Encapsulation, Polymorphism, Inheritance

Guttag Chapter 10

Goals

- wrap up oop concepts
- study getattr()
- run example code

Wrap up oop concepts

Guttag Chapter 10

OOP Concept: Decomposition

Definition

- defining functions that can tackle individual/isolated parts of a problem

Example

- palindrome_recursive
- palindrome_reverse
- get_human_readable_boolean
- to_char

OOP Concept: Encapsulation

Definition

- bringing together both data and functions (methods) that operate on the same data

Example

```
class Vehicle():
  """Abstract data type representing a vehicle."""

  def __init__(self, num_seats: int, num_doors: int, engine_type: str):
    """Define the constructor."""
    self._seats = num_seats
    self._doors = num_doors
    self._engine = engine_type
    self._milage = 0.0

  def drive(self, num_miles: float):
    """Add milage to the vehicle."""
    self._milage += num_miles
    return None

  def milage(self):
    """Get the milage of the vehicle."""
    return self._milage

  def __repr__(self):
    """Define the printable representation of the vehicle."""
    return f"{self._engine} vehicle with {self._seats} seats, " +\
           f"{self._doors} doors, and {self._milage} miles."
```

OOP Concept: Information ~~Hiding~~ Abstraction

Definition

- protecting data inside of a class
- defining class-related things in the class
- Viewing the inside world as separate from the outside world

Example

- implementing changes inside the class would not impact the outside usage

```
class Vehicle():
  """Abstract data type representing a vehicle."""

  def __init__(self, num_seats: int, num_doors: int, engine_type: str):
    """Define the constructor."""
    self._seats = num_seats
    self._doors = num_doors
    self._engine = engine_type
    self._milage = 0.0

  def drive(self, num_miles: float):
    """Add milage to the vehicle."""
    self._milage += num_miles
    return None

  def milage(self):
    """Get the milage of the vehicle."""
    return self._milage

  def __repr__(self):
    """Define the printable representation of the vehicle."""
    return f"{self._engine} vehicle with {self._seats} seats, " +\
           f"{self._doors} doors, and {self._milage} miles."
```

OOP Concept: Polymorphism

Definition

- defining methods that other objects also use
- overloading

Example

- _ _ repr _ _
- _ _ lt _ _
- _ _ eq _ _
- _ _ add _ _

```
class Vehicle():
  """Abstract data type representing a vehicle."""

  def __init__(self, num_seats: int, num_doors: int, engine_type: str):
    """Define the constructor."""
    self._seats = num_seats
    self._doors = num_doors
    self._engine = engine_type
    self._milage = 0.0

  def drive(self, num_miles: float):
    """Add milage to the vehicle."""
    self._milage += num_miles
    return None

  def milage(self):
    """Get the milage of the vehicle."""
    return self._milage

  def __repr__(self):
    """Define the printable representation of the vehicle."""
    return f"{self._engine} vehicle with {self._seats} seats, " +\
           f"{self._doors} doors, and {self._milage} miles."
```

OOP Concept: Inheritance

Definition

- building off of a superclass

Example

- implementing a class that inherits from Vehicle, but is specifically a sub-type
- bike

```
# A Bicycle IS A Vehicle
# (go with me on this)

class Bicycle(Vehicle):
  def __init__(self):
    Vehicle.__init__(self, num_seats=1, num_doors=0, engine_type="human")

  def ride(self, num_miles: float):
    """Add milage to the bike."""
    self._milage += num_miles

  def __repr__(self):
    """Redefine the printable representation for convenience."""
    return f"I'm a bike, my engine is a {self._engine}.... but also {Vehicle.__repr__(self)}"
```

Getattr

Getattr

Definition

- a built-in python function that can look up attributes or methods using a string!
- it turns a string into code!
- indirect way to use dot notation when the only available option is a string!
- https://docs.python.org/3/library/functions.html

Example

- getattr(bike, 'foobar') is equivalent to x.foobar

Implement it yourself!

Explicitly check (hardcode) for an attribute

```
def my_getattr(object: Vehicle, string: str):
    if string == "mileage":
        return object.mileage
    else:
        return "your string is either not an attribute, or the attribute is private!"
```

Example Code

Python Notebook

Summary

Classes are useful because they can be used for **Abstraction** and **Encapsulation**, i.e., to organize code and protect data

Polymorphism, or overloading, can make new objects compatible with old/standard syntax

Inheritance, or building off a superclass, can facilitate organization