5. előadás

Többszörös öröklődés Python-ban.

Programozás (2) előadás

2022. Október 10.

Halász Gábor

Általános tudnivalók

Ajánlott irodalom:

- Nyékyné G. Judit (szerk): Programozási nyelvek, Kiskapu, 2003.
- Juhász, István: Magas szintű programozási nyelvek 2, elektronikus egyetemi jegyzet, 2009
- Tarczali, Tünde: UML diagramok a gyakorlatban, Typotex Kiadó, 2011.
- Angster, Erzsébet: Objektumorientált tervezés és programozás: JAVA, 4KÖR Bt., 2002, ISBN: 9632165136
- Bird, S., Klein, E., Loper, E.: Natural Language Processing with Python, O'Reilly Media, 2009

Félév teljesítésének feltételei: jelenlét + 2 gyakorlati + 1 elméleti ZH

Érdemjegy: $1 < 60\% \leq 2 < 70\% \leq 3 < 80\% \leq 4 < 90\% \leq 5$

További részletek: `https://elearning.unideb.hu/`

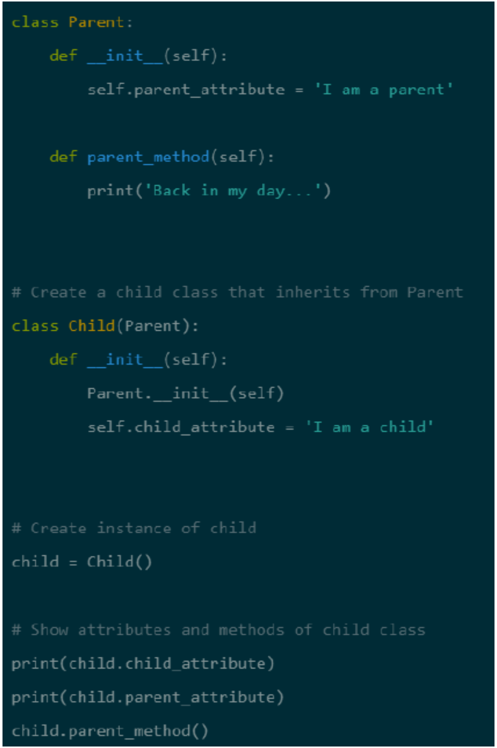
Öröklődés

Quick overview of inheritance

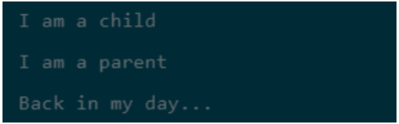
- As you grow your Python projects and packages, you'll want to utilize classes and apply the DRY (don't-repeat-yourself) principle.
- Class inheritance is a fantastic way to create a class based on another class in order to stay DRY.

Quick overview of inheritance

- So what is class inheritance?
- Similarly to genetics, a child class can ’inherit’ attributes and methods from a parent.
- In the next code block we’ll demonstrate inheritance with a Child class inheriting from a Parent class.



```
class Parent:
    def __init__(self):
        self.parent_attribute = 'I am a parent'

    def parent_method(self):
        print('Back in my day...')


# Create a child class that inherits from Parent
class Child(Parent):
    def __init__(self):
        Parent.__init__(self)
        self.child_attribute = 'I am a child'


# Create instance of child
child = Child()

# Show attributes and methods of child class
print(child.child_attribute)
print(child.parent_attribute)
child.parent_method()
```



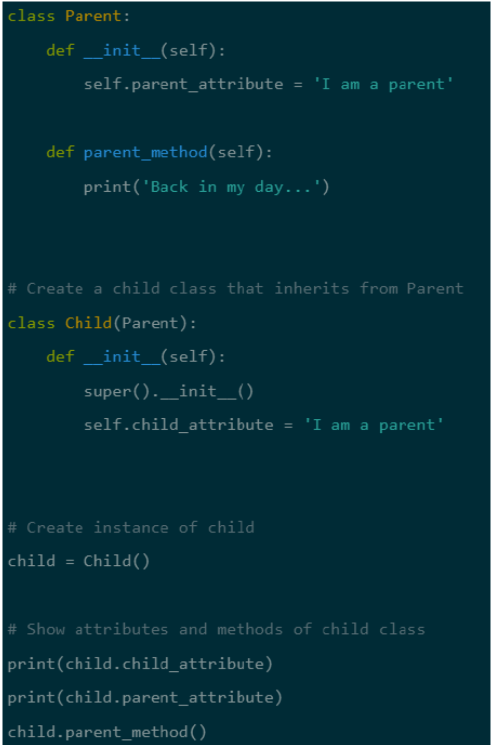
```
I am a child
I am a parent
Back in my day...
```

