PROGRAMMING IN PYTHON

MCA-161A

4 Credits (3-0-2)

MCA 5th Sem (2020-21)

R K Dwivedi
Assistant Professor
Department of ITCA
MMMUT Gorakhpur

UNIT III: OOP and File Handling

A. Object Oriented Programming:

Classes and objects
Attributes and methods
Constructors and destructors
Inheritance
Polymorphism
Exception Handling (Try…Except)

B. Management of text files:

Type of files
Various file operations on text files
Creating a text file
Opening a file
Closing a file
Reading a text file
Writing into a text file
Copying a file to another file

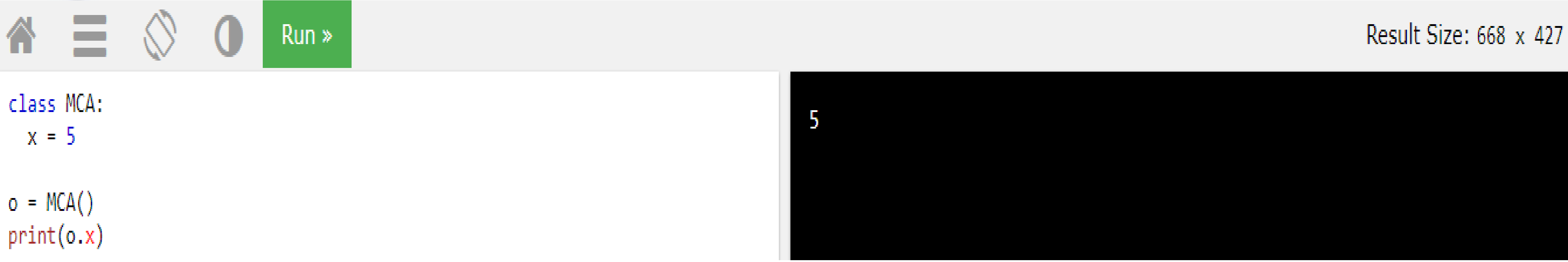
C. Some more concepts:

Packages and Modules
Replacement of Access Specifiers in Python
Replacement of Switch Case in Python

A. Object Oriented Programming

1. Classes and Objects

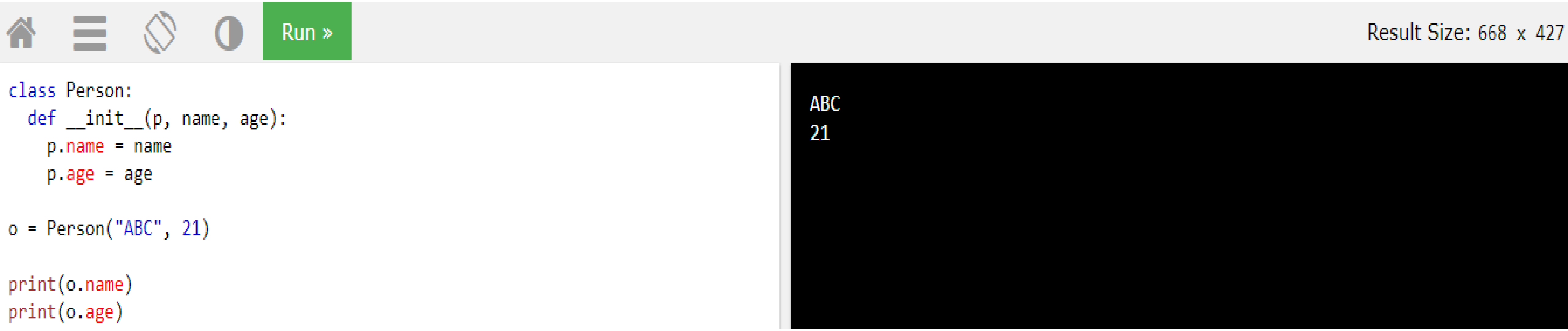
1. Classes and Objects



```
class MCA:
  x = 5

o = MCA()
print(o.x)
```

```
5
```

- All classes have a function called __init__( ), which is always executed when the class is being initiated.
- The __init__( ) function is called automatically every time the class is being used to create a new object.
- The first parameter of __init__( ) is a reference to the current instance of the class, and is used to access the variables that belongs to the class.

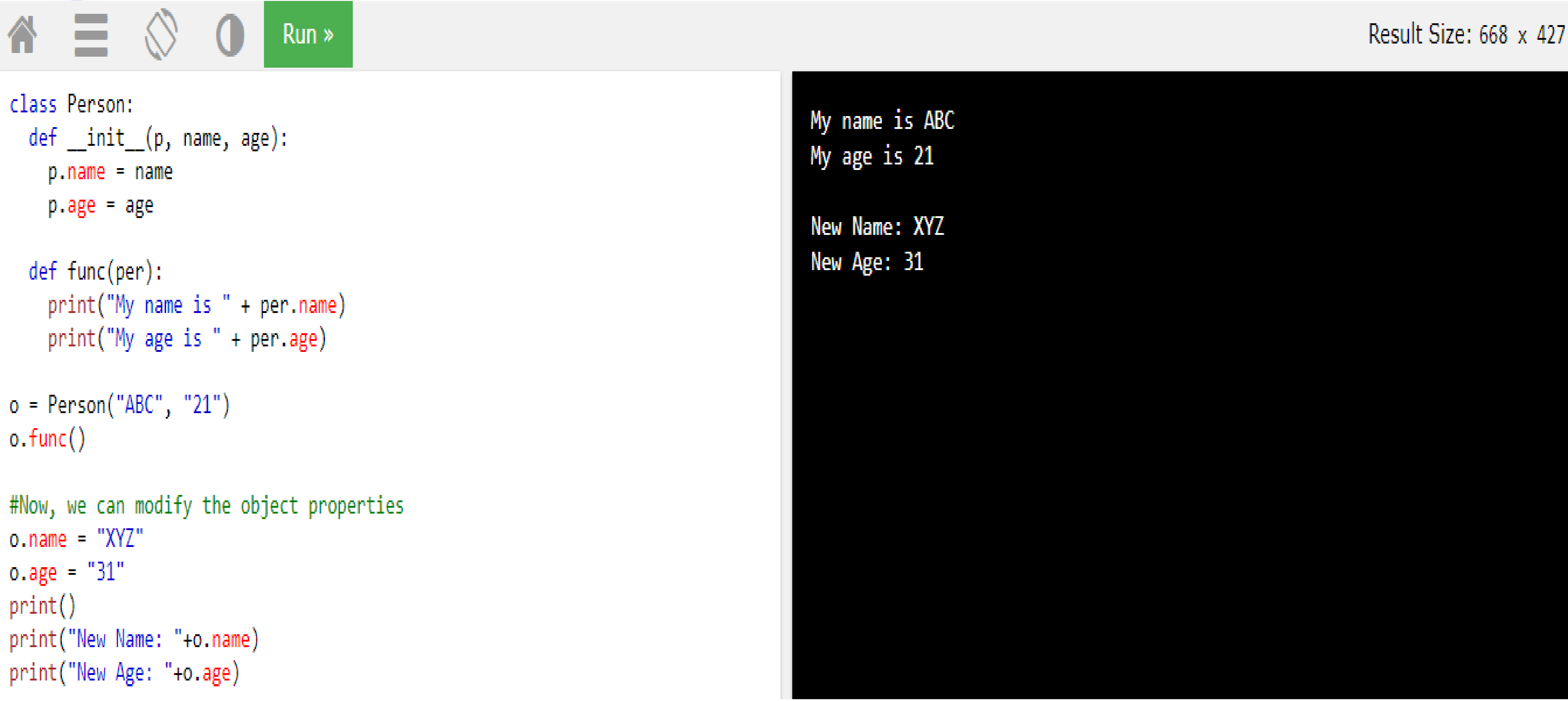


```
class Person:
  def __init__(p, name, age):
    p.name = name
    p.age = age

o = Person("ABC", 21)

print(o.name)
print(o.age)
```

```
ABC
21
```

2. Attributes and Methods

2. Attributes and Methods



```
class Person:
  def __init__(p, name, age):
    p.name = name
    p.age = age

  def func(per):
    print("My name is " + per.name)
    print("My age is " + per.age)

o = Person("ABC", "21")
o.func()

#Now, we can modify the object properties
o.name = "XYZ"
o.age = "31"
print()
print("New Name: "+o.name)
print("New Age: "+o.age)
```

```
My name is ABC
My age is 21

New Name: XYZ
New Age: 31
```

3. Constructors and Destructors

3. Constructors and Destructors

Constructors

- Constructors are generally used for instantiating an object.
- In Python the **__init__**( ) method is called the constructor and is always called when an object is created.
- The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.

Types of constructors :

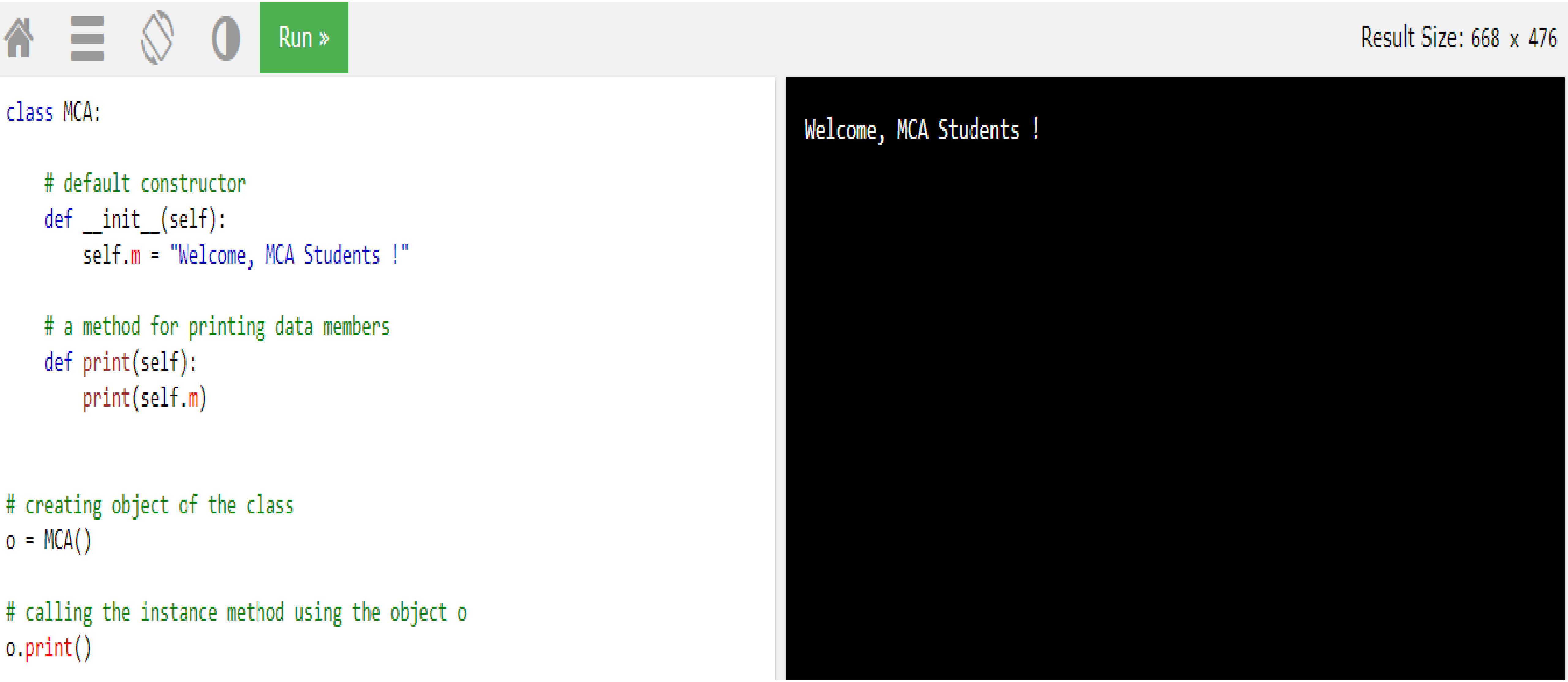
1. Default constructor : The default constructor is simple constructor which doesn't accept any arguments. It's definition has only one argument which is a reference to the instance being constructed.

2. Parameterized constructor : Constructor with parameters is known as parameterized constructor. Parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

3. Constructors and Destructors

...continued

A. Default Constructors



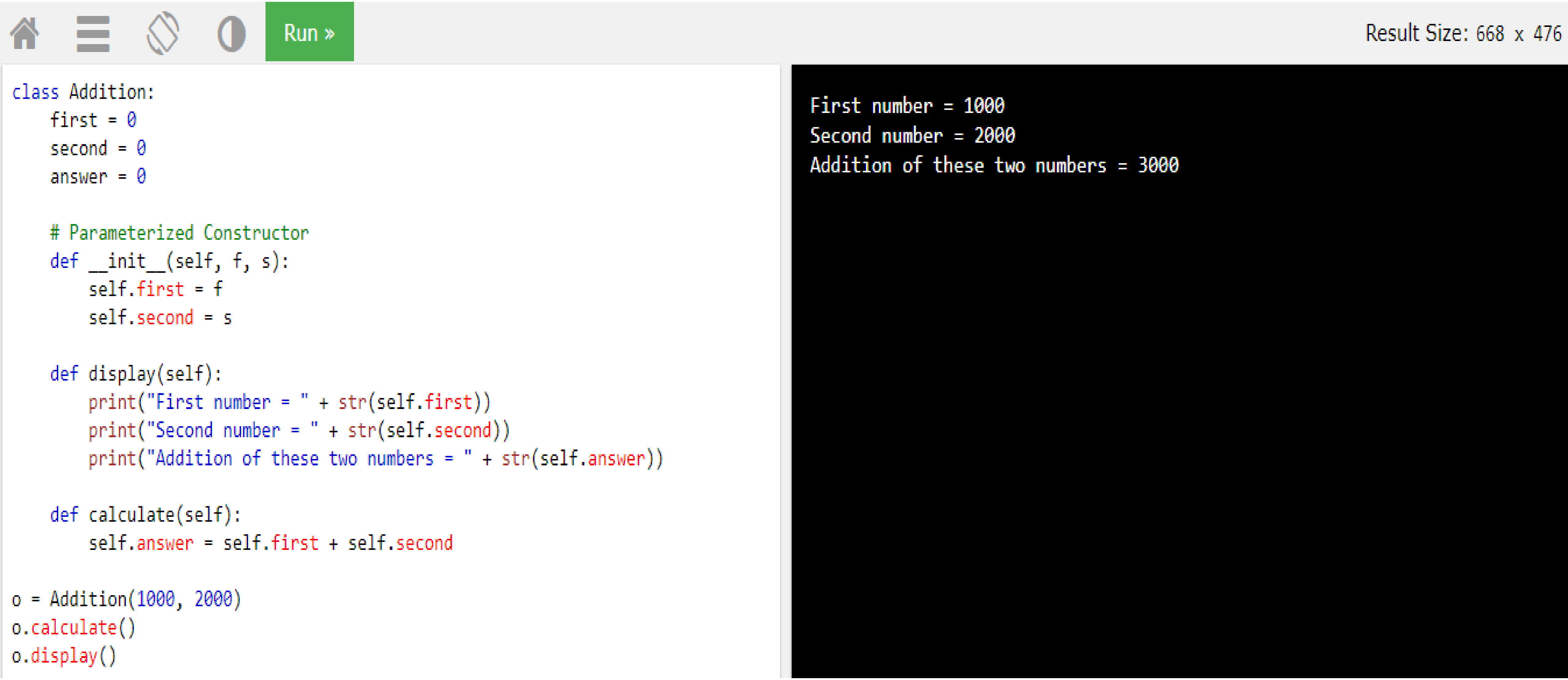
```
class MCA:

    # default constructor
    def __init__(self):
        self.m = "Welcome, MCA Students !"

    # a method for printing data members
    def print(self):
        print(self.m)


# creating object of the class
o = MCA()

# calling the instance method using the object o
o.print()
```

```
Welcome, MCA Students !
```

3. Constructors and Destructors

…continued

B. Parameterized Constructors



```
class Addition:
    first = 0
    second = 0
    answer = 0

    # Parameterized Constructor
    def __init__(self, f, s):
        self.first = f
        self.second = s

    def display(self):
        print("First number = " + str(self.first))
        print("Second number = " + str(self.second))
        print("Addition of these two numbers = " + str(self.answer))

    def calculate(self):
        self.answer = self.first + self.second

o = Addition(1000, 2000)
o.calculate()
o.display()
```

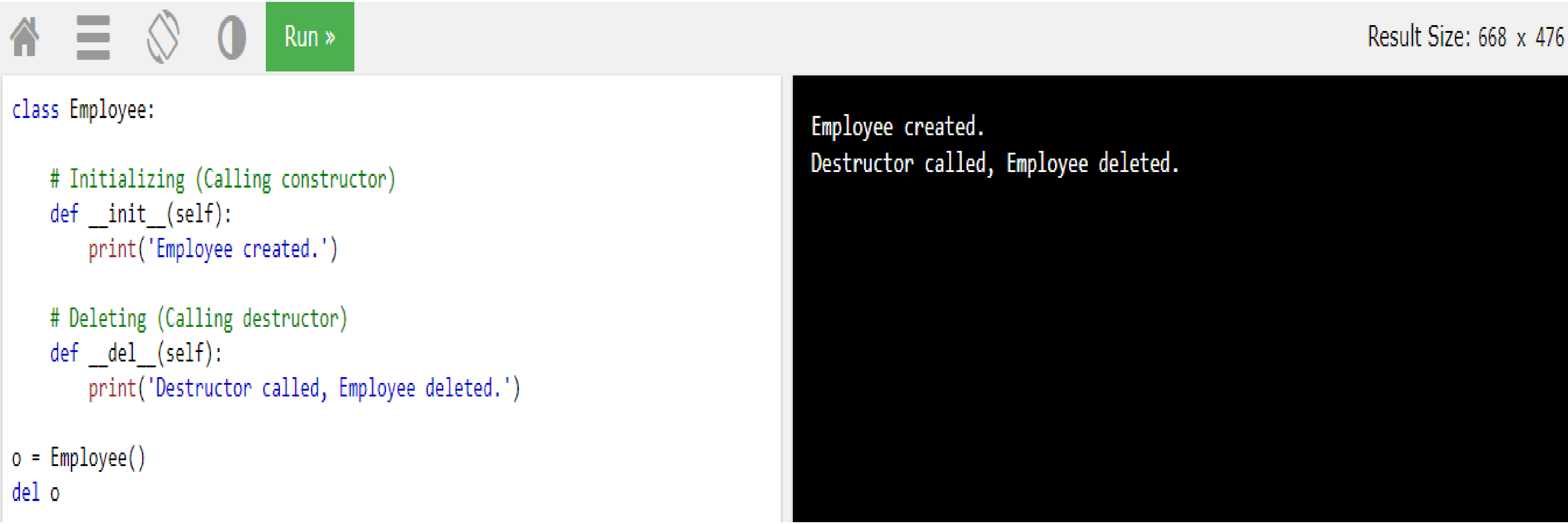
```
First number = 1000
Second number = 2000
Addition of these two numbers = 3000
```

3. Constructors and Destructors

...continued

Destructors

- Destructors are called when an object gets destroyed.
- The **__del__**( ) method is a known as a destructor method in Python.
- It is called when all references to the object have been deleted i.e. when an object is garbage collected.
- In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.



```
class Employee:

    # Initializing (Calling constructor)
    def __init__(self):
        print('Employee created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')

o = Employee()
del o
```

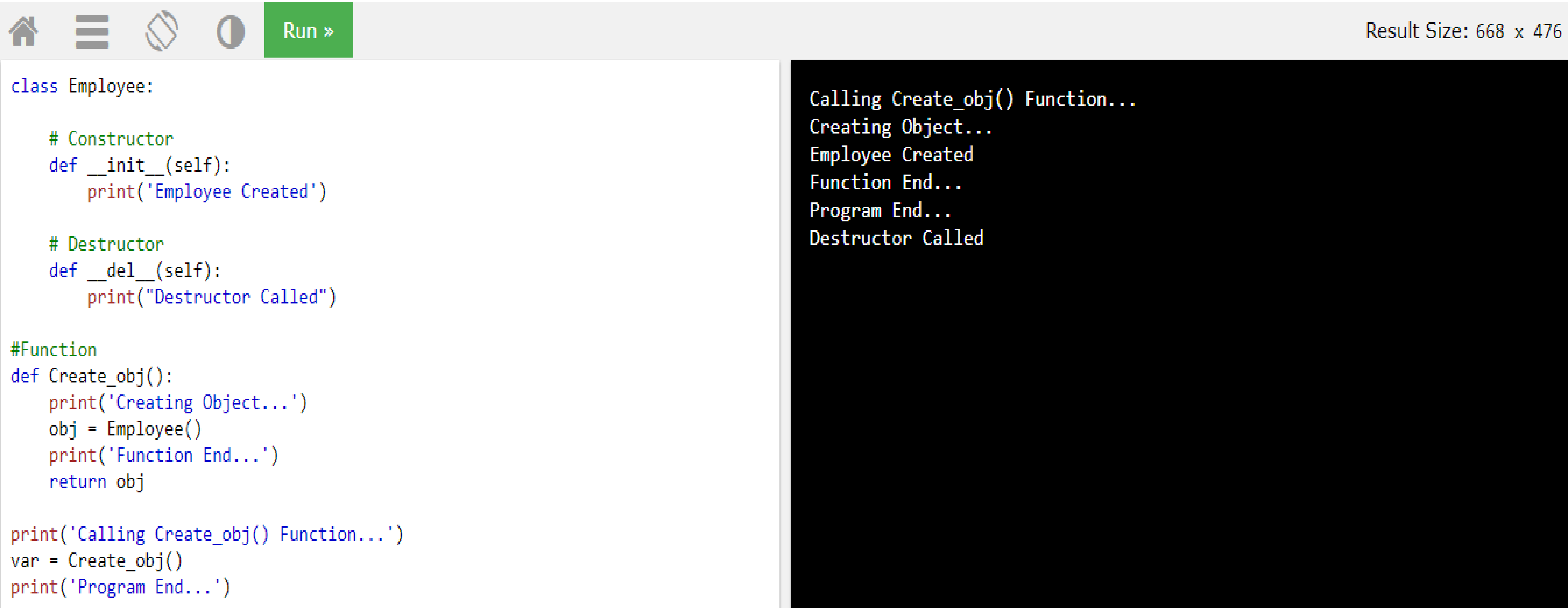
```
Employee created.
Destructor called, Employee deleted.
```

3. Constructors and Destructors

…continued

Destructors (Order of invocation)

The destructor is called **after the program is ended** or **when all the references to object are deleted** (i.e. when the reference count becomes zero, not when the object goes out of scope).



