Uma visão geral do Python para professores que usam C

Marco A L Barbosa malbarbo.pro.br

29 de maio de 2024

Sumário

1 Introdução

Neste texto apresentamos uma visão geral da linguagem Python para professores que vão ministrar algoritmos ou estrutura de dados em Python e que têm experiência com o ensino usando C. O objetivo é esclarecer alguns pontos e mostrar como as construções algorítmicas de C podem ser expressas em Python. Note que o objetivo não é apresentar todas as construções do Python e nem descrever uma metodologia de ensino de algoritmos e estruturas de dados.

Começamos com algumas diferenças entre as duas linguagens, depois discutimos as principais construções do

Python, em seguida mostramos a implementação de alguns algoritmos e estruturas de dados e por fim apresentamos algumas convenções de código para Python e indicamos algumas referências.

2 Diferenças entre Python e C

As linguagens Python e C têm muitos aspectos distintos, pois são comumente usadas em situações distintas. Nessa seção discutimos algumas dessas diferenças. Usamos os seguintes exemplos para ilustrar algumas dessas diferenças.

Código em C

```c
#include <stdio.h>

/*
 * Devolve a quantidade de items em valores que estão no intervalo [min, max].
 *
 * n é quantidade de items em valores que é analisada.
 */
int conta_no_intervalo(int* valores, int n, int min, int max)
{
    // O correto seria declarar n, i, quant e o retorno da função como size_t.
    // Veja https://stackoverflow.com/q/66527638/5189607
    int quant = 0;
    for (int i = 0; i < n; i++) { // as chaves poderiam ser omitidas nesse caso
        if (min <= valores[i] && valores[i] <= max)
            quant++;
    }
    return quant;
}

int main(int argc, char* argv[])
{
    printf("Este programa conta a quantidade de valores em um determinado intervalo.\n");

    int n;
    printf("Digite a quantidade de valores: ");
    if (scanf("%d", &n) != 1 || n <= 0) {
        printf("Valor inválido.\n");
        return 1;
    }

    // Alocação dinâmica implícita na pilha
    int valores[n];
    // Ou alocação explícita no heap
    // int* valores = (int *) malloc(n * sizeof(int));

    // O retorno de scanf não é verificado nas
    // próximas chamadas para simplificar o código.
    printf("Digite %d número(s) separados por espaço.\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &valores[i]);
    }

    int min;
    printf("Digite o limite inferior do intervalo: ");
    scanf("%d", &min);

    int max;
    printf("Digite o limite superior do intervalo: ");
    scanf("%d", &max);
```

```c
    printf("Existe(m) %d valor(es) no intervalo.\n", conta_no_intervalo(valores, n, min, max));

    // Desalocação explícita no caso de alocação no heap
    // free(valores);
}
```

Código em Python

```python
def conta_no_intervalo(valores: list[int], min: int, max: int) -> int:
    '''
    Devolve a quantidade de items em *valores* que estão no intervalo [min, max].

    Exemplo
    >>> conta_no_intervalo([2, 5, 1, 4, 6, 8], 2, 6)
    4
    '''
    quant = 0
    # Com tipo explícito
    # quant: int = 0
    for valor in valores:
        if min <= valor and valor <= max:
            # Com comparações encadeadas
            # min <= valor <= max
            quant = quant + 1
            # Com atribuição com incremento
            # quant += 1
    return quant

def main():
    print('Este programa conta a quantidade de valores em um determinado intervalo.')

    s = input('Digite os valores separados por espaço: ')
    valores = []
    for valor in s.split():
        valores.append(int(valor))

    min = int(input('Digite o limite inferior do intervalo: '))
    max = int(input('Digite o limite superior do intervalo: '))

    print('Existe(m)', conta_no_intervalo(valores, min, max), 'valor(es) no intervalo.')

if __name__ == "__main__":
    main()
```

2.1 Forma geral da sintaxe

A sintaxe do C é baseada em blocos delimitados por chaves (`{}`), a indentação do código não influencia a semântica. As chaves são opcionais nas instruções de controle quando apenas uma sentença é especificada. Na definição de estruturas e funções as chaves são sempre necessárias. As expressões/sentenças das instruções de controle devem ser especificadas entre parênteses. Os operadores lógicos são especificados com símbolos (`!`, `||`, `&&`).

Os blocos em Python inciam com ":" e são delimitados pela indentação. As expressões/sentenças das instruções de controle não precisam ser especificadas entre parênteses. Os operados lógicos são especificados com as palavras chaves **`not`**, **`or`** e **`and`**.

Em Python os operadores relacionais pode ser encadeados, dessa forma é possível escrever `a < b < c` ao invés de `a < b and b < c`.

Em C os tipos das variáveis são especificadas antes do nome da variável. Já em Python, os tipos são especificados após o nome da variável, pela sintaxe `: tipo`. As especificação do tipo em C é obrigatória, em Python é opcional. Aspectos da semântica de tipos são discutidas na seção 2.3.

Python utiliza a palavra chave `def` para definições de funções e o tipo de retorna da função é especificado com `-> tipo`

Em Python não existe operadores de pós e pré incremento/decremento, mas existem os operadores de atribuição com incremento/decremento (entre outros).

Strings em Python podem ser especificadas com aspas (`"`) ou apóstrofo (`'`). Strings com múltiplas linhas são especificadas com três aspas ou apóstrofos.

Em C existem dois tipos de comentário, o de linha (`//`) e o de bloco (`/* */`). Python também tem dois tipos de comentário, o de linha (`#`) e o de documentação (`''' '''`), que deve aparecer apenas como primeiro item de uma definição.

2.2 Forma de execução

C é usado comumente como um linguagem compilada. Isso implica que para testar qualquer trecho de código em C é necessário escrever um programa completo (com função `main`), compilar e depois executar o programa. Além disso, para exibir o valor de uma variável é necessário chamar uma função como `printf` explicitamente (também é possível consultar os valores das variáveis usando um depurador).

Já o Python é usado comumente como uma linguagem interpretada de duas maneira, a primeira é especificando o nome de um arquivo `.py`, nesse caso o código do arquivo é executado diretamente (compilado para código de uma máquina virtual e depois executado).

