《人工智能与Python程序设计》— 面向对象编程（二）

人工智能与Python程序设计 教研组

人工智能与
Python程序设计

回顾

1. 类和实例
2. 数据封装
3. 访问限制

Python 面向对象编程

- 面向对象编程（Object Oriented Programming， OOP）是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了**数据**和操作数据的**函数（方法）**
  - 面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。
- 在Python中，**所有数据类型**都可以视为对象
- 也可以自定义对象。自定义的对象数据类型就是面向对象编程中的类（class）的概念。
  - 通常使用class语句自定义类

类和实例

- 类和实例是OOP中最为重要的概念
  - 从求解问题的角度，类是实例的**抽象**
    - Michael，Kristen都是Student
  - 从程序实现的角度，类是实例的**模板**
    - 通过__init__函数绑定实例共有的属性
    - 通过定义方法，定义实例共有的功能

```
class Student(object):

    def __init__(self, name, score):
        # 类的构造函数，  self代表类的实例
        self.name  = name
        self.score = score

    def print_score(self):
        # 类的方法
        print('%s: %d' % (self.name, self.score))
```

数据封装

- OOP 的一个重要特点就是数据封装。
  - 可以通过在类的内部定义函数来访问实例的数据
  - 例如：
    - Student类中，每个实例拥有各自的name和score数据
    - 在类内部定义print_score函数，打印某个学生的成绩
    - 封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法。

```
def print_score(std):
    print('%s: %d' % (std.name, std.score))

print_score(Michael)
```

```
Michael: 98
```

```
class Student(object):

    def __init__(self, name, score):
        # 类的构造函数，  self代表类的实例
        self.name  = name
        self.score = score

    def print_score(self):
        # 类的方法
        print('%s: %d' % (self.name, self.score))
```

数据封装

- 要定义一个方法：
  - 在class语句引导的代码块中定义一个函数
  - 除了第一个参数是self外，其他和普通函数一样

```
class Student(object):

    def __init__(self, name, score):
        # 类的构造函数，  self代表类的实例
        self.name  = name
        self.score = score

    def print_score(self):
        # 类的方法
        print('%s: %d' % (self.name, self.score))

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

数据封装

- 要调用一个方法，只需要在实例变量上直接调用即可
  - 除了self不用传递，其他参数正常传入
  - 从调用方来看Student 类
    - 只需在创建实例时给定name和score
    - print_score会在Student 类的内部实现
    - 这些数据和逻辑被『封装』起来，调用会变得容易。

```
Michael = Student('Michael Simon', 98)
Michael.print_score()
```

```
Michael Simon: 98
```

访问限制

- 在class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，从而隐藏了内部的复杂逻辑。
  - 但默认情况下，属性是公有（public）的，外部代码可以修改一个实例的属性

```
Michael = Student('Michael Simon', 98)
Michael.print_score()
Michael.score=80
Michael.print_score()
```

```
Michael Simon: 98
Michael Simon: 80
```

访问限制

- 在class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，从而隐藏了内部的复杂逻辑。
  - 为了让让内部属性不被外部访问，可以把属性的名称前加上两个下划线__，使其变为一个私有变量(private)
  - 私有变量只有内部可以访问，外部不能访问；

```
class Student(object):

    def __init__(self, name, score):
        self.__name  = name   # 私有变量
        self.__score = score  # 私有变量

    def print_score(self):
        print('%s: %d' % (self.__name, self.__score))
```

```
Michael = Student('Michael', 98) # 创建实例对象
Michael.print_score()
print(Michael.__score)
```

```
Michael: 98

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-26-bb01d8fad8b6> in <module>
      1 Michael = Student('Michael', 98) # 创建实例对象
      2 Michael.print_score()
----> 3 print(Michael.__score)

AttributeError: 'Student' object has no attribute '__score'
```

访问限制

- 通过引入私有变量，可以确保外部代码不能随意修改对象内部的状态。即，通过访问限制的保护，使得代码更加健壮。
- 如果外部代码想访问、修改私有变量，可通过增加类的方法进行实现。

```
class Student(object):
    ...

