第 7 次课 类简介

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

类（Class）= 数据（变量）+ 方法（函数）

- 类将数据（变量集合）和函数封装到一个单元中
- 程序员可以通过创建新类，定义新的对象类型（就如 float、list 一样）
- 类有类似于模块，“全局变量”加上属于这个模块的函数
- 模块只有一个实例，而类可以拥有多个实例（拷贝）

Dog Class

Properties:

- Breed
- Age

Methods:

- Eat()
- Sleep()

Object1

- Breed: Bulldog
- Age: 4

Object2

- Breed: Corgi
- Age: 2

Object3

- Breed: Husky
- Age: 6

使用类来实现函数：回顾

考虑以下时间 t 的函数，带有一个参数 v_0

$$y(t; v_0) = v_0 t - \frac{1}{2} g t^2$$

将 t 和 v_0 作为输入

```
def y(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

将 t 作为输入，v_0 为全局变量

```
def y(t):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

使用类来实现函数：思路

思路

- 使用类，包含函数 $y(t)$，和数据 v_0、g
- 该类封装了数据 v_0、g 和函数 y

Y Class

- `__init__` `# 初始化函数，用来初始化 v0、g`
- `value` `# 计算 y(t)`

- `g` `# = 9.8，由 __init__ 初始化`
- `v0` `# 用户通过 __init__ 初始化`

使用类来实现函数：代码

程序实现 Class_Y.py

```
class Y:
    def __init__(self, v0): # 构造函数
        self.v0 = v0
        self.g = 9.81
    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2

y = Y(v0=3) #创建实例（对象）
v = y.value(0.1)
print(v)
```

```
>>>
0.25095
```

使用类来实现函数：构造函数 & 参数 self

构造函数 __init__

- y = Y(v0=3) 声明（创建）了 Y 的对象（实例、变量）y
- Y(3) 实际调用了以下构造函数

```
def __init__(self, v0): # 构造函数
    self.v0 = v0
    self.g = 9.81
```

参数 self

- 可以把 self 理解为你要创建的对象（如 y）
- self.v0 = v0：为 self（也就是 y）增加一个数据 v0，并赋值为 v0（参数，不同于前一个 v0）
- Y(3) 相当于 Y.__init__(y, 3)，即 self = y，v0 = 3
- 注意：self 永远是第一个参数，但调用的时候不需要管它
- 执行 y = Y(3) 后，y 拥有 v0 和 g 两个数据：print(y.v0, y.g)

使用类来实现函数：value 方法

- 类中的变量叫属性，类中的函数叫方法
- 方法 value:

```
def value(self, t):
    return self.v0*t - 0.5*self.g*t**2
```

- 函数调用 v = y.value(0.1)，程序会自动将 y 作为 self 参数传递给方法，相当于

```
Y.value(y, t=0.1)
```

- 函数的执行相当于

```
return y.v0*t - 0.5*y.g*t**2
```

- self 帮助访问类中定义的“全局变量”

使用类来实现函数：y.value 可作为 t 的函数用于它处

Class_Y.py

```
from numpy import linspace, pi, sin, exp

def table(f, tstop, n):
    print('-'*12)
    for t in linspace(0, tstop, n):
        print('%4.2f %7.2f' % (t, f(t)) )

def g(t):
    return sin(t)*exp(-t)

table(g, 2*pi, 6)          # 调用普通函数

y = Y(6.5)
table(y.value, 2*pi, 6) # 调用类方法
```

IDLE

```
------------
0.00    0.00
1.26    0.27
2.51    0.05
3.77   -0.01
5.03   -0.01
6.28   -0.00
------------
0.00    0.00
1.26    0.42
2.51  -14.65
3.77  -45.21
5.03  -91.26
6.28 -152.80
```

使用类来实现函数：一般形式

对于包含一个变量 x 和 $n+1$ 个参数的函数 $f(x; p_0, p_1, \cdots, p_n)$

- 封装 $n+1$ 个数据（p0, p1, ..., pn）
- 再封装一个方法 value(self, x) 来计算 f(x)

```
class MyFunc:
    def __init__(self, p0, p1, p2, ..., pn):
        self.p0 = p0
        self.p1 = p1
        ...
        self.pn = pn

    def value(self, x):
        return ...
```

