11

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

11

Busca e ordenação em sequências

Prof. Me. Lucio Nunes de Lira
Prof. MSc. Rafael Maximo Carreira Ribeiro

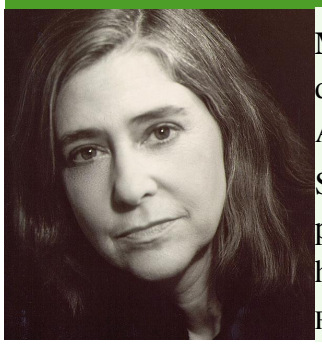
Resumo

Nesta aula os objetivos são: (I) entender como buscar itens em sequências; (II) conhecer e implementar os algoritmos de busca linear e busca binária; (III) construir uma função para ordenação de sequências; (IV) resumir o funcionamento de outros algoritmos de ordenação; (V) conhecer recursos integrados ao Python para ordenação; (VI) aprender como chamar funções com argumentos nomeados e como defini-las com argumentos padrão.

11.1. Motivação

Ao trabalhar com grande quantidade de dados, duas operações costumam ocorrer com frequência: (I) a busca por dados específicos e (II) a ordenação dos dados por algum critério, o que pode, inclusive, influenciar nas buscas. Neste capítulo estudaremos os conceitos envolvidos nesses algoritmos e construiremos programas para essa finalidade. Também veremos mais características e recursos que podem ser adicionados ao uso e criação de funções, são os argumentos nomeados e argumentos padrão.

💎 VOCÊ CONHECE?



Margaret Hamilton, uma cientista da computação, engenheira de software e ex-diretora da Divisão de Software do MIT. Atuou no programa de voo do projeto Apollo 11 da NASA.

Seu *software* permitiu que o pouso na Lua não fosse abortado, priorizando instruções críticas. Pelos seus feitos notáveis, foi homenageada pelo Google: https://youtu.be/B7CnVGtd1_Y

Fonte da imagem: https://computerhistory.org/profile/margaret-hamilton/

11.2. Busca

Ao trabalhar com coleções[1] de dados é comum que buscas sejam feitas para verificar se um item pertence à coleção ou para acessar dados associados ao item buscado, também chamados de dados satélites, caso o item seja encontrado.

Há vários algoritmos de busca, alguns exigem pré-condições para que funcionem, como a *busca binária*, um algoritmo com ótimo desempenho, mas que supõe que os itens estejam ordenados em sequência crescente ou decrescente. Esse algoritmo possibilita encontrar um item dentre 1 bilhão de itens com, no máximo, 31 verificações!

Outros algoritmos funcionam em coleções sem pré-condições, como a *busca linear*. Neste algoritmo, o item buscado será comparado com cada item da coleção até que a sequência termine ou até que seja encontrado. O custo dessa flexibilidade é o baixo desempenho, por não ser possível aplicar estratégias mais sofisticadas.

Para uma abordagem inicial, pense no problema de encontrar uma palavra em um dicionário de papel, ou um nome em uma lista telefônica, se alguém ainda se recorda.

Sabemos que nestas situações os itens estão em ordem alfabética, o que permite abrir o dicionário aproximadamente ao meio e olhar se a palavra buscada está naquela página, ou se ela deveria estar em uma página antecessora ou sucessora. Caso a palavra não esteja na página aberta, podemos descartar essa página, juntamente com as antecessoras (se a palavra estiver alfabeticamente à diante), ou juntamente com as sucessoras (se a palavra estiver alfabeticamente atrás), e buscar a palavra novamente, com o mesmo procedimento, nas páginas que restaram. Assim funciona a busca binária.

Agora, imagine que as palavras estão distribuídas aleatoriamente no dicionário, sem uma ordem conhecida. Neste caso, o procedimento descrito anteriormente não funcionaria, pois seria impossível ter certeza que a palavra buscada está antes ou após aquelas da página corrente. Uma solução seria abrir o dicionário na primeira página e ler todas as palavras sequencialmente até encontrar a palavra buscada. Assim funciona a busca linear, também conhecida como *busca sequencial*.

11.2.1. Busca linear

Vimos que em Python é possível buscar um item, em alguns tipos de sequência, com o método `index`, que recebe como argumento um valor e, caso encontre-o, retorna o índice da primeira ocorrência. Veja a Codificação 11.1 para relembrar como usá-lo.

```
>>> lista = [3, 6, 5, 8, 0, 8, 2]
>>> lista.index(8)
3
```

Codificação 11.1: Utilização do método `index`.

[1] Coleção é uma estrutura que permite armazenar itens. É um termo mais genérico do que "sequência".

