On turtles and properties

The white Helge turtle

Jens: Helge, please take this white piece of chalk

Helge takes the chalk from Jens

Jens: Helge, please move to the left blackboard

Helge moves to the left-most blackboard

Jens: Helge, draw a circle with the chalk on the blackboard

Helge draws a circle with the chalk

The yellow Helge turtle

Jens: Helge, please take this yellow piece of chalk

Helge takes the chalk from Jens

Jens: Helge, please move to the center blackboard

Helge moves to the middle blackboard

Jens: Helge, draw a circle with the chalk on the blackboard

Helge draws a circle with the chalk

The blue Helge turtle

> **Jens**: Helge, please take this blue piece of chalk
>
> *Helge takes the chalk from Jens*
>
> **Jens**: Helge, please move to the right blackboard
>
> *Helge moves to the right-most blackboard*
>
> **Jens**: Helge, draw a circle with the chalk on the blackboard
>
> *Helge draws a circle with the chalk*

Reflection

Take a minute to reflect what happened:

- What changed in the three scenarios?
- What was the functionality that Helge performed?

Translation to pseudo code

```
1. Give the turtle a color
2. Ask the turtle to move to a certain position
3. Ask the turtle to draw a circle
```

But, wait. How do we "*give something a color*"?!

Introducing objects

An object is a **thing** that has **properties** and that can **do** things

- Example property: Color, starting position
- Example functionality: Draw circle

Turtle objects

Turtles are actually objects we can work with. The code below creates a new `Turtle` object, and **stores** it in the `blue_turtle` variable:

```
from turtle import Turtle

blue_turtle = Turtle()
```

We can now ask that turtle to draw a circle (functionality):

```
blue_turtle.circle(100)
```

We can also ask it to change color:

```
blue_turtle.color('blue')
```

So, what happens if we ask it to draw a circle after changing the color?

```
blue_turtle.circle(100)
```

Note that the order is important! The colour change happens in between drawing circles. Your program will:

1. Draw a *black* circle
2. Draw a *blue* circle

You can do all sorts of things with the turtle. Try to set the shape of the turtle like so:

```
blue_turtle.shape('turtle')
```

On cats and properties

Let's say we have a cat. And that it is called Bob. Already here we distinguish between

- The type of animal called "cat"
- Bob, a specific cat

In Python this is known as:

- The **class** (the type of animal called "cat")
- The **object** (Bob, the specific cat)

Classes and objects

Other analogies for classes and objects are:

Class	Objects
Car	Ford, Saab, BMW
Person	Helge, Jana, Viktor, Morten, Anna
Phone	Nokia 3210, iPhone 10, Huawei Honor 6X

Notice that all of them have **properties** and **functionality**.

Back to Bob

Let's say that Cats have the property `color` . And that all cats are now `grey` .

To express this in Python we have to create the *general class* called **Cat**. And that *class* contains a **property** called `color` that is assigned to the string value `grey` .

In [1]:

```python
class Cat:
    color = 'grey'
```

We can now go from the *general class* to a concrete cat. That means we're creating a new object. To re-use it later we store it in a variable:

In [2]:

```python
a_cat = Cat()
```

Since all cats have the property `color` , we can now print the `color` of the specific cat stored in `a_cat` :

In [3]:

```python
print(a_cat.color)
```

```
grey
```

Adding names to our cats

Let's add names to our cats. We can do it by changing the `Cat` class like so:

In [4]:

```python
class Cat:
    color = 'grey'
    name = 'Bob'
```

In [5]:

```python
bob_the_cat = Cat()
print(bob_the_cat.name)
```

```
Bob
```

But wait a minute. Now all our cats are grey and are called bob! What if we have a cat called `Omen` ? Can we change the class definition?

In [6]:

```python
class Cat:
    color = 'grey'
    name = 'Omen'
```

In [7]:

```python
omen_the_cat = Cat()
print(omen_the_cat.name)
```

```
Omen
```

In [8]:

```python
bob_the_cat = Cat()
print(bob_the_cat.name)
```

```
Omen
```

Adding parameters to classes

What we need are parameters. We need to create the `Cat` objects with **specific** names and colours. We can do that when we **construct** the class.