Quick overview of inheritance

- We see that the Child class ’inherited’ attributes and methods from the Parent class.
- To get the benefits of the `Parent.__init__()` method we needed to explicitly call the method and pass self.
- This is because when we added an `__init__` method to Child, we overwrote the inherited `__init__`.

Intro to Super

- In the simplest case, the super function can be used to replace the explicit call to `Parent.__init__(self)`.
- Our example from the first section can be rewritten with super as seen below.
- Note, that the below code block is written in Python 3, earlier versions use a slightly different syntax.



```
class Parent:
    def __init__(self):
        self.parent_attribute = 'I am a parent'

    def parent_method(self):
        print('Back in my day...')


# Create a child class that inherits from Parent
class Child(Parent):
    def __init__(self):
        super().__init__()
        self.child_attribute = 'I am a parent'


# Create instance of child
child = Child()

# Show attributes and methods of child class
print(child.child_attribute)
print(child.parent_attribute)
child.parent_method()
```

Intro to Super

Cons: It can be argued that using super here makes the code less explicit. "Explicit is better than implicit."

Pros: There is a maintainability argument that can be made for super even in single inheritance. If for whatever reason your child class changes its inheritance pattern then there’s no need find and replace all the lingering references to ParentClass.method_name

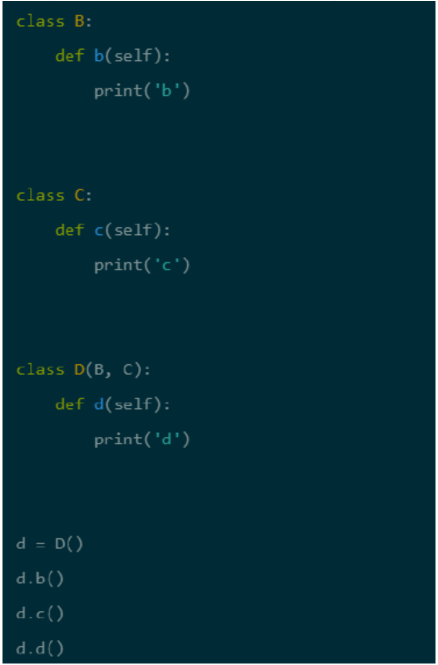
Super and multiple inheritance

- First off, what is multiple inheritance?
- So far the example code has covered a single child class inheriting from a single parent class.
- In multiple inheritance, there’s more than one parent class. A child class can inherit from 2, 3, 10, etc. parent classes.
- Here is where the benefits of super become more clear.

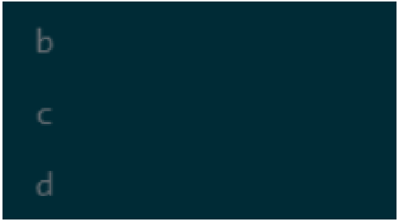
Super and multiple inheritance

- Let’s look at an example of multiple inheritance that avoids modifying any parent methods and in turn avoids super

Super and multiple inheritance



```
class B:
    def b(self):
        print('b')


class C:
    def c(self):
        print('c')


class D(B, C):
    def d(self):
        print('d')


d = D()
d.b()
d.c()
d.d()
```