Para compreender o algoritmo da busca linear, criaremos uma função semelhante ao método `index`. A função deverá receber um valor e uma sequência e retornar o índice da primeira ocorrência do valor na sequência, se encontrado. Veja uma solução válida na Codificação 11.2 e tente elaborar outra versão da mesma função.

```
def busca_linear(valor, sequencia):
    for i, item in enumerate(sequencia):
        if item == valor:
            return i
    return None
```

Codificação 11.2: Exemplo de implementação da busca linear.

Na Codificação 11.2, usamos a função integrada `enumerate` para gerar uma nova sequência a partir da sequência argumento. Cada item da nova sequência é uma tupla de dois valores, simplificadamente: o 1º é um índice e o 2º é o item da sequência argumento com aquele índice. Assim, podemos usar o desempacotamento de sequências no laço `for` para, a cada iteração, ter acesso tanto ao índice quanto ao valor do item, sem precisar gerenciar a contagem dos índices manualmente com uma variável contadora.

Ao encontrar um item igual ao valor procurado, a função é encerrada com o retorno do índice do item na sequência, que corresponde à variável `i`. Caso a sequência seja completamente percorrida e nenhum item corresponda ao valor procurado, será retornado o valor `None`. Essa foi uma decisão do programador, pois não está estipulada na especificação do algoritmo, porém poderíamos seguir o comportamento do método `index` e levantar um erro de execução, mas isso não está no escopo desta disciplina.

VAMOS PRATICAR!

1) Crie uma função semelhante àquela da Codificação 11.2, porém que retorne `-1` como resposta para quando não encontrar o valor buscado. Esta versão mantém a coerência do tipo do valor retornado em todos os casos, imprescindível em diversas linguagens.

2) Crie uma função semelhante àquela da Codificação 11.2, porém que o valor de retorno seja uma lista com os índices de todos os itens iguais ao valor buscado.

3) Crie uma função semelhante à do exercício anterior, porém que a busca parta do último item e avance até o primeiro.

11.2.2. Busca binária

A implementação da busca binária é mais elaborada que a da busca linear, mas quando sabemos que a lista está ordenada, este método proporciona desempenho muito superior, que fica mais evidente conforme a busca é realizada em sequências maiores.

Por exemplo, para encontrar um item em uma sequência de tamanho 1 bilhão, a busca binária precisa de, no máximo 31, comparações. Em uma sequência com 2 bilhões de itens, no máximo 32. Isso mesmo! Dobrando o tamanho da sequência aumenta-se apenas uma verificação, pois a cada comparação, o intervalo de busca é reduzido pela metade. Veja na Codificação 11.3 uma implementação da função de busca binária.

```
def busca_binaria(valor, sequencia):
    inicio = 0
    fim = len(sequencia) - 1

    while inicio <= fim:
        meio = (inicio + fim) // 2

        if valor == sequencia[meio]:
            return meio
        elif valor < sequencia[meio]:
            fim = meio - 1
        else:
            inicio = meio + 1

    return None
```

Codificação 11.3: Exemplo de implementação da busca binária.

Você pode adicionar uma variável contadora e algumas saídas para compreender melhor cada passo da Codificação 11.3, que se baseia no seguinte algoritmo:

1) A função recebe um `valor` que deve ser buscado na `sequencia`;
2) Atribuímos a `inicio` o índice do primeiro item da sequência, que é zero;
3) Atribuímos a `fim` o índice do último item da sequência, dado pelo tamanho da sequência - 1 (não usaremos índices negativos);
4) As variáveis `inicio` e `fim` definem o intervalo de busca, a princípio equivale à sequência integral, ou seja, os índices no intervalo [`inicio`..`fim`] são todos aqueles de `sequencia`. Repare que só faz sentido realizar uma busca em uma sequência em que `inicio <= fim`, caso contrário a sequência seria vazia e, evidentemente, o item buscado não estaria nela;
5) O valor buscado será comparado com o item que está no índice do meio do intervalo de busca. Esse índice é obtido pela média aritmética dos extremos `(inicio+fim)//2`. Note que o resultado será um número natural;
6) Caso sejam iguais, a função é encerrada com o retorno do índice deste item;
7) Caso não sejam iguais, o valor pode estar entre os itens menores que o do meio ou entre os itens maiores que ele;
8) Caso o valor buscado seja menor que o item do meio, o procuraremos dentre os menores, atualizando o intervalo de busca para [`inicio`..`meio-1`];

9) Caso o valor buscado seja maior que o item do meio, o procuraremos dentre os maiores, atualizando o intervalo de busca para [`meio+1`..`fim`];
10) O procedimento continuará voltando ao passo 5, enquanto o valor buscado não for encontrado e o intervalo de busca [`inicio`..`fim`] não for vazio;
11) Se [`inicio`..`fim`] ficou vazio, isto é, `inicio > fim`, o valor não foi encontrado e será retornado um valor indicativo, neste caso, `None`.

