In Python, you can pass values to functions and methods in different ways. Here are the main methods:

**1. Positional Arguments**

Values are passed in the same order as the function parameters.

def greet(name, age):

print(f"Hello, my name is {name} and I am {age} years old.")

greet("Alice", 25)

**2. Keyword Arguments**

Arguments are passed using parameter names, so order doesn't matter.

greet(age=25, name="Alice")

**3. Default Arguments**

If a parameter has a default value, you can skip passing it.

def greet(name="Guest"):

print(f"Hello, {name}!")

greet() # Uses default value "Guest"

greet("Alice") # Overrides default value

**4. Variable-Length Arguments (\*args)**

Allows passing multiple positional arguments as a tuple.

def sum\_all(\*args):

return sum(args)

print(sum\_all(1, 2, 3, 4, 5)) # Output: 15

**5. Variable-Length Keyword Arguments (\*\*kwargs)**

Allows passing multiple keyword arguments as a dictionary.

def display\_info(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

display\_info(name="Alice", age=25, city="New York")

**6. Passing by Reference (Mutable Objects)**

Lists, dictionaries, and other mutable objects are passed by reference, meaning changes inside the function affect the original object.

def modify\_list(lst):

lst.append(4)

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(my\_list) # Output: [1, 2, 3, 4]

**7. Passing by Value (Immutable Objects)**

Immutable objects (like integers, strings, tuples) are passed by value, meaning modifications inside the function do not affect the original variable.

def modify\_string(s):

s += " World"

msg = "Hello"

modify\_string(msg)

print(msg) # Output: "Hello" (unchanged)

**8. Passing Functions as Arguments**

Functions are first-class objects in Python, so they can be passed as arguments.

def apply\_function(func, value):

return func(value)

def square(x):

return x \* x

print(apply\_function(square, 5)) # Output: 25

**9. Passing Generators**

Instead of passing a list, you can pass a generator to save memory.

def generator():

for i in range(5):

yield i

def consume(iterable):

for value in iterable:

print(value)

consume(generator()) # Prints numbers 0 to 4

**10. Unpacking Arguments**

Using \* or \*\* to unpack values from a list or dictionary.

def add(a, b, c):

return a + b + c

nums = [1, 2, 3]

print(add(\*nums)) # Output: 6

info = {'a': 1, 'b': 2, 'c': 3}

print(add(\*\*info)) # Output: 6

These techniques allow for flexible function calls and improve code readability.