

# print

Statement that when executed shows (or prints) on the screen the argument inside the parentheses.

## display text

Write text characters that should be displayed on the screen between single or double quotes.

```
print("Hello world")
```

```
>> Hello world
```

## display numbers

The number can stand alone or be the result of a math expression. We don't use quotation marks in these cases.

```
print(150 + 50)
```

```
>> 200
```

# strings

Strings in Python are a data type made up of sequences of characters of any type, that make up some text.

## concatenation

Union of text strings:

```
print("Hello" + " " + "world")  
>> Hello world
```

## special characters

We can tell the console that the character after the \ symbol should be treated as a special character.

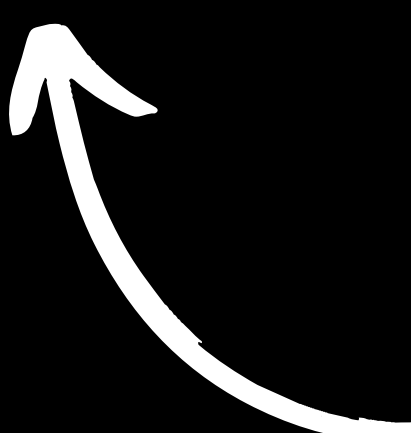
```
\ " > Print quotes  
\n > Separate text on a new line  
\t > Print a tab  
\& > Print a backslash
```

# input

Function that allows the user to enter information through the keyboard when executed, while giving an instruction about the requested entry. The code will continue to run after the user performs the action.

```
input("Write your name: ")
```

```
>> Write your name: |
```



Waiting for user input  
(keyboard)

```
print("Your name is " + input("Write your  
name: "))
```

```
>> Write your name: Federico
```

```
>> Your name is Federico
```

# Day 1 Python Challenge

**The most awaited moment of your first day of study has arrived.** You are going to create your first solo program, and you are going to do it applying everything you have learned throughout this day.

Imagine the situation: Your best friend has started a brewery and has everything ready. His product is fantastic. It has body, good flavor, good color, and just the right level of foam. But it lacks an identity. He can't think of a name for his beer that would give it a unique and original identity.

So, you come to him and say: *“Don't worry. I'm going to create a program that will ask you two questions and then tell you what to name your beer. It's as simple as that.”*

I know that in the real world, we wouldn't need to develop a software just to ask it two questions. But until we learn a bit more functionality, well, our programs are going to have to stay in the realm of simplicity. Yet if you're just starting out, this is going to be quite a challenge.

**You're going to create Python code that asks your friend to answer two questions that require a single word each, and then displays those combined words on the screen to form a creative brand.**

**You can use any questions you want. The idea is for the outcome to be original, creative, and even funny. And if you want to add some difficulty to your challenge, I suggest you try to have the name of the beer printed in quotation marks on the screen.**

Remember that there are different ways for the print function to show the quotes without interrupting the stream, and it's also possible to print out text in at least two lines using line breaks within the code.

Well, try to do it on your own, and if it gets complicated, don't worry, we will solve it together in the next lecture.

Cheers and good luck.

# data types

In Python we have several data structures, which are fundamental in programming since they store information and allow us to manage it.

**text (str)**

"Python"  
"750"

**numbers**

**int** 250  
**float** 12.50

**booleans**

True  
False

**structures**

mutable

ordered

duplicates

lists []



tuples ()



sets {}



dictionaries {}



\*: In Python 3.7+, there are new features \*\*: key is unique ; value can be repeated

# variables

Variables are containers for storing data values of different types, and (as their name suggest), they can change over the code's execution. A variable is created the moment you first assign a value to it, so you don't need to declare them first.

## some examples

```
country = "Thailand"
```

```
name = input('Enter your name: ')\nprint("Your name is " + name)
```

```
num1 = 55\nnum2 = 45\nprint(num1 + num2)\n>> 100
```



# naming variables

There are conventions and best practices associated with naming variables in Python. They are aimed towards readability and maintainability of the code.

## rules

1. **Readable:** variable name should be relevant to its content
2. **Unit:** there are no spaces (instead, use underscores)
3. **Plain:** omit certain language-specific signs, such as ñ, è, ç
4. **Numbers:** Variable names must not start with numbers (although we can have them at the end of the variable name)
5. **Symbols:** Must not include : " ' , < > / ? | \ ( ) ! @ # \$ % ^ & \* ~ - +
6. **Keywords:** we do not use Python reserved words

# integers & floats

There are two basic numeric data types in Python: int and float. Like any variable in Python, its type is defined the moment we assign a value to a variable. You can get the data type of a variable with the `type()` function.

