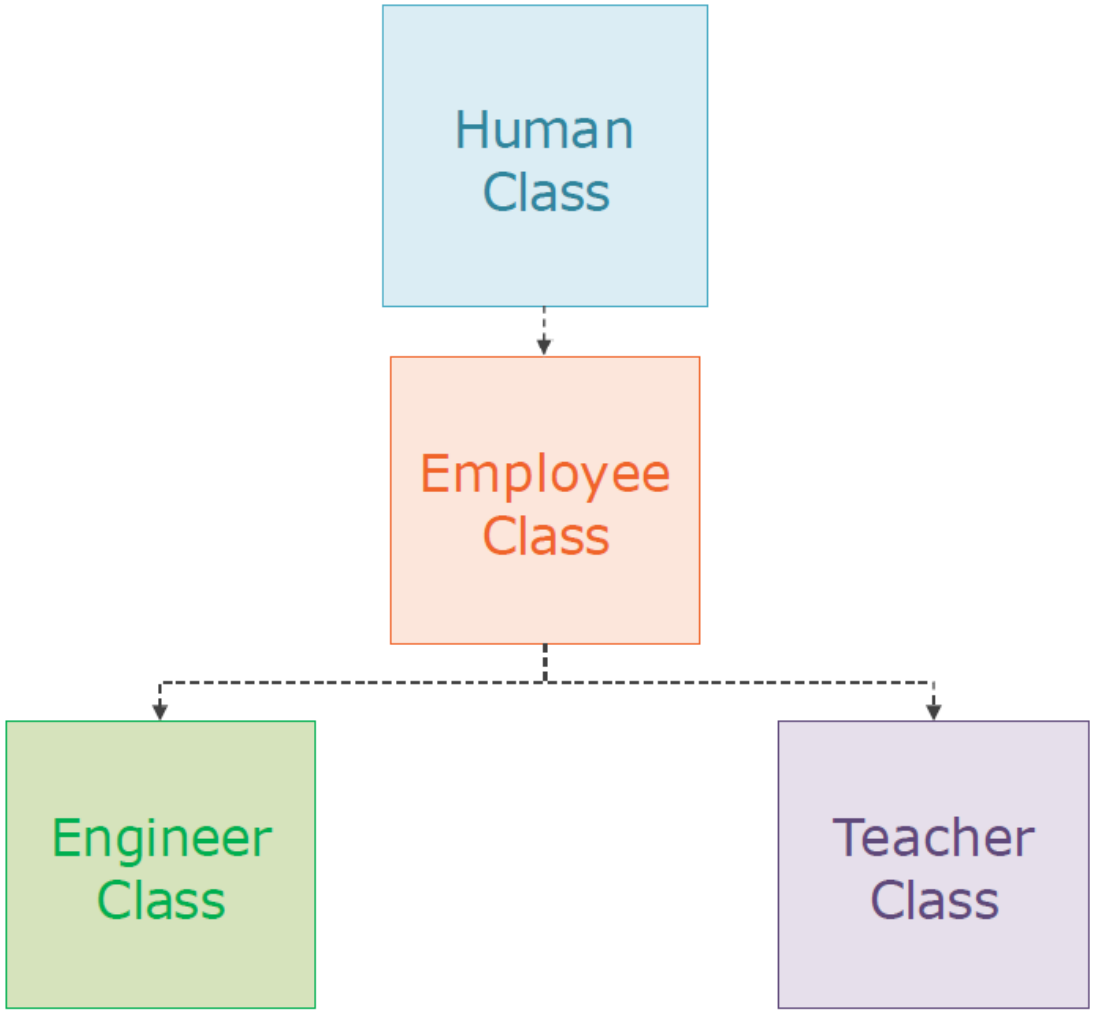
OOP
USING PYTHON

Outline

- Inheritance
- Polymorphism
- Encapsulation
- Special Methods

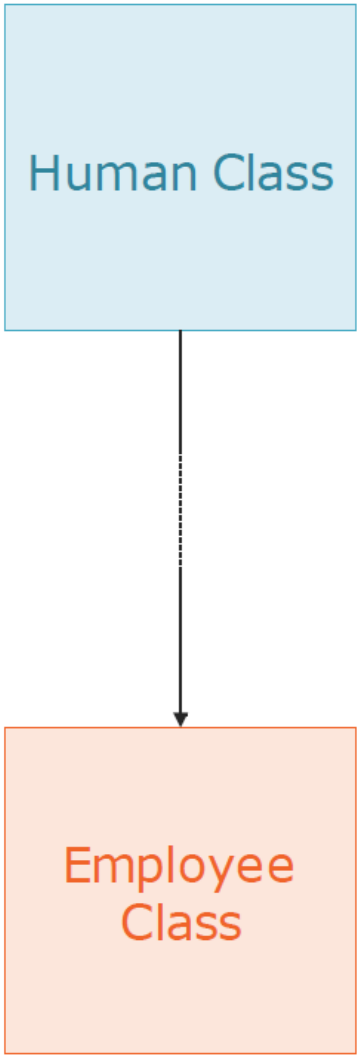
Inheritance

- Parent class
- Child Class
- Is a relation



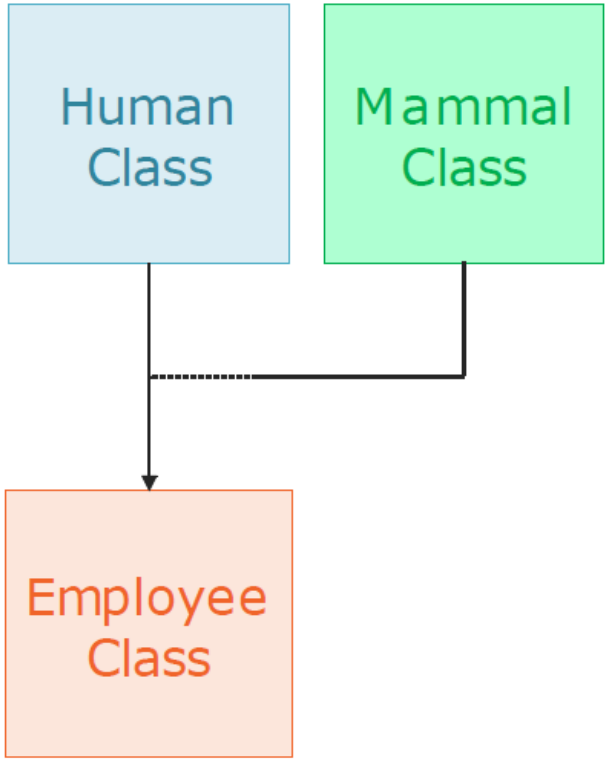
Example

```
class Human:
    def_init_(self, name):
        self.name = name
    def speak(self):
        print("My Name is "+self.name);
class Employee(Human):
    def_init_(self, name, salary):
        super(Employee, self)._init_(name)
        self.salary = salary
    def work(self):
        print("I'm working now");
emp = Employee("Ahmed", 500)
emp.speak()
emp.work()
```



Multiple Inheritance

Python supports Multiple Inheritance



Report**:

1- How **super** Function handle **Multiple** Inheritance.

