INF1-IP

# Introduction to Programming

Lecture 16 – Objects 2: Inheritance

# > Course updates

- Assignment 3 released today
- Assignment 2 grades released today
- Upcoming assessment:
  - Third assignment quiz – Week 10 tutorials
  - Third class test slot – Thursday Week 11
  - Catch-up assessment (assignment quiz and/or class test) – Thursday Week 12 plus additional session for those with exam clashes

# Inheritance

# > Short diversion: UML diagrams

Unified Modelling Language (UML) diagrams is a standard visual notation for describing software systems, including class diagrams and relationships between classes.

Why are we looking at this?

- UML diagrams are language agnostic so are not part of Python.  But useful to know, so fits into an Intro to Programming course.
- Not related to inheritance, but very helpful in visualising it – so now seems a good time!

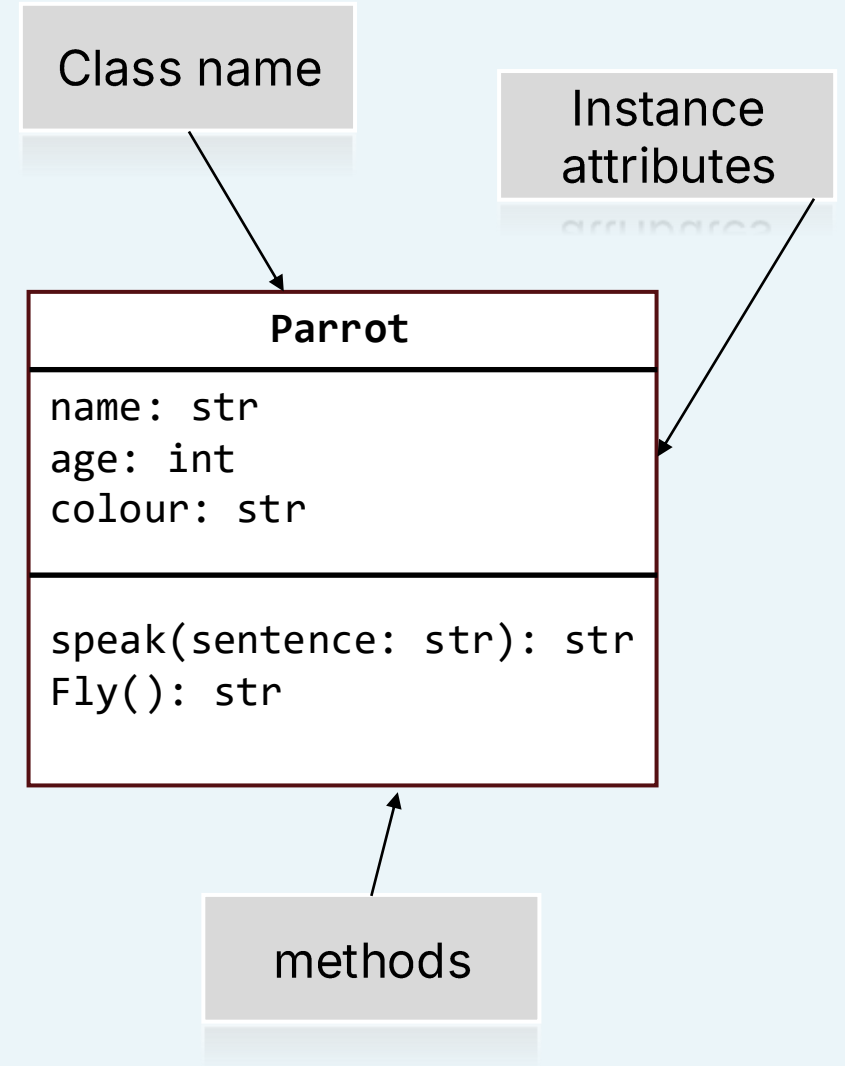
# > UML diagrams

```python
class Parrot:
    animal_class = "bird"
    can_fly = True

    def __init__(self, name, age, colour):
        self.name = name
        self.age = age
        self.colour = colour

    def speak(self, sentence):
        return f"{self.name} says: {sentence}!"

    def fly(self):
        if self.can_fly:
            print(f"{self.name} is flying!")
        else:
            print(f"{self.name} is grounded...")
```



