Seleção

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Vamos revisar como o Python executa um programa.

Execução de funções

```
1  def dobro_mais_um(n: int) -> int:
2      a = 2 * n
3      return a + 1
4
5  def main():
6      a = 5
7      n = dobro_mais_um(a + 4) + 1
8      print(n)
9
10 main()
```

Qual é o valor exibido pelo programa?

Não tente “executar” a chamada da função `dobro_mais_um`, pense apenas no seu propósito, sem olhar para o seu corpo.

Então, qual é o valor exibido na tela? 20.

Qual é a ordem em que as linhas são executadas?

10, 6, 7, 2, 3, 7, 8, 10.

Confira a execução desse código no Python Tutor (note que, diferente do que fizemos em sala, as linhas do `def` são mostradas como sendo executadas).

Quando executar passo a passo?

Ao analisar a execução de uma função, temos duas abordagens:

- **Alto nível**: pensar no propósito da função, sem olhar para a implementação.
- **Passo a passo**: acompanhar a execução linha por linha.

Quando usar cada uma?

- Se conhecemos o propósito de uma função, não precisamos olhar para a implementação — basta pensar no que ela faz (alto nível).
- Se estamos tentando entender como uma função funciona ou procurando um erro, acompanhamos a execução passo a passo.

Depois de muito trabalho aprendendo os conceitos básicos e o processo de projeto de funções, hoje é um dia especial: chegou a hora de subir de nível!

Nível 1

★ Seleção ★

Seleção

O fluxo “normal” de execução de um programa é sequencial, isto é, as linhas são executadas uma após a outra. Algumas instruções alteram esse fluxo, como por exemplo, as chamadas e retornos de funções.

Agora veremos a instrução de seleção “se senão” (`if else` em inglês), que permite, a partir de uma condição, escolher qual conjunto de instruções executar.

A forma geral do `if else` é:

```
if condição:
    instruções então
else:
    instruções senão
```

Como a instrução `if else` é executada?

O Python avalia a condição e verifica o resultado

- Se o resultado for `True`, então as instruções do bloco “instruções então” são executadas;
- Senão (o resultado é `False`), as instruções do bloco “instruções senão” são executadas;

Exemplo

```
1  a = 10
2  b = 20
3  if a > b:
4      m = a
5  else:
6      m = b
7  print(m)
```

Qual é o valor exibido pelo programa? 20.

Em que ordem as linhas são executadas para gerar esse resultado? 1, 2, 3, 6, 7.

```
1  a = 15
2  b = 8
3  if a > b:
4      m = a
5  else:
6      m = b
7  print(m)
```

Qual é o valor exibido pelo programa? 15.

Em que ordem as linhas são executadas para gerar esse resultado? 1, 2, 3, 4, 7

Qual é o propósito do `if else` nesses exemplos? Determinar o valor máximo entre `a` e `b`.

Exemplo - máximo

Vamos aproveitar esse exemplo e projetar uma função para encontrar o máximo entre dois números.

Exemplo - máximo

```
1  def maximo(a: int, b: int) -> int:
2      '''
3      Devolve o valor máximo entre
4      *a* e *b*.
5      Exemplos
6      >>> maximo(10, 8) # a é o máximo
7      10
8      >>> maximo(-2, -1) # b é o máximo
9      -1
10     '''
11     if a > b:
12         m = a
13     else:
14         m = b
15     return m
```

Vamos treinar mais uma vez a execução passo a passo.

Qual é a ordem em que as linhas são executadas para o exemplo `maximo(10, 8)`?

11, 12, 15.

Qual é a ordem em que as linhas são executadas para o exemplo `maximo(-2, -1)`?

11, 14, 15.

Exemplo - atualização número de telefone

Como “descobrimos” que precisamos utilizar uma instrução de seleção?

Vamos voltar ao exemplo da atualização do número do telefone.

Exemplo - atualização número de telefone

No período de 2015 a 2016 todos os números de telefones celulares no Brasil passaram a ter nove dígitos. Na época, os números de telefones que tinham apenas oito dígitos foram alterados adicionando-se o 9 na frente do número. Embora oficialmente todos os números de celulares tenham nove dígitos, na agenda de muitas pessoas ainda é comum encontrar números registrados com apenas oito dígitos. Projete uma função que adicione o nono dígito em um dado número de telefone celular caso ele ainda não tenha o nono dígito. Considere que os números de entrada são dados com o DDD entre parênteses e com um hífen separando os últimos quatro dígitos. Exemplos de entradas: (44) 9787-1241, (51) 95872-9989, (41) 8876-1562. A saída deve ter o mesmo formato, mas garantindo que o número do telefone tenha 9 dígitos.

