Repetição e arranjos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Vimos anteriormente que devemos definir uma estrutura para representar uma informação quando ela consiste de dois ou mais itens que juntos descrevem uma entidade.

- No problema da conversão de segundos para horas, minutos e segundos, definimos a estrutura `Tempo`.
- No problema da loteria, definimos a estrutura `SeisNumeros`.

O `Tempo` era composto de três “itens” que foram representados pelos campos horas, minutos e segundos.

Já para `SeisNumeros` cada item não tinha uma interpretação particular, então não usamos nomes significativos, tivemos que “inventar” os nomes de `a`, ..., `f`.

Como faríamos se ao invés de 6 itens tivéssemos 20? E 1.000? E 1.000.000? Ou ainda, uma quantidade indefinida? E como escrever o código para processar esse tipo de dado?

Vamos ver como fazer essas coisas!

Arranjos

Quando precisamos representar uma coleção de valores da mesma natureza (todos os itens são notas, nomes, pontos, janelas, etc), utilizamos arranjos.

Os arranjos em Python são dinâmicos, isto é, podem mudar de tamanho, e são representados pelo tipo `list`.

Vamos ver algumas operações com listas.

Listas - inicialização e indexação

```
>>> # Inicialização
>>> x: list[int] = [9 + 1, 1, 7, 2]
>>> x
[10, 1, 7, 2]

>>> # Lista vazia
>>> y = [] # ou list()
>>> y
[]

>>> # Número de elementos
>>> len(x)
4
>>> len(y)
0
```

```
>>> # Indexação
>>> nomes = ['Maria', 'João', 'Paulo']
>>> nomes[1]
'João'
>>> # Acesso fora da faixa
>>> nomes[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> # Sublistas
>>> x = [4, 1, 5, 7, 3]
>>> x[:2]
[4, 1]
>>> x[2:]
[5, 7, 3]
```

Listas - alteração, acréscimo e concatenação

```
>>> # Substituição de um elemento
>>> y = [4, 2]
>>> y[1] = 7
>>> y
[4, 7]

>>> # Acrésimo de um elemento
>>> y.append(5) # list.append(y, 5)
>>> y
[4, 7, 5]
>>> y.append(3)
>>> y
[4, 7, 5, 3]

>>> # Concatenação
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
```

Note que a função (método) `append` não produz valor de saída.

Mas qual é a utilidade de uma função que não produz valor de saída!?

Além de produzir uma saída, as funções podem ter **efeitos colaterais**, como por exemplo, modificar algum dos seus argumentos (função `append`), exibir algo na tela (função `print`), etc. Uma função que produz uma saída também pode ter um efeito colateral, que o caso da função `input`.

Então, utilizamos funções sem saída pelo efeito colateral que elas produzem.

Valores mutáveis e imutáveis

Nós vimos que os valores do tipo lista e de tipos estruturas podem ser alterados depois que são criados, por isso são chamados de valores **mutáveis**.

Já alguns valores não podem ser alterados, que é o caso dos valores dos tipos `int`, `float`, `bool` e `str`. Chamamos esses valores de **imutáveis**.

Essa diferenciação é importante na prática. Vamos discutir mais sobre isso em breve.

Tanto as estruturas quanto os arranjos são utilizados para representar informações com dois ou mais itens. Então, como escolher qual utilizar?

- Usamos estruturas quando cada item da informação tem uma interpretação particular (na estrutura `Tempo`, temos os componentes `horas`, `minutos` e `segundos`)
- Usamos arranjos quando os itens da informação são da mesma natureza (todos são nomes, notas, etc)