# Inheritance – actually, this time ...

# > Related classes

An important coding principle we have seen before:

Don’t Repeat Yourself (DRY coding)

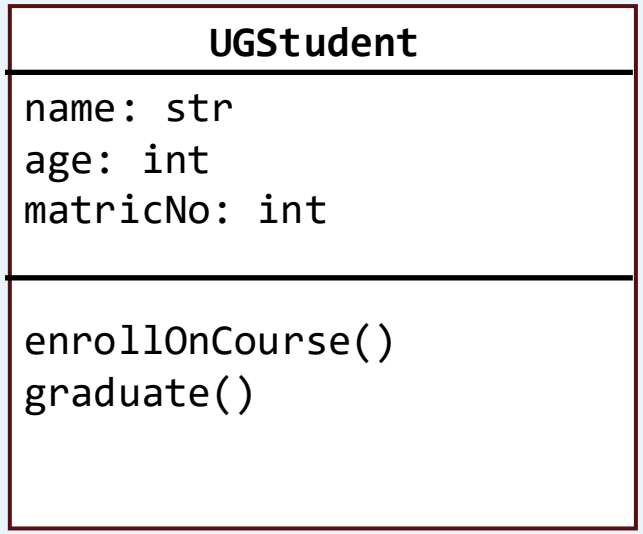
# > Related classes

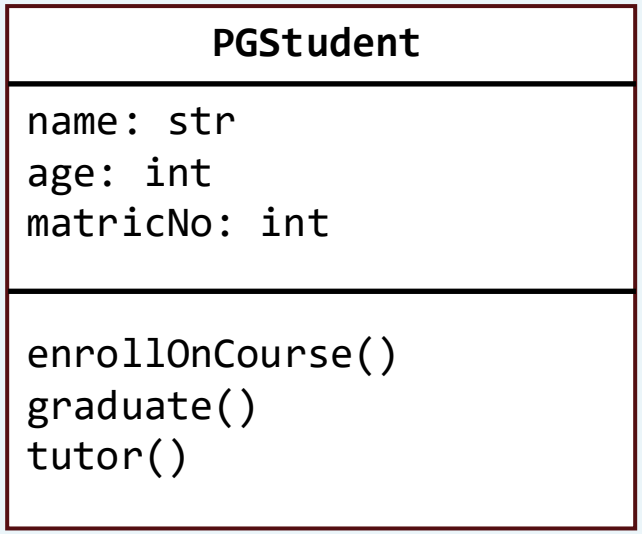
An important coding principle we have seen before:

Don’t Repeat Yourself (DRY coding)

What if we have two or more classes that have lots of similarities, but some differences?

