Malhas de repetição condicionais

Imagine que você queira fazer um programa que exibe os números de 1 até 5, em ordem crescente. Uma possibilidade seria:

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

Porém, imagine que os requisitos do programa acabam sendo alterados, e agora o seu programa deverá ir até 1000. Ou, pior, imagine que o usuário irá digitar um valor e seu programa deverá contar apenas até o valor digitado. Note como fica difícil resolver esses problemas apenas copiando e colando linhas de código.

Vamos pensar em outro tipo de problema. Na aula passada, fizemos um exercício onde precisávamos validar algumas entradas do usuário. Uma dessas entradas era a idade, e gostaríamos de aceitar apenas valores entre 0 e 150. Sua solução provavelmente foi parecida com o código abaixo:

```
idade = int(input('Digite a idade: '))

if idade < 0 or idade > 150:
  print('Erro')
```

Mas imagine que, ao invés de apenas mostrar uma mensagem de erro, nós devêssemos obrigar o usuário a continuar digitando valores novos para idade até que ele digite um valor válido (entre 0 e 150). Isso não seria possível utilizando apenas **if**, **elif** e **else**.

1.1. Enquanto

Os problemas enunciados acima podem ser resolvidos utilizando estruturas do tipo "enquanto". Em Python, a instrução **while** é bastante parecida com o **if**: ela possui uma expressão lógica, e seu conteúdo só será executado se a expressão for verdadeira. Porém, após chegar ao final, ela retorna ao início e testa novamente a condição.

Se ela for verdadeira, seu conteúdo será executado de novo. Ao final da nova execução, a condição é testada novamente, e assim sucessivamente. A execução só será interrompida quando o teste se tornar falso. Vejamos como resolver o problema da idade utilizando o **while**:

```
idade = int(input('Digite a idade: '))

while idade < 0 or idade > 150:
  print('Erro! Idade deve estar entre 0 e 150!')
  idade = int(input('Digite a idade: '))

print('Obrigado!')
```

Faça alguns testes com o programa acima. Note que se você digitar uma idade válida desde o início, ele nunca chega a mostrar erro: o **while** é como um **if** e será ignorado se sua condição for falsa. Porém, caso você digite valores inválidos, a condição será verdadeira e ele irá executar *enquanto* você estiver digitando valores falsos.

Estruturas do tipo "enquanto" são conhecidas como *malhas de repetição* ou *loops*.

1.2. Condição de parada

No exemplo anterior, o que determina se o *loop* prossegue ou não é o valor de idade. Esse valor, por sua vez, pode mudar em cada execução do *loop*, já que temos um **input** lá dentro. Experimente rodar o programa sem aquele **input** e verifique o que ocorre.

```
idade = int(input('Digite a idade: '))

while idade < 0 or idade > 150:
  print('Erro! Idade deve estar entre 0 e 150!')
print('Obrigado!')
```

O que ocorreu é o que chamamos de *loop infinito*: se a condição for verdadeira uma vez, ela será para sempre, já que nunca mais alteramos o valor da variável envolvida no teste lógico. É importante criar caminhos para que a condição possa se tornar falsa em algum momento. Isso é o que chamamos de **condição de parada** do nosso loop.

1.3. Sequências numéricas

Iniciamos essa aula enunciando um problema onde gostaríamos de exibir números sequencialmente na tela. Isso é possível de resolver utilizando *loops*. Primeiro, observe o exemplo abaixo e responda: qual valor aparecerá na tela?

```
x = 5
x = x + 1
print(x)
```

Essa construção parece pouco intuitiva porque na matemática o operador = é bidirecional: a expressão "**a = b**" significa que **a** é igual a **b** e **b** é igual a **a**. Ao vermos **x** aparecendo em ambos os lados, parece que podemos simplesmente cortar dos dois lados, resultando em **0 = 1**, o que é uma inverdade.

