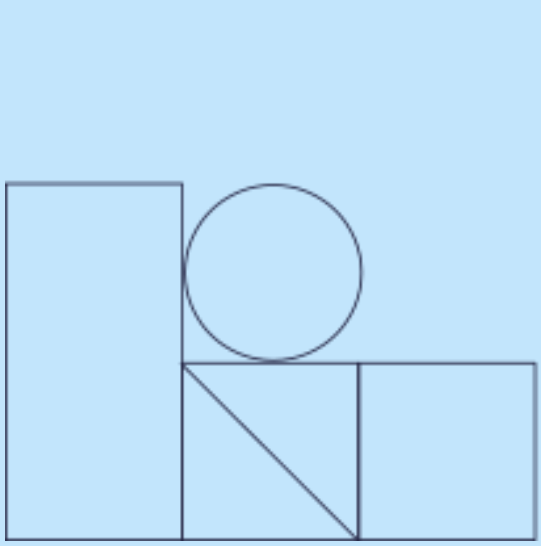
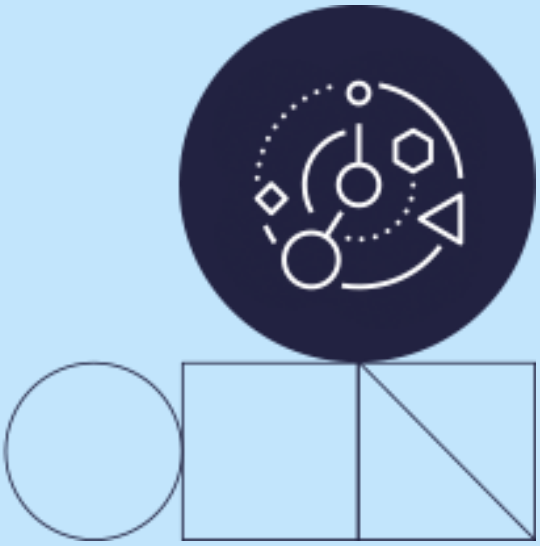
IBM SkillsBuild

IBM

Programação em Python

Estruturas de dados





Índice

Introdução

Até agora, vimos como armazenar um dado em uma variável para poder trabalhar com ele mais tarde. Agora vamos ver as estruturas que a Python tem que ser capaz de trabalhar com a coleta de dados.

As estruturas de dados em Python podem ser entendidas como um tipo de dados compostos, porque na mesma variável podemos armazenar não apenas um dado, mas um número infinito deles. Tais estruturas podem ter características e funcionalidades diferentes.

Existem quatro tipos de estruturas de coleta de dados na linguagem de programação Python:

- A lista (list) é uma coleção ordenada e modificável. Ela permite a duplicação de membros.
- Tupla (tuple); é uma coleção ordenada e imutável. Ela permite a duplicação de membros.
- Conjuntos (Set); é uma coleção que não é ordenada nem indexada. Não há membros duplicados.
- Dicionário (Dictionary); uma coleção não ordenada, modificável e indexada. Sem duplicação de membros.

Ao escolher um tipo de coleção, é útil entender as propriedades desse tipo. Escolher o tipo certo para um determinado conjunto de dados pode significar maior eficiência e segurança.

Listas

Uma lista é um tipo de coleção ordenada e modificável. Ou seja, uma seqüência de **valores de qualquer tipo, ordenados e de tamanho variável**.

As listas em Python são representadas pelo tipo **list** e a sintaxe utilizada para defini-los consiste em indicar uma lista de objetos separados por vírgulas e entre parênteses rectos: [obj1, obj2, ..., objn]

Listas são estruturas de dados que nos permitem armazenar um grande número de valores (equivalentes a arrays em outras linguagens de programação). Eles podem ser expandidos dinamicamente adicionando novos elementos, ou seja, o número de valores que eles contêm pode variar ao longo da execução do programa.

As listas podem conter qualquer tipo de dados: números, cordas, booleans, ... e também outras listas e até mesmo funções, objetos... etc. Diferentes tipos de dados podem ser misturados. **As listas são mutáveis**.

Uma das características importantes das listas é que elas correspondem a uma coleção ordenada de objetos. A ordem na qual os elementos são especificados quando uma lista é definida é relevante e é mantida ao longo de sua vida útil.

