4

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

4

Operadores e expressões relacionais e lógicas, estruturas de seleção simples e composta

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

Resumo

Nesta aula os objetivos são: (I) apresentar as estruturas de controle de fluxo de execução; (II) conhecer as expressões relacionais e lógicas, assim como seus operadores; (III) entender a associatividade dos operadores relacionais; (IV) compreender o que são expressões equivalentes e complementares; (V) conceituar a avaliação de curto-circuito; (VI) construir estruturas de seleção simples e composta.

4.1. Motivação

Os primeiros computadores eletrônicos foram chamados de "cérebros eletrônicos", causando a impressão de que poderiam pensar como humanos. Embora sejam máquinas extremamente complexas, computadores simplesmente executam aquilo que lhes é instruído, sem autonomia para tomar decisões por conta própria… ao menos por enquanto. Assim, não há mais inteligência em um computador do que há nas instruções que lhe são dadas, e sua vantagem está em conseguir executar uma sequência de instruções de modo mais confiável e rápido do que uma pessoa poderia fazer.

Com estruturas sequenciais de controle de fluxo, já conseguimos resolver diversos problemas simples, mas muitas vezes precisamos escolher se iremos ou não executar determinadas instruções com base em dados que só serão conhecidos em tempo de execução, muitas vezes dependentes de entradas fornecidas pelo usuário do programa. Para isso, veremos as estruturas de seleção, que permitirão que nossos programas reajam de acordo com decisões pré-determinadas e com base no que ocorrerá durante seu fluxo de execução.

4.2. Estruturas de controle

A ordem na qual as instruções de um programa são executadas pode ser chamada de *fluxo de execução*, sendo assim, uma estrutura de controle do fluxo de execução é uma forma de controlar em qual ordem as instruções serão executadas.

Fundamentalmente existem três tipos de estruturas para controlar o fluxo de execução de um programa, cuja representação pode ser vista na Figura 4.1. São elas:

I. Estrutura sequencial;
II. Estrutura de seleção ou condicional; e
III. Estrutura de repetição ou iterativa, também chamada de laço.

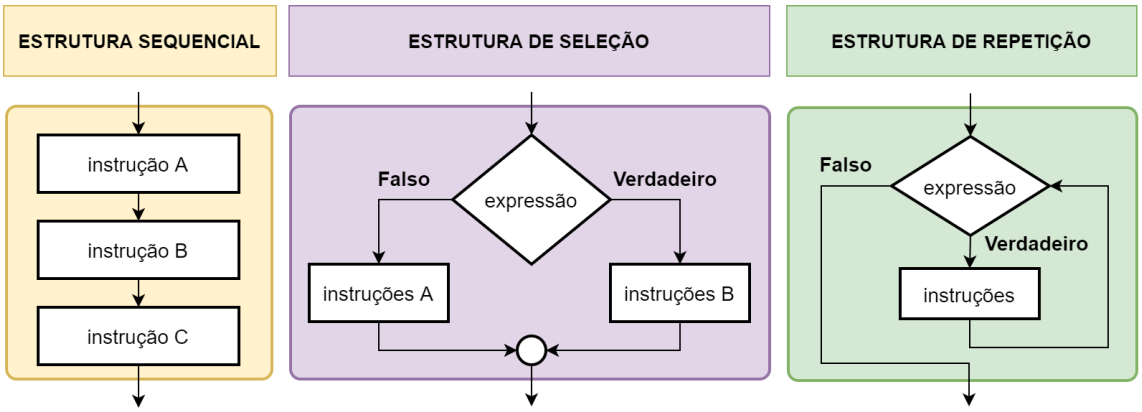

Figura 4.1: Estruturas de controle de fluxo básicas. Fonte: Elaborado pelo autor.

Por enquanto, os problemas que lidamos requisitaram apenas soluções com estrutura de controle sequencial, em que a ordem que as instruções são executadas é idêntica a ordem que foram escritas. Agora, aprenderemos o segundo tipo de estrutura de controle, a estrutura de seleção ou estrutura condicional, na qual a execução de uma sequência de instruções ficará condicionada ao resultado da avaliação de uma expressão.

