What is Python?
Features of Python
Applications of Python
Python Installation
First Python Program

What is Python?

- Python is a **general-purpose**, **dynamic**, **high-level**, and **interpreted** programming language.
- It is designed to be **simple** and **easy to learn**, making it an ideal choice for beginners.
- One of the key strengths of Python is its versatility.
- Python supports the **object-oriented programming** approach, allowing developers to create applications with organized and reusable code.

Features of Python

- **Readability:** Python's syntax is designed to be clear and readable, making it easy for both beginners and experienced programmers to understand and write code.
- **Simplicity:** Python emphasizes simplicity and avoids complex syntax, making it easier to learn and use compared to other programming languages.
- **Dynamic Typing:** Python is dynamically typed, meaning you don't need to explicitly declare variable types.
- **Large Standard Library:** Python provides a vast standard library with ready-to-use modules and functions for various tasks, saving developers time and effort in implementing common functionalities.
- **Object-Oriented Programming(OOP):** Python supports the object-oriented programming paradigm, allowing for the creation and manipulation of objects, classes, and inheritance.
- **Cross-Platform Compatibility:** Python is available on multiple platforms, including Windows, macOS, and Linux, making it highly portable and versatile.
- **Extensive Third-Party Libraries:** Python has a vast ecosystem of third-party libraries and frame works that expand its capabilities in different domains, such as web development, data analysis, and machine learning.
- Interpreted Nature: Python is an interpreted language, meaning it does not require compilation. This results in a faster development cycle as code can be executed directly without the need for a separate compilation step.
- Integration Capabilities: Python can easily integrate with other languages like C, C++, and Java, allowing developers to leverage existing code bases and libraries

Applications of Python

Python is widely used in various domains and offers numerous applications due to its flexibility and ease of use. Here are some key areas where Python finds application:

- **Web Development:** Python is extensively used in web development frameworks such as Django and Flask. These frameworks provide efficient tools and libraries to build dynamic websites and web applications.
- **Data Analysis and Visualisation:** Python's rich ecosystem of libraries, including NumPy, Pandas, and Matplotlib, make it a popular choice for data analysis and visualization. It enables professionals to process, manipulate, and visualize data effectively.
- **Machine Learning and Artificial Intelligence:** Python has become the go-to language for machine learning and AI projects. Libraries like TensorFlow, Keras, and scikit-learn provide powerful tools for implementing complex algorithms and training models.
- **Automation and Scripting:** Python's easy-to-read syntax and rapid development cycle make it an ideal choice for automation and scripting tasks. It is commonly used for tasks such as file manipulation, data parsing, and system administration.

Python Installation

To download and install Python, follow these steps:

For Windows:

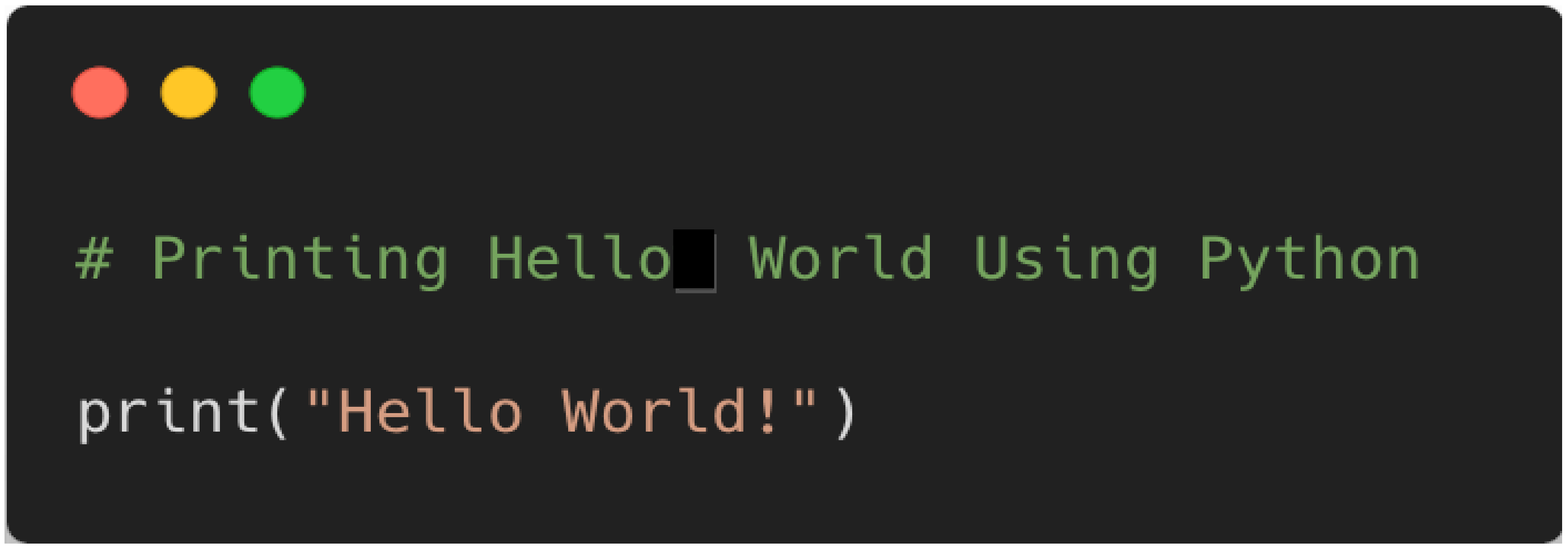
- Visit the official Python website at www.python.org/downloads/
- Download the Python installer that matches your system requirements.
- On the Python Releases for Windows page, select the linkfor the latest Python 3.x.x release.
- Scroll down and choose either the "Windows x86-64 executable Installer" for 64-bit or the "Windows x86 executable installer" for 32-bit.
- Run the downloaded installer and follow the instructions to install Python on your Windows system.

For Linux (specifically Ubuntu):

- Open the Ubuntu Software Center folder on your Linux system.
- Developer Tools.
- Locate the entry for Python3.x.x and double-click on it.
- Click on the Install button to initiate the installation process.
- Once the installation is complete, close the Ubuntu Software Center folder.

First Python Program

Writing your first Python program is an exciting step towards learning the language. Here's a simple example to get you started:



```python
# Printing Hello World Using Python

print("Hello World!")
```

Let's break down the code:

- The print() function is used to display the specified message or value on the console.
- In this case, we pass the string "Hello,World! "as an argument to the print() function. The string is enclosed in double quotes.
- The # symbol indicates a comment in Python. Comments are ignored by the interpreter and are used to provide explanations or notes to the code.

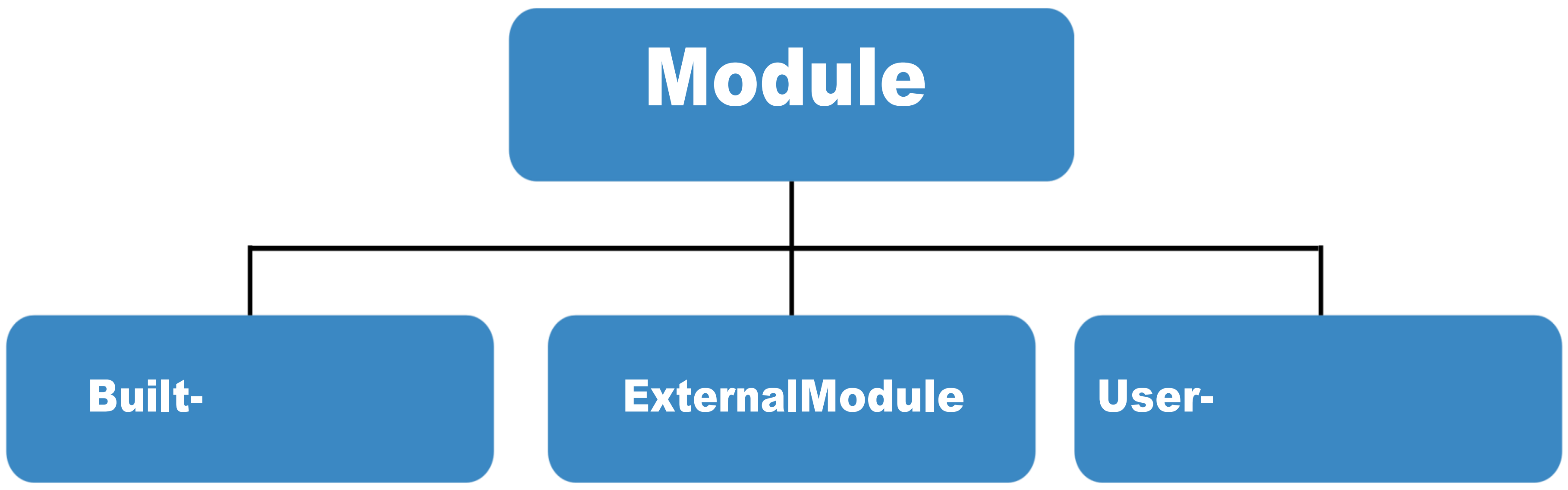
Topics:

- **Modules in Python**
- **Types of Modules**
- **Comments in Python**
- **PIP**

Modules in Python:

- Modules provide away to organize your code logically. Instead of having all your code in a single file, you can split it into multiple modules based on their purpose
- For example, you might have one module for handling input/output operations, another for mathematical calculations, and another for data manipulation.
- When you want to use the functionality from a module, you can import it into your current program or another module.
- This allows you to access and use the functions, classes, and variables defined within that module. By importing a module, you can avoid writing the same code repeatedly and instead reuse the code defined in the module.

Three Main Types of Modules



Built-in Modules:

- These are modules that come **pre-installed with Python** as part of the **standard library**.
- They provide a wide range of functionalities to simplify programming tasks.
- Common examples include:
  - math – for mathematical operations
  - random – for generating random numbers
  - datetime – for handling dates and times
  - os – for interacting with the operating system
- Built-in modules are **readily available** and do **not require separate installation**.

External Modules:

- These are modules created by **third-party developers** and are **not part of Python's standard library**.
- They extend Python's capabilities by adding functionalities tailored for specific domains.
- External modules can be **installed using package managers** like pip (Python Package Index).
- Popular examples include:
    - numpy – for numerical computations
    - pandas – for data manipulation and analysis
    - matplotlib – for data visualization
    - requests – for making HTTP requests

User-Defined Modules:

- These are modules **created by the programmers themselves** to organize code more effectively.
- They help in **modularizing and reusing** functionality across multiple scripts or projects.
- A user-defined module can include **functions, classes, variables**, and any other Python code.
- These modules are simply Python files (e.g., my_module.py) that can be **imported and reused** in other Python programs using the import statement.