Análise

Ajustar o número de um telefone adicionando 9 como o nono dígito se necessário.

Definição de tipo de dados

O número de telefone é uma string no formato (XX) XXXX-XXXX ou (XX) XXXXX-XXXX, onde X pode ser qualquer dígito.

Especificação

A seguir.

Exemplo - atualização número de telefone

```
def ajusta_numero(numero: str) -> str:
    '''
    Ajusta *numero* adicionando o 9 como nono dígito se necessário, ou seja, se
    *numero* tem apenas 8 dígitos (sem contar o DDD).

    Requer que numero esteja no formato (XX) XXXX-XXXX ou (XX) XXXXX-XXXX, onde
    X pode ser qualquer dígito.

    Exemplos
    >>>
    >>> ajusta_numero('(51) 95872-9989')
    '(51) 95872-9989'
    >>>
    >>> ajusta_numero('(44) 9787-1241')
    '(44) 99787-1241'
    '''
    return numero
```

Exemplo - atualização número de telefone

```
def ajusta_numero(numero: str) -> str:
    '''
    Ajusta *numero* adicionando o 9 como nono dígito se necessário, ou seja, se
    *numero* tem apenas 8 dígitos (sem contar o DDD).

    Requer que numero esteja no formato (XX) XXXX-XXXX ou (XX) XXXXX-XXXX, onde
    X pode ser qualquer dígito.

    Exemplos
    >>> # não precisa de ajuste, a saída é a própria entrada
    >>> ajusta_numero('(51) 95872-9989')
    '(51) 95872-9989'
    >>> # '(44) 9787-1241'[:5] + '9' + '(44) 9787-1241'[5:]
    >>> ajusta_numero('(44) 9787-1241')
    '(44) 99787-1241'
    '''
    return numero
```

Exemplo - atualização número de telefone

Como “descobrimos” que precisamos utilizar uma instrução de seleção?

Até agora, todas as funções que projetamos tinham apenas uma “forma” de gerar o resultado.

Na função `ajusta_numero`, existem duas “formas” para a resposta: o próprio número ou o número ajustado.

Como escolher quando cada forma deve ser utilizada na resposta da função? Utilizando um condição:

- Se a quantidade de caracteres de `numero` for 15, então a resposta é `numero`;
- Senão a resposta é `numero[:5] + '9' + numero[5:]`.

Quando a resposta depende de uma ou mais condições, usamos uma instrução de seleção!

Exemplo - atualização número de telefone

```python
def ajusta_numero(numero: str) -> str:
    if len(numero) == 15:
        ajustado = numero
    else:
        ajustado = numero[:5] + '9' + numero[5:]
    return ajustado
```

Exemplo - máximo de 3

Projete uma função que encontre o valor máximo entre três números.

Exemplo - máximo de 3

Análise

- Encontrar o valor máximo entre três números dados

Tipos de dados

- Os valores serão números inteiros

Especificação (assinatura e propósito)

```
def maximo3(a: int, b: int, c: int) -> int:
    '''
    Encontra o valor máximo entre *a*, *b* e *c*.
    '''
```

Exemplo - máximo de 3

```
>>> maximo3(20, 10, 12) # a é o máximo
20
>>> maximo3(20, 12, 10)
20
>>> maximo3(20, 12, 12)
20
>>> maximo3(20, 20, 20)
20
>>> maximo3(5, 12, 3) # b é o máximo
12
>>> maximo3(3, 12, 5)
12
>>> maximo3(5, 12, 5)
12
>>> maximo3(4, 8, 18) # c é o máximo
18
>>> maximo3(8, 4, 18)
18
>>> maximo3(8, 8, 18)
18
```

Implementação

Quantas “formas” de resposta nós temos? 3. Ou a resposta é **a**, ou a resposta é **b**, ou a resposta é **c**.

Se temos formas de respostas diferentes, então a resposta depende de uma ou mais condições. Então, usamos instruções de seleção.

