جامعة بيرزيت
BIRZEIT UNIVERSITY

Data Science and Analytics
Comp 4381

Ch2: Introduction to Python Part 2

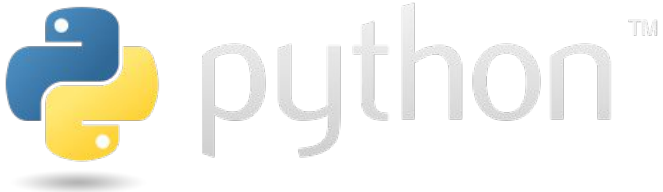
References

- **Books**:
  - Python for Data Analysis 3rd edition - Wes McKinney – O'RIELLY (Ch 2-10)
  - Python data science handbook 2nd edition - Jake VanderPlas – O'RIELLY (Ch 37-40)
  - Statistics unplugged 4th edition – Sally Cardwell - Wadsworth: (Ch 1, 2)
- **Material & Notebooks**:
  - Mr. Hussein Soboh.
- **Additional Resources:**
  - Computational and Inferential Thinking: The Foundations of Data Science 2nd Edition by Ani Adhikari, John DeNero, David Wagner. Link
  - https://www.w3schools.com/python

Introduction to Python 2

Content

- Data structures
  - Tuples
  - Dictionary
  - Pandas
- Logic and control flow
- Loops
- Matplotlib



Tuples

Tuples

- A tuple is a collection which is ordered and **unchangeable**.
- **Unchangeable** (immutable)
  - Meaning that once created, the elements within a tuple **cannot be changed, added, or removed**

Key characteristics of tuples:

- **Immutable**: Unlike lists, tuples cannot be modified after their creation.
  - This makes tuples useful for storing data that should remain constant throughout a program.
- **Ordered**: Tuples maintain the order of elements.
  - This means you can access items based on their position.
- **Allow duplicate elements**: Like lists, tuples can contain elements that appear more than once.
- **Heterogeneous:** A tuple can contain elements of different data types (e.g., strings, integers, floats).

When to use tuples

- **Immutable data:** Tuples are useful when you need to store data that should not change.
  - Which help protect the integrity of data and ensuring that values are not accidentally modified.
- **Faster performance:** Tuples are generally faster than lists due to their immutability.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Create tuple

- Tuples are created by placing items inside parentheses **()** and separating them with commas
- Tuples can also be created **without parentheses** (using just commas)
- It is also possible to use the **tuple()** constructor to make a tuple

```
# Creating a tuple with parantheses
my_tuple = (1, 2, "apple", 4.5)

# Creating a tuple without parantheses
another_tuple = 1, 2, "apple", 4.5

# Creating a tuple using constructor
cons_tuple=tuple((1, 2, "apple", 4.5))
```

```
print(my_tuple)
print(another_tuple)
print(cons_tuple)
```

```
(1, 2, 'apple', 4.5)
(1, 2, 'apple', 4.5)
(1, 2, 'apple', 4.5)
```

Accessing tuple elements

- You can access tuple elements using indexing, just like with lists. The index starts at **0**

Using Index

```
fruits = ("apple", "banana", "cherry")

# Access the first item
print(fruits[0])  # Output: apple

# Access the last item
print(fruits[-1])  # Output: cherry
```

```
apple
cherry
```

slice a tuple

```
my_tuple = (0, 1, 2, 3, 4)
slice = my_tuple[1:4]
slice
```

```
(1, 2, 3)
```

Unpacking a Tuple

- When we create a tuple, we normally assign values to it. This is called "**packing**" a tuple
- We are also allowed to extract the values back into variables. This is called "**unpacking**"

```
fruits = ("apple", "banana", "cherry")
```

packing

```
(green, yellow, red) = fruits
```

unpacking

```
print(green)
print(yellow)
print(red)
```

```
apple
banana
cherry
```

Note: The number of variables must match the number of values in the tuple, if not, you must use an **asterisk(*)** to collect the remaining values as a list

Tuple Methods

- Python has two built-in methods that you can use on tuples

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Dictionary

Dictionary

- A **dictionary** is a data structure that stores data in **key-value pairs**
- Dictionaries are similar to real-life dictionaries where you look up a word **(key)** to find its definition **(value)**
- In Python, they are extremely useful for storing and accessing data quickly

Key features of dictionaries:

- **Unordered**: The items are not stored in a specific order; they are accessed using keys.
- **Mutable**: Dictionaries can be changed after creation—keys and values can be added, modified, or removed.
- **Unique keys**: Keys must be **unique** within a dictionary; no duplicates are allowed

Create Dictionary

- You can create a dictionary using curly braces {} with key-value pairs separated by commas:

```
# Create dictionary
population = {
    'Jerusalem': 428304,
    'Gaza': 410000,
    'Khan Yunis': 173183,
    'Jabalya': 168568,
    'Hebron':160470
}

print(population)
```

```
{'Jerusalem': 428304, 'Gaza': 410000, 'Khan Yunis': 173183, 'Jabalya': 168568,
'Hebron': 160470}
```

Accessing dictionary elements

- Access the values in a dictionary using their keys inside square brackets `[]` or the `get()` method
- If you use a key that does not exist, `[]` will raise an error, while `get()` will return None (or a default value if specified)

```
# Accessing values
print(population['Jerusalem'])
```

```
428304
```

```
print(population['Ramallah'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[4], line 1
----> 1 print(population['Ramallah'])

KeyError: 'Ramallah'
```

