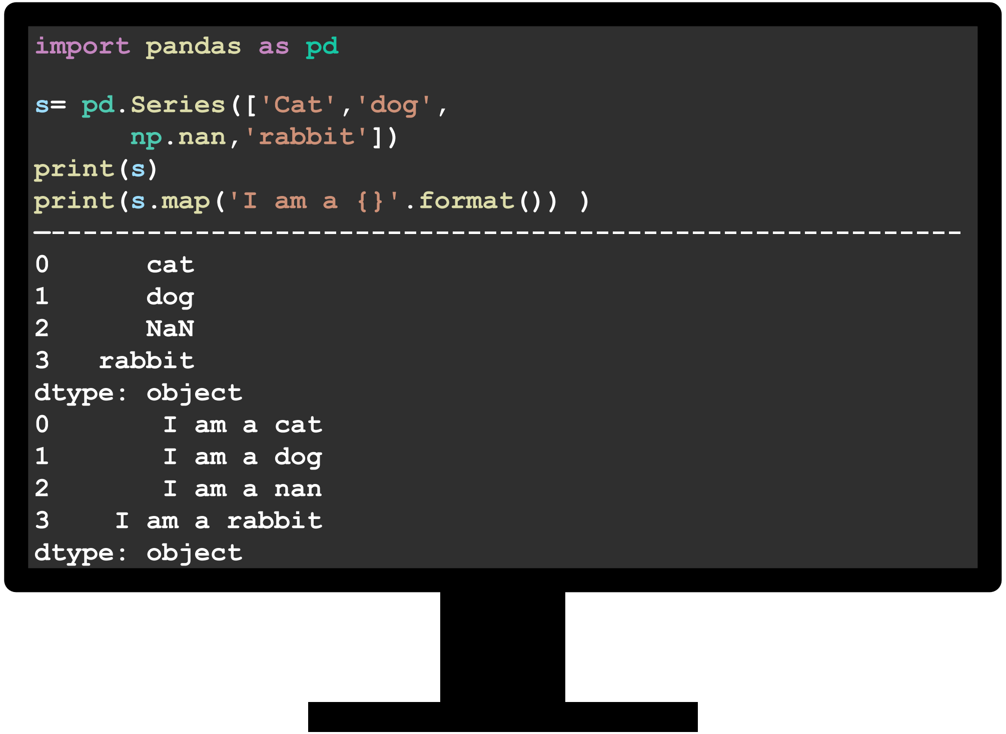
- ➔ Series.groupby(*by=None, dropna=True*)
  - If Series is multi Indexed, specify the index in by, otherwise we pass series name
  - Returns groupby object that is waiting for a math function
- ➔ Series.unique()
  - Returns a numpy array of unique values



```
import pandas as pd

programming = pd.Series(['Python','Java',
      'C++','Java'])
print(programming.groupby(programming))
print(programming.groupby(programming).count())
print(programming.unique())
----------------------------------------------------
<pandas.core.groupby.generic.SeriesGroupBy object at
0x7f3cd4372370>
C++       1
Java      2
Python    1
dtype: int64
array(['Python', 'Java', 'C++'], dtype=object)
```

Manipulation

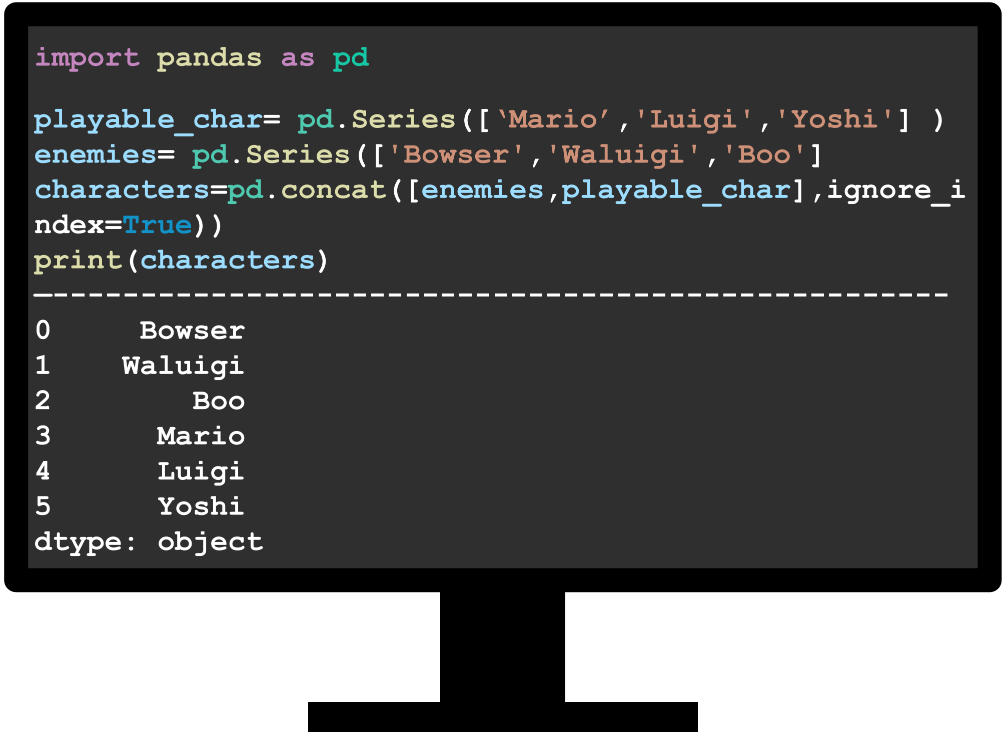
- ➔ **Series.map(*arguments*)**
  - Used to substitute each value in a Series with another value, that is derived from a function, dictionary, or Series.
  - A passed function will be used to edit each value
  - A passed dictionary or Series replaces values based on the key value pairs



```
import pandas as pd

s= pd.Series(['Cat','dog',
      np.nan,'rabbit'])
print(s)
print(s.map('I am a {}'.format()) )
----------------------------------------------------------
0      cat
1      dog
2      NaN
3   rabbit
dtype: object
0       I am a cat
1       I am a dog
2       I am a nan
3    I am a rabbit
dtype: object
```

Manipulation

- ➔ Pandas.concat(*objs**, *ignore_index=False*)
  - Combines 2 or more Pandas objects by stacking them
  - Pandas objects are joined in the order passed
  - If ignore_index is set to True, a new index is created



```
import pandas as pd

playable_char= pd.Series([‘Mario’,'Luigi','Yoshi'] )
enemies= pd.Series(['Bowser','Waluigi','Boo']
characters=pd.concat([enemies,playable_char],ignore_i
ndex=True))
print(characters)
---------------------------------------------------
0     Bowser
1    Waluigi
2        Boo
3      Mario
4      Luigi
5      Yoshi
dtype: object
```

Series I/O functions for Series

Series Function	Function Definition
Series.to_csv(path=None)	Write object to a csv file.
Series.to_json(path=None)	Convert the object to a JSON string.
Series.to_pickle(path)	Compresses a file and writes it to memory
Series.to_list()	Returns a list of the values
Series.to_dict()	Convert Series to {label -> value} dict or dict-like object.

