## ⬛ `If` Statements

The `if` statement executes a block of code if a specified condition is `True`.

```
age = 18
if age >= 18:
    print("You are an adult.")
```

## ⬛ `Else` Statements

The `else` statement executes a block of code if the `if` condition is `False`.

```
age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are not an adult.")
```

## ⬛ Else If `(Elif)` Statements

The `elif` statement allows you to check multiple conditions. It stands for "else if" and can be used when you need to check more than one condition.

```
age = 16
if age >= 18:
    print("You are an adult.")
elif age >= 13:
    print("You are a teenager.")
else:
    print("You are a child.")
```

## ⬛ Combining `If`, `Elif`, and `Else` Statements

```
score = 75

if score >= 90:
    print("Grade: A")
```

```
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
elif score >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

## ⬛ Nested `If` Statements

You can also nest `if` statements within other `if` statements to check more complex conditions.

```
age = 20
has_permission = True

if age >= 18:
    if has_permission:
        print("You can enter the club.")
    else:
        print("You need permission to enter the club.")
else:
    print("You are not allowed to enter the club.")
```

### if Else use case

- **User Authentication**: Check if the entered username and password match the stored credentials and grant or deny access.
- **Form Validation**: Validate user input in forms and provide feedback or error messages.
- **Payment Processing**: Determine if a payment transaction is successful or if an error occurred, and handle each case accordingly.
- **Data Filtering**: Filter data based on specific criteria, such as filtering out invalid entries from a dataset.
- **Weather Forecasting**: Display different messages or actions based on weather conditions, such as suggesting an umbrella if it's going to rain.
- **Inventory Management**: Check if stock levels are sufficient to fulfill an order and alert if more inventory is needed.
- **Game Logic**: Determine game outcomes based on player actions or states, such as winning, losing, or drawing a game.

- **Personalized Greetings**: Provide personalized greetings or messages based on the time of day or user preferences.
- **Discount Application**: Apply discounts to purchases based on customer status, such as member, non-member, or special promotions.
- **File Handling**: Check if a file exists before attempting to read or write to prevent errors and handle cases where the file is missing.

---

## ⬛ `for` Loop in Python

The `for` loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) or other iterable objects.

### Iterating Over a List

```
# Example of iterating over a list
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

### Iterating Over a String

```
# Example of iterating over a string
word = "hello"
for letter in word:
    print(letter)
```

### Using `range()` Function

```
# Example of using range() function
for i in range(5):
    print(i)
```

### Iterating Over a Dictionary

```
# Example of iterating over a dictionary
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
for student, score in student_scores.items():
```

```
    print(f"{student}: {score}")
```

### Iterating Over a Set

```
# Example of iterating over a set
unique_numbers = {1, 2, 3, 4, 5}
for number in unique_numbers:
    print(number)
```

### Using `break` Statement

```
# Example of using break statement
for number in range(10):
    if number == 5:
        break
    print(number)
```

### Using `continue` Statement

```
# Example of using break statement
for number in range(10):
    if number == 5:
        break
    print(number)
```

## for loop use case

- **Data Processing**: Iterate over a list of data points to perform calculations or transformations.
- **File Handling**: Read and process lines in a file sequentially.
- **Generating Reports**: Create summaries or reports by iterating over data records.
- **Batch Processing**: Apply operations to a batch of items, such as resizing images or processing transactions.
- **Automating Tasks**: Automate repetitive tasks like sending emails or making API calls.
- **Iterating Over Dictionaries**: Access keys and values in a dictionary for tasks like configuration or data analysis.

- **Matrix Operations**: Perform operations on matrices or 2D arrays, such as addition, multiplication, or transposition.
- **Building User Interfaces**: Generate dynamic UI components by iterating over data models.
- **Simulation and Modeling**: Run simulations by iterating over time steps or model parameters.
- **Web Scraping**: Extract information from web pages by iterating over HTML elements.

---

## ⬛ `While` Loops in Python

### Iterating Over a List

```
# Iterating over a list with a while loop
fruits = ['apple', 'banana', 'cherry']
index = 0
while index < len(fruits):
    print(fruits[index])
    index += 1
```

### Iterating Over a String

```
# Iterating over a string with a while loop
word = "hello"
index = 0
while index < len(word):
    print(word[index])
    index += 1
```

### Using range() Function

```
# Simulating range() with a while loop
start = 0
end = 5
while start < end:
    print(start)
    start += 1
```

### Iterating Over a Dictionary

```
# Iterating over a dictionary with a while loop
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
keys = list(student_scores.keys())
index = 0
while index < len(keys):
    key = keys[index]
    print(f"{key}: {student_scores[key]}")
    index += 1
```

### Iterating Over a Set

```
# Iterating over a dictionary with a while loop
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
keys = list(student_scores.keys())
index = 0
while index < len(keys):
    key = keys[index]
    print(f"{key}: {student_scores[key]}")
    index += 1
```

### Using break Statement

```
# Using break statement in a while loop
counter = 0
while counter < 10:
    if counter == 5:
        break
    print(counter)
    counter += 1
```

