Python
study material

"Hey there, it's me, **Py**!”

your friendly Python serpent,
ready to guide you through a
sssspectacular Python journey

Let'sssss sssslither
into the world of
Python magic,
sssshall we?"

https://github.com/Vinodhini96

INDEX

Chapter 1. Introduction to Python

1.1 What is Python?

- Python is a versatile and user-friendly programming language.
- It's known for its simplicity and readability, making it great for beginners.
- Python is used in various fields like web development, data analysis, artificial intelligence, and scientific computing.
- Its flexibility and extensive libraries make tasks easier and more efficient.

1.2 Why Python?

1. Friendly and Approachable:

Py is known for being super friendly and approachable, especially for beginners. He's like the friendly snake you'd want to meet in a forest.

2. Versatile and Adaptable:

Py has a magical ability to transform into whatever you need! Need a calculator? Py's got you covered. Want to surf the web? Py's browser mode is ready. And when it comes to visualizing data, Py's a pro!

3. Huge Python Community:

Py isn't alone; he's part of a massive Python community. Together, they help and support each other in the Python jungle. You're never alone when you have Py as your guide.

4. Simple and Readable Code:

Py is all about clarity. He loves writing code that's easy to read and understand, making Python code a breeze for programmers of all levels.

5. Solves Problems with Ease:

Py is like a superhero in the coding world. He can tackle complex problems with ease, saving the day for programmers and businesses worldwide.

6. Always Evolving:

Py is constantly shedding its old skin (versions) to become better and more powerful. With each update, Python becomes more incredible than ever.

1.3 Applications of Python?

- Web Development
- Machine Learning and Artificial Intelligence
- Data Analysis and Visualization
- Data Science
- Game Development
- Software Development
- Desktop GUI
- Web Scraping Application
- Scientific Computing
- Finance and Trading

1.4 what is IDE?

Integrated Development Environments (IDEs) are like your go-to toolkit for programming. They simplify the coding process, help you find and fix mistakes, and provide a comfortable environment for writing and testing code.

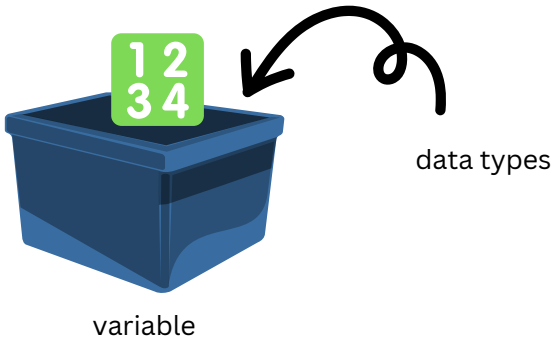
- **All-in-One Workspace**: IDEs are like a complete workshop for developers. They gather everything you need in one place, from writing code to testing and debugging.
- **Writing Made Easy:** In IDEs, writing code becomes a breeze. They highlight syntax errors, suggest completions, and help you navigate your code, making writing code faster and less error-prone.
- **Spotting Mistakes:** Debugging in IDEs is like searching for mistakes in a puzzle. They let you pause your code at specific points, inspect variables, and step through each line to find and fix issues.
- **Project Organization:** IDEs help you keep your projects tidy. You can organize files, manage dependencies, and switch between different parts of your project effortlessly.
- **Testing Ground:** You can run and test your code directly within IDEs. They have built-in tools that compile or interpret your code, letting you see the results immediately.
- **Working Together:** Some IDEs let you collaborate with others on the same codebase. You can share your work, track changes, and work on code together in real-time.
- **Easy to Use:** IDEs come with user-friendly interfaces tailored to specific programming languages. They make it easy to write and manage code, even for beginners.

Following are a few Python IDEs:

Chapter 2 : Variables

2.1 Variables

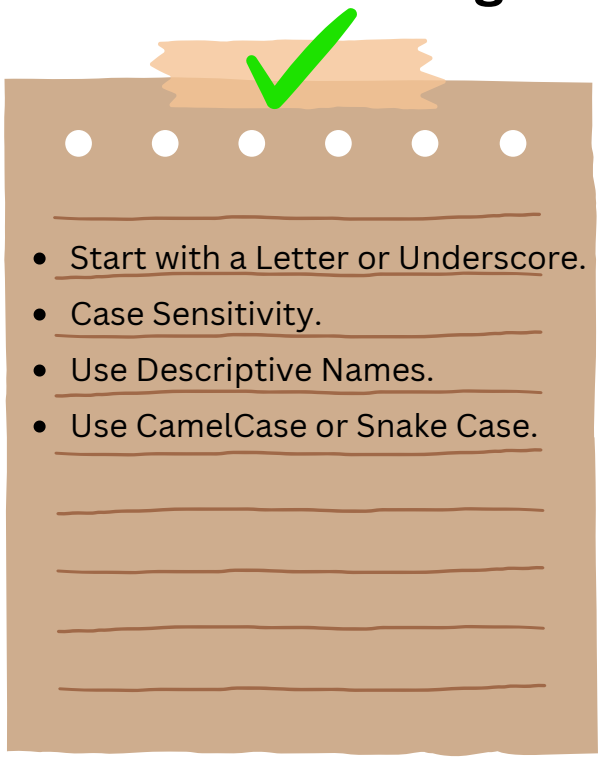
- Variables are like containers that hold information or values in a program.
- They act as labels for data stored in the computer's memory.
- you can think of them as boxes that can hold different types of things.



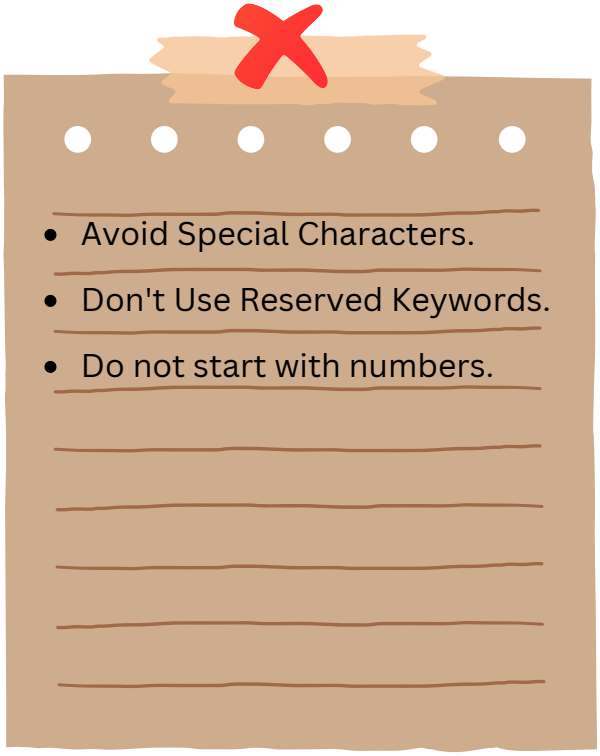
sample code

```
number = 23                    # number is a variable name
words = “hello”                # words is a variable name
```

2.2 Variable naming rules



- Start with a Letter or Underscore.
- Case Sensitivity.
- Use Descriptive Names.
- Use CamelCase or Snake Case.

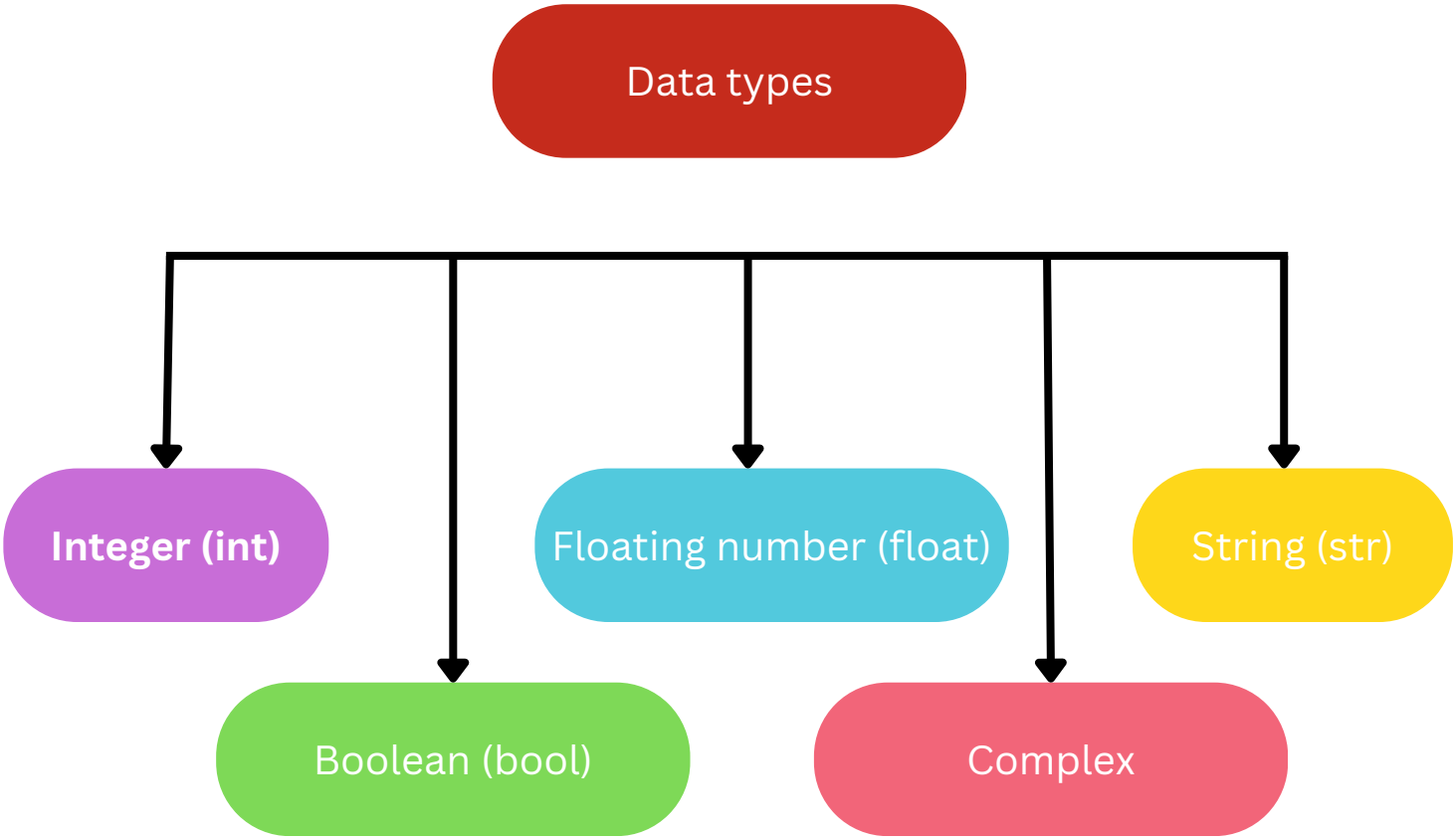


- Avoid Special Characters.
- Don't Use Reserved Keywords.
- Do not start with numbers.

Chapter 3 : Data Types

Data types specify the kind of data a variable can hold. Python has several built-in data

3.1 Different types of Data



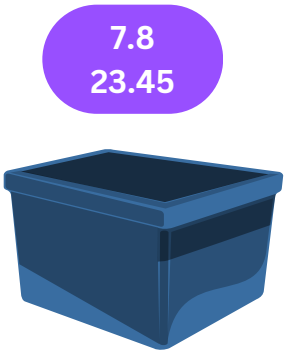
3.1.1. Integer (int)

- Integers are whole numbers without any decimal point.
- They can be positive or negative.
- **Example:** Pincode, account no., marks, phone no., age etc.

```
# integer
student_count = 40
print(student_count)
```

3.1.2. Floating number (float):

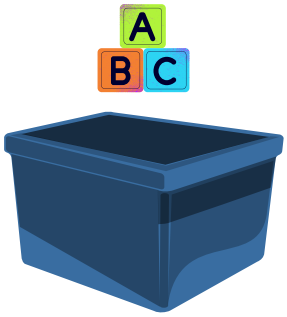
- Represents decimal numbers. Like measuring someone's height.
- **Example:** measurements, height, weight, tempertaure, time, percentage, salary, conversion etc.




```
# float
temperature = 98.5
print(temperature)
```

3.1.3. String (str):

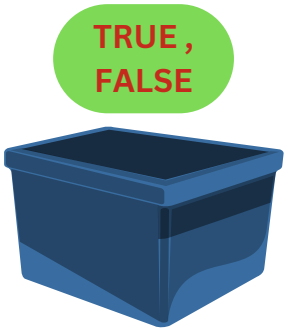
- Strings are sequences of characters enclosed within single (' ') or double (" ") quotes.
- They can contain letters, digits, symbols, and whitespace.
- **Example:** “suresh”, place name, things name, planets, vehicles,
- alphanumeric = "123abc", pan number, vehicle number, register number, IFSC number, passport number, password, "$%^%&&"




```
# string
name = "Vinie"
print(name)
```

3.1.4. Boolean (bool):

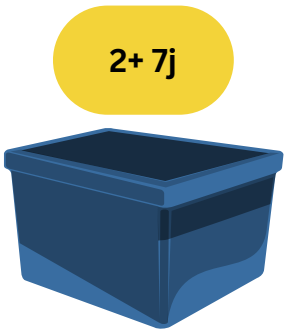
- Booleans represent truth values: True or False.
- They are often used for logical operations and conditional statements.
- **Example:** True/ False, 1/0, on/off, yes/no,




```
# boolean
is_eligible = True
is_eligible
```

3.1.5. Complex (complex):

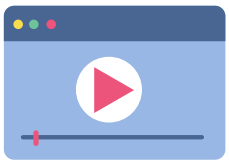
- Complex numbers have a real part and an imaginary part, represented as a + bj, where 'a' is the real part and 'b' is the imaginary part.
- 'j' is used to represent the imaginary unit (√(-1)).
- **Example:** 3 + 4j, -2.5 - 1.8j.




```
# complex
comp_numbe = 3+ 5j
print(comp_number)
```

Each data type in Python has its specific characteristics and is suitable for different types of data manipulation tasks.

For more knowledge , Click on the link below to get a video tutorial of Python Variables.



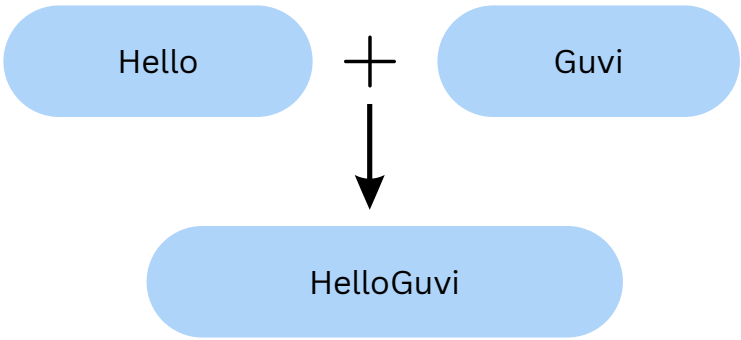
3.2 String operations

String operations in Python allow you to manipulate and work with text data efficiently. Here are some common operations and methods used with strings:

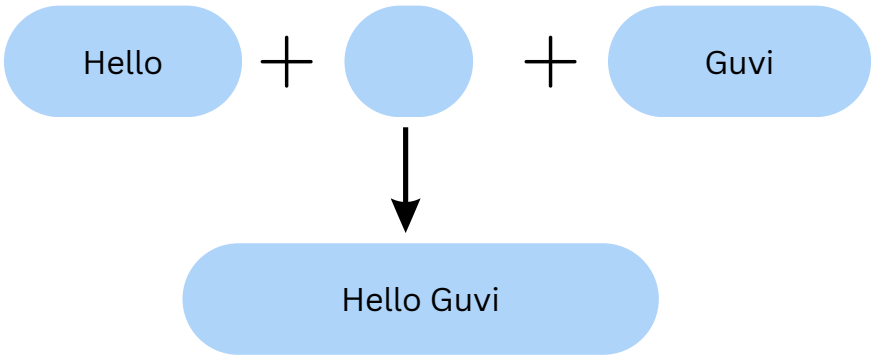
1. Concatenation:

Combining strings using the (+) operator

```
greeting = "Hello, "
name = "Guvi"
message = greeting + name                    # Result: "HelloGuvi"
```




```
greeting = "Hello, "          # Adding space between 2 strings
name = "Guvi"
message = greeting + " " + name                    # Result: "Hello Guvi"
```

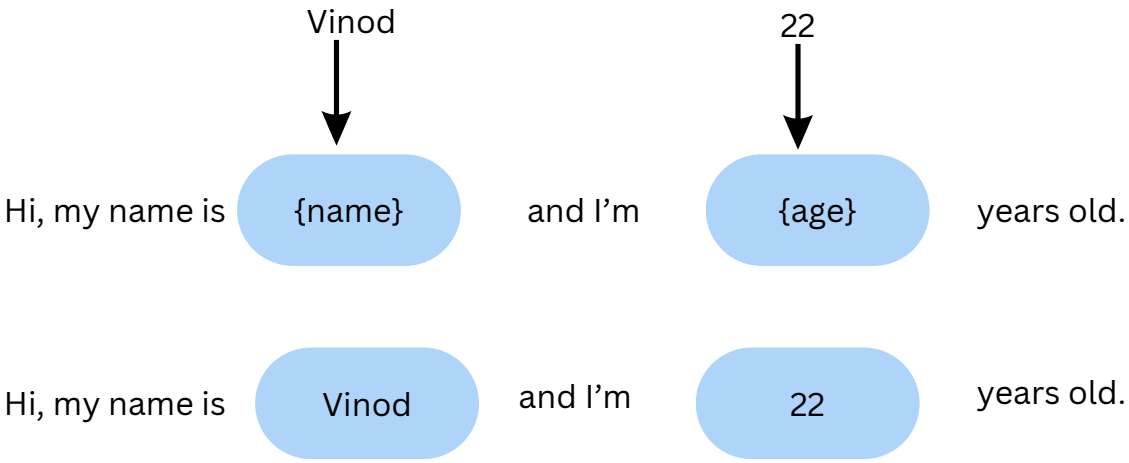


2. String Interpolation / Formatting:

Combining variables and strings using f-strings or the format() method:

```
# format() method
name = "Vicky"
age = 20
message = "Hi, my name is {} and I'm {} years old.".format(name, age)
```

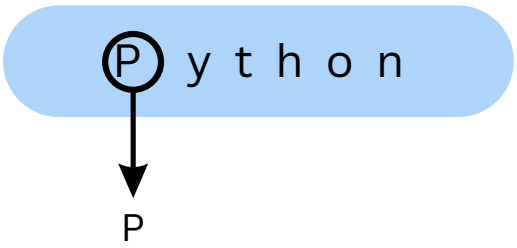
```
name = "Vinod"    # f-string
age = 22
message = f"Hi, my name is {name} and I'm {age} years old."
```



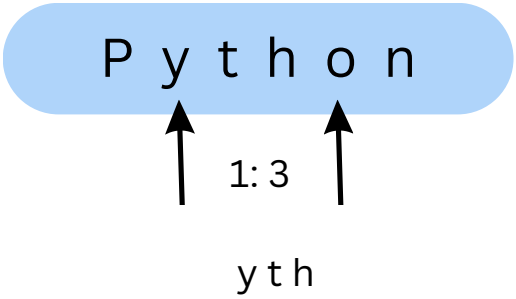
3. Indexing and Slicing:

Accessing specific characters or portions of a string.

```
text = "Python"
print(text[0])      # Accessing the first character: 'P'
```




```
print(text[1:4])      # Slicing from index 1 to 3: 'yth'
```



4. Length:

Getting the length of a string using the len() function:

```
text = "Hello, World!"
length = len(text)      # Result: 13
```

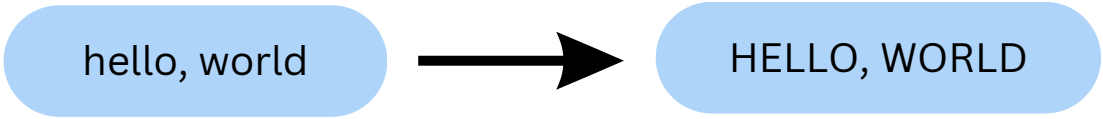
How lengthy is it?