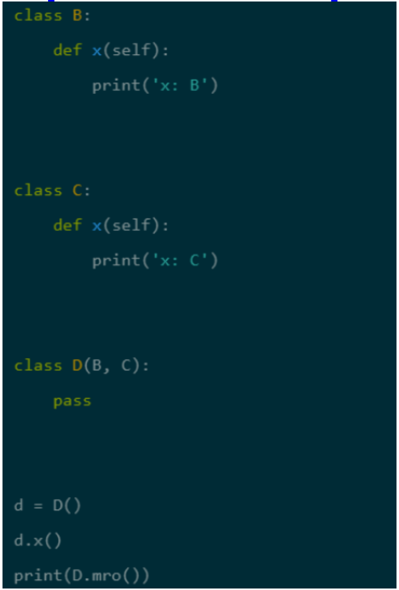


```
b
c
d
```

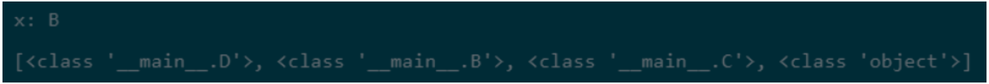
Super and multiple inheritance

- So what if both B and C both had a method with the same name?
- This is where a concept called ’multiple-resolution order’ comes into play or MRO for short.
- The MRO of a child class is what decides where Python will look for a given method, and which method will be called when there’s a conflict.

Super and multiple inheritance



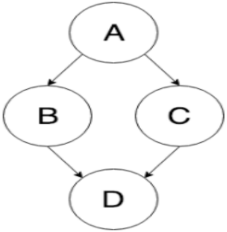
```
class B:
    def x(self):
        print('x: B')


class C:
    def x(self):
        print('x: C')


class D(B, C):
    pass


d = D()
d.x()
print(D.mro())
```



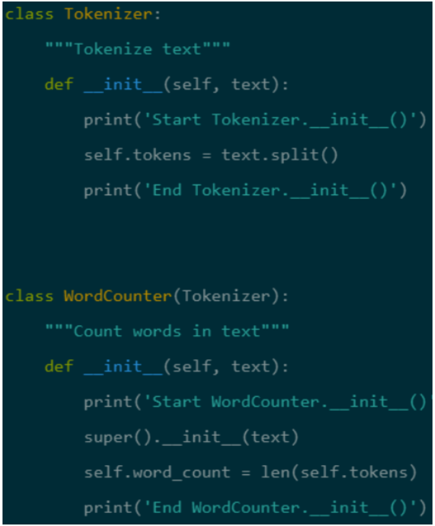
```
x: B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
```

Multiple inheritance, super, and the diamond problem

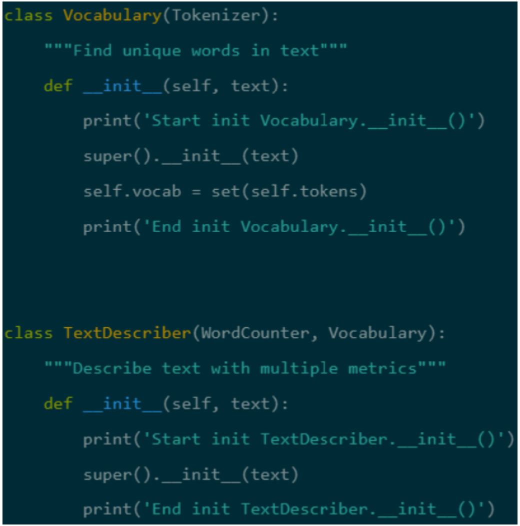
- Below is an example of using super to handle MRO of init in a way that's beneficial.
- We'll create 4 classes, and the structure for inheritance will follow the structure in the below diagram.



Multiple inheritance, super, and the diamond problem

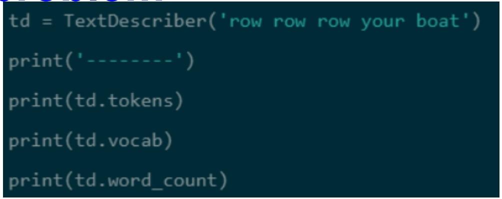


```python
class Tokenizer:
    """Tokenize text"""
    def __init__(self, text):
        print('Start Tokenizer.__init__()')
        self.tokens = text.split()
        print('End Tokenizer.__init__()')


class WordCounter(Tokenizer):
    """Count words in text"""
    def __init__(self, text):
        print('Start WordCounter.__init__()'
        super().__init__(text)
        self.word_count = len(self.tokens)
        print('End WordCounter.__init__()')
```

