Lecture 2

Programming with Python

Python

Pros

- Readability: Python functions from the same library exhibit uniform syntax for better readability.
- Popularity: Python's popularity in data science and other industries ensures abundant learning resources.
- Innovation: New data science models and technologies are constantly added to Python.

Cons

- Consistency: Different libraries may have different syntax conventions.
- Memory: Python uses more computer memory than other programming languages.
- Speed: Other programming languages such as Julia perform computations on datasets more quickly than Python.

Programs, Variables, and Expressions

- A computer **program** executes code line by line. Each line contains an instruction.
  - A **variable** stores a named value.
  - Variables are created via **assignments** (e.g., `predicted_temp = 90`).
  - **Expressions** yield values (e.g., `temp_error = actual_temp - predicted_temp`).

- **Python interpreter** processes Python code, making it human-readable.

- Entering code line by line is inefficient. An **IDE** streamlines code writing, testing, and sharing in data science.

Python Functions

- Data scientists employ pre-defined **functions** for modeling and visualizing data:
  - **Parameters** are inputs like datasets or options, some with defaults.
  - **Arguments** are provided values (e.g., `predict(traffic='high')`).
  - **Returns** are outputs, like numbers, tables, or plots.
- Functions enhance consistency, readability, and efficiency.
- Models with complex formulas benefit from concise function use.

Example; Plot

Data scientist's task: Plot car weight against gas consumption (mpg):

- Use scatterplot() function for a two-feature plot.
- scatterplot() has parameters: data, x, and y.
- Assign 'weight' and 'mpg' as x and y arguments.
- Assign car_data dataset to data parameter.
- Function returns weight on x-axis, mpg on y-axis.

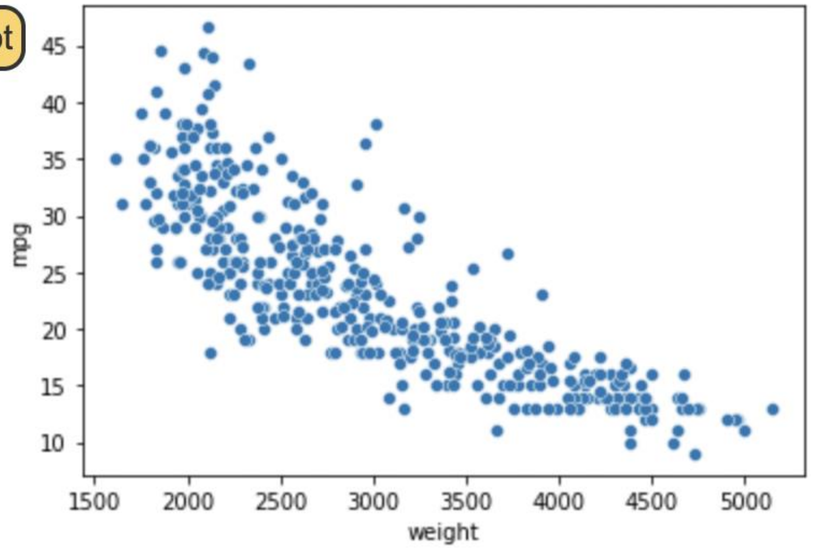
Dataset car_data

mpg	cylinders	displacement	horsepower	weight
18.0	8	307.0	130.0	3504
15.0	8	350.0	165.0	3693
18.0	8	318.0	150.0	3436
16.0	8	304.0	150.0	3433
17.0	8	302.0	140.0	3449

Code

```
scatterplot(data=car_data, x='weight', y='mpg')
```

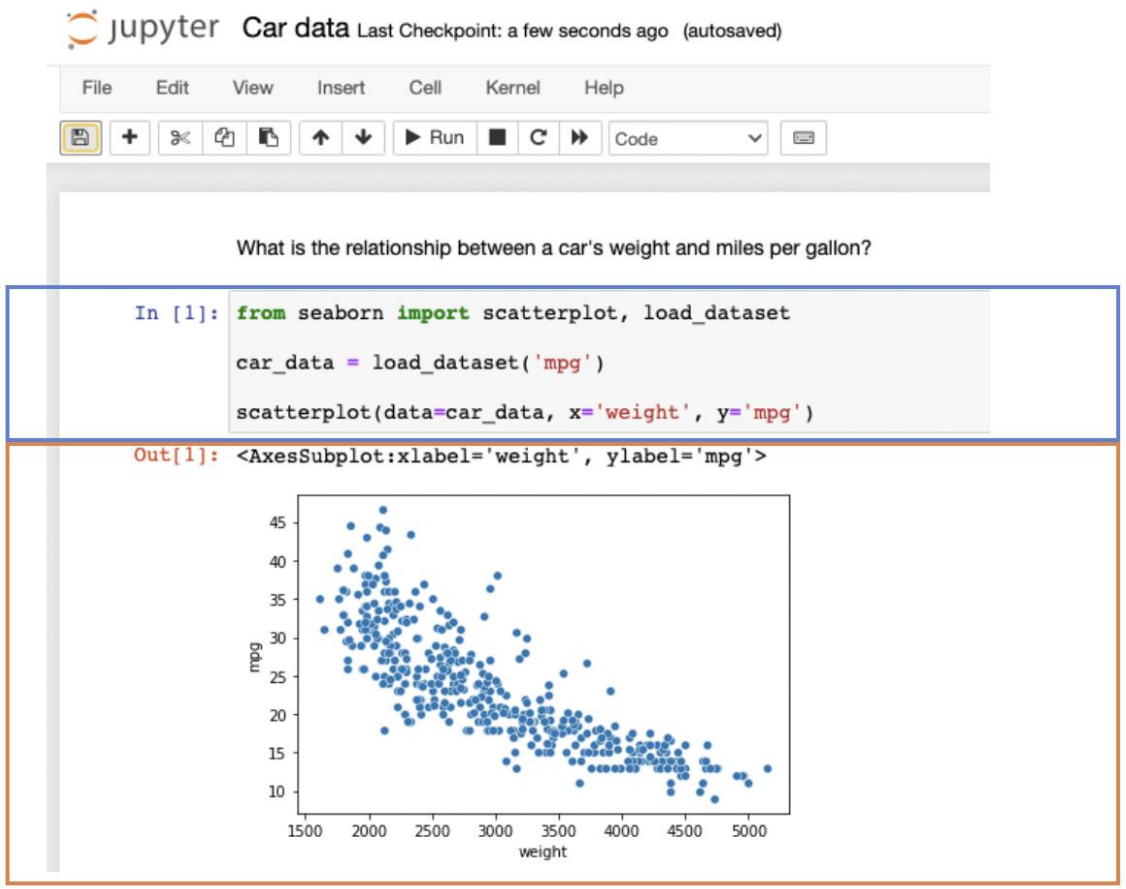
Plot



Example; Jupyter Notebooks

Jupyter notebooks contain three cell types: code, text, and output.

- First cell is text: "Car weight vs. miles per gallon relationship."
- Code cells have numbers and "In" labels.
- Text cells lack labels.
- "Run" button executes code in a code cell.
- Code execution yields scatter plot in an output cell.
- Output cells bear "Out" label, matching code cell number.



What is the relationship between a car's weight and miles per gallon?

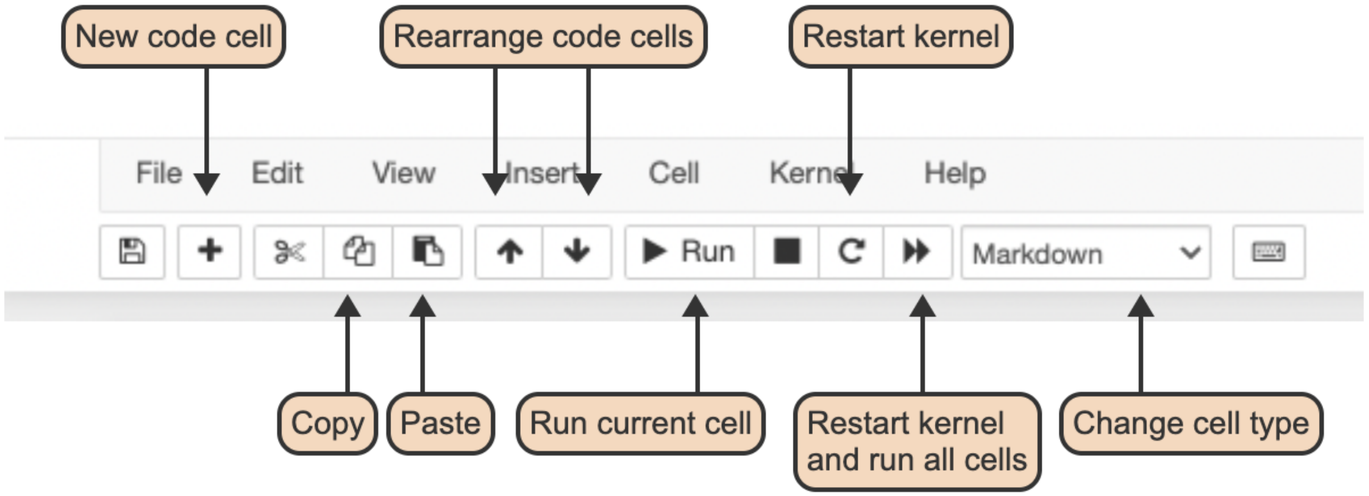
```
In [1]: from seaborn import scatterplot, load_dataset

        car_data = load_dataset('mpg')

        scatterplot(data=car_data, x='weight', y='mpg')
```

```
Out[1]: <AxesSubplot:xlabel='weight', ylabel='mpg'>
```

Jupyter Notebooks

Jupyter notebook menu bar shortcuts:

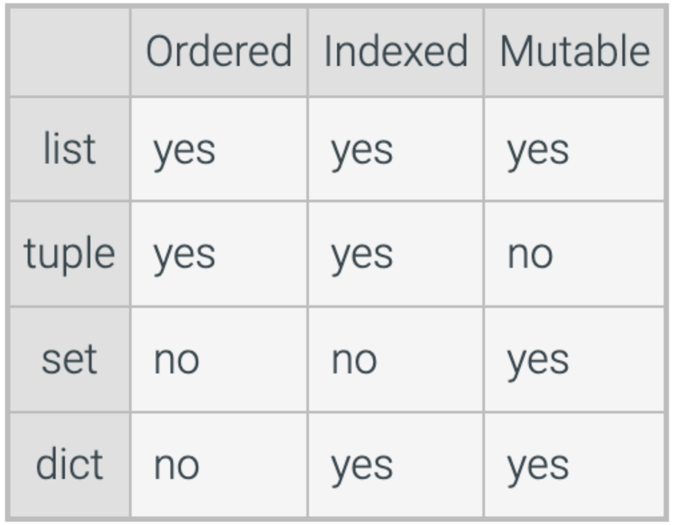
- New Cell adds code below active cell.
- Copy duplicates active code cell.
- Paste inserts copied cell.
- Arrows reorder code cells.
- Restart button refreshes code kernel.
- Run button executes current cell.
- Restart and Run All refreshes and runs all.
- Cell Type dropdown changes input type.



