Programação I / Introdução à Programação

Apêndice D, "Classes and Objects"

João Pedro Pedroso

2024/2025

Última aula

- Programação orientada a objetos

Hoje:

- Programação orientada a objetos (cont.).

Programação orientada a objetos: exemplo com frações

- Classe para representar frações:
  - métodos: operações habituais com frações
- Frações:
  - dois inteiros: numerador e denominador
  - devem ser dados ao *construtor*

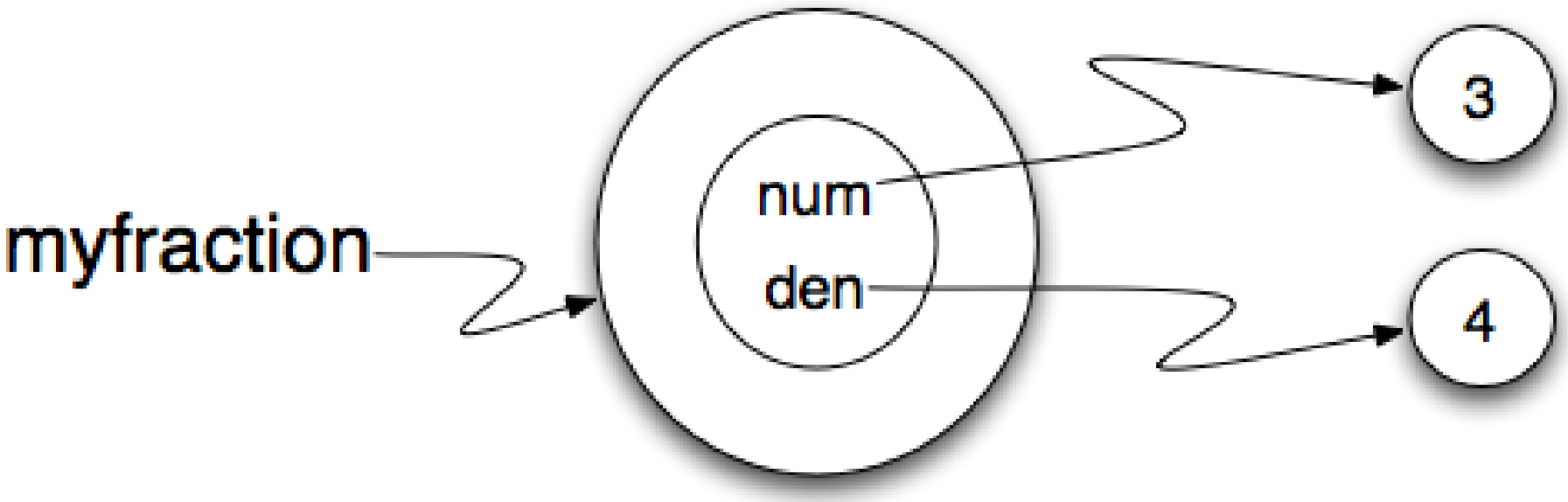
Fração: implementação parcial

```
class Fraction:
    def __init__(self, top, bottom):
        self.num = top          # the numerator is on top
        self.den = bottom       # the denominator is on the bottom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def getNum(self):
        return self.num

    def getDen(self):
        return self.den
```

- output:

```
>>> myfraction = Fraction(3,4)
>>> print(myfraction)
3/4
>>> myfraction.getNum()
3
>>> myfraction.getDen()
4
```

Estado e métodos: `Fraction`



Objectos são mutáveis

- Podemos alterar os atributos de uma instância:

```
myfraction.num = 5
myfraction.den = 3
```

- Podemos fazer o mesmo em métodos da classe, utilizado `self.num` e `self.den`
- Situação em que isto é útil: simplificação de frações
  - exemplo: 12/16 e 6/8 podem ambas ser representadas por fração irredutível: 3/4
- Redução: divisão do numerador e do denominador pelo *maior divisor comum*

Algoritmo de Euclides:

- Para reduzir uma fracção aos termos mais simples:
  - encontrar o maior divisor comum (MDC)
  - dividir numerador e denominador por MDC
  - (podemos reduzir objectos `Fraction` sempre que são criados)
- Algoritmo de Euclides:

  Se n divide m, então n é o MDC. Caso contrário, o MDC é o MDC a n e ao resto da divisão de m por n.

Algoritmo de Euclides:

- Para reduzir uma fracção aos termos mais simples:
  - encontrar o maior divisor comum (MDC)
  - dividir numerador e denominador por MDC
  - (podemos reduzir objectos `Fraction` sempre que são criados)
- Algoritmo de Euclides:

  Se n divide m, então n é o MDC. Caso contrário, o MDC é o MDC a n e ao resto da divisão de m por n.

