Object Oriented Programming

What is Object Oriented Programming?

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects". (https://en.wikipedia.org/wiki/Object-oriented_programming)

As your program grows in complexity, you may need something more than simple built-in data types such as `str`, `int`, or `list`. For example, when you create a game, you may need an `Avatar`, or `Weapon`, etc. In these cases, it is easier to organize your code around objects. You can think of objects as your own user-defined data types. Later you will see that these objects have two main things:

- attributes: which defines the characteristic of the object, and
- methods: which defines what the object can do Attributes and methods define your object.

You actually have worked with objects if you use `list` and `str` data type in your program. These are called **built-in** objects in Python. Python has provided these objects for you to use. What we will do in this section is to create your own **user-defined** objects.

We will see that user-defined objects are made of other data (attributes) and computations (methods). Moreover, we will see that **any** code can be abstracted as an **object** since any computer code are made of data (attributes) and some computations (methods). In these lessons, we will see how OOP will be used for both creating user-defined data type as well as for abstracting the whole program.

Attributes and Methods

For example, let's say you want to create a computer game with a Robot Turtle as its character. In this case, you may want to define a new data type called `RobotTurtle`. `RobotTurtle` will have the **attributes** `speed` and `name`. Attributes describes the object and its properties. It is usually a *noun* and it is defined as a kind of variable within the object. On the other hand a Robot Turtle can `move`. So the data type `RobotTurtle` would have a **method** called `move`. Methods are a kind of functions which apply to our user-defined data type. A method describes what the object can do and so it is usually a *verb*. In order to create our user-defined objects, we have to do the following:

1. Define a class, which defines the object with its attributes and methods
2. Instantiate an object, which actually creates the object

The class definition tells Python about your user-defined object and how to create it. It tells Python what attributes this object has using some existing built-in data types or other defined objects. It tells Python what methods the object can do. But it is important to note that a class definition is just like a kind of contract on a piece of paper. The contract does not create the object. *Instantiation* is the step that actually creates the object in the computer's memory. We will show these two steps below.

First, let's start by defining our `RobotTurtle` class.

In [10]:

```
# Class definition
class RobotTurtle:
    # Attributes:
    def __init__(self, name, speed=1):
        self._name = name
        self._speed = speed
        self._pos = (0, 0)

    # Methods:
    def move(self, direction):
        update = {'up' : (self._pos[0], self._pos[1] + self._speed),
                  'down' : (self._pos[0], self._pos[1] - self._speed),
                  'left' : (self._pos[0] - self._speed, self._pos[1]),
                  'right' : (self._pos[0] + self._speed, self._pos[1])}
        self._pos = update[direction]

    def tell_name(self):
        print(f"My name is {self._name}")
```

Some notes on the class definition:

Some notes on the class definition:

- Notice the above code starts with a Class Definition. To define a class, we use the keyword `class` followed by the class name `RobotTurtle`.
- The keyword `def` inside the class defines the **method** which tells Python what the object can do.
- The first method is special and it is called `__init__()`. This method is always called during *object instantiation*. This special method is called to *initialize* the object's attributes during *instantiation*. In this definition, we see that during instantation, we ask Python to initialize three attributes:
  - `_name` which is a string and is initialized using the first argument during object instantiation.
  - `_speed` which is a number and is initialized using the second optional argument.
  - `_pos` which is the position using a tuple of two numbers and is initalized to `(0,0)`.
- The class definition also contains two other **user-defined** methods:
  - `move(direction)` which is to move the Robot Turtle to certain direction according to its speed.
  - `tell_name()` which is to print out the name of the Robot Turtle.

