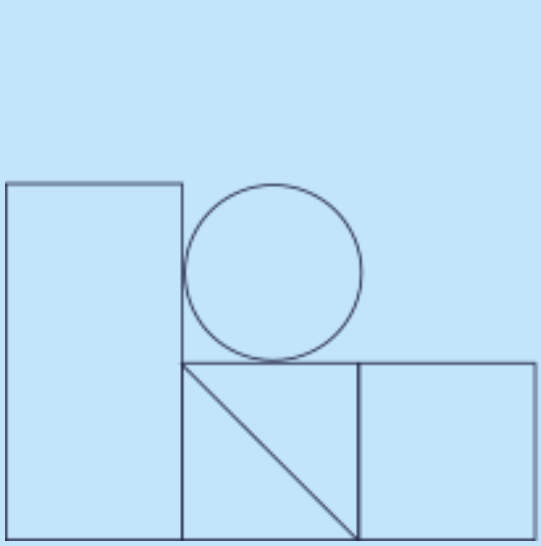
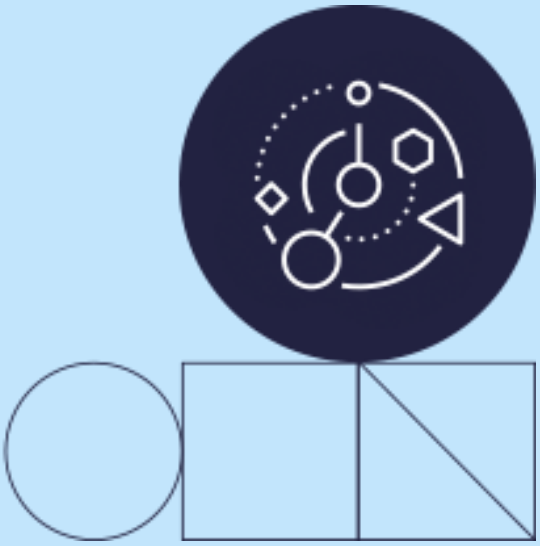
IBM SkillsBuild

IBM

Noções básicas e sintaxe de Python

Sentenças condicionais em Python





Índice

Introdução

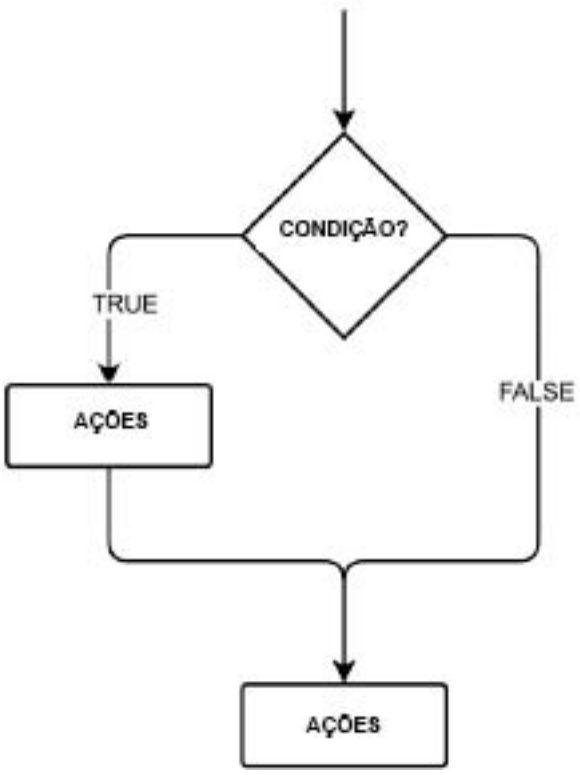
Python também possui modificadores de fluxo de execução, tais como condicionadores e loops com os quais se pode tomar decisões ou repetir o código várias vezes.

Controle de Fluxo em Python

Nesta unidade, vamos conhecer as sentenças condicionais.

A declaração if

A primeira declaração que nos permite controlar o fluxo do nosso programa é a declaração **if**. Esta declaração nos permite executar condicionalmente pedaços de código, como vemos no seguinte fluxograma:



Sintaxe:

```
>>> if CONDICION:
...     print('Se ejecuta si CONDICION es TRUE')
...     print('También se ejecuta')
>>> print('Ya estamos fuera del if')
```

Aqui está um exemplo:

```
>>> a = 10
>>> b = 3

>>> if a > b:
...    print ('SI se cumple la condición')
# Bloque identado 4 espacios
...
>>> print ('Ya estamos fuera del if')

SI se cumple la condición
Ya estamos fuera del if
```

Como vemos a declaração **if** em uma sintaxe que será familiar para você. Basicamente, é uma avaliação de uma expressão que devolve um booleano (neste caso ***a > b***). Se a expressão retorna ***TRUE*** então inserimos um bloco específico de código (o primeiro **print** do exemplo). Se a expressão retorna ***FALSE***, não inserimos esse bloco de código.

