Revisão

Estruturas de Dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Antes de começarmos o conteúdo da disciplina, vamos fazer uma revisão:

- Processo de projeto de programas
- A linguagem Python
- Projeto de programas em Python

Projeto de programas

Quais são as principais atividades no projeto de um programa?

- Análise (identificação do problema)
- Especificação (descrição do que o programa deve fazer)
- Implementação
- Verificação (a implementação atende à especificação?)

Podemos detalhar esse processo para o projeto de uma função específica:

- Análise: identificar o problema a ser resolvido
- Definição dos tipos de dados: identificar e definir como as informações serão representadas
- Especificação: especificar com precisão o que a função deve fazer
- Implementação: implementar a função de acordo com a especificação
- Verificação: verificar se a implementação está de acordo com a especificação
- Revisão: identificar e fazer melhorias

Mas como projetar programas?

Um programa é composto de várias funções, então decompomos o programa em funções e aplicamos esse processo para projetar cada função.

A linguagem Python

Como aprender uma nova linguagem?

Para aprender uma linguagem de programação, temos que, de maneira simplificada, aprender:

- Tipos primitivos de dados
- Operações primitivas
- Definição de novas operações
- Estruturas de controle
- Definição de novos tipos de dados

Instalação

O Python é um software livre e pode ser baixado e instalado de https://python.org.

Apesar do Python vir com um ambiente de desenvolvimento e aprendizado chamado IDLE, nessa disciplina não vamos usá-lo.

Para desenvolver os nossos programas vamos utilizar o vscode.

O Python tem diversos **tipos numéricos**, os dois principais são

Inteiros (`int`)

```
>>> 102
102
>>> -18
-18
```

Ponto flutuante (`float`), representação aproximada de números reais

```
>>> 1.3
1.3
>>> 0.345
0.345
>>> # Notação científica
>>> 1.23e2 # 1.23 * 10^2
123.0
```

Podemos usar as quatro operações aritméticas básicas com esses tipos numéricos e algumas outras operações.

Operadores básicas

```
>>> # Soma e subtração
>>> 4 + 2
6
>>> 4 + 2.0 - 5
1.0
```

```
>>> # Multiplicação e divisão
>>> 3 * 5.0
15.0
>>> 7 / 2
3.5
```

```
>>> # Divisão sempre produz float
>>> 8 / 4
2.0
```

```
>>> # Piso da divisão
>>> 7 // 2
3
>>> 5 // 1.3
3.0
```

```
>>> # Módulo
>>> 14 % 3
2
>>> -14 % 3
1
```

```
>>> # float é uma aproximação dos reais
>>> 5 % 1.3
1.0999999999999999
```

Comentários

O símbolo # (cerquilha), é utilizado para indicar um **comentário**. O comentário inicia na # e vai até o final da linha. Os comentários são ignorados pelo interpretador do Python, mas são utilizados para adicionar informações relevantes para os leitores do código.

Exponenciação

```
>>> # Exponenciação e radiciação
>>> 3 ** 4 # 3 elevado a 4
81
>>> 2 ** 80
1208925819614629174706176
>>> # raiz quadrada, o mesmo que 16 ** (1 / 2)
>>> 16 ** 0.5
4.0

>>> # A exponenciação tem prioridade sobre a divisão
>>> # O mesmo que (27 ** 1) / 3
>>> 27 ** 1 / 3
9.0
>>> # Usamos parênteses para mudar a prioridade
>>> 27 ** (1 / 3) # raiz cúbica
3.0
```

```
>>> # Arredondamento
>>> round(3.4)
3
>>> round(3.5)
4
>>> round(-1.6)
-2
>>> round(3.5134, 2)
3.51
```

```
>>> # Conversão entre int e float
>>> int(7.6)
7
>>> int(-2.3)
-2
>>> float(4)
4.0
```

Piso e teto

```
>>> # Importação do módulo
>>> import math
```

```
>>> # Piso
>>> # maior inteiro <= ao número
>>> math.floor(4.2)
4
>>> math.floor(4.0)
4
>>> math.floor(-2.3)
-3
```

```
>>> # Teto
>>> # menor inteiro >= ao número
>>> math.ceil(4.2)
5
>>> math.ceil(4.0)
4
>>> math.ceil(-2.3)
-2
```

Outro tipo de dado pré-definido em Python é a cadeia de caracteres (`str`), *string* em inglês.