Qual é a condição para a resposta ser **a**?

```
a >= b and a >= c
```

Qual é a condição para a resposta ser **b**?

```
b >= a and b >= c
```

Qual é a condição para a resposta ser **c**?

```
c >= a and c >= b
```

Agora podemos escrever o corpo da função!

Exemplo - máximo de 3

```python
def maximo3(a: int, b: int, c: int) -> int:
    '''
    Encontra o valor máximo entre
    *a*, *b* e *c*.
    '''
    if a >= b and a >= c:
        m = a
    else:
        if b >= a and b >= c:
            m = b
        else: # c >= a and c >= b
            m = c
    return m
```

Vamos treinar mais uma vez a execução passo a passo.

Qual é a ordem em que as linhas são executadas para o exemplo a seguir:

`maximo3(10, 6, 8)`? 6, 7, 13.

`maximo3(10, 15, 8)`? 6, 9, 10, 13.

`maximo3(10, 15, 20)`? 6, 9, 12, 13.

Confira a execução desse código no Python Tutor.

Exemplo - máximo de 3

```
def maximo3(a: int, b: int, c: int) -> int:
    '''
    Encontra o valor máximo entre
    *a*, *b* e *c*.
    '''
    if a >= b and a >= c:
        m = a
    else:
        if b >= a and b >= c:
            m = b
        else: # c >= a and c >= b
            m = c
    return m
```

Verificação: ok.

Revisão

Podemos modificar o código para torná-lo mais fácil de ler e entender? Sim!

O Python permite “juntar” um `else` seguido de um `if` em um `elif`. Isto ajuda a diminuir os níveis de indentação, facilitando a escrita e leitura do código.

Exemplo - máximo de 3

```
def maximo3(a: int, b: int, c: int) -> int:
    '''
    Encontra o valor máximo entre
    *a*, *b* e *c*.
    '''
    if a >= b and a >= c:
        m = a
    elif b >= a and b >= c:
        m = b
    else: # c >= a and c >= b
        m = c
    return m
```

Análise de casos

Vamos parar por um momento e relembrar como fazemos a implementação de uma função.

Olhamos para a especificação, com atenção especial para os exemplos, e perguntamos: quantas formas de resposta temos nos exemplos?

- Se existe apenas uma forma de resposta, isto é, a resposta dos exemplos são sempre calculadas da mesma forma, então usamos essa forma para implementar a função.
- Se existe mais de uma forma, isto é, a resposta para pelo menos dois exemplos tem forma distinta, então precisamos usar seleção. Para cada forma de resposta identificamos uma condição e usamos as condições e as formas de resposta para implementar a função (o que fizemos na implementação da função `maximo3`).

Chamamos essa estratégia de **análise de casos**: identificar as formas de resposta nos exemplos e, para cada forma, determinar a condição que a distingue das demais.

A análise de casos pode ser implementada de duas maneiras:

- **Seleção direta**: criamos um caso para cada forma de resposta, com uma condição (possivelmente composta) para cada uma (como fizemos na implementação da função `maximo3`).
- **Seleção aninhada** (árvore de decisão): dividimos as formas de resposta em grupos verificando partes da condição separadamente.

No caso de mais de uma forma de resposta, a condição de cada forma pode ser composta, como no exemplo `maximo3`, onde a condição para a resposta ser `a` era `a >= b and a >= c` (a condição é composta por duas partes).

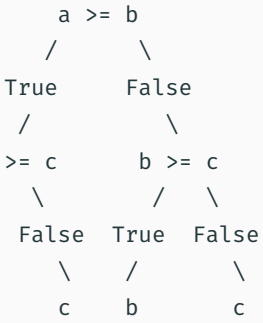
Nesses casos, podemos verificar cada parte da condição de forma separada. A cada verificação, dividimos as formas de resposta em dois grupos, as que precisam que a condição seja verdadeira e as que precisam que a condição seja falsa. Usando verificações subsequentes, vamos restringindo as formas de resposta até chegar em apenas uma forma.

Vamos tentar utilizar essa abordagem para fazer um implementação alternativa da função `maximo3`.

Exemplo - máximo de 3

Se `a >= b` é `True`, quais valores podem ser o máximo? Os valores de **a** e **c**. E como descobrimos quem é o máximo entre **a** e **c**? Fazendo outra seleção.