Comments in Python

Comments in Python are used to **add explanatory notes within the code**. They are **not executed** or interpreted by the Python interpreter and serve only as documentation for developers. Comments are helpful for:

- Improving code readability.
- Leaving reminders or explanations for yourself or others working on the code.

Syntax

- Single-line comments start with a **hash symbol #**, followed by the comment text.

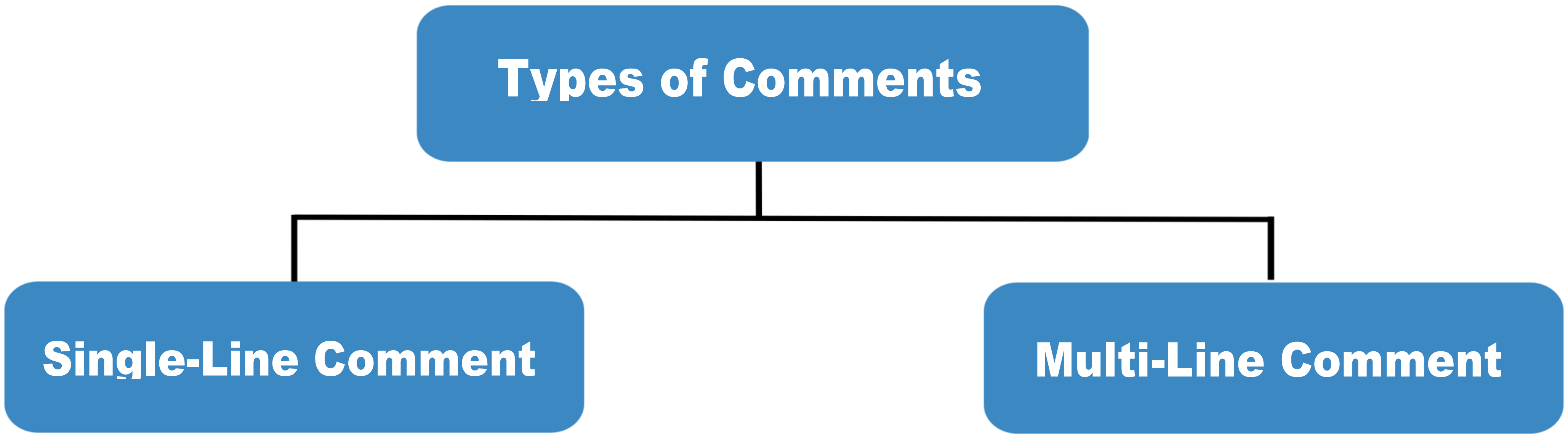
```
# This is a single-line comment
print("Hello, world!")  # This prints a message to the console
```

Important Notes:

- Comments are **ignored by the Python interpreter**.
- They have **no effect** on the program's functionality or performance.
- Use comments wisely to make your code **more maintainable and understandable**.

Using Triple Quotes (“’ or “””) (Acts as a string, not a true comment):

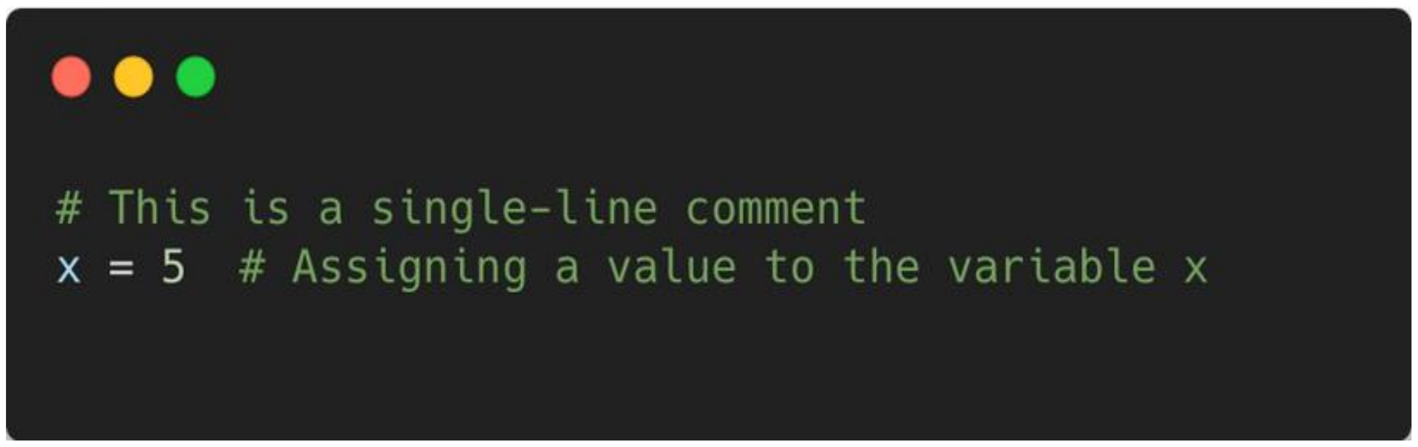
- Triple-quoted strings can also be used to span multiple lines.
  Although they are **meant for doc strings**, they can act like multi-line comments **if not assigned to a variable or used as a doc string**.



Single-line comments:

- Single-line comments are used to add explanatory notes or comments on a single line of code.
- They start with a hash symbol (#) and continue until the end of the line.
- Anything written after the hash symbol is considered a comment and is ignored by the Python interpreter.

Here's an example:

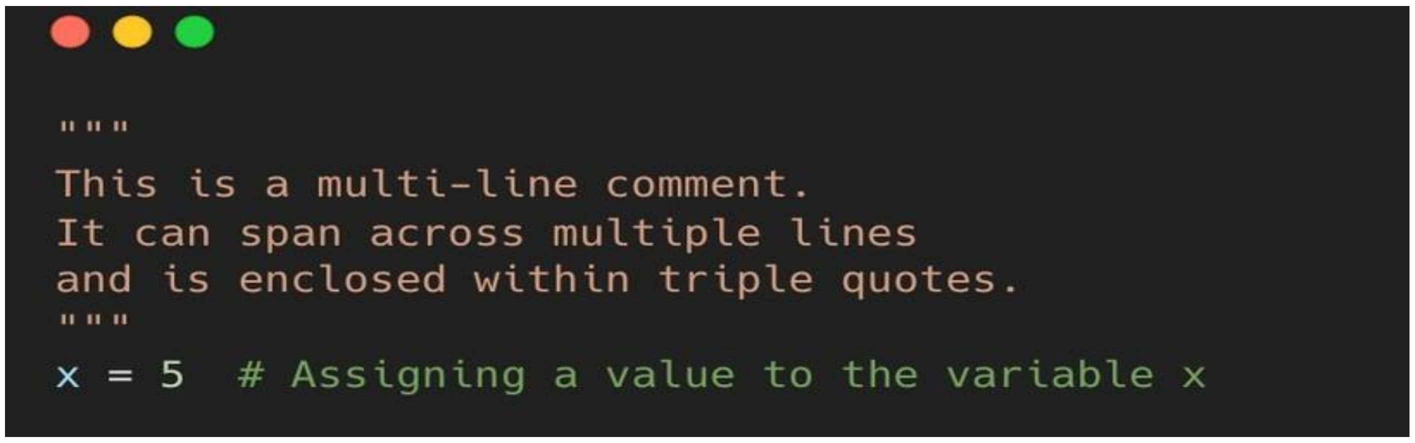


```python
# This is a single-line comment
x = 5  # Assigning a value to the variable x
```

Multi-line comments:

- Multi-line comments, also known as block comments, allow you to add comments that span multiple lines.
- Python does not have a built-in syntax specifically for multi-line comments, but you can achieve this by using triple quotes (either single or double quotes) to create a string that is not assigned to any variable. Since it is not used elsewhere in the code, it acts as a comment.

Here's an example:



```
"""
This is a multi-line comment.
It can span across multiple lines
and is enclosed within triple quotes.
"""
x = 5  # Assigning a value to the variable x
```

What is a pip?

- **pip** is a **package manager for Python**.
- The name **"pip"** stands for *"Pip Installs Packages"* or *"Pip Installs Python"*.
- It allows you to **install, upgrade, manage, and uninstall** external Python libraries and modules.
- These external packages extend Python's functionality beyond the standard library and are often created by the Python community.
- **pip** works with the **Python Package Index (PyPI)**, which hosts thousands of open-source libraries.

Why Use PIP?

- You need to use third-party packages like numpy, pandas, flask, etc.
- pip makes the installation process as easy as a single command.

Command	Description		
`pip install package_name`	Installs a package		
`pip install package==1.2.3`	Installs a specific version		
`pip install -r requirements.txt`	Installs all packages listed in a file		
`pip list`	Shows installed packages		
`pip show package_name`	Shows details about a package		
`pip uninstall package_name`	Removes a package		

Topics:

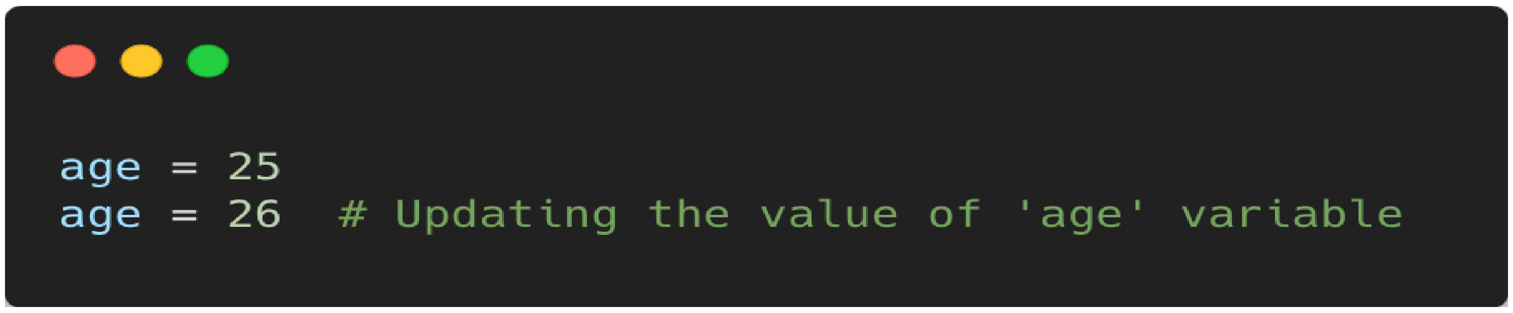
- **Variables**
- **Data Types**
- **Keywords**

Variables in Python

- In Python, variables are used to store values that can be used later in a program. You can think of variables as containers that hold data "="operator.
- For example, you can create a variable called "name" and assign it a value like this:

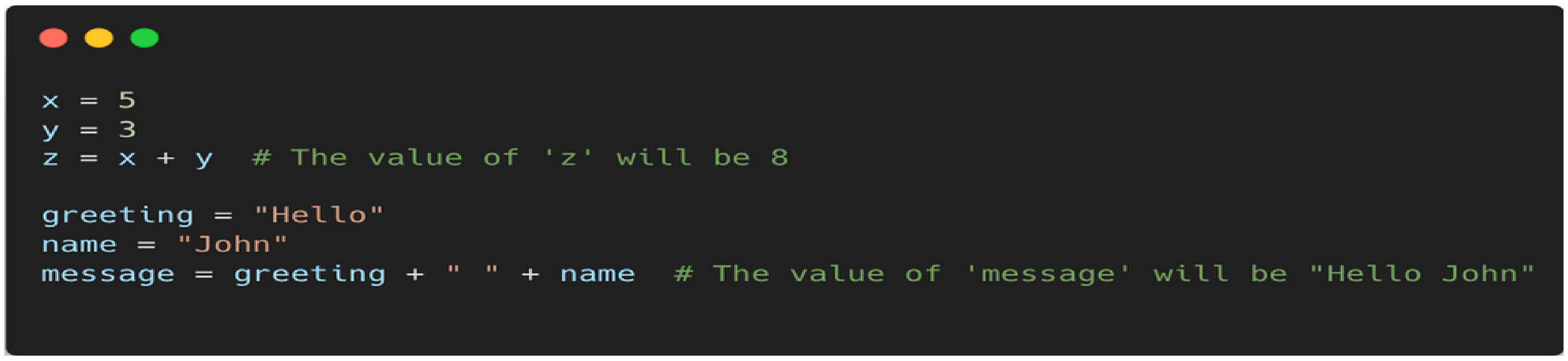
```
name = “Sailesh”
```

Here, "name" is the variable name, and "Sailesh" is the value assigned to it. Python will automatically determine the type of the variable based on the value assigned to it. In this case, the type of the variable "name" is a string.
Variables in Python can hold different types of data, such as numbers, strings, lists, or even more complex objects.
You can change the value of a variable at any time by assigning an value to it.
For instance:



```
age = 25
age = 26   # Updating the value of 'age' variable
```

Python also allows you to perform operations on variables. For example, you can add, subtract, multiplication, division, modulo division with containing numbers. You can even different types using operators.
For instance:



```
x = 5
y = 3
z = x + y  # The value of 'z' will be 8

greeting = "Hello"
name = "John"
message = greeting + " " + name  # The value of 'message' will be "Hello John"
```

Variables provide a way to store and manipulate data in Python, making it easier to work with information throughout your program. By giving meaningful names to variables, you can make your code more readable and easy to understand.

Identifier in Python

In Python, an identifier is a name used to identify a variable, function, class, module, or any other user-defined object. An identifier can be made up of letters (both uppercase and lowercase), digits, and underscores (_). However, it must start with a letter or an underscore.

Here are some important rules to keep in mind when working with identifiers in Python:

- **Valid Characters:**
  (A-Z), digits(0-9), and underscore not contain spaces or special characters like @, #, or $.
- **Case Sensitivity:**
  Python is case-sensitive, meaning uppercase and lowercase letters are considered different. So, "myVar" and "myvar" are treated as two different identifiers.
- **Reserved Words:**
  Python has reserved words, also known as keywords, that have predefined meanings in the language. These words cannot be used as identifiers.

  Examples of reserved words include "if", "while", and "def."
- **Length:**
  Identifiers can be of any length. However, it is recommended to use meaningful and descriptive names that are not excessively long.
- **Readability:**
  It is good practice to choose descriptive names for identifiers that convey their purpose or meaning. This helps make the code more readable and understandable.
  Here are some examples of valid identifiers in Python:
  - my_variable
  - count
  - total_sum
  - PI
  - MyClass
- And here are some examples of invalid identifiers:
  - 123abc (starts with a digit)
  - my-variable(contains a hyphen)
    if(a reserved word)
  - myvar(containsaspace)

Data Types in Python

Data types in Python refer to the different kinds of values that can be assigned to variables. They determine the nature of the data and the operations that can be performed on them. Python provides several built-in data types, including:

Numeric:

Python supports different numerical data types, including integers (whole numbers), floating-point numbers (decimal numbers), and complex numbers (numbers with real and imaginary parts).

a. **Integers (int):** Integers represent whole numbers without any fractional part. For example, age = 25.
b. **Floating-Point Numbers(float):**Floating-point numbers represent numbers with decimal points or fractions. For example, pi=3.14.
c. **Complex Numbers(complex):**Complex numbers have

With I or J. For example, z=2+3j.

Dictionary:

Dictionaries are key-value pairs enclosed in curly braces. Each value is associated with a unique key, allowing for efficient lookup and retrieval. For example,
person ={'name':'John','age':25,'city':'NewYork'}.

Boolean:

Boolean (bool): Booleans represent truth values, either True or False. They are used for logical operations and conditions. For example, is_valid =True.

Set:

Sets (set): Sets are unordered collections of unique elements enclosed in curly braces. They are useful for mathematical operations such as union, intersection, and difference. For example, fruits = {'apple', 'banana', 'orange'}.

Sequence Type:

Sequences represent a collection of elements and include data types like strings, lists, and tuples. Strings are used to store textual data, while lists and tuples are used to store ordered collections of items.

- **Strings (str):** Strings represent sequences of characters enclosed within single or double quotes. For example, name = "John".
- **Lists (list):** Lists are ordered sequences of elements enclosed in square

brackets. Each element can be of any data type.
For example, numbers = [1, 2, 3, 4].

- **Tuples (tuple):** Tuples are similar to lists but are immutable, meaning their elements cannot be changed once defined. They are enclosed in parentheses. For example, coordinates=(3,4).

Keywords in Python

- Keywords in Python are special words that have specific meanings and purposes within the Python language.
- They are reserved and cannot be used as variable names or identifiers.
- Keywords play a crucial role in defining the structure and behavior of Python programs.
- Keywords are like building blocks that allow us to create conditional statements, They help in controlling the flow of the program and specify how different parts of the code should behave.
- For example,
- the if key word is used to check conditions and perform specifications based on those conditions.
- The for and while key words are used to create loops that repeat a block of code multiple times.
- The def keyword is used to define functions, which are reusable blocks of code that perform specific tasks.

List of all the keywords in Python:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Topic:

- **Arithmetic operators**
- **Comparison operators**
- **Assignment operators**
- **Logical operators**
- **Bitwise operators**
- **Membership operators**
- **Identity Operators**

Operators in Python

Operators in Python are symbols or special characters that are used to perform specific operations on variables and values.
Python provides various types of operators to manipulate and work with different data types. Here are some important categories of operators in Python:

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators:

Arithmetic operators in Python are used to perform mathematical calculations on numeric values. The basic arithmetic operators include:

- **Addition (+):** Adds two operands together. For example, if we have a=10 and b=10, then a+bequals20.
- **Subtraction (-):** Subtracts the second operand from the first operand. If the first operand is smaller than the second operand, the result will be negative. For example, if we have a = 20 and b = 5, then a - b equals 15.
- **Division (/):** Divides the first operand by the second operand and returns the quotient. For example, if we have a = 20 and b = 10, then a / b equals 2.0.
- **Multiplication (*):** Multiplies one operand by the other. For example, if we have a = 20 and b = 4, then a * b equals 80.
- **Modulus (%):** Returns the remainder after dividing the first operand by the second operand. For example, if we have a=20 and b=10, then a%b equals 0.
- **Exponentiation (**) or Power:** Raises the first operand to the power of the second operand. For example, if we have a=2 and b=3, then a**b equals 8.
- **Floor Division (//):** Provides the floor value of the quotient obtained by dividing the two operands. It returns the largest integer that is less than or equal to the result. For example, if we have a=20 and b=3, then a//b equals 6.

Comparison Operators:

Comparison operators in Python are used to compare two values and return a Boolean value (True or False) based on the comparison. Common comparison operators include:

- Equal to (==): Checks if two operands are equal.
- Not equal to (!=): Checks if two operands are not equal.
- Greater than (>): Checks if the left operand is greater than the right operand.
- Less than (<): Checks if the left operand is less than the right operand.
- Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.
- Less than (<): Checks if the left operand is less than the right operand.
- Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.
- Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

Assignment Operators:

Assignment operators are used to **assign values** to variables. Python supports simple as well as compound assignment operators.

Operator	Example	Equivalent To	Description
`+=`	`x += 3`	`x = x + 3`	Adds and assigns
`-=`	`x -= 2`	`x = x - 2`	Subtracts and assigns
`*=`	`x *= 4`	`x = x * 4`	Multiplies and assigns
`/=`	`x /= 5`	`x = x / 5`	Divides and assigns (float result)
`//=`	`x //= 2`	`x = x // 2`	Integer divides and assigns
`%=`	`x %= 3`	`x = x % 3`	Modulus and assigns
`**=`	`x **= 2`	`x = x ** 2`	Exponentiation and assigns
`&=`	`x &= y`	`x = x & y`	Bitwise AND and assigns
`	=`	`x	= y`
`^=`	`x ^= y`	`x = x ^ y`	Bitwise XOR and assigns
`>>=`	`x >>= 1`	`x = x >> 1`	Right shift and assigns
`<<=`	`x <<= 1`	`x = x << 1`	Left shift and assigns

Logical Operators:

Logical operators in Python are used to perform logical operations on Boolean values. The main logical operators are:

- Logical AND (and&&): Returns True if both operands are True, otherwise False.

- Logical OR (or||): Returns True if at least one of the operands is True, otherwise False.
- Logical NOT (not !): Returns the opposite Boolean value of the operand.

Bitwise Operators:

Bitwise operators perform operations on individual bits of binary numbers. Some common bitwise operators in Python are:

- Bitwise AND (&): Performs a bitwise AND operation on the binary representations of the operands.
- Bitwise OR (|): Performs a bitwise OR operation on the binary representations of the operands.
- Bitwise XOR (^): Performs a bitwise exclusive OR operation on the binary representations of the operands.
- Bitwise complement (~): Inverts the bits of the operand.
- Left shift (<<): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
- Right shift (>>): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

Membership Operators:

Membership operators are used to test whether a value is a member of a sequence (e.g., string, list, tuple). They include:

- In: Returns True if the value is found in the sequence.
- Not in: Returns True if the value is not found in the sequence.

Identity Operators:

Identity operators are used to compare the identity of two objects. They include:

- Is: Returns True if both operands refer to the same object.
- Is not: Returns True if both operands do not refer to the same object.

