Outras formas de repetição

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Existem situações em que a repetição com o “para cada” não é adequada ou suficiente.

Veremos agora outras formas de repetição.

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 o 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.

Exemplo - verificação de ordem

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

Exemplo - verificação de ordem - especificação

```
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 único 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 - implementação

```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 - implementação

```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 da repetição, que está implícita, é a existência de elementos (do intervalo) na lista ainda não processados.

Para situações que precisamos de um processo incremental que depende de uma condição mais geral utilizamos a instrução “enquanto” (**`while`** 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.

O que estamos ganhando se não é a simplicidade? Por enquanto nada, o que queremos é ganhar desempenho fazendo a repetição parar assim que um elemento fora de ordem for encontrado. Como fazemos isso? 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:

- 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 quando vistos da direita para esquerda ou da esquerda para a direita.

Exemplo - palíndromo - especificação

```
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 - implementação

```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
```

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 - implementação

```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 - implementação

```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 - implementação

```
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`.

```
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 - implementação

```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
```

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 a matriz

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

em Python fazemos

```
>>> a: 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 os elementos de uma matriz

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

```python
def cria_matriz_nula(m: int, n: int) -> list[list[int]]:
    '''
    Cria uma matriz nula com *m* linhas e *n* colunas.

    Requer que m > 0 e n > 0.

    Exemplos
    >>> cria_matriz_nula(2, 3)
    [[0, 0, 0], [0, 0, 0]]
    '''

    a = []
    for i in range(m):
        linha = []
        for j in range(n):
            linha.append(0)
        a.append(linha)
    return a
```

Exemplo - matriz nula

```python
def cria_matriz_nula(m: int, n: int) -> list[list[int]]:
    '''
    Cria uma matriz nula com *m* linhas e *n* colunas.

    Requer que m > 0 e n > 0.

    Exemplos
    >>> cria_matriz_nula(2, 3)
    [[0, 0, 0], [0, 0, 0]]
    '''
    a = []
    for i in range(m):
        linha = []
        for j in range(n):
            linha.append(0)
        a.append(linha)
    return a
```

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

11 (`a = []`)

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

Exemplo - 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.

Exemplo - matriz regular

```
def eh_regular(a: list[list[int]]) -> bool:
    '''Produz True se *a* é uma matriz
    regular, isto é, todas as linhas tem a
    mesma quantidade de elementos.
    Exemplos
    >>> eh_regular([])
    True
    >>> eh_regular([[2]])
    True
    >>> eh_regular([[2], [4]])
    True
    >>> eh_regular([[2], [4, 1]])
    False
    >>> eh_regular([[2, 1, 6], [4, 0, 1]])
    True
    >>> eh_regular([[2, 1], [4, 0, 1]])
    False
    >>> eh_regular([[2], [4], [7]])
    True
    '''
```

```
def eh_regular(a: list[list[int]]) -> bool:
    regular = True
    for linha in a:
        if len(linha) != len(a[0])
            regular = False
    return regular
```

Revisão: Podemos parar antes.

```
def eh_regular(a: list[list[int]]) -> bool:
    regular = True
    i = 1
    while i < len(a) and regular:
        if len(a[0]) != len(a[i]):
            regular = False
        i = i + 1
    return regular
```

Vamos utilizar apenas matrizes regulares.

Exemplo - quantidade de zeros

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

Exemplo - quantidade de zeros

```python
def conta_zeros(a: list[list[int]]) -> int:
    '''
    Conta a quantidade de zeros da matriz *m*.

    Exemplos
    >>> conta_zeros([[1, 0, 7], [0, 1, 0]])
    3
    >>> conta_zeros([[1, 0], [1, 2], [0, 2]])
    2
    '''

    num_zeros = 0
    for linha in a:
        for elem in linha:
            if elem == 0:
                num_zeros = num_zeros + 1
    return num_zeros
```

```python
def conta_zeros(a: list[list[int]]) -> int:
    '''
    Conta a quantidade de zeros da matriz *m*.

    Exemplos
    >>> conta_zeros([[1, 0, 7], [0, 1, 0]])
    3
    >>> conta_zeros([[1, 0], [1, 2], [0, 2]])
    2
    '''

    num_zeros = 0
    for i in range(len(a)):
        for j in range(len(a[i])):
            if a[i][j] == 0:
                num_zeros = num_zeros + 1
    return num_zeros
```

Exemplo - matriz transposta

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

Exemplo - matriz transposta

```python
def transposta(a: list[list[int]]) -> list[list[int]]:
    '''
    Cria a matriz transposta de *m*.

    Requer que *m* seja regular.

    Exemplos
    >>> transposta([[4, 5, 1], [7, 8, 9]])
    [[4, 7], [5, 8], [1, 9]]
    >>> transposta([[4, 1], [7, 8], [2, 6], [5, 3]])
    [[4, 7, 2, 5], [1, 8, 6, 3]]
    '''

    t = []
    for j in range(len(a[0])):
        coluna = []
        for i in range(len(a)):
            coluna.append(a[i][j])
        t.append(coluna)
    return t
```

Até agora todos os problemas que resolvemos 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 - especificação

```
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 - implementação

```
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)`

```
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`.

```
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 - especificação

```
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 - implementação

```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 - revisão

```
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 - revisão

```
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
```

Revisão

Usamos instruções de repetição quando queremos computar algo de forma incremental.

Vimos as seguintes formas de repetição:

- Para cada
- Para cada no intervalo
- Enquanto

O para cada é mais restrito mas é mais simples de utilizar, o enquanto é mais genéricos mas é mais complicado, por isso, quando possível, preferimos utilizar o para cada.

Em algumas situações fazemos uma implementação inicial usando o para cada e depois, na revisão, mudamos para o enquanto se tivermos algum benefício, como a simplificação do código ou ganho de desempenho.

Durante a implementação de uma função usando a abordagem incremental, pode ser difícil responder as perguntas: como os valores são atualizados e qual é a condição de repetição. Nesses casos, podemos utilizar a estratégia de generalização.

Começamos com uma repetição física de código para entradas restritas (tamanho ou valores fixos) e depois transformamos a repetição física de código em uma repetição lógica.

Algumas funções, como a função `primo`, requerem diversas revisões. Nesses casos é importante balancear o tempo gasto nas revisões com o benefício que elas trazem.