# Capítulo 3

# Capítulo 3: Tipos de dados collections

Existem tipos de dados mais avançados chamados de `collections`, que funcionam como contêineres que pode armazenar um ou mais dados de tipos variados. Neste capítulo nós falaremos das **collections que já vem com python**, chamadas de *built-in collections*.

## Pensando como um programador

Antes de começarmos a ver o conteúdo de mais um capítulo, eu gostaria de falar sobre como pensar com um programador. E o que isso quer dizer, exatamente?

Bem, a cada exercício, nós estamos resolvendo problemas mais complexos. Nós começamos com problemas que só precisavam de variáveis e tipos de dados para serem resolvidos; depois resolvemos problemas que combinavam estes com controle de fluxo. Nós estamos aprendendo ferramentas e técnicas isoladas que podem ser combinadas para resolver problemas maiores.

E é a isto que me refiro quando digo "pensar como um programador". Eu quero que você aprenda a olhar para problemas com outros olhos. Quero que divida problemas complexos em problemas menores, que possam ser resolvidos com as ferramentas que você tem a sua disposição.

Quando encontramos um problema grande e complexo, normalmente não sabemos nem por onde começar a resolvê-lo. O que devemos fazer é dividi-lo em problemas menores, de resolução mais simples, e resolvermos um por um; pedacinho por pedacinho.

Uma técnica bem útil pra este objetivo é utilizar o `TODO`, que pode ser traduzido como "para fazer". Imagine que você foi contratado para fazer um programa. O primeiro passo, é definir quais serão as funcionalidades dele (dividindo um problema grande e complexo em problemas menores) e, para cada funcionalidade, definimos os `TODO`, ou seja, definimos o que cada funcionalidade fará, quantas variáveis vai precisar, de que tipos de dados, como o controle de fluxo será, e assim sucessivamente.

Eu espero que, a partir de agora, você passe a olhar todos os problemas desta maneira. Pensando em quais passos você terá que dar para resolvê-los. E olhar para o que estamos aprendendo como novas ferramentas e novas maneiras de fazer o mesmo. Existem diversas formas de resolver o mesmo problema e cada um usará as ferramentas de sua maneira.

---

# Lista | List

A lista é um tipo de dado que contém um ou mais elementos ordenados. Dentro de uma lista nós podemos ter strings, números, outra listas, entre outros.

```
my_list = ['Hello', 'World', 1, 2.0, 3]
print(my_list)
```

```
['Hello', 'World', 1, 2.0, 3]

Process finished with exit code 0
```

Neste exemplo temos uma lista com 5 elementos de diversos tipos. Quando a imprimimos, python imprime a lista inteira, incluindo os colchetes.

> ***Nota:*** Python reconhece uma lista por causa dos colchetes.

Antes de vermos exemplos práticos de uso de uma lista, precisamos aprender a manipulá-la.

## Acessando uma lista

### Acessando os elementos da lista

Como falamos anteriormente, listas são ordenadas, ou seja, todos os elementos de uma lista estão em uma posição fixa dentro dela. Sabendo disso, nós podemos acessá-los através de seu índice (index em inglês).

```
my_list = ['Hello', 'World', 1, 2.0, 3]
# indice:    [0]      [1]   [2] [3] [4]
print(my_list[0])
```

Aqui estamos dizendo para python imprimir o elemento que se encontra na posição 0 dentro da lista.

```
Hello

Process finished with exit code 0
```

> ***Importante:*** O índice sempre começa em 0. Então, numa lista de 5 elementos, nós teremos os índices 0, 1, 2, 3, 4. índices são sempre `Int`.

### Acessando lista dentro de uma lista

Uma lista pode conter elementos de qualquer tipo, incluindo `lists` , `dictionaries` e outros tipos de dados que veremos mais à frente.

```
names_and_numbers = [['Rebeca', 'Chelsey', 'Caroline'], [21, 73, -102]]
print(names_and_numbers[0])
```

Quando tentamos acessar o índice 0 da lista `names_and_numbers`, python imprimirá a lista de nomes.

```
['Rebeca', 'Chelsey', 'Caroline']

Process finished with exit code 0
```

Então como eu acesso a string `'Rebeca'`?

```
print(names_and_numbers[0][0])
```

Primeiro acessamos o elemento no índice 0 da lista `names_and_numbers`, que é a lista de nomes, depois acessamos o elemento de índice 0 nela.

```
Rebeca

Process finished with exit code 0
```

### Acessando o ultimo elemento

Python lê os elementos de uma lista da esquerda pra direita.

```
fruits = ['Apple', 'Banana', 'Grape', 'Strawberry', 'Orange']
#            [0]      [1]      [2]        [3]          [4]
```

Quando passamos um índice negativo, python passa a ler a lista da direita para a esquerda.

```
fruits = ['Apple', 'Banana', 'Grape', 'Strawberry', 'Orange']
#          [-5]     [-4]      [-3]       [-2]        [-1]
```

Sendo assim, o elemento de índice `-1` é o ultimo elemento da lista.

## Modificando uma lista

Agora que nós sabemos como acessar cada elemento dentro de uma lista, vamos ver como modificá-los.

### Substituindo elementos

Para substituir um elemento na lista, nós primeiro acessamos o elemento e então atribuímos um valor, como fizemos com variáveis anteriormente.

```python
fruits = ['Apple', 'Banana', 'Grape']
fruits[0] = 'Orange'
print(fruits)
```

```
['Orange', 'Banana', 'Grape']

Process finished with exit code 0
```

## Adicionando elementos no final

Para adicionar elementos no final de uma lista, nós utilizamos o método `list_name.append()`.

```python
fruits = []
fruits.append('Orange')
fruits.append('Banana')
fruits.append('Grape')
print(fruits)
```

Neste exemplo, estamos criando uma lista vazia chamada `fruits` e, a seguir, utilizamos o método `append()` para adicionar elementos no seu final, resultando em:

```
['Orange', 'Banana', 'Grape']

Process finished with exit code 0
```

> ***Nota:*** Nós estudaremos métodos a fundo posteriormente. Por enquanto, apenas aprenda a usá-los.

## Adicionando elementos em posições específicas

O método que utilizamos para adicionar um elemento a uma lista em uma posição especifica é o `insert()`. Ao utilizá-lo, nós precisamos passar o índice e o dado que queremos adicionar.

```python
fruits = ['Orange', 'Banana', 'Grape']
fruits.insert(1, "apple")
print(fruits)
```

Nós estamos dizendo para python adicionar a `string "apple"` no índice 1 da lista `fruits`. É importante ressaltar que python não vai substituir "Banana" por "apple", ele vai empurrar "Banana" pra frente e adicionar "apple" onde "Banana" estava.

```
['Orange', 'apple', 'Banana', 'Grape']

Process finished with exit code 0
```

## Removendo elementos por índice

Para remover elementos de uma lista nós utilizamos o comando `del`.

```
fruits = ['Orange', 'Banana', 'Grape']
del fruits[1]
print(fruits)
```

```
['Orange', 'Grape']

Process finished with exit code 0
```

Com o comando `del` nós simplesmente deletamos o elemento da lista.

## Removendo e atribuindo um elemento da lista com pop()

O método `pop()` tem 3 usos. Ele pode:

1. remover o ultimo elemento da lista
2. remover um elemento específico da lista
3. remover o elemento e atribuí-lo a uma variável

Quando apenas utilizamos o método `pop()`, ele removerá o ultimo elemento da lista:

```
fruits = ['Orange', 'Banana', 'Grape']
fruits.pop()
print(fruits)
```

```
['Orange', 'Banana']

Process finished with exit code 0
```

Quando passamos o índice do elemento como argumento, ele removerá o elemento especifico. No exemplo a seguir, removeremos o elemento do índice 0.

```
fruits = ['Orange', 'Banana', 'Grape']
fruits.pop(0)
print(fruits)
```

```
['Banana', 'Grape']

Process finished with exit code 0
```

Quando queremos remover o item de uma lista e atribuí-lo a uma variável, também utilizamos o `pop()`

```
fruits = ['Orange', 'Banana', 'Grape']
my_favorite_fruit = fruits.pop(1)
print(fruits)
print(my_favorite_fruit)
```

Neste último caso, nós não estamos apenas acessando o elemento na lista e atribuindo-o à variável. Nós estamos removendo `'Banana'` e atribuindo à variável `my_favorite_fruit`

```
['Orange', 'Grape']
Banana

Process finished with exit code 0
```

## Removendo um elemento por valor

Até agora só vimos maneiras de remover um elemento por índice. Mas e quando sabemos o valor e não sabemos onde ele se encontra na lista? Para estes casos, utilizamos o método `remove()`.

```
fruits = ['Orange', 'Banana', 'Grape']
fruits.remove('Banana')
print(fruits)
```

```
['Orange', 'Grape']

Process finished with exit code 0
```

Aqui estamos dizendo para python encontrar a palavra 'Banana' na lista e removê-la.

