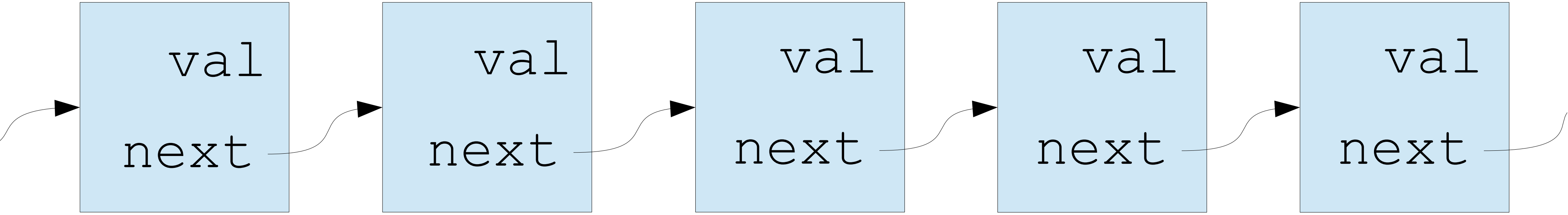
CSc 120

Introduction to Computer Programming II

Classes and Objects

What is Object-Oriented Programming?

- An **object** is a container that can contain multiple named **attributes** (data fields).
- A **class** is a pattern that is used to build many similar objects.



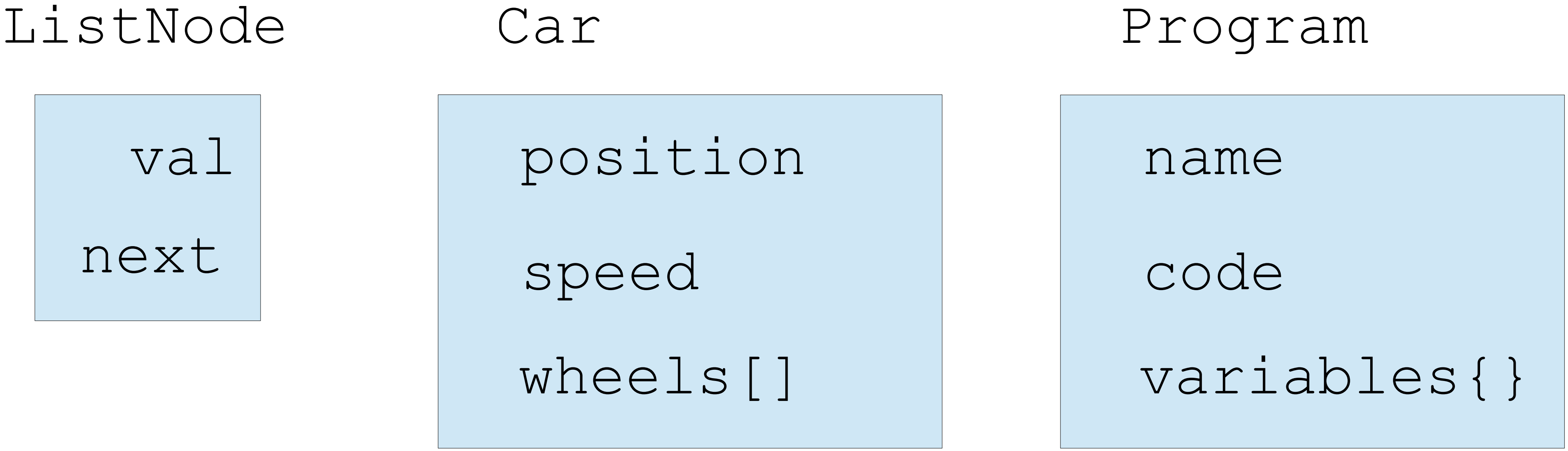
Objects

- Objects can represent many different things
  - Physical object
  - Data record
  - Number, string, or other value
  - Program feature
  - Mathematical construct

  - Anything you can imagine...

Object

- An object is a collection of variables
  - Mix and match any types you like
  - Give them any names you want



ListNode

val

next

Car

position

speed

wheels[]

Program

name

code

variables{}

Why Objects?

- Why use objects?
  - –

Group Discussion:

What are some situations where creating an object – instead of simply having lots of variables – would make your program easier to write?

Why Objects?

- Why use objects?
  - Many copies of the same pattern
  - Organize variables into a logical group

  ...and...

```
ball.move_left(3)
```

Consider:
We are modeling a billiard ball, on a table. We want to move it a few inches to the left.