Se `a >= b` é `False`, quais valores podem ser o máximo? Os valores de **b** e **c**. E como descobrimos quem é o máximo entre **b** e **c**? Fazendo outra seleção.



Exemplo - máximo de 3

Versão alternativa

```
def maximo3(a: int, b: int, c: int) -> int:
    if a >= b:
        if a >= c:
            m = a
        else:
            m = c
    else:
        if b >= c:
            m = b
        else:
            m = c
    return m
```

Primeira versão

```
def maximo3(a: int, b: int, c: int) -> int:
    if a >= b and a >= c:
        m = a
    elif b >= a and b >= c:
        m = b
    else: # c >= a and c >= b
        m = c
    return m
```

Qual versão é mais fácil de entender? A primeira...

Podemos melhorar? Sim!

Exemplo - máximo de 3

```
1  def maximo3(a: int, b: int, c: int) -> int:
2      if a >= b:
3          if a >= c:
4              m = a
5          else:
6              m = c
7      else:
8          if b >= c:
9              m = b
10         else:
11             m = c
12     return m
```

Qual é o propósito do bloco das linhas 3 a 6? Encontrar o máximo entre **a** e **c**.

Qual é o propósito do bloco das linhas 8 a 11? Encontrar o máximo entre **b** e **c**.

Já temos uma função para encontrar o máximo entre dois números? Sim! A função `maximo` que fizemos anteriormente.

Então vamos usar a função!

Exemplo - máximo de 3

```
1  def maximo3(a: int, b: int, c: int) -> int:
2      if a >= b:
3          m = maximo(a, c)
4      else:
5          m = maximo(b, c)
6      return m
```

Qual é o propósito da seleção da linha 2? Encontrar o máximo entre **a** e **b**... Nós já temos uma função para fazer isso!

```
def maximo3(a: int, b: int, c: int) -> int:
    return maximo(maximo(a, b), c)
```

Poderíamos ter construído essa implementação na primeira vez?

Sim, mas nesse caso, deveríamos ter visto que as três formas de resposta distintas poderiam ter sido generalizadas em uma única forma, que é `maximo(maximo(a, b), c)`. Essa generalização direta requer prática, por enquanto, podemos fazer os casos distintos e tentar, durante a revisão, simplificar o código.

Exemplo - máximo de 3

```
1  def maximo(a: int, b: int) -> int:
2      if a > b:
3          m = a
4      else:
5          m = b
6      return m
7
8  def maximo3(a: int, b: int, c: int) -> int:
9      return maximo(maximo(a, b), c)
10
11 maximo3(10, 2, 15)
```

Vamos treinar mais uma vez a execução passo a passo.

Qual é a ordem em que as linhas são executadas para o exemplo ao lado?

11, 9, 2, 3, 6, 9, 2, 5, 6, 9, 11.

Confira o processo de execução desse código no Python Tutor.

Exemplo - ponto final

Em um determinado programa é necessário que o texto digitado pelo usuário termine com um ponto. Projete uma função que coloque um ponto final em um texto se ele ainda não terminar com ponto.

Análise

- Colocar um ponto final em um texto caso ele ainda não termine com ponto.

Definição dos tipos de dados

- O texto é representado por uma string.

Ponto final - especificação

```
def ponto_final(texto: str) -> str:
    '''
    Coloca um ponto final em *texto* se
    *texto* não termina com ponto final.

    Exemplos
    >>> # Não adiciona o ponto
    >>> ponto_final('Talvez.')
    'Talvez.'
    >>> # Adiciona ponto
    >>> ponto_final('Sim, eu gostaria')
    'Sim, eu gostaria.'
    '''
```

Essa especificação está completa? Não!

Está faltando considerar um caso extremo, quando `texto` é vazio.

Como proceder nesse caso? Temos duas opções:

- Definimos que vazio não é uma entrada válida (função parcial); ou
- Definimos uma saída para a entrada vazia (função total).

Vamos explorar as duas possibilidades.

Ponto final - especificação

Fazendo `ponto_final` uma função parcial.