> ***Importante:*** O método `remove()` remove apenas o primeiro elemento com este valor!

```
fruits = ['Orange', 'Banana', 'Grape', 'Banana']
fruits.remove('Banana')
print(fruits)
```

```
['Orange', 'Grape', 'Banana']

Process finished with exit code 0
```

## Combinando listas

Listas podem ser concatenadas, ou combinadas, com o operador `+`:

```
numbers = [1, 2, 3]
animals = ['cat', 'dog', 'capybara']
```

```
numbers_and_animals = numbers + animals

print(numbers_and_animals)
```

```
[1, 2, 3, 'cat', 'dog', 'capybara']

Process finished with exit code 0
```

### Repetindo os valores

Quando multiplicamos uma lista por uma `int`, repetimos seus elementos dentro dela, como vemos abaixo:

```
animals = ['cat', 'dog', 'capybara']
print(animals * 2)
```

```
['cat', 'dog', 'capybara', 'cat', 'dog', 'capybara']

Process finished with exit code 0
```

## Loops e listas

Agora que já sabemos como manipular uma lista, nós podemos falar do seu verdadeiro poder. Nós podemos utilizar um loop para iterar por todos os elementos da lista e executar um bloco de código para cada um dos elementos. Calma, eu sei que tá começando a soar complicado de novo. Então vamos ver isso com calma.

Imagina que nós temos uma lista de nomes de convidados para uma festa. Nós queremos criar um programa que dirá "Olá" para todos os convidados. Com o que aprendemos até agora, fazer algo assim seria muito trabalhoso.

```
print("Hello, Joseph!")
print("Hello, Johnny!")
print("Hello, Richard!")
print("Hello, Sabine!")
print("Hello, Jessica!")
```

Já que estamos lidando com uma grande quantidade de valores, podemos utilizar loops e listas em conjunto para lidar com isso.

Com o que aprendemos até agora, nós podemos acessar os elementos de uma lista por seus índices, então podemos utilizar um `for` loop para isto.

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']
```

```
for index in range(len(names)):
        print(names[index])
```

Aqui estamos criando um `for` loop que começa em 0 e vai até o tamanho da lista `names`. A lista `names` tem 5 elementos, então a função `range()` function irá de 0 a 5, não incluso. Na primeira iteração, python imprimirá `names[0]`, que é o valor `"Joseph"`, na segunda iteração, imprimirá `names[1]`, que é `"Johnny"` e assim sucessivamente.

É assim que normalmente faríamos em outras linguagens. Python, entretanto, nos oferece uma solução muito mais elegante para este problema.

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']

for name in names:
    print(f"Hello, {name}!")
```

Neste código, o `for` loop vai começar e em sua primeira iteração, pegará o primeiro elemento em `names` e atribuirá à variável `name`, ou seja, na primeira iteração `name = "joseph"`, python imprimirá a mensagem na tela, e o loop voltará ao começo, na segunda iteração, `name = "Johnny"`, e assim sucessivamente até chegar ao final da lista, resultando em:

```
Hello, Joseph!
Hello, Johnny!
Hello, Richard!
Hello, Sabine!
Hello, Jessica!

Process finished with exit code 0
```

Caso ainda não tenha ficado claro, tente ler o código da seguinte maneira: "para cada nome em `names`, faça:".

Como você pode ver, com poucas linhas de código nós podemos modificar diversos valores de uma única vez combinando listas e loops!

## Os operadores `in` e `not in`

Nós usamos os operadores `in` e `not in` para checar se um valor está na lista ou não, respectivamente.

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']

if "Johnny" in names:
    print("yay")
```

Este código checa se a `string` "Johnny" faz parte da lista `names` e imprime "yay" caso faça.

```
yay

Process finished with exit code 0
```

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']

if "Rebeca" not in names:
    print("boo")
```

Já este, checa se a `string` "Rebeca" não faz parte da lista `names` e, caso não faça, imprime "boo" na tela.

```
boo

Process finished with exit code 0
```

> ***Importante:*** Letras maiúsculas e minúsculas são vistas como diferentes pelo computador. Sendo assim `"Rebeca"` não é a mesma coisa que `rebeca`.

## List Slices

Nós já sabemos como acessar um elemento dentro de uma lista, ou acessar todos os elementos de uma lista. Mas e quando nós quisermos acessar apenas alguns elementos dentro da lista? Uma opção seria criar um `for` loop e utilizar condições para determinar quais elementos serão selecionados. Mas python nos permite fazer isso de uma forma: utilizando list slices.

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']

print(names[2:5])
```

Neste bloco de código, nós estamos dizendo à python para imprimir os elementos na lista `names` do índice 2 até o 5, não incluso, ou seja, índices 2, 3 e 4.

```
['Richard', 'Sabine', 'Jessica']

Process finished with exit code 0
```

List slices funcionam da mesma forma que a função `range()` que aprendemos anteriormente.

```
list_name[começo:fim:passo]
```

> ***Nota:*** O valor padrão para `passo` é 1, o que significa "de um em um".

> ***Importante:*** Não esquecer que o slice vai de um número até o outro sem incluí-lo. Se eu digo `[0:3]`, eu estou dizendo do 0 até o 3 sem incluí-lo, ou seja, 0, 1, 2.

Quando não colocamos um valor para o começo, estamos dizendo "comece do índice 0".

```
numbers = [10, 20, 30, 40, 50, 60]
my_lucky_numbers = numbers[:3]

print(my_lucky_numbers)
```

Neste bloco de código nós temos um variável do tipo `list` chamada `numbers`, e estamos criando uma nova variável chamada `my_lucky_numbers` que também será uma lista, e estamos atribuindo a ela os elementos do começo da lista `numbers` até o elemento de índice 3, não incluso.

```
[10, 20, 30]

Process finished with exit code 0
```

Quando não colocamos um valor para o fim, estamos dizendo "vá até o final".

```
numbers = [10, 20, 30, 40, 50, 60]
my_lucky_numbers = numbers[2:]

print(my_lucky_numbers)
```

```
[30, 40, 50, 60]

Process finished with exit code 0
```

Você consegue me dizer o que o bloco de código no próximo exemplo faz?

```
numbers = [10, 20, 30, 40, 50, 60]
my_lucky_numbers = numbers[::2]

print(my_lucky_numbers)
```

Para descobrirmos, precisamos lembrar que list slices funcionam com três valores `[começo:fim:passo]`. Não passamos nenhum valor para o começo, então estamos dizendo "comece do índice 0"; também não passamos nenhum valor para o fim, o que quer dizer "vá até o final"; e, por fim, estamos dizendo para ir de 2 em 2. Este código então irá ler toda a lista e atribuirá à `my_lucky_numbers` os elementos de índice 0, 2 e 4.

```
[10, 30, 50]

Process finished with exit code 0
```

## Função sorted()

A função `sorted()` organiza a lista sem alterar a original.

```python
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']

print(f"Sorted: {sorted(names)}")
print(f"Original: {names}")
```

```
Sorted: ['Jessica', 'Johnny', 'Joseph', 'Richard', 'Sabine']
Original: ['Joseph', 'Johnny', 'Richard', 'Sabine', 'Jessica']

Process finished with exit code 0
```

## Função len()

A função `len()` verifica a quantidade de elementos em uma lista.

```python
numbers = [1, 2, 3, 4, 5, 6]
length_of_list = len(numbers)
print(length_of_list)
```

```
6

Process finished with exit code 0
```

> ***Importante:*** Esta função também pode ser utilizada com outros tipos de dados, como `dictionary`, `tuple`, `string`, entre outros.

## Função max()

A função `max()` retorna o maior valor em uma lista.

```python
numbers = [10, 20, 30]

print(max(numbers))
```

```
30

Process finished with exit code 0
```

## Função min()

A função `min()` returna o menor valor em uma lista.

```
numbers = [10, 20, 30]

print(min(numbers))
```

```
10

Process finished with exit code 0
```

## Métodos mais comuns

Posteriormente nós veremos com calma o que são métodos e como criá-los. Por enquanto, só precisamos saber que eles são chamados utilizando a "Notação de ponto", em inglês, "Dot Notation", que nós já utilizamos:

```
numbers = []
numbers.append(1)
```

Como podemos ver, dot notation nada mais é do que utilizar um ponto para chamar um método. Nós veremos outros métodos antes de aprender como criá-los, e todos eles serão chamados através da dot notation.

### index()

O método `index()` retorna, ou seja, têm como resultado, o índice de um elemento da lista.

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']
jessica_index = names.index("Jessica")
print(jessica_index)
```

```
4

Process finished with exit code 0
```

### sort()

O método `sort()` organiza uma lista, seja em ordem do menor para o maior ou em ordem alfabética.

