



## Day02 Presentation Slides

---



# Contents

- String
- Functions
- Import & from
- Loops
- Custom Class
- Tuple



# String Intro

- A string is an object that represents **sequences of characters**
- A string object is **immutable**, Once it is created it can't be altered
- String objects are surrounded by either **single quotes** or **double quotes**

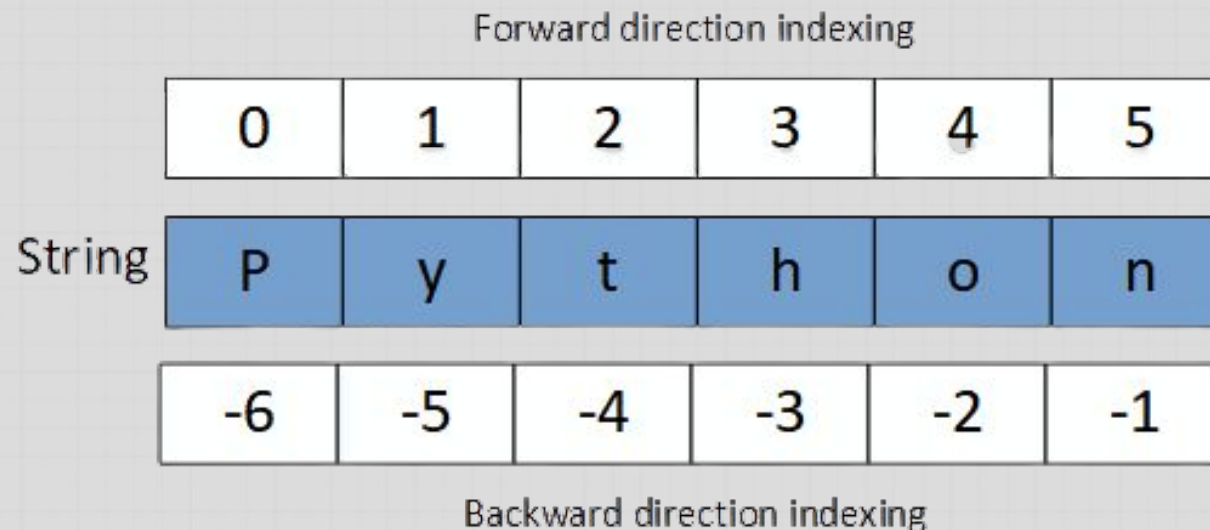
```
name = 'Wooden Spoon'  
language = "Python"
```

```
sentence1 = 'This is a string'  
sentence2 = "This is also a string"
```



# String: Sequences of characters

- Strings are ordered sequences of characters, and each character has two index numbers:
  - **Forward Index:** starts from 0 (**first** character) and increases by 1 for each next character to the **right**
  - **Reverse Index:** starts from -1 (**last** character) and decreases by 1 for each next character to the **left**



# String: Indexing

- Indexing allows us to access individual characters of string by using brackets [ ]

```
name = 'Python'

# forward indexes: 0, 1, 2, 3, 4, 5
# backward indexes: -1, -2, -3, -4, -5, -6

print(name[0])    # prints 'p'
print(name[1])    # prints 'y'

print(name[-1])   # prints 'n'
print(name[-2])   # prints 'o'

print(name[10])   # string index out of range
print(name[-10])  # string index out of range
```



# Slicing Strings: Start & End Indexes

- Creates a range of characters (substring) by using the slice syntax `[ start : end ]`
- We can specify the start index and end index (excluded), separated by a colon, to create the substring

```
word = 'Wooden Spoon'
```

```
s1 = word[0:6]
```

```
print(s1) # prints 'Wooden'
```

```
s2 = word[2:6]
```

```
print(s2) # prints 'oden'
```

Slices "Wooden Spoon" starting from index 0 to index 6 (**index 6 is excluded**)

Slices "Wooden Spoon" starting from index 2 to index 6 (**index 6 is excluded**)



# Slicing Strings: Slice From the Start

- By not giving the **start** index, the slicing starts from first character **[ : end ]**

```
word = 'Wooden Spoon'
```

```
s1 = word[:7] ←  
print(s1) # prints 'Wooden'
```

Slices "Wooden Spoon" starting from index 0 to index 7 (**index 7 is excluded**)

```
word = "Hello Cydeo"
```

```
s1 = word[:5] ←  
print(s1) # prints 'Hello'
```

Slices "Hello Cydeo" starting from index 0 to index 5 (**index 5 is excluded**)





# Slicing Strings: Slice to the End

- By not giving the **end** index, the slicing starts from the start index to the end of the string `[ start : ]`

```
word = 'Wooden Spoon'
```

```
s1 = word[7:]  
print(s1) # prints 'Spoon'
```

Slices "Wooden Spoon" starting from index 7 to the **end**

```
word = "Hello Cydeo"
```

```
s2 = word[6:]  
print(s2) # prints 'Cydeo'
```

Slices "Hello Cydeo" starting from index 6 to the **end**





# String Methods

- Python has a built-in string class named `str`
- The built-in string class (`str`) has a set of built-in methods that we can use
- A string object is `immutable`, methods of string `can not` change the original string, therefore they return new values



# String Methods

Method Name	Method Name	Method Name
<code>lower()</code>	<code>upper()</code>	<code>capitalize()</code>
<code>title()</code>	<code>strip()</code>	<code>index()</code>
<code>rindex()</code>	<code>replace()</code>	<code>count()</code>
<code>swapcase()</code>	<code>startswith()</code>	<code>endswith()</code>
<code>islower()</code>	<code>isupper()</code>	<code>isdigit()</code>
<code>isalpha()</code>	<code>istitle()</code>	



# Function

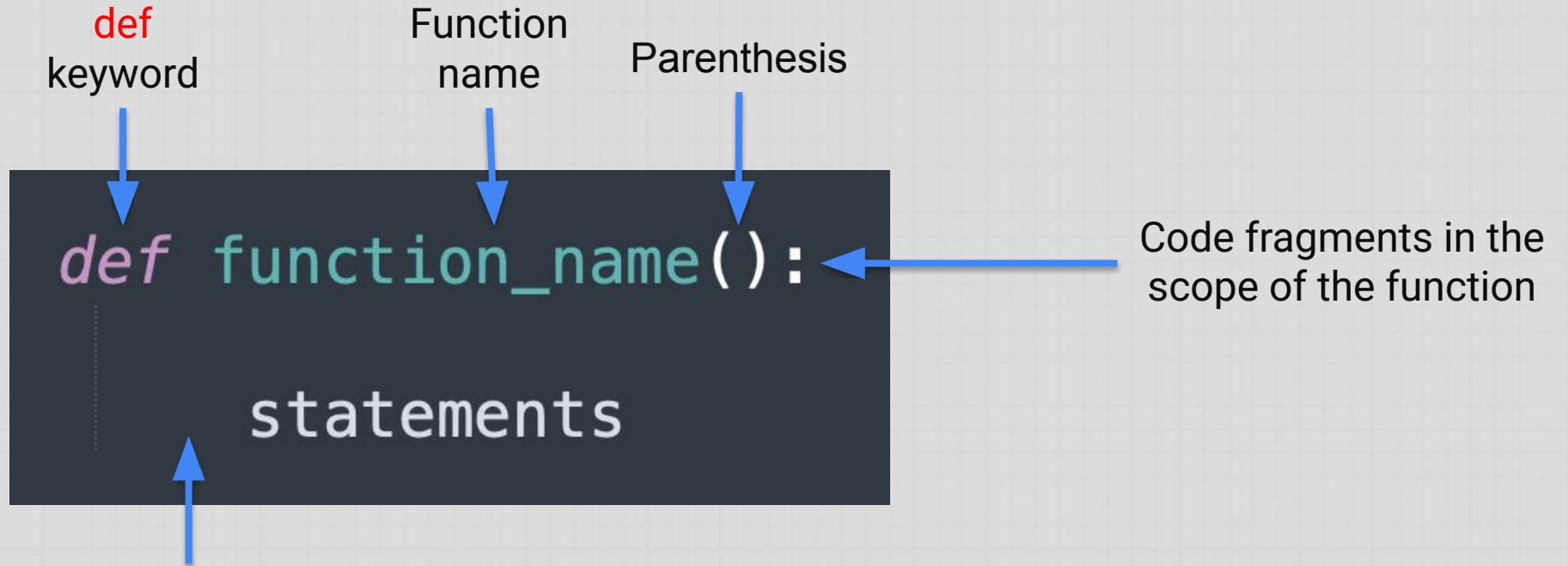
- Grouping a series of code fragments to perform a task
- Allows us to reuse the function rather than repeating the same set of statements
- Improves the reusability and efficiency of our codes

```
print()  
input()  
cube()  
display_message()  
reverse()  
sort()  
...
```



# Declaring A Function

- A function must be declared before we call it and use it
- To create a function, we need to use the **def** keyword