# > Related classes



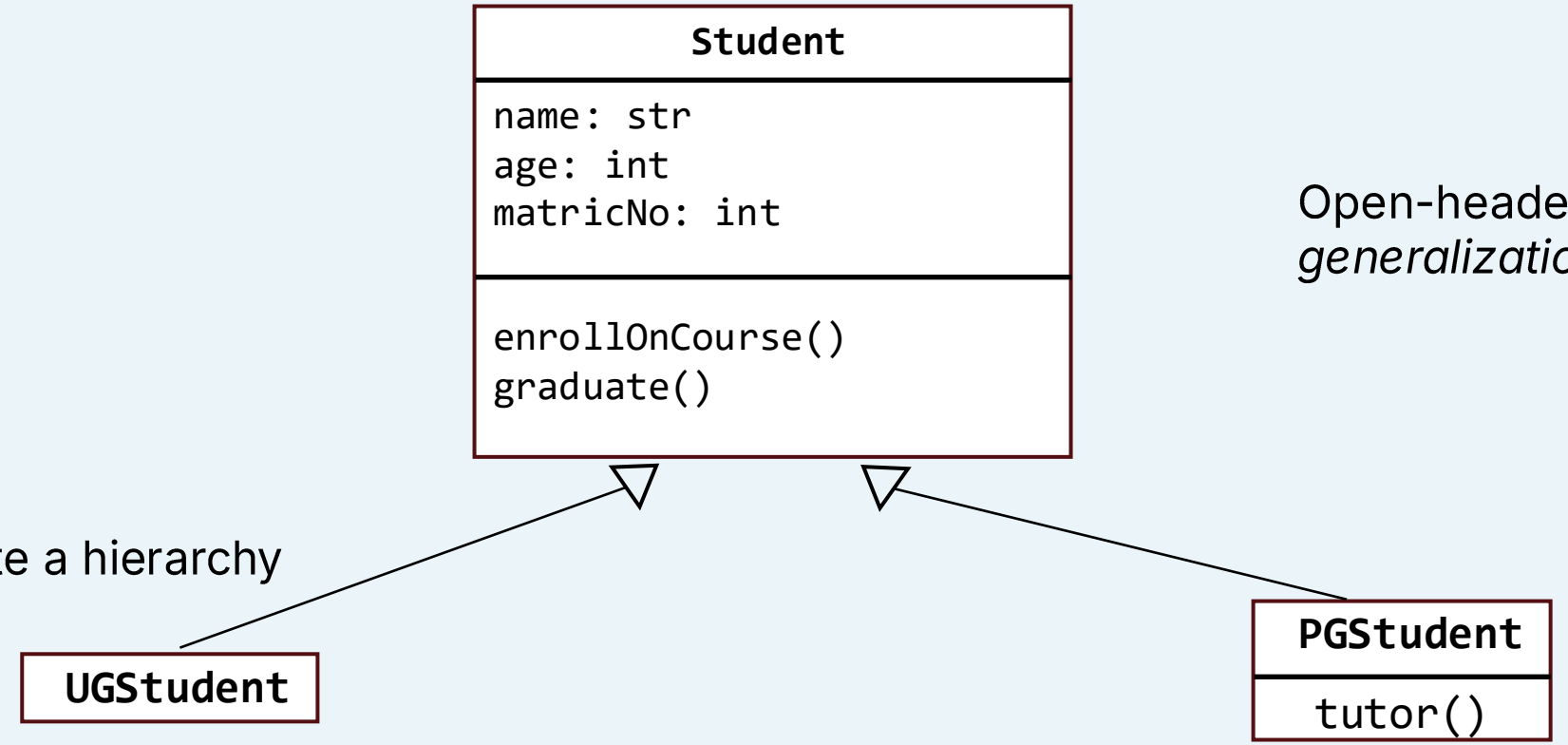




These are not the same class, but they have methods in common.

- We don't want to make them the same class, because these differences are important
- If `graduate()` and `enrollOnCourse()` are the same for both, we don't want to create local copies. Why? DRY coding!

# > Inheritance hierarchies

So we *extract what is common* to both classes:




Open-headed arrows indicated *generalization* in UML class diagrams

Create a hierarchy

Subclasses (UG, PG) *inherit* from superclass (Student)

# What does this look like in Python?

```python
class Student:
    def __init__(self, name: str, age: int, matricNo: str):
        self.name = name
        self.age = age
        self.matricNo = matricNo
        self.courses = []
        self.hasGraduated = False

    def enrollOnCourse(self, course: str) -> None:
        self.courses.append(course)

    def graduate(self) -> None:
        self.hasGraduated = True
```

```python
class UGStudent(Student):
    pass


class PGStudent(Student):
    def tutor(self) -> None:
        print(f"{self.name} is tutoring.")
```

When creating subclasses, the superclass (or base class) appears in round brackets after the name.

# { What does this look like in Python?

# > Should we use an inheritance hierarchy?

You should only use an inheritance hierarchy if you have an *is-a* relationship.

Does it make sense to say an instance of class X *is a(n)* instance of class Y?  If so, this is a subclass relationship.

Examples:
Kitchen subclass-of Room
Bash subclass of TraitorsCompetitor
Violinist subclass of Musician
Musician subclass of Person

# > Should we use an inheritance hierarchy?

Let's test some examples - wooclap

Remember:

- For a subclass relationship, every member of the subclass must inevitably be a member of the superclass.
  - *It's not enough for that sometimes to be true*
- Beware of part-of relationships – these do not belong in inheritance hierarchies!