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']
names.sort()
print(names)
```

```
['Jessica', 'Johnny', 'Joseph', 'Richard', 'Sabine']

Process finished with exit code 0
```

Nós também podemos organizar a lista de maneira reversa com ele:

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']
names.sort(reverse=True)
print(names)
```

```
['Sabine', 'Richard', 'Joseph', 'Johnny', 'Jessica']

Process finished with exit code 0
```

## reverse()

O método `reverse()` reverte a ordem da lista.

```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']
names.reverse()
print(names)
```

```
['Jessica', 'Sabine', 'Richard', 'Johnny', 'Joseph']

Process finished with exit code 0
```

## count()

O método `count()` recebe um argumento e conta quantas vezes este argumento aparece na lista.

```
names = ["Jessica", 'Jessica', 'Richard', 'Sabine', 'Jessica']
counter = names.count("Jessica")

print(counter)
```

Neste exemplo, passamos como argumento a string "Jessica". Então python contará quantas vezes a string "Jessica" aparece na lista e atribuirá este valor à variável `counter`. Resultando em:

```
3

Process finished with exit code 0
```

O elemento "Jessica" aparece 3 vezes na lista `names`.

## Embelezando listas

Pode acontecer de nossas listas serem muito grandes, terem muitos elementos, o que as tornam difíceis de ler. Nestes casos, podemos escrever a lista de uma maneira um pouco diferente, para facilitar a leitura.

```
hello_list = [
    "hello",
    "World",
    "This",
    "Is",
    "Doggo"
]
```

A lista continuará funcionando perfeitamente.

> ***Importante:*** Não se esqueça da vírgula entre os elementos!

## Exercícios 5

1. Crie um programa que, dada a lista abaixo, encontre o maior e o menor número e imprima-os na tela.

```
numbers = [191, 78, 67, 195, 51, 154, 28, 45, 186, 106]
```

2. **Bonus:** Faça o exercício 1 sem utilizar as funções `min()` e `max()`.
3. Crie um programa que imprima a lista abaixo sem nenhum número repetido e em ordem crescente:

```
numbers = [6, 2, 5, 6, 2, 7, 1, 9, 1, 7, 6, 4, 2, 6]
```

4. Ainda utilizando a lista anterior, crie um programa que ache o número que mais se repete na lista, imprima-o na tela, juntamente com quantas vezes ele se repete.
5. **Bonus:** Faça o exercício 3 sem utilizar o método `count()`.

---

# Revisitando strings

Nós já sabemos o que são `strings` e como elas funcionam, entretanto, agora que aprendemos mais conceitos, podemos olhá-las de uma outra maneira. `Strings` são como listas de caracteres alfanuméricos. Sendo assim, nós podemos, por exemplo, acessar cada caractere através do seu índice:

```
word = "hello"
print(word[0])
```

Aqui nós estamos acessando o elemento de índice 0 na `string`.

```
h

Process finished with exit code 0
```

Nós também podemos acessar os elementos da `string` utilizando um `for` loop:

```
word = "hello"

for letter in word:
    print(letter, end=" ")
```

```
h e l l o
Process finished with exit code 0
```

Ou usar a função `len()` para descobrir quantos elementos há na `string`:

```
word = "hello"

letters = len(word)
print(letters)
```

```
5

Process finished with exit code 0
```

Porém, `string` é um tipo de dado diferente de `list`. Por conta disto, ele tem métodos diferentes. Neste módulo, nós veremos algumas outras formas de manipulação e métodos relacionados à `string`.

## String slices

Assim como listas, nós também podemos usar slices em strings com seus índices. Funciona exatamente da mesma forma que **list slices**.

```
name = "Rebeca"

print(name[2:5])
```

Com nosso conhecimento de list slices, o que você acha que será impresso?

```
bec

Process finished with exit code 0
```

Correto! Python imprime do índice 2 ao índice 5, não incluso. Então, índices 2, 3 e 4 da string `name`.

> ***Nota:*** Lembre-se que índices começam do 0!

Tudo o que nós aprendemos sobre **list slices** é aplicado à strings. E esta é uma ferramenta muito poderosa e útil para manipulá-las. Não hesite em voltar e ler novamente!

## Aspas triplas

As `string` que vimos até agora estavam todas em uma unica linha. Nós podemos utilizar o que aprendemos até agora para construir strings com várias linhas, como, por exemplo, com quebras de linha, concatenação, ou até mesmo diversos `print()`. Existe, porém, uma outra forma: as aspas tripas.

```
text = """My dear,

I'm sending you this text because I will not be able to get there on time.
I'm stuck in traffic.
Save me some cake!

Love,
me.
"""

print(text)
```

```
My dear,

I'm sending you this text because I will not be able to get there on time. I'm stuck in traffic.
Save me some cake!

Love,
me.


Process finished with exit code 0
```

> ***Nota:*** Podem ser usadas aspas únicas `'''` ou duplas `"""`.

## Raw strings

Nós vimos que podemos ignorar caracteres e adicionar caracteres especiais utilizando a barra inversa, ou *escape character*. Mas e se nós quisermos que o que quer que seja

digitado pelo usuário seja mantido na `string`, incluindo barras inversas. É aqui que entram as raw strings or `r string`.

```
print(r"hello, \"my friends\"!")
```

```
hello, \"my friends\"!

Process finished with exit code 0
```

Na `r string` python trata tudo na string como parte dela, mesmo que você tente passar caracteres especiais.

## Métodos mais comuns

Existem outros métodos e você pode encontrá-los na documentação oficial, livros, procurando no google ou perguntando a inteligências artificiais. Estes são apenas os mais comumente usados.

### upper()

O método `upper()` faz com que as letras das palavras fiquem maiúsculas.

```
name = "Fatma"
name = name.upper()

print(name)
```

Aqui nós estamos criando uma varável do tipo `string` com o valor "Fatma", depois estamos atribuindo à variável `name` o valor original modificado para que todas as letras estejam maiúsculas.

```
FATMA

Process finished with exit code 0
```

### lower()

O método `lower()` faz com que as letras fiquem minúsculas.

```
name = "Fatma"
name = name.lower()

print(name)
```

```
fatma

Process finished with exit code 0
```

## isupper() e islower()

Checam se os caracteres da `string` são todos maiúsculos ou minúsculos, respectivamente.

```
word1 = "HELLO"
word2 = "WORLD"

print(f"{word1} is upper? = {word1.isupper()}")
print(f"{word2} is lower? = {word2.islower()}")
```

```
HELLO is upper? = True
WORLD is lower? = False

Process finished with exit code 0
```

## capitalize()

Transforma a primeira letra da `string` em maiúscula.

```
text = 'rodrigo has a blue car.'
text = text.capitalize()

print(text)
```

```
Rodrigo has a blue car.

Process finished with exit code 0
```

## title()

Transforma todas as palavras da `string`, fazendo com que elas comecem com letras maiúsculas.

```
name = "james bond"
name = name.title()

print(name)
```

```
James Bond

Process finished with exit code 0
```

## startswith() and endswith()

Checam se a `string` começa ou termina com os argumentos passados para eles, respectivamente.

```
message = "Hello, world! I'm here to learn how to code in python."

print(message.startswith("Hello"))
print(message.endswith("world"))
```

```
True
False

Process finished with exit code 0
```

## split()

O método `split()` pode separa uma string em elementos e retorna uma lista deles.

```
message = "Python is so much fun. I wish I had learned it sooner"
message = message.split()

print(message)
```

```
['Python', 'is', 'so', 'much', 'fun.', 'I', 'wish', 'I', 'had', 'learned', 'it', 'sooner']

Process finished with exit code 0
```

Por padrão, ele separa a string pelos espaços, mas nós podemos alterar este comportamento passando o argumento pelo qual queremos separar a string. A seguir, vamos separar a mesma `string` pelo ponto final.

```
message = "Python is so much fun. I wish I had learned it sooner"
message = message.split(".")

print(message)
```

```
['Python is so much fun', ' I wish I had learned it sooner']

Process finished with exit code 0
```

## strip(), rstrip() and lstrip()

Os métodos `strip()`, `rstrip()` e `lstrip()` removem elementos da string. `strip()` remove elementos dos dois lados, `rstrip()` remove apenas do lado direito e `lstrip()` remove apenas do lado esquerdo. Por padrão, eles removem espaços em branco no começo, no final ou em ambos os lados da string.

```
message = "     hello, world!       "

print(f"strip: {message.strip()}")
print(f"rstrip: {message.rstrip()}")
print(f"lstrip: {message.lstrip()}")
```

```
strip: hello, world!
rstrip:      hello, world!
lstrip: hello, world!

Process finished with exit code 0
```

Porém, é possível passar como argumento o elemento que gostaríamos de remover. Talvez seu uso fique mais claro com o exemplo abaixo:

```
message = "______hello, world!______"

print(f"strip: {message.strip('_')}")
print(f"rstrip: {message.rstrip('_')}")
print(f"lstrip: {message.lstrip('_')}")
```

```
strip: hello, world!
rstrip: ______hello, world!
lstrip: hello, world!______

Process finished with exit code 0
```

## replace()

Este método substitui um elemento da `string` por outro. Ele recebe dois argumentos: a parte da string que você quer substituir e a parte da string pela qual quer substituir.

```
text = "oh no, my cat ate all my food!"
print(text)

text = text.replace("cat", "dog")
# troca a palavra "cat" na string por "dog"
print(text)
```

```
oh no, my cat ate all my food!
oh no, my dog ate all my food!

Process finished with exit code 0
```

Neste exemplo nós substituímos a string "cat" por "dog".