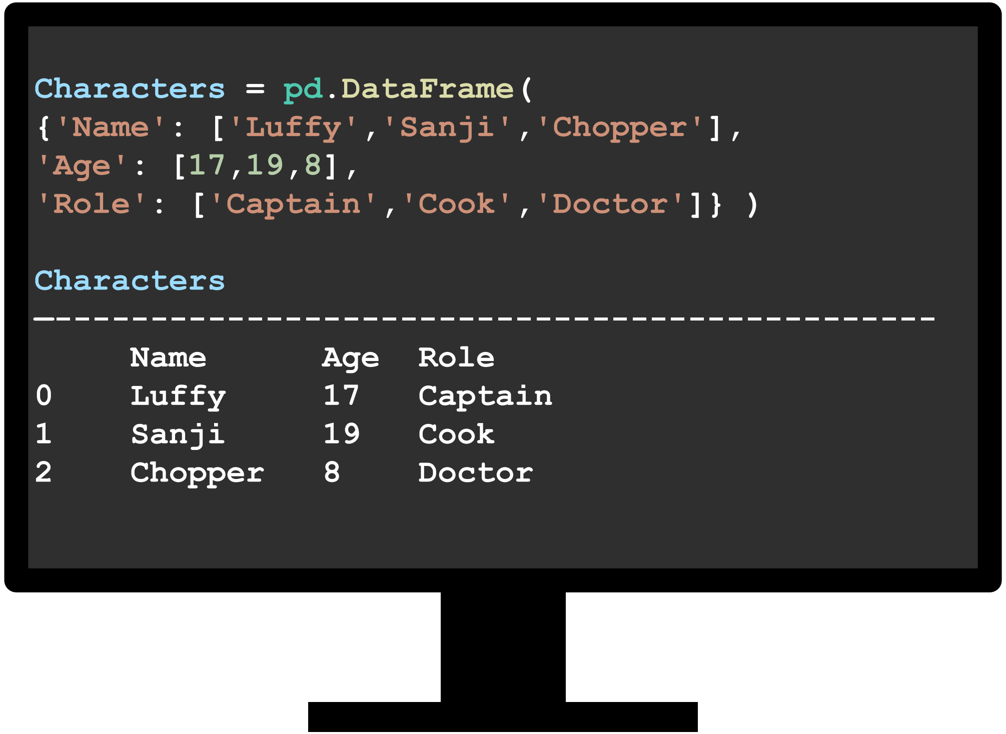
Student Exercise

- ➔ Create the following Pandas Series
  - ◆ Employee ID: type int
  - ◆ Employee Name: type str
  - ◆ Employee Salary: type int
  - ◆ Employee Department: type str
  - ◆ Employee Start Date: type datetime
  - ◆ Currently Employed: type bool
- ➔ Populate each Series with an equal number of entries
- ➔ Set the Index of the other series to be the Employee id

DataFrame

DataFrame

- ➔ DataFrame is the other primary data structures offered in Pandas
  - ○ They are 2 dimensional tables
  - ○ Each column in a DataFrame is a Series
- ➔ They can be created from ***array-like objects***, ***iterable objects***, ***dictionaries***, or ***DataFrame***
- ➔ Many Series functions will also work on DataFrame objects



```
Characters = pd.DataFrame(
{'Name': ['Luffy','Sanji','Chopper'],
'Age': [17,19,8],
'Role': ['Captain','Cook','Doctor']} )

Characters
------------------------------------------------

     Name       Age  Role
0    Luffy      17   Captain
1    Sanji      19   Cook
2    Chopper    8    Doctor
```

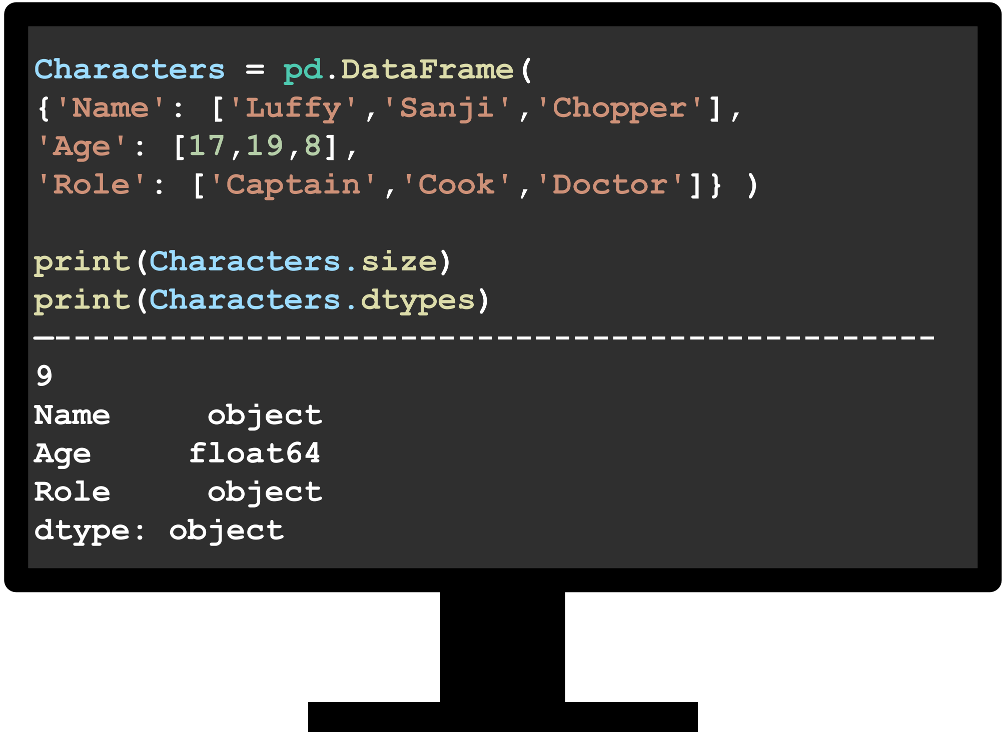
Constructor

```
pd.DataFrame(data=None, index=None, columns=None, dtype=None)
```

Parameter	Expected value
data	Array-like, iterable, dict, or a DataFrame. Dictionary keys are column names
index	array-like or Index (1d)
columns	Labels to use for column names
dtype	Data type to enforce.Can only choose 1 dtype

Attributes

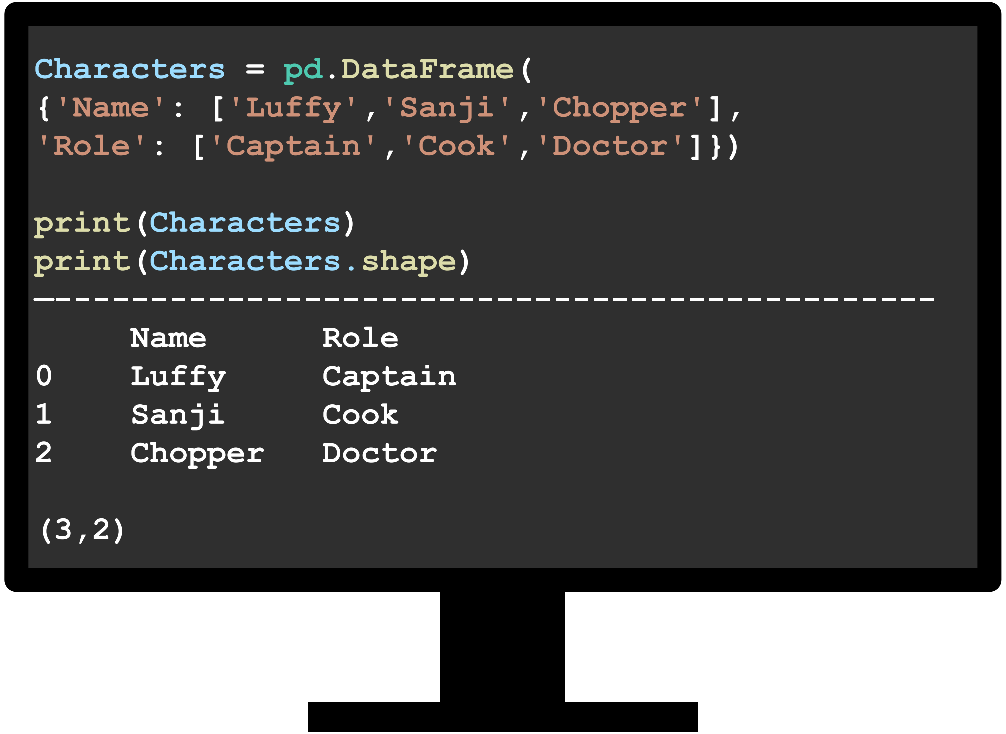
- ➔ Just like in Series, we can see different DataFrame attributes including
  - ○ DataFrame.size
  - ○ DataFrame.index
  - ○ DataFrame.empty
- ➔ We can also get the data types of each column using
  - ○ DataFrame.dtypes



```
Characters = pd.DataFrame(
{'Name': ['Luffy','Sanji','Chopper'],
'Age': [17,19,8],
'Role': ['Captain','Cook','Doctor']} )

print(Characters.size)
print(Characters.dtypes)
-----------------------------------------------
9
Name      object
Age      float64
Role      object
dtype: object
```

