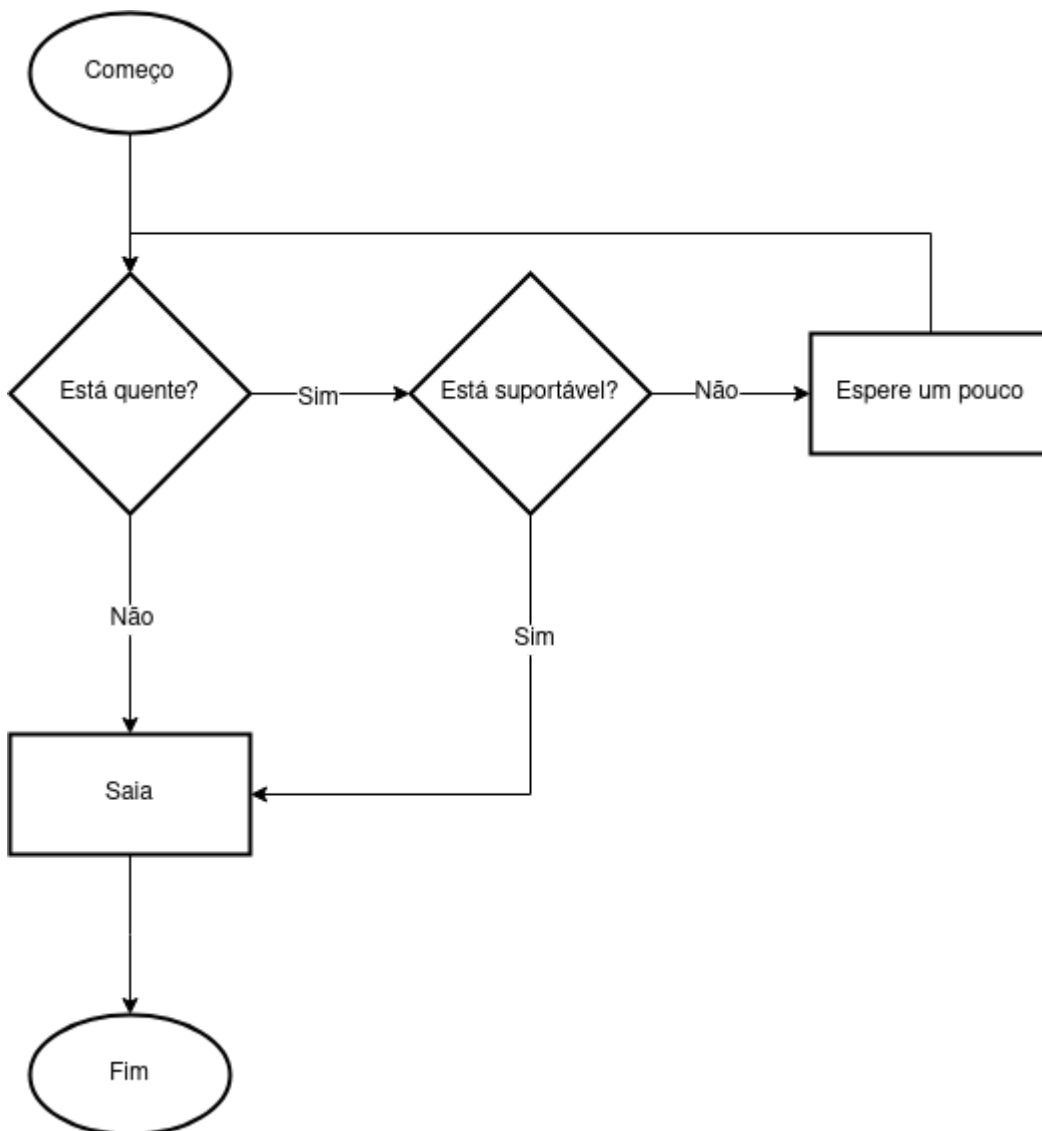


Capítulo 2

Capítulo 2: Controle de Fluxo | Flow Control

Quando falamos de controle de fluxo, nos referimos à habilidade de um programa de decidir o que fazer ou quantas vezes fazer, dependendo de uma condição. Dê uma olhadinha no fluxograma a seguir:



Neste exemplo, antes de decidirmos o que fazer, avaliamos certas condições. Está quente? Se não estiver, saímos. Caso esteja, verificamos então se está suportável, se estiver, saímos, se não estiver, esperamos um pouco e verificamos novamente.