A good object design will allow us to express this directly.

What if we didn't design our code in an object-oriented style?

```
# move the ball left
ball_x -= 3

# did it hit any other balls?
for <each ball on the table>:
    if <collision>:
        ...
    # TODO: someday, account for
    #       hitting more than one

if <hit edge of table>:
    ...

if <landed in pocket>:
    ...
```

A seemingly simple operation...

```
# move the ball left
ball_x -= 3
```

A seemingly simple operation...

```
# did it hit any other balls?
for <each ball on the table>:
    if <collision>:
        ...
    # TODO: someday, account for
    #       hitting more than one

if <hit edge of table>:
    ...

if <landed in pocket>:
    ...
```

...often has a thousand little details.

Abstraction

Group Exercise:

Write a series of instructions that will tell your instructor how to walk from the front desk, to the other end of the room.

Be as specific as you can – imagine that you are giving instructions to a computer!

Abstraction

Class Demo:

Let's try out some table's instructions. Did you remember to:

- Tell me to stand up, if I'm sitting down?
- Tell me how to walk?
- Tell me how to balance?
- Tell me how to avoid obstacles?
- Tell me when to stop?
- Tell me how to breathe while I'm walking?

Abstraction

Discussion:

Consider a single car: a single object.

Go *down* in the layers (show more detail). What are important details that we have ignored about our car?

Go *up* in the layers (show less detail). How might we represent a car in a larger system? Or how might it be *part* of a larger system?

Abstraction

Discussion:

List all of the things that a student might normally carry in a purse, or in a backpack.

Why do we put all of these things into a container, instead of carrying them individually?

Abstraction

Discussion:

Name some things which are critically important about a person (in the real world), but which we might ignore:

- When they are buying a hamburger
- When they are paying their taxes
- When they are the driver in another car
- When they are hiking in the mountains

Abstraction

- An **abstraction** is something that allows us to think about something as simpler than it really is.
  - Only address “interesting” things
  - Ignore the details

Why Objects?

- Why use objects?
  - Many copies of the same pattern
  - Organize variables into a logical group
  - *Think at a higher level of* ***abstraction!***

How to Use an Object?

- In Python, we access the fields of an object using the “dot” syntax.

```
car.position.x += car.speed.x
neo.is_one = True
print(harry_potter.favorite_subject)
```

How to Use an Object?

- In Python, we access the fields of an object using the "dot" syntax.

```
car.position.x += car.speed.x
neo.is_one = True
print(harry_potter.favorite_subject)
```

This is the object.

How to Use an Object?

- In Python, we access the fields of an object using the “dot” syntax.

```
car.position.x += car.speed.x
neo.is_one = True
print(harry_potter.favorite_subject)
```

This is name of the field inside the object

How to Use an Object?

- In Python, we access the fields of an object using the “dot” syntax.

```
car.position.x += car.speed.x
neo.is_one = True
print(harry_potter.favorite_subject)
```

