Introduction to `Functions`

`Concept` :

Functions are reusable blocks of code that perform a specific task. They help in organizing code, making it more readable, and reducing redundancy.

`Scenario`

Imagine you're working in a bakery. Every time a customer places an order, you follow the same steps: take the order, process payment, and pack the goods. Instead of repeating these steps each time, you can create a function to handle the process

In [ ]:
```
def pass_butter(location):
    print(f'passing the butter to rick from {location}!')

pass_butter(location='top corner of the table')
```

```
passing the butter to rick from top corner of the table!
```

In [ ]:
```
#Function Definition
def process_order(order):
    print(f"Taking order for: {order}")
    print("Processing payment")
    print("Packing goods")
    print(f"Order for {order} completed")

# Using the function
process_order("cake")
process_order("cookies")
```

```
Taking order for: cake
Processing payment
Packing goods
Order for cake completed
Taking order for: cookies
Processing payment
Packing goods
Order for cookies completed
```

Function Definition and Calling

`Concept` :

Defining a Function: Use the `def` keyword followed by the function `name` and `parentheses` .

Calling a Function: Use the function `name` followed by `parentheses`

`Scenario` :

You want to automate the greeting process at your bakery, greeting each customer by their name.

In [ ]:
```python
def greet_customer(name):
    print(f"Hello, {name}! Welcome to our bakery.")

# Calling the function
greet_customer("Alice")
greet_customer("Bob")
```

```
Hello, Alice! Welcome to our bakery.
Hello, Bob! Welcome to our bakery.
```

Function Parameters and Arguments

`Concept` :

Functions can take `parameters` (inputs) to customize their behavior.

`Scenario`:

You want to create a function that calculates the total price of an order, considering the quantity and price per item.

In [ ]:
```
#Function definition
def calculate_total_price(quantity, price_per_item):
    total = quantity * price_per_item
    return total

# Calling the function and storing the result
total = calculate_total_price(5, 2.5)
print(f"The total price is: ${total}")
total1= calculate_total_price(2,2.5)
print(f"The total price is: ${total1}")
```

```
The total price is: $12.5
The total price is: $5.0
```

In [ ]:
```
def add(a,b):
    return a+b

def subtract(a,b):
    return a-b

add(5,6)
subtract(7,3)
```

Out[ ]: 4

Default Parameters

`Concept` :

Functions can have default parameter values, which are used if no argument is provided.

`Scenario` :

You want to create a function that greets customers, with a default greeting message.

In [ ]:
```
def greet_customer(name, message="Welcome to our bakery!"):
    print(f"Hello, {name}! {message}")

# Calling the function
greet_customer(name="Alice")
greet_customer(name="Bob", message="Hope you have a great day!")
```

```
Hello, Alice! Welcome to our bakery!
Hello, Bob! Hope you have a great day!
```

Keyword Arguments

`Concept` :

When calling functions, you can specify arguments using the parameter names.

`Scenario` :

You want to specify the order of items without worrying about their position.

In [ ]:
```
def place_order(item, quantity, price_per_item):
    total = quantity * price_per_item
    print(f"Order placed: {quantity} {item}(s) for ${total}")

# Using keyword arguments
```

```
place_order(item="cake", quantity=2, price_per_item=10)
#place_order(price_per_item=1.5, item="cookie",quantity=5)
```

```
{'item': 'cake', 'quantity': 2, 'price_per_item': 10}
Order placed: 2 cake(s) for $20
```

Variable-Length Arguments

`Concept` :

Functions can accept a variable number of arguments using `*args` for positional arguments and `**kwargs` for keyword arguments.

`Scenario` :

You want to create a function that can take multiple items and their quantities.

In [ ]:
```
# def place_order(*items):
#     #print(f"Order placed for the following items: {items}")
#     for item in items:
#         print(item)

# # Calling the function
# list=["cake", "cookies", "bread"]
# place_order("cake", "cookies", "bread")

#Ict={'a':['abracadabra','atrocious'],'b':['bonkers','beautiful']}


# def order_details(**details):
#     print(f"Order details:: {details}")
#     for key, value in details.items():
#         print(f"{key}: {value}")

# # Calling the function
# order_details(item="cake", quantity=2, price=20)
```

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


myDict={'a':['abracadabra','atrocious'],'b':['bonkers','beautiful']}
process_dict(a=['abracadabra','atrocious'],b=['bonkers','beautiful'])

print(myDict.keys())


#C

#*var-->Pointer

#python

#*var--> pass a list of names


#Python, C++, java --> Object Oriented Languagues(High-level)
#C --> low-level language.
#Haskell, Mathematica --> Functional Programing
```

```
a: ['abracadabra', 'atrocious']
b: ['bonkers', 'beautiful']
dict_keys(['a', 'b'])
```

Higher-Order Functions

`Concept` :

Functions that take other functions as arguments or return them.

`Scenario` :

You want to create a function that applies a discount to a total price using a discount function.

In [ ]:
```python
def apply_discount(total, discount_func):
    return discount_func(total)

# Discount functions
def ten_percent_discount(total):
    return total * 0.9

def twenty_percent_discount(total):
    return total * 0.8

# Using the higher-order function
print(apply_discount(100, ten_percent_discount))
print(apply_discount(100, twenty_percent_discount))
```

```
90.0
80.0
```

Lambda Functions

`Concept` :

Lambda functions are small anonymous functions defined using the lambda keyword.

`Scenario` :

You want a quick function to calculate the square of a number without defining a full function.

In [ ]:
```python
# #Example 1: Doubling Numbers
# res=lambda x: x * x

# print(res(4))
# print(res(7))


# #Example 2: Sorting a List of Tuples
# points = [(1, 2), (3, 1), (0, 4)]
# sorted_points = sorted(points, key=lambda p: p[1])
# print(sorted_points)
```

```
# #Example 3: Filtering Even Numbers
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda n: n % 2 == 0, numbers))
print(even_numbers)  #---> #[2,4,6]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[71], line 15
      1 # #Example 1: Doubling Numbers
      2 # res=lambda x: x * x
      3
   (...)
     12
     13 # #Example 3: Filtering Even Numbers
     14 numbers = [1, 2, 3, 4, 5, 6]
---> 15 even_numbers = list(filter(lambda n: n % 2 == 0, numbers))
     16 print(even_numbers)

TypeError: 'list' object is not callable
```

Understanding Scope

`Concept` :

`Scope` determines the visibility of variables. Variables defined inside a function are local, while those outside are global.

`Scenario` :

You want to understand how variables inside and outside a function interact.

In [ ]:
```
# Global variable
total_orders = 0

def process_order(order):
    global total_orders
    total_orders += 1
```

```
    print(f"Processing order for: {order}")

# Calling the function
process_order("cake")
process_order("cookies")
process_order("biscuits")
print(f"Total orders processed: {total_orders}")
```

```
Processing order for: cake
Processing order for: cookies
Processing order for: biscuits
Total orders processed: 3
```

In [ ]:
```
#Local scope
def myFunc(var=5):
    var1=0
    if var%2==0:
        var1+=1

myFunc()
```

In [ ]:
```
var=['hello','sir']

for x in var:
    print(f'{x} how are you')
```

```
hello how are you
sir how are you
```