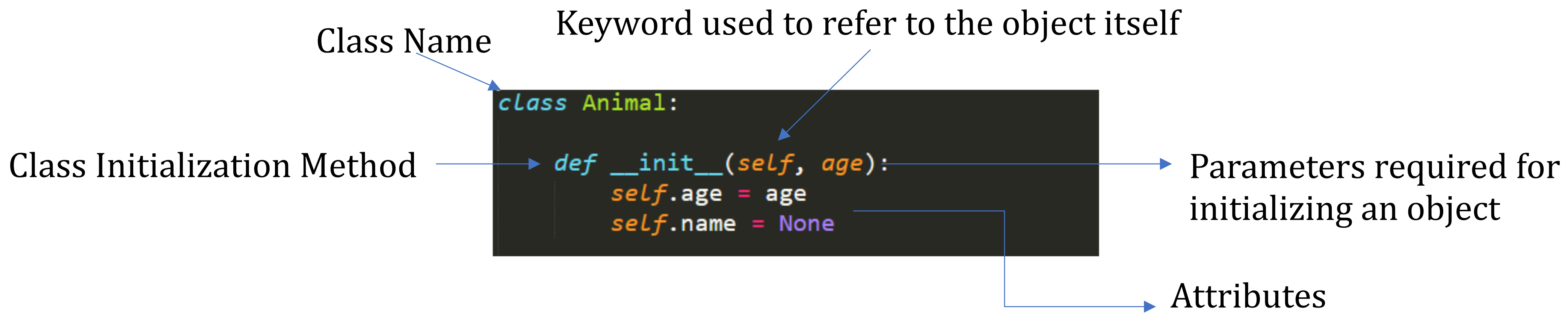
JOHNS HOPKINS
WHITING SCHOOL *of* ENGINEERING

