* Relationships
  + OOP models three types of real-world relationships: **is‑a**, **has‑a** and **is‑a‑type‑of**
    - Donald **is a** duck
    - A duck **has a** bill
    - A duck **is a type of** bird
* Is-a 🡪 Instantiation
  + “X is a Y” means “the specific object X is an object of the type Y”
  + Is-a is modelled by classes and instances:
    - “Donald is a duck” 🡪 “Donald is an instance of the class Duck”
* Has-a 🡪 Composition
  + “X has a Y” means “an object of type X possesses an object of type Y”
  + OOP models this by having a field on X which holds a reference to an instance of Y
    - “A duck has a bill” 🡪 “The class Duck has a field which points to an instance of the class Bill”
* Is-a-type-of 🡪 Inheritance
  + “X is a type of Y” means “If an object is of type X, then it is also of type Y”
    - “A duck is a type of bird” 🡪 “If something is a duck, then it is also a bird”
    - “Every duck is a bird”
    - “If something is true for all birds, then it must be true for ducks”
  + In OOP terms, this is called **inheritance**
* Inheritance
  + Recall: an object is a collection of fields (data) and methods (code)
  + Recall: the class defines which fields and methods an object possesses
  + “X is a type of Y” 🡪 class X **inherits** from class Y
  + Class X inherits all of the fields and methods from class Y, as well as any fields and methods of its own
* When to inherit?
  + When modelling an is-a-type-of relationship from the real world
  + When several classes can share some fields and/or methods
    - I.e. to minimise code duplication
  + When several classes should have methods with the same names, but which do different things
    - This is called **polymorphism** – more on this later
* Inheritance in Python
* Inheritance in Python
* Inheritance in Python
* Inheritance in Python
* Inheritance in Python
* Chains of inheritance
  + “A mallard is a type of duck, which is a type of bird, which is a type of vertebrate, which is a type of animal…”
  + Is-a-type-of is **transitive**
    - If A is-a-type-of B and B is-a-type-of C, then A is-a-type-of C
  + Likewise: class A inherits from class B, which inherits from class C, …
    - “Inherits from” is also transitive
* A possible inheritance hierarchy
* OOP: Polymorphism
* Polymorphism
  + From Greek: “many-shape-ism”
  + Different classes can have the same public interface
  + Thus we can write code that uses this interface, but doesn’t need to worry about the implementation behind it
* Method overriding
  + A class can override methods defined in the class from which it inherits
  + The overridden method can call the method from the base class, but it doesn’t have to
* Without polymorphism
  + We have a list of shapes, and want to draw them all
  + This approach is messy and difficult to maintain
* Polymorphism to the rescue!
  + All subclasses of Shape implement draw
  + We can call shape.draw() without worrying which type of shape it is
* Abstract classes and methods
  + Some classes should never be instantiated directly, as they only exist to be inherited from
    - Shape is an example
  + Such classes are called **abstract**
  + **Abstract methods** are methods of an abstract class which are left unimplemented, so must be implemented in subclasses
    - draw is an example
* OOP: Access control
* Access control
  + For encapsulation, it is a good idea to restrict access to certain attributes and methods from outside the class
  + **Private** members are only accessible from the class’s own methods
  + **Protected** members are accessible from the class’s own methods, and methods defined in subclasses
  + **Public** members are accessible from outside the class
* Access control
  + The **public** interface of an object is how it interacts with other objects and the rest of the program
  + The **protected** interface of an object is what allows subclasses to change the way the base class behaves
  + The **private** members of an object are implementation details, hidden from the outside world
* Access control in Python
  + A field or method whose name begins with \_ is protected
  + A field or method whose name begins with \_ \_ is private
    - Names which begin and end with \_ \_ (like \_\_init\_\_) are special – don’t use trailing underscores in your own names
  + Everything else is public
* Pedantic detail
  + Access control is merely a convention in Python
    - The interpreter won’t stop you from accessing protected members from outside the class or its subclasses, but PyCharm will warn you
    - Private members are “name mangled”, but you can still access them if you know how
  + Almost all other languages enforce access control with compile-time errors
* Summary
  + OOP models three main types of real-world relationships
    - Is-a 🡪 instantiation
    - Has-a 🡪 composition
    - Is-a-type-of 🡪 inheritance
  + Inheritance allows polymorphism: different classes with the same public interface
  + Access control is an important tool in designing reusable, encapsulated objects