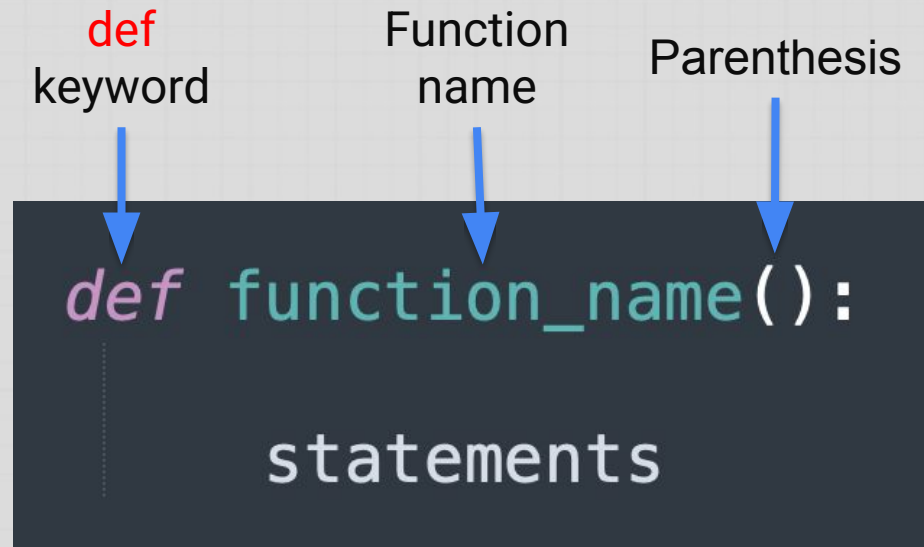


To define the scope of the function, **indentation** (whitespaces at the beginning of the line) is needed



# Components of Function

- The **def** keyword: indicates that the start of the function
- The **function name**: Descriptive name of the function
- The **parenthesis**: function/method name is always followed by a set of parenthesis, can be capable of receiving **arguments**




The diagram shows a function definition on a dark background. Above the code, three labels with arrows point to specific parts: 'def keyword' points to 'def', 'Function name' points to 'function\_name', and 'Parenthesis' points to '()'. The code itself is 'def function\_name(): statements', with 'def' in pink, 'function\_name' in teal, '()' in white, and 'statements' in white.

```
def keyword      Function name      Parenthesis
  ↓              ↓                   ↓
def function_name():
    statements
```



# Calling a Function

- When we need to script to perform the task the function does
- The function executes the codes in its scope from top to bottom
- When it has finished, the code continues to run from the point where it was initially called




The diagram illustrates the execution flow of a function call. It features a dark gray code block with four numbered blue circles (1, 2, 3, 4) to its left. A red arrow points upwards from circle 4 to circle 3, and another red arrow points upwards from circle 3 to circle 2. A red line also connects the function definition code to the call site.

```
def display_message():  
    print("Hello World")  
  
# Code before calling function  
display_message():  
  
# Code before calling function
```



# Calling a Function

`display_message()`



```
def display_message():  
    print("Hello World!")  
    print("I love Python")
```





# Passing Parameters to Function

- When we declare a function, **parameters** can be given
- Parameters passed to the function act like variables within the function's scope
- Used for providing additional information the function **must** have to perform its task

Parameter




```
def display_value(value):  
    print(f"The value is {value}")
```



# Calling a Function that Needs Information

- Must specify the values the method should use
- Values need to be given in the parentheses that follow the function name
- The values we passed to the method are called **arguments**
- Arguments can be provided as values or as variables

Argument



```
display_value(100)
```



# Calling a Function that Needs Information

```
display_value(100)
```

Argument **100** is copied into  
the parameter variable value


```
def display_value(value):  
    print(f"The value is {value}")
```



# Return Values From Functions

- A function can return a value by using a return statement
- The value that the function returned can be used outside the function's scope
- When the function is being called, it will give the value that the function returned

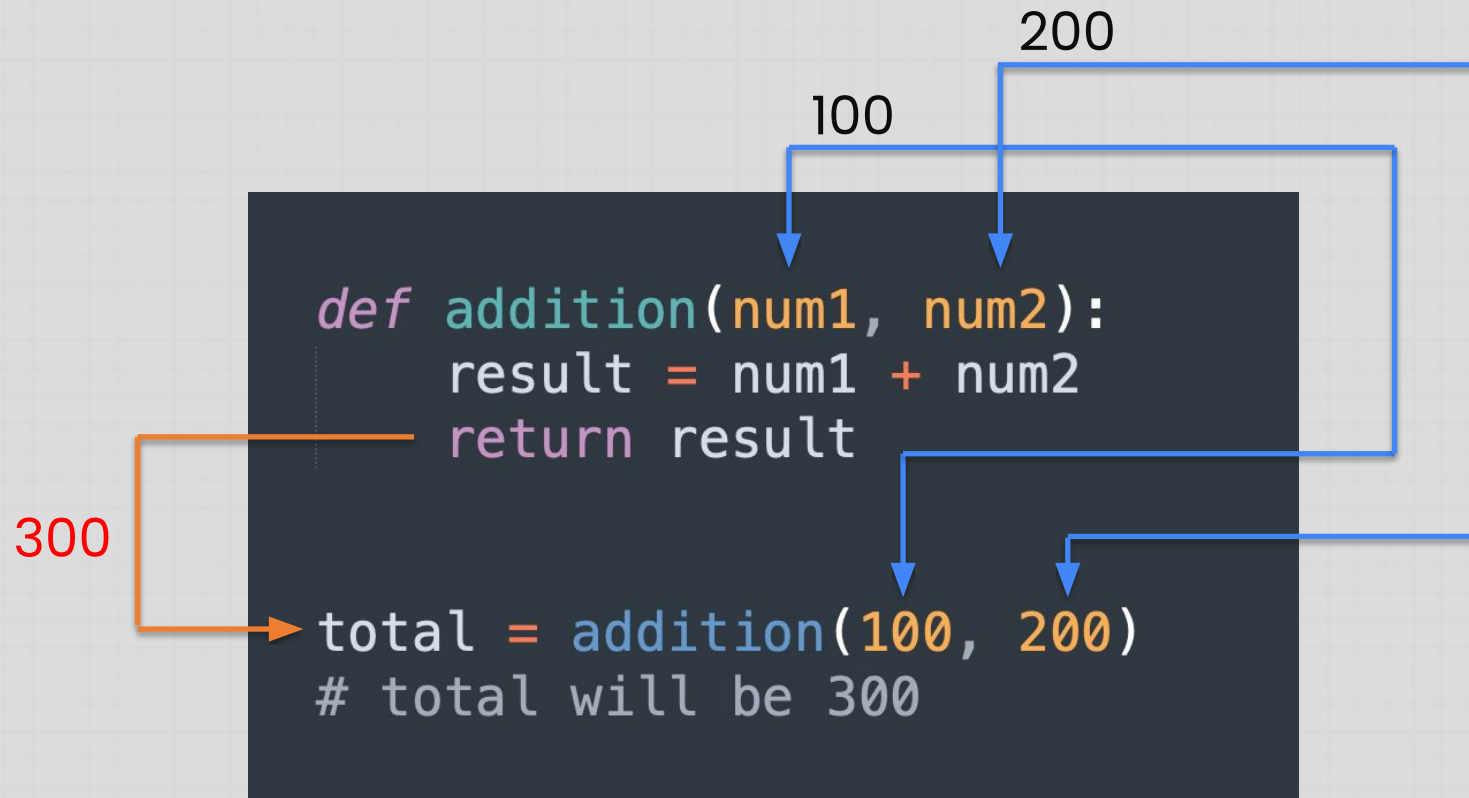
**Return  
Expression**



```
def addition(num1, num2):  
    result = num1 + num2  
    return result
```



# Calling a Value Returning Function



# Import Statement

- Allows us to reuse the functions/features of one python file (.py) in another
- We need to import the python file in order to use its properties in other python files

My\_Library.py

```
def square(num):  
    return num * num  
  
def cube(num):  
    return square(num) * num  
  
def print_each(sequence):  
    for each in sequence:  
        print(each)
```

Test.py

```
import My_Library  
  
n1 = My_Library.square(9)  
  
n2 = My_Library.cube(5)  
  
list1 = {'Python', 'Cydeo', 'Wooden Spoon'}  
My_Library.print_each(list1)
```



# Define Alias for Import Statement

- We can create an alias name by using the **as** keyword when we import a python file

My\_Library.py

```
def square(num):  
    return num * num  
  
def cube(num):  
    return square(num) * num  
  
def print_each(sequence):  
    for each in sequence:  
        print(each)
```

Test.py

```
import My_Library as lib  
  
n1 = lib.square(9)  
  
n2 = lib.cube(5)  
  
list1 = {'Python', 'Cydeo', 'Wooden Spoon'}  
lib.print_each(list1)
```