Sintaxe e aninhamento em Python

Ao contrário de outras línguas próximas ao C, Python remove bastantes elementos para aumentar a legibilidade do código. Por exemplo, a mesma afirmação em C++ seria parecida com esta:

```
>>> if (a > b){
      printf (“a es mayor que b”);
}
```

Considerando que, em Python, o mesmo código seria:

```
>>> if a > b:              # if en Python
...    print ('a es mayor que b')
```

As principais diferenças são que Python acrescenta dois pontos (:) no final da avaliação condicional

para indicar que está iniciando um novo bloco. Por sua vez, ele remove:

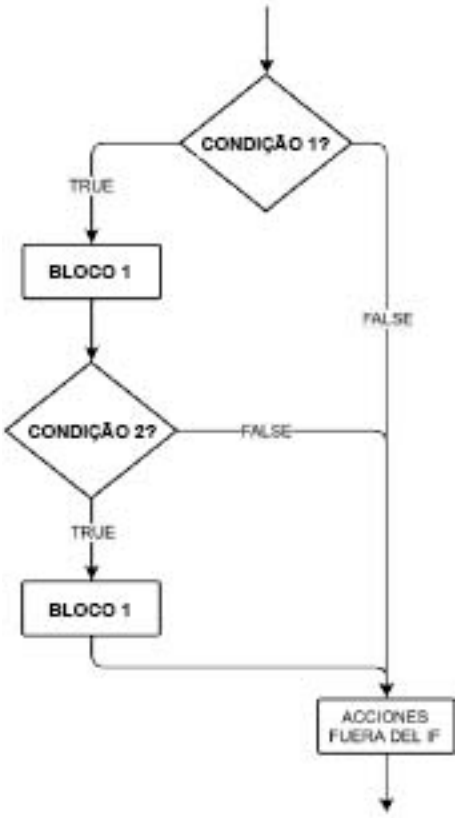
- **Parênteses de avaliação**. Em Python, eles são opcionais.
- **Chaves que bloqueiam o bloco**. Python não usa aparelho para separar blocos de código. Em vez disso, o aninhamento é indicado pelo uso de blocos indentados, geralmente por quatro (4) espaços.
- **Ponto-e-vírgula**. Em Python é opcional e normalmente é usado para delimitar várias declarações na mesma linha (não é uma prática recomendada).

Quando Python foi projetado, ele foi projetado com a legibilidade em mente, então foi decidido simplificar o número de elementos que povoam cada afirmação. Além disso, a eliminação de suportes para separar blocos de código significava que o recuo era usado como um meio de separar blocos de código. Isto obriga os programadores a cuidar da recuo ao escrever, o que torna o código mais ordenado e coerente, aumentando assim sua legibilidade. Isto nos obriga a ser sempre coerentes com indentações, algo que não acontece em outras linguagens de programação onde o fato de usar aparelhos leva muitos programadores a serem descuidados com indentações, de modo que muitas vezes vemos diferentes estratégias de indentação no mesmo programa. Apesar de ser uma das características mais irritantes para os recém-chegados, esta é uma das muitas decisões de design da linguagem onde sua elegância pode ser apreciada.

O agrupamento em blocos é feito em grupos de 4 espaços. Portanto, se tivermos múltiplos ninhos, aumentaremos em 4 espaços em cada bloco.

```
>>> if ___:                  # Raíz. 1er nivel de indentación (0 espacios)
...     ___                      # Bloque 1: 4 espacios
...     if ___:                  # Seguimos en Bloque 1: 4 Espacios
...         ___                      # Bloque 2: 4 + 4 espacios
...         ___                      # Seguimos en bloque 2. 4 + 4 espacios
...     ___                      # Hemos vuelto al Bloque 1.
>>> ___                      # Raíz. Ya hemos salido del 'if' 1er nivel
```

Em um fluxograma, isto corresponderia a:



Múltiplos ramos condicionais

Com o interlúdio de sintaxe, vejamos como podemos criar múltiplas ramificações condicionais. Para fazer isso, inserimos as palavras-chave ***else*** e ***elif*** (else if).

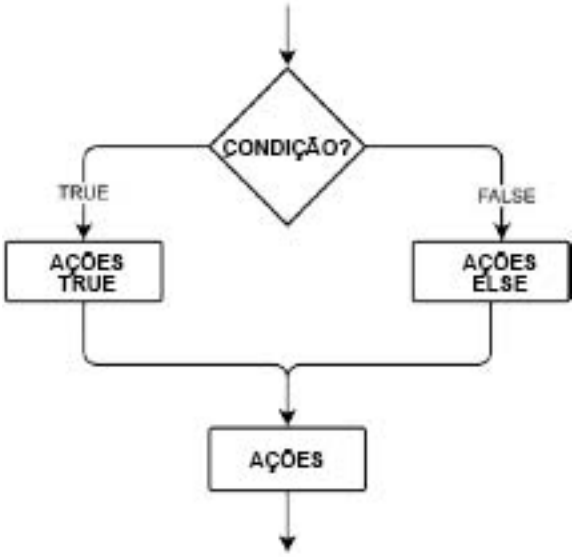
```
>>> a = 10
>>> b = 3

>>> if a > b:
...    print ('Se ha cumplido la condición')
... else:
...    print ('No se ha cumplido la
condición')
...
>>> print ('Ya estamos fuera del if')

No se ha cumplido la condición
Ya estamos fuera del if
```

Neste primeiro caso, podemos ver como, por meio da palavra-chave ***else*** nos permite executar ramos de código no caso de a condição avaliada na expressão do ***if***. Note que a palavra-chave ***else*** está no mesmo nível que o ***if*** ao qual está ligado.

Neste caso, o fluxograma seria o seguinte:



No caso em que gostaríamos de avaliar mais condições, usamos a palavra-chave ***elif*** (else if).

```
>>> a = 10
>>> b = 10

>>> if a > b:
...    print ('A es menor que B')
... elif a == b
...    print ('A es igual a B')
... else:
...    print ('A es mayor que B')

...
>>> print ('Ya hemos salido del
condicional')


A es igual a B
Ya hemos salido del condicional
```

A palavra-chave ***elif*** nos permite fazer avaliações alternativas à do ***if***. No caso em que o ***if*** retornou ***False***, avaliaríamos então o bloco ***elif*** e executar seu código associado, no caso de retornar ***True***. Como em outras linguagens de programação, podemos usar o número de ***elif*** que precisamos. Da mesma forma, se não ***elif*** é cumprido, temos um bloco ***else*** para executar um determinado código no caso de nenhuma das avaliações ter sido cumprida (todas retornaram ***False***).