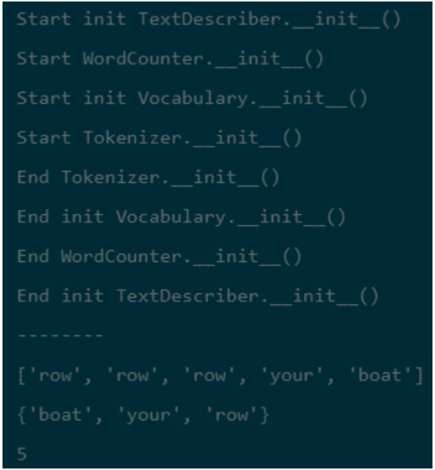


```python
class Vocabulary(Tokenizer):
    """Find unique words in text"""
    def __init__(self, text):
        print('Start init Vocabulary.__init__()')
        super().__init__(text)
        self.vocab = set(self.tokens)
        print('End init Vocabulary.__init__()')


class TextDescriber(WordCounter, Vocabulary):
    """Describe text with multiple metrics"""
    def __init__(self, text):
        print('Start init TextDescriber.__init__()')
        super().__init__(text)
        print('End init TextDescriber.__init__()')
```

Multiple inheritance, super, and the diamond problem



```
td = TextDescriber('row row row your boat')

print('--------')

print(td.tokens)

print(td.vocab)

print(td.word_count)
```



```
Start init TextDescriber.__init__()

Start WordCounter.__init__()

Start init Vocabulary.__init__()

Start Tokenizer.__init__()

End Tokenizer.__init__()

End init Vocabulary.__init__()

End WordCounter.__init__()

End init TextDescriber.__init__()

--------

['row', 'row', 'row', 'your', 'boat']

{'boat', 'your', 'row'}

5
```

Multiple inheritance, super, and the diamond problem

- We learned about the super function and how it can be used to replace `ParentName.method` in single inheritance.
- We learned about multiple inheritance and how we can pass on the functionality of multiple parent classes to a single child class.
- We learned about multiple-resolution order and how it decides what happens in multiple inheritance when there’s a naming conflict between parent methods.
- We learned about the diamond problem and saw an example of how the use of super navigates the diamond.

Abstract Classes in Python

Abstract Classes in Python

- An abstract class can be considered as a blueprint for other classes, allows you to create a set of methods that must be created within any child classes built from your abstract class.
- A class which contains one or more abstract methods is called an abstract class.
- An abstract method is a method that has declaration but not has any implementation.
- Abstract classes are not able to instantiated and it needs subclasses to provide implementations for those abstract methods which are defined in abstract classes.

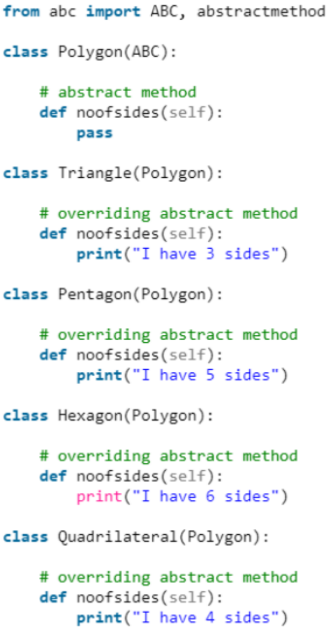
Abstract Classes in Python

- Abstract classes allow partially to implement classes when it completely implements all methods in a class, then it is called interface.
- Abstract classes allow you to provide default functionality for the subclasses. By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses.
- In python by default, it is not able to provide abstract classes, but python comes up with a module which provides the base for defining Abstract Base Classes(ABC) and that module name is ABC.

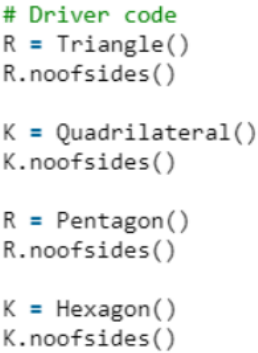
Abstract Classes in Python

- Abstract classes are incomplete because they have methods which have no body.
- If python allows creating an object for abstract classes then using that object if anyone calls the abstract method, but there is no actual implementation to invoke.
- So we use an abstract class as a template and according to the need we extend it and build on it before we can use it.
- Due to the fact, an abstract class is not a concrete class, it cannot be instantiated. When we create an object for the abstract class it raises an *error*.

