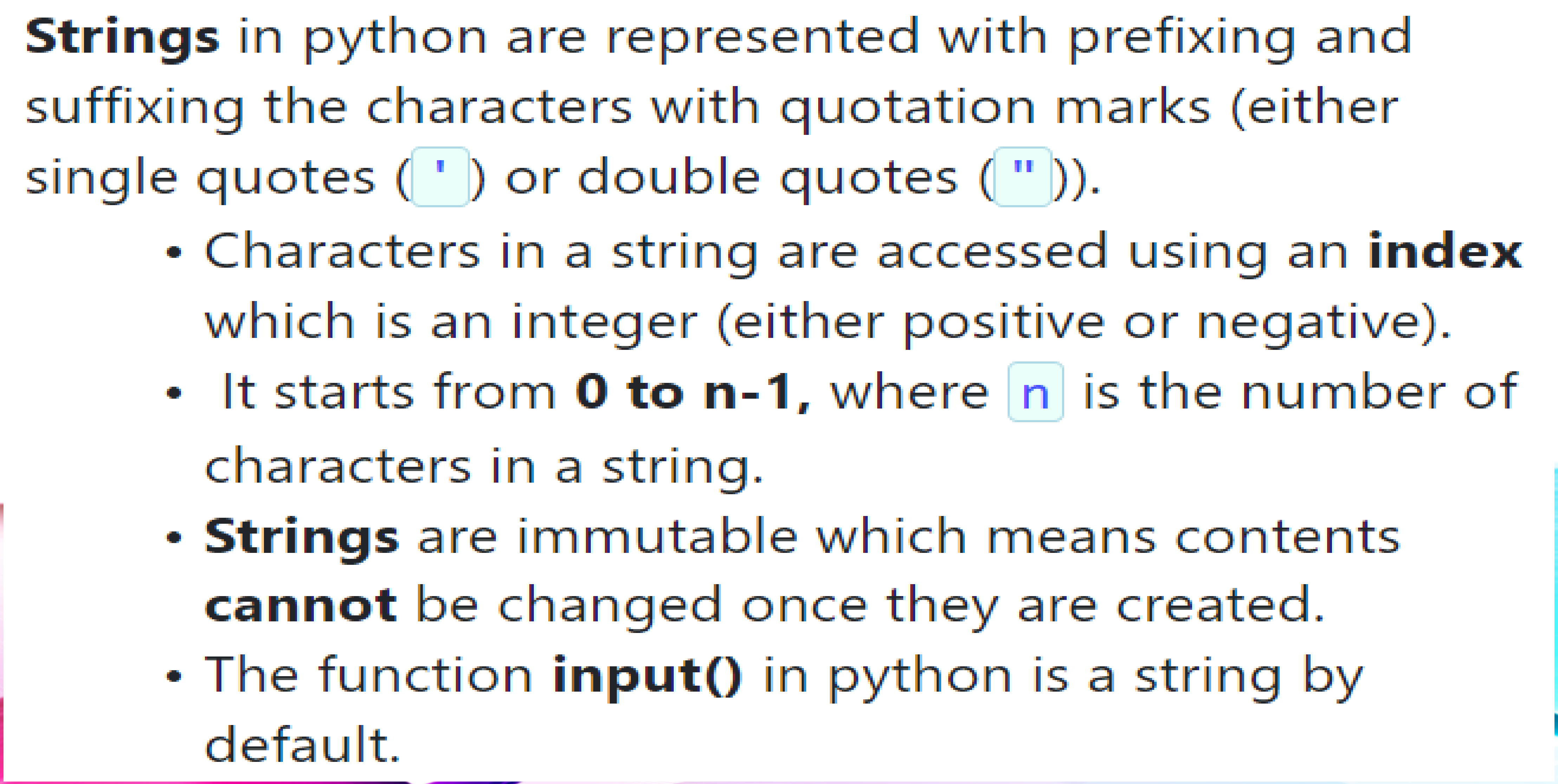
INTRODUCTION TO STRINGS



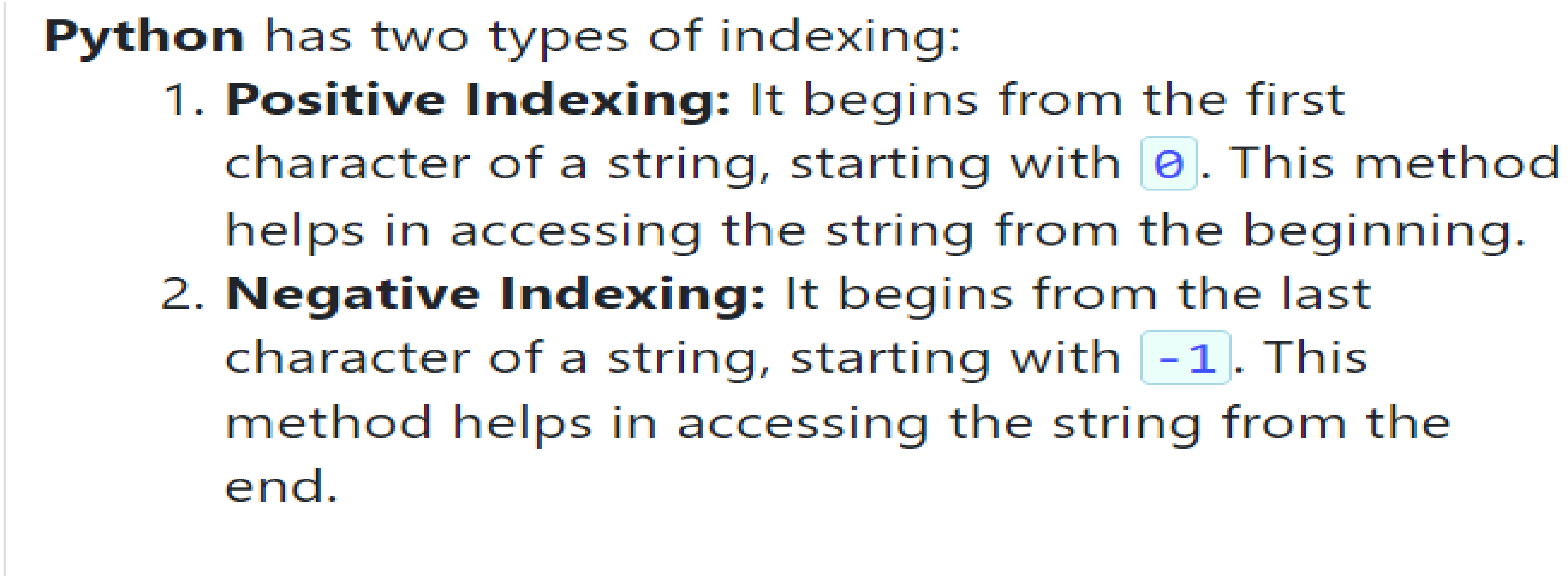
Strings in python are represented with prefixing and suffixing the characters with quotation marks (either single quotes (`'`) or double quotes (`"`)).

- Characters in a string are accessed using an **index** which is an integer (either positive or negative).
- It starts from **0 to n-1,** where `n` is the number of characters in a string.
- **Strings** are immutable which means contents **cannot** be changed once they are created.
- The function **input()** in python is a string by default.

STRING OPERATIONS

In **Python**, there are **5** fundamental operations which can be performed on strings:

1. **Indexing**
2. **Slicing**
3. **Concatenation**
4. **Repetition**
5. **Membership**



Python has two types of indexing:

1. **Positive Indexing:** It begins from the first character of a string, starting with `0`. This method helps in accessing the string from the beginning.
2. **Negative Indexing:** It begins from the last character of a string, starting with `-1`. This method helps in accessing the string from the end.

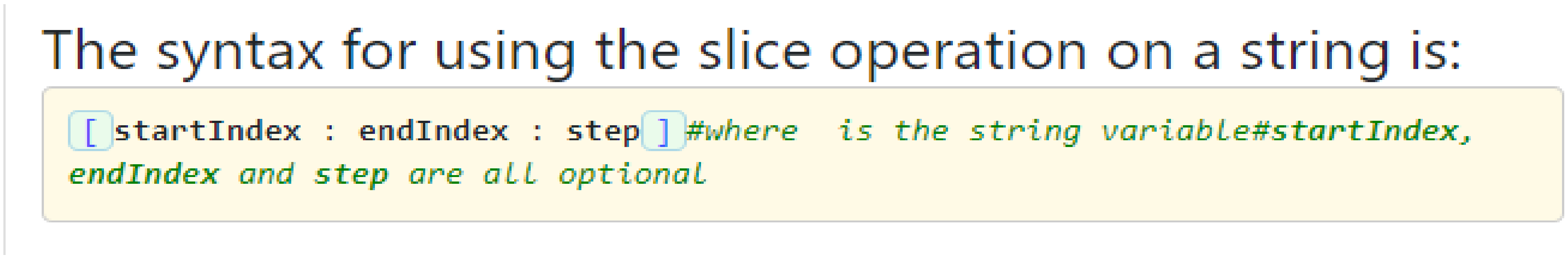
Given a string **"This is my first String"**. Write the code to achieve the following.

1. print the entire string
2. print the character `f` using positive/forward indexing
3. print the character `S` using negative/backward indexing

SLICING

Python provides many ways to extract a part of a string using a concept called **Slicing**.

Slicing makes it possible to access parts (segments) of the string by providing a **startIndex** and an **endIndex**.



The syntax for using the slice operation on a string is:

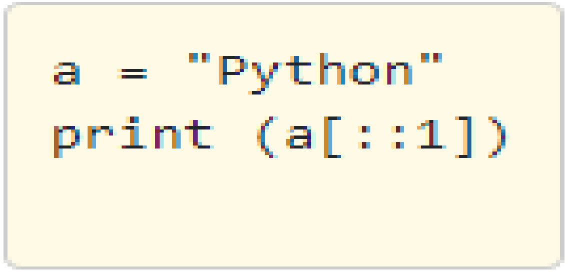
```
[ startIndex : endIndex : step ]#where  is the string variable#startIndex,
endIndex and step are all optional
```

step is the increment or decrement . A `positive` step will travel left to right and increments the index by step.

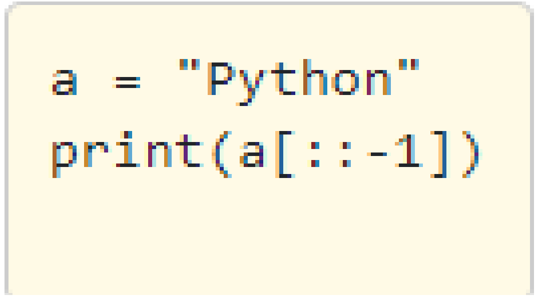
A `negative` step will travel from right to left and decrements the index by step.

If we are not providing starting of index position in Slice `[ ]` operator then interpreter takes starting index zero(**0**) as default.

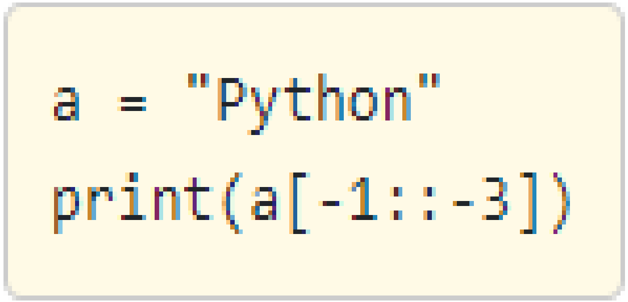
If end index is not specified for slice `[ ]` operator then Interpreter takes the end of String as default stop index.



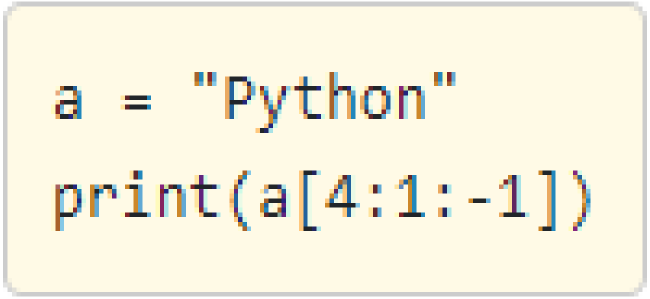
```
a = "Python"
print (a[::1])
```



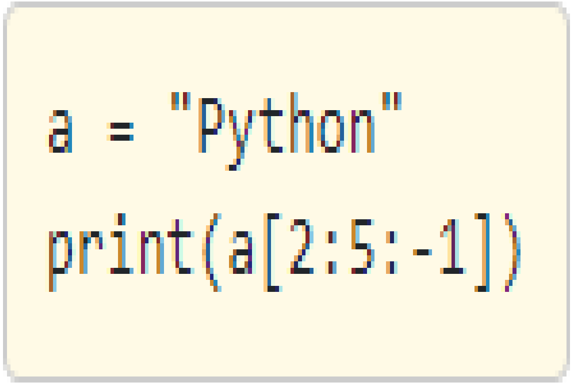
```
a = "Python"
print(a[::-1])
```



```
a = "Python"
print(a[-1::-3])
```



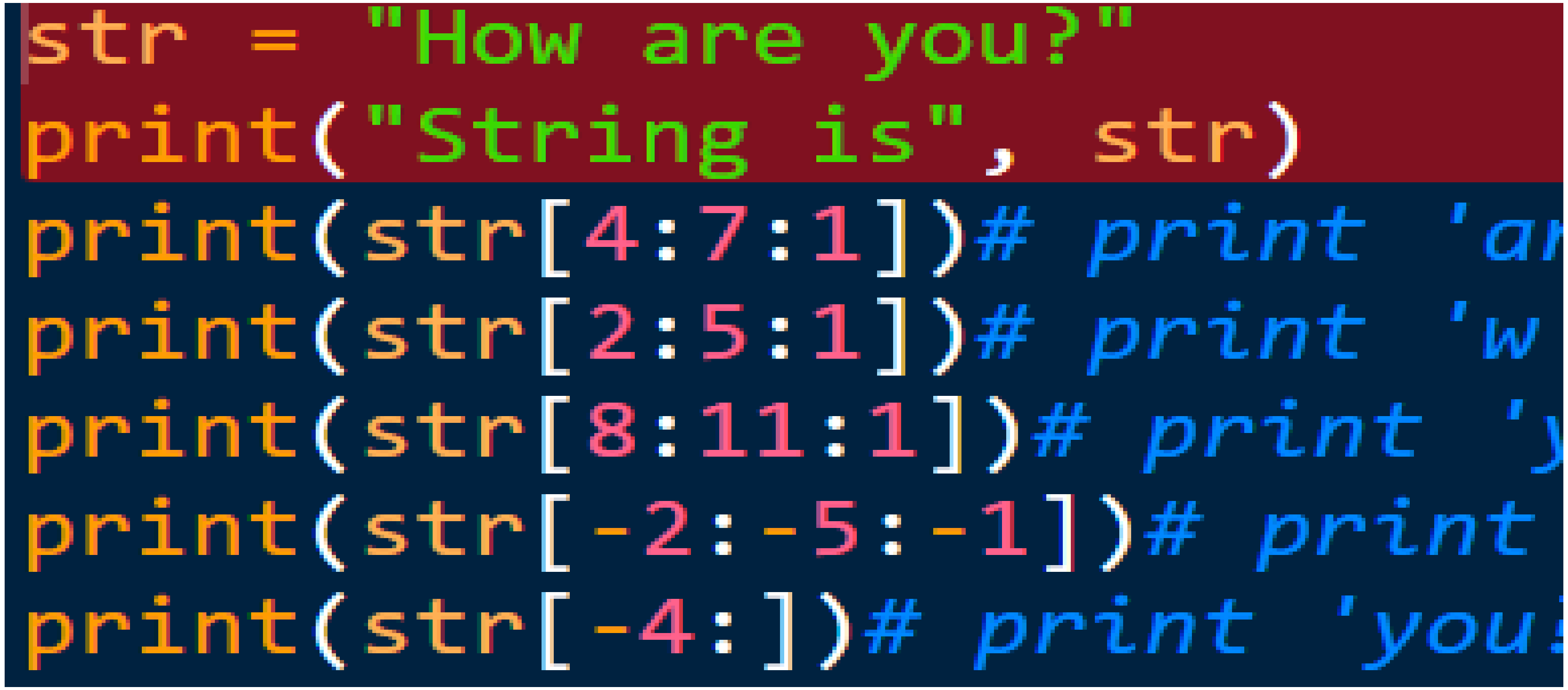
```
a = "Python"
print(a[4:1:-1])
```



```
a = "Python"
print(a[2:5:-1])
```

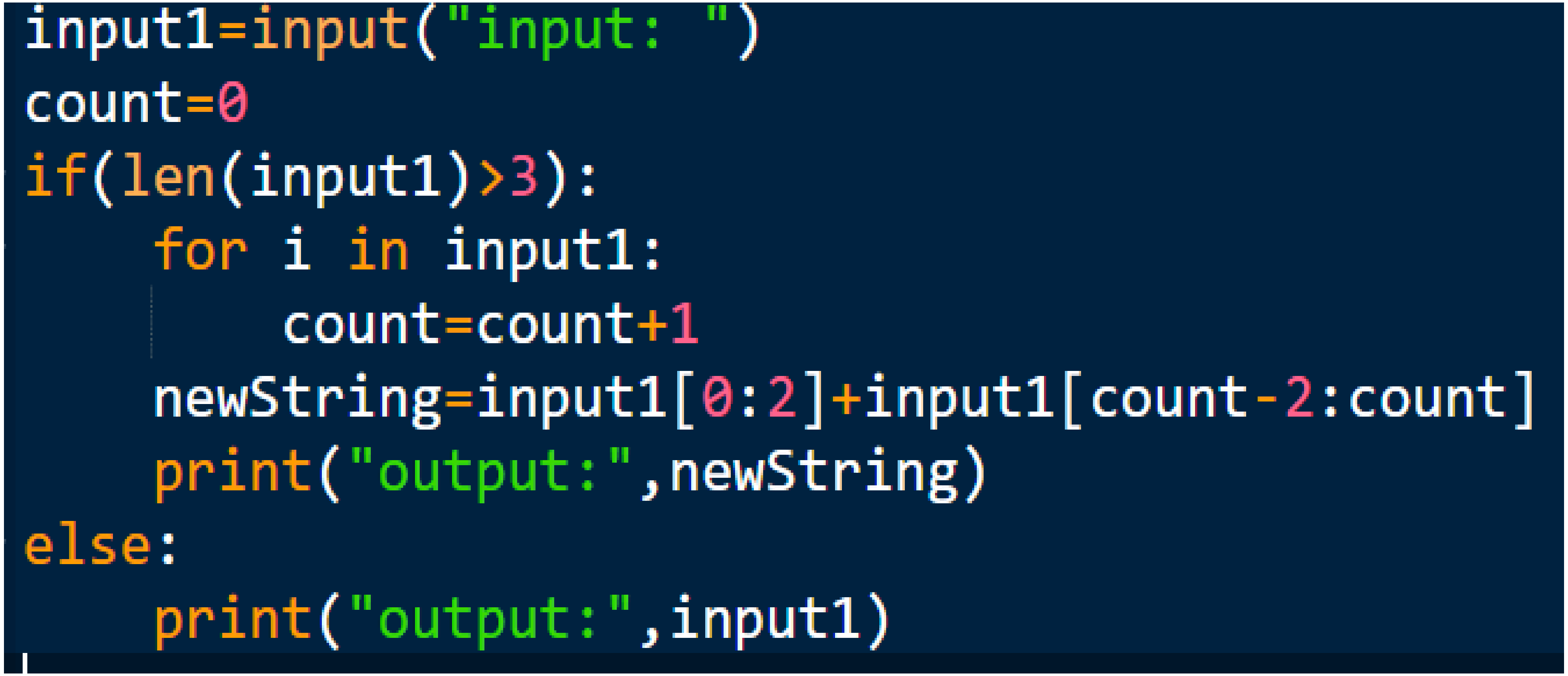
Sample Input and Output:

```
String is How are you?
are
w a
you
uoy
you?
```



```
str = "How are you?"
print("String is", str)
print(str[4:7:1])# print 'ar
print(str[2:5:1])# print 'w
print(str[8:11:1])# print 'y
print(str[-2:-5:-1])# print
print(str[-4:])# print 'you
```

Write a program to take a string as input from the user. Print first and last two characters of a string. If the length of the string is less than **3** then print the input string as it is.



```
input1=input("input: ")
count=0
if(len(input1)>3):
    for i in input1:
        count=count+1
    newString=input1[0:2]+input1[count-2:count]
    print("output:",newString)
else:
    print("output:",input1)
```

Write a program to take string as input from the user using `input()` function. Remove **first** and **last** **characters** of given string.

