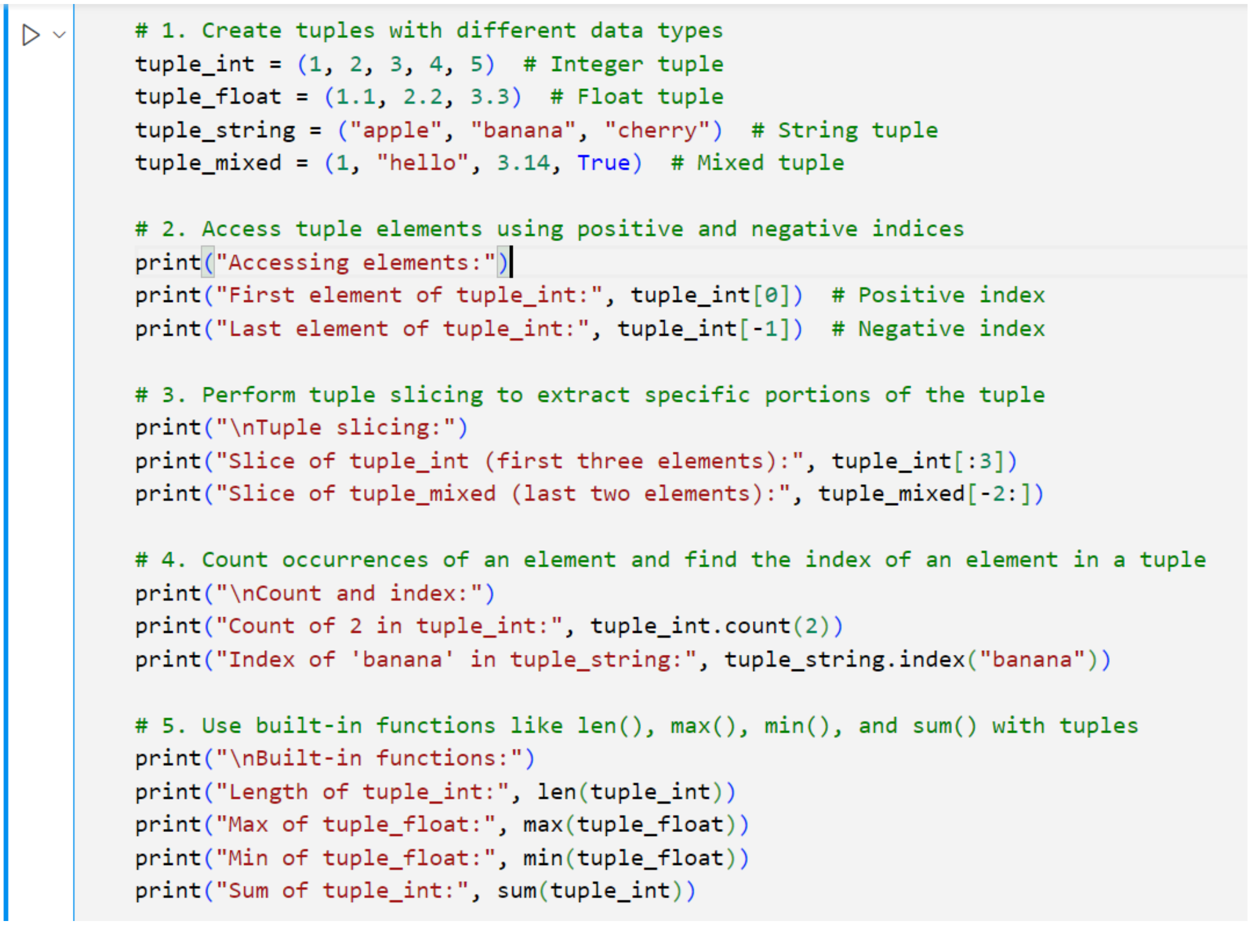
Practical 2: a) Implement the following operation using the Python tuple concept.

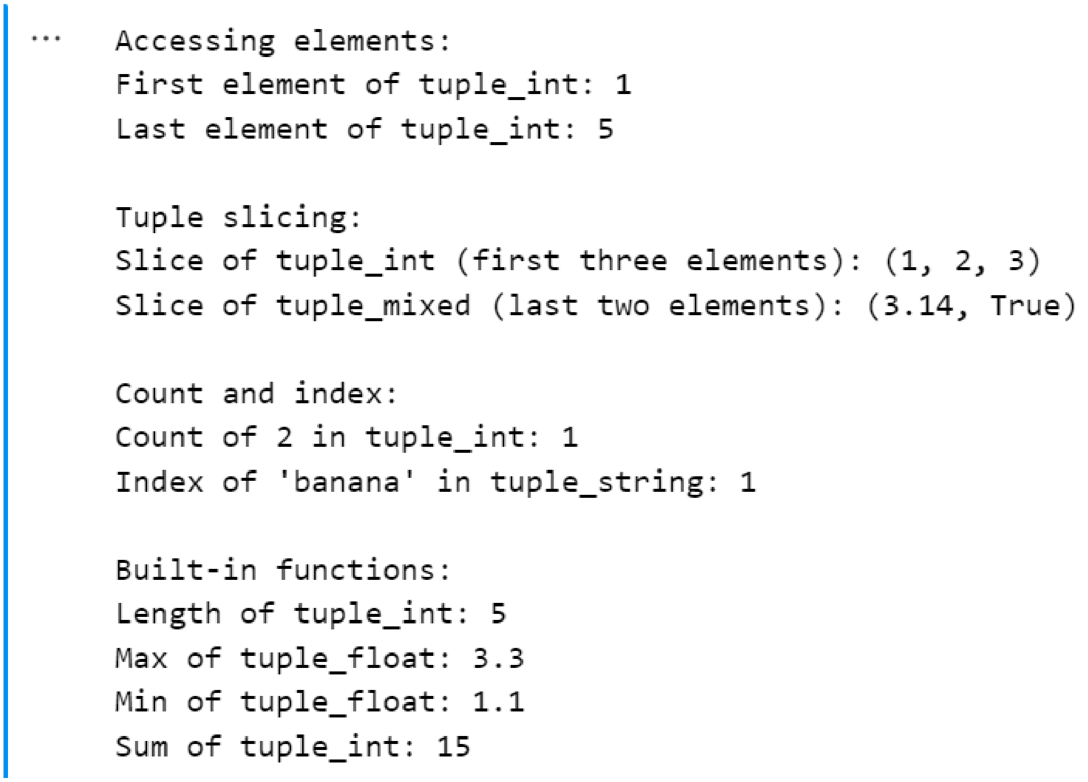
- Tuple operation

1. Create tuples with different data types (integer, float, string, and mixed).
2. Access tuple elements using positive and negative indices.
3. Perform tuple slicing to extract specific portions of the tuple.
4. Count occurrences of an element and find the index of an element in a tuple.
5. Use built-in functions like len(), max(), min(), and sum() with tuples.
6. Write a program to count and print distinct elements from a tuple.
7. Convert a list to a tuple and vice versa.
8. Demonstrate unpacking of tuples into individual variables.

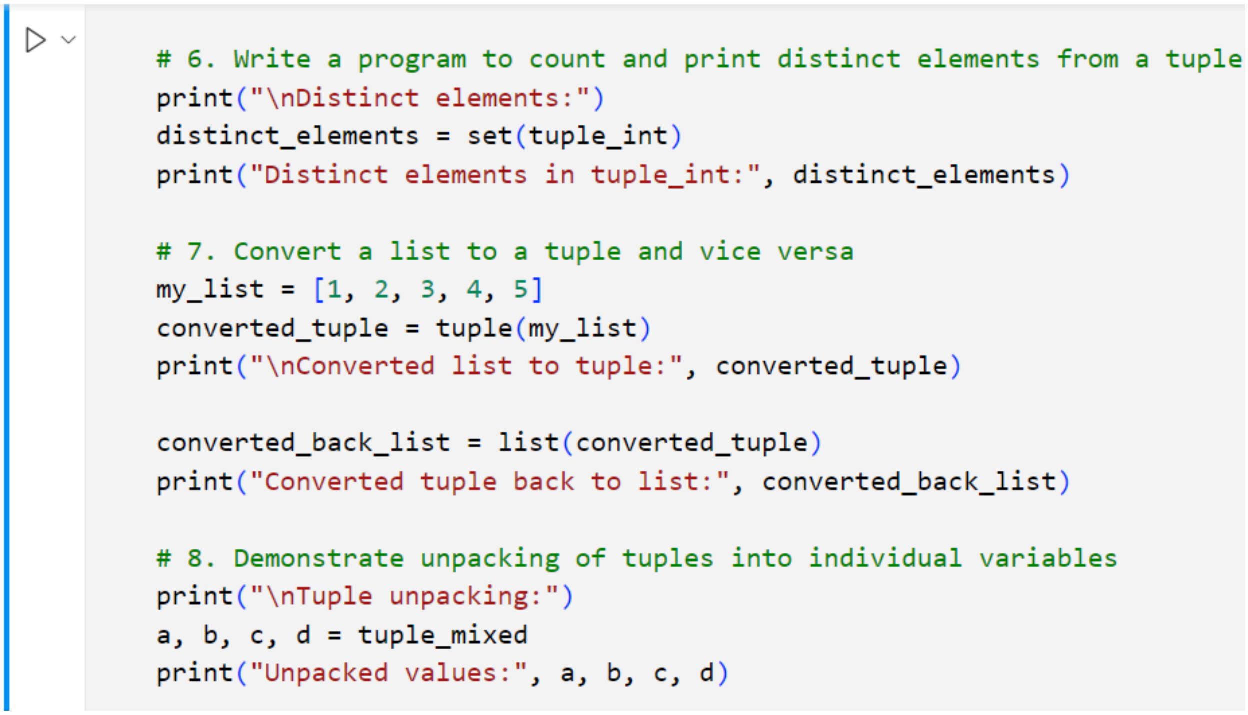


```
# 1. Create tuples with different data types
tuple_int = (1, 2, 3, 4, 5)  # Integer tuple
tuple_float = (1.1, 2.2, 3.3)  # Float tuple
tuple_string = ("apple", "banana", "cherry")  # String tuple
tuple_mixed = (1, "hello", 3.14, True)  # Mixed tuple

# 2. Access tuple elements using positive and negative indices
print("Accessing elements:")
print("First element of tuple_int:", tuple_int[0])  # Positive index
print("Last element of tuple_int:", tuple_int[-1])  # Negative index

# 3. Perform tuple slicing to extract specific portions of the tuple
print("\nTuple slicing:")
print("Slice of tuple_int (first three elements):", tuple_int[:3])
print("Slice of tuple_mixed (last two elements):", tuple_mixed[-2:])

# 4. Count occurrences of an element and find the index of an element in a tuple
print("\nCount and index:")
print("Count of 2 in tuple_int:", tuple_int.count(2))
print("Index of 'banana' in tuple_string:", tuple_string.index("banana"))

# 5. Use built-in functions like len(), max(), min(), and sum() with tuples
print("\nBuilt-in functions:")
print("Length of tuple_int:", len(tuple_int))
print("Max of tuple_float:", max(tuple_float))
print("Min of tuple_float:", min(tuple_float))
print("Sum of tuple_int:", sum(tuple_int))
```

Output:

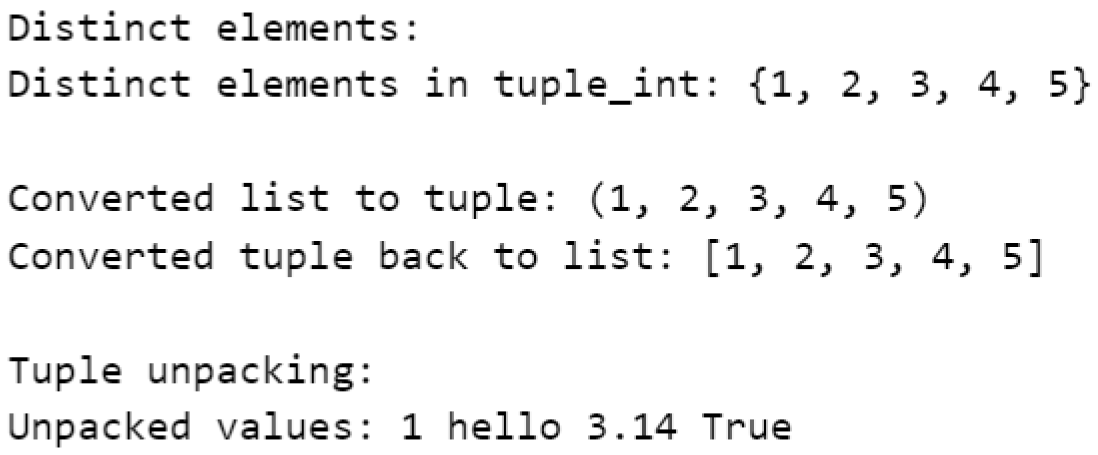


```
Accessing elements:
First element of tuple_int: 1
Last element of tuple_int: 5

Tuple slicing:
Slice of tuple_int (first three elements): (1, 2, 3)
Slice of tuple_mixed (last two elements): (3.14, True)

Count and index:
Count of 2 in tuple_int: 1
Index of 'banana' in tuple_string: 1

Built-in functions:
Length of tuple_int: 5
Max of tuple_float: 3.3
Min of tuple_float: 1.1
Sum of tuple_int: 15
```



```python
# 6. Write a program to count and print distinct elements from a tuple
print("\nDistinct elements:")
distinct_elements = set(tuple_int)
print("Distinct elements in tuple_int:", distinct_elements)

# 7. Convert a list to a tuple and vice versa
my_list = [1, 2, 3, 4, 5]
converted_tuple = tuple(my_list)
print("\nConverted list to tuple:", converted_tuple)

converted_back_list = list(converted_tuple)
print("Converted tuple back to list:", converted_back_list)

# 8. Demonstrate unpacking of tuples into individual variables
print("\nTuple unpacking:")
a, b, c, d = tuple_mixed
print("Unpacked values:", a, b, c, d)
```

Output:

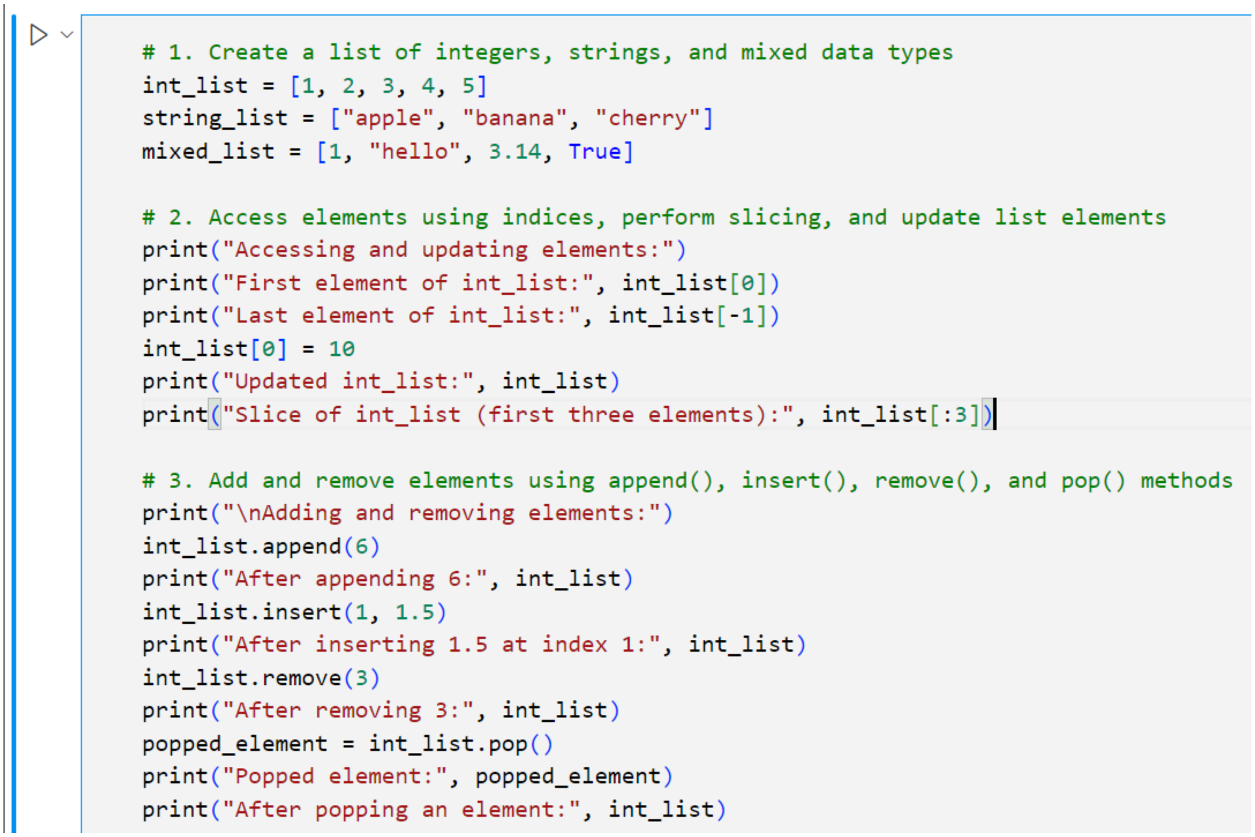


```
Distinct elements:
Distinct elements in tuple_int: {1, 2, 3, 4, 5}

Converted list to tuple: (1, 2, 3, 4, 5)
Converted tuple back to list: [1, 2, 3, 4, 5]

Tuple unpacking:
Unpacked values: 1 hello 3.14 True
```

b) Implement following operation using Python List concept.

