Aula 05: 16/AGO/2018

Aula passada

Implementação parcial de uma classe `Polinomio` e de um cliente dessa classe. Foram escritos os métodos

- método `__init__()`
- método `__str__()`
- método `__call__()`

Hoje

Pilhas usando listas em Python, com os métodos `list.append()` e `list.pop()`

Problema

Decidir se em um dada string de parênteses, colchetes e chaves está bem-formada.

Um string de parênteses, colchetes e chaves é **bem-formada** se se parênteses, colchetes e chaves são fechados na ordem inversa àquela em que foram abertos.

Por exemplo, a primeira das sequências abaixo está bem-formada enquanto a segunda não está:

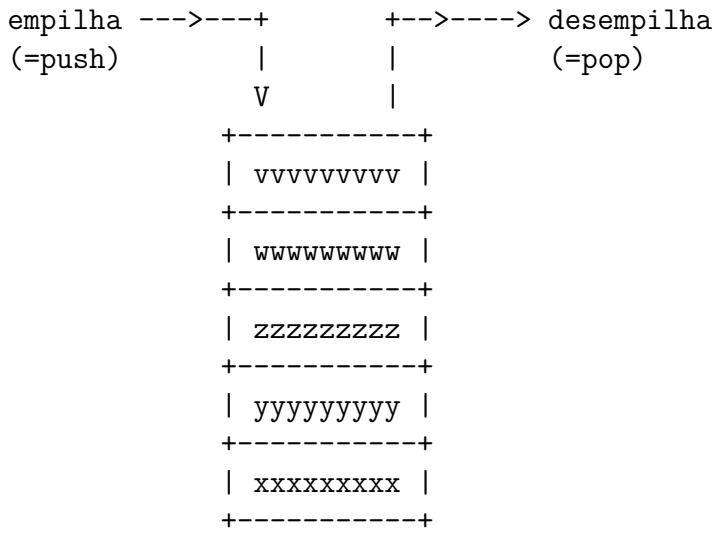
```
( ( ) [ ( ) ] )          ( [ ) ]
```

Pilhas

Uma **pilha** (=*stack*) é uma lista dinâmica em que todas as operações:

- inserções;
- remoções; e
- consultas

são feitas em uma mesma extremidade chamada de **topo**.



Solução

```
.............................................................
# Usando append() e pop()

TESTE = False
PROMPT = "exp >>> "
ABRE_PARENTESES  = "("
FECHA_PARENTESES = ")"
ABRE_CHAVES      = "{"
FECHA_CHAVES     = "}"
ABRE_COLCHETES   = "["
FECHA_COLCHETES  = "]"
ABRE  = "([{"
FECHA = ")]}"
QUIT = ''

def main():
    '''
    Resolve um problema levemente mais geral.

    Recebe uma sequência de strings e para cada string verifica se a substring
    formada apenas pelos seus parênteses, colchetes e chaves é bem formada.
    '''
    print("Verificador de sequencias bem formadas.")
    print("[Tecle ENTER para encerrar o programa.]")
    sequencia = input(PROMPT)
    while sequencia != QUIT:
        if bem_formada(sequencia):
            print("bem-formada: sim")
        else:
            print("bem_formada: não")
        sequencia = input(PROMPT)

#-----------------------------------------------------------
def bem_formada(sequencia):
    ''' (str) -> bool

    Retorna True se a sequência é bem formada e False em caso contrário.
    '''
    pilha = []

    for item in sequencia:
        if item in [ABRE_PARENTESES,ABRE_COLCHETES,ABRE_CHAVES]: #
            pilha.append(item)
        elif item in [FECHA_PARENTESES,FECHA_COLCHETES,FECHA_CHAVES]:
            if pilha == []: # pilha vazia -- erro frequente
                return False
            # verifique se o topo da pilha tem o abre certo
            item_topo = pilha.pop()
```

```
            if   item == FECHA_PARENTESES and item_topo != ABRE_PARENTESES:
                return False
            elif item == FECHA_COLCHETES  and item_topo != ABRE_COLCHETES:
                return False
            elif item == FECHA_CHAVES     and item_topo != ABRE_CHAVES:
                return False

    if len(pilha) > 0: ### pilha não vazia -- erro frequente
        return False

    # passou pelos testes
    return True


# outra versão que usa o modo index()
#-----------------------------------------------------
def bem_formada(sequencia):
    ''' (str) -> bool

    Recebe um string e verifica se a substring formada apenas por parênteses,
    chaves e colchetes da string é bem-formada.
    Retorna True se a sequência é bem formada e False em caso contrário.
    '''
    pilha = []

    for item in sequencia:
        if item in ABRE:
            pilha.append(item)
        elif item in FECHA:
            if pilha == []: # pilha vazia -- erro frequente
                return False
            # verifique se o topo da pilha tem o abre certo
            item_topo = pilha.pop()
            if ABRE.index(item_topo) != FECHA.index(item):
                return False

    if len(pilha) > 0: # pilha não vazia -- erro frequente
        return False

    # passou pelos testes
    return True

#-------------------------------------------
if __name__ == "__main__":
    main()
```

Problema

Dado um número inteiro representado na base 10 exibir a sua representação na base 2.