5. Conversion:

Changing the case of strings using **upper(), lower(), capitalize(), or title()** methods

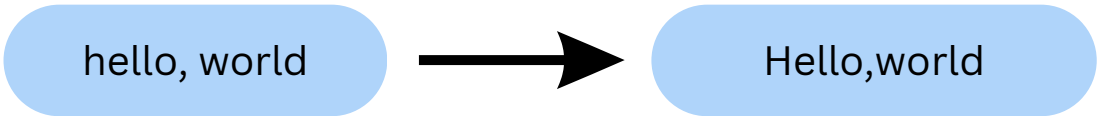
a) upper():

```
# Convert to uppercase:
text = "hello, world!"
print(text.upper())          #Output: 'HELLO, WORLD!'
```



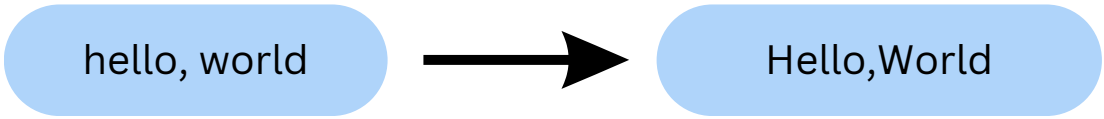
b) capitalize():

```
# Capitalize the first letter:
print(text.capitalize())          # Output: 'Hello, world!'
```



c) title():

```
#title()
print(text.title())          # Output: Hello, World!
```



d) lower():

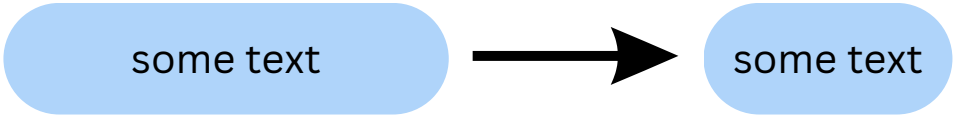
```
# Convert to lowercase:
text = "HELLO WORLD!"
print(text.lower())        #  Output: hello world!
```

6. Strip:

Removing whitespace or specific characters from the beginning or end of a string using strip(), lstrip(), or rstrip() methods:

a) strip():

```
# Remove leading/trailing whitespace:
text = "    Some text    "
print(text.strip())          # Output : 'Some text'
```



b) lstrip():

```
# Remove leading whitespace  - lsrtip()
print(text.lstrip())              # Output: 'Some text  '
```

c) rstrip():

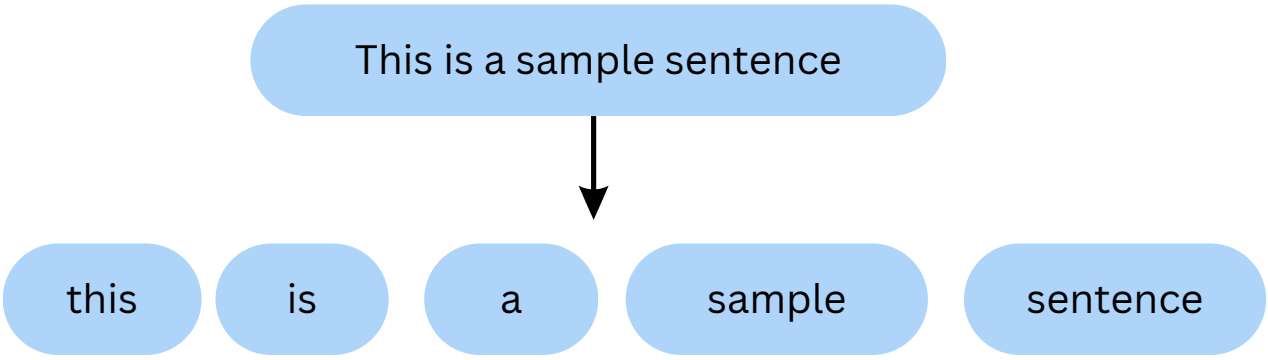
```
#  Remove trailing whitespace
print(text.rstrip())             # Output: ' Some text'
```

7. Splitting and Joining:

Splitting a string into a list of substrings using split() and joining a list of strings into a single string using join():

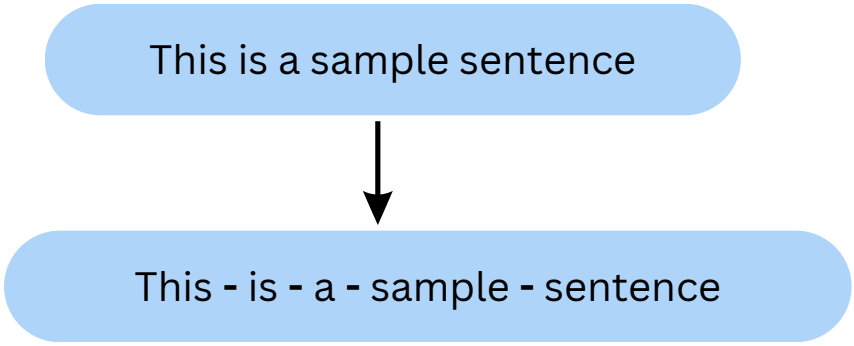
a) split():

```
#split()
sentence = "This is a sample sentence"
words = sentence.split()  # Split by spaces, resulting in a list of words
print(words)
# Output: ['This', 'is', 'a', 'sample', 'sentence']
```



b) join():

```
# join()
joined_sentence = '-'.join(words)
# Using join to concatenate the words with a hyphen
print(joined_sentence)
# Output : This-is-a-sample-sentence
```

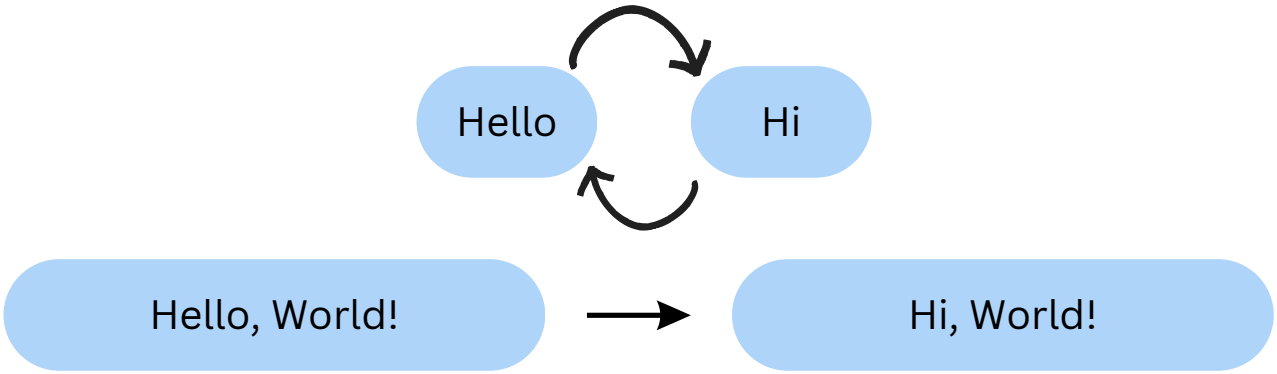


8. Find and Replace:

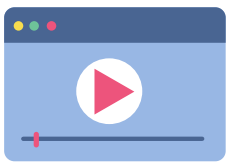
Finding substrings within a string using find() or index() and replacing substrings using replace():

```
text = "Hello, World!"

#find()
print(text.find("World"))       # Find the index of 'World'.
#Output : 7
```


```
# replace()
print(text.replace("Hello", "Hi"))     # Replace 'Hello' with 'Hi'
# Output:  'Hi, World!'
```



For more knowledge , Click on the link below to get a video tutorial of Python String Operations



Chapter 4 : Operators

Operators in programming are symbols that perform operations on variables or values.

4.1 Arithmetic Operators:

1. Addition (+):

Adds values together.

```
a = 5 + 3          # Result: 8
```

2. Subtraction (-):

Subtracts one value from another.

```
b = 7 - 4          # Result: 3
```

3.Multiplication (*):

Multiplies values.

```
c = 2 * 6          # Result: 12
```

4.Division(/):

Divides one value by another.

```
d = 10 / 2

# Result: 5.0 (even if the result is a whole number, Python sometimes represents it as a float)
```

5.Modulo (%):

Gives the remainder after division.

```
e = 17 % 5         # Result: 2 (remainder of 17 divided by 5)
```

3.2 Comparison Operators:

1. Equal (==):

Checks if two values are equal.

```
3 == 3                  # Result: True
```

2. Not Equal (!=):

Checks if two values are not equal.

```
4 != 2                  # Result: True
```

3.Greater Than (>) / Less Than (<):

Checks if one value is greater/less than the other.

```
5 > 3                   # Result: True
4 < 1                   # Result: False
```

4.DGreater Than or Equal (>=) / Less Than or Equal (<=):

Checks if one value is greater/less than or equal to the other.

```
6 >= 6                  # Result: True
2 <= 1                  # Result: False
```

3.3 Logical Operators:

1. AND (and):

Returns True if both conditions are True.

```
(3 > 1) and (5 < 10)            # Result: True
```

2. OR (or):

Returns True if at least one condition is True.

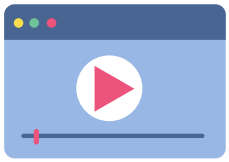
```
(2 == 2) or (4 != 4)            # Result: True
```

3. NOT (not):

Reverses the logical state of the operand.

```
not (3 > 1)                    # Result: False
```

For more knowledge , Click on the link below to get a video tutorial of Python Operators



Chapter 5: Conditional statements

Conditional statements in Python are used to execute different code blocks based on specified conditions.

Imagine you are getting ready to leave your house for the day. You have a decision to make about what to wear based on the weather outside. This decision-making process is similar to how conditional statements work in programming.

```
weather = "sunny"

if weather == "rainy":                  # checks if weather is rainy
   print("wear_clothes: raincoat and carry an umbrella")
elif weather == "sunny":                # checks if weather is sunny
   print("wear_clothes: shorts and a T-shirt")
else:
   print("wear_clothes: jeans and a shirt")
```

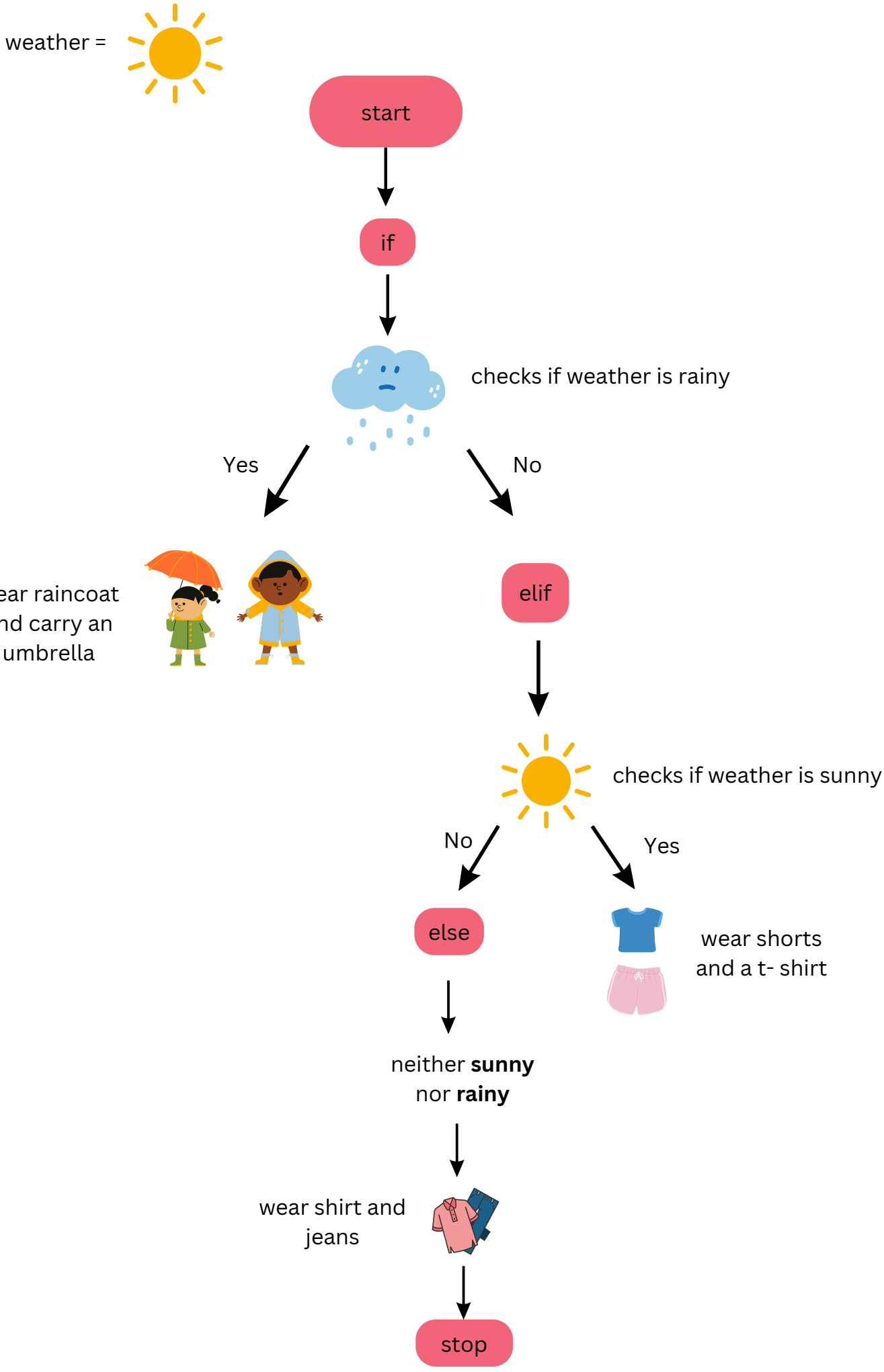
1. **Decision-Making Process:** You look outside and check the weather.
2. **Conditional Check:**

if block
If the weather is rainy (weather == "rainy" is True), you decide to wear clothes like raincoat and carry an umbrella. Otherwise,

elif block
If the weather is sunny(weather == "sunny" is True), you decide to wear clothes like shorts and a t-shirt

else block
if it's neither, you opt for warmer attire like a shirt and jeans.

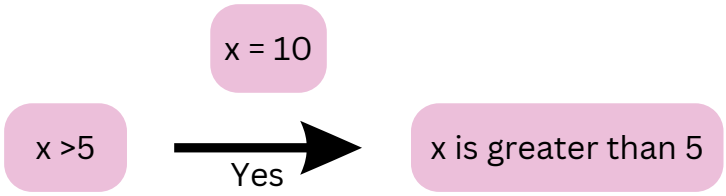
3. **Applying the Decision:** You wear the chosen outfit based on the weather condition.


weather =
start
if
checks if weather is rainy
Yes
No
wear raincoat and carry an umbrella
elif
checks if weather is sunny
No
Yes
else
wear shorts and a t- shirt
neither **sunny** nor **rainy**
wear shirt and jeans
stop

5.1 If Statement:

Executes a block of code if a condition is true.

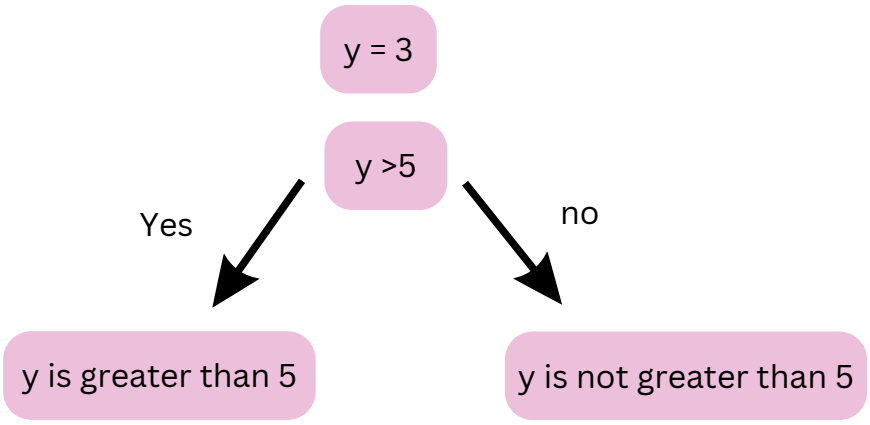
```
x = 10
if x > 5:
print("x is greater than 5")
```



5.2 If-Else Statement:

Executes one block of code if the condition is true and another block if the condition is false.

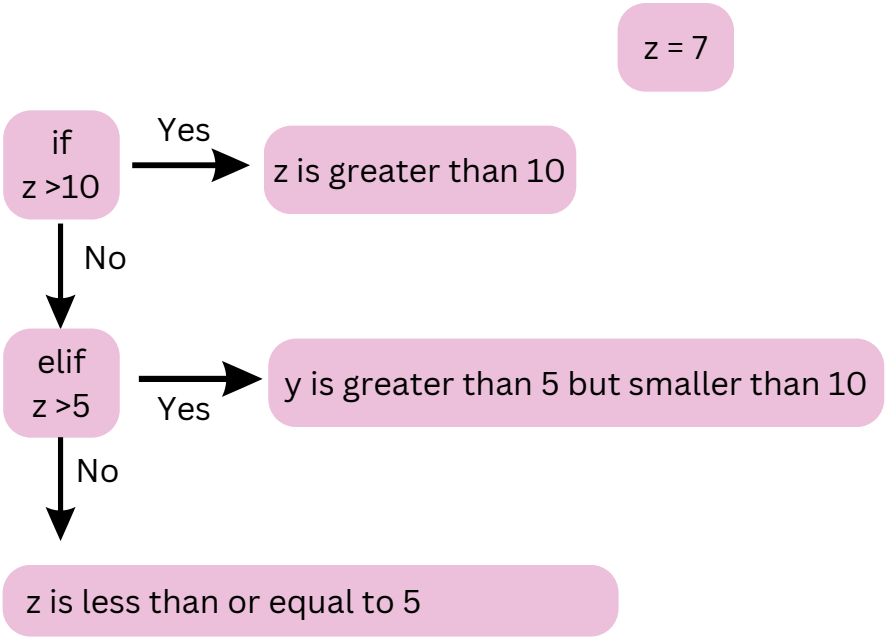
```
y = 3
if y > 5:
    print("y is greater than 5")
 else:
    print("y is not greater than 5")
```



5.3 Else-if (elif) Statement:

Executes one block of code if the condition is true and another block if the condition is false.

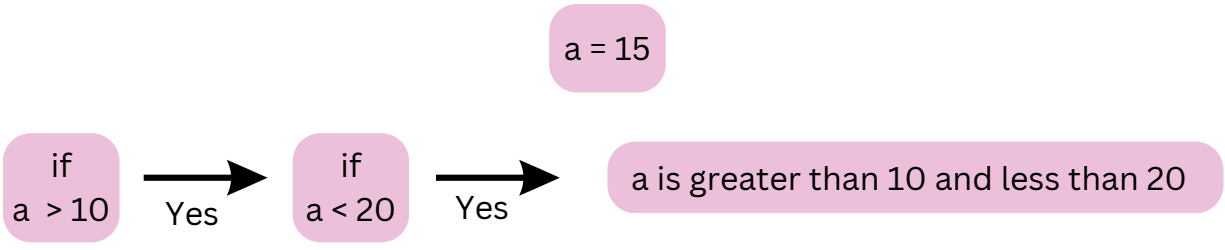
```
z = 7
if z > 10:
   print("z is greater than 10")
elif z > 5:
   print("z is greater than 5 but smaller than 10")
else:
   print("z is less than or equal to 5")
```



5.4 Nested Conditional Statements:

Using conditional statements within each other to check more complex conditions.

```
a = 15
if a > 10:
   if a < 20:
      print("a is greater than 10 and less than 20")
```

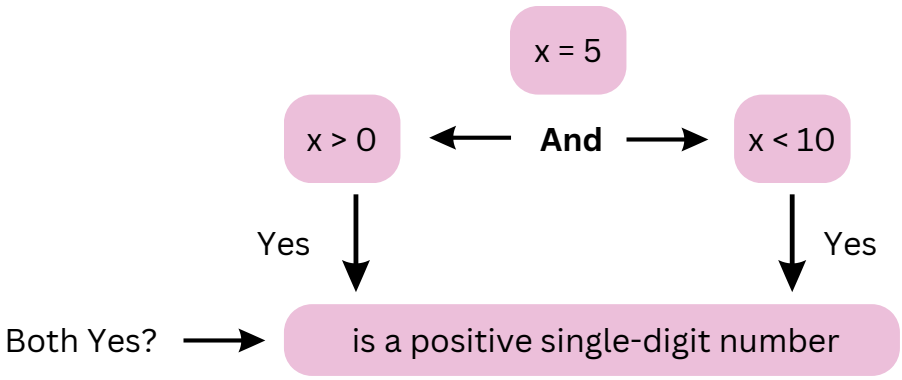


5.5 Logical Operators (and, or, not):

Combining conditions using logical operators.

a) and operator:

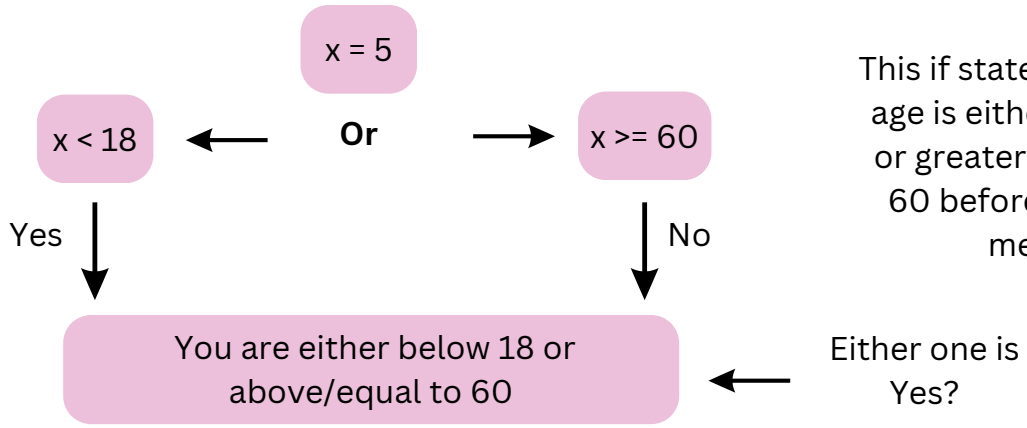
```
x = 5
if x > 0 and x < 10:
    print("x is a positive single-digit number")
```



This if statement checks if x is both greater than 0 and less than 10 before printing the message.

b) or operator:

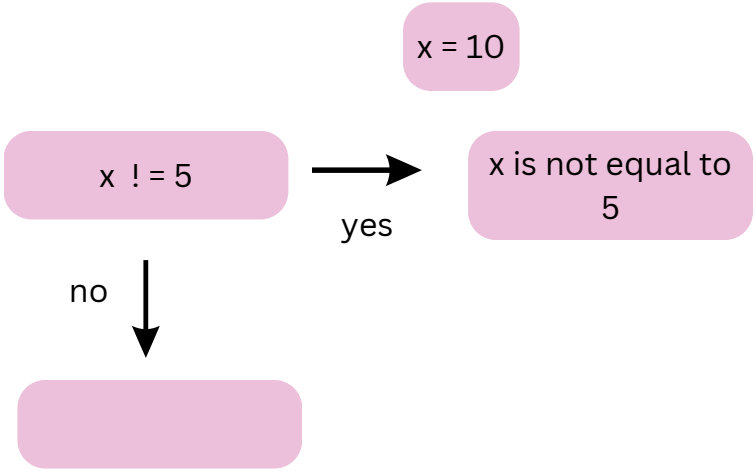
```
age = 5
if age < 18 or age >= 60:
    print("You are either below 18 or above/equal to 60")
```



This if statement checks if age is either less than 18 or greater than/equal to 60 before printing the message.