## int

An integer, positive or negative, without decimals, of indeterminate length.

```
num1 = 7  
print(type(num1))  
>> <class 'int'>
```

## float

Number that can be positive or negative, which in turn contains one or more decimal places.

```
num2 = 7.525587  
print(type(num2))  
>> <class 'float'>
```



# type conversions

Python performs implicit data type conversions automatically to operate on numeric values. If you want to specify the data type, you can use constructor functions.

```
int(var)  
>> <class 'int'>
```



Setting the data type as integer

```
float(var)  
>> <class 'float'>
```



Setting the data type as float

# formatting strings

We have two main tools to mix static text and variables into strings:

- **Format function:** enables you to concatenate parts of a string at desired intervals. Multiple pairs of curly braces can be used while formatting the string. Python will replace the placeholders with values in order.

```
print("My car is {}, and it's license  
plate is {}".format(car_color, plate))
```

- **formatted string literals (f-strings):** the newer way to format strings (Python 3.8+), with a simple and less verbose syntax: just include `f` at the beginning of the string and call the variables inside curly brackets.

```
print(f"My car is {car_color} and it's  
license plate is {plate}")
```

# arithmetic operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Addition: **+**

Subtraction: **-**

Multiplication: **\***

Division: **/**

Floor division: **//**

Modulus: **%**

useful for detecting  
even values :)

Exponentiation: **\*\***

Square root: **\*\*0.5**

just a special case  
of exponentiation

# round

Rounding makes it easier to read calculated values by limiting the number of decimal places displayed on the screen. It also allows us to approximate decimal values to the nearest integer.

`round(number, ndigits)`

number to be rounded ←

→ number of decimals to use  
(default is 0 = int)

## examples

```
print(round(100/3))  
>> 33
```

```
print(round(12/7, 2))  
>> 1.71
```

# Day 2 Python Challenge

**Congratulations:** You've finished learning all the content of **Day 2**, and now you're able to create a slightly more complex program.

You already know how to do things like mathematical operations, data type conversions, string formatting and rounding. Let's celebrate with a new challenge.

**The situation is this: You work in a company where salespeople receive 13% commissions on their total sales, and your boss wants you to help the salespeople calculate their commissions by creating a program that asks them their name and how much they've sold this month.**

**Your program will answer them with a sentence that includes their name and the amount of commissions they are entitled to.**

Now, this is not a complex program, but that's okay, you're still learning. Even though what you have learned so far is very simple, putting it all together in one program can make your new skills get a little bit messy.

A couple of guidelines:

- This program should start by asking the user for things. So, you're going to need to use input to receive user input, and you should use variables to store that input. Remember that user inputs are stored as strings, so you should convert one of those inputs to float to be able to do operations on it.
- And what operation do you need to do? Well, calculate 13% of the number that the user has entered. That is, multiply the number by 13, then divide by 100.
- It would be good to print the result on the screen, so make sure this information has no more than two decimals, so it is going to be easy to read. Then, organize all that in a string that you would format. Remember, we learned two ways to do it, and any of them is valid.

All right, go on. Try to solve it. And if it gets complicated, Federico will do it with you in the next lesson.

Go ahead and good luck.

# index()

The `index()` method returns the index position at which an item is found in a list or a string. We can use it to explore strings, since it allows us to find the occurrence of a character or a substring in another string.



`string[i]` → returns the character at index `i`\*

*\*: In Python, the first position index is 0*



# substrings

We can extract chunks of text using a string manipulation tool known as slicing.

start index of substring  
(included)

optional argument that  
determines the increment  
between each index

`string[start:stop:step]`

where the slicing stops  
(not included)

```
my_var = " H e l l o W o r l d "
```

```
print(my_var[2:6])
```

```
>> llow
```

```
H e l l o W o r l d
```

```
0 1 2 3 4 5 6 7 8 9
```

```
print(my_var[3::3])
```

```
>> lod
```

```
H e l l o W o r l d
```

```
0 1 2 3 4 5 6 7 8 9
```

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

```
>> dlroWolleH
```

```
H e l l o W o r l d
```

```
-9 -8 -7 -6 -5 -4 -3 -2 -1 0
```

# strings: methods

## analysis methods

**count()** : returns the **number of times** a specified set of characters is repeated.

```
"Hello world".count("Hello")  
>> 1
```

**find()** & **index()** return the **location** (starting at 0) where the given argument is found. They differ in that **index** raises **ValueError** when the argument is not found, while **find** returns **-1**.

```
"Hello world".find("Bye")  
>> -1
```

**rfind()** & **rindex()** To search for a set of characters starting from the end.

```
"C:/python36/python.exe".rfind("/")  
>> 11
```

