PythonCompact2

September 4, 2019

Short introduction to Python Faculty of Science Lund University
February, 2018
Claus Führer, Numerical Analysis, Matematikcentrum
Najmeh Abiri, Computational Biology, Fysikum

1 Python Compact - Part 2: Classes in Python 3.6

from scipy import *from matplotlib.pyplot import* %matplotlib inline

1.1 Introduction

```
In [1]: class RationalNumber:
            pass
```

```
In [2]: a=RationalNumber()
        if isinstance(a, RationalNumber):
            print('Indeed it belongs to the class RationalNumber')
```

```
Indeed it belongs to the class RationalNumber
```

1.1.1 The `__init__` method

```
In [3]: class RationalNumber:
            def __init__(self, numerator, denominator):
                self.numerator = numerator
                self.denominator = denominator
```

```
In [4]: q = RationalNumber(10, 20)        # Defines a new object
        q.numerator # returns 10
```

```
Out[4]: 10
```

```
In [5]: q.denominator # returns 20
```

```
Out[5]: 20
```

1.2 Attributes

```
In [6]: q = RationalNumber(3, 5) # instantiation
        q.numerator          # attribute access
```

```
Out[6]: 3
```

```
In [7]: q.denominator
```

```
Out[7]: 5
```

```
In [8]: a = array([1, 2]) # instantiation
        a.shape
```

```
Out[8]: (2,)
```

```
In [9]: z = 5 + 4j # instantiation
        z.imag
```

```
Out[9]: 4.0
```

```
In [10]: q = RationalNumber(3, 5)
         q.numerator
```

```
Out[10]: 3
```

```
In [11]: r = RationalNumber(7, 3)
         q.numerator = 17
         q.numerator
```

```
Out[11]: 17
```

```
In [12]: del r.denominator
```

```
In [13]: class RationalNumber:
             def __init__(self, numerator, denominator):
                 self.numerator = numerator
                 self.denominator = denominator
             def convert2float(self):
                 return float(self.numerator) / float(self.denominator)
```

```
In [14]: q = RationalNumber(10, 20)        # Defines a new object
         q.convert2float() # returns 0.5
```

```
Out[14]: 0.5
```

```
In [15]: RationalNumber.convert2float(q)
```

```
Out[15]: 0.5
```

```
In [16]: q.convert2float(15)    # returns error
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-16-d2abe2c23504> in <module>()
----> 1 q.convert2float(15)   # returns error


TypeError: convert2float() takes 1 positional argument but 2 were given
```

1.2.1 Special Methods

- The method repr

```
In [17]: class RationalNumber:
             def __init__(self, numerator, denominator):
                 self.numerator = numerator
                 self.denominator = denominator
             def convert2float(self):
                 return float(self.numerator) / float(self.denominator)
             def __repr__(self):
                 return '{} / {}'.format(self.numerator,self.denominator)
```

```
In [18]: q = RationalNumber(10, 20)
         q
```

```
Out[18]: 10 / 20
```

- The method __add__

```
In [19]: class RationalNumber:
             def __init__(self, numerator, denominator):
                 self.numerator = numerator
                 self.denominator = denominator
             def convert2float(self):
                 return float(self.numerator) / float(self.denominator)
             def __repr__(self):
                 return '{} / {}'.format(self.numerator,self.denominator)
             def __add__(self, other):
                 p1, q1 = self.numerator, self.denominator
                 if isinstance(other, int):
                     p2, q2 = other, 1
                 else:
                     p2, q2 = other.numerator, other.denominator
                 return RationalNumber(p1 * q2 + p2 * q1, q1 * q2)
```

```
In [20]: q = RationalNumber(1, 2)
         p = RationalNumber(1, 3)
         q + p # RationalNumber(5, 6)
```

```
Out[20]: 5 / 6

In [21]: q.__add__(p)

Out[21]: 5 / 6

In [22]: class RationalNumber:
             def __init__(self, numerator, denominator):
                 self.numerator = numerator
                 self.denominator = denominator
             def convert2float(self):
                 return float(self.numerator) / float(self.denominator)
             def __repr__(self):
                 return '{} / {}'.format(self.numerator,self.denominator)
             def __add__(self, other):
                 p1, q1 = self.numerator, self.denominator
                 if isinstance(other, int):
                     p2, q2 = other, 1
                 else:
                     p2, q2 = other.numerator, other.denominator
                 return RationalNumber(p1 * q2 + p2 * q1, q1 * q2)
             def __eq__(self, other):
                 return self.denominator * other.numerator == \
                        self.numerator * other.denominator

In [23]: p = RationalNumber(1, 2) # instantiation
         q = RationalNumber(2, 4) # instantiation
         p == q # True

Out[23]: True

In [24]: p = RationalNumber(1, 2) # instantiation
         p + 5  # corresponds to p.__add__(5)

Out[24]: 11 / 2

In [25]:  5 + p  # returns an error


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-25-eef2f71f5bf2> in <module>()
    ----> 1 5 + p  # returns an error


        TypeError: unsupported operand type(s) for +: 'int' and 'RationalNumber'
```

