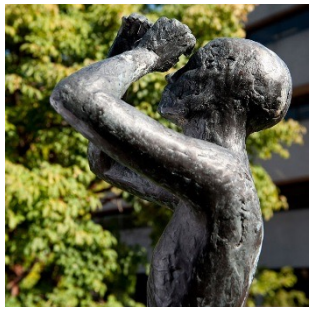

Lecture 7 – Inheritance

CS2513

Cathal Hoare

A TRADITION OF INDEPENDENT THINKING

UCC

University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Lecture Contents

Inheritance

Inheritance – How we really use Classes

- In an advanced program, many classes would be combined in various ways to produce a computer program.
- Rather than write one monolithic block of code that was tightly bound (can't easily reuse part of the code), we write numerous classes that encapsulate a piece of functionality.
- These classes can be reused

Inheritance – How we really use Classes

- Take the development of a Graphical User Interface...
  - classes that implement buttons and text boxes and all of the other components in a GUI can be reused time and time again:
    - We don’t write all of the code that we need for a button each time.
    - We specialise the button by setting its state or perhaps adding behaviour, but the bulk of the code is reused time and time again.

Inheritance – How we really use Classes

- By modularising our code, we:
  - Allow parallel development within a team of software engineers.
  - Simplify the code - rather than one large complex block, we have numerous simpler classes. We say the code is less tightly bound. Because each class is relatively simple, it is more likely to be bug free.
  - If a bug exists, we fix it in the responsible class. This class is imported into many
  - programs, so the fix propagates to where the code is used. If all of the code for the button is in one class, and there is a bug, we fix it once and it propagates to every instance of the button we create.
  - We can also replace code with better versions; if we develop a better button class, as long as the class name and method headers are the same, we can replace the class and not break any code that uses the class.
    - If we implemented the button from scratch every time, we would have had to replace and test the code in numerous places.

Mechanisms for Reuse

- Inheritance - captures an ‘is-a’ relationship between classes. It allows us to take an existing class and specialise and extend it.
  - We don’t have to rewrite the code of the existing class, just add any new or specific state and behaviour that the class requires.
- Composition - captures a ‘has-a’ relationship between classes.
  - We can divide a class into sub-classes where the sub-class might be reused in a different implementation later.

Implementing Inheritance

- Suppose we need to implement a payroll system for a company to manage its employees.
- Everyone has a name, a social security number and salary.
- Everyone can receive a pay rise (inherited for everybody), but this is calculated differently for managers (specialised for managers).
- Engineers have a special skill, but mangers don't (besides management).
- However, managers are in charge of a project, while engineers are not.

Implementing Inheritance

- We could write two classes engineer and manager, and have name, ssn and salary instance variables in both classes. We could also have getters and setters for these.
- This might be manageable where there are just two employee types. But now we hear the system is being extended to include secretaries, craftsmen, company officers, and many more types of employee.
- Do we rewrite that same code for each employee type? And if something changes in the future, rewrite that code numerous times?
- Bad Plan! We use inheritance instead.

Implementing Inheritance

- We would use inheritance to reduce the amount of code that we must write.
  - We would analyse the requirements to see what is in common between employee types (name, ssn and salary and would write a class to represent an employee)
  - Next, we would write two other classes, Manager and Engineer. Each ‘is-a’ type of employee. These would inherit the functionality of the Employee class.
    - As far as anyone using our Manager or Engineer class, it would seem that they have all of the state and behaviour of an Employee.
    - In reality this code is not part of their class, but rather is written once in the Employee
    - If we need to change any part of the Employee class, the next time we run the programme, it will appear that the change was made to both Engineer and Manager as well.
  - We would add only the state and behaviour unique to the Manager or Employee to these classes.

Language of Inheritance

- In this case, Manager and Engineer inherit from Person
  - Person is the parent of Engineer and Manager
  - Engineer and Manager are children of Person
  - Manager and Engineer are subclasses of Person
  - Person is the superclass of Manager and Engineer

Indicating Inheritance in Code

```
from person import Person       #Need to indicate make the Person code
                                 #available

class Manager(Person):          #Indicate parent class between brackets

        def __init__(self, identity, name, job, salary, project):
                super().__init__(identity, name, job, salary).    #1
                self._project = project                           #2

                #1 – here we call the constructor for the parent class
                #2 – After we've initialised the parent we add and initialise
                #      extra attributes for our class
```

Using Inheritance

- First we must specify the inheritance – what class is inheriting from which.
- Where we want to add sub-class specific functionality we must decide whether to extend or specialise.
  - Extend – that is add new state and behaviour. For example, our manager class must describe the project that the manager manages. This is not represented at all in the Person class so we must add it to the Manager.
  - Specialise – that is where functionality exists in the super class, add a version of that functionality that is specific to the sub-class. For example, givePayRaise() is defined in Person, but we need a specific version for manager.

Using Inheritance - Extend

- Extending is the same as adding any method or attribute to a class. In this case we add a protected project attribute and add getter/setter property to access it

```
def getProject(self):
        return self._project

def setProject(self, project):
        self._project = project

project = property(getProject, setProject)
```

Using Inheritance - specialisation

- Specialisation requires some consideration (see handling of givePayRaise() in our Manager class):
  - When we call a method against a manager instance, the Python interpreter will check if the manager class has a method of that name.
  - If it doesn't, it will look in the superclass for the method.
  - In fact it will go from object to object in the inheritance hierarchy until it finds a version to execute.

Using Inheritance - specialisation

- When we use specialisation, we should use the superclass' functionality where we can.
- This promotes maintainability in our code
  - For example, if a bug occurs in givePayRaise() the fewer places where code manages this functionality the better

Bad – replicating code

- When we use specialisation, we should use the superclass' functionality where we can.
- This promotes maintainability in our code
  - For example, if a bug occurs in givePayRaise() the fewer places where code manages this functionality the better
  - In the example below we now have two places where the calculation occurs – two changes, two fixes, etc.

```
def getPayRaise(self, percentage):
        self._salary += self._salary * (percentage / 50)
```

Good – reusing code

- When we use specialisation, we should use the superclass' functionality where we can.
- This promotes maintainability in our code
  - For example, if a bug occurs in givePayRaise() the fewer places where code manages this functionality the better

```
def getPayRaise(self, percentage):

        super().givePayRaise(percentage * 2)  #1

#1 – use of super() means we are calling givePayRaise()
#       from parent class
```

- Of course, this might not always be possible. But this is fine – where the functionality is too different it should be in its own method anyway.


Next Time:
Inheritance