Tipos de dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Definição de tipos de dados

Durante a etapa de definição de tipos de dados identificamos as informações e definimos como elas são representadas no programa.

Essa etapa pode ter parecido, até então, muito simples ou talvez até desnecessária, isto porque as informações que precisávamos representar eram “simples”.

No entanto, essa etapa é muito importante no projeto de programas, de fato, uma representação adequada pode facilitar a escrita do programa e diminuir as possibilidades de erros, aumentando a confiabilidade do programa.

Mas o que exatamente é um tipo de dado e como projetar um tipo de dado adequado para representar uma informação?

Tipos de dados

Um **tipo de dado** é o conjunto de valores que uma variável pode assumir.

Exemplos

- `bool` $= \{$ `True` , `False` $\}$
- `int` $= \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- `float` $= \{\ldots, -0.1, -0.0, 0.0, 0.1, \ldots\}$
- `str` $= \{$ `''`, `'a'`, `'b'`, $\ldots\}$

Requisitos de um tipo de dado

Um inteiro é adequado para representar a quantidade de pessoas em um planeta?

- Não é adequado pois um número inteiro pode ser negativo mas a quantidade de pessoas em um planeta não pode, ou seja, o tipo de dado *permite a representação de valores inválidos*.

O ideal seria um número natural, mas o Python não tem um tipo de dado específico para representar apenas números naturais. Outras linguagens oferecem outras opções. Por exemplo, em Rust temos `u32` (0 a 4.294.967.296) e `u64` (0 a 18.446.744.073.709.551.616).

`u32` seria adequado para representar a quantidade de pessoas em um planeta?

- Não pois o número de pessoas no planeta terra não está no intervalo de valores válidos para o tipo, ou seja, *nem todos os valores válidos poder ser representados*.

Requisitos de um tipo de dado

Durante a etapa de definição de tipos de dados devemos levar em consideração as seguintes diretrizes:

- Faça os valores válidos representáveis.
- Faça os valores inválidos irrepresentáveis.

Quando fizemos o projeto da função `indica_combustivel` escolhemos o tipo `str` para representar a informação do tipo de combustível.

```
def indica_combustivel(preco_alcool: float, preco_gasolina: float) -> str
```

Essa escolha é adequada?

Não! Muitos valores de `str` não correspondem a nenhum valor válido para a informação do tipo de combustível. Além disso, para o leito do código, a saída do tipo `str` sugere que qualquer string é possível como resposta, o que não é verdade.

Como proceder nesse caso? Vamos definir um novo tipo onde apenas os valores para álcool e gasolina são válidos.

Tipos enumerados

Em um **tipo enumerado** todos os valores do tipo são enumerados explicitamente.

A forma geral para definir tipos enumerados é

```
from enum import Enum, auto

class NomeDoTipo(Enum):
    VALOR1 = auto()
    ...
    VALORN = auto()
```

Vamos definir um tipo enumerado para representar o tipo de combustível.

Definição do tipo `Combustivel`

```
class Combustivel(Enum):
    '''O tipo do combustivel em um abastecimento'''
    ALCOOL = auto()
    GASOLINA = auto()
```

`auto()` é utilizado para associar automaticamente um número com o valor da enumeração. Se quisermos, podemos escolher um número explicitamente.

Sempre vamos adicionar um comentário sobre o propósito do tipo, se necessário, adicionamos comentários para os valores da enumeração.

Uso de tipo enumerado

Cada valor da enumeração tem dois atributos: `name` e `value`.

```
>>> c: Combustivel = Combustivel.ALCOOL
>>> c
<Combustivel.ALCOOL: 1>
>>> c.value
1
>>> c.name
'ALCOOL'
```

```
>>> c = Combustivel.GASOLINA
>>> c
<Combustivel.GASOLINA: 2>
>>> c.value
2
>>> c.name
'GASOLINA'
```

Assim como uma variável do tipo `bool` só pode armazenar os valores `True` e `False`, uma variável do tipo `Combustivel` só pode armazenar o valor `Combustivel.ALCOOL` ou `Combustivel.GASOLINA`, se tentarmos atribuir um valor diferente, o `mypy` indicará um erro.

```
c: Combustivel = 'Alcool'
```

Erro

```
error: Incompatible types in assignment (expression has type "str",
                                         variable has type "Combustivel")
```

Quando usar tipos enumerados?