Demo

Programming with Python and R

Python; Data Types



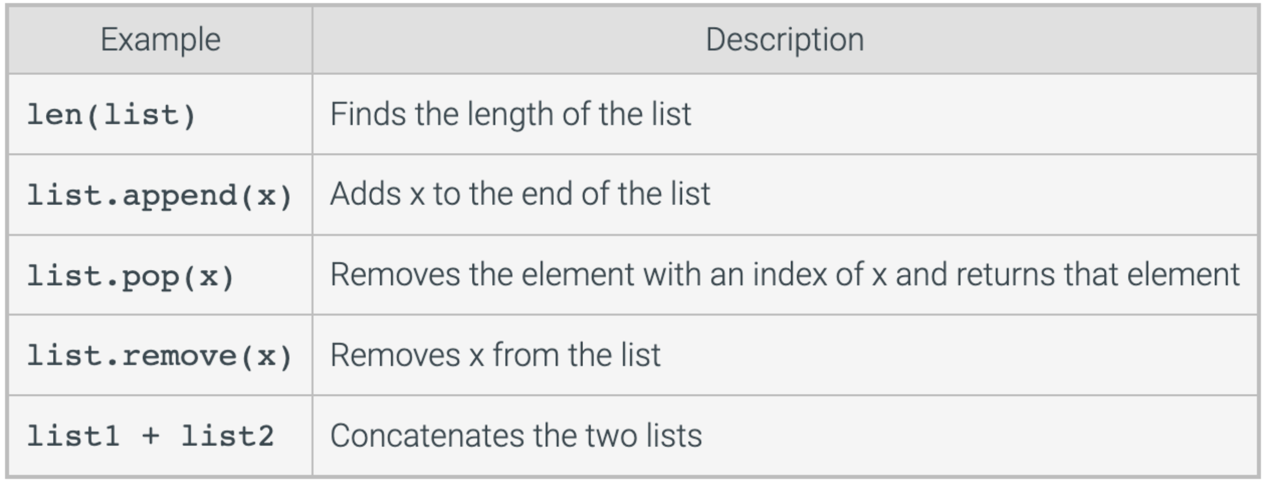
	Ordered	Indexed	Mutable
list	yes	yes	yes
tuple	yes	yes	no
set	no	no	yes
dict	no	yes	yes

Containers in Python hold multiple **elements**, each being a value of any type. Built-in types include list, tuple, set, and dictionary; more can be added with modules. Elements within a container can have varied types, e.g., ['abc', 91, False].

Some containers allow **ordering** and **indexing** (list, tuple, dictionary), while sets don't. The order of elements matters in lists, and indexing allows access (e.g., list[0] = 'abc').

Containers can be **mutable** (changeable) like list, set, dict, or **immutable** like tuple. For instance, list[2] = 3.29 is valid, but tuple[2] = 3.29 isn't.

Python; List Type



Example	Description
`len(list)`	Finds the length of the list
`list.append(x)`	Adds x to the end of the list
`list.pop(x)`	Removes the element with an index of x and returns that element
`list.remove(x)`	Removes x from the list
`list1 + list2`	Concatenates the two lists

List: ordered, indexed, mutable. Create with brackets around variables/literals. Ex:

- list = [10, 'abc'] creates a list with elements 10 and 'abc'.
- list = [] creates an empty list.

List elements are accessed via integer indices in brackets. Indices start at 0. Lists are mutable, allowing updates, additions, and removals. Example: list[0] = 33 replaces the first element with 33.

Python; Tuple Type

Tuples - **Immutable, Ordered, Indexed**:

- Tuples are similar to lists but have a key difference: they are **immutable**, meaning their elements cannot be changed after creation.
- Like lists, tuples maintain order and are accessed using **integer indices**.
- You can't modify, add, or remove individual elements of tuples.
- len(tuple): to get the number of elements
- (tuple1 + tuple2): tuple concatenation.
- Tuples are a good choice when you have elements that are fixed and won't change over time.

Python; Set Type

- A set is a mutable container, unordered and unindexed. Its literal is curly braces, indicating no order or duplicates. Ex:
  - `set = {33, 4,'abc'}` creates a set of three elements.
  - `{33, 4,'abc'}` and `{'abc', 4, 33}` are the same set.
- A string, list, or tuple can be converted to a set with the set() function. Duplicate elements are removed. Ex: `set([33, 4,'abc', 4])` returns `{33, 4,'abc'}`.
- Since sets are not indexed, individual elements cannot be accessed with bracket notation.
- Since sets are mutable, elements can be added or removed.

Python; Dictionary Type

- A dictionary is a mutable container, unordered and indexed with curly braces. Its index can be any immutable type (int, str, tuple), unlike list or tuple indexes. Ex:
    - dict = {'Messi': 10, 'Ronaldo': 7} creates a dictionary with indexes 'Messi' and 'Ronaldo' and values 10 and 7.
    - {'Messi': 10, 'Ronaldo': 7} and {'Ronaldo': 7, 'Messi': 10} are the same dictionary, since dictionaries are not ordered.
- dict = {} creates an empty dictionary.
- Access a dictionary element by appending its index in brackets to the variable name. Being mutable, dictionaries allow updates, additions, and removals. Ex:
- del dict['Rachel'] removes the element with key 'Rachel'.
- dict['Amelia'] = 'A+' either changes the value of an existing 'Amelia' element to 'A+' or adds a new 'Amelia' element.
- The dictionary index is often called a key and the elements are often called key:value pairs.

Python; Functions

- Python functions: built-in or user-defined, defined by a header and body.
- Header: `def` keyword, function name, parameters (optional), colon.
- Body: indented statements, including possible returns.
- Calls: function name, args in parentheses, assigned to params.
- Arguments: expressions for compatible parameter types.
- Call executes body, returns specified expression.
- No `return` or expression = return value `None`.
- Function can return single/multiple element container.

Python; Functions; Example

```python
def calcPizzaVolume(pizzaDiameter, pizzaHeight):
    piVal = 3.14159265
    pizzaRadius = pizzaDiameter / 2.0
    pizzaArea = piVal * pizzaRadius * pizzaRadius
    pizzaVolume = pizzaArea * pizzaHeight
    return pizzaVolume

print('12.0 x 0.3 inch pizza is', calcPizzaVolume(12.0, 0.3), 'cubic inches’)
print('16.0 x 0.8 inch pizza is', calcPizzaVolume(16.0, 0.8), 'cubic inches’)
```

```
>>12.0 x 0.3 inch pizza is 33.9 cubic inches.
>>16.0 x 0.8 inch pizza is 160.9 cubic inches.
```

Python; Scope of Variables/Functions

```
def changeName():
    employeeName = 'Juliet'

employeeName = 'Romeo'
changeName()
print('Employee name:', employeeName)
```

```
>> Employee name: Romeo
```

```
def changeName():
    global employeeName
    employeeName = 'Juliet'

employeeName = 'Romeo'
changeName()
print('Employee name:', employeeName)
```

```
>> Employee name: Juliet
```

1. The changeName() function changes the employeeName variable.
2. employeeName is a local variable inside the function and a global variable in the main program.
3. The function changes the local variable, but the main program prints out the global variable.
4. employeeName is declared global inside the function.
5. Now the function changes the global variable.

Python; Function Objects

- In Python, values are **objects** with a value, data type, and unique identifier, often a memory address.

- Objects are generated when executing value-containing statements (e.g., `print('Hello')`).

- Assigning a variable links it to an object; a variable corresponds to one object, but an object can link to multiple variables.

Python; Pass by Assignments

- Objects are **mutable** or **immutable**.
- Mutable objects' value can change (e.g., list, set, dictionary).
- Immutable objects' value can't change (e.g., integer, float, string).
- Variables can hold mutable or immutable objects.
- Functions assign parameter names with argument objects (**pass by assignment**).
- For mutable arguments, changes can propagate outside the function.
- Immutable arguments can't be changed, only reassigned.
- Functions can alter mutable default parameters.
- Immutable defaults are generally recommended.

Python; Parameter Passing Alternatives

- Many programming languages pass parameters by value or by reference rather than by assignment.

- **Pass by value** assigns parameters with a copy of argument objects. The function cannot access the original objects and therefore cannot change the arguments.

- **Pass by reference** assigns parameters original arguments, similar to pass by assignment. But they differ: in pass by assignment, reassigning creates a new object, not changing the argument. Yet, in pass by reference, reassignment alters the argument object.

- Pass by assignment is sometimes called **pass by object reference**, which is not the same as pass by reference.

Python; Data Science Packages

A Python package extends Python's functionality with functions, classes, and objects loaded from the hard drive.

- Loaded using "import" (e.g., "import pandas").
- Alias with "as" for shorter names (e.g., "import pandas as pd").
- Use dot notation to access functions
  - e.g., "pd.read_csv('datafile.csv')"
- Import a single function with "from”
  - e.g., "from sklearn.preprocessing import StandardScaler".

Python; Obtaining Packages

Python has numerous packages not pre-installed. Packages are fetched from repositories when needed. Dependencies arise when one package needs another for function. Package managers handle installs, updates, and dependencies.

- Common Python package managers: `pip` and `conda`.
- `pip` is bundled with Python, usable through terminal.
- Terminals are text interfaces for typed commands.
- Terminals launch within Jupyter or externally in the OS.

Python; Common DS Packages

Import Name	Common Alias	Description
numpy	np	NumPy includes functions and classes that aid in numerical computation. NumPy is used in many other data science packages.
pandas	pd	pandas provides methods and classes for tabular and time-series data.
sklearn	sk	scikit-learn provides implementations of many machine learning algorithms with a uniform syntax for preprocessing data, specifying models, fitting models with cross-validation, and assessing models.
matplotlib.pyplot	plt	matplotlib allows the creation of data visualizations in Python. The functions mostly expect NumPy arrays.
seaborn	sns	seaborn also allows the creation of data visualizations but works better with pandas DataFrame.
scipy.stats	sp.stats	SciPy provides algorithms and functions for computing problems that arise in science, engineering and statistics. scipy.stats provides the functions for statistics.
statsmodels	sm	statsmodels adds functionality to Python to estimate many different kinds of statistical models, make inferences from those models, and explore data.

Python; NumPy Package

NumPy offers math functions like polynomials, matrices, and stats. It's "num-pie". Scientists use it for 1D or 2D data with **ndarray**. This type is ordered, indexed, and mutable, holding elements of the same type. Array() makes ndarrays.

An array may have zero, one, or many dimensions:

- A zero-dimensional array consists of a scalar object. Ex: 2.
- A one-dimensional array consists of a container of scalars. Ex: [2, 4, 6, 8].
- A two-dimensional array consists of a container of containers of scalars. Ex: [[2, 4, 6, 8], [12, 14, 16, 18]].