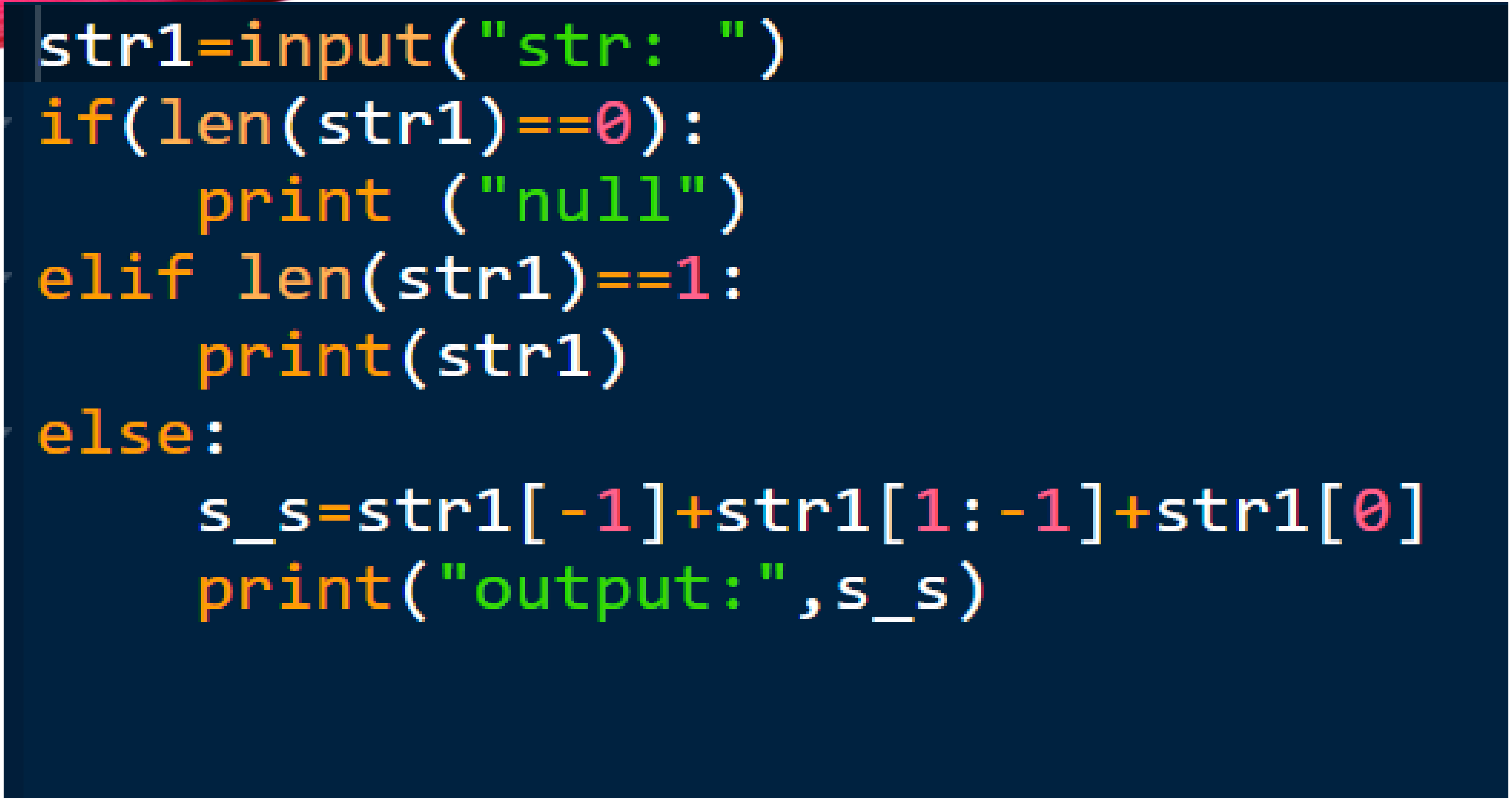
Write a program to take a string as input from the user. Swap the first and last characters of the given string. Print the resultant string as shown in the sample test case.

Follow the below constraints while writing the program:

- If length of the string is 1 then print the input string as it is.
- If length of the string is 0 (zero) then print **null** as output.

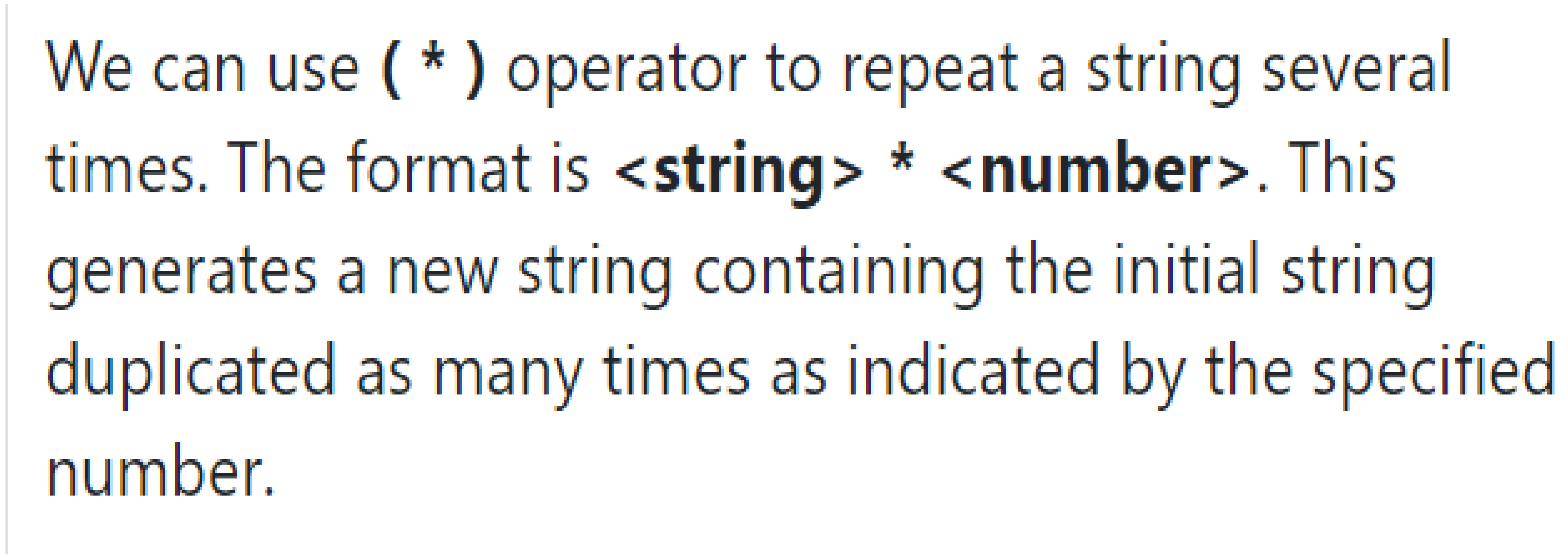
Sample Input and Output 1:

```
str: Active
output: ectivA
```



```
str1=input("str: ")
if(len(str1)==0):
    print ("null")
elif len(str1)==1:
    print(str1)
else:
    s_s=str1[-1]+str1[1:-1]+str1[0]
    print("output:",s_s)
```

REPETITION OF STRING



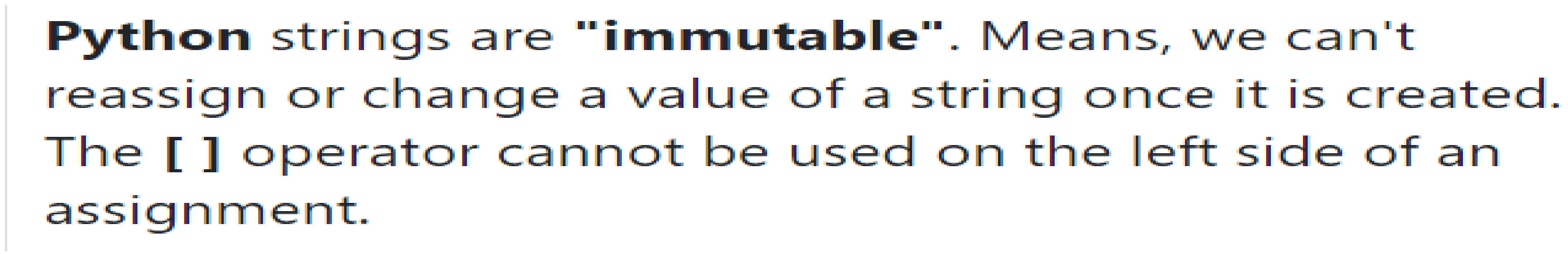
We can use **(*)** operator to repeat a string several times. The format is **<string> * <number>**. This generates a new string containing the initial string duplicated as many times as indicated by the specified number.

Example:

```
a = "Mouse"
print(3 * a)   # Output: MouseMouseMouse
print(a * 3)   # Output: MouseMouseMouse
b = 3
print(((b - 2) * 2) * a)   # Output: MouseMouse (repeated 2 times)
```

Write a program to take two inputs, string `str` and integer `n` from the user and print the given string `str` which repeats `n` times.

UNDERSTANDING STRING IMMUTABLITY



Python strings are **"immutable"**. Means, we can't reassign or change a value of a string once it is created. The **[ ]** operator cannot be used on the left side of an assignment.

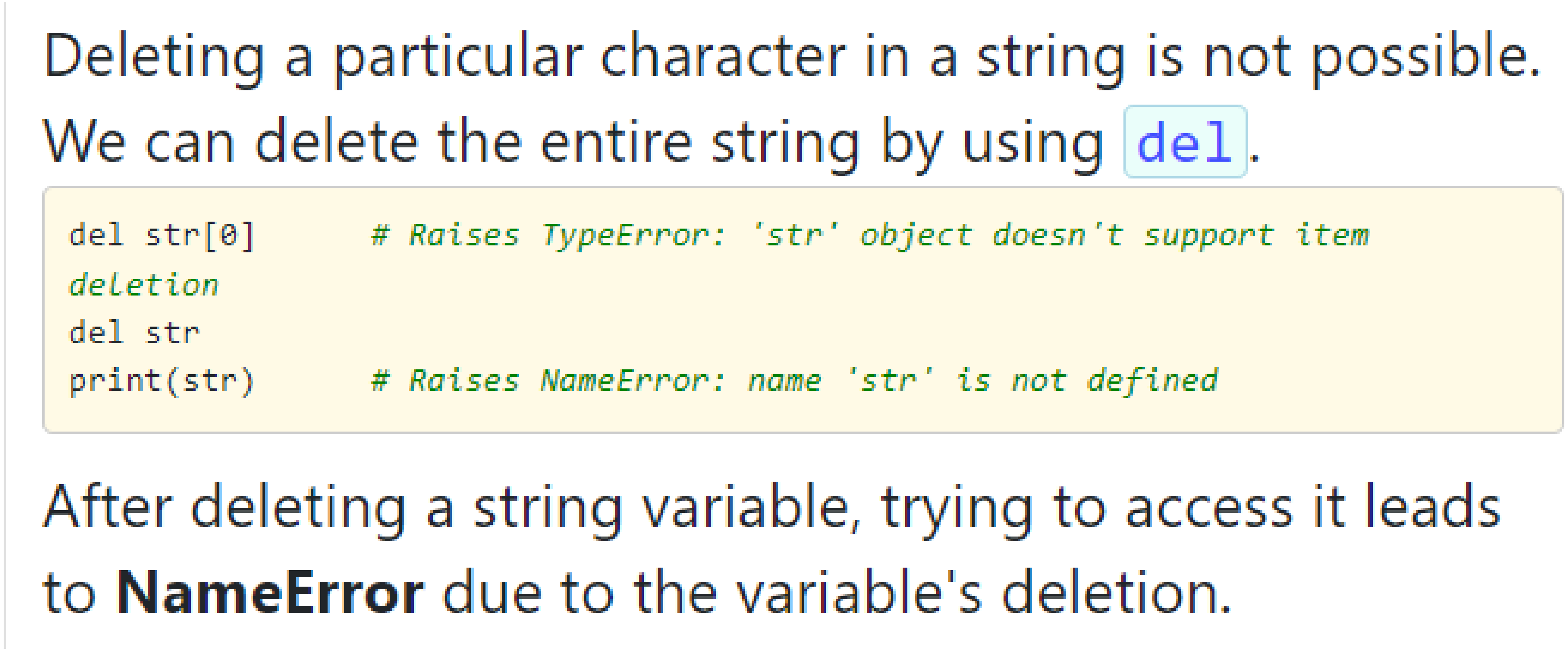
Consider the below example:

```
str = "Python"
str[0] = 'J'    # Attempt to change the 0th index 'P' with 'J'
```

Due to string immutability, this results a **TypeError:** 'str' object does not support item assignment.

However, we can create a new string:

```
newstr = "J" + str[1:]
print(newstr)   # Output: "Jython"
```

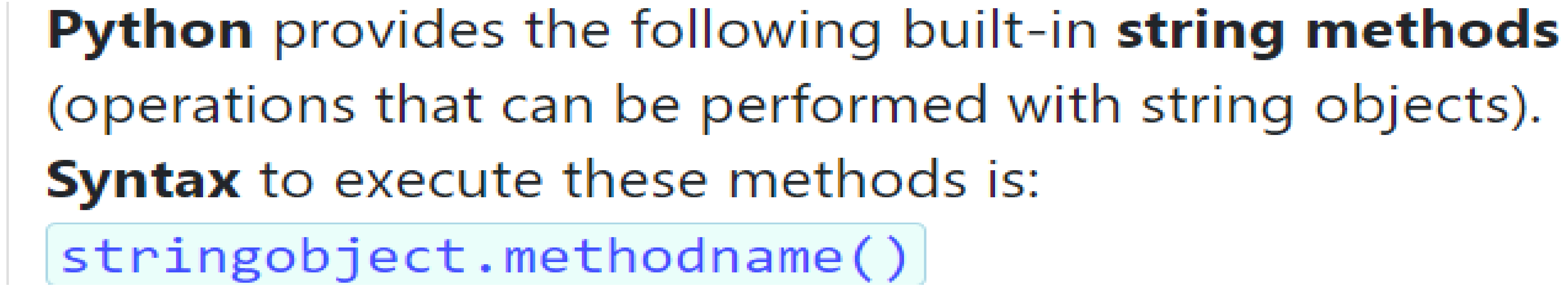


Deleting a particular character in a string is not possible. We can delete the entire string by using `del`.

```
del str[0]      # Raises TypeError: 'str' object doesn't support item
deletion
del str
print(str)      # Raises NameError: name 'str' is not defined
```

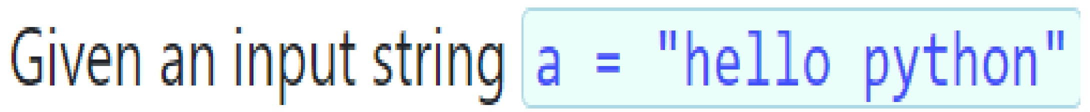
After deleting a string variable, trying to access it leads to **NameError** due to the variable's deletion.

BUILT-IN STRING METHODS



Python provides the following built-in **string methods** (operations that can be performed with string objects). **Syntax** to execute these methods is:

```
stringobject.methodname()
```



Given an input string `a = "hello python"`

1) **capitalize()** - Capitalizes first letter of a string.

```
print(a.capitalize())    # Result: Hello python
```

2) **upper()** - Converts the string to uppercase.

```
print(a.upper())          # Result: 'HELLO PYTHON'
```

3) **lower()** - Converts the string to uppercase.

```
print(a.lower())          # Result: 'hello python'
```

4) **title()** - Converts the string to title case. i.e., first characters of all the words of string are capitalized.

```
print(a.title())           # Result: 'Hello Python'
```

5) **swapcase()** - Swap the case of characters. i.e., lowercase into uppercase and vice versa.

```
print(a.swapcase())        # Result: 'HELLO PYTHON'
```

6) **split()** function returns a list of words separated by space.

```
print(a.split())           # Result: ['hello', 'python']
```

7) **center(width,"fillchar")** Center the string within a specified **width** using a **fill character**. Observe the below example:

```
print(a.center(20, '*'))   # Result: '****hello python****'
```

Here, width is `20` and string length is `12`, so now we need to fill the remaining width(20 - 12 = 8) with `'*'` special character.

8) **count()** returns the number of occurrences of substring in particular string. If the substring does not exist, it returns **zero**.

```
a = "happy married life happy birthday birthday"
print(a.count('happy'))     # Result: 2 (happy word occurred two times in a
string)
print(a.count('birthday'))  # Result: 2
print(a.count('life'))      # Result: 1
```

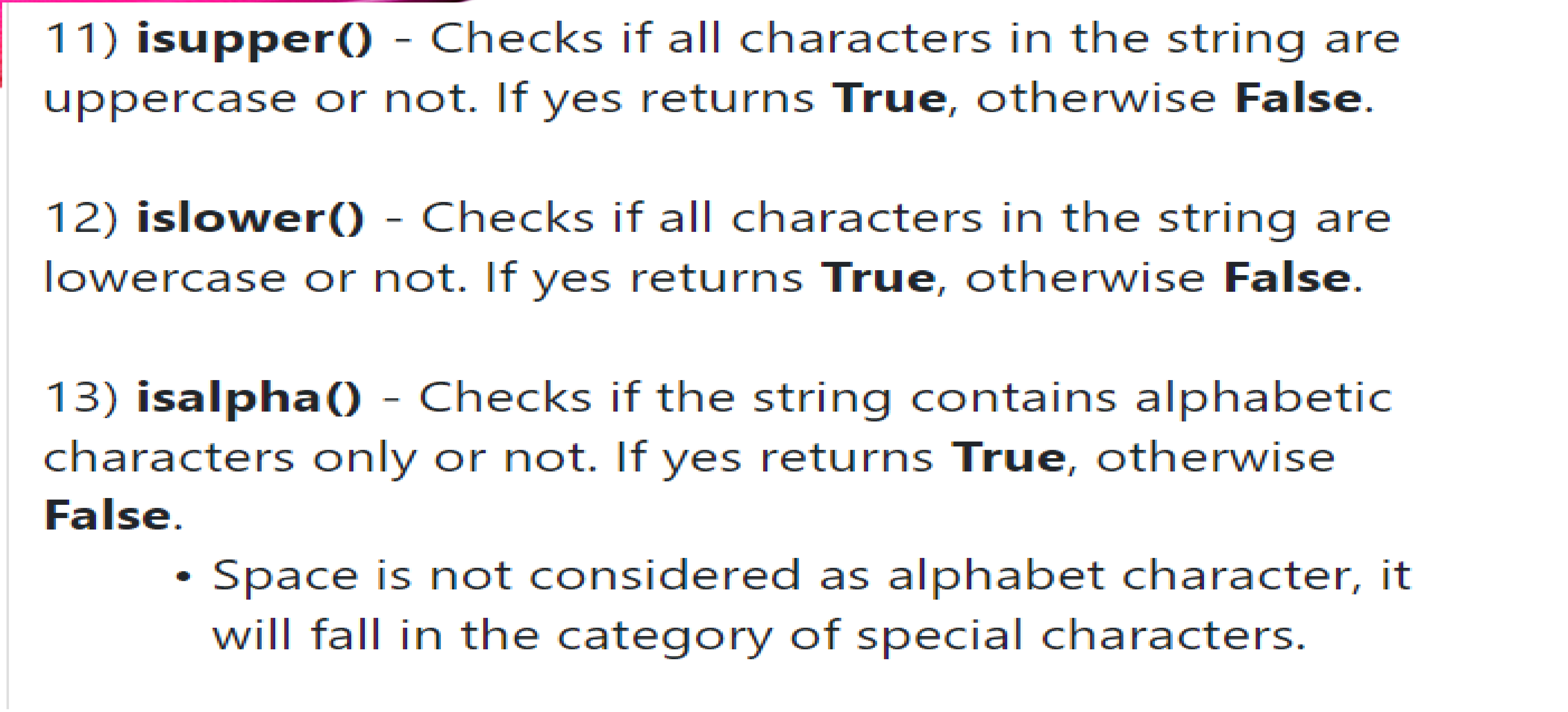
9) **replace(old, new)**, this method replaces all old substrings with new substrings. If the old substring does not exist, no modifications are done.

```
a = "java is simple"
print(a.replace('java' ,'Python'))   # Result: 'Python is simple'
```

Here **java** word is replaced with **Python**.

10) **join()** returns a string concatenated with the elements of an iterable. (Here “L1” is iterable)

```
b = '.'
L1 = ["www", "codetantra", "com"]
print(b.join(L1))    # Result: 'www.codetantra.com'
```



11) **isupper()** - Checks if all characters in the string are uppercase or not. If yes returns **True**, otherwise **False**.

12) **islower()** - Checks if all characters in the string are lowercase or not. If yes returns **True**, otherwise **False**.

13) **isalpha()** - Checks if the string contains alphabetic characters only or not. If yes returns **True**, otherwise **False**.

- Space is not considered as alphabet character, it will fall in the category of special characters.

14) **isalnum()** - Checks if the string contains alphanumeric characters only or not. If yes returns **True**, otherwise **False**.

