1. Dictionary Comprehension Exercise

Write a dictionary comprehension that creates a dictionary where the keys are numbers from 1 to 10 and the values are their squares. Print the resulting dictionary.

```
dict = {i: i**2 for i in range(1,11)}
print(dict)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

2. List Comprehension Exercise

Create a list comprehension that generates a list of even numbers from 1 to 20. Print the resulting list.

```
List = [i for i in range(1,21) if i%2==0]
print(List)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

3. Nested Dictionary Comprehension

Write a nested dictionary comprehension that creates a dictionary where the keys are tuples of (x, y) for x in range(2) and y in range(2), and the values are the sum of x and y.

```
nested_dict = {(x, y): x + y for x in range(2) for y in range(2)}
print(nested_dict)
```

```
{(0, 0): 0, (0, 1): 1, (1, 0): 1, (1, 1): 2}
```

4. Lambda and filter()

Create a list of numbers from 1 to 10 and use the filter() function with a lambda expression to create a new list that contains only the odd numbers. Print the resulting list.

```
# Creating a list of numbers from 1 to 10
numbers = list(range(1, 11))

# Using filter() with a lambda expression to filter odd numbers
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))

# Printing the resulting list
print(odd_numbers)
```

```
[1, 3, 5, 7, 9]
```

5. Email Validation

Write a Python program that validates an email address using a regular expression.

The email should follow the pattern: username@domain.extension. Print "Valid

Email" if the email is valid, otherwise print "Invalid Email".

```
import re

def validate_email(email):
    # Define the regular expression for a valid email address
    email_regex = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    # Use the re.match() function to check if the email matches the regex
    if re.match(email_regex, email):
        return True
    else:
        return False

# Example usage
email = input("Enter an email address: ")
if validate_email(email):
    print(f"'{email}' is a valid email address.")
else:
    print(f"'{email}' is not a valid email address.")
```

```
Enter an email address: rishushrivastav18@gmail.com
'rishushrivastav18@gmail.com' is a valid email address.
```

6. Password Strength Checker

Create a function that checks if a password is strong based on the following criteria:

- At least one uppercase letter
- At least one lowercase letter
- At least one digit
- At least one special character
- Length between 8 and 18 characters

Use a regular expression to validate the password and print "Valid Password" or "Invalid Password".

```
import re

# Function to validate password strength
def check_password_strength(password):
    # Regular expression for a strong password
    pattern = (
        r'^(?=.*[A-Z])'          # At least one uppercase letter
        r'(?=.*[a-z])'           # At least one lowercase letter
        r'(?=.*\d)'              # At least one digit
        r'(?=.*[@$!%*?&#])'      # At least one special character
        r'[A-Za-z\d@$!%*?&#]{8,18}$'  # Length between 8 and 18
    )

    # Match the password against the pattern
    if re.match(pattern, password):
        print("Valid Password")
    else:
        print("Invalid Password")

# Get input from the user
password = input("Enter a password to check its strength: ")
check_password_strength(password)
```

```
Enter a password to check its strength: rishu@18
Invalid Password
```

7. Extracting URLs

Write a Python function that extracts all URLs from a given text. Use a regular expression pattern that matches typical URL formats (e.g., starting with http:// or https://). Test the function with a sample text containing multiple URLs.

```
import re

# Function to extract URLs from text
def extract_urls(text):
    # Regular expression pattern for URLs
    pattern = r'https?://[^\s<>"]+|www\.[^\s<>"]+'

    # Find all matches in the text
    urls = re.findall(pattern, text)

    return urls

# Sample text containing multiple URLs
sample_text = """
Here are some useful resources:
1. Visit https://www.python.org for Python documentation.
2. Check out https://www.github.com for version control repositories.
3. You can also visit www.example.com for a sample domain.
"""

# Extract and print the URLs
extracted_urls = extract_urls(sample_text)
print("Extracted URLs:", extracted_urls)
```

```
Extracted URLs: ['https://www.python.org', 'https://www.github.com', 'www.example.com']
```

8. UPI ID Validator

Implement a function that validates a UPI ID based on the following rules:

▪ It must contain an '@' symbol.

▪ It should not contain whitespace.

