COM6115: Text Processing

Python Introductory Materials

Object Oriented Programming: Introduction, Defining Classes in Python, Inheritance

Mark Hepple

Department of Computer Science
University of Sheffield

Let's talk about meaning

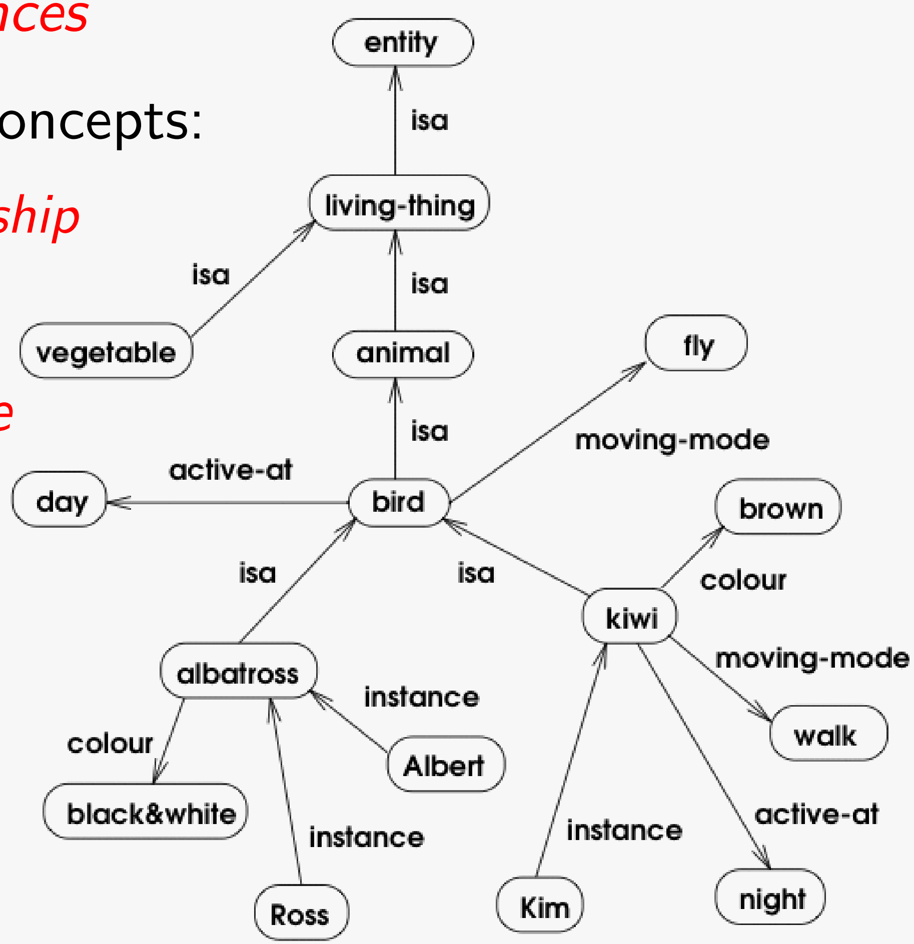
The picture shows a (simplified) example of a *semantic network*.

This is a *knowledge representation* used in AI, which has:

- nodes: represent *concepts* or *instances*
- links: represent relations between concepts:
  - ◇ instance: indicates class *membership*
  - ◇ isa: indicates class *inclusion*
- nodes *inherit* properties from *above*

  e.g. know albatrosses fly,
  because we know birds do!