EN.540.635 Software Carpentry

Lecture 9
Inheritance | Object-oriented Programming

Review: Classes

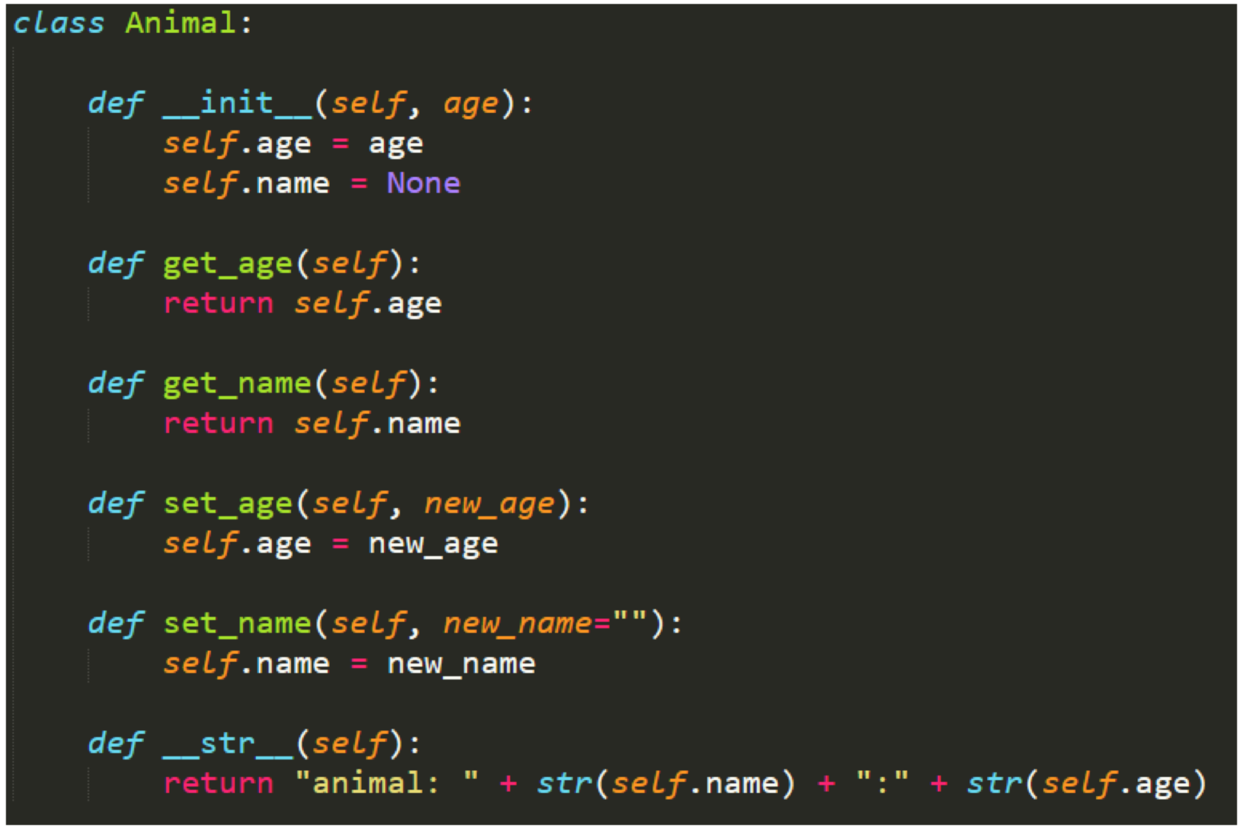

Class Name
Keyword used to refer to the object itself
Class Initialization Method
Parameters required for initializing an object
Attributes

```python
class Animal:

    def __init__(self, age):
        self.age = age
        self.name = None
```

- All instances of a class have the same names for methods and attributes, but have different values

Example taken from MIT OCW: Introduction to Python

Animal Class



```
class Animal:

    def __init__(self, age):
        self.age = age
        self.name = None

    def get_age(self):
        return self.age

    def get_name(self):
        return self.name

    def set_age(self, new_age):
        self.age = new_age

    def set_name(self, new_name=""):
        self.name = new_name

    def __str__(self):
        return "animal: " + str(self.name) + ":" + str(self.age)
```

It’s good practice to only modify object attributes using ‘getter’ and ‘setter’ functions, as it sticks to the OOP principle of ‘Encapsulation’