▪ It may or may not contain a dot (.) or hyphen (-).

Use a regular expression to check if the UPI ID is valid and print "Valid UPI ID" or "Invalid UPI ID".

```
import re

# Function to validate UPI ID
def validate_upi_id(upi_id):
    # Regular expression for UPI ID
    pattern = r'^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+$'

    # Validate the UPI ID using the regex pattern
    if re.match(pattern, upi_id) and ' ' not in upi_id:
        print("Valid UPI ID")
    else:
        print("Invalid UPI ID")

# Test the function
upi_id = input("Enter a UPI ID to validate: ")
validate_upi_id(upi_id)
```

```
Enter a UPI ID to validate: rishushrivastav@sbi
Valid UPI ID
```

9. Bank Account Class:

Implement a class BankAccount with attributes for account number, account holder's name, and balance. Include methods for depositing and withdrawing money, and ensure that the balance cannot go negative.

```
class BankAccount:
    def __init__(self, account_number, account_holder, balance=0.0):
        """
        Initialize the bank account with account number, holder's name, and balance.
        """
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        """
        Deposit money into the account. The amount must be positive.
        """
        if amount > 0:
            self.balance += amount
            print(f"Deposited ₹{amount}. New balance: ₹{self.balance}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        """
        Withdraw money from the account. Ensure the balance does not go negative.
        """
        if amount > 0:
            if amount <= self.balance:
                self.balance -= amount
                print(f"Withdrew ₹{amount}. New balance: ₹{self.balance}")
            else:
                print("Insufficient balance.")
        else:
            print("Withdrawal amount must be positive.")

    def display_balance(self):
        """
        Display the current balance of the account.
        """
        print(f"Account Balance: ₹{self.balance}")

# Example usage
account = BankAccount("1234567890", "John Doe", 5000.0)
account.display_balance()    # Display initial balance
account.deposit(2000)        # Deposit ₹2000
account.withdraw(3000)       # Withdraw ₹3000
account.withdraw(5000)       # Attempt to withdraw more than the balance
account.display_balance()    # Display final balance
```

```
Account Balance: ₹5000.0
Deposited ₹2000. New balance: ₹7000.0
Withdrew ₹3000. New balance: ₹4000.0
Insufficient balance.
Account Balance: ₹4000.0
```

10. Inheritance Example:

Create a base class Person with attributes for name and age. Derive a subclass Student that adds an attribute for student_id and a method to display student details. Create an object of the Student class and display its information.

```
class Person:
    def __init__(self, name, age):
        """
        Initialize the Person with a name and age.
        """
        self.name = name
        self.age = age

    def display_details(self):
        """
        Display the details of the person.
        """
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
```

```
class Student(Person):
    def __init__(self, name, age, student_id):
        """
        Initialize the Student with a name, age, and student ID.
        """
        super().__init__(name, age)  # Call the base class constructor
        self.student_id = student_id

    def display_student_details(self):
        """
        Display the details of the student.
        """
        self.display_details()  # Call the base class method
        print(f"Student ID: {self.student_id}")


# Create a Student object and display its information
student = Student("Alice", 21, "S12345")
student.display_student_details()
```

```
Name: Alice
Age: 21
Student ID: S12345
```

11. Polymorphism with Shapes

Create a base class Shape with a method area(). Derive two subclasses: Circle and Square, each implementing the area() method to calculate the area based on their 2 specific formulas. Create objects of both classes and demonstrate polymorphism by calling the area() method.

```
import math

class Shape:
    def area(self):
        """
        Base method for calculating the area of a shape.
        Should be overridden by subclasses.
        """
        raise NotImplementedError("Subclasses must implement this method.")

class Circle(Shape):
    def __init__(self, radius):
        """
        Initialize the Circle with a radius.
        """
        self.radius = radius

    def area(self):
        """
        Calculate and return the area of the circle.
        Formula: π * radius^2
        """
        return math.pi * (self.radius ** 2)

class Square(Shape):
    def __init__(self, side):
        """
        Initialize the Square with a side length.
        """
        self.side = side

    def area(self):
        """
        Calculate and return the area of the square.
        Formula: side^2
        """
        return self.side ** 2

# Demonstrate polymorphism
shapes = [
    Circle(5),  # Create a Circle object with radius 5
    Square(4)   # Create a Square object with side length 4
]

for shape in shapes:
    print(f"The area of the {type(shape).__name__} is: {shape.area():.2f}")
```

```
The area of the Circle is: 78.54
The area of the Square is: 16.00
```