Expressões condicionais

Se você tiver programado em C/C++ ou javascript, você provavelmente conhece o operador ternário:

(a > b) ? 20: 30.

Este operador retorna 20 se **a** é maior do que ***b*** ou 30 se não for. Em Python, temos um operador equivalente, mas com uma sintaxe muito mais legível:

```
>>> a = 10
>>> b = 3
>>> 20 if a > b else 30
20
```

Esta expressão é chamada de expressão condicional ou operador ternário. Ao contrário dos ***if***, este operador é uma expressão em si e não uma declaração. Sua notação é a seguinte:

```
x = true_value if condición else
false_value
```

Embora a expressão seja lida da esquerda para a direita, ela é na verdade avaliada na seguinte ordem. Em primeiro lugar, a condição é avaliada. Se for verdade, então é retornado ***true_value***. Se não for, é retornado ***false_value***.

Note que esta é uma forma muito reduzida de escrever os condicionamentos, pois equivale ao seguinte bloco de código:

```
>>> if condición:
...     x = true_value
... else:
...    x = false_value
```

Entretanto, é aconselhável não abusar destas expressões e usá-las apenas quando estamos escrevendo pequenas avaliações que não requerem muito texto ou condições complexas para serem compreendidas.

Além disso, embora não seja necessário, muitas vezes é conveniente embrulhar esta expressão entre parênteses para melhorar sua legibilidade:

```
x = (true_value if condición else
false_value)
```

Recomendamos que sempre que você usar esta expressão, a coloque entre parênteses. Muitos de vocês podem se perguntar por que não lidamos com as declarações **switch-case** nesta unidade. A resposta é simples, Python não tem nenhum declarações **switch-case** portanto, este tipo de condições múltiplas deve ser resolvido usando **if-elif**.

Vamos fazer alguns exemplos práticos do uso de condicional em Python:

Exemplo 1:

Criar um programa simples que pede uma nota por console e avalia se o aluno passou ou reprovou, dependendo se a nota é de pelo menos 5.

Nota: Para realizar estes exemplos, vamos pedir ao usuário que introduza valores através do console. Para fazer isso, precisaremos fazer uso de uma função que ainda não vimos, mas que vamos adiante. Esta é a função **input()**. Com ela poderemos

introduzir em uma variável um valor que o usuário introduz dinamicamente. Deve ser levado em conta que, em Python, qualquer valor que seja introduzido por meio de um **input()** será considerado como **string**, mesmo que seja um número, então se quisermos realizar cálculos matemáticos ou comparações com esse valor, teremos que passá-lo de **string** a número. Em nosso caso, nós gastamos **string** a **int** a fim de poder verificar posteriormente se seu valor é inferior a 5.

```
notaIn=int(input("Introduzca nota:"))

if notaIn<5:
    calificacion="Suspenso"
else: calificacion="Aprobado"

print(calificacion)
```

Exemplo 2:

Criar um programa que peça uma idade por console e avalie se o usuário tem idade legal ou não, dependendo se a variável é menor de 18 anos. Se o usuário for menor de 18 anos, será exibida uma mensagem de proibição de entrada, se o usuário for maior de 18 anos, a entrada será permitida. Além disso, se a idade entrada for maior que 100 anos, será considerada errada.

```
edad=int(input("Introduce edad: "))

if edad<18:
    print("No puedes pasar")
elif edad>100:
    print("Edad incorrecta")
else:
    print("Adelante")
```

Exemplo 3:

Criar um programa que peça uma nota por console e avalie as possíveis notas do aluno, sendo uma reprovação inferior a 5, passar entre 5 e 7, destacando-se entre 7 e 9 e A+ acima de 9.

```
nota=int(input("Introduce tu nota: "))
if nota<5:
    print("Suspenso")
elif nota<7:
    print("Aprobado")
elif nota<9:
    print("Notable")
else:
    print("Sobresaliente")
```

Laço while

Com estes três exemplos, pudemos ver o uso de condicionadores e também pudemos rever os tópicos anteriores. Em seguida, analisaremos os loops e os combinaremos com declarações condicionais.

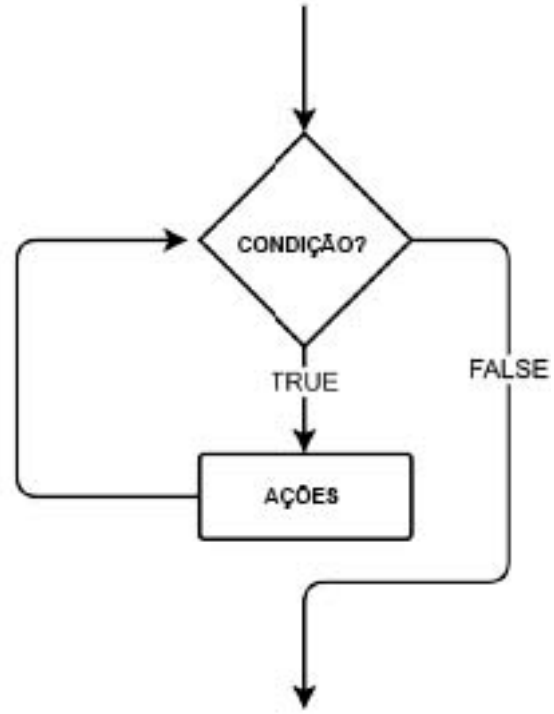
Outra das afirmações mais usadas em um programa são os laços **while**. Como em outras linguagens de programação, o laço **while** repete um pedaço de código iterativamente enquanto uma determinada condição for cumprida.

```
>>> a = 0
>>> while a<3:
...       print (a, end=' ')   # Acabamos
con espacios en lugar de salto de línea
...       a += 1               # Equivalente
a: a = a + 1
>>> print (a)                  # Estamos
fuera del while
>>> print('Hemos salido fuera del while')

0 1 2 3
Hemos salido fuera del while
```

