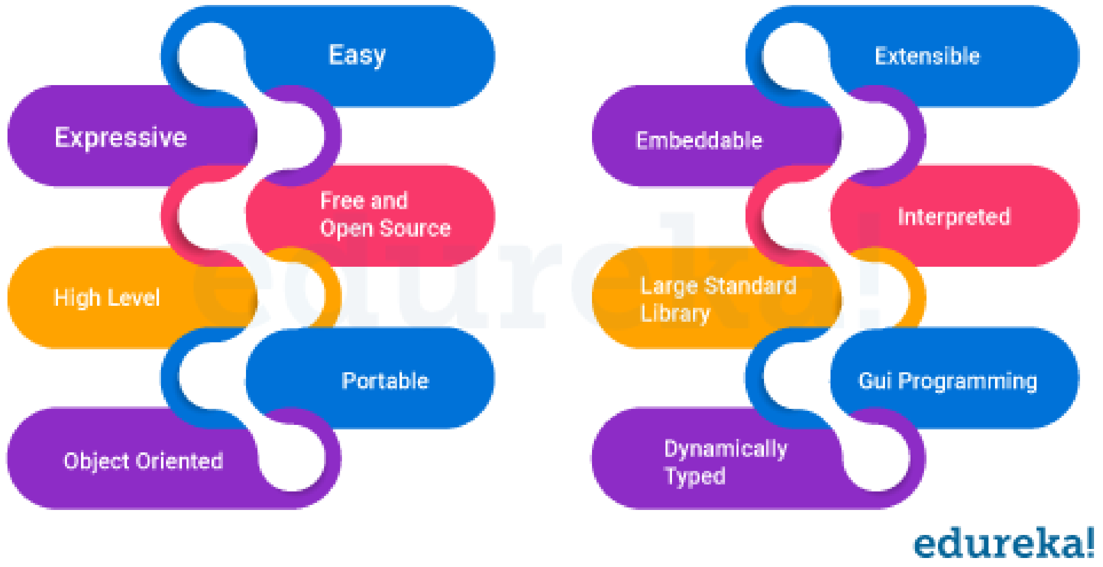
Python for data science

Python for Data science

- Python is a popular programming language widely used in the field of data science.
- The simplicity and readability of Python code make it a favorable choice for data scientists.
- Python has in-built mathematical libraries and functions, making it easier to calculate mathematical problems and to perform data analysis.



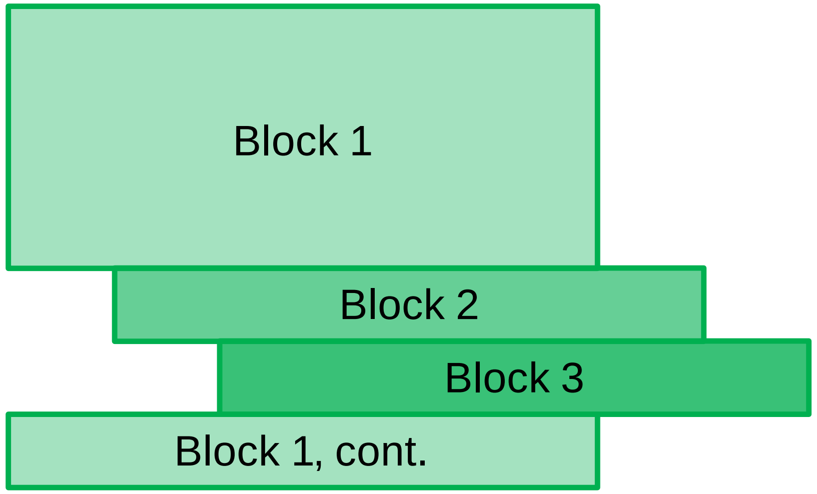
Structure of a python program

- Code blocks are defined by their indentations

  → Indentation is a requirement in Python! → Whitespace Within Lines Does Not Matter

- End-of-Line Terminates a Statement.
- Structures that introduce blocks end with a colon ":"

```
from math import sqrt

my_list = [1,2,3,4]
result = 0
for i in my_list:
    if i%2 == 0:
        result += sqrt(i)
print(result)
```



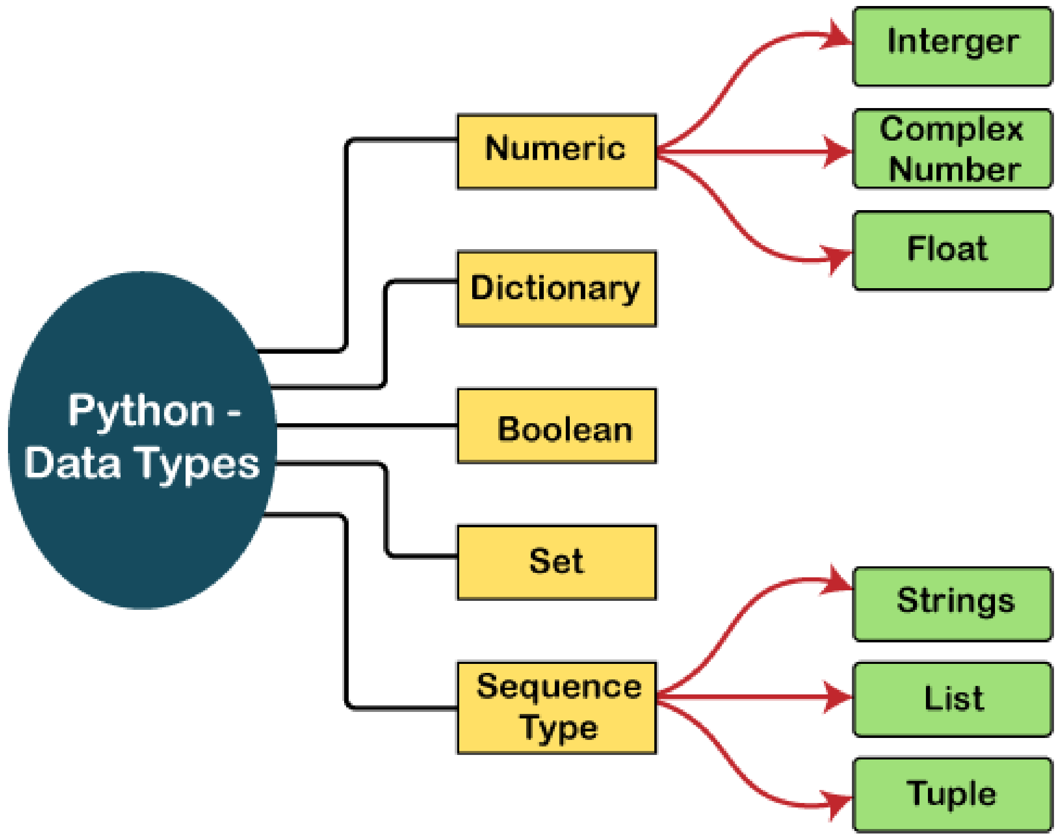
Data types

- Python is dynamically typed; the type of the variable is derived from the value it is assigned.
- You can check what type of object is assigned to a variable using Python's built-in **type()** function.
- Variables can be *cast* to a different type.
    - Type names are used as type converter functions

```
x = 5 #x is an int

#converting to float
x_float = float(x)
```

Built-in Data types

- Python supports various built-in data types.

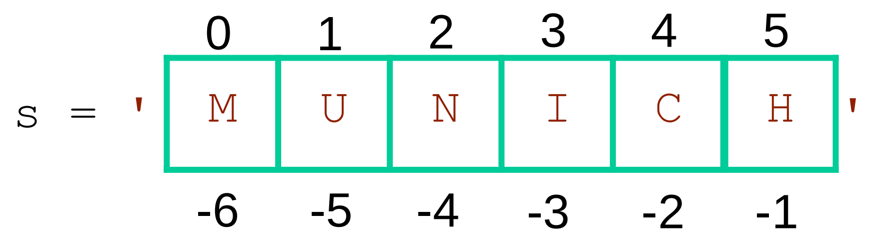


Built-in data types: Numeric and boolean

- **Integer:**
  - Normal Integer, e.g. `i = 345`
  - Octal Literals, e.g. `i = 0o10`
  - Hexadecimal Literals, e.g. `i = 0x1F`
  - Binary Literals, e.g. `i = 0b10110`
- **Floating-point numbers**
  - E.g. `i = 1.234e-2`
- **Complex numbers**
  - Composed of <real part> + <imaginary part>j, e.g. `i = 3+4j`
- **Boolean Values**
  - `True` and `False`

Built-in Data types: Strings

- Strings are sequences of characters, enclosed in either single (' ') or double (" ") quotes.
- Strings are immutable: they cannot be changed after creation.
- **Concatenation**: Combine strings using the + operator.
- **Indexing and Slicing**: Access individual characters or substrings using indices.



```
>>> s[2]     >>> s[-1]     >>> s[2:]     >>> s[:-2]     >>> s[2:-2]
'N'          'H'           'NICH'        'MUNI'         'NI'
```

Built-in Data types: Strings

- Python provides a lot of built-in functions to manipulate strings.
- String formatters allow us to print characters and values at once.

```
name = "Python"

message = f"Hello, {name}!"

substring = message[7:13]

uppercase_message = message.upper()
```

Common Operators

• Arithmetic	+	-	*	//	/	%	**	
• Comparison	==	!=	>	<	>=	<=		
• Assignment	=	+=	-=	*=	//=	/=	%=	**=
• Logical	and	or	not					
• Bitwise	&	|	^	~	>>	<<		
• Membership	in	not in						
• Identity	is	is not						

Data structures: Lists

```
[0b100, ['times',True], 'is', 4.]
```

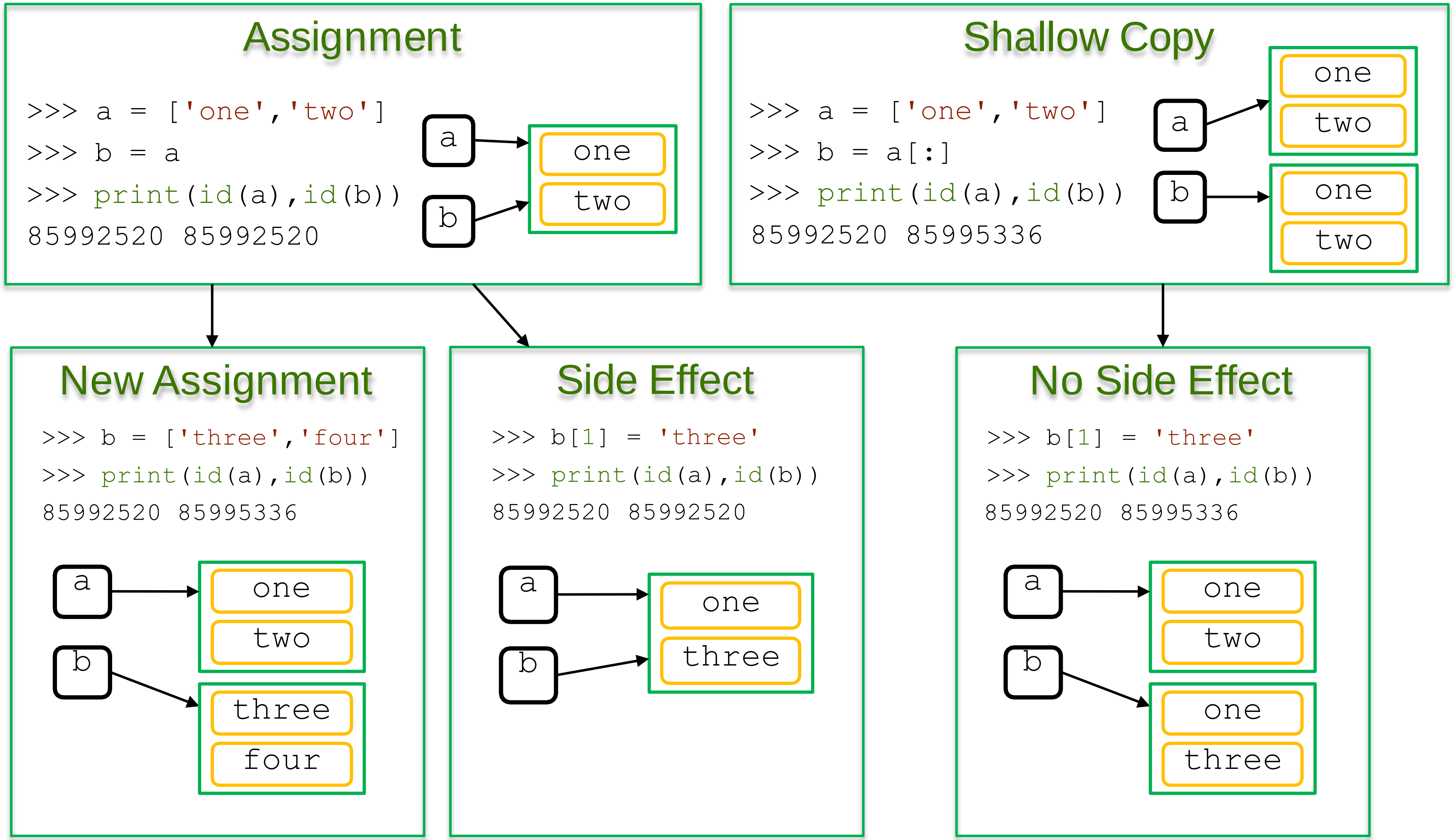
- Most widely used data structure
- Items do not need to have the same type
- Lists:
  - are ordered
  - can grow dynamically. A list grow and shrink as needed
  - are mutable and elements can be accessed by their index
  - are supported by many built-in functions

Data structures: Lists

- List comprehension as an elegant way to create lists

```
>>> a = [x**2 for x in range(7)]
>>> a
[0, 1, 4, 9, 16, 25, 36]
>>> sum(a)  91
>>> a + [x**2 for x in range(7,9)]
[0, 1, 4, 9, 16, 25, 36, 49, 64]
>>> del a[:3]
[9, 16, 25, 36, 49, 64]
```

Copying in Python



Assignment

```
>>> a = ['one','two']
>>> b = a
>>> print(id(a),id(b))
85992520 85992520
```

Shallow Copy

```
>>> a = ['one','two']
>>> b = a[:]
>>> print(id(a),id(b))
85992520 85995336
```

New Assignment

```
>>> b = ['three','four']
>>> print(id(a),id(b))
85992520 85995336
```

Side Effect

```
>>> b[1] = 'three'
>>> print(id(a),id(b))
85992520 85992520
```

No Side Effect

```
>>> b[1] = 'three'
>>> print(id(a),id(b))
85992520 85995336
```

Data structures: Tuples

```
('A tuple with', 3, 'entries')
```

- A tuple is a sequence of comma separated values
- Values can have different types
- Tuples are immutable (but can contain mutable values)
- Tuples are faster than lists

```
>>> t = 1, [2], 'tuple' #tuple packing
>>> t[2]
'tuple'
>>> t[0] = 3
TypeError
>>> t[1][0] = 3
>>> t
(1, [3], 'tuple')
>>> x, y, z = t #sequence unpacking
```

Data structures: Dictionnaries

```
{'Munich': 1.5, 'Berlin': 3.5, 'Hamburg': 1.8 }
```

- Dictionaries are unordered and indexed collections of (key,value) pairs
- Accessed by keys, so keys are unique
- Keys must be immutable, values can be of arbitrary type
- Dictionaries are not sequence types like strings, lists or tuples

Data structures: Dictionnaries

```
>>> d = { i**2: i for i in range(7)}
>>> d
{0: 0, 1: 1, 4: 2, 9: 3, 16: 4, 25: 5, 36: 6}
>>> d[4]
2
>>> for entry in d.items():
        if entry[0] == 4: print(entry)
(4,2)
>>> [key for key in d.keys()] #iterating over values is supported, too  [0,
1, 4, 9, 16, 25, 36]
>>> d[49] = 7 #delete values by using del key word, e.g. del d[36]
>>> d
{0: 0, 1: 1, 4: 2, 49: 7, 9: 3, 16: 4, 25: 5, 36: 6}
```

Data structures: Sets

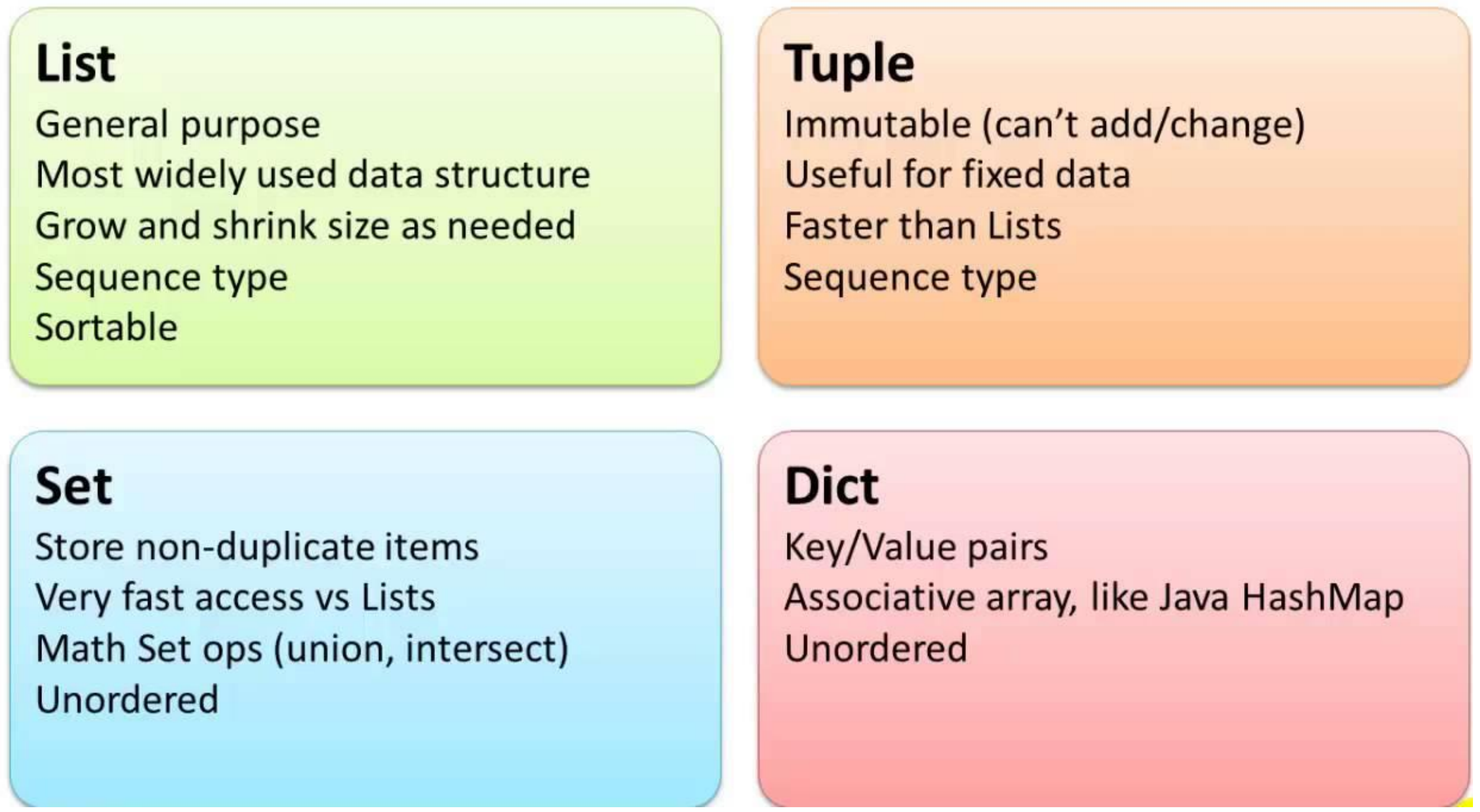
```
{'USA', 'Canada', 'Germany', 'Japan’}
```