## join()

O método `join()` recebe uma lista ou tuple como argumento e concatena cada elemento usando a string como separador.

```
words = ["Hello", "my", "friend", "Doug"]
phrase = " - blah - ".join(words)
print(phrase)
```

```
Hello - blah - my - blah - friend - blah - Doug

Process finished with exit code 0
```

Eu concordo que pareça um método um pouco estranho. Mas talvez um outro exemplo deixe seu uso mais claro. Imagine que você tem uma `list` e deseja colocar todos os seus elementos dentro de uma `string`, separados por espaço.

```
words = ["Hello", "world", "this", "is", "dog"]
phrase = " ".join(words)
print(phrase)
```

Python pegará a `string` que passamos, neste caso, um espaço vazio `" "` e colocará entre a cada um dos elementos da lista passada como argumento.

```
Hello world this is dog

Process finished with exit code 0
```

Em outras palavras, `" ".join(words)` quer dizer "pegue cada elemento da lista `words` e coloque em uma string, separando-os com `" "` ".

## count()

O método `count()` recebe um argumento do tipo `string` e conta quantas vezes ele aparece na frase, semelhante ao programa que criamos no exercício 5-3.

```
text = "oh no, my cat ate all my food!"
```

```
my_counter = text.count("my")
print(my_counter)
```

A palavra "my" aparece duas vezes na `string` dada.

```
2

Process finished with exit code 0
```

## Exercícios 6

1. Crie um programa que recebe uma string do usuário e imprima na tela a mesma string, mas com a primeira metade toda em letras maiúsculas e a segunda metade toda em letras minúsculas.
2. Crie um programa que receba uma frase do usuário e imprima na tela a frase ao contrário. Ex: "Olá mundo" -> "mundo Olá"
3. Crie um programa que verifica se a palavra ou frase digitada pelo usuário é um palíndromo (palavra ou frase que se pode ler, indiferentemente, da esquerda para a direita ou vice-versa).

---

# Compreensão de lista | List comprehension

List comprehension é uma forma mais concisa de se criar listas, e é encontrada em apenas algumas linguagens. Em uma única linha, nós podemos criar um `for` loop, adicionar os elementos na lista e até utilizar condições.

Normalmente, nós criaríamos uma lista da seguinte maneira:

```
numbers = []

for number in range(1, 11):
    numbers.append(number)

print(numbers)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Process finished with exit code 0
```

Porém, com list comprehension, nós podemos escrever a mesma coisa em apenas uma linha:

```
numbers = [number for number in range(1, 11)]
print(numbers)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Process finished with exit code 0
```

O que nós estamos fazendo aqui? Primeiro, criamos uma variável `numbers` e dentro dos colchetes é onde colocamos a list comprehension. Ali, temos `number for number in range(1, 11)` , ou seja, adicione `number` à lista para cada valor de `number` de 1 até 11 (não incluso).

> ***Nota:*** O primeiro `number` dentro da list comprehension é a variável que será adicionada à lista.

Uma outra forma de **ler** a list comprehension é a seguinte:

```
numbers = [numbers.append(number) for number in range(1, 11)]
```

A primeira variável dentro da list comprehension - `number` - será adicionada à lista sendo criada.

Vamos ver outros exemplos. Criaremos uma lista apenas com números pares de 1 até 20 (não incluso).

Sem list comprehension:

```
numbers = []

for number in range(1, 20):
    if number % 2 == 0:
        numbers.append(number)

print(numbers)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18]

Process finished with exit code 0
```

Com list comprehension:

```
numbers = [number for number in range(1, 20) if number % 2 == 0]
print(numbers)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18]

Process finished with exit code 0
```

No exemplo abaixo temos uma lista de nomes e queremos criar outra lista apenas com os nomes começados com a letra a.

Sem list comprehension:

```
names = ["João", "Alice", "Janaína", "Ana", "Bruna", "Eduarda"]
names_with_a = []

for name in names:
    if name.startswith('A'):
        names_with_a.append(name)

print(names_with_a)
```

```
['Alice', 'Ana']

Process finished with exit code 0
```

Com list comprehension:

```
names = ["João", "Alice", "Janaína", "Ana", "Bruna", "Eduarda"]
names_with_a = [name for name in names if name.startswith('A')]
print(names_with_a)
```

```
['Alice', 'Ana']

Process finished with exit code 0
```

## Estrutura

List comprehension podem ser um pouco difíceis de entender a princípio. O que eu quero que você entenda é como ela é estruturada.

1. Primeiro vem a variável da qual você quer adicionar à lista, juntamente com qualquer condição para filtrá-la.

```
list_comprehension = [variable]
```

2. Então vem o `for` loop

```
list_comprehension = [variable for variable in range(10)]
```

3. E finalmente você tem as condições para filtrar o `for` loop.

```
list_comprehension = [variable for variable in range(10) if variable > 5]
```

O exemplo acima é equivalente a isto:

```
list_comprehension = []

for variable in range(10):
    if variable > 5:
        list_comprehension.append(variable)
```

Vamos ver alguns outros exemplos para esclarecer tudo. Preste atenção à estrutura.

Abaixo nós temos uma lista dos meus pokémon favoritos:

```
my_favorite_pokemon = ["Dragonite", "Milotic", "Lapras", "Meganium",
"Gengar", "Serperior"]
```

Mas quando eu jogo, eu só gosto de usar pokémon que começa com a letra M. Então, usando list comprehension, vamos criar uma lista dos meus pokémon favoritos que começam com a letra M.

```
my_favorite_pokemon = ["Dragonite", "Milotic", "Lappras", "Meganium",
"Gengar", "Serperior"]

my_pokemon = [pokemon for pokemon in my_favorite_pokemon if
pokemon.startswith("M")]
```

Eu iterarei por todos os nomes de pokémon na lista `my_favorite_pokemon` e, se seus nomes começarem com a letra M, eu os adicionarei à lista `my_pokemon`.

Novamente: primeiro o que será adicionado à lista, depois o loop e por fim as condições.

Mais um exemplo.

Eu tenho uma lista de pokémon:

```
pokemon = ["Kilowattrel", "Entei", "Lugia", "Crabominable", "Snorlax"]
```

Mas eu acredito firmemente que qualquer pokémon com um nome mais longo que 10 caracteres não é legal. Então eu quero criar uma nova lista que diz "Cool" se o nome do pokémon for menor que 10 caracteres ou "Not cool" se não for..

```
pokemon = ["Kilowattrel", "Entei", "Lugia", "Crabominable", "Snorlax"]

cool_pokemon = ["Cool" if len(name) > 10 else "Not cool" for name in
pokemon]
```

Aqui eu estou adicionando "Cool" se o tamanho do nome for maior que 10 e "Not cool" se não for, para cada nome na lista `pokemon`.

Um último:

Vamos usar a lista `my_favorite_pokemon` e pegar todos os nomes de pokémon que começam com a letra M. Se o tamanho do nome for maior que 7 caracteres, adicionaremos "Cute", se não for, adicionaremos "Not cute".

```
my_favorite_pokemon = ["Dragonite", "Milotic", "Lappras", "Meganium",
"Gengar", "Serperior"]

cute_pokemon = ["Cute" if len(pokemon) <= 7 else "Not cute" for pokemon in
my_favorite_pokemon if pokemon.startswith("M")]
```

Aqui estamos dizendo a python que adicione "Cute" se `len(pokemon) <= 7` ou então "Not cute" se não for, para cada pokémon em `my_favorite_pokemon` que começa com a letra M.

## Exercícios 7

> ***Nota:*** Utilize *list comprehension* para resolver estes exercícios.

1. Dadas as listas abaixo, crie um programa que imprima na tela uma lista apenas com os números em comum entre elas.

```
first_list = [2, 7, 33, 27, 92, 40, 3, 28, 56]
second_list = [90, 12, 23, 7, 38, 29, 56, 13, 2]
```

2. Crie um programa que gere uma lista de 'par' ou 'ímpar' para cada número de 1 até 20 (não incluso). Ex: [ímpar, par, ímpar, par...]
3. Crie um programa que peça ao usuário uma frase e imprima uma lista de todas as palavras com 4 letras ou menos.

---

# Dicionário | Dictionary

`dictionary` ou `dict` é um tipo de dado que guarda diversos elementos, assim como uma lista, e são utilizados para guardar elementos que estão relacionados entre si. Eles são

armazenados em pares chamados de "key" (palavra-chave) e "value" (valor), separados por dois pontos `:`. Para criá-los, utilizamos chaves em vez de colchetes.

```
# person = {key: value, key: value, key: value, ...}
person = {'name': 'Mario', 'age': 25, 'location': 'Mushroom Kingdom'}
```

Além de ser criado com chaves em vez de colchetes, outra grande diferença é que para acessarmos um valor, nós não usamos um índice numérico, nós usamos as palavras-chave.

```
person = {'name': 'Mario', 'age': 25, 'location': 'Mushroom Kingdom'}
print(person['name'])
print(person['age'])
```

```
Mario
25

Process finished with exit code 0
```

Uma outra diferença é que os elementos de um `dictionary` são desordenados, enquanto os de uma lista são ordenados.

```
my_list = [1, 2, 3]
my_other_list = [3, 2, 1]
print(my_list == my_other_list)
```

Neste caso, o resultado é `False` pois os valores de cada índice são diferentes. Listas são ordenada, ou seja, a ordem de cada valor é importante.

```
my_dict = {'first': 1, 'second': 2, 'third': 3}
my_other_dict = {'second': 2, 'third': 3, 'first': 1}
print(my_dict == my_other_dict)
```

Já aqui, o resultado é `True` pois dicionários não são ordenados. Python não se importa com a ordem dos elementos, apenas que eles sejam iguais.