例子：用类实现需要4个参数的函数

$$v(r;\,\beta,\mu_0,n,R)=\left(\frac{\beta}{2\mu_0}\right)^{\frac{1}{n}}\frac{n}{n+1}\left(R^{1+\frac{1}{n}}-r^{1+\frac{1}{n}}\right)$$

VelocityProfile.py

```
class VelocityProfile:
    def __init__(self, b, u0, n, R):
        self.b, self.u0, self.n, self.R = b, u0, n, R
    def value(self, r):
        b, u0, n, R = self.b, self.u0, self.n, self.R
        v = (b/(2*u0))**(1/n)*(n/(n+1))*\
            (R**(1+1/n) - r**(1+1/n))
        return v

v = VelocityProfile(0.06, 0.02, 0.1, 1)
print('%.2f' % v.value(r=0.1))
```

```
>>>
5.24
```

Python 类的一般形式

MyClass.py

```
class MyClass:
    def __init__(self, p1, p2):
        self.attr1, self.attr2 = p1, p2
    def method1(self, arg):
        self.attr3 = arg      # 可以在构造函数外增加变量
        return self.attr1 + self.attr2 + self.attr3
    def method2(self):
        print('Hello!')

m = MyClass(4, 10)
print(m.method1(-2))
m.method2()
```

```
>>>
12
Hello!
```

- 一般会有定义变量的构造函数，但也可以没有，变量可在需要时添加

到底该如何理解 self？

两种选择

- 仔细理解 self 的说明和程序执行原理，或者
- 先不用管，类的程序写多了自然就懂了

在方法中，self 就是执行该方法的对象

语句 y = Y(3) 可以理解为

- 首先：Y.__init__(y, 3) # 之前 y 必须是 Y 的对象
- 然后：self.v0 = v0 实际上就是 y.v0 = 3

语句 v = y.value(2) 也可以写成：v = Y.value(y, 2)

到底该如何理解 self?

id(obj) 返回对象在 Python 中的唯一标识 SelfExplorer.py

```
class SelfExplorer: # Class for computing a*x
    def __init__(self, a):
        self.a = a
        print('i: id(self)=%d' % id(self))
    def value(self, x):
        print('v: id(self)=%d' % id(self))
        return(self.a*x)
```

```
>>> s1 = SelfExplorer(1)
i: id(self)=139162150142208
>>> id(s1)
139162150142208
>>> s1.value(4)
v: id(self)=139162150142208
4
>>> SelfExplorer.value(s1,4)
v: id(self)=139162150142208
4
>>> s2 = SelfExplorer(2)
i: id(self)=139162150142880
>>> id(s2)
139162150142880
```

例子：银行账号

- 属性：姓名（name）、账号（account_number）、余额（balance）
- 方法：存款（deposit）、取款（withdraw）、输出（dump）

Account.py

```
class Account:
    def __init__(self, name, account_number, init_amount):
        self.name = name
        self.no = account_number
        self.balance = init_amount
    def deposit(self, amount):  self.balance += amount
    def withdraw(self, amount): self.balance -= amount
    def dump(self):
        print('%s, %s, balance: %s' %\
        (self.name, self.no, self.balance))
```

例子：银行账号

使用 Account 类

```
>>> a1 = Account('Zhang san', '198706093029', 20000)
>>> a2 = Account('Li si', '199407292637', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a2.withdraw(10500)
>>> a1.withdraw(3500)
>>> print("a1's balance:", a1.balance)
a1's balance: 13500
>>> a1.dump()
Zhang san, 198706093029, balance: 13500
>>> a2.dump()
Li si, 199407292637, balance: 9500
```

例子：银行账号

程序允许但实际错误的用法

```
>>> a1.name = 'Wang wu'
>>> a1.balance = 100000
>>> a1.no = '19371564768'
>>> a1.dump()
Wang wu, 19371564768, balance: 100000
```