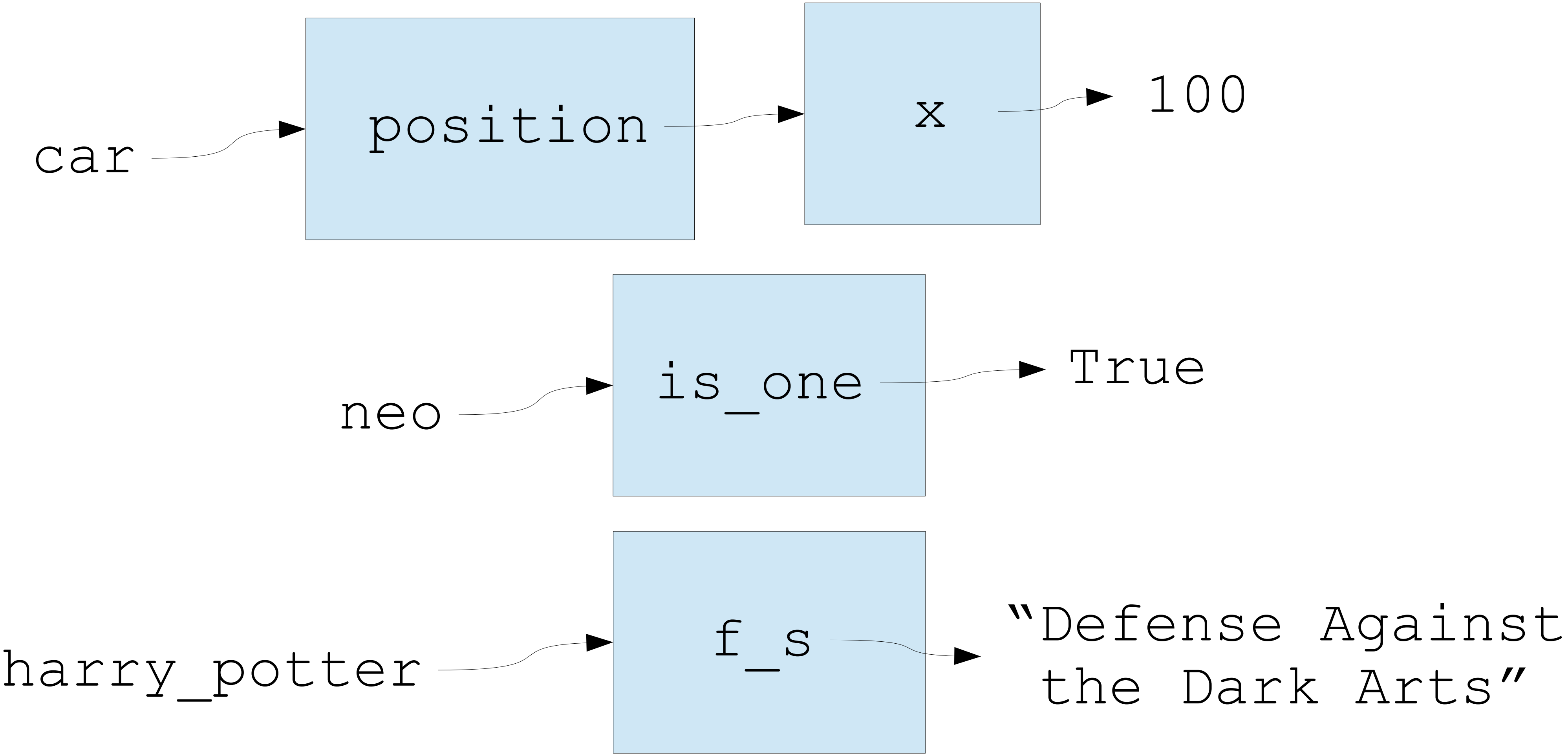
Sometimes, the field is another object, with its own fields.

Group Exercise:

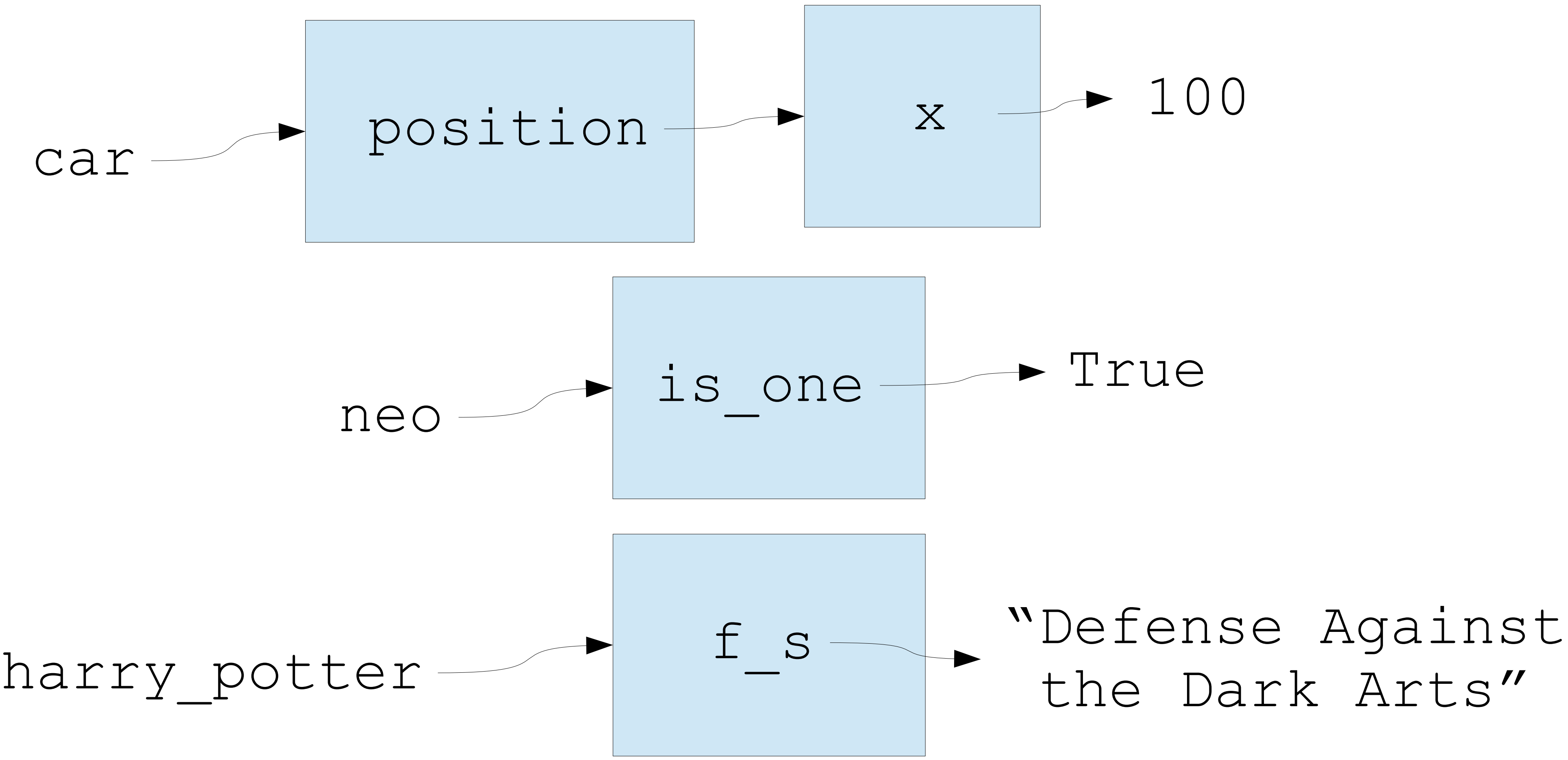
Draw a picture of the three objects shown in this little code snippet. What fields do you know that they have?

```
car.position.x += car.speed.x
neo.is_one = True
print(harry_potter.favorite_subject)
```

```
car.position.x += car.speed.x
neo.is_one = True
print(harry_potter.favorite_subject)
```