Neste capítulo você aprenderá a utilizar o controle de fluxo para aumentar as funcionalidades dos seus programas.

Operadores de comparação | Comparison Operators

Antes de começarmos a utilizar o controle de fluxo nos nossos códigos, precisamos falar sobre os operadores de comparação. Eles são utilizados para formar expressões, que nada mais são do que uma combinação de operadores e valores que resulta em um valor. Nós já utilizamos expressões anteriormente, por exemplo:

```
number = 10 + 2
```

10 + 2 é uma expressão que forma o valor 12. Nada de novo no horizonte. Com os operadores de comparação, nós podemos criar expressões um pouco mais complexas. Como, por exemplo:

```
is_less_than = 2 < 10  
# is_less_than = True
```

Nesta expressão, python vai checar se 2 é menor que 10 e resultado vai ser um `boolean`, ou seja, verdadeiro ou falso.

Os operadores de comparação são:

Operador	O que faz
==	igual a
!=	não é igual a
>	maior que
<	menor que
>=	maior ou igual a
<=	menor ou igual a

```
number1 = 10  
number2 = 2  
  
print(f"{number1} é menor que {number2}? {number1 < number2}")  
print(f"{number1} é maior que {number2}? {number1 > number2}")  
print(f"{number1} é igual a {number2}? {number1 == number2}")  
print(f"{number1} não é igual a {number2}? {number1 != number2}")
```

```
10 é menor que 2? False  
10 é maior que 2? True  
10 é igual a 2? False  
10 não é igual a 2? True  
  
Process finished with exit code 0
```

Nota: Não confunda o operador de atribuição (=) com o operador de igualdade (==).

If / Elif / Else

A primeira forma de controle de fluxo que nós veremos é o `if / else`. A ideia é simples: Se a condição for verdadeira, faça isto, caso contrário, faça aquilo.

```
name = "Jorge"

if name == "Jorge":
    print("Hello, Jorge")
else:
    print("Hello, person")
```

Neste exemplo, comparamos se o valor da variável `name` é igual a "Jorge", python avalia esta expressão como verdadeira então faz os comandos seguintes. Resultando em:

```
Hello, Jorge

Process finished with exit code 0
```

Caso o valor da variável `nome` não fosse igual a "Jorge", python executaria os comandos após `else`.

Tá, e o que é o `elif`? O `elif` é utilizado quando nós temos mais de duas opções.

```
if name == "Jorge":
    print(f"Hello, Jorge")
elif name == "Janine":
    print("Hello, Janine")
else:
    print(f"Hello, person")
```

Primeiro python checa se o valor da variável `name` é igual a "Jorge"; Caso não seja, ele checa se o valor é igual a "Janine"; Se também não for, ele executa o código em `else`.

Nota: Lembre-se que os códigos são executados de cima pra baixo. Uma vez que uma condição é verdadeira, python vai executar os códigos em seu bloco e não vai mais verificar se as outras condições também são verdadeiras.

Importante: `if` não precisa de um `else` para funcionar. Você pode ter um `if` sozinho e se a condição não for verdadeira, python pula o `if` e continua rodando o resto do código. O `if` pode também terminar com um `elif` em vez de um `else`.