Topics:

- **If statement**
- **Else and Elif statements**
- **Nested If statement**
- **While Loop**
- **For Loop**
- **Loop control statements**
- **Use range() in For Loop**

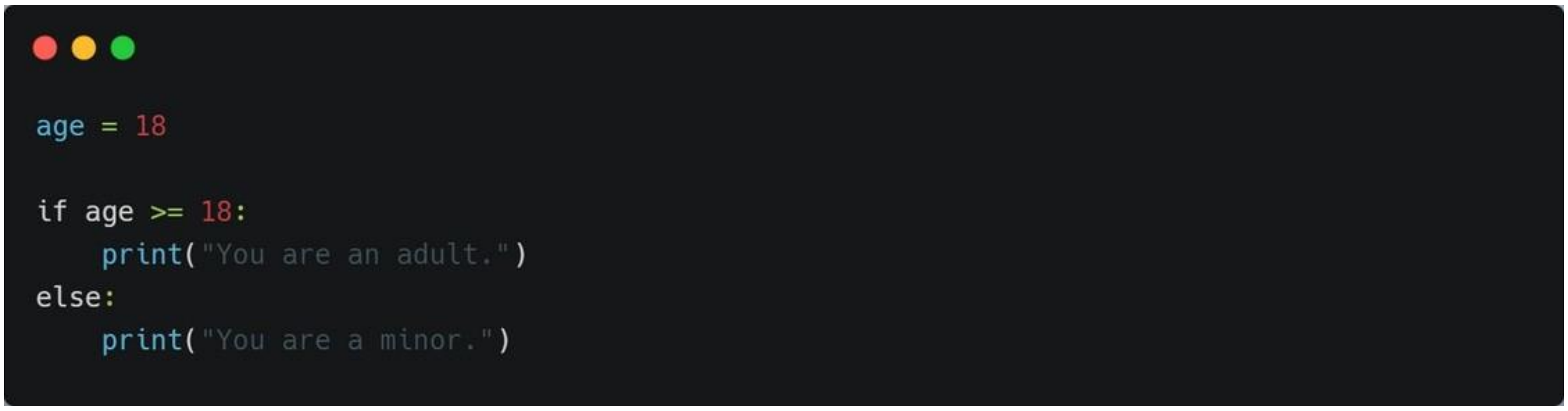
Control Flow

Control flow directs program execution through structures like loops, conditionals, and functions, determining the order and path of operations.

If statements

An `if` statement in python checks whether a condition is true or false. If the condition is true, the code inside the `if` block runs. If false, the code is skipped. It's used to make decisions in the program, executing specific actions based on conditions

example:



```python
age = 18

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

Else and elif statements

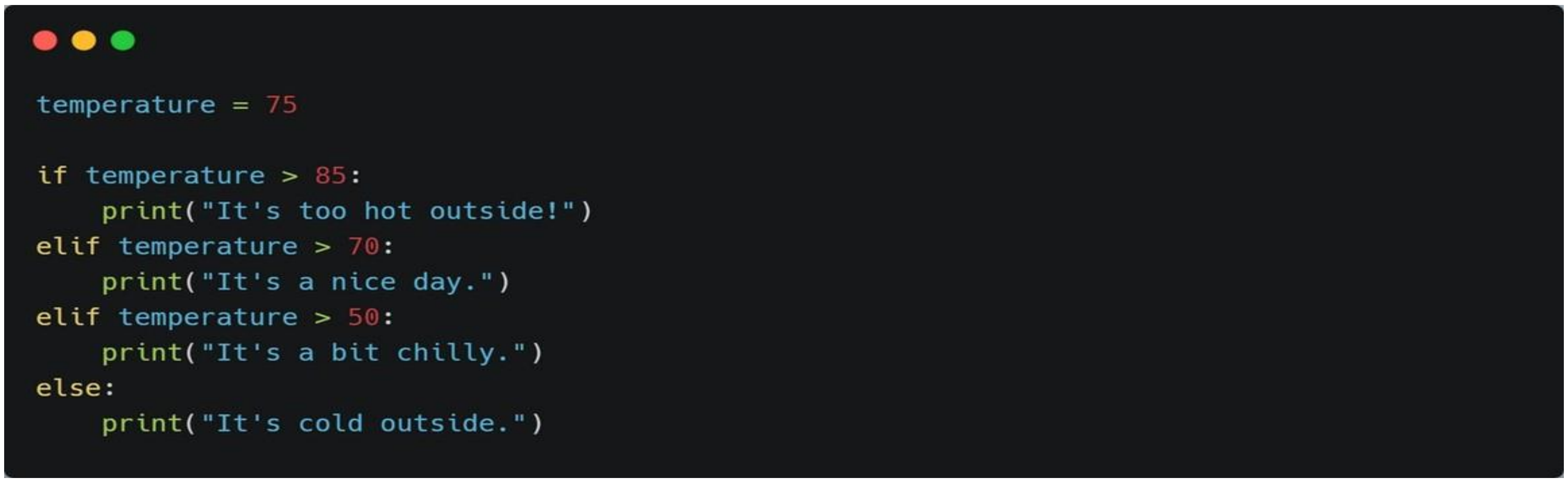
In Python, else and elif statements are used along side if to handle multiple conditions and alternative actions.

elif (else if): Checks another condition if the previous if was false. You can have multiple elif statements.

else: Runs when none of the if or elif conditions are true.

It's the "default" action.

example:



```python
temperature = 75

if temperature > 85:
    print("It's too hot outside!")
elif temperature > 70:
    print("It's a nice day.")
elif temperature > 50:
    print("It's a bit chilly.")
else:
    print("It's cold outside.")
```

Nested if Statement

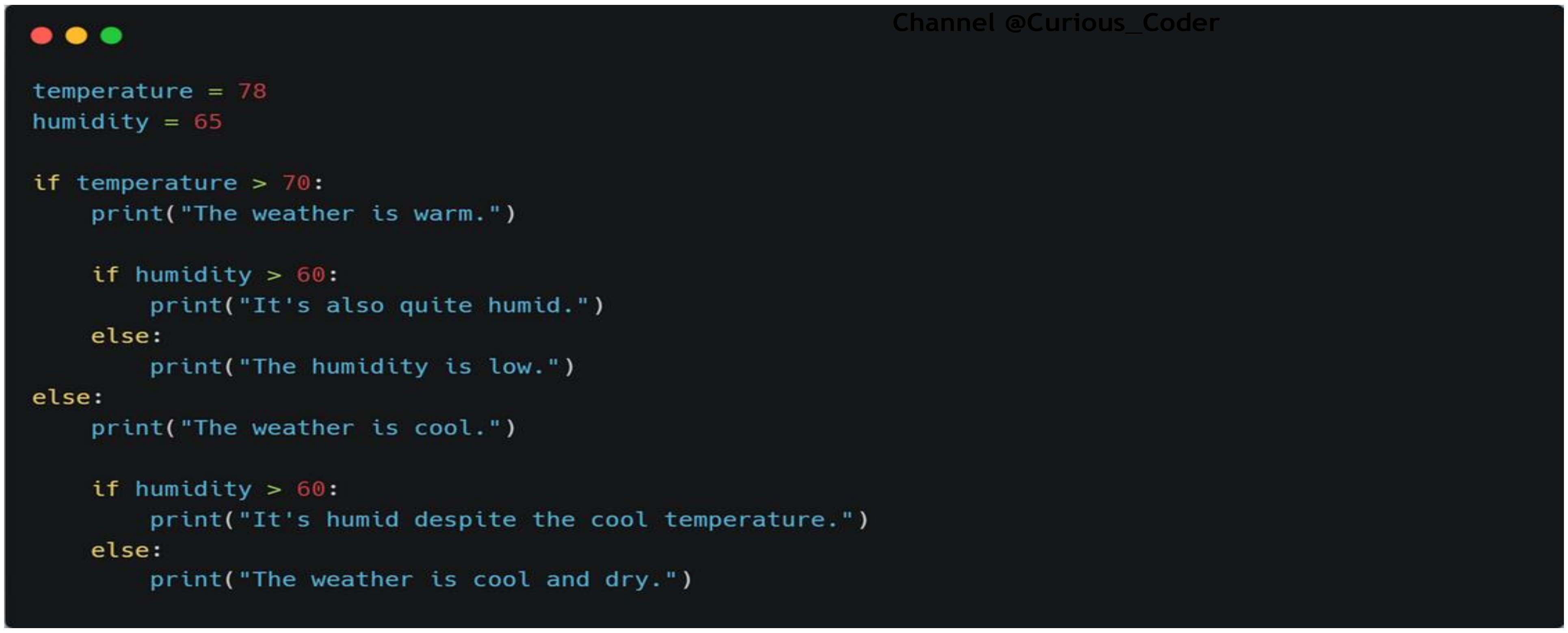
A nested `if` statement in python is an `if` statement inside another `if`. It lets you check multiple related conditions in sequence.

For example:

If you first check the weather and it's sunny, you can then check how many guests are coming. Depending on the number of guests, you decide between different activities, like a barbeque or picnic.

If the weather isn't sunny, you skip the nested checks and go straight to an alternative action, like staying indoors**.**

example:



```python
temperature = 78
humidity = 65

if temperature > 70:
    print("The weather is warm.")

    if humidity > 60:
        print("It's also quite humid.")
    else:
        print("The humidity is low.")
else:
    print("The weather is cool.")

    if humidity > 60:
        print("It's humid despite the cool temperature.")
    else:
        print("The weather is cool and dry.")
```

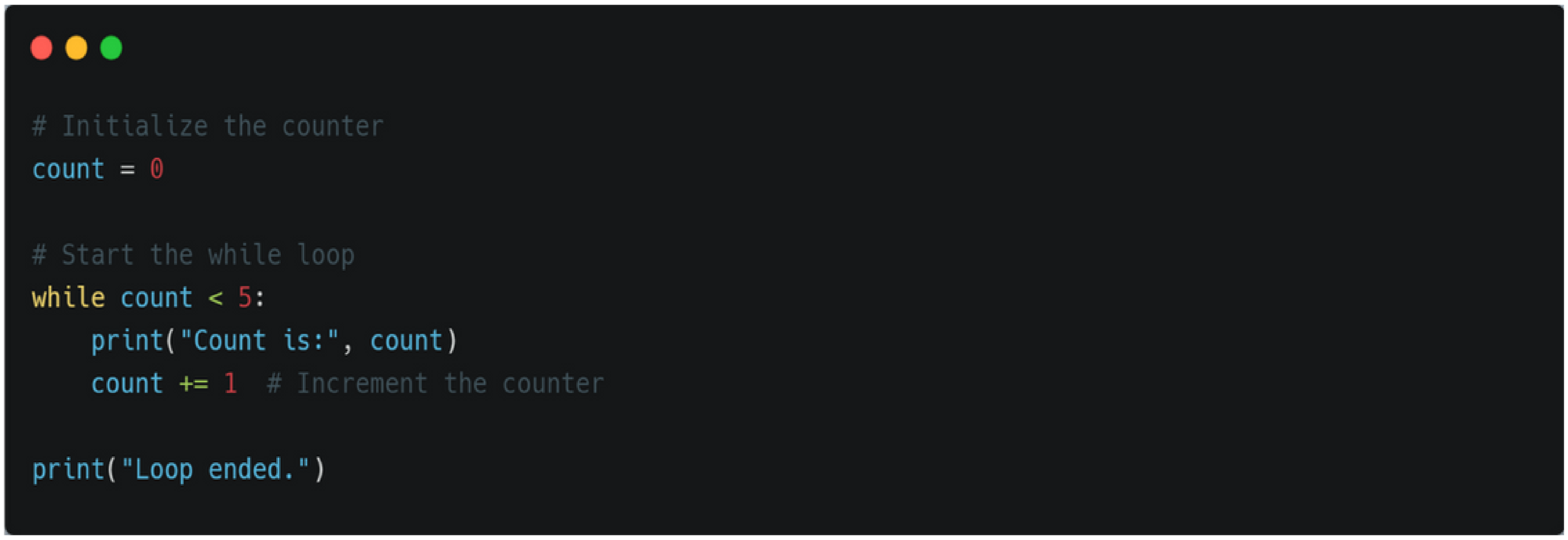
The While Loop

A `while` loop in Python repeatedly executes a block of code as long as a specified condition is true.

It first checks the condition, if true, the code inside runs. After each iteration, the condition is rechecked. The loop continues until the condition becomes false.

For example

A `while` loop can keep counting up as long as the count is below a certain number.



