Understanding Object Oriented Programming in Python

Steven Wingett, Babraham Bioinformatics

version 2020-08

Babraham Bioinformatics

Licence

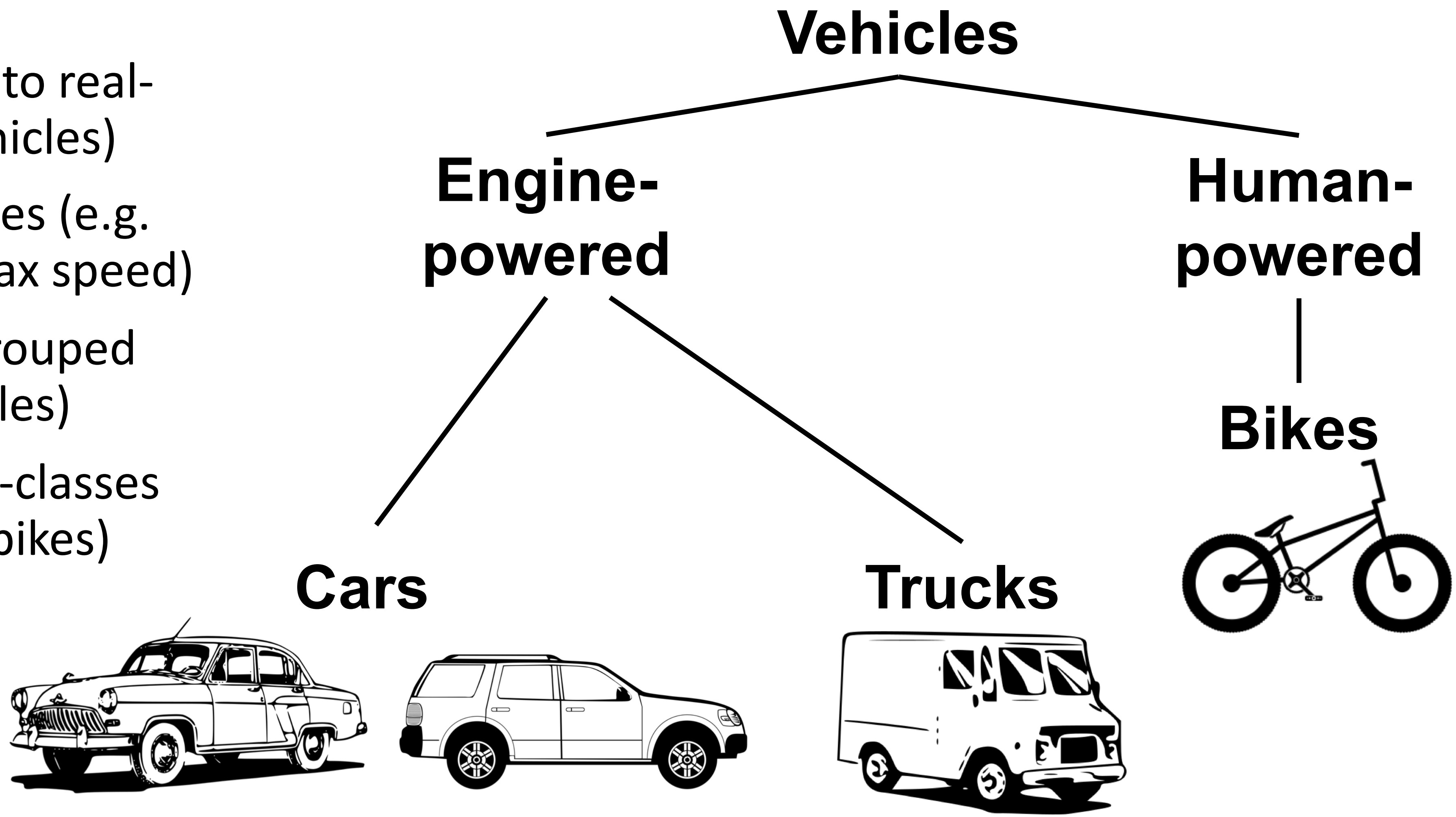
Babraham Bioinformatics

Introduction

- So far dealt with Python as a **procedural** language – a series of instructions (like a food recipe)
- Easy to loose track of everything for big projects
- Object-oriented programming (OOP) designed to make it easier to writing more complex projects
- It is better suited to the human brain