- Characters those are not alphanumeric are: (space) ! # % & ? etc.
- Numerals (0-9), alphabets (A-Z, a-z) will fall into the category of alphanumeric characters.

15) **isdigit()** is used to check whether a string consists of digits only or not. If the string contains only digits then it returns **True**, otherwise **False**.

```
a = "123"
print(a.isdigit()) # will print result as follows
True
Here the string 'a' contains only digits so it returns True.
```

```
a = "123hello"
print(a.isdigit()) # will print result as follows
False
```

16) **isspace()** checks whether a string consists of spaces only or not. If it contains only white spaces then it returns **True** otherwise **False**.

```
a = "        "
print(a.isspace()) # will print the result as follows
True
Here the string 'a' contains only white spaces so it returns True.
```

```
a = " h "
print(a.isspace()) # will print the result as follows
False
```

Here the string 'a' contains white spaces with one character so it returns **False**.

```
a = "hello python"
print(a.isspace()) # will print the result as follows
False
```

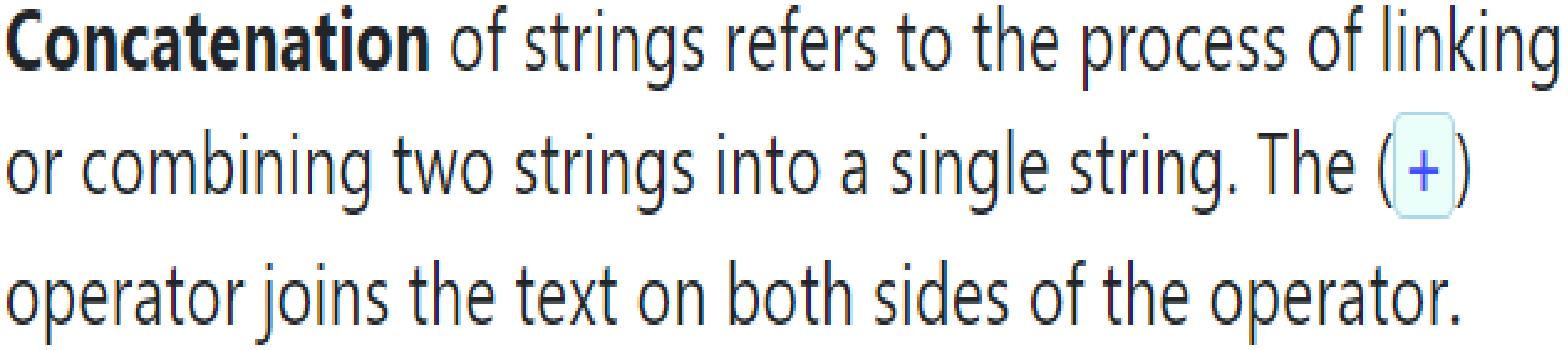
17) **istitle()** checks whether every word of a string starts with upper case letter or not.

```
a = "Hello Python"
print(a.istitle()) # will print the result as follows
True
```

```
a = "hello python"
print(a.istitle()) # will print the result as follows
False
```

Take two strings `str1` and `str2` as input from the console using `input()` function, print `str1` 3 times followed by `str2`. Print the result to the console as shown in the example.

CONCATENATION OF STRINGS



Concatenation of strings refers to the process of linking or combining two strings into a single string. The (+) operator joins the text on both sides of the operator.

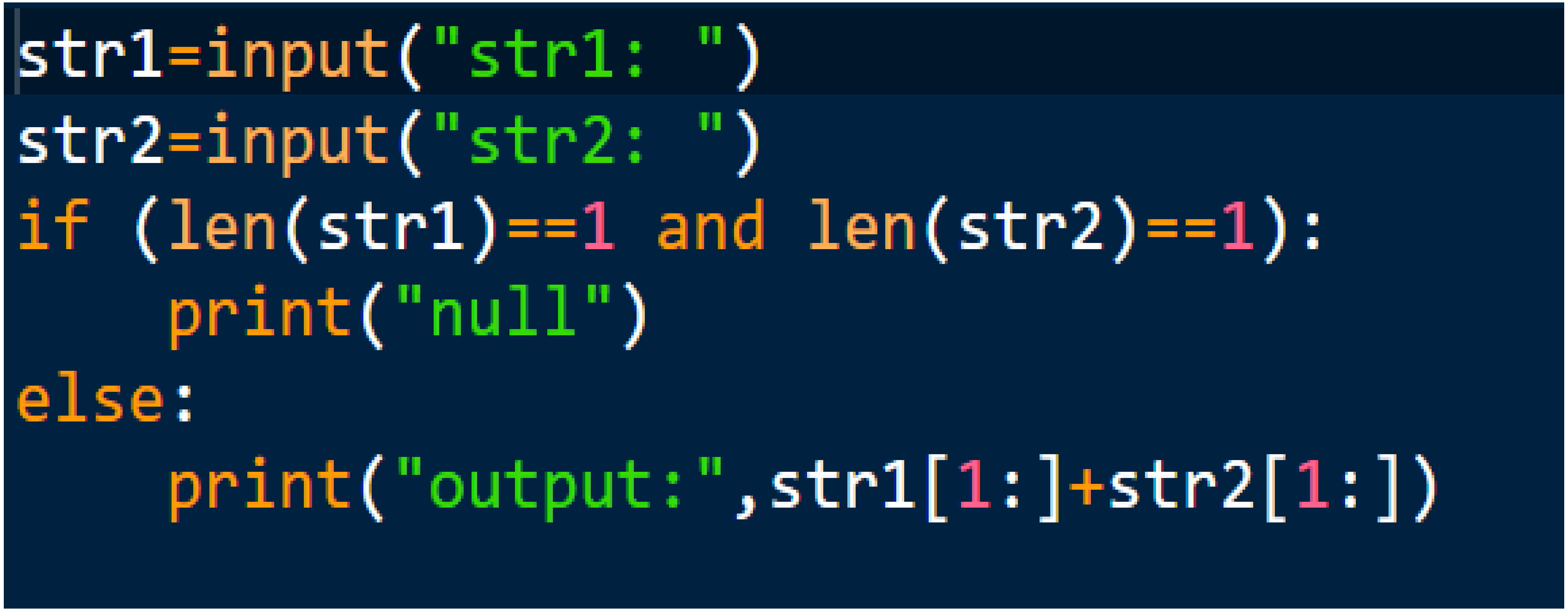
Write a program to take two strings as input from the user. Concatenate those two strings by removing the first character from both and print the result as shown in the sample test case.

Sample Input and Output 1:

```
str1: Python
str2: Java
output: ythonava
```

Sample Input and Output 2:

```
str1: a
str2: b
null
```



```
str1=input("str1: ")
str2=input("str2: ")
if (len(str1)==1 and len(str2)==1):
    print("null")
else:
    print("output:",str1[1:]+str2[1:])
```

ESCAPE CHARACTER

Escape characters are used to solve the problem of using special characters inside a string declaration. It directs the interpreter to take suitable actions mapped to that character. We denote escape characters with a backslash `(\)` at the beginning.

Let us discuss each escape character one by one:

- `\n` it is used for providing **new line**

```
print("hello\npython") # will print result as follows
hello
python
```

- `\\` it is used to represent **backslash**. It returns one single backslash.

```
print("hello\\how are you") # will print result as follows
hello\how are you
```

- `\'` it is used to print a Single Quote(').

```
print("It\'s very powerful") # will print result as follows
It's very powerful
```

- `\"` it is used to represent double Quote(")

```
print("We can represent Strings using \" ") # will print result as follows
We can represent Strings using "
```

- `\t` it is used to provide a **tab space**

```
print("Hello\tPython") # will print result as follows
Hello	Python
```

REPR()

The repr() function returns a printable representation of the given object.

```
numbers = [1, 2, 3, 4, 5]

# create a printable representation of the list
printable_numbers = repr(numbers)
print(printable_numbers)

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

Escape sequences can sometimes be printed as they are, without being treated as special characters. To achieve this, you can use the `repr()` function or the `r` / `R` prefix.

Example using `repr()`:

```
str = "Hello\tPython\nPython is very interesting"
print(str) # will print result as follows
Hello        Python
Python is very interesting

print(repr(str))# will print result as follows
'Hello\tPython\nPython is very interesting'
```

Example using `'r'` and `'R'`:

```
print(r"Hello\tPython\nPython is very interesting")  # will print result as follows
Hello\tPython\nPython is very interesting

print(R"Hello\tPython\nPython is very interesting") # will print result as follows
Hello\tPython\nPython is very interesting
```

repr() Parameters

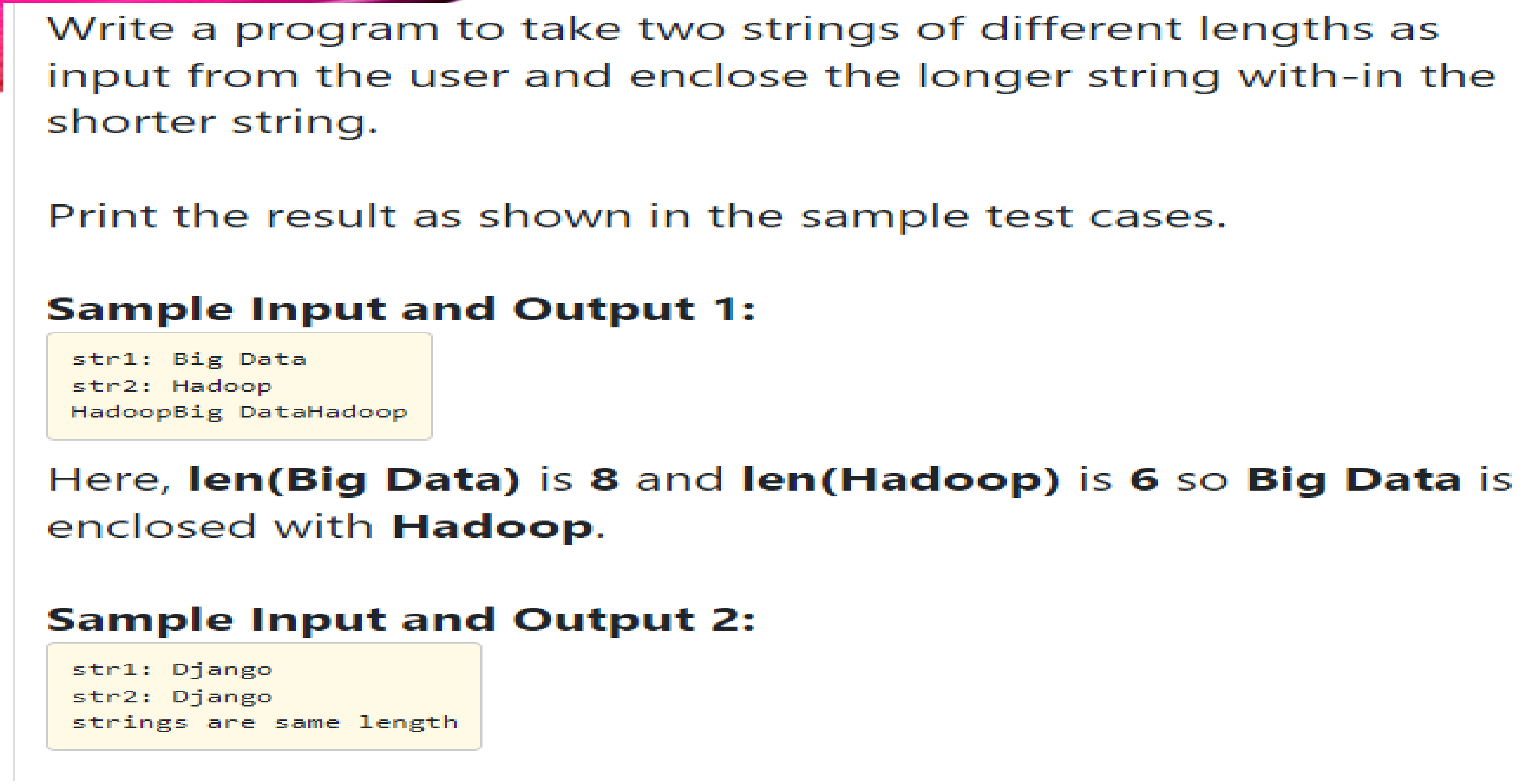
The repr() function takes a single parameter:

obj - the object whose printable representation has to be returned

```
repr(obj)

var = 'foo'

print(repr(var))
```



Write a program to take two strings of different lengths as input from the user and enclose the longer string with-in the shorter string.

Print the result as shown in the sample test cases.

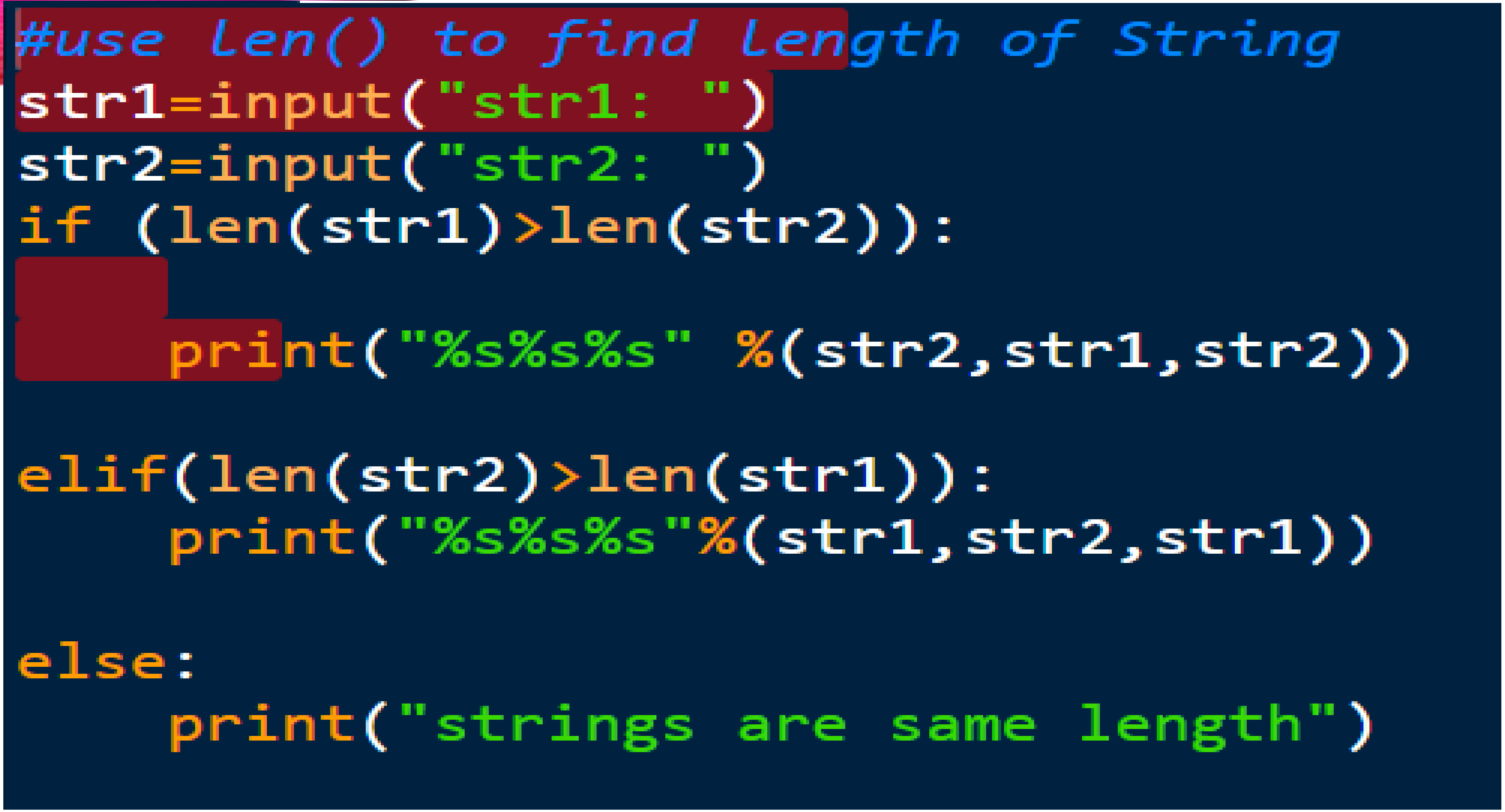
Sample Input and Output 1:

```
str1: Big Data
str2: Hadoop
HadoopBig DataHadoop
```

Here, **len(Big Data)** is **8** and **len(Hadoop)** is **6** so **Big Data** is enclosed with **Hadoop**.

Sample Input and Output 2:

```
str1: Django
str2: Django
strings are same length
```



```
#use len() to find length of String
str1=input("str1: ")
str2=input("str2: ")
if (len(str1)>len(str2)):

    print("%s%s%s" %(str2,str1,str2))

elif(len(str2)>len(str1)):
    print("%s%s%s"%(str1,str2,str1))


else:
    print("strings are same length")
```

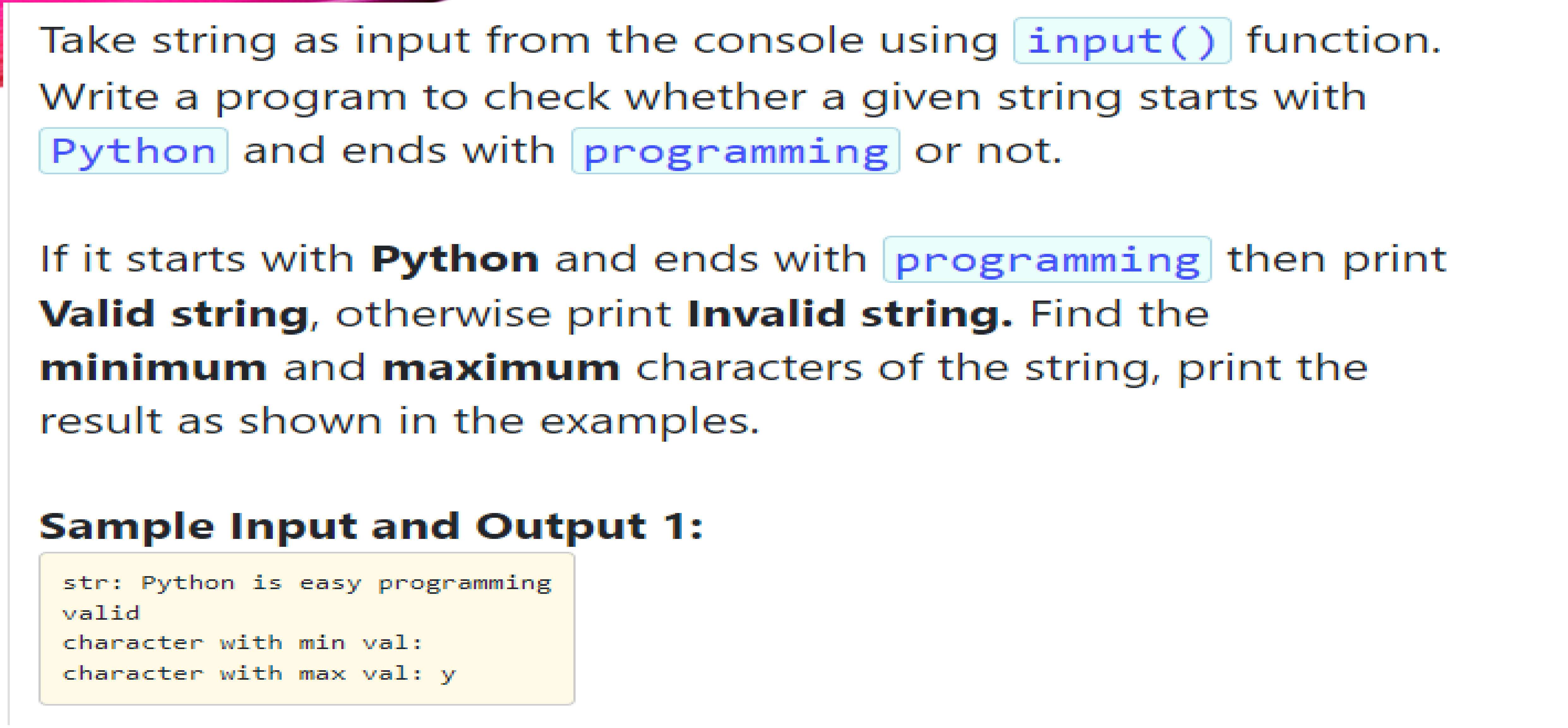
22) **min(a)** returns the minimum character in the string

```
a = "Hello Python"
print(min(a)) # will print result as follows# min(a) prints space because space is minimum value in Hello Python.
```

Here it returns **white space** because in the given string white space is the minimum of all characters.

23) **max(a)** returns the maximum character in the string.