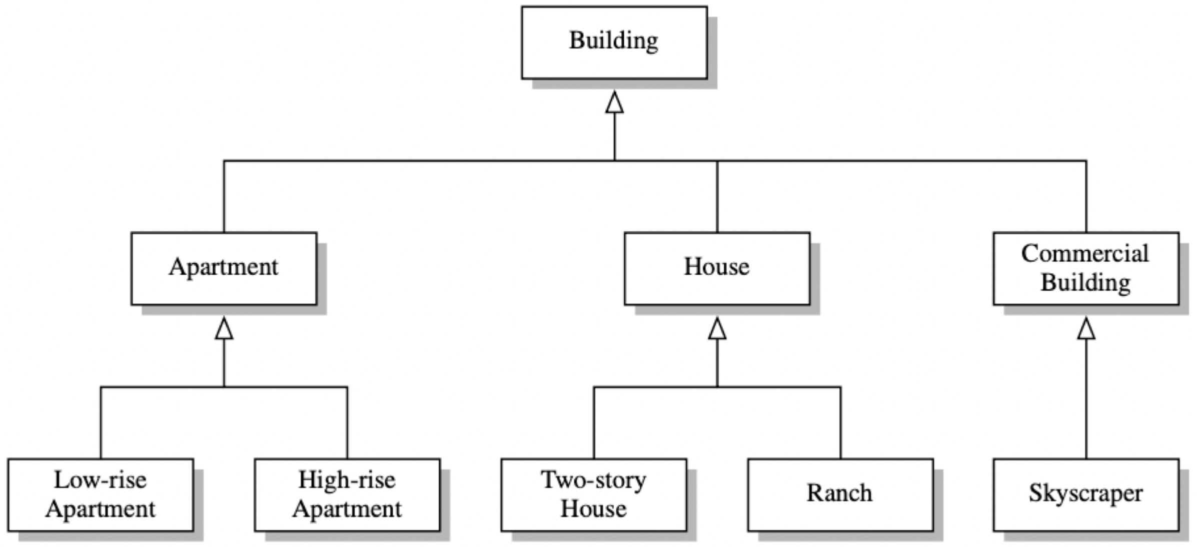
Inheritance

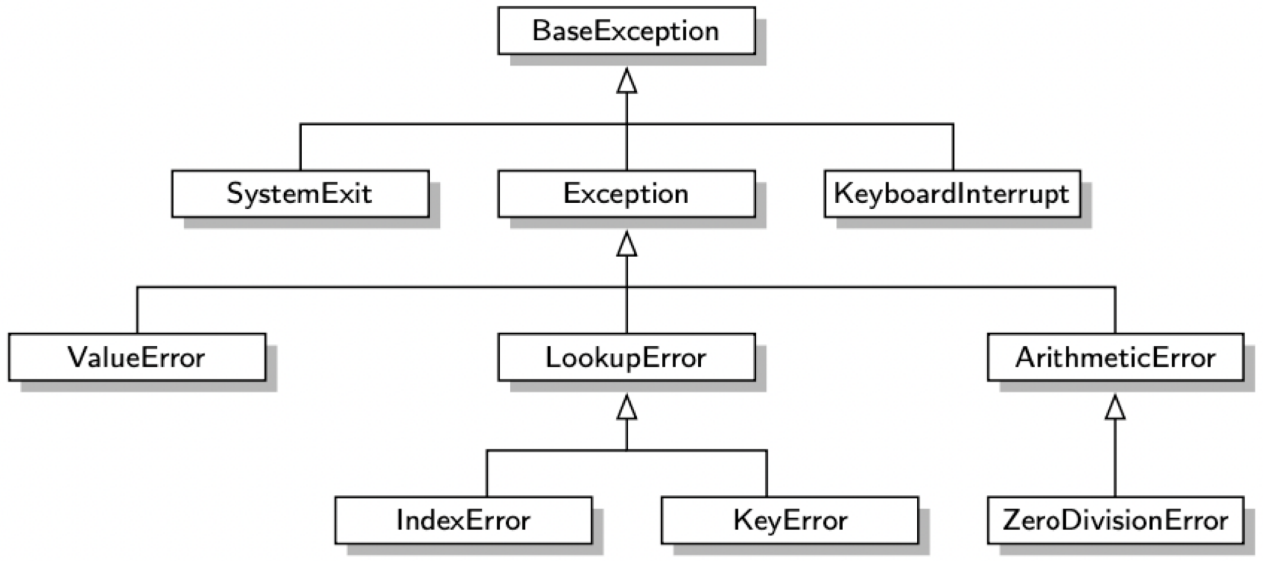
- What if the Animal class was not able to satisfy our requirements?
    - We want something more specific

- The OOP principle of ‘Inheritance’:
    - Allows us to avoid starting over
    - Build off of existing code and modify it according to our needs

Inheritance

- a ***hierarchical*** fashion
- a level-by-level manner that goes from specific to more general

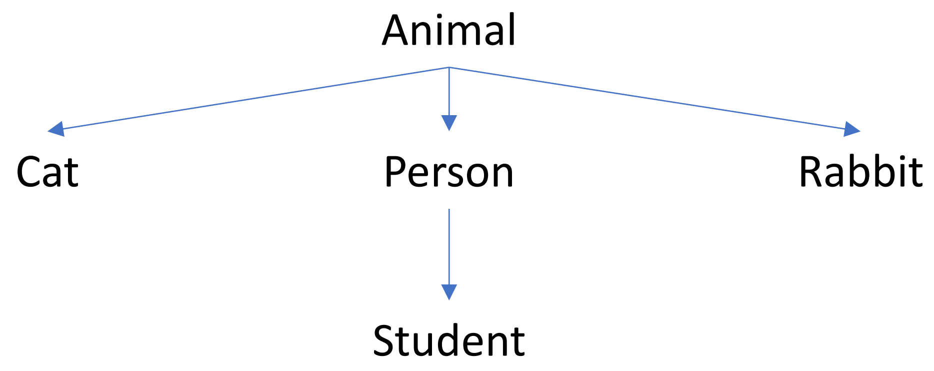




Example taken from "Data Structures and Algorithms in Python by Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser"