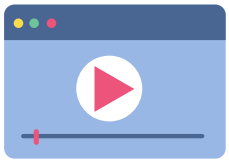
c) not operator:

```
x = 10
if not x == 5:
    print("x is not equal to 5")
```



The not operator negates the condition, checking if x is not equal to 5 before printing the message.

For more knowledge , Click on the link below to get a video tutorial of Python Conditonal Statements.



Chapter 6 : Loops

Loops in programming are structures that allow you to execute a block of code repeatedly.
They help automate repetitive tasks and perform operations on a collection of items or until a certain condition is met.

Types of Loop

For Loop

- Used when the number of iterations is known or can be determined.
- The loop variable is automatically initialized by iterating over the elements of the iterable.
- Iterates over a sequence (list, tuple, string, etc.) or any iterable object.
- Follows a definite control flow, and the loop continues until all elements in the iterable are processed.

While Loop

- Used when the number of iterations is not known in advance and depends on a condition.
- The loop variable must be initialized before entering the loop.
- Continues to iterate as long as the specified condition is True.
- The loop may continue indefinitely if the condition is not met or may execute zero or more times.

For Loop Example:
Filling a Bucket with Water Using a Mug

- Imagine you have an empty bucket, and you want to fill it with water using a mug.
- The for loop here is like a repetitive action of scooping water from a source (like a tap) and pouring it into the bucket until the bucket is full.
- You know where to start (an empty bucket) and where to stop (when the bucket is full).
- Each time you scoop water with the mug, you add it to the bucket until the desired level is reached.

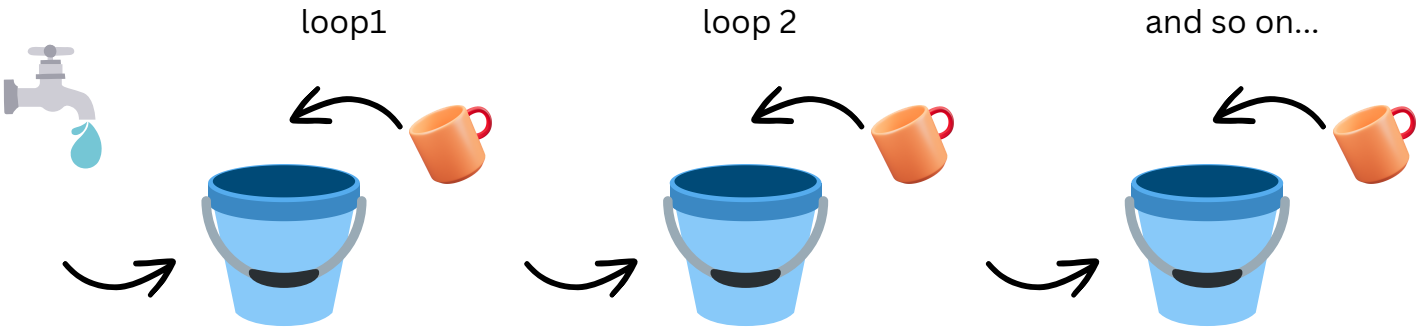
While Loop Example:
Treadmill Running

- Think of a person running on a treadmill. They keep running until they reach a certain condition or goal, such as running for a specific duration or achieving a target distance.
- The while loop represents this continuous action of running on the treadmill until the condition to stop is met.
- The runner doesn't have a fixed number of steps or laps to take;
- instead, they keep running as long as the condition (e.g., time or distance) hasn't been fulfilled.
- Once the condition is met, they stop running.

In both cases:

- The for loop involves iterating over a sequence (e.g., scooping water with a mug) until a specific endpoint (e.g., filling the bucket).
- The while loop continues executing a block of code as long as a condition remains true (e.g., the runner keeps running until a target is reached).

For Loop



start from empty bucket

stop when the bucket is full

While Loop

start running on a treadmill

keep on running

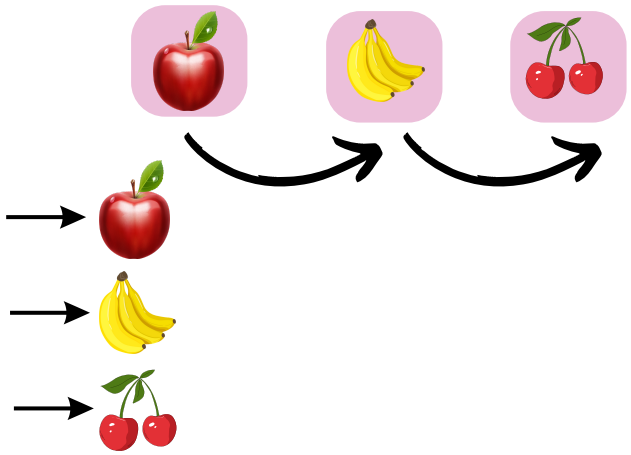
Stop when you are asked to

Condition to stop the loop

6.1 For Loops:

For loops are used for iterating over a sequence (such as lists, tuples, strings, or other iterable objects) and executing a block of code for each element in the sequence.

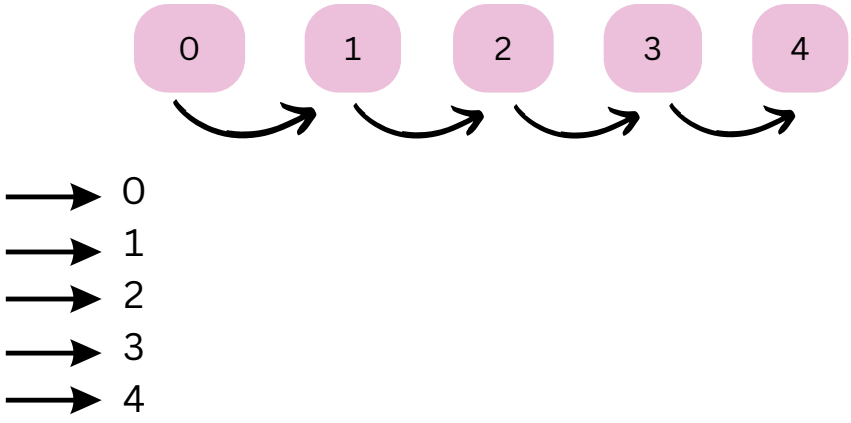
```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)          # Iterates through each fruit in the list and prints it
```



6.1.1 Range Function with For Loops:

Using the range() function to iterate a specified number of times.

```
for i in range(5):
    print(i)          # Prints numbers from 0 to 4
```

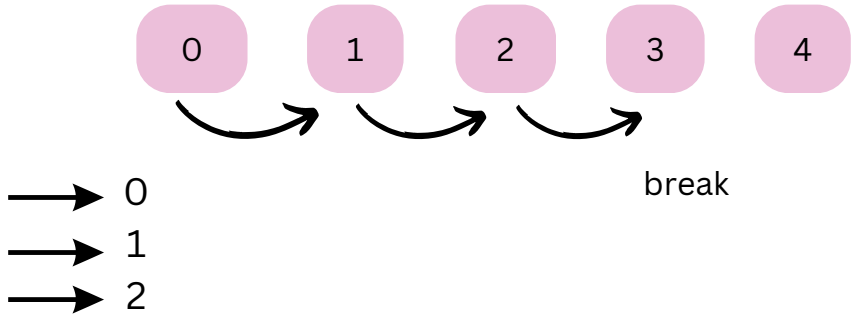


6.1.2 Loop Control Statements:

- **break:** Terminates the loop prematurely based on a condition.
- **continue:** Skips the current iteration and moves to the next one.

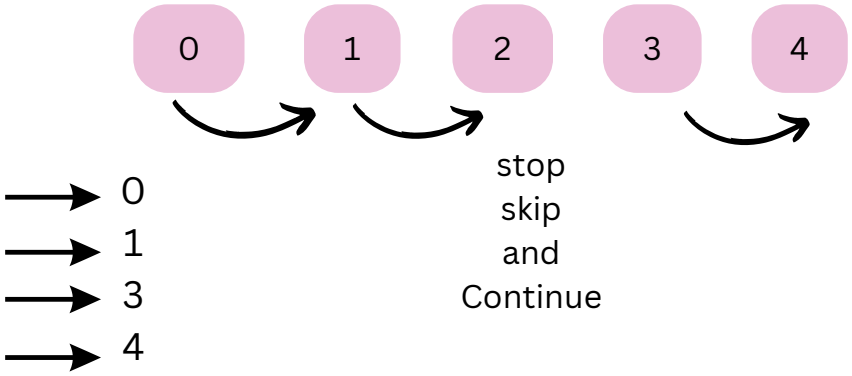
Break

```
for i in range(5):
if i == 3:
    break           # Terminates the loop when i becomes 3
print(i)            # Prints numbers from 0 to 2
```



Continue

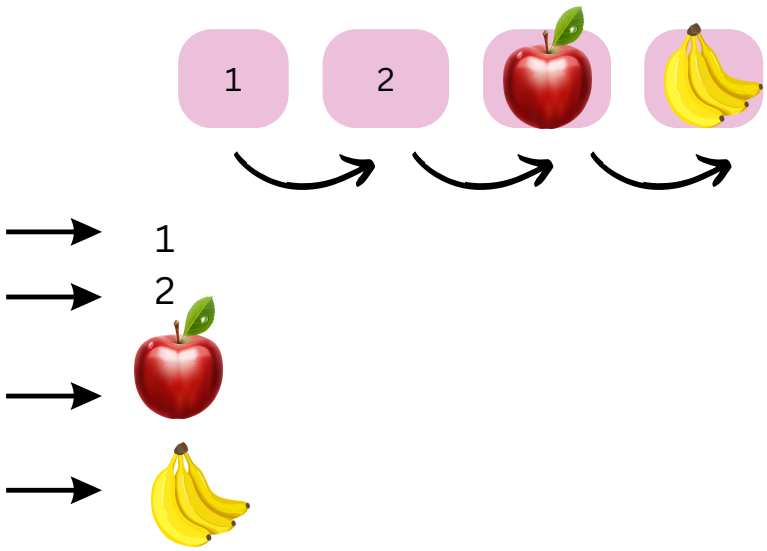
```
for i in range(5):
if i == 2:
    continue        # Skips printing '2' and continues to the next iteration
print(i)            # Prints numbers from 0 to 4, skipping '2'
```



6.1 3 Looping through a Tuple:

Tuples are immutable, but you can iterate through their elements using a for loop

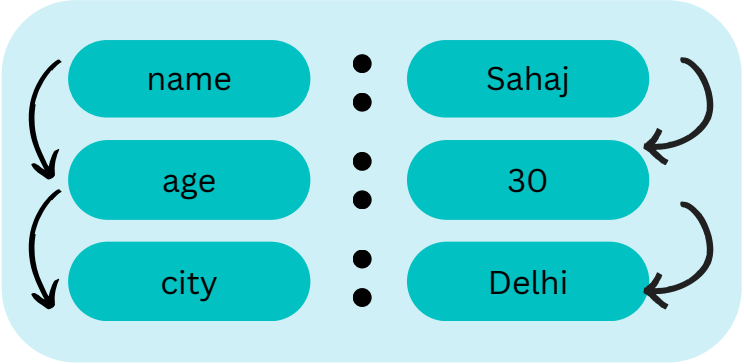
```
my_tuple = (1, 2, 'apple', 'banana')          # create  a tuple

# Looping through elements in the tuple
for item in my_tuple:
    print(item)                               # Prints each item in the tuple
```

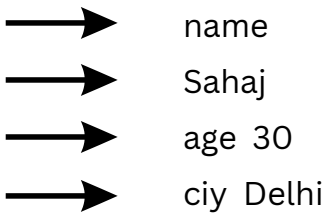


6.1 4 Looping Through Dictionaries:

Iterating through keys, values, or key-value pairs in dictionaries.

```
person = {                                    # create a dictionary
            'name': 'Alice',
            'age': 30,
            'city': 'New York'}

for key, value in person.items():
   print(key, value)                          # Prints key-value pairs in the dictionary
```

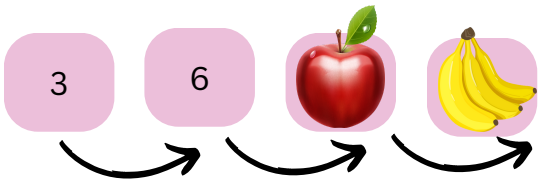


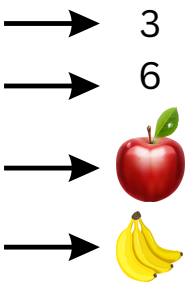


6.1.5 Looping through a Set:

Sets are unordered collections of unique elements. You can loop through them using a for loop as well:

```
my_set = {3, 6, 'apple', 'banana'}

# Looping through elements in the set
for item in my_set:
    print(item)                 # Prints each unique item in the set
```

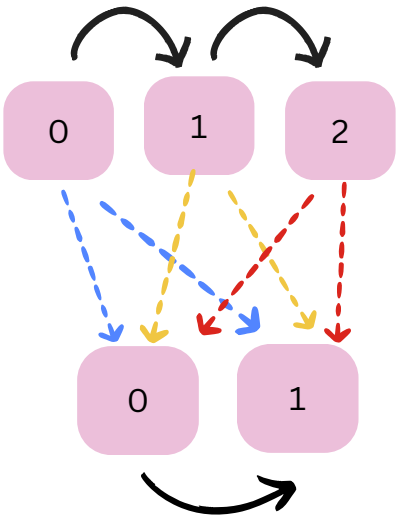


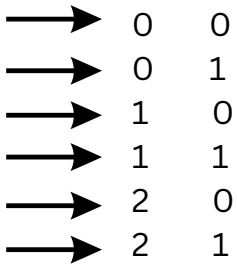


6.1.6 Nested Loops:

Using one loop inside another loop.

```
for i in range(3):
    for j in range(2):
        print(i, j)  # Prints combinations of i and j for each iteration
```

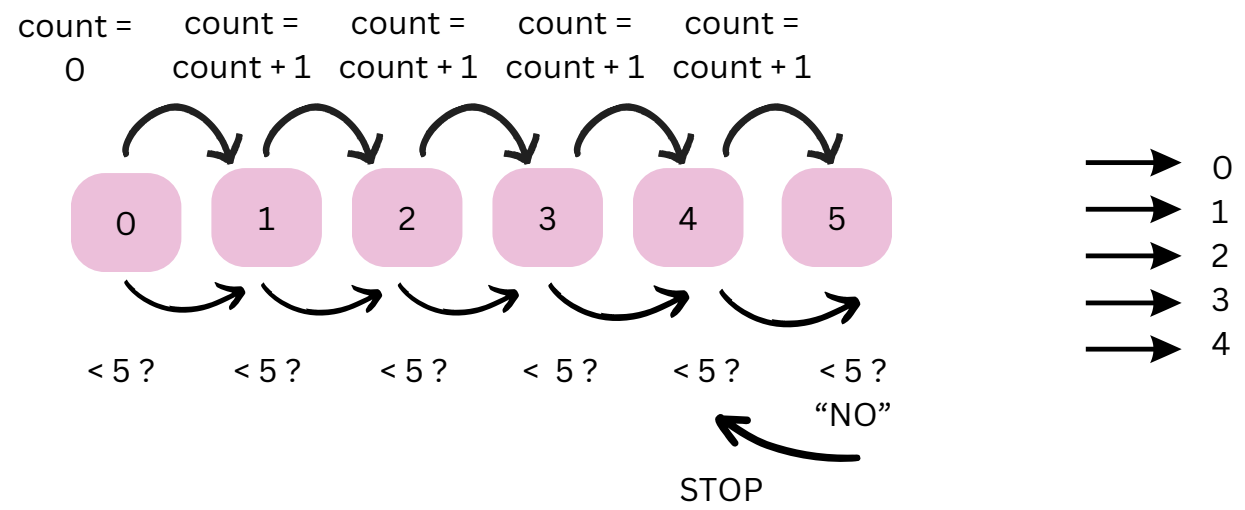




6.2 While Loop:

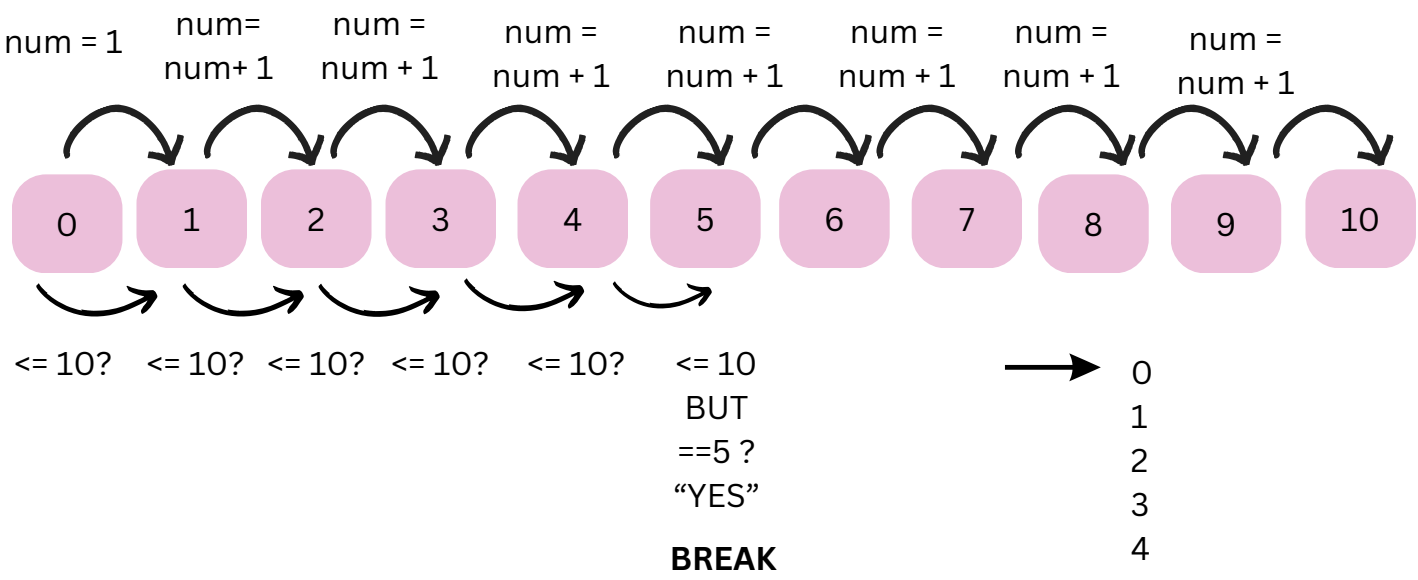
The while loop in Python executes a block of code repeatedly as long as a specified condition remains True.

```
count = 0
while count < 5:
    print(count)
    count += 1
```



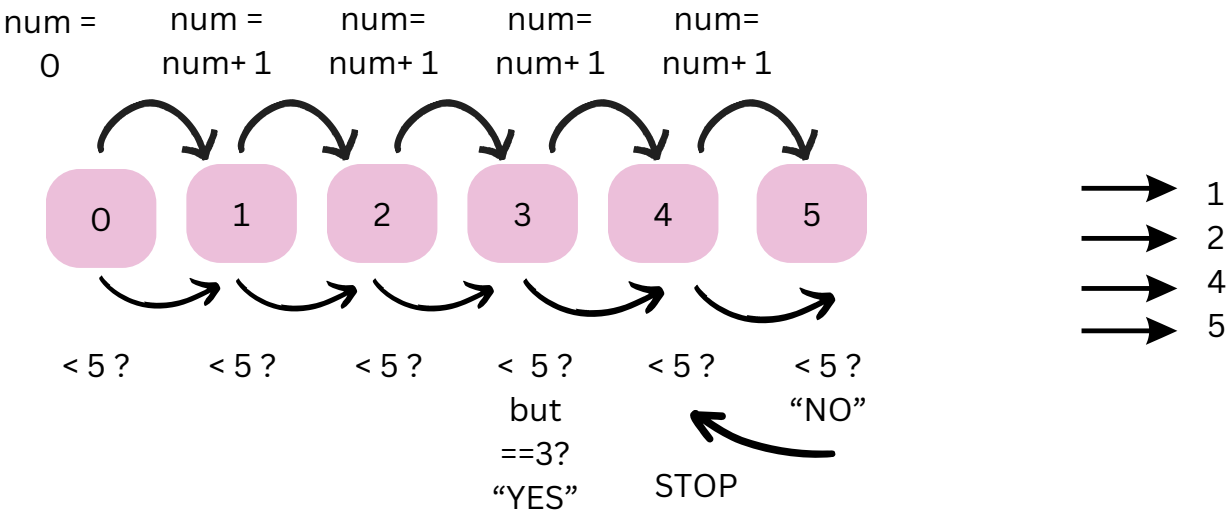
break:

```
num = 0
while num <= 10:
    print(num)
    num += 1
    if num == 5:
        break          # Exit the loop when num equals 5
```



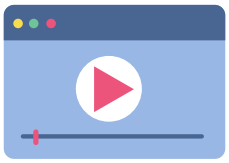
continue:

```
num = 0
while num < 5:
    num += 1
    if num == 3:
        continue                    # Skip printing '3'
    print(num)
```



SKIP

For more knowledge , Click on the link below to get a video tutorial of Python Loops.



Chapter 7: Data Structures

A data structure in programming is like a specific way to organize and store data so that it can be efficiently used, managed, and accessed.

