

python面向对象编程

类和对象（与c++类似，基础不赘述）

```
class Dog:
    def __init__(self, name, age):
        self.name=name
        self.age=age
    def sit(self):
        print(f"{self.name} is now sitting.")
    def roll_over(self):
        print(f"{self.name} rolled over!")
'''创建实例'''
my_dog=Dog('willie',6)
print(my_dog.name)
print(my_dog.age)
my_dog.sit()
my_dog.roll_over()
```

```
"E:\OneDrive - cumt.edu.cn\program\python\pythonN+\Scripts\python.exe
willie
6
willie is now sitting.
willie rolled over!

进程已结束,退出代码0
```

```
class Car:
    def __init__(self, make, model, year):
        self.make=make
        self.model=model
        self.year=year
        self.odometer_reading=0
    def get_descriptive_name(self):
        long_name=f"{self.year}{self.make}{self.model}"
        return long_name.title()
    def update_odometer(self, mileage):
        if mileage>=self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")
my_new_car=Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
print(my_new_car.odometer_reading)
'''修改属性值'''
#直接修改属性
my_new_car.odometer_reading=1
print(my_new_car.odometer_reading)
```

```
#通过方法修改属性的值
my_new_car.update_odometer(2)
print(my_new_car.odometer_reading)
```

```
"E:\OneDrive - cumt.edu.cn\program\python\2019Audia4
0
1
2

进程已结束,退出代码0
```

访问限制

如果要想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`__`，在Python中，实例的变量名如果以`__`开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问。

练习

请把下面的 `Student` 对象的 `gender` 字段对外隐藏起来，用 `get_gender()` 和 `set_gender()` 代替，并检查参数有效性：

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

```
class Student(object):
    def __init__(self, name, gender):
        self.name = name
        self.__gender = gender
    def get_gender(self):
        return self.__gender
student1=Student('张三','女')
print(student1.name)
print(student1.get_gender())
```

```
"E:\OneDrive - cumt.edu.cn\program\python\python
张三
女

进程已结束,退出代码0
```

继承多态

继承

```
class Car:
    def __init__(self, make, model, year):
        self.make=make
        self.model=model
        self.year=year
        self.odometer_reading=0
    def get_descriptive_name(self):
        long_name=f"{self.year}{self.make}{self.model}"
        return long_name.title()
    def read_odometer(self):
        print(f"This car has{self.odometer_reading} miles on it.")
    def update_odometer(self, mileage):
        if mileage >=self.odometer_reading:
            self.odometer_reading=mileage
        else:
            print("You can't roll back an odometer")
    def increment_odometer(self, miles):
        self.odometer_reading+=miles
```

#创建子类时，父类必须包含在当前文件中，且位于子类前面。定义了子类**ElectricCar**。定义子类时，必须在圆括号内指定父类的名称。方法**__init__()**接受创建**Car**实例所需的信息。

```
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)
        self.battery_size=75#电瓶车特殊属性
    def describe_battery(self):
        print(f"This car has a {self.battery_size}-kWh battery.")
        #super() 是一个特殊函数，让你能够调用父类的方法。这行代码让Python调用Car 类的方法
        __init__()，让ElectricCar 实例包含这个方法中定义的所有属性。
my_tesla=ElectricCar('tesla', 'models', 2019)
print(my_tesla.get_descriptive_name())
```

```
"E:\OneDrive - cumt.edu.cn\program\python\pythonN-
2019Teslamodels

进程已结束,退出代码0
```

```
my_tesla.describe_battery()
```

```
"E:\OneDrive - cumt.edu.cn\program\python\pythonN-
2019Teslamodels

This car has a 75-kWh battery.

进程已结束,退出代码0
```