Accessing dictionary elements

Example :

```
# Accessing values
print(population.get('Jerusalem'))
```

```
428304
```

```
print(population.get('Ramallah'))
```

```
None
```

```
print(population.get('Ramallah', 100000))
```

```
100000
```

Modifying Dictionary

- You can add new key-value pairs or update existing ones:

```
# Adding a new key-value pair
population['Nablus'] = 130326

# Updating an existing value
population['Hebron'] = 160000

print(population)
```

Removing Elements From a Dictionary

- **pop(key):** Removes the item with the specified key and returns its value.
- **del:** Removes the item with the specified key. (May delete completely dictionary)
- The **clear()** method empties the dictionary

```
removed_value = population.pop('Nablus')
print(population)
print(removed_value)
```

```
del population['Hebron']
print(population)
```

Loop Through a Dictionary

- When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(x)

brand
model
year
```

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(thisdict[x])

Ford
Mustang
1964
```

```
for x in thisdict.keys():
  print(x)
```

Loop through both *keys* and *values*

```
for x, y in thisdict.items():
  print(x, y)
```

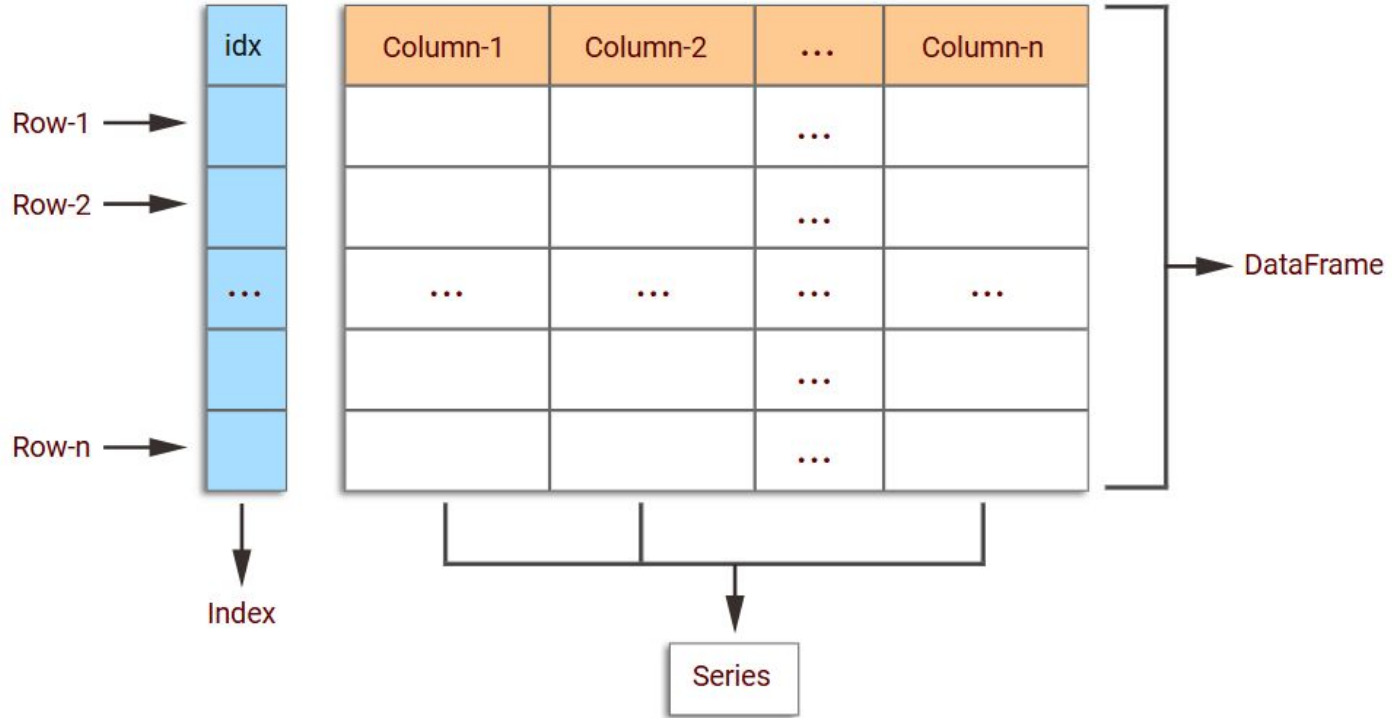
```
# You can also use the values() method to return values of a dictionary:
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.values():
  print(x)
```

Pandas

Pandas

- **Pandas** is a powerful and widely-used open-source Python library designed for data manipulation and analysis
- It provides flexible and efficient data structures, primarily DataFrames and Series, which simplify working with structured data such as tables, spreadsheets, or SQL databases.
  - **Series**: A one-dimensional array with index
  - **DataFrame**: A two-dimensional data structure with one or more series having the same index
- Pandas is built on top of NumPy, which provides fast, low-level array operations
- To use Pandas, you need to install it (if not already installed) using:
  - ***pip install pandas***

Pandas Data Structure



Pandas : Creating Series

- A Series is a one-dimensional array that can hold data of any type (integers, floats, strings, etc.) and is labeled by an index

```
import pandas as pd

# Creating a Series
data = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(data)
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

