ITE3162 – PYTHON FUNDAMENTALS

PELIN

OUTLINES

Python installation

Python data types, variables and user inputs

How Python defines data types (integers, strings, etc.) and how to convert between them

Create interactive program has a conversation with the user or data

PYTHON INSTALLATION & DEVELOPMENT ENVIRONMENT SETUP

https://python.org

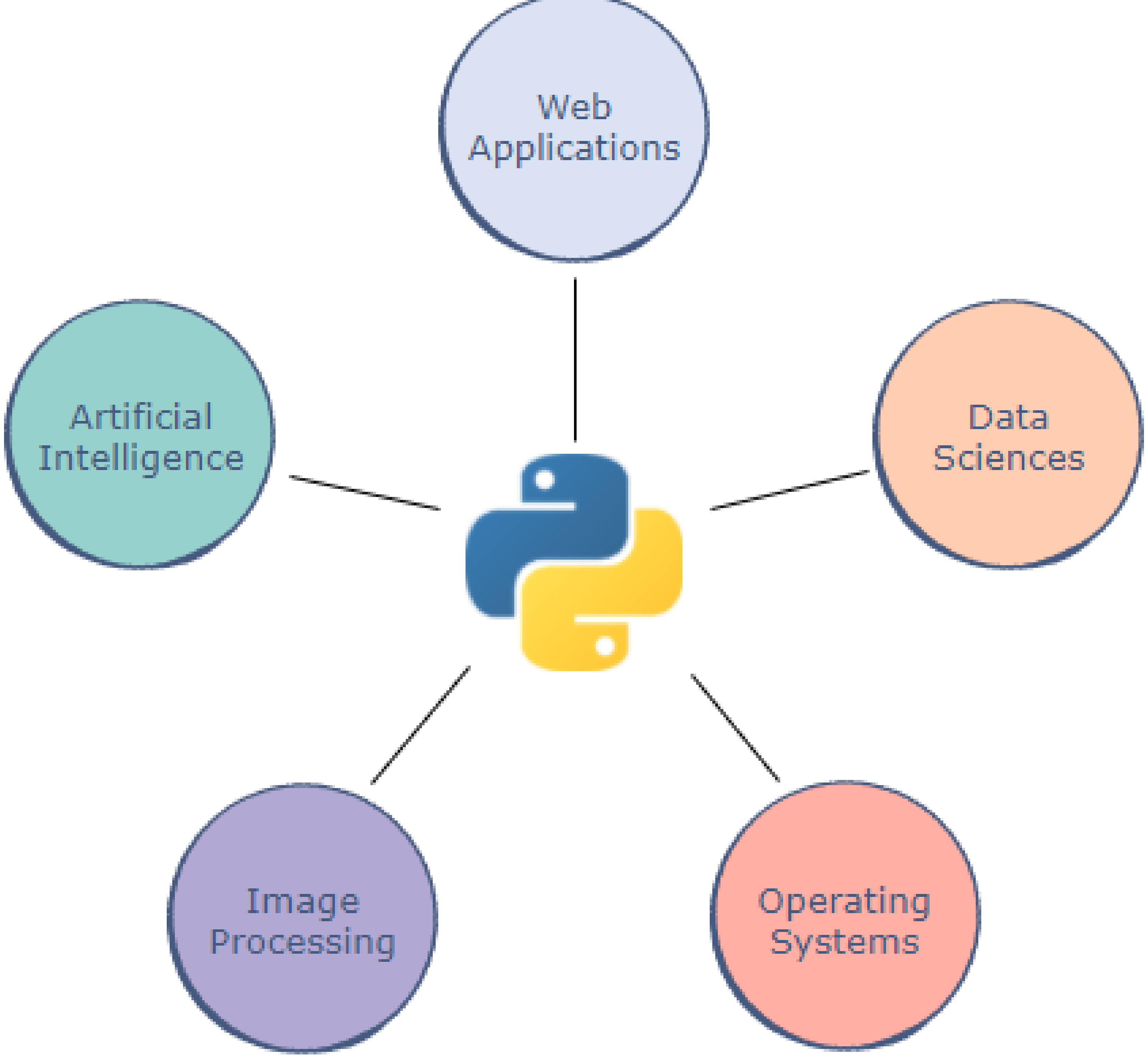
https://code.visualstudio.com/

WHAT IS PYTHON?

Developed in 1990, Python is one of the most popular general-purpose programming languages in modern times.

Characteristics of python:

- A High-level language
- Readability
- Applications



PYTHON VERSIONS

Python has had several major updates in the past.

Python 2.7 was widely used for a very long time, even after the release of newer versions

Python 2.7 has been deprecated as of January 01 2020, and replaced completely by 3.xx

To keep up with the latest technologies, we'll be dealing with Python 3 for the entirety of this course.

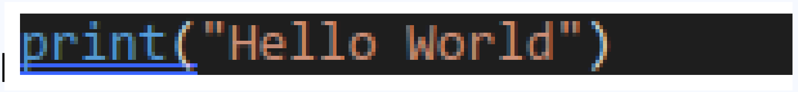
WRITING OUR FIRST CODE - **THE PRINT STATEMENT**

Every language has a different syntax for displaying or printing something on the screen

We can print data on the terminal by simply using the print statement.

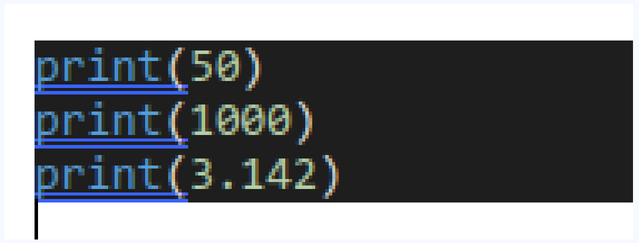
```
print (data)
```

Let's try printing "Hello World" on the termi



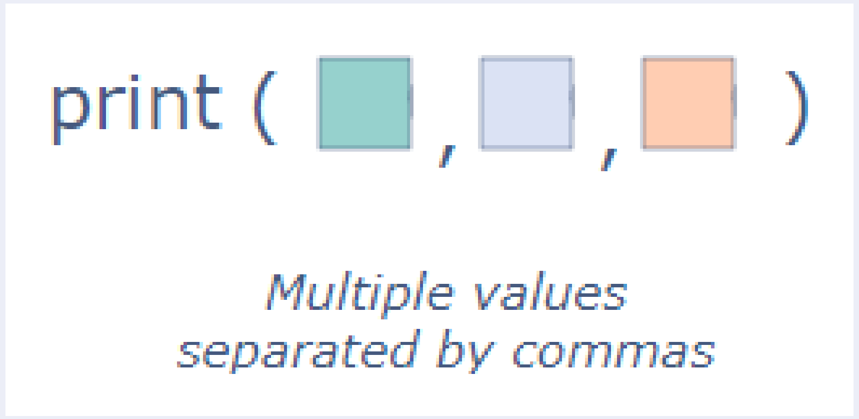
```python
print("Hello World")
```

Next, we'll print a few numbers. Each call to print moves the output to a new line:



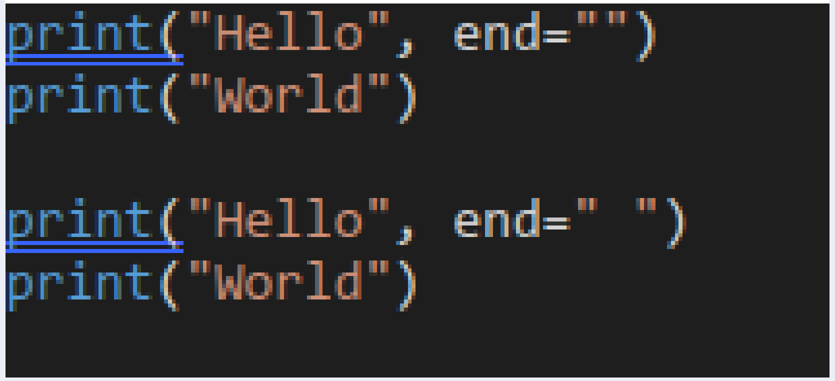
```python
print(50)
print(1000)
print(3.142)
```

PRINTING MULTIPLE PIECES OF DATA



Printing multiple things in a single print command; we just must separate them using **commas**

If we want multiple print statements to print in the same line, we use "end" parameter

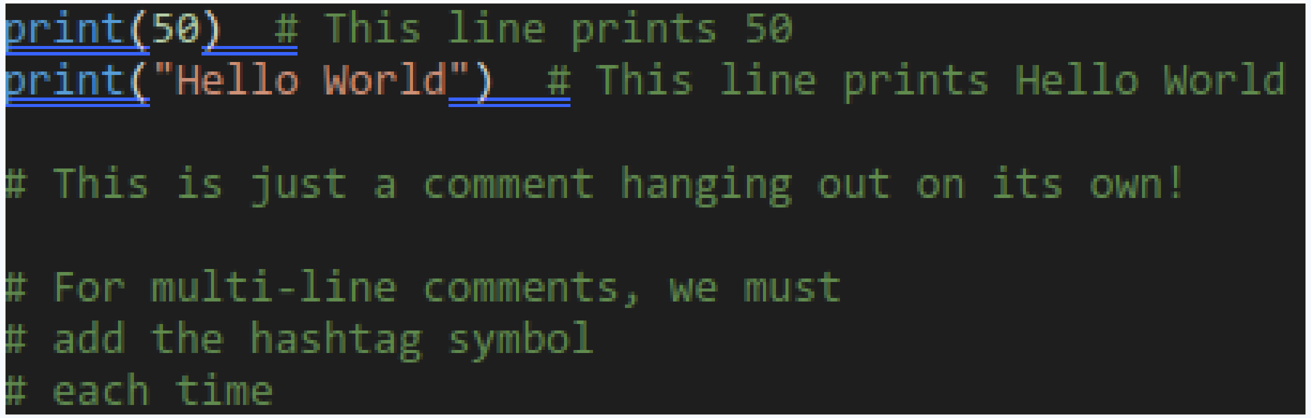


```
print("Hello", end="")
print("World")

print("Hello", end=" ")
print("World")
```

COMMENTS

Comments are pieces of text used to describe what is happening in the code.

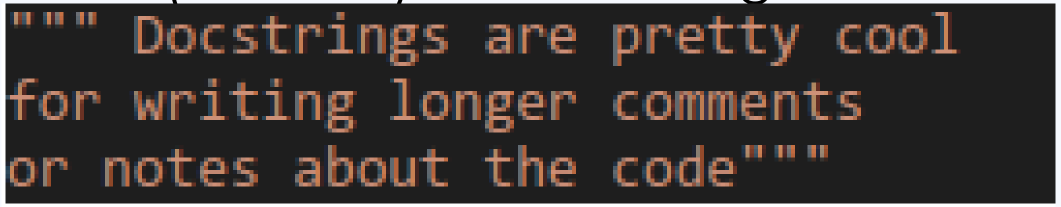
A comment can be written using the **#** character:



```
print(50)  # This line prints 50
print("Hello World")  # This line prints Hello World

# This is just a comment hanging out on its own!

# For multi-line comments, we must
# add the hashtag symbol
# each time
```

An alternative to these multi-line comments (line 4 - 8) are docstrings.

They are encased in triple quotes, """



```
""" Docstrings are pretty cool
for writing longer comments
or notes about the code"""
```

EXAMPLE & EXERCISES

Self-check

1. What is the command to show the installed version of python?
2. How do you open vscode in your current directory
3. What do you think the following code will print " **print(Let's begin)** "
4. How do we run python programs in visual studio code?

Exercise

Task:

1. Use vscode to create a python file with your name
2. Write a two lines of code that outputs your name and the phrase "python is fun" on one line.
3. Execute the program and see the output
4. Send it to the trainer for assessment

DATA TYPES AND VARIABLES

"The data type of an item defines the type and range of values that item can have".

"A variable is simply a name to which a value can be assigned".

PYTHON'S DATA TYPES

Unlike many other languages, Python does not place a strong emphasis on defining the data type of an object, which makes coding much simpler.

The language provides three main data types:

Numbers

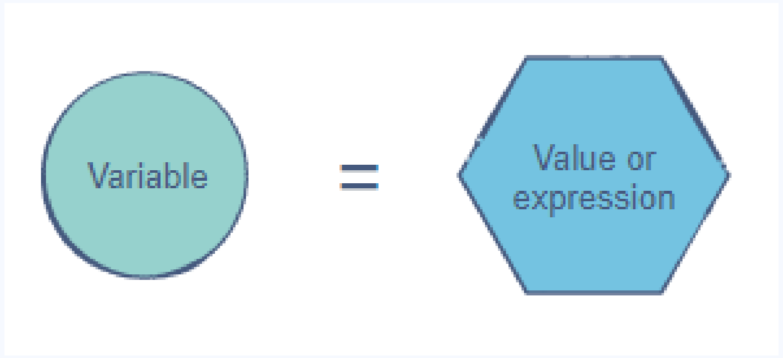
Strings

Booleans

VARIABLES

Variables allow us to give meaningful names to data

The simplest way to assign a value to a variable is through the **=** operator



Variables are **mutable**

Variables allow us to store data so that we can use it later to perform operations in the code

NAMING CONVENTION

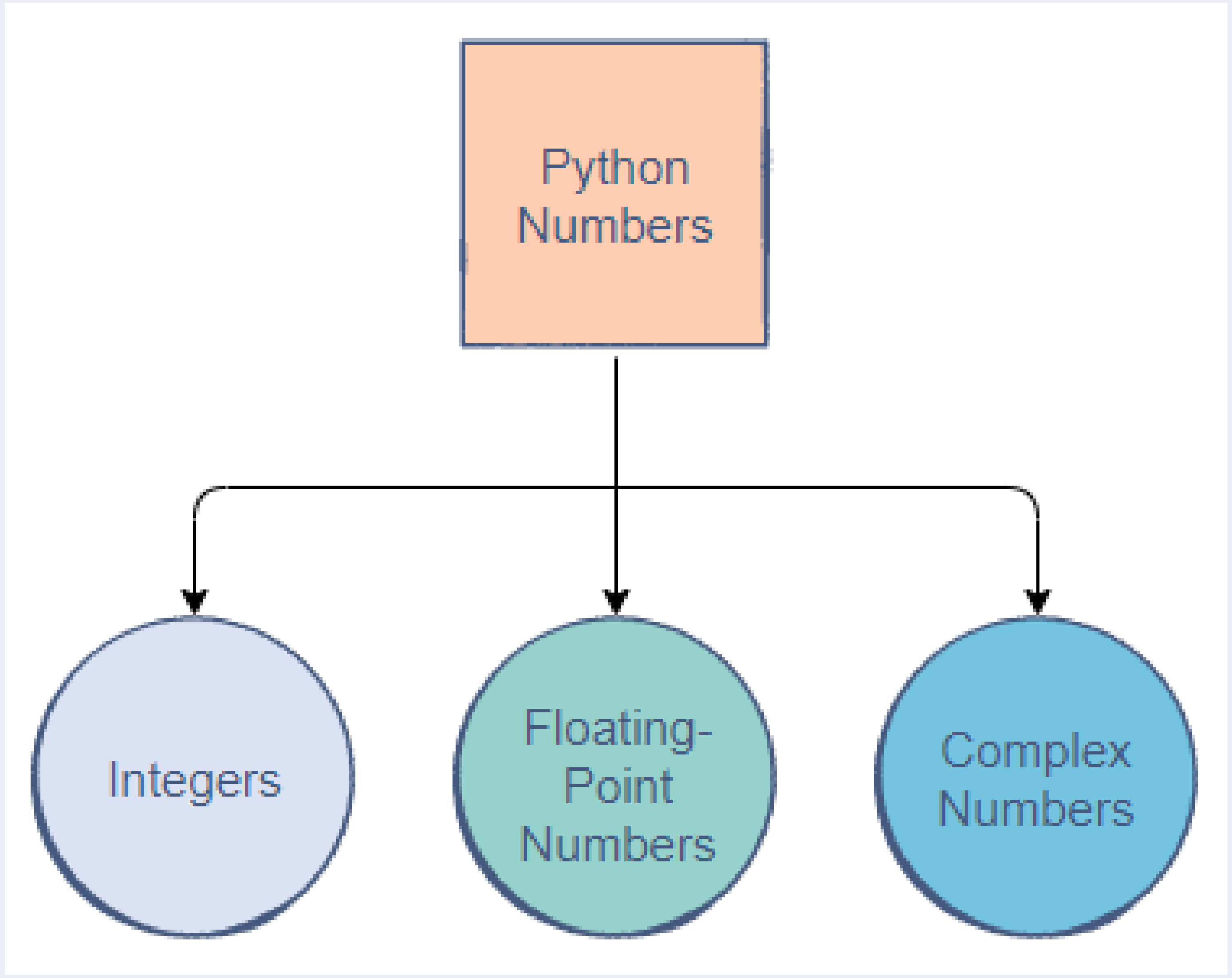
There are certain rules we have to follow when picking the name for a variable:

- The name can start with an upper or lower case alphabet.
- A number can appear in the name, but not at the beginning.
- The _ character can appear anywhere in the name.
- Spaces are not allowed. Instead, we must use ***snake_case*** to make variable names readable.
- The name of the variable should be something meaningful that describes the value it holds, instead of being random characters.

1. NUMBERS

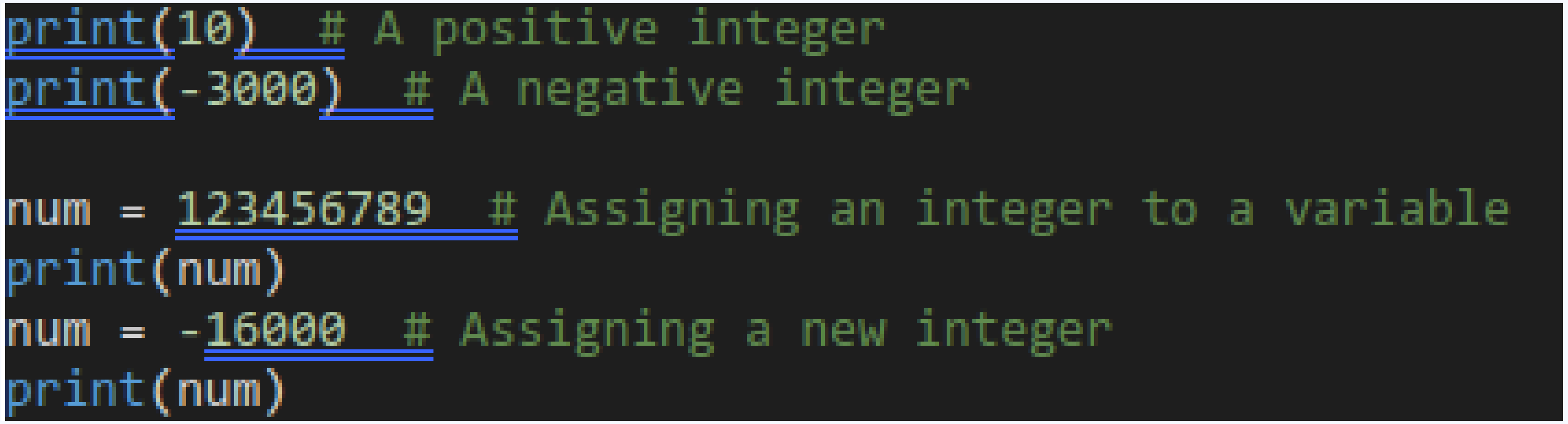
Python is one of the most powerful languages when it comes to manipulating numerical data

There are three main types of numbers in Python:



INTEGER NUMBERS

The integer data type is comprised of all the positive and negative whole numbers



```
print(10)   # A positive integer
print(-3000)   # A negative integer

num = 123456789   # Assigning an integer to a variable
print(num)
num = -16000   # Assigning a new integer
print(num)
```

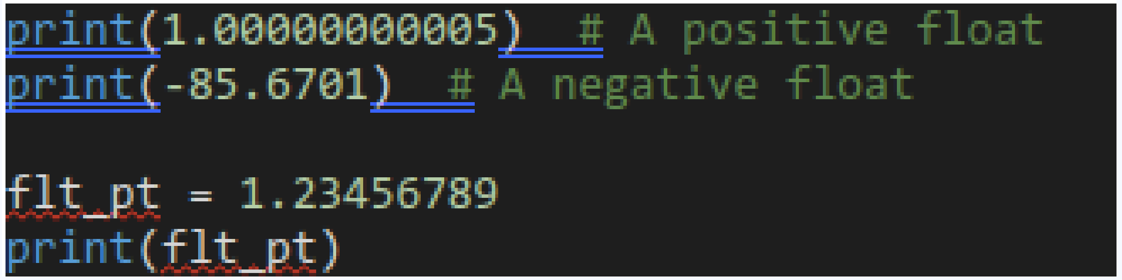
Note: In Python, all negative numbers start with the - symbol.

FLOATING POINT NUMBERS

Floating-point numbers, or floats, refer to positive and negative decimal numbers

Python allows us to create decimals up to a very high decimal place

This ensures accurate computations for precise values.



```
print(1.00000000005)  # A positive float
print(-85.6701)  # A negative float

flt_pt = 1.23456789
print(flt_pt)
```

In Python, **5** is considered to be an integer while **5.0** is a float

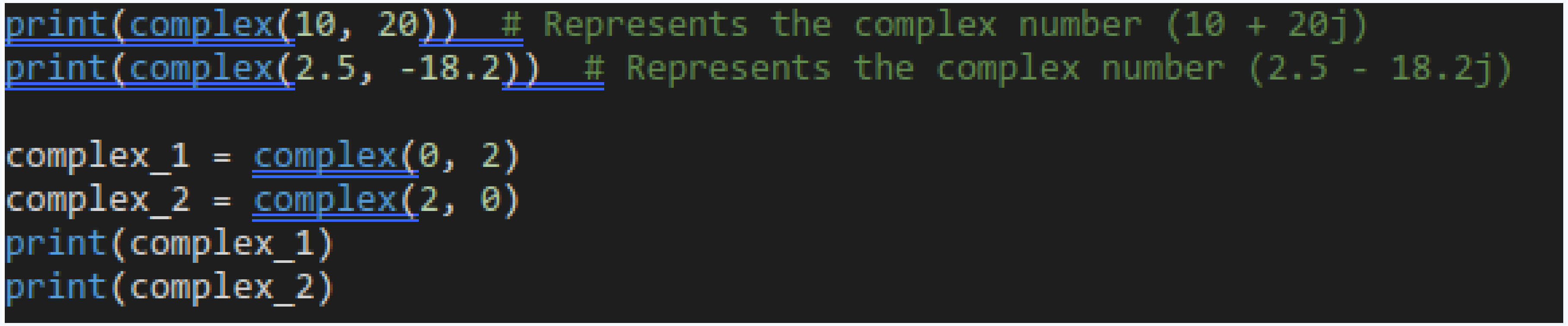
COMPLEX NUMBERS

Python also supports complex numbers

Complex numbers are made up of a real and an imaginary part.