重写父类

重写父类方法python将忽略父类自身的方法，运行子类的方法

将实例用作属性

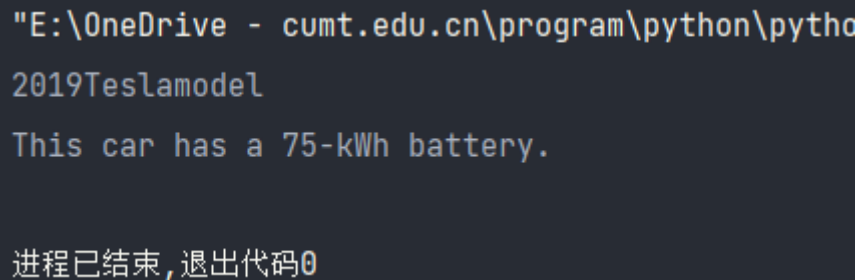
例：将ElectricCar类中的某些方法和属性提取出来，放到Battery的类中，将Battery实例作为ElectricCar类的属性：

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):
        long_name = f"{self.year}{self.make}{self.model}"
        return long_name.title()
    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")
    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer")
    def increment_odometer(self, miles):
        self.odometer_reading += miles

class Battery:
    def __init__(self, battery_size=75):
        self.battery_size = battery_size # 定义Battery
    def describe_battery(self):
        print(f"This car has a {self.battery_size}-kWh battery.")

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)
        self.battery = Battery() # 定义一个battery

my_tesla = ElectricCar('tesla', 'model', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery() # 访问tesla里面的battery中的方法
```



```
"E:\OneDrive - cumt.edu.cn\program\python\pytho
2019Tesla model
This car has a 75-kWh battery.
进程已结束,退出代码0
```

获取对象信息

基本类型都可以用 `type()` 判断

如果一个变量指向函数或者类，也可以用 `type()` 判断

`isinstance()` 函数

判断某对象是否属于某类型

```
isinstance(my_tesla,ElectricCar)
True
```

dir()函数

如果要获得一个对象的所有属性和方法，可以使用 `dir()` 函数，它返回一个包含字符串的list

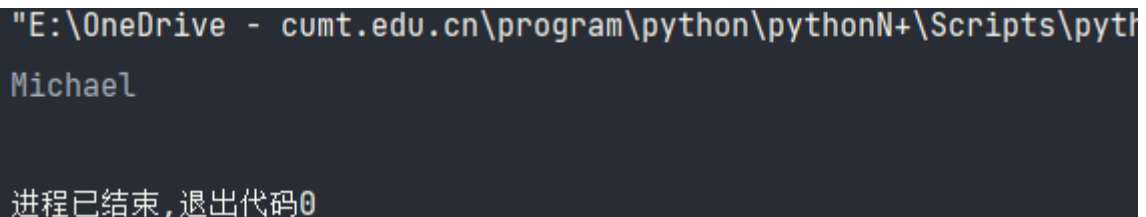
类似 `__xxx__` 的属性和方法在Python中都是有特殊用途的，比如 `__len__` 方法返回长度

面向对象高级编程

使用slots

当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性

```
from types import MethodType
class Student(object):
    def __init__(self, name='', age=0):
        self.name=name
        self.age=age
    def set_age(self, age):
        self.age=age
student1=Student()
student1.name='Michael'
print(student1.name)
student1.set_age=MethodType(set_age, student1) #给实例绑定一个方法
student1.set_age(25)
print(student1.age)
```



```
"E:\OneDrive - cumt.edu.cn\program\python\pythonN+\Scripts\pyth
Michael
进程已结束,退出代码0
```

只允许对Student实例添加 `name` 和 `age` 属性,为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的 `__slots__` 变量，来限制该class实例能添加的属性。

```
class Student(object):
    def __init__(self, name='', age=0, score=0):
        self.name=name
        self.age=age
        self.score=score
    def set_age(self, age):
        self.age=age
    __slots__=('name', 'age')
student1=Student()
student1.name='Michael'
student1.age=25
student1.score=99
```

