5

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

5

Strings e estruturas de seleção aninhadas

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

Resumo

Nesta aula os objetivos são: (I) avançar o estudo de strings; (II) introduzir o conceito de flags booleanas; (III) apresentar as estruturas de seleção aninhadas e encadeadas; (IV) entender o funcionamento do comando elif.

5.1. Motivação

Conhecemos vários tipos de dados em Python, inclusive as *strings*. No entanto, conforme os problemas se sofisticam, precisaremos de mais recursos para manipular textos, por isso vamos nos aprofundar nas operações que podemos realizar sobre *strings*.

Também sabemos criar programas que tomam decisões simples, selecionando um entre dois caminhos, mas isso é insuficiente para lidar com problemas mais complexos, envolvendo árvores de decisão, em que uma decisão está dentro de outra. Portanto, ampliaremos e sofisticaremos essa estrutura de controle, com seleções mais elaboradas, possibilitando a resolução de problemas envolvendo mais níveis de decisão.

💎 VOCÊ CONHECE?

Grace Hopper foi uma importante programadora, formada em matemática e física que, dentre diversos feitos, construiu compiladores que foram base para o projeto que gerou a popular linguagem de programação Cobol.

Tornou-se tenente e trabalhou com cálculos secretos no Mark I, o primeiro computador eletromecânico dos Estados Unidos.

Fonte da imagem: https://www.biography.com/scientist/grace-hopper

5.2. *Strings*

Já utilizamos *strings* em nossos programas, são usadas para representar cadeias de caracteres, ou seja, textos. Agora poderemos aprofundar um pouco mais sobre esse tipo de dados, aprendendo a concatenar, repetir, comparar e formatar *strings*.

5.2.1. Concatenação

Quando precisamos juntar duas ou mais *strings*, realizamos uma operação chamada de concatenação de *strings*. Em Python, essa operação é feita usando o operador `+`. Veja na Codificação 5.1 um exemplo executado na *Shell*.

```
>>> 'Bom' + ' ' + 'dia' + '!'
'Bom dia!'
```

Codificação 5.1: Exemplo de concatenação de *strings*.

É importante notar que a operação de concatenação utiliza o mesmo operador que vimos anteriormente como *adição* ou *identidade* (`+`), porém tal definição se aplica apenas quando os operandos envolvidos na expressão são numéricos. Isso é comum em Python e também em outras linguagens de programação.

VOCÊ SABIA?

Quando um operador tem comportamento diferente dependendo dos operandos em que é aplicado, dizemos que ocorreu uma *sobrecarga do operador*. Essa característica é útil e permite facilidades de sintaxe como a concatenação. Veja mais em: https://pense-python.caravela.club/17-classes-e-metodos/07-sobrecarga-de-operadores.html

A concatenação não altera seus operandos, mas gera uma nova *string* que pode, inclusive, ser atribuída a uma variável. Veja isso na Codificação 5.2.

```
>>> nome = 'Megan'
>>> idade = 34
>>> x = 'Olá ' + nome + '! Você tem ' + str(idade) + ' anos.'
>>> print(x)
>>> print(nome, idade + 5)
```

Codificação 5.2: Exemplo de concatenação de *strings* seguida de atribuição.

A função integrada `str` recebe como argumento um valor de qualquer tipo e retorna a representação do valor como uma *string*. Tivemos que usar essa função para converter o valor da variável `idade`, um inteiro, para *string* e assim poder fazer a concatenação com as demais *strings*. Para aprendizado, tente realizar a concatenação sem a conversão e veja o que acontece.

5.2.2. Repetição

Uma forma simples de gerar uma nova *string* cujo conteúdo seja a repetição do conteúdo de outra é com o operador `*`, que é sobrecarregado quando um de seus operandos é uma *string* e o outro é um número inteiro, conforme a Codificação 5.3.

```
>>> risada_timida = 'kk'
>>> risada_longa = risada_timida * 3
>>> risada_longa
'kkkkkk'
```

Codificação 5.3: Exemplo de repetição de *string*.

5.2.3. Comparação

Podemos realizar comparações entre *strings* com operadores relacionais, os mesmos usados para verificar relações entre números. Exemplos na Codificação 5.4.

```
>>> 'maria' != 'MaRiA' # Exemplo 1
True
>>> 'maria' == 'MARIA' # Exemplo 2
False
>>> 'Meg' < 'Megan'    # Exemplo 3
True
>>> 'beatriz' > 'bia'  # Exemplo 4
False
```

Codificação 5.4: Exemplo de comparação entre *strings*.

Talvez surjam dúvidas sobre o Exemplo 4, em que `'beatriz' > 'bia'` resulta em `False`, neste caso, possivelmente o motivo da expressão do Exemplo 3 resultar em `True`, também não foi completamente entendido. Para melhor compreensão, precisamos de mais detalhes sobre como a comparação entre *strings* é realizada.