Just like the ***print()*** statement is used to print values, ***complex()*** is used to create complex numbers

complex(real, imaginary)



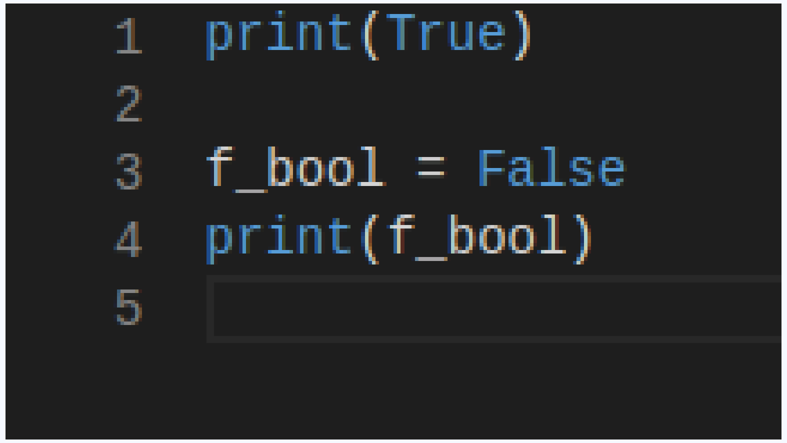
```
print(complex(10, 20))  # Represents the complex number (10 + 20j)
print(complex(2.5, -18.2))  # Represents the complex number (2.5 - 18.2j)

complex_1 = complex(0, 2)
complex_2 = complex(2, 0)
print(complex_1)
print(complex_2)
```

BOOLEANS

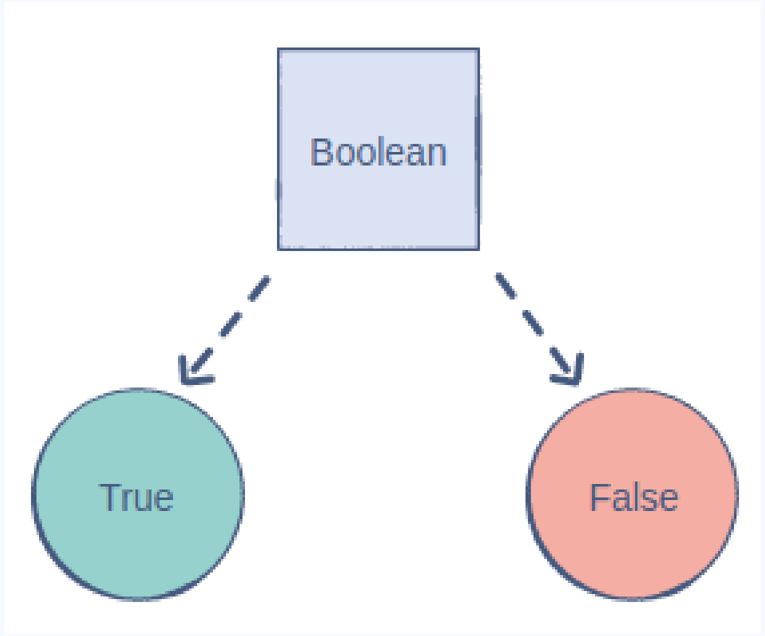
The Boolean (also known as bool) data type allows us to choose between two values: true and false

A Boolean is used to determine whether the logic of an expression or a comparison is correct.

It plays a huge role in data comparisons



```
print(True)

f_bool = False
print(f_bool)

```

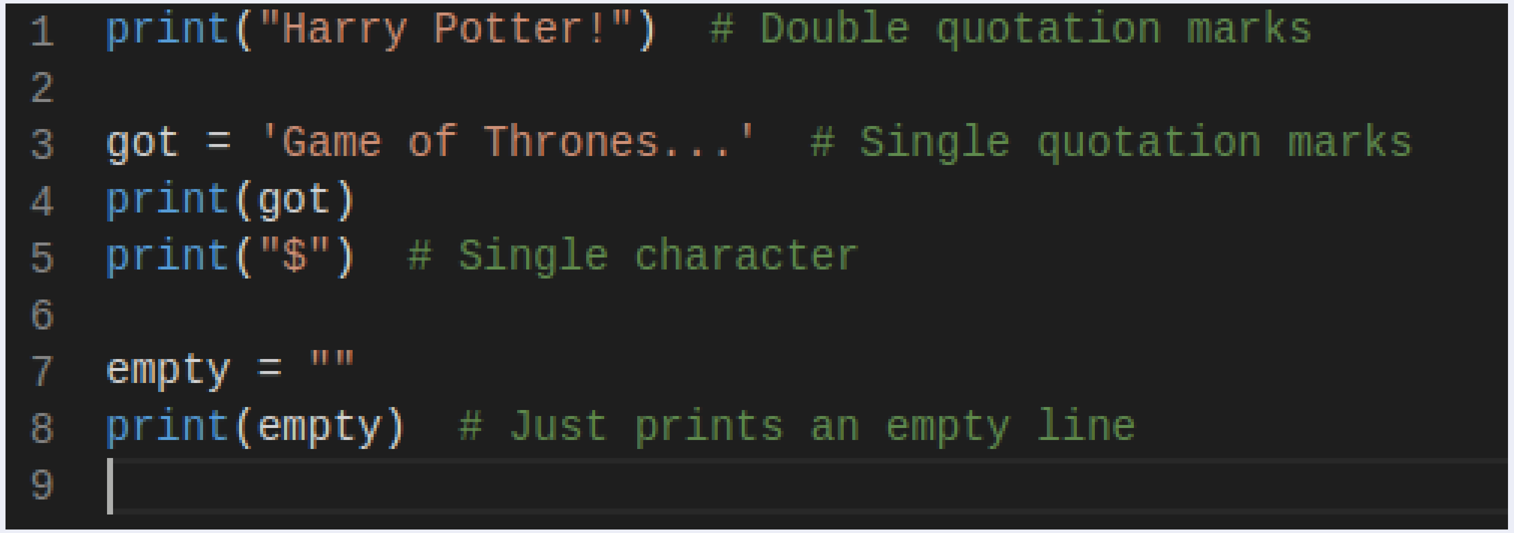


STRINGS

A string is a collection of characters closed within single or double quotation marks.

A group of characters such as **"Hello World"** is an example of the string data type.

A string can also contain a single character or be entirely empty

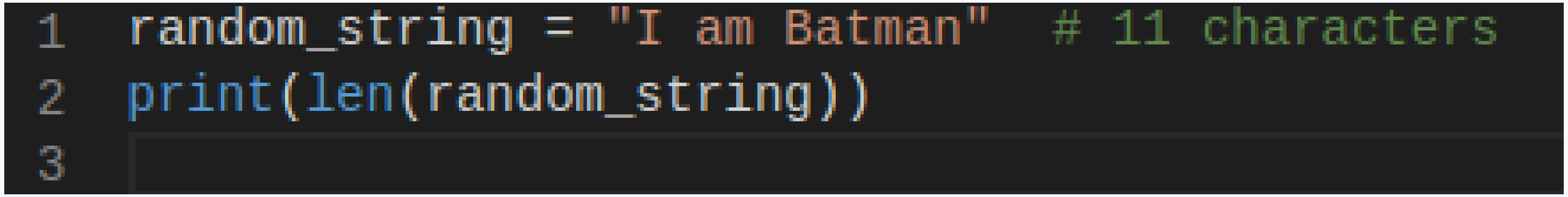


```python
print("Harry Potter!")  # Double quotation marks

got = 'Game of Thrones...'  # Single quotation marks
print(got)
print("$")  # Single character

empty = ""
print(empty)  # Just prints an empty line

```

THE LENGTH OF A STRING

The length of a string can be found using the **len** statement

This length indicates the number of characters in the string:

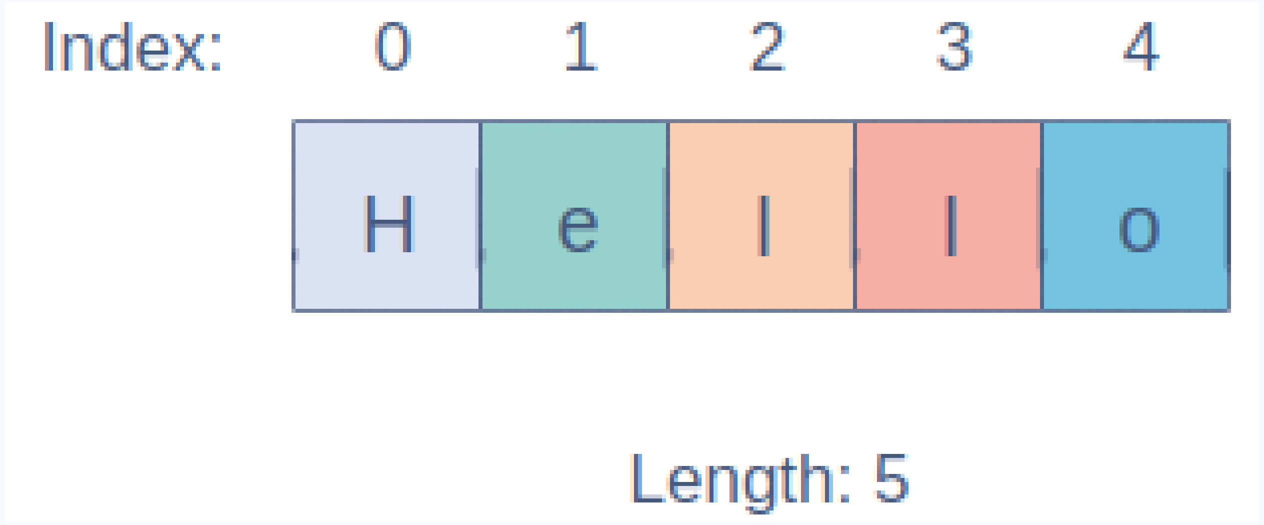


```python
random_string = "I am Batman"  # 11 characters
print(len(random_string))

```

STRING INDEXING

In a string, every character is given a numerical **index** based on its position.

A string in Python is indexed from 0 to n-1 where n is its length



MANIPULATING STRINGS

Accessing Characters:

- Each character in a string can be accessed using its index.
- The index must be closed within square brackets, [], and appended to the string

Reverse Indexing:

- We can also change our indexing convention by using negative indices
- Negative indices start from the opposite end of the string

String Slicing: Slicing is the process of obtaining a portion (substring) of a string by using its indices

```
string[start:end]
```

MANIPULATING STRINGS …..

Slicing with a Step:

- we can define a **step** through which we can skip characters in the strin
- The default step is 1

```
string[start:end:step]
```

Reverse Slicing: Strings can also be sliced to return a reversed substring.

- In this case, we would need to switch the order of the **start** and **end** indices.

Partial Slicing: One thing to note is that specifying the **start** and **end** indices is optional.

If **start** is not provided, the substring will have all the characters until the **end** index.

- If **end** is not provided, the substring will begin from the **start** index and go all the way to the end

EXAMPLE & EXERCISES

Given the following variable

my_string = "This is My String !"

print(my_string[0])

Print(my_string[len(my_string)])

How would I print the last ! Mark in my_string variable

print(my_string[len(my_string) - 1])

print(my_string[-1])

print(my_string[1:7])

print(my_string[0:7:2])

print(my_string[13:2:-1])

Print(my_string[17:0:-2])

print(my_string[:])

print(my_string[:: -1])

GETTING USER INPUT

The Input is nothing but some value from a system or user.

In Python, we have the following two functions to handle input from a user and system

- **input(prompt)** to accept input from a user
- **print()** to display output on the console.

Example:

```
name = input("Enter Employee Name ")
salary = input("Enter salary ")
company = input("Enter Company name ")

print("\n")
print("Printing Employee Details")
print("Name", "Salary", "Company")
print(name, salary, company)
```

DATA TYPE CONVERSION

When programming, there are times we need to convert values between types in order to manipulate values in a different way

- **Converting Integers to Floats :** we use float() method
- **Converting Floats to Integers:** we use int() method
- **Converting Numbers to Strings:** we use str() method
- **Converting Strings to Numbers:** we use the int() and float() methods.

EXAMPLE & EXERCISES

1. Write a simple program that prints the total number of characters of a string entered by the user.

CONTROL STRUCTURE IN PYTHON

OUTLINES

Python Operators

Conditional Statements

PYTHON OPERATORS

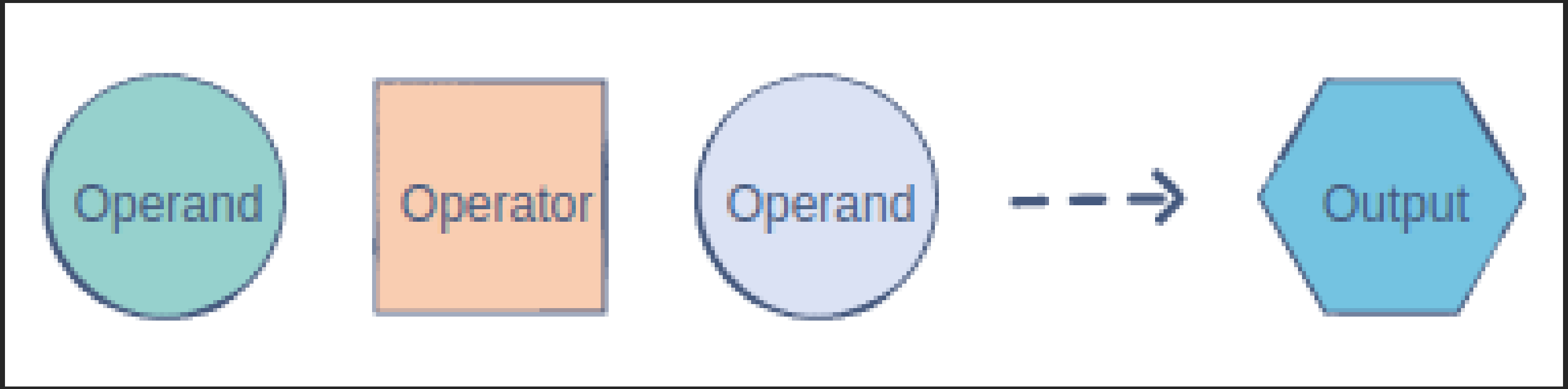
"**Operators** are used to perform arithmetic and logical operations on data. They enable us to manipulate and interpret data to produce useful outputs."

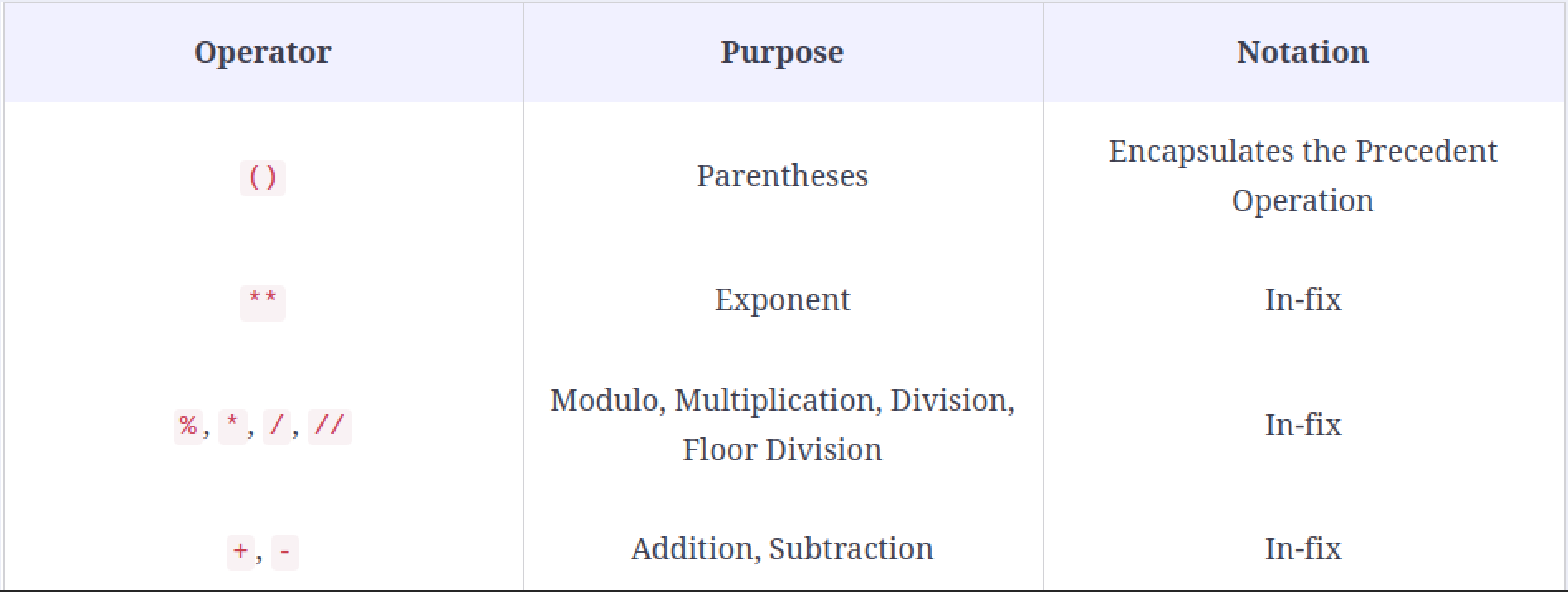
OPERATORS

Python operators follow the **in-fix** or **prefix** notations

The 5 main operator types in Python are:

- arithmetic operators
- comparison operators
- assignment operators
- logical operators
- bitwise operators

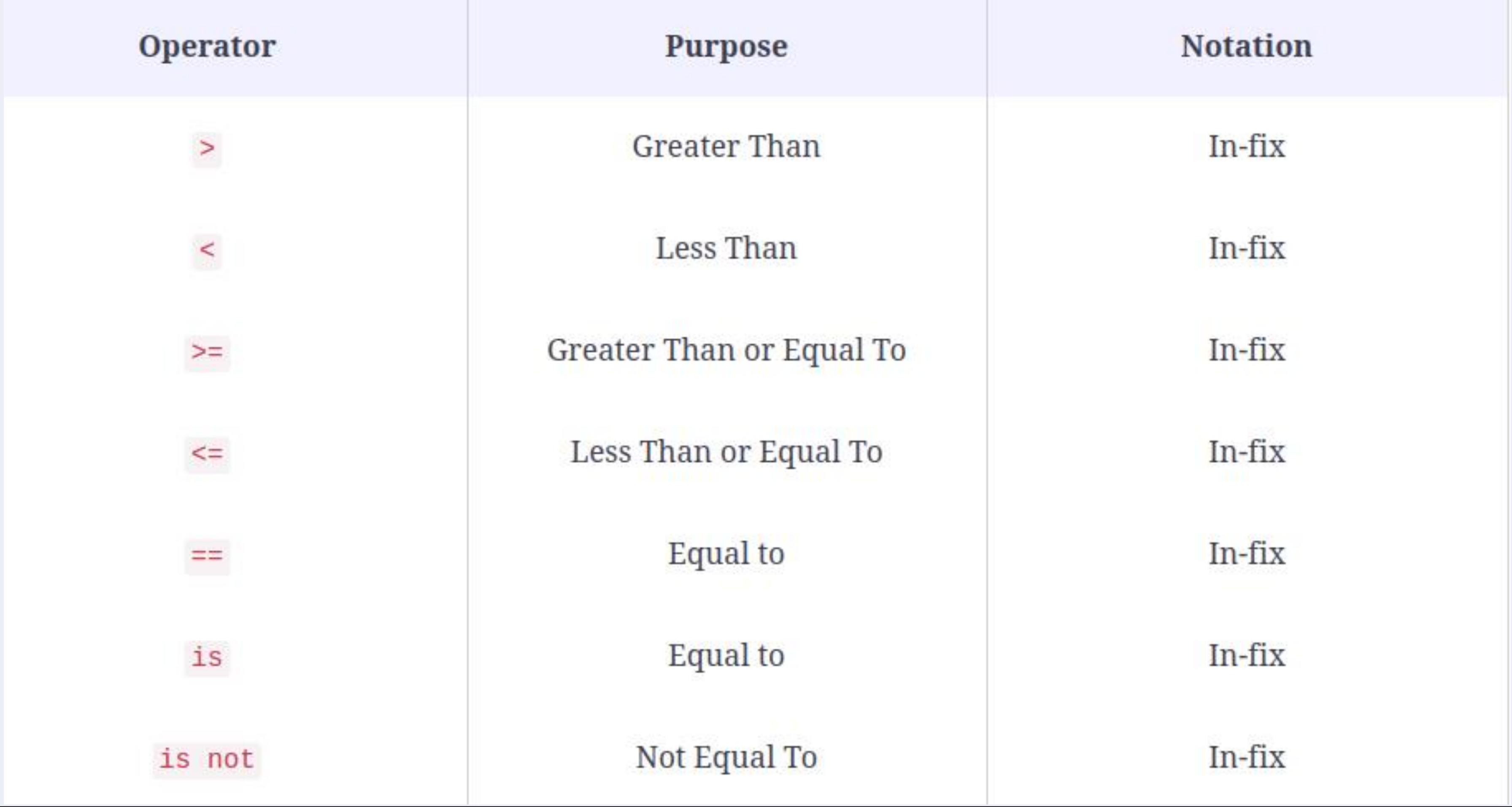




Operator	Purpose	Notation
`()`	Parentheses	Encapsulates the Precedent Operation
`**`	Exponent	In-fix
`%`, `*`, `/`, `//`	Modulo, Multiplication, Division, Floor Division	In-fix
`+`, `-`	Addition, Subtraction	In-fix

