Python Fundamentals: By King BABA

Python has emerged as a highly popular programming language, favored for its versatility and its design that prioritizes readability, making it an excellent choice for individuals venturing into the world of programming for the first time. The extensive and supportive community surrounding Python further contributes to its appeal for new learners. A strong grasp of the fundamental concepts of Python is essential as it lays the groundwork for developing more intricate and sophisticated programs in the future. This report aims to provide a comprehensive explanation of these foundational elements, suitable for those with no prior programming experience.

1. Understanding Python Data Types and Structures:

In Python, the concept of a data type serves to classify the kind of values that a variable can hold. This classification is crucial as it dictates how the data is stored within the computer's memory and what types of operations can be performed on that data.

It is important to recognize that in Python, everything is treated as an object, and these data types are essentially classes that define the properties and behaviors of these objects.

1.1 Integers

Integers in Python represent whole numbers, which can be either positive or negative and do not include a decimal point. A common example of using an integer would be to store someone's age, such as age = 30. When an integer is created in Python, the language employs dynamic memory allocation

> This means that the amount of memory reserved for an integer can automatically increase as the size of the number grows, offering a level of flexibility that eliminates concerns about integer overflow that are present in some other programming languages with fixed-size integer types.

For smaller integers, specifically those within the range of -5 to 256, python often utilizes a caching mechanism for optimization. This means that these frequently used small integer values are pre-allocated in memory and reused whenever they are needed, which can lead to improved performance.

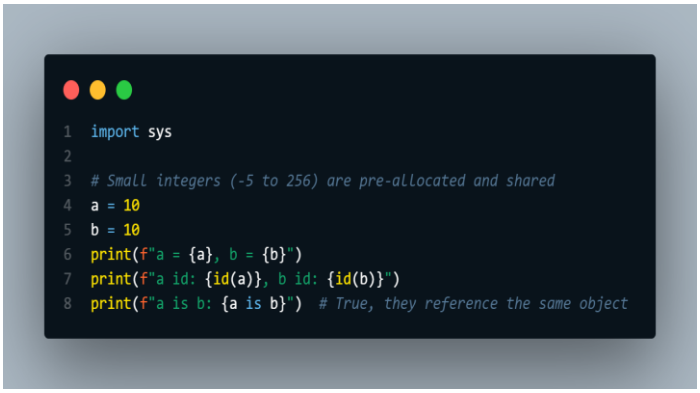
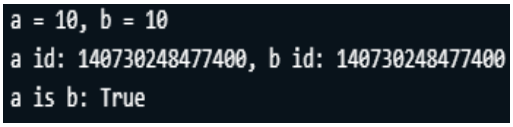
suppose when you create a variable and assign it an integer (like x = 10), your computer doesn't store 10 directly inside x. Instead:

1. **The number 10 is an object stored somewhere in memory (one of the boxes in the warehouse)**.
2. **The variable x is just a label (or a pointer) that tells your program where to find 10 in memory.**

So, when you use x, Python looks at its reference (or pointer) and retrieves value 10 from that specific memory location.

This system helps Python manage memory efficiently, especially when multiple variables refer to the same number.

Integers in Python are considered **immutable.** This means that once an integer object is created, its value cannot be changed directly. When an operation that appears to modify an integer is performed, such as incrementing its value, Python creates a brand-new integer object with the updated value, and the variable is then reassigned to refer to this new object. The original integer object remains unchanged.

Input	Output
	

1.2 Floating Point Numbers (Float):

Floating-point numbers, or oats, in Python are used to represent real numbers, which are numbers that have a decimal point. For example, the mathematical constant pi can be stored as a float: pi = 3.14159.
In terms of memory storage, Python follows a system called **IEEE 754 double-precision format** to store floats. This means:

Each float takes up 64 bits in memory (which is 8 bytes).
These 64 bits are divided into **three parts**:

1. **Sign bit** (1 bit) ⟶ Tells if the number is positive or negative.
2. **Exponent** (11 bits) ⟶ Determines how big or small the number is.
3. **Mantissa (or fraction)** (52 bits) ⟶ Stores the actual digits of the number.

This allows Python to represent **very small** numbers (like 0.0000000001) and **very large** numbers (like 1,000,000,000,000) while keeping the decimal precision.

Since floats require more bits to store details like the decimal part and exponent, they generally **use more memory** than integers.

A key characteristic to be aware of is that due to the way floating-point numbers are represented in computer memory with a fixed number of bits, they have a limited precision.

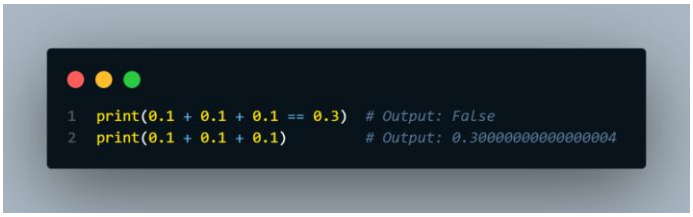
Computers store numbers in **binary (0s and 1s)**, and floats are stored using **64 bits** following the **IEEE 754 standard**. Because the number of bits is **fixed**, it cannot perfectly represent every possible decimal number.

For example:

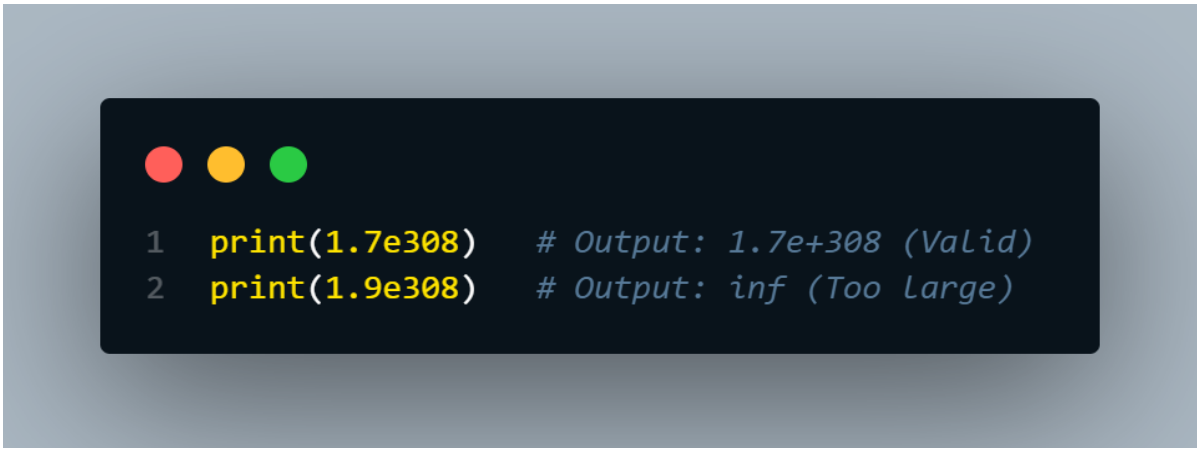
- The decimal number **0.1** cannot be exactly stored in binary.
- Instead, Python stores a very close approximation, which might be **0.1000000000000000555** in memory.

This tiny difference can cause **rounding errors** in calculations.

Example:

Input	**Output**
	

Python's oat type can also represent special values such as infinity, denoted as inf, which can result from operations like dividing by zero with oats, and "Not a Number", denoted as NaN, which can occur in cases of undefined mathematical operations. The approximate maximum value that a floating-point number can represent in Python is around **$1.7976931348623157 \times 10^{308}$**



Like integers, floating-point numbers in Python are **immutable**. When an operation that appears to modify a float is performed, such as adding to it, a new float object with the updated value is created in memory, and the variable then references this new object. The original float object remains unchanged. This behavior can be confirmed by observing that the memory address of the variable changes are the operation.

The float() function can be used to explicitly convert other data types, such as integers or Booleans, into floating-point numbers.

1.3 Strings (str):

Strings in Python are used to represent text and are defined as ordered sequences of characters. These sequences are typically enclosed within either single quotes (') or double quotes ("). For instance, a name can be stored as a string: name = "Alice".
When a string variable is declared, the string object is stored in a specific location in memory managed by Python's memory manager. For strings smaller than 512 bytes, Python's **pymalloc** allocator is often used, which is optimized for handling small objects efficiently. Larger strings might be managed by the standard **malloc** function from the C library.

