**Objetivos**
- Entender o funcionamento do fluxo de execução do código, das estruturas de decisão e das estruturas de repetição;
- Combinar estruturas de decisão e de repetição para resolver problemas;
- Desenvolver códigos com estruturas de decisão e de repetição.

## 2.1 Introdução

Os códigos que desenvolvemos até o momento possui um fluxo de execução direto de forma que uma linha é executada após a outra sem nenhum tipo de desvio. Entretanto, para resolver diversos tipos de problemas precisamos ter um comportamento diferente no fluxo de execução. Nesses casos, pode ser necessário usar as estruturas de decisão e de repetição. Nas próximas seções estudaremos esses tipos de estruturas na linguagem Python.

## 2.2 Estruturas de decisão

Para resolver diversos tipos de problemas, precisamos usar as estruturas de decisão (também conhecidas como desvios condicionais ou estruturas condicionais). As estruturas de decisão são testes efetuados para decidir se uma determinada ação deve ser realizada (BORGES, 2010).

### 2.2.1 Estrutura de decisão simples

A estrutura de decisão mais simples utiliza uma única instrução **if**, seguida por uma expressão lógica e pelo bloco de instruções a ser executado se o valor da expressão lógica for verdadeiro (CEDER, 2018). Considere, por exemplo, o problema de verificar se um aluno foi aprovado. Isso é feito testando se a nota do aluno é maior ou igual a 60. A Figura 94 mostra o código para resolver esse problema.

```
1 nota = float(input('Informe a nota: '))
2 if nota >= 60:
3     print('Aprovado')
4 print('Boas férias')
```

Figura 26 – Estrutura de decisão simples para testar aprovação de aluno
Fonte: Elaborado pelo Autor.

A endentação deve ser feita corretamente para especificar quais instruções estão dentro da estrutura condicional. Utilizamos quatro espaços para endentar um bloco de instruções. Observe que, depois da expressão lógica seguida por dois pontos (**:**), começa o bloco de instruções do **if**. Esse bloco deve ser obrigatoriamente endentado. No código, a mensagem “Aprovado” é escrita na

tela somente quando a nota é maior ou igual a 60. Por outro lado, a mensagem “Boas férias” é mostrada incondicionalmente, pois não está dentro do desvio condicional.

### 2.2.2 Estrutura de decisão composta

Na estrutura de decisão simples, executamos alguma ação somente se a expressão lógica testada for verdadeira. Por outro lado, em certas situações, também precisamos realizar alguma medida se o teste for falso. Nesse caso, precisamos utilizar uma estrutura de decisão composta acrescentando a instrução **else** após o bloco de código da instrução **if**. A instrução **else** é seguida por outro bloco de código que será executado quando o teste for falso.

Como exemplo, vamos reconsiderar o problema de aprovação do aluno e escrever uma mensagem quando o mesmo for reprovado. A Figura 94 mostra o novo código contendo a mensagem “Reprovado” quando o aluno tem nota menor que 60.

```
1 n = float(input('Informe a nota: '))
2 if n >= 60:
3     print('Aprovado')
4 else:
5     print('Reprovado')
6 print('Boas férias')
```

Figura 27 – Código com estrutura de decisão composta
Fonte: Elaborado pelo Autor.

### 2.2.3 Estruturas de decisão aninhadas

Na estrutura de decisão simples, ocorre um único teste e o fluxo de execução do algoritmo pode seguir por um ou dois caminhos. Em determinadas situações, precisamos de algoritmos com vários testes e, por consequência, vários fluxos de execução. Para fazer isso, temos que inserir uma estrutura de decisão dentro de outra estrutura de decisão. Nesse caso, dizemos que as estruturas de decisão estão aninhadas.

Como exemplo, vamos considerar mais uma vez o problema de aprovação de aluno, mas, agora, vamos tratar as seguintes situações:

- Se o aluno tiver nota maior ou igual a 60, será aprovado;
- Se a nota for menor que 40, ao aluno é reprovado;
- Por fim, se a nota for maior ou igual a 40 e menor do que sessenta o aluno está de recuperação.

A Figura 94 apresenta o código para resolver o problema considerando as novas situações. Observe que, dentro do primeiro **else**, quando o aluno não é aprovado, ocorre um segundo teste para verificar se o aluno foi reprovado ou está de recuperação. O código pode ser escrito de outras formas, mudando a ordem dos testes, e obter o mesmo resultado.

```
1 n = float(input('Informe a nota: '))
2 if n >= 60:
3     print('Aprovado')
4 else:
5     if n >= 40:
6         print('Reavaliação')
7     else:
8         print('Reprovado')
9 print('Boas férias')
```

Figura 28 – Código com estruturas de decisão aninhadas
Fonte: Elaborado pelo Autor.

