

Comando condicional

É comum que, em um algoritmo, seja necessária a tomada de decisões baseadas em valores contidos em objetos. Por exemplo, considere um algoritmo que tenha dois objetos de tipo inteiro A e B previamente carregados. Caso seja necessário calcular a divisão de A por B e o conteúdo do objeto B for zero, ocorrerá um erro, como pode ser visto no código a seguir. Isso ocorre porque divisões por zero não são permitidas.

Exemplo 3.1 Erro causado por uma divisão por zero

```
>>> A = 16
```

```
>>> B = 0
```

```
>>> R = A / B
```

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module> R = A / B

ZeroDivisionError: division by zero

No caso desse algoritmo, a situação de erro é indesejável, então, é preciso tomar o cuidado de evitá-la. Uma das maneiras de se conseguir isso é utilizar o comando condicional if-else.

Ao utilizá-lo, será necessário formular uma condição cujo resultado será falso ou verdadeiro, e em função desse resultado o programa será escrito de modo a executar diferentes comandos em cada caso. Então, pode-se formular a seguinte ideia: "se B for igual a zero, então apresente a mensagem 'Não é possível calcular a divisão', senão (ou seja, B é diferente de zero) calcule e apresente na tela A / B".

Agora, é preciso escrever isso em Python. Assim, tem-se o exemplo a seguir, em que é feita a leitura dos objetos A e B utilizando-se o comando input.



Exemplo 3.2 Uma possível maneira de evitar o erro mostrado no Exemplo 3.1

```
A = int(input("Digite um valor para A: ")) # linha 1
```

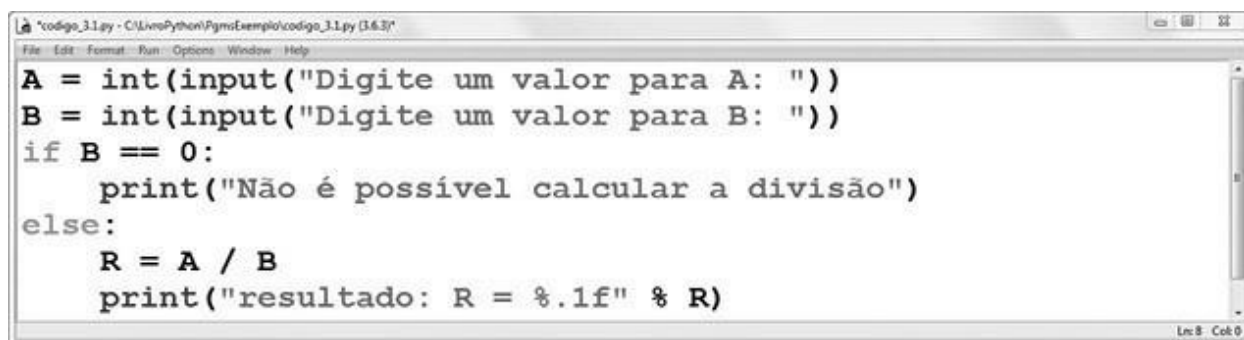
```
B = int(input("Digite um valor para B: ")) # linha 2 if B == 0: # linha 3
```

```
print("Não é possível calcular a divisão") # linha 4 else: # linha 5
```

```
R = A / B # linha 6
```

```
print("resultado: R = %.1f" % R) # linha 7
```

Teste esse pequeno programa no Python escrevendo-o na forma de script. Para isso, abra o Python e acione o comando do menu *File* → *New File*, escreva o programa exibido no Exemplo 3.2, salve-o e execute-o (para executá-lo, utilize o comando *Run* ou a tecla de atalho F5). A Figura 3.1 mostra como ficará esse programa após sua digitação no editor do Python.



```
"codigo_3.1.py - C:\Users\Python\PgmExemplo\codigo_3.1.py (3.6.3)"
File Edit Format Run Options Window Help
A = int(input("Digite um valor para A: "))
B = int(input("Digite um valor para B: "))
if B == 0:
    print("Não é possível calcular a divisão")
else:
    R = A / B
    print("resultado: R = %.1f" % R)
```

Uso do comando condicional if-else.

Na execução do Exemplo 3.2, caso seja fornecido o valor zero para B, será apresentada a mensagem “Não é possível calcular a divisão” e, caso B seja diferente de zero, então, será apresentado o resultado do cálculo. Rode algumas vezes esse programa, testando-o com diferentes valores para A e B.

Explicando em detalhes o Exemplo 3.2

Nas linhas 1 e 2, é feita a leitura dos objetos A e B, de modo que qualquer valor inteiro pode ser inserido para qualquer um deles.

Na linha 3 há o comando if (se) e sua condição. Nessa condição, a pergunta que se está fazendo é se o conteúdo de B é igual a zero. Para isso, foi utilizado o



operador relacional “==”, que avalia se B que está do lado esquerdo contém um valor igual a zero, que está do lado direito. Essa condição será avaliada pelo processador e um resultado será gerado. Esse resultado pode ser falso ou verdadeiro. Caso seja verdadeiro, o programa seguirá para a linha 4 e executará o print. Caso seja falso, o programa desviará (pulará) a linha 4 e seguirá para a execução das linhas 6 e 7, que estão subordinadas ao else (senão) da linha 5.

Note que há um caractere “:” (dois-pontos) no final das linhas 3 e 5. Em Python, é obrigatória a inserção desse caractere no comando if-else, pois é por meio dele que o interpretador Python identifica o término do cabeçalho do comando e o início dos comandos que lhe estão subordinados.

Essa relação de subordinação é importante na lógica do algoritmo. Nesse exemplo, a linha 4 está subordinada ao if da linha 3 e as linhas 6 e 7 estão subordinadas ao else da linha 5.

Identação

O interpretador Python identifica a relação de subordinação descrita no parágrafo anterior pelo recuo que há na digitação das linhas do programa. Note que as linhas subordinadas estão digitadas com alguns espaços em branco à esquerda. Isso recebe o nome de identação e, em Python, ela é obrigatória sempre que houver um ou mais comandos subordinados a outro. O else, por sua vez, não é identado, e para que o programa fique correto é preciso que ele fique exatamente no mesmo alinhamento do if ao qual está associado.

Generalizando, em Python, todo conjunto de comandos subordinados deve estar identado em relação ao seu comando proprietário. Isso vale para if-else, while, for, try, def e qualquer outro em que exista a relação de subordinação.

Construindo condições simples

No exemplo anterior, foi construída uma condição que avaliava se um valor contido no objeto B era igual a zero ou não. Para isso, utilizou-se a construção `B == 0`, onde B é um objeto e 0 é um número literal. A construções desse tipo dá-se o nome de condições simples.

Generalizando, as condições podem ser escritas da seguinte maneira:



{Expressão esquerda} {operador} {Expressão direita} em que as **expressões** em ambos os lados podem ser:

- um literal (geralmente número ou texto);
- um objeto;
- uma fórmula (expressão aritmética);
- uma chamada de função (ver Capítulo 5).

O **operador** é um dos seis operadores relacionais exibidos no Quadro 3.1. No caso dos operadores que contêm dois caracteres, não é permitido haver espaço em branco entre eles.