Indentação | Indentation

Indentação é o espaço no começo de uma linha. Na maioria das linguagens, ela só existe para organizar e embelezar o código. Em outras, como python, ela é extremamente importante pois define onde o bloco de código começa e termina. A indentação é feita com 4 espaços. A maioria dos editores de texto e IDEs permitem indendar com a tecla *tab*.

```
if name == "Jorge":
    # este código está indentado e pertence ao bloco de código do "if"
    print("Hello, Jorge")
else:
    # este código está indentado e pertence ao bloco de código do
    "else"
    print("Hello, person")
```

Um bloco de código pode conter outros blocos de código, como vemos no seguinte código:

```
username = "Ushi"
password = "litterbox"

if username == "Ushi":

    print("Hello, Ushi")

    if password == "litterbox":
        print("Login successful.")
    else:
        print("wrong password.")

else:
    print("unknown user.")
```

Vamos ver passo a passo como python executará este código:

1. Primeiro, python checará se `username == "Ushi"`, caso seja, ele rodará o seguinte bloco de código:

```
username = "Ushi"
password = "litterbox"

if username == "Ushi":
    print("Hello, Ushi")

    if password == "litterbox":
        print("Login successful.")
    else:
        print("wrong password.")
else:
    print("unknown user.")
```

2. Então executará `print("Hello, Ushi")` e checará se `password == "litterbox"`, caso seja, ele rodará o bloco de código mostrado abaixo:

```
username = "Ushi"
password = "litterbox"

if username == "Ushi":
    print("Hello, Ushi")

    if password == "litterbox":
        print("Login successful.")
    else:
        print("wrong password.")
else:
    print("unknown user.")
```

3. Caso `password` não seja "litterbox", ele executará o seguinte bloco:

```
username = "Ushi"
password = "litterbox"

if username == "Ushi":
    print("Hello, Ushi")

    if password == "litterbox":
        print("Login successful.")
    else:
        print("wrong password.")
else:
    print("unknown user.")
```

4. Se `username` não for "Ushi", ele executará o seguinte bloco:

```
username = "Ushi"
password = "litterbox"

if username == "Ushi":
    print("Hello, Ushi")

    if password == "litterbox":
        print("Login successful.")
    else:
        print("wrong password.")
else:
    print("unknown user.")
```

Quebras de linha | Line breaks

Ao contrário da indentação, que é tão importante em python, line breaks são ignoradas e não afetam seu programa de nenhuma forma. Sendo assim, podemos usá-las para melhor organizar nosso código, deixando-o mais legível.

Operadores Lógicos | Logical Operators

Operadores lógicos são utilizados quando queremos avaliar duas ou mais expressões. Existem 3 operadores lógicos em python: `and`, `or`, `not`.

Nota: Para ajudar na leitura, algumas pessoas gostam de colocar as expressões entre parênteses.

and

Quando utilizamos o operador `and`, python só executará o bloco de código se todas as expressões forem verdadeiras.

```
first_number = 15
second_number = 6

if (first_number > 10) and (second_number < 20):
    print("yay")
```

Aqui, `first_number` é maior que 10 e `second_number` é menor que 20. Ambas as expressões são verdadeiras, logo, python executará o bloco de código.

or

Quando utilizamos o operador `or`, python executará o bloco de código caso uma das expressões sejam verdadeiras.

```
first_number = 15
second_number = 6

if first_number > 10 or second_number < 5:
    print("yay")
```

Neste caso, `first_number > 10` é verdadeiro e `second_number < 5` é falso. Python executará o bloco de código pois uma das expressões é verdadeira.

not

Quando utilizamos o operador `not`, invertemos o valor booleano. Se a expressão for verdadeira, ela passará a ser falsa; Se for falsa, passa a ser verdadeira.

```
number = 6

if not number < 5:
    print("yay")
```

In this example, `number < 5` é falso, como utilizamos o operador `not`, o bloco de código será executado pois `not False` é `True`.

Operador Ternário | Ternary Operator

O operador ternário é um "if de uma linha", é uma maneira mais concisa de se escrever um if/else simples.

```
valor_produto = 999.99
posso_pagar = "não" if valor >= 1200 else "sim"
```

O valor da variável `posso_pagar` será "não" se o valor do produtor for maior ou igual a 1200, caso contrário, será "sim".

