Chapter 3

Chapter 3: Collections data type

There are more advanced data types called `collections` that work like containers that can hold one or more data of various types. In this chapter we'll talk about python's **built-in collections**.

Thinking like a programmer

Before we begin with the contents of this chapter, I would like to talk about how to think like a programmer. And what does this mean, exactly?

Well, with each exercise we're solving more complex problem. We started by solving problems that only required variables and data types to be solved; then we solved problems that combined these with flow control. We are learning isolated tools and techniques that can be combined to solve bigger problems.

And I refer to this when I say "thinking like a programmer". I want you to learn to look at problem through different lenses. I want you to divide complex problems into smaller problems that can be solved with the tools you have at your disposal.

When we find a big and complex problem, normally we don't even know where we begin to solve it. What we must do is to divide it into smaller problems that have a simpler answer, and solve them one by one; piece by piece.

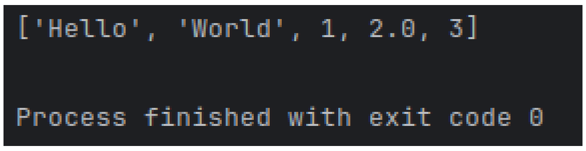
A useful technique to achieve this goal is `TODO`. Imagine you were hired to make a program. The first step is to define its functionalities (dividing a bigger problem into smaller, less complex ones) and, for each functionality, we define the `TODO`, in other words, we define what each functionality will do, how many variables it will need, which data types, how the flow control will be and so on.

I hope that, from now on, you'll look at problems this way. Thinking about which steps you'll have to take to solve them. And look to what we're learning as new tools and new ways of doing the same. There are multiple ways to solve the same problem, and each will use the tools in a different way.

List

A list is a data type that contains one or more ordered elements. Within a list we can have strings, numbers, other lists, among others.

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



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

Process finished with exit code 0
```

In this example we have a list with 5 elements of various types. When we print it, python prints the entire list, including the square brackets.

> ***Note:*** Python recognizes a list because of the square brackets.

Before we see practical uses of a list, we need to learn how to manipulate it.

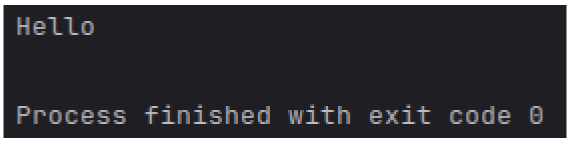
Accessing a list

Accessing the elements of a list

As we've said, lists are ordered, which means that each element has a fixed position within them. Knowing this, we can access them by their index.

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

Here we're telling python to print the element that is found in the position or index 0 within the list.



```
Hello

Process finished with exit code 0
```

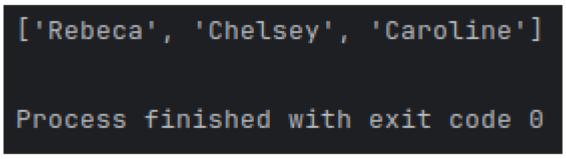
> ***Important:*** The index always begins at 0. So, in a list of 5 elements, we have indexes 0, 1, 2, 3, 4. Indexes are always `int`.

Accessing a list within a list

A list can contain elements of various types, including `lists` , `dictionaries` and other types we'll see later.

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

When we try to access the index 0 of the list `names_and_numbers`, python will print the list of names.

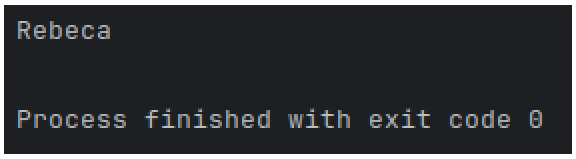


```
['Rebeca', 'Chelsey', 'Caroline']

Process finished with exit code 0
```

So how do I access the string `'Rebeca'`?

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

First we access the element of index 0 of the list `names_and_numbers`, which is the list of names, then we access the element of index 0 of it.



```
Rebeca

Process finished with exit code 0
```

Accessing the last element

Python reads the elements of a list from left to right.

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

When we use negative indexes, python reads them from right to left.

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

So the element of index `-1` is the last element of the list.

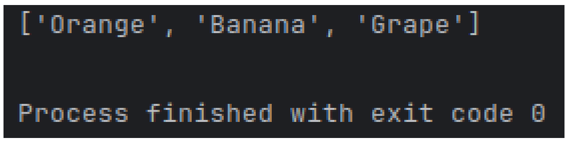
Modifying a list

Now that we've learned how to access each element in a list, we'll see how to modify them.

Replacing elements

To replace a element in a list, we first must access the element and then assign a value, as we've done with variables previously.

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



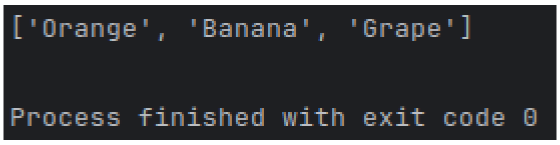
```
['Orange', 'Banana', 'Grape']

Process finished with exit code 0
```

Adding elements to the end

To add elements to the end of a list we use the method `list_name.append()`.

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

In this example, we're creating an empty list and calling it `fruits`, we then use the method `append()` to add elements to its end, resulting in:



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

Process finished with exit code 0
```

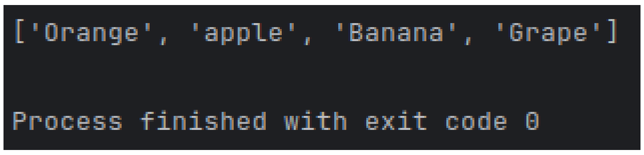
> ***Note:*** We'll study methods in a deeper manner in the future. For now, just learn how to use them.

Adding elements to specific positions

The method we utilize to add an element to a list in a specific position is `insert()`. When utilizing it, we need to pass the index and the data we'd like to add.

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

We are telling python to add a `string "apple"` into index 1 of the `fruits` list. It is important to note that python will not replace "Banana" with "apple". It will push "Banana" forward and add "apple" where "Banana" was.



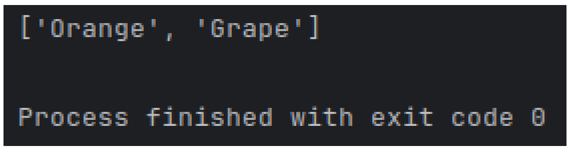
```
['Orange', 'apple', 'Banana', 'Grape']

Process finished with exit code 0
```

Removing elements by index

To remove elements of a list we utilize the `del` command.

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

```python
print(fruits)
```



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

Process finished with exit code 0
```

With the `del` command we can simply delete an element of a list.

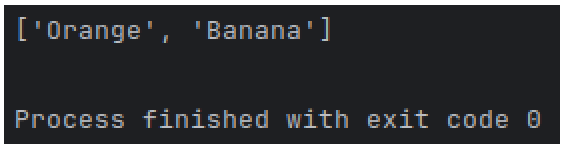
Removing and assigning an element from a list with pop()

The method `pop()` has 3 uses. It can:

1. remove the last element from a list
2. remove a specific element from a list
3. remove the element and assign it to a variable

When we only use the method `pop()`, it will remove the last element from a list:

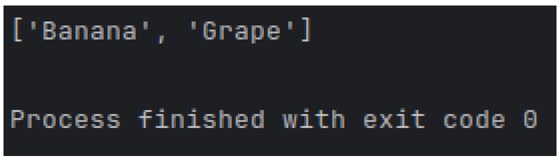
```python
fruits = ['Orange', 'Banana', 'Grape']
fruits.pop()
print(fruits)
```



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

Process finished with exit code 0
```

When we pass the index of an element as an argument, it will remove the specific element. In the following examples, we'll remove the element in index 0.

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

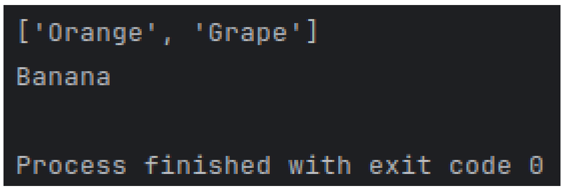


```
['Banana', 'Grape']

Process finished with exit code 0
```

When we want to remove the item from a list and assign it to a variable, we also use `pop()`

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

In this example, we are not only accessing the element in the list and assigning it to the variable. We're removing `'Banana'` and assigning it to the variable `my_favorite_fruit`

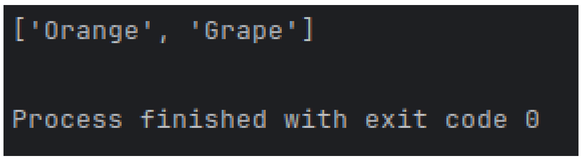


```
['Orange', 'Grape']
Banana

Process finished with exit code 0
```

Removing an element by value

So far we've only seen ways of removing an element by index. But what about when we know the value but not where it is? In these cases we utilize the `remove()` method.

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

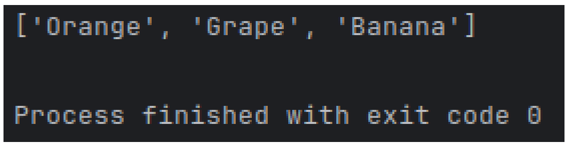


```
['Orange', 'Grape']

Process finished with exit code 0
```

Here we are telling python to find the word 'Banana' in the list and remove it.

> ***Important:*** The `remove()` method removes only the first instance of the value!

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

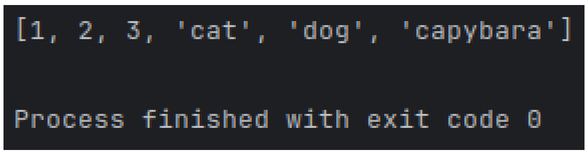


```
['Orange', 'Grape', 'Banana']

Process finished with exit code 0
```

Combining lists

Lists can be concatenated, or combined, with the `+` operator:

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

numbers_and_animals = numbers + animals

print(numbers_and_animals)
```

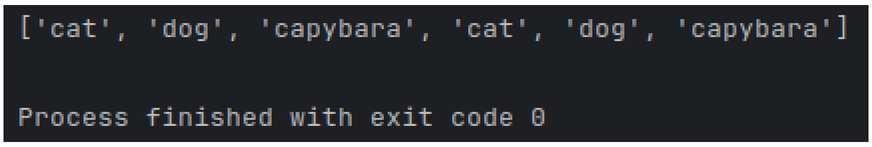


```
[1, 2, 3, 'cat', 'dog', 'capybara']

Process finished with exit code 0
```

Repeating the values

When we multiple a list by an `int`, we repeat its elements inside of it, as we can see below:

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



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

Process finished with exit code 0
```

Loops and lists

Now that we know how to manipulate a list, we can talk about its true power. We can utilize a loop to iterate through all the elements of a list and execute a block of code for each of the elements. Wait, I know this is starting to sound complicated again. So let's go step by step.

Imagine that we are having a party and we have a list of guest names. We could make a program that will say "Hello" to all the guests. With what we've learned so far, making something like this would be a lot of work.