```
print(max(a)) # will print result as follows
y
```

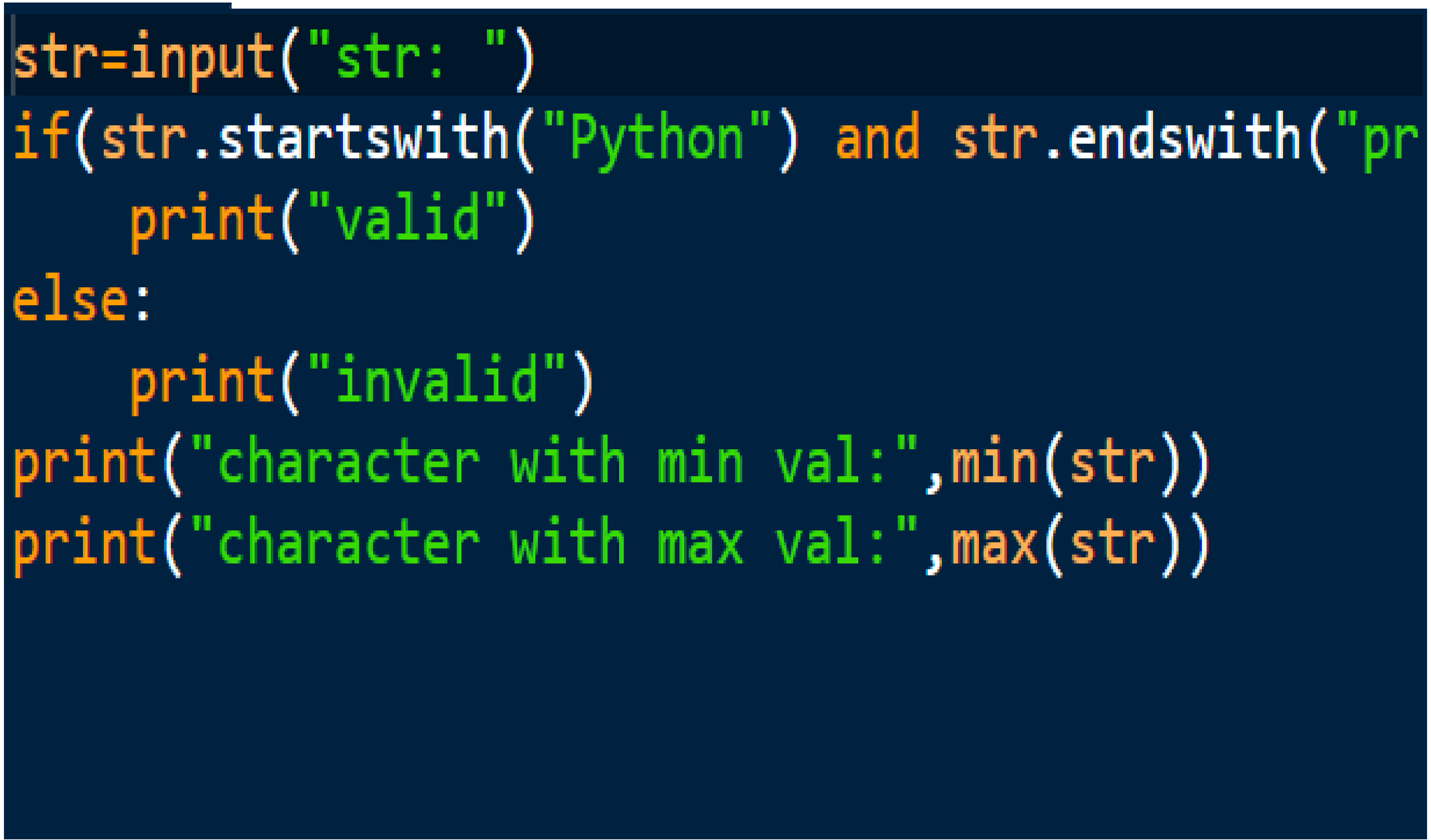


Take string as input from the console using `input()` function. Write a program to check whether a given string starts with `Python` and ends with `programming` or not.

If it starts with **Python** and ends with `programming` then print **Valid string**, otherwise print **Invalid string.** Find the **minimum** and **maximum** characters of the string, print the result as shown in the examples.

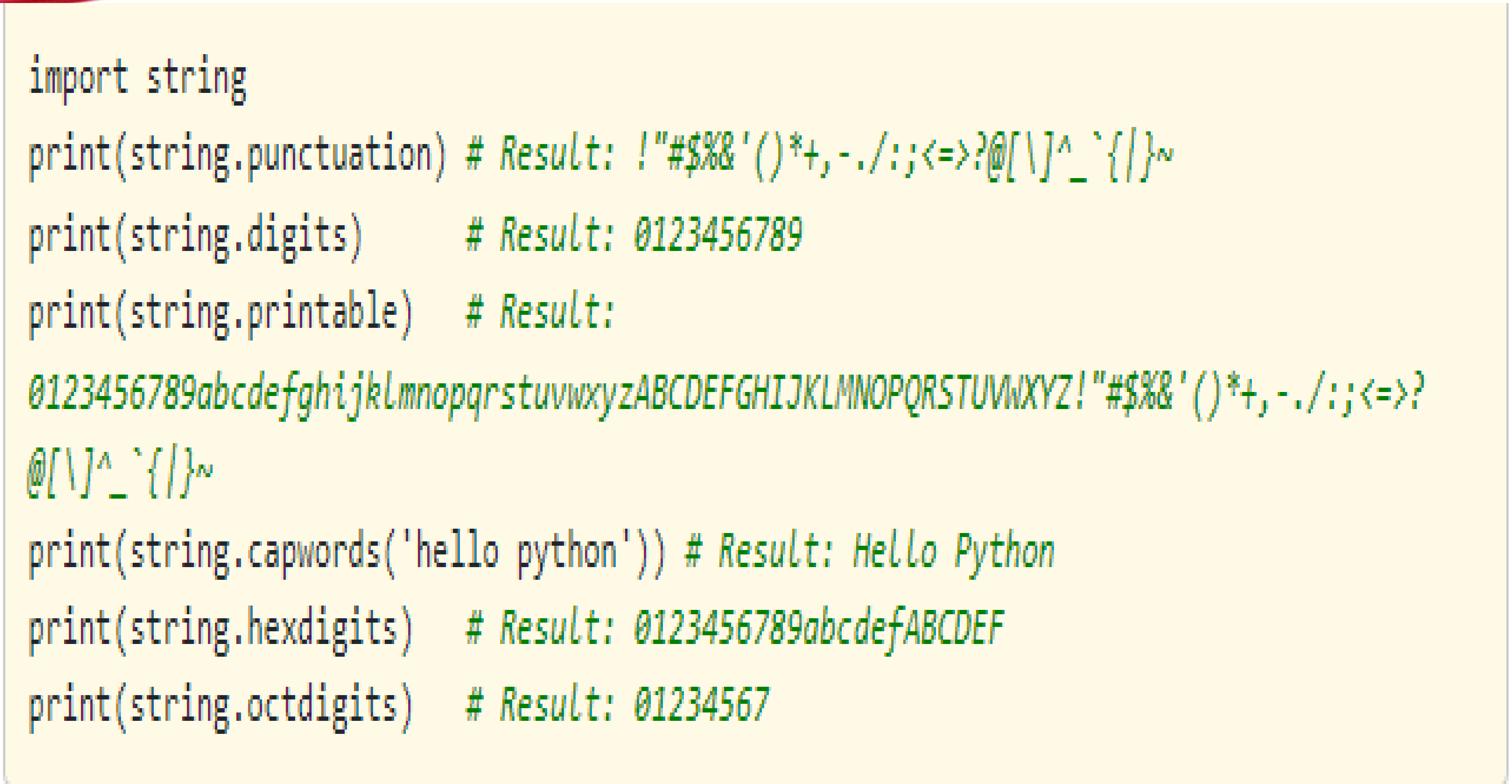
Sample Input and Output 1:

```
str: Python is easy programming
valid
character with min val:  
character with max val: y
```


```
str=input("str: ")
if(str.startswith("Python") and str.endswith("pr
    print("valid")
else:
    print("invalid")
print("character with min val:",min(str))
print("character with max val:",max(str))
```

MODULES

A **module** is a file containing **Python definitions**, **functions** and **statements**.Standard library of **Python** is extended as `modules`. To use modules, we need to **import** them. After importing a module, we can use its functions and variables in our code. Python offers numerous standard modules. We will learn about modules in detail later.



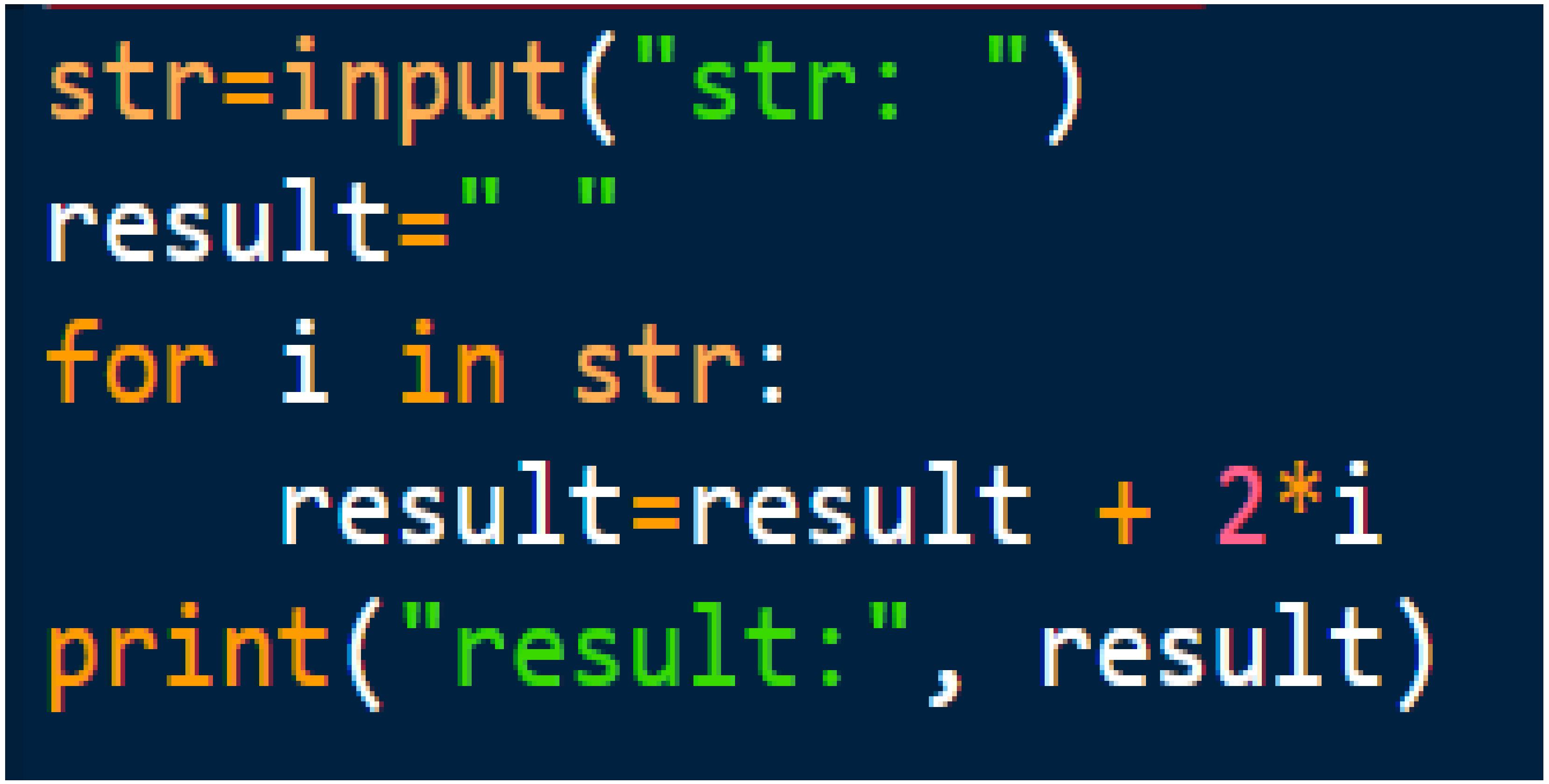
```
import string
print(string.punctuation) # Result: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
print(string.digits)      # Result: 0123456789
print(string.printable)   # Result:
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./:;<=>?
@[\]^_`{|}~
print(string.capwords('hello python')) # Result: Hello Python
print(string.hexdigits)   # Result: 0123456789abcdefABCDEF
print(string.octdigits)   # Result: 01234567
```

Write a program to print every character of the given input string twice.

Print the result to the console as shown in the sample test case.

Sample Input and Output:

```
str: Lists
result:  LLiissttss
```



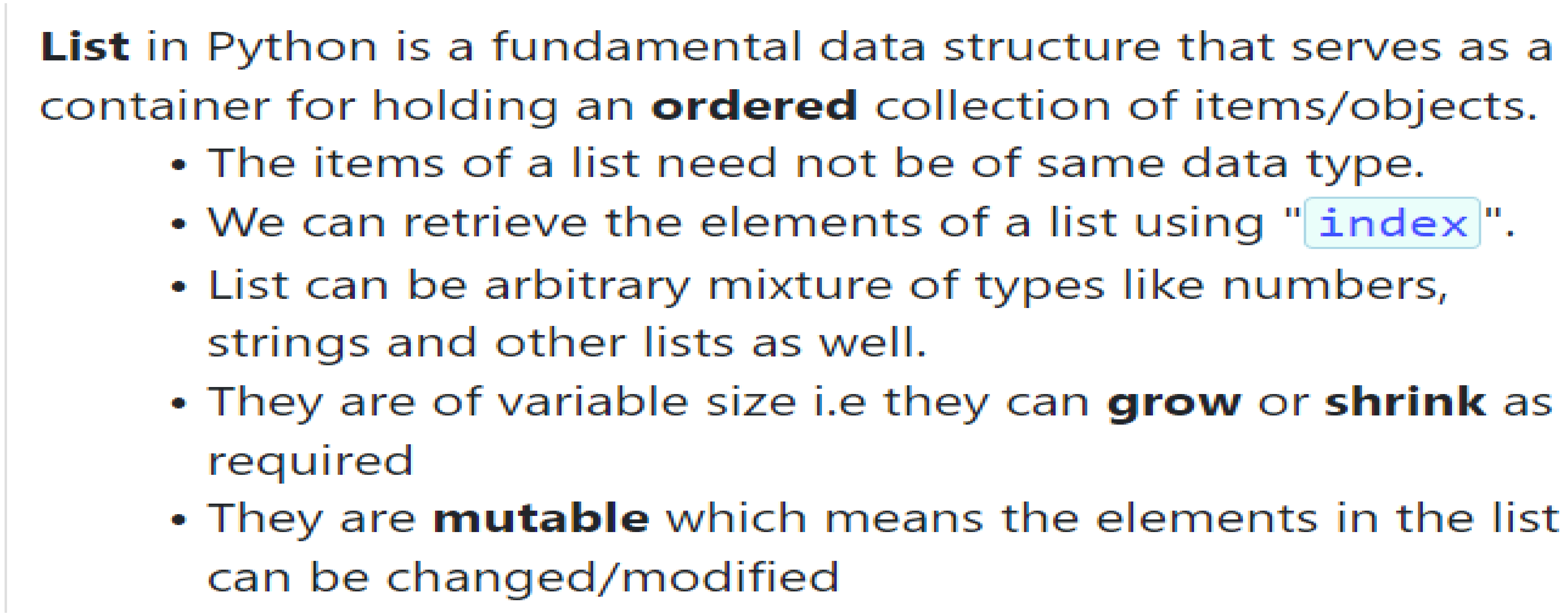
```
str=input("str: ")
result=" "
for i in str:
    result=result + 2*i
print("result:", result)
```

isprintable() - this method returns **True** if all the characters are printable, otherwise **False**..

The printable characters include **alphabets, digits, symbols, punctuation** and **white space**.

'`\n`' is not an alphabet character so `isprintable()` returns '**False**'.

INTRODUCTION TO LISTS



List in Python is a fundamental data structure that serves as a container for holding an **ordered** collection of items/objects.

- The items of a list need not be of same data type.
- We can retrieve the elements of a list using "`index`".
- List can be arbitrary mixture of types like numbers, strings and other lists as well.
- They are of variable size i.e they can **grow** or **shrink** as required
- They are **mutable** which means the elements in the list can be changed/modified

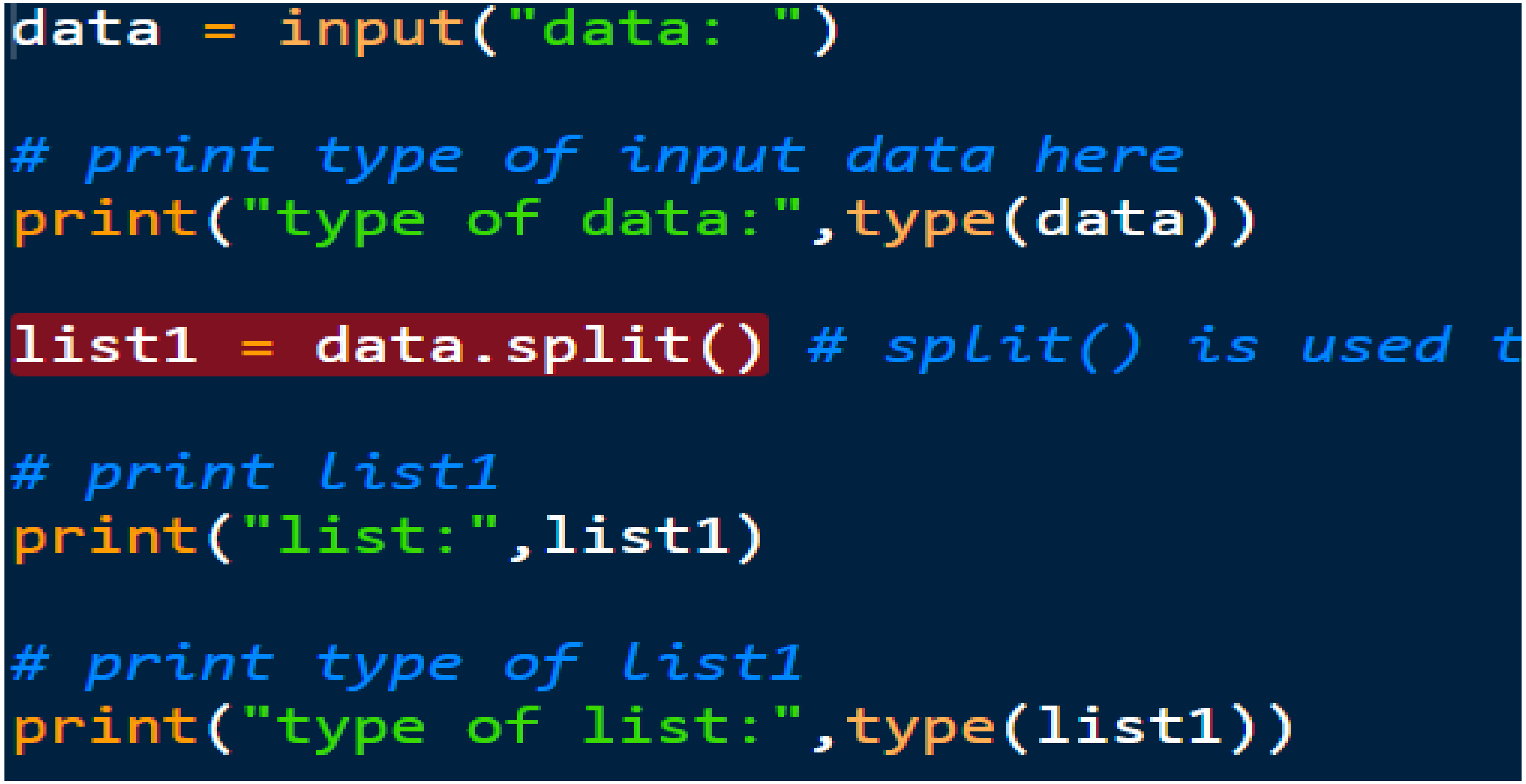
Note:

- We are using `split()` function of string, which converts a `string` into `list`.
- `split()` function takes `separator` as argument, if argument is not passed then it takes space as default separator.

Write a program to create a list with multiple string values and print the result as shown in the sample test case.

Sample Input and Output:

```
data: James John Jacob
type of data: <class 'str'>
list: ['James', 'John', 'Jacob']
type of list: <class 'list'>
```



```
data = input("data: ")

# print type of input data here
print("type of data:",type(data))

list1 = data.split() # split() is used t

# print list1
print("list:",list1)

# print type of list1
print("type of list:",type(list1))
```

Operations	Example	Description

Create a List	`a = [2 ,3, 4, 5, 6, 7, 8, 9, 10]`
	`print(a) # [2 ,3, 4, 5, 6, 7, 8, 9, 10]`
		Create a comma separated list of elements and assign to variable 'a'

Indexing	`print(a[0])     # 2`
	`print(a[-1])    # 10`
		Access the item at position '0'
		Access the last item using negative indexing

Slicing	`print(a[0:3]) # [2, 3, 4]`
	`print(a[0:])  # [2, 3, 4, 5, 6, 7, 8, 9, 10]`
		Print a part of list starting at index '0' till index '2'
		Print a part of the list starting at index '0' till the end of list

Concatenation	`b = [20, 30]`
	`print(a + b)  # [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30]`
		Concatenate two lists and display its output

Updating	`a[2] = 100`
	`print(a)    # [2, 3, 100, 5, 6, 7, 8, 9, 10]`
		Update the list element at index 2

`a = [1, 2, 3, 4, 5]`
`a[0:3] = [100, 100, 100]`
`print(a) # Output: [100, 100, 100, 4, 5]`
	Changing multiple elements

`a = [1, 2, 3, 4, 5]`
`a[0:3] = [ ]`
`print(a) # Output: [4, 5]`
	Certain elements from a list can also be removed by assigning an empty list to them

`a = [1, 2, 3, 4, 5]`
`a[0:0] = [20, 30, 45]`
`print(a) # Output: [20, 30, 45, 1, 2, 3, 4, 5]`
	The elements can be inserted into a list by squeezing them into an empty slice at the desired location

Create a list with the user-given inputs. Take an index value n from the user. Write a program to update a list with user given element based on index n, if the index is not in the range, print the error message as shown in the sample test cases.

