MORE PYTHON CLASS METHODS
(downloads slides and .py files to follow along)
6.100L Lecture 18
Ana Bell
IMPLEMENTING THE CLASS vs USING THE CLASS

- Write code from two different perspectives

**Implementing** a new object type with a class
- **Define** the class
- Define **data attributes** (WHAT IS the object)
- Define **methods** (HOW TO use the object)

**Using** the new object type in code
- Create **instances** of the object type
- Do **operations** with them

Class abstractly captures **common** properties and behaviors

Instances have **specific values** for attributes
RECALL THE COORDINATE CLASS

- Class **definition** tells Python the **blueprint** for a type Coordinate

```python
class Coordinate(object):
    """ A coordinate made up of an x and y value """
    def __init__(self, x, y):
        """ Sets the x and y values """
        self.x = x
        self.y = y
    def distance(self, other):
        """ Returns euclidean dist between two Coord obj """
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```
Methods are functions that **only work with objects of this type**.

```python
class Coordinate(object):
    """ A coordinate made up of an x and y value """
    def __init__(self, x, y):
        """ Sets the x and y values """
        self.x = x
        self.y = y
    def distance(self, other):
        """ Returns euclidean dist between two Coord obj """
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def to_origin(self):
        """ always sets self.x and self.y to 0,0 """
        self.x = 0
        self.y = 0
```
MAKING COORDINATE INSTANCES

- Creating **instances** makes actual Coordinate **objects in memory**
- The objects can be **manipulated**
  - Use **dot notation** to call methods and access data attributes

```python
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(f"c's x is {c.x} and origin's x is {origin.x}"),
print(c.distance(origin))
c.to_origin()
print(c.x, c.y)
```

x data attr has a value of 3
y data attr has a value of 4
Method didn’t return anything,
just set c’s x and y to 0.
CLASS DEFINITION OF AN OBJECT TYPE vs INSTANCE OF A CLASS

- Class name is the **type**
  ```python
class Coordinate(object)
  ```
- Class is defined generically
  - Use `self` to refer to some instance while defining the class
    ```python
    (self.x - self.y)**2
    ```
  - `self` is a parameter to methods in class definition
- Class defines data and methods common across all instances

- Instance is **one specific object**
  ```python
  coord = Coordinate(1,2)
  ```
- Data attribute values vary between instances
  ```python
  c1 = Coordinate(1,2)
c2 = Coordinate(3,4)
  ```
  - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects

- Instance has the **structure of the class**
USING CLASSES TO BUILD OTHER CLASSES

- Example: use Coordinates to build Circles
- Our implementation will use 2 data attributes
  - Coordinate object representing the center
  - int object representing the radius
CIRCLE CLASS: DEFINITION and INSTANCES

class Circle(object):
    def __init__(self, center, radius):
        self.center = center  # Will be a Coordinate object
        self.radius = radius  # Will be an int

        center = Coordinate(2, 2)  # Data attributes, do not need to have the same names as params
        my_circle = Circle(center, 2)
YOU TRY IT!

- Add code to the init method to check that the type of center is a Coordinate obj and the type of radius is an int. If either are not these types, raise a ValueError.

```python
def __init__(self, center, radius):
    self.center = center
    self.radius = radius
```
class Circle(object):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
    def is_inside(self, point):
        """ Returns True if point is in self, False otherwise """
        return point.distance(self.center) < self.radius

center = Coordinate(2, 2)
my_circle = Circle(center, 2)
p = Coordinate(1,1)
print(my_circle.is_inside(p))
YOU TRY IT!

- Are these two methods in the Circle class functionally equivalent?

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

    def is_inside1(self, point):
        return point.distance(self.center) < self.radius

    def is_inside2(self, point):
        return self.center.distance(point) < self.radius
```
EXAMPLE: FRACTIONS

- Create a **new type** to represent a number as a fraction
- **Internal representation** is two integers
  - Numerator
  - Denominator
- **Interface** a.k.a. **methods** a.k.a **how to interact** with **Fraction objects**
  - Add, subtract
  - Invert the fraction
- Let’s write it together!
NEED TO CREATE INSTANCES

class SimpleFraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
MULTIPLY FRACTIONS

```python
class SimpleFraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
    def times(self, oth):
        top = self.num*oth.num
        bottom = self.denom*oth.denom
        return top/bottom
```

SimpleFraction objects so they each have

* num
* denom

Access num or denom to do the math
ADD FRACTIONS

class SimpleFraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d

    def plus(self, oth):
        top = self.num*oth.denom + self.denom*oth.num
        bottom = self.denom*oth.denom
        return top/bottom
LET’S TRY IT OUT

```python
f1 = SimpleFraction(3, 4)
f2 = SimpleFraction(1, 4)
print(f1.num)    # 3
print(f1.denom)  # 4
print(f1.plus(f2))  # 1.0
print(f1.times(f2))  # 0.1875
```
You try it!