It is important to remember that the class definition is just a description of the object and works as a kind of template or contract. The definition does not create the object itself. The object creation happens by doing the following:

In [11]:

```
# Object Instantiation
my_robot = RobotTurtle("T1")
```

The above line is what we call as **object instantiation**. When Python executes this lines, it *instantiates* an object of the type `RobotTurtle` in the memory. A few notes on object instantation:

- The object is created or *instantiated* by using the class name followed by some values used to initialze the object. In this case: `RobotTurtle("T1")`. The argument "T1" is passed on to initialize the object's name. This object is then pointed to by the variable `my_robot`.
- Each of the argument in the *object instantiation* is passed on to the `__init__()` method. In this case, "T1" is passed on to the formal argument `name` in `__init__()`.
- The **first** argument of any method in a class is always called `self` following Python's PEP8. The `self` argument is also found as the first argument inside the method `move` and `tell_name`. The first arugment `self` refers to the particular object instance of the class. It can also be used to access methods and attributes of the current object.
- At the end of object instantiation, the object `my_robot` would have the following attributes initalized:
  - `_name` with a value `T1`
  - `_speed` with a value of `1`
  - `_pos` with a value of `(0,0)`

Once the object is created, we can access its attributes and methods. For example, you can ask the robot to tell its name.

In [12]:

```
# Accessing object's method
my_robot.tell_name()
```

```
My name is T1
```

- `my_robot.tell_name()` is calling the method `tell_name()` using the **dot operator**. To call any method, we use the format of

```
object.method_name(arguments)
```

- If you run the cell above, you will see "My name is T1" printed on the output

You can actually access the attributes directly and change it, for example

In [13]:

```
# accessing object's attribute
print(my_robot._speed)
my_robot._speed = 2
print(my_robot._speed)
```

```
1
2
```

- the first and the third line access the object's attribute using the **dot operator**.
- the second line assigned the value 2 into the object's `_speed` attribute.
- if you run the cell above, you will see the speed changes from 1 to 2.

The following examples show more examples on how one can access object's attributes and methods using the dot operator.

In [14]:

```
my_robot = RobotTurtle("T2", 2)

print(f'Robot {my_robot._name} initially at {my_robot._pos}')
for _ in range(4):
    my_robot.move('up')
    print(f'Robot {my_robot._name} now at {my_robot._pos}')
    my_robot.move('right')
    print(f'Robot {my_robot._name} now at {my_robot._pos}')
```

```
Robot T2 initially at (0, 0)
Robot T2 now at (0, 2)
Robot T2 now at (2, 2)
Robot T2 now at (2, 4)
Robot T2 now at (4, 4)
Robot T2 now at (4, 6)
Robot T2 now at (6, 6)
Robot T2 now at (6, 8)
Robot T2 now at (8, 8)
```

Note:

- We create a new object with the name "T2" and speed of 2.
- We first printed its initial position by access `my_robot._pos` attribute.
- Then, we iterate four times. Since we don't use of the iteration variable, we make us of `_`.
- In the iteration, we move up and then move right. After each movement, we printed the position.

Encapsulation and Properties

One important concept of Object Oriented programming is called **Encapsulation**. The idea of encapsulation is that data should be bundled together with some methods to access it. The data itself should be hidden from those outside of the object. With encapsulation, the state of the object is hidden from those outside of the object. If anyone would like to change the state of the object or enquire about the state of the object, it has to do so using some **methods**.

Why would we want to have this encapsulation? One of the purpose is to make the object transparent. Anyone working with the object does not need to know how the state or the data inside the object is implemented. For example, we implement the position attribute in our Robot Turtle object as a tuple of two numbers. This assumes those assigning value to this position always assign a tuple with two numbers. What if they don't? Let's illustrate this with an example

If we let others access the attributes direclty, one can assign non number data into the position attribute, such as the following example.

In [15]:

```
my_robot._pos = "This is not supposed to be allowed"
print(my_robot._pos)
```

```
This is not supposed to be allowed
```

Such assignment should not be allowed in the first place. If it is allowed, then our `move()` method will produce an error now as shown by running the following cell.