Em Python a comparação entre *strings* ocorre caractere a caractere, ou seja, compara-se o primeiro caractere da *string* à esquerda com o primeiro caractere da *string* à direita do operador. Enquanto as *strings* não acabarem (existirem caracteres em ambas as *strings* que ainda não foram comparados) e os caracteres comparados forem iguais, compara-se o próximo par. A Figura 5.1 ilustra o procedimento com base no Exemplo 4 da Codificação 5.4.

Strings de tamanhos distintos, ou seja, com quantidades diferentes de caracteres, são consideradas diferentes entre si, assim como *strings* com caracteres diferentes.

Note que não é o tamanho da *string* que determina se ela é menor ou maior que outra, mas sim seu conteúdo. Portanto, `'beatriz'` não é maior do que `'bia'`, pois o caractere `'e'` de `'beatriz'` não é maior do que `'i'` de `'bia'`.

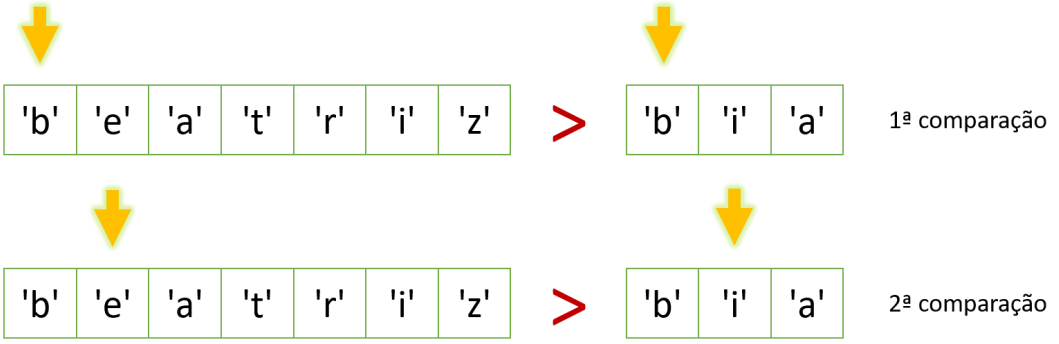

Figura 5.1: Procedimento de comparação entre *strings*. Fonte: Elaborado pelo autor.

Mas qual a razão para um caractere ser identificado como "menor" ou "maior" do que outro? Intuitivamente, podemos assumir a ordem alfabética, em que um caractere X é considerado menor do que um caractere Y se X antecede Y no alfabeto. Porém isso é apenas uma simplificação, que não é suficiente para o exemplo da Codificação 5.5.

```
>>> 'oi!' < 'oi?'
True
```

Codificação 5.5: Exemplo de comparação entre *strings*.

Na Codificação 5.5, as *strings* têm o mesmo tamanho e conteúdo, exceto pelo último caractere. Como saber se `'!'` é menor do que `'?'` se não estão em nosso alfabeto? Usaremos a função integrada `ord` que recebe como argumento um caractere e devolve como resposta um número natural correspondente, conforme a Codificação 5.6.

```
>>> ord('!')
33
>>> ord('?')
63
```

Codificação 5.6: Exemplo de uso da função `ord`.

Quando caracteres são comparados, internamente ocorre a comparação entre números naturais que correspondem a esses caracteres. Logo, é fácil perceber que 33 é menor do que 63, justificando a resposta de `'oi!' < 'oi?'`. A dúvida é *"o que são esses números que correspondem aos caracteres?"*. São códigos da tabela Unicode.

VAMOS LER!

Unicode é uma padronização que permite que computadores representem e manipulem caracteres de quase todos os sistemas de escrita existentes. Na tabela Unicode existem códigos associados a símbolos, sendo que cada código está mapeado para apenas um símbolo. Há milhares de códigos, basta consultá-los em: https://unicode-table.com/pt/

Não nos aprofundaremos neste tópico, sendo suficiente saber que os códigos são sequenciais e que nosso alfabeto está disposto nesta tabela da forma como estamos

habituados, consequentemente `'A'` é menor que `'B'`, `'B'` é menor que `'C'` e assim sucessivamente. Também é importante saber que as letras maiúsculas são listadas nesta tabela antes das letras minúsculas, logo os códigos das maiúsculas são menores que os das minúsculas, justificando o resultado da Codificação 5.7.

```
>>> 'ANA' < 'ana'
True
>>> print('A =', ord('A'), '| a =', ord('a'))
A = 65 | a = 97
```

Codificação 5.7: Comparação de *strings* com letras maiúsculas e minúsculas.