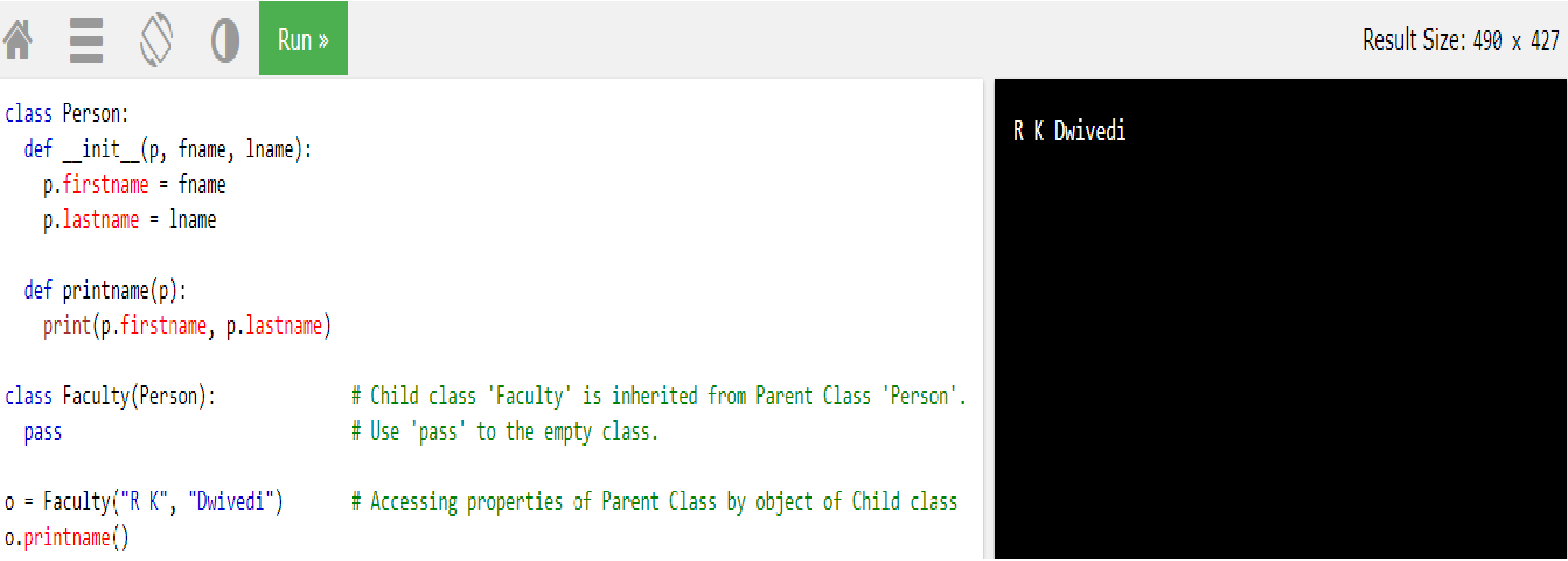
```python
class Employee:

    # Constructor
    def __init__(self):
        print('Employee Created')

    # Destructor
    def __del__(self):
        print("Destructor Called")

#Function
def Create_obj():
    print('Creating Object...')
    obj = Employee()
    print('Function End...')
    return obj

print('Calling Create_obj() Function...')
var = Create_obj()
print('Program End...')
```

```
Calling Create_obj() Function...
Creating Object...
Employee Created
Function End...
Program End...
Destructor Called
```

4. Inheritance

4. Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent (Base/ Super) class** is the class that is being inherited from another class.
- **Child (Derived/ Sub) class** is the class that inherits from another class.
- In Python, every class whether built-in or user-defined is derived from the **object** class and all the objects are instances of the class **object**. Hence, **object class is the base class for all the other classes**.



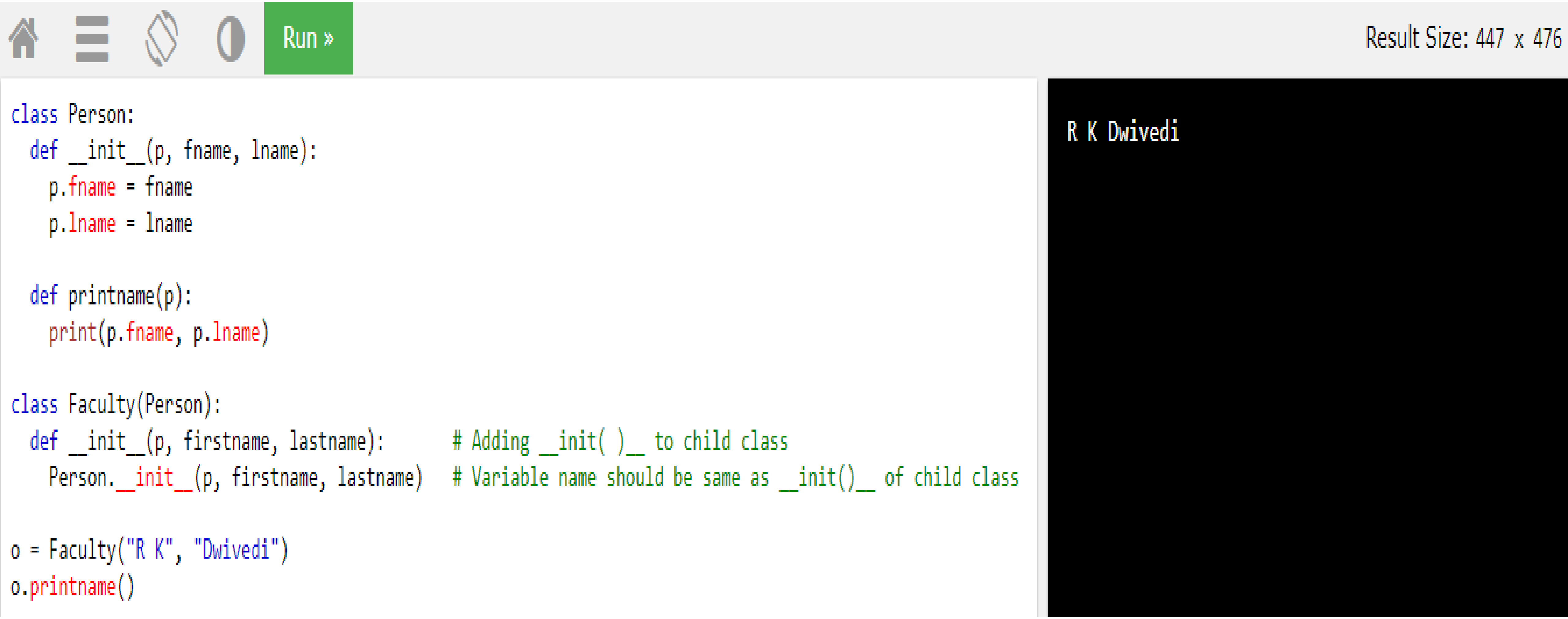
```python
class Person:
  def __init__(p, fname, lname):
    p.firstname = fname
    p.lastname = lname

  def printname(p):
    print(p.firstname, p.lastname)

class Faculty(Person):          # Child class 'Faculty' is inherited from Parent Class 'Person'.
  pass                          # Use 'pass' to the empty class.

o = Faculty("R K", "Dwivedi")   # Accessing properties of Parent Class by object of Child class
o.printname()
```

```
R K Dwivedi
```

4. Inheritance

...continued

- When you add the __init__( ) function in the child class, the child class will no longer inherit the parent's __init__( ) function.
- The child's __init__( ) function overrides the inheritance of the parent's __init__( ) function.
- **To keep the inheritance of the parent's __init__( ) function, add a call to the parent's __init__( ) function**



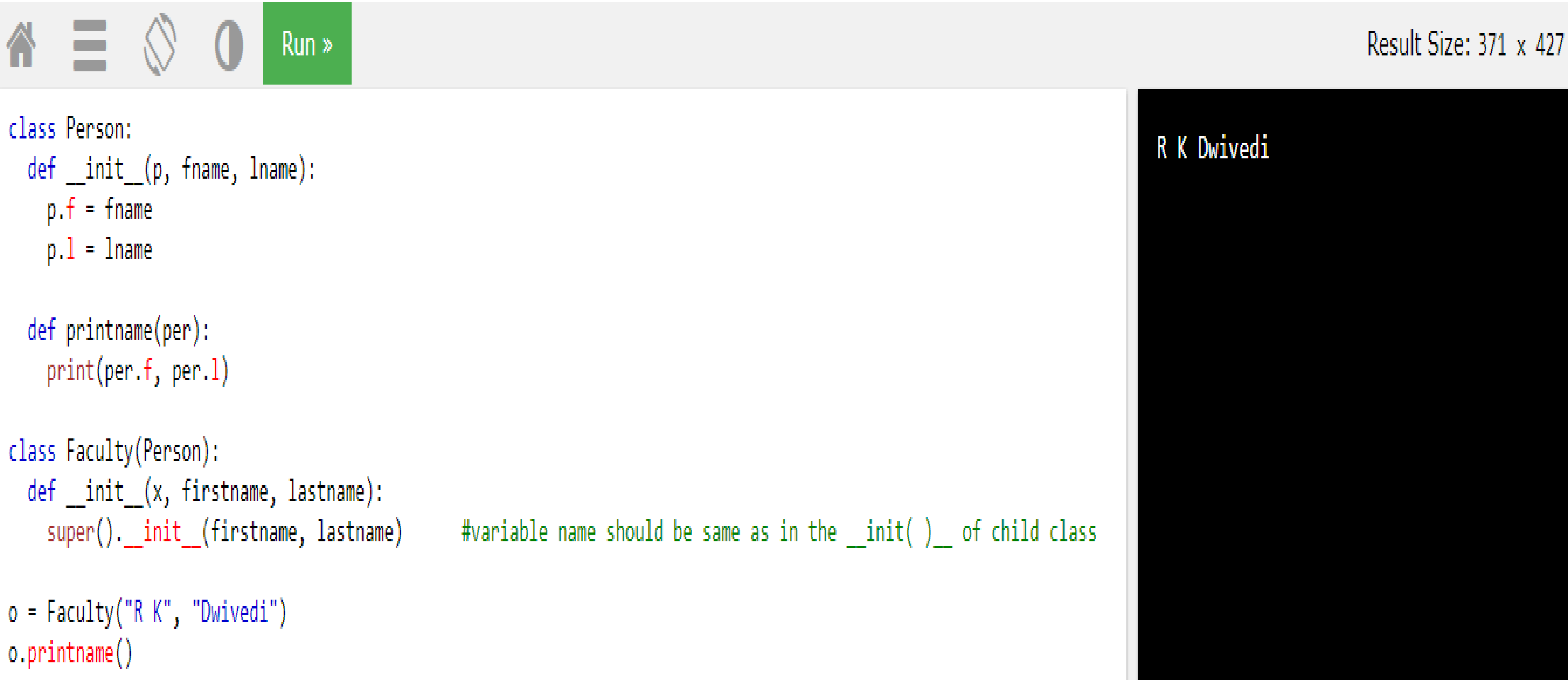
```
class Person:
  def __init__(p, fname, lname):
    p.fname = fname
    p.lname = lname

  def printname(p):
    print(p.fname, p.lname)

class Faculty(Person):
  def __init__(p, firstname, lastname):      # Adding __init( )__ to child class
    Person.__init__(p, firstname, lastname)  # Variable name should be same as __init()__ of child class

o = Faculty("R K", "Dwivedi")
o.printname()
```

```
R K Dwivedi
```

4. Inheritance

…continued

- Python also has a **super( )** function that will make the **child class to inherit all the methods and properties from its parent**.



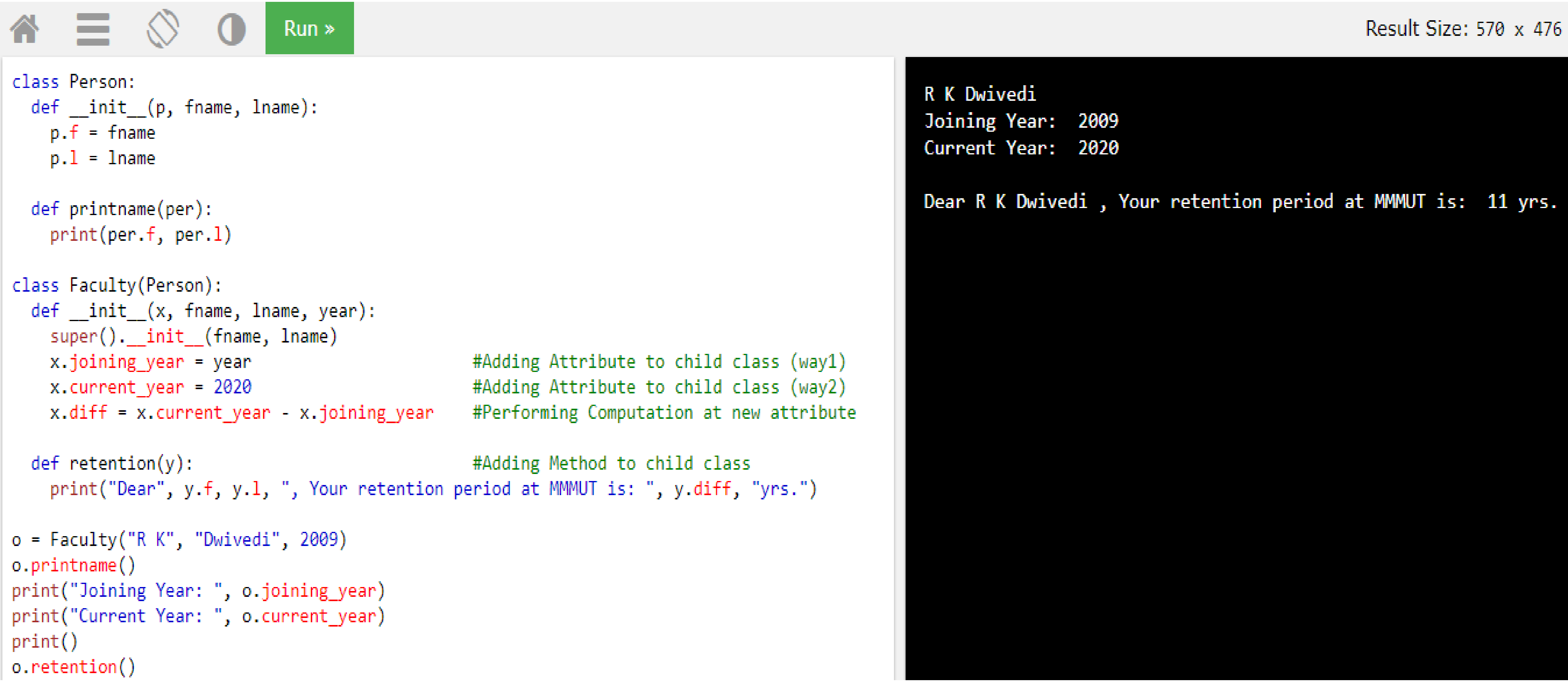
```python
class Person:
  def __init__(p, fname, lname):
    p.f = fname
    p.l = lname

  def printname(per):
    print(per.f, per.l)

class Faculty(Person):
  def __init__(x, firstname, lastname):
    super().__init__(firstname, lastname)      #variable name should be same as in the __init( )__ of child class

o = Faculty("R K", "Dwivedi")
o.printname()
```

```
R K Dwivedi
```

4. Inheritance

...continued

- We can also add **attributes** and **methods** to **child class**.



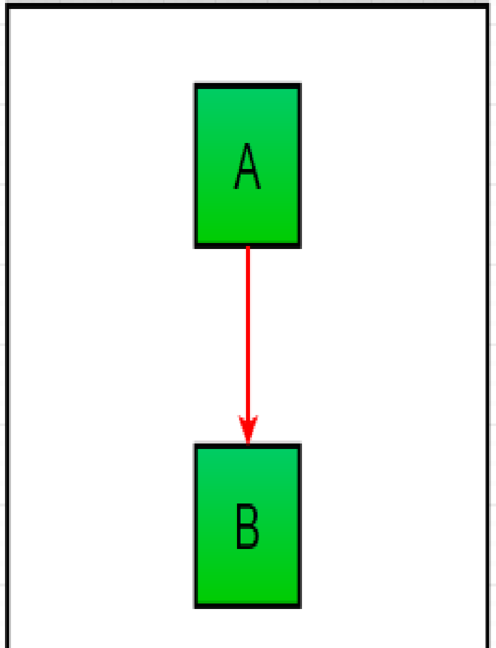
```
class Person:
  def __init__(p, fname, lname):
    p.f = fname
    p.l = lname

  def printname(per):
    print(per.f, per.l)

class Faculty(Person):
  def __init__(x, fname, lname, year):
    super().__init__(fname, lname)
    x.joining_year = year                 #Adding Attribute to child class (way1)
    x.current_year = 2020                 #Adding Attribute to child class (way2)
    x.diff = x.current_year - x.joining_year    #Performing Computation at new attribute

  def retention(y):                       #Adding Method to child class
    print("Dear", y.f, y.l, ", Your retention period at MMMUT is: ", y.diff, "yrs.")

o = Faculty("R K", "Dwivedi", 2009)
o.printname()
print("Joining Year: ", o.joining_year)
print("Current Year: ", o.current_year)
print()
o.retention()
```

```
R K Dwivedi
Joining Year:   2009
Current Year:   2020

Dear R K Dwivedi , Your retention period at MMMUT is:  11 yrs.
```

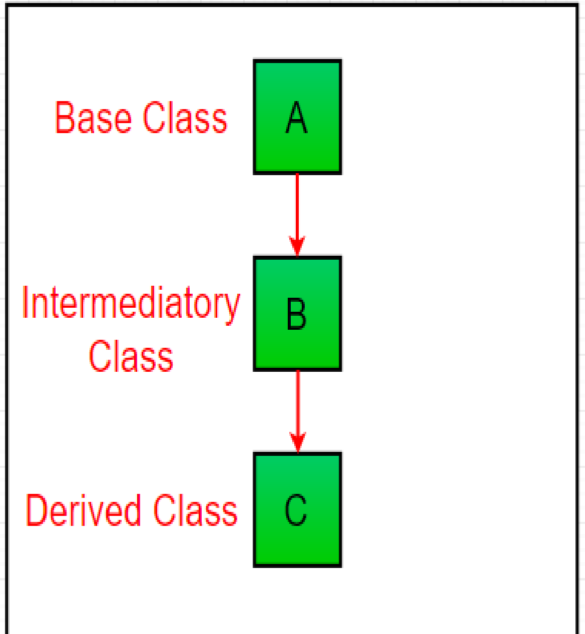
4. Inheritance

…continued

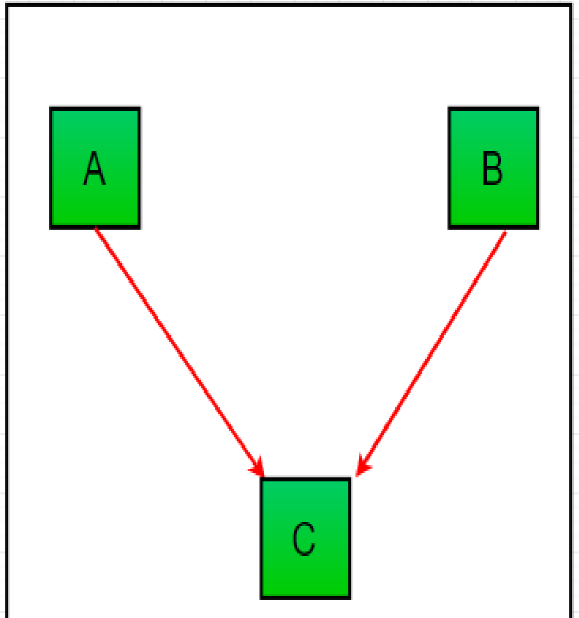
Types of Inheritance (Single, Multilevel, Multiple, Hierarchical, Hybrid)



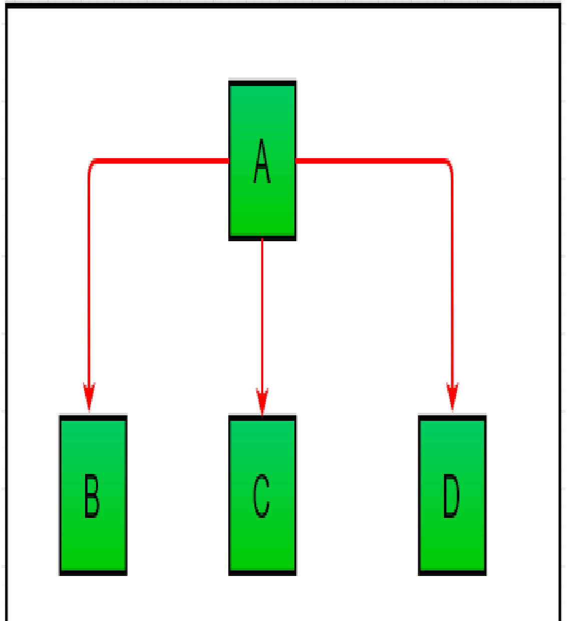
Single Inheritance



Multilevel Inheritance



Multiple Inheritance



Hierarchical Inheritance

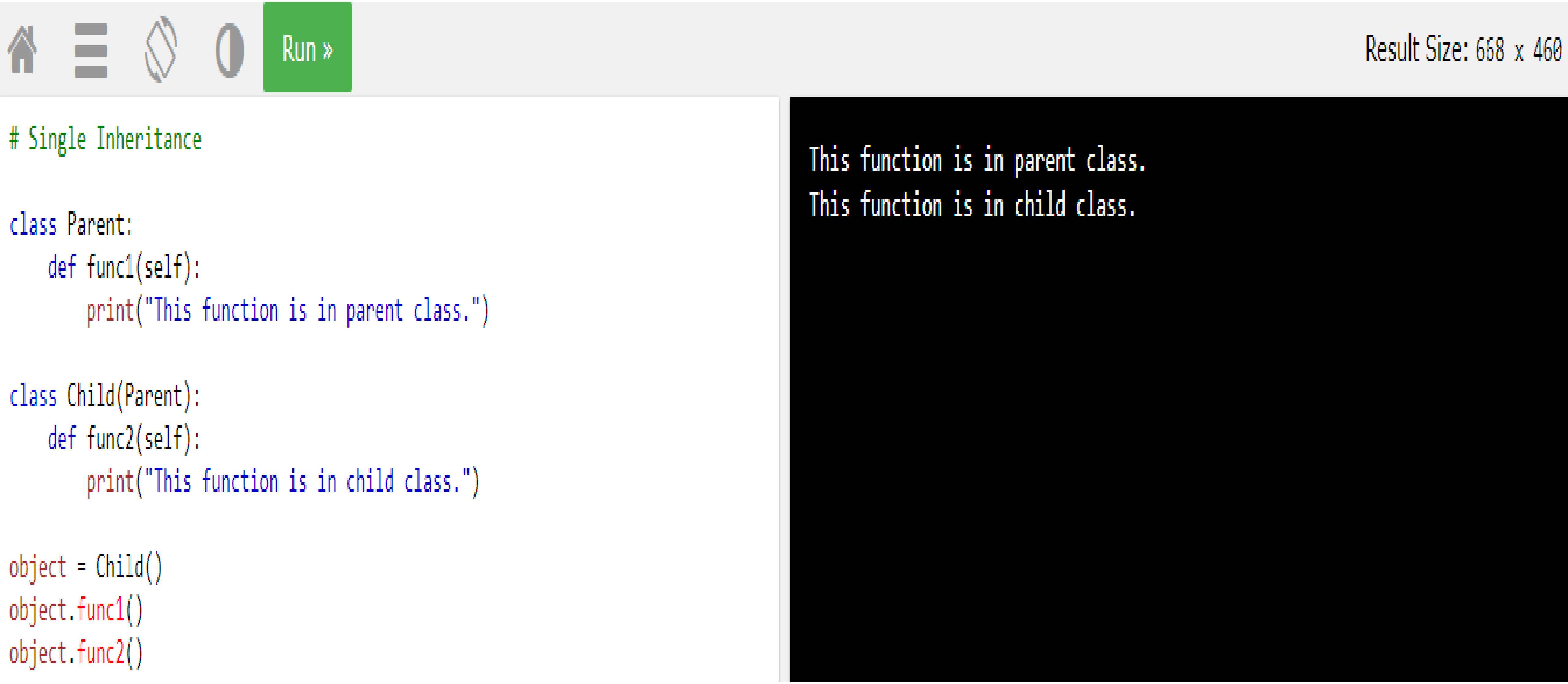
Properties of Inheritance:

- Reusability of Code: We don’t have to write the same code again and again. It also allows us to add more features to a class without modifying it.
- Transitivity: If class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