```
Traceback (most recent call last):
  File "E:\OneDrive - cumt.edu.cn\program\pythonN+\main.py", line 9, in <module>
    student1=Student()
  File "E:\OneDrive - cumt.edu.cn\program\pythonN+\main.py", line 5, in __init__
    self.score=score
AttributeError: 'Student' object has no attribute 'score'
```

使用@property

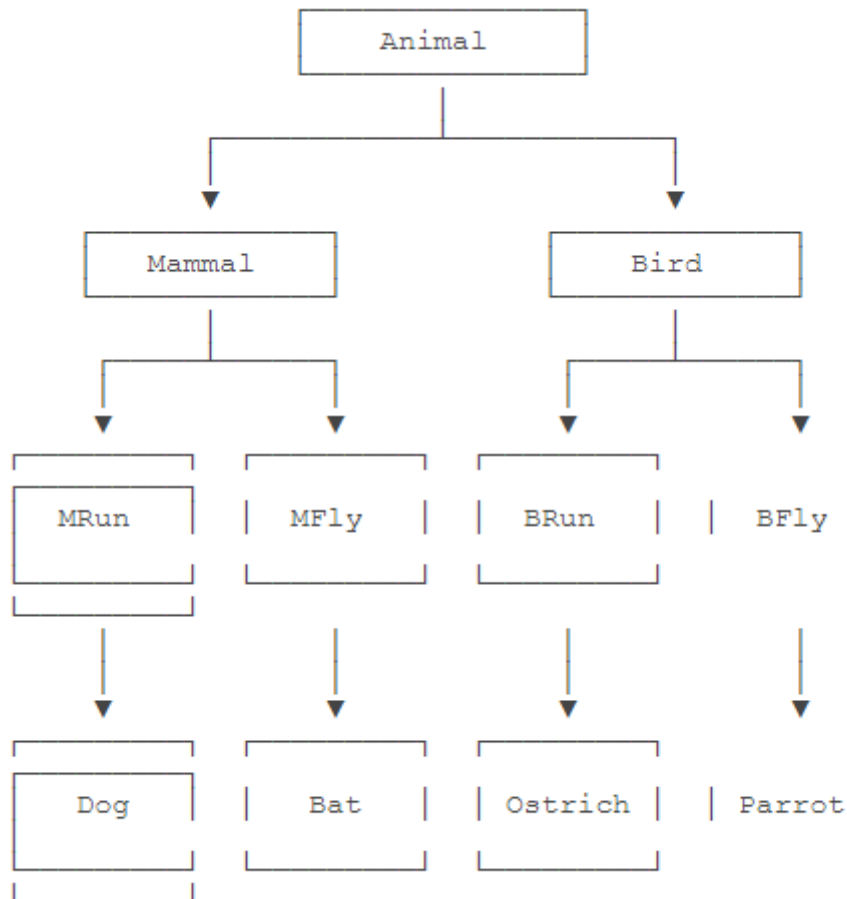
Python内置的@property装饰器是负责把一个方法变成属性调用

```
class Student(object):
    def __init__(self, name='', age=0, score=0, birth=0):
        self.name=name
        self.age=age
        self.score=score
        self.birth=birth
    def set_age(self, age):
        self.age=age
    @property
    def birth(self):
        return self.birth
    @birth.setter
    def birth(self, value):
        self.birth=value
    @property
    def age(self):
        return 2021-self.birth
```

birth 是可读写属性

age 就是一个只读属性

多重继承



```
class Animal(object):  
    .....  
class Mammal(Animal):  
    .....  
class Bird(Animal):  
    .....  
class Runnable(object):  
    .....  
class Flyable(object):  
    .....  
class Dog(Mammal,Runnable):  
    .....
```

Mixin

在设计类的继承关系时，通常，主线都是单一继承下来的，但是，如果需要“混入”额外的功能，通过多重继承就可以实现，这种设计通常称之为Mixin。

定制类