Sintaxe das listas:

Criar uma lista é tão simples quanto indicar entre parênteses rectos e separados por vírgulas, os valores que queremos incluir na lista:

nombreLista=**[**elem1**,** elem2**,** elem3...**]**

Diferentes tipos de elementos podem ser misturados.

```
miLista=["Angel", 43, 667767250]
miLista2 = [22, True, "una lista", [1,
2]]
type(miLista)
```

Eu voltaria: list

Eles também podem ser criados a partir de strings:

```
>>> list('Python')    # También se pueden
crear desde strings
['P', 'y', 't', 'h', 'o', 'n']
```

Podemos acessar cada um dos elementos da lista digitando o nome da lista e indicando o índice do elemento entre parênteses rectos.

```
print(miLista[0])        # Imprimiría: Angel
```

Se quisermos acessar um elemento de uma lista incluída em outra lista, teremos que usar este operador duas vezes, primeiro para indicar qual posição da lista externa queremos acessar, e o segundo para selecionar o elemento da lista interna:

```
miLista = [“una lista”, [1, 2]]
mi_var = miLista [1][0]          # mi_var
vale 1
```

Também podemos usar este operador para modificar um item da lista, colocando-o no lado esquerdo de uma atribuição:

```
miLista = [22, True]
miLista [0] = 99                 # Con esto
miLista valdrá [99, True]
```

Para imprimir a lista completa, podemos colocar:

print(miLista[:])

Você pode declarar uma lista vazia, sem elementos.

Como em Java, o primeiro elemento da lista está no índice ou posição **zero**.

Nota: Se eu estabelecer índices negativos o que ele faz **Python** é contar do final ao início a partir de -1, ou seja, no meu caso o elemento

"Angel" poderia ser a posição [0] ou o [-3]. Como no **strings**.

Se eu colocar:

Lista2=lista[-2:] Cria em mim uma sublista com os elementos [43, 667767250]

Podemos utilizar os operadores vistos no tópico anterior para comparar listas

```
In [16]: lista2 = ['t2', 't1', 't3']

In [17]: lista1 == lista2
Out[17]: False

In [20]: lista1 in lista2
Out[20]: False

In [18]: lista1 == ['t1', 't2', 't3']
Out[18]: True

In [19]: lista1 is ['t1', 't2', 't3']
Out[19]: False
```

Uma coisa interessante é que uma lista pode até conter uma função

```
In [23]: def func():
             print("Hola mundo")

In [24]: func
Out[24]: <function __main__.func()>

In [31]: lista = ["texto1", "texto2", func]

In [32]: print(lista)
         ['texto1', 'texto2', <function func at 0x0000012D476D0EE0>]
```

Os elementos de uma lista não têm que ser únicos. O mesmo item pode ser repetido várias vezes na mesma lista

```
In [28]: lista = ["texto", "texto", "texto"]

In [29]: print(lista)
         ['texto', 'texto', 'texto']
```

Métodos de listas

Método	Descrição
append()	Acrescenta um item ao final da lista.
clear()	Elimina items da lista.
copy()	Devolve uma cópia da lista.
count()	Retorna o número de vezes que um item é encontrado na lista.
extend()	Acrescenta itens de uma lista a outra.
index()	Retorna o índice do primeiro elemento cujo valor é o especificado.
insert()	Adiciona um elemento na posição especificada.
pop()	Elimina o item na posição especificada e devolve o item eliminado.
remove()	Elimina o elemento com o valor especificado.
reverse()	Inverta a ordem da lista.
sort()	Ordene a lista.

Os métodos mais comumente utilizados são:

len(), append(), pop(), insert() y **remove().**

Como as listas são coleções *mutáveis*, muitos dos métodos de listas modificam a lista *in-place* em vez de criar uma nova lista, por exemplo **sort()** ou **reverse()**.

```
miLista1 = ["Angel", "Maria", "Manolo",
"Antonio", "Pepe"]
miLista2 = ["Maria", 2, 5.56, True] # Se
puede mezclar diferentes elementos

print (miLista[1])    # Para un elemento
en concreto, se empieza a contar desde la
posición cero.
print (miLista[0:2])    # Empezando desde
cero incluido hasta el dos sin incluir,
esto es, “Angel y María”.
```

Se escrevermos três números (start:end:skip) em vez de dois, o terceiro é usado para determinar quantas posições adicionar um item à lista,

Exemplo:

```
miLista1 = ["Angel", "Maria", "Manolo",
"Antonio", "Pepe"]
miLista2 = ["Maria", 2, 5.56, True] # Se
puede mezclar diferentes elementos

print (miLista[1])    # Para un elemento
en concreto, se empieza a contar desde la
posición cero.
print (miLista[0:2])    # Empezando desde
cero incluido hasta el dos sin incluir,
esto es, “Angel y María”.
```

Listas aninhadas

Uma lista pode conter qualquer tipo de objeto. Isto inclui outra lista. Uma lista pode conter sub-listas, que por sua vez podem conter sub-listas, e assim por diante a uma profundidade arbitrária.

