Aula 04: 14/AGO/2018

Aula passada

Simulação da classe `Fracao`:

- método `__init__()`
- método `__str__()`
- método `__add__()`

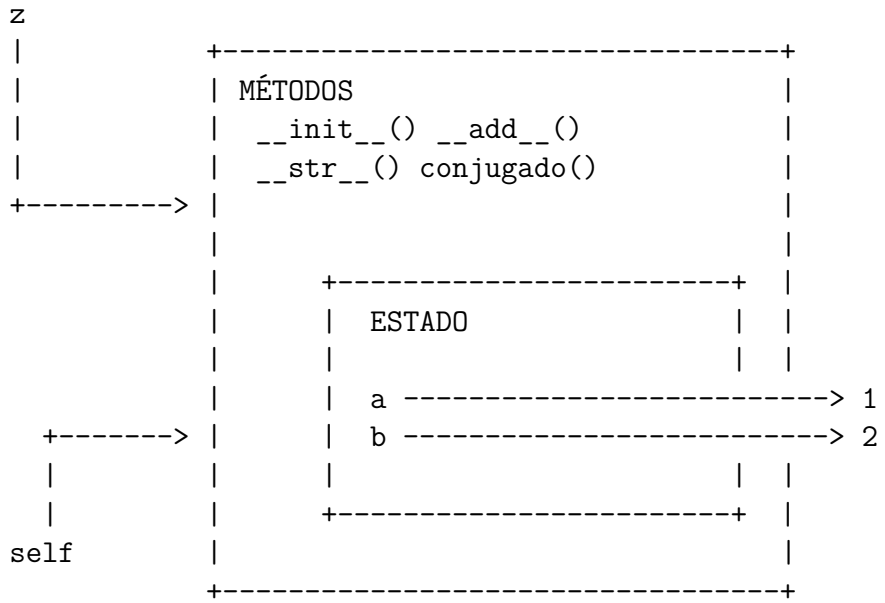
Hoje

- Revisão mostrando alguns métodos da classe `Complexo`:
  - mostrar como desejamos a classe se comporte;
  - `__init__()`;
  - `__str__()`;
  - `__add__()`; e
  - comentar que existem: `__sub__()`, `__mul__()`, `__truediv__()`, `__eq__()`, etc.
- Construção de uma classe `Polinomio`.

Classe Complexo

```
Esqueleto da classe Complexo.
```

```
z = Complexo(1,2)
```



Uma implementação

```
class Complexo:
    def __init__(self, real, imag):
        '''(Complexo, int/float, int/float) -> None

        Usado pelo contrutor para montar um número complexo.
        '''
        self.re = real
        self.im = imag

    def __str__(self):
        '''(Complexo) -> str

        Recebe um referência `self` a um Complexo e retorna
        o string usado por print() para exibi-lo.
        '''
        s = str(self.re)
        if self.im != 0:
            s += "+" + str(self.im) + "i"
        return s

    def __eq__(self, other):
        '''(Complexo, Complexo) -> bool

        Recebe referência `self` e `other`para números complexos
        e retorna True se forem iguais e False em casocontrário.

        Usado pelo Python quando escrevemos "Complexo == Complexo"
        '''
        return self.re == other.b and self.im == other.b


    def __add__(self, other):
        '''(Complexo, Complexo) -> Complexo
        '''
        real = self.re + other.a
        imag = self.im + other.b
        return Complexo(real, imag)

    def __sub__(self, other):
        '''(Complexo, Complexo) -> Complexo
        '''
        real = self.re - other.a
        imag = self.im - other.b
        return Complexo(real, imag)

    def __mul__(self, other):
        '''(Complexo, Complexo) -> Complexo
        '''
```

```
        real = self.re*other.a - self.im*other.b
        imag = self.re*other.b + self.im*other.a
        return Complexo(real, imag)

    def __truediv__(self, other):
        other_conj = other.conjugado()
        z_num = self  * other_conj
        z_den = other * other_conj
        real  = z_num.a / z_den.a
        imag  = z_num.b / z_den.a
        return Complexo(real, imag)


    def conjugado(self):
        '''(Complexo) -> Complexo
        '''
        return Complexo(self.re, -self.im)
```

Problema Polinomio

Escreva um programa que leia:

- um inteiro `k`;
- os `k+1` coeficientes de um polinômio de grau `k`;
- um inteiro `n`; e
- outros `n` números reais,

e imprima o valor do polinômio e de suas derivadas primeira e segunda para cada um desses `n` números.