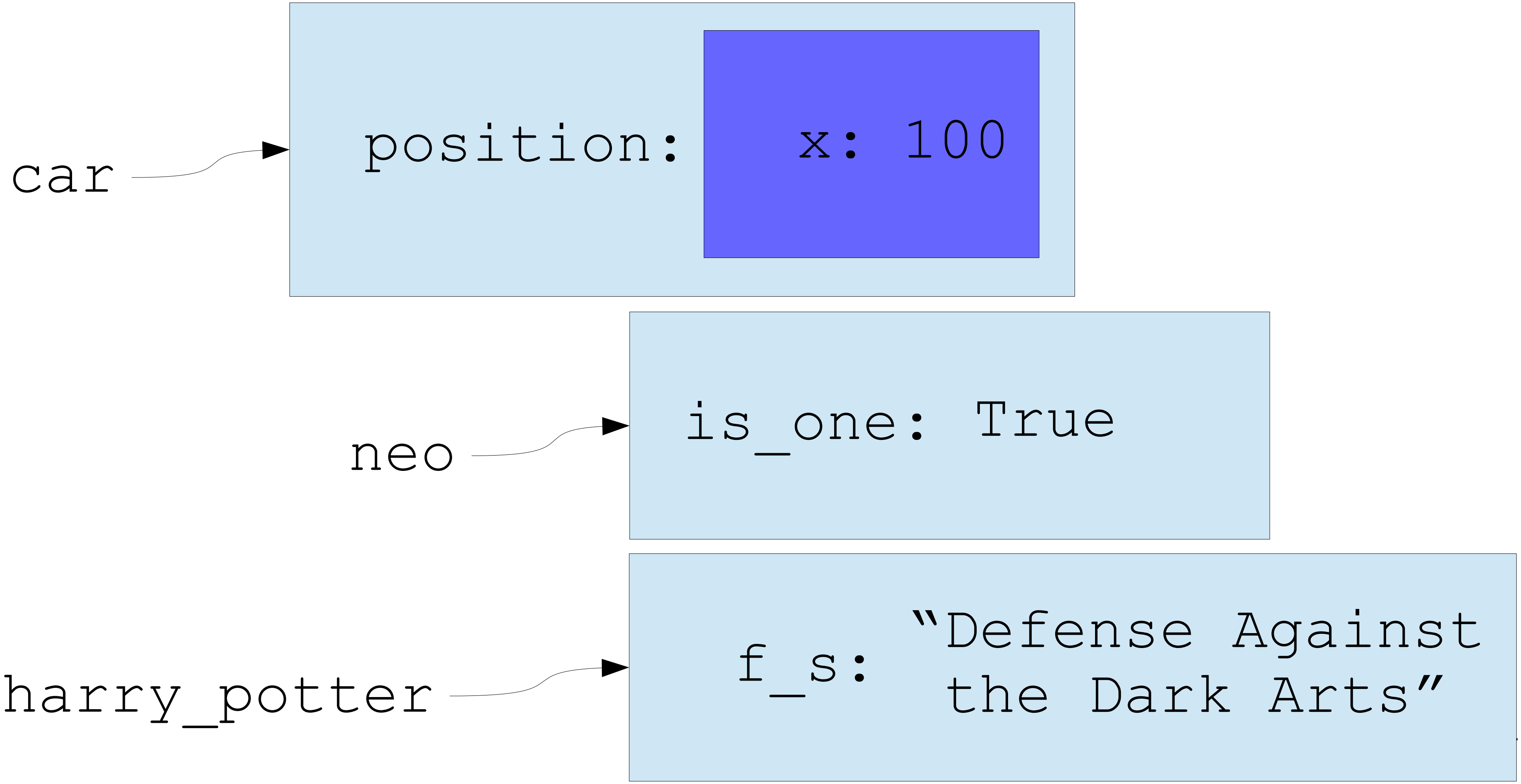


This is a good picture of how the objects are literally arranged in memory.



But sometimes, it looks better to put the values *inside* the object.

Either is OK; which is clearer depends on the program.



Methods

- Sometimes, it would be really useful to be able to functions which run “inside” an object. This is known as a **method.**

```
car.honk_horn()
neo.eat(cookie)
harry_potter.set_invisible(True)
```

Methods

- **Methods** are just like ordinary functions, except:
  - You have to call them with the “dot” syntax
  - They have a automatic **`self`** paramter