A segunda maneira é sem especificar um arquivo, nesse caso é iniciado um modo interativo para avaliação de trechos de código (REPL - *Read, Eval, Print and Loop*). O *prompt* padrão é identificado com `>>>`. O usuário digita um trecho de código, que é executado e o resultado é exibido em seguida. Por exemplo

```
>>> 'Resposta ' + str(2 + 10 * 4)
'Resposta 42'
```

O modo interativo é uma ferramente muito valioso, pois permite a rápida exploração do funcionamento de trechos de código, mas é importante instruir o aluno a usar esse modo apenas para exploração e não desenvolvimento do código final (a escrita do programa deve ser feita seguindo o processo de projeto de programas).

Não é necessário uma função `main` em Python, mas é uma boa prática criar uma e chamá-la explicitamente (como no exemplo do início dessa seção – documentação).

O Python tem alguns mecanismos para escrita e execução de testes, o mais simples é o `doctest`. Os exemplos escritos na documentação podem ser executados e verificados de forma automática. Por exemplo, se a função `conta_no_intervalo` está em um arquivo chamado `x.py`, os exemplos podem ser verificados com o comando `python -m doctest -v x.py`.

2.3 Sistema de tipos

C é comumente classificada como estaticamente tipada, isto é, os tipos são vinculados as variáveis em tempo de compilação (note que `void*` é um escape para essa regra que é amplamente utilizado pelas funções da biblioteca padrão, entre elas `malloc`/`free`). Os compiladores de C detectam muitos erros de tipo durante a compilação, no entanto, C permite a conversão implícita entre muitos tipos de dados, o que diminui a confiabilidade e pode ser confuso para o aluno iniciante (por exemplo, `double` e `float` para `int`, arranjos para ponteiros, etc).

Historicamente Python tem sido classificada como dinamicamente tipada, isto é, os tipos são vinculados em tempo de execução aos valores e não as variáveis. Isso implica, por exemplo, que uma mesma variável pode em momentos distintos armazenar valores de tipos distintos. Python tem algumas conversões implícitas de valores (como `int` para `float`), mas menos do que C, por esse e outro motivos, o Python é considerado mais fortemente tipada do que C.

Atualmente o Python suporta a especificação de tipos tanto de variáveis como funções. Essa característica foi adicionada inicialmente no Python 3.5 (lançado em 2015) e têm sido aprimorado a cada versão (o Python 2 não suporta especificação de tipos).

O interpretador padrão do Python ignora todas as anotações de tipos, mas existem ferramentes que podem utilizar essas anotações para fazer verificações estáticas no código. Uma dessas ferramentes é o mypy. O `mypy` faz uma verificação estática dos tipos e indica os erros se forem encontrados. Por exemplo, no código a seguir, declaramos `n` como inteiro mas estamos atribuindo um ponto flutuante, o quê o `mypy` identifica como erro.

```
n: int = 10.2
```

Dessa forma, usando o `mypy`, podemos desenvolver os programas em Python como se eles fossem estaticamente tipados, mesmo que de fato as anotações de tipo não alterem a forma como o programa é executado. O sistema de tipos do `mypy` é muito poderoso e permite expressar muitas restrições estaticamente. A seção 4 apresenta mais informações sobres essas questões.

Outro aspecto diferente entre as duas linguagem é que em C todos os tipos são "tipos valores", enquanto que em Python os tipos são "tipos referências".

O principal efeito dessa diferença é visto na atribuição e na passagem de parâmetros. Discutimos essa questão na próxima seção.

2.4 Atribuição e passagem de parâmetros

Como em C "tudo é um valor", todas as atribuições e passagem de parâmetros são feitas por cópia (não tem referência!). No entanto, o uso de ponteiros permite contornar essa limitação, pois a cópia de um ponteiro em uma atribuição ou passagem de parâmetro acaba tendo o mesmo efeito de referências múltiplas para o mesmo valor.

Já em Python como "tudo é uma referência", todas as atribuições e passagem de parâmetros são feitas por referência. Os operadores `is` e `is not` são usados para verificar se duas referências são iguais (referenciam o mesmo objeto). O exemplo a seguir mostra como as atribuições de referências funcionam e a diferença entre igualdade de referências e igualdade de valores.

```
>>> x = [1, 4]
>>> # y passa a referenciar o mesmo valor referenciado por x
>>> y = x
>>> # is produz True se as referências referenciam o mesmo valor
>>> x is y
True
>>> # == produz True se os dois valores referenciados são iguais
>>> x == y
True

>>> # Alteração da lista referenciada por x, que também é referenciada por y
>>> x.append(6)
>>> x
[1, 4, 6]
>>> y
[1, 4, 6]
>>> x is y
True
>>> x == y
True

>>> # x passa a referenciar uma nova lista, y não é alterado
>>> x = [1, 4, 6]
>>> y
[1, 4, 6]
>>> # Os valores referenciados por x e y são iguais
>>> x == y
True
>>> # As referências não são iguais
>>> x is y
False
>>> x is not y
True
```

Os tipos em Python podem ser mutáveis ou imutáveis. Os tipos numéricos, strings, tuplas, entre outros, são imutáveis, ou seja, dado uma referência para um número, não é possível alterar esse número. Já as listas, conjuntos, dicionários e outros tipos são mutáveis. Uma consequência disso é que se um valor de tipo imutável é passado como parâmetro, não é possível devolver um resultado alterando esse valor (pois o tipo é imutável). Outra consequência é que para passar parâmetros de tipos mutáveis por cópia, é preciso fazer a cópia explicitamente.

Considere por exemplo a implementação de uma função de busca em um TAD de dicionário (string para inteiro).

Os argumentos de entrada seriam o dicionário e a chave e a saída uma indicação se existe um elemento associado com a chave e qual é o elemento. Em C poderíamos definir uma função que devolve `bool` indicando o resultado da busca e utilizar um argumento do tipo ponteiro para inteiro para indicar o elemento encontrado. A assinatura da função seria algo como

```c
/* Devolve true se chave está presente em dic, false caso contrário.
 * Se a chave estiver presente, o valor associado com a chave é
 * armazenado em valor.
 */
bool busca_chave(Dicionario *dic, char* chave, int* valor);
```

Em Python não é possível fazer dessa forma. Considere uma função com a assinatura

```python
def busca_chave(dic: Dicionario, chave: str, valor: int) -> bool
```

Se um inteiro é atribuído para `valor` no corpo de `busca_chave`, `valor` passa a referenciar um novo valor, o valor referenciado anteriormente permanece inalterado (números são imutáveis). Discutimos uma forma de resolver essa situação na seção 3.3.3.