```
0    1
1    7
2    2
dtype: int64
```

Pandas : Key/Value Objects as Series

- You can also use a key/value object, like a dictionary, when creating a Series.

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories)

print(myvar)

day1    420
day2    380
day3    390
dtype: int64
```

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories, index = ["day1", "day2"])

print(myvar["day2"])

380
```

Pandas : DataFrame

- A DataFrame is a two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes

```
# Creating a DataFrame from dictionary
data = {
    'Name': ['Rami', 'Ahmad', 'Layla'],
    'Age': [25, 30, 28],
    'City': ['Nablus', 'Gaza', 'Jerusalem']
}
df = pd.DataFrame(data)
print(df)
```

```
    Name  Age       City
0   Rami   25     Nablus
1  Ahmad   30       Gaza
2  Layla   28  Jerusalem
```

```
df.index.values
```

```
array([0, 1, 2], dtype=int64)
```

Pandas : Reading from CSV file

- Pandas supports reading and writing data from various formats, making it easy to import and export data.
- **Read CSV**: `pd.read_csv('file.csv')`
- **Write to CSV**: `df.to_csv('output.csv')`

Tray: df.head(5)

```
# Reading data from a CSV file
df = pd.read_csv('data/supermarket_sales.csv', index_col='Invoice ID')
df
```

Invoice ID	Branch	City	Customer type	Gender	Product line	Unit price	Quantity	Tax 5%	Total	Date	Time	Payment	cogs	gross margin percentage	gross income	Rating
750-67-8428	A	Yangon	Member	Female	Health and beauty	74.69	7	26.1415	548.9715	1/5/2019	13:08	Ewallet	522.83	4.761905	26.1415	9.1
226-31-3081	C	Naypyitaw	Normal	Female	Electronic accessories	15.28	5	3.8200	80.2200	3/8/2019	10:29	Cash	76.40	4.761905	3.8200	9.6

Pandas : Accessing Data in DataFrames

- Pandas provides several ways to access and manipulate data within DataFrames:
  - **Selecting Columns:** Use column names to select one or more columns.

```
# Using column name as a key
quantities = df['Quantity']

# Using dot notation (if the column name has no spaces or special characters)
quantities = df.Quantity
quantities.head(5)
```

```
Invoice ID
750-67-8428    7
226-31-3081    5
631-41-3108    7
123-19-1176    8
373-73-7910    7
Name: Quantity, dtype: int64
```

Pandas : Accessing Data in DataFrames

- **Selecting Multiple Columns:**

```
# Selecting multiple columns
subset = df[['Product line', 'Unit price', 'Quantity']]
subset.head(5)
```

Invoice ID	Product line	Unit price	Quantity
750-67-8428	Health and beauty	74.69	7
226-31-3081	Electronic accessories	15.28	5
631-41-3108	Home and lifestyle	46.33	7
123-19-1176	Health and beauty	58.22	8
373-73-7910	Sports and travel	86.31	7

Pandas : Accessing Data in DataFrames

- **Selecting Rows:**
  - Use `.loc[]` for index-based (**label**) selection and `.iloc[]` for position-based selection.

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

```
Branch                          A
City                       Yangon
Customer type              Member
Gender                     Female
Product line    Health and beauty
Unit price                 74.69
Quantity                       7
Tax 5%                   26.1415
Total                   548.9715
Date                    1/5/2019
Time                       13:08
Payment                  Ewallet
cogs                      522.83
gross margin percentage  4.761905
gross income             26.1415
Rating                       9.1
Name: 750-67-8428, dtype: object
```

```
# Selecting rows by index
df.loc['750-67-8428']
```

```
Branch                          A
City                       Yangon
Customer type              Member
Gender                     Female
Product line    Health and beauty
Unit price                 74.69
Quantity                       7
Tax 5%                   26.1415
Total                   548.9715
Date                    1/5/2019
Time                       13:08
Payment                  Ewallet
cogs                      522.83
gross margin percentage  4.761905
gross income             26.1415
Rating                       9.1
Name: 750-67-8428, dtype: object
```

Pandas : Accessing Data in DataFrames

- **Selecting Rows:**
  - Use `.loc[]` for index-based (**label**) selection and `.iloc[]` for position-based selection.

```
# Selecting rows by position
df.iloc[0]
```

```
# Selecting range of rows by position
df.iloc[:3]
```

```
<div> ●●●
```

```
df.iloc[-1]
```

```
Branch A ●●●
```

Pandas : Accessing Data in DataFrames

- **Selecting Rows and Columns.**

```
# Selecting rows and columns
df.loc[['750-67-8428', '631-41-3108', '373-73-7910'],['Product line', 'Unit price', 'Quantity']]
```

Invoice ID	Product line	Unit price	Quantity
750-67-8428	Health and beauty	74.69	7
631-41-3108	Home and lifestyle	46.33	7
373-73-7910	Sports and travel	86.31	7

Pandas : Analyzing DataFrames

- `head()` method returns the headers and a specified number of rows, starting from the top
- if the number of rows is not specified, the `head()` method will return the top **5 rows**
- `tail()` method returns the headers and a specified number of rows, starting from the bottom
- `info()` gives you more information about the data set

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

print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  164 non-null    float64
dtypes: float64(1), int64(3)
```

There are 5 rows with no value at all, in the "Calories" column,

Logic & Control Flow

Logic & Control Flow

- logic and control flow are essential for decision-making in your code.
- You can control the flow of your program using comparison operators, boolean operators, and conditional statements like `if`, `elif`, and `else.`
- **Comparison operators** are used to compare values and return a Boolean value (`True` or `False`). They are often used in decision-making processes.