# From Keyword

- The **from** keyword allows us to import only parts of python files properties

My\_Library.py

```
def square(num):  
    return num * num  
  
def cube(num):  
    return square(num) * num  
  
def print_each(sequence):  
    for each in sequence:  
        print(each)
```

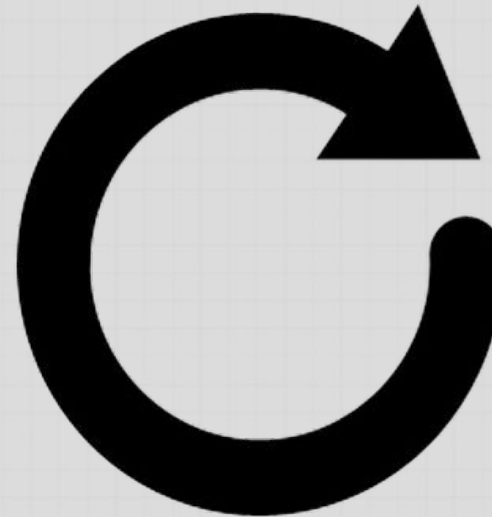
Test.py

```
from My_Library import print_each, cube  
  
n = cube(20)  
  
tuple1 = {'Cherry', 'Apple', 'Lemon'}  
print_each(tuple1)
```



# Loops

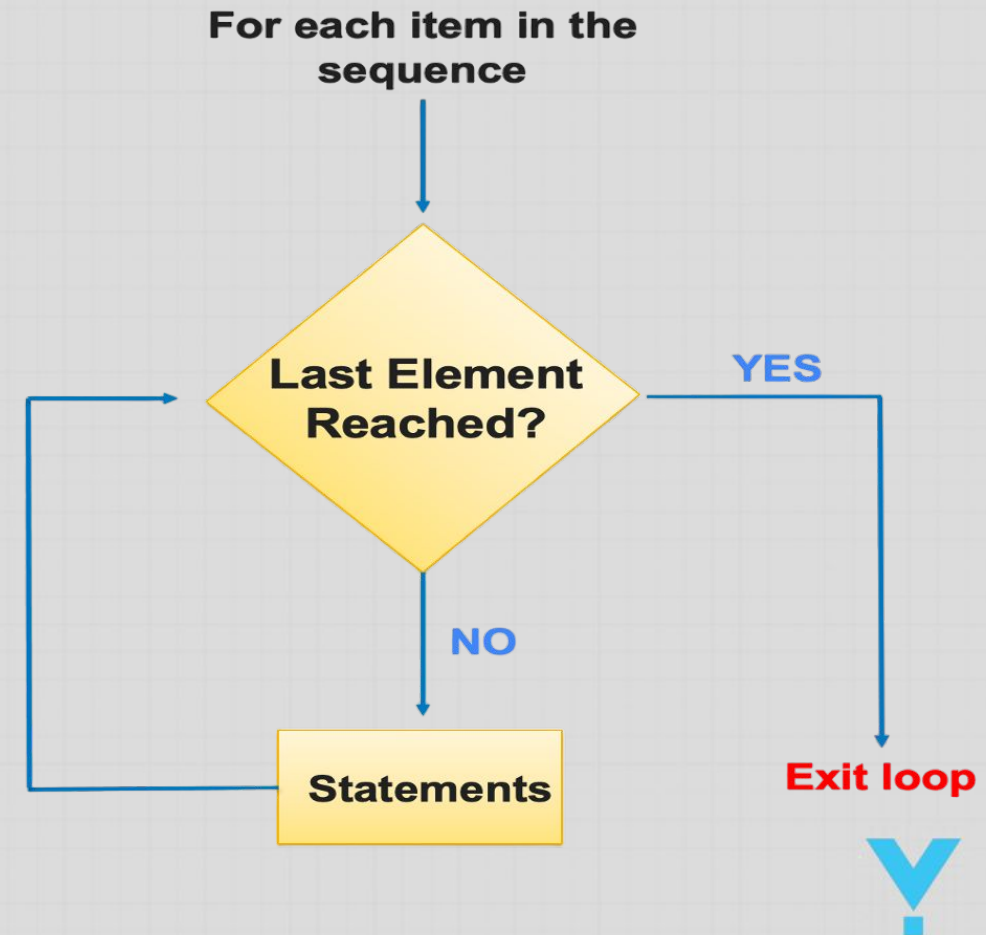
- Used for repeating a set of statements
- There are two types of loops:
  - For Loop
  - While Loop



LOOPS REPEAT  
ACTIONS...  
SO YOU DON'T HAVE TO

# For Loop

- For loop is used to access each successive value of a **sequence** or a **data structure**
- The **built-in** sequences and data structures are:
  - **String**
  - **Tuple**
  - **List**
  - **Set**
  - **Dictionary**



# For Loop Syntax

A variable name (**can be any name**), which used for containing each element of the sequence during each execution of the loop

Reference name of a sequence

```
for varName in sequence:  
    statements
```

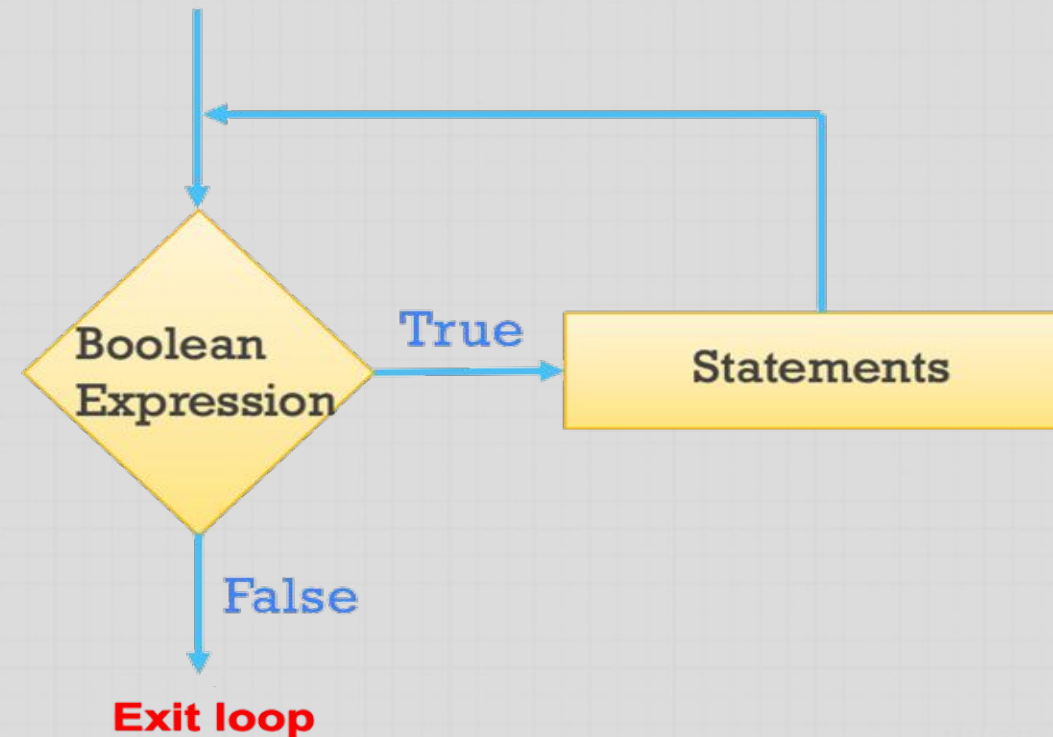
This code fragment is within the scope of the for loop

To define the scope of the while loop, **indentation** (whitespaces at the beginning of the line) is needed



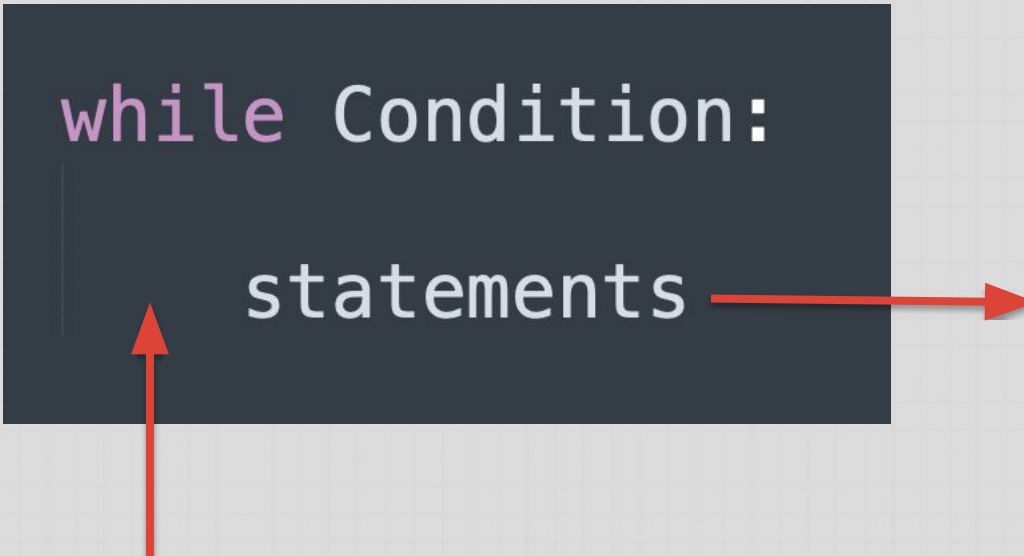
# While Loop

- Repeated If Statement
- While Loop checks a condition
- If the condition returns **true**, a code block will run
- The condition will be checked again
- It repeats until the condition returns **false**



# While Loop Syntax

```
while Condition:  
    statements
```

A diagram illustrating the syntax of a while loop. It features a dark gray rectangular box containing the code 'while Condition:' on the first line and 'statements' on the second line, which is indented. A red arrow points upwards from the text below to the start of the 'statements' line. Another red arrow points from the 'statements' line to the explanatory text on the right.