2.5 Gerência de memória

Os objetos em C podem ser alocados na pilha ou no heap. A alocação/desalocação dos objetos na pilha é feita automaticamente pelo ambiente de execução. Já a alocação/desalocação de objetos no heap deve ser feita de forma explícita pelo programador.

Em Python todos os objetos são alocados implicitamente no heap. A desalocação é feita de forma automática usando uma combinação de contagem de referências e coleta de lixo. Dessa forma, dois dos erros mais comuns em C, a tentativa de desalocação de memória já desalocada e a tentativa de uso de memória já desalocada, não podem acontecer em Python. (Um outro erro comum em C, acesso de arranjo fora do intervalo válido, é detectado e reportado com uma exceção em Python, conforme discutido na seção 3.1.4).

2.6 Biblioteca padrão

A biblioteca padrão de C é pequena quando comparada com outras linguagens. Ela inclui diversas funções para comunicação com o sistema operacional, alguns funções para strings e números e poucas funções algorítmicas e de estruturas de dados.

Por outro lado, o Python tem uma biblioteca padrão bastante extensa, que é um dos motivos pelos quais o Python é bastante utilizado. No entanto, também pode ser motivo de preocupação para alguns professores, pois existe a possibilidade dos alunos usarem as funcionalidade prontas e deixarem de implementar coisas que são importante para o aprendizado. Mas isso não precisa ser dessa forma, o professor pode deixar claro o que pode ou não ser usado e o que é importante implementar.

Nas próximas duas seção mostramos um subconjunto do Python e como esse subconjunto pode ser usado na implementação de alguns algoritmos e estruturas de dados.

3 O básico de Python

O Python é uma linguagem extensa e tem uma biblioteca padrão extensa. O sistema de tipos do `mypy` é bastante poderoso e também extenso. Dessa forma, para evitar que os alunos se percam na linguagem e deixem de focar nos fundamentos, é importante delimitar um subconjunto das construções da linguagem para ser utilizado pelos alunos.

Apresentamos a seguir um subconjunto suficiente para escrever qualquer algoritmo e estruturas de dados. Note que o código escrito nesse subconjunto pode ficar um pouco mais extenso e não ser considerado idiomático (pythônico) pela comunidade.

3.1 Tipos e operações pré-definidas

Nesta seção apresentamos alguns dos principais tipos pré-definidos em Python e algumas operações sobre esses tipos.

3.1.1 Números

Os principais tipos numéricos em Python são `int` e `float`. O tipo `int` representa inteiros de tamanho arbitrário enquanto `float` números de pontos flutuantes no padrão IEEE 754 binary64 (mesmo que o `double` em C).

Além das quatro operações básicas, outras operações comuns em Python com números são a de módulo (%), exponenciação (**) e piso da divisão (//). O operador / sempre gera um `float` como resposta, mesmo que os operandos sejam inteiros. Outras operações matemáticas estão disponíveis no módulo `math`.

Em operações aritméticas que envolvem inteiros e floats, os inteiros são convertidos para floats antes da execução das operações.

A seguir mostramos alguns exemplos com valores numéricos.

```
>>> # Soma
>>> 4 + 2
6
>>> 4 + 2.0
6.0

>>> # Divisão
>>> 7 / 2
3.5

>>> # Piso da divisão
>>> 14 // 3
4
>>> 5 // 1.3
3.0

>>> # Módulo
>>> 14 % 3
2
>>> 5 % 1.3
1.0999999999999999
>>> -7 % 5
3

>>> # Exponenciação
>>> 2 ** 80
1208925819614629174706176

>>> # Arredondamento, piso e teto
>>> round(3.4)
3
>>> round(3.5)
4
>>> round(3.5134, 2)
3.51
>>> import math
>>> math.floor(4.2)
4
>>> math.ceil(4.2)
5

>>> # Conversão entre int e float
>>> int(7.6)
7
>>> float(4)
4.0

>>> # Conversão entre números e str
>>> str(10)
```

```
'10'
>>> int('123')
123
>>> str(3.2)
'3.2'
>>> float('5.6')
5.6
```

3.1.2 Booleano

Os booleanos são representados pelo tipo `bool`, e podem assumir os valores `True` ou `False`. As operações com booleanos são `not`, `and` e `or`.

```
>>> not 3 > 5
True
>>> 3 > 5 or 3 < 5
True
>>> # and tem prioridade sobre or
>>> True or False and False
True
```

3.1.3 String

As strings são representadas pelo tipo `str` (armazenadas em utf-8). As strings são imutáveis em Python. Algumas operações comuns com strings são mostradas a seguir.

```
>>> # Concatenação
>>> 'Idade: ' + str(32)
'Idade 32'

>>> # Repetição
>>> 'abc' * 3
'abcabcabc'
>>> 'nada' * -1
''

>>> # Remoção de espaços da direita e esquerda
>>> ' alguns espaços  '.strip()
'alguns espaços'
>>> # Versão com chamada na forma de função ao invés de método
>>> str.strip(' alguns espaços  ')
'alguns espaços'

>>> # Substring
>>> din = 'departamento de informatica'
>>> din[13:15] # inicio:fim, inicio está incluido e fim não está
'de'

>>> # Quantidade de caracteres (code points)
>>> len('teste')
5

>>> # Verificação se uma string é substring de outra
>>> 'este' in 'um teste simples'
True
```

O Python não tem um tipo específico para representar um caractere, strings com um *code point* são usadas com esse propósito.