Operador	O que ele faz
= =	Igual a
!=	Diferente de
<	Menor que
<=	Menor ou igual a
>	Maior que
>=	Maior ou igual a

Quadro 3.1 Operadores relacionais.

O Quadro 3.2 exemplifica a construção de condições simples.

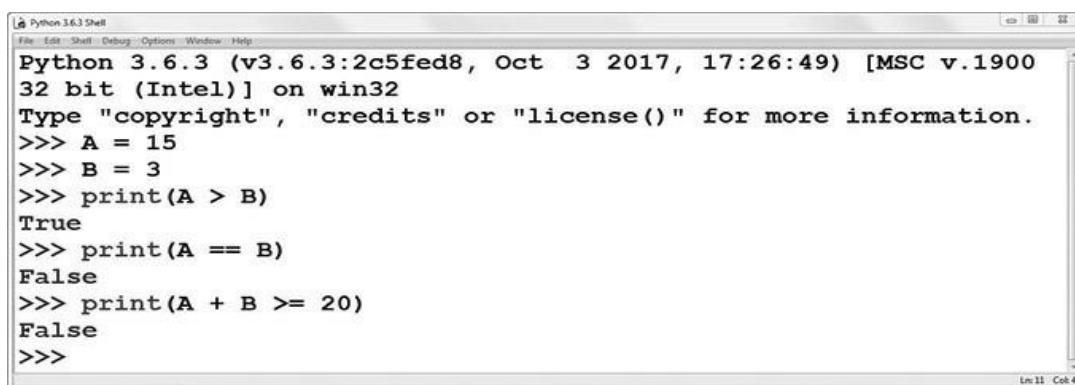
Condição	Interpretação	Elementos envolvidos
$A > 0$	A maior que zero	Compara objeto com literal numérico (0).
$X \leq Y$	X menor ou igual a Y.	Compara dois objetos.
$X \neq A + B$	X é diferente de A + B	Compara objeto com o resultado da expressão aritmética.
$C > 2 * (A + B)$	C é maior que $2(A + B)$	Compara os resultados de duas expressões aritméticas.



10*A < 100*B	10A é maior que 100B	Compara os resultados de duas expressões aritméticas.
S == ""	S igual a string vazio	Compara objeto com literal texto vazio.
S != "SIM"	S diferentes de "SIM"	Compara objeto com literal texto não vazio.

Quadro 3.2 Exemplos de condições simples.

No IDLE, atribua valores aos objetos sugeridos nos exercícios e utilize o comando print para exibir o resultado produzido pela condição, conforme exemplificado.



```
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> A = 15
>>> B = 3
>>> print(A > B)
True
>>> print(A == B)
False
>>> print(A + B >= 20)
False
>>>
```

Exemplo de testes de condições simples.

Construindo condições compostas

Muitas vezes, é preciso negar uma condição simples ou combinar duas ou mais condições simples em uma condição composta. Assim, as condições simples vistas anteriormente são a base para a construção desse novo tipo de condição.

A construção de uma condição composta tem uma das seguintes formas: not {Condição 1} {Condição 1} {Operador Lógico and/or} {Condição 2} em que as condições 1 e 2 são duas condições simples, como as que já foram vistas no Item 3.1.3.

O operador lógico é um dos operadores apresentados no Quadro. Antes de seguir adiante, é preciso saber como avaliar expressões que contenham esses operadores not, and e or.

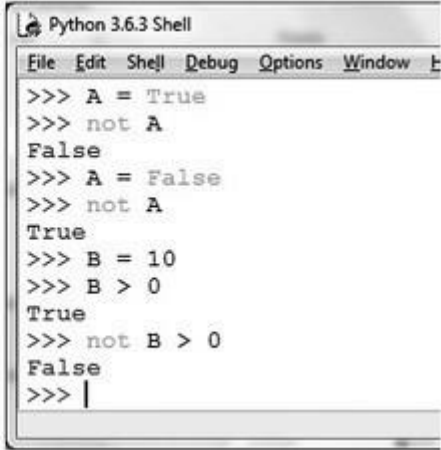


Operador	O que ele faz	Descrição
not	Negação	Nega a condição à qual é aplicado.
and	Conjunção operação lógica E	Resultado verdadeiro se forem verdadeiras as duas condições às quais é aplicado.
or	Disjunção operação lógica OU	Resulta verdadeiro se for verdadeira pelo menos uma das duas condições às quais é aplicado.

Quadro - Operadores lógicos.

A Figura traz a forma de avaliação do operador lógico not e um exemplo de uso.

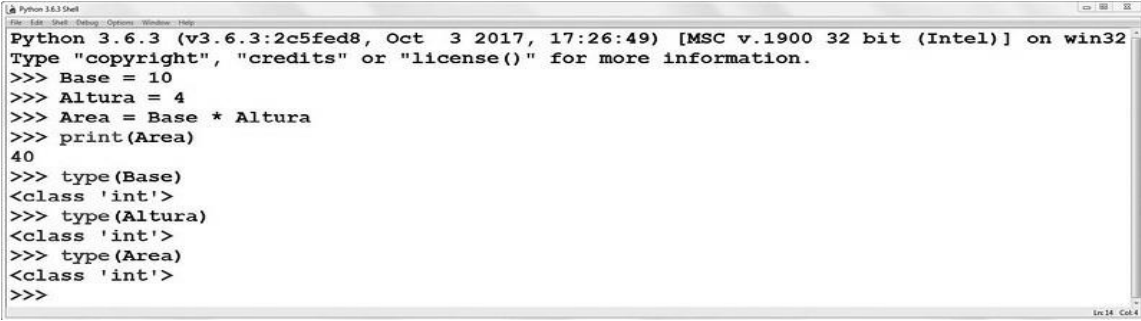
Tabela Verdade do operador not not C1 produz a negação do resultado de C1	
Condição C1	not C1
False	True
True	False



```
>>> A = True
>>> not A
False
>>> A = False
>>> not A
True
>>> B = 10
>>> B > 0
True
>>> not B > 0
False
>>> |
```

Exemplo de uso do operador lógico not.

A Figura mostra a forma de avaliação do operador lógico and e um exemplo de uso.



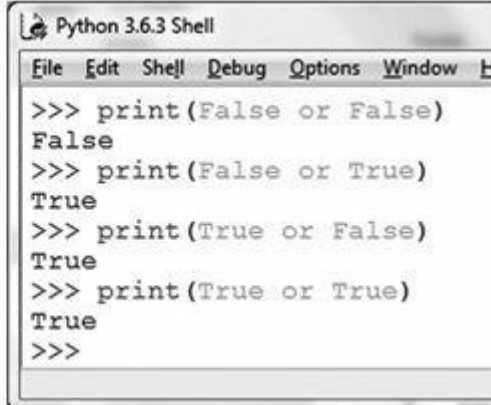
```
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Base = 10
>>> Altura = 4
>>> Area = Base * Altura
>>> print(Area)
40
>>> type(Base)
<class 'int'>
>>> type(Altura)
<class 'int'>
>>> type(Area)
<class 'int'>
>>>
```

Exemplo de uso do operador lógico and.



Já a Figura 3.5 traz a forma de avaliação do operador lógico or e um exemplo de uso.