ARITHMETIC OPERATORS

THE OPERATOR LISTED HIGHER WILL BE COMPUTED FIRST



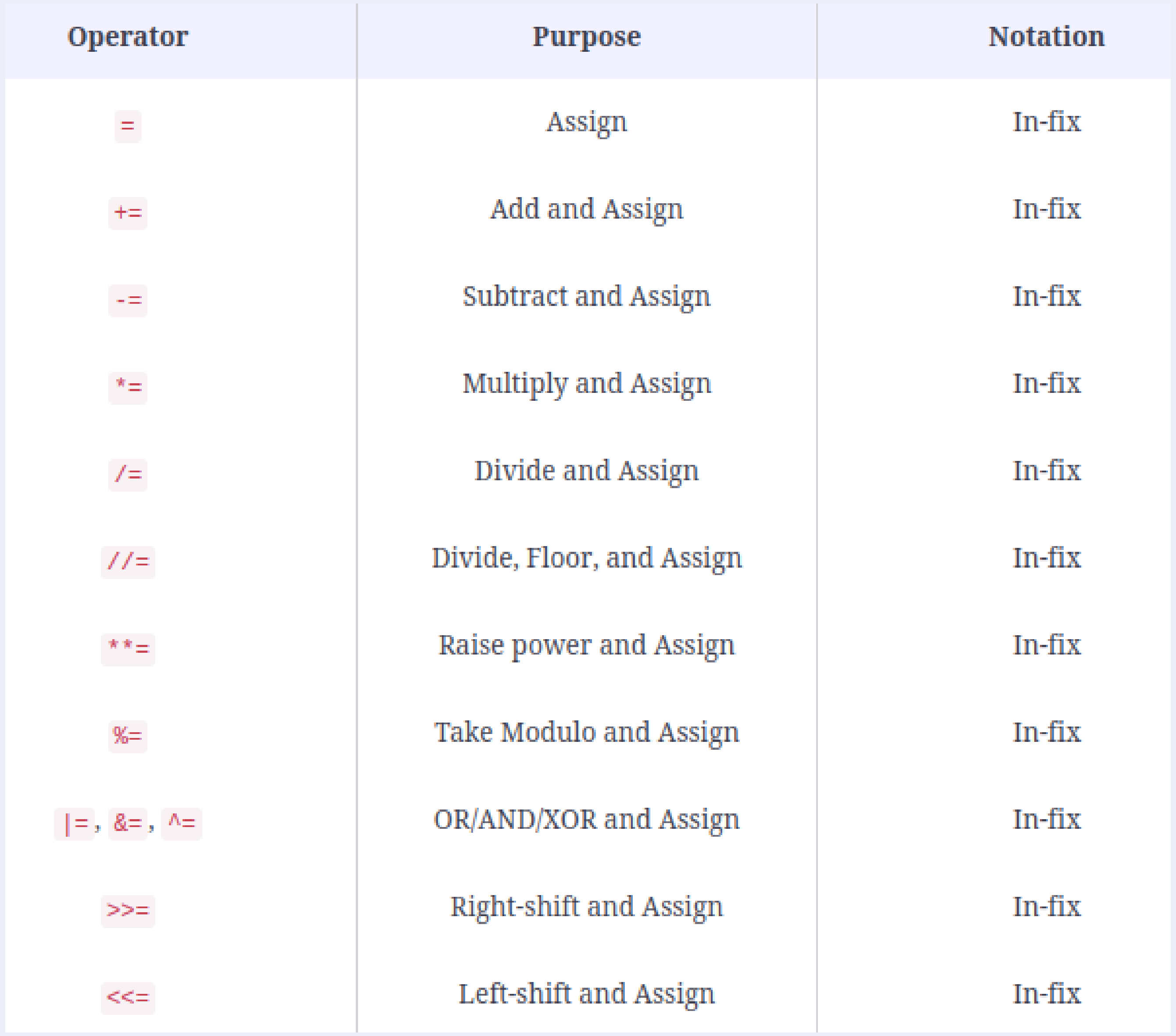
Operator	Purpose	Notation
`>`	Greater Than	In-fix
`<`	Less Than	In-fix
`>=`	Greater Than or Equal To	In-fix
`<=`	Less Than or Equal To	In-fix
`==`	Equal to	In-fix
`is`	Equal to	In-fix
`is not`	Not Equal To	In-fix

COMPARISON OPERATORS

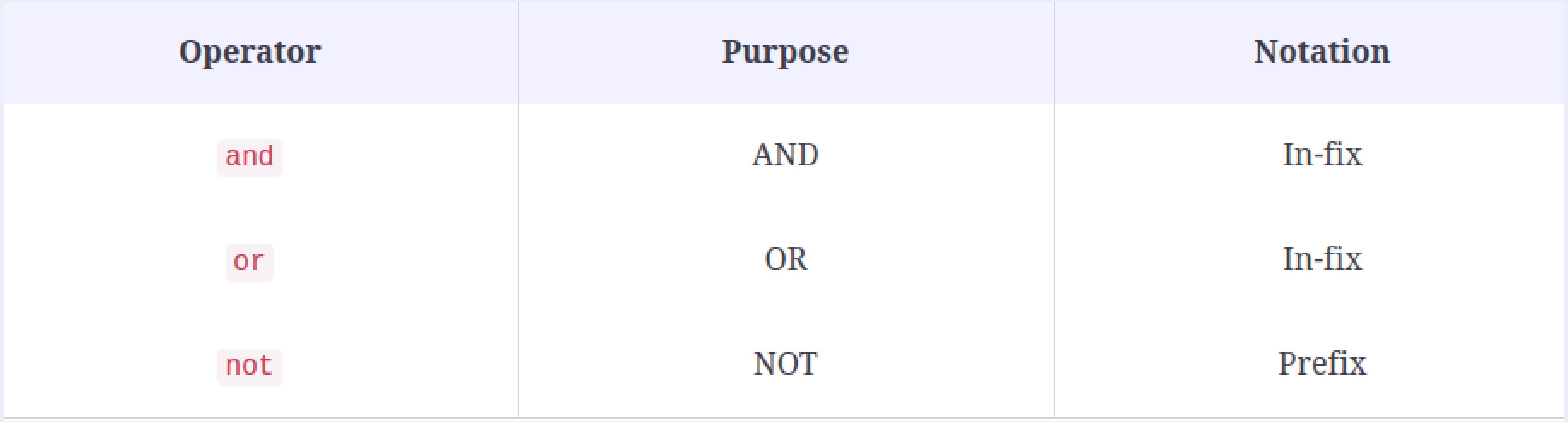
CAN BE USED TO COMPARE VALUES IN MATHEMATICAL TERMS.

ASSIGNMENT OPERATORS

USED TO ASSIGN VALUES TO A VARIABLE.



Operator	Purpose	Notation
=	Assign	In-fix
+=	Add and Assign	In-fix
-=	Subtract and Assign	In-fix
*=	Multiply and Assign	In-fix
/=	Divide and Assign	In-fix
//=	Divide, Floor, and Assign	In-fix
**=	Raise power and Assign	In-fix
%=	Take Modulo and Assign	In-fix
|=, &=, ^=	OR/AND/XOR and Assign	In-fix
>>=	Right-shift and Assign	In-fix
<<=	Left-shift and Assign	In-fix



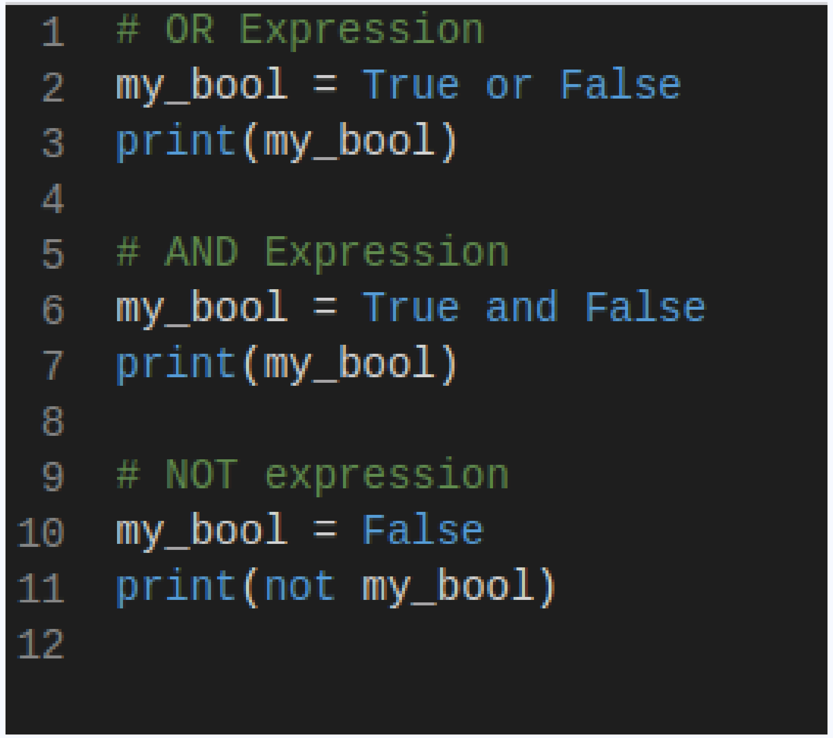
Operator	Purpose	Notation
`and`	AND	In-fix
`or`	OR	In-fix
`not`	NOT	Prefix

LOGICAL OPERATORS

CAN BE USED TO COMPARE VALUES IN MATHEMATICAL TERMS.

LOGICAL EXPRESSIONS

Logical operators are used to manipulate the logic of *Boolean expressions*



```python
# OR Expression
my_bool = True or False
print(my_bool)

# AND Expression
my_bool = True and False
print(my_bool)

# NOT expression
my_bool = False
print(not my_bool)

```

STRING OPERATIONS

The string data type has numerous utilities that make string computations much easier

Comparison Operators : Strings are compatible with the comparison operators

When two strings have different lengths, the string which comes first in the dictionary is said to have the smaller value.

Concatenation: The **+** operator can be used to merge two strings together

The ***** operator allows us to multiply a string, resulting in a repeating pattern

Search: The **in** keyword can be used to check if a particular substring exists in another string. If the substring is found, the operation returns **true**

EXAMPLE & EXERCISES

1. Assuming that variables have the following boolean values:

a = True
b = False
c = a and not b

Enter the result of evaluating the expression:

a and (not c or b)

Exercise

1. Please, print() the result of the following expression: raise **31** to the power of **331** and then get the remainder of the result's division by **20**.

2. What is the result of the following Python expression? **Print(((3 + 5) // 2 * 2 ** 3) % 7)**.

3. Find out if the result of dividing A by B is an odd number.

   Sample Input 1: 99 & 3 **Sample Output 1:** True

EXERCISES

Exercise

The movie theater has cinema halls that can accommodate a certain number of viewers each. Figure out if a movie theater can hold a given number of viewers that plan to visit it on a particular day.

The input format

The first line is number of halls, the second line is their capacity, and the third line is the planned number of viewers.

The output format

True or False.

Sample Input 1:

9

68

589

Sample Output 1:

True

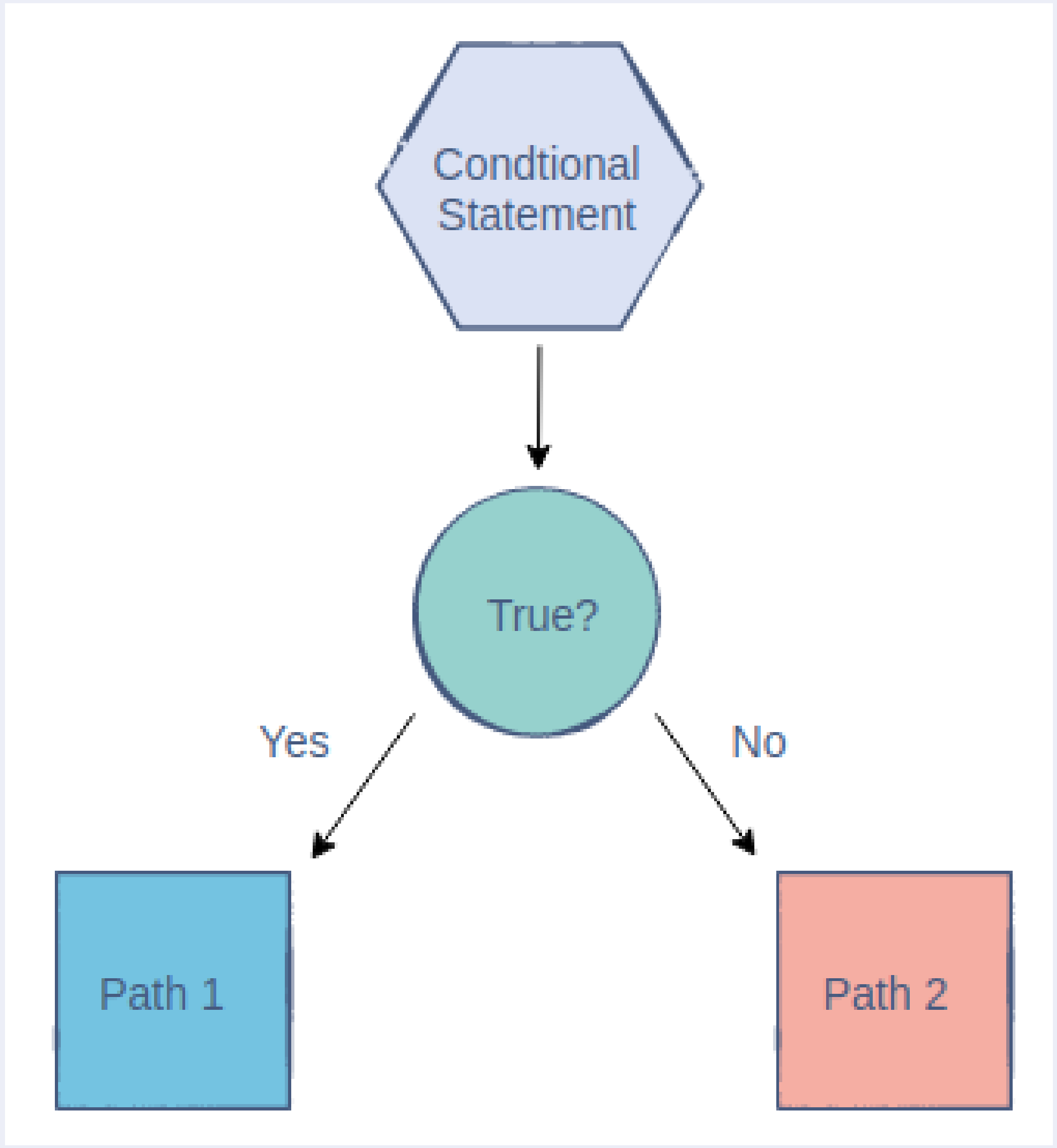
CONDITIONAL STATEMENTS

"A conditional statement is a Boolean expression that, if **True**, executes a piece of code".

CONDITIONAL STATEMENTS

It allows programs to branch out into different paths based on Boolean expressions result in **True** or **False** outcomes

conditional statements control the flow of the code and allow the computer to think



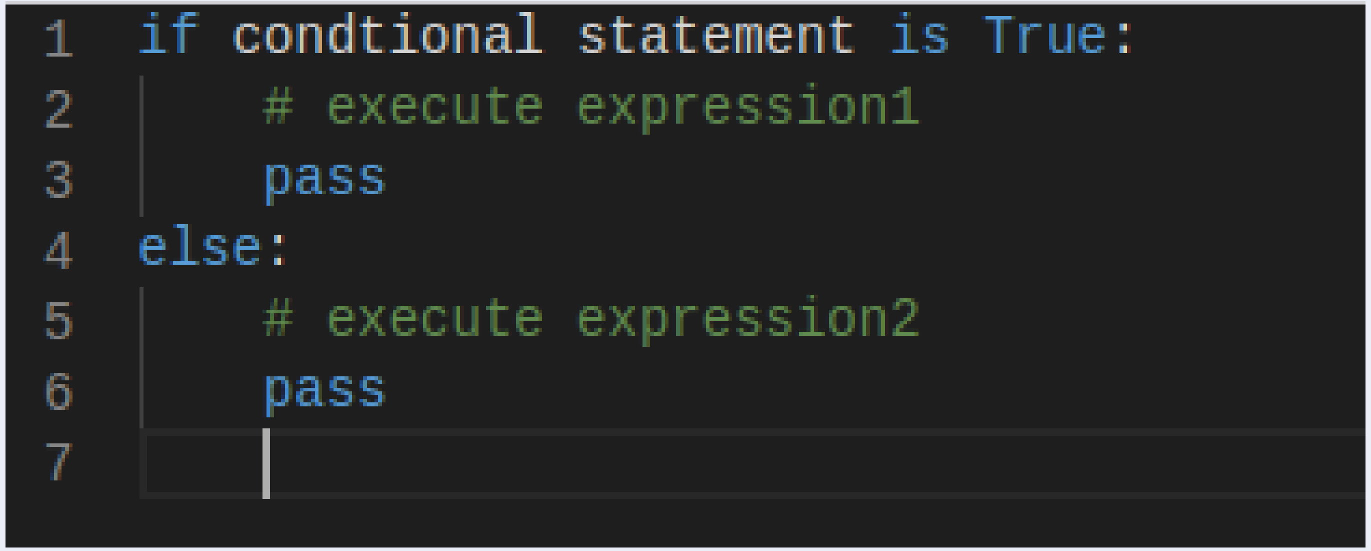
CONDITIONAL STATEMENTS IN PYTHON

There are three types of conditional statements in Python:

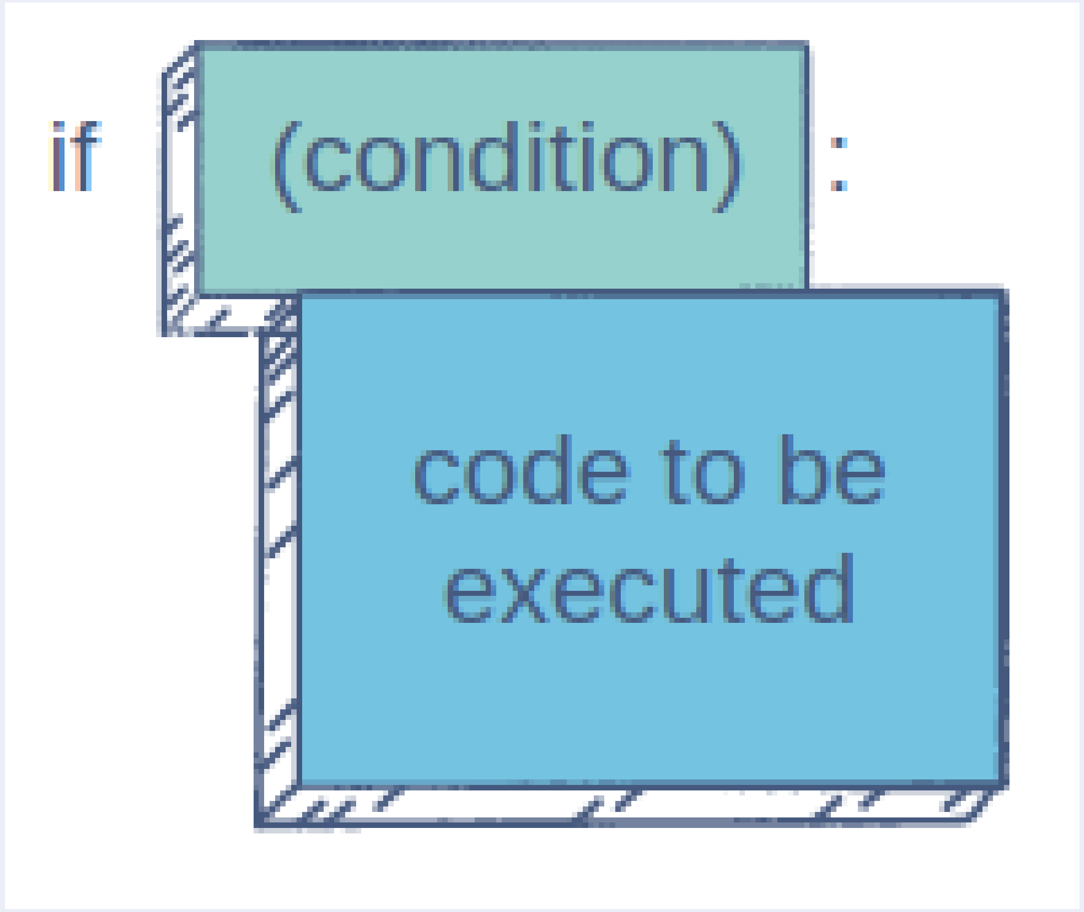
if

if-else

if-elif-else



```
if condtional statement is True:
    # execute expression1
    pass
else:
    # execute expression2
    pass

```



THE IF STATEMENT

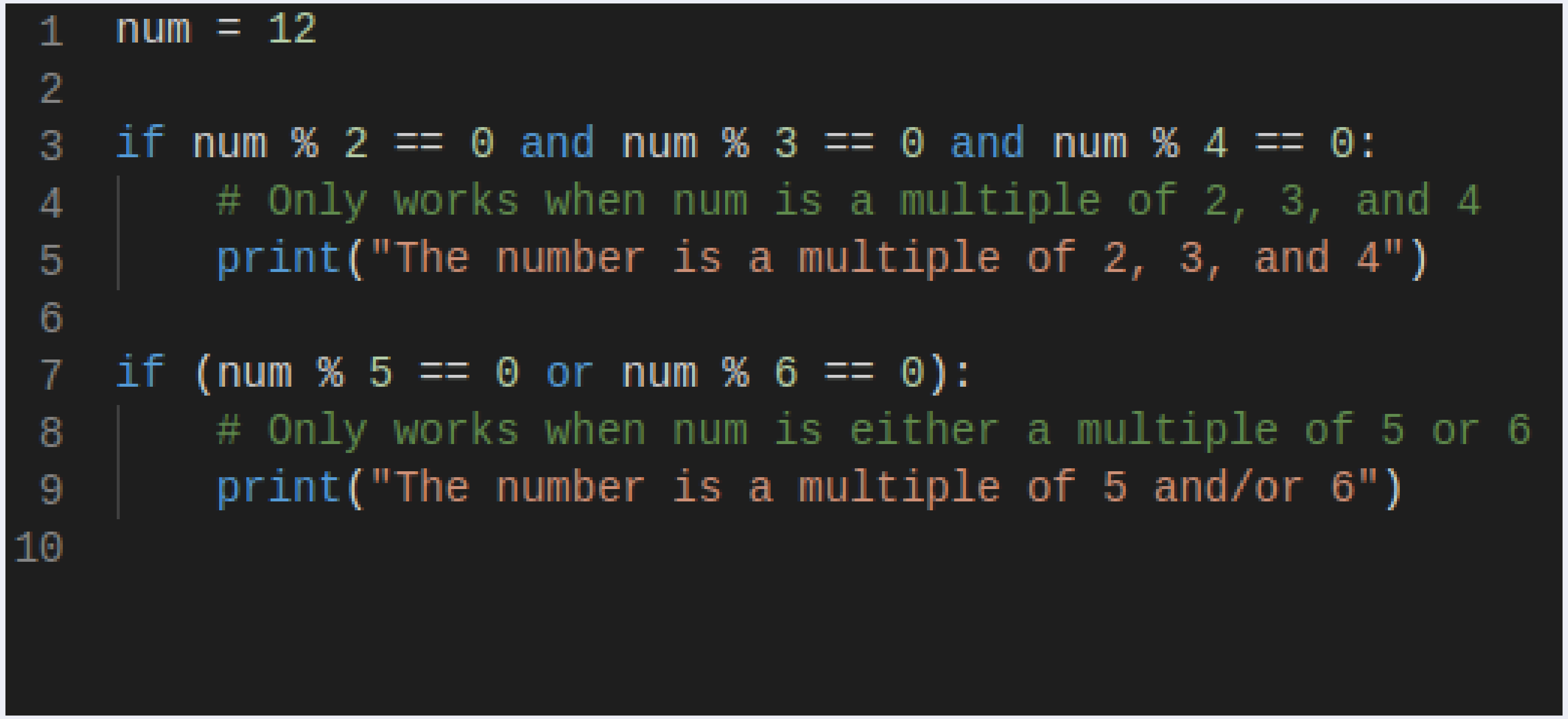
The simplest conditional statement that we can write is the if statement. It comprises of two parts:

The **condition**

The **code to be executed**

CONDITIONS WITH LOGICAL OPERATORS

We can use logical operators to create more complex conditions in the **if** statement.