- Sets are unordered collection of unique values
- Duplicates are automatically eliminated
- Sets support various set operations, such as union, intersection

Data structures: Sets

```
>>> set1 = {1, 2, 3, 4}
>>> set2 = {3, 4, 5, 6}
>>> set1.add(4)  # Adding an existing element does not change the set
>>> set1
{1, 2, 3, 4}

>>> union = set1.union(set2) # we can also use the | operator
>>> union
{1, 2, 3, 4, 5, 6}
>>> intersection = set1.intersection(set2) # we can also use the & operator
>>> intersection
{3, 4}
```

Data structures: List, Tuple, Set and Dict



List
General purpose
Most widely used data structure
Grow and shrink size as needed
Sequence type
Sortable

Tuple
Immutable (can't add/change)
Useful for fixed data
Faster than Lists
Sequence type

Set
Store non-duplicate items
Very fast access vs Lists
Math Set ops (union, intersect)
Unordered

Dict
Key/Value pairs
Associative array, like Java HashMap
Unordered

Conditional statements and loops

if-else statement

```
if condition:
    # execute code if condition is true
else:
    # execute code if condition is false
```

for loop

```
for element in sequence:
    # execute code
```

if-elif statement

```
if condition1:
    # execute code if condition1 is true
elif condition2:
    # execute code if condition2 is true
```

while loop

```
while condition:
    # execute code
```

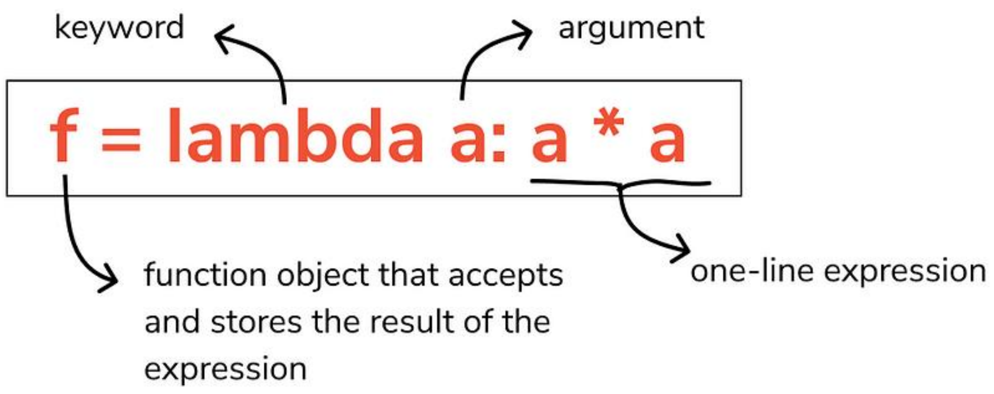
Functions

- A function is a block of reusable code which only runs when it is called.
- Example of simple Python function

```
>>> def add(a, b):
        return a+b


>>> add(2, 3)
5
```

Lambda Functions

- Lambda function:
  - Small, anonymous functions without a name.
  - Can have any number of arguments, but only one expression.


keyword
argument
f = lambda a: a * a
function object that accepts and stores the result of the expression
one-line expression

```
>>> add = lambda x, y : x+y
>>> add(2, 3)
5
```

Map, filter and reduce functions

- **map(func,seq)**

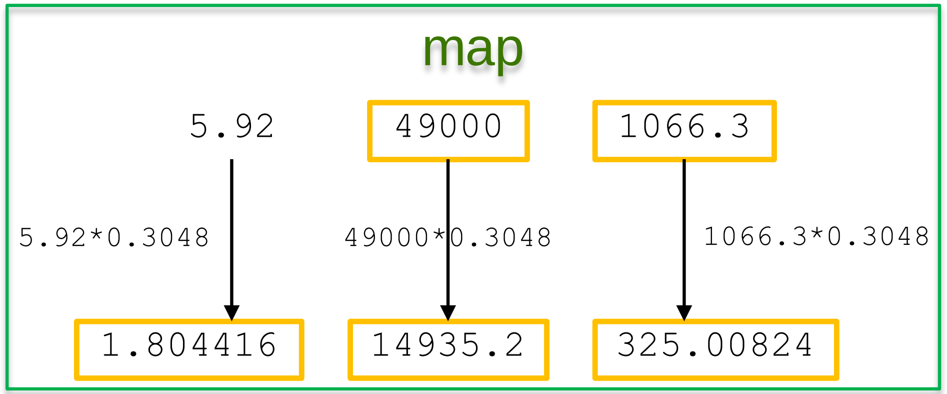
```
>>> feet = [5.92, 49000, 1066.3]
>>> list(map(lambda x: x*0.3048, feet))
[1.804416, 14935.2, 325.00824]
```

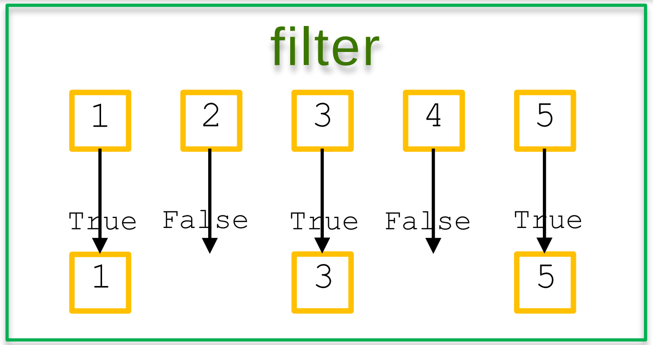
- **filter(func,seq)**

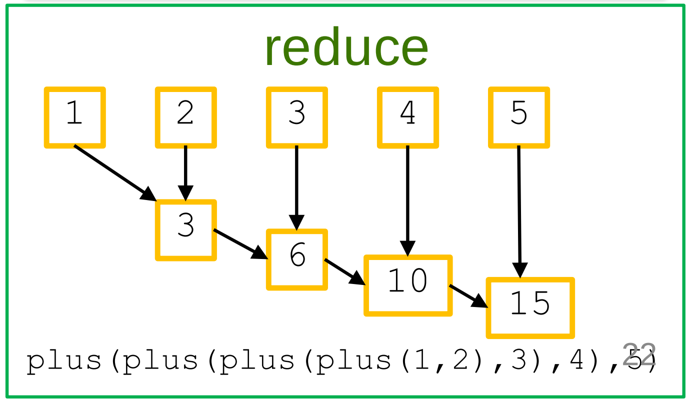
```
>>> list(filter(lambda x: x%2==1, [1,2,3,4,5]))
[1, 3, 5]
```

- **reduce(func,seq)**

```
>>> from functools import reduce
>>> reduce(lambda x,y: x+y, [1,2,3,4,5])
15
```

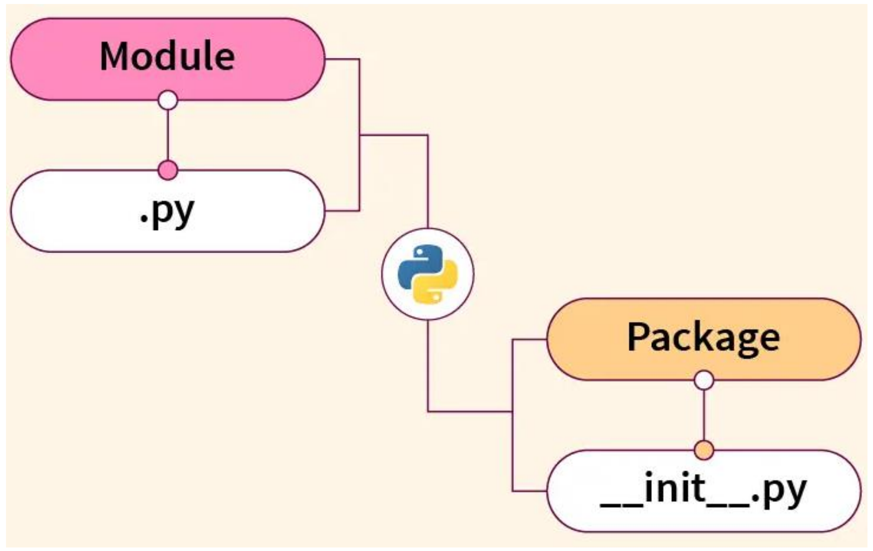






Modules and packages

- As program gets longer, need to organize code for easier access and easier maintenance.
- Modules are individual Python files containing code.
- Packages are directories containing multiple modules.
    - A package must contain an **__init__.py** file.
- To use an external code in your program, you use the **import** statement



Python Libraries for Data Science

Many popular Python toolboxes/libraries:

- NumPy
- SciPy
- Pandas
- SciKit-Learn

Visualization libraries

- matplotlib
- Seaborn

and many more …

Python Libraries for Data Science

NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects

- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance

- many other python libraries are built on NumPy

Link: http://www.numpy.org/

Python Libraries for Data Science

SciPy:

- collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more
- part of SciPy Stack
- built on NumPy

Link: https://www.scipy.org/scipylib/

Python Libraries for Data Science

Pandas:

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)

- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.

- allows handling missing data

Link: http://pandas.pydata.org/

Python Libraries for Data Science

SciKit-Learn:

- provides machine learning algorithms: classification, regression, clustering, model validation etc.

- built on NumPy, SciPy and matplotlib

Link: http://scikit-learn.org/

Python Libraries for Data Science

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

Python Libraries for Data Science

Seaborn:

- based on matplotlib
- provides high level interface for drawing attractive statistical graphics
- Similar (in style) to the popular ggplot2 library in R

Link: https://seaborn.pydata.org/

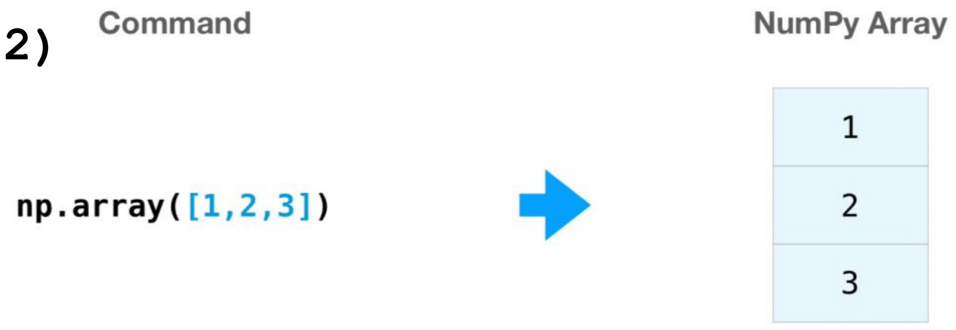
NumPy

- NumPy stands for **Num**erical **Py**thon.
- It is a powerful library for numerical computations in Python.
- Important features:
  - Multidimensional Arrays: **ndarray** for creating multiple dimensional arrays
  - Mathematical operations: Standard math functions for fast operations on entire arrays of data without having to write loops
  - Broadcasting: a powerful and robust method for executing operations to arrays of various shapes and sizes.
  - Vectorization: NumPy Arrays are important because they enable you to express batch operations on data without writing any for loops.

NumPy arrays

- NumPy arrays, also known as ndarray (n-dimensional array), are fixed-size containers of items of the same type and size.

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]],dtype=np.float32)
print a.ndim, a.shape, a.dtype
```



- Basic properties:

1. Arrays can have any number of dimensions, including zero (a scalar).
2. Arrays are typed: np.uint8, np.int64, np.float32, np.float64
3. Arrays are dense. Each element of the array exists and has the same type.

NumPy arrays, creation

Using Python sequences

- np.array()

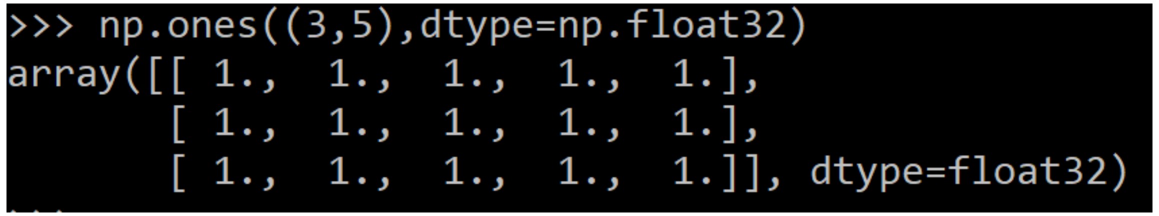
Using intrinsic NumPy array creation functions: (40 functions)

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random
- ….

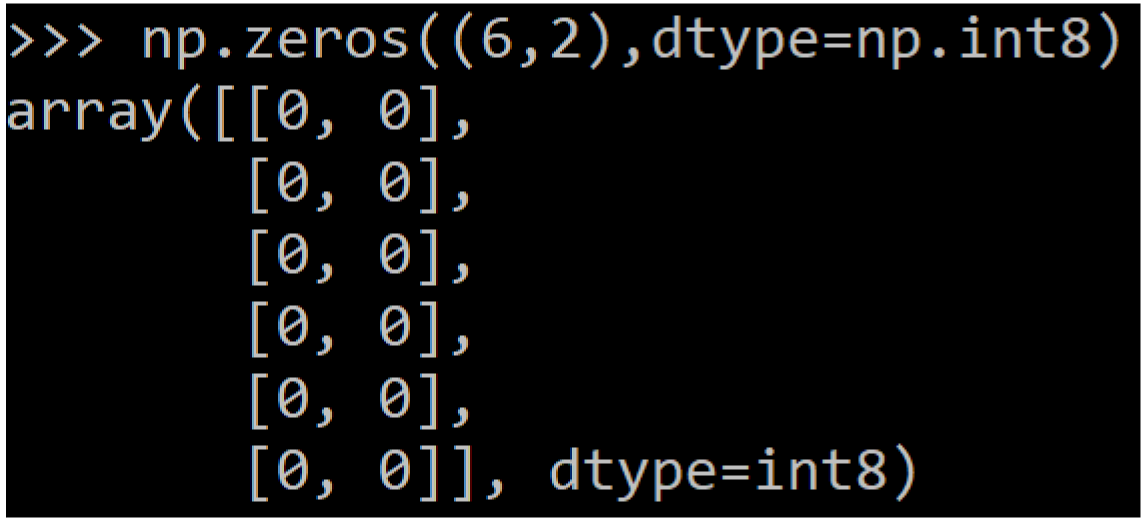
NumPy arrays, creation

Some built-in functions:

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random



```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

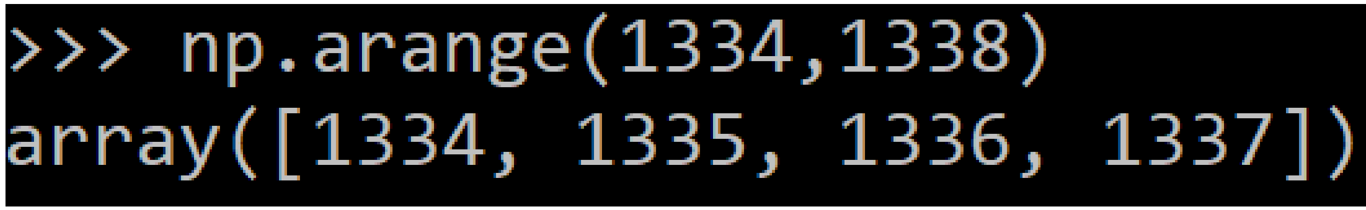


```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

NumPy arrays, creation

Some built-in functions:

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

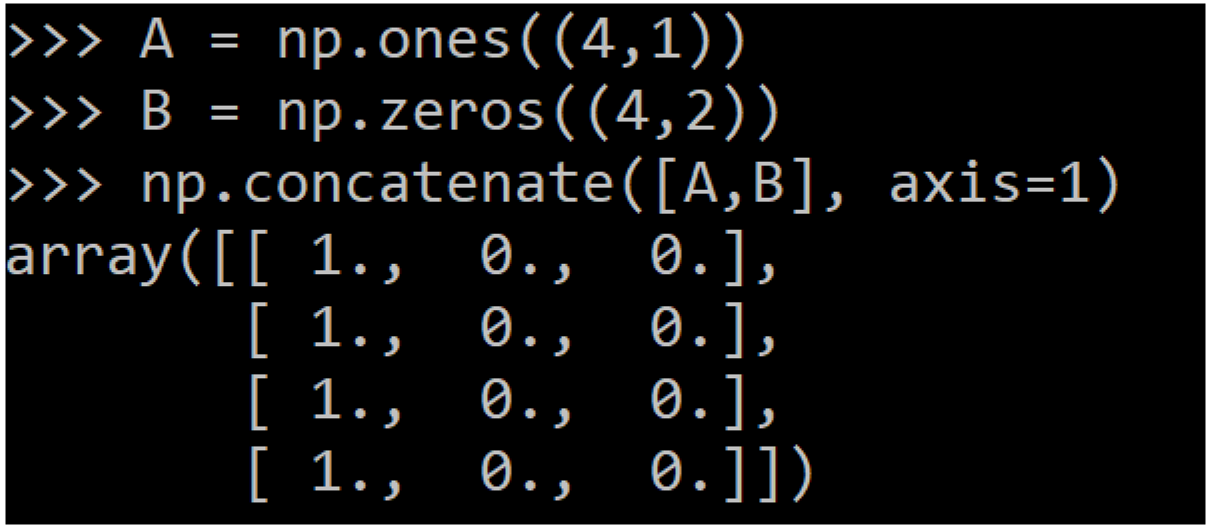


```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```

NumPy arrays, creation

Some built-in functions:

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

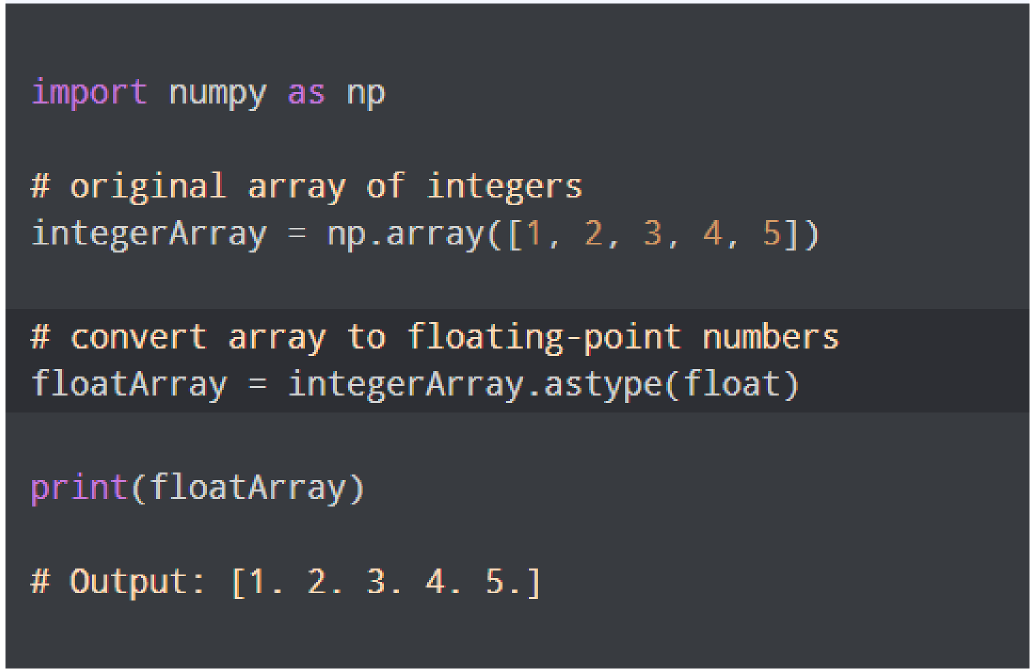


```
>>> A = np.ones((4,1))
>>> B = np.zeros((4,2))
>>> np.concatenate([A,B], axis=1)
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

NumPy arrays, creation

Some built-in functions:

- np.ones, np.zeros
- np.arange
- np.concatenate
- **np.astype**
- np.zeros_like, np.ones_like
- np.random.random

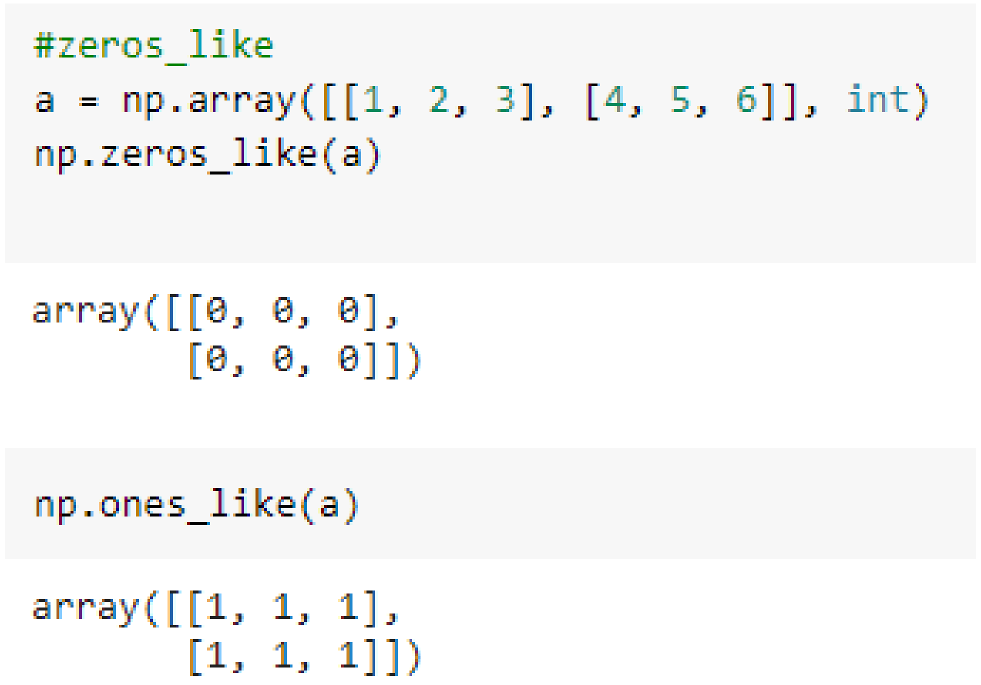


```
import numpy as np

# original array of integers
integerArray = np.array([1, 2, 3, 4, 5])

# convert array to floating-point numbers
floatArray = integerArray.astype(float)

print(floatArray)

# Output: [1. 2. 3. 4. 5.]
```

NumPy arrays, creation

Some built-in functions:

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- **np.zeros_like, np.ones_like**
- np.random.random



```
#zeros_like
a = np.array([[1, 2, 3], [4, 5, 6]], int)
np.zeros_like(a)
```

```
array([[0, 0, 0],
       [0, 0, 0]])
```

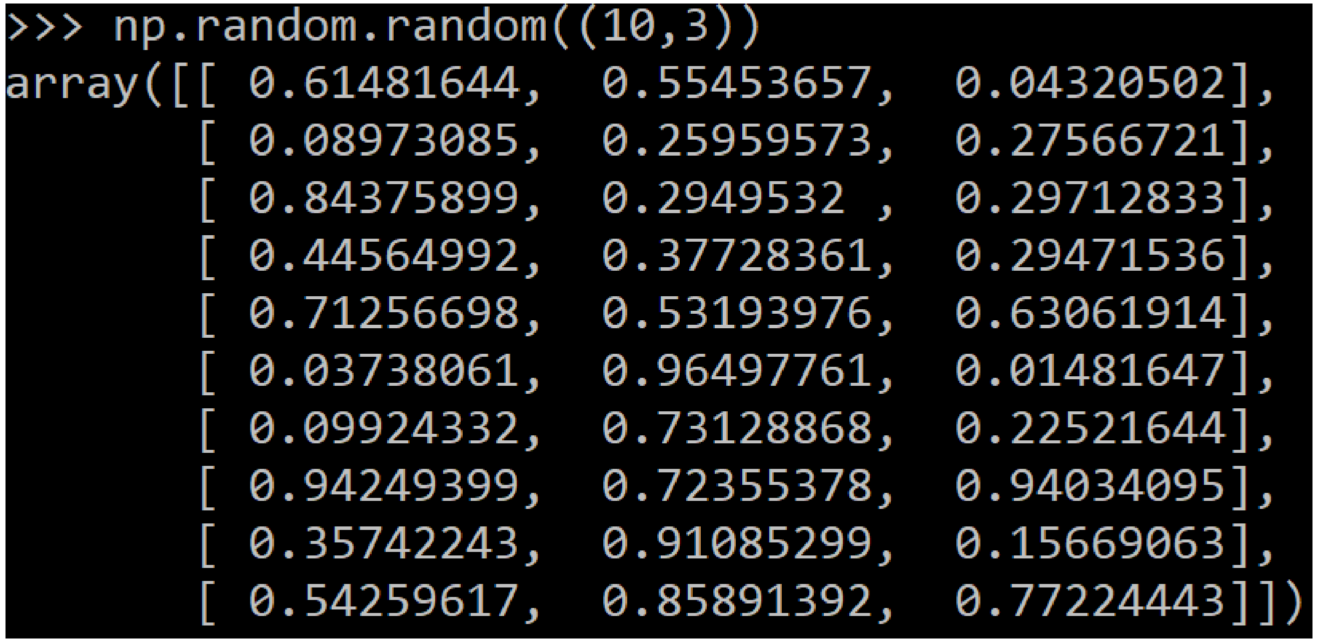
```
np.ones_like(a)
```

```
array([[1, 1, 1],
       [1, 1, 1]])
```

NumPy arrays, creation

Some built-in functions:

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- **np.random.random**



```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

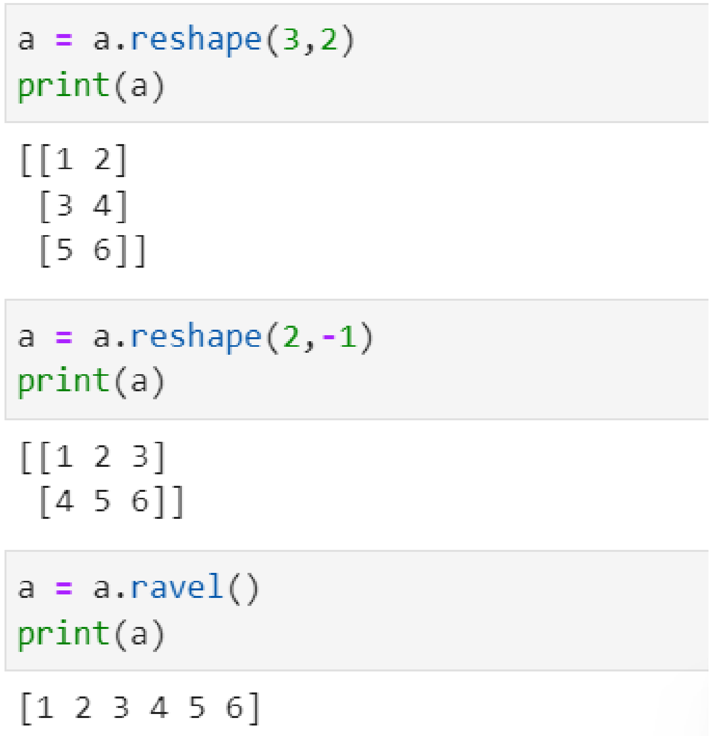
Reshaping

- Changing array shape using **reshape** function
  - Total number of elements cannot change.
  - Use -1 to infer axis shape

```
a = np.array([1,2,3,4,5,6])
a = a.reshape(3,2)
a = a.reshape(2,-1)
```

- Flatten out a Numpy array using the **ravel** function

```
a = a.ravel()
```



```
a = a.reshape(3,2)
print(a)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

```
a = a.reshape(2,-1)
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

```
a = a.ravel()
print(a)
```

```
[1 2 3 4 5 6]
```

Arithmetic with NumPy Arrays

- Any arithmetic operation between equal-size arrays applies the operation **element-wise**:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(arr)
[[1. 2. 3.]
 [4. 5. 6.]]
print(arr * arr)
[[ 1.  4.  9.]
 [16. 25. 36.]]
print(arr - arr)
[[0. 0. 0.]
 [0. 0. 0.]]
```

Arithmetic with NumPy Arrays

- Arithmetic operations with scalars propagate the scalar argument to each element in the array (*broadcasting*).

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(arr)
[[1. 2. 3.]
 [4. 5. 6.]]

print(arr **2)
[[ 1.  4.  9.]
 [16. 25. 36.]]
```

Arithmetic with NumPy Arrays

- Comparisons between arrays of the same size yield boolean arrays:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(arr)
[[1. 2. 3.]
 [4. 5. 6.]]

arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
print(arr2)
[[ 0.  4.  1.]
 [ 7.  2. 12.]]

print(arr2 > arr)
[[False  True False]
 [ True False  True]]
```

Indexing and Slicing

- One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
arr = np.arange(10)
print(arr)            # [0 1 2 3 4 5 6 7 8 9]
print(arr[5])      #5
print(arr[5:8]) #[5 6 7]
```

- if you assign a scalar value to a slice the value is propagated (or *broadcasted*) to the entire selection.

```
arr[5:8] = 12
print(arr)      #[ 0 1 2 3 4 12 12 12 8 9]
```

Indexing and Slicing

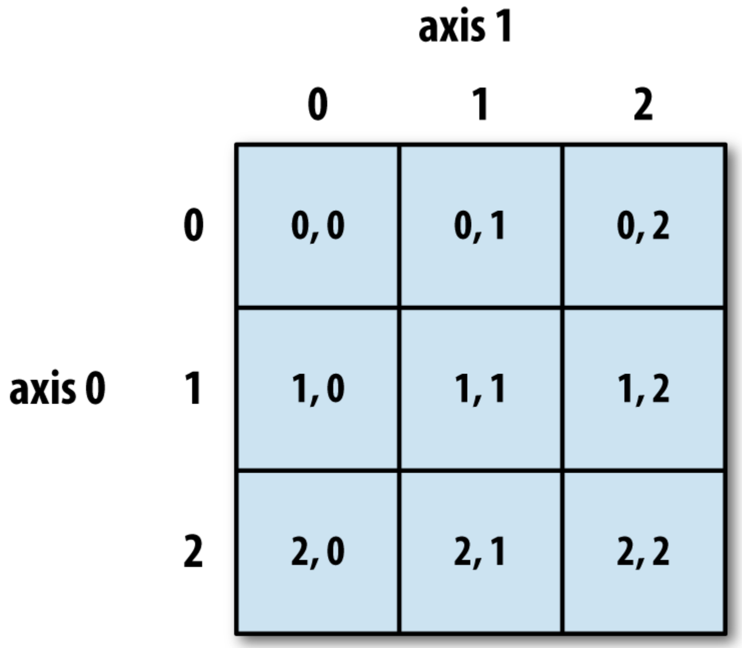
- Slices are views. Writing to a slice overwrites the original array.

```
arr = np.arange(10)
print(arr)            # [0 1 2 3 4 5 6 7 8 9]


arr_slice = arr[5:8]
print(arr_slice)                # [5 6 7]
arr_slice[1] = 12345
print(arr)                      # [0   1   2   3   4   5 12345   7   8   9]
arr_slice[:] = 64
print(arr)                      # [0   1   2   3   4  64  64  64   8   9]
```

Indexing and Slicing

- Multi-dimensional indices are comma-separated.

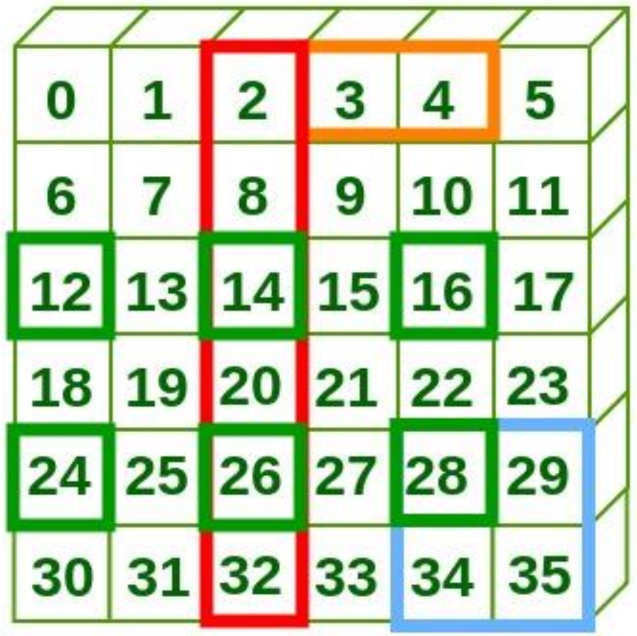
```
x[0,0]      # top-left element
x[0,-1]     # first row, last column
x[0,:]      # first row (many entries)
x[:,0]      # first column (many entries)
```

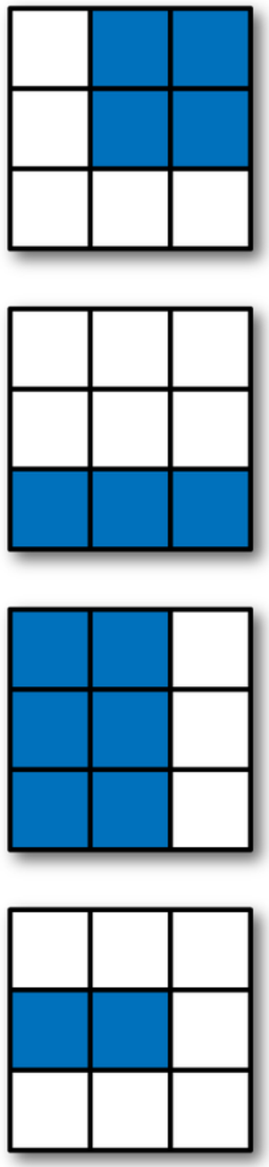


Indexing and Slicing

- Multi-dimensional array slicing

```
>>> a[0,3:5]
array( [3,4] )

>>> a[4:, 4:]
array( [ 28, 29],
       [ 34, 35] ] )

>>> a[ :, 2]
array( [2, 8, 14, 20, 26, 32] )

>>> a[2 : : 2, : : 2]
array( [ 12, 14, 16],
       [ 24, 26, 28] ] )
```





Expression	Shape
arr[:2, 1:]	(2, 2)
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)
arr[:, :2]	(3, 2)
arr[1, :2]	(2,)
arr[1:2, :2]	(1, 2)

Activity

- Consider the two-dimensional array, arr2d.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

- Write a code to slice this array to display the last column,

```
[[3]
[6]
[9]]
```

- Write a code to slice this array to display the last 2 elements of middle array,

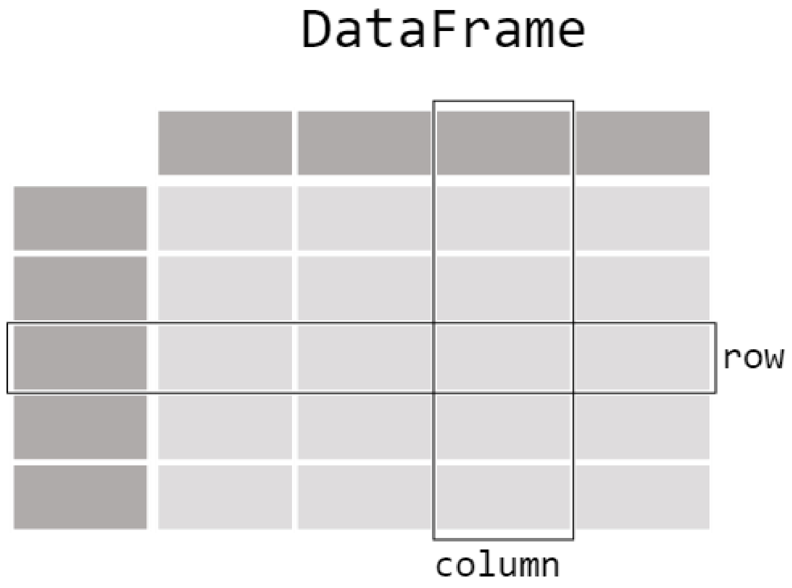
```
[5 6]
```

Lab 1:

NUMPY practice

Pandas

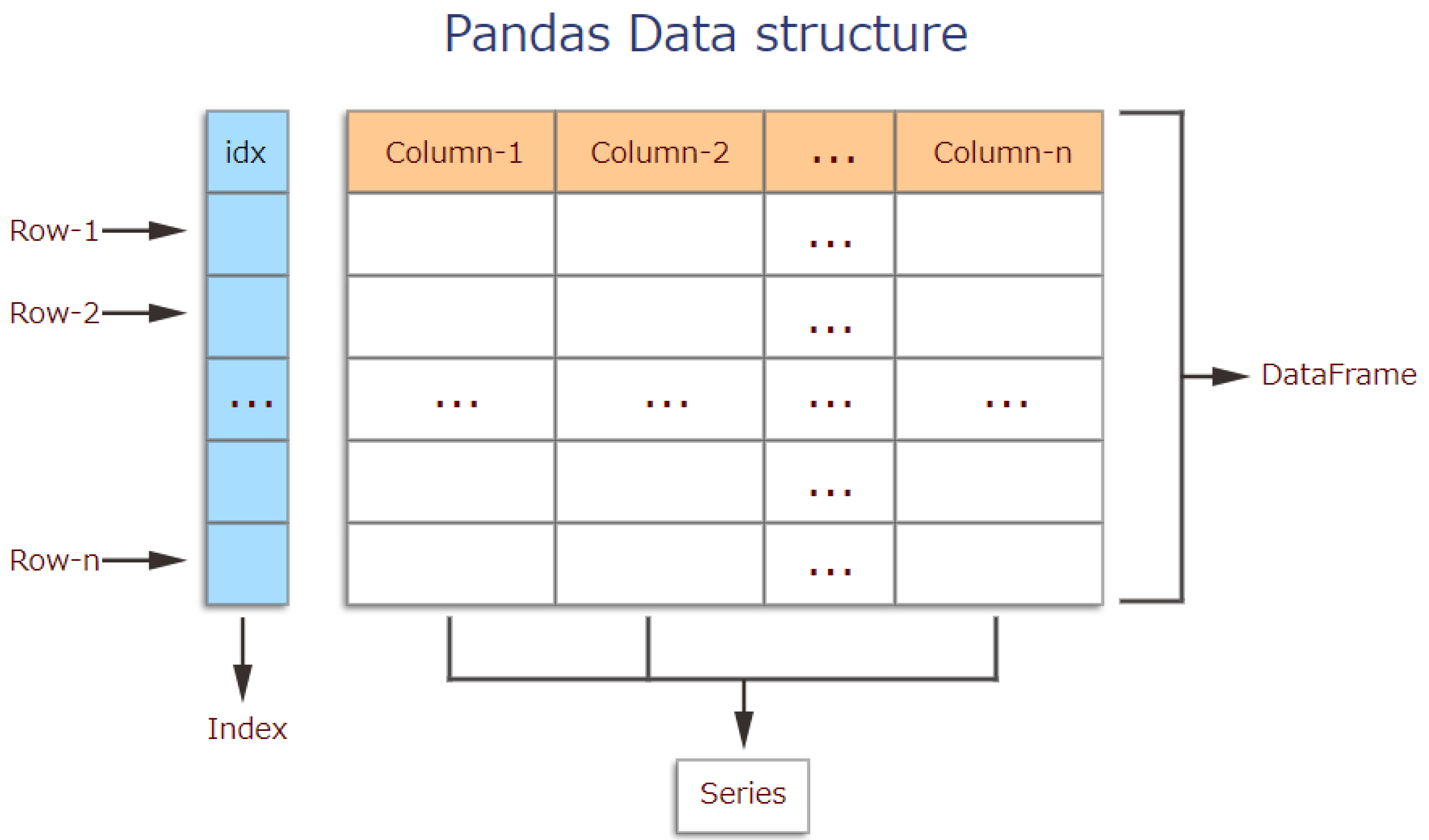
- Pandas is a popular data manipulation library in Python.
- Open source python package built on top of NumPy.
- Designed for data analysis, it provides high-performance, easy-to-use data structures and powerful tools for data cleaning, exploration, and manipulation.
- It primarily handles structured data, data that can be organized into rows and columns.