No exemplo da loteria, os itens da aposta e dos sorteios têm a mesma natureza, são todos números, então devemos utilizar arranjos ao invés de estruturas. Vamos alterar o código!

```python
def sorteado(n: int,
             sorteados: SeisNumeros)
             -> bool:
    em_sorteados = False
    if n == sorteados.a:
        em_sorteados = True
    if n == sorteados.b:
        em_sorteados = True
    if n == sorteados.c:
        em_sorteados = True
    if n == sorteados.d:
        em_sorteados = True
    if n == sorteados.e:
        em_sorteados = True
    if n == sorteados.f:
        em_sorteados = True
    return em_sorteados
```

```python
def sorteado(n: int,
             sorteados: list[int])
             -> bool:
    em_sorteados = False
    if n == sorteados[0]:
        em_sorteados = True
    if n == sorteados[1]:
        em_sorteados = True
    if n == sorteados[2]:
        em_sorteados = True
    if n == sorteados[3]:
        em_sorteados = True
    if n == sorteados[4]:
        em_sorteados = True
    if n == sorteados[5]:
        em_sorteados = True
    return em_sorteados
```

```python
def numero_acertos(aposta: SeisNumeros,
                   sorteados: SeisNumeros)
                   -> int:
    acertos = 0
    if sorteado(aposta.a, sorteados):
        acertos = acertos + 1
    if sorteado(aposta.b, sorteados):
        acertos = acertos + 1
    if sorteado(aposta.c, sorteados):
        acertos = acertos + 1
    if sorteado(aposta.d, sorteados):
        acertos = acertos + 1
    if sorteado(aposta.e, sorteados):
        acertos = acertos + 1
    if sorteado(aposta.f, sorteados):
        acertos = acertos + 1
    return acertos
```

```python
def numero_acertos(aposta: list[int],
                   sorteados: list[int])
                   -> int:
    acertos = 0
    if sorteado(aposta[0], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[1], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[2], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[3], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[4], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[5], sorteados):
        acertos = acertos + 1
    return acertos
```

E então, o código melhorou? Ainda não! Ele continua repetitivo!

Agora vamos trocar a repetição física do código por uma repetição lógica, usando uma nova estrutura de controle. Isso é possível porque os elementos de um arranjo têm a mesma natureza.

Para cada

Em Python, uma das construções de repetição é o “para cada”, que tem a seguinte forma geral

```
for var in lista:
    instruções
```

O “para cada” funciona da seguinte maneira:

- O primeiro valor de `lista` é atribuído para `var` e as `instruções` são executadas;
- O segundo valor de `lista` é atribuído para `var` e as `instruções` são executadas;
- ...
- E assim por diante até que todos os valores de `lista` tenham sidos atribuídos para `var`.

Ou seja, o “para cada” executa as mesmas instruções atribuindo cada valor de `lista` para `var`, por isso ele chama “para cada”!

Para cada

```
def sorteado(n: int,
             sorteados: list[int])
             -> bool:
    em_sorteados = False
    if n == sorteados[0]:
        em_sorteados = True
    if n == sorteados[1]:
        em_sorteados = True
    if n == sorteados[2]:
        em_sorteados = True
    if n == sorteados[3]:
        em_sorteados = True
    if n == sorteados[4]:
        em_sorteados = True
    if n == sorteados[5]:
        em_sorteados = True
    return em_sorteados
```

Nesse código, queremos executar as mesmas instruções, uma vez para cada valor de `sorteados`, então, podemos utilizar o “para cada”.

```
def sorteado(n: int,
             sorteados: list[int])
             -> bool:
    em_sorteados = False
    for x in sorteados:
        if n == x:
            em_sorteados = True
    return em_sorteados
```

Para cada

```
def numero_acertos(aposta: list[int],
                   sorteados: list[int])
                   -> int:
    acertos = 0
    if sorteado(aposta[0], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[1], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[2], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[3], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[4], sorteados):
        acertos = acertos + 1
    if sorteado(aposta[5], sorteados):
        acertos = acertos + 1
    return acertos
```

Nesse código, queremos executar as mesmas instruções, uma vez para cada valor de `aposta`, então, podemos utilizar o “para cada”.

```
def numero_acertos(aposta: list[int],
                   sorteados: list[int])
                   -> int:
    acertos = 0
    for n in aposta:
        if sorteado(n, sorteados):
            acertos = acertos + 1
    return acertos
```

Execução passo a passo do “para cada”

```
1  def sorteado(n: int,
2               sorteados: list[int])
3               -> bool:
4      em_sorteados = False
5      for x in sorteados:
6          if n == x:
7              em_sorteados = True
8      return em_sorteados
9
10 sorteado(35, [1, 7, 32, 35, 50, 51])
```

Vamos ver como a execução passo a passo funciona para o “para cada”.

Qual é a ordem que as linhas são executadas?

10, 4 (`em_sorteados = False`)

5 (`x = 1`), 6,

5 (`x = 7`), 6,

5 (`x = 32`), 6,

5 (`x = 35`), 6, 7 (`em_sorteados = True`),

5 (`x = 50`), 6,

5 (`x = 51`), 6,

5 (identifica que não tem mais elementos), 8 (devolve `True`), 10.

No exemplo da loteria, vimos como uma repetição física de código pode ser substituída por uma repetição lógica.

Em geral, não precisamos ter uma repetição física de código para depois trocarmos por uma repetição lógica, podemos projetar uma função usando uma repetição lógica diretamente.

Vamos ver como fazer isso!