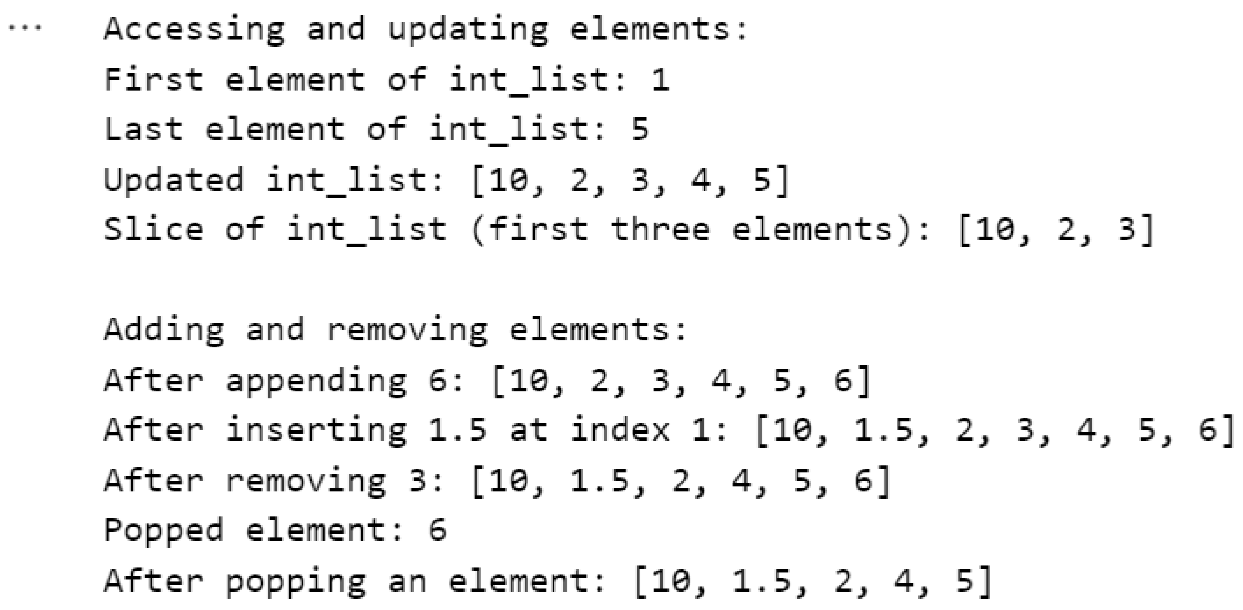
- List Operation

1. Create a list of integers, strings, and mixed data types.
2. Access elements using indices, perform slicing, and update list elements.
3. Add and remove elements using append(), insert(), remove(), and pop() methods.
4. Concatenate and repeat lists using operators.
5. Create a list of squares of the first 10 natural numbers using list comprehension.
6. Filter even numbers from a list using list comprehension.
7. Demonstrate sorting, reversing, and copying lists.
8. Write a program to remove duplicates from a list.

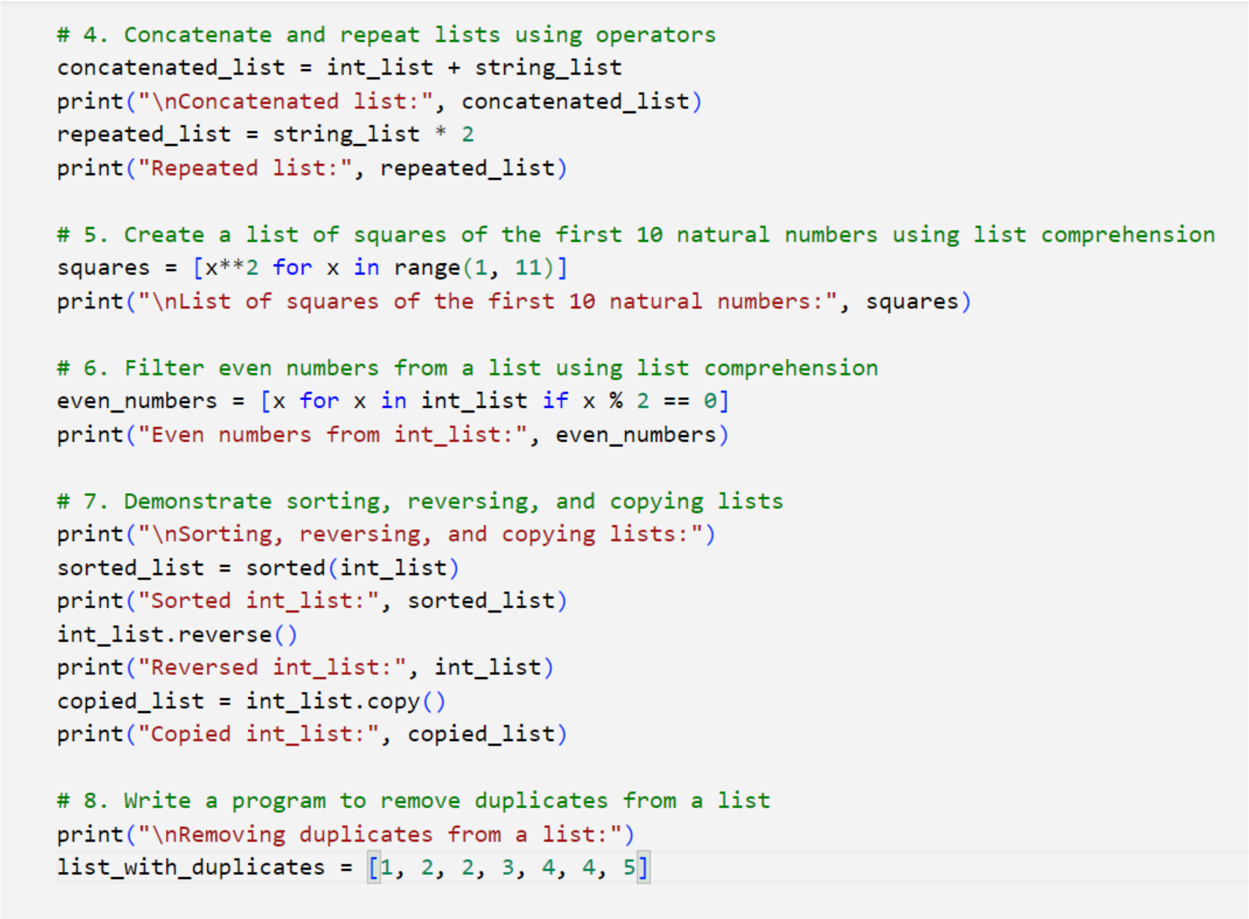


```python
# 1. Create a list of integers, strings, and mixed data types
int_list = [1, 2, 3, 4, 5]
string_list = ["apple", "banana", "cherry"]
mixed_list = [1, "hello", 3.14, True]

# 2. Access elements using indices, perform slicing, and update list elements
print("Accessing and updating elements:")
print("First element of int_list:", int_list[0])
print("Last element of int_list:", int_list[-1])
int_list[0] = 10
print("Updated int_list:", int_list)
print("Slice of int_list (first three elements):", int_list[:3])

# 3. Add and remove elements using append(), insert(), remove(), and pop() methods
print("\nAdding and removing elements:")
int_list.append(6)
print("After appending 6:", int_list)
int_list.insert(1, 1.5)
print("After inserting 1.5 at index 1:", int_list)
int_list.remove(3)
print("After removing 3:", int_list)
popped_element = int_list.pop()
print("Popped element:", popped_element)
print("After popping an element:", int_list)
```

Output:

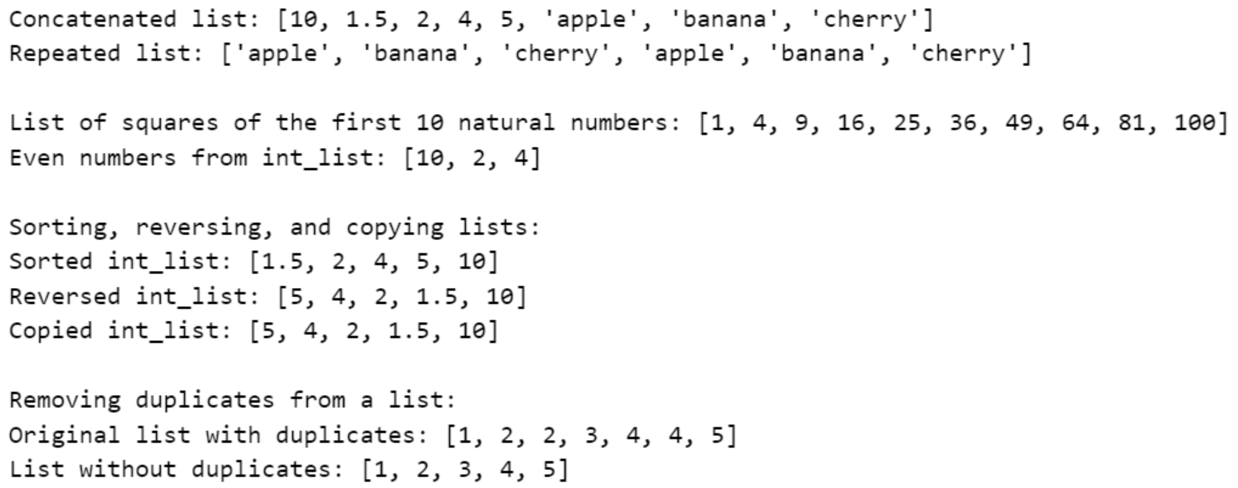


```
Accessing and updating elements:
First element of int_list: 1
Last element of int_list: 5
Updated int_list: [10, 2, 3, 4, 5]
Slice of int_list (first three elements): [10, 2, 3]

Adding and removing elements:
After appending 6: [10, 2, 3, 4, 5, 6]
After inserting 1.5 at index 1: [10, 1.5, 2, 3, 4, 5, 6]
After removing 3: [10, 1.5, 2, 4, 5, 6]
Popped element: 6
After popping an element: [10, 1.5, 2, 4, 5]
```



```python
# 4. Concatenate and repeat lists using operators
concatenated_list = int_list + string_list
print("\nConcatenated list:", concatenated_list)
repeated_list = string_list * 2
print("Repeated list:", repeated_list)

# 5. Create a list of squares of the first 10 natural numbers using list comprehension
squares = [x**2 for x in range(1, 11)]
print("\nList of squares of the first 10 natural numbers:", squares)

# 6. Filter even numbers from a list using list comprehension
even_numbers = [x for x in int_list if x % 2 == 0]
print("Even numbers from int_list:", even_numbers)

# 7. Demonstrate sorting, reversing, and copying lists
print("\nSorting, reversing, and copying lists:")
sorted_list = sorted(int_list)
print("Sorted int_list:", sorted_list)
int_list.reverse()
print("Reversed int_list:", int_list)
copied_list = int_list.copy()
print("Copied int_list:", copied_list)

# 8. Write a program to remove duplicates from a list
print("\nRemoving duplicates from a list:")
list_with_duplicates = [1, 2, 2, 3, 4, 4, 5]
```

Output:

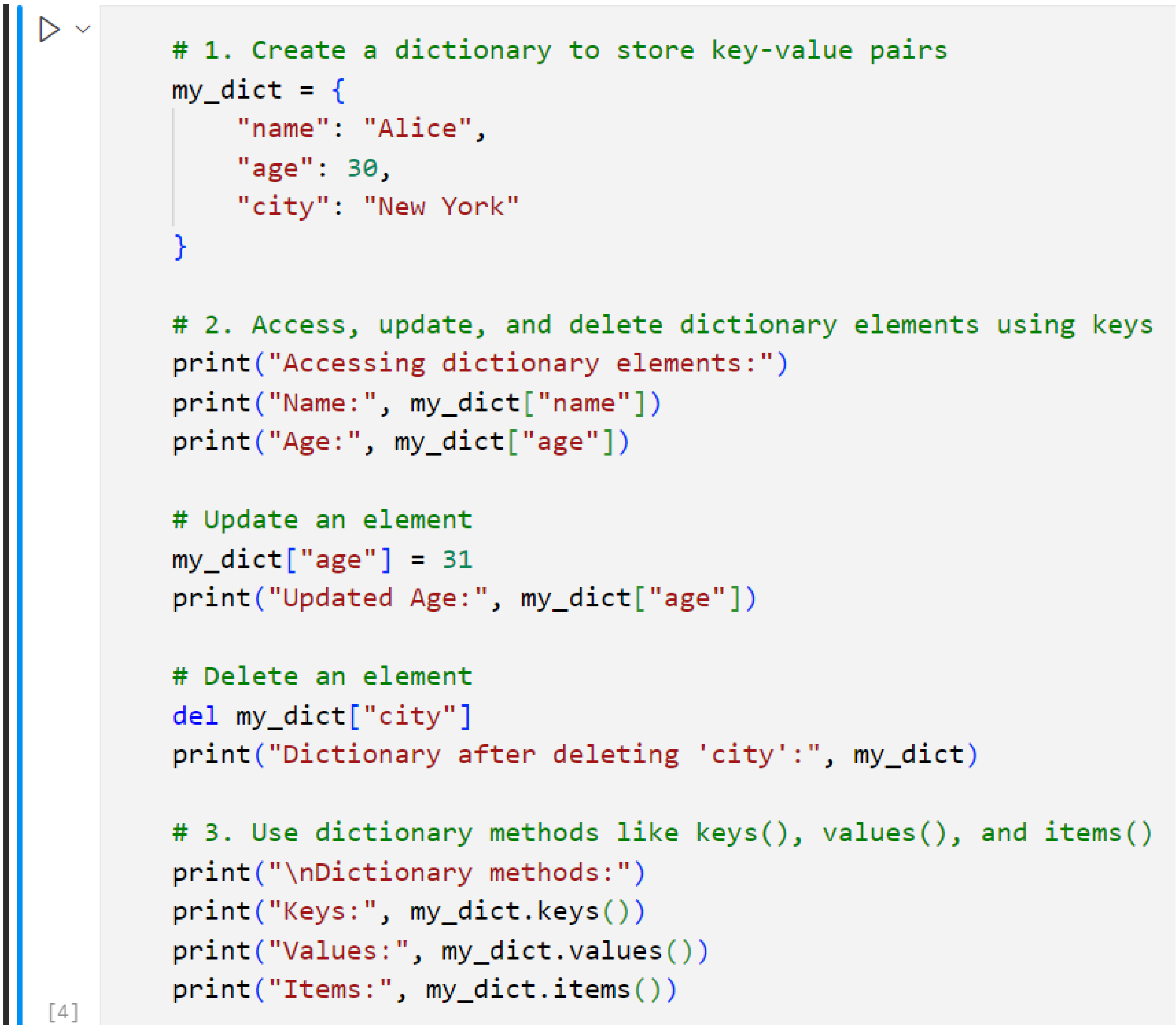


```
Concatenated list: [10, 1.5, 2, 4, 5, 'apple', 'banana', 'cherry']
Repeated list: ['apple', 'banana', 'cherry', 'apple', 'banana', 'cherry']

List of squares of the first 10 natural numbers: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Even numbers from int_list: [10, 2, 4]

Sorting, reversing, and copying lists:
Sorted int_list: [1.5, 2, 4, 5, 10]
Reversed int_list: [5, 4, 2, 1.5, 10]
Copied int_list: [5, 4, 2, 1.5, 10]

Removing duplicates from a list:
Original list with duplicates: [1, 2, 2, 3, 4, 4, 5]
List without duplicates: [1, 2, 3, 4, 5]
```

c) Implementing following operation using python dictionaries concept.

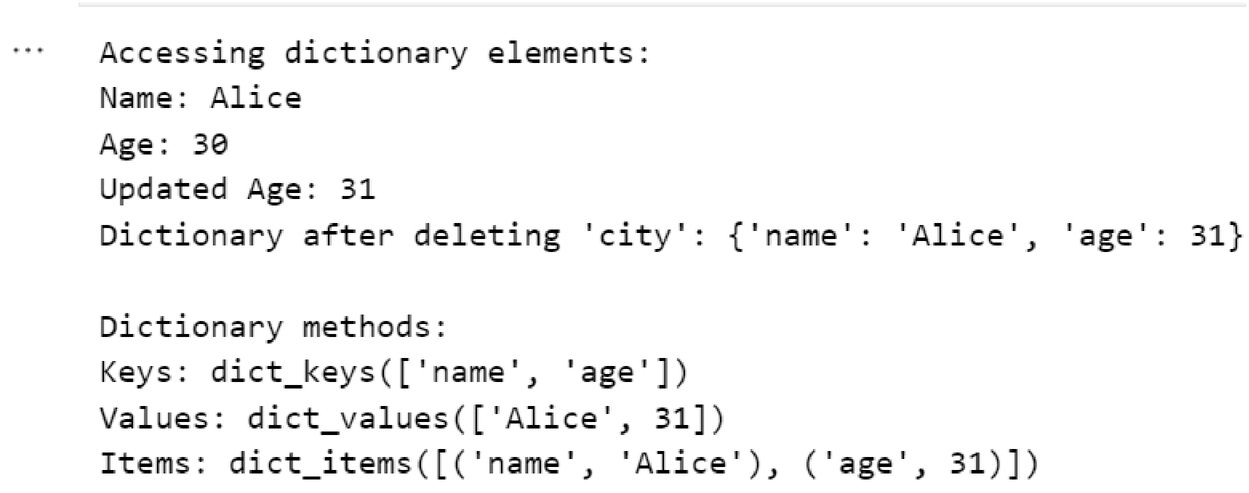
- Dictionary Operation:

1. Create a dictionary to store key-value pairs.
2. Access, update, and delete dictionary elements using keys.
3. Use dictionary methods like keys(), values(), and items().
4. Add a new key-value pair and remove an existing key-value pair.
5. Create a nested dictionary to store student details (like name, age, and marks).
6. Access and update elements in a nested dictionary.
7. Merge two dictionaries using update().
8. Write a program to sort a dictionary based on its values.