Quando usar tipos enumerados?

Quando todos os valores válidos para o tipo podem ser nomeados.

Por que utilizar tipos enumerados?

Para expressar mais claramente o propósito do código e evitar a utilização de valores inválidos (como `'alcoo'` em uma variável string que representa o tipo do combustível).

Revisão do projeto de `indica_combustivel`

```python
def indica_combustivel(preco_alcool: float, preco_gasolina: float) -> Combustivel:
    '''
    Exemplos
    >>> indica_combustivel(4.00, 6.00).name
    'ALCOOL'
    >>> indica_combustivel(3.50, 5.00).name
    'ALCOOL'
    >>> indica_combustivel(4.00, 5.00).name
    'GASOLINA'
    '''
    if preco_alcool <= 0.7 * preco_gasolina:
        combustivel = Combustivel.ALCOOL
    else:
        combustivel = Combustivel.GASOLINA
    return combustivel
```

Exemplo - Semáforo

Projete uma função que receba como entrada a cor atual de um semáforo de trânsito e devolva a próxima cor que será exibida (considere um semáforo com três cores: verde, amarelo e vermelho).

Análise

- Determinar a próxima cor de um semáforo dado a cor atual

Exemplo - Semáforo

Projeto de tipos de dados

- Quais são as informações? A cor do semáforo.
- Como representar essa informação? Com um tipo enumerado.

```
from enum import Enum, auto

class Cor(Enum):
    '''O cor de um semáforo de trânsito'''
    VERDE = auto()
    VERMELHO = auto()
    AMARELO = auto()
```

Exemplo - Semáforo

Especificação

```
def proxima_cor(atual: Cor) -> Cor:
    '''
    Produz a próxima cor de uma semáfaro
    que está na cor *atual*.
    '''
```

Quantos exemplos precisamos? Pelo menos 3, um para cada valor da enumeração.

```
    >>> proxima_cor(Cor.VERDE).name
    'AMARELO'
    >>> proxima_cor(Cor.AMARELO).name
    'VERMELHO'
    >>> proxima_cor(Cor.VERMELHO).name
    'VERDE'
```

Implementação

Para a implementação podemos olhar ou para as formas de respostas ou para a entrada que é uma enumeração. Nesse caso, vamos olhar para entrada. Como são três valores de entrada, então podemos começar o código com uma condição para cada valor.

```
def proxima_cor(atual: Cor) -> Cor:
    if atual == Cor.VERDE:
        ...
    elif atual == Cor.AMARELO:
        ...
    elif atual == Cor.VERMELHO:
        ...
    return ...
```

Exemplo - Semáforo

Especificação

```
def proxima_cor(atual: Cor) -> Cor:
    '''
    Produz a próxima cor de uma semáfaro
    que está na cor *atual*.
    '''
```

Quantos exemplos precisamos? Pelo menos 3, um para cada valor da enumeração.

```
    >>> proxima_cor(Cor.VERDE).name
    'AMARELO'
    >>> proxima_cor(Cor.AMARELO).name
    'VERMELHO'
    >>> proxima_cor(Cor.VERMELHO).name
    'VERDE'
```

Implementação

Para a implementação podemos olhar ou para as formas de respostas ou para a entrada que é uma enumeração. Nesse caso, vamos olhar para entrada. Como são três valores de entrada, então podemos começar o código com uma condição para cada valor.

```
def proxima_cor(atual: Cor) -> Cor:
    if atual == Cor.VERDE:
        proximo = Cor.AMARELO
    elif atual == Cor.AMARELO:
        proximo = CoR.VERMELHO
    elif atual == Cor.VERMELHO:
        proximo = Cor.VERDE
    return ...
```

Exemplo - tempo

Em um determinado programa é necessário exibir para o usuário o tempo que uma operação demorou. Esse tempo está disponível em segundos, mas exibir essa informação em segundos para o usuário pode não ser interessante, afinal, ter uma noção razoável de tempo para 14678 segundos é difícil!

a) Projete uma função que converta uma quantidade de segundos para uma quantidade de horas, minutos e segundos equivalentes.

b) Projete uma função que converta uma quantidade de horas, minutos e segundos em uma string amigável para o usuário (algo como 1 hora, 10 minutos e 2 segundos). A string não deve conter valores zeros.