Tabela Verdade do operador or		
Para que or resulte verdadeiro pelo menos um dos dois, C1 ou C2, deve ser verdadeiro		
Condição C1	Condição C2	C1 or C2
False	False	False
False	True	True
True	False	True
True	True	True



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window H
>>> print(False or False)
False
>>> print(False or True)
True
>>> print(True or False)
True
>>> print(True or True)
True
>>>
```

Exemplo de uso do operador lógico or.

Condição	Interpretação
A > 0 and B > 0	O resultado da condição será verdadeira se A e B forem ambos iguais a zero.
X <= Y and Y != 0	O resultado da condição composta será verdadeiro somente se X for menor ou igual Y, ao mesmo tempo que Y seja diferente de zero.
X == 0 or X > 2000	O resultado da condição composta será verdadeiro se X for igual a zero ou se X for maior que 1000.
A < 0 or B < 0	O resultado da condição composta será verdadeiro se pelo menos um dos objetos A e B contiver valor negativo.
not (X == 2)	O resultado da condição composta será verdadeiro se X for diferente de 2. Equivale a X != 2.

Quadro - Exemplos de condições compostas e sua interpretação.

Condições compostas mistas

Em uma única condição composta é possível misturar not, and e or. Quando isso ocorre, é necessário ter atenção à precedência com que esses operadores são considerados. Existe uma ordem de prioridade a ser respeitada. Essa prioridade obedece à seguinte ordem: not primeiro, and em seguida e or por último.

Assim, na avaliação de uma condição composta mista, a prioridade supracitada



sempre será seguida. É necessário ter o devido cuidado ao construir condições assim e verificar que a falta de atenção pode levar a erros. Como exemplo, veja as duas expressões a seguir e avalie-as para $A = 15$, $B = 9$, $C = 9$.

$B == C$ or $A < B$ and $A < C$ Resultará Verdadeiro ($B == C$ or $A < B$) and $A < C$ Resultará Falso

O uso de parênteses serve para alterar a ordem de prioridade na avaliação de expressões lógicas. Uma vez inseridos, os parênteses estabelecem qual(is) parte(s) serão avaliada(s) primeiro.

Comando condicional completo

Retomando agora as explicações sobre o comando condicional, a seguir está sua forma completa.

```
if {condição 1}:  
    {bloco de comandos 1} elif {condição 2}:  
    {bloco de comandos 2}  
    elif {condição 3}:  
    {bloco de comandos 3}  
  
...  
else:  
    {bloco de comandos do else}
```

As partes if e else já foram explicadas anteriormente. A parte elif permite que sejam utilizadas condições adicionais e confere a possibilidade de tomada de decisão entre múltiplas opções.

A execução desse comando inicia pela avaliação da {condição 1}, e se ela for verdadeira será executado o {bloco de comandos 1} e pulam-se todos os demais; caso a {condição 1} seja falsa, passa-se para a avaliação da {condição 2} e, caso seja verdadeira, será executado o {bloco de comandos 2}, pulando-se os demais, e assim sucessivamente. Ao final, se nenhuma das condições postas for verdadeira, então executa-se o {bloco de comandos do else}.

Não há limites para a quantidade de partes elif a serem utilizadas, de modo que o programador é livre para utilizar tantas dessas partes quanto for a necessidade do algoritmo.

E, por fim, é preciso dizer que as partes elif e else são opcionais, de modo que,



se o programador não precisar incluí-las em seu algoritmo, elas podem simplesmente ser omitidas.

Exemplo 3.3 Uso da forma completa if-elif-else PH = float(input("Digite um valor do PH: ")) if PH < 7.0:

```
print("Solução ácida")
```

```
elif PH == 7.0:
```

```
print("Solução neutra") else:
```

```
print("Solução básica")
```

No Exemplo 3.3 é feita a leitura de um número real que representa o valor de pH de uma solução, para o qual há três possibilidades, segundo a química. Caso seja menor que 7,0, a solução é ácida; caso seja igual a 7,0, é neutra; e caso seja maior que 7,0, ela é básica. É exatamente isso que está implementado no código desse exemplo, em que foi empregada a construção if-elif-else para implementá-la.

Comandos condicionais aninhados

É frequente que existam situações em que é necessário colocar um if dentro de outro if. Isso pode ser feito normalmente, conforme mostrado no Exemplo 3.4. O único cuidado é que se respeite a indentação para que a relação de subordinação entre os comandos fique correta.

O que se deseja fazer no Exemplo 3.4 é que três números sejam carregados em A, B e C, e o programa deve mostrá-los em ordem crescente. Na solução, o primeiro if decide se A é o menor; caso seja, então, há um if aninhado a ele para decidir dentre os outros dois, B e C, qual é o menor. Se A não for o menor, então, o elif do primeiro if decide se B é o menor dos três e, caso seja, há um if aninhado a ele para decidir quem é o menor entre A e C. Por fim, o else do primeiro if será executado caso o menor seja C, e aí há um if aninhado para decidir quem é o menor entre A e B. Teste esse programa para as seis combinações possíveis de valor caso os três valores sejam diferentes entre si. Essas combinações são mostradas no Quadro 3.5. Em todos esses casos, o programa deve exibir na tela a mensagem: "Ordem crescente: 1, 2, 3".



Para os testes, convida-se o leitor a montar outras combinações em que dois valores sejam iguais e o terceiro seja diferente. Para isso, use os vazios do Quadro

A	1	1	2	3	2	3
B	2	3	1	1	3	2
C	3	2	3	2	1	1

3.5.

Quadro 3.5 Possibilidades de valores para testar o Exemplo 3.4.

Exemplo 3.4 Comandos condicionais aninhados

```
A = int(input("Digite um valor para A: "))
```

```
B = int(input("Digite um valor para B: "))
```

```
C = int(input("Digite um valor para C: "))
```

```
if A <= B and A <= C: # A é o menor dos três
    if B <= C: # decide quem é o menor entre B e C
```

```
        print("Ordem crescente: {}, {}, {}".format(A, B, C))
    else:
```

```
        print("Ordem crescente: {}, {}, {}".format(A, C, B))
elif B <= A and B <= C: # B é o menor dos três
```

```
    if A <= C: # decide quem é o menor entre A e C
```

```
        print("Ordem crescente: {}, {}, {}".format(B, A, C))
    else:
```

```
        print("Ordem crescente: {}, {}, {}".format(B, C, A))
```

```
else: # C é o menor dos três (opção que sobrou, por isso else)
    if A <= B: # decide quem é o menor entre A e B
        print("Ordem crescente: {}, {}, {}".format(C, A, B))
```

```
    else:
```

```
        print("Ordem crescente: {}, {}, {}".format(C, B, A))
```



Comando de repetição

É muito frequente que, para implementar determinada lógica, um programador precise repetir um trecho de programa um certo número de vezes. Isso pode ser realizado com o uso do comando de repetição `while`. Com ele, é possível repetir um conjunto de comandos enquanto uma condição especificada for verdadeira. Esse tipo de trecho repetitivo de código também é conhecido como laço ou, em inglês, como *loop*.

O comando `while` em Python tem a construção básica a seguir, que pode ser interpretada como: “enquanto a condição for verdadeira, execute o conjunto de comandos”.