Em Python o operador = na verdade **não é** o operador de igualdade da matemática, e sim o operador de **atribuição** de valores. Ou seja, o que ele diz é "pegue o resultado da expressão à direita e guarde na variável à esquerda". Portanto, o exemplo acima pega primeiro o valor antigo de **x**, que era 5, adiciona 1, resultando em 6, e guarda este novo resultado na variável **x**, substituindo o valor antigo. Logo, a resposta na tela é 6.

Se colocarmos uma expressão desse tipo dentro de um loop, podemos gerar sequências numéricas:

```
final = int(input('Digite o valor final da sequência: '))
numero = 1

while numero <= final:
  print(numero)
  numero = numero + 1
```

O programa acima pede para o usuário digitar um número, que será o valor final da sequência. Então ele irá imprimir a variável **numero**, que vale 1, e somar +1 nela. Em seguida imprimirá de novo a variável, agora valendo 2, e somará +1 nela. E assim sucessivamente até que ela ultrapasse o valor final, quando o *loop* deixará de ser executado.

Você consegue modificar o programa acima para fazer uma sequência decrescente? E para gerar a tabuada de um número dado pelo usuário? Você precisará mexer na expressão lógica do loop e no incremento de **numero**.

> Em expressões onde uma variável aparece de ambos os lados, podemos utilizar uma abreviação. Por exemplo, a expressão `x = x + 5` Pode ser reescrita como: `x += 5` Isso vale para todas as outras expressões aritméticas (subtração, multiplicação, divisão etc.).

2. Comandos de manipulação de fluxo

É possível manipular de algumas maneiras a forma como uma malha de repetição se comporta: nós podemos interromper sua execução sem que sua condição de parada tenha sido atingida e podemos saltar para o próximo passo sem finalizar o atual.

2.1. Break

Vamos montar um exemplo simples: imagine que você irá fazer um joguinho onde o usuário terá 10 tentativas para adivinhar um número secreto. Um bom primeiro passo seria criar um loop que conta as 10 tentativas:

```
numero_secreto = 42

contador = 0

while contador < 10:
  tentativa = int(input('Adivinhe o número secreto: '))
```

```
  if tentativa == numero_secreto:
    print('Acertou')
  else:
    print('Errou')
  contador += 1
```

No momento, mesmo que o usuário acerte, ele irá contar até a décima tentativa. Com o que já aprendemos até o momento, poderíamos consertar isso colocando uma segunda condição de parada:

```
numero_secreto = 42

contador = 0

tentativa = 0

while contador < 10 and tentativa != numero_secreto:
  tentativa = int(input('Adivinhe o número secreto: '))
  if tentativa == numero_secreto:
    print('Acertou')
  else:
    print('Errou')
  contador += 1
```

Se você executar o programa, verá que ele funciona: caso o usuário acerte **ou** 10 tentativas sejam feitas, o programa encerra sua execução. Mas note que o código ficou um pouquinho mais bagunçado: estamos testando duas vezes o valor de `tentativa`: na condição do `while` e na condição do `if`. Não seria mais prático dentro do próprio `if`, logo após informar para o usuário que ele acertou, se a gente já pudesse falar para o loop parar de ser executado?

É aí que entra o **break**: quando estamos em uma malha de repetição e encontramos o comando **break**, a malha é interrompida imediatamente. Podemos reescrever o programa acima utilizando esse comando:

```
numero_secreto = 42

contador = 0

while contador < 10:
  tentativa = int(input('Adivinhe o número secreto: '))
  if tentativa == numero_secreto:
    print('Acertou')
    break
```

```
  print('Errou')
  contador += 1
```

Alguns programadores utilizam `while True:` (ou seja, um loop a princípio infinito) e no corpo do loop espalham combinações de `if` + `break`. Exceto em situações muito específicas e raras, isso é uma má prática e deve ser evitada, pois compromete bastante a legibilidade do código, e consequentemente sua manutenção no futuro.