VOCÊ SABIA?

O Python possui o módulo integrado `bisect` que possui recursos para inserir itens em uma lista mantendo-a ordenada, além de recursos para busca.
Veja mais em: https://docs.python.org/pt-br/3/library/bisect.html

11.3. Ordenação

Ordenar uma sequência consiste em reorganizar seus itens seguindo uma ordem, por exemplo, dispondo-os crescentemente ou decrescendo. Geralmente, sequências ordenadas possibilitam acessos mais eficientes aos seus itens. Há diversos algoritmos para ordenação, na seção 11.3.5 abordaremos alguns populares. Para saber como utilizar a função e o método de ordenação integrados do Python, vá para a seção 11.3.6.

11.3.1. Troca entre itens da sequência

Começaremos com um problema que aborda uma das principais operações feitas durante a ordenação, a *troca* entre itens. Para isso, criaremos uma lista e trocaremos dois de seus itens, de modo que após a troca a lista fique ordenada. Veja a Codificação 11.4.

```
>>> lista = [10, 40, 30, 20, 50]
>>> temp = lista[1]
>>> lista[1] = lista[3]
>>> lista[3] = temp
>>> lista
[10, 20, 30, 40, 50]
```

Codificação 11.4: Troca entre itens de uma lista usando uma variável temporária.

Usar uma variável temporária para preservar o valor de um dos itens que serão trocados é uma abordagem comum, natural e genérica para diversas linguagens. Porém, em Python, podemos melhorar o código utilizando a atribuição paralela, que pode ser feita com o uso de desempacotamento.

Veja na Codificação 11.5 a troca entre dois itens da lista sem a necessidade de uma variável temporária e lembre-se que a expressão à direita da atribuição é uma tupla, porém sem parênteses, e será avaliada antes que os valores sejam atribuídos. Portanto, só após os valores à direita estarem definidos que as atribuições são executadas.

```
>>> lista = [10, 40, 30, 20, 50]
>>> lista[1], lista[3] = lista[3], lista[1]
>>> lista
[10, 20, 30, 40, 50]
```

Codificação 11.5: Troca entre itens de uma lista usando atribuição paralela.

Podemos melhorar! Criaremos uma função, que receberá como argumentos uma sequência mutável `s` e os índices `i` e `j` válidos em `s`. A função trocará `s[i]` com `s[j]`. Veja essa função na Codificação 11.6 e um teste na Codificação 11.7.

```
def troca(s, i, j):
    s[i], s[j] = s[j], s[i]
```

Codificação 11.6: Função que realiza a troca entre dois itens da sequência argumento.

```
>>> lista = [10, 40, 30, 20, 50]
>>> troca(lista, 1, 3)
>>> lista
[10, 20, 30, 40, 50]
```

Codificação 11.7: Troca entre itens de uma lista usando a função `troca`.

11.3.2. Empurrando o item máximo da sequência

Como preparação para o algoritmo de ordenação por flutuação, veremos o funcionamento de sua operação núcleo, baseada em *empurrar* (flutuar) o item máximo de uma sequência para sua última posição.

O algoritmo se baseia em comparar todos os pares de itens adjacentes (vizinhos) da sequência e, se o 1º item do par for maior que o 2º, realizar uma troca entre eles. Veja a Codificação 11.8 que realiza esse procedimento com uma lista.

```
lista = [40, 30, 20, 50, 10]
for i in range(len(lista)-1):
    if lista[i] > lista[i+1]:
        troca(lista, i, i+1)
print(lista)
```

Codificação 11.8: Código que empurra o item máximo para o fim da sequência.

Há um ponto de atenção na Codificação 11.8. A variável `i` assumirá os índices de `lista`, desde o primeiro até seu penúltimo. Isso ocorre pois a condição do `if` se baseia em comparar pares de itens adjacentes, em que o 1º está na posição `i` e o 2º, evidentemente, na posição `i+1`. Caso `i` também assumisse o último índice, não seria possível formar um par, pois não existiria item no índice `i+1`, pois se existisse, estaria após o último, o que é incoerente.

Veja na Tabela 11.1 o teste de mesa da Codificação 11.8 e observe a dinâmica das trocas, em que o item máximo da dupla é empurrado em direção ao fim da lista.

