Lab Work - Define Your Own Function

Objective

This lab helps develop a clear understanding of how Python functions work and how they are used in practical engineering and AI-related tasks. The focus is on creating user-defined functions, using different argument types, writing docstrings, applying annotations, and testing how Python behaves when arguments are missing or mixed. Through electrical and real-world examples such as series resistance, Ohm's Law, solar inverter systems, and basic computations, the lab strengthens function design, data handling, and code documentation skills in a simple and applied way.

Task 1 — Student Record System

Make a function taking name and registration number.

Print both values.

Call using positional, keyword, and default arguments.

Try missing arguments to see the error.

Test if positional after keyword is allowed.

In [ ]:
```
def student_record(name, reg_no="N/A"):
    print("Student Name:", name)
    print("Registration No:", reg_no)
print("1. Positional Arguments")
student_record("Azka", "2025-CS-01")
print("2. Keyword Arguments")
student_record(name="Anza", reg_no="2025-EE-07")
print("3. Default Argument")
student_record("Noor")
print("4. Fewer Arguments (will cause error)")
print("5. Positional after Keyword (invalid)")
code = 'student_record(name="Sonia", "2025-ME-12")'
```

```
1. Positional Arguments
Student Name: Azka
Registration No: 2025-CS-01
2. Keyword Arguments
Student Name: Anza
Registration No: 2025-EE-07
3. Default Argument
Student Name: Noor
Registration No: N/A
4. Fewer Arguments (will cause error)
5. Positional after Keyword (invalid)
```

Task 2 — Ohm’s Law Calculator

Make a function taking current and resistance.

Multiply to get voltage.

Call using positional, keyword, and default arguments.

Mix arguments and observe behavior.

Check the error when skipping a required argument.

In [1]:
```
def ohms_law(I: float, R: float = 1.0) -> float:
    return I * R
print("Positional Arguments:")
print("Voltage:", ohms_law(2, 5), "V")

print("\nKeyword Arguments:")
print("Voltage:", ohms_law(I=3, R=4), "V")

print("\nDefault Arguments:")
print("Voltage:", ohms_law(7), "V")

print("\nMixing Positional and Keyword Arguments:")
print("Voltage:", ohms_law(5, R=10), "V")
```

```
Positional Arguments:
Voltage: 10 V

Keyword Arguments:
Voltage: 12 V

Default Arguments:
Voltage: 7.0 V

Mixing Positional and Keyword Arguments:
Voltage: 50 V
```

Task 3 — Solar Battery Charging

Function takes SOC value.

Print “Charging” if SOC < 20.

Print “Battery full, no action” if SOC > 90.

Print charging message with SOC otherwise.

Add docstring + annotations.

Print docstring.

In [2]:
```python
def charge_from_solar(SOC: float) -> None:
    """
    Controls the charging process of a battery based on State of Charge (SOC).

    Parameters:
        SOC (float): The state of charge of the battery in percentage (0–100).

    Returns:
        None: This function only prints the charging status.
    """

    if SOC < 20:
        print("Charging")
    elif SOC > 90:
        print("Battery full, no action")
    else:
        print(f"Battery charging, SOC = {SOC}%")

charge_from_solar(20)
charge_from_solar(95)
charge_from_solar(55)
print("\nDocstring of the function:")
print(charge_from_solar.__doc__)
```

```
Battery charging, SOC = 20%
Battery full, no action
Battery charging, SOC = 55%

Docstring of the function:

    Controls the charging process of a battery based on State of Charge (SOC).

    Parameters:
        SOC (float): The state of charge of the battery in percentage (0–100).

    Returns:
        None: This function only prints the charging status.
```

Task 4 — Resistance, Voltage & Current Lists

Function 1 returns list of 5 resistances.

Function 2 returns voltages for each resistor.

Function 3 calculates currents using both lists.

Display all results in main program.

Do one version as tuple, one as dictionary.

