What Are Objects?

```
An object is a custom data structure containing both data (variables,
called attributes) and code (functions, called methods).

It represents a unique instance of some concrete thing.

An object represents an individual thing, and its methods define how it
interacts with other things.

For example, the integer object with the value 7 is an object that
facilitates methods such as addition and multiplication.

Again 8 is a different object.

This means there’s an integer class built in somewhere in Python, to
which both 7 and 8 belong.

The strings 'cat' and 'duck' are also objects in Python, and have
string methods such as capitalize() and replace().

Let’s start with basic object classes.
```

Define a Class with class

```
To create a new object that no one has ever created before, you first define a class that indicate

To create your own custom object in Python, you first need to define a class by using the class ke

Suppose that you want to define objects to represent information about cats.

Each object will represent one feline.

You’ll first want to define a class called Cat.
```

```
# we needed to say pass to indicate that this class was empty.
# This definition is the bare minimum to create an object.
```

```
class Cat():
  pass
```

```
# You create an object from a class by calling the class name as though it were a function
a_cat = Cat()
another_cat = Cat()
# In this case, calling Cat() creates two individual objects from the Cat class, and we
# assigned them to the names a_cat and another_cat. But our Cat class had no other
# code, so the objects that we created from it just sit there and can’t do much else.
```

Attributes

An attribute is a variable inside a class or object.

During and after an object or class is created, you can assign attributes to it.

An attribute can be any other object.

Let’s make two cat objects again:

```
class Cat():
  pass
a_cat = Cat()
a_cat
```

```
<__main__.Cat at 0x7fe9c8a6b9d0>
```

```
another_cat = Cat()
another_cat
```

```
<__main__.Cat at 0x7fe9c8a6bf40>
```

When we defined the Cat class, we didn’t specify how to print an object from that class.

Python jumps in and prints something like

<__main__.Cat at 0x7efd9a04e9d0> and <__main__.Cat at 0x7efd9a04e4d0>.

Now assign a few attributes to our first object:

```
a_cat.age = 3
a_cat.name = "Mr. Fuzzybuttons"
a_cat.nemesis = another_cat
```

```
a_cat.age
```

```
3
```

```
a_cat.name
```

```
'Mr. Fuzzybuttons'
```

```
a_cat.nemesis
```

```
<__main__.Cat at 0x7fe9c8a6bf40>
```

Because nemesis was an attribute referring to another Cat object, we can use a_cat.nemesis to access it, but this other object doesn't have a name attribute yet:

```
a_cat.nemesis.name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-86a423fa8243> in <module>
----> 1 a_cat.nemesis.name

AttributeError: 'Cat' object has no attribute 'name'
```

SEARCH STACK OVERFLOW

```
a_cat.nemesis.name = "Mr. Bigglesworth"
a_cat.nemesis.name
```

Even the simplest object like this one can be used to store multiple attributes.

So, you can use multiple objects to store different values, instead of using something like a list or dictionary.

Methods

A method is a function in a class or object. A method looks like any other function.

Initialization

If you want to assign object attributes at creation time, you need the special Python object initialization method __init__():

```
class Cat:
  def __init__(self):
    pass
```

This is what you'll see in real Python class definitions.

__init__() is the special Python name for a method that initializes an individual object from its class definition.

The self argument specifies that it refers to the individual object itself.

When you define __init__() in a class definition, its first parameter should be named self.

Although self is not a reserved word in Python, it's common usage.

```
class Cat():
  def __init__(self, name):
    self.name = name
```

Now we can create an object from the Cat class by passing a string for the name parameter:

```
furball = Cat('Grumpy')
```

Here's what this line of code does:

- Looks up the definition of the Cat class
- Instantiates (creates) a new object in memory
- Calls the object's __init__() method, passing this newly created object as self and the other argument ('Grumpy') as name
- Stores the value of name in the object

- Returns the new object
- Attaches the variable furball to the object

This new object is like any other object in Python.

You can use it as an element of a list, tuple, dictionary, or set.

You can pass it to a function as an argument, or return it as a result.

What about the name value that we passed in?

It was saved with the object as an attribute.

You can read and write it directly:

```
print('Our latest addition: ', furball.name)
```

Remember, inside the Cat class definition, you access the name
attribute as self.name.

When you create an actual object and assign it to a variable like
furball, you refer to it as furball.name.

