Practice Exercises

Functions, Modules, and Collections in Python

INDIVIDUAL EXERCISES

Exercise 1: Create a function that takes two parameters for : name and age, and outputs a Birthday message " Happy Birthday 'name' I hear you are 'age' today!"

→ Start by defining the function using the 'def' keyword followed by the function name and its parameters enclosed in parentheses.
E.g.: *def birthday_message(name, age):*

→ Inside the function, construct the birthday message using string concatenation or string formatting. Include the name and age parameters in the message.
E.g.: *message = "Happy Birthday " + name + "! I hear you are " + str(age) + " today!"*

→ Use the return keyword to return the constructed message from the function.
E.g.: *return message*

→ Outside the function, call it with specific values for name and age parameters, and print the returned message. E.g.:

```
name = "Rob"
age = 30
print(birthday_message(name, age))
```

→ The complete code looks like this:

```
def birthday_message(name, age):
    #Generate a birthday message.
    message = "Happy Birthday " + name + "! I hear you are " + str(age) + " today!"
    return message

# Example usage:
name = "Alice"
age = 30
print(birthday_message(name, age))
```

Exercise 2: Create a function that takes two parameters: size and type of drink, and then outputs the drinks order.

→ Start by defining the function using the 'def' keyword followed by the function name and its parameters enclosed in parentheses.
E.g.: *def order_drink(size, drink_type):*

→ Inside the function, construct the drink order message using string concatenation or string formatting. Include the size and drink_type parameters in the message.

E.g,: *order = "You ordered a " + size + " " + drink_type + "."*

→ Use the return keyword to return the constructed message from the function.

E.g.: *return order*

→ Outside the function, call it with specific values for size and drink_type parameters, and print the returned message.e.g.:

```
size = "medium"
drink_type = "latte"
print(order_drink(size, drink_type))
```

→ The complete code looks like this:

```
def order_drink(size, drink_type):
   #Generate a drink order.
   order = "You ordered a " + size + " " + drink_type + "."
   return order

# Example usage:
size = "medium"
drink_type = "latte"
print(order_drink(size, drink_type))
```

Exercise 3: Create a cash machine

→ Define the withdraw_cash function: Start by defining a function called withdraw_cash that takes four parameters: pin, balance, pin_attempt, and amount. This function will handle the withdrawal process.

e.g.:*def withdraw_cash(pin, balance, pin_attempt, amount):*

→ Inside the withdraw_cash function, check if the pin_attempt matches the actual pin. If they match, proceed with the withdrawal process; otherwise, print an error message indicating an incorrect PIN.

E.g.: *if pin_attempt == pin:*

→ If the PIN is correct, check if the amount to withdraw is less than or equal to the balance. If there are sufficient funds, proceed with the withdrawal; otherwise, print an error message indicating insufficient funds.

E.g.: *if amount <= balance:*

→ If both the PIN and balance checks pass, update the balance by subtracting the amount to withdraw and print a message indicating that cash is being dispensed, along with the new balance. e.g.:

```
balance -= amount
print("Cash is being dispensed...")
print("New balance:", balance)
```

→ If the PIN is incorrect or there are insufficient funds, print the appropriate error message. E.g:

```
        else:
          print("Insufficient funds. Unable to withdraw.")
 else:
   print("Incorrect PIN. Transaction cancelled.")
```

→ *Define a main function to drive the program. Inside this function, set the initial pin and balance, take input from the user for the PIN attempt and amount to withdraw, and call the withdraw_cash function.e.g:*

```
def main():
    pin = 1234
    balance = 1000
    pin_attempt = int(input("Enter your PIN: "))
    amount = float(input("Enter the amount to withdraw: "))
    withdraw_cash(pin, balance, pin_attempt, amount)

if __name__ == "__main__":
    main()
```

→ The complete code looks like this:

```
def withdraw_cash(pin, balance, pin_attempt, amount):
    if pin_attempt == pin:
        if amount <= balance:
            balance -= amount
            print("Cash is being dispensed...")
            print("New balance:", balance)
        else:
            print("Insufficient funds. Unable to withdraw.")
    else:
        print("Incorrect PIN. Transaction cancelled.")

# Example usage:
def main():
    # Set PIN and initial balance
    pin = 1234
    balance = 1000

    # Take input from the user for PIN and amount to withdraw
    pin_attempt = int(input("Enter your PIN: "))
    amount = float(input("Enter the amount to withdraw: "))

    # Withdraw cash and update balance
    withdraw_cash(pin, balance, pin_attempt, amount)

if __name__ == "__main__":
```

main()

Exercise 4: On List

→ Create a tuple of your favorite sport games.
E.g.: *favorite_sports = ['Football', 'Basketball', 'Tennis', 'Cricket']*

