PCC-CS393: IT Workshop (Python)

Unit 5, 6, 7 - (Lists, Tuple and Dictionaries)

Sourav Das (Ph.D. Scholar (IIITK), M.Tech (CSE), MCA), MACM

Assistant Professor

Department of Computer Science & Engineering (AIML and IOT CS Including BCT)

Future Institute of Technology, Kolkata

Python List - Introduction

- A list is similar to an array that consists of a group of elements or items.
- Like an array, a list can store elements.
- However, unlike array, a list can store various types of elements.
- Hence, lists are more flexible and versatile than array.
- List is a datatype in Python.

What is a List?

- In daily scenarios, we use situations like list everyday.

- For instance, we take marks of a student for five subjects as:

  82, 78, 90, 75, 86

- The interesting thing to observe here is that, since this is a collection of similar (integer) type of elements, these elements could also be declared using an array.

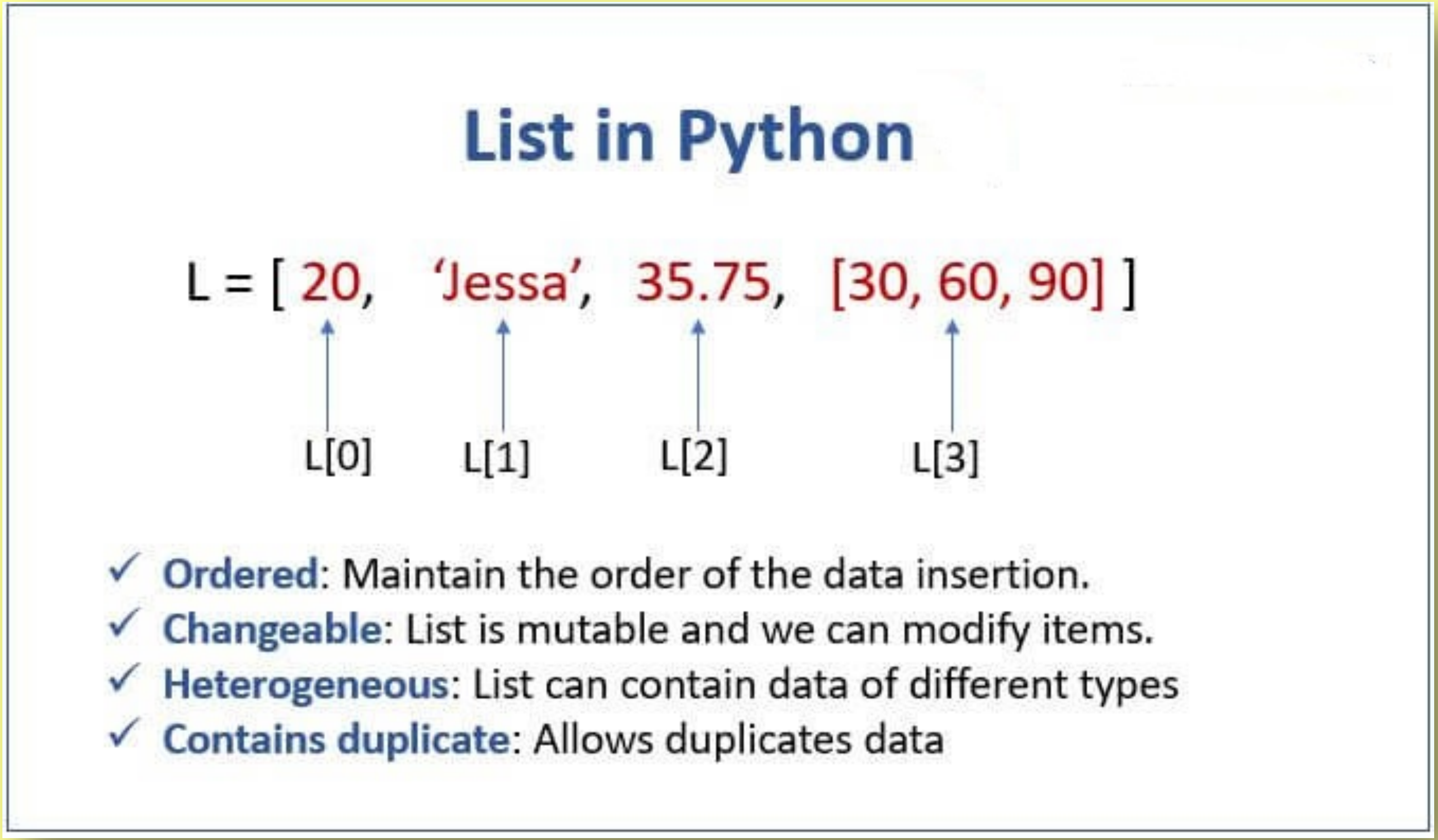
- However, elements like the following:

  9A, Python, 87, 95, 81

- Could also be declared and contained by a list.

List Properties

- Features of Python lists:



List in Python

```
L = [ 20,  'Jessa',  35.75,  [30, 60, 90] ]
```

✓ **Ordered**: Maintain the order of the data insertion.
✓ **Changeable**: List is mutable and we can modify items.
✓ **Heterogeneous**: List can contain data of different types
✓ **Contains duplicate**: Allows duplicates data

List Properties

- **Properties of a list:**

• **Mutable:** The elements of the list can be modified. We can add or remove items to the list after it has been created.

• **Ordered:** The items in the lists are ordered. Each item has a unique index value. The new items will be added to the end of the list.

• **Heterogenous:** The list can contain different kinds of elements i.e.; they can contain elements of string, integer, Boolean, or any type.

• **Duplicating:** The list can contain duplicates i.e., lists can have two items with the same values.

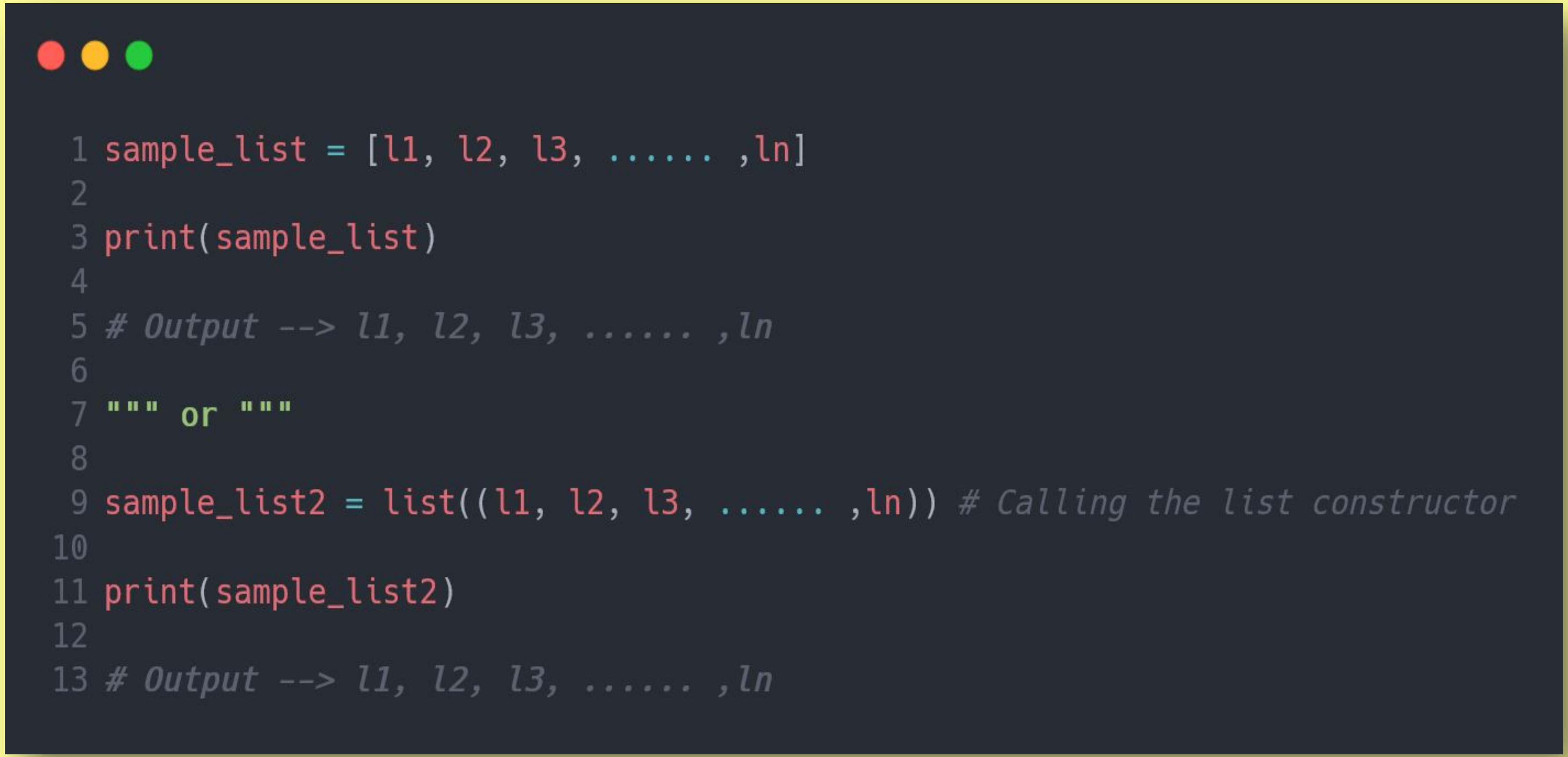
List Usage

❑ **Why to use a list?**

- The list data structure is very flexible It has many unique inbuilt functionalities like pop(), append(), etc. which makes it easier, where the data keeps changing.
- Also, the list can contain duplicate elements i.e. two or more items can have the same values.
- Lists are Heterogeneous i.e., different kinds of objects/elements can be added
- As Lists are mutable it is used in applications where the values of the items change frequently.

List Syntax

❑ **Declaring and printing a simple list in Python:**



```python
sample_list = [l1, l2, l3, ...... ,ln]

print(sample_list)

# Output --> l1, l2, l3, ...... ,ln

""" or """

sample_list2 = list((l1, l2, l3, ...... ,ln)) # Calling the list constructor

print(sample_list2)

# Output --> l1, l2, l3, ...... ,ln
```

Accessing Items of a List

- The items in a list can be accessed through indexing and slicing.

- Using indexing, we can access any item from a list using its index number.

- Using slicing, we can access a range of items from a list.

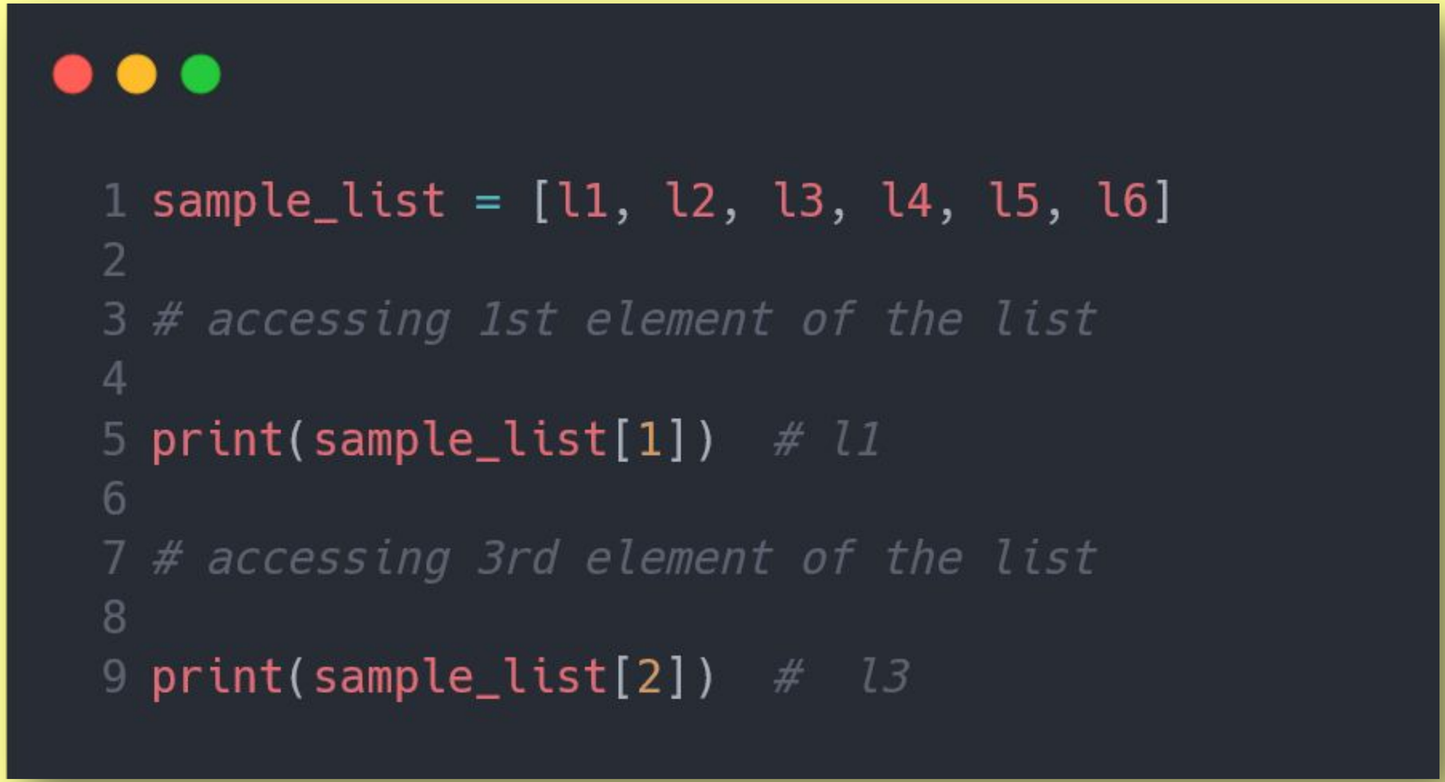
List Indexing

- The list elements can be accessed using the “indexing” technique. Lists are ordered collections with unique indexes for each item. We can access the items in the list using this index number.