  - ◇ but can also encode *exceptions* (Kiwis don't fly!)
  - ◇ gives *economy* of representation i.e. avoids *redundancy*



Let's talk about meaning (ctd)

- Semantic networks also used in:
  - Psychology: esp. models of human knowledge / memory
  - Linguistics: models of language meaning
- Related ideas explored in Philosophy
  - esp. under headings of Taxonomy and Ontology
- The *Semantic Web* relies critically on *ontologies*
  - ontologies: a form of knowledge representation (related to SNs)
  - used to associate *semantics* (meaning) with web content
  - so machines can address this in a more *intelligent* fashion
- Why the hell is he telling he telling us this stuff??
  - this is supposed to be a programming course, isn't it?!
- . . . *because* related ideas adopted in work on programming
  - giving approach known as *Object Oriented Programming*
  - now the *dominant paradigm*; includes e.g. `Java`, `C++`

Let's talk about meaning (ctd)

- To reiterate the main points ...
- Key notion: *CONCEPT*
  - ◇ general idea of a *class of things* with particular properties in common

  e.g. *person, bird, animal, vehicle, chair, gun, etc.*
- A concept has *INSTANCES*
  - ◇ actual occurrences in the world

  e.g. concept *person* has *instances* such as: *Me! You! Obama!*
- For a given concept, expect certain *attributes*

  e.g. for *person*, expect: *age, gender, height, etc.*
  - ◇ a specific *actual* person will *instantiate* these attributes

    i.e. provide specific *values* for them
- Concept may also have associated *expected behaviours*

  e.g. for *person* — *walk, talk, read, hoover, give birth*

  e.g. for *bird* — *fly, lay eggs*

Object Oriented Programming

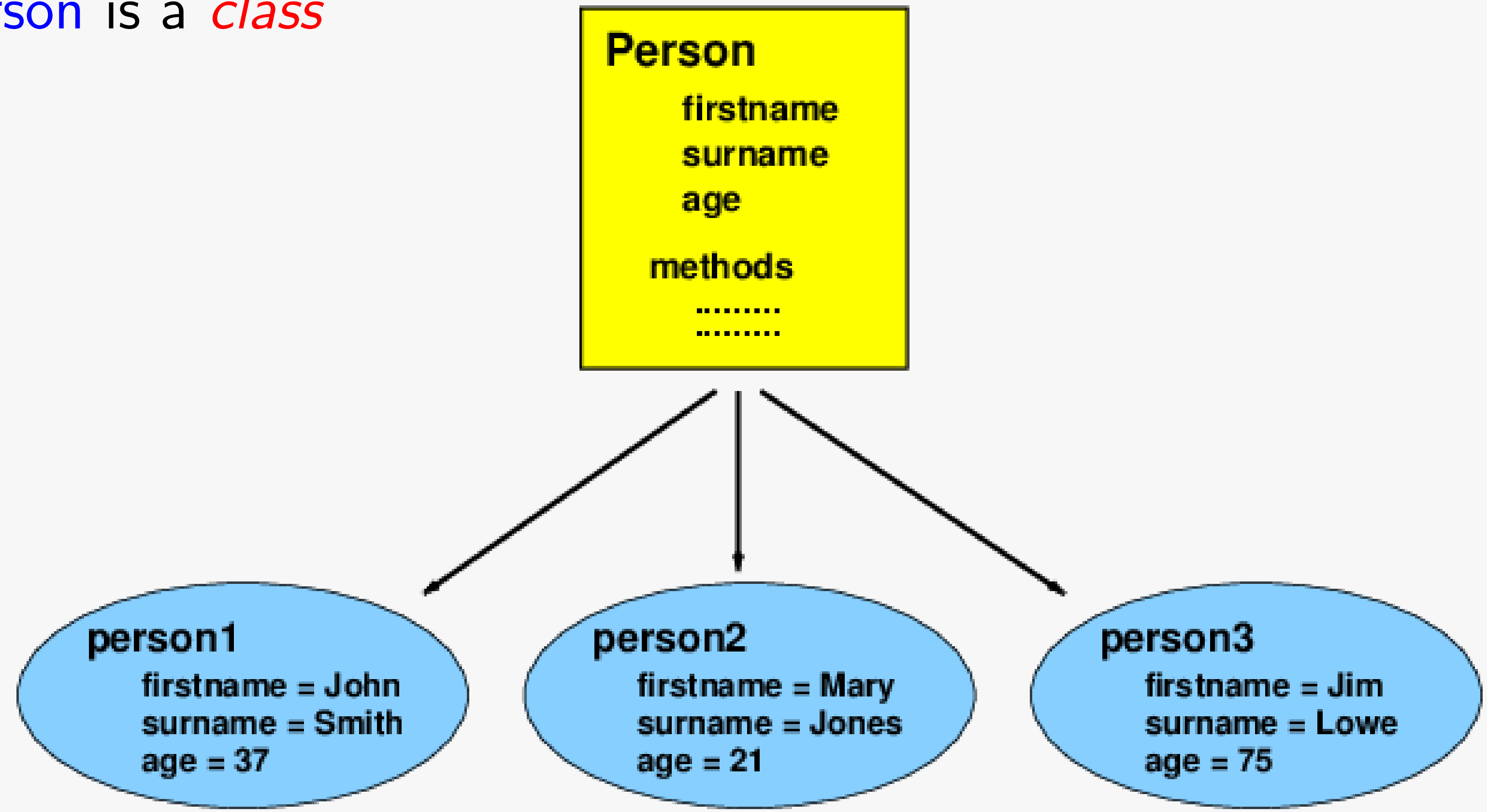
- A *programming paradigm* is a particular *approach to* or *'style' of* programming
- So far, we have used a *procedural programming* paradigm
  - focus is on writing *functions* or *procedures* to operate on data
- Alternative paradigm: *object oriented programming*
  - over last 20 years, has become the *dominant* programming paradigm
  - developed to make it easier to create and/or modify large, complex software systems
- In Object Oriented Programming (OOP):
  - focus is on creating *objects*
  - objects contain both *data and functionality*

