**Key takeaways**

1. Os operadores de **comparação** (ou os chamados *relacionais*) são usados para comparar valores. A tabela abaixo ilustra como os operadores de comparação funcionam, assumindo que x = 0, y = 1, e z = 0:

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descrição** | **Exemplo** |
| == | retorna se os valores dos operandos forem iguais, eFalse caso contrário | x == y *# False*  x == z *# True* |
| != | retorna Truese os valores dos operandos não forem iguais, eFalse caso contrário | x != y *# True*  x != z *# False* |
| > | Truese o valor do operando esquerdo for maior que o valor do operando direito, eFalse caso contrário | x > y *# False*  y > z *# True* |
| < | Truese o valor do operando esquerdo for inferior ao valor do operando direito, eFalse caso contrário | x < y *# True*  y < z *# False* |
| ≥ | Truese o valor do operando esquerdo for maior ou igual ao valor do operando direito, eFalse caso contrário | x >= y *# False*  x >= z *# True*  y >= z *# True* |
| ≤ | Truese o valor do operando esquerdo for inferior ou igual ao valor do operando direito, eFalse caso contrário | x <= y *# True*  x <= z *# True*  y <= z *# False* |

2. Quando quiser executar algum código apenas se uma determinada condição for cumprida, pode usar uma **declaração condicional**:

* uma únicaif declaração, por exemplo:

x = 10

if x == 10: *# condition*

print("x is equal to 10") *# Executed if the condition is True.*

* uma série de if declarações, por exemplo:

x = 10

if x > 5: *# condition one*

print("x is greater than 5") *# Executed if condition one is True.*

if x < 10: *# condition two*

print("x is less than 10") *# Executed if condition two is True.*

if x == 10: *# condition three*

print("x is equal to 10") *# Executed if condition three is True.*

Cada ifdeclaração é testada separadamente.

* uma if-elsedeclaração, por exemplo:

x = 10

if x < 10: *# Condition*

print("x is less than 10") *# Executed if the condition is True.*

else:

print("x is greater than or equal to 10") *# Executed if the condition is False.*

* uma série de if declarações seguidas por um else, por exemplo:

x = 10

if x > 5: *# True*

print("x > 5")

if x > 8: *# True*

print("x > 8")

if x > 10: *# False*

print("x > 10")

else:

print("else will be executed")

Cada if é testado separadamente. O corpo de else é executado se o último if for False.

* A if-elif-elsedeclaração, por exemplo:

x = 10

if x == 10: *# True*

print("x == 10")

if x > 15: *# False*

print("x > 15")

elif x > 10: *# False*

print("x > 10")

elif x > 5: *# True*

print("x > 5")

else:

print("else will not be executed")

Se a condição if for False, o programa verifica as condições dos elif blocos subsequentes - o primeiro elif bloco que True é executado. Se todas as condições forem False, o else bloco será executado.

* Declarações condicionais nested, por exemplo:

x = 10

if x > 5: *# True*

if x == 6: *# False*

print("nested: x == 6")

elif x == 10: *# True*

print("nested: x == 10")

else:

print("nested: else")

else:

print("else")

# Key takeaways: continuação

**Exercício 1**

Qual é o output do seguinte snippet?

x = 5

y = 10

z = 8

print(x > y)

print(y > z)

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

x, y, z = 5, 10, 8

print(x > z)

print((y - 5) == x)

Verifique

**Exercício 3**

Qual é o output do seguinte snippet?

x, y, z = 5, 10, 8

x, y, z = z, y, x

print(x > z)

print((y - 5) == x)

Verifique

**Exercício 4**

Qual é o output do seguinte snippet?

x = 10

if x == 10:

print(x == 10)

if x > 5:

print(x > 5)

if x < 10:

print(x < 10)

else:

print("else")

Verifique

**Exercício 5**

Qual é o output do seguinte snippet?

x = "1"

if x == 1:

print("one")

elif x == "1":

if int(x) > 1:

print("two")

elif int(x) < 1:

print("three")

else:

print("four")

if int(x) == 1:

print("five")

else:

print("six")

Verifique

**Exercício 6**

Qual é o output do seguinte snippet?

x = 1

y = 1.0

z = "1"

if x == y:

print("one")

if y == int(z):

print("two")

elif x == y:

print("three")

else:

print("four")

Verifique

**Key takeaways**

1. Existem dois tipos de loops em Python: while e for:

* o loop while executa uma declaração ou um conjunto de declarações, desde que uma condição booleana especificada seja verdadeira, por exemplo:

*# Example 1*

while True:

print("Stuck in an infinite loop.")