No início de cada iteração, a expressão inicial do **while** é avaliada. Se a expressão for satisfeita (devolve **True**), você entra novamente no **while**. e não for cumprido (devolve **False**), o **while** deixa de executar e passamos para a próxima declaração após o **while**.

É assim que seria expresso em um fluxograma:



Os laços e as declarações continue, break, pass e os blocos else

Agora que sabemos como fazer loop em **Python**, vejamos algumas declarações que nos permitem alterar o fluxo natural dos loops. Estas são as seguintes declarações:

- ***break*:** Interrompe o fluxo e as saídas fora do laço.
- ***continue*:** Salta para o início da próxima iteração do laço.
- ***pass*:** Ela não faz nada. É uma frase vazia.
- ***Else:*** No final de um laço: Executado somente se o laço tiver terminado normalmente. Ou seja, ele é executado se o loop terminar sem ter executado um ***break***.

Declaração break

Vejamos alguns exemplos desses julgamentos, começando com a declaração ***break***.

```
>>> a = 5
>>> while a:                # Utilizamos la
propia variable como condición
...       print (a, end=' ')
...       a == 2:
...           break
...       a -= 1
...
>>> print ('\nFuera del Bucle.')
>>> print('Valor de "a": {}'.format(a))

5 4 3
Fuera del Bucle.
```

Como podemos ver, esta declaração interrompe imediatamente o loop e sai para a próxima declaração no código.

Note que neste caso a expressão que estamos avaliando é a própria variável ***a***. Isto pode parecer estranho no início, mas é uma forma muito elegante de avaliar as expressões neste tipo de afirmação. Note que no ***while*** sempre avalia a uma expressão booleana. Ou seja, estamos realmente avaliando ***bool(a)*** que, sendo um número inteiro, sempre retorna ***True*** até ***a*** sea 0. Você verá este tipo de expressão muito freqüentemente em **Python**.

Declaração continue

A declaração **continue** nos permite interromper a iteração atual, ou seja, saltamos um laço de volta:

```
>>> a = 7
>>> while bool(a):
...       a -= 1
...       if a % 2 == 0
...            continue   # Saltamos a la
siguiente iteracción si es a es par.
...       print (a, end=' ')
...
>>> print ('\nFuera del Bucle.')

5 3 1
Fuera del Bucle.
```

A declaração pass

A declaração **pass** não faz absolutamente nada. É um espaço reservado para laços vazios.

```
>>> a = 5
>>> while a:
...       pass    # Presiona Ctrl-C para
abortar la ejecución
...
```

Note que este laço está bloqueado porque não fazemos nada dentro dele. Se quisermos abortar a execução, teremos de pressionar ***Ctrl-C.***

O Blocos else no final dos laços

Como já vimos, temos duas formas de terminar os laços. A primeira é uma terminação limpa e ininterrupta, e a segunda é usando a declaração **break**. Para saber se um loop terminou de certa forma, em muitas linguagens de programação são usadas bandeiras (flags) no código, que depois precisam ser avaliadas. Em **Python**, isso não é necessário, pois podemos usar blocos **else** no final de um laço.

Por exemplo, vejamos um exemplo onde avaliamos se um número é primo ou não dentro de um laço **while**.

```
>>> a = 13
>>> b = a // 2           # División entera.
P. ej. 13 // 2 == 6
>>> while b > 1:
...       if a % b == 0:  # % es el operador
resto. P. ej. 10 % 5 == 0
...           print('{b} es factor de
{a}'.format(b,a))
...           break
...       b -= 1
... else:
...           print('{} es primo'.format(a))
...
>>> print ('\nFuera del Bucle.')

13 es primo
Fuera del Bucle.
```

Neste caso, procuramos fatores do número 13 e, não tendo encontrado nenhum, não corremos **break** que tínhamos dentro do condicional. Assim, o laço terminou de forma limpa, então o bloco **else** foi executado no final do **while**.

O laço for

A sintaxe do laço **for** é muito simples e semelhante à do laço **while**com a exceção de que à frente do **for**, em vez de avaliar uma expressão para cada iteração, atribuímos elementos de um iterador a uma variável.

```
>>> for elem in objeto:  # Vamos asignado elementos de objeto en elem
...     sentencias
...     if test:        # Podemos usar breaks
...         break
...     if test:        # Podemos usar continues
...         continue
... else:               # Si no hemos salido a través de break
...     sentencias
```

Vamos começar com um exemplo simple:

```
>>> for s in ['Me', 'gusta', 'Python!']:
...     print(s, end=' ')
Me gusta Python!
```

Neste exemplo, estamos percorrendo de bicicleta elementos de uma lista de **strings** e imprimi-las na tela. Vamos olhar para outro exemplo, desta vez somando números:

```
>>> a = 0
>>> for x in [1, 2, 3, 4]:
...     a += x
...
>>> print(a)
10
```

E agora outro exemplo, desta vez com **strings**:

```
>>> for c in 'Me gusta Python!':
...     print(c, end=' ')
...
M e g u s t a P y t h o n !
```

Este laço passa por um **string** carta por carta e os exibe na tela, acrescentando um espaço após cada impressão na tela.

