第 8 次课 面向对象编程

Python 科学计算

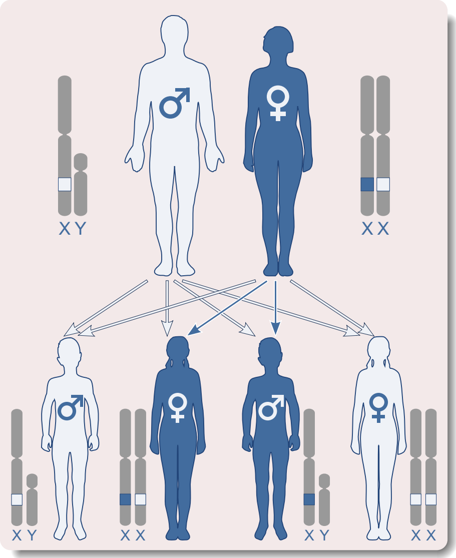
周吕文

宁波大学，机械工程与力学学院

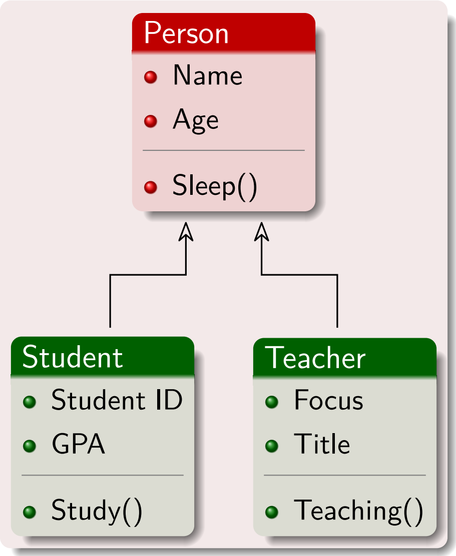
2024 年 9 月 1 日

提要

生物继承和类继承



X Y
X X
X Y
X X
X Y
X X



Person
Name
Age
Sleep()
Student
Student ID
GPA
Study()
Teacher
Focus
Title
Teaching()

面向对象（Object-oriented programming, OOP）编程

面向对象的两重意思

- 基于类的编程（基于对象的编程）✓
- 使用类的层次结构（类家族-继承）进行编程

概念

- 类的层次结构：一组关联紧密的类
- 继承：子类（派生类）可继承父类（基类）的属性和方法，减少代码重复

面向对象的学习

- OO 的概念不容易掌握，可能要花较多时间理解和消化
- 从例子出发，多尝试多实践
- 在 C++、Java 中 OO 更重要，在 Python 中相对优势较小
- 目标：编写用于数值计算的通用、可重用模块（如数值微分、积分等）

直线类 Line

Line.py: 计算 $y = c_0 + c_1 x$

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): # 返回 L, R 之间 n 个点的表格
        s = '%4s %4s\n' % ('x', 'y')
        import numpy as np
        for x in np.linspace(L, R, n):
            s += '%4g %4g\n' % (x, self(x))
        return s
if __name__=='__main__':
    L = Line(1, -2)
    print('y(2) =', L(2))
    print(L.table(0, 1, 3))
```

```
>>>
y(2) = -3
   x    y
   0    1
 0.5    0
   1   -1
```

抛物线类 Parabola

Parabola.py: 计算 $y = c_0 + c_1 x + c_2 x^2$

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2
    def __call__(self, x):
        return self.c0 + self.c1*x + self.c2*x**2
    def table(self, L, R, n): # 返回 L, R 之间 n 个点的表格
        s = '%4s %4s\n' % ('x', 'y')
        import numpy as np
        for x in np.linspace(L, R, n):
            s += '%4g %4g\n' % (x, self(x))
        return s
if __name__=='__main__':
    p = Parabola(1, -2, 2)
    print('y(2) =', p(2))
    print(p.table(0, 1, 3))
```

```
>>>
y(2) = 5
   x    y
   0    1
 0.5  0.5
   1    1
```

抛物线类的派生实现

- 抛物线类代码 = 直线类代码 + 少量 c2 相关的部分
- 继承可以让抛物线类使用直线类的代码，只需修改和 c2 相关的部分

  ```python
  class Parabola(Line):
      pass
  ```

