Repetição, arranjos e conjuntos

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 + x`.

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 (`x = 5`), 4 (`soma = 5`),

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

3 (`x = 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

```python
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

```
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)
```

Exemplo - Índice máximo

Projete uma função que encontre o índice (posição) da primeira ocorrência do valor máximo de uma lista não vazia de números.

Exemplo - Índice máximo - Especificação

```
def indice_maximo(lst: list[int]) -> int:
    '''
    Encontra o índice da primeira ocorrência
    do valor máximo de *lst*.
    Requer que *lst* seja não vazia.
    Exemplos
    >>> indice_maximo([5])
    0
    >>> indice_maximo([5, 6])
    1
    >>> indice_maximo([5, 6, 6])
    1
    >>> indice_maximo([5, 6, 6, 8])
    3
    '''
    return 0
```

Qual estratégia podemos utilizar? A incremental.

Qual o resultado queremos calcular? O índice `imax` do máximo de `lst`.

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

Se estamos analisando um elemento `n` de `lst`, como atualizamos `imax`? Não tem como! Precisamos atualizar `imax`, que é um índice, mas só temos acesso ao elemento `n`.

Como procedemos?

Exemplo - Índice máximo - Especificação

```
def indice_maximo(lst: list[int]) -> int:
    '''
    Encontra o índice da primeira ocorrência
    do valor máximo de *lst*.
    Requer que *lst* seja não vazia.
    Exemplos
    >>> indice_maximo([5])
    0
    >>> indice_maximo([5, 6])
    1
    >>> indice_maximo([5, 6, 6])
    1
    >>> indice_maximo([5, 6, 6, 8])
    3
    '''
    return 0
```

Qual estratégia podemos utilizar? A incremental.

Vamos calcular duas coisas simultaneamente, o índice `imax` do máximo e o índice `i` do elemento atual.

Com qual valor iniciamos `imax` e `i`? `0`.

Se estamos analisando um número `n` de `lst`, como atualizamos `imax` e `i`? Atribuímos `i` para `imax` se `n > lst[imax]` e `i` é incrementado de `1`.

Exemplo - Índice máximo - Implementação

```python
def indice_maximo(lst: list[int]) -> int:
    assert len(lst) != 0
    i = 0
    imax = 0
    for n in lst:
        if n > lst[imax]:
            imax = i
        i = i + 1
    return imax
```

Revisão: não está claro qual é a relação entre `n` e `i`...

Podemos mudar `n` para `lst[i]`.

```python
def indice_maximo(lst: list[int]) -> int:
    assert len(lst) != 0
    i = 0
    imax = 0
    for n in lst:
        if lst[i] > lst[imax]:
            imax = i
        i = i + 1
    return imax
```

Revisão: `n` não é mais utilizado...

A questão é que não queremos mais acessar os elementos da lista diretamente, queremos usar um índice para acessar os elementos. Vamos utilizar uma variante do "para cada" que é mais apropriada para essa situação.

Para cada no intervalo

Podemos escrever o “para cada” com a seguinte forma alternativa:

```
for var in range(inicio, fim):
    instruções
```

O funcionamento dessa forma é a seguinte:

- `var` é inicializado com `inicio`
- Se `var < fim`, as `instruções` são executadas, `var` é incrementado de `1` e esse processo é executado novamente
- Senão, o “para cada” é finalizado

O valor `inicio` pode ser omitido, nesse caso, `var` é inicializado com `0`.

Vamos ver um exemplo.

Para cada no intervalo

```python
def soma(lst: list[int]) -> int:
    soma = 0
    for n in lst:
        soma = soma + n
    return soma
```

```python
def soma(lst: list[int]) -> int:
    soma = 0
    for i in range(len(lst)):
        soma = soma + lst[i]
    return soma
```

Qual das duas soluções é mais simples? A da esquerda.

Quando usamos o “para cada no intervalo”?

Quando estamos interessados em um intervalo dos elementos da lista (que pode ser todos) junto com seus índices

Exemplo - Índice máximo - Implementação

```python
def indice_maximo(lst: list[int]) -> int:
    assert len(lst) != 0
    i = 0
    imax = 0
    for n in lst:
        if lst[i] > lst[imax]:
            imax = i
        i = i + 1
    return imax
```

Como nesse caso estamos interessados nos índices dos elementos, então é mais adequado utilizar o “para cada no intervalo”.

Além disso, não precisamos analisar o primeiro elemento.