```
harry_potter.set_invisible(True)
```

When we call `set_invisible()`, `self` is set to point to `harry_potter`.

Methods

```
class Wizard:
    def set_invisible(self, val):
        ...
```

This is a tiny portion of the declaration for the `Wizard` class.

```
harry_potter.set_invisible(True)
```

Methods

```
class Wizard:
    def set_invisible(self, val):
        ...
```

A **method** is simply a function that is declared inside the class.

```
harry_potter.set_invisible(True)
```

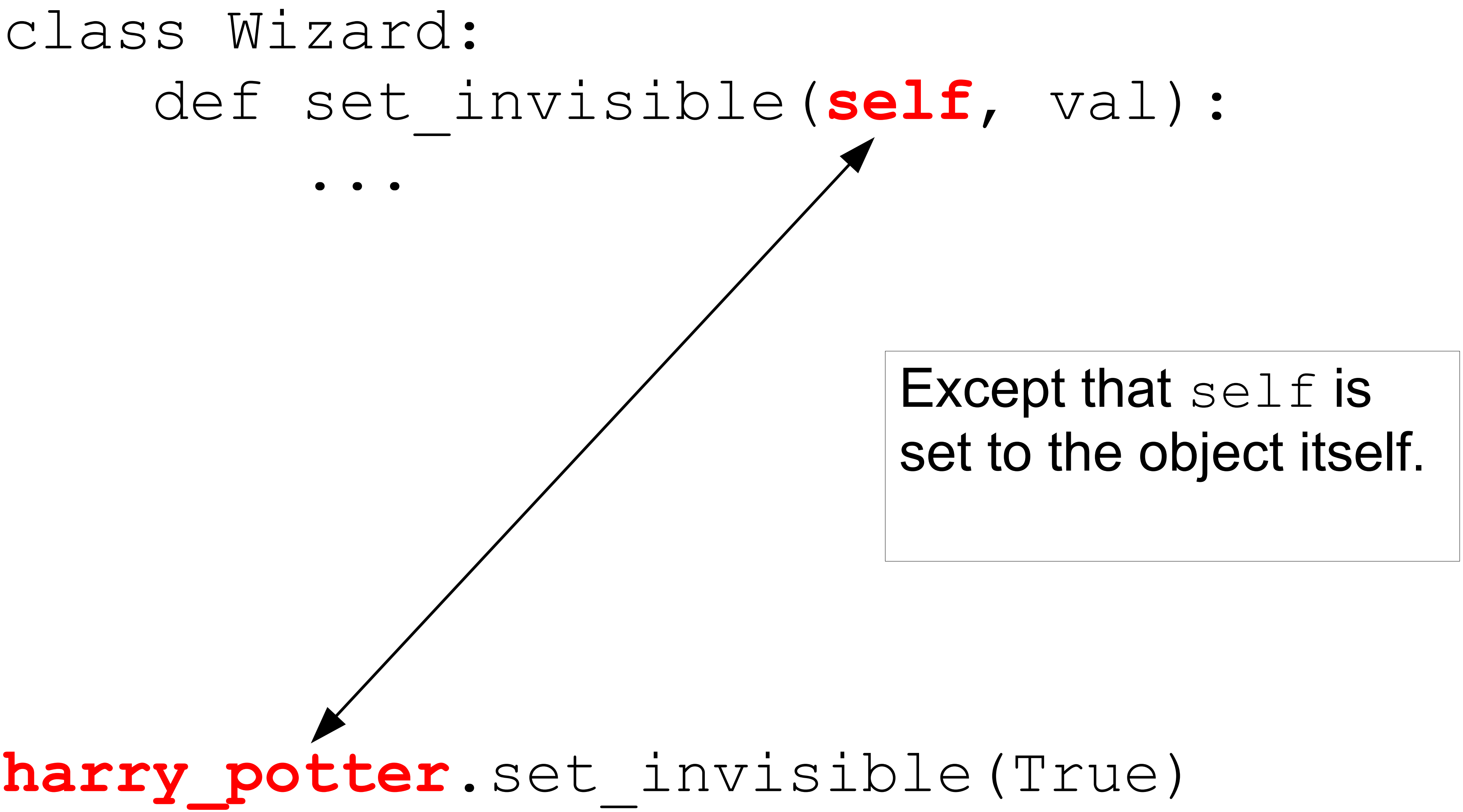
Methods

```
class Wizard:
    def set_invisible(self, val):
        ...
```

