SI100B: Introduction to Information Science and Technology

Lecture 11

By Ana Bell of MIT

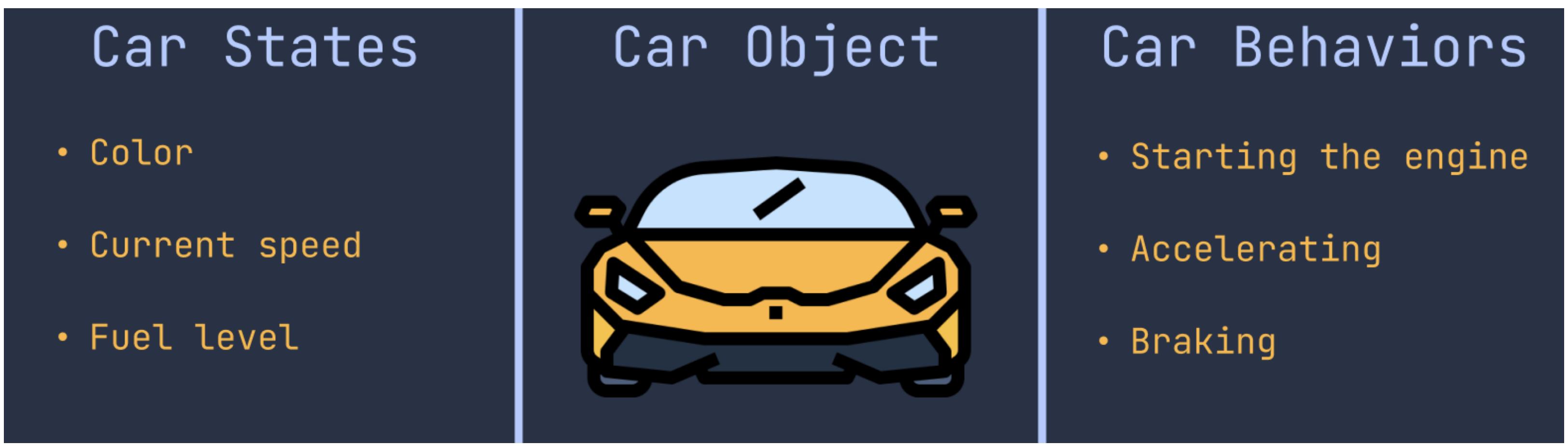
Object Oriented Programming

OBJECTS & TYPES

- **EVERYTHING IN PYTHON IS AN OBJECT**
  - Can create new objects of some type
  - Can manipulate objects
  - Can destroy objects
    - Explicitly using `del` or just “forget” about them
    - Python system will reclaim destroyed or inaccessible objects – called “garbage collection”
- **EVERY OBJECT HAS A TYPE**
  - This lecture: create new types with class

OBJECTS & TYPES

- Objects of a specific type have…
  - An internal representation
    - Through data attributes
  - An interface for interacting with object
    - Through methods (i.e., procedural attributes)
    - Defines behaviors but hides implementation

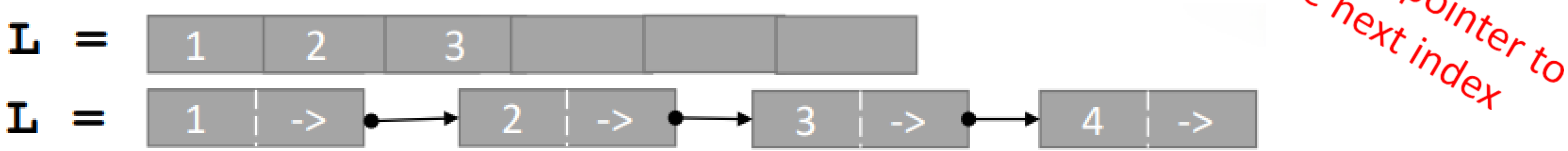


REAL-LIFE EXAMPLES

- Elevator: a box that can change floors
  - Represent using length, width, height, max_capacity, current_floor
  - Move its location to a different floor, add people, remove people
- Employee: a person who works for a company
  - Represent using name, birth_date, salary
  - Can change name or salary
- Queue at a store: first customer to arrive is the first one helped
  - Represent customers as a list of str names
  - Append names to the end and remove names from the beginning
- Stack of pancakes: first pancake made is the last one eaten
  - Represent stack as a list of str
  - Append pancake to the end and remove from the end