```python
def indice_maximo(lst: list[int]) -> int:
    assert len(lst) != 0
    imax = 0
    for i in range(1, len(lst)):
        if lst[i] > lst[imax]:
            imax = i
    return imax
```

Qual das duas soluções é mais simples? A da direita.

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.

Por enquanto, vimos duas formas de implementar a abordagem incremental no Python:

- O “para cada”, que utilizamos quando estamos interessados em analisar todos os elementos de uma lista
- O “para cada no intervalo”, quando estamos interessados em analisar um intervalo dos elementos de uma lista (que pode ser todos) junto com seus índices

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.

Exemplo: verificação de ordem

Projete uma função que verifique se os elementos de uma lista estão em ordem não decrescente.

Exemplo: verificação de ordem

```
def nao_decrescente(lst: list[int]) -> bool:
    '''
    Produz True se os elementos de lst estão em
    ordem não decrescente, False caso contrário.
    Exemplos
    >>> nao_decrescente([])
    True
    >>> nao_decrescente([4])
    True
    >>> nao_decrescente([4, 6])
    True
    >>> nao_decrescente([4, 2])
    False
    >>> nao_decrescente([4, 6, 6])
    True
    >>> nao_decrescente([4, 6, 5])
    False
    >>> nao_decrescente([4, 3, 5])
    False
    '''
```

Como proceder com a implementação dessa função? Usando a estratégia incremental.

Como calculamos manualmente a resposta dos exemplos? Comparando cada elemento com o próximo (ou anterior).

Essa forma parece diferente... Antes era necessário analisar um elemento da lista a cada iteração, agora temos que analisar dois elementos.

Como proceder nesse caso?

Vamos implementar a função para uma lista de 5 elementos usando repetição física de código e depois vamos transformar a repetição física em repetição lógica.

Exemplo: verificação de ordem

```python
def nao_decrescente(lst: list[int]) -> bool:
    assert len(lst) == 5
    # Assumimos com em_ordem = True que lst
    # está em ordem não decrescente, se
    # encontramos um elemento "fora de ordem",
    # mudamos em_ordem para False.
    em_ordem = True
    if lst[0] > lst[1]:
        em_ordem = False
    if lst[1] > lst[2]:
        em_ordem = False
    if lst[2] > lst[3]:
        em_ordem = False
    if lst[3] > lst[4]:
        em_ordem = False
    return em_ordem
```

Vamos transformar essa repetição física de código em uma repetição lógica.

Devemos usar o “para cada” ou o “para cada no intervalo”? Precisamos dos índices, então “para cada no intervalo”.

Qual é o intervalo? `range(0, 4)` ou `range(1, 5)`.

```python
def nao_decrescente(lst: list[int]) -> bool:
    assert len(lst) == 5
    em_ordem = True
    for i in range(1, 5):
        if lst[i - 1] > lst[i]:
            em_ordem = False
    return em_ordem
```

Exemplo: verificação de ordem

```python
def nao_decrescente(lst: list[int]) -> bool:
    assert len(lst) == 5
    em_ordem = True
    for i in range(1, 5):
        if lst[i - 1] > lst[i]:
            em_ordem = False
    return em_ordem
```

Como **generalizar** esse código para que ele funcione para listas de qualquer tamanho? Modificando o limite do intervalo de 5 para `len(lst)`.

