05 - Functions

What is a function?

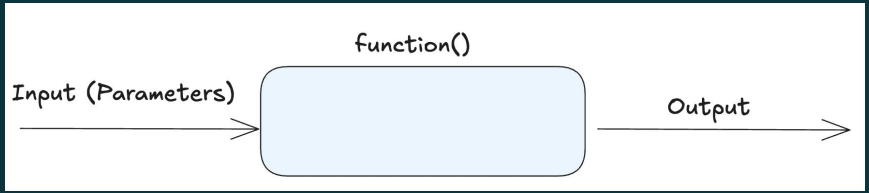
What is a function?

A function is a block of code or a set of instructions that get executed when the function is called.

```
# Creating a function
def some_function():
    x = 10
    print(x)



# Calling the function
some_function()
```

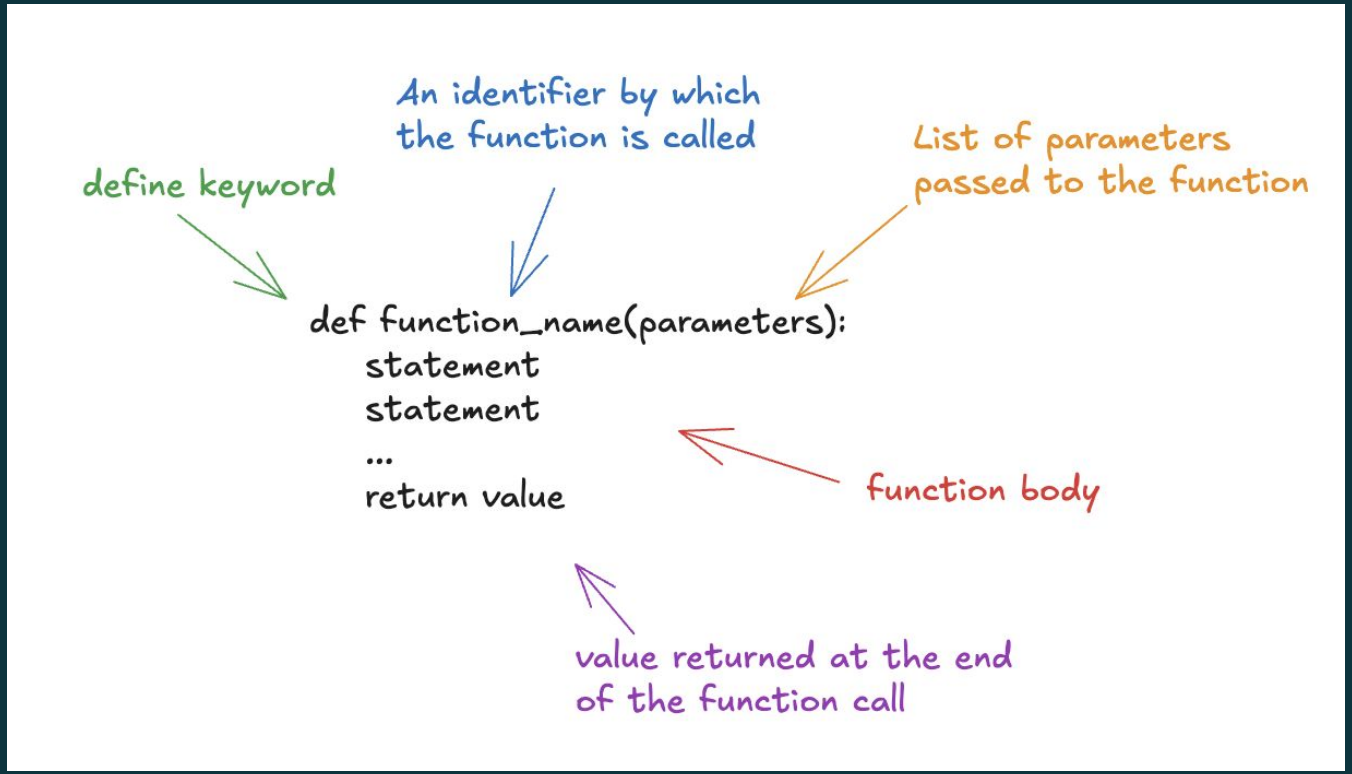
What is a function?

A function is a block of code or a set of instructions that get executed when the function is called.




```
# Creating a function
def some_function():
    x = 10
    print(x)



# Calling the function
some_function()
```

Function Syntax



What is a function?

You can also define inputs to the function known as **parameters**.

```
# Creating a function
def say_hello(name):
    x = 10
    print(f"Hello {name}!")


# Calling the function
say_hello("Ali")
```

What is a function?

You can also define inputs to the function known as **parameters**.

Note that **arguments** and **parameters** are sometimes used interchangeably.

Argument: passing a value to a function.

Parameter: defining an input to a function.

```
# Creating a function
def say_hello(name):
    x = 10
    print(f"Hello {name}!")


# Calling the function
say_hello("Ali")
```

Types of function arguments

1. Keyword
2. Default
3. Positional
4. Arbitrary Positional
5. Arbitrary Keyword

Keyword Arguments

Keyword arguments are passed to a function by explicitly specifying the parameter name.

```python
def greet(name, message):
    print(f"{message}, {name}!")


greet(name="Alice", message="Welcome")  # Output: Welcome, Alice!
greet(message="Hi", name="Boc")         # Output: Hi, Bob!
```

Default Arguments

Default arguments are parameters that assume a default value if a value is not provided in the function call.

```python
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()          # Output: Hello, Guest!
greet("Alice")   # Output: Hello, Alice!
```

Positional Arguments

Positional arguments are passed to a function in the order in which they are defined.

```python
def greet(name, message):
    print(f"{message}, {name}!")

greet("Dave", "Good morning")  # Output: Good morning, Dave!
greet("Good morning", "Dave")  # Output: Dave, Good morning!
```

Arbitrary Positional Arguments

Arbitrary positional arguments allow a function to accept any number of positional arguments, which are passed in as a tuple.

```python
def grocery(*names):
    print("Grocery List:", names)

grocery("Bread", "Eggs", "Milk")  # Grocery List: ('Bread', 'Eggs', 'Milk')
```

Arbitrary Keyword Arguments

Arbitrary keyword arguments allow a function to accept any number of keyword arguments, which are passed in as a dictionary.

```python
def display(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

display(name="Bob", age=30, city="New York", phone="1234567890")
# Output:
# name: Bob
# age: 30
# city: New York
# phone: 1234567890

display(name="Alice", age=25)
# Output:
# name: Alice
# age: 25
```

Bringing it all together

Here’s an example that combines positional, default, arbitrary positional, and arbitrary keyword arguments.