Pymalloc: -

- **Definition**: pymalloc is Python's specialized memory allocator, optimized for managing small memory allocations (typically less than 512 bytes).
- **Purpose**: It was designed to improve the performance of frequent memory allocations and deallocations, which are common in Python programs.

How it Works:

- pymalloc allocates memory in **pools** of fixed size (e.g., 512 bytes) for small objects like integers, floats, and short strings.
- It divides these pools into smaller **blocks** to reduce fragmentation.
- It is faster than the standard malloc because it avoids system calls for each small memory request.

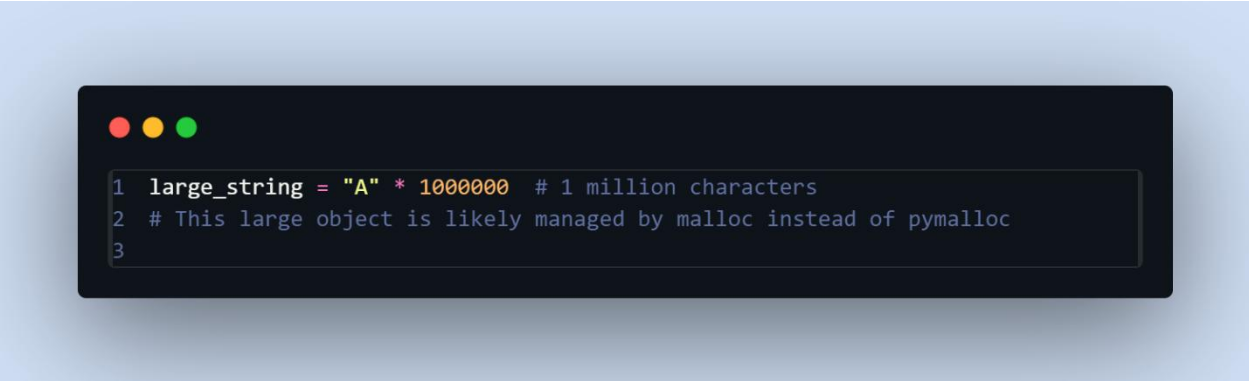
Malloc: -

- Definition: malloc (memory allocation) is a standard function from the C standard library used to allocate memory dynamically at runtime. It is commonly used in lower-level programming languages like **C**, but Python internally leverages malloc for managing large memory blocks.

- Purpose: malloc directly interacts with the **operating system** to request free memory on the **heap**. It is primarily used when **dynamic memory allocation** is required for large objects or data structures.

How it works:

- It allocates a **contiguous block of memory** on the **heap** and returns a **pointer** to the beginning of the block.
- The allocated memory is **uninitialized**, meaning it may contain garbage values until explicitly assigned.
- In **Python**, larger objects such as large **strings**, **lists**, or **dictionaries** may bypass **pymalloc** and use **malloc** due to **size limitations**.

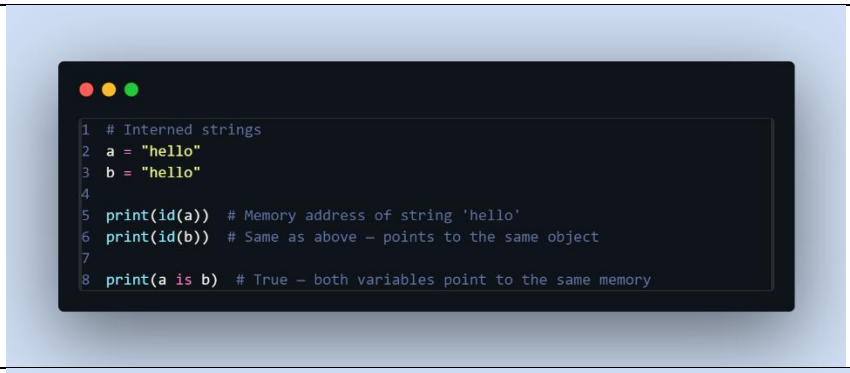
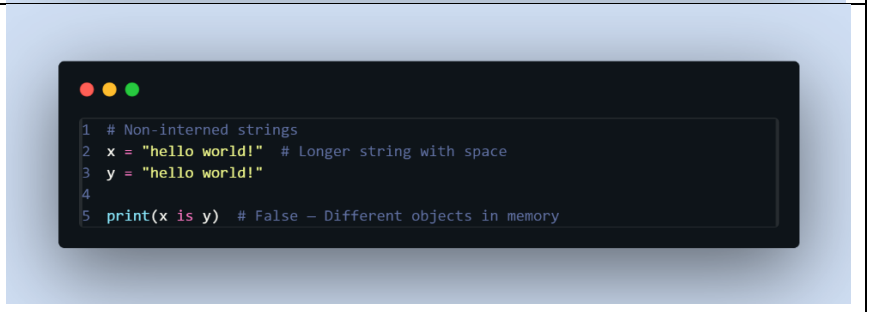


```
large_string = "A" * 1000000  # 1 million characters
# This large object is likely managed by malloc instead of pymalloc

```

Feature	pymalloc	malloc
Purpose	Optimized for **small objects** (<512 bytes)	Used for **large objects** and general-purpose memory allocation
Speed	Faster due to minimal system calls	Slower because it requires system calls
Memory Pooling	Uses pre-allocated **memory pools**	Allocates memory on the **heap**
Usage in Python	Used for small strings, integers, and floats	Used for large strings, lists, and dictionaries

Before a string is stored in memory, Python checks if an identical string literal already exists. If it does, Python might use a memory optimization technique called string interning, where identical string literals share the same memory location. This helps to save memory and improve performance, especially in

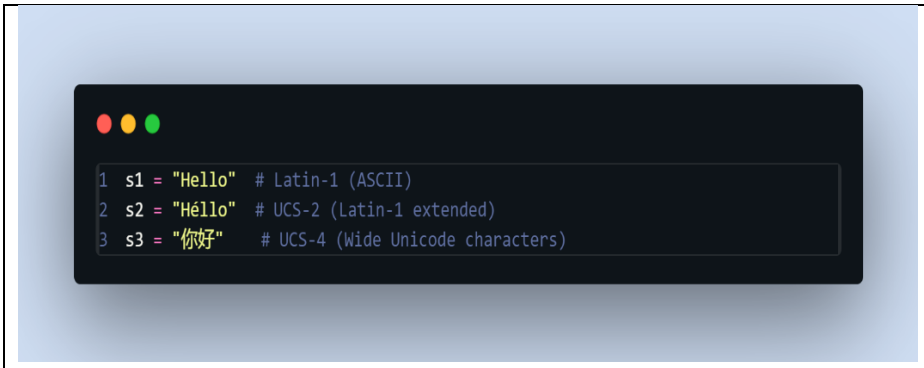
programs that use many literal repeated strings. This interning is often managed using a dictionary where unique string objects are keys, and their memory addresses are the values. Depending on the characters present in a string, Python might use different internal encodings, such as Latin-1 (1 byte per character), UCS-2 (2 bytes per character), or UCS-4 (4 bytes per character).

String Interning	**String interning** is a **memory optimization technique** used by Python where identical **immutable strings** share the **same memory location** instead of creating separate copies. This helps to: • **Reduce memory usage** — No need to store multiple copies of the same string. • **Improve performance** — String comparisons become faster since only memory addresses are compared.
How String Interning Works:	Before storing a string in memory, **Python checks if an identical string already exists**. If it does, **Python reuses the existing string object** instead of creating a new one. Interning is commonly applied to: • **Short strings** (less than 20 characters). • **Alphanumeric strings** (without special characters). • **Compile-time constants** (e.g., literal strings in code).
Explanation: Python automatically interns short, simple strings like "hello". Both a and b point to the **same memory location**, saving space	
When String Interning May Not Happen Automatically: Explanation: Python does not automatically intern longer strings or strings with special characters. However, you can manually force interning using sys.intern().	

✅ Memory Encoding of Strings:

Python uses **different internal encodings** to store strings based on the characters present:

Encoding	Bytes Per Character	When Used
Latin-1	1 byte	For ASCII characters (0-255)
UCS-2	2 bytes	For Unicode strings with no wide characters
UCS-4	4 bytes	For Unicode strings with wide characters



```
1 s1 = "Hello"  # Latin-1 (ASCII)
2 s2 = "Héllo"  # UCS-2 (Latin-1 extended)
3 s3 = "你好"    # UCS-4 (Wide Unicode characters)
```

Explanation:
Python dynamically chooses the most memory-efficient encoding for strings.