实际准则

- 银行有些数据不能被修改，如姓名、账号
- 银行有些数据的修改需要被控制，如余额只能在取款、存款时被修改

补救方法

- 以双下划线开始的属性和方法是私有的，不可以在类外使用
- 以单下划线开始的属性和方法具有受保护属性，但在类外仍可访问

例子：银行账号

AccountP.py

```
class AccountP:
    def __init__(self, name, account_number, init_amount):
        self.__name = name
        self.__no = account_number
        self.__balance = init_amount
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):
        self.__balance -= amount
    def get_balance(self): # 获取余额
        return self.__balance
    def dump(self):
        print('%s, %s, balance: %s' % \
            (self.__name, self.__no, self.__balance))
```

例子：银行账号

使用 AccountP 类

```
>>> a1 = AccountP('Zhang san', '199407292637', 20000)
>>> a1.withdraw(4000)
>>> a1.__name                # 同样，a1.__balance 也会报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AccountP' object has no attribute '__name'
>>> a1.__name = 'Li si' # 增加了一个普通属性，但不改变私有属性
>>> a1.__name                # 此时访问的是普通属性
'Li si'
>>> a1._AccountP__name  # 私有属性外部访问方式
'Zhang san'
>>> a1.dump()
Zhang san, 199407292637, balance: 16000
>>> a1.get_balance()     # a1.__balance 会报错
16000
```

例子：电话簿

电话簿是人员信息的列表

数据：

- 姓名
- 手机号码
- 工作电话
- 电子邮箱

方法：

- 构造函数，初始化姓名 ……
- 更新（没有的话则增加）手机号码
- 更新（没有的话则增加）工作电话
- 更新（没有的话则增加）电子邮箱

例子：电话簿

Person.py

```
class Person:
    def __init__(self, name, mobile=None, \
                 office=None, email=None):
        self.name, self.mobile= name, mobile
        self.office, self.email = office, email
    def add_mobile(self, number): self.mobile = number
    def add_office(self, number): self.office = number
    def add_email(self, address): self.email = address
    def dump(self):
        s = self.name + '\n'
        for x in ['mobile', 'office', 'email']:
            if eval('self.'+x) is not None:
                s += '%s: %s\n' % (x, eval('self.'+x))
        print(s)
```

例子：电话簿

使用 Person 类

```
>>> p1 = Person('Zhou', email='zhou@nbu.edu.cn')
>>> p1.add_mobile('18888888888')
>>> p1.dump()
Zhou
mobile: 18888888888
email: zhou@nbu.edu.cn

>>> p2 = Person('Yang', office='87609384')
>>> p2.add_email('young@gmail.com')
>>> p2.dump()
Yang
office: 87609384
email: young@gmail.com
```

课堂练习

1 创建圆类 文件名：Circle.py

圆可以用中点坐标 (x_0, y_0) 和半径 R 表示，为圆定义类 Circle。类 Circle 具有如下**属性**：

- x0、y0：圆心坐标
- R：圆的半径

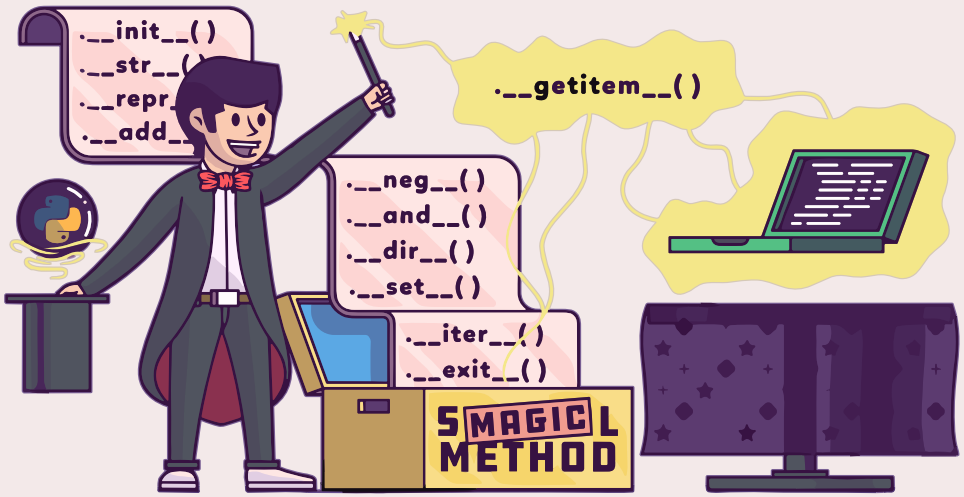
类 Circle 具有如下**方法**：

- dump：输出圆的基本信息
- area：求圆的面积
- circumference：求圆的周长

为类 Circle 编写测试函数 test_Circle。

提要

特殊方法：以双下划线开始和结尾的方法




```
def __init__(self, ...)

def __call__(self, ...)

def __add__(self, other)
```