Com essa estrutura de controle, poderemos criar programas que decidam, em tempo de execução, quais instruções serão executadas com base em uma expressão que resulte em um *valor lógico* (também conhecido como *valor booleano*), ou seja, *verdadeiro* ou *falso*. Portanto, antes de tratarmos da estrutura de seleção em si, veremos como criar e avaliar expressões que resultam valores lógicos.

4.3. Expressões relacionais e lógicas

Anteriormente lidamos com expressões aritméticas, nas quais existem apenas operadores aritméticos e operandos numéricos, quando essas expressões são avaliadas o resultado é um número. Agora estudaremos expressões relacionais e lógicas, em que a avaliação resultará em um valor verdadeiro ou falso, representados em Python pelas constantes `True` e `False`, respectivamente.

Expressões relacionais são aquelas em que dois ou mais operandos são comparados para verificar se uma determinada relação entre eles é verdadeira ou falsa. A relação é indicada por meio de operadores relacionais (maior, menor, diferente, etc.) e o resultado é um valor booleano. Um exemplo pode ser visto na Codificação 4.1.

```
>>> idade = 23
>>> pode_tirar_cnh = idade >= 18  # Resulta True
```

Codificação 4.1: Exemplo de expressão relacional.

Na Codificação 4.1, `idade` é comparada com o inteiro `18`, o resultado da comparação será `True` se o operando à esquerda for maior ou igual (`>=`) ao da direta, ou resultará `False` caso contrário. Note que este exemplo poderia ser aplicado à condição básica para um brasileiro, em condições normais, tentar obter sua carteira de motorista.

Já expressões lógicas são aquelas em que os operandos são valores booleanos e usamos os operadores lógicos (E, OU e NÃO), análogos aos da lógica matemática, para obter um resultado também booleano, indicando se a expressão como um todo é verdadeira ou falsa. Veja na Codificação 4.2 uma continuação da Codificação 4.1.

```
>>> aprovado_detran = True
>>> pode_dirigir = pode_tirar_cnh and aprovado_detran
```

Codificação 4.2: Exemplo de expressão lógica.

No Brasil, para estar habilitado a dirigir, não basta ter 18 anos, é preciso também ser aprovado pelo Detran, por isso, na Codificação 4.2, `pode_dirigir` receberá `True` apenas se `pode_tirar_cnh` e (`and`) `aprovado_detran` estiverem com o valor `True`.

Portanto, para compreender as possibilidades de relações verificáveis entre os operandos nas expressões relacionais e lógicas, é necessário conhecer os operadores relacionais e lógicos disponíveis na linguagem de programação Python.

VOCÊ SABIA?

Os valores lógicos também são conhecidos como valores *booleanos* por causa de George Boole, um matemático e filósofo britânico, criador da álgebra booleana no século XIX, fundamental para o desenvolvimento da computação moderna. Veja mais: Como matemático inventou há mais de 150 anos a fórmula de buscas usada pelo Google

4.3.1. Operadores relacionais

Os operadores relacionais em Python são mostrados na Tabela 4.1 e, evidentemente, são todos binários, pois é necessário relacionar um operando a outro.

Esses operadores relacionais também estão presentes na matemática, mas vale lembrar que a interpretação em Python difere daquelas em equações e inequações

matemáticas. Na matemática tais sinais são uma afirmação sobre a (in)equação, se dizemos que $2x + 2 = x + 5$, sabemos que ambos os lados da equação devem ter o mesmo resultado e podemos então calcular o valor de x.

Tabela 4.1: Operadores relacionais em Python.

Operador	Descrição	Exemplos
`==`	igual a	`5 == 3  # False` `8 == 8  # True`
`!=`	diferente de	`5 != 3  # True` `8 != 8  # False`
`>`	maior que	`5 > 3   # True` `8 > 8   # False`
`>=`	maior ou igual a	`5 >= 3  # True` `8 >= 8  # True`
`<`	menor que	`5 < 3   # False` `8 < 8   # False`
`<=`	menor ou igual a	`5 <= 3  # False` `8 <= 8  # True`

Sabemos que em Python o sinal de igual simboliza o operador de atribuição simples (`=`), indicando que a variável à esquerda do operador receberá o valor resultante da avaliação da expressão à direita do operador, consequentemente a equação mostrada no parágrafo anterior é inválida em Python.