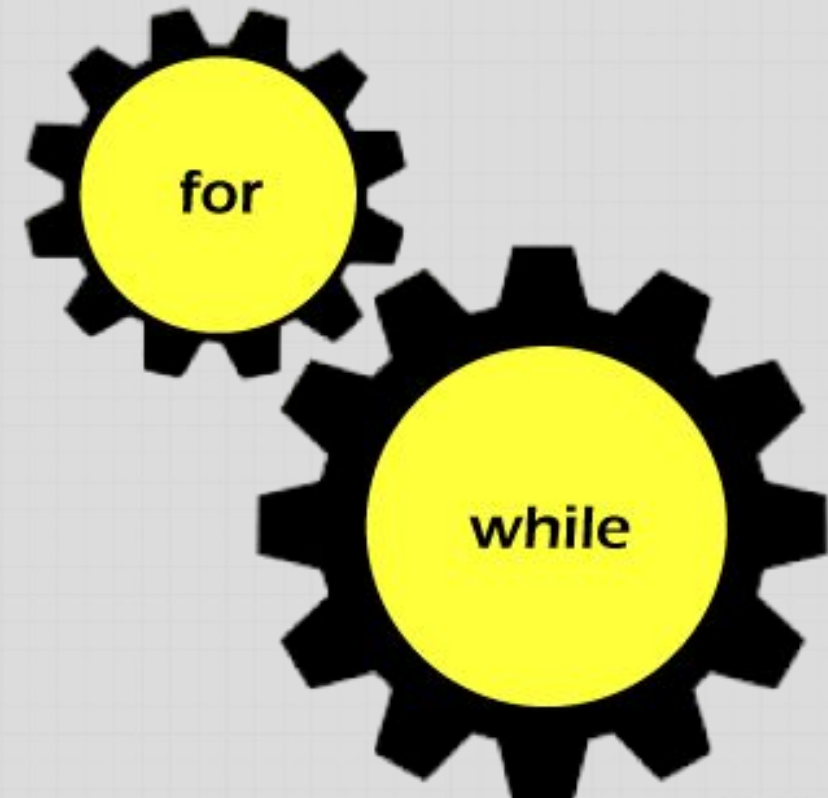
This code fragment is within the scope of the while loop

To define the scope of the while loop, **indentation** (whitespaces at the beginning of the line) is needed



# For Loop vs While Loop

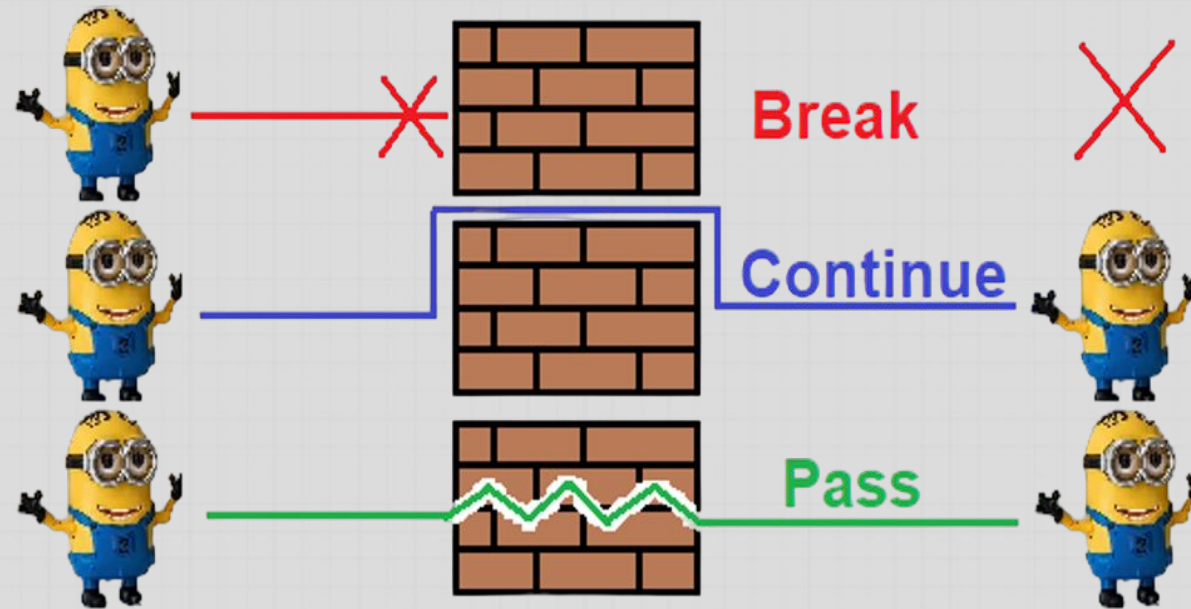
- You can use a **for loop**, a **while loop**, whichever is convenient
- A **for-loop** maybe used if the number of repetition is known in advance
- A **while loop** may be used used if the number of repetition is not fixed





# Branching Statements

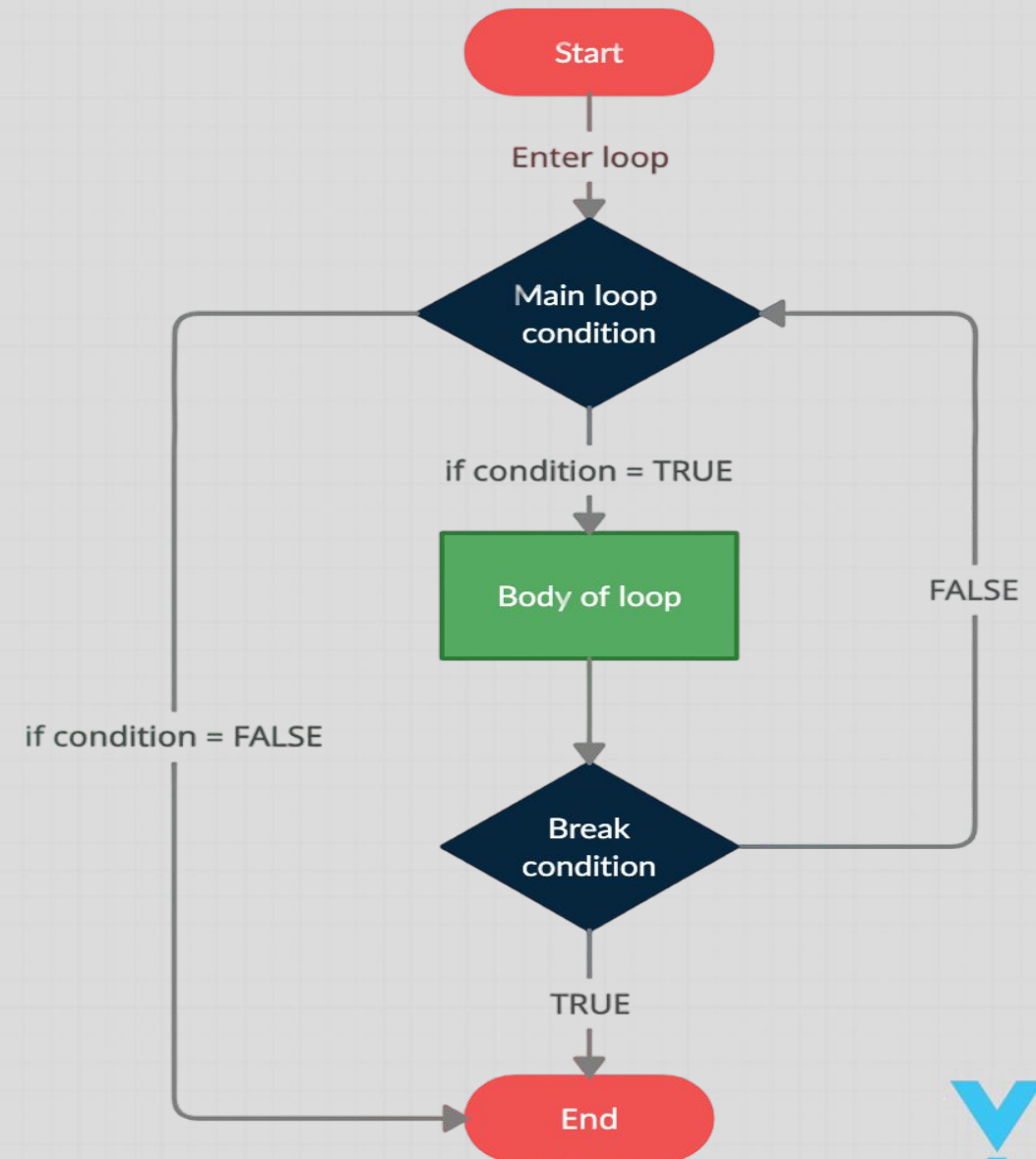
- Used to change the normal flow of execution
- There are three branching statements:
  - Break
  - Continue
  - Pass