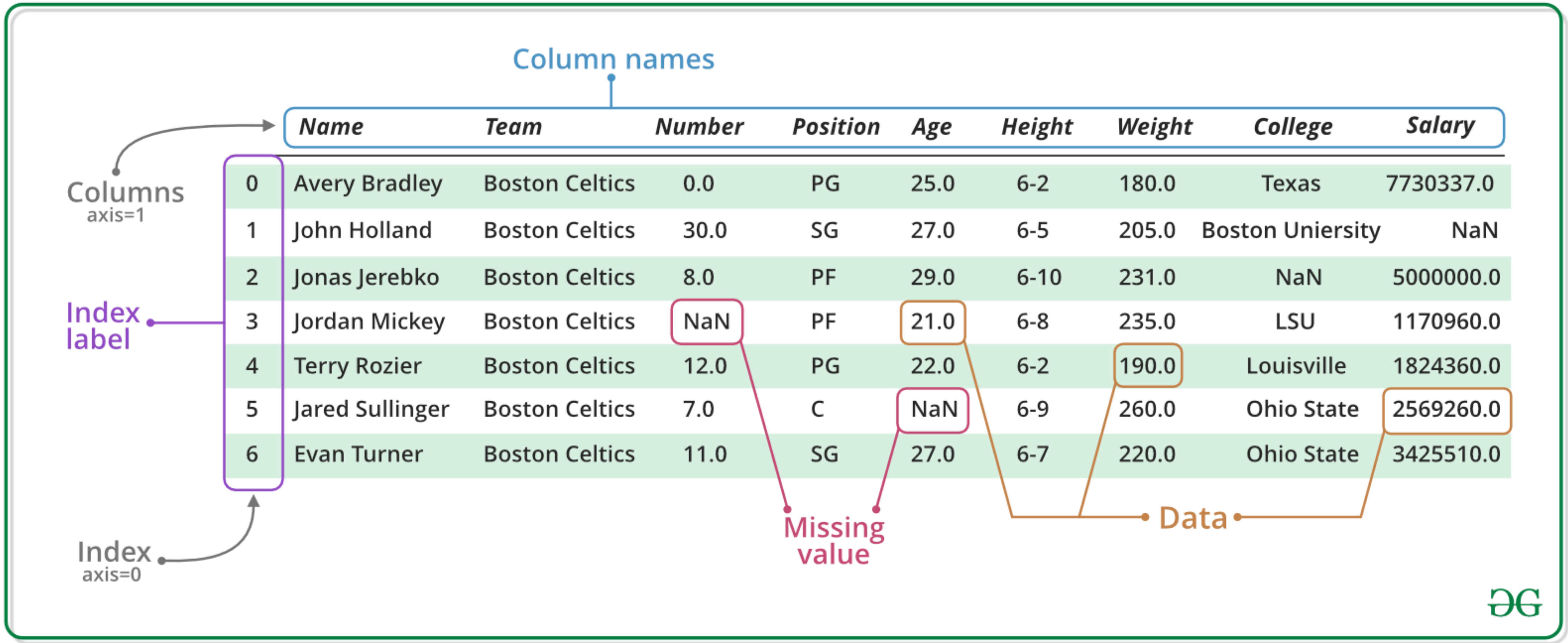
Core components of pandas: **Series** & **DataFrames**

- The primary two components of pandas are the Series and DataFrame.
  - Series is essentially a column. It is a one-dimensional array (1D Array) like structure with homogeneous data.
  - DataFrame is a two-dimensional array (2D Array) with heterogeneous data. It is made up of a collection of Series.
- DataFrames and Series are quite similar in that many operations that you can do with one you can do with the other, such as filling in null values and calculating the mean.

Pandas Data structure



Example of a Dataframe



	Name	*Team*	*Number*	*Position*	*Age*	*Height*	*Weight*	*College*	*Salary*
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston Uniersity	NaN
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0	6-8	235.0	LSU	1170960.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0	6-2	190.0	Louisville	1824360.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN	6-9	260.0	Ohio State	2569260.0
6	Evan Turner	Boston Celtics	11.0	SG	27.0	6-7	220.0	Ohio State	3425510.0

Creating a Pandas DataFrame

- A pandas DataFrame can be created in different ways and using various inputs such as:
  - Lists
  - dictionnary
  - Series
  - Numpy ndarrays
  - Another DataFrame
- In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage (CSV file, SQL database…)

Creating a Pandas Dataframe

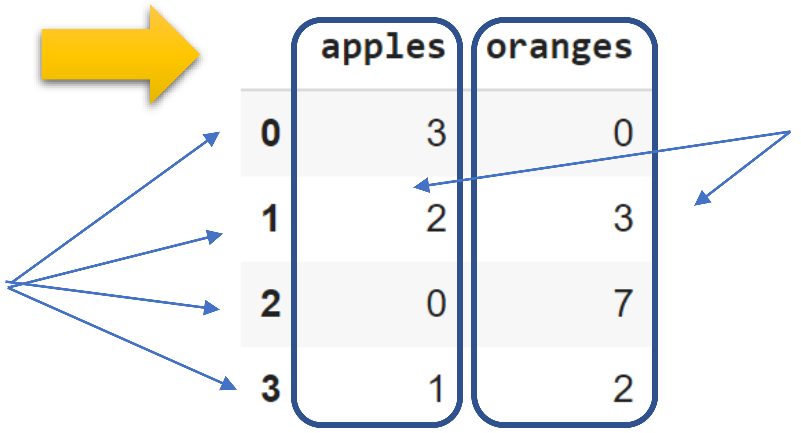
```
pandas.DataFrame(data, index , columns , dtype , copy )
```

- `data`: data can be *`ndarray`*, *`series`*, *`lists`*, *`dict`*, constants and also another *`DataFrame`*.
- `index`: **row labels** for the resulting frame, Default is *np.arrange(n)* if no index is passed.
- `columns`: For **column labels**, the optional default syntax is - *np.arrange(n)*. This is only true if no index is passed.
- `dtype`: Data type of each column.
- `copy`: Boolean used to specify whether data is copied from the inputs.

E.g. creating Dataframe from a dictionnary

- Let's say we have a fruit stand that sells apples and oranges. We want to:
  - have a column for each fruit and
  - a row for each customer purchase.
- We can create a dictionary and pass it to the pandas DataFrame constructor:

```
data = { 'apples':[3, 2, 0, 1] , 'oranges':[0, 3, 7, 2] }

df = pd.DataFrame(data)
```



	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

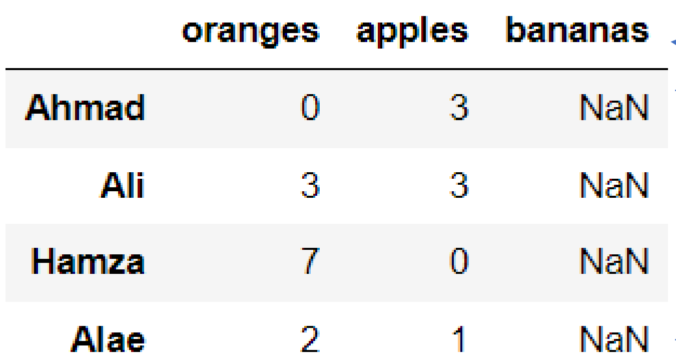
Each (key, value) item in data corresponds to a column

Row index assigned by default, Pandas uses numbers starting from 0

E.g. creating Dataframe from a dictionnary

- Specifying columns and indices
  - Order of columns/rows can be specified.
  - Columns not in data will have NaN.
- We can set customer names as the index and fruit names as column label

```
df = pd.DataFrame(data, columns=['oranges’, 'apples’, 'bananas’],
index=['Ahmad', 'Ali', ’Hamza', ’Alae'])
```



	oranges	apples	bananas
Ahmad	0	3	NaN
Ali	3	3	NaN
Hamza	7	0	NaN
Alae	2	1	NaN

Same columns order

Column bananas initialized with NaN

Reading data from a file

- Read a CSV file

```
df = pd.read_csv("data.csv")
```

The above command has many optional arguments to fine-tune the data import process

- There is a number of pandas commands to read other data formats

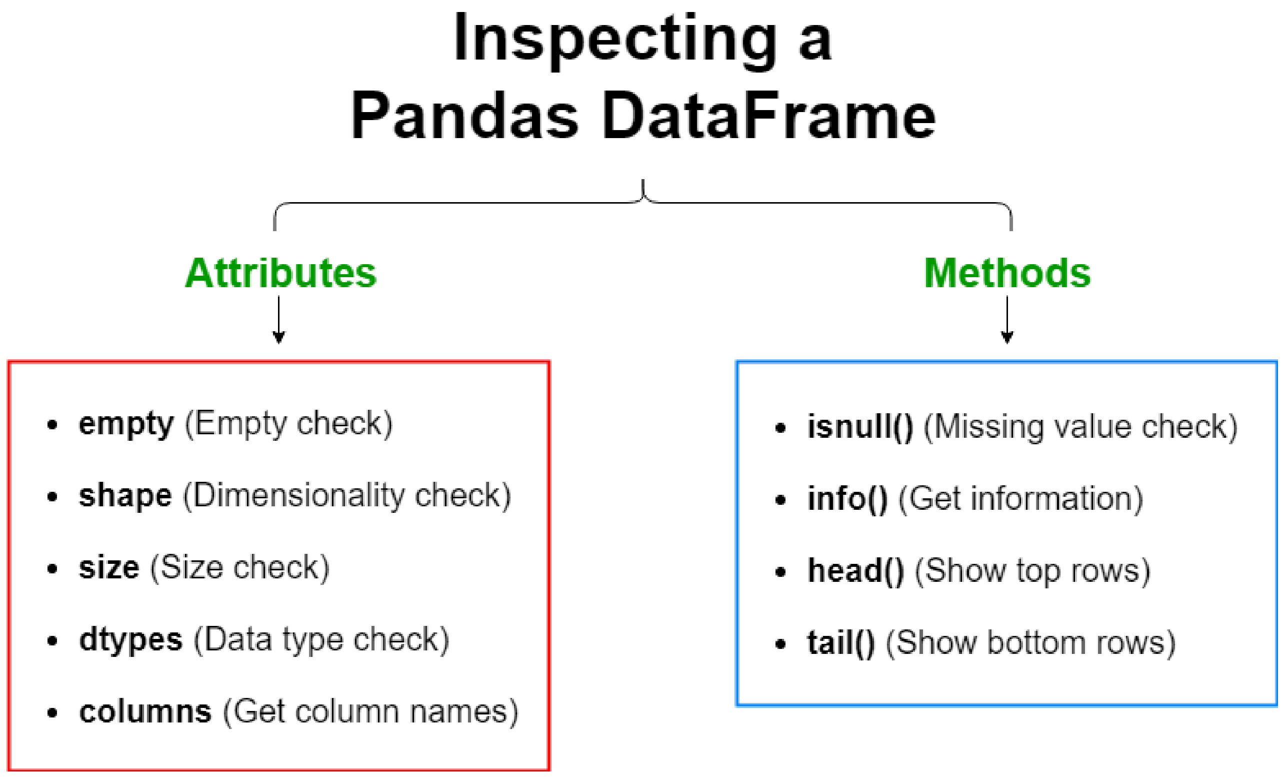
```
pd.read_excel('data.xlsx',sheet_name='Sheet1', index_col=None, na_values=['NA'])

pd.read_stata('data.dta')

pd.read_sas('data.sas7bdat')

pd.read_hdf('data.h5','df')
```

Dataframes – data types

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python’s standard library)	Values meant to hold time data. Look into these for time series experiments.

Dataframes – attributes and methods

- Like Python objects, dataframes have *attributes* and *methods*.



Attributes:
- **empty** (Empty check)
- **shape** (Dimensionality check)
- **size** (Size check)
- **dtypes** (Data type check)
- **columns** (Get column names)

Methods:
- **isnull()** (Missing value check)
- **info()** (Get information)
- **head()** (Show top rows)
- **tail()** (Show bottom rows)

All attributes and methods can be listed with a *dir()* function: `dir(df)`

Activity

- Load the iris dataset at

  https://media.githubusercontent.com/media/neurospin/pystatsml/master/datasets/iris.csv

- Inspect the dataset:
  - What is the shape, the size?
  - Column names? Data types?
  - Try to read the first and last 10 rows?
  - Display 10 random rows?

Combining dataframes

- Combining Data Across Rows or Columns
  - `concat` function append either columns or rows from one DataFrame to another.

```
df1 = pd.DataFrame({ 'A':[1, 2, 3] , 'B':[4, 5, 6] })
df2 = pd.DataFrame({ 'A':[7, 8, 9] , 'B':[10, 11, 12] })
```

```
Result = pd.concat([df1,df2])
```

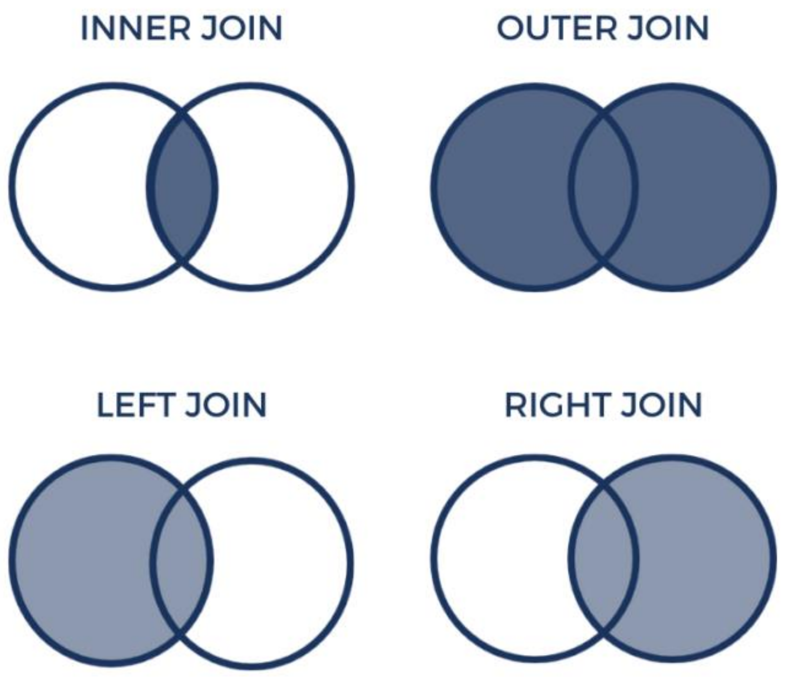
	A	B
0	1	4
1	2	5
2	3	6
0	7	10
1	8	11
2	9	12

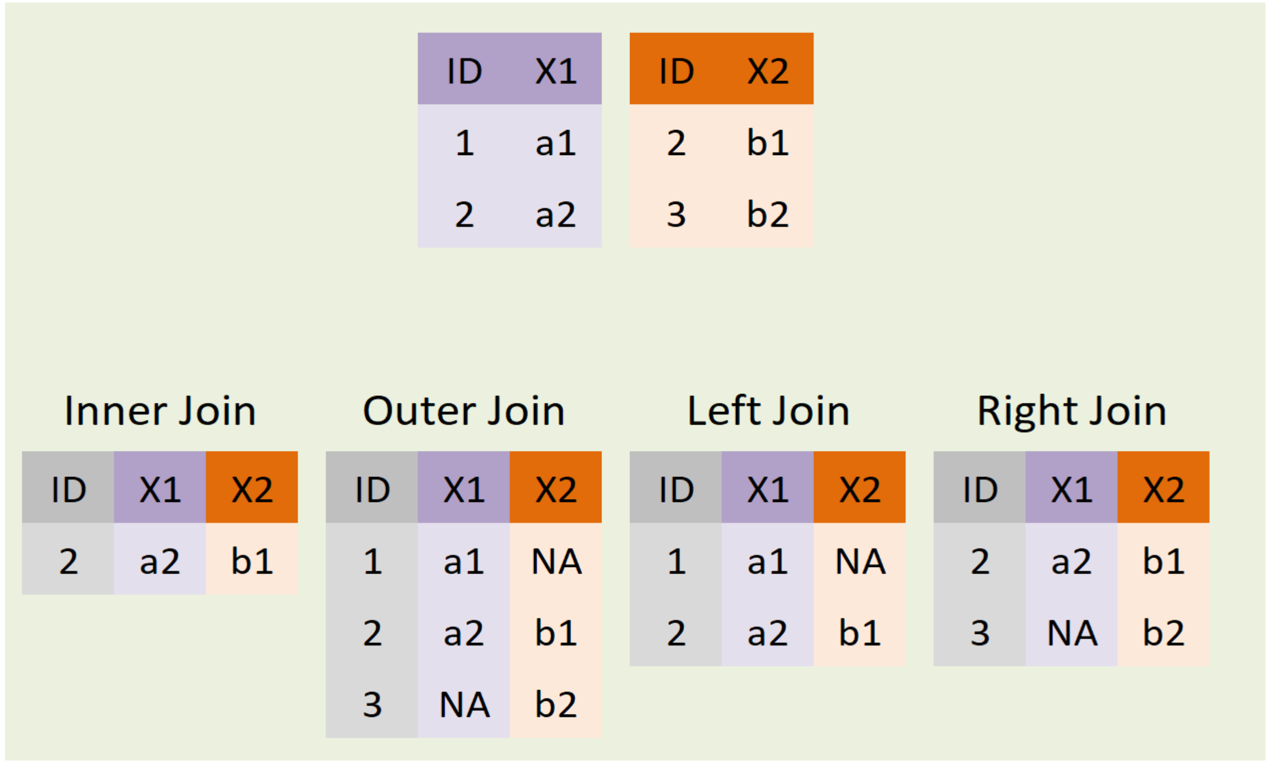
```
Result = pd.concat([df1,df2], axis=1)
```

	A	B	A	B
0	1	4	7	10
1	2	5	8	11
2	3	6	9	12

Combining dataframes

- Database-style joining
  - `merge()` implements common SQL style joining operations.