Já com os operadores relacionais, é como se fizéssemos uma pergunta em relação a ambos os operandos, como: “eles são iguais?”, “são diferentes?”, “o da esquerda é maior que o da direita?”, “o da esquerda é menor ou igual ao da direita?” e assim sucessivamente, obtendo sempre como resposta um valor verdadeiro ou falso, que, como já mencionado, são representados em Python por `True` e `False`.

Em Python todos os operadores relacionais possuem a mesma precedência entre si e são não associativos, porém possuem uma interpretação especial quando encadeados, algo que veremos adiante.

4.3.2. Operadores lógicos

Os operadores lógicos permitem a combinação de comparações ou valores booleanos, possibilitando a construção de expressões mais complexas. Consideraremos três operadores lógicos:

I. *NÃO:* operador unário que inverte o valor do operando associado, ou seja, é a negação do operando. Negar `True` resulta em `False` e negar `False` resulta em `True`;
II. *E:* operador binário que resulta `True` apenas se ambos os operandos forem `True`, e `False` caso contrário. Portanto, basta um operando `False` para que o resultado da expressão resulte em `False`;
III. *OU:* operador binário que resulta `True` se pelo menos um dos operandos for `True`, e `False` caso contrário. Portanto, resulta em `False` apenas se ambos operandos forem `False`.

Podemos resumir o funcionamento dos operadores lógicos em uma *tabela verdade* para cada operador, como mostrado na Tabela 4.2.

Tabela 4.2: Tabela verdade dos operadores lógicos.

a	NÃO a
True	False
False	True

a	b	a E b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a OU b
True	True	True
True	False	True
False	True	True
False	False	False

A Tabela 4.3 contém as palavras reservadas em Python correspondentes a cada operador lógico mostrado até agora.

Tabela 4.3: Operadores lógicos em Python.

Significado	Python
NÃO	`not`
E	`and`
OU	`or`

VAMOS PRATICAR!

1) Crie na *Shell* do Python as variáveis a seguir: a = 4; b = 10; c = 50; d = 1; e = 5. Em seguida faça a avaliação das seguintes expressões (tente antecipar o resultado da *Shell*):

i) `a == c`
ii) `a < b`
iii) `d > b`
iv) `c != e`
v) `a == b`
vi) `c < d`
vii) `b > a`
viii) `c <= e`
ix) `c <= c`
x) `c <= e`
xi) `d != a`
xii) `e != e`

2) Avalie mentalmente as expressões a seguir e confira o resultado digitando-as na *Shell*:

```
>>> True and False
>>> True or False
>>> not True or True
>>> not (True or True)
>>> True and not False
```

```
>>> (10 < 0) and (10 > 2)
>>> (10 < 0) or (10 > 2)
>>> not (3 != 0) or (8 > 5)
>>> not (3 != 0 or 8 > 5)
>>> not not True
```

Talvez tenha sentido dificuldade em avaliar algumas expressões do exercício 2, principalmente por não ter certeza sobre qual parte seria avaliada primeiro. Para resolver isso precisamos saber as regras de precedência e associatividade desses operadores.

4.3.3. Precedência e associatividade

A Tabela 4.4 traz os operadores relacionais e lógicos em ordem decrescente de prioridade, juntamente com suas associatividades.

Tabela 4.4: Precedência e associatividade dos operadores relacionais e lógicos.

Operador	Descrição	Associatividade
`==`, `!=`, `>`, `>=`, `<`, `<=`	Operadores relacionais.	Não associativos
`not`	Negação lógica.	À esquerda
`and`	E lógico.	
`or`	OU lógico.	

Precisamos entender o que ocorre ao digitarmos expressões com operadores relacionais encadeados, afinal são operadores não associativos. Veja a Codificação 4.3.

```
>>> 3 < 10 < 23  # Resulta em True
```

Codificação 4.3: Exemplo de expressão com encadeamento de operadores relacionais.