Observação: Talvez seja melhor ler apenas os coeficientes do polinômio e usar `split()`.

```
# Polinomio.__init__(), Polinomio.__str__(), Polinomio.__call__(), Polinomio.derive()
from polinomio import Polinomio

#---------------------------------------------------------------------
def main():
    grau = int(input("Digite o grau do polinônio: "))

    coef =[]
    i = 0
    while i < grau+1:
        print("Digite o coeficiente de x^%d: " %(i), end='')
        valor = int(input())
        coef.append(valor)
        i += 1

    # crie um objeto da classe Polinomio
    p = Polinomio(coef)
    print("Polinomio p: ", p)

    # crie um Polinomio que represente a derivada de p
    dp = p.derive()
    print("Derivada primeira de p: ", dp)

    # crie um Polinomio que represente a derivada de dp
    ddp = dp.derive()
    print("Derivada segunda  de p: ", ddp)

    # leia os valores x_0,...,x_n-1 e calcule os valores dos polinômios
    n = int(input("Digite n: "))
    i = 0
    while i < n:
        x = float(input("Digite x%d: " %(i)))
        print("p(%f) = %f" %(x,p(x)))
        print("dp(%f) = %f" %(x,dp(x)))
        print("ddp(%f) = %f" %(x,ddp(x)))
        i += 1

#-----------------------
main()
```

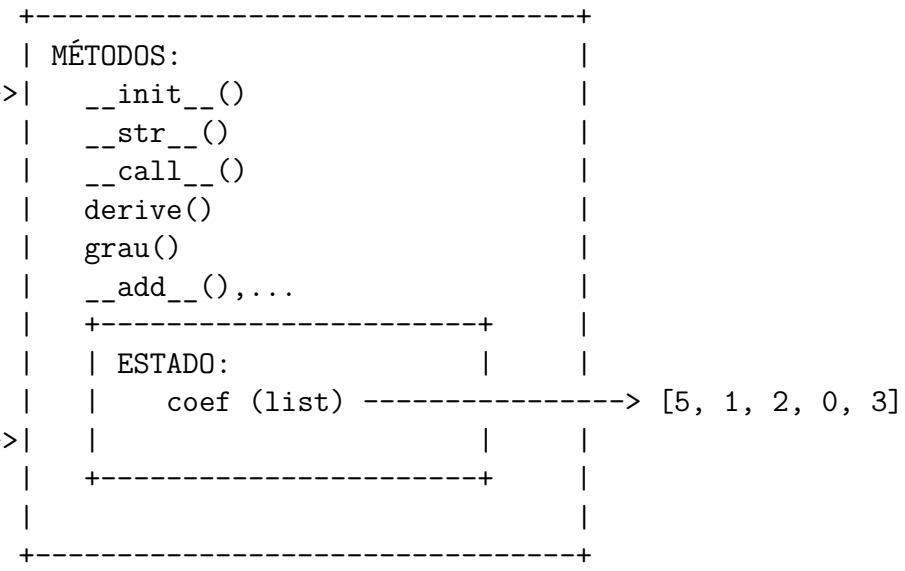
Polinômios

Um polinômio de uma variável pode ser representado pela lista de seus coeficientes. Por exemplo, o polinômio $5 + x + 2x^2 + 3x^4$ é representado pela lista `[5, 1, 2, 0, 3]`. Nesta representação:

- o número na posição `0` é o coeficiente de x^0, o número na posição `1` é o coeficiente de x^1. Em geral, o número na posição `i` é o coeficiente de x^i;
- se o polinômio é diferente de `0`, então o comprimento da lista é o grau do polinômio mais 1 e o último elemento da lista é diferente de 0;
- o polinômio 0 é representado pela lista vazia `[ ]`.