```
while {condição}:
```

```
{conjunto de comandos}
```

A condição segue exatamente as mesmas regras utilizadas nas condições já vistas quando foi abordado o comando `if-else`. Quanto ao conjunto de comandos subordinados ao `while`, podem ser quaisquer comandos válidos em Python, em quaisquer quantidade e extensão. Assim como no comando `if-else`, a indentação é importante, pois define a relação de subordinação entre o cabeçalho do comando e seu conjunto subordinado.

Para exemplificar a implementação de um laço, tem-se o código a seguir, no qual se quer exibir na tela todos os números inteiros entre 1 e 10, sendo um valor em cada linha.

Exemplo 3.5 Funcionamento do comando de repetição – laço contador
`print(“Início do Programa”)`

```
Cont = 1 # linha 1
```

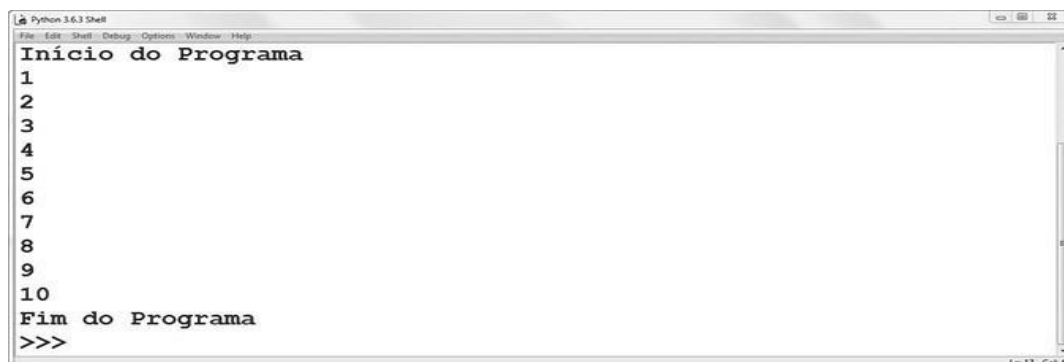
```
while Cont <= 10: # linha 2 print(Cont) # linha 3
```

```
Cont = Cont + 1 # linha 4 print(“Fim do Programa”)
```

No Exemplo 3.5, na linha 1 é definido um objeto inteiro identificado por `Cont` e inicializado com o valor 1. Em seguida – linha 2 –, tem-se o comando `while` construído com a condição `Cont <= 10`. A avaliação dessa condição resulta em *True* (verdadeiro), de modo que o conjunto de comandos subordinado, constituído pelas linhas 3 e 4, é executado uma primeira vez. Com isso, o valor inicial de `Cont` é exibido e 1 é somado



a Cont, que passará a ser 2. Após a execução da linha 4, o programa retorna para a linha 2, a condição é avaliada e novamente resultará *True*, pois Cont é menor



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Início do Programa
1
2
3
4
5
6
7
8
9
10
Fim do Programa
>>>
```

que 10. Isso fará que o print e a soma de 1 em Cont sejam executados uma segunda vez. Com isso, Cont passará a conter o valor 3 e o programa seguirá sucessivamente. Ao final, dez linhas terão sido exibidas na tela contendo os valores de 1 a 10.

Execução do programa do Exemplo 3.5.

Laços como esses são denominados “laços contadores”, pois seu controle é efetuado por meio de um objeto de controle que sofre um incremento a cada repetição. Nem todo laço é contador, como o que está implementado no Exemplo 3.6, descrito a seguir.

Lógica de funcionamento do laço while

No Exemplo 3.5 foi mostrado como utilizar o comando *while*. Trata-se de um comando de laço no qual o teste da condição é feito no início do laço. A Figura 3.7 ilustra essa situação, na qual a avaliação da condição é feita antes de se executar o conjunto de comando subordinado.





Diagrama ilustrativo da sequência de operações do comando de repetição.

Esse conceito é importante, porque, sempre que a condição for previamente falsa, o conjunto subordinado não será executado nenhuma vez, dado que a avaliação da condição é feita antes.

Todo laço, para ser implementado, requer quatro elementos: inicialização, condição, iteração e o corpo. Os três primeiros dizem respeito à construção e ao controle do laço. A inicialização constitui-se de todo código necessário para determinar a situação inicial do laço. A condição é uma expressão lógica, que pode ser simples ou composta, cujo resultado é avaliado em falso ou verdadeiro, que determina se o laço termina ou prossegue, respectivamente. A iteração é todo comando (pode ser um ou mais de um) que modifica os objetos envolvidos na condição, a cada execução do laço. Por fim, o corpo do laço é constituído pelos comandos que devem ser executados repetidas vezes.

No Exemplo 3.5, a inicialização está na linha 1, a condição é a linha 2 e a iteração é a linha 4. O corpo do laço é a linha 3.

As condições usadas neste comando de repetição são exatamente iguais às usadas no comando if-else, vistas no Item 3.1.

No Exemplo 3.6 pede-se que se escreva um programa que permaneça em laço enquanto um valor X lido for diferente de zero. Para cada valor de X deve-se apresentar na tela se o mesmo é par ou ímpar.

Exemplo 3.6a Par ou ímpar X = 1 # linha 1



```
while X != 0: # linha 2
```

```
X = int(input("Digite X: ")) # linha 3 if X % 2 == 0: # linha 4
```

```
print("%d é par" % X) # linha 5 else: # linha 6
```

```
print("%d é ímpar" % X) # linha 7
```

Nessa solução, o controle do laço não é implementado por meio de um contador que permitirá a repetição um certo número conhecido de vezes. Neste caso, não se sabe quantas vezes o laço repetirá. O que se sabe apenas é que termina quando zero for digitado para X.

Para garantir que o laço seja iniciado, o objeto X deve ser criado contendo qualquer valor diferente de 0, o que é feito na linha 1. Essa é a linha de inicialização. A iteração é implementada na linha 3, que altera o valor do objeto X. O novo X lido logo no início do laço também é utilizado em seu corpo, que é constituído pelas linhas 4 a 7. O resto da divisão de X por 2 é calculado e comparado com zero. Se o resultado dessa comparação for verdadeiro, então, o número é par; caso contrário, é ímpar. Quando zero for digitado, o programa dirá que zero é par e terminará.

Outra solução possível para esse problema está implementada a seguir, no Exemplo 3.6b. Nessa segunda solução, a inicialização do objeto X é feita a partir da leitura do teclado. Caso X digitado seja diferente de zero, o laço é iniciado, o corpo do laço é executado e a leitura de um novo valor para X – linha 3 da solução anterior – foi transferida para o final do corpo do laço. Nela, X é lido e, em seguida, o laço retorna para seu cabeçalho para avaliar a condição e decidir se continuará ou não.

Exemplo 3.6b Par ou ímpar

```
X = int(input("Digite X: ")) # linha 1 while X != 0: # linha 2
```

```
if X % 2 == 0: # linha 4 print("%d é par" % X) # linha 5 else: # linha 6
```

```
print("%d é ímpar" % X) # linha 7
```

```
X = int(input("Digite X: ")) # linha 3 (transferida para cá)
```



No próximo exemplo, pede-se que escreva um programa que mostre na tela os dez primeiros termos de uma progressão aritmética (PA) com primeiro termo P e razão R. Os dois números P e R são inteiros e devem ser lidos do teclado.