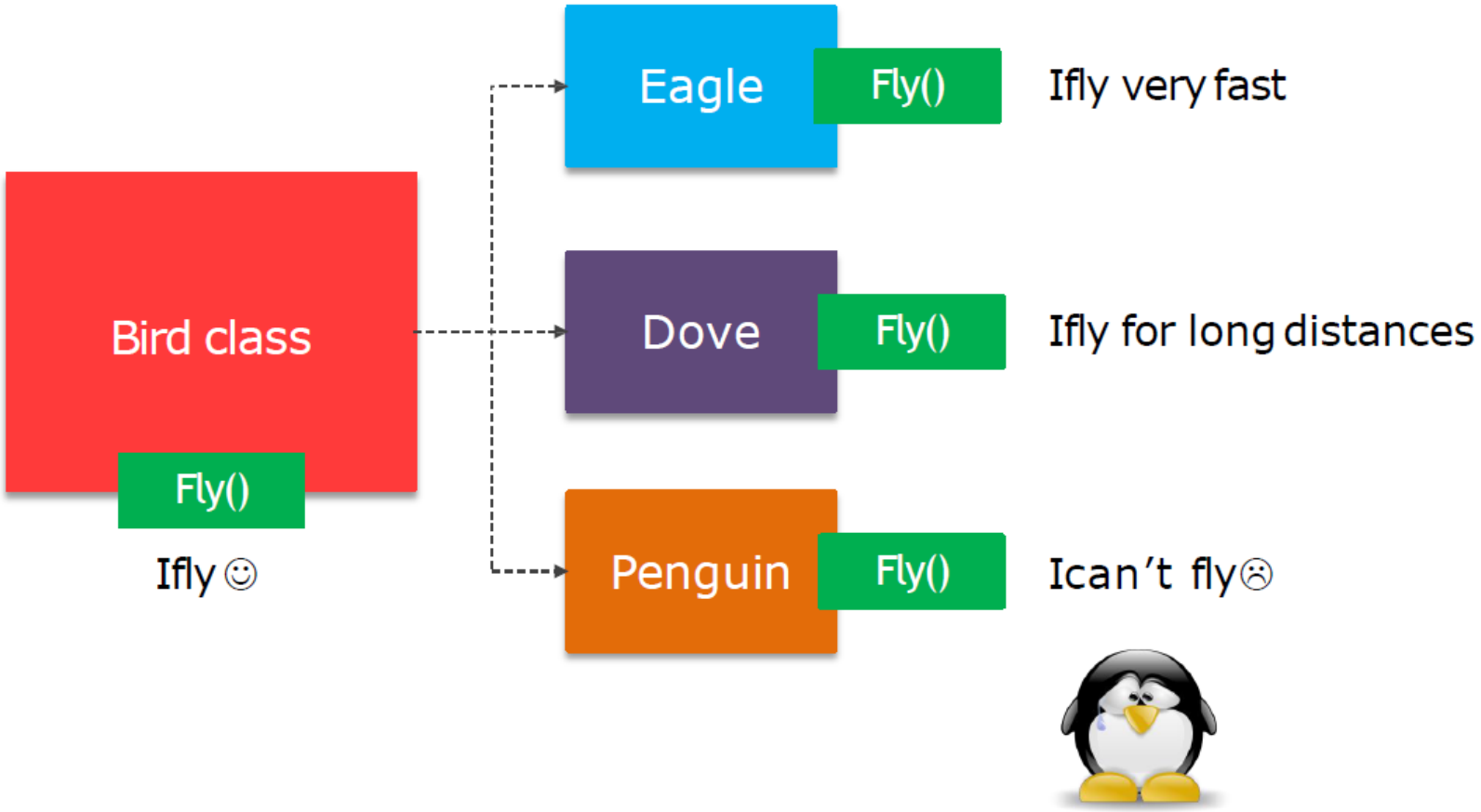
2- If Human and Mammal Have the same method like **eat** but with different Implementation. When Child [Employee] calls eat method how python handle this case.

**Prove your opinion with examples.

Polymorphism

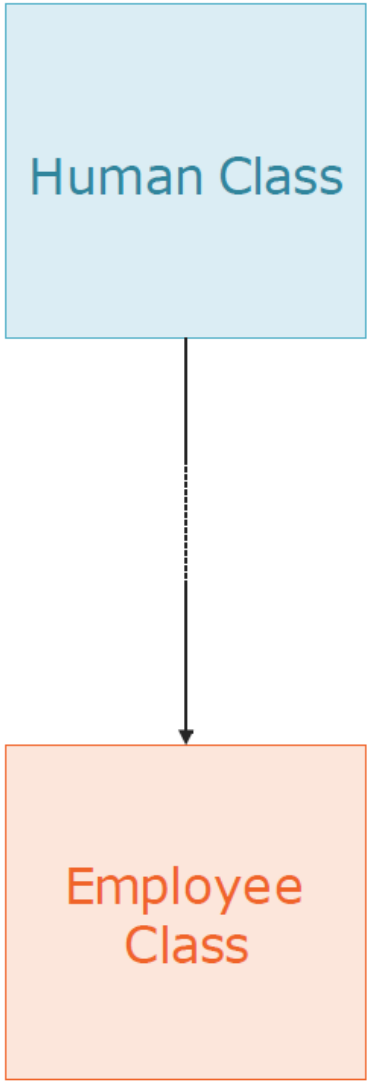
Polymorphism

Poly means "*many*" and **morphism** means "*forms*". Different classes might define the same method or property.



Method Overriding

```
class Human:
    def_init_(self, name):
          self.name = name
    def speak(self):
        print("My Name is "+self.name);
class Employee(Human):
    def_init_(self, name, salary):
        super(Employee, self)._init_(name)
        self.salary = salary
    def speak(self):
        print("My salary is "+self.salary);
emp = Employee("Ahmed", 500)
emp.speak() #My Salary is 500
```



Method Overloading

- Two or more methods have the **same name** but **different** numbers of parameters or different types of parameters, or both.
- python does **not support** method overloading by default. But there are different ways to achieve method overloading in Python.