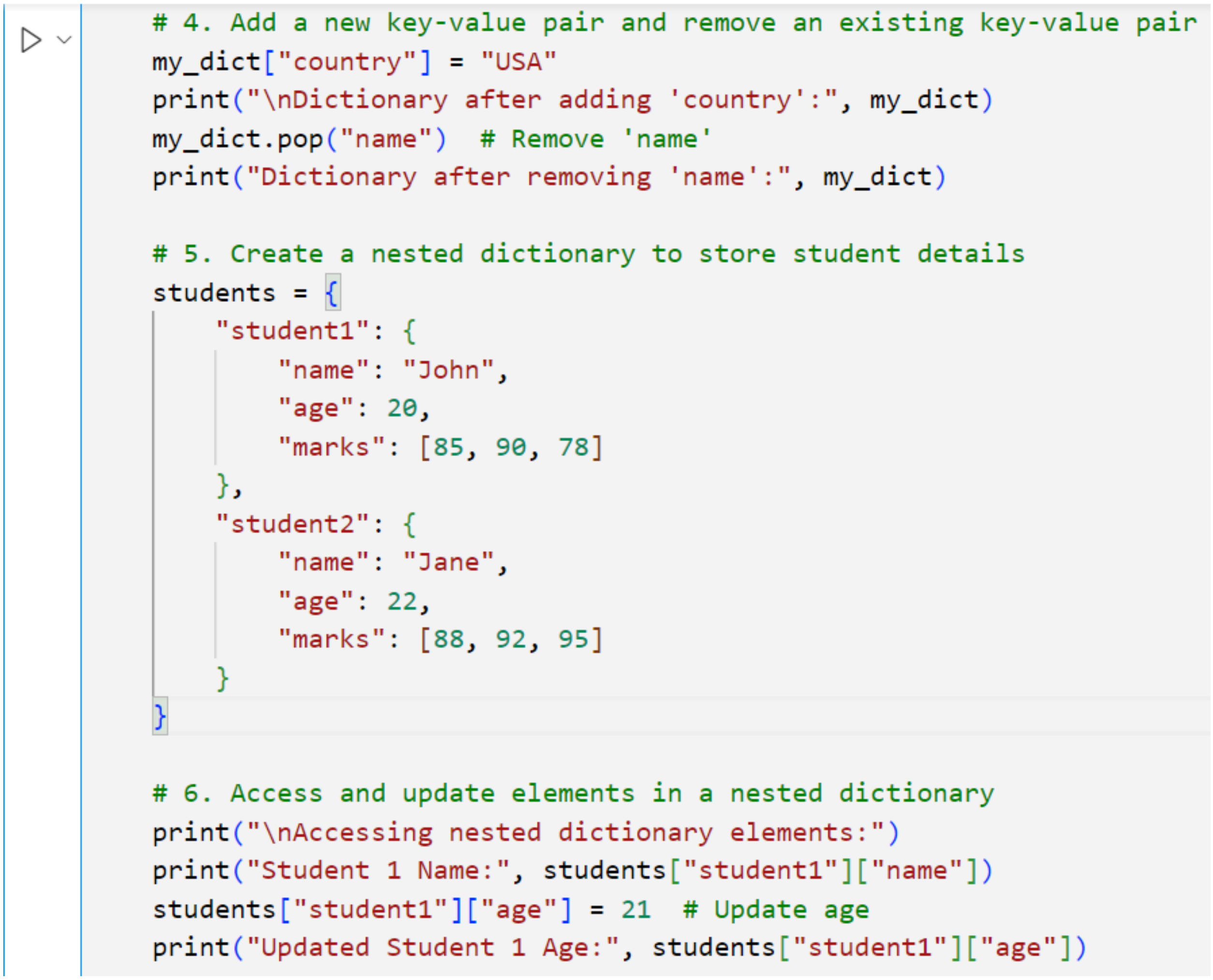


```
# 1. Create a dictionary to store key-value pairs
my_dict = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

# 2. Access, update, and delete dictionary elements using keys
print("Accessing dictionary elements:")
print("Name:", my_dict["name"])
print("Age:", my_dict["age"])

# Update an element
my_dict["age"] = 31
print("Updated Age:", my_dict["age"])

# Delete an element
del my_dict["city"]
print("Dictionary after deleting 'city':", my_dict)

# 3. Use dictionary methods like keys(), values(), and items()
print("\nDictionary methods:")
print("Keys:", my_dict.keys())
print("Values:", my_dict.values())
print("Items:", my_dict.items())
```

Output:

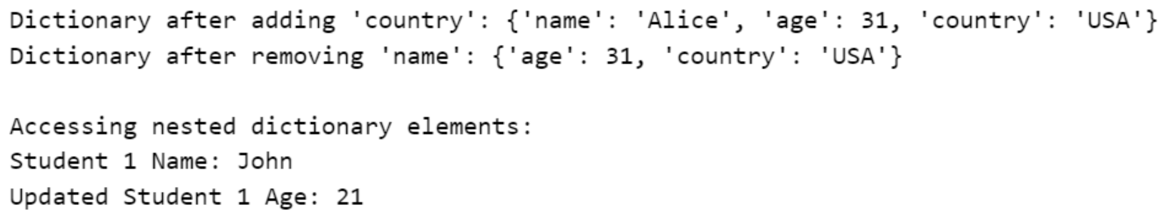


```
Accessing dictionary elements:
Name: Alice
Age: 30
Updated Age: 31
Dictionary after deleting 'city': {'name': 'Alice', 'age': 31}

Dictionary methods:
Keys: dict_keys(['name', 'age'])
Values: dict_values(['Alice', 31])
Items: dict_items([('name', 'Alice'), ('age', 31)])
```

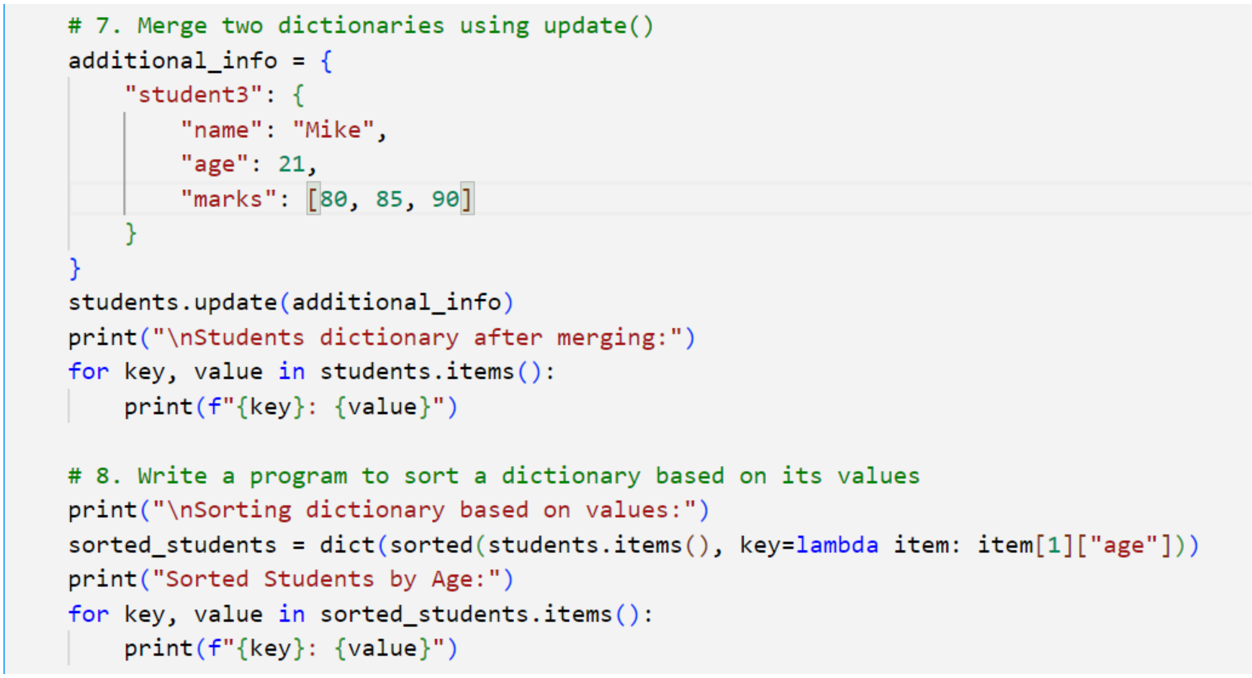


```python
# 4. Add a new key-value pair and remove an existing key-value pair
my_dict["country"] = "USA"
print("\nDictionary after adding 'country':", my_dict)
my_dict.pop("name")  # Remove 'name'
print("Dictionary after removing 'name':", my_dict)

# 5. Create a nested dictionary to store student details
students = {
    "student1": {
        "name": "John",
        "age": 20,
        "marks": [85, 90, 78]
    },
    "student2": {
        "name": "Jane",
        "age": 22,
        "marks": [88, 92, 95]
    }
}

# 6. Access and update elements in a nested dictionary
print("\nAccessing nested dictionary elements:")
print("Student 1 Name:", students["student1"]["name"])
students["student1"]["age"] = 21  # Update age
print("Updated Student 1 Age:", students["student1"]["age"])
```

Output:

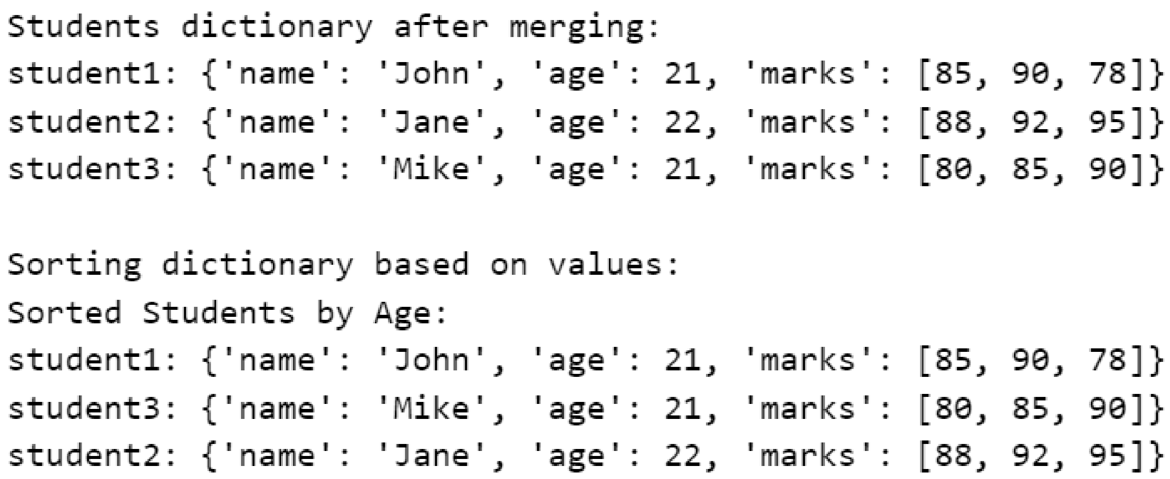


```
Dictionary after adding 'country': {'name': 'Alice', 'age': 31, 'country': 'USA'}
Dictionary after removing 'name': {'age': 31, 'country': 'USA'}

Accessing nested dictionary elements:
Student 1 Name: John
Updated Student 1 Age: 21
```



```python
# 7. Merge two dictionaries using update()
additional_info = {
    "student3": {
        "name": "Mike",
        "age": 21,
        "marks": [80, 85, 90]
    }
}
students.update(additional_info)
print("\nStudents dictionary after merging:")
for key, value in students.items():
    print(f"{key}: {value}")

# 8. Write a program to sort a dictionary based on its values
print("\nSorting dictionary based on values:")
sorted_students = dict(sorted(students.items(), key=lambda item: item[1]["age"]))
print("Sorted Students by Age:")
for key, value in sorted_students.items():
    print(f"{key}: {value}")
```

Output:

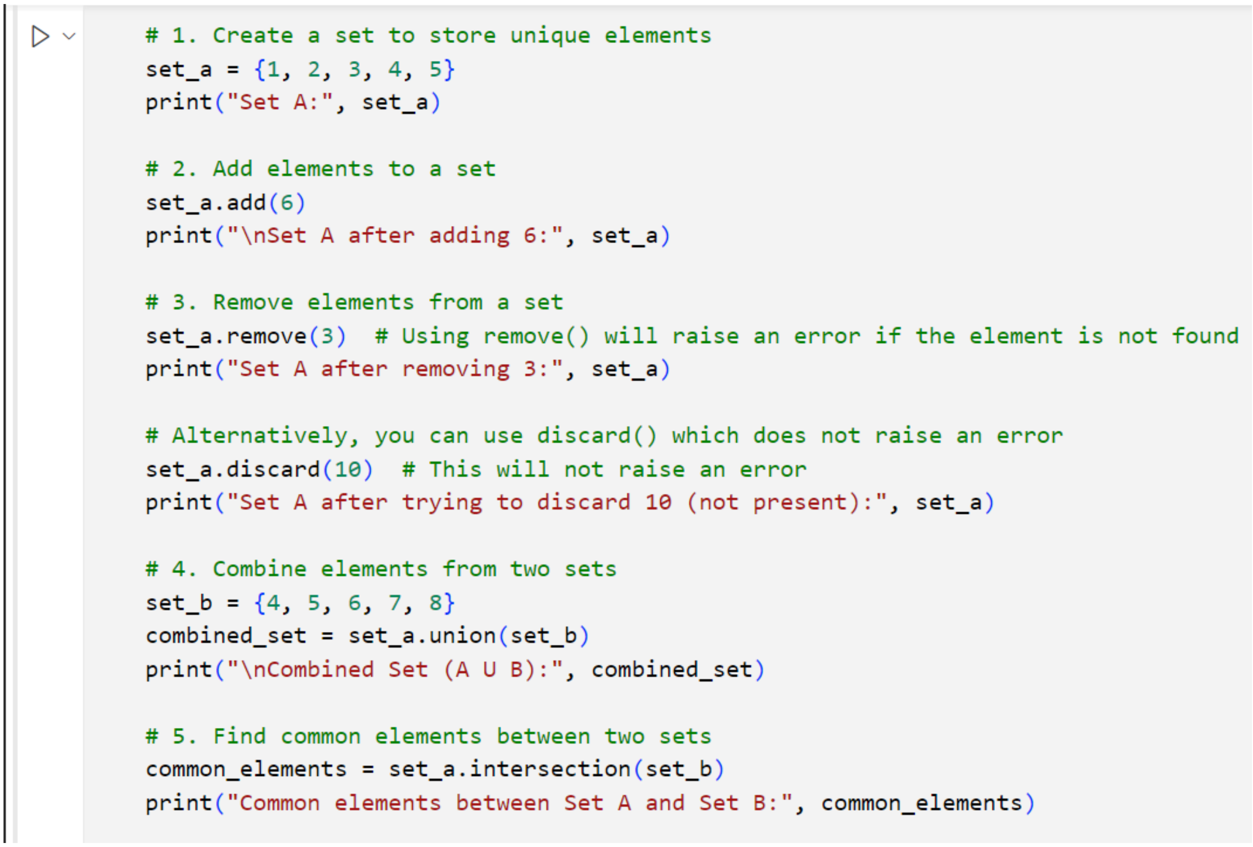


```
Students dictionary after merging:
student1: {'name': 'John', 'age': 21, 'marks': [85, 90, 78]}
student2: {'name': 'Jane', 'age': 22, 'marks': [88, 92, 95]}
student3: {'name': 'Mike', 'age': 21, 'marks': [80, 85, 90]}

Sorting dictionary based on values:
Sorted Students by Age:
student1: {'name': 'John', 'age': 21, 'marks': [85, 90, 78]}
student3: {'name': 'Mike', 'age': 21, 'marks': [80, 85, 90]}
student2: {'name': 'Jane', 'age': 22, 'marks': [88, 92, 95]}
```

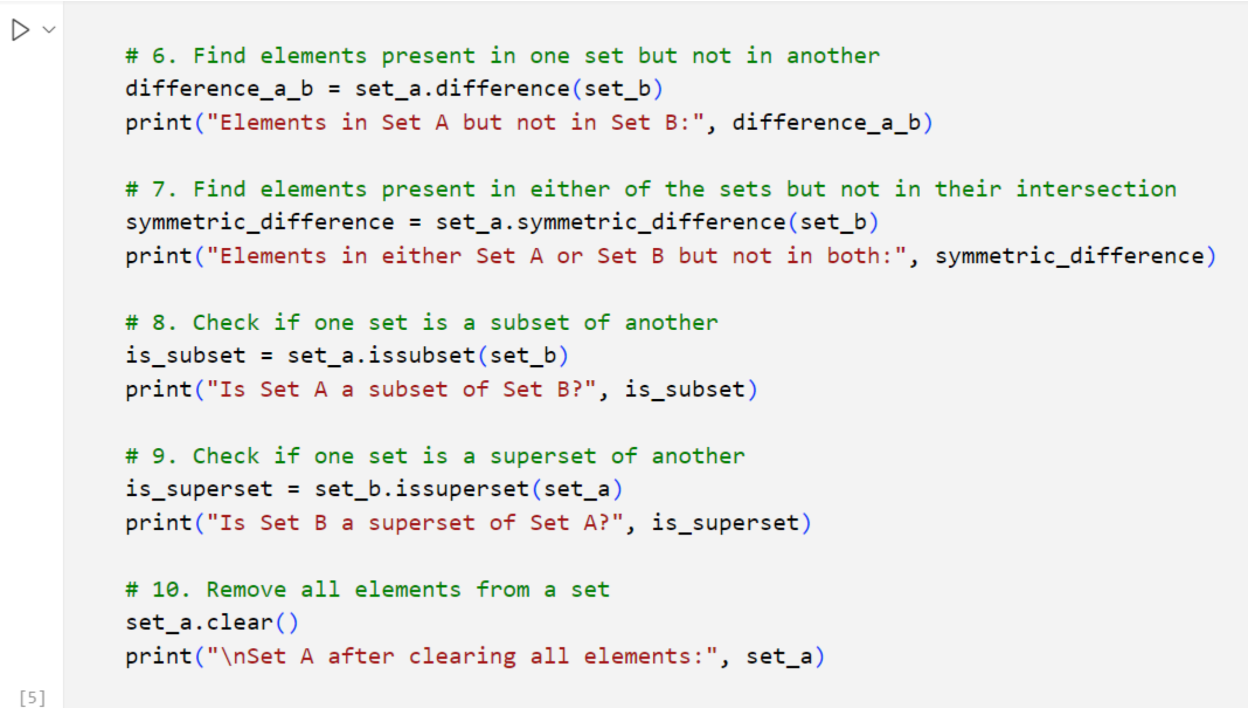
d) Implementing following operation using python set concept.

- Set Operation:

1. Create a set to store unique elements.
2. Add elements to a set.
3. Remove elements from a set.
4. Combine elements from two sets.
5. Find common elements between two sets.
6. Find elements present in one set but not in another.
7. Find elements present in either of the sets but not in their intersection.
8. Check if one set is a subset of another.
9. Check if one set is a superset of another.
10. Remove all elements from a set.