Se houvesse associatividade dos operadores relacionais em Python, poderíamos ter uma entre duas possíveis interpretações:

a) Com associatividade à esquerda: `3 < 10 < 23` ⇨ `True < 23` ⇨ `?`
b) Com associatividade à direita: `3 < 10 < 23` ⇨ `3 < True` ⇨ `?`

Podemos observar que a interpretação da comparação de um valor booleano com um valor numérico, ainda que possível em Python (teste na *Shell* a comparação de `True` e `False` com valores numéricos), não faz sentido neste contexto. Outra forma de interpretação seria "o número 10 está entre os números 3 e 23?". Também poderíamos interpretá-la como de costume na matemática, "3 é menor que 10 que é menor que 23?"

Muitas linguagens de programação não permitem o encadeamento de operadores relacionais, evitando o problema mencionado. Porém Python, visando facilidade de

leitura e escrita de código, permite o encadeamento, aproximando o código à notação matemática. Neste caso, o interpretador traduz a expressão como um encadeamento de expressões binárias, repetindo[1] os operandos entre os operadores relacionais e unindo-os com `and`. Vejamos passo a passo como Python avalia a expressão da Codificação 4.3.

1) `3 < 10 < 23` ⇨ `3 < 10 and 10 < 23`
2) `3 < 10 and 10 < 23` ⇨ `True and 10 < 23`
3) `True and 10 < 23` ⇨ `True and True`
4) `True and True` ⇨ `True`

É comum que expressões relacionais e lógicas incluam também operadores aritméticos, então é importante saber a precedência destes operadores não apenas entre si, mas com relação aos demais tipos de operadores. A Tabela 4.5 mostra a ordem em que os operadores são resolvidos pelo Python, de acordo com sua precedência, sendo o operador de maior prioridade resolvido primeiro.

Tabela 4.5: Prioridade dos operadores aritméticos, relacionais e lógicos.

Ordem de resolução	Operador	Descrição	Associatividade
1°	`**`	Exponenciação.	À direita
2°	`+`, `-` (unários)	Identidade e negação.	À esquerda
3°	`*`, `/`, `//`, `%`	Multiplicação, divisão real, divisão inteira e resto da divisão.	
4°	`+`, `-` (binários)	Adição e subtração.	
5°	`==`, `!=`, `>`, `>=`, `<`, `<=`	Operadores relacionais.	Não associativos
6°	`not`	Negação lógica.	À esquerda
7°	`and`	E lógico.	
8°	`or`	OU lógico.	

Dentre os operadores vistos até o momento, a atribuição, simples ou composta, tem a menor prioridade independentemente da instrução. Isso é necessário para que a expressão à direita do operador de atribuição possa ser completamente avaliada e o resultado atribuído à variável à esquerda.

[1] O operando entre os dois operadores relacionais só será avaliado uma vez, mesmo que internamente a expressão relacional encadeada seja transformada em duas expressões menores conectadas por um `and`. O valor da avaliação é guardado e usado nas duas expressões. Isso é importante pois garante economia de recursos computacionais e evita problemas quando, por exemplo, o operando é uma chamada a uma função ou uma expressão mais complexa, em que uma dupla avaliação poderia modificar o operando.

4.3.4. Expressões equivalentes e complementares

Duas expressões são equivalentes quando possuem exatamente o mesmo significado, apesar de escritas de forma diferente. Por exemplo, na Codificação 4.1, fizemos a comparação da idade do usuário com o número 18, verificando se a idade era maior ou igual a 18, mas poderíamos verificar se 18 era menor ou igual à idade, e chegaríamos ao mesmo resultado em ambas as expressões, conforme a Codificação 4.4.

```
>>> idade = 23
>>> idade >= 18
True
>>> 18 <= idade
True
```

Codificação 4.4: Exemplo de expressões equivalentes.