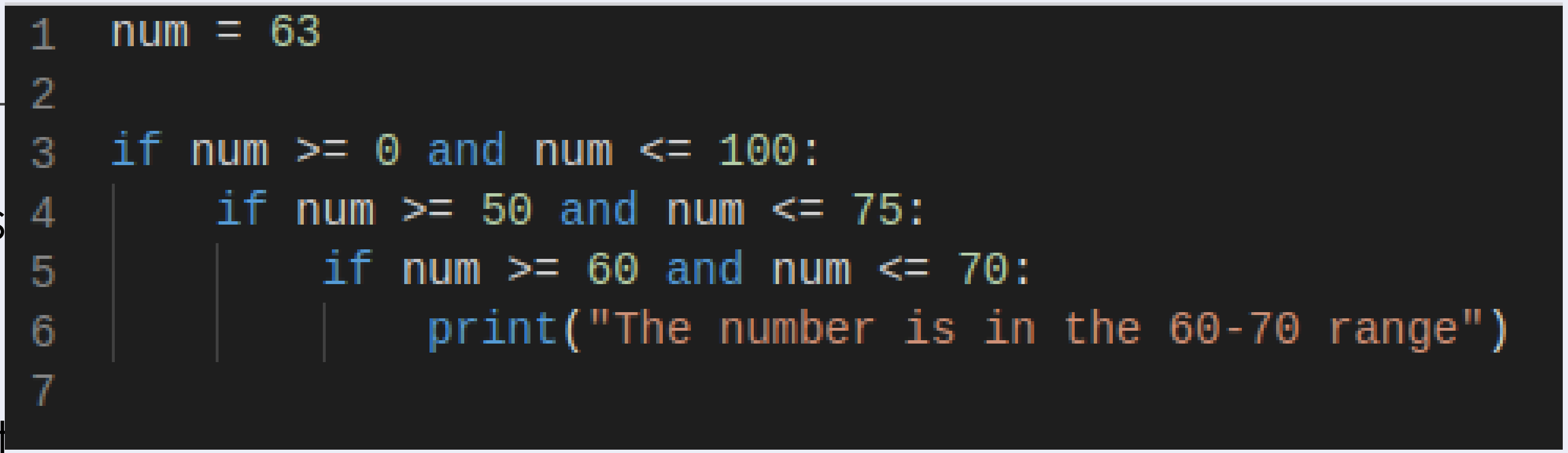


```python
num = 12

if num % 2 == 0 and num % 3 == 0 and num % 4 == 0:
    # Only works when num is a multiple of 2, 3, and 4
    print("The number is a multiple of 2, 3, and 4")

if (num % 5 == 0 or num % 6 == 0):
    # Only works when num is either a multiple of 5 or 6
    print("The number is a multiple of 5 and/or 6")

```

NESTED IF STAT EMENTS

A cool feature of conditional statements is that we can nest them.

This means that there could be an **if** statement inside another!

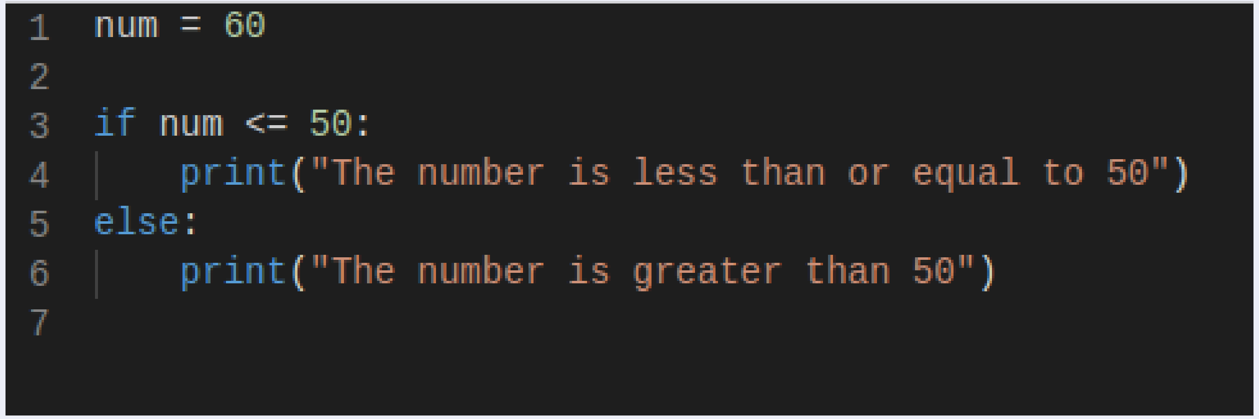


```
num = 63

if num >= 0 and num <= 100:
    if num >= 50 and num <= 75:
        if num >= 60 and num <= 70:
            print("The number is in the 60-70 range")

```

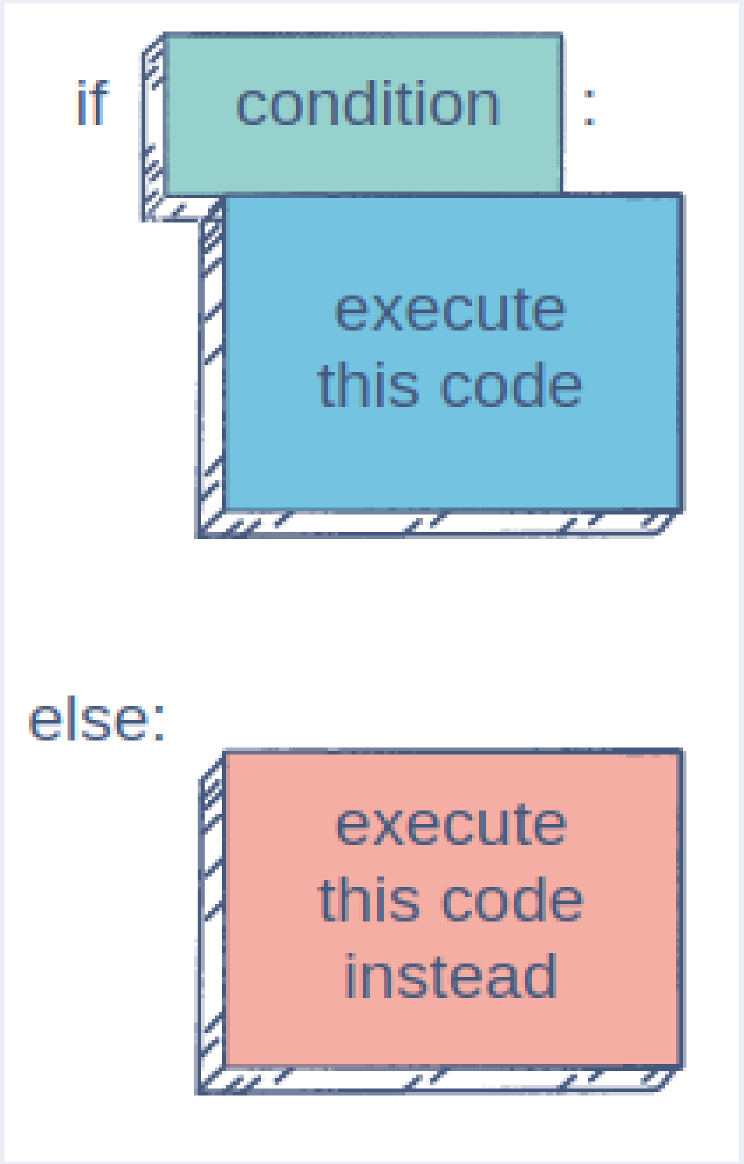
THE IF-ELSE STATEMENT

~~What if we wanted to execute a different set of~~ operations in case the **if** condition turns out to be **False**?

The **else** keyword will be on the same indentation level as the **if** keyword. Its body will be indented one tab to the right just like the **if** statement.



```
num = 60

if num <= 50:
    print("The number is less than or equal to 50")
else:
    print("The number is greater than 50")

```



CONDITIONAL EXPRESSIONS

Conditional expressions use the functionality of an **if-else** statement in a different way

A conditional expression can be written in the following way:

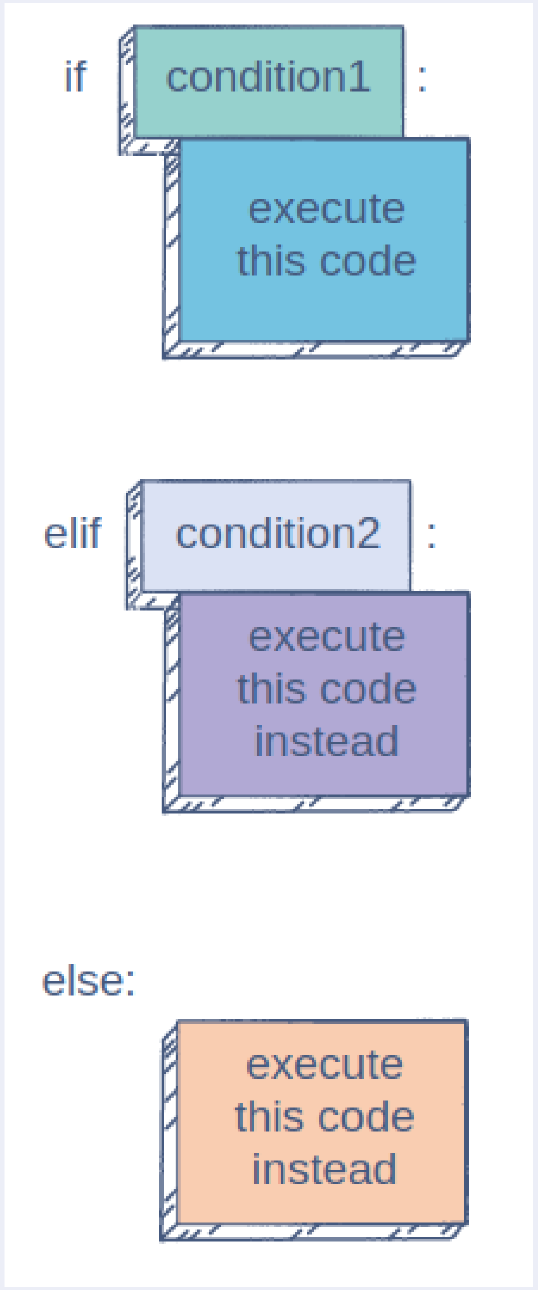
```
output_value1 if condition else output_value2
```

If the **if** condition is fulfilled, the output would be **output_value1**. Otherwise, it would be **output_value2.**

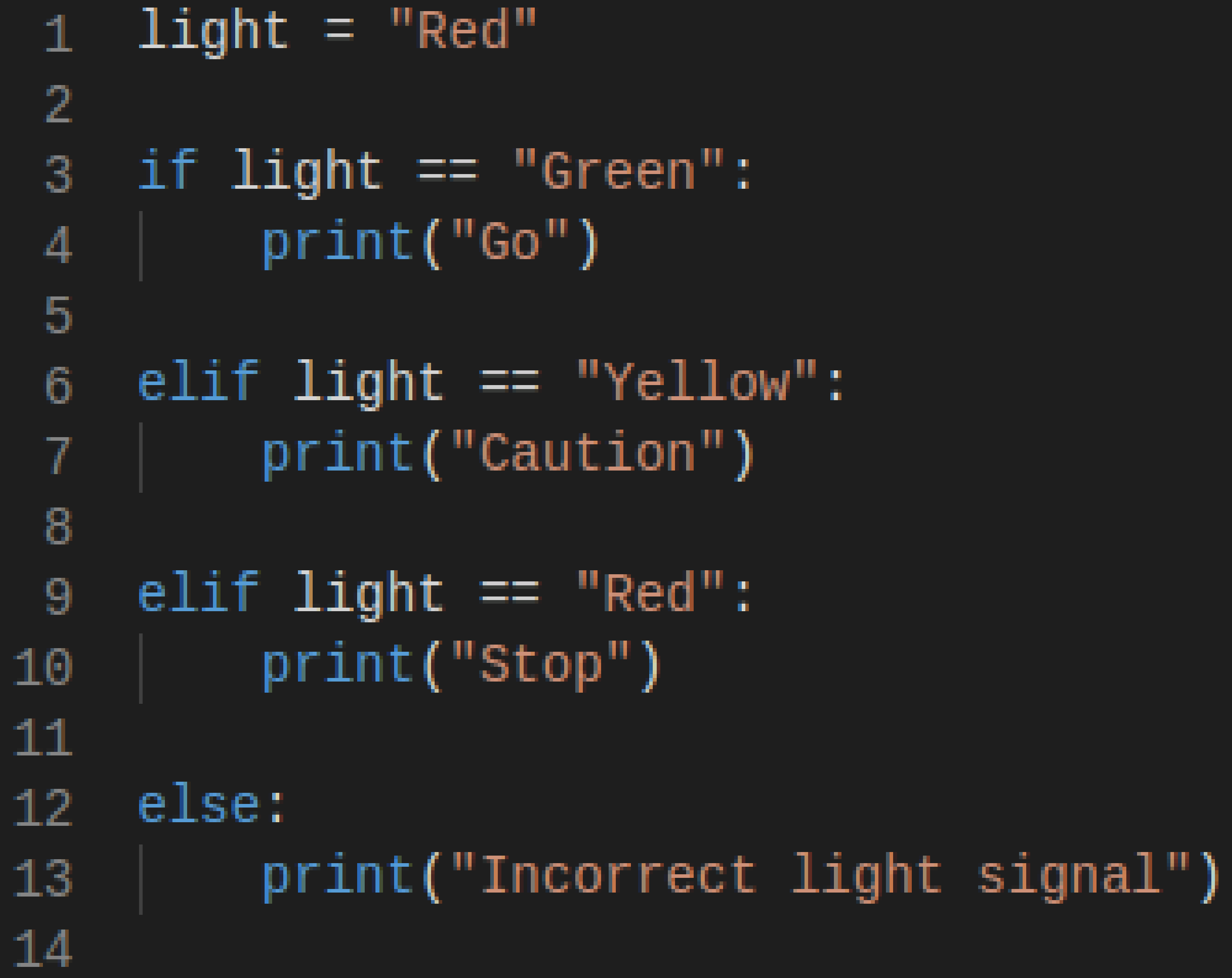
```
num = 60

output = "The number is less than or equal to 50" \
    if num <= 50 else "The number is greater than 50"

print(output)

```

THE IF-ELIF-ELSE STATEMENT

It is the most comprehensive conditional statement because it allows us to create multiple conditions easily.



IF-ELIF EXAMPLE

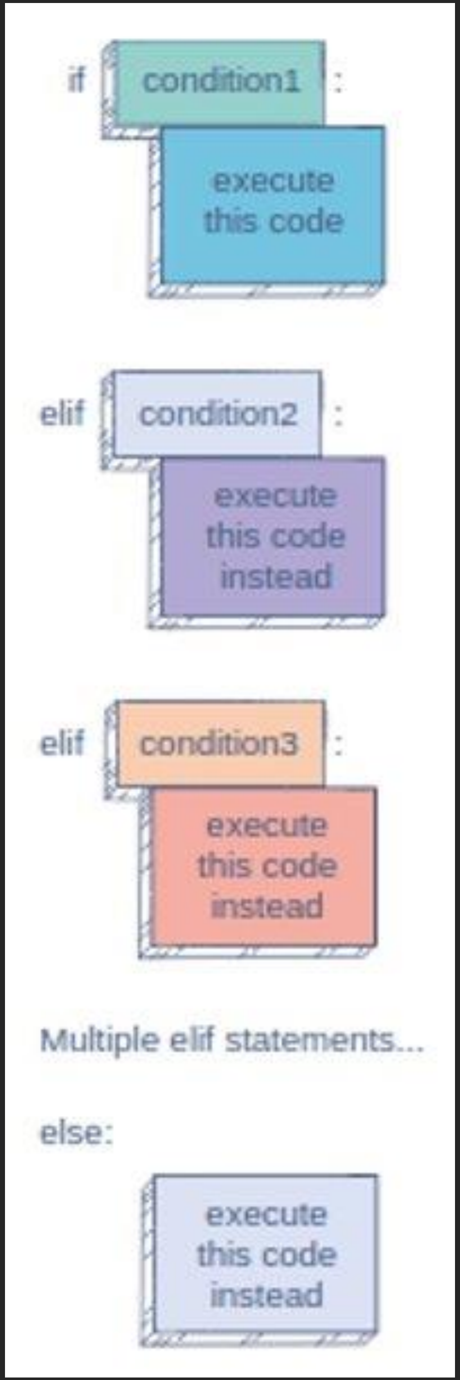


```python
light = "Red"

if light == "Green":
    print("Go")

elif light == "Yellow":
    print("Caution")

elif light == "Red":
    print("Stop")

else:
    print("Incorrect light signal")

```

MULTIPLE ELIF ST ATEMENTS

This is the beauty of the **if-elif-else** statement.

We can have as many **elifs** as we require, as long as they come between **if** and **else**


if
condition1
:
execute this code
elif
condition2
:
execute this code instead
elif
condition3
:
execute this code instead
Multiple elif statements...
else:
execute this code instead

EXAMPLE & EXERCISES

1. **Discounted Price**

In this Exercise, you must discount a price according to its value.

- If the price is 300 or above, there will be a 30% discount.
- If the price is between 200 and 300 (200 inclusive), there will be a 20% discount.
- If the price is between 100 and 200 (100 inclusive), there will be a 10% discount.
- If the price is less than 100, there will be a 5% discount.
- If the price is negative, there will be no discount

Sample Input: price = 250
Sample Output: price = 200

OUTLINES

What are loops

For Loops

Nested For Loops

While Loops

Exercises

LIST

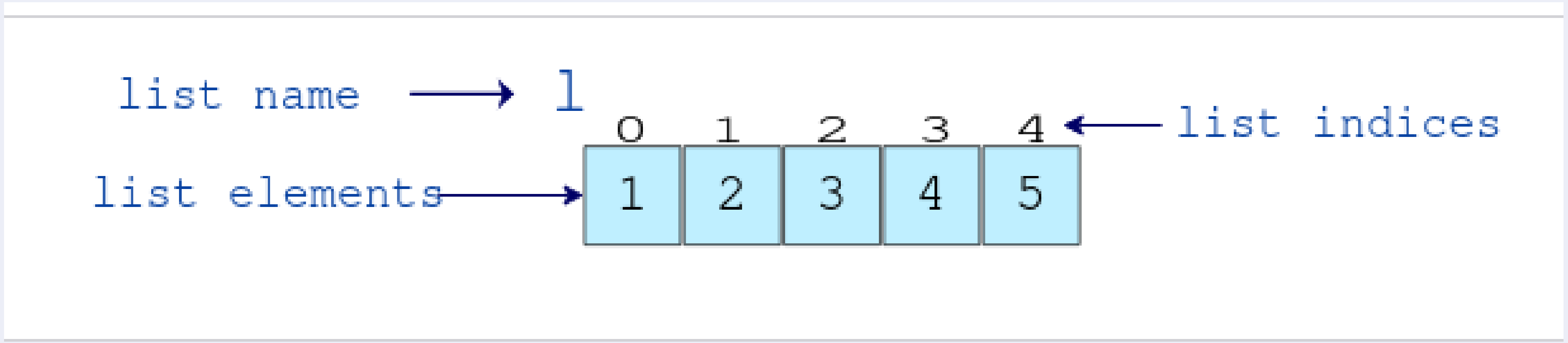
WHAT ARE LISTS ?

Python lists are data structures that group sequences of elements.

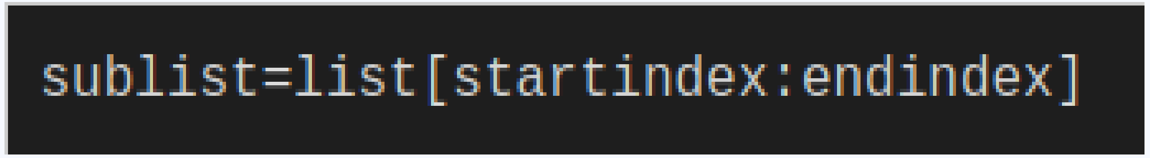
Lists **can have elements of several types**, and you can also **mix different types** within the same list (although all elements are usually of the same datatype).

Lists are created using square brackets [], and the elements are separated by commas (,).

The elements in a list can be accessed by their positions, starting with 0 as the index of the first element.



SUBLIST



```
sublist=list[startindex:endindex]
```

Sometimes you want just a small portion of a list—a **sublist.**

Sublists are simply subsets of a list; they can be retrieved using a technique called *slicing*.

Note: The start index is inclusive and end index is exclusive in the range.