4. Inheritance

A. Single Inheritance

…continued



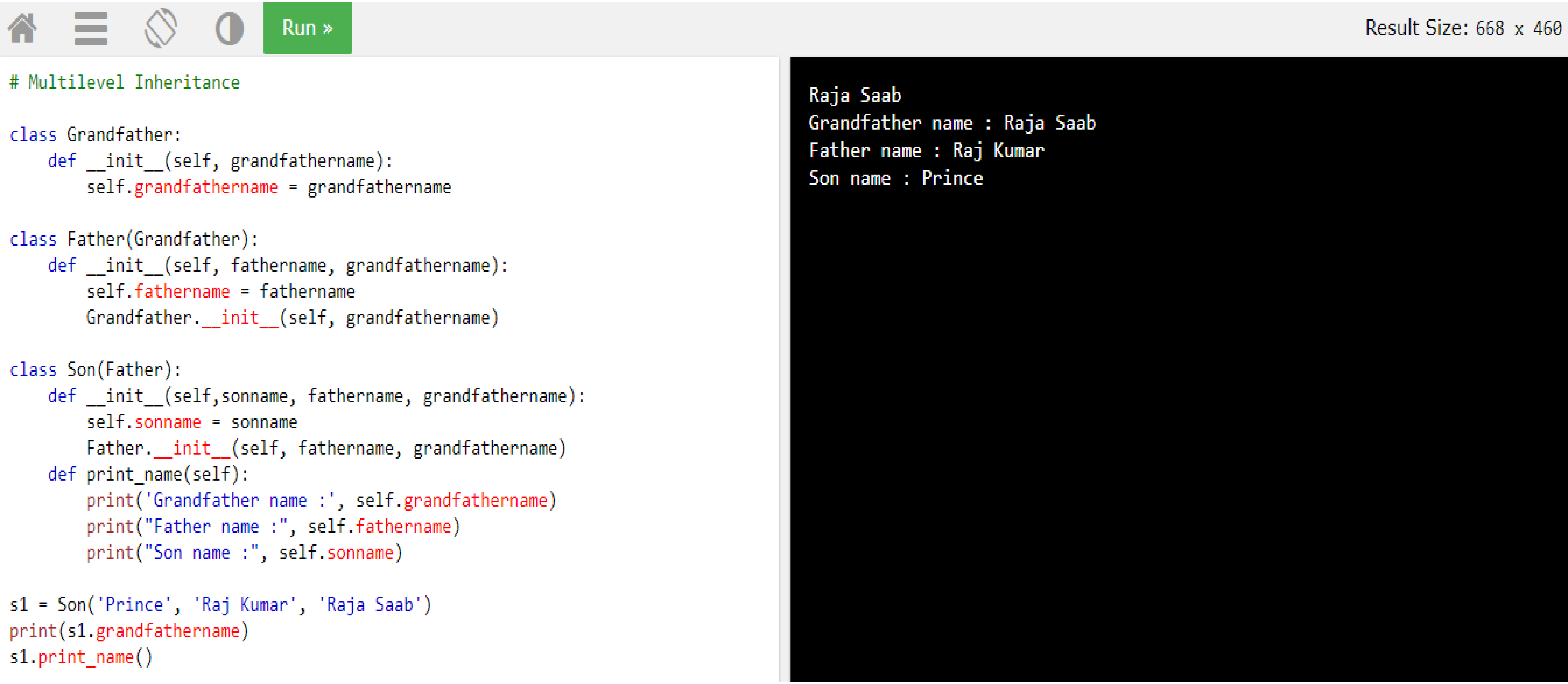
```python
# Single Inheritance

class Parent:
    def func1(self):
        print("This function is in parent class.")

class Child(Parent):
    def func2(self):
        print("This function is in child class.")

object = Child()
object.func1()
object.func2()
```

```
This function is in parent class.
This function is in child class.
```

4. Inheritance

…continued

B. Multilevel Inheritance



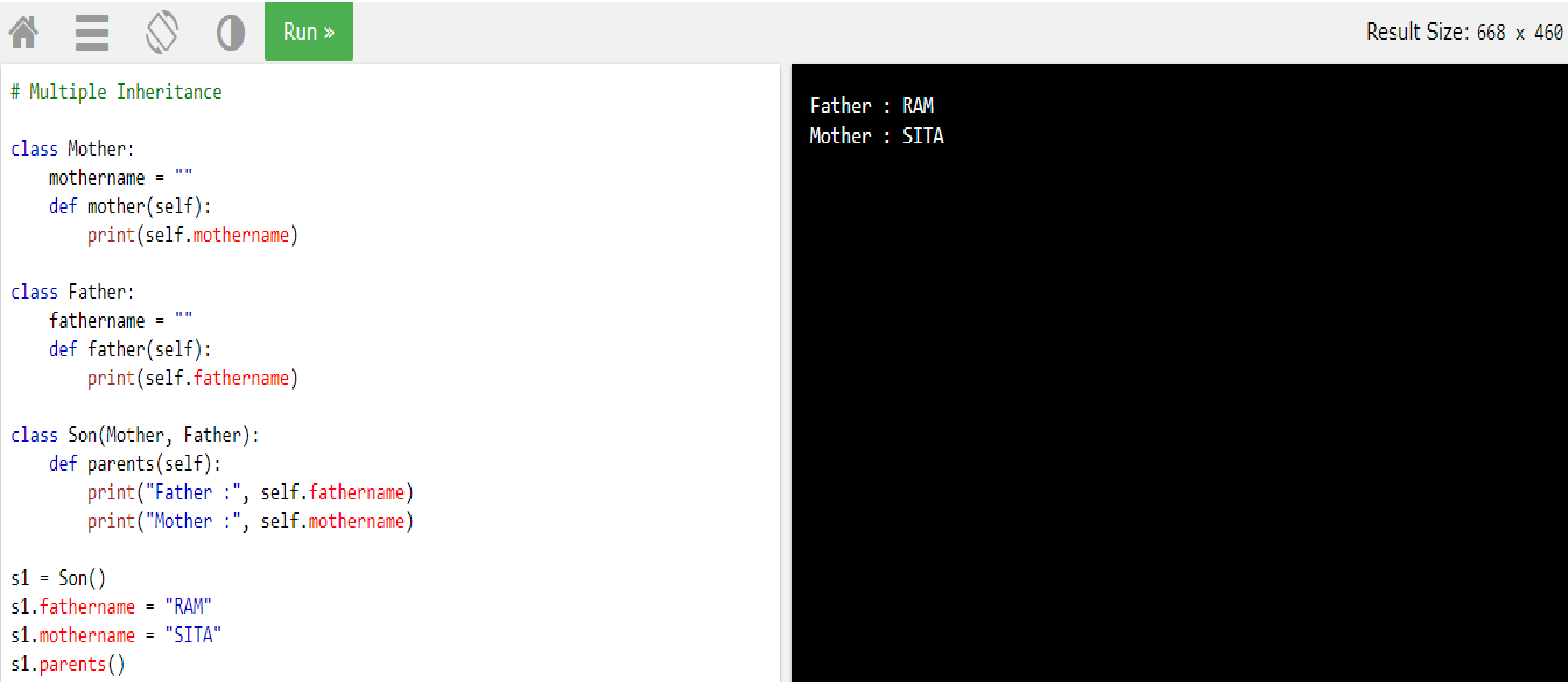
```python
# Multilevel Inheritance

class Grandfather:
    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername
        Grandfather.__init__(self, grandfathername)

class Son(Father):
    def __init__(self,sonname, fathername, grandfathername):
        self.sonname = sonname
        Father.__init__(self, fathername, grandfathername)
    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)

s1 = Son('Prince', 'Raj Kumar', 'Raja Saab')
print(s1.grandfathername)
s1.print_name()
```

```
Raja Saab
Grandfather name : Raja Saab
Father name : Raj Kumar
Son name : Prince
```

4. Inheritance

C. Multiple Inheritance

…continued



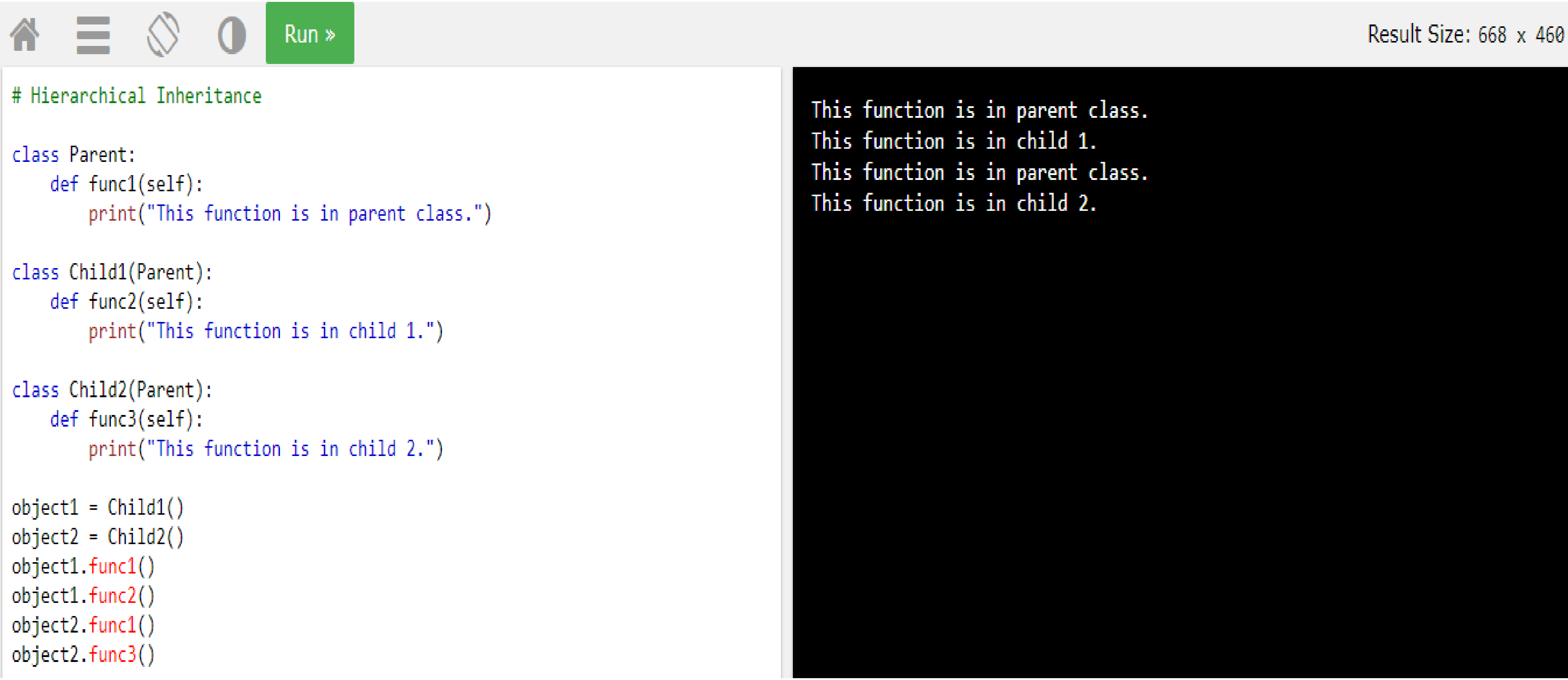
```python
# Multiple Inheritance

class Mother:
    mothername = ""
    def mother(self):
        print(self.mothername)

class Father:
    fathername = ""
    def father(self):
        print(self.fathername)

class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

```
Father : RAM
Mother : SITA
```

4. Inheritance

...continued

D. Hierarchical Inheritance



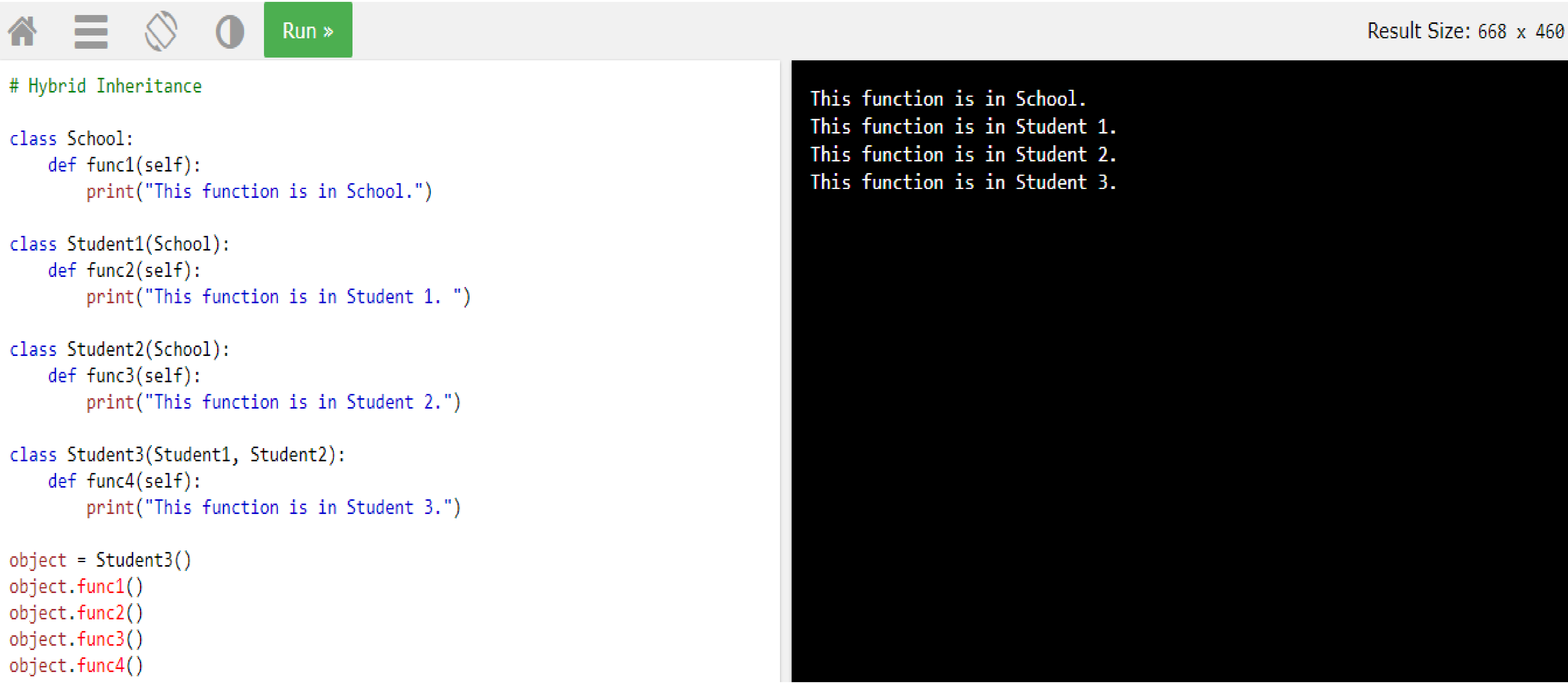
```python
# Hierarchical Inheritance

class Parent:
    def func1(self):
        print("This function is in parent class.")

class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")

object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

4. Inheritance

...continued

E. Hybrid Inheritance: Hierarchical + Multiple



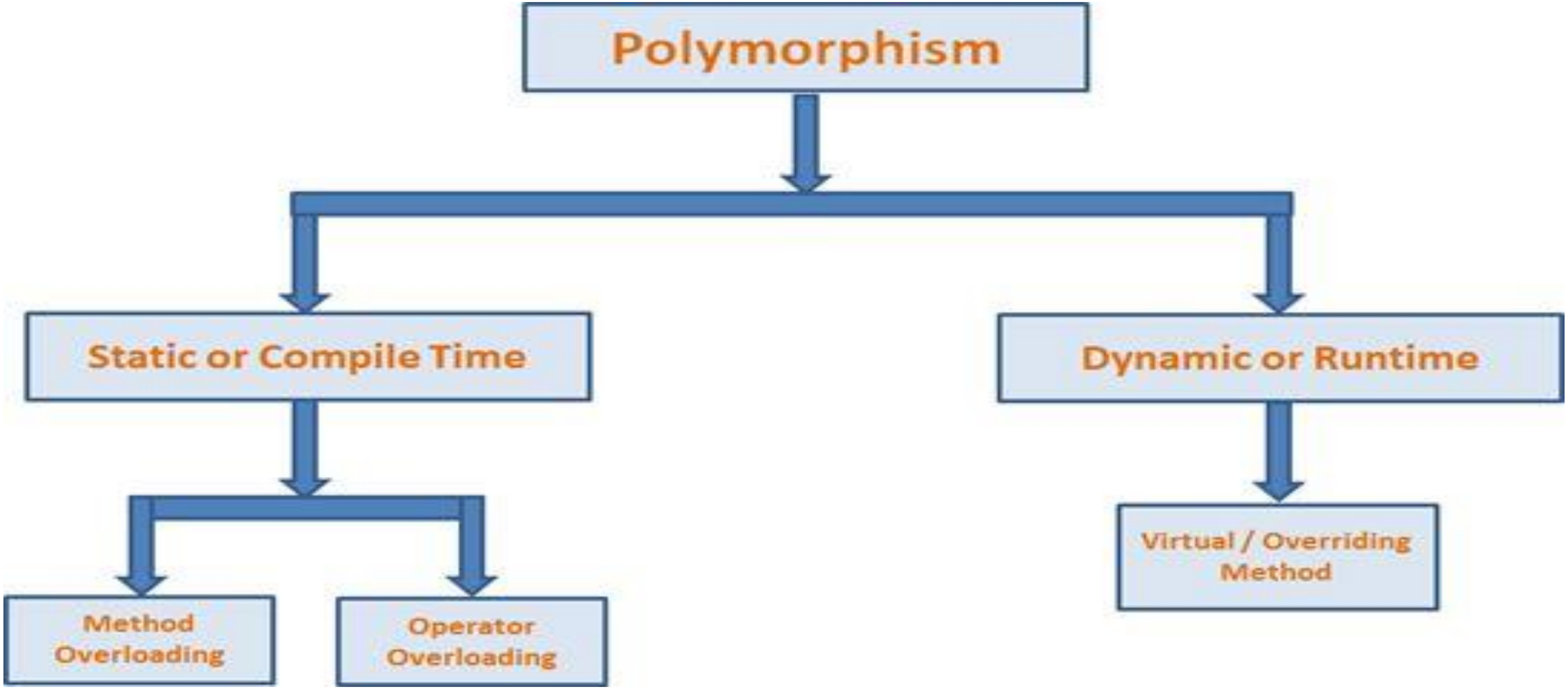
```python
# Hybrid Inheritance

class School:
    def func1(self):
        print("This function is in School.")

class Student1(School):
    def func2(self):
        print("This function is in Student 1. ")

class Student2(School):
    def func3(self):
        print("This function is in Student 2.")

class Student3(Student1, Student2):
    def func4(self):
        print("This function is in Student 3.")

object = Student3()
object.func1()
object.func2()
object.func3()
object.func4()
```

```
This function is in School.
This function is in Student 1.
This function is in Student 2.
This function is in Student 3.
```

5. Polymorphism

5. Polymorphism

Types of Polymorphism



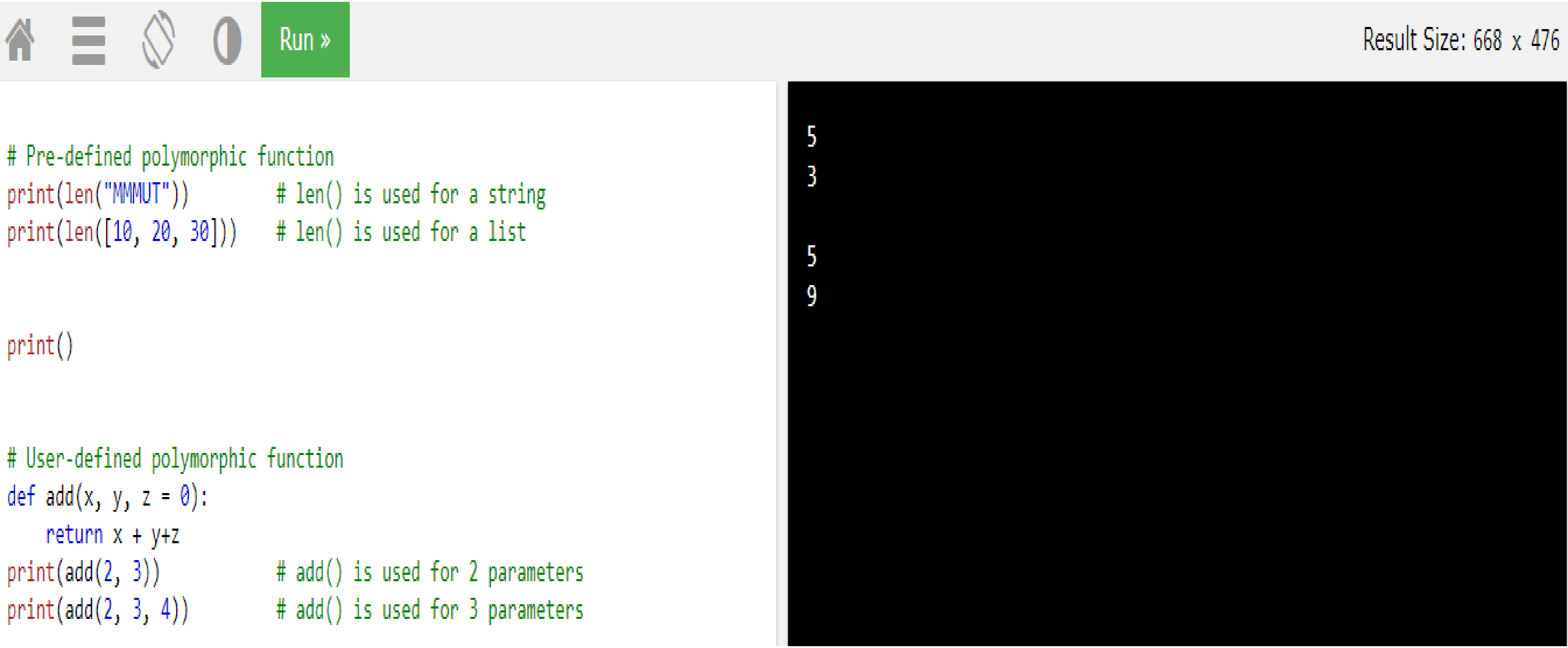
A. Method Overloading

5. Polymorphism

…continued

Method Overloading

Method overloading means same function name being used differently (different signatures).