EXAMPLE: [1,2,3,4] has type list

- How are lists represented internally?
  - Does not matter for so much for us as users (private representation)

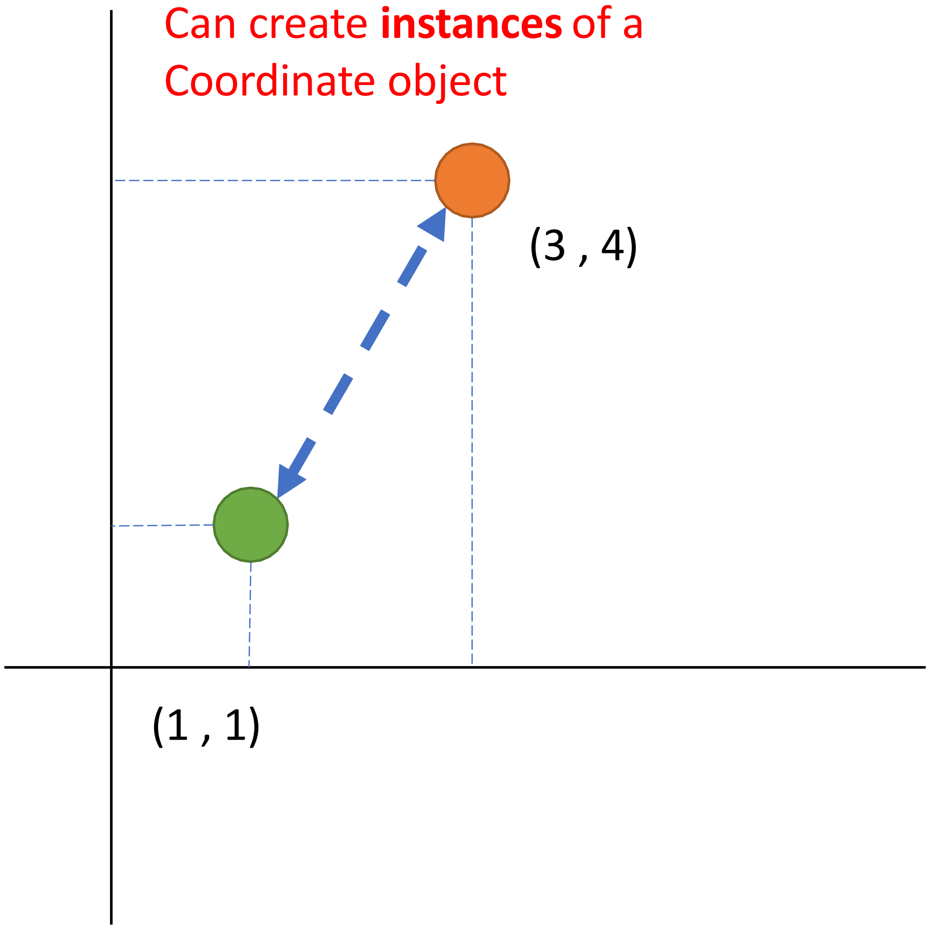


- How to interface with, and manipulate, lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(), L.extend(), L.count(), L.index(), L.insert(), L.pop(), L.remove(), L.reverse(), L.sort()`
- Internal representation should be private
- Correct behavior may be compromised if you manipulate internal representation directly

CREATING AND USING YOUR OWN TYPES WITH CLASSES

- Class = type of object
- Creating the class involves
  - Defining the class name
  - Defining class attributes
    - Data attributes: representation
    - Procedural attributes: interface
  - *for example, a list class*
- Using the class involves
  - Creating new instances of the class
  - Doing operations on the instances
  - *for example,* `L=[1,2]` *and* `len(L)`

COORDINATE TYPE DESIGN DECISIONS



- Decide what data elements constitute an object
  - In a 2D plane
  - A coordinate is defined by an x and y value
- Decide what to do with coordinates
  - Tell us how far away the coordinate is on the x or y axes
  - Measure the distance between two coordinates

DEFINE YOUR OWN TYPES

- Use the class keyword to define a new type

class definition name/type class parent

```
class Coordinate(object):
    #define attributes here
```