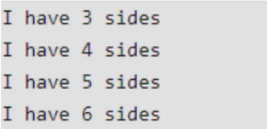
Abstract Classes in Python



```python
from abc import ABC, abstractmethod

class Polygon(ABC):

    # abstract method
    def noofsides(self):
        pass

class Triangle(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")

class Pentagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")

class Quadrilateral(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 4 sides")
```



```python
# Driver code
R = Triangle()
R.noofsides()

K = Quadrilateral()
K.noofsides()

R = Pentagon()
R.noofsides()

K = Hexagon()
K.noofsides()
```

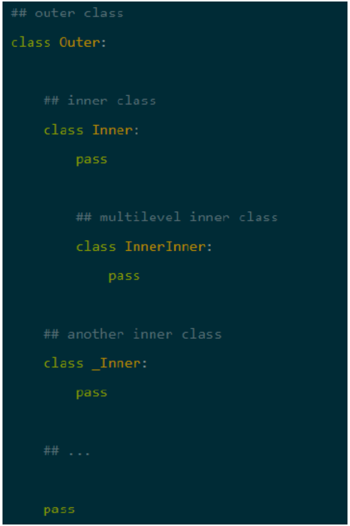


```
I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides
```

Inner Classes in Python

Inner Classes in Python

- **Inner or Nested Class** is defined inside another class. See the structure of *inner or nested classes*.



```
## outer class
class Outer:

    ## inner class
    class Inner:
        pass

        ## multilevel inner class
        class InnerInner:
            pass

    ## another inner class
    class _Inner:
        pass

    ## ...

    pass
```

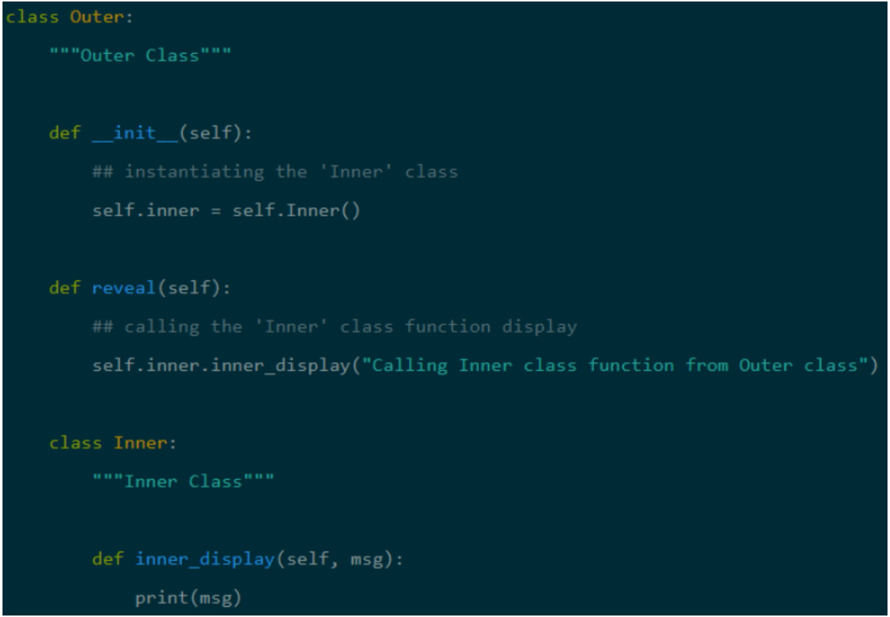
Inner Classes in Python

- Why Inner Classes?
  - Grouping of two or more classes. Suppose you have two classes **Car** and **Engine**. Every *Car* needs an *Engine*. But, *Engine* won't be used without a *Car*. So, you make the *Engine* an inner class to the *Car*.
  - It helps save code.
  - Hiding code is another use of *Nested classes*. You can hide the *Nested classes* from the outside world.
  - It's easy to understand the classes. Classes are closely related here. You don't have to search for the classes in the code. They are all together.