Exemplo - tempo - parte a

Análise

- Converter uma quantidade de segundos em horas, minutos e segundos.

Definição de tipos de dados

- O segundos da entrada será representado com um número inteiro positivo
- A saída são três números inteiros positivos... As funções em Python só podem produzir um valor de saída, como proceder? Vamos criar um novo tipo de dado que agrupa esses três valores.

Tipos de dados

Vamos relembrar alguns tipos de dados que utilizamos até agora:

- Tipos atômicos pré-definidos na linguagem: `int`, `float`, `bool`, `str`
- Tipos enumerados definidos pelo usuário: `Combustivel`, `Cor`

Os tipos atômicos têm esse nome porque não são compostos por partes.

Podemos criar novos tipos agregando partes (campos) de tipos já existentes.

Uma forma de fazer isso é através de tipos compostos (estruturas).

Um **tipo composto** é um tipo de dado composto por um conjunto fixo de campos cada um com nome e tipo.

A forma geral para definir um tipo composto é

```
from dataclasses import dataclass

@dataclass
class NomeDoTipo:
    campo1: Tipo1
    ...
    campon: TipoN
```

Tipos compostos

Podemos definir um novo tipo para representar um tempo da seguinte forma

```
@dataclass
class Tempo:
    horas: int
    minutos: int
    segundos: int
```

O que está faltando nessa definição? Um comentário sobre o propósito do tipo.

Podemos definir um novo tipo para representar um tempo da seguinte forma

```
@dataclass
class Tempo:
    '''
    Representa o tempo de duração de um evento.
    horas, minutos e segundos devem ser positivos.
    '''
    horas: int
    minutos: int
    segundos: int
```

O que está faltando nessa definição? Um comentário sobre o propósito do tipo.

Para criar um valor de um tipo composto, chamamos o **construtor** (função que é criada automaticamente pelo Python) para o tipo e especificamos os valores dos campos na ordem que eles foram declarados.

```
>>> t1: Tempo = Tempo(0, 20, 10)
>>> t1
Tempo(horas=0, minutos=20, segundos=10)

>>> # A anotação do tipo é opcional
>>> t2 = Tempo(4, 0, 20)
>>> t2
Tempo(horas=4, minutos=0, segundos=20)
```

Como valores do tipo `Tempo` são compostos de outros valores (campos), podemos acessar e alterar cada campo de forma separada.

```
>>> t1 = Tempo(0, 20, 10)
>>> t1.segundos
10
>>> t1.minutos
20
>>> t1.horas
0
```

```
>>> t1.horas = 3
>>> t1
Tempo(horas=3, minutos=20, segundos=10)
>>> # Podemos deixar o valor em um
>>> # estado inconsistente...
>>> t1.segundos = 70
Tempo(horas=3, minutos=20, segundos=70)
```

Exemplo - tempo - parte a

Vamos voltar para o problema.

Análise

- Converter uma quantidade de segundos em horas, minutos e segundos.

Definição de tipos de dados

- O segundos da entrada será representado com um número inteiro positivo
- As horas, minutos e segundos serão representadas por um dado composto `Tempo`.

Como fica a assinatura da função?

```
def segundos_para_tempo(segundos: int) -> Tempo
```

Agora vamos concluir o projeto da função.

Exemplo - tempo - parte a

```
def segundos_para_tempo(segundos: int) -> Tempo:
    '''
    Converte a quantidade *segundos* para o tempo equivalente
    em horas, minutos e segundos.
    '''
    return Tempo(0, 0, 0)
```

Qual deve ser a resposta para `segundos_para_tempo(3760)`?

Pelo propósito, pode ser `Tempo(0, 0, 3760)`? Pode! Então precisamos corrigir o propósito!

Qual de fato é a resposta que esperamos? `Tempo(1, 2, 40)`.

Qual é o processo que usamos para encontrar a resposta?