```
Sample Input and Output 1:
data: James,John,Jacob
before updation: ['James', 'John', 'Jacob']
index: -2
element: Oliver
after updation: ['James', 'Oliver', 'Jacob']

Sample Input and Output 2:
data: 10,20,54,26
before updation: ['10', '20', '54', '26']
index: 4
invalid
```

Membership	`5 in a        # True`
	`102 in a      # False`
		Returns True if element is present in list.
		
		Otherwise returns false.

Comparison	`a = [2, 3, 4, 5, 6, 7, 8, 9, 10]`
	`b = [2, 3, 4]`
	`a == b        # False`
		Returns True if all elements in both lists are same.
		
		Otherwise returns false

Repetition	`a = [1, 2, 3]`
	`print(a * 3)   # [1, 2, 3, 1, 2, 3, 1, 2, 3]`
		Here the * operator repeats a list for the given number of times.

Write a program that takes input for two comma-separated lists, along with a number. The program should repeat and print each list the specified number of times and then combine the lists and display the final merged list as shown in the sample test cases.

Sample Input and Output 1:

```
data1: Python,Java
data2: Perl,Swift
num: 2
['Python', 'Java', 'Python', 'Java']
['Perl', 'Swift', 'Perl', 'Swift']
extending list1 with list2: ['Python', 'Java', 'Perl', 'Swift']
```

Sample Input and Output 2:

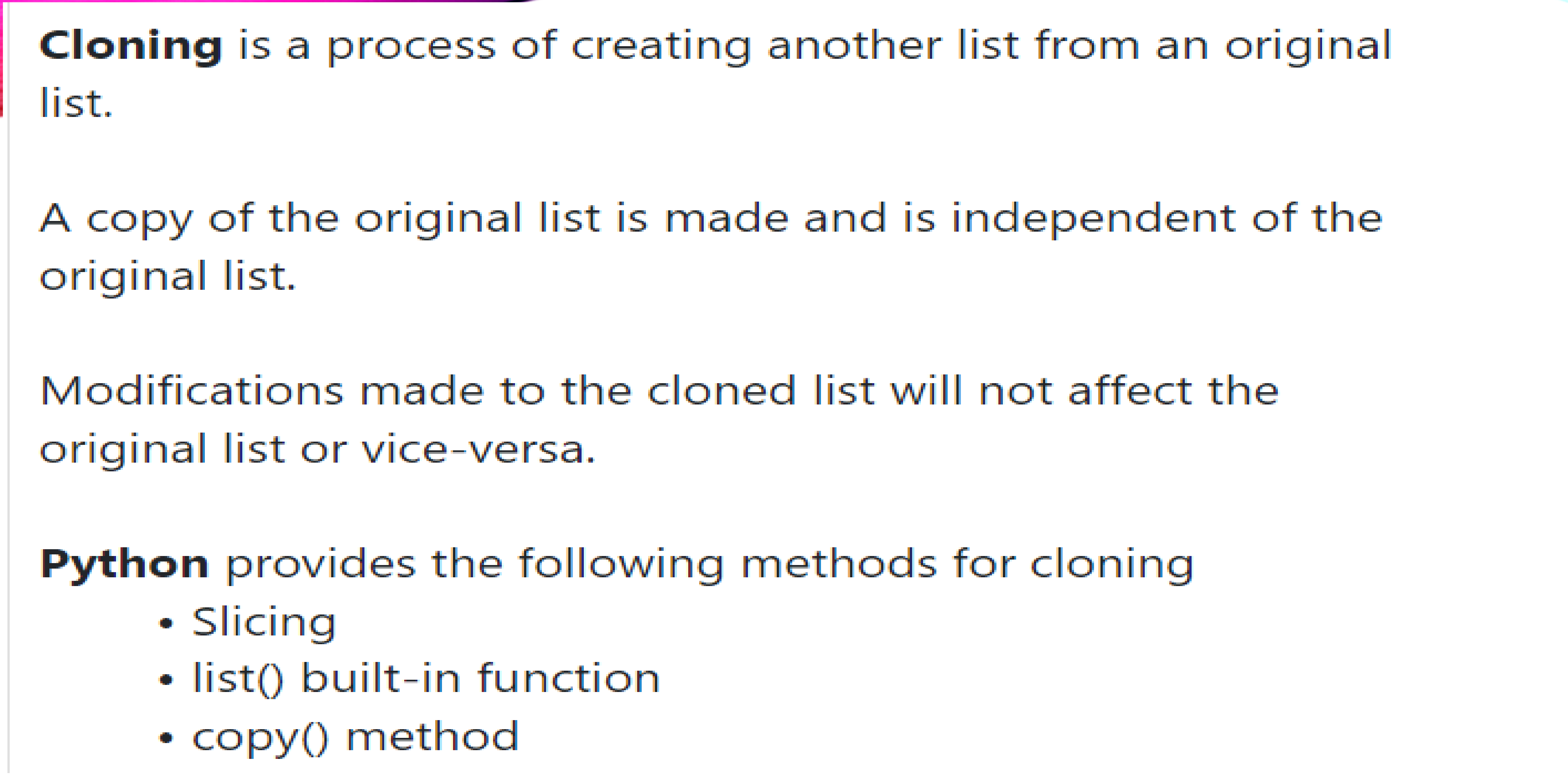
```
data1: 10,20,30
data2: 40,50
num: 3
['10', '20', '30', '10', '20', '30', '10', '20', '30']
['40', '50', '40', '50', '40', '50']
extending list1 with list2: ['10', '20', '30', '40', '50']
```

```
data1 = input("data1: ")
list1 = data1.split(",")
data2 = input("data2: ")
list2 = data2.split(",")
num = int(input("num: "))
print(list1 * num)
print(list2 * num)
list1.extend(list2)
print("extending list1 with list2:", list1)
```

Aliasing occurs when a list is assigned to another variable, making both point to the same memory location. This leads to the same list being referenced by different names, causing changes from one reference to affect the other.

For example,

```
a = [1, 2, 3, 4, 5, 6]
b = a
print(a)  # Output: [1, 2, 3, 4, 5, 6]
print(b)  # Output: [1, 2, 3, 4, 5, 6]
b is a    # Returns: True
a is b    # Returns: True
a[0] = 100
print(a)  # Output: [100, 2, 3, 4, 5, 6]
print(b)  # Output: [100, 2, 3, 4, 5, 6]
```



Cloning is a process of creating another list from an original list.

A copy of the original list is made and is independent of the original list.

Modifications made to the cloned list will not affect the original list or vice-versa.

Python provides the following methods for cloning

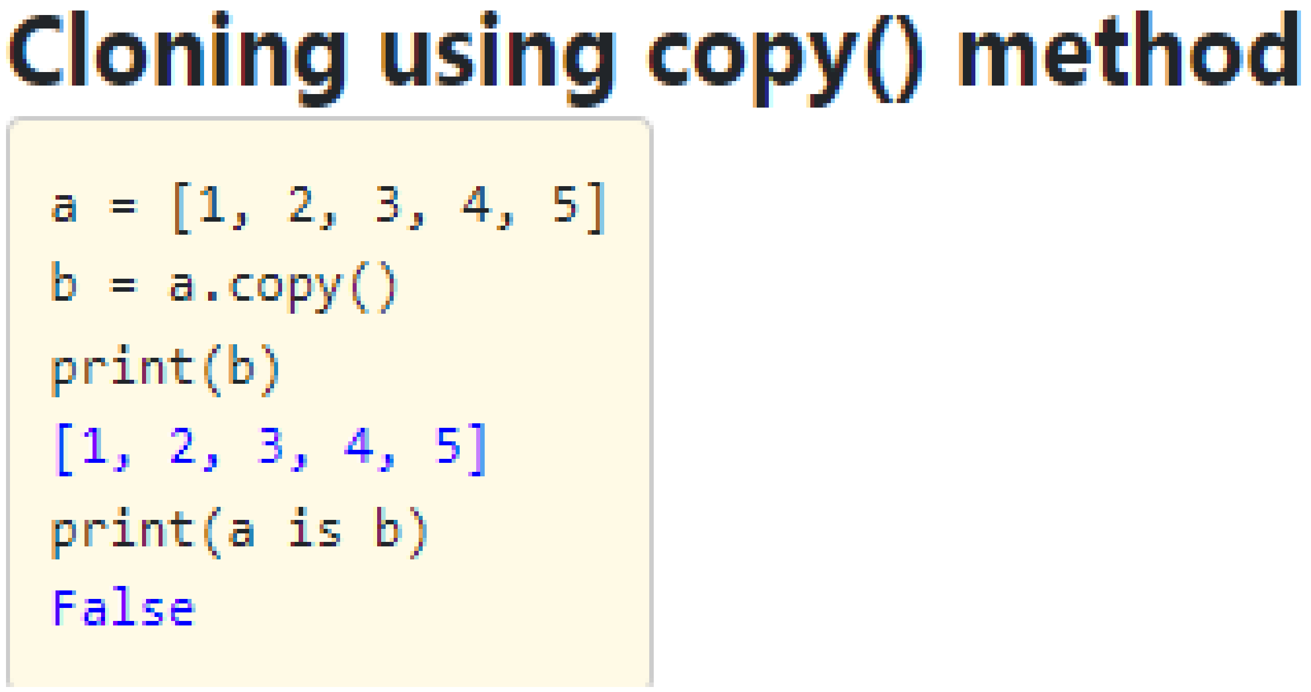
- Slicing
- list() built-in function
- copy() method

Cloning using Slicing

```
a = [1, 2, 3, 4, 5]
b = a[:]
print(b)
[1, 2, 3, 4, 5]
print(a is b)
False
```

Cloning using List() function

```
a = [1, 2, 3, 4, 5]
b = list(a)
print(b)
[1, 2, 3, 4, 5]
print(a is b)
False
a[0] = 100
print(a)
[100, 2, 3, 4, 5]
print(b)
[1, 2, 3, 4, 5]
```

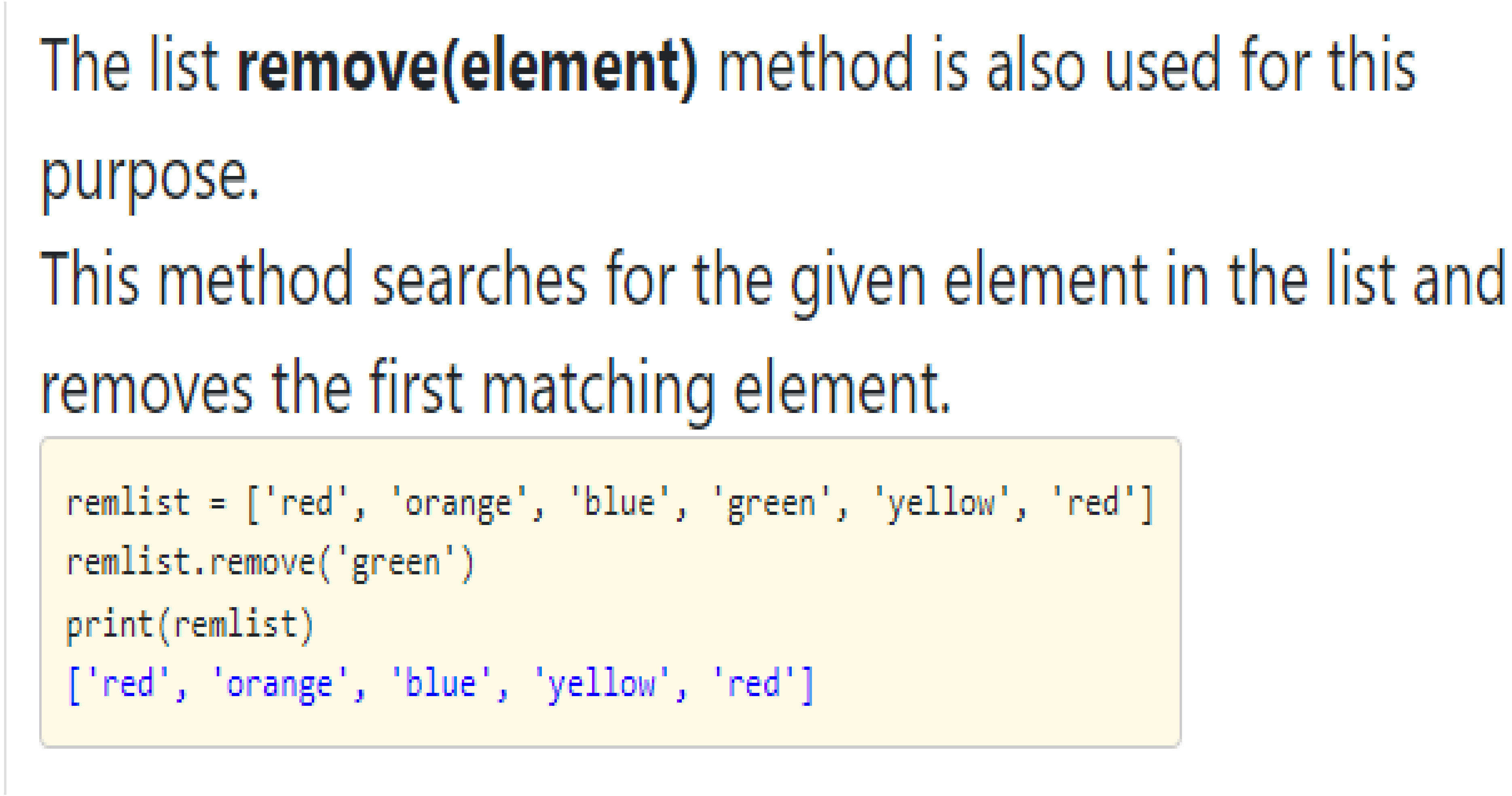


Cloning using copy() method

```
a = [1, 2, 3, 4, 5]
b = a.copy()
print(b)
[1, 2, 3, 4, 5]
print(a is b)
False
```

One or more elements of a list can be deleted using the keyword **del**. This can as well delete the entire list.

```
dlist = ['red', 'orange', 'blue', 'green', 'yellow', 'red']
print(dlist)
['red', 'orange', 'blue', 'green', 'yellow', 'red']
del dlist[5]
print(dlist)
['red', 'orange', 'blue', 'green', 'yellow']
del dlist[2:]
print(dlist)
['red', 'orange']
del dlist
```



The list **remove(element)** method is also used for this purpose.
This method searches for the given element in the list and removes the first matching element.

```
remlist = ['red', 'orange', 'blue', 'green', 'yellow', 'red']
remlist.remove('green')
print(remlist)
['red', 'orange', 'blue', 'yellow', 'red']
```

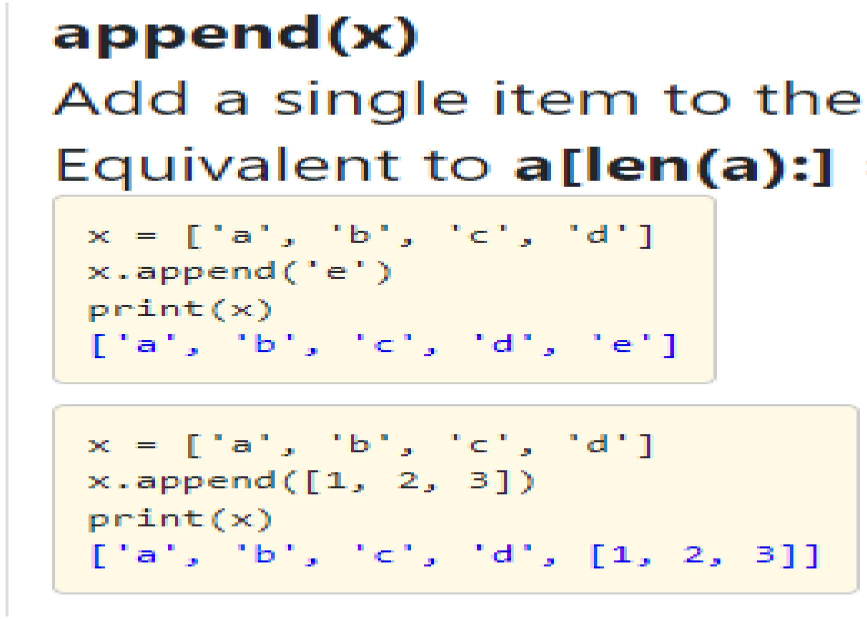
The **pop(index)** method removes and returns the item at index, if index is provided.
If index is not provided as an argument, then pop() method removes and returns the last element.

```
plist = ['red', 'orange', 'blue', 'green', 'yellow', 'cyan']
elem = plist.pop()
print(elem)
'cyan'
elem = plist.pop(-1)
print(elem)
'yellow'
print(plist)
['red', 'orange', 'blue', 'green']
```

The **clear()** method is to used to clear the contents of a list and make it empty.

```
plist.clear()
print(plist)
[]
```

LIST METHODS



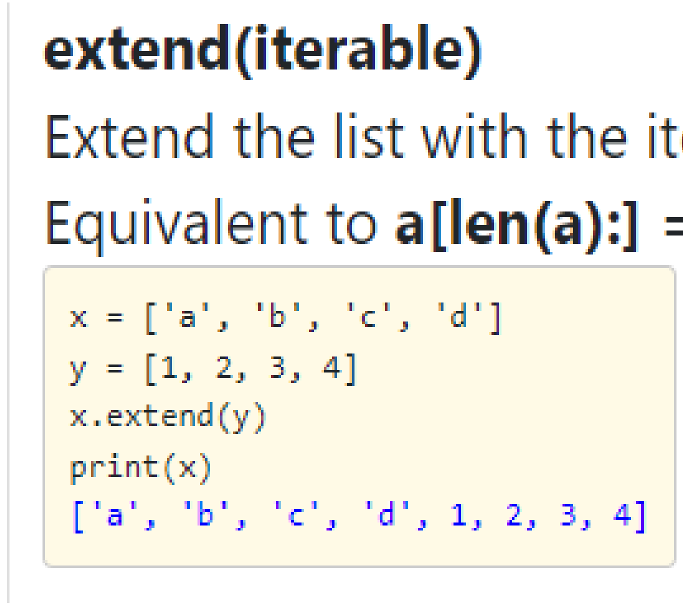
append(x)

Add a single item to the end of the list

Equivalent to **a[len(a):] = [x]**

```
x = ['a', 'b', 'c', 'd']
x.append('e')
print(x)
['a', 'b', 'c', 'd', 'e']
```

```
x = ['a', 'b', 'c', 'd']
x.append([1, 2, 3])
print(x)
['a', 'b', 'c', 'd', [1, 2, 3]]
```



extend(iterable)

Extend the list with the items of another list or iterable

Equivalent to **a[len(a):] = iterable**

```
x = ['a', 'b', 'c', 'd']
y = [1, 2, 3, 4]
x.extend(y)
print(x)
['a', 'b', 'c', 'd', 1, 2, 3, 4]
```

insert(index, item)

Insert an item at a position before the element given by the index.

a.insert(0, x) inserts at the front of the list.

a.insert(len(a), x) is equivalent to **a.append(x)** which inserts an element at the end of list.

