Lecture 22: Object Oriented Programming
Dunder Methods

Comp 102

Forman Christian University

Recap

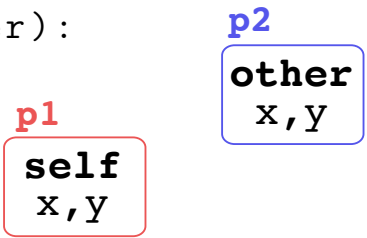
Suppose you have the following point class:

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5      def get_x(self):
6          return self.x
7      def get_y(self):
8          return self.y
9      def print_point(self):
10         print(f"({self.x}, {self.y})")
```

You Try!

Write a method `distance` that takes another point as an argument and returns the distance between the two points.

```
1 class Point:
2     ...
3     def distance(self, other):
4         ...
```



You Try!

```
1  class Point:
2      ...
3      def distance(self, other):
4          return ((self.x - other.x)**2 +
5                  (self.y - other.y)**2)**0.5
6
7
8
9
```

You Try!

```
1  class Point:
2      ...
3      def distance(self, other):
4          return ((self.x - other.x)**2 +
5                  (self.y - other.y)**2)**0.5
6
7
8
9
```

Abstraction Violation

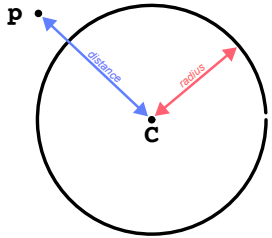
You Try!

Abstraction Violation

```
1  class Point:
2      ...
3      def distance(self, other):
4          return ((self.x - other.x)**2 +
5                  (self.y - other.y)**2)**0.5
6
7      def distance(self, other):
8          return ((self.get_x() - other.get_x())**2 +
9                  (self.get_y() - other.get_y())**2)**0.5
```

Circle class

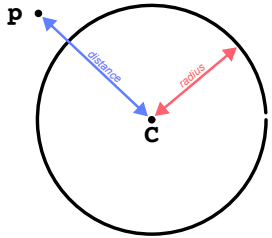
Attempt # 1:




```
1 class Circle:
2     def __init__(self,x,y,radius):
3         self.x = x
4         self.y = y
5         self.radius = radius
```

Circle class

Attempt # 1:




```
1 class Circle:
2     def __init__(self,x,y,radius):
3         self.x = x
4         self.y = y
5         self.radius = radius
```

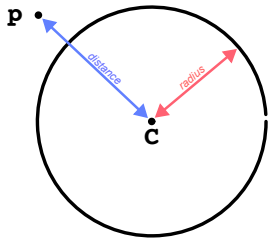
Abstraction violation

could've used 'Point' class

Using Objects to Build Other Objects

Circle class

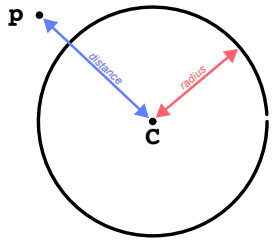
Attempt # 2:




```
1 class Circle:
2   def __init__(self,center,radius):
3     self.center = center #object
4     self.radius = radius
5
6 p1 = Point(0,0)
7 c1 = Circle(p1,5)
```

Circle class

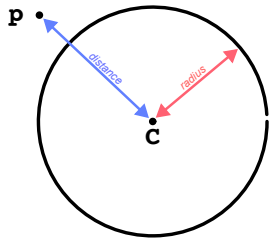
Attempt # 2:




```
1 class Circle:
2   def __init__(self,center,radius):
3     self.center = center #object
4     self.radius = radius
5
6 p1 = Point(0,0)
7 c1 = Circle(p1,5)
```

Circle class

Attempt # 2:




```
1  class Circle:
2    def __init__(self,center,radius):
3      self.center = center #object
4      self.radius = radius
5
6  p1 = Point(0,0)
7  c1 = Circle(p1,5)
```

c1 p1

You Try!

Add code to the `__init__` method to check that the type of center is a `Point` object and the type of radius is an int. If either are not these types, raise a `ValueError`.

```
1 def __init__(self, center, radius):
2     self.center = center
3     self.radius = radius
```

You Try!

Add code to the `__init__` method to check that the type of center is a `Point` object and the type of radius is an int. If either are not these types, raise a `ValueError`.