→ *Print the tuple,*
E.g *print(favorite_sports)*

→ Add 'Hockey' to your list
E.g.: *favorite_sports.append('Hockey')*

→ *Print the tuple,*
E.g *print(favorite_sports)*

→ Access the first 4 elements.
E.g *: first_four_elements = favorite_sports[:4]*

→ *Print the tuple,*
E.g *print(first_four_elements)*

→ Access the only 5th element.
E.g.: *fifth_element = favorite_sports[4]*

→ *Print the tuple,*
E.g *print(fifth_element)*

→ Replace Hockey with Ice-Hockey
e.g.: *favorite_sports[favorite_sports.index('Hockey')] = 'Ice-Hockey'*

→ *Print the tuple,*
E.g print(favorite_sports)

→ Reverse the list
e.g.: *favorite_sports.reverse()*

→ *Print the tuple,*
E.g print(favorite_sports)

→ Sort the list,
e.g.: *favorite_sports.sort()*

→ *Print the tuple,*
E.g print(favorite_sports)

→ Remove Ice-Hockey off the list e.g.*: favorite_sports.pop(favorite_sports.index('Ice-Hockey'))*

→ *Print the tuple,*
E.g print(favorite_sports)

→ Clear all the elements from that list
e.g.: *favorite_sports.clear()*

→ *Print the tuple,*
E.g print(favorite_sports)

→

GROUP EXERCISES

Exercise 1: Roll The Dice

→ Start by importing the random module, which allows you to generate random numbers.

E.g.: *import random*

→ Create a function called roll_dice() to simulate rolling a dice. Inside this function, use random.randint(1, 6) to generate a random number between 1 and 6, inclusive.
e.g.: *return random.randint(1, 6)*

→ Create a function called main() to drive the program. Inside this function, call the roll_dice() function to get the result of the dice roll.
e.g:

```
def main():
   # Roll the dice
    result = roll_dice()
```

→ *After rolling the dice, check if the result is equal to 6. If it is, print "Congrats! Move 2 spaces!". If it's not, print "Try again!".e.g:*

```
if result == 6:
        print("Congrats! Move 2 spaces!")
    else:
        print("Try again!")
```

→ Add the conditional statement if __name__ == "__main__": at the bottom of your script to ensure that the main() function is only called when the script is executed directly, not when it's imported as a module. e.g:

```
if __name__ == "__main__":
    main()
```

→ The complete code looks like this:

```
import random

def roll_dice():
  #Simulate rolling a dice.
   return random.randint(1, 6)

def main():
   # Roll the dice
   result = roll_dice()

   # Check the result
   if result == 6:
      print("Congrats! Move 2 spaces!")
   else:
      print("Try again!")

if __name__ == "__main__":
   main()
```

Exercise 2: Tuple Scenario

→ Define tuples for different items in stock, each tuple should contain information in the format (name, price, quantity).

E.g.:

```
item1 = ('Apple', 0.5, 100)
item2 = ('Banana', 0.3, 150)
item3 = ('Milk', 1.2, 50)
item4 = ('Bread', 1.0, 80)
```

→ Print out the details of each item using string formatting.

E.g.:

```
print("Items in Stock:")
print("1. Name: {}, Price: ${}, Quantity: {}".format(item1[0], item1[1], item1[2]))
print("2. Name: {}, Price: ${}, Quantity: {}".format(item2[0], item2[1], item2[2]))
print("3. Name: {}, Price: ${}, Quantity: {}".format(item3[0], item3[1], item3[2]))
print("4. Name: {}, Price: ${}, Quantity: {}".format(item4[0], item4[1], item4[2]))
```

→ Calculate the total value of the inventory by multiply the price of each item by its quantity and sum them up.

E.g.:

```
 total_value = (item1[1] * item1[2]) + (item2[1] * item2[2]) + (item3[1] * item3[2])
+ (item4[1] * item4[2])
```

→ Identify items that need restocking by check the quantity of each item. If the quantity is less than 10, it needs restocking.