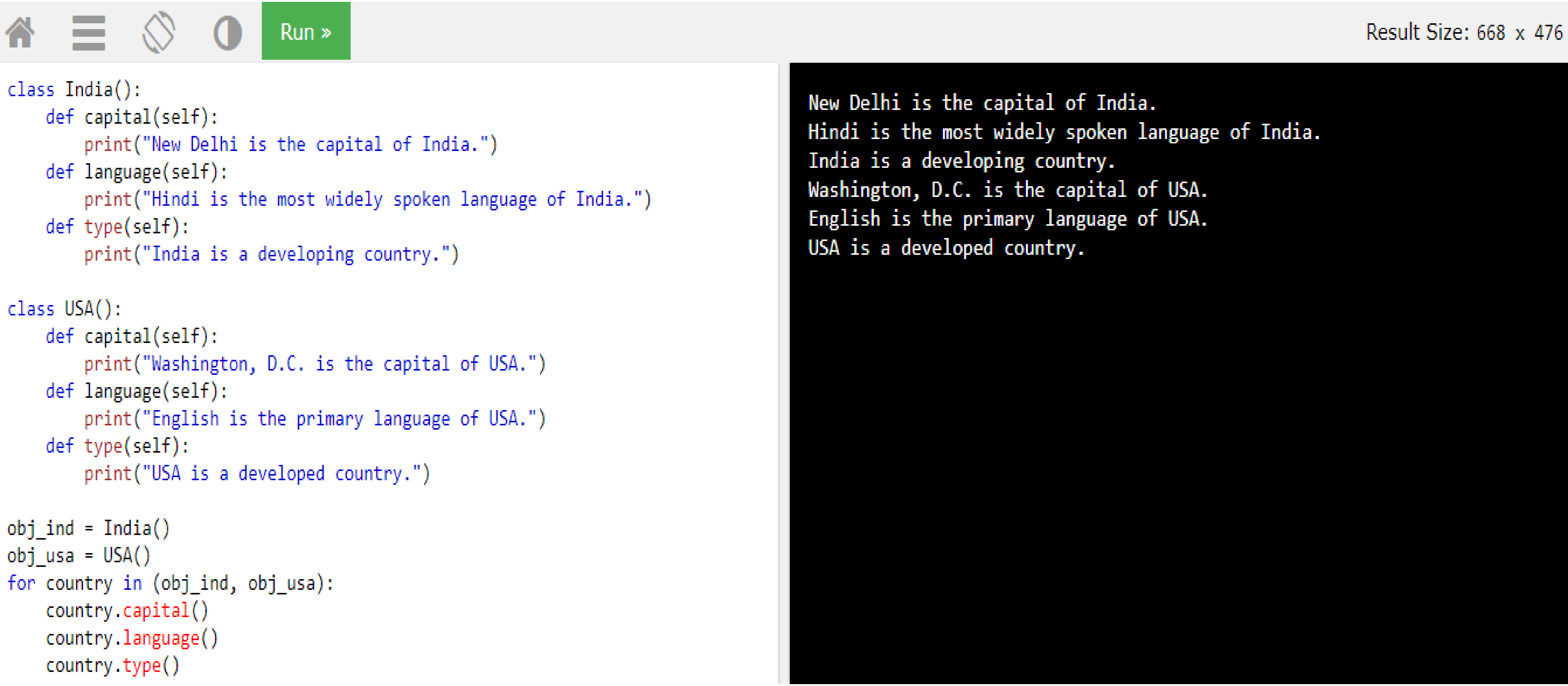
```
# Pre-defined polymorphic function
print(len("MMMUT"))        # len() is used for a string
print(len([10, 20, 30]))   # len() is used for a list


print()


# User-defined polymorphic function
def add(x, y, z = 0):
    return x + y+z
print(add(2, 3))           # add() is used for 2 parameters
print(add(2, 3, 4))        # add() is used for 3 parameters
```

```
5
3

5
9
```

5. Polymorphism

...continued

Polymorphism with class methods (use of loop)



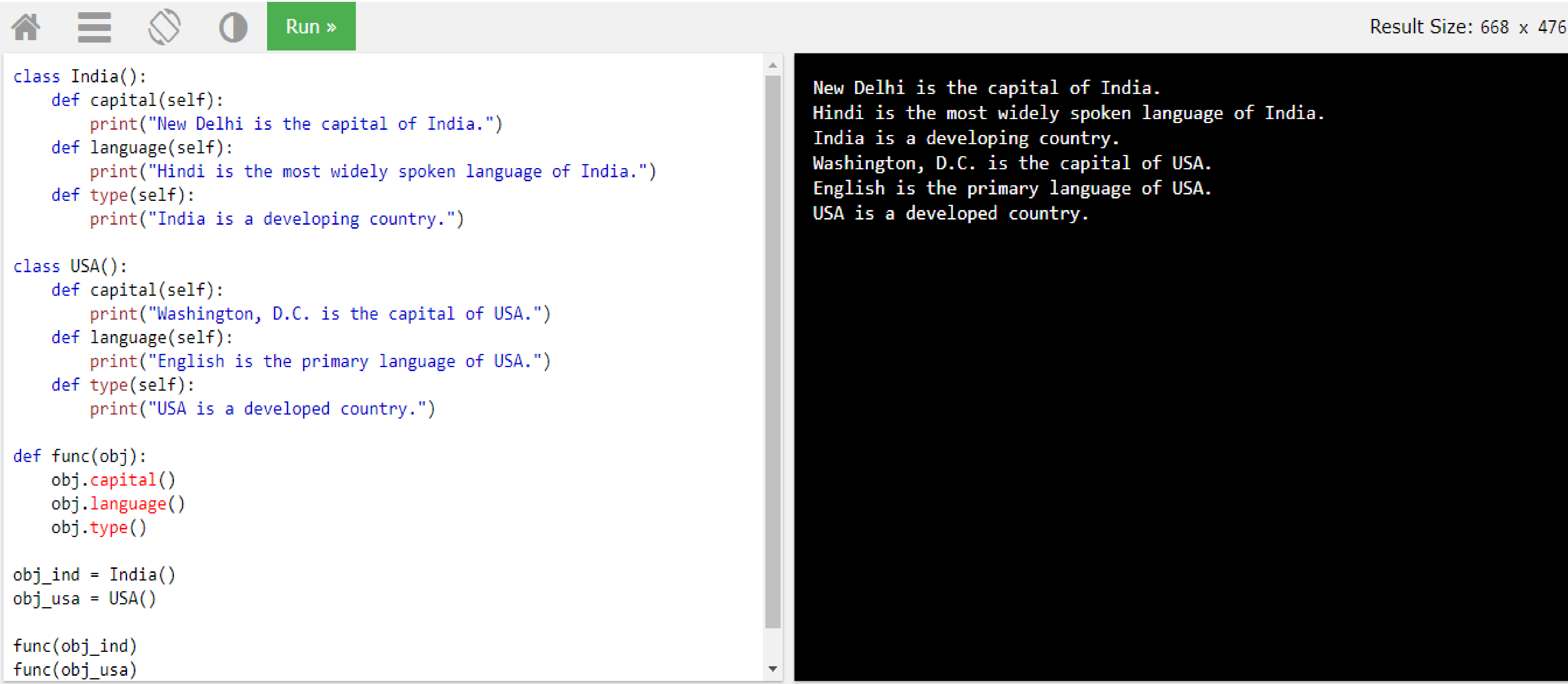
```python
class India():
    def capital(self):
        print("New Delhi is the capital of India.")
    def language(self):
        print("Hindi is the most widely spoken language of India.")
    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")
    def language(self):
        print("English is the primary language of USA.")
    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

5. Polymorphism

…continued

Polymorphism with class methods (use of Object as Function Argument)



```python
class India():
    def capital(self):
        print("New Delhi is the capital of India.")
    def language(self):
        print("Hindi is the most widely spoken language of India.")
    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")
    def language(self):
        print("English is the primary language of USA.")
    def type(self):
        print("USA is a developed country.")

def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

B. Operator Overloading

5. Polymorphism

…continued

Operator Overloading

- Operator overloading is the process of using an operator in different ways depending on the operands.
- Python has some magic methods or special functions for operator overloading

UNARY OPERATORS	MAGIC METHOD
–	__NEG__(SELF, OTHER)
+	__POS__(SELF, OTHER)
~	__INVERT__(SELF, OTHER)

BINARY OPERATORS	MAGIC METHOD
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

5. Polymorphism

Operator Overloading

…continued

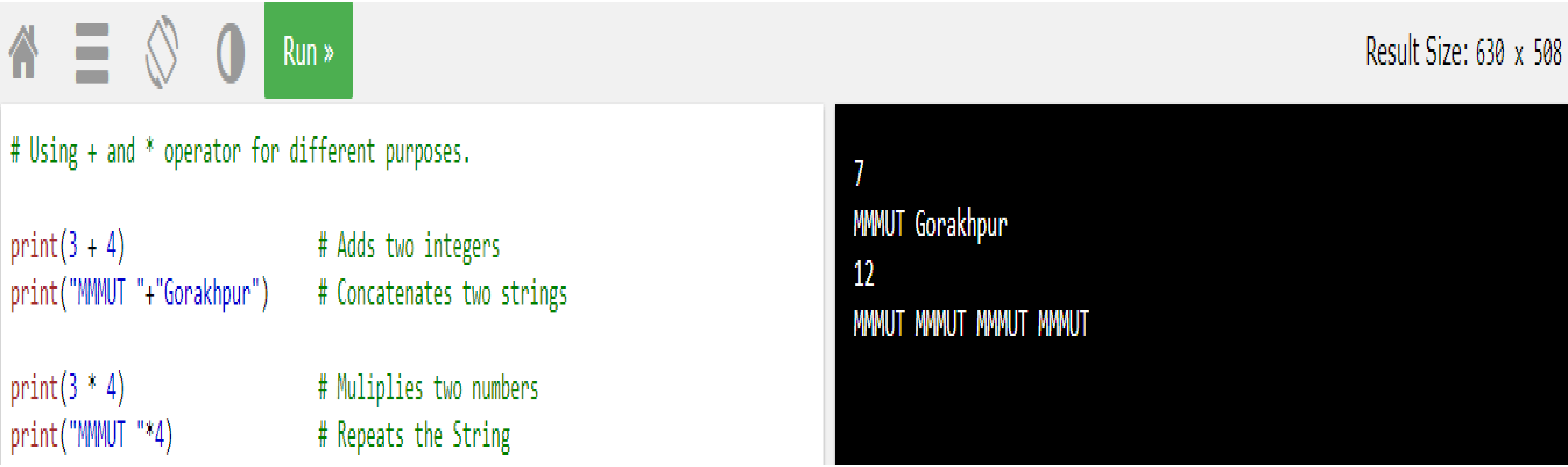
COMPARISON OPERATORS	MAGIC METHOD
<	__LT__(SELF, OTHER)
>	__GT__(SELF, OTHER)
<=	__LE__(SELF, OTHER)
>=	__GE__(SELF, OTHER)
==	__EQ__(SELF, OTHER)
!=	__NE__(SELF, OTHER)

ASSIGNMENT OPERATORS	MAGIC METHOD
-=	__ISUB__(SELF, OTHER)
+=	__IADD__(SELF, OTHER)
*=	__IMUL__(SELF, OTHER)
/=	__IDIV__(SELF, OTHER)
//=	__IFLOORDIV__(SELF, OTHER)
%=	__IMOD__(SELF, OTHER)
**=	__IPOW__(SELF, OTHER)
>>=	__IRSHIFT__(SELF, OTHER)
<<=	__ILSHIFT__(SELF, OTHER)
&=	__IAND__(SELF, OTHER)
|=	__IOR__(SELF, OTHER)
^=	__IXOR__(SELF, OTHER)

5. Polymorphism

Operator Overloading

...continued



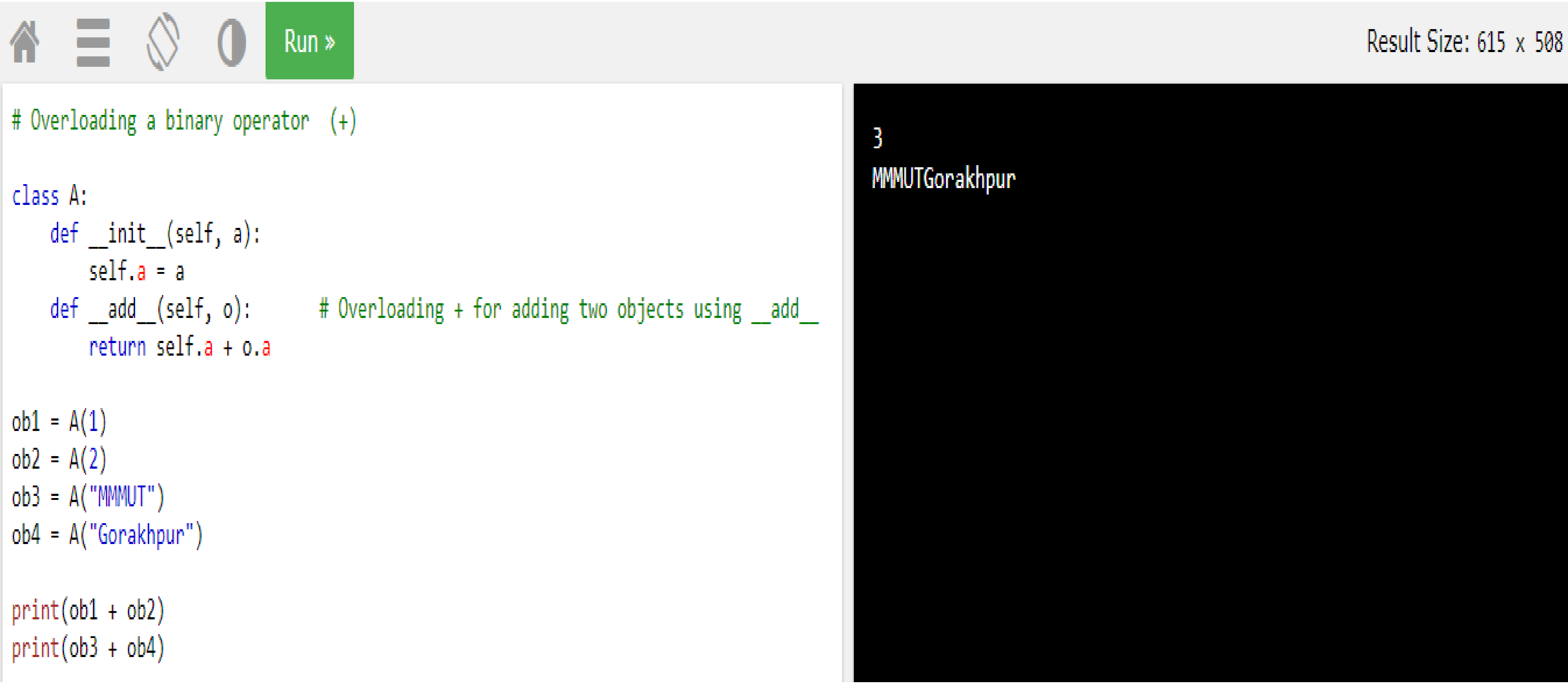
```
# Using + and * operator for different purposes.

print(3 + 4)                # Adds two integers
print("MMMUT "+"Gorakhpur")    # Concatenates two strings

print(3 * 4)                # Muliplies two numbers
print("MMMUT "*4)            # Repeats the String
```

```
7
MMMUT Gorakhpur
12
MMMUT MMMUT MMMUT MMMUT
```

5. Polymorphism

Operator Overloading

...continued



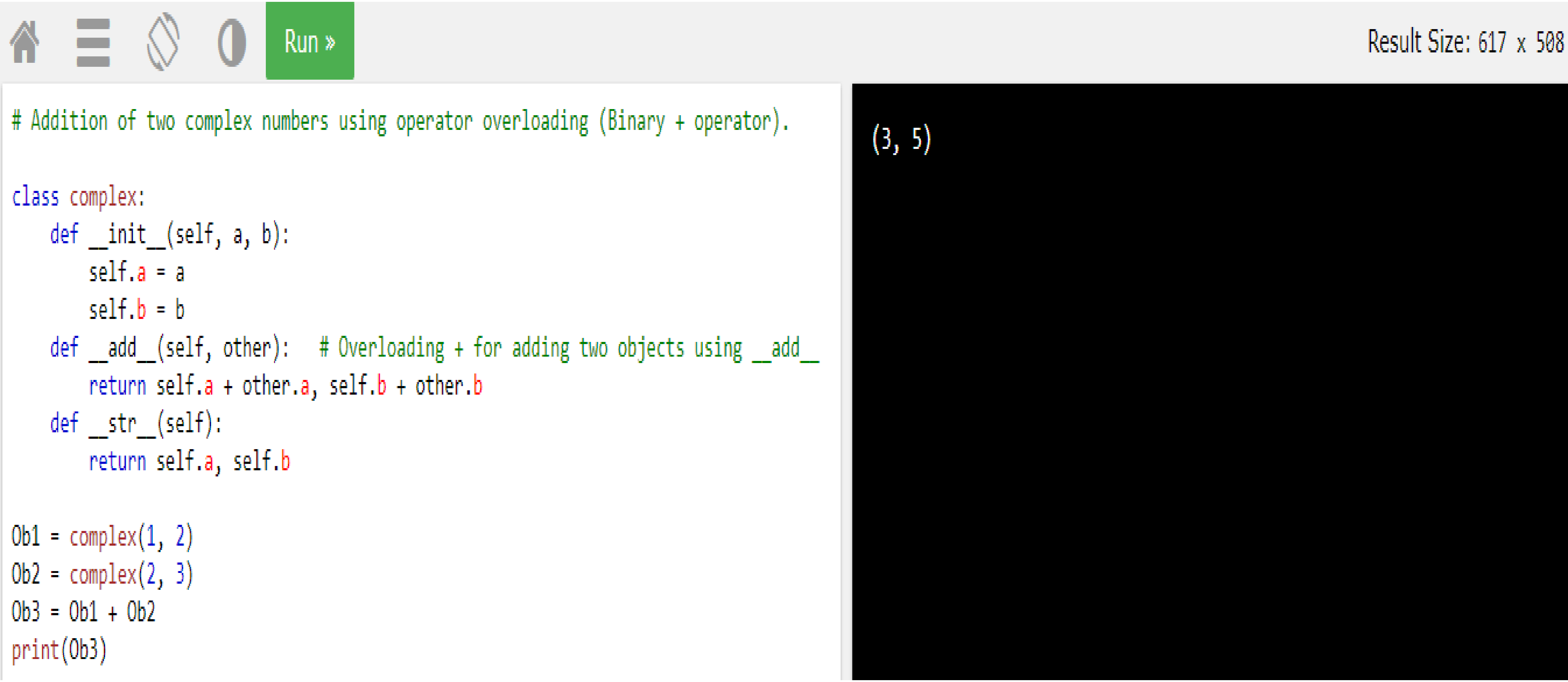
```python
# Overloading a binary operator  (+)

class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, o):       # Overloading + for adding two objects using __add__
        return self.a + o.a

ob1 = A(1)
ob2 = A(2)
ob3 = A("MMMUT")
ob4 = A("Gorakhpur")

print(ob1 + ob2)
print(ob3 + ob4)
```

```
3
MMMUTGorakhpur
```

5. Polymorphism

Operator Overloading

…continued



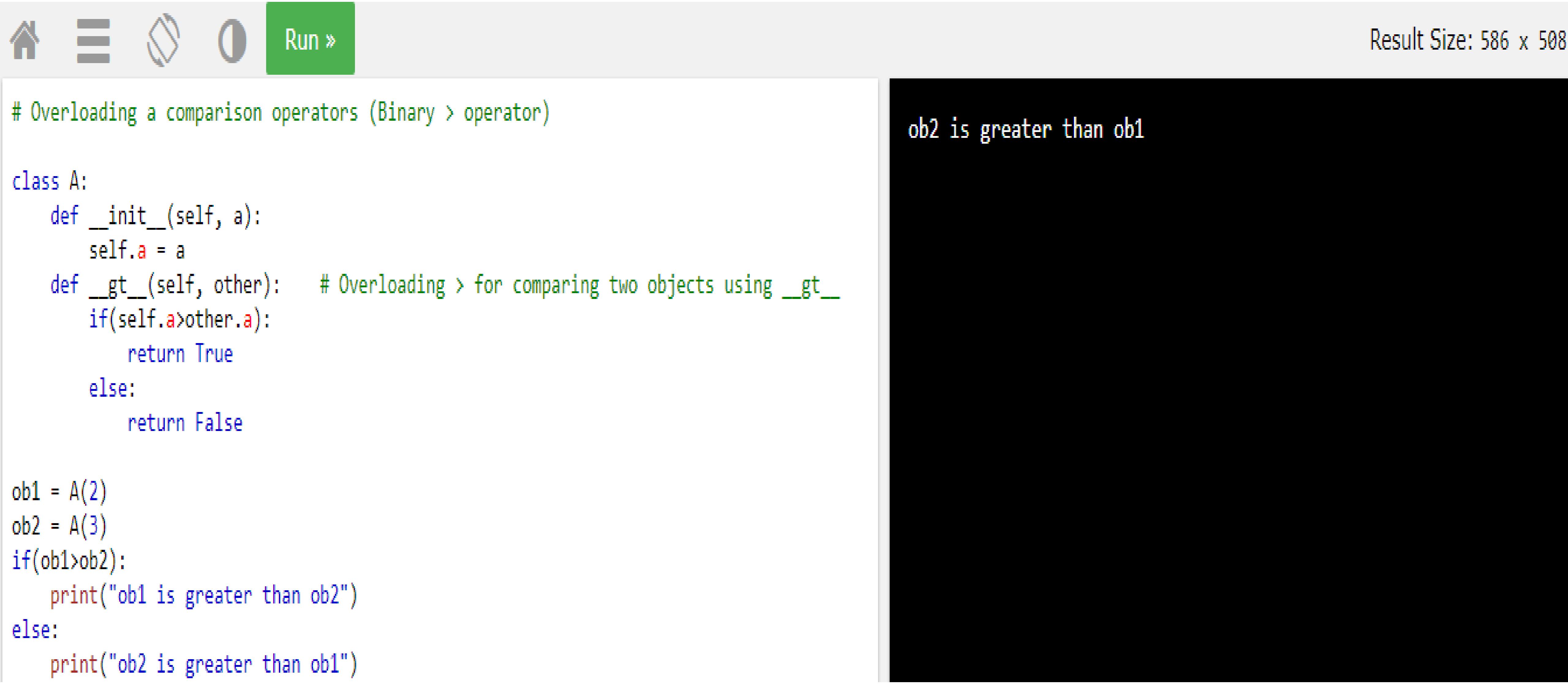
```
# Addition of two complex numbers using operator overloading (Binary + operator).

class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __add__(self, other):   # Overloading + for adding two objects using __add__
        return self.a + other.a, self.b + other.b
    def __str__(self):
        return self.a, self.b

Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)
```

```
(3, 5)
```

5. Polymorphism

Operator Overloading

...continued



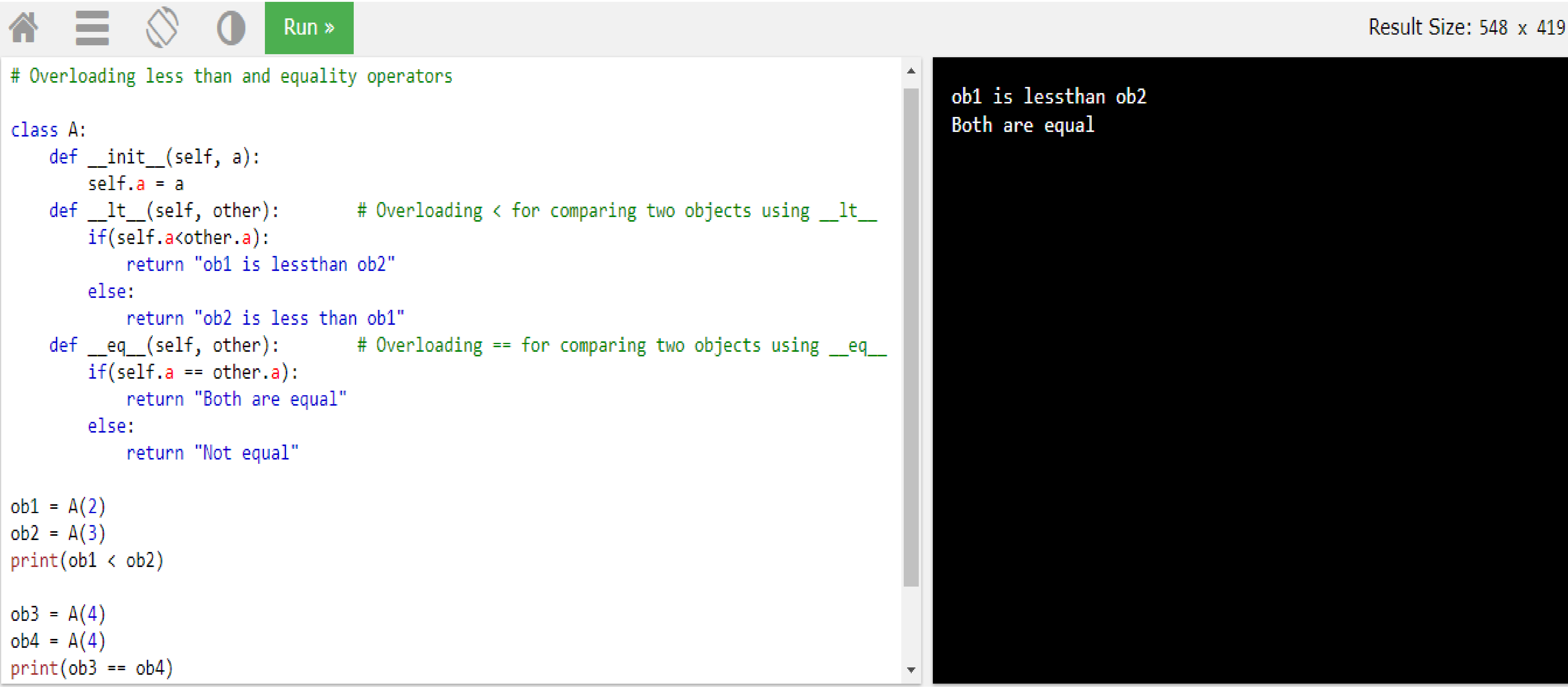
```
# Overloading a comparison operators (Binary > operator)

class A:
    def __init__(self, a):
        self.a = a
    def __gt__(self, other):    # Overloading > for comparing two objects using __gt__
        if(self.a>other.a):
            return True
        else:
            return False

ob1 = A(2)
ob2 = A(3)
if(ob1>ob2):
    print("ob1 is greater than ob2")
else:
    print("ob2 is greater than ob1")
```

```
ob2 is greater than ob1
```