```
1 def __init__(self, center, radius):
2     self.center = center
3     self.radius = radius
```

Solution:

```
1 def __init__(self, center, radius):
2     if not isinstance(center, Point):
3         raise ValueError("center must be a Point object")
4     if not isinstance(radius, int):
5         raise ValueError("radius must be an int")
6     self.center = center
7     self.radius = radius
```

Adding more Methods to the Circle class

```
1  class Circle(object):
2      def __init__(self, center, radius):
3          self.center = center
4          self.radius = radius
5
6      def is_inside(self, point):
7          """ Returns True if point is in self, False otherwise """
8          return point.distance(self.center) < self.radius
9
10 center = Coordinate(2, 2)
11 circle = Circle(center, 2)
12 p = Coordinate(1,1)
13 print(circle.is_inside(p)) # method only works with circle object
```

You Try!

Are these two methods in the Circle class functionally equivalent?

```
1  class Circle(object):
2      def __init__(self, center, radius):
3          self.center = center
4          self.radius = radius
5
6      def is_inside1(self, point):
7          return point.distance(self.center) < self.radius
8
9      def is_inside2(self, point):
10         return self.center.distance(point) < self.radius
```

Dunder Methods

What are Dunder Methods?

What are Dunder Methods?

dunder = Double **UNDER**score

What are Dunder Methods?

dunder = **D**ouble **UNDER**score

Methods that start and end with __

What are Dunder Methods?

dunder = **D**ouble **UNDER**score

Methods that start and end with `__`

`__init__` you already know this one!

What are Dunder Methods?

dunder = **D**ouble **UNDER**score

Methods that start and end with `__`

`__init__` you already know this one!

`__str__` string representation

What are Dunder Methods?

dunder = **D**ouble **UNDER**score

Methods that start and end with __

`__init__`	you already know this one!
`__str__`	string representation
`__eq__`	equality comparison

What are Dunder Methods?

dunder = Double **UNDER**score

Methods that start and end with `__`

`__init__`	you already know this one!
`__str__`	string representation
`__eq__`	equality comparison
`__add__`	addition operator

What are Dunder Methods?

dunder = **D**ouble **UNDER**score

Methods that start and end with `__`

`__init__`	you already know this one!
`__str__`	string representation
`__eq__`	equality comparison
`__add__`	addition operator
`__lt__`	less-than comparison

What are Dunder Methods?

dunder = **D**ouble **UNDER**score

Methods that start and end with `__`

`__init__`	you already know this one!
`__str__`	string representation
`__eq__`	equality comparison
`__add__`	addition operator
`__lt__`	less-than comparison

Python calls these **automatically** *behind the scenes!*

The Problem

```
1 p1 = Point(1, 2)
2 p2 = Point(1, 2)
```

The Problem

```
1  p1 = Point(1, 2)
2  p2 = Point(1, 2)
```

```
>>> print(p1)
```

```
<__main__.Point object at 0x7f3b...>
```

The Problem

```
1  p1 = Point(1, 2)
2  p2 = Point(1, 2)
```

```
>>> print(p1)
```

```
<__main__.Point object at 0x7f3b...>
```

```
>>> p1 == p2
```

```
False
```

— *even though same coordinates!*

The Problem

```
1  p1 = Point(1, 2)
2  p2 = Point(1, 2)
```

```
>>> print(p1)
```

```
<__main__.Point object at 0x7f3b...>
```

```
>>> p1 == p2
```

```
False
```

— *even though same coordinates!*

```
>>> p1 + p2
```

```
TypeError
```

Big Idea

Dunder methods let you control how your objects behave with Python's **built-in operations**.

__str__

`__str__` — Custom String Representation

```
1 p1 = Point(1, 2)
2 print(p1)
```

Output:

```
<__main__.Point object at 0x7f3b2c>
```

`__str__` — Custom String Representation

```
1 p1 = Point(1, 2)
2 print(p1)
```

Output:

```
<__main__.Point object at 0x7f3b2c>
```

Ugly and unhelpful!

The Fix: Define `__str__`

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6
7
8
9
10
11
```

The Fix: Define `__str__`

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __str__(self):
7          return f"({self.x}, {self.y})"
8
9
10
11
```

The Fix: Define `__str__`

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __str__(self):
7          return f"({self.x}, {self.y})"
8
9  p1 = Point(1, 2)
10 print(p1)
11 str(p1)
```

```
(1, 2)
"(1, 2)"
```

Big Idea

When you call `print()` on an object, Python automatically calls its `__str__()` method.

You Try!