Estas operações funcionam para pequenas matrizes e operações simples. Entretanto, se quisermos operar em grandes matrizes ou em problemas complexos, **Python** bibliotecas estão disponíveis para este tipo de uso. O exemplo principal é ***Numpy***, um projeto de código aberto com forte apoio da comunidade científica e de numerosas organizações privadas. O projeto ***Numpy*** é uma das principais razões por trás do tremendo sucesso de **Python** no campo de **Data Science**.

Tuplas

As tuplas é um tipo de dado complexo, particular da linguagem de programação Python. Um tuple é um objeto idêntico a uma lista, exceto pelas seguintes propriedades:

- Como listas, eles definem uma coleção ordenada de objetos, porém, utilizam a sintaxe (obj1, obj2, ..., objn) em vez de [obj1, obj2, ..., objn].
- As tuplas são imutáveis, ou seja, não podem ser modificados após sua criação.
- Eles não permitem adicionar, apagar, mover elementos (no append, extend, remove).
- Eles permitem extrair porções, mas o resultado da extração é um novo tuple.
- Não permitir buscas (no index)
- Eles permitem verificar se um elemento está no tupla.

As tuplas são representadas dentro de Python pelo tipo de dados **tuple**.

Qual é a utilidade ou vantagem dos tuplos sobre as listas?

- Mais rápido
- Espaço para as mãos (mais otimização)
- Formatar strings.
- Pode ser usado como chave em um dicionário (listas não podem).

A sintaxe básica de uma tupla seria:

```
nombreTupla = (elem1, elem2, elem3…)
```

Os parênteses são opcionais, mas recomendados. As posições são como em listas, **elem1** está na posição 0, el **elem2** na posição 1...etc.

Podemos usar o operador **[]** porque as tuplas, como as listas, fazem parte de uma classe de objetos chamada **seqüências**. As cadeias de texto também são seqüências, portanto não deve ser surpresa que possamos fazer coisas como estas:

```
c = "hola mundo"
c[0]       # h
c[5:]      # mundo
c[::3]     # hauo
```

Como eles são imutáveis, não podemos logicamente fazer um **append**, **pop**...etc.

Quando usar uma tupla ao invés de uma lista?

Há certos casos de uso onde pode ser aconselhável usar um tuple em vez de uma lista:

- A execução do programa é mais rápida quando se manipula um tuple do que quando se lida com uma lista equivalente (isto provavelmente não é perceptível quando a lista ou tuple é pequena).
- Se os valores na coleção devem permanecer constantes durante a vida do programa, o uso de um tuple em vez de uma lista protege contra modificações acidentais.
- Há um outro tipo de dados Python que em breve introduziremos chamado dicionário, que requer um valor imutável como um de seus componentes. Um tuple pode ser usado para este fim, enquanto uma lista não pode.

Acesso aos elementos da tupla

Você pode acessar os elementos tuple referindo-se ao número do índice, entre parênteses rectos:

Exemplo: Imprime o segundo elemento no tupla:

```
thistuple =
("apple", "banana", "cherry")
print(thistuple[1])
```

Indexação Negativa

A indexação negativa significa começar do final, `-1` refere-se ao último elemento, `-2` refere-se ao segundo último elemento, etc.

Exemplo: Imprimir o último elemento do tupla

```
thistuple =
("apple", "banana", "cherry")
print(thistuple[-1])
```

Faixa de índice

Você pode especificar uma faixa de índice especificando por onde começar e onde terminar a faixa.

Quando você especifica um intervalo, o valor de retorno será um novo tuple com os elementos especificados.

Exemplo: Devolve o terceiro, quarto e quinto elementos

```
thistuple =
("apple", "banana", "cherry", "orange"
, "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Nota: A busca deve começar no índice 2 (incluído) e terminar no índice 5 (não incluído).

Lembre-se que o primeiro elemento tem o índice 0.

Faixa de índices negativos

Especifique índices negativos se você quiser iniciar a busca a partir do final do tupla:

Exemplo:

Este exemplo retorna os elementos do índice -4 (incluído) para o índice -1 (excluído)

```
thistuple =
("apple", "banana", "cherry", "orange"
, "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

Mudança de valores da tupla

Uma vez criado um tuple, não se pode mudar seus valores. Os tuplos são imutáveis.