Objects and Classes — *an example*

- A Person class might:
  - have *attributes* (variables) for:
    - name, age, height, address, tel.no., job, etc
  - have *methods* (functions) to:
    - update address
    - update job status
    - work out if they are adult or child
    - work out if they pay full fare on the bus
    - *etc.*
- There might be many *objects* of the Person class
  - each representing a *different person*
    - *with different specific data*
  - but all store *similar information* and *behave similarly*

Objects and Classes — *an example*

- Person is a *class*



◇ person2, person2 & person3 are *objects*

Defining Classes in Python

- Definition opens with keyword class + class name
- Class needs an *initialisation* method
  - called when an instance is created
  - has 'special' name: `__init__`
  - establishes the *attributes* (i.e. vars) belonging to objects

```
class Person:
    def __init__(self):
        self.firstname = None
        self.surname = None
        self.age = None
        self.species = 'homo sapiens'
```

  - note use of *special variable* `self` here
  - it is the instance's way of *referring to itself*

    e.g. `self.species` above means "the `species` attribute of *this instance*"

Defining Classes in Python (ctd)

- `Person` class with its *initialisation* method, again:

```
class Person:
    def __init__(self):
        self.firstname = None
        self.surname = None
        self.age = None
        self.species = 'homo sapiens'
```

- Can create an object (i.e. *instance*) of this class as follows:

```
>>> p1 = Person()
>>> p1.species
'homo sapiens'
```

  ◇ here, call to `Person()` creates a new instance of the Person class
    - the `__init__` method is called automatically, to initialise the object
    - the object is assigned to `p1`

Defining Classes in Python (ctd)

- Last example again:

```
>>> p1 = Person()
>>> p1.species
'homo sapiens'
```

  - ◇ statement `p1.species` accesses `p1`'s species *attribute* directly

    i.e. that value is accessed in the e.g. above, and printed by the interpreter

- Can think of a objects as being like "*bundles of data*"
  - ◇ each object is a different *bundle of data*, storing info about a *different instance of the class*
  - ◇ Note extra `self` arg in:

```
class Person:
    def __init__(self):
        self.firstname = None
            ...
```

    - is object's way of talking about *itself*, i.e. *the bundle that I am*
    - info stored with a `self.` attribute becomes *part of the bundle*

      — is carried around with it, and is *always available*

Defining Classes in Python (ctd)