表面上

```
y = Y(4)
print(y(2))
z = Y(6)
print(y + z)
```

$\Longrightarrow$

实际上

```
Y.__init__(y, 4)
print(Y.__call__(y, 2))
Y.__init__(z, 6)
print(Y.__add__(y, z))
```

例子：使用 call 函数替代 value

Class_Y2.py

```
class Y:
    def __init__(self, v0):
        self.v0, self.g = v0, 9.81
    def __call__(self, t): # def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

把对象 y 当函数使，使用 call 替代 value 方法语法更自然

```
>>> y = Y(3)
>>> v = y(0.1)   # v = y.__call__(0.1) 或
>>>             # v = Y.__call__(y, 0.1)
>>> print(v)
0.25095
```

用类实现函数（使用 __call__）

实现一个变量和 $n+1$ 个参数的函数 $f(x; p_0, p_1, \cdots, p_n)$ 类

- 属性：p0, p1, ···, pn
- 方法：__call__(x) 实现 $f(x)$

```
class MyFunc:
    def __init__(self, p0, p1, p2, ..., pn):
        self.p0 = p0
        self.p1 = p1
        ...
        self.pn = pn
    def __call__(self, x):
        return ...
```

求函数的微分

对于函数 $f(x)$，实现一个类 Derivative，使之能求 $f(x)$ 的微分

例如对于函数

```
def f(x):
    return x**3
```

以下程序能计算函数 $f(x) = x^3$ 在 $x = 2$ 处的微分值：

```
dfdx = Derivative(f)
dfdx(2) # = 3*x**2 = 12
```

求函数的数值微分，可以使用下面的公式

$$f'(x) = \frac{f(x+h) - f(x)}{h}, \quad h \text{ 足够小，如 } h = 10^{-5}$$

求函数的微分：类程序实现和使用

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
```

```
>>> from math import sin, cos, pi
>>> df = Derivative(sin)
>>> df(pi) # cos(pi)
-0.9999999999898844
>>> g = lambda t: t**3
>>> dg = Derivative(g)
>>> dg(1)
3.000030000110953
```

求函数的微分：类的测试函数

测试方法

1. 手工计算 $(f(x+h)-f(x))/h$
2. 使用线性方程，其微分值跟 h 无关

Derivative.py

```
def test_Derivative():
    f = lambda x: a*x + b # 线性函数，和 h 无关
    a = 3.5; b = 8
    dfdx = Derivative(f, h=0.5)
    diff = abs(dfdx(4.5) - a)
    assert diff < 1E-14, \
           'bug in class Derivative, diff=%s' % diff
```

求函数的微分：类的测试函数

嵌套函数

```
def test_Derivative():
    f = lambda x: a*x + b          # <=> def f(x):
    a = 3.5; b = 8                 #         a*x + b
    dfdx = Derivative(f, h=0.5)
    dfdx(4.5)
```

Derivative.__call__ 调用 f(x) 时怎么知道 a 和 b 值的？ → 闭包

- f 即使在类的 __call__ 中被调用，仍可访问 test_Derivative 中的变量 a 和 b。在计算机科学中，f 被称为闭包（closure）。
- 在函数内构建并返回的函数叫闭包，它可记住父函数内局部变量的值。
- 猜猜以下程序的输出结果：

```
def make_multiplier_of(n): return lambda x: x*n
times3 = make_multiplier_of(3); print(times3(5))
times5 = make_multiplier_of(5); print(times5(9))
```

求函数的微分：使用符号 SymPy 计算

```
>>> from sympy import *
>>> def g(t):
...     return t**3
...
>>> t = Symbol('t')
>>> dgdt = diff(g(t), t)
>>> dgdt
3*t**2

>>> dg = lambdify([t],dgdt) # 以计算机函数的形式定义 dgdt(t)
>>> dg(1)
3
```