- The word object means that Coordinate is a Python object and inherits all its attributes (will see in future lects)
  - Can be omitted
- Similar to def, indent code to indicate which statements are part of the class definition

ATTRIBUTES

- Data attributes
  - Think of data as other objects that represent the object
  - *for example, a coordinate is made up of two numbers*
- Methods (i.e., procedural attributes)
  - Think of methods as functions that only work with this class
  - How to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

Initialize data attributes

- Use a special method called __init__ to initialize some data attributes or perform initialization operations when creating an instance of class

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

what data initializes a Coordinate object

parameter to refer to an instance of the class without having created one yet

special method to create an instance is double underscore

two data attributes make up your type

- self allows you to create variables that belong to this object
- Without self, you are just creating regular variables!

ACTUALLY CREATING AN INSTANCE OF A CLASS

Recall the __init__ method in the class def:

```
def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

- Don't provide argument for `self`, Python does this automatically

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
```

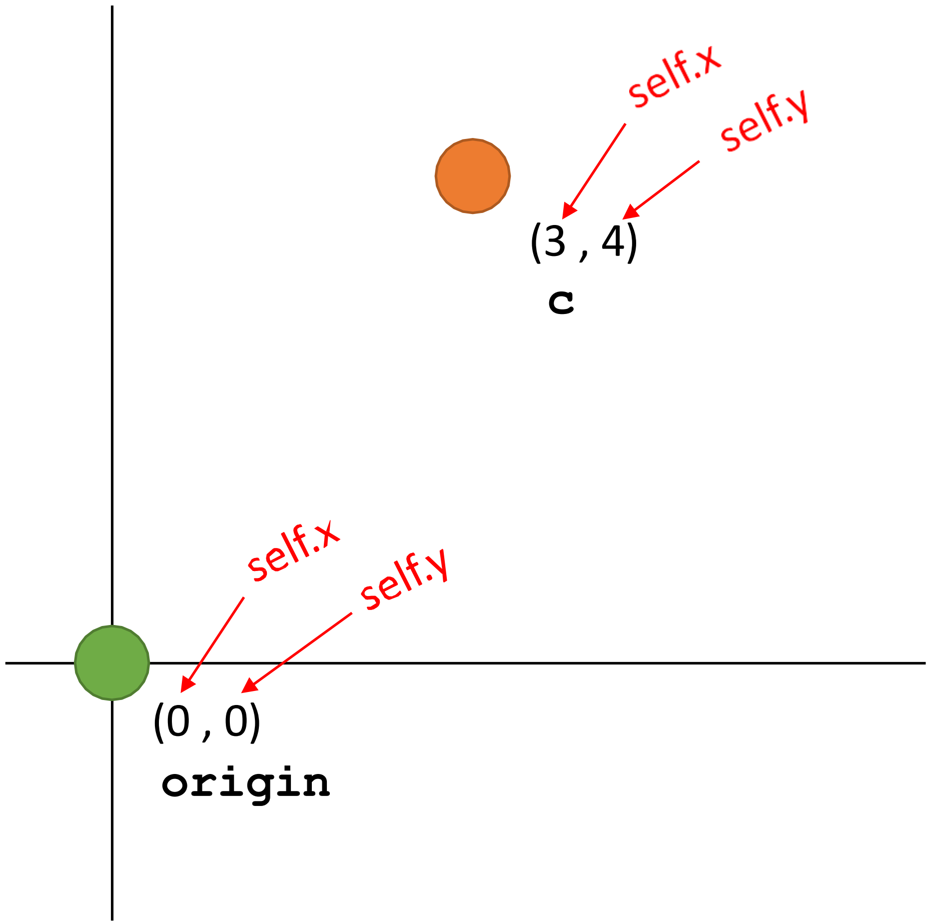
create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