```python
def nao_decrescente(lst: list[int]) -> bool:
    em_ordem = True
    for i in range(1, len(lst)):
        if lst[i - 1] > lst[i]:
            em_ordem = False
    return em_ordem
```

Revisão: mesmo encontrando valores “fora de ordem” a repetição continua e analisa toda a lista...

Enquanto

Usamos o “para cada” e o “para cada no intervalo” quando queremos analisar todos os elementos (de um intervalo) da lista.

Nesse tipo de repetição a condição de parada é analisar todos os elementos (do intervalo) da lista.

Para situações que precisamos de um processo incremental que depende de uma condição mais geral utilizamos o **`while`** (enquanto em inglês).

A forma geral do **`while`** é:

```
while condição:
    instruções
```

O funcionamento do **`while`** é o seguinte:

- A `condição` é avaliada
- Se ela for `True`, as `instruções` são executadas e o processo se repete
- Senão, o **`while`** termina

Enquanto - Exemplo

```python
def nao_decrescente(lst: list[int]) -> bool:
    em_ordem = True
    for i in range(1, len(lst)):
        if lst[i - 1] > lst[i]:
            em_ordem = False
    return em_ordem
```

Vamos reescrever o corpo da função usando o **while**.

```python
def nao_decrescente(lst: list[int]) -> bool:
    em_ordem = True
    i = 1
    while i < len(lst):
        if lst[i - 1] > lst[i]:
            em_ordem = False
        i = i + 1
    return em_ordem
```

O código está mais simples? Não, o controle do índice **i**, que era automático, agora é feito explicitamente.

Resolvemos o problema do processamento continuar após um elemento fora de ordem ser encontrado? Não... Como podemos resolver esse problema? Alterando a condição do **while** para prosseguir apenas se **em_ordem** for **True**.

Enquanto - Exemplo

```python
# Versão com for
def nao_decrescente(lst: list[int]) -> bool:
    em_ordem = True
    for i in range(1, len(lst)):
        if lst[i - 1] > lst[i]:
            em_ordem = False
    return em_ordem

# Versão com enquanto e ajuste da condição
def nao_decrescente(lst: list[int]) -> bool:
    em_ordem = True
    i = 1
    while i < len(lst) and em_ordem:
        if lst[i - 1] > lst[i]:
            em_ordem = False
        i = i + 1
    return em_ordem
```

```python
# Versão com enquanto
def nao_decrescente(lst: list[int]) -> bool:
    em_ordem = True
    i = 1
    while i < len(lst):
        if lst[i - 1] > lst[i]:
            em_ordem = False
        i = i + 1
    return em_ordem
```

Enquanto - execução passo a passo

```
1  def nao_decrescente(lst: list[int]) -> bool:
2      em_ordem = True
3      i = 1
4      while i < len(lst) and em_ordem:
5          if lst[i - 1] > lst[i]:
6              em_ordem = False
7          i = i + 1
8      return em_ordem
9
10 nao_decrescente([1, 3, 3, 2, 7, 8])
```

Qual é a ordem que as linhas são executadas?

10

2 (`em_ordem = True`)

3 (`i = 1`)

4, 5, 7 (`i = 2`)

4, 5, 7 (`i = 3`)

4, 5, 6 (`em_ordem = False`), 7 (`i = 4`)

4

8 (produz `False`)

10

Para implementar uma função com o método incremental usando o **`while`** precisamos determinar as mesmas três coisas

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

e mais

- Qual é a condição da repetição.

Exemplo: palíndromo

Projete uma função que verifique se uma lista de inteiros é palíndromo, isto é, tem os mesmos elementos quanto vistos da direita para esquerda ou da esquerda para a direita.

Exemplo: palíndromo

