Introduction To Python

Dr.Hajialiasgari

Tehran University
Of
Medical Science

November 17, 2024

TEHRAN UNIVERSITY
OF
MEDICAL SCIENCES

Object Oriented

- A program is made up of many cooperating objects
- Instead of being the whole program - each object is a little island within the program and cooperatively working with other objects
- A program is made up of one or more objects working together - objects make use of each others capabilities

Object

- An Object is a bit of self-contained Code and Data
- A key aspect of the Object approach is to break the problem into smaller understandable parts (divide and conquer)
- Objects have boundaries that allow us to ignore un-needed detail
- We have been using objects all along: String Objects, Integer Objects, Dictionary Objects, List Objects...

Definitions

- `Class` - a template
- `Method or Message` - A defined capability of a class
- `Field or attribute` - A bit of data in a class
- `Object or Instance` - A particular instance of a class

Terminology: Class

- Defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). One might say that a class is a blueprint or factory that describes the nature of something. For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors).

Terminology: `Instance`

- One can have an instance of a class or a particular object. The instance is the actual object created at runtime. In programmer jargon, the Lassie object is an instance of the Dog class. The set of values of the attributes of a particular object is called its state. The object consists of state and the behavior that's defined in the object's class.

Terminology: `Method`

- An object's abilities. In language, methods are verbs. Lassie, being a Dog, has the ability to bark. So bark() is one of Lassie's methods. She may have other methods as well, for example sit() or eat() or walk() or save`_timmy()`. Within the program, using a method usually affects only one particular object; all Dogs can bark, but you need only one particular dog to do the barking

Some Python Objects

```
>>> x = 'abc'
>>> type(x)
<class 'str'>
>>> type(2.5)
<class 'float'>
>>> type(2)
<class 'int'>
>>> y = list()
>>> type(y)
<class 'list'>
>>> z = dict()
>>> type(z)
<class 'dict'>
```

Some Python Objects

```
>>> dir(x)
[  'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find',
'format',  'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> dir(y)
[ 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir(z)
[, 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys',
'pop', 'popitem', 'setdefault', 'update', 'values']
^^I
```

A Simple Class

```
class Car:
# Constructor to initialize attributes
def __init__(self, make, model, year):
self.make = make     # Car manufacturer
self.model = model   # Car model
self.year = year     # Manufacturing year
^^I
# Method to start the car
def start(self):
return f"The {self.year} {self.make} {self.model} starts. Vroom!"
^^I
# Method to display car details
def details(self):
return f"{self.year} {self.make} {self.model}"^^I
```

A Simple Class

```
# Create instances of the Car class
car1 = Car("Toyota", "Corolla", 2020)
car2 = Car("Tesla", "Model 3", 2023)
^^I
# Access methods and attributes
print(car1.start())
# Output: The 2020 Toyota Corolla starts. Vroom!
print(car1.details())
# Output: 2020 Toyota Corolla
print(car2.start())
# Output: The 2023 Tesla Model 3 starts. Vroom!
print(car2.details())
# Output: 2023 Tesla Model 3^^I
```

Key Features in Previous Example:

- `Attributes:` make, model, and year define the properties of the car.
- `Methods:` start and details add functionality specific to a car.
- `Multiple Instances:` You can create multiple cars (car1, car2) with different attributes, showcasing reusability.
- class is a reserved word that defines a template for making objects.
- Each `car` object has a bit of code.

Playing with `dir()` and `type()`

- A Nerdy Way to Find Capabilities :
- The `dir()` command lists capabilities
- Ignore the ones with underscores - these are used by Python itself
- The rest are real operations that the object can perform
- It is like `type()` - it tells us something *about* a variable

Code Example

```
>>> y = list()
>>> type(y)
<class 'list'>
>>> dir(y)
['__add__', '__class__', '__class_getitem__', '__contains__',
'__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
... 'append', 'clear', 'copy','count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

Code Example

```
class Car:
def __init__(self):
self.speed = 0  # Initialize the speed attribute

def accelerate(self):
self.speed += 10
print("Speed:", self.speed)

# Create an instance of Car
car = Car()

# Check the type and attributes of the object
print("Type", type(car)) # Type of the object
print("Dir ", dir(car))  # Directory of attributes and methods
print("Type", type(car.speed)) # Type of the speed attribute
print("Type", type(car.accelerate)) # Type of the accelerate method
```

Code Example

```
Type <class '__main__.Car'>
Dir  ['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'accelerate', 'speed']
Type <class 'int'>
Type <class 'method'>
```

Inheritance

- When we make a new class - we can reuse an existing class and inherit all the capabilities of an existing class and then add our own little bit to make our new class
- Another form of store and reuse
- Write once - reuse many times
- The new class (child) has all the capabilities of the old class (parent) - and then some more

Terminology: `Inheritance`

- `Subclasses` are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own.

Code Example

```
# Parent class

class Car:
def __init__(self, make, model, year):
self.make, self.model
self.year, self.speed = make, model, year, 0

def accelerate(self):
self.speed += 10
print(f"{self.make} {self.model} accelerates. Speed:{self.speed}")

def brake(self):
self.speed = max(0, self.speed - 10)
print(f"{self.make} {self.model} slows down. Speed: {self.speed}")
```

Code Example

```
# Subclass

class ElectricCar(Car):
def __init__(self, make, model, year, battery_capacity):
super().__init__(make, model, year)
self.battery_capacity = battery_capacity

def charge(self):
print(f"{self.make} {self.model} is charging. Battery: {self.battery
```

Code Example

```
# Using the classes
car1 = Car("Toyota", "Corolla", 2020)
car2 = ElectricCar("Tesla", "Model 3", 2023, 75)
^^I
car1.accelerate()
car1.brake()
car2.accelerate()
car2.charge()
^^I
#output
Toyota Corolla accelerates. Speed: 10 km/h
Toyota Corolla slows down. Speed: 0 km/h
Tesla Model 3 accelerates. Speed: 10 km/h
Tesla Model 3 is charging. Battery: 75 kWh^^I
```

Summary

- Object Oriented programming is a very structured approach to code reuse
- We can group data and functionality together and create many independent instances of a class

End of Object Oriented Programming