3. Formatting Output with `print()`

The `print()` function is versatile and offers multiple ways to format and present data.

a. Printing Multiple Items:

You can print multiple items by separating them with commas.

```python
name = "Alice"
age = 30
print("Name:", name, "Age:", age)
```

b. String Concatenation:

Strings can be joined together using the `+` operator.

python Copy code

```python
print("Hello " + name + "! You are " + str(age) + " years old.")
```

Note: The `str()` function converts the integer `age` to a string for concatenation.

c. Using f-Strings (Python 3.6+):

This modern method allows for embedded expressions within string literals.

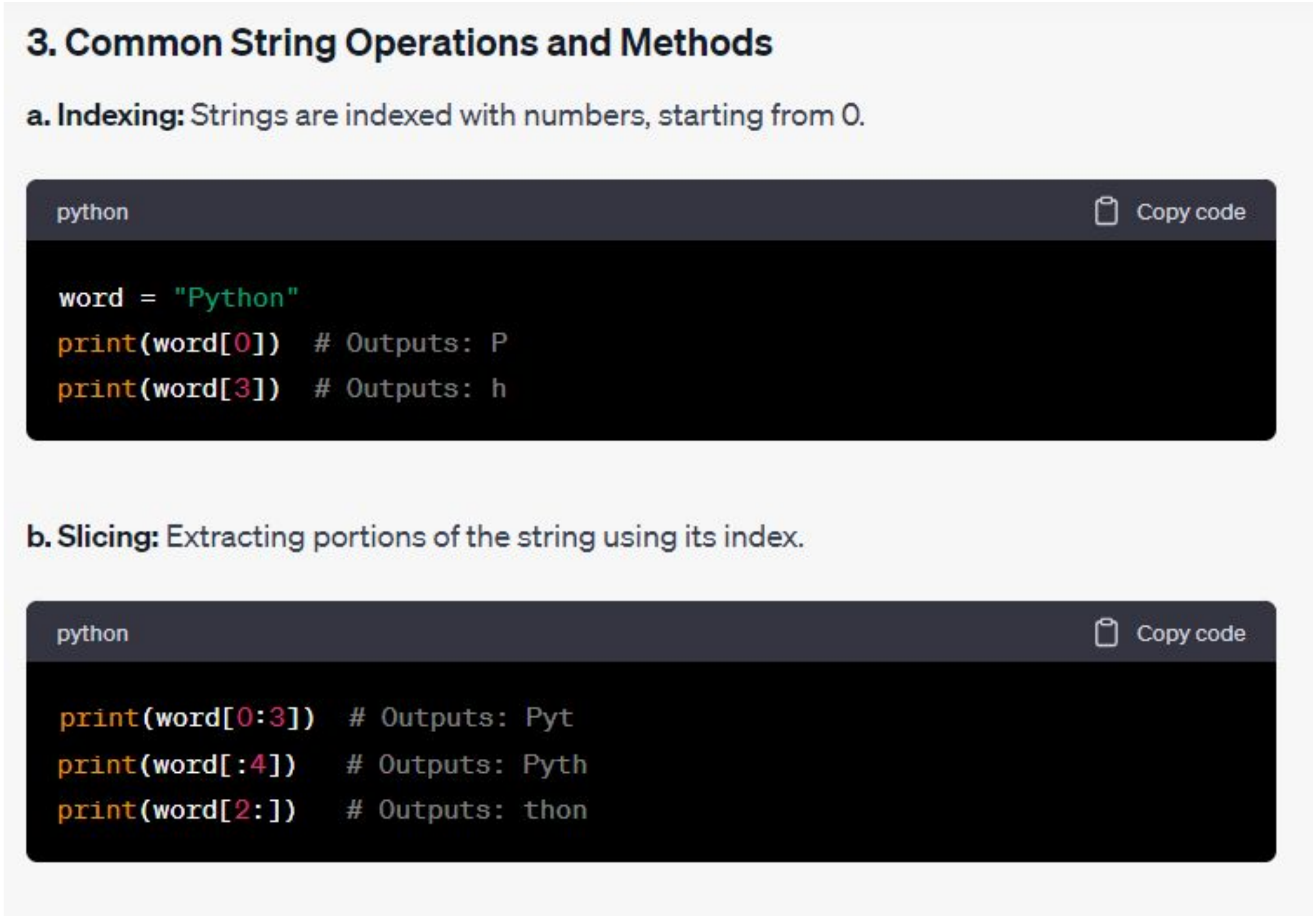
python Copy code

```python
print(f"Hello {name}! You are {age} years old.")
```

d. Using `print()` with Special Characters:

- `\n` represents a newline.
- `\t` represents a tab.

```python
print("Hello\nWorld!")  # Prints "Hello" and "World!" on separate lines
print("Hello\tWorld!")  # Prints "Hello" and "World!" separated by a tab
```



3. Common String Operations and Methods

a. Indexing: Strings are indexed with numbers, starting from 0.

```python
word = "Python"
print(word[0])  # Outputs: P
print(word[3])  # Outputs: h
```

b. Slicing: Extracting portions of the string using its index.

```python
print(word[0:3])  # Outputs: Pyt
print(word[:4])   # Outputs: Pyth
print(word[2:])   # Outputs: thon
```

c. Concatenation: Combining two or more strings.

python Copy code

```python
str1 = "Hello"
str2 = "World"
print(str1 + ", " + str2 + "!")  # Outputs: Hello, World!
```

d. Length: Determine the number of characters in a string using the `` `len()` `` function.

python Copy code

```python
print(len("Hello"))  # Outputs: 5
```

e. String methods: Strings come with a set of built-in methods that allow you to perform various operations.

python Copy code

```python
text = "hello world"
print(text.upper())        # Outputs: HELLO WORLD
print(text.capitalize())   # Outputs: Hello world
print(text.replace("world", "Python"))  # Outputs: hello Python
```

4. String Formatting

Strings can be formatted using a couple of different methods in Python.

a. Using the `format()` method:

python Copy code

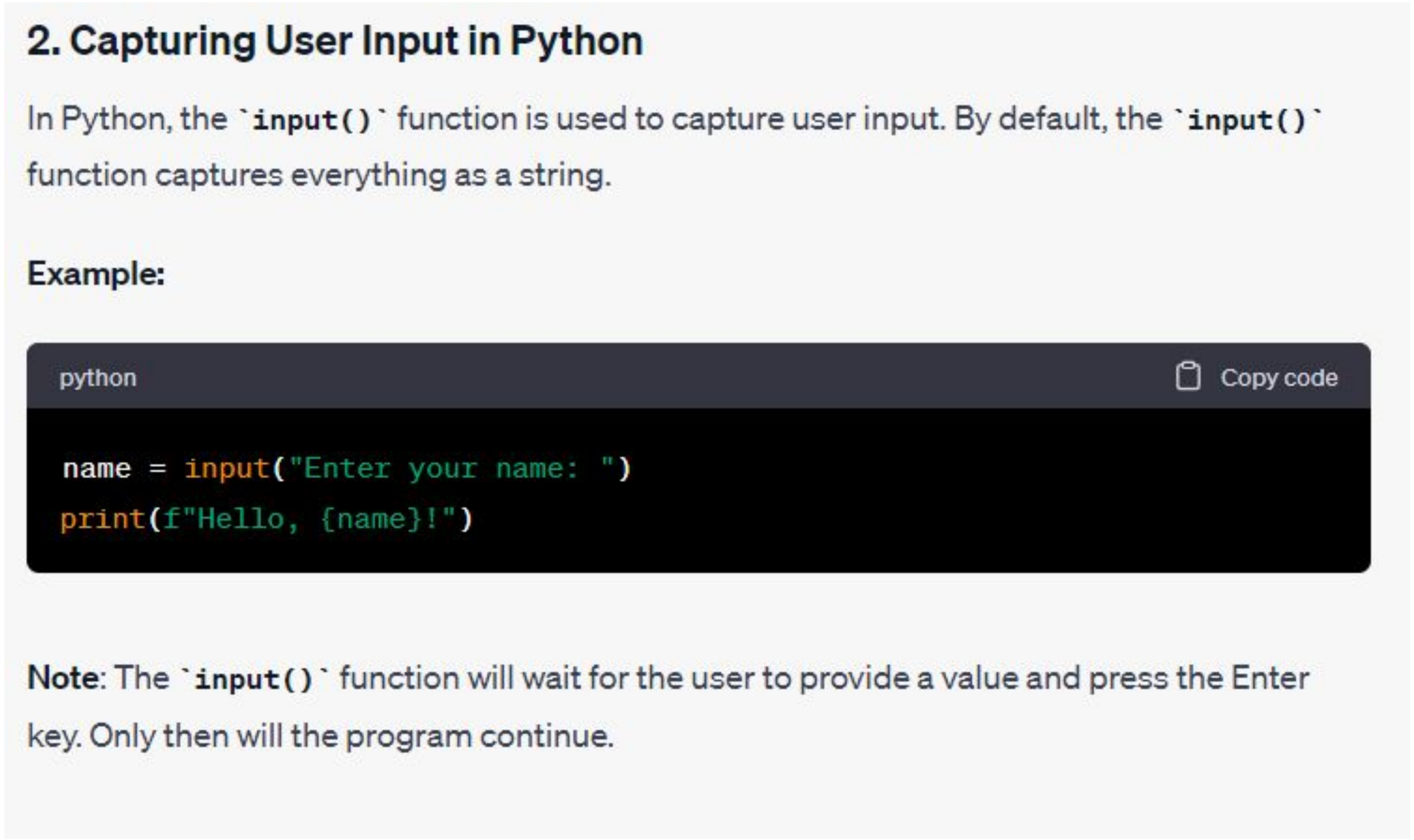
```python
name = "Alice"
print("Hello, {}!".format(name))  # Outputs: Hello, Alice!
```

b. f-Strings (Python 3.6+):

This is a more modern approach and can be more concise and readable.

python Copy code

```python
name = "Bob"
print(f"Hello, {name}!")  # Outputs: Hello, Bob!
```



2. Capturing User Input in Python

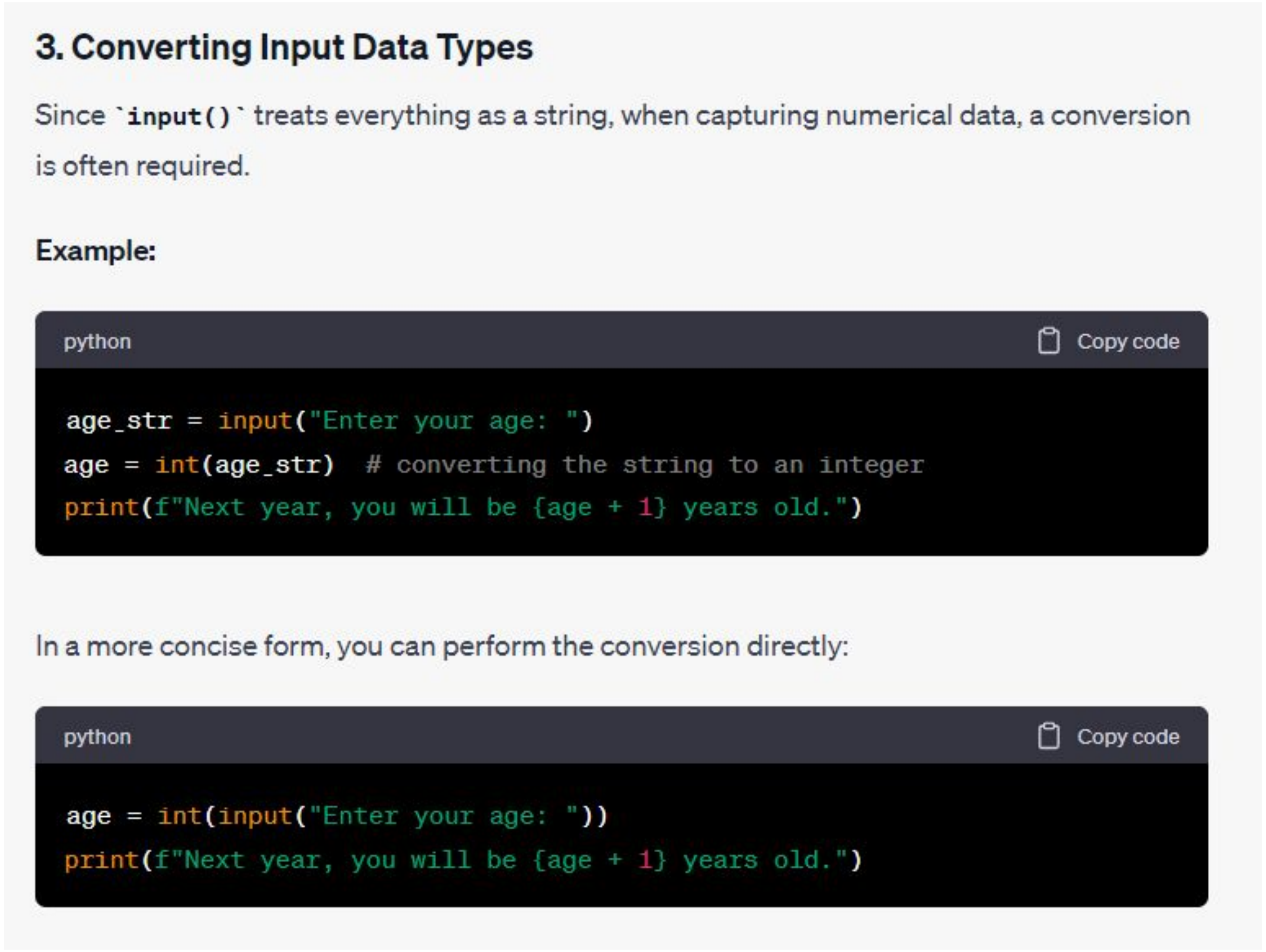
In Python, the `input()` function is used to capture user input. By default, the `input()` function captures everything as a string.