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

Since we're dealing with a huge number of values, we can utilize loops and lists together to tackle this problem.

With what we've seen so far, we can access the elements of a list by their indexes, so we can use a `for` loop to do it.

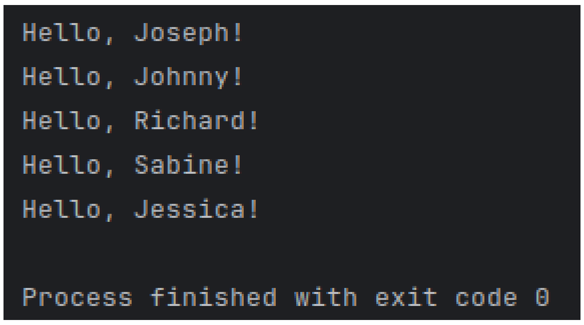
```
names = ["Joseph", 'Johnny', 'Richard', 'Sabine', 'Jessica']

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

Here I'm creating a `for` loop that starts at 0 and goes up to the length of the `names` list. The `names` list has 5 elements, so the `range()` function will go from 0 to 5, non-inclusive. In the first iteration, python will print `names[0]`, which is the value `"Joseph"`, in the second iteration, it will print `names[1]`, which is `"Johnny"` and so on.

This is how you'd normally do it in other languages. Python, however, offers us a much more elegant solution to this problem.

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

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

In this code, the `for` loop will begin and, in its first iteration, it will get the first element in `names` and assign it to the variable `name`. In other words, in the first iteration `name = "joseph"`. Python will print the message on screen, and the loop will start again. In the second iteration, `name = "Johnny"`, and so on until it reaches the end of the list, resulting in:



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

Process finished with exit code 0
```

In case it still isn't clear, try reading the code like this: "for each name in `names`, do:".

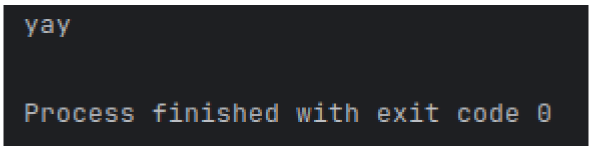
As you can see, with only a few lines of code we can modify multiple values at once by combining lists and loops!

The `in` and `not in` operators

We use the `in` and `not in` operators to check if a value is in a list or not, respectively.

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

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

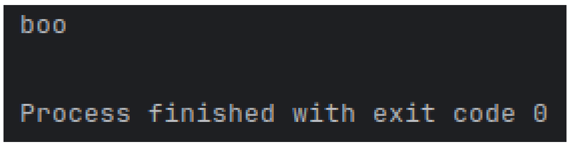
This code checks if the `string` "Johnny" is part of the list `names` and prints "yay" in case it is.



```
yay

Process finished with exit code 0
```

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

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

In this code, however, it checks if the `string` "Rebeca" is not a part of the list `names` and, in case it isn't, it prints "boo" on screen.



```
boo

Process finished with exit code 0
```

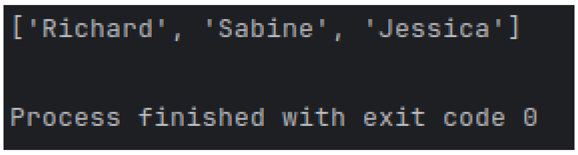
> ***Important:*** Uppercase and lowercase letters are seen differently by the computer. So `"Rebeca"` is not the same as `rebeca`.

List Slices

We already know how to access an element within a list, or how to access all the elements of a list. But what about when we only want to access some elements of a list? One option would be to create a `for` loop and use conditions to determine which elements will be selected. But python allows us to do it in a different way: using list slices.

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

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

In this block of code, we are telling python to print the elements of the `names` from index 2 to index 5, non-inclusive. So indexes 2, 3, and 4.



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

Process finished with exit code 0
```

List slices work the same way as the `range()` function we've learned previously.

```
list_name[beginning:end:step]
```

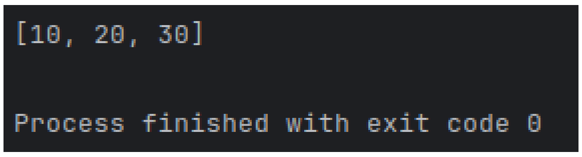
> ***Note:*** The standard value for `step` is 1, which means "one by one".

> ***Important:*** Do not forget that the slice goes from one number to the next, non-inclusively. If I say `[0:3]`, I'm saying from 0 up to but not including 3, in other words, 0, 1, 2.

When we don't put in a value for the beginning, we're saying "start from index 0".

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

print(my_lucky_numbers)
```

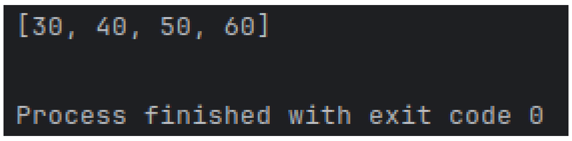
In this block of code, we have a variable of type `list` called `numbers`, and we're making a new variable called `my_lucky_numbers`, which is also a list, and we're assigning to it the elements from the beginning of the `numbers` list up to the element of index 3, non-inclusive.



```
[10, 20, 30]

Process finished with exit code 0
```

When we don't put in a value for end, we're saying "go until the end".

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

print(my_lucky_numbers)
```

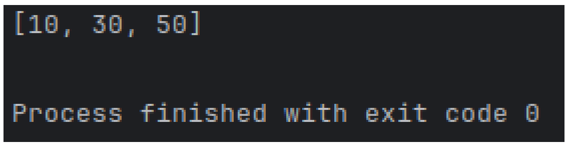


```
[30, 40, 50, 60]

Process finished with exit code 0
```

Can you tell me what the block of code of the next example does?

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

print(my_lucky_numbers)
```

To find out, we need to remember that list slices work with three values `[beginning:end:step]`. We didn't pass any value for the beginning, so we're starting from index 0; we also didn't pass any value for the end, and this means "go until the end"; and, lastly, we're telling python to go in twos. This code will then read the entire list and it will assign to `my_lucky_numbers` the elements in indexes 0, 2 and 4.

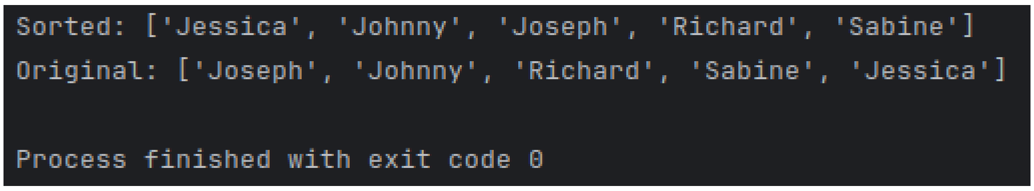


```
[10, 30, 50]

Process finished with exit code 0
```

sorted() function

The `sorted()` function organizes the list without altering the original.

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

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

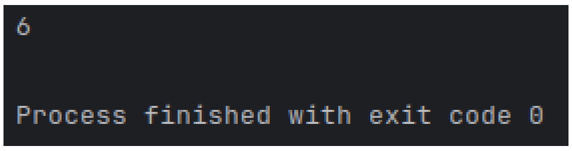


```
Sorted: ['Jessica', 'Johnny', 'Joseph', 'Richard', 'Sabine']
Original: ['Joseph', 'Johnny', 'Richard', 'Sabine', 'Jessica']

Process finished with exit code 0
```

len() function

The `len()` function counts how many elements there are in a list.

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



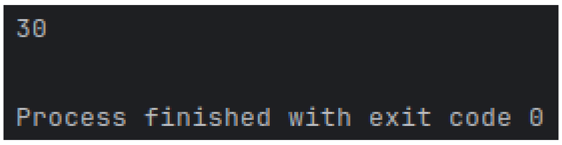
```
6

Process finished with exit code 0
```

> ***Important:*** This function can also be used with other data types, such as `dictionary`, `tuple`, `string`, among others.

max() function

The `max()` function returns the largest value in a list.

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

print(max(numbers))
```

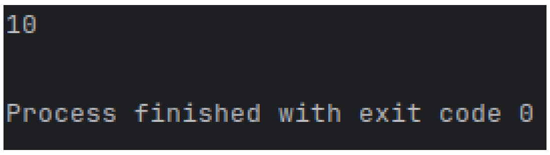


```
30

Process finished with exit code 0
```

min() function

The `min()` function returns the smallest value in a list.

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

print(min(numbers))
```



```
10

Process finished with exit code 0
```

Most common methods

Later we'll see in detail what methods are and how to create them. But, for now, we only need to know that we call them utilizing the "Dot Notation", which we have utilized before:

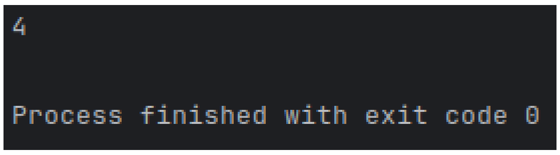
```
numbers = []
numbers.append(1)
```

As we can see, the dot notation is nothing more than utilizing a dot to call a method. We'll see other methods before we learn how to create them, and all of them will be called via dot notation.

index()

The `index()` method returns, in other words, has as a result, the index of an element from a list.

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

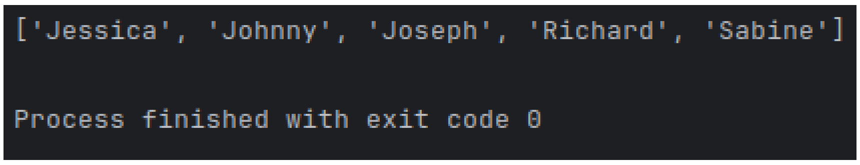


```
4

Process finished with exit code 0
```

sort()

The `sort()` method organizes the list, in ascending order, or in alphabetical order.

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

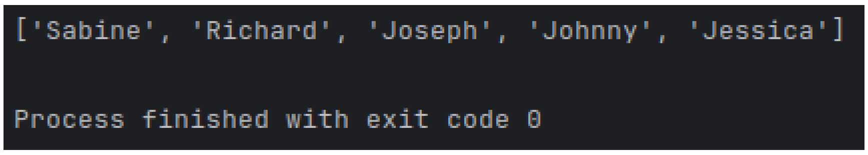


```
['Jessica', 'Johnny', 'Joseph', 'Richard', 'Sabine']

Process finished with exit code 0
```

We can also organize the list in reverse, or descending order with it:

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

```python
print(names)
```

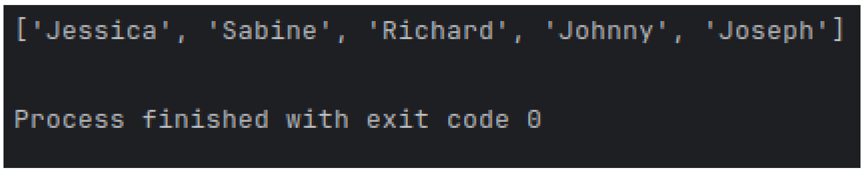


```
['Sabine', 'Richard', 'Joseph', 'Johnny', 'Jessica']

Process finished with exit code 0
```

reverse()

The `reverse()` method reverses the order of a list.