5. Polymorphism

Operator Overloading

...continued



```python
# Overloading less than and equality operators

class A:
    def __init__(self, a):
        self.a = a
    def __lt__(self, other):        # Overloading < for comparing two objects using __lt__
        if(self.a<other.a):
            return "ob1 is lessthan ob2"
        else:
            return "ob2 is less than ob1"
    def __eq__(self, other):        # Overloading == for comparing two objects using __eq__
        if(self.a == other.a):
            return "Both are equal"
        else:
            return "Not equal"

ob1 = A(2)
ob2 = A(3)
print(ob1 < ob2)

ob3 = A(4)
ob4 = A(4)
print(ob3 == ob4)
```

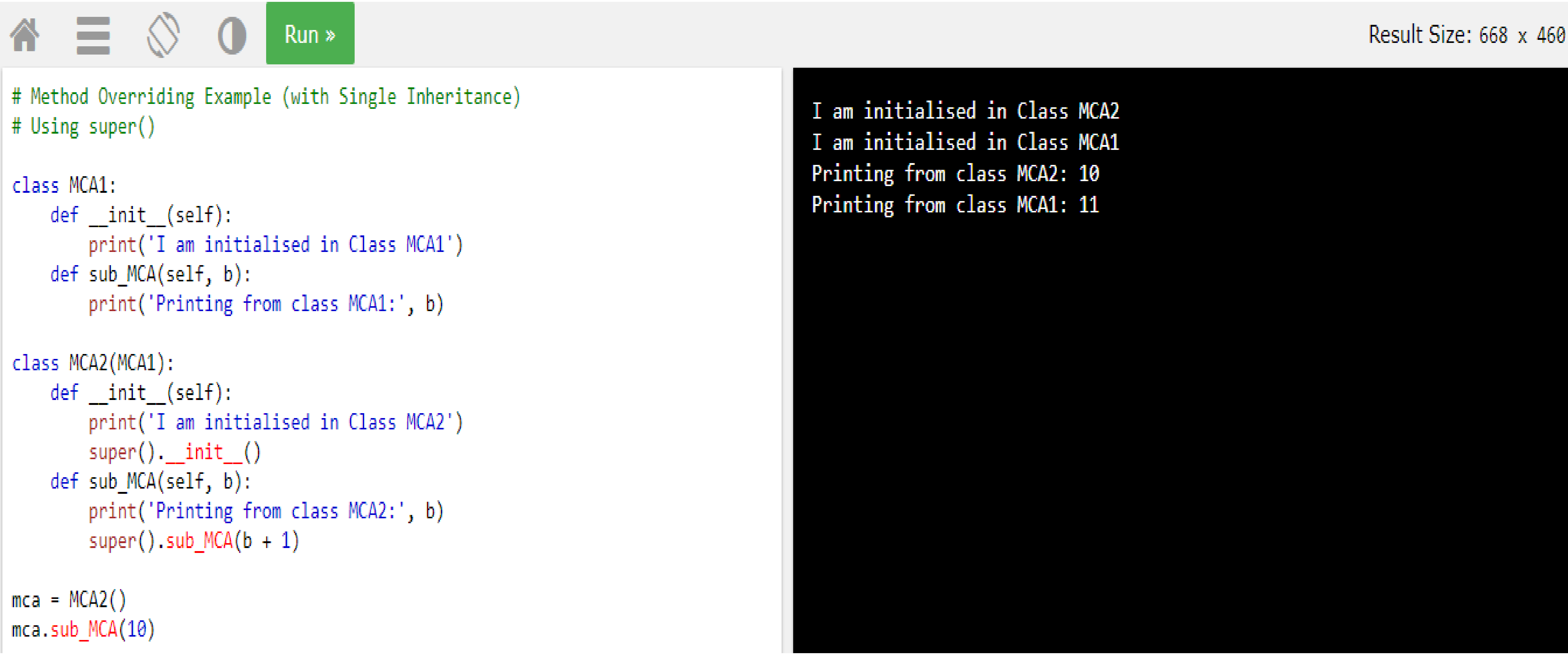
```
ob1 is lessthan ob2
Both are equal
```

C. Method Overriding

5. Polymorphism

...continued

Method Overriding (Polymorphism with Single Inheritance)
(Using super())



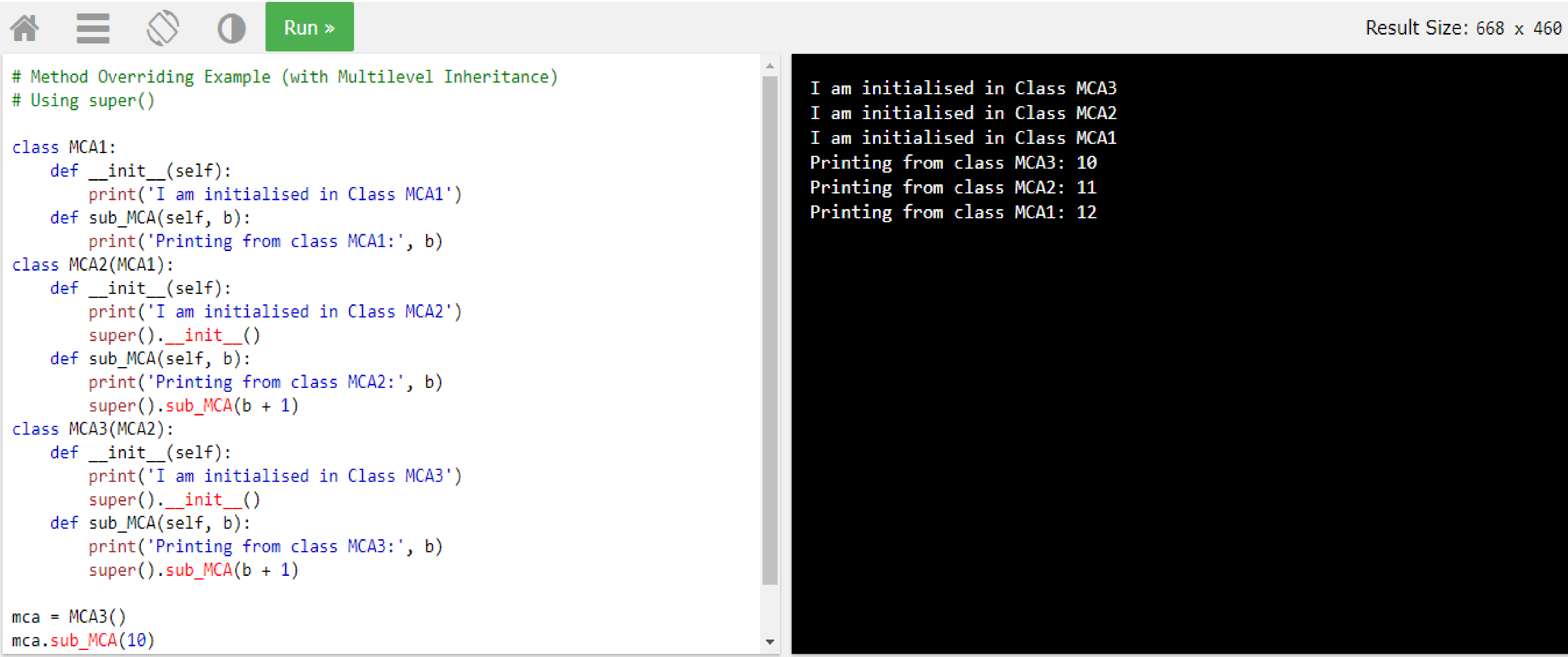
```python
# Method Overriding Example (with Single Inheritance)
# Using super()

class MCA1:
    def __init__(self):
        print('I am initialised in Class MCA1')
    def sub_MCA(self, b):
        print('Printing from class MCA1:', b)

class MCA2(MCA1):
    def __init__(self):
        print('I am initialised in Class MCA2')
        super().__init__()
    def sub_MCA(self, b):
        print('Printing from class MCA2:', b)
        super().sub_MCA(b + 1)

mca = MCA2()
mca.sub_MCA(10)
```

```
I am initialised in Class MCA2
I am initialised in Class MCA1
Printing from class MCA2: 10
Printing from class MCA1: 11
```

5. Polymorphism

…continued

Method Overriding (Polymorphism with Multilevel Inheritance)
(Using super())



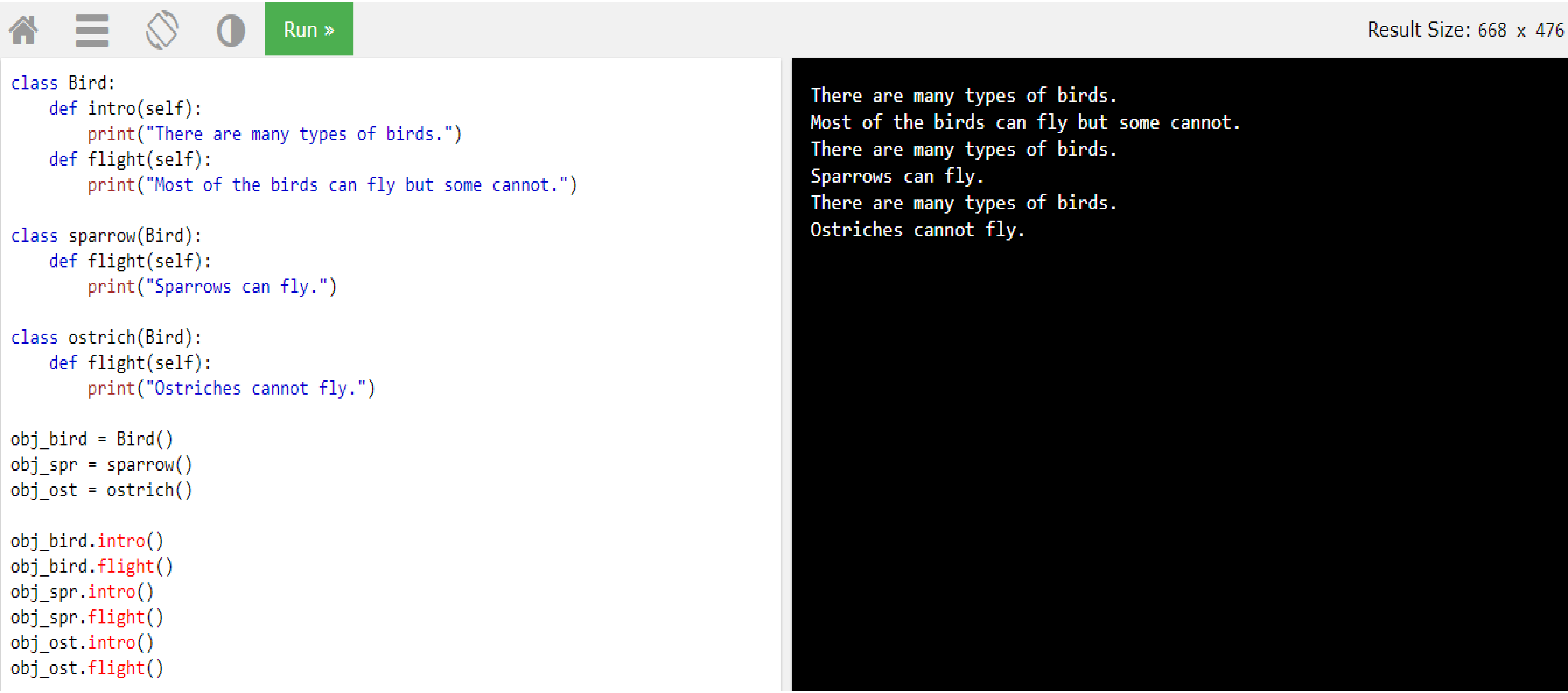
```python
# Method Overriding Example (with Multilevel Inheritance)
# Using super()

class MCA1:
    def __init__(self):
        print('I am initialised in Class MCA1')
    def sub_MCA(self, b):
        print('Printing from class MCA1:', b)
class MCA2(MCA1):
    def __init__(self):
        print('I am initialised in Class MCA2')
        super().__init__()
    def sub_MCA(self, b):
        print('Printing from class MCA2:', b)
        super().sub_MCA(b + 1)
class MCA3(MCA2):
    def __init__(self):
        print('I am initialised in Class MCA3')
        super().__init__()
    def sub_MCA(self, b):
        print('Printing from class MCA3:', b)
        super().sub_MCA(b + 1)

mca = MCA3()
mca.sub_MCA(10)
```

```
I am initialised in Class MCA3
I am initialised in Class MCA2
I am initialised in Class MCA1
Printing from class MCA3: 10
Printing from class MCA2: 11
Printing from class MCA1: 12
```

5. Polymorphism

…continued

Method Overriding (Polymorphism with Hierarchical Inheritance)

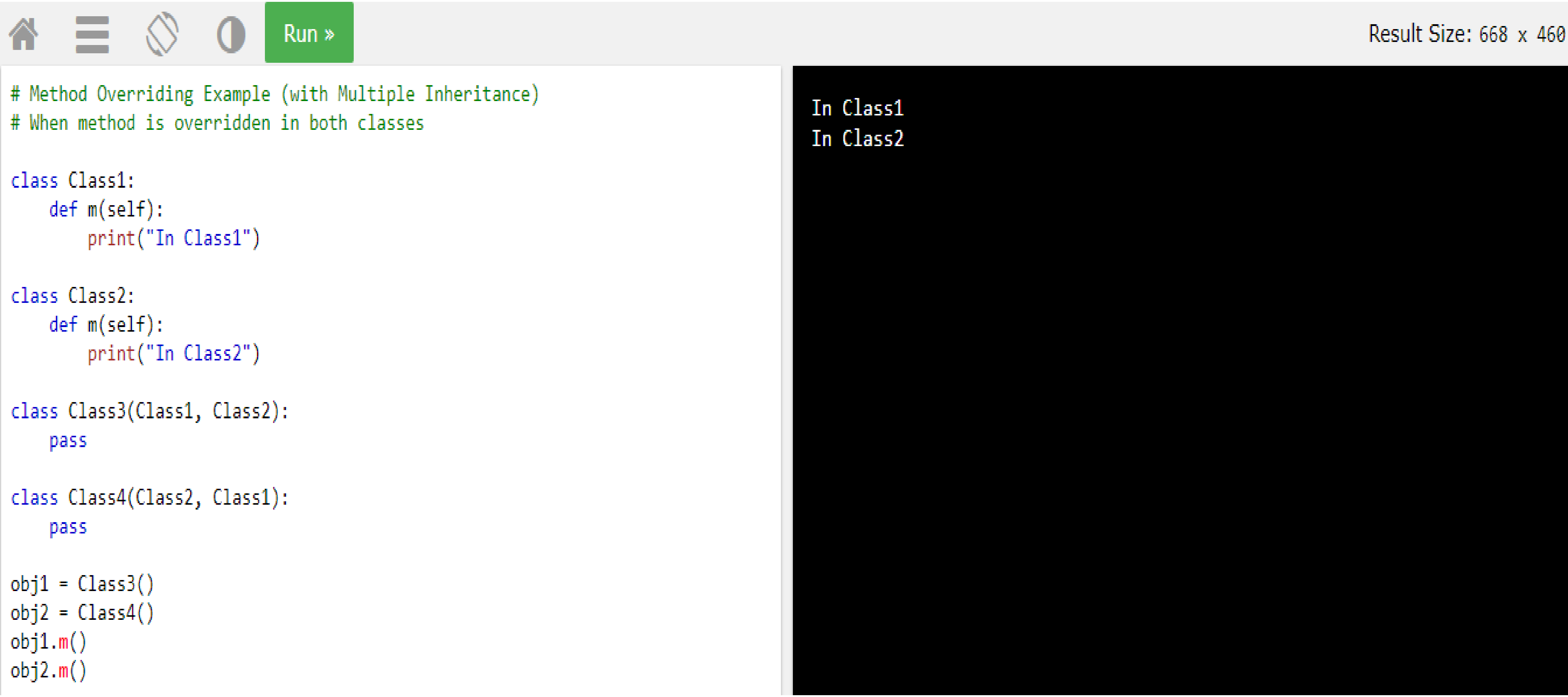


```python
class Bird:
    def intro(self):
        print("There are many types of birds.")
    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()
obj_spr.intro()
obj_spr.flight()
obj_ost.intro()
obj_ost.flight()
```

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

5. Polymorphism ...continued

Method Overriding (Polymorphism with Multiple Inheritance)



```python
# Method Overriding Example (with Multiple Inheritance)
# When method is overridden in both classes

class Class1:
    def m(self):
        print("In Class1")

class Class2:
    def m(self):
        print("In Class2")

class Class3(Class1, Class2):
    pass

class Class4(Class2, Class1):
    pass

obj1 = Class3()
obj2 = Class4()
obj1.m()
obj2.m()
```

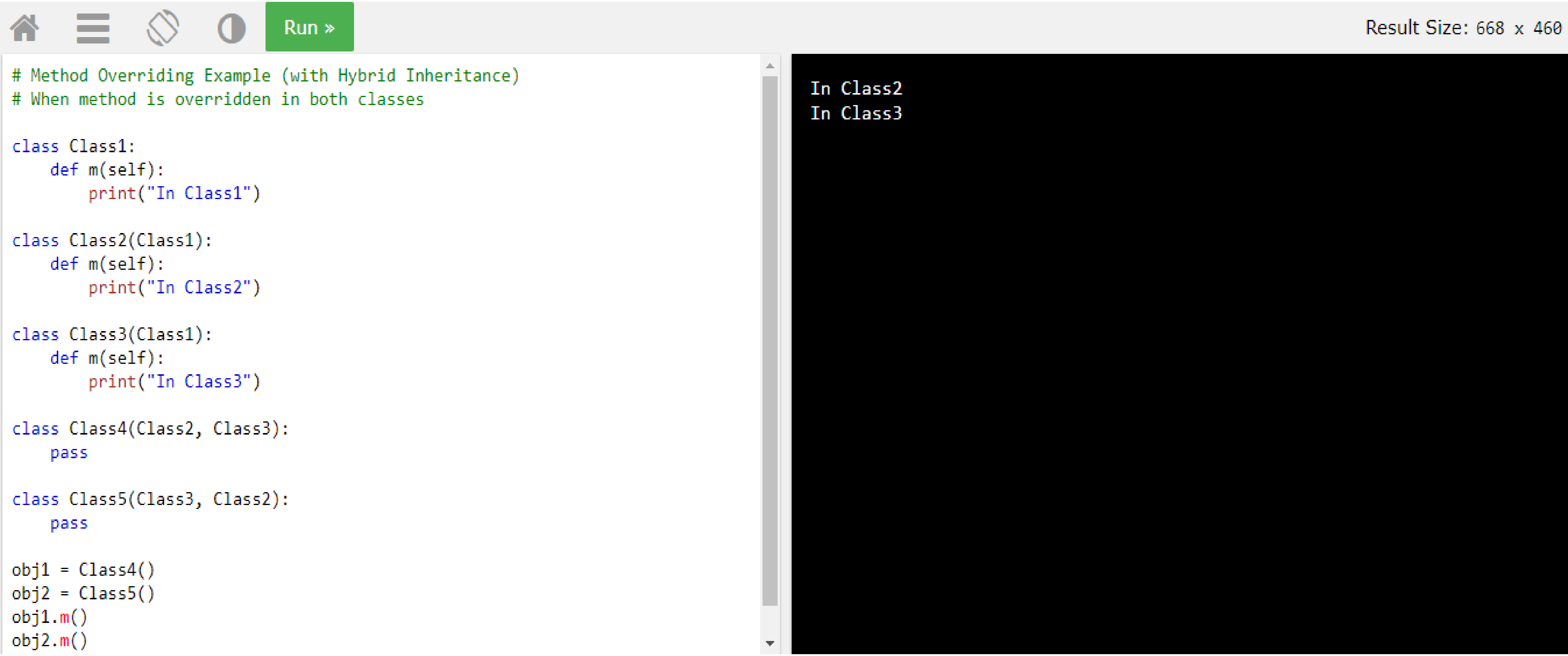
```
In Class1
In Class2
```

5. Polymorphism

…continued

Method Overriding (Polymorphism with Hybrid Inheritance: Hierarchical + Multiple)

(When method is overridden in both classes Class2 and Class3)



```python
# Method Overriding Example (with Hybrid Inheritance)
# When method is overridden in both classes

class Class1:
    def m(self):
        print("In Class1")

class Class2(Class1):
    def m(self):
        print("In Class2")

class Class3(Class1):
    def m(self):
        print("In Class3")

class Class4(Class2, Class3):
    pass

class Class5(Class3, Class2):
    pass

obj1 = Class4()
obj2 = Class5()
obj1.m()
obj2.m()
```

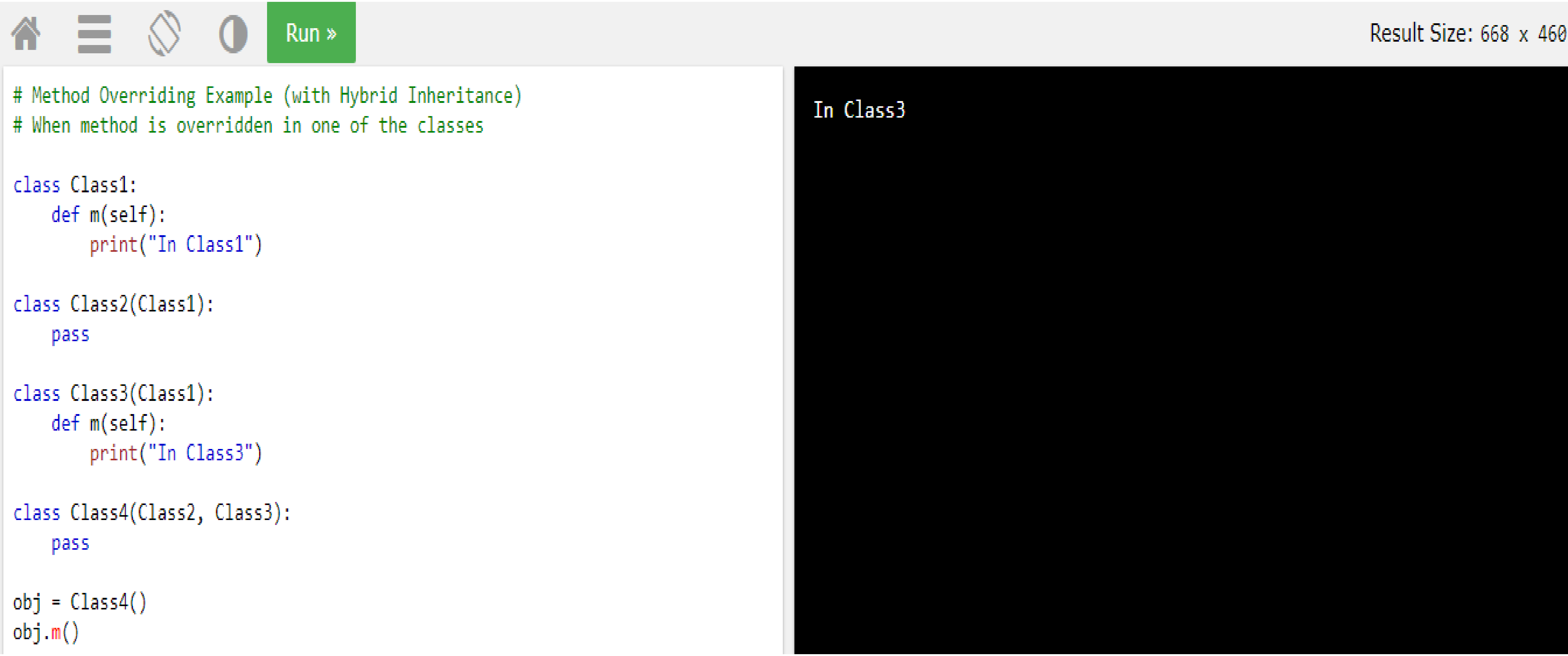
```
In Class2
In Class3
```

5. Polymorphism

…continued

Method Overriding (Polymorphism with Hybrid Inheritance : Hierarchical + Multiple)