```
def palindromo(lst: list[int]) -> bool:
    '''Produz True se *lst* é palíndromo, isto
    é, tem os mesmos elementos quando vistos
    da direira para esquerda e da esquerda
    para direita. Produz False caso contrário.
    >>> palindromo([])
    True
    >>> palindromo([4])
    True
    >>> palindromo([1, 1])
    True
    >>> palindromo([1, 2])
    False
    >>> palindromo([1, 2, 1])
    True
    >>> palindromo([1, 5, 5, 1])
    True
    >>> palindromo([1, 5, 1, 5])
    False
    '''
```

Como proceder com a implementação dessa função? Usando a estratégia incremental.

Como calculamos manualmente as respostas dos exemplos? Comparando o primeiro com o último, o segundo com o penúltimo, etc.

Vamos implementar a função para uma lista de 7 elementos usando repetição física de código e depois vamos transformar a repetição física em repetição lógica.

Exemplo: palíndromo

```
def palindromo(lst: list[int]) -> bool:
    '''
    >>> palindromo([3, 2, 1, 7, 5, 2, 3])
    False
    '''
    assert len(lst) == 7
    eh_palindromo = True
    if lst[0] != lst[6]:
        eh_palindromo = False
    if lst[1] != lst[5]:
        eh_palindromo = False
    if lst[2] != lst[4]:
        eh_palindromo = False
    return eh_palindromo
```

Como transformar essa repetição física de código em uma repetição lógica?

Nas transformações que fizemos em `sorteado`, `numero_acertos` e `nao_decrescente` introduzimos uma repetição diretamente.

Nesse exemplo parece que isso é mais complicado pois o código que se repete é menos parecido.

Vamos deixar os trechos que se repetem mais parecidos introduzindo variáveis para os índices.

Exemplo: palíndromo

```python
def palindromo(lst: list[int]) -> bool:
    '''
    >>> palindromo([3, 2, 1, 7, 5, 2, 3])
    False
    '''
    assert len(lst) == 7
    eh_palindromo = True
    if lst[0] != lst[6]:
        eh_palindromo = False
    if lst[1] != lst[5]:
        eh_palindromo = False
    if lst[2] != lst[4]:
        eh_palindromo = False
    return eh_palindromo
```

```python
def palindromo(lst: list[int]) -> bool:
    assert len(lst) == 7
    eh_palindromo = True
    i = 0
    j = 6
    if lst[i] != lst[j]:
        eh_palindromo = False


    if lst[i] != lst[j]:
        eh_palindromo = False


    if lst[i] != lst[j]:
        eh_palindromo = False


    return eh_palindromo
```

Como os índice `i` e `j` devem ser atualizados?

Exemplo: palíndromo

```python
def palindromo(lst: list[int]) -> bool:
    '''
    >>> palindromo([3, 2, 1, 7, 5, 2, 3])
    False
    '''
    assert len(lst) == 7
    eh_palindromo = True
    if lst[0] != lst[6]:
        eh_palindromo = False
    if lst[1] != lst[5]:
        eh_palindromo = False
    if lst[2] != lst[4]:
        eh_palindromo = False
    return eh_palindromo
```

```python
def palindromo(lst: list[int]) -> bool:
    assert len(lst) == 7
    eh_palindromo = True
    i = 0
    j = 6
    if lst[i] != lst[j]:
        eh_palindromo = False
    i = i + 1
    j = j - 1
    if lst[i] != lst[j]:
        eh_palindromo = False
    i = i + 1
    j = j - 1
    if lst[i] != lst[j]:
        eh_palindromo = False
    i = i + 1
    j = j - 1
    return eh_palindromo
```

Como os índice `i` e `j` devem ser atualizados? Somando e subtraindo 1.

Exemplo: palíndromo

```python
def palindromo(lst: list[int]) -> bool:
    assert len(lst) == 7
    eh_palindromo = True
    i = 0
    j = 6
    if lst[i] != lst[j]:
        eh_palindromo = False
    i = i + 1
    j = j - 1
    if lst[i] != lst[j]:
        eh_palindromo = False
    i = i + 1
    j = j - 1
    if lst[i] != lst[j]:
        eh_palindromo = False
    i = i + 1
    j = j - 1
    return eh_palindromo
```

Agora podemos transformar essa repetição física de código para repetição lógica.

Os valores que são calculados, a inicialização e a atualização já estão claras no código. O que precisamos determinar? A condição de repetição, que é `i < j and eh_palindromo`.