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score

    def set_score(self, score):
        self.__score = score
```

访问限制

那么，类的方法会比直接通过外部访问/修改有什么优势呢？

```
class Student(object):
    ...

    def set_score(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

在类的方法中，可以对参数做相关的检查，避免出现异常错误。

访问限制

- 注意：
  - 类似于__xxx的变量是private变量，外部代码不能直接访问。
  - 类似于_xxx的变量允许外部代码访问，但按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但请把我视为private 变量，不要随意访问”
  - 类似于__xxx__的变量是特殊变量，不是private 变量，外部代码可直接访问。

访问限制

- 再回来分析这个示例：

```
Michael = Student('Michael Simon', 98)
Michael.print_score()
Michael.__score=80
Michael.print_score()
```

```
Michael Simon: 98
Michael Simon: 98
```

```
print(Michael._Student__score)
print(Michael.__score)
```

```
98
80
```

内部的__score变量已经被Python解释器自动改成了_Student__score，
外部代码给Michael实例新增了一个__score变量

人工智能与
Python程序设计

提纲

1. 实例属性和类属性
2. 继承和多态
3. 获取对象信息
4. 学生信息类示例

实例属性与类属性

- 由于Python是动态语言，根据类创建的实例可以任意绑定属性。
  - 给实例绑定属性的方法是通过实例变量，或者通过self变量

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

s = Student('Bob')
s.score = 90
```

实例属性与类属性

- 由于Python是动态语言，根据类创建的实例可以任意绑定属性。
  - 给实例绑定属性的方法是通过实例变量，或者通过self变量
- 给类本身绑定属性，可以直接在class中定义属性，称为类属性，归类所有

```
class Student(object):
    stu_num = 396

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

实例属性与类属性

- 由于Python是动态语言，根据类创建的实例可以任意绑定属性。
  - 给实例绑定属性的方法是通过实例变量，或者通过self变量
- 给类本身绑定属性，可以直接在class中定义属性，称为类属性，归类所有

```
class Student(object):
    stu_num = 396

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

s = Student('Bob')
print(s.name)
print(s.stu_num)
```

```
Bob
396
```

定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。

实例属性与类属性

- 注意：
  - 在编写程序的时候，不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性

```
class Student(object):
    name = 'student'

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

s = Student('Bob')
print(s.name)
```

```
Bob
```

继承与多态

- 在OOP 中定义一个class 时，可以从某个现有的class继承，新的class 称为子类(subclass)，而被继承的class 称为基类、父类或超类(base class，super class)。
- 继承：一个派生类(derived class)继承基类的字段和方法，即子类获得了父类的全部功能。
- 举例：Undergraduate类继承Student类，Dog类继承Animal类

```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

继承与多态

- 对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。
- 由于Animal实现了run()方法，Dog和Cat也拥有了run()方法。

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

```
Animal is running...
Animal is running...
```

继承与多态

- 在继承时，也可对子类增添新的方法，如

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')
```

```
dog = Dog()
dog.run()
```

```
Dog is running...
```

子类的run()覆盖了父类的run()方法，即多态性

添加了新的子类方法

继承与多态

- 多态：为不同数据类型的实体提供统一的接口。
  - 简单而言：相同的消息给予不同的对象会引发不同的动作，如前面的Animal类和Dog类在处理run()方法上动作的差异。

继承与多态

- 进一步理解多态
  - 定义一个class，实际上就定义了一种新的数据类型，类似于Python内置数据类型

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

继承与多态

- 进一步理解多态
  - 定义一个class，实际上就定义了一种新的数据类型，类似于Python内置数据类型
  - 判断一个变量是否是某个类型可以用isinstance()：

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

```
print(isinstance(a, list))
print(isinstance(b, Animal))
print(isinstance(c, Dog))
print(isinstance(c, Animal))
```

```
True
True
True
True
```

继承与多态

- 进一步理解多态
  - 定义一个class，实际上就定义了一种新的数据类型，类似于Python内置数据类型
  - 判断一个变量是否是某个类型可以用isinstance()：

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

```
print(isinstance(a, list))
print(isinstance(b, Animal))
print(isinstance(c, Dog))
print(isinstance(c, Animal))
```

```
True
True
True
True
```

a, b, c分别是list, Animal, Dog类型，但c还是Animal类型

继承与多态

- 进一步理解多态
  - 定义一个class，实际上就定义了一种新的数据类型，类似于Python内置数据类型
  - 判断一个变量是否是某个类型可以用isinstance()：

```
print(isinstance(b, Dog))
```

```
False
```

在继承关系中，如果一个实例的数据类型是某个子类，则它的数据类型也可看做是其父类，但反过来不行

继承与多态

- 进一步理解多态
  - 再来看一个例子

```
def run_animal(animal):
    animal.run()

run_animal(Animal())
run_animal(Dog())
run_animal(Cat())
```

```
Animal is running...
Dog is running...
Cat is running...
```