# Break Statement

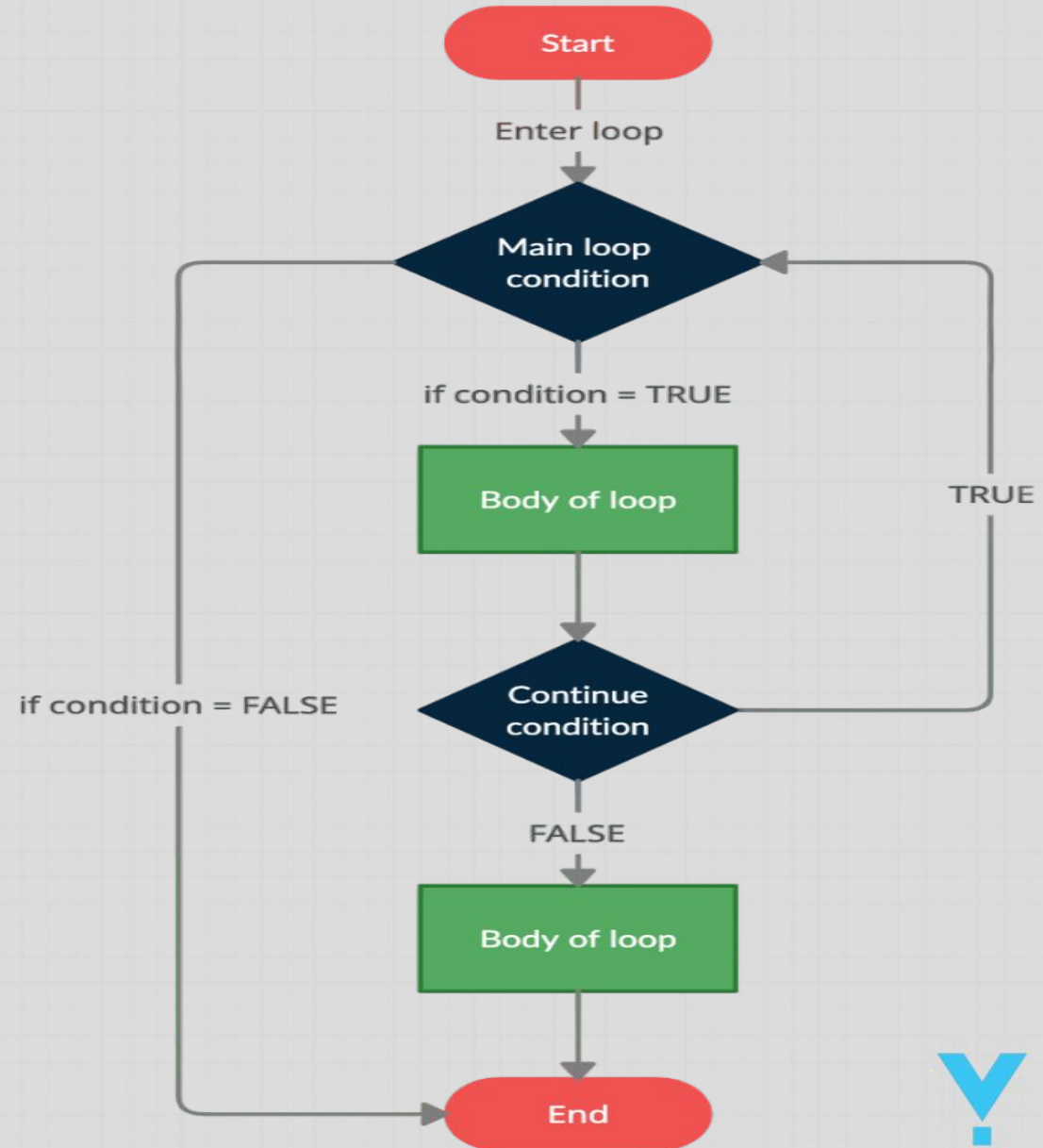
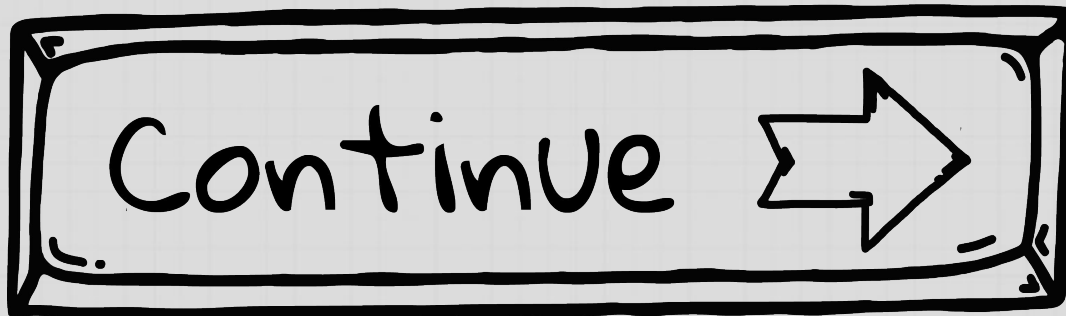
- Causes the **termination** of the loop
- Tells the interpreter to go on to the next statement of code **outside** of the loop

# BREAK IT



# Continue Statement

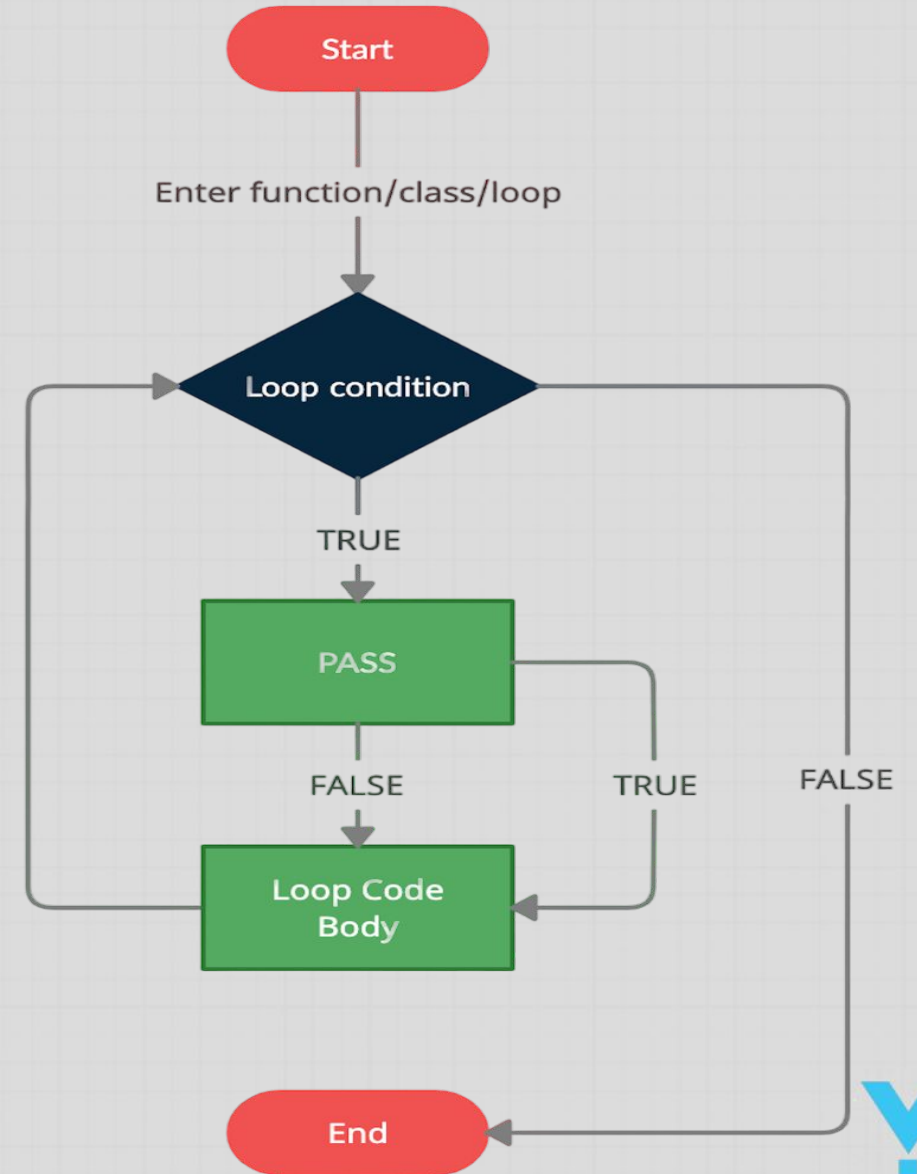
- Skips the current iteration of the loop
- Tells the interpreter to jump to the **next** iteration