**startswith()** & **endswith()** indicate whether the string in question **begins** or **ends** with the set of characters passed as an argument, and return **True** or **False** accordingly.

```
"Hello world".startswith("Hello")  
>> True
```

# strings: methods

## analysis methods

**isdigit()**: returns **True** if all the characters in the string are digits, or can form numbers, including those corresponding to oriental languages.

```
"abc123".isdigit()  
>> False
```

**isnumeric()**: returns **True** if all characters in the string are numbers, it also includes characters with numeric connotation that are not necessarily digits (for example, a fraction).

```
"1234".isnumeric()  
>> True
```

**isdecimal()**: returns **True** if all characters in the string are decimals, that is, formed by digits from 0 to 9.

```
"1234".isdecimal()  
>> True
```

**isalnum()**: returns **True** all characters in the string are alphanumeric.

```
"abc123".isalnum()  
>> True
```

**isalpha()**: returns **True** if all characters in the string are alphabetic.

```
"abc123".isalpha()  
>> False
```

# strings: methods

## analysis methods

**islower()**: returns **True** if all characters in the string are **lowercase**.

```
"abcdef".islower()  
>> True
```

**isupper()**: returns **True** if all characters in the string are **uppercase**.

```
"ABCDEF".isupper()  
>> True
```

**isprintable()**: returns **True** if all characters in the string are **printable** (that is, **not special characters** indicated by **\...**).

```
"Hello \t world!".isprintable()  
>> False
```

**isspace()**: returns **True** if all characters in the string are **spaces**.

```
"Hello world".isspace()  
>> False
```

# strings: methods

## transformation methods

*Strings are actually immutable objects, so in reality all the methods below do not act on the original object but return a new one.*

**capitalize()** returns the string with its **first letter capitalized**.

```
"hello world".capitalize()  
>> 'Hello world'
```

**encode()** **encodes** the string with the **specified character map** and returns an instance of type bytes.

```
"Hello world".encode("utf-8")  
>> b'Hello world'
```

**replace()** **replaces** one string with another.

```
"Hello world".replace("world", "everyone")  
>> 'Hello everyone'
```

**lower()** returns a copy of the string with all its letters in **lowercase**.

```
"Hello World".lower()  
>> 'hello world'
```

**upper()** returns a copy of the string with all its letters in **uppercase**.

```
"Hello world".upper()  
>> 'HELLO WORLD'
```



# strings: methods

## transformation methods

**swapcase()** **change** uppercase to lowercase and vice versa.

```
"Hello World".swapcase()  
>> 'hELLO wORLD'
```

**strip()**, **lstrip()** & **rstrip()** **remove whitespaces** that precede and/or follow the string.

```
" Hello world ".strip()  
>> 'Hello world'
```

**center()**, **ljust()** & **rjust()** **align a string to the center, left, or right**. A second argument indicates with which character to **fill** the empty spaces (by default: blank space).

```
"hello".center(9, "*")  
>> '**hello**'
```

## splitting and joining methods

**split()** **splits** a string based on a **separator character** (defaults: blanks). A second argument indicates the maximum number of splits that can take place (-1 by default, meaning an unlimited number of splits).

```
"Hello world!\nHello everyone!".split()  
>> ['Hello', 'world!', 'Hello', 'everyone!']
```



# strings: methods

## splitting and joining methods

**splitlines()** splits a string with each occurrence of a line break.

```
"Hello world!\nHello everyone!".splitlines()  
>> ['Hello world!', 'Hello everyone!']
```

**partition()** returns a tuple of three elements: the block of characters before the first occurrence of the separator, the separator itself, and the block after.

```
"Hello world! Hello everyone!".partition(" ")  
>> ('Hello', ' ', 'world! Hello everyone!')
```

**rpartition()** operates in the same way as the previous one, but starting from right to left.

```
"Hello world! Hello everyone!".rpartition(" ")  
>> ('Hello world! Hello', ' ', 'everyone!')
```

**join()** must be called from a string that acts as a separator to join the elements of a list into the same resulting string.

```
", ".join(["C", "C++", "Python", "Java"])  
>> 'C, C++, Python, Java'
```

# string: properties

Here's what to keep in mind when working with strings in Python:

- **they are immutable:** once created, their parts cannot be modified, but the values of the variables can be reassigned through string methods
- **concatenable:** it is possible to join strings with the symbol +
- **multipliable:** it is possible to concatenate repetitions of a string with the symbol \*
- **multilinear:** can be written on multiple lines by enclosing them in triple single quotes (""" """) or double quotes (""" """)
- **determine its length:** through the `len(my_string)` function
- **check its content:** through the keywords `in` and `not in`. The result of this verification is a boolean (`True` / `False`).