- To access the elements in the list from left to right, the index value starts from zero to (length of the list-1) can be used. For example, if we want to access the 3rd element we need to use 2 since the index value starts from 0.

List Indexing

- Accessing a list elements in Python using the index values:



```python
sample_list = [l1, l2, l3, l4, l5, l6]

# accessing 1st element of the list

print(sample_list[1])  # l1

# accessing 3rd element of the list

print(sample_list[2])  #  l3
```

List Indexing

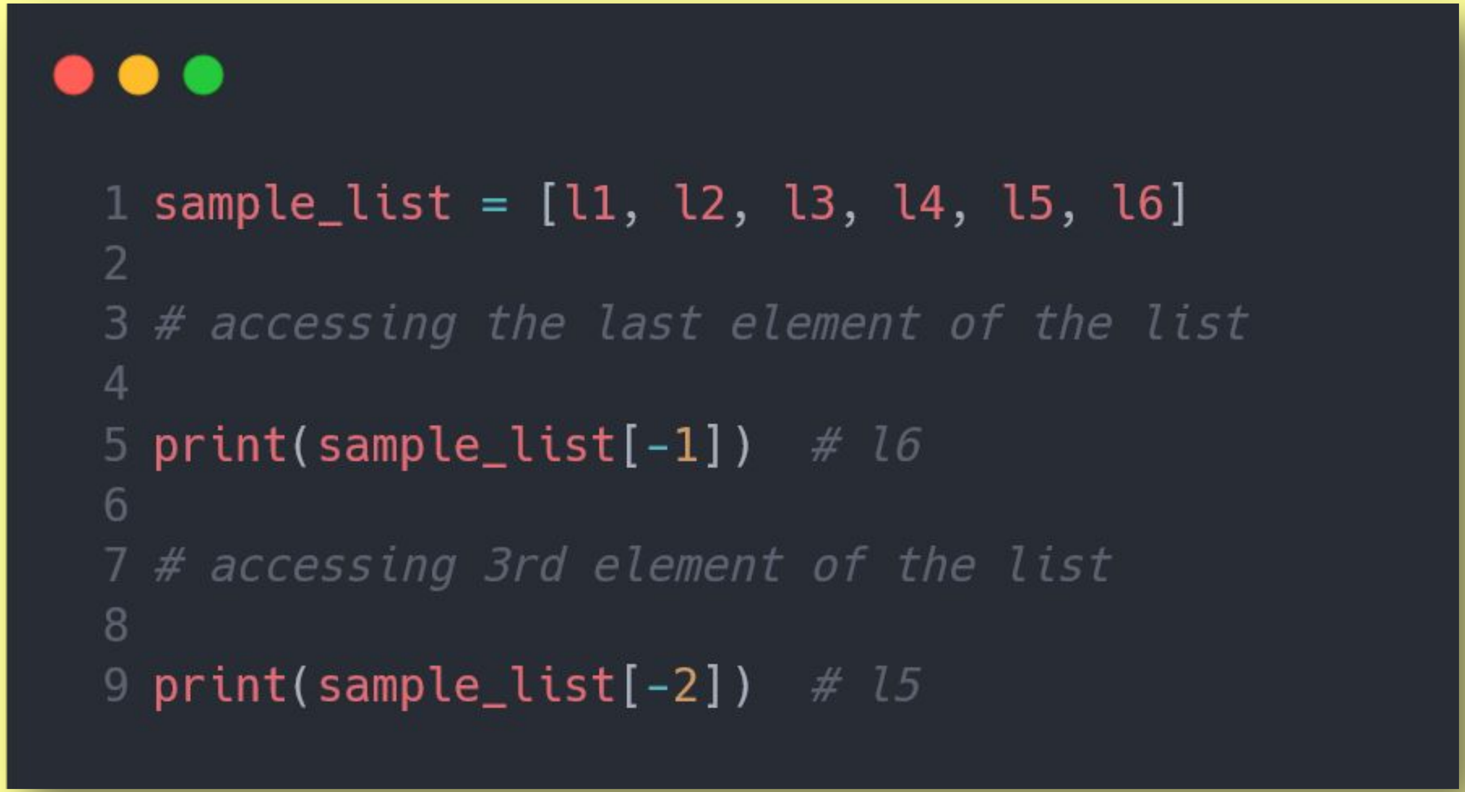
- ❑ **Negative Indexing:**

  - The elements in the list can be accessed from right to left by using negative indexing.

  - The negative value starts from -1 to -length of the list. It indicates that the list is indexed from the reverse/backward.

List Indexing

- Accessing a list elements in reverse order using the negative index values:



```python
sample_list = [l1, l2, l3, l4, l5, l6]

# accessing the last element of the list

print(sample_list[-1])  # l6

# accessing 3rd element of the list

print(sample_list[-2])  # l5
```

List Indexing

- As Lists are ordered sequences of items, the index values start from 0 to the Lists length.
- Whenever we try to access an item with an index more than the Lists length, it will throw the “Index Error”.
- Similarly, the index values are always an integer. If we give any other type, then it will throw Type Error.

List Slicing

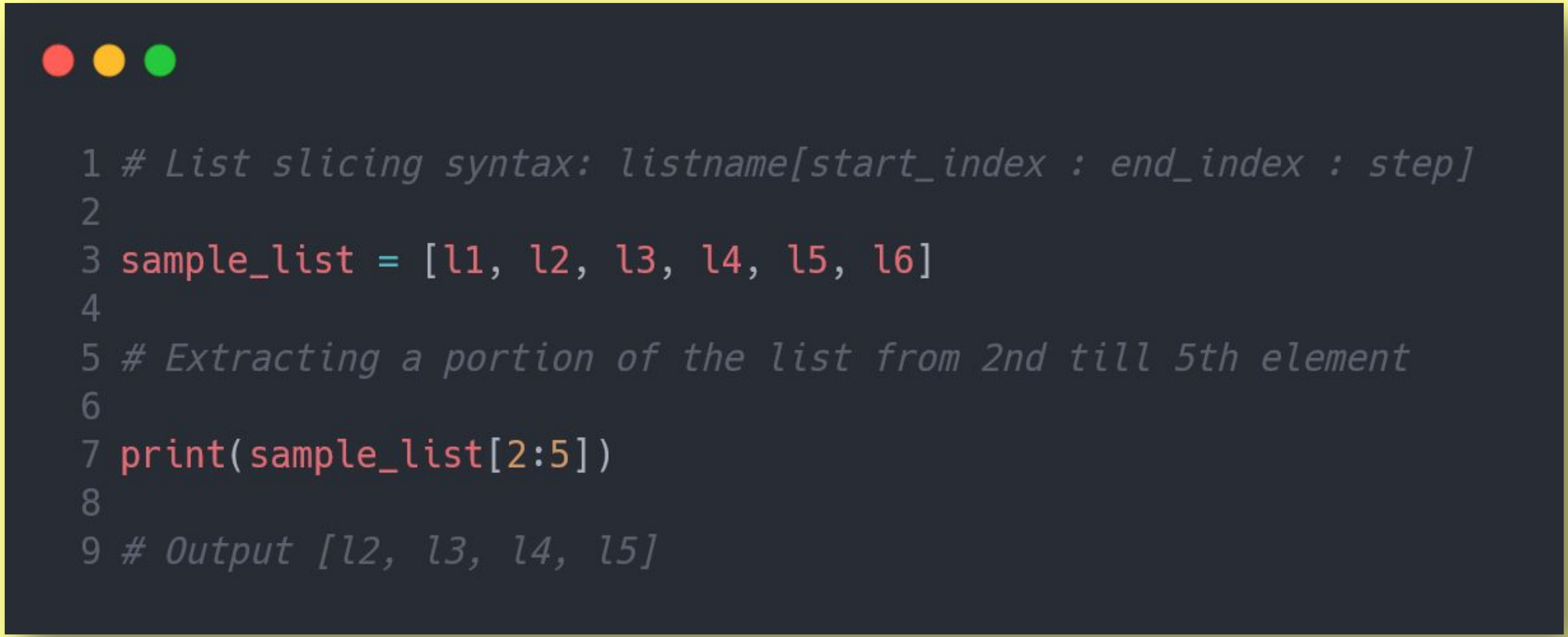
- Slicing a list implies that accessing a range of elements in a list.

- For example, if we want to get the elements in the position from 3 to 7, we can use the slicing method. We can even modify the values in a range by using this slicing technique.

- The ***start_index*** denotes the index position from where the slicing should begin and the ***end_index*** parameter denotes the index positions till which the slicing should be done.
- The step allows us to take each nth-element within a ***start_index:end_index*** range.

List Slicing

- Declaring a list and slicing a sub-list in Python:



```python
# List slicing syntax: listname[start_index : end_index : step]

sample_list = [l1, l2, l3, l4, l5, l6]

# Extracting a portion of the list from 2nd till 5th element

print(sample_list[2:5])

# Output [l2, l3, l4, l5]
```

Iterating a List

- The objects in the list can be iterated over one by one, by using a for a loop.

❑ **Iterate along with an index number:**

- The index value starts from 0 to (length of the list - 1). Hence using the function range() is ideal for this scenario.

- The range function returns a sequence of numbers. By default, it returns starting from 0 to the specified number (increments by 1). The starting and ending values can be passed according to our needs.

Adding List Elements

- We can add a new element/list of elements to the list using the list methods such as append(), insert(), and extend().

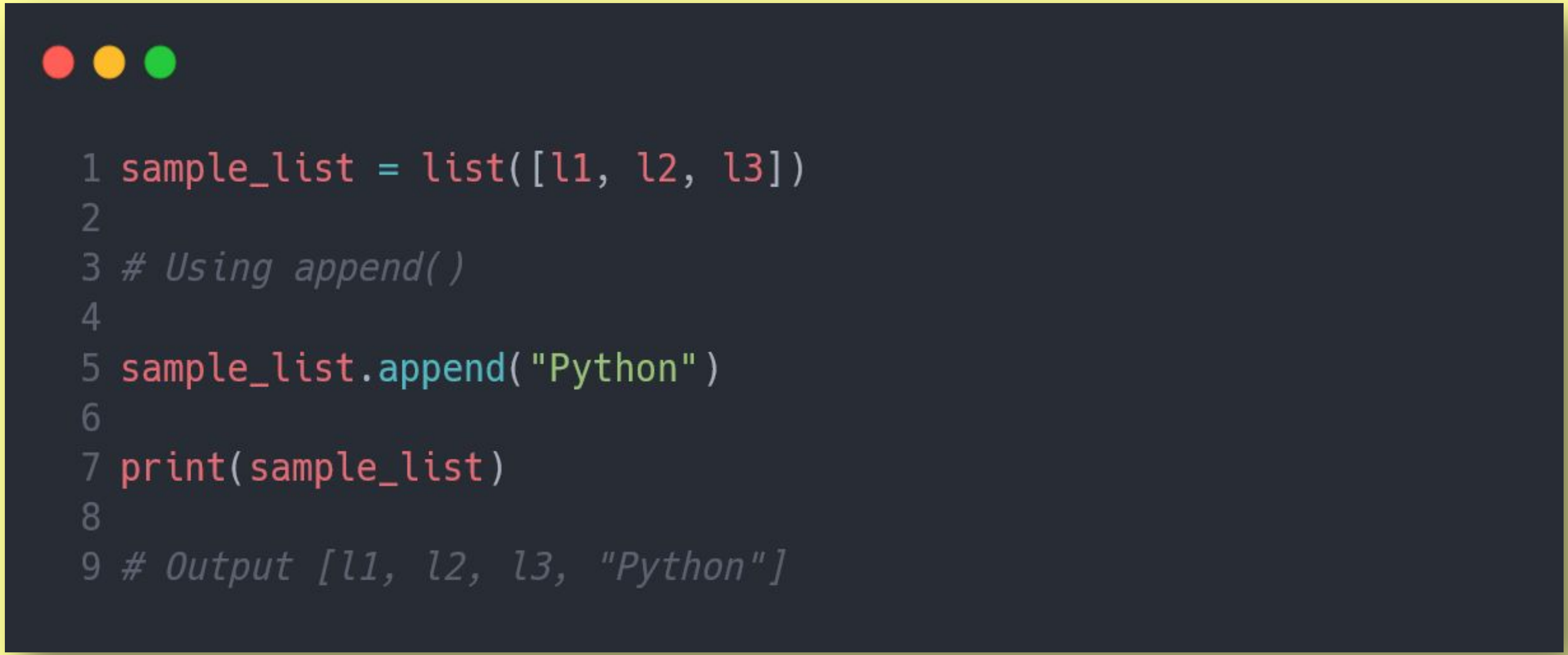
❑ **Append item at the end of the list:**

- The append() method will accept only one parameter and add it at the end of the list.

- Let’s see the example to add the element “Python” at the end of a list.