OPERATIONS ON LIST

List Concatenation

- Lists can be concatenated using the **+** operator

Traverse a List

- Lists can be iterated over using for loops in Python

List Methods

Python List append()

- Add a single element to the end of the list [**list.append(item)**]

Python List clear()

- Removes all Items from the List [**list.clear()**]

LIST METHODS

Python List copy()

- returns a shallow copy of the list

Python List count()

- returns count of the element in the list [**list.count(element)**]

Python List extend()

- adds iterable elements to the end of the list [**list1.extend(iterable)**]

Python List index()

- returns the index of the element in the list [**list.index(element, start, end)**]

Python List insert()

- insert an element to the list [**list.insert(i, elem)**]

LIST METHODS

Python List pop()

- Removes element at the given index [**list.pop(index)**]

Python List remove()

- Removes item from the list **[list.remove(element)]**

Python List reverse()

- reverses the list [**list.reverse()**]

Python List sort()

- sorts elements of a list [**list.sort(key=..., reverse=...)**]

EXERCISES

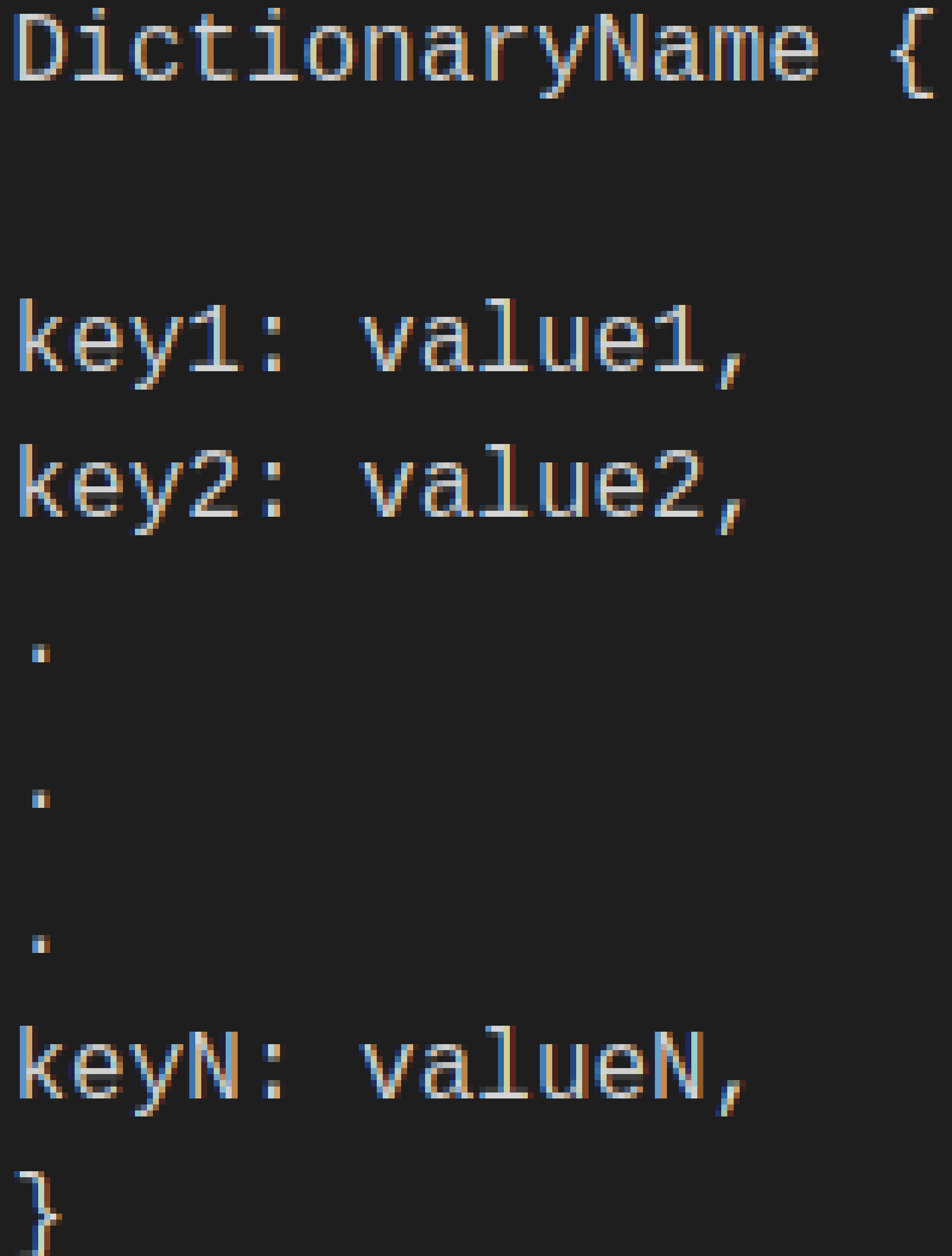
DICTIONARIES

WHAT ARE DICTIONARIES

Dictionaries are data structures that index values by a given key **(key-value pairs)**.

Dictionaries are written with curly brackets **{}**, and they have keys and values.

The general syntax for creating a dictionary is:

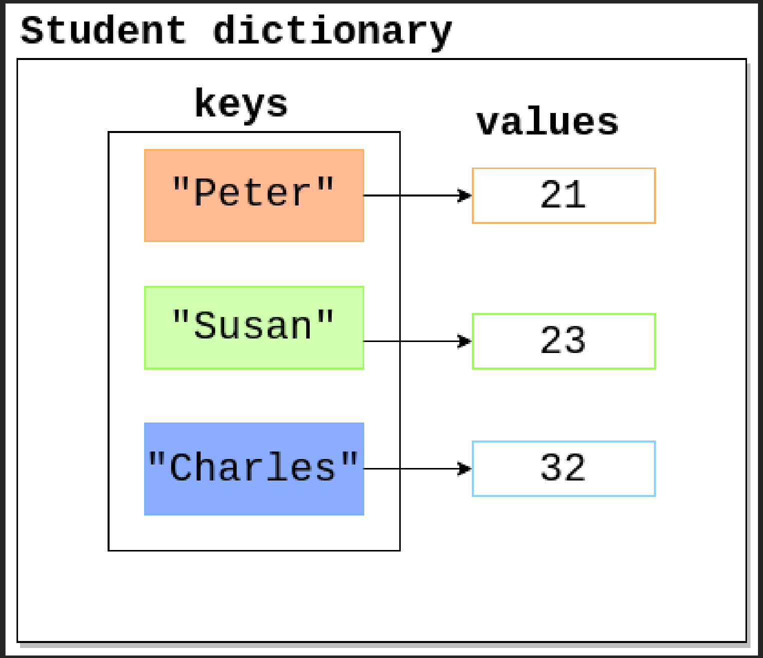


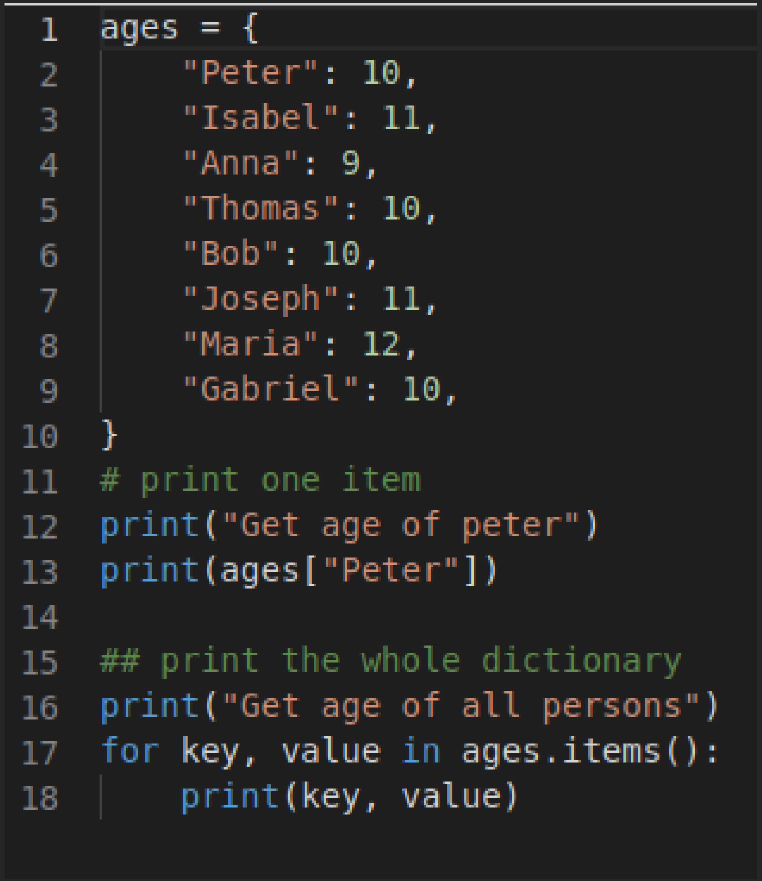
```
DictionaryName {

key1: value1,
key2: value2,
.
.
.
keyN: valueN,
}
```

CREATING A DICTIONARY

Creating an empty dictionary:

- new_dict = dict()
- new_dict = {}

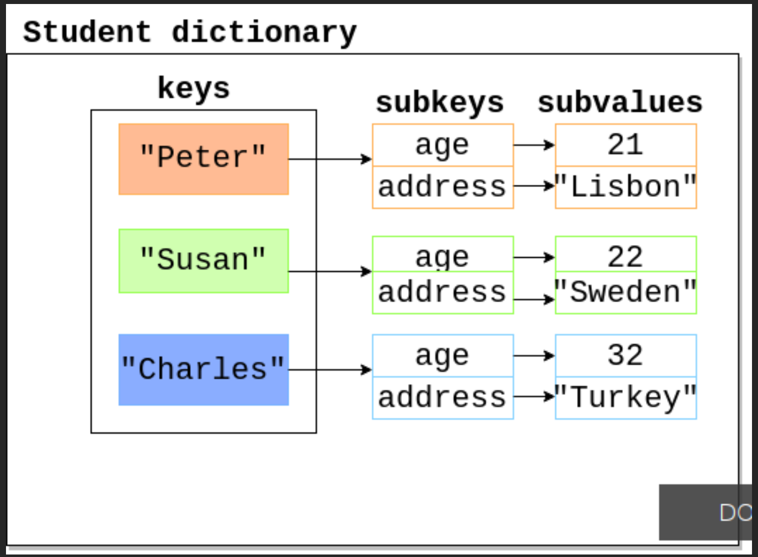


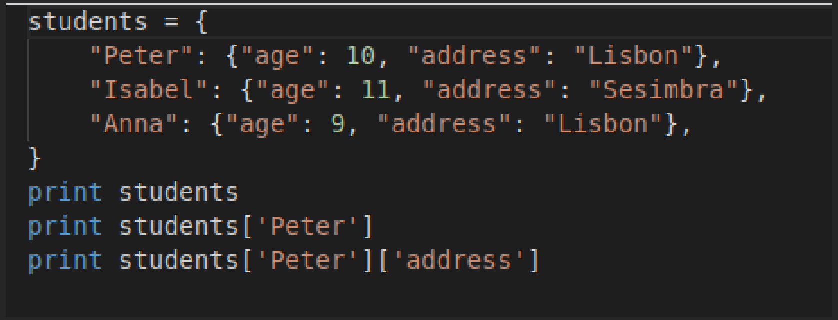


```python
ages = {
    "Peter": 10,
    "Isabel": 11,
    "Anna": 9,
    "Thomas": 10,
    "Bob": 10,
    "Joseph": 11,
    "Maria": 12,
    "Gabriel": 10,
}
# print one item
print("Get age of peter")
print(ages["Peter"])

## print the whole dictionary
print("Get age of all persons")
for key, value in ages.items():
    print(key, value)
```

NESTED DICTIONARY

A dictionary can be made within a dictionary, and you can also use other dictionaries as values.





```
students = {
    "Peter": {"age": 10, "address": "Lisbon"},
    "Isabel": {"age": 11, "address": "Sesimbra"},
    "Anna": {"age": 9, "address": "Lisbon"},
}
print students
print students['Peter']
print students['Peter']['address']
```

WHAT ARE LOOPS?

"A **loop** is a control structure that is used to perform a set of instructions for a specific number of times."

DEFINITION

Loops solve the problem of having to write the same set of instructions repeatedly.

Just like conditional statements, a loop is classified as a control structure because it directs the flow of a program by making varying decisions in its **iterations**

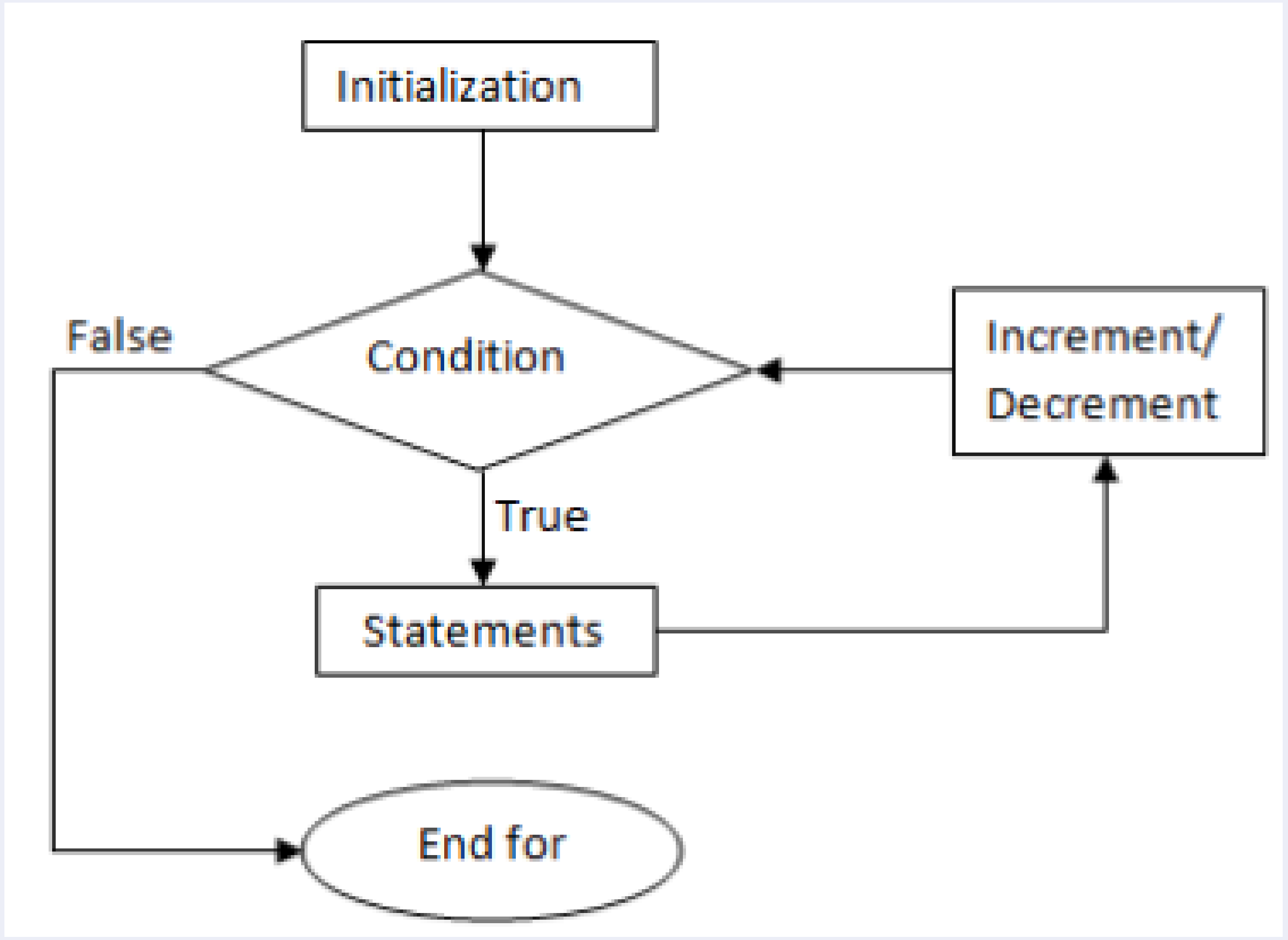
Loops are a crucial part of many popular programming languages such as C++, Java, and JavaScript.

LOOPS IN PYTHON

There are two types of loops that we can use in Python:

The for loop

The while loop



THE FOR LOOP

A for loop uses an **iterator** to traverse a sequence

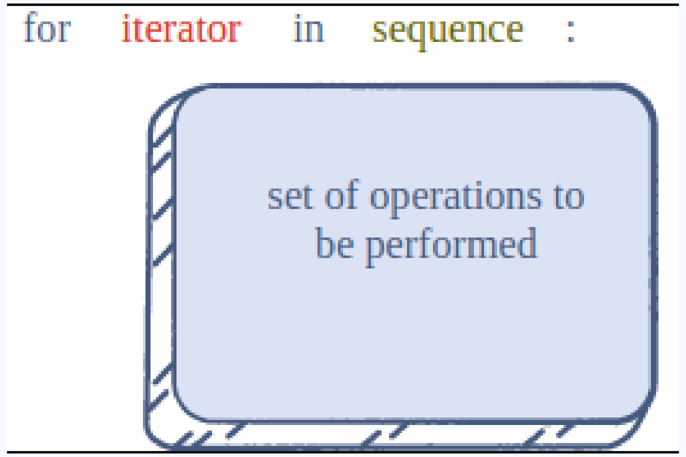
The iterator starts from the beginning of the sequence.

In each iteration, the iterator updates to the next value in the sequence.

The loop ends when the iterator reaches the end.

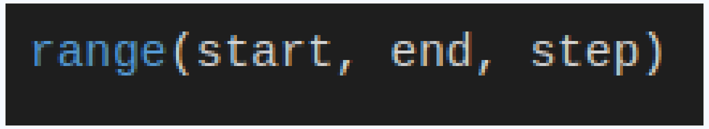
Structure:

- The name of the iterator
- The sequence to be traversed
- The set of operations to perform



LOOPING THROUGH A RANGE

In Python, the built-in **range()** function can be used to create a sequence of integers



```
range(start, end, step)
```

The **end** value is not included in the list.

If the **start** index is not specified, its default value is **0**

The optional **step** decides the number of steps the iterator jumps ahead after each iteration.

if we don’t specify it, the default step is 1

EXAMPLES

Sequence of numbers from 1 to 10

A sequence of even numbers between 0 and 20 (inclusive)

A sequence of numbers from 1 to 10 with a step of 3

Print individual letters of a string "PYTHON"

Given the list numbers = [2, 7,10, 16, 23, 54] print a list of square of each number in the list

Given the float_list = [2.5, 16.42, 10.77, 8.3, 34.21] print how many numbers are less than 10.

NESTED FOR LOOPS

"Python lets us easily create loops within loops"

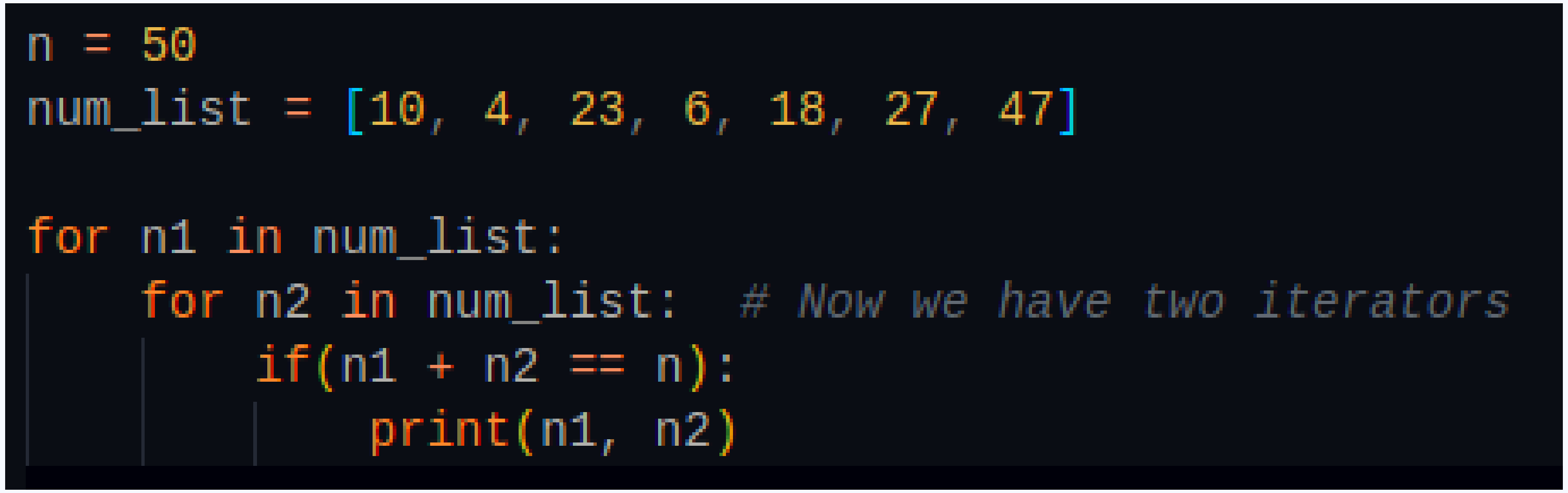
EXECUTION OF NESTED LOOPS

The inner loop will always complete before the outer loop.

Example: Suppose we want to print two elements whose sum is equal to a certain number **n**

Any Suggestion of the algorithm to use ??

SOLUTION



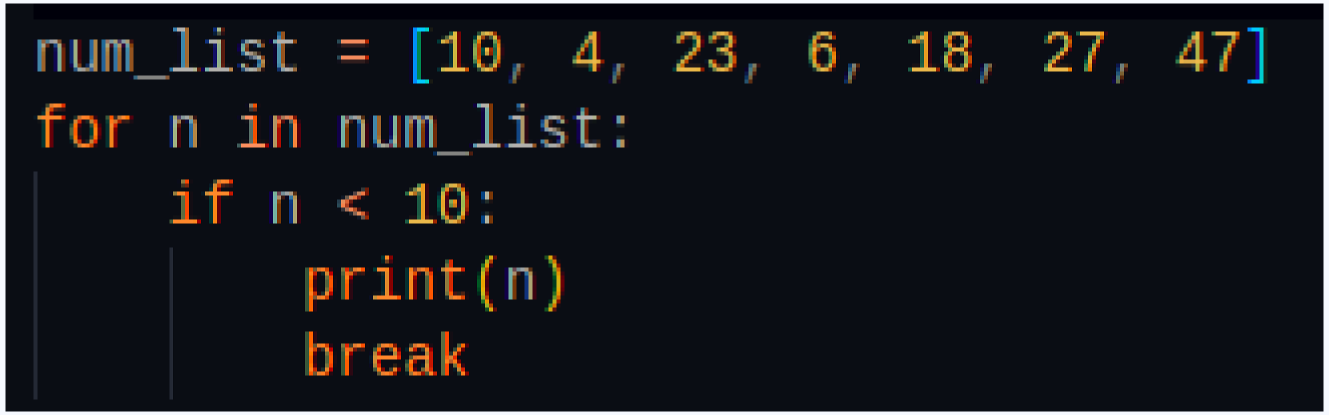
```python
n = 50
num_list = [10, 4, 23, 6, 18, 27, 47]

for n1 in num_list:
    for n2 in num_list:  # Now we have two iterators
        if(n1 + n2 == n):
            print(n1, n2)
```

THE BREAK KEYWORD

Sometimes, we need to exit the loop before it reaches the end.

This can happen if we have found what we were looking for and don't need to make any more computations in the loop.

That's what the **break** keyword is for. It can *break* the loop whenever we want.

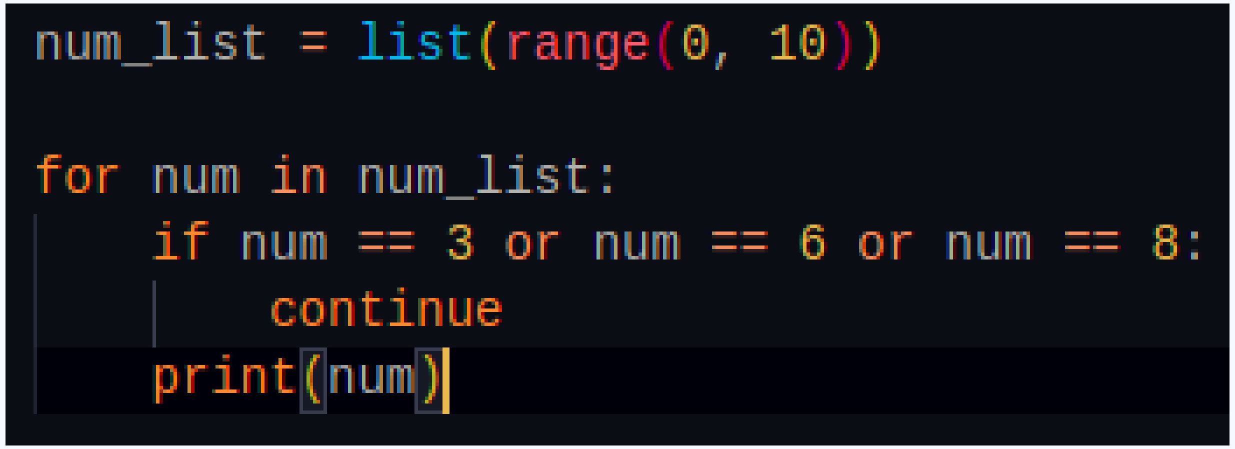


```
num_list = [10, 4, 23, 6, 18, 27, 47]
for n in num_list:
    if n < 10:
        print(n)
        break
```

THE CONTINUE KEYWORD

When the **continue** keyword is used, the rest of that particular iteration is skipped.