Tabela 11.1: Teste de mesa da Codificação 11.6.

lista					i	lista[i]	lista[i+1]	troca?
[0]	[1]	[2]	[3]	[4]				
40	30	20	50	10	0	40	30	True
30	40	20	50	10	1	40	20	True
30	20	40	50	10	2	40	50	False
30	20	40	50	10	3	50	10	True
30	20	40	10	50				

Podemos melhorar! Criaremos uma função que receberá como argumentos uma sequência mutável `s` e seu *tamanho lógico* `n`. A função deverá empurrar o item máximo da sequência `s` para o final dela. Dizemos que `n` é o tamanho lógico de `s`, pois pode diferir do *tamanho físico*, dado pela função `len`. Essa abordagem é útil em algoritmos onde é necessário trabalhar com apenas parte da sequência, reduzindo-a só logicamente, sem excluir itens. Veja essa função na Codificação 11.9 e um teste na Codificação 11.10.

```
def empurra(s, n):
    for i in range(n-1):
        if s[i] > s[i+1]:
            troca(s, i, i+1)
```

Codificação 11.9: Função que empurra o item máximo para o fim da sequência.

```
>>> lista = [40, 30, 20, 50, 10]
>>> empurra(lista, len(lista))
>>> lista
[30, 20, 40, 10, 50]
```

Codificação 11.10: Empurrando o item máximo da sequência usando a função `empurra`.

11.3.3. Ordenação por flutuação (*Bubble Sort*)

Na Codificação 11.10, após a chamada à `empurra`, você observou que a lista passada como argumento não ficou ordenada. Entretanto, notou que o último item ficou na exata posição onde deveria estar se a lista estivesse ordenada. Isso ocorreu porque `empurra` deslocou o item máximo da lista de tamanho cinco para sua 5ª posição.

Assim, a lista que inicialmente estava com cinco itens não ordenados, após a 1ª chamada à função `empurra(lista, 5)`[2], ordenou um item e manteve quatro não

[2] Lembre-se que na Codificação 11.10, `len(lista)` resultou no valor 5, que era o tamanho físico da lista.

ordenados. Logo, se chamássemos pela segunda vez a função `empurra`, mas definindo o tamanho lógico da lista como quatro, ela empurraria o item máximo da lista com quatro itens para a 4ª posição, aumentando a parte ordenada (agora com dois itens) e reduzindo a parte não ordenada (agora com três itens).

Seguindo esse raciocínio, poderíamos chamar a função `empurra` diversas vezes, sempre com um tamanho lógico menor, que corresponderia à quantidade de itens da parte não ordenada que, naturalmente, é reduzida a cada item máximo empurrado para o fim da sequência. Veja essa estratégia ilustrada na Figura 11.1.

Figura 11.1: Sequência de chamadas à função `empurra`. Fonte: Elaborado pelo autor.

Analisando a Figura 11.1, notamos que ao chamar `empurra` quatro vezes, a sequência tornou-se ordenada, pois a parte não ordenada foi sequencialmente reduzida até ficar vazia. Pense o porquê não faz sentido chamar `empurra` com `n <= 1`.

Agora é possível criar a função de ordenação `bubble_sort`, que empurra os itens da sequência mutável `s` para deixá-la ordenada, conforme Codificação 11.11. Veja um teste desta função na Codificação 11.12, teste-a no Python Tutor.

```
def bubble_sort(s):
    n = len(s)
    while n > 1:
        empurra(s, n)
        n -= 1
```

Codificação 11.11: Definição da função `bubble_sort`.

```
>>> lista = [40, 30, 20, 50, 10]
>>> lista
[40, 30, 20, 50, 10]
>>> bubble_sort(lista)
>>> lista
[10, 20, 30, 40, 50]
```

Codificação 11.12: Exemplo de utilização da função `bubble_sort`.

A Codificação 11.11 é um exemplo de função com efeito colateral, pois altera o valor referenciado pelo parâmetro, mas também poderíamos criar uma versão sem efeitos colaterais, isto é, ao invés de ordenar a lista passada como argumento, retornaria como resposta uma cópia ordenada da lista original. Veja como na Codificação 11.13 e um exemplo de utilização na Codificação 11.14.

```
def bubble_sort_2(lista):
    lista = lista[:]
    n = len(lista)
    while n > 1:
        empurra(lista, n)
        n -= 1
    return lista
```

Codificação 11.13: Função `bubble_sort_2`, ordenação sem efeitos colaterais.

```
>>> lista = [40, 30, 20, 50, 10]
>>> lista
[40, 30, 20, 50, 10]
>>> nova_lista = bubble_sort_2(lista)
>>> nova_lista
[10, 20, 30, 40, 50]
>>> lista
[40, 30, 20, 50, 10]
```

Codificação 11.14: Exemplo de utilização da função `bubble_sort_2`.