3.1.4 Lista

O tipo mais comum para sequência de valores é o tipo `list`. O tipo `list` representa arranjos dinâmicos (os elementos são contíguos). Python não tem um tipo para arranjos de tamanho fixo. Os elementos de `list` são

indexados a partir de 0 e o acesso é checado, se o índice estiver fora da faixa, uma exceção é lançada. Algumas operações com listas são mostradas a seguir.

```
>>> # Inicialização com alguns elementos
>>> lst = [3, 2, 4]

>>> # Acesso pelo índice
>>> lst[0]
3
>>> lst[2]
4
>>> lst[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> # Quantidade de elementos - tempo O(1)
>>> len(lst)
3

>>> # Verificação de pertinência
>>> 2 in lst
True
>>> 7 in lst
False
>>> 7 not in lst
True

>>> # Inicialização sem nenhum elemento
>>> lst = [] # ou list()

>>> # Adição de elemento no final - tempo amortizado de O(1)
>>> lst.append(3)
>>> lst.append(1)
>>> lst
[3, 1]

>>> # Concatenação gerando uma nova lista
>>> lst + [4, 0, 1]
[3, 1, 4, 0, 1]

>>> # Concatenação alterando a lista
>>> lst.extend([4, 0, 1])
>>> lst
[3, 1, 4, 0, 1]

>>> # Remoção do final - tempo O(1)
>>> lst.pop()
1
>>> lst
[3, 1, 4, 0]

>>> # Criação de cópia - tempo O(n)
>>> lst.copy()
[3, 1, 4, 0]

>>> # Sublista
>>> lst[1:3]
[1, 4]

>>> # Remoção de todos os elementos - tempo O(n)
```

```
>>> lst.clear()
>>> lst
[]
```

Repetição de elementos

Assim como o tipo `str`, o tipo `list` também tem operação de repetição de elementos. A operação de repetição é geralmente utilizada para inicializar uma lista com uma quantidade pré-definida de elementos, conforme os exemplos abaixo

```
>>> # Inicialização com 5 zeros
>>> lst = [0] * 5
>>> lst
[0, 0, 0, 0, 0]

>>> # Modificação da lista
>>> lst[0] = 10
>>> lst[4] = 2
>>> lst
[10, 0, 0, 0, 2]
```

Apesar de útil, é preciso cuidado para usar essa operação. A questão é que, conforme discutido nas seções 2.3 e 2.4, todos os tipos em Python são tipos referências, dessa forma, quando os elementos de uma lista são repetidos, o que de fato está sendo repetido (copiado) são as referências para os elementos. Para tipos imutáveis, como no exemplo anterior, isso não é um problema, mas para tipos mutáveis, isso pode gerar resultados inesperados.

A seguir apresentamos uma tentativa de inicializar uma lista de listas (arranjo bidimensional 3 x 3). Usamos a variável `linha` para ficar mais claro o ponto que vamos destacar, mas não é necessário, poderíamos escrever `m = [[0, 0, 0]] * 3` e a questão seria a mesmo.

```
>>> linha = [0, 0, 0]
>>> m = [linha] * 3
>>> m
[[0, 0, 0], [0, 0, 0], [0, 0,0]]
```

Por enquanto, parece que está tudo certo. No entanto, se notarmos que `linha`, `m[0]`, `m[1]` e `m[2]` referenciam a mesma lista (objeto), então sabemos que uma mudança feita nessa lista por qualquer dessas variáveis, será refletida em todas as outras!

```
>>> m[0][0] = 3
>>> m
[[3, 0, 0], [3, 0, 0], [3, 0, 0]]
```

Como proceder nesse caso? Existem formas compactas para fazer isso em Python, mas não são discutidas nesse texto. Usando apenas as construções mais básicas podemos fazer

```
m = []
for i in range(3):
    linha = []
    for j in range(3):
        linha.append(0)
    m.append(linha)
```

Mistura de tipos

Nós vimos na seção 2.3 que Python é geralmente utilizada como uma linguagem com vinculação dinâmica de tipo, isto é, os tipos são associados aos valores, não as variáveis. Uma consequência desse fato é que os valores em uma lista podem ter tipos distintos, ou mesmo mudarem de tipo, como no exemplo a seguir:

```
>>> # Mistura de valores com tipos diferentes
>>> lst = [10, 'casa', False]

>>> # Alteração do tipo do valor armazenado em 0
>>> lst[0] = [1, 2]
```

```
>>> lst
[[1, 2], 'casa', False]
```

Embora este tipo de construção seja permitida, ela pode ser confusa se usada de forma arbitrária, por isso é importante estabelecer um fundamento sobre a utilidade desse tipo de construção. Discutimos essa questão mais a fundo na seção 3.3.3, por hora, podemos usar anotações de tipos e o programa `mypy` para identificar como erro esse tipo de construção. Por exemplo, se quiséssemos restringir os tipos de `lst` para inteiros, escreveríamos:

```
lst: list[int] = [10, 'casa', False]
```

Nesse caso, o seguinte erro seria gerado pelo `mypy`:

```
x.py:1: error: List item 1 has incompatible type "str"; expected "int"  [list-item]
Found 1 error in 1 file (checked 1 source file)
```

Note que só é possível usar o `mypy` para processar arquivos `.py`, não é possível utilizar o `mypy` no modo interativo. É importante lembrar também que o interpretador padrão do Python ignora as anotações de tipo, então, mesmo que o `mypy` indique erro, o código é executado pelo interpretador (gerando erro apenas quando uma operação é invocada com tipos inválidos).

3.2 Estruturas de controle

Nessa seção apresentamos as principais estruturas de controle do Python.

3.2.1 Funções

A funções em Python são definidas com a palavra chave `def`. A palavra chave `return` é usada para indicar o valor produzido pela função e pode ser usada mais que uma vez no corpo da função. Funções que não produzem valores explicitamente (não usam `return`) produzem como saída o valor `None`. O exemplo a seguir mostra a definição de uma função:

```
def par(n: int) -> bool:
    '''
    Produz True se *n* é par, isto é, *n* é múltiplo de 2. False caso contrário.

    Exemplos
    >>> par(10)
    True
    >>> par(3)
    False
    >>> par(0)
    True
    '''
    return n % 2 == 0
```

3.2.2 Seleção

A principal estrutura de seleção do Python é o `if`, (o Python 3.10 introduziu a construção `match/case`, que é de certa forma similar ao `switch/case` do C, mas mais geral).