Duas expressões são complementares quando uma é a negação da outra, ou seja, juntas cobrem todo o espaço possível para os valores envolvidos, sem sobreposição. Por exemplo, a expressão `idade < 18` é complementar à `idade >= 18`, e vice-versa, como ilustrado na Figura 4.2.

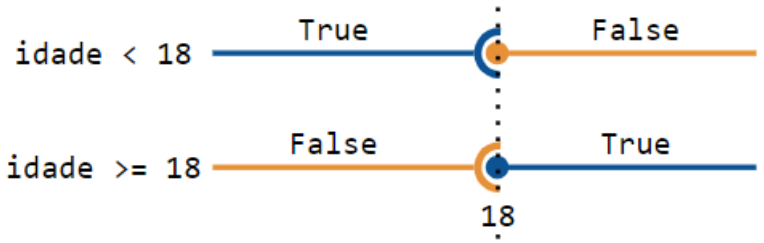

Figura 4.2: Representação de expressões complementares na reta dos números reais.
Fonte: Elaborado pelo autor.

Essas expressões, quando juntas, cobrem toda a reta dos números reais sem nenhuma sobreposição (observe que `idade < 18` não inclui o número 18), isso significa que para cada possível valor de `idade`, se uma das expressões for verdadeira, a outra será obrigatoriamente falsa. Veja outros exemplos na Codificação 4.5.

```
>>> 'Maria' != 'Megan'      # Expressão A
>>> 'Maria' == 'Megan'      # Expressão B (complementar à A)
>>> not('Maria' == 'Megan') # Expressão C (complementar à B)
>>> not('Maria' != 'Megan') # Expressão D (complementar à A)
```

Codificação 4.5: Exemplo de expressões complementares.

É comum o surgimento de dúvidas ao buscar expressões complementares de expressões que contenham, além de operadores relacionais, operadores lógicos. Veja na Codificação 4.6 um exemplo em que pessoas acima de 80 anos não possam obter carteira de motorista.

```
>>> pode_tirar_cnh = idade >= 18 and idade <= 80
```

Codificação 4.6: Expressão com operadores relacionais e lógicos.

Podemos reescrever a expressão da Codificação 4.6 com uma expressão equivalente `18 <= idade and idade <= 80` ou também `18 <= idade <= 80`, no entanto, prosseguiremos com a versão inicial, que inclui o operador lógico "E", cuja representação na reta dos números reais pode ser vista na Figura 4.3.

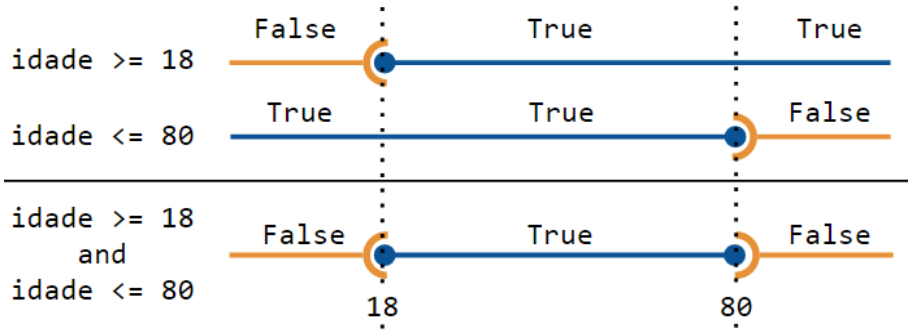

Figura 4.3: Representação da junção de duas expressões relacionais com o operador lógico E. Fonte: Elaborado pelo autor.

Se quisermos encontrar a expressão complementar a esta, ou seja, a expressão que resulte `True` para aqueles que **não** podem obter carta de motorista (negação da expressão inicial), podemos começar negando os valores booleanos na figura e a partir dela, tentar chegar em uma expressão. Veja o resultado na Figura 4.4.

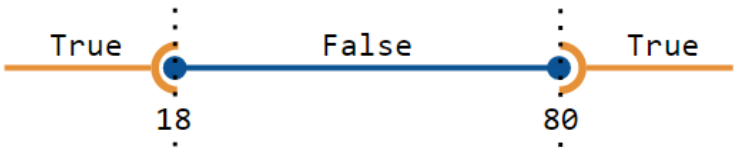

Figura 4.4: Representação da expressão complementar à expressão dada na Figura 4.3. Fonte: Elaborado pelo autor.