# lists

Lists are ordered sequences of objects. These objects can be data of any type: strings, integers, floats, booleans, lists, among others. They are mutable data types.

mutable ✓

ordered ✓

<sup>allow</sup>  
duplicates ✓

```
list_1 = ["C", "C++", "Python", "Java"]  
list_2 = ["PHP", "SQL", "Visual Basic"]
```

**indexing:** we can access the elements of a list through their indices [start:end:step]

```
print(list_1[1:3])  
>> ["C++", "Python"]
```

**item count:** through property len( )

```
print(len(list_1))  
>> 4
```

**concatenation:** we add the elements of several lists with the + symbol

```
print(list_1 + list_2)  
>> ['C', 'C++', 'Python', 'Java', 'PHP', 'SQL', 'Visual Basic']
```

# lists

```
list_1 = ["C", "C++", "Python", "Java"]  
list_2 = ["PHP", "SQL", "Visual Basic"]  
list_3 = ["d", "a", "c", "b", "e"]  
list_4 = [5, 4, 7, 1, 9]
```

**append()** function: add an element to a list *in place*

```
list_1.append("R")  
print(list_1)  
>> ["C", "C++", "Python", "Java", "R"]
```

**pop()** function: removes an element from the list given its index, and returns the value removed

```
print(list_1.pop(4))  
>> "R"
```

**sort()** function: sort list items *in place*

```
list_3.sort()  
print(list_3)  
>> ['a', 'b', 'c', 'd', 'e']
```

**reverse()** function: reverses the order of elements *in place*

```
list_4.reverse()  
print(list_4)  
>> [9, 1, 7, 4, 5]
```

➡ reverse is not the opposite of sort

# dictionaries

Dictionaries are data structures that store information in key:value pairs. They are especially useful for saving and retrieving information from the names of their keys (not using indexes).

mutable ✓

ordered ✗\*

allow  
duplicates ✗: ✓  
value ↖  
key ↘

```
mi_dict = {"course": "TOTAL Python", "class": "Dictionaries"}
```

add new data, or modify it

```
mi_dict["resources"] = ["notes", "exercises"]
```

access to values through the name of the keys

```
print(mi_dict["resources"][1])  
>> "exercises"
```

methods to list the names of keys, values, and key:value pairs

keys() ←      ↓      ↘  
values()      items()

*\*: As of Python 3.7+, dictionaries are ordered data types, in the way that their order is maintained according to their insertion order to increase memory efficiency.*



# tuples

Tuples are data structures that store multiple elements in a single variable. They are characterized by being ordered and immutable. This feature makes them more memory efficient and damage-proof.

mutable ✗

ordered ✓

<sup>allow</sup>  
duplicates ✓

```
my_tuple = (1, "two", [3.33, "four"], (5.0, 6))
```

indexing (access to data)

```
print(my_tuple[3][0])  
>> 5.0
```

casting (data type conversion)

```
lista_1 = list(mi_tuple)  
print(lista_1)  
>> [1, "two", [3.33, "four"], (5.0, 6)]
```

*is now a mutable structure* ↗

unpacking (data extraction)

```
a, b, c, d = my_tuple  
print(c)  
>> [3.33, "four"]
```



# sets

Sets are another type of data structures. They differ from lists, tuples, and dictionaries because they are a mutable collection of immutable elements, unordered, and without repeated data.

mutable ✓

ordered ✗

<sup>allow</sup>  
duplicates ✗

```
my_set_a = {1, 2, "three"}    my_set_b = {3, "three"}
```

## methods

**add(item)** add an element to the set

```
my_set_a.add(5)
print(my_set_a)
>> {1, 2, "three", 5}
```

**clear()** remove all elements from a set

```
my_set_a.clear()
print(my_set_a)
>> set()
```

**copy()** return a copy of the set

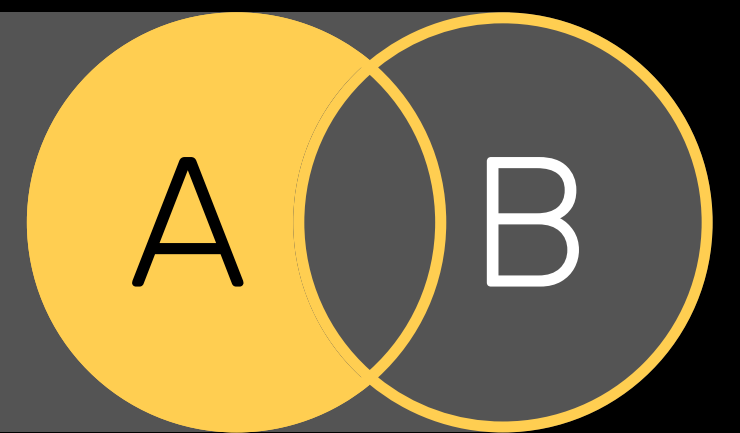
```
my_set_c = my_set_a.copy()
print(my_set_c)
>> {1, 2, "three"}
```