Exemplo 3.7 Progressão aritmética

```
P = int(input("Digite o primeiro termo: ")) # linha 1
R = int(input("Digite a razão: ")) # linha 2
Cont = 0 # linha 3

while Cont < 10: # linha 4
    print(P) # linha 5
    P = P + R # linha 6

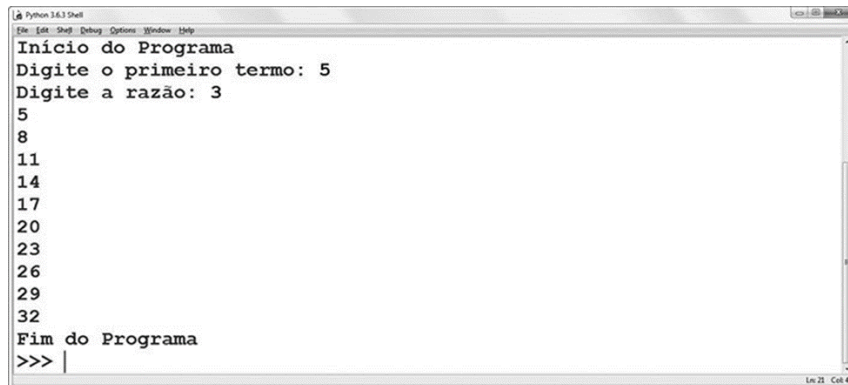
    Cont = Cont + 1 # linha 7
```

Nas linhas 1 e 2 são feitas as leituras dos dados de entrada. Na linha 3 é feita a inicialização do laço, atribuindo-se 0 ao objeto Cont. Na linha 4 está a condição de continuidade do laço que foi construída como $\text{Cont} < 10$. E a iteração é formada pela linha 7, na qual se soma 1 ao Cont. O corpo do laço é formado pelas linhas 5 e 6, nas quais é exibido o conteúdo de P e calculado o próximo termo a ser exibido, somando-se R ao objeto P e armazenando o resultado no próprio objeto P. Com isso, P é preparado para a próxima iteração. Na primeira vez em que o laço é executado será mostrado o primeiro termo da PA, contido em P. Ao somar R em P, este passa a conter o segundo termo. Quando o laço for executado pela segunda vez, será mostrado o segundo termo e calculado o terceiro. Ao mesmo tempo, 1 é somado em Cont a cada repetição, de modo que ele aumentará sempre até atingir o valor 10. Quando isso ocorrer, a condição da linha 4 será avaliada em False e o laço terminará.

Propositalmente, no Exemplo 3.7 o objeto Cont foi inicializado com 0 e a condição foi escrita como $\text{Cont} < 10$. Compare-a com o Exemplo 3.5, em que o objeto Cont foi iniciado com 1 e a condição escrita como $\text{Cont} \leq 10$. São duas maneiras diferentes de implementar um laço que executa o mesmo número de vezes. Em casos assim, a escolha entre uma forma ou outra depende apenas do que o programador considerar mais apropriado.



Ao executar esse programa para um caso de teste em que $P = 5$ e $R = 3$, tem-se o resultado mostrado na Figura.



Resultado da execução do Exemplo 3.8.

No Exemplo 3.8, vamos escrever um programa que permaneça em laço enquanto um valor X lido for diferente de zero. Totalize (some todos) e conte os valores digitados, exceto o zero, e apresente esses valores na tela. Use o caso de teste a seguir para verificar se o programa está correto:

Entrada 6 8 -30 9 -4 8 16 50 15 7 2 -5 0

Saída Total dos valores digitados = 82
Quantidade de valores = 12

Aspectos específicos dos comandos de repetição em Python

Até agora, o objetivo foi apresentar os conceitos básicos e gerais relativos a laços. Tais conceitos aplicam-se à maioria das linguagens de programação. A seguir, serão abordados alguns aspectos específicos que se aplicam à linguagem Python.

Existem dois comandos em Python que são utilizados no controle do fluxo de execução de laços. Estes comandos são `continue` e `break`, e só têm significado se estiverem inseridos no bloco de comandos subordinado a um laço. Podem ser usados, tanto em laços `while`, quanto em laços `for` (que serão vistos no Capítulo 4) e em nenhum outro comando de Python.

`continue`

Encerra a iteração atual e desvia a execução do programa para o cabeçalho do `while` em que esteja inserido. Nova avaliação da condição será feita, e terá início uma nova iteração, caso a mesma seja verdadeira. No Exemplo 3.9, caso o valor lido



para o objeto X seja menor ou igual a zero, a condição da linha 4 será True e o continue desviará a execução de volta para a linha 2, sem executar o bloco de comandos que inicia na linha 6.

Exemplo 3.9 Uso do comando continue X = 1

```
while X != 0: # linha 2
```

```
    X = int(input("Digite um valor: ")) if X <= 0: # linha 4  
        continue
```

```
    # bloco de comandos # linha 6
```

```
break
```

Este comando termina o laço e desvia a execução para fora do mesmo. No Exemplo 3.10 foi criado um laço while True:, que é sempre verdadeiro e, por consequência, executará indefinidamente. Porém, na linha 4 há um comando break que encerrará esse laço quando X for igual a zero.

Exemplo 3.10 Uso do comando break

```
while True: # linha 1 – a condição é sempre True X = int(input("Digite um valor:  
"))
```

```
    if X == 0: # linha 3
```

```
        break # linha 4 - encerra o laço # bloco de comandos # linha 5
```

else em laços do Python

Outro aspecto a ser abordado aqui é o bloco else contido nos comandos de repetição de Python.

Esse é um recurso que causa muita estranheza nos programadores que já conhecem outras linguagens de programação. Todas as linguagens têm else associado ao comando if, para implementar o conceito de que “Se a condição for verdadeira faça isso, senão faça aquilo”. O else, nesse caso, é uma contraposição ao if.



Assim sendo, qual seria a função do `else` no comando `while`? O bloco de comandos subordinado ao `else` é executado se, e somente se, o `while` terminar normalmente, pelo fato de sua condição de continuidade se tornar falsa. Caso o `while` seja encerrado por um comando `break`, então, o `else` não é executado. Assim, esse bloco `else` implementa o conceito “Repita o laço e, quando este terminar normalmente, então, execute o `else`”. O `else`, neste caso, representa justamente o oposto do que significa a palavra “senão”.

Ramalho (2015) sugere que a escolha da palavra `else` foi infeliz e que a palavra `then` (então) teria sido uma opção melhor. No entanto, a palavra utilizada para esse bloco é `else` e não será alterada, de modo que o programador Python precisa se acostumar a essa ideia.

Em função da estranheza mencionada, muitos programadores, que construíram seu conhecimento de programação usando outras linguagens, subestimam esse recurso e tendem a não o utilizar, porém, essa não é a melhor das decisões. Seu uso torna o código mais simples e legível e evita a frequente situação em que é necessário criar objetos de sinalização (*flags*) e condições associadas para controlar se algo aconteceu ou não dentro de um laço e produzir um resultado.

O Exemplo 3.11 ilustra o uso do bloco `else` no comando `while`. Para que tenha uma ideia bem clara desse uso, serão desenvolvidas duas soluções, uma sem utilizar o bloco `else` e outra utilizando-o.

No Exemplo 3.11, escreva um programa que leia um número inteiro N e exiba na tela se ele é ou não primo.

