In partial fulfillment of the requirements for the

CS 223 - Object-Oriented Programming

BEVERAGES SYSTEM

Presented to:
Dr. Unife O. Cagas
Professor

Prepared by:
Salvaleon, John Mike T.

BSCS-2A: Computer Science

May 16, 2024

Project Description

The Beverage System is a Python program that organizes beverage inventory and operations. It employs object-oriented principles like inheritance, encapsulation, abstraction, and polymorphism to model different beverage types efficiently. Through classes like Beverage, HotBeverage, and ColdBeverage, it manages attributes and behaviors such as name, price, and taste. The system allows users to create, describe, and differentiate between hot and cold beverages, providing a foundation for extending functionalities like inventory management and sales tracking.

Objectives

1. To present a concise menu featuring hot and cold beverages along with their respective prices and flavor descriptions to entice customers to make informed purchasing decisions.
2. Explain Abstraction, Encapsulation, Inheritance, and Polymorphism using code examples.
3. Analyze how classes are related through inheritance, creating specialized subclasses.
4. Discuss how instance variables are encapsulated within classes.
5. Explain how methods can be overridden to provide different implementations.
6. Discuss how abstract methods and simplified interfaces hide complexity.

7. Explain the process of creating objects and calling methods on them.

Importance and Contributions

This project demonstrates how to write organized and flexible code using object-oriented programming. It's helpful for both learners and teachers to understand coding concepts better. Plus, industries that use software can benefit from its lessons to improve their own programs.

The Four Principles of OOP

1. **Abstraction**: Simplifying complex systems by focusing on essential properties and hiding unnecessary details.

```
1 class Beverage:  # Abstraction
```

In this line class Beverage: # Abstraction, the comment # Abstraction means that the Beverage class simplifies things by representing any kind of drink without getting into specific

```
6     def taste(self):  # Abstraction
7         return "Default Taste"
```

This line def taste(self): # Abstraction defines a method called taste() in the Beverage class. The comment # Abstraction indicates that this method represents a simplified idea, giving a default taste without specifying specific

```
8
9     def describe(self):  # Abstraction
10        return f"This is a {self.name} and it costs ${self.price}. It tastes {self.taste()}."
11
```

This line def describe(self): # Abstraction defines a method named describe() within the Beverage class. It's marked with # Abstraction to indicate that it simplifies the process of describing a beverage by combining its name, price, and taste into a single message.

2. **Encapsulation:** Bundling data and methods into a single unit to control access and prevent external interference.

```
2     def __init__(self, name, price):  # Encapsulation
3         self.name = name
4         self.price = price
```

This line def __init__(self, name, price): # Encapsulation is where a new beverage is initialized with its name and price. The comment # Encapsulation indicates that this encapsulates the essential details of the beverage within the class, keeping everything tidy and organized.

```
13      def __init__(self, name, price):  # Encapsulation
14          super().__init__(name, price)
```

This line def __init__(self, name, price): # Encapsulation within the HotBeverage subclass defines a constructor method, indicating encapsulation. By calling super().__init__(name, price), it inherits and encapsulates the initialization behavior from its superclass, Beverage, ensuring proper initialization of

3. **Inheritance**: Allowing a subclass to inherit attributes and methods from a superclass, promoting code reuse.

```
11
12  class HotBeverage(Beverage):  # Inheritance
```

The line of code class HotBeverage(Beverage): indicates inheritance because it defines HotBeverage as a subclass of Beverage, meaning HotBeverage inherits all properties and methods from the Beverage class, allowing HotBeverage to reuse and extend the

```
18
19  class ColdBeverage(Beverage):  # Inheritance
```

The line of code class ColdBeverage(Beverage): indicates inheritance because it defines ColdBeverage as a subclass of Beverage, meaning ColdBeverage inherits all properties and methods from the Beverage class, allowing ColdBeverage to reuse and extend the

4. **Polymorphism**: Allowing objects of different classes to be treated as objects of a common superclass, enabling flexibility in programming.

```
16      def taste(self):  # Polymorphism
17          return "Mmmmmm.. Hot!"
```

The taste method shows polymorphism because it overrides the taste method in the Beverage class. This means the same method call taste can produce different outputs depending on whether it's called on an instance of HotBeverage or another subclass of