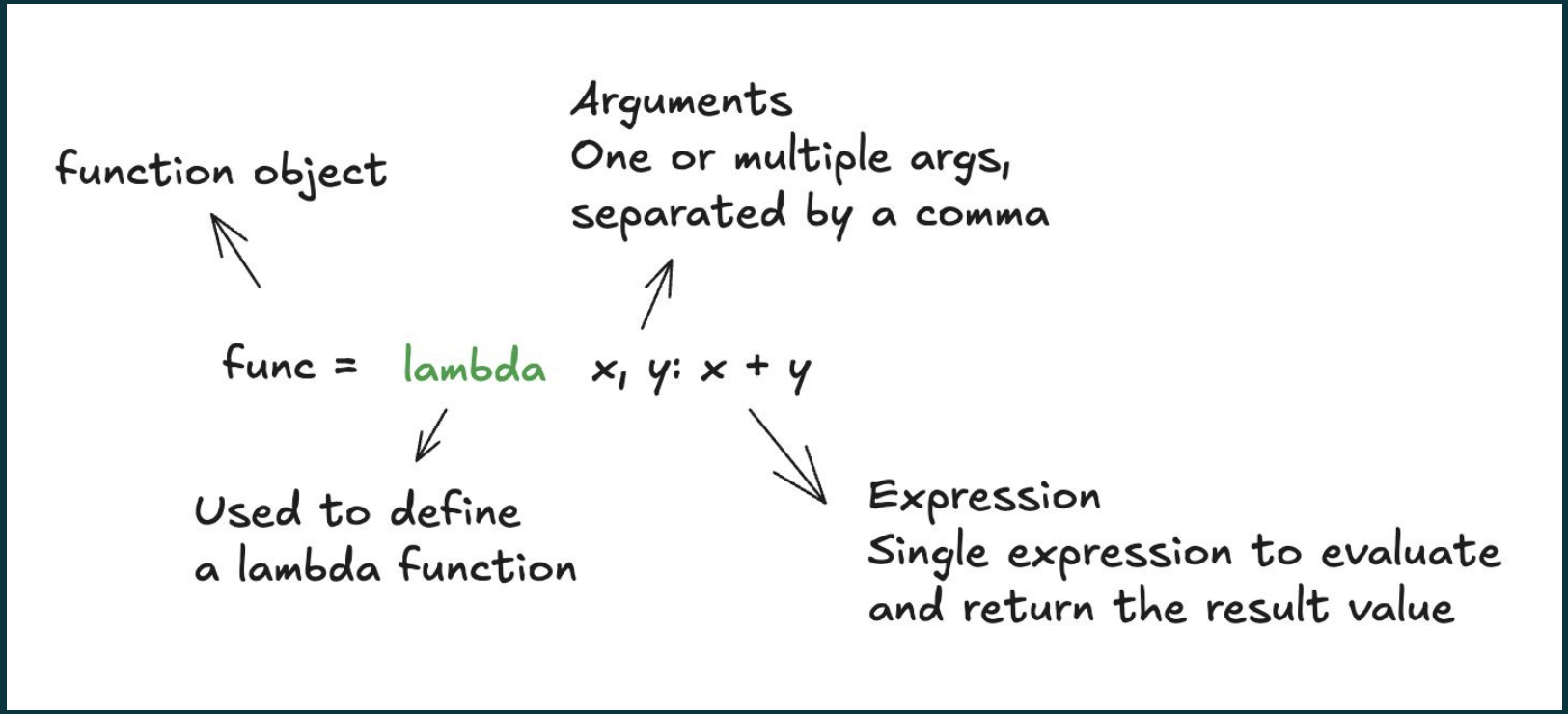
```python
def customer_info(first_name, last_name, country="USA", *phone_numbers, **info):
    print(f"First Name: {first_name}")
    print(f"Last Name: {last_name}")
    print(f"Country: {country}")
    print("Phone Numbers:", phone_numbers)
    print("Additional Information:", info)

# Example call to the function
customer_info(
    "John",
    "Doe",
    "Canada",
    "123-456-7890",
    "987-654-3210",
    email="john.doe@example.com",
    age=30,
)
# Output
# First Name: John
# Last Name: Doe
# Country: Canada
# Phone Numbers: ('123-456-7890', '987-654-3210')
# Additional Information: {'email': 'john.doe@example.com', 'age': 30}
```


Lambda Function

Lambda Function

A lambda function is an anonymous function (defined with `**lambda**` keyword) that can take any number of arguments, but can return and evaluate one expression (unlike normal functions).

Lambda Function



Lambda Function

Syntax:

lambda arguments : expression

```
add = lambda a, b: a + b
print(add(5, 10))             # Output: 15
```

Lambda Function

Syntax:

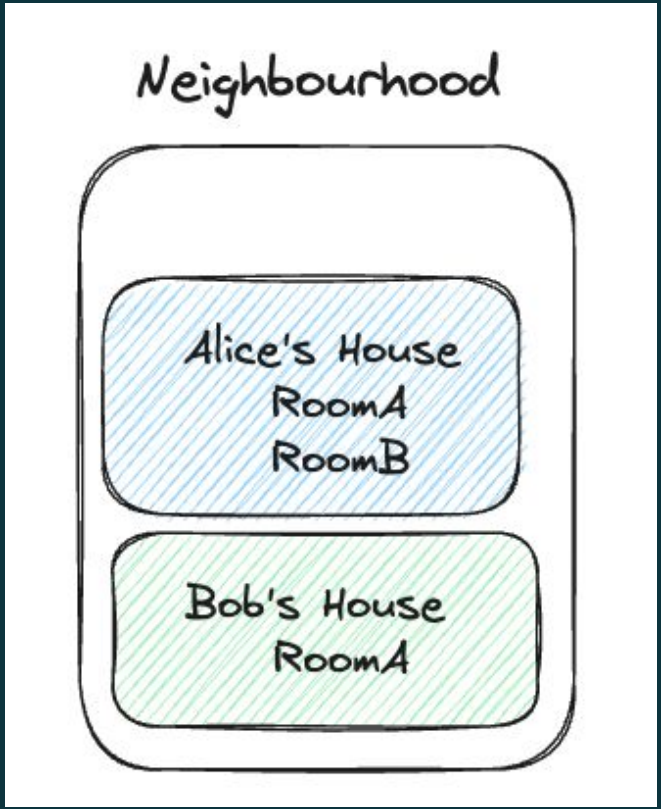
lambda arguments : expression

Lambda functions are sometimes useful for readability and flexibility. They're generally used for a short-lived, line, or local-scope operations.

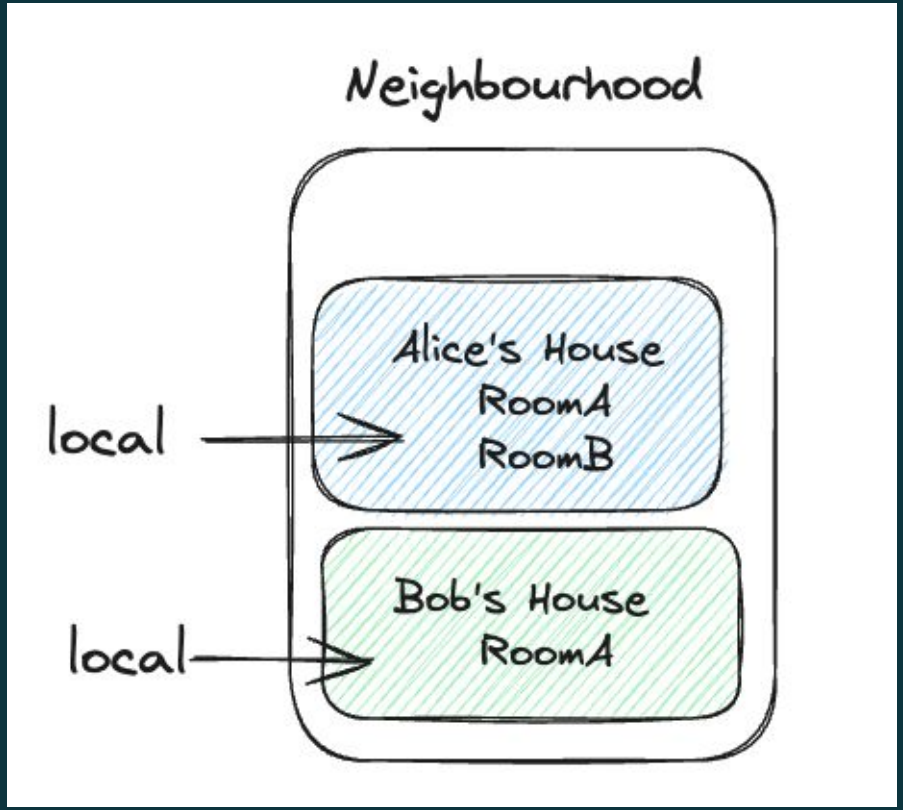
```
add = lambda a, b: a + b
print(add(5, 10))              # Output: 15
```

