Seleção, enumerações e estruturas

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Antes de estudarmos instruções de seleção, 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 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 que as linhas são executadas?

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

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

Agora veremos a **instrução de seleção** `if else` (se e senão 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 o valor exibido pelo programa? 20.

Em que ordem as linhas são executas 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 o valor exibido pelo programa? 15.

Em que ordem as linhas são executas 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`.

Máximo

Vamos projetar uma função para encontrar o máximo entre dois números.

```
1  def maximo(a: int, b: int) -> int:
2      '''Devolve o valor máximo entre *a* e *b*.
3      Exemplos
4      >>> # a é o máximo
5      >>> maximo(10, 8)
6      10
7      >>> # b é o máximo
8      >>> maximo(-2, -1)
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 que as linhas são executadas para o exemplo `maximo(10, 8)`?

11, 12, 15

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

11, 14, 15

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

Vamos voltar ao exemplo da atualização do número do telefone.

No período de 2015 à 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 frete do número. Embora oficialmente todos os número 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.

```
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 e 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
```

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!

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

Máximo de 3

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

Análise

- Encontrar o valor máximo entre três número 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*.
    '''
```

```
>>> 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!

```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 que as linhas são executadas para o exemplo a seguir:

`maximo(10, 6, 8)`? 6, 7, 13

`maximo(10, 15, 8)`? 6, 9, 10, 13

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

```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
```

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.

```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
    elif b >= a and b >= c:
        m = b
    else: # c >= a and c >= b
        m = c
    return m
```

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 a 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`).

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 da condição e as que não precisam da condição. Usando verificação subsequentes, vamos restringindo as opções de forma de resposta até chegar em apenas uma forma.

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

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.

Implementação alternativa

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!

```
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 o propósito do bloco das linhas de 3 à 6? Encontrar o máximo entre `a` e `c`.

Qual o propósito do bloco das linhas de 8 à 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!

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

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

Poderíamos ter chegado nessa 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.

Execução passo a passo

```
1  def maximo(a: int, b: int) -> int:
2      if a > b:
3          m = a
4      else:
5          m = b
6      return a
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 que as linhas são executadas para o exemplo ao lado?

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

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; ou
- Definimos uma saída para a entrada vazia.

Vamos explorar as duas possibilidades.

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 != ''
    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 (crasha) 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 crashar 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 crasha, porque utilizar o **assert**? Para que a falha tenha uma causa mais precisa, facilitando a depuração do programa.

```
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

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

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

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.

```
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'
    '''
```

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
```

Verificação: ok.

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

Vamos parar um momento e conversar sobre a etapa de 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?

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 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 temos que 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. Essa escolha é adequada?

Não! Muitos valores válidos para `str` não correspondem a nenhum valor válido para a informação do tipo de combustível.

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 válidos para o 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.

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

```
>>> c = 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`

```
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

Projeto de tipos de dados

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

Exemplo - Semáforo

```
from enum import Enum, auto

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

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

Implementação

São três formas de resposta, então usamos seleção com uma condição para cada forma.

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

Verificação: Ok.

Revisão: Ok.

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

- Os segundos da entrada serão representados com números inteiros positivos
- 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 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:
    '''
    Representa o tempo de duração de um evento.
    horas, minutos e segundos devem ser positivos.
    minutos e segundos devem ser menores que 60.
    '''
    horas: int
    minutos: int
    segundos: int
```

Assim como para definição de tipos enumerados, sempre vamos adicionar um comentário sobre o propósito do tipo.

Para inicializar uma variável de um tipo composto, chamamos o construtor (função) 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)
```

```
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 resposta nós temos?

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

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

Verificação: ok

Revisão: ok

Dados compostos

Quando utilizamos dados compostos?

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

Exemplo - tempo - parte b

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.

```
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 ''
```

```
>>> # 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)'
```

Quantas formas de resposta 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.

Implementação

```
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)'
    # Temos 7 formas distintas
    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
```

Implementação alternativa

```
def tempo_para_string(Tempo t) -> str:
    # usado para separar cada componente de t
    sep = ''
    msg = ''
    if t.segundos > 0:
        sep = ' e '
        msg = str(t.segundos) + ' segundo(s)'

    if t.minutos > 0:
        msg = str(t.minutos) + ' minuto(s)' + sep + msg
        if t.segundos > 0:
            sep = ', '
        else:
            sep = ' e '

    if t.horas > 0:
        msg = str(t.horas) + ' hora(s)' + sep + msg

    if msg == '':
        msg = '0 segundo(s)'

    return msg
```

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!

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.

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.