Mas há uma solução. Podemos converter o tuple em uma lista, mudar a lista e converter a lista de volta para uma tupla.

Exemplo: Converter o tuple em uma lista para que ele possa ser modificado:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Travessar uma tupla

Você pode fazer loop através dos elementos do tuple usando um loop for.

Exemplo: Iterar através dos elementos e imprimir os valores.

```
thistuple =
("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Verifique se o item existe

Para determinar se um elemento específico está presente em uma tupla, use a palavra-chave in:

Exemplo: Verificar se a "maçã" está presente no tupla.

```
thistuple =
("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits
tuple")
```

Comprimento da tupla

Para determinar quantos elementos estão em uma tupla, use o método len():

Exemplo: Imprimir o número de elementos no tupla.

```
thistuple =
("apple", "banana", "cherry")
print(len(thistuple))
```

Adicionar items

Uma vez criado uma tupla, não se pode acrescentar elementos a ele. As tuplas são **imutáveis**.

Exemplo: Você não pode adicionar elementos a uma tupla.

```
thistuple =
("apple", "banana", "cherry")
thistuple[3] = "orange" # This will
raise an error
print(thistuple)
```

Criar tupla com um item

Para criar uma tupla com um único elemento, você deve adicionar uma vírgula após o elemento, a menos que Python não reconheça a variável como uma tupla.

Exemplo: Tupla de um artigo, lembre-se da vírgula.

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Eliminar items

Nota: Não se pode apagar elementos em um tuple.

Tuplas não podem ser **mudadas**, para que você não possa remover itens dele, mas você pode removê-los completamente:

Exemplo: A palavra-chave `del` você pode apagar completamente o tuple.

```
thistuple =
("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an
error because the tuple no longer
exists
```

Junte dois tuplas

Para unir dois ou mais tuplas, você pode usar o operador `+` :

Exemplo: Junte-se a dois tuple.

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

O construtor tuple ()

Também é possível utilizar o construtor `tuple ()` para fazer uma tupla.

Exemplo: Usando o método tuple() para fazer uma tupla.

```
thistuple =
tuple(("apple", "banana", "cherry")) #
note the double round-brackets
print(thistuple)
```

Métodos das tuplas

Python tem dois métodos embutidos que você pode usar em tuplas.

Método	Descrição
count()	Retorna o número de vezes que um valor especificado ocorre em uma tuple.
index()	Procura a tupla por um valor especificado e retorna a posição de onde ele foi encontrado.

```
miTupla=("Angel", 4, 5.345, True, 4)
tuplaUnitaria=("Sara",)          #Tupla unitaria. Hay que poner al final ","

print(miTupla[2])                #Igual que en las listas
miLista=list(miTupla)            #Con list convierto una tupla en una lista
miTupla2=tuple(miLista)          #Con tuple convierto una lista en una tupla.
print("Angel" in miTupla)        #Está "Angel" en miTupla?, devuelve True o False
print(miTupla.count(4))          #Cuantas veces se encuentra el elemento 4 en miTupla?
print(len(miTupla))        #Longitud de miTupla
```

Podemos definir o tuple sem os parênteses, isto é conhecido como "embalagem tupla". Em princípio, não é recomendado:

```
miTupla="Angel", 4, 5.345, True, 4
#Desempaquetado de tupla. Permite asignar todos los elementos de una tupla a diferentes variables:
miTupla=("Angel", 4, 5.345, True, 4)
nombre, num1, num2, valor1, num3=miTupla
```

Só isso já nos atribuiu os valores de miTupla para as variáveis que definimos, podemos testar imprimindo-as:

```
print(nombre)
print(num1)
print(num2)
print(valor1)
print(num3)
```

Dicionários

Os dicionários, também chamados arrays associativos, devem seu nome ao fato de serem coleções que relacionam uma chave e um valor.

Um dicionário é uma coleção não ordenada, modificável e indexada. Em Python, os dicionários são escritos entre aparelhos e têm chaves e valores.

São estruturas de dados que nos permitem armazenar valores de diferentes tipos (números, strings... etc.) e até mesmo listas e outros dicionários.

A principal característica dos dicionários é que os dados são armazenados associados a uma chave de tal forma que uma associação de valor chave é criada para cada elemento.

Os elementos armazenados não são encomendados.

Exemplo: Criar e imprimir um dicionário.