```python
# Initialize the counter
count = 0

# Start the while loop
while count < 5:
    print("Count is:", count)
    count += 1  # Increment the counter

print("Loop ended.")
```

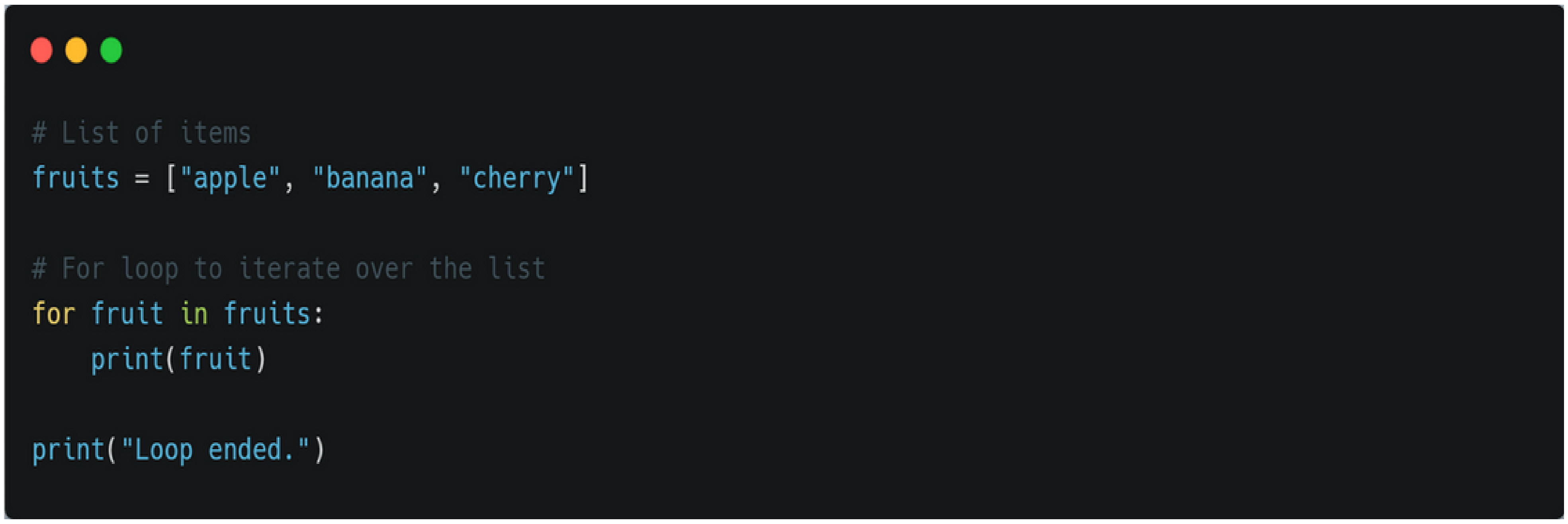
The For Loop

A `for` loop in Python is used to iterate over a sequence, such as a list, tuple, or range, executing a block of code for each item in the sequence. Unlikea `while` loop, which runs until a condition is false, a `for` loop runs a set number of times based on the length of the sequence.

For example

It can go through a list of numbers, processing each one in turn. It's ideal for repetitive tasks like iterating over data collections.

example:



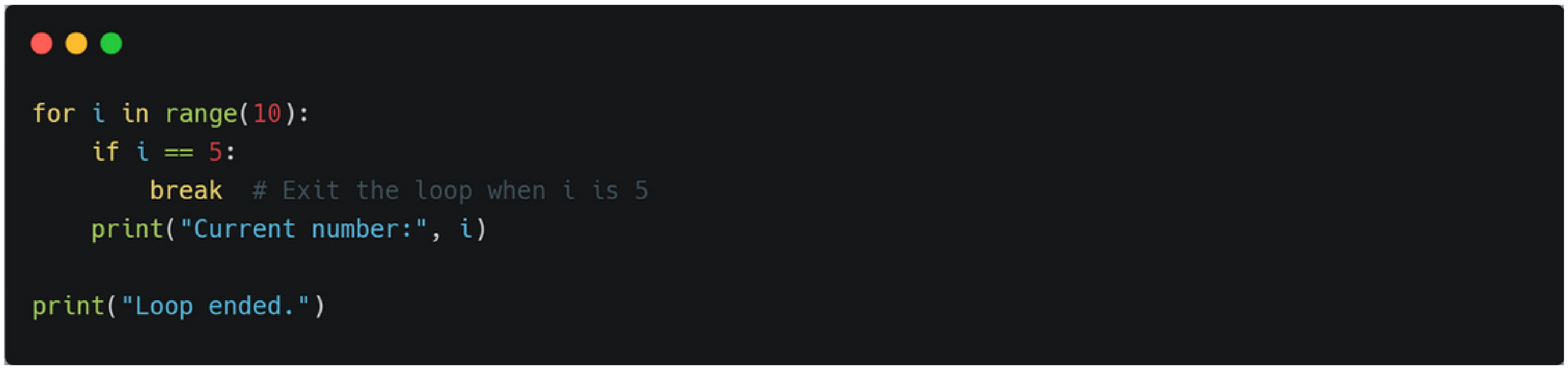
```python
# List of items
fruits = ["apple", "banana", "cherry"]

# For loop to iterate over the list
for fruit in fruits:
    print(fruit)

print("Loop ended.")
```

Loop Control Statements

Loop control statements in Python allow you to alter the flow of a loop’s execution. They include:

`break`: This statement immediately exits the loop, regardless of whether the loop's condition is still true. It’s useful for stopping a loop when a specific condition is met, like when searching for an item in a list and finding it before the loop has iterated through the entire list.

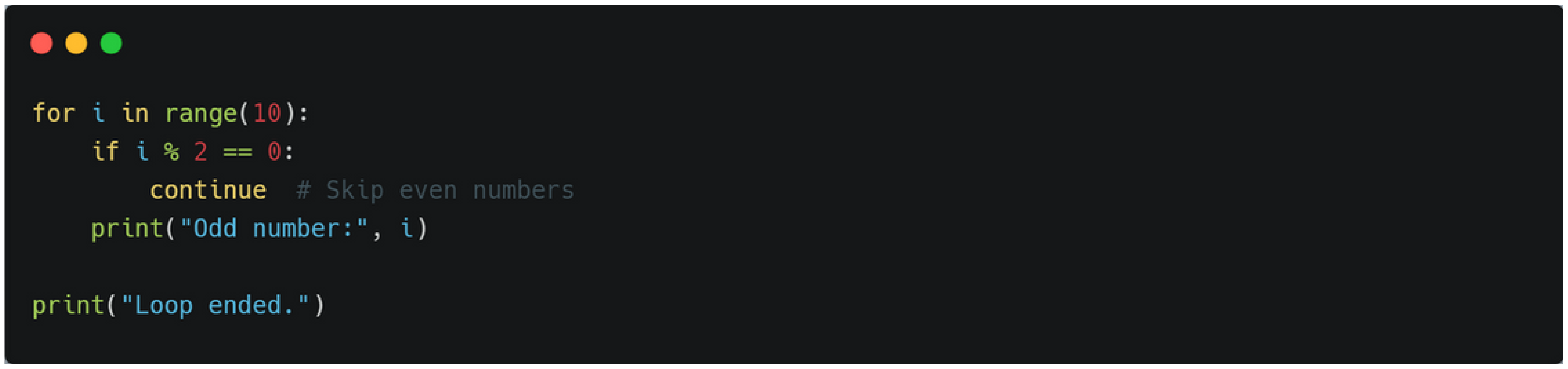
example:



```python
for i in range(10):
    if i == 5:
        break  # Exit the loop when i is 5
    print("Current number:", i)

print("Loop ended.")
```

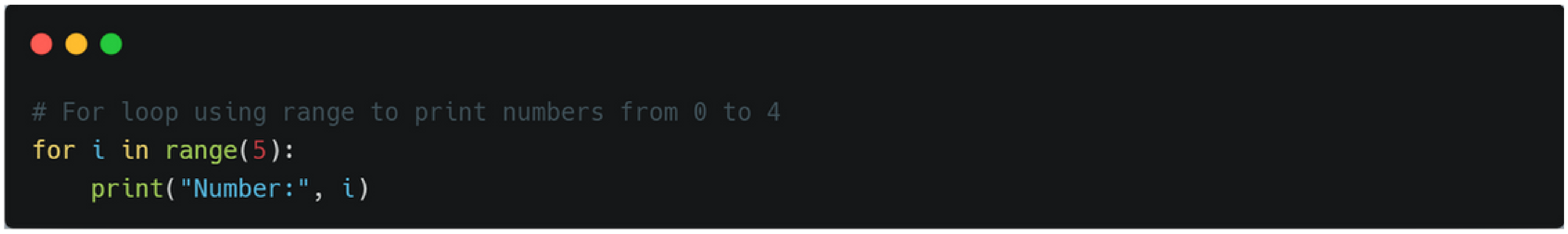
'continue': This statement skips the rest of the current loop iteration and proceeds to the next iteration. It’s helpful for by passing certain parts of the loop based on a condition, like skipping even numbers in a loop that processes a range of numbers.

Example :