Type the following into Mu and explain the output to your neighbour:

In [32]:

```python
class Cat:
    def __init__(self, color, name, paws=4):
        self.color = color
        self.name = name
        self.number_of_paws = paws
        self.paws = 4

bob_the_cat = Cat('grey', 'Bob')
omen_the_cat = Cat('black', 'Omen', 3)

print(omen_the_cat.number_of_paws)
omen_the_cat.beard = 'blond'
print(omen_the_cat.beard)
print(bob_the_cat.beard)
```

```
3
blond

---------------------------------------------------------------------
--------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-32-ec57499a7001> in <module>
     12 omen_the_cat.beard = 'blond'
     13 print(omen_the_cat.beard)
---> 14 print(bob_the_cat.beard)

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

Constructing objects

When we create an **object** from a **class** we say that we **instantiate** or **construct** the object.

When that happens the **init** function is called. And in there we can assign specific properties to our object. Like putting `'Bob'` into `self.name`.

Discuss with your neighbour:

- What is the difference between the `self` variable and the two cat objects `bob_the_cat` and `omen_the_cat`?
- Can you change the color of `bob_the_cat`?

The `self` object

The `self` object **is the object that you are currently constructing**. This makes sense because we need all cats to be different. Omen and Bob are two different cats, so we have to put that information somewhere that does **not** impact all other cats.

On cats and functionality

That was classes, objects and properties. Now we can talk about functionality. Let's say that all cats can `make_soud` . That is a **function** every cat can do. So it can be described in the general class of `Cat` :

In [33]:

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

    def make_sound(self):
        return 'Miaaouuuu'
```

In [34]:

```
bob_the_cat = Cat('grey', 'Bob')
print(bob_the_cat.make_sound())
```

```
Miaaouuuu
```

Cool, that worked! Notice that we still need the `self` argument. We need it because the *functionality* of the cat *may* depend on the `Cat` 's properties like `color` or `name` .

Here is an example:

In [35]:

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

    def make_sound(self):
        return 'The ' + self.color + ' cat miaous'
```

In [36]:

```python
bob_the_cat = Cat('grey', 'Bob')
print(bob_the_cat.make_sound())

omen_the_cat = Cat('black', 'Omen')
print(omen_the_cat.make_sound())
```

```
The grey cat miaous
The black cat miaous
```

Functionality == method

This is really cool because we didn't even need to tell the `make_sound` function that the cat is `grey` or `black`. It knew that already because the object already had the properties.

Because this is not a normal stand-alone function, we call functions inside a `Class` a **method**.

You now learned:

- Class
  - An abstract idea about, for instance, a `Cat` or a `Person`
- Object
  - A concrete implementation of a `class` like `Bob` or `Helge`
- Properties
  - Attributes of a certain object, like colour
- Methods
  - Functionality that an object can perform, like `make_sound` or "draw a circle" (turtle)

Exercise

- Add a method to your `Cat` class called `purr` that writes `'<name> the cat purrs'` where `<name>` is replaced by the name of the cat
- Create a third cat called `'Pinky'`. Give it an appropriate color.
- Talk to your neighbour: what happens when you ask `Pinky` to `make_sound` and `purr`? Why?

Object-oriented Programming

Object-oriented programming is one of the most common ways to write and structure software. In object-oriented programming you write classes that represent real-world things and situations, and you create objects based on these classes.

When you write a class, you define the general behavior that a whole category of objects can have.

When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire.

Making an object from a class is called *instantiation*, and you work with instances of a class.

On foxes and sounds

We just agreed that the cat could make a sound. And when it did, it would miau.

But a fox can make a sound as well! According to Norwegians it goes something like 'RIKA DING-DING-DING'.

In [37]:

```python
class Fox:
    def make_sound(self):
        return 'RIKA DING-DING-DING'
```

In [38]:

```python
a_fox = Fox()
print(a_fox.make_sound())
```

```
RIKA DING-DING-DING
```

These foxes are much simpler than our cats, because they have no properties. They simply make their weird sound.

But notice that the *name* of the method is the same: `make_sound` . And when you think about it, doesn't all animals make a sound?

On animals and sounds

We know that all animals can make a sound. So if you want to be an animal, you **need** a sound (also the fox!).

That means we have a general property for all animals: `make_sound` . In a way this is also a contract. You **promise** to have a sound when you are an animal.

But the exact sound depends on the animal.