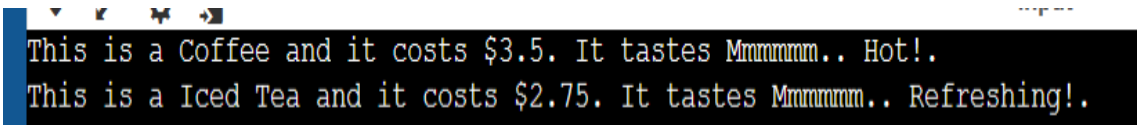
```
22
23      def taste(self):  # Polymorphism
24          return "Mmmmmm.. Refreshing!"
```

The taste method in this line demonstrates polymorphism because it overrides the taste method from the Beverage class, providing a different implementation. When called on an instance of ColdBeverage, it returns "Mmmmmm.. Refreshing!", showing that the same method can behave differently

Hardware and Software Used

- Laptop
- Mobile Phone
- Online GDB Compiler
- ChatGpt (Assistance Tool)

Output



Description:

By the used of the four principles of OOP, this output is achieved that describes two beverages. The first beverage is a coffee that costs $3.5 and is described as tasting "Mmmmmm.. Hot!". The second beverage is an iced tea that costs $2.75 and is described as tasting "Mmmmmm.. Refreshing!".

The Code

```python
class Beverage:  # Abstraction
    def __init__(self, name, price):  # Encapsulation
        self.name = name
        self.price = price

    def taste(self):  # Abstraction
        return "Default Taste"

    def describe(self):  # Abstraction
        return f"This is a {self.name} and it costs ${self.price}. It tastes {self.taste()}."

class HotBeverage(Beverage):  # Inheritance
    def __init__(self, name, price):  # Encapsulation
        super().__init__(name, price)

    def taste(self):  # Polymorphism
        return "Mmmmmm.. Hot!"

class ColdBeverage(Beverage):  # Inheritance
    def __init__(self, name, price):  # Encapsulation
        super().__init__(name, price)

    def taste(self):  # Polymorphism
        return "Mmmmmm.. Refreshing!"

hot_coffee = HotBeverage("Coffee", 3.50)  # Object
print(hot_coffee.describe())  # Object

iced_tea = ColdBeverage("Iced Tea", 2.75)  # Object
print(iced_tea.describe())  # Object
```

Description:

This code demonstrates the use of Object-Oriented Programming (OOP) principles to describe two beverages: coffee and iced tea. Encapsulation is utilized by defining a Beverage class that encapsulates the properties (name, cost, description) and behavior (method to display details) of a beverage. Abstraction is achieved through the display method, which abstracts how the beverage details are presented. Inheritance is demonstrated by creating Coffee and IcedTea classes that inherit from the Beverage class, allowing them to reuse and extend its properties and methods. Polymorphism is applied by using a loop to call the display method on instances of both Coffee and IcedTea, resulting in different outputs based on the specific

beverage object.

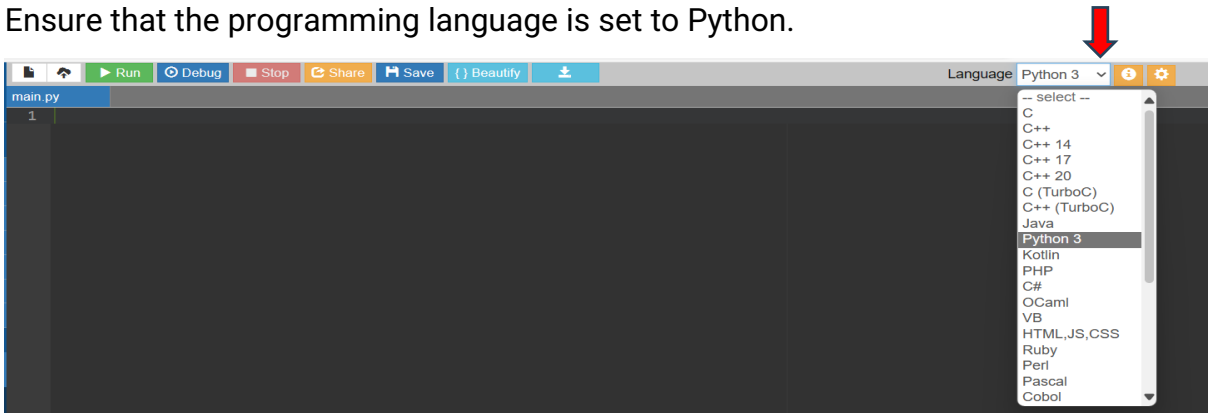
User Guide

1. Open OnlineGDB

Go to OnlineGDB.