Think of it this way:

Imagine you're organizing your clothes in your wardrobe:

- **Different Ways to Organize:** You might arrange them by type (shirts, pants), color, or size. Each way you organize them represents a different data structure.
- **Efficient Access:** When you need a specific shirt, you know exactly where to look because you've organized them in a way that makes it easy to find what you need quickly.
- **Different Structures for Different Needs:** Just like you might use hangers for shirts and drawers for socks, in programming, different data structures suit different purposes or make certain operations more efficient.

Why Are Data Structures Important?

- **Efficiency:** They help perform operations like searching, sorting, and organizing data more efficiently.
- **Optimized Access:** Choosing the right data structure can significantly speed up data access and manipulation.
- **Problem Solving:** They're crucial for solving complex problems by providing organized ways to handle and process data.

Just like organizing your things at home makes life easier, using the right data structure in programming helps manage and use data effectively, making programs more efficient and easier to work with.

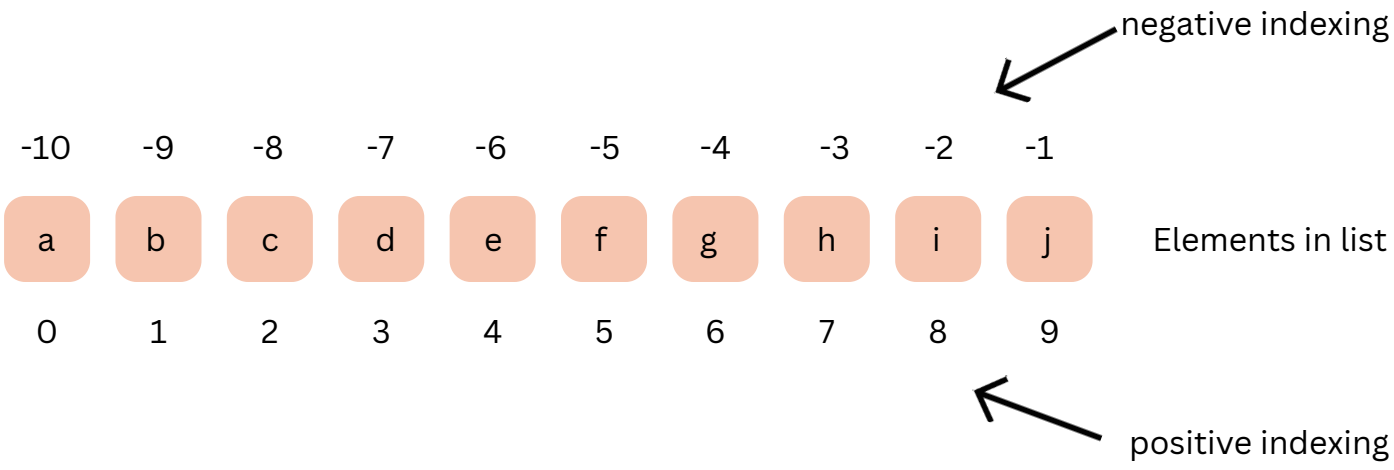
7.1 Lists

Lists in Python are like containers that hold a collection of items.
In daily life, a list can represent things to do, items to buy, or elements in a collection. It's a handy way to keep things organized and easily accessible, just like your real-life lists!

7.1. 1 Characteristics of Lists

- **Mutable:** Lists can be changed after creation; you can add, remove, or modify items.
- **Ordered:** Lists maintain the order of items as they were inserted.
- **Indexing:** Access items by their position (index) in the list.
- **Supports Different Types:** Lists can hold different data types (e.g., strings, numbers, other lists).
- a list is created using **square brackets []** and can hold any combination of elements separated by commas.

7.1. 2 Indexing in Lists

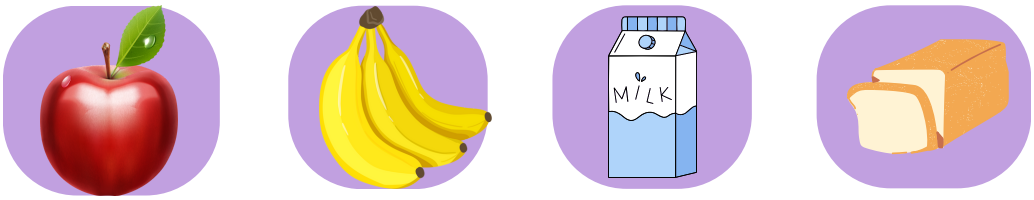


Always remember!! **Positive Index** in list starts with **0** and **negative index** in list starts with **-1**

Lets create a list in Python and Imagine a shopping list.

7.1. 3 Creating Lists

```
shopping_list = ["apples", "bananas", "milk", "bread"]
```



7.1.4 Basic List operations

1. Adding an Item to the List:

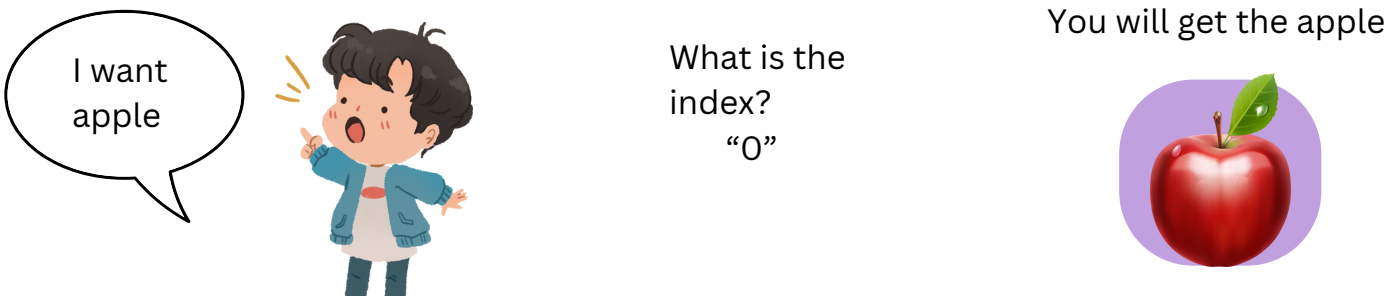
You realize you need eggs too, so you add them to the list.

```
shopping_list.append("eggs")
    # Output: ["apples", "bananas", "milk", "bread", "eggs"]
```

2. Accessing Items in the List:

You grab the first item on the list.

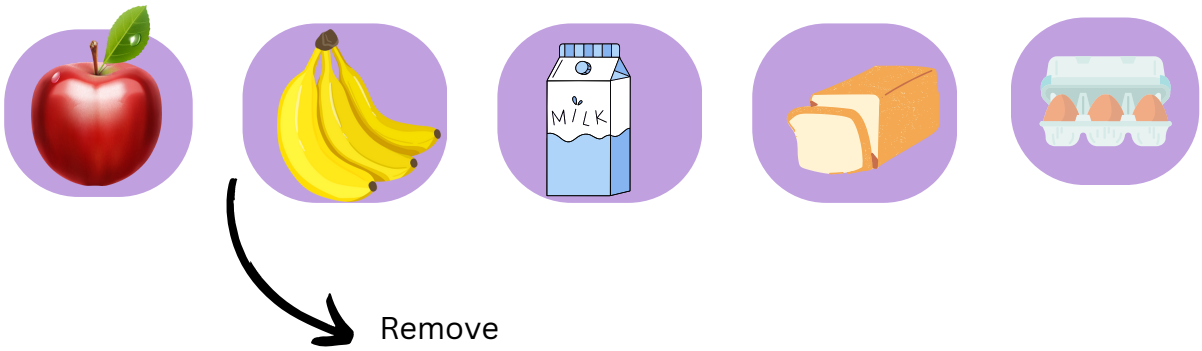
```
first_item = shopping_list[0]    # This will be "apples"
```



3. Removing an Item from the List:

You bought the bananas, so you remove them from your list:

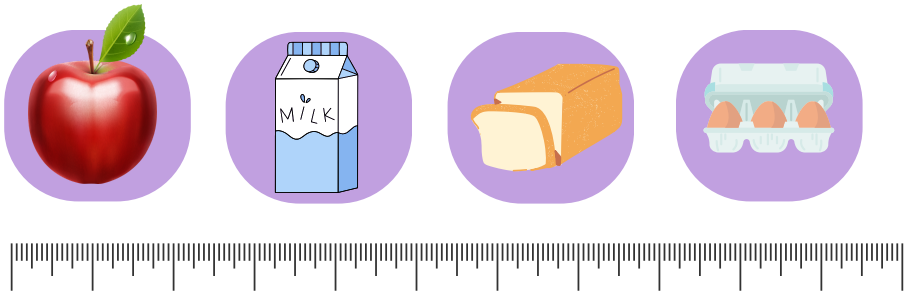
```
shopping_list.remove("bananas")
  # Output: ["apples", "milk", "bread", "eggs"]
```



4. Length of the List:

You want to know how many items are on your list:

```
number_of_items = len(shopping_list)        # Output : 4
```



how lengthy
is the list?

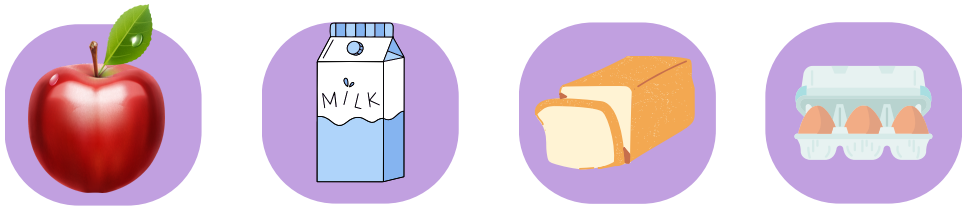
5. Index of an Element

You check where 'milk' is positioned in your shopping list:

```
index_of_milk = shopping_list.index("milk")        # Output : 1
```

where is milk?

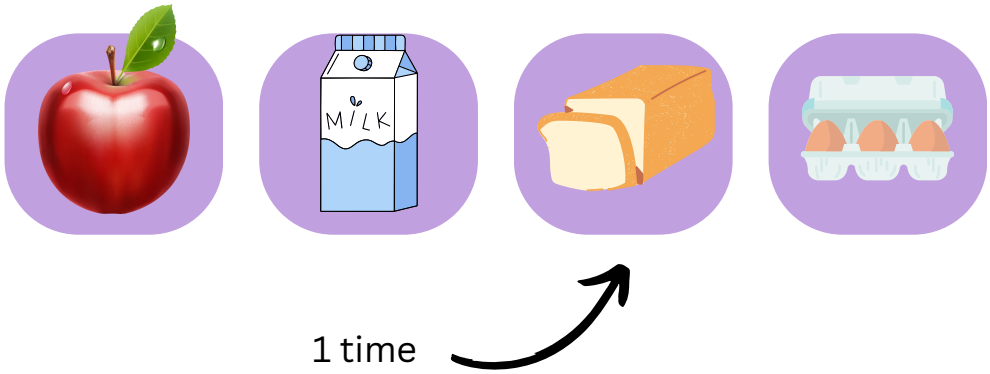
here



6. Count of an Element:

You want to know how many times 'bread' appears in your shopping list:

```
count_of_bread = shopping_list.count("bread")      #Output : 1
```

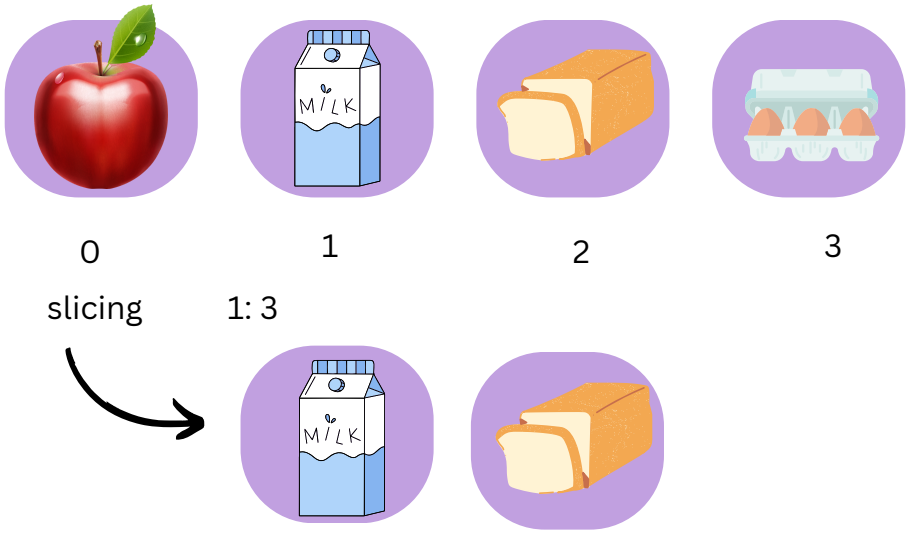


1 time

7. List Slicing:

You decide to create a sublist with items 'bread' and 'milk':

```
sliced_list = shopping_list[1:3]
    # Output: ['milk', 'bread']
```



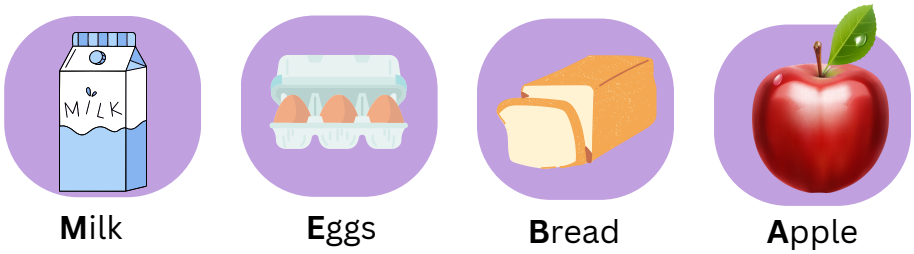
8. Sorting the List:

You decide to organize your list in alphabetical order:

```
shopping_list.sort()
  # Output: ['apples', 'bread', 'eggs', 'milk']
```

or in descending order:

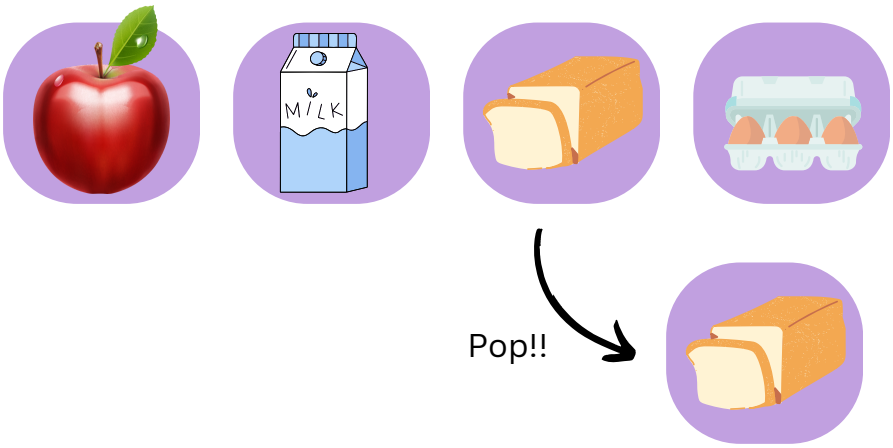
```
shopping_list.sort(reverse = True)
     # Output: ['milk', 'eggs', 'bread', 'apples']
```



9. Pop an Element:

Oops! You accidentally added 'bread' to your list. You decide to remove it:

```
popped_item = shopping_list.pop(2)
  # Output: 'bread'
```

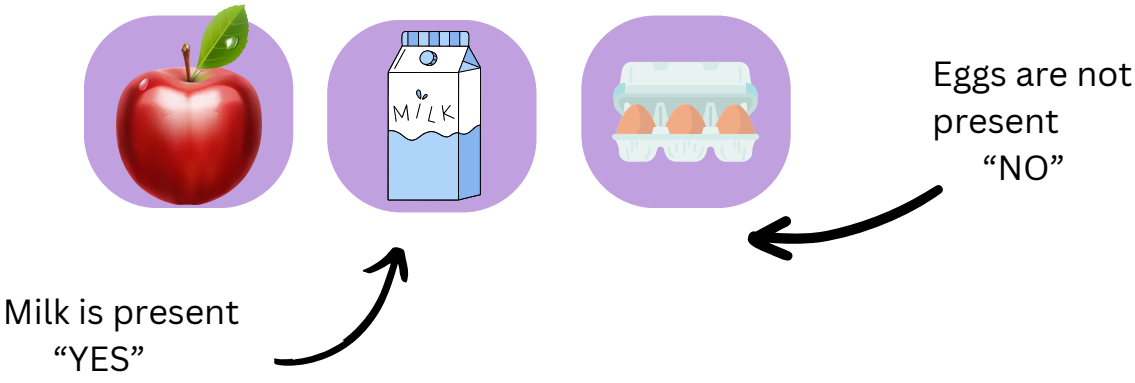


10.Checking Membership:

You check if 'milk' is present and 'eggs' are not in your shopping list:

```
# in method
is_milk_present = "milk" in shopping_list        # Output : True
```

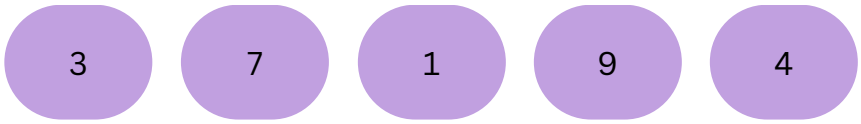
```
# not in method
is_eggs_not_present = "eggs" not in shopping_list        # Output : False
```



11.Max and Min Values:

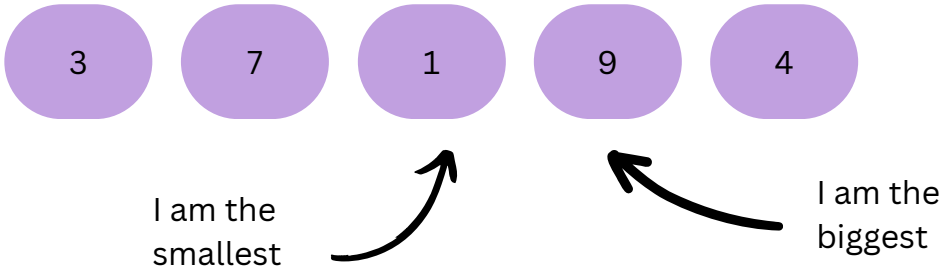
You check the maximum and minimum values in your list:

```
# Creating a new list
numbers = [3, 7, 1, 9, 4]
```




```
# Maximum value
max_value = max(numbers)          # Output : 9
```

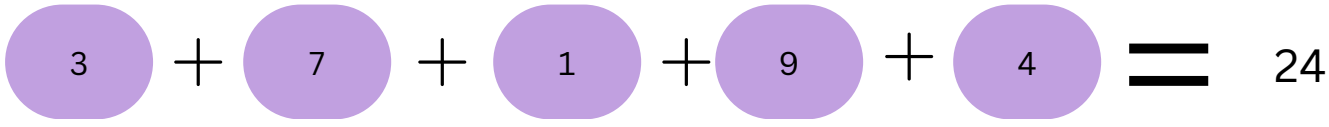
```
# minimum value
min_value = min(numbers)          # Output : 1
```

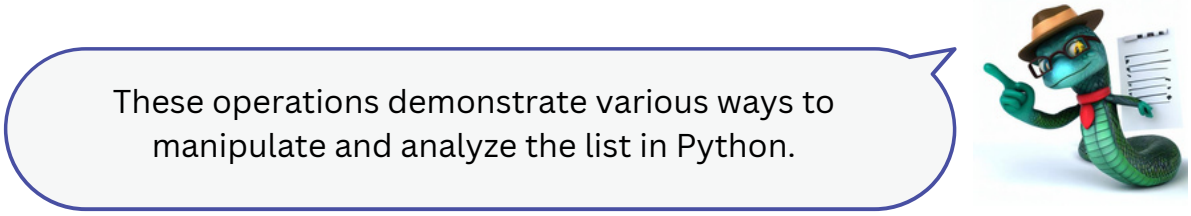


12.Sum of Elements:

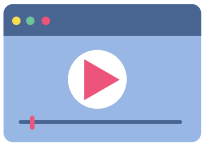
You want to know the sum of all the items on your list:

```
sum_of_elements = sum(numbers)          # Output : 24
```





For more knowledge , Click on the link below to get a video tutorial of Python List



7.2 Tuple

Tuples are similar to lists in Python but with a key distinction: tuples are immutable, meaning their elements cannot be changed after creation.

7.2. 1 Characteristics of Lists

- **Immutable:** Unlike lists, Tuple cannot be changed after creation; you cannot add, remove, or modify items.
- **Ordered:** Tuples maintain the order of items as they were inserted.
- **Indexing:** Access items by their position (index) in the Tuple.
- **Supports Different Types:** Tuple can hold different data types (e.g., strings, numbers, other lists or tuples).
- A Tuple is created using **parentheses ()** and can hold any combination of elements separated by commas.