(When method is overridden in one of the classes: Class3)



```python
# Method Overriding Example (with Hybrid Inheritance)
# When method is overridden in one of the classes

class Class1:
    def m(self):
        print("In Class1")

class Class2(Class1):
    pass

class Class3(Class1):
    def m(self):
        print("In Class3")

class Class4(Class2, Class3):
    pass

obj = Class4()
obj.m()
```

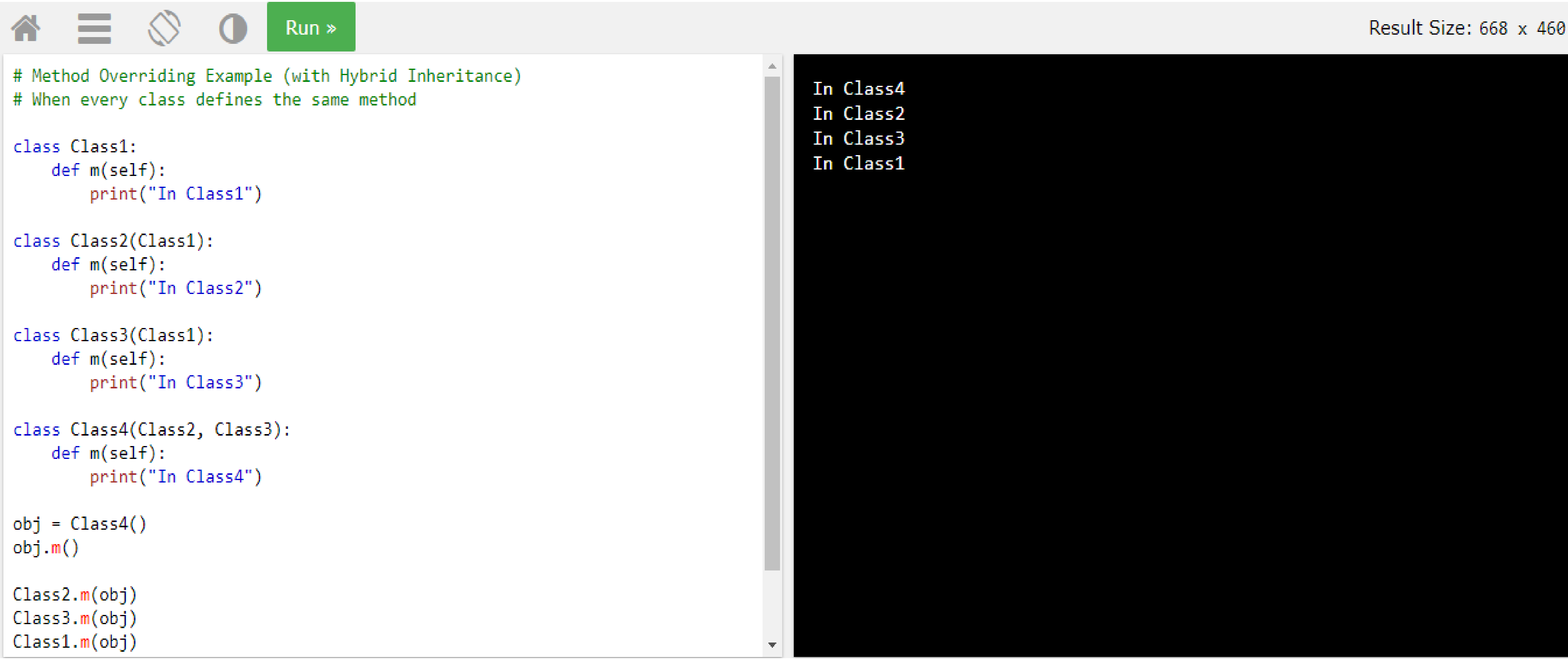
```
In Class3
```

5. Polymorphism

…continued

Method Overriding (Polymorphism with Hybrid Inheritance : Hierarchical + Multiple)

(When every class defines the same method)



```
# Method Overriding Example (with Hybrid Inheritance)
# When every class defines the same method

class Class1:
    def m(self):
        print("In Class1")

class Class2(Class1):
    def m(self):
        print("In Class2")

class Class3(Class1):
    def m(self):
        print("In Class3")

class Class4(Class2, Class3):
    def m(self):
        print("In Class4")

obj = Class4()
obj.m()

Class2.m(obj)
Class3.m(obj)
Class1.m(obj)
```

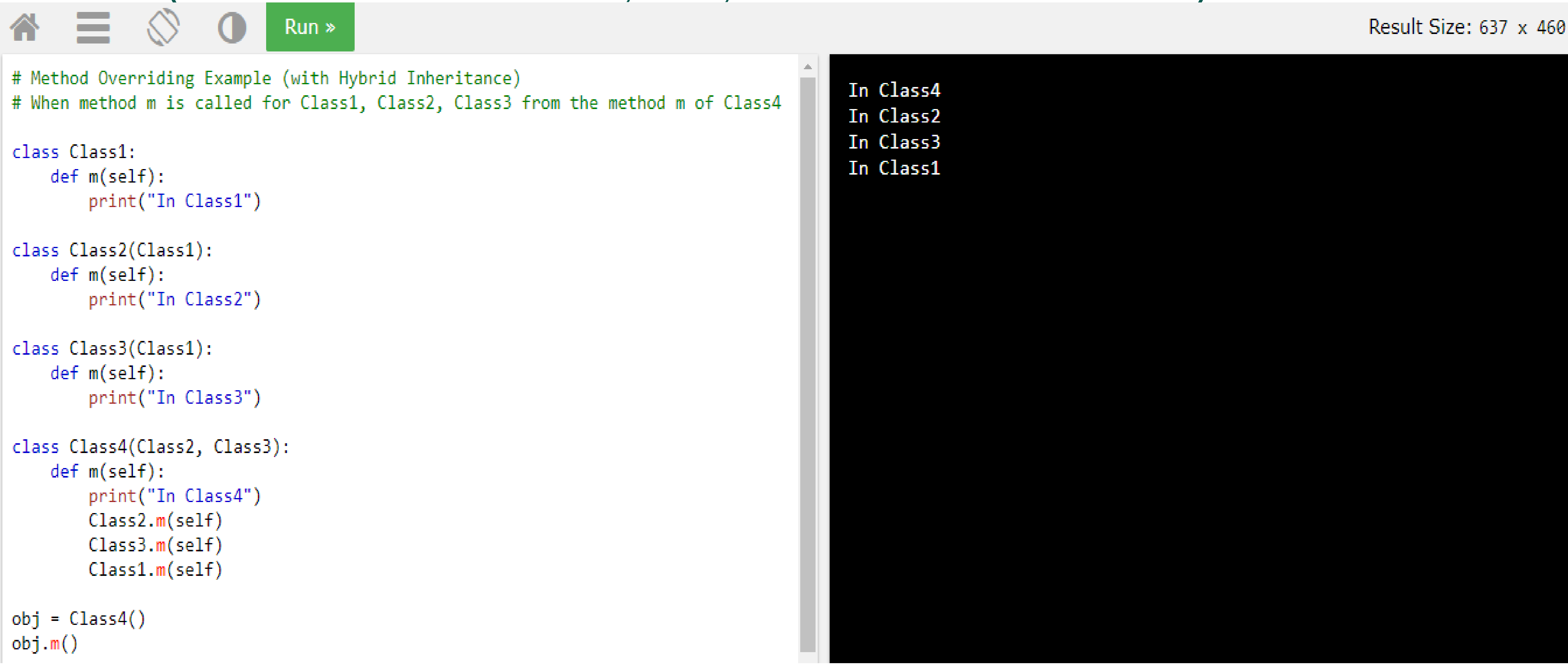
```
In Class4
In Class2
In Class3
In Class1
```

5. Polymorphism

…continued

Method Overriding (Polymorphism with Hybrid Inheritance : Hierarchical + Multiple)

(When method m is called for Class1, Class2, Class3 from the method m of Class4)



```python
# Method Overriding Example (with Hybrid Inheritance)
# When method m is called for Class1, Class2, Class3 from the method m of Class4

class Class1:
    def m(self):
        print("In Class1")

class Class2(Class1):
    def m(self):
        print("In Class2")

class Class3(Class1):
    def m(self):
        print("In Class3")

class Class4(Class2, Class3):
    def m(self):
        print("In Class4")
        Class2.m(self)
        Class3.m(self)
        Class1.m(self)

obj = Class4()
obj.m()
```

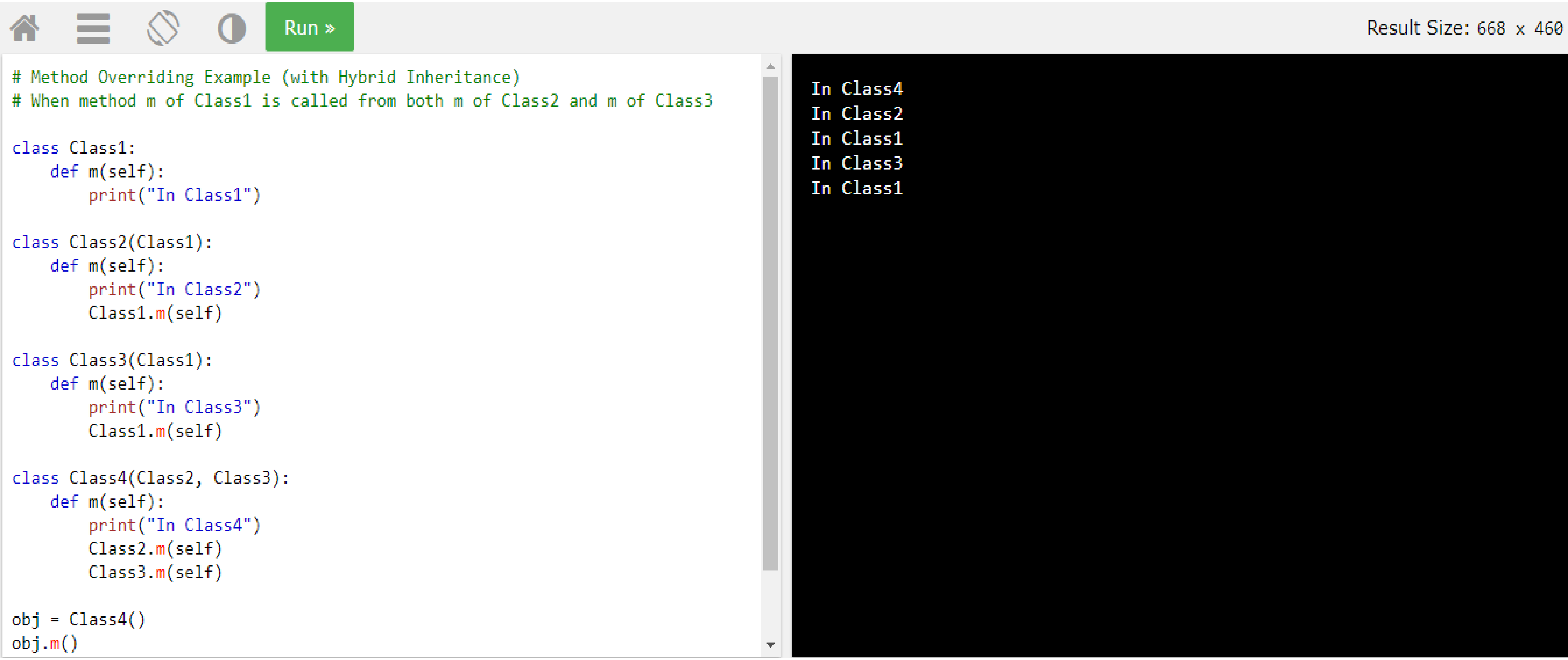
```
In Class4
In Class2
In Class3
In Class1
```

5. Polymorphism

…continued

Method Overriding (Polymorphism with Hybrid Inheritance : Hierarchical + Multiple)

(When method m of Class1 is called from m of Class2 and m of Class3 both)



```python
# Method Overriding Example (with Hybrid Inheritance)
# When method m of Class1 is called from both m of Class2 and m of Class3

class Class1:
    def m(self):
        print("In Class1")

class Class2(Class1):
    def m(self):
        print("In Class2")
        Class1.m(self)

class Class3(Class1):
    def m(self):
        print("In Class3")
        Class1.m(self)

class Class4(Class2, Class3):
    def m(self):
        print("In Class4")
        Class2.m(self)
        Class3.m(self)

obj = Class4()
obj.m()
```

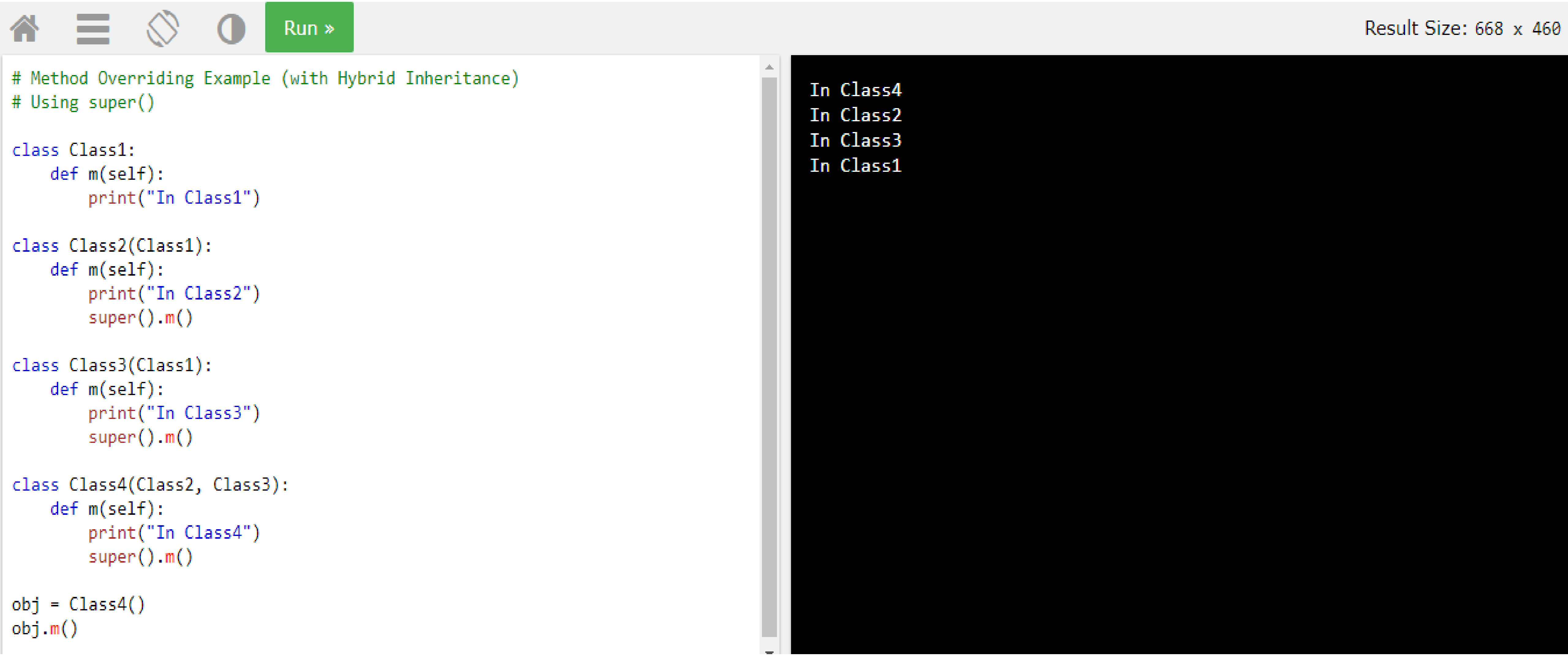
```
In Class4
In Class2
In Class1
In Class3
In Class1
```

5. Polymorphism

...continued

Method Overriding (Polymorphism with Hybrid Inheritance : Hierarchical + Multiple)

(Using super())



```
# Method Overriding Example (with Hybrid Inheritance)
# Using super()

class Class1:
    def m(self):
        print("In Class1")

class Class2(Class1):
    def m(self):
        print("In Class2")
        super().m()

class Class3(Class1):
    def m(self):
        print("In Class3")
        super().m()

class Class4(Class2, Class3):
    def m(self):
        print("In Class4")
        super().m()

obj = Class4()
obj.m()
```

```
In Class4
In Class2
In Class3
In Class1
```

5. Polymorphism

…continued

Method Overriding (Overriding Abstract Method : Abstract Base Class)

- An abstract class is not a concrete class, it cannot be instantiated.
- When we create an object for the abstract class it raises an error.
- A class which contains one or more abstract methods is called an abstract class.
- An abstract method is a method that has a declaration but does not have an implementation.
- Abstract methods must be implemented by its subclasses.
- By default, **Python does not provide abstract classes**.
- Python comes with a **module ABC** which provides the base class for defining **Abstract Base Classes(ABC).**
- First, abstract methods are created in base class and then concrete classes are implemented on basis of them.
- While we are designing large functional units, we use an abstract class.
- When we want to provide a common interface for different implementations of a component, we use an abstract class.

Why to use Abstract Base Classes :

By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

5. Polymorphism

…continued

Method Overriding (Overriding Abstract Method : Abstract Base Class)

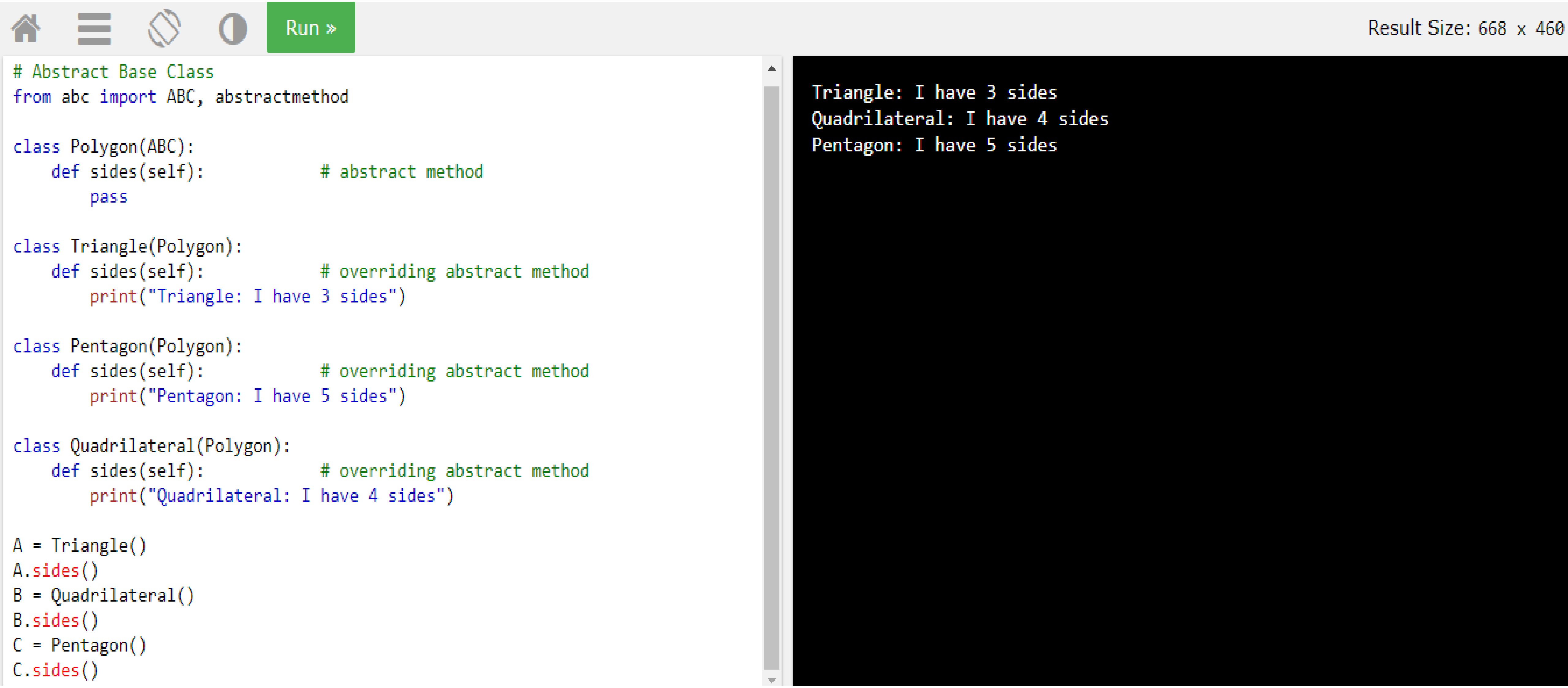
Abstract Classes and Pure Virtual Functions in C++

```
class A
{
   public:
      virtual void s() = 0;                              // Pure Virtual Function
};

class B:public A
{
   public:
      void s()
     {
         cout << "Virtual Function in Derived class\n";
      }
};
```

5. Polymorphism

…continued

Method Overriding (Overriding Abstract Method : Abstract Base Class)



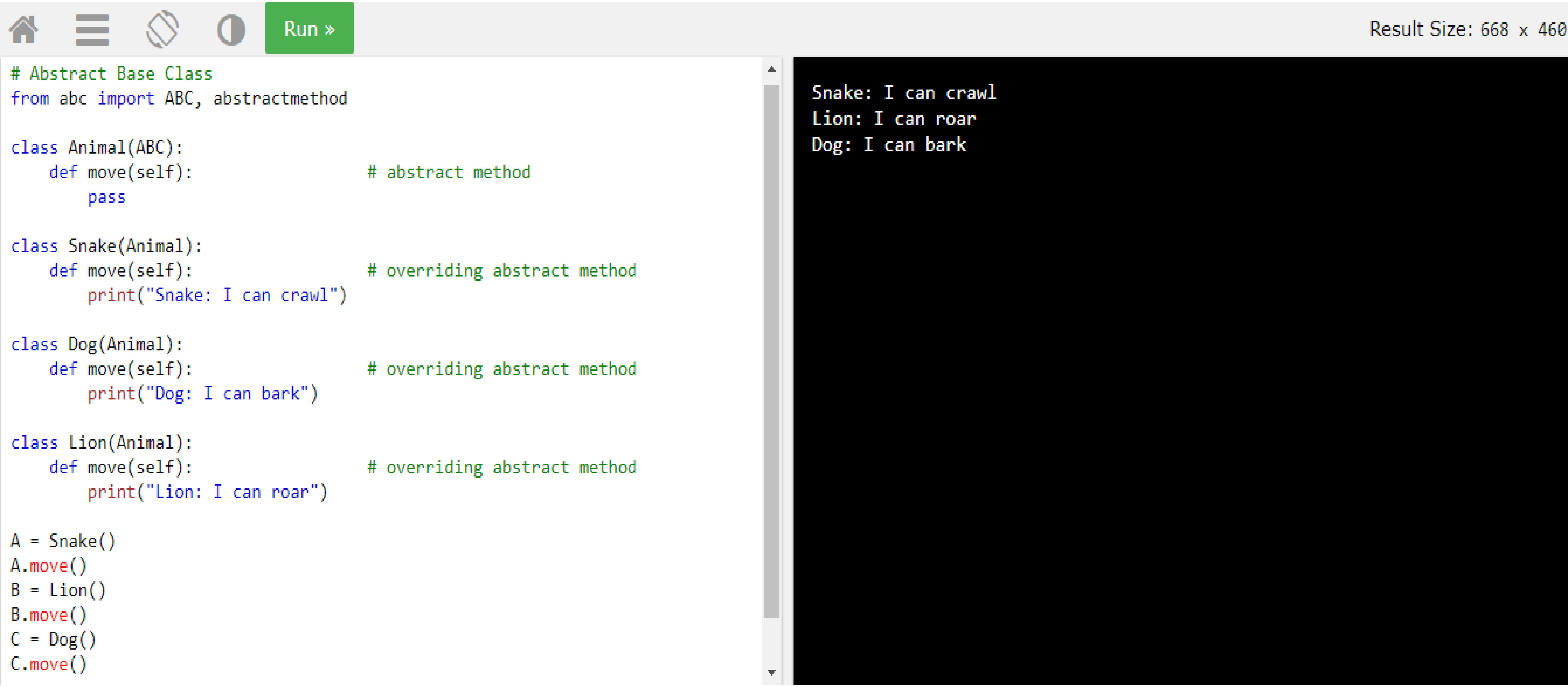
```
# Abstract Base Class
from abc import ABC, abstractmethod

class Polygon(ABC):
    def sides(self):            # abstract method
        pass

class Triangle(Polygon):
    def sides(self):            # overriding abstract method
        print("Triangle: I have 3 sides")

class Pentagon(Polygon):
    def sides(self):            # overriding abstract method
        print("Pentagon: I have 5 sides")

class Quadrilateral(Polygon):
    def sides(self):            # overriding abstract method
        print("Quadrilateral: I have 4 sides")

A = Triangle()
A.sides()
B = Quadrilateral()
B.sides()
C = Pentagon()
C.sides()
```

```
Triangle: I have 3 sides
Quadrilateral: I have 4 sides
Pentagon: I have 5 sides
```