In [16]:

```
my_robot.move("up")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-16-5345a1aa5909> in <module>
----> 1 my_robot.move("up")

<ipython-input-10-012bc05876e6> in move(self, direction)
      9     # Methods:
     10     def move(self, direction):
---> 11         update = {'up' : (self._pos[0], self._pos[1] + self._speed),
     12                   'down' : (self._pos[0], self._pos[1] - self._speed),
     13                   'left' : (self._pos[0] - self._speed, self._pos[1]),

TypeError: can only concatenate str (not "int") to str
```

Therefore, it is important that we do encapsulation. Encapsulation ensures that any access to the data should be done through some specific methods. There are two kinds of methods for this purpose:

- enquiry or *getter*: this method is used to get or enquire the state of the object
- modifier or *setter*: this method is used to modify or set the state of the object.

In Python, we do this using the concept of **property**. A *property* represents an attribute with its getter and setter. Let's rewrite the class using property this time. We are going to create two properties, one for `name` and the other one for `speed`. On the other hand, we wil create a property for position only with a getter. The reason is that we want position to be modified only by calling the `move()` method.

In [17]:

```
# Class definition
class RobotTurtle:
    # Attributes:
    def __init__(self, name, speed=1):
        self.name = name
        self.speed = speed
        self._pos = (0, 0)

    # property getter
    @property
    def name(self):
        return self._name

    # property setter
    @name.setter
    def name(self, value):
        if isinstance(value, str) and value != "":
            self._name = value

    # property getter
    @property
    def speed(self):
        return self._speed

    # property setter
    @speed.setter
    def speed(self, value):
        if isinstance(value, int) and value > 0:
            self._speed = value

    # property getter
    @property
    def pos(self):
        return self._pos

    # Methods:
    def move(self, direction):
        update = {'up' : (self.pos[0], self.pos[1] + self.speed),
                  'down' : (self.pos[0], self.pos[1] - self.speed),
                  'left' : (self.pos[0] - self.speed, self.pos[1]),
                  'right' : (self.pos[0] + self.speed, self.pos[1])}
        self._pos = update[direction]


    def tell_name(self):
        print(f"My name is {self.name}")
```

We define a property for `name` as follows:

```
    # property getter
        @property
        def name(self):
            return self._name

        # property setter
        @name.setter
        def name(self, value):
            if isinstance(value, str) and value != "":
                self._name = value
```

Note:

- We use the syntax `@property` to define a getter with the name `name`. This is what is called as **decorator** in Python. A decorator allows you to modify the function defined in the line just after it. In our case, it changes the method `def name(self)` into a **getter** method for a property called `name`.
- The setter is defined using a decoratory `@name.setter`. In this setter method, we ensure that only those of the type `str` and not empty string can be assigned to the attribute `_name`.
- This setter will be called in the `__init__()` since the argument is assigned to the **property** `name` and not to the **attribute** `_name`, i.e. `self.name = name`.

The property for the `speed` is defined similarly.

```
    # property getter
        @property
        def speed(self):
            return self._speed

        # property setter
        @speed.setter
        def speed(self, value):
            if isinstance(value, int) and value > 0:
                self._speed = value
```

Note:

- The setter decorator is `@speed.setter` where the name before the dot is the name of the *property*.
- The setter ensures that only integer greater than 0 can be assigned to the attribute `_speed`.

Let's see some examples on how to use the properties.

In [18]:

```
# this is to create a new object with property, make sure you run the cell with the class definiti
on first
my_robot = RobotTurtle("T4")
```

In [19]:

```
# enquire name and speed
print(my_robot.name)
print(my_robot.speed)
```

```
T4
1
```

Notice that you use the property name, which are `name` and `speed` respectively instead of its attributes name, i.e. `__name` and `__speed`. This access calls the **getter** method of the respective properties.

Moreover, you can also change the value using the assignment operator which will call the **setter** method.

In [20]:

```
my_robot.name = "T4new"
print(my_robot.name)
my_robot.name = ""
print(my_robot.name)
```

```
T4new
T4new
```

Notice that in the second assignment, the name is not assigned to an empty string. It remains as `T4new`. The reason is that our setter only assigns the value if the value is a string and non-empty. Similarly, we can see the same behaviour for speed property.