*# Example 2*

counter = 5

while counter > 2:

print(counter)

counter -= 1

* o loop for executa um conjunto de declarações várias vezes; é usado para iterar sobre uma sequência (por exemplo, uma lista, um dicionário, um tuple, ou um conjunto - aprenderá sobre eles em breve) ou outros objetos que são iteráveis (por exemplo, strings). Pode utilizar o loop for para iterar sobre uma sequência de números usando a função range . Veja os exemplos em baixo:

*# Example 1*

word = "Python"

for letter in word:

print(letter, end="\*")

*# Example 2*

for i in range(1, 10):

if i % 2 == 0:

print(i)

2. Pode utilizar as declarações break e continue para alterar o fluxo de um loop:

* Utilize break para sair de um loop, por exemplo:

text = "OpenEDG Python Institute"

for letter in text:

if letter == "P":

break

print(letter, end="")

* Utilize continue para ignorar a iteração atual e continuar com a próxima iteração, por exemplo:

text = "pyxpyxpyx"

for letter in text:

if letter == "x":

continue

print(letter, end="")

3. Os loops while e for também podem ter uma cláusula else em Python. A cláusula else executa-se após o loop terminar a sua execução, desde que não tenha sido terminado por break, por exemplo.:

n = 0

while n != 3:

print(n)

n += 1

else:

print(n, "else")

print()

for i in range(0, 3):

print(i)

else:

print(i, "else")

4. O objeto da exceção range() gera uma sequência de números. Aceita números inteiros e devolve objetos de range. A sintaxe de range() parece como se segue: range(start, stop, step), onde:

* start é um parâmetro opcional que especifica o número inicial da sequência (0 por padrão)
* stop é um parâmetro opcional que especifica o fim da sequência gerada (não está incluído),
* e step é um parâmetro opcional que especifica a diferença entre os números na sequência (1 por padrão.)

Código de exemplo:

for i in range(3):

print(i, end=" ") *# Outputs: 0 1 2*

for i in range(6, 1, -2):

print(i, end=" ") *# Outputs: 6, 4, 2*

# Principais takeaways: continuação

**Exercício 1**

Crie um loop for que conta de 0 a 10, e imprime os números ímpares no ecrã. Use o esqueleto abaixo:

for i in range(1, 11):

# Line of code.

# Line of code.

Verifique

**Exercício 2**

Crie um loop while que conta de 0 a 10, e imprime os números ímpares no ecrã. Use o esqueleto abaixo:

x = 1

while x < 11:

# Line of code.

# Line of code.

# Line of code.

Verifique

**Exercício 3**

Crie um programa com um loop for e uma declaração break . O programa deve iterar sobre os caracteres de um endereço de e-mail, sair do loop quando chegar ao símbolo @ , e imprimir a parte antes de @ numa linha. Use o esqueleto abaixo:

for ch in "john.smith@pythoninstitute.org":

if ch == "@":

# Line of code.

# Line of code.

Verifique

**Exercício 4**

Crie um programa com um loop for e uma declaração continue . O programa deve iterar sobre uma string de dígitos, substituir cada 0 com xe imprimir a string modificada no ecrã. Use o esqueleto abaixo:

for digit in "0165031806510":

if digit == "0":

# Line of code.

# Line of code.

# Line of code.

Verifique

**Exercício 5**

Qual é o output do seguinte código?

n = 3

while n > 0:

print(n + 1)

n -= 1

else:

print(n)

Verifique

**Exercício 6**

Qual é o output do seguinte código?

n = range(4)

for num in n:

print(num - 1)

else:

print(num)

Verifique

**Exercício 7**

Qual é o output do seguinte código?

for i in range(0, 6, 3):

print(i)

Verifique

Vamos facilitar as coisas:

* OPERADORES BITWISE TRABALHAM COM VALORES EM BITS
* & requer exatamente dois 1para fornecer 1 como resultado;
* | requer pelo menos um 1 para fornecer 1 como resultado;
* ^ requer exatamente um 1 para fornecer 1 como resultado.