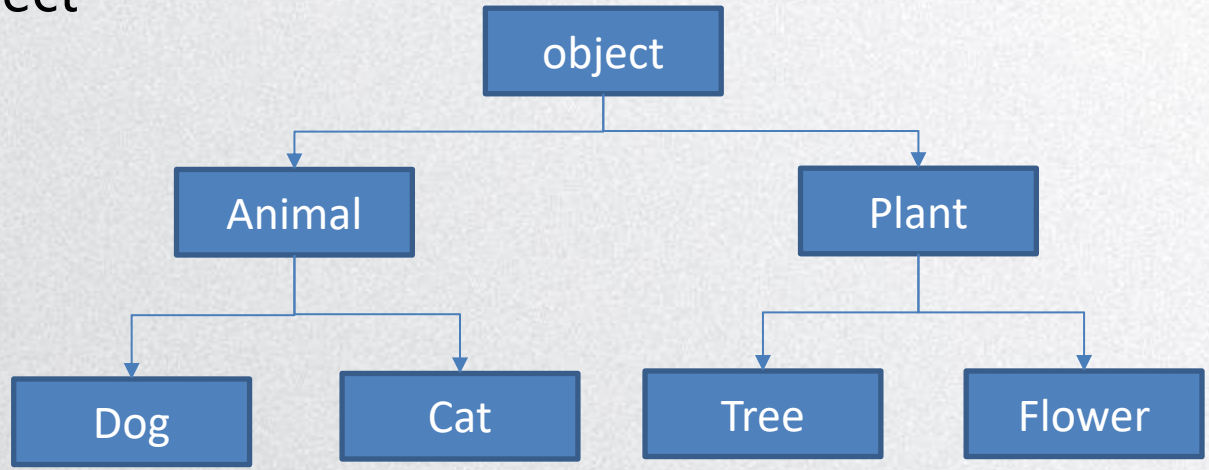
对于一个变量，只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal，Dog还是Cat对象上，由运行时该对象的确切类型决定。

继承与多态

- 进一步理解多态
  - 调用方只管调用，不管细节．当新增一种Animal 的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。
  - 这就是著名的“开闭”原则(对扩展开放，对修改关闭)。

继承与多态

- 继承可以一级一级地继承下来，而任何类最终都可以追溯到根类object



继承属性?

- 如何让子类具备父类的属性?

```
class Student(object):

    def __init__(self, name, score):
        self.__name  = name
        self.__score = score

    def print_score(self):
        print('%s: %d' % (self.__name, self.__score))

class Undergraduate(Student):
    pass
```

```
x = Undergraduate()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/4m/kw1nckdn0qzcz68fpxyc8yj80000gn/T/ipykernel_31570/2082171742.py in <module>
----> 1 x = Undergraduate()

TypeError: __init__() missing 2 required positional arguments: 'name' and 'score'
```

```
x = Undergraduate("ZhangSan", 90)
x.print_score()
```

```
ZhangSan: 90
```

  - 子类会通过继承父类__init__方法，自动获取父类的属性!

继承属性?

- 如何让子类具有更多的属性?
  - 覆盖父类的__init__方法

```
class Undergraduate(Student):

    def __init__(self, major):
        self.major = major
```

```
x = Undergraduate("AI")
print(x.major)
x.print_score()
```

```
AI
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/var/folders/4m/kw1nckdn0qzcz68fpxyc8yj80000gn/T/ipykernel_31570/901582988.py in <module>
      1 x = Undergraduate("AI")
      2 print(x.major)
----> 3 x.print_score()

/var/folders/4m/kw1nckdn0qzcz68fpxyc8yj80000gn/T/ipykernel_31570/433064877.py in print_score(self)
      6
      7     def print_score(self):
----> 8         print('%s: %d' % (self.__name, self.__score))
      9
     10 class Undergraduate(Student):

AttributeError: 'Undergraduate' object has no attribute '_Student__name'
```

继承属性?

