**Assignment 8.4**

**2403A52110**

**Batch 04**

* **Email Validator**:  
  **Task:** to create a email validator.

**Prompt:** Create test cases to validate an email address (e.g., contains @ and .com); Use AI assistance to implement the validate\_email() function.

**Code:**





**Test:***import* unittest

*from* email\_validator *import* validate\_email

class TestEmailValidation(unittest.TestCase):

    """Test cases for the email validation function."""

    def test\_valid\_emails(*self*):

        """Test valid email addresses."""

*# Standard valid emails*

        self.assertTrue(validate\_email("user@example.com"))

        self.assertTrue(validate\_email("test@domain.org"))

        self.assertTrue(validate\_email("admin@company.net"))

        self.assertTrue(validate\_email("info@website.co.uk"))

*# Emails with numbers and special characters*

        self.assertTrue(validate\_email("user123@example.com"))

        self.assertTrue(validate\_email("test.email@domain.com"))

        self.assertTrue(validate\_email("user+tag@example.com"))

        self.assertTrue(validate\_email("user\_name@example.com"))

*# Short domain names*

        self.assertTrue(validate\_email("a@b.co"))

        self.assertTrue(validate\_email("test@example.co"))

*# Long but valid emails*

        self.assertTrue(validate\_email("very.long.email.address@very.long.domain.name.com"))

    def test\_invalid\_emails\_missing\_at(*self*):

        """Test emails missing @ symbol."""

        self.assertFalse(validate\_email("userexample.com"))

        self.assertFalse(validate\_email("test.domain.com"))

        self.assertFalse(validate\_email("invalid-email"))

        self.assertFalse(validate\_email("justtext"))

    def test\_invalid\_emails\_missing\_domain(*self*):

        """Test emails missing domain part."""

        self.assertFalse(validate\_email("user@"))

        self.assertFalse(validate\_email("test@"))

        self.assertFalse(validate\_email("@domain.com"))

        self.assertFalse(validate\_email("@"))

    def test\_invalid\_emails\_missing\_extension(*self*):

        """Test emails missing .com or similar extension."""

        self.assertFalse(validate\_email("user@domain"))

        self.assertFalse(validate\_email("test@example"))

        self.assertFalse(validate\_email("admin@company"))

    def test\_invalid\_emails\_multiple\_at(*self*):

        """Test emails with multiple @ symbols."""

        self.assertFalse(validate\_email("user@@example.com"))

        self.assertFalse(validate\_email("test@domain@com"))

        self.assertFalse(validate\_email("user@example@domain.com"))

    def test\_invalid\_emails\_invalid\_characters(*self*):

        """Test emails with invalid characters."""

        self.assertFalse(validate\_email("user name@example.com"))

        self.assertFalse(validate\_email("user@exam ple.com"))

        self.assertFalse(validate\_email("user@example.c om"))

        self.assertFalse(validate\_email("user@example..com"))

        self.assertFalse(validate\_email("user@@example.com"))

    def test\_invalid\_emails\_empty\_or\_whitespace(*self*):

        """Test empty or whitespace-only emails."""

        self.assertFalse(validate\_email(""))

        self.assertFalse(validate\_email("   "))

        self.assertFalse(validate\_email(" "))

        self.assertFalse(validate\_email("\t"))

        self.assertFalse(validate\_email("\n"))

    def test\_invalid\_emails\_edge\_cases(*self*):

        """Test edge cases and malformed emails."""

        self.assertFalse(validate\_email("@.com"))

        self.assertFalse(validate\_email("user@.com"))

        self.assertFalse(validate\_email("@example.com"))

        self.assertFalse(validate\_email("user@example."))

        self.assertFalse(validate\_email(".user@example.com"))

        self.assertFalse(validate\_email("user.@example.com"))

        self.assertFalse(validate\_email("user@example.com."))

    def test\_invalid\_emails\_wrong\_extension(*self*):

        """Test emails with invalid or missing extensions."""

        self.assertFalse(validate\_email("user@example"))

        self.assertFalse(validate\_email("user@example."))

        self.assertFalse(validate\_email("user@example.c"))

        self.assertFalse(validate\_email("user@example.123"))

    def test\_valid\_emails\_various\_extensions(*self*):

        """Test valid emails with different extensions."""

        self.assertTrue(validate\_email("user@example.com"))

        self.assertTrue(validate\_email("user@example.org"))

        self.assertTrue(validate\_email("user@example.net"))

        self.assertTrue(validate\_email("user@example.edu"))

        self.assertTrue(validate\_email("user@example.gov"))

        self.assertTrue(validate\_email("user@example.co.uk"))

        self.assertTrue(validate\_email("user@example.info"))