7.2. 2 Creating a Tuple

```
my_tuple = (1, 2, 'apple', 'banana')
```

7.2. 3 Basic Tuple operations

1.Accessing Elements:

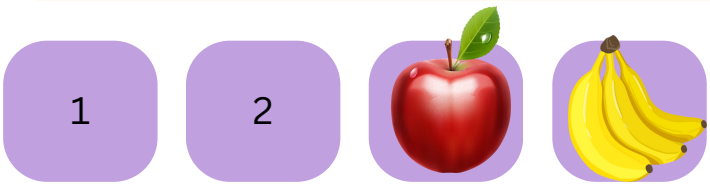
Elements in tuples are accessed using zero-based indexing, just like lists.

```
print(my_tuple[0])      # Accessing the first element: 1
```

2. Slicing:

Extracting portions of a tuple using slicing notation.

```
print(my_tuple[1:3])  # Slicing elements from index 1 to 2: (2, 'apple')
```

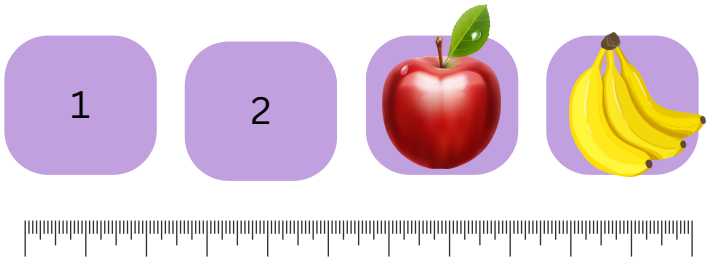


Slicing 1:3

3. Length:

Getting the length of a tuple using the len() function.

```
print(len(my_tuple))  # Getting the length of the tuple
```

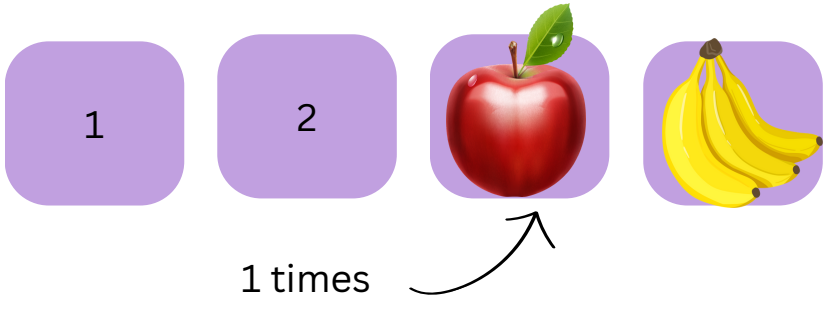


What is the length of the tuple?

4. Count:

Counting occurrences of a particular element in a tuple.

```
count_apple = my_tuple.count('apple')
    # Counting occurrences of 'apple'
```



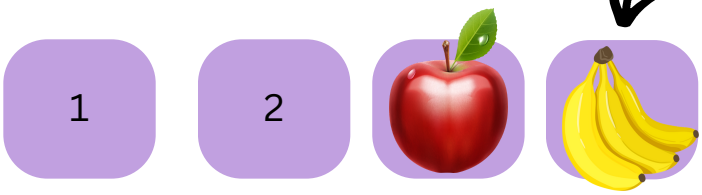
5. Index:

Finding the index of a particular element in a tuple.

```
index_banana = my_tuple.index('banana')
    # Finding the index of 'banana'
```

Where is banana?

here



6. Single Element Tuple:

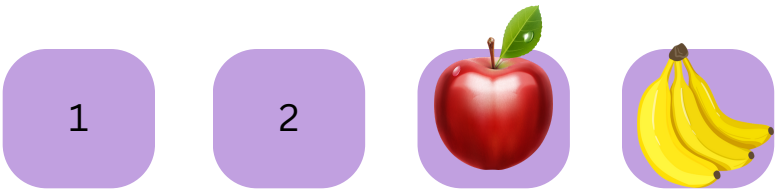
- A tuple with a single element needs a trailing comma to distinguish it from a parenthesized expression.
- Without the comma, Python might interpret the expression as just the element itself enclosed in parentheses. Adding the comma ensures that it is recognized as a tuple with a single element.

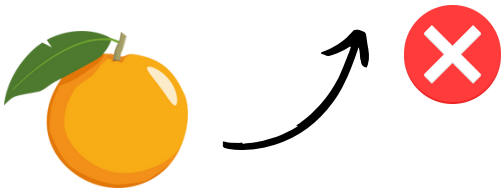
```
single_element_tuple = (5,)  # Single-element tuple
```

7. Immutable:

Once a tuple is created, its elements cannot be modified, added, or removed.

```
# Trying to modify a tuple will result in an error
my_tuple[2] = 'orange'    # This line will cause an error
```



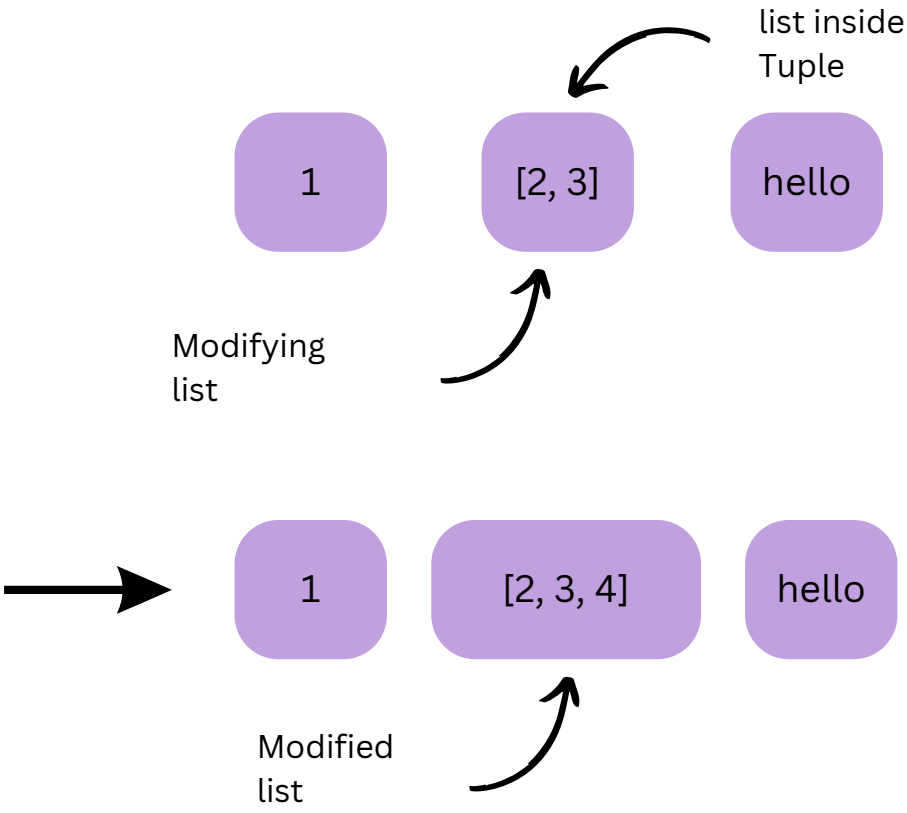


8.Immutable but Flexible:

Although tuples are immutable, they can contain mutable objects like lists as their elements. However, the contents of those mutable objects can be changed.

```
# Creating a list inside the tuple
mixed_tuple = (1, [2, 3], 'hello')
```

```
# Modifying the list inside the tuple
mixed_tuple[1].append(4)
```



8. Membership Checking:

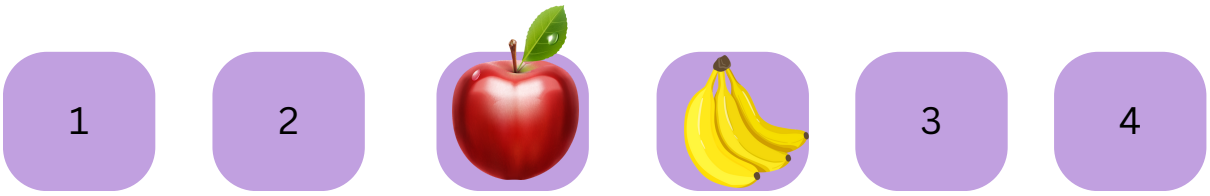
Verifying if an element exists in a tuple.

```
print('banana' in my_tuple)
    # Checking if 'banana' is in the tuple (returns True or False)
```

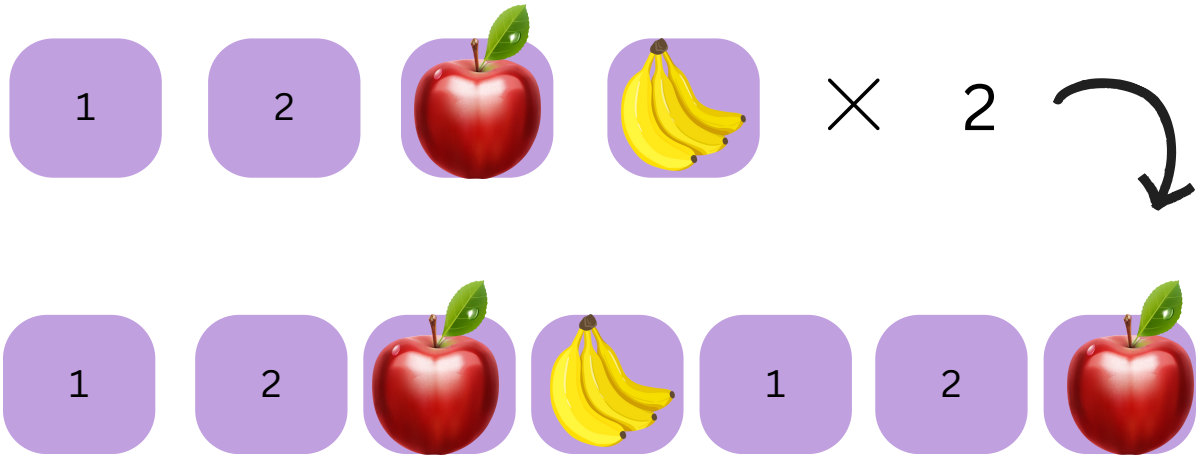
9. Concatenation and Repetition:

Creating new tuples by combining or repeating existing tuples.

```
combined_tuple = my_tuple + (3, 4)      # Concatenating tuples
```

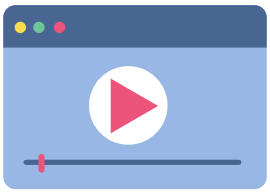



```
repeated_tuple = my_tuple * 2      # Repeating the tuple
```



These operations allow manipulation and exploration of tuples without modifying the original tuple, as tuples are immutable data structures in Python.

For more knowledge , Click on the link below to get a video tutorial of Python Tuple



7.3 Sets

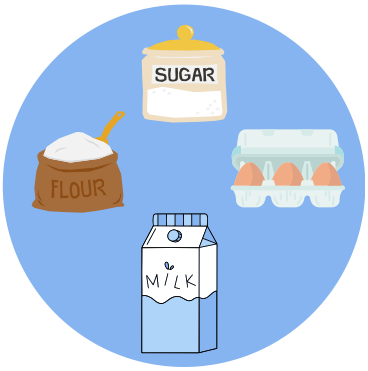
set is a collection of unique elements with no duplicates Sets are useful when dealing with unique collections of items, like unique IDs, unique values in data, or removing duplicates from a list.

7.3. 1 Characteristics of Sets

- **Uniqueness:** Sets only contain unique elements. If you try to add a duplicate, it won't be added.
- **No Indexing:** Sets are unordered, so you can't access items by an index because they don't have a specific order.
- **Mutable:** You can add or remove elements from a set after it's created.
- **Mathematical Set Operations:** Sets in Python support various mathematical operations like union, intersection, difference, etc.

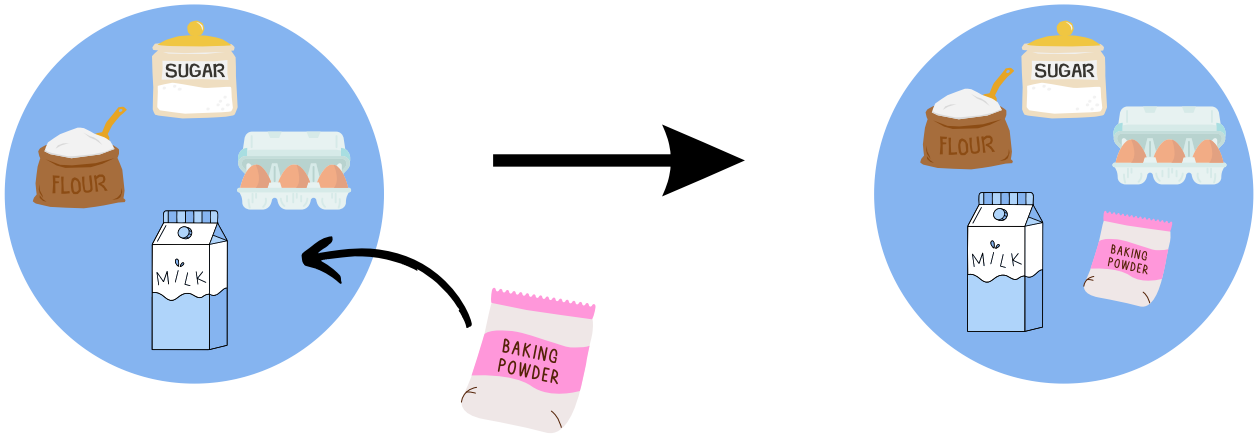
7.3. 2 Creating Sets

```
unique_ingredients = {"flour", "sugar", "eggs", "milk"}
```



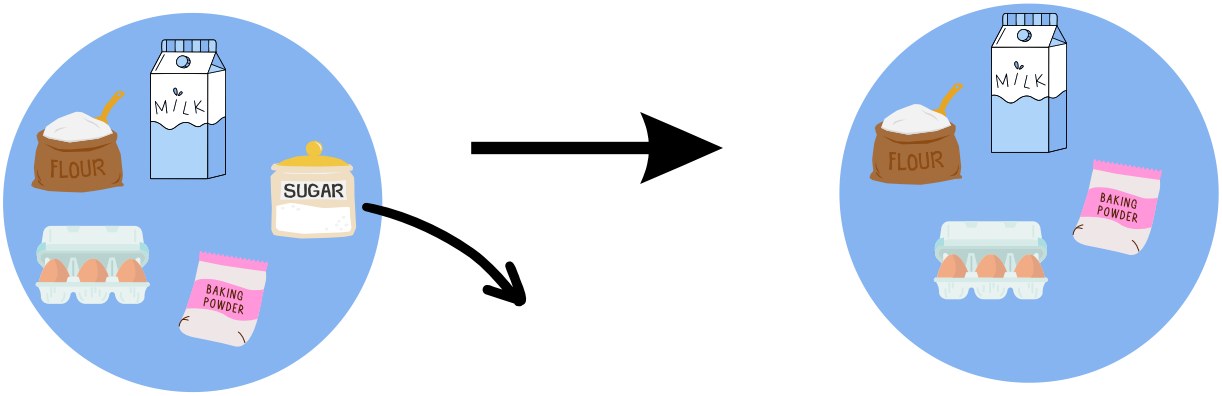
Adding elements to a set:

```
unique_ingredients.add("baking powder")
```



Removing elements from a set:

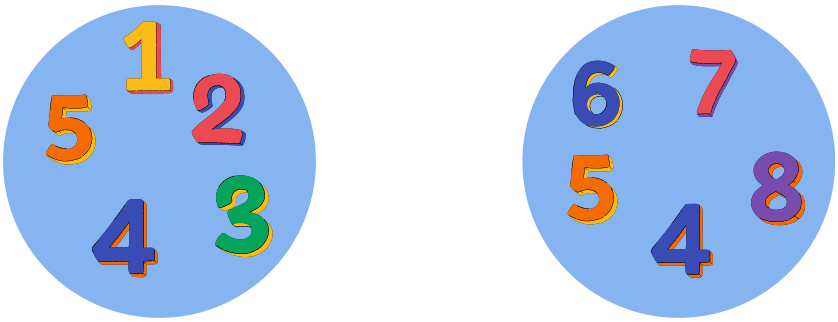
```
unique_ingredients.remove("sugar")
```



7.3. 2 Basic set operations

Let's define two sets to illustrate these operations

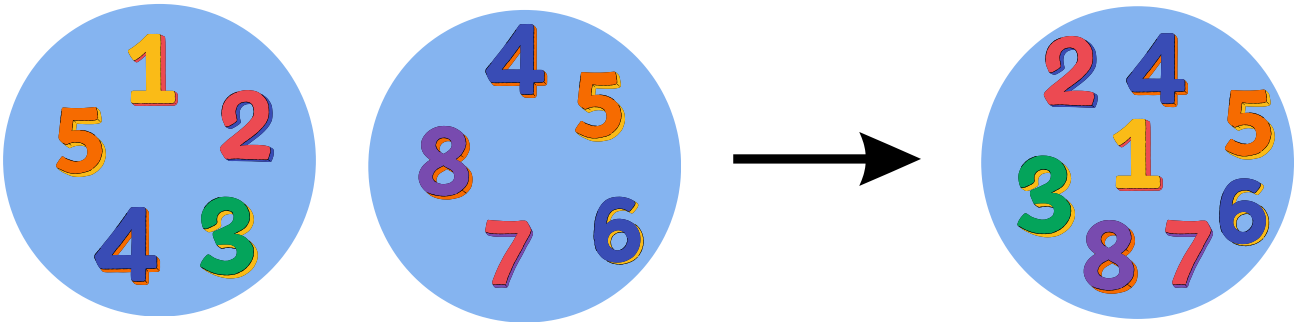
```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
```



1.Union (|):

Combines unique elements from both sets.

```
union_set = set1 | set2          # Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

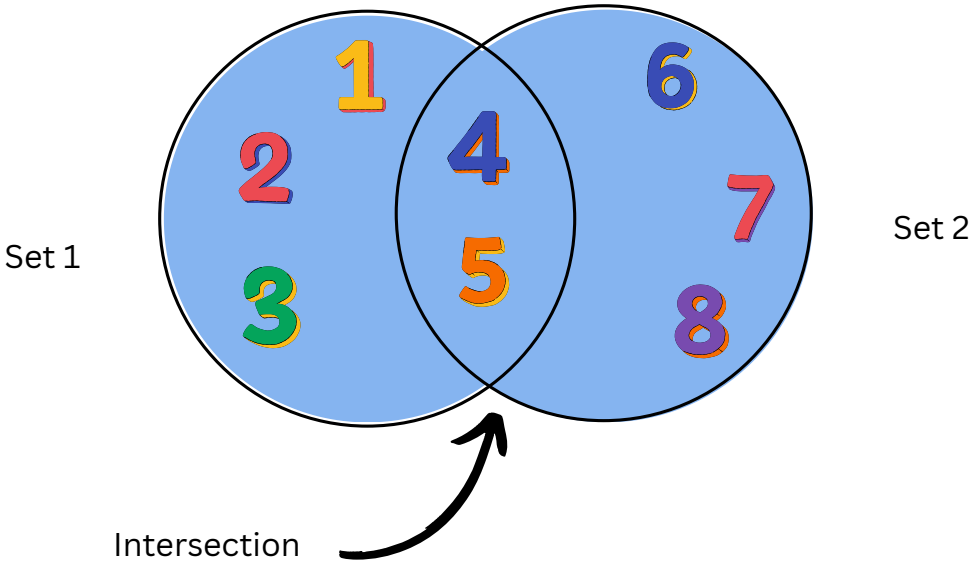


Set 1 Set 2

2.Intersection (&):

Finds common elements between sets.

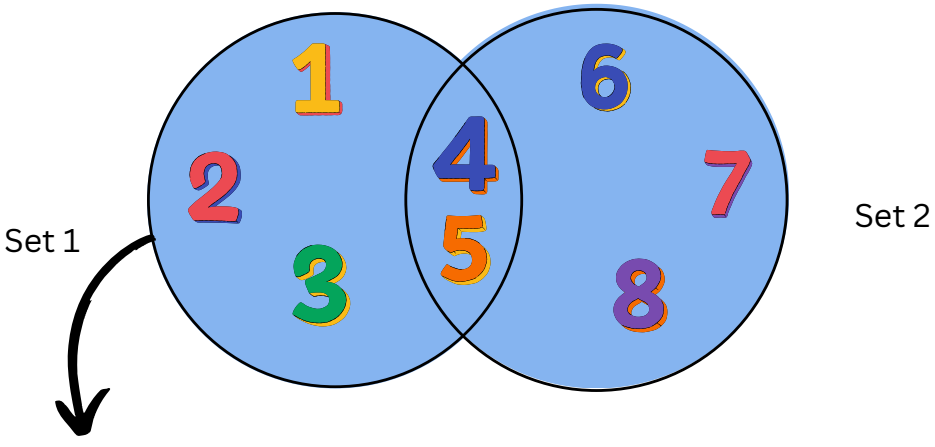
```
intersection_set = set1 & set2          # Output: {4, 5}
```



3.Difference (-):

Finds elements in the first set but not in the second.

```
difference_set = set1 - set2          # Output: {1, 2, 3}
```

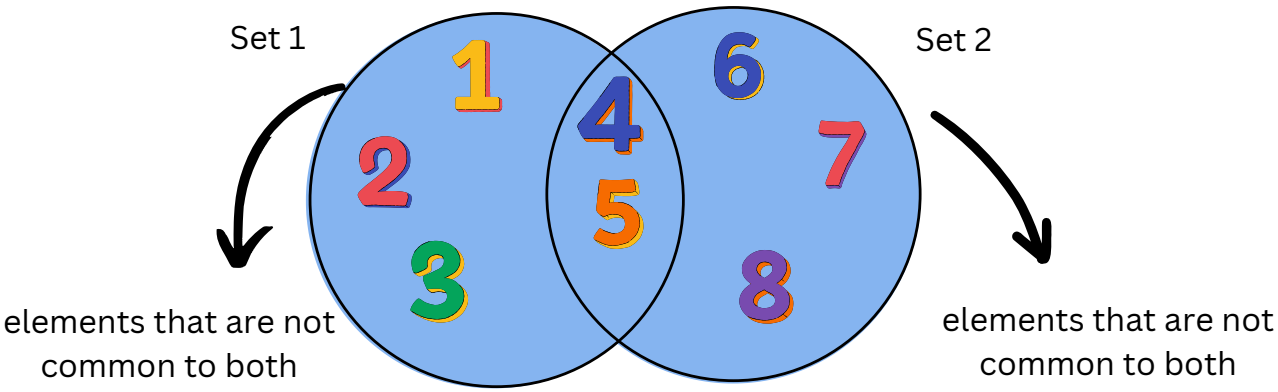


1, 2 , 3 are not present in set 2

4.Symmetric Difference (^):

Finds elements in either set, but not in both.