Finalmente, vejamos um exemplo que passa por um dicionário, como um dos usos mais freqüentes do bloco **for** é atravessar objetos iteráveis, dicionários, tuploa, etc.:

```
>>> for k in d:
...     print(k, end=' ')
Apellidos edad nombre
```

Por padrão, quando você passa um dicionário para um ***for***nós a atravessamos através de suas chaves. Há várias maneiras de extrair valores de um dicionário. Mais tarde, aprenderemos a atravessar chaves e valores simultaneamente, mas por enquanto vamos olhar para uma maneira muito simples de fazê-lo:

```
>>> keys = ['nombre', 'apellidos', 'edad']
>>> values = ['Guido', 'van Rossum', '60']
>>> d = dict(zip(keys, values))   # Creamos el diccionario

>>> for k in d:
...     info = '{}: {}'.format(k, d[k])
# Texto formateado con claves y valores

...     print(info)
Apellidos: van Rossum
edad: 60
nombre: Guido
```

O método **str.format** substitui as chaves na corda pelos parâmetros que lhe passamos quando a chamamos. Em cada iteração do exemplo, estamos passando a chave **(k)** do dicionário e o valor do dicionário correspondente a essa chave **(d[k])**.

Designação em tuplas

Um conceito que ainda não tínhamos visto é que Python permite a **atribuição em tuplas**. Ou seja, podemos atribuir os elementos de um tuple a várias variáveis ao mesmo tempo.

```
>>> a, b = (3, 4)     # Asignamos los elementos de la tupla a cada variable
>>> print(a, b)
3 4
```

Exploraremos essas capacidades Python com mais detalhes mais tarde. Mas por enquanto é suficiente saber que isto é muito conveniente em laços **for**. Por exemplo:

```
>>> t = [(1, 2), (3, 4), (5, 6)]
>>> for x, y in t:
...     print (x + y, end= '  ')
3 7 11
```

Neste exemplo, estamos atravessando uma lista de tuplas e os atribuímos a duas variáveis simultaneamente (**x**, e **y**) que depois usamos dentro do **for** para somá-las. Este processo é chamado de **desempacotar em tuplas**.

Você verá que isto é muito útil em muitas situações, por exemplo, quando queremos passar por duas listas em paralelo:

```
>>> keys = ['nombre', 'apellidos', 'edad']
>>> values = ['Guido', 'van Rossum', 60]

>>> d = dict(zip(keys, values))

>>> for k, v in d.items():    # d.items devuelve tupla con clave, valor
...     print('{}: {}'.format(k, v))

apellidos: van Rossum
edad: 60
nombre: Guido
```

Vemos que o método **dict.items** retorna uma tupla (chave, valor) correspondente a uma entrada de dicionário em cada iteração do **for**.

Vistas em dicionários

Observe que a partir de Python 3.5, o método ***dict.items*** retorna uma visão dos elementos do dicionário, ou seja, não retorna os objetos em si até que os atravessemos ou os convertamos explicitamente em listas. Outros métodos com o mesmo comportamento são ***dict.keys*** e ***dict.values***, que retornam visões das chaves e dos valores respectivamente.

```
>>> for k in d.keys():
...     print(k, end= ' ')
apellidos edad nombre

>>> for v in d.values():
...     print(v, end= ' ')
Van Rossum 60 Guido
```

Se ao invés de atravessá-los, tentamos extrair todas as chaves ou elementos, vemos que eles não retornam uma lista, mas são vistas no dicionário:

```
>>> d.keys()
dict_keys(['nombre', 'apellidos', 'edad'])

>>> d.keys()
dict_keys

>>> d.values()
dict_values(['van Rossum', 60, 'Guido']

>>> type(d.values())
dict_values

>>> d.items()

dict_items([('apellidos', 'van Rossum'),
('edad', 60, ('nombre', 'Guido')])


>>> type(d.items())
dict_items
```

Note que não podemos acessar estas listas diretamente, mas temos que atravessá-las como vimos antes, ou envolvê-las em listas para indexá-las, parti-las em pedaços, etc.:

```
>>> d.keys()[1]      # No podemos acceder a
una vista de diccionario
-----------------------------------
Error. Texto omitido por simplicidad
TypeError: 'dict_keys' object does not
support indexing

>>> list (d.keys())[1]
'edad'
```

Laços for e balcões

A esta altura, você já deve ter se surpreendido com as diferenças entre os **for** de outras linguagens, como C/C++ ou Java com as de Python. Os laços **for** do **Python** são mais parecidos com os **for-each** de outras linguagens de programação.
Ou seja, eles são destinados a atravessar os elementos de seqüências ou iterables.
Antes de mais nada, vamos ver o que um **for** para o estilo C e o estilo Pythônico. O modo pitônico é, em quase todos os casos, mais legível e mais rápido de executar.

```
>>> letras = list('abcdefghijklmnopqrstuvwxyz')

>>> for i in range(len(letras)):     # Versión C/C++. No lo uséis!
...     print(letras[i], end=' ')
...
a b c d e f g h i j k l m n o p q r s t u v w x y z


>>> for c in letras:                 # Versión Pythónica. Más legible y rápido
...     print(c, end=' ')
...
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Como podemos ver, é possível fazer um **for** al estilo C/C++. A função **range** retorna uma lista de números consecutivos do comprimento que passamos por ela. Neste caso, passamos o comprimento da lista de cartas.