Geralmente usamos strings para armazenar informações simbólicas, como por exemplo palavras e textos.

Uma string em Python é escrita entre apóstrofos (`'`) ou aspas (`"`)

```
>>> 'casa'
'casa'
>>> "gota d'agua"
"gota d'agua"
>>> "mesa"
'mesa'
```

Operações com strings

Assim como existem operações pré-definidas para números, também existem operações pré-definidas para strings.

```
>>> # Concatenação
>>> 'casa' + ' da ' + 'sogra'
'casa da sogra'

>>> # Repetição
>>> 'abc' * 3
'abcabcabc'

>>> 'algum' * 0
''
>>> 'algum' * -4
''
```

```
>>> # Quantidade de caracteres
>>> len('ciência da computação')
21

>>> # Conversão maiúscula
>>> 'José'.upper() # ou str.upper('José')
'JOSÉ'

>>> # Conversão minúscula
>>> 'José'.lower() # ou str.lower('José')
'josé'
```

```
>>> # Indexação de caractere
>>> # O primeiro caractere tem índice 0
>>> 'casa'[0] # ou str.__getitem__('casa', 0)
'c'

>>> 'casa'[1]
'a'

>>> # Acesso de índice fora do intervalo
>>> 'casa'[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

```
>>> # Substring do início até 3 - 1
>>> 'veja isso'[:3] # ou str.__getitem__('veja isso', slice(None, 3))
'vej'

>>> # Substring de 4 até o final
>>> 'veja isso'[4:] # ou str.__getitem__('veja isso', slice(4, None))
' isso'

>>> # Substring de 2 até 6 - 1
>>> 'veja isso'[2:6] # ou str.__getitem__('veja isso', slice(2, 6))
'ja i'
```

Conversão entre strings e números

```
>>> # Conversão de int para str
>>> str(127)
'127'
>>> # Conversão de float para str
>>> str(4.1)
'4.1'

>>> # Concatenação de str e int
>>> 'Idade: ' + str(19)
'Idade: 19'
```

```
>>> # Conversão de str para int
>>> int('127')
127

>>> # Conversão de str para float
>>> float('25')
25.0
>>> float('12.67')
12.67
```

Inicialmente as expressões que vimos usavam apenas operadores matemáticos

```
30 * 2
```

Depois vimos que as expressões podem conter chamadas de funções

```
round(3.5)
```

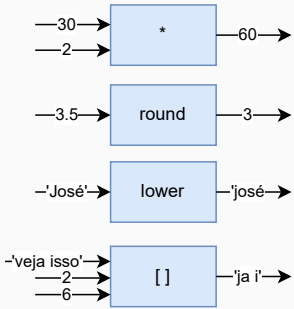
e chamadas de métodos

```
'José'.lower()
```

Por fim, vimos que strings podem ser indexadas

```
'veja isso'[2:6]
```

Embora a forma de utilizar operadores, funções, métodos e indexação seja diferente, o propósito dessas construções é o mesmo: computar valores de saída a partir de valores de entrada.



Se o propósito é o mesmo, por que não usar a mesma forma?

Por conveniência!

Por exemplo, se não tivéssemos a forma de operadores e apenas a forma de chamada de funções, então, para escrever a expressão `30 * 2 + 3` teríamos que escrever

```
int.__add__(int.__mul__(30, 2), 3)
```

Além da conveniência de escrita, a forma de chamada métodos e indexação tem outras vantagens, que não vamos discutir nessa disciplina.

```
>>> # Maior e maior ou igual
>>> 4 > 4
False
>>> 4 >= 4
True
```

```
>>> # Menor e menor ou igual
>>> 6.0 < 6.0
False
>>> 6.0 <= 1.0 + 5.0
True
```

```
>>> # Igual
>>> 5 == 6
False
>>> 9 == 5 + 2 ** 2
True
```

```
>>> # Diferente
>>> 3 * 2 != 4 + 2 ** 2
True
>>> 9 != 4 + 2 ** 2
False
```

Quem tem maior prioridade, os operadores relacionais ou aritméticas? Os aritméticos.

As operações relacionais podem ser utilizadas com outros tipos, incluindo strings e booleanos.

As strings são comparadas lexicograficamente, o que pode gerar algumas surpresas.