Exemplo - Soma

Projete uma função que some os números de uma lista.

Exemplo - Soma - Especificação

```
def soma(lst: list[int]) -> int:
    '''
    Soma os elementos de *lst*.
    Exemplos
    >>> soma([])
    0
    >>> soma([3])
    3
    >>> soma([3, 7])
    10
    >>> soma([3, 7, 2])
    12
    '''
    return 0
```

Qual abordagem podemos utilizar para implementar essa função? A incremental.

Na abordagem incremental, iniciamos o resultado com um valor, e vamos atualizando o resultado conforme processamos os dados de entrada, no final, temos o resultado da função.

Qual é o resultado que queremos calcular? A `soma` dos elementos de `lst`.

Com qual valor iniciamos `soma`? `0`.

Se estamos analisando um número `n` de `lst`, como atualizamos `soma`? Adicionando `n` em `soma`, isto é, `soma = soma + n`.

Exemplo - Soma - Implementação

```
1  def soma(lst: list[int]) -> int:
2      soma = 0
3      for n in lst:
4          soma = soma + n
5      return soma
6
7  soma([5, 1, 4])
```

Vamos exercitar mais uma vez a execução passo a passo.

Qual é a ordem que as linhas são executadas?

7, 2 (`soma = 0`),

3 (`n = 5`), 4 (`soma = 5`),

3 (`n = 1`), 4 (`soma = 6`),

3 (`n = 4`), 4 (`soma = 10`),

3 (identifica que não tem mais elementos), 5 (devolve `10`),

7

Exemplo - Strings que começam com A

Projete uma função que encontre as strings que começam com `'A'` de uma lista de strings.

Exemplo - Strings que começam com A - Especificação

```
def encontra_comeca_a(lst: list[str]) -> list[str]:
    '''
    Encontra os elementos de *lst* que começam com 'A'.
    Exemplos
    >>> encontra_comeca_a([])
    []
    >>> encontra_comeca_a(['Ali'])
    ['Ali']
    >>> encontra_comeca_a(['Ali', 'ala'])
    ['Ali']
    >>> encontra_comeca_a(['Ali', 'ala', 'Alto'])
    ['Ali', 'Alto']
    >>> encontra_comeca_a(['Ali', 'ala', 'Alto', ''])
    ['Ali', 'Alto']
    '''
    return []
```

Qual abordagem podemos utilizar para implementar essa função? A incremental.

Qual é o resultado que queremos calcular? A lista `comeca_a` com os elementos de `lst` que começam com `'A'`.

Com qual valor iniciamos `comeca_a`? `[]`.

Se estamos analisando uma string `s` de `lst`, como atualizamos `comeca_a`? Adicionando `s` em `comeca_a` (`comeca_a.append(s)`) se `s` começa com `'A'`, isto é, `s != '' and s[0] == 'A'`.

Exemplo - Strings que começam com A - Implementação

```python
def encontra_comeca_a(lst: list[str]) -> list[str]:
    '''
    Encontra os elementos de *lst* que começam com 'A'.
    Exemplos
    >>> encontra_comeca_a([])
    []
    >>> encontra_comeca_a(['Ali'])
    ['Ali']
    >>> encontra_comeca_a(['Ali', 'ala'])
    ['Ali']
    >>> encontra_comeca_a(['Ali', 'ala', 'Alto'])
    ['Ali', 'Alto']
    >>> encontra_comeca_a(['Ali', 'ala', 'Alto', ''])
    ['Ali', 'Alto']
    '''
    comeca_a = []
    for s in lst:
        if s != '' and s[0] == 'A':
            comeca_a.append(s)
    return comeca_a
```

Exemplo - Máximo

Projete uma função que encontre o valor máximo em uma lista não vazia de inteiros.

Exemplo - Máximo - Especificação

```
def maximo(lst: list[int]) -> int:
    '''
    Encontra o valor máximo de *lst*.
    Requer que *lst* seja não vazia.
    Exemplos
    >>> maximo([2])
    2
    >>> maximo([2, 4])
    4
    >>> maximo([2, 4, 3])
    4
    >>> maximo([2, 4, 3, 7])
    7
    '''
    return 0
```

Qual abordagem podemos utilizar para implementar essa função? A incremental.

Qual é o resultado que queremos calcular? O valor `maximo` de `lst`.

Com qual valor iniciamos `maximo`? `lst[0]`.

Se estamos analisando um número `n` de `lst`, como atualizamos `maximo`? Atribuindo `n` para `maximo` se `n > maximo`.