Agora, podemos escrever as duas expressões relacionais que resultarão `True`:

1) `idade < 18`
2) `idade > 80`

Observamos que as expressões não podem ser verdadeiras ao mesmo tempo, pois não existe nenhum número que seja simultaneamente menor que 18 e maior que 80, portanto basta que a primeira OU a segunda resulte em verdadeiro. Com isso, chegamos de maneira intuitiva à seguinte expressão: `idade < 18 or idade > 80`.

Podemos concluir que, ao negar a expressão inicial, obtemos sua expressão complementar e, mais importante, que o operador lógico "E" é substituído pelo "OU" e os operadores relacionais "maior ou igual" e "menor ou igual" são substituídos por, respectivamente, "menor" e "maior". Algo que pode ser visto no seguinte resumo:

a) Expressão inicial: `idade >= 18 and idade <= 80`
b) Expressões complementares (em ordem crescente de simplificação):
    1) `not (idade >= 18 and idade <= 80)`
    2) `not (idade >= 18) or not (idade <= 80)`
    3) `idade < 18 or idade > 80`

Se negarmos qualquer uma das expressões complementares, retornaremos à expressão inicial, fazendo as substituições inversas. Neste exemplo, qualquer uma das três expressões é considerada complementar à expressão inicial, no entanto, a última é a mais simples de ser lida por um humano, por ser mais direta usando menos operadores, e menos custosa de ser executada por um computador, por ter menos operações.

VAMOS LER!

A negação (complementar) de conjunções (E lógico) e disjunções (OU lógico) foi um assunto tratado por Augustus De Morgan, um importante matemático e lógico britânico, no século XIX e que originou às "Leis De Morgan", amplamente usadas para resolver problemas de lógica. Veja mais: https://pt.wikipedia.org/wiki/Teoremas_de_De_Morgan

4.3.5. Avaliação de curto-circuito

Algumas linguagens de programação definem operadores lógicos com a característica de *avaliação de curto-circuito*, também chamada de *avaliação mínima*. Operadores lógicos com avaliação de curto-circuito estabelecem que seu segundo operando só será avaliado caso a avaliação do primeiro não seja suficiente para concluir o valor da expressão completa.

Em Python, os operadores lógicos `and` e `or` são operadores com avaliação de curto-circuito. Veja o comportamento de cada um deles:

a) **`and`**: o segundo operando só será avaliado caso o primeiro resulte em `True`, pois em uma expressão com `and`, o resultado só pode ser antecipado se o primeiro operando resultar em `False`, afinal neste caso não importaria o valor do segundo operando, a expressão sempre resultaria em `False`;
b) **`or`**: o segundo operando só será avaliado caso o primeiro resulte em `False`, pois em uma expressão com `or`, o resultado só pode ser antecipado se o primeiro operando resultar em `True`, afinal neste caso não importaria o valor do segundo operando, a expressão sempre resultaria em `True`.

Para exemplificar a consequência e utilidade da avaliação mínima, veja a Codificação 4.7 e a Codificação 4.8, em que essa característica é essencial para o correto funcionamento dos programas.

```
# Crie um programa que receba dois números naturais e exiba uma
# mensagem indicando se o primeiro é divisível pelo segundo.

a = int(input('Primeiro: '))
b = int(input('Segundo: '))
print(f'{a} é divisível por {b}: {b != 0 and a % b == 0}')
```

Codificação 4.7: Programa que usufrui do curto-circuito do operador `and`.

Note na Codificação 4.7 que, sem a avaliação de curto-circuito, caso o usuário inserisse o valor zero para `b`, o programa geraria um erro pela tentativa de divisão por zero. Porém, `a % b == 0` só será avaliada após `b != 0` resultar `True`, logo o erro não ocorrerá. Como teste, inverta a ordem das sub-expressões e insira valor zero para `b`.

```
# Crie um programa que receba dois números naturais e exiba uma
# mensagem indicando se o primeiro não é divisível pelo segundo.

a = int(input('Primeiro: '))
b = int(input('Segundo: '))
print(f'{a} não é divisível por {b}: {b == 0 or a % b != 0}')
```

Codificação 4.8: Programa que usufrui do curto-circuito do operador `or`.