ID	X1
1	a1
2	a2

ID	X2
2	b1
3	b2

Inner Join

ID	X1	X2
2	a2	b1

Outer Join

ID	X1	X2
1	a1	NA
2	a2	b1
3	NA	b2

Left Join

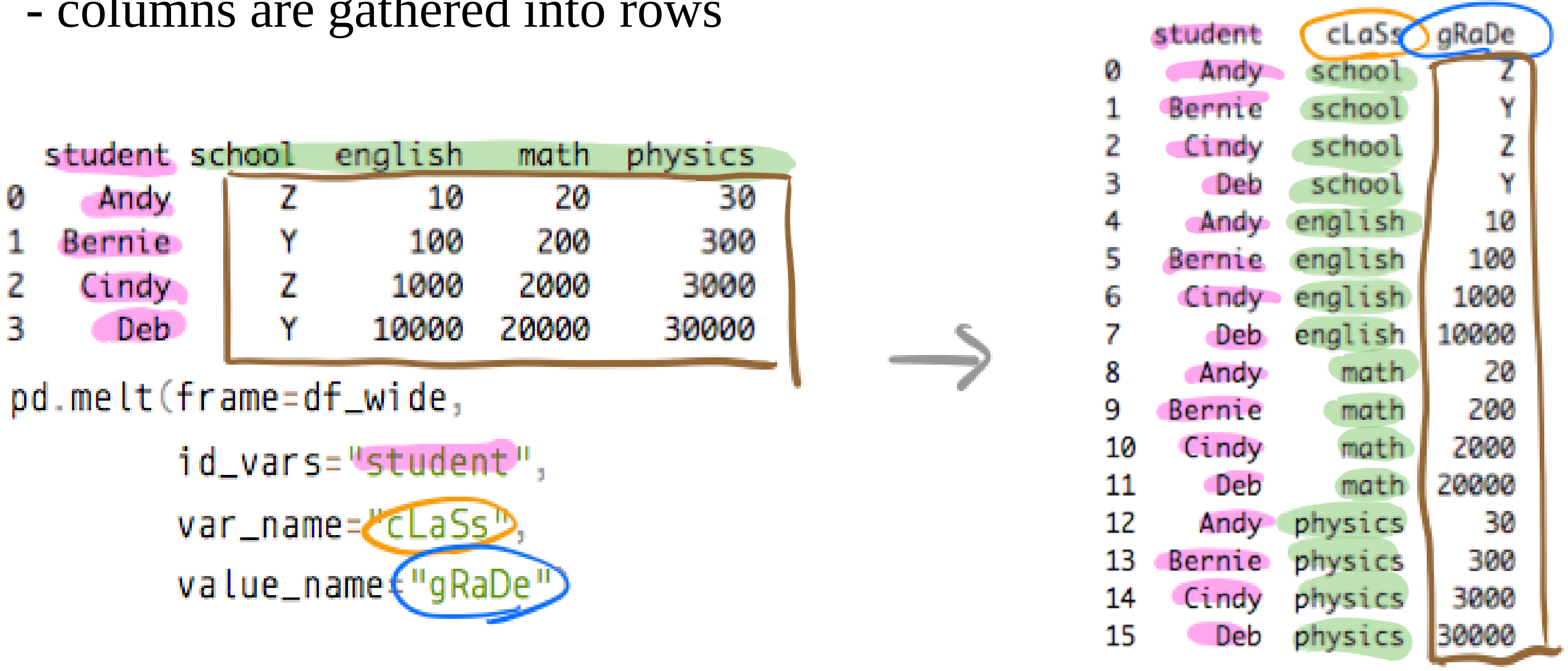
ID	X1	X2
1	a1	NA
2	a2	b1

Right Join

ID	X1	X2
2	a2	b1
3	NA	b2

Reshaping by pivoting

- “Unpivots” a DataFrame from wide format to long (stacked) format
  - columns are gathered into rows

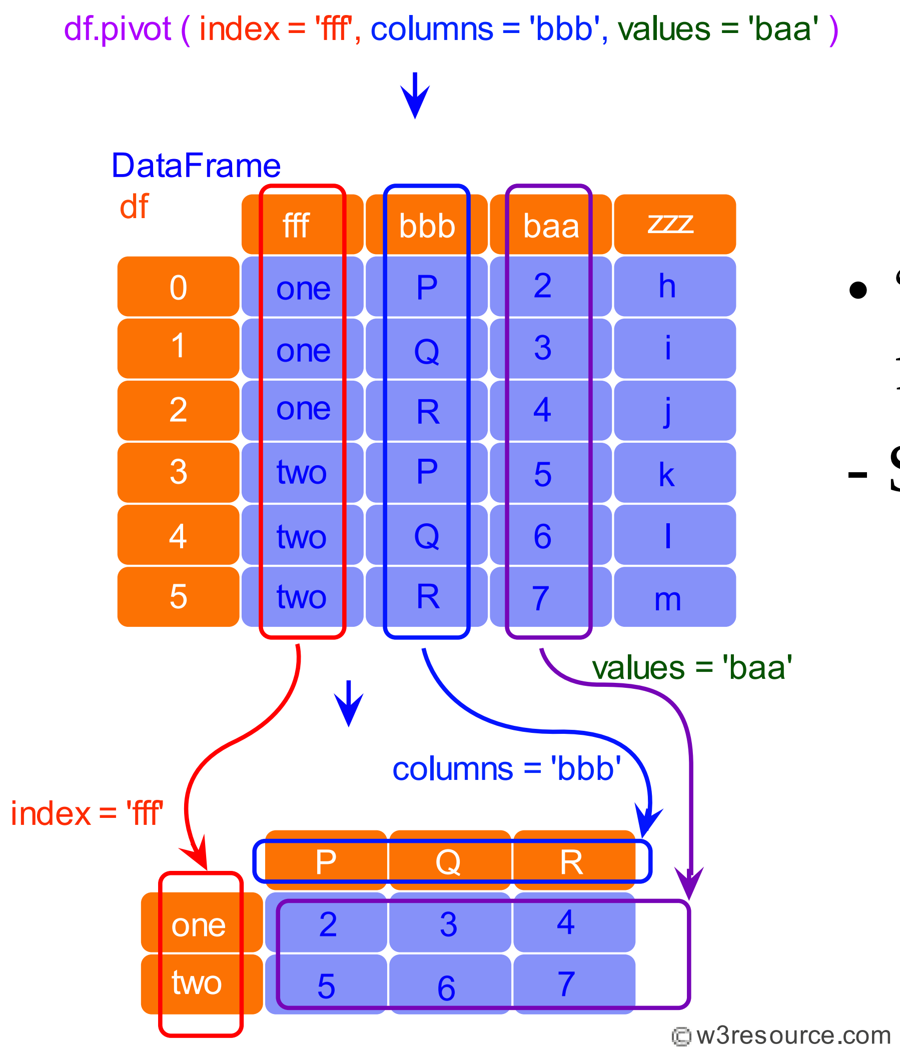


```
   student school  english   math  physics
0     Andy      Z       10     20       30
1   Bernie      Y      100    200      300
2    Cindy      Z     1000   2000     3000
3      Deb      Y    10000  20000    30000
```

```
pd.melt(frame=df_wide,
        id_vars="student",
        var_name="cLaSs",
        value_name="gRaDe")
```

```
   student    cLaSs  gRaDe
0     Andy   school      Z
1   Bernie   school      Y
2    Cindy   school      Z
3      Deb   school      Y
4     Andy  english     10
5   Bernie  english    100
6    Cindy  english   1000
7      Deb  english  10000
8     Andy     math     20
9   Bernie     math    200
10   Cindy     math   2000
11     Deb     math  20000
12    Andy  physics     30
13  Bernie  physics    300
14   Cindy  physics   3000
15     Deb  physics  30000
```

Reshaping by pivoting

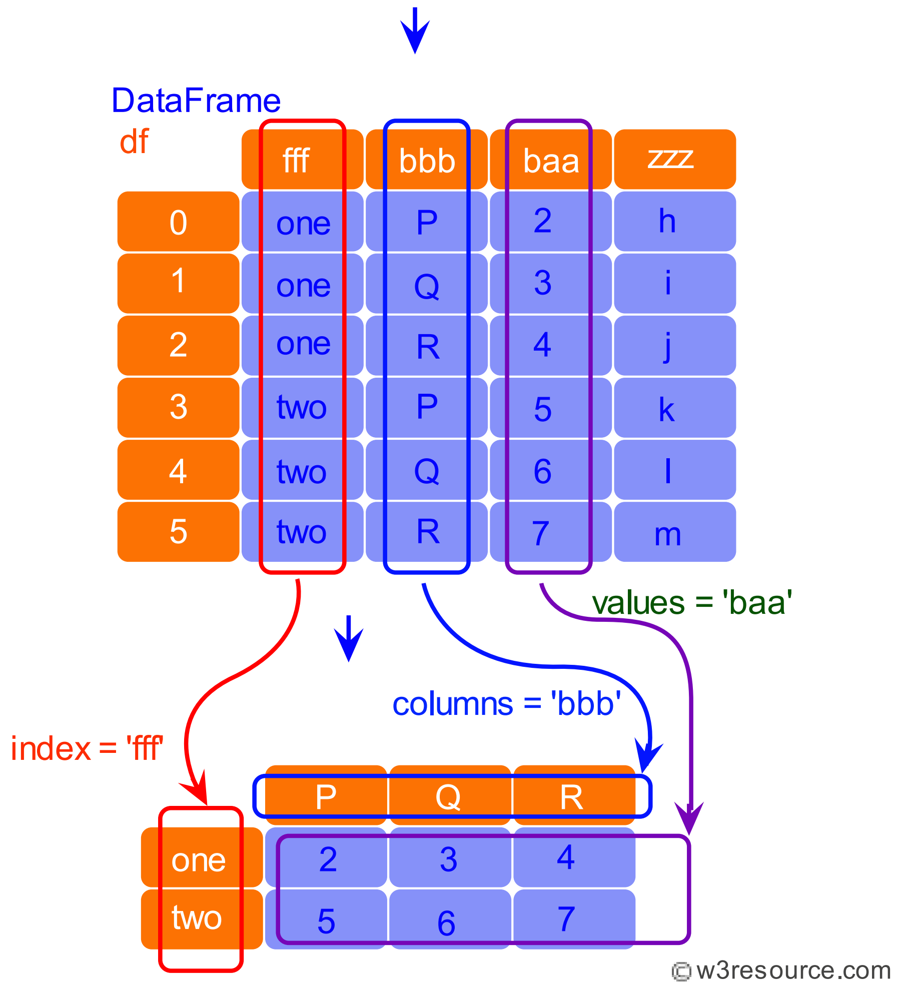


- “pivots” a DataFrame from long (stacked) format to wide format

- Spread rows into columns

pivot vs pivot_table

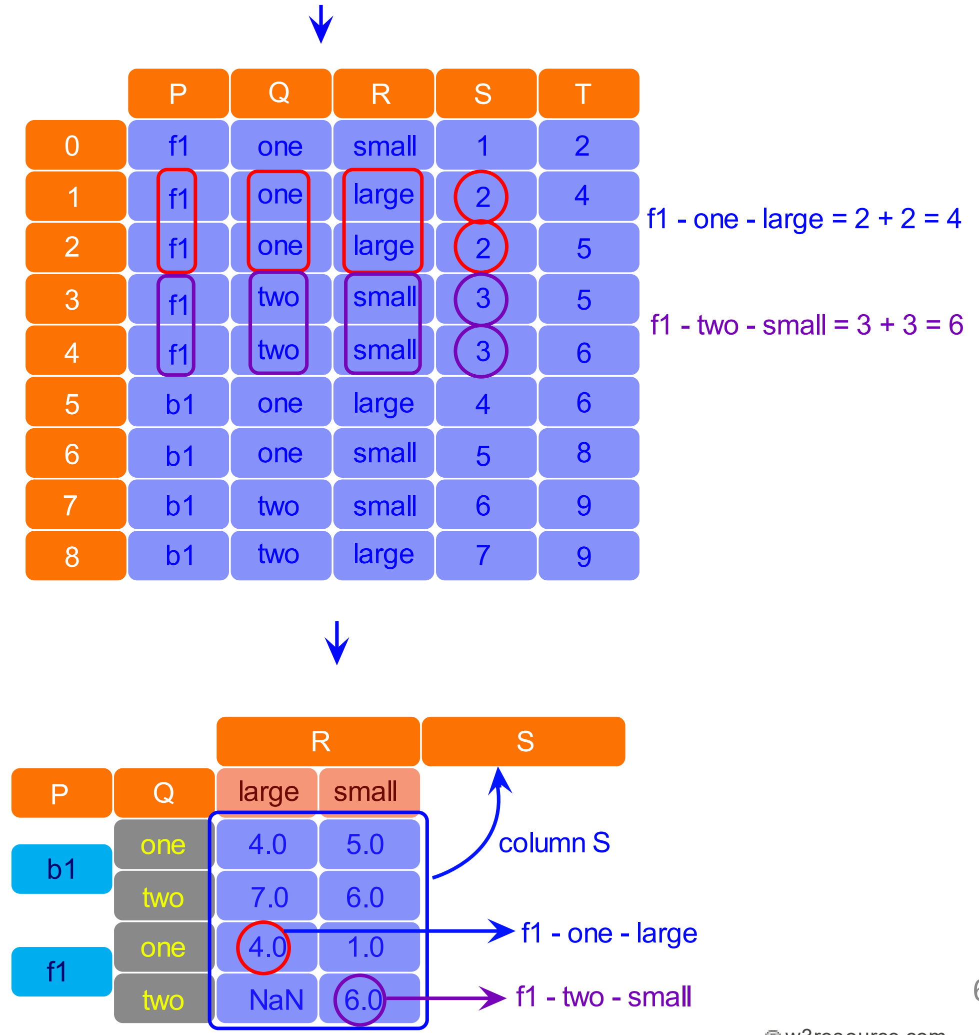
```
df.pivot ( index = 'fff', columns = 'bbb', values = 'baa' )
```



	fff	bbb	baa	zzz
0	one	P	2	h
1	one	Q	3	i
2	one	R	4	j
3	two	P	5	k
4	two	Q	6	l
5	two	R	7	m

	P	Q	R
one	2	3	4
two	5	6	7

```
table = pd.pivot_table ( df, values = 'S', index = [ 'P', 'Q' ],
 columns = [ 'R' ], aggfunc = np.sum )
```



	P	Q	R	S	T
0	f1	one	small	1	2
1	f1	one	large	2	4
2	f1	one	large	2	5
3	f1	two	small	3	5
4	f1	two	small	3	6
5	b1	one	large	4	6
6	b1	one	small	5	8
7	b1	two	small	6	9
8	b1	two	large	7	9

		R		S
P	Q	large	small	
b1	one	4.0	5.0	
	two	7.0	6.0	
f1	one	4.0	1.0	
	two	NaN	6.0	

Dataframe – getting columns and rows

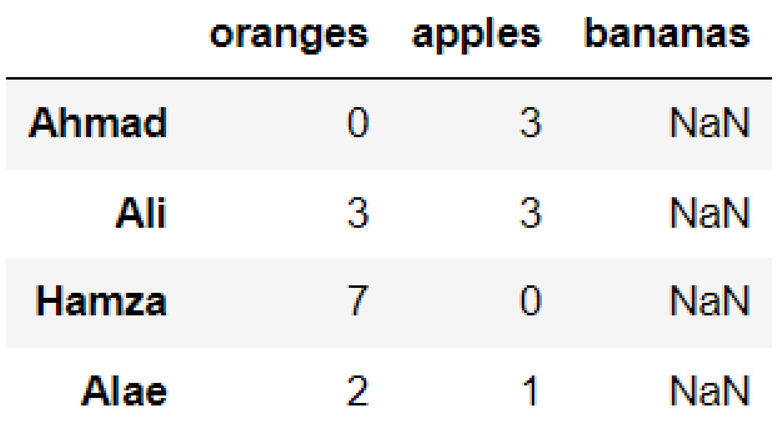
- A column in a DataFrame can be retrieved as a Series by dict-like notation or as attribute

```
df['oranges']
```

```
Ahmad     0
Ali       3
Hamza     7
Alae      2
Name: oranges, dtype: int64
```

```
df.oranges
```

```
Ahmad     0
Ali       3
Hamza     7
Alae      2
Name: oranges, dtype: int64
```



	oranges	apples	bananas
Ahmad	0	3	NaN
Ali	3	3	NaN
Hamza	7	0	NaN
Alae	2	1	NaN

Dataframe – getting columns and rows

- Columns and rows can be retrieved using **`loc`** and **`iloc`**
  - `loc` gets rows (or columns) with particular **labels** from the index.
  - `iloc` gets rows (or columns) at particular **positions** in the index (so it only takes integers).