```
x = ['a', 'b', 'c', 'd']
x.insert(0, 1)
print(x)
[1, 'a', 'b', 'c', 'd']
x.insert(len(x), 'e')
print(x)
[1, 'a', 'b', 'c', 'd', 'e']
```

remove(element)

Remove the first item in the list whose value is element

Error if the item doesn't exist with the value element in the list

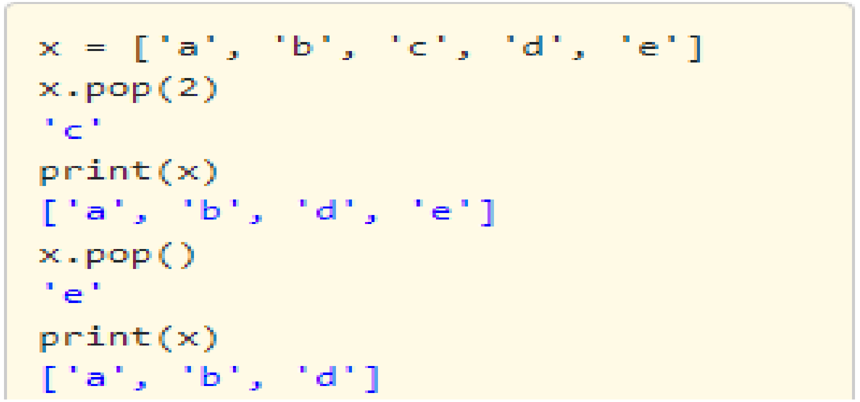
```
L1 = [1, 'a', 'b', 'c', 'd', 'e']
L1.remove(1)
print(L1)
['a', 'b', 'c', 'd', 'e']
L1.remove('f')
Traceback (most recent call last):
  File "", line 1, in
ValueError: list.remove(x): x not in list
```

pop(), pop(index)

Removes an item at the given position specified by index

Removes and returns the last element if the index is not specified

If an invalid index is specified, then an IndexError is thrown



```
x = ['a', 'b', 'c', 'd', 'e']
x.pop(2)
'c'
print(x)
['a', 'b', 'd', 'e']
x.pop()
'e'
print(x)
['a', 'b', 'd']
```

count(x)

Return the number of times item x appears in the list

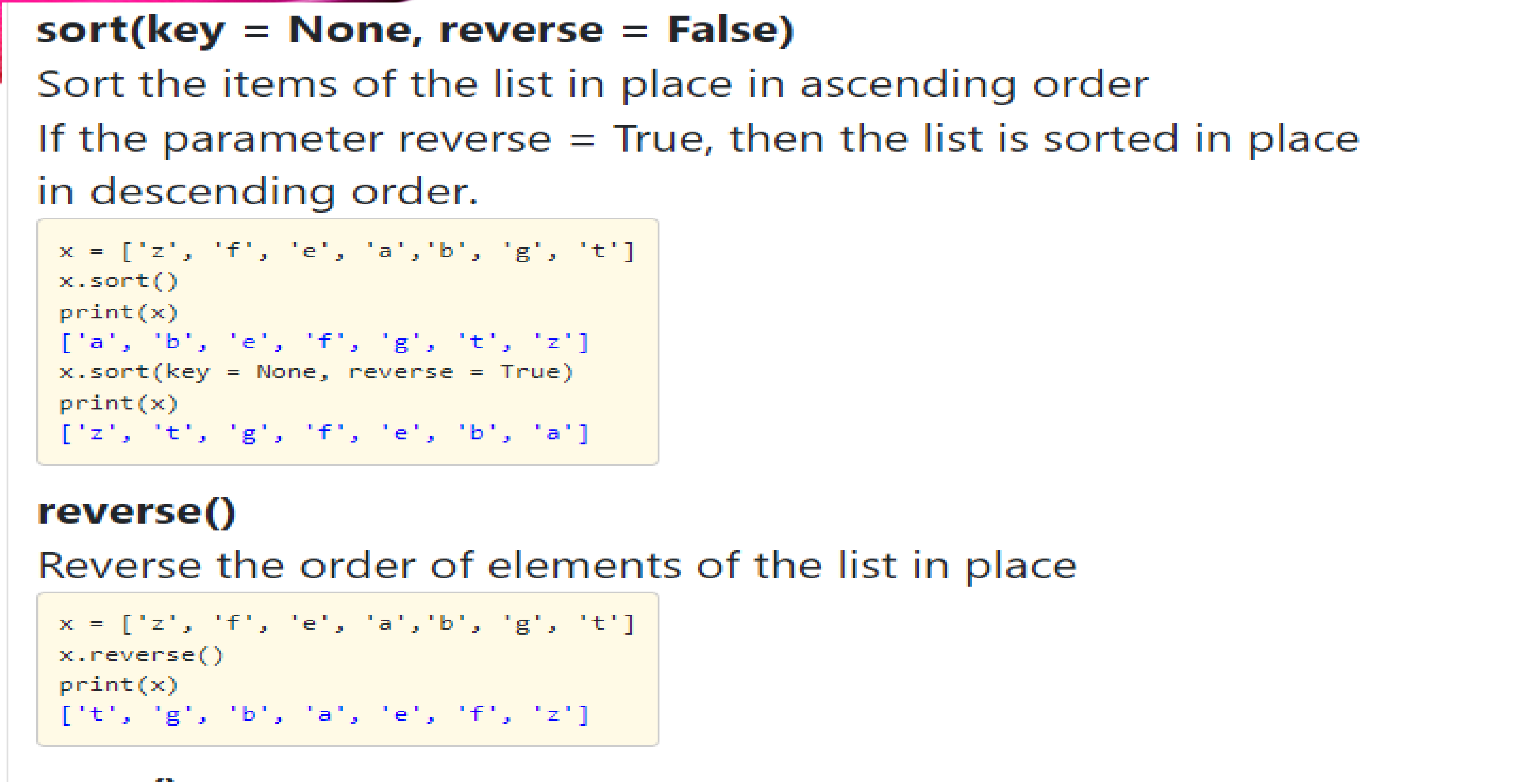
```
x = ['a', 'b', 'a', 'c', 'd', 'e', 'a']
x.count('a')
3
```

sort(key = None, reverse = False)

Sort the items of the list in place in ascending order

If the parameter reverse = True, then the list is sorted in place in descending order.

```
x = ['z', 'f', 'e', 'a','b', 'g', 't']
x.sort()
print(x)
['a', 'b', 'e', 'f', 'g', 't', 'z']
x.sort(key = None, reverse = True)
print(x)
['z', 't', 'g', 'f', 'e', 'b', 'a']
```



sort(key = None, reverse = False)

Sort the items of the list in place in ascending order

If the parameter reverse = True, then the list is sorted in place in descending order.

```
x = ['z', 'f', 'e', 'a','b', 'g', 't']
x.sort()
print(x)
['a', 'b', 'e', 'f', 'g', 't', 'z']
x.sort(key = None, reverse = True)
print(x)
['z', 't', 'g', 'f', 'e', 'b', 'a']
```

reverse()

Reverse the order of elements of the list in place

```
x = ['z', 'f', 'e', 'a','b', 'g', 't']
x.reverse()
print(x)
['t', 'g', 'b', 'a', 'e', 'f', 'z']
```

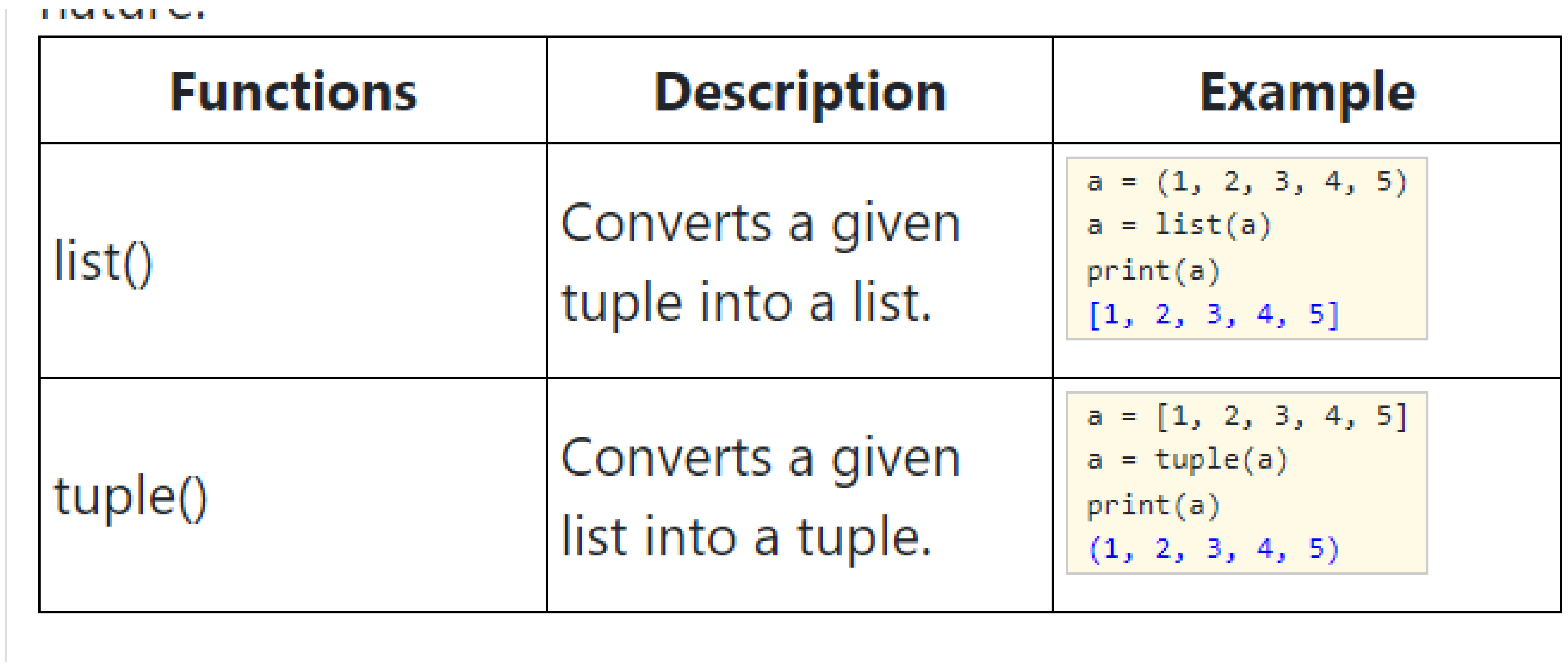
copy()

Return the shallow copy of the list

Equivalent to **a[:]**

```
x = ['z', 'f', 'e', 'a','b', 'g', 't']
y = x.copy()
print(y)
['z', 'f', 'e', 'a', 'b', 'g', 't']
print(x is y)
False
```

INTRODUCTION TO TUPLES



nature.

Functions	Description	Example

list()	Converts a given tuple into a list.	`a = (1, 2, 3, 4, 5)`
		`a = list(a)`
		`print(a)`
		`[1, 2, 3, 4, 5]`

tuple()	Converts a given list into a tuple.	`a = [1, 2, 3, 4, 5]`
		`a = tuple(a)`
		`print(a)`
		`(1, 2, 3, 4, 5)`

Benefits of Tuple:

- Tuples are faster than lists.
- Since a tuple is immutable, it is preferred over a list to have the data write-protected.
- Tuples can be used as keys in dictionaries unlike lists.

BUILT-IN TUPLE FUNCTIONS

- **all()**
  Returns **True** if all the items in the tuple has a **True** value.

```
print(all((' ', ',', '1', '2'))) # Result: True
print(all((False,' ', ',', '1'))) # Result: False
```

- **any()**
  Returns **True** if atleast one item in the tuple has a **True** value.

```
print(any((' ', ',', '1', '2'))) # Result: True
print(any((False,' ', ',', '1', '2'))) # Result: True
print(any((False, False, False))) # Result: False
```

- **enumerate()**
  Returns an enumerate object consisting of the **index** and the **value** of all items of a tuple as pairs.

```
x = (1, 2, 3, 4, 5, 6)
print(tuple(enumerate(x)))
((0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)) # Result
```

- **len()**
  It calculates the **length** of the tuple i.e., the number of elements.

```
x = (1, 2, 3, 4, 5, 6)
print(len(x)) # Result: 6
```

- **max()**
  It returns the item with the **highest value** in the tuple.

```
print(max((1, 2, 3, 4, 5, 6))) # Result: 6
```

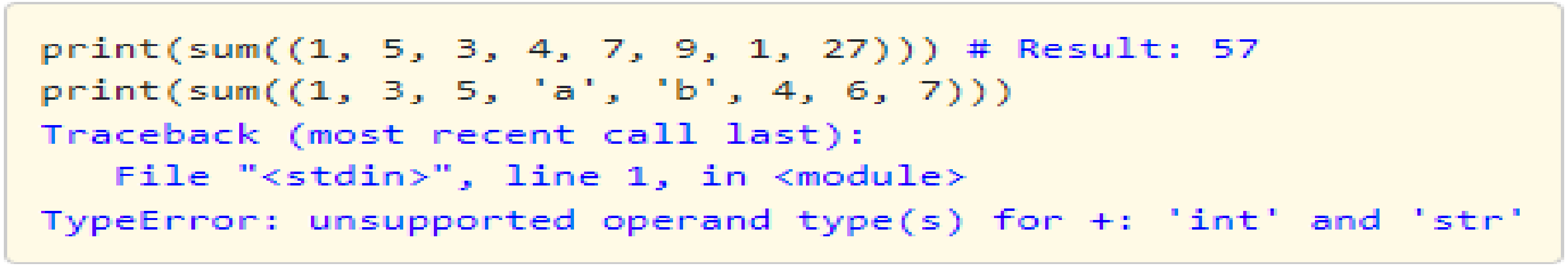
- **min()**
  It returns the item with the **lowest value** in the tuple.

```
print(min((1, 2, 3, 4, 5, 6))) # Result: 1
```

- **sorted()**
  It returns a **sorted** result of the tuple as a list, with the original tuple unchanged.

```
origtup = (1, 5, 3, 4, 7, 9, 1, 27)
sorttup = sorted(origtup)
print(sorttup) # Result: [1, 1, 3, 4, 5, 7, 9, 27]
print(origtup) # Result: (1, 5, 3, 4, 7, 9, 1, 27)
```

- **sum()**
  It returns the **sum** of all elements in the tuple. It works only on numeric values in the tuple and will error out if the tuple contains any other type of data.



```
print(sum((1, 5, 3, 4, 7, 9, 1, 27))) # Result: 57
print(sum((1, 3, 5, 'a', 'b', 4, 6, 7)))
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

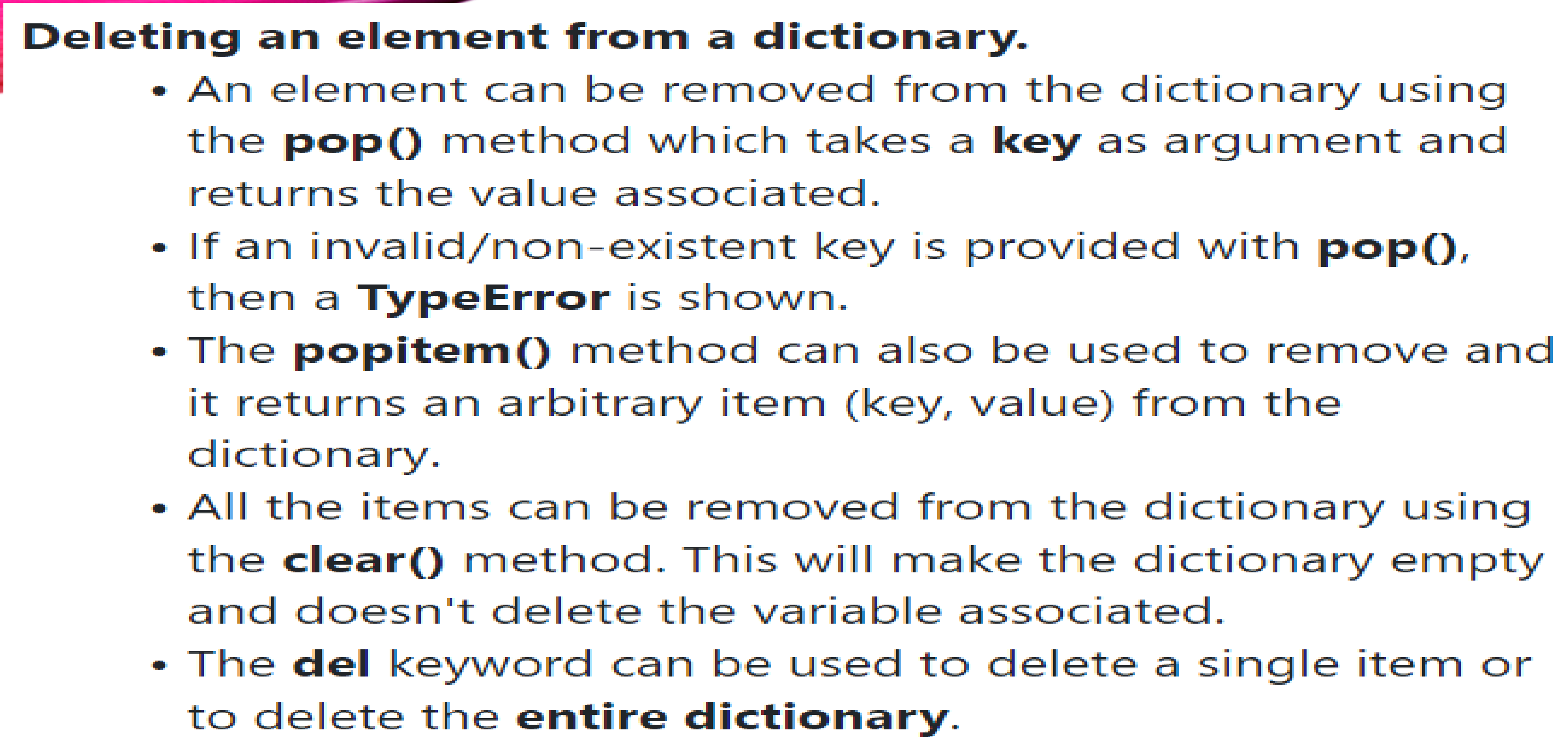
BASIC DICTIONARY OPERATIONS

1. **Creating a dictionary:**

```
dict1 = {"name":"Jay", "number":514, "age":12}
print(dict1)               #Result : {'name': 'Jay', 'number': 514, 'age': 12}
```

4. **We can change (update) the values in a dictionary.**

```
dict1['name'] = "Krithika"
print(dict1)          #{'name':'Krithika' ,'number':514, 'age':12}
```



Deleting an element from a dictionary.

- An element can be removed from the dictionary using the **pop()** method which takes a **key** as argument and returns the value associated.
- If an invalid/non-existent key is provided with **pop()**, then a **TypeError** is shown.
- The **popitem()** method can also be used to remove and it returns an arbitrary item (key, value) from the dictionary.
- All the items can be removed from the dictionary using the **clear()** method. This will make the dictionary empty and doesn't delete the variable associated.
- The **del** keyword can be used to delete a single item or to delete the **entire dictionary**.

Let's consider an example:

```
fruits = {1: 'apple', 2: 'orange', 3: 'mango', 4: 'grapes'}
print(type(fruits)) # Result: <type 'dict'>
print(fruits.pop(4)) #Result: grapes
print(fruits) # Result: {1: 'apple', 2: 'orange', 3: 'mango'}
print(fruits.popitem()) # Result: (1, 'apple')
print(fruits) # Result: {2: 'orange', 3: 'mango'}
del fruits[3]
print(fruits) # Result: {2: 'orange'}
fruits.clear()
print(fruits) # Result: {}
del fruits
print(fruits)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fruits' is not defined
```

The following table illustrates the dictionary methods.