2.2. Else

Você pode observar que no programa acima, respondemos "Errou" para cada chute errado do usuário. Mas o programa ainda não informou para ele que as tentativas dele se esgotaram. Temos algumas possibilidades aqui!

Uma delas seria testar o valor do contador no final do loop:

```
numero_secreto = 42

contador = 0

while contador < 10:
  tentativa = int(input('Adivinhe o número secreto: '))
  if tentativa == numero_secreto:
    print('Acertou')
    break
  print('Errou')
  contador += 1
  if contador == 10:
    print('Acabaram as tentativas. Você perdeu.')
```

Outra seria utilizar uma **flag**: uma variável booleana que indica se entramos no `if` ou não:

```
numero_secreto = 42

contador = 0

acertou = False

while contador < 10:
  tentativa = int(input('Adivinhe o número secreto: '))
  if tentativa == numero_secreto:
    acertou = True
    break
```

```
    print('Errou')
    contador += 1

if acertou:
    print('Acertou!')
else:
    print('Acabaram as tentativas. Você perdeu.')
```

Na maioria das linguagens de programação, teríamos que optar por uma dessas alternativas. Comandos que estudamos aqui em Python são comuns a várias linguagens diferentes, incluindo o par `if`/`else`, o `while` e o `break`.

Mas o Python possui uma ferramenta adicional bastante incomum, mas que pode simplificar problemas desse tipo. Ele permite a utilização de um `else` para um loop. A estrutura deve ser a seguinte:

```
while (condicao_principal):
    ...
    ...
    if (condicao_secundaria):
        break
    ...
    ...
else:
    ...
```

Esse código funcionará da seguinte maneira: se o loop parar pela condição principal (ou seja, executou a quantidade "correta" de repetições), o **else** será executado. Se o loop parar pela condição secundária (ou seja, por conta de um **break**), o **else** será ignorado.

Sendo assim, podemos reescrever nosso programa principal de maneira mais *pythonica* utilizando esse recurso:

```
numero_secreto = 42

contador = 0

while contador < 10:
    tentativa = int(input('Adivinhe o número secreto: '))
    if tentativa == numero_secreto:
        print('Acertou!')
        break
    print('Errou')
    contador += 1
```

```
else:
  print('Acabaram as tentativas. Você perdeu.')
```

Execute o programa e veja que ele só irá mostrar a mensagem de derrota quando as 10 tentativas forem concluídas.

2.3. Continue

Existe outro comando de desvio de fluxo de malhas de repetição: o `continue`. A diferença entre ele e o `break` é que o `continue` encerra apenas o passo atual de repetição, mas ele não encerra o loop como um todo. Quando executamos esse comando, o loop irá voltar para o topo, testar novamente sua condição de parada, e caso ela não tenha sido atingida, ele iniciará uma nova **iteração** (ou seja, um novo passo em um loop).

Vamos reescrever nosso programa anterior invertendo a verificação para vermos o `continue` em ação.

```
numero_secreto = 42

contador = 0

while contador < 10:
  tentativa = int(input('Adivinhe o número secreto: '))
  contador += 1
  if tentativa != numero_secreto:
    print('Errou!')
    continue
  print('Acertou!')
  break
else:
  print('Acabaram as tentativas. Você perdeu.')
```

Sempre que o usuário errar o chute, o `continue` irá desviar o fluxo de execução de volta para o início do loop, impedindo que as duas últimas linhas do loop sejam executadas. Ou seja, ele não irá dizer "acertou", tampouco executar o `break`.

> **Atenção:** todos os desvios estudados aqui podem ser utilizados, caso contrário, não existiriam. Porém, em certas situações eles podem tornar o código mais confuso. Por exemplo, quando temos diversos loops aninhados, pode não ficar claro para alguém lendo o código qual dos loops está sendo encerrado. Sempre que for utilizar esses recursos, verifiquem se eles estão melhorando ou piorando a legibilidade do código.