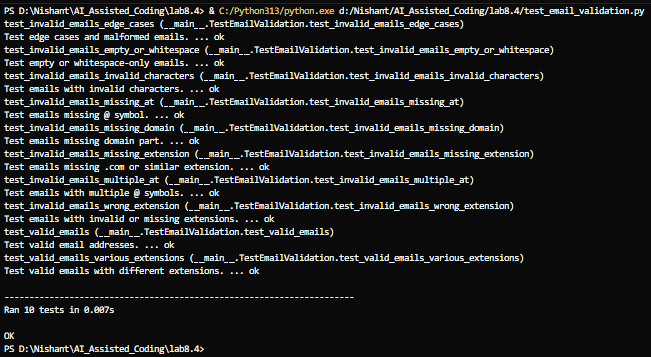
        self.assertTrue(validate\_email("user@example.biz"))

*if* \_\_name\_\_ == '\_\_main\_\_':

*# Run the tests*

    unittest.main(*verbosity*=2)

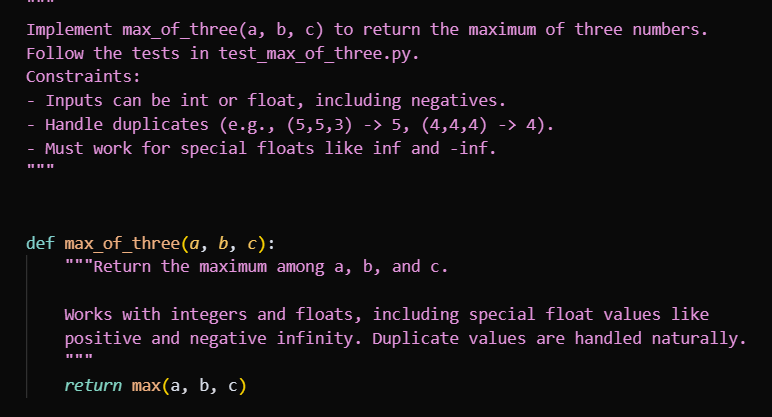
OP:



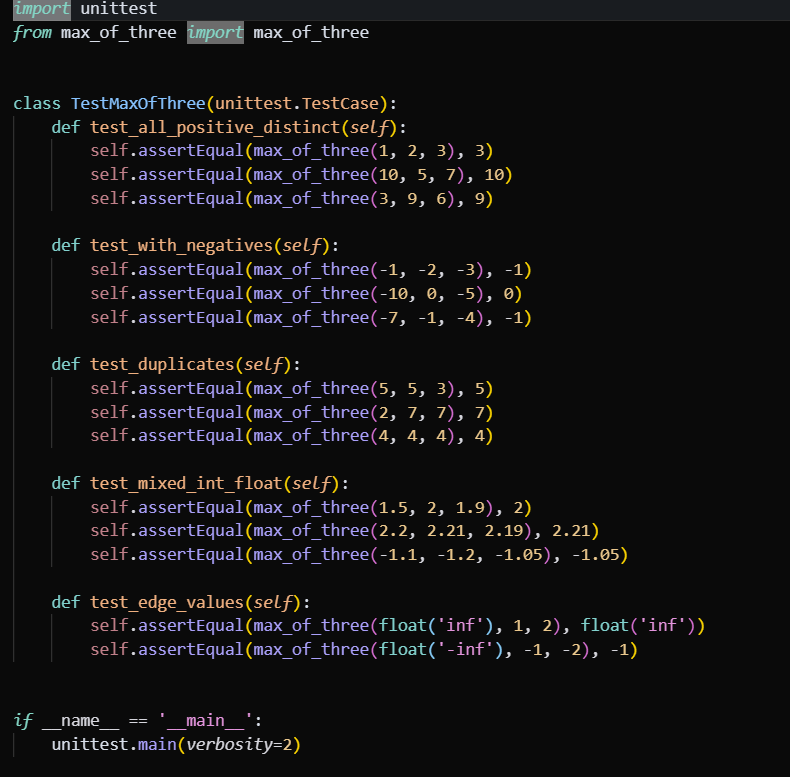
* **Max of three:**

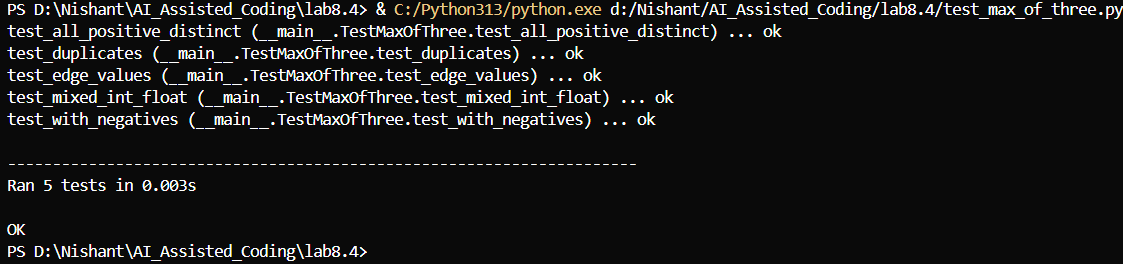
**Task:** To find the max of three:

**Prompt:** Write test cases for a function that returns the maximum of three numbers. Prompt Copilot/Cursor to write the logic based on tests



Test:



Op:  


* **Shopping cart:  
  Task:** to create a code for shopping cart.

**Prompt:** Use TDD to write a shopping cart class with methods to add, remove, and get total price.First write tests for each method, then generate code using AI.

**CODE:**

class ShoppingCart:

    """Simple shopping cart to add, remove items and compute total price."""

    def \_\_init\_\_(*self*):

*# items: dict[name] = {"price": float, "quantity": int}*

        self.\_items = {}

    def add\_item(*self*, *name*, *price*, *quantity*=1):

*if* price <= 0:

*raise* ValueError("price must be positive")

*if* quantity <= 0:

*raise* ValueError("quantity must be positive")

*if* name in self.\_items:

*# Accumulate quantity, keep latest price for simplicity*

            self.\_items[name]["quantity"] += quantity

            self.\_items[name]["price"] = price

*else*:

            self.\_items[name] = {"price": float(price), "quantity": int(quantity)}

    def remove\_item(*self*, *name*, *quantity*=1):

*if* quantity <= 0:

*raise* ValueError("quantity must be positive")

*if* name not in self.\_items:

*return*  *# noop*

        current\_qty = self.\_items[name]["quantity"]

*if* quantity >= current\_qty:

*del* self.\_items[name]

*else*:

            self.\_items[name]["quantity"] = current\_qty - quantity

    def get\_total\_price(*self*):

        total = 0.0

*for* item *in* self.\_items.values():

            total += item["price"] \* item["quantity"]

*return* round(total, 2)

**Test:**

*import* unittest

*from* shopping\_cart *import* ShoppingCart

class TestShoppingCart(unittest.TestCase):

    def setUp(*self*):

        self.cart = ShoppingCart()

    def test\_initial\_total\_is\_zero(*self*):

        self.assertEqual(self.cart.get\_total\_price(), 0.0)

    def test\_add\_single\_item(*self*):

        self.cart.add\_item("apple", *price*=1.20, *quantity*=1)

        self.assertEqual(self.cart.get\_total\_price(), 1.20)

    def test\_add\_multiple\_quantities(*self*):

        self.cart.add\_item("banana", *price*=0.50, *quantity*=4)

        self.assertEqual(self.cart.get\_total\_price(), 2.00)

    def test\_add\_multiple\_different\_items(*self*):

        self.cart.add\_item("milk", *price*=2.50, *quantity*=1)

        self.cart.add\_item("bread", *price*=1.75, *quantity*=2)

        self.assertEqual(self.cart.get\_total\_price(), 2.50 + 1.75 \* 2)

    def test\_add\_same\_item\_accumulates\_quantity(*self*):

        self.cart.add\_item("eggs", *price*=0.25, *quantity*=6)

        self.cart.add\_item("eggs", *price*=0.25, *quantity*=6)

        self.assertEqual(self.cart.get\_total\_price(), 0.25 \* 12)

    def test\_remove\_item\_reduces\_total(*self*):

        self.cart.add\_item("cheese", *price*=3.00, *quantity*=1)

        self.cart.add\_item("ham", *price*=2.00, *quantity*=2)

        self.cart.remove\_item("ham", *quantity*=1)

        self.assertEqual(self.cart.get\_total\_price(), 3.00 + 2.00)

    def test\_remove\_item\_entirely\_when\_quantity\_exceeds(*self*):

        self.cart.add\_item("juice", *price*=1.50, *quantity*=2)

        self.cart.remove\_item("juice", *quantity*=5)

        self.assertEqual(self.cart.get\_total\_price(), 0.0)

    def test\_remove\_nonexistent\_item\_is\_noop(*self*):

        self.cart.add\_item("yogurt", *price*=1.00, *quantity*=3)

        self.cart.remove\_item("not-in-cart", *quantity*=1)

        self.assertEqual(self.cart.get\_total\_price(), 3.00)

    def test\_invalid\_add\_raises(*self*):