# Pass Statement

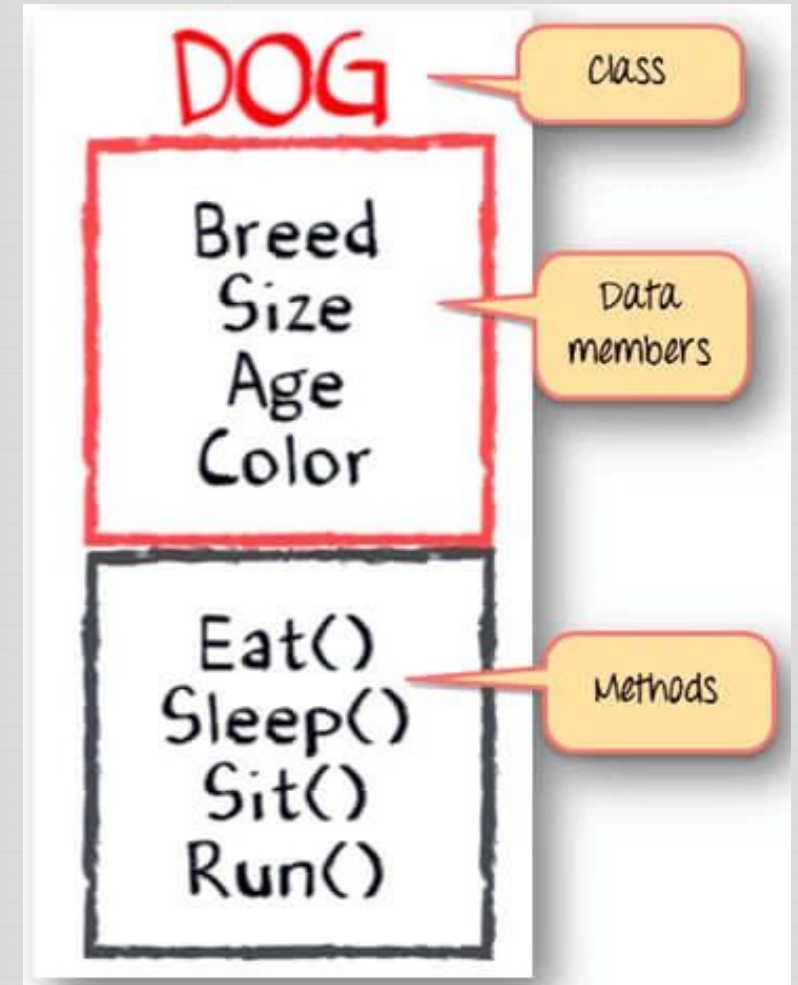
- Used as a placeholder in a loop/function/class
- Nothing happens when the pass is executed
- Results in no operation

**PASS-ON**



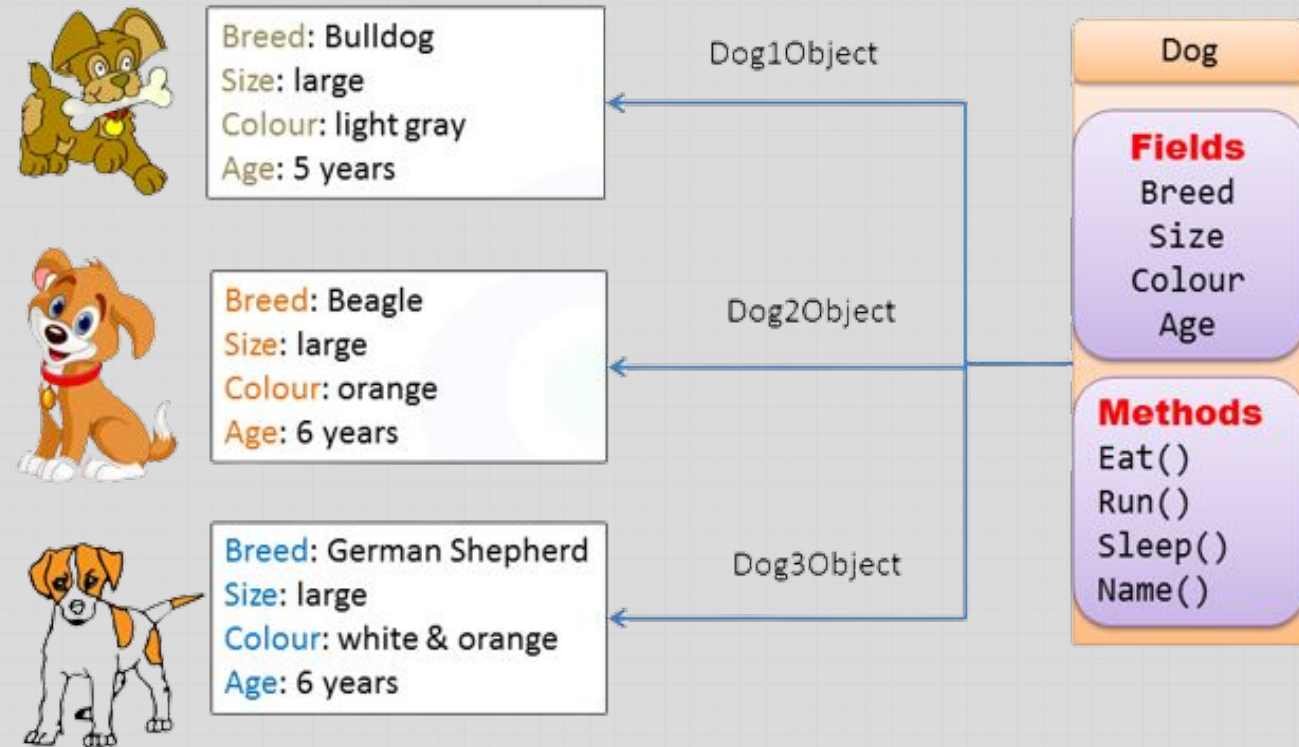
# What is A Class?

- Where **objects** came from
- A **blueprint** or set of instructions to build a specific type of Object
- No memory allocated for a class



# What is An Object?

- An **instance** of a class
- Multiple objects can be created from a class
- Each object has its **own** memory
- The data stored in an object are called **fields**





# Writing A Custom Class

**Class Name**

Dog

**Fields  
(Attributes)**

name  
breed  
size  
age  
color  
...

**Methods  
(Actions)**

eat()  
drink()  
play()  
...