## Acessando elementos

Apesar de termos acabado de ver como acessar os elementos de um `dictionary`, eu quero explicar novamente para que, caso você queira relembrar posteriormente, consiga encontrar facilmente aqui.

Nós acessamos os elementos de um `dictionary` através das palavras-chave.

```
doggo = {"name": "Nugget", "age": 3, "breed": "Golden Retriever"}
```

Neste `dictionary` nós temos 3 palavras-chave - `name`, `age` e `breed` - e nós as usaremos para acessar seus respectivos valores.

```
doggo = {"name": "Nugget", "age": 3, "breed": "Golden Retriever"}

print(f"Name: {doggo['name']}")
print(f"Age: {doggo['age']}")
print(f"Breed: {doggo['breed']}")
```

```
Name: Nugget
Age: 3
Breed: Golden Retriever

Process finished with exit code 0
```

# Modificando um dicionário

## Adicionando elementos

Para adicionar elementos, precisamos passar a palavra-chave a ser adicionada e seu respectivo valor:

```
doggo = {"name": "Nugget", "age": 3, "breed": "Golden Retriever"}

doggo["favorite_toy"] = "bone"
print(doggo)
```

A palavra-chave "favorite_toy" não existe no `dictionary`, então python adiciona a mesma e atribui a ela o valor de "bone".

```
{'name': 'Nugget', 'age': 3, 'breed': 'Golden Retriever', 'favorite_toy': 'bone'}

Process finished with exit code 0
```

## Modificando elementos

Para modificar elementos, apenas atribuímos um novo valor aos mesmos.

```
doggo = {"name": "Nugget", "age": 3, "breed": "Golden Retriever"}

doggo["name"] = "Dorito"
print(doggo)
```

```
{'name': 'Dorito', 'age': 3, 'breed': 'Golden Retriever'}

Process finished with exit code 0
```

### Deletando elementos

Para deletarmos elementos, utilizamos o comando `del`.

```
doggo = {"name": "Nugget", "age": 3, "breed": "Golden Retriever"}

del doggo["breed"]
print(doggo)
```

```
{'name': 'Nugget', 'age': 3}

Process finished with exit code 0
```

> ***Nota:*** O elemento é permanente deletado.

## Loops e dicionários

Assim como fizemos com os dados de tipo `list`, nós podemos utilizar loops para iterar por todos os elementos de um `dictionary`. Mas, para isso, precisamos utilizar alguns métodos.

### keys()

Este método nós da acesso apenas às palavras-chave de um `dictionary`.

```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

print(catto.keys())
```

```
dict_keys(['name', 'age', 'color', 'weight'])

Process finished with exit code 0
```

E, assim, podemos utilizar um loop para acessar todas as palavras-chave.

```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

for key in catto.keys():
    print(key)
```

```
name
age
color
weight

Process finished with exit code 0
```

## values()

Assim como o método `keys()`, o método `values()` nos dá acesso apenas aos valores de um `dictionary`.

```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

print(catto.values())
```

```
dict_values(['KitKat', 5, 'orange', 5.0])

Process finished with exit code 0
```

E com ele nós podemos utilizar um loop para acessar apenas os valores do `dictionary`.

```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

for values in catto.values():
    print(values)
```

```
KitKat
5
orange
5.0

Process finished with exit code 0
```

## items()

O método `ìtems()` nos dá acesso tanto às palavras-chave quanto aos valores.

```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

print(catto.items())
```

```
dict_items([('name', 'KitKat'), ('age', 5), ('color', 'orange'), ('weight', 5.0)])

Process finished with exit code 0
```

Por conta disto, o `for` loop é um pouco diferente. Ele tem duas variáveis.

```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

for key, value in catto.items():
    print(f"Key: {key} -> Value: {value}")
```

```
Key: name -> Value: KitKat
Key: age -> Value: 5
Key: color -> Value: orange
Key: weight -> Value: 5.0

Process finished with exit code 0
```

Como este loop tem duas variáveis, uma para a palavra-chave (`key`) e a outra para o valor (`value`), nós podemos manipular as duas dentro do `for` loop.

> ***Nota:*** Assim como vimos anteriormente, o nome das variáveis passadas no `for` loop pode ser qualquer coisa. Eu escolhi "key" e "value".

## Métodos mais comuns

### get()

O método `get()` pode ter um ou dois argumentos. Quando tentamos acessar uma palavra-chave que não existe dentro de um `dictionary`, python levantará um erro. O método `get()` retornará o valor se a palavra-chave existir e, caso não exista, retornará o valor que passarmos como argumento.

```
items = {"sword": 3, "shield": 1, "dagger": 2, "bow": 1}

print(items.get("bow", 0))
```

Neste exemplo, estamos dizendo para python imprimir o valor da palavra-chave "bow", caso não exista no `dictionary`, diga que o valor é 0. A palavra-chave existe no `dictionary`, então python imprime seu valor.

```
1

Process finished with exit code 0
```

```
items = {"sword": 3, "shield": 1, "dagger": 2}

print(items.get("bow", 0))
```

Já neste exemplo, a palavra-chave "bow" não existe no `dictionary`, então python imprimirá o valor padrão (default).

```
0

Process finished with exit code 0
```

> ***Nota:*** Caso você só passe um argumento, imprimirá o valor se a palavra-chave existir no `dictionary` e, caso não exista, imprimirá o valor `None`, que é um tipo de dado que significa "nenhum valor existente".

## copy()

Este método permite que nós criemos uma cópia de um `dictionary`.

```
items = {"sword": 3, "shield": 1, "dagger": 2}
new_items = items.copy()

print(f"Items: {items}")
print(f"New items: {new_items}")
```

```
Items: {'sword': 3, 'shield': 1, 'dagger': 2}
New items: {'sword': 3, 'shield': 1, 'dagger': 2}

Process finished with exit code 0
```

## clear()

`clear()` remove todos os elementos de um `dictionary`.

```
items = {"sword": 3, "shield": 1, "dagger": 2}

items.clear()
print(f"Items: {items}")
```

```
Items: {}

Process finished with exit code 0
```

## setdefault()

O método `setdefault()` faz com que o `dictionary` sempre tenha determinada palavra-chave e valor.

```
items = {"sword": 3, "shield": 1, "dagger": 2}

items.setdefault("heartstone", 1)
print(f"Items: {items}")
```

No exemplo acima, nós definimos que o `dictionary` deve ter ao menos 1 "heartstone" obrigatoriamente. Como não colocamos no dicionário, python colocará.

```
Items: {'sword': 3, 'shield': 1, 'dagger': 2, 'heartstone': 1}

Process finished with exit code 0
```

Caso já existisse uma palavra-chave "heartstone" no `dictionary` python não faria nada.

```python
items = {"sword": 3, "shield": 1, "dagger": 2, "heartstone": 3}

items.setdefault("heartstone", 1)
print(f"Items: {items}")
```

```
Items: {'sword': 3, 'shield': 1, 'dagger': 2, 'heartstone': 3}

Process finished with exit code 0
```

## pop()

`pop()` é um método que recebe uma palavra-chave como argumento e remove do `dictionary`.

```python
items = {"sword": 3, "shield": 1, "dagger": 2}

items.pop("sword")
print(f"Items: {items}")
```

```
Items: {'shield': 1, 'dagger': 2}

Process finished with exit code 0
```

## Embelezando dicionários

Assim como como dados do tipo `list`, nós podemos escrever dados do tipo `dictionary` de tal maneira que fiquem mais legíveis, como no exemplo abaixo:

```python
person = {
    "name": "Luigi",
    "age": 24,
    "location": "Mushroom Kingdom"
}

print(person)
```

```
{'name': 'Luigi', 'age': 24, 'location': 'Mushroom Kingdom'}

Process finished with exit code 0
```

> ***Importante:*** Não esqueça de colocar virgula entre os elementos!

## Exercícios 8

Utilize o dicionário abaixo para os exercícios 1 e 2:

```
people = {
    'James': 30,
    'Mary': 23,
    'Robert': 83,
    'Patricia': 42,
    'John': 19,
    'Jennifer': 27,
    'Michael': 36,
    'Linda': 65,
    'David': 76
}
```

1. Encontre a pessoa mais velha e imprima na tela seu nome e idade.
2. Encontre a média de idade das pessoas e imprima-a na tela com 2 casas decimais.
3. Você foi contratado para criar um programa para entrevistar 10 pessoas sobre o que preferem, pizza ou sushi. Crie um programa para esta pesquisa que ao final imprima na tela o que a maioria prefere e quantos votos o ganhador recebeu.