Method	Example	Description

clear()	`emp = {'Name': Anil, 'Age':26, 'Exp':5}`
	`print(len(emp)) # Result: 3`
	`emp.clear()`
	`print(len(emp)) # Result: 0`
		Removes all elements from a dictionary.

copy()	`fruits = {'apple':'red', 'orange':'orange', 'mango':'yellow'}`
	`dict2 = fruits.copy()`
	`print(dict2)`
	`# Result: {'orange': 'orange', 'mango': 'yellow', 'apple': 'red'}`
		Returns a shallow copy of dictionary

fromkeys(sequence, value)	`Seq = ('one', 'two', 'three')` `dict = dict.fromkeys(Seq,10)` `print(dict)` `# Result: {'three': 10, 'two': 10, 'one': 10}`	Creates a new dictionary using elements from sequence as keys and values set to value
get(key, default = None)	`dict = {'cyan':1, 'violet':2, 'green':3}` `print(“Value : %s “ % dict.get('violet')) # Result: Value : 2` `print((“Value: %s“ %dict.get('red')) # Result: Value: None`	Returns a value for the key if found, else return the default which is None
items()	`dict = {'cyan':1, 'violet':2, 'green':3}` `print(dict.items())` `# Result: [('cyan', 1), ('green', 3), ('violet', 2)]`	Returns a list of (key,value) tuple pairs

keys()	`dict = {'cyan':1,`
	`'violet':2, 'green':3}`
	`print(dict.keys()) #`
	`Result: ['cyan', 'green',`
	`'violet']`
		Returns the keys of dictionary as a list

setdefault(key, default = None)	`dict1 = {'cyan':1,`
	`'violet':2, 'green':3}`
	`dict1.setdefault('violet',`
	`10) # Result: 2`
	`dict1.setdefault('red',`
	`10) # Result: 10`
	`dict1`
	`# Result: {'cyan': 1,`
	`'violet': 2, 'green': 3,`
	`'red': 10}`
		This method is similar to the method get(). It will set dict[key] = default, if key is not found

update()	`dict1 = {'cyan':1,`
	`'violet':2, 'green':3}`
	`dict2 = {'red':4,`
	`'white':5}`
	`dict1.update(dict2)`
	`print(dict1)`
	`# Result: {'cyan': 1,`
	`'violet': 2, 'green': 3,`
	`'red': 4, 'white': 5}`
		Appends one dictionary(dict2 in example) key:value pairs to the calling dictionary(dict1 in example)

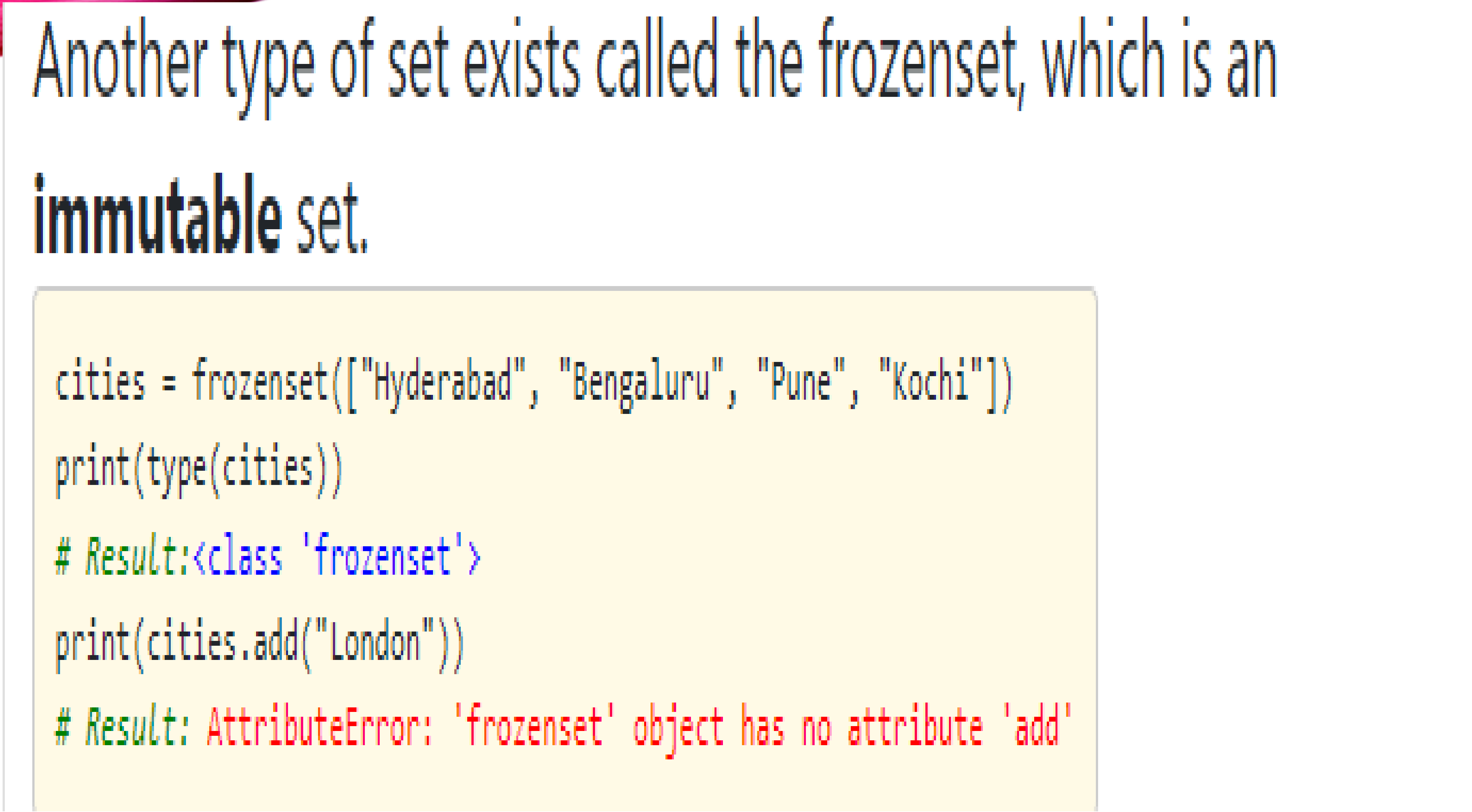
values()	`dict1 = {'cyan': 1, 'violet': 2, 'green': 3, 'red': 4}`
	`dict1.values()`
	`# Result: dict_values([1, 2, 3, 4])`
		Returns a list of dictionary's values

As we have seen that a set is mutable which can allow addition/deletion of elements from it.

The **elements in a set** should be immutable i.e., they cannot be modified/changed.

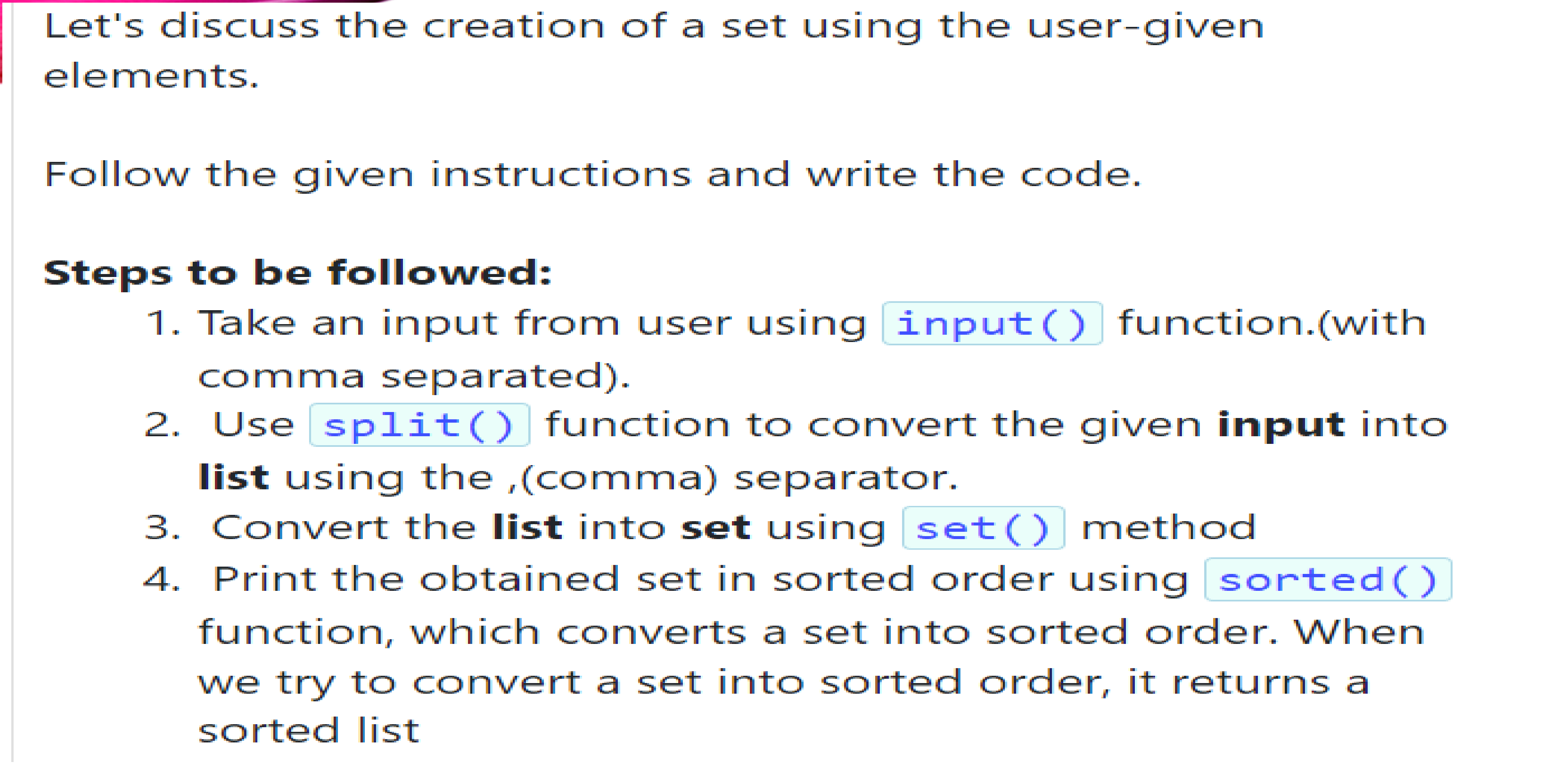
Hence, Lists cannot be used as elements of a set.

```
set1 = set((("C", "C++"), ("Java", "OOPS")))
print(type(set1))
# Result:<class 'set'>
set2 = set((["C", "C++"], ["Java", "OOPS", "Scala"]))
# Result:TypeError: unhashable type: 'list'
```



Another type of set exists called the frozenset, which is an **immutable** set.

```
cities = frozenset(["Hyderabad", "Bengaluru", "Pune", "Kochi"])
print(type(cities))
# Result:<class 'frozenset'>
print(cities.add("London"))
# Result: AttributeError: 'frozenset' object has no attribute 'add'
```



Let's discuss the creation of a set using the user-given elements.

Follow the given instructions and write the code.

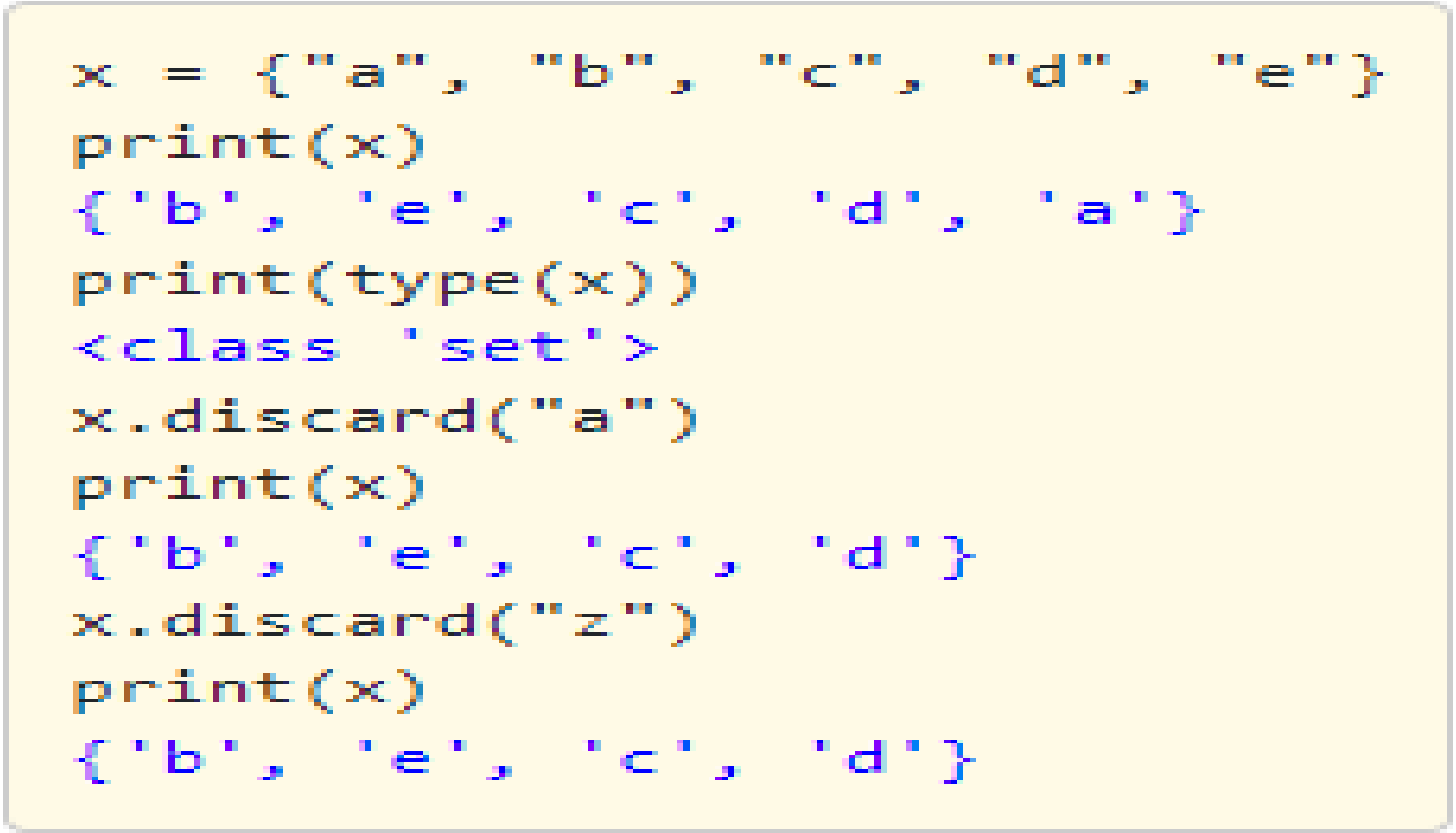
Steps to be followed:

1. Take an input from user using `input()` function.(with comma separated).
2. Use `split()` function to convert the given **input** into **list** using the ,(comma) separator.
3. Convert the **list** into **set** using `set()` method
4. Print the obtained set in sorted order using `sorted()` function, which converts a set into sorted order. When we try to convert a set into sorted order, it returns a sorted list

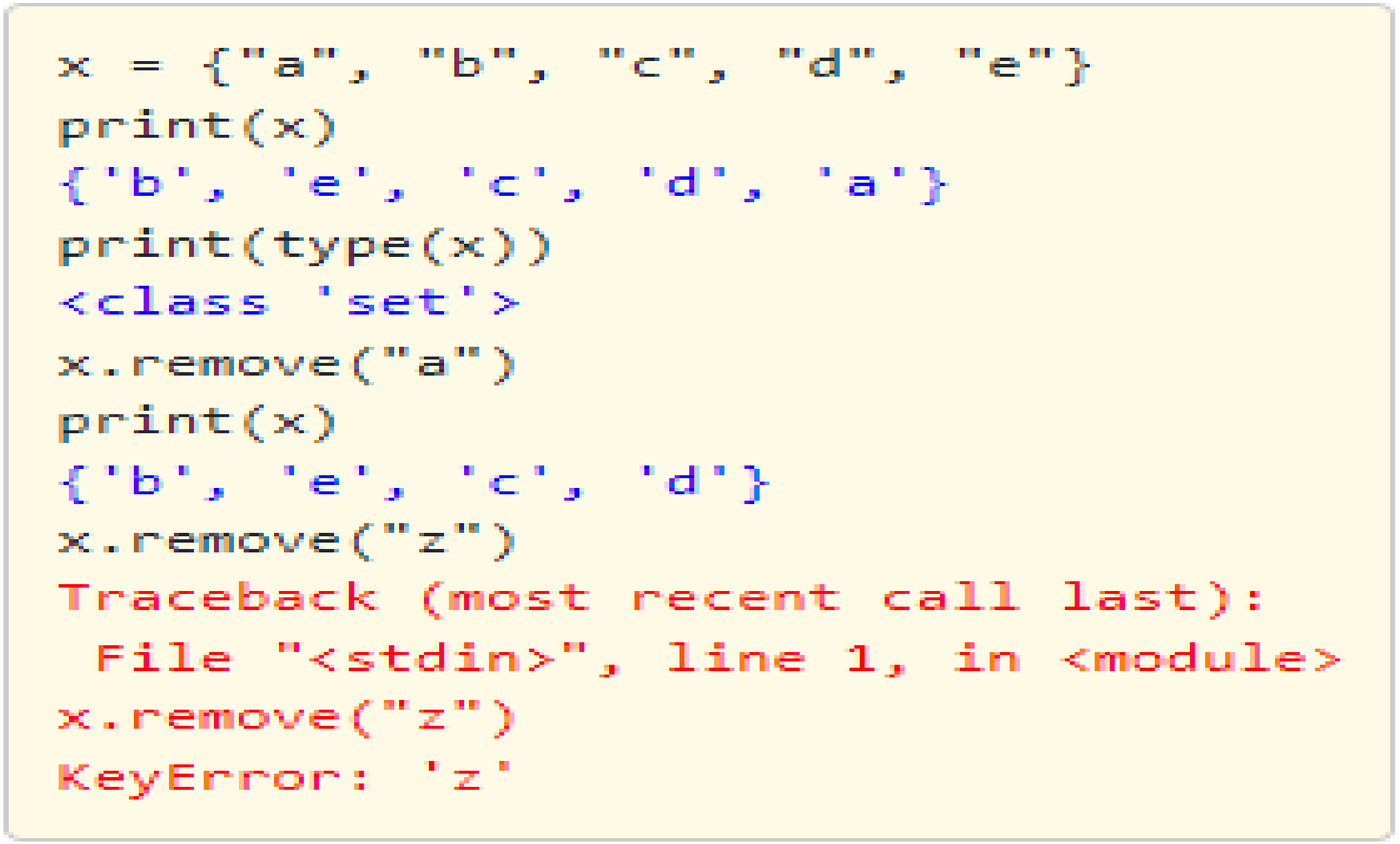
An element in a set can be removed using the **discard()** and **remove()** methods. The main difference between the two methods is

when an element doesn't exist then,

- **discard()** does nothing.
- **remove()** will raise an error.



```
x = {"a", "b", "c", "d", "e"}
print(x)
{'b', 'e', 'c', 'd', 'a'}
print(type(x))
<class 'set'>
x.discard("a")
print(x)
{'b', 'e', 'c', 'd'}
x.discard("z")
print(x)
{'b', 'e', 'c', 'd'}
```



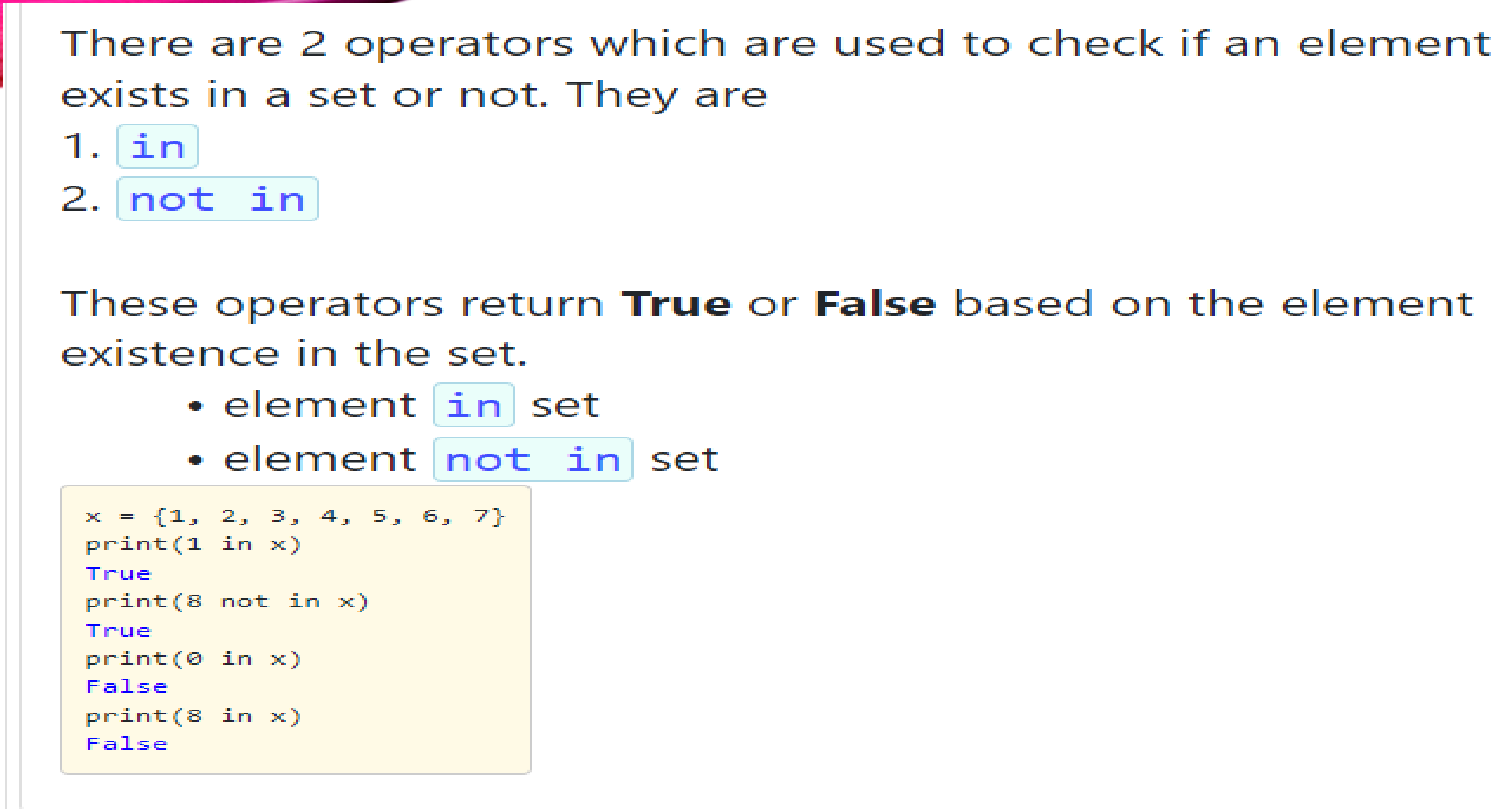
```
x = {"a", "b", "c", "d", "e"}
print(x)
{'b', 'e', 'c', 'd', 'a'}
print(type(x))
<class 'set'>
x.remove("a")
print(x)
{'b', 'e', 'c', 'd'}
x.remove("z")
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
x.remove("z")
KeyError: 'z'
```

The **pop()** method can also be used to remove an arbitrary element from the set.