Exemplo - tempo - parte a

```
def segundos_para_tempo(segundos: int) -> Tempo:
    '''
    Converte a quantidade *segundos* para o tempo
    equivalente em horas, minutos e segundos.
    A quantidade de segundos e minutos da resposta
    é sempre menor que 60.
    Requer que segundos seja não negativo.

    Exemplos
    >>> # 160 // 60 -> 2 mins, 160 % 60 -> 40 segs
    >>> segundos_para_tempo(160)
    Tempo(horas=0, minutos=2, segundos=40)
    >>> # 3760 // 3600 -> 1 hora
    >>> # 3760 % 3600 -> 160 segundos que sobraram
    >>> # 160 // 60 -> 2 mins, 160 % 60 -> 40 segs
    >>> segundos_para_tempo(3760)
    Tempo(horas=1, minutos=2, segundos=40)
    '''
    return Tempo(0, 0, 0)
```

Quantas formas de respostas nós temos?

Podemos generalizar para apenas uma forma que utiliza uma sequência de instruções.

```
def segundos_para_tempo(segundos: int) -> Tempo:
    assert segundos >= 0, 'segundos precisa ser >= 0
    h = segundos // 3600
    # segundos que não foram
    # convertidos para hora
    restantes = segundos % 3600
    m = restantes // 60
    s = restantes % 60
    return Tempo(h, m, s)
```

Quando usar dados compostos?

Quando usar dados compostos?

Quando a informação consiste de dois ou mais itens que juntos descrevem uma entidade.

Em um determinado programa é necessário exibir para o usuário o tempo que uma operação demorou. Esse tempo está disponível em segundos, mas exibir essa informação em segundos para o usuário pode não ser interessante, afinal, ter uma noção razoável de tempo para 14678 segundos é difícil!

a) Projete uma função que converta uma quantidade de segundos para uma quantidade de horas, minutos e segundos equivalentes.

b) Projete uma função que converta uma quantidade de horas, minutos e segundos em uma string amigável para o usuário (algo como 1 hora, 10 minutos e 2 segundos). A string não deve conter valores zeros.

Agora vamos fazer o item b.

Exemplo - tempo - parte b

Análise

- Converter um tempo (horas, minutos, segundos) para uma string amigável para o usuário, sem escrever os componentes que sejam 0.

Definição de tipos de dados

- Já fizemos

Exemplo - tempo - parte b

Especificação

```
def tempo_para_string(t: Tempo) -> str:
    '''
    Converte *t* em uma mensagem para o usuário. Cada componente de *t* aparece
    com a sua unidade, mas se o valor do componente for 0, ele não aparece na
    mensagem. Os componentes são separados com "e" ou "," respeitando as regras
    do Português. Se *t* for Tempo(0, 0, 0), devolve "0 segundo(s)".
    '''
    return ''
```

Quantos exemplos precisamos?

Cada componente pode ser 0 ou não, como são três componentes teríamos $2 \times 2 \times 2 = 8$ casos distintos.

De fato só precisamos de 7, pois o valor dos segundos não importa no caso em que horas e minutos são zero.

Exemplo - tempo - parte b

```
>>> # horas == 0 and minutos == 0
>>> tempo_para_string(Tempo(0, 0, 0))
'0 segundo(s)'
>>> tempo_para_string(Tempo(0, 0, 1))
'1 segundo(s)'
>>> tempo_para_string(Tempo(0, 0, 10))
'10 segundo(s)'

>>> # horas == 0 and minutos != 0 \
>>> #            and segundos != 0
>>> tempo_para_string(Tempo(0, 1, 20))
'1 minuto(s) e 20 segundo(s)'

>>> # horas == 0 and minutos != 0 \
>>> #            and segundos == 0
>>> tempo_para_string(Tempo(0, 2, 0))
'2 minuto(s)'
```

```
>>> # horas != 0 and minutos != 0 and segundos != 0
>>> tempo_para_string(Tempo(1, 2, 1))
'1 hora(s), 2 minuto(s) e 1 segundo(s)'

>>> # horas != 0 and minutos == 0 and segundos != 0
>>> tempo_para_string(Tempo(4, 0, 25))
'4 hora(s) e 25 segundo(s)'

>>> # horas != 0 and minutos != 0 and segundos == 0
>>> tempo_para_string(Tempo(2, 4, 0))
'2 hora(s) e 4 minuto(s)'

>>> # horas != 0 and minutos == 0 and segundos == 0
>>> tempo_para_string(Tempo(3, 0, 0))
'3 hora(s)'
```

Exemplo - tempo - parte b

Quantas formas de respostas existem? 7! Então temos que usar seleção.

A implementação direta usando as condições de cada forma fica com exercício.