```
>>> 'a' < 'b'
True
>>> 'á' < 'b'
False
>>> 'Paulo' < 'andré'
True
```

```
>>> 'Abacaxi' < 'Abacate'
False
>>> 'André' < 'paulo'
True
>>> 'casa' == 'Casa'
False
>>> 'A' != 'a'
True
```

O valor `False` é considerado menor que o valor `True`, isso porque o `False` quando convertido para `int` é `0` e o `True` é `1`.

```
>>> int(False)
0
>>> int(True)
1
>>> False < True
True
>>> True > False
True
```

```
>>> False == False
True
>>> False == True
False
>>> True == False
False
>>> True == True
True
```

Operadores lógicos

Assim como existem operações com números e strings, também existem operações com booleanos, que são chamadas de **operações lógicas**.

As operações mais comuns com booleanos são: `not` (negação), `or` (ou) e `and` (e).

O **not** é um operador unário, que produz o valor contrário do seu argumento.

```
>>> not True
False
>>> not False
True
>>> not not True
True
```

```
>>> # 4 > 4.0 é False
>>> not 3 + 1 > 2 + 2.0
True
>>> # 14 == 14 é True
>>> not 2 + 3 * 4 == 14
False
```

Qual é a precedência do **not** em relação aos operadores relacionais e aritméticos? É menor.

O **and** é um operador binário que só produz `True` se os dois operandos forem `True`.

```
>>> # Tabela verdade do and
>>> False and False
False
>>> False and True
False
>>> True and False
False
>>> True and True
True
```

Qual é a precedência do **and** em relação aos operadores relacionais e aritméticos? É menor.

```
>>> # 15 > 8 é True
>>> # 4 == 3 é False
>>> 15 > 2 ** 3 and 4 == 1 + 2
False
>>> # 2 == 2 é True
>>> # 3 != 4 é True
>>> 2 == 1 + 1 and 3 != 4
True
```

O **or** é um operador binário que produz **True** se pelo menos um dos operandos for **True**.

```
>>> # Tabela verdade do or
>>> False or False
False
>>> False or True
True
>>> True or False
True
>>> True or True
True
```

Qual é a precedência do **or** em relação aos operadores relacionais e aritméticos? É menor.

```
>>> # 15 > 8 é True
>>> # 4 == 3 é False
>>> 15 > 2 ** 3 or 4 == 1 + 2
True
>>> # 2 == 3 é False
>>> # 3 + 1 != 4 é False
>>> 2 == 2 + 1 or 3 + 1 != 4
False
```

Quem tem maior prioridade, o **and** ou o **or**? O **and**. Vamos criar uma expressão que mostre que isso é verdade.

```
>>> True or False and False
True

>>> # É equivalente a expressão anterior
>>> True or (False and False)
True

>>> # Se o or tivesse prioridade...
>>> (True or False) and False
False
```

Os arranjos em Python são dinâmicos, isto é, podem mudar de tamanho, e são representados pelo tipo `list`.

Listas - inicialização e indexação

```
>>> # Inicialização
>>> x: list[int] = [9 + 1, 1, 7, 2]
>>> x
[10, 1, 7, 2]

>>> # Lista vazia
>>> y = [] # ou list()
>>> y
[]

>>> # Número de elementos
>>> len(x)
4
>>> len(y)
0
```

```
>>> # Indexação
>>> nomes = ['Maria', 'João', 'Paulo']
>>> nomes[1]
'João'
>>> # Acesso fora da faixa
>>> nomes[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> # Sublistas
>>> x = [4, 1, 5, 7, 3]
>>> x[:2]
[4, 1]
>>> x[2:]
[5, 7, 3]
```

Listas - alteração, acréscimo e concatenação

```
>>> # Substituição de um elemento
>>> y = [4, 2]
>>> y[1] = 7
>>> y
[4, 7]

>>> # Acréscimo de um elemento
>>> y.append(5) # list.append(y, 5)
>>> y
[4, 7, 5]
>>> y.append(3)
>>> y
[4, 7, 5, 3]