Inner Classes in Python

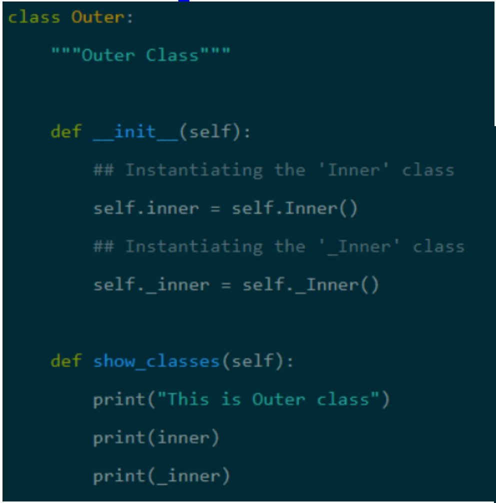
- *Inner or Nested* classes are not the most commonly used feature in *Python*. But, it can be a good feature to implement code.
- The code is straightforward to organize when you use the *inner or nested classes*.
- You can access the **inner class** in the **outer class** using the `self` keyword. So, you can quickly create an instance of the **inner class** and perform operations in the **outer class** as you see fit.
- You can't access the **outer class** in an **inner class**.

Inner Classes in Python

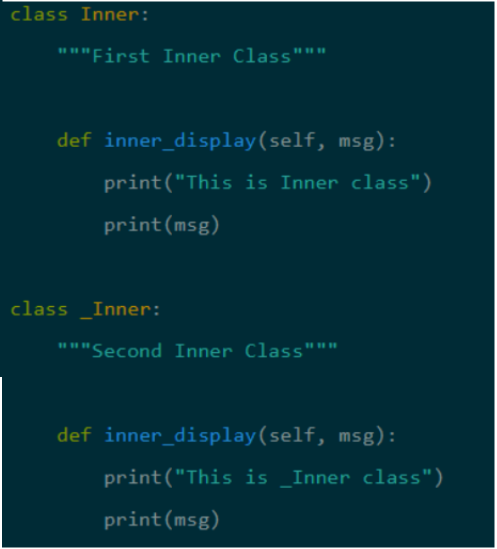


```
class Outer:
    """Outer Class"""

    def __init__(self):
        ## instantiating the 'Inner' class
        self.inner = self.Inner()

    def reveal(self):
        ## calling the 'Inner' class function display
        self.inner.inner_display("Calling Inner class function from Outer class")

    class Inner:
        """Inner Class"""

        def inner_display(self, msg):
            print(msg)
```

Multiple Inner Classes

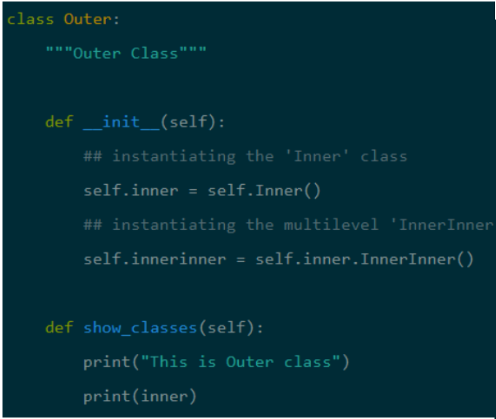


```python
class Outer:
    """Outer Class"""

    def __init__(self):
        ## Instantiating the 'Inner' class
        self.inner = self.Inner()
        ## Instantiating the '_Inner' class
        self._inner = self._Inner()

    def show_classes(self):
        print("This is Outer class")
        print(inner)
        print(_inner)
```

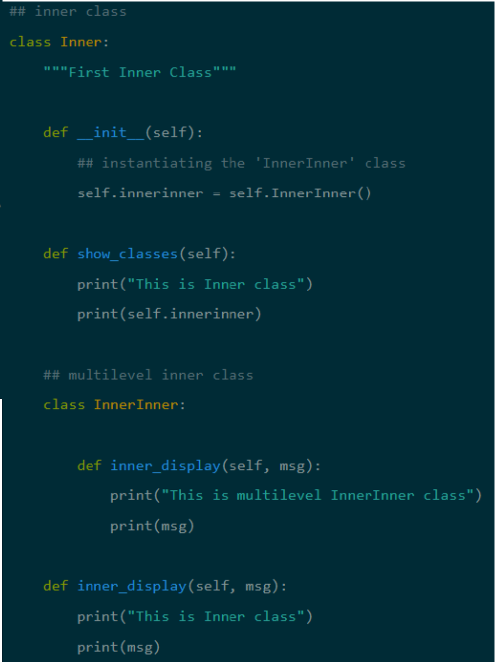


```python
class Inner:
    """First Inner Class"""

    def inner_display(self, msg):
        print("This is Inner class")
        print(msg)

class _Inner:
    """Second Inner Class"""

    def inner_display(self, msg):
        print("This is _Inner class")
        print(msg)
```