The loop *continues* on to the next iteration.



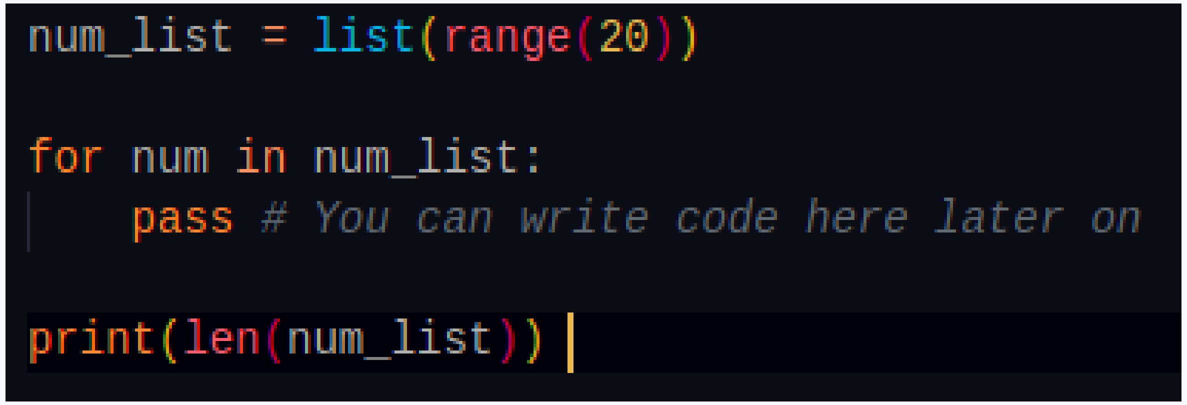
```python
num_list = list(range(0, 10))

for num in num_list:
    if num == 3 or num == 6 or num == 8:
        continue
    print(num)
```

THE PASS KEYWORD

In all practical meaning, the **pass** statement does nothing to the code execution.

It can be used to represent an area of code that needs to be written.

it is used to assist you when you haven't written a piece of code but still need your entire program to execute



```python
num_list = list(range(20))

for num in num_list:
    pass # You can write code here later on

print(len(num_list))
```

THE WHILE LOOP

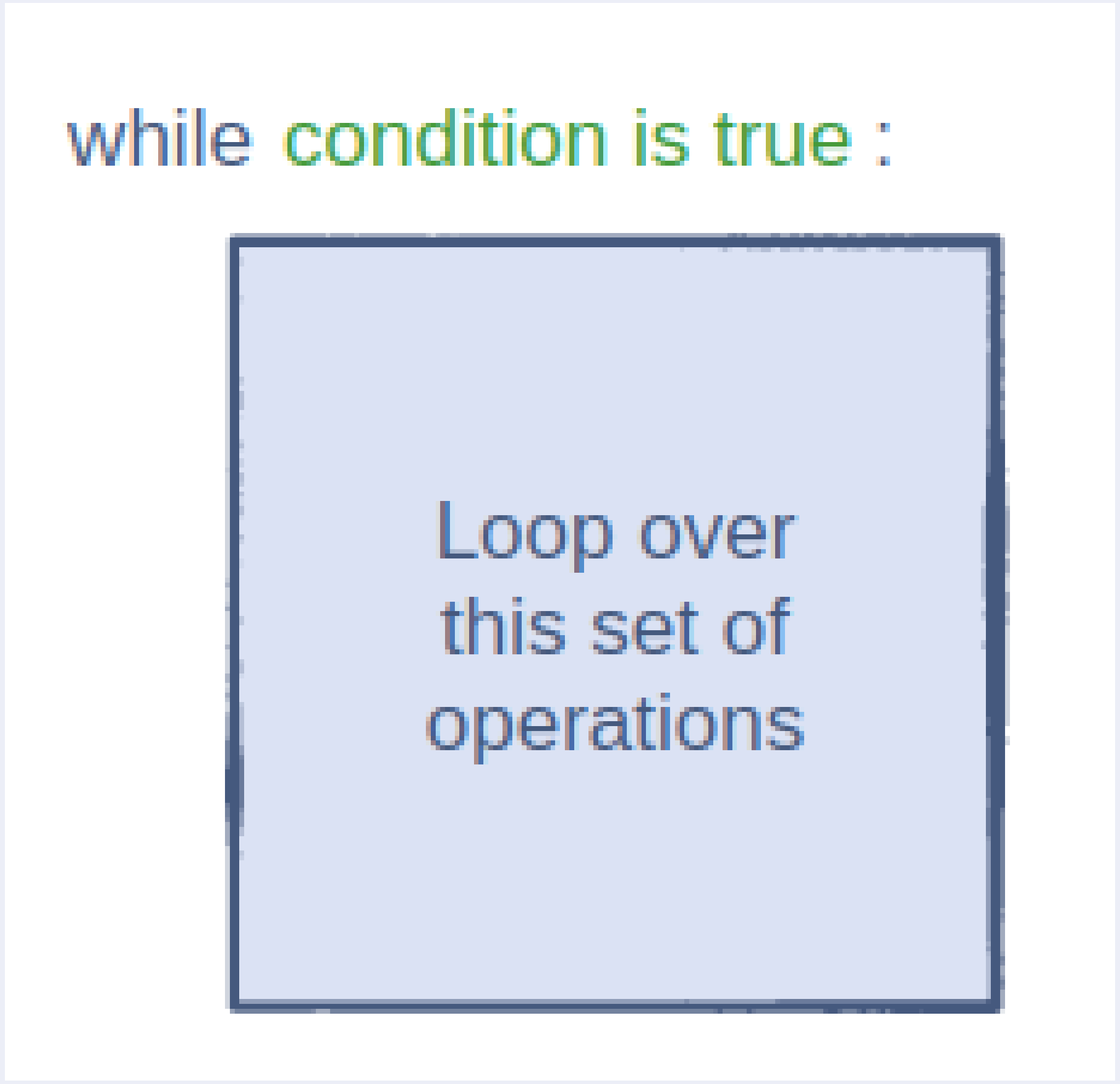
"The while loop keeps iterating over a certain set of operations as long as a certain **condition** holds **True**

While this condition is true, keep the loop running"

WHILE LOOP STRUCTURE

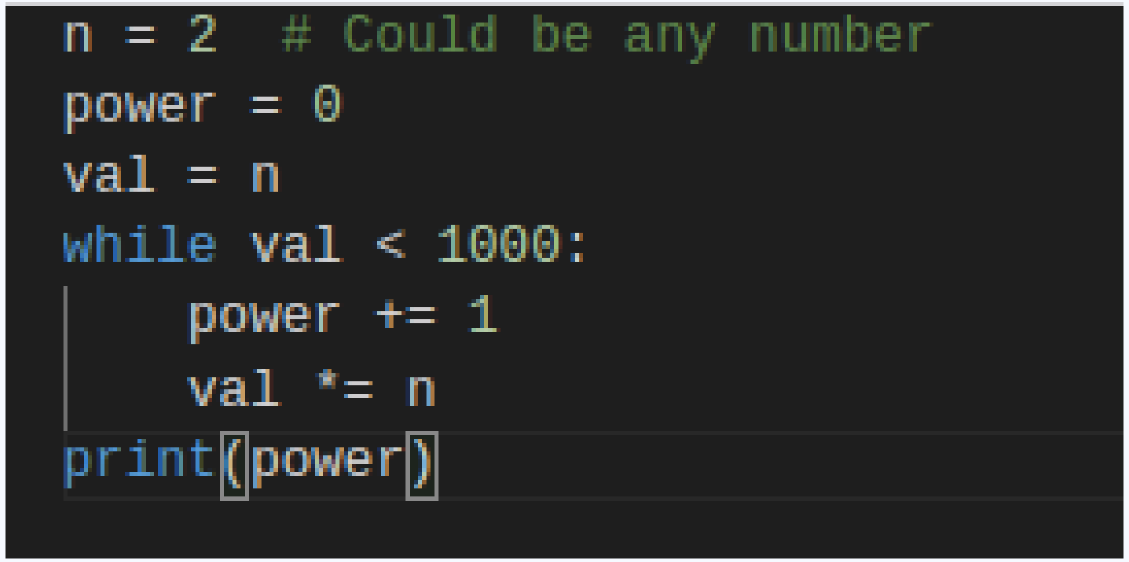
while loop is not always restricted to a fixed range

Its execution is based solely on the condition associated with it.



THE WHILE LOOP IN ACTION

Here's a while loop that finds out the maximum power of n before the value exceeds 1000

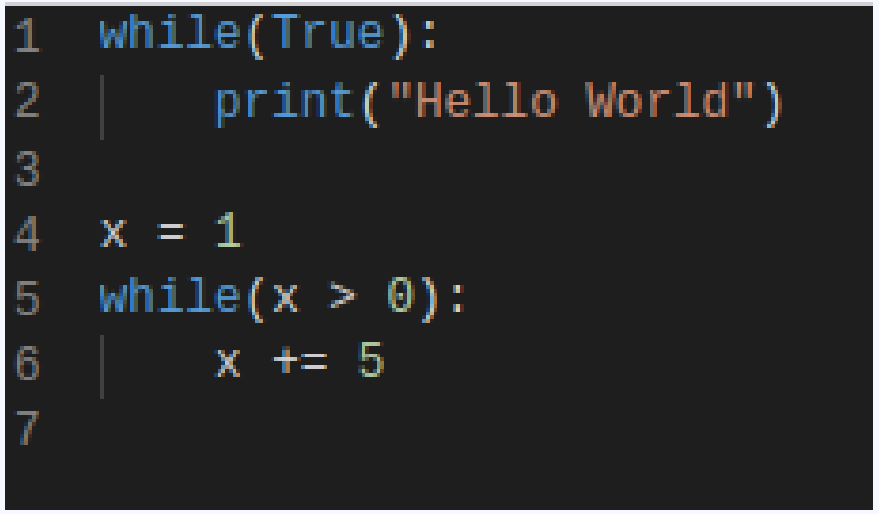


```
n = 2  # Could be any number
power = 0
val = n
while val < 1000:
    power += 1
    val *= n
print(power)
```

CAUTIONARY MEASURES

Compared to for loops, we should be more careful when creating while loops.

This is because a while loop has the potential to never end. This could crash a program!



```
while(True):
    print("Hello World")

x = 1
while(x > 0):
    x += 5

```

EXERCISES

1. Create a loop that counts from 0 to 100
2. Make a multiplication table using a loop
3. Output the numbers 1 to 10 backwards using a loop
4. Create a loop that counts all even numbers to 10
5. Create a loop that sums the numbers from 100 to 200

FUNCTIONS IN PYTHON

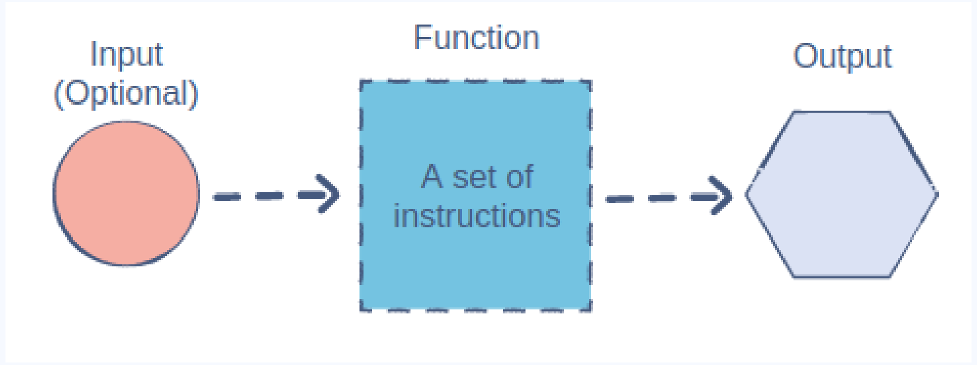
"A function is a reusable set of operations"

WHY FUNCTIONS?

Functions are useful because they make the code concise and simple.

The primary benefits of using functions are:

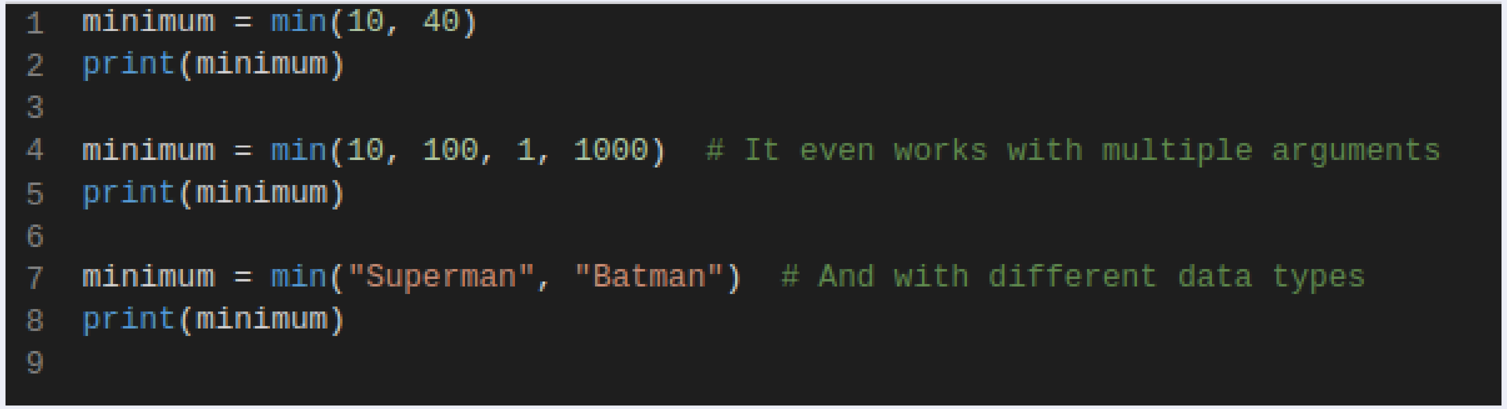
- **Reusability**: A function can be used over and over again. You do not have to write redundant code.
- **Simplicity**: Functions are easy to use and make the code readable. We only need to know the inputs and the purpose of the function without focusing on the inner workings.



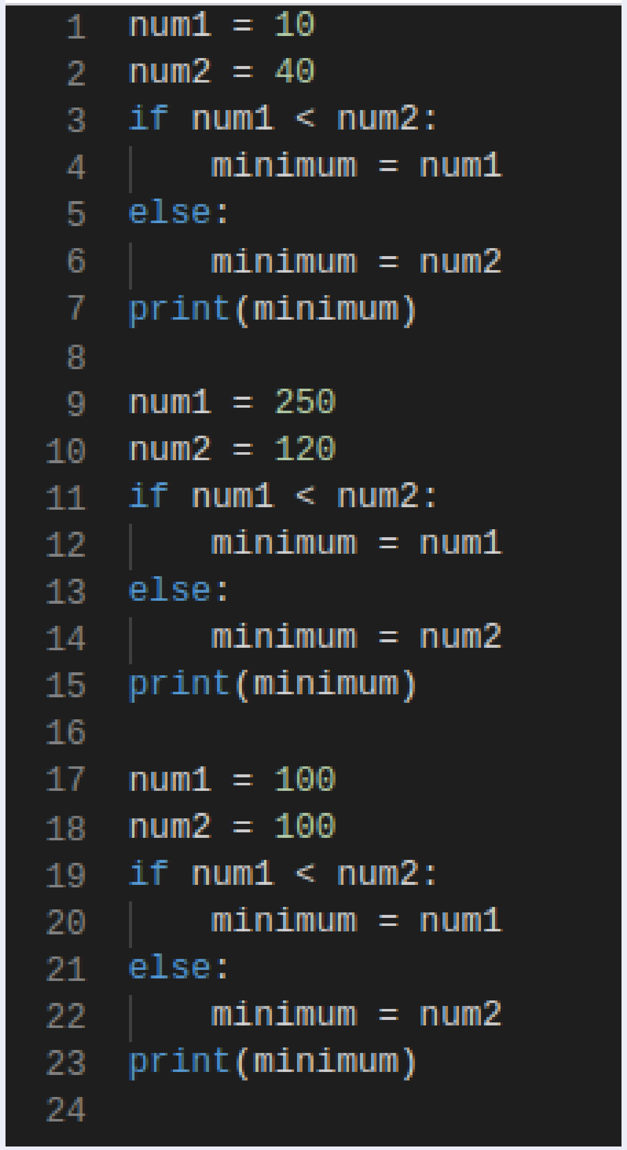
EXAMPLE

Suppose we want to find the smaller value between two integers.

The good news is that Python already has the min() function:



```python
minimum = min(10, 40)
print(minimum)

minimum = min(10, 100, 1, 1000)  # It even works with multiple arguments
print(minimum)

minimum = min("Superman", "Batman")  # And with different data types
print(minimum)

```



```python
num1 = 10
num2 = 40
if num1 < num2:
    minimum = num1
else:
    minimum = num2
print(minimum)

num1 = 250
num2 = 120
if num1 < num2:
    minimum = num1
else:
    minimum = num2
print(minimum)

num1 = 100
num2 = 100
if num1 < num2:
    minimum = num1
else:
    minimum = num2
print(minimum)

```

TYPES OF FUNCTIONS IN PYTHON

There are two basic types of functions in Python:

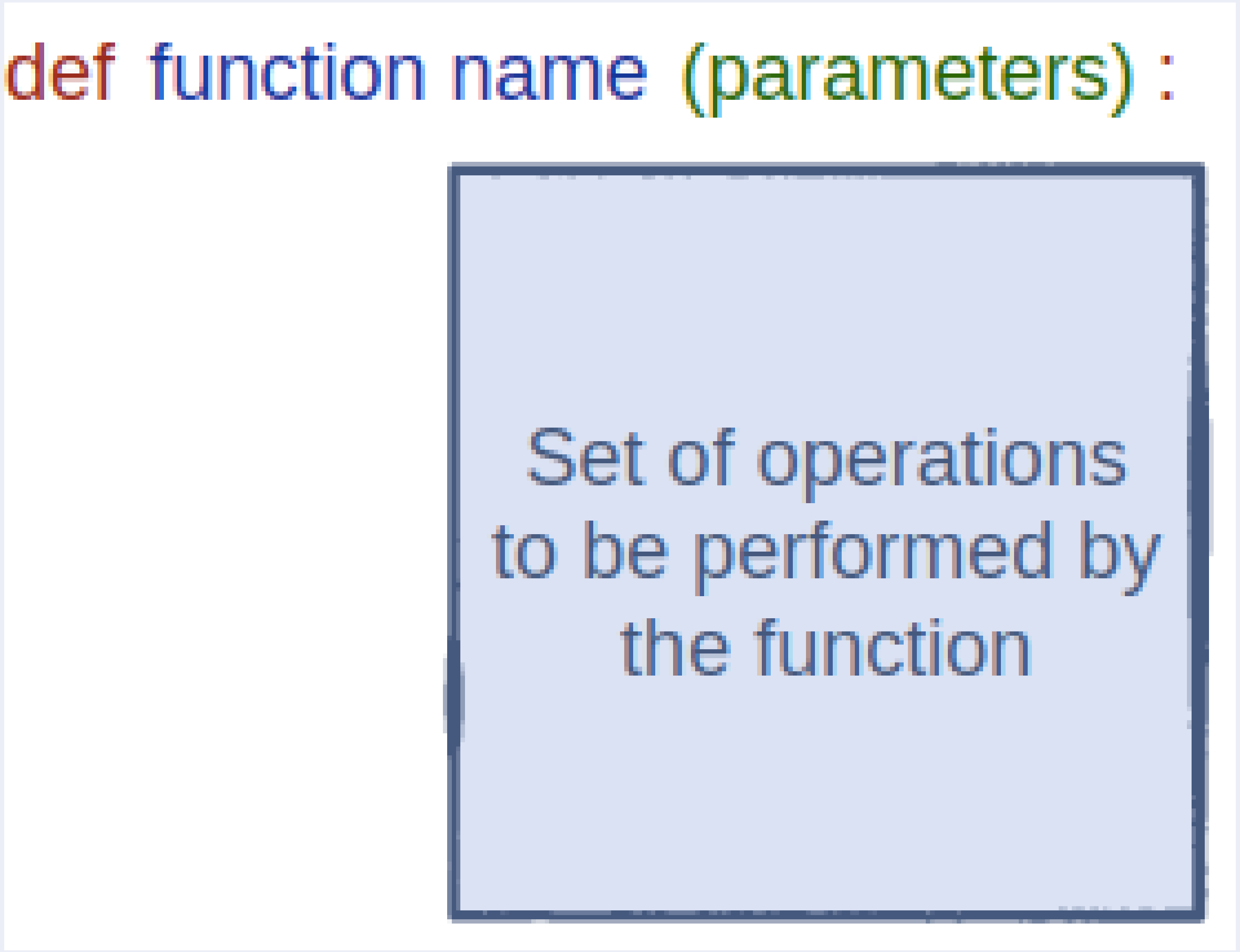
- Built-in functions: len(), min(), print(),….
- User-defined functions

COMPONENTS OF A FUNCTION

In Python, a function can be *defined* using the def keyword

The function name is simply the name we'll use to identify the function.