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



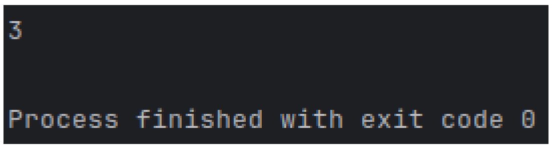
```
['Jessica', 'Sabine', 'Richard', 'Johnny', 'Joseph']

Process finished with exit code 0
```

count()

The `count()` method receives one argument and counts how many times it appears in the last.

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

print(counter)
```

In this example, we're passing the string "Jessica" as an argument. Python, then, will count how many times the string "Jessica" appears in the list and will assign this value to the variable `counter`. Resulting in:



```
3

Process finished with exit code 0
```

The element "Jessica" appears 3 times in the `names` list.

Beautifying lists

It can happen that our lists are too big, that they have too many elements, which make them hard to read. In these cases, we can write the list in a different way, to make it easier to read.

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

The list will continue to work perfectly.

> ***Important:*** Do not forget the comma between elements!

Exercise 5

1. Create a program that, given the list below, finds the largest and smallest numbers, then prints them.

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

2. **Bonus:** Do exercise 1 without using the `min()` and `max()` functions.
3. Create a program that will print the list below without any duplicated numbers and in ascending order:

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

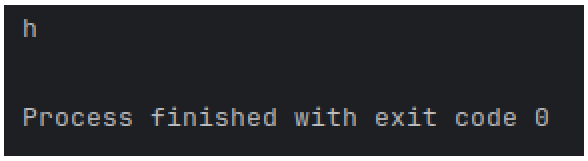
4. Still utilizing the previous list, create a program that will find the most common number in the list and print it alongside with how many times it appears on the list.
5. **Bonus:** Do exercise 3 without using the `count()` method.

Revisiting strings

We already know what `strings` are and how they work; however, now that we've learned more concepts, we can look at them from a different angle. `Strings` are like lists of alphanumeric characters. With this in mind, we can, for example, access each character through its index:

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

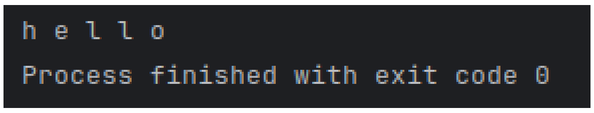
Here we're accessing the element of index 0 in the `string`.



```
h

Process finished with exit code 0
```

We can also access elements of a `string` using a `for` loop:

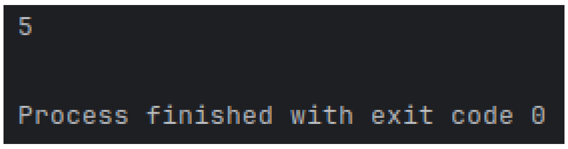
```python
word = "hello"

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



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

Or utilize the `len()` function to count how many elements are in the `string`:

```python
word = "hello"

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



```
5

Process finished with exit code 0
```

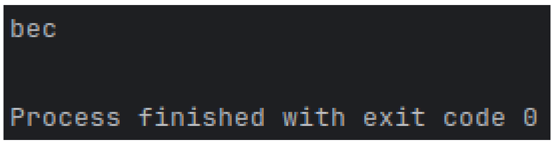
`string` is a different data type than a `list`; thus, it has different methods. In this module we'll see other ways to manipulate and methods related to the `string` data type.

String slices

Just like lists, we can also slice strings using their indexes. It works in the exact same way as **list slices**.

```python
name = "Rebeca"

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

With your knowledge of list slices, do you know what will be printed?



```
bec

Process finished with exit code 0
```

That's correct! It prints the characters from index 2 up to index 5, non-inclusive. So indexes 2, 3 and 4 of the string `name`.

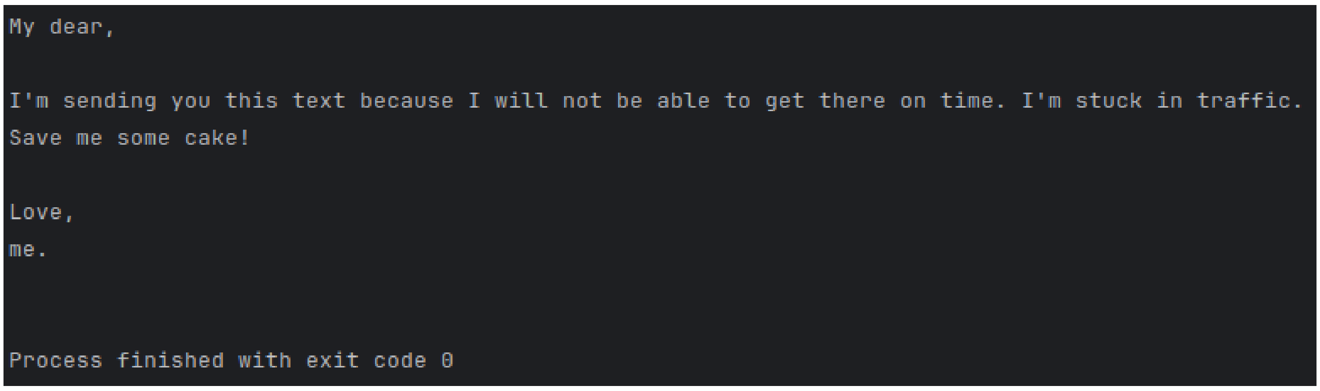
> ***Note:*** Remember that indexes start at 0!

Everything that we've learned about **list slices** is applicable to strings. And this is a very powerful and useful tool to manipulate them. So don't hesitate to go back and read it again!

Triple quotation marks

The `string` we've seen so far were all in the same line. We can utilize what we've learned to create strings with multiple lines, like, for example, with line breaks, concatenation, or even multiple `print()` functions. There is, however, another way: the quotation marks.

```
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)
```



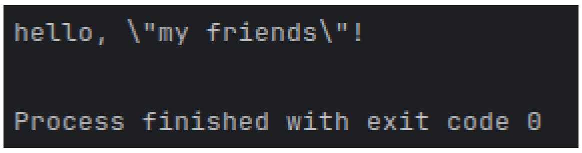
```
My dear,

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
```

> ***Note:*** There can be used single `'''` ou double `"""` quotes.

Raw strings

We've seen that we can ignore characters and add special ones with the backslash, also known as the *escape character*. But what if we want that whatever it is type by the use is kept in the `string`, including backslashes. This is where the raw string or `r string` come into play.

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



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

Process finished with exit code 0
```

In the `r string` python treats everything as a part of it, even if you pass special characters.

Most common methods

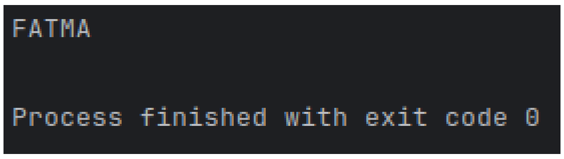
There are other methods and you can find them in the official documentation, books, on google or asking AIs. These are just the more commonly used ones.

upper()

The `upper()` method puts all the letters in the string in uppercase.

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

print(name)
```

Here we're creating a variable of type `string` with the value "Fatma", then we're assigning to the variable `name` the original value modified so that all the letters are in uppercase.

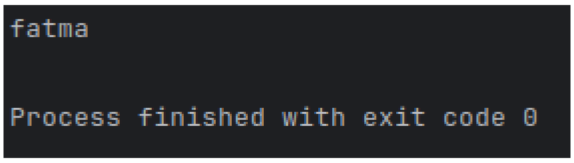


```
FATMA

Process finished with exit code 0
```

lower()

The `lower()` method puts all the letters in the string in lowercase.

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

print(name)
```

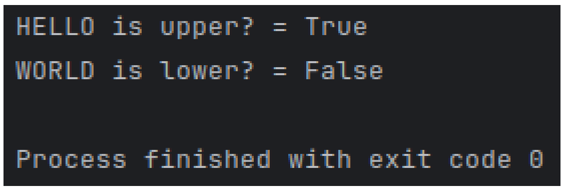


```
fatma

Process finished with exit code 0
```

isupper() e islower()

They check if the characters in the string `string` are all in uppercase or lowercase, respectively.

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

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

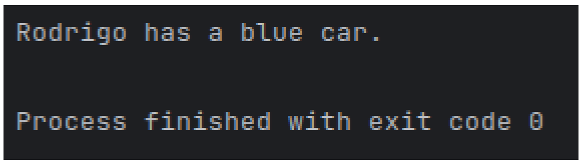


```
HELLO is upper? = True
WORLD is lower? = False

Process finished with exit code 0
```

capitalize()

It transforms the first letter of the `string` in an uppercase letter.

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

print(text)
```

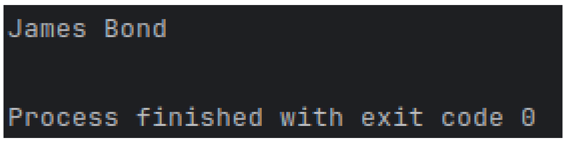


```
Rodrigo has a blue car.

Process finished with exit code 0
```

title()

It makes each word in the `string` start with an uppercase letter.

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

print(name)
```

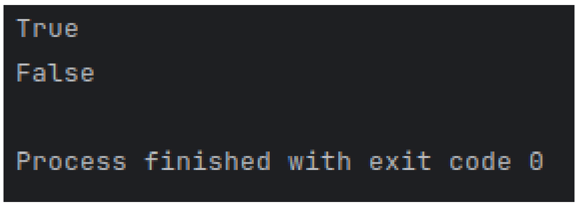


```
James Bond

Process finished with exit code 0
```

startswith() and endswith()

They check if the `string` begins or ends with the passed arguments, respectively.

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

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

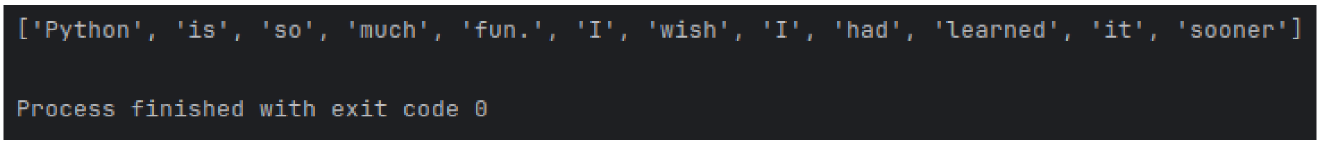


```
True
False

Process finished with exit code 0
```

split()

The `split()` method separates a string into elements and returns a list of them.

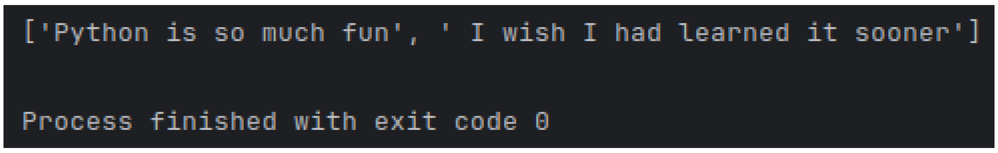
```python
message = "Python is so much fun. I wish I had learned it sooner"
message = message.split()

print(message)
```



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

Process finished with exit code 0
```

By default, it separates the string by its white spaces. But we can alter this behavior by passing an argument by which we want to separate the string. Next, we'll separate the same `string` by the full stop mark.