In [21]:

```
my_robot.speed = 2
print(my_robot.speed)
my_robot.speed = -2
print(my_robot.speed)
```

```
2
2
```

Notice that the second assignment to -2 did not go through because of our setter method's checking.

On the other hand, we do not have any setter for position. The reason is that we want position to always start from `(0, 0)` and it can only change its position through the method `move()`. Note, however, that we are using a **single leading underscore** as a convention for people not to touch it. We can still enquire the position using the property's getter.

In [22]:

```
print(my_robot.pos)
```

```
(0, 0)
```

To change its position, it should call the `move()` method.

In [23]:

```
my_robot.move("up")
my_robot.move("up")
print(my_robot.pos)
```

```
(0, 4)
```

Note that we use the **properties**'s names `self.pos` and `self.speed` in updating the attribute `_pos` and `_speed`. See the `move()` method.

```
    def move(self, direction):
            update = {'up' : (self.pos[0], self.pos[1] + self.speed),
                      'down' : (self.pos[0], self.pos[1] - self.speed),
                      'left' : (self.pos[0] - self.speed, self.pos[1]),
                      'right' : (self.pos[0] + self.speed, self.pos[1])}
            self._pos = update[direction]
```

You can actually still access the attributes since Python does not have a concept of private attribute. This is how you access the attributes with a double leading underscore in its name.

In [24]:

```
my_robot._pos
```

Out[24]:

```
(0, 4)
```

But it is a convention in Python that when you use a single leading underscore, people should not touch it directly. On the other hand, one can also use a **double leading underscores**. This allows Name Mangling that prevents accidental overloading of methods and name conflicts when you extend a class.

methods and name conflicts when you extend a class.

In summary on the use of leading underscore for attribute's name:

- When in doubt, leave it "public". This means that we should not add anything to obscure the name of your class' attribute.
- If you really want to send the message "Can't touch this!" to your users, the usual way is to precede the variable with one underscore. This is just a convention, but people understand it and take double care when dealing with such stuff.
- The double underscore magic is used mainly to avoid accidental overloading of methods and name conflicts with superclasses' attributes. It can be quite useful if you write a class that is expected to be extended many times. We will talk about inheritance to extend a class in the subsequent lessons.

The above summary are taken from this article.

Computed Property

Both `name` and `speed` are what is commonly called **stored** properties. For each stored property there is a corresponding attribute. We can also create what is called **computed** property. A computed property retrieves its value from some other attributes and does not have a setter. To illustrate, let's create a new user-defined object called `Coordinate`.

In [25]:

```
import math

class Coordinate:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    @property
    def distance(self):
        return math.sqrt(self.x * self.x + self.y * self.y)
```

In the above class, we have two attributes `x` and `y`. We do not create any properties for these attributes for simplicity. Python encourages simplicity anway. But here, we create a computed property called `distance`. This property returns the distance of the current x and y from its origin (0, 0). We can test by instantiating the object and assign some values to its attributes.

In [26]:

```
# object instantiation
p1 = Coordinate(3, 4)
print(p1.x, p1.y)
print(p1.distance)
```

```
3 4
5.0
```

The last line prints the computed property `distance` which is computed from the two attributes `x` and `y`. Notice here that `distance` is printed without parenthesis and so it is not a **method** but rather a **property**.

So we may ask when should we use a method that returns a value and when to use a computed property. Here are some considerations:

- A method can have arguments. This means that if your returned value requires some input other than the attributes of its object, you must use a method rather than a computed property.
- A method describes an action. If the code performs some actions and return the output of that action, then a method is more suitable.

So when should we use a computed property?

- When the property describes some intrinsic quality of the object. Property is similar in many ways to attribute and it is usually a "noun". It should describes some kind of property of the object rather than some action that the object can do.
- When the computation is simple and cheap. We should prefer property for simple values you can get by doing a quick calculation. Distance property in the example above is a good example of this.
- When you can compute the value only with the object's attributes. Remember that getter of a property does not take any other argument besides `self`. This means that the computed value must be obtained only from the object's attributes.