Multilevel Inner Classes



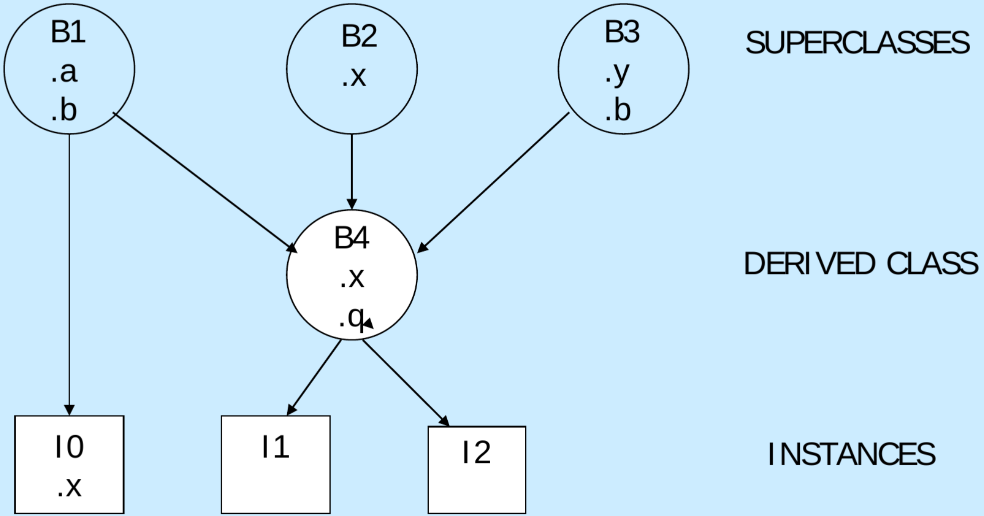
```
class Outer:
    """Outer Class"""

    def __init__(self):
        ## instantiating the 'Inner' class
        self.inner = self.Inner()
        ## instantiating the multilevel 'InnerInner
        self.innerinner = self.inner.InnerInner()

    def show_classes(self):
        print("This is Outer class")
        print(inner)
```



```
## inner class
class Inner:
    """First Inner Class"""

    def __init__(self):
        ## instantiating the 'InnerInner' class
        self.innerinner = self.InnerInner()

    def show_classes(self):
        print("This is Inner class")
        print(self.innerinner)

    ## multilevel inner class
    class InnerInner:

        def inner_display(self, msg):
            print("This is multilevel InnerInner class")
            print(msg)

    def inner_display(self, msg):
        print("This is Inner class")
        print(msg)
```

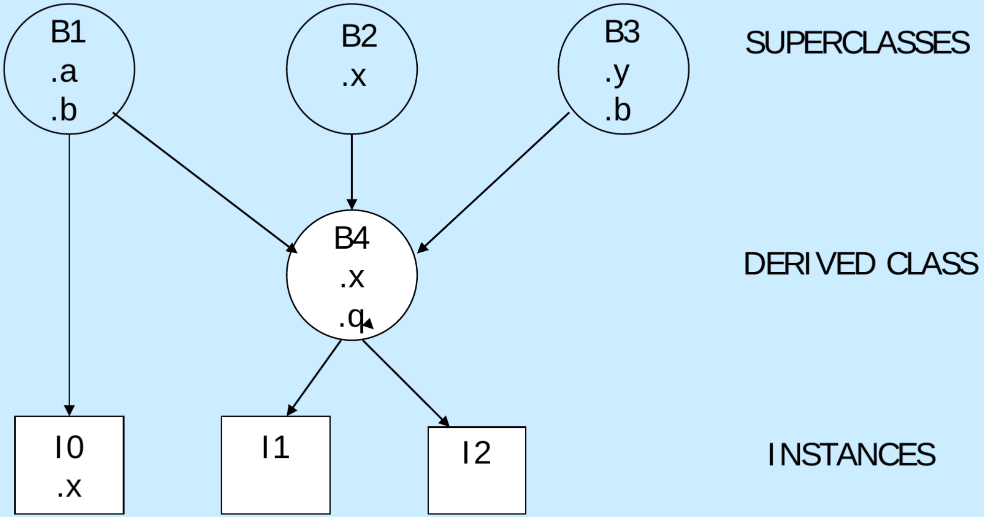
Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.
9. Make sure your class does the right thing.