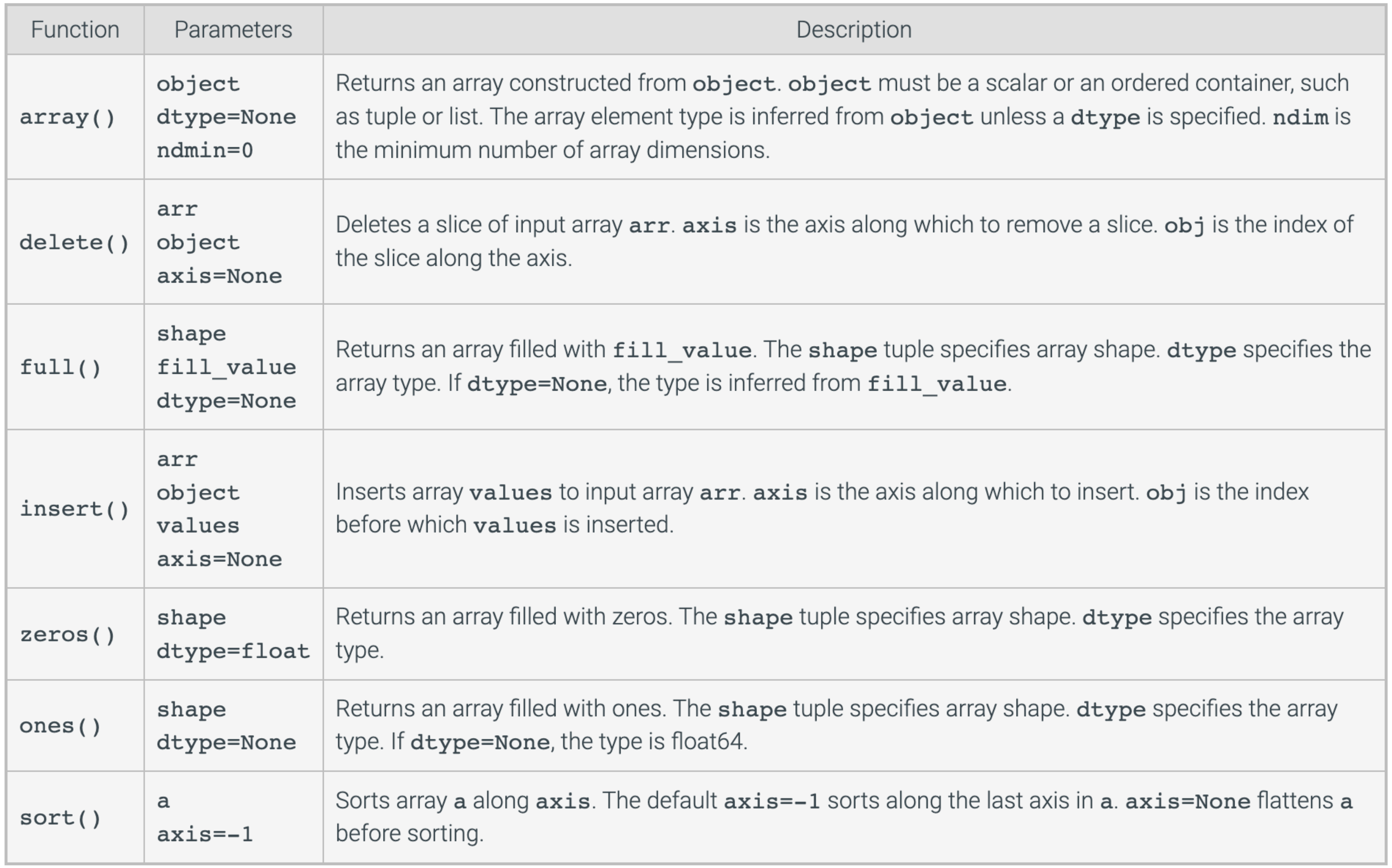
An N-dimensional array has nested containers at N levels, all with the same size. Array **shape** is a tuple of these sizes, like (2, 4) for [[2, 4, 6, 8], [12, 14, 16, 18]]. Access elements with indices like array[2, 5] for 3rd row, 6th column.

Python; NumPy Package; Array Functions

- NumPy functions are written with the prefix 'numpy' or an alias. The tables omit this prefix. Ex: sort(array) stands for numpy.sort(array).
- Many NumPy functions have equivalent ndarray methods. Ex: sort(array) is equivalent to array.sort(). The table includes functions only and omit equivalent methods.
- More functions are in this material and NumPy API reference.

- See the table in the next slide!

Python; NumPy Package; Array Functions



Function	Parameters	Description

`array()`	`object`
	`dtype=None`
	`ndmin=0`
		Returns an array constructed from **`object`**. **`object`** must be a scalar or an ordered container, such as tuple or list. The array element type is inferred from **`object`** unless a **`dtype`** is specified. **`ndim`** is the minimum number of array dimensions.

`delete()`	`arr`
	`object`
	`axis=None`
		Deletes a slice of input array **`arr`**. **`axis`** is the axis along which to remove a slice. **`obj`** is the index of the slice along the axis.

`full()`	`shape`
	`fill_value`
	`dtype=None`
		Returns an array filled with **`fill_value`**. The **`shape`** tuple specifies array shape. **`dtype`** specifies the array type. If **`dtype=None`**, the type is inferred from **`fill_value`**.

`insert()`	`arr`
	`object`
	`values`
	`axis=None`
		Inserts array **`values`** to input array **`arr`**. **`axis`** is the axis along which to insert. **`obj`** is the index before which **`values`** is inserted.

`zeros()`	`shape`
	`dtype=float`
		Returns an array filled with zeros. The **`shape`** tuple specifies array shape. **`dtype`** specifies the array type.

`ones()`	`shape`
	`dtype=None`
		Returns an array filled with ones. The **`shape`** tuple specifies array shape. **`dtype`** specifies the array type. If **`dtype=None`**, the type is float64.

`sort()`	`a`
	`axis=-1`
		Sorts array **`a`** along **`axis`**. The default **`axis=-1`** sorts along the last axis in **`a`**. **`axis=None`** flattens **`a`** before sorting.

Python; NumPy Package; Shape Functions

- Data scientists sometimes need to change array shapes. For example, two-dimensional data from a CSV becomes one-dimensional after reading, requiring reshaping.

- Reshaping alters dimensions while keeping data. [[1, 3, 5], [2, 4, 6]] can become [[1, 2], [3, 4], [5, 6]]. Flattening makes N-dimensional arrays one-dimensional.

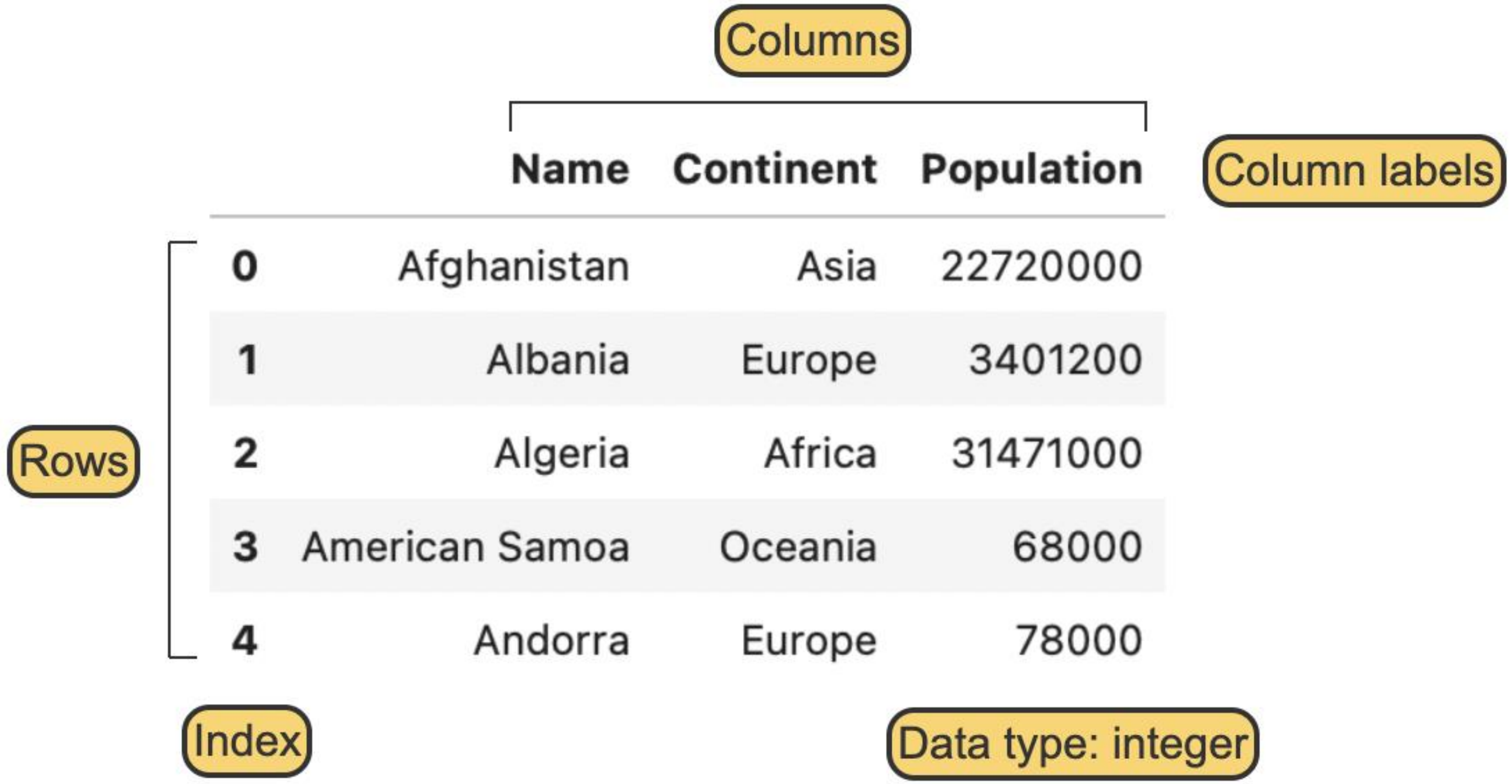
- For 2D arrays, reshaping follows row/column priority. Row-major keeps row order; column-major keeps column order. Flattening [[1, 3, 5], [2, 4, 6]] yields:
  - [1, 3, 5, 2, 4, 6] in row-major order.
  - [1, 2, 3, 4, 5, 6] in column-major order.
- Row and column priority are for 3D+ arrays. Row-major preserves order in 1st, 2nd dimensions, etc. Column-major reverses it.

- Certain shape functions use 'order' parameter. 'C' is row-major, 'F' is column-major. 'C' and 'F' come from C and FORTRAN, storing in row or column order.

