Inheritance

Inheritance

BIG topic in OOP

Person class

```python
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __str__(self):
        return self.first_name + " " + self.last_name

person = Person("John", "Doe")
print(person)
```

Teacher class

What if we want to define a Teacher class?

A teacher ***is*** a person, and should be able to do everything a person can do.

Teacher class: a copy of Person class

One approach is to simply copy-paste the Person class…

```
class Teacher:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __str__(self):
        return self.first_name + " " + self.last_name
```

What's wrong with this approach???

Teacher class: a copy of Person class

What’s wrong with copy-paste Person class into Teacher class?

- Lots of code duplication
- If we want to print the last name before the first name, now we have to change it in two places

And let’s consider the real-world scenario:

- A person/teacher would have hundreds if not more attributes
- There are many other types of persons: students, doctors, accountants, etc…

Very quickly this becomes an <u>unmaintainable</u> code base...

Teacher class: by inheritance

The parent, or **base**, or super-, class

```
class Teacher(Person):
    pass
```

The child, or **derived**, or sub-, class

`pass` is use to indicate an empty block (in class, function, conditional, etc)

What can the Teacher class do?

Teacher class: by inheritance

The parent, or **base**, or super-, class

```
class Teacher(Person):
    pass
```

The child, or **derived**, or sub-, class

`pass` is use to indicate an empty block (in class, function, conditional, etc)

What can the Teacher class do?

Derived class

What can a **derived** class do?

The **derived** class has all the capabilities of its **parent** Person class. Literally, it **inherits** everything (attributes, method, everything).

```
teacher = Teacher("Z", "Yang")
print(teacher)
```

Derived class

How does inheritance address our previous concerns?

- Lots of code duplication
- If we want to print the last name before the first name, now we have to change it in two places

Derived class

How does inheritance address our previous concerns?

- Lots of code duplication
- If we want to print the last name before the first name, now we have to change it in two places

Well, the benefits of inheritance (partial list):

- Code reuse (the Teacher class has almost no code)
- Common behavior can be easily modified (let's try print the last name first)

is-a relationship

The relationship between the **derived** class and its **base** class is a “**is-a**” relationship.

Examples:

- A teacher is a person.
- An electric car is a car
- A dog is an animal

is-a relationship vs ...

"**is-a**" relationship is in contrast to ___ (a relationship that we've been using):

Examples:

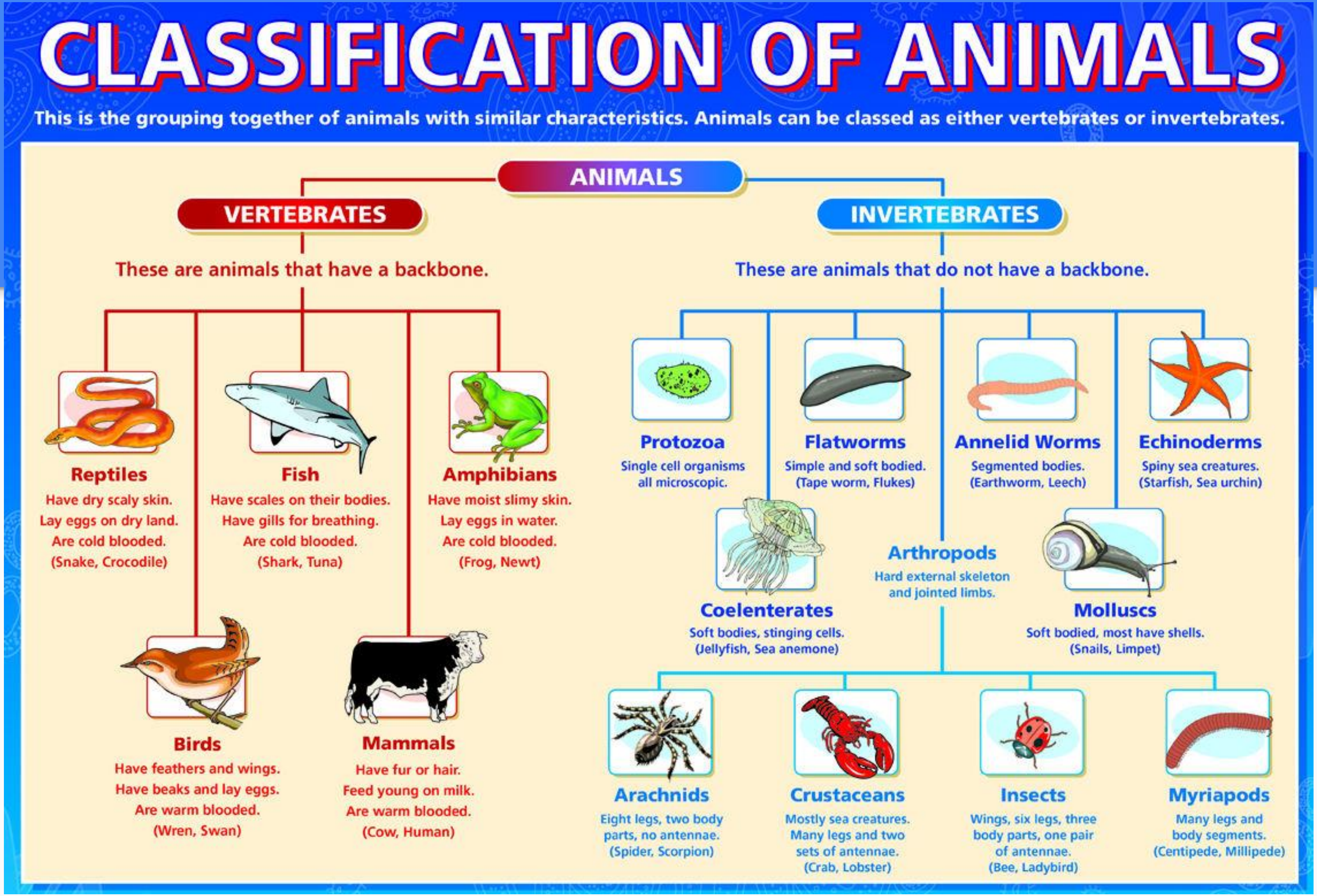
- A person ___ a name.
- A car ___ an engine
- A dog ___ a leg