  可以让抛物线类拥有直线类的所有属性和方法

- Line 是父类（基类），Parabola 是子类（派生类）
- Parabola 类需要在 Line 类的 `__init__` 和 `__call__` 函数基础上添加与 c2 相关的代码
- Line 类的 table 方法可直接被 Parabola 类使用
- 原则：尽可能多地重用 Line 中的代码避免重复代码

抛物线类的派生实现

派生类可以调用基类的方法

```
superclass.method(self, arg1, arg2, ...)
```

Line_Parabola.py

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

- 抛物线类继承了直线类的属性和方法，不需要重写 c0 和 c1 的初始化以及 c0 + c1*x 的计算
- 抛物线类也有 table 方法 – 继承
- 方法 __init__ 和 __call__ 被重定义了

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
⇒ class Line:
       def __init__(self, c0, c1):
           self.c0, self.c1 = c0, c1
       def __call__(self, x):
           return self.c0 + self.c1*x
       def table(self, L, R, n): ...
   class Parabola(Line):
       def __init__(self, c0, c1, c2):
           Line.__init__(self, c0, c1)
           self.c2 = c2
       def __call__(self, x):
           return Line.__call__(self, x) + self.c2*x**2

   p = Parabola(1, -2, 2)
   print('y(2) =', p(2))
   print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
⇒ class Line:
       def __init__(self, c0, c1):
           self.c0, self.c1 = c0, c1
       def __call__(self, x):
           return self.c0 + self.c1*x
       def table(self, L, R, n): ...
⇒ class Parabola(Line):
       def __init__(self, c0, c1, c2):
           Line.__init__(self, c0, c1)
           self.c2 = c2
       def __call__(self, x):
           return Line.__call__(self, x) + self.c2*x**2

   p = Parabola(1, -2, 2)
   print('y(2) =', p(2))
   print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
  class Line:
      def __init__(self, c0, c1):
          self.c0, self.c1 = c0, c1
      def __call__(self, x):
          return self.c0 + self.c1*x
      def table(self, L, R, n): ...
⇒ class Parabola(Line):
      def __init__(self, c0, c1, c2):
          Line.__init__(self, c0, c1)
          self.c2 = c2
      def __call__(self, x):
          return Line.__call__(self, x) + self.c2*x**2

⇒ p = Parabola(1, -2, 2)
  print('y(2) =', p(2))
  print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
⇒   def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

⇒ p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
⇒   def __init__(self, c0, c1, c2):
⇒       Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```python
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
⇒       self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
⇒       self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
⇒       self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
⇒ print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
⇒   def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
⇒ print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
⇒   def __call__(self, x):
⇒       return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
⇒   def __call__(self, x):
⇒       return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
⇒       return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
⇒       return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
⇒       return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
⇒ print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
y(2) = 5
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
⇒ print('y(2) =', p(2))
⇒ print(p.table(0, 1, 3))
```

```
>>>
y(2) = 5
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
print(p.table(0, 1, 3))
```

```
>>>
y(2) = 5
```

抛物线类的派生实现：程序执行过程

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1
    def __call__(self, x):
        return self.c0 + self.c1*x
⇒   def table(self, L, R, n): ...
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2

p = Parabola(1, -2, 2)
print('y(2) =', p(2))
⇒ print(p.table(0, 1, 3))
```

```
>>>
y(2) = 5
     x      y
     0      1
   0.5    0.5
     1      1
```

实例、类型、子类、类名的判断

```
>>> from Line_Parabola import Line, Parabola

>>> L = Line(-1,1)
>>> isinstance(L,Line)
True
>>> isinstance(L,Parabola)
False

>>> P = Parabola(-1,0,10)
>>> isinstance(P,Parabola)
True
>>> isinstance(P,Line)
True
```

```
>>> issubclass(Parabola,Line)
True
>>> issubclass(Line,Parabola)
False

>>> P.__class__ == Parabola
True
>>> P.__class__.__name__
'Parabola'
```