Introduction (2)

- Object are analogous to real-word objects (e.g. vehicles)
- Objects have properties (e.g. number of wheels, max speed)
- Related objects are grouped into classes (i.e. vehicles)
- And grouped into sub-classes (e.g. cars, trucks and bikes)



Defining classes

- **Let’s define a dog class** (this is not *a* dog, but the *concept* of a dog)
- (Maybe surprisingly, classes are objects as well!)

```
class Dog:
    pass
```

- Type `Dog()` into the interpreter:

```
<__main__.dog object at 0x0341D7B0>
```

- `__main__` is the name of the module to which the dog class belongs (main is the Python interpreter)
- Next is the name of the class followed by an internal memory address (written in hexadecimal)
- Classes by convention begin with capital letters

Instantiation

- To make an instance of the `dog` class, simply call the class as you would a function:

```
snoopy = Dog()
```

- This is known as **instantiation**

- This instance of the dog class is named `snoopy`. Similar to before, you may view its memory location:

```
>>> Dog
<__main__.dog object at 0x0410D7F0>
```

Instance attributes

- Instances of a class may have **methods** (such as already seen with built-in objects) and store information in what is known as **fields**
- Collectively, methods and fields are known as **attributes**
- Both of these may be accessed using the dot notation:

```
snoopy.colour = 'White'
```

- All other instances of the dog class will not have a colour field; only snoopy will be changed by this statement
- Although this is a simple and quick way to edit the snoopy instance, there are better ways to do this

Access methods

- Access method returns field values of an instance
- Use `def` to define a method (similar to a function)
- `self` refers to the **current instance** of a class

```
class Dog:
    def get_colour(self):
        return self.colour
```

```
>>> snoopy.get_colour()
'White'
```

- Why not simply use `snoopy.colour`? Well, with our method above, we can change the internal class code without causing problems.

Access methods (2)

- Access methods do not simply have to return a value:

Class: dog

```
class Dog:
    def get_colour(self):
        return self.colour

    def animate(self):
        if self.mood == 'Happy':
            return('Wag Tail')
        elif self.mood == 'Angry':
            return('Bite')
        else:
            return('Bark')
```

Code interacting with dog

```
snoopy = Dog()

snoopy.mood = "Happy"
print((snoopy.animate()))
snoopy.mood = "Angry"
print((snoopy.animate()))
>>>
Wag Tail
Bite
```

Predicate methods

- Return either a `True` or `False`
- By convention, begin with an `is_` prefix (or sometimes `has_`)

```
class Dog:
    stomach_full_percentage = 20
    def is_hungry(self):
        if(self.stomach_full_percentage < 30):
            return True
        else:
            return False

snoopy = Dog()
print(snoopy.is_hungry())
```

Babraham Bioinformatics

Predicate methods (2)

- Important method is the ability to compare and sort instances
- By convention, define an **`__lt__`** method to do this
- This method returns `True` or `False` (so is a predicate method)

Babraham Bioinformatics

Predicate methods (3)

Class: dog

```
class Dog:
    def get_age(self):
        return self.age

    def __lt__(self, other):
        if type(self) != type(other):
            raise Exception(
                'Incompatible argument to __lt__:' +
                str(other))
        return self.get_age() < other.get_age()
```

Code interacting with dog

```
snoopy = Dog()
snoopy.age = 9

scooby = Dog()
scooby.age = 6

print(snoopy.__lt__(scooby))

>>>
False
```

Initialisation methods

- Useful to set (or **initialise**) variables at time of creation
- Special initialisation method: **`__init__`**
- This is the usual way to assign values to all fields in the class (even if they are assigned to `None`)
- By convention, the `__init__` method should be at the top of the code in a class
- In the example, we pass self (first) and data to the `__init__` method

```
class Dog:
    def __init__(self, data):
        self.age = data

    def get_age(self):
        return self.age

snoopy = Dog(10)
print(snoopy.get_age())

>>>
10
```