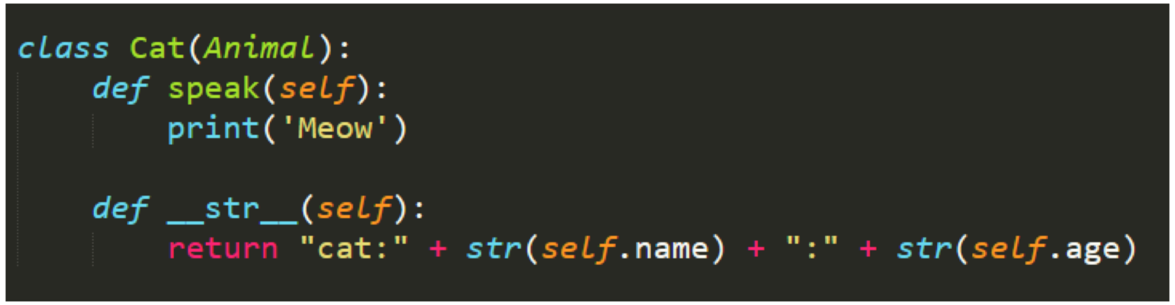
Inheritance

- Inheritance allows for creating type ‘hierarchies’, in which you can create many different related types that inherit different attributes and methods from their parents
  - Parent Class: Super class
  - Child Class: Subclass
    - By default inherits all attributes and methods
    - Can be modified to add more attributes and methods
    - Inherited methods can be overwritten



Cat class

- The following Class 'Cat' inherits all its attributes and methods from the Animal class

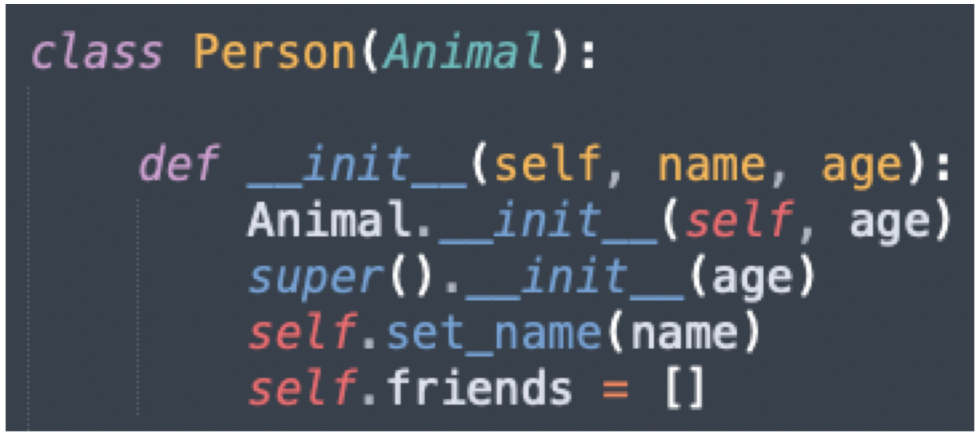


```
class Cat(Animal):
    def speak(self):
        print('Meow')

    def __str__(self):
        return "cat:" + str(self.name) + ":" + str(self.age)
```

- Has an additional methods called speak
- Overwrites the existing '__str__' method

Person class

- '__init__' using super().



```
class Person(Animal):

    def __init__(self, name, age):
        Animal.__init__(self, age)
        super().__init__(age)
        self.set_name(name)
        self.friends = []
```

- The mechanism for calling the inherited constructor relies on the syntax, super().