keyword

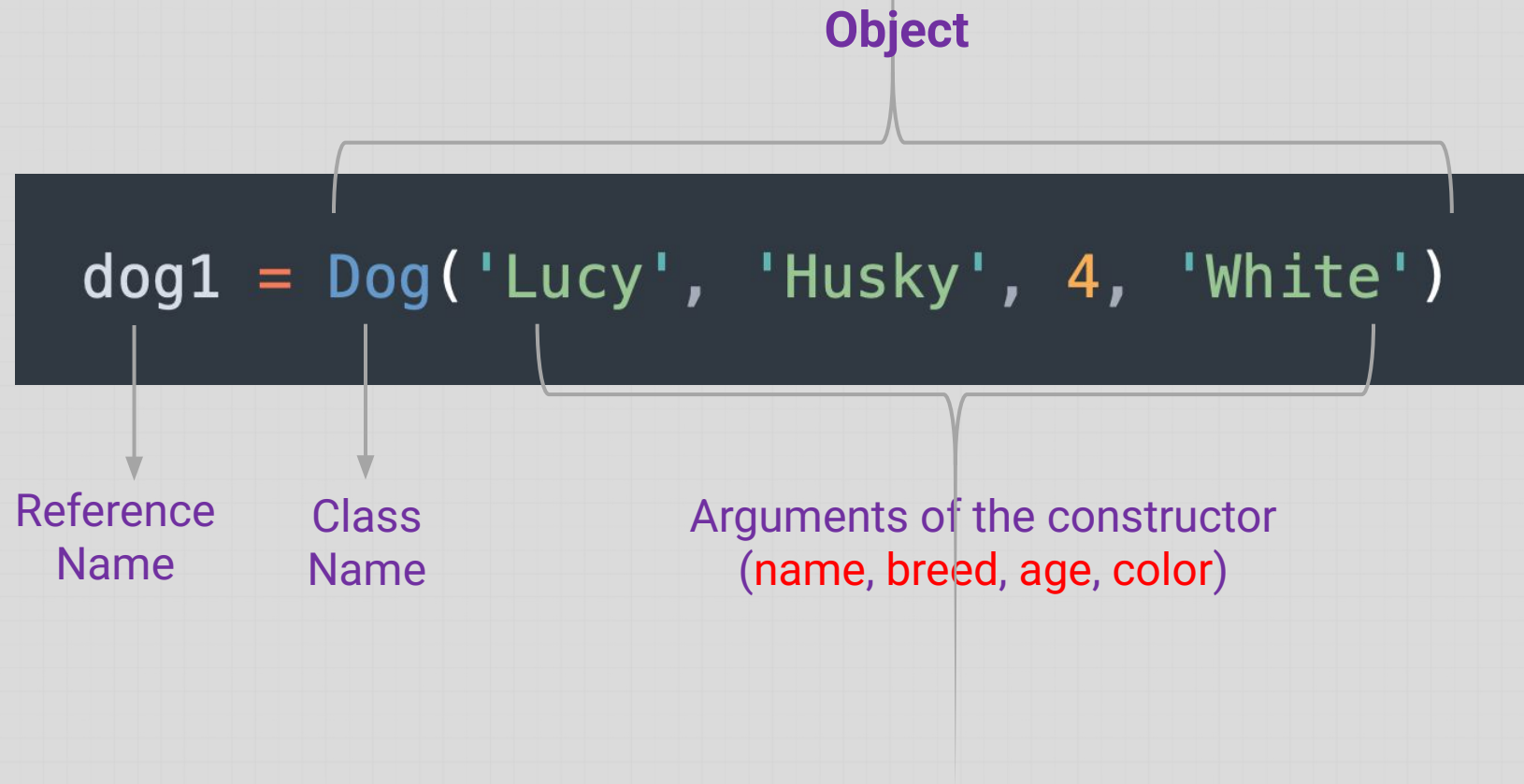
Class Name

`class Dog:`

```
def __init__(self, name, breed, age, color):  
    self.name = name  
    self.breed = breed  
    self.age = age  
    self.color = color  
  
def eat(self):  
    print(f'{self.name} is eating dog food')  
  
def drink(self):  
    print(f'{self.name} is drinking water')  
  
def play(self):  
    print(f'{self.name} is playing')
```



# Creating An Object/Instance



# The `__init__` Method

- Build-in `__init__` method used for defining & initializing the attributes
- Belongs to the object, and each object has its own memory
- Gets executed when an object is created from the class

```
class Dog:
    def __init__(self, name, breed, age, color):
        self.name = name
        self.breed = breed
        self.age = age
        self.color = color
```



# Object Methods

- Objects can share the methods created within the class
- Methods can be called through the object once it's instantiated

```
class Dog:

    def __init__(self, name, breed, age, color):
        self.name = name
        self.breed = breed
        self.age = age
        self.color = color

    def eat(self):
        print(f'{self.name} + is eating dog food')

    def drink(self):
        print(f'{self.name} + is drinking water')

    def play(self):
        print(f'{self.name} + is playing')
```

```
dog1 = Dog('Lucy', 'Husky', 4, 'White')
# Dog Object is created

dog1.eat()
dog1.drink()
dog1.play()
```



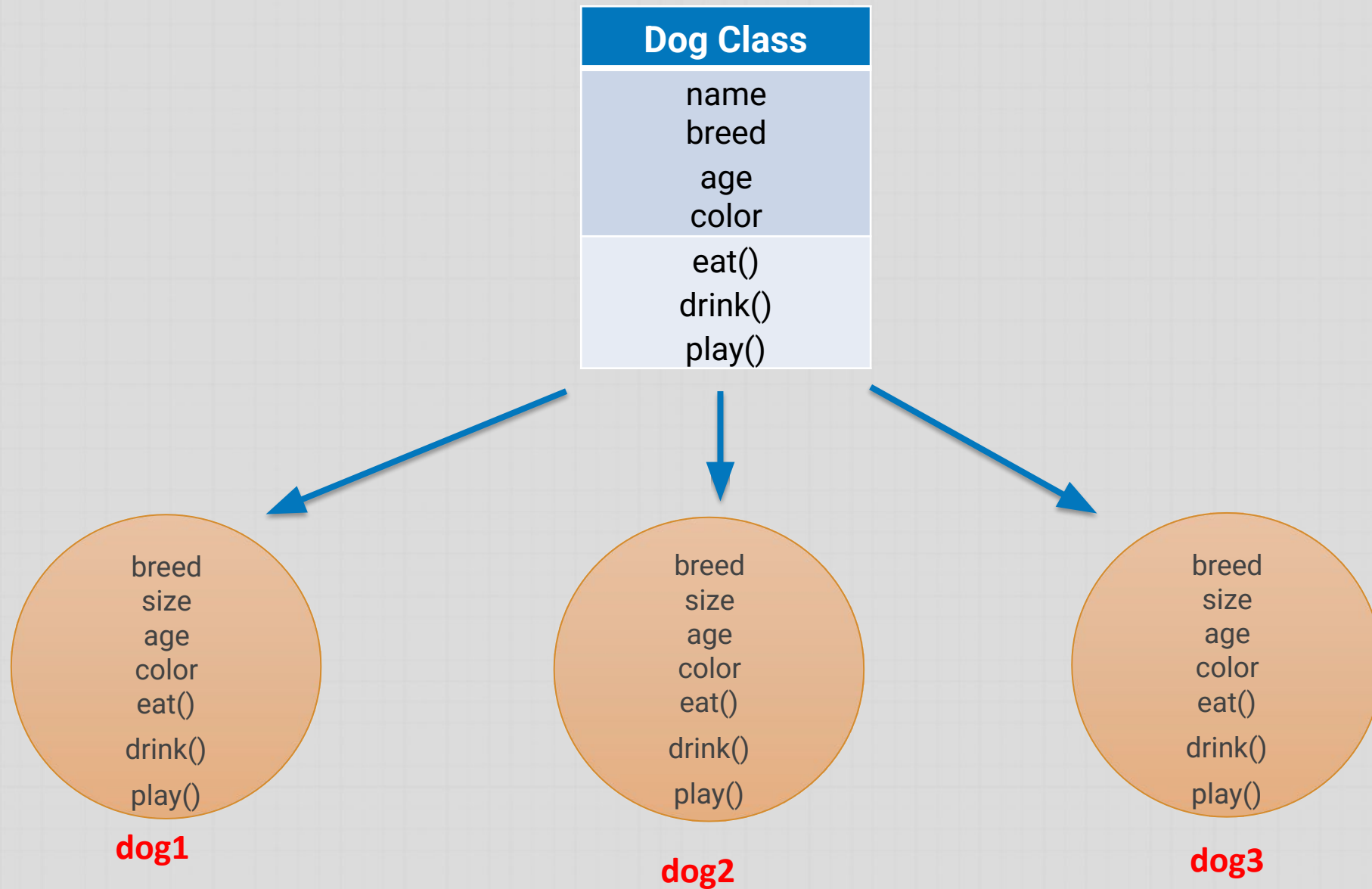
# The first parameter in object methods

- The first parameter **self** keyword  
references the instance of the class
- Used for accessing the attributes of  
the class

```
def __init__(self, name, breed, age, color):  
    self.name = name  
    self.breed = breed  
    self.age = age  
    self.color = color  
  
def eat(self):  
    print(f'{self.name} is eating dog food')  
  
def drink(self):  
    print(f'{self.name} is drinking water')  
  
def play(self):  
    print(f'{self.name} is playing')
```



# Topic Example



# Accessing an object's data and methods

- An Object's members refer to its data fields and methods. After the object is created its data can be accessed and its methods can be invoked using the **dot operator** (.)

```
dog1 = Dog('Lucy', 'Husky', 4, 'White')

print(dog1.name)
print(dog1.breed)
print(dog1.color)

dog1.eat()
dog1.drink()
dog1.play()
```





# The `__str()` Method

- Build-in `__str()` method is used for controlling what should be returned when the class object is represented as a string

```
class Dog:

    def __init__(self, name, breed, age, color):
        self.name = name
        self.breed = breed
        self.age = age
        self.color = color

    def __str__(self):
        return f'Dog[name:{self.name}, breed:{self.breed}]'
```

```
dog1 = Dog('Lucy', 'Husky', 4, 'White')

print(dog1)
# Dog[name:Lucy, breed:Husky]
```