12. Encapsulation in a Class

Implement a class Motorcycle with private attributes for color, engine_size, and max_speed. Provide public methods to set and get these attributes. Create an object of the class and demonstrate the use of encapsulation by accessing the attributes through the provided methods.

```
class Motorcycle:
    def __init__(self, color, engine_size, max_speed):
        """
        Initialize the Motorcycle with private attributes.
        """
        self.__color = color          # Private attribute
        self.__engine_size = engine_size  # Private attribute
        self.__max_speed = max_speed      # Private attribute

    # Getter methods
    def get_color(self):
        return self.__color

    def get_engine_size(self):
        return self.__engine_size

    def get_max_speed(self):
        return self.__max_speed

    # Setter methods
    def set_color(self, color):
        self.__color = color

    def set_engine_size(self, engine_size):
        self.__engine_size = engine_size

    def set_max_speed(self, max_speed):
        self.__max_speed = max_speed

# Demonstrate encapsulation
motorcycle = Motorcycle("Red", "500cc", "200 km/h")

# Access private attributes using public methods
print("Motorcycle Details:")
print(f"Color: {motorcycle.get_color()}")
print(f"Engine Size: {motorcycle.get_engine_size()}")
print(f"Max Speed: {motorcycle.get_max_speed()}")

# Modify private attributes using public methods
motorcycle.set_color("Blue")
motorcycle.set_engine_size("650cc")
motorcycle.set_max_speed("250 km/h")

# Display updated details
print("\nUpdated Motorcycle Details:")
print(f"Color: {motorcycle.get_color()}")
print(f"Engine Size: {motorcycle.get_engine_size()}")
print(f"Max Speed: {motorcycle.get_max_speed()}")
```

```
Motorcycle Details:
Color: Red
Engine Size: 500cc
Max Speed: 200 km/h

Updated Motorcycle Details:
Color: Blue
Engine Size: 650cc
Max Speed: 250 km/h
```

13. Basic Decorator Creation

Write a simple decorator called uppercase_decorator that converts the return value of a function to uppercase. Test it with a function that returns a string.

```python
# Define the decorator
def uppercase_decorator(func):
    def wrapper():
        # Call the original function and convert its return value to uppercase
        result = func()
        return result.upper()
    return wrapper

# Define a test function and apply the decorator
@uppercase_decorator
def greet():
    return "hello, world"

# Test the decorated function
print(greet())
```

```
HELLO, WORLD
```

14. Timing Decorator

Create a decorator named time_it that measures the execution time of a function. The decorator should print the time taken to execute the function when it is called.

```python
import time

# Define the timing decorator
def time_it(func):
    def wrapper(*args, **kwargs):
        # Record the start time
        start_time = time.time()

        # Call the original function
        result = func(*args, **kwargs)

        # Record the end time
        end_time = time.time()

        # Calculate and print the execution time
        execution_time = end_time - start_time
        print(f"Execution time of {func.__name__}: {execution_time:.6f} seconds")

        return result
    return wrapper

# Define a test function and apply the decorator
@time_it
def example_function(n):
    total = 0
    for i in range(n):
        total += i
    return total

# Test the decorated function
print(f"Result: {example_function(1000000)}")
```

```
Execution time of example_function: 0.067388 seconds
Result: 499999500000
```

15. Parameterized Decorator

Implement a parameterized decorator called repeat that takes an integer n as an argument and repeats the execution of the decorated function n times. Test it with a function that prints a message.

```python
# Define the parameterized decorator
```

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for i in range(n):
                print(f"Execution {i + 1}:")
                func(*args, **kwargs)
        return wrapper
    return decorator

# Define a test function and apply the parameterized decorator
@repeat(3)  # Repeat the function 3 times
def print_message():
    print("Hello, this is a repeated message!")