Por fim, temos a função integrada `chr` que realiza o inverso da função `ord`, isto é, recebe como argumento um número natural representando um código Unicode e retorna como resposta o caractere correspondente. Veja o exemplo na Codificação 5.8.

```
>>> print(chr(48), chr(49), chr(50), chr(51), chr(9733))
0 1 2 3 ★
```

Codificação 5.8: Exemplo de uso da função `chr`.

5.2.4. Formatação

Às vezes é necessário formatar *strings*, estipulando, por exemplo, quantidade de colunas, alinhamento, quantidade de casas decimais para representar *floats*, etc. O Python possui algumas formas para criar *strings* formatadas[1], dentre as quais citamos:

1) *Interpolação:* formatação utilizando o operador de interpolação (`%`), com sintaxe próxima ao estilo usado na função `printf` da Linguagem C. É compatível com todas as versões do Python, mas seu uso é desencorajado para novos projetos;
2) *Método format:* método de *strings* em que os argumentos são formatados e inseridos nos marcadores identificados por pares de chaves. Aceito a partir do Python 2.6. É um avanço em relação à interpolação, com maior controle sobre a formatação, mas aumenta a verbosidade podendo reduzir a legibilidade do código. É recomendado para projetos Python que antecedem à versão 3.6;
3) *f-strings ou "strings literais formatadas":* é a abordagem mais recente para formatação de *strings*, basicamente uma *string* comum prefixada com a letra `f` ou `F` (antes da abertura de aspas/apóstrofos) e com valores ou expressões entre pares de chaves que serão formatados e inseridos na *string*. Compatível com as versões do Python a partir da 3.6, é a forma recomendada para novos projetos;
4) *Template strings:* é uma forma especial de criação de *strings* e é recomendada para situações em que a formatação da *string* será fornecida pelo usuário final, pois seu funcionamento traz algumas proteções contra injeção de código malicioso. Não será abordada neste curso.

[1] Para saber mais leia Python String Formatting Best Practices – Real Python

Nesta disciplina optamos pelas *f-strings*, pois possuem vantagens em legibilidade e desempenho, além de serem indicadas pela comunidade (Bader, 2018).

Por ser um tópico extenso, aprenderemos inicialmente a usar *f-strings* com as formatações mais recorrentes em nossos programas. Começaremos com um exemplo simples, como pode ser visto na Codificação 5.9.

```
>>> mensagem = f'2 + 3 = {2 + 3}, entendeu?'
>>> mensagem
'2 + 3 = 5, entendeu?'
```

Codificação 5.9: Exemplo de formatação de *strings*.

Note que a expressão entre o par de chaves é avaliada e o resultado inserido no mesmo ponto em que está escrita na *string*. A avaliação ocorre em tempo de execução, o que permite o uso de *f-strings* com variáveis e expressões, como na Codificação 5.10.

```
>>> salario = float(input('Seu salário: '))
Seu salário: 1000.00
>>> f'20% a mais em R$ {salario} dará R$ {salario * 1.2}'
'20% a mais em R$ 1000.0 dará R$ 1200.0'
```

Codificação 5.10: Exemplo de formatação de *strings*.

Vimos que utilizar *f-strings* para formatação melhora a legibilidade do código quando comparado com a concatenação ou com a passagem de diversos argumentos para a função `print`. Entretanto, só isso não seria suficiente para problemas que exigem formatação mais elaborada, então veremos algumas possibilidades de manipulação de texto com o uso de *f-strings*.

No Codificação 5.10, seria útil exibir os valores monetários com duas casas decimais. Para especificar a formatação de um valor em uma *f-string*, o pós-fixamos com `:` e acrescentamos alguns especificadores.

Para formatar um dado do tipo *float*, podemos usar o seguinte padrão de formatação: `f'{<valor>:<colunas>.<decimais>f}'`. Onde:

- `colunas`: a quantidade mínima de colunas reservadas para o valor na *string* formatada, note que cada caractere do valor ocupa uma coluna, inclusive o ponto. Se omitido, assume-se a quantidade mínima para expressar o número;
- `decimais`: o número total de casas decimais que serão representadas na *string*, a última casa à direita é arredondada. Se omitido, o padrão será de seis casas;
- `f`: indica que a formatação será feita para o tipo *float*, podendo haver uma conversão automática entre tipos compatíveis, como o *int*.

Veja a Codificação 5.11, em seguida reescreva a Codificação 5.10, porém formatando os valores em reais com duas casas decimais.

```
>>> pi = 3.14159265
>>> f'{pi:f}'  # sem especificação o padrão são 6 casas decimais
'3.141593'
>>> f'{pi:.3f}' # note o arredondamento na última casa decimal
'3.142'
>>> f'{pi:7.3f}' # 7 colunas com 3 reservadas para parte decimal
'  3.142'
```

Codificação 5.11: Exemplos de formatação de *strings* com especificadores.

