Introduction to Programming (Adv)

School of Computer Science, University of Sydney

Copyright Warning

Lecture 8: Inheritance

Types and sub types

“is-a”

Types can be subtypes of other types.

- *cat* is a kind of *pet*
- a *tiger* is a kind of cat
- an integer is a kind of number

The $\boxed{\text{“is-a”}}$ relationship is often held as a very important descriptor for the way different types are related in Object-Oriented programming.

Sub classing

How do we say that one thing is a kind of another thing?

We use the class definition to denote the "is-a" relationship:

```
class DigiSet:
    # all my code for a set of digits
    ...
```

```
class DigiSetBitwise (DigiSet):
    # because I want to do some things differently
    ...
```

This means that my `DigiSetBitwise` will *inherit* all the methods that are not marked `private` in `DigiSet`, and it might have more methods and fields too.

sub- and super-types

If one class ‘A’ *subclasses* another class ‘B’ we say A is a *subtype* of B. Equivalently, if A “is-a” B then A is a *subtype* of B.

In this situation, B is also called a *supertype* of A.

When both A and B are classes, A is a subclass of B and B is a superclass of A.

E.g., *Pet* is a superclass of *DomesticDog* and *DomesticCat*. *DomesticCat* is a subclass of *Pet*.

Apples and Oranges are Fruit

I've claimed that apples and oranges are both Fruit.
So `Apple` and `Orange` are subtypes of `Fruit`.

In fact I can also say that apples and orange are both *fruit*.
I can then write something like this:

```
class Fruit:
    def squeeze(self):
        # somehow extract the juice
        ...
    def blend(self):
        # the most important question: will it blend?
        ...
```

```
class Apple (Fruit):
    '''Inherits all the methods.
    May have additional methods.'''
```

(Similarly for `Orange` class).

Subtypes inherit behaviour

The *subtype* of a thing by default gets all the behaviour that the *supertype* has.

That means methods available in the superclass that are not marked can be used by *subtypes*.

Let's see how this works for a simple example.

Inheritance example

```python
class Foo:
    '''objects in Foo class and all subclasses could access'''

    def __init__(self, s): # constructor
        self.name = s    # set the name to the argument s

    def foo_specific(self):
        return "At my core, I am a foo! - " + self.name

    def greet(self): # return a friendly message
        return "Hello, I am a Foo! and my name is " + self.name

    def __str__(self):
        # overwrites the string version of this object
        return self.greet()
```

Inheritance example (cont.)

```python
from Foo import Foo as Foo

class Bar(Foo):
    # constructor
    def __init__(self, s="I have no name. :("):
        # "super" by itself calls the constructor of the superclass
        super().__init__(s)
        # call the Foo constructor that takes a string

    # This indicates greet() is overriding a method in Foo.
    # Optional, but the compiler will check if it's present.
    def greet(self):
        return "Hello, I am a Bar, which is a kind of Foo!"
```

Inheritance example (cont.)

```python
from Foo import Foo as Foo
from Bar import Bar as Bar

foo = Foo("Harry")
bar = Bar()
c = Bar("James")

fooArray = [ foo, bar, c ]
# apply idiom to do something with all Foo types
for f in fooArray:
    print( f.greet() )
```

What does this print?

Inheritance example (cont.)

```python
from Foo import Foo as Foo
from Bar import Bar as Bar

foo = Foo("Harry")
bar = Bar()
c = Bar("James")

fooArray = [ foo, bar, c ]
# apply idiom to do something with all Foo types
for f in fooArray:
    print( f.greet() )
```

What does this print?

Running the Foo/Bar program:

```
Hello, I am a Foo! and my name is Harry
Hello, I am a Bar, which is a kind of Foo!
Hello, I am a Bar, which is a kind of Foo!
```

All `Bar`s are `Foo`s

If all `Bar` objects are `Foo` objects, then we could treat a collection of `Foo` and `Bar` objects all as `Foo`s. Right?

All Bars are Foos (cont.)

Right!

```
from Foo import Foo as Foo
from Bar import Bar as Bar

foo = Foo("Harry")
bar = Bar()
c = Bar("James")

fooArray = [ foo, bar, c ]
for f in fooArray:
    # treat it as a Foo object
    print( f.foo_specific() )
```

Running the Foo/Bar program again:

```
At my core, I am a foo! - Harry
At my core, I am a foo! - I have no name. :(
At my core, I am a foo! - James
```

Just Bar objects?

```
from Foo import Foo as Foo
from Bar import Bar as Bar

foo = Foo("Harry")
bar = Bar()
c = Bar("James")

array = [ foo, bar, c ]

for b in array:
   # additional safeguard
   if isinstance(b, Bar):
      print( b.greet() )
```

```
Hello, I am a Bar, which is a kind of Foo!
Hello, I am a Bar, which is a kind of Foo!
```

The super constructor

When you create an object that is a subclass of another class (e.g., an Apple, which is a subclass of a Fruit), then you need to handle the constructor of the subclass specially.

You must construct the instance of the super class *first* and then do any other construction for the derived class.

To call the constructor for the super class you use the `super` keyword, such as here:

```
class Vehicle:
   def Vehicle(self, mass):
        self.mass = mass
```

The super constructor (cont.)

```python
class Hovercraft(Vehicle):
   def Hovercraft(self, model):
        mass = 500
        if model == "Viper 5":
            mass = 265
        elif model == "Pomornik":
            mass = 340000
        super().__init__(mass)
```

```python
class Motorcycle(Vehicle):
    def Motorcycle(self, mass, ccs):
        super().__init__(mass)
        self.ccs = ccs
```

The super constructor (cont.)

```
class Car(Vehicle):
    def Car(self, mass, doors):
        super().__init__(mass)
        if doors < 1 or doors > 5:
            raise VehicleTypeException("The number of doors for
                Car is outside the expected range [1,5]" + str(
                doors))
        self.doors = doors
```

super

To access something in the superclass, we use the method `super()`:

`super()` calls the parent superclass

`super.myMethod()` calls the `myMethod()` method in the superclass.

No, it's not possible to call `super().super()`.

Summary

Identify new data types, there can be a commonality among them

Using subclassing allows use of inherited behaviour

Using subclassing allows one to override inherited behaviour, for specialisation