Classes Inheritance and Polymorphism

Inheritance

Common ancestor with Amphibia - Common ancestor with Reptiles - Mammals

In this scheme Common ancestor with Amphibia is the ancestor of the following taxa

In python we have quite the same situation

```
class PreDragon:
   has_scale = True
   can_fly = False

   def __init__(self, age, name):
       self.age = age
       self.name = name


class Dragon(PreDragon):
   public_animal_type = 5
   can_fly = True

   def __init__(self, age, name):
       super().__init__(age, name)
       self.length = 2
```

```
dragon_izera = Dragon(4, 'Izera')

print(dragon_izera.age, dragon_izera.length)
4 2

print(dragon_izera.has_scale, dragon_izera.can_fly)
True True
```

Redefining attribute or method in the child class is called overload

What's going on?

```
class Class:
    def __init__(self):
        pass

class Descendant(Class):
    def __init__(self):
        super().__init__()
```

Class ancestor can be passed in the class definition

`super()` - function to get ancestor class

super

`super([type, [object]])` - both parameters are optional, by default they refer to this class and object of this class; superclass of the passed will be inferred

Multiple Class Inheritance

With it you can create more flexible classes, and make your code really complicated.

Looking ahead, there are some alternatives - have a look at composition and mixins

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

    def battlecry(self, message='Charge!'):
        print(f'{self.name}: {message}')

class Bull:
    def bullfight(self):
        print('Attacking...')


class Minotaur(Human, Bull):
    pass
```

```
mino = Minotaur('Darkstorn')

mino.battlecry()
Darkstorn: Charge!
mino.bullfight()
Attacking...
```

As you can see all methods are available for Minotaur

Method Resolution Order

What if we had methods with the same name in both ancestors?

One of them will be executed - from the first superclass

What if you want from another one?

How to resolve this issue?

- Change order of classes (doesn't look as an awesome solution for me)
- Class cooperation - make distinct signatures for methods in classes
- Use desired class name before invoked method

Polymorphism

Different behaviour of objects despite the same invoked method. So you can use the same code for objects of different classes (though, you might get different results)

Based on the overload - redefining methods in classes

```python
class PreDragon:
    has_scale = True
    can_fly = False

    def __init__(self, age, name):
        self.age = age
        self.name = name

    def attack(self):
        print(f'{self.name} dealt 5 damage with claws')
```

```python
class FireDragon(PreDragon):
    can_fly = True

    def __init__(self, age, name):
        super().__init__(age, name)

    def attack(self):
        print(f'{self.name} dealt 7 damage with fire')


class TerrestrialDragon(PreDragon):
    def __init__(self, age, name):
        super().__init__(age, name)

    def attack(self):
        print(f'{self.name} dealt 6 damage with beak')
```

```
ogonek = FireDragon(7, 'Ogonek')
tuzik = TerrestrialDragon(8, 'Tuzik')

ogonek.attack()
Ogonek dealt 7 damage with fire
tuzik.attack()
Tuzik dealt 6 damage with beak
```

Another polymorphism example

```python
class Dragon:
   def __init__(self, age, name):
       self.age = age
       self.name = name
       self.length = 2

   def __len__(self):
       return self.length

   def __gt__(self, other):
       if isinstance(other, Dragon):
           return len(self) > len(other)
       raise ArithmeticError
```

```
pushok = Dragon(3, 'Pushok')
sharik = Dragon(3, 'Sharik')
sharik.length = 3

print(len(pushok))
2
print(len(sharik))
3


print(sharik > pushok)
True
```

Abstract Classes

Classes with abstract methods
Abstract method - method which should be overridden in the subclass

So, abstract classes are intended to be subclassed, they act like a predefined contract

Abstract class can't be instantiated and all its abstract method must be overriden to make an instantiable class

```python
from abc import ABC, abstractmethod


class AbstractDragon(ABC):
   @abstractmethod
   def be_dragonic(self):
       """Each dragon should be dragonic..."""
       pass

a = AbstractDragon()
a.be_dragonic()

TypeError: Can't instantiate abstract class
AbstractDragon with abstract methods be_dragonic
```

```python
class BlackDragon(AbstractDragon):
    def be_dragonic(self):
        """Each dragon should be dragonic..."""
        print('Fly and terrify')

a = BlackDragon()
a.be_dragonic()
```

```
Fly and terrify
```