```python
for i in range(10):
    if i % 2 == 0:
        continue  # Skip even numbers
    print("Odd number:", i)

print("Loop ended.")
```

The range() function in Python is commonly used in for loops to iterate over a sequence of numbers.

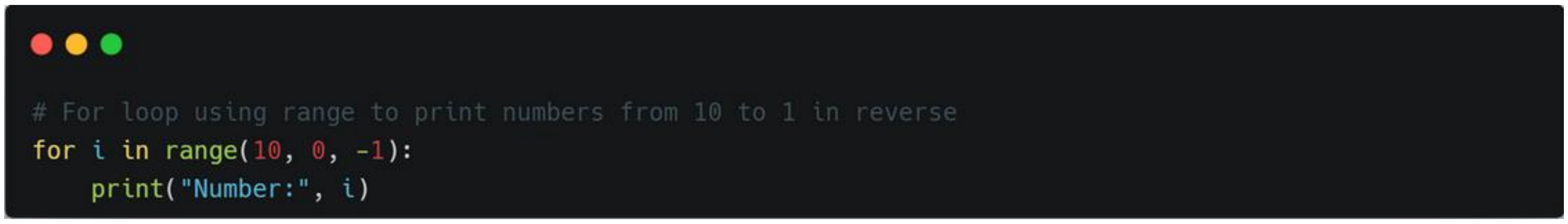


```python
# For loop using range to print numbers from 0 to 4
for i in range(5):
    print("Number:", i)
```

Here's a basic run down of how it works:

start: The starting value of the sequence (inclusive). If omitted, it defaults to 0.
stop: The ending value of the sequence (exclusive). The loop will run until it reaches this value.
step: The amount by which the sequence is incremented. If omitted, it defaults to 1.
You can use range() in a for loop for various tasks like iterating through lists, generating sequences of numbers, or performing repetitive actions a specific number of times.

example:



```python
# For loop using range to print numbers from 10 to 1 in reverse
for i in range(10, 0, -1):
    print("Number:", i)
```

Topics:

- **list and Methods**
- **Indexing and Slicing**
- **List Comprehension**
- **Tuples**
- **Tuples packing, and unpacking**

Lists and Tuples

- Lists and tuples in Python both store collections of items. Lists are mutable, allowing changes like adding or removing elements, and use square brackets (`[]`).
- Tuples are immutable, meaning they can't be modified after creation, and use parentheses (`()`). Use lists for dynamic data and tuples for fixed collections.

Lists and Methods

Lists in Python are ordered, mutable collections used to store multiple items in a single variable. They are defined using square brackets ([ ]) and can contain elements of different data types, such as integers, strings, or even other lists. You can add, remove, or modify elements within a list, making them highly versatile for managing dynamic datasets.

Key Features:

- Mutable: Elements can be changed, added, or removed.
- Ordered: Elements have a defined order, and you can access them using indices, starting from 0.

Basic Operations:

- **Create a list:** my_list=[1,2,3,4]
- **Access elements:** my_list[0](returns1)
- **Modify elements:** my_list[1]="new_value"
- **Append elements:** my_list.append(5)
- **Remove elements:** my_list.remove(2) (removesthefirstoccurrenceof2)
- **append()**

  Adds a single element to the end of the list.

  Syntax: list.append(element)
- **extend()**
  Extends the list by appending elements from an iterable (like another list).

  Syntax: list.extend(iterable)
- **insert()**

  Inserts an element at a specified position in the list.

  Syntax: list.insert (index,element)

- **remove()**
  Removes the first occurrence of a specified element from the list.

  Syntax: list.remove(element)
- **pop()**
  Removes and returns the element at a specified position (index).If no index is specified, it removes and returns the last element.

  Syntax: list.pop(index)

Indexing

- Indexing in Python refers to accessing individual elements within a sequence, such as a list, tuple, string, or other iterable objects.
- Each element in a sequence is assigned a numerical index, starting from 0 for the first element and increasing by 1 for each subsequent element.

Key Points:

Positive Indexing:

- The first element has an index of 0.
- The second element has an index of 1, and soon.

Negative Indexing:

- Allows you to access elements from the end of the sequence.
- The last element has an index of -1, the second last is -2, and soon.

Indexing in Strings:

Indexing works similarly with strings, where each character has an index.

Out-of-Range Index:

Accessing an index that is beyond the length of the sequence raises an Index Error.

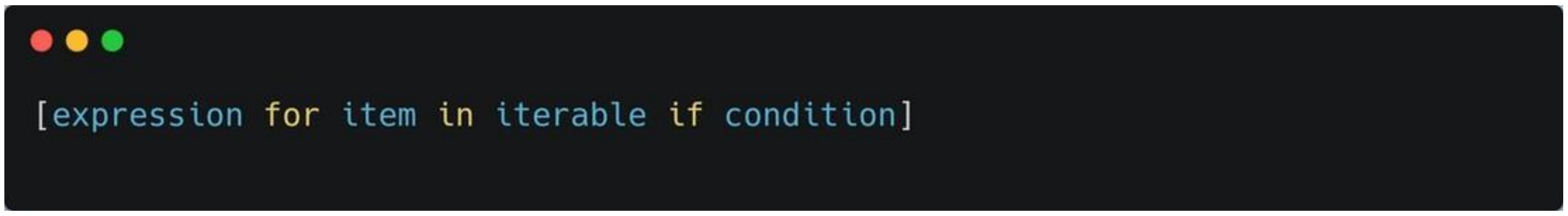
Slicing

- Slicing in Python is a technique used to access a subset of elements from sequences like lists, tuples, or strings.
- It allows you to retrieve a portion of the sequence by specifying a start, stop, and optional step index.
  - **start:** The index where the slice begins (inclusive). If omitted, it defaults to the beginning of the sequence (0).
  - **stop:** The index where the slice ends (exclusive). If omitted, it defaults to the end of the sequence.
  - **step:** The step size or interval between elements in the slice. If omitted, it defaults to 1.

List Comprehension

- List comprehension is a concise and powerful feature in Python that allows you to create lists in a single line of code.
- It combines the process of creating and populating a list with an expression and, optionally, one or more loops and conditions.

Syntax:



```
[expression for item in iterable if condition]
```

expression: The value or operation applied to each item.
item: The variable representing the current element in the iteration.
iterable: The collection (like a list, tuple, or range) that you are iterating over.
condition: (Optional) A filter that decides whether the expression should be applied to the current item.

Benefits of List Comprehension:

- Concise: It reduces the lines of code needed to create and populate lists.
- Readable: Once you get familiar with the syntax, it can be easier to read and understand.
- Efficient: It often runs faster than traditional for-loop approaches due to its optimized implementation.

Tuples and their Immutabilty

Tuples in Python are ordered collections of items, similar to lists, but with one key difference: tuples are immutable. This means that once a tuple is created, its elements cannot be modified, added, or removed. This immutability makes tuples useful for representing fixed data that should not change through out the program.

Key Features of Tuples:

- Ordered: Like lists, tuples maintain the order of elements, and you can access elements by their index.
- Immutable: Once a tuple is created, you cannot change its content. This includes:
- Modifying elements: You cannot change the value of any item in the tuple.
- Adding elements: You cannot append or insert new items.

- Removing elements: You cannot remove items from a tuple.

Defined with Parentheses: Tuples are created by placing a sequence of values separated by commas inside parentheses (()).

Tuples Packing and Unpacking

Tuple packing and unpacking are two related concepts in Python that make working with tuples more convenient and intuitive.

Tuple Packing:

- Tuple packing refers to the process of assigning multiple values to a single tuple variable.
- When you place multiple values separated by commas into a variable, Python automatically packs them into a tuple.

Tuple Unpacking:

- Tuple unpacking is the reverse process, where the values in a tuple are extracted and assigned to individual variables.
- The number of variables on the left must match the number of elements in the tuple.

Dictionaries and Sets

Dictionaries and sets are two important data structures in Python, each serving different purposes and offering unique functionalities.

Here's a basic overview of each:.

Dictionaries

- A dictionary in Python is a collection of key-value pairs where each key is unique and is used to retrieve the corresponding value.
- Dictionaries are unordered collections and are mutable, meaning you can change their content after creation.

Sets

- A set is an unordered collection of unique elements.
- Sets are useful for membership testing and removing duplicates from a collection.

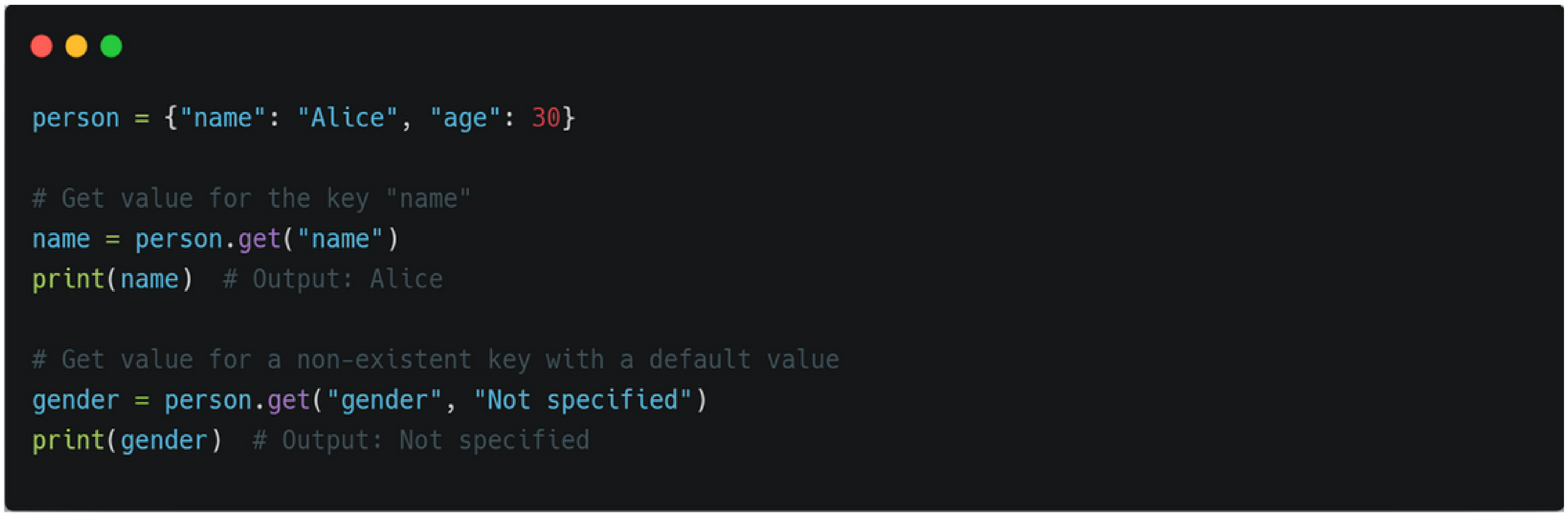
Dictionaries Methods

Dictionaries in Python come with several built-in methods that help you manage and manipulate key-value pairs efficiently.

dict.get()

The get() method retrieves the value for a specified key. If the key is not found, it returns None or a specified default value.

examples:

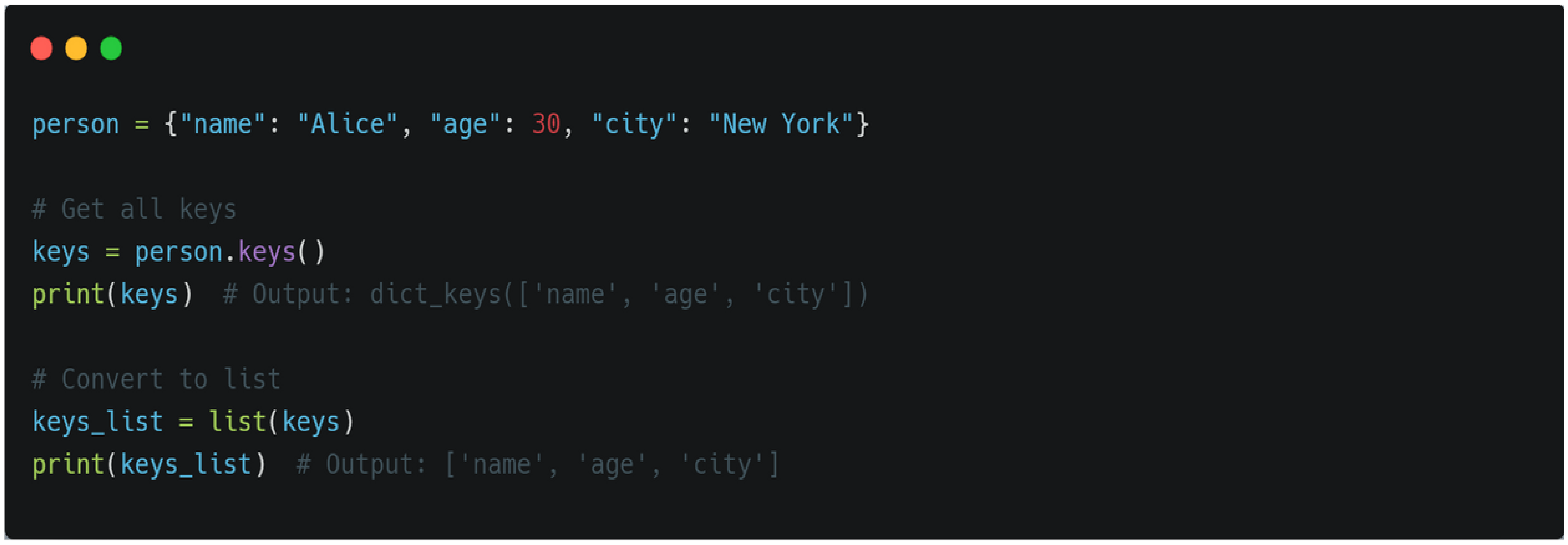