Existem outros especificadores para formatação de *float*, como exibição em notação científica, e também para outros tipos, como *int* e *string*. A lista completa com as explicações e exemplos de uso pode ser vista na documentação oficial do Python[2].

Assim como visto com valores do tipo *float*, podemos definir o tamanho mínimo de colunas para qualquer *string* formatada, independente do tipo do dado. O padrão para isso é `f'{<valor>:<colunas>}'`, como consta na Codificação 5.12.

```
>>> nome = 'Megan'
>>> f'{nome:10}'
'Megan     '
```

Codificação 5.12: Formatação de quantidade mínima de colunas para um valor qualquer.

Nos exemplos de formatação de *float*, percebemos que o alinhamento padrão de números é à direita, já para valores do tipo *string* o padrão é à esquerda, mas em ambos os casos podemos controlar o alinhamento adicionando um dos símbolos da Tabela 5.1, de acordo com o padrão `f'{<valor>:<alinhamento><colunas>}'`.

Tabela 5.1: Símbolos de alinhamento.

Símbolo	Alinhamento
<	À esquerda.
>	À direita.
^	Centralizado.

Teste a Codificação 5.13 na *Shell*, lembrando que o valor também poderia estar em uma variável ou ser o resultado de uma expressão.

Por padrão, colunas não ocupadas por caracteres do valor são preenchidas com espaços. Para personalizar o caractere de preenchimento adiciona-se outro especificador

[2] Format Specification Mini-Language (Python Software Foundation, 2021).

antes do alinhamento: `f'{<valor>:<caractere><alinhamento><colunas>}'`. Veja um exemplo na Codificação 5.14.

```
>>> f'{"à esquerda":<30}'
'à esquerda                    '
>>> f'{"à direita":>30}'
'                    à direita'
>>> f'{"centro":^30}'
'            centro            '
```

Codificação 5.13: Formatação de alinhamento do conteúdo de uma *string*.

Observe que o ponto (`.`) usado na formatação de cada uma das variáveis na Codificação 5.14 tem significado distinto, pois estão em posições diferentes no padrão que define a formatação.

```
>>> item = 'camiseta'
>>> preco = 49.9
>>> f'{item:.<20} R$ {preco:.2f}'
'camiseta..... R$ 49.90'
```

Codificação 5.14: Formatação de preenchimento de colunas não utilizadas pelo valor.

5.3. *Flags* booleanas

O termo *flag* vem do inglês e significa "bandeira" e bandeiras servem para sinalizar algo, então uma *flag* booleana é usada para sinalizar algo no código. Podemos usá-las para facilitar a leitura de estruturas de seleção e também de repetição indefinida, que veremos mais à frente no curso.

Uma *flag* booleana nada mais é do que uma variável que guarda o resultado de uma comparação ou expressão relacional/lógica. Assim, ao acessarmos o valor dessa variável ao longo do fluxo de execução do código, podemos saber se a condição em questão foi atendida ou não, podemos pensar em uma bandeira levantada (*True*) ou não (*False*). Veja a Codificação 5.15, base para a Codificação 5.16 que usará *flag* booleana.

```
idade = int(input('Qual a sua idade? '))

if idade >= 18:
    print('Você é maior de idade.')
```

Codificação 5.15: Programa que verifica se o usuário é maior de idade (sem *flag*).

Na codificação acima, o bloco de código do `if` só será executado quando o usuário fornecer como entrada um inteiro maior ou igual a 18. Podemos reescrever este exemplo utilizando uma *flag* booleana para guardar o "resultado da condição" que

queremos averiguar. Recomenda-se que o nome da variável seja representativo em relação ao que será avaliado. O código implementado está na Codificação 5.16.

```
idade = int(input('Qual a sua idade? '))
maior_de_idade = idade >= 18

if maior_de_idade:
    print('Você é maior de idade.')
```

Codificação 5.16: Programa que verifica se o usuário é maior de idade (com *flag*).

Note que a Codificação 5.15 e Codificação 5.16 são equivalentes, porém, nesta última, não há operadores na condição do `if`, apenas uma variável com valor booleano. Vamos incluir mais condições para podermos visualizar as vantagens no uso de *flags* booleanas. Insira a Codificação 5.17 no editor, teste a sua execução e em seguida pense em como substituir as condições por *flags* para facilitar a leitura do código.

```
idade = int(input('Qual a sua idade? '))
cnh = input('Você tem CNH? (s/n)')

if idade >= 18 and cnh == 's':
    print('Você pode dirigir.')
```

Codificação 5.17: Programa que verifica se o usuário pode dirigir (sem *flag*).

