Q1 solve:

```
def calculate_square(a, b):
    return a**2 + b**2 + 2*a*b

a = int(input("Enter value for a: "))
b = int(input("Enter value for b: "))
print("Result:", calculate_square(a, b))
```

Q2 solve:

```
square_formula = lambda a, b: a**2 + b**2 + 2*a*b

a = int(input("Enter value for a: "))
b = int(input("Enter value for b: "))
print("Result using lambda:", square_formula(a, b))
```

Q3 solve:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
n = int(input("Enter a number: "))
print(f"Factorial of {n} is {factorial(n)}")
```

Q4 solve:

```
def is_prime(number):
    if number <= 1:
        return False
    for i in range(2, int(number**0.5) + 1):
        if number % i == 0:
            return False
    return True


number = int(input("Enter a number: "))
if is_prime(number):
    print(f"{number} is a prime number.")
else:
```

```
    print(f"{number} is not a prime number.")
```

List

Q1 solve:

```
a = [1, 3, 5, 7, 4]
print(a[-2], a[2])
```

Q2 solve:

```
print(len(a), type(a))
```

Q3 solve:

```
a[3] = 50
a[2] = 9
print(a)
```

Q4 solve:

```
a.append(100)
a.insert(2, 200)
print(a)
```

Q5 solve:

```
a.pop()  # Remove last element
a.pop(1)  # Remove element at index 1
print(a)
```

Q6 solve:

```
new_list = [2, 4, 6]
a.extend(new_list)
print(a)
```

Q7 solve:

```
b = a.copy()
b.sort()
print(b)
```

Q8 solve:

```
for element in a:
    if element == 5:
        break
    print(element)
```

Q9 solve:

```
print(max(a))
```

Tuple

Q1 solve:

```
print(sum(i for i in a if i % 2 != 0))
```

Q2:

```
print(a.index(5))  # Replace 5 with the value to search
```

Q3:

```
odd_count = sum(1 for i in a if i % 2 != 0)
even_count = len(a) - odd_count
print(odd_count, even_count)
```

Q4:

```
a += (2, 4, 6)
print(a)
```

Q5:

```
a = a[:2] + (400,) + a[2:]
print(a)
```

Q6:

```
a = a[:-1]
print(a)
```

Q7:

```
print(a[-4::-1])
```

Q8:

```
for i in a:
    if i == 5:
        continue
    print(i)
```

Set

Q1 solve:

```
a = {1, 3, 5, 8, 3, 7}
b = {0, False, 1, 5}
```

```
print("Set a:", a)
print("Set b:", b)
print("Length of a:", len(a), "Type of a:", type(a))
print("Length of b:", len(b), "Type of b:", type(b))

a.add(10)
print("Set a after adding 10:", a)

a.discard(8)  # Using discard to avoid errors if 8 isn't in the set
print("Set a after removing 8:", a)

print("Union:", a.union(b))
print("Intersection:", a.intersection(b))
print("Difference (a - b):", a.difference(b))
print("Symmetric Difference:", a.symmetric_difference(b))
print("Is b a subset of a?:", b.issubset(a))


new_list = [2, 3, 4]
a.update(new_list)
print("Set a after adding the new list:", a)
```

Dictionary

```
employee = {
    "name": "A",
    "age": 40,
    "type": {"developer": ["iOS", "android"]},
    "permanent": True,
    "salary": 30000,
    100: (1, 2, 3),
    4.5: [2, 3, 1]
}


print("Length of employee:", len(employee))
print("Type of employee:", type(employee))

print("Accessing key 'type':", employee["type"]["developer"])

employee["permanent"] = False
print("After changing permanent:", employee)
```

```
employee["gender"] = "male"
print("After adding gender:", employee)


employee.pop("age", None)  # Avoid KeyError if key doesn't exist
print("After removing age:", employee)

print("Keys:", list(employee.keys()))
print("Values:", list(employee.values()))
print("Items:", list(employee.items()))

print("Iterating through dictionary:")
for key, value in employee.items():
    print(f"{key}: {value}")
```

string