Scope

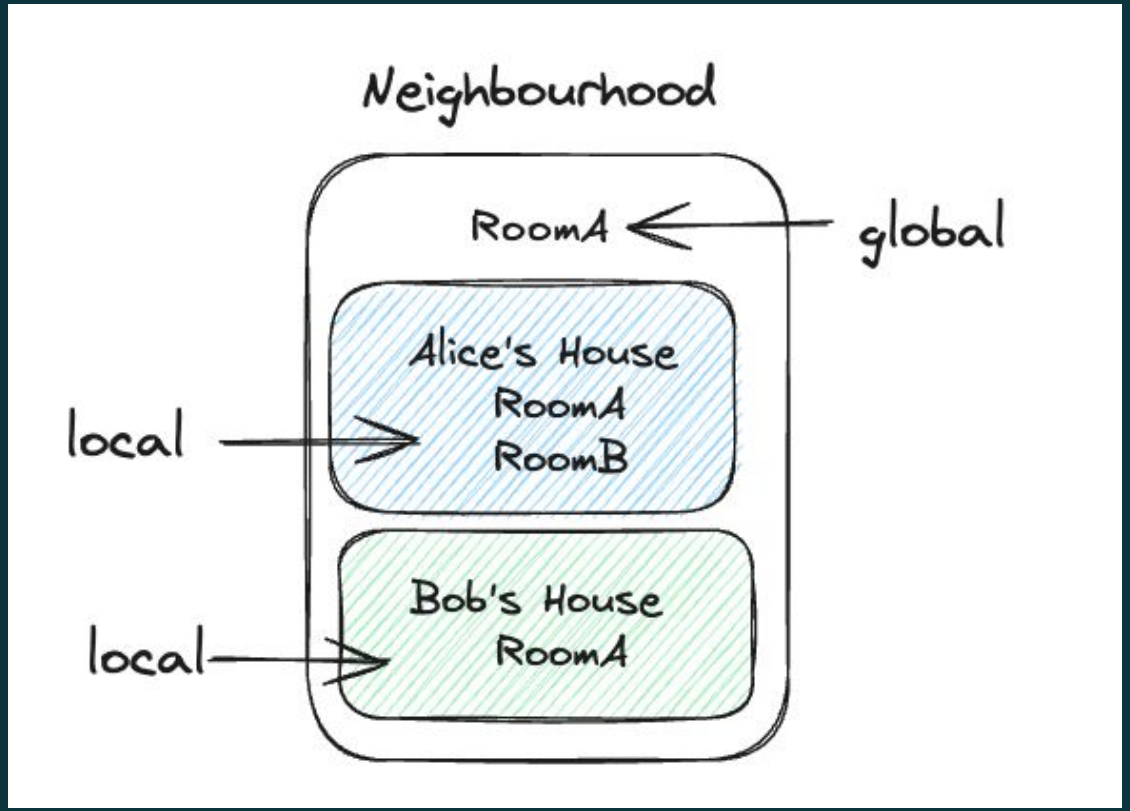
Scope



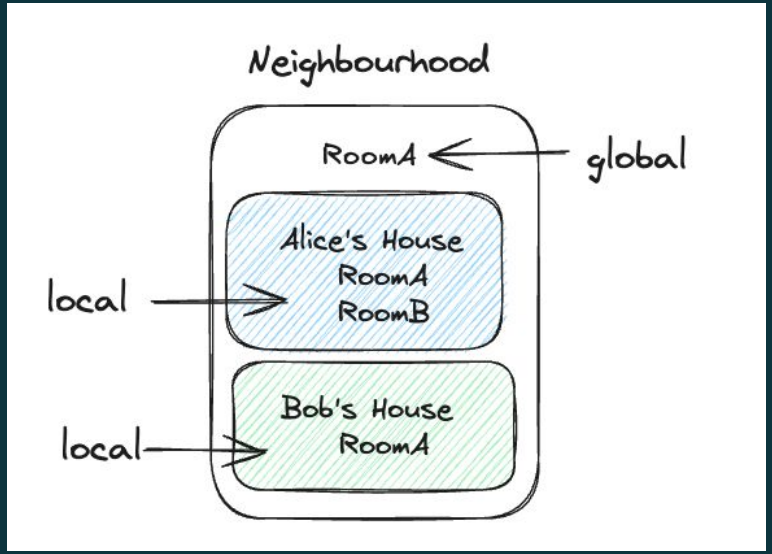
Scope

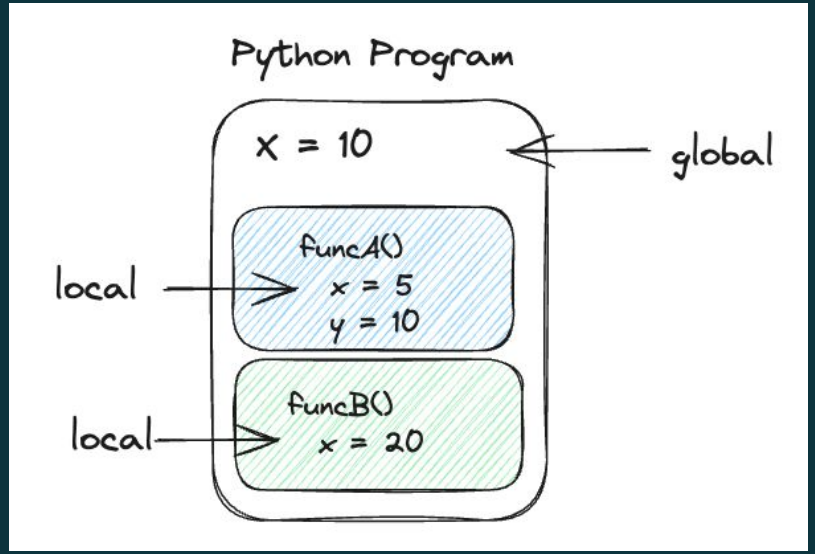


Scope



Scope





Scope

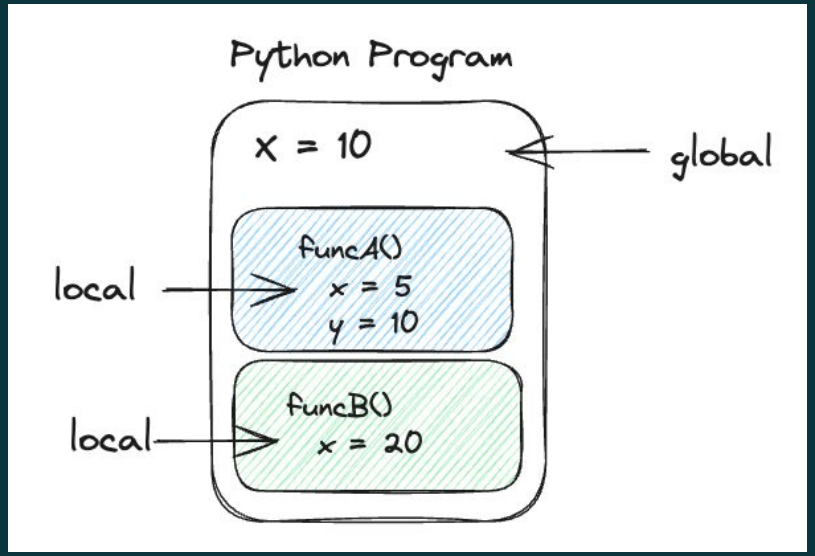
Where a variable is only available from inside the region it is created.