When you call a method, parameters are passed, just like ordinary functions...

```
harry_potter.set_invisible(True)
```

Methods



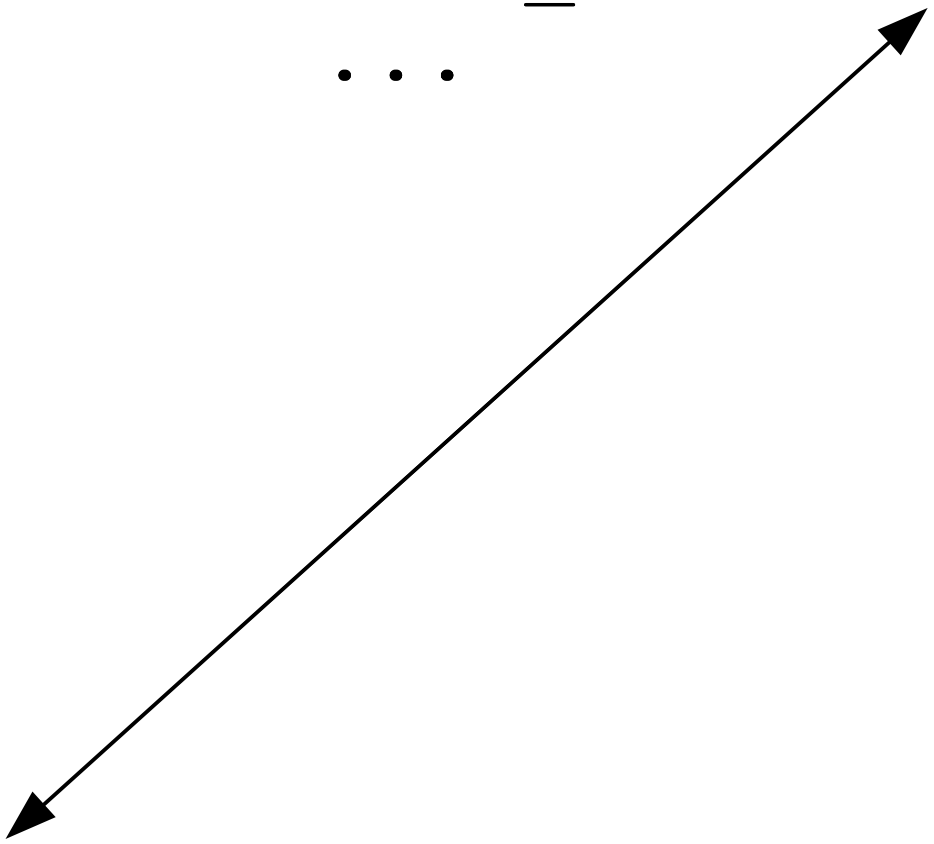
```
class Wizard:
    def set_invisible(self, val):
        ...
```

Except that `self` is set to the object itself.

```
harry_potter.set_invisible(True)
```

Methods

```
class Automobile:
    def honk_horn(self):
        ...
```



```
car.honk_horn()
```

If the method has no explicit parameters, then it *only* has the `self` paramter.