2. Set Up the Environment

Ensure that the programming language is set to Python.



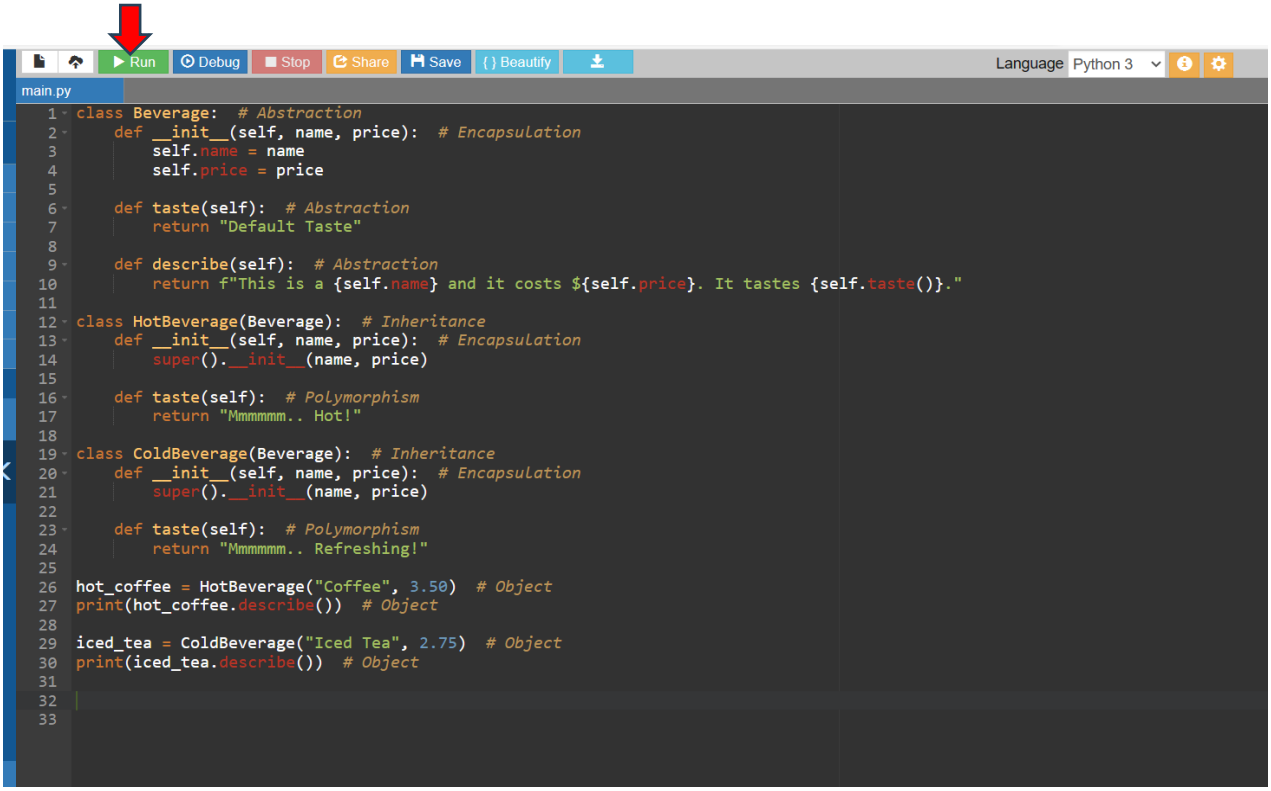
3. Enter the Code

Copy and paste the provided code into the code editor:

```
class Beverage:  # Abstraction
    def __init__(self, name, price):  # Encapsulation
        self.name = name
        self.price = price

    def taste(self):  # Abstraction
        return "Default Taste"

    def describe(self):  # Abstraction
        return f"This is a {self.name} and it costs ${self.price}. It tastes {self.taste()}."

class HotBeverage(Beverage):  # Inheritance
    def __init__(self, name, price):  # Encapsulation
        super().__init__(name, price)

    def taste(self):  # Polymorphism
        return "Mmmmmm.. Hot!"


class ColdBeverage(Beverage):  # Inheritance
    def __init__(self, name, price):  # Encapsulation
        super().__init__(name, price)

    def taste(self):  # Polymorphism
        return "Mmmmmm.. Refreshing!"

hot_coffee = HotBeverage("Coffee", 3.50)  # Object
print(hot_coffee.describe())  # Object

iced_tea = ColdBeverage("Iced Tea", 2.75)  # Object
print(iced_tea.describe())  # Object
```