Python; panadas Package

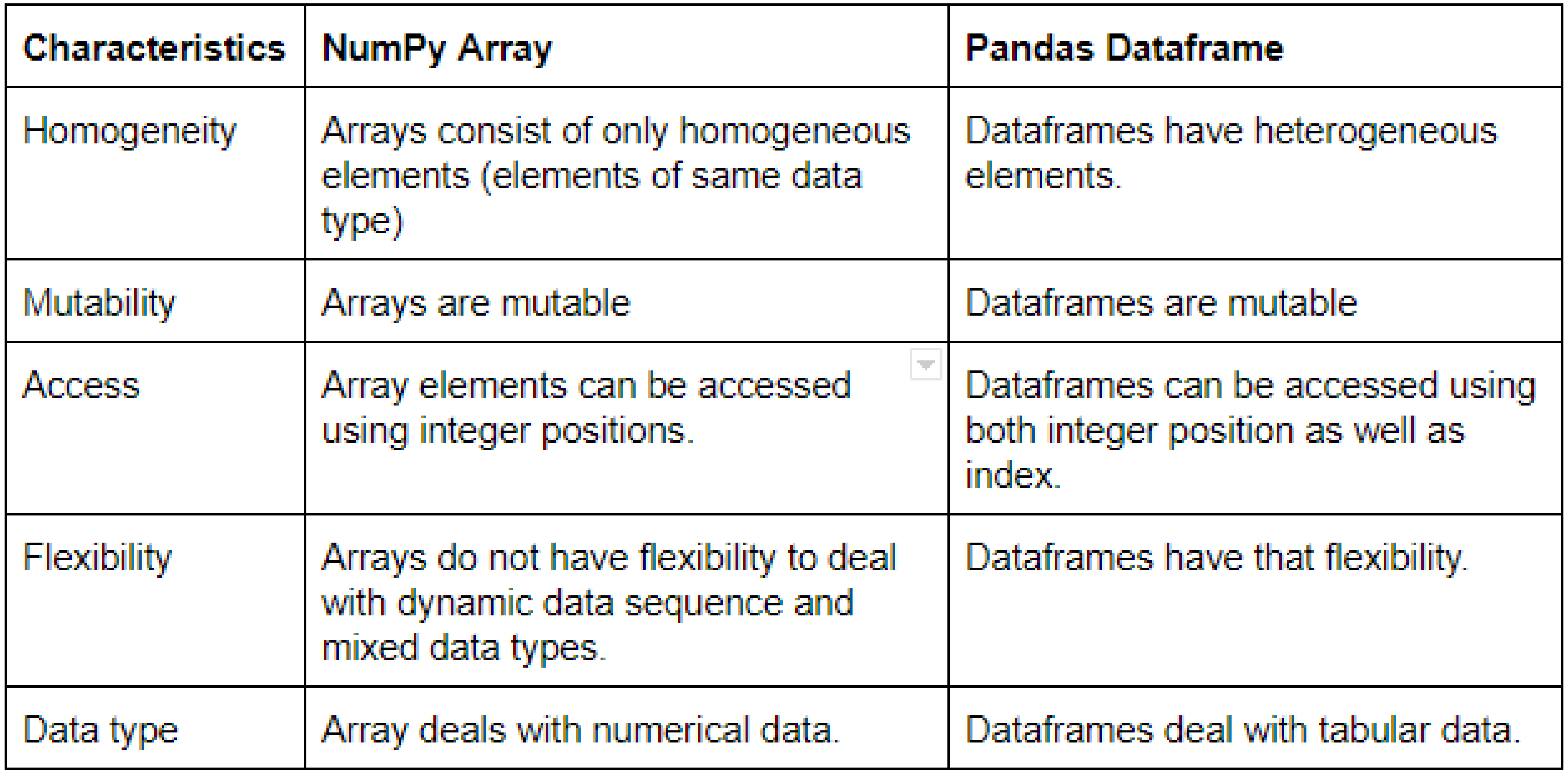
- **pandas** is a Python package for dataset storage and manipulation, using **DataFrames**.
- DataFrames contain rows (instances) and columns (features), labeled with integers or strings.
- Index is row labels, columns are column labels.
- Types in a column match, but columns can differ. Common types are int, float, string.

Python; panadas Package; DataFrame



	Name	Continent	Population
0	Afghanistan	Asia	22720000
1	Albania	Europe	3401200
2	Algeria	Africa	31471000
3	American Samoa	Oceania	68000
4	Andorra	Europe	78000

Python; panadas vs NumPy



Characteristics	NumPy Array	Pandas Dataframe
Homogeneity	Arrays consist of only homogeneous elements (elements of same data type)	Dataframes have heterogeneous elements.
Mutability	Arrays are mutable	Dataframes are mutable
Access	Array elements can be accessed using integer positions.	Dataframes can be accessed using both integer position as well as index.
Flexibility	Arrays do not have flexibility to deal with dynamic data sequence and mixed data types.	Dataframes have that flexibility.
Data type	Array deals with numerical data.	Dataframes deal with tabular data.

Python; panadas Package; Subsetting Data

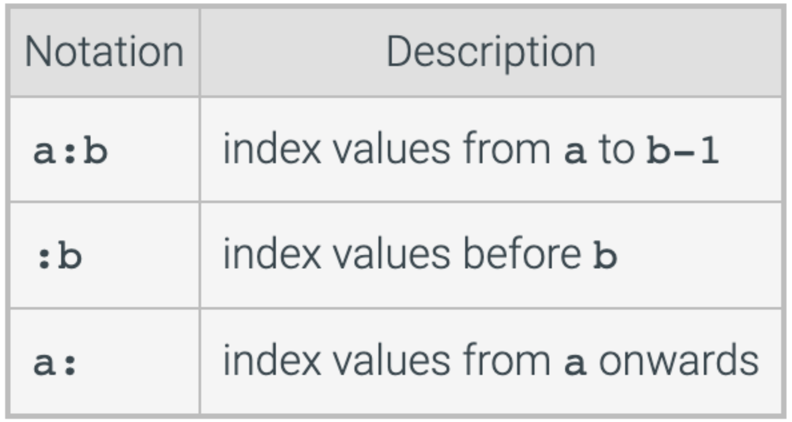
- Data **subsetting** picks rows and columns from dataframes with labels, indices, and slices.

- To pick a column, use its label like country['Population'] or country.Population. For multiple columns, use an array like country[['Name', 'Population']].

- Using iloc(x,y), get an element at index x, y. For slices, use ":". Ex: country.iloc[:5,1:3] gets rows before 5 and columns 1 to 3.

- loc(x,y) also subsets, but y is an array of labels. Ex: country.iloc[:7,1:3] and country.loc[:6,['Continent','Population']] are the same.

- For the loc() method, the index b is included in the resulting output.



Notation	Description
`a:b`	index values from `a` to `b-1`
`:b`	index values before `b`
`a:`	index values from `a` onwards

Python; panadas Package; DataFrame Methods

Method	Parameters	Description

`drop()`	`labels=None`
	`axis=0`
	`inplace=False`
		Removes rows (`axis=0`) or columns (`axis=1`) from `dataframe`. `labels` specifies the labels of rows or columns to drop.

`drop_duplicates()`	`subset=None`
	`inplace=False`
		Removes duplicate rows from `dataframe`. `subset` specifies the labels of columns used to identify duplicates. If `subset=None`, all columns are used.

`dropna()`	`axis=0`
	`how='any'`
	`subset=None`
	`inplace=False`
		Removes rows (`axis=0`) or columns (`axis=1`) containing missing values from `dataframe`. `subset` specifies labels on the opposite axis to consider for missing values. `how` indicates whether to drop the row or column if `any` or if `all` values are missing.

`insert()`	`loc`
	`column`
	`value`
		Inserts a column to `dataframe`. `loc` specifies the integer position of the new column. `column` specifies a string or numeric column label. `value` specifies column values as a Scalar or Series.

`replace()`	`to_replace=None`
	`value=`
	`NoDefault.no_default`
	`inplace=False`
		Replaces `to_replace` values in `dataframe` with `value`. `to_replace` and `value` may be string, dictionary, list, regular expressions, or other data types.

`sort_values()`	`by`
	`axis=0`
	`ascending=True`
	`inplace=False`
		Sorts `dataframe` columns or rows. `by` specifies indexes or labels on which to sort. `axis` specifies whether to sort rows (0) or columns (1). `ascending` specifies whether to sort ascending or descending. `inplace` specifies whether to sort `dataframe` or return a new dataframe.

Python; matplotlib Package

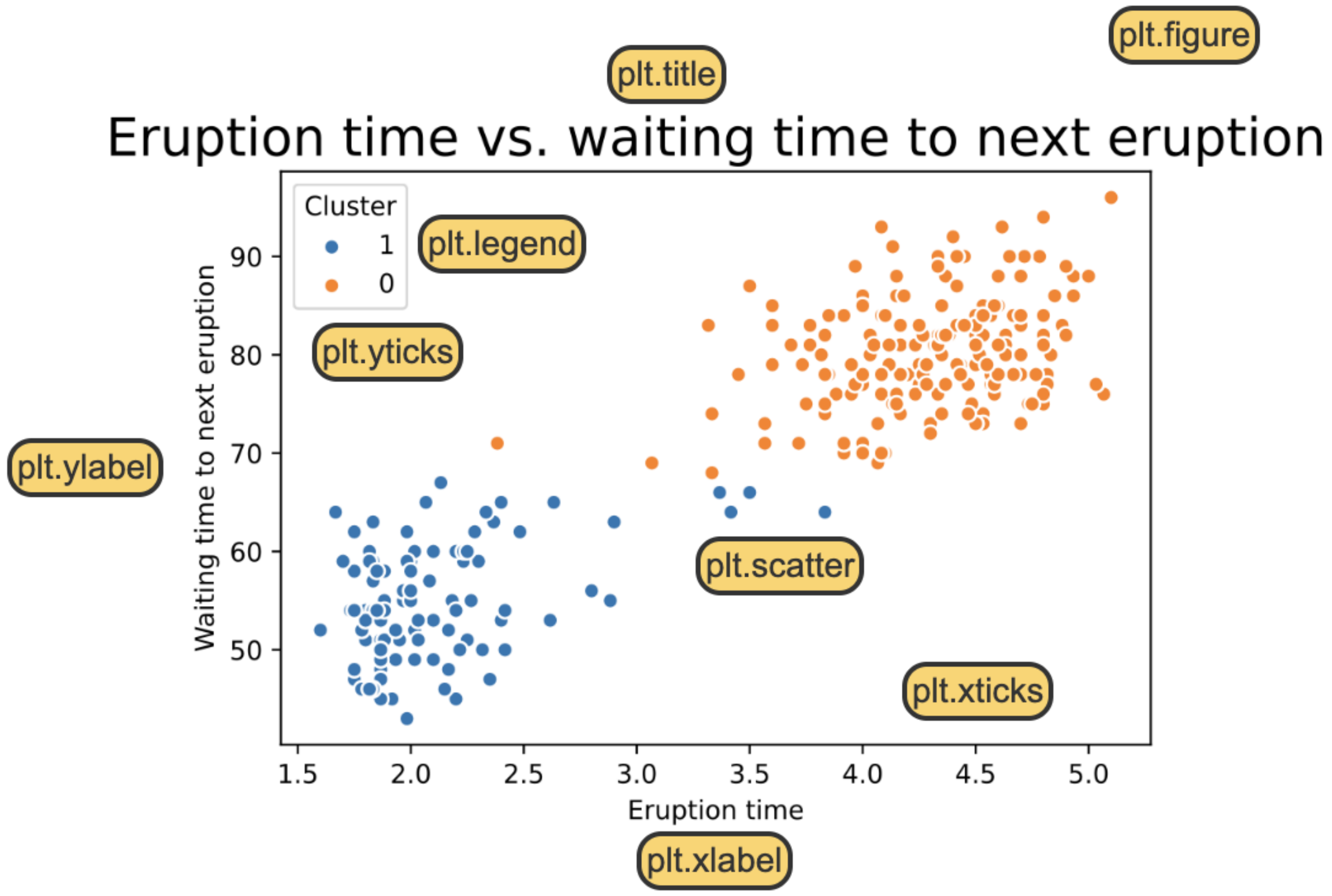
- To plot using pyplot, start by importing the library: import matplotlib.pyplot as plt.