- More generally, initialisation method can have *parameters*
  - ◇ can be used to set *initial values of attributes*

```
def __init__(self, firstname, surname, age):
    self.firstname = firstname
    self.surname = surname
    self.age = age
    self.species = 'homo sapiens'
```

  - ◇ example of creating an instance:

```
>>> p1 = Person('John', 'Smith', 37)
>>> p1.firstname
'John'
>>> p1.age
37
```

  - ◇ note `__init__` has 4 args, but 3 given when object created – *Why?*
    - first `self` is left *implicit* – stands for *this object* (i.e. *bundle of data*)
    - that object stored as `p1`, can access bundle data directly, e.g. `p1.age`

Defining Classes — *adding functionality*

- Can define (more) functions — in OOP, are known as *methods*

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

    def greeting_informal(self):
        print('Hi', self.firstname)

    def greeting_formal(self):
        print('Welcome, Citizen', self.surname)
```

  - ◇ as before, self appears as 1st arg of every method
    - shows that this is an object method, i.e. will be *called from an object*
  - ◇ self again refers to *this instance*, allowing access to *its own data*
    - thus, `self.firstname` above *means* value of *my `firstname` attribute*
    - that value, stored with this bundle of data, is accessed and used

Defining Classes — *adding functionality* (ctd)

- Example: here create two instances:

```
>>> p1 = Person('Harry', 'Potter', 12)
>>> p2 = Person('Hermione', 'Grainger', 12)
```

- Call newly defined methods from instances:

```
>>> p1.greeting_informal()
Hi Harry
>>> p1.greeting_formal()
Welcome, Citizen Potter
>>> p2.greeting_formal()
Welcome, Citizen Grainger
```

  - ◇ note that 1st `self` arg from definition again absent – *i.e. is left implicit*
  - ◇ when `p1.greeting_informal()` is called, `p1` stores an instance, and `self` aspects of definition are *about that instance*
  - ◇ thus, method calls access data (e.g. `surname`) from given instance (`p1` or `p2`), and output depends on that

Defining Classes — *adding functionality* (ctd)

- Another method ...

```
class Person:
        ...

    def greeting_age_based(self):
        if self.age < 18:
            print('Welcome, Young', self.firstname)
        elif self.age > 60:
            print('Welcome - oh Venerable', self.firstname)
        else:
            self.greeting_formal()
```

- ◇ here see behaviour that *uses* instance data (`firstname`) and that *is conditioned on* instance data (age)
- ◇ note: 'else' case *calls* another method of the instance
  - does so in form: `self.greeting_formal()`
  - uses `self`, as it is *this object's* method being used
  - but `self` is *prefixed*, not supplied as arg

Defining Classes — *adding functionality* (ctd)

- Example
  - ◇ First create three instances:

```
>>> p1 = Person('Harry', 'Potter', 12)
>>> p2 = Person('Sirius', 'Black', 38)
>>> p3 = Person('Minerva', 'McGonagall', 66)
>>>
```

  - ◇ Call method – behaviour is conditioned on the person's age:

```
>>> p1.greeting_age_based()
Welcome, Young Harry
>>> p2.greeting_age_based()
Welcome, Citizen Black
>>> p3.greeting_age_based()
Welcome - oh Venerable Minerva
>>>
```

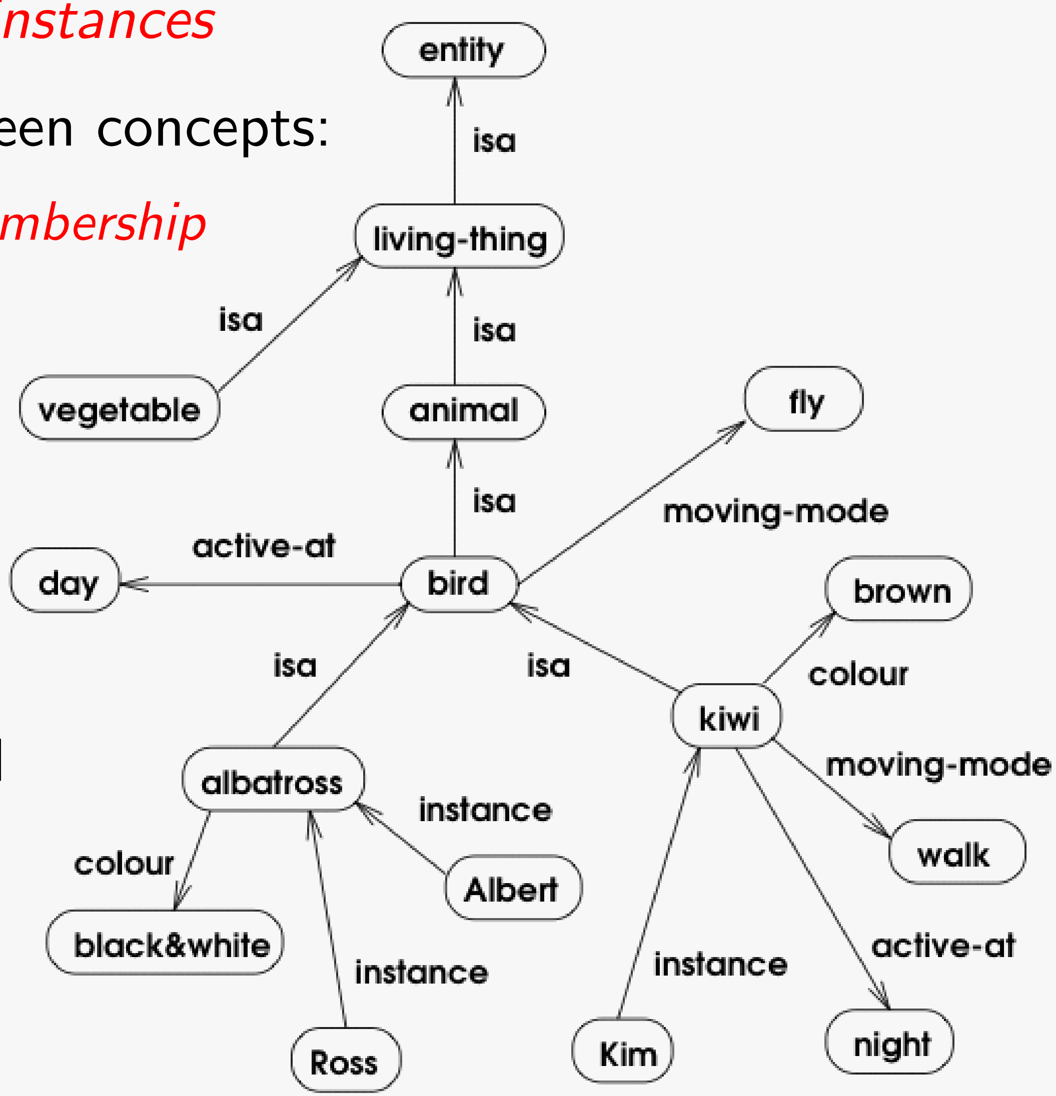
Concepts and Inheritance — *revisited*

Picture shows an example *semantic network*, in which:

- nodes: represent *concepts* or *instances*
- links: represent relations between concepts:
  - ◇ instance: indicates class *membership*
  - ◇ isa: indicates class *inclusion*
- nodes *inherit* properties from *above*

  e.g. know albatrosses fly, because birds do!
- but can also *override* inherited properties, specify *exceptions*

  e.g. Kiwis don't fly!



Defining Classes with Inheritance

- Recall the `Person` class:

```
class Person:
    def __init__(self, firstname, surname, age):
        self.firstname = firstname
        self.surname = surname
        self.age = age
        self.species = 'homo sapiens'

    def greeting_informal(self):
        print('Hi', self.firstname)

    def greeting_formal(self):
        print('Welcome, Citizen', self.surname)

    def greeting_age_based(self):
            ...
            ...
```

Defining Classes with Inheritance (ctd)

- Consider that we might want to define a class `Wizard`
  - ◇ `Wizard`s have *much in common with* `Person`s, e.g.:
    - have *firstnames, surnames, ages, etc* that must be stored
    - also *exchange greetings*, etc.
  - ◇ But also *differ* from ordinary Persons (muggles) *in various ways*
    - can cast *spells*, etc.
- Could simply define `Wizard` class from scratch

  e.g. specifying *firstname, surname, age, etc attributes*

  e.g. methods for *greetings*, etc
  - ◇ BUT such code will be *highly redundant* with that for `Person` class
- ALTERNATIVE: use *inheritance*
  - ◇ `Wizard` class inherits from `Person` class – provides *basic set up*
  - ◇ this base is then *modified*, e.g. with *new attributes*, *new methods*

Defining Classes with Inheritance (ctd)

- Following `Wizard` class *inherits* from the Person class

```
class Wizard(Person):
    def __init__(self, firstname, surname, age):
        self.firstname = firstname
        self.surname = surname
        self.age = age
        self.species = 'homo magicus'
```

  - starts essentially as a *copy* of the Person class
    - say `Wizard` *inherits from* the Person class
  - can then *overwrite* methods of Person class, to modify behaviour
    - above code defines the `__init__` method
    - this definition *overwrites* definition from the Person class
  - can add new methods, to add new behaviour
- In above example:
  - `Person` is the *more general* class — it is the *superclass*
  - `Wizard` is the *more specific* class — the *subclass*

Defining Classes with Inheritance (ctd)

- Wizard class, again — *inherits* from Person class

```
class Wizard(Person):
          ...

    def greeting_formal(self):
        print('Welcome, Magician', self.surname)
```

- Above code overrides greeting_formal method of Person class:

  e.g. compare:

```
>>> p1 = Person('Harry', 'Potter', 12)
>>> p1.greeting_formal()
Welcome, Citizen Potter
```

  with:

```
>>> p1 = Wizard('Harry', 'Potter', 12)
>>> p1.greeting_formal()
Welcome, Magician Potter
```

Defining Classes with Inheritance — *initialisation*

- By default, *inherit* `__init__` method of superclass
  - so this works exactly as before
- Alternatively, can choose to *redefine* it, as we did above, i.e. in:

```
class Wizard(Person):
    def __init__(self, firstname, surname, age):
        self.firstname = firstname
        self.surname = surname
        self.age = age
        self.species = 'homo magicus'
```

  - new definition *overwrites* old definition, in usual way
  - BUT some of work done by this `__init__` method is the *same* as work done by the superclass `__init__` method

    i.e. there is some *redundancy*

Defining Classes with Inheritance — *initialisation* (ctd)

- Some *redundancy* between `__init__` methods of the Wizard and Person classes in preceding example:

```
class Person:
    def __init__(self, firstname, surname, age):
        self.firstname = firstname
        self.surname = surname
        self.age = age
        self.species = 'homo sapiens'
```

```
class Wizard(Person):
    def __init__(self, firstname, surname, age):
        self.firstname = None
        self.surname = None
        self.age = None
        self.species = 'homo magicus'
```

Defining Classes with Inheritance — *initialisation* (ctd)

- It may be that we want to *change only a part* of the initialisation behaviour of the superclass
  - ◇ in that case, can invoke superclass `__init__` method *directly*
    - method performs its usual initialisation of current data bundle
  - ◇ then have additional commands to modify the initialisation
- Example: new `__init__` method definition for Wizard class:

```
class Wizard(Person):
    def __init__(self, firstname, surname, age):
        Person.__init__(self, firstname, surname, age)
        self.species = 'homo magicus'
```

  - ◇ definition invokes the superclass (Person) `__init__` method with call:

    `Person.__init__(self, firstname, surname, age)`

  - ◇ note how `self` special var is explicitly passed as first argument

Defining Classes with Inheritance — *adding extra methods*

- Can also add completely *new methods*
  - to add functional behaviour that doesn't exist for superclass

```
class Wizard(Person):
                ...

    def stun(self):
        print('Expelliarmus!')
```

  - here, instances of `Wizard` class have a `stun` method
  - instances of `Person` class have no such method

Defining Classes — *FURTHER* inheritance

- Can use inheritance *again*, to build further classes *on top of* the `Wizard` class, e.g.