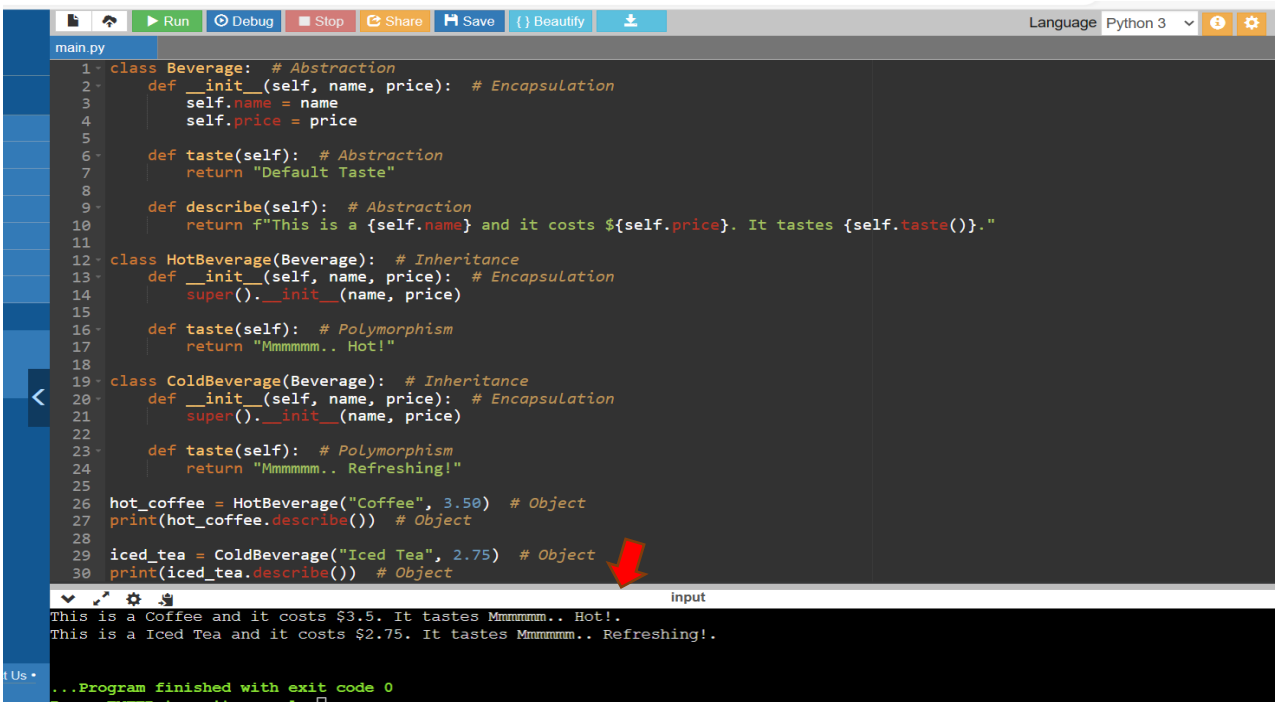
4. Run the Code

Click the "Run" button to execute the code.



5. View the Output

After clicking "Run," the output area will display below the code editor. This area shows the results of your code execution.



References

https://www.tutorialspoint.com/python/python_oops_concepts.htm

https://www.w3schools.com/python/python_inheritance.asp

https://realpython.com/learning-paths/object-oriented-programming-oop-python/

https://www.geeksforgeeks.org/abstract-classes-in-python/

https://www.geeksforgeeks.org/polymorphism-in-python/

https://www.w3schools.com/python/python_inheritance.asp

https://www.programiz.com/python-programming/object-oriented-programming

GDB online Debugger | Compiler - Code, Compile, Run, Debug online C, C++ (onlinegdb.com)

https://chatgpt.com/