- Versão recursiva

```
1  def gcd (m, n):
2    if m % n == 0:
3      return n
4    else:
5      return gcd(n, m%n)
```

- Versão iterativa:

```
1  def gcd(m,n):
2      while m%n != 0:
3          m, n = n, m%n
4      return n
```

- Utilização:

```
>>> gcd(12,16)
4
```

Simplificação de frações

- Com a função `gcd` podemos implementar um método para simplificar frações:

```
class Fraction:
    [... ]
    def simplify(self):
        common = gcd(self.num, self.den)

        self.num = self.num // common
        self.den = self.den // common
```

- Notas:
  - `gcd` é uma função, não é um método da classe *(helper function)*;
  - o método `simplify` não devolve nada, apenas altera o objeto *(mutator method)*;

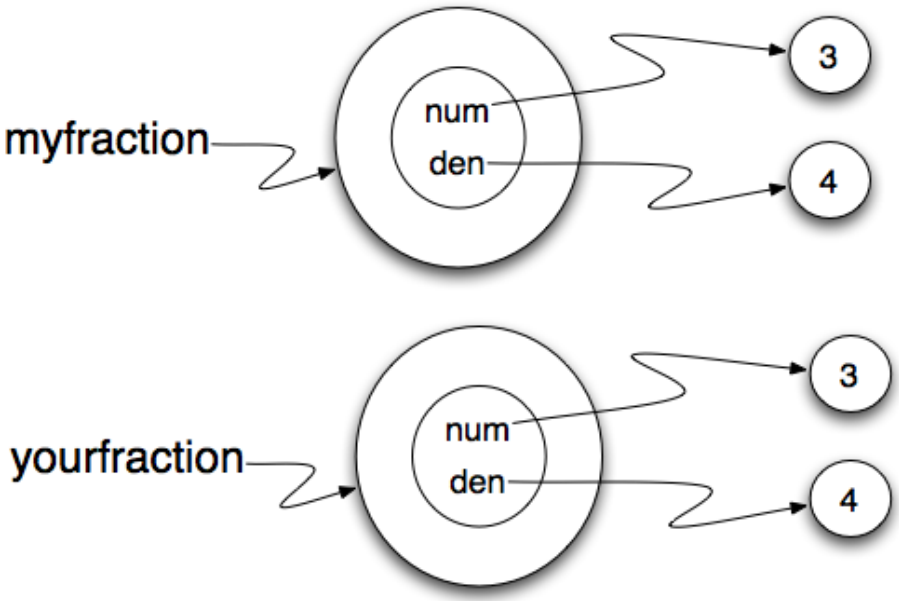
Utilização

```
>>> a = Fraction(5,10)
>>> a.simplify()
>>> print(a)
1/2
```

Identicidade de dois objetos

- Tal como na realidade, dois objetos podem ser iguais sem serem o mesmo; esta ambiguidade existe também em objetos Python;
- Para verificar se duas referências dizem respeito ao mesmo objeto, utiliza-se o operador `is`:

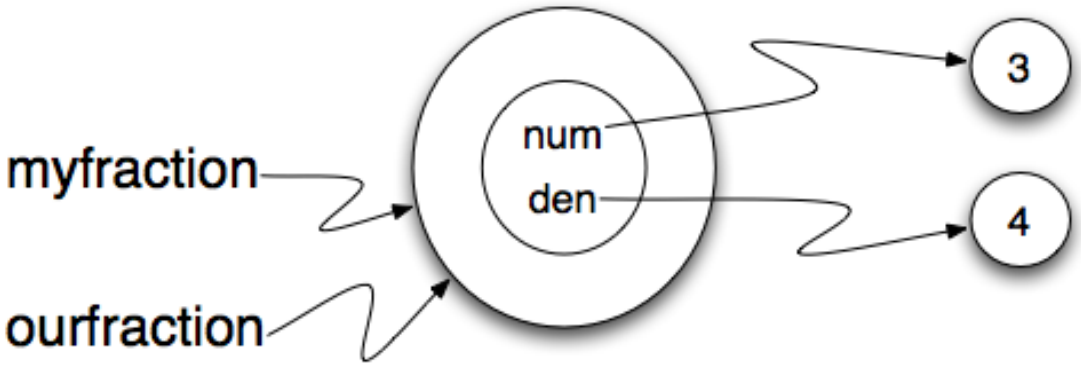
```
>>> myfraction = Fraction(3,5)
>>> yourfraction = Fraction(3,5)
>>> myfraction is yourfraction
False
```



Igualdade superficial de dois objetos

- Se fizermos a atribuição `myfraction = yourfraction`, as variáveis passam a ser aliases para o mesmo objecto
- Chama-se a isso *igualdade superficial* — shallow equality

```
>>> ourfraction = myfraction
>>> myfraction is ourfraction
True
```