Composition

An object can be composed of other objects. For example, we have seen that our `RobotTurtle` object is made up of other objects such as `str` for its name, `int` for its speed and tuple for its position. We can also compose an object from other **user-defined** object. For example, instead of using a tuple for its position, our Robot Turtle class can make use of the `Coordinate` class.

In [28]:

```
# Class definition
class RobotTurtle:
    # Attributes:
    def __init__(self, name, speed=1):
        self.name = name
        self.speed = speed
        self._pos = Coordinate(0, 0)

    # property getter
    @property
    def name(self):
        return self._name

    # property setter
    @name.setter
    def name(self, value):
        if isinstance(value, str) and value != "":
            self._name = value

    # property getter
    @property
    def speed(self):
        return self._speed

    # property setter
    @speed.setter
    def speed(self, value):
        if isinstance(value, int) and value > 0:
            self._speed = value

    # property getter
    @property
    def pos(self):
        return self._pos

    # Methods:
    def move(self, direction):
        update = {'up' : Coordinate(self.pos.x, self.pos.y + self.speed),
                  'down' : Coordinate(self.pos.x, self.pos.y - self.speed),
                  'left' : Coordinate(self.pos.x - self.speed, self.pos.y),
                  'right' : Coordinate(self.pos.x + self.speed, self.pos.y)}
        self._pos = update[direction]


    def tell_name(self):
        print(f"My name is {self.name}")
```

We made two main changes. First, in the `__init__()` instead of initializing to a tuple, we instantiate an object `Coordinate()`.

```
    def __init__(self, name, speed=1):
        self.name = name
        self.speed = speed
        self._pos = Coordinate(0, 0)
```

The initial position is still at (0, 0) but now the type is no longer a tuple, but rather, a `Coordinate` class. The second change is on the `move()` method.

```
    def move(self, direction):
        update = {'up' : Coordinate(self.pos.x, self.pos.y + self.speed),
                  'down' : Coordinate(self.pos.x, self.pos.y - self.speed),
                  'left' : Coordinate(self.pos.x - self.speed, self.pos.y),
                  'right' : Coordinate(self.pos.x + self.speed, self.pos.y)}
        self._pos = update[direction]
```

Instead of using indices like `self.pos[0]` and `self.pos[1]`, we now use the dot operator with its attribute names like `self.pos.x` and `self.pos.y`. This is much a clearer and easy to read as compared to using indices. Moreover, instead of using a tuple, we instantiate `Coordinate()` object as the value of the dictionary `update`.

We can now create the object and test our new class as follows.

In [20]:

```
my_robot = RobotTurtle("T with Coordinate")
print(my_robot.pos)
```

```
<__main__.Coordinate object at 0x7fca20dccf10>
```

Notice that now `pos` is a `Coordinate` object. We can access its attributes as usual.

In [21]:

```
print(my_robot.pos.x, my_robot.pos.y)
```

```
0 0
```

We can move the robot using the `move()` method.

In [22]:

```
my_robot.move("right")
my_robot.move("down")
print(my_robot.pos.x, my_robot.pos.y)
```

```
1 -1
```

Special Methods

Some methods' name in Python are special and can be overridden. One example of special method that you have encountered is `__init__()` method. This method is always called during object instantiation. There are many other special methods, but for now, we will introduce one more, which is the `__str__()` method. This method is called when Python tries to convert the object to an `str` object. One common instance of this is when you print the object.

If we print the `Coordinate()` object, we will see the following output.

In [23]:

```
p1 = Coordinate(2, 3)
print(p1)
```

```
<__main__.Coordinate object at 0x7fc9e077df90>
```

Python basically does not understand how to print a `Coordinate()`. But we can tell Python how to convert this object into an `str` which Python can display into the screen. Let's override the method `__str__()`.