```python
def palindromo(lst: list[int]) -> bool:
    assert len(lst) == 7
    eh_palindromo = True
    # começa dos extremos
    i = 0
    j = 6
    while i < j and eh_palindromo:
        if lst[i] != lst[j]:
            eh_palindromo = False
        # vai para o centro
        i = i + 1
        j = j - 1
    return eh_palindromo
```

Exemplo: palíndromo

```python
def palindromo(lst: list[int]) -> bool:
    assert len(lst) == 7
    eh_palindromo = True
    # começa dos extremos
    i = 0
    j = 6
    while i < j and eh_palindromo:
        if lst[i] != lst[j]:
            eh_palindromo = False
        # vai para o centro
        i = i + 1
        j = j - 1
    return eh_palindromo
```

Como **generalizar** esse código para que ele funcione para listas de qualquer tamanho? Modificando a inicialização `j = 6` para `j = len(lst) - 1`.

```python
def palindromo(lst: list[int]) -> bool:
    eh_palindromo = True
    # começa dos extremos
    i = 0
    j = len(lst) - 1
    while i < j and eh_palindromo:
        if lst[i] != lst[j]:
            eh_palindromo = False
        # vai para o centro
        i = i + 1
        j = j - 1
    return eh_palindromo
```

Até agora todos os problemas que utilizamos a abordagem incremental (repetição) envolviam uma lista de valores.

Agora veremos o uso da abordagem incremental em problemas que não envolvem uma lista de valores.

Exemplo: fatorial

O fatorial de um número natural n é o produto de todos os números naturais de 1 até n, isto é, $1 \times \cdots \times (n-1) \times n$. Projete uma função que determine o fatorial de um número n.

Exemplo: fatorial

```
def fatorial(n: int) -> int:
    '''
    Calcula o produto de todos os naturais
    entre 1 e n, isto é, 1 * ... * (n - 1) * n.
    Exemplos
    >>> fatorial(0)
    1
    >>> fatorial(1)
    1
    >>> fatorial(2)
    2
    >>> fatorial(3)
    6
    >>> fatorial(4)
    24
    '''
    return 0
```

Como fazer a implementação? Generalizando soluções específicas!

Como determinar de forma incremental o fatorial de 5?

```
def fatorial(n: int) -> int:
    assert n == 5
    fat = 1
    fat = fat * 2
    fat = fat * 3
    fat = fat * 4
    fat = fat * 5
    return fat
```

Exemplo: fatorial

```python
def fatorial(n: int) -> int:
    assert n == 5
    fat = 1
    fat = fat * 2
    fat = fat * 3
    fat = fat * 4
    fat = fat * 5
    return fat
```

Que construção de repetição podemos utilizar para transformar essa repetição física de código em uma repetição lógica?

O “para cada no intervalo”. E qual é o intervalo? `range(2, 6)`

```python
def fatorial(n: int) -> int:
    assert n == 5
    fat = 1
    for i in range(2, 6):
        fat = fat * i
    return fat
```

Como **generalizar** esse código para que ele funcione para qualquer valor de `n`? Alterando o limite do intervalo de `6` para `n + 1`.

```python
def fatorial(n: int) -> int:
    fat = 1
    for i in range(2, n + 1):
        fat = fat * i
    return fat
```

Exemplo: número primo

Um número inteiro positivo n é primo se ele tem exatamente dois divisores distintos, 1 e n.

Projete uma função que verifique se um número inteiro positivo é primo.

Exemplo: número primo

```
def primo(n: int) -> bool:
    '''
    Produz True se *n* é um número primo,
    isto é, tem exatamente dois divisores
    distintos, 1 e ele mesmo. Produz False
    se *n* não é primo.
    Exemplos
    >>> primo(1) # 1
    False
    >>> primo(2) # 1 2
    True
    >>> primo(3) # 1 3
    True
    >>> primo(5) # 1 5
    True
    >>> primo(8) # 1 2 4 8
    False
    >>> primo(11) # 1 11
    True
    '''
```

Como fazer a implementação? Generalizando soluções específicas!

Como determinar de forma incremental se o número 5 é primo?