List Append

- Appending an element at the end of a list in Python:



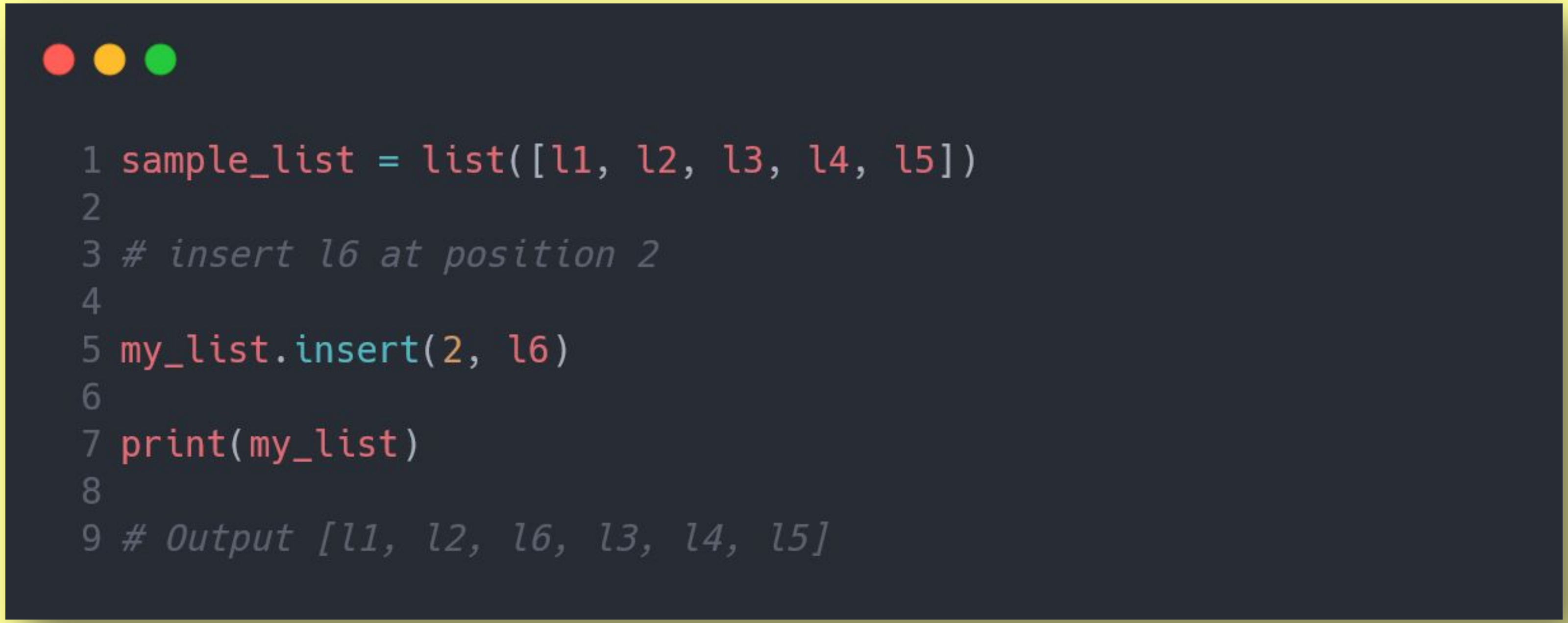
```
sample_list = list([l1, l2, l3])

# Using append()

sample_list.append("Python")

print(sample_list)

# Output [l1, l2, l3, "Python"]
```

List Insert

- We can insert a list element by using the insert() method to add the element at the specified position in the list. The insert method accepts two parameters position and object.

insert(index, object)

- It will insert the object in the specified index.



```python
sample_list = list([l1, l2, l3, l4, l5])

# insert l6 at position 2

my_list.insert(2, l6)

print(my_list)

# Output [l1, l2, l6, l3, l4, l5]
```

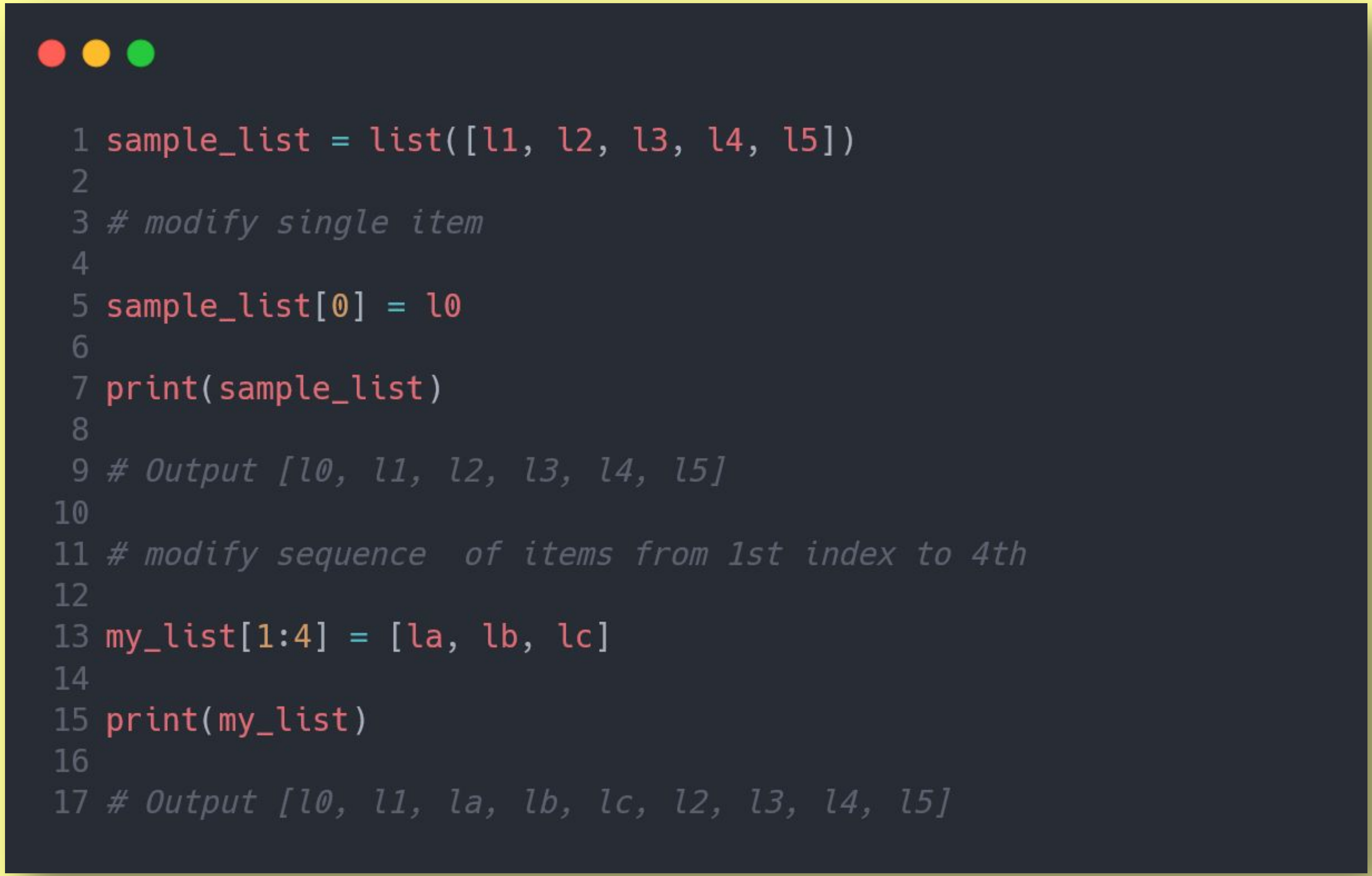
Modifying a List

- The Python list is a mutable sequence of iterable objects. It means we can modify the items of a list. Use the index number and assignment operator (=) to assign a new value to an item.

❑ **We can modify a list by two approaches:**

- Modify the individual item.
- Modify the range of items

Modifying a List



```
sample_list = list([l1, l2, l3, l4, l5])

# modify single item

sample_list[0] = l0

print(sample_list)

# Output [l0, l1, l2, l3, l4, l5]

# modify sequence  of items from 1st index to 4th

my_list[1:4] = [la, lb, lc]

print(my_list)

# Output [l0, l1, la, lb, lc, l2, l3, l4, l5]
```

Removing List Elements

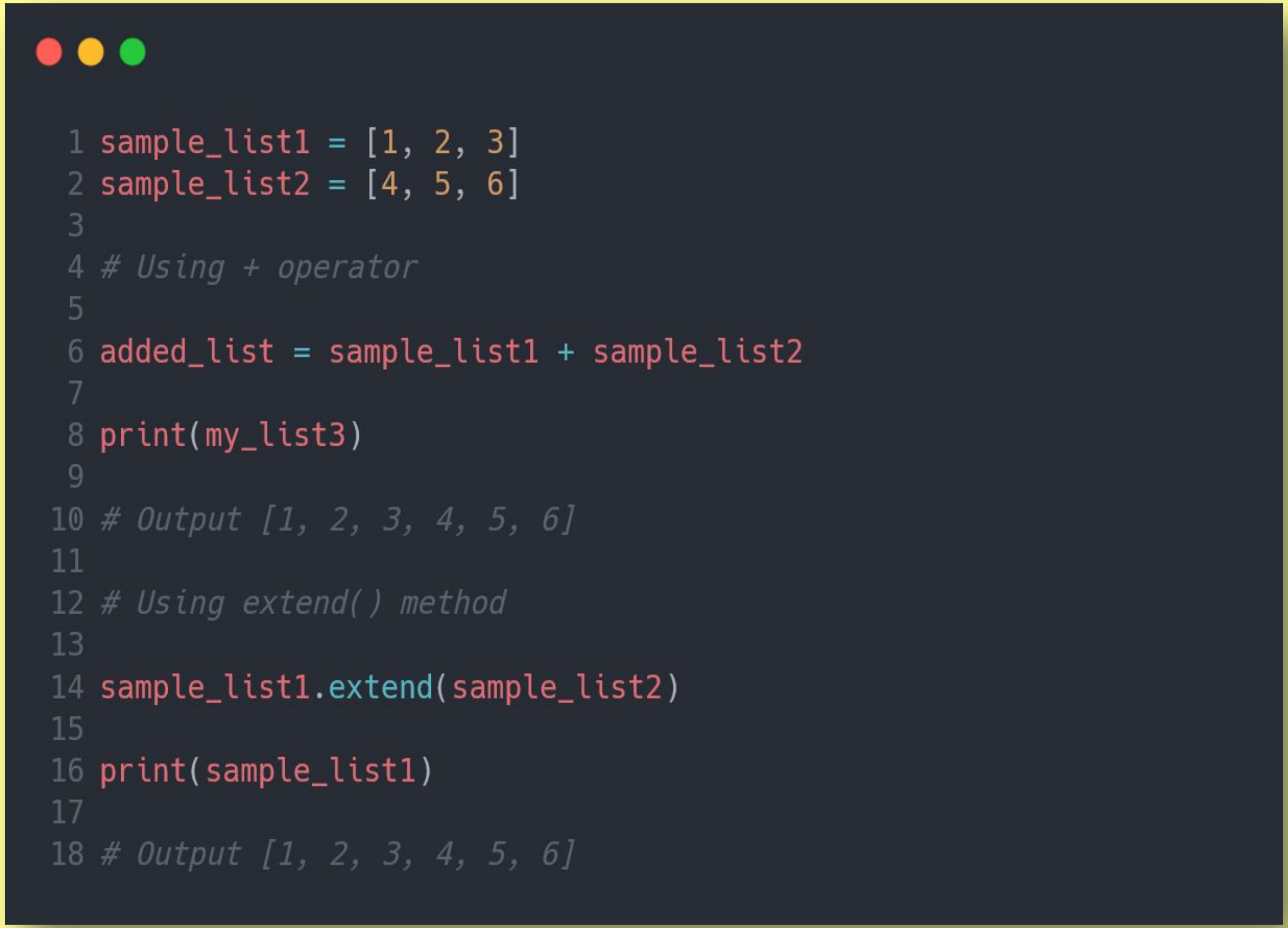
- The elements from the list can be removed using the following list methods.

Functions	Descriptions
o **remove(item)**	To remove the first occurrence of the item from the list.
o **pop(index)**	To remove and return the item at the given index from the list.
o **clear()**	To remove all items from the list. The output will be an empty list.
o **del list_name**	To delete the entire list.

Concatenating Lists

- The concatenation of two lists means merging of two lists. There are two ways to do that:

- Using the ***+*** operator.
- Using the ***extend()*** method. The ***extend()*** method appends the new list's items at the end of the calling list.



```
sample_list1 = [1, 2, 3]
sample_list2 = [4, 5, 6]

# Using + operator

added_list = sample_list1 + sample_list2

print(my_list3)

# Output [1, 2, 3, 4, 5, 6]

# Using extend( ) method

sample_list1.extend(sample_list2)

print(sample_list1)

# Output [1, 2, 3, 4, 5, 6]
```

List Copy

- There are two ways by which a copy of a list can be created.

❑ **Using assignment operator (=):**

- This is a straightforward way of creating a copy. In this method, the new list will be a deep copy. The changes that we make in the original list will be reflected in the new list.

- This is called deep copying.

List Copy

- The changes made to the original list are reflected in the copied list as well.

- When we assign ***list1 = list2***, we are making them refer to the same list object, so when we modify one of them, all references associated with that object reflect the current state of the object.

- Hence, we should not use the assignment operator to copy the dictionary, instead we can use the ***copy()*** method.

List Copy

❑ **Using the copy() method:**

- The copy method can be used to create a copy of a list.
- This will create a new list and any changes made in the original list will not reflect in the new list. This is shallow copying.

List Functions

- There are a diverse set of in-built list functions in Python. We can use them to create, alter or even remove the elements of the lists or the lists themselves in multiple ways.

- We can perform list organization operations over the list by using certain functions like ***sort()***, ***reverse()***, ***clear()*** etc.

❑ **Sort List using sort():**

- The sort function sorts the elements in the list in ascending order.

List Functions

- **Reverse a List using reverse():**

  - The reverse function is used to reverse the elements in the list.

- **Using max() & min():**

  - The max function returns the maximum value in the list while the min function returns the minimum value in the list.

List Functions

- **Using sum():**

  - The sum function returns the sum of all the elements in the list.

- **Using all():**

  - In the case of all() function, the return value will be true only when all the values inside the list are true.

List Functions

- **Using any():**

  - The ***any()*** method will return true if there is at least one true value.
  - In the case of Empty List, it will return false.

Nested List

- The list can contain another list (sub-list), which in turn contains another list and so on. This is termed a nested list.