Write a `__str__` method for the `Circle` class that returns:
`"Circle at (2, 3) with radius 5"`

```
1  class Circle:
2      def __init__(self, center, radius):
3          self.center = center
4          self.radius = radius
5
6      def __str__(self):
7          ...
8
```

You Try!

Write a `__str__` method for the `Circle` class that returns:
`"Circle at (2, 3) with radius 5"`

```
class Circle:
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def __str__(self):

        return f"Circle at {self.center} with radius {self.radius}"
```

This works because `Point` already has `__str__` defined!

__eq__

__eq__ — Equality Comparison

```
p1 = Point(1, 2)
p2 = Point(1, 2)
print(p1 == p2)
```

__eq__ — Equality Comparison

```
1 p1 = Point(1, 2)
2 p2 = Point(1, 2)
3 print(p1 == p2)
```

Output: `False`

__eq__ — Equality Comparison

```
p1 = Point(1, 2)
p2 = Point(1, 2)
print(p1 == p2)
```

Output: `False`

By default, == checks if they are the **same object** in memory (identity), not if they have the same **values**.

The Fix: Define __eq__

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6
7
8
9
10
11
```

The Fix: Define __eq__

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __eq__(self, other):
7          return self.x == other.x and self.y == other.y
8
9
10
11
```

The Fix: Define __eq__

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __eq__(self, other):
7          return self.x == other.x and self.y == other.y
8
9  p1 = Point(1, 2)
10 p2 = Point(1, 2)
11 print(p1 == p2)
```

True

Big Idea

`__eq__` lets you define what "equal" **means** for your objects.

You Try!

Write an `__eq__` method for the `Circle` class. Two circles are equal if they have the same center and the same radius.

```
1  class Circle:
2      def __init__(self, center, radius):
3          self.center = center
4          self.radius = radius
5
6      def __eq__(self, other):
7          ...
8
```

You Try!

Write an `__eq__` method for the `Circle` class. Two circles are equal if they have the same center and the same radius.

```
1  class Circle:
2      def __init__(self, center, radius):
3          self.center = center
4          self.radius = radius
5
6      def __eq__(self, other):
7
           return self.center == other.center and self.radius == other.
               radius
```

This works because `Point` already has `__eq__` defined!

Operator Overloading

What if operators worked with our objects?

```
1 p1 = Point(1, 2)
2 p2 = Point(3, 4)
3 p3 = p1 + p2
```

What if operators worked with our objects?

```
1 p1 = Point(1, 2)
2 p2 = Point(3, 4)
3 p3 = p1 + p2
```

```
TypeError: unsupported operand type(s) for +
```

What if operators worked with our objects?

```
1 p1 = Point(1, 2)
2 p2 = Point(3, 4)
3 p3 = p1 + p2
```

```
TypeError: unsupported operand type(s) for +
```

We can fix this with **operator overloading***!*

How Python Operators Work

When you use an operator, Python translates it to a dunder method call:

```
a + b  ⟶  a.__add__(b)
```

How Python Operators Work

When you use an operator, Python translates it to a dunder method call:

```
a + b  ⟶  a.__add__(b)
a - b  ⟶  a.__sub__(b)
```

How Python Operators Work

When you use an operator, Python translates it to a dunder method call:

```
a + b  ⟶  a.__add__(b)
a - b  ⟶  a.__sub__(b)
a * b  ⟶  a.__mul__(b)
```

How Python Operators Work

When you use an operator, Python translates it to a dunder method call:

```
a + b  ⟶  a.__add__(b)
a - b  ⟶  a.__sub__(b)
a * b  ⟶  a.__mul__(b)
a == b ⟶  a.__eq__(b)
```

How Python Operators Work

When you use an operator, Python translates it to a dunder method call:

```
a + b   ⟶  a.__add__(b)
a - b   ⟶  a.__sub__(b)
a * b   ⟶  a.__mul__(b)
a == b  ⟶  a.__eq__(b)
a < b   ⟶  a.__lt__(b)
```

How Python Operators Work

When you use an operator, Python translates it to a dunder method call:

```
 a + b  ⟶  a.__add__(b)
 a - b  ⟶  a.__sub__(b)
 a * b  ⟶  a.__mul__(b)
a == b  ⟶  a.__eq__(b)
 a < b  ⟶  a.__lt__(b)
str(a)  ⟶  a.__str__()
```

__add__ — The Addition Operator

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __str__(self):
7          return f"({self.x}, {self.y})"
8
9      def __add__(self, other):
10
11
12
13
14 p1 = Point(1, 2)
15 p2 = Point(3, 4)
16 p3 =
17 prir
```