- **Getting the first row**

```
df.iloc[0]
```

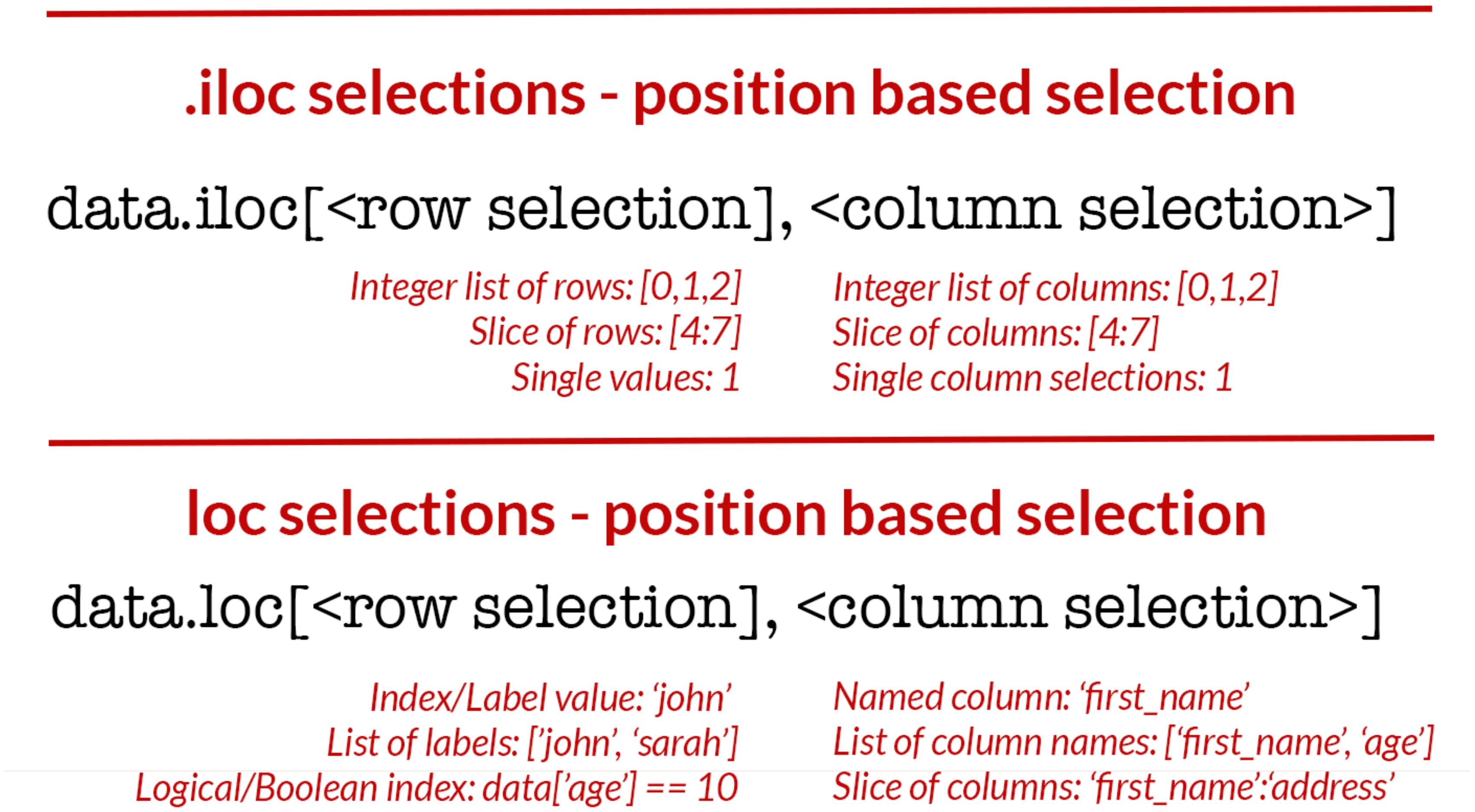
```
oranges      0
apples       3
bananas    NaN
Name: Ahmad, dtype: object
```

- **Getting the row with an index label**

```
df.loc['Ahmad']
```

```
oranges      0
apples       3
bananas    NaN
Name: Ahmad, dtype: object
```

Dataframe – getting columns and rows



.iloc selections - position based selection

```
data.iloc[<row selection>, <column selection>]
```

Integer list of rows: [0,1,2]	*Integer list of columns: [0,1,2]*
Slice of rows: [4:7]	*Slice of columns: [4:7]*
Single values: 1	*Single column selections: 1*

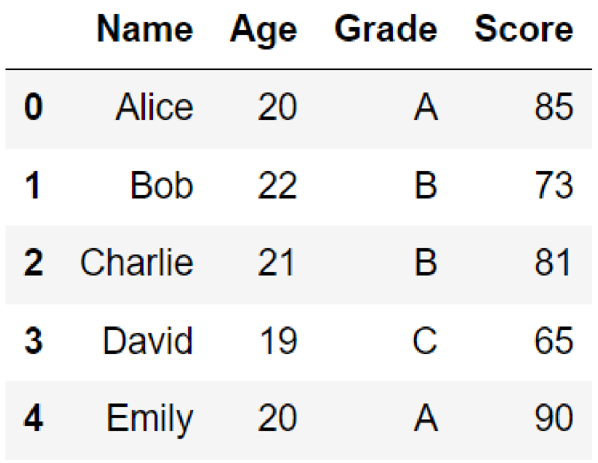
loc selections - position based selection

```
data.loc[<row selection>, <column selection>]
```

Index/Label value: 'john'	*Named column: 'first_name'*
List of labels: ['john', 'sarah']	*List of column names: ['first_name', 'age']*
Logical/Boolean index: data['age'] == 10	*Slice of columns: 'first_name':'address'*

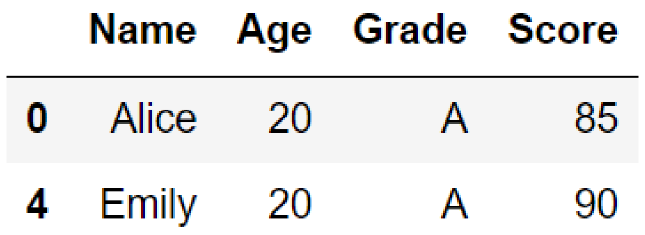
Rows selection (logic filtering)

- Selection based on conditions
  - Boolean indexing is commonly known as a filter.



	Name	Age	Grade	Score
0	Alice	20	A	85
1	Bob	22	B	73
2	Charlie	21	B	81
3	David	19	C	65
4	Emily	20	A	90

```
df[(df['Grade'] == 'A') & (df['Score'] > 80)]
```



	Name	Age	Grade	Score
0	Alice	20	A	85
4	Emily	20	A	90

Any Boolean operator can be used to subset the data:

$>$ greater; $>=$ greater or equal;

$<$ less; $<=$ less or equal;

$==$ equal; $!=$ not equal;

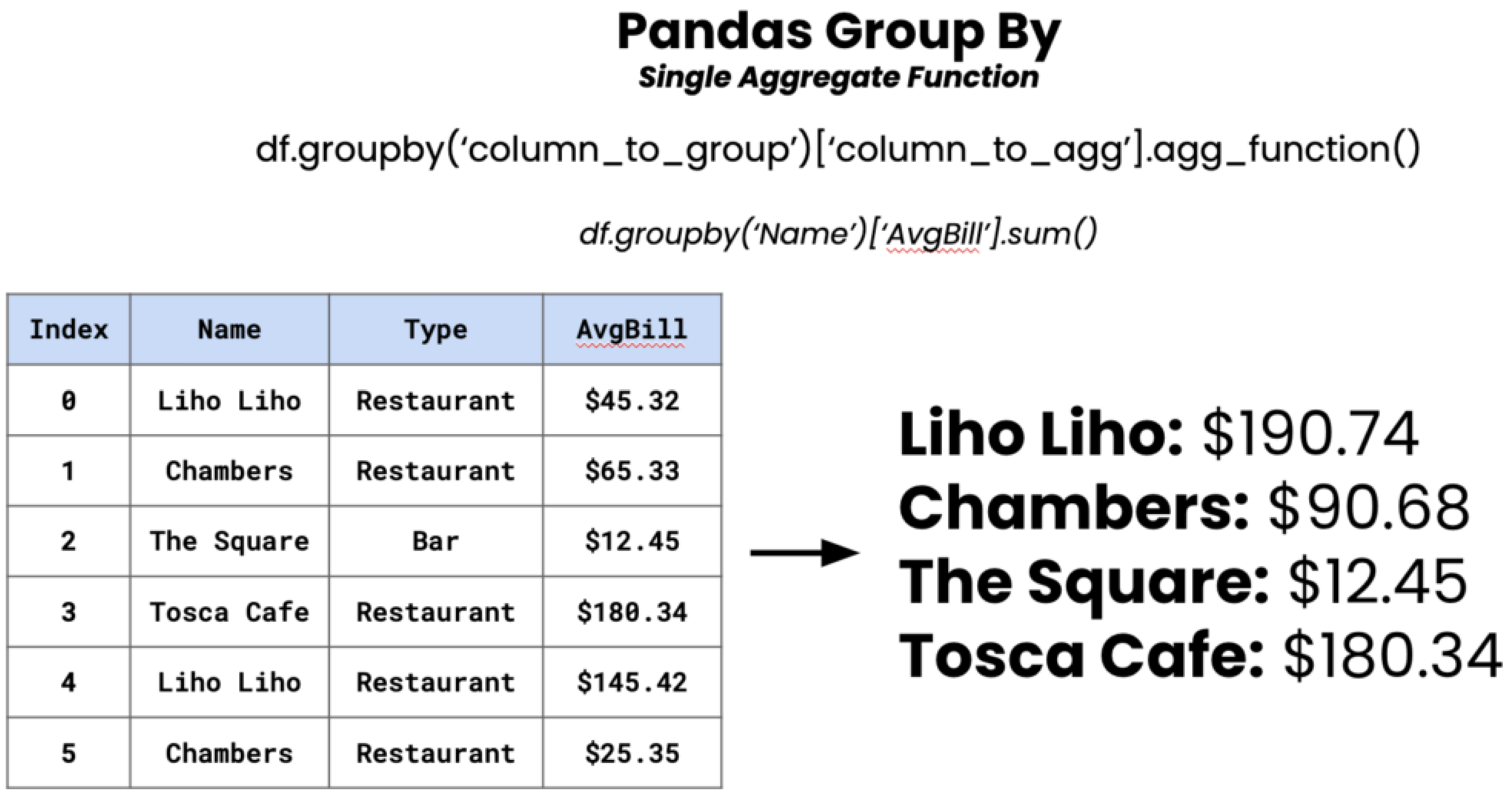
Activity

- In the iris dataset
  - Select rows where the species is '*setosa*'.
  - Select rows where petal width is greater than 2.0.
  - Select rows where sepal length is between 5.0 and 6.0.
  - Create a new column that represents the ratio of sepal length to sepal width.

Groupby

- Using "group by" method we can:
  - Split the data into groups based on some criteria
  - Calculate statistics (or apply a function) to each group

Groupby



Pandas Group By

Single Aggregate Function

```
df.groupby('column_to_group')['column_to_agg'].agg_function()
```

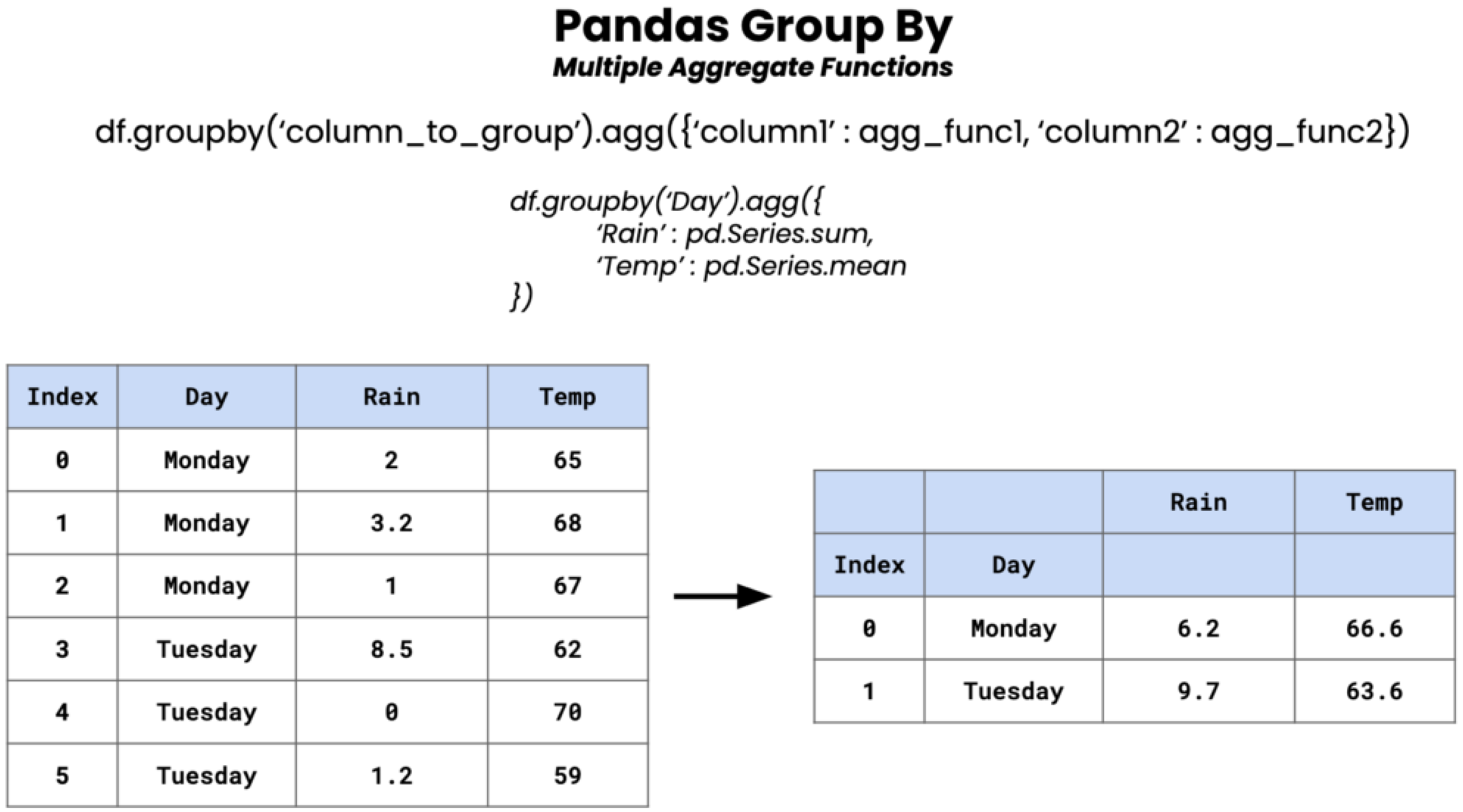
df.groupby('Name')['AvgBill'].sum()

Index	Name	Type	AvgBill
0	Liho Liho	Restaurant	$45.32
1	Chambers	Restaurant	$65.33
2	The Square	Bar	$12.45
3	Tosca Cafe	Restaurant	$180.34
4	Liho Liho	Restaurant	$145.42
5	Chambers	Restaurant	$25.35

$\longrightarrow$

Liho Liho: $190.74
Chambers: $90.68
The Square: $12.45
Tosca Cafe: $180.34

Groupby



Pandas Group By

Multiple Aggregate Functions

```
df.groupby('column_to_group').agg({'column1' : agg_func1, 'column2' : agg_func2})
```

```
df.groupby('Day').agg({
        'Rain' : pd.Series.sum,
        'Temp' : pd.Series.mean
})
```

Index	Day	Rain	Temp
0	Monday	2	65
1	Monday	3.2	68
2	Monday	1	67
3	Tuesday	8.5	62
4	Tuesday	0	70
5	Tuesday	1.2	59

		Rain	Temp
Index	Day		
0	Monday	6.2	66.6
1	Tuesday	9.7	63.6

Activity

- In the iris dataset
  - Group the data by species and calculate the mean of each feature for each species.
  - Calculate the count of each species in the dataset.

Quality check

- **Identifying and removing duplicates**
  - **`df.duplicated()`**: returns a boolean Series indicating duplicate rows.
    - By default, it marks all duplicates as `True` except for the first occurrence.
  - **`df.drop_duplicates()`**: removes duplicate rows from the DataFrame.
    - `‘keep’` parameter determines which duplicates to keep (default is 'first'). Values can be 'first', 'last', or False.
    - `‘subset’` parameter: Allows specifying columns to consider for identifying and dropping duplicates.

Quality check

- **Identifying and handling missing values (NaN)**
  - **`df.isnull():`** Check for missing and returns a dataframe of boolean values (True where missing, False otherwise).
  - **`df.notnull():`** check for non-missing values. Opposite to isnull().
  - **`df.dropna():`** Drop missing observations
  - **`df.dropna(how='all’):`** Drop observations where all cells is NA
  - **`df.dropna(axis=1, how='all’):`** Drop column if all the values are missing
  - **`df.dropna(thresh = 5):`** Drop rows that contain less than 5 non-missing values
  - **`df.fillna(0):`** Replace missing values with zeros

Lab 2:

Pandas practice

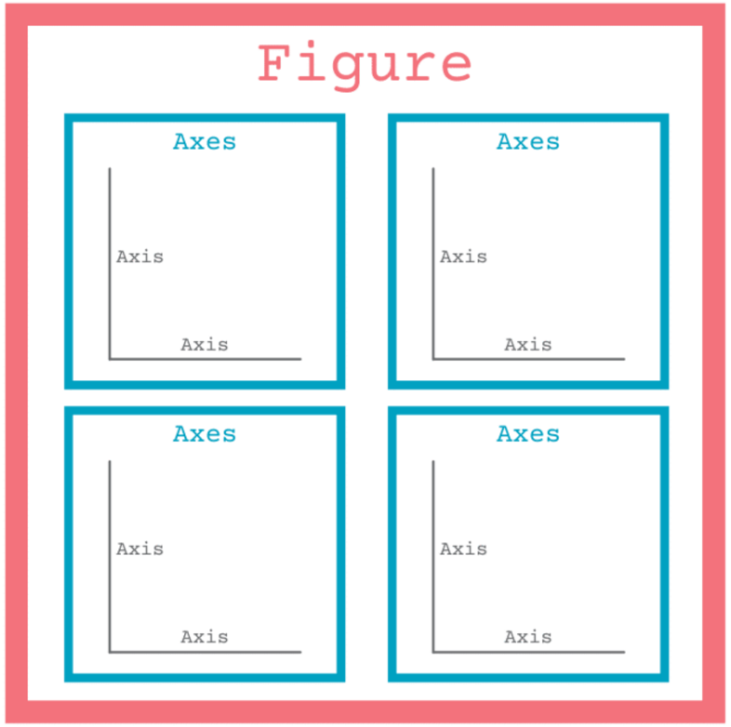
Matplotlib

matplotlib

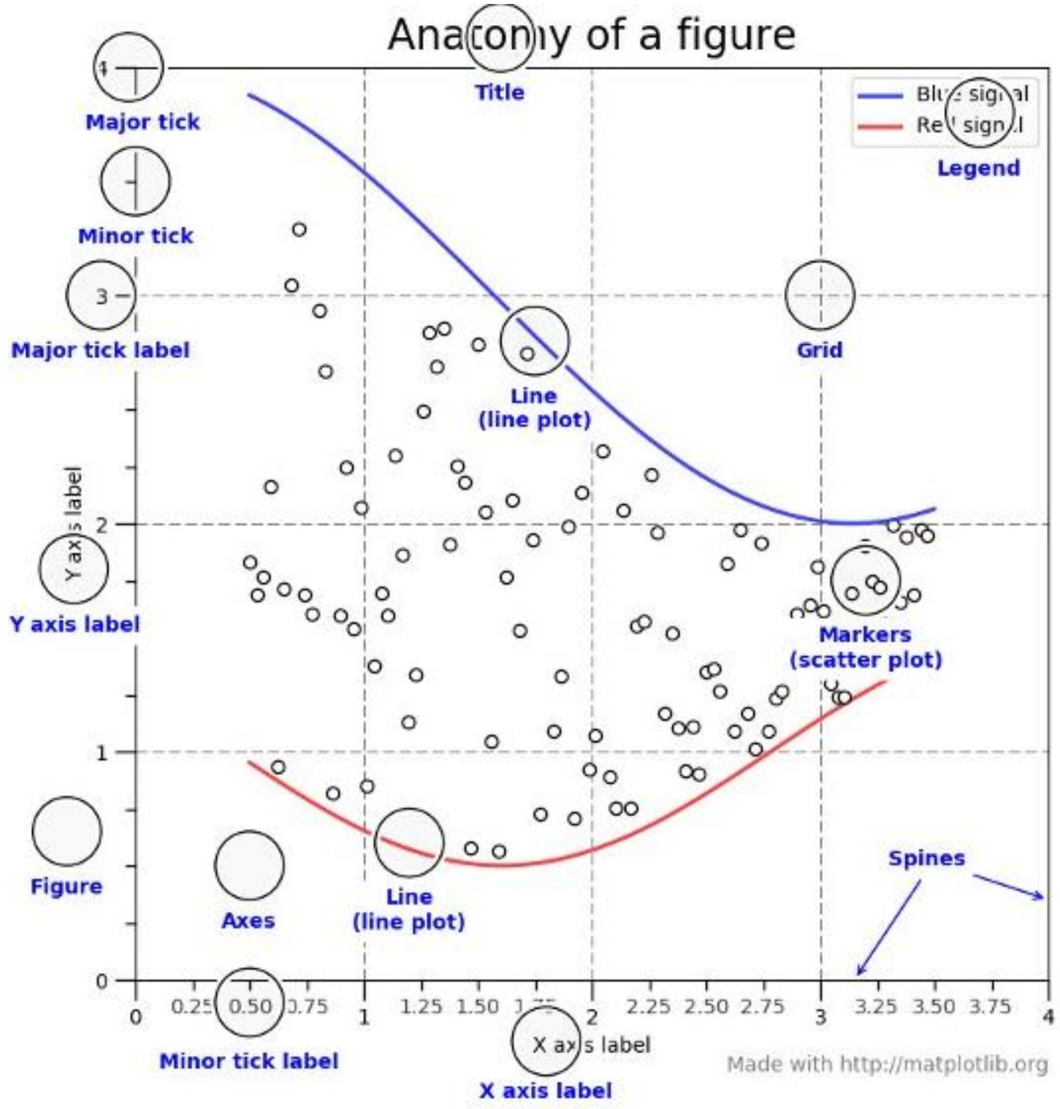
- Matplotlib is one of the most popular Python library used for creating visualizations in various formats.
- It is a cross-platform library for making 2D plots from data in arrays.
- It provides a flexible and comprehensive set of tools for generating plots, charts, and figures.
- It is highly customized: Users have fine-grained control over plot elements, allowing them to customize colors, markers, labels, and other aesthetics.