Números primos são aqueles divisíveis apenas por 1 e por eles mesmos. Em termos práticos, visando escrever um programa que resolva esse problema, pode-se dizer que N não é divisível por nenhum valor contido no intervalo fechado $[2, N-1]$. A solução que será apresentada é a mais simples possível e também a menos eficiente. O objetivo aqui é um melhor entendimento por parte dos iniciantes, de modo que se pede licença aos mais experientes para que aceitem essa solução.

A solução consiste em criar um laço no qual se calcule o resto da divisão de N por todos os valores do intervalo $[2, N-1]$ e contar quantas vezes ocorreu resto igual a zero. Caso o laço termine com a contagem igual a zero, então, o número é primo, caso contrário, ele não é.

Exemplo 3.11 Verificar se um número é primo (versão A) $N = \text{int}(\text{input}(\text{"Digite"}$



```
N: ")))
```

```
Cont = 0
```

```
i = 2
```

```
while i < N:
```

```
    R = N % i if R == 0:
```

```
        Cont += 1
```

```
    i += 1
```

```
if Cont == 0: # linha 9 print("{} é primo".format(N)) else:
```

```
    print("{} não é primo".format(N))
```

Nessa solução, o objeto Cont é empregado como *flag*. Começa com zero e para toda ocorrência de R igual a 0 tem seu valor incrementado. Após o término do laço o if da linha 9, verifica o valor de Cont e exibe a mensagem apropriada.

Compare essa solução com a versão B. Ambas produzem o mesmo resultado, mas essa segunda versão é mais enxuta. Verifique-se que nela não existe o objeto Cont, ou seja, o flag tornou-se dispensável, e também não existe mais o if posterior ao comando while.

E não é só isso. Perceba que a segunda versão se tornou mais eficiente para os casos em que N não é primo, pois basta encontrar um primeiro resto igual a zero que o laço é encerrado com break. Caso todas as divisões sejam feitas, a condição torna-se falsa quando i chega ao valor de N, e nesse caso a execução é desviada para o else, que exibe que o número é primo.

Exemplo 3.12 Verificar se um número é primo (versão B) N = int(input("Digite N: "))

```
i = 2
```

```
while i < N:
```

```
    R = N % i
```



```
if R == 0: # ao encontrar o primeiro R == 0

    print("{} não é primo".format(N)) # exibe que não é primo break # e encerra o
    laço while
    i += 1

else:

    print("{} é primo".format(N))
```

do-while não existe

Por fim, no estudo dos aspectos específicos do comando `while` em Python existe a questão da não existência de um comando de laço que se assemelhe ao `do-while` da linguagem C e está presente em boa parte das linguagens de programação.

Nota

Conforme visto, no comando *while* a condição é testada no início e, caso seja verdadeira, o bloco de comandos é executado. Outra possibilidade é a existência de um comando de laço no qual a avaliação da condição seja feita após a execução do conjunto de comandos subordinado.

Em outras linguagens de programação, como C e Java, existe um segundo tipo de comando de laço, no caso é o *do-while*, no qual a avaliação da condição é feita **APÓS** a execução dos comandos subordinados.

Isso implica que o conjunto de comandos subordinado obrigatoriamente será executado pelo menos uma vez, mesmo que a condição de continuidade do laço seja previamente falsa. Isso é útil em algumas situações.

Em Python não existe essa opção.

Essa questão foi motivo de intenso debate na comunidade Python mundial. Conforme mencionado no Capítulo 1, essa comunidade é muito ativa e mantém o índice de PEPs (*Python Enhancement Proposal*), que é um banco de dados criado para, como o próprio nome diz, reunir todas as propostas de melhoria da linguagem Python.



Pois bem, a PEP 315 (ROSSUM, PEP-315) trata exatamente dessa questão, em que é feita a sugestão de criar uma alteração na sintaxe do comando while com o objetivo de que ele também pudesse ser utilizado, à semelhança de do- while (entenda-se: executar primeiro o bloco de comandos para depois verificar a condição).

Essa PEP já está encerrada e foi rejeitada. Uma mensagem enviada pelo próprio Guido von Rossum (ROSSUM, PEP315) recomenda a rejeição indicando que implementar tal comando não traria um benefício direto para a linguagem nem a tornaria mais legível ou fácil de se aprender. Além disso, tal resultado pode ser obtido com a seguinte estrutura:

```
while True: # este laço sempre inicia
```

```
<código> # executa o código do laço
```

```
if condição: # testa a condição e se for True break # termina o laço
```

O comportamento desse laço é exatamente o que se espera de um do-while em C, então, a questão está resolvida.

Tratamento de exceções

O termo “exceção” em programas de computador diz respeito a uma situação em que algum evento ocorrido durante a execução do programa necessita de atenção e tratamento apropriado. Nessas situações, diz-se que a exceção precisa ser “tratada”.

O básico sobre exceções e seu tratamento

No Exemplo 3.1 foi mostrada uma situação em que uma mensagem de erro foi emitida indicando a ocorrência de uma situação de divisão por zero. Situações como essas são delicadas, pois fazem que o programa seja abortado no momento em que ocorre, se não houver um tratamento. No Exemplo 3.2 foi elaborada uma solução baseada em um comando if que evita o erro e cria uma alternativa a sua ocorrência, no caso, a exibição de uma mensagem.

Ocorre que, nos tempos atuais, os programas têm ficado cada vez maiores e mais complexos, de modo que usar condições para prever tudo o que pode dar errado em um programa torna-o muito mais complexo que o necessário e não garante que



todas as possibilidades foram previstas e cobertas.

Assim, surgiu no campo da Ciência da Computação a necessidade de se desenvolver um modo mais racional, organizado e bem estruturado de lidar com essas situações. A solução desenvolvida para dar atendimento a tal demanda é o **tratamento de exceções**. Trata-se de um recurso disponível na maioria das linguagens modernas, muito inteligente, poderoso e ao mesmo tempo simples.

O Exemplo 3.13 mostra como fica o código do Exemplo 3.1 caso se adote o tratamento de exceções como forma de resolver a questão do erro de divisão por zero. Ou seja, esse exemplo é uma alternativa à solução apresentada no Exemplo 3.2.

Exemplo 3.13 Tratamento de exceção

```
A = int(input("Digite um valor para A: "))  
  
B = int(input("Digite um valor para B: ")) try: # linha 3  
R = A / B  
  
print("resultado: R = %.1f" % R) except: # linha 6  
print("Não é possível calcular a divisão")
```

O comando try inicia o bloco de comandos que estará protegido pelo tratamento de exceções e a cláusula except contém o código que será executado em caso de erro.

Teste a execução desse programa fornecendo valores, como mostrado na Figura 3.10. Observe que quando B recebeu o valor zero a mensagem de resultado foi omitida (pulada) e foi exibida a mensagem de exceção, pois o fluxo de execução do programa foi desviado do bloco try para o bloco except.




```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
===== RESTART: C:\LivroPython\PgmsExemplo\codigo_3.8.py =====
Digite um valor para A: 15
Digite um valor para B: 3
resultado: R = 5.0
>>>
===== RESTART: C:\LivroPython\PgmsExemplo\codigo_3.8.py =====
Digite um valor para A: 15
Digite um valor para B: 0
Não é possível calcular a divisão
>>> |
```

Uso do tratamento de exceções.