has-a relationship

The relationship between a **class** and its **attributes** is as a “**has-a**” relationship.

Examples:

- A person has a name.
- A car has an engine
- A dog has a leg


CLASSIFICATION OF ANIMALS
This is the grouping together of animals with similar characteristics. Animals can be classed as either vertebrates or invertebrates.
ANIMALS
VERTEBRATES
These are animals that have a backbone.
INVERTEBRATES
These are animals that do not have a backbone.
Reptiles
Have dry scaly skin.
Lay eggs on dry land.
Are cold blooded.
(Snake, Crocodile)
Fish
Have scales on their bodies.
Have gills for breathing.
Are cold blooded.
(Shark, Tuna)
Amphibians
Have moist slimy skin.
Lay eggs in water.
Are cold blooded.
(Frog, Newt)
Birds
Have feathers and wings.
Have beaks and lay eggs.
Are warm blooded.
(Wren, Swan)
Mammals
Have fur or hair.
Feed young on milk.
Are warm blooded.
(Cow, Human)
Protozoa
Single cell organisms
all microscopic.
Flatworms
Simple and soft bodied.
(Tape worm, Flukes)
Annelid Worms
Segmented bodies.
(Earthworm, Leech)
Echinoderms
Spiny sea creatures.
(Starfish, Sea urchin)
Coelenterates
Soft bodies, stinging cells.
(Jellyfish, Sea anemone)
Arthropods
Hard external skeleton
and jointed limbs.
Molluscs
Soft bodied, most have shells.
(Snails, Limpet)
Arachnids
Eight legs, two body
parts, no antennae.
(Spider, Scorpion)
Crustaceans
Mostly sea creatures.
Many legs and two
sets of antennae.
(Crab, Lobster)
Insects
Wings, six legs, three
body parts, one pair
of antennae.
(Bee, Ladybird)
Myriapods
Many legs and
body segments.
(Centipede, Millipede)

Derived class: adding capabilities

A teacher has attributes that a normal person doesn't, e.g. salary.

So the constructor `__init__()` takes an extra parameter:

```
class Teacher(Person):
    def __init__(self, first_name, last_name, salary):
        self.first_name = first_name
        self.last_name = last_name
        self.salary = salary

teacher = Teacher("Z", "Yang", 1)
print(teacher)
```

This **overrides** the base's `__init__()`

Additional instance attribute, `self.salary`

Derived class: adding capabilities

So the `__init__()`, which didn't show the cwid. We should update that, too.