A Codificação 5.18 é equivalente à Codificação 5.17, mas com *flags*. Não existe apenas uma versão válida, o importante é usar identificadores significativos para o problema, minimizando ambiguidades e prezando pela simplicidade.

```
idade = int(input('Qual a sua idade? '))
cnh = input('Você tem CNH? (s/n)')

maior_de_idade = idade >= 18
possui_cnh = cnh == 's'
pode_dirigir = maior_de_idade and possui_cnh

if pode_dirigir:
    print('Você pode dirigir.')
```

Codificação 5.18: Programa que verifica se o usuário pode dirigir (com *flag*).

O uso de *flags* booleanas pode facilitar a legibilidade do código, aproximando a leitura do código-fonte ao que normalmente falaríamos em nosso idioma natural, mas é preciso cuidado, pois a utilização de nomes inadequados pode acarretar erros de interpretação sutís, de difícil correção. Logo, é extremamente importante que haja uma relação direta entre a condição/expressão avaliada e o nome escolhido para a *flag*.

Note que a escolha entre usar ou não *flags* booleanas dependerá da situação e experiência da programadora/programador, algo naturalmente desenvolvido com treino. Cabe a cada um encontrar um equilíbrio em seu uso, assim como qualquer recurso em programação, portanto, procure praticar todas as diferentes formas, com foco primeiramente em entender como funcionam e não em sua aplicabilidade definitiva. Cada recurso aprendido é um novo instrumento em sua caixa de ferramentas.

As principais vantagens de se usar *flags* booleanas são:

- Melhorar legibilidade do código, facilitando a construção e o entendimento de condições mais complexas;
- Guardar a informação relativa a uma determinada condição avaliada durante a execução do código;
- Aprimorar o desempenho do programa quando uma condição deve ser avaliada diversas vezes, por exemplo em laços de repetição, assunto ainda não abordado.

Vale lembrar que o uso exagerado de *flags* booleanas, seja colocando-as em todas as situações possíveis, seja usando-as muitas vezes ao longo do código, pode resultar no efeito contrário ao esperado, elevando a complexidade do código e piorando sua legibilidade, pois pode aumentar desnecessariamente a verbosidade do código.

O excesso de verbosidade pode ocorrer especialmente em problemas simples, como os vistos nos exemplos desta aula. Em uma situação real na qual fosse necessária a verificação de idade de uma pessoa, boa parte dos desenvolvedores provavelmente realizaria a comparação diretamente na condição do `if`, pois `idade >= 18` é uma comparação com interpretação simples (conhecendo o contexto entende-se que o objetivo é saber se uma pessoa é maior de idade), então o uso da *flag* `maior_de_idade` aumenta desnecessariamente a complexidade do código.

Um questionamento comum é relativo ao consumo extra de memória pelas *flags*, o que poderia piorar o desempenho da solução. Quanto a isso, é seguro dizer que para a maioria das aplicações, o consumo extra é insignificante.

Na implementação do Python em C[3], são alocados 28 *bytes* para uma variável com o valor `True` e 24 *bytes* para o valor `False`, mais alguns *bytes* para guardar o nome da variável em si. Portanto, na maioria das situações, é desnecessária a preocupação com a quantidade de memória gasta com variáveis simples, e a decisão sobre o uso de uma *flag* booleana deve ser feita com base na legibilidade do código, com foco em deixá-lo mais simples e fácil de entender.

[3] A implementação padrão, e mais utilizada, do interpretador do Python é feita na linguagem C, porém há também interpretadores feitos em Java, C#, Rust e no próprio Python, entre outras linguagens. Lembrando que o interpretador é o responsável por traduzir nossos programas para linguagem de máquina (binário).

5.4. Estruturas de seleção aninhadas

A única regra para o código dentro de um bloco de `if` ou `else` é que ele deve ser um código válido em Python. Portanto, é possível que exista uma estrutura de seleção dentro de outra. Basta seguir as mesmas regras que vimos na definição de blocos de código em Python e de como `if` e `else` se relacionam:

1) O bloco de código é introduzido por "dois pontos";
2) Instruções pertencentes ao bloco estão no mesmo nível de indentação entre si e indentadas em 4 espaços[4] relativamente à instrução que introduz o bloco.