E.g.:

```
 items_to_restock = [item for item in [item1, item2, item3, item4] if item[2] < 10]
if items_to_restock:
    print("\nItems to Restock:")
    for item in items_to_restock:
        print("Name: {}, Quantity: {}".format(item[0], item[2]))
else:
    print("\nAll items are well stocked.")
```

→ The complete code looks like this:

```
# Define tuples for different items in stock
item1 = ('Apple', 0.5, 100)
item2 = ('Banana', 0.3, 150)
item3 = ('Milk', 1.2, 50)
item4 = ('Bread', 1.0, 80)

# Display the items in stock
print("Items in Stock:")
print("1. Name: {}, Price: ${}, Quantity: {}".format(item1[0], item1[1], item1[2]))
print("2. Name: {}, Price: ${}, Quantity: {}".format(item2[0], item2[1], item2[2]))
print("3. Name: {}, Price: ${}, Quantity: {}".format(item3[0], item3[1], item3[2]))
print("4. Name: {}, Price: ${}, Quantity: {}".format(item4[0], item4[1], item4[2]))

# Calculate the total value of the inventory
total_value = (item1[1] * item1[2]) + (item2[1] * item2[2]) + (item3[1] * item3[2]) +
(item4[1] * item4[2])
print("\nTotal Value of Inventory: ${}".format(total_value))

# Identify items that need restocking based on their quantity
items_to_restock = [item for item in [item1, item2, item3, item4] if item[2] < 10]
if items_to_restock:
        print("\nItems to Restock:")
        for item in items_to_restock:
                print("Name: {}, Quantity: {}".format(item[0], item[2]))
else:
        print("\nAll items are well stocked.")
```

Exercise 3: Set Scenario

→ You can create sets for regular attendees, VIP attendees, and speakers.

E.g.:

```
regular_attendees = {'Alice', 'Bob', 'Charlie'}
vip_attendees = {'David', 'Eva', 'Fiona'}
speakers = {'Grace', 'Henry', 'Isabel'}
```

→ Add attendees to the respective sets:.

E.g.:

```
regular_attendees.add('John')
vip_attendees.add('Kate')
speakers.add('James')
```

→ Perform set operations to analyze the attendee data such as finding common attendees between different categories or identifying unique attendees in each category.

E.g.:

```
# Find common attendees between regular attendees and VIP attendees
common_attendees = regular_attendees.intersection(vip_attendees)
print("Common Attendees:", common_attendees)

# Identify unique attendees in each category
unique_regular_attendees = regular_attendees.difference(vip_attendees,
speakers)
unique_vip_attendees = vip_attendees.difference(regular_attendees, speakers)
unique_speakers = speakers.difference(regular_attendees, vip_attendees)

print("Unique Regular Attendees:", unique_regular_attendees)
print("Unique VIP Attendees:", unique_vip_attendees)
print("Unique Speakers:", unique_speakers)
```

→ The complete code looks like this:

```
# Create sets representing different categories of attendees
regular_attendees = {'Alice', 'Bob', 'Charlie'}
vip_attendees = {'David', 'Eva', 'Fiona'}
speakers = {'Grace', 'Henry', 'Isabel'}

# Add attendees to the respective sets
regular_attendees.add('John')
vip_attendees.add('Kate')
speakers.add('James')

# Perform set operations to analyze the attendee data
common_attendees = regular_attendees.intersection(vip_attendees)
print("Common Attendees:", common_attendees)

unique_regular_attendees = regular_attendees.difference(vip_attendees,
speakers)
unique_vip_attendees = vip_attendees.difference(regular_attendees, speakers)
unique_speakers = speakers.difference(regular_attendees, vip_attendees)

print("Unique Regular Attendees:", unique_regular_attendees)
print("Unique VIP Attendees:", unique_vip_attendees)
print("Unique Speakers:", unique_speakers)
```

Exercise 4: Dictionary Scenario

→ Create a dictionary representing the inventory:

E.g.:

```
inventory = {
        'apple': 100,
        'banana': 150,
        'milk': 50,
         'bread': 80
}
```

→ Update the inventory

E.g.:

```
# New product arrived
inventory['orange'] = 120

# Sold some bread
inventory['bread'] -= 20
```

→ Perform dictionary operations to analyze the inventory data such as finding the total quantity of products, identifying low-stock items, or retrieving product information.

E.g.:

```
 # Total quantity of products
total_quantity = sum(inventory.values())
print("Total Quantity of Products:", total_quantity)

# Identify low-stock items (quantity less than 50)
low_stock_items = {product: quantity for product, quantity in inventory.items() if
quantity < 50}
print("Low-Stock Items:", low_stock_items)

# Retrieve product information
product_info = inventory.get('banana')
print("Product Information (Banana):", product_info)
```

→ The complete code looks like this:

```
# Create a dictionary representing the inventory
inventory = {
   'apple': 100,
   'banana': 150,
   'milk': 50,
   'bread': 80
}

# Update the inventory
```

```
inventory['orange'] = 120
inventory['bread'] -= 20

# Perform dictionary operations to analyze the inventory data
total_quantity = sum(inventory.values())
print("Total Quantity of Products:", total_quantity)

low_stock_items = {product: quantity for product, quantity in inventory.items() if
quantity < 50}
print("Low-Stock Items:", low_stock_items)

product_info = inventory.get('banana')
print("Product Information (Banana):", product_info)
```