Attributes

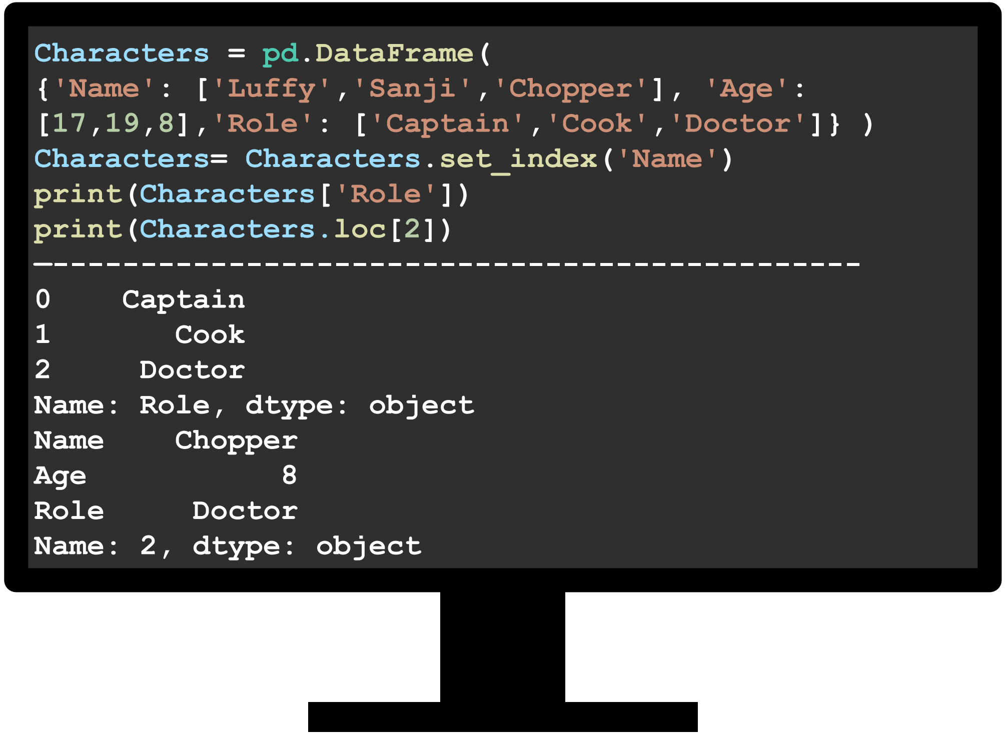
- ➔ We can get the the shape, or how many rows and columns are in a dataframe with
  - ○ DataFrame.shape
- ➔ Shape is returned in tuple with rows listed first and columns listed second



```
Characters = pd.DataFrame(
{'Name': ['Luffy','Sanji','Chopper'],
'Role': ['Captain','Cook','Doctor']})

print(Characters)
print(Characters.shape)
------------------------------------------------
     Name     Role
0    Luffy    Captain
1    Sanji    Cook
2    Chopper  Doctor

(3,2)
```

Indexing

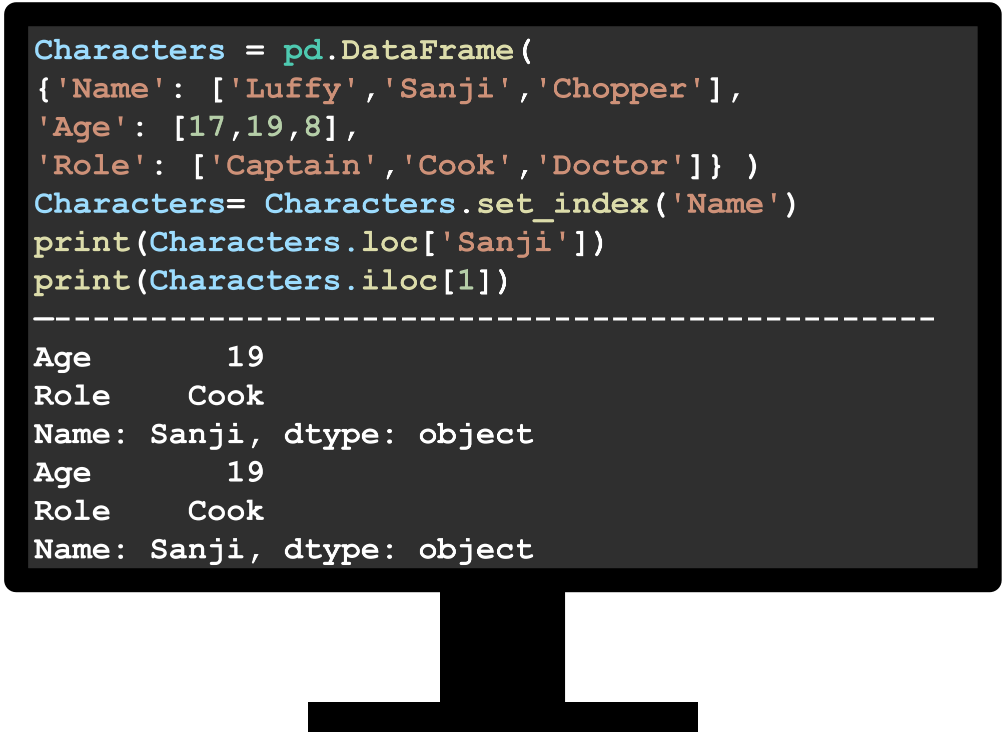
- ➔ To select a column as a whole, use the command
  - ○ df[*<column label>*]
  - ○ We can also use **df.column**
- ➔ To select particular rows by their labels, use
  - ○ df.loc[*<label>*]
  - ○ The labels passed can be a single label, a list of labels, a slice object with label, or a boolean



```
Characters = pd.DataFrame(
{'Name': ['Luffy','Sanji','Chopper'], 'Age':
[17,19,8],'Role': ['Captain','Cook','Doctor']} )
Characters= Characters.set_index('Name')
print(Characters['Role'])
print(Characters.loc[2])
-----------------------------------------------
0    Captain
1       Cook
2     Doctor
Name: Role, dtype: object
Name    Chopper
Age           8
Role     Doctor
Name: 2, dtype: object
```