This **overrides** the base's `__str__()`

```
class Teacher(Person):
    ...
    def __str__(self):
        return self.first_name + " " + self.last_name + ", " + self.salary

teacher = Teacher("Z", "Yang", 1)
print(teacher)
```

Derived class: adding capabilities

What's wrong with this:

```
class Teacher(Person):
    def __init__(self, first_name, last_name, salary):
        self.first_name = first_name
        self.last_name = last_name
        self.salary = salary

    def __str__(self):
        return self.first_name + " " + self.last_name + ", " + self.cwid
```

Derived class: adding capabilities

What's wrong with this:

```
class Teacher(Person):
   def __init__(self, first_name, last_name, salary):
       self.first_name = first_name
       self.last_name = last_name
       self.salary = salary

   def __str__(self):
       return self.first_name + " " + self.last_name + ", " + self.cwid
```

The benefits of inheritance are:

- Code reuse
- Common behavior can be easily modified

But by adding new capabilities this way, we lose both benefits.

Derived class: super()

There's a better way, by calling the base's methods.

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

class Teacher(Person):
   def __init__(self, first_name, last_name, salaray):
       self.first_name = first_name
       self.last_name = last_name
       self.salary = salary
```

Same two lines of code

Derived class: super()

There's a better way, by calling the base's methods.

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

class Teacher(Person):
    def __init__(self, first_name, last_name, salaray):



        self.salary = salary
```

Can we just remove the two lines?
NO. The instance won't have first_name and last_name attributes.

Derived class: super()

There's a better way, by calling the base's methods.

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

class Teacher(Person):
    def __init__(self, first_name, last_name, salary):
        super().__init__(first_name, last_name)

        self.salary = salary
```

`super()` refers to the base's instance, so `super().__init__()` call this

Derived class: super()

There's a better way, by calling the base's methods.

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

class Teacher(Person):
    def __init__(self, first_name, last_name, salary):
        Person.__init__(self, first_name, last_name)

        self.salary = salary
```

Alternative syntax, but it's not a good idea to explicitly identify the base class, since the inheritance hierarchy can change

Derived class: super()

How about the `__str__()` method?

Derived class: super()

How about the `__str__()` method?

```
class Person:
    ...
    def __str__(self):
        return self.first_name + " " + self.last_name

class Teacher(Person):
    ...
    def __str__(self):
        return super().__str__() + " (salary=" + str(self.salary) + ")"
```

Derived class: overriding method

How are these two overrides different?

```
class Person:
    def __init__(self, fname, lname):
        ...

class Teacher(Person):
    def __init__(self, fname, lname, sal):
        ...
```

```
class Person:
    def __str__(self):
        ...

class Teacher(Person):
    def __str__(self):
        ...
```

Derived class: overriding method

How are these two overrides different?

```python
class Person:
    def __init__(self, fname, lname):
        ...

class Teacher(Person):
    def __init__(self, fname, lname, sal):
        ...
```

Different “signature”

```python
class Person:
    def __str__(self):
        ...

class Teacher(Person):
    def __str__(self):
        ...
```

Same signature

Derived class: overriding method

```
class Person:
    def __str__(self):
        ...

class Teacher(Person):
    def __str__(self):
        ...
```

When a **base** class's method is overridden by a method with the same signature, it cannot be accessed from a **derived** class any more (other than through `super()`).

Derived class: overriding method

```
class Person:
    def __init__(self, fname, lname):
        ...

class Teacher(Person):
    def __init__(self, fname, lname, salary):
        ...

teacher = Teacher("Z", "Yang")
```

Accessing the base class's `__init__()`?

What about a base class's method is overridden by a method with a <u>different</u> signature (same name, different number of parameters)?

Derived class: overriding method

```
class Person:
    def __init__(self, fname, lname):
        ...

class Teacher(Person):
    def __init__(self, fname, lname, salary):
        ...

teacher = Teacher("Z", "Yang")
```

Accessing the base class's `__init__()`?
NO. Recall that in the symbol table, a **name** can only be associated with one method. So in the **derived** class, the name `__init__` is now associated with the new definition, replacing the definition in the base.

Derived class: non-overridden methods

```
class Person:
    ...
    def initial(self):
        return self.first_name[0] + self.last_name[0]

class Teacher(Person):
    ...
    def sign_with_initial(self):
        print "Signing initial " + super().initial()
```

Call the base class's method. This works, but note that `initial()` is <u>not overridden</u>...

Derived class: non-overridden methods

```
class Person:
    ...
    def initial(self):
        return self.first_name[0] + self.last_name[0]

class Teacher(Person):
    ...
    def sign_with_initial(self):
        print "Signing initial " + super().initial()
```

BUT, if we now override the `initial()` method, this will not call it, because it's hard-coded to call the **base**'s `initial()` method.