- Data attributes of an instance are called instance variables
  - Data attributes are accessible with dot notation for the lifetime of the object
  - All instances have these data attributes, but with different values!

```
print(c.x)
print(origin.x)
```

use the dot notation to access an attribute of instance `c`

VISUALIZING INSTANCES: draw it



The template for a Coordinate type

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

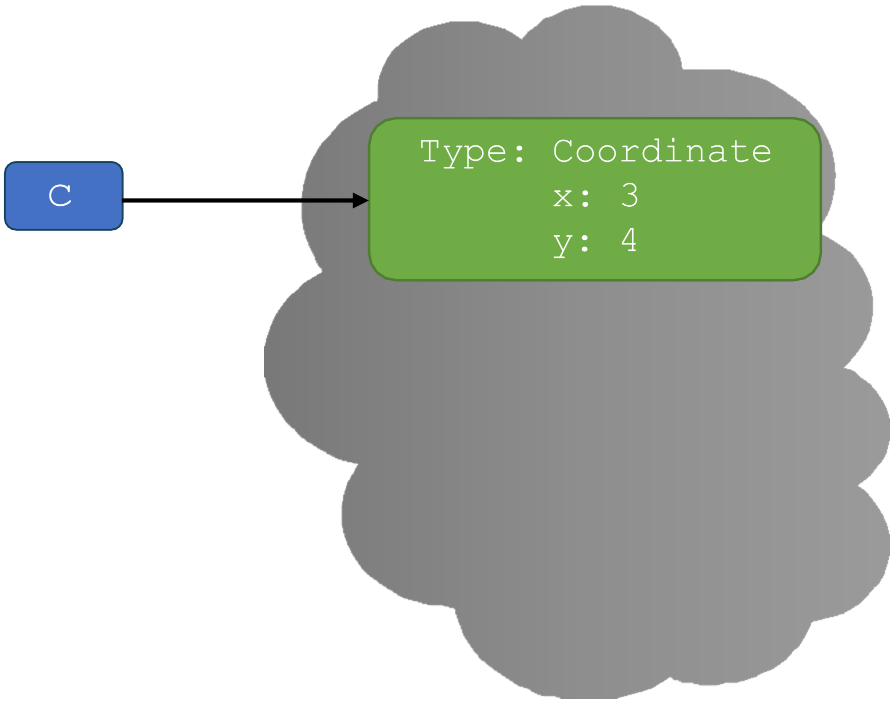
Code to make actual tangible Coordinate objects (aka instances)

VISUALIZING INSTANCES

- Suppose we create an instance of a coordinate

  `c = Coordinate(3,4)`

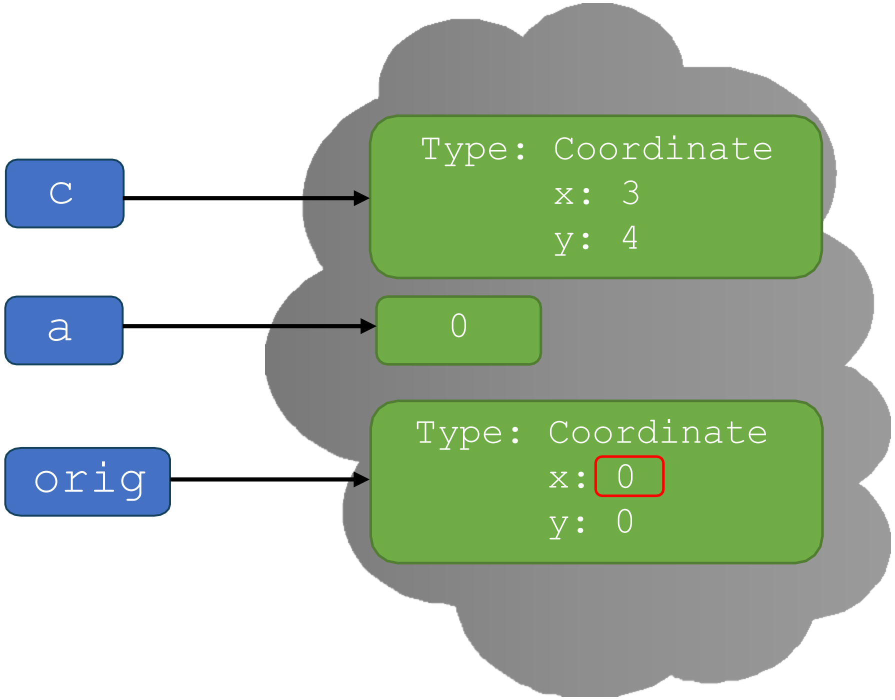
- Think of this as creating a structure in memory
- Then evaluating `c.x` looks up the structure to which `c` points, then finds the binding for `x` in that structure



VISUALIZING INSTANCES: in memory

- Make another instance using a variable

```
a = 0
orig = Coordinate(a,a)
orig.x
```



WHAT IS A METHOD?