E aqui está a **tabela de prioridades atualizada**, contendo todos os operadores introduzidos até agora:

|  |  |  |
| --- | --- | --- |
| **Prioridade** | **Operador** |  |
| 1 | ~, +, - | unário |
| 2 | \*\* |  |
| 3 | \*, /, //, % |  |
| 4 | +, - | binário |
| 5 | <<, >> |  |
| 6 | <, <=, >, >= |  |
| 7 | ==, != |  |
| 8 | & |  |
| 9 | | |  |
| 10 | =, +=, -=, \*=, /=, %=, &=, ^=, |=, >>=, <<= |  |

**Key takeaways**

1. O Python suporta os seguintes operadores lógicos:

* and → se ambos os operandos forem verdadeiros, a condição é verdadeira, por exemplo, (True and True) é True,
* or → se algum dos operandos for verdadeiro, a condição é verdadeira, por exemplo, (True or False) é True,
* not → retorna falso se o resultado for verdadeiro, e retorna verdadeiro se o resultado for falso, por exemplo, not True é False.

2. Pode utilizar operadores bitwise para manipular bits únicos de dados. Os seguintes dados de amostra:

* x = 15, que é 0000 1111 em binário,
* y = 16, que é 0001 0000 em binário.

serão utilizados para ilustrar o significado de operadores bitwise em Python. Analise os exemplos em baixo:

* & faz um *bitwise and*, por exemplo, x & y = 0, que é 0000 0000 em binário,
* | faz um *bitwise ou*, por exemplo, x | y = 31, que é 0001 1111 em binário,
* ˜  faz um *bitwise não*, por exemplo, ˜ x = 240\*, que é 1111 0000 em binário,
* ^ faz um *bitwise xor*, por exemplo, x ^ y = 31, que é 0001 1111 em binário,
* >> faz um *bitwise right shift*, por exemplo, y >> 1 = 8, que é 0000 1000 em binário,
* << faz um *bitwise left shift*, por exemplo, y << 3 = , que é 1000 0000 em binário,

\* -16 (decimal do complemento assinado de 2) — leia mais sobre a[Two's complement](https://en.wikipedia.org/wiki/Two%27s_complement) operation (operação de complemento de dois).

**Exercício 1**

Qual é o output do seguinte snippet?

x = 1

y = 0

z = ((x == y) and (x == y)) or not(x == y)

print(not(z))

Verifique

False

**Exercício 2**

Qual é o output do seguinte snippet?

x = 4

y = 1

a = x & y

b = x | y

c = ~x # tricky!

d = x ^ 5

e = x >> 2

f = x << 2

print(a, b, c, d, e, f)

Verifique

0 5 -5 1 1 16

# Adicionar elementos a uma lista: append() e insert()

Um novo elemento pode ser *colado* no fim da lista existente:

list.append(value)

Tal operação é realizada por um método chamado append(). Toma o valor do seu argumento e coloca-o **no final da lista** que possui o método.

O comprimento da lista aumenta então em um.

O método insert() é um pouco mais inteligente - pode acrescentar um novo elemento **em qualquer lugar da lista**, e não apenas no final.

list.insert(location, value)

# Adicionar elementos a uma lista: continuação

Pode **iniciar a vida de uma lista tornando-a vazia** (isto é feito com um par de parêntesis retos vazio) e depois adicionando-lhe novos elementos conforme necessário.

Veja o snippet no editor. Tente adivinhar o seu output após a execução do loop for . Execute o programa para verificar se estava certo.

Será uma sequência de números inteiros consecutivos a partir de 1 (depois adiciona-se um a todos os valores anexados) até 5.