Em geral, é mais seguro criar funções que não possuam efeitos colaterais, e essa abordagem pode ser vista em diversas funções integradas do Python, como por exemplo na função integrada para ordenação que será apresentada na seção 11.3.6. Em Python, geralmente, observam-se as seguintes recomendações:

- Métodos de objetos mutáveis tem efeito colateral, alterando as características do próprio objeto, como a ordem dos itens em uma lista;
- Funções que recebem objetos mutáveis como argumentos, não alteram os objetos, retornando um novo objeto gerado a partir do original.

Caso seja necessário criar uma função com efeitos colaterais, seja por questões de otimização de desempenho ou por não ser necessário preservar o argumento, é importante que isso esteja bem documentado para os potenciais usuários da função.

11.3.4. Estabilidade

Dizemos que um algoritmo de ordenação é estável quando elementos iguais mantêm a mesma ordem entre si após serem ordenados. Essa característica é útil quando são feitas ordenações sucessivas em um mesmo conjunto de dados, como por exemplo, ordenar uma lista de clientes pelo nome e depois por nascimento, de modo que clientes com o mesmo aniversário ainda permaneçam ordenados alfabeticamente. Ou seja, é útil em aplicações onde uma sequência de itens podem ser ordenados por diversos filtros, como por exemplo, no refinamento da busca por um produto em um site de compras.

11.3.5. Algoritmos de ordenação populares

Existem dezenas de algoritmos de ordenação desenvolvidos e aprimorados por matemáticos e cientistas de diferentes países. Muitos algoritmos são aperfeiçoamentos de anteriores, para consumir menos recursos ou para lidar com problemas específicos.

Sabemos que um algoritmo indica os passos para resolver um problema, portanto ao escrever um programa, instruímos ao computador como executar os passos de um algoritmo. Assim, podem existir diversas implementações de um mesmo algoritmo, que podem apresentar desempenhos diferentes, mesmo que gerem o mesmo resultado.

Essa disciplina não busca analisar a eficiência de algoritmos, apenas introduzir o conceito de ordenação. Além do método de ordenação por flutuação (*bubble sort*), exporemos resumidamente alguns algoritmos de ordenação bastante conhecidos:

- ***Selection Sort*** (ordenação por seleção): consiste em percorrer a parte não ordenada de uma sequência buscando o maior item (ou menor), e então trocá-lo com o último (ou primeiro) item da parte não ordenada. Deste modo, aumenta-se a parte ordenada e reduz-se a não ordenada. O processo é repetido até a sequência estar completamente ordenada. Essa é a forma que normalmente ordenamos uma "mão" de cartas, selecionando uma carta por vez e deslocando-a para sua posição final. É um algoritmo estável e eficiente no consumo de memória, porém inviável em sequências grandes pelo grande número de comparações necessárias;
- ***Insertion Sort*** (ordenação por inserção): assume-se que a parte ordenada da sequência é composta inicialmente só pelo primeiro item, todos os demais pertencem a parte não ordenada. Então, percorre-se a sequência a partir do primeiro item não ordenado, inserindo-o na parte ordenada, de modo que ela permaneça assim, até que a parte não ordenada fique vazia. Essa é a forma

que normalmente ordenamos cartas ao "comprarmos" uma de cada vez, inserindo-a em ordem dentre as cartas que já estão em ordem na mão. Como o *Selection Sort*, a ordenação por inserção é estável e eficiente no uso de memória, mas igualmente impraticável em sequências grandes;

- ***Merge Sort*** (ordenação por mistura/intercalação): consiste em dividir a sequência sucessivamente até chegar a subsequências de um item. Então, faz-se o processo inverso, unindo (misturando) as sequências duas a duas, comparando seus itens e colocando-os em ordem. Este algoritmo é estável e muito mais eficiente em quantidade de comparações quando contrastado com *Selection Sort* e *Insertion Sort*, mas requer mais memória para ser executado;
- ***Quicksort*** (ordenação rápida): consiste na escolha de um item arbitrário da sequência como pivô, então divide-se a sequência em duas partes, uma com os itens menores que o pivô e outra com itens maiores que ele. Assim, o pivô estará na sua posição ordenada, mas teremos duas subsequências não ordenadas. Portanto, repete-se o processo em cada subsequência até que estejam ordenadas. Algoritmo bastante eficiente no número de comparações e uso de memória, mas, por padrão, não garante estabilidade na ordenação.