Derived class: non-overridden methods

```
class Person:
    ...
    def initial(self):
        return self.first_name[0] + self.last_name[0]

class Teacher(Person):
    ...
    def sign_with_initial(self):
        print "Signing initial " + self.initial()
```

Since `initial()` is <u>not overridden</u>, can use `self`; if a method is not found in the derived class, Python interpreter will go <u>up</u> the class hierarchy to look for it.

Derived class: calling base class’s methods

So why do we need `super()`?

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

class Teacher(Person):
    def __init__(self, first_name, last_name, salary):
        self.__init__(first_name, last_name)

        self.salary = salary
```

Why not do the same here?

Derived class: calling base class's methods

So why do we need `super()`?

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

class Teacher(Person):
    def __init__(self, first_name, last_name, salary):
        self.__init__(first_name, last_name)

        self.salary = salary
```

Why not do the same here? This method is overridden in the derived class, so now it's a recursive call to itself!!!

Derived class: calling base class's methods

To call a **base** class's methods…

- should generally dereference `self`

unless…

- we are calling the **base** class's method from inside the **derived** class's **overridden** method (of the same name, obviously), in which case `super()` should be used (to avoid infinite recursion)

Derived class: calling base class's methods

When should you call the base class's methods?

- `__init__():`
- `__str__():`
- Others:

Derived class: calling base class's methods

When should you call the base class's methods?

- `__init__()`: should (almost?) always call `super().__init__()`, to ensure the instance of the **derived** class ***is*** an instance of the **base** class
- `__str__()`: usually calls super.__str__(), and append more info to it
- Others: depends

Polymorphism

Polymorphism

Overriding a **base** class's method in the **derived** class allows the derived class to behave differently. It is an example of **polymorphism**.

It's a very useful concept in computer science.

(Polymorphism: the ability to assume different forms or shapes.)

Overriding method: another example

```python
class Person:
    ...
    def worries(self):
        return ["money"]

class Teacher(Person):
    ...
    def worries(self):
        return super().worries() + ["lectures"]

teacher = Teacher("Z", "Yang", 1)
print(str(teacher) + " worries about " + teacher.worries())
```

Power of polymorphism

Let’s say, given a list of persons, we want to print their worries.

```
def print_worries(people):
    for p in people:
        print(str(p) + " worries about " + str(p.worries()))

print_worries([person, teacher])
```

⇨

```
John Doe worries about ['money']
Z Yang (salary=1) worries about ['money', 'lectures']
```

Same method call.
Different behaviors..

Power of polymorphism: real-world example

Content Management System (CMS) that stores different types of content:

- PDF (creator, created date, size)
- Word .doc (creator, created date, size)
- Pictures (creator, created date, size, dimension)
- Videos (creator, created date, size, dimension, duration)

Power of polymorphism: real-world example

Minecraft (apologies to non-gamers) blocks:

- Dirt block: `touched()` -> disappears
- Stone block: `touched()` -> nothings happens
- Fire block: `touched()` -> burned, reduce health
- Diamond block: `touched()` -> gets stored in inventory

And to introduce new type of blocks to the system, define the block with `touched()` method; the game engine does <u>not</u> change.

Power of polymorphism

Using the new type of person:

```
def print_worries(people):
   for p in people:
       print(str(p) + " worries about " + str(p.worries()))

...
student = Student("Jane", "Smith", 1)
print_worries([person, teacher, student])
⇨
John Doe worries about ['money']
Z Yang (salary=1) worries about ['money', 'lectures']
Jane Smith (cwid=1234) worries about ['money', 'quizzes', 'assignments']
```

<u>No</u> change in code.
<u>New</u> behavior due to new <u>type</u>.

Polymorphism: type checking

It might make sense to ensure the list has only `Person` in it:

Does this check always work?

```
def print_worries(people):
    for p in people:
        if type(p) == Person:
            print(str(p) + " worries about " + str(p.worries()))

print_worries([person, teacher, student, "not a person"])
```

Polymorphism: type checking

It might make sense to ensure the list has only `Person` in it:

```
def print_worries(people):
    for p in people:
        if type(p) == Person:
            print(str(p) + " worries about " + str(p.worries()))

print_worries([person, teacher, student, "not a person"])
```

Does this check always work?
NO. Some of the people's **types** are `Student`, `Teacher`.

Polymorphism: type checking

It might make sense to ensure the list has only `Person` in it:

```
def print_worries(people):
    for p in people:
        if isinstance(p, Person):
            print(str(p) + " worries about " + str(p.worries()))

print_worries([person, teacher, student, "not a person"])
```

`Student`, `Teacher` are `Person`, so check if the instance is <u>an instance of the type</u> `Person`, using `isinstance()`, which understands inheritance.