Example:

python   Copy code

```python
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

Note: The `input()` function will wait for the user to provide a value and press the Enter key. Only then will the program continue.



3. Converting Input Data Types

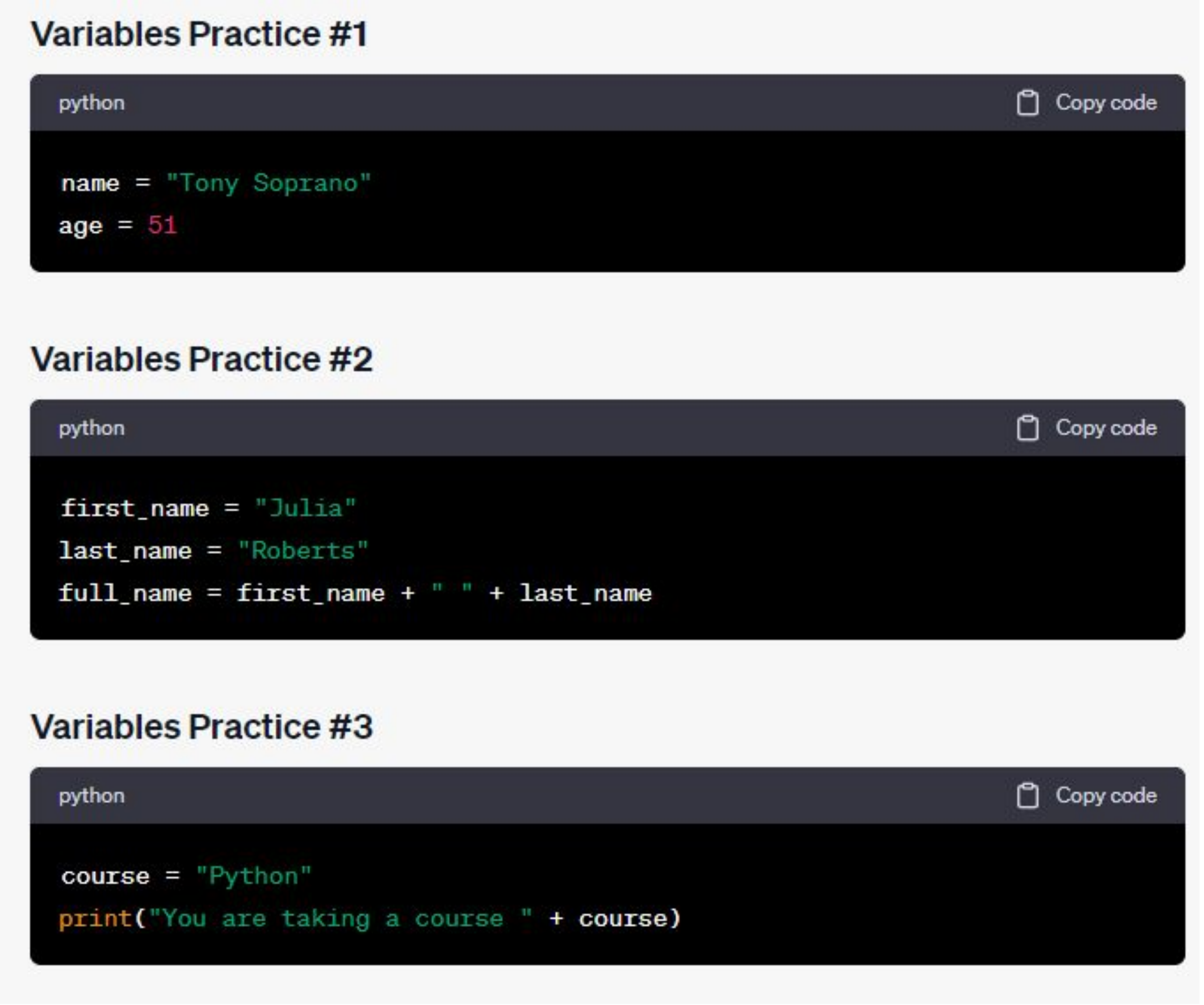
Since `input()` treats everything as a string, when capturing numerical data, a conversion is often required.

Example:

```python
age_str = input("Enter your age: ")
age = int(age_str)  # converting the string to an integer
print(f"Next year, you will be {age + 1} years old.")
```

In a more concise form, you can perform the conversion directly:

```python
age = int(input("Enter your age: "))
print(f"Next year, you will be {age + 1} years old.")
```



Variables Practice #1

```python
name = "Tony Soprano"
age = 51
```

Variables Practice #2

```python
first_name = "Julia"
last_name = "Roberts"
full_name = first_name + " " + last_name
```

Variables Practice #3

```python
course = "Python"
print("You are taking a course " + course)
```

Working with Numbers

Integers Practice:

```python
int_num = 5
print(type(int_num))
```

Floats Practice:

```python
decimal_num = 3.14
print(type(decimal_num))
```

Data Types Practice:

```python
num1 = 7.5
num2 = 2.5
result = num1 + num2
print(type(result))
```

1. **Addition**:

python Copy code

```python
a = 5
b = 3
result = a + b
print(result)  # Output: 8
```

1. **Multiplication**:

python Copy code

```python
result = a * b
print(result)  # Output: 15
```

1. **Division**:

python Copy code

```python
result = a / b
print(result)  # Output: 1.6666666666666667
```

1. **Modulo** (remainder after division):

python Copy code

```python
result = a % b
print(result)  # Output: 2
```

1. **Powers** (raising a number to an exponent):

python Copy code

```python
result = a ** b
print(result)  # Output: 125
```

1. **Get the max and min of a number**:

python Copy code

```python
print(max(a, b))  # Output: 5
print(min(a, b))  # Output: 3
```

1. **Round a number**:

python Copy code

```python
number = 1.6666666666666667
print(round(number))  # Output: 2
```

1. **Absolute value**:

python Copy code

```python
c = -7
print(abs(c))  # Output: 7
```

1. **Order of operations**:

python Copy code

```python
result = a + b * 2  # Here, multiplication has precedence over addition
print(result)  # Output: 11
```

To do more, you need to import special math libraries from Python:

You can use the `math` module to access more advanced mathematical functions.

Floor method (rounds down):

```python
import math

number = 1.6666666666666667
print(math.floor(number))  # Output: 1
```

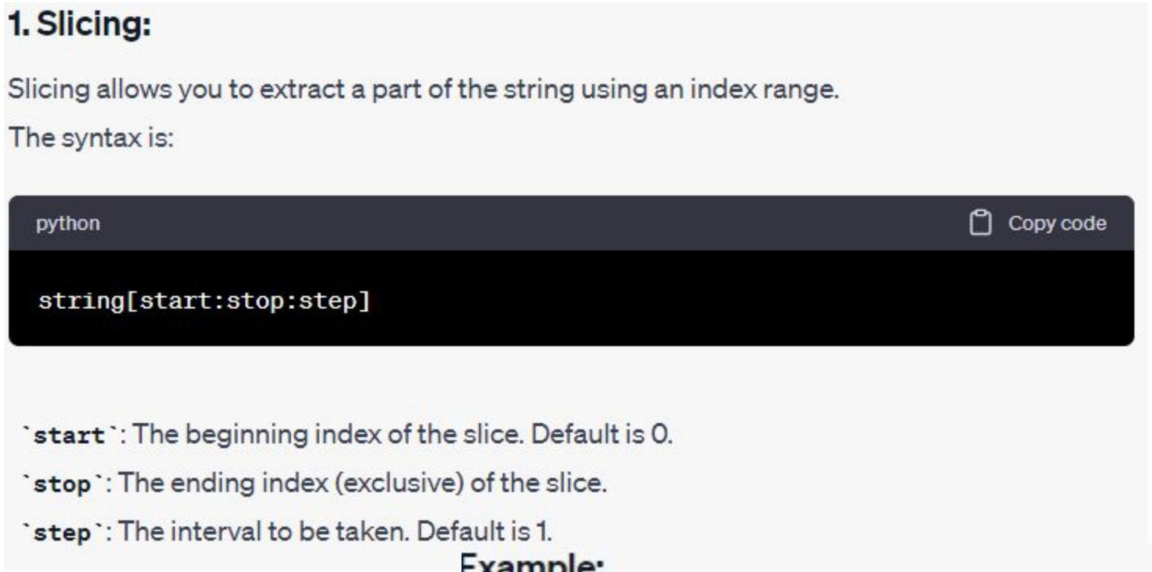
Ceil method (rounds up):

```python
print(math.ceil(number))  # Output: 2
```

Sqrt method (square root):

```python
d = 16
print(math.sqrt(d))  # Output: 4.0
```

The primary method of dealing with substrings in Python is through slicing and some built-in methods.



1. Slicing:

Slicing allows you to extract a part of the string using an index range.

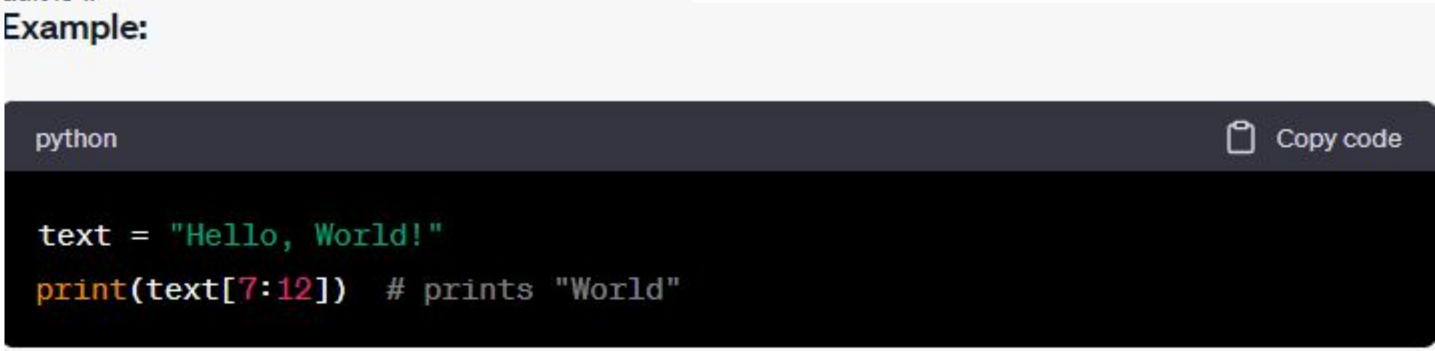
The syntax is:

```python
string[start:stop:step]
```

`start`: The beginning index of the slice. Default is 0.

`stop`: The ending index (exclusive) of the slice.

`step`: The interval to be taken. Default is 1.



Example:

```python
text = "Hello, World!"
print(text[7:12])  # prints "World"
```

2. Built-in methods:

Python has a variety of built-in methods to work with substrings.

a. `str.find()`:

This method returns the lowest index of the substring if found in the given string. If it's not found, it returns -1.

Example:

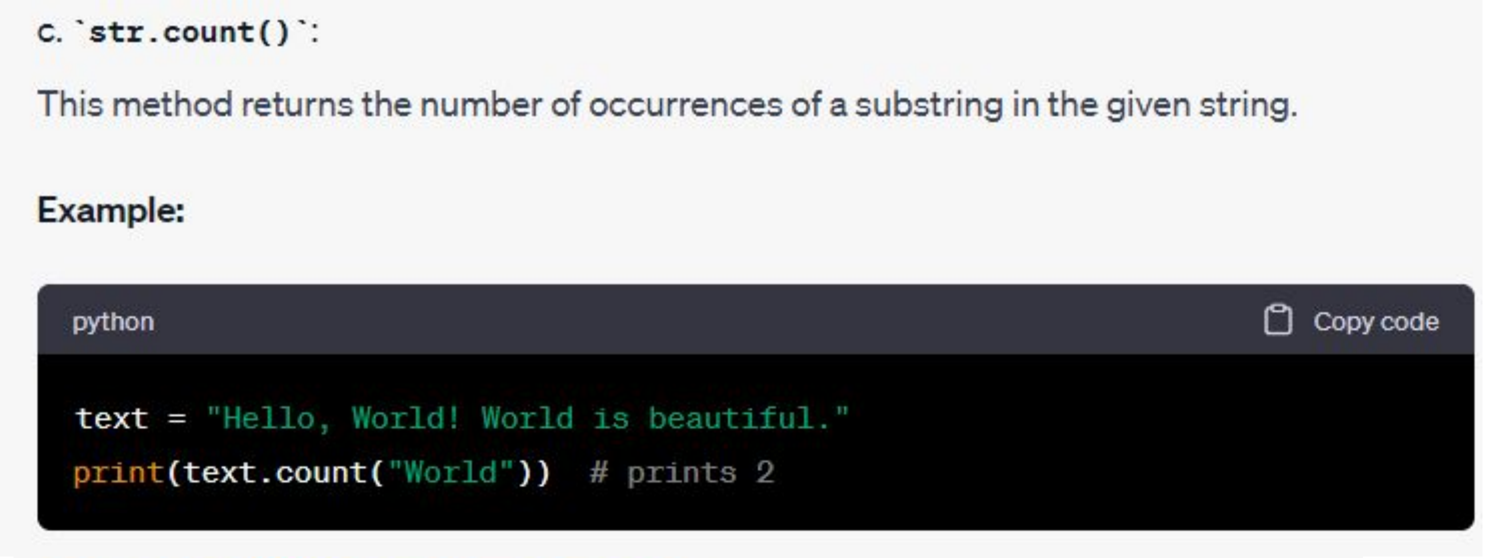
```python
text = "Hello, World!"
print(text.find("World"))  # prints 7
print(text.find("Earth"))  # prints -1
```

b. `str.index()`:

Similar to `find()`, but raises a `ValueError` if the substring is not found.

Example:

```python
text = "Hello, World!"
print(text.index("World"))  # prints 7
# print(text.index("Earth"))  # raises ValueError
```



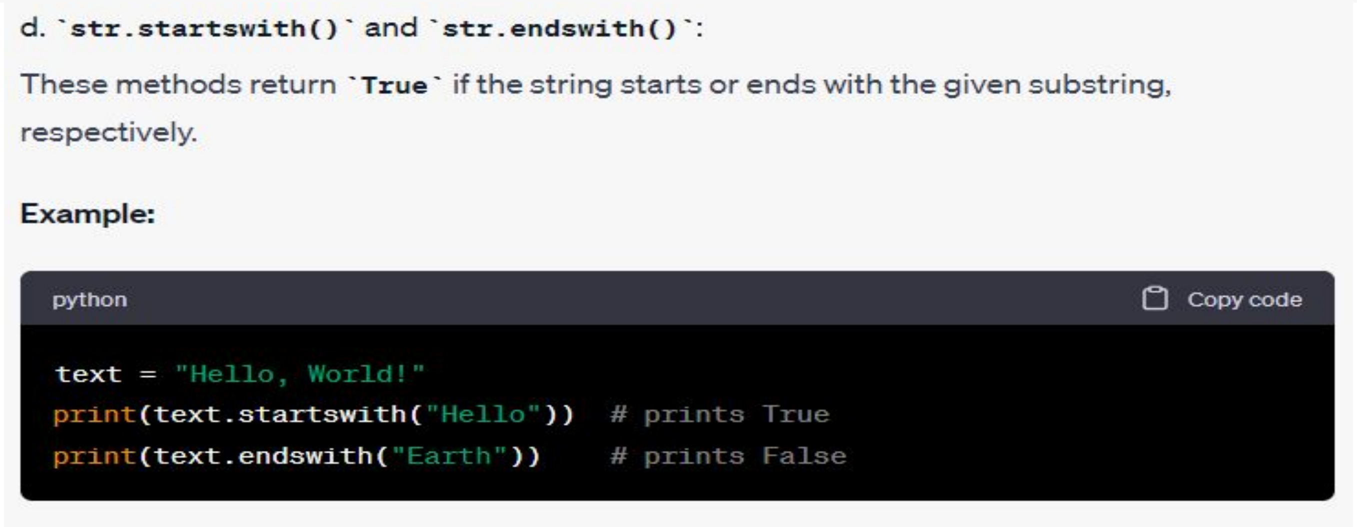
c. `str.count()`:

This method returns the number of occurrences of a substring in the given string.

Example:

python Copy code

```python
text = "Hello, World! World is beautiful."
print(text.count("World"))  # prints 2
```



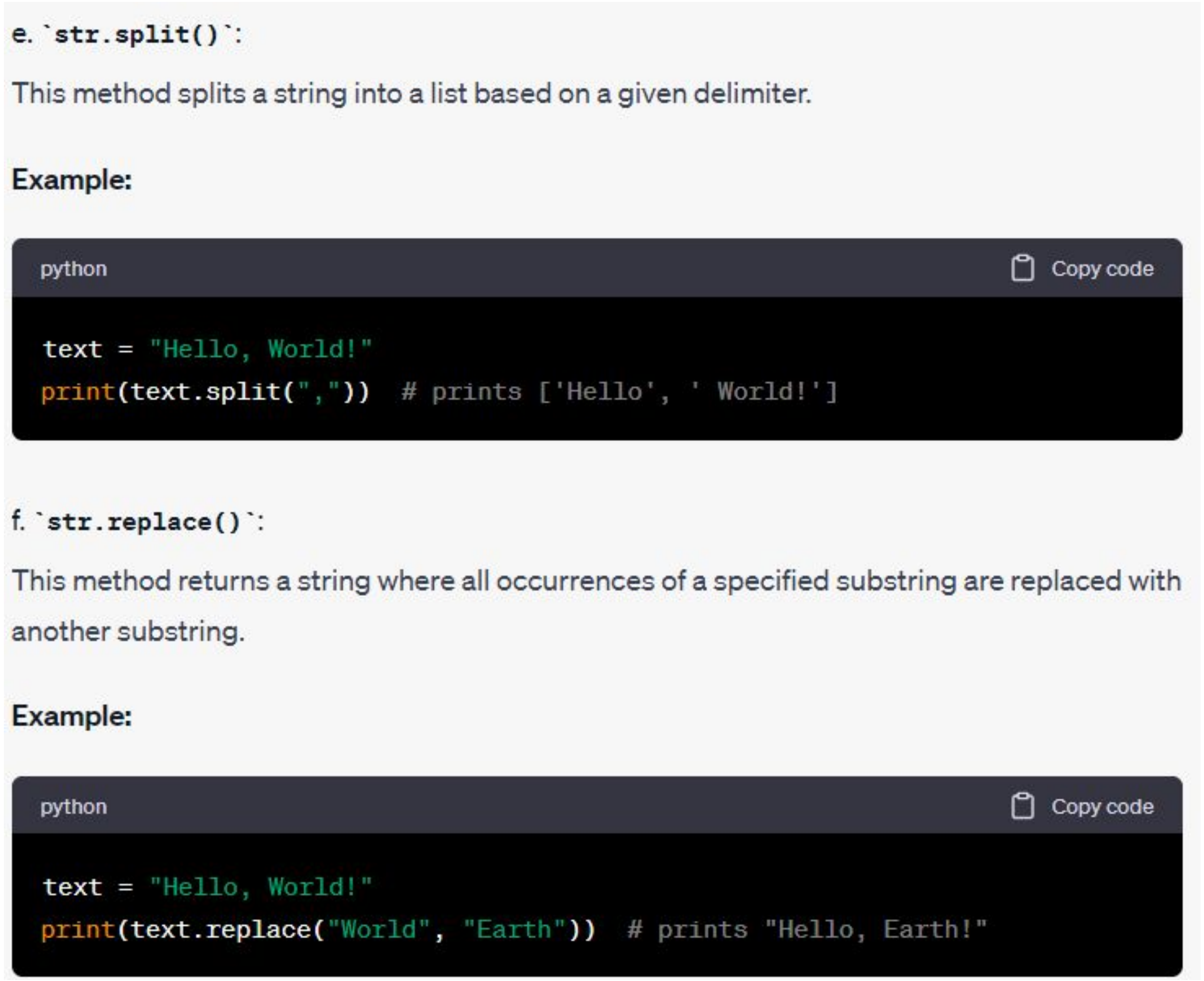
d. `str.startswith()` and `str.endswith()`:

These methods return `True` if the string starts or ends with the given substring, respectively.

Example:

python Copy code

```python
text = "Hello, World!"
print(text.startswith("Hello"))  # prints True
print(text.endswith("Earth"))    # prints False
```



e. `str.split()`:

This method splits a string into a list based on a given delimiter.

Example:

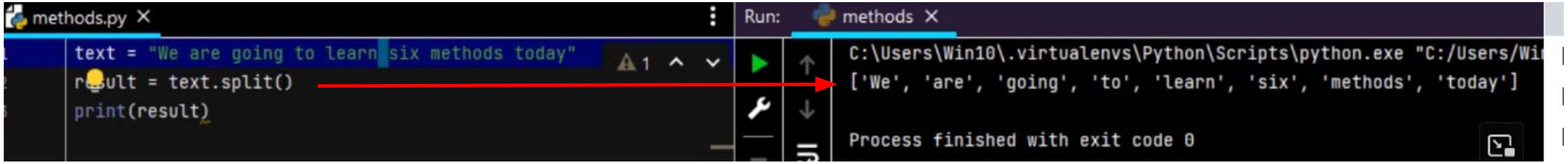
```python
text = "Hello, World!"
print(text.split(","))  # prints ['Hello', ' World!']
```

f. `str.replace()`:

This method returns a string where all occurrences of a specified substring are replaced with another substring.

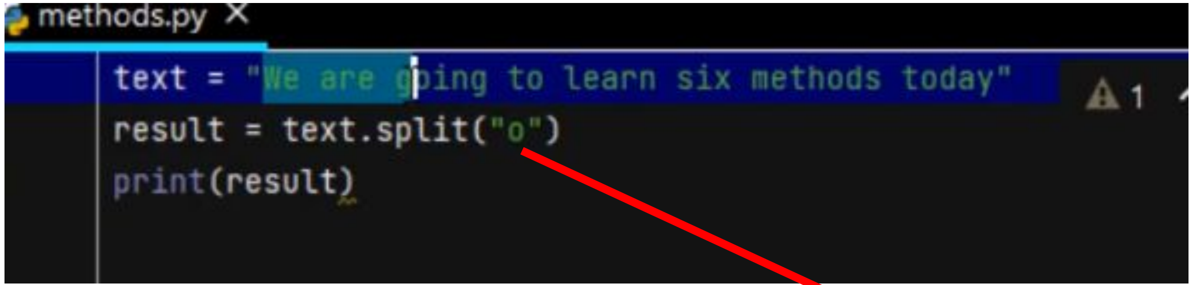
Example:

```python
text = "Hello, World!"
print(text.replace("World", "Earth"))  # prints "Hello, Earth!"
```

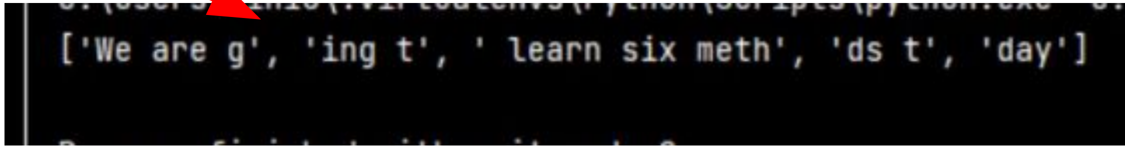


```
text = "We are going to learn six methods today"
result = text.split()
print(result)
```

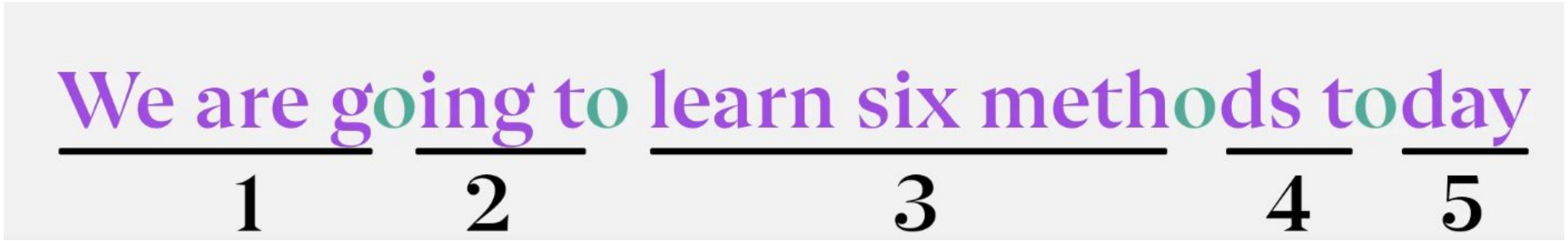
```
C:\Users\Win10\.virtualenvs\Python\Scripts\python.exe "C:/Users/Wi
['We', 'are', 'going', 'to', 'learn', 'six', 'methods', 'today']

Process finished with exit code 0
```

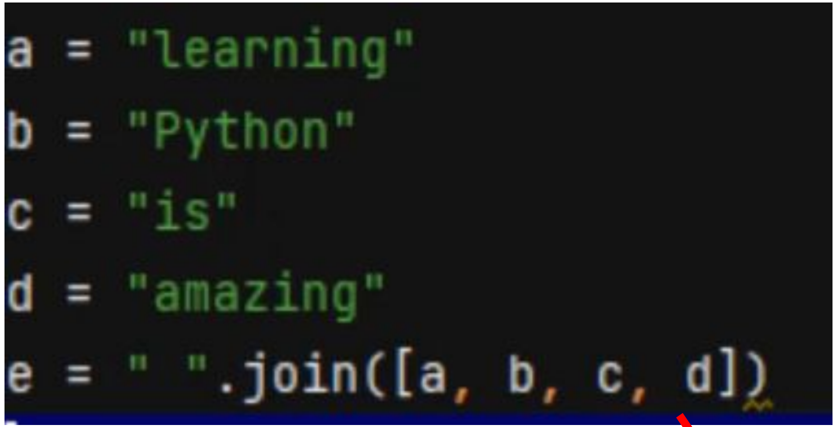


```
text = "We are going to learn six methods today"
result = text.split("o")
print(result)
```

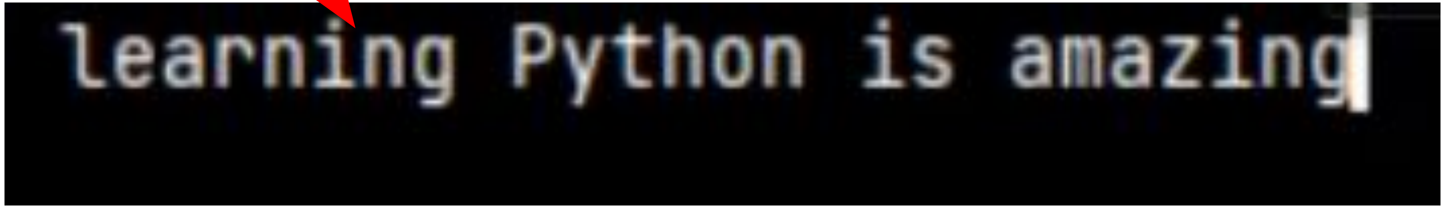