Add two methods to invert fraction object according to the specs below:

class SimpleFraction(object):
    """ A number represented as a fraction """
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom
    def get_inverse(self):
        """ Returns a float representing 1/self """
        pass
    def invert(self):
        """ Sets self's num to denom and vice versa. Returns None. """
        pass

# Example:
f1 = SimpleFraction(3,4)
print(f1.get_inverse())  # prints 1.33333333 (note this one returns value)
f1.invert()               # acts on data attributes internally, no return
print(f1.num, f1.denom)   # prints 4 3
LET’S TRY IT OUT WITH MORE THINGS

```python
f1 = SimpleFraction(3, 4)
f2 = SimpleFraction(1, 4)
print(f1.num)  # 3
print(f1.denom)  # 4
print(f1.plus(f2))  # 1.0
print(f1.times(f2))  # 0.1875
print(f1)  # <__main__.SimpleFraction object at 0x00000234A8C41DF0>
print(f1 * f2)  # Error!
```

What if we want to keep as a fraction?
And what if we want to have print and * work as we would expect?
SPECIAL OPERATORS IMPLEMENTED WITH DUNDER METHODS

- +, -, ==, <, >, len(), print, and many others are shorthand notations
- Behind the scenes, these get replaced by a method!
  https://docs.python.org/3/reference/datamodel.html#basic-customization
- Can override these to work with your class
SPECIAL OPERATORS IMPLEMENTED WITH DUNDER METHODS

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

  ```python
  __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
  ```

  ... and others
PRINTING OUR OWN DATA TYPES
PRINT REPRESENTATION OF AN OBJECT

```python
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **Uninformative** print representation by default
- Define a `__str__` method for a class
- Python calls the `__str__` method when used with `print` on your class object
- You choose what it does! Say that when we print a `Coordinate` object, want to show

```python
>>> print(c)
<3,4>
```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
    def distance(self, other):
        x_diff_sq = (self.x - other.x)**2
        y_diff_sq = (self.y - other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return '<' + str(self.x) + ',' + str(self.y) + '>

def __str__(self):
    return str(self.x)
Can ask for the type of an object instance

```python
>>> c = Coordinate(3, 4)
>>> print(c)
<3,4>
>>> print(type(c))
<class '__main__.Coordinate'>
```

This makes sense since

```python
>>> print(Coordinate)
<class '__main__.Coordinate'>
>>> print(type(Coordinate))
<type 'type'>
```

Use `isinstance()` to check if an object is a `Coordinate`

```python
>>> print(isinstance(c, Coordinate))
True
```
EXAMPLE: FRACTIONS WITH DUNDER METHODS

- Create a **new type** to represent a number as a fraction
- **Internal representation** is two integers
  - Numerator
  - Denominator
- **Interface** a.k.a. **methods** a.k.a **how to interact** with Fraction objects
  - Add, sub, mult, div to work with +, -, *, /
  - Print representation, convert to a float
  - Invert the fraction
- Let’s write it together!
CREATE & PRINT INSTANCES

class Fraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d
    def __str__(self):
        return str(self.num) + "/" + str(self.denom)

Concatenation means that every piece has to be a str
LET’S TRY IT OUT

```python
f1 = Fraction(3, 4)
f2 = Fraction(1, 4)
f3 = Fraction(5, 1)
print(f1)  # 3/4
print(f2)  # 1/4
print(f3)  # 5/1
```

Ok, but looks weird!
YOU TRY IT!

- Modify the str method to represent the Fraction as just the numerator, when the denominator is 1. Otherwise its representation is the numerator then a / then the denominator.