Observe que, no código da Figura 94, tivemos que endentar mais o **if** mais interno. Caso tenhamos muitas estruturas de decisão aninhadas, a legibilidade pode ficar prejudicada. Assim, outra maneira de usar as estruturas de decisão aninhadas é de forma consecutiva, como mostrado na Figura 94. A instrução **elif** funciona como uma junção do **else** com o **if**. Essa instrução é especialmente útil quando temos muitos testes consecutivos a serem feitos.

```
1 n = float(input('Informe a nota: '))
2 if n >= 60:
3     print('Aprovado')
4 elif n >= 40:
5     print('Reavaliação')
6 else:
7     print('Reprovado')
8 print('Boas férias')
```

Figura 29 – Código com estruturas de decisão consecutivas usando **elif**
Fonte: Elaborado pelo Autor.

## 2.2.4 Resolvendo problemas com estruturas de decisão

Como as estruturas de decisão são muito usadas, vamos resolver alguns problemas utilizando essas estruturas antes de avançarmos para o próximo conteúdo.

### Ano bissexto

Como primeiro problema, criaremos um código para verificar se um ano é bissexto ou não. Um ano é bissexto se é múltiplo de 400, ou então se é múltiplo de quatro, mas não é múltiplo de 100. Por exemplo, 2012 (múltiplo de 4, mas não múltiplo de 100) é bissexto, 1900 (múltiplo de quatro e de 100) não é bissexto, 2000 (múltiplo de 400 é bissexto). A Figura 94 mostra o código para resolver esse problema.

```
1 ano = int(input('Informe o ano: '))
2 if (ano % 400 == 0) or (ano % 4 == 0 and ano % 100 != 0):
3     print('Ano bissexto')
4 else:
5     print('Ano não bissexto')
```

Figura 30 – Código para detectar ano bissexto
Fonte: Elaborado pelo Autor.

Observe que temos um teste que combina expressões relacionais com operadores lógicos na linha 2. As expressões relacionais são usadas para testar se a variável **ano** é ou não múltipla de certos

números usando o operador de resto de divisão (**%**). Note que os parênteses na expressão após o **or** são obrigatórios porque temos que saber se o ano é múltiplo de 4 e não é múltiplo de 100 ao mesmo tempo. Poderíamos construir o código sem os operadores lógicos, mas precisaríamos de mais linhas com mais estruturas de decisão.

## Tipos de triângulos

Agora construiremos um código para classificar triângulos. Antes de mais nada, temos que testar se os três lados fornecidos pelo usuário são válidos para formar um triângulo. Isso é verdade apenas se nenhum lado for maior que a soma dos outros dois. Depois disso, temos que classificar o triângulo em isósceles (dois lados iguais), equilátero (três lados iguais) ou escaleno (três lados diferentes).

A Figura 94 apresenta o código classificador de triângulos. Assim como no problema anterior, combinamos expressões relacionais com operadores lógicos nas estruturas de decisão. Os parênteses delimitando as expressões relacionais (linhas 5, 8 e 10) poderiam ser omitidos, mas seu uso melhora a legibilidade do código.

```
1  print('Informes os 3 lados do triângulo:')
2  a = float(input('Lado 1: '))
3  b = float(input('Lado 2: '))
4  c = float(input('Lado 3: '))
5  if (a > b + c) or (b > a + c) or (c > a + b):
6      print('Triângulo inválido')
7  else:
8      if (a == b) and (b == c):
9          print('Triângulo equilátero')
10     elif (a == b) or (b == c) or (a == c):
11         print('Triângulo isósceles')
12     else:
13         print('Triângulo escaleno')
```

Figura 31 – Código de um classificador de triângulos
Fonte: Elaborado pelo Autor.

## Equações de segundo grau

Por fim, consideraremos a solução de equações de segundo grau no formato $Ax^2 + Bx + C = 0$. O algoritmo para resolver esse problema deve fazer o seguinte:

1) Ler os termos A, B e C;

2) Garantir que temos uma equação de segundo grau testando se A é diferente de zero;

3) Se for uma equação de segundo grau, calculamos o delta ($\Delta = B^2 - 4 \times A \times C$);

4) Após o cálculo, temos três situações para o delta:

- Se o delta for menor que zero, a equação não possui raízes;
- Se o delta for igual a zero, então a equação possui uma única raiz;
- Por fim, se o delta é maior que zero temos duas raízes $X_1 = \frac{-B+\sqrt{\Delta}}{2\times A}$ e $X_2 = \frac{-B-\sqrt{\Delta}}{2\times A}$.