```
symmetric_difference_set = set1 ^ set2      # Output: {1, 2, 3, 6, 7, 8}
```



5.Subset and Superset

- **Subset :**

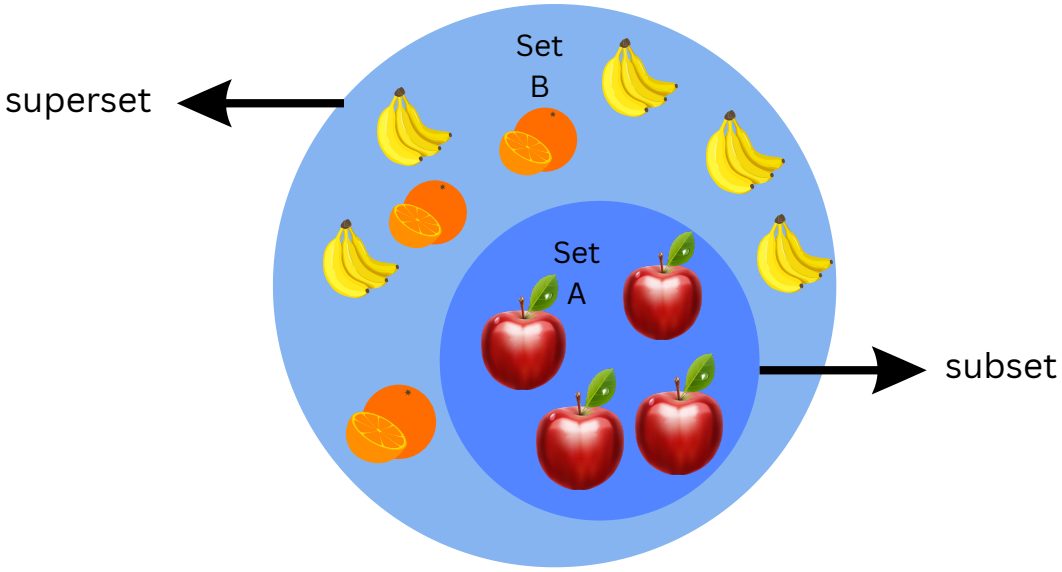
A set A is a subset of set B if every element of A is also an element of B.

Example: Imagine a set of fruits {apple, banana, orange}. If you have a set {apple}, it is a subset of the original set because every element in {apple} is also in {apple, banana, orange}.

- **Superset:**

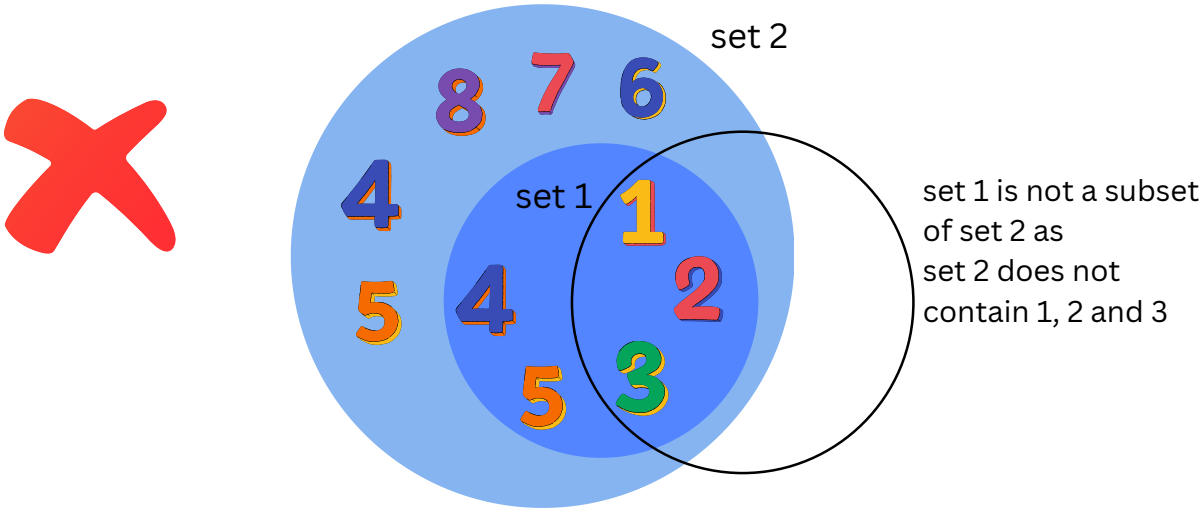
A set B is a superset of set A if every element of A is also an element of B.

Example: Using the same set of fruits {apple, banana, orange}. If you have a set {apple, banana, orange}, it is a superset of the original set because every element in {apple} is also in {apple, banana, orange}.As Apple is a fruit



a) Subset (<=):
Checks if one set is a subset of another.

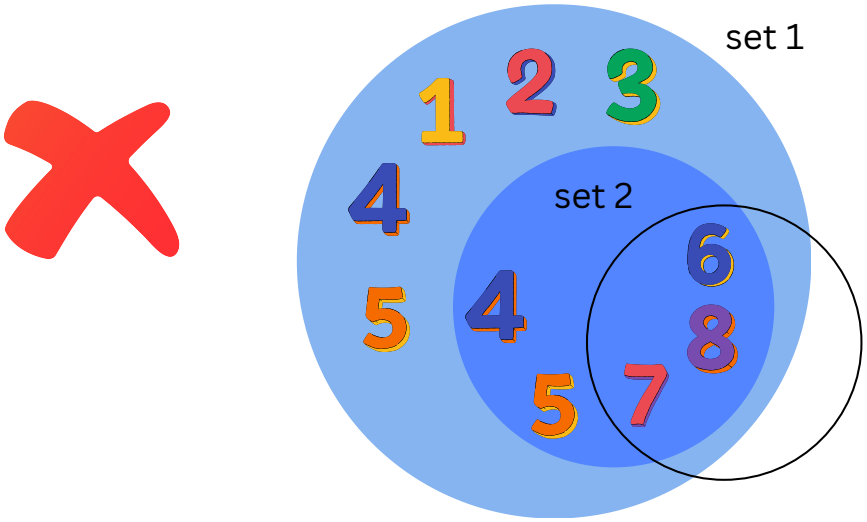
```
is_subset = set1 <= set2
    # Output: False (set1 is not a subset of set2)
```



set 1 is not a subset of set 2 as set 2 does not contain 1, 2 and 3

6.Superset (>=):
Checks if one set is a superset of another.

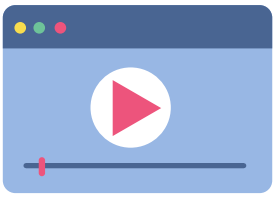
```
is_superset = set1 >= set2
    # Output: False (set1 is not a superset of set2)
```



set 1 is not a superset of set 2 as set 1 does not contain 6,7 and 8

These operations help manipulate and compare sets based on their elements, making it easy to perform set theory operations and comparisons efficiently in Python.

For more knowledge , Click on the link below to get a video tutorial of Python Sets



7.4 Dictionary

A dictionary in Python is a data structure that stores a collection of data as key-value pairs. It's similar to how words and their meanings are organized in a real-world dictionary.

7.4.1 Structure of a Dictionary:

Key

Unique identifiers within the dictionary, like words in a real dictionary.

Value

Corresponding to each key, these hold the associated information or data.

In Python, dictionaries are defined using **curly braces {}** and consist of **key-value pairs** separated by commas and a **colon (:)** between the key and value.

7.4.2 Characteristics of a Dictionary:

- **Unordered Collection:** The items in a dictionary are not stored in any specific order.
- **Mutable::** Dictionaries can be modified after creation. You can add, remove, or modify key-value pairs.
- **Key Uniqueness:** Every key in a dictionary must be unique. If you try to add a key that already exists, it will overwrite the existing value.

7.4.3 Creating a Dictionary:

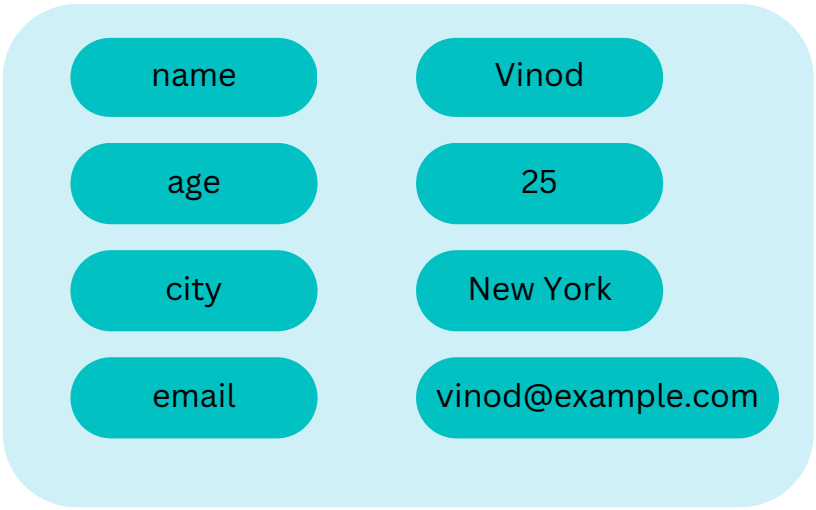
1.Creating an empty dictionary

```
empty_dict = {}
print(empyt_dict)
```

2. Creating a dictionary with values

```
person = {
            "name": "Vinod",
             "age": 25,
             "city": "New York",
             "email": "vinod@example.com"
 }
```

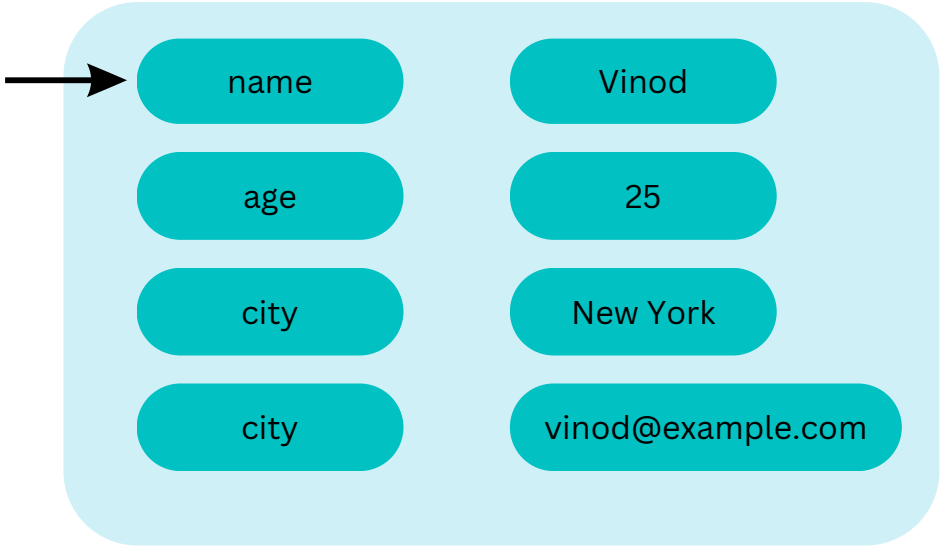
person =



3.Accessing Values:

```
name = person["name"]
       # Accessing the value associated with the key "name"
```

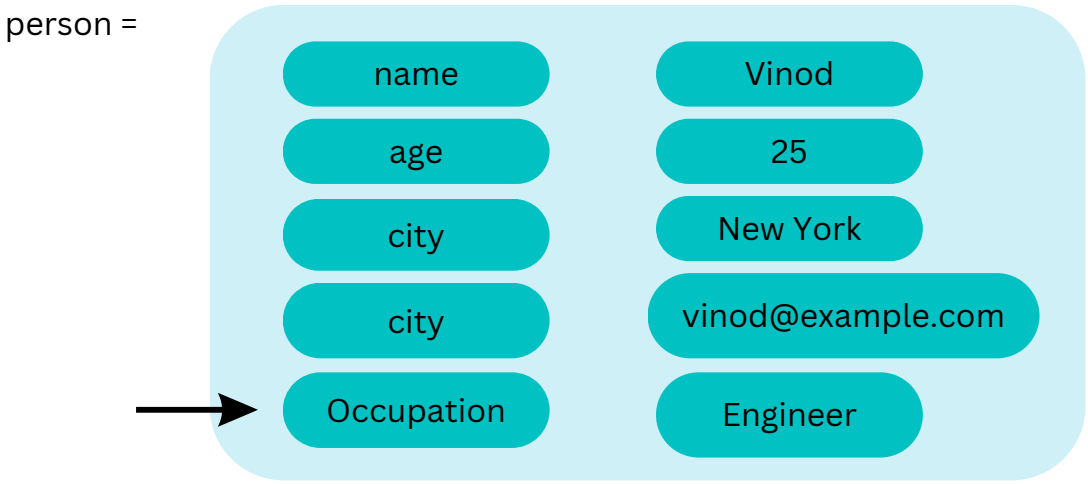
person =



→ Vinod

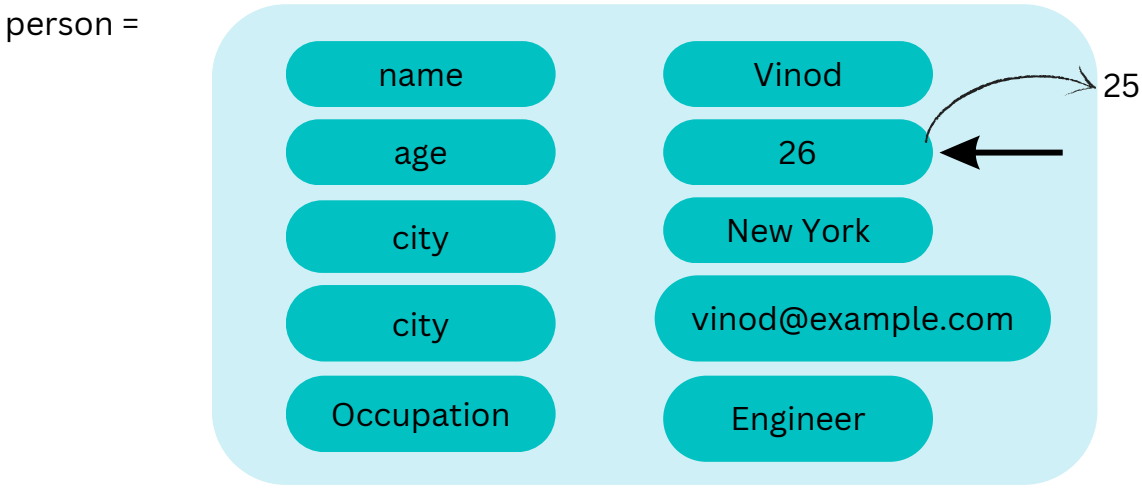
4. Adding Values:

```
person["occupation"] = "Engineer"
    # Adding a new key-value pair
```



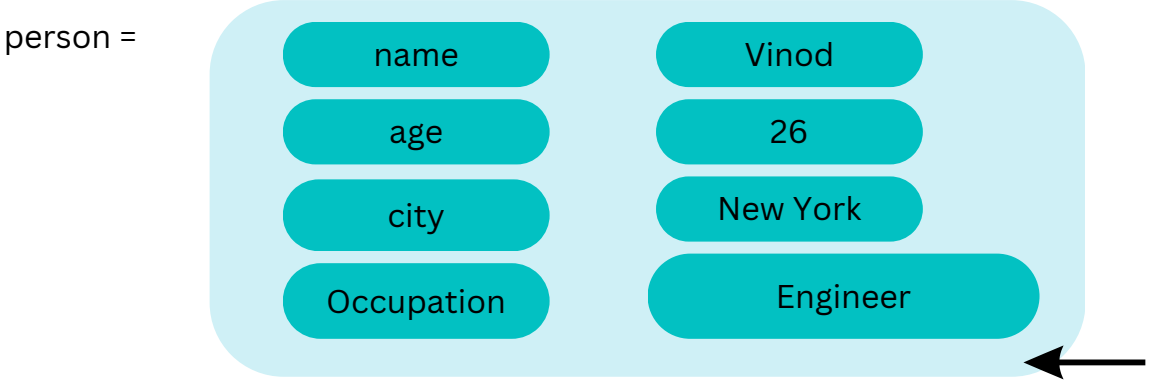
5. Updating Values

```
person["age"] = 26
    # Updating the value associated with the key "age"
```



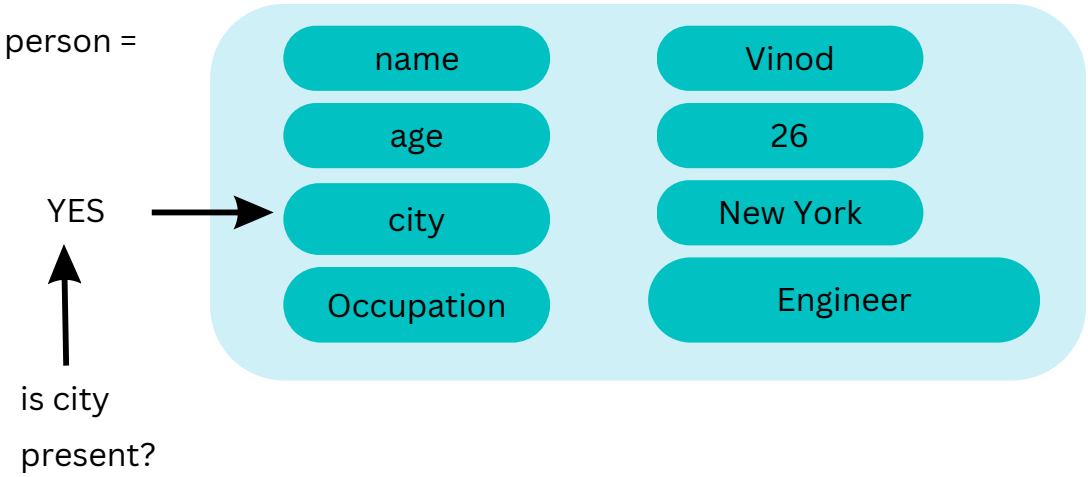
6. Removing Items:

```
del person["email"]
    # Removing the key-value pair with the key "email"
```



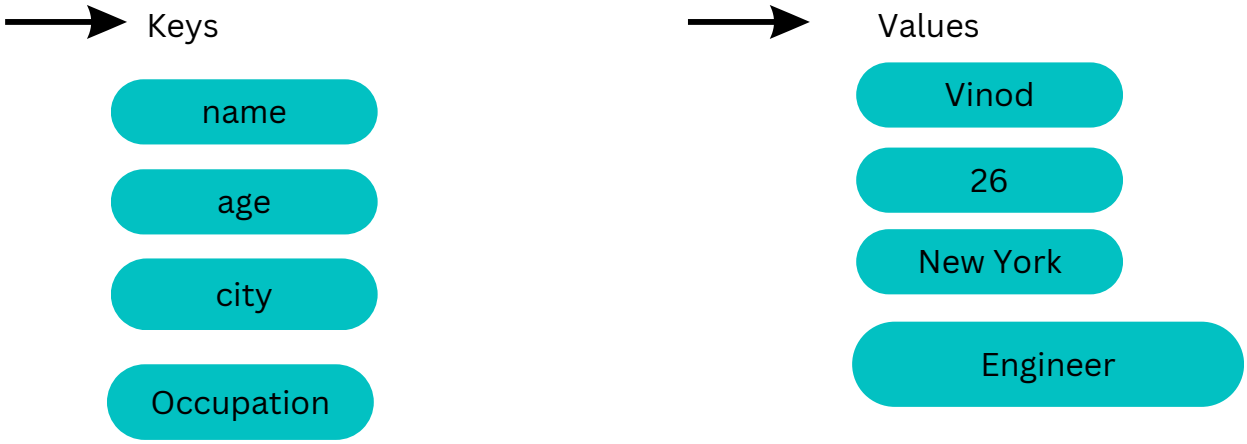
7. Checking Key Existence:

```
is_city_present = "city" in person
    # Checking if the key "city" is present in the dictionary
```



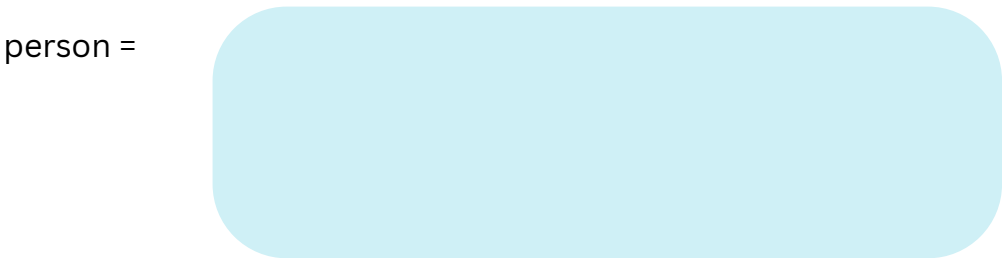
8. Getting Keys or Values:

```
keys = person.keys()        # Getting all keys in the dictionary
values = person.values()    # Getting all values in the dictionary
```



9. Clearing a Dictionary:

```
person.clear()        # Removes all items from the dictionary
```



10. Dictionary Copy:

```
new_person = person.copy()      # Creating a copy of the dictionary
```

person =

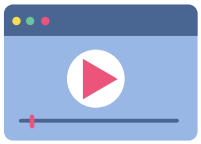
name	Vinod
age	25
city	New York
city	vinod@example.com
Occupation	Engineer

new_person =

name	Vinod
age	25
city	New York
city	vinod@example.com
Occupation	Engineer

These operations help manipulate, access, and manage dictionaries in Python, making it easy to add, remove, update, and retrieve data associated with specific keys.

For more knowledge , Click on the link below to get a video tutorial of Python Dictionary



Chapter 8 : Functions

A function is like a mini-program inside your program. It's a block of code that does a specific task. You can use functions to organize your code and avoid repeating the same code in multiple places.

Think of it this way

Let's consider a simple daily work example involving making a cup of tea. We'll use a function called **MakeTea** to encapsulate the steps involved:

```
# The function to make a cup of  green tea

def MakeTea():
    print("1. Boil water.")
    print("2. Place a green tea bag in a cup.")
    print("3. Pour hot water into the cup.")
    print("4. Let it steep for a few minutes.")
    print("5. Remove the tea bag and enjoy your green tea!")