```
['We are g', 'ing t', ' learn six meth', 'ds t', 'day']
```

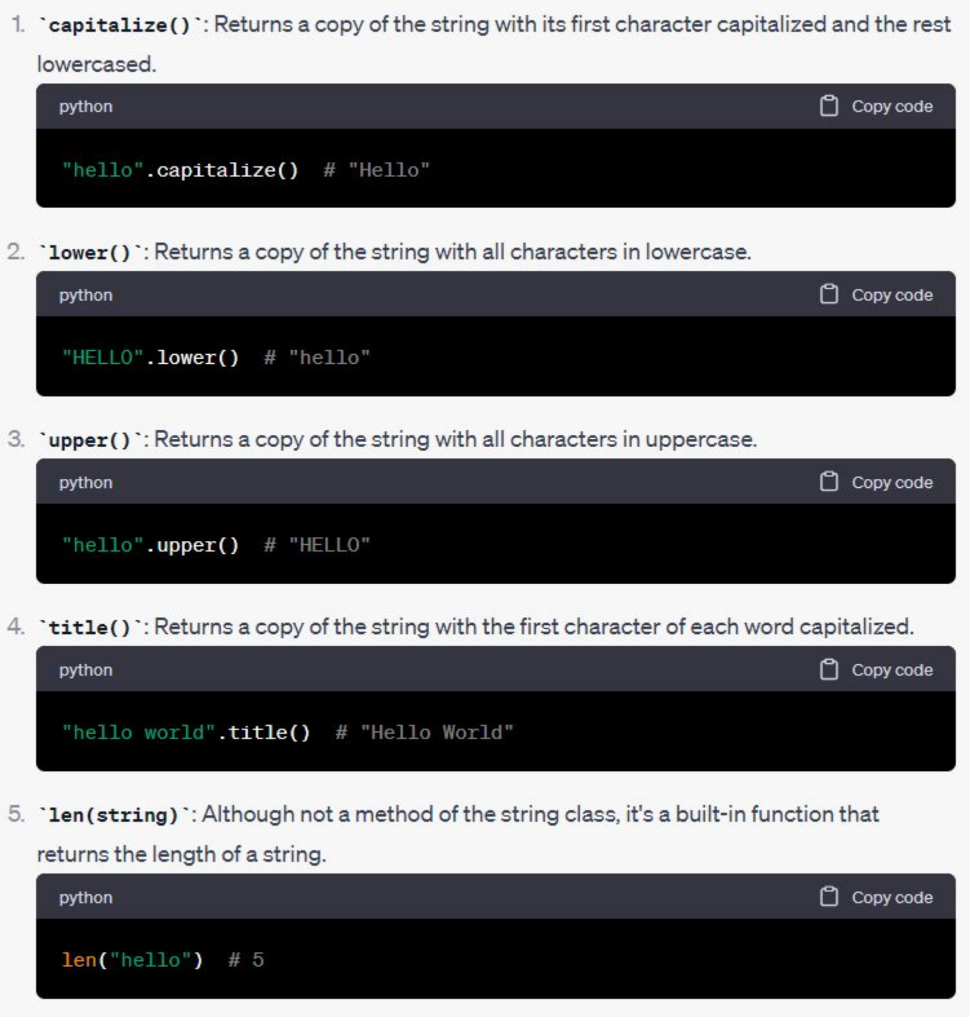




```python
a = "learning"
b = "Python"
c = "is"
d = "amazing"
e = " ".join([a, b, c, d])
```



```
learning Python is amazing
```



1. `capitalize()`: Returns a copy of the string with its first character capitalized and the rest lowercased.

```python
"hello".capitalize()  # "Hello"
```

2. `lower()`: Returns a copy of the string with all characters in lowercase.

```python
"HELLO".lower()  # "hello"
```

3. `upper()`: Returns a copy of the string with all characters in uppercase.

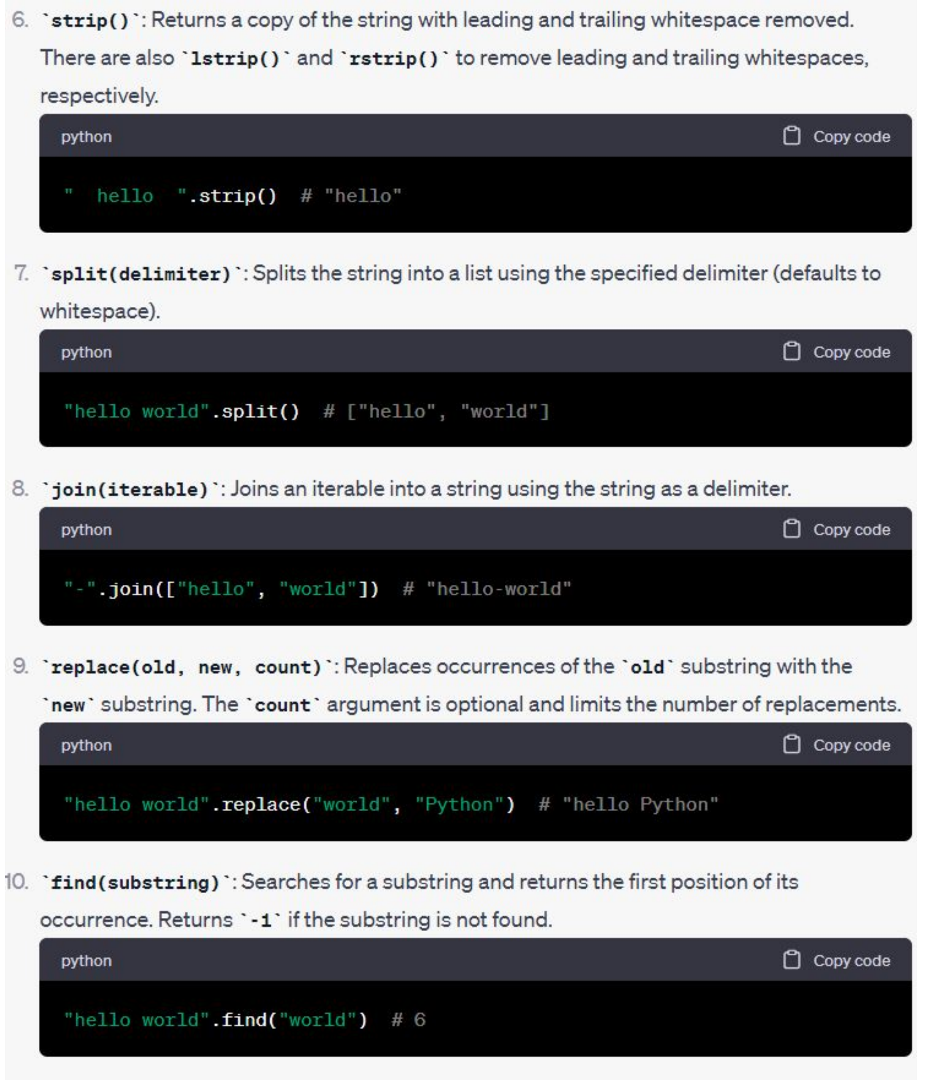
```python
"hello".upper()  # "HELLO"
```

4. `title()`: Returns a copy of the string with the first character of each word capitalized.

```python
"hello world".title()  # "Hello World"
```

5. `len(string)`: Although not a method of the string class, it's a built-in function that returns the length of a string.

```python
len("hello")  # 5
```



6. `strip()`: Returns a copy of the string with leading and trailing whitespace removed. There are also `lstrip()` and `rstrip()` to remove leading and trailing whitespaces, respectively.

```python
"  hello  ".strip()  # "hello"
```

7. `split(delimiter)`: Splits the string into a list using the specified delimiter (defaults to whitespace).

```python
"hello world".split()  # ["hello", "world"]
```

8. `join(iterable)`: Joins an iterable into a string using the string as a delimiter.

```python
"-".join(["hello", "world"])  # "hello-world"
```

9. `replace(old, new, count)`: Replaces occurrences of the `old` substring with the `new` substring. The `count` argument is optional and limits the number of replacements.

```python
"hello world".replace("world", "Python")  # "hello Python"
```

10. `find(substring)`: Searches for a substring and returns the first position of its occurrence. Returns `-1` if the substring is not found.

```python
"hello world".find("world")  # 6
```

11. `index(substring)`: Similar to `find()`, but raises a `ValueError` if the substring is not found.

```python
"hello world".index("world")  # 6
```

12. `count(substring)`: Returns the number of occurrences of a substring in the string.

```python
"hello hello hello".count("hello")  # 3
```

13. `startswith(prefix)` and `endswith(suffix)`: Check if the string starts or ends with the given prefix or suffix, respectively.

```python
"hello".startswith("he")  # True
"hello".endswith("lo")    # True
```

14. `isalpha()`, `isdigit()`, `isspace()`, `islower()`, `isupper()`, `istitle()`: These are some of the methods to check the characteristics of the string, like if it contains only alphabetic characters, digits, etc.
15. `center(width, char)`: Returns a centered string of length `width` padded with `char`.

```python
"hello".center(9, "-")  # "--hello--"
```

16. `zfill(width)`: Returns a string left-padded with zeroes to fill a width.

```python
"42".zfill(5)  # "00042"
```

17. `encode(encoding="UTF-8",errors="strict")`: Returns an encoded version of the string. Default encoding is "UTF-8".
18. `expandtabs(tabsize=8)`: Returns a copy of the string where all tab characters are replaced by the appropriate number of spaces.

In Python, strings are objects of the `str` class, and they have certain inherent properties that define their behavior and characteristics:

Immutability: One of the most fundamental properties of strings in Python is their immutability. Once a string is created, its contents cannot be altered. Any operation that appears to modify a string will actually create and return a new string.

```python
s = "hello"
# s[0] = 'y'  # This will raise an error because strings are immutable.
```

Ordered Sequence: Strings are ordered sequences of characters. This means the characters in a string have a definite order, and each character can be accessed by its index.

```python
s = "hello"
print(s[0])  # prints 'h'
```

Iterable: Strings are iterable, meaning you can loop over each character in the string using a for loop or other iteration mechanisms.

```python
for char in "hello":
    print(char)
```

Length: A string has a definite length, which can be determined using the built-in `len()` function.

```python
s = "hello"
print(len(s))  # prints 5
```

5. **Unicode Representations**: Strings in Python 3 are sequences of Unicode characters, which means they can store a wide range of characters, including those from various languages and scripts. For example:

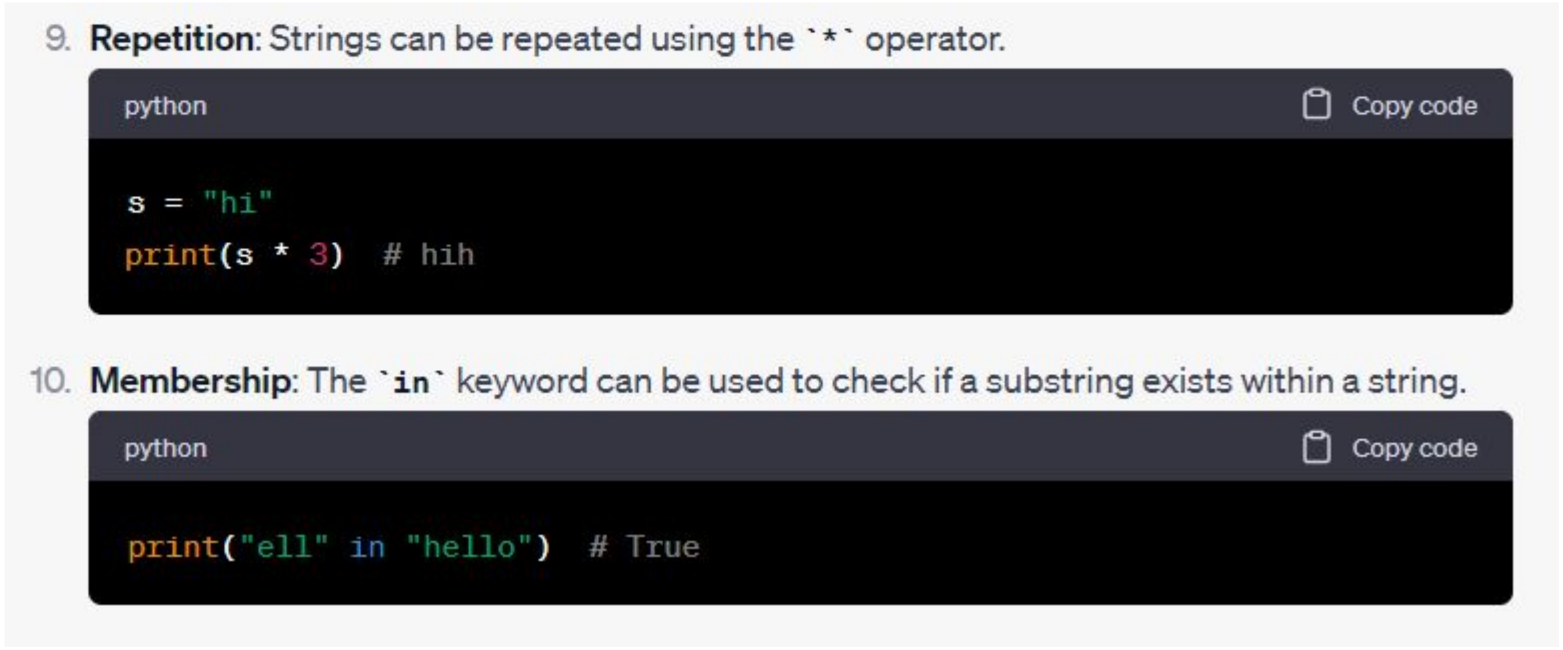
```python
s = "こんにちは"
print(s)  # prints the Japanese word for "Hello"
```

6. **Escape Sequences**: Strings can contain escape sequences, which are character combinations that start with a backslash (`\`). These sequences have special meanings. For instance, `\n` represents a newline, and `\t` represents a tab.