- Procedural attribute
  - Think of it like a function that works only with this class

DEFINE A METHOD FOR THE Coordinate CLASS

```
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
```

- Python always passes the object as the first argument
  - Convention is to use self as the name of the first argument of all methods
- Other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO CALL A METHOD?

- The "." operator is used to access any attribute
  - A data attribute of an object (e.g., `c.x`)
  - A method of an object (e.g., `c.distance(orig)`)
- Dot notation

```
<object_variable>.<method>(<parameters>)
```

Object to call method on, becomes self in the class def

Name of method

Not including self. `self` is the obj before the dot!

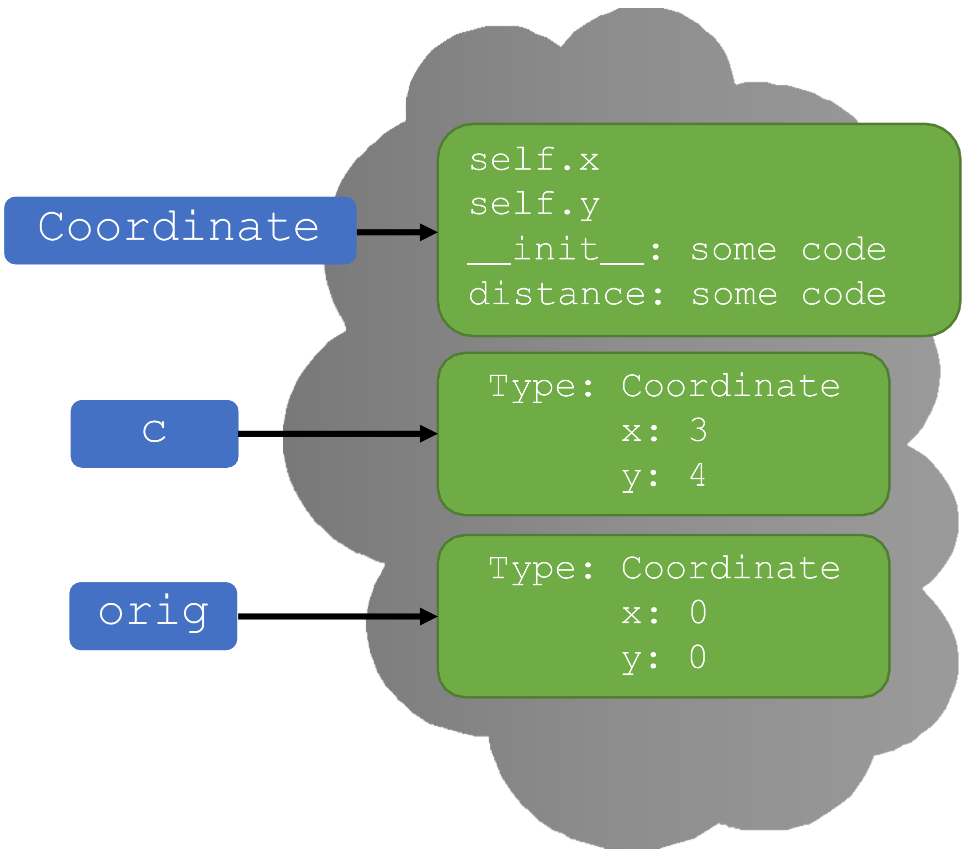
- Familiar?

```
my_list.append(4)
my_list.sort()
```

VISUALIZING INVOCATION

- Coordinate class is an object in memory
  - From the class definition
- Create two Coordinate objects

```
c = Coordinate(3,4)
orig = Coordinate(0,0)
```