Operator	Meaning	Example	Result
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`5 > 3`	`True`
`<`	Less than	`3 < 5`	`True`
`>=`	Greater than or equal to	`5 >= 5`	`True`
`<=`	Less than or equal to	`3 <= 5`	`True`

Logic & Control Flow

- **Numpy recap**

```
import numpy as np
bmi = np.array([21.85171573, 20.97505669, 21.75028214, 24.7473475 , 21.44127836])
```

```
bmi > 22
```

```
array([False, False, False,  True, False])
```

```
bmi[bmi > 22]
```

```
array([24.7473475])
```

Logic & Control Flow

- **Example comparisons**

```
2 <= 3
```

```
True
```

```
'two' <= 'three'
```

```
False
```

```
2 <= 'three'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 1
----> 1 2 <= 'three'

TypeError: '<=' not supported between instances of 'int' and 'str'
```

Logic & Control Flow

- **Boolean operators**

Operator	Description	Example	Result
`and`	Returns `True` if **both** statements are `True`	`True and False`	`False`
`or`	Returns `True` if **at least one** is `True`	`True or False`	`True`
`not`	Reverses the Boolean value	`not True`	`False`

```
# Create two NumPy arrays
a = np.array([10, 20, 30, 40])
b = np.array([15, 20, 25, 50])

# Element-wise comparisons
print(a < b)   # [ True False False  True ]
print(a > b)   # [False False  True False ]
print(a == b)  # [False  True False False ]
print(a != b)  # [ True False  True  True ]
```

```
bmi = np.array([21.8, 20.9, 21.7, 24.7 , 21.4, 22.1 ])
print(bmi[bmi > 21])
```

```
[21.8 21.7 24.7 21.4 22.1]
```

Logic & Control Flow

- **NumPy logical functions**

```
bmi = np.array([20, 22, 24])
mask = bmi > 21 and bmi < 23
```

This gives a `ValueError: The truth value of an array with more than one element is ambiguous.`

- To overcome this issue, you can use the functions **`logical_and`**, **`logical_or`** and **`logical_not`**, the "array equivalents" of and or and not in numpy arrays:

```
bmi = np.array([20, 22, 24])

# Correct way to filter values between 21 and 23
mask = np.logical_and(bmi > 21, bmi < 23)
print(mask)          # Output: [False  True False]
print(bmi[mask])     # Output: [22]
```

Conditional Statements

Conditional Statements

- Allow you to execute different code blocks depending on certain conditions.
  - `if`: Executes a block of code if a condition is True.
  - `elif`: Used for additional conditions after the initial if.
  - `else`: Executes a block of code if none of the previous conditions were True.

```
age = 25

if age < 18:
    print("You are a minor.")
elif age >= 18 and age < 65:
    print("You are an adult.")
else:
    print("You are a senior.")
```

```
You are an adult.
```

Filtering Data in DataFrames (Using Pandas):

- In Pandas, you can filter data based on conditions, similar to how you use conditional statements in Python. You can apply comparison and boolean operators to filter rows in a DataFrame
- Example :

```
import pandas as pd
df = pd.read_csv("https://drive.google.com/uc?export=download&id=1B4w7L2eAehzIN-l4pZ4dD7Evp0cg5Lq8", index_col='Invoice ID')
df.head(5)
```

Invoice ID	Branch	City	Customer type	Gender	Product line	Unit price	Quantity	Tax 5%	Total	Date	Time	Payment	cogs	gross margin percentage	gross income	Rating
750-67-8428	A	Yangon	Member	Female	Health and beauty	74.69	7	26.1415	548.9715	1/5/2019	13:08	Ewallet	522.83	4.761905	26.1415	9.1
226-31-3081	C	Naypyitaw	Normal	Female	Electronic accessories	15.28	5	3.8200	80.2200	3/8/2019	10:29	Cash	76.40	4.761905	3.8200	9.6
631-41-3108	A	Yangon	Normal	Male	Home and lifestyle	46.33	7	16.2155	340.5255	3/3/2019	13:23	Credit card	324.31	4.761905	16.2155	7.4

Filtering Data in DataFrames (Using Pandas):

- **Filtering using comparison operators (**compare two values)
- Find invoices with more than 1000 Total ?

```
df['Total'] > 1000
```

```
Invoice ID
750-67-8428    False
226-31-3081    False
631-41-3108    False
123-19-1176    False
373-73-7910    False
               ...
233-67-5758    False
303-96-2227     True
727-02-1313    False
347-56-2442    False
849-09-3807    False
Name: Total, Length: 1000, dtype: bool
```

```
df[df['Total'] > 1000]
```

```
df[df['Total'] > 1000]
```

Invoice ID	Branch	City	Customer type	Gender	Product line	Unit price	Quantity	Tax 5%	Total	Date	Time	Payment	cogs	gross margin percentage	gross income	Rating
234-65-2137	C	Naypyitaw	Normal	Male	Home and lifestyle	95.58	10	47.790	1003.590	1/16/2019	13:32	Cash	955.8	4.761905	47.790	4.8
687-47-8271	A	Yangon	Normal	Male	Fashion accessories	98.98	10	49.490	1039.290	2/8/2019	16:20	Credit card	989.8	4.761905	49.490	8.7

Filtering Data in DataFrames (Using Pandas):

- **Filtering using boolean operators**
- Find invoices where the Total exceeds 1000 and the Product line belongs to Home and lifestyle.

```
df[df['Total'] > 1000 and df['Product line'] == 'Home and lifestyle']
```

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().