Indexing

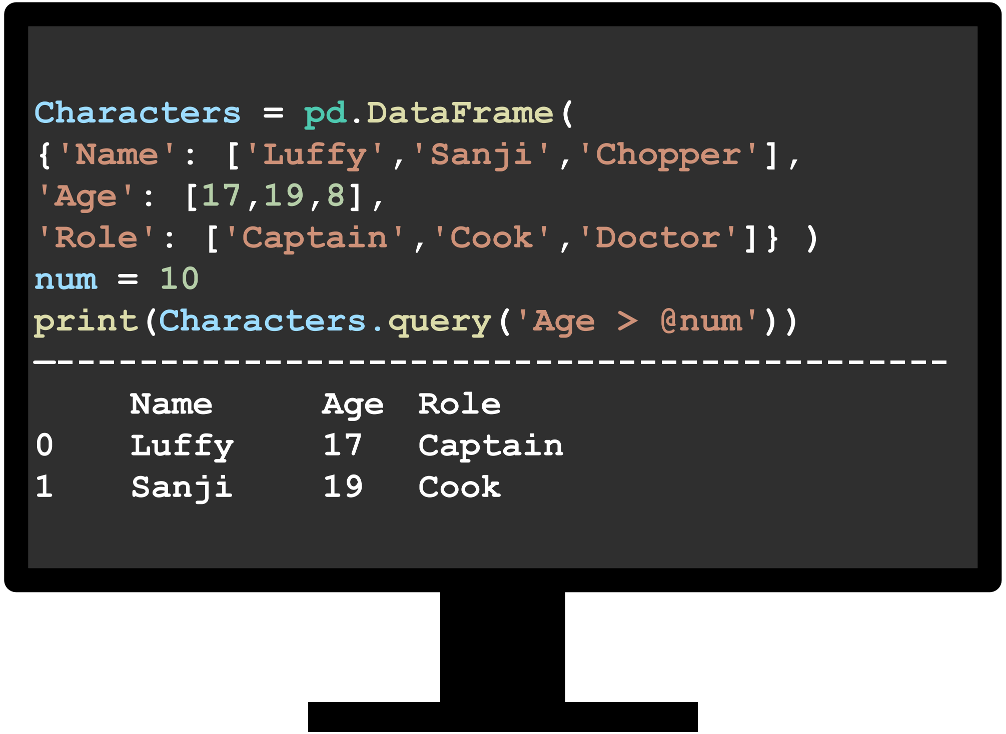
- ➔ To select particular rows by their index position, use
  - ○ df.iloc[*< index>*]
  - ○ Can be passed a integer, list of integers, slice object with integers, or boolean
- ➔ Both loc and iloc will raise errors if a label or index is not found



```
Characters = pd.DataFrame(
{'Name': ['Luffy','Sanji','Chopper'],
'Age': [17,19,8],
'Role': ['Captain','Cook','Doctor']} )
Characters= Characters.set_index('Name')
print(Characters.loc['Sanji'])
print(Characters.iloc[1])
-----------------------------------------------
Age       19
Role    Cook
Name: Sanji, dtype: object
Age       19
Role    Cook
Name: Sanji, dtype: object
```

Indexing

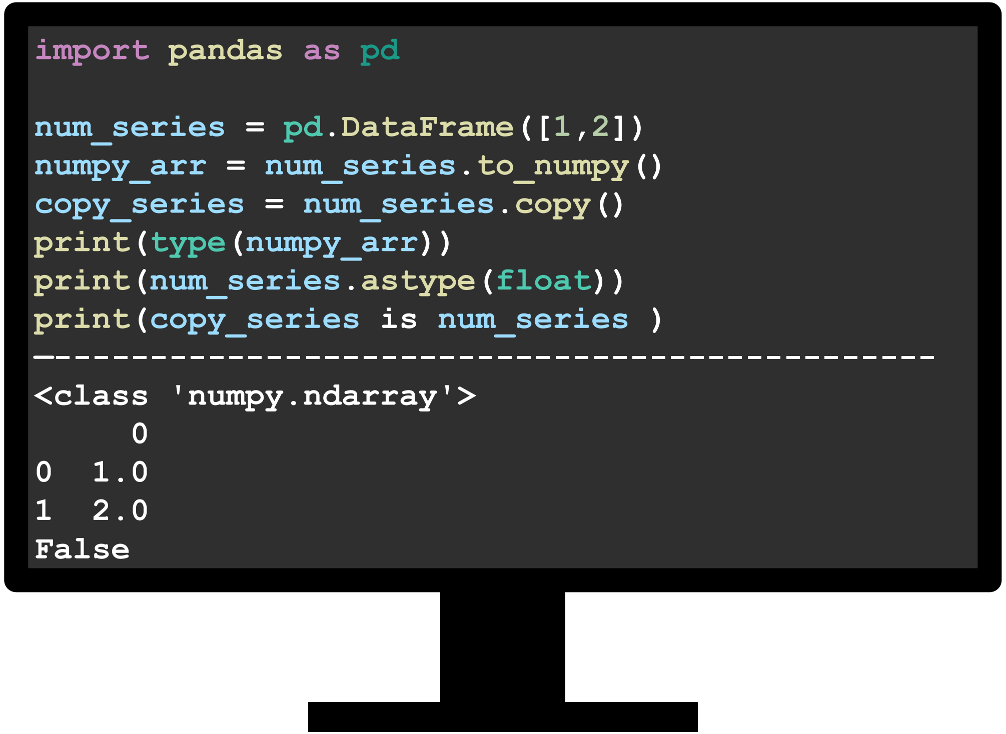
- ➔ We can also use DataFrame.query to index particular rows of data
  - ○ DataFrame.query(*expression, inplace=False*)
- ➔ You can refer to variables in the environment by prefixing them with an ‘@’ character like @a + b



```
Characters = pd.DataFrame(
{'Name': ['Luffy','Sanji','Chopper'],
'Age': [17,19,8],
'Role': ['Captain','Cook','Doctor']} )
num = 10
print(Characters.query('Age > @num'))
--------------------------------------------
     Name     Age  Role
0    Luffy    17   Captain
1    Sanji    19   Cook
```

Conversions

- ➔ We can also call and apply these functions across DataFrames
  - ○ DataFrame.to_numpy()
  - ○ DataFrame.copy()
- ➔ We can change each individual column’s datatype by passing a dictionary to
  - ○ DataFrame.astype({*column_name: data type*})



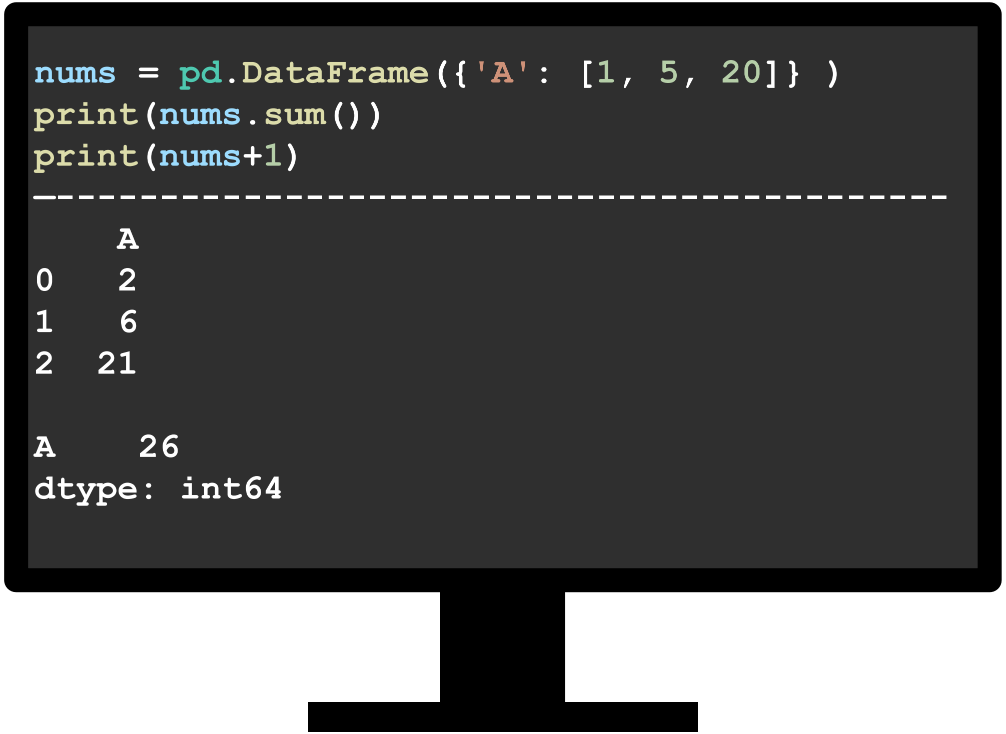
```
import pandas as pd

num_series = pd.DataFrame([1,2])
numpy_arr = num_series.to_numpy()
copy_series = num_series.copy()
print(type(numpy_arr))
print(num_series.astype(float))
print(copy_series is num_series )
-----------------------------------------------
<class 'numpy.ndarray'>
     0
0  1.0
1  2.0
False
```