# Using the function to make tea
MakeTea()
```

In this example:

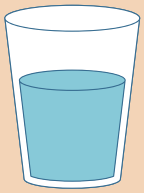
- **MakeTea** is like a set of instructions (a function) that performs the steps needed to make tea.
- When you want to make tea, you don't need to remember each step; you just call the **MakeTea** function.
- The function organizes the steps, avoids repetition, and makes the process more straightforward.
- If you decide to make another cup of tea, you can reuse the **MakeTea** function without rewriting the steps

So, just like a function can simplify making tea, it can do the same for various tasks in your daily work, making your code more organized and efficient.

let's create a Python function that simulates a coffee making machine. also include parameters for the amount of coffee, sugar, and milk needed. We'll also add steps to add sugar and milk to the coffee:

COFFEE  MACHINE

STEP 1



Add water to the coffee machine

STEP 2

Add coffee powder to the machine

STEP 3

Add sugar to the coffee

STEP 4

Add milk to the coffee

Parameters ⟶

2 spoons

1 spoon

50 ml

STEP 5

Brew the coffee

STEP 6

Pour the coffee into a cup

STEP 7

Serve the coffee

To enjoy your coffee instantly, simply flick the switch on the coffee machine, and your ready-made cup will be served, skipping the hassle of going through all the steps.

let's encapsulate the functionality of the coffee machine into a Python function block:

```
def make_coffee(amount_coffee, amount_sugar, amount_milk):
    # Step 1: Add water to the coffee machine
    print("Step 1: Adding water to the coffee machine")

    # Step 2: Add coffee powder to the machine
    print(f"Step 2: Adding {amount_coffee} spoons of coffee powder to the machine")

    # Step 3: Add sugar to the coffee
    print(f"Step 3: Adding {amount_sugar} spoons of sugar to the coffee")

    # Step 4: Add milk to the coffee
    print(f"Step 4: Adding {amount_milk} ml of milk to the coffee")

    # Step 5: Brew the coffee
    print("Step 5: Brewing the coffee")

    # Step 6: Pour the coffee into a cup
    print("Step 6: Pouring the coffee into a cup")

    # Step 7: Serve the coffee
    print("Step 7: Serving the coffee")

# Call the function with specified amounts of coffee, sugar, and milk
make_coffee(amount_coffee=2, amount_sugar=1, amount_milk=50)
```

- Step 1: Adding water to the coffee machine Step
- 2: Adding 2 spoons of coffee powder to the machine Step
- 3: Adding 1 spoons of sugar to the coffee Step
- 4: Adding 50 ml of milk to the coffee Step
- 5: Brewing the coffee Step
- 6: Pouring the coffee into a cup Step
- 7: Serving the coffee

In this example:

- We've added three parameters (**amount_coffee, amount_sugar, amount_milk)** to the function to specify the amount of **coffee, sugar, and milk** needed.
- Inside the function, we print the amount of **coffee, sugar, and milk** being added to the coffee machine.
- Step 3: We add the **specified amount of sugar** to the coffee.
- Step 4: We add the **specified amount of milk** to the coffee.
- The function then proceeds with the brewing process, pouring the coffee into a cup, and serving it.

8.1 How to Create a Function:

1.Define the Function:

Choose a name for your function and define it using def.

```
def greet():
```

2. Add Code Inside:

Write the code that you want the function to execute. This can be anything you want!

```
def greet():
    print("Hello, welcome!")
```

3. Call the Function:

To run the code inside the function, you need to "call" or "invoke" the function.

```
def greet():
    print("Hello, welcome!")

greet()                # Call the function
```

4. Adding Parameters

You can make your function more flexible by adding parameters. Parameters are like placeholders that you can fill in when you call the function.

```
# Define a function with a parameter
def greet(name):
    print("Hello, " + name + "! Welcome!")

# Call the function with an argument
greet("Sahaj")
```

5. Returning Values

A function can also return a value. This can be useful when you want the function to give you back some information.

```
  # Define a function that returns a value
def add_numbers(x, y):
  result = x + y
  return result

# Call the function and store the result
sum_result = add_numbers(3, 4)
print("Sum:", sum_result)
```

6. Putting It All Together:

```
  # Define a function
def greet(name):
    message = "Hello, " + name + "! Welcome!"
    return message

 # Call the function and store the result
greeting_message = greet("Vinie")

 # Print the result
print(greeting_message)
```

8.2 Types of Functions

functions can be categorized into several types based on their characteristics and purposes. Here are some common types of functions:

8.2.1 Built-in Functions:

These are functions that are already available in Python and can be directly used without the need for additional definitions. Examples: **print(), len(), max(),** and **min(), etc.**

```
print("Hello, Guvi!")      # Example of a built-in function
```

8.2.2 User-Defined Functions:

These are functions created by the programmer to perform specific tasks. They are defined using the **def** keyword. User-defined functions enhance code modularity and reusability.

```
def greet(name):
    print("Hello, " + name + "!")

greet("Sahaj")  # Example of a built-in function
```

8.2.3 Recursive Functions:

A function that calls itself during its execution is called a recursive function. Recursive functions are used for tasks that can be broken down into simpler sub-problems.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)          # Example of a recursive function
```

8.2.4 Lambda Functions (Anonymous Functions):

Lambda functions are concise, one-line functions defined using the lambda keyword. They are often used for short-term tasks and can be assigned to variables.

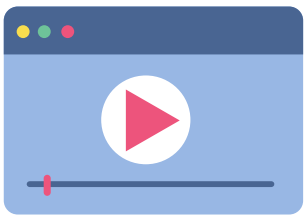
```
square = lambda x: x**2          # Example of a lambda function
result = square(4)
```

8.2.5 Generator Functions:

Generator functions use the **yield** keyword to produce a sequence of values over time. They are memory-efficient and are often used with iterators.

```
def count_up_to(limit):                    # Example of a generator function
    count = 1
    while count <= limit:
        yield count
        count += 1

numbers = count_up_to(5)    # Iterate over the generator and print each value
for number in numbers:
    print(number)
```

For more knowledge , Click on the link below to get a video tutorial of Python Function



Chapter 9 : Map and Filters

9.1 Map

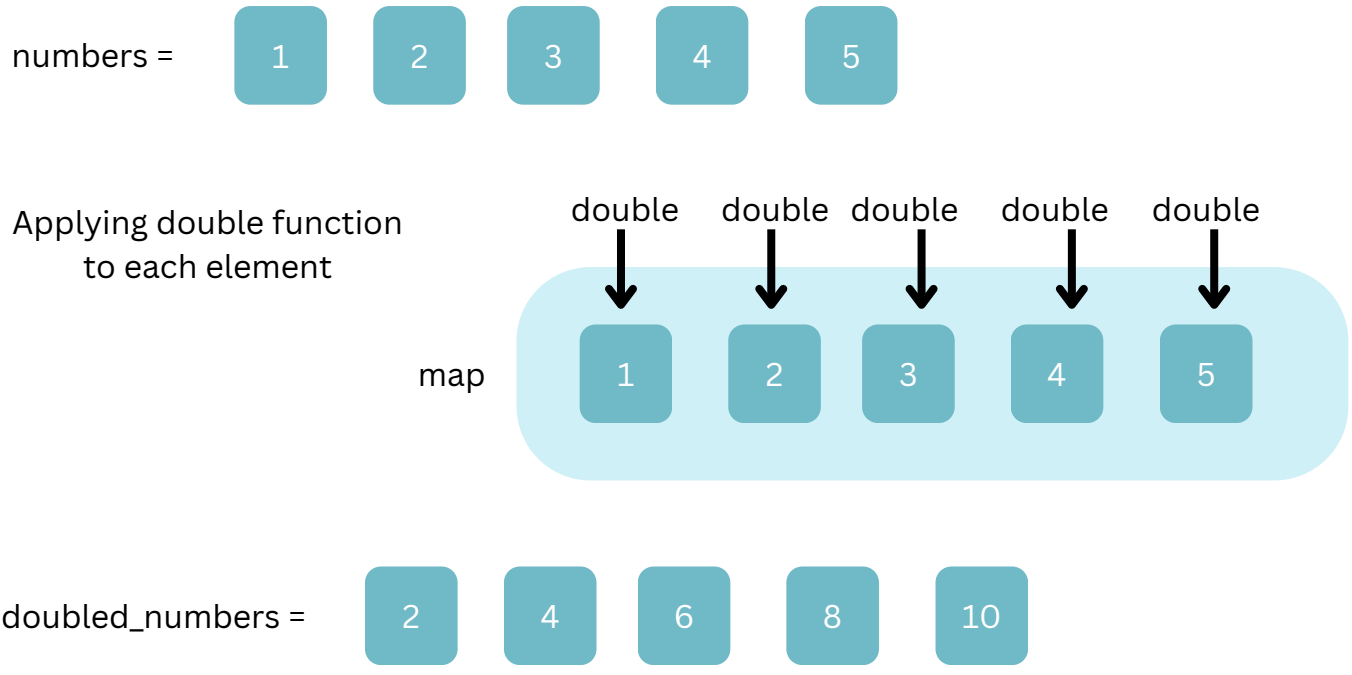
The map() function in Python is used to apply a specified function to each item in an iterable (like lists, tuples, etc.) and returns a map object (an iterator) containing the results.

```
# Syntax:
map(function, iterable)
```

The function to be applied to each element of the iterable.

The iterable (list, tuple, etc.) whose elements will be processed by the function.

```
# Doubling each element in a list using map
numbers = [1, 2, 3, 4, 5]

def double(x):        # creating a function which will multiply the number with 2
    return x * 2

doubled_numbers = map(double, numbers)     # applying the map function
print(list(doubled_numbers))               # Output: [2, 4, 6, 8, 10
```



9.2. Filter

filter in Python is like a sieve that allows you to separate or extract specific elements from a collection based on a condition or criteria you define. It helps you narrow down a set of data to only include the items that meet certain requirements.

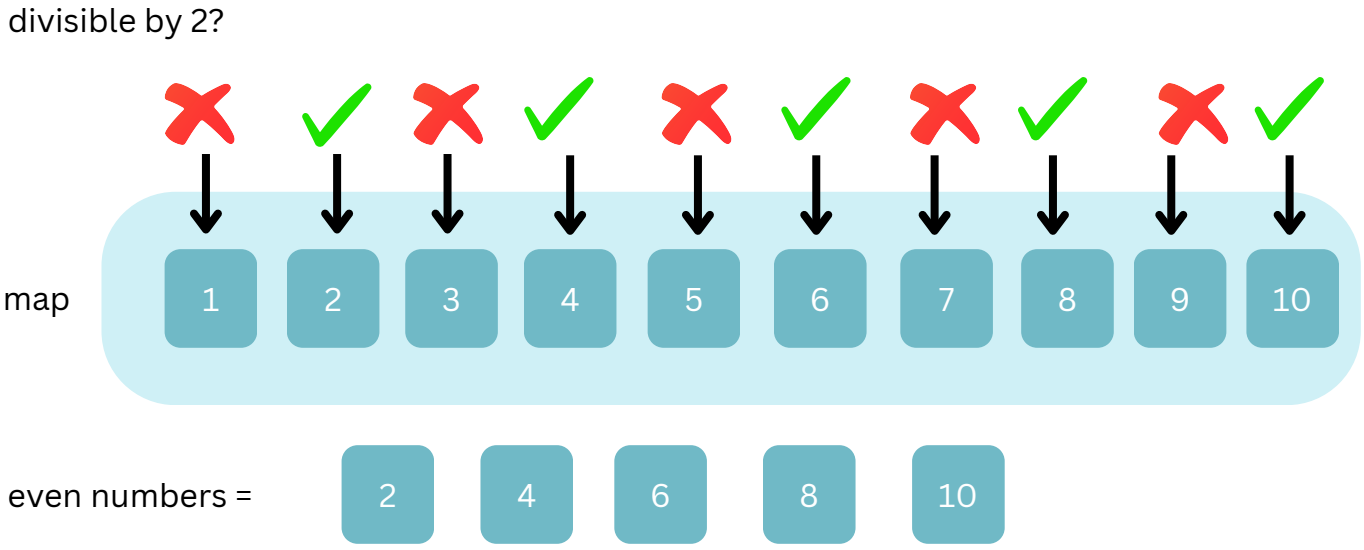
```
# Syntax:
filter(function, iterable)
```

The rule or criterion used to decide which elements to include or exclude from the collection.

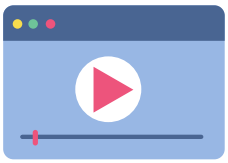
The iterable (list, tuple, etc.) whose elements you want to filter out.

```
# Define a list of numbers from 1 to 10
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# The lambda function checks if a number is even (divisible by 2)
# The filter() function returns an iterator containing only the elements from the
 original list that satisfy the condition

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)      # Print the list of even numbers
```

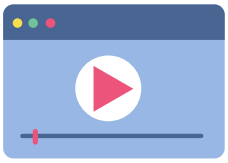
Applying filter function to each element
here, we are filtering out elements which are divisible by 2



For more knowledge , Click on the link below to get a video tutorial of Python Map



For more knowledge , Click on the link below to get a video tutorial of Python Filters

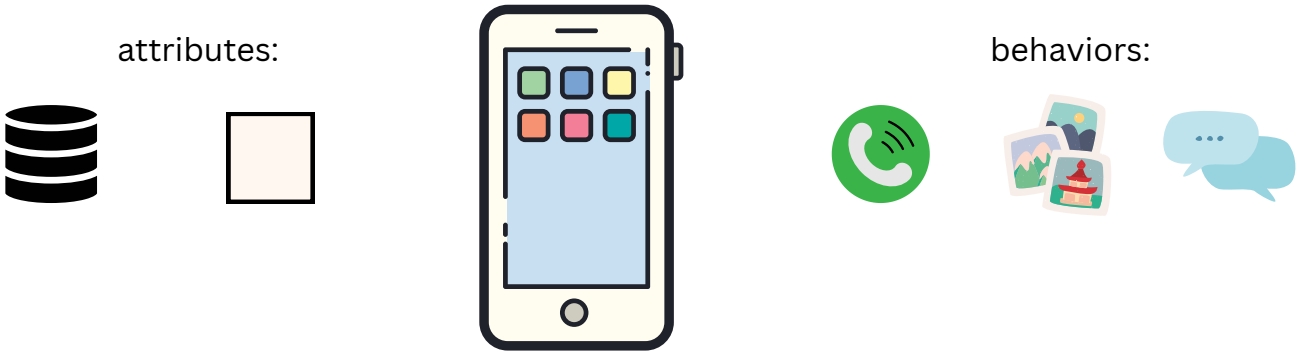


Chapter 10 : Object Oriented Programming (OOPs)

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of "objects," which are instances of classes. It's a way of structuring code to organize and manage complex systems by modeling them after real-world objects and interactions.

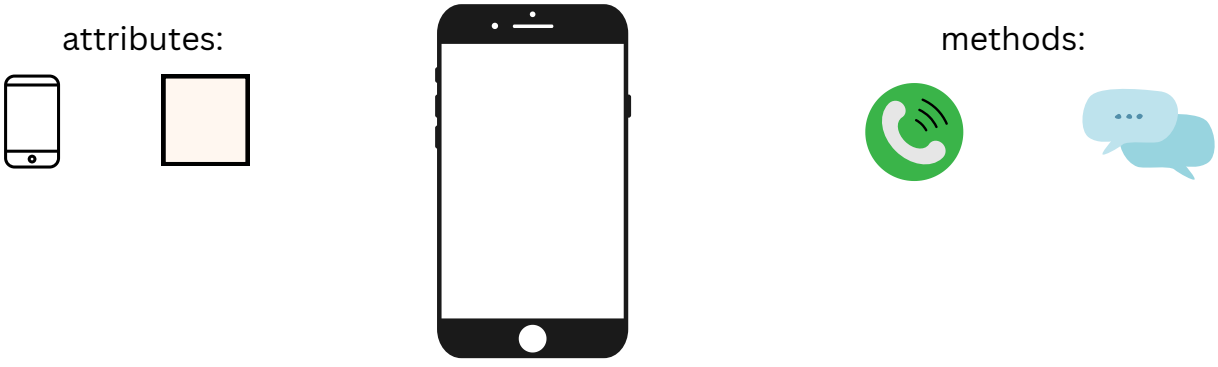
10.1 Objects:

- Objects are the fundamental building blocks of OOP. They represent concrete entities in the program, each having its own unique characteristics (attributes) and behavior (methods).
- Mobile Phone: An object represents a specific mobile phone, such as an iPhone or Samsung Galaxy. Each mobile phone has its own unique attributes (color, model, storage capacity) and behaviors (making calls, sending messages, taking photos).



10.2 Classes:

- Classes are blueprints or templates for creating objects. They define the structure and behavior of objects by specifying attributes and methods. Objects are instances of classes, created based on these blueprints.
- Mobile Phone Class: A class is like a blueprint for creating mobile phones. It defines the structure and behavior of mobile phones by specifying attributes (color, model) and methods (make_call(), send_message()) that all mobile phones should have.



Let's illustrate classes and objects using a Phone example:

```
# Define the Phone class
class Phone:
    # Constructor to initialize attributes
    def __init__(self, brand, model, color):
        self.brand = brand
        self.model = model
        self.color = color

    # Method to make a call
    def make_call(self, number):
        print(f"Making a call from {self.brand} {self.model} to {number}...")

    # Method to send a message
    def send_message(self, number, message):
        print(f"Sending a message from {self.brand} {self.model} to {number}: {message}")

# Creating phone objects
phone1 = Phone("Samsung", "Galaxy S21", "Black")
phone2 = Phone("Apple", "iPhone 12", "White")

# Using phone methods
phone1.make_call("1234567890")
phone2.send_message("9876543210", "Hello!")

# Accessing phone attributes
print(f"{phone1.brand} {phone1.model} - Color: {phone1.color}")
print(f"{phone2.brand} {phone2.model} - Color: {phone2.color}")
```

```
# Output
Making a call from Samsung Galaxy S21 to 1234567890...
Sending a message from Apple iPhone 12 to 9876543210: Hello!
Samsung Galaxy S21 - Color: Black
Apple iPhone 12 - Color: White
```

In this example:

- We define a **Phone** class with **attributes** (brand, model, and color), and **methods** (make_call and send_message).
- We create two **phone objects** phone1 and phone2 representing a Samsung Galaxy S21 and an iPhone 12.
- We use the **methods** of the phone objects to **make calls** and **send messages.**
- We access the **attributes** of the phone objects to display their **brand, model, and color**.

Constructor:

- A constructor is a special method in a class that gets automatically called when you create an object (instance) of that class.
- It initializes (sets up) the object's attributes or performs any necessary setup tasks.

init method:

- In Python, the constructor method is named __init__.
- It stands for "initialize" because its main purpose is to initialize the object's attributes.

Instances:

- instances are like the real objects created based on those blueprints. Each instance has its own set of attributes and can perform actions defined by the class.
- Instances allow you to work with specific, tangible objects in your code.

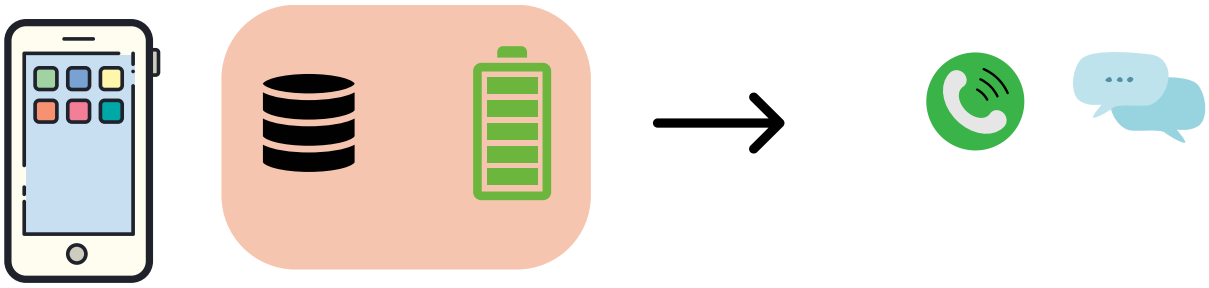
self:

- self is a reference to the current instance of the class.
- It allows you to access the object's attributes and methods within the class.

When you create an object of a class, Python automatically calls the __init__ method of that class. This method takes at least one argument, self, which refers to the newly created object. Inside the __init__ method, you can set up the initial state of the object by assigning values to its attributes.