A implementação a seguir usa condições aninhadas.

Exemplo - tempo - parte b

```
def tempo_para_string(Tempo t) -> str:
    h = str(t.horas) + ' hora(s)'
    m = str(t.minutos) + ' minuto(s)'
    s = str(t.segundos) + ' segundo(s)'
    if t.horas > 0:
        if t.minutos > 0:
            if t.segundos > 0:
                msg = h + ', ' + m + ' e ' + s
            else:
                msg = h + ' e ' + m
        elif t.segundos > 0:
            msg = h + ' e ' + s
        else:
            msg = h
    elif t.minutos > 0:
        if t.segundos > 0:
            msg = m + ' e ' + s
        else:
            msg = m
    else:
        msg = s
    return msg
```

Compare com a sua implementação direta.

Qual das duas é mais simples e fácil de entender?

Exercício

Modifique a especificação e implementação da função anterior para que o plural dos componentes fique de acordo com o Português.

Exemplo - Loteria

Em um jogo de loteria os apostadores fazem apostas escolhendo 6 números distintos entre 1 e 60. No sorteio são sorteados 6 números de forma aleatória. Os apostadores que acertam 4, 5 ou 6 números são contemplados com prêmios.

a) Projete uma função que verifique se um número está entre os sorteados.

b) Projete uma função que determine quantos números uma determinada aposta acertou.

Análise

- Determinar se um número está entre 6 números sorteados.
- Determinar o número de acertos de uma aposta de 6 números sendo que 6 números foram sorteados;
- Os números estão entre 1 e 60;

Definição de tipos de dados

- Quais são as informações? A aposta e os sorteados, as duas informações são compostas de 6 números.
- Como representar a aposta de 6 números? E os 6 número sorteados? Com um dado composto.

```python
from dataclasses import dataclass

@dataclass
class SeisNumeros:
    '''Coleção de 6 números distintos entre 1 e 60.'''
    a: int
    b: int
    c: int
    d: int
    e: int
    f: int
```

As apostas e os números sorteados serão representados pela estrutura `SeisNumeros`.

Vamos fazer a especificação da primeira função.

Exemplo - Loteria - especificação `sorteado`

```
def sorteado(n: int, sorteados: SeisNumeros) -> bool:
    '''
    Produz True se *n* é um dos números
    em *sorteados*. False caso contrário.
    '''
    return False
```

Quantos exemplos precisamos? 7, `n` igual a cada um dos sorteados e `n` diferentes de todos.

Exemplo - Loteria - especificação `sorteado`

```
def sorteado(n: int, sorteados: SeisNumeros) -> bool:
    '''
    >>> sorteados = SeisNumeros(1, 7, 10, 40, 41, 60)
    >>> sorteado(1, sorteados)
    True
    >>> sorteado(7, sorteados)
    True
    >>> sorteado(10, sorteados)
    True)
    >>> sorteado(40, sorteados)
    True
    >>> sorteado(41, sorteados)
    True
    >>> sorteado(60, sorteados)
    True
    >>> sorteado(2, sorteados)
    False
    '''
    return False
```

Quantas formas de respostas nós temos? Duas, ou a resposta é `True` ou a resposta é `False`.

Então precisamos usar seleção. Qual a condição para a resposta `True`?

```
n == sorteados.a or \
    n == sorteados.b or \
    n == sorteados.c or \
    n == sorteados.d or \
    n == sorteados.e or \
    n == sorteados.f
```

Agora podemos fazer a implementação.

Exemplo - Loteria - implementação `sorteado`

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

Verificação: ok

Revisão: podemos melhorar o código?

Sim!

O código de `sorteado` tem forma:

```
if condição:
    r = True
else:
    r = False
return r
```

que pode ser simplificada para

```
return condição
```

Exemplo - Loteria - implementação `sorteado`

```
def sorteado(n: int,
             sorteados: SeisNumeros)
             -> bool:
    return n == sorteados.a or \
        n == sorteados.b or \
        n == sorteados.c or \
        n == sorteados.d or \
        n == sorteados.e or \
        n == sorteados.f
```

Podemos fazer uma implementação alternativa? Sim, fazendo seleções aninhadas uma condição por vez:

Se `n == a`, então a resposta é `True`;

Senão, quais podem ser as respostas? `True` ou `False` , então precisamos de outra seleção:

- Se `n == b`, então a resposta é `True`;
- Senão, quais podem ser as respostas? `True` ou `False`...