- Pyplot displays various objects in a figure. Common ones include plt.plot(x, y) for line plots (x and y are arrays of the same size), and plt.scatter(x, y) for scatter plots of x-y pairs.

- Other useful functions in the pyplot library are:

  - plt.figure(): creates a new figure.
  - plt.show(): displays the figure and all the objects the figure contains.
  - plt.savefig(fname): saves the figure in the current working directory with the filename fname.

Python; matplotlib Package



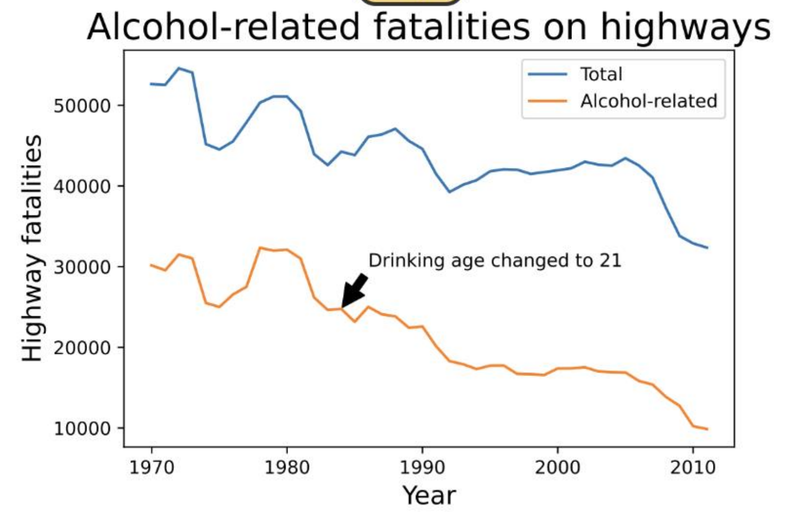
Python; matplotlib Package; Labeling Plots

- The plt.plot() functions display line plots: The line plot on top displays total highway fatalities, and the line plot on the bottom displays alcohol-related fatalities.
- The plt.title(), plt.xlabel(), and plt.ylabel() functions add text for the axes.
- The plt.annotate() function adds text linked to specific coordinates within the figure. Here, the text is located at (1986, 30000) and the arrow points to (1984, 24762).
- The plt.legend() function adds a legend within the figure. matplotlib chooses the best location for the legend, but the location can be specified manually.

Python code

```python
df = pd.read_csv('fatalities.csv')
plt.plot(df['Year'], df['Total'], label='Total')
plt.plot(df['Year'], df['Alcohol-related'],
         label='Alcohol-related')
plt.title('Alcohol-related fatalities on highways', fontsize=20)
plt.xlabel('Year', fontsize=14)
plt.ylabel('Highway fatalities', fontsize=14)
plt.annotate('Drinking age changed to 21',
             arrowprops=dict(facecolor='black', shrink=0.05),
             xy=(1984, 24762),
             xytext=(1986, 30000))
plt.legend()
```

Output



Lecture 2

Programming with R

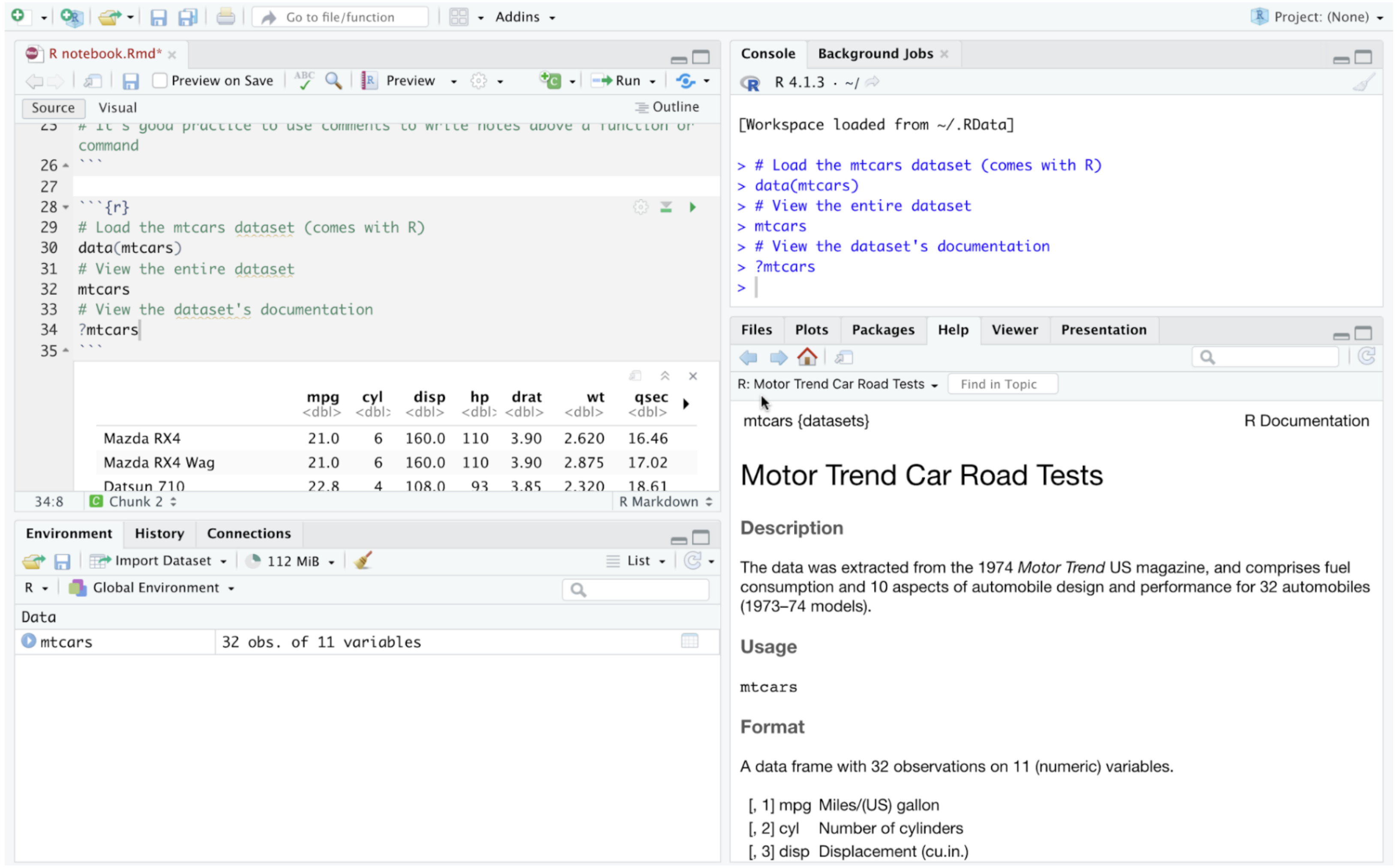
R

- Computer program: Lines of code executed sequentially
- Written in specific language
- R: Programming language for statistics
- Supports statistical methods creation & application
- Useful in data science's statistical analysis
- Users apply existing methods
- Programmers create new methods
- Shared code for insights from data

R for Data Science

- R simplifies data tasks.
- code readability/modification is easy in R.
- R uses "#" symbol for comments.
- R alone sufficient for data science code; other programs aid sharing/reproducibility.
- IDE like RStudio for writing/testing code in one interface, and it enhances data science workflow.

RStudio



```
# It's good practice to use comments to write notes above a function or
command
```

```{r}
# Load the mtcars dataset (comes with R)
data(mtcars)
# View the entire dataset
mtcars
# View the dataset's documentation
?mtcars
```

```
                 mpg   cyl  disp   hp    drat   wt     qsec
                 <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>
Mazda RX4        21.0   6    160.0  110  3.90   2.620  16.46
Mazda RX4 Wag    21.0   6    160.0  110  3.90   2.875  17.02
Datsun 710       22.8   4    108.0   93  3.85   2.320  18.61
```

```
[Workspace loaded from ~/.RData]

> # Load the mtcars dataset (comes with R)
> data(mtcars)
> # View the entire dataset
> mtcars
> # View the dataset's documentation
> ?mtcars
>
```

Data

```
mtcars    32 obs. of 11 variables
```

mtcars {datasets} R Documentation

Motor Trend Car Road Tests

Description

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Usage

```
mtcars
```

Format

A data frame with 32 observations on 11 (numeric) variables.

[, 1]	mpg	Miles/(US) gallon
[, 2]	cyl	Number of cylinders
[, 3]	disp	Displacement (cu.in.)

https://posit.co/products/open-source/rstudio/

R; Variables and Expressions

- **Variables** are created in R by performing an assignment using the `<-` symbol. Ex: `predicted_temp <- 90`.

- **Expressions** are lines of code that return a specific value when evaluated. Expressions usually describe simple calculations. Ex: `temp_error <- actual_temp - predicted_temp`.

R; Functions

Most data scientists in R use pre-defined functions for modeling and visualization, as functions are commands for computations.

- **Parameters**: Function inputs like dataset, feature, option
- **Arguments**: Values for function parameters (e.g., `mean(x=mtcars)`) where `x` is the parameter and `mtcars` is the argument.
- **Returns**: Function outputs, assignable (numbers/tables) or plots, not always auto-displayed

R; Vectors and Types

- Vector: holds elements of possibly different types. Ex: `x <- 5` forms a vector "x" with 1 element, 5.
- The expression `letters <- c("A", "B", "C")` uses `c()` to create a vector named `letters` with three elements: A, B, and C.
- Vector elements accessed with brackets. Ex: `letters[1]` from vector "`letters`" gets "A" (first element).
- For multiple: `letters[2:3]` gives "B" and "C", `letters[c(1,3)]` offers "A" and "C".
- Vector elements' types inform their behavior in R.
- Error in `x+y` (`x <- 5`, `y <- "five"`) due to differing types.
- Four key types: characters, doubles, integers, logicals.
- Integers/doubles might be 'numeric’, and characters termed 'strings'.