*with* self.assertRaises(ValueError):

            self.cart.add\_item("apple", *price*=-1.0, *quantity*=1)

*with* self.assertRaises(ValueError):

            self.cart.add\_item("apple", *price*=1.0, *quantity*=0)

    def test\_invalid\_remove\_raises(*self*):

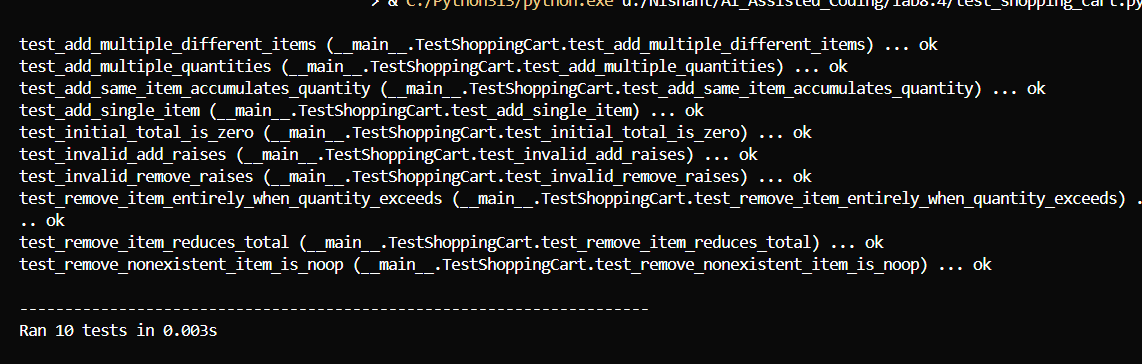
*with* self.assertRaises(ValueError):

            self.cart.remove\_item("apple", *quantity*=0)

*if* \_\_name\_\_ == '\_\_main\_\_':

    unittest.main(*verbosity*=2)

**Op:**



* **Square:**

**Task:** To find the square of the numbers.

**Prompt:** Write a test case to check if a function returns the square of a number. Then write the function

**CODE:**  
def square(*number*):

    """

    Calculate the square of a number.

    Args:

        number (int or float): The number to be squared

    Returns:

        int or float: The square of the input number

    Examples:

*>>>* square(5)

        25

*>>>* square(-3)

        9

*>>>* square(2.5)

        6.25

    """

*return* number \*\* 2

Test:  
*import* unittest

*from* square\_function *import* square

class TestSquareFunction(unittest.TestCase):

    """Test cases for the square function."""

    def test\_positive\_integers(*self*):

        """Test square function with positive integers."""

        self.assertEqual(square(1), 1)

        self.assertEqual(square(2), 4)

        self.assertEqual(square(5), 25)

        self.assertEqual(square(10), 100)

    def test\_negative\_integers(*self*):

        """Test square function with negative integers."""

        self.assertEqual(square(-1), 1)

        self.assertEqual(square(-2), 4)

        self.assertEqual(square(-5), 25)

        self.assertEqual(square(-10), 100)

    def test\_zero(*self*):

        """Test square function with zero."""

        self.assertEqual(square(0), 0)

    def test\_positive\_floats(*self*):

        """Test square function with positive floating point numbers."""

        self.assertAlmostEqual(square(1.5), 2.25, *places*=5)

        self.assertAlmostEqual(square(2.5), 6.25, *places*=5)

        self.assertAlmostEqual(square(0.5), 0.25, *places*=5)

    def test\_negative\_floats(*self*):

        """Test square function with negative floating point numbers."""

        self.assertAlmostEqual(square(-1.5), 2.25, *places*=5)

        self.assertAlmostEqual(square(-2.5), 6.25, *places*=5)

        self.assertAlmostEqual(square(-0.5), 0.25, *places*=5)

    def test\_large\_numbers(*self*):

        """Test square function with large numbers."""

        self.assertEqual(square(1000), 1000000)

        self.assertEqual(square(-1000), 1000000)

    def test\_small\_numbers(*self*):

        """Test square function with very small numbers."""

        self.assertAlmostEqual(square(0.001), 0.000001, *places*=8)

        self.assertAlmostEqual(square(-0.001), 0.000001, *places*=8)

*if* \_\_name\_\_ == '\_\_main\_\_':

*# Run the tests*

    unittest.main(*verbosity*=2)

OP:  