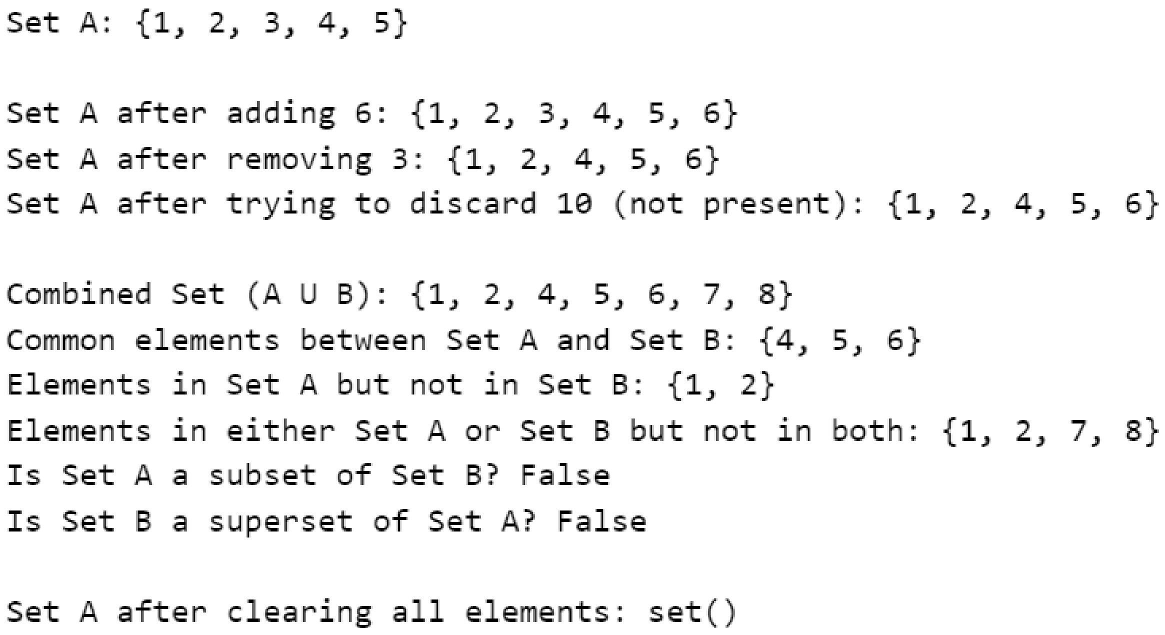


```python
# 1. Create a set to store unique elements
set_a = {1, 2, 3, 4, 5}
print("Set A:", set_a)

# 2. Add elements to a set
set_a.add(6)
print("\nSet A after adding 6:", set_a)

# 3. Remove elements from a set
set_a.remove(3)  # Using remove() will raise an error if the element is not found
print("Set A after removing 3:", set_a)

# Alternatively, you can use discard() which does not raise an error
set_a.discard(10)  # This will not raise an error
print("Set A after trying to discard 10 (not present):", set_a)

# 4. Combine elements from two sets
set_b = {4, 5, 6, 7, 8}
combined_set = set_a.union(set_b)
print("\nCombined Set (A U B):", combined_set)

# 5. Find common elements between two sets
common_elements = set_a.intersection(set_b)
print("Common elements between Set A and Set B:", common_elements)
```



```
# 6. Find elements present in one set but not in another
difference_a_b = set_a.difference(set_b)
print("Elements in Set A but not in Set B:", difference_a_b)

# 7. Find elements present in either of the sets but not in their intersection
symmetric_difference = set_a.symmetric_difference(set_b)
print("Elements in either Set A or Set B but not in both:", symmetric_difference)

# 8. Check if one set is a subset of another
is_subset = set_a.issubset(set_b)
print("Is Set A a subset of Set B?", is_subset)

# 9. Check if one set is a superset of another
is_superset = set_b.issuperset(set_a)
print("Is Set B a superset of Set A?", is_superset)

# 10. Remove all elements from a set
set_a.clear()
print("\nSet A after clearing all elements:", set_a)
```

Output:

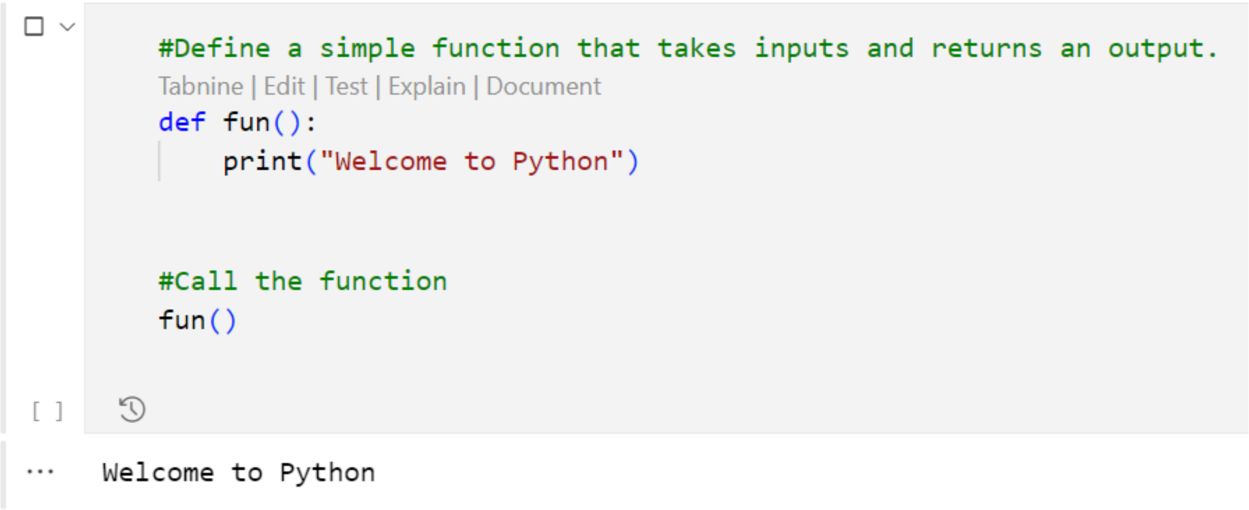


```
Set A: {1, 2, 3, 4, 5}

Set A after adding 6: {1, 2, 3, 4, 5, 6}
Set A after removing 3: {1, 2, 4, 5, 6}
Set A after trying to discard 10 (not present): {1, 2, 4, 5, 6}

Combined Set (A U B): {1, 2, 4, 5, 6, 7, 8}
Common elements between Set A and Set B: {4, 5, 6}
Elements in Set A but not in Set B: {1, 2}
Elements in either Set A or Set B but not in both: {1, 2, 7, 8}
Is Set A a subset of Set B? False
Is Set B a superset of Set A? False

Set A after clearing all elements: set()
```

Practical 3: Function and loop in python.

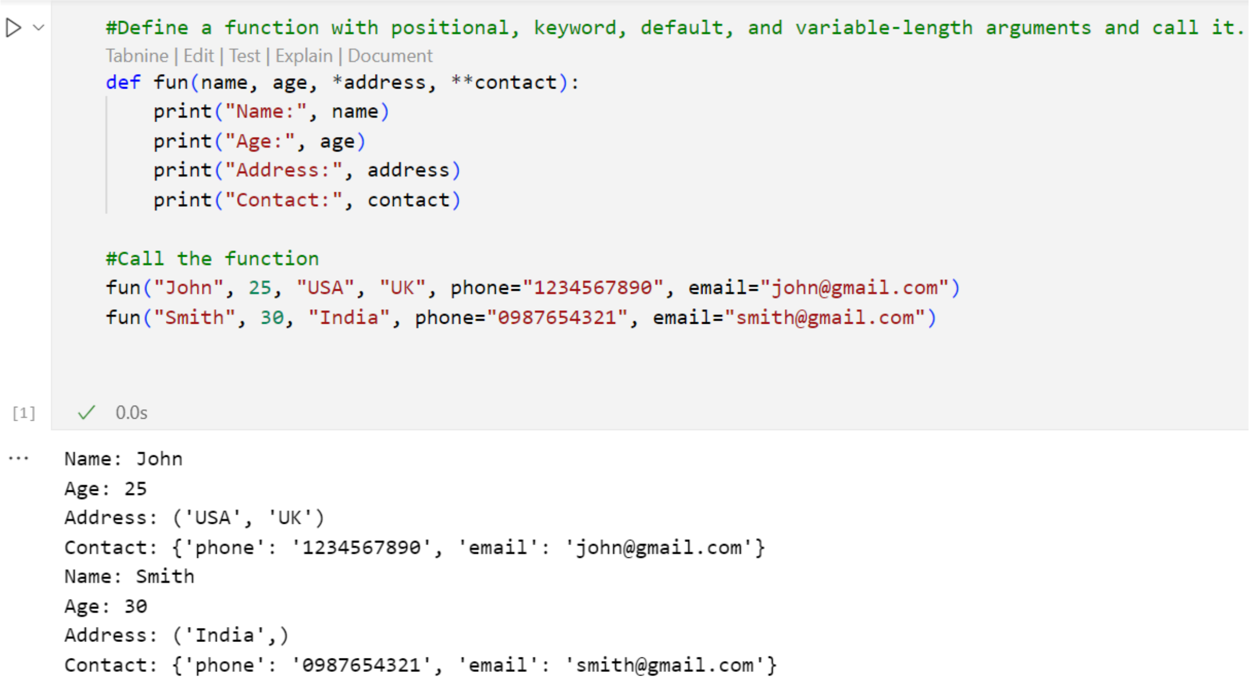
- Basics of function

1. Define a simple function that takes inputs and returns an output.
2. Define a function with positional, keyword, default, and variable- length arguments
3. Define a function that returns the multiple values using tuple.
4. Define an anonymous function using lamba keyword
5. Define a function inside another function.
6. Create and use decorators to modify the behavior of functions.
7. Define a function that calls itself to solve a problem recursively.
8. Define functions that take other functions as arguments or return functions as results.
9. Add docstrings to functions to document their purpose and usage.
10. Use type annotations to specify the expected types of function arguments and return values.



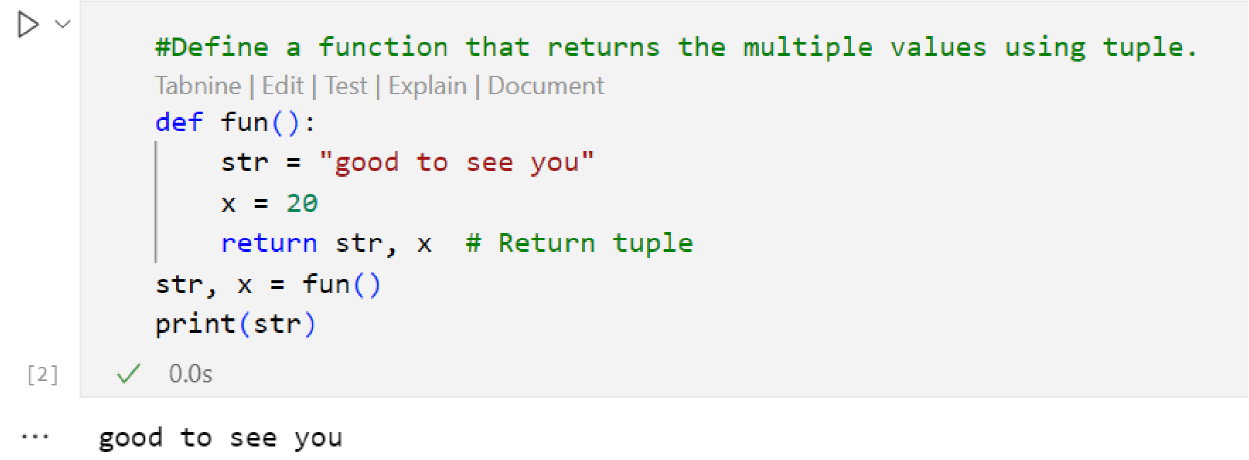
```
#Define a simple function that takes inputs and returns an output.
def fun():
    print("Welcome to Python")


#Call the function
fun()
```

```
Welcome to Python
```



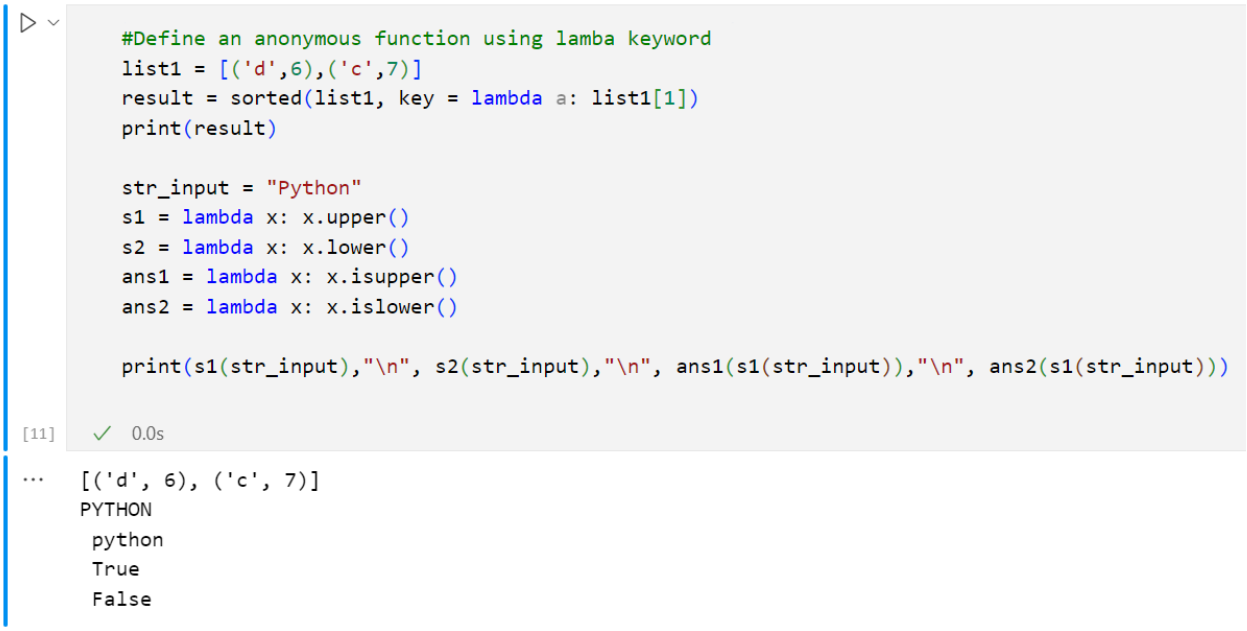
```python
#Define a function with positional, keyword, default, and variable-length arguments and call it.
def fun(name, age, *address, **contact):
    print("Name:", name)
    print("Age:", age)
    print("Address:", address)
    print("Contact:", contact)

#Call the function
fun("John", 25, "USA", "UK", phone="1234567890", email="john@gmail.com")
fun("Smith", 30, "India", phone="0987654321", email="smith@gmail.com")
```

```
Name: John
Age: 25
Address: ('USA', 'UK')
Contact: {'phone': '1234567890', 'email': 'john@gmail.com'}
Name: Smith
Age: 30
Address: ('India',)
Contact: {'phone': '0987654321', 'email': 'smith@gmail.com'}
```



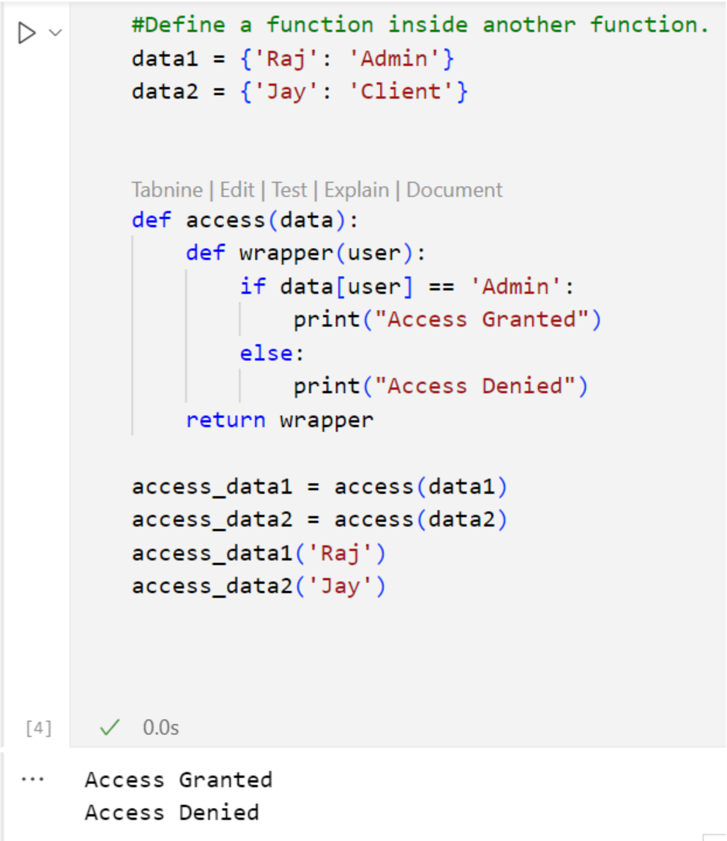
```python
#Define a function that returns the multiple values using tuple.
def fun():
    str = "good to see you"
    x = 20
    return str, x  # Return tuple
str, x = fun()
print(str)
```

```
good to see you
```



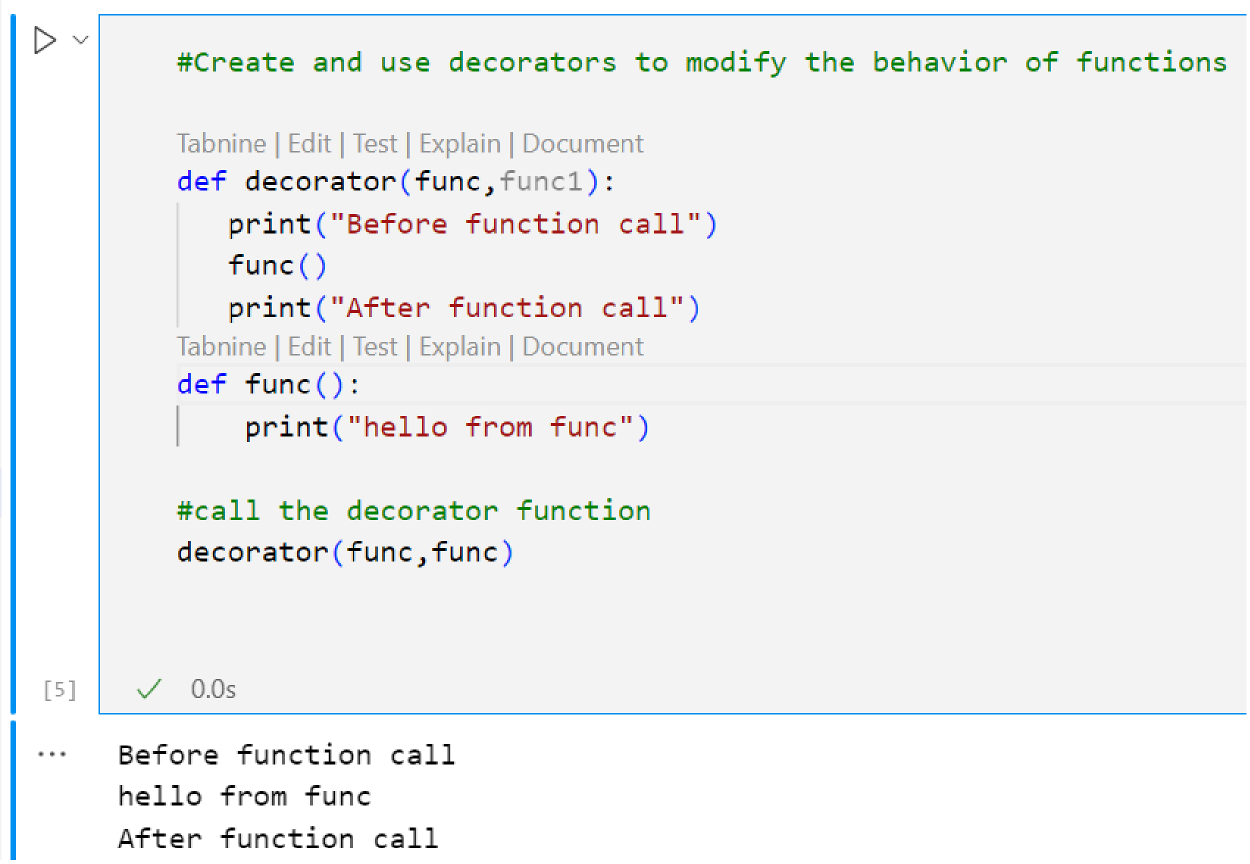
```python
#Define an anonymous function using lamba keyword
list1 = [('d',6),('c',7)]
result = sorted(list1, key = lambda a: list1[1])
print(result)

str_input = "Python"
s1 = lambda x: x.upper()
s2 = lambda x: x.lower()
ans1 = lambda x: x.isupper()
ans2 = lambda x: x.islower()

print(s1(str_input),"\n", s2(str_input),"\n", ans1(s1(str_input)),"\n", ans2(s1(str_input)))
```

```
[('d', 6), ('c', 7)]
PYTHON
 python
 True
 False
```