❑ **Syntax of general nested lists:**

sample_list = [3, 4, 5, 6, 3, [1, 2, 3], 4]

Sample_list = [1, 2, 3, [4, 5, 6 [7, 8, 9], 10], 11, 12]

Python Tuple - Introduction

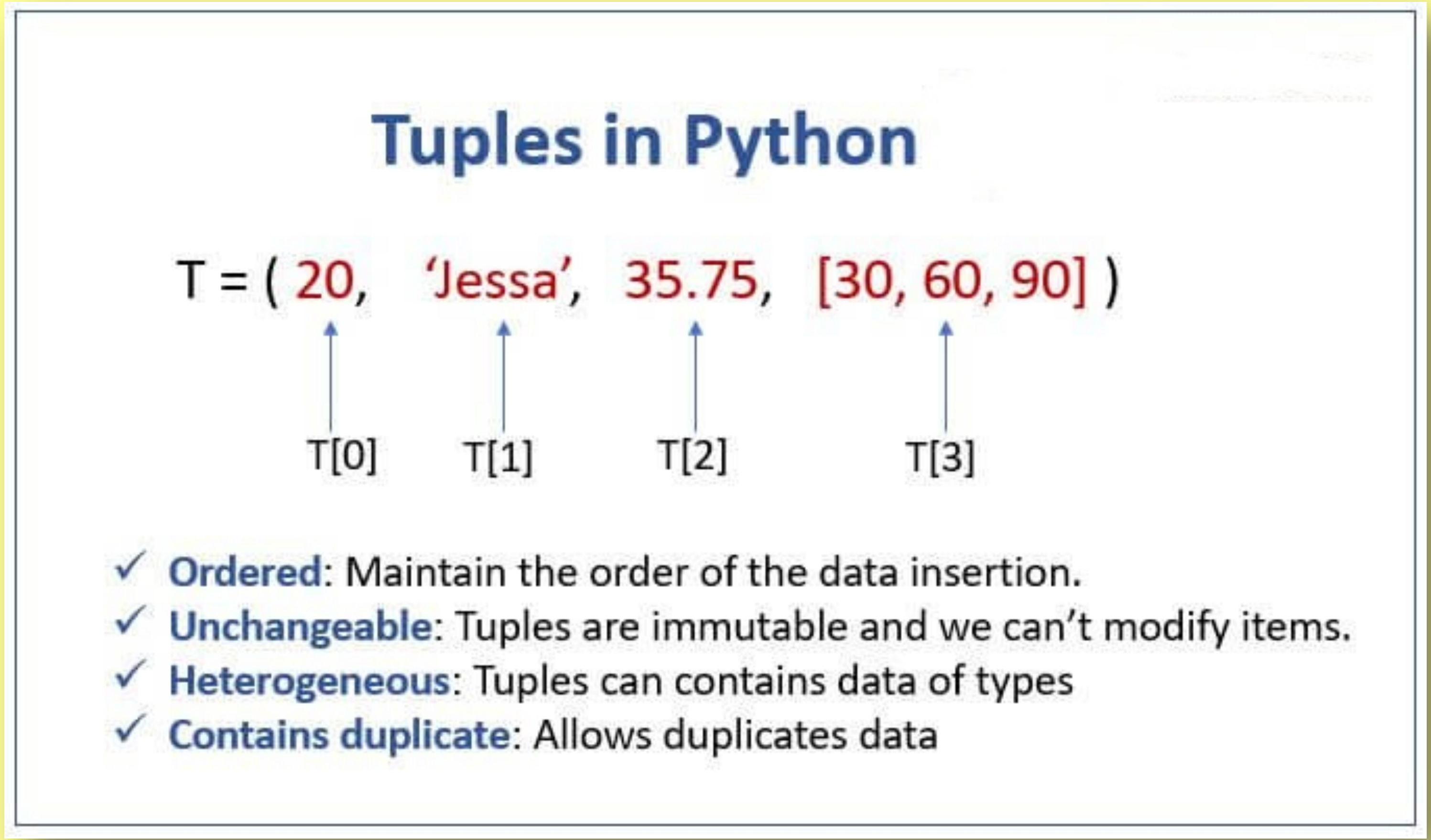
- Tuples are ordered collections of heterogeneous data that are unchangeable. Heterogeneous means tuple can store variables of all types.

❑ **Tuple has the following characteristics:**

- **Ordered:** Tuples are part of sequence data types, which means they hold the order of the data insertion. It maintains the index value for each item.
- **Unchangeable:** Tuples are unchangeable, which means that we cannot add or delete items to the tuple after creation.
- **Heterogeneous:** Tuples are a sequence of data of different data types (like integer, float, list, string, etc) and can be accessed through indexing and slicing.
- **Contains Duplicates:** Tuples can contain duplicates, which means they can have items with the same value.

Tuple Properties

- **Features of Python tuple:**


Tuples in Python

- ✓ Ordered: Maintain the order of the data insertion.
- ✓ Unchangeable: Tuples are immutable and we can't modify items.
- ✓ Heterogeneous: Tuples can contains data of types
- ✓ Contains duplicate: Allows duplicates data

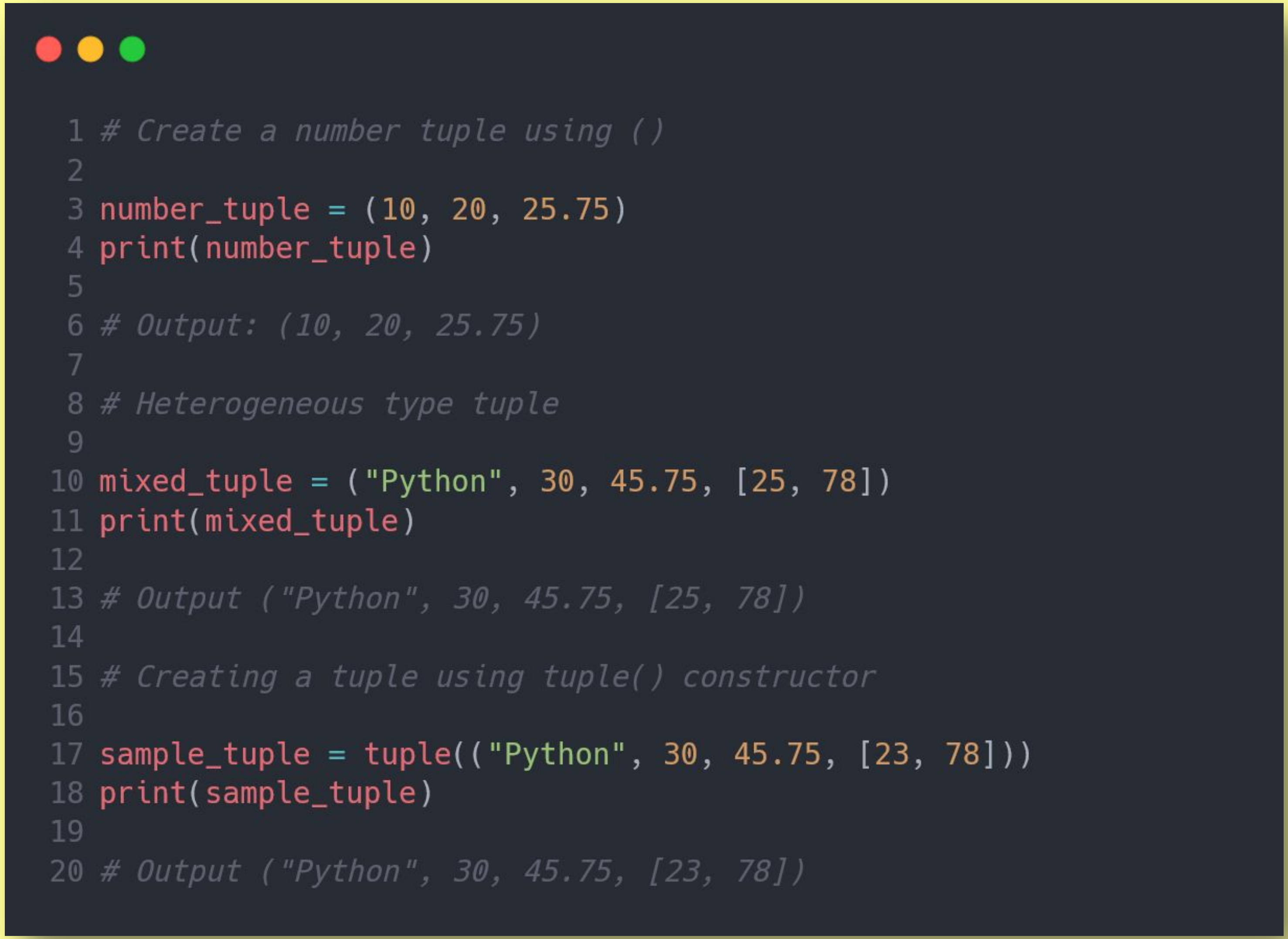
Creating a Tuple

- **We can create a tuple using the two ways:**

- **Using parenthesis ():** A tuple is created by enclosing comma-separated items inside rounded brackets.

- **Using a tuple() constructor:** Create a tuple by passing the comma-separated items inside the tuple().

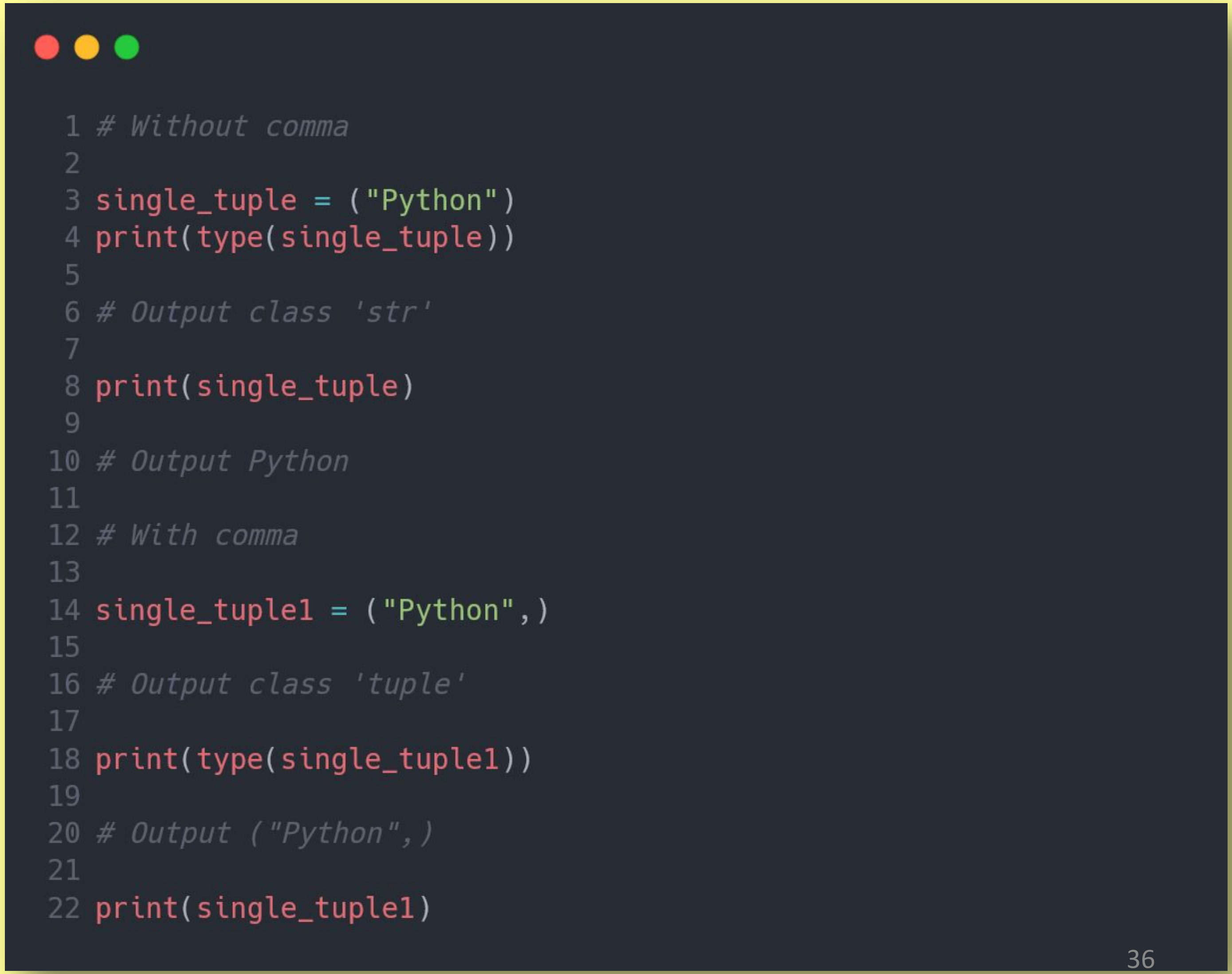
Creating a Tuple



```python
# Create a number tuple using ()

number_tuple = (10, 20, 25.75)
print(number_tuple)

# Output: (10, 20, 25.75)

# Heterogeneous type tuple

mixed_tuple = ("Python", 30, 45.75, [25, 78])
print(mixed_tuple)

# Output ("Python", 30, 45.75, [25, 78])

# Creating a tuple using tuple() constructor

sample_tuple = tuple(("Python", 30, 45.75, [23, 78]))
print(sample_tuple)

# Output ("Python", 30, 45.75, [23, 78])
```