This method returns the element removed.

Calling the **pop()** method on an empty set returns a `KeyError` : 'pop from an empty set'

```
x = {"a", "b", "c", "d", "e"}
print(x)
{'b', 'e', 'c', 'd', 'a'}
print(x.pop())
'b'
```



There are 2 operators which are used to check if an element exists in a set or not. They are

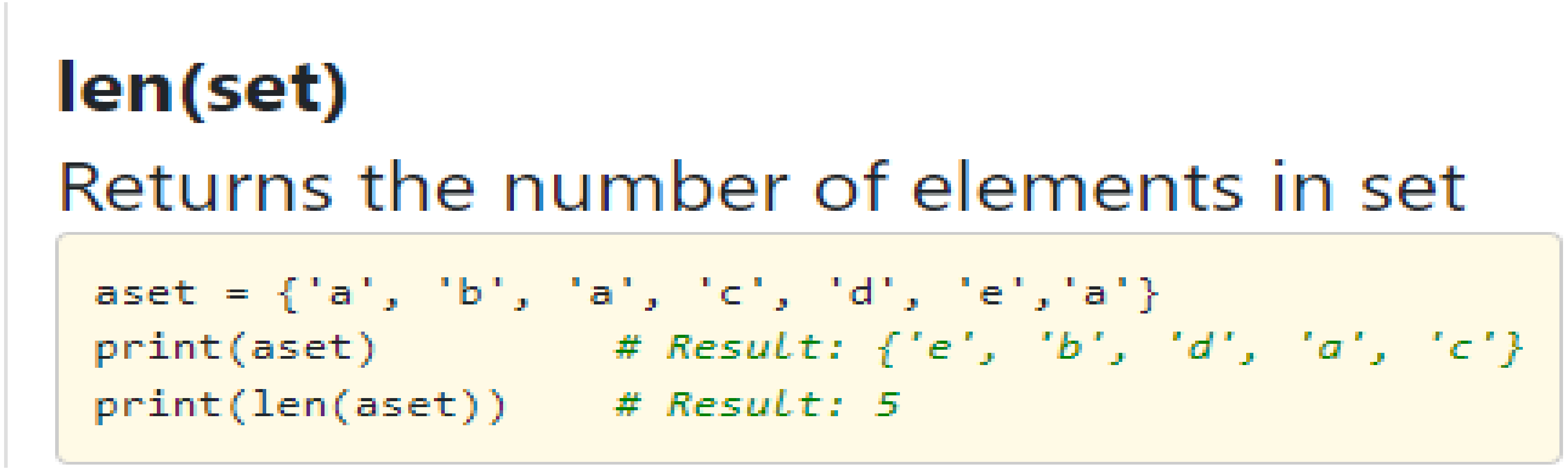
1. `in`
2. `not in`

These operators return **True** or **False** based on the element existence in the set.

- element `in` set
- element `not in` set

```
x = {1, 2, 3, 4, 5, 6, 7}
print(1 in x)
True
print(8 not in x)
True
print(0 in x)
False
print(8 in x)
False
```

BUILT-IN SET FUNCTIONS AND METHODS



len(set)

Returns the number of elements in set

```
aset = {'a', 'b', 'a', 'c', 'd', 'e','a'}
print(aset)              # Result: {'e', 'b', 'd', 'a', 'c'}
print(len(aset))         # Result: 5
```

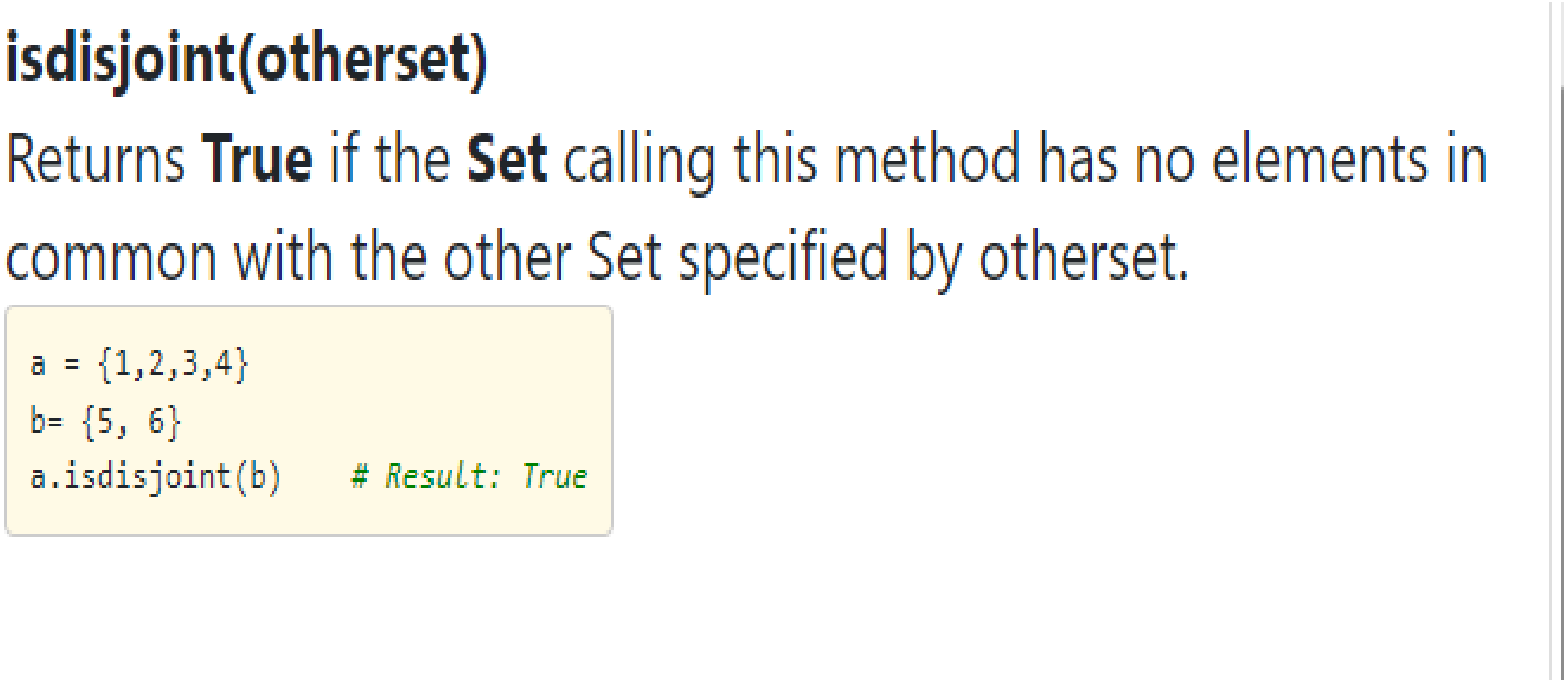
copy()

Return a new set with a shallow copy of set

```
x = {1, 2, 3, 4, 5, 6, 7}
print(len(x))       # Result: 7
y = x.copy()
print(y)            # Result: {1, 2, 3, 4, 5, 6, 7}
```

Copy is different from assignment. Copy will create a separate set object, assigns the reference to y and `=` will assign the reference to same set object to **y**

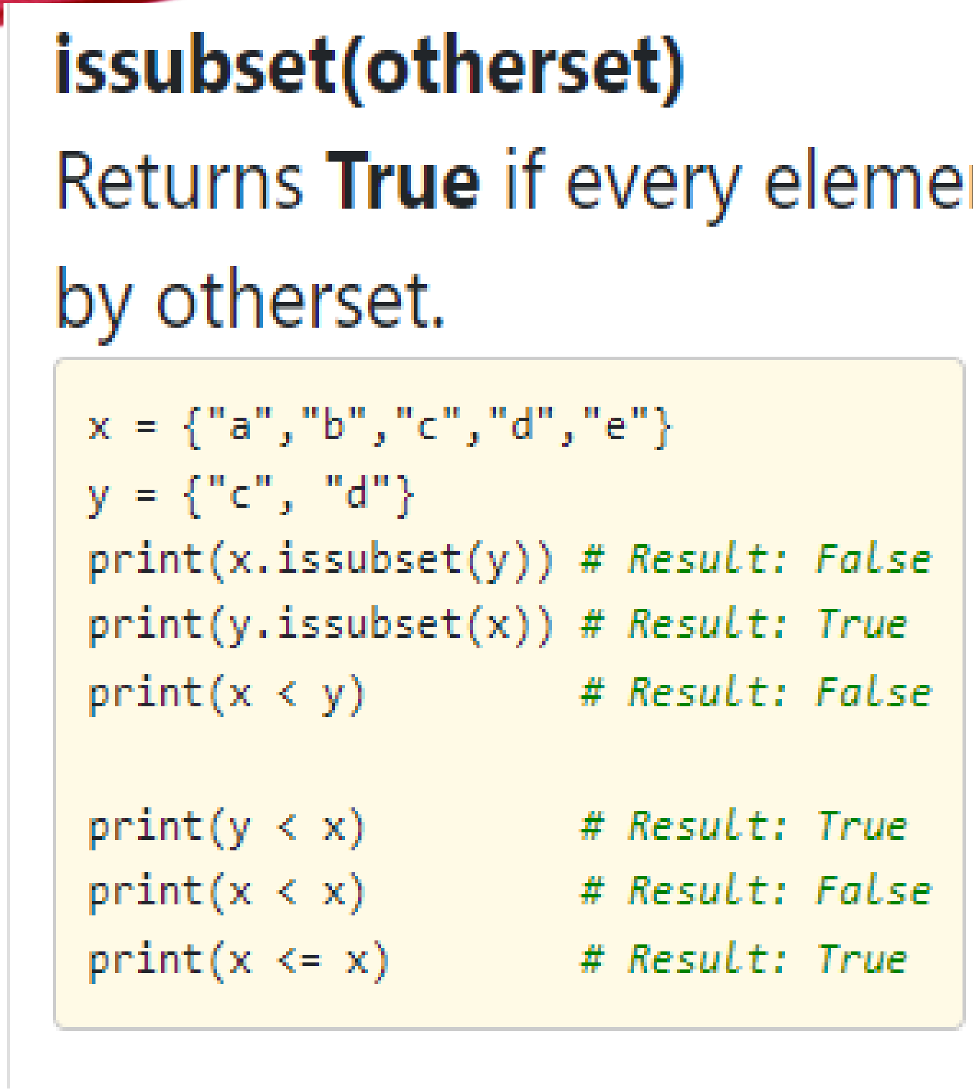
```
print(x.clear())
print(y)            # Result: {1, 2, 3, 4, 5, 6, 7}
```



isdisjoint(otherset)

Returns **True** if the **Set** calling this method has no elements in common with the other Set specified by otherset.

```
a = {1,2,3,4}
b= {5, 6}
a.isdisjoint(b)    # Result: True
```



issubset(otherset)

Returns **True** if every element in the set is in other set specified by otherset.

```
x = {"a","b","c","d","e"}
y = {"c", "d"}
print(x.issubset(y)) # Result: False
print(y.issubset(x)) # Result: True
print(x < y)         # Result: False

print(y < x)         # Result: True
print(x < x)         # Result: False
print(x <= x)        # Result: True
```

issuperset(otherset)

Returns **True** if every element in other set specified by otherset is in the set calling this method.

```
x = {"a","b","c","d","e"}
y = {"c", "d"}
print(x.issuperset(y)) # Result: True
print(x > y)           # Result: True
print(x >= x)          # Result: True
print(x > x)           # Result: False
print(x.issuperset(x)) # Result: True
```

LIST COMPREHENSIONS

Comprehensions are used to construct the sequences to be built from other sequences.

We can use List Comprehensions to create Lists based on existing lists.

List comprehension is a complete substitute for the lambda function as well as the **map()**, **filter()** and **reduce()** functions.

List Comprehension consists of brackets containing an expression followed by a for clause and then may be followed by an additional zero or more for clause or if statements

The result of List Comprehension may be of evolving an expression through iterating the for loop.

The result of a List Comprehension will always be a List.

Syntax for List Comprehensions:

```
[ expression for an item in iterate if condition ]
```

Let's consider the below example to print **1** to **10** numbers in List using **for clause**.

```
list1 = []
for i in range(1, 11):
    list1.append(i)
print(list1)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Here we consider an empty List and iterate the for loop for **10** times and every element in **range(1, 11)** will be appended to empty list. and then print list.

Let's try the above example using List Comprehension:

```
[i for i in range(1, 11)] # will print the result as
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

It's very simple and takes only one line to print the *list* that contains **1** to **10** numbers

```
[i for i in range(1, 11)]
```

Here `i` is an expression which gives final output, for i in range(1, 11) means repeats a for loop from **1** to **10** numbers.

Every time the for loop repeats, the `i` value will be stored in expression `i`.

We can also use nested for clauses in List Comprehensions.

There is no limit on the number of clauses in List Comprehensions, but you should remember that the order of the loops should be the same in both the original code and the list comprehension.

We can also add optional if conditions after every for clause if needed.

```
[ expression for item in iterable (optional if condition)
             for item in iterable (optional if condition)
             for item in iterable (optional if condition)......]
```

Example 1: Let's take multiplication of two lists using nested list comprehension.

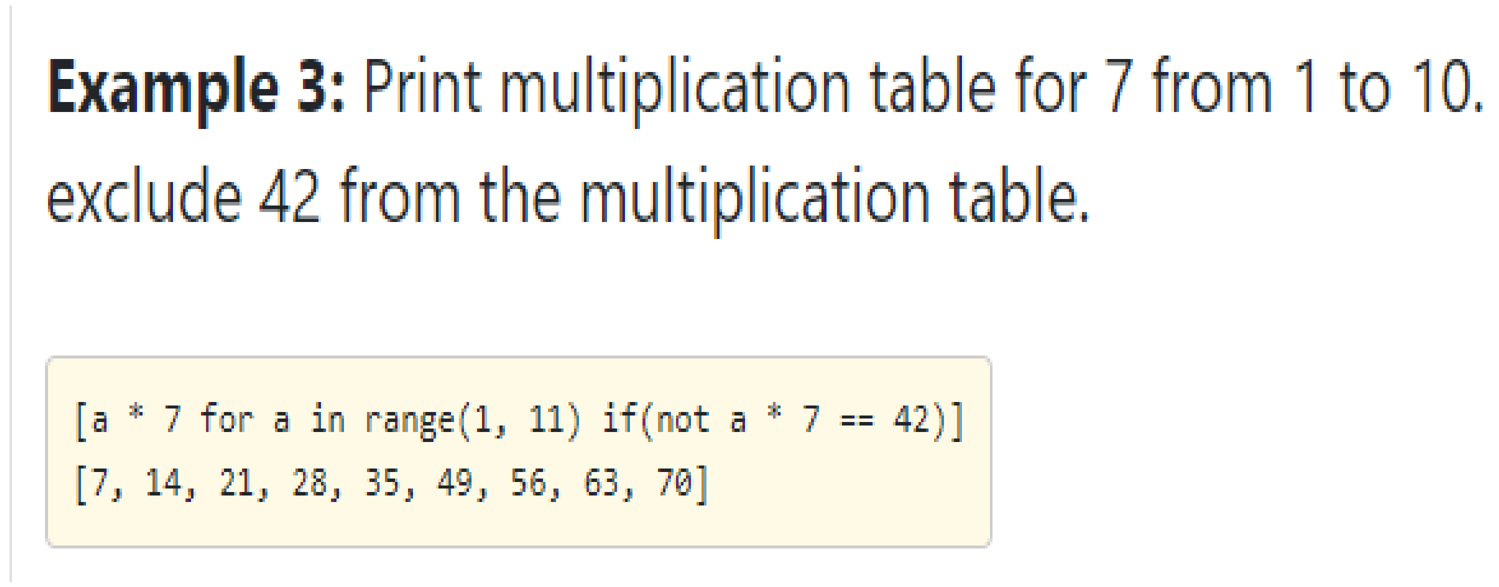
```
[a * b for a in [1, 2, 3] for b in [10, 20, 30]]
# Result: [10, 20, 30, 20, 40, 60, 30, 60, 90]
```

Here we take two for loops in list comprehension. while iteration of two loops, the given list elements get multiplied.

Example 2: Let's discuss how to find common elements of two lists.

```
[a for a in [10, 8, 5, 4] for b in [4, 7, 5, 10] if a == b]
[10, 5, 4]
```

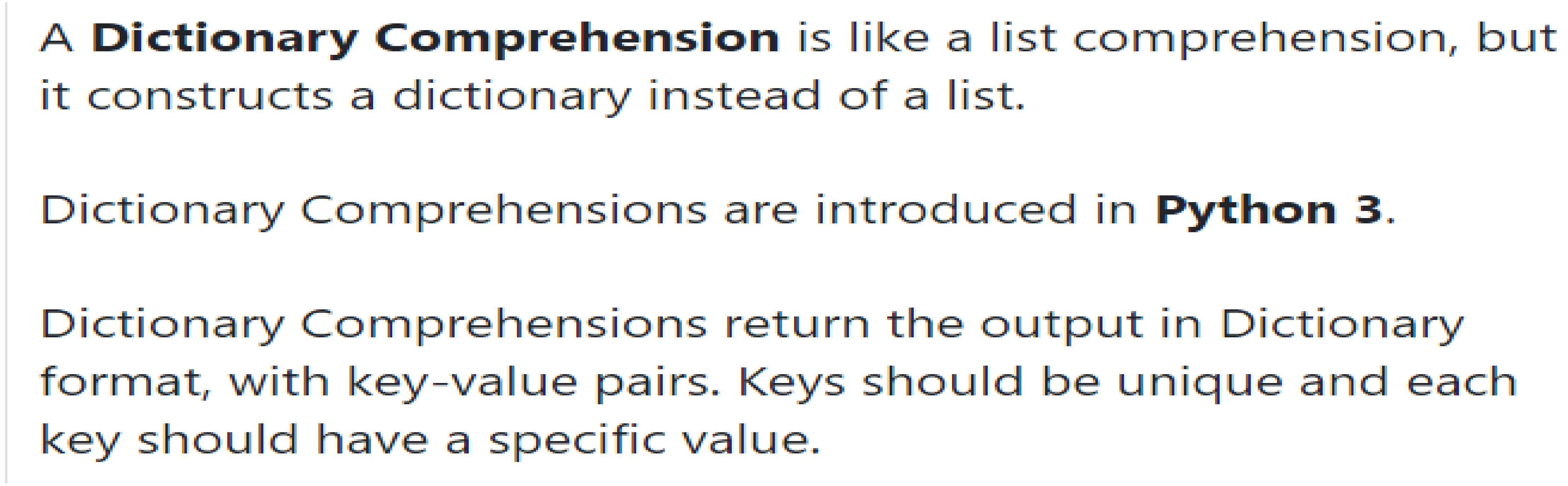
Here we check the condition for elements of one list with elements of another list to find common elements in the two lists.



Example 3: Print multiplication table for 7 from 1 to 10. exclude 42 from the multiplication table.

```
[a * 7 for a in range(1, 11) if(not a * 7 == 42)]
[7, 14, 21, 28, 35, 49, 56, 63, 70]
```

DICTIONARY COMPREHENSIONS



A **Dictionary Comprehension** is like a list comprehension, but it constructs a dictionary instead of a list.

Dictionary Comprehensions are introduced in **Python 3**.

Dictionary Comprehensions return the output in Dictionary format, with key-value pairs. Keys should be unique and each key should have a specific value.

Syntax for Dictionary Comprehension:

```
{key : value for (key , value) in iterable}
```

Just like in list comprehensions, you can use an 'if clause' in dictionary comprehension to filter items from the input sequence based on an expression.

Let us consider a simple example using Dictionary Comprehension:

```
list1 = [10, 20, 30]

dict1 = {key : value for key, value in enumerate(list1)}

print(dict1) # will print result as follows

{0: 10, 1: 20, 2: 30}
```

```
dict2 = {i : i ** 2 for i in range(1, 11)}
print(dict2) # will print result as follows

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

We consider the expression as `i : i ** 2`, where **i** represents the values of **i**, and **i ** 2** represents the square of the **i** value.