>>> # Concatenação
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
```

Aprendizagem de uma nova linguagem

- ~~Tipos primitivos de dados~~
- ~~Operações primitivas~~
- Definição de novas operações
- Estruturas de controle
- Definição de novos tipos de dados

A forma geral para definição de funções em Python é:

```
def nome(entrada1: tipo, entrada2: tipo, ...) -> tipo:
    instruções
    return exp
```

```
def soma_quadrados(a: int, b: int) -> int:
    '''Calcula a soma de *a* ao quadrado e *b* ao quadrado'''
    a2 = a ** 2
    return a2 + b * b
```

Uso da nova função

```
>>> soma_quadrados(3, 4)
25
```

Veja o material de Memória e passagem de parâmetros na página https://malbarbo.pro.br/ensino/2025/6879/.

Veja o arquivo `parametros.py` no arquivo com os exemplos para download na página da disciplina.

Use a página https://pythontutor.com/visualize.html para executar os exemplos (lembre-se de remover as anotações de tipo).

- ~~Tipos primitivos de dados~~
- ~~Operações primitivas~~
- ~~Definição de novas operações~~
- Estruturas de controle
- Definição de novos tipos de dados

A forma geral do **`if else`** é:

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

Exemplo seleção

```
def maximo(a: int, b: int) -> int:
    '''
    Devolve o valor máximo
    entre *a* e *b*.
    '''
    if a > b:
        m = a
    else:
        m = b
    return m
```

Exemplos

```
>>> maximo(10, 8)
10
>>> maximo(-2, -1)
-1
```

Seleção aninhada

```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
```

```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
```

A forma geral do “para cada” é

```
for var in lista:
    instruções
```

O “para cada” funciona da seguinte maneira:

- O primeiro valor de `lista` é atribuído para `var` e as `instruções` são executadas;
- O segundo valor de `lista` é atribuído para `var` e as `instruções` são executadas;
- ...
- E assim por diante até que todos os valores de `lista` tenham sido atribuídos para `var`.

Ou seja, o “para cada” executa as mesmas instruções atribuindo cada valor de `lista` para `var`, por isso ele se chama “para cada”!

Exemplo do “para cada”

```
>>> for x in [6, 1, 4, 5]:
...     print(x)

6
1
4
5
```

Exemplo do “para cada”

```python
def soma(lst: list[int]) -> int:
    '''Soma os elementos de *lst*.'''
    soma = 0
    for n in lst:
        soma = soma + n
    return soma
```

Exemplos

```
>>> soma([])
0
>>> soma([3])
3
>>> soma([3, 7])
10
>>> soma([3, 7, 2])
12
```

Exemplo de execução passa a passo do “para cada”

```
1  def soma(lst: list[int]) -> int:
2      soma = 0
3      for n in lst:
4          soma = soma + n
5      return soma
6
7  soma([5, 1, 4])
```

Qual é a ordem em que as linhas são executadas?

7, 2 (`soma = 0`),
3 (`n = 5`), 4 (`soma = 5`),
3 (`n = 1`), 4 (`soma = 6`),
3 (`n = 4`), 4 (`soma = 10`),
3 (identifica que não tem mais elementos), 5 (devolve `10`),
7

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`.

Exemplo do “para cada no intervalo”

```
>>> for i in range(3, 7):
...     print(i)

3
4
5
6
```

```
>>> lst = [4, 1, 5, 2]
>>> for i in range(len(lst)):
...     print(i, lst[i])

0 4
1 1
2 5
3 2
```

Exemplo do “para cada no intervalo”

Função que soma os valores de uma lista

```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 formas é mais simples? A da esquerda!

Exemplo do “para cada no intervalo”

Função que encontra o índice da primeira ocorrência do valor máximo de uma lista não vazia.

```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
```

```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!

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

Exemplos do “enquanto”

```
>>> i = 4
>>> while i < 10:
...     print(i)
...     i = 2 * i

4
8
```

```
>>> i = 0
>>> x = 12
>>> while x >= 3:
...     x = x - 3
...     i = i + 1
>>> i

4
```

Exemplo de execução passo a passo do “enquanto”

```
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 em 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

Quando utilizar cada forma de repetição?