```python
person = {"name": "Alice", "age": 30}

# Get value for the key "name"
name = person.get("name")
print(name)  # Output: Alice

# Get value for a non-existent key with a default value
gender = person.get("gender", "Not specified")
print(gender)  # Output: Not specified
```

dict.keys()

The keys() method returns a view object that displays a list of all the keys in the dictionary.

examples

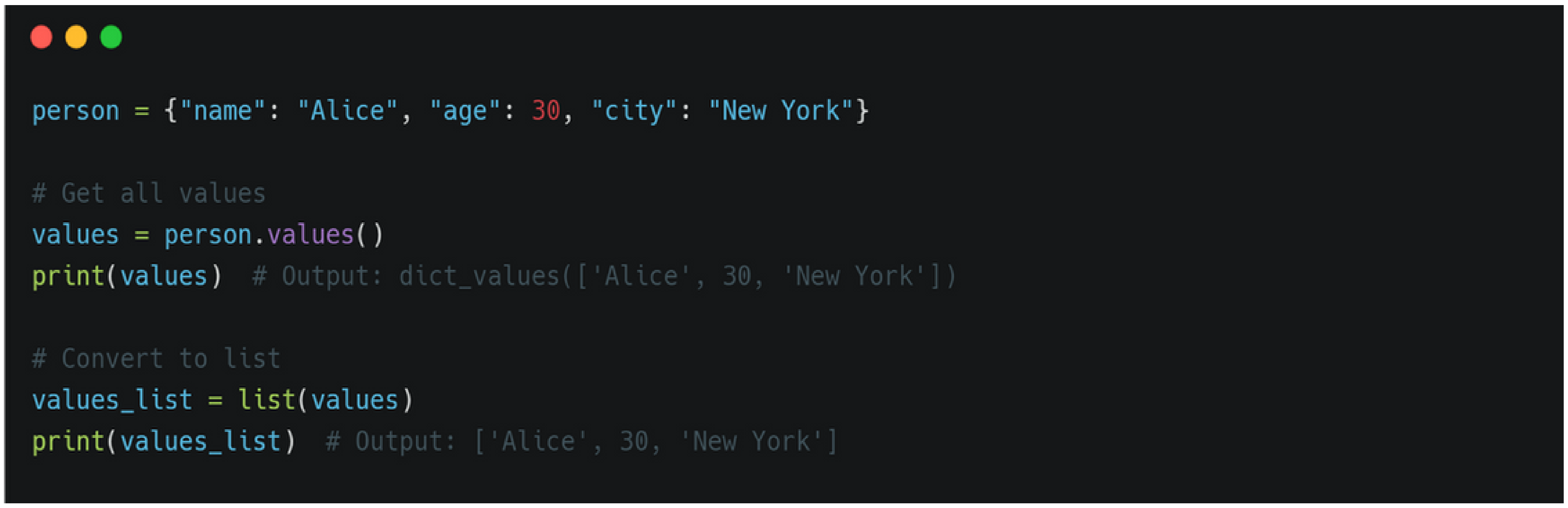


```python
person = {"name": "Alice", "age": 30, "city": "New York"}

# Get all keys
keys = person.keys()
print(keys)  # Output: dict_keys(['name', 'age', 'city'])

# Convert to list
keys_list = list(keys)
print(keys_list)  # Output: ['name', 'age', 'city']
```

dict.values()

The values() method returns a view object that displays a list of all the values in the dictionary.

examples:

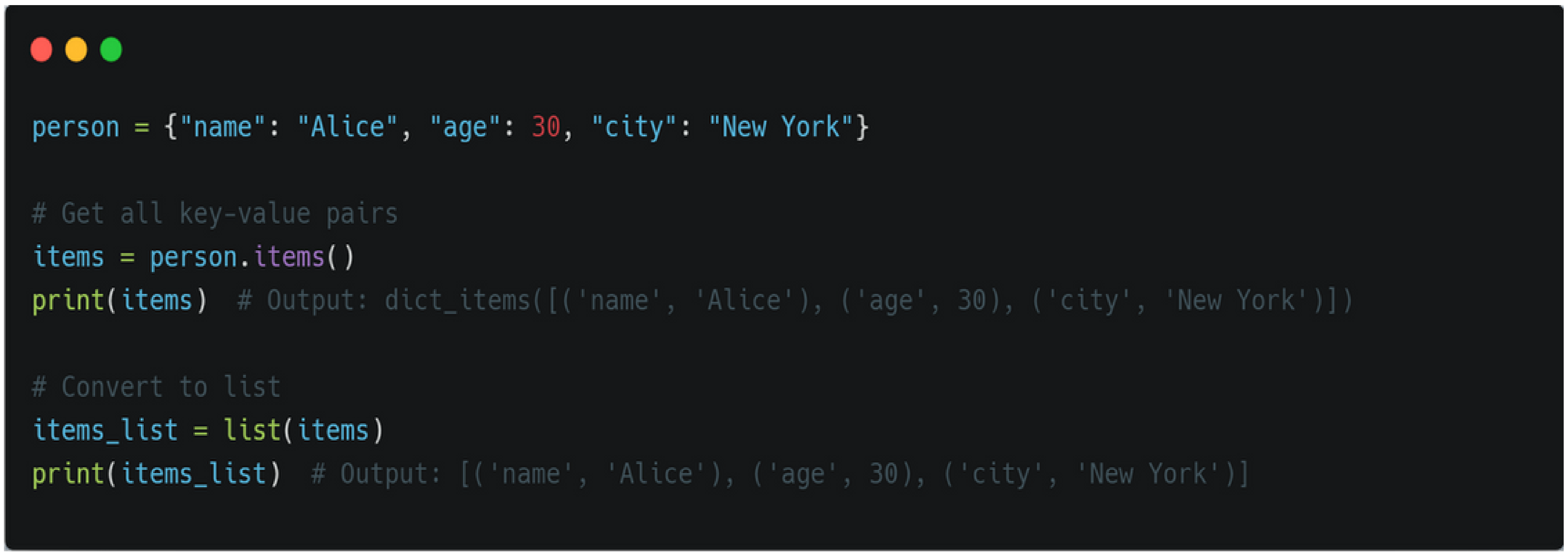


```
person = {"name": "Alice", "age": 30, "city": "New York"}

# Get all values
values = person.values()
print(values)  # Output: dict_values(['Alice', 30, 'New York'])

# Convert to list
values_list = list(values)
print(values_list)  # Output: ['Alice', 30, 'New York']
```

dict.items()

The items() method returns a view object that displays a list of tuples, where each tuple is a key-value pair.

examples:

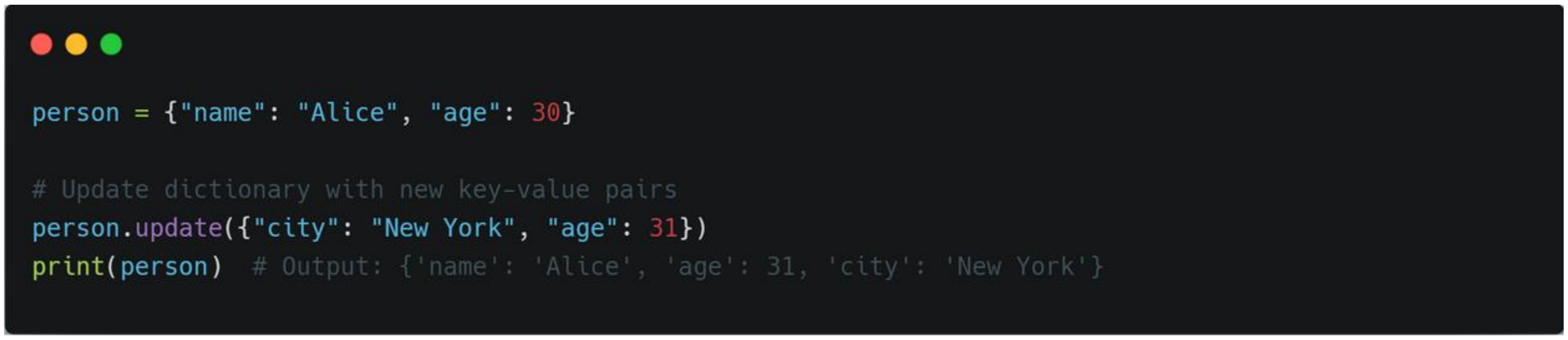