Ampliando as possibilidades – exceções nomeadas

Toda exceção em Python tem um nome identificador, e isso pode ser utilizado pelo programador para fazer distinção entre diferentes tipos de exceções e dar um tratamento próprio a cada uma delas.

Considere-se que no caso do Exemplo 3.14 possam ocorrer outras situações que levem a erros. Se o usuário do programa digitar texto ou um número real no momento da leitura de A ou de B, como está sendo utilizada a função `int()` para converter o dado lido para número inteiro, isso causará erro.

Exemplo 3.14 Outra situação possível e indesejada

```
>>> A = int(input("Digite um valor para A: ")) Digite um valor para A: texto
Traceback (most recent call last):
```

```
File "<pyshell#2>", line 1, in <module> A = int(input("Digite um valor para A: "))
```

```
ValueError: invalid literal for int() with base 10: 'texto'
```

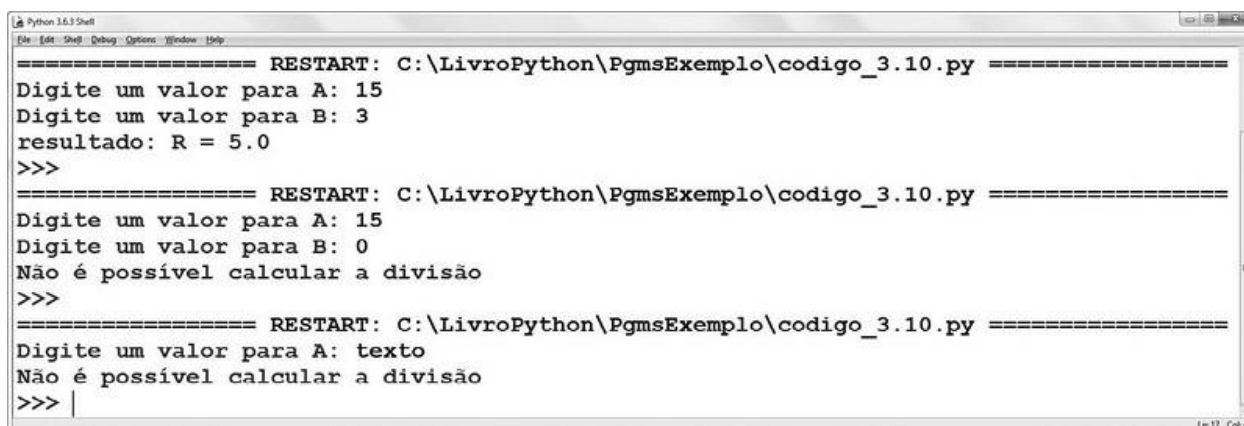
Para evitar tal erro o programa pode ser escrito como mostrado no Exemplo 3.15, que é a mesma solução do Exemplo 3.13, com a única diferença de que as duas linhas de leitura dos objetos A e B foram transferidas para dentro do bloco protegido pelo `try`.

Exemplo 3.15 Ampliando o tratamento de exceção `try`:

```
A = int(input("Digite um valor para A: ")) # a leitura está
```



```
B = int(input("Digite um valor para B: ")) # protegida tb R = A / B
print("resultado: R = %.1f" % R) except: # linha 6
print("Não é possível calcular a divisão")
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
===== RESTART: C:\LivroPython\PgmsExemplo\codigo_3.10.py =====
Digite um valor para A: 15
Digite um valor para B: 3
resultado: R = 5.0
>>>
===== RESTART: C:\LivroPython\PgmsExemplo\codigo_3.10.py =====
Digite um valor para A: 15
Digite um valor para B: 0
Não é possível calcular a divisão
>>>
===== RESTART: C:\LivroPython\PgmsExemplo\codigo_3.10.py =====
Digite um valor para A: texto
Não é possível calcular a divisão
>>> |
```

Ao executar essa solução, as duas situações possíveis de erro já mencionadas provocam o desvio para o bloco except e a mesma mensagem acaba sendo exibida. Nos programas modernos, convém oferecer a seus usuários uma solução mais precisa, em que cada exceção tenha tratamento próprio e diferenciado.

Assim, entram em cena as exceções nomeadas. Observe-se que, no caso anterior, a mensagem exibida no Exemplo 3.14 tem a identificação “ValueError” e o erro do Exemplo 3.1 é identificado por “ZeroDivisionError”. Estes são os nomes das exceções levantadas.

O Exemplo 3.16 mostra como usar esses nomes e tornar o programa mais preciso no tratamento de cada caso. Nas linhas 8 e 10 estão especificadas as exceções “ZeroDivisionError” e “ValueError” respectivamente. Na linha 12 foi mantida a cláusula except genérica (não nomeada) para capturar todas as outras exceções que não foram explicitamente previstas.

Nas linhas 5 e 6 foi propositalmente incluído um if no qual se pretende calcular o cosseno de A, caso A seja negativo. Porém, este cálculo vai falhar, com a exceção “NameError”, pois a biblioteca matemática não foi importada e a função cos() não será reconhecida pelo interpretador. Uma vez que não foi previsto tratamento específico para tal caso, quando isso ocorrer, o programa desviará para a exceção genérica na linha 12.

Não é obrigatório que o programador utilize a exceção genérica, e caberá a ele decidir se tal uso é ou não apropriado, em função das necessidades do programa que



está desenvolvendo. Se não a usar e ocorrer uma exceção fora das previstas, então, o comportamento padrão será executado pelo interpretador e o programa será abortado.

Exemplo 3.16 Uso de exceções nomeadas try:

```
A = int(input("Digite um valor para A: "))
```

```
B = int(input("Digite um valor para B: "))
```

```
R = A / B
```

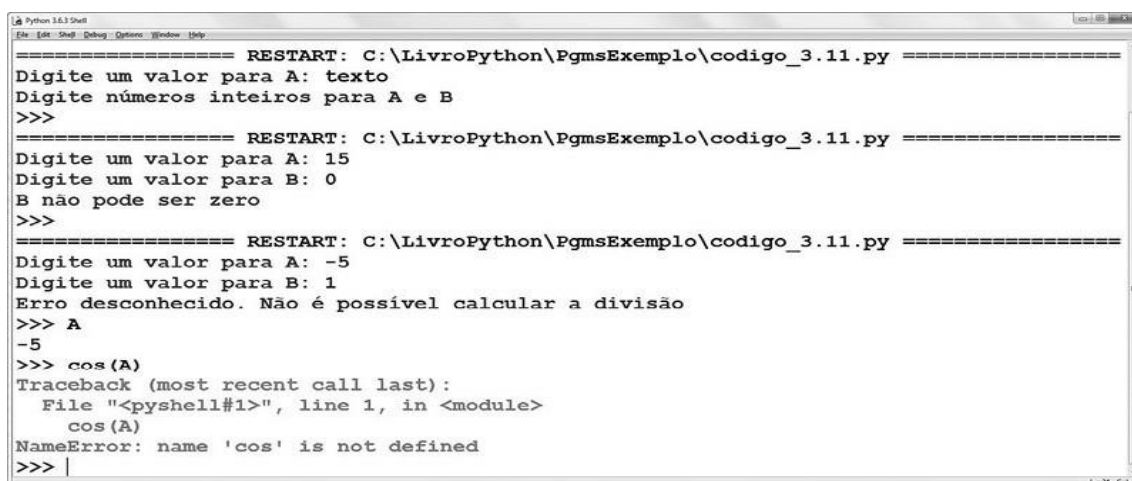
```
if A < 0: # linha 5 C = cos(A)
```

```
print("resultado: R = %.1f" % R) except ZeroDivisionError: # linha 8 print("B não  
pode ser zero")
```

```
except ValueError: # linha 10
```

```
print("Digite números inteiros para A e B") except: # linha 12
```

```
print("Erro desconhecido. Não é possível calcular a divisão")
```



```
==== RESTART: C:\LivroPython\PgmsExemplo\codigo_3.11.py =====  
Digite um valor para A: texto  
Digite números inteiros para A e B  
>>>  
==== RESTART: C:\LivroPython\PgmsExemplo\codigo_3.11.py =====  
Digite um valor para A: 15  
Digite um valor para B: 0  
B não pode ser zero  
>>>  
==== RESTART: C:\LivroPython\PgmsExemplo\codigo_3.11.py =====  
Digite um valor para A: -5  
Digite um valor para B: 1  
Erro desconhecido. Não é possível calcular a divisão  
>>> A  
-5  
>>> cos(A)  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  
    cos(A)  
NameError: name 'cos' is not defined  
>>> |
```