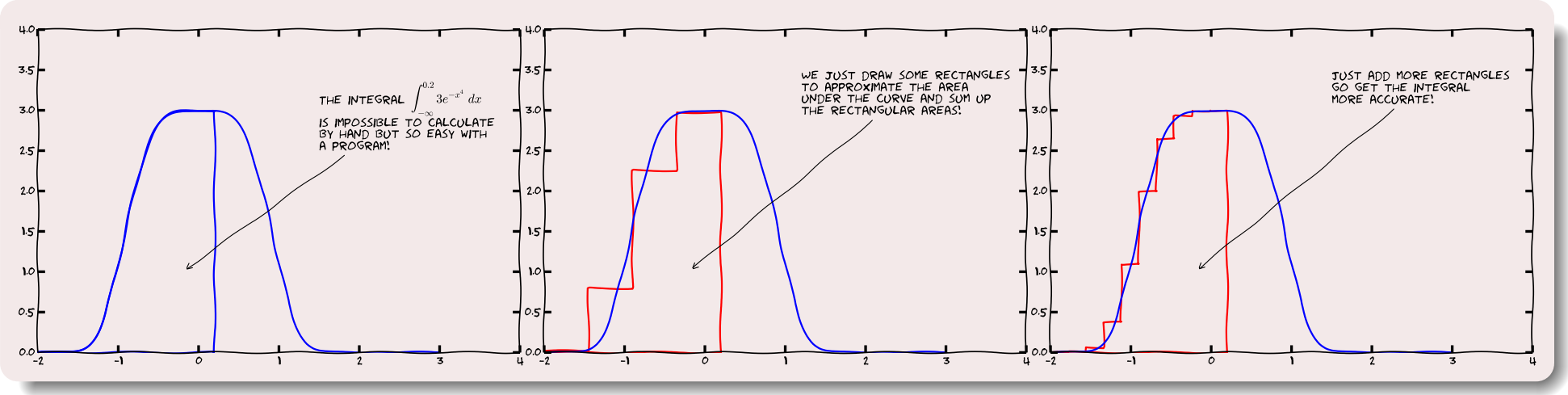
求函数的微分：使用符号 SymPy 计算

Derivative_sympy.py

```
import sympy as sp
class Derivative_sympy:
    def __init__(self, f):
        self.x = sp.Symbol('x')
        self.sympy_dfdx = sp.diff(f(self.x), self.x)
    def __call__(self,n):
        func = sp.lambdify([self.x], self.sympy_dfdx)
        return func(n)
```

```
>>> dg = Derivative_sympy(lambda t: t**3)
>>> dh = Derivative_sympy(lambda y: sp.sin(y))
>>> print(dg(1))  # 3*t**2 for t=1
3
>>> print(dh(pi)) # cos(y) for y=pi
-1.0
```

求函数的积分



求给定函数 $f(x)$ 积分：中点法或者梯形法，这里选择梯形法

$$F(x,a) = \int_a^x f(t)\mathrm{d}t$$

$$\approx h\left(\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(a+ih) + \frac{f(x)}{2}\right), \qquad h = \frac{x-a}{n}$$

求函数的积分：类程序实现和使用

Integral.py

```
def trapezoidal(f, a, x, n):
    h = (x-a)/n; I = (f(a)+f(x))/2
    for i in range(1, n):
        I += f(a + i*h)
    return I*h

class Integral:
    def __init__(self, f, a, n=100):
        self.f, self.a, self.n = f, a, n
    def __call__(self, x):
        return trapezoidal(self.f, self.a, x, self.n)
```

```
>>> F = Integral(f = lambda x: exp(-x**2), a=0, n=200)
>>> F(1.2)
0.8067430522430105
```

求函数的积分：类的测试函数

测试方法

1. 对某个函数 f 和较小的 n 手动计算积分结果
2. 对线性函数积分，结果和 n 的大小无关

Integral.py

```
def test_Integral():
    f = lambda x: 2*x + 5 # 线性函数积分，结果和 h 无关
    F = lambda x: x**2 + 5*x
    a, x = 2, 6
    I = Integral(f, a, n=4)
    diff = abs(I(x) - (F(x) - F(a)))
    assert diff < 1E-15, \
           'bug in class Integral, diff=%s' % diff
```