In [24]:

```
import math

class Coordinate:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    @property
    def distance(self):
        return math.sqrt(self.x * self.x + self.y * self.y)

    def __str__(self):
```

```
        return f"({self.x}, {self.y})"
```

In the above method `__str__()` we return a string whenever Python tries to convert this object into a string. Once we define this special method, we can print a `Coordinate` object.

In [25]:

```
p1 = Coordinate(2, 3)
print(p1)
```

```
(2, 3)
```

Once Coordinate has this method, it can be used whenever the object has some `Coordinate` attributes. For example, we can print our robot position simply by doin gthe following.

In [26]:

```
my_robot = RobotTurtle("T with Coordinate")
print(my_robot.pos)
```

```
(0, 0)
```

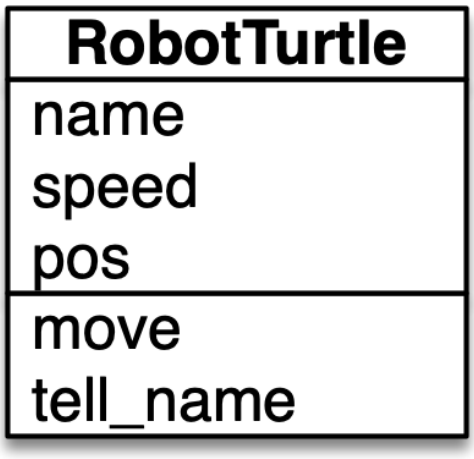
Recall, that previously you have to specify it as

```
    print(my_robot.pos.x, my_robot.pos.y)
```

But now it is no longer necessary and Python knows how to convert your `Coordinate` object into a string which can be displayed into the standard output.

UML Diagram

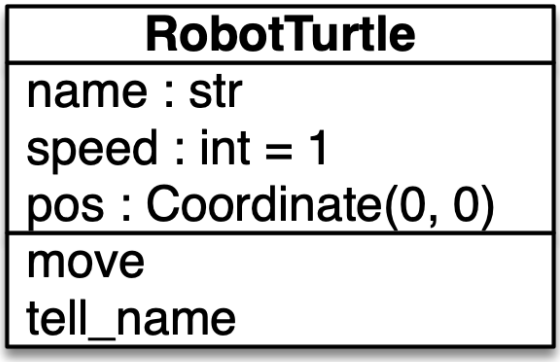
In designing Object Oriented programs, we usually use a UML diagram. UML stands for *Unified Modeling Language* and it gives some specifications how to represent the classes visually. For example, our `RobotTurtle` class is drawn as the following UML class diagram.



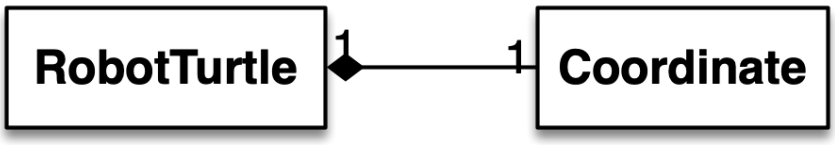
The UML Class diagram consists of three compartment:

- The first compartment on the top: this specifies the class name
- The second compartment in the middle: this lists down all the properties and attributes
- The third compartment at the bottom: this lists down all the methods

Sometime, it is useful to identify the property's type especially when there is a case of composition as in our `pos` property. In this case, we know that `pos` is of the type `Coordinate`. This is drawn in UML diagram as follows.



UML diagram also allows us to specify the relationship between different classes. For example, `RobotTurtle` and `Coordinate` relationship can be drawn as shown below.



In this diagram, we see that one `RobotTurtle` can have one `Coordinate`. This is a specific kind of *association* relationship called **composition**. This means that `RobotTurtle` is composed of a `Coordinate`. When the object `RobotTurtle` is destroyed, the `Coordinate` object associated with it is also destroyed. There are other kinds of relationship which we will introduce along the way.