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

print(message)
```

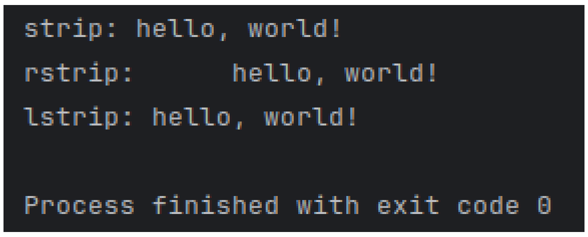


```
['Python is so much fun', ' I wish I had learned it sooner']

Process finished with exit code 0
```

strip(), rstrip() and lstrip()

The `strip()`, `rstrip()` and `lstrip()` methods remove elements of a string. `strip()` removes elements from both sides, `rstrip()` removes only form the right side, and `lstrip()` removes only from the left side. By default, they remove white spaces in the beginning, end or in both sides of the string.

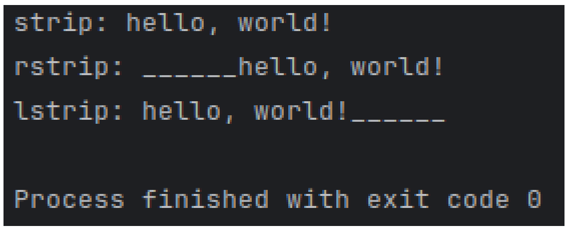
```python
message = "     hello, world!       "

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
```

However, it is possible to pass as an argument, the element you wish to remove. Maybe their uses will be clearer with the example below:

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

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

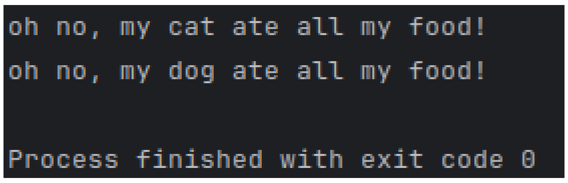


```
strip: hello, world!
rstrip: ______hello, world!
lstrip: hello, world!______

Process finished with exit code 0
```

replace()

This method replaces an element of the `string` with another. It receives two arguments: the part of the string that you want to replace and the part of the string with which you want to replace it.

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

text = text.replace("cat", "dog")
# changes the word "cat" in the string for the word "dog"
print(text)
```



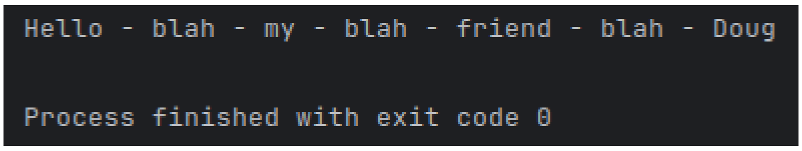
```
oh no, my cat ate all my food!
oh no, my dog ate all my food!

Process finished with exit code 0
```

In this example we're replacing the string "cat" with "dog".

join()

The `join()` method receives a list or tuple as an argument and concatenates each element a string as separator.

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

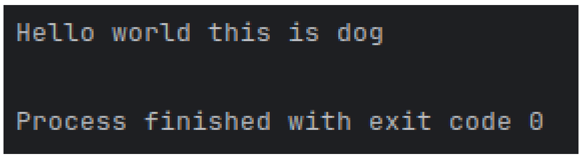


```
Hello - blah - my - blah - friend - blah - Doug

Process finished with exit code 0
```

I agree that it is a bit of a strange method. But maybe another example makes it clearer. Imagine you have a `list` and wishes to put all of its elements into a `string`, separated by a space.

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

Python will take the `string` we've passed, in this case, a white space `" "` and put it between each of the elements of the list passed as an argument.



```
Hello world this is dog

Process finished with exit code 0
```

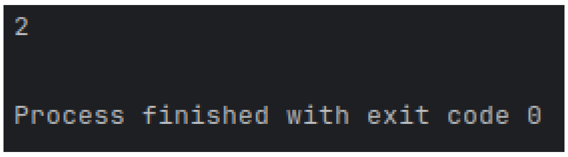
In other words, `" ".join(words)` means "take each element of the list `words` and put them into a string, separating them with `" "` ".

count()

The `count()` method takes a `string` as an argument and counts how many times it appears in the sentence. It's similar to the program we made on exercise 5-3.

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

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

The word "my" appears twice in the given `string`.



```
2

Process finished with exit code 0
```

Exercise 6

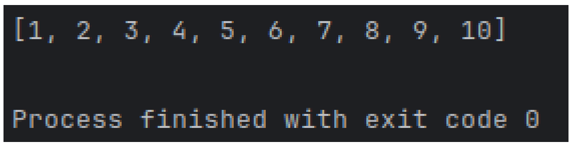
1. Create a program that receives a string from the user and prints the same string but with the first half all in uppercase and the second half all in lowercase.
2. Make a program that receives a phrase from the user and prints it backwards. Ex: "Hello world" -> "world Hello"
3. Make a program that verifies if the world or phrase given by the user is a palindrome (a word, phrase, or sequence that reads the same backward as forward)

List comprehension

List comprehension is a more concise form of creating a list, and it only exists in a few languages. In a single line, we can create a `for` loop, add the elements and even conditions.

Normally, we'd create a list in the following way:

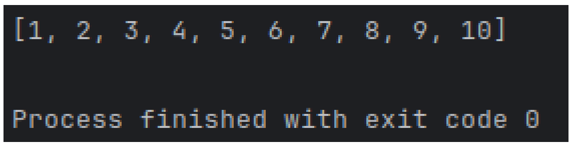
```python
numbers = []

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
```

However, with list comprehension, we can write the same thing in just one line:

```python
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
```

What are we doing here? First, we create a variable `numbers` and inside of the square brackets is where we put a list comprehension. There, we have `number for number in range(1, 11)`, in other words, add `number` to the list for each value of `number` from 1 to 11 (non-inclusive).

> ***Note:*** The first `number` in the list comprehension is the variable that will be added to the list.

Another way of **reading** the list comprehension is the following:

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

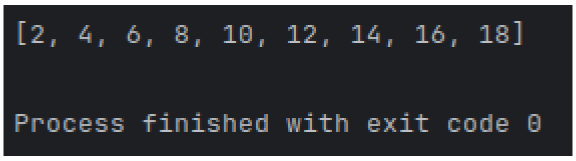
> ***Note:*** This code does not work, it's simply to show how to read a list comprehension.

The first variable inside the list comprehension - `number` - will be appended to the list being created.

Let's see some other examples. We'll make a list with only even numbers from 1 to 20 (non-inclusive).

Without list comprehension:

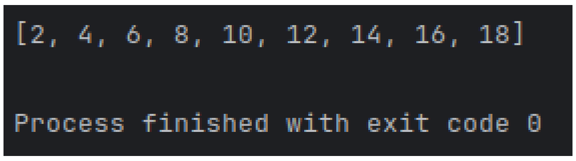
```
numbers = []

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
```

With list comprehension:

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

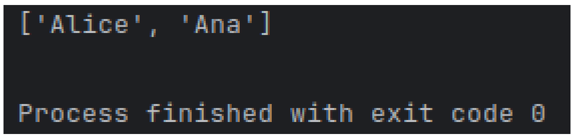


```
[2, 4, 6, 8, 10, 12, 14, 16, 18]

Process finished with exit code 0
```

In the example below, we have a list of names and we want to create another list only with the names that start with the letter a.

Without list comprehension:

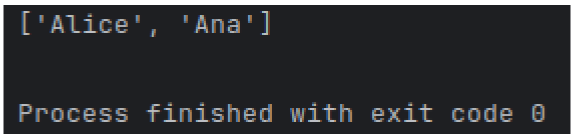
```
names = ["João", "Alice", "Janaína", "Ana", "Bruna", "Eduarda"]
names_with_a = []

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
```

With list comprehension:

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

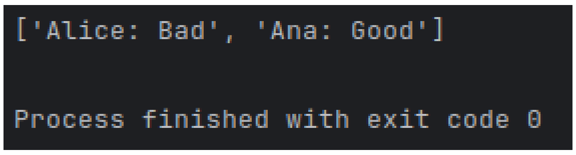


```
['Alice', 'Ana']

Process finished with exit code 0
```

Now we want to create a list that has only the names that start with A and, if their length is smaller than 4, I want to append "name: good", else I want to append "name: bad".

Without list comprehension:

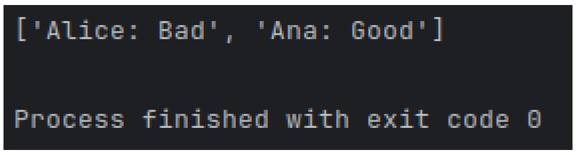
```python
names = ["João", "Alice", "Janaína", "Ana", "Bruna", "Eduarda"]
names_with_a = []

for name in names:
    if name.startswith('A'):
        if len(name) <= 4:
            names_with_a.append(f"{name}: Good")
        else:
            names_with_a.append(f"{name}: Bad")

print(names_with_a)
```



```
['Alice: Bad', 'Ana: Good']

Process finished with exit code 0
```

With list comprehension:

```python
names_with_a = [f"{name}: Good" if len(name) < 4 else f"{name}: Bad" for
name in names if name.startswith("A")]

print(names_with_a)
```



```
['Alice: Bad', 'Ana: Good']

Process finished with exit code 0
```

Structure

List comprehension can be a little hard to grasp at first. What i want you to understand is how it is structured.

1. First comes the variable that will be appended to the list, along with any filtering conditions

```
list_comprehension = [variable]
```

2. then comes the `for` loop

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

3. and finally you have the conditions to filter the `for` loop.

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

The example above is the equivalent of this:

```
list_comprehension = []

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

Let's see a few more examples to make it clearer. Pay attention to the structure.

Below I have a list of my favorite animals:

```
favorite_animals = ["Toucan", "Penguin", "Otter", "Wolf", "Owl", "Ocelot"]
```

Using list comprehension, let's create a list of my favorite animals that start with the letter O.

```
favorite_animals = ["Toucan", "Penguin", "Otter", "Wolf", "Owl", "Ocelot"]

animals_with_o = [animal for animal in favorite_animals if
animal.startswith("O")]
```

I'm going to iterate through all the animal names in the list `favorite_animals` and, if their names start with O, I'll append them to the list `animals_with_o`.

Again: first what's going to be appended, then the for loop, and finally the conditions.

One more example: I have a list of animals.

```
animals = ["Bear", "Zebra", "Sheep", "Lion", "Sloth", "Gecko"]
```

But I firmly believe that any animal with a name longer than 5 characters is not cool. So I want to create a new list that has "Cool", if the animal's name is smaller than 5 characters or "not cool", if it's not.

