Problem Set

1. What is the purpose of decorators in Python?

   a. They are used to create property managers for class attributes.

   b. They add functionality to classes consistent with the Python language.

   c. They transform existing code for better readability.

   d. All of the above.

2. Which magic method is used to provide a human readable string representation of a class?

   a. `__init__`

   b. `__str__`

   c. `__eq__`

   d. `__repr__`

3. Explain the purpose of the `property` function in Python and how it can be used to create getter and setter methods for class attributes.

4. List at least 3 magic methods in Python and briefly explain what they are used for.

5. Describe the concept of inheritance in Python and provide an example of multiple inheritance.

Solution Set

1. Answer: d. All of the above.

2. Answer: b. `__str__`

3. The `property` function in Python is used to create a property manager for class attributes. It allows us to define getter and setter methods for accessing and modifying the attribute. The getter method is decorated with `@property`, the setter method with `@<attribute>.setter`, and the deleter method (optional) with `@<attribute>.deleter`. This provides a more intuitive way of working with class attributes and allows for additional logic to be executed when accessing or modifying the attribute.

4. - `__init__`: This magic method is used as a constructor and is called when a new instance of a class is created.
   - `__str__`: This magic method is used to provide a human readable string representation of a class and is called by the `str()` function.
   - `__eq__`: This magic method is used to define the behavior of the equality operator (==) for instances of a class.

5. Inheritance in Python allows a class (derived class) to inherit attributes and behaviors from another class (base class). It promotes code reuse and allows for the creation of more specialized classes. Multiple inheritance refers to a situation where a derived class has more than one base class. This allows the derived class to inherit attributes and behaviors from multiple classes. For example:

```
class BaseClass1:
    def method1(self):
        # code for method1

class BaseClass2:
    def method2(self):
        # code for method2

class DerivedClass(BaseClass1, BaseClass2):
    def method3(self):
        # code for method3
```

In this example, `DerivedClass` inherits `method1` from `BaseClass1` and `method2` from `BaseClass2`. `DerivedClass` can also define its own methods, such as `method3`.