10.3 Encapsulation:

- Encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on the data within a single unit (a class). It hides the internal details of how an object works from the outside world, allowing for better control and security.
- **Encapsulation in Mobile Phone:** Encapsulation bundles data and methods related to the mobile phone within the Mobile Phone class. For example, the mobile phone's storage capacity and battery life are encapsulated within the class, and methods like make_call() and send_message() operate on this data.



```
# Define the Phone class
class MobilePhone:
    def __init__(self, color, model):
        self._color = color        # Encapsulated attribute
        self._model = model      # Encapsulated attribute
        self.__imei = '1234567890'            # Private encapsulated attribute

    def make_call(self, number):
        print(f"Making a call to {number} with {self._color} {self._model}...")

    def get_imei(self):
        return self.__imei # Encapsulated method to access private attribute


# Creating an object of the MobilePhone class
iphone = MobilePhone("black", "iPhone 12")

# Accessing encapsulated attributes directly (not recommended)
print(f"Color: {iphone._color}, Model: {iphone._model}")

# Calling an encapsulated method to access encapsulated data
print(f"IMEI: {iphone.get_imei()}")
```

```
#Output:
Color: black, Model: iPhone 12
IMEI: 1234567890
```

```
# Trying to access a private encapsulated attribute (will raise an error)
print(iphone.__imei)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-6-119a179e1b9c> in <cell line: 2>()
      1 # Trying to access a private encapsulated attribute (will raise an error)
----> 2 print(iphone.__imei)
AttributeError: 'MobilePhone' object has no attribute '__imei'
```

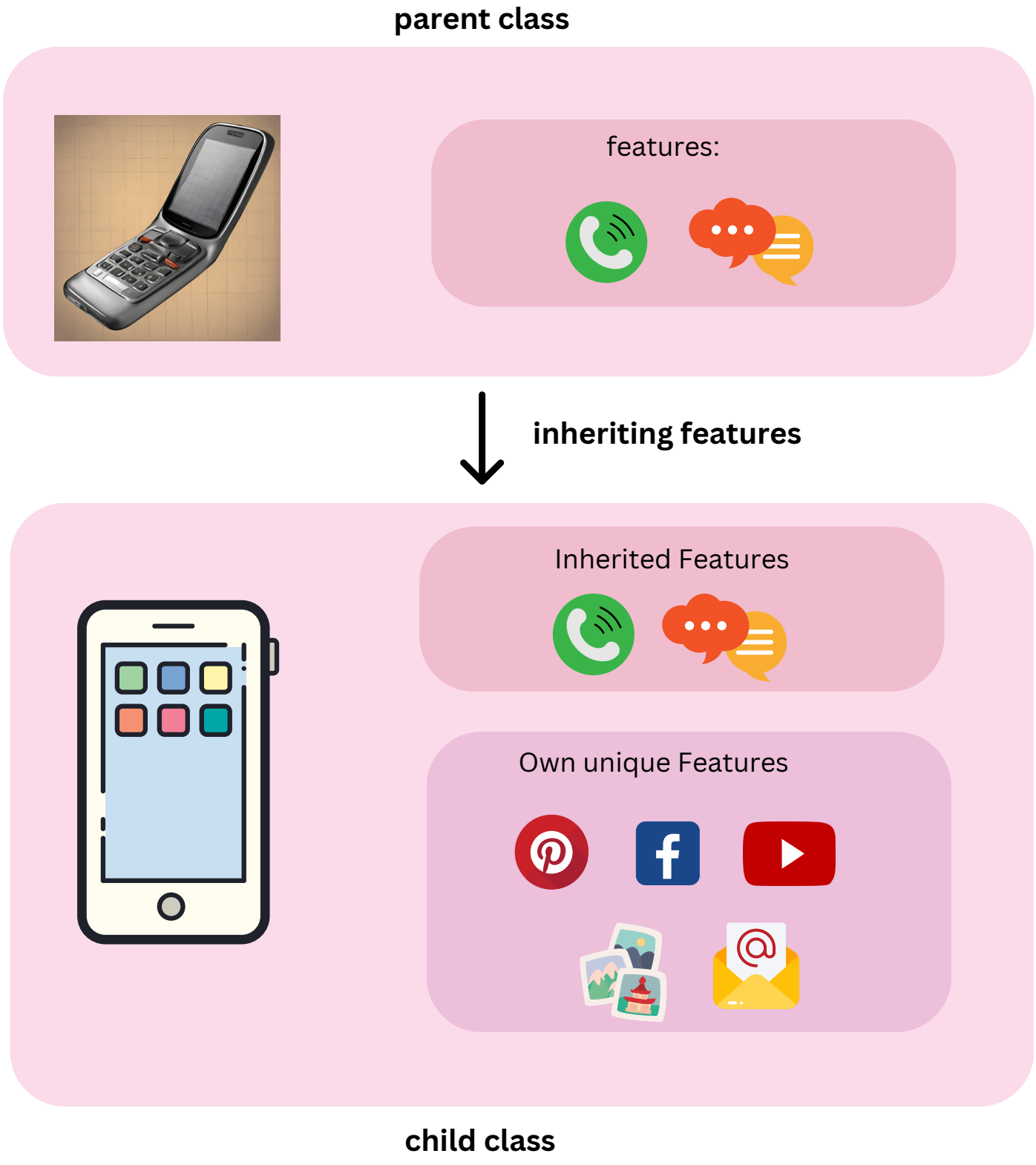
In this example:

- We have a **MobilePhone** class with encapsulated attributes **_color, _model, and __imei**.
- The **make_call() method** operates on encapsulated attributes **_color and _model**.
- We provide a **method get_imei()** to access the **private encapsulated attribute __imei.**
- Encapsulation is demonstrated by accessing encapsulated attributes through methods rather than directly accessing them.
- Attempting to access the private encapsulated attribute **directly (iphone.__imei) raises an error** because it's **inaccessible** from outside the class.

Encapsulation helps in hiding the internal implementation details of a class, which promotes data integrity and security.

10.4 Inheritance:

- Inheritance enables a new class (subclass) to inherit attributes and methods from an existing class (superclass). It promotes code reuse and allows for the creation of specialized classes that share common attributes and behaviors.
- Smartphone Class (Subclass): Inheritance allows for the creation of specialized classes that inherit attributes and methods from a more general class. For instance, a Smartphone class can inherit common features from the Mobile Phone class (such as make_call()) and add its own unique features (like access to mobile apps).



```
# Parent class (or superclass) - MobilePhone
class MobilePhone:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def make_call(self, number):
        print(f"Making a call from {self.brand} {self.model} to {number}...")

# Child class (or subclass) - Smartphone (inherits from MobilePhone)
class Smartphone(MobilePhone):
    def __init__(self, brand, model, os):
        # Call the constructor of the parent class
        super().__init__(brand, model)
        self.os = os

    def install_app(self, app_name):
        print(f"Installing {app_name} on {self.brand} {self.model} running {self.os}...")

# Creating instances of the Smartphone class
iphone = Smartphone("Apple", "iPhone 12", "iOS")
samsung = Smartphone("Samsung", "Galaxy S21", "Android")

# Using methods of both parent and child classes
iphone.make_call("1234567890")
samsung.install_app("WhatsApp")
```

```
#Output:
Making a call from Apple iPhone 12 to 1234567890...
Installing WhatsApp on Samsung Galaxy S21 running Android...
```

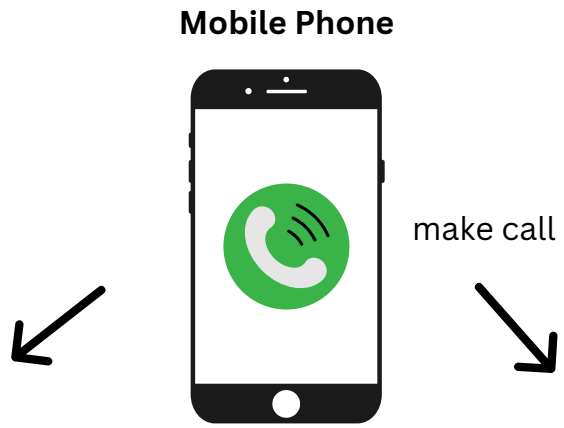
In this example:

- We have a parent class **MobilePhone** that represents basic mobile phones.
- We define a child class **Smartphone** that represents smartphones, which **inherit from MobilePhone.**
- The Smartphone class adds additional functionality like **installing apps.**
- When creating a Smartphone object, we provide **brand, model, and OS information.**
- We use the **make_call** method from the parent class MobilePhone for both iphone and samsung objects.
- We use the **install_app** method specific to the Smartphone class for the samsung object.

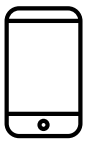
This demonstrates inheritance where the Smartphone class inherits attributes and methods from the MobilePhone class and adds its own functionality.

10.5 Polymorphism:

- Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables flexibility and reusability by allowing the same method to behave differently based on the object it is called on.
- **Polymorphic Behavior:** Polymorphism allows different types of mobile phones (e.g., iPhone, Samsung Galaxy) to share common behaviors (e.g., make_call()). Each mobile phone may implement the make_call() method differently, but they all adhere to the same interface.

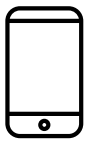


iphone



- Enable FaceTime
- Dial Number

Samsung Galaxy



- Start Call
- Activate Speakerphone
- Initiate Video Call

Explanation:

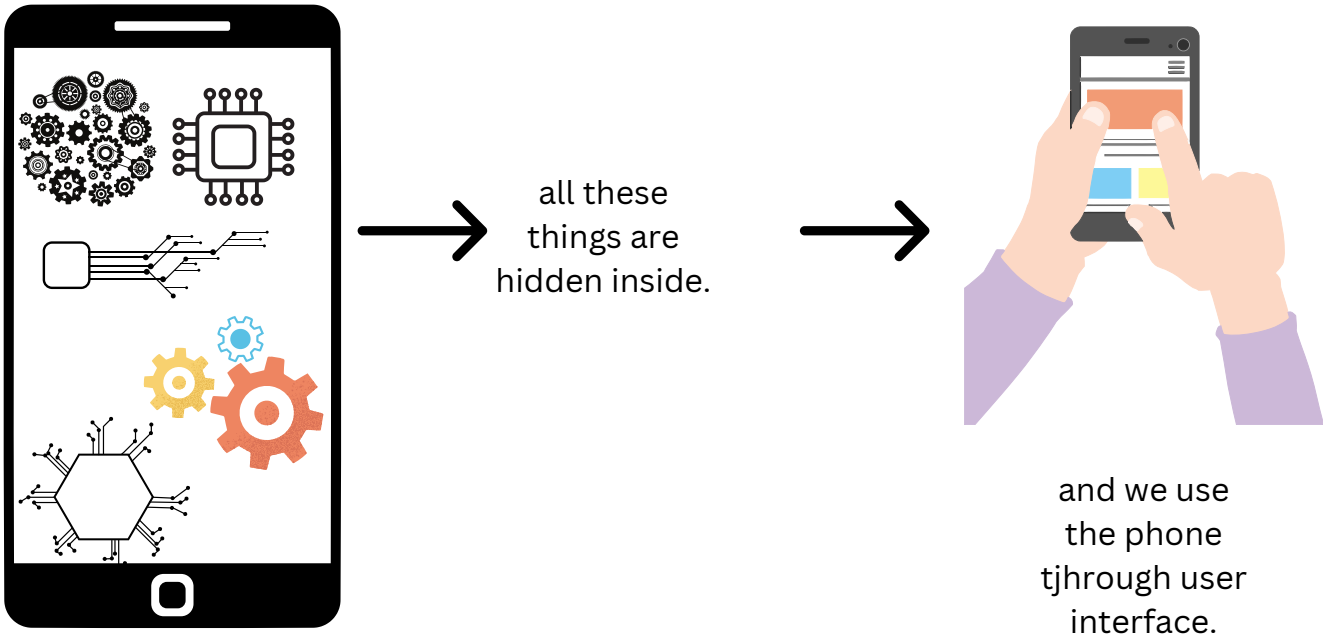
- We define a **MobilePhone** class as the common interface with a **make_call() method,** serving as a placeholder.
- Subclasses **IPhone** and **SamsungGalaxy** inherit from **MobilePhone** and override the **make_call() method** with their specific implementations.
- We define a function **make_phone_call()** that takes a **MobilePhone object** as an argument and calls its **make_call() method**, demonstrating **polymorphism.**
- We create instances of **IPhone** and **SamsungGalaxy** and pass them to **make_phone_call()** to make phone calls, showcasing polymorphism in action.

```
class MobilePhone:
    def make_call(self):
        pass    # Placeholder for the common interface

class IPhone(MobilePhone):
    def make_call(self):
        print("Initiating FaceTime call...")

class SamsungGalaxy(MobilePhone):
    def make_call(self):
        print("Starting regular call...")

# Using polymorphism with the common interface
def make_phone_call(phone):
    phone.make_call()

# Creating instances of different types of mobile phones
iphone = IPhone()
samsung = SamsungGalaxy()

# Making phone calls using polymorphism
print("Making a call from iPhone:")
make_phone_call(iphone)


print("\nMaking a call from Samsung Galaxy:")
make_phone_call(samsung)
```

```
#Output:
Making a call from iPhone:
Initiating FaceTime call...

Making a call from Samsung Galaxy:
Starting regular call...
```

10.6 Abstraction:

- Abstraction focuses on hiding the complex implementation details of a class and only showing the essential features of the object. It emphasizes what an object does rather than how it does it, making code easier to understand and maintain.
- Abstraction in Mobile Phone: Abstraction focuses on hiding the internal details of how a mobile phone works and exposing only the essential features to the user. For example, a user interacts with a mobile phone through its user interface (UI) without needing to understand the underlying hardware or software processes.



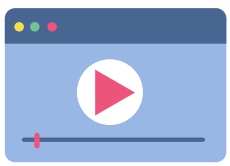
Explanation:

- We define an abstract class **Phone** using Python's abc module, with an abstract method **make_call.**
- The **make_call** method is abstract, meaning it has no implementation in the parent class. Subclasses must provide their own implementation.
- We define concrete **subclasses LandlinePhone** and **Smartphone** that inherit from Phone and provide their own implementations of the **make_call** method.
- The **dial_number** function demonstrates abstraction by taking a **Phone object** as an argument and calling its **make_call method** without needing to know the specific implementation details of each phone type.
- When we create instances of **LandlinePhone** and **Smartphone** and pass them to the **dial_number function**, it invokes the **make_call method** of each object, providing the necessary abstraction.

```
from abc import ABC, abstractmethod

# Abstract class - Phone
class Phone(ABC):
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    @abstractmethod
    def make_call(self):
        pass

# Concrete subclass 1 - LandlinePhone
class LandlinePhone(Phone):
    def make_call(self):
        return f"Dialing the number on a landline phone ({self.brand} {self.model})..."

# Concrete subclass 2 - Smartphone
class Smartphone(Phone):
    def make_call(self):
        return f"Dialing the number on a smartphone ({self.brand} {self.model})..."

# Function demonstrating abstraction
def dial_number(phone):
    print(phone.make_call())

# Creating instances of different phone types
landline = LandlinePhone("Panasonic", "KX-TG6822")
smartphone = Smartphone("Samsung", "Galaxy S20")

# Using the function with different phone objects
dial_number(landline)
dial_number(smartphone)
```

```
#Output:
Dialing the number on a landline phone (Panasonic KX-TG6822)...
Dialing the number on a smartphone (Samsung Galaxy S20)...
```

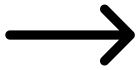
For more knowledge , Click on the link below to get a video tutorial of Object Oriented Programming in Python

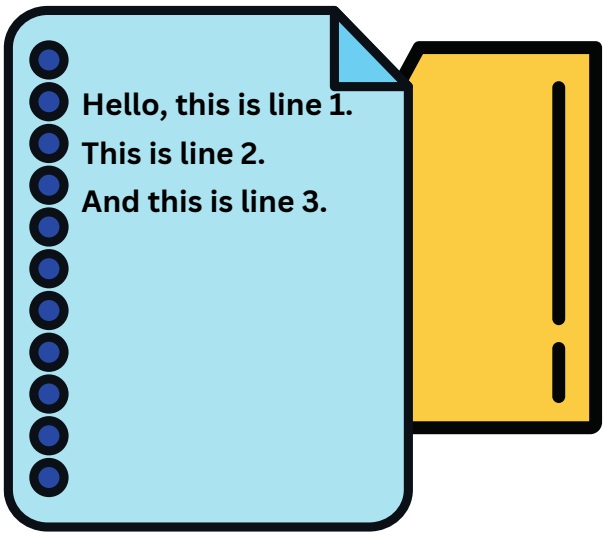


Chapter 11 : File Handling

File handling in Python refers to the process of working with files on your computer's file system. It involves tasks such as reading data from files, writing data to files, and performing other operations like creating, deleting, and modifying files.

File handling is essential for tasks such as reading configuration files, processing data from external sources, logging information, and much more. Python provides built-in functions and methods to perform these file handling operations efficiently, making it easy to work with files in your programs.





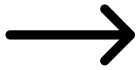
Let's say we have a file named **"example.txt"** stored on our local system, and now we'll attempt to access this file and perform file handling using Python.

11.1 Opening a File:

Use the open() function to open a file. Specify the file path and the mode

- **'r' for reading,**
- **'w' for writing,**
- **'a' for appending,**
- **'r+' for both reading and writing.**

```
file = open('example.txt', 'r')
print("File opened successfully!")
file.close()
```



File opened successfully!

11.2 Reading from a File:

Once the file is opened for reading, you can use various methods to read its contents:

- **read():** Reads the entire file as a single string.
- **readline():** Reads a single line from the file.
- **readlines():** Reads all lines from the file into a list.

1.read()

```
 # Opening the file 'example.txt' in read mode ('r') using a 'with' statement
with open('example.txt', 'r') as file:
# Reading the entire content of the file into a variable 'content'
 content = file.read()

# Printing the content of the file
print("Content:", content)
```

```
#output
 Hello, this is line 1.
This is line 2.
 And this is line 3.
```

2. readline()

```
 # Opening the file 'example.txt' in read mode ('r') using a 'with' statement
with open('example.txt', 'r') as file:
 # Reading the first line of the file into a variable 'line'
 line = file.readline()

# Printing the the first line of the file
print("Line:", line)
```

```
#output
 Hello, this is line 1.
```

3. readlines()

```
# Opening the file 'example.txt' in read mode ('r') using a 'with' statement
 with open('example.txt', 'r') as file:
 # Reading all remaining lines of the file into a list 'lines'
 lines = file.readlines()

# Printing all lines of the file.
 print("Lines:", lines)
```

```
#output
Lines: ['Line 1: Hello, this is line 1.\n', 'Line 2: This is line 2.\n', 'Line 3: And this is line 3.\n']
```

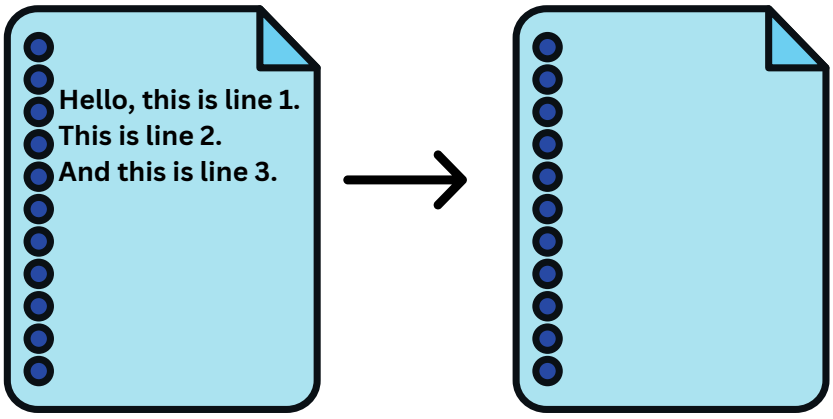
11.3 Writing to a File:

If the file is opened for writing or appending, you can use the write() method to write data to the file.

1.w - write

```
with open('example.txt', 'w') as file:
file.write('Hello, world!')
print("Data written to file successfully!")
```

```
#output
Data written to file successfully!
```

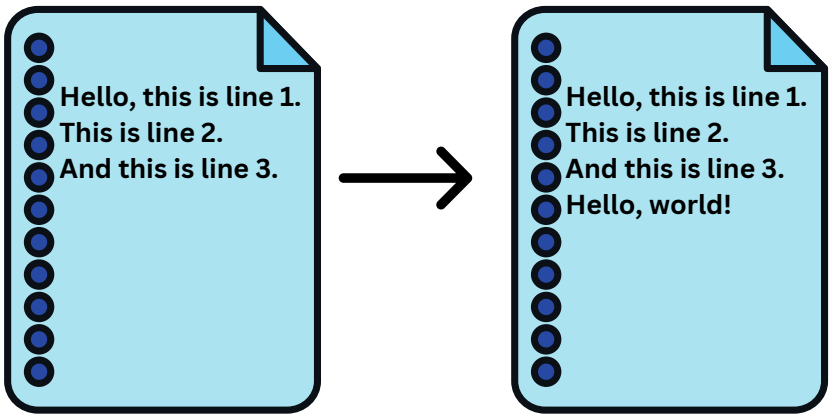


the contents of the file '**example.txt**' will be **overwritten** with the new content **"Hello, world!"** as a result of using the '**w**' mode in the open() function, which truncates the file before writing to it**.**

2. a - append

```
with open('example.txt', 'a') as file:
file.write('\nHello, world!')
print("Data appended to file successfully!")
```

```
#output
Data appended to file successfully!
```



- The **open()** function is used with mode '**a**' to open the file in append mode. This allows new content to be **added to the end of the file** without overwriting existing content.
- The **write()** method is then used to **append** the string **"Hello, world!"** to the file.

11.4 Closing a File:

After you finish working with a file, it's important to close it using the close() method to free up system resources.

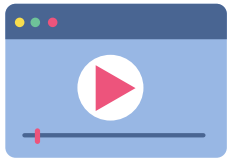
```
file = open('example.txt', 'r')
content = file.read()
file.close()
print("File closed successfully!")
print("Content:", content)


Hello, world!
```

```
#output
File closed successfully!

# output :
Content: Hello, this is line 1.
         This is line 2.
         And this is line 3.
         Hello, world!
```

The file is closed now.

These examples demonstrate various file handling operations in Python along with their respective outputs. Understanding these concepts and their outputs will help you effectively work with files in your Python programs.

For more knowledge , Click on the link below to get a video tutorial of Python File Handling



Time to sssslip away for now, my friendssss!

if you ever sssslither back thiss way, I'll be here to help.

Until then, ssstay ssssafe and keep exploring!

Before that!! Don't miss out on testing your Python skills.

Click below to explore basic, medium, and advanced Python questions.

Ready to dive into the Python knowledge pool?"

Basic

Medium

Advanced