String methods

- Methods that define how a class should be displayed
- `__str__` returned after calling `print`
- `__repr__` returned by the interpreter
- In example, human-friendly name returned instead of:
  `<__main__.dog object at 0x0405D6B0>`

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

    def __str__(self):
        return 'Dog:' + self.name

    def __repr__(self):
        return self.name

>>> dog1
Snoopy
>>> print(dog1)
Dog:Snoopy
```

Babraham Bioinformatics

Modification methods

Methods that **modify** class fields:

Code

```
class Dog:
    def __init__(self):
       self.mood = "Sad"

    def get_mood(self):
        return self.mood

    def set_mood(self, data):
        self.mood  = data

dog1 = Dog()
print(dog1.get_mood())
dog1.set_mood("Happy")
print(dog1.get_mood())
```

Output

```
>>>
Sad
Happy
```

Class attributes

- Up until now we have looked at attributes that work at the level of **each instance** of a class
- In contrast, there attributes which operate at the level of the whole class
- **Class fields** are declared at the top-level and begin with a capital letter
- **Class methods** have the special indicator **@classmethod** on the line immediately above
- Let’s see an example

Exercises

- Exercise 1.1 & 1.2

Class attributes (2)

Code

```
class Sheep:
    Counter = 0

    @classmethod
    def AddOne(self):
        self.Counter += 1

    def __init__(self):
        self.AddOne()
        self.id = self.Counter

    def get_id(self):
        return self.id

dolly = Sheep()
flossy = Sheep()
print(dolly.get_id())
print(flossy.get_id())
```

Class field

Class method

Output

```
>>>
1
2
```

Static methods

- Methods that can be called directly from a class, **without the need for creating an instance of that class**
- Special indicator **@staticmethod placed** on the line immediately above the definition
- Useful when we need to make use of a class's functionality but that class is not needed at any other point in the code

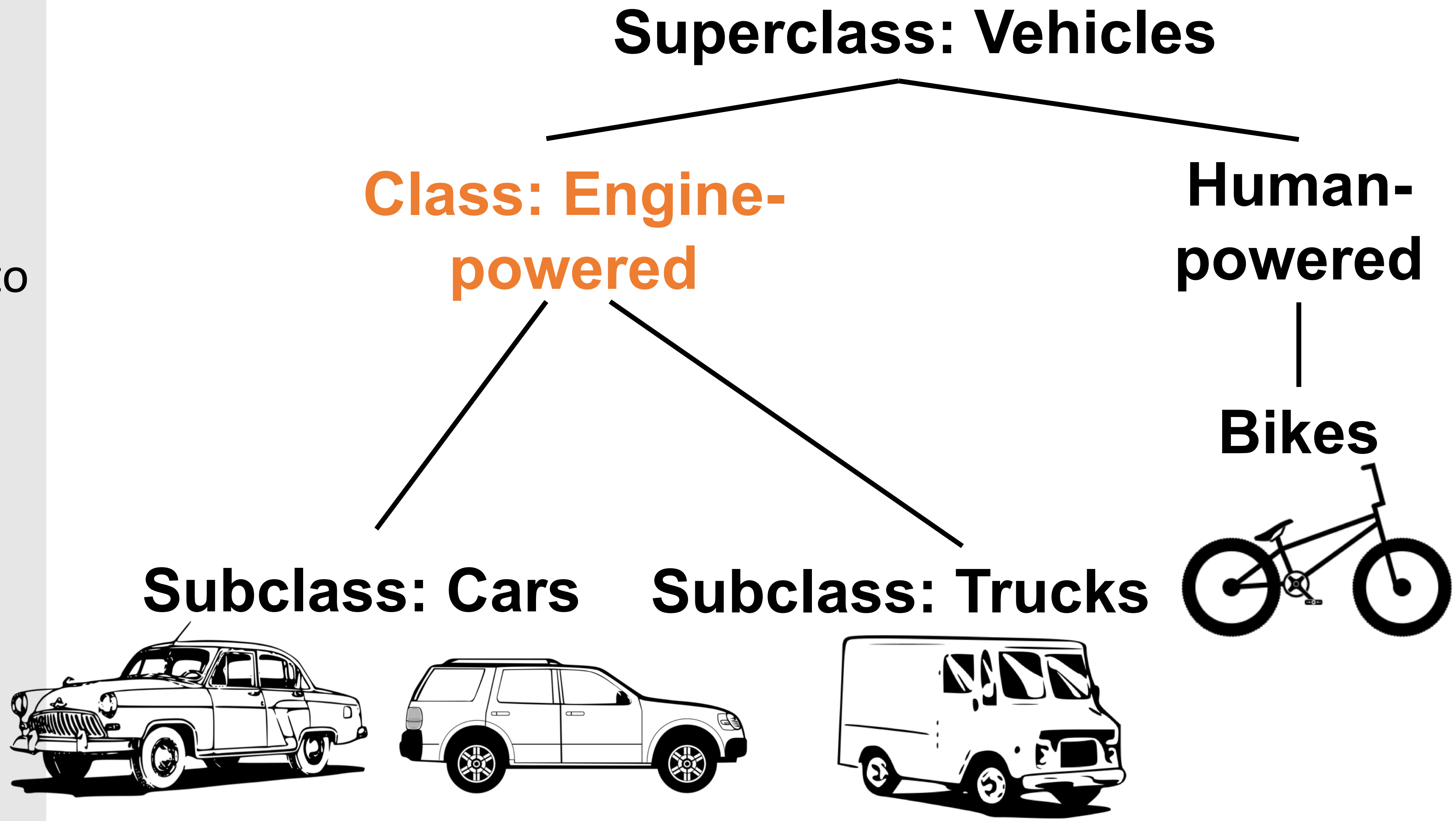
```
class Utilities:
    @staticmethod
    def miles_to_km(miles):
        return(miles * 1.60934)