A Figura 94 apresenta o código para resolver equações de segundo grau.

```
1  import math
2  print('Informe os termos da equação Ax² + Bx +C')
3  a = float(input('A: '))
4  b = float(input('B: '))
5  c = float(input('C: '))
6  if a == 0:
7     print('Não é uma equação de segundo grau')
8  else:
9     delta = b**2 - 4 * a * c
10    if delta < 0:
11       print('A equação não tem raízes')
12    elif (delta == 0):
13       x1 = b * (-1) / 2 * a
14       print('A equação possui a raiz:', x1, '')
15    else:
16       raiz_delta = math.sqrt(delta)
17       x1 = (b * (-1) + raiz_delta)/ 2 * a
18       x2 = (b * (-1) - raiz_delta)/ 2 * a
19       print('A equação possui duas raízes:')
20       print('x1 =', x1)
21       print('x2 =', x2)
```

Figura 32 – Código para resolver equações de segundo grau
Fonte: Elaborado pelo Autor.

**Atividade:** Resolva os exercícios a seguir. As respostas estão na próxima página, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

## 2.2.5 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Receber três números e informar o maior deles.

B) Testar se um número é ímpar ou par, sem usar o operador %.

C) Calcular o imposto de renda de um salário considerando as seguintes alíquotas:

- Até R$ 1.903,98: isento;
- De R$ 1.903,99 até R$ 2.826,65: 7,5%;
- De R$ 2.826,66 até R$ 3.751,05: 15%;
- De R$ 3.751,06 até R$ 4.664,68: 22,5%;
- Acima de R$ 4.664,68: 27,5%.

As respostas estão na próxima página, mas é importante que você tente resolver os problemas e use as respostas apenas para conferência.

## 2.2.6 Respostas dos exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Receber três números e informar o maior deles.

```
1  print('Informe três números')
2  a = int(input('A: '))
3  b = int(input('B: '))
4  c = int(input('C: '))
5  if (a > b) and (a > c):
6     print('O maior número é ', a)
7  elif b > c:
8     print('O maior número é ', b)
9  else:
10    print('O maior número é ', c)
```

B) Testar se um número é ímpar ou par, sem usar o operador %.

```
n = int(input('Informe um número: '))
if n / 2 * 2 - n == 0:
    print('O número é par!')
else:
    print('O número é ímpar')
```

C) Calcular o imposto de renda de um salário considerando as seguintes alíquotas:

- Até R$ 1.903,98: isento;
- De R$ 1.903,99 até R$ 2.826,65: 7,5%;
- De R$ 2.826,66 até R$ 3.751,05: 15%;
- De R$ 3.751,06 até R$ 4.664,68: 22,5%;
- Acima de R$ 4.664,68: 27,5%.

```
salario = float(input('Informe o salário: '))
if salario <= 1903.98:
    imposto = 0
elif salario <= 2826.65:
    imposto = salario * 7.5 / 100
elif salario <= 3751.05:
    imposto = salario * 15 / 100
elif salario <= 4664.68:
    imposto = salario * 22.5 / 100
else:
    imposto = salario * 27.5 / 100
print('O imposto de renda  é', imposto)
```

## 2.3 Estruturas de repetição

As estruturas de repetição possibilitam executar repetidamente blocos de instruções por um número definido de vezes ao até que uma dada condição seja atendida.

### 2.3.1 Laço de repetição while

O laço de repetição **while** é indicado quando não sabe o número de repetições a serem executadas. Por exemplo, a listagem dos números pares menores do que um número informado pelo usuário. Nesse problema, não é possível saber o número de repetições, pois não tem como prever o número que o usuário informará.

No laço **while**, as repetições ocorrem enquanto uma determinada condição for verdadeira (CORRÊA, 2020). Assim, é importante garantir que a condição de parada realmente aconteça e o laço não fique repetindo indefinidamente. Vamos considerar novamente o problema da listagem dos números pares. Nesse caso, a condição de parada pode ser quando chegarmos a um número maior do que o número informado pelo usuário.