- The reverse method `__radd__`

In [26]:
```
class RationalNumber:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
    def convert2float(self):
        return float(self.numerator) / float(self.denominator)
    def __repr__(self):
        return '{} / {}'.format(self.numerator,self.denominator)
    def __add__(self, other):
        p1, q1 = self.numerator, self.denominator
        if isinstance(other, int):
            p2, q2 = other, 1
        else:
            p2, q2 = other.numerator, other.denominator
        return RationalNumber(p1 * q2 + p2 * q1, q1 * q2)
    def __eq__(self, other):
        return self.denominator * other.numerator == \
               self.numerator * other.denominator
    def __radd__(self, other):
        return self
```

In [27]:
```
p = RationalNumber(1, 2)
5 + p  # no error message any more
```

Out[27]:
```
1 / 2
```

In [28]:
```
import itertools

class  Recursion3Term:
    def __init__(self, a0, a1, u0, u1):
        self.coeff = [a1, a0]
        self.initial = [u1, u0]
    def __iter__(self):
        u1, u0 = self.initial
        yield u0  # (see chapter on generators)
        yield u1
        a1, a0 = self.coeff
        while True :
            u1, u0 = a1 * u1 + a0 * u0, u1
            yield u1
    def __getitem__(self, k):
        return list(itertools.islice(self, k, k + 1))[0]
```

In [29]:
```
r3 = Recursion3Term(-0.35, 1.2, 1, 1)
for i, r in enumerate(r3):
    if i == 7:
        print(r)  # returns 0.194167
        break
```

```
0.194167
```

```
In [30]: r3[7]
```

```
Out[30]: 0.194167
```

1.2.2 Attributes that depend on each other

```
In [31]: class Triangle:
             def __init__(self,  A, B, C):
                 self.A = array(A)
                 self.B = array(B)
                 self.C = array(C)
                 self.a = self.C - self.B
                 self.b = self.C - self.A
                 self.c = self.B - self.A
             def area(self):
                 return abs(cross(self.b, self.c)) / 2
```

```
In [32]: tr = Triangle([0., 0.], [1., 0.], [0., 1.])
```

```
In [33]: tr.area()
```

```
Out[33]: 0.5
```

```
In [34]: tr.B = [12., 0.]
         tr.area() # still returns 0.5, should be 6 instead.
```

```
Out[34]: 0.5
```

The function property

```
In [35]: class Triangle:
             def __init__(self, A, B, C):
                 self._A = array(A)
                 self._B = array(B)
                 self._C = array(C)
                 self._a = self._C - self._B
                 self._b = self._C - self._A
                 self._c = self._B - self._A
             def area(self):
                 return abs(cross(self._c, self._b)) / 2.
             def set_B(self, B):
                 self._B = B
                 self._a = self._C - self._B
                 self._c = self._B - self._A
             def get_B(self):
                 return self._B
             def del_Pt(self):
                 raise Exception('A triangle point cannot be deleted')
             B = property(fget = get_B, fset = set_B, fdel = del_Pt)
```

```
In [36]: tr = Triangle([0., 0.], [1., 0.], [0., 1.])
         tr.area()

Out[36]: 0.5

In [37]: tr.B = [12., 0.]
         tr.area() # returns 6.0

Out[37]: 6.0

In [38]: del tr.B  # raises an exception

        ---------------------------------------------------------------------------

        Exception                                 Traceback (most recent call last)

        <ipython-input-38-dd7bb3a190c2> in <module>()
    ----> 1 del tr.B  # raises an exception


        <ipython-input-35-e3a0218bc995> in del_Pt(self)
         16         return self._B
         17     def del_Pt(self):
    ---> 18         raise Exception('A triangle point cannot be deleted')
         19     B = property(fget = get_B, fset = set_B, fdel = del_Pt)


        Exception: A triangle point cannot be deleted
```