Latin-1 for simple English text.

UCS-2 or **UCS-4** for Unicode or wide characters.

In Python, a **string object** is more than just the sequence of characters you see. Internally, it has a **metadata header** followed by the **actual character data**. This design helps Python efficiently manage strings by keeping track of essential information.

1. **Metadata Header:**
   - Contains important information about the string.
   - Typically, **16 to 32 bytes** in size, depending on the system architecture (32-bit vs. 64-bit) and the Python implementation.
2. **Actual Character Data:**
   - This is where the characters of the string are stored.
   - The data is encoded using **Latin-1**, **UCS-2**, or **UCS-4**, depending on the characters.

Important Information Stored in the String Metadata Header:

Field	Description	Why It's Important	Example
Reference Count	Number of references (pointers) to the string object.	Helps in **garbage collection**; objects are deleted when count drops to **0**.	sys.getrefcount("Hello")
Object Type	Identifies that the object is a **string** (Unicode).	Help the interpreter handle data appropriately.	type("Hello") # str
Length	Total number of characters in the string.	**O(1)** length lookup instead of scanning the string.	len("Hello") # 5

Hash Value	Cached hash value of the string (used for hashing).	Speeds up **dictionary lookups** (no need to recompute).	hash("Hello")
Encoding Format	Specifies how characters are stored (Latin-1, UCS-2, or UCS-4).	Optimizes memory usage based on character type.	Latin-1 for ASCII, UCS-4 for Unicode.
Interning Flag	Indicates whether the string is **interned** (shared) or not.	Avoids duplication of identical strings.	"a" is "a" (True)
Immutable Flag	Confirm that the string is **immutable** (cannot be changed).	Ensures string safety; allows memory optimization.	Strings cannot be modified after creation.

Strings in Python are immutable 4. This means that once a string object is created, its contents cannot be changed directly. Any operation that appears to modify a string, such as concatenation or replacement, results in the creation of a new string object in memory. Attempting to modify a string in place by assigning a new character at a specific index will result in a TypeError.

1.4 Booleans (bool):

Booleans in Python represent truth values and can only be one of two possible states: True or False. These values are fundamental for controlling the flow of execution in programs, particularly in conditional statements. In terms of how they are stored in memory, Python Booleans are represented as integers under the hood, where True is equivalent to 1 and False is equivalent to 0. Despite their underlying numerical representation, Booleans are treated as distinct objects in Python and have their own unique memory addresses.

It is worth noting that in CPython, the standard implementation of Python, the bool type is a subclass of the int type.
This means that **True** and **False** are not separate fundamental types; they are essentially **integers** with specific values:

- True is equivalent to the integer **1**.
- False is equivalent to the integer **0**.

This explains why Booleans can sometimes be used in arithmetic operations, although it is generally considered better practice to use them primarily for their intended purpose of representing truth values to maintain code clarity.

```
# Arithmetic with booleans
print(True + True)          # Output: 2 (1 + 1)
print(True * 10)            # Output: 10 (1 * 10)
print(False + 5)            # Output: 5 (0 + 5)
print(True - False)         # Output: 1 (1 - 0)
```

Memory Size and Address:

- **Memory Size:**
  - The size of a boolean object can be inspected using **sys.getsizeof()**.
  - In CPython (64-bit), a boolean takes **28 bytes** of memory — the same as a small integer.
- **Memory Address:**
  - **id()** can be used to check the **memory address** of a boolean object.
  - Python pre-allocates True and False as **singletons**, meaning all instances of True or False point to the same object.

A **singleton** is a **design pattern** where **only one instance** of a particular object is created. A **singleton** means **only one object** is created, and **everyone uses the same object**. No matter how many times you create or assign it, **it always points to the same memory location**

In Python, certain immutable objects, including **True** and **False**, are **pre-allocated singletons**, meaning:

- **Only one copy** of True and False exists in memory.
- Every time you use True or False, Python does **not** create a new object; instead, it **reuses the existing one**.

Think of it like a **single water bottle** in a room:

- No matter how many people ask for water, they all share **the same bottle**.
- If one person drinks from it, it affects everyone because it's **the same bottle**

In Python, True and False are singletons.

- **Python creates them once**, and every time you use True or False, it **reuses the same object**.

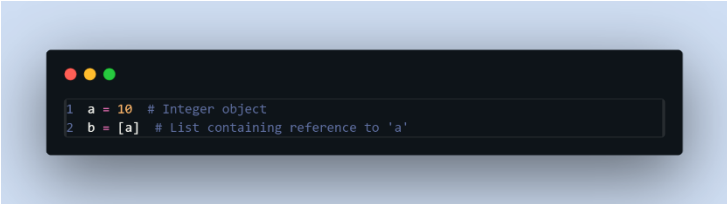
```python
# Non-singleton example (List)
x = [1, 2, 3]
y = [1, 2, 3]

# Singleton example (Boolean)
a = True
b = True

# Check memory addresses
print(id(x), id(y))   # Different addresses
print(id(a), id(b))   # Same address
```

Booleans in Python are **immutable**. Once a Boolean object (True or False) is created, its value cannot be changed. When a variable assigned to a Boolean value appears to change, it is being reassigned to a new Boolean object with a different identity in memory. The original Boolean object remains unchanged.

1.6 List:

Lists in Python are ordered collections of items. They are highly versatile as they can contain duplicate items and elements of different data types within the same list. Lists are created using square brackets '[ ]',with individual items separated by commas.

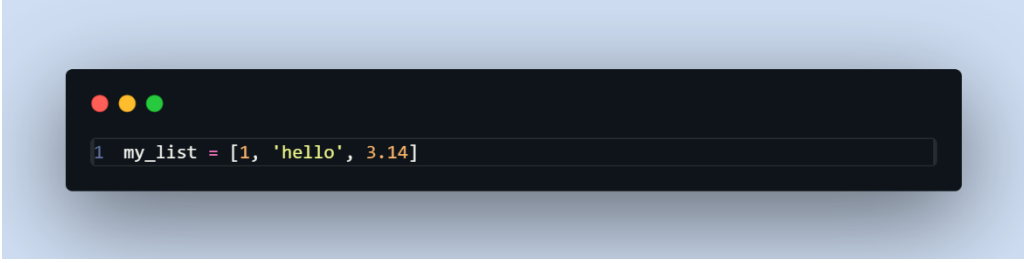
A **pointer** (or **reference**) is essentially a memory address that tells the computer where an object is stored. In Python, **everything** is an object, including integers, strings, and lists themselves. When you put objects into a list, the list holds **references** to these objects, not the objects themselves.



Here, B doesn't hold the number 10. directly. It holds a **reference** to the object 10 that exists somewhere in memory.

In Python, **lists** are implemented as **dynamic arrays of pointers (or references)** to objects. This means:
List Structure:

- A list is a container that stores references (or memory addresses) to **actual objects**.
- The **list itself** doesn't hold the data directly; it just keeps **references** (like pointers) to where the data is stored in memory.



In this case, my_list holds references to three items: an integer (1), a string ('hello'), and a float (3.14).
The actual data (1, 'hello', and 3.14) are stored elsewhere in memory.
So, my_list doesn't contain the **values** directly but rather **points to** the locations where these values are stored.
Due to their mutable nature, lists employ a strategy called over-allocation

Over-allocation refers to a strategy where Python allocates **extra memory** when creating or extending a list.
When you add elements to a list, Python doesn't just allocate memory for the exact number of elements that are currently in the list. Instead, it allocates **extra space** to allow for future elements, reducing the need to frequently resize the list when new items are added.

Imagine a list starts with space for 4 elements but only has 2. When you add a 3rd and 4th item, it doesn't just allocate exactly 1 more space for the 5th item. It might allocate **space for 8 elements** instead, even if the list only contains 4.
This is why appending to a list doesn't always result in an immediate memory resize.