# { Your turn

- Create an `Animal` class with a name attribute and an `eat()` method that returns a message.
- Make `Rabbit` a subclass without defining any new methods.
- Create a rabbit and call its inherited `eat()` method.

# { Your turn

```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def eat(self) -> str:
        return f"{self.name} is eating."


class Rabbit(Animal):
    pass


rabbit = Rabbit("Clover")
print(rabbit.eat())  # Clover is eating.
```

# > Should we use an inheritance hierarchy?

We've seen that `PGStudent` has a method that its superclass `Student` does not have.
Subclasses can also have attributes that their superclasses don't have.
This is more tricky, as attributes need to be initialised.

What if PGStudent needs an attribute to say if the student has done tutor training?

# Initialising variables in subclasses

```
class PGStudent(Student):
    def tutor(self) -> None:
        print(f"{self.name} is tutoring.")
```

Case 1: extra methods, no extra attributes:
→ just add extra methods in the subclass definitio

# Initialising variables in subclasses

```python
class PGStudent(Student):
    def tutor(self) -> None:
        print(f"{self.name} is tutoring.")
```

```python
class PGStudent(Student):
    def __init__(
        self,
        name: str,
        age: int,
        matricNo: str,
        tutorTraining: bool = False
    ):
        super().__init__(name, age, matricNo)
        self.tutorTraining = tutorTraining

    def tutor(self) -> None:
        print(f"{self.name} is tutoring.")
```

Case 1: extra methods, no extra attributes:
→ just add extra methods in the subclass definitio

Case 2: extra attributes:
→ these attributes need to be initalised in the subclass definition.

# Initialising variables in subclasses

```python
class PGStudent(Student):
    def tutor(self) -> None:
        print(f"{self.name} is tutoring.")
```

```python
class PGStudent(Student):
    def __init__(
        self,
        name: str,
        age: int,
        matricNo: str,
        tutorTraining: bool = False
    ):
        super().__init__(name, age, matricNo)
        self.tutorTraining = tutorTraining

    def tutor(self) -> None:
        print(f"{self.name} is tutoring.")
```

Case 1: extra methods, no extra attributes:
→ just add extra methods in the subclass definitio

Case 2: extra attributes:
→ these attributes need to be initalised in the subclass definition.

1. Use __init__ to initialise all attributes of the subclass

# Initialising variables in subclasses

```python
class PGStudent(Student):
    def tutor(self) -> None:
        print(f"{self.name} is tutoring.")
```

```python
class PGStudent(Student):
    def __init__(
        self,
        name: str,
        age: int,
        matricNo: str,
        tutorTraining: bool = False
    ):
        super().__init__(name, age, matricNo)
        self.tutorTraining = tutorTraining

    def tutor(self) -> None:
        print(f"{self.name} is tutoring.")
```

Case 1: extra methods, no extra attributes:
→ just add extra methods in the subclass definitio

Case 2: extra attributes:
→ these attributes need to be initalised in the subclass definition.

1. Use __init__ to initialise all attributes of the subclass
2. But we (usually) want the attributes that are shared by the superclass to be initialised by the superclass:
   Use super().__init__ to initialize attributes vias the superclass

# { Your turn

- Create an `Animal` class with a name attribute and an `eat()` method that returns a message.
- Make `Rabbit` a subclass without defining any new methods.
- Create a rabbit and call its inherited `eat()` method.
- We now want to record information about burrow location for rabbits. This is something that doesn't apply to animals in general, but only to rabbits. Add an attribute to `Rabbit` to reflect this.
- Create a rabbit with this new structure and print out its name, burrow location and it's eating status.