Para compreender as estruturas de seleções aninhadas e a necessidade de usá-las, veremos a Codificação 5.19, um programa que solicita a idade do usuário e, de acordo com a resposta, realiza outras perguntas para descobrir se ele pode dirigir. Tente entender os possíveis caminhos dentro deste código e desenhe um mapa onde cada `if` é uma bifurcação na "estrada" do fluxo de execução e cada instrução é uma "cidade" neste mapa. Se um `if` não possui um comando `else` associado, podemos interpretá-lo como uma bifurcação cujo lado `else` não contém nenhuma "cidade".

```
idade = int(input('Qual a sua idade? '))                        #1
if idade >= 18:                                                 #2
    cnh = input('Você tem CNH? (s/n)')                          #3
    if cnh == 's':                                              #4
        cnh_valida = input('Ela está válida? (s/n)')            #5
        if cnh_valida == 's':                                   #6
            print('Você pode dirigir.')                        #7
        else:
            print('Você precisa renovar sua CNH.')             #8
    else:
        print('Você precisa tirar a CNH para dirigir.')        #9
else:
    print('Você ainda não pode dirigir.')                       #10
    tempo_falta = 18 - idade                                    #11
    if tempo_falta <= 2:                                        #12
        print('Falta pouco tempo para você poder dirigir.') #13
    else:
        print('Demorará para você poder dirigir.')             #14
    print('Mas você pode dirigir um kart se quiser.')          #15
print('Fim!')                                                   #16
```

Codificação 5.19: Programa com estruturas de seleção aninhadas.

[4] O Python aceita uma quantidade qualquer de espaços, desde que todas as instruções do mesmo bloco tenham o mesmo número de espaços, mas o padrão recomendado é de 4 espaços (Rossum et. al., 2013).

Após desenhar, veja se consegue dizer quando cada instrução é executada, isto é, qual o conjunto de entradas que precisam ser dadas para que o fluxo de execução passe por aquela instrução em particular. Em seguida, compare seu mapa com o da Figura 5.2.

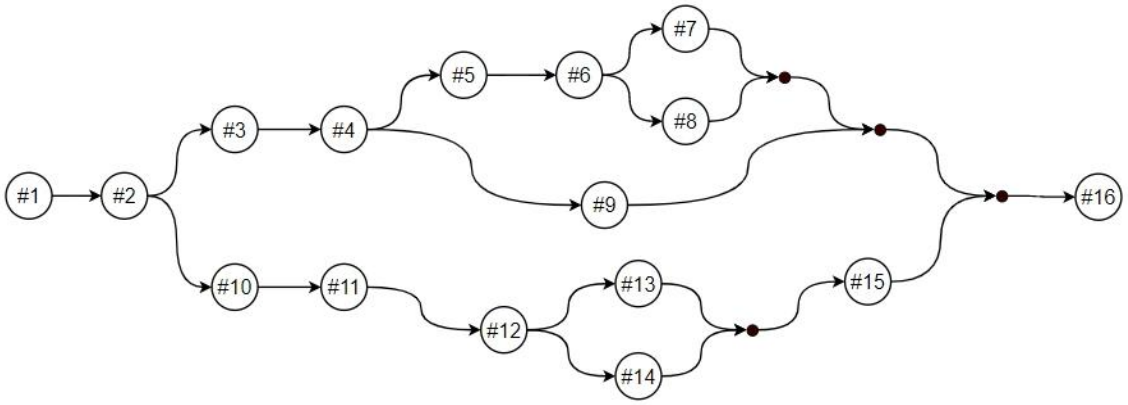

Figura 5.2: Mapa do fluxo de execução da Codificação 5.19. Fonte: Elaborado pelo autor.

Uma forma de entender um trecho de código mais complexo é com a construção destes mapas ou de fluxogramas. Podemos pensar inicialmente em um mapa visto de longe, com apenas 2 caminhos, como mostra a Figura 5.3, e conforme aumentamos o "zoom" neste mapa, caminhos menores surgirão dentro dos caminhos maiores, e assim sucessivamente para cada novo bloco interno, como podemos ver na Figura 5.4. Dessa forma, analisamos o código a partir do bloco mais externo para o mais interno.

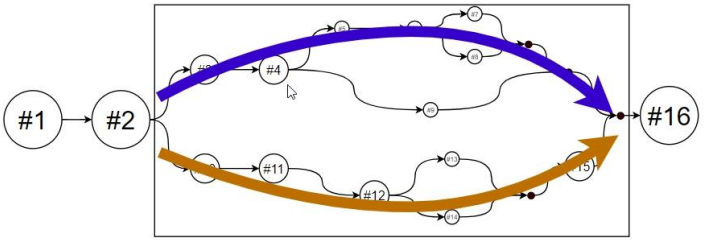

Figura 5.3: Visão da estrutura de seleção mais externa. Fonte: Elaborado pelo autor.

