In [6]:
```python
# Factorial using recursion

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(7))
```

```
5040
```

In [ ]:
```python

```

Python Scope

    A variable is only available from inside the region it is created.
    This is called scope.

Local Scope

    A variable created inside a function belongs to the local scope of
    that function, and can only be used inside that function.

In [8]:
```python
# A variable created inside a function is available inside that functio

def myfunc():
  x = 300
  print(x)

myfunc()
```

```
300
```

Global Scope

    A variable created in the main body of the Python code is a global
    variable and belongs to the global scope.
    Global variables are available from within any scope, global and lo
    cal.

In [10]:
```python
#  variable created outside of a function is global and can be used by
x = 300

def myfunc():
  print(x)

myfunc()
print(x)
```

```
300
300
```

```
# Naming Variables
    If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):
```

In [4]:
```python
# The function will print the local x, and then the code will print the
x = 300

def myfunc():
    x = 200
    print(x)

myfunc()

```

```
200
```

Global Keyword

- If you need to create a global variable, but are stuck in the local scope, you can use the global keyword. The global keyword makes the variable global.

In [3]:
```python
# If you use the global keyword, the variable belongs to the global sco

def myfunc():
    global x
    x = 300

myfunc()

print(x)
```

```
300
```

In [2]:
```python
# To change the value of a global variable inside a function, refer to

x = 300

def myfunc():
    global x
    x = 200

myfunc()

print(x)
```

```
200
```

In [9]:
```python
# Keyword Arguments

def my_function(child1,child2,child3):
    print("The youngest child is " + child3)

my_function(child1 = "instagram", child2 = "facebook", child3 = "sharec
```

```
The youngest child is sharechat
```

# Variable-length arguments

```
 - Variable-length arguments refer to a feature that allows

   - a function to accept a variable number of arguments in Python. It
is also known as the argument that can also accept an unlimited amount
of data as input inside the function. There are two types in Python:

        Non – Keyworded Arguments (*args)
        Keyworded Arguments (**kwargs)
```

```
 - *args and **kwargs allow functions to accept a variable number of
arguments:


        *args (arguments) allows you to pass a variable number of
positional arguments to a function.

        **kwargs (keyword arguments) allows you to pass a variable
number of keyword arguments (key-value pairs) to a function.
```

```
- Difference between *args and **kwargs in Python?

    - *args collects additional positional arguments as a tuple, while
**kwargs collects additional keyword arguments as a dictionary.
```

In [ ]:
```
#  a function sum_all that accepts any number of arguments.

The *args syntax collects all the arguments into a tuple named args. In
we iterate through the args tuple and calculate the sum of all the numb
```

In [7]:
```
def sum_all(*args):
    result = 0
    for num in args:
        result += num
    return result

print(sum_all(1, 2, 3, 4, 5))
```

```
15
```

```
    What is Python **kwargs?
      -  In Python, **kwargs is used to pass a keyworded, variable-
length argument list. We call kwargs with a double star.
      - The reason for this is that the double star allows us to pass
over keyword arguments (in any order).

       - Arguments are collected into a dictionary within the function
that allow us to access them by their keys.
```

In [8]:
```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_info(name="Alice", age=30, city="New York")
```

```
name: Alice
age: 30
city: New York
```

In [6]:
```
def example_function(*args, **kwargs):
    print(args)    # tuple of positional arguments
    print(kwargs)  # dictionary of keyword arguments

example_function(1, 2, 3, name='Alice', age=30)

```

```
(1, 2, 3)
{'name': 'Alice', 'age': 30}
```

In [ ]:
```

```