Option 1:

```
df[np.logical_and(df['Total'] > 1000, df['Product line'] == 'Home and lifestyle')]
```

Filtering Data in DataFrames (Using Pandas):

- **Option 2:**
- If **a** and **b** are Boolean NumPy arrays, the **&** operation returns the elementwise-and of them.
- You can combine multiple conditions using **&** (for **and**) and **|** (for **or**). Ensure to wrap each condition in parentheses.

```
df[(df['Total'] > 1000) & (df['Product line'] == 'Home and lifestyle')]
```

Invoice ID	Branch	City	Customer type	Gender	Product line	Unit price	Quantity	Tax 5%	Total	.....
234-65-2137	C	Naypyitaw	Normal	Male	Home and lifestyle	95.58	10	47.790	1003.590	
751-41-9720	C	Naypyitaw	Normal	Male	Home and lifestyle	97.50	10	48.750	1023.750	
744-16-7898	B	Mandalay	Normal	Female	Home and lifestyle	97.37	10	48.685	1022.385	

Loops

Loops

- The **`for`** loop is used to iterate over a sequence (like a list, dictionary, string, or any other iterable object) and execute a block of code for each item in the sequence.

```
# Looping over a list
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

- The **`while`** loop continues executing a block of code as long as a specified condition is **`True`**.

```
# Using a while loop
count = 0
while count < 5:
    print(count)
    count += 1
```

Using enumerate() in Loops

- The `enumerate()` function allows you to loop over a sequence while keeping track of both the index and the item.

```
# Using enumerate to get index and value
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(f"Index: {index}, Fruit: {fruit}")
```

```
Index: 0, Fruit: apple
Index: 1, Fruit: banana
Index: 2, Fruit: cherry
```

Looping Over Strings

- Since strings are sequences of characters, you can loop over them just like a list

```
# Looping over a string
word = "Python"
for char in word:
    print(char)
```

```
P
y
t
h
o
n
```

Looping over Dictionaries

- In Python, you can loop over dictionaries to access keys, values, or both

Looping over keys:

```
person = {'name': 'Ahmad', 'age': 25, 'city': 'Jerusalem'}
for key in person:
    print(key)
```

```
name
age
city
```

Looping over values:

```
for value in person.values():
    print(value)
```

```
Ahmad
25
Jerusalem
```

Looping over key-value pairs:

```
for key, value in person.items():
    print(f"{key}: {value}")
```

```
name: Ahmad
age: 25
city: Jerusalem
```

Looping over NumPy Array

- With NumPy arrays, you can use loops, but the power of NumPy lies in its ability to handle **element-wise operations efficiently**.

```
import numpy as np

# Creating a NumPy array
arr = np.array([10, 20, 30, 40])

# Looping over a NumPy array
for element in arr:
    print(element)
```

```
10
20
30
40
```

Pandas DataFrames :Looping over Rows

- Pandas DataFrames are 2D structures, and you can loop over their rows or columns
- Although vectorized operations are preferred in Pandas for performance reasons, you can use loops for specific cases

Looping over rows:

iterrows

```
# Looping over rows using iterrows()
for index, row in df.iterrows():
    print(f"Index: {index}, Product: {row['Product line']}, Quantity: {row['Quantity']}")
```

```
Index: 750-67-8428, Product: Health and beauty, Quantity: 7
Index: 226-31-3081, Product: Electronic accessories, Quantity: 5
Index: 631-41-3108, Product: Home and lifestyle, Quantity: 7
Index: 123-19-1176, Product: Health and beauty, Quantity: 8
Index: 373-73-7910, Product: Sports and travel, Quantity: 7
Index: 699-14-3026, Product: Electronic accessories, Quantity: 7
Index: 355-53-5943, Product: Electronic accessories, Quantity: 6
```

Pandas DataFrames :Looping over Columns

- Pandas DataFrames are 2D structures, and you can loop over their rows or columns
- Although vectorized operations are preferred in Pandas for performance reasons, you can use loops for specific cases

Looping over columns:

```
# Looping over columns
for col in df:
    print(f"Column: {col}")
```

```
Column: Branch
Column: City
Column: Customer type
Column: Gender
Column: Product line
Column: Unit price
Column: Quantity
Column: Tax 5%
```

Using apply() instead of loops:

- In data analysis, looping over large datasets can be inefficient and slow.
- Instead of using loops, the **`apply()`** function in Pandas is a powerful tool that allows you to apply a function to each element in a DataFrame or Series without explicitly writing loops.
- It's especially useful when you want to transform data or perform operations on each row or column.
- ***Applying row-wise (`axis=1`):***
- You can use `apply()` to apply a function to each row. Suppose we want to apply a 10% discount to all invoices with a total greater than 500. We can achieve this by creating a new column and using the apply function, which processes each row and calculates the discount for rows where the total exceeds 500:

```
def calc_discount(row):
    return row['Total'] * 0.10 if row['Total'] >= 500 else 0

df['Discount'] = df.apply(calc_discount, axis = 1)
df.head(5)
```

Using apply() instead of loops:

- ***Applying column-wise (`axis=0`):***
- You can also use **`apply()`** to apply a function to each column. Suppose we want to calculate the percentage of missing values in each column of the dataset