Para cada

- Quando queremos processar todos os elementos de uma sequência na ordem que eles aparecem

Para cada no intervalo

- Quando queremos processar um intervalo dos elementos de uma sequência; ou
- Quando precisamos dos índices dos elementos da sequência; ou
- Quando precisamos de uma sequência de números em progressão aritmética

Enquanto

- Quando o uso do “para cada” e do “para cada no intervalo” não são adequados

- ~~Tipos primitivos de dados~~
- ~~Operações primitivas~~
- ~~Definição de novas operações~~
- ~~Estruturas de controle~~
- Definição de novos 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'`, `...` $\}\}$

Durante a etapa de definição de tipos de dados (do processo de projeto de funções) temos que determinar quais são as informações e como elas serão representadas.

Algumas informações podem ser representadas diretamente com os tipos primitivos da linguagem. Para outras informações, precisamos definir novos tipos de dados.

Quais características são desejáveis no projeto/definição de um tipo de dado?

- Que as informações válidas possam ser representadas.
- Que as informações inválidas não possam ser representadas.

Em um programa de simulação precisamos representar a cor de um semáforo (que pode ser verde, vermelha ou amarela) e projetar uma função que indique qual deve ser a próxima cor de uma semáforo a partir da cor atual.

Qual tipo de dados podemos utilizar?

String é um tipo adequado? Não, pois `'casa'` é uma string mas não representa uma informação (cor) válida.

Como fazemos nesse caso? Criamos um tipo enumerado.

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()
```

Exemplo tipo enumerado

```python
from enum import Enum, auto

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

```
>>> c = Cor.VERDE
>>> c
<Cor.VERDE: 1>
>>> c.value
1
>>> c.name
'VERDE'
>>> c == Cor.VERDE
True
>>> Cor.VERDE == Cor.VERMELHO
False
```

Exemplo tipo enumerado

```
def proxima_cor(atual: str) -> str:
    '''
    Produz a próxima cor de uma semáfaro
    que está na cor *atual*.
    '''
    if atual == 'verde':
        proxima = 'amarelo'
    elif atual == 'amarelo':
        proxima = 'vermelho'
    elif atual == 'vermelho':
        proxima = 'verde'
    return proxima
```

```
>>> proxima_cor('verde')
'amarelo'
```

```
def proxima_cor(atual: Cor) -> Cor:
    '''
    Produz a próxima cor de um semáfaro
    que está na cor *atual*.
    '''
    if atual == Cor.VERDE:
        proxima = Cor.AMARELO
    elif atual == Cor.AMARELO:
        proxima = Cor.VERMELHO
    elif atual == Cor.VERMELHO:
        proxima = Cor.VERDE
    return proxima
```

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

Observe que o uso de tipo enumerado deixa a intenção do código mais clara:

- Entrada e saída do tipo `str` pode ser qualquer string!
- Entrada e saída do tipo `Cor` deixa claro quais valores são válidos.

O uso de tipo enumerado também permite a detecção de erros mais facilmente:

- Se digitarmos `'amarela'` ao invés de `'amarelo'` o programa ainda é “válido”.
- Se digitarmos `Cor.AMARELA` ao invés de `Cor.AMARELO` o `mypy` identifica o erro.

Em um determinado programa o tempo de uma atividade é medida em segundos, mas é preciso exibir esse tempo em horas, minutos e segundos. Para isso precisamos projetar uma função que converte uma quantidade de segundos para uma quantidade de horas, minutos e segundos equivalentes.

Os segundos da entrada da função pode ser representados com um número inteiro positivo, mas como representar a saída (o tempo em h, m, s)?

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: `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
```

Exemplo 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.

Exemplo tipos compostos

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)
```

Exemplo tipos compostos

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 tipos compostos

```
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.
    '''
    assert segundos >= 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)
```

Exemplos

```
>>> segundos_para_tempo(160)
Tempo(horas=0, minutos=2, segundos=40)
>>> segundos_para_tempo(3760)
Tempo(horas=1, minutos=2, segundos=40)
```

- ~~Processo de projeto de programas~~
- ~~A linguagem Python~~
- Projeto de programas na linguagem Python

Projeto de programas em Python

O projeto de programa envolve:

- Análise (identificação o problema)
- Especificação (descrição do que o programa deve fazer)
- Implementação
- Verificação (a implementação atende a especificação?)