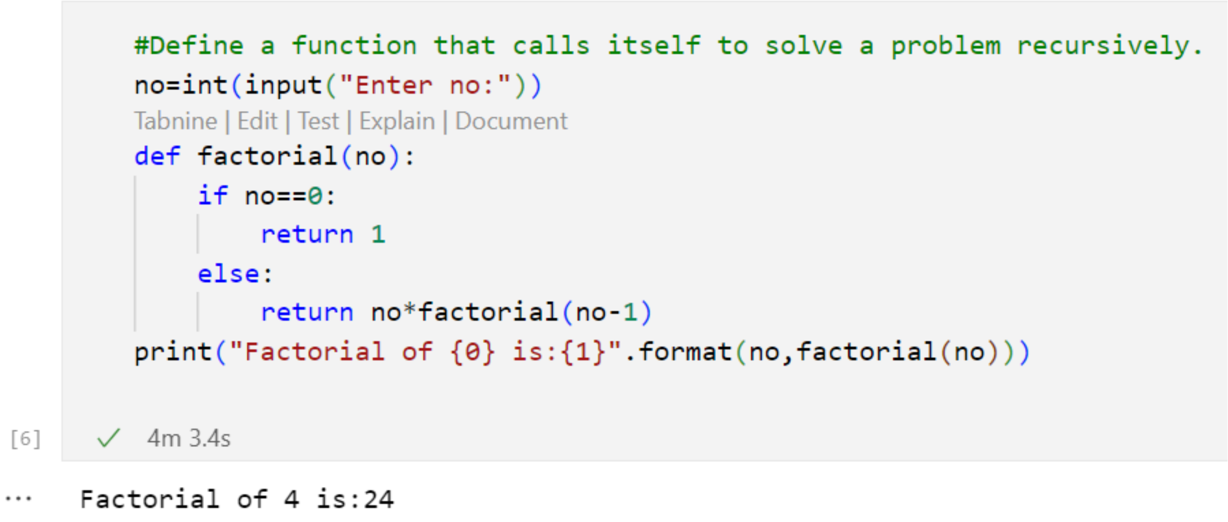
```python
#Define a function inside another function.
data1 = {'Raj': 'Admin'}
data2 = {'Jay': 'Client'}


def access(data):
    def wrapper(user):
        if data[user] == 'Admin':
            print("Access Granted")
        else:
            print("Access Denied")
    return wrapper

access_data1 = access(data1)
access_data2 = access(data2)
access_data1('Raj')
access_data2('Jay')
```

```
Access Granted
Access Denied
```



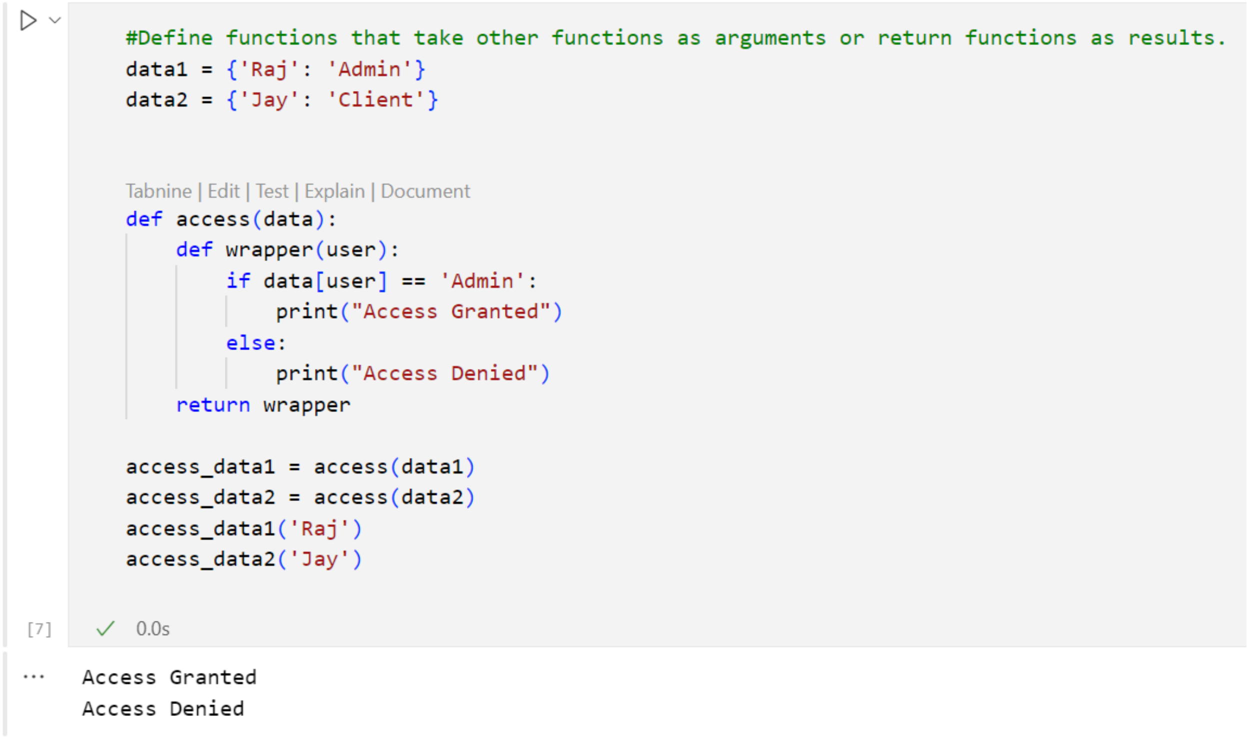
```python
#Create and use decorators to modify the behavior of functions

def decorator(func,func1):
    print("Before function call")
    func()
    print("After function call")
def func():
     print("hello from func")

#call the decorator function
decorator(func,func)
```

```
Before function call
hello from func
After function call
```



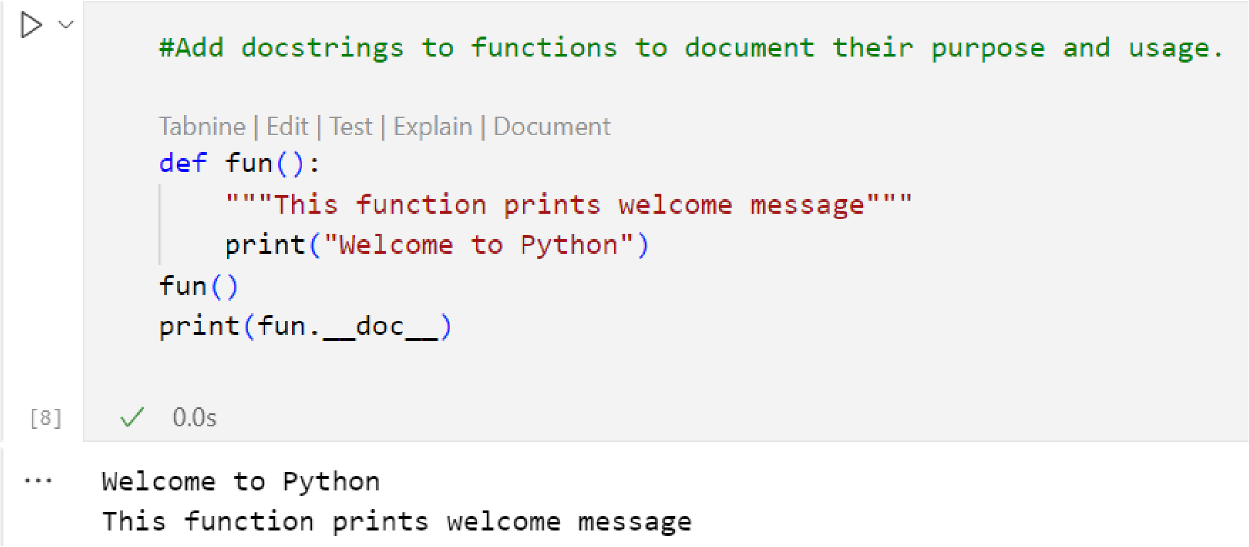
```python
#Define a function that calls itself to solve a problem recursively.
no=int(input("Enter no:"))
def factorial(no):
    if no==0:
        return 1
    else:
        return no*factorial(no-1)
print("Factorial of {0} is:{1}".format(no,factorial(no)))
```

```
Factorial of 4 is:24
```



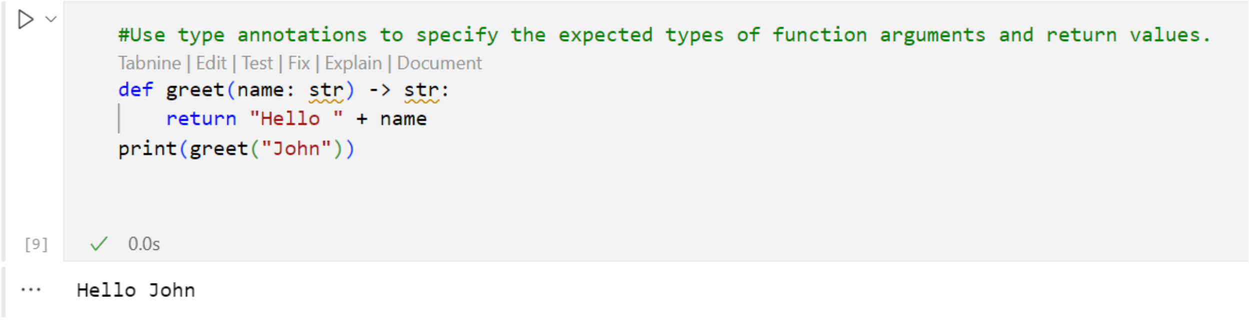
```python
#Define functions that take other functions as arguments or return functions as results.
data1 = {'Raj': 'Admin'}
data2 = {'Jay': 'Client'}


def access(data):
    def wrapper(user):
        if data[user] == 'Admin':
            print("Access Granted")
        else:
            print("Access Denied")
    return wrapper

access_data1 = access(data1)
access_data2 = access(data2)
access_data1('Raj')
access_data2('Jay')
```

```
Access Granted
Access Denied
```



```python
#Add docstrings to functions to document their purpose and usage.

def fun():
    """This function prints welcome message"""
    print("Welcome to Python")
fun()
print(fun.__doc__)
```

```
Welcome to Python
This function prints welcome message
```

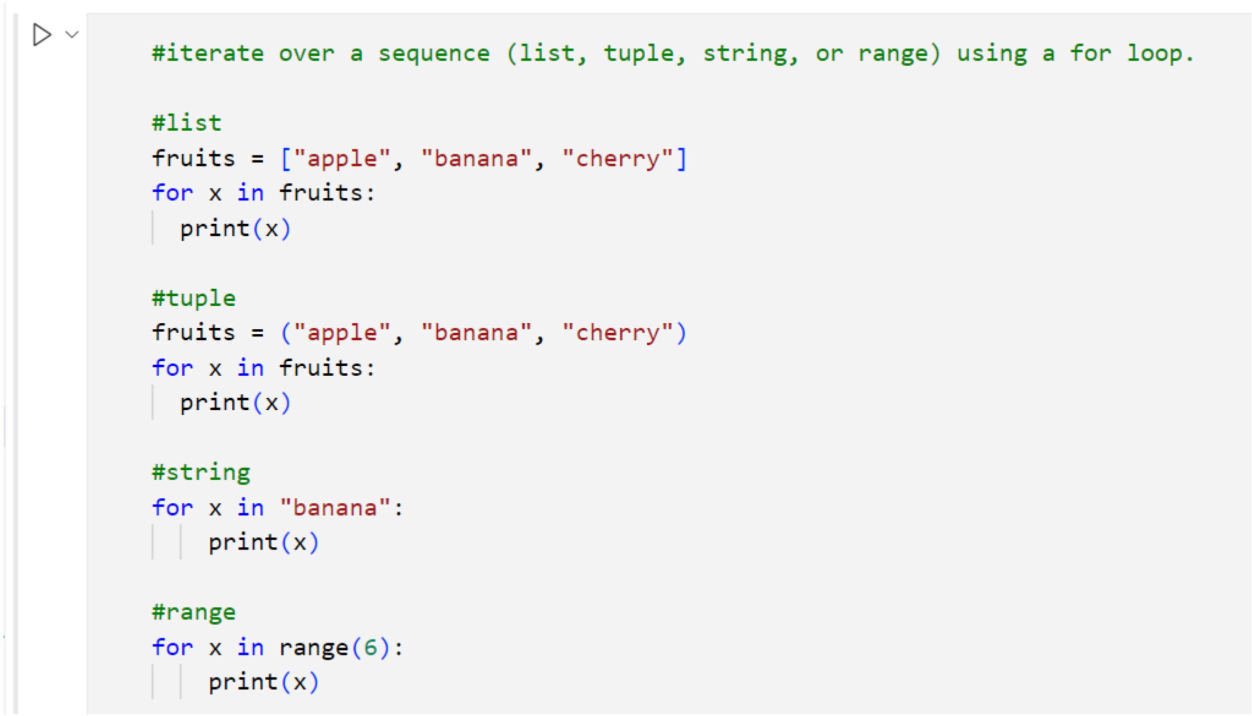


```python
#Use type annotations to specify the expected types of function arguments and return values.
def greet(name: str) -> str:
    return "Hello " + name
print(greet("John"))
```

```
Hello John
```

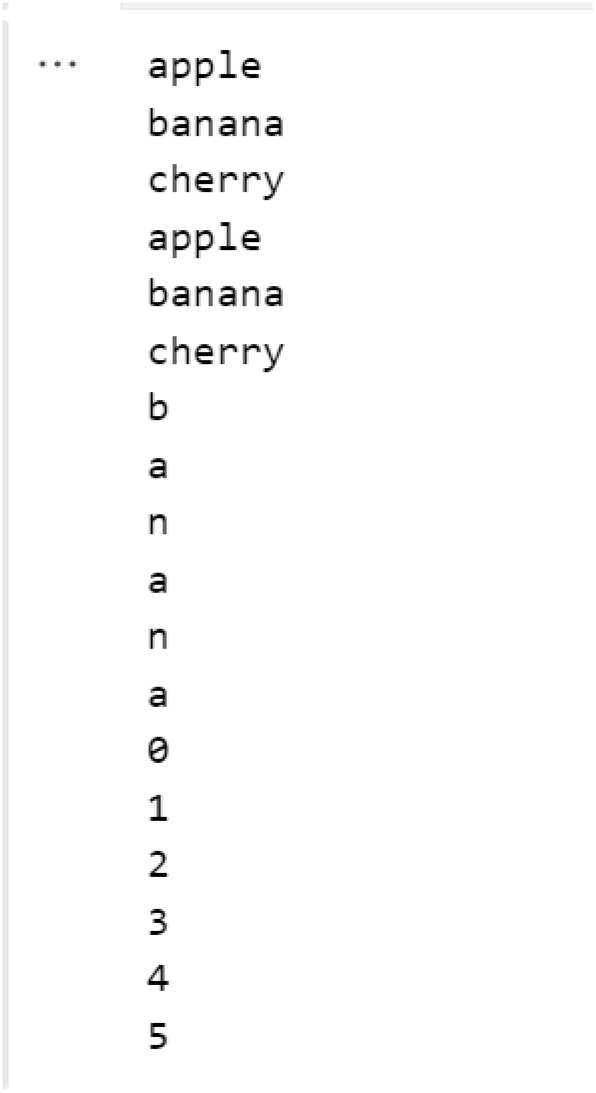
- Basics of loops

1. Iterate over a sequence (list, tuple, string, or range) using a for loop.
2. Repeat a block of code as long as a condition is true using a while loop.
3. Use loops inside other loops to handle multi-dimensional data structures.
4. Use break, continue, and pass to control the flow of loops.
5. Use the enumerate function to get both the index and value while iterating over a sequence.
6. Use the range function to generate a sequence of numbers for iteration.
7. Iterate over the key-value pairs of a dictionary using a for loop.
8. Use list comprehensions to create new lists by applying an expression to each item in an existing list.