```
class HogwartsTeacher(Wizard):
    def __init__(self, firstname, surname, age, subject):
        Wizard.__init__(self, firstname, surname, age)
        self.subject = subject

    def greeting_formal(self):
        print('Welcome, Professor', self.surname)
```

- Some example behaviour:

```
>>> p2 = HogwartsTeacher('Severus', 'Snape', 38,  'potions')
>>> p2.subject
'potions'
>>> p2.greeting_formal()
Welcome, Professor Snape
```

What problem does OO Programming solve?

- In simpler '*imperative*' (non-OO) programming:
  - program may be one "long" list of commands
  - might group smaller sections of statements into functions / subroutines
  - common for data to be '*global*'

    i.e. accessible from any part of the program
    - hence, any statement/function might modify any piece of data
- Found to be very difficult to build large programs
  - approach allows *bugs* to have *far-reaching consequences*
  - may be very hard to track down
- Example: large program might require several programmers
  - work on different parts of code
  - one programmer's code makes change to data
  - second programmer does not anticipate this
    - second programmer's code now crashes *or*
    - appears to work but produces incorrect results

What problem does OO Programming solve? (ctd)

- Solution to problem through *modularity*:
  - break code down into suitably-sized chunks: modules
  - limit interactions between different components
- Object-Oriented Programming is most successful approach to achieving modularity
  - modules realised as Classes
  - group functionality / data-handling together
  - external code should not interfere with 'inner workings' of class
  - rather, should only access/modify data stored by class via *methods* that the class provides
  - these methods together constitute an *interface* for external code to use the class
  - in theory, I should be able to change inner workings of my class, without affecting external code that uses my class

    i.e. provided interface stays the same, and code delivers same functionality

Inheritance and Real-world Programming Problems

- In lab session, will see interesting case where *inheritance* is useful, i.e.
  - superclass of `MovingShapes`
  - specific shapes (e.g. `Triangle`, `Square`, etc), that inherit from it
    - but also have *own special behaviour*
  - that's a very specialised, abstract example
  - but is inheritance needed for real-world programming tasks?
- *EXAMPLE*:
  - Company wants to keep records of its employees (past and present)
  - BUT, company has various kinds of employee, e.g.
    - employee (standard, full-time)
    - employee (standard, part-time)
    - employee (executive)
    - employee (retired)
    - employee (zero-hours contract worker)

Inheritance and Real-world Programming Problems (ctd)

- *EXAMPLE (continued) . . .*
  - ◇ Records of different employee types will have much *in common* — both for *information stored*, and *actions required*

    e.g. name, D.O.B., address, etc

    e.g. function to update address, etc
  - ◇ But records of different employee types will also have *differences* — again, w.r.t. both *information* and *actions*

    e.g. executive may need additional fields for "type of company car", "expenses quota", etc

    e.g. different National Insurance contributions calculations may be needed, say, for "zero-hours" vs. "full-time" employees
  - ◇ Hence, want:
    - superclass for `Employee_Record`, to capture the commonality
    - subclasses for different employee types — so different characteristics can be handled, without reduncancy

Organising your Code via OOP

- Your programming task may not be complicated enough to need inheritance
- Even so, OO Classes are an excellent way to organise your code
- Without OOP, may have large collection of functions
  - must always check what is the right data to provide as input
  - must collect and store results of function calls, etc
- Often, much tidier to collect together as a Class
  - *initialisation* method can define (and initialise) all attributes needed to store data to be handled
  - then just provide as many methods as needed, to operate on data, with results often being stored back into object
  - conceptually, a much cleaner way to work