Exemplo - Loteria - implementação `sorteado`

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

Podemos fazer outra implementação? Sim!

Forma de implementação de função

Até agora vimos três formas de implementar uma função:

- Direta, uma única expressão (ou sequência de expressões): `custo_viagem`, `massa_tubo_ferro`, `segundos_para_tempo`, `sorteado` - uma versão, etc.
- Seleção direta, seleção com uma condição para cada forma de resposta: `maximo`, `ajusta_numero`, `indica_combustivel`, `maximo3` - uma versão, `sorteado` - uma versão, etc.
- Seleção aninhada, seleção aninhada até determinar a forma da resposta: `maximo3` - uma versão, `tempo_para_string`, `sorteado` - uma versão, etc.

Agora veremos uma nova forma de implementação: a forma incremental.

Abordagem incremental

Na **abordagem incremental**, iniciamos a resposta com um valor e vamos atualizando a resposta conforme o processamento avança, no final, temos a resposta da função.

Vamos aplicar esse abordagem para implementar a função `sorteado`.

A resposta que queremos é se o número `n` está entre os `sorteados`. O que precisamos?

- Um valor inicial para a resposta;
- Uma forma de atualizar a resposta conforme analisamos a entrada.

Começamos a resposta com `False`, se `n == a` mudamos a resposta pra `True`, se `n == b` mudamos a resposta pra `True`, e assim por diante.

Exemplo - Loteria - implementação `sorteado`

```
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
```

Observe que, apesar do código ser semelhante ao da implementação anterior, o processo pelo qual escrevemos esse código foi diferente.

Além disso, essa forma vai nos permitir um tipo de simplificação que veremos em breve.

Agora podemos ir para a segunda função do problema da loteria: determinar o número de acertos de uma aposta.

Exemplo - Loteria - especificação num_acertos

```
def numero_acertos(aposta: SeisNumeros, sorteados: SeisNumeros) -> int:
    '''
    Determina quantos números da *aposta* estão em *sorteados*.
    Exemplos
    >>> numero_acertos(SeisNumeros(1, 2, 3, 4, 5, 6), SeisNumeros(8, 12, 20, 41, 52, 57))
    0
    >>> numero_acertos(SeisNumeros(8, 2, 3, 4, 5, 6), SeisNumeros(8, 12, 20, 41, 52, 57))
    1
    >>> numero_acertos(SeisNumeros(8, 12, 3, 4, 5, 6), SeisNumeros(8, 12, 20, 41, 52, 57))
    2
    >>> numero_acertos(SeisNumeros(8, 12, 20, 4, 5, 6), SeisNumeros(8, 12, 20, 41, 52, 57))
    3
    >>> numero_acertos(SeisNumeros(8, 12, 20, 41, 5, 6), SeisNumeros(8, 12, 20, 41, 52, 57))
    4
    >>> numero_acertos(SeisNumeros(8, 12, 20, 41, 52, 6), SeisNumeros(8, 12, 20, 41, 52, 57))
    5
    >>> numero_acertos(SeisNumeros(8, 12, 20, 41, 52, 57), SeisNumeros(8, 12, 20, 41, 52, 57))
    6
    '''
    return 0
```

Exemplo - Loteria - especificação `num_acertos`

Que estratégia nós usamos para calcular as respostas dos exemplos? Ou ainda, que estratégia podemos utilizar para implementar a função? A estratégia incremental!

O que precisamos para implementar a função usando a estratégia incremental?

- Um valor inicial para a resposta;
- Uma forma de atualizar a resposta conforme analisamos a entrada.

Exemplo - Loteria - especificação `num_acertos`

Começamos o número de acertos com zero.

Depois verificamos se o primeiro número está entre os sorteados, se sim, aumentamentos os acertos em 1.

Depois verificamos se o segundo número está entre os sorteados, se sim, aumentamentos os acertos em 1.

E assim com o restante dos números.

No final, temos a quantidade de acertos.

```
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
```

Verificação: ok

Revisão: assim como para a função `sorteado`, o código parece repetitivo... Como resolver essa questão?

Usando instrução de repetição! Vamos continuar na próxima aula.