用于输出的特殊方法：__str__

- 预定义类型的对象可以用 print(a) 直接输出
- 自定义类的对象若定义了方法 __str__，也可以使用 print 直接输出

Class_Y2.py

```
class Y:
    def __init__(self, v0):
        self.v0, self.g = v0, 9.81
    def __call__(self, t): # def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

```
>>> y = Y(3)
>>> y(0.1)
0.25095
```

```
>>> print(y)
v0*t - 0.5*g*t**2; v0=3
```

多项式类

一个多项式可以用系数的列表来表示

$1 - x^2 + 2x^3 \iff 1 + 0x - 1x^2 + 2x^3 \iff$ `[1, 0, -1, 2]`

期望效果如下，怎么实现？

```
>>> p1 = Polynomial([1, -1])
>>> print(p1)
1 - x
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = p1 + p2
>>> print(p3.coeff)
[1, 0, 0, 0, -6, -1]
>>> print(p3)
1 - 6*x^4 - x^5
>>> p2.differentiate()
>>> print(p2)
1 - 24*x^3 - 5*x^4
```

多项式类：准备

Polynomial.py: __init__ & __call__

```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s
```

多项式类：实现加法

Polynomial.py: __add__

```
class Polynomial():
    ... ...
    def __add__(self, other): # Return self + other
        # Start with the longest list and add in the other
        if len(self.coeff) > len(other.coeff):
            coeff = self.coeff[:]  # copy!
            for i in range(len(other.coeff)):
                coeff[i] += other.coeff[i]
        else:
            coeff = other.coeff[:] # copy!
            for i in range(len(self.coeff)):
                coeff[i] += self.coeff[i]
        return Polynomial(coeff)
```

多项式类：实现乘法

多项式乘法公式

$$\left(\sum_{i=0}^{M} c_i x^i\right)\left(\sum_{j=0}^{N} d_j x^j\right) = \sum_{i=0}^{M}\sum_{j=0}^{N} c_i d_j x^{i+j}$$

Polynomial.py: __mul__

```
class Polynomial():
    ... ...
    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1)
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i]*other.coeff[j]
        return Polynomial(coeff)
```

多项式类：实现微分

多项式微分公式

$$\frac{\mathrm{d}}{\mathrm{d}x}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}$$

Polynomial.py: __differentiate__ & __derivative__

```
class Polynomial():
    ... ...
    def differentiate(self): # change self
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i*self.coeff[i]
        del self.coeff[-1]
    def derivative(self):     # return new polynomial
        dpdx = Polynomial(self.coeff[:])  # copy
        dpdx.differentiate()
        return dpdx
```

多项式类：实现输出

Polynomial.py: __str__

```
class Polynomial():
    ... ...
    def __str__(self):
        s = ' '
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += '+ %g*x^%d ' % (self.coeff[i], i)
        # fix layout (lots of special cases):
        s = s.replace('+ -', '- ')
        s = s.replace('x^0', '1')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^1 ', 'x ')
        if s[0:3] == ' + ':  s = s[3:]
        if s[0:3] == ' - ':  s = '-' + s[3:]
        return s
```

多项式类：使用

$$p_1(x) = 1 - x$$
$$p_2(x) = x - 6x^4 - x^5 \quad \Longrightarrow \quad p_3(x) = p_1(x) + p_2(x) = 1 - 6x^4 - x^5$$

```
>>> p1 = Polynomial([1, -1])
>>> print(p1)
1 - x
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = p1 + p2
>>> print(p3.coeff)
[1, 0, 0, 0, -6, -1]
>>> print(p3)
1 - 6*x^4 - x^5
>>> p2.differentiate()
>>> print(p2)
1 - 24*x^3 - 5*x^4
```

运算符重载函数的定义完全取决于程序员

__add__(self, other) 应该如何定义？

- 对于列表：