Student Exercise

- ➔ Use the Employee Series you created to make a DataFrame
- ➔ This DataFrame should have the Employee ID as its index, and the other Series columns
- ➔ Try different methods of creating your DataFrame!
  - ◆ Appending Series to an existing DF
  - ◆ Creating from a list of Series
  - ◆ Creating from a list of dictionaries
  - ◆ Creating from a dictionary of Series
- ➔ Access data from your DataFrame using the accessor methods

Math

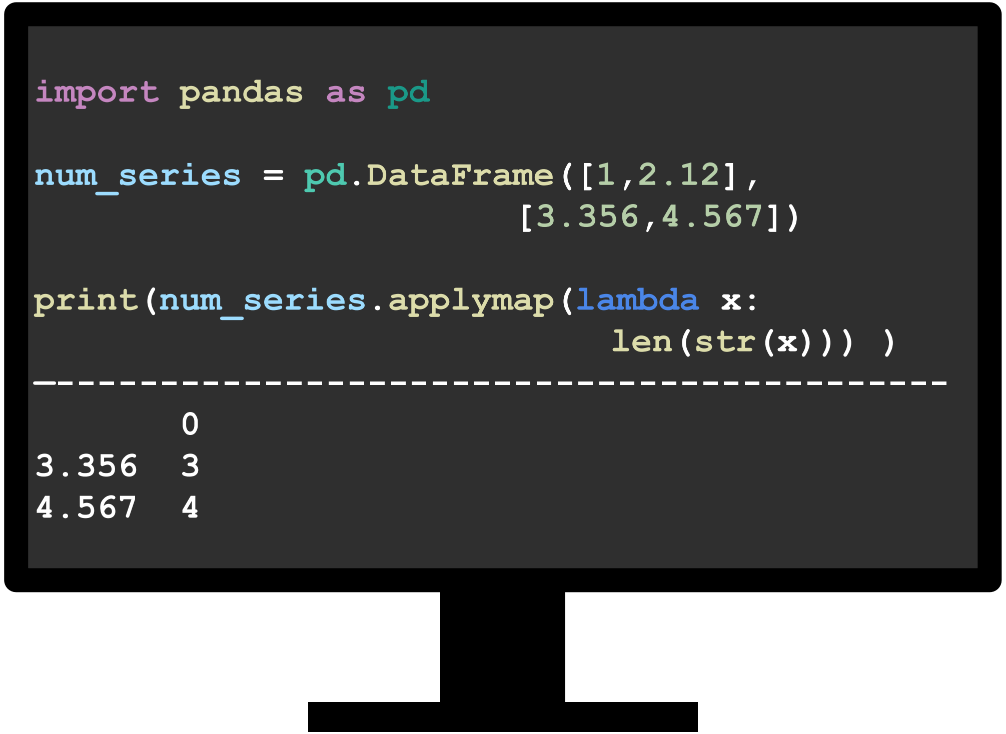
- ➔ All Series math functions can also be applied across **DataFrame** as a whole
  - ○ Both DataFrame and Series also support binary operators



```
nums = pd.DataFrame({'A': [1, 5, 20]} )
print(nums.sum())
print(nums+1)
--------------------------------------------
    A
0   2
1   6
2  21

A    26
dtype: int64
```

Changing Data

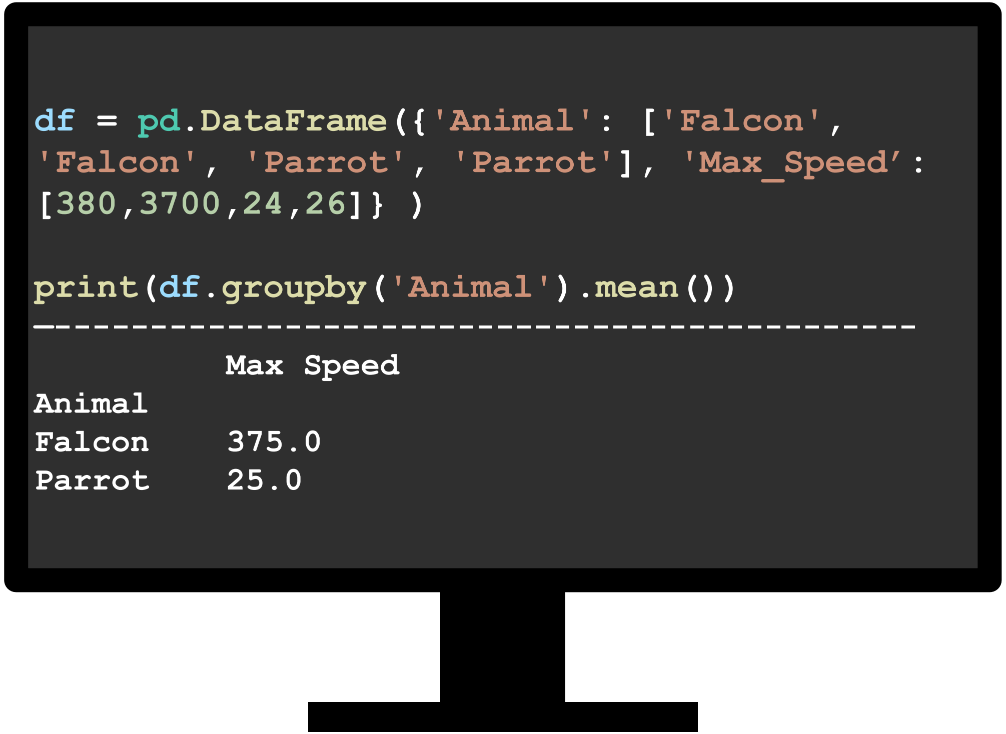
- We can change each individual element of a DataFrame with
  - DataFrame.applymap(*Func*)
  - The passed function must be callable
  - The function will be applied to element
  - A **DataFrame** is returned



```
import pandas as pd

num_series = pd.DataFrame([1,2.12],
                        [3.356,4.567])

print(num_series.applymap(lambda x:
                            len(str(x))) )
--------------------------------------------
       0
3.356  3
4.567  4
```

Changing Data

- ➔ We can group data exactly like how we group data in SQL
  - ○ DataFrame.groupby(*by= None*)
- ➔ The groupby function returns a ***DataFrameGroupBy*** object which contains information about the groups
- ➔ Must use some math or aggregate function to view data as DataFrame