Assim como na Codificação 4.7, na Codificação 4.8, sem a avaliação de curto-circuito, se o usuário inserisse o valor zero para `b`, o programa geraria um erro. Porém, como `a % b != 0` só será avaliada após `b == 0` resultar `False`, o erro não ocorrerá. Novamente, inverta a ordem das sub-expressões e insira valor zero para `b`.

4.4. Estruturas de seleção ou estruturas condicionais

São as estruturas usadas para selecionar se um trecho de código será executado com base na avaliação de uma expressão, dizemos que a execução do trecho de código está condicionada ao resultado desta expressão, por isso a *expressão de decisão* também pode ser chamada de *condição*. Inicialmente, abordaremos dois tipos de estruturas de seleção, que também estão ilustradas na Figura 4.5:

1) *Seleção simples:* usada quando condicionamos um bloco de código a ser executado apenas quando a condição resultar `True`. Logo, quando a condição resultar em `False` o bloco condicionado será pulado/ignorado;
2) *Seleção composta:* usada quando há dois blocos de código condicionados, sendo que um deles será executado apenas quando a condição resultar `True` e o outro apenas quando a condição resultar `False`. Portanto, ao executar um bloco, seja qual for, o outro obrigatoriamente será pulado/ignorado.

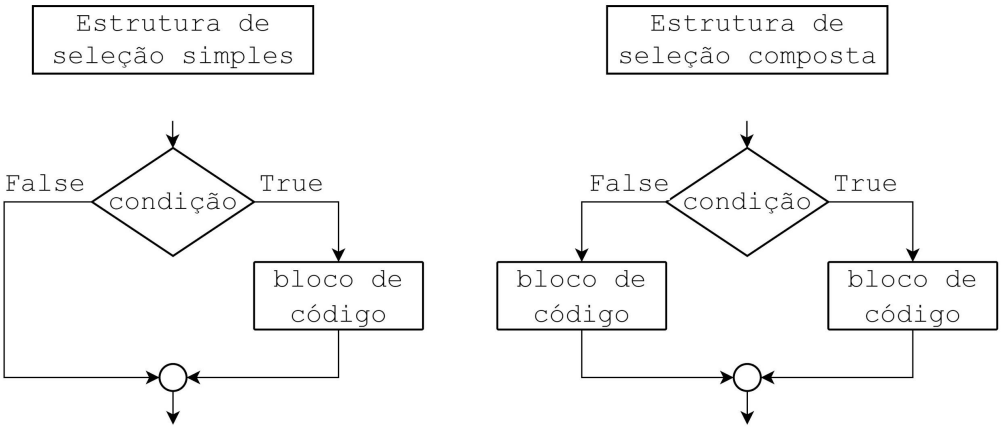

Figura 4.5: Fluxograma das estruturas de seleção simples e composta.
Fonte: Elaborado pelo autor.

4.4.1. Estruturas de seleção simples

As estruturas de seleção simples são escritas em Python com o uso do comando `if`, uma das palavras reservadas do Python, como na Codificação 4.9.

```
if <condição>:
    <bloco de código>
```

Codificação 4.9: Estrutura da seleção simples `if`.

A definição de um *bloco de código* em Python é dada por dois fatores:

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[2] relativamente à instrução que introduz o bloco.

A indentação é o espaçamento de uma instrução em relação à sua margem esquerda, recomenda-se o uso de caracteres de espaços e não caracteres de tabulação. Na maioria dos editores de código-fonte para Python, incluindo o IDLE, a configuração padrão insere automaticamente quatro espaços ao pressionarmos a tecla `[TAB]`.