---

# Tuple

Também conhecidas como "tupla" no português, `tuple` são tipos de dados iteráveis, assim como `list` e `dictionary`. Os elementos de um `tuple` são ordenados, então podemos acessá-los com índices; porém, um `tuple` é imutável. Uma vez criado, ele não pode ser modificado.

Um `tuple` é criado colocando os dados entre parênteses.

```
numbers = (10, 20, 30)

print(numbers)
```

```
(10, 20, 30)

Process finished with exit code 0
```

> ***Importante:*** Uma `list` ou um `dictionary` dentro de um `tuple` ainda pode ser modificados, porém, a estrutura do `tuple` não pode.

## Acessando elementos

Como `tuple` é ordenado, seus elementos podem ser acessados através de índices, assim como `list`.

```
numbers = (10, 20, 30)

print(numbers[1])
```

```
20

Process finished with exit code 0
```

> ***Importante:*** Os índices começam sempre em 0.

## Tuple slices

Também podemos acessar os elementos de `tuple` com slices.

```
numbers = (10, 20, 30, 40, 50)

print(numbers[:3])
```

```
(10, 20, 30)

Process finished with exit code 0
```

```
numbers = ("Banana", "Apple", "Grape")

print(numbers[::-1])
```

```
('Grape', 'Apple', 'Banana')

Process finished with exit code 0
```

## Métodos

### count()

Conta a quantidade de vezes que um elemento aparece no `tuple`.

```
numbers = (10, 20, 30, 30, 40, 20, 30, 10)

print(numbers.count(30))
```

```
3

Process finished with exit code 0
```

### index()

Mostra o índice do dado passado.

```
names = ("George", "Geoff", "Gob")

print(names.index("Geoff"))
```

```
1

Process finished with exit code 0
```

## Unpacking tuple

Unpacking tuple significa colocar cada elemento da `tuple` dentro de uma variável. Python nos permite fazer isso de forma bem elegante.

```
numbers = (10, 20, 30)
a, b, c = numbers

print(a)
print(b)
print(c)
```

```
10
20
30

Process finished with exit code 0
```

## Trocando variáveis

Podemos também trocar o valor de variáveis com `tuple`. Normalmente, se quiséssemos fazer isso, faríamos da forma abaixo:

```
a = 10
b = 20

print(f"a: {a}, b: {b}")

c = a
a = b
```

```
b = c

print(f"a: {a}, b: {b}")
```

```
a: 10, b: 20
a: 20, b: 10

Process finished with exit code 0
```

Temos duas variáveis, `a` de valor 10 e `b` de valor 20, e queremos trocar seus valores para que `a` seja 20 e `b` seja 10. Para isto, criamos uma variável temporária `c`, depois colocamos o valor de `a` em `c`, o de `b` em `a` e o de `c` em `b`. É como se tivéssemos dois copos, refrigerante de laranja no copo A e refrigerante de uva no copo B, e nós quiséssemos colocar o de uva no A e o de laranja no B. Para isso, nós usaríamos um terceiro copo vazio chamado de C. Colocaríamos o refrigerante de laranja no C, o de uva no copo A e o de laranja, que está no copo C, no copo B.

Com `tuple` podemos fazer esta troca de uma maneira muito mais simples.

```
a = 10
b = 20

print(f"a: {a}, b: {b}")

a, b = b, a

print(f"a: {a}, b: {b}")
```

```
a: 10, b: 20
a: 20, b: 10

Process finished with exit code 0
```

## Concatenando tuple

Apesar de não podermos modificar tuples, podemos concatená-las para criar novas tuples.

```
numbers = (1, 2, 3)
numbers_total = numbers + (4, 5, 6)

print(numbers_total)
```

```
(1, 2, 3, 4, 5, 6)

Process finished with exit code 0
```

## Exercícios 9

Utilize a `tuple` abaixo para resolver os exercícios 1, 2 e 3:

```
numbers = (7, 42, 93, 58, 12, 24, 30)
```

1. Imprima os três últimos números.
2. imprima a média de todos os elementos
3. imprima o tuple ao contrário
4. Você está trabalhando com um time para criar um jogo 2D. Nestes jogos, a posição dos personagens é definida pelas coordenadas (x, y). Você ficou encarregado de criar um novo poder para o personagem do jogador que troca sua posição com seu inimigo. Crie o código para esta mecânica e imprima na tela as posições antigas e atuais.

---

# Set

`set` é um tipo de dado iterável que remove elementos duplicados. Seus elementos não são organizados, ou seja, não podemos acessá-los através de índices (é comum que os elementos mudem de ordem cada vez que são impressos).

```
numbers = {10, 20, 10, 30, 10, 40}

print(numbers)
```

```
{40, 10, 20, 30}

Process finished with exit code 0
```

Como podemos ver, todos os números repetidos foram removidos e a ordem em que foram impressos está diferente da ordem em que o `set` foi escrito.

Os dados do tipo `set` só podem conter dados imutáveis, ou seja, dados que não podem ser mudados. Se você tentar colocar um dado do tipo `list` ou `dict` dentro de um `set`, python levantará um erro.

> ***Importante:*** Cuidado para não confundir `set` com `dictionary`. Ambos usam chaves mas `dictionary` tem pares de palavras-chave e valores.

## Acessando elementos

Já que os elementos de um `set` não estão em ordem, não podemos acessá-los com índices. Por isso, a única forma de acessá-los é através de um loop.

```python
vowels = {"a", "e", "i", "o", "u"}

for vowel in vowels:
    print(vowel)
```

```
i
a
u
e
o

Process finished with exit code 0
```

# Modificando um set

## adicionando elementos

### add()

Através do método `add()` nós podemos adicionar um elemento ao `set`.

```python
letters = {"a", "b", "c"}

letters.add("d")
print(letters)
```

```
{'c', 'b', 'd', 'a'}

Process finished with exit code 0
```

### update()

O método `update()` recebe um iterável como argumento e, assim, nos permite adicionar vários elementos de uma vez.

```python
languages = {"python", "gdscript", "kotlin"}
languages_to_learn = ["java", "kotlin"]

languages.update(languages_to_learn)
print(languages)
```

```
{'python', 'java', 'gdscript', 'kotlin'}

Process finished with exit code 0
```

# Removendo elementos

### remove()

O método `remove()` remove um elemento do `set`. Se o elemento não estiver no `set`, ele levanta um erro.

```
languages = {"python", "gdscript", "kotlin"}

languages.discard("kotlin")
print(languages)
```

```
{'python', 'gdscript'}

Process finished with exit code 0
```

> ***Nota:*** Nós aprenderemos a lidar com erros posteriormente.

### discard()

`discard()` também remove um elemento do `set`, mas caso o elemento não exista, ele não levanta um erro.

```
languages = {"python", "gdscript", "kotlin"}

languages.discard("kotlin")
print(languages)
```

```
{'python', 'gdscript'}

Process finished with exit code 0
```

## pop()

O método `pop()` com `set` tem 2 usos. Ele pode:

1. remover um elemento **aleatório** do set
2. remover um elemento **aleatório** e atribuí-lo a uma variável

Primeiro uso:

```
languages = {"python", "gdscript", "kotlin"}

languages.pop()
print(languages)
```

```
{'kotlin', 'gdscript'}

Process finished with exit code 0
```

Segundo uso:

```
languages = {"python", "gdscript", "kotlin"}

random_language = languages.pop()
print(languages)
print(random_language)
```

```
{'kotlin', 'gdscript'}
python

Process finished with exit code 0
```

## Operações com set

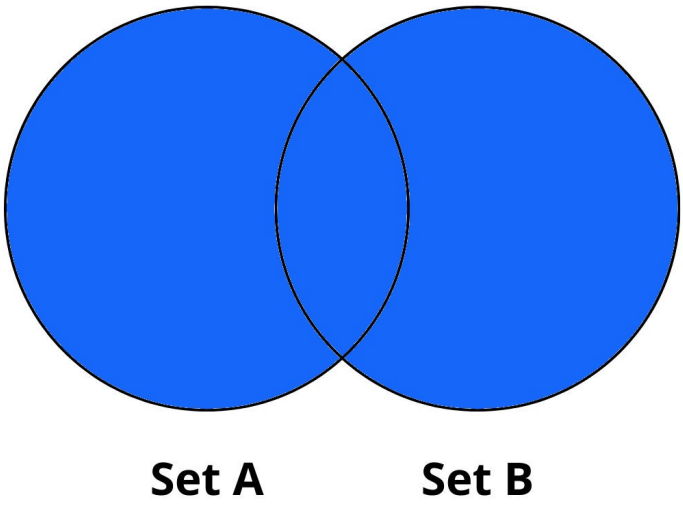
Assim como na matemática, operações de conjuntos como união, intersecção e diferença, podem ser feitas com o set. Eu sei que a gente se assusta quando lê que tem matemática envolvida. Mas eu vou explicar em detalhes e desenhar pra você ver que não é nenhum bicho de sete cabeças.

Existem operadores e métodos para cara uma dessas operações. Eu mostrarei das duas formas.