A Figura 94 mostra o código para listagem de números pares. Usamos a variável **atual** para armazenar o número atual, começando pelo zero. O laço **while** usa a condição

de parada **atual < n**, para verificar se o número atual ainda é menor que **n** informado pelo usuário.

```
1 atual = 0
2 n = int(input('Informe um número: '))
3 while atual < n:
4     print(atual)
5     atual += 2
```

Figura 33 – Listagem de números pares
Fonte: Elaborado pelo Autor.

A estrutura de repetição **while** possui uma condição que é testada antes mesmo de executar a primeira repetição. Enquanto essa condição for verdadeira, as repetições continuam acontecendo. Se o usuário informar zero, por exemplo, não acontece nenhuma repetição. Dentro do laço, somamos mais dois ao número atual para chegarmos ao próximo número par.

### 2.3.2 Laço de repetição for

O laço de repetição **for** é indicado quando se sabe o número de repetições a serem feitas. Basicamente, o laço **for** percorre de forma automática os elementos de uma estrutura de lista. A maneira mais comum de utilizar o laço **for** é com a função **range()**. Essa função recebe um número inteiro **n** e gera um intervalo de números de 0 até **n – 1** (CORRÊA, 2020).

Para exemplificar o uso do laço **for**, vamos considerar o problema de somar 10 números informados pelo usuário. A Figura 94 mostra o código para resolver esse problema. Observe que, no laço, for usamos a função **range(10)** e a variável **cont** para percorrer os números do intervalo gerado pela função **range()**. Assim, na primeira repetição, a variável **cont** vale **0**, na segunda, vale **1**, e assim por diante até assumir o valor **9**.

```
1 soma = 0
2 for cont in range(10):
3     n = float(input('Informe um número: '))
4     soma += n
5 print(soma)
```

Figura 34 – Soma de 10 números usando o laço **for**
Fonte: Elaborado pelo Autor.

É possível notar, no código para soma dos 10 números, que a única função da variável **cont** é controlar as repetições do laço **for**. Quando temos uma variável que não é usada em nenhuma outra parte do código, podemos substituir essa variável pela variável anônima **_** (sublinhado).

Outro detalhe importante é a função **range()**. Além do fim do intervalo, podemos definir o início e a periodicidade dos números. Se usarmos, por exemplo, a instrução **range(2, 6)**, será retornado o intervalo de números "**2, 3, 4, 5**", ou seja, o primeiro número é o início e o segundo número é o fim do intervalo. Lembrando que o número do fim não é incluído no intervalo. Quando incluímos um terceiro número, definimos a periodicidade dos números. Como exemplo, se usarmos a instrução **range(3, 19, 4)** teremos a sequência "**3, 7, 11, 15**". A sequência começa em **3**, e os demais números são obtidos somando **4** ao número atual, até atingir o fim do intervalo. Além disso, no lugar dos números, podemos usar qualquer variável ou expressão que retorne um número inteiro. Também podemos obter intervalos em ordem decrescente, por exemplo, a instrução **range(5,0,-1)** retorna a sequência "**5, 4, 3, 2, 1**".

### 2.3.3 Interrupção e continuação de laços de repetição

O laço de repetição **while** precisa testar a condição de parada antes mesmo da primeira repetição. Entretanto, em diversos momentos, precisamos executar a primeira repetição antes de testar a condição de parada. Nesse caso, podemos criar um laço com a instrução **while True** e utilizar a instrução **break** para finalizar o laço de repetição.

Como exemplo, vamos considerar a soma de uma quantidade indeterminada de números informados pelo usuário. Devemos parar de somar apenas quando o usuário informar o número **0** (zero). A Figura 94 apresenta o código para resolver esse problema. É importante tomar um certo cuidado com laços **while True**, temos que garantir que a instrução **break** será executada em algum momento e o laço não fique repetido indefinidamente.

```
1 soma = 0
2 while True:
3     n = float(input('Informe um número: '))
4     if n == 0:
5         break
6     soma = soma + n
7 print('Soma dos números:', soma)
```

Figura 35 – Soma indefinida de números
Fonte: Elaborado pelo Autor.