Declaring a Class

- In Python, we **declare a class** with the `class` keyword

```
class Example:
    def foo(self, ... ):
        ...
```

Declaring a Class

- In Python, most classes are made up entirely of methods, no (explicit) data declarations

```
class Example:
    def foo(self, ... ):
        ...
    def bar(self, ... ):
        ...
```

C++/Java Programmers:
Data declarations are not required in Python. To create an attribute, just set it in the constructor.

`__init__()`

C++/Java Programmers:
In Python, always use `self`. Python doesn't support the "implicit `this`" syntax allowed by C++/Java.

- A **constructor** is a method which runs when a new object is created
  - Initializes the data fields
- In Python, the constructor is named **__init__()**

```
class Example:
    def __init__(self):
        self.x = 1
        self.y = 2
        self.vals = [10,11,12]
```

`__init__()`

Group Exercise:

Assume that we've created two new Example objects, named `ralph` and `vanellope`.

Draw the diagram for those objects.

```
class Example:
    def __init__(self):
        self.x = 1
        self.y = 2
        self.vals = [10,11,12]
```

Creating an Object

- Every object is built from a class. An object is known as an **instance** of the class.
- To create an **instance** in Python, call the class name as if it was a function.
  - Each call creates a new object, separate from the last.

```
ralph      = Example()
vanellope = Example()
```

Creating an Object

```
class Example:
    def __init__(self):
        self.x = 1
        self.y = 2
        self.vals =
```

When you ask for a new object, Python allocates memory for it, and then calls `__init__()`. `self` points to the new memory.

```
ralph     = Example()
vanellope = Example()
```

Creating an Object

```
class Example:
    def __init__(self):
        self.x = 1
        self.y = 2
        self.vals =
```

When `__init__()` completes, the new object is returned, and you can save it into a variable.

```
ralph     = Example()
vanellope = Example()
```

__init__() Parameters

- The **constructor** may take parameters.

```
class IceCreamCone:
    def __init__(self, num_scoops):
        self.scoops = ...
```

```
mine  = IceCreamCone(3)
yours = IceCreamCone(1)
```

Group Exercise:

Write a class called `Counter`, which represents a click-counter like the one pictured.

Include a **constructor**.

Include a **method** which, when called, will increment an internal counter by 1.

Include a **method** which returns the current count.

Include a **method** which will reset it to zero.

Create three **instances** of this class.

```
class Counter:
    def __init__(self):
        self.count = 0

    def click(self):
        self.count += 1

    def get_count(self):
        return self.count

    def reset(self):
        self.count = 0

a = Counter()
b = Counter()
c = Counter()
```

Encapsulation

Discussion:

Why did `Counter` have lots of methods to do simple things – instead of just accessing the fields directly?

That is, why is better for code, outside the class, to call

```
  counter1.click()
```

instead of just doing it by hand?

```
  counter1.count += 1
```

```
counter1 = Counter()
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
```

Seems easy enough...what's the problem?

```
counter1 = Counter()
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
counter1.count += 1
...
```

Group Exercise:

Add three features to the Counter:

- When it hits 10000, wrap around to 0
- Keep track of how many times it's been reset
- Add support for adding more than 1 on each click.

(no, don't actually do this)

Encapsulation

A Good Principle:

- Minimize how much of the internal state is visible to code outside the class **(encapsulation)**

- Allows you to change the implementation
- Allows you to think at a higher level of **abstraction**

Encapsulation

Convention:

- Use a leading underscore to indicate that an attribute or method is “private”

```
_count
```

Encapsulation

Convention:

- Use a leading underscore to indicate that an attribute or method is “private”

Code outside the class should never touch this attribute.

```
class Counter:
    def __init__(self):
        self._count = 0

    def click(self):
        self._count += 1
```

But calling this method is fine.