Exercício 3

1. Crie um programa que peça para o usuário as notas de 3 provas, calcule a média, imprima-a na tela e, se ela for maior ou igual 6, imprima "passou! (:", se estiver entre 4 e 6, imprima "está de recuperação" e, se a média for menor que 4, imprima na tela que o aluno foi reprovado.
2. Crie um programa que peça para o usuário digitar seu peso e sua altura, calcule o IMC (Índice de Massa Corporal), mostre-o em duas casas decimais e imprima na tela as informações seguindo a tabela abaixo:

IMC	CLASSIFICAÇÃO
<18,5	Baixo peso
18,5 a 24,9	Peso normal
25,0 a 29,9	Excesso de peso
>30,0	Obeso

3. Crie um programa que calcule o valor da gorjeta. Peça ao usuário o valor da conta e quantos por cento do valor ele quer deixar de gorjeta, e imprima na tela o valor da conta, a porcentagem e o valor total da conta com a gorjeta inclusa.
-

Laços | Loops

O que são loops? Imagine que você está desenhando um círculo, quando você chega ao final do desenho, você encontra o ponto inicial do desenho. Na programação, loops são usados na repetição de códigos, quando um chega ao seu final, ele recomeça. Até agora, nossos códigos começaram e terminaram, de cima pra baixo. Mas e se quiséssemos que o código rodasse uma certa quantidade de vezes, ou se rodasse até uma condição ser verdadeira?

Nota: Cada "volta" de um loop é chamada de iteração.

For loop

O `for` loop é utilizado quando sabemos quantas vezes o código precisa ser repetido. Ele é utilizado, normalmente, junto com a função `range`. Vamos analisar o `for` loop um pouco mais a fundo:

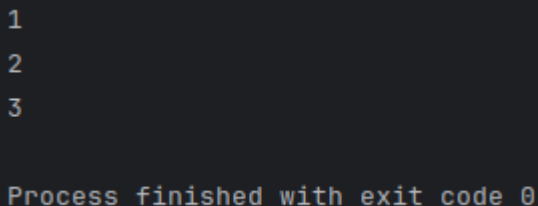
```
for number in range(1, 4):  
    print("Hello")
```

Primeiro utilizamos o comando `for`, depois criamos uma variável que eu decidi chamar de `number`, usamos o comando `in` e chamamos a função `range`. Outra forma de ler isto, seria: "para cada número de 1 até 4, não incluindo 4, faça isto."

Quando o loop começar, a variável `number` vai ter o valor de 1, python vai rodar os comandos no bloco de código e vai voltar para o começo do `for` loop, desta vez, `number` vai ter o valor de 2, o bloco de código vai rodar novamente e voltar para o começo, e agora `number` terá o valor de 3, o bloco de código vai rodar e o loop vai acabar.

Nós podemos acessar a variável `number` dentro do `for` loop.

```
for number in range(1, 4):  
    print(number)
```



```
1  
2  
3  
  
Process finished with exit code 0
```

Como você pode ver aqui, no primeiro loop `number` tinha o valor de 1 e isso foi impresso na tela, no segundo tinha o valor de 2 e no terceiro o valor de 3.

Laço for aninhado | Nested for loops

Assim como com o `if`, nós podemos criar um `for` loop dentro de outro.

```
for i in range(1, 5):  
    print(i, end=': ')  
  
    for j in range(1, 5):  
        print(j, end=' ')  
  
    print()
```

Eu sei que este código assusta um pouco, então vamos lê-lo com calma. O primeiro loop começa e a variável `i` tem o valor de 1, o bloco de código então roda. Nele, imprimimos `i`; Em seguida o segundo `for` loop começa. Neste, a variável `j` será impressa 4 vezes (de 1 a 5 não incluindo 5), o segundo `for` termina de ser executado, voltamos o bloco de código do primeiro loop, e pulamos uma linha com a função `print()`.