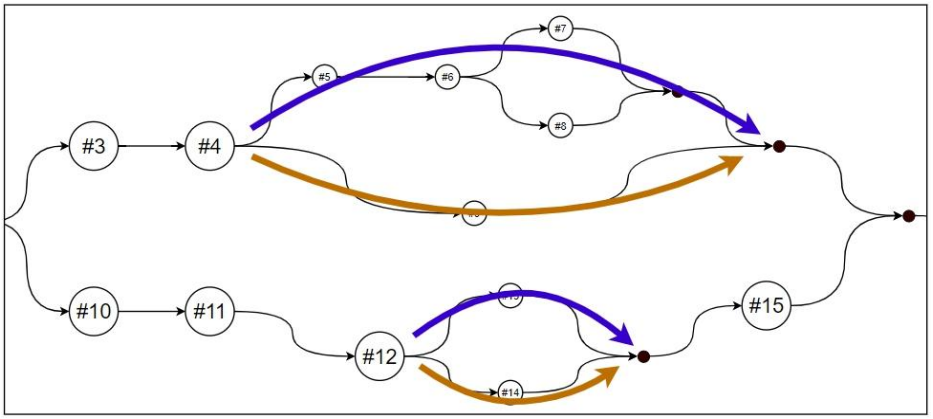

Figura 5.4: Visão das estruturas de seleção no 2º nível. Fonte: Elaborado pelo autor.

Lembre-se de testar esse exemplo também no Python Tutor, para visualizar passo a passo as instruções que são executadas, em função das entradas fornecidas.

Você possivelmente deve ter concluído que em estruturas de seleção aninhadas a seleção mais externa funciona como um filtro para suas seleções internas, ou seja, para que o fluxo de execução do programa atinja o ponto de uma seleção interna, necessariamente já deve ter passado pela decisão da seleção mais externa.

5.5. Estruturas de seleção encadeadas

As estruturas de seleção encadeadas são um subtipo das estruturas aninhadas. Dizemos que uma estrutura de seleção A está encadeada em uma estrutura de seleção B, quando A está dentro do bloco `else` de B, e A é a única instrução desse bloco.

Quando isso ocorre, dizemos que as seleções estão encadeadas entre si, isto é, cada seleção do encadeamento só é analisada se todas as anteriores resultarem em `False`, e no momento que uma das condições é avaliada `True`, seu bloco de código é executado e as demais condições subsequentes, se existirem, serão ignoradas.

Vejamos outro exemplo com um programa que lê a nota de um aluno em uma escala de 0 a 10, e converte-a para uma escala usando letras, como mostra a Figura 5.5.

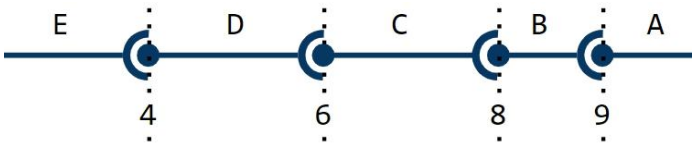

Figura 5.5: Intervalos de classificação das notas. Fonte: Elaborado pelo autor.

O programa correspondente está na Codificação 5.20. Observe que:

- A primeira bifurcação no fluxo de execução ocorre no `if` da instrução `#2`. Caso a condição `nota >= 9` resulte `True` o fluxo será direcionado para `#3`, caso contrário será direcionado para a primeira instrução do bloco `else` correspondente, ou seja, a instrução `#4`, garantindo que nota é menor que 9;
- O bloco do `else` só é executado quando a condição do `if` correspondente resultar `False`. No exemplo anterior, todas as instruções a partir da `#4` até `#10` fazem parte do bloco do primeiro `else`, evidente pela indentação, portanto, se `#3` é executada, a próxima instrução deste caminho será `#11`;
- Caso a nota seja menor do que 9, então a instrução `#4` será executada, e haverá mais uma divisão de caminhos, entre notas maiores ou iguais a 8 e a negação disso, notas menores que 8. Aqui, vale ressaltar que não é necessário verificar se `nota < 9`, pois ao executar `#4`, sabemos que a condição da instrução `#2` resultou em `False` e portanto o valor de `nota` é obrigatoriamente menor que 9.

```
nota = float(input('Qual a nota? '))  #1

if nota >= 9:  #2
    letra = 'A'  #3
else:
    if nota >= 8:  #4
        letra = 'B'  #5
    else:
        if nota >= 6:  #6
            letra = 'C'  #7
        else:
            if nota >= 4:  #8
                letra = 'D'  #9
            else:
                letra = 'E'  #10

print(f'Sua letra é: {letra}')  #11
```

Codificação 5.20: Programa com estruturas de seleção encadeadas.

5.5.1. O comando elif

O comando `elif` é uma fusão dos comandos `else` e `if` quando se encontram na situação de uma estrutura de seleção encadeada, como acabamos de ver. Ele é mais um exemplo de "açúcar sintático", introduzido na linguagem para facilitar a escrita e, principalmente, a leitura do código.