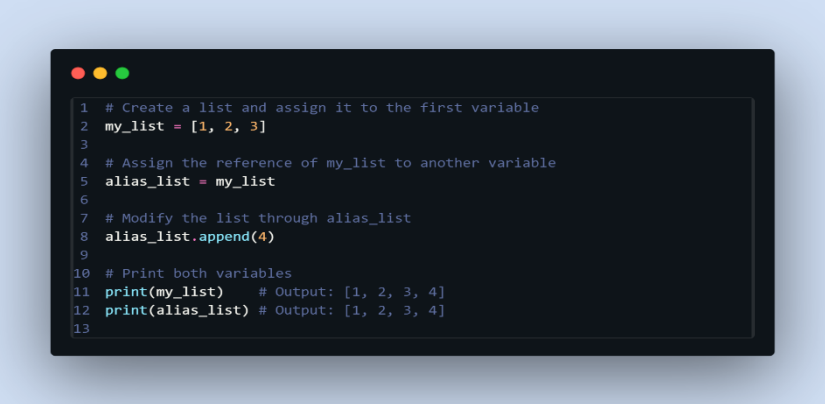
- **Performance Benefit:**
    - Over-allocation minimizes the number of times Python must resize the list when elements are added.
    - Without over-allocation, every time a list grows, Python would have to **reallocate memory** and **move all the items** to the new location. This is an expensive operation.
- **How it Works:**
    - When the list grows beyond the allocated space, Python **resizes** the list by allocating **a larger block of memory** than needed. The extra space is used to accommodate future additions.

When a list is created or extended, Python allocates more memory than is currently needed. This pre-allocation of extra space helps to minimize the frequency of memory resizing operations as new elements are added, which improves performance but can result in a slightly higher memory overhead compared to some other data structures like tuples.

It is important to note that the elements of a list are not necessarily stored contiguously in memory; the list object itself holds the contiguous references to these elements.
When we say that **the elements of a list are not necessarily stored contiguously in memory**, we mean:

- The **actual data** or **objects** that the list points to (the elements themselves) **do not have to be stored in adjacent memory locations**.
- The list only holds **references (pointers)** to these elements, and those references **could point to different locations** in memory.

Because lists store references to objects, it is possible for multiple variables to point to the same list in memory. When this happens, modifying the list through one variable will affect all other variables that are referencing the same list. This phenomenon is known as aliasing and is a direct consequence of the mutability of lists and the way variables store references to objects.



```python
# Create a list and assign it to the first variable
my_list = [1, 2, 3]

# Assign the reference of my_list to another variable
alias_list = my_list

# Modify the list through alias_list
alias_list.append(4)

# Print both variables
print(my_list)    # Output: [1, 2, 3, 4]
print(alias_list) # Output: [1, 2, 3, 4]
```

When we do my_list = [1, 2, 3], the variable my_list holds a **reference** to the list [1, 2, 3] in memory.
When we assign alias_list = my_list, the variable alias_list doesn't create a new list. Instead, it **points to the same list** in memory as my_list.
Therefore, when we **modify the list** through alias_list.append(4), we are actually modifying the **same list** in memory that my_list is referencing.
As a result, both my_list and alias_list show the same list [1, 2, 3, 4], because they are both referencing the same object.

Unexpected Side Effects:

- If you're not careful, aliasing can lead to **unexpected side effects** in your code. For instance, if you modify a list through one variable, you might unintentionally affect other variables that reference the same list.

Memory Efficiency:

- On the positive side, aliasing can save memory because you're not duplicating the list; you're just using references to the same object. This is a form of **memory sharing**.

Avoiding Unintended Modifications:

- To avoid aliasing when you don't want one variable's changes to affect another, you can create a **copy** of the list.

Lists in Python are mutable. This means that after a list is created, its elements can be changed, new elements can be added, and existing elements can be removed in place, without creating a new list object. Python provides several methods to modify lists, such as append() to add an element to the end, insert() to add an element at a specific position, and extend() to add multiple elements from another list or iterable to the end. Elements can be removed using remove() to remove the first occurrence of a specific value, pop() to remove an element at a given index (or the last element if no index is specified), and the del statement to delete an element at a specific index. Individual elements can be modified by accessing them using their index and assigning a new value.

1.7 Tuples:

Tuples in Python are another type of ordered collection that can hold multiple items, allowing duplicates and a mix of different data types, like lists 2. However, the key difference between lists and tuples is that tuples are immutable. Tuples are created using parentheses (), with elements separated by commas.

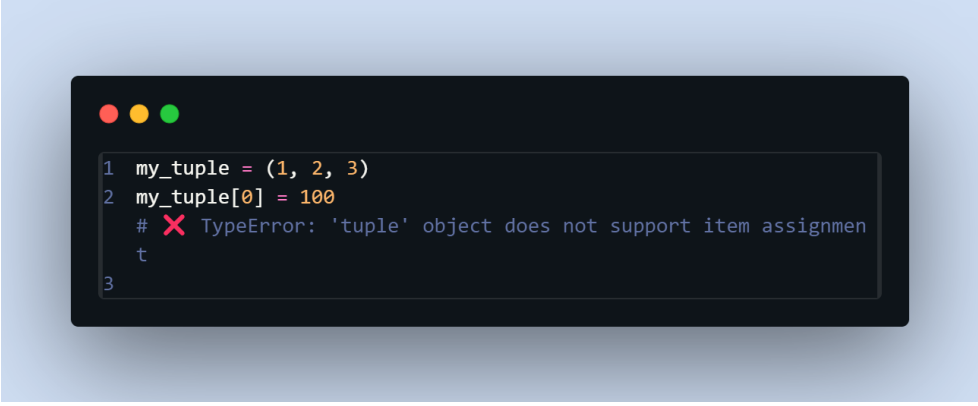
For example, coordinates = (10, 20) creates a tuple representing a point in a two-dimensional space. In terms of memory storage, tuples, like lists, store references to their elements. However, because tuples are immutable, their size is fixed at the time of creation.

This fixed size allows Python to allocate memory for tuples more efficiently, typically resulting in a lower memory overhead compared to lists, especially when dealing with a large amount of data that does not need to be modified. Unlike lists, tuples do not employ over-allocation as their size cannot change.

The immutability of tuples makes them well-suited for representing collections of data that should not be changed after they are created. This immutability also offers performance advantages in certain scenarios, as Python can perform optimizations knowing that the tuple's contents will remain constant.

While the tuple itself is immutable, it is important to note that if a tuple contains mutable objects, such as lists, the contents of those mutable objects can still be changed.

Tuples in Python are immutable. Once a tuple is created, you cannot add, remove, or change its elements. Attempting to do so will result in a TypeError. If you need to modify a tuple, you will typically create a new tuple with the desired changes.



```
my_tuple = (1, 2, 3)
my_tuple[0] = 100
# ❌ TypeError: 'tuple' object does not support item assignment
```

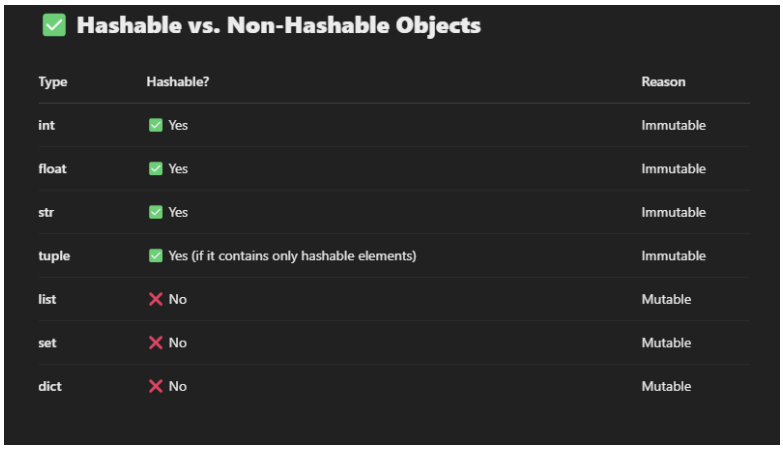
1.8 Sets:

A set is an unordered collection of unique elements in Python. Sets are mutable, meaning you can add or remove elements, but their elements must be immutable (e.g., numbers, strings, tuples).

Sets are useful for operations like removing duplicates from a collection and performing mathematical set operations (like union, intersection, difference).