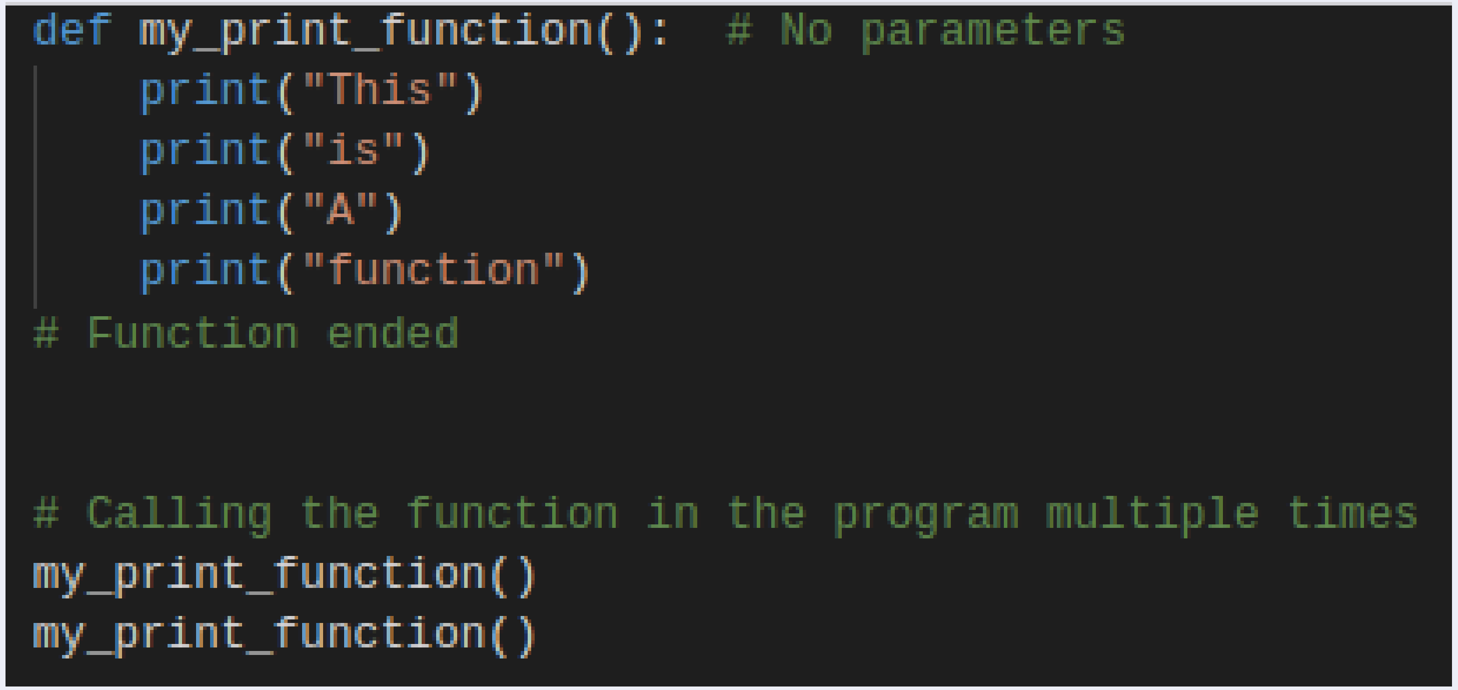
The parameters of a function are the inputs for that function. We can use these inputs within the function.


def function name (parameters) :
Set of operations to be performed by the function

IMPLEMENTATION

Let’s start by making a plain function that prints four lines of text.

We can call the function in our code using its name along with empty parentheses.



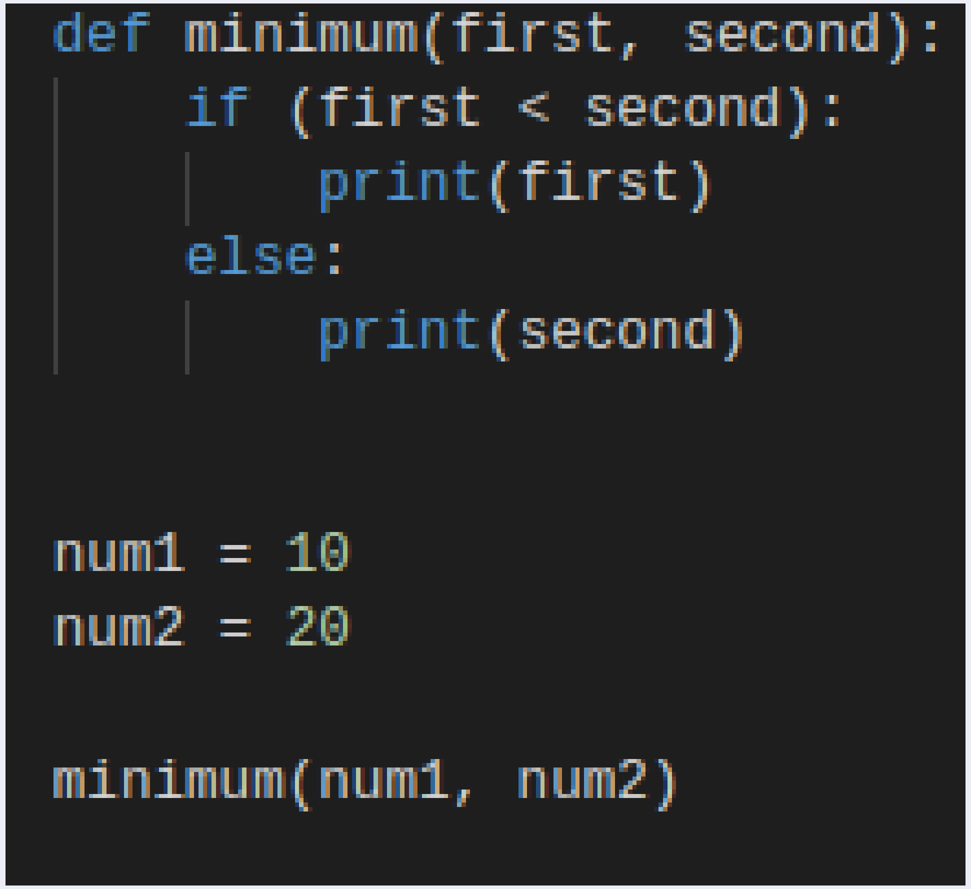
```python
def my_print_function():  # No parameters
    print("This")
    print("is")
    print("A")
    print("function")
# Function ended


# Calling the function in the program multiple times
my_print_function()
my_print_function()
```

FUNCTION PARAMETERS

Parameters are a crucial part of the function structure

They are the means of passing data to the function. This data can be used by the function to perform a meaningful task

The actual values/variables passed into the parameters are known as **arguments**



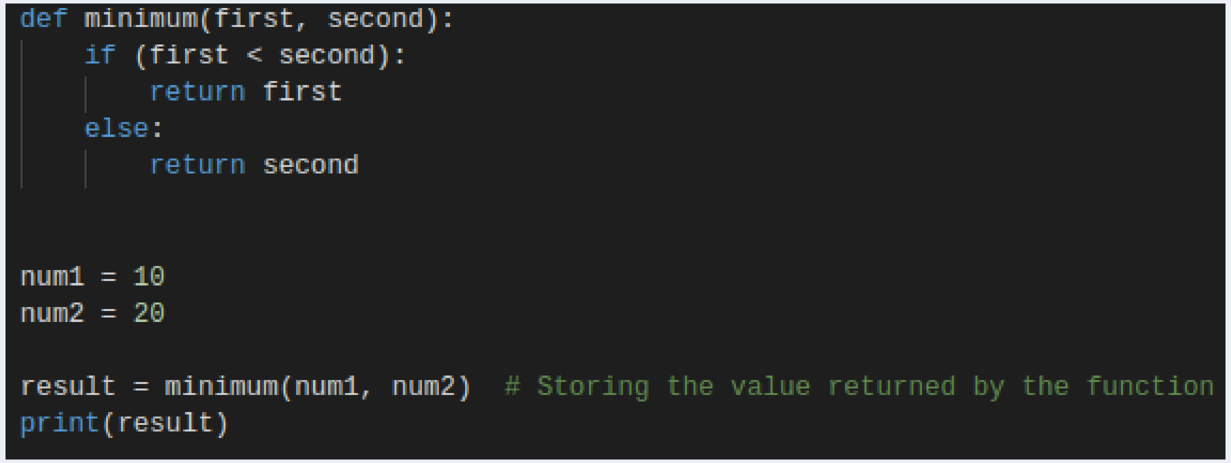
```python
def minimum(first, second):
    if (first < second):
        print(first)
    else:
        print(second)


num1 = 10
num2 = 20

minimum(num1, num2)
```

THE RETURN STATEMENT

To return something from a function, we must use the return keyword

Keep in mind that once the return statement is executed, the compiler ends the function

Any remaining lines of code after the return statement will not be executed.



```python
def minimum(first, second):
    if (first < second):
        return first
    else:
        return second


num1 = 10
num2 = 20

result = minimum(num1, num2)  # Storing the value returned by the function
print(result)
```

EXAMPLES & EXERCISES

1. Write a Python function that accepts a string and calculate the number of upper-case letters and lower-case letters

Sample string: PythOn

Number of Uppercase letters: 2

Number of Lowercase letters: 4

2. Write a function that returns the sum of multiples of 3 and 5 between 0 and **limit** (parameter). For example, if limit is 20, it should return the sum of 3, 5, 6, 9, 10, 12, 15, 18, 20.

3. Write a Python function to sum all the numbers in a list.
Sample List : (8, 2, 3, 0, 7)
Expected Output : 20

OOP IN PYTHON

OBJECT-ORIENTED PROGRAMMING, IT IS A PROGRAMMING PARADIGM THAT INCLUDES, OR RELIES, ON THE CONCEPT OF CLASSES AND OBJECTS.

OUTLINE

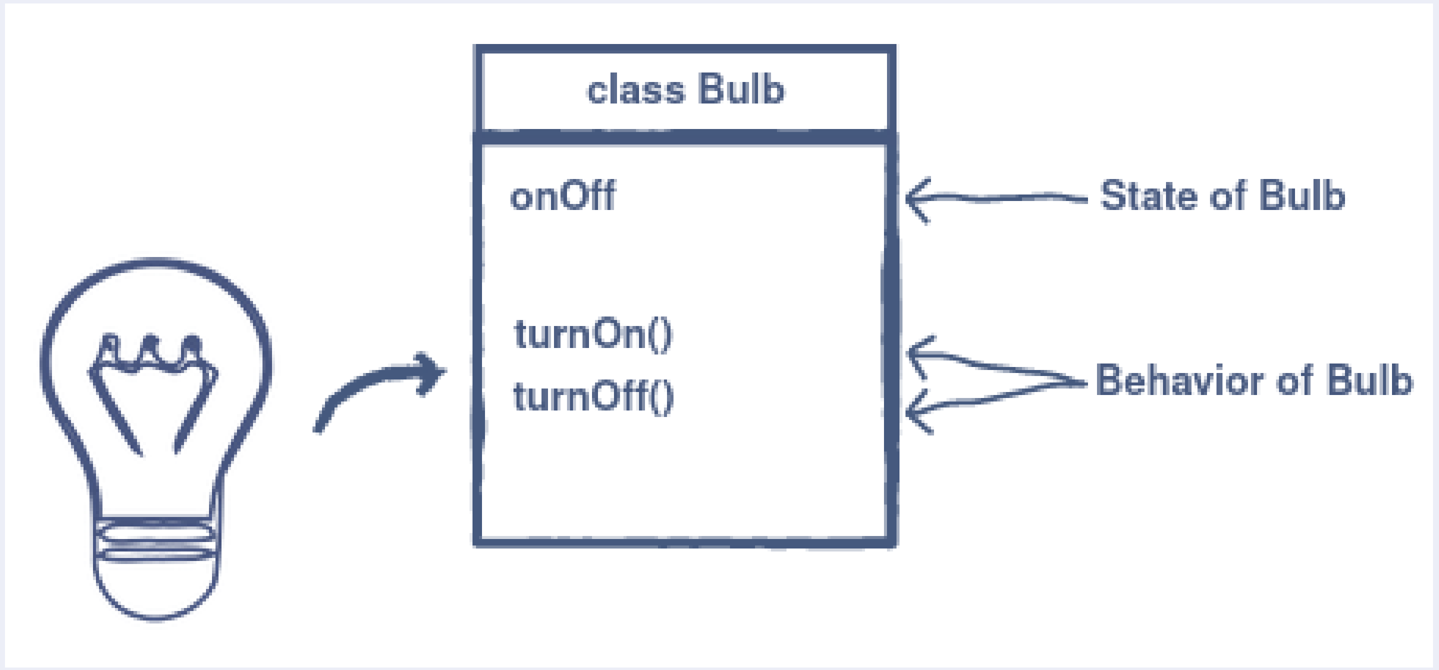
In this Unit, you will learn:

ANATOMY OF OBJECTS AND CLASSES

The basic idea of OOP is to divide a sophisticated program into a number of **objects** that talk to each other.

Objects are a collection of **data** and their **behaviors**.

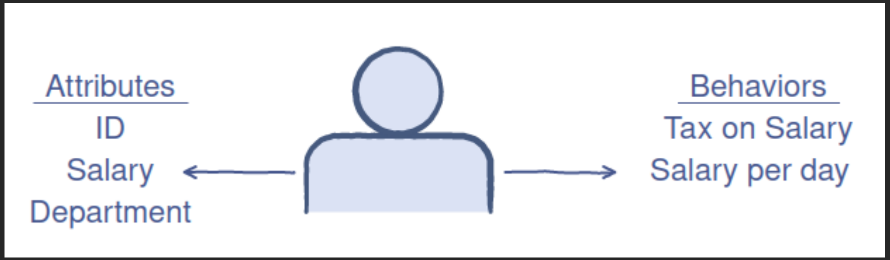
A **class** can be thought of as a *blueprint* for creating objects.

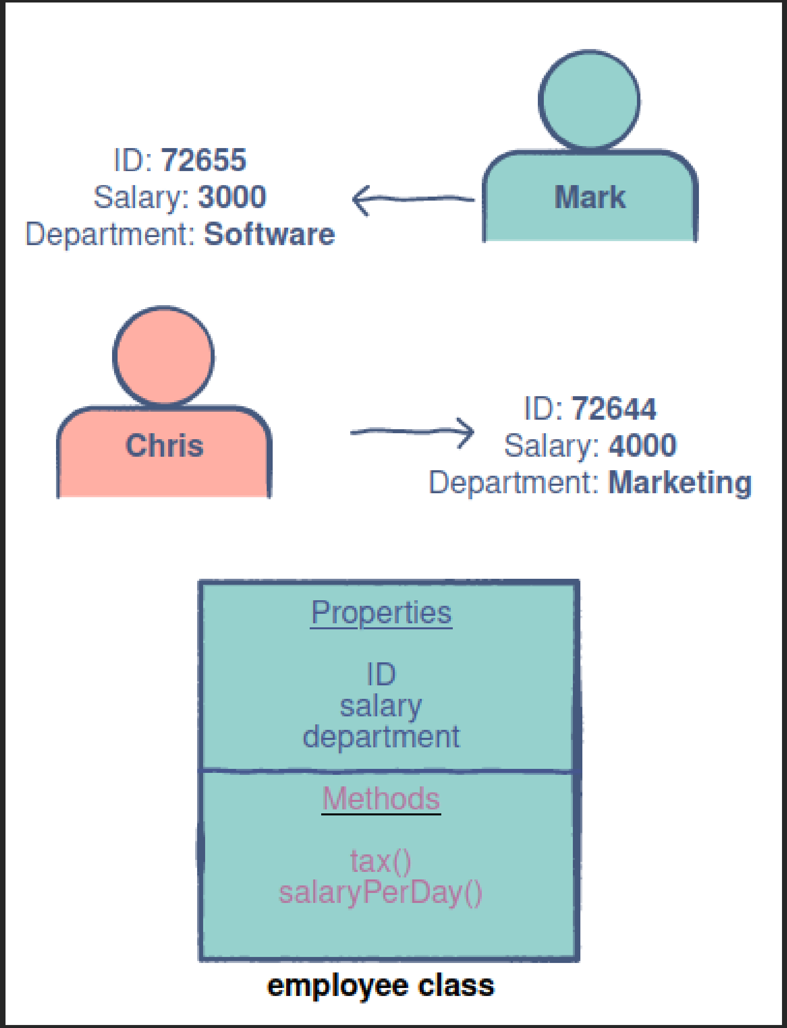


INTRODUCTION TO OBJECTS AND CLASSES

Attributes are also referred to as properties or members. For consistency, we will be using properties throughout the course.

Behaviors are also referred to as member functions or methods. For consistency, we will be using methods throughout the course.

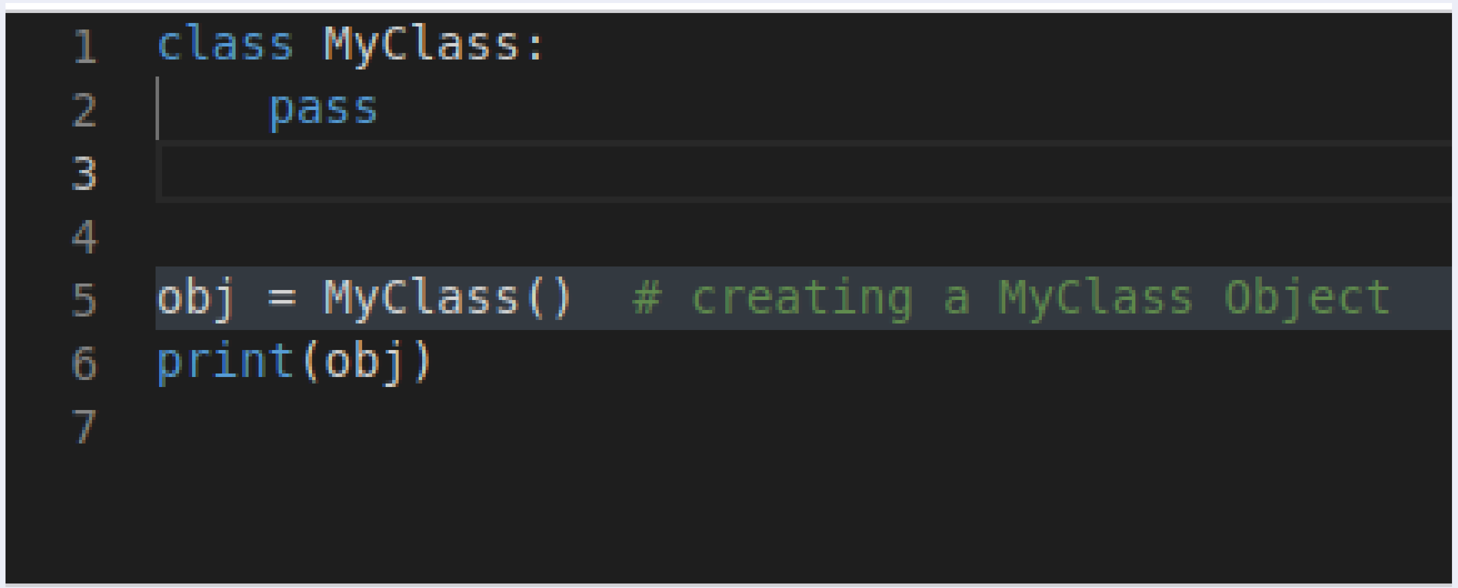




DECLARING A CLASS IN PYTHON

Must start with a *letter* or *underscore*

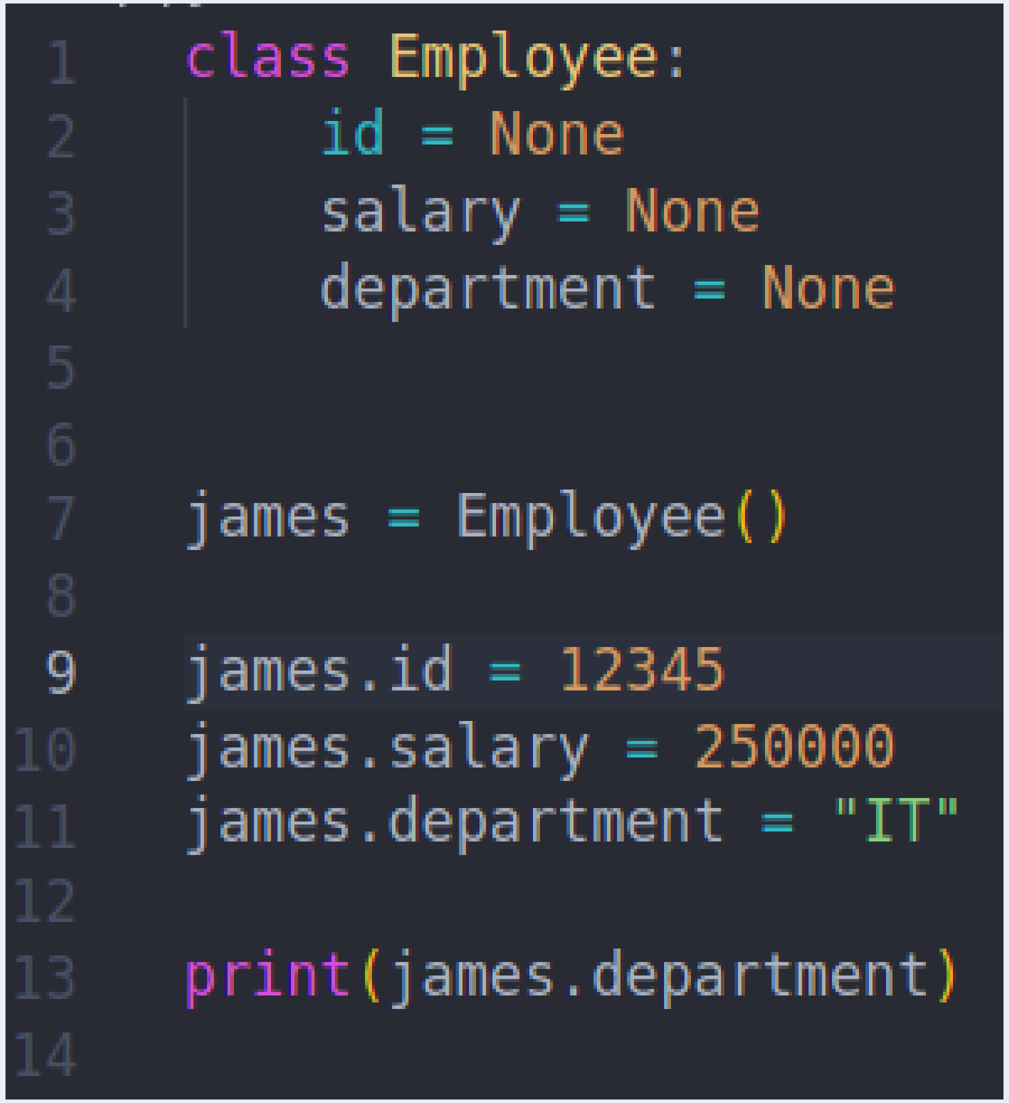
Should only be comprised of *numbers*, *letters*, or *underscores*



```python
class MyClass:
    pass


obj = MyClass()  # creating a MyClass Object
print(obj)

```

IMPLEMENTATION OF THE EMPLOYEE CLASS

To access properties of an object, the **dot** notation is used.



```python
class Employee:
    id = None
    salary = None
    department = None


james = Employee()

james.id = 12345
james.salary = 250000
james.department = "IT"

print(james.department)
```

INITIALIZING OBJECTS

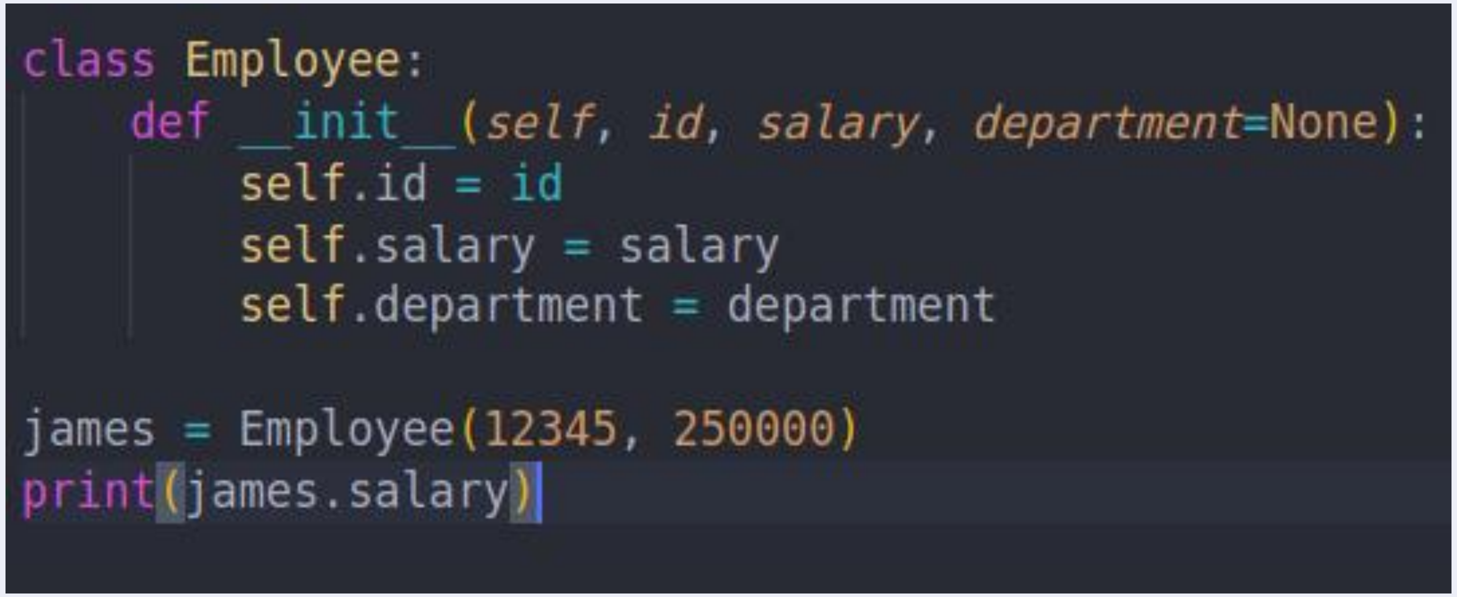
the **initializer (__init__)** is used to *initialize* an object of a class.