There are two types of scope:

- **Local** - A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.
- **Global** - A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Scope

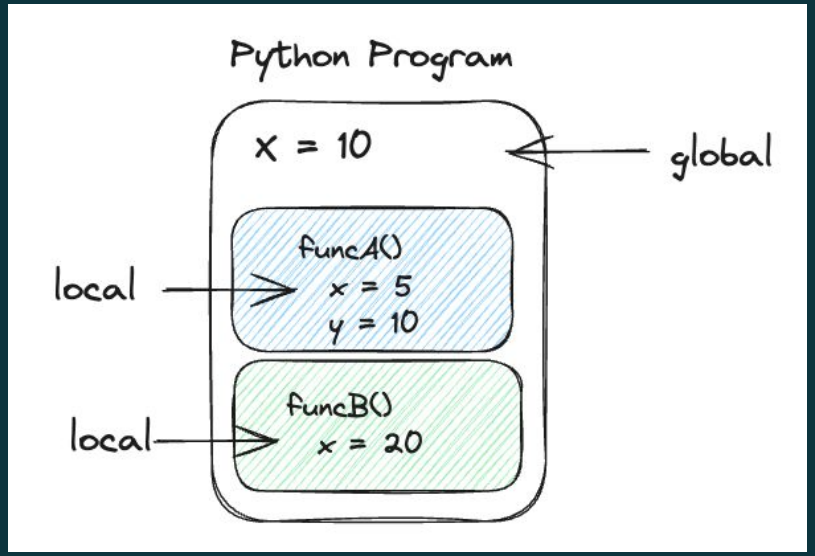
Is there a way to use the same variable x?



Scope

Is there a way to use the same variable x?

Yes, by changing its scope.



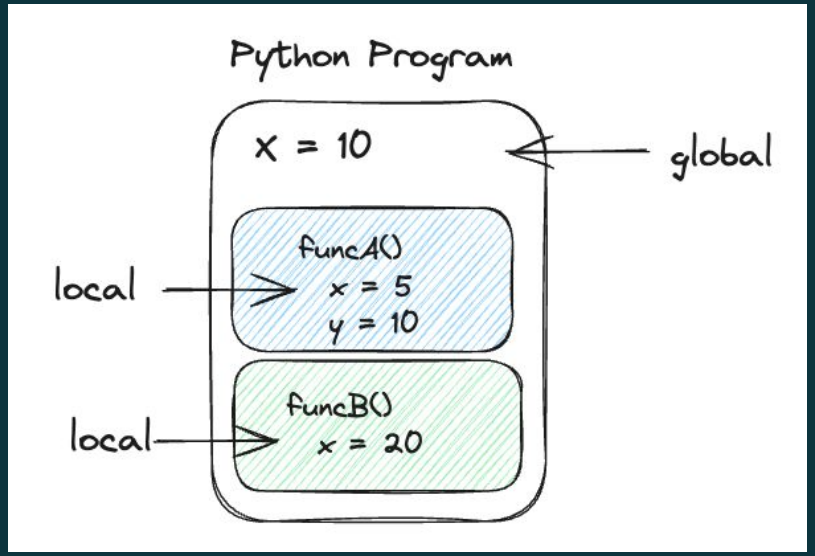
Scope

Is there a way to use the same variable x?

Yes, by changing its scope.

There are two ways to change scope:

- `nonlocal` keyword
- `global` keyword



Using `nonlocal`

Using `nonlocal`

The **nonlocal** keyword is used to indicate that a variable inside a nested function refers to a variable in the nearest enclosing scope that is not global.

This allows you to modify the value of the variable in the outer function from within the inner function.

```
def outer_function():

    x = 10
    def inner_function():
        nonlocal x
        x += 5
        print("Inner function:", x)   # 15

    inner_function()
    print("Outer function:", x)       # 15

outer_function()
```

Using `global`

Using `global`

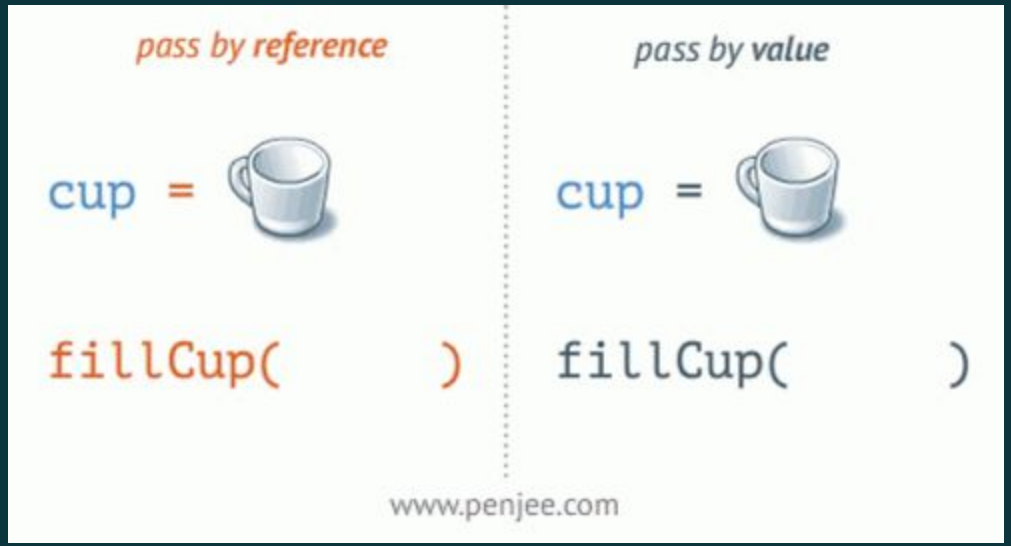
The **global** keyword is used to declare that a variable inside a function should refer to the globally defined variable of the same name, instead of creating a new local variable.