Single Tuple

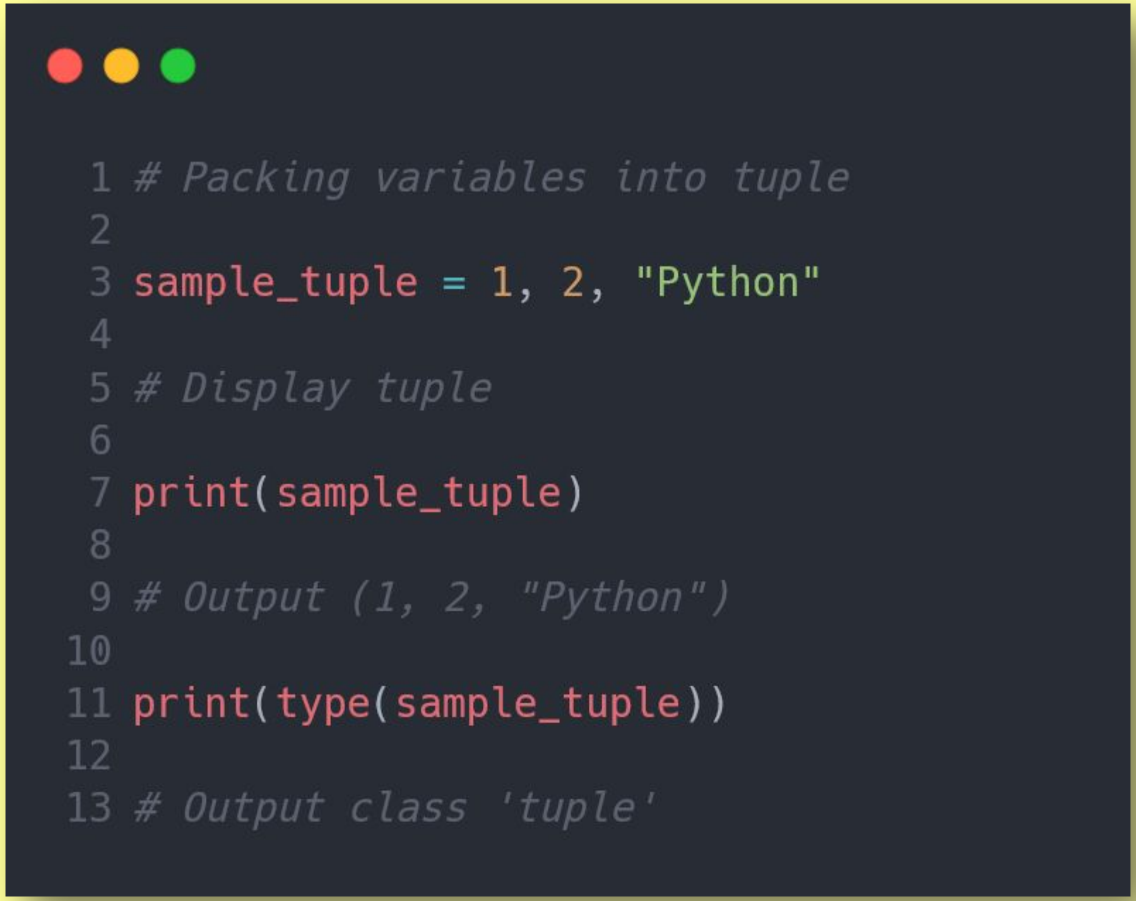
- A single item tuple is created by enclosing one item inside parentheses followed by a comma. If the tuple time is a string enclosed within parentheses and not followed by a comma, Python treats it as a str type.
- We can see in the output the first time, we did not add a comma after the "***Python***". So the variable type was ***class str***, and the second time it was a ***class tuple***.



```python
# Without comma

single_tuple = ("Python")
print(type(single_tuple))

# Output class 'str'

print(single_tuple)

# Output Python

# With comma

single_tuple1 = ("Python",)

# Output class 'tuple'

print(type(single_tuple1))

# Output ("Python",)

print(single_tuple1)
```

Packing Tuple

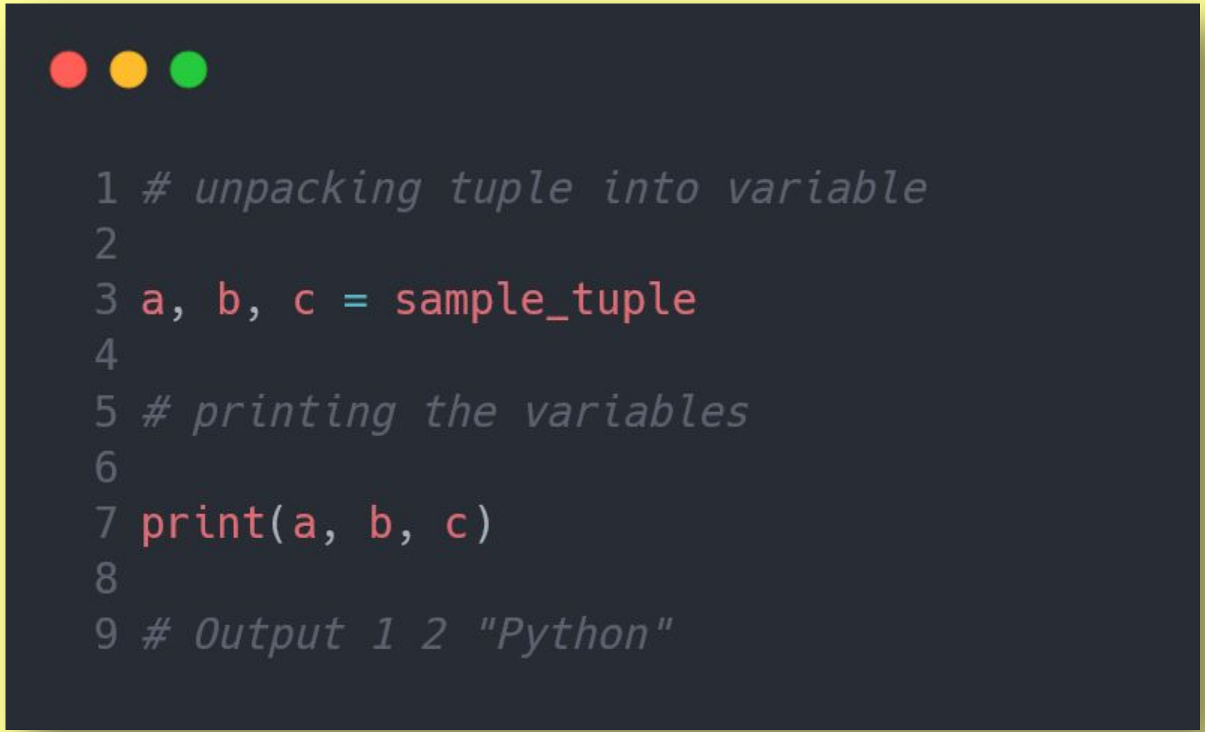
- A tuple can also be created without using a ***tuple()*** constructor or enclosing the items inside the parentheses. It is called the variable "***Packing***".
- In Python, we can create a tuple by packing a group of variables. Packing can be used when we want to collect multiple values in a single variable. Generally, this operation is referred to as tuple packing.



```python
# Packing variables into tuple

sample_tuple = 1, 2, "Python"

# Display tuple

print(sample_tuple)

# Output (1, 2, "Python")

print(type(sample_tuple))

# Output class 'tuple'
```

Unpacking Tuple

- Quite similar way, we can unpack the items by just assigning the tuple items to the same number of variables. This process is called "***Unpacking***".

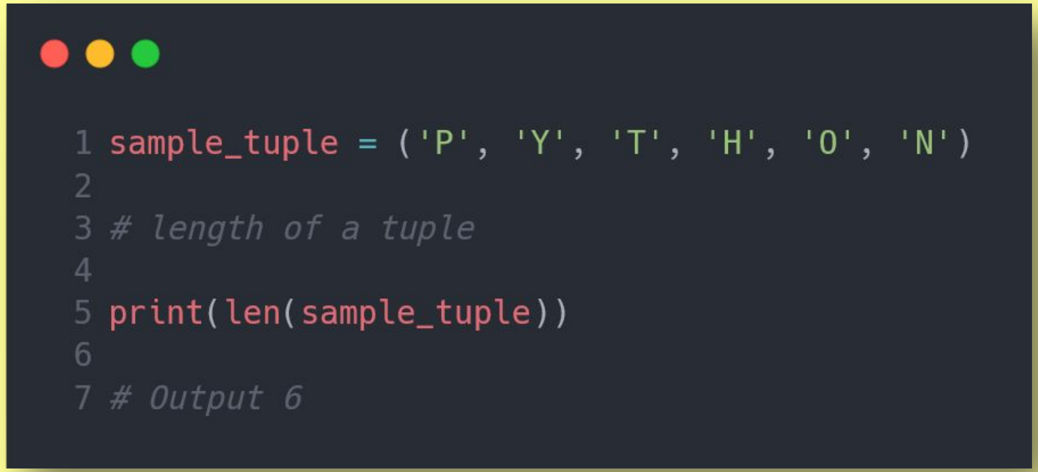


```python
# unpacking tuple into variable

a, b, c = sample_tuple

# printing the variables

print(a, b, c)

# Output 1 2 "Python"
```

- Here, the three tuple items are assigned to individual variables a, b, c respectively.
- In case we assign fewer variables than the number of items in the tuple, we will get the value error with the message "***too many values to unpack***".

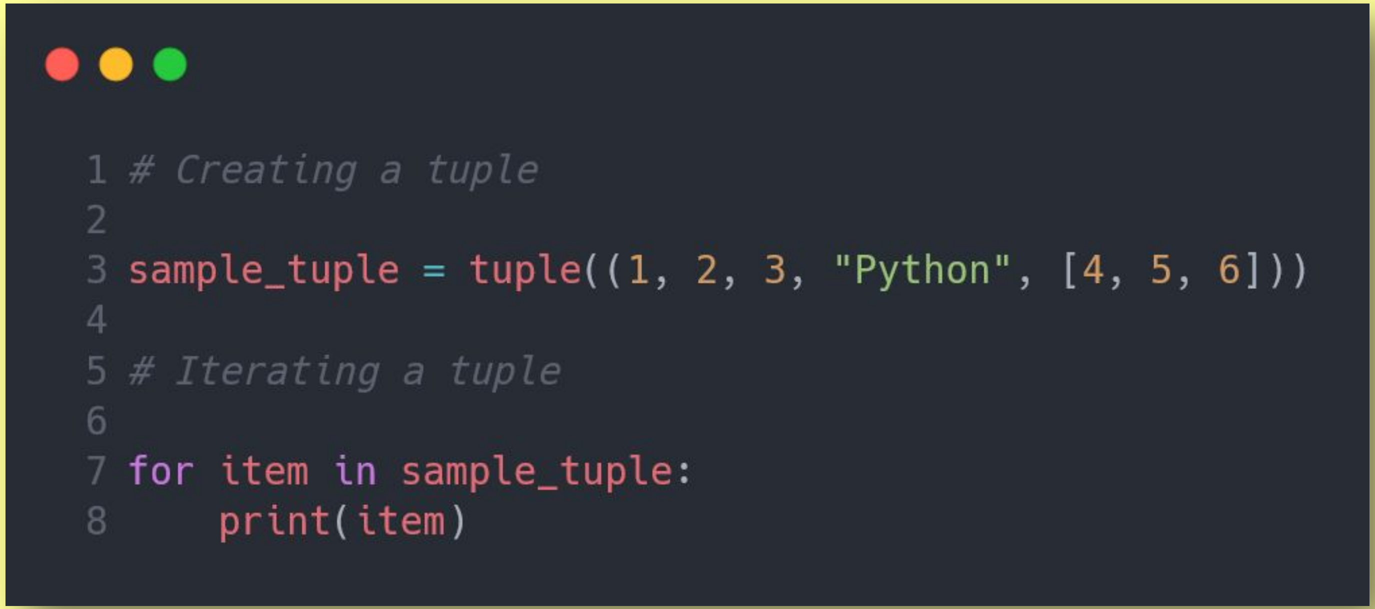
Tuple Length and Iteration

- ❑ **Tuple Length:**



```python
sample_tuple = ('P', 'Y', 'T', 'H', 'O', 'N')

# length of a tuple

print(len(sample_tuple))

# Output 6
```

- ❑ **Tuple Iteration:**



```python
# Creating a tuple

sample_tuple = tuple((1, 2, 3, "Python", [4, 5, 6]))

# Iterating a tuple

for item in sample_tuple:
    print(item)
```

Accessing Items of a Tuple

- Tuple can be accessed through indexing and slicing.

❑ **The two ways of accessing tuple items are:**

- Using ***indexing***, where we can access any item from a tuple using its index number.
- Using ***slicing***, where we can access a range of items from a tuple.

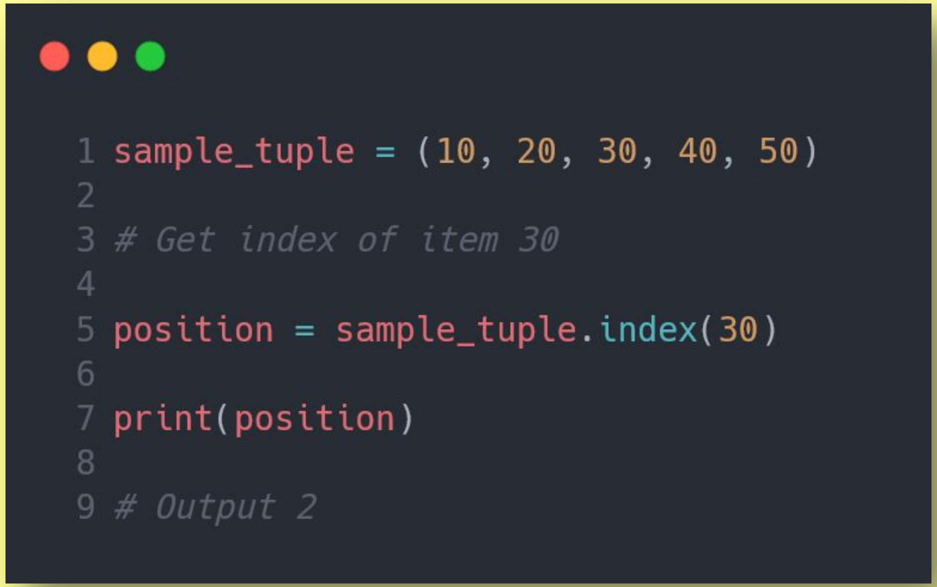
Finding Tuple Item

- We can search for a certain item in a tuple using the index() method and it will return the position of that particular item in the tuple.