```python
s = "Hello\nWorld"
print(s)
# prints:
# Hello
# World
```

7. **String Literals**: Strings can be represented in multiple ways using string literals:
   - Single quotes: `'Hello'`
   - Double quotes: `"Hello"`
   - Triple single quotes (multiline strings): `'''Hello\nWorld'''`
   - Triple double quotes (multiline strings): `"""Hello\nWorld"""`
8. **Concatenation**: Strings can be concatenated using the `+` operator.

```python
s1 = "Hello"
s2 = "World"
print(s1 + s2)  # HelloWorld
```

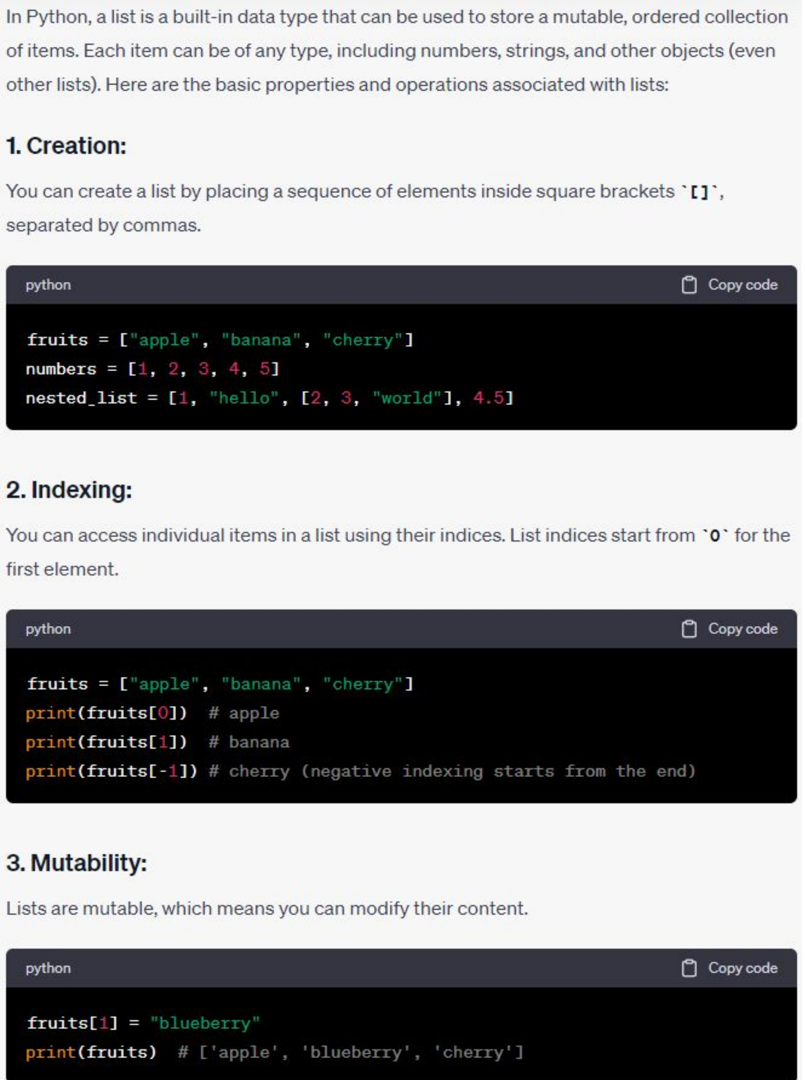


9. **Repetition**: Strings can be repeated using the `*` operator.

```python
s = "hi"
print(s * 3)  # hih
```

10. **Membership**: The `in` keyword can be used to check if a substring exists within a string.

```python
print("ell" in "hello")  # True
```



In Python, a list is a built-in data type that can be used to store a mutable, ordered collection of items. Each item can be of any type, including numbers, strings, and other objects (even other lists). Here are the basic properties and operations associated with lists:

1. Creation:

You can create a list by placing a sequence of elements inside square brackets `[]`, separated by commas.

```python
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
nested_list = [1, "hello", [2, 3, "world"], 4.5]
```

2. Indexing:

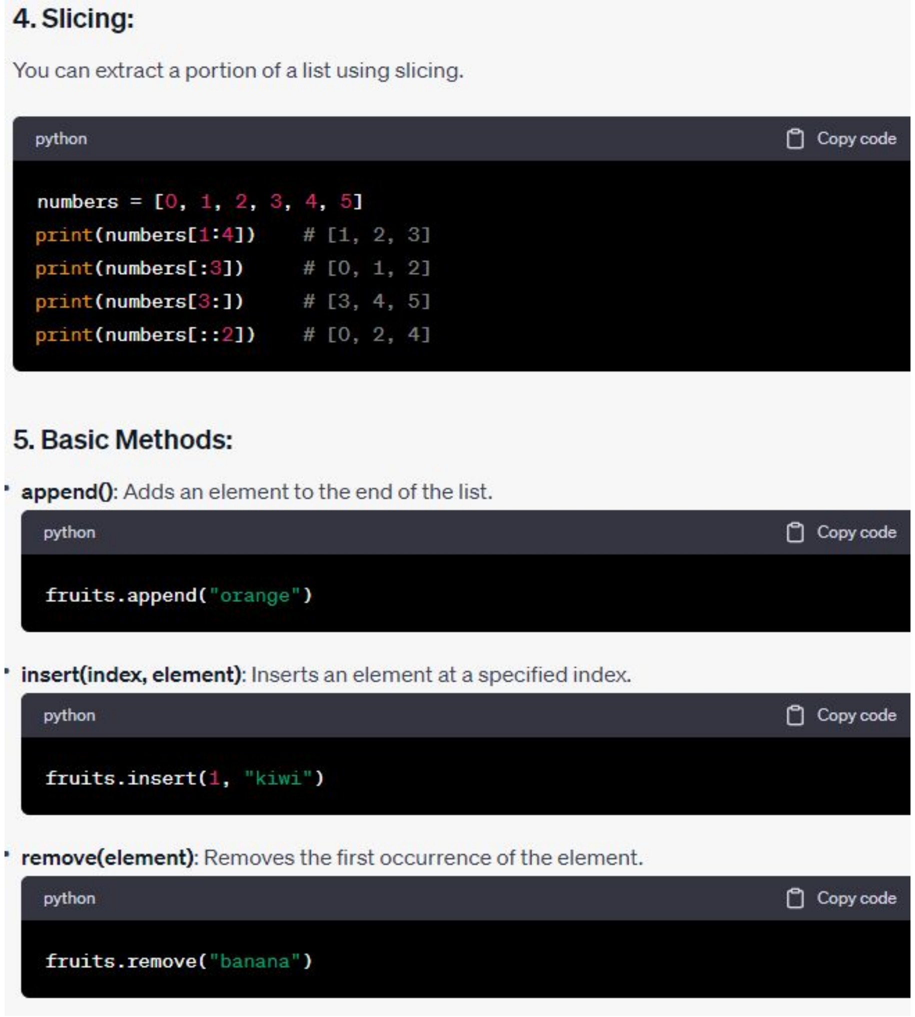
You can access individual items in a list using their indices. List indices start from `0` for the first element.

```python
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # apple
print(fruits[1])  # banana
print(fruits[-1]) # cherry (negative indexing starts from the end)
```

3. Mutability:

Lists are mutable, which means you can modify their content.

```python
fruits[1] = "blueberry"
print(fruits)  # ['apple', 'blueberry', 'cherry']
```



4. Slicing:

You can extract a portion of a list using slicing.

```python
numbers = [0, 1, 2, 3, 4, 5]
print(numbers[1:4])    # [1, 2, 3]
print(numbers[:3])     # [0, 1, 2]
print(numbers[3:])     # [3, 4, 5]
print(numbers[::2])    # [0, 2, 4]
```

5. Basic Methods:

- **append()**: Adds an element to the end of the list.

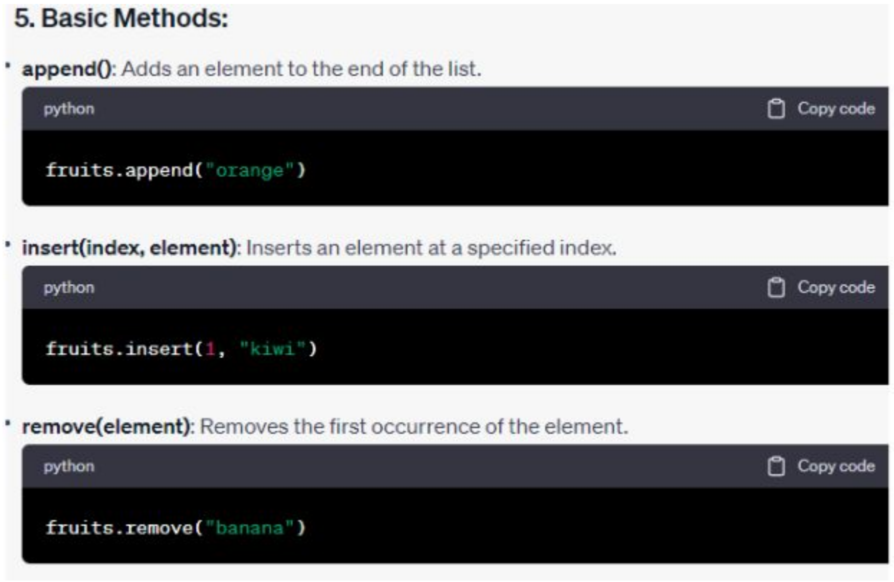
```python
fruits.append("orange")
```

- **insert(index, element)**: Inserts an element at a specified index.

```python
fruits.insert(1, "kiwi")
```

- **remove(element)**: Removes the first occurrence of the element.

```python
fruits.remove("banana")
```



5. Basic Methods:

- **append()**: Adds an element to the end of the list.

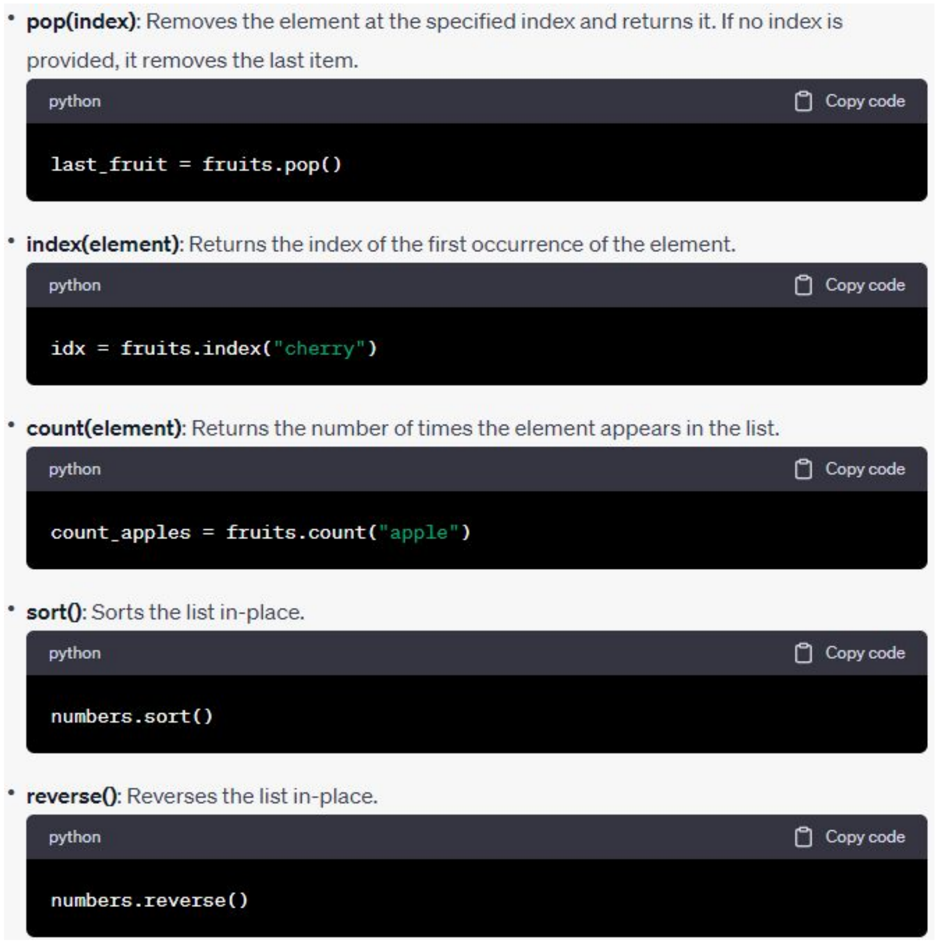
```python
fruits.append("orange")
```

- **insert(index, element)**: Inserts an element at a specified index.