Podemos detalhar esse processo para o projeto de uma função específica:

- Análise: identificar o problema a ser resolvido
- Definição dos tipos de dados: identificar e definir como as informações serão representadas
- Especificação: especificar com precisão o que a função deve fazer
- Implementação: implementar a função de acordo com a especificação
- Verificação: verificar se a implementação está de acordo com a especificação
- Revisão: identificar e fazer melhorias

O que é específico da linguagem e ainda não vimos como fazer em Python:

- Especificação
- Verificação

O que é a especificação?

Uma descrição precisa do que a função faz (deve fazer).

Note que escrevemos a especificação **antes de fazer a implementação**.

Como fazer a implementação se não sabemos exatamente o que precisa ser feito!?

Especificação em Python

A especificação de uma função consiste de

- Assinatura (nome, entradas e saída)
- Propósito (o que a função deve fazer / faz)
- Exemplos

Em Python, escrevemos o propósito e os exemplos em um comentário dentro da função.

Vamos escrever a especificação de uma função que calcula a quantidade de vezes que um inteiro aparece em uma lista.

```
def conta(lst: list[int], n: int) -> int:
    '''
    Conta a quantidade de vezes que o
    valor de *n* aparece em *lst*.

    Exemplos
    >>> conta([], 3)
    0
    >>> conta([5, 1, 3], 3)
    1
    >>> conta([4, 1, 2, 4], 4)
    2
    '''
    return 0
```

Note que para o código ficar bem formado deixamos um **return** com um valor padrão.

O que escrever no propósito?

No propósito escrevemos **o que** a função deve fazer / faz.

Devemos usar os nomes dos parâmetros na descrição do propósito para que a relação da entrada e da saída fique clara.

Para diferenciar o nome do parâmetro de uma palavra “normal”, colocamos um asterisco antes e depois do nome do parâmetro:

```
'''
Conta a quantidade de vezes que o valor de *n* aparece em *lst*.
'''
```

No propósito também escrevemos as **restrições** da entrada. Por exemplo, em uma função que encontra o valor máximo de uma lista, a lista não pode ser vazia:

```
def maximo(lst: list[int]) -> int:
    '''
    Encontra o valor máximo de *lst*.

    Requer que *lst* não seja vazia.
    '''
    return 0
```

A seguir discutimos alguns **erros comuns** na escrita do propósito.

Escrever **como** a função faz ao invés de escrever **o que** a função faz:

```
'''
Analisa cada elemento de *lst* e soma 1 em um contador
caso o elemento seja igual a *n*. No final, devolve
o contador.
'''
```

Isto é como a função faz e pode ser visto na implementação, não é preciso "narrar" o que já está na implementação!

Compare com o que a função faz:

```
'''
Conta a quantidade de vezes que o valor de *n* aparece em *lst*.
'''
```

Normalmente existe mais de uma forma de implementar uma função, e para usar uma função, em geral, é mais útil saber o que a função faz do que como a função faz.

Vamos ver ao longo da disciplina a importância de separar a interface (o que) da implementação (como).

Não referenciar os parâmetros.

```
'''
Conta quantas vezes um número aparece em uma lista.
'''
```

Qual Número? Qual lista?

Usar palavras desnecessárias.

```
'''
A função conta...

Esta função...
'''
```

Usar verbo no infinitivo.

```
'''
Contar quantas vezes...
'''
```

A função não contar, a função conta!

Qual o propósito dos exemplos?

O primeiro propósito é ajudar o projetista a entender como a saída pode ser determinada a partir das entradas. Por isso escrevemos os exemplos **antes de fazer a implementação**.

Depois os exemplos são usados como parte da verificação.

Como escrever os exemplos?

O exemplos são escritos na forma de uma sessão iterativa do Python. Uma linha que inicia com >>> indica uma instrução, as seguintes (até uma linha sem nada) indicam a resposta esperada:

```
'''
Exemplos
>>> conta([], 3)
0
>>> conta([5, 1, 3], 3)
1
>>> conta([4, 1, 2, 4], 4)
2
'''
```

Quantos exemplos escrever?