Ponto final - especificação - vazio inválido

```
def ponto_final(texto: str) -> str:
    '''
    Coloca um ponto final em *texto* se
    *texto* não termina com ponto final.

    Requer que *texto* não seja vazio.

    Exemplos
    >>> # Não adiciona o ponto
    >>> ponto_final('Talvez.')
    'Talvez.'
    >>> # Adiciona ponto
    >>> ponto_final('Sim, eu gostaria')
    'Sim, eu gostaria.'
    '''
```

Implementação

Como temos duas formas de resposta, adiciona ou não o ponto, usamos seleção. A condição para não adicionar ponto é que `texto` termine com ponto.

```
def ponto_final(texto: str) -> str:
    assert texto != '', "texto não pode ser ''"
    if texto[len(texto) - 1] == '.':
        com_ponto = texto
    else:
        com_ponto = texto + '.'
    return com_ponto
```

Usamos o `assert` quando queremos expressar uma condição que precisa ser verdadeira para que o código continue executando. Caso a condição não seja verdadeira, o programa é interrompido (falha) com uma mensagem de erro.

O que acontece na função `ponto_final` se não utilizarmos o `assert` e a função for chamada com o argumento `''`?

Vai falhar na expressão `texto[len(texto) - 1]`, pois estamos querendo acessar o último caractere de uma string vazia.

Se usando ou não o `assert` o programa falha, porque utilizar o `assert`? Para que a falha tenha uma causa mais precisa, facilitando a depuração do programa.

Assert

```
>>> # sem assert
>>> ponto_final('')
Traceback (most recent call last):
    ...
    if texto[len(texto) - 1] == '.':
       ~~~~~^^^^^^^^^^^^^^^^
IndexError: string index out of range
>>> # Reação do usuário da função:
>>> # Que erro é esse?
```

```
>>> # com assert
>>> ponto_final('')
Traceback (most recent call last):
    ...
    ...
    ...
AssertionError: texto não pode ser ''
>>> # Reação do usuário da função:
>>> # Entendi.
```

Podemos pensar na especificação de uma função como um **contrato** entre o fornecedor (quem implementa a função) e o usuário (quem chama a função).

- O **usuário** se compromete a satisfazer as precondições (o `Requer` da especificação);
- O **fornecedor** se compromete a produzir o resultado descrito no propósito.

O `assert` é o mecanismo que usamos para verificar se o usuário está cumprindo a sua parte no contrato.

Ponto final - especificação

Fazendo `ponto_final` uma função total.

Ponto final - especificação - vazio válido

```
def ponto_final(texto: str) -> str:
    '''
    Coloca um ponto final em *texto* se
    *texto* não termina com ponto final
    e não é ''. Devolve *texto* caso
    contrário.

    Exemplos
    >>> # Não adiciona o ponto
    >>> ponto_final('')
    ''
    >>> ponto_final('Talvez.')
    'Talvez.'
    >>> # Adiciona ponto
    >>> ponto_final('Sim, eu gostaria')
    'Sim, eu gostaria.'
    '''
```

Implementação

Quantas formas temos? Três formas. Então usamos seleção.

```
def ponto_final(texto: str) -> str:
    if texto == ''
        com_ponto = ''
    elif text[len(texto) - 1] == '.':
        com_ponto = texto
    else:
        com_ponto = texto + '.'
    return com_ponto
```

Exemplo - centralizar string

Em uma determinada aplicação as strings precisam ser exibidas com pelo menos *n* caracteres, onde *n* pode variar dependendo da situação. Se uma string não tem *n* caracteres, é necessário adicionar espaços em branco no início e fim da string, deixando ela centralizada entre os espaços, para que ela seja exibida corretamente. Projete uma função que ajuste uma string dessa forma. Assuma que a string de entrada não tenha espaços no início e no final.

Exemplo - centralizar string

Análise

- Deixar uma string que tem menos de n caracteres com n caracteres adicionando espaços no início e no final da string.

Especificação

- A seguir

Exemplo - centralizar string

```
def centraliza(s: str, n: int) -> str:
    '''
    Produz uma string adicionando espaços
    no início e fim de *s*, se necessário,
    de modo que ela fique com *n*
    caracteres.

    Se *s* tem mais que *n* caracteres,
    devolve *s*.

    Exemplos
    >>> centraliza('casa', 3)
    'casa'
    >>> centraliza('', 0)
    ''
    '''
```

Qual deve ser a resposta para `centraliza('casa', 5)`? `' casa'` ou `'casa '`?

Não está claro no propósito da função, então vamos voltar e esclarecer esse ponto.