I0.a és I0.b a B1-ben vannak definiálva;

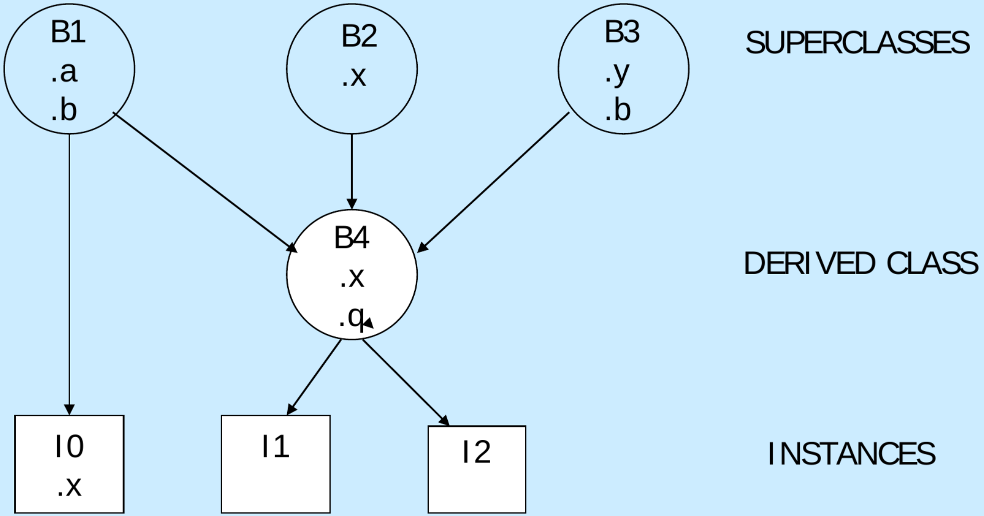
I0.x viszont I0-ban



I1.a, I1.b, I2.a és I2.b

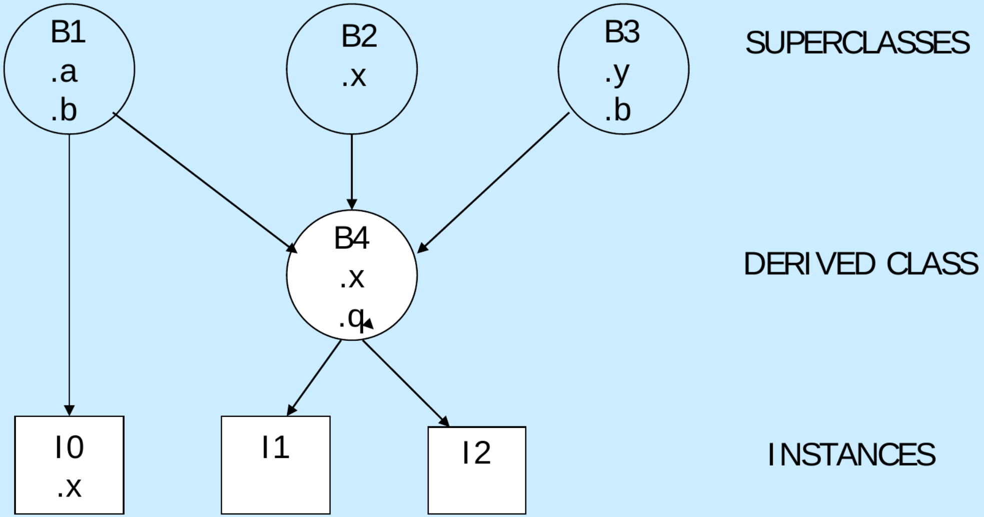
a B1-ben vannak definiálva;

I1.x és I2.x viszont B4-ben



I0.y, I1.y és I2.y a B3-ban vannak definiálva;

és így tovább



Hozzáférhetünk-e valahogyan (írás vagy olvasás céljából) az `I0` objektum `B2` vagy `B4` osztályokban definiált `x` ill. a `B3`-ban definiált `b` attribútumához?