journey = 10
journey_km = Utilities.miles_to_km(journey)
print(journey_km)

>>>
16.0934
```

Exercises

- Exercise 1.3

Inheritance

- **Inheritance** central to OOP
- **Subclass** "inherits" properties of parent class (now referred to as the **superclass**)
- Subclass can be modified to have different properties from parent class i.e. similar, but different
- Enables coders to produce objects with reduced codebase
- Reduces code duplication
- Changes only need to be made in one place



Inheritance (2)

Class Code

```
class Dog:
    def __init__(self):
       self.mood = "Sad"

    def get_mood(self):
        return self.mood

    def set_mood(self, data):
        self.mood  = data
```

"Main body" code

```
dog1 = Dog()
print(dog1.get_mood())
```

Output

```
>>>
Sad
```

Inheritance (3)

Superclass Code	Subclass Code	"Main body" code	Output

`class Dog:`
`    def __init__(self):`
`       self.mood = "Sad"`
 	
`    def get_mood(self):`
`        return self.mood`
 	
`    def set_mood(self, data):`
`        self.mood  = data`
	`class Rottweiler(Dog):`
	`    pass`
		`rottweiler1 = Rottweiler()`
		`print(rottweiler1.get_mood())`
			`>>>`
			`Sad`

Inheritance and super() (2)

- What was the point of that? The Rottweiler class does exactly the same as the dog class

- Well, once we have created a subclass, we can build on it. See the following example

Inheritance and super() (3)

Superclass

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width
```

Subclass

```
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
```

- In geometry, a square is a special type of rectangle
- Here, a `Square` is a subclass of `Rectangle`
- Unlike the rectangle, we only need to define the square's length on instantiation
- The keyword `super` refers to the superclass
- When initialising a square, we pass length twice to the initialisation method of the rectangle class
- We have therefore overridden the `__init__` method of rectangle
- We can override any superclass method be redefining it in the subclass

Babraham Bioinformatics

Exercises

- Exercise 2

Exercises

- Exercise 3, 4 and 5*

How do you get to Carnegie Hall? Practice, practice, practice.

Happy coding!

The Babraham Bioinformatics Team

Babraham Bioinformatics