1.2.3 Bound and unbound methods

```
In [39]: class A:
             def func(self, arg):
                 pass
         A.func  # <unbound method A.func>

Out[39]: <function __main__.A.func>

In [40]: instA = A()  # we create an instance
         instA.func  #  <bound method A.func of ... >

Out[40]: <bound method A.func of <__main__.A object at 0x7f22925a7588>>

In [41]: A.func(1)

        ---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)

<ipython-input-41-398762213524> in <module>()
----> 1 A.func(1)


TypeError: func() missing 1 required positional argument: 'arg'
```

In [42]:
```
instA.func(1)
```

1.2.4 Class attributes

In [43]:
```
class Newton:
    tol = 1e-8 # this is a class attribute
    def __init__(self,f):
        self.f = f # this is not a class attribute
    ...
```

In [44]:
```
N1 = Newton(sin)
N2 = Newton(cos)
```

In [45]:
```
N1.tol
```

Out[45]:
```
1e-08
```

In [46]:
```
N2.tol
```

Out[46]:
```
1e-08
```

In [47]:
```
Newton.tol = 1e-10
```

In [48]:
```
N1.tol
```

Out[48]:
```
1e-10
```

In [49]:
```
N2.tol
```

Out[49]:
```
1e-10
```

In [50]:
```
N2.tol = 1.e-4
N1.tol # still 1.e-10
```

Out[50]:
```
1e-10
```

In [51]:
```
Newton.tol = 1e-5 # now all instances of the Newton classes have 1e-5
N1.tol # 1.e-5
N2.tol # 1e-4 but not N2.
```

Out[51]:
```
0.0001
```

Class Methods

```
In [52]: class Polynomial:
             def __init__(self, coeff):
                 self.coeff = array(coeff)
             @classmethod
             def by_points(cls, x, y):
                 degree = x.shape[0] - 1
                 coeff = polyfit(x, y, degree)
                 return cls(coeff)
             def __eq__(self, other):
                 return allclose(self.coeff, other.coeff)
```

```
In [53]: p1 = Polynomial.by_points(array([0., 1.]), array([0., 1.]))
         p2 = Polynomial([1., 0.])

         print(p1 == p2)
```

```
True
```

1.3 Subclassing and Inheritance

```
In [54]: class OneStepMethod:
             def __init__(self, f, x0, interval, N):
                 self.f = f
                 self.x0 = x0
                 self.interval = [t0, te] = interval
                 self.grid = linspace(t0, te, N)
                 self.h = (te - t0) / N

             def generate(self):
                 ti, ui = self.grid[0], self.x0
                 yield ti, ui
                 for t in self.grid[1:]:
                     ui = ui + self.h * self.step(self.f, ui, ti)
                     ti = t
                     yield ti, ui

             def solve(self):
                 self.solution = array(list(self.generate()))

             def plot(self):
                 plot(self.solution[:, 0], self.solution[:, 1])

             def step(self, f, u, t):
                 raise NotImplementedError()
```

```
In [55]: class ExplicitEuler(OneStepMethod):
             def step(self, f, u, t):
                 return f(u, t)
```

```
In [56]: class MidPointRule(OneStepMethod):
             def step(self, f, u, t):
                 return f(u + self.h / 2 * f(u, t), t + self.h / 2)
```

```
In [57]: def f(x, t):
             return -0.5 * x

         euler = ExplicitEuler(f, 15., [0., 10.], 20)
         euler.solve()
         euler.plot()
         hold(True)
         midpoint = MidPointRule(f, 15., [0., 10.], 20)

         midpoint.solve()
         midpoint.plot()
```

```
/home/claus/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:7: MatplotlibDeprecati
    Future behavior will be consistent with the long-time default:
    plot commands add elements without first clearing the
    Axes and/or Figure.
  import sys
/home/claus/anaconda3/lib/python3.6/site-packages/matplotlib/__init__.py:805: MatplotlibDeprec
  mplDeprecation)
/home/claus/anaconda3/lib/python3.6/site-packages/matplotlib/rcsetup.py:155: MatplotlibDepreca
  mplDeprecation)
```

```
In [58]: argument_list = [f, 15., [0., 10.], 20]
         euler = ExplicitEuler(*argument_list)
         midpoint = MidPointRule(*argument_list)
```

```
In [59]: class ExplicitEuler(OneStepMethod):
             def __init__(self,*args, **kwargs):
                 self.name='Explicit Euler Method'
                 super(ExplicitEuler, self).__init__(*args,**kwargs)
             def step(self, f, u, t):
                 return f(u, t)
```