❑ **The index() method accepts the following three arguments:**

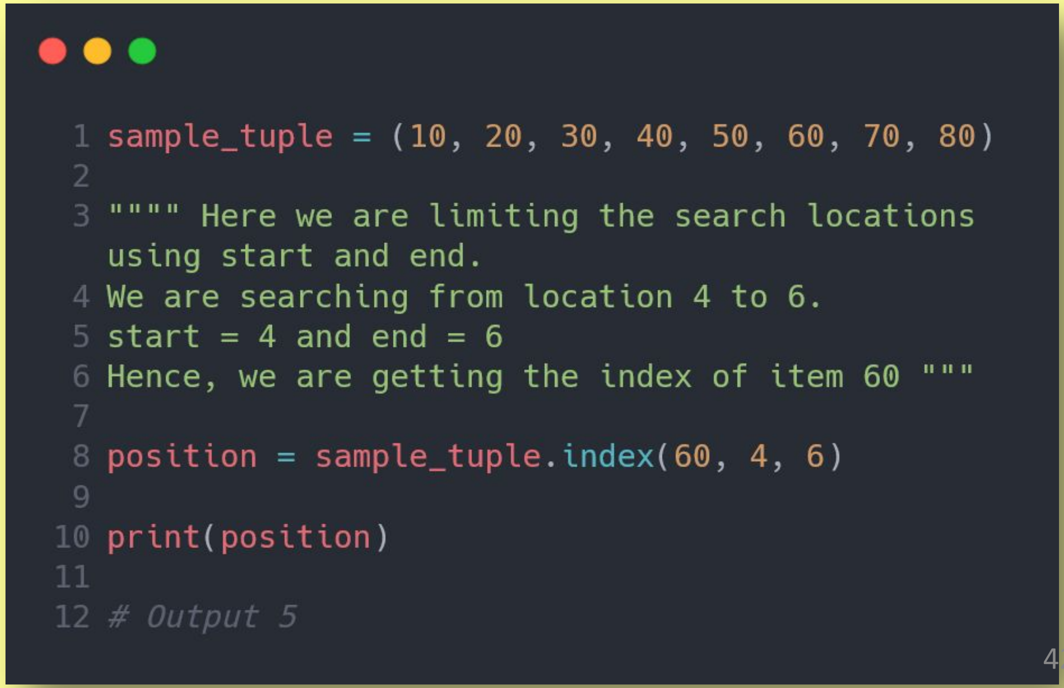
- **item:** The item which needs to be searched.
- **start (Optional):** The starting value of the index from which the search will start.
- **end (Optional):** The end value of the index search.

Finding Tuple Item



```python
sample_tuple = (10, 20, 30, 40, 50)

# Get index of item 30

position = sample_tuple.index(30)

print(position)

# Output 2
```

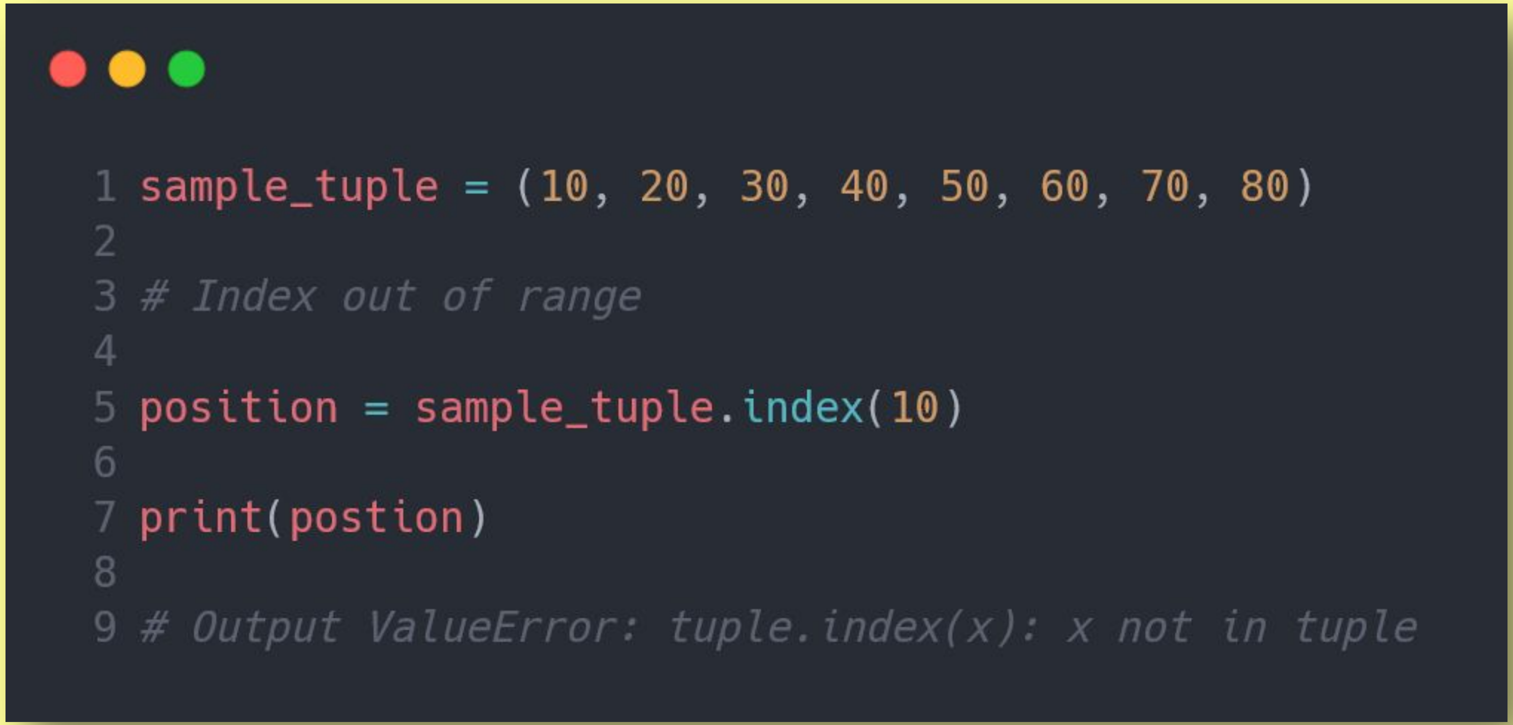
- Finding a tuple item within a range:



```python
sample_tuple = (10, 20, 30, 40, 50, 60, 70, 80)

"""" Here we are limiting the search locations using start and end.
We are searching from location 4 to 6.
start = 4 and end = 6
Hence, we getting the index of item 60 """

position = sample_tuple.index(60, 4, 6)

print(position)

# Output 5
```

Finding Tuple Item

- In case we mention any item that is not present then it will throw a value error like the following:



```python
sample_tuple = (10, 20, 30, 40, 50, 60, 70, 80)

# Index out of range

position = sample_tuple.index(10)

print(postion)

# Output ValueError: tuple.index(x): x not in tuple
```

Modifying Tuple Items

- A list is a mutable type, which means we can add or modify values in it, but tuples are immutable, so they cannot be changed.

- Also, because a tuple is immutable there are no built-in methods to add items to the tuple.

- If we try to modify the value, we will get an error.

- As a way around solution, we can convert the tuple to a list, add items, and then convert it back to a tuple. As tuples are ordered collection like lists the items always get added in the end.

Removing Tuple Item

- Tuples are immutable so there are ***no pop() or remove()*** methods for the tuple.

❑ **We can remove the items from a tuple using the following two ways:**

- Using the ***del*** keyword,
- By converting it into a list.

Removing Tuple Item

- **By converting it into a List:**

  - We can convert a tuple into a list and then remove any one item using the ***remove()*** method.

  - Then again we will convert it back into a tuple using the ***tuple()*** constructor.

Item Occurrence in Tuple

- Since a tuple can contain duplicate items.
- To determine how many times a specific item occurred in a tuple, we can use the ***count()*** method of a tuple object.
- The ***count()*** method accepts any value as a parameter and returns the number of times a particular value appears in a tuple.

Copying a Tuple

- We can create a copy of a tuple using the assignment operator ***"="*** .
- This operation creates only a reference copy and not a deep copy because tuples are immutable.

Tuple Concatenation

- We can concatenate two or more tuples in different ways. One thing to note here is that tuples allow duplicates, so if two tuples have the same item, it will be repeated twice in the resultant tuple.

❑ **Tuple concatenation can be done by three ways as following:**

- Using ***+*** operator,
- Using the ***sum(***) function,
- Using the ***chain()*** function.

Tuple Concatenation

- We can use the Python built-in function ***sum()*** to concatenate two tuples.

- However, the sum function of two iterables like tuples always needs to start with Empty Tuple. Let us see this with an example.

Tuple Concatenation

- We can connect (concatenate) Python tuples using the ***chain()*** function.
- The ***chain()*** function is part of the ***itertools*** module in python.
- It makes an iterator, which will return all the first iterable items (a tuple in our case), which will be followed by the second iterable.
- We can pass any number of tuples to the ***chain()*** function.

Nested Tuple

- Nested tuples are tuples within a tuple i.e., when a tuple contains another tuple as its member then it is called a nested tuple.

- In order to retrieve the items of the inner tuple we need a nested for loop.

Tuple Functions

- Like Python list, tuple also has several built-in functions for various tasks.

❑ **min() and max():**

- As the name suggests the max() function returns the maximum item in a tuple and min() returns the minimum value in a tuple.

❑ **all():**

- In the case of all() function, the return value will be true only when all the values inside are true

Tuple Functions

❑ **all():**

Item values in a tuple	Return value
○ All values are True	True
○ One or more False values	False
○ All False values	False
○ Empty tuple	True

Tuple Functions

- ***any():***

  - The any() method will return true if there is at least one true value.
  - In the case of the empty tuple, it will return false.

Item values in a tuple	Return value
○ All values are True	True
○ One or more False values	True
○ All False values	False
○ Empty tuple	False

Python Dictionary - Introduction

- In Python, dictionaries are unordered collections of unique values stored in (***Key-Value***) pairs.

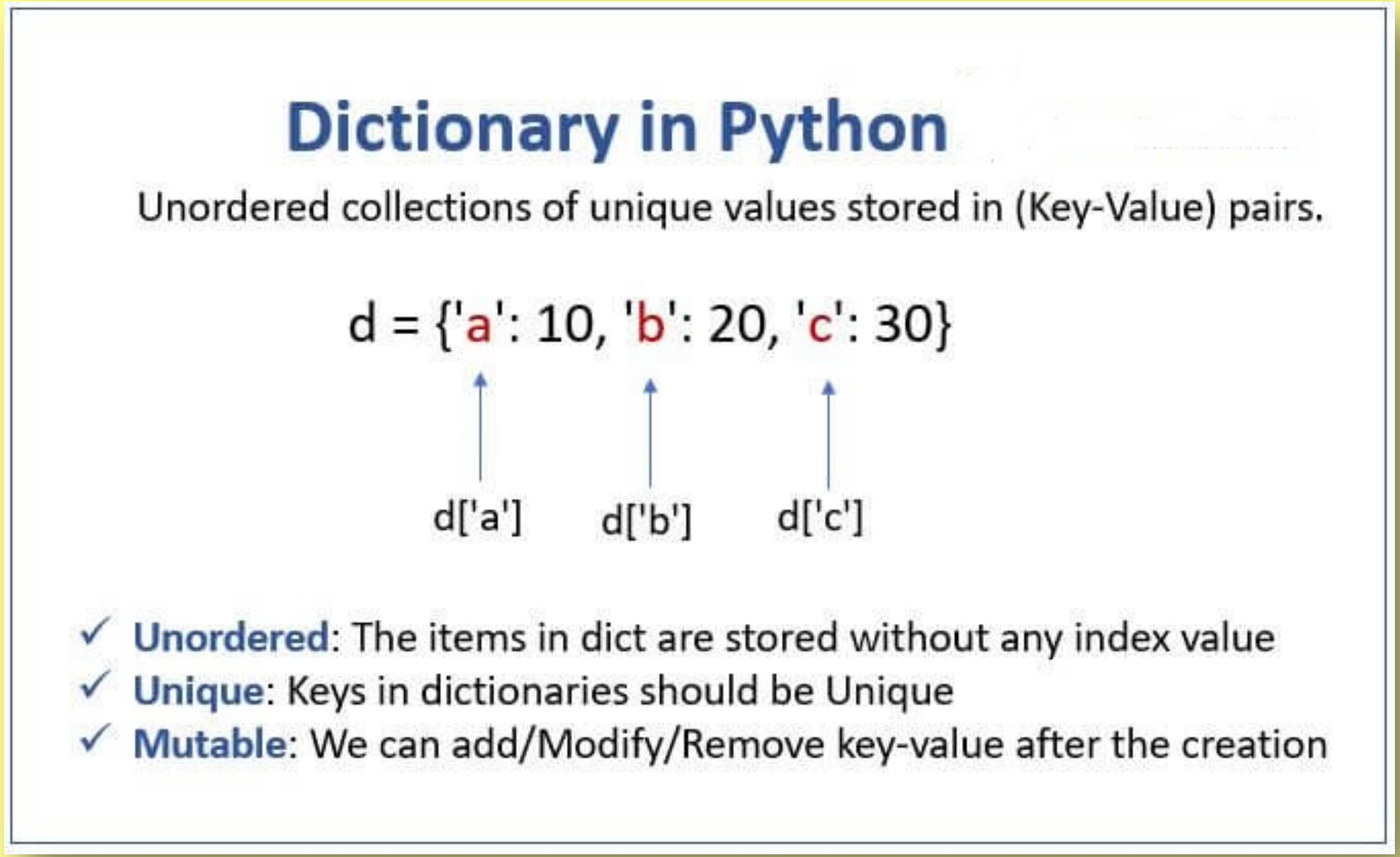
- Python dictionary represents a mapping between a key and a value. In simple terms, a Python dictionary can store pairs of keys and values.

- Each key is linked to a specific value. Once stored in a dictionary, we can later obtain the value using just the key.

- For example, we can consider the Phone lookup where it is very easy and fast to find the phone number (***value***) when we know the name (***key***) associated with it.

Dictionary Properties

❑ **Features of Python lists:**



Dictionary in Python

Unordered collections of unique values stored in (Key-Value) pairs.

```
d = {'a': 10, 'b': 20, 'c': 30}
```

- ✓ **Unordered**: The items in dict are stored without any index value
- ✓ **Unique**: Keys in dictionaries should be Unique
- ✓ **Mutable**: We can add/Modify/Remove key-value after the creation

Dictionary Properties

- **Unordered:** The items in dictionaries are stored without any index value, which is typically a range of numbers. They are stored as ***Key-Value*** pairs, and the keys are their index, which will not be in any sequence.