In [3]:
```python
def resistance() -> list[float]:
    """
```

```
    Returns a list of 5 resistor values (Ω).
    """
    return [10, 20, 30, 40, 50]

def voltage() -> list[float]:
    """
    Returns a list of 5 voltage values (V).
    """
    return [5, 10, 15, 20, 25]
def current_tuple(R: list[float], V: list[float]) -> tuple:
    """
    Calculate currents using Ohm's Law (I = V / R).
    Returns a tuple of current values.
    """
    return tuple(V[i] / R[i] for i in range(len(R)))

def current_dict(R: list[float], V: list[float]) -> dict:
    """
    Calculate currents using Ohm's Law (I = V / R).
    Returns a dictionary with keys current_1, current_2, ...
    """
    return {f"current_{i+1}": V[i] / R[i] for i in range(len(R))}

R = resistance()
V = voltage()

I_tuple = current_tuple(R, V)

I_dict = current_dict(R, V)

print("Resistance List:", R)
print("Voltage List:", V)
print("\nCurrent (Tuple Version):", I_tuple)
print("\nCurrent (Dictionary Version):")
for key, value in I_dict.items():
    print(f"{key}: {value:.2f} A")
```

```
Resistance List: [10, 20, 30, 40, 50]
Voltage List: [5, 10, 15, 20, 25]

Current (Tuple Version): (0.5, 0.5, 0.5, 0.5, 0.5)

Current (Dictionary Version):
current_1: 0.50 A
current_2: 0.50 A
current_3: 0.50 A
current_4: 0.50 A
current_5: 0.50 A
```

Task 5 — Series Resistance (Again)

Take a list/tuple of resistances.

Add all resistances.

Return total.

Add docstring + annotations.

Print annotations.

In [4]:
```python
def series(resistances: list[float] | tuple[float, ...]) -> float:
    """
    Calculate the total resistance of resistors connected in series.

    Parameters:
        resistances (list[float] or tuple[float]): A list or tuple containing

    Returns:
        float: The total resistance of the series combination.
    """
    return sum(resistances)

R_list = [10, 20, 30, 40, 50]
total_R = series(R_list)

print("Resistance List:", R_list)
print("Total Series Resistance:", total_R, "Ω")

print("\nFunction Annotations:")
print(series.__annotations__)
```

```
Resistance List: [10, 20, 30, 40, 50]
Total Series Resistance: 150 Ω

Function Annotations:
{'resistances': list[float] | tuple[float, ...], 'return': <class 'float'>}
```

Task 6 — Parallel Resistance

Take list/tuple of resistances.

Apply parallel formula (1/R).

Return equivalent resistance.

Add docstring + annotations.

Print annotations.

In [5]:
```python
def parallel(resistances: list[float] | tuple[float, ...]) -> float:
    """
    Calculate the equivalent resistance of resistors connected in parallel.

    Parameters:
```

```
        resistances (list[float] or tuple[float]): Resistance values in ohms.

    Returns:
        float: Equivalent parallel resistance.
    """
    return 1 / sum(1 / r for r in resistances)

R_list = [10, 20, 30]
print("Parallel Resistance:", parallel(R_list), "Ω")

print("\nAnnotations:")
print(parallel.__annotations__)
```

```
Parallel Resistance: 5.454545454545454 Ω

Annotations:
{'resistances': list[float] | tuple[float, ...], 'return': <class 'float'>}
```

Task 7 — Series Resistance with args

Use *args to accept any number of resistances.

Add all values.

Return total resistance.

Add docstring + annotations.

Print annotations.

In [6]:
```
def series(*args: float) -> float:
    return sum(args)
print("Series Resistance (*args):", series(10, 20, 30, 40), "Ω")
print("\nAnnotations:")
print(series.__annotations__)
```

```
Series Resistance (*args): 100 Ω

Annotations:
{'args': <class 'float'>, 'return': <class 'float'>}
```

Task 8 — Parallel Resistance with args

Use *args to input any number of resistances.

Apply parallel formula.

Return equivalent resistance.

Add docstring + annotations.

Print annotations.