# { Your turn

```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def eat(self) -> str:
        return f"{self.name} is eating."


class Rabbit(Animal):
    def __init__(self, name: str, burrowLocation: str):
        super().__init__(name)
        self.burrowLocation = burrowLocation


rabbit = Rabbit("Clover", "Under the oak tree")

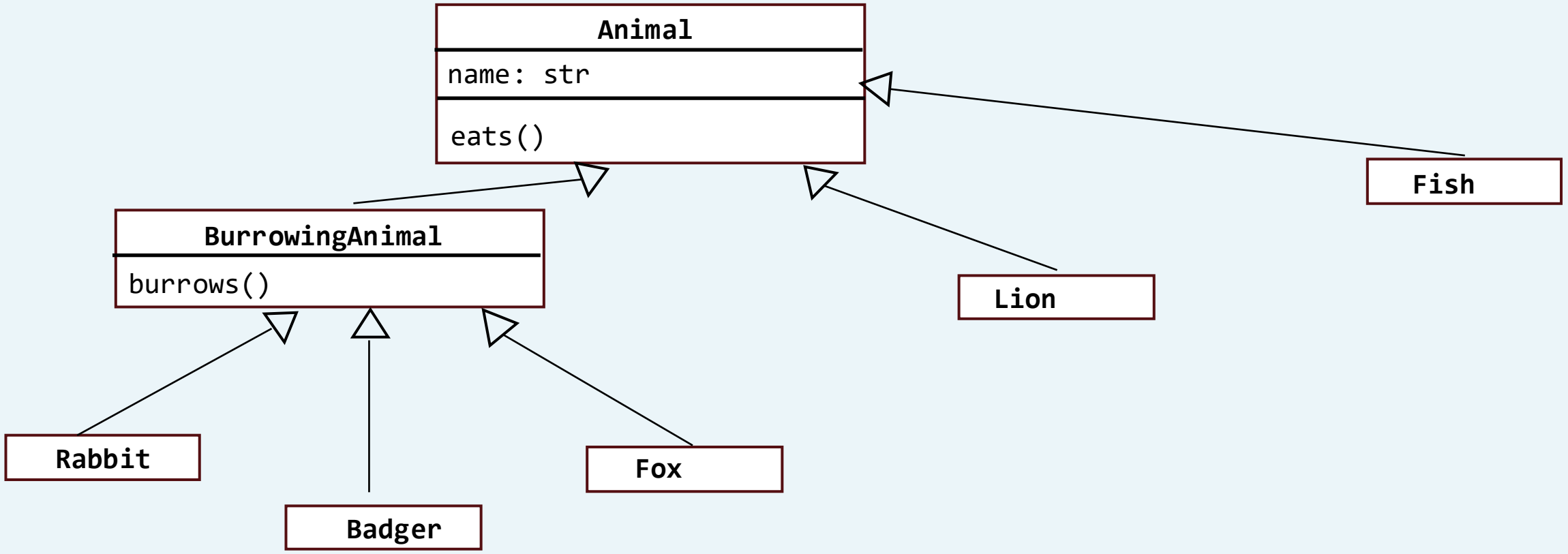
print(rabbit.name)            # Clover
print(rabbit.burrowLocation)  # Under the oak tree
print(rabbit.eat())           # Clover is eating.
```

# ! Where do attributes belong?

Attributes belong to the *most general class that it applies to.*

- My animals are: lion, dog, rabbit, fish
  - Eating belongs in `Animal` (all animals eat)
  - Burrowing belongs in `Rabbit` (only rabbits burrow)
- Now my animals are: lion, dog, fox, rabbit, fish, badger
  - I have several animals that burrow → it might make sense to make a ‘burrowing animals’ class.

# ! Where do attributes belong?



# > How do you create a hierarchy?

We have divided animals up into burrowing and not burrowing animals.
But there are lots of ways we could do this!
e.g., mammals/non-mammals; European/Asian/African/American; has fur/has scales/other, etc

This decision is *pragmatic*. If we are wanting to create code that involves burrowing, or whether an animal is above or below ground, creating a category for burrowing animals might be useful. Otherwise, it might not.

# > Overriding superclass attributes

Subclass inherit methods from superclasses.

How does a rabbit eat?  This is defined by the Animal class.

But if we want we can override this?

We just define this method within Rabbit.

Within an inheritance hierarchy, Python will check the subclass for the method first, and if it doesn't find it, it will go to the next class up.