```
# Ejemplos de uso de range
# Hay que envolverlo en lista o recorrerlo
en for

>>> list(range(5))   # Devuelve 5 elementos
empezando en 0
[0, 1, 2, 3, 4]

>>> list(range(-5, 5))     # Devuelve
elementos en el rango -5, 5
[-4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(range(-5, 5, 2))  # Elementos -5 a
5 en saltos de 2
[-5, -3, -1, 1, 3]
```

Em geral, é sempre aconselhável usar o laço **for** Estilo Python. Quando começarmos a usar loops, você será tentado a usar os **for** da forma típica de outros idiomas. Com algumas exceções, é sempre melhor fazê-lo da maneira Pythônica.

Aqui estão alguns daqueles casos típicos em que os recém-chegados a **Python** têm a tendência de programar laços **for** não Pythônicos. Também damos a alternativa a ser utilizada.

O primeiro destes exemplos que vimos antes. É um caso de passar por várias listas simultaneamente.

```
>>> import random

>>> l1 = letras[:8]                    # Creamos 3 sub-listas a partir de trozos
>>> l2 = letras[8:16]
>>> l3 = letras[16:]

>>> random.shuffle(l1)                 # Barajamos cada trozo
>>> random.shuffle(l2)
>>> random.shuffle(l3)

>>> for i in range(len(l1)):           # Versión NO Pythónica. NO RECOMENDADA
...     print(l1[i] + l2[i] + l3[i], end=' ')
...
cpv emq dlt hnr fis gow akz bjx

>>> for a, b, c in zip(l1, l2, l3):    # Versión Pythónica.
...     print(a + b + c, end=' ')
...
cpv emq dlt hnr fis gow akz bjx
```

Neste exemplo, estamos cortando a lista de letras em três partes, depois embaralhamos cada uma das partes e corremos através das três listas simultaneamente para exibir uma letra de cada lista a cada iteração. A versão Pythônica utiliza a função **zip**, que já vimos antes. A única diferença é que, neste caso, em vez de juntar duas listas, estamos fazendo isso com três. A grande vantagem do zip (além de ser mais legível) é que não temos que nos preocupar com o fato de todas as listas terem o mesmo comprimento. Quando uma das listas é concluída, **zip** interrompe a execução. Na alternativa não Pythônica, teríamos que olhar para o comprimento de cada lista e fazer o loop através do loop seguindo os índices da lista mais curta.

Outro exemplo típico em que os programadores Python novatos tendem a fazer uso de **for** não Pythônicos é quando estamos fazendo buscas de índice dos elementos de uma lista. Por exemplo:

```
>>> letras = list('abcdefghijklmnopqrstuvwxyz')
>>> vocales = 'aeiou'

>>> random.shuffle(letras)
>>> print(''.join(letras))
rqkmjgvzblsaoicfntxewduphy

>>> for i in range(len(letras)):                    # Manera NO Pythónica
...     if letras[i] in vocales:
...         print('{} en la posición {}'.format(letras[i], i))
...
a en la posición 11
o en la posición 12
i en la posición 13
e en la posición 19
u en la posición 22

>>> for i, letra in enumerate(letras):              # Manera Pythónica
...     if letra in vocales:
...         print('{} en la posición {}'.format(letra, i))
...
a en la posición 11
o en la posición 12
i en la posición 13
e en la posición 19
u en la posición 22
```

Neste exemplo, estamos procurando as posições das vogais em um alfabeto sem ordem. Para isso, usamos a função **enumerate**, que pede uma seqüência e retorna a mesma seqüência associada a seus índices:

```
# Ejemplos de enumerate. Hay que envolverlo en list() o recorrerlo en un for
>>> abcde = sorted(letras)[:5]

>>> list(enumerate(abcde))              # Devuelve secuencia con sus índices
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]

>>> list(enumerate(abcde, 10))          # Podemos decirle en qué índice empieza
[(10, 'a'), (11, 'b'), (12, 'c'), (13, 'd'), (14, 'e')]
```

Iteradores

A seguir, vamos entender o que são os *iteradores*, e qual é o protocolo que permite que qualquer seqüência Python seja passada através de um ***for***. Além disso, veremos que, sobrecarregando um par de funções, podemos proporcionar a nossas próprias classes estas capacidades de iteração.

Objetos Iteríveis

Como sabemos, podemos passar por qualquer seqüência em laço ***for:***

```
>>> for num in [1, 2, 3, 4, 5, 6]:
...     print(num ** 2, end= ' ')
1 4 9 16 25 36

>>> for num in [12, 38, 99, 1]:
...     print(num / 2, end= ' ')
6.0 19.0 49.5 0.5

>>> for letra in 'Python':
...     print(letra.upper(), end=' ')
P Y T H O N
```

Na verdade, é possível atravessar estas seqüências porque elas são **iteráveis**. O conceito de **iterável** é uma generalização do termo seqüência. Um objeto iterável é um objeto que atende a uma destas condições:

- É fisicamente armazenado como uma seqüência
- Ele produz um resultado após o outro no contexto de uma ferramenta de iteração, como um laço for, uma lista para compreensão, etc.

Outra maneira de entender o conceito de um objeto **iterável** é entender que um **iterável** é ou uma seqüência ordenada fisicamente (como uma lista, tupla, etc.) ou um objeto que se comporta virtualmente como uma seqüência.

O Protocolo de Iteração de Python

Mas o que significa que um objeto se comporta como uma seqüência virtual? Basicamente significa que ele implementa um protocolo, chamado protocolo de iteração, que define como um objeto deve se comportar para que ele seja capaz de retornar um elemento após o outro sob demanda quando quisermos atravessá-lo.

A melhor maneira de entender o protocolo de iteração é vê-lo funcionando em objetos construídos em Python. Vamos olhar para isso com o tipo **file**, que usaremos para passar linha por linha.

Primeiro, vamos criar o arquivo, que conterá uma versão aparada do Zen de Python:

```
>>> zen = '''\
... Bello es mejor que feo.
... Explícito es mejor que implícito.
... Simple es mejor que complejo.
... Complejo es mejor que complicado.
... '''
...
>>> f = open('short.zen.txt', 'w')
>>> f.writelines(zen)   # Escribe el
fichero
>>> f.close().          # Cierra el fichero
```

Agora que o temos, vamos ler esse arquivo linha por linha, à mão:

```
>>> f = open('short_zen.txt', 'r')
>>> f.readline()
'Bello es mejor que feo.\n'

>>> f.readline()
'Explícito es mejor que implícito.\n'

>>> f.readline()
'Simple es mejor que complejo.\n'

>>> f.readline()
'Complejo es mejor que complicado.\n'

>>> f.readline()          # Devuelve una cadena vacía cuando termina el fichero
''
```

O método ***readline*** nos permite ler o arquivo linha por linha até encontrarmos um fio vazio. Entretanto, como este método funciona?

Na verdade, os arquivos Python implementam um método que tem um comportamento muito semelhante. O método **__next__** também lê uma linha do arquivo cada vez que ele é chamado. Entretanto, quando o arquivo termina, ele gera uma exceção do tipo ***StopIteration***.

```
>>> f = open('short_zen.txt', 'r')
>>> f.__next__()
'Bello es mejor que feo.\n'

>>> f.__next__()
'Explícito es mejor que implícito.\n'

>>> f.__next__()
'Simple es mejor que complejo.\n'

>>> f.__next__()
'Complejo es mejor que complicado.\n'

>>> f.__next__()
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-55-39ec527346a9> in <module>()
----> 1 f.__next__()

StopIteration:
```

Esta função é o que chamamos de *protocolo de iteração* Python. Qualquer objeto que implemente esta função para prosseguir para o próximo resultado e lança a exceção ***StopIteration*** após o último resultado ser considerado como um **iterador**.

Isto implica que qualquer objeto que implemente estas duas regras pode ser incluído em um laço ***for***, em uma lista por compreensão, etc., porque o que esses mecanismos realmente fazem é chamar a função **__next__** em cada iteração.

Na verdade, isto é o que acontece como tipo builtin de os arquivos Python: podemos atravessá-lo em um laço ***for*** para processar suas linhas uma a uma:

```
>>> for linea in open('short_zen.txt'):

...     print(linea.upper(), end='')
BELLO ES MEJOR QUE FEO.
EXPLÍCITO ES MEJOR QUE IMPLÍCITO.
SIMPLE ES MEJOR QUE COMPLEJO.
COMPLEJO ES MEJOR QUE COMPLICADO.
```

A função next

Para simplificar a iteração manual com ***__next__***, Python fornece a função builtin **next** o que nos permite acessar a iteração manual do ***__next__*** mais facilmente. O que realmente a função ***next(obj)*** é chamar diretamente a ***obj.__next()__:***

```
>>> f = open('short.zen.txt', 'r')
>>> f.__next__()
'Bello es mejor que feo.\n'

>>> next(f)
'Explícito es mejor que implícito.\n'

>>> next(f)
'Complejo es mejor que complicado.\n'

>>> next(f)
----------------------------------------
StopIteration         Traceback(most recent
call last)
<ipython-input-77-468f0afdf1b9> in
<module>()
----> 1 next(f)

StopIteration
```

Iteradores e iterables

Agora que começamos a entender como o protocolo de iteração funciona em Python, vamos cavar um pouco mais fundo para entender as diferenças entre objetos **iteráveis**, como as seqüências que vimos no início da unidade e os **iteradores** que acabamos de conhecer.

A função iter

Acabamos de ver como os arquivos Python são **iteradores** desde que eles implementam a função ***__next__***. Vamos ver se o mesmo se aplica a outros tipos de dados, tais como listas:

```
>>> lista = [1, 2, 3]
>>> next(lista)
----------------------------------------
TypeError         Traceback(most recent
call last)
<ipython-input-79-b6aa4a0df19a> in
<module>()
      1 lista = [1, 2, 3]
----> 2 next(lista)


TypeError: 'list' object is not an
interator
```

O que aconteceu? A função ***next*** está produzindo uma exceção do tipo ***TypeError*** nos dizendo que as listas não são **iteradores**. Mas então, como é possível que possamos passar por uma lista em laço ***for***?

Se você se lembra, no início da unidade dissemos que as listas são **iteráveis**, e com o erro que acabamos de receber é claro que um **iterável** não é o mesmo que um **iterador**. Portanto, é hora de entender o que é um **iterável** e como ele difere de um **iterador:**

- **Iterável:** Um objeto iterável é um objeto que devolve um iterador. Ele implementa o ***__iter__***.
- **Iterador:** Um objeto iterador implementa ***__next__***, permitindo que seja iterada em laços ***for***, etc.

Como vemos, los objetos iterables no son iteradores, sino que devuelven iteradores cuando se les pide. Veamos un ejemplo con nuestra lista:

```
>>> li = lista.__iter__()
>>> li.__next__()
1

>>> next(li)
2

>>> next(li)
3

>>> next(li)
----------------------------------------
StopIteration         Traceback(most recent
call last)
<ipython-input-91-deb767b63ff8> in
<module>()
----> 1 next(li)

StopIteration:
```

Para simplificar a iteração manual, Python implementa a função ***iter***. Desta forma ***iter(obj_iterable)*** é equivalente a ***obj_iterable.__iter()__:***