Assim como no `if` do C, a cláusula `else` do `if` do Python é opcional. Para evitar o aumento de níveis na indentação de `if`s aninhamos, o Python permite a união de um `else` seguido de `if` com a palavra chave `elif`. O exemplo a seguir mostra o uso do `if`.

```
def sinal(n: int) -> int:
    '''
    Devolve o sinal de *n*, isto é, 1 se n > 0, -1 se n < 1, 0 se n == 0.

    Exemplos
    >>> sinal(3)
    1
    >>> sinal(0)
    0
    >>> sinal(-4)
    -1
```

```
    '''
    if n > 0:
        return 1
    else:
        if n < 0:
            return -1
        else:
            return 0
    # Ou usando o elif
    # if n > 0:
    #     return 1
    # elif n < 0:
    #     return -1
    # else:
    #     return 0
```

3.2.3 Repetição

O `for` clássico do C não existe em Python. Em Python o `for` é usado para fazer iteração pelos elementos de uma estrutura de dados. A outra forma de repetição do Python é o `while`. O exemplo a seguir mostra o uso do `for` e do `while`.

```
def ordena(nomes: list[str]) -> list[str]:
    '''
    Cria uma nova lista com os mesmos elementos de *nomes* mas em ordem alfabética.

    Exemplo
    >>> ordena(['Paulo', 'Ana', 'Maria', 'João'])
    ['Ana', 'João', 'Maria', 'Paulo']
    '''
    r = []
    for nome in nomes:
        r.append(nome)
        i = len(r) - 1
        # r[i] é o nome que foi inserido
        # troca r[i] com r[i - 1] até que ele fique na posição adequada
        while i > 0 and r[i - 1] > r[i]:
            ri = r[i]
            r[i] = r[i - 1]
            r[i - 1] = ri

            i = i - 1
    return r
```

Quando for necessário o índice dos elementos em uma iteração, podemos usar o `while` ou o `for` combinado com a função `range`.

```
def indice_maximo(valores: list[float]) -> int:
    '''
    Encontra o índice do valor máximo em *valores*.

    Caso o valor máximo ocorra em mais de um índice,
    o menor índice é devolvido.

    Requer que valores não seja vazio.

    Exemplos
    >>> indice_maximo([2, 1, 4, -2])
    2
    >>> indice_maximo([-1, -4, -1])
    0
    '''
```

```
    assert len(valores) != 0, "valores não pode ser vazio"
    imax = 0
    for i in range(1, len(valores)):
        if valores[imax] < valores[i]:
            imax = i
    return imax
```

3.2.4 Asserção

No exemplo anterior usamos uma outra estrutura de controle, o **assert**. O **assert** pode ser usado com uma ou duas expressões. O Python avalia a primeira expressão, se o resultador for `True`, a execução continua para a próxima linha, caso contrário, uma exceção é gerada com uma mensagem padrão ou com o resultado da segunda expressão do **assert** (se existir).

3.3 Definição de tipos de dados

Python é uma linguagem multiparadigma, mas a forma de criação de tipos é através de classes. De fato, quase tudo em Python é um objeto, com atributos e métodos. Por exemplo, valores inteiros são objetos e podemos invocar métodos com eles:

```
>>> # Invoca explicitamente o método __add__ (operação +)
>>> x = 1
>>> x.__add__(2) # ou (1).__add__(2)
3
```

Antes de vermos a forma geral de definição de classes, vamos ver formas simplificadas que são similares a definição de estruturas e enumerações em C.

3.3.1 Estruturas

Uma forma simplificada de declarar uma classe é usando o decorador `@dataclass`. Um decorador é um mecanismo de meta-programação utilizado para modificar classes, métodos e atributos. Embora o funcionamento dos decoradores possa ser bastante elaborado, o seu uso em geral é direto e simples.

Utilizamos o decorador `@dataclass` quando queremos criar uma classe que se comporte de forma semelhante a uma **struct** em C. No exemplo a seguir criamos uma classe ponto e mostramos algumas operações:

```
from dataclasses import dataclass

@dataclass
class Ponto:
    x: int
    y: int

>>> # Construtor
>>> p = Ponto(10, 20)
>>> p.x
10
>>> p.y
20
>>> p.x = p.x + 1
>>> p
Ponto(x=11, y=20)
>>> # Comparação
>>> p == Ponto(11, 20)
True
```

Observe que uma classe criada com `@dataclass` tem construtor (que recebe todos os campos como parâmetro na ordem que eles foram definidos), operador de igualdade (que compara os valores de cada campo), entre outras operações.

O exemplo a seguir mostra a definição de uma classe e o seu uso em uma função (note a forma dos exemplos):

```
from dataclasses import dataclass
```

```
@dataclass
class Desempenho:
    '''O desempenho de um time em um campeonato de futebol'''
    # O nome do time
    time: str
    # Pontos (3 por vitória e 1 por empate)
    pontos: int
    # Saldo de gols (gols pro - gols contra)
    saldo: int

def atualiza_desempenho(d: Desempenho, gols_pro: int, gols_contra: int):
    '''
    Atualiza o desempenho *d* considerando *gols_pro* e *gols_contra*.

    O saldo de gols é atualizado somando *gols_pro* e subtraindo *gols_contra*.

    Se *gols_pro > gols_contra*, o número de pontos aumenta em 3.

    Se *gols_pro == gols_contra*, o número de pontos aumenta em 1.

    Exemplos
    >>> d = Desempenho('Maringá', 0, 0)
    >>> # Vitória
    >>> atualiza_desempenho(d, 5, 1)
    >>> d
    Desempenho(time='Maringá', pontos=3, saldo=4)
    >>> # Empate
    >>> atualiza_desempenho(d, 2, 2)
    >>> d
    Desempenho(time='Maringá', pontos=4, saldo=4)
    >>> # Derrota
    >>> atualiza_desempenho(d, 0, 3)
    >>> d
    Desempenho(time='Maringá', pontos=4, saldo=1)
    '''
    d.saldo = d.saldo + gols_pro - gols_contra

    if gols_pro > gols_contra:
        d.pontos = d.pontos + 3
    elif gols_pro == gols_contra:
        d.pontos = d.pontos + 1
```

3.3.2 Enumerações

Outra forma simplificada para criar classe é usando o Enum. Usamos essa forma quando queremos criar um tipo enumerado, isto é, quando os valores permitidos para o tipo podem ser enumerados explicitamente. Classes criadas com esse mecanismos são semelhantes a tipos criados com **enum** em C. No exemplo a seguir, mostramos a declaração e uso de uma classe para um tipo enumerado (observe que, por convenção, escrevemos os valores enumerados em maiúsculas).

```
from enum import Enum, auto

class Cor(Enum):
    '''
    Representa a cor de um semáforo.
    '''
    VERMELHO = auto()
    AMARELO = auto()
    VERDE = auto()

def proxima_cor(c: Cor) -> Cor:
```

```
    '''
    Determina a próxima cor de um semáforo que está na cor *c*.

    Exemplos
    >>> proxima_cor(Cor.VERMELHO).name
    'VERDE'
    '''
    if c == Cor.VERMELHO:
        return Cor.VERDE
    elif c == Cor.AMARELO:
        return Cor.VERMELHO
    elif c == Cor.VERDE:
        return Cor.AMARELO
```

Além do campo `name`, cada valor da enumeração também tem um campo `value`, que é um valor inteiro gerado automaticamente (podemos indicar os valores de cada item da enumeração diretamente ao invés de usar o `auto` na declaração).