```
animals = ["Bear", "Zebra", "Sheep", "Lion", "Sloth", "Gecko"]

cool_animals = ["Cool" if len(animal) < 5 else "Not cool" for animal in
animals]
```

Here i'm appending "Cool" if the length of name is smaller than 5 and "Not cool" if it's not, for each name in the list `animals`.

A last one:

Let's go through the list `favorite_animals` and get all the animal names that start with O. If they have more than 5 characters of length, we'll append "Cute", of not, we'll append "Not cute."

```
favorite_animals = ["Toucan", "Penguin", "Otter", "Wolf", "Owl", "Ocelot"]

cute_animals = ["Cute" if len(animal) > 5 else "Not cute" for animal in
favorite_animals if animal.startswith("O")]
```

Here we're telling python to append "Cute" if `len(animal) > 5` else "Not Cute" for each animal in `favorite_animals` that starts with the letter O.

Exercise 7

> ***Note:*** Use *list comprehension* to solve these exercises.

1. Given the lists below, create a program that prints a list with only the numbers in common between them.

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

2. Make a program that generates a list of "even" or "odd" for each number from 1 to 20 (non-inclusive). Ex: [odd, even, odd, even...]
3. Make a program that asks the user for a sentence and print a list of all the words with 4 or less letters from it.

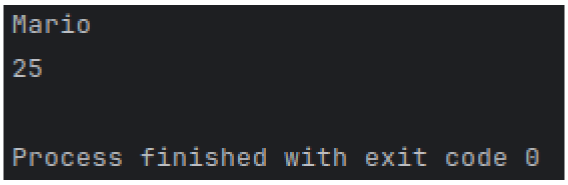
Dictionary

`dictionary` or `dict` is a data type that stores multiple elements, just like a list, and are used to store elements that are related. They are stored in pairs called "key" and "value", separated by a colon `:`. To create them, we utilize curly braces instead of square brackets.

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

In addition to being created with curly braces instead of square brackets, another big difference is that to access a value, we dont use a numeric index, we use the keys.

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



```
Mario
25

Process finished with exit code 0
```

Another difference is that the elements of a `dictionary` are unordered, while list elements are ordered.

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

In this case, the result is `False` because the values of each index are different. Lists are ordered, meaning that the order of each value is important.

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

Here, however, the result is `True` because dictionaries are unordered. Python doesn't care about the order of the elements, as long as they're all the same, the dictionaries will be equal.

Accessing elements

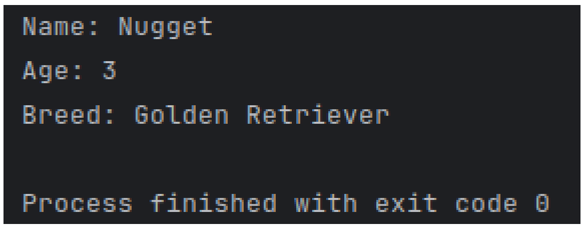
Though we've just seen how to access the elements of a `dictionary`, I want to explain it again so that, in case you'd like to remember later on, you can easily find it here.

We access the elements of a `dictionary` with their keys.

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

In this `dictionary` we have 3 keys - `name`, `age` and `breed` - and we use them to access their respective values.

```
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
```

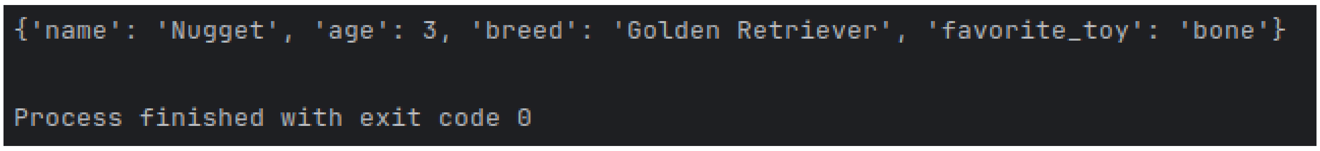
Modifying a dictionary

Adding elements

To add elements, we need to pass a key to be added and its respective value:

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

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

The key "favorite_toy" does not exist in the `dictionary`, so python will add it and assign to it the value "bone"

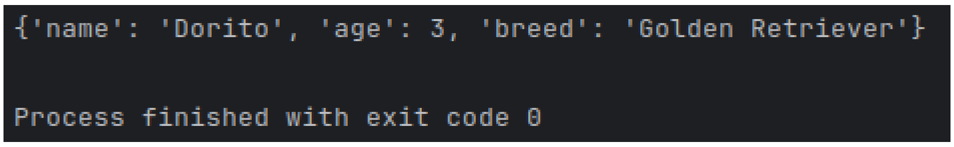


```
{'name': 'Nugget', 'age': 3, 'breed': 'Golden Retriever', 'favorite_toy': 'bone'}

Process finished with exit code 0
```

Modifying elements

To modify elements, we simply assign a new value to them.

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

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

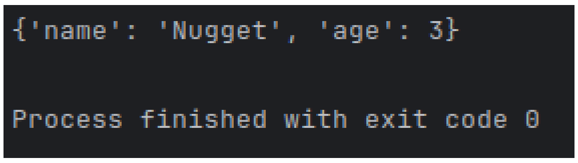


```
{'name': 'Dorito', 'age': 3, 'breed': 'Golden Retriever'}

Process finished with exit code 0
```

Deleting elements

To delete elements we utilize the `del` command.

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

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



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

Process finished with exit code 0
```

> ***Note:*** The element is permanently deleted.

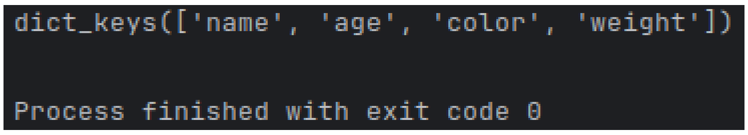
Loops and dictionaries

Just like we've done with the `list` data type, we can use loops to iterate through all the elements of a `dictionary`. But, for this, we need to utilize a few methods.

keys()

This method gives us access to only the keys of a `dictionary`.

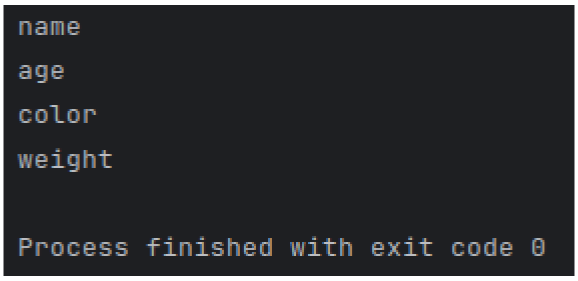
```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

print(catto.keys())
```



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

Process finished with exit code 0
```

And, this way, we can use a loop to access all the keys.

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

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

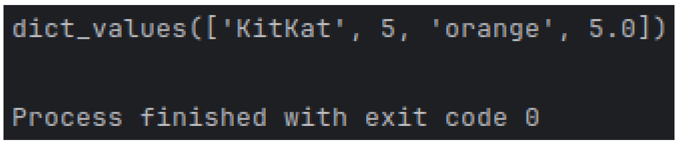


```
name
age
color
weight

Process finished with exit code 0
```

values()

Just like the `keys()` method, the `values()` method gives us access only to the values of a `dictionary`.

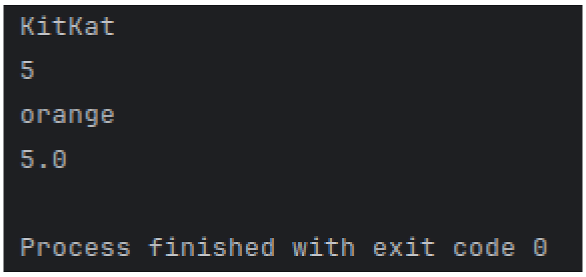
```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

print(catto.values())
```



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

Process finished with exit code 0
```

And with it we can use a loop to access only the values of a `dictionary`.

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

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

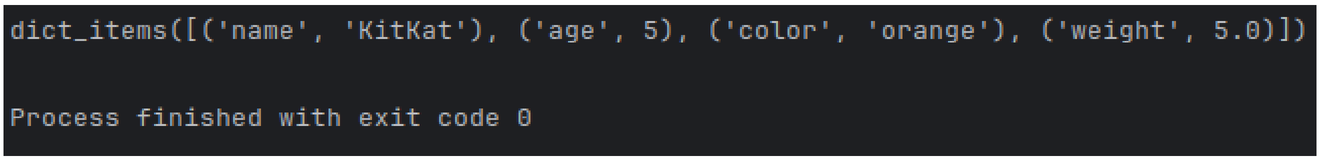


```
KitKat
5
orange
5.0

Process finished with exit code 0
```

items()

The `items()` method gives us access to both keys and values.

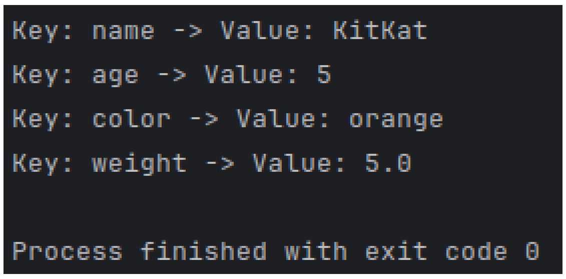
```
catto = {"name": "KitKat", "age": 5, "color": "orange", "weight": 5.0}

print(catto.items())
```



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

Process finished with exit code 0
```

Because of that, the `for` loop is a little different. It has two variables.

```python
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
```

Since this loop has two variables, one for `key` and another one for `value`, we can manipulate both inside the `for` loop.

> ***Note:*** Just like we've seen before, the name of the variables passed in the `for` loop can be anything. I've chosen "key" and "value".

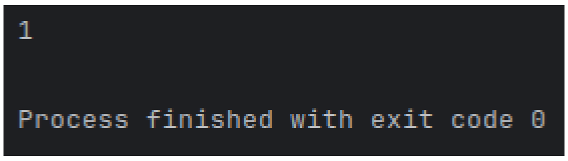
Most common methods

get()

The `get()` method can have one or two arguments. When we try to access a key that does not exist in a `dictionary`, python will raise an error. The `get()` method will return the value if the key exist and, if it doesn't, it will return the value we've passed as an argument.

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

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

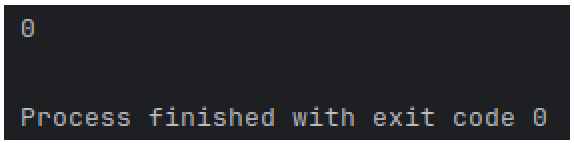
In this example, we're telling python to print the value of the key "bow", if it doesn't exist in the `dictionary`, print that the value is 0. The key exists in the `dictionary`, so python prints its value.