- **Unique:** As we have discussed above, each value has a Key; the Keys in Dictionaries should be unique. If we store any value with a Key that already exists, then the most recent value will replace the old value.

- **Mutable:** The dictionaries are collections that are changeable, which implies that we can add or remove items after the creation.

Creating a Dictionary

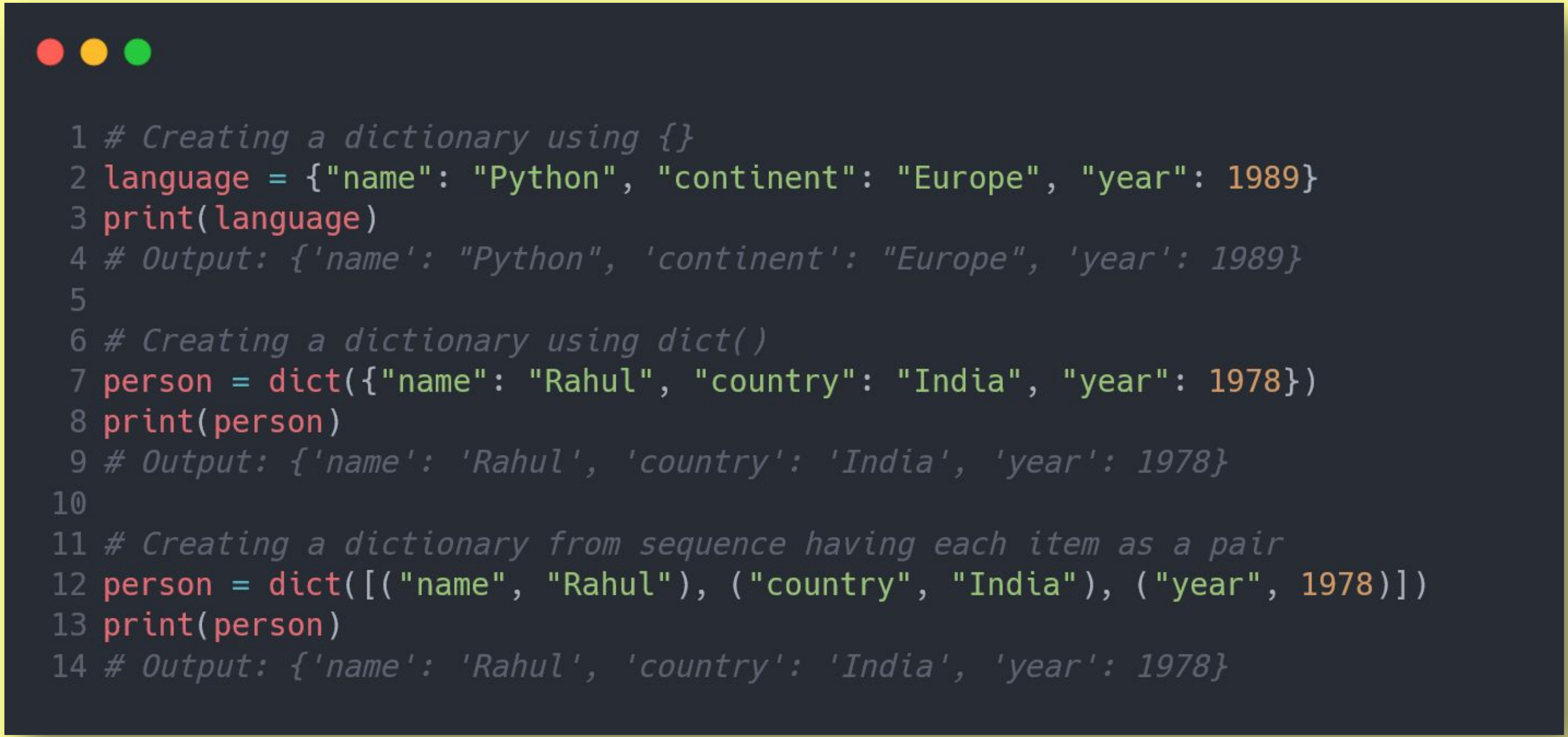
- **There are following three ways to create a dictionary in Python:**

- **Using dictionary scope:** The dictionaries are created by enclosing the comma-separated Key: Value pairs inside the “***{}***” (curly) brackets. The ***“:”*** (colon) is used to separate the key and value in a pair.

- **Using dict() constructor:** Create a dictionary by passing the comma-separated key: value pairs inside the ***dict()***.

- Using sequence having each item as a pair (***Key-Value***).

Creating a Dictionary



```python
# Creating a dictionary using {}
language = {"name": "Python", "continent": "Europe", "year": 1989}
print(language)
# Output: {'name': "Python", 'continent': "Europe", 'year': 1989}

# Creating a dictionary using dict()
person = dict({"name": "Rahul", "country": "India", "year": 1978})
print(person)
# Output: {'name': 'Rahul', 'country': 'India', 'year': 1978}

# Creating a dictionary from sequence having each item as a pair
person = dict([("name", "Rahul"), ("country", "India"), ("year", 1978)])
print(person)
# Output: {'name': 'Rahul', 'country': 'India', 'year': 1978}
```

Empty Dictionary

- When we create a dictionary without any elements inside the curly braces, then it becomes an empty dictionary.

❑ **Sample syntax:**

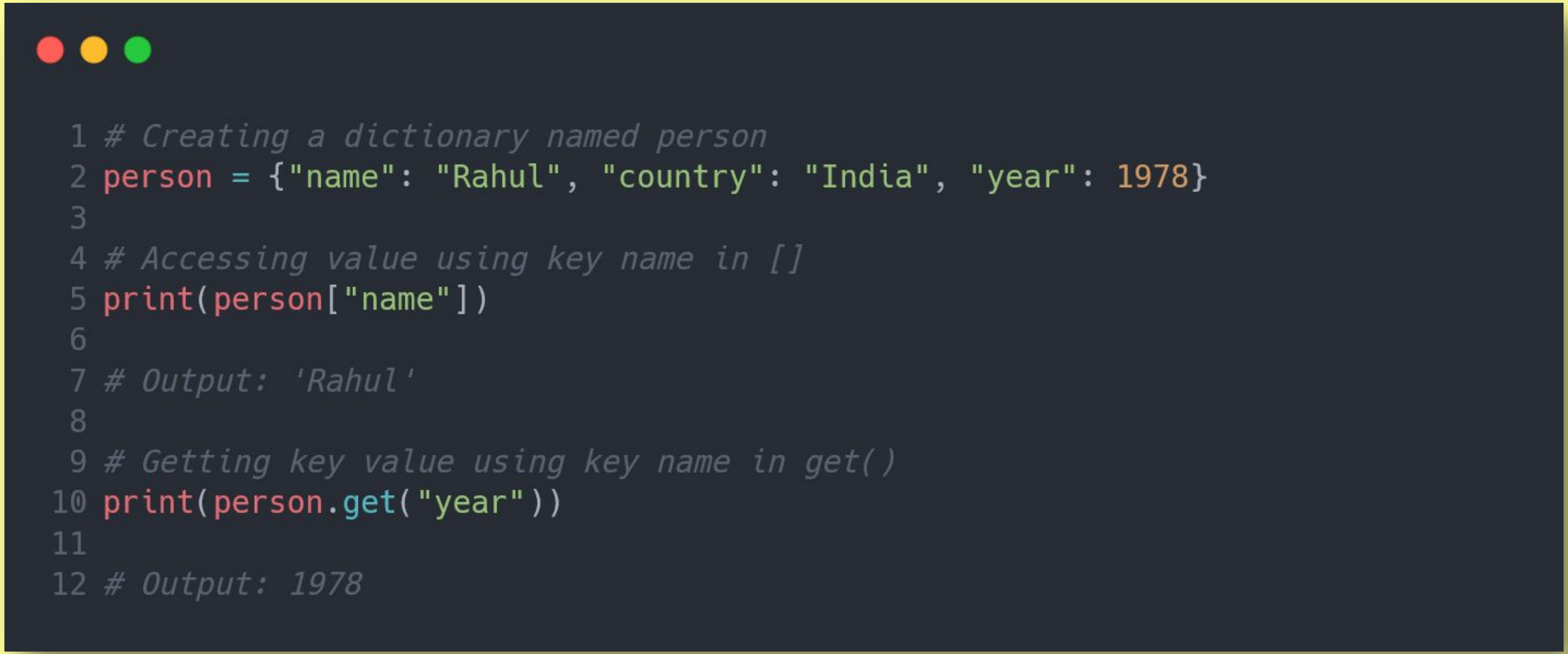
```
any_dictionary = {}
print(type(any_dictionary))

# Output class 'dict'
```

Accessing Dictionary Elements

- **There are two different ways to access the elements of a dictionary:**

  - Retrieve value using the key name inside the ***"[ ]"*** (square) brackets.
  - Retrieve value by passing key name as a parameter to the ***get()*** method of a dictionary.

Accessing Dictionary Elements



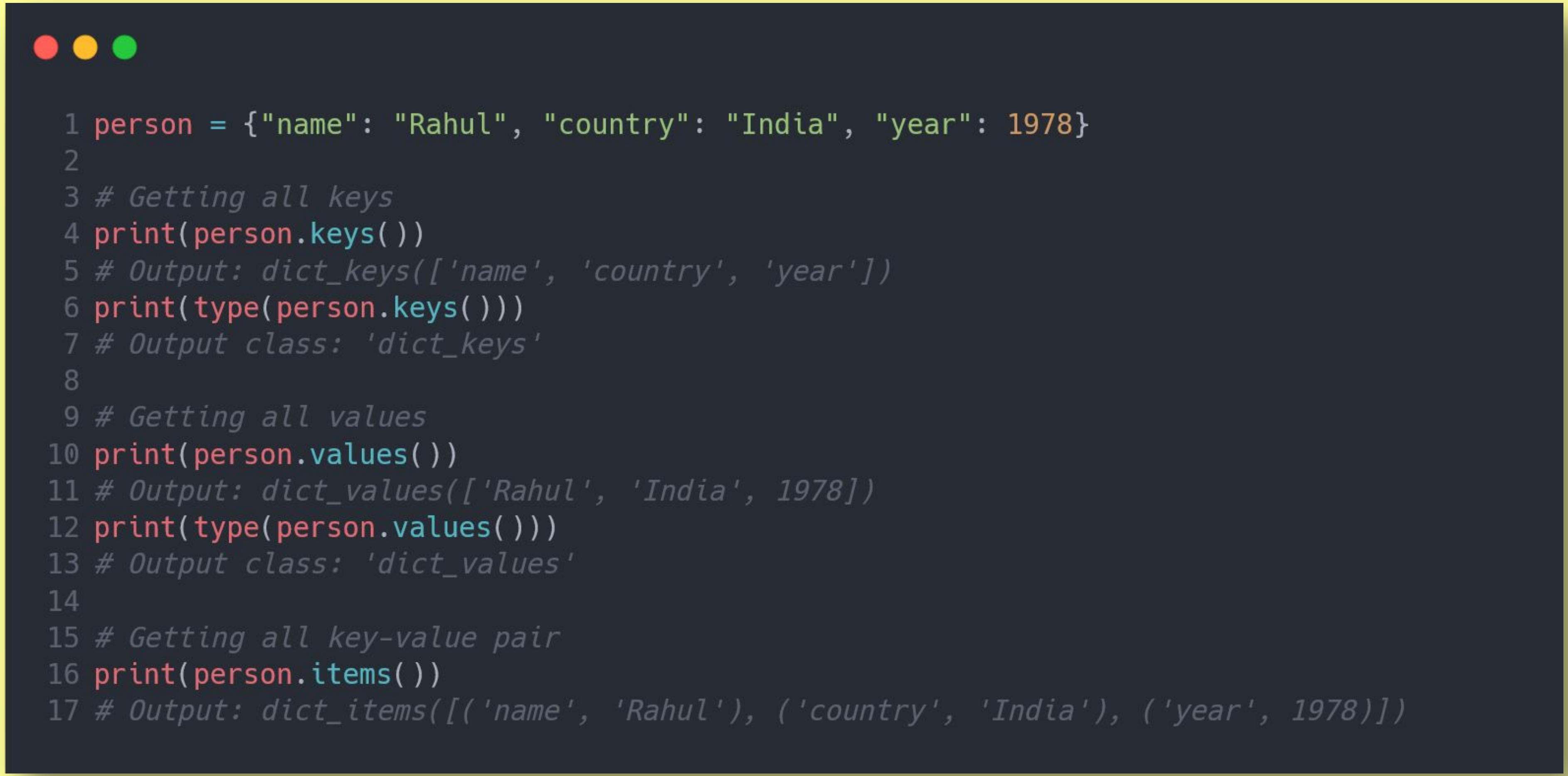
```python
# Creating a dictionary named person
person = {"name": "Rahul", "country": "India", "year": 1978}

# Accessing value using key name in []
print(person["name"])

# Output: 'Rahul'

# Getting key value using key name in get()
print(person.get("year"))

# Output: 1978
```

Accessing Keys and Values

- We can use the following in-built Python dictionary functions to retrieve all key and values at once:

Methods	Description
○ keys()	Returns the list of all keys present in the dictionary.
○ values()	Returns the list of all values present in the dictionary
○ items()	Returns all the items present in the dictionary. Each item will be inside a tuple as a key-value pair.