```
def primo(n: int) -> bool:
    assert n == 5
    num_divisores = 0
    if n % 1 == 0:
        num_divisores = num_divisores + 1
    if n % 2 == 0:
        num_divisores = num_divisores + 1
    if n % 3 == 0:
        num_divisores = num_divisores + 1
    if n % 4 == 0:
        num_divisores = num_divisores + 1
    if n % 5 == 0:
        num_divisores = num_divisores + 1
    return num_divisores == 2
```

Exemplo: número primo

```python
def primo(n: int) -> bool:
    assert n == 5
    num_divisores = 0
    if n % 1 == 0:
        num_divisores = num_divisores + 1
    if n % 2 == 0:
        num_divisores = num_divisores + 1
    if n % 3 == 0:
        num_divisores = num_divisores + 1
    if n % 4 == 0:
        num_divisores = num_divisores + 1
    if n % 5 == 0:
        num_divisores = num_divisores + 1
    return num_divisores == 2
```

Que construção de repetição podemos utilizar para transformar essa repetição física de código em uma repetição lógica?

O "para cada no intervalo". E qual é o intervalo? `range(1, 6)`

```python
def primo(n: int) -> bool:
    assert n == 5
    num_divisores = 0
    for i in range(1, 6):
        if n % i == 0:
            num_divisores = num_divisores + 1
    return num_divisores == 2
```

Como **generalizar** esse código para que ele funcione para qualquer valor de `n`? Alterando o limite do intervalo de `6` para `n + 1`.

Exemplo: número primo

```
def primo(n: int) -> bool:
    num_divisores = 0
    for i in range(1, n + 1):
        if n % i == 0:
            num_divisores = num_divisores + 1
    return num_divisores == 2
```

Revisão: quando `num_divisores` fica maior que 2 a repetição pode ser interrompida.

```
def primo(n: int) -> bool:
    num_divisores = 0
    i = 1
    while i < n + 1 and num_divisores <= 2:
        if n % i == 0:
            num_divisores = num_divisores + 1
        i = i + 1
    return num_divisores == 2
```

Revisão: 1 e n são sempre divisores de n, além disso, nenhum divisor de n (exceto n), é maior que `n // 2`. Vamos verificar se não existe nenhum divisor de n no intervalo de 2 a `n // 2`.

```
def primo(n: int) -> bool:
    num_divisores = 0
    i = 2
    while i < n // 2 and num_divisores == 0:
        if n % i == 0:
            num_divisores = num_divisores + 1
        i = i + 1
    return num_divisores == 0
```

Verificação: a função falha para `n = 1`...

Vamos alterar o **`return`** para `n != 1 and num_divisores == 0`.

Exemplo: número primo

```
def primo(n: int) -> bool:
    num_divisores = 0
    i = 2
    while i < n // 2 and num_divisores == 0:
        if n % i == 0:
            num_divisores = num_divisores + 1
        i = i + 1
    return n != 1 and num_divisores == 0
```

Revisão: `num_divisores` só pode assumir dois valores: `0` ou `1`. Então vamos mudar para `bool`.

```
def primo(n: int) -> bool:
    eh_primo = True
    i = 2
    while i < n // 2 and eh_primo:
        if n % i == 0:
            eh_primo = False
        i = i + 1
    return n != 1 and eh_primo
```

Revisão: `eh_primo` não diz de fato se é primo pois ainda depende da condição `n != 1`.