A Codificação 5.21 contém lado a lado a sintaxe de uma estrutura de seleção encadeada sem `elif` (à esquerda) e com `elif` (à direita). Os dois códigos apresentam os mesmos fluxos de execução, veja na Figura 5.6, a única diferença está na sintaxe.

```
if <condição 1>:       #1
    <bloco 1>
else:                  #2
    if <condição 2>:   #3
        <bloco 2>
    else:              #4
        <bloco 3>
```

```
if <condição 1>:     #1
    <bloco 1>
elif <condição 2>: #2 + #3
    <bloco 2>
else:                #4
    <bloco 3>
```

Codificação 5.21: Estrutura de seleção encadeada sem `elif` (à esq.) e com `elif` (à dir.).

Na Codificação 5.21, note que ao unirmos `#2` com `#3` (`else` e `if`) formarmos o `elif`, e as consequências dessa união são:

- O comando `elif` começa com um `else`, portanto é sempre um complemento de um `if` ou outro `elif`, e não pode existir isoladamente;
- O comando `elif` termina com um `if`, portanto podemos complementá-lo com um `else` ou outro `elif` na sequência, e assim por diante sem quantidade limite;
- O comando `else` sempre encerra uma estrutura de seleção, seja ele usado após um `if` ou `elif`;
- Esse encadeamento garante que o bloco executado será aquele que estiver dentro da primeira estrutura de seleção que a condição resultar `True`, e que as seleções posteriores no encadeamento serão puladas.

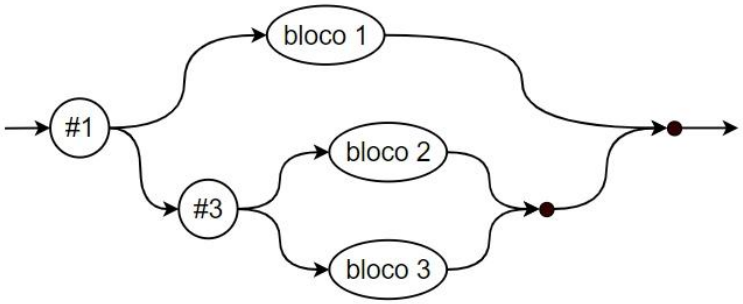

Figura 5.6: Mapa de caminho possíveis da Codificação 5.21. Fonte: Elaborado pelo autor.

Podemos então aplicar este novo comando à Codificação 5.20 gerando a Codificação 5.22. Observe como o código permanece com apenas dois níveis de indentação, mais conciso, simples e com melhor legibilidade.

```
nota = float(input('Qual a nota? '))

if nota >= 9:
    letra = 'A'
elif nota >= 8:
    letra = 'B'
elif nota >= 6:
    letra = 'C'
elif nota >= 4:
    letra = 'D'
else:
    letra = 'E'

print(f'Sua letra é: {letra}')
```

Codificação 5.22: Estrutura de seleção encadeada com `elif`.

VAMOS PRATICAR!

1) Crie um programa que solicite ao usuário um número de 1 à 7 e exiba o dia da semana correspondente. Assuma que a semana começa no domingo (1) e termina no sábado (7). Use apenas seleção simples, ou seja, sem `else` nem `elif`.

2) Refaça o exercício anterior, agora utilizando a seleção encadeada, mas sem usar o comando `elif`, de modo que seja observada a importância do alinhamento correto de indentação entre os diversos comandos `if` e `else`.

3) Refaça o exercício anterior, agora utilizando `elif`.

4) Com o Python Tutor, compare o fluxo de execução dos códigos dos exercícios anteriores.

Bibliografia e referências

BADER, D. Python String Formatting Best Practices. **Real Python**, 2018. Disponível em: <https://realpython.com/python-string-formatting/>. Acesso em: 26 jan. 2021.

JABLONSKI, J. Python 3's f-Strings: An Improved String Formatting Syntax (Guide). **Real Python**, 2018. Disponível em: <https://realpython.com/python-f-strings/>. Acesso em: 26 jan. 2021.

PYTHON SOFTWARE FOUNDATION. **Common string operations**. 2021. Disponível em: <https://docs.python.org/3/library/string.html>. Acesso em: 26 jan. 2021.

PYTHON SOFTWARE FOUNDATION. **Input and output**. 2021. Disponível em: <https://docs.python.org/3/tutorial/inputoutput.html>. Acesso em: 26 jan. 2021.

ROSSUM, G. V., WARSAW, B., COGHLAN, N. **Style Guide for Python Code**. 2013. Disponível em: <https://www.python.org/dev/peps/pep-0008/>. Acesso em: 27 jan. 2021.

STURTZ, J. Basic Input, Output, and String Formatting in Python. **Real Python**, 2019. Disponível em: <https://realpython.com/python-input-output/>. Acesso em: 26 jan. 2021.