```python
fruits.insert(1, "kiwi")
```

- **remove(element)**: Removes the first occurrence of the element.

```python
fruits.remove("banana")
```



- **pop(index)**: Removes the element at the specified index and returns it. If no index is provided, it removes the last item.

```python
last_fruit = fruits.pop()
```

- **index(element)**: Returns the index of the first occurrence of the element.

```python
idx = fruits.index("cherry")
```

- **count(element)**: Returns the number of times the element appears in the list.

```python
count_apples = fruits.count("apple")
```

- **sort()**: Sorts the list in-place.

```python
numbers.sort()
```

- **reverse()**: Reverses the list in-place.

```python
numbers.reverse()
```

6. Iteration:

You can iterate over a list using a `for` loop.

```python
for fruit in fruits:
    print(fruit)
```

7. Length:

The `len()` function returns the number of items in the list.

```python
length = len(fruits)
```

8. Nested Lists:

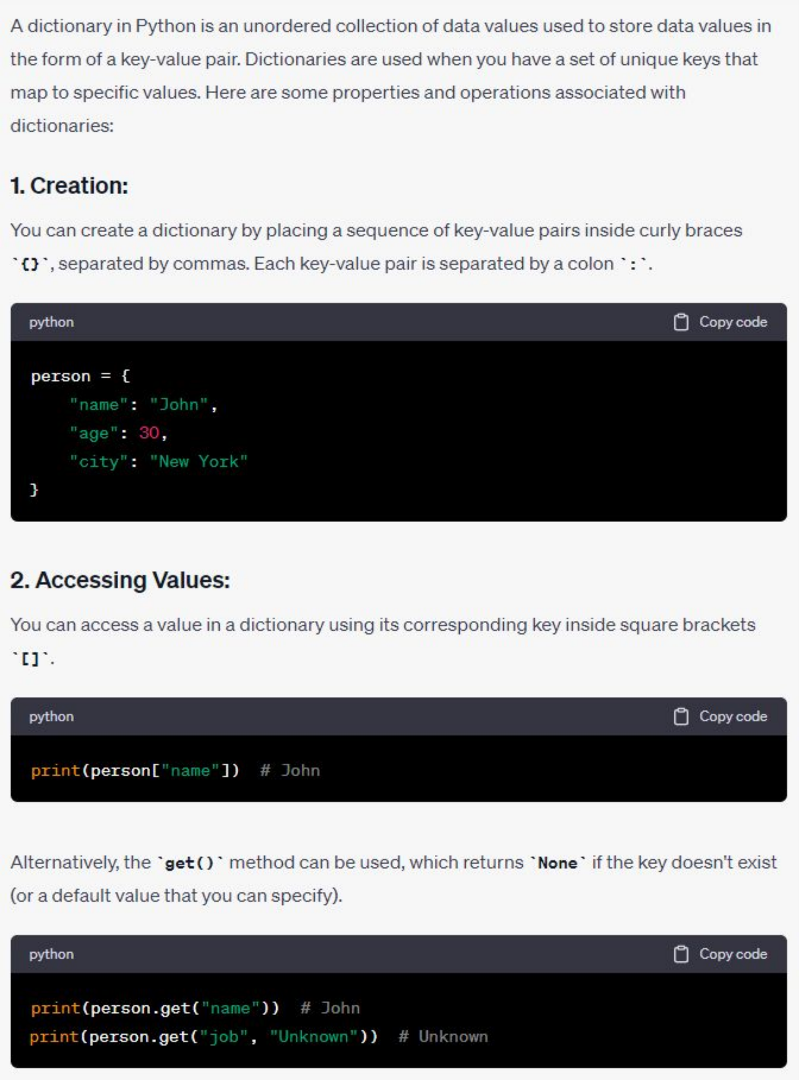
Lists can contain other lists as items, which is useful for creating matrix-like structures or trees.

```python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

9. List Comprehension:

Python also supports a concise way to create lists using list comprehension.

```python
squared_numbers = [x**2 for x in numbers if x % 2 == 0]  # squares of even
```



A dictionary in Python is an unordered collection of data values used to store data values in the form of a key-value pair. Dictionaries are used when you have a set of unique keys that map to specific values. Here are some properties and operations associated with dictionaries:

1. Creation:

You can create a dictionary by placing a sequence of key-value pairs inside curly braces `{}`, separated by commas. Each key-value pair is separated by a colon `:`.

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

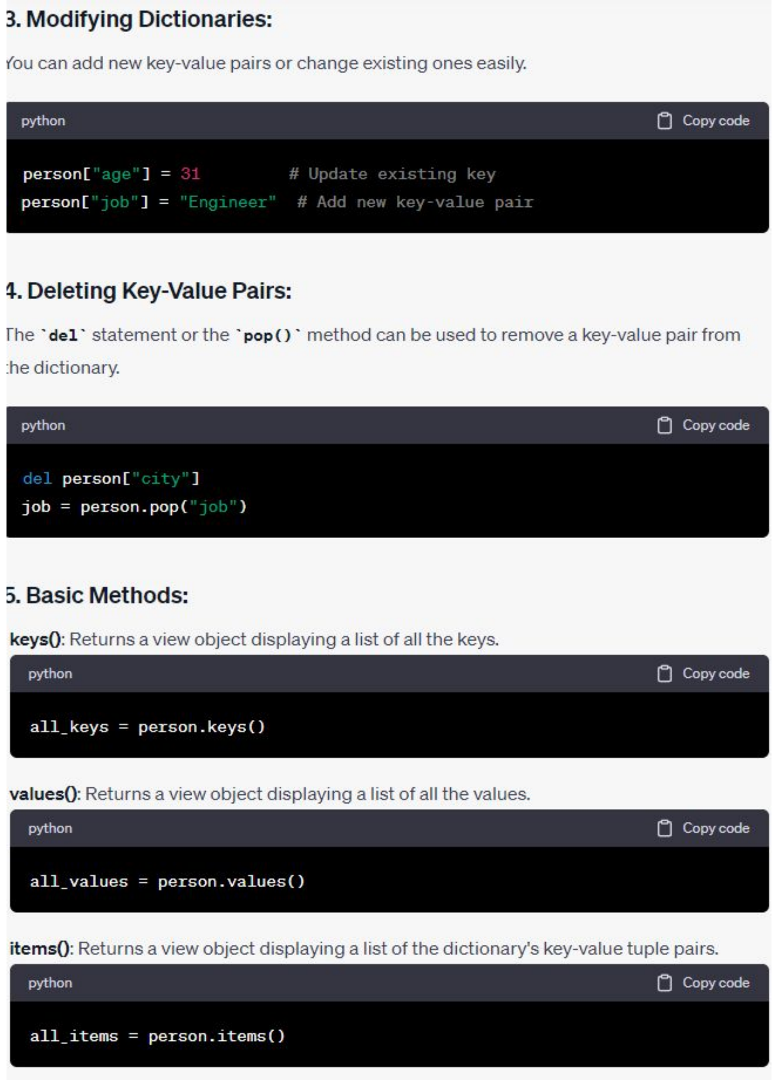
2. Accessing Values:

You can access a value in a dictionary using its corresponding key inside square brackets `[]`.

```python
print(person["name"])  # John
```

Alternatively, the `get()` method can be used, which returns `None` if the key doesn't exist (or a default value that you can specify).

```python
print(person.get("name"))  # John
print(person.get("job", "Unknown"))  # Unknown
```



3. Modifying Dictionaries:

You can add new key-value pairs or change existing ones easily.

```python
person["age"] = 31          # Update existing key
person["job"] = "Engineer"  # Add new key-value pair
```

4. Deleting Key-Value Pairs:

The `del` statement or the `pop()` method can be used to remove a key-value pair from he dictionary.

```python
del person["city"]
job = person.pop("job")
```

5. Basic Methods:

keys(): Returns a view object displaying a list of all the keys.

```python
all_keys = person.keys()
```

values(): Returns a view object displaying a list of all the values.

```python
all_values = person.values()
```

items(): Returns a view object displaying a list of the dictionary's key-value tuple pairs.

```python
all_items = person.items()
```

- **update(another_dict)**: Merges the dictionary with another dictionary or with an iterable of key-value pairs.

python Copy code

```python
person.update({"city": "London", "country": "UK"})
```

- **clear()**: Removes all items from the dictionary.

python Copy code

```python
person.clear()
```

6. Iteration:

You can iterate over a dictionary's keys, values, or key-value pairs.

python Copy code

```python
for key in person:
    print(key, person[key])

# or using items() for both key and value:
for key, value in person.items():
    print(key, value)
```

7. Length:

The `len()` function returns the number of key-value pairs in the dictionary.

python Copy code

```python
print(len(person))  # 3
```

8. Nested Dictionaries:

Dictionaries can contain other dictionaries as values (or any other data type), allowing you to create tree-like structures.

python Copy code

```python
team = {
    "engineer": {
        "name": "John",
        "age": 30
    },
    "manager": {
        "name": "Jane",
        "age": 35
    }
}
```

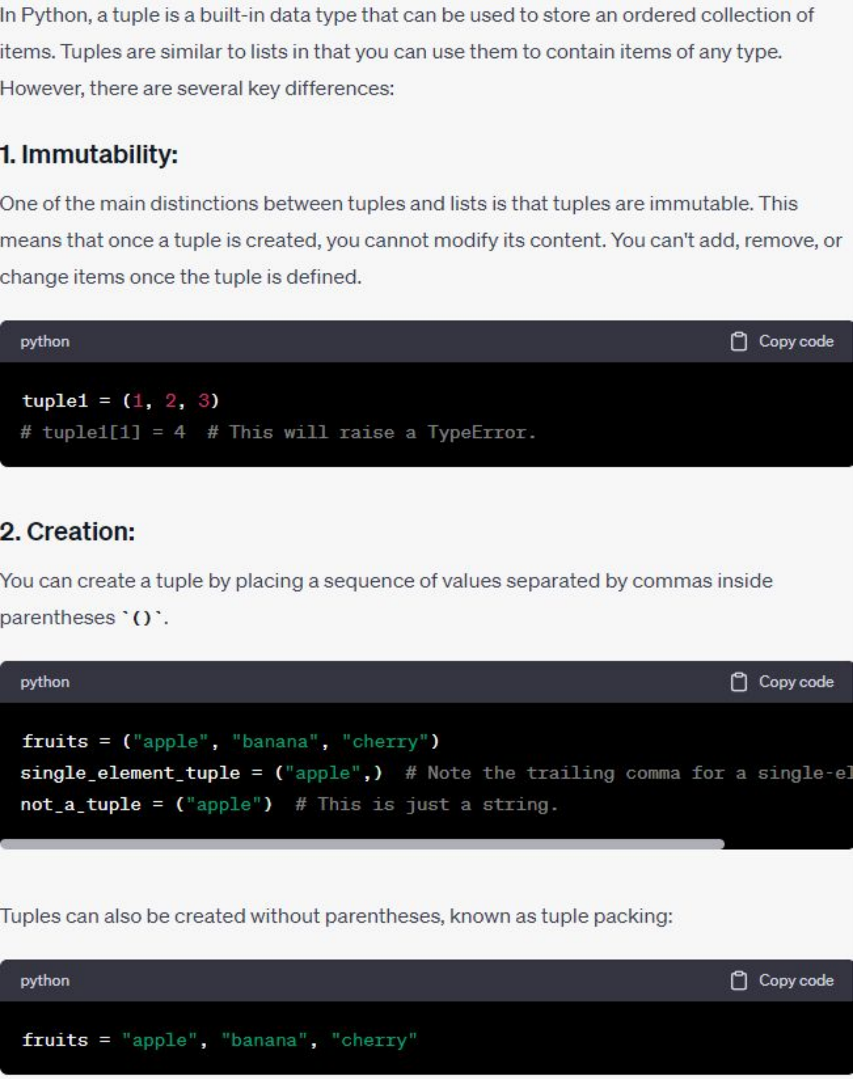
9. Dictionary Comprehension:

Just like list comprehension, Python supports dictionary comprehension for concise dictionary creation.

python Copy code

```python
squares = {x: x**2 for x in (1, 2, 3, 4, 5)}
```

Dictionaries in Python are very versatile and useful, especially when you need to represent data with key-value pairs, such as configuration settings, JSON-like data structures, or


In Python, a tuple is a built-in data type that can be used to store an ordered collection of items. Tuples are similar to lists in that you can use them to contain items of any type. However, there are several key differences:

1. Immutability:

One of the main distinctions between tuples and lists is that tuples are immutable. This means that once a tuple is created, you cannot modify its content. You can't add, remove, or change items once the tuple is defined.

```python
tuple1 = (1, 2, 3)
# tuple1[1] = 4  # This will raise a TypeError.
```