Vamos considerar agora uma modificação no problema de somar números. Além do que já foi mencionado, suponha que os números negativos não devam ser somados. Diante disso, podemos usar a instrução **continue** para ignorar os números negativos e *pular* para a próxima repetição do laço (CEDER, 2018). O código da Figura 94 mostra a solução para a modificação do problema. As instruções **break** e **continue** podem ser usadas tanto no laço **while** quanto no laço **for**.

```
1 soma = 0
2 while True:
3     n = float(input('Informe um número: '))
4     if n < 0:
5         continue
6     if n == 0:
7         break
8     soma = soma + n
9 print('Soma dos números:', soma)
```

Figura 36 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

### 2.3.4 Resolvendo problemas com estruturas de repetição

Assim como as estruturas de decisão, as estruturas de repetição também são extensamente usadas para resolver diversas categorias de problemas. Portanto, demostraremos como usá-las para resolver alguns problemas e praticar o uso das estruturas antes de avançarmos para o próximo conteúdo.

#### Cálculo de MDC

Um problema interessante para ser resolvido com laço de repetição é o cálculo de máximo divisor comum (MDC) com a técnica de Euclides (WIKIPÉDIA, 2021a). Lembrando que o MDC de números é o maior número que os divide sem deixar resto. Basicamente, a técnica de Euclides consiste nos seguintes passos:

- Dividimos o maior número pelo menor e verificamos se o resto da divisão é zero;
- Em caso afirmativo, o menor número é o MDC;
- Caso contrário, substituímos o maior número pelo menor, o menor número pelo resto da divisão e repetimos o processo.

Repare que a repetição do processo no terceiro ponto caracteriza uma estrutura de repetição. Como exemplos, demonstraremos como calcular o MDC de 144 e 56. O processo é o seguinte:

- Começamos dividindo 144 por 56. Como o resto da divisão é 32, vamos considerar os números 56 e 32 e repetir o processo;
- Dividimos 56 por 32 e temos 24 como resto. Agora, consideramos 32 e 32 e dividimos novamente;
- Na divisão de 32 por 24, chegamos a 8 como resto. Assim, a próxima divisão será 24 por 8;
- Por fim, dividimos 24 por 8 e temos zero como resto. Logo, o MDC é o número 8.

O laço de repetição mais adequado para implementar o algoritmo de Euclides é o **while**, pois não tem como prever quantas divisões precisam ser feitas. A Figura 94 mostra o código do algoritmo de Euclides.

A condição de parada do laço é **True**. Todavia, na linha 11, testamos se o resto da última divisão for zero, interrompemos o laço com a instrução **break** (na linha 12). Além do laço de repetição, usamos uma estrutura de decisão na linha 5 para testar se os números são válidos (positivos maiores que zero). O **print()** da linha 10 não é necessário, ele foi incluído apenas para mostrar as divisões do processo.

```
1  print('Informe dois números')
2  n1 = int(input('N1: '))
3  n2 = int(input('N2: '))
4
5  if n1 < 1 or n2 < 1:
6     print('Números inválidos para MDC')
7  else:
8     while True:
9        resto = n1 % n2
10       print (n1, '/' , n2, '-> resto:', resto)
11       if resto == 0:
12          break
13       n1 = n2
14       n2 = resto
15    print('O MDC é ', n2)
```

Figura 37 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

## Combinações de elementos

Agora, vamos considerar o problema de gerar as combinações de dois elementos a partir de um conjunto de números naturais com **n** elementos, ou seja, um conjunto A = {1, 2, 3, ..., n}. Antes de gerar as combinações, o código deve perguntar o número de elementos do conjunto ao usuário. A Figura 94 mostra o código para resolver o problema descrito.

```
1  n = int(input('Informe o número de elementos do conjunto: '))
2
3  print('Elementos:', end=' ')
4  for cont in range(1, n+1):
5      print(cont, end=' ')
6
7  print('\nCombinações:', end='')
8  for cont in range(1, n+1):
9      for cont2 in range(1, n+1):
10         print('(', cont, ', ', cont2, ')', sep='', end=' ')
```

Figura 38 – Combinações com dois elementos de um conjunto
Fonte: Elaborado pelo Autor.

Após pegar o número de elementos informado pelo usuário (linha 1), escrevemos todos os elementos do conjunto na tela (linhas 3 a 5). No laço de repetição usamos a instrução **range(1, n+1)** para termos a sequência de elemento de **1** até **n**. Repare também que usamos o parâmetro **end= ' '**, na função **print()** para evitar a quebra de linha e os elementos ficarem um após o outro.

Nas linhas 7 a 10, escrevemos as combinações dos elementos. Usamos dois laços de repetição aninhados para que cada elemento seja combinado com os demais (inclusive, com ele mesmo). O primeiro elemento é a variável **cont** e o segundo elemento é a variável **cont2**. A função **print()** dentro do laço mais interno escreve cada uma das combinações. Foram usados os parâmetros **sep=''** e **end=' '** para que os elementos fiquem juntos e as combinações fiquem separadas.

