4 Aula 04: 13/AGO/2019

4.1 Aula passada

Simulação da classe `Fracao`:

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

```
def __radd__(self, other):
    return self + other # return self.__add__(other)
```

4.2 Hoje

- revisão mostrando alguns métodos da classe Complexo.
  - `__init__()`
  - `__str__()`
  - `__add__()`
  - existem: `__sub__()`, `__mul__()`, `__truediv__()`, `__eq__()`, etc
- métodos da classe Polinomio: `__init__()`, `__str__()`, `derivada()`, `__cal__()`

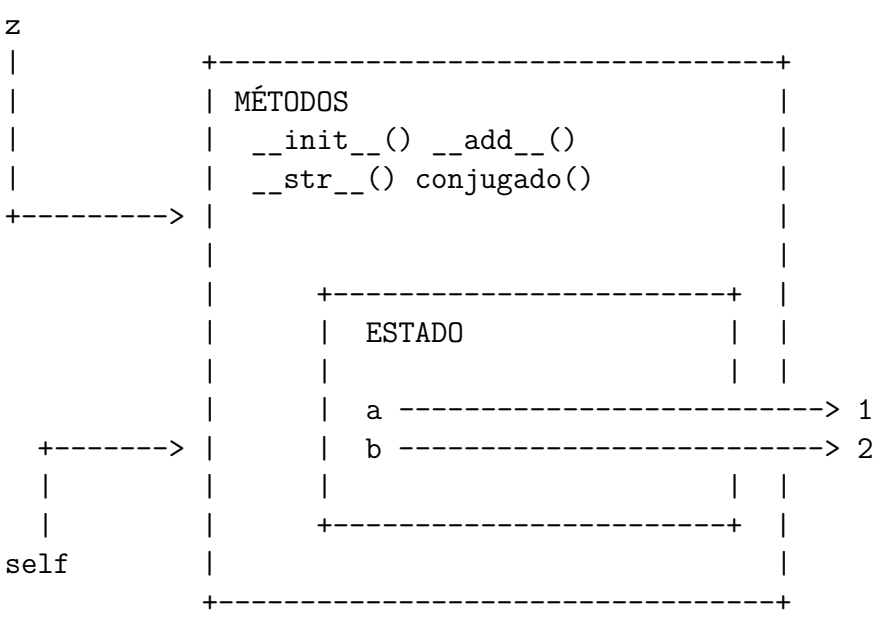
4.3 Cliente

```
def main():
    z1 = Complexo()   # __init__()
    z2 = Complexo(3)  #
    z3 = Complexo(0.5, 2)
    z = z2 + z3 # __add__(), __mul__
    print(z)    # __str__()
```

4.4 Classe Complexo

Esqueleto da classe Complexo.

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



4.5 Complexo: implementação

```
import math

class Complexo:
    def __init__(self, real = 0, imag = 0):
        '''(Complexo, int|float, int|float) -> None

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

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

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

```
    if self.imag != 0:
        s += str(self.imag) + "i"
    return s

def __repr__(self):
    '''(Complexo) -> str '''
    return str(self)

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.real == other.imag and self.imag == other.imag


def __add__(self, other):
    '''(Complexo, Complexo|int|float) -> Complexo

    Recebe Complexos `self` e `other` e retorna a
    sua soma

    Usado pelo Python quando escrevemos "Complexo + [Complexo|int|float]"
    '''
    if type(other) in [int, float]:
        real = self.real + other
        imag = self.imag
    else:
        real = self.real + other.real
        imag = self.imag + other.imag
    return Complexo(real, imag)

def __radd__(self, other):
    '''(Complexo, int|float) -> Complexo

    Recebe Complexos `self` e `other` e retorna a
    sua soma

    Usado pelo Python quando escrevemos "[int|float]+Complexo"
    '''
    return self + other

def __sub__(self, other):
    '''(Complexo, Complexo) -> Complexo

    Usado pelo Python quando escrevemos "Complexo - [Complexo|int|float]"
    '''
```