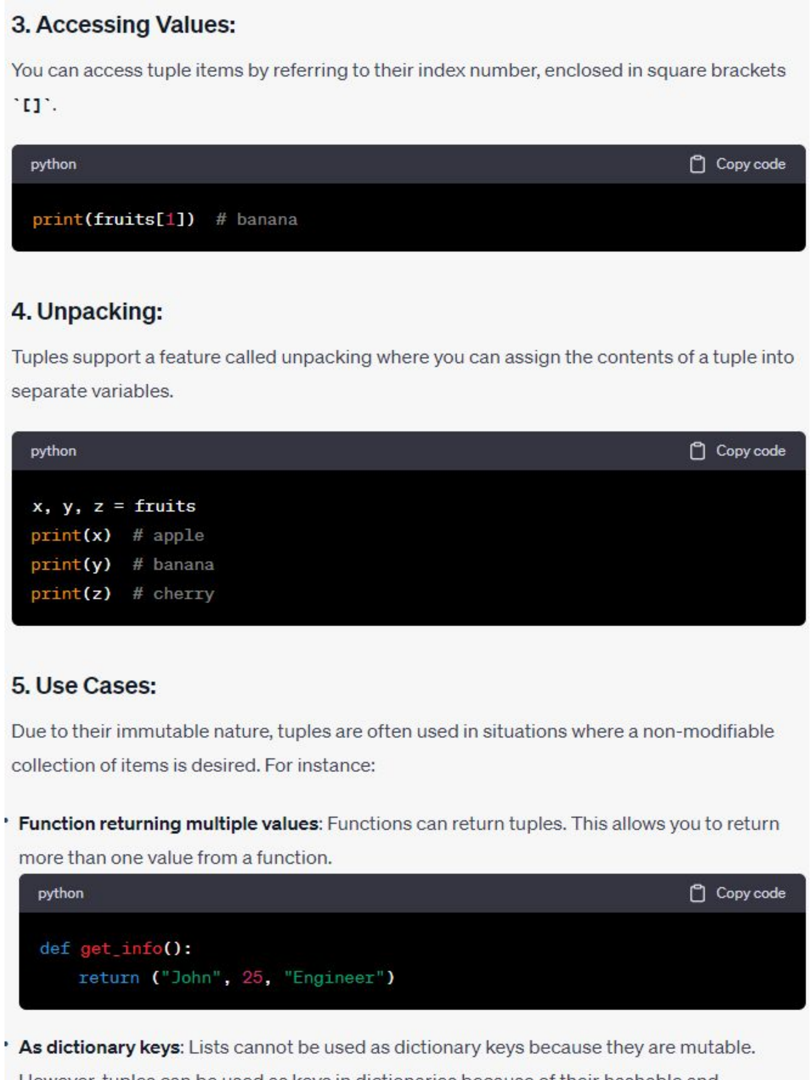
2. Creation:

You can create a tuple by placing a sequence of values separated by commas inside parentheses `()`.

```python
fruits = ("apple", "banana", "cherry")
single_element_tuple = ("apple",)  # Note the trailing comma for a single-el
not_a_tuple = ("apple")  # This is just a string.
```

Tuples can also be created without parentheses, known as tuple packing:

```python
fruits = "apple", "banana", "cherry"
```


3. Accessing Values:

You can access tuple items by referring to their index number, enclosed in square brackets `[]`.

```python
print(fruits[1])  # banana
```

4. Unpacking:

Tuples support a feature called unpacking where you can assign the contents of a tuple into separate variables.

```python
x, y, z = fruits
print(x)  # apple
print(y)  # banana
print(z)  # cherry
```

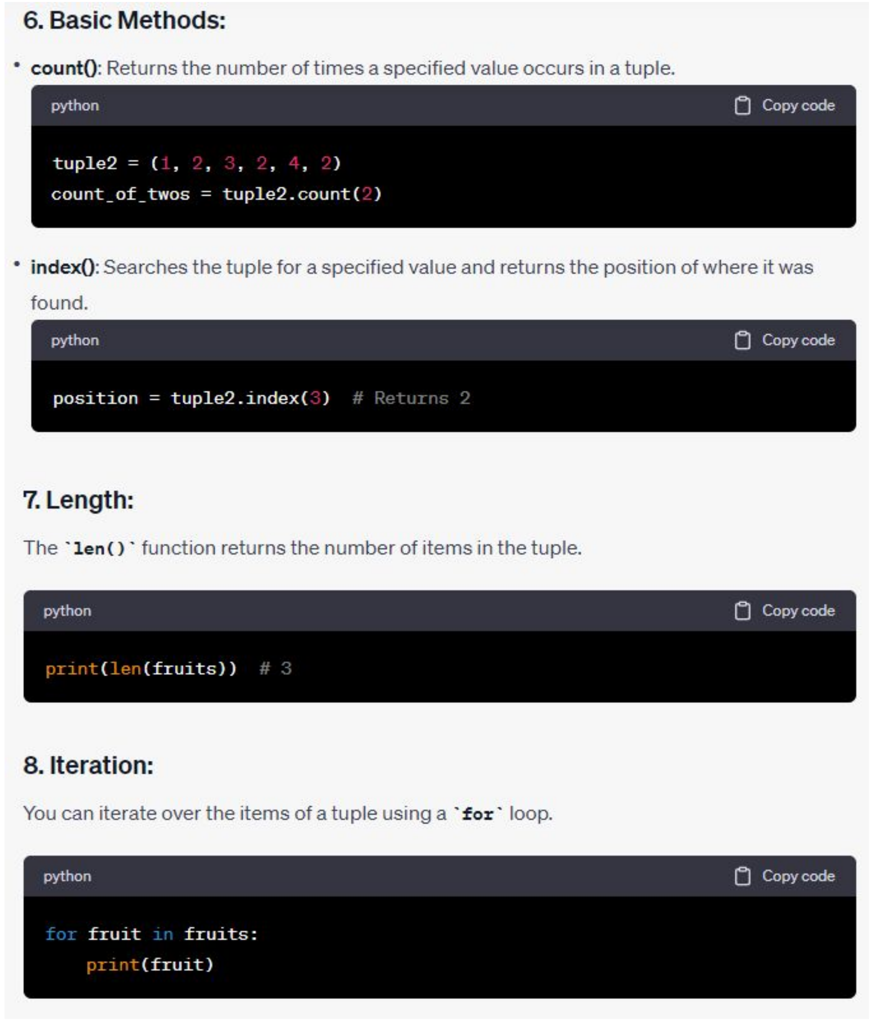
5. Use Cases:

Due to their immutable nature, tuples are often used in situations where a non-modifiable collection of items is desired. For instance:

- **Function returning multiple values**: Functions can return tuples. This allows you to return more than one value from a function.

```python
def get_info():
    return ("John", 25, "Engineer")
```

- **As dictionary keys**: Lists cannot be used as dictionary keys because they are mutable. However, tuples can be used as keys in dictionaries because of their hashable and



6. Basic Methods:

- **count()**: Returns the number of times a specified value occurs in a tuple.

```python
tuple2 = (1, 2, 3, 2, 4, 2)
count_of_twos = tuple2.count(2)
```

- **index()**: Searches the tuple for a specified value and returns the position of where it was found.

```python
position = tuple2.index(3)  # Returns 2
```

7. Length:

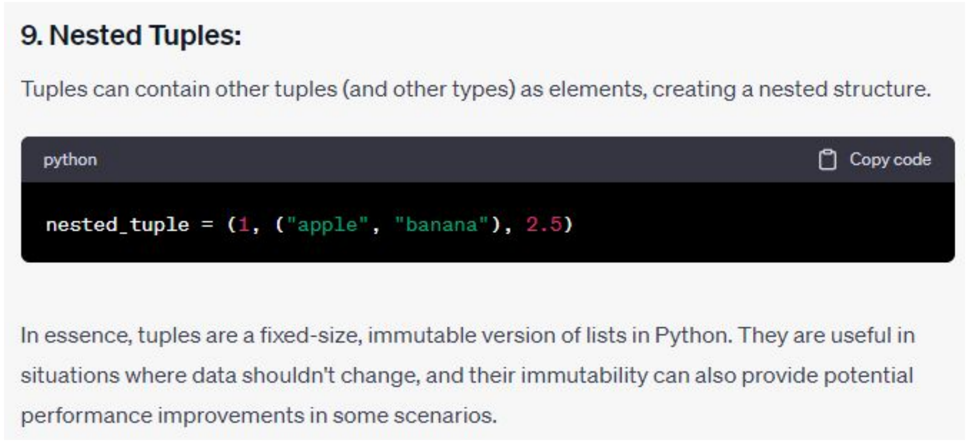
The `len()` function returns the number of items in the tuple.

```python
print(len(fruits))  # 3
```

8. Iteration:

You can iterate over the items of a tuple using a `for` loop.

```python
for fruit in fruits:
    print(fruit)
```



9. Nested Tuples:

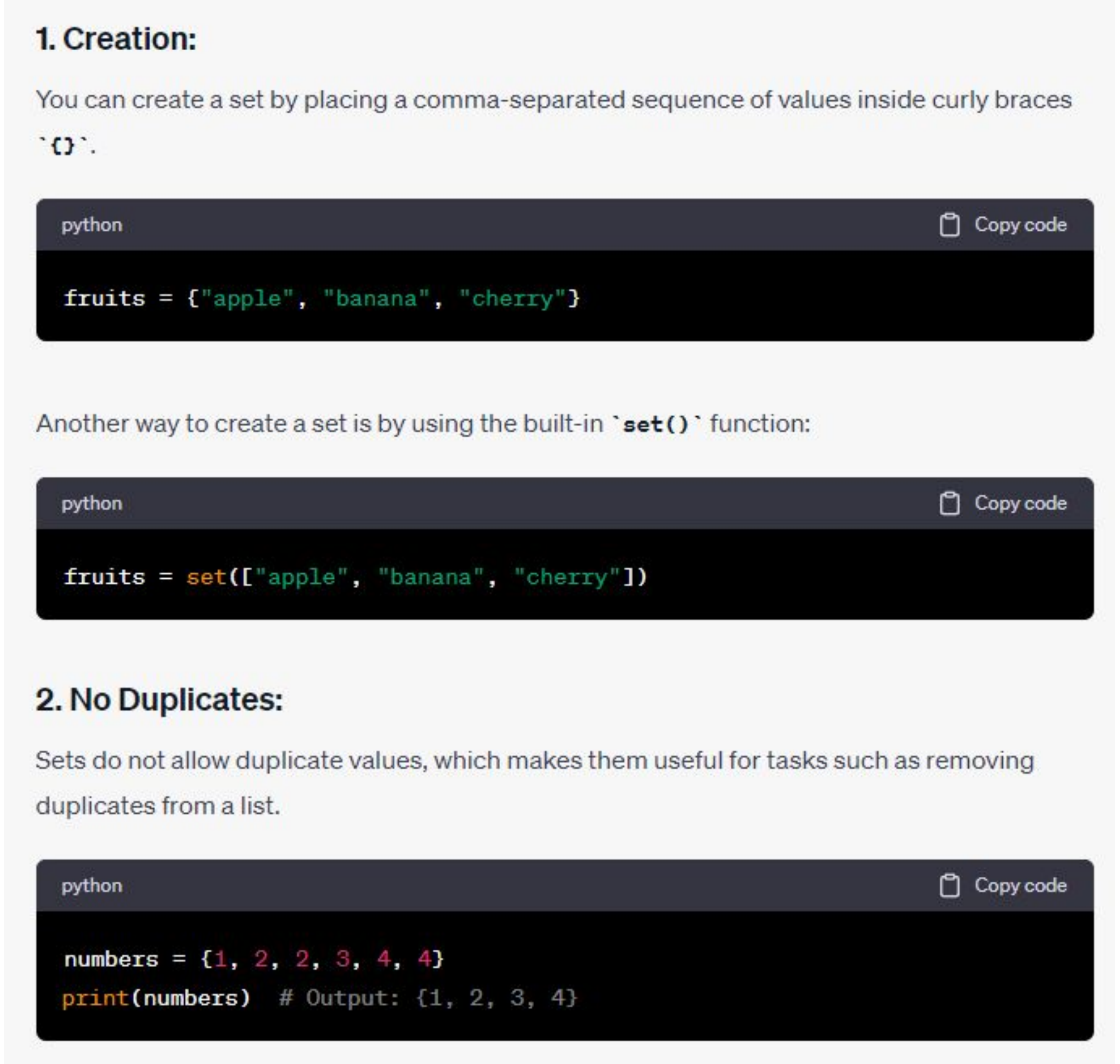
Tuples can contain other tuples (and other types) as elements, creating a nested structure.

```python
nested_tuple = (1, ("apple", "banana"), 2.5)
```

In essence, tuples are a fixed-size, immutable version of lists in Python. They are useful in situations where data shouldn't change, and their immutability can also provide potential performance improvements in some scenarios.

SETS

In Python, a set is a built-in data type that represents an unordered collection of unique items. Sets are similar to lists and tuples, but they do not allow duplicate values and do not maintain the items in any specific order.



1. Creation:

You can create a set by placing a comma-separated sequence of values inside curly braces `{}`.

python Copy code

```python
fruits = {"apple", "banana", "cherry"}
```

Another way to create a set is by using the built-in `set()` function:

python Copy code

```python
fruits = set(["apple", "banana", "cherry"])
```

2. No Duplicates:

Sets do not allow duplicate values, which makes them useful for tasks such as removing duplicates from a list.

python Copy code

```python
numbers = {1, 2, 2, 3, 4, 4}
print(numbers)  # Output: {1, 2, 3, 4}
```

3. Unordered:

Sets are unordered, which means that they don't record element position or order of insertion. Therefore, sets do not support indexing, slicing, or any sequence-like behavior.