> ***Nota:*** Nos exemplos eu utilizei dois sets apenas, mas é possível fazer operações com três ou mais sets também.

### União

Quando fazemos a união de sets, nós pegamos todos os elementos contidos em todos os sets.




Em python, nós podemos utilizar o método `union()` ou o operador `|`.

```
numbers = {1, 2, 3, 4}
more_numbers = {3, 4, 5, 6}

print(f"Method: {numbers.union(more_numbers)}")
print(f"Operator: {numbers | more_numbers}")
```

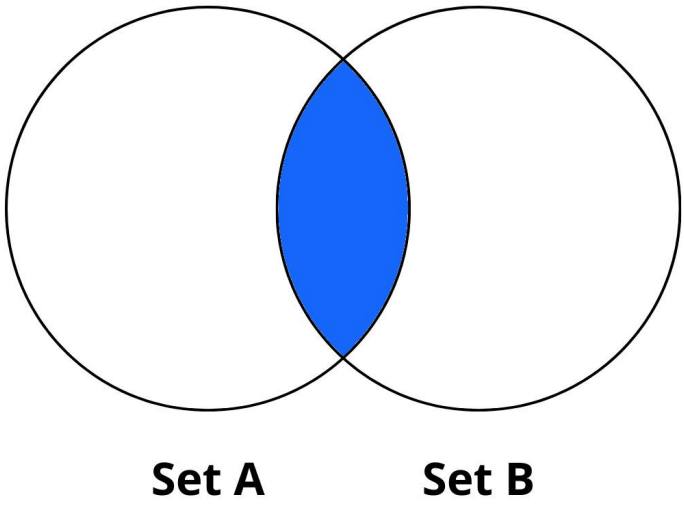
```
Method: {1, 2, 3, 4, 5, 6}
Operator: {1, 2, 3, 4, 5, 6}

Process finished with exit code 0
```

Como podemos ver, foi criado um novo set contendo todos os elementos contidos nos sets `numbers` e `more_numbers`, sem repeti-los, pois dados do tipo `set` não aceitam repetições de valores.

## Interseção

Quando fazemos a interseção de sets, nós pegamos apenas os elementos únicos que existem em todos eles.




Em python, nós podemos utilizar o método `intersection()` ou o operador `&`.

```
numbers = {1, 2, 3, 4}
more_numbers = {3, 4, 5, 6}

print(f"Method: {numbers.intersection(more_numbers)}")
print(f"Operator: {numbers & more_numbers}")
```

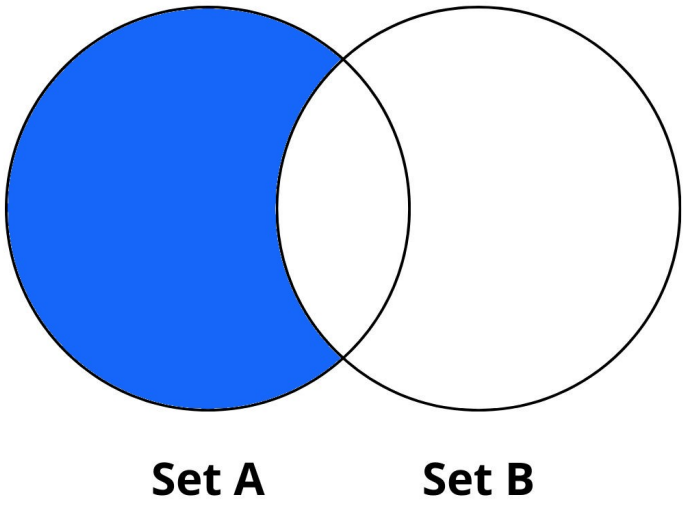
```
Method: {3, 4}
Operator: {3, 4}

Process finished with exit code 0
```

Aqui foi criado um novo set contendo apenas os elementos que aparecem tanto no set `numbers` quanto no set `more_numbers`. Novamente: sem repeti-los pois dados do tipo `set` não aceitam repetição de valores.

## Diferença

Quando fazemos a diferença entre sets, nós pegamos apenas os elementos que existem em um set mas não existem nos outros.




Em python, nós podemos utilizar o método `difference()` ou o operador `-`.

```
numbers = {1, 2, 3, 4}
more_numbers = {3, 4, 5, 6}

print(f"Method: {numbers.difference(more_numbers)}")
print(f"Operator: {numbers - more_numbers}")
```

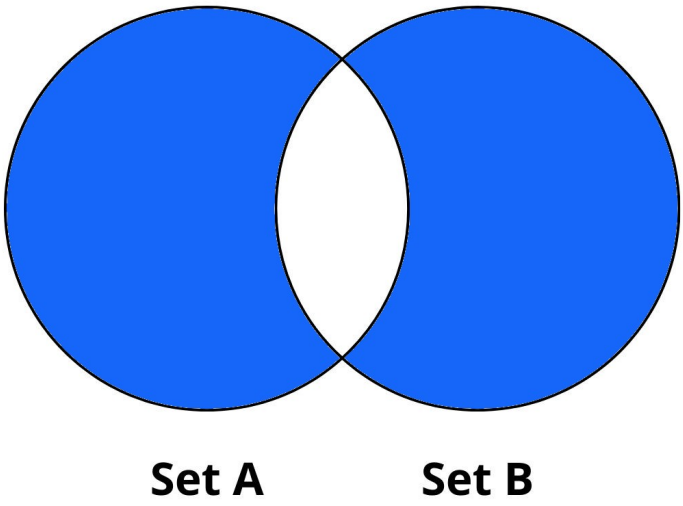
```
Method: {1, 2}
Operator: {1, 2}

Process finished with exit code 0
```

Um novo set foi criado contendo apenas os números que pertencem ao set `numbers` que não pertencem a outros sets.

## Diferença simétrica

Quando fazemos a diferença simétrica entre sets, nós pegamos todos os elementos que são únicos de cada set. Qualquer elemento que apareça em mais de um set é ignorado.




Em python, nós podemos utilizar o método `symmetric_difference()` ou o operador `^`.

```
numbers = {1, 2, 3, 4}
more_numbers = {3, 4, 5, 6}

print(f"Method: {numbers.symmetric_difference(more_numbers)}")
print(f"Operator: {numbers ^ more_numbers}")
```

```
Method: {1, 2, 5, 6}
Operator: {1, 2, 5, 6}

Process finished with exit code 0
```

Neste caso, criamos um novo set com todos os elementos únicos que pertencem ao set `numbers` e ao set `more_numbers`. Qualquer elemento que pertença aos dois sets não foi incluído.

---

# Frozenset

Dados do tipo `frozenset` são sets imutáveis, ou seja, o original não pode ser modificado.

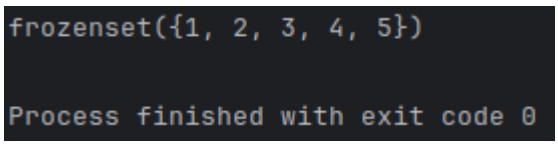
Nós podemos criá-los através da função `frozenset()`.

Se não passarmos argumentos, a função cria um dado do tipo `frozenset` vazio. Caso passemos um dado iterável como argumento para a função, ela faz type casting e transforma o dado em um frozenset.

```
numbers = [1, 2, 3, 4, 5]
numbers = frozenset(numbers)

print(numbers)
```

Neste exemplo, passamos uma lista como argumento, fazemos o type casting para frozenset e este é o resultado:




---

## Mais type casting

Nós sabemos que podemos alterar o tipo de um dado através do que chamamos de type casting. E também podemos fazer isso com dados do tipo collection.

De maneira geral, eles funcionam da mesma forma dos que vimos anteriormente. A única diferença é que eles têm que receber um dado iterável (que nós podemos acessar usando um loop) como argumento, como uma string ou uma collection.

```
list(dado)
```

Converte o dado para o tipo `list`.

```
tuple(dado)
```

Converte o dado para o tipo `tuple`.

```
set(dado)
```

Converte o dado para o tipo `set`.

E nós podemos visualizar isso com a função `type()` que nos diz o tipo do dado passado.

```
numbers = [1, 2, 3]
print(numbers)
print(type(numbers), end="\n\n")

numbers = tuple(numbers)
print(numbers)
print(type(numbers), end="\n\n")
```

```
numbers = set(numbers)
print(numbers)
print(type(numbers))
```

```
[1, 2, 3]
<class 'list'>

(1, 2, 3)
<class 'tuple'>

{1, 2, 3}
<class 'set'>

Process finished with exit code 0
```

> ***Nota:*** Nós aprenderemos posteriormente a criar nossas próprias classes.

Type casting para o tipo de dado `dictionary` é um pouquinho diferente. Ele recebe uma lista, um set ou um tuple de tuples de tamanho 2 e então os converte para um dicionário.