It is not necessary to have an __init__() method in every class
definition; it's used to do anything that's needed to distinguish this
object from others created from the same class.

It's not what some other languages would call a "constructor."

Python already constructed the object for you.

Think of __init__() as an initializer.

Inheritance

inheritance is creating a new class from an existing class, but with
some additions or changes.

It's a good way to reuse code.

When you use inheritance, the new class can automatically use all the code from the old class but without you needing to copy any of it.

Inherit from a Parent Class

You define only what you need to add or change in the new class, and this overrides the behavior of the old class.

The original class is called a parent, superclass, or base class; the new class is called a child, subclass, or derived class.

These terms are interchangeable in object-oriented programming.

So, let’s inherit something.

In the next example, we define an empty class called Car.

Next, we define a subclass of Car called Yugo.

You define a subclass by using the same class keyword but with the parent class name inside the parentheses (class Yugo(Car) here):

```
class Car():
  pass

class Yugo(Car):
  pass
```

You can check whether a class is derived from another class by using issubclass():

```
issubclass(Yugo, Car)
```

Next, create an object from each class:

```

```

```
give_me_a_car = Car()
give_me_a_yugo = Yugo()
```

A child class is a specialization of a parent class; in object-oriented lingo, Yugo is-a Car.

The object named give_me_a_yugo is an instance of class Yugo, but it also inherits whatever a Car can do.

let’s try new class definitions that actually do something:

```
class Car():
  def exclaim(self):
    print("I'm a Car!")

class Yugo(Car):
  pass
```

Finally, make one object from each class and call the exclaim method:

```
give_me_a_car = Car()
give_me_a_yugo = Yugo()
```

```
give_me_a_car.exclaim()

give_me_a_yugo.exclaim()
```

Without doing anything special, Yugo inherited the exclaim() method from Car.
In fact, Yugo says that it is a Car, which might lead to an identity crisis.
Let’s see what we can do about that.

Override a Method

As you just saw, a new class initially inherits everything from its parent class.

Moving forward, you'll see how to replace or override a parent method.

Yugo should probably be different from Car in some way; otherwise,
what's the point of defining a new class?

Let's change how the exclaim() method works for a Yugo:

```
class Car():
  def exclaim(self):
    print("I'm a Car!")

class Yugo(Car):
  def exclaim(self):
    print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
```

Now make two objects from these classes:

```
give_me_a_car = Car()
give_me_a_yugo = Yugo()
```

```
give_me_a_car.exclaim()
```

```
give_me_a_yugo.exclaim()
```

In these examples, we overrode the exclaim() method.

We can override any methods, including __init__().

Here's another example that uses a Person class.

Let's make subclasses that represent doctors (MDPerson) and lawyers (JDPerson):

```
class Person():
  def __init__(self, name):
    self.name = name

class MDPerson(Person):
  def __init__(self, name):
    self.name = "Doctor " + name

class JDPerson(Person):
  def __init__(self, name):
    self.name = name + ", Esquire"
```

```
#  here Esquire means a polite title appended to a man's name when no other title is used,
#  typically in the address of a letter or other documents. for example J. C. Pearson Esqu
```

In these cases, the initialization method __init__() takes the same arguments as the parent Person class but stores the value of name differently inside the object instance:

```
person = Person('Fudd')
doctor = MDPerson('Fudd')
lawyer = JDPerson('Fudd')
```

```
print(person.name)
print(doctor.name)
print(lawyer.name)
```

Add a Method

The child class can also add a method that was not present in its parent class.

Going back to classes Car and Yugo, we'll define the new method need_a_push() for class Yugo only:

```
class Car():
  def exclaim(self):
    print("I'm a Car!")

class Yugo(Car):
  def exclaim(self):
    print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
  def need_a_push(self):
    print("A little help here?")
```

```
give_me_a_car = Car()
give_me_a_yugo = Yugo()
```

A Yugo object can react to a need_a_push() method call:

```
give_me_a_yugo.need_a_push()
```

```
#But a generic Car object cannot:
give_me_a_car.need_a_push()
#At this point, a Yugo can do something that a Car cannot, and the distinct personality of
```

Get Help from Your Parent with super()

We saw how the child class could add or override a method from the parent.

What if it wanted to call that parent method?