```
person = {"name": "Alice", "age": 30, "city": "New York"}

# Get all key-value pairs
items = person.items()
print(items)  # Output: dict_items([('name', 'Alice'), ('age', 30), ('city', 'New York')])

# Convert to list
items_list = list(items)
print(items_list)  # Output: [('name', 'Alice'), ('age', 30), ('city', 'New York')]
```

dict.update()

The update() method updates the dictionary with elements from another dictionary or iterable of key-value pairs. If a key already exists, its value will be updated.

examples:

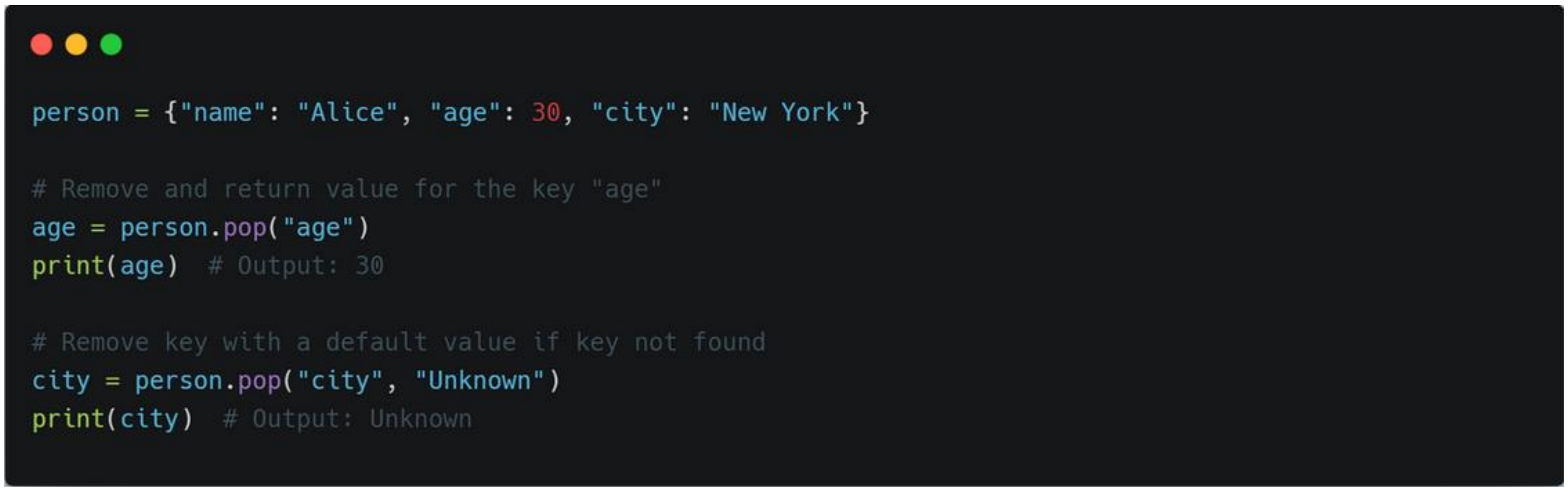


```python
person = {"name": "Alice", "age": 30}

# Update dictionary with new key-value pairs
person.update({"city": "New York", "age": 31})
print(person)  # Output: {'name': 'Alice', 'age': 31, 'city': 'New York'}
```

dict.pop()

The pop() method removes a specified key and returns its corresponding value. If the key is not found, it raises a Key Error, unless a default value is provided.

Examples:

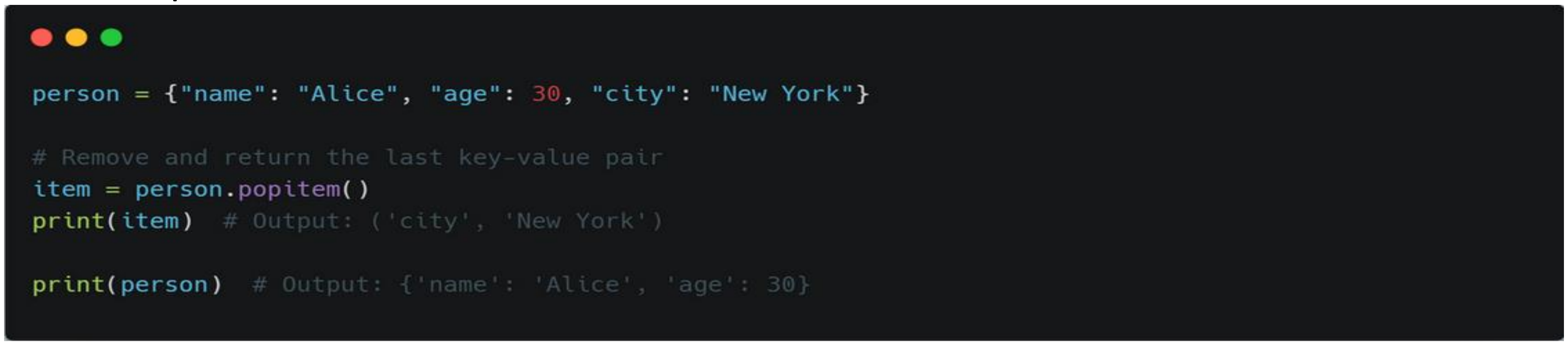


```python
person = {"name": "Alice", "age": 30, "city": "New York"}

# Remove and return value for the key "age"
age = person.pop("age")
print(age)  # Output: 30

# Remove key with a default value if key not found
city = person.pop("city", "Unknown")
print(city)  # Output: Unknown
```

dict.popitem()

The pop item() method removes and returns a (key, value) pair as a tuple. It removes the last item added to the dictionary (in Python 3.7 and later, dictionaries maintain insertion order).

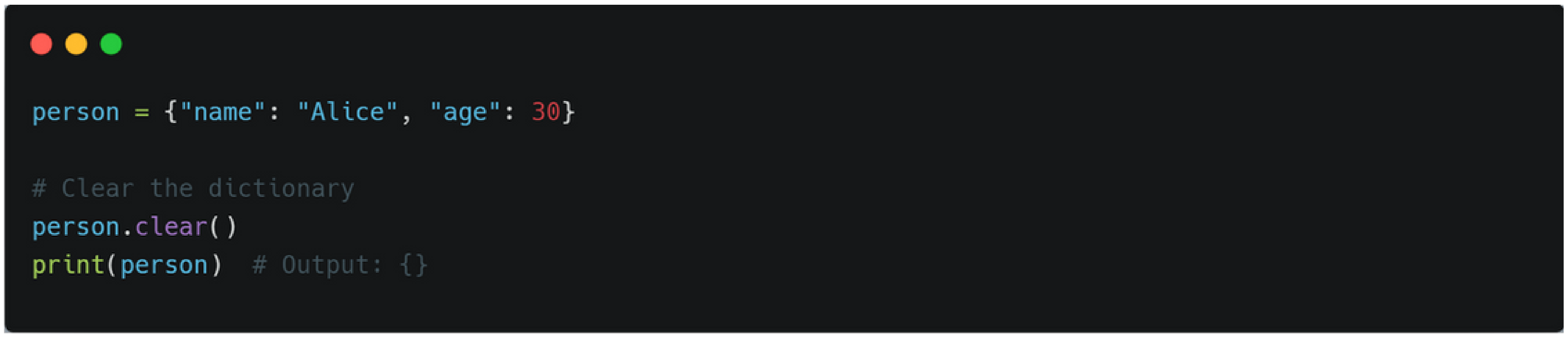
examples:



```python
person = {"name": "Alice", "age": 30, "city": "New York"}

# Remove and return the last key-value pair
item = person.popitem()
print(item)  # Output: ('city', 'New York')

print(person)  # Output: {'name': 'Alice', 'age': 30}
```

dict.clear()

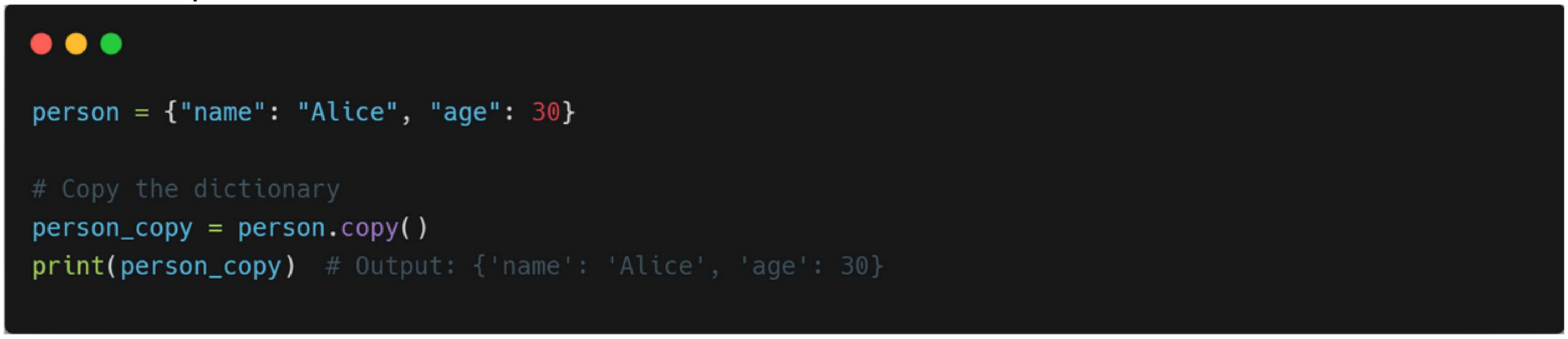
The clear() method removes all items from the dictionary, leaving it empty.
examples:



```python
person = {"name": "Alice", "age": 30}

# Clear the dictionary
person.clear()
print(person)  # Output: {}
```

dict.copy()

The copy() method returns a shallow copy of the dictionary.
examples:



```python
person = {"name": "Alice", "age": 30}

# Copy the dictionary
person_copy = person.copy()
print(person_copy)  # Output: {'name': 'Alice', 'age': 30}
```