将 Line 作为 Parabola 的子类？ 当然可以，全凭程序员

- 直线 $y = c_0 + c_1 x$ 是抛物线 $y = c_0 + c_1 x + c_2 x^2$ 的特例
- 子类一般是父类的特例，将 Line 作为 Parabola 的子类更自然一些

Parabola_Line.py

```python
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2
    def __call__(self, x):
        return self.c0 + self.c1*x + self.c2*x**2
    def table(self, L, R, n): ...

class Line(Parabola):
    def __init__(self, c0, c1):
        Parabola.__init__(self, c0, c1, 0)
```

课堂练习

1 三、四次多项式函数类　　文件名：Cubic_Poly4.py

- 我们已经实现了一次函数类 `Line` 和二次函数类 `Parabola`。
- 请创建一个用于三次函数的类 `Cubic`，形式如下

$$c_3x^3 + c_2x^2 + c_1x + c_0$$

  该类需要包含一个用于计算的 `__call__` 方法和一个 `table` 方法。要求通过从类 `Parabola` 继承来实现 `Cubic` 类，并以与 `Parabola` 类调用 `Line` 类功能相同的方式调用 `Parabola` 类中的功能。
- 此外，通过继承 `Cubic` 类的方式，创建一个类似的类 `Poly4` 用于四次多项式，形式如下：

$$c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

提要

回顾数值微分（第 7 次课）

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \qquad h \text{ 是较小的数}$$

Derivative.py

```
class Derivative:
    def __init__(self, f, h=1E-5):
        self.f, self.h = f, float(h)
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

def g(t):
    return t**3

dg = Derivative(g)
print("g(1)' = %g" % dg(1))
```

```
>>>
g(1)' = 3.00003
```

数值微分公式

一阶前向/后向差分

$$f'(x)=\frac{f(x+h)-f(x)}{h}+\mathcal{O}(h),\quad f'(x)=\frac{f(x)-f(x-h)}{h}+\mathcal{O}(h)$$

二阶中心差分

$$f'(x)=\frac{f(x+h)-f(x-h)}{2h}+\mathcal{O}(h^2)$$

四阶中心差分

$$f'(x)=\frac{4}{3}\frac{f(x+h)-f(x-h)}{2h}-\frac{1}{3}\frac{f(x+2h)-f(x-2h)}{4h}+\mathcal{O}(h^4)$$

怎么建立一个包实现所有这些方法？

```
class Forward1:
    def __init__(self, f, h=1E-5):
        self.f, self.h = f, h
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
class Backward1:
    def __init__(self, f, h=1E-5):
        self.f, self.h = f, h
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h
class Central4:
    # 同样的构造函数
    # 在__call__中实现需要的公式
```

这种写法有什么问题？

```
...
class Backward1:
    def __init__(self, f, h=1E-5):
        self.f, self.h = f, h
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h
class Central4:
    # 同样的构造函数
    # 在__call__中实现需要的公式
```

所有类的构造函数都一样，代码重复了！ 怎么解决？

- OO 思想：把多个类中重复的代码放入基类，然后继承就好了
- 可以定义一个基类 `Diff`，实现构造函数
- 各派生类从基类继承构造函数，自己实现 `__call__` 函数就可以了

函数微分的继承实现

Diff.py

```
class Diff:                    # 基类
    def __init__(self, f, h=1E-5):
        self.f, self.h = f, h

class Forward1(Diff):          # 一阶前向差分的派生类
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

...
class Central4(Diff):          # 四阶中心差分的派生类
    def __call__(self, x):
        f, h = self.f, self.h
        return 4/3*(f(x+h)   - f(x-h))   /(2*h) - \
                1/3*(f(x+2*h) - f(x-2*h))/(4*h)
```

函数微分的继承实现

若 $f(x)=\sin(x)$，使用一阶前向和四阶中心差分计算 $f'(\pi)$

```
>>> from Diff import *
>>> from math import *

>>> dsin_forward1 = Forward1(sin)
>>> dsin_forward1(pi)
-0.9999999999898844

>>> dsin_central4 = Central4(sin)
>>> dsin_central4(pi)
-1.000000000065512
```