5. Polymorphism

…continued

Method Overriding (Overriding Abstract Method : Abstract Base Class)



```
# Abstract Base Class
from abc import ABC, abstractmethod

class Animal(ABC):
    def move(self):                    # abstract method
        pass

class Snake(Animal):
    def move(self):                    # overriding abstract method
        print("Snake: I can crawl")

class Dog(Animal):
    def move(self):                    # overriding abstract method
        print("Dog: I can bark")

class Lion(Animal):
    def move(self):                    # overriding abstract method
        print("Lion: I can roar")

A = Snake()
A.move()
B = Lion()
B.move()
C = Dog()
C.move()
```

```
Snake: I can crawl
Lion: I can roar
Dog: I can bark
```

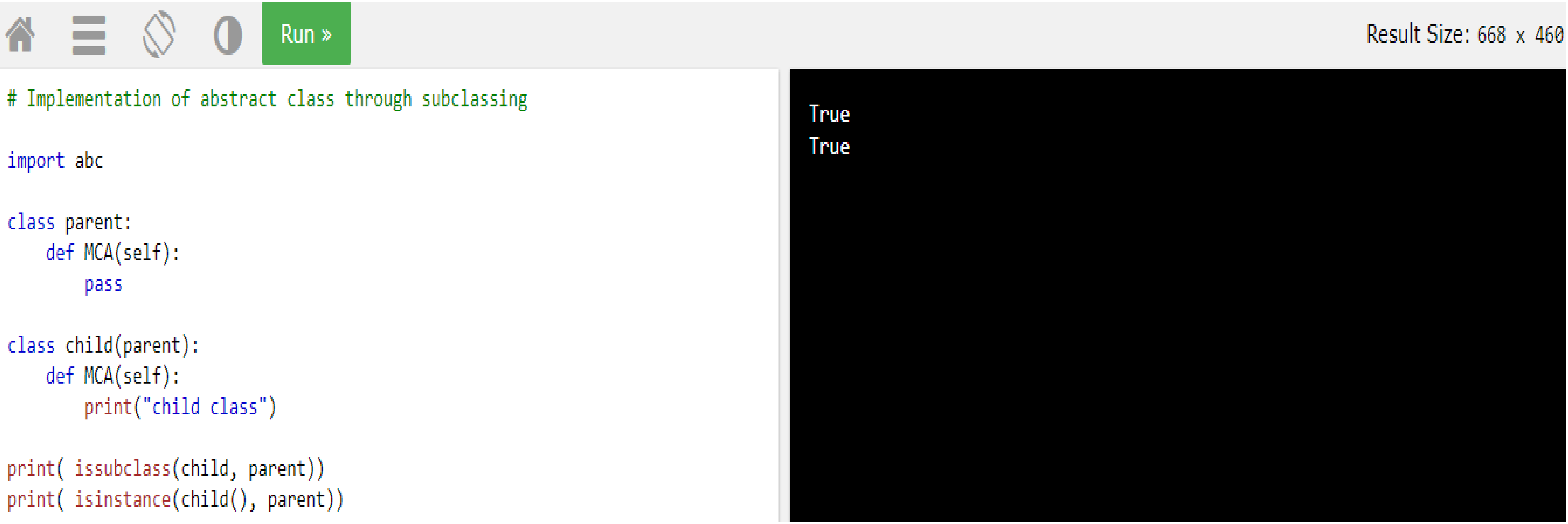
5. Polymorphism

…continued

Method Overriding (Overriding Abstract Method : Abstract Base Class)

Implementation of abstract class through subclassing

- By **subclassing** directly from the base, we can avoid the need to register the class explicitly.
- **ABC** introduces **virtual subclasses**, which are classes that don’t inherit from a class but are still recognized by **issubclass()** and **isinstance()** functions.



```
# Implementation of abstract class through subclassing

import abc

class parent:
    def MCA(self):
        pass

class child(parent):
    def MCA(self):
        print("child class")

print( issubclass(child, parent))
print( isinstance(child(), parent))
```

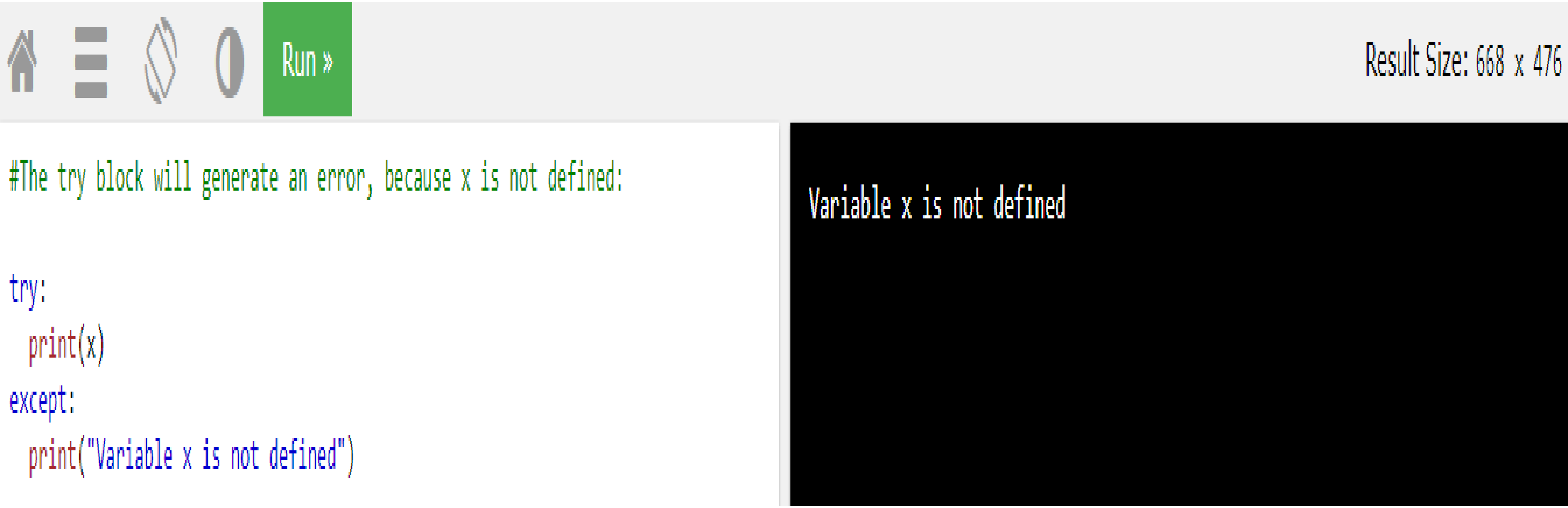
```
True
True
```

6. Exception Handling

6. Exception Handling (Try…Except)

**When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
These exceptions can be handled using the try statement.**

**The `try` block lets you test a block of code for errors.
The `except` block lets you handle the error.
The `finally` block will be executed regardless if the try block raises an error or not.
The else block can be used to define a block of code to be executed if no errors were raised by try block.
The raise keyword allows to throw an exception if a particular condition occurs.**

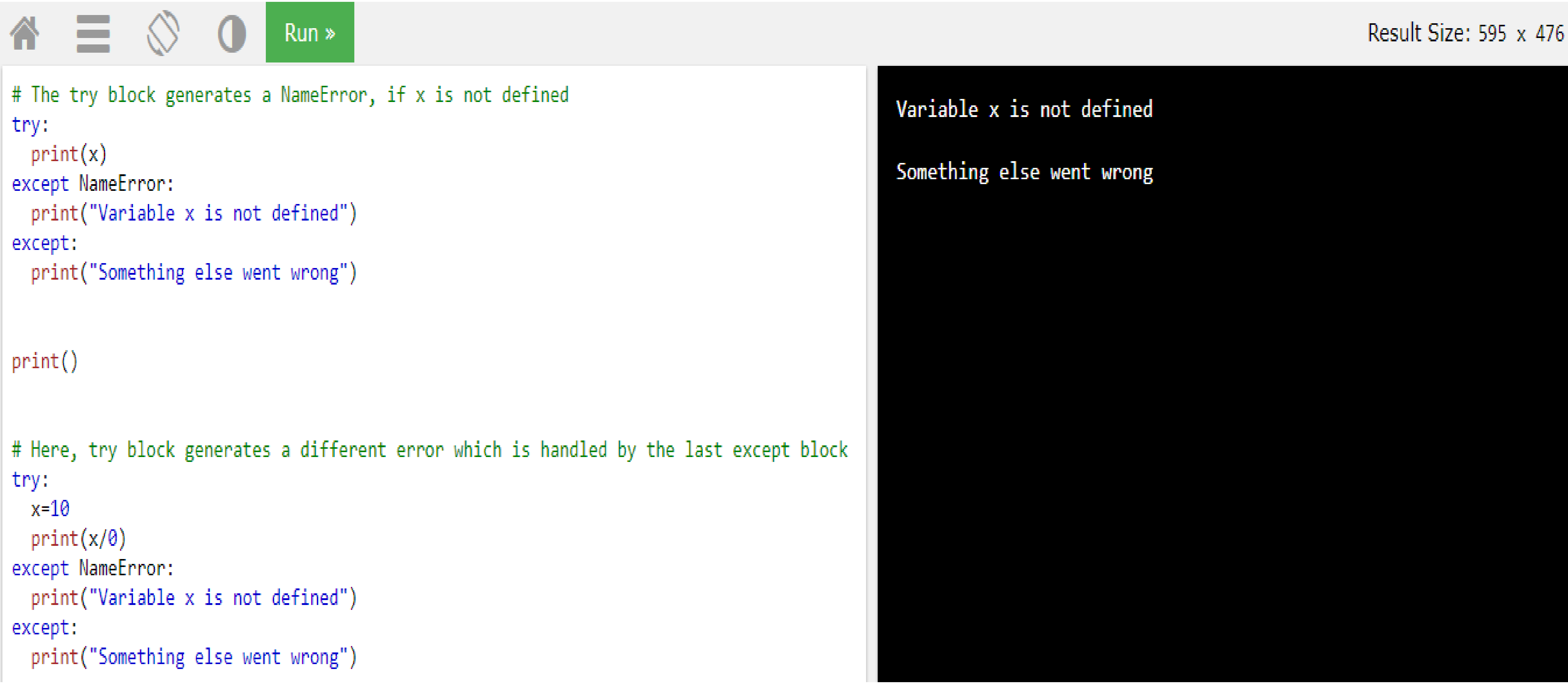


```
#The try block will generate an error, because x is not defined:


try:
  print(x)
except:
  print("Variable x is not defined")
```

```
Variable x is not defined
```

6. Exception Handling (Try…Except) …continued

Multiple Exceptions



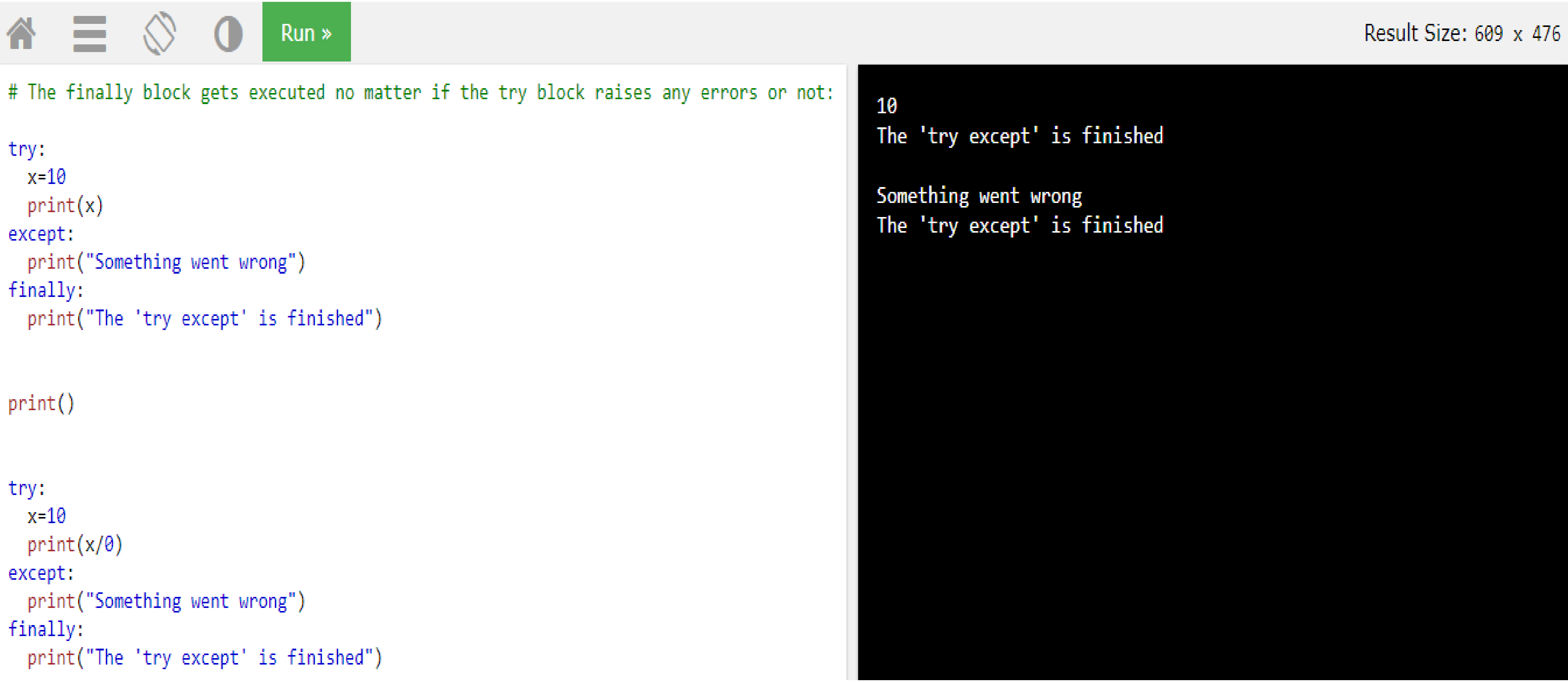
```
# The try block generates a NameError, if x is not defined
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")



print()



# Here, try block generates a different error which is handled by the last except block
try:
  x=10
  print(x/0)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

```
Variable x is not defined

Something else went wrong
```

6. Exception Handling (Try…Except)

…continued

finally block



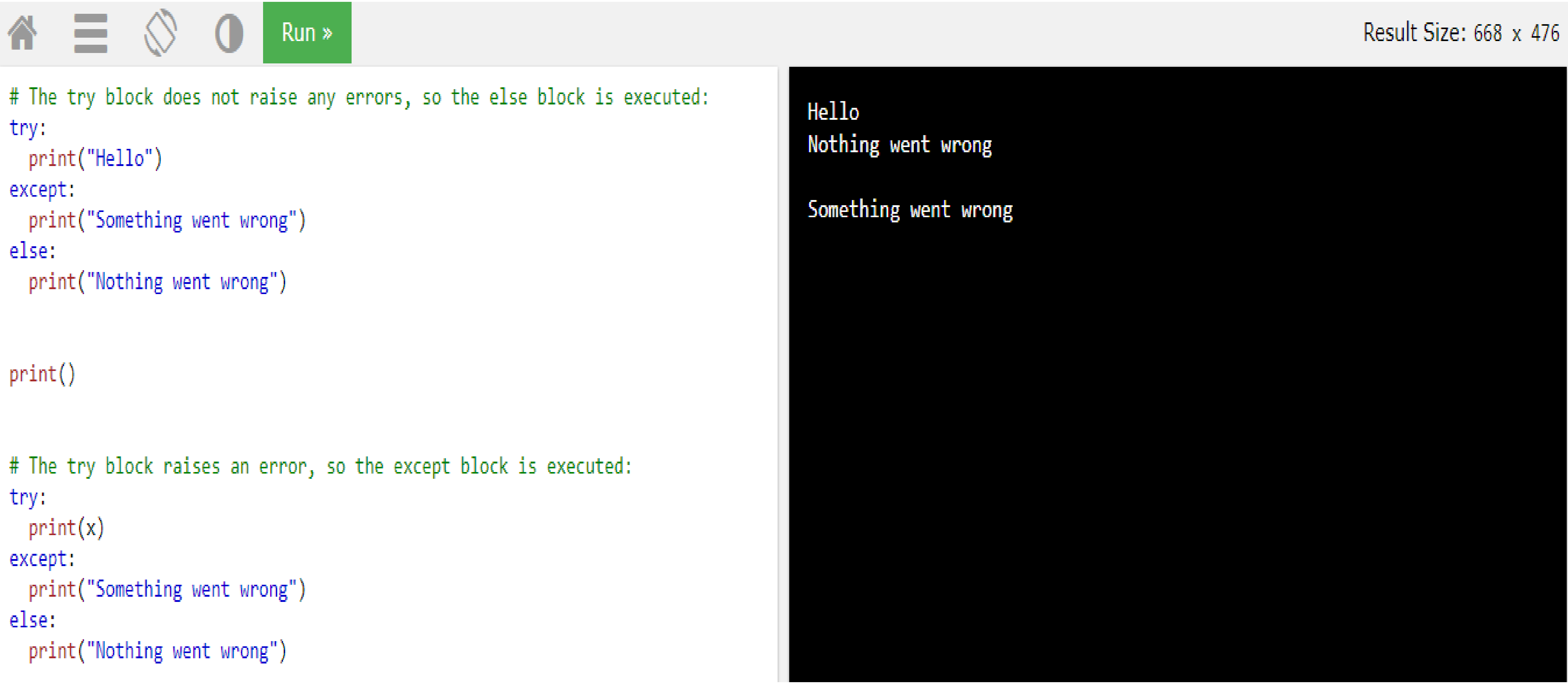
```
# The finally block gets executed no matter if the try block raises any errors or not:

try:
  x=10
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")


print()


try:
  x=10
  print(x/0)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

```
10
The 'try except' is finished

Something went wrong
The 'try except' is finished
```

6. Exception Handling (Try…Except)

…continued

else block



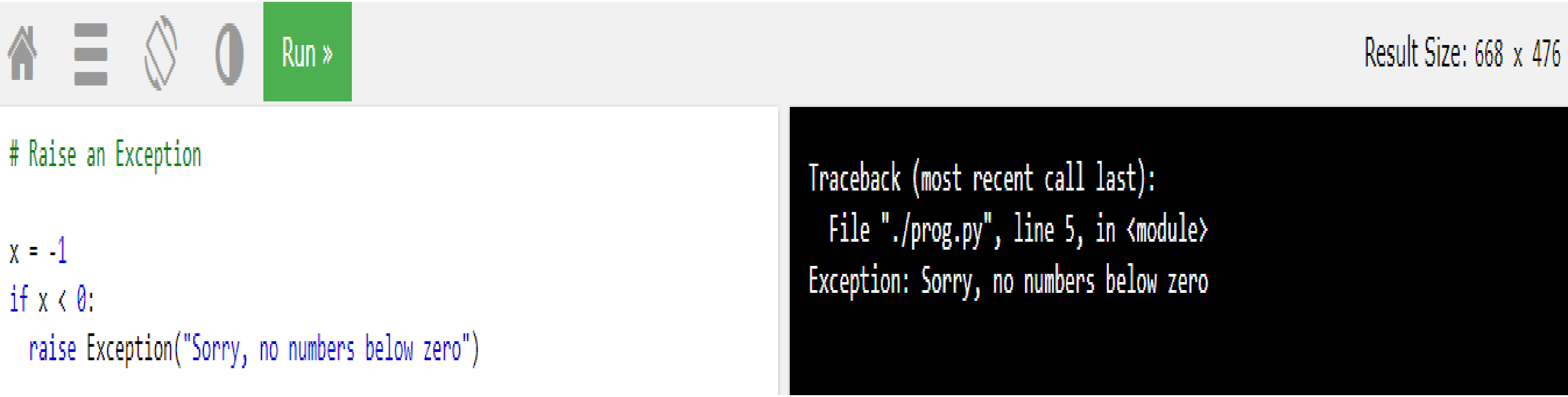
```
# The try block does not raise any errors, so the else block is executed:
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")


print()


# The try block raises an error, so the except block is executed:
try:
  print(x)
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

```
Hello
Nothing went wrong

Something went wrong
```

6. Exception Handling (Try…Except)

…continued

Raise an Exception



```python
# Raise an Exception

x = -1
if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

```
Traceback (most recent call last):
  File "./prog.py", line 5, in <module>
Exception: Sorry, no numbers below zero
```

B. Management of text files

File Handling and Basic File Operations

Types of files

Text, Binary

Opening a file

```
f = open(“filename OR filepath“, “mode”)                    # Returns a file object
```

“mode” argument is not mandatory. If it is not passed, then Python will assume it to be “ **r** ” by default.

Modes: r(read), w(write), a(append), x (create)

“r” for reading - returns an error if the file does not exist

“ r+ ” for both reading and writing - returns an error if the file does not exist

"w" for writing (Overwriting) - will create a file if the specified file does not exist

"a" for appending (Adding at end) - will create a file if the specified file does not exist

"x" Create - will create a file if the specified file does not exist, returns an error if the file exist

Closing a file

```
f.close()
```

You should always close your files.

In some cases, due to buffering, changes made to a file may not show until you close the file.

File Handling and Basic File Operations

…continued

Reading a file

```
print(f.read())
```

Reading part of the file

```
print(f.read(5))                          # Returns the 5 first characters of the file
```

Reading one line of the file

```
print(f.readline())                       # Returns one line of the file
```

Reading all lines of the file

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Writing into a file

```
f.write(“Message")              # File should be opened in ‘w’ mode
```

Appending into a file

```
f.write(“Message")              # File should be opened in ‘a’ mode
```

File Handling and Basic File Operations ...continued

Deleting a file

```
import os                                  # importing the OS module
os.remove(“filename")
```

Deleting a file after checking if it exists

```
import os
if os.path.exists(“filename"):
  os.remove(“filename")
else:
  print("The file does not exist")
```

Deleting a folder (Removing a directory)

```
import os
os.rmdir("foldername")                     # You can only remove empty folders.
```

File Handling and Basic File Operations

…continued

With statement

```
with open (“filename”, “mode”) as f:                    # There is no need to call f.close() while using with statement.
```

Copying the contents of a file ABC.txt to another file XYZ.txt

```
f2 = open(“XYZ.txt", "w")
with open(“ABC.txt", "r") as f1:
   f2.write(f1.read())
f2.close()
```

OR

```
with open(“ABC.txt") as f1:
   with open(“XYZ.txt", "w") as f2:
      for x in f1:
         f2.write(x)
```

C. Some more concepts

1. Packages and Modules

1. Packages and Modules

Package

Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist. Each package in Python is a directory which contains Python Modules and a special file called __init__.py. Due to __init__.py file, the interpreter interprets this directory as a Python Package. The __init__.py file can be empty. The package can also contain sub-packages inside it.