```
1

Process finished with exit code 0
```

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

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

In this example, however, the key "bow" doesn't exist in the `dictionary`, so python will print the default value.



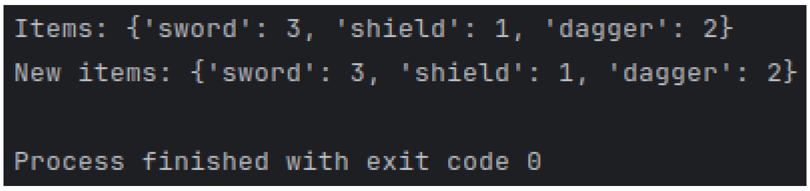
```
0

Process finished with exit code 0
```

> ***Note:*** If you only pass one argument, it will print the value if the word exists in the `dictionary` and, in case it doesn't, it will print the value `None`, which is a data type that means "no value".

copy()

This method allows us to create a copy of a `dictionary`.

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

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

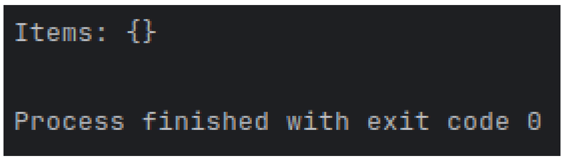


```
Items: {'sword': 3, 'shield': 1, 'dagger': 2}
New items: {'sword': 3, 'shield': 1, 'dagger': 2}

Process finished with exit code 0
```

clear()

`clear()` removes all the elements of a `dictionary`.

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

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



```
Items: {}

Process finished with exit code 0
```

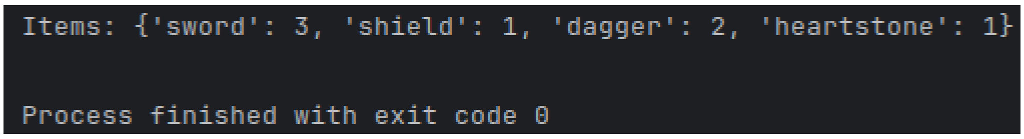
setdefault()

The `setdefault()` method makes the `dictionary` always have a certain key and value.

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

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

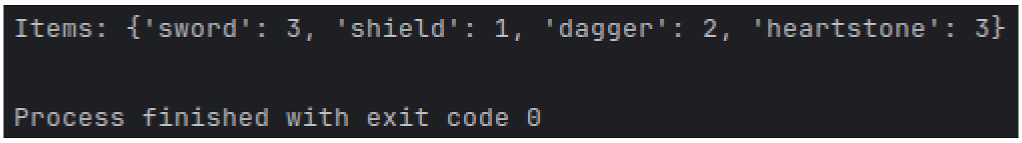
In the example above, we're defining that the `dictionary` must have at least 1 "heartstone". Since we didn't put one in the dictionary, python will put it.



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

Process finished with exit code 0
```

In case there is already a "heartstone" key in the `dictionary`, python won't do anything.

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

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

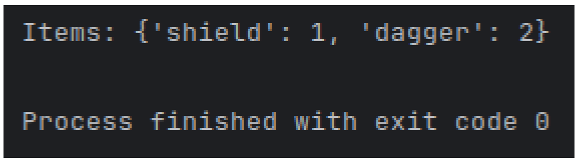


```
Items: {'sword': 3, 'shield': 1, 'dagger': 2, 'heartstone': 3}

Process finished with exit code 0
```

pop()

`pop()` is a method that receives a key as an argument and remove it from the `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
```

Beautifying dictionaries

Just like with the `list` data type, we can write data of type `dictionary` in such way that they're more readable, as we can see in the example below:

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

print(person)
```

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

Process finished with exit code 0
```

> ***Important:*** Do not forget to put a comma between the elements!

Exercise 8

Use the dictionary below for exercises 1 and 2:

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

1. Find the oldest person and print their name and age.
2. Find the average age of people and print it with 2 decimal places.
3. You were hired to create a program to interview 10 people about what they prefer, pizza or sushi. Make a program for this research that, in the end, print what the majority prefers and how many votes the winner received.

Tuple

`tuple` is an iterable data type, just like `list` and `dictionary`. The elements of a `tuple` are ordered, so we can access them by their indexes; however, a `tuple` is immutable. Once created, it cannot be modified.

A `tuple` is created by putting data in-between parenthesis.

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

print(numbers)
```

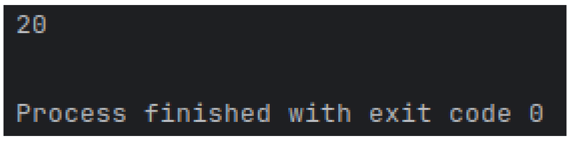
```
(10, 20, 30)

Process finished with exit code 0
```

> ***Important:*** A `list` or a `dictionary` inside of a `tuple` can still be modified; but the structure of the `tuple` cannot.

Accessing elements

Since `tuple` is ordered, its elements can be accessed by their indexes, just like `list`.

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

print(numbers[1])
```



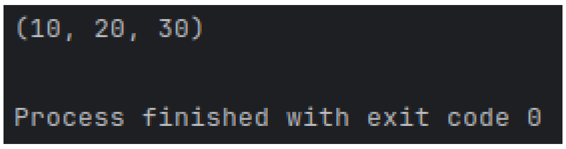
```
20

Process finished with exit code 0
```

> ***Important:*** The indexes always begin at 0.

Tuple slices

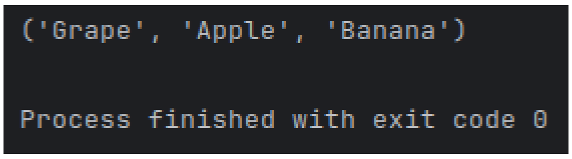
We can also access the elements of a `tuple` with slices.

```
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
```

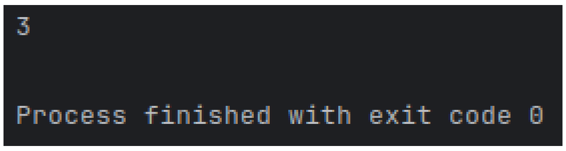
Methods

count()

It counts the amount of times an element appears in the `tuple`.

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

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

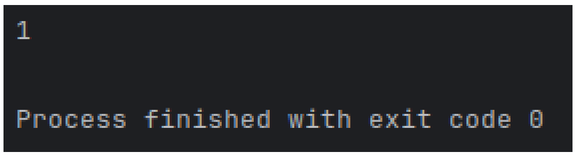


```
3

Process finished with exit code 0
```

index()

It shows the index of the element passed as an argument.

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

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

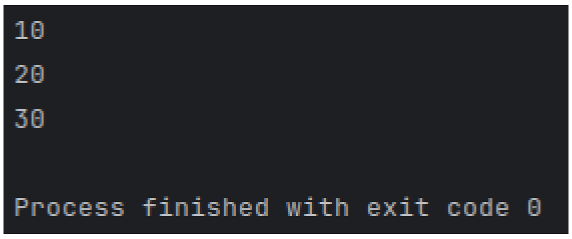


```
1

Process finished with exit code 0
```

Unpacking tuple

Unpacking tuple means to put each element of the `tuple` inside of a variable. Python allows us to do this in a very elegant manner.

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

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



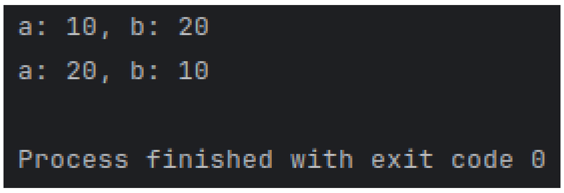
```
10
20
30

Process finished with exit code 0
```

Swapping variables

We can also swap the value of variables with `tuple`. Normally, if we want to do this, we would do it in the way shown below:

```
a = 10
b = 20

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

```python
c = a
a = b
b = c

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

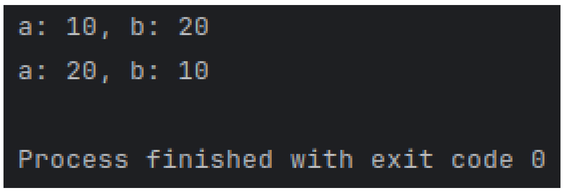


```
a: 10, b: 20
a: 20, b: 10

Process finished with exit code 0
```

We have two variables, `a` of value 10 and `b` of value 20, and want to swap their values so that `a` becomes 20 and `b` becomes 10. To do this, we create a new temporary variable `c`, then put the value of `a` in `c`, the value of `b` in `a` and of `c` in `b`. It's as if we had two cups, orange soda in cup A and grape soda in cup B, and wanted to put grape soda in cup A and orange soda in cup B. For this, we'd use a third empty cup called C. We'd put orange soda in cup C, grape soda in cup A and orange soda, which is currently in cup C, in cup B.

With `tuple` we can do this in a much easier way.

```python
a = 10
b = 20

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

a, b = b, a

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

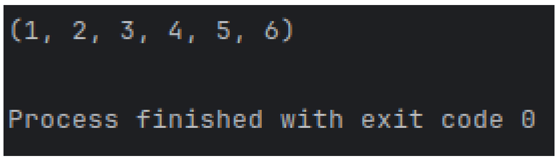


```
a: 10, b: 20
a: 20, b: 10

Process finished with exit code 0
```

Concatenating tuple

Though we cannot modify tuples, we can concatenate them to create new ones.

```python
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
```

Exercise 9

Use the `tuple` below to solve the exercises 1, 2 and 3:

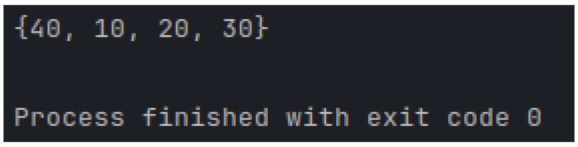
```
numbers = (7, 42, 93, 58, 12, 24, 30)
```

1. Print the last three numbers
2. Print the average of all the elements
3. Print the tuple backwards
4. You're working with a team to create a 2D game. In such games, character's positions are defined by the coordinates (x, y). You're asked to create a new spell for the player character that swaps their location with their enemy. Create the code for this mechanic, print the previous positions and the current ones.

Set

`set` is an iterable data type that removes duplicated elements. Its elements are unorganized, meaning that we cannot access them by indexes (it is common that elements change order when printed).

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

print(numbers)
```



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

Process finished with exit code 0
```

As we can see, all the duplicated numbers were removed and the order in which they were printed is different than the order in which the `set` is written.

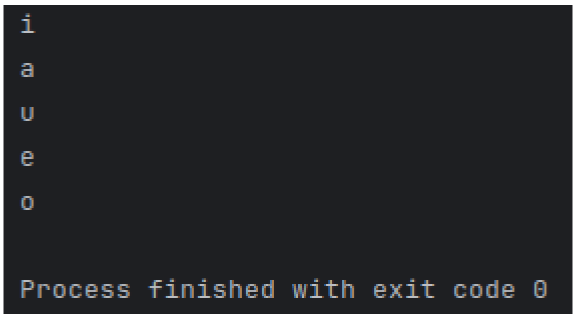
The data of type `set` can only contain immutable data, which means data that cannot be altered. If you try to add a data of type `list` or `dict` inside of a `set`, python will raise an error.