# Test the decorated function
print_message()
```

```
Execution 1:
Hello, this is a repeated message!
Execution 2:
Hello, this is a repeated message!
Execution 3:
Hello, this is a repeated message!
```

16. Caching with functools.lru_cache

Use the @lru_cache decorator from the functools module to create a function that computes the nth Fibonacci number. Test the function with various inputs to demonstrate the caching effect.

```
from functools import lru_cache

# Define a function to compute the nth Fibonacci number with caching
@lru_cache(maxsize=None)  # Unlimited cache size
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Test the function with various inputs
print(f"Fibonacci(10): {fibonacci(10)}")
print(f"Fibonacci(20): {fibonacci(20)}")
print(f"Fibonacci(30): {fibonacci(30)}")
print(f"Fibonacci(10): {fibonacci(10)} (cached result)")
```

```
Fibonacci(10): 55
Fibonacci(20): 6765
Fibonacci(30): 832040
Fibonacci(10): 55 (cached result)
```

17. Registration Decorator

Create a registration decorator called register that keeps track of all functions it decorates in a global list. Write a function that returns the list of registered functions.

```
# Global list to store registered functions
registered_functions = []

# Define the registration decorator
def register(func):
    def wrapper(*args, **kwargs):
        registered_functions.append(func)  # Register the function
        return func(*args, **kwargs)
    return wrapper

# Define some test functions and apply the register decorator
@register
def greet():
    print("Hello!")

@register
def farewell():
    print("Goodbye!")

# Function to get the list of registered functions
def get_registered_functions():
    return registered_functions

# Test the functions
greet()
farewell()

# Retrieve and print the list of registered functions
print("Registered Functions:")
for func in get_registered_functions():
    print(func.__name__)
```

```
Hello!
Goodbye!
Registered Functions:
greet
farewell
```

18. Closure Example

Write a function make_counter that returns a nested function (closure) that counts how many times it has been called. The outer function should return the inner function, and the inner function should print the current count each time it is called.

```
def make_counter():
    count = 0  # Outer function keeps track of the count

    def counter():
        nonlocal count  # Use the nonlocal keyword to modify the outer count
        count += 1
        print(f"Count: {count}")

    return counter

# Create a counter instance
counter_function = make_counter()

# Call the counter multiple times
counter_function()  # Output: Count: 1
counter_function()  # Output: Count: 2
counter_function()  # Output: Count: 3
```

```
Count: 1
Count: 2
Count: 3
```

19. Stacked Decorators

Create two decorators: bold and italic. The bold decorator should wrap the output of a function in **tags, and the italic decorator should wrap it in *tags. Stack these decorators on a function that returns a string, and test the output.***

```
# Bold decorator
def bold(func):
    def wrapper():
        return f"<b>{func()}</b>"
    return wrapper

# Italic decorator
def italic(func):
    def wrapper():
        return f"<i>{func()}</i>"
    return wrapper

# Applying both decorators
```

```
@bold
@italic
def display_text():
    return "Hello, World!"

# Test the decorated function
print(display_text())
```

```
<b><i>Hello, World!</i></b>
```

20. Multiple Dispatch Decorator

Implement a simple version of a multiple dispatch decorator that can handle different types of arguments. Create a function that takes either an integer or a string and returns a different message based on the type of the argument. Use the decorator to manage the different behaviors.

```
# Multiple dispatch decorator
def multiple_dispatch(func):
    def wrapper(*args, **kwargs):
        if all(isinstance(arg, int) for arg in args):
            return func('integer', *args, **kwargs)
        elif all(isinstance(arg, str) for arg in args):
            return func('string', *args, **kwargs)
        else:
            return "Unsupported types"
    return wrapper

# Function with different behaviors based on argument type
@multiple_dispatch
def message_dispatch(type, *args):
    if type == 'integer':
        return f"Processing integer: {sum(args)}"
    elif type == 'string':
        return f"Processing string: {', '.join(args)}"

# Test cases
print(message_dispatch(1, 2, 3))       # Output: Processing integer: 6
print(message_dispatch("Hello", "World"))  # Output: Processing string: Hello, World
print(message_dispatch(1, "Hello"))    # Output: Unsupported types
```

```
Processing integer: 6
Processing string: Hello, World
Unsupported types
```

Start coding or generate with AI.