```
    if type(other) in [int, float]:
        real = self.real - other
        imag = self.imag
    else:
        real = self.real - other.real
        imag = self.imag - other.imag
    return Complexo(real, imag)

def __rsub__(self, other):
    '''(Complexo, int|float) -> Complexo

    Recebe Complexos `self` e `other` e retorna a
    sua soma

    Usado pelo Python quando escrevemos "[int|float]-Complexo"
    '''
    return self - other

def __mul__(self, other):
    '''(Complexo, Complexo|int|float) -> Complexo

    Usado pelo Python quando escrevemos "Complexo * [Complexo|int|float]"
    '''
    if type(other) in [int, float]:
        real = self.real*other
        imag = self.imag*other
    else:
        real = self.real*other.real - self.imag*other.imag
        imag = self.real*other.imag + self.imag*other.real
    return Complexo(real, imag)

def __rmul__(self, other):
    '''(Complexo, Complexo|int|float) -> Complexo

    Usado pelo Python quando escrevemos "[Complexo|int|float]*Complexo"
    '''
    return self * other

def __truediv__(self, other):
    '''(Complexo, Complexo/int/float) -> Complexo

    Usado pelo Python quando escrevemos "Complexo / Complexo|int|float"
    '''
    if type(other) in [int, float]:
        real = self.real / other
        imag = self.imag / other
    else:
        other_conj = other.conjugado()
        z_num = self  * other_conj
        z_den = other * other_conj
```

```
            real  = z_num.real / z_den.real
            imag  = z_num.imag / z_den.real
        return Complexo(real, imag)

    def __rtruediv__(self, other):
        '''(Complexo, Complexo/int/float) -> Complexo

        Usado pelo Python quando escrevemos "Complexo / Complexo|int|float"
        '''
        return self / other

    def norma(self):
        '''(Complexo) -> float

        Retorna a norma do Complexo `self`
        '''
        return math.sqrt(self.real*self.real + self.imag*self.imag)

    def conjugado(self):
        '''(Complexo) -> Complexo
        '''
        return Complexo(self.real, -self.imag)
```

4.6 Cliente

```
from polinomio import Polinomio

def main():
    # crie lista de coeficientes
    coefs = [5, 1, 2, 0, 3]

    # crie um objeto da classe polinomio
    p = Polinomio(coefs) # __init__()
    print("Polinomio p:", p) # __str__()

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

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

    # calcule o valor dos polinômio
    valores = [1, 0.5, 3]
    for x in valores:
        print("p(%f) = %f" %(x, p(x))) # __call__()
        print("dp(%f) = %f" %(x, dp(x)))
        print("ddp(%f) = %f" %(x, ddp(x)))
```

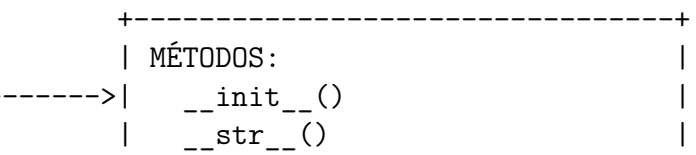
4.7 Polinômios

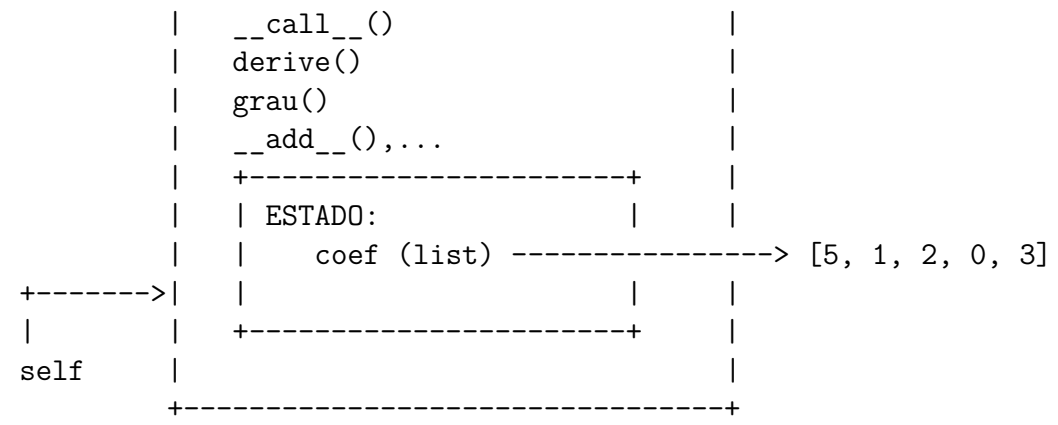
Um polinômio de uma variável pode ser representado pela lista de seus coeficientes. Por exemplo, o polinômio $5 + x + 2x^2 + 3x^4$ é representado pela lista `[5, 1, 2, 0, 3]`. Nesta representação:

- 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 `[ ]`.

4.8 classe `Polinomio`

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





4.9 Polinomio: implementação

```
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.

        Método mágico/especial: usado pelo Python quando
            escrevemos Polinomio() para criar um Polinomio.
            Não tem return.
        '''
        # 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.

    Método mágico/especial: 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.

    Método mágico/especial: 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.

    Método mágico/especial: 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
```

4.10 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)
```

4.11 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)
```