Here, we define a new class called EmailPerson that represents a Person
with an email address.
First, our familiar Person definition:

```
class Person():
  def __init__(self, name):
    self.name = name
```

Notice that the __init__() call in the following subclass has an
additional email parameter:

```
class EmailPerson(Person):
  def __init__(self, name, email):
    super().__init__(name)
    self.email = email
```

When you define an __init__() method for your class, you're replacing
the __init__() method of its parent class, and the latter is not called
automatically anymore.

As a result, we need to call it explicitly.

Here's what's happening:
- The super() gets the definition of the parent class, Person.
- The __init__() method calls the Person.__init__() method. It takes
care of passing the self argument to the superclass, so you just need
to give it any optional arguments. In our case, the only other argument
Person() accepts is name.

- The self.email = email line is the new code that makes this EmailPerson different from a Person.

Moving on, let's make one of these creatures:

```
bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

We should be able to access both the name and email attributes:

```
bob.name
```

```
bob.email
```

Why didn't we just define our new class as follows?

```
class EmailPerson(Person):
  def __init__(self, name, email):
    self.name = name
    self.email = email
```

We could have done that, but it would have defeated our use of inheritance.
We used super() to make Person do its work, the same as a plain Person object would.

There's another benefit: if the definition of Person changes in the future, using super() will ensure that the attributes and methods that EmailPerson inherits from Person will reflect the change.

Use super() when the child is doing something its own way but still needs something from the parent (as in real life).

Multiple Inheritance

Actually, objects can inherit from multiple parent classes.

If your class refers to a method or attribute that it doesn't have, Python will look in all the parents.

What if more than one of them has something with that name?

Who wins?

inheritance in Python depends on method resolution order.

Each Python class has a special method called mro() that returns a list
of the classes that would be visited to find a method or attribute for
an object of that class.

A similar attribute, called __mro__, is a tuple of those classes.

Here, we define a top Animal class, two child classes (Horse and Donkey)
, and then two derived from these:

```
class Animal:
  def says(self):
    return 'I speak!'

class Horse(Animal):
  def says(self):
    return 'Neigh!'

class Donkey(Animal):
  def says(self):
    return 'Hee-haw!'

class Mule(Donkey, Horse):
  pass

class Hinny(Horse, Donkey):
  pass
# A mule has a father donkey and mother horse;
# A hinny has a father horse and mother donkey.
```

If we look for a method or attribute of a Mule, Python will look at the
following things, in this order:

1. The object itself (of type Mule)
2. The object’s class (Mule)
3. The class’s first parent class (Donkey)
4. The class’s second parent class (Horse)
5. The grandparent class (Animal) class

```
Mule.mro()
```

```
Hinny.mro()
```

```
mule = Mule()
hinny = Hinny()
```

```
mule.says()
```

```
hinny.says()
```

If the Horse and Donkey did not have a says() method, the mule or hinny would have used the grandparent Animal class's says() method, and returned 'I speak!'.

In self Defense

Python uses the self argument to find the right object's attributes and methods.

For an example, I'll show how you would call an object's method, and what Python actually does behind the scenes.

Remember class Car from earlier examples?

Let's call its exclaim() method again:

```
a_car = Car()
a_car.exclaim()
```

Here's what Python actually does, under the hood:

- Look up the class (Car) of the object a_car
- Pass the object a_car to the exclaim() method of the Car class as the self parameter

Just for fun, you can even run it this way yourself and it will work the same as the normal (a_car.exclaim()) syntax:

```
```

```
Car.exclaim(a_car)
```

Attribute Access

```
In Python, object attributes and methods are normally public.
```

Direct Access

```
As you've seen, you can get and set attribute values directly:
```

```
class Duck:
  def __init__(self, input_name):
    self.name = input_name

fowl = Duck('Daffy')
fowl.name
```

```
But what if someone misbehaves?
```

```
fowl.name = 'Daphne'
fowl.name
```

```
The next two sections show ways to get some privacy for attributes that
you don't want anyone to stomp by accident.
```

Getters and Setters

```
Some object-oriented languages support private object attributes that
can’t be accessed directly from the outside.

Programmers then may need to write getter and setter methods to read
and write the values of such private attributes.

Python doesn’t have private attributes, but you can write getters and setters with obfuscated attr

In the following example, we define a Duck class with a single instance
attribute called hidden_name.

We don’t want people to access this directly, so we define two methods:
a getter (get_name()) and a setter (set_name()).

Each is accessed by a property called name.