R; Atomic Vectors vs. Lists

- Datasets frequently combine different data types. For instance, a penguin dataset may include the "species" feature as character vectors and "bill_length_mm" as doubles.

- Atomic vectors maintain uniform element types. All vectors shown earlier are atomic since they contain elements of a single type.

- Lists or generic vectors allow diverse element types. For instance, creating a list like "firstList" with character and integer vectors:
  - firstList <- list(c("A", "B", "C"), c(1L, 2L))

R; Atomic Vectors vs. Lists

Characteristic	Atomic vector	List
Possible element types	Element must be a base type. Ex: integer, double, logical, character are all base types.	More elements are possible. Ex: base types, atomic vectors, lists
Can elements have different types?	No	Yes
Element names?	Optional, not as common	Optional, commonly used
Accessing elements	Square brackets Ex: `myVector[1]`	Double brackets or $ Ex: `myList[[1]]` or `myList$elementName`
Function to create	`c()`	`list()`

R; Missing Values

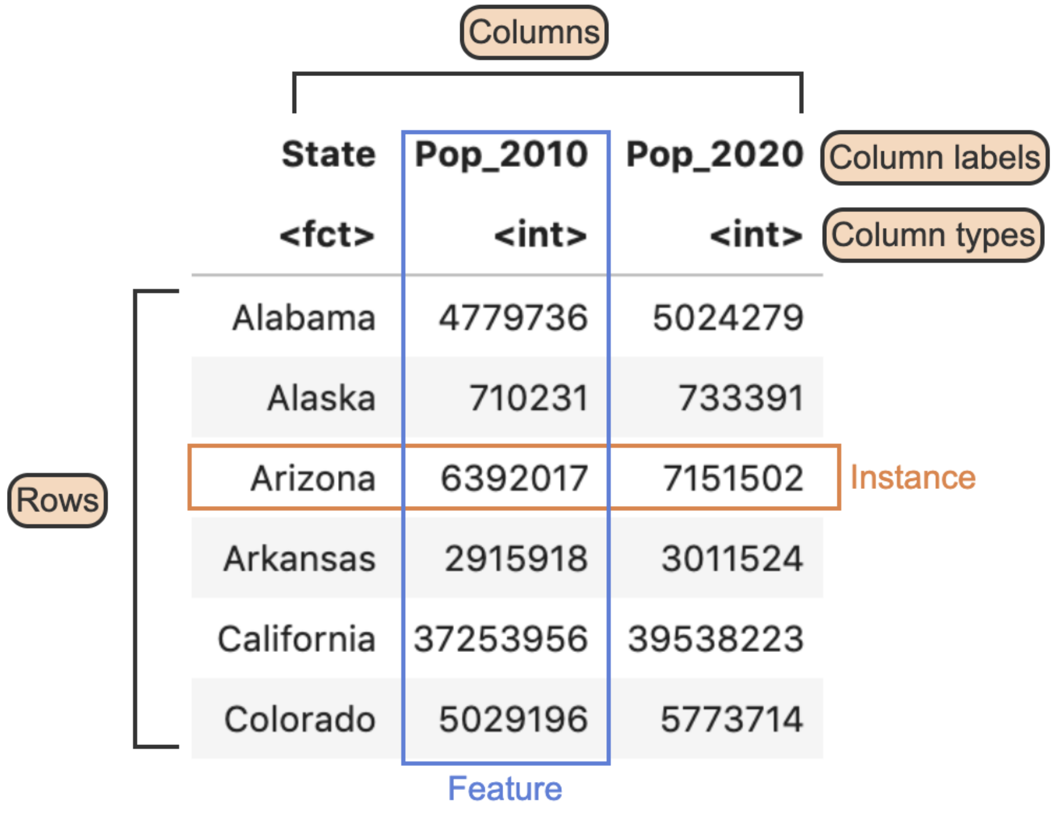
- Missing data is common, like an unanswered survey question.
- In R, NA denotes missing data (e.g., `responses <- c(1, 2, NA)`).
- R functions assume handling of missing data (e.g., `mean()` returns NA by default with NA values). The function `mean()` will return NA because the default argument of the `na.omit` parameter is `FALSE`.

R; Anticipating R’s Behavior

- Predicting R function's return is vital for accurate code and problem-solving (e.g., `c(1, 2, 3) + 1` outcome).
- Function is **vectorized** when applied to all vector elements (e.g., `+` operation).
- R's **coercion** alters input type for successful execution (e.g., `toupper(10)` turns integer to "10").
- Coercion varies among functions, reading warnings ensures intended execution.

R; Dataframes

- In R, datasets are usually **dataframes**, resembling spreadsheets.
- Dataframes have rows (instances) and columns (features).
- Rows are labeled with integers from 1, columns with specified strings. Each column has a consistent type (e.g., double, character, integer).



State	Pop_2010	Pop_2020
<fct>	<int>	<int>
Alabama	4779736	5024279
Alaska	710231	733391
Arizona	6392017	7151502
Arkansas	2915918	3011524
California	37253956	39538223
Colorado	5029196	5773714

R; Dataframes

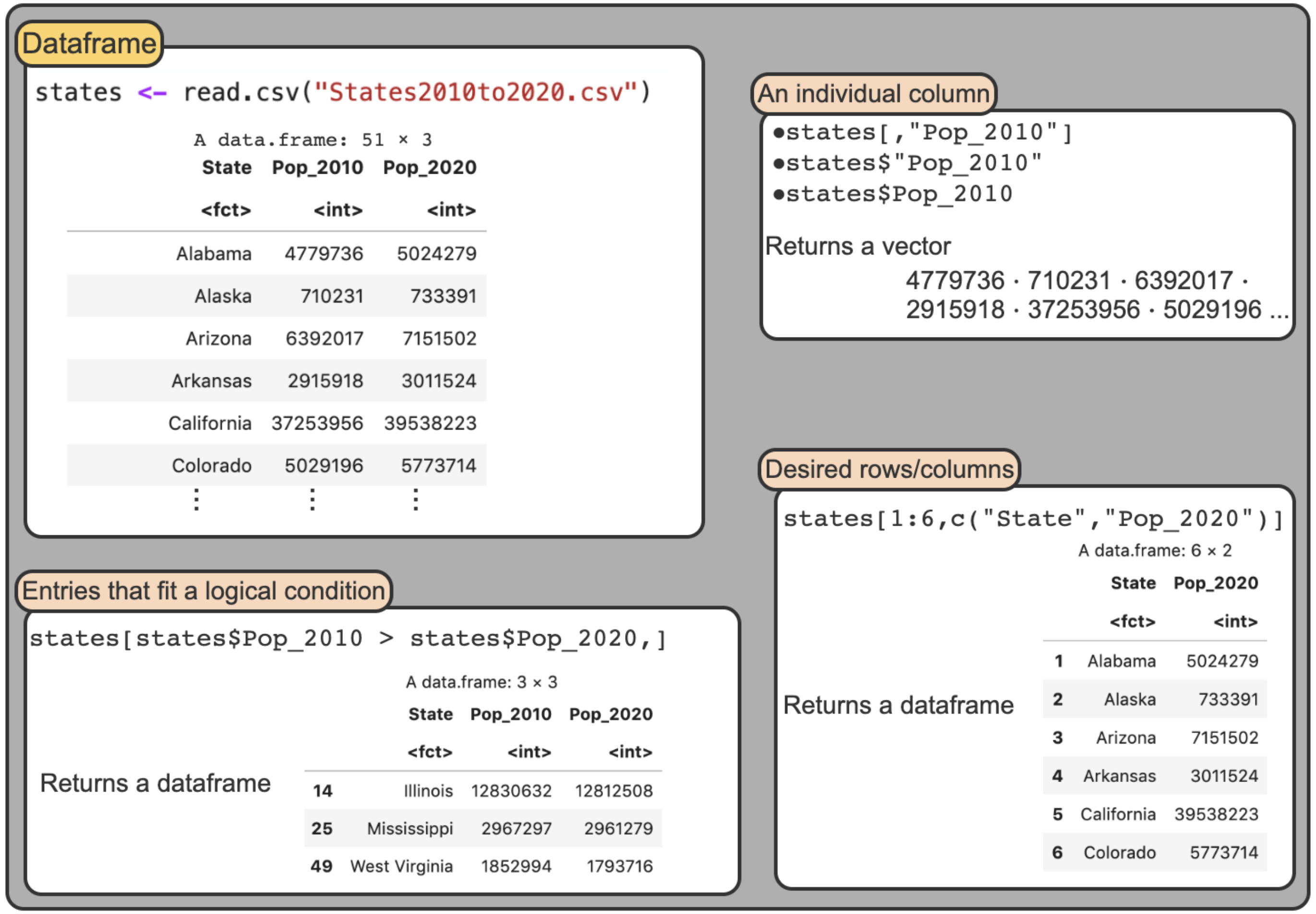
- To create a dataframe manually, use `data.frame()` with feature vectors as inputs. See documentation for full parameter list.

- Import previously collected data from .csv files in the working directory using `read.csv()`. By default, spaces in column names become dots. Preserve spaces with `check.names=FALSE`.

- The full list of parameters and information about other file types that R can load is found in the official documentation.

R; Dataframes; Example

- Import .csv file for 51 x 3 states dataframe.
- Use brackets/dollar sign for column vector.
- Select rows/columns for smaller dataframe, keeping original indices.
- Apply logical condition for smaller dataframe with original indices.



Dataframe

```
states <- read.csv("States2010to2020.csv")
```

A data.frame: 51 × 3

State	Pop_2010	Pop_2020
<fct>	<int>	<int>
Alabama	4779736	5024279
Alaska	710231	733391
Arizona	6392017	7151502
Arkansas	2915918	3011524
California	37253956	39538223
Colorado	5029196	5773714
⋮	⋮	⋮

An individual column

- `states[,"Pop_2010"]`
- `states$"Pop_2010"`
- `states$Pop_2010`

Returns a vector

4779736 · 710231 · 6392017 · 2915918 · 37253956 · 5029196 ...