```
df.apply(lambda col: col.isnull().sum() / len(col) * 100, axis=0)
```

```
Product line    0.0
Unit price      0.0
Quantity        0.0
Tax 5%          0.0
Total           0.0
cogs            0.0
gross income    0.0
Discount        0.0
dtype: float64
```

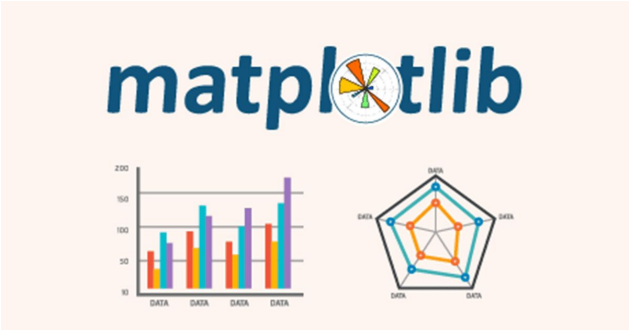
Using lambda functions with apply()

- A **`lambda`** function is a small, anonymous function in Python, defined using the lambda keyword.
- Unlike regular functions defined with **def**, lambda functions can have only a single expression and do not require a name.
- They're often used for quick, one-off operations.
- E.g: calculate the discount using lambda function instead of **`calc_discount()`** function:

```
df['Discount'] = df.apply(lambda row: row['Total'] * 0.10 if row['Total'] >= 500 else 0, axis = 1)
df.head(5)
```

Invoice ID	Product line	Unit price	Quantity	Tax 5%	Total	cogs	gross income	Discount
750-67-8428	Health and beauty	74.69	7	26.1415	548.9715	522.83	26.1415	54.89715
226-31-3081	Electronic accessories	15.28	5	3.8200	80.2200	76.40	3.8200	0.00000

Matplotlib


matplotlib
200
150
100
50
0
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA

Matplotlib

- **Matplotlib** is one of the most popular and widely used **data visualization** libraries in Python
- It provides an easy way to create **static**, **animated**, and **interactive** plots and charts, making it an essential tool for anyone working with data

Why data visualization is important ?

- Data visualization is a critical aspect of data analysis because it transforms raw data into a visual format
- Making it **easier to understand and interpret.**
- Visualizations transform complex data into clear and simple visuals, making patterns, trends, and outliers more apparent.

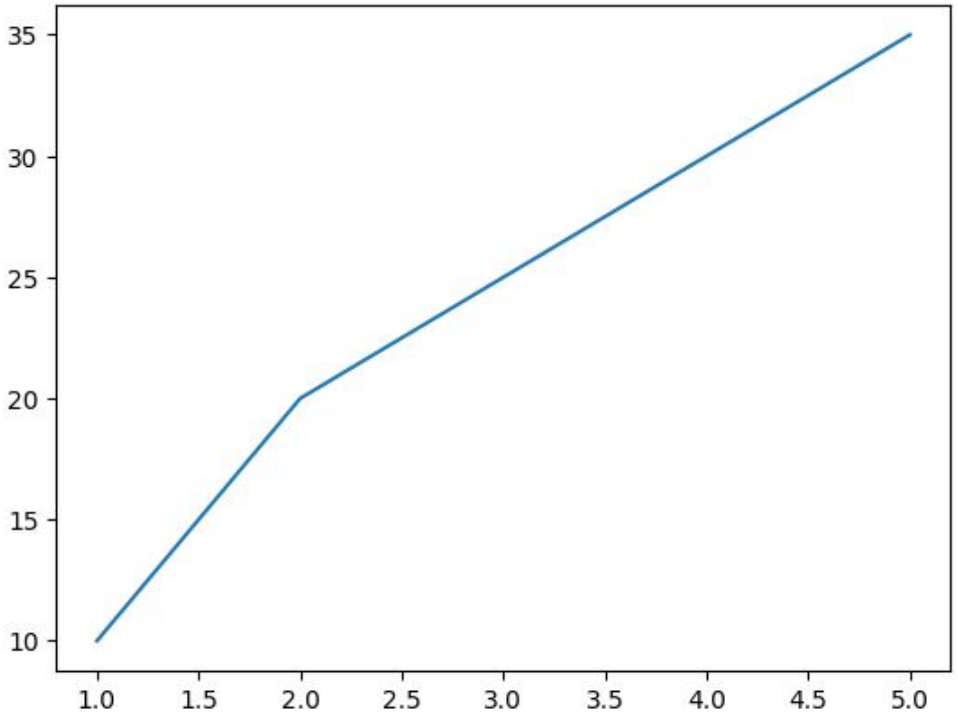
Basic Usage of Matplotlib

- The most common way to use Matplotlib is through its `pyplot` module, which provides a simple interface for creating basic plots

Matplotlib

- Example :

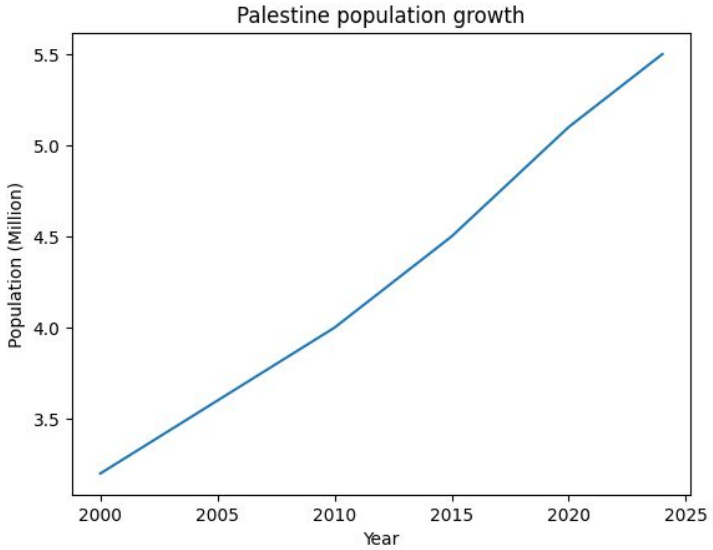
```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 35]