I’ve added a print() statement to each method to show when it’s being called:
```

```
class Duck():
  def __init__(self, input_name):
    self.hidden_name = input_name
  def get_name(self):
    print('inside the getter')
    return self.hidden_name
  def set_name(self, input_name):
    print('inside the setter')
    self.hidden_name = input_name

don = Duck('Donald')
don.get_name()
```

```
don.set_name('Donna')
```

```
don.get_name()
```

```
class Apu:
  def __init__(self,myname):
    self.mname=myname
```

```
a=Apu('APURBA')
a.mname
```

Properties for Attribute Access

```
The Pythonic solution for attribute privacy is to use properties.

There are two ways to do this.

The first way is to add name = property(get_name,set_name) as the final line of our previous Duck
```

```
class Duck():
  def __init__(self, input_name):
    self.hidden_name = input_name
  def get_name(self):
    print('inside the getter')
    return self.hidden_name
  def set_name(self, input_name):
    print('inside the setter')
    self.hidden_name = input_name
  name = property(get_name, set_name)
```

The old getter and setter still work:

```
don = Duck('Donald')
don.get_name()
```

```
don.set_name('Donna')
```

```
don.get_name()
```

```
But now you can also use the property name to get and set the hidden name:
```

```
don = Duck('Donald')
```

```
don.name
```

```
don.name = 'Donna'
```

```
don.name
```

In the second method, you add some decorators and replace the method names get_name and set_name with name:

- @property, which goes before the getter method
- @name.setter, which goes before the setter method

Here's how they actually look in the code:

```
class Duck():
  def __init__(self, input_name):
    self.hidden_name = input_name
  @property
  def name(self):
    print('inside the getter')
    return self.hidden_name
  @name.setter
  def name(self, input_name):
    print('inside the setter')
    self.hidden_name = input_name
```

You can still access name as though it were an attribute:

```
fowl = Duck('Howard')
```

```
fowl.name
```

```
fowl.name = 'Donald'
```

```
fowl.name
```

Properties for Computed Values

In the previous examples, we used the name property to refer to a single attribute (hidden_name) stored within the object.

A property can also return a computed value.

Let's define a Circle class that has a radius attribute and a computed diameter property:

```
class Circle():
  def __init__(self, radius):
```

```
    self.radius = radius
  @property
  def diameter(self):
    return 2 * self.radius
```

Create a Circle object with an initial value for its radius:

```
c = Circle(5)
c.radius
```

We can refer to diameter as if it were an attribute such as radius:

```
c.diameter
```

```
Here's the fun part: we can change the radius attribute at any time,
and the diameter property will be computed from the current value of radius:
```

```
c.radius = 7
c.diameter
```

```
If you don't specify a setter property for an attribute, you can't set
it from the outside.


This is handy for read-only attributes:
```

```
c.diameter = 20
```

```
There's one more advantage of using a property over direct attribute
access: if you ever change the definition of the attribute, you need to
fix only the code within the class definition, not in all the callers.
```

Name Mangling for Privacy

In the Duck class example a little earlier, we called our (not completely) hidden attribute hidden_name.

Python has a naming convention for attributes that should not be visible outside of their class definition: begin with two underscores (__).

Let's rename hidden_name to __name, as demonstrated here:

```
class Duck():
  def __init__(self, input_name):
    self.__name = input_name
  @property
  def name(self):
    print('inside the getter')
    return self.__name
  @name.setter
  def name(self, input_name):
    print('inside the setter')
    self.__name = input_name
```

Take a moment to see whether everything still works:

```
fowl = Duck('Howard')
```

```
fowl.name
```

```
fowl.name = 'Donald'
```

```
fowl.name
```

Looks good. And you can't access the __name attribute:

```
fowl.__name
```

This naming convention doesn't make it completely private, but Python does mangle the attribute name to make it unlikely for external code to stumble upon it.