In [7]:
```python
def parallel(*args: float) -> float:
    return 1 / sum(1 / r for r in args)
print("Parallel Resistance (*args):", parallel(10, 20, 30), "Ω")
print("\nAnnotations:")
print(parallel.__annotations__)
```

```
Parallel Resistance (*args): 5.454545454545454 Ω

Annotations:
{'args': <class 'float'>, 'return': <class 'float'>}
```

Task 9 — Solar Inverter Data (List/Tuple)

Function accepts a list/tuple.

Unpack SOC, generation, price.

Print values clearly.

Add docstring + annotations.

Print annotations.

In [ ]:
```python
def solar_data(data: list[int] | tuple[int, int, int]) -> None:
    """
    Print solar inverter data (SOC, solar generation, price).

    Parameters:
        data (list or tuple): [SOC, solar_generation, price]

    Returns:
        None
    """
    SOC, solar_gen, price = data
    print(f"Battery SOC: {SOC}%")
    print(f"Solar Generation: {solar_gen} kW")
    print(f"Electricity Price: {price} PKR/kWh")
solar_data([60, 2, 60])
print("\nAnnotations:")
print(solar_data.__annotations__)
```

```
Battery SOC: 60%
Solar Generation: 2 kW
Electricity Price: 60 PKR/kWh

Annotations:
{'data': list[int] | tuple[int, int, int], 'return': None}
```

Task 10 — Solar Inverter Data (Dictionary)

Function receives a dictionary.

Unpack using keys.

Print SOC, generation, and price.

Add docstring + annotations.

Print annotations.

In [ ]:
```python
def solar_data_dict(info: dict[str, int]) -> None:
    """
    Print solar inverter data from dictionary.

    Parameters:
        info (dict): Dictionary with keys "SOC", "solar_generation", "price".

    Returns:
        None
    """
    SOC, solar_gen, price = info["SOC"], info["solar_generation"], info["price
    print(f"Battery SOC: {SOC}%")
    print(f"Solar Generation: {solar_gen} kW")
    print(f"Electricity Price: {price} PKR/kWh")
solar_data_dict({"SOC": 60, "solar_generation": 2, "price": 60})
print("\nAnnotations:")
print(solar_data_dict.__annotations__)
```

```
Battery SOC: 60%
Solar Generation: 2 kW
Electricity Price: 60 PKR/kWh

Annotations:
{'info': dict[str, int], 'return': None}
```

Task 11 — Solar Inverter Data (Keyword Arguments)

Function receives keyword arguments.

Values collected into a dictionary.

Print SOC, generation, and price.

Add docstring + annotations.

Print annotations.

In [ ]:
```python
def solar(**kwargs: int) -> None:
    """
    Print solar inverter data using keyword arguments.

    Parameters:
        **kwargs (int): SOC, solar_generation, price.

    Returns:
        None
```

```
    """
    for key, value in kwargs.items():
        print(f"{key}: {value}")

solar(SOC=60, solar_generation=2, price=60)

print("\nAnnotations:")
print(solar.__annotations__)
```

Task 12 — Derivative Function dy/dx

Function takes x value.

Apply derivative formula 2x.

Return the result.

In [ ]:
```
def dy_dx(x: float) -> float:
    return 2 * x
print("dy/dx at x=1:", dy_dx(1))   # Expected 2
print("dy/dx at x=5:", dy_dx(5))   # Expected 10
print("dy/dx at x=-3:", dy_dx(-3)) # Expected -6
print("\nAnnotations:")
print(dy_dx.__annotations__)
```

```
dy/dx at x=1: 2
dy/dx at x=5: 10
dy/dx at x=-3: -6

Annotations:
{'x': <class 'float'>, 'return': <class 'float'>}
```

Function Arguments and Annotations:

Task 1: Function with All Types of Arguments:

def f(a, b, *args, **kwargs): print(f'a = {a}') print(f'b = {b}') print(f'args = {args}') print(f'kwargs = {kwargs}') f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)

What to observe:

• a and b are normal positional arguments. • Extra values are stored in args (tuple). • Named arguments are stored in kwargs (dictionary).