- 如何让子类具有更多的属性?
  - 覆盖父类的__init__方法的同时，要调用父类的__init__方法，以保证子类实例具备父类实例应该具有的属性
  - 使用super()函数

```
class Undergraduate(Student):

    def __init__(self, name, score, major):
        super(Undergraduate, self).__init__(name, score)
        self.major = major
```

```
x = Undergraduate("LiSi", 95, "AI")
print(x.major)
x.print_score()
```

```
AI
LiSi: 95
```

人工智能与
Python程序设计

提纲

1. 实例属性和类属性
2. 继承和多态
3. 获取对象信息
4. 学生信息类示例

获取对象信息

- 获取对象类型与方法
  - 判断对象类型，使用type()函数

```
print(type(123))
print(type('123'))
print(type([1,2,3]))

class Animal(object):
    def run(self):
        print('Animal is running ...')

a = Animal()
print(type(a))
```

```
<class 'int'>
<class 'str'>
<class 'list'>
<class '__main__.Animal'>
```

获取对象信息

- 获取对象类型与方法
  - 判断对象类型，使用type()函数

```
print(type(123))
print(type('123'))
print(type([1,2,3]))

class Animal(object):
    def run(self):
        print('Animal is running ...')

a = Animal()
print(type(a))
```

```
<class 'int'>
<class 'str'>
<class 'list'>
<class '__main__.Animal'>
```

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')

b = Dog()
print(type(b))
```

```
<class '__main__.Dog'>
```

父类与子类的class差异

获取对象信息

- 获取对象类型与方法
  - 判断对象类型，使用type()函数
  - 针对class的继承关系，使用isinstance()函数判断class类型。

  例如继承关系：object -> Animal -> Dog -> Husky

```
a = Animal()
d = Dog()
h = Husky()
```

```
print(isinstance(h, Husky))
print(isinstance(h, Dog))
print(isinstance(h, Animal))
```

```
True
True
True
```

```
print(isinstance(d, Husky))
print(isinstance(h, Dog))
print(isinstance(d, Animal))
```

```
False
True
True
```

获取对象信息

- 获取对象类型与方法
  - 判断对象类型，使用type()函数
  - 针对class的继承关系，使用isinstance()函数判断class类型。
  - 使用dir()来获得一个对象的所有属性和方法，并以一个字符串列表返回

```
a = Animal()
d = Dog()
h = Husky()
```

```
print(dir(d))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq
__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash_
_', '__init__', '__init_subclass__', '__le__', '__lt__', '__module_
_', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakr
ef__', 'eat', 'run']
```

```
print(d.__class__)
```

```
<class '__main__.Dog'>
```

获取对象信息

- 获取对象类型与方法
  - 判断对象类型，使用type()函数
  - 针对class的继承关系，使用isinstance()函数判断class类型。
  - 使用dir()来获得一个对象的所有属性和方法，并以一个字符串列表返回

```
a = Animal()
d = Dog()
h = Husky()
```

```
print(dir(d))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq
__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash_
_', '__init__', '__init_subclass__', '__le__', '__lt__', '__module_
_', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakr
ef__', 'eat', 'run']
```

```
print(d.__class__)
```

```
<class '__main__.Dog'>
```

类似__xxx__的属性和方法在Python中都是有特殊用途的，比如__class__方法返回所属类。

获取对象信息

- 获取对象类型与方法
  - 判断对象类型，使用type()函数
  - 针对class的继承关系，使用isinstance()函数判断class类型。
  - 使用dir()来获得一个对象的所有属性和方法，并以一个字符串列表返回

```
a = Animal()
d = Dog()
h = Husky()
```

```
print(dir(d))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq
__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash_
_', '__init__', '__init_subclass__', '__le__', '__lt__', '__module_
_', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakr
ef__', 'eat', 'run']
```

```
print(d.__class__)
```

```
<class '__main__.Dog'>
```

dir()也会返回类内的方法

获取对象信息

- 获取对象类型与方法
  - 判断对象类型，使用type()函数
  - 针对class的继承关系，使用isinstance()函数判断class类型。
  - 使用dir()来获得一个对象的所有属性和方法，并以一个字符串列表返回

dir()亦可用于Python内置数据类型

获取对象信息

- 获取对象类型与方法
  - 判断对象类型，使用type()函数
  - 针对class的继承关系，使用isinstance()函数判断class类型。
  - 使用dir()来获得一个对象的所有属性和方法，并以一个字符串列表返回
  - 仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，可直接操作一个对象的状态

```
class Animal(object):
    def __init__(self):
        self.able_to_move = True
    def run(self):
        print('Animal is running...')
```

```
a = Animal()
print(hasattr(a, 'able_to_move'))
setattr(a, 'able_to_move', False)
print(getattr(a, 'able_to_move'))
```

```
True
False
```

获取对象信息

- 注意：
  - 通过内置的一系列函数，我们可以对任意一个Python对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。

如果可以直接写