```
dic = {
    "Nombre":"Santiago",
    "Apellido":"Hernandez",
    "Pais":"España",
    "Ciudad":"Madrid"
}
print(dic)

# Otra forma de definir diccionarios con
la funcion dict()
dic2 = dict(
    Nombre="Santiago",
    Apellido="Hernandez",
    Pais="España",
    Ciudad="Madrid"
)
```

Em **Python** um dicionário seria semelhante ao que em **Java** um **hashtable** ou em **PHP** um **array associativo**.

O primeiro valor é a chave e o segundo é o valor associado à chave. Como chave podemos usar qualquer valor imutável: poderíamos usar números, cordas, booleanos, tufos, ... mas não listas ou dicionários, já que são mutáveis. Isto porque os dicionários são implementados como tabelas de hash, e quando um novo par de valores-chave é introduzido no dicionário, o hash da chave é calculado a fim de encontrar rapidamente a entrada correspondente. Se o objeto chave for modificado após ter sido inserido no dicionário, seu hash obviamente também mudará e ele não poderá ser encontrado.

Métodos de dicionário

Python tem um conjunto de métodos embutidos que podemos usar em dicionários.

Método	Descrição
clear()	Deleta todos os itens de um dicionário.
copy()	Devolve uma cópia de um dicionário.
fromkeys()	Devolve um dicionário com as chaves e valores especificados.
get()	Retorna o valor de uma chave.
items()	Retorna uma lista contendo um tuple para cada par de valores-chave.
keys()	Retorna uma lista contendo as chaves do dicionário.
pop()	Elimina o item com a chave especificada.
popitem()	Elimina o último par de valores-chave inserido.
setdefault()	Retorna o valor da chave especificada. Se não existir, insira a chave com o valor especificado.
update()	Atualiza o dicionário com o par de valores-chave que você especificar.
values()	Retorna uma lista de valores de dicionários.

Bytes e Bytearray

O que são bytes em Python?

Um **byte** é um local de memória com um tamanho de 8 bits. Um objeto **bytes** é uma seqüência imutável de **bytes**, conceitualmente semelhante a um **string**.

El objeto **bytes** é importante porque qualquer tipo de dado gravado em disco é escrito como uma seqüência de **bytes**, Os inteiros ou cadeias de texto são seqüências de **bytes**. Isto faz com que a cadeia de **bytes** em uma cadeia de texto ou um número inteiro, é a forma pela qual é interpretado.

Sets, Conjuntos

Os conjuntos são um tipo de dados Python que permite que vários itens sejam armazenados em uma única variável. Ao contrário das listas, a coleção de elementos que compõem um conjunto:

- Não pode ser indexado.
- Não respeita uma ordem.
- Não pode conter valores duplicados.

Os sets são representados dentro da pitão com o tipo de dados **set**.

A sintaxe utilizada para definir um conjunto em Python é a seguinte:

{val1, val2, ..., valn}

Também chamados **sets**. Os **sets** (conjuntos) são um tipo de dados básicos Python diferentes de seqüências e dicionários, mas ainda assim muito úteis. Os **sets** são conjuntos de objetos mutáveis não ordenados, (únicos e não ordenados), que são baseados na noção matemática de conjuntos.

De hecho, implementan una serie de métodos que permiten trabajar con conjuntos de objetos de la misma manera que lo hacemos en conjuntos matemáticos. Podemos hacer intersecciones, uniones, diferencias...etc.

No obstante, un conjunto no puede incluir objetos mutables como listas, diccionarios, e incluso otros conjuntos.

Son útiles cuando queremos trabajar con datos masivos y queremos extraer información relevante de ellos.

Definição de métodos

Python tem um conjunto de métodos embutidos que você pode usar em conjuntos.

Método	Descrição
add()	Acrescentar um item ao set.
clear()	Elimina todos os elementos do set.
copy()	Devolve uma cópia do set.
difference()	Devolve um conjunto contendo as diferenças entre dois ou mais sets.
difference_update()	Deleta elementos do set que estão incluídos em outro conjunto.
discard()	Elimina o item especificado.
intersection()	Devolve um set que é a intersecção resultante de dois outros conjuntos.
intersection_update()	Deleta elementos do set que não estão presentes em outro conjunto.
isdisjoint()	Relata se dois sets têm ou não uma interseção.
issubset()	Relata se outro set contém ou não este conjunto.
issuperset()	Relata se este set contém ou não outro conjunto.
pop()	Apaga um item do set, não poderemos escolher qual deles.
remove()	Elimina um item específico.
symmetric_difference()	Devolve um set com as diferenças simétricas de dois sets.
symmetric_difference_update()	Insira as diferenças simétricas entre este set e outro.
union()	Devolve um set com a união de dois sets.
update()	Atualize o set com a união deste set e outros.