Pelo menos um exemplo para cada “forma” de calcular a saída.

Note que devemos evitar escrever muitos exemplos “iguais”. Por exemplo, para uma função que indica se um número natural é par ou ímpar, basta fazer um ou dois exemplos de números pares e ímpares, não é preciso fazer cinco exemplos de números pares.

O que é a verificação?

É o processo para determinar se a implementação está de acordo com a especificação.

A verificação pode ser:

- Estática: sem executar o programa
- Dinâmica: executando o programa

Apesar de ser possível provar, usando verificação estática, que um programa está de acordo com a sua especificação, este processo é custoso e inviável para a maioria dos programas.

No entanto, podemos usar a verificação estática para identificar diversas inconsistências entre a especificação e a implementação, principalmente em relação aos tipos de dados.

Para fazer a verificação estática dos tipos vamos utilizar o programa `mypy`.

Instalação do `mypy`:

```
pip install mypy
```

Execução do `mypy`:

```
mypy arquivo.py
```

Verificação estática

```python
def conta(lst: list[int], n: int) -> int:
    '''
    Conta a quantidade de vezes que o
    valor de *n* aparece em *lst*.

    Exemplos
    >>> conta([], 3)
    0
    >>> conta([4, 1, 2, 4], 4)
    2
    '''
    vezes = 0.0
    for x in lst:
        if x == n:
            vezes = vezes + 1
    return vezes
```

```
conta.py:16: error: Incompatible return
value type (got "float", expected "int")
Found 1 error in 1 file (checked 1 source file)
```

Existem muitas estratégias de verificação dinâmica, a que vamos utilizar inicialmente é baseada em exemplos, onde é verificado se as funções para entradas específicas produzem as saídas esperadas (que já são conhecidas).

Como já temos exemplos de entradas e saídas na especificação, podemos usar inicialmente esses exemplos para fazer a verificação dinâmica.

O módulo `doctest`, que já vem com o Python, analisa os comentários das funções, identifica e executa automaticamente os exemplos indicando caso haja divergências entre os valores esperados e os valores obtidos com a execução dos exemplos.

Execução do `doctest`:

```
python -m doctest -v arquivo.py
```

Verificação dinâmica

```python
def conta(lst: list[int], n: int) -> int:
    '''
    Conta a quantidade de vezes que o
    valor de *n* aparece em *lst*.

    Exemplos
    >>> conta([], 3)
    0
    >>> conta([4, 1, 2, 4], 4)
    2
    '''
    vezes = 0
    for x in lst:
        if x == n:
            vezes = vezes + 10
    return vezes
```

```
Trying:
    conta([4, 1, 2, 4], 4)
Expecting:
    2
**********************************************
File "conta.py", line 9, in conta.conta
Failed example:
    conta([4, 1, 2, 4], 4)
Expected:
    2
Got:
    20
1 items had no tests:
    conta
**********************************************
1 items had failures:
   1 of   2 in conta.conta
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

Verificação dinâmica

```python
def conta(lst: list[int], n: int) -> int:
    '''
    Conta a quantidade de vezes que o
    valor de *n* aparece em *lst*.

    Exemplos
    >>> conta([], 3)
    0
    >>> conta([4, 1, 2, 4], 4)
    2
    '''
    vezes = 0
    for x in lst:
        if x == n:
            vezes = vezes + 1
    return vezes
```

```
Trying:
    conta([], 3)
Expecting:
    0
ok
Trying:
    conta([4, 1, 2, 4], 4)
Expecting:
    2
ok
1 items had no tests:
    conta
1 items passed all tests:
   2 tests in conta.conta
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Exemplo

A seguir vamos ver um problema e o projeto de uma função para resolver o problema.

Para exemplos mais detalhados, veja esse documento.

Exemplo

Considere um jogo onde o personagem está em um tabuleiro (semelhante a um tabuleiro de jogo de xadrez). As linhas e colunas do tabuleiro são enumeradas de 1 a 10, dessa forma, é possível representar a posição (casa) do personagem pelo número da linha e da coluna que ele se encontra. O personagem fica virado para uma das direções: norte, sul, leste ou oeste. O jogador pode avançar o seu personagem qualquer número de casas na direção que ele está virado, mas é claro, não pode sair do tabuleiro. Projete uma função que indique a partir das informações do personagem, qual é o número máximo de casas que ele pode avançar na direção que ele está virado.