Representação de números

Na **base decimal** são usados apenas os dígitos $0, 1, \ldots, 9$:

- $1_{10} = 1 \times 10^0$
- $2_{10} = 1 \times 10^0$
- $3_{10} = 1 \times 10^0$
- $9_{10} = 9 \times 10^0$
- $89_{10} = 8 \times 10^1$
- $1234_{10} = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^3 + 4 \times 10^0$

Na **base binária** são usados apenas os dígitos 0 e 1:

- $1_2 = 1 \times 2^0$
- $10_2 = 1 \times 2^1 + 0 \times 2^0$
- $11_2 = 1 \times 2^1 + 1 \times 2^0$
- $1001_2 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
- $1011001_2 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
- $10011010010_2 = 1 \times 10^{11} + 0 \times 10^{12} + \cdots$

Algoritmo

```
233 | 2
----+----
  1 | 166 | 2                                        1  <-- base
      ----+----
        0 |  58 | 2                                  0
            ----+----
              0 |  29 | 2                            0
                  ----+----
                    1 |  14 | 2                      1
                        ----+----
                          0 |   7 | 2                0
                              ----+----
                                1 |   3 | 2          1
                                    ----+----
                                      1 |   1 | 2    1
                                          ----+----
                                            1 | 0    1   <-- topo

233 = 11101001
```

Solução

```
#
# Conversor de número decimais para binarios
# usando append() e pop() de lista
#

PROMPT = "base 10 >>> "
QUIT = ''

def main():
    '''
    Recebe uma string e verifica se a substring formada apenas dos
    parênteses, colchetes e chaves da string é bem formada.
    '''
    print("Conversor de decimal para binário.")
    print("[Tecle ENTER para encerrar o programa.]")

    decimal_str = input(PROMPT)
    while decimal_str != QUIT:
        decimal = int(decimal_str)
        print("base 2:    ", to_string2(decimal))
        decimal_str = input(PROMPT)

#-----------------------------------------------------
def to_string2(dec):
    ''' (int) -> str

    Recebe um número inteiro e retorna uma string que representa
    o número na base 2.
    '''
    if dec == 0: return '0'
    bin_str = ''
    pilha = []
    negativo = False
    if dec < 0:
        negativo = True
        dec = -dec

    # determine os dígitos de dec na base 2
    while dec > 0:
        dig_2 = dec % 2
        pilha.append(dig_2)
        dec //= 2

    # converta os dígitos para str
    while pilha != []:
        dig_str  = str(pilha.pop())
        bin_str += dig_str
```

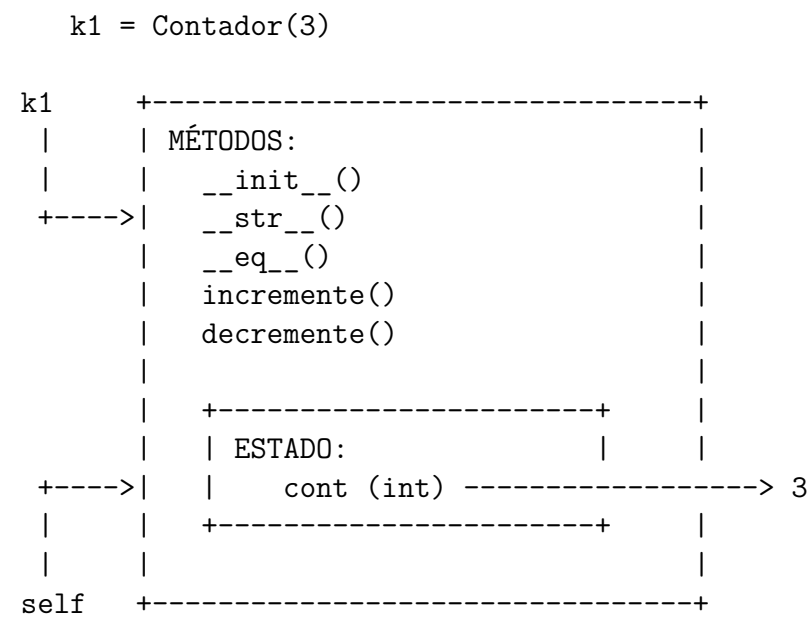
```
    if negativo: bin_str = "-" + bin_str
    return bin_str

#-----------------------------------------
if __name__ == "__main__":
    main()
```

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.
        '''
        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 em Python

```
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 em Python

```
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)
```