This allows you to modify the global variable from within a function.

```
x = 10  # Global variable

def modify_global():
    global x  # Declare x as global within this function
    x = 20    # Modify the global variable x


print("Before function call, x =", x)     # 10
modify_global()
print("After function call, x =", x)      # 20
```

Pass by object reference

Pass by object reference



Pass by object reference

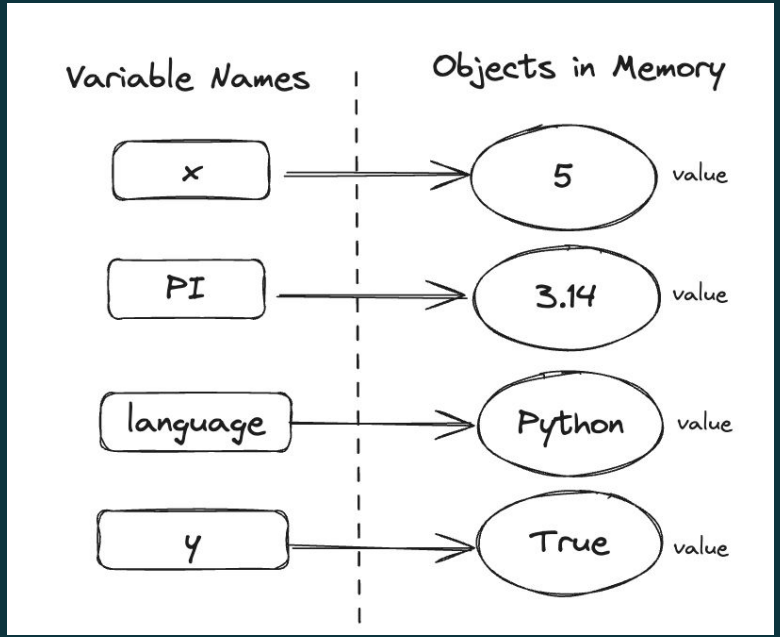
In other programming languages like C/C++, there are different terms used to describe how arguments are passed to functions:

- **Pass by reference**
- **Pass by value**

However, it's important to clarify that Python uses a different mechanism that can be more accurately described as **Pass By Object Reference**.

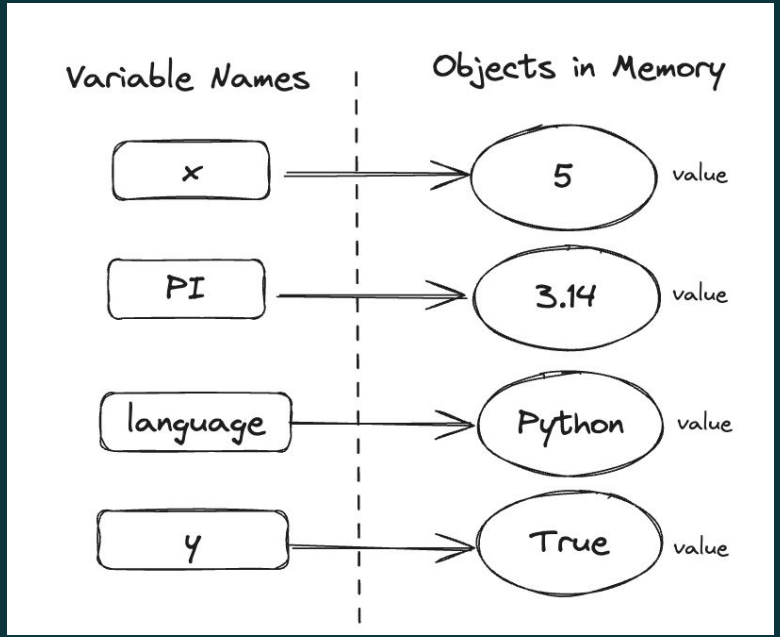
What is pass by object reference?

- In Python, everything is an object.



What is pass by object reference?

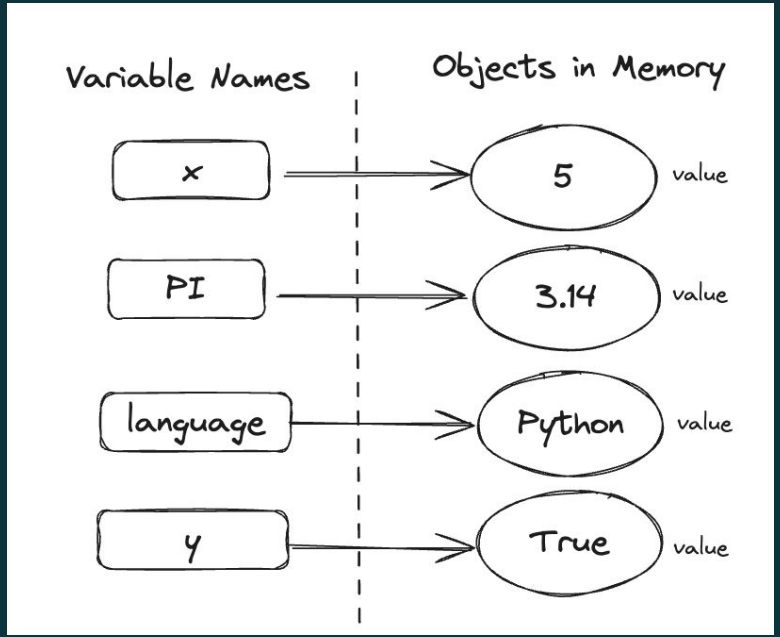
- In Python, everything is an object.
- Variables hold references to objects rather than the objects themselves.



What is pass by object reference?

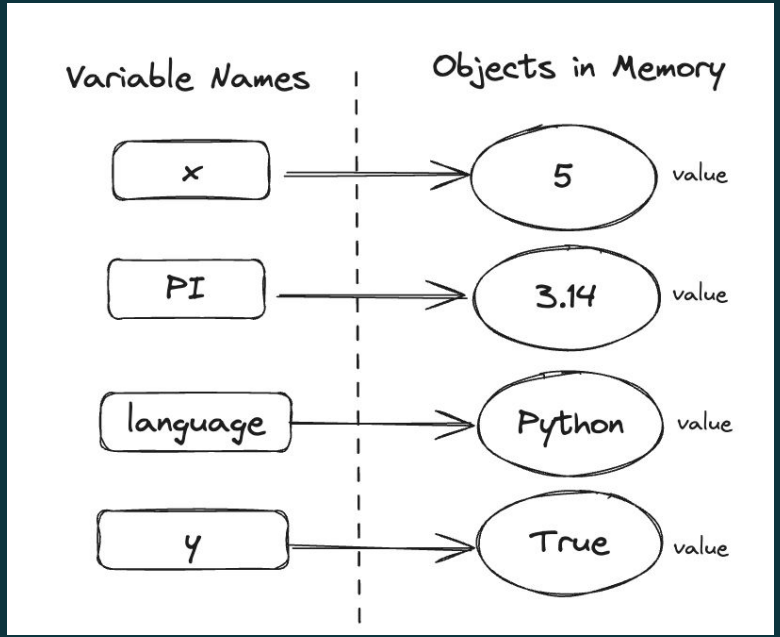
- In Python, everything is an object.
- Variables hold references to objects rather than the objects themselves.

- When you pass an argument to a function in Python, you are passing the reference to the object, not the actual object itself.



What is pass by object reference?

- In Python, everything is an object.
- Variables hold references to objects rather than the objects themselves.



- When you pass an argument to a function in Python, you are passing the reference to the object, not the actual object itself.
- Python objects can be either **mutable** (e.g., lists) or **immutable** (e.g., int, strings).
- The behaviour of "pass by object reference" can depend on whether the object is mutable or immutable.

Passing immutable objects

```
def modify_immutable(x):
    x = x + 1  # Creates a new int object


a = 10
modify_immutable(a)
print(a)  # Output: 10 (a remains unchanged)
```

Passing immutable objects

When you pass immutable objects (like integers, strings) to a function, you can't modify the object itself within the function because you're working with a copy of the reference.