```
>>> [315, 211] + ['N', 'B', 'U']
[315, 211, 'N', 'B', 'U']
```

- 对于数字：

```
>>> 315 + 211
526
```

- 对于字符串：

```
>>> 'NBU' + ' ' + 'YYDS'
'NBU YYDS'
```

- 对于多项式：

```
>>> print(p1 + p2)
2*1 - x - 24*x^3 - 5*x^4
```

- 取决于程序员的设计，由 object1 + object2 的意义决定。

特殊方法

算术运算符

- `c = a + b` *# c = a.__add__(b)*
- `c = a - b` *# c = a.__sub__(b)*
- `c = a*b` *# c = a.__mul__(b)*
- `c = a/b` *# c = a.__truediv__(b)，区别于 py2 中的 __div__*
- `c = a**e` *# c = a.__pow__(e)*

比较运算

- `a == b` *# a.__eq__(b)*
- `a != b` *# a.__ne__(b)*
- `a < b` *# a.__lt__(b)*
- `a <= b` *# a.__le__(b)*
- `a > b` *# a.__gt__(b)*
- `a >= b` *# a.__ge__(b)*

平面向量类

平面向量操作

- $(a,b)-(c,d)=(a-c,b-d)$ $(a,b)+(c,d)=(a+c,b+d)$
- $(a,b)\cdot(c,d)=ac+bd$
- $(a,b)=(c,d)$ if $a=c$ and $b=d$

期望效果如下，怎么实现?

```
>>> u, v = Vec2D(0,1), Vec2D(1,0)
>>> a = u + v
>>> print(a)
(1, 1)
>>> w = Vec2D(1,1)
>>> a == w
True
>>> print(u*v)
0
```

平面向量类：代码

Vec2D.py

```
class Vec2D:
    def __init__(self, x, y): self.x, self.y = x, y
    def __add__(self, other):
        return Vec2D(self.x + other.x, self.y + other.y)
    def __sub__(self, other):
        return Vec2D(self.x - other.x, self.y - other.y)
    def __mul__(self, other):
        return self.x*other.x + self.y*other.y
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
    def __abs__(self):
        return (self.x**2 + self.y**2)**0.5
    def __ne__(self, other): return not self.__eq__(other)
```

特殊方法 repr: 让 eval(repr(p)) 可以创建 p

MyClass_repr.py

```
class MyClass:
    def __init__(self, a, b):
        self.a, self.b = a, b
    def __str__(self):
        return 'a=%s, b=%s' % (self.a, self.b)
    def __repr__(self):
        return 'MyClass(%s, %s)' % (self.a, self.b)
```

```
>>> m = MyClass(1, 5)
>>> m          # m.__repr__()
MyClass(1, 5)
>>> print(m) # m.__str__()
a=1, b=5
>>> str(m)    # m.__str__()
'a=1, b=5'
```

```
>>> # m.__repr__()
>>> s = repr(m)
>>> s
'MyClass(1, 5)'
>>> m2 = eval(s)
>>> m2
MyClass(1, 5)
```

__str__ 与 __repr__ 的区别

- __str__：对象的非正式、易于阅读的字符串描述，当 str(object) 时会被调用，以及会被内置函数 format() 和 print() 调用。
- __repr__：对象的官方字符串描述，会被内置函数 repr() 方法调用，它的描述必须是信息丰富的和明确的。

```
>>> import datetime
>>> d = datetime.datetime(1994, 7, 29, 11, 30, 29)
>>> print(d)  # print 调用的是 __str__
1994-07-29 11:30:29
>>> d          # 交互模式下，直接输入再回车，调用的是 __repr__
datetime.datetime(1994, 7, 29, 11, 30, 29)
>>> str(d)
'1994-07-29 11:30:29'
>>> repr(d)
'datetime.datetime(1994, 7, 29, 11, 30, 29)'
```

复数类

- Python 已经有针对复数的类 complex
- 重新实现可以帮助我们更好地理解类，特别是特殊方法的编程

期望效果如下，怎么实现？

```
>>> u = Complex(2,-1)
>>> u
Complex(2, -1)
>>> v = Complex(1)  # 虚部为 0
>>> v
Complex(1, 0)
>>> w = u + v
>>> w
Complex(3, -1)
>>> w != u
True
>>> u*v
Complex(2, -1)
>>> w + 4
Complex(7, -1)
>>> 4 - w
Complex(1, 1)
```

复数类：程序

Complex.py