Entries that fit a logical condition

```
states[states$Pop_2010 > states$Pop_2020,]
```

Returns a dataframe

A data.frame: 3 × 3

	State	Pop_2010	Pop_2020
	<fct>	<int>	<int>
14	Illinois	12830632	12812508
25	Mississippi	2967297	2961279
49	West Virginia	1852994	1793716

Desired rows/columns

```
states[1:6,c("State","Pop_2020")]
```

Returns a dataframe

A data.frame: 6 × 2

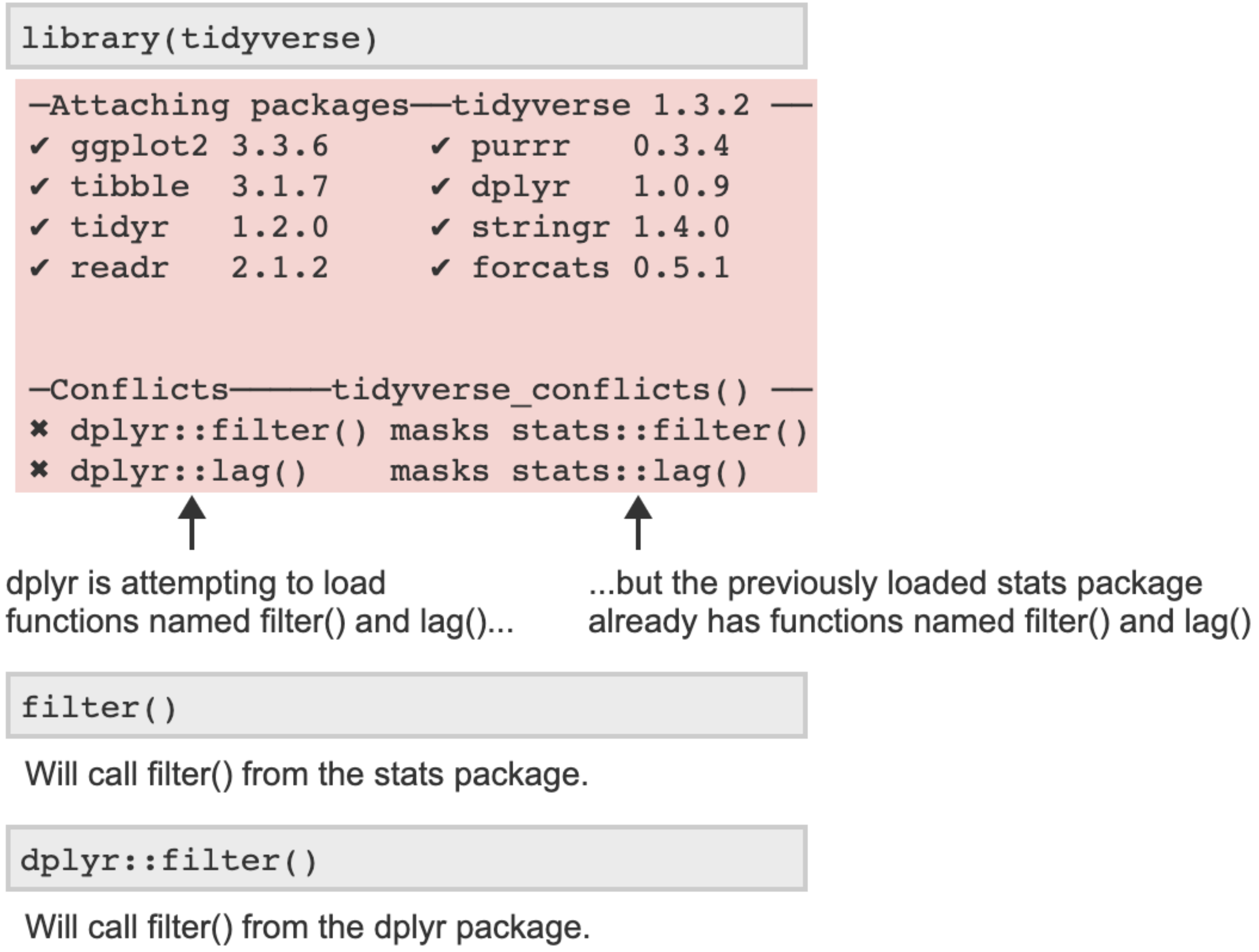
	State	Pop_2020
	<fct>	<int>
1	Alabama	5024279
2	Alaska	733391
3	Arizona	7151502
4	Arkansas	3011524
5	California	39538223
6	Colorado	5773714

R; Packages

- A **package** in R extends functionality with functions, code, data, and docs. Install and load before use.
- Use install.packages() to install locally; library() to load.
- The stats, graphics, grDevices, utils, datasets, methods, and base packages come installed with R and are loaded by default in every session.

R; Package Conflicts

- Package conflict emerges when two packages share an object name. On using library(), conflicts generate a console warning. Priority goes to the first loaded package. Resolve by employing the double-colon operator (::) to access specific objects, especially for conflicted function calls.



```
library(tidyverse)
```

```
—Attaching packages———tidyverse 1.3.2 ——
✔ ggplot2 3.3.6      ✔ purrr   0.3.4
✔ tibble  3.1.7      ✔ dplyr   1.0.9
✔ tidyr   1.2.0      ✔ stringr 1.4.0
✔ readr   2.1.2      ✔ forcats 0.5.1


—Conflicts———————tidyverse_conflicts() ——
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
```

dplyr is attempting to load functions named filter() and lag()...

...but the previously loaded stats package already has functions named filter() and lag()

```
filter()
```

Will call filter() from the stats package.

```
dplyr::filter()
```

Will call filter() from the dplyr package.

R; Packages

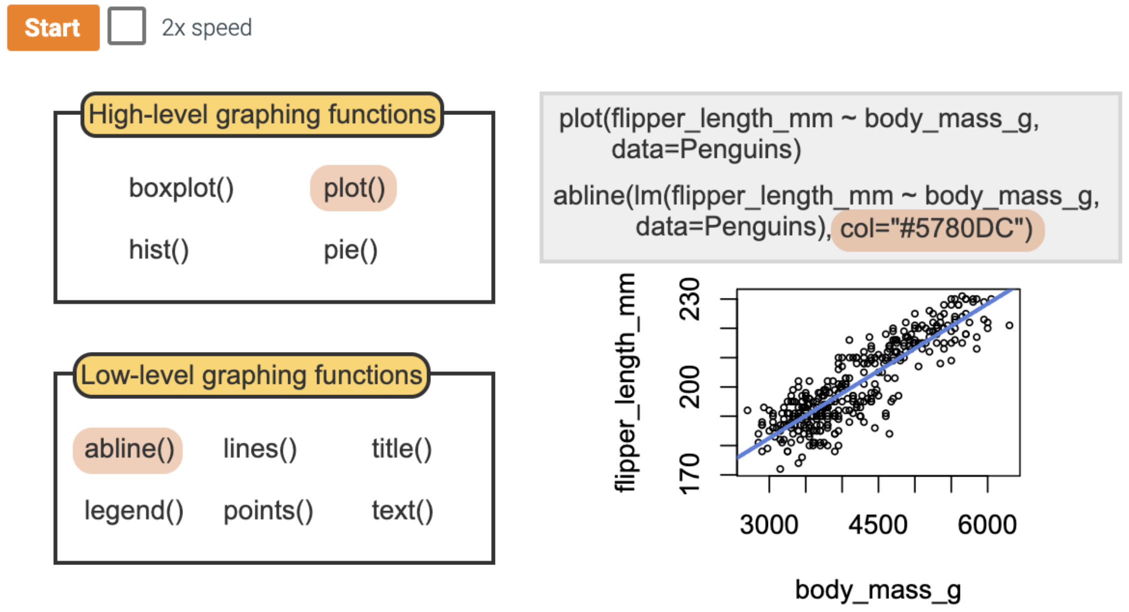
- **Documentation** serves to inform about object details, encompassing parameters, properties, and usage. It can be quickly accessed using "?" or online sources (see more here).

- Documentation often contains examples for individual functions within a package. However, it may not cover how these functions work together in a broader context. This is where **vignettes** come in. Vignettes are comprehensive tutorials showcasing package workflows with varying levels of detail.

- For instance, "Welcome to the Tidyverse" vignette explains the relationships between tidyverse packages. On the other hand, "Aesthetic Specifications" delves into ggplot2 settings. Both documentation and vignettes can be accessed locally or online. IDEs like RStudio offer the browseVignettes() function to conveniently explore package vignettes.

R; Base Graphics

1. High-level functions: boxplot(), hist(), plot(), pie(). Initial graph creation.
2. For instance, plot() generates a scatter plot of flipper lengths vs. body masses in penguins.
3. Low-level functions enhance high-level graphs with lines, points, or text. Example: abline() adds a linear regression line to a scatter plot.
4. Graphs start with defaults but can be customized. E.g., alter abline()'s default black line color using the col parameter.



```
plot(flipper_length_mm ~ body_mass_g,
     data=Penguins)
abline(lm(flipper_length_mm ~ body_mass_g,
     data=Penguins), col="#5780DC")
```

R; the plot() Function

- plot(x, y) is a versatile high-level function creating diverse graph types. It adapts based on x and optional y parameters' classes. For instance, for factor class like city in the trains dataset, plot(x=trains$city) generates a bar plot using plot.factor().

- plot() commonly graphs two features, using different accepted methods. The following two are frequently utilized.
  - Specifying the x and y parameters individually.

  Ex: plot(x=trains$distance, y=trains$fare).
  - Specifying a formula of the form y ~ x.

  Ex: plot(trains$fare ~ trains$distance) or plot(fare ~ distance, data=trains).

- Additional information about the plot() function can be found in the plot documentation.

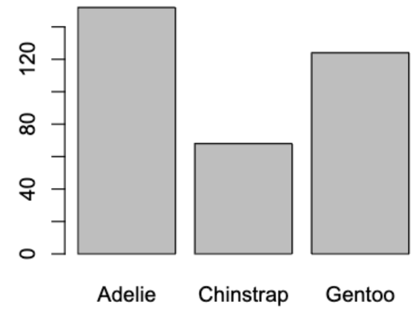
R; the plot() Function

Graph type	Function	Description
Bar plot	`barplot(height, horiz=FALSE, beside=FALSE)`	Creates a bar plot with vertical (`horiz=FALSE`) or horizontal (`horiz=TRUE`) bars. `height` is a vector or matrix of values describing the bars' heights. If `height` is a matrix and `beside=FALSE`, each column corresponds to heights of stacked bars.
Box plot	`boxplot(x, horizontal=FALSE)` `boxplot(formula, data, horizontal=FALSE)`	Creates a box plot of the numeric vector `x` or box plots defined by the formula `y ~ grp` where `y` is a numeric vector of values to be split into groups according to `grp`.
Histogram	`hist(x, breaks, freq)`	Creates a histogram of the numeric vector `x` where the y-axis represents either frequencies (`freq=TRUE`) or probability densities (`freq=FALSE`). Bin breakpoints can be adjusted through the `breaks` parameter.