If you re-assign the variable inside the function, it creates a new local reference, not affecting the original object outside the function.

```
def modify_immutable(x):
    x = x + 1  # Creates a new int object


a = 10
modify_immutable(a)
print(a)  # Output: 10 (a remains unchanged)
```

Passing mutable objects

```
def modify_mutable(lst):
    lst.append(2)  # This modifies the list in-place


my_list = [1]
modify_mutable(my_list)
print(my_list)  # Output: [1, 2] (my_list is modified)
```

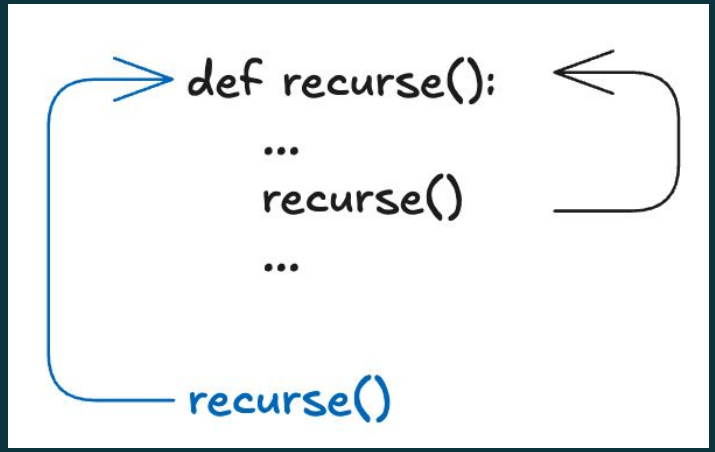
Passing mutable objects

When you pass mutable objects (like lists, dictionaries) to a function, changes made to the object within the function will persist outside the function because you're still working with the same object through its reference.

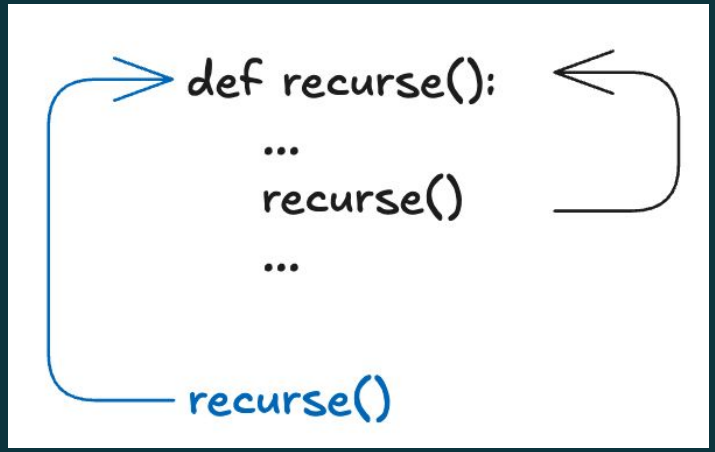
```
def modify_mutable(lst):
    lst.append(2)  # This modifies the list in-place


my_list = [1]
modify_mutable(my_list)
print(my_list)  # Output: [1, 2] (my_list is modified)
```

Recursion

Recursion



Recursion




```
def hello_world():
    print("Hello world!")
    hello_world()   # Calls itself!


hello_world() # Call the first time
```

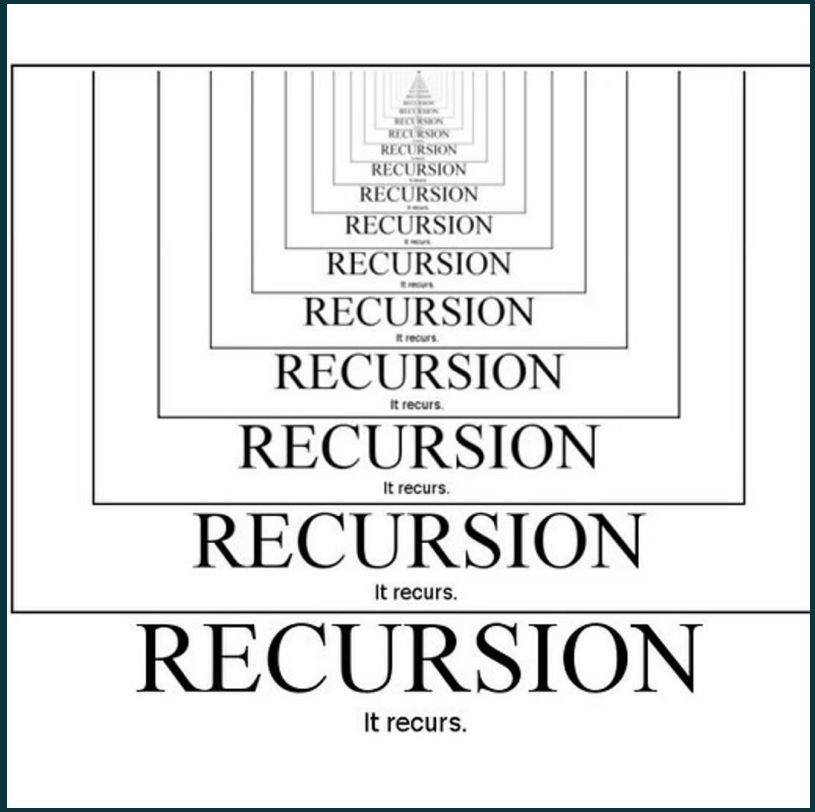
Recursion

Recursion is the process where a function calls itself as a sub-function, with different inputs.

```
def countdown(n):
    if n <= 0:
        print("Blastoff!")
    else:
        print(n)
        countdown(n - 1)

# Example usage:
countdown(5)
```

Recursion

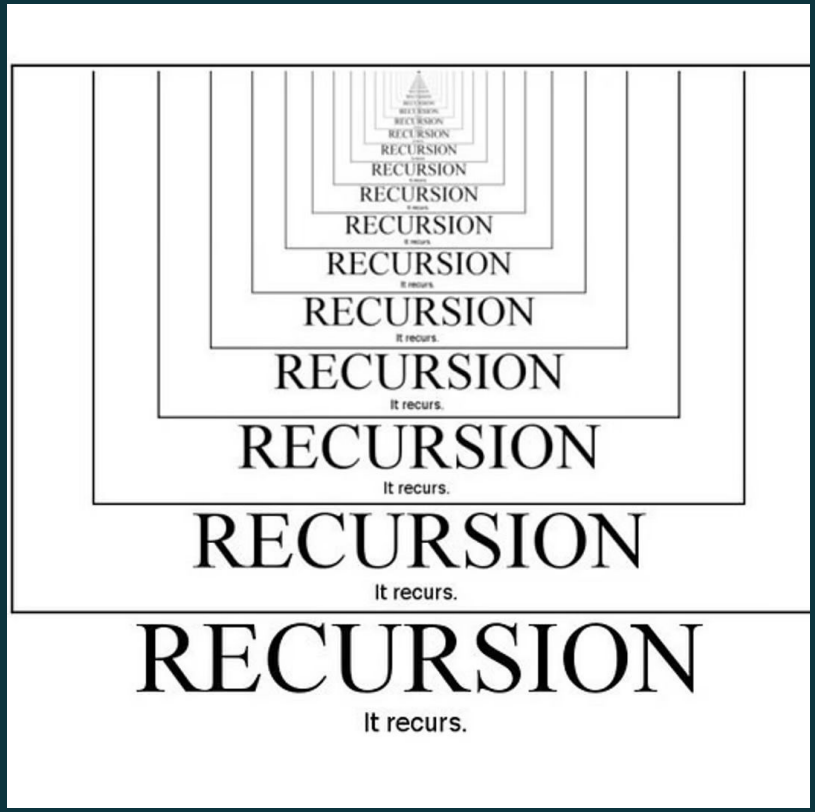
What if we infinitely call our function?