```
a = "hello"
b = "b2b2b2"
c = "3g3g"

d = a + b + c
print("Concatenated string d:", d)

print("Length of d:", len(d))
print("Slice of d:", d[:])

print("'a2' in d:", "a2" in d)

print("Uppercase:", d.upper())
print("Lowercase:", d.lower())
print("Title:", d.title())
print("Strip:", d.strip())
print("Is digit:", d.isdigit())
print("Find '3g':", d.find("3g"))
print("Capitalize:", d.capitalize())
print("Is alnum:", d.isalnum())
print("Count 'b2':", d.count("b2"))
print("Split:", d.split())
print("Swapcase:", d.swapcase())
print("Lstrip:", d.lstrip())
```

```
print("Replace 'hello' with 'python':", d.replace("hello", "python"))
```

Class and object

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def display_detail(self):
        print(f"Name: {self.name}, Price: {self.price}")

class ElectronicProduct(Product):
    def __init__(self, name, price, warranty):
        super().__init__(name, price)
        self.warranty = warranty

    def display_detail(self):
        super().display_detail()
        print(f"Warranty: {self.warranty} years")

product = Product("Generic Product", 100)
product.display_detail()

electronic_product = ElectronicProduct("Laptop", 1200,

class Shape:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def display_info(self):
        print(f"Shape: {self.name}")


class Rectangle(Shape):
    def __init__(self, name, length, width):
        super().__init__(name)
        self.length = length
        self.width = width

    def area(self):
```

```
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

    def display_info(self):
        super().display_info()
        print(f"Length: {self.length}, Width: {self.width}, Area: {self.area()}, Perimeter:
{self.perimeter()}")

rectangle = Rectangle("Rectangle", 10, 5)
rectangle.display_info()
```

Inheritance

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def display(self):
        print(f"Name: {self.first_name} {self.last_name}")


class Student(Person):
    def __init__(self, first_name, last_name, graduation_year):
        super().__init__(first_name, last_name)
        self.graduation_year = graduation_year

    def display(self):
        super().display()
        print(f"Graduation Year: {self.graduation_year}")

class Teacher(Person):
    def __init__(self, first_name, last_name, joining_year):
        super().__init__(first_name, last_name)
        self.joining_year = joining_year

    def display(self):
        super().display()
        print(f"Joining Year: {self.joining_year}")

class Alumni(Student):
```

```
    def __init__(self, first_name, last_name, graduation_year, passing_year):
        super().__init__(first_name, last_name, graduation_year)
        self.passing_year = passing_year

    def display(self):
        super().display()
        print(f"Passing Year: {self.passing_year}")

class CurrentStudent(Student):
    def __init__(self, first_name, last_name, graduation_year, current_semester):
        super().__init__(first_name, last_name, graduation_year)
        self.current_semester = current_semester

    def display(self):
        super().display()
        print(f"Current Semester: {self.current_semester}")

student = CurrentStudent("Alice", "Smith", 2025, "6th")
student.display()

teacher = Teacher("Bob", "Johnson", 2010)
teacher.display()
```

Encapsulation

```
class Vehicle:
    def __init__(self, color):
        self.color = color

    def vehicle_info(self):
        print(f"Color: {self.color}")


class Taxi(Vehicle):
    def __init__(self, color, model, capacity, variant):
        super().__init__(color)
        self.__model = model
        self.__capacity = capacity
        self.__variant = variant


    def get_model(self):
        return self.__model
```

```
    def set_model(self, model):
        self.__model = model

    def get_capacity(self):
        return self.__capacity

    def set_capacity(self, capacity):
        self.__capacity = capacity

    def get_variant(self):
        return self.__variant

    def set_variant(self, variant):
        self.__variant = variant

    def vehicle_info(self):
        super().vehicle_info()
        print(f"Model: {self.__model}, Capacity: {self.__capacity}, Variant: {self.__variant}")


taxi1 = Taxi("Yellow", "Sedan", 4, "Basic")
taxi1.vehicle_info()

taxi2 = Taxi("Black", "SUV", 6, "Premium")
taxi2.vehicle_info()
```

polymorphism

Q1 solve:

```
class Department:
    def display_name(self):
        print("This is a Department")

class Teacher(Department):
    def display_name(self):
        print("This is a Teacher")

class Author(Department):
    def display_name(self):
        print("This is an Author")

def display(department_obj):
    department_obj.display_name()
```

```
obj1 = Teacher()
obj2 = Author()

display(obj1)  # Output: This is a Teacher
display(obj2)  # Output: This is an Author
```

Q2 solve:

```
class TeacherAuthor:
    def write_article(self):
        print("Writing an article...")

    def publish_blog(self):
        print("Publishing a blog...")

    def schedule_class(self):
        print("Scheduling a class...")

obj = TeacherAuthor()
obj.write_article()
obj.publish_blog()
obj.schedule_class()
```

Q3 solve:

```
class Teacher:
    def profile(self):
        print("Teacher's Profile")

class Author:
    def profile(self):
        print("Author's Profile")

class TeacherAuthor(Teacher, Author):
    pass

obj = TeacherAuthor()
obj.profile()  # Output: Teacher's Profile (due to method resolution order in Python)
```

Exception handling

Q1 solve:

```
class InvalidVoterException(Exception):
```

```
    pass

age = int(input("Enter your age: "))
try:
    if age < 18:
        raise InvalidVoterException("Age must be 18 or above to vote!")
    else:
        print("You are eligible to vote.")
except InvalidVoterException as e:
    print(e)
```

Q2 solve:

```
class SalaryNotInRange(Exception):
    pass

salary = int(input("Enter your salary: "))
try:
    if salary < 10000 or salary > 50000:
        raise SalaryNotInRange("Salary must be between 10000 and 50000!")
    else:
        print(f"Your salary is {salary}.")
except SalaryNotInRange as e:
    print(e)
```

Q3 solve:

```
arr = [10, 5, 15, 20]
divisor = int(input("Enter a divisor: "))

try:
    for num in arr:
        print(num / divisor)
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid value for division!")
except Exception as e:
    print(f"An error occurred: {e}")
```

Custom exception handling

```
class InsufficientBalanceException(Exception):
    pass
```

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientBalanceException("Not enough balance!")
        else:
            self.balance -= amount
            print(f"Withdrawn: {amount}. Remaining Balance: {self.balance}")

account = BankAccount(5000)
try:
    account.withdraw(6000)
except InsufficientBalanceException as e:
    print(e)
```

Numpy function

```
 import numpy as np
score = np.array([85, 90, 78, 92, 88])

Qa solve:
score_float = score.astype(float)
print("Float Score:", score_float)

Qb solve:
a_score = score.copy() + 5
print("Original Score:", score)
print("Updated a_score:", a_score)

QC solve:

print("Shape:", score.shape)
print("Number of Dimensions (ndim):", score.ndim)
print("Size (Total elements):", score.size)
print("Item Size (Bytes):", score.itemsize)
print("Data Type (dtype):", score.dtype)
print("Sorted Score:", np.sort(score))

Qd solve:

indices = np.where(score > 80)
print("Indices with scores > 80:", indices[0])
```

Qe solve:

```
print("Minimum:", np.min(score))
print("Maximum:", np.max(score))
print("Standard Deviation:", np.std(score))
print("Variance:", np.var(score))
print("Sum:", np.sum(score))
print("Mean:", np.mean(score))
```

Qf solve:

```
print("Axis-wise Mean:", np.mean(score, axis=0))
```

score[::2] -> Every second element

```
print("Every second element (score[::2]):", score[::2])
```

score[-3::-1] -> Reverse from the 3rd last element

```
print("Reversed from the 3rd last (score[-3::-1]):", score[-3::-1])
```

score[1:4] -> Elements from index 1 to 3

```
print("Elements from index 1 to 3 (score[1:4]):", score[1:4])
```