Access Control in Python

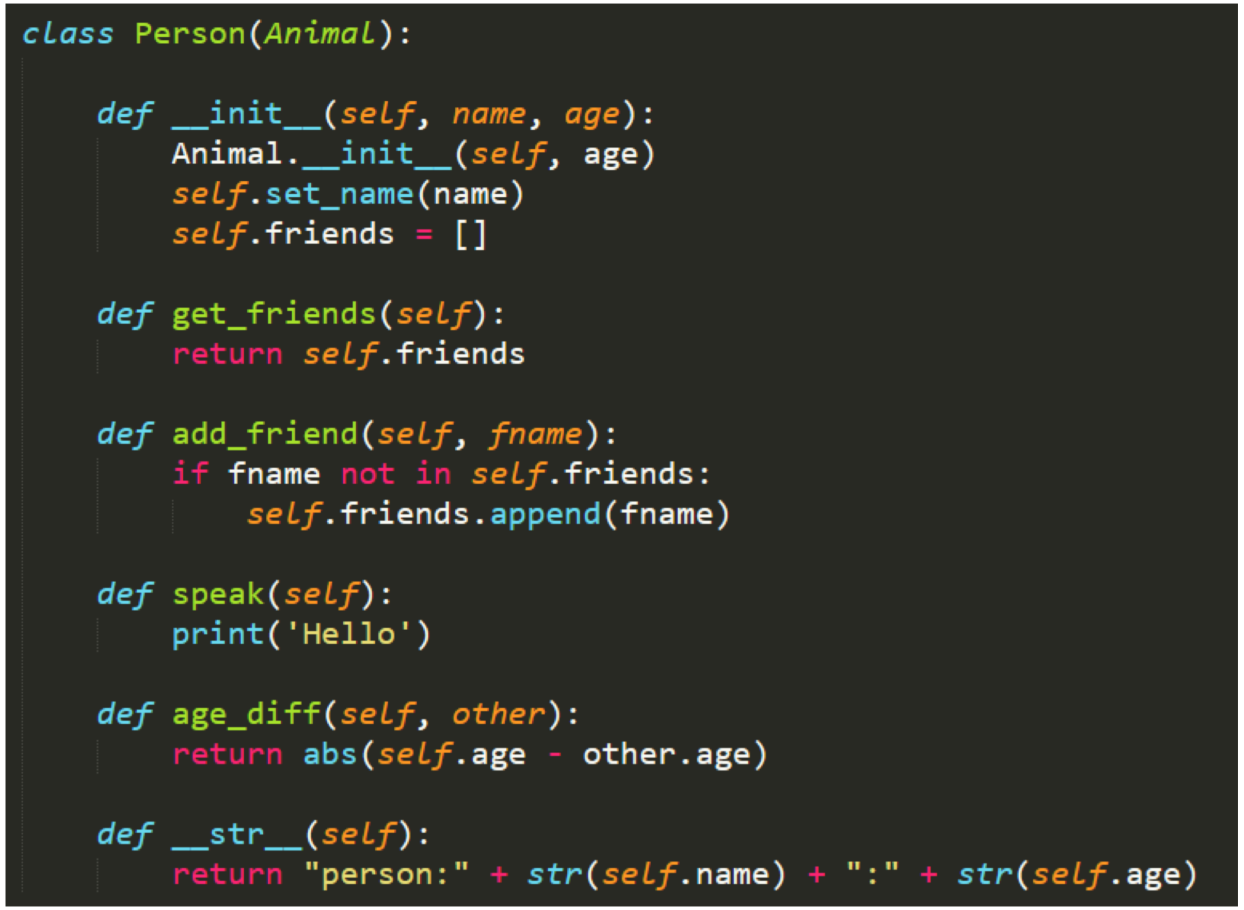
- Python does not support formal access control
- names beginning with a single underscore are conventionally akin to protected
- names beginning with a double underscore (other than special methods) are akin to private

- ***protected*** (e.g., Java, C++)
    - Members accessible to subclasses, but not to the general public
- private
    - Members that are declared as private are not accessible to either

Person class

- The Person class also inherits from the Animal class
  - Makes use of the existing '__init__' function
  - Has additional attributes and methods
  - Also modifies the '__str__' function