Python identifica que o bloco de código pertencente à estrutura de seleção terminou ao encontrar a primeira instrução, subsequente ao início do bloco, que regride em nível de indentação, ou seja, a primeira instrução deslocada de volta para a esquerda. Caso haja instruções com alguma indentação inesperada, incluindo um bloco vazio, isso é marcado como erro de sintaxe, impossibilitando a execução do código até a correção.

Como exemplo de programa que utiliza uma estrutura de seleção simples, insira no editor do IDLE a Codificação 4.10 e teste seu funcionamento.

[2] O Python aceita quantidade qualquer de espaços, desde que todas as instruções do mesmo bloco tenham o mesmo número de espaços, mas o valor padrão recomendado pela comunidade é de 4 espaços.

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

if idade >= 18:
    # Bloco executado apenas quando a condição é verdadeira
    print('Você pode ter uma CNH.')
    print('Desejamos boa sorte!')

print('Fim')  # executado sempre
```

Codificação 4.10: Exemplo de programa com uma estrutura de seleção simples.

4.4.2. Estruturas de seleção composta

As estruturas de seleção composta são escritas em Python de forma semelhante às estruturas de seleção simples, porém adicionando o comando `else`, como um complemento ao `if`, conforme a codificação 4.11.

```
if <condição>:
    <bloco de código 1>
else:
    <bloco de código 2>
```

Codificação 4.11: Estrutura da seleção composta `if...else`.

Após o comando `else` não é preciso (nem permitido) colocar uma segunda condição, pois será considerada automaticamente a condição complementar àquela do `if`. Consequentemente, quando a condição do `if` resultar em `True`, o bloco 1 será executado e o bloco 2 ignorado e, quando a condição do `if` resultar em `False`, ocorrerá o oposto, pois o bloco 2 será executado e o bloco 1 ignorado.

Como `else` é um complemento do `if`, jamais deve ser escrito isoladamente. Repare que Python identifica o `if` correspondente ao `else` por meio da indentação, a regra é simples: cada `else` corresponde ao `if` mais próximo que o antecede no mesmo nível de indentação. Só pode existir um `else` por `if`.

Como exemplo de programa que utiliza uma estrutura de seleção composta, insira no editor do IDLE a Codificação 4.12 e teste seu funcionamento.

Um erro frequente com iniciantes em programação é assumir que todo `if` deve ser complementado com um `else`. Não faz sentido! Quando não existem dois blocos de instruções mutuamente exclusivos (ao executar um bloco o outro deve necessariamente ser ignorado), usa-se seleção simples que, como já explicado, não tem `else`. Veja um exemplo deste erro de lógica na Codificação 4.13.

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

if idade >= 18:
    # Bloco executado quando a condição resulta verdadeiro
    print('Você pode ter uma CNH.')
    print('Desejamos boa sorte!')
else:
    # Bloco executado quando a condição resulta falso
    print('Você ainda não completou 18 anos, ', end='')
    print('portanto não pode ter uma CNH.')

print('Fim')  # executado sempre
```

Codificação 4.12: Exemplo de programa com uma estrutura de seleção composta.

```
# Crie um programa que receba como entrada o valor de um produto
# e a quantidade comprada. O programa concederá 10% de desconto
# para compras com total maior ou igual à R$ 100,00.

valor = float(input('Valor: '))
quantidade = int(input('Quantidade: '))
total = valor * quantidade

if total >= 100.0:
    total = total * 0.9
else:
    total = total

print('Total:', total)
```

Codificação 4.13: Exemplo de programa com estrutura de seleção composta inútil.

Na Codificação 4.13, quando o valor é inferior a cem reais, não é preciso fazer nada especial, portanto o `else` é desnecessário, veja que seu bloco contém uma instrução inútil, que consome processamento e não altera o valor da variável.

Bibliografia e referências

PROGRAMIZ. **Python if...else Statement**. 2020. Disponível em: <https://www.programiz.com/python-programming/if-elif-else>. Acesso em: 14 fev. 2021.

PYTHON SOFTWARE FOUNDATION. **Expressions**. 2020. Disponível em: <https://docs.python.org/3/reference/expressions.html>. Acesso em: 21 jan. 2021.