4. Mutable:

While sets themselves are mutable (you can add or remove items), they can only contain immutable (hashable) data types. This means you can have a tuple inside a set, but not lists or dictionaries.

5. Basic Operations:

add(): Adds an element to the set.

```python
fruits.add("orange")
```

remove(): Removes a specified element from the set. Raises an error if the element doesn't exist.

```python
fruits.remove("banana")
```

discard(): Removes a specified element from the set. Does not raise an error if the element doesn't exist.

```python
fruits.discard("banana")
```

pop(): Removes and returns an arbitrary item from the set. Raises an error if the set is empty.

```python
fruit = fruits.pop()
```

- **clear()**: Removes all elements from the set.

```python
fruits.clear()
```

6. Set Operations:

Sets support several standard mathematical operations:

- **Union (`|`)**: Combines elements from two sets. Can also use the `union()` method.

```python
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)  # {1, 2, 3, 4, 5}
```

- **Intersection (`&`)**: Returns only the elements that are common to both sets. Can also use the `intersection()` method.

```python
print(a & b)  # {3}
```

- **Difference (`-`)**: Returns the elements from the first set that are not in the second set. Can also use the `difference()` method.

```python
print(a - b)  # {1, 2}
```

- **Symmetric Difference (`^`)**: Returns elements that are in one of the sets, but not in both. Can also use the `symmetric_difference()` method.

```python
print(a ^ b)  # {1, 2, 4, 5}
```

7. Membership Testing:

You can check if an item exists in a set using the `in` keyword:

```python
if "apple" in fruits:
    print("Apple is in the set.")
```

8. Length:

You can get the number of items in a set using the `len()` function:

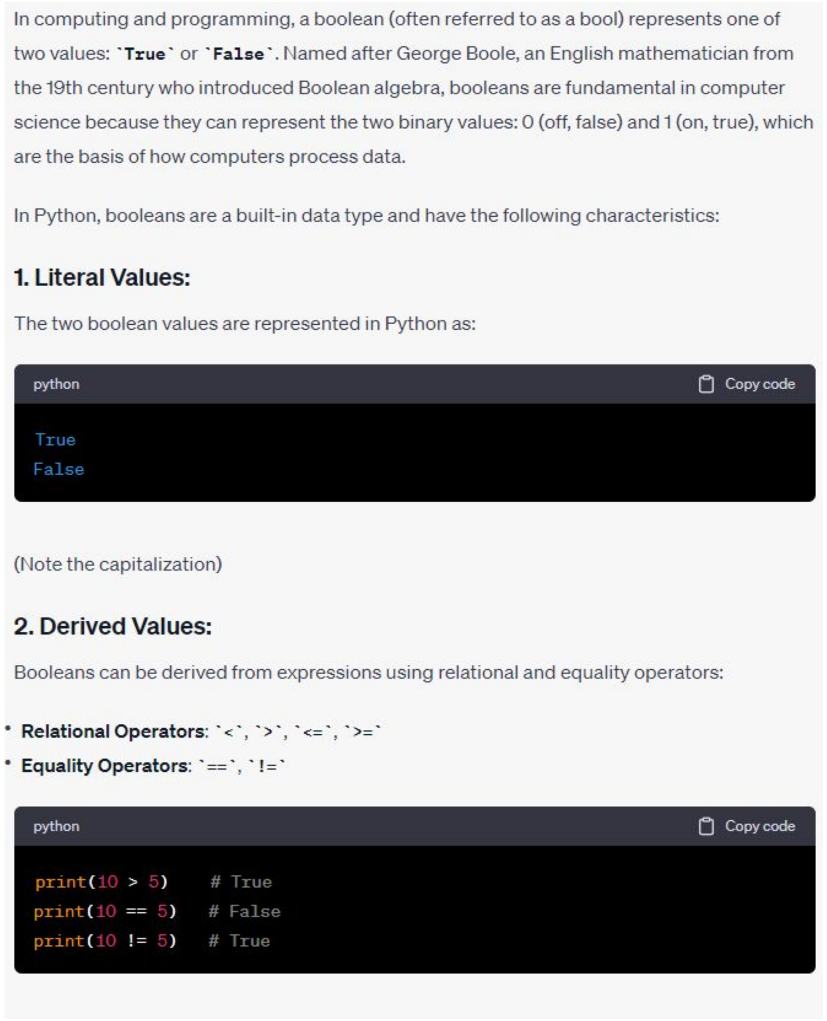
```python
print(len(fruits))
```

9. Immutability with `frozenset`:

If you need an immutable version of a set (for example, if you want to use a set as a key in a dictionary), Python provides the `frozenset` data type.

```python
immutable_set = frozenset(["apple", "banana", "cherry"])
```

In summary, sets in Python provide a powerful tool for mathematical set operations and for situations where you want to ensure the presence of unique elements. They can be especially handy when working with large datasets where checking for duplicates or finding common elements is necessary.



In computing and programming, a boolean (often referred to as a bool) represents one of two values: `True` or `False`. Named after George Boole, an English mathematician from the 19th century who introduced Boolean algebra, booleans are fundamental in computer science because they can represent the two binary values: 0 (off, false) and 1 (on, true), which are the basis of how computers process data.

In Python, booleans are a built-in data type and have the following characteristics:

1. Literal Values:

The two boolean values are represented in Python as:

python Copy code

```python
True
False
```

(Note the capitalization)

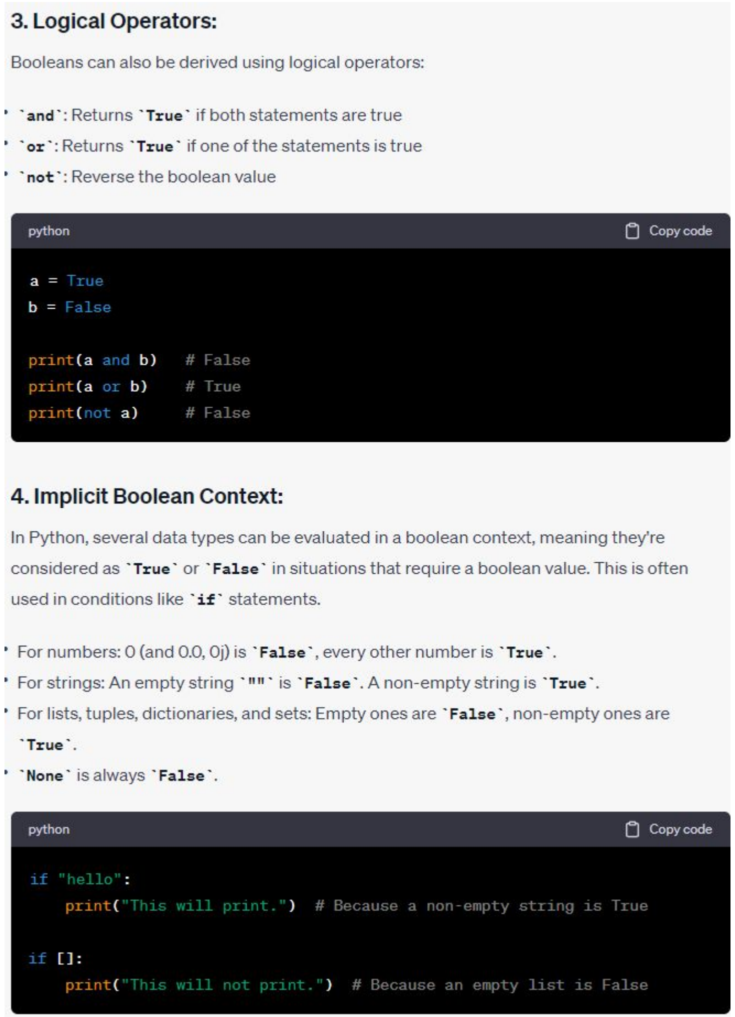
2. Derived Values:

Booleans can be derived from expressions using relational and equality operators:

- **Relational Operators**: `<`, `>`, `<=`, `>=`
- **Equality Operators**: `==`, `!=`

python Copy code

```python
print(10 > 5)    # True
print(10 == 5)   # False
print(10 != 5)   # True
```



3. Logical Operators:

Booleans can also be derived using logical operators:

- `and`: Returns `True` if both statements are true
- `or`: Returns `True` if one of the statements is true
- `not`: Reverse the boolean value

python Copy code

```python
a = True
b = False

print(a and b)    # False
print(a or b)     # True
print(not a)      # False
```

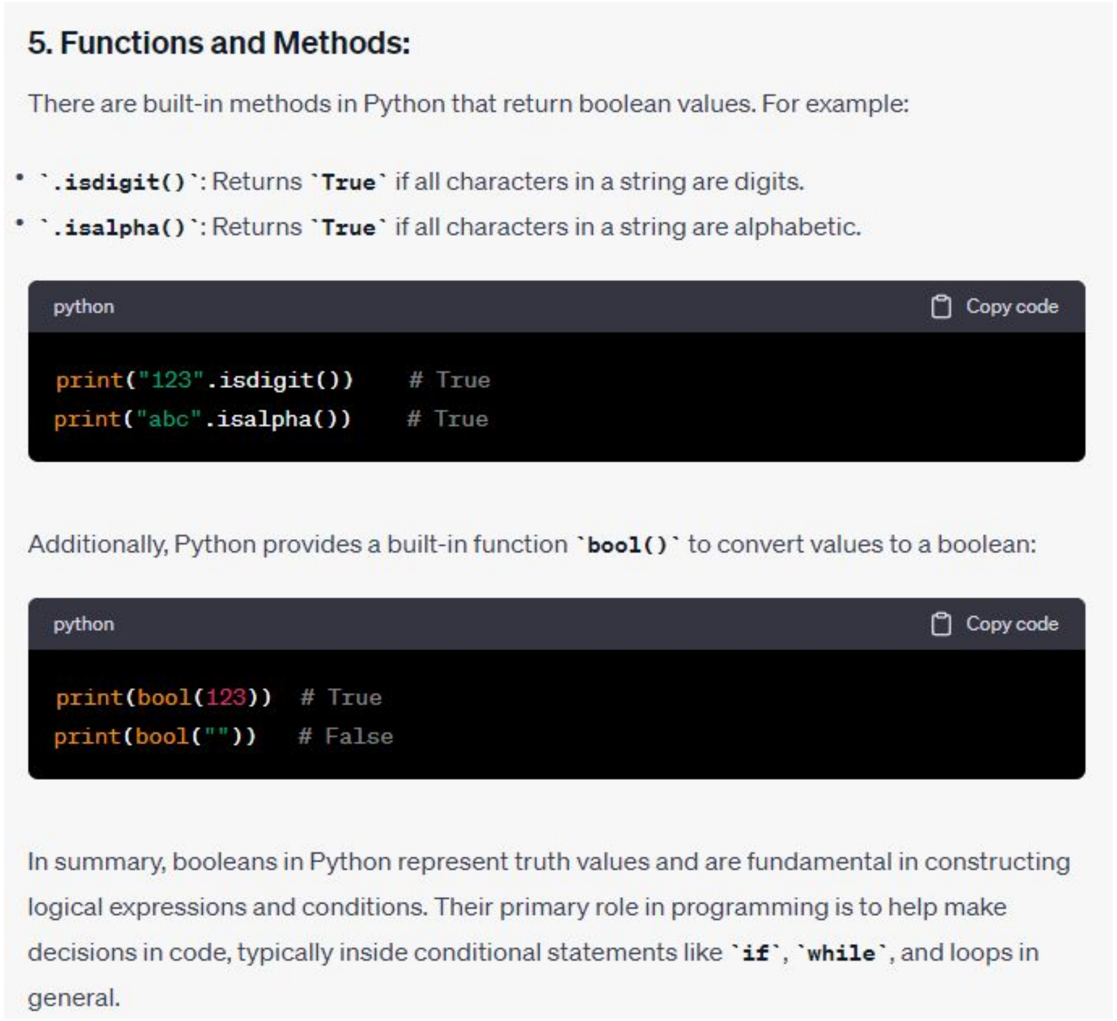
4. Implicit Boolean Context:

In Python, several data types can be evaluated in a boolean context, meaning they're considered as `True` or `False` in situations that require a boolean value. This is often used in conditions like `if` statements.

- For numbers: 0 (and 0.0, 0j) is `False`, every other number is `True`.
- For strings: An empty string `""` is `False`. A non-empty string is `True`.
- For lists, tuples, dictionaries, and sets: Empty ones are `False`, non-empty ones are `True`.
- `None` is always `False`.

python Copy code

```python
if "hello":
    print("This will print.")  # Because a non-empty string is True

if []:
    print("This will not print.")  # Because an empty list is False
```



5. Functions and Methods:

There are built-in methods in Python that return boolean values. For example:

- **`.isdigit()`**: Returns **`True`** if all characters in a string are digits.
- **`.isalpha()`**: Returns **`True`** if all characters in a string are alphabetic.

```python
print("123".isdigit())    # True
print("abc".isalpha())    # True
```

Additionally, Python provides a built-in function **`bool()`** to convert values to a boolean:

```python
print(bool(123))  # True
print(bool(""))   # False
```

In summary, booleans in Python represent truth values and are fundamental in constructing logical expressions and conditions. Their primary role in programming is to help make decisions in code, typically inside conditional statements like **`if`**, **`while`**, and loops in general.