classe `Polinomio`

```
p = Polinomio([5, 1, 2, 0, 3])
```



Classe Polinomio

```
class Polinomio:
    #--------------------------------------------------------------------
    def __init__(self, coef = []):
        '''(Polinomio, list) -> None

        Chamado pelo construtor da classe:
           Recebe uma referência/apelido para um objeto
               `self` a ser construído e uma referência/apelido
               `coef` para os coeficientes de um polinômio e cria
           e retorna um Polinomio.

        Decisões de projeto:

            * valor de self.coef[i] é o coeficiente do termo x^i;
```

```
        * polinômio nulo é representado pela lista [];
        * coeficiente do termo dominante de um polinômio não nulo
           é diferente de zero; e, portanto
        * len(self.coef)-1 é o grau do polinômio e o grau do polinômio
            nulo é -1.
    '''
    # perigo, coef é um apelido para a lista
    # determine o grau do polinômio
    grau = len(coef)-1
    while grau > -1 and coef[grau] == 0:
        grau -= 1
    # crie um clone
    coef_clone = []
    i = 0
    while i < grau+1:
        coef_clone.append(coef[i])
        i += 1
    # atualize o atributo coef
    self.coef = coef_clone

#-----------------------------------------------------------------
def __str__(self):
    '''(self) -> str

    Recebe uma referência `self` para um Polinomio e cria
    e retorna um string que representa o polinômio.

    Usado pelo Python quando usamos print() em um Polinomio.
    Também é usado pelo Python quando usamos str().

    '''
    s = ''
    coef = self.coef
    grau = self.grau() # len(coef)-1
    if grau == -1:
        s += '0'
    else:
        # encontre primeiro coeficiente não nulo
        i = 0
        while coef[i] == 0:
            i += 1
        # aqui coef[i] != 0
        s += str(coef[i])
        if i > 0:
            s += "*x^%d" %i
        i += 1
        while i < grau+1:
            valor = coef[i]
            if   valor > 0:
                s += " + " + str(valor) + "x^%d" %i
```

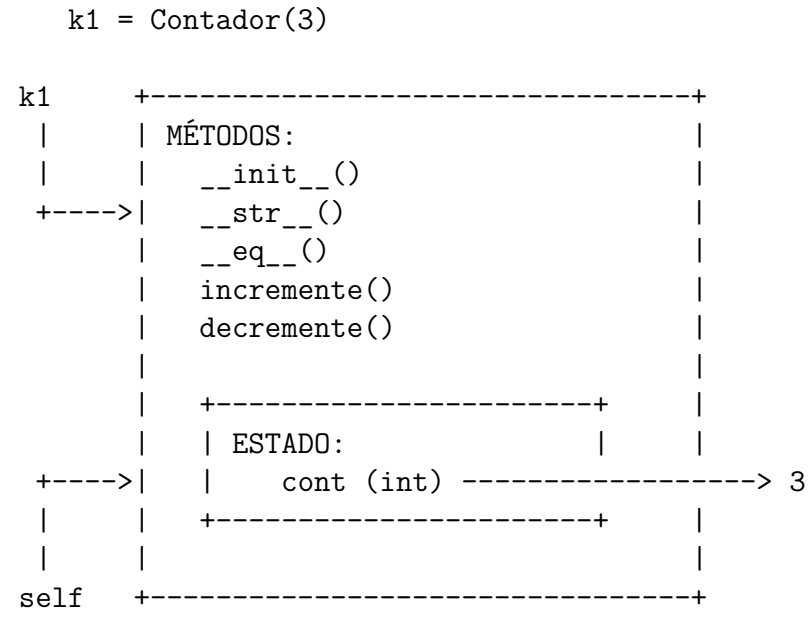
```
            elif valor < 0:
                s += " - " + str(-valor) + "x^%d" %i
            i += 1
        return s

    #-------------------------------------------------------------------
    def __call__(self, x):
        '''(Polinomio, número) -> número

        Recebe uma referência/apelido `self` a um Polinomio e
            um número x.
            Calcula e retorna o valor do polinômio em x.

        Usado pelo Python quando chamamos um  Polinomio com um
            argumento: p(x).
        '''
        valor = 0
        coef = self.coef
        grau = self.grau() # len(coef)-1
        i = 0
        xi = 1
        while i < grau+1:
            valor += coef[i] * xi
            xi *= x
            i += 1
        return valor

    #-------------------------------------------------------------------
    def __add__(self, other):
        '''(Polinomio, Polinomio) -> Polinomio

        Recebe referências/apelidos  `self` e `other` para Polinomios
        e cria e retorna um Polinomio que é sua soma.

        Usado pelo Python quando escrevemos `Polinomio + Polinomio`.

        A solução do problema proposto não usa esse método.
        '''
        # apelidos
        coef1 = self.coef
        grau1 = self.grau() # len(coef1)-1
        coef2 = other.coef
        grau2 = other.grau() # len(coef2)-1

        # calcule os coeficientes da soma dos polinômios
        coef_soma = []
        i = 0
        while i < grau1+1 or i < grau2+1:
            valor1 = 0
            valor2 = 0
```

```
        if i < grau1+1: valor1 = coef1[i]
        if i < grau2+1: valor2 = coef2[i]
        valor = valor1 + valor2
        coef_soma.append(valor)
        i += 1

    soma = Polinomio(coef_soma)
    return soma

#-----------------------------------------------------------------
def grau(self):
    '''(Polinomio) -> int

    Recebe uma referência/apelido `self` para um Polinomio
    e retorna o seu grau.
    '''
    return len(self.coef) - 1

#-----------------------------------------------------------------
def derive(self):
    '''(Polinomio) -> Polinomio

    Recebe uma referência/apelido 'self' para um Polinomio
    e retorna um Polinomio que representa a sua derivada.
    '''
    # apelidos
    coef    = self.coef
    grau    = self.grau() # len(self.coef)-1
    coef_dp = []
    i = 1
    while i < grau+1:
        # calcule o coeficiente de x^(i-1) da derivada
        valor = i*coef[i]
        coef_dp.append(valor)
        i += 1
    # crie um polinomio
    dp = Polinomio(coef_dp)
    return dp
```