Com sequências pequenas, de algumas dezenas de itens, os algoritmos *Selection Sort* e *Insertion Sort* têm melhor desempenho por terem instruções mais simples e que demandam menos recursos computacionais. Porém, se a sequência é grande, o número de comparações torna-os inviáveis. Nestes casos, aplicam-se algoritmos mais sofisticados como *Merge Sort* e *Quick Sort*, que requerem mais recursos, mas são mais eficientes.

Visando melhor desempenho, alguns algoritmos de ordenação integrados às linguagens de programação são híbridos. Por exemplo, começam a ordenação com o *Merge Sort,* mas ao atingir subsequências pequenas, o trocam pelo *Insertion Sort*.

11.3.6. Funções e métodos integrados de ordenação

Uma forma simples de ordenar sequências em Python é com a função integrada `sorted`. Essa função recebe qualquer tipo de sequência como argumento e retorna uma nova lista com os itens em ordem crescente. Veja um exemplo na Codificação 11.15.

```
>>> lista = [40, 30, 20, 50, 10]
>>> nova_lista = sorted(lista)
>>> nova_lista
[10, 20, 30, 40, 50]
>>> lista
[40, 30, 20, 50, 10]
```

Codificação 11.15: Ordenação de uma sequência com a função `sorted`.

Note que `sorted` não possui efeitos colaterais, logo a sequência argumento permanece inalterada. Para situações em que a sequência não precisa ser preservada, há o método `sort`, exclusivo para listas. Em comparação com a função `sorted`, é ligeiramente mais eficiente em tempo de processamento e bem mais eficiente em consumo de memória, pois não duplica a lista. Veja um exemplo na Codificação 11.16.

```
>>> lista = [40, 30, 20, 50, 10]
>>> lista.sort()
>>> lista
[10, 20, 30, 40, 50]
```

Codificação 11.16: Ordenação de uma lista com o método `sort`.

VOCÊ SABIA?

O algoritmo de ordenação usado na função `sorted` e no método `sort` é o *Timsort*, um algoritmo híbrido bastante eficiente, uma combinação do *Merge Sort* e *Insertion Sort*. Foi implementado por Tim Peters em 2002 (Peters, 2002), e também é utilizado em diversas outras linguagens de programação.

O algoritmo garante a estabilidade da ordenação e seu código-fonte, feito em linguagem C, pode ser visto no GitHub (Rossum, 2021), na implementação dos objetos *list*. Recebeu contribuições de mais de 70 membros da comunidade Python ao redor do mundo, desde que se tornou padrão na linguagem, a partir do Python 2.3.

Para aprender outras formas de uso da função e método de ordenação integrados, veja o tutorial oficial na documentação do Python (PSF, 2021a). E para saber mais sobre a implementação do algoritmo *Timsort* no Python, leia a documentação criada por Tim Peters, disponível também no GitHub (Peters, 2021).

11.4. Argumentos nomeados e argumentos com valor padrão

Sabemos que é possível alterar o comportamento da função `print` estipulando a *string* separadora de argumentos e a *string* terminadora de impressão. Por omissão, o comportamento padrão estabelece um espaço como separador e um quebra de linha como terminador. Relembre com os exemplos da Codificação 11.17.

```
>>> print('Ana', 'Bia', 'Clô')
Ana Bia Clô
>>> print('Ana', 'Bia', 'Clô', sep=' & ', end='!')
Ana & Bia & Clô!
```

Codificação 11.17: Exemplos de chamada à função `print`.

Essa forma de uso é possível porque `print` foi definida com *argumentos com valor padrão* e podemos chamá-la passando *argumentos nomeados*. Veremos como incluir esse comportamento em nossas próprias funções.

11.4.1. Argumentos nomeados

Por padrão, argumentos são *posicionais*, isso significa que ao chamar uma função os argumentos são associados aos parâmetros de acordo com a posição em que são escritos. Esse comportamento pode ser alterado com argumentos nomeados. Para isso, chama-se a função passando os argumentos como valores atribuídos aos nomes dos parâmetros, por isso é dito que os argumentos são "nomeados".