VISUALIZING INVOCATION

- Evaluate the method call

`c.distance(orig)`

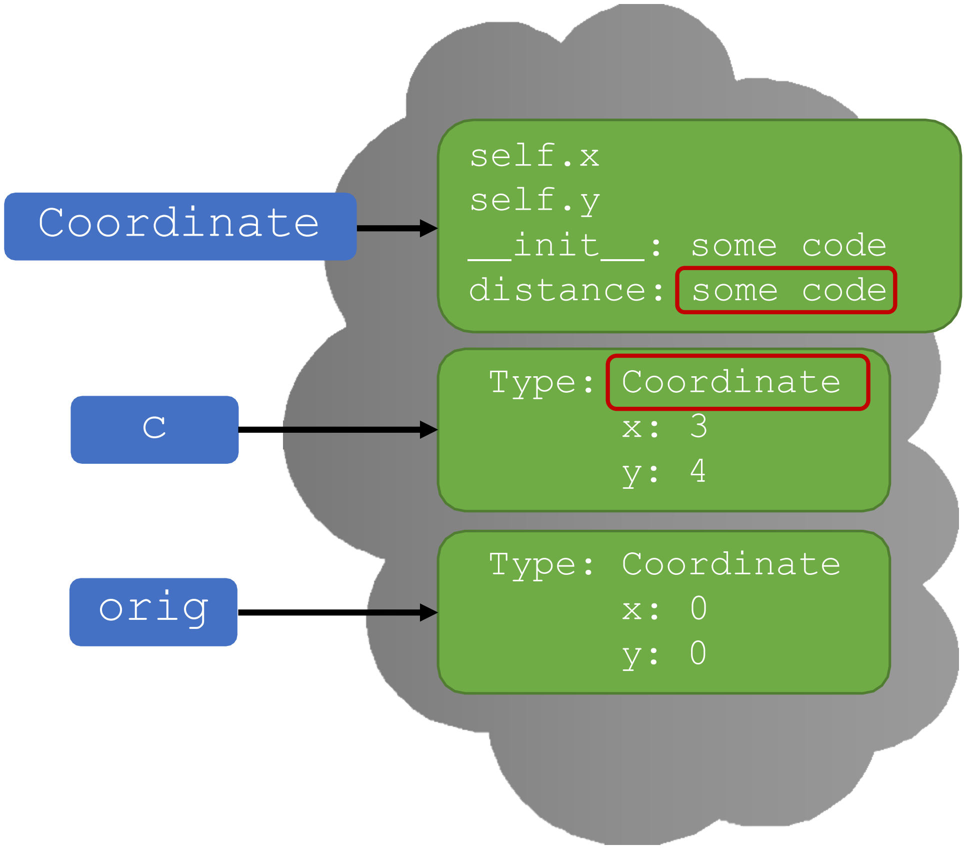
(1) The object is before the dot

(2) Looks up the type of `c`

(3) The method to call is after the dot.

(4) Finds the binding for `distance` in that object class

(5) Invokes that method with `c` as `self` and `orig` as `other`



HOW TO USE A METHOD

- Conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
c.distance(zero)
```

object to call method on, this is self in the class def

name of method

parameters not including `self` (`self` is implied to be `c`)

- Equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
Coordinate.distance(c, zero)
```

name of class (NOT an object of type Coordinate)

name of method

parameters, including an object to call the method on, representing `self`

BIG IDEA

The . operator accesses either data attributes or methods.

Data attributes are defined with self.something

Methods are functions defined inside the class with self as the first parameter.

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

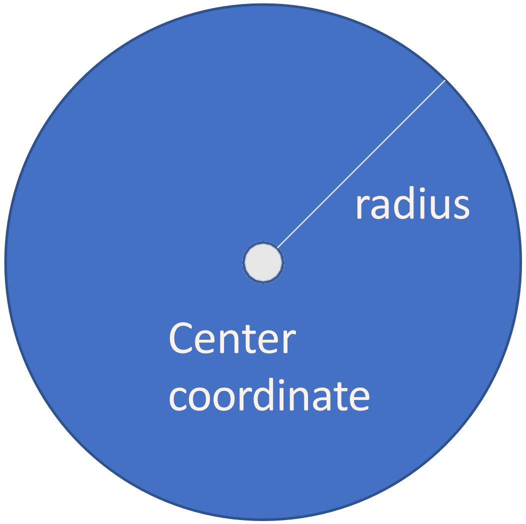
- Instances have **specific values** for attributes

Object Oriented Programming (OOP)

- Bundle related data into packages together with procedures that work on them through well-defined interfaces
- Modular development
  - Implement and test behavior of each class separately
  - Increased modularity reduces complexity