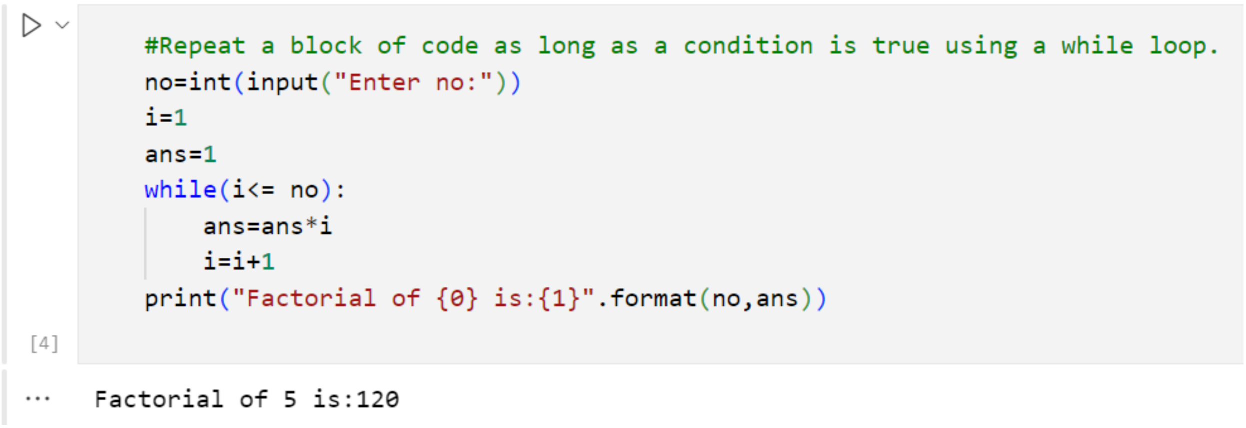


```
#iterate over a sequence (list, tuple, string, or range) using a for loop.

#list
fruits = ["apple", "banana", "cherry"]
for x in fruits:
 print(x)

#tuple
fruits = ("apple", "banana", "cherry")
for x in fruits:
 print(x)

#string
for x in "banana":
  print(x)

#range
for x in range(6):
  print(x)
```

Output:

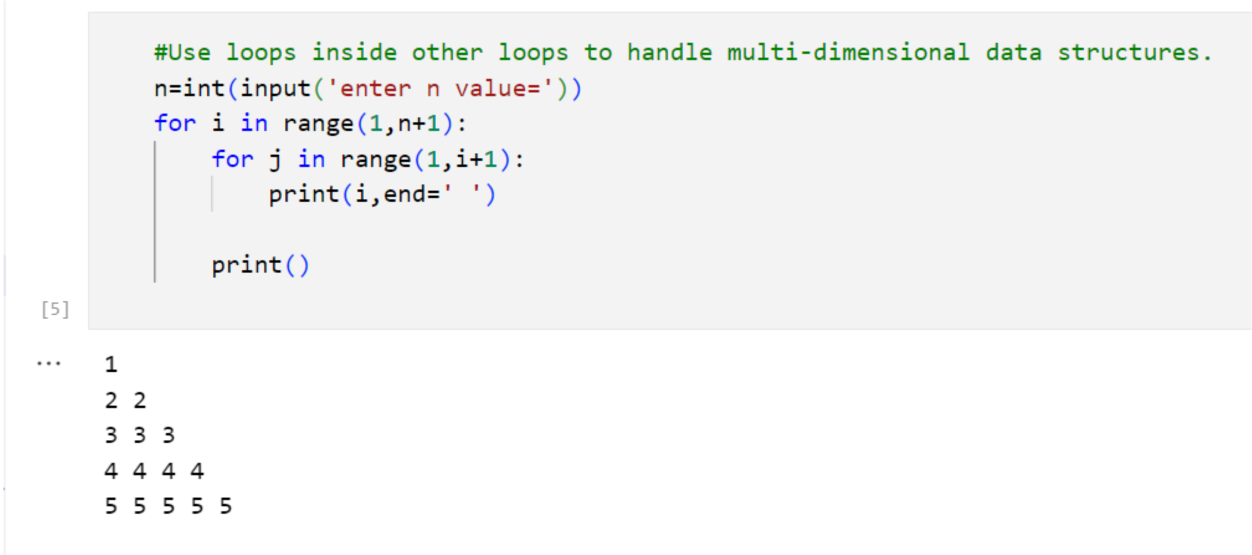


```
apple
banana
cherry
apple
banana
cherry
b
a
n
a
n
a
0
1
2
3
4
5
```



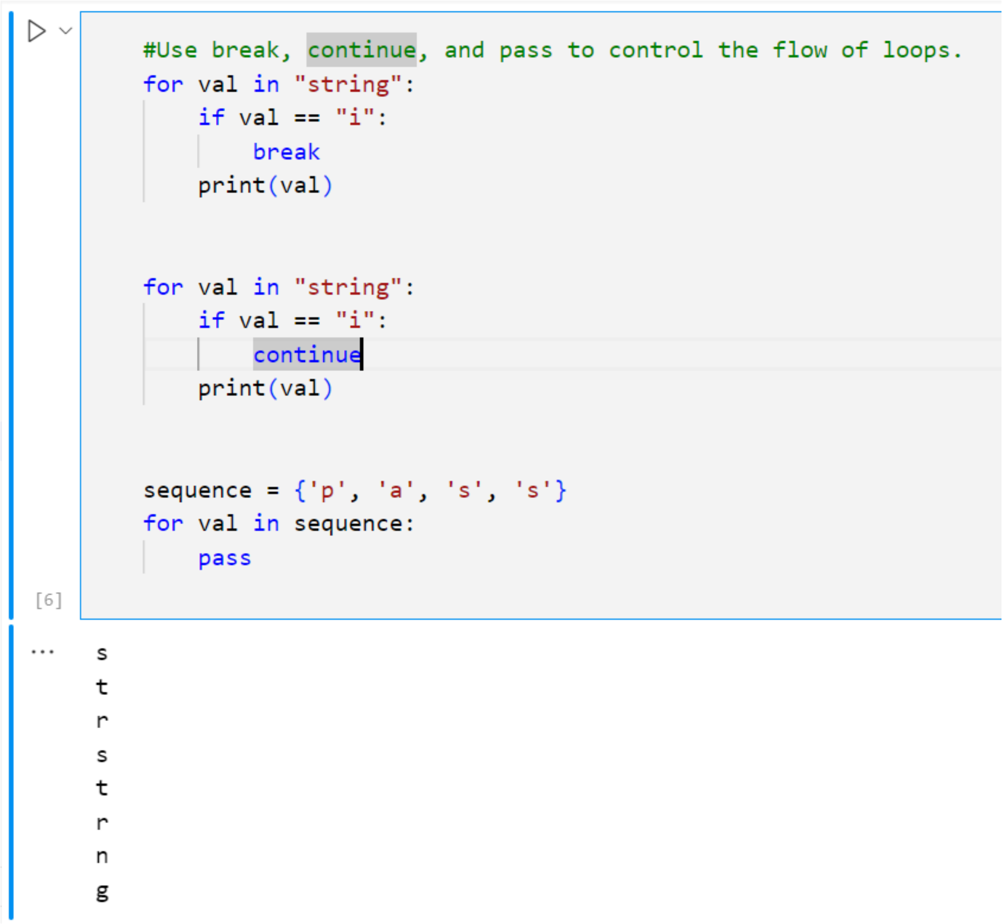
```
#Repeat a block of code as long as a condition is true using a while loop.
no=int(input("Enter no:"))
i=1
ans=1
while(i<= no):
    ans=ans*i
    i=i+1
print("Factorial of {0} is:{1}".format(no,ans))
```

```
Factorial of 5 is:120
```



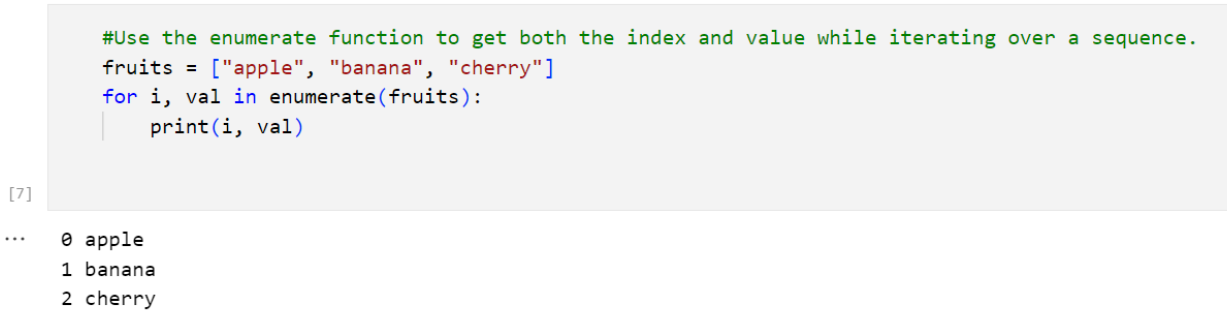
```python
#Use loops inside other loops to handle multi-dimensional data structures.
n=int(input('enter n value='))
for i in range(1,n+1):
    for j in range(1,i+1):
        print(i,end=' ')

    print()
```

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```



```python
#Use break, continue, and pass to control the flow of loops.
for val in "string":
    if val == "i":
        break
    print(val)


for val in "string":
    if val == "i":
        continue
    print(val)


sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

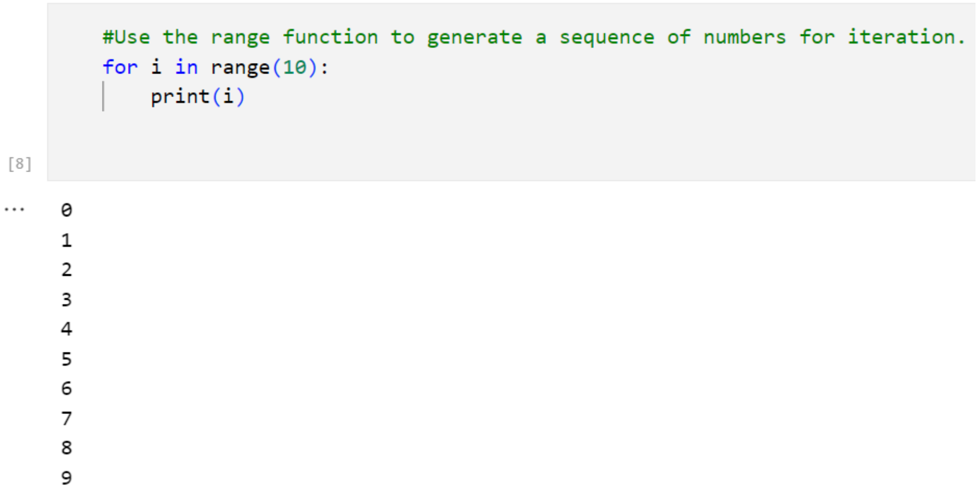
```
s
t
r
s
t
r
n
g
```



```python
#Use the enumerate function to get both the index and value while iterating over a sequence.
fruits = ["apple", "banana", "cherry"]
for i, val in enumerate(fruits):
    print(i, val)
```

[7]

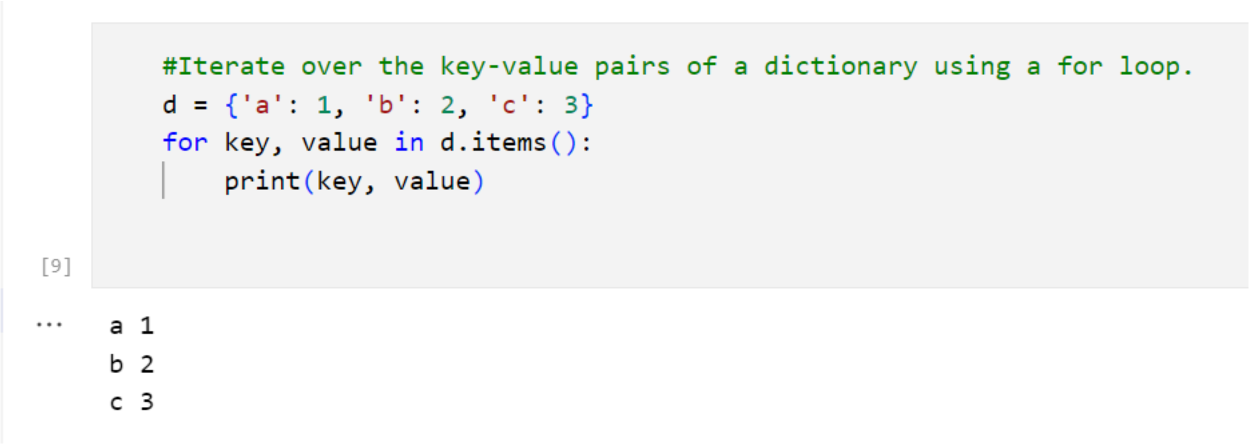
```
0 apple
1 banana
2 cherry
```



```python
#Use the range function to generate a sequence of numbers for iteration.
for i in range(10):
    print(i)
```

[8]

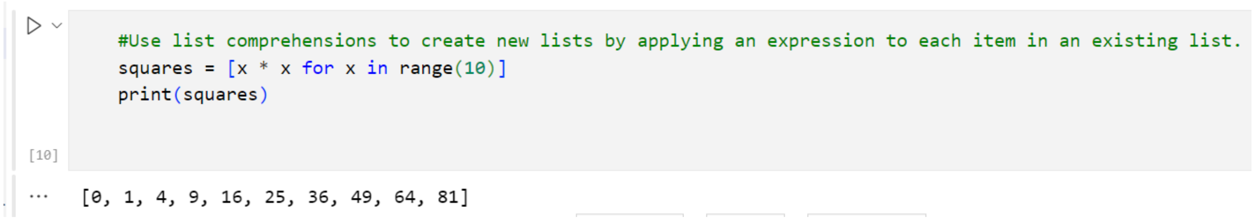
```
0
1
2
3
4
5
6
7
8
9
```



```python
#Iterate over the key-value pairs of a dictionary using a for loop.
d = {'a': 1, 'b': 2, 'c': 3}
for key, value in d.items():
    print(key, value)
```

[9]

```
a 1
b 2
c 3
```



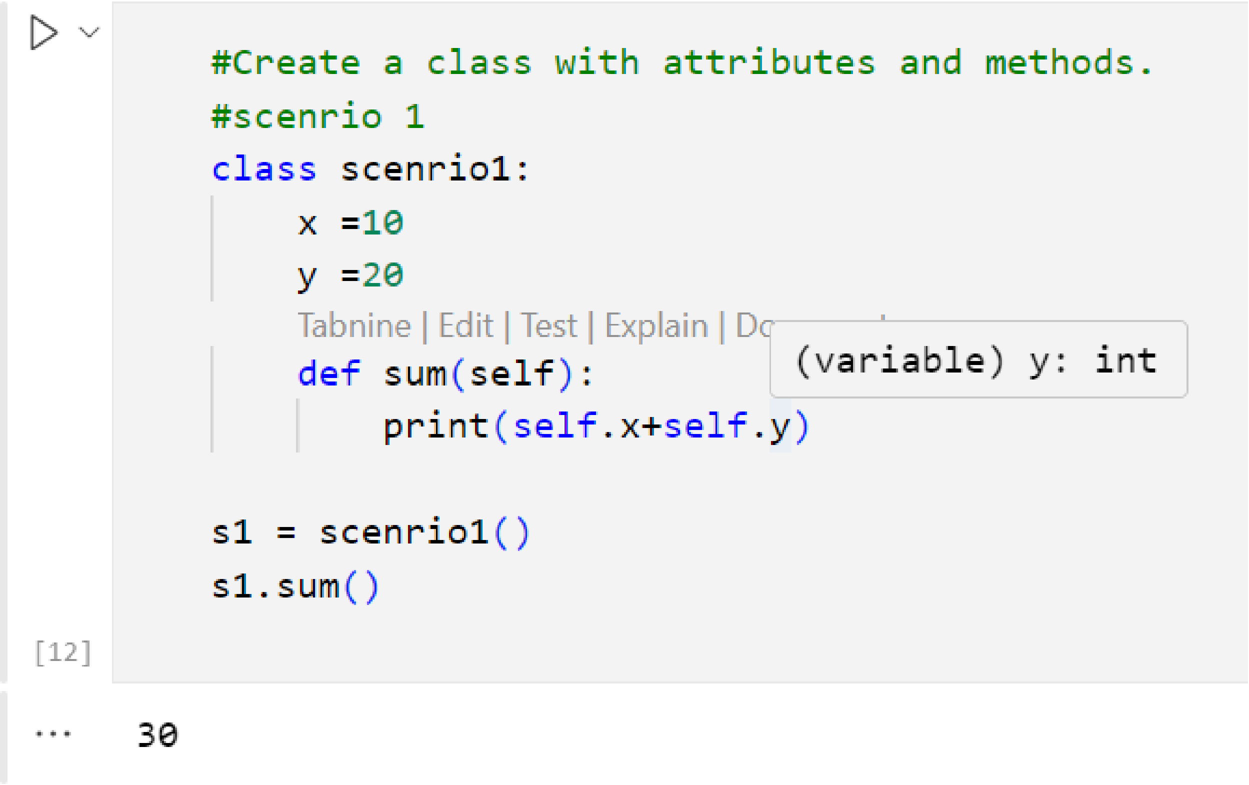
```python
#Use list comprehensions to create new lists by applying an expression to each item in an existing list.
squares = [x * x for x in range(10)]
print(squares)
```

[10]

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

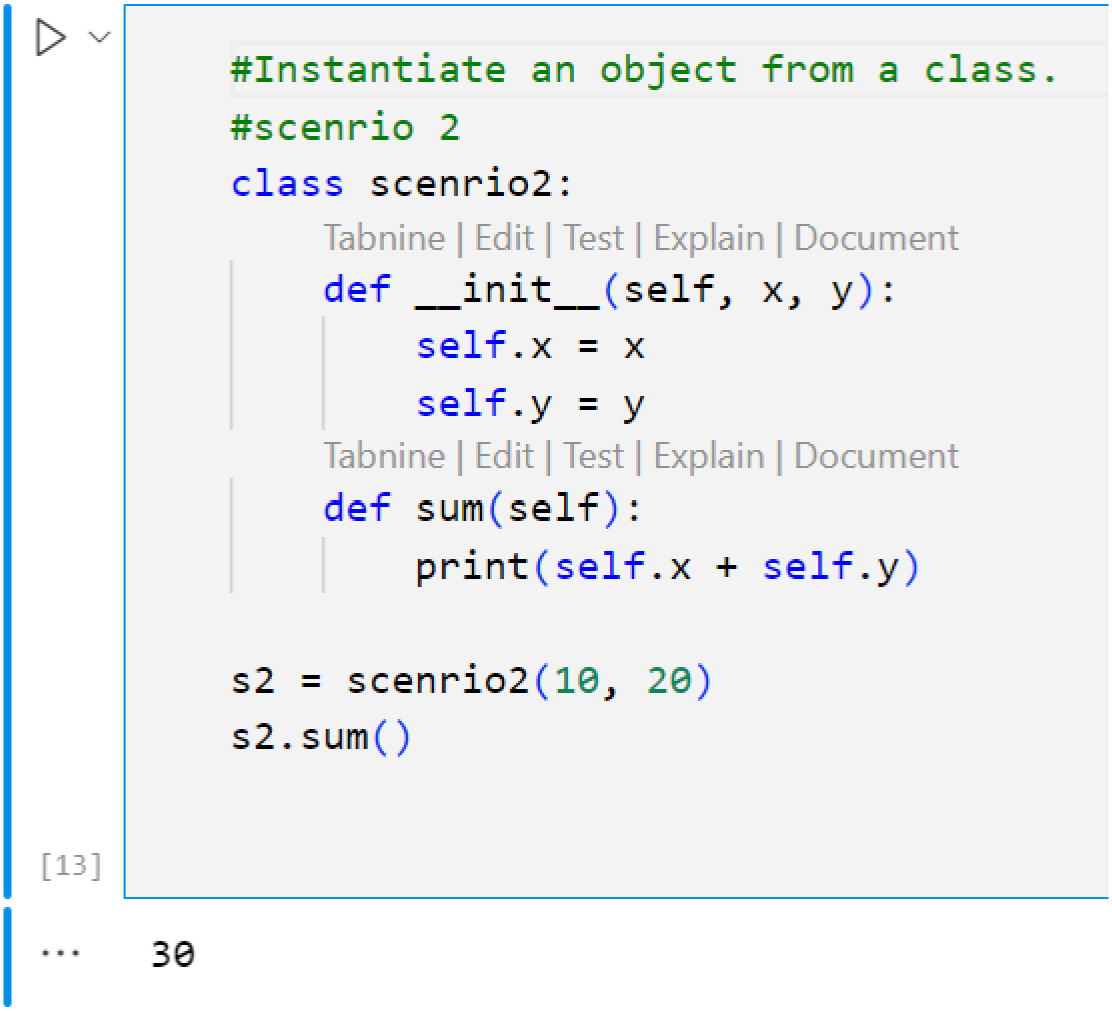
Practical 4: Class, Objects and Inheritance.

1. Create a class with attributes and methods.
2. Instantiate an object from a class.
3. Access and modify the attributes of an object.
4. Call methods defined in a class using an object.
5. Access and modify the attributes of an object using getter and setter methods.
6. Create a subclass that inherits from a superclass.
7. Override methods in a subclass to provide specific implementations.
8. Call methods from the superclass using the super() function.
9. Define and use class variables that are shared among all instances of a class.
10. Define and use instance variables that are unique to each object.
11. Create a class that inherit from multiple superclass.
12. Understand and use the method resolution order to determine the order in which base classes are searched.