Recursion

What if we infinitely call our function?

Answer: We get a **Stack overflow error**



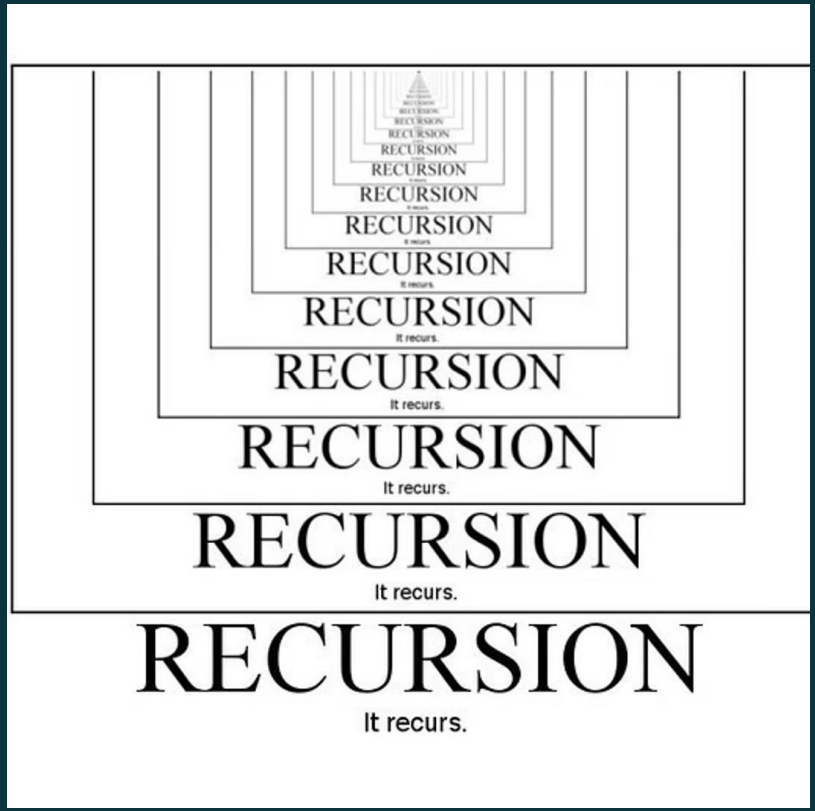
Recursion

What if we infinitely call our function?

Answer: We get a **Stack overflow error**

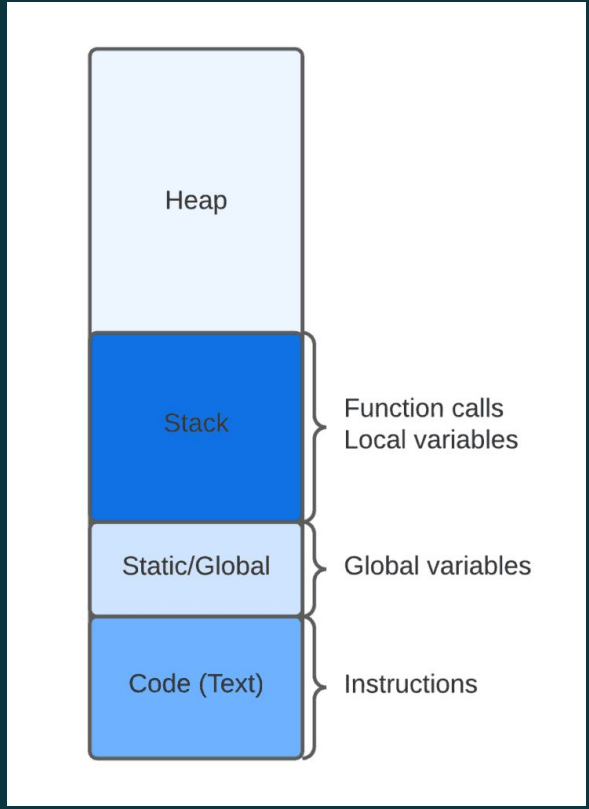
What is a Stack overflow?

A stack overflow is a type of error that occurs when a computer program tries to use more memory space than what was allocated to the stack.



Application Memory

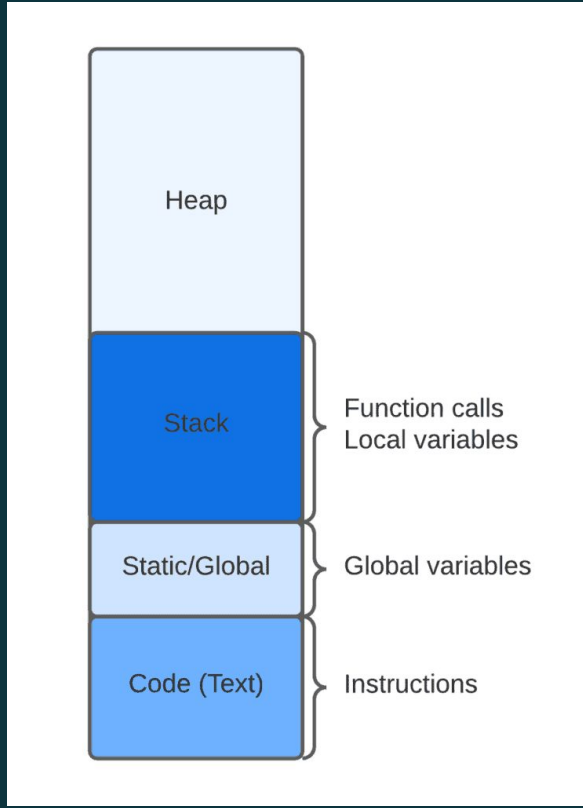
There are main 4 sections in the memory when running a Python program.



Application Memory

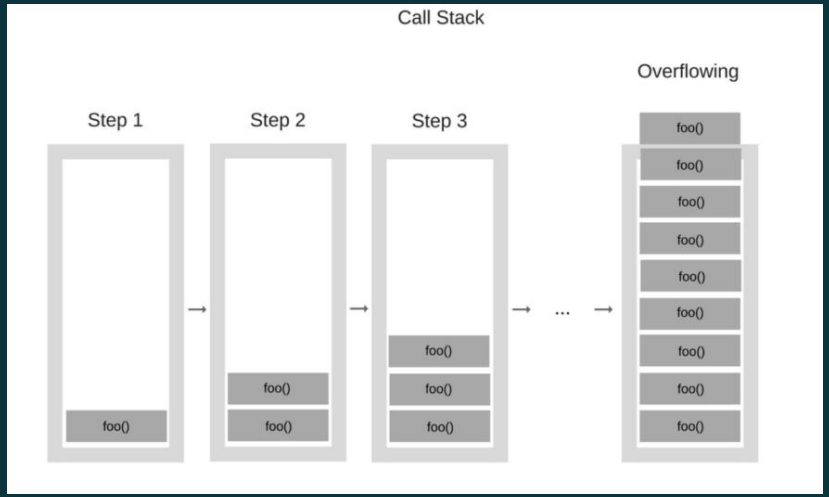
There are main 4 sections in the memory when running a Python program.