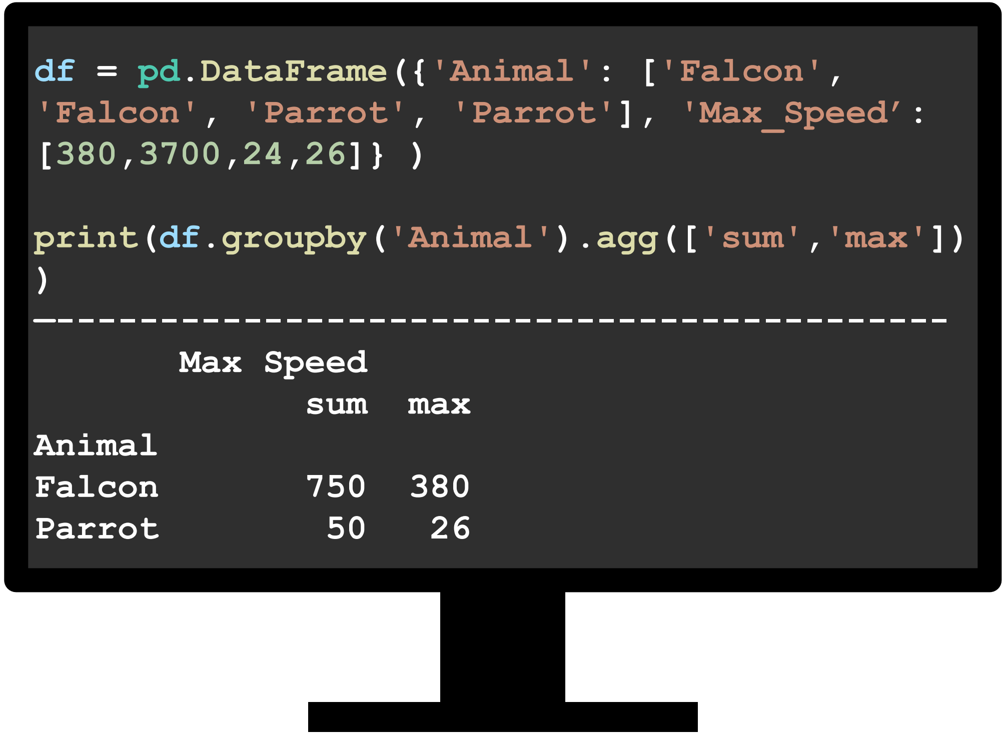


```
df = pd.DataFrame({'Animal': ['Falcon',
'Falcon', 'Parrot', 'Parrot'], 'Max_Speed’:
[380,3700,24,26]} )

print(df.groupby('Animal').mean())
------------------------------------------
          Max Speed
Animal
Falcon     375.0
Parrot     25.0
```

Changing Data

We can also use an aggregate function in order to use one or more operations across a specified axis or grouped data

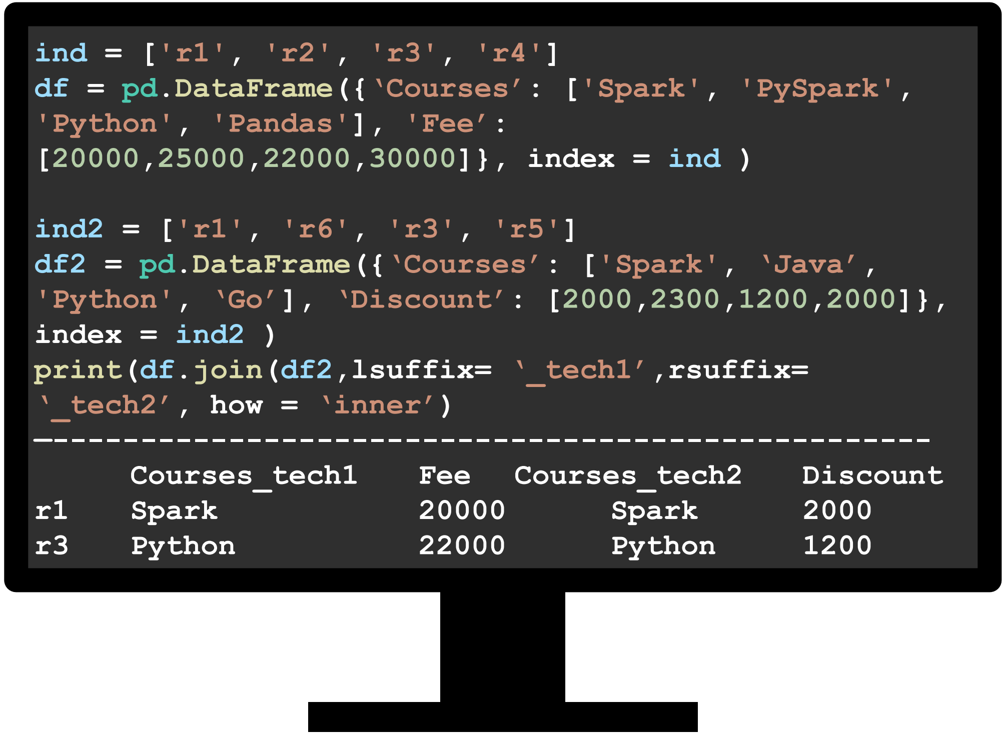
- ➔ DataFrame.agg(*function= None, axis=0*)
- ➔ DataFrame.GroupBy.agg(*function= None*)
- ➔ .agg accepts functions, function names, a list of functions/function names, or a dictionary of axis labels and function names



```
df = pd.DataFrame({'Animal': ['Falcon',
'Falcon', 'Parrot', 'Parrot'], 'Max_Speed’:
[380,3700,24,26]} )

print(df.groupby('Animal').agg(['sum','max'])
)
-----------------------------------------------
        Max Speed
              sum  max
Animal
Falcon        750  380
Parrot         50   26
```

Combining Data

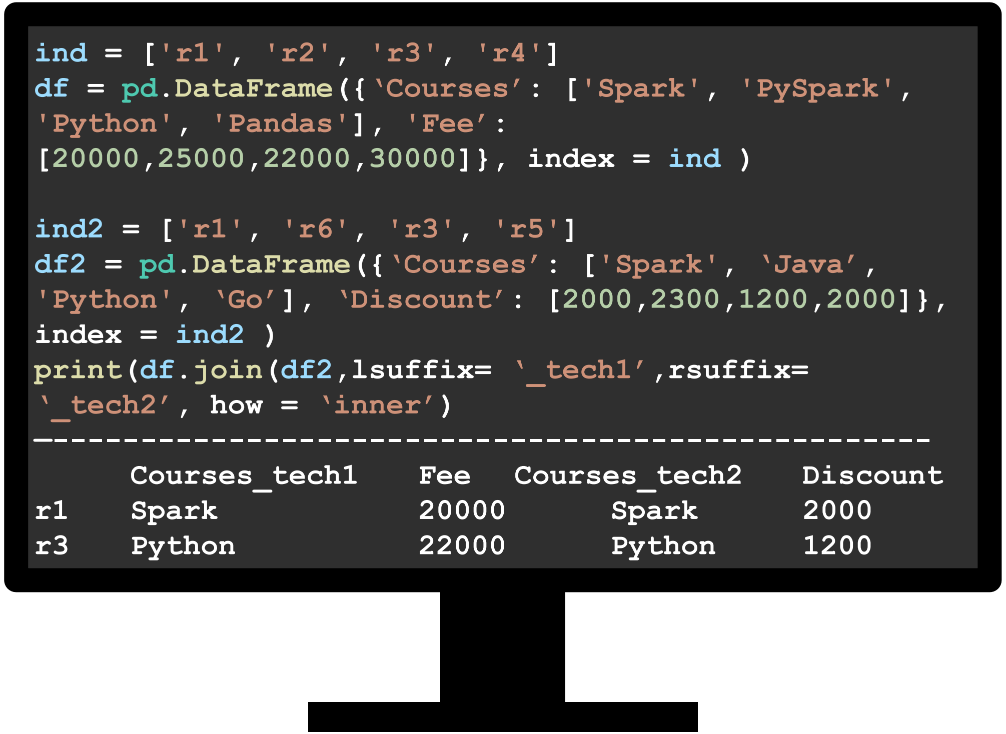
- ➔ We can combine the columns of 2 DataFrame using the following
  - ○ DataFrame.join(*other, on=None, how='left', lsuffix='', rsuffix=''*)
  - ○ This acts similar to SQL join, joining the columns of one dataframe with another dataframe on the index



```
ind = ['r1', 'r2', 'r3', 'r4']
df = pd.DataFrame({‘Courses’: ['Spark', 'PySpark',
'Python', 'Pandas'], 'Fee’:
[20000,25000,22000,30000]}, index = ind )

ind2 = ['r1', 'r6', 'r3', 'r5']
df2 = pd.DataFrame({‘Courses’: ['Spark', ‘Java’,
'Python', ‘Go’], ‘Discount’: [2000,2300,1200,2000]},
index = ind2 )
print(df.join(df2,lsuffix= ‘_tech1’,rsuffix=
‘_tech2’, how = ‘inner’))
----------------------------------------------------
     Courses_tech1    Fee   Courses_tech2    Discount
r1    Spark           20000       Spark       2000
r3    Python          22000       Python      1200
```