- The problem with method overloading in Python is **that we may overload the methods but can only use the latest defined method**.

Method 1 (Not The Most Efficient Method):

- We can use the arguments to make the same function work differently i.e. as per the arguments.

```
# Function to take multiple arguments
2 usages
def add(datatype, *args):
    # if datatype is int , initialize answer as 0
    if datatype == 'int':
        answer = 0
    # if datatype is str, initialize answer as ''
    if datatype == 'str':
        answer = ''
    # Traverse through the arguments
    for x in args:
        # This will do addition if the
        # arguments are int. Or concatenation
        # if the arguments are str
        answer = answer + x
    print(answer)
```

Method 2 (Not the efficient one)

- by user defined function using "None" keyword as default parameter .

```
def add(a=None, b=None):
    # Checks if both parameters are available
    # if statement will be executed if only one
    # parameter is available
    if a != None and b == None:
        print(a)
    # else will be executed if both are available
    # and returns addition of two
    else:
        print(a+b)


# two arguments are passed, returns addition of two
add( a: 2,  b: 3)
# only one argument is passed, returns a
add(2)
```

Method 3 (Efficient One)

- By Using Multiple Dispatch Decorator
- Multiple Dispatch Decorator Can be installed by:
- pip3 install multipledispatch

```
from multipledispatch import dispatch
@dispatch( *types: int, int)
def product(first, second):
    result = first*second
    print(result)
@dispatch( *types: int, int, int)
def product(first, second, third):
    result = first * second * third
3 usages
@dispatch( *types: float, float, float)
def product(first, second, third):
    result = first * second * third
    print(result)
```

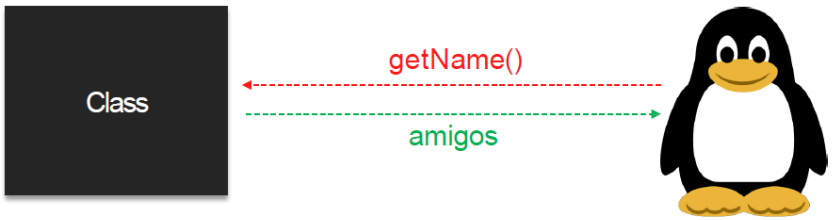
```
#calling
product( first: 2, second: 3) # this will give output of 6
product( first: 2, second: 3, third: 2) # this will give out
product( first: 2.2, second: 3.4, third: 2.3) # this will gi
```

Encapsulation

Special Methods

Encapsulation

- packing of data and functions into one component (for example, a class) and then controlling access to that component



Protected members

- Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its Childs classes.
- In python prefixing the name of the member by a single underscore “_”
- Although the protected variable can be

accessed out of the class

as well as in the derived

class (modified too in derived class),

it is customary(convention not a rule)

to not access the protected out the class body

```
class Base:
    def __init__(self):

        # Protected member
        self._a = 2
```

Private members

- define a private member prefix the member name with double underscore “__”

- Mangling

Encapsulation

```
class Human:
    def _init_(self, name):
          self._name = name
    def getName(self):
        return self.__name

man = Human(“Mahmoud”)
print(man.__name)
AttributeError: 'Human' object has no attribute '__name'

print(man.getName())
#output: Mahmoud
```

@property

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

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age > 0:
            self._age = age
        if age <= 0:
            self._age = 0

man = Human(23)
print(man.age)   # 23
man.age = -25
print(man.age)   # 0
```

Special Methods

Dunder or Magic Methods

__str__

- **Method that controls how Object treats as printable**

```
class Human:
    def_init_(self, name):
        self.name = name

    def __str__(self):
        return "Hi, I'm Human and my name is "+ self.name

man = Human("Ahmed")
print(man)
#output: <_main_.Human object at 0x000000FD81804400>
print(man)
#output: Hi, I'm Human and my name is Ahmed
```

__call__

- **Method that controls how Object can show as callable**

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

    def __call__(self):
        print("You called me !")

man = Human("Ahmed")

man()

#output: You called me !
```

__len__

- **Method that controls when measure the Object length**

```
class Animal:
    def_init_(self, legs):
            self.legs = legs

    def_len__(self):
        return self.legs

dog = Animal(4)

len(dog)

#output: 4
```

Exercises