Como sabemos, dicionários armazenam elementos em pares de palavras-chave e valores. Sendo assim, para transformarmos um dado para o tipo `dict`, precisamos passar como argumento dados do tipo `tuple` de tamanho 2.

```
name_and_age = [("Whiskers", 3), ("Bubbles", 6)]

print(dict(name_and_age))
```

Neste exemplo, temos uma lista de dados do tipo `tuple` (mas poderia ser um set de tuples ou um tuple de tuples), e estamos convertendo-o para o tipo `dict`.

```
{'Whiskers': 3, 'Bubbles': 6}

Process finished with exit code 0
```

---

# Exercício 10

Dadas as listas abaixo:

```
names1 = ['Rachel', 'Augusto', "Giorgio"]
names2 = ['Pedro', 'Conan', 'Rachel',]
names3 = ['Conan', 'Giorgio', 'Rodrigo']
```

1. Imprima na tela os elementos que aparecem em uma lista mas não nas outras.

2. Imprima na tela todos os elementos das três listas sem duplicações.
3. Crie um programa para checar se a lista abaixo tem números repetidos e imprima na tela "Sim, tem números repetidos." ou "Não, não tem números repetidos".

```
numbers = [12, 7, 5, 46, 32, 26, 1, 90, 88, 7, 12, 26, 1]
```

---

# Função zip()

A função `zip()` recebe 2 ou mais dados iteráveis como argumento e os transforma em um objeto do tipo `zip`, que contém um conjunto de tuples e é iterável. Eu sei que soa muito complicado mas com exemplos tudo ficará mais claro.

```
positions = [1, 2, 3]
months = ["January", "February", "March"]
my_zip = zip(positions, months)

print(my_zip)
```

Neste exemplo temos duas listas e estamos utilizando a função `zip()` para uni-los. Se tentarmos imprimir diretamente, python imprimirá que é um objeto do tipo `zip` e onde ele se encontra na memória.

```
<zip object at 0x7f9a1e5f3780>

Process finished with exit code 0
```

Objetos do tipo `zip` são iteráveis, então podemos utilizar um loop para acessar seus elementos.

```
positions = [1, 2, 3]
months = ["January", "February", "March"]
my_zip = zip(positions, months)

for element in my_zip:
    print(element)
```

```
(1, 'January')
(2, 'February')
(3, 'March')

Process finished with exit code 0
```

Ao acessar seus elementos, podemos ver que o objeto do tipo `zip` pegou os elementos de cada lista e criou tuples. Primeiro elemento com primeiro elemento, segundo elemento com segundo elemento, e assim por diante.

Então o object do tipo `zip` contém um ou mais elementos do tipo `tuple`. E nós podemos usar type casting para transformar um objeto `zip` em uma lista, um set, um tuple, ou até um dict.

No exemplo abaixo, o transformamos em uma lista de tuples:

```
positions = [1, 2, 3]
months = ["January", "February", "March"]
zip_list = list(zip(positions, months))

print(zip_list)
```

```
[(1, 'January'), (2, 'February'), (3, 'March')]

Process finished with exit code 0
```

Já neste exemplo, o transformamos em um dict:

```
positions = [1, 2, 3]
months = ["January", "February", "March"]
zip_dict = dict(zip(positions, months))

print(zip_dict)
```

```
{1: 'January', 2: 'February', 3: 'March'}

Process finished with exit code 0
```

> ***Importante:*** Só é possível transformar um objeto do tipo `zip` em um objeto do tipo `dict` quando temos no zip apenas tuples de tamanho 2, pois dicts só aceitam pares de elementos.

## Iteráveis de tamanhos diferentes

Quando nós passamos iteráveis de tamanhos diferentes como argumentos para a função `zip()`, o objeto zip terá o tamanho igual ao do menor argumento. Deixa eu explicar isso com um exemplo:

```
positions = [1, 2, 3, 4, 5, 6, 7]
months = ["January", "February", "March"]
zip_list = list(zip(positions, months))
```

```
print(zip_list)
```

Aqui estamos passando dois argumentos para a função `zip()`: a lista `positions` que tem um tamanho 7, ou seja, ela tem 7 elementos; e a lista `months` que tem apenas 3. Neste caso, o objeto zip criado terá um tamanho igual ao tamanho do menor argumento passado, que é a lista `month`. O tamanho do objeto zip será 3.

```
[(1, 'January'), (2, 'February'), (3, 'March')]

Process finished with exit code 0
```

---

# Otimização

Eu sei que nós vimos muitos tipos de dados e seus diferentes usos. Mas como sei qual dado usar ou quando usar? Não existe resposta certa pra esta pegunta. Existem diversas soluções para um mesmo problema. Você pode chegar ao mesmo resultado utilizando diferentes tipos de dados e cabe a você escolher o que achar melhor.

Com isso em mente, é importante falarmos da otimização. Dizemos que um código é otimizado quando ele é eficiente. Ele usa a quantidade mínima de memória necessária para funcionar ou chega a um resultado de maneira rápida. E escolher o tipo de dado necessário para seu algoritmo faz diferença. Por exemplo, se você sabe que um conjunto de dados não será e nem poderá ser alterado durante a execução de um programa, utilize um dado do tipo `tuple` em vez de uma lista, pois o tuple tem estas características especificamente - ele é otimizado para isto.

Quando nós criamos mais de um algoritmo para resolver um mesmo problema, nós podemos avaliar qual é o mais eficiente através do que chamamos de Notação Big O, em inglês, Big O Notation. Imagine que você criou dois algoritmos que modificam dados do tipo `string`. Quando trabalhamos com strings curtas, ambos os algoritmos são bem rápidos e resolvem o problema em milésimos de segundo. Mas e quando trabalhamos com strings com mil caracteres, ou cem mil caracteres, qual deles seria mais rápido? A Notação Big O serve para responder essa pergunta, avaliando-os e classificando-os.

## O mais importante é fazer funcionar

Eu não acredito que seja relevante estudarmos otimização por enquanto, então não quero que você se preocupe com isso. Estou comentando apenas para que você saiba que isto existe. Nós aprenderemos mais sobre otimização num capítulo futuro.

Eu fiz questão de colocar no título para que isso fique bem claro: A otimização vem depois, caso haja necessidade. **O primeiro e mais importante passo é fazer o programa**

**funcionar.** Se você quiser voltar e otimizar seu código posteriormente, ou caso haja necessidade, okay. Excelente. Mas caso não veja necessidade, siga em frente.

A intenção aqui é de que você conheça estes termos e tenha uma noção do que eles significam. É comum ouvir relatos de excelentes programadores que aprenderam sozinhos a programar e se sentiram completamente perdidos ao entrar na industria e ouvir estes termos técnicos dos quais nunca haviam ouvido falar.

Então não se preocupe com isso por agora. Siga estudando pois primeiro é necessário aprender como fazer, depois nós podemos focar como melhorar o que foi feito.

---

# Bonus: Uma análise mais aprofundada sobre variáveis

Neste ponto você já está bem confortável com variáveis, com o que são e como usá-las. Então agora eu gostaria de explicá-las um pouco mais a fundo.

Na maioria dos cursos que eu fiz e livros que li, variáveis eram descritas como sendo "diferentes das variáveis que existem na matemática". E se você já fez outros cursos ou já viu variáveis sendo explicadas desta forma, provavelmente ficou confuso como eu fiquei. Porque se pararmos pra pensar, as variáveis que usamos até agora são idênticas às variáveis matemáticas. Na matemática, uma variável `x` tem um valor que pode ser um número inteiro ou um número decimal, por exemplo. Então por que é explicado dessa forma? Esta é uma pergunta que, na maioria dos cursos, fica sem resposta; então nós a responderemos aqui.

Nós sabemos que as variáveis são armazenadas em endereços na memória. Quando criamos uma variável e a damos um nome, este nome é simplesmente uma referência, ou apelido, para o endereço na memória onde este dado está localizado.

Imagine que você está voltando para casa e encontra um amigo, ele pergunta "Aonde você está indo?" e você responde "Estou indo para a minha casa." Quando você diz "minha casa", seu amigo entende que você está se referindo ao endereço, que é Rua Muçarela, número 70, Bairro Queijo. Então ou você pode falar o seu endereço, ou pode usar uma referência mais descritiva. E é assim que funciona com as variáveis.

Quando criamos uma variável chamada `hello` e chamamos a função `print()` para imprimi-la na tela, python entende que quando você diz `hello`, você está se referindo ao dado que se encontra naquele endereço na memória.

Para que isto fique um pouco mais claro, vamos analisar a função `zip()` que vimos anteriormente.

```
positions = [1, 2, 3]
months = ["January", "February", "March"]
```

```
my_zip = zip(positions, months)

print(my_zip)
```

Quando tentamos imprimir na tela o valor da variável `my_zip`, python entende que o nome da variável é simplesmente uma referência a um endereço na memória.

```
<zip object at 0x7f9a1e5f3780>

Process finished with exit code 0
```

Quando a imprimimos, ele nos diz que dentro do endereço `0x7f9a1e5f3780` na memória há um objeto zip. Python entende que a variável de nome `my_zip` é uma referência ao endereço `0x7f9a1e5f3780` na memória.

E é assim que a variável computacional se difere da variável matemática. Ela não é simplesmente um nome com um valor, ela é um nome que aponta para um endereço na memória, onde um valor se encontra.

---