- **Heap:** used for dynamic memory allocation, typically for objects and data structures like lists, dictionaries, and classes.
- **Stack:** used for storing function calls, local variables, and control flow.
- **Static/Global:** also called the **data segment**, holds global variables and static data initialised before runtime.
- **Code:** also called the **text segment**, is the part of memory where the compiled machine code of the program (i.e. the instructions) resides.



Recursion

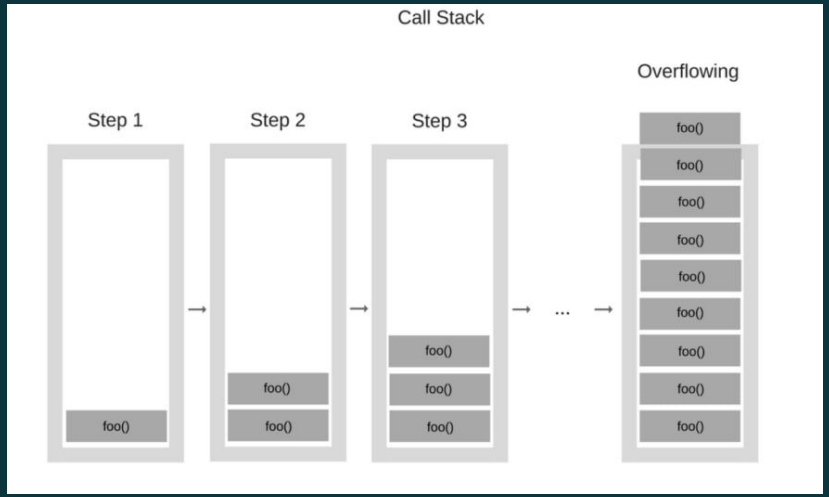
A stack overflow is a type of error that occurs when a computer program tries to use more memory space than what was allocated to the stack.



Recursion

A stack overflow is a type of error that occurs when a computer program tries to use more memory space than what was allocated to the stack.

This is a common problem when writing recursive functions for the first time. We need to hit a **base condition** that stops it from calling itself and go back.



Recursion

Recursion is the process where a function calls itself as a sub-function, with different inputs.

```
def countdown(n):
    if n <= 0:    # BASE CASE
        print("Blastoff!")
    else:
        print(n)
        countdown(n - 1) # RECURSIVE CASE

# Example usage:
countdown(5)
```

Recursion

Recursion is the process where a function calls itself as a sub-function, with different inputs.

Recursive functions should have:

- **One base case (at least)**
- **One recursive case (at least)**

```
def countdown(n):
    if n <= 0:    # BASE CASE
        print("Blastoff!")
    else:
        print(n)
        countdown(n - 1) # RECURSIVE CASE

# Example usage:
countdown(5)
```

Iterative vs Recursive

Iterative vs Recursive

Iterative: Uses loops (e.g., for, while) to repeat a set of instructions.

Recursive: A function calls itself to solve a smaller instance of the same problem.

```
def countdown_iterative(n):
    while n > 0:
        print(n)
        n -= 1
    print("Blastoff!")


def countdown_recursive(n):
    if n <= 0:    # BASE CASE
        print("Blastoff!")
    else:
        print(n)
        countdown(n - 1) # RECURSIVE CASE
```

Which approach is better?

```
def countdown_iterative(n):
    while n > 0:
        print(n)
        n -= 1
    print("Blastoff!")


def countdown_recursive(n):
    if n <= 0:    # BASE CASE
        print("Blastoff!")
    else:
        print(n)
        countdown(n - 1) # RECURSIVE CASE
```

Which approach is better?

Iterative

- Easier to implement (especially for beginners)
- Suitable for tasks where the number of repetitions is known or can be easily determined.
- Tends to be more memory-efficient as it avoids excessive function calls.

Recursive

- Useful for problems that can be broken down into smaller, similar subproblems (e.g., Fibonacci sequence, tree traversal).
- Can lead to more elegant and shorter code but may be less memory-efficient due to multiple function calls.

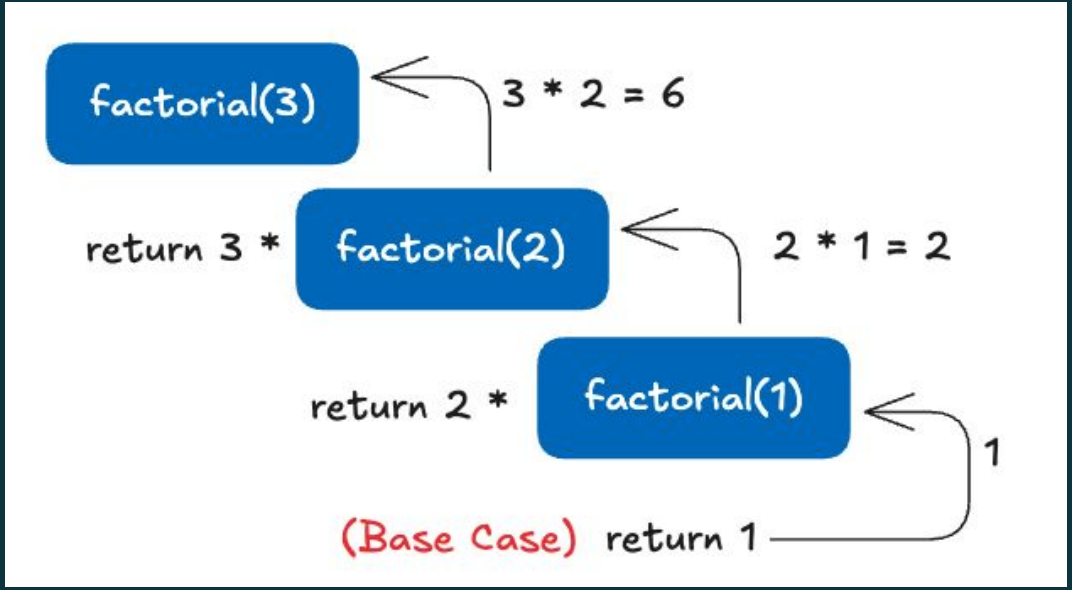
```
def countdown_iterative(n):
    while n > 0:
        print(n)
        n -= 1
    print("Blastoff!")


def countdown_recursive(n):
    if n <= 0:    # BASE CASE
        print("Blastoff!")
    else:
        print(n)
        countdown(n - 1) # RECURSIVE CASE
```

Example: Factorial using Recursion

$$n! = n \times (n-1) \times \cdots \times 1$$

Example: Factorial using Recursion

$$n! = n \times (n-1) \times \cdots \times 1$$



Type Hinting

Type Hinting

Type hinting allows you to specify the expected data types of variables and function arguments/returns.

Type hints are optional, introduced in **Python 3.5**.

Improves code readability and helps detect type-related errors.

```
def add(x: int, y: int) -> int:
    return x + y


def greet(name: str) -> str:
    return f"Hello, {name}!"

age: int = 25 # Type hint for a variable
```

Conclusion

- What is a function?
  - Five types of arguments (keyword, default, positional, *args, **kwargs)
- Lambda functions
- Scope
  - **nonlocal** vs **global** keywords
- Pass by object reference
- Recursion
- Type Hinting