```
class Complex:
    def __init__(self, real, imag=0.0):
        self.real, self.imag = real, imag
    def __add__(self, other):
        return Complex(self.real + other.real,
                       self.imag + other.imag)
    def __sub__(self, other):
        return Complex(self.real - other.real,
                       self.imag - other.imag)
    def __mul__(self, other):
        real = self.real*other.real - self.imag*other.imag
        imag = self.imag*other.real + self.real*other.imag
        return Complex(real, imag)

    ... ...
```

Complex.py

```
class Complex:
    ... ...
    def __truediv__(self, other):
        ar, ai = self.real, self.imag
        br, bi = other.real, other.imag
        r = float(br**2 + bi**2)
        return Complex((ar*br+ai*bi)/r, (ai*br-ar*bi)/r)
    def __abs__(self):
        return sqrt(self.real**2 + self.imag**2)
    def __neg__(self):   # defines -c (c is Complex)
        return Complex(-self.real, -self.imag)
    def __eq__(self, other):
        return self.real == other.real and \
               self.imag == other.imag
    def __ne__(self, other):
        return not self.__eq__(other)

    ... ...
```

复数类：程序

Complex.py

```
class Complex:
    ... ...
    def __str__(self):
        return '(%g, %g)' % (self.real, self.imag)
    def __repr__(self):
        return 'Complex' + str(self)

    def __pow__(self, power):
        raise NotImplementedError\
        ('power operation is not yet impl. for Complex')

    def __gt__(self, other):  self._illegal('>')
    def __ge__(self, other):  self._illegal('>=')
    def __lt__(self, other):  self._illegal('<')
    def __le__(self, other):  self._illegal('<=')
```

复数类：程序

怎么实现 4.5 + u 的计算？

```
>>> u + 4.5        # u.__add__(Complex(4.5))
Complex(6.5, -1)
>>> 4.5 + u
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'float' and 'Co
```

Complex.py: __radd__(self, other) 系统解释为 other + self

```
class Complex:
    ... ...
    def __radd__(self, other):
        return self.__add__(other)
```

复数类：程序

减法怎么办？

```
>>> u - 4.5       # u.__sub__(Complex(4.5))
Complex(-2.5, -1)
>>> 4.5 - u
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'float' and 'Co
```

Complex.py: __rsub__(self, other) 系统解释为 other - self

```
class Complex:
    ... ...
    def __rsub__(self, other):
        if isinstance(other, (float,int)):
            other = Complex(other)
        return other.__sub__(self)
```

怎么查看类（实例）

```
>>> u = Complex(2,-1)
>>> u.__dict__
{'real': 2, 'imag': -1}
>>> dir(u)
['__abs__', '__add__', '__class__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__module__', '__mul__', '__ne__',
'__neg__', '__new__', '__pow__', '__radd__',
'__reduce__', '__reduce_ex__', '__repr__', '__rsub__',
'__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__weakref__',
'imag', 'real']
```

可以随时在实例中增加属性

```
>>> class A:
...     """ demo class."""
...     def __init__(self, value):
...         self.v = value
...
>>> a = A([1,2])
>>> a.__doc__
' demo class.'
>>> a.newvar = 10
>>> a.__dict__
{'v': [1, 2], 'newvar': 10}
>>> dir(a)
[..., '__doc__', ..., '__init__',..., 'newvar', 'v']
>>> b = A(1)
>>> dir(b)
[..., '__doc__', ..., '__init__',..., 'v']
```

2 区间算术类　　文件名：IntervalMath.py

设计一个区间算术类 `IntervalMath`，假设有以下两个区间

$$p = [a,\ b], \qquad q = [c,\ d]$$

则相应的算术表达式应进行的实际运算如下

- $p + q = [a + c,\ b + d]$
- $p - q = [a - d,\ b - c]$
- $pq = [\min(ac,\ ad,\ bc,\ bd),\ \max(ac,\ ad,\ bc,\ bd)]$
- $p/q = \left[\min\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right), \max\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right)\right]$，其中 $0 \notin [c,\ d]$

要有相应的 `__str__` 和 `__repr__` 方法。

The End!