```
#Create a class with attributes and methods.
#scenrio 1
class scenrio1:
    x =10
    y =20

    def sum(self):
        print(self.x+self.y)

s1 = scenrio1()
s1.sum()
```

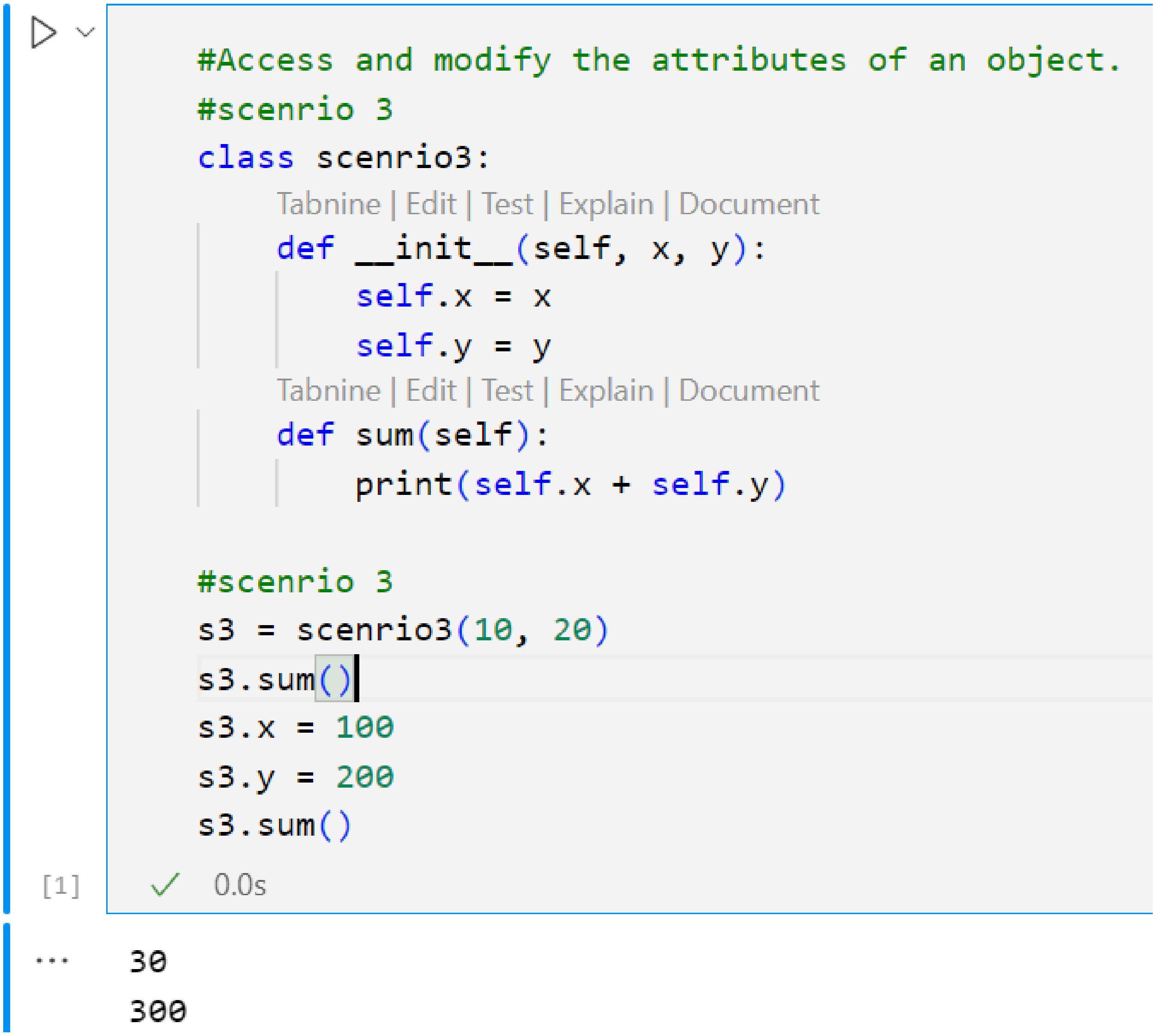
```
30
```



Tabnine | Edit | Test | Explain | Document
Tabnine | Edit | Test | Explain | Document
[13]
...

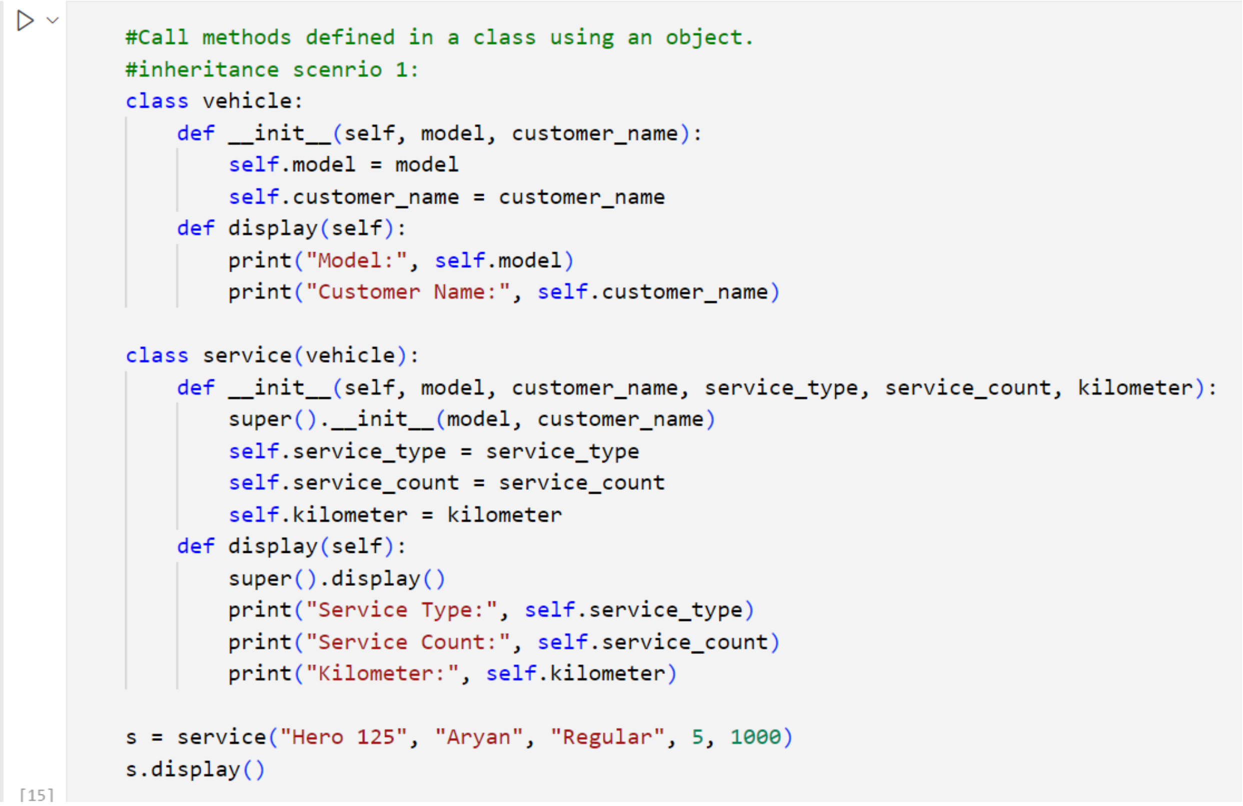
```python
#Instantiate an object from a class.
#scenrio 2
class scenrio2:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def sum(self):
        print(self.x + self.y)

s2 = scenrio2(10, 20)
s2.sum()
```

```
30
```


Tabnine | Edit | Test | Explain | Document
Tabnine | Edit | Test | Explain | Document
[1]
✓ 0.0s
...

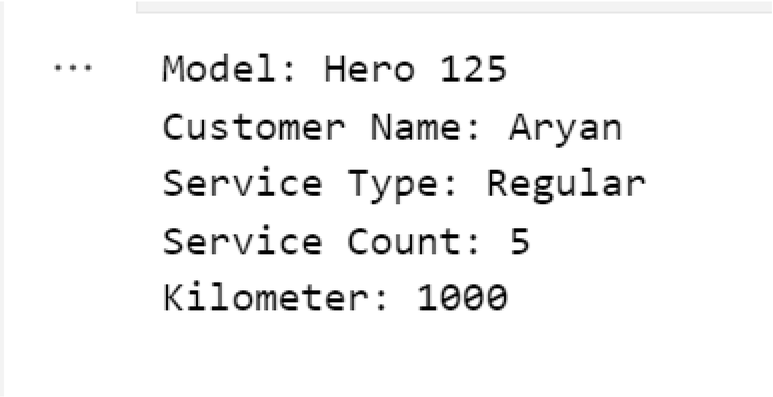
```python
#Access and modify the attributes of an object.
#scenrio 3
class scenrio3:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def sum(self):
        print(self.x + self.y)

#scenrio 3
s3 = scenrio3(10, 20)
s3.sum()
s3.x = 100
s3.y = 200
s3.sum()
```

```
30
300
```

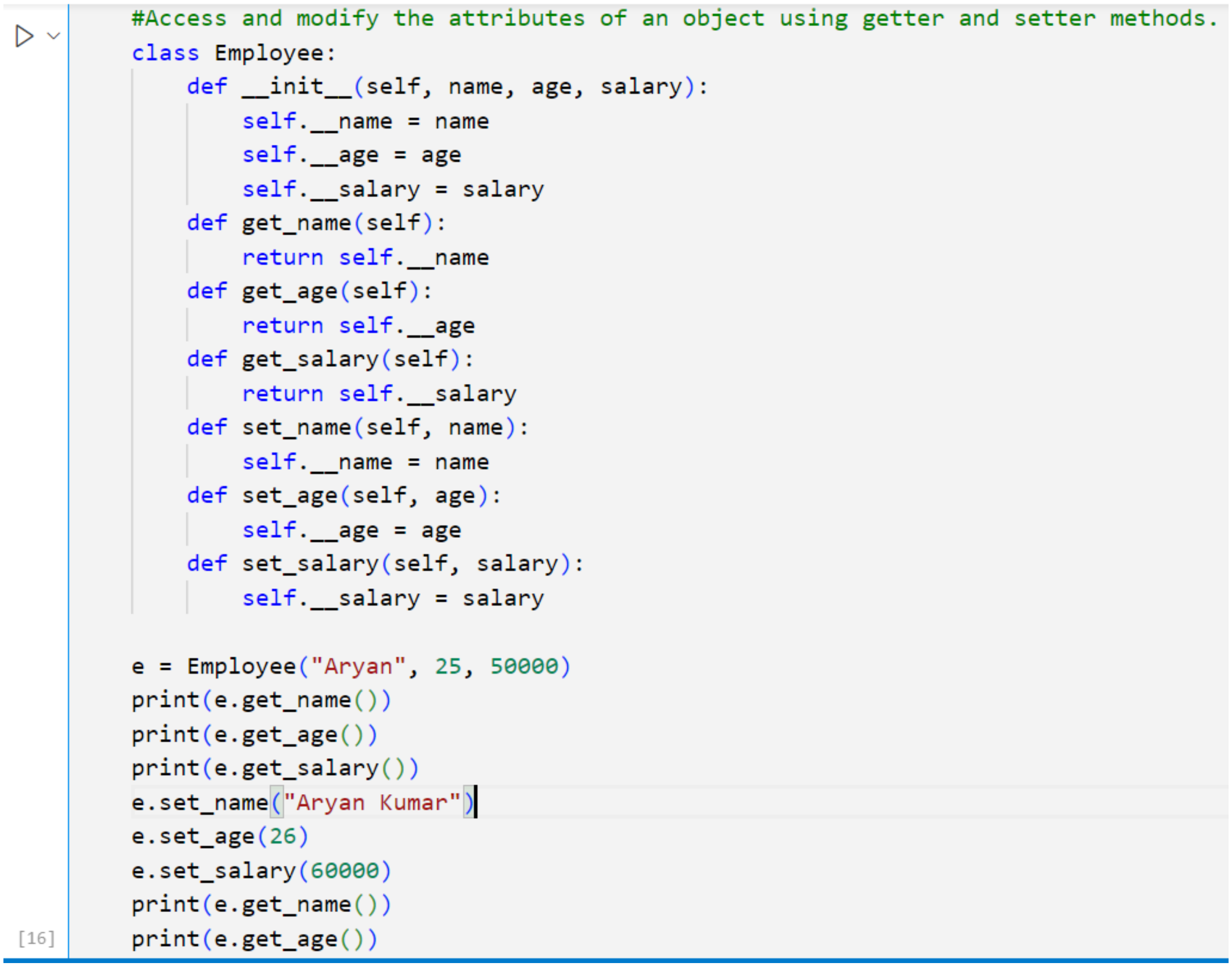


```python
#Call methods defined in a class using an object.
#inheritance scenrio 1:
class vehicle:
    def __init__(self, model, customer_name):
        self.model = model
        self.customer_name = customer_name
    def display(self):
        print("Model:", self.model)
        print("Customer Name:", self.customer_name)

class service(vehicle):
    def __init__(self, model, customer_name, service_type, service_count, kilometer):
        super().__init__(model, customer_name)
        self.service_type = service_type
        self.service_count = service_count
        self.kilometer = kilometer
    def display(self):
        super().display()
        print("Service Type:", self.service_type)
        print("Service Count:", self.service_count)
        print("Kilometer:", self.kilometer)

s = service("Hero 125", "Aryan", "Regular", 5, 1000)
s.display()
```

Output:

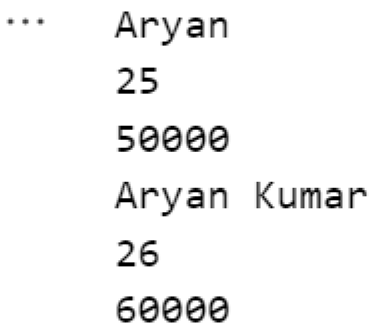


```
Model: Hero 125
Customer Name: Aryan
Service Type: Regular
Service Count: 5
Kilometer: 1000
```

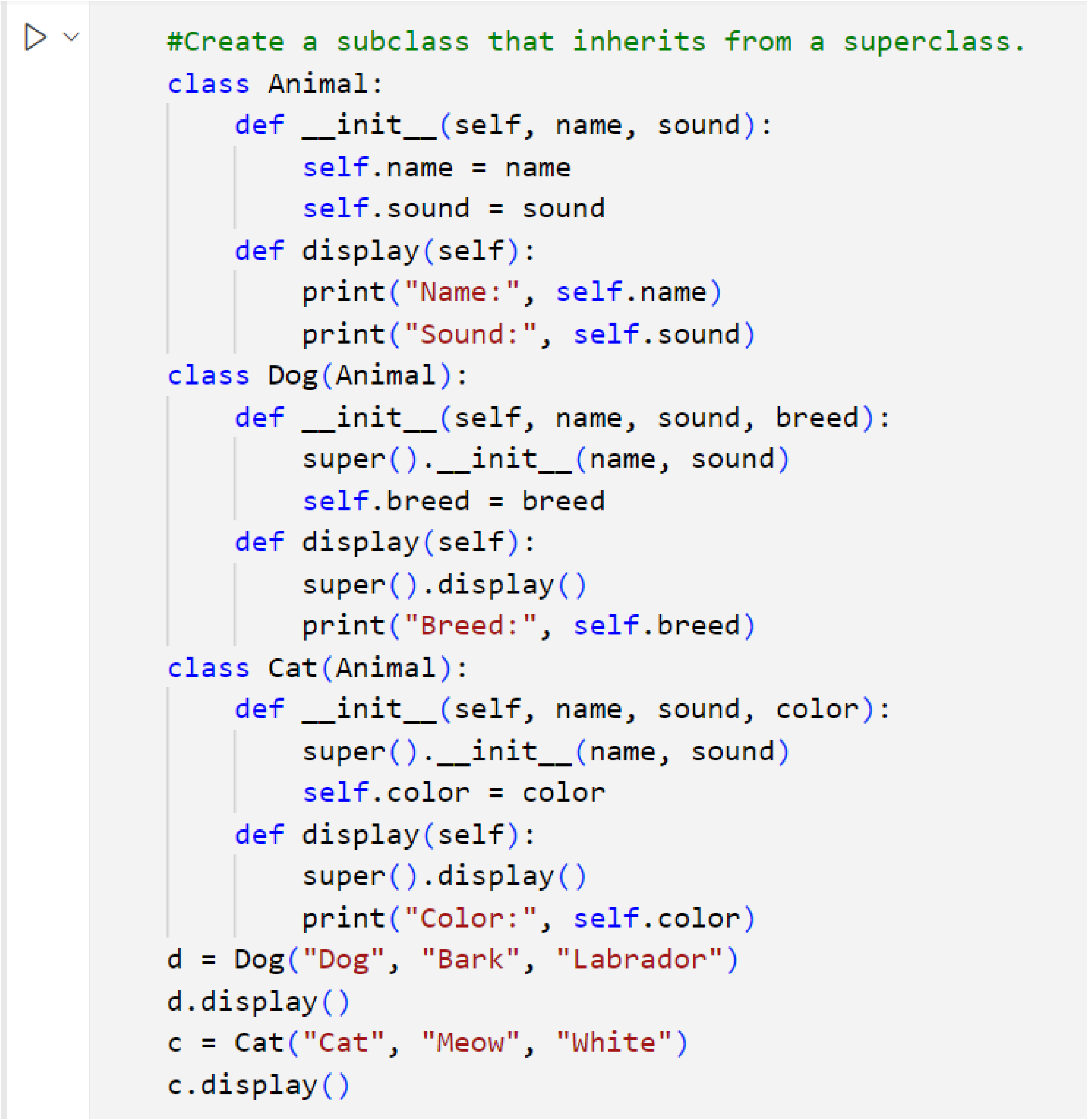