Combining Data

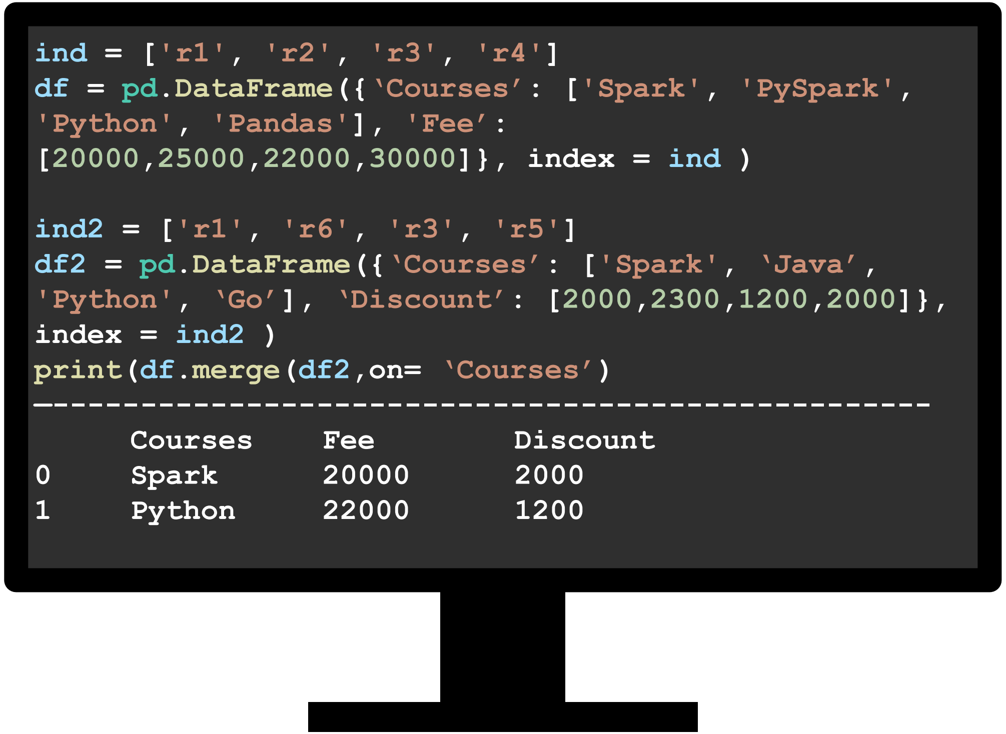
- → We can pass a list of DataFrames
- → When we join 2 DataFrames, use the parameters ***‘lsuffix’*** and ***‘rsuffix’*** to mark which DataFrame columns originally came from
- → We can also specify the type of join we’d like including ***‘left’***, ***‘right’***, ***‘outer’***, and ***‘inner’***, but it will default to ***inner***



```
ind = ['r1', 'r2', 'r3', 'r4']
df = pd.DataFrame({‘Courses’: ['Spark', 'PySpark',
'Python', 'Pandas'], 'Fee’:
[20000,25000,22000,30000]}, index = ind )

ind2 = ['r1', 'r6', 'r3', 'r5']
df2 = pd.DataFrame({‘Courses’: ['Spark', ‘Java’,
'Python', ‘Go’], ‘Discount’: [2000,2300,1200,2000]},
index = ind2 )
print(df.join(df2,lsuffix= ‘_tech1’,rsuffix=
‘_tech2’, how = ‘inner’)
---------------------------------------------------
     Courses_tech1    Fee   Courses_tech2   Discount
r1   Spark            20000         Spark   2000
r3   Python           22000         Python  1200
```

Changing Data

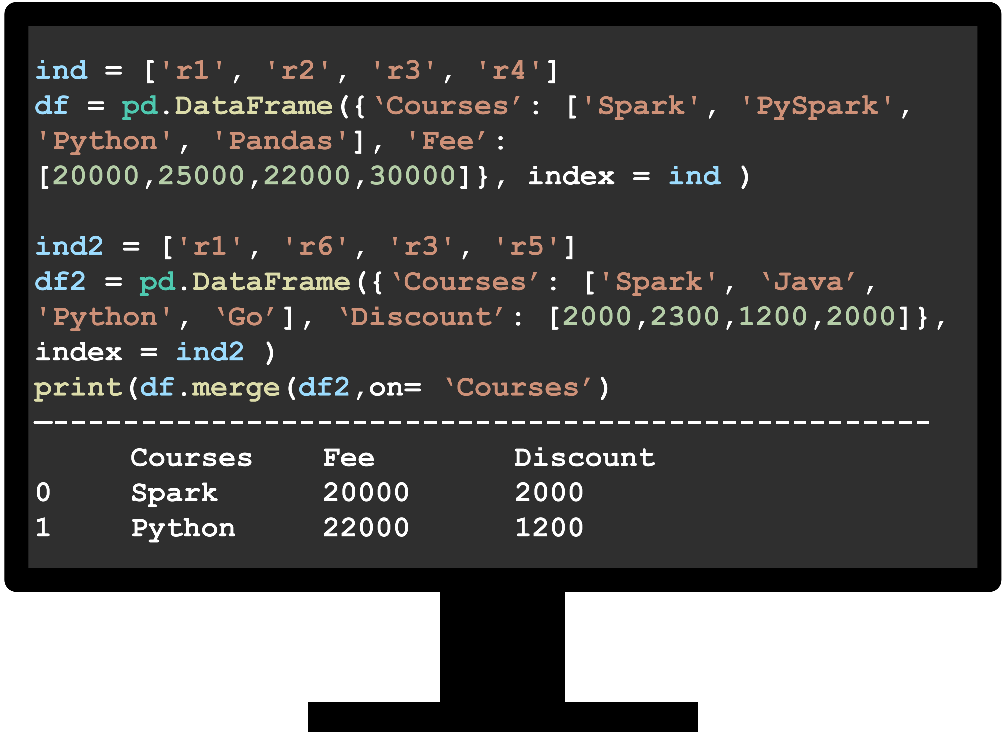
- ➔ DataFrame.merge(*right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, suffixes=('_x', '_y')*)
    - ○ Lets us join 2 DataFrames together on non index columns



```
ind = ['r1', 'r2', 'r3', 'r4']
df = pd.DataFrame({‘Courses’: ['Spark', 'PySpark',
'Python', 'Pandas'], 'Fee’:
[20000,25000,22000,30000]}, index = ind )

ind2 = ['r1', 'r6', 'r3', 'r5']
df2 = pd.DataFrame({‘Courses’: ['Spark', ‘Java’,
'Python', ‘Go’], ‘Discount’: [2000,2300,1200,2000]},
index = ind2 )
print(df.merge(df2,on= ‘Courses’)
--------------------------------------------------
     Courses    Fee        Discount
0    Spark      20000      2000
1    Python     22000      1200
```