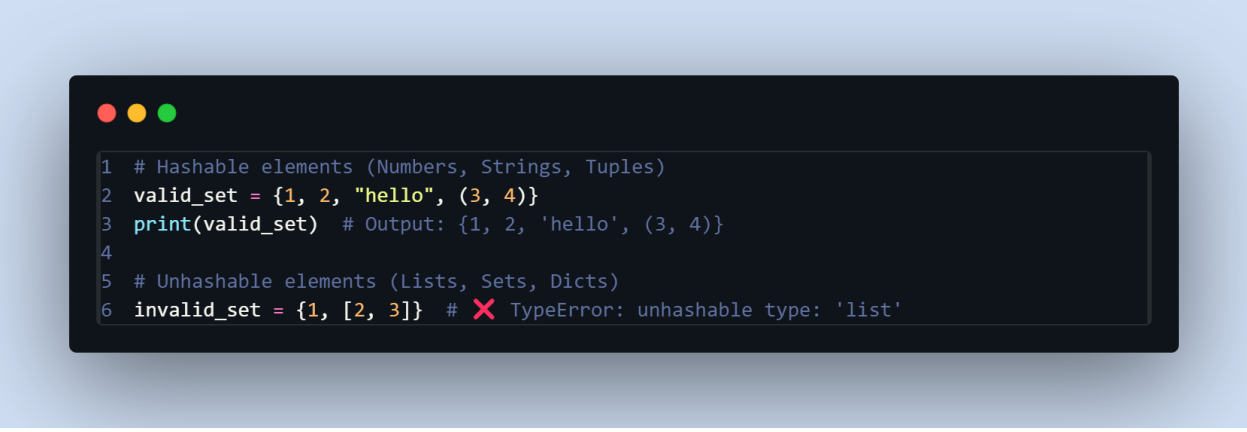
Elements within a set must be hashable, which generally includes immutable data types like numbers, strings, and tuples.

In Python, hashability means that an object has a unique identifier (a hash value) that never changes during its lifetime. This is crucial for storing elements in sets and dictionary keys because these data structures use hash tables for fast lookups.



✅ Hashable vs. Non-Hashable Objects

Type	Hashable?	Reason
int	✅ Yes	Immutable
float	✅ Yes	Immutable
str	✅ Yes	Immutable
tuple	✅ Yes (if it contains only hashable elements)	Immutable
list	❌ No	Mutable
set	❌ No	Mutable
dict	❌ No	Mutable



```python
# Hashable elements (Numbers, Strings, Tuples)
valid_set = {1, 2, "hello", (3, 4)}
print(valid_set)  # Output: {1, 2, 'hello', (3, 4)}

# Unhashable elements (Lists, Sets, Dicts)
invalid_set = {1, [2, 3]}  # ❌ TypeError: unhashable type: 'list'
```

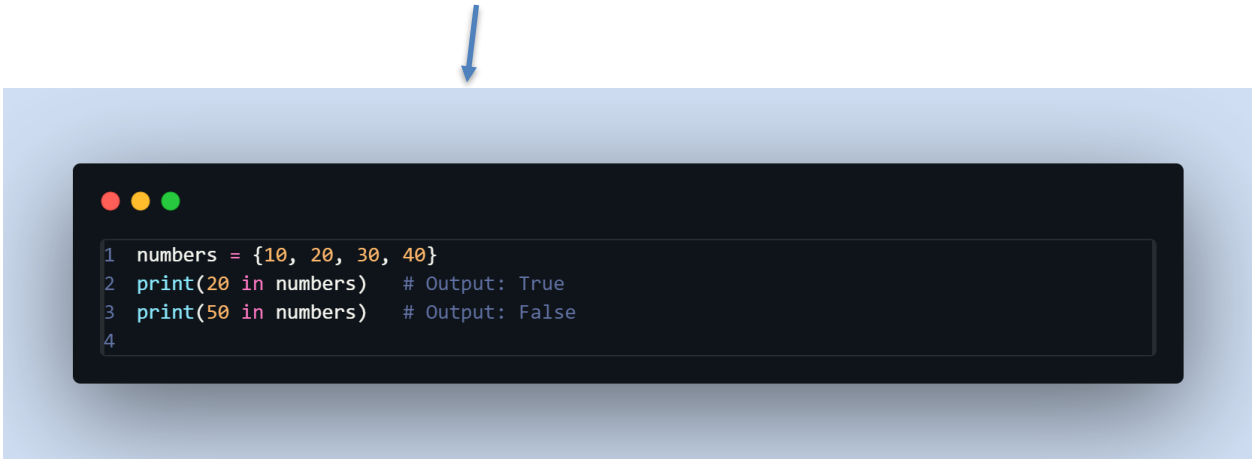
Lists cannot be added to a set because they can change (mutable), making their hash value unreliable.

A **set** in Python is implemented using a **hash table**, which means every element inside a set **must be hashable**.

- **Lists are mutable**, meaning their content can change after creation.
- If an object can change, its **hash value** could change too.
- Sets rely on hash values to store and retrieve elements efficiently.
- If an element's hash value changes after being added to a set, Python wouldn't be able to find it again, breaking the hash table's integrity.

Operation	Allowed?	Method
Add a new element	✅ Yes	s.add(x)
Remove an element	✅ Yes	s.remove(x), s.discard(x), s.pop()
Replace an element	❌ No	Must remove old and add new manually
Modify an element in place	❌ No	Not possible

You can use the built-in set() constructor with an iterable like a list or tuple. For example, unique_numbers = {1, 2, 3} creates a set of integers. Like dictionaries, sets in Python are typically implemented using hash tables. This allows for efficient membership testing and ensures the uniqueness of elements.



```python
numbers = {10, 20, 30, 40}
print(20 in numbers)   # Output: True
print(50 in numbers)   # Output: False
```

The fundamental property of sets is that they only store unique elements. If you try to add an element that is already present in the set, the set remains unchanged. This makes sets very useful for efficiently removing duplicate entries from a collection of items.

Sets in Python are mutable. You can add new elements to a set after it has been created using the add() method for a single element or the update() method to add multiple elements from another iterable. Elements can be removed using methods like remove(), which will raise a KeyError if the element is not found, discard(), which will remove the element if it is present but does not raise an error if it's not, and pop(), which removes and returns an arbitrary element from the set. The clear() method can be used to remove all elements from the set.

Useful methods include len() to find the number of elements in a set, copy() to create a shallow copy, isdisjoint() to check if two sets have no elements in common, issubset() to check if one set contains all elements of another set, and issuperset() to check if one set contains all elements of another set.

1.9 Frozenset:

A frozenset is an immutable version of a set. It has the same properties as a regular set, but once it's created, its elements cannot be changed (you cannot add, remove, or modify elements).

Feature	**Set**	**frozenset**
Mutable?	✅ Yes	❌ No
Ordered?	Yes	❌ No
Allows Duplicates?	❌ No	❌ No
Can be a Dictionary Key?	❌ No	✅ Yes
Supports Set Operations?	✅ Yes	✅ Yes (union, intersection, difference)

Data Type	**Mutable?**	**Ordered? (from 3.7+)**	**Allows Duplicates?**	**Key Storage Characteristics**
Integer	No	Yes	Yes	Dynamic memory allocation, caching
Float	No	Yes	Yes	64-bit double precision
String	No	Yes	Yes	Memory manager, interning
Boolean	No	Yes	Yes	Integer representation(0 or 1)
List	Yes	Yes	Yes	Dynamic array of references
Tuple	No	Yes	Yes	Fixed-size array of references
Dictionary	Yes	Yes	No (for keys)	Hash table
Set	Yes	Yes	No	Hash table

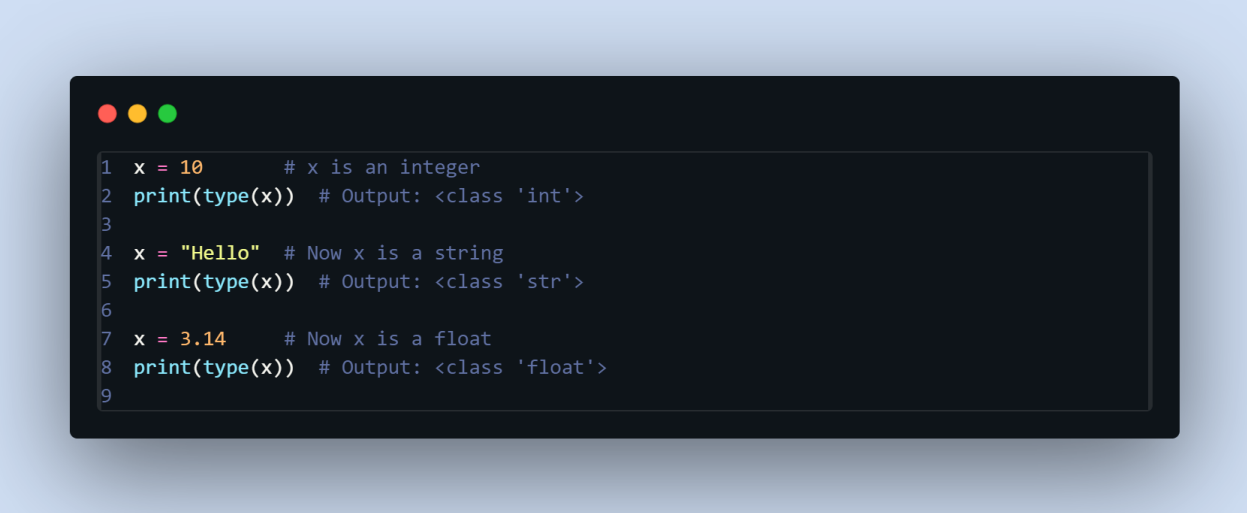
2. Variables and Assignment in Python:

In Python, variables serve as names that are used to store and refer to data within program. The fundamental operation for associating a value with a variable name is assignment, which is performed using the equals symbol = as the assignment operator. Unlike some other programming languages, Python does not require an explicit declaration of a variable before it is used. A variable is automatically created the first time a value is assigned to it. For example, the statement message = "Hello" creates a variable named message and assigns the string value "Hello" to it.

Python uses a **dynamic typing system**, meaning **you don't have to declare the type of variable explicitly**. The type of variable is determined **at runtime** based on the value assigned to it.

Key Characteristics of Dynamic Typing

1. **No Explicit Type Declaration**
   - You do not need to specify the type when creating a variable.
   - The interpreter assigns a type based on the assigned value.
2. **Variables Can Change Types**
   - A variable can hold values of different types at different times.
3. **Memory is Managed Automatically**
   - Python dynamically allocates and deallocates memory based on variable usage.



```python
x = 10       # x is an integer
print(type(x))  # Output: <class 'int'>

x = "Hello"  # Now x is a string
print(type(x))  # Output: <class 'str'>

x = 3.14     # Now x is a float
print(type(x))  # Output: <class 'float'>

```

Here, the variable x **changes dynamically** depending on the value assigned.

Comparison: Dynamic Typing vs. Static Typing

Feature	Dynamic Typing (Python)	Static Typing (C, Java)
Type Declaration	Not required	Required
Type Checking	At runtime	At compile-time
Flexibility	High (variables can change types)	Low (fixed variable types)
Error Detection	Errors appear at runtime	Errors caught at compile-time
Example	x = "Hello"	String x = "Hello";

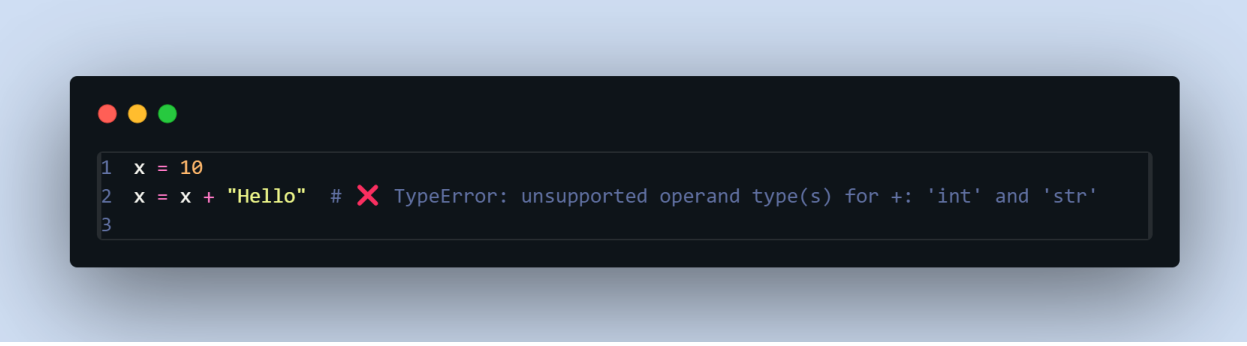
Advantages of Dynamic Typing

☑ **More Flexibility** – Can assign different types of values to the same variable.
☑ **Less Code** – No need for explicit type declarations.
☑ **Easier Prototyping** – Useful for rapid development.

Disadvantages of Dynamic Typing

❌ **Runtime Errors** – Type-related errors only appear when the code runs.
❌ **Performance Overhead** – Extra processing is required to determine variable types.
❌ **Less Readability** – Harder to understand expected data types in large projects.

Issue:



Since x was first an int, but later used in a string operation, this **causes a TypeError at runtime**.

This dynamic nature provides a great deal of flexibility to programmers. However, it also implies that type-related errors might not be detected until the program is running, so careful coding practices are important to mitigate this potential issue.

When a variable is assigned a value in Python, it does not directly store the value itself. Instead, variables hold references, or pointers, to objects that are stored in memory, specifically in the heap.

In Python, **variables are just references (pointers) to objects stored in memory**, rather than storing the actual values themselves. The memory allocation works as follows:

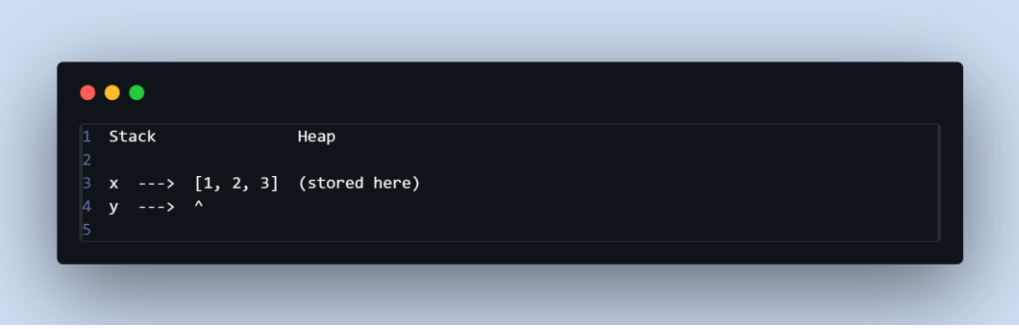
- **How Variables and Objects are Stored in Memory**
  1. **Variable Name (Reference)**
     - The variable itself is a small reference (pointer) stored in the **stack**.
     - This reference points to an object stored in the **heap**.
  2. **Object Storage**
     - The actual object (data) is stored in the **heap memory**, which is managed dynamically.
  3. **Memory Size for Variable Names**
     - Since a variable name is just a reference, its size is **small and fixed** (it doesn't depend on the size of the object it refers to).

```
x = [1, 2, 3]  # A list is created in memory
y = x  # y is another reference to the same list
```

Memory Breakdown

- The **variable x** is stored in the **stack**, pointing to an **object in the heap**.
- The **list [1, 2, 3]** is stored in the **heap**.
- The **variable y** now also points to the same list in the **heap** (not a copy).

Visual Representation



Since both x and y point to the same object, modifying the list via y will also affect x.

Understanding that variables are references is crucial for comprehending how assignment works, especially with mutable objects like lists and dictionaries. When you assign one variable to another (e.g., y = x), you are essentially making both variables refer to the same object in memory. If that object is mutable and is modified through one variable, the changes will be reflected when accessing it through the other variable as well. The id() function in Python can be used to illustrate this, as it returns the unique identity (which corresponds to the memory address in CPython) of an object. If two variables have the same id(), they are referring to the same object.

To use variables effectively, it is important to follow the rules for naming them. Variable names can only contain letters, digits, and the underscore character (_). They cannot start with a digit. Python is case-sensitive, so myVar and myvar are treated as different variables.

It is also essential to avoid using Python keywords (reserved words like if, else, for, etc.) as variable names. Following best practices for variable naming can greatly improve the readability and maintainability of code. This includes choosing meaningful names that clearly indicate the purpose of the variable (e.g., user_age instead of ua) and using conventions like snake case for multi-word variable names (e.g., number_of_students).

3. Operators and Expressions in Python:

In Python, operators are special symbols that perform operations on values, which are referred to as operands. An expression is a combination of operands (which can be variables, literals, or function calls) and operators that can be evaluated to produce a value. Python supports a wide range of operators for different purposes.