# sets

```
my_set_a = {1, 2, "three"}    my_set_b = {3, "three"}
```

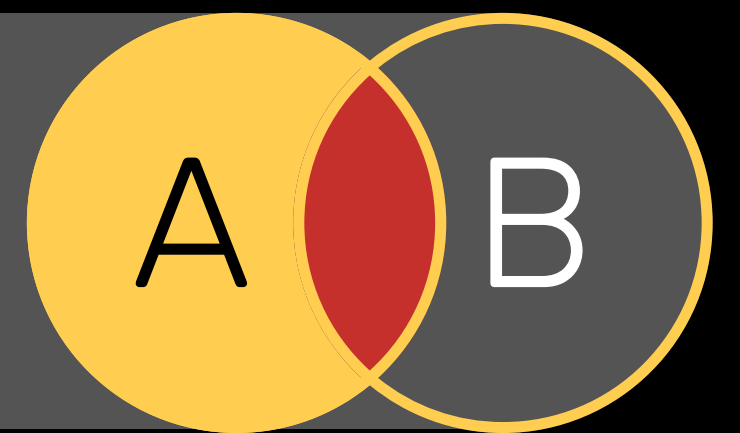
**difference(set)** returns the set formed by all the elements that **only exist in set A**

```
my_set_c = my_set_a.difference(my_set_b)
print(my_set_c)
>> {1, 2}
```



**difference\_update(set)** **removes** from A all elements **common** to A and B

```
my_set_a.difference_update(my_set_b)
print(my_set_a)
>> {1, 2}
```

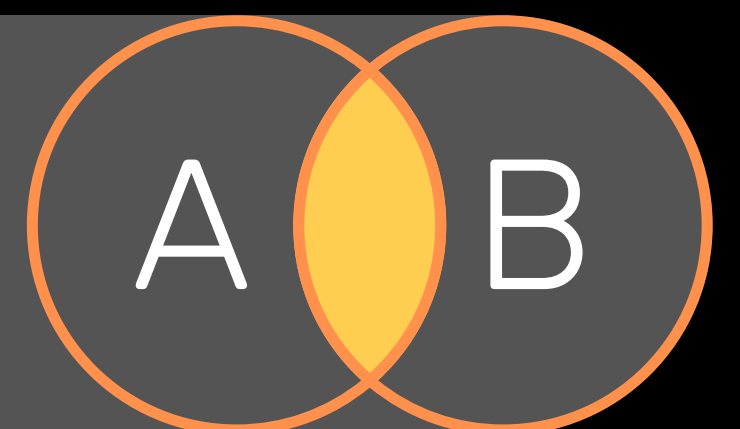


**discard(item)** **remove** an **element** from the set

```
my_set_a.discard("three")
print(my_set_a)
>> {1, 2}
```

**intersection(set)** returns the set formed by **all the elements that exist in A and B simultaneously**.

```
my_set_c = my_set_a.intersection(my_set_b)
print(my_set_c)
>> {'three'}
```

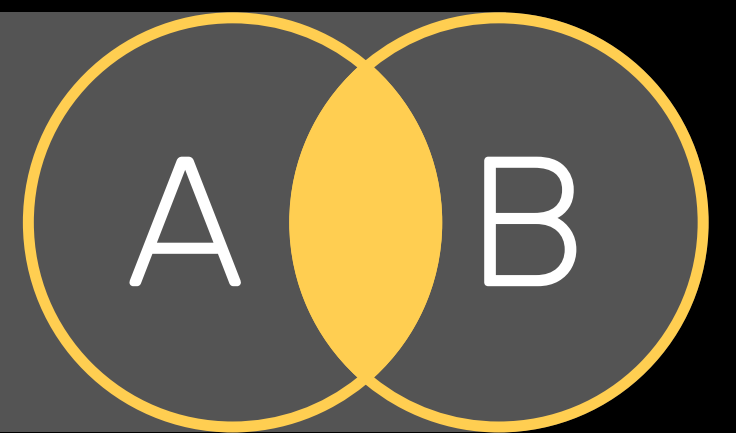


# sets

```
my_set_a = {1, 2, "three"}    my_set_b = {3, "three"}
```