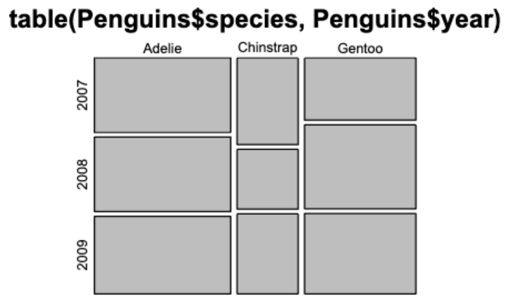
For additional info see the documentation

plot(x, y)

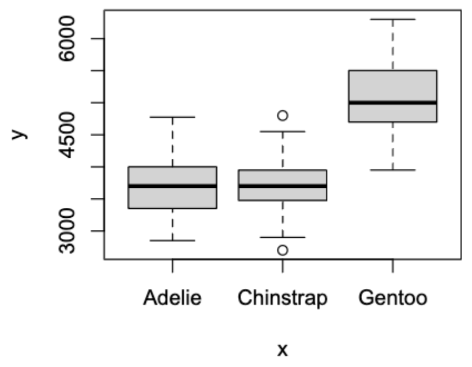
plot(x=Penguins$species)



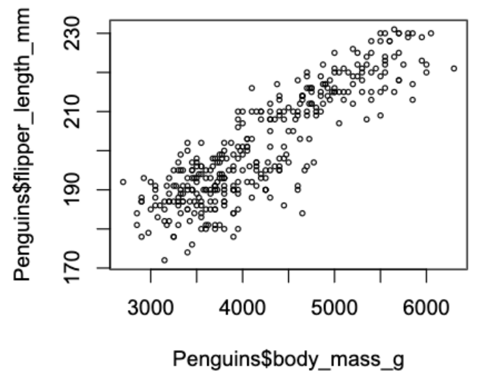
plot(table(Penguins$species, Penguins$year))



plot(x=Penguins$species, y=Penguins$body_mass_g)



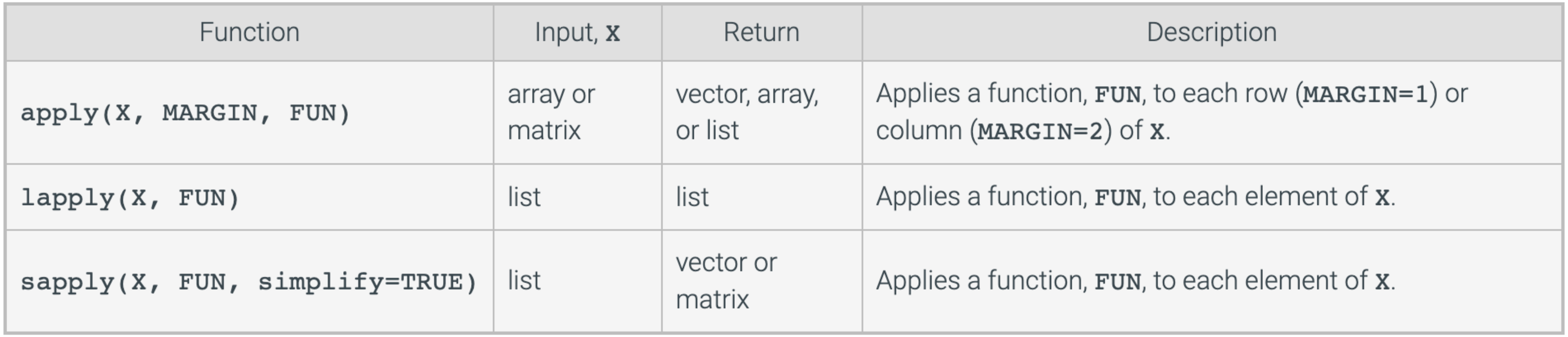
plot(x=Penguins$body_mass_g, y=Penguins$flipper_length_mm)
plot(Penguins$flipper_length_mm ~ Penguins$body_mass_g)



R; The apply(), lapply(), and sapply() Functions

- Data science in R often involves repetitive operations on dataset features.
- The apply function family in the base package addresses this by systematically applying functions to structured data, benefiting from R's vectorization.
- Using apply(), like for computing column means, streamlines the process by looping through columns, converting to vectors, and applying the function, leading to efficient and concise code compared to manual loops.
- Choose apply function based on object type and desired return. Common ones: apply(), lapply(), sapply().

R; The apply(), lapply(), and sapply() Functions

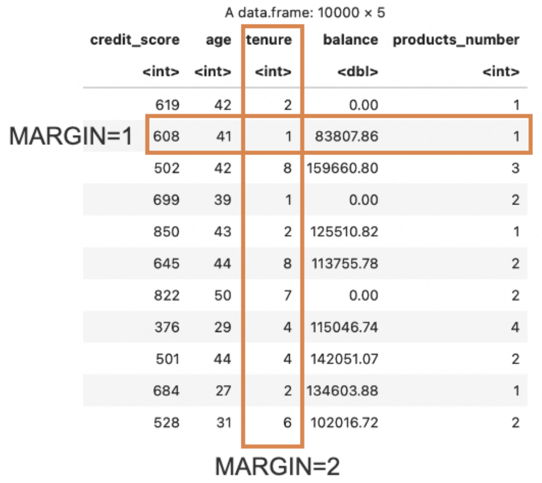


Function	Input, `X`	Return	Description
`apply(X, MARGIN, FUN)`	array or matrix	vector, array, or list	Applies a function, `FUN`, to each row (`MARGIN=1`) or column (`MARGIN=2`) of `X`.
`lapply(X, FUN)`	list	list	Applies a function, `FUN`, to each element of `X`.
`sapply(X, FUN, simplify=TRUE)`	list	vector or matrix	Applies a function, `FUN`, to each element of `X`.

Additional details can be found in the apply() and lapply() documentation. Information about the other apply functions available can be found in the base documentation.

R; The apply() Function

```
cols=c("credit_score", "age", "tenure",
           "balance", "products_number")
bank[ ,cols]
```


MARGIN=1
MARGIN=2

A data.frame: 10000 × 5

credit_score	age	tenure	balance	products_number
<int>	<int>	<int>	<dbl>	<int>
619	42	2	0.00	1
608	41	1	83807.86	1
502	42	8	159660.80	3
699	39	1	0.00	2
850	43	2	125510.82	1
645	44	8	113755.78	2
822	50	7	0.00	2
376	29	4	115046.74	4
501	44	4	142051.07	2
684	27	2	134603.88	1
528	31	6	102016.72	2

Manual loop

```
means<-numeric(length(cols))
for(i in 1:length(cols))
  means[i]<-mean(bank[ ,cols[i]])
means
```

```
650.5288   38.9218   5.0128 76485.8893   1.5302
```

Apply

```
means<-apply(X=bank[ ,cols], MARGIN=2, FUN=mean)
means
```

```
credit_score        age      tenure        balance   products_number
    650.5288    38.9218      5.0128     76485.8893            1.5302
```

```
apply(X=bank[,cols] , MARGIN=1, FUN=mean)
apply(X=bank[,cols] , MARGIN=2, FUN=mean, na.rm=TRUE)
```

1. European bank sampled 10,000 customers, analyzing 5 columns numerically.
2. Manual loop finds column means using for() function.
3. apply() simplifies code, more efficient. MARGIN=2 extracts columns, FUN=mean.
4. apply() with MARGIN=1 computes row means. Additional mean() arguments like na.rm=TRUE applicable.

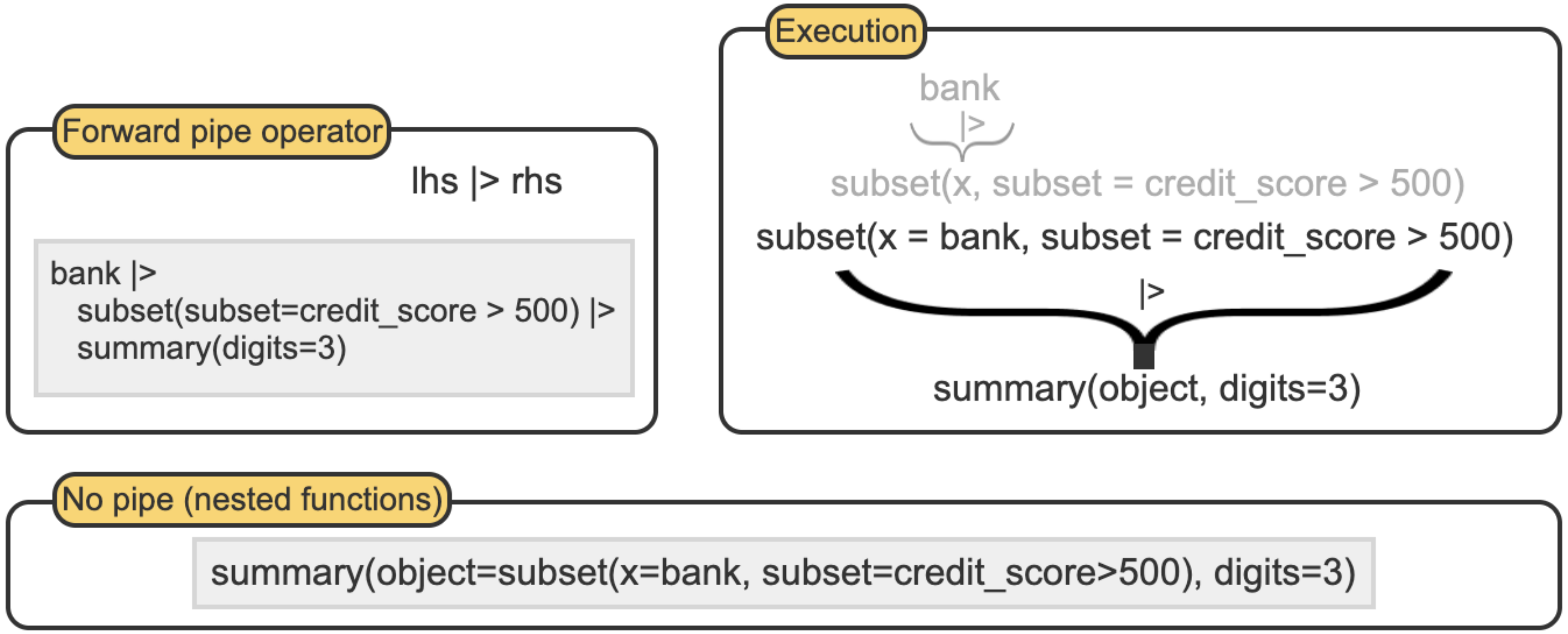
R; Writing Functions

- User-defined functions in R are created by users, not pre-defined. They streamline tasks, making code concise and clear. To write one, three components are needed:
    - the function name,
    - the parameters or inputs, and
    - the code to be executed including the return or output.

- The function `function()` is used to create user-defined functions in R. Additional information about the use of `function()` from the `base package` can be found in the function documentation.

R; The Base R Pipe Operator

- The R pipe operator, `|>`, forwards lhs into rhs, like `x |> mean()`.
- For instance, `x |> mean()` executes `mean(x)`.
- rhs must be a **function** call, not just a name like `x |> mean`.
- The pipe operator enhances code clarity, efficiency, replacing nested parentheses, and storing steps as objects.


Forward pipe operator
lhs |> rhs
Execution
bank
|>
subset(x, subset = credit_score > 500)
|>
No pipe (nested functions)

```
bank |>
  subset(subset=credit_score > 500) |>
  summary(digits=3)
```

```
subset(x = bank, subset = credit_score > 500)
summary(object, digits=3)
```

```
summary(object=subset(x=bank, subset=credit_score>500), digits=3)
```

Case Study

Hawks

Next Lecture

Probability and Statisitcs