Changing Data

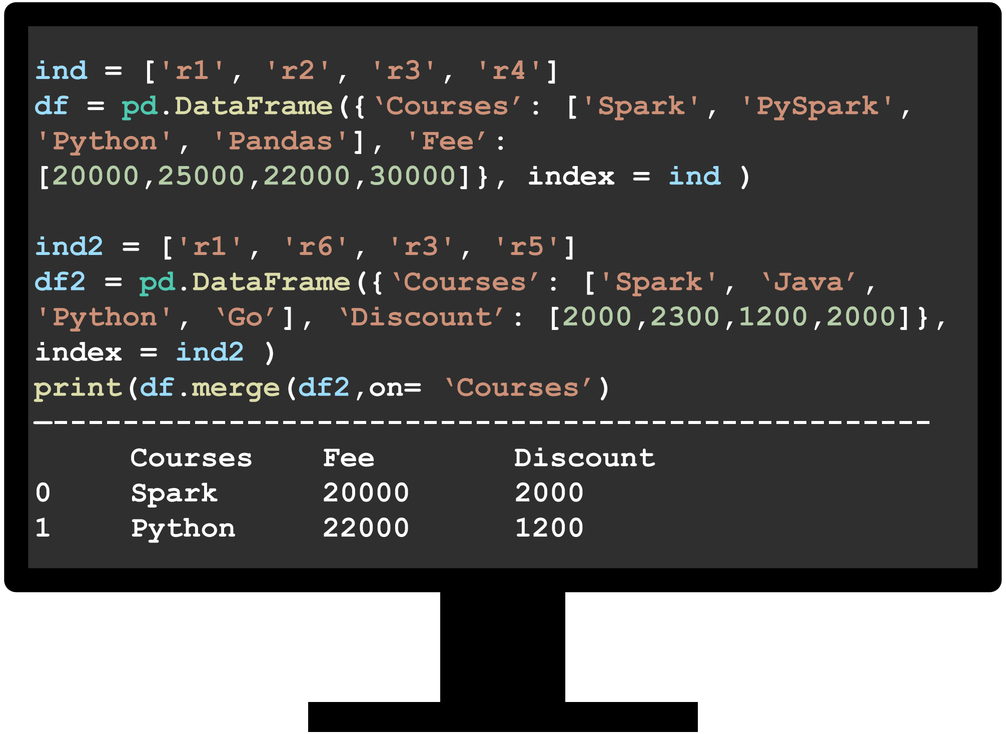
- ➔ *right*
  - ○ This is where we specify the dataframe on the right of the join
- ➔ *how='inner',*
  - ○ Here we can specify the type of join between ‘inner’, ‘outer’, ‘left’, ‘right’, ‘cross’



```
ind = ['r1', 'r2', 'r3', 'r4']
df = pd.DataFrame({‘Courses’: ['Spark', 'PySpark',
'Python', 'Pandas'], 'Fee’:
[20000,25000,22000,30000]}, index = ind )

ind2 = ['r1', 'r6', 'r3', 'r5']
df2 = pd.DataFrame({‘Courses’: ['Spark', ‘Java’,
'Python', ‘Go’], ‘Discount’: [2000,2300,1200,2000]},
index = ind2 )
print(df.merge(df2,on= ‘Courses’)
----------------------------------------------------
     Courses     Fee         Discount
0    Spark       20000       2000
1    Python      22000       1200
```

Changing Data

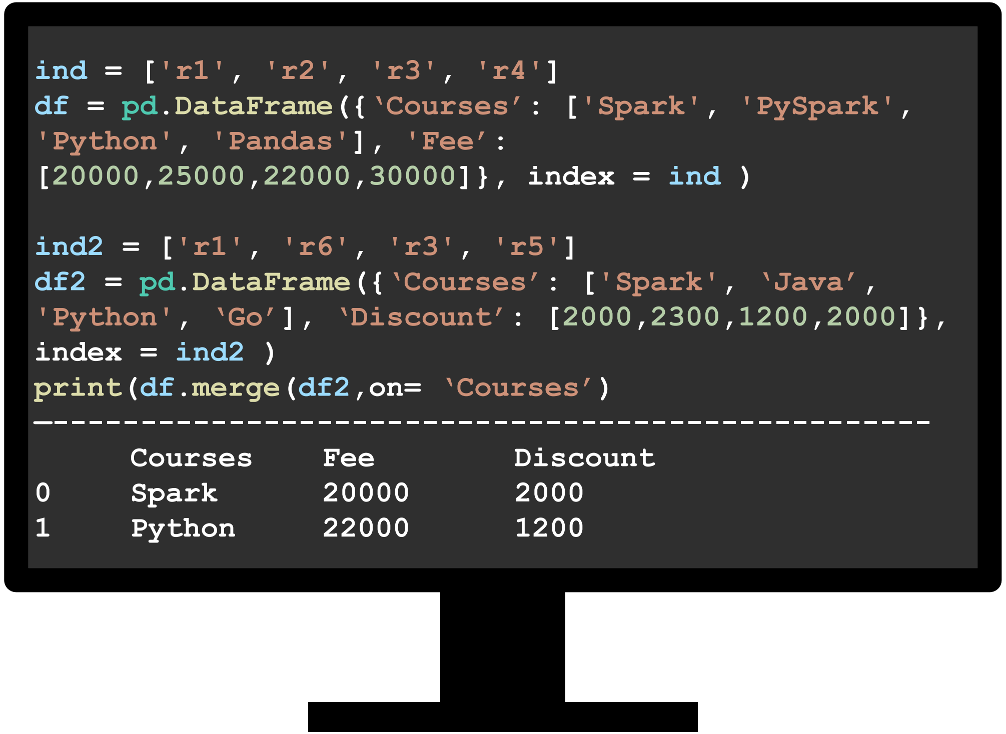
- ➔ ***on=None,***
  - ○ Specify the column to join on. This must be in both DataFrames
  - ○ If blank, defaults to the intersection of the columns
- ➔ ***left_on=None, right_on=None***
  - ○ Specify the column to join on for each left and right DataFrame



```
ind = ['r1', 'r2', 'r3', 'r4']
df = pd.DataFrame({‘Courses’: ['Spark', 'PySpark',
'Python', 'Pandas'], ‘Fee’:
[20000,25000,22000,30000]}, index = ind )

ind2 = ['r1', 'r6', 'r3', 'r5']
df2 = pd.DataFrame({‘Courses’: ['Spark', ‘Java’,
'Python', ‘Go’], ‘Discount’: [2000,2300,1200,2000]},
index = ind2 )
print(df.merge(df2,on= ‘Courses’)
----------------------------------------------------
     Courses    Fee       Discount
0    Spark      20000     2000
1    Python     22000     1200
```

Changing Data

- ➔ *left_index=False, right_index=False*
  - ○ A bool value for if the join should use the index instead of a column
- ➔ *suffixes=('_x', '_y')*
  - ○ A sequence of 2 suffixes to add to the columns from the left and right DataFrame respectively



```
ind = ['r1', 'r2', 'r3', 'r4']
df = pd.DataFrame({‘Courses’: ['Spark', 'PySpark',
'Python', 'Pandas'], 'Fee’:
[20000,25000,22000,30000]}, index = ind )

ind2 = ['r1', 'r6', 'r3', 'r5']
df2 = pd.DataFrame({‘Courses’: ['Spark', ‘Java’,
'Python', ‘Go’], ‘Discount’: [2000,2300,1200,2000]},
index = ind2 )
print(df.merge(df2,on= ‘Courses’))
-------------------------------------------------
      Courses     Fee        Discount
0     Spark       20000      2000
1     Python      22000      1200
```

Student Exercise

- Find the sum, average, minimum, and maximum salaries.
  - Then find these measurements by department
- Create a DataFrame of only active employees
- Construct a separate DataFrame of Departments and merge it onto your Employees DataFrame
- Everyone gets a raise! Increase the salaries of all employees by 10%

Student Exercise

FIN