# Lecture 15 - More About OOP

## Computer Programming

Samia Hilal

Marianopolis College

April 25, 2022

# Programming with classes and objects

- OOP works by defining *classes*. Each class is a template from which we can create many *objects*.
- Objects are *mutable*!
- Class methods are called like functions.
- The `'.'` operator is used to identify the attribute whose value we want to use or modify.
- The associated object is passed as the first argument to the method (`self`).
- We can use special methods to better integrate our new class with Python features, especially Python operators.

# Information hiding

- Code using our classes is able to “see” the attributes of our class.
- As a result, client code can inspect or change the attributes in an object.
- Python provides some mechanisms to “hide” these attributes.
- Most basic: two leading underscores.
- More advanced: property decorators.
- We will be using the basic method only!

# Leading underscores

- Any attribute whose name begins with two underscores is treated specially:

```
class point(object):
    def __init__(self, new_x, new_y):
        self.__x = new_x
        self.__y = new_y
    def distance(self, pt):
        dx = self.__x - pt.__x
        dy = self.__y - pt.__y
        return (dx * dx + dy * dy) ** 0.5
```

- Code within the class definition can access the attributes as named.

# Leading underscores, continued

- Code outside the class definition cannot access the same name:

```
class point(object):
    def __init__(self, new_x, new_y):
        self.__x = new_x
        self.__y = new_y
    def distance(self, pt):
        dx = self.__x - pt.__x
        dy = self.__y - pt.__y
        return (dx * dx + dy * dy) ** 0.5

>>> a = point(1, 1)
>>> print(a.__x)
'point' object has no attribute '__x'
```

# Name 'mangling'

- The attribute with leading underscores is still there, but Python has “mangled” its name:

```
>>> a = point(5, 6)
>>> print(a.__x)
'point' object has no attribute '__x'
>>> print(a._point__x, a._point__y)
5 6
```

- The mangled version of the name just has an underscore and the class name prepended to it.
- We *can* access it under this altered name.

# Class variables

- Variables defined in a class become attributes shared among all instances of the class.
- Only a single copy is created.
- They can be referenced using the class name:

  ```
  ClassName.class_variable
  ```

- They can *also* be referenced from each instance of the class.
- They are generally mutable!

# Class variable example

```
class point(object):
    ’’’... documentation ...’’’
    tolerance = 0.1

    def is_close_to(self, pt):
        temp = self.distance(pt)
        return temp < point.tolerance

>>> a = point(3, 4)
>>> print(a.x, a.y, a.tolerance)
3 4 0.1
>>> print(point.tolerance)
0.1
>>> print(point.x)
’point’ has no attribute ’x’
```

# Modifying a class variable

```
>>> a,b = point(3, 4), point(5, 12)
>>> print(a.x, a.y, a.tolerance)
3 4 0.1
>>> print(b.x, b.y, b.tolerance)
5 12 0.1
>>> point(0, 0).is_close_to(a)
False
>>> point.tolerance *= 100
>>> point(0, 0).is_close_to(a)
True
>>> print(a.x, a.y, a.tolerance)
3 4 10.0
>>> print(b.x, b.y, b.tolerance)
5 12 10.0 # Change is visible everywhere.
```

# Special methods

- There are many special methods available. We will only study a few.

- `__add__`
- `__sub__`
- `__mul__`
- `__truediv__`
- `__floordiv__`
- `__mod__`
- `__eq__`
- `__ne__`
- `__lt__`
- `__gt__`
- `__le__`
- `__ge__`
- `__init__`
- `__repr__`
- `__str__`
- `__bool__`
- `__bytes__`
- `__format__`
- `__len__`

# Example:`__str__` & `__repr__` methods

- The `__str__` method is used to convert to a human-readable format, as in `print()`

```
def __str__(self):
    fmt = '({}, {})'
    return fmt.format(self.x, self.y)
```

- The `__repr__` method works in the same way but more general.
- If the `__repr__` is defined, the `__str__` method is ignored.
- With `__repr__`, you view object without print.

```
>>> x = point(1,2)
>>> x
(1, 2)
```

# Example: the __eq__ method.

- If our class defines a __eq__() method, Python uses it to implement the == operator.
- The method takes 2 arguments, both points
- Our method should return either True or False.

```
def __eq__(self, pt):
 return self.x == pt.x and self.y == pt.y
```

- Now we can do this:

```
>>> a = point(0, 0)
>>> b = point(1, 1)
>>> print(a == b)
False
```

# Example: the __add__ method.

- If our class defines a `__add__()` method, Python uses it to implement the + operator.
- The method takes 2 arguments, both points.
- Our method should return a new point.

```
def __add__(self, pt):
    return point(self.x + pt.x,
                 self.y + pt.y)
```

- Now this code will work:

```
>>> a = point(1, 2)
>>> b = point(2, 1)
>>> print(a + b)
(3, 3)
```

# The `isinstance()` builtin function.

- As the name implies, `isinstance()` checks that an object is of a particular type or class.

```
>>> x, y, z = 1, 2.0, ’String’
>>> isinstance(x, int)
True
>>> isinstance(y, float)
True
>>> isinstance(z, str)
True
>>> isinstance(z, float)
False
>>> isinstance(True, bool)
True
```