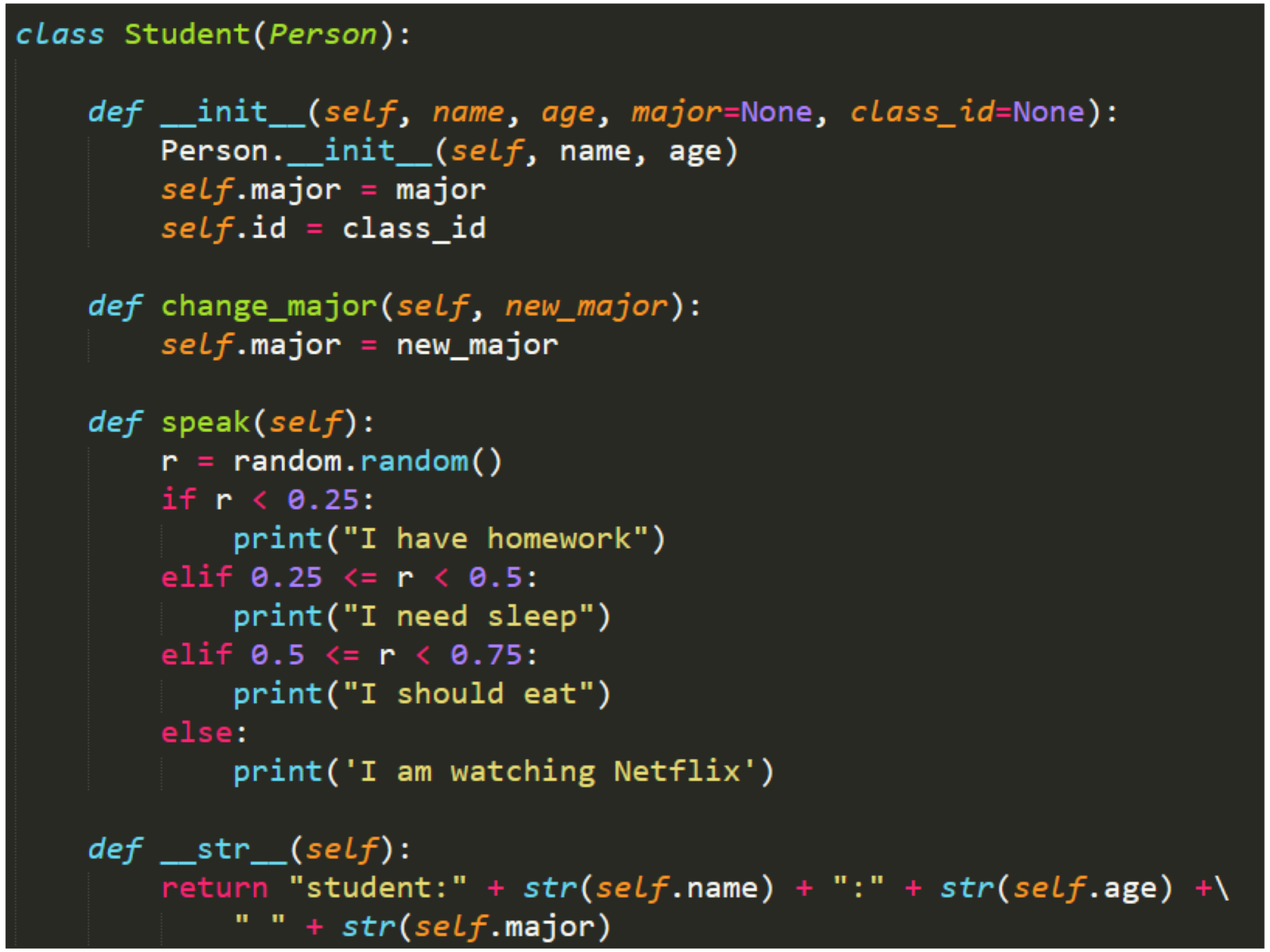
- Instead of explicitly specifying the parent class the 'super()' function can also be used



```python
class Person(Animal):

    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []

    def get_friends(self):
        return self.friends

    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)

    def speak(self):
        print('Hello')

    def age_diff(self, other):
        return abs(self.age - other.age)

    def __str__(self):
        return "person:" + str(self.name) + ":" + str(self.age)
```