```
class Student(object):  
    def __init__(self,name):  
        self.name=name  
    def __str__(self):  
        return 'Student object (name: %s)'  
    def __iter__(self):  
        return self
```

枚举类

```
from enum import Enum
Month=Enum('Month',
('Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec'))
```

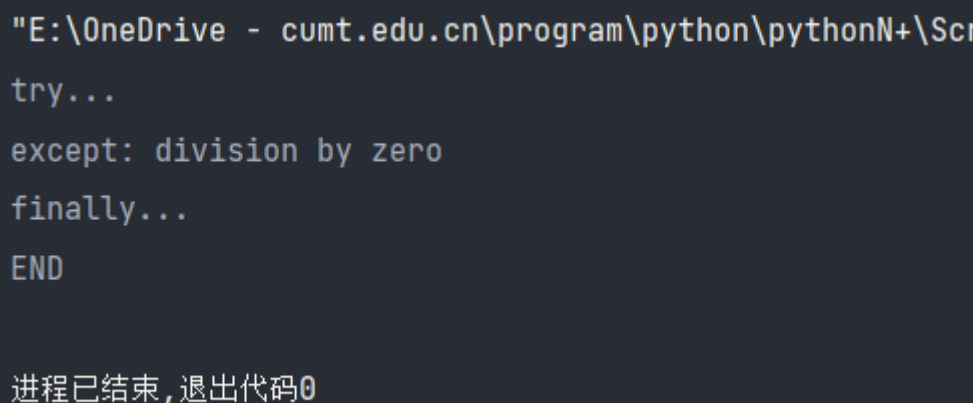
元类

metaclass:当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

错误处理

高级语言通常都内置了一套 `try...except...finally...` 的错误处理机制，Python也不例外。

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
    print('except:', e)
finally:
    print('finally...')
print('END')
```



```
"E:\OneDrive - cumt.edu.cn\program\python\pythonN+\Scr
try...
except: division by zero
finally...
END

进程已结束,退出代码0
```

调试

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE

Io编程

文件读写

要以读文件的模式打开一个文件对象，使用Python内置的 `open()` 函数，传入文件名和标示符

文件读取成功，调用 `read()` 方法可以一次读取文件的全部内容，Python把内容读到内存，用一个 `str` 对象表示。调用 `close()` 方法关闭文件。


```
f=open('111.txt','w')
print(f.read())
f.write('Hello,world!')
f.close()
```

名称	状态	修改
.idea	✓	2021
111.txt	✓	2021
main.py	✓	2021

*111.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

12345

```
main x
"E:\OneDrive - cumt.edu.cn\program\python\pythonN+\Scripts\python.exe"
12345

进程已结束,退出代码0
```

```
f=open('111.txt','r')
print(f.read())
f.close()
f=open('111.txt','w')
f.write('Hello,world!')
f.close()
```

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

Hello,world!

```
"E:\OneDrive - cumt.edu.cn\program\python\pythonN+\Scripts\python.exe"
进程已结束,退出代码0
```

StringIO

StringIO顾名思义就是在内存中读写str。要把str写入StringIO，我们需要先创建一个StringIO，然后，像文件一样写入即可

```
from io import StringIO
f=StringIO()
f.write('hello')
f.write(' ')
f.write('world')
print(f.getvalue())
```

```
"E:\OneDrive - cumt.edu.cn\program\py
hello world

进程已结束,退出代码0
```

BytesIO

要操作二进制数据，就需要使用BytesIO

```
from io import BytesIO
f=BytesIO()
f.write('中文'.encode('utf-8'))
print(f.getvalue())
```

```
"E:\OneDrive - cumt.edu.cn\program\python
b'\xe4\xb8\xad\xe6\x96\x87'

进程已结束,退出代码0
```

多进程多线程

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择。

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。`multiprocessing` 模块就是跨平台版本的多进程模块。

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。