## Subconjuntos

Por fim, vamos considerar o problema de gerar os subconjuntos de dois elementos a partir de um conjunto com **n** números naturais. A princípio, esse problema pode ser parecido com o anterior. No entanto, pela definição matemática, temos que considerar os seguintes pontos:

- Os subconjuntos não podem ter elementos repetidos;
- A ordem dos elementos não altera o conjunto;
- Não devemos ter subconjuntos repetidos.

A Figura 94 mostra o código que resolve esse problema. Veja que esse código é muito parecido com a solução para as combinações. A diferença está no laço mais interno, o intervalo de elementos desse laço começar com o primeiro elemento maior do que **cont**. Isso garante que não combinaremos um elemento com ele mesmo nem com seus antecessores.

```
1  n = int(input('Informe o número de elementos do conjunto: '))
2
3  print('Elementos:', end=' ')
4  for cont in range(1, n+1):
5      print(cont, end=' ')
6
7  print('\nSubconjuntos:', end='')
8  for cont in range(1, n+1):
9      for cont2 in range(cont+1, n+1):
10         print('{', cont, ', ', cont2, '}', sep='', end=' ')
```

Figura 39 – Subconjuntos com dois elementos de um conjunto
Fonte: Elaborado pelo Autor.

**Atividade:** Resolva os exercícios a seguir. As respostas estão na próxima página, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

### 2.3.5 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Leia uma quantidade indeterminada de números. A cada número informado, o usuário deve informar se deseja continuar ou parar. Ao final, o código deve retornar o maior e o menor número recebido.

B) Calcule o fatorial de um número. O fatorial de um número n, representado por n!, é calculado como n! = n × (n − 1) × ... × 2 × 1. Sendo que 1! = 0! = 1.

C) Simular uma calculadora simples. O código deve solicitar ao usuário a operação desejada (soma, multiplicação, divisão, subtração ou potência) ou então sair. Quando o usuário escolher uma operação, o código deve solicitar dois números, realizar a operação sobre estes números e exibir o resultado. O código deve sempre solicitar uma nova operação até que o usuário escolha sair.

### 2.3.6 Respostas dos exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Leia uma quantidade indeterminada de números. A cada número informado, o usuário deve informar se deseja continuar ou parar. Ao final, o código deve retornar o maior e o menor número recebido.

```
maior = float('-inf') # -∞, qualquer número pode ser maior
menor = float('inf')  # +∞, qualquer número pode ser menor
while True:
    n = float(input('Informe um número: '))
    if n > maior:
        maior = n
    if n < menor:
        menor = n
    resp = input('Deseja continuar? (S/N)')
    if resp.lower() == 'n':
        break
print('Maior número informado:', maior)
print('Menor número informado:', menor)
```

B) Calcule o fatorial de um número. O fatorial de um número n, representado por n!, é calculado como n! = n × (n − 1) × ... × 2 × 1. Sendo que 1! = 0! = 1.

```
n = int(input('Informe um número: '))
fat = 1
for cont in range(n, 1, -1):
    fat *= cont
print('O fatorial de', n, 'é', fat)
```

C) Simular uma calculadora simples. O código deve solicitar ao usuário a operação desejada (soma, subtração, multiplicação ou divisão) ou então sair. Quando o usuário escolher uma operação, o código deve solicitar dois números, realizar a operação sobre estes números e exibir o resultado. O código deve sempre solicitar uma nova operação até que o usuário escolha sair.

```
while True:
    print('Calculadora (+, -, /, *)')
    print('s: sair')
    resp = input('Informe a operação desejada (s para sair): ')
    if resp == 's':
        break
    print('Informe dois números:')
    n1 = float(input('N1: '))
    n2 = float(input('N1: '))
    if resp == '+':
        r = n1 + n2
    elif resp == '-':
        r = n1 - n2
    elif resp == '*':
        r = n1 * n2
    elif resp == '/':
        r = n1 / n2
    else:
        print('Operação inválida!')
        continue
    print(n1, resp, n2, '=', r)
```

## 2.4 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

**Mídia digital:** Antes de avançarmos os estudos, vá até a sala virtual e assista ao vídeo “Revisão da Segunda Semana” para recapitular tudo que aprendemos.

Nos encontramos na próxima semana.

Bons estudos!