Este é o resultado:

```
1: 1 2 3 4
2: 1 2 3 4
3: 1 2 3 4
4: 1 2 3 4

Process finished with exit code 0
```

Nota: A função `print()`, normalmente, pula uma linha depois da impressão. Nós podemos alterar este comportamento com o parâmetro `end=`. Com ele podemos definir se vai ter um espaço depois de cada impressão, uma vírgula, etc.

É importante ressaltar que a variável `i` pode ser acessada em qualquer lugar dentro do bloco de código do primeiro `for` loop, até mesmo dentro do segundo `for` loop, por isso utilizei nomes diferentes para os dois.

```
for i in range(1, 5):
    print(i)
    # A variável i pode ser acessada aqui, mas a j não.

    for j in range(1, 5):
        print(i, j)
        # tanto a variável i quanto a j podem ser acessadas aqui.
    print()
```

range()

Apesar de já termos visto como a função `range()` funciona, quero apenas reforçar o que vimos e adicionar mais uma coisinha. A função `range` pode ter até 3 parâmetros:

```
range(começo, fim, passo)
```

O primeiro, que aqui chamei de `começo`, define em que número a contagem vai começar; o segundo, que chamei de `fim`, define em que número a contagem vai terminar, não incluindo este número; e a terceira, que chamei de `passo` determina a que passo a contagem é feita, de um em um, dois em dois, três em três. O padrão é de um em um.

```
for num in range(0, 11, 2):
```

```
print(num)
```

```
0
2
4
6
8
10

Process finished with exit code 0
```

Nota: Se passarmos apenas um valor como argumento, python conta de 0 até este valor, sem incluí-lo. `range(3)` terá um resultado de 0, 1, 2.

Laço while | While loop

O `while` loop é executado enquanto uma condição for verdadeira. Ele é utilizado quando não sabemos quantas vezes o código precisará ser executado.

```
number = 1

while number <= 5:
    print(number)
    number += 1
```

Neste exemplo, python vai testar se `number <= 5` é verdadeiro, caso seja, ele vai rodar o bloco de código, imprimir o valor de `number` e somar 1 ao mesmo; caso não seja, o loop acaba.

```
1
2
3
4
5

Process finished with exit code 0
```

Importante: Cuidado para não criar um loop infinito. Se a condição nunca se tornar falsa, o programa vai rodar infinitamente e, eventualmente, "crashar", ou seja, parar de funcionar e fechar.

Valores truthy e falsy | truthy and falsy values

Quando tentamos passar dados como condição, ao invés de uma expressão, python avalia aquele dado como `True` ou `False`. Quando o valor é verdadeiro, nós chamamos de "truthy", e quando é falso, de "falsy". Por exemplo:

```
while 10:
    print("truthy")
    # isto é um loop infinito
```

Os dados `0`, `0.0`, e `''`, são falsy. Qualquer outro dado será truthy.

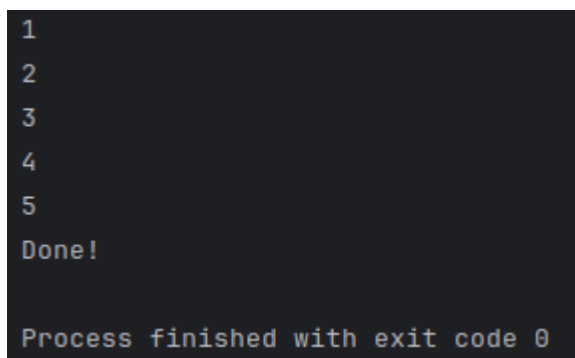
Break

`break` é um comando utilizado dentro de um loop. Ele força python a parar o loop e sair dele.