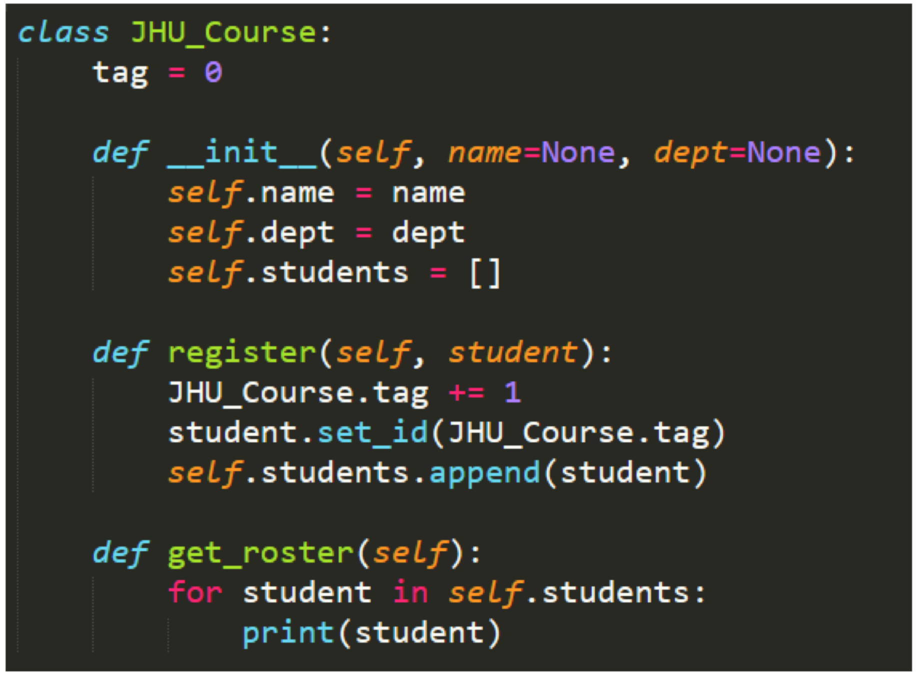
Student class

- The Student class inherits from the Person class
- You can also use Animal class functions as well, as the Person class inherits from it
- You can inherit from multiple different classes as well

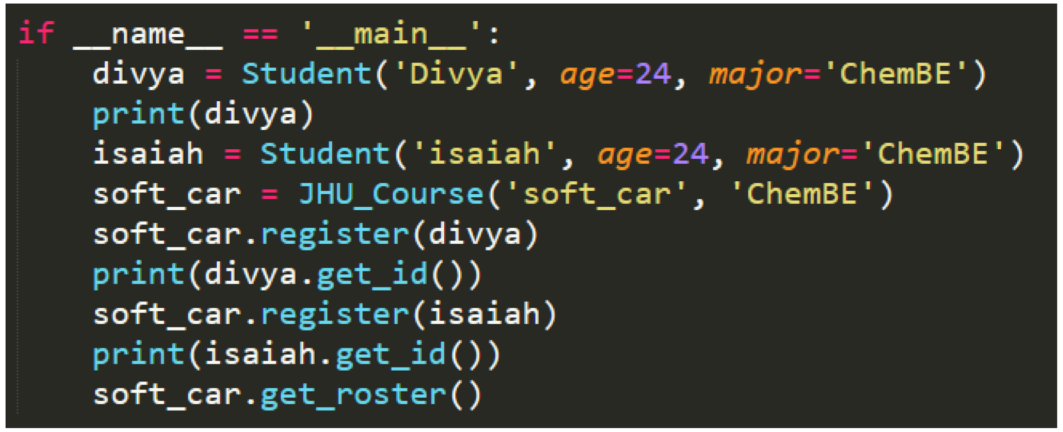


```python
class Student(Person):

    def __init__(self, name, age, major=None, class_id=None):
        Person.__init__(self, name, age)
        self.major = major
        self.id = class_id

    def change_major(self, new_major):
        self.major = new_major

    def speak(self):
        r = random.random()
        if r < 0.25:
            print("I have homework")
        elif 0.25 <= r < 0.5:
            print("I need sleep")
        elif 0.5 <= r < 0.75:
            print("I should eat")
        else:
            print('I am watching Netflix')

    def __str__(self):
        return "student:" + str(self.name) + ":" + str(self.age) +\
            " " + str(self.major)
```

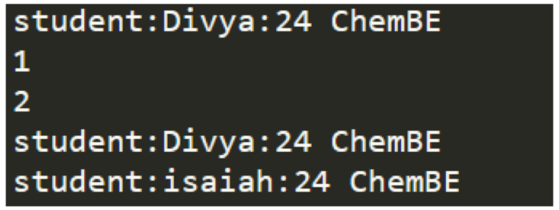
Class Variables



```python
class JHU_Course:
    tag = 0

    def __init__(self, name=None, dept=None):
        self.name = name
        self.dept = dept
        self.students = []

    def register(self, student):
        JHU_Course.tag += 1
        student.set_id(JHU_Course.tag)
        self.students.append(student)

    def get_roster(self):
        for student in self.students:
            print(student)
```