# Create a simple plot
plt.plot(x, y)

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



Types of Plots: Line Plot

- Line plot is a a basic plot used to display information as a series of data points connected by straight lines.
- Perfect for **visualizing trends over time**

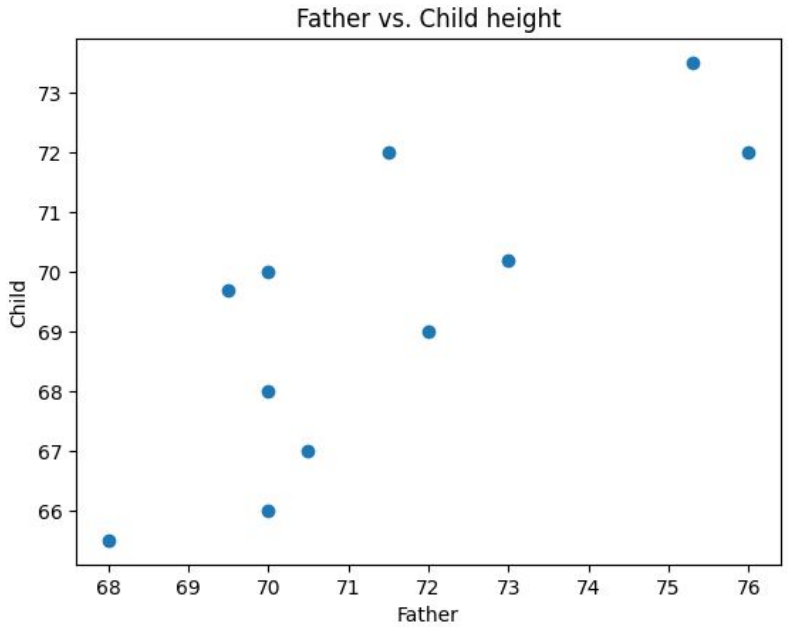
```
year = [2000, 2005, 2010, 2015, 2020, 2022, 2023, 2024]
population = [3.2, 3.6, 4, 4.5, 5.1, 5.3, 5.4, 5.5]

# Create a line plot
plt.plot(year, population)

# Add title and labels
plt.title("Palestine population growth")
plt.xlabel("Year")
plt.ylabel("Population (Million)")

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



Types of Plots: Scatter Plot

- **Scatter plot** shows the relationship between two sets of data by plotting individual data point

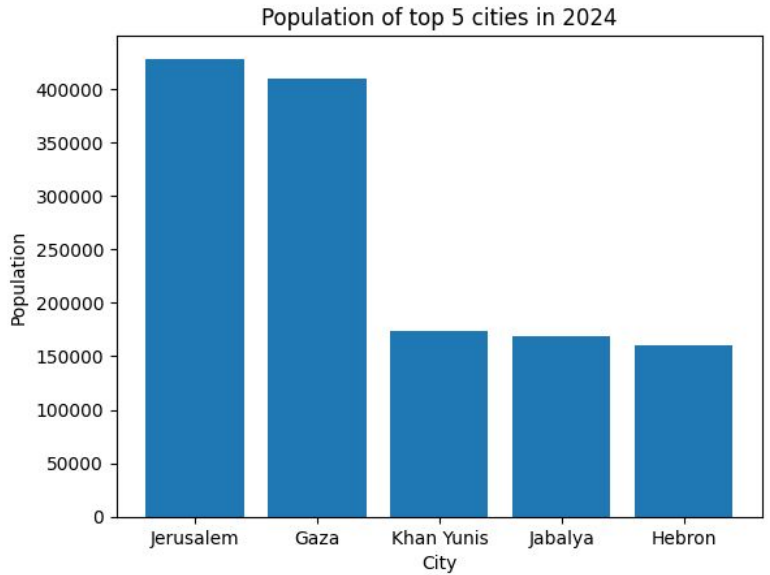
```
father = [76, 68, 73, 72, 71.5, 70, 70, 70.5, 69.5, 75.3, 70]
child = [72, 65.5, 70.2, 69, 72, 66, 70, 67, 69.7, 73.5, 68]