### Using continue Statement

```
# Using continue statement in a while loop
counter = 0
while counter < 10:
    counter += 1
    if counter % 2 == 0:
        continue
```

```
    print(counter)
```

### While loop use case

- **User Input Validation**: Continuously prompt the user for input until valid data is provided.
- **Reading Files**: Read data from a file until the end of the file is reached.
- **Polling for Changes**: Continuously check for changes in data or status until a condition is met.
- **Implementing Timers**: Create countdown timers or delay loops.
- **Game Loops**: Run the main loop of a game, which continues until the game is over.
- **Retry Logic**: Retry an operation until it succeeds or a maximum number of attempts is reached.
- **Simulations**: Run simulations that proceed until a certain condition is met.
- **Processing Queues**: Process items from a queue until it is empty.
- **Progress Tracking**: Track and update progress until a task is complete.
- **Generating Sequences**: Generate a sequence of numbers or data until a certain condition is reached.

## ⬛ What about other's type of loop

- In Python, there are `for` and `while` loops, but there is no direct equivalent to the `do-while` loop found in some other programming languages.
- Additionally, there is no `for in`, `for of`, or `forEach` loop syntax specifically like in JavaScript

---

## ⬛ Logical Operators in Python

Logical operators are used to combine conditional statements. The most common logical operators in Python are `and`, `or`, and `not`.

### 1. `and` Operator

The `and` operator returns `True` if both conditions are `True`. If either condition is `False`, the result is `False`.

```
age = 20
has_permission = True
```

```
if age >= 18 and has_permission:
    print("You can enter the club.")
else:
    print("You cannot enter the club.")
```

## 2. `or` Operator

The `or` operator returns `True` if at least one of the conditions is `True`. If both conditions are `False`, the result is `False`.

```
age = 16
has_permission = True

if age >= 18 or has_permission:
    print("You can enter the club.")
else:
    print("You cannot enter the club.")
```

## 3. `not` Operator

The `not` operator inverts the result of the condition. If the condition is `True`, `not` makes it `False`, and if the condition is `False`, `not` makes it `True`.

```
age = 16

if not age >= 18:
    print("You are not an adult.")
else:
    print("You are an adult.")
```

## 4. Combining Logical Operators

You can combine multiple logical operators to form more complex conditions.

```
age = 20
has_permission = False
is_vip = True

if (age >= 18 and has_permission) or is_vip:
    print("You can enter the club.")
```

```
else:
    print("You cannot enter the club.")
```

### Logical Operators Use case

- **Access Control**: Check multiple conditions to grant or deny access to resources.
- **Input Validation**: Validate multiple input criteria simultaneously.
- **Search Functionality**: Filter search results based on multiple criteria.
- **Feature Toggles**: Enable or disable features based on various conditions.
- **Data Filtering**: Filter data records based on multiple conditions.
- **E-commerce**: Apply discounts and promotions based on combined conditions.
- **Game Development**: Determine game state changes based on multiple player actions or game conditions.
- **Scheduling**: Check for multiple availability conditions before scheduling an event.
- **Configuration Management**: Apply configuration settings based on multiple environment variables or settings.
- **Monitoring and Alerts**: Trigger alerts based on combined system monitoring conditions.

---

## ⬛ Comparison Operators in Python

Comparison operators are used to compare two values and return a Boolean result (`True` or `False`). These operators are essential for making decisions in your code using conditional statements.

### 1. Equal to (`==`)

The `==` operator checks if two values are equal.

```
x = 5
y = 5
print(x == y)  # True
```

### 2. Not equal to (`!=`)

The `!=` operator checks if two values are not equal.

```
x = 5
y = 3
print(x != y)  # True
```

### 3. Greater than ( `>` )

The `>` operator checks if the value on the left is greater than the value on the right.

```
x = 7
y = 5
print(x > y)  # True
```

### 4. Less than ( `<` )

The `<` operator checks if the value on the left is less than the value on the right.

```
x = 3
y = 5
print(x < y)  # True
```

### 5. Greater than or equal to ( `>=` )

```
x = 5
y = 5
print(x >= y)  # True
```

### 6. Less than or equal to ( `<=` )

```
x = 5
y = 7
print(x <= y)  # True
```

#### Comparison Operators Use case

- **User Authentication**: Verify if entered credentials match stored credentials.
- **Input Validation**: Ensure user input meets specific criteria, such as age or date range.
- **Sorting Data**: Compare elements to sort lists, tuples, or other data structures.

- **Conditional Formatting**: Apply different formatting based on data values, such as highlighting high scores.
- **Inventory Management**: Check stock levels and trigger reorder processes if inventory falls below a certain threshold.
- **Financial Transactions**: Validate if transactions exceed credit limits or fall within acceptable ranges.
- **Performance Monitoring**: Compare current system metrics against baseline values to trigger alerts.
- **Game Development**: Determine outcomes based on player scores or in-game conditions.
- **Access Control**: Grant or deny access based on user roles or permissions.
- **Data Analysis**: Filter and segment data based on comparison criteria.