```
sum = obj.x + obj.y
```

就不要写

```
sum = getattr(obj, 'x') + getattr(obj, 'y')
```

人工智能与
Python程序设计

提纲

1. 实例属性和类属性
2. 继承和多态
3. 获取对象信息
4. 学生类示例

学生信息类示例

- 学生信息类：
  - 学生信息：姓名，家乡，电话，课程及成绩
  - 学生信息的输入，获取
  - 学生课程成绩的评价
  - 学生信息的输出

```
class Student(object):
    # 学生信息类
    def __init__(self, name, hometown='None', phone='None'):...

    def input_hometown(self, hometown):...

    def input_phone(self, phone):...

    def input_course_score(self, course, score):...

    def get_name(self):...

    def get_phone(self):...

    def get_hometown(self):...

    def get_course_grade(self, course):...

    def print_student_info(self):...
```

学生信息类示例

- 学生信息类：
  - 学生信息类的初始化（构造函数）

```
def __init__(self, name, hometown='None', phone='None'):
    '''
    学生信息类的构造函数
    参数self: 创建的实例本身
    参数name: 学生姓名
    参数hometown: 学生家乡
    参数phone: 学生电话
    '''
    self.__name = name
    self.__hometown = hometown
    self.__phone = str(phone)
    self.__courses = {}     #课程字典：课程名->课程成绩
```

学生信息类示例

- 学生信息类：
  - 类的方法：学生信息的输入

```
def input_hometown(self, hometown):
    '''
    类的方法：输入学生实例的家乡
    '''
    if self.__hometown != 'None':
        print('Warning: The existing hometown {0} is being changed into {1}.'.format(self.__hometown, hometown))
    self.__hometown = hometown

def input_phone(self, phone):
    '''
    类的方法：输入学生实例的电话
    '''
    if self.__phone != 'None':
        print('Warning: The existing phone {0} is being changed into {1}.'.format(self.__phone, phone))
    self.__phone = str(phone)

def input_course_score(self, course, score):
    '''
    类的方法：输入学生实例的指定课程的成绩
    '''
    if score >=0 and score <= 100:
        if self.__courses.get(course, -1) != -1: #如课程名不在字典key中，则返回-1，判断为假
            print('Warning: The existing {0} grade {1:.2f} is being changed into {2:.2f}.'.format(course, self.__courses.get(course, -1), score))
        self.__courses[course] = score
    else:
        print("Please input one valid course score!")
```

学生信息类示例

- 学生信息类：
  - 类的方法：学生信息的获取

```
def get_name(self):
    '''
    类的方法：获取学生实例的姓名
    '''
    return self.__name

def get_phone(self):
    '''
    类的方法：获取学生实例的电话
    '''
    return self.__phone

def get_hometown(self):
    '''
    类的方法：获取学生实例的家乡
    '''
    return self.__hometown
```

学生信息类示例

- 学生信息类：
  - 类的方法：学生课程成绩的评价

```
def get_course_grade(self, course):
    '''
    类的方法：获取学生实例的指定课程的评级
    '''
    try:
        if self.__courses[course] >= 90:
            return 'A'
        elif self.__courses[course] >= 60:
            return 'B'
        else:
            return 'C'
    except:
        print("Course {0} does exist".format(course))
```

学生信息类示例

- 学生信息类：
  - 类的方法：学生信息的输出

```
def print_student_info(self):
    '''
    类的方法：获取学生实例的全部信息
    '''
    print('Name: {0}'.format(self.__name))
    print('Hometown: {0}'.format(self.__hometown))
    print('Phone: {0}'.format(self.__phone))
    if len(self.__courses) > 0:
        for course, score in self.__courses.items():
            print('Course {0}: {1:.2f}'.format(course, score))
```

学生信息类示例

- 学生信息类：
  - 类的实例化（范例一）

```
ZhangSan = Student(name='ZhangSan', hometown='TianJin', phone='18623443212')
ZhangSan.input_course_score('Algebra', 90)
ZhangSan.input_course_score('Python', 95)

print('ZhangSan Phone number is {0}'.format(ZhangSan.get_phone()))
print('ZhangSan {0} grade is {1}'.format('Python', ZhangSan.get_course_grade('Python')))
print('ZhangSan {0} grade is {1}'.format('English', ZhangSan.get_course_grade('English')))
ZhangSan.print_student_info()
```