Comparação de dois objetos

- O operador ==, se não for redefinido, devolve a igualdade superficial:

```
>>> a = Fraction(3,5)
>>> b = Fraction(3,5)
>>> a == b
False
```

- Para compararmos o *conteúdo* de dois objectos podemos definir uma função:

```
def sameFraction(f1,f2):
    return (f1.getNum() == f2.getNum()) \
           and (f1.getDen() == f2.getDen())
>>> sameFraction(a, b)
True
```

  - *igualdade profunda* — deep equality

Comparação e os seus riscos

- Python permite-nos escolher o significado dos operadores
  - tais como ==

```
p = Point(4, 2)
s = Point(4, 2)
print("== on Points returns", p == s)  # shallow equality test here

a = [2,3]
b = [2,3]
print("== on lists returns",  a == b)  # deep equality test on lists
```

- em `Point` comparação com == devolve `False`;
- em `list` comparação com == devolve `True`.
- veremos à frente como se pode redefinir == na classe `Point`

Recomendação:

Beware of ==

> *"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean — neither more nor less."*

Alice in Wonderland

Parentesis: Cópia superficial de objectos

- Para duplicarmos um objecto (cópia *shallow*):

```
>>> import copy
>>> a = Fraction(4,5)
>>> b = copy.copy(a)
>>> a == b
False
>>> sameFraction(a,b)
True
```

Parentesis: Cópia profunda de objectos

- Cópia em profundidade: `deepcopy` (copia o objecto, e todos objectos nele embebidos):

```
>>> a = [[1,2,3],4]
>>> b = copy.copy(a)
>>> a[0][1] = 999
>>> b
[[1, 999, 3], 4]
```

```
>>> a = [[1,2,3],4]
>>> b = copy.deepcopy(a)
>>> a[0][1] = 999
>>> b
[[1, 2, 3], 4]
```

- Com `deepcopy`, `a` e `b` ficam objectos completamente separados.

Fração: mais métodos

Recordemos implementação inicial:

```
class Fraction:
    def __init__(self,top,bottom):
        self.num = top          # the numerator is on top
        self.den = bottom       # the denominator is on the bottom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def getNum(self):
        return self.num

    def getDen(self):
        return self.den

    def simplify(self):
        common = gcd(self.num, self.den)
        self.num = self.num // common
        self.den = self.den // common
```

Operações aritméticas

- Podemos definir uma função que calcula a soma de duas frações:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + cb}{bd}$$

```
class Fraction:
    [...]
    def add(self,otherfraction):

        newnum = self.num*otherfraction.den + self.den*otherfraction.num
        newden = self.den * otherfraction.den

        common = gcd(newnum,newden)

        return Fraction(newnum//common,newden//common)
```

- Utilização:

```
>>> f1 = Fraction(1,2)
>>> f2 = Fraction(1,4)
>>> f3 = f1.add(f2)
>>> print(f3)
3/4
```

Operações aritméticas: métodos especiais

- Podemos definir um método especial para a adição de duas frações com o operador '+', que torna a utilização de frações mais natural;
- O método existente para isso é __add__:

```
class Fraction:
    [...]
    def __add__(self,otherfraction):

        newnum = self.num*otherfraction.den + self.den*otherfraction.num
        newden = self.den * otherfraction.den

        common = gcd(newnum,newden)

        return Fraction(newnum//common,newden//common)
```

- Utilização:

```
>>> f1 = Fraction(1,2)
>>> f2 = Fraction(1,4)
>>> f3 = f1 + f2
>>> print(f3)
3/4
```

Operações aritméticas: métodos especiais

Método	Resultado
`__add__`	`self + other`
`__sub__`	`self - other`
`__mul__`	`self * other`
`__truediv__`	`self / other`
`__floordiv__`	`self // other`
...	

Operadores relacionais: métodos especiais

Método	Uso	Descrição
`__lt__`	`x < y`	devolve `True` se x for menor do que y
`__le__`	`x <= y`	devolve `True` se x for menor do que, ou igual a y
`__eq__`	`x == y`	devolve `True` se x for igual a y
`__ne__`	`x != y`	devolve `True` se x for diferente y
`__ge__`	`x >= y`	devolve `True` se x for maior do que, ou igual a y
`__gt__`	`x > y`	devolve `True` se x for maior do que y

Métodos especiais

- A lista completa de métodos especiais pode ser consultada em: `https://docs.python.org/3/reference/datamodel.html#special-method-names`
- Incluí metodos para:
  - Adaptação/personalização de tipos *(type customization)*;
  - Comparação/relação entre objetos;
  - Controlar acesso a atributos da classe;
  - Implementar tipos compostos e o respetivo acesso (e.g., mapas/dicionários personalizados);
  - Emular tipos e conversões numéricos;
  - ...