Getters and Setters

- A **getter** is a method that reads a property
  - *Often but not always* just returns a private attribute
- A **setter** is a method that changes a property
  - *Often but not always* just sets a private attribute

```
class Person:
    def __init__(self):
        self._name = “<unknown>”

    def set_name(self, name):
        self._name = name
    def get_name(self):
        return self._name
```

Group Exercise:

Write a class called `Turtle`, which represents an object that moves around on a 2D field.

Include a **constructor.** Initialize its position to (0,0), its direction to North (+y) and its speed to 1.

Include **methods** which turn the turtle left or right by 90 degrees per call, as well as one which changes the speed.

Include a **method** called `one_step(),` which moves the turtle, in the current direction, ahead by the current speed.

Include a **method** called `get_pos(),` which returns the current (x,y) position as a **tuple.**

All **attributes** must be private.

Special Methods

- Python supports some special methods, which use double underscores at front and back.
  - You may implement these or not
  - If you implement them, then Python will call it in certain situations

- We've already seen `__init__` . What else is there?

__str__

__str__ is called when Python wants to convert an object to its string representation.

```
class Thing:
    def __init__(self):
        self._x = 1
        self._y = 2
    def __str__(self):
        return "(%d,%d)" % (self._x,self._y)

tmp = Thing()
print(tmp)
string_version = str(tmp)
```

__str__

__str__ is called when Python wants to convert an object to its string representation.

```
class Thing:
    def __init__(self):
        self._x = 1
        self._y = 2
    def __str__(self):
        return “(%d,%d)” % (s

tmp = Thing()
print(tmp)
string_version = str(tmp)
```

When you try to print an object, Python calls __str__ to know what to print.

__str__

__str__ is called when Python wants to convert an object to its string representation.

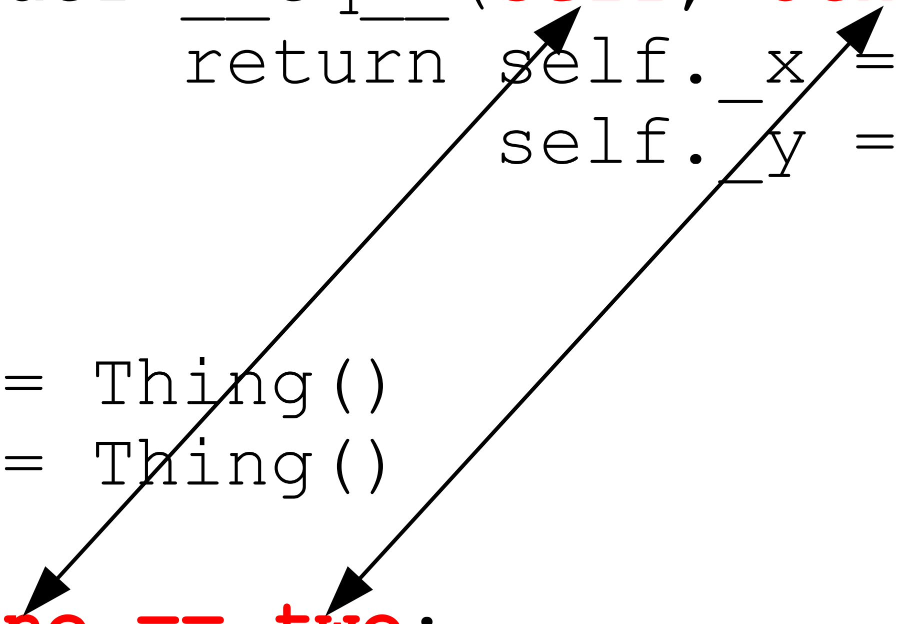
```
class Thing:
    def __init__(self):
        self._x = 1
        self._y = 2
    def __str__(self):
        return “(%d,%d)” % (s

tmp = Thing()
print(tmp)
string_version = str(tmp)
```

When you call str(), Python calls __str__() for you.

__eq__

__eq__ is called when Python wants to check to see if this object is equal to another.

```
class Thing:
    def __init__(self):
        self._x = 1
        self._y = 2
    def __eq__(self, other):
        return self._x == other._x and
               self._y == other._y
```



```
class Thing:
    def __init__(self):
        self._x = 1
        self._y = 2
    def __eq__(self, other):
        return self._x == other._x and
               self._y == other._y


one = Thing()
two = Thing()

if one == two:
    print(“same!”)
```

If you compare two objects, Python calls `__eq__()` on the left-hand object.

More

- Python supports many other special methods, which you can investigate on your own time:

  `__ne__` , `__lt__` , etc.

  `__len__` , `__contains__` , etc.

  `__add__` , `__sub__` , etc.