- `Forward1(sin)` 执行了基类的构造函数
- `dsin_forward1(pi)` 执行了派生类中的 `__call__` 方法

2 数值微分类扩展　　文件名：Diff_plus.py

- 我们已经实现了一阶前向差分类和四阶中心差分类。
- 请在此基础上，通过继承的方式创建二阶中心差分，公式如下：

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2)$$

- 请在此基础上，通过继承的方式创建六阶中心差分，公式如下：

$$\begin{aligned} f'(x) = \frac{3}{2}\frac{f(x+h) - f(x-h)}{2h} - \frac{3}{5}\frac{f(x+2h) - f(x-2h)}{4h} + \\ + \frac{1}{10}\frac{f(x+3h) - f(x-3h)}{6h} + \mathcal{O}(h^6) \end{aligned}$$

- 通过继承的方式，创建更多差分类用于数值积分。

提要

数值积分一般公式，其中 x_i 为点，w_i 为权值

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=0}^{n-1} w_i f(x_i)$$

- 中点法：$h = \dfrac{b-a}{n}$，$x_i = a + ih + \dfrac{h}{2}$，$w_i = h$
- 梯形法：

$$h = \frac{b-a}{n-1},\ \ x_i = a + ih,\ \ w_0 = w_{n-1} = \frac{h}{2},\ \ w_i = h\ (i \neq 0,\ n-1)$$

- 辛普森方法：

$$x_i = a + ih,\ \ h = \frac{b-a}{n-1},\ \ w_0 = w_{n-1} = \frac{h}{6},\ \ w_i = \begin{cases} h/3 & i\ \text{为偶} \\ 2h/3 & i\ \text{为奇} \end{cases}$$

为什么要将这些公式实现为类的继承？

问题：代码重复不是什么好事情

- 数值积分公式可以实现为一个类：a, b, n 是数据，而积分方法是函数
- 所有这些方法的 $\sum_j w_j f(x_j)$ 计算是相同的，仅是点和权值的定义不同

解决：把不同类的相同代码放到基类中，然后继承

- 可以把 $\sum_j w_j f(x_j)$ 的计算放到基类中，派生类继承基类的方法。
- 派生类各自实现对 w_i 和 x_i 的计算。

函数积分的继承实现

Integrate.py：实现基类 Integrator

```
import numpy as np
class Integrator:
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.points, self.weights = self.construct_method()
    def construct_method(self):
        raise NotImplementedError('no rule in class %s' %
                                  self.__class__.__name__)
    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s
    def vectorized_integrate(self, f):
        return np.dot(self.weights, f(self.points))
```

函数积分的继承实现

Integrate.py：实现子类 Midpoint & Trapezoidal

```
class Midpoint(Integrator):
    def construct_method(self):
        a, b, n = self.a, self.b, self.n  # quick forms
        h = (b-a)/float(n)
        x = np.linspace(a + 0.5*h, b - 0.5*h, n)
        w = np.zeros(len(x)) + h
        return x, w
class Trapezoidal(Integrator):
    def construct_method(self):
        x = np.linspace(self.a, self.b, self.n)
        h = (self.b - self.a)/float(self.n - 1)
        w = np.zeros(len(x)) + h
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

函数积分的继承实现

使用梯形法计算 $\int_0^2 x^2 \, dx$

```
>>> from Integrate import *
>>> f = lambda x: x**2

>>> method = Trapezoidal(0, 2, 101)
>>> method.integrate(f)
2.6668000000000001
```

- Trapezoidal(0, 2, 101) 调用基类的构造函数，该构造函数调用定义于 Trapezoidal 中的 construct_method 方法
- integrate(f) 调用从基类中继承的方法

3 数值积分类扩展　　文件名：Integrate_plus.py

- 我们已经实现了中点法和梯形法数值积分类。
- 请在此基础上，通过继承的方式创建辛普森法数值积分类，公式如下：

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=0}^{n-1} w_i f(x_i)$$

其中

$$x_i = a + ih,\quad h = \frac{b-a}{n-1},\quad w_0 = w_{n-1} = \frac{h}{6},\quad w_i = \begin{cases} h/3 & i \text{ 为偶} \\ 2h/3 & i \text{ 为奇} \end{cases}$$

The End!