Matplotlib components

- There are three main parts to a Matplotlib figure:
  - **Figure**: The canvas on which plots are drawn.
  - **Axes**: The actual plot area containing data visualization.
  - **Axis**: The X and Y-axis (and possibly Z-axis) that display the scale of the plot.

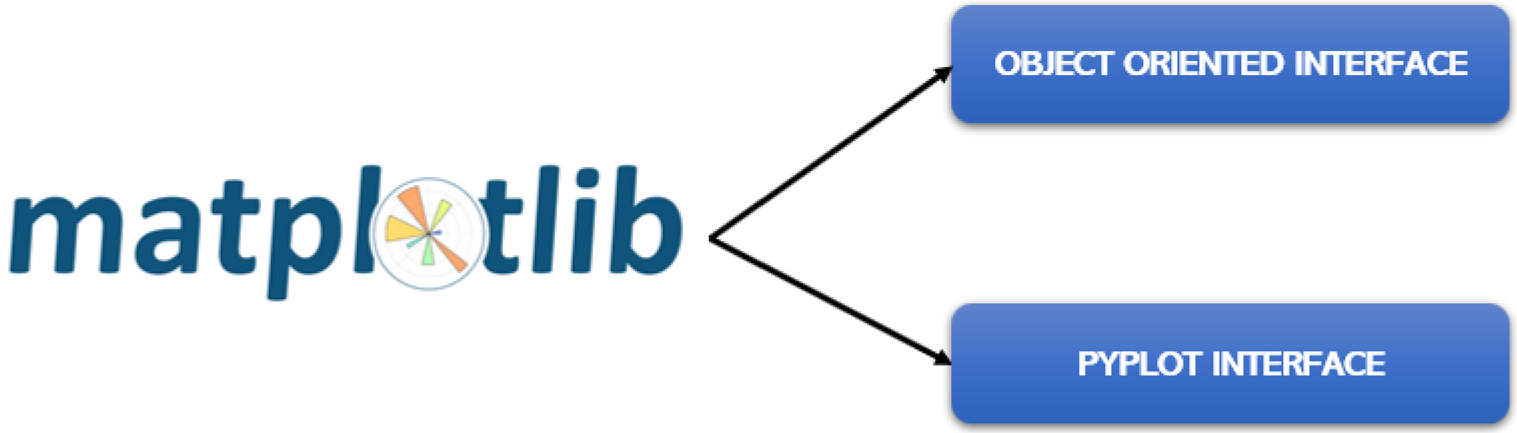


Anatomy of a figure



Matplotlib application interfaces

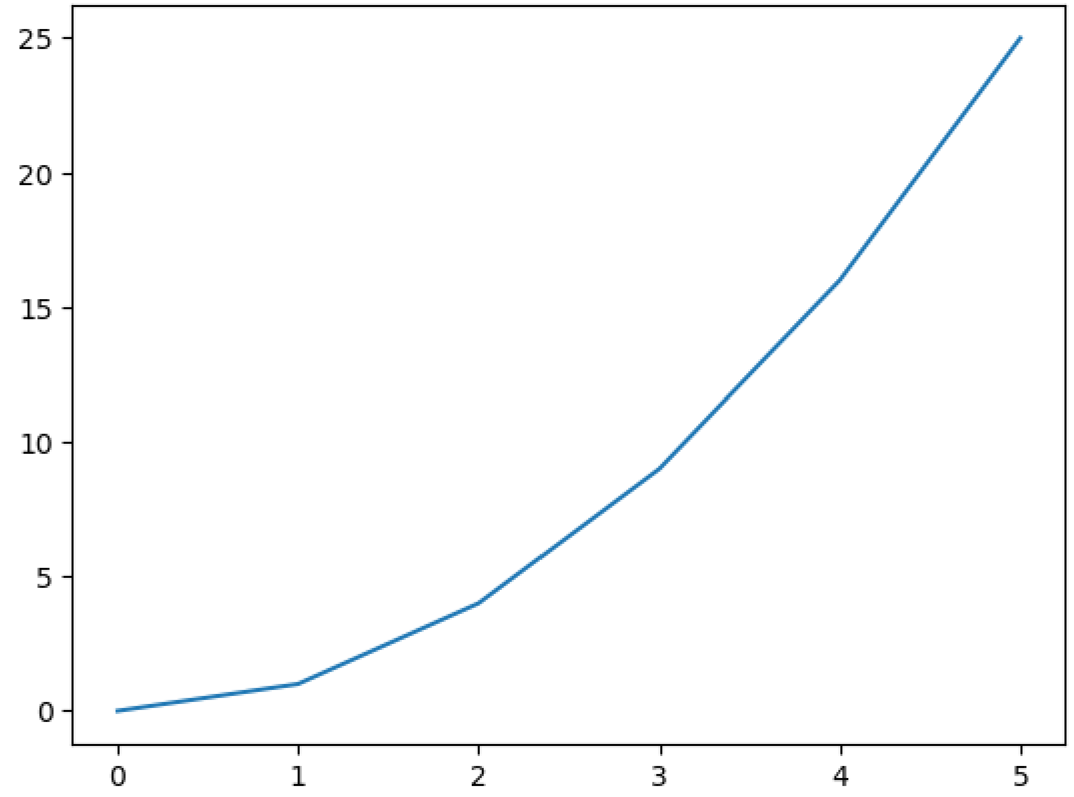
- Matplotlib offers two ways of creating a figure:
  - Explicitly create **Figures** and **Axes**, and call methods on them (the **"object-oriented (OO) style"**).
  - Rely on **pyplot** to implicitly create and manage the **Figures** and **Axes**, and use **pyplot** functions for plotting.



Basic Plotting with pyplot

- Creating a Simple Line Plot

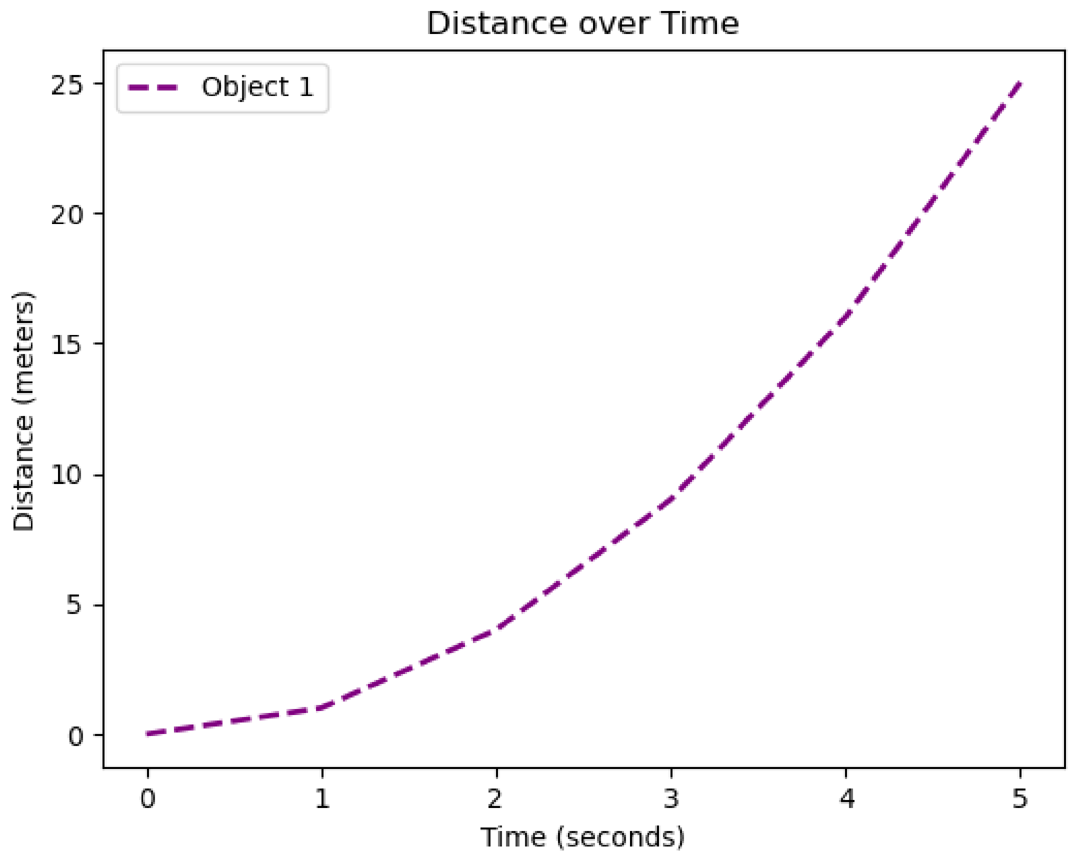
```
import matplotlib.pyplot as plt

# Create two lists, time (0 to 5 seconds)
and distance (in meters)
time = [0, 1, 2, 3, 4, 5]
distance = [0, 1, 4, 9, 16, 25]

# Create a line plot
plt.plot(time, distance)

# Display the plot
plt.show()
```



Customizing plots

- Matplotlib allows us to control line styles, font properties, axes properties, etc.

```
plt.plot(time, distance, color='purple',
linestyle='--', linewidth=2)

# Adding title and labels
plt.title('Distance over Time')
plt.xlabel('Time (seconds)')
plt.ylabel('Distance (meters)')

# Adding a legend
plt.legend(['Object 1'])

# Display the plot
plt.show()
```



Customizing plots

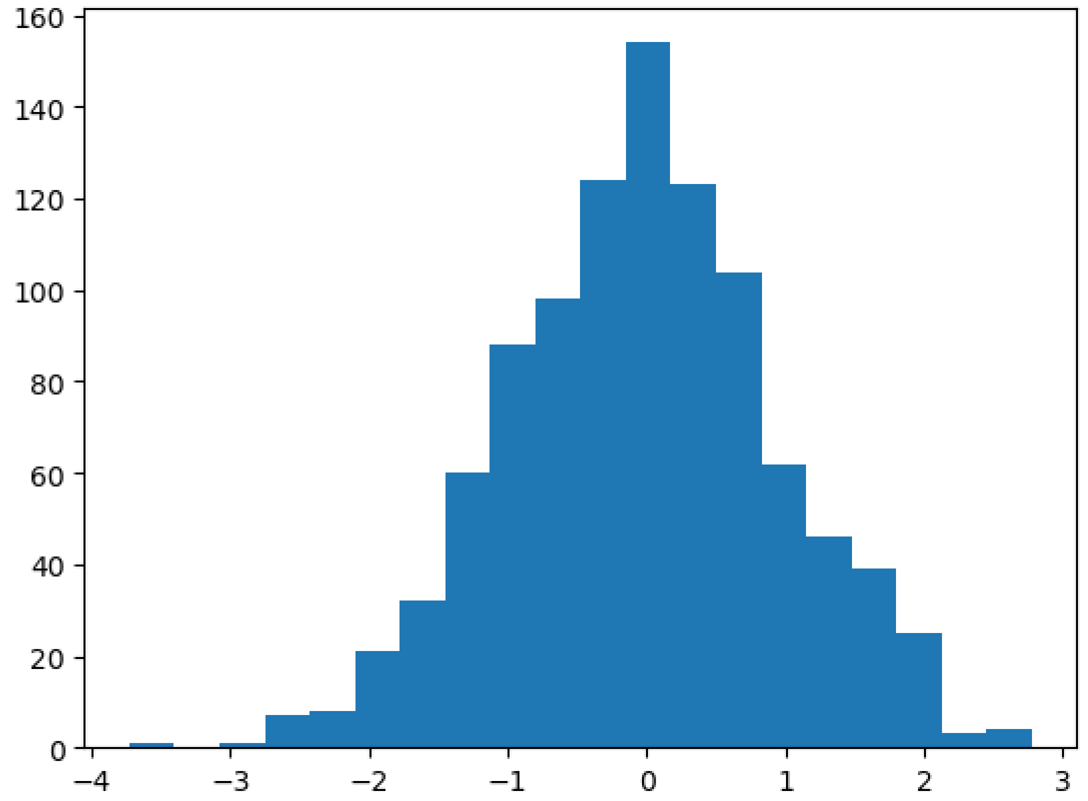
- **`text()`** : adds text in an arbitrary location
- **`xlabel()/ylabel()`**: adds text to the x-axis / y-axis
- **`xlim()/ ylim()`**: setting x and y axis range
- **`title()`** : adds title to the plot
- **`clear()`** : removes all plots from the axes.
- **`savefig()`**: saves your figure to a file
- **`legend()`** : shows a legend on the plot

All methods are available on **`pyplot`** and on the axes instance generally.

Different types of plots

- **Histograms** are useful for understanding the distribution of continuous numerical data.

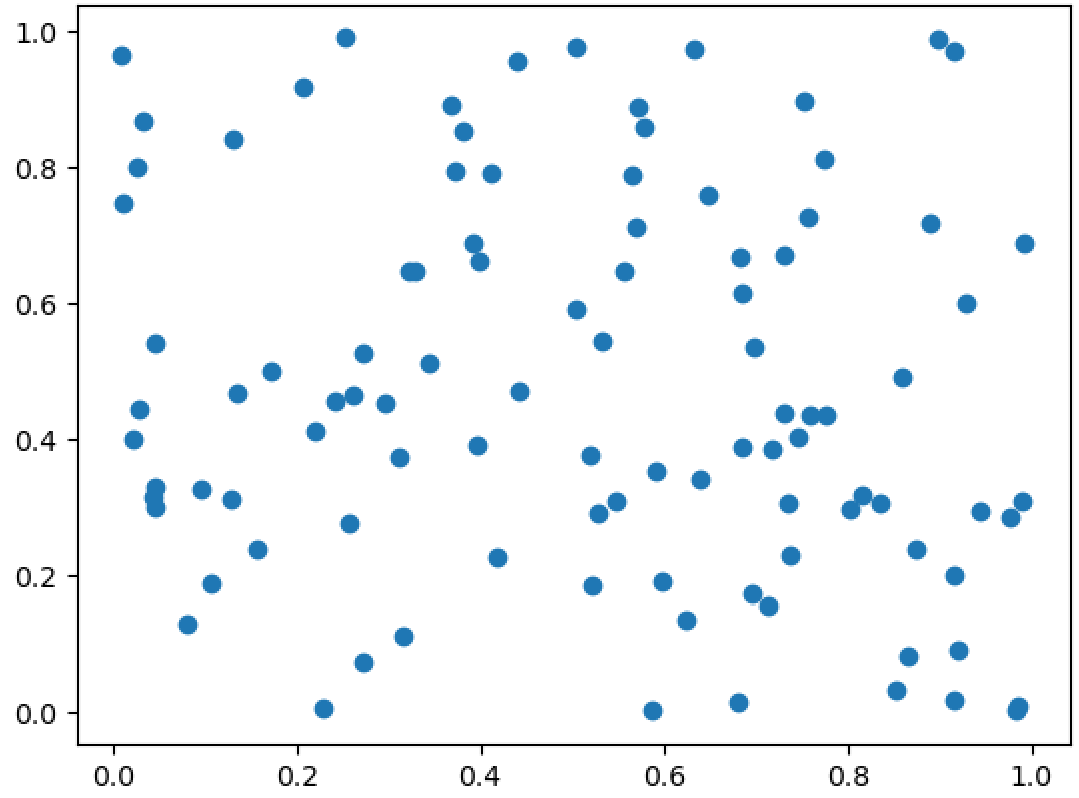
```
# Generating 1000 random values
random_data = np.random.randn(1000)

# Creating histogram
plt.hist(random_data, bins=20)

# Displaying the histogram
plt.show()
```



Different types of plots

- **Scatter Plots** are used to understand the relationship or correlation between two numerical variables.