```
Animal:                         Cow is-an Animal:             Cat is-an A
nimal:

   make_sound: <REQUIRED>          make_sound: Moooo             make_sou
nd: Miau
```

In [39]:

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

    def make_sound(self):
        return 'The ' + self.color + ' cat miaous'

class Fox:
    def make_sound(self):
        return 'RIKA DING-DING-DING'
```

In [40]:

```python
class Cow:
    def make_sound(self):
        return "Moooo"

print(Cow().make_sound())
```

```
Moooo
```

In Python we can express the fact that both `Cow` , `Fox` and `Cat` all are `Animal` s. Meaning, that they have the same behaviour as all animals. In our case this means that they can `make_sound` .

This is handy because everyone claiming to be an animal **promises** to behave like the animal.

Translated to Python this means that animals **promise** to have the same **properties** and **methods.**

In [42]:

```python
class Animal:
    def make_sound(self):
        return "The animal makes a sound"

print(Animal().make_sound())
```

```
The animal makes a sound
```

We can now update our cat to become an `Animal` . That will **automatically inherit** the `make_sound` function:

In [43]:

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

In [44]:

```python
bob_the_cat = Cat('grey', 'Bob')
print(bob_the_cat.make_sound())
```

```
The animal makes a sound
```

We can also **override** the method from `Animal` :

In [46]:

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

    def make_sound(self):
        return 'The ' + self.color + ' cat miaous'
```

In [47]:

```python
bob_the_cat = Cat('grey', 'Bob')
bob_the_cat.make_sound()
```

Out[47]:

```
'The grey cat miaous'
```

Exercise

- Write the `Animal` class into Mu

    ```python
    class Animal:
        def make_sound(self):
            return "The animal makes a sound"
    ```

- Let the `Fox` `Class` be an `Animal`, but without any properties or methods like so:

    ```python
    class Fox(Animal):
        pass
    ```

- Talk to your neighbour: what happens when you create a new `Fox` object and asks it to make a sound?
- Make the `Fox` say `'TCHJING-A-DING-DING'` by overwriting the `make_sound` method from `Animal`

Inheritance

Now, we have what we call class **inheritance**. The `Fox` and `Cat` classes **inherit** the properties and methods from `Animal`.

In this scenario `Animal` is a **superclass** and `Fox` and `Cat` are **subclasses**:

```
              Animal
                △
                |
       +--------+--------+
       |        |        |
      Cat      Fox      Cow
```

Checking *instance of*

Sometimes, during development, you would like to know of which class a certain variable is instance of. You can do so with the built-in method `isinstance()` as illustrated in the following.

In [48]:

```
isinstance(bob_the_cat, Cat)
```

Out[48]:

```
True
```

In [49]:

```
isinstance(bob_the_cat, Animal)
```

Out[49]:

```
True
```

Modifying an Attribute’s Value Through a Method

It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

In [ ]:

```
class Cat()
    def __init__(self):
        self.colour = 'grey'

    def set_colour(self, colour):
        self.colour = colour

boris_the_cat = Cat()
boris_the_cat.set_colour('black')
```

Another class example: books

You can model almost anything using classes. Let’s start by writing a simple class, `Book` , that represents a book and its contents—not one book in particular, but any book.

In [ ]:

```
class Book():
    ...
```

What do we know about most books? Well, they likely all have a *title* an *author*, and contain some *chapters*. We also know that you can *read* books and *open* them to inspect them further.

Those two pieces of information (title and author) and those two behaviors (read and open) will be constituents of our Book class because they are common to most books. After our class is written, we will use it to create individual instances, each of which represents one specific book.

Creating the Book Class

Each instance created from the Book class will store a `title` an `author` , and its `chapters` .

```
class Book():

    def __init__(self, title, author, chapters=[]):
        """Initialize title, the author, and the chapters."""
        self.title = title
        self.author = author
        self.chapters = chapters
```

The init() Method

A function that’s part of a class is a *method*. Everything you learned about functions applies to methods as well; the only practical difference for now is the way we will call methods.

The `__init__()` method is a special method Python runs automatically whenever we create a new instance based on the `Book` class. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python’s default method names from conflicting with your method names.