```python
#Access and modify the attributes of an object using getter and setter methods.
class Employee:
    def __init__(self, name, age, salary):
        self.__name = name
        self.__age = age
        self.__salary = salary
    def get_name(self):
        return self.__name
    def get_age(self):
        return self.__age
    def get_salary(self):
        return self.__salary
    def set_name(self, name):
        self.__name = name
    def set_age(self, age):
        self.__age = age
    def set_salary(self, salary):
        self.__salary = salary

e = Employee("Aryan", 25, 50000)
print(e.get_name())
print(e.get_age())
print(e.get_salary())
e.set_name("Aryan Kumar")
e.set_age(26)
e.set_salary(60000)
print(e.get_name())
print(e.get_age())
```

Output:

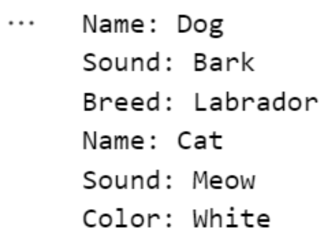


```
Aryan
25
50000
Aryan Kumar
26
60000
```

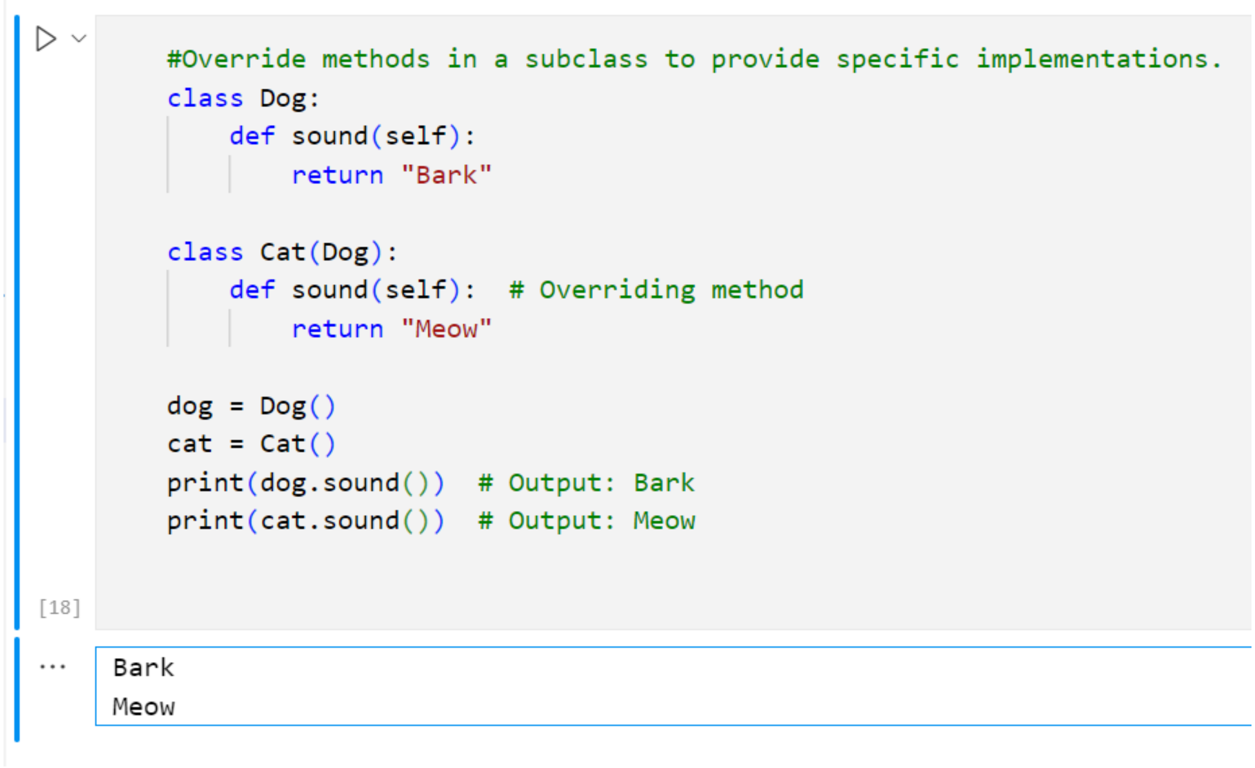


```python
#Create a subclass that inherits from a superclass.
class Animal:
    def __init__(self, name, sound):
        self.name = name
        self.sound = sound
    def display(self):
        print("Name:", self.name)
        print("Sound:", self.sound)
class Dog(Animal):
    def __init__(self, name, sound, breed):
        super().__init__(name, sound)
        self.breed = breed
    def display(self):
        super().display()
        print("Breed:", self.breed)
class Cat(Animal):
    def __init__(self, name, sound, color):
        super().__init__(name, sound)
        self.color = color
    def display(self):
        super().display()
        print("Color:", self.color)
d = Dog("Dog", "Bark", "Labrador")
d.display()
c = Cat("Cat", "Meow", "White")
c.display()
```

Output:

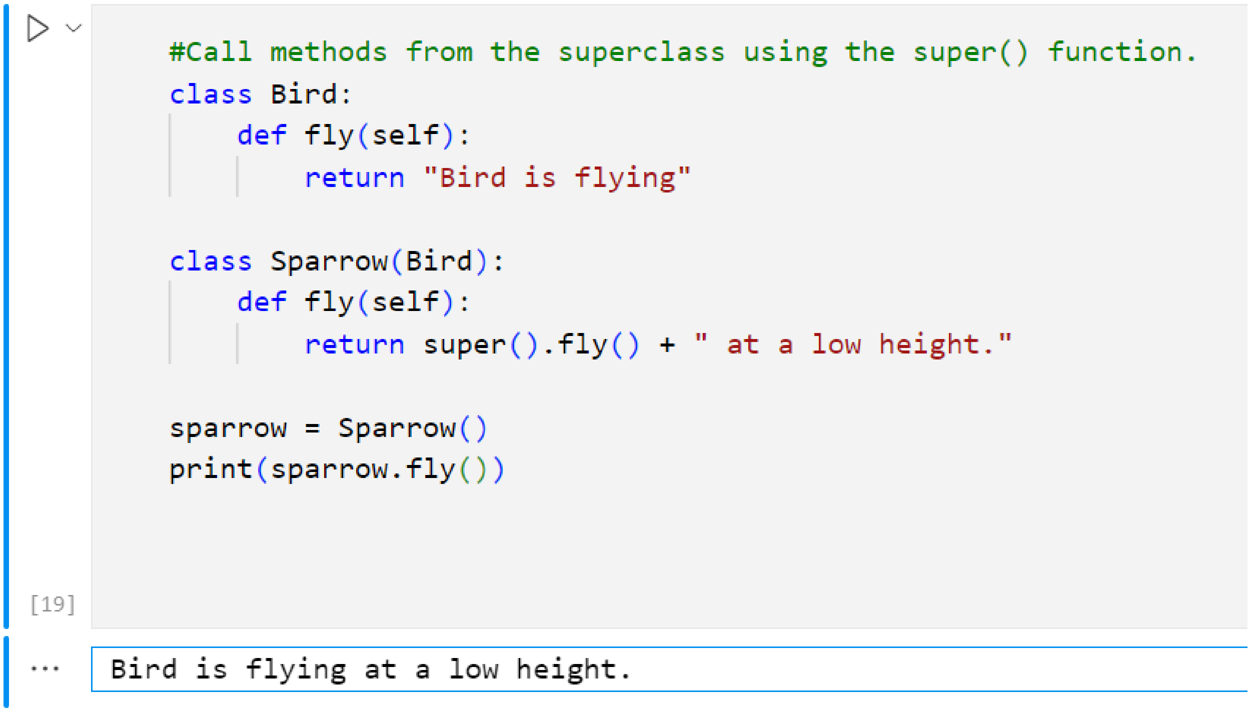


```
...  Name: Dog
     Sound: Bark
     Breed: Labrador
     Name: Cat
     Sound: Meow
     Color: White
```



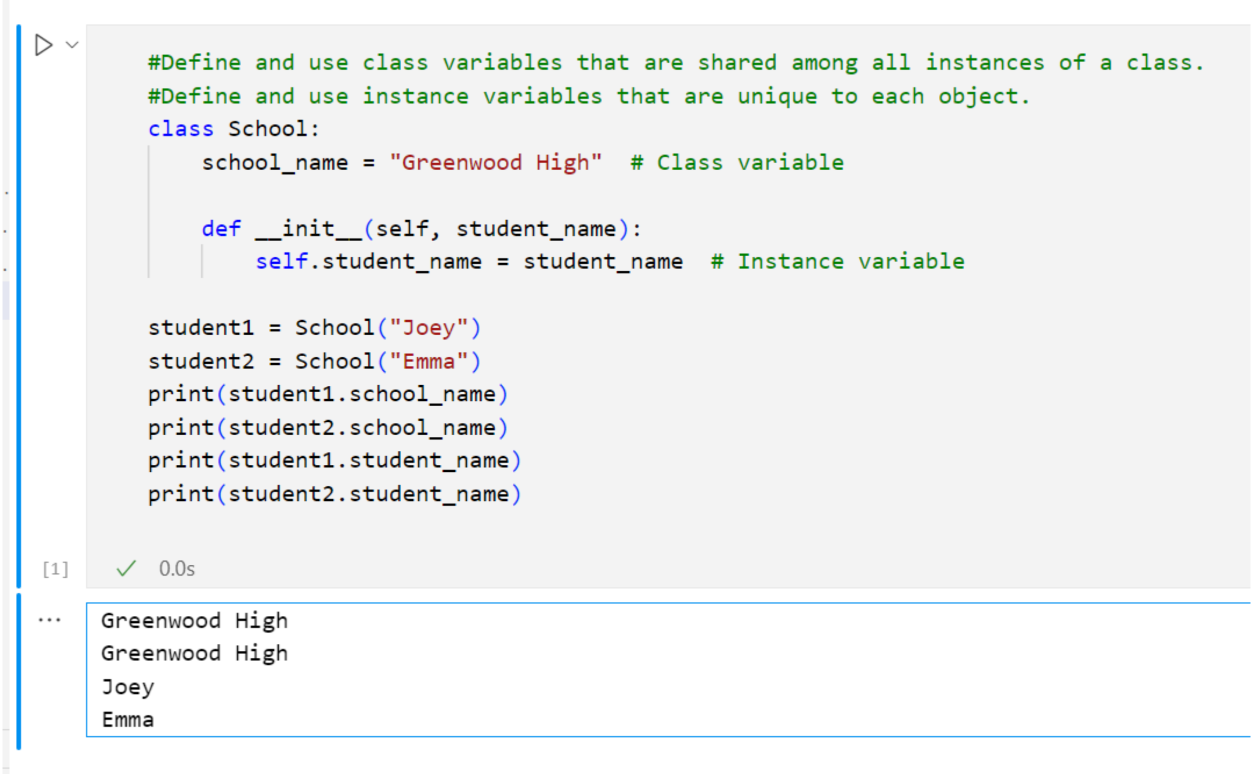
```
#Override methods in a subclass to provide specific implementations.
class Dog:
    def sound(self):
        return "Bark"

class Cat(Dog):
    def sound(self):  # Overriding method
        return "Meow"

dog = Dog()
cat = Cat()
print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow
```

```
Bark
Meow
```



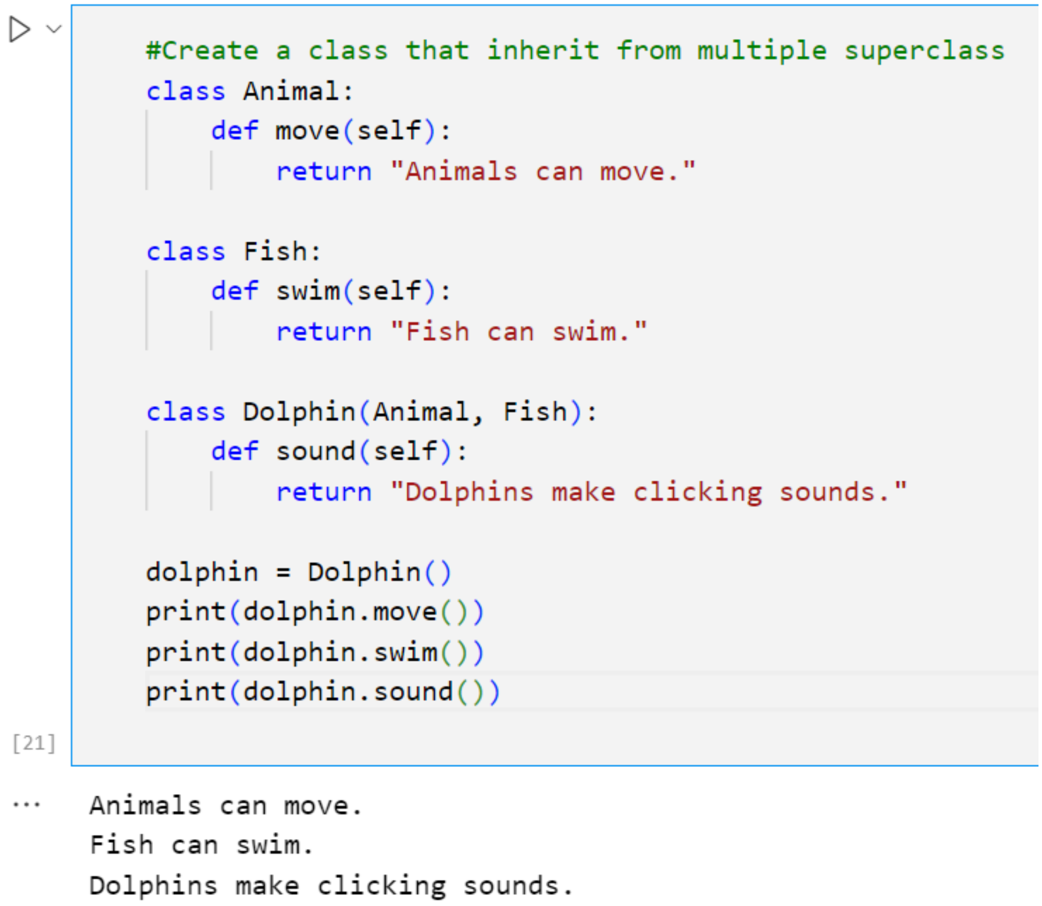
```
#Call methods from the superclass using the super() function.
class Bird:
    def fly(self):
        return "Bird is flying"

class Sparrow(Bird):
    def fly(self):
        return super().fly() + " at a low height."

sparrow = Sparrow()
print(sparrow.fly())
```

```
Bird is flying at a low height.
```



```python
#Define and use class variables that are shared among all instances of a class.
#Define and use instance variables that are unique to each object.
class School:
    school_name = "Greenwood High"  # Class variable

    def __init__(self, student_name):
        self.student_name = student_name  # Instance variable

student1 = School("Joey")
student2 = School("Emma")
print(student1.school_name)
print(student2.school_name)
print(student1.student_name)
print(student2.student_name)
```

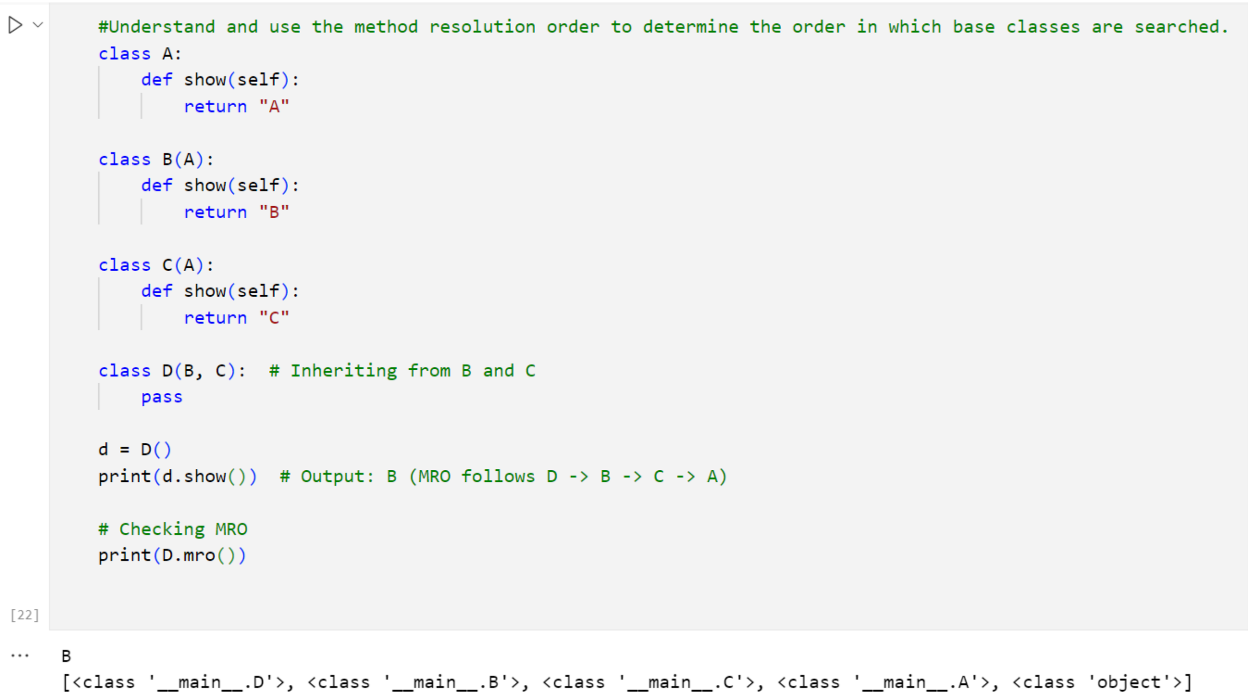
```
Greenwood High
Greenwood High
Joey
Emma
```



```
#Create a class that inherit from multiple superclass
class Animal:
    def move(self):
        return "Animals can move."

class Fish:
    def swim(self):
        return "Fish can swim."

class Dolphin(Animal, Fish):
    def sound(self):
        return "Dolphins make clicking sounds."

dolphin = Dolphin()
print(dolphin.move())
print(dolphin.swim())
print(dolphin.sound())
```

[21]

```
Animals can move.
Fish can swim.
Dolphins make clicking sounds.
```



```
#Understand and use the method resolution order to determine the order in which base classes are searched.
class A:
    def show(self):
        return "A"

class B(A):
    def show(self):
        return "B"

class C(A):
    def show(self):
        return "C"

class D(B, C):  # Inheriting from B and C
    pass

d = D()
print(d.show())  # Output: B (MRO follows D -> B -> C -> A)

# Checking MRO
print(D.mro())
```

[22]

```
B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```