```
>>> li = iter(lista)
>>> next(li)
1

>>> next(li)
2

>>> next(li)
3

>>> next(li)
----------------------------------------
StopIteration         Traceback(most recent
call last)
<ipython-input-95-deb767b63ff8> in
<module>()
----> 1 next(li)

StopIteration:
```

Na verdade, o que fazem os laços ***for***listas, listas por compreensão, etc., é chamar a função ***__iter__*** do objeto **iterável** que eles vão atravessar antes de começar a percorrê-lo. Isto lhe dá o **iterador**, que é o que você realmente atravessa.

Note também que ao passarmos por um objeto **iterador**, consumimo-lo e não podemos mais voltar atrás. Quando chegamos ao fim do iterador e obtemos a exceção ***StopIteration***, dizemos que *consumimos o iterador*. Para executá-lo novamente, teríamos que pedir ao ***iterável*** que criou o **iterador** que o gerasse novamente.

```
>>> li = iter(lista)
>>> next(li)
1

>>> next(li)
2

>>> next(li)
3

>>> next(li)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-100-deb767b63ff8> in <module>()
----> 1 next(li)

StopIteration:


>>> next(li)              # Hemos consumido el iterador: no podemos iterar más
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-101-deb767b63ff8> in <module>()
----> 1 next(li)

StopIteration:


>>> li = iter(lista)      # Generamos otro iterador para volver a recorrerlo
>>> next(li)
1
```

Criando nossos próprios iteradores

Agora que entendemos plenamente como funciona o protocolo de iteração Python e o que são **iteráviles** e **iteradores**, estamos prontos para criar nossos próprios objetos iteradores e iteradores.

Como já vimos, um objeto iterador é aquele que implementa o método ***__next__*** especificando qual será o próximo elemento a retornar após cada chamada.

Sabendo disso, vamos criar nosso primeiro objeto iterador:

```
>>> rep = Repetidor(3, ):
...
...     def __init__(self, veces, item):
...         self.veces = veces
...         self.item = item
...         self.counter = 0
...
...     def __next__(self):
...         if self.counter == self.veces:
...            raise StopIteration('Iterador
consumido')
...         self.counter += 1
...         return self.item
```

Neste exemplo, estamos criando uma classe que nos permite gerar iterações do parâmetro que passamos para o construtor. Com isso, criamos nosso primeiro **iterador**.

Vamos passar por isso manualmente:

```
>>> class Repetidor(3, 'Python', )

>>> next(rep)
'Python'

>>> next(rep)
'Python'

>>> next(rep)
'Python'

>>> next(rep)
---------------------------------------
StopIteration         Traceback(most recent
call last)
<ipython-input-150-000231fd72d3> in
<module>()

----> 1 next(rep)

<ipython-input-146-6c109aef3369> in
__next__(self)
     8   def __next__(self):
     9       if self.counter == self.veces:
----> 10        raise
StopIteration('Iterador consumido')
     11  self.counter += 1
     12  return self.item

StopIteration: Iterador consumido
```

Mas repare que ainda não é possível fazer um laço através dele ***for*** porque nossa classe não é ***iterável:***

```
>>> rep r in Repetidor(3, 'Python!'):
...     print(r, end=' ')
---------------------------------------
TypeError         Traceback(most recent
call last)
<ipython-input-151-e97e3e194464> in
<module>()
----> 1 for r in Repetidor(3, 'Python!'):
      2     print(r, end=' ')

TypeError: 'Repetidor' object is not
iterable
```

Para torná-lo **iterável** você simplesmente implementa o método ***__iter__:***

```
>>> class Repetidor():
...
...     def __init__(self, veces, item):
...         self.veces = veces
...         self.item = item
...         self.counter = 0
...
...     def __next__(self):
...         if self.counter == self.veces:
...           raise StopIteration('Iterador
consumido')
...         self.counter += 1
...         return self.item
...
...     def __iter__(self):
...         return self

>>> for r in Repetidor(3, 'Python!'):
...     print(r, end=' ')

Python! Python! Python!
```

Note que, como o ***Repetidor*** já era um **iterador**, ele mesmo pode retornar usando a referência a ***self.***

Tendo definido os métodos ***__iter__*** y ***__next__***, nossa classe agora é ao mesmo tempo **iterável** e **iteradora**.

Note que é possível criar iteradores que não são iteradores: se eles devolvem um objeto iterador, que não tem que ser o mesmo, isto é suficiente:

```
>>> class Repetidor():
...
...     def __init__(self, veces, *items):
...         self.veces = veces
...         self.items = items
...
...     def __iter__(self):
...         return.items(self.items *
self.veces)     # Devuelve iterador de la
lista

>>> for r in Repetidor(3, 'a', 'b', 'c'):
...     print(r, end=' ')
a b c a b c a b c
```

Neste exemplo, modificamos nossa classe ***Repetidor*** de modo que ela aceita vários elementos e os repete concatenados um certo número de vezes.

Repetidor é agora apenas um **iterável**, pois não implementa o método ***__next__***. Isto significa que com ***__iter__*** não retorna mais por si só, mas retorna um iterador que consiste na lista resultante da combinação dos parâmetros de entrada pelo número de repetições.

```
>>> rep =  Repetidor(3, 'a', 'b', 'c')
>>> it = iter(rep)
>>> isinstance(it,Repetidor)
False
```

A principal vantagem disto é que agora o objeto não consome a si mesmo, pois gera um iterador que é consumido. Dependendo de suas necessidades de projeto, você pode precisar de uma ou outra opção.