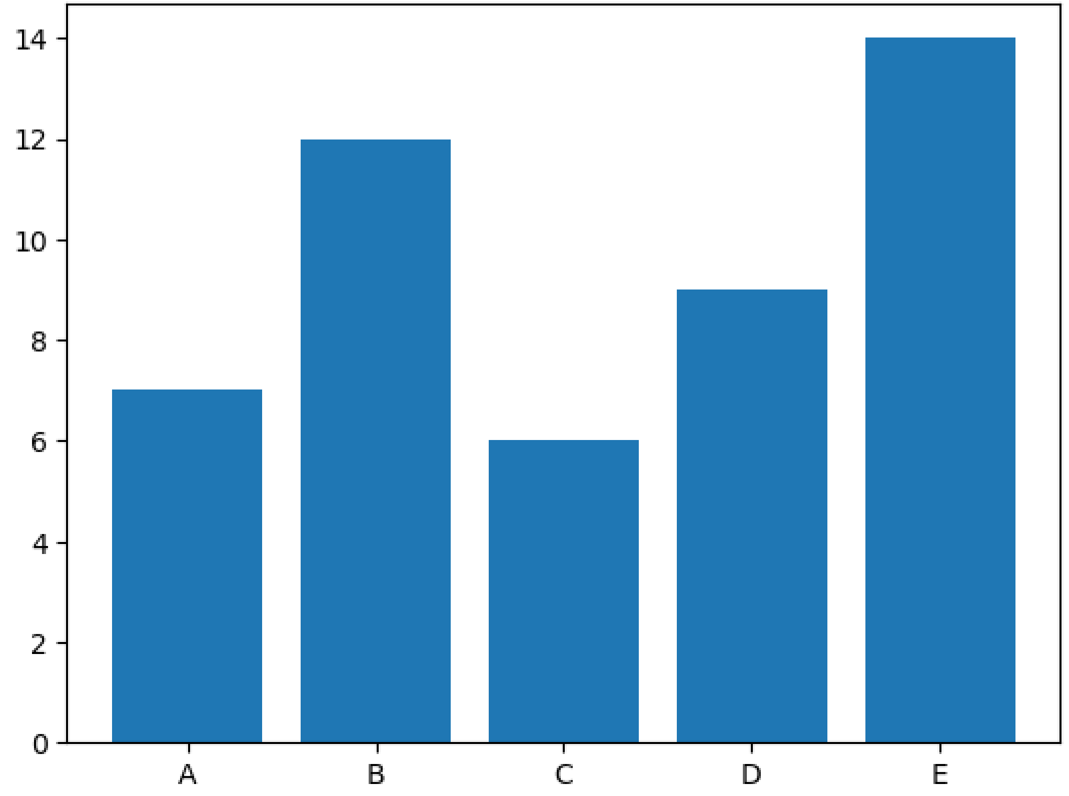
```
# Generating 100 random values for x and y
x = np.random.rand(100)
y = np.random.rand(100)

# Creating scatter plot
plt.scatter(x, y)

# Displaying the scatter plot
plt.show()
```



Different types of plots

- **Bar Plots** are used to compare quantities of different categories.

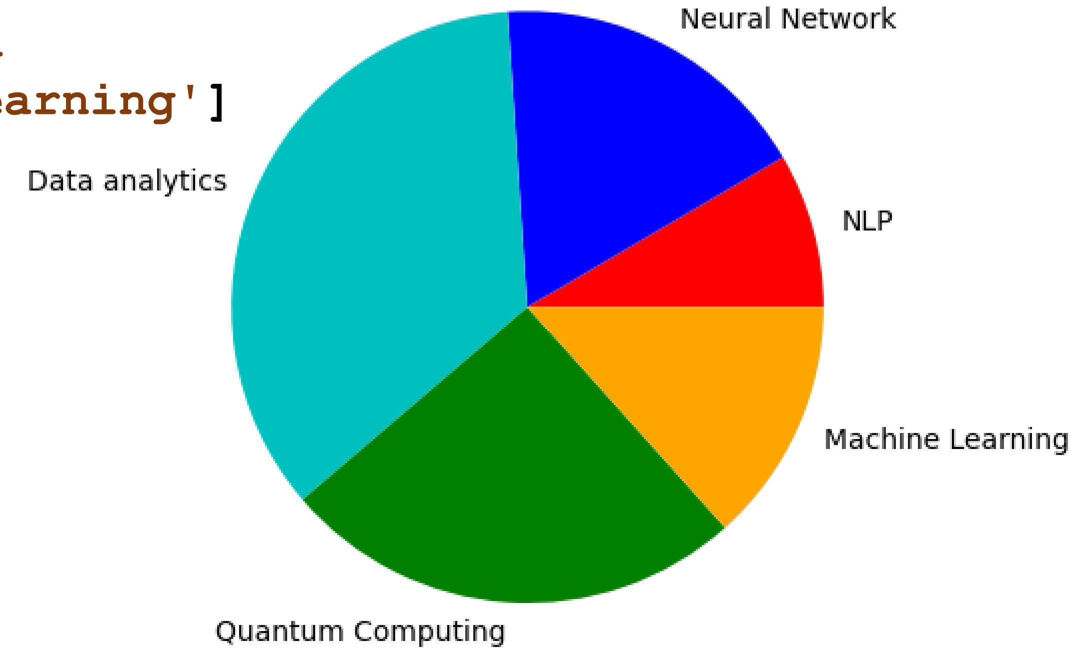
```
# Creating a list of categories and their
values
categories = ['A', 'B', 'C', 'D', 'E'])
values = [7, 12, 6, 9, 14]

# Creating bar plot
plt.bar(categories, values)

# Displaying the bar plot
plt.show()
```



Different types of plots

- **Pie charts** are used to represent the percentage or proportional data in which each pie slice represents an item

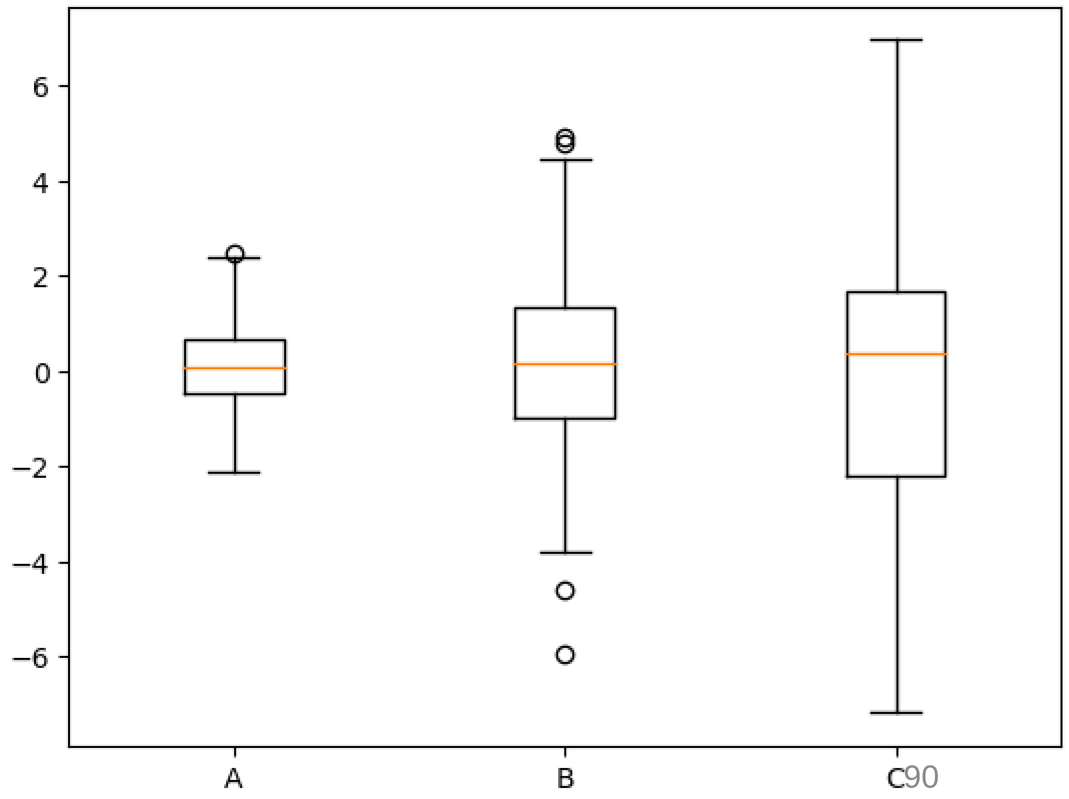
```
# defining labels
activities = ['NLP’, 'Neural Network', 'Data
analytics', 'Quantum Computing', 'Machine Learning']

# portion covered by each label
slice = [12, 25, 50, 36, 19]

# plotting the pie chart
plt.pie(slice,labels =activities)


# Print the chart
plt.show()
```



Different types of plots

- **Boxplots** are used to demonstrate graphically the locality, spread and skewness groups of numerical data through their quartiles.

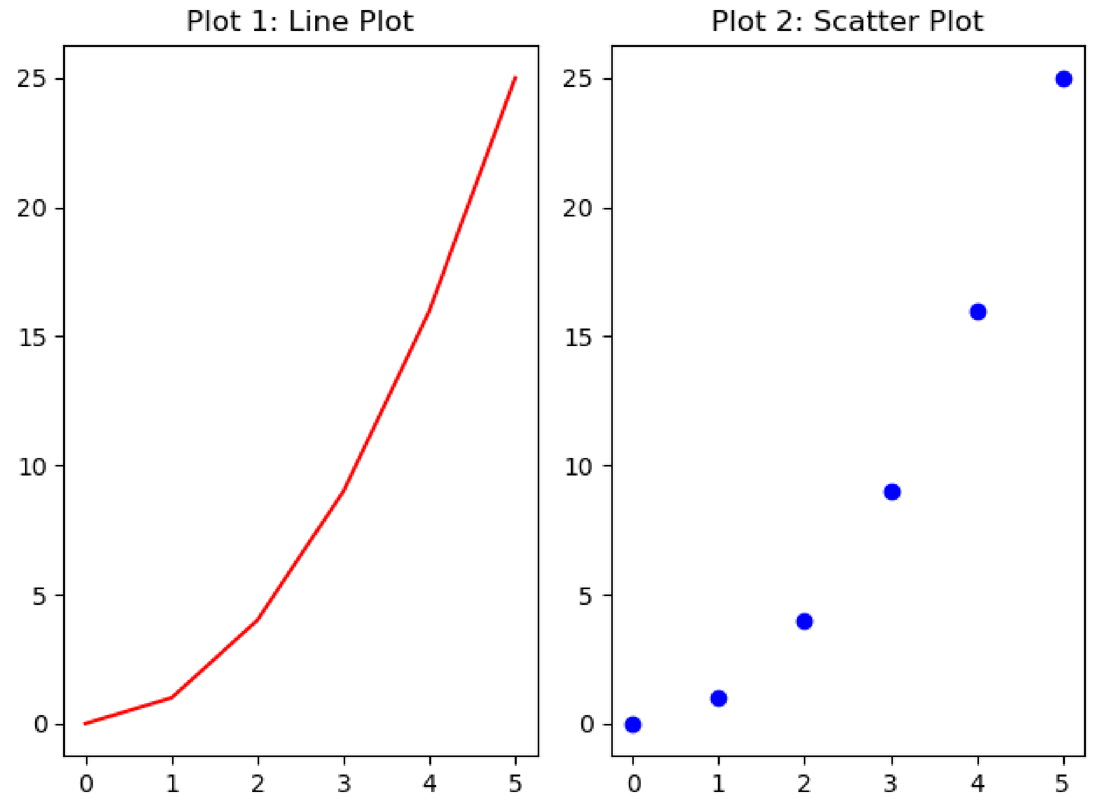
```
# Generating random data
data = [np.random.normal(0, std, 100) for std
in range(1, 4)]

# Creating a boxplot
plt.boxplot(data, labels=['A', 'B', 'C'])

# Displaying the plot
plt.show()
```



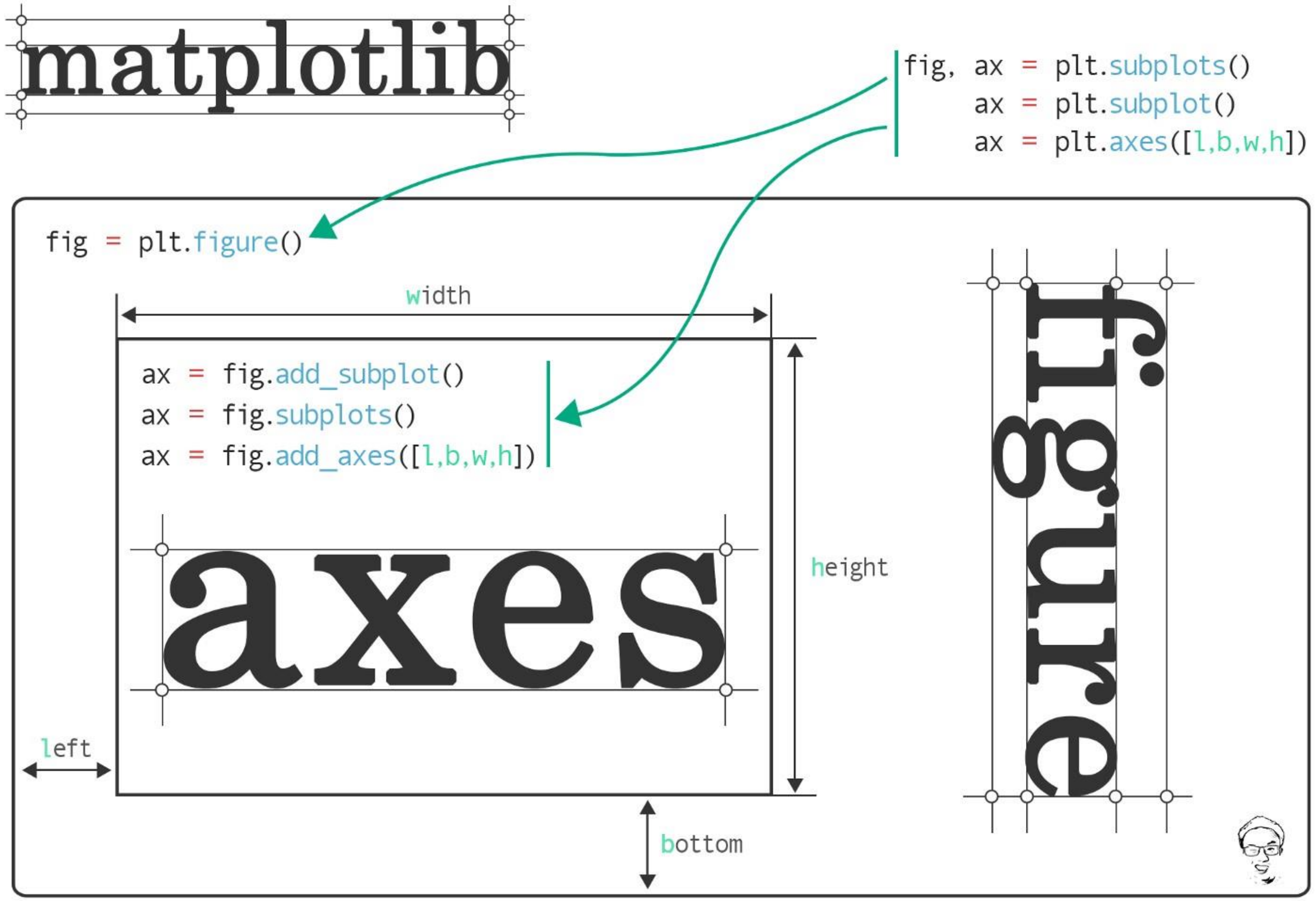
Working with Multiple Plots

- Matplotlib provides the `subplot()` function to display multiple plots in one figure

```
# First subplot
plt.subplot(1, 2, 1)
plt.plot(time, distance, 'r-')
plt.title('Plot 1: Line Plot')

# Second subplot
plt.subplot(1, 2, 2)
plt.scatter(time, distance, color='blue')
plt.title('Plot 2: Scatter Plot')

# Adjust distance between the two plots
plt.tight_layout()

# Show the plots
plt.show()
```



Creating Plots with OO Interface

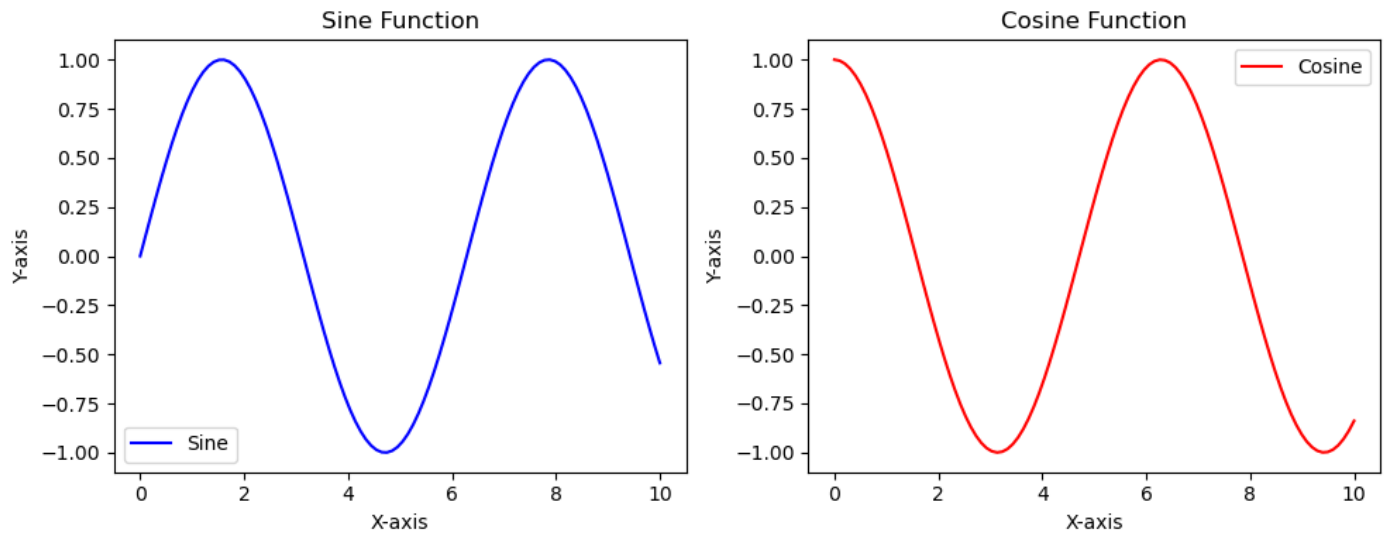


Creating Plots with OO Interface

1. Use **`plt.subplots()`** to create a **Figure** object and an **Axes** object (or multiple axes). Specify the number of rows and column if you want multiple subplots.
2. Use **Axes** Methods to to create plots (e.g., ax.plot(), ax.scatter()…) For multiple subplots, access specific axes objects and plot accordingly.
3. Use various methods of the **Axes** object to customize the plots
4. display the plot using **`plt.show()`**

Plots with OO interface

```
# Data for the plots
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create Figure and Axes objects for subplots
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))  # 1 row, 2 columns

# Plotting on the first subplot (left)
ax[0].plot(x, y1, color='blue', label='Sine')
ax[0].set_title('Sine Function')
ax[0].set_xlabel('X-axis')
ax[0].set_ylabel('Y-axis')
ax[0].legend()

# Plotting on the second subplot (right)
ax[1].plot(x, y2, color='red', label='Cosine')
ax[1].set_title('Cosine Function')
ax[1].set_xlabel('X-axis')
ax[1].set_ylabel('Y-axis')
ax[1].legend()

# Adjust layout for better spacing
plt.tight_layout()

# Display the subplots
plt.show()
```



Matplotlib

- This is just a presentation of essentials of matplotlib. More resources about how you can create more colorful, detailed, and vibrant graphs can be found.
- There are a lot more graphs available in the matplotlib library as well as other popular libraries available in python, including:
  - seaborn
    - https://seaborn.pydata.org/
  - pandas plot (pandas.DataFrame.plot)
    - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html
  - plotly (Plotly Python Open Source Graphing Library)
    - https://plotly.com/python/