# Create a simple scatter plot
plt.scatter(father, child)

# Add title and labels
plt.title("Father vs. Child height")
plt.xlabel("Father")
plt.ylabel("Child")

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



Types of Plots: Bar Plot

- **Bar plot** used to compare quantities across different categories

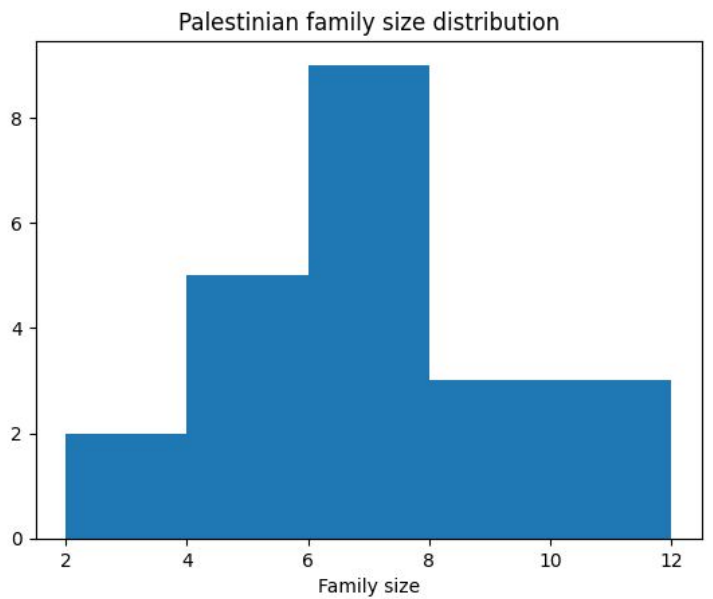
```
city = ['Jerusalem', 'Gaza', 'Khan Yunis', 'Jabalya', 'Hebron']
population = [428304, 410000, 173183, 168568, 160470]

plt.bar(city, population)

# Add title and labels
plt.title("Population of top 5 cities in 2024")
plt.xlabel("City")
plt.ylabel("Population")

plt.show()
```



Types of Plots: Histogram

- **Histogram** visualizes the distribution of a dataset by dividing it into bins and plotting the frequency of each bin

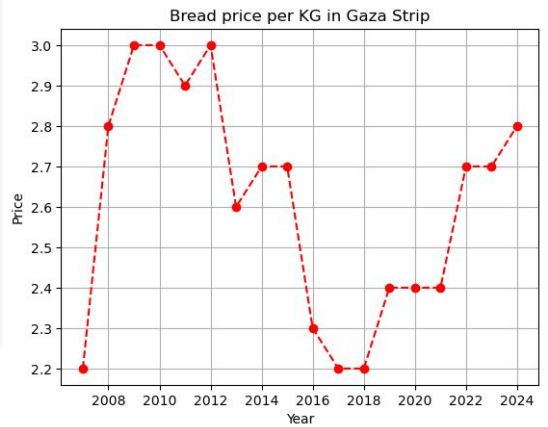
```
family_size = [5, 3, 7, 5, 6, 7, 8, 7, 6, 4, 10, 12, 7, 9, 10, 6, 8, 7, 7, 2, 4, 4]
plt.hist(family_size, bins=5)

# Add title and labels
plt.title("Palestinian family size distribution")
plt.xlabel("Family size")

plt.show()
```



Customization in Matplotlib

- **Change Colors**: You can specify custom colors for lines, bars, or markers
- **Add Legends:** Use legends to describe each plot in case of multiple plots on the same figure
- **Modify Axes:** Adjust the x and y axis scales, add grid lines, or change the limits of the plot

```
import datetime
year = [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018,
        2019, 2020, 2021, 2022, 2023, 2024]

bread_price = [2.2, 2.8, 3, 3, 2.9, 3, 2.6, 2.7 , 2.7 , 2.3, 2.2 , 2.2, 2.4, 2.4, 2.4, 2.7, 2.7, 2.8]

year = [datetime.datetime(y,1,1) for y in year]
plt.plot(year, bread_price, color='red', linestyle='--', marker='o')

plt.title("Bread price per KG in Gaza Strip")
plt.xlabel("Year")
plt.ylabel("Price")
plt.grid(True)
plt.show()
```

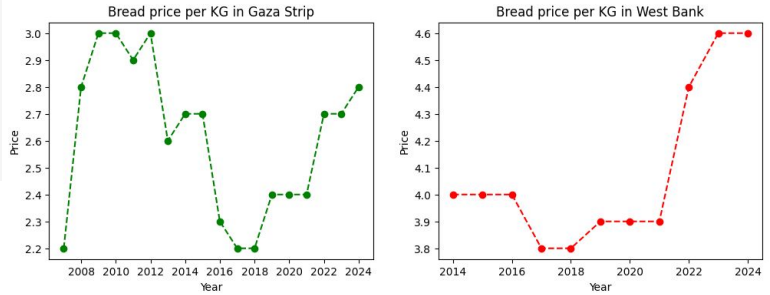


Subplots and Multiple Plots

- You can create multiple plots in a single figure using **subplots**.
- This is useful when you want to visualize different aspects of the data side by side

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

year_wb = [2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024]
bread_price_wb = [4. , 4. , 4. , 3.8, 3.8, 3.9, 3.9, 3.9, 4.4, 4.6, 4.6]

axes[0].plot(year, bread_price, color='green', linestyle='--', marker='o')
axes[0].set_title("Bread price per KG in Gaza Strip")
axes[0].set_xlabel("Year")
axes[0].set_ylabel("Price")

axes[1].plot(year_wb, bread_price_wb, color='red', linestyle='--', marker='o')
axes[1].set_title("Bread price per KG in West Bank")
axes[1].set_xlabel("Year")
axes[1].set_ylabel("Price")

plt.show()
```

1 row and 2 columns = 2 subplots side by side



End Of Slides

Thank you.