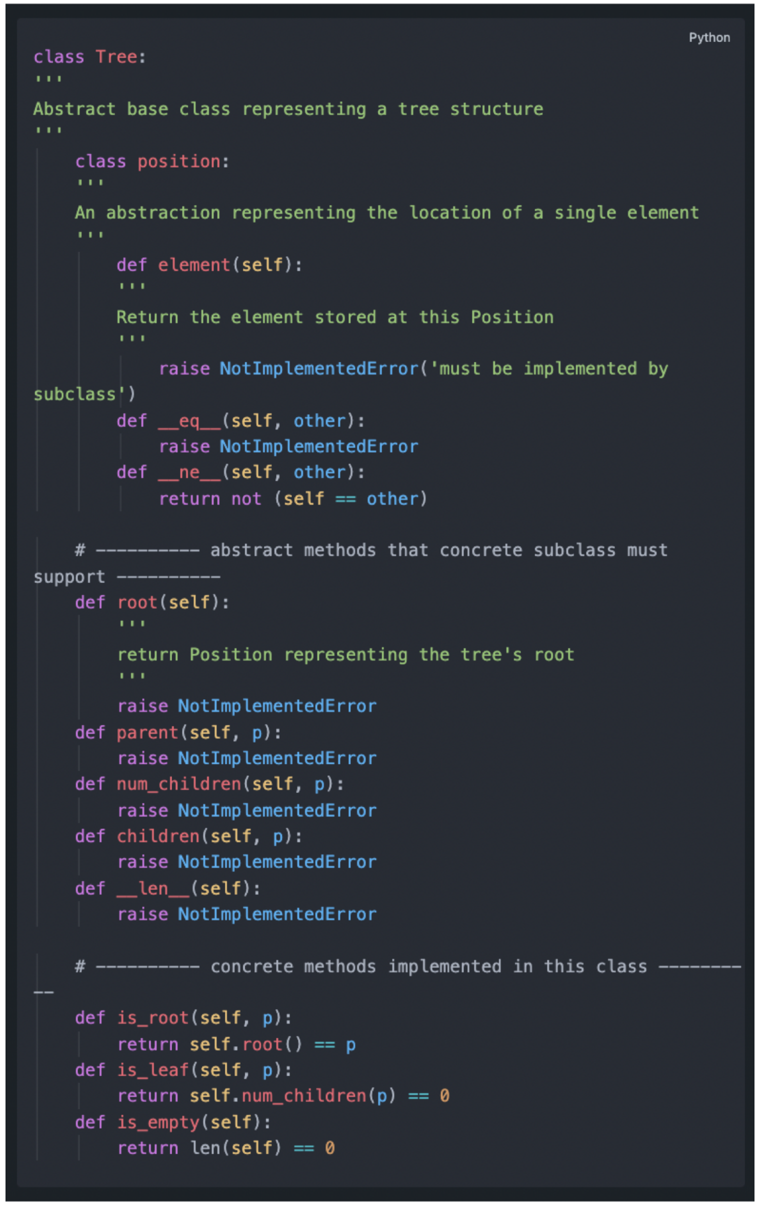
```python
if __name__ == '__main__':
    divya = Student('Divya', age=24, major='ChemBE')
    print(divya)
    isaiah = Student('isaiah', age=24, major='ChemBE')
    soft_car = JHU_Course('soft_car', 'ChemBE')
    soft_car.register(divya)
    print(divya.get_id())
    soft_car.register(isaiah)
    print(isaiah.get_id())
    soft_car.get_roster()
```



```
student:Divya:24 ChemBE
1
2
student:Divya:24 ChemBE
student:isaiah:24 ChemBE
```

Abstract Base Class (ABC)

- Python supports abstract base class (ABC).
- An abstract base class *cannot be instantiated* (i.e., you cannot directly create an instance of that class),
- but defines common methods that all implementations of the abstraction must have
- An ABC is realized by one or more *concrete classes that inherit from the abstract base class* while providing implementations for those method declared by the ABC



```python
class Tree:
'''
Abstract base class representing a tree structure
'''
    class position:
    '''
    An abstraction representing the location of a single element
    '''
        def element(self):
        '''
        Return the element stored at this Position
        '''
            raise NotImplementedError('must be implemented by
subclass')
        def __eq__(self, other):
            raise NotImplementedError
        def __ne__(self, other):
            return not (self == other)

    # ---------- abstract methods that concrete subclass must
support ----------
    def root(self):
        '''
        return Position representing the tree's root
        '''
        raise NotImplementedError
    def parent(self, p):
        raise NotImplementedError
    def num_children(self, p):
        raise NotImplementedError
    def children(self, p):
        raise NotImplementedError
    def __len__(self):
        raise NotImplementedError

    # ---------- concrete methods implemented in this class --------
--
    def is_root(self, p):
        return self.root() == p
    def is_leaf(self, p):
        return self.num_children(p) == 0
    def is_empty(self):
        return len(self) == 0
```