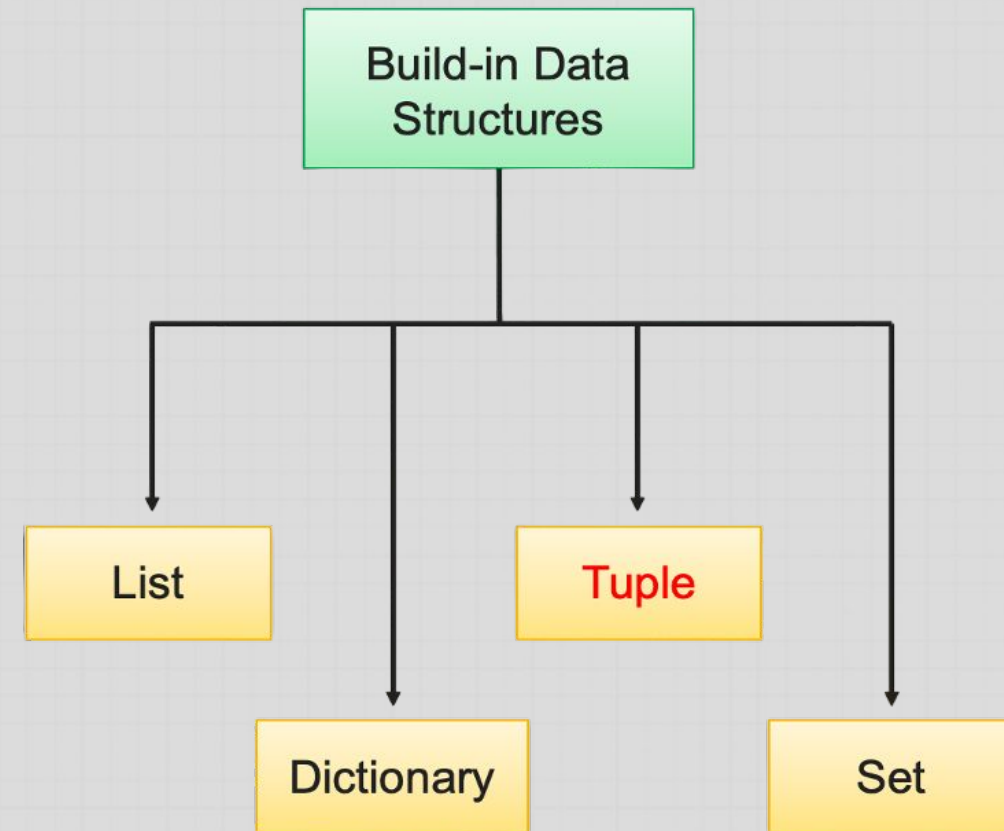
# Class vs Object

Class	Object
Class is a collection of similar objects	Object is an instance of a class
Class is conceptual (is a template)	Object is real
No memory is allocated for a class	Each object has its own memory
Class can exist without any objects	Objects can not exist without a class



# Tuple

- A special type of variable
- Used to store multiple values of any types
- Size is **fixed**, and can not be increased/decreased
- The values in the tuple are **unchangeable**
- Every element in a tuple has 2 index numbers:
  - **Forward index**
  - **Reverse index**



# Creating A Tuple

- Created by placing all the elements inside parentheses (**Optional**) separated by commas
- A comma **must** be given after the element, otherwise, it won't be recognized as a tuple
- Elements are **ordered**, **unchangeable**, can be **duplicated**, and can be of **any data type**

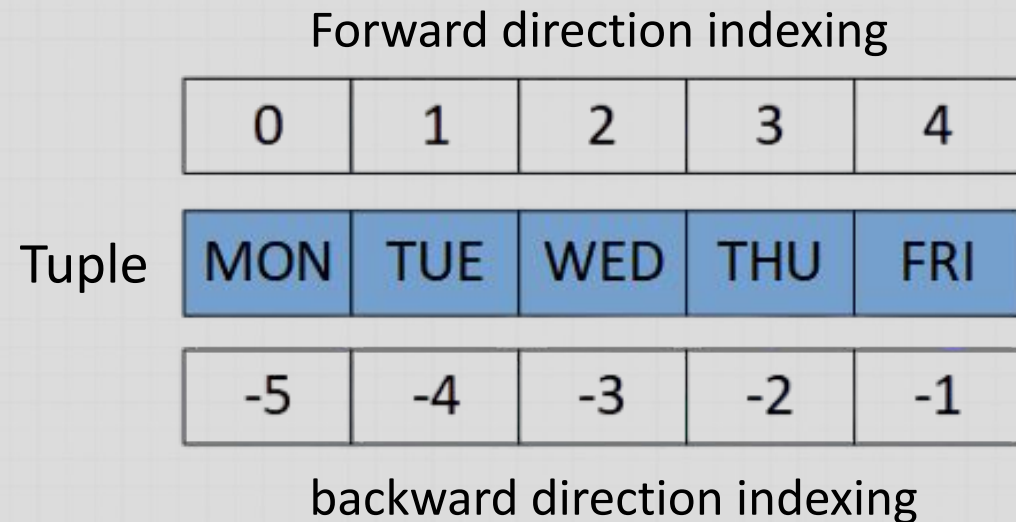
```
days = ("MON", "TUE", "WED", "THU", "FRI", 'SAT', 'SUN')
fruits = "Cherry", "Lemon", "Cherry", 'Orange', 'Kiwi'
scores = 75, 78, 85, 90, 93, 95, 85, 75, 86, 92, 85
tuple1 = ("A", 'B', 'C', 1, 2, 3, True, False)
```



# Accessing Tuple Elements

- Elements of a tuple can be accessed by using the square brackets `[ ]`
- The index number (forward/backward index) of the element needs to be provided

```
days = ("MON", 'TUE', "WED", "THU", 'FRI')  
  
print(days[0]) # prints "MON"  
  
print(days[-1]) # prints "FRI"
```



# Slicing Tuple: Start & End Indexes

- Creates a range of elements (**sub-tuples**) by using the slice syntax **[start : end]**
- We can specify the start index and end index (**excluded**), separated by a colon, to create the sub-tuples

```
days = ("MON", "TUE", "WED", "THU", "FRI")  
work_days = days[2:5]   
print(work_days) # prints ('WED', 'THU', 'FRI')  
good_days = days[1:4]   
print(good_days) # prints ('TUE', 'WED', 'THU')
```

Slices the tuple days starting from index 2 to index 5 (**index 5 is excluded**)

Slices the tuple days starting from index 1 to index 4 (**index 4 is excluded**)



# Slicing Tuple: Slice From The Start

- By not giving the **start** index, the slicing starts from the first element **[ : end ]**

```
days = ("MON", "TUE", "WED", "THU", "FRI")
```

```
work_days = days[:3]  
print(work_days) # prints ('MON', 'TUE', 'WED')
```

Slices the tuple days starting from index 0 to index 3 (**index 3 is excluded**)

```
numbers = 10, 20, 30, 40, 50, 60
```

```
some_nums = numbers[:3]  
print(some_nums) # prints (10, 20, 30)
```

Slices the tuple numbers starting from index 0 to index 3 (**index 3 is excluded**)





# Slicing Tuple: Slice To The End

- By not giving the **end** index, the slicing starts from the start index to the end of the tuple **[start : ]**

```
days = ("MON", "TUE", "WED", "THU", "FRI")
```

```
work_days = days[3:]  
print(work_days) # prints ('THU', 'FRI')
```

Slices the tuple days starting from index 3 to the **end**

```
numbers = 10, 20, 30, 40, 50, 60
```

```
some_nums = numbers[3:]  
print(some_nums) # prints (40, 50, 60)
```

Slices the tuple numbers starting from index 3 to the **end**





# Merging Tuples

- To merge two or more tuples, we can use the **+** operator to merge them

```
tuple1 = 10, 20, 30
tuple2 = (40, 50, 60)

numbers = tuple1 + tuple2

print(numbers)
# prints (10, 20, 30, 40, 50, 60)
```

```
tuple1 = ("Cherry", "Apple", "Banana")
tuple2 = ("Orange", "Wooden Spoon", "Lemon")
tuple3 = ("Eggs", "Milk", "Salt", "Sugar")

items = tuple1 + tuple2 + tuple3
```



# Multiplying Tuples

- To multiply the content of a tuple, we can use `*` operator

```
tuple1 = (1, 2, 3)

new_tuple = tuple1 * 2

print(new_tuple)
# prints (1, 2, 3, 1, 2, 3)
```

```
tuple1 = ("Python",)

new_tuple = tuple1 * 2

print(new_tuple)
# prints ('Python', 'Python')
```



# Tuple Methods

- Tuple has the following two build-in methods:
  - `index()`: returns the forward index number of a specified element from the tuple
  - `count()`: returns the frequency of a specified element from the tuple

```
numbers = (10, 20, 30, 40, 50, 60, 70, 10, 10, 10)

print(numbers.index(30)) # prints 2
print(numbers.index(60)) # prints 5

print(numbers.count(10)) # prints 4
print(numbers.count(200)) # prints 0
```