Apêndice

Classe `Contador`

Atributos



`main()`

```
# importa a classe Contador
from contador import Contador

def main():
    # crie contadores: __init__() é invocado
    k1 = Contador()
    k2 = Contador(3)

    # imprima contadores: __str__() é invocado
    print("k1: ", k1)
    print("k2: ", k2)

    # chamada de métodos
    k1.incremente()
    k2.decremente()
    k2.decremente()

    # imprima contadores: __str__() é invocado
    print("k1: ", k1)
    print("k2: ", k2)

    # hmmmm, para quem achar que dá tempo
    # comparação de contadores: __eq__() é invocado
    if k1 == k2:
        print("iguais")
```

```
    else:
        print("diferentes")


#-----------------------------------------------------
main()
```

Classe `Contador`

```
#-----------------------------------------------------
# módulo contador.py
class Contador:
    #-------------------------------------------------
    def __init__(self, ini = 0):
        '''(Contador, int) -> None

        Construtor: contrói um objeto da classe Contador.

        Método mágico/especial: não tem return
        '''
        self.cont = ini

    #-------------------------------------------------
    def __str__(self):
        '''(Contador) -> str

        Recebe uma referência/apelido `self` a um objeto da classe
        Contador e retorna um string que pode ser usado por
        print() para exibi-lo.

        Método mágico/especial: usado por print() e str()
        '''
        s = "Contador: valor = %d" %(self.cont)
        return s

    #-------------------------------------------------
    def __eq__(self,other):
        '''(Contador,Contador) -> bool

        Recebe referências/apelidos `self` e `other` a objetos
        da classe Contador e retorna True se "são iguai" e
        False em caso contrário.

        Método mágico/especial: usado pelo Python quando escrevemos
                    Contador == Contador.
        '''
        # return self.cont == other.cont
        iguais = False
        if self.cont == other.cont:
```

```
            iguais = True
        return iguais

    #-----------------------------------------------
    def incremente(self):
        '''(Contador) -> None

        Recebe uma referência/apelido `self` a um Contador e
        incrementa-o de 1.
        '''
        self.cont += 1

    #-----------------------------------------------
    def decremente(self):
        '''(Contador) -> None

        Recebe uma referência/apelido `self` a um Contador e
        decrementa-o de 1.
        '''
        self.cont -= 1
```

Saída

```
meu_prompt> python simulacao1.py
k1:  Contador: valor = 0
k2:  Contador: valor = 3
k1:  Contador: valor = 1
k2:  Contador: valor = 1
iguais
```

Sobrecarga de operadores aritméticos

```
Operador         Método
-----------------------
+          object.__add__(self, other)
-          object.__sub__(self, other)
*          object.__mul__(self, other)
//         object.__floordiv__(self, other)
/          object.__truediv__(self, other)
%          object.__mod__(self, other)
**         object.__pow__(self, other[, modulo])
<<         object.__lshift__(self, other)
>>         object.__rshift__(self, other)
&          object.__and__(self, other)
^          object.__xor__(self, other)
|          object.__or__(self, other)
```

Sobrecarga de operadores relacionais

```
Operador          Método
-----------------------
<          object.__lt__(self, other)
<=         object.__le__(self, other)
==         object.__eq__(self, other)
!=         object.__ne__(self, other)
>          object.__gt__(self, other)
>=         object.__ge__(self, other)
```