```
If you’re curious and promise not to tell everyone,here’s what it becomes:
```

```
fowl._Duck__name
```

```
Notice that it didn’t print inside the getter. Although this isn’t perfect protection, name mangli
```

Class and Object Attributes

```
You can assign attributes to classes, and they’ll be inherited by their child objects:
```

```
class Fruit:
  color = 'red'
```

```
blueberry = Fruit()
```

```
Fruit.color
```

```
blueberry.color
```

```
But if you change the value of the attribute in the child object, it doesn’t affect the class attr
```

```
blueberry.color = 'blue'
blueberry.color
```

```
Fruit.color
```

```
If you change the class attribute later, it won’t affect existing child objects:
```

```
Fruit.color = 'orange'
Fruit.color
```

```
blueberry.color
```

But it will affect new ones:

```
new_fruit = Fruit()
new_fruit.color
```

Method Types

Some methods are part of the class itself, some are part of the objects that are created from that class, and some are none of the above:

- If there's no preceding decorator, it's an instance method, and its first argument should be self to refer to the individual object itself.
- If there's a preceding @classmethod decorator, it's a class method, and its first argument should be cls (or anything, just not the reserved word class), referring to the class itself.
- If there's a preceding @staticmethod decorator, it's a static method, and its first argument isn't an object or class.

Instance Methods

When you see an initial self argument in methods within a class definition, it's an instance metho

These are the types of methods that you would normally write when creating your own classes.

The first parameter of an instance method is self, and Python passes the object to the method wher

These are the ones that you've seen so far.

Class Methods

In contrast, a class method affects the class as a whole.

Any change you make to the class affects all of its objects.

Within a class definition, a preceding @classmethod decorator indicates
that that following function is a class method.

Also, the first parameter to the method is the class itself.

The Python tradition is to call the parameter cls,because class is a
reserved word and can’t be used here.

Let’s define a class method for A that counts how many object instances
have been made from it:

```
class A():
  count = 0
  def __init__(self):
    A.count += 1
  def exclaim(self):
    print("I'm an A!")
  @classmethod
  def kids(cls):
    print("A has", cls.count, "little objects.")
```

```
easy_a = A()
```

```
breezy_a = A()
```

```
wheezy_a = A()
```

```
A.kids()
```

Notice that we referred to A.count (the class attribute) in __init__() rather than self.count (whi

In the kids() method, we used cls.count, but we could just as well have used A.count.

Static Methods

```
A third type of method in a class definition affects neither the class nor its objects; it's just

It's a static method,preceded by a @staticmethod decorator, with no initial self or cls parameter.

Here's an example that serves as a commercial for the class CoyoteWeapon:
```

```
class CoyoteWeapon():
  @staticmethod
  def commercial():
    print('This CoyoteWeapon has been brought to you by Acme')
```

```
CoyoteWeapon.commercial()
```

```
Notice that we didn't need to create an object from class CoyoteWeapon to access this method. Very
```

Magic Methods

```
When you type something such as a = 3 + 8, how do the integer objects
with values 3 and 8 know how to implement +?

Or, if you type name = "Daffy" + " " + "Duck", how does Python know
that + now means to concatenate these strings?

And how do a and name know how to use = to get the result?

You can get at these operators by using Python's special methods (or, more dramatically, magic met

The names of these methods begin and end with double underscores (__).

Why? They're very unlikely to have been chosen by programmers as variable names.

You've already seen one: __init__() initializes a newly created object from its class definition a

Suppose that you have a simple Word class, and you want an equals()
method that compares two words but ignores case.

That is, a Word containing the value 'ha' would be considered equal to one containing 'HA'.

The example that follows is a first attempt, with a normal method we're
calling equals().

self.text is the text string that this Word object contains, and the
equals() method compares it with the text string of word2 (another Word
object):
```

```
class Word():
  def __init__(self, text):
    self.text = text
  def equals(self, word2):
    return self.text.lower() == word2.text.lower()
```

```
#Then, make three Word objects from three different text strings:
first = Word('ha')
second = Word('HA')
third = Word('eh')
```

```
#When strings 'ha' and 'HA' are compared to lowercase, they should be equal:
first.equals(second)
```

```
#But the string 'eh' will not match 'ha':
first.equals(third)
```

We defined the method equals() to do this lowercase conversion and comparison.

It would be nice to just say if first == second, just like Python’s built-in types.

So,let’s do that.

We change the equals() method to the special name __eq__() (you’ll see why in a moment):

```
class Word():
  def __init__(self, text):
    self.text = text
  def __eq__(self, word2):
    return self.text.lower() == word2.text.lower()
```

```
#Let’s see whether it works:
first = Word('ha')
second = Word('HA')
third = Word('eh')
first == second
```

```
first == third
```

All we needed was the Python’s special method name for testing equality, __eq__().

Table 10-1. Magic methods for comparison