1.4 Encapsulation

```
In [60]: class Function:
             def __init__(self, f):
                 self.f = f
             def __call__(self, x):
                 return self.f(x)
```

```
        def __add__(self, g):
            def sum(x):
                return self(x) + g(x)
            return type(self)(sum)
        def __mul__(self, g):
            def prod(x):
                return self.f(x) * g(x)
            return type(self)(prod)
        def __radd__(self, g):
            return self + g
        def __rmul__(self, g):
            return self * g
```

```
In [61]: T5 = Function(lambda x: cos(5 * arccos(x)))
         T6 = Function(lambda x: cos(6 * arccos(x)))
```

```
In [62]: import scipy.integrate as sci

         weight = Function(lambda x: 1 / sqrt((1 - x ** 2)))
         [integral, errorestimate] = \
                 sci.quad(weight * T5 * T6, -1, 1) # [7.7e-16, 4.04e-14)

         integral, errorestimate
```

```
Out[62]: (6.510878470473995e-17, 1.3237018925525037e-14)
```

1.5 Classes as decorators

```
In [63]: class echo:
             text = 'Input parameters of {name}\n'+\
                    'Positional parameters {args}\n'+\
                    'Keyword parameters {kwargs}\n'
             def __init__(self, f):
                 self.f = f
             def __call__(self, *args, **kwargs):
                 print(self.text.format(name = self.f.__name__,
                     args = args, kwargs = kwargs))
                 return self.f(*args, **kwargs)
```

```
In [64]: @echo
         def line(m, b, x):
             return m * x + b
```

```
In [65]: line(2., 5., 3.)
```

```
Input parameters of line
Positional parameters (2.0, 5.0, 3.0)
Keyword parameters {}
```

Out[65]: 11.0

In [66]: line(2., 5., x=3.)

```
Input parameters of line
Positional parameters (2.0, 5.0)
Keyword parameters {'x': 3.0}
```

Out[66]: 11.0

In [67]:
```python
class CountCalls:
    """Decorator that keeps track of the number of times
       a function is called."""
    instances = {}
    def __init__(self, f):
        self.f = f
        self.numcalls = 0
        self.instances[f] = self
    def __call__(self, *args, **kwargs):
        self.numcalls += 1
        return self.f(*args, **kwargs)
    @classmethod
    def counts(cls):
        """Return a dict of {function: # of calls} for all
           registered functions."""
        return dict([(f.__name__, cls.instances[f].numcalls)
                     for f in cls.instances])
```

In [68]:
```python
@CountCalls
def line(m, b, x):
    return m * x + b
@CountCalls
def parabola(a, b, c, x):
    return a * x ** 2 + b * x + c
line(3., -1., 1.)
parabola(4., 5., -1., 2.)
CountCalls.counts() # returns {'line': 1, 'parabola': 1}
```

Out[68]: {'line': 1, 'parabola': 1}

In [69]:
```python
parabola.numcalls # returns 1
```

Out[69]: 1