> ***Important:*** Be careful not to confuse `set` with `dictionary`. Both use curly braces but `dictionary` has key-value pairs.

Accessing elements

Since the elements of a `set` are not ordered, we can't access them by indexes. Because of this, the only way of accessing them is through a 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
```

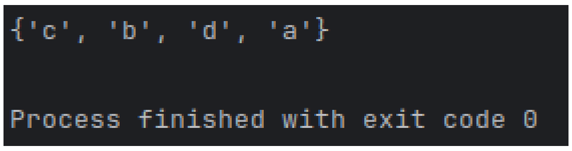
Modifying a set

Adding elements

add()

Through the `add()` method we can add an element to a `set`.

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

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

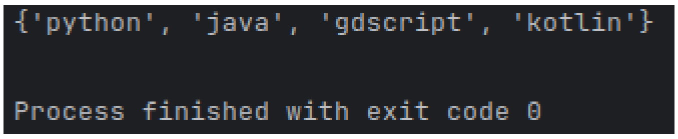


```
{'c', 'b', 'd', 'a'}

Process finished with exit code 0
```

update()

The `update()` method takes an iterable data as an argument and, this way, allows us to add multiple elements at once.

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

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



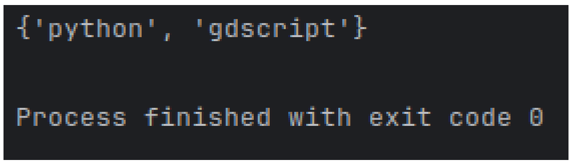
```
{'python', 'java', 'gdscript', 'kotlin'}

Process finished with exit code 0
```

Removing elements

remove()

The `remove()` method removes an element from the `set`. If the element is not in the `set`, it raises an error.

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

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



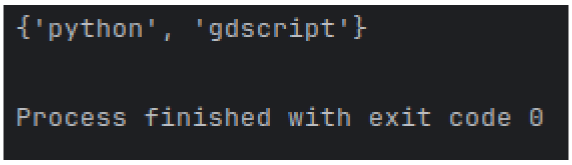
```
{'python', 'gdscript'}

Process finished with exit code 0
```

> ***Note:*** We'll learn how to deal with errors later.

discard()

`discard()` also removes an element from a `set`, but if the element doesn't exist, it doesn't raise an error.

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

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



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

Process finished with exit code 0
```

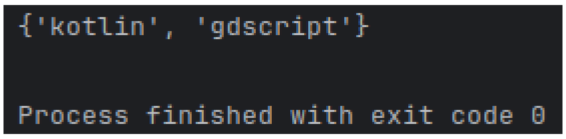
pop()

The `pop()` method with a `set` has to use cases. It can:

1. remove a **random** element from the set
2. remove a **random** element and assign it to a variable

First use case:

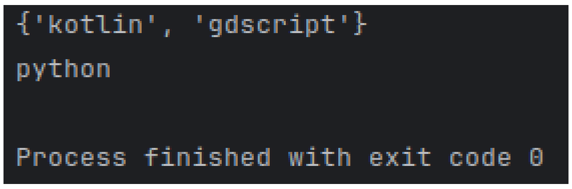
```python
languages = {"python", "gdscript", "kotlin"}

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



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

Process finished with exit code 0
```

Second use case:

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

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



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

Process finished with exit code 0
```

Set operations

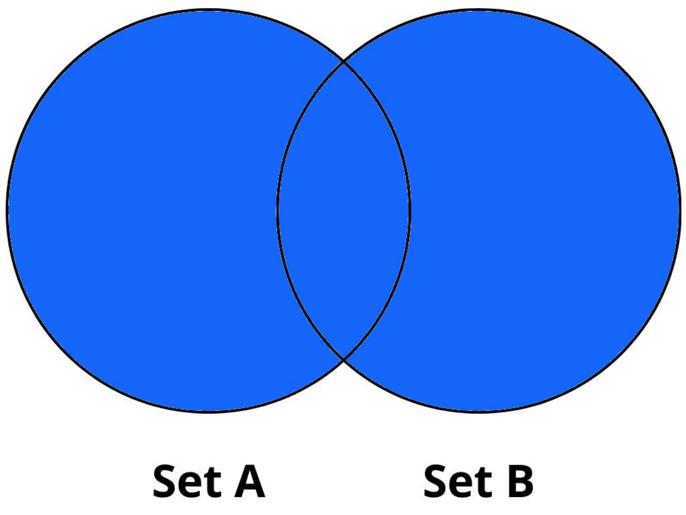
Just like in math, set operations such as union, intersection and difference can be done with a set. I know that we get scared when math is involved. But I'll try to explain it all in detail and draw for you to see that it is not as hard as it seems.

There are operators and methods for each of these operations. I'll be showing both.

> ***Note:*** In the examples I used two sets only, but it is also possible to do operations with three or more.

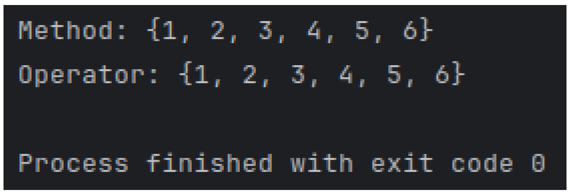
Union

When we make a union of sets, we take all the elements contained in all the sets.



In python, we can utilize the `union()` or the `|` operator.

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

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

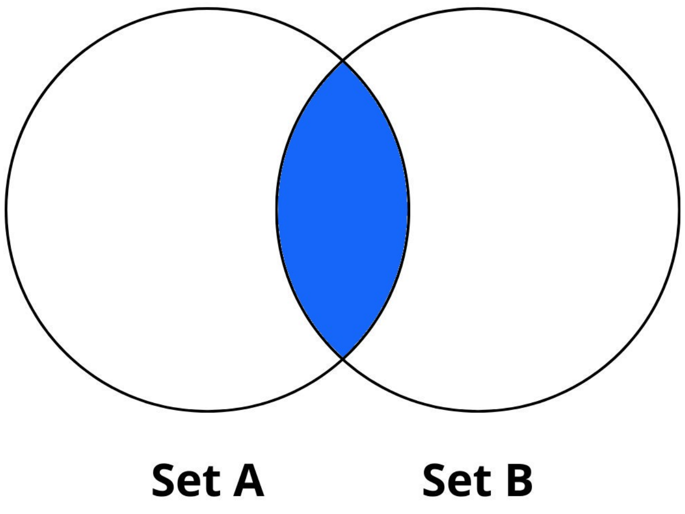


```
Method: {1, 2, 3, 4, 5, 6}
Operator: {1, 2, 3, 4, 5, 6}

Process finished with exit code 0
```

As we can see, a new set was created containing all the elements of the sets `numbers` and `more_numbers`, with no duplicates, since `set` data types don't allow them.

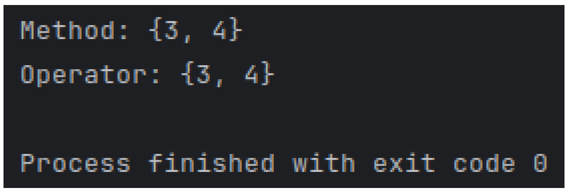
Intersection

When we do the intersection of sets, we simply take the unique elements that belong to all the sets.



In python, we can utilize the `intersection()` method or the `&` operator.

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

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

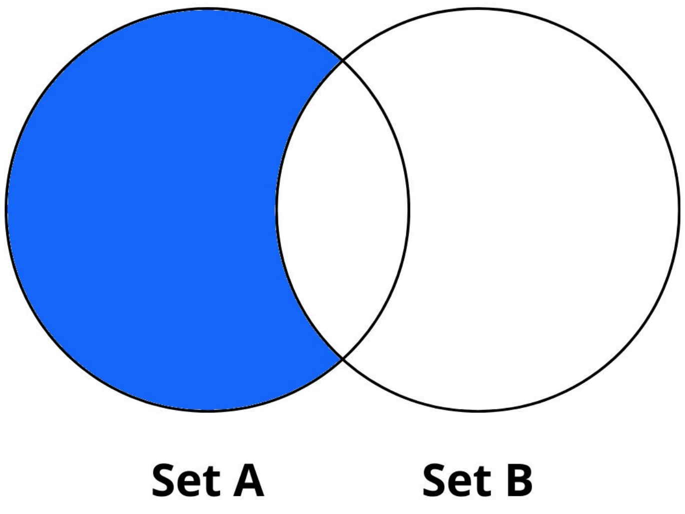


```
Method: {3, 4}
Operator: {3, 4}

Process finished with exit code 0
```

Here a new set was created containing the elements that appear in both the `numbers` set and the `more_numbers` set. Again, without duplicates because data of type `set` does not allow for duplicated values.

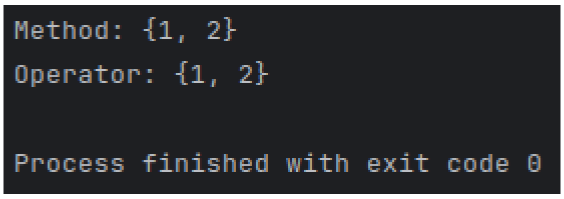
Difference

When we do the difference between sets, we take only the elements that exist in one set but not the others.



In python, we can use the `difference()` method or the `-` operator.

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

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

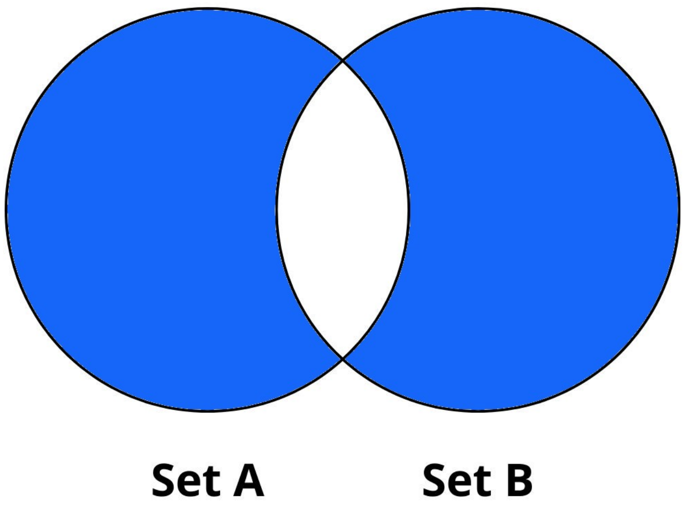


```
Method: {1, 2}
Operator: {1, 2}

Process finished with exit code 0
```

A new set was created containing only the numbers that are present in the `numbers` set, but that are not present in the other sets.

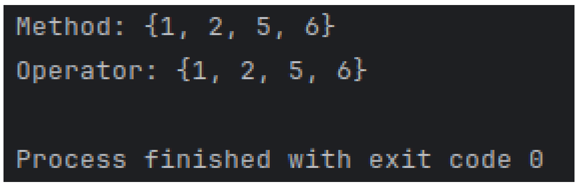
Symmetric difference

When we do the symmetric difference of sets, we take all the elements that are unique to each set. Any element that is present in both sets is ignored.



In python, we can use the `symmetric_difference()` method or the `^` operator.