```python
class Fraction(object):
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom
    def __str__(self):
        return str(self.num) + "/" + str(self.denom)

# Example:
a = Fraction(1,4)
b = Fraction(3,1)
print(a)   # prints 1/4
print(b)   # prints 3
```
IMPLEMENTING
+ - * /
float()
COMPARING METHOD vs. DUNDER METHOD

```
class SimpleFraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d

    def times(self, oth):
        top = self.num*oth.num
        bottom = self.denom*oth.denom
        return top/bottom

class Fraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d

    def __mul__(self, other):
        top = self.num*other.num
        bottom = self.denom*other.denom
        return Fraction(top, bottom)
```

When we use this method, Python evaluates and returns this expression, which creates a float.

Note: we are creating and returning a new instance of a Fraction.
LET'S TRY IT OUT

```python
a = Fraction(1,4)
b = Fraction(3,4)
print(a)
c = a * b
print(c)
```

1/4

3/16

Calls the \_\_mul\_\_ method behind the scenes. This method returns Fraction(3,16)

Uses \_\_str\_ for a Fraction object
CLASSES CAN HIDE DETAILS

- These are all equivalent
  
  ```python
  print(a * b)
  print(a.__mul__(b))
  print(Fraction.__mul__(a, b))
  ```

- Every operation in Python comes back to a method call

- The first instance makes clear the operation, without worrying about the internal details!

  **Abstraction at work**
BIG IDEA

Special operations we’ve been using are just methods behind the scenes.

Things like:
print, len
+, *, -, /, <, >, <=, >=, ==, !=
[]
and many others!
CAN KEEP BOTH OPTIONS BY ADDING A METHOD TO CAST TO A float

class Fraction(object):
    def __init__(self, n, d):
        self.num = n
        self.denom = d

......
    def __float__(self):
        return self.num/self.denom

c = a * b
print(c)  
print(float(c))
LETS TRY IT OUT SOME MORE

```python
a = Fraction(1,4)
b = Fraction(2,3)
c = a * b
print(c)  # 2/12
```

- Not quite what we might expect? It’s not reduced.
- Can we fix this?
class Fraction(object):
    ....
    def reduce(self):
        def gcd(n, d):
            while d != 0:
                (d, n) = (n%d, d)
            return n
        if self.denom == 0:
            return None
        elif self.denom == 1:
            return self.num
        else:
            greatest_common_divisor = gcd(self.num, self.denom)
            top = int(self.num/greatest_common_divisor)
            bottom = int(self.denom/greatest_common_divisor)
            return Fraction(top, bottom)

c = a*b
print(c)
print(c.reduce())
class Fraction(object):
    .........
    def reduce(self):
        def gcd(n, d):
            while d != 0:
                (d, n) = (n%d, d)
            return n
        if self.denom == 0:
            return None
        elif self.denom == 1:
            return self.num
        else:
            greatest_common_divisor = gcd(self.num, self.denom)
            top = int(self.num/greatest_common_divisor)
            bottom = int(self.denom/greatest_common_divisor)
            return Fraction(top, bottom)

Is this a good idea?
It does not return a Fraction so
we can no longer add or multiply
this by other Fractions
CHECK THE TYPES, THEY’RE DIFFERENT

```python
a = Fraction(4,1)
b = Fraction(3,9)
ar = a.reduce()
br = b.reduce()
print(ar, type(ar))  # 4 <class 'int'>
print(br, type(br))  # 1/3 <class '__main__.Fraction'>
c = ar * br
```

Error! It’s trying to multiply an `int` with a `Fraction`. We never defined how to do this – only a `Fraction` with another `Fraction`
YOU TRY IT!

- Modify the code to return a Fraction object when denominator is 1

```python
class Fraction(object):
    def reduce(self):
        def gcd(n, d):
            while d != 0:
                (d, n) = (n%d, d)
            return n
        if self.denom == 0:
            return None
        elif self.denom == 1:
            return self.num
        else:
            greatest_common_divisor = gcd(self.num, self.denom)
            top = int(self.num/greatest_common_divisor)
            bottom = int(self.denom/greatest_common_divisor)
            return Fraction(top, bottom)

# Example:
f1 = Fraction(5,1)
print(f1.reduce())   # prints 5/1 not 5
```

6.100L Lecture 18
WHY OOP and BUNDLING THE DATA IN THIS WAY?

- Code is **organized** and **modular**
- Code is easy to **maintain**
- It’s easy to **build upon** objects to make more complex objects

- **Decomposition and abstraction** at work with Python classes
  - Bundling data and behaviors means you can use objects consistently
  - Dunder methods are abstracted by common operations, but they’re just methods behind the scenes!