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

Will be a Coordinate object

Will be an int

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

Data attributes, do not need to have the same names as params

```
center = coordinate(2,2)
my_circle = Circle(center, 2)
```

CIRCLE CLASS: DEFINITION and INSTANCES

```
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
```

self is a Circle object

point is a Coordinate object

Coordinate object

Method called on a Coordinate obj

Coordinate object

```
center = Coordinate(2, 2)
my_circle = Circle(center, 2)
p = Coordinate(1,1)
print(my_circle.is_inside(p))
```

Method that only works with obj of type Circle

Coordinate object

Circle object

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

```
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

```
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"""

    def invert(self):
        """Sets self's num to denom and vice versa.
        Returns None."""

# 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
```

Dunder Methods

LET'S TRY IT OUT WITH MORE THINGS

```
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
```

What if we want to keep as a fraction?

```
print(f1)
print(f1 * f2)
```

<__main__.SimpleFraction object at 0x00000234A8C41DF0>

Error!

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
  - "syntactic sugar"
- 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

```
__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, other)        →  len(self)
__str__(self, other)        →  print(self)
__float__(self, other)      →  float(self) i.e. cast
__pow__(self, other)        →  self**other
```

… and others

BIG IDEA

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

Things like: print, len +, *, -, /, , <=, >=, ==, != [] and many others!

PRINT REPRESENTATION OF AN OBJECT

```
>>> 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

```
>>> print(c)
<3,4>
```

DEFINING YOUR OWN PRINT METHOD

```
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)+">"
```

name of special method

must **return a string**

EXAMPLE: FRACTIONS WITH DUNDER METHODS

- The `Fraction` class
  - 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

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

3/4

1/4

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.

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

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
```

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

```
class Fraction(object):
  def __init__(self, n, d):
    self.num = n
    self.denom = d
    .........
  def __mul__(self, oth):
    top=self.num*oth.num
    bottom=self.denom*oth.denom
    return Fraction(top, bottom)
```

Note: we are creating and returning a new **instance of a Fraction**

LETS TRY IT OUT

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

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

```
print(a * b)
```

Shorthand (nice and clear!)

```
print(a.__mul__(b))
```

Call to dunder method, bad style with dunder methods!

```
print(Fraction.__mul__(a, b))
```

Explicit class call, passing in val for self, bad style in general!

- 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

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
```

A float since it does the division directly

```
c = a * b
print(c)          →    3/16
print(float(c))   →    0.1875
```

Repr for Fraction(3,16)

LETS TRY IT OUT SOME MORE

```
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?

ADD A METHOD

```
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)
```

Function to find the greatest common divisor

Call it inside the method

Still want a Fraction object back

```
c = a*b
print(c)
print(c.reduce())
```

→ **2/12**

→ **1/6**

WE HAVE SOME IMPROVEMENTS TO MAKE

```
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 can no longer add or multiply this by other Fractions

CHECK THE TYPES, THEY'RE DIFFERENT

```
a  =  Fraction(4,1)
b  =  Fraction(3,9)
ar  =  a.reduce()
br  =  b.reduce()
print(ar,  type(ar))
print(br,  type(br))
print( ar * br )
```

```
4
1/3
4  <class 'int'>
1/3  <class '__main__.Fraction'>
```

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

```
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)
```

MORE IMPROVEMENTS

- But what if…

```
a = Fraction(4,1)
print( a * 2 )
```

- More improvement

```
class Fraction(object):
    .........
    def __mul__(self, oth):
        if type(oth) == Fraction:
            top=self.num*oth.num
            bottom=self.denom*oth.denom
        elif type(oth) == int:
            top=self.num*oth
            bottom=self.denom
        else:
            raise TypeError
        return Fraction(top, bottom)
```

Type-Checking

- It is generally not recommended to use

```
type(oth) == Fraction
```

- Instead, use `isinstance()` to check if an object is a `Fraction`

```
isinstance(oth, Fraction)
```

- Why?
  - Inheritance…

OOP Summary

- Bundle related data into packages together with procedures that work on them through well-defined interfaces
  - Dunder methods behind the scenes of common operations
- Advantages?
  - Code is organized and modular, thus easy to maintain
  - Bundling data and behaviors means you can use objects consistently
  - It's easy to build upon objects to make more complex objects