Programação orientada a objetos: exemplo completo

- Classe para representar polinómios
- Construtor: parametrizado pelos coeficientes
  - passados como uma lista
  - índice $i \rightarrow$ coeficiente de x^i
  - exemplo: `Polynomial([1,0,-1,2])`

$$1 + 0 \cdot x - 1 \cdot x^2 + 2 \cdot x^3 = 1 - x^2 + 2x^3$$

- Operadores:
  - adição, multiplicação, ...
  - operador `()` ("chamar") $\rightarrow$ avaliar o polinómio

```
p = Polynomial([1,0,-1,2])
y = p(2)
print(y)   # 1 - 3**2 + 2*3**3 = 46
```

Construtor

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

Operador ()

- Avaliar o polinómio:

$$\sum_{i=0}^{n} c_i x^i$$

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

Pretty printing

```
class Polynomial:
    [...]
    def __str__(self):
        s = []
        for i in range(len(self.coeff)):
            if self.coeff[i] != 0:
                s.append('{}*x^{}'.format(self.coeff[i], i))

        return " + ".join(s)
```

Operador +

- Para todos os índices relevantes: $d_i = c_i + c'_i$

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

Operador -

Operador *

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

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

Operador de diferenciação

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

```
class Polynomial:
    [...]
    def differentiate(self):
        """Differentiate this polynomial in-place."""
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i*self.coeff[i]
        del self.coeff[-1]
```

Derivada de um polinómio

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

```
class Polynomial:
    [...]
    def derivative(self):
        """Copy this polynomial and return its derivative."""
        dpdx = Polynomial(self.coeff[:])  # make a copy
        dpdx.differentiate()
        return dpdx
```

Utilização

```
>>> from polynomial import Polynomial
>>> p1 = Polynomial([1, -1])
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = p1 + p2
>>> print(p3)
1*x^0 + 0*x^1 + 0*x^2 + 0*x^3 + -6*x^4 + -1*x^5
>>> p4 = p1*p2
>>> print(p4)
0*x^0 + 1*x^1 + -1*x^2 + 0*x^3 + -6*x^4 + 5*x^5 + 1*x^6
>>> p5 = p2.derivative()
>>> print(p5)
1*x^0 + 0*x^1 + 0*x^2 + -24*x^3 + -5*x^4
```

Noções estudadas esta semana

classe tipo composto definido pelo programador

instanciar criar um objeto de uma determinada classe

instância o mesmo que objeto

objeto variável de um determinado tipo (a sua classe), frequentemente utilizada para modelar uma coisa ou conceito do mundo real; junta *informação* (dados) e *operações* relevantes para esse tipo composto de dados

construtor cada classe é uma *fábrica* de objetos; se tiver definido um método de inicialização, esse método será executado ao criar um objeto, para dar um valor correto aos atributos

método de inicialização em Python: `__init__`

método função definida dentro de uma classe, que é invocada partindo de instâncias dessa classe

atributo um dos dados que constituí uma instância

Noções estudadas esta semana

programação orientada a objetos estilo de programação em que dados e operações que os manipulam são definidos em classes e métodos

linguagem orientada a objetos linguagem com capacidades de programação orientada a objetos

igualdade superficial *(shallow equality)* igualdade de referências: verifica se duas variáveis se referem ao mesmo objeto

igualdade profunda *(deep equality)* igualdade dos valores guardados num objeto

cópia superficial *(shallow copy)* cópia do conteúdo de um objeto e das *referências* a outros objetos nele imbricados

cópia profunda *(deep copy)* cópia do conteúdo de um objeto e, recursivamente, de todos os objetos que nele estão definidos

Noções estudadas esta semana

- métodos "mágicos":

Método	Uso	Descrição
`__add__`	`x + y`	soma
`__sub__`	`x - y`	subtração
`__mul__`	`x * y`	multiplicação
`__truediv__`	`x / y`	divisão
`__floordiv__`	`x // y`	divisão inteira
...		
`__lt__`	`x < y`	`True` se x for menor do que y
`__le__`	`x <= y`	`True` se x for menor do que, ou igual a y
`__eq__`	`x == y`	`True` se x for igual a y
`__ne__`	`x != y`	`True` se x for diferente y
`__ge__`	`x >= y`	`True` se x for maior do que, ou igual a y
`__gt__`	`x > y`	`True` se x for maior do que y

Aula de hoje:

- Programação orientada a objetos, exemplo baseado em:
  A Primer on Scientific Programming with Python
  Hans Petter Langtangen
- Ver também este link: polynomial.py

Próxima aula

- Exceções