Discussion-6: Module-6 List Dictionary Class

Discussion Topic:

Select only one of the following questions and offer a good, substantive reply. Some repetition is inevitable, but please aim to reply to a question that has not already been answered. Please include the question number you respond to, so we all can better understand your post, such as “#1”, “#2”, etc.:

1. Beyond the constructor of a class, there are other OOP concepts that we may discuss to further our learning. Take your pick from the following list --only one of them, please!-- let's discuss it here, and be sure to include Python code examples:
    1. Classes and objects.
    2. Instantiation.
    3. Encapsulation.
    4. Inheritance.
    5. Polymorphism.
2. Is OOP (object-oriented programming) related to software reuse? Explain in detail why or why not.

Python is an OOP language -- object-oriented programming language. This week we will analyze and discuss the handful of OOP-related concepts and how they are implemented in Python. Learning these few terms well is essential for understanding practically any writing related to OOP, so it is important that we commit to memory very well what each of these terms means. Let's do this and dive into classes, objects, instantiation, encapsulation, inheritance, and polymorphism!

My Post:

Hello class,
For this discussion, I chose question #1 – Polymorphism #5.
Polymorphism is a Greek word that means many-shape or many-forms.
Polymorphism is a fundamental concept in object-oriented programming (OOP). Python is polymorphic, meaning that in Python objects have the ability to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways. (Vishal, 2021) Furthermore, in Python everything is an object/a class.
"Guido van Rossum has designed the language according to the principle "first-class everything". He wrote: "One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status." (Klein, 2022, 1. Object Oriented Programming)
To understand Polymorphism, it is important to understand the "Duck Typing" concept.
"If it looks like a duck and quacks like a duck, then it probably is a duck." In programming, this means that the suitability of an object is determined by the presence of certain methods and properties, rather than the actual type of the object.
In Python, Duck Typing is the concept where the 'suitability' of an object is determined by the presence of certain methods or attributes, rather than the actual type of the object.
In other words, Polymorphism in Python means that a single operator, function, or class method can have multiple forms/behaviors depending on the context.

1. **Operator polymorphism**
   Or operator overloading allows an operator like + to perform different operations based on the operand types. (Jergenson, 2022)
   For example:
   Two integers

```
int_1 = 10
int_2 = 45
print(str(int_1 + int_2))
>>>
55
```

Two strings

```
str_1 = "10"
str_2 = "45"
print(str_1 + str_2)
>>>
1045
```

2. **Function Polymorphism**
   Built-in functions like len() can act on multiple data types (e.g. strings, lists) and provide the length measured appropriately for each type.
   For example:

```
str_1 = "polymorphic"
print(str(len(str_1)))
>>>
11

my_lst = [1, 2, 3, 4, 5]
print(str(len(my_lst))
>>>
5
```

3. **Class method polymorphism**
   Allows subclasses to override methods inherited from the parent class.
   For example:

```
#  Parent class
class Animal:
   def make_sound(self):
      pass

# Child Class
class Dog(Animal):
   def make_sound(self):
      return "Woof!"

# Child Class
class Cat(Animal):
   def make_sound(self):
      return "Meow!"

def animal_sound(animal):
   print(animal.make_sound())

dog = Dog()
cat = Cat()

animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Output: Meow!
```

4. **Independent classes can also define methods with the same name that behave differently.**
   For example:

```
def enter_obj(obj):
   return obj.action()
```

```
# Independent class
class Animal:
   def __init__(self, food):
      self.food = food
   # same name as the Circle method different functionality
   def action(self):
      print(f"eats {self.food}")

# Independent class
class Circle:
      def __init__(self, radius):
         self.radius = radius
      # same name as the Animal method different functionality
      def action(self):
         return 3.14 * (self.radius ** 2)

cow = Animal("grass")
circ = Circle(7)

enter_obj(cow)
print(str(enter_obj(circ)))
>>>
eats grass
153.86
```

In conclusion, polymorphism is a powerful feature of Python. It allows objects to take on multiple forms and behave differently based on the context. Python's duck typing enables polymorphism by focusing on the presence of certain methods or attributes rather than the actual type of the object.

-- Alex

References:
Jergenson, C. (2022, May 31). *What is polymorphism in python?*. Educative. https://www.educative.io/blog/what-is-polymorphism-python
Klein, B. (2022, February 1). *object oriented programming / OPP*. python-course. https://python-course.eu/oop/object-oriented-programming.php
Vishal. (2021, October 21). Polymorphism in python. PYnative. https://pynative.com/python-polymorphism/