O `mypy` é bastante robusto na verificação do uso de enumerações, especialmente se todos os valores estão sendo considerados. Use o verificador online do `mypy` e teste o código anterior removendo algum dos casos e observe o erro gerado.

3.3.3 Uniões

As uniões em Python são discriminadas (contém um rótulo que diz o tipo armazenado) e são usadas em diversas situações. O operador `|` (a partir do Python 3.10) é utilizado para especificar a união de tipos e a função `isinstance` é utilizada para verificar o tipo (rótulo) de um valor. O exemplo a seguir mostra o uso de uniões (\ é usado para permitir que a expressão continue na próxima linha).

```
from dataclasses import dataclass

@dataclass
class Envelope:
    comprimento: int
    largura: int

@dataclass
class Caixa:
    comprimento: int
    largura: int
    altura: int

@dataclass
class Rolo:
    comprimento: int
    diametro: int

'''Uma embalagem para envio nos correios.'''
Embalagem = Envelope | Caixa | Rolo

def dentro_limites(emb: Embalagem) -> bool:
    '''
    Produz True se *emb* está dentro dos limites, False caso contrário.

    Uma embalagem está dentro dos limites se suas medidas obedecem as seguintes
    restrições (em cm).

                        mínimo  máximo
    Envelope
        comprimento     16      60
        largura         11      60
    Caixa
```

```
        comprimento     15     100
        largura         10     100
        altura           1     100
    Rolo
        comprimento     18     100
        diametro         5      21

    Exemplos (alguns omitidos para diminuir o espaço)
    >>> dentro_limites(Envelope(20, 40))
    True
    >>> dentro_limites(Envelope(15, 40))
    False
    >>> dentro_limites(Caixa(30, 40, 80))
    True
    >>> dentro_limites(Caixa(30, 101, 80))
    False
    >>> dentro_limites(Rolo(50, 18))
    True
    >>> dentro_limites(Rolo(50, 4))
    False
    '''
    if isinstance(emb, Envelope):
        return 16 <= emb.comprimento <= 60 and \
               11 <= emb.largura <= 60
    elif isinstance(emb, Caixa):
        return 15 <= emb.comprimento <= 100 and \
               10 <= emb.largura <= 100 and \
                1 <= emb.altura <= 100
    elif isinstance(emb, Rolo):
        return 18 <= emb.comprimento <= 100 and \
                5 <= emb.diametro <= 21
```

Um uso bastante comum de uniões é na representação de ausência de valores. Por exemplo, um nó em uma lista pode ou não ter um próximo elemento, um nó em uma árvore binária pode ou não ter filhos a direita e a esquerda, uma busca em um dicionário pode ou não encontrar um valor associado com uma chave (veja a seção 2.4), etc. Para ilustrar de forma breve esse uso, mostramos a seguir a assinatura de uma função que busca o valor associado com uma chave em um dicionário, exemplos completos são apresentados na seção 4.

```
def busca_chave(dic: Dicionario, chave: str) -> None | int:
    '''Devolve o valor associado com chave em dic, ou None se chave não estiver em dic.'''
```

Assim como para enumerações, o `mypy` também é bastante robusto no tratamento de uniões devido a análise de refinamento de tipos. No exemplo da embalagem, o `mypy` só permite que o campo `diametro` seja acessado através da variável `emb` se ele puder provar que `emb` é uma instância de `Rolo`, que é o caso no exemplo devido ao `if`. Use o verificador online do `mypy` e tente construções inválidas que só seriam identificas pelo Python em tempo de execução e veja que o `mypy` identifica essas construções de forma estática.

O refinamento de tipos é bastante útil, e nos permite, por exemplo, garantir que não haverá acesso a referências nulas em tempo de execução (como mostrado na seção 4)

3.3.4 Classes

A forma geral para definição de tipos em Python é a construção `class`. A seguir mostramos a declaração de uma classe e um método e alguns exemplos de uso:

```
class Ponto:
    x: int
    y: int

    # Construtores que recebem parâmetros precisam ser criados explicitamente
    # O parâmetro self é a instância do ponto que está sendo inicializada
    def __init__(self, x: int, y: int):
        self.x = x
```

```
        self.y = y

    def distancia_origem(self) -> float: # self é uma referência para uma instância
        '''
        Calcula a distância do ponto a origem.
        Exemplos
        >>> p = Ponto(3, 4)
        >>> p.distancia_origem()
        5.0
        '''
        return (x ** 2 + y ** 2) ** 0.5
```

```
>>> p = Ponto(10, 20)
>>> p.x
10
>>> p.y
20
>>> p.x = p.x + 1
>>> # Por padrão, a representação do objeto é opaca
>>> p
<__main__.Ponto object at 0x7fad1fac8530>
>>> # Por padrão, a comparação é por referência e não pelo conteúdo
>>> p == Ponto(11, 20)
False
>>> p == p
True
```

Quando devemos usar `@dataclass` e classes "normais"? Usamos `@dataclass` quando queremos apenas agrupar dados e o encapsulamento não é importante. Classes "normais" devem ser usadas nas demais situações, como por exemplo, para criar tipos abstratos de dados (veja a seção 4.3).

3.4 Entrada e saída

A principal função de entrada em Python é a função `input`, que recebe um argumento (mensagem a ser exibida) e retorna uma linha lida (string) da entrada padrão. O Python não tem funções para fazer entrada de dados formatada.