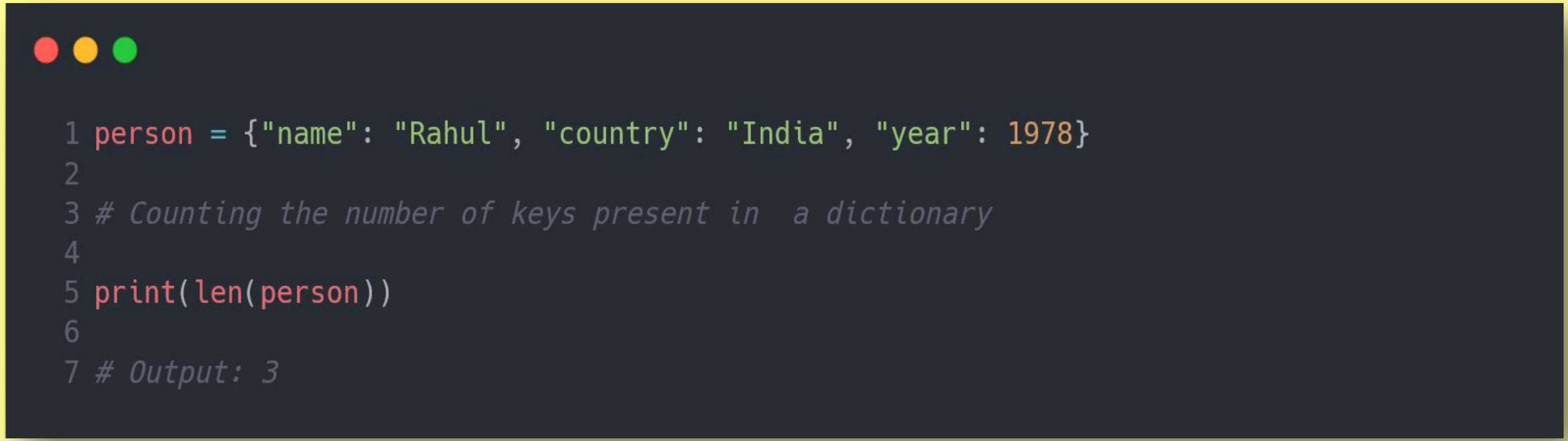
Accessing Keys and Values



```python
person = {"name": "Rahul", "country": "India", "year": 1978}

# Getting all keys
print(person.keys())
# Output: dict_keys(['name', 'country', 'year'])
print(type(person.keys()))
# Output class: 'dict_keys'

# Getting all values
print(person.values())
# Output: dict_values(['Rahul', 'India', 1978])
print(type(person.values()))
# Output class: 'dict_values'

# Getting all key-value pair
print(person.items())
# Output: dict_items([('name', 'Rahul'), ('country', 'India'), ('year', 1978)])
```

Dictionary Length

- In order to find the number of items in a dictionary, we can use the ***len()*** function.
- Let us have a look at a dictionary of personal details where we find its length.



```python
person = {"name": "Rahul", "country": "India", "year": 1978}

# Counting the number of keys present in  a dictionary

print(len(person))

# Output: 3
```

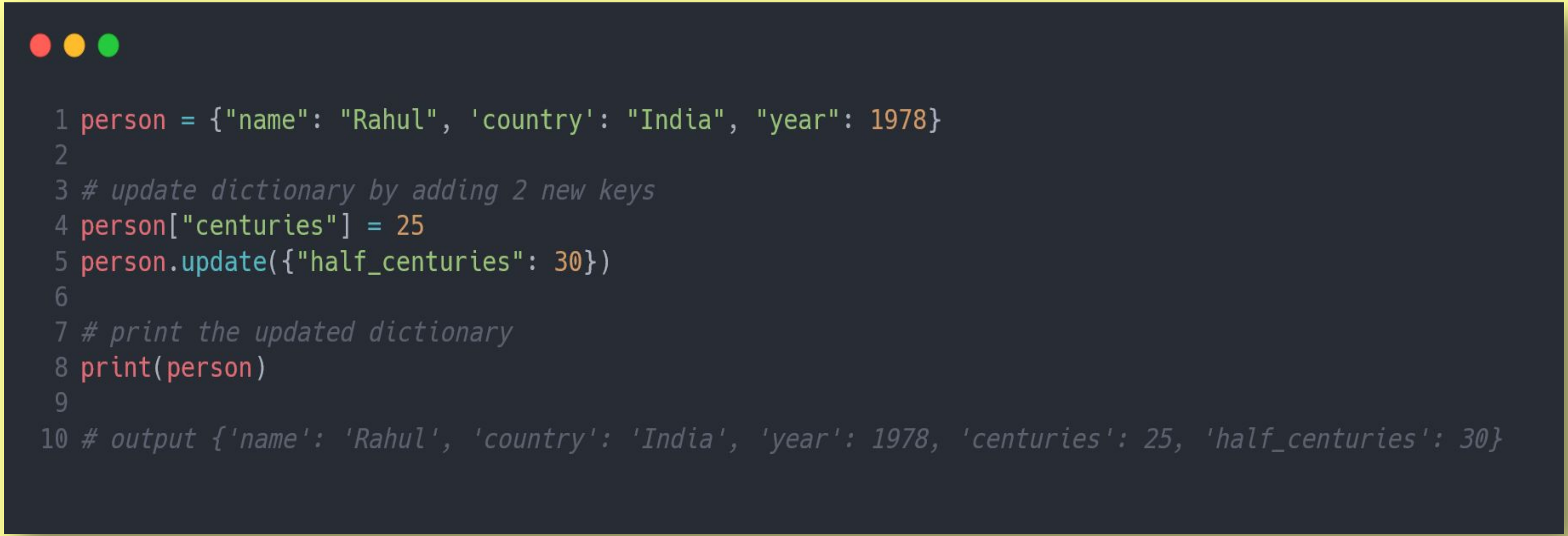
Adding Dictionary Items

❑ **We can add new items to the dictionary using the following two ways:**

- **Using key-value assignment:** Using a simple assignment statement where value can be assigned directly to the new key.

- **Using update() Method:** In this method, the item passed inside the ***update()*** method will be inserted into the dictionary. The item can be another dictionary or any iterable like a tuple of **key-value** pairs.

Adding Dictionary Items



```python
person = {"name": "Rahul", 'country': "India", "year": 1978}

# update dictionary by adding 2 new keys
person["centuries"] = 25
person.update({"half_centuries": 30})

# print the updated dictionary
print(person)

# output {'name': 'Rahul', 'country': 'India', 'year': 1978, 'centuries': 25, 'half_centuries': 30}
```

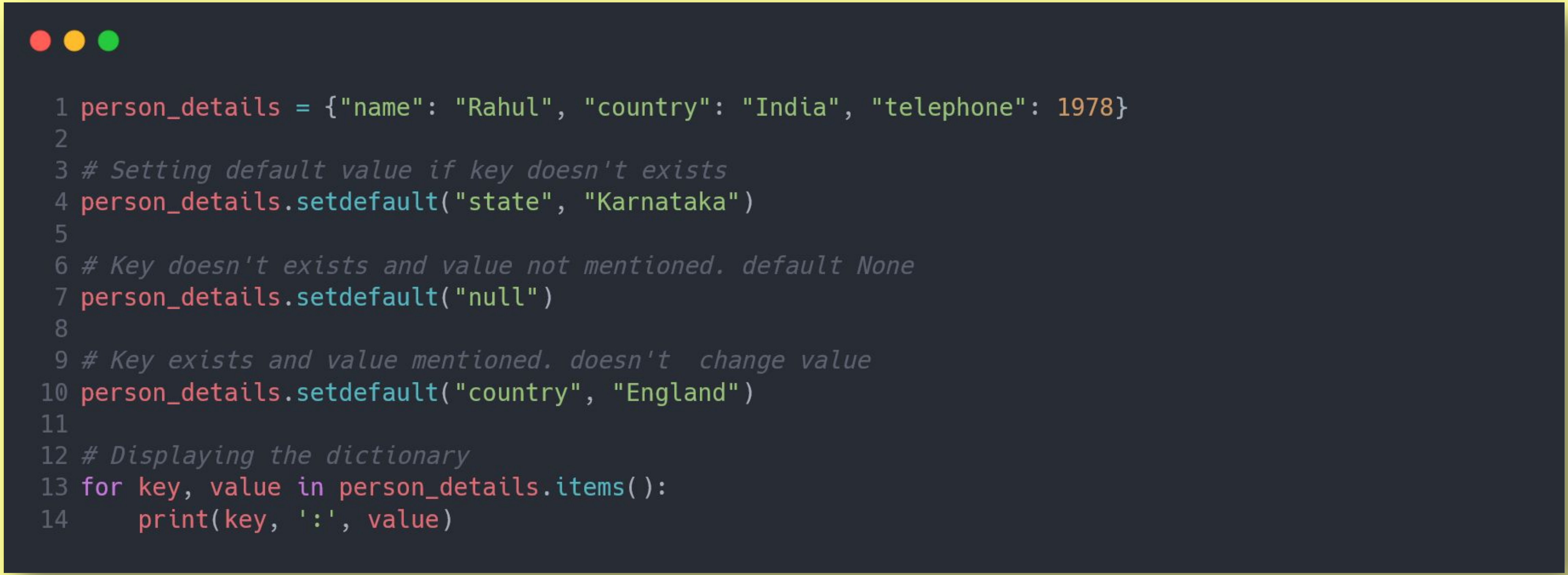
Default Key

- We can use the ***setdefault()*** method in Python where the default value can be assigned to a key in the dictionary.

- In case the key does not exist already, then the key will be inserted into the dictionary, and the value becomes the default value, and None will be inserted if a value is not mentioned.

- In case the key exists, then it will return the value of a key.

Default Key



```python
person_details = {"name": "Rahul", "country": "India", "telephone": 1978}

# Setting default value if key doesn't exists
person_details.setdefault("state", "Karnataka")

# Key doesn't exists and value not mentioned. default None
person_details.setdefault("null")

# Key exists and value mentioned. doesn't  change value
person_details.setdefault("country", "England")

# Displaying the dictionary
for key, value in person_details.items():
    print(key, ':', value)
```

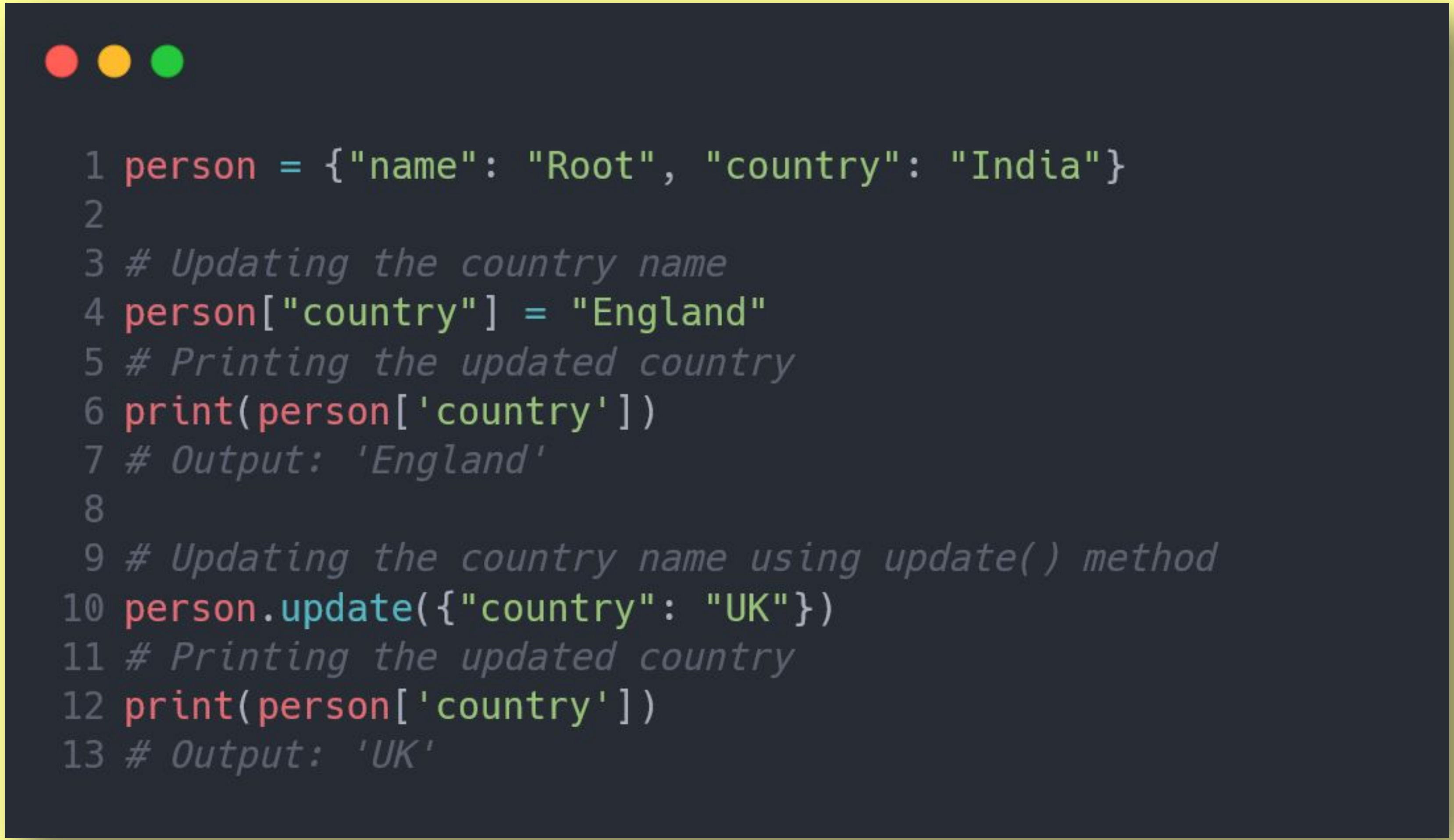
Modifying Dictionary Values

- ❑ **We can modify the values of the existing dictionary keys using the following two ways:**

- **Using key name:** We can directly assign new values by using its key name. The key name will be the existing one and we can mention the new value.

- **Using update() method:** We can use the update method by passing the ***key-value*** pair to change the value. Here the key name will be the existing one, and the value to be updated will be new.

Modifying Dictionary Values



```python
person = {"name": "Root", "country": "India"}

# Updating the country name
person["country"] = "England"
# Printing the updated country
print(person['country'])
# Output: 'England'

# Updating the country name using update() method
person.update({"country": "UK"})
# Printing the updated country
print(person['country'])
# Output: 'UK'
```