```python
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
```

In this case, we're creating a new set with all the elements that belong to the `numbers` set and to the `more_numbers` set. Any other element that is present in both sets was not included.

Frozenset

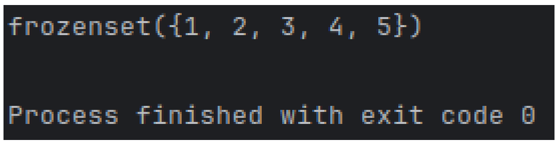
Data of the type `frozenset` are immutable sets, meaning the original cannot be modified.

We can create them using the `frozenset()` function.

If we don't pass any arguments, the function will create an empty data of type `frozenset`. In case we do pass an iterable as an argument to the function, it will do a type casting and turn the data into a frozenset.

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

print(numbers)
```

In this example, we're passing a list as an argument and doing the type casting to frozenset. This is the result:



```
frozenset({1, 2, 3, 4, 5})

Process finished with exit code 0
```

More type casting

We know that we can alter the type of a data through what we call type casting. And we can also do this with collection data types.

In general, they work the say way as we've seen before. The only difference is that they have to take an iterable (that we can access using a loop) data as an argument, like a string or a collection.

```python
list(data)
```

Converts the data to the `list` type.

```python
tuple(data)
```

Converts the data to the `tuple` type.

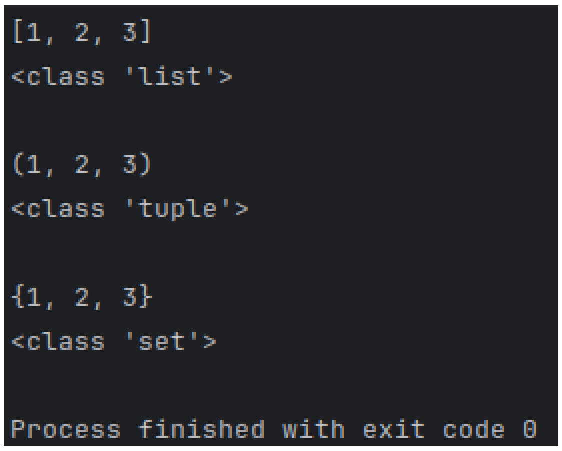
```python
set(data)
```

Converts the data to the `set` type.

And we can visualize this with the `type()` function, which tells us the type of the passed data.

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

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

```python
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
```

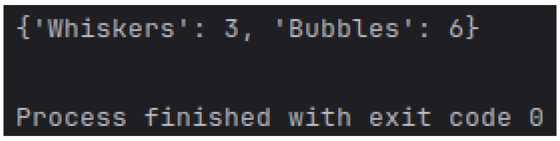
> ***Note:*** We will later learn how to create our own classes.

Type casting to the `dictionary` data type is a little different. It receives a list, a set or a tuple of tuples of length 2 and then converts them to a dictionary.

As we know, dictionaries store elements in key-value pairs. Because of this, to transform a data to the `dict` type, we need to pass as arguments data of type `tuple` of length 2.

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

print(dict(name_and_age))
```

In this example, we hae a list of data of `tuple` type (but it could be a set of tuples or a tuple of tuples), and we're converting it to the `dict` type.



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

Process finished with exit code 0
```

Exercise 10

Given the lists below:

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

1. Print the elements that appear in one list but not the others.

2. Print all the elements of all three lists without duplicates.
3. Create a program to check if the list below has repeated numbers and print "Yes, it has repeated numbers" or "No, it doesn't have repeated numbers".

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

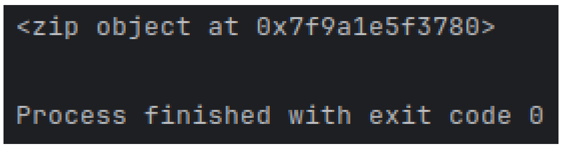
---

zip() function

The `zip()` function takes 2 or more iterable data as arguments and turns them into an object of type `zip`, which contains a group of tuples and is iterable. I know it sounds too complicated, but it will be clearer with some examples.

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

print(my_zip)
```

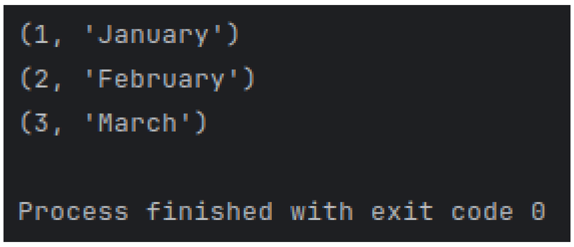
In this example we have two lists and we're using the `zip()` function to join them. If we try to print it directly, python will print that it is an object of type `zip` and where it is located in memory.



```
<zip object at 0x7f9a1e5f3780>

Process finished with exit code 0
```

Objects of type `zip` are iterable, so we can use a loop to access its elements.

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

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



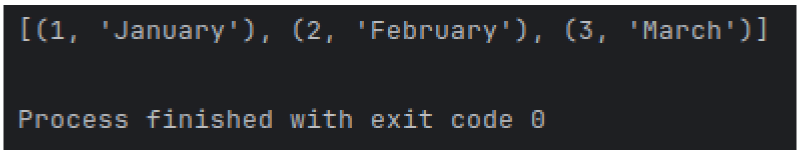
```
(1, 'January')
(2, 'February')
(3, 'March')

Process finished with exit code 0
```

When accessing them, we can see that the object of type `zip` took the elements of each list and created tuples. The first element with the first element, the second element with the second element and so on.

So the object of type `zip` contains one or more elements of type `tuple`. And we can use type casting to turn a `zip` object into a list, a set, a tuple or even a dict.

In the example below, we transform it into a list of tuples:

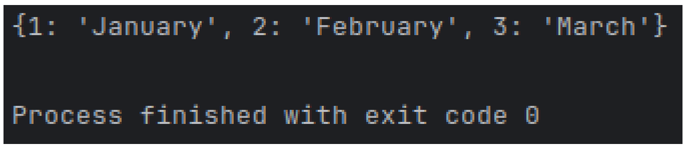
```python
positions = [1, 2, 3]
months = ["January", "February", "March"]
zip_list = list(zip(positions, months))

print(zip_list)
```



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

Process finished with exit code 0
```

In this example, we turn it into a dict:

```python
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
```

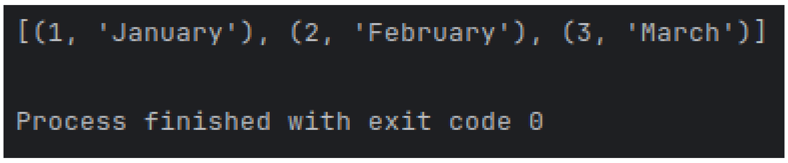
> ***Important:*** It is only possible to turn an object of type `zip` in an object of type `dict` when the zip has only tuples of size 2, because dicts only accept pairs of elements.

Iterables of different lengths

When we pass iterables of different lengths as arguments to the `zip()` function, the zip object will have its length equal to the length of the shortest argument. Let me use an example to demonstrate this:

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

print(zip_list)
```

Here we're passing two arguments to the `zip()` function: the `positions` list that has a length of 7, meaning it has 7 elements; and the `months` list that has only 3. In this case, the created zip object will have its length equal to the shortest argument passed, the `month` list. The length of the zip object will be 3.



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

Process finished with exit code 0
```

Optimization

I know that we've seen many data types and their different uses. But how do I know which data type to use? There is no correct answer to this question. There are many solutions to the same problem. You can get to the same result utilizing different data types and it is up to you to choose which one you think is best.

With this in mind, it is important that we talk about optimization. We say a code is optimized when it is efficient. It uses the minimum amount of memory required for it to work or it reaches a result in a fast manner. And choosing which data type is needed for your algorithm does make a difference. For example, if you know that a set of data will not and cannot be changed during runtime, use a data of type `tuple` instead of a list, for the tuple has these characteristics specifically - it is optimized for it.

When we create more than one algorithm to solve a problem, we can evaluate which is more efficient using what is called Big O Notation. Imagine you've created two algorithms that modify a data of type `string`. When we're working with short strings, both algorithms are fast and solve the problem in milliseconds. But what about when we work with strings with a thousand characters, or one hundred thousand, which one would be faster? The Big O Notation is used to solve this question, evaluating them and classifying them.

The most important thing is to make it work

I don't believe that it is relevant for us to study optimization for now, so I don't want you to worry about it. I'm only mentioning it so you know this exists. We'll learn more about it on a later chapter.

I made sure to put this in the title so that it is clear: Optimization comes later, if there is need for it. **The first and most important part is to make the program work.** If you'd like to go back and optimize your code later down the road, if there is need, okay. Great. But in case there is no need, move on.

The intention here is that you know these terms and have an idea of what they mean. It is common to hear cases in which excellent programmers learned to code by themselves and felt completely lost once they entered the industry and started hearing these technical terms they had never heard of.

So don't worry about it for now. Let's continue studying because first we need to learn how to do it, then we can learn how to improve what we've made.

Bonus: A deeper look into variables

You're already quite comfortable with variables, what they are and how to use them. So now I'd like to explain them a little further.

In most courses I've done and books I've read, variables were described as "different than the variables in math". And if you've done other courses or have seen them being described in such way, you were probably as confused as I was; because if we stop to think about it, the variables we've used so far are identical to the variables in math. In mathematics, a variable `x` has a value that can be an integer or a floating-point number, for example. So why is it that they are explained in such way? This is the question that is not answered in most courses, so we'll answer it here.

We know that variables are stored in memory addresses. When we create a variable and give it a name, this name is simply a reference, or a label, to the memory address in which the data is located.

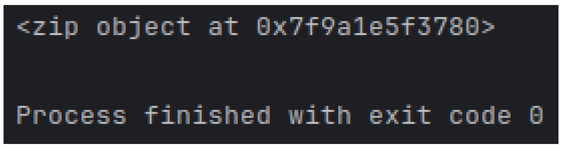
Imagine you're returning home and run into a friend, and he asks "Where are you going?" and you asnwer "i'm going home." When you say "home", your friend understands that you're referring to your address, which is Mozzarella street, number 70, Cheese county. So you can either say your full address, or you can use a more descriptive reference. And this is how it works with variables.

When we create a variable called `hello` and call the `print()` function to print it, python understand that when you say `hello`, you're referring to some data that is located on that address in memory.

To make this a little more visible, let's take another look at the `zip()` function we saw earlier.

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

print(my_zip)
```

When we try to print the variable `my_zip`, python understands that the variable name is simply a reference to an address in memory.


```
<zip object at 0x7f9a1e5f3780>

Process finished with exit code 0
```

When we print it, it tells us that inside the memory address `0x7f9a1e5f3780` there is a zip object. Python understands that the variable name `my_zip` is a reference to the memory address `0x7f9a1e5f3780`.

And this is how a computational variable differs from the mathematical variable. It isn't simply a name with a value, it is a name that points to memory address, where a value is located.