O formato completo do comando try

O comando try tem outros dois blocos ainda não mencionados, que serão vistos agora. Sua forma completa contém, além de try e dos vários possíveis except, os blocos else e finally, que são opcionais.

O bloco de comandos 1 é posto em execução. Caso nenhuma exceção ocorra e esse bloco termine normalmente, então, o bloco except (ou blocos, no caso de exceções nomeadas) é pulado. Se uma exceção ocorrer no bloco 1, então, todos os



comandos posicionados abaixo da linha onde ocorreu a exceção são pulados, a execução do bloco 1 termina e o programa é desviado para o bloco 2 para tratar a exceção.

Caso não ocorra exceção e o try termine normalmente, então, será executado o bloco de comandos 3, que está subordinado ao else. Por fim, se o bloco finally estiver presente, ele sempre será executado, ocorra exceção ou não.

Além disso, é possível construir um bloco try que não contenha o except, mas contenha apenas try-finally. Esse tipo de construção permite ao programa ignorar eventuais exceções e executar um bloco de final (bloco de comandos 4) que faça algum tipo de tarefa de recuperação ou limpeza nos objetos do programa.

try:

{bloco de comandos 1} except:

{bloco de comandos 2} else:

{bloco de comandos 3} finally:

{bloco de comandos 4}

No Exemplo 3.17, escreva um programa que leia um número inteiro no intervalo [100, 500]. Caso o usuário digite um número fora do intervalo ou um dado não numérico, utilize o tratamento de exceção para avisá-lo.

Exemplo 3.17 Tratamento de exceções $N = 0$

while $N < 100$ or $N > 500$:

try:

$S = \text{input}(\text{"Digite N no intervalo [100, 500]: "})$ $N = \text{int}(S)$

except:

$\text{print}(\text{"\{ \} não é um número."}.\text{format}(S))$

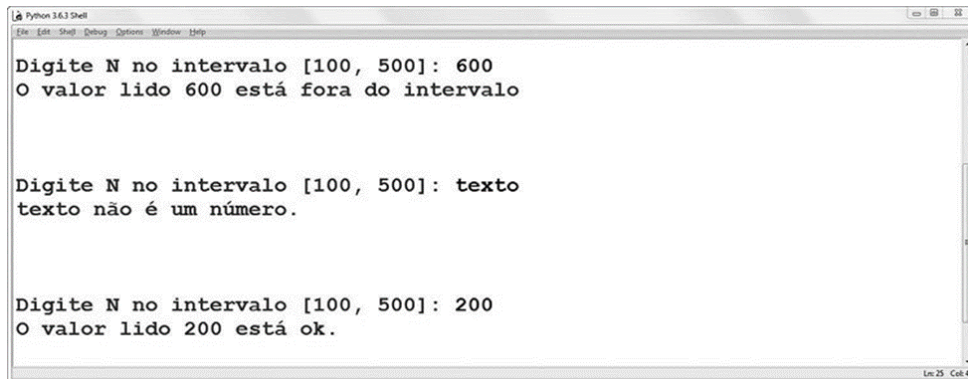
$N = 0$



else:

if $N < 100$ or $N > 500$:

```
print("O valor lido {} está fora do \
intervalo".format(N)) else:
print("O valor lido {} está ok.".format(N)) finally:
print("\n\n")
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help

Digite N no intervalo [100, 500]: 600
O valor lido 600 está fora do intervalo

Digite N no intervalo [100, 500]: texto
texto não é um número.

Digite N no intervalo [100, 500]: 200
O valor lido 200 está ok.
```

Execução do Exemplo 3.17.

Nessa solução foi usada uma exceção genérica (não nomeada) para capturar a entrada não texto do usuário. Nessa cláusula é exibida a mensagem avisando que o dado digitado não é número e o objeto N é zerado. O bloco `else` será executado caso tenha sido digitado um dado numérico, e nele a mensagem avisa se N está ou não dentro do intervalo. No bloco `finally` são executados alguns pulos de linha para que a exibição de dados na tela fique mais espaçada. Todo esse bloco está dentro de um laço `while`, que só terminará quando N digitado estiver no intervalo pedido.

Programa que totaliza um conjunto de valores

Escreva um programa que leia um número inteiro N e, em seguida, gere N números aleatórios no intervalo $[1, 50]$ e totalize-os. Para gerar números aleatórios, use a função `randint`, disponível na biblioteca `random`.

Antes de começar: em sistemas computacionais, é frequente a necessidade de serem gerados números aleatórios (gerar um banco de dados de testes, fazer simulações, gerar dados para um jogo etc.). Em Python está disponível a biblioteca



random, que contém um variado conjunto de funções destinadas a esse propósito. Para conhecer as possibilidades existentes, abra o IDLE e digite as duas linhas a seguir:

```
>>> import random
```

```
>>> help(random)
```

Como resultado da execução do comando help, serão listados todos os recursos contidos na biblioteca.

Aqui será utilizada a função `randint(a, b)`. Essa função retorna um número inteiro tal que: $a \leq \text{randint}(a, b) \leq b$. Assim, tem-se:

Programa que gera a sequência de Fibonacci

Escreva um programa que leia um número inteiro N e, em seguida, mostre na tela os N primeiros termos da sequência de Fibonacci. Faça o programa de modo que N seja no mínimo 2.

A sequência de Fibonacci é uma sequência de números inteiros que tem as seguintes regras de formação: os dois primeiros termos são 0 e 1; do terceiro em diante cada termo é a soma dos dois anteriores.

Se $N = 10$, então: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

A solução mais simples consiste em carregar dois objetos, A e B , com os valores iniciais e apresentá-los na tela. Após isso, inicie um laço no qual, na primeira repetição, seja calculado e exibido o terceiro termo e atualize A recebe o valor de B , que, por sua vez, recebe o valor calculado, preparando a próxima iteração.