__add__ — The Addition Operator

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __str__(self):
7          return f"({self.x}, {self.y})"
8
9      def __add__(self, other):
10         new_x = self.x + other.x
11         new_y = self.y + other.y
12         return Point(new_x, new_y)
13
14 p1 = Point(1, 2)
15 p2 = Point(3, 4)
16 p3 =
17 prir
```

__add__ — The Addition Operator

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __str__(self):
7          return f"({self.x}, {self.y})"
8
9      def __add__(self, other):
10         new_x = self.x + other.x
11         new_y = self.y + other.y
12         return Point(new_x, new_y)
13
14 p1 = Point(1, 2)
15 p2 = Point(3, 4)
16 p3 = p1 + p2
17 print(p3)
```

Behind the Scenes

When you write:

```
p3 = p1 + p2
```

Behind the Scenes

When you write:

```
p3 = p1 + p2
```

Python translates it to:

```
p3 = p1.__add__(p2)
```

Big Idea

Operator Overloading lets you define how +, −, ==, etc. work with **your** objects.

You Try!

Write a `__sub__` method for the `Point` class that subtracts one point from another.

```
class Point:
    ...
    def __sub__(self, other):
        ...

p1 = Point(5, 7)
p2 = Point(2, 3)
p3 = p1 - p2
print(p3)          # (3, 4)
```

__lt__ and Sorting

Can we sort a list of Points?

```
1 points = [Point(3,4),
2           Point(1,1),
3           Point(2,2)]
4 sorted(points)
```

Can we sort a list of Points?

```
1 points = [Point(3,4),
2           Point(1,1),
3           Point(2,2)]
4 sorted(points)
```

```
TypeError: '<' not supported between instances
of 'Point'
```

Can we sort a list of Points?

```
1 points = [Point(3,4),
2           Point(1,1),
3           Point(2,2)]
4 sorted(points)
```

```
TypeError: '<' not supported between instances
of 'Point'
```

Python's `sorted()` needs to compare objects using <
We need to define `__lt__`!

`__lt__` — Less Than

Compare points by their **distance from the origin**:

```
1  class Point:
2      ...
3      def __lt__(self, other):
4          d1 = (self.x**2 + self.y**2)**0.5
5          d2 = (other.x**2 + other.y**2)**0.5
6          return d1 < d2
7
8  p1 = Point(1, 1)    # distance = 1.41
9  p2 =                           = 5.0
10 pri
```

`__lt__` — Less Than

Compare points by their **distance from the origin**:

```
1  class Point:
2      ...
3      def __lt__(self, other):
4          d1 = (self.x**2 + self.y**2)**0.5
5          d2 = (other.x**2 + other.y**2)**0.5
6          return d1 < d2
7
8  p1 = Point(1, 1)    # distance = 1.41
9  p2 = Point(3, 4)    # distance = 5.0
10 print(p1 < p2)            True
```

Now `sorted()` works!

```
1 points = [Point(3,4),
2           Point(1,1),
3           Point(2,2)]
4
5 for p in sorted(points):
6     print(p)
```

Now `sorted()` works!

```
1 points = [Point(3,4),
2           Point(1,1),
3           Point(2,2)]
4
5 for p in sorted(points):
6     print(p)
```

```
(1, 1)
(2, 2)
(3, 4)
```

Big Idea

Define `__lt__` and Python's `sorted()` works **automatically** with your objects!

Dunder Methods Reference

- Define them with **double underscores** before/after

```
__add__(self, other)    →    self + other
__sub__(self, other)    →    self - other
__mul__(self, other)    →    self * other
__truediv__(self, other) →   self / other
__eq__(self, other)     →    self == other
__lt__(self, other)     →    self < other
__len__(self)           →    len(self)
__str__(self)           →    print(self)
__float__(self)         →    float(self) i.e cast
__pow                   →    self**other
```

Summary

Method	Triggered by	Purpose
`__init__`	`Point(1,2)`	Constructor
`__str__`	`print(p)`	String form
`__eq__`	`p1 == p2`	Equality
`__add__`	`p1 + p2`	Addition
`__sub__`	`p1 - p2`	Subtraction
`__lt__`	`p1 < p2` / `sorted()`	Comparison

Questions?