# { Your turn

- Create an `Animal` class with a name attribute and an `eat()` method that returns a message.
- Make `Rabbit` a subclass without defining any new methods.
- Create a rabbit and call its inherited `eat()` method.
- We now want to record information about burrow location for rabbits. This is something that doesn't apply to animals in general, but only to rabbits. Add an attribute to `Rabbit` to reflect this.
- Create a rabbit with this new structure and print out its name, burrow location and it's eating status.
- Now let's change the way rabbits eat to be different to other animals

# { Your turn

```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def eat(self) -> str:
        return f"{self.name} is eating."


class Rabbit(Animal):
    def __init__(self, name: str, burrowLocation: str):
        super().__init__(name)
        self.burrowLocation = burrowLocation

    def eat(self) -> str:
        return f"{self.name} is nibbling grass."


rabbit = Rabbit("Clover", "Under the oak tree")
print(rabbit.eat())  # Clover is nibbling grass.
```

# > Checking class relationship

As we've seen before, `type(value)` gives the exact class of a value.

`isinstance(value, Class)` returns `True` if the class of `value` is `Class`, or any superclass of `Class`.

Why? Think about the is-a relationship.

```
flopsy = Rabbit("Flopsy", "by the pond")
isinstance(flopsy, Rabbit) → True
isinstance(flopsy, BurrowingAnimal) → True
isinstance(flopsy, Animal) → True
isinstance(flopsy, Carnivore) → False
```

# > Creating a hierarchy

1. Look for objects that have common attributes and behaviours.
2. Design a class that represents the common state and behaviour.
3. Decide if a subclass needs method implementations that are specific to that particular subclass type.
4. Carry out further abstraction by looking for groups of subclasses that might have common behaviours

# > Encapsulation

When you use build-in methods, you don't care *how* they do things – you just need to know what they do.

This can also be true for the methods you are creating – if you or other people call these later, you often don't want to know the detail of how they work, you just need to know inputs and ouputs.

This is **encapsulation** – grouping an object's data and the methods that work on it into a class, providing a clear way to use the object without needing to know its internal details.

# > Encapsulation

```python
class Student:
    def __init__(self, name: str):
        self.name = name
        self._courses = []

    def enrollOnCourse(self, course: str) -> None:
        if course not in self._courses:
            self._courses.append(course)

    def getCourses(self) -> list[str]:
        return self._courses.copy()
```

```python
student = Student("Aisha")

student.enrollOnCourse("Python")
student.enrollOnCourse("Python")

print(student.getCourses())  # ['Python']
```

This allows the designing of the code to control how it works.

In this case, `enrollOnCourse` does not allow duplicates, so users calling it can't enroll one student on the same course twice.

# > Encapsulation

```python
class Student:
    def __init__(self, name: str):
        self.name = name
        self._courses = []

    def enrollOnCourse(self, course: str) -> None:
        if course not in self._courses:
            self._courses.append(course)

    def getCourses(self) -> list[str]:
        return self._courses.copy()
```

```python
student = Student("Aisha")

student.enrollOnCourse("Python")
student.enrollOnCourse("Python")

print(student.getCourses())  # ['Python']
```

Why does `_courses` start with an underscore?

This is to indicate to users that an attribute is internal and should not be changed outside the method.

# ! Be careful about respecting internal methods

In some languages, like Java, you can make attributes as *private*, which means they can only be changed within the method.

In Python, an underscore is used to show that an attribute *should not* be changed outwith the method, but you can still do this if you want.

```
student = Student("Aisha")

# Intended use: the method prevents duplicate courses.
student.enrollOnCourse("Python")

# Direct access is possible, despite the leading underscore.
student._courses.append("Python")

print(student.getCourses())  # ['Python', 'Python']
```

Doing this can create lots of problems – usually best avoided!

# > Is inheritance useful?

Yes, it is very powerful, but it has its downsides and should not be overused:

- Easy to get muddled with inheritance hierarchies – people sometimes put *part-of* relationships in inheritance hierarchies and this doesn't work!
- Subclass is tightly coupled with superclass.
- Changes in superclass can break subclass — fragile base class problem.

# > Summary

- Inheritance is useful when classes share attributes but have some differences
- Only works with an *is-a* relationship
- Subclasses use the methods of the superclass if they don't have their own methods defined
- You can *override* superclass methods by defining versions within subclasses