O Python tem diversas funções de saída (incluindo saída formatada), a mais comum é a função `print`. A função `print` recebe um número variado de argumentos (de qualquer tipo), converte cada argumento para uma string com a função `str` e exibe os valores separando-os por espaço e adicionando final de linha (esse comportamento pode ser alterado).

O exemplo a seguir mostra como ler um nome e a data de nascimento de uma pessoa:

```
nome = input('Qual é o seu nome?')
data = input('Qual é a sua data de nascimento?')
# Fazemos um processamento para obter o ano.
# Omitimos o tratamento de erro.
ano = int(data.split('/')[2])
print(nome + ',', 'em 2050 você terá', 2050 - ano, 'anos.')
```

Execução

```
Qual é o seu nome? João
Qual é a sua data de nascimento? 01/02/2003
João, em 2050 você terá 47 anos.
```

4 Exemplos

Nesta seção mostramos alguns exemplos de algoritmos e estruturas de dados implementados em Python.

4.1 Ordenação por intercalação

O exemplo a seguir mostra a implementação do algoritmo de ordenação por intercalação (merge sort). Note que o código poderia ser simplificado utilizando sublistas (*slices*), mas a intenção do exemplo é mostrar a implementação usando apenas as construções que são comuns na maioria das linguagens.

```
def ordena_intercalacao(valores: list[int], inicio: int, fim: int):
    '''
    Ordena o subarranjo *valores[inicio:fim]* em ordem não decrescente.

    A ordenação é feita usando o algoritmo de intercalação.

    Requer que 0 <= inicio <= fim <= len(valores).

    Exemplos
    >>> valores = [3, 1, 6, 1, 2, 5, 3]
    >>> ordena_intercalacao(valores, 0, len(valores))
    >>> valores
    [1, 1, 2, 3, 3, 5, 6]
    '''
    assert 0 <= inicio <= fim <= len(valores)
    if fim > inicio + 1:
        m = (inicio + fim) // 2
        ordena_intercalacao(valores, inicio, m)
        ordena_intercalacao(valores, m, fim)
        intercala(valores, inicio, m, fim)

def intercala(valores: list[int], inicio: int, m: int, fim: int):
    '''
    Intercala os elementos dos subarranjos *valores[inicio:m]* e *valores[m:fim]*.

    Requer que 0 <= inicio < m < fim <= len(valores) e que os subarranjos
    valores[inicio:m] e valores[m:fim] estajam ordenados.

    Exemplo
    >>> valores = [1, 5, 2, 4, 1, 3, 5, 4, 6]
    >>> intercala(valores, 2, 4, 7)
    >>> valores
    [1, 5, 1, 2, 3, 4, 5, 4, 6]
    '''
    assert 0 <= inicio < m < fim <= len(valores)

    # Copia os elementos a direita de m para dir
    dir = []
    for i in range(inicio, m):
        dir.append(valores[i])

    # Copia os elementos a partir de m e a esquerda para esq
    esq = []
    for i in range(m, fim):
        esq.append(valores[i])

    i = 0
    j = 0
    k = inicio
    # Faz a intercação enquanto houver elementos em dir e esq
    while i < len(dir) and j < len(esq):
        if dir[i] < esq[j]:
            valores[k] = dir[i]
            i += 1
        else:
            valores[k] = esq[j]
```

```
            j += 1
        k += 1

    # Copia o resto de dir se houver
    while i < len(dir):
        valores[k] = dir[i]
        i += 1
        k += 1

    # Copia o resto de esq se houver
    while j < len(esq):
        valores[k] = esq[j]
        j += 1
        k += 1
```

4.2 TAD dicionário implementado com funções

O exemplo a seguir mostra a implementação de um TAD dicionário usando busca linear e funções. Apenas as operações de inserção e busca são mostradas. Note que usamos `@dataclass` para a implementação ficar mais semelhante a que seria feita em C. Na próxima seção mostramos um exemplo que utiliza classe "normal" para enfatizar o encapsulamento.

```
from dataclasses import dataclass

@dataclass
class Item:
    '''Um par (chave, valor) em Dicionario.'''
    chave: str
    valor: int

@dataclass
class Dicionario:
    '''
    Um dicionario que associa strings com inteiros.

    Este dicionário mantém uma lista de pares (chave, valor) em uma lista. As
    adições são feitas no final da lista e as pesquisas são feitas de forma
    linear na lista.

    Este dicionário não suporta a remoção de items.
    '''
    items: list[Item]

def cria_dicionario() -> Dicionario:
    '''
    Cria um novo dicionário
    '''
    return Dicionario([])

def adiciona(dic: Dicionario, chave: str, valor: int) -> bool:
    '''
    Associa *chave* com *valor* em *dic* se ainda não existe valor associado
    com *chave*. Produz True se a associação foi criada, False caso contrário.

    Exemplos
    >>> dic = cria_dicionario()
    >>> adiciona(dic, 'Pedro', 24)
    True
    >>> adiciona(dic, 'Ana', 21)
    True
    >>> adiciona(dic, 'Pedro', 40)
```

```
    False
    '''
    if busca_chave(dic, chave) is None:
        dic.items.append(Item(chave, valor))
        return True
    else:
        return False

def busca_chave(dic: Dicionario, chave: str) -> None | int:
    '''
    Devolve o valor associado com *chave* em *dic*, ou None se *chave* não
    estiver em *dic*.

    Exemplos
    >>> dic = cria_dicionario()
    >>> adiciona(dic, 'verde', 10)
    True
    >>> adiciona(dic, 'amarelo', 20)
    True
    >>> busca_chave(dic, 'verde')
    10
    >>> busca_chave(dic, 'azul') is None
    True
    '''
    for item in dic.items:
        if item.chave == chave:
            return item.valor
    return None
```

4.3 TAD lista ordenada implementado com classes

Nos exemplos a seguir mostramos a implementação de um TAD de lista ordenada usando encadeamento. No primeiro exemplo os nós são alocados dinamicamente. No segundo exemplo os nós são alocados quando a lista é criada (simulando alocação estática) e o encadeamento é feitos com índices (cursores). Note que são duas implementações diferentes para o mesmo TAD. Apenas a função de inserção é mostrada.