```
def primo(n: int) -> bool:
    eh_primo = n != 1
    i = 2
    while i < n // 2 and eh_primo:
        if n % i == 0:
            eh_primo = False
        i = i + 1
    return eh_primo
```

Matrizes

O tipo `list` (arranjo) que vimos é unidimensional. Algumas linguagens suportam arranjos com mais dimensões. Os arranjos bidimensionais são chamados de matrizes.

O Python não suporta nativamente matrizes, mas podemos usar lista de listas como matrizes.

Por exemplo, para representar matriz

$$A = \begin{bmatrix} 1 & 4 & 2 & 8 \\ -1 & 0 & 9 & 1 \\ 4 & 7 & -2 & 0 \end{bmatrix}$$

em Python fazemos

```
>>> m: list[list[int]] = [[1, 4, 2, 8], [-1, 0, 9, 1], [4, 7, -2, 0]]
```

Usamos as operações que já conhecemos para acessar e modificar a matriz

```
>>> m: list[list[int]] = [[1, 4, 2, 8], [-1, 0, 9, 1], [4, 7, -2, 0]]
>>> m[1]
[-1, 0, 9, 1]
>>> m[1][2]
9
>>> len(m)
3
>>> len(m[0])
4
>>> m[2][1] = 0
>>> m
[[1, 4, 2, 8], [-1, 0, 9, 1], [4, 0, -2, 0]]
```

Exemplo: matriz nula

Projete uma função que receba dois números inteiros positivos, m e n, e crie uma matriz $A_{m \times n}$, com m linhas e n colunas, com todos os elementos zeros.

Exemplo: matriz nula

```
1  def cria_matriz_nula(m: int, n: int) -> list[list[int]]:
2      '''
3      Cria uma matriz nula com *m* linhas e *n* colunas.
4
5      Requer que m > 0 e n > 0.
6
7      Exemplos
8      >>> cria_matriz_nula(2, 3)
9      [[0, 0, 0], [0, 0, 0]]
10     '''
11     a = []
12     for i in range(m):
13         linha = []
14         for j in range(n):
15             linha.append(0)
16         a.append(linha)
17     return a
```

Para a chamada `cria_matriz_nula(2, 3)`, qual é a ordem que as linhas são executas?

11 (`m = []`)

12 (`i = 0`)

13 (`linha = []`)

14 (`j = 0`), 15 (`linha = [0]`), 14 (`j = 1`), 15 (`linha = [0, 0]`), 14 (`j = 2`), 15 (`linha = [0, 0, 0]`), 14 (`j = 3`)

16 (`m = [[0, 0, 0]]`), 12 (`i = 1`)

13 (`linha = []`)

14 (`j = 0`), 15 (`linha = [0]`), 14 (`j = 1`), 15 (`linha = [0, 0]`), 14 (`j = 2`), 15 (`linha = [0, 0, 0]`), 14 (`j = 3`)

16 (`m = [[0, 0, 0], [0, 0, 0]]`), 12 (`i = 2`), 17

Exemplos: matriz regular

Uma matriz é regular quando todos as linhas têm a mesma quantidade de elementos. Projete uma função que verifique se uma matriz é regular.

Feito em sala.

Daqui para frente só vamos utilizar matrizes regulares.

Exemplos: quantidade de zeros

Projete uma função que conte a quantidade de elementos nulos de uma matriz.

Feito em sala.

Exemplos: matriz transposta

Projete uma função que crie a matriz transposta de uma data matriz.

Feito em sala.

Vimos o tipo `list`, pré-definido em Python, que serve para representar uma sequência de valores.

Um conjunto em Python é semelhante a uma lista, mas não contém elementos repetidos, a ordem dos elementos não é definida, os elementos não são indexados e os elementos precisam ser imutáveis.

Conjuntos

```
>>> # conjunto vazio
>>> a: set[int] = set()
>>> a
{}

>>> # inicialização com elementos
>>> a = {10, 4}
>>> a
{10, 4}

>>> # adição de elementos
>>> a.add(3)
>>> a.add(4)
>>> a
{10, 3, 4}
```

```
>>> # pertinência
>>> 4 in a
True
>>> 20 in a
False

>>> # tamanho
>>> len(a)
3

>>> # iteração
>>> soma = 0
>>> for e in a:
...     soma = soma + e
...
>>> soma
17
```