```
ZhangSan Phone number is 18623443212
ZhangSan Python grade is A
Course English does exist
ZhangSan English grade is None
Name: ZhangSan
Hometown: TianJin
Phone: 18623443212
Course Algebra: 90.00
Course Python: 95.00
```

学生信息类示例

- 学生信息类：
  - 类的实例化（范例二）

```
LiSi = Student(name='LiSi', hometown='WuHan')
print('LiSi Phone number is {0}'.format(LiSi.get_phone()))
LiSi.input_phone(1827324834)
print('LiSi Phone number is {0}'.format(LiSi.get_phone()))
```

```
LiSi Phone number is None
LiSi Phone number is 1827324834
Name: LiSi
Hometown: WuHan
Phone: 1827324834
```

学生信息类示例

- 研究生信息类：
  - 继承学生信息类
  - 添加额外的本科专业信息和学术文章发表信息

```python
class Graduate(Student):
    # 研究生信息类
    def __init__(self, name, hometown='None', phone='None'):
        '''
        研究生信息类的构造函数，继承自学生信息类（父类）
        参数self：创建的实例本身
        参数name：学生姓名
        参数hometown：学生家乡
        参数phone：学生电话
        '''
        super(Graduate, self).__init__(name, hometown, phone)  # 运行父类的构造函数
        self.__bachelor_major = 'None'
        self.__publication = []
```

学生信息类示例

- 研究生信息类：
  - 类的方法：学生发表文章和本科专业信息的输入

```
def input_paper(self, papers):
    '''
    类的方法：输入学生实例发表的文章
    '''
    if isinstance(papers, list):
        self.__publication = self.__publication + papers
    elif isinstance(papers, str):
        self.__publication.append(papers)
    else:
        print("Please organize the papers in list or str type!")

def input_bachelor_major(self, major):
    '''
    类的方法：输入学生实例的本科专业
    '''
    self.__bachelor_major = major
```

学生信息类示例

- 研究生信息类：
  - 类的方法：学生发表文章和本科专业信息的获取

```
def get_publication(self):
    '''
    类的方法：获取学生实例的所有发表文章信息
    '''
    return self.__publication

def get_bachelor_major(self):
    '''
    类的方法：获取学生实例的本科专业
    '''
    return self.__bachelor_major
```

学生信息类示例

- 研究生信息类：
  - 类的方法：研究生课程成绩的评价，覆盖学生类（父类）的课程评价

```
def get_course_grade(self, course):
    '''
    类的方法：获取学生实例的课程成绩评级 （覆盖父类对应方法）
    '''
    try:
        if self._Student__courses[course] >= 85:
            return 'A'
        elif self._Student__courses[course] >= 65:
            return 'B'
        else:
            return 'C'
    except:
        print("Course {0} does exist".format(course))
```

学生信息类示例

- 研究生信息类：
  - 类的方法：研究生信息的输出，覆盖学生类（父类）的信息输出方法

```
def print_student_info(self):
    '''
    类的方法：获取学生实例的全部信息（覆盖父类对应方法）
    '''
    print('Name: {0}'.format(self._Student__name))
    print('Hometown: {0}'.format(self._Student__hometown))
    print('Phone: {0}'.format(self._Student__phone))
    if len(self._Student__courses) > 0:
        for course, score in self._Student__courses.items():
            print('Course {0}: {1:.2f}'.format(course, score))
    if len(self.__publication) > 0:
        print('Publication:')
        for paper in self.__publication:
            print('{0:11} {1}'.format('', paper))
```

学生信息类示例

- 研究生信息类：
  - 类的实例化

```
WangWu = Graduate(name='WangWu', hometown='Xi\'an')
WangWu.input_course_score('Computer Vision', 86)
WangWu.input_course_score('Machine Learning ', 93)
print('WangWu {0} grade is {1}'.format('Computer Vision', WangWu.get_course_grade('Computer Vision')))
WangWu.input_paper('WangWu et al. Paper_one')
WangWu.input_paper(['WangWu et al. Paper_two','WangWu et al. Paper_three'])
WangWu.print_student_info()
```

```
WangWu Computer Vision grade is A
Name: WangWu
Hometown: Xi'an
Phone: None
Course Computer Vision: 86.00
Course Machine Learning : 93.00
Publication:
            WangWu et al. Paper_one
            WangWu et al. Paper_two
            WangWu et al. Paper_three
```

人工智能与
Python程序设计

回顾

1. 实例属性和类属性
2. 继承和多态
3. 获取对象信息
4. 学生类示例

谢谢！