Objetivo: identificar o problema que deve ser resolvido.

Um personagem de um jogo está em um tabuleiro com 10 linhas e 10 colunas (enumeradas de 1 a 10) e está virado para uma das direções: norte, sul, leste ou oeste.

Para o norte, aumenta o número da linha (sul diminui) – essa é uma decisão que tomamos pois não é informado no enunciado.

Para o leste, aumenta o número da coluna (oeste diminui).

Determinar quantas casas no máximo um personagem pode avançar a partir da sua posição e direção (o personagem não pode sair do tabuleiro)

Objetivo: identificar e definir como as informações serão representadas.

Informações: direção, posição, personagem.

Tipos de dados

```
class Direcao(Enum):
    '''
    A direção em que o personagem está virado.
    Para o norte o número da linha aumenta,
    para o sul diminui.
    Para o leste o número da coluna aumenta,
    para o sul diminui.

    ^
    |     N
    |     |
    | O -   - L
    |     |
    |     S
    --------------->
    '''
    NORTE = auto()
    LESTE = auto()
    SUL = auto()
    OESTE = auto()
```

```
@dataclass
class Personagem:
    '''
    A posição e direção que um personagem se
    encontra no tabuleiro.
    '''
    # A linha, deve estar entre 1 e 10
    lin: int
    # A coluna, deve estar entre 1 e 10
    col: int
    dir: Direcao
```

Objetivo: especificar com precisão e com exemplos o que o programa deve fazer.

Para cada direção, precisamos de dois exemplos: um em que é possível avançar e outro que não.

Especificação

```
def maximo_casas(p: Personagem) -> int:
    ''' Determina o número máximo de casas que o personagem *p* pode avançar
    considerando a sua posição atual e a direção que ele está virado.
    >>> maximo_casas(Personagem(lin=4, col=2, dir=Direcao.NORTE))
    6
    >>> maximo_casas(Personagem(lin=10, col=2, dir=Direcao.NORTE))
    0
    >>> maximo_casas(Personagem(lin=4, col=2, dir=Direcao.SUL))
    3
    >>> maximo_casas(Personagem(lin=1, col=2, dir=Direcao.SUL))
    0
    >>> maximo_casas(Personagem(lin=4, col=2, dir=Direcao.LESTE))
    8
    >>> maximo_casas(Personagem(lin=4, col=10, dir=Direcao.LESTE))
    0
    >>> maximo_casas(Personagem(lin=4, col=2, dir=Direcao.OESTE))
    1
    >>> maximo_casas(Personagem(lin=4, col=1, dir=Direcao.OESTE))
    0
    '''
```

Objetivo: escrever o corpo da função para que ela faça o que está na especificação.

A forma da resposta da função depende da direção, então fazemos um caso para cada direção.

```
def maximo_casas(p: Personagem) -> int:
    if p.dir == Direcao.NORTE:
        casas = 10 - p.lin
    elif p.dir == Direcao.SUL:
        casas = p.lin - 1
    elif p.dir == Direcao.LESTE:
        casas = 10 - p.col
    elif p.dir == Direcao.OESTE:
        casas = p.col - 1
    return casas
```

Objetivo: verificar se a implementação está de acordo com a especificação.

Usamos o `mypy` para fazer uma verificação estática:

```
$ mypy maximo_casas.py
Success: no issues found in 1 source file
```

Usamos o `doctest` para verificar se os exemplos produzem as respostas esperadas (verificação dinâmica).

```
$ python -m doctest -v maximo_casas.py
...
1 items passed all tests:
   8 tests in maximo_casas.maximo_casas
8 tests in 7 items.
8 passed and 0 failed.
Test passed.
```

Objetivo: alterar a organização do programa para que fique mais fácil de ser lido, entendido e modificado.

Veja o material da página https://malbarbo.pro.br/ensino/2025/6879/.

Faça os exercícios da página https://malbarbo.pro.br/ensino/2025/6879/.