Para descobrir os nomes dos parâmetros, podemos: (a) consultar a documentação da função; (b) usar o editor de código, que pode exibir a assinatura da função enquanto escrevemos seu nome ou; (c) chamar a função integrada `help`, que recebe como argumento o nome da função que se deseja consultar. Veja na Codificação 11.18 exemplos de chamadas à uma função usando argumentos nomeados.

```
def divide(numerador, denominador):
      return numerador / denominador

a = 10
b = 5
div1 = divide(numerador=a, denominador=b)
div2 = divide(denominador=b, numerador=a)
div3 = divide(numerador=b, denominador=a)
print(f'1º: {a}/{b} = {div1}')  # 1º: 10/5 = 2.0
print(f'2º: {a}/{b} = {div2}')  # 2º: 10/5 = 2.0
print(f'3º: {b}/{a} = {div3}')  # 3º: 5/10 = 0.5
```

Codificação 11.18: Chamada à uma função passando argumentos nomeados.

A ordem da passagem dos argumentos nomeados não importa, mas é inválido passar argumentos nomeados antes de posicionais, pois seria impossível determinar quais parâmetros receberiam os posicionais. Veja na Codificação 11.19 uma chamada válida e outra inválida de acordo com a combinação de argumentos posicionais e nomeados.

```
>>> divide(6, denominador=2)  # chamada válida
3
>>> divide(numerador=6, 2)  # chamada inválida
SyntaxError: positional argument follows keyword argument
```

Codificação 11.19: Chamada inválida, argumento nomeado antes de posicional.

Passar dois argumentos para o mesmo parâmetro, ou seja, um posicional e outro nomeado, causará erro de execução, como mostra a Codificação 11.20.

```
>>> divide(6, numerador=2)
...
TypeError: divide() got multiple values for argument 'numerador'
```

Codificação 11.20: Chamada inválida, dois argumentos para o mesmo parâmetro.

11.4.2. Argumentos com valor padrão

Geralmente, quando uma função é chamada, passa-se a quantidade de argumentos equivalente à quantidade de parâmetros definidos em sua criação, logo uma função com três parâmetros é chamada com três argumentos, por exemplo. Porém, é possível criar funções com argumentos pré-definidos, nestas funções é possível chamá-las omitindo alguns argumentos. Assim, a passagem desses argumentos padrão é opcional.

Para criar funções com argumentos padrão, basta definir parâmetros com uma atribuição de valor, como `senha` na Codificação 11.21.

```
def recepcao_cliente(nome, senha=1):
    print(f'Olá {nome}, sua senha é {senha}')
```

Codificação 11.21: Função com argumento padrão para o parâmetro `senha`.

O argumento correspondente à `senha` poderá ser omitido quando a função for chamada e, se isso ocorrer, assumirá o valor `1`, como podemos ver na Codificação 11.22.

```
>>> recepcao_cliente('Megan')
Olá Megan, sua senha é 1
>>> recepcao_cliente('Megan', 23)
Olá Megan, sua senha é 23
```

Codificação 11.22: Chamadas à uma função com argumento padrão.

A chamada a uma função com argumentos nomeados e a definição argumentos padrão são recursos distintos, mas podem ser combinados. Como na Codificação 11.23.

```
>>> recepcao_cliente(nome='Megan')
Olá Megan, sua senha é 1
>>> recepcao_cliente(senha=23, nome='Megan')
Olá Megan, sua senha é 23
```

Codificação 11.23: Uso combinado de argumento nomeado e argumento padrão.

Na definição da função, parâmetro com argumento padrão não pode anteceder parâmetro sem argumento padrão. Veja um exemplo de função inválida por violação desta regra na Codificação 11.24 e o erro resultante na Figura 11.2.

```
def desconto(porcentagem=0.05, valor):
    return valor * porcentagem
```

Codificação 11.24: Função inválida pela ordem do parâmetro com argumento padrão.

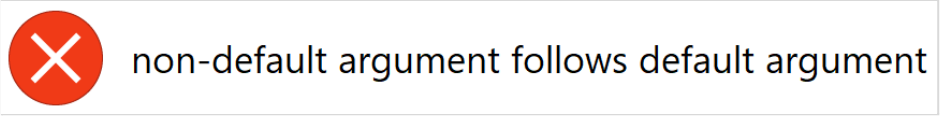


Figura 11.2: Erro de sintaxe na Codificação 11.24. Fonte: Elaborado pelo autor.

11.4.3. Objetos mutáveis como argumentos padrão

É importante entender o comportamento dos objetos mutáveis em Python quando são usados como argumentos padrão. Ao definir uma função com argumentos padrão, Python criará os objetos de cada argumento apenas uma vez, no momento em que a função for definida. Portanto, mesmo que a função seja chamada várias vezes, os valores dos argumentos padrão não serão reconstruídos e eventuais alterações serão mantidas.