```python
class Book():

    def __init__(self, title, author, chapters=[]):
        """Initialize title, the author, and the chapters."""
        self.title = title
        self.author = author
        self.chapters = chapters
```

We define the `__init__()` method to have four parameters: `self`, `title`, `author`, and `chapters=`. The `self` parameter is **required** in the method definition, and it must come **first** before the other parameters. It must be included in the definition because when Python calls this `__init__()` method later (to create an instance of `Book`), the method call will automatically pass the `self` argument.

Every method call associated with a class automatically passes `self`, which is a reference to the instance itself; it gives the individual instance access to the attributes and methods in the class. When we make an instance of `Book`, Python will call the `__init__()` method from the `Book` class. We will pass `Book()` a `title`, an `author`, and `chapters=` as arguments; `self` is passed automatically, so we do not need to pass it. Whenever we want to make an instance from the `Book` class, we will provide values for only the last three parameters.

The three variables defined each have the prefix `self`. Any variable prefixed with `self` is available to every method in the class, and we will also be able to access these variables through any instance created from the class. `self.title = title` takes the value stored in the parameter name and stores it in the variable name, which is then attached to the instance being created. Variables that are accessible through instances like this are called attributes.

The `Book` class will have two other methods: `read` and `open`. They will give each book the 'ability' to get `read()` and to get opened `open_book()`.

In [ ]:

```python
class Book():
    """A simple book model consisting of chapters, which in
    turn consist of paragraphs."""

    def __init__(self, title, author, chapters=[]):
        """Initialize title, the author, and the chapters."""
        self.title = title
        self.author = author
        self.chapters = chapters

    def read(self, chapter=1):
        """Simulate reading a chapter, by calling the reading
        method of a chapter."""
        self.chapters[chapter - 1].read()

    def open_book(self, chapter=1):
        """Simulate opening a book, which returns a chapter
        object."""
        return self.chapters[chapter - 1]
```

Why object-oriented programming

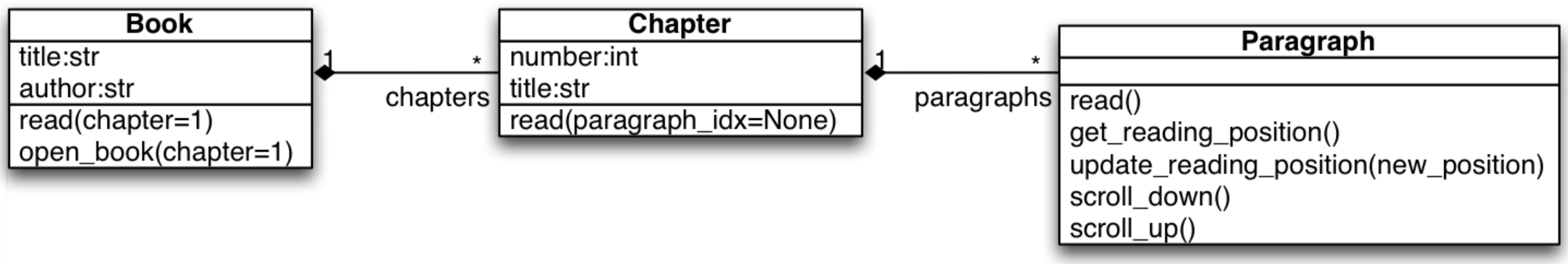
Historically: we have data and functionality. Why not merge them?

Cognitive: we 'group' things into objects in the real world. Like cats, persons, vehicles, etc. Why not do the same with programs.

Illustration: it is easy for us to associate one color with Bob and another color with Omen.

This is called *encapsulation*. We encapsulate things into one object. Changing them doesn't affect other objects. Next week: Excel spreadsheets.

In today's session we will create a model for books, which can contain chapters, which in turn can contain paragraphs.



Making an Instance from a Class

Think of a class as a set of instructions for how to make an instance. The class `Book` is a set of instructions that tells Python how to make individual instances representing specific books.

We tell Python to create a book whose title is `'The Empty Book'` and whose author is `'Helge'`. When Python reads the line, `empty_book = Book('The Empty Book', 'Helge')` it calls the `__init__()` method in `Book` with the arguments `'The Empty Book'` and `'Helge'`. The `__init__()` method creates an instance representing this particular book and sets the `author` and `title` attributes using the values we provided. The `__init__()` method has no explicit return statement, but Python automatically returns an instance representing this book. We store that instance in the variable `empty_book`. The *naming convention* is helpful here: we can usually assume that a capitalized name like `Book` refers to a class, and a lowercase name like `empty_book` refers to a single instance created from a class.

Accessing Attributes

To access the attributes of an instance, you use 'dot' notation. We access the value of `empty_book`'s attribute `author` by writing: `empty_book.author`.

In [ ]:

```python
empty_book = Book('The Empty Book', 'Helge')

print(empty_book.author)
print(empty_book.title)
print(len(empty_book.chapters))
```

In [ ]:

```python
class Chapter():

    def __init__(self, number, title, paragraphs):
        """A chapter consists of multiple paragraphs."""
        self.number = number
        self.title = title
        self.paragraphs = []
        for paragraph_lines in paragraphs:
            new_pragraph = Paragraph(paragraph_lines)
            self.paragraphs.append(new_pragraph)

    def read(self, paragraph_idx=None):
        """A paragraph can be read."""
        if paragraph_idx != None:
            self.paragraphs[paragraph_idx].read_all()
        else:
            for paragraph in self.paragraphs:
                paragraph.read_all()
```

In [ ]:

```python
class Paragraph():
    """A paragraph consists of a list of lines."""

    def __init__(self, lines):
        """Initialize the paragraph with its lines of text."""
        self.lines = lines

    def read_all(self):
        """Simulate reading a paragraph by printing its contents."""
        for line in self.lines:
            print(line)
```

Now, we get some text from a real book to build an instance of it using the above clases.

In [ ]:

```python
# open it and read all lines in the text file
with open('bones_in_london.txt') as f:
    content = f.readlines()


def get_text(lower_bound, upper_bound):
    """A utility function which allow us to read slices of lines"""
    chapter_content = []
    paragraph = []
    for line in content[lower_bound:upper_bound]:
        if line == '\n':
            chapter_content.append(paragraph)
            paragraph = []
        else:
            paragraph.append(line.strip())

    return chapter_content
```

Now, that we can read some real text from a file, we can create instances of chapters and an instance of a book. For simplicity, our book consists only of the first two chapters.

In [ ]:

```python
chapter_1 = Chapter(1, 'Bones and Big Business', get_text(82, 762))
chapter_2 = Chapter(2, 'Hidden Treasure', get_text(769, 1455))
book = Book('Bones in London', 'Edgar Wallace', [chapter_1, chapter_2])
```

In [ ]:

```python
print(book.author)
print(book.title)
print(len(book.chapters))
```

Calling Methods

After we create an instance from the class `Book` , we can use 'dot' notation to call any method defined in `Book` . To call a method, give the name of the instance (in this case, `book`) and the method you want to call, separated by a dot. When Python reads `book.read(chapter=1)` , it looks for the method `read(chapter)` in the class `Book` and runs that code.

In [ ]:

```
book.read(chapter=1)
```

In [ ]:

```
book.open_book(chapter=2).read(paragraph_idx=3)
```

In [ ]:

```
book.open_book(chapter=2).read(3)
```

Creating Multiple Instances

You can create as many instances from a class as you need. Even if we used the same `title` , `author` , and `chapters` for the second book, Python would still create a separate instance from the `Book` class. You can make as many instances from one class as you need, as long as you give each instance a unique variable name or it occupies a unique spot in a list or dictionary.

In [ ]:

```
chapter_1 = Chapter(1, 'Bones and Big Business', get_text(82, 762))
chapter_2 = Chapter(2, 'Hidden Treassure', get_text(769, 1455))
book = Book('Bones in London', 'Edgar Wallace', [chapter_1, chapter_2])

empty_book = Book('The Empty Book', 'Helge')

print(book)
print(empty_book)
```

Modifying Attribute Values

You can change an attribute’s value in three ways:

- you can change the value directly through an instance,
- set the value through a method,
- or increment the value (add a certain amount to it) through a method.

Let’s look at each of these approaches.

Modifying an Attribute’s Value Directly

The simplest way to modify the value of an attribute is to access the attribute directly through an instance. Here we set the `reading_position` to `3` directly.

In [ ]:

```
chapter_1.title = chapter_1.title + '!'
print(chapter_1.title)
```

Inheritance

You do not always have to start from scratch when writing a class. If the class you are writing is a specialized version of another -readily available- class, you can use inheritance to implement your extensions.

When one class inherits from another, it automatically takes on all the attributes and methods of the first class. The original class is called the *parent class*, and the new class is the *child class*. The child class inherits every attribute and method from its parent class but is also free to define new attributes and methods of its own.

The init() Method for a Child Class

The first task Python has when creating an instance from a child class is to assign values to all attributes in the parent class. To do this, the `__init__()` method for a child class needs help from its parent class.

As an example, let's model a comic book. A comic book is just a specific kind of book, with less text and more images :). Consequently, we can base our new `ComicBook` class on the `Book` class we wrote earlier. Then we'll only have to write code for the attributes and behavior specific to comic books.

When you create a child class, the parent class **must be part of the current file and must appear before the child class in the file**. The name of the parent class must be included in parentheses in the definition of the child class. The `__init__()` method takes the information required to make a `ComicBook` instance.

The `super()` function is a special function that helps Python make connections between the parent and child class. It tells Python to call the `__init__()` method from `ComicBook` 's parent class, which gives a `ComicBook` instance all the attributes of its parent class. The name super comes from a convention of calling the parent class a **superclass** and the child class a **subclass**.

In [ ]:

```python
class Book():
    """A simple book model consisting of chapters, which in
    turn consist of paragraphs."""

    def __init__(self, title, author, chapters=[]):
        """Initialize title, the author, and the chapters."""
        self.title = title
        self.author = author
        self.chapters = chapters

    def read(self, chapter=1):
        """Simulate reading a chapter, by calling the reading
        method of a chapter."""
        self.chapters[chapter - 1].read()

    def open_book(self, chapter=1):
        """Simulate opening a book, which returns a chapter
        object."""
        return self.chapters[chapter - 1]


class ComicBook(Book):
    def __init__(self, title, author, chapters=[]):
        """Initialize attributes of the parent class."""
        super().__init__(title, author, chapters=chapters)
```

In [ ]:

```python
comic = ComicBook('Le Grand Mort', 'Loisel')
comic.title
```

Defining Attributes and Methods for the Child Class

Once you have a child class that inherits from a parent class, you can add any new attributes and methods necessary to differentiate the child class from the parent class.

Let's add an attribute that is specific to comic books, such as, images, and a method to report on this attribute.

In [ ]:

```python
class ComicBook(Book):
    def __init__(self, title, author, chapters=[]):
        """Initialize attributes of the parent class."""
        super().__init__(title, author, chapters=chapters)
        self.images = []

    def get_images(self):
        return self.images
```

There is no limit to how much you can specialize the `ComicBook` class. You can add as many attributes and methods as you need to model acomic book to whatever degree of accuracy you need. An attribute or method that could belong to any book, rather than one that is specific to a comic book, should be added to the `Book` class instead of the `ComicBook` class. Then anyone who uses the `Book` class will have that functionality available as well, and the `ComicBook` class will only contain code for the information and behavior specific to comic books.

Overriding Methods from the Parent Class

You can override any method from the parent class that does not fit what you are trying to model with the child class. To do this, you define a method in the child class with **the same name as the method you want to override in the parent class**. Python will disregard the parent class method and only pay attention to the method you define in the child class.

In [ ]:

```
class Animal:
    def sound(self):
        raise Exception("What kind of animal are you, a fox?!")

class Fox(Animal):
    def sound(self):
        print("YOUFYOUFIFIFIF")
```

In [ ]:

```
red_fox = Fox()
red_fox.sound()
```

Importing Classes

As you add more functionality to your classes, your files can get long, even when you use inheritance properly. In keeping with the overall philosophy of Python, you will want to keep your files as uncluttered as possible. To help, Python lets you store classes in modules and then import the classes you need into your main program.

You can store as many classes as you need in a single module, although each class in a module should be related somehow.

Importing a Single Class or Multiple Classes

We include a module-level docstring that briefly describes the contents of this module. You should write a docstring for each module you create.

Now we make a separate file called `book.py` . From this file will import the `Book` class and then create an instance from that class.

You can import as many classes as you need into a program file. If we want to make a regular car and an electric car in the same file, we need to import both classes, Car and ElectricCar

In [ ]:

```python
from book import Book
```

In [ ]:

```python
book?
```

In [ ]:

```python
Book?
```

In [ ]:

```python
from book import Book, Chapter, Paragraph

chapter_1 = Chapter(1, 'Bones and Big Business', get_text(82, 762))
chapter_2 = Chapter(2, 'Hidden Treassure', get_text(769, 1455))
book = Book('Bones in London', 'Edgar Wallace', [chapter_1, chapter_2])

empty_book = Book('The Empty Book', 'Helge')

empty_book
```

- Have a look on https://michael0x2a.com/blog/turtle-examples (https://michael0x2a.com/blog/turtle-examples) to see some examples of how to draw turtle graphics with Python.