Exemplo - centralizar string

```
def centraliza(s: str, n: int) -> str:
    '''
    Produz uma string adicionando espaços
    no início e fim de *s*, se necessário,
    de modo que ela fique com *n*
    caracteres.

    Se *s* tem mais que *n* caracteres,
    devolve *s*.

    A quantidade de espaços adicionados no
    início é igual ou um a mais do que a
    quantidade adicionada no fim.
    '''
```

```
>>> centraliza('casa', 3)
'casa'
>>> centraliza('', 0)
''
>>> centraliza('casa', 10)
'   casa   '
>>> centraliza('casa', 9)
'   casa  '
>>> centraliza('apenas', 10)
'  apenas  '
>>> centraliza('apenas', 9)
'  apenas '
```

Exemplo - centralizar string

Temos dois casos: adiciona ou não os espaços.

Qual é a condição para não adicionar espaços? `len(s) >= n`.

Qual é o processo para adicionar os espaços?

Descobrir a quantidade de espaços, dividir em duas quantidades, a do início e a do fim, adicionar os espaços.

```
def centraliza(s: str, n: int) -> str:
    if len(s) >= n:
        r = s
    else:
        faltando = n - len(s)
        fim = faltando // 2
        inicio = faltando - fim
        r = ' ' * inicio + s + ' ' * fim
    return r
```

Exemplo - álcool ou gasolina

Depois que você fez o programa para o André, a Márcia, amiga em comum de vocês, soube que você está oferecendo serviços desse tipo e também quer a sua ajuda. O problema da Márcia é que ela sempre tem que fazer a conta manualmente para saber se deve abastecer o carro com álcool ou gasolina. A conta que ela faz é verificar se o preço do álcool é até 70% do preço da gasolina, se sim, ela abastece o carro com álcool, senão ela abastece o carro com gasolina. Você pode ajudar a Márcia também?

Análise

- Determinar o combustível que será utilizado. Se o preço do álcool for até 70% do preço da gasolina, então deve-se usar álcool, senão gasolina.

Definição de tipos de dados

- O preço do litro do combustível será representado por um número positivo;
- O tipo de combustível será representado por uma string.

Exemplo - álcool ou gasolina - especificação

```
def indica_combustivel(preco_alcool: float, preco_gasolina: float) -> str:
    '''
    Indica o combustível que deve ser utilizado no abastecimento. Produz
    'alcool' se *preco_alcool* for menor ou igual a 70% do *preco_gasolina*,
    caso contrário produz 'gasolina'.

    Exemplos
    >>> # 'alcool'
    >>> indica_combustivel(4.00, 6.00) # 4.00 <= 0.7 * 6.00 é True
    'alcool'
    >>> indica_combustivel(3.50, 5.00) # 3.50 <= 0.7 * 5.00 é True
    'alcool'
    >>> # 'gasolina'
    >>> indica_combustivel(4.00, 5.00) # 4.00 <= 0.7 * 5.00 é False
    'gasolina'
    '''
```

Exemplo - álcool ou gasolina - implementação

Quantas formas para a resposta existem? Duas: `'alcool'` e `'gasolina'`. Então precisamos usar seleção. Qual é a condição para que a resposta seja `'alcool'`?

```
preco_alcool <= 0.7 * preco_gasolina
```

```
def indica_combustivel(preco_alcool: float, preco_gasolina: float) -> str:
    if preco_alcool <= 0.7 * preco_gasolina:
        combustivel = 'alcool'
    else:
        combustivel = 'gasolina'
    return combustivel
```

Exemplo - álcool ou gasolina - verificação e revisão

Verificação: ok.

Revisão: string não parece ser um tipo de dado apropriado...

Vamos continuar na próxima aula...

Quando precisamos usar seleção na implementação de uma função?

- Quando a análise de casos identifica mais de uma forma de resposta nos exemplos.

Quais são as duas formas de implementar a análise de casos?

- Seleção direta: um caso para cada forma de resposta, com condição possivelmente composta.
- Seleção aninhada (árvore de decisão): divide as formas em grupos verificando partes da condição separadamente.

O que é o contrato de uma função?

- O usuário se compromete a satisfazer as precondições e o fornecedor se compromete a produzir o resultado descrito no propósito.

Para que serve o `assert`?

- Para verificar se o usuário está cumprindo as precondições. Se a condição não for verdadeira, o programa falha com uma mensagem de erro clara.