Vejamos o seguinte problema: *"crie uma função que receba como argumentos um objeto e uma lista, a função deve adicionar o objeto ao final da lista e retorná-la. Caso não seja fornecida uma lista na chamada, a função deve criar uma nova lista vazia"*. Após aprender sobre argumentos padrão, algum iniciante em Python poderia elaborar a solução da Codificação 11.25.

```
def adiciona_item(x, lista=[]):
    lista.append(x)
    return lista
```

Codificação 11.25: Solução inicial usando uma lista vazia como argumento padrão.

Se testarmos essa função na *Shell* do Python, passando uma lista como argumento, vemos que ela se comporta como esperado, veja um exemplo na Codificação 11.26.

```
>>> x = adiciona_item(5, [1, 3])
>>> x
[1, 3, 5]
>>> y = adiciona_item('c', ['a', 'b'])
>>> y
['a', 'b', 'c']
```

Codificação 11.26: Utilização da função `adiciona_item` com 2 argumentos.

Porém, veja na Codificação 11.27 o que ocorre ao não passar o argumento lista.

```
>>> x = adiciona_item(3)
>>> x
[3]
>>> y = adiciona_item('a')
>>> y
[3, 'a']
>>> z = adiciona_item(1)
>>> z
[3, 'a', 1]
```

Codificação 11.27: Utilização da função `adiciona_item` com apenas um argumento.

Observamos que o Python está adicionando os objetos sempre à mesma lista, pois a cada chamada usa-se o mesmo argumento criado na definição da função.

Para evitar a criação do objeto mutável na definição da função e gerar um novo a cada execução, podemos usar `None` como argumento padrão e criar o objeto necessário caso não seja passado um argumento (PSF, 2021b, PSF, 2021c). Veja essa reformulação na Codificação 11.28 e exemplos de execução na Codificação 11.29.

```
def adiciona_item(x, lista=None):
    if lista is None:
        lista = []

    lista.append(x)
    return lista
```

Codificação 11.28: Reformulação da função `adiciona_item`.

```
>>> x = adiciona_item(3)
>>> x
[3]
>>> y = adiciona_item('a')
>>> y
['a']
>>> z = adiciona_item(1)
>>> z
[1]
```

Codificação 11.29: Chamadas à nova versão da função `adiciona_item`.

O programador deve ficar atento a esse comportamento, pois se mal utilizado pode gerar problemas difíceis de serem rastreados. Entretanto, há soluções que podem ser beneficiadas como a criação de uma memória temporária e compartilhada entre as chamadas de uma função, por exemplo, para implementar a memoização[3].

Um exemplo de uma função com argumento padrão que é um objeto mutável pode ser visto na Codificação 11.30, uma função que conta quantas vezes foi executada.

```
def teste(execucoes=[0]):
    execucoes[0] += 1
    print(f'execução número: {execucoes[0]}')
```

Codificação 11.30: Função que guarda a quantidade de vezes que foi executada.

Na documentação da função da Codificação 11.30, é importante avisar aos programadores que o argumento do parâmetro `execucoes` não deve ser fornecido na chamada da função, pois caso seja, a execução não será contabilizada. Faça um teste no Python Tutor, chamando a função algumas vezes.

[3] Sim, o termo é *memoização*, uma técnica para otimizar a resolução de problemas repetitivos, pois armazena as respostas previamente calculadas em um tipo de cache. Não será abordada na disciplina.

Bibliografia e referências

DOWNEY, A. B., **Pense em Python**. 1 ed. São Paulo: Novatec Editora Ltda., 2016.

PETERS, T., **Mailing lists: [Python-Dev] Sorting**. 2002. Disponível em: <https://mail.python.org/pipermail/python-dev/2002-July/026837.html>. Acesso em: 14 mar. 2021.

PETERS, T., **Descrição do algoritmo de ordenação do Python**. 2021. Disponível em: <https://github.com/python/cpython/blob/master/Objects/listsort.txt>. Acesso em: 14 mar. 2021.

PSF. **Python HOWTOs: Ordenação**. 2021a. Disponível em: <https://docs.python.org/pt-br/3/howto/sorting.html#sortinghowto>. Acesso em: 14 mar. 2021.

PSF. **The Python Language Reference: Function Definition**. 2021b. Disponível em: <https://docs.python.org/3/reference/compound_stmts.html#function-definitions>. Acesso em: 14 mar. 2021.

PSF. **The Python Tutorial: More Control Flow Tools**. 2021c. Disponível em: <https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions>. Acesso em: 15 mar. 2021.

ROSSUM, G. V. **Código fonte dos objetos de lista**. 2021. Disponível em: <https://github.com/python/cpython/blob/master/Objects/listobject.c>. Acesso em: 14 mar. 2021.