Arithmetic Operators

Operator	Name	Example	Result
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	7 * 6	42
/	Division	15 / 4	3.75
//	Floor Division	15 // 4	3
%	Modulus (Remainder)	15 % 4	3
**	Exponentiation	2 ** 3	8

Comparison Operators

Operator	Operator	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 6	True
>	Greater than	5 > 3	True
<	Less than	5 < 7	True
>=	Greater than or equal	5 >= 5	True
<=	Less than or equal	5 <= 4	False

Logical Operator

Operator	Name	Example	Result
and	Logical AND	True and False	False
or	Logical OR	True or False	True
not	Logical NOT	not True	False

Truth Table for Logical Operators

A	B	A AND B	A OR B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Assignment Operators

Operator	Name	Example	Equivalent To
=	Assign	x = 10	x = 10
+=	Add and Assign	x += 5	x = x + 5
-=	Subtract and Assign	x -= 3	x = x - 3
*=	Multiply and Assign	x *= 2	x = x * 2
/=	Divide and Assign	x /= 4	x = x / 4
//=	Floor Divide and Assign	x //= 2	x = x // 2
%=	Modulus and Assign	x %= 3	x = x % 3
**=	Exponentiate and Assign	x **= 2	x = x ** 2

Bitwise Operators

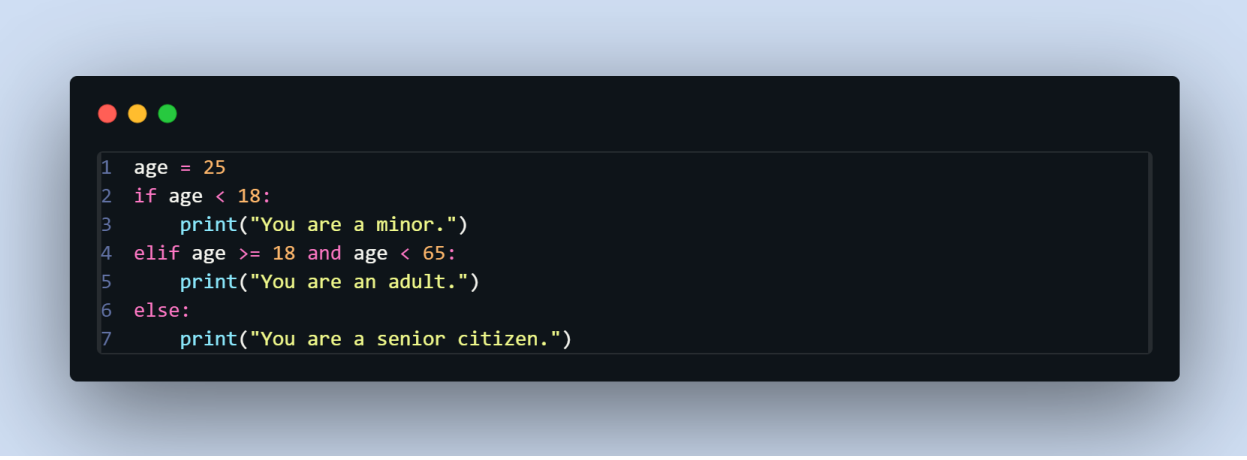
Operator	Name	Description	Example (a = 5, b = 3)	Binary Operation	Result (Decimal)
&	Bitwise AND	Returns 1 if both bits are 1, else 0	5 & 3	0101 & 0011 → 0001	1
|	Bitwise OR	Returns 1 if at least one bit is 1	5 | 3	0101 | 0011 → 0111	7
^	Bitwise XOR	Returns 1 if bits are different, else 0	5 ^ 3	0101 ^ 0011 → 0110	6
~	Bitwise NOT	Flips all bits (inverts 0 to 1 and vice versa)	~5	~0101 → 1010 (two's complement: -6)	-6
<<	Left Shift	Shifts bits left by n places (multiply by 2^n)	5 << 1	0101 << 1 → 1010	10
>>	Right Shift	Shifts bits right by n places (divide by 2^n)	5 >> 1	0101 >> 1 → 0010	2

When an expression contains multiple operators, the order in which they are evaluated is determined by operator precedence. For example, in the expression 3 + 4 * 2, the multiplication (*) has higher precedence than addition (+), so 4 * 2 is evaluated first, resulting in 8, and then 3 + 8 is evaluated to get 11. Parentheses can be used to override the default precedence; for example, in (3 + 4) * 2, the addition is performed first, resulting in 7, and then 7 * 2 is evaluated to get 14. When operators have the same precedence, their associativity determines the order of evaluation. Most arithmetic operators have left-to-right associativity, meaning that in an expression like 10 - 5 - 2, the subtraction is performed from left to right: (10 - 5) - 2, resulting in 3.

4. Controlling Program Flow with Conditional Statements and Loops:

Control flow statements are fundamental to programming as they allow you to dictate the order in which the code is executed. Python provides conditional statements to execute different blocks of code based on whether certain conditions are true or false, and looping constructs to repeat a block of code multiple times.

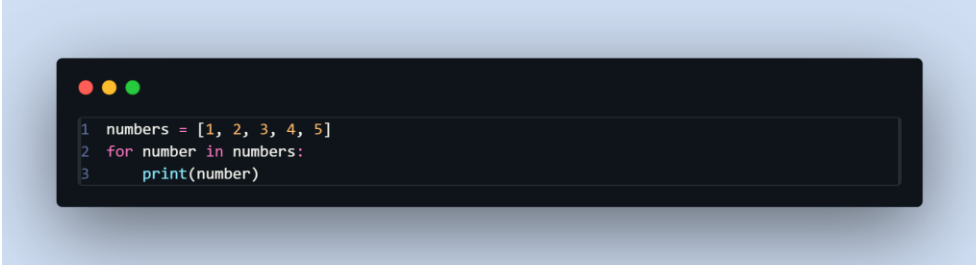
Conditional statements in Python are implemented using the if, elif (else if), and else keywords. The basic structure starts with an if statement, which consists of the if keyword followed by a condition (an expression that evaluates to a boolean value) and a colon. The block of code that should be executed if the condition is true is indented below the if statement. You can include one or more elif statements to check additional conditions if the initial if condition is false. Finally, you can have an optional else block, which contains code that will be executed if none of the preceding if or elif conditions were true. Here is a simple example:



```python
age = 25
if age < 18:
    print("You are a minor.")
elif age >= 18 and age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

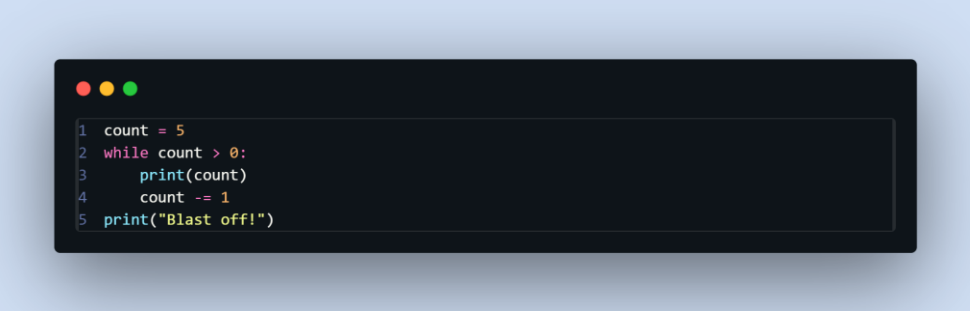
In this example, the program checks the value of the age variable and prints a different message based on its value. Boolean expressions are commonly used as conditions in if statements, allowing for complex decision-making logic.

Looping constructs in Python allow you to repeat a block of code if a certain condition is met or for each item in a sequence. Python provides two main types of loops: the for loop and the while loop. The for loop is used to iterate over a sequence, such as a list, a string, or a range of numbers. For each item in the sequence, the code inside the loop is executed. For example, to print each number in a list, you could do:



```python
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number)
```

The range() function is often used with for loops to iterate a specific number of times. For example, for i in range(5): will execute the loop body five times, with the variable i taking on the values 0, 1, 2, 3, and 4. The while loop is used to repeatedly execute a block of code as long as a specified condition remains true. The loop continues until the condition becomes false. It is crucial to ensure that the condition will eventually become false to avoid creating an infinite loop. Here's an example of a while loop that counts down from 5:



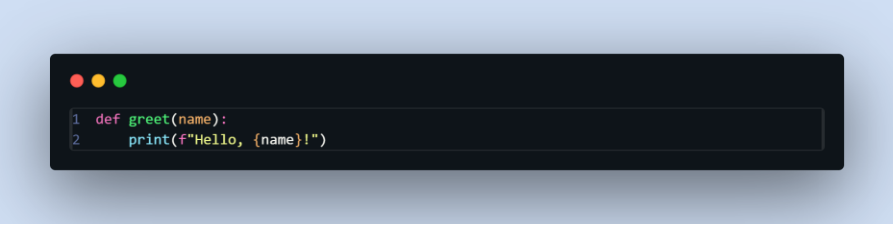
In this case, the loop will execute if the value of count is greater than 0. In each iteration, the value of count is decreased by 1, so eventually, the condition will become false, and the loop will terminate.

Python allows for the nesting of control structures, meaning you can place if statements inside loops, or loops inside if statements, or even loops inside other loops. This allows for the creation of more complex control flow logic. For example, you might have a loop that iterates through a list of lists, and inside the outer loop, you have an if statement that checks a condition on an element of the inner list. Understanding how to use and nest these control flow structures is essential for writing programs that can perform complex tasks based on different conditions and data.

5. Defining and Using Functions in Python:

Functions in Python are reusable blocks of code that are designed to perform a specific task. They help to organize code, make it more modular, and avoid repetition. Defining and using functions is a key aspect of writing efficient and maintainable Python programs.

To define a function in Python, you use the def keyword, followed by the name of the function, a pair of parentheses that may contain parameters (arguments), and a colon. The body of the function, which contains the code to be executed when the function is called, is indented below the def line. Here's a simple example of a function that greets a person:



In this example, greet is the name of the function, and name is a parameter that the function accepts.

To call or execute a function, you simply use its name followed by parentheses, and if the function expects any arguments, you provide those values inside the parentheses. For the greet function defined above, you would call it like this: greet("Alice"), which would print "Hello, Alice.

Functions can accept different types of arguments. Positional arguments are passed to the function based on their order in the function definition. For example, if a function is defined as def add(x, y):, then when you call it as add(5, 3), the value 5 is assigned to the parameter x and 3 is assigned to y.

Keyword arguments allow you to pass arguments using the parameter name followed by an equals sign. This means the order in which you pass them doesn't matter. For the add function, you could call it as add(y=3, x=5), which would still result in x being 5 and y being 3.

Default arguments are defined by providing a default value for a parameter in the function definition. If a value for that argument is not provided in the function call, the default value is used. For example, def power(base, exponent=2): defines a function where exponent has a default value of 2. You can call it as power(4) (in which case it will calculate 4 to the power of 2) or as power(4, 3) (to calculate 4 to the power of 3).

Functions can also return values back to the caller using the return statement. When a return statement is encountered in a function, the function execution stops, and the specified value is returned. If a function does not have a return statement, or if the return statement is used without a value, the function implicitly returns None. You can capture the returned value by assigning the result of the function call to a variable.

For example,

```python
def multiply(a, b):
    return a * b
result = multiply(6, 7)
print(result)  # Output: 42
```

The scope of a variable refers to the region of the program where that variable can be accessed. Python has two main types of scope: local and global.

Local Scope: Variables defined inside a function have local scope. This means they can only be accessed from within that function. They are created when the function is called and destroyed when the function finishes executing.

Global scope: Variables defined outside of any function have global scope. They can be accessed from anywhere in the program, including within functions. However, to modify a global variable from within a function, you typically need to use the global keyword to indicate that you are referring to the global variable rather than creating a new local variable with the same name. Understanding the scope of variables is important to avoid naming conflicts and to manage the data flow in your program effectively.

If you try to modify a global variable inside a function **without the global keyword**, Python **treats it as a new local variable**, which leads to unexpected results.

```python
x = 10  # Global variable

def update():
    x = 20  # This creates a new local variable, doesn't modify the global one
    print("Inside function:", x)

update()
print("Outside function:", x)  # Global 'x' remains unchanged
```

To **modify a global variable inside a function**, use the global keyword:

```python
x = 10  # Global variable

def update():
    global x  # Declaring that we are modifying the global 'x'
    x = 20
    print("Inside function:", x)

update()
print("Outside function:", x)  # Now, the global 'x' is updated!
```

7. Introduction to Object-Oriented Programming (OOP) in Python:

1) What is Object-Oriented Programming (OOP)?

OOP is a programming paradigm where **data and behavior** are grouped together into **objects**.
It makes programs **organized, reusable, and scalable**.

📌 **Key ideas of OOP**:

- **Objects**: Self-contained entities with data and behavior.
- **Classes**: Blueprints for creating objects.
- **Attributes**: Variables that store data inside an object.
- **Methods**: Functions that define an object's behavior.

2)What is a Class?

A **class** is like a **blueprint** for creating objects.

```python
class Dog:
    def __init__(self, name, breed):  # Constructor
        self.name = name   # Attribute
        self.breed = breed # Attribute

    def bark(self):  # Method
        print("Woof!")
```

Explanation

- **class Dog:** → Defines a new class named **Dog**.
- **__init__(self, name, breed):** → This is a **constructor**. It runs **automatically** when we create a new Dog object.
- **self.name** = name → Assigns a **name** to the dog.
- **self.breed** = breed → Assigns a **breed** to the dog.
- **bark(self):** → A method that makes the dog **bark**.

3) What is an Object?

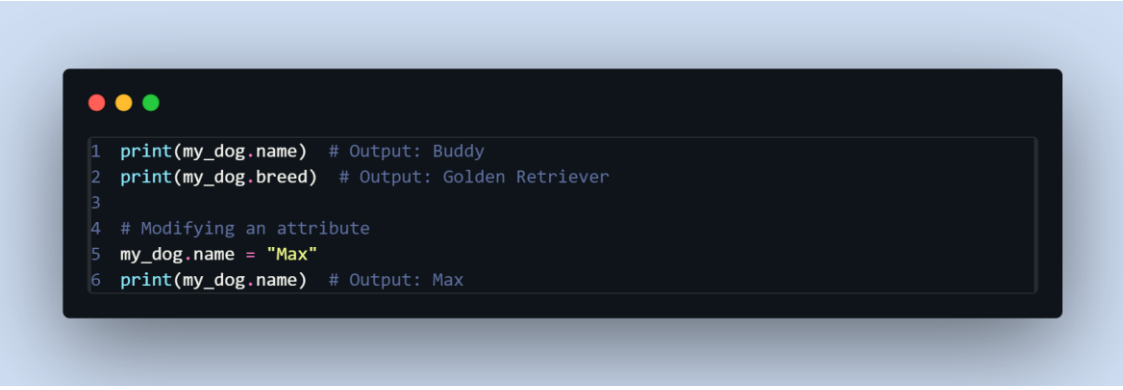
An object is an instance of a class. It is created by calling the class as a function.

```python
my_dog = Dog("Buddy", "Golden Retriever")  # Create an object
```

my_dog is an **instance** of the **Dog** class.
"Buddy" is the **name**.
"Golden Retriever" is the **breed**.

Attributes in OOP:

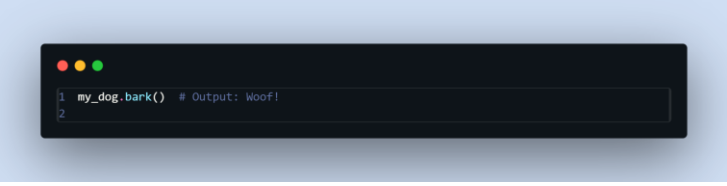
Attributes are **variables** inside an object. You can **access and modify** them using **dot notation**.



```python
print(my_dog.name)  # Output: Buddy
print(my_dog.breed)  # Output: Golden Retriever

# Modifying an attribute
my_dog.name = "Max"
print(my_dog.name)  # Output: Max
```

Methods in OOP:

Methods are **functions** inside a class that define an object's behavior.



```python
my_dog.bark()  # Output: Woof!
```

We call the bark() method using my_dog.bark().
Since bark() prints "Woof!", that's the output.

Advanced OOP Concepts

OOP is powerful because it allows:

- **Encapsulation**: Keeping data safe inside an object.
- **Inheritance**: Creating new classes based on existing ones.
- **Polymorphism**: Using the same method in different ways.