Implementação com encadeamento ilimitado

```
# Esse import é necessário devido a autorreferência na definição de No
from __future__ import annotations
from dataclasses import dataclass

@dataclass
class No:
    '''Um nó em uma lista encadeada.'''
    # O item armazenado no nó
    item: int
    # O próximo nó
    prox: None | No

class Lista:
    '''
    Uma lista ordenada de números.

    Essa implementação usa encadeamento de nós.
    '''
    primeiro: No | None

    def __init__(self):
        '''
        Cria uma nova lista.
```

```
        Exemplo
        >>> lst = Lista()
        >>> lst.para_list()
        []
        '''
        self.primeiro = None

    def insere_ordenado(self, item: int) -> bool:
        '''
        Insere *item* na lista de maneira que ela permaneça ordenada.

        Sempre produz True indicando que o item foi inserido.

        Exemplos
        >>> lst = Lista()
        >>> lst.insere_ordenado(2)
        True
        >>> lst.insere_ordenado(6)
        True
        >>> lst.insere_ordenado(4)
        True
        >>> lst.insere_ordenado(1)
        True
        >>> lst.para_list()
        [1, 2, 4, 6]
        '''
        # Procura a posição de inserção
        pred = None
        atual = self.primeiro
        while atual is not None and atual.item < item:
            pred = atual
            atual = atual.prox

        # Efetura a inserção
        if pred is None:
            # Não tem predecessor, então insere como primeiro
            self.primeiro = No(item, self.primeiro)
        else:
            # Insere item entre pred e atual
            # antes : pred -> atual
            # depois: pred -> novo -> atual
            pred.prox = No(item, pred.prox)
        return True

    def para_list(self) -> list[int]:
        '''
        Converte a lista para uma list do Python.

        Este método é útil para testar o funcionamento da lista.
        '''
        resultado = []
        atual = self.primeiro
        while atual is not None:
            resultado.append(atual.item)
            atual = atual.prox
        return resultado
```

Implementação com encadeamento de tamanho máximo fixo

```
from dataclasses import dataclass

@dataclass
```

```python
class No:
    '''Um nó em uma lista encadeada.'''
    # O item armazenado no nó
    item: int
    # O índice do próximo nó no encadeamento
    prox: None | int

class Lista:
    '''
    Uma lista ordenada de números.

    A implementação é feita com um encadeamento de tamanho máximo fixo.

    Os nós são alocados quando a lista é criada e a quantidade de nós não é
    alterada. O campo prox de cada nó é utilizado para indicar o próximo nó da
    lista (se o nó está na lista), ou o próximo nó livre (se o nó não está na
    lista). Se o nó está disponível, o campo item não armazena nenhum valor
    válido.
    '''
    # Os nós pré-alocados
    nos: list[No]
    # O índice em nos do primeiro nó da lista ou None se a lista está vazia
    primeiro: None | int
    # O índice em nos do primeiro nó disponível ou None se não existe nenhum
    disponivel: None | int
    # O número de itens na lista
    num_itens: int

    def __init__(self, n: int):
        '''
        Cria uma nova lista com tamanho máximo *n*.

        Requer que n > 0.

        Exemplo
        >>> lst = Lista(3)
        >>> lst.para_list()
        []
        '''
        assert n > 0
        self.nos = []
        self.primeiro = None
        self.disponivel = 0
        self.num_itens = 0
        # Faz o encadeamento dos disponíveis, começando com 0
        for i in range(n):
            self.nos.append(No(0, i + 1))
        # O último nó não tem um próximo
        self.nos[n - 1].prox = None

    def insere_ordenado(self, item: int) -> bool:
        '''
        Insere *item* na lista de maneira que ela permaneça ordenada.

        Produz True se existia espaço na lista e *item* foi inserido, False caso contrário.

        Exemplos
        >>> lst = Lista(4)
        >>> lst.insere_ordenado(2)
        True
        >>> lst.insere_ordenado(6)
```

```
        True
        >>> lst.insere_ordenado(4)
        True
        >>> lst.insere_ordenado(1)
        True
        >>> # Não tem mais espaço
        >>> lst.insere_ordenado(7)
        False
        >>> lst.para_list()
        [1, 2, 4, 6]
        '''
        if self.disponivel is None:
            return False

        # Usa o primeiro nó disponível
        novo = self.disponivel
        self.disponivel = self.nos[novo].prox
        self.nos[novo].item = item

        # Procura a posição de inserção
        pred = None
        atual = self.primeiro
        while atual is not None and self.nos[atual].item < item:
            pred = atual
            atual = self.nos[atual].prox

        # Efetura a inserção
        if pred is None:
            # Não tem predecessor, então insere como primeiro da self
            self.nos[novo].prox = self.primeiro
            self.primeiro = novo
        else:
            # Insere novo entre pred e atual
            # antes : pred -> atual
            # depois: pred -> novo -> atual
            self.nos[pred].prox = novo
            self.nos[novo].prox = atual

        self.num_itens += 1
        return True

    def para_list(self) -> list[int]:
        '''
        Converte a lista para uma list do Python.

        Este método é útil para testar o funcionamento da lista.
        '''
        resultado = []
        atual = self.primeiro
        while atual is not None:
            resultado.append(self.nos[atual].item)
            atual = self.nos[atual].prox
        return resultado
```

5 Convenções de código

Essas são algumas convenções oficiais para escrita de código em Python.

- Indente o código usando 4 espaços, não utilize tabulações;
- Nomeie tipos da forma `CapitalizedWords`;

- Nomeie funções e variáveis da forma `lower_case_with_underscores`;
- Nomeie constantes da forma `UPPER_CASE_WITH_UNDERSCORES`;
- Use um `import` por linha;
- Evite espaçamentos extras (`spam(ham[1], {eggs: 2})` ao invés de `spam( ham[ 1 ], { eggs: 2 } )`);
- Use espaços entre operadores (`c += (a + b) * (a - b)` ao invés de `c+=(a+b)*(a-b)`)

6 Bibliografia

A seguir estão algumas referências bibliográficas que podem ser úteis tanto para os professores quanto para os alunos.

Introdução a programação

- A Data-Centric Introduction to Computing
- How to Design Programs (vídeos)
- SICP in Python (vídeos)
- Think Python 2e (tradução em português)
- Notas de aula de introdução a programação e de estruturas de dados em Python (do mesmo autor desse texto)

Estrutura de dados

- Open data structures (em pseudo código que foi gerado automaticamente a partir do código Python)

Linguagem Python

- Dive into Python 3