Modificámos um pouco o snippet:

my\_list = [] # Creating an empty list.

for i in range(5):

my\_list.insert(0, i + 1)

print(my\_list)

insert adiciona a lista de forma reversa

para uma variável assumir o valor de outra, exsitem duas formas

É assim que se pode fazer de forma clássica:

variable\_1 = 1

variable\_2 = 2

auxiliary = variable\_1

variable\_1 = variable\_2

variable\_2 = auxiliary

O Python oferece uma forma mais conveniente de fazer a troca - veja:

variable\_1 = 1

variable\_2 = 2

variable\_1, variable\_2 = variable\_2, variable\_1

my\_list = [10, 1, 8, 3, 5]

length = len(my\_list)

for i in range(length // 2):

my\_list[i], my\_list[length - i - 1] = my\_list[length - i - 1], my\_list[i]

print(my\_list)

Nota:

* nós atribuímos a variável length com o comprimento da lista atual (isto torna o nosso código um pouco mais claro e mais curto)
* lançámos o loop for para correr através do seu corpo length // 2 vezes (isto funciona bem para listas com comprimentos pares e ímpares, porque quando a lista contém um número ímpar de elementos, o do meio permanece intocado)
* trocamos o i-ésimo elemento (desde o início da lista) com o que tem um index igual a (length - i - 1) (do final da lista); no nosso exemplo, para i igual a 0 o ramo (lenght - i - 1) dá 4; para i igual a 1, dá 3 - Isto é exatamente o que precisávamos.

As listas são extremamente úteis, e irá encontrá-las com muita frequência.

# Key takeaways

1. A **lista é um tipo de dados** em Python usada para **armazenar vários objetos**. É uma **coleção ordenada e mutável** de ítens separados por vírgulas, entre parêntesis retos, por exemplo:

my\_list = [1, None, True, "I am a string", 256, 0]

2. As listas podem ser **indexadas e atualizadas**, por exemplo:

my\_list = [1, None, True, 'I am a string', 256, 0]

print(my\_list[3]) *# outputs: I am a string*

print(my\_list[-1]) *# outputs: 0*

my\_list[1] = '?'

print(my\_list) *# outputs: [1, '?', True, 'I am a string', 256, 0]*

my\_list.insert(0, "first")

my\_list.append("last")

print(my\_list) *# outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']*

3. As listas podem ser **nested**, por exemplo:

my\_list = [1, 'a', ["list", 64, [0, 1], False]]

Aprenderá mais sobre o nesting no módulo 3.1.7 - por enquanto, só queremos que esteja ciente de que algo como isto também é possível.

4. Os elementos da lista e as listas podem ser **excluídos**, por exemplo:

my\_list = [1, 2, 3, 4]

del my\_list[2]

print(my\_list) *# outputs: [1, 2, 4]*

del my\_list *# deletes the whole list*

Novamente, aprenderá mais sobre isto no módulo 3.1.6 - não se preocupe. Por enquanto, basta tentar experimentar o código acima e verificar como a sua alteração afeta o output.

5. As listas podem ser **iteradas** através da utilização do loop for , por exemplo:

my\_list = ["white", "purple", "blue", "yellow", "green"]

for color in my\_list:

print(color)

6. A função len() pode ser usada para **verificar o comprimento da lista**, por exemplo:

my\_list = ["white", "purple", "blue", "yellow", "green"]

print(len(my\_list)) *# outputs 5*

del my\_list[2]

print(len(my\_list)) *# outputs 4*

7. Uma invocação de **função** típica parece-se com a seguinte: result = function(arg), enquanto uma invocação de **método** típica parece-se com isto:result = data.method(arg).

**Exercício 1**

Qual é o output do seguinte snippet?

lst = [1, 2, 3, 4, 5]

lst.insert(1, 6)

del lst[0]

lst.append(1)

print(lst)

Verifique

[6, 2, 3, 4, 5, 1]

**Exercício 2**

Qual é o output do seguinte snippet?

lst = [1, 2, 3, 4, 5]

lst\_2 = []

add = 0

for number in lst:

add += number

lst\_2.append(add)

print(lst\_2)

Verifique

[1, 3, 6, 10, 15]

**Exercício 3**

O que acontece quando executa o seguinte snippet?

lst = []

del lst

print(lst)

Verifique

NameError: name 'lst' is not defined

**Exercício 4**

Qual é o output do seguinte snippet?

lst = [1, [2, 3], 4]

print(lst[1])

print(len(lst))

Verifique

[2, 3]

3