It's a special method that outlines the steps that are performed when an object of a class is created in the program.

It's used to define and assign values to **instance variables.**

The initializer is a special method because it **does not have a return type**.

The first parameter of **__init__** is **self**,



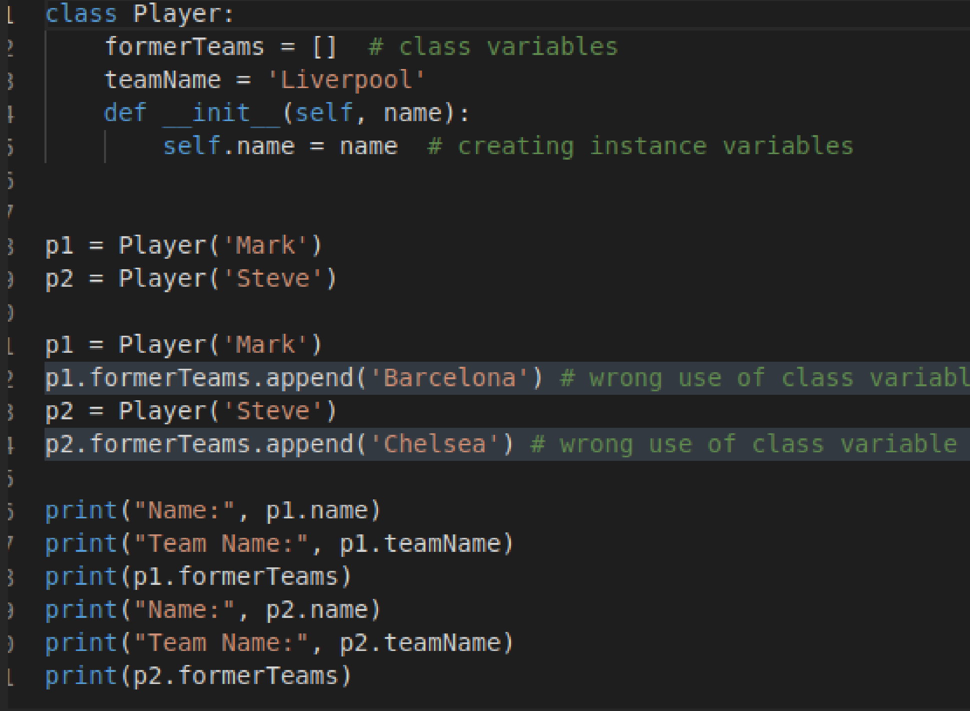
```python
class Employee:
    def __init__(self, id, salary, department=None):
        self.id = id
        self.salary = salary
        self.department = department

james = Employee(12345, 250000)
print(james.salary)
```

USE OF CLASS VARIABLES

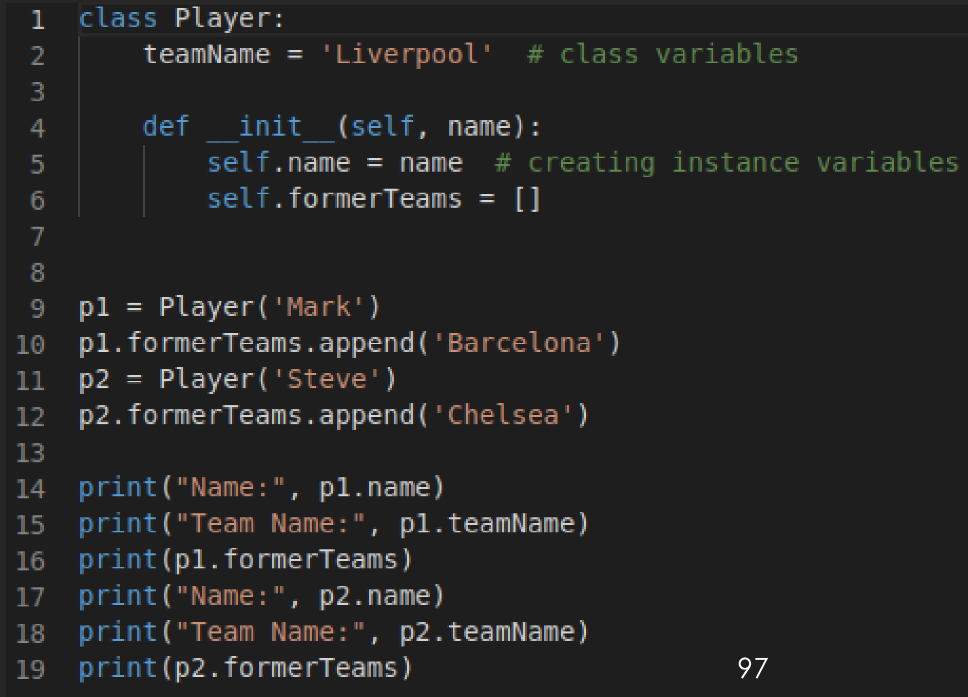
The **class variables** are shared by all instances or objects of the classes.

The **instance variables** are unique to each instance or object of the class.

Class variables are defined **outside** the initializer and instance variables are defined inside the initializer.



```python
class Player:
    formerTeams = []  # class variables
    teamName = 'Liverpool'
    def __init__(self, name):
        self.name = name  # creating instance variables


p1 = Player('Mark')
p2 = Player('Steve')

p1 = Player('Mark')
p1.formerTeams.append('Barcelona') # wrong use of class variabl
p2 = Player('Steve')
p2.formerTeams.append('Chelsea') # wrong use of class variable

print("Name:", p1.name)
print("Team Name:", p1.teamName)
print(p1.formerTeams)
print("Name:", p2.name)
print("Team Name:", p2.teamName)
print(p2.formerTeams)
```



```python
class Player:
    teamName = 'Liverpool'  # class variables

    def __init__(self, name):
        self.name = name  # creating instance variables
        self.formerTeams = []


p1 = Player('Mark')
p1.formerTeams.append('Barcelona')
p2 = Player('Steve')
p2.formerTeams.append('Chelsea')

print("Name:", p1.name)
print("Team Name:", p1.teamName)
print(p1.formerTeams)
print("Name:", p2.name)
print("Team Name:", p2.teamName)
print(p2.formerTeams)
```

IMPLEMENTING METHODS IN A CLASS

A **method** is a group of statements that performs some operations and may or may not return a result.

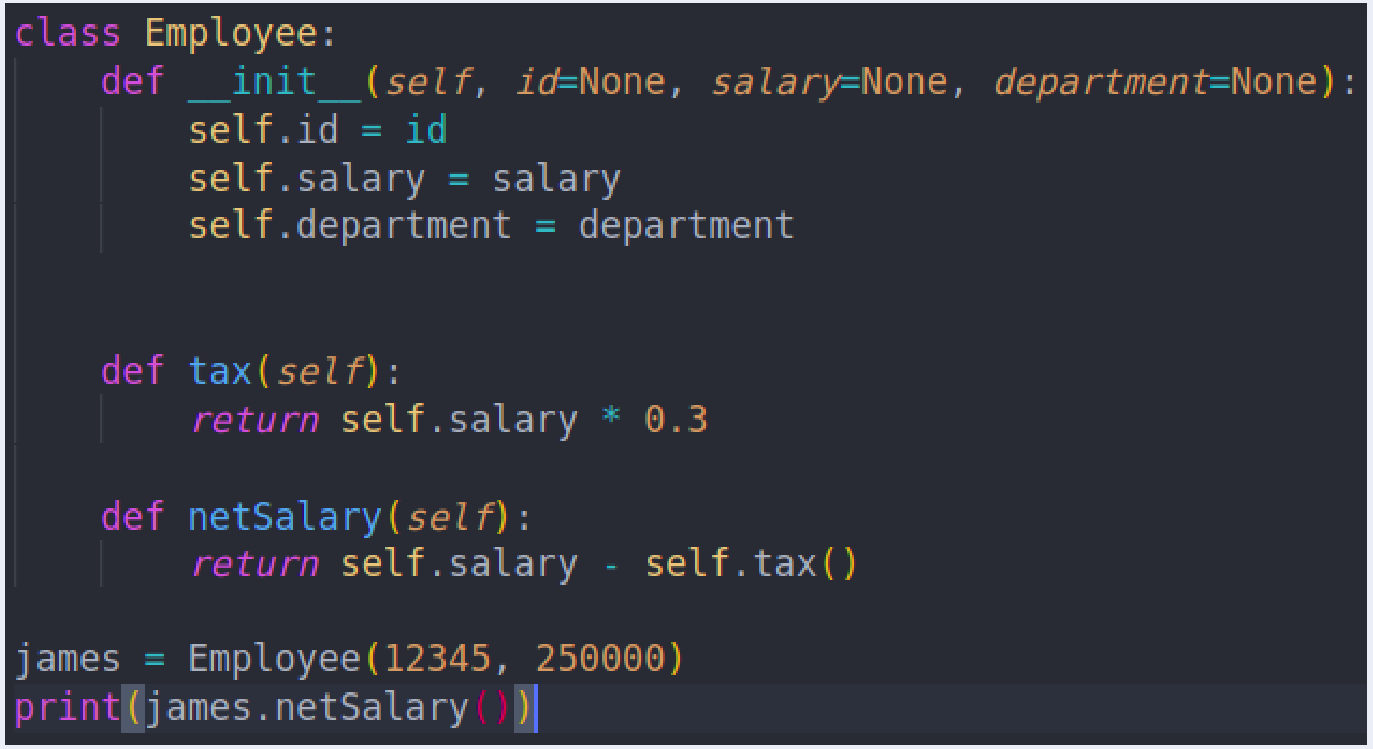
There are three types of methods in Python:

instance methods (mostly used in Python OOP)

class methods

static methods

These methods can either alter the content of the properties or use their v alues to perform a particular computation.



```
class Employee:
    def __init__(self, id=None, salary=None, department=None):
        self.id = id
        self.salary = salary
        self.department = department


    def tax(self):
        return self.salary * 0.3

    def netSalary(self):
        return self.salary - self.tax()

james = Employee(12345, 250000)
print(james.netSalary())
```

CLASS METHODS AND STATIC METHODS

Class methods are accessed using the class name and can be accessed without creating a class object.

we use the **@classmethod** decorator to declare a class method.

Just like instance methods, all class methods have **at least one** argument.

Static methods can be accessed using the class name or the object name.

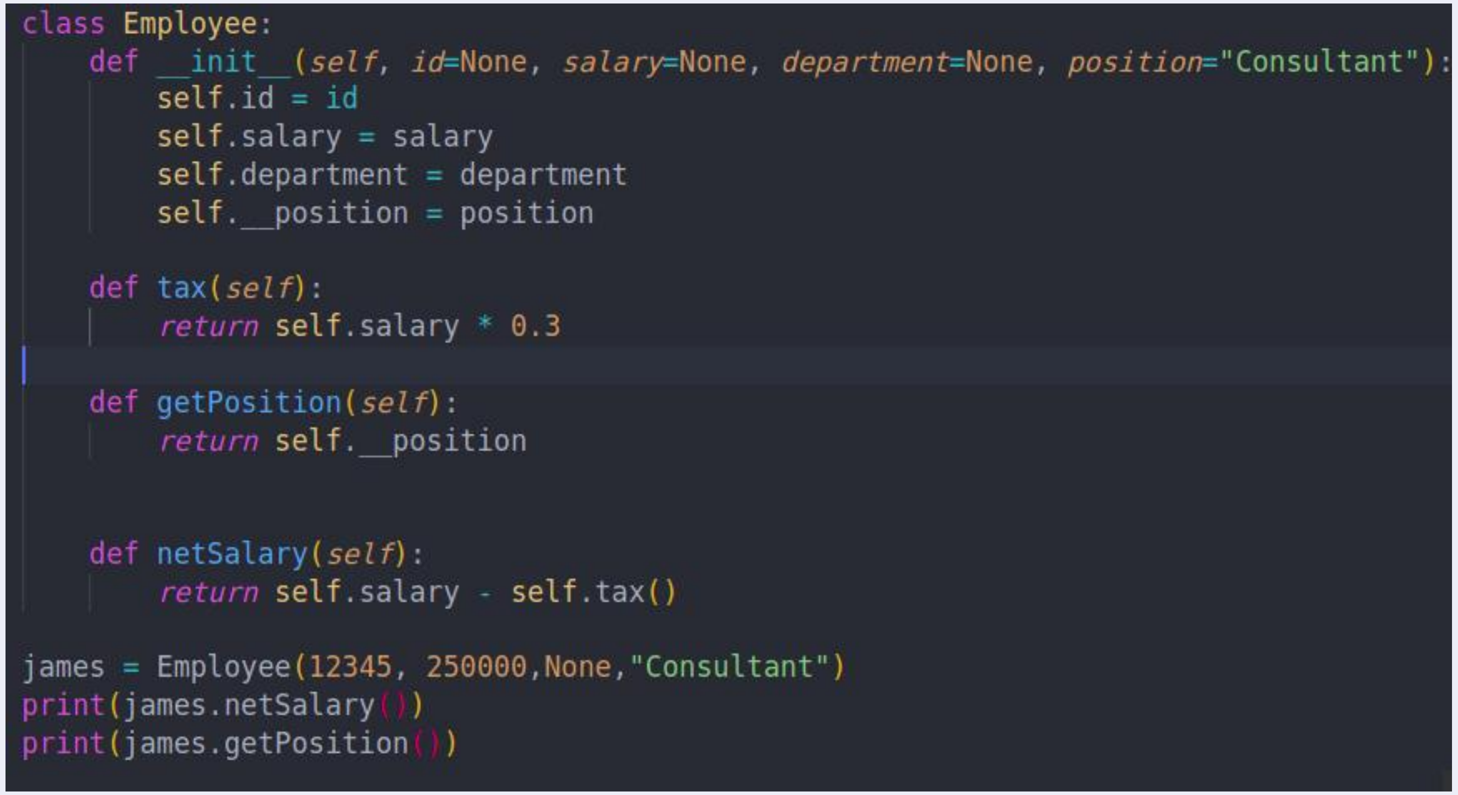
we use the **@staticmethod** decorator to declare a startic method

ACCESS MODIFIERS IN PYTHON

Public attributes are those that can be accessed inside the class and outside the class.

Private attributes cannot be accessed directly from outside the class but can be accessed from inside the class.

By default methods and attributes are public



```python
class Employee:
    def __init__(self, id=None, salary=None, department=None, position="Consultant"):
        self.id = id
        self.salary = salary
        self.department = department
        self.__position = position

    def tax(self):
        return self.salary * 0.3

    def getPosition(self):
        return self.__position


    def netSalary(self):
        return self.salary - self.tax()

james = Employee(12345, 250000,None,"Consultant")
print(james.netSalary())
print(james.getPosition())
```

PYTHON MODULES & PACKAGES

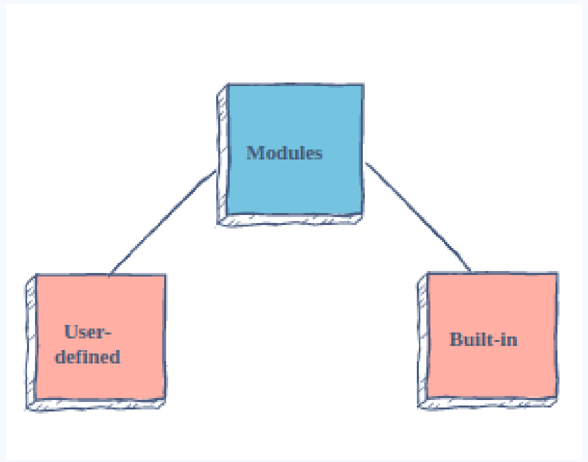
WHAT ARE PYTHON MODULES

A **Python module** is a Python file containing a set of functions and variables to be used in an application.

The variables can be of any type (arrays, dictionaries, objects, etc.)

Types of modules:

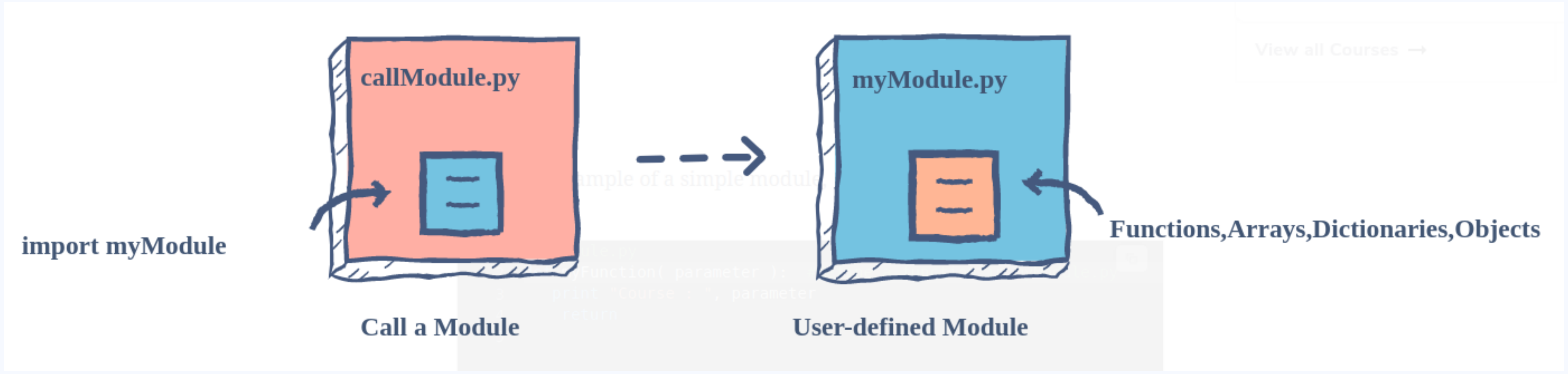
- Built-in
- User defined



USER-DEFINED MODULES

To create a module, create a Python file with a .py extension.

Modules created with a .py extension can be used in another Python source file, using the import statement.

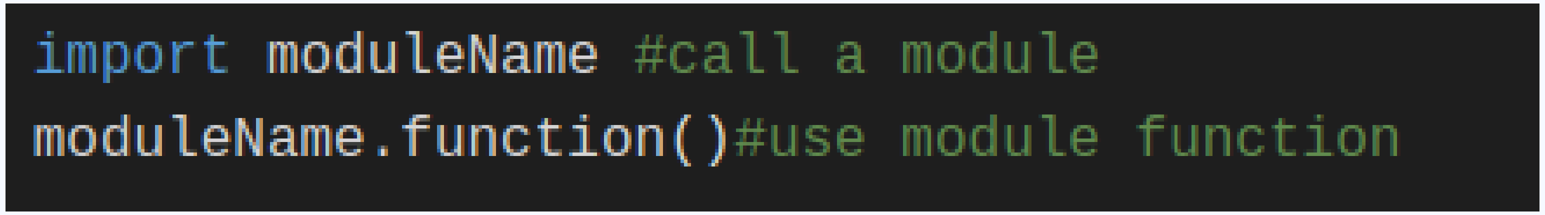


BUILT-IN MODULES

There are several built-in modules in Python, which you can import whenever you like.

Benefits of Modules in Python

- Structure Code
  - Code is logically organized by being grouped into one Python file which makes development easier and less error-prone.
  - Code is easier to understand and use.
- Reusability
  - Functionality defined in a single module can be easily reused by other parts of the application.



```
import moduleName #call a module
moduleName.function()#use module function
```

PYTHON PACKAGES

A package, is like a directory holding sub-packages and modules.

For a directory to become a python Package it needs to have an **__init__.py** inside it so that python can differentiate it from a normal directory.

We Can create our own packages or use publicly available packages

Python Package Index(PyPI): is a public repository for python packages

Python has a package manager called **pip** which is used to install packages

Packages can be installed system-wide(entire system) or withing a specific environment know as **virtual Environment.**

Here's the command used to install a package

pip install module-name

IMPORT MODULES IN PYTHON

To import modules we use the import statement

import module_name

Import module_name.member_name can be used to import some members(functions) within the module

From module_name import * with this statement we can import all the members of a module.

From module_name import member_name1, member_name2,.. Using the same approach you can select the members you want to import by separating each with a comma.

PYTHON VIRTUAL ENVIRONMENT

A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them.

Virtualenv is one of the modules used to create python environments. Alternatives are **venv, pipenv**, **conda**,....

Installing virtualenv: **pip install virtualenv**

To create a virtual environment with virtualenv: **virtualenv env_name**

After running this command, a directory named env_name will be created, and it will have all the necessary tools python needs to create a python environment.

ACTIVATING VIRTUAL ENVIRONMENT

A python virtual env. Needs to be activated. In the directory where you create the environment you can run the following command.

On windows: **env_name\Scripts\activate**

On linux/mac: **source env_name/bin/activate**

Once active the name of the environment will be displayed on the left side of your terminal / command line interface

Deactiving a virtual env:

deactivate

THANK YOU