No código a seguir, o loop vai rodar e imprimir o valor de `num` enquanto seu valor não for maior que 5, quando isso acontecer, python sairá do loop e continuar com o código fora do `for` loop.

```
for num in range(1, 10):
    if num > 5:
        break
    print(num)

print("Done!")
```



```
1
2
3
4
5
Done!

Process finished with exit code 0
```

Continue

O comando `continue` também é utilizado dentro de um loop. Quando python chega neste comando, ele passa para a próxima iteração do loop.

```
for num in range(1, 11):
    if 2 < num < 8:
        continue
    print(num)

print("Done!")
```

Aqui, estamos dizendo para python que, quando `num` estiver entre 2 e 8, para pular para a próxima iteração do loop. Ou seja, quando `num` for igual a 3, python vai chegar no comando

`continue` e vai voltar pro começo, e `num` passará a ter o valor de 4, e assim sucessivamente até `num` ser maior ou igual a 8.

Este será nosso resultado:

```
1
2
8
9
10
Done!

Process finished with exit code 0
```

Exercício 4

1. Crie um programa que imprima números de 1 a 10 e diga se são pares ou ímpares.
2. Crie um programa que imprima os números de 1 a 50, porém, caso o número seja divisível por 3, imprima "Fizz", caso seja divisível por 5, imprima "Buzz", caso seja divisível por 3 e por 5, imprima "FizzBuzz".
3. Crie um programa que imprima a seguinte figura:

```
#
##
###
####
#####

Process finished with exit code 0
```

Bonus: Controle de fluxo em Java

Você já pode ter visto códigos com uma sintaxe diferente de python, códigos com parenteses, chaves, com uma aparência bem mais assustadora. Mas a verdade é que não passa disso: aparência. Os conceitos são os mesmos em todas as linguagens. Claro, cada linguagem tem suas particularidades, mas a base é sempre a mesma.

Nesta seção bonus eu quero te mostrar como seriam os controles de fluxo na linguagem java e explicá-los para que, caso encontre com outras linguagens que não sejam python, consiga ler os códigos sem problema algum.

if/else if/else

```
int number = 16;

if (number <= 10) {
    System.out.println("Small number");
} else if (number <= 30) {
    System.out.println("Medium number");
} else {
    System.out.println("Big number");
}
```

Pode ser que você consiga ler este código em java sem muitos problemas, mas vamos analisá-lo juntos, com calma, mesmo assim.

Primeiro nós criamos uma variável do tipo `int` de valor 16 e então começamos nosso `if`. Entre parenteses nós temos a condição que será verificada e, dentro das chaves, nós temos o bloco de código que será executado se a condição for verdadeira. Neste caso, o comando `System.out.println()`, que é o equivalente, em java, ao `print()` em python. Outra coisa diferente é que, em java, em vez de `elif`, nós usamos `else if`.

Laço for | For loop

```
for (int i = 0; i < 10; i++) {
    System.out.println("i = " + i);
}
```

Este é um pouco mais assustador, não é?

Neste tipo de `for` loop, dentro dos parênteses, nós criamos uma variável `i` (mas pode ter qualquer outro nome, apesar deste ser o padrão) de valor 0, definimos que o loop rodará enquanto `i < 10`, e depois escrevemos `i++`, que é o equivalente a `i += 1` em python. Ou seja, após rodar o bloco de código que está entre chaves, 1 será somado ao valor de `i`. Este código imprimirá na tela a `string` concatenada `"i = " + i`.

O equivalente a este `for` loop em python é:

```
for i in range(0, 10):
    print(f"i = {i}")
```

Laço while | While loop

```
int i = 0;

while (i < 10) {
    System.out.println("i = " + i);
}
```

```
        i++;  
    }
```

O `while` loop já tem uma sintaxe parecida com a de python. Primeiro criamos uma variável do tipo `int` de valor 0 para criar a condição. O loop rodará enquanto `i < 10`, imprimirá na tela `"i = " + i` e acrescentará 1 ao valor de `i` antes de checar a condição novamente.

O equivalente a este `while` loop em python é:

```
i = 0  
  
while i < 10:  
    print(f"i = {i}")  
    i += 1
```