Exemplo - Máximo - Implementação

```
def maximo(lst: list[int]) -> int:
    '''
    Encontra o valor máximo de *lst*.
    Requer que *lst* seja não vazia.
    Exemplos
    >>> maximo([2])
    2
    >>> maximo([2, 4])
    4
    >>> maximo([2, 4, 3])
    4
    >>> maximo([2, 4, 3, 7])
    7
    '''
    assert len(lst) != 0
    maximo = lst[0]
    for n in lst:
        if n > maximo:
            maximo = n
    return maximo
```

Exemplo - Média tamanho strings

Projete uma função que calcule a média dos tamanhos das strings de uma lista não vazia de strings.

Exemplo - Média tamanho strings - Especificação

```
def media_tamanho(lst: list[str]) -> float:
    '''
    Calcula a média dos tamanhos das
    strings de *lst*.
    Requer que *lst* seja não vazia.
    Exemplos
    >>> media_tamanho(['casa'])
    4.0
    >>> media_tamanho(['casa', 'da'])
    3.0
    >>> media_tamanho(['casa', 'da', ''])
    2.0
    >>> media_tamanho(['casa', 'da', '', 'onça'])
    2.5
    '''
    return 0.0
```

Qual abordagem podemos utilizar para implementar essa função? A incremental.

Qual é o resultado que queremos calcular? A `media` dos tamanhos das strings de `lst`.

Com qual valor iniciamos a `media`? `len(lst[0])`.

Se estamos analisando o elemento `s` de `lst`, como atualizamos `media`? Não tem com! Se `media` é `100.0` e `s` é `'nova'`, qual é o novo valor de `media`? Não temos informações suficientes para responder essa pergunta!

Como procedemos então? Como calculamos as respostas dos exemplos? Primeiro calculamos a soma dos tamanhos das strings e depois a média.

Quando a solução de um problema não pode ser expressa apenas com uma das formas que vimos até agora (direta, seleção direta, seleção aninhada e incremental), então podemos tentar uma combinação dessas formas.

Para isso, primeiro criamos um **esboço de solução**, que é uma descrição em alto nível das etapas do processamento da função, e depois implementamos cada etapa usando a forma apropriada.

Exemplo - Média tamanho strings - Implementação

```python
def media_tamanho(lst: list[str]) -> float:
    '''
    Calcula a média dos tamanhos das
    strings de *lst*.
    Requer que *lst* seja não vazia.
    Exemplos
    >>> media_tamanho(['casa'])
    4.0
    >>> media_tamanho(['casa', 'da'])
    3.0
    >>> media_tamanho(['casa', 'da', ''])
    2.0
    >>> media_tamanho(['casa', 'da', '', 'onça'])
    2.5
    '''
    # Calcular a soma dos tamanhos
    # Calcular a média
    return 0.0
```

Soma dos tamanhos

Qual estratégia podemos utilizar? A incremental. Qual é o resultado que queremos calcular? A `soma` dos tamanhos. Com qual valor iniciamos `soma`? `0`. Se estamos analisando o elemento `s` de `lst`, como atualizamos `soma`?
`soma = soma + len(s)`.

Média

Como calculamos a média?
`soma / len(lst)`.

Exemplo - Média tamanho strings - Implementação

```python
def media_tamanho(lst: list[str]) -> float:
    assert len(lst) != 0

    # Soma dos tamanhos
    soma = 0
    for s in lst:
        soma = soma + len(s)

    # Média
    return soma / len(lst)
```

Quando utilizamos a abordagem incremental?

Quando precisamos computar algo de forma incremental! Ou seja, quando não é possível calcular a resposta de forma direta ou usando apenas seleção.

O que precisamos determinar quando vamos utilizar a abordagem incremental?

- Quais valores queremos calcular;
- Como os valores são inicializados;
- Como os valores são atualizados.

O quê pode nos impedir de utilizar a abordagem incremental?

- Se não conseguirmos definir como os valores são inicializados
- Se não conseguirmos definir como os valores são atualizados, que foi o caso de `media_tamanhos`

Como procedemos nesses casos?

Ao invés de computar a resposta final de forma incremental, definimos um esboço de solução, que calcula valores intermediários que serão utilizados para calcular o valor final.

No caso de `media_tamanhos`, primeiro calculamos a soma dos tamanhos de forma incremental, e depois calculamos a média diretamente.

Por enquanto, vimos uma forma de implementar a abordagem incremental no Python, o “para cada”, que utilizamos quando estamos interessados em analisar todos os elementos de uma lista.

Essa forma pode não ser adequada ou suficiente para resolver alguns problemas.

Veremos a seguir outras possibilidades.