In [18]:
```
def f(a, b, *args, **kwargs):
    print(f'a = {a}')
    print(f'b = {b}')
    print(f'args = {args}')
    print(f'kwargs = {kwargs}')
f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)
```

```
a = 1
b = 2
args = ('foo', 'bar', 'baz', 'qux')
kwargs = {'x': 100, 'y': 200, 'z': 300}
```

Task 2: Function with Only *args: def f(*args): for i in args: print(i) a = [1, 2, 3] t = (4, 5, 6) s = {7, 8, 9} f(*a, *t, *s) What to observe: • You can unpack lists, tuples, and sets into a function. • *args collects them into a single tuple.

In [19]:
```
def f(*args):
    for i in args:
        print(i)
a = [1, 2, 3]
t = (4, 5, 6)
s = {7, 8, 9}
f(*a, *t, *s)
```

```
1
2
3
4
5
6
8
9
7
```

Task 3: Function with Only **kwargs: def f(**kwargs): for k, v in kwargs.items(): print(k, '->', v) d1 = {'a': 1, 'b': 2} d2 = {'x': 3, 'y': 4} f(**d1, **d2) What to observe: • **kwargs collects named arguments into a dictionary. • Dictionaries can be unpacked directly into a function.

In [20]:
```
def f(**kwargs):
    for k, v in kwargs.items():
        print(k, '->', v)
d1 = {'a': 1, 'b': 2}
d2 = {'x': 3, 'y': 4}
f(**d1, **d2)
```

```
a -> 1
b -> 2
x -> 3
y -> 4
```

Task 4: Mixing *args in Function Calls: def f(*args): for i in args: print(i) f(*[1, 2, 3], *[4, 5, 6]) What to observe: • You can pass multiple unpacked lists into *args. • All values are combined into a single tuple.

In [21]:
```
def f(*args):
    for i in args:
        print(i)
```

```
f(*[1, 2, 3], *[4, 5, 6])
```

```
1
2
3
4
5
6
```

Task 5: Mixing **kwargs in Function Calls: def f(**kwargs): for k, v in kwargs.items(): print(k, '->', v) f(**{'a': 1, 'b': 2}, **{'x': 3, 'y': 4}) What to observe: • Multiple dictionaries can be unpacked into one **kwargs.

In [22]:
```
def f(**kwargs):
    for k, v in kwargs.items():
        print(k, '->', v)
f(**{'a': 1, 'b': 2}, **{'x': 3, 'y': 4})
```

```
a -> 1
b -> 2
x -> 3
y -> 4
```

Task 6: Using Annotations: def f(a: int, b: str) -> float: print(a, b) return 3.5 f(1, 'foo') print(f.**annotations**) What to observe: • Annotations are type hints, not restrictions. • They show expected input/output types.

In [23]:
```
def f(a: int, b: str) -> float:
    print(a, b)
    return 3.5
f(1, 'foo')
print(f.__annotations__)
```

```
1 foo
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}
```

Task 7: Annotations with Default Values: def f(a: int = 12, b: str = 'baz') -> float: print(a, b) return 3.5 print(f.**annotations**) f() What to observe: • Default values work along with annotations. • Annotations do not enforce type rules.

In [24]:
```
def f(a: int = 12, b: str = 'baz') -> float:
    print(a, b)
    return 3.5
print(f.__annotations__)
f()
```

```
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}
12 baz
```

Out[24]:
```
3.5
```

Task 8: Annotations Ignored by Python: def f(a: int, b: str) -> float: print(a, b) return

1, 2, 3 f('foo', 2.5) What to observe: • Function annotations don’t force types. • You can break the rules, and Python will not complain.

In [25]:
```
def f(a: int, b: str) -> float:
    print(a, b)
    return 1, 2, 3
f('foo', 2.5)
```

```
foo 2.5
```

Out[25]:
```
(1, 2, 3)
```