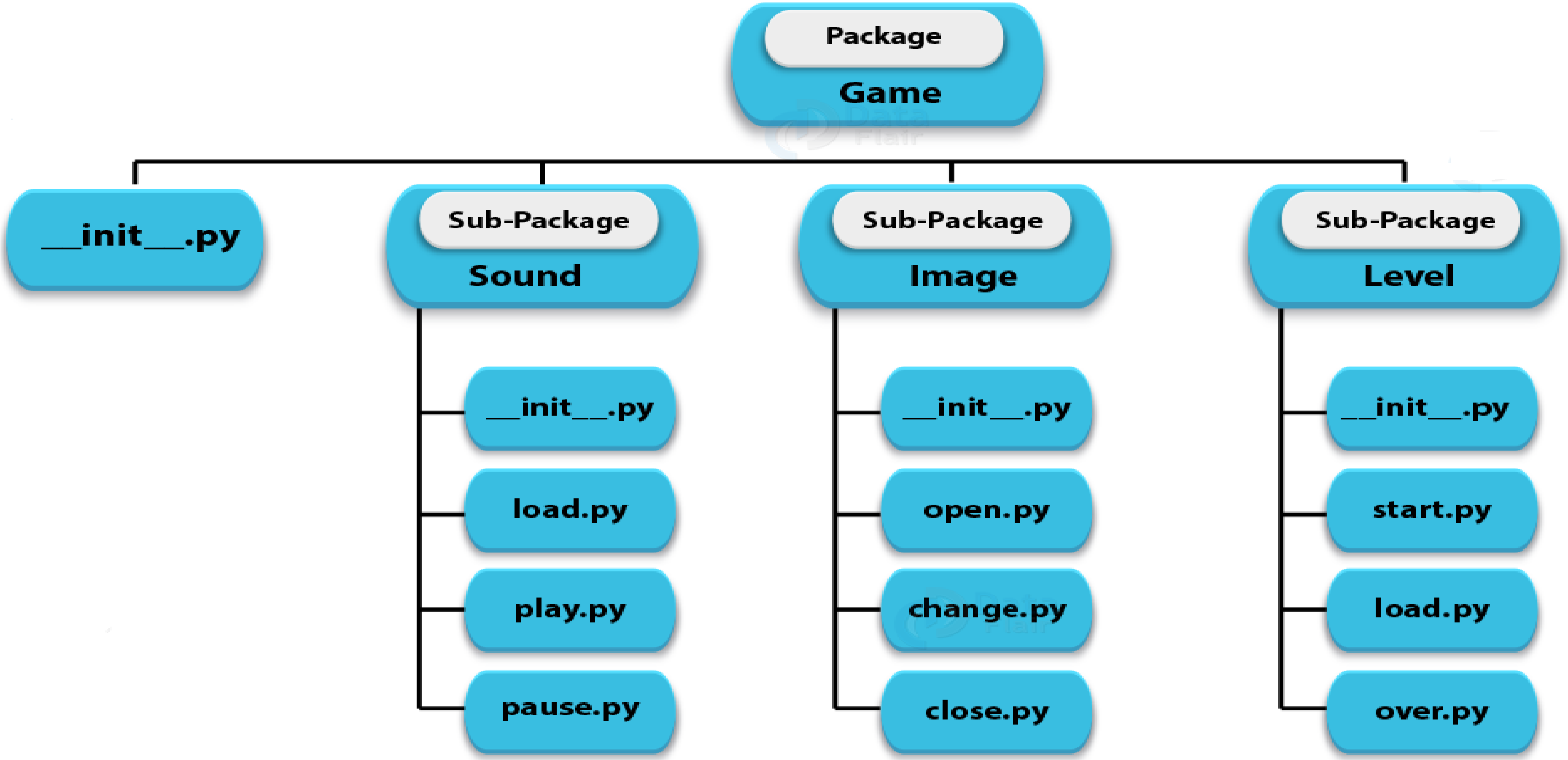
Module

A module is a piece of software that has a specific functionality. It is basically a simple Python file (with .py extension) that contains collections of functions and global variables. It is an executable file. Each module is a different file, which can be edited separately. For example, when building a ping pong game, one module would be responsible for the game logic, and another module would be responsible for drawing the game on the screen.

1. Packages and Modules

…continued

Package Module Structure



1. Packages and Modules

…continued

Importing packages and modules

If we create a directory or package called Package_ABC, we can then create a module inside that package called Module_XYZ. We also must not forget to add the __init__.py file inside the Package_ABC directory.

Packages can be imported the same way a module can be imported. To use the modules, we can import as follows:

A. Importing specific module objects to the current namespace

```
from Package_ABC import Module_XYZ
          OR
import Package_ABC. Module_XYZ
```

B. Importing all objects to the current namespace

```
from Package_ABC import *
          OR
import Package_ABC. *
```

1. Packages and Modules

…continued

Creating packages

```
Student(Package)
            | __init__.py (Constructor)
            | studentDetails.py (Module)
            | studentMarks.py (Module)
            | collegeDetails.py (Module)
```

Creating modules

A. Save the code in file called demo_module.py.

```
def myModule(name):
        print("This is My Module : "+ name)
```

B. Import module named demo_module and call myModule function inside it.

```
import demo_module
demo_module.myModule("Math")
```

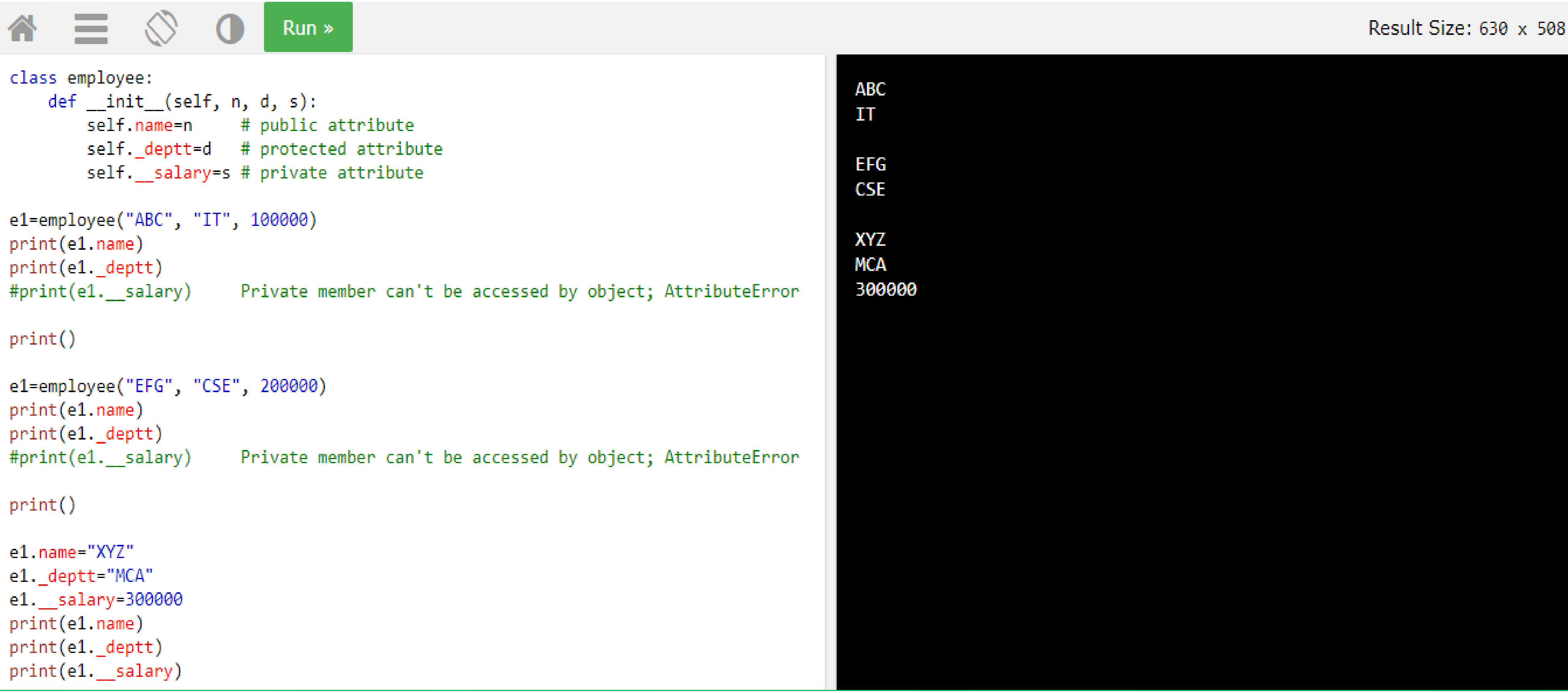
2. Replacement of Access Specifiers

2. Access Specifiers

- **Private** members (generally **attributes** declared in a class) of a class are denied access from the environment outside the class. They can be handled only from within the class.
- **Public** members (generally **methods** declared in a class) are accessible from within the class as well as outside the class. It can be accessed by anyone anywhere. The object of the same class is required to invoke a public method.
- **Protected** members of a class are accessible from within the class and are also available to its **sub-classes**. No other environment is permitted access to it.
- All members in a Python class are ***public** by default*. Any member can be accessed from outside the class.
- Python prescribes a convention of **prefixing** the name of the variable/method with **single or double underscore** to emulate the behaviour of protected and private access specifiers.
- Python doesn't have any mechanism that **effectively restricts access** to any instance variable or method.
- In Python, ***a single underscore _ prefixed to a variable makes it protected.***
- In fact, this doesn't prevent instance variables from accessing or modifying the instance. You can still modify the variable. Hence, the responsible programmer would refrain from accessing and modifying the instance variables prefixed with _ from outside its class.
- Similarly, ***a double underscore __ prefixed to a variable makes it private.***
- It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an **AttributeError**. Python performs name mangling of private variables. Every member with double underscore will be changed to ***object._class__variable*.** If so required, it can still be accessed from outside the class, but the practice should be refrained.

2. Access Specifiers

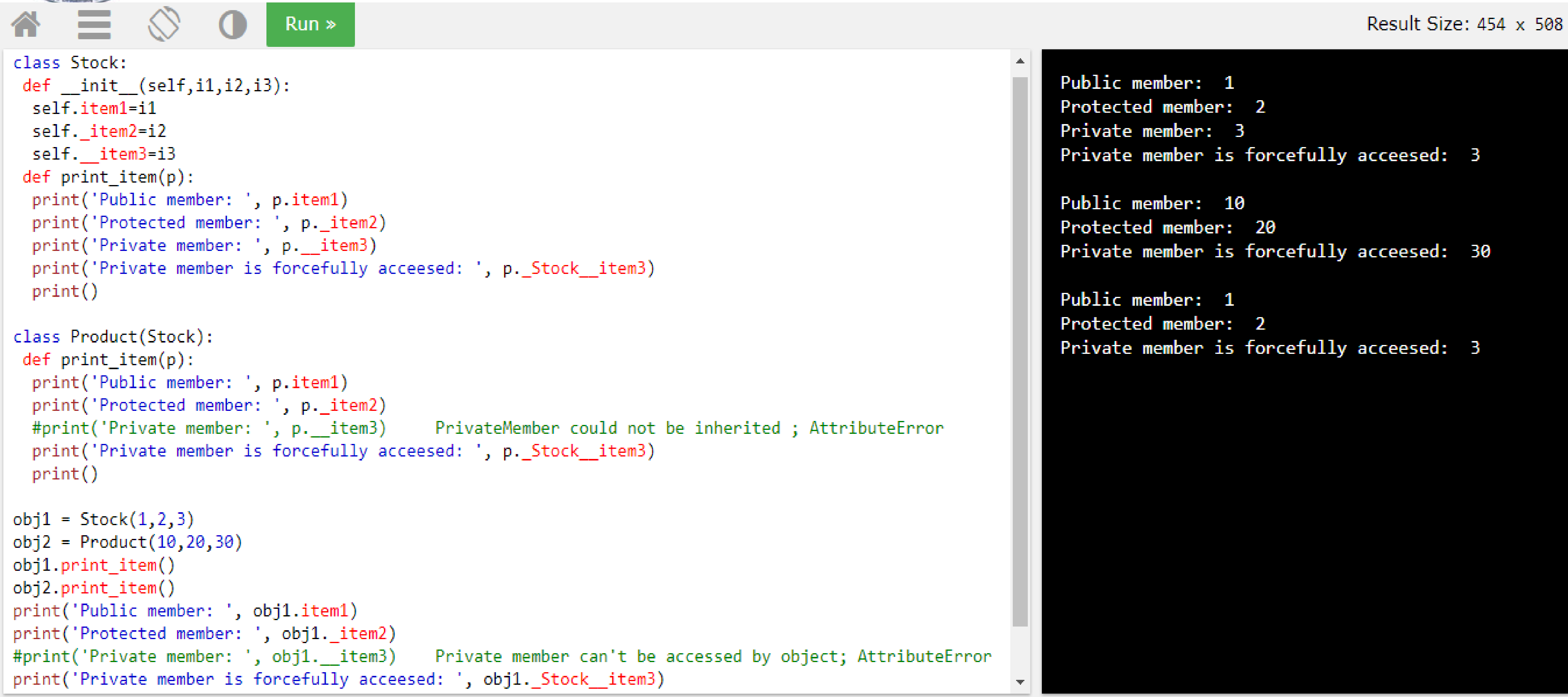
...continued



```
class employee:
    def __init__(self, n, d, s):
        self.name=n     # public attribute
        self._deptt=d   # protected attribute
        self.__salary=s # private attribute

e1=employee("ABC", "IT", 100000)
print(e1.name)
print(e1._deptt)
#print(e1.__salary)     Private member can't be accessed by object; AttributeError

print()

e1=employee("EFG", "CSE", 200000)
print(e1.name)
print(e1._deptt)
#print(e1.__salary)     Private member can't be accessed by object; AttributeError

print()

e1.name="XYZ"
e1._deptt="MCA"
e1.__salary=300000
print(e1.name)
print(e1._deptt)
print(e1.__salary)
```

```
ABC
IT

EFG
CSE

XYZ
MCA
300000
```

2. Access Specifiers …continued

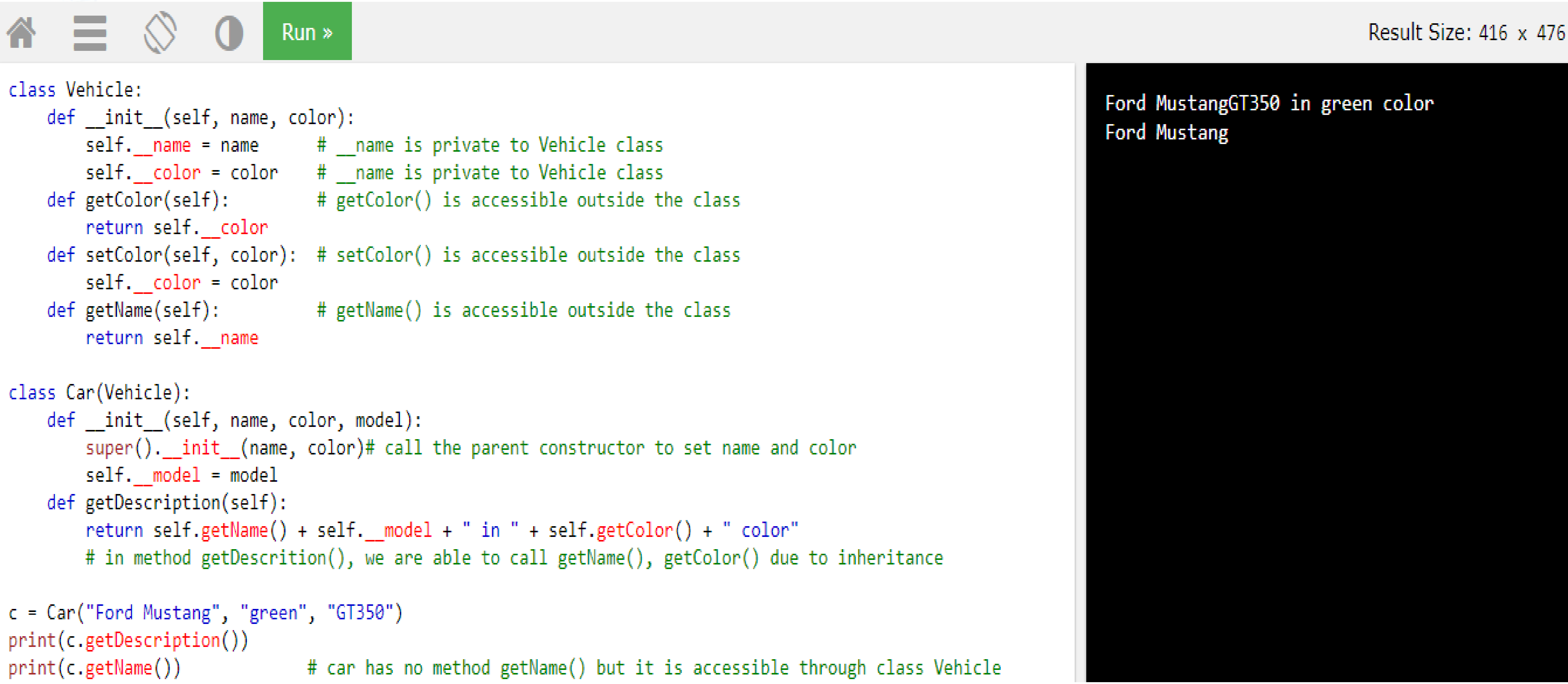


```
class Stock:
 def __init__(self,i1,i2,i3):
  self.item1=i1
  self._item2=i2
  self.__item3=i3
 def print_item(p):
  print('Public member: ', p.item1)
  print('Protected member: ', p._item2)
  print('Private member: ', p.__item3)
  print('Private member is forcefully acceesed: ', p._Stock__item3)
  print()

class Product(Stock):
 def print_item(p):
  print('Public member: ', p.item1)
  print('Protected member: ', p._item2)
  #print('Private member: ', p.__item3)     PrivateMember could not be inherited ; AttributeError
  print('Private member is forcefully acceesed: ', p._Stock__item3)
  print()

obj1 = Stock(1,2,3)
obj2 = Product(10,20,30)
obj1.print_item()
obj2.print_item()
print('Public member: ', obj1.item1)
print('Protected member: ', obj1._item2)
#print('Private member: ', obj1.__item3)    Private member can't be accessed by object; AttributeError
print('Private member is forcefully acceesed: ', obj1._Stock__item3)
```

```
Public member:  1
Protected member:  2
Private member:  3
Private member is forcefully acceesed:  3

Public member:  10
Protected member:  20
Private member is forcefully acceesed:  30

Public member:  1
Protected member:  2
Private member is forcefully acceesed:  3
```

2. Access Specifiers

…continued



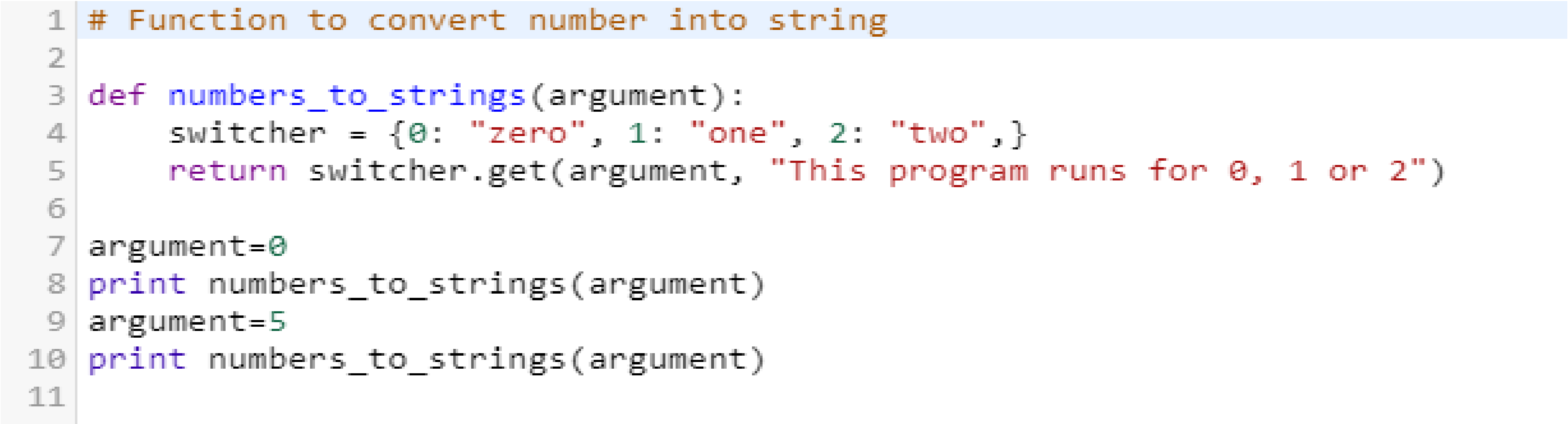
```python
class Vehicle:
    def __init__(self, name, color):
        self.__name = name      # __name is private to Vehicle class
        self.__color = color    # __name is private to Vehicle class
    def getColor(self):         # getColor() is accessible outside the class
        return self.__color
    def setColor(self, color):  # setColor() is accessible outside the class
        self.__color = color
    def getName(self):          # getName() is accessible outside the class
        return self.__name

class Car(Vehicle):
    def __init__(self, name, color, model):
        super().__init__(name, color)# call the parent constructor to set name and color
        self.__model = model
    def getDescription(self):
        return self.getName() + self.__model + " in " + self.getColor() + " color"
        # in method getDescrition(), we are able to call getName(), getColor() due to inheritance

c = Car("Ford Mustang", "green", "GT350")
print(c.getDescription())
print(c.getName())           # car has no method getName() but it is accessible through class Vehicle
```

```
Ford MustangGT350 in green color
Ford Mustang
```

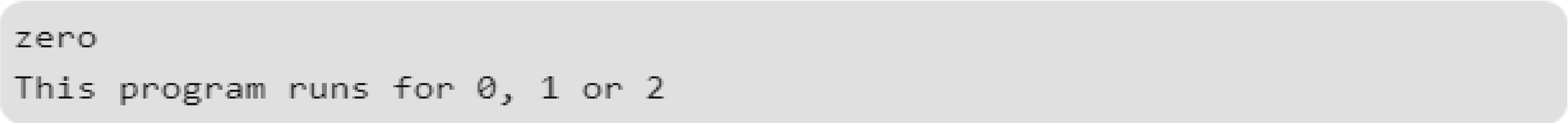
3. Replacement of switch case

3. Replacement of Switch Case in Python

- **Python does not have a switch or case statement.** To get around this fact, we use dictionary mapping.
- **Switcher** is **dictionary** data type
- **get()** method of dictionary data type returns **value of passed argument** if it is present in dictionary **otherwise second argument** will be assigned as **default value** of passed argument



```
# Function to convert number into string

def numbers_to_strings(argument):
    switcher = {0: "zero", 1: "one", 2: "two",}
    return switcher.get(argument, "This program runs for 0, 1 or 2")

argument=0
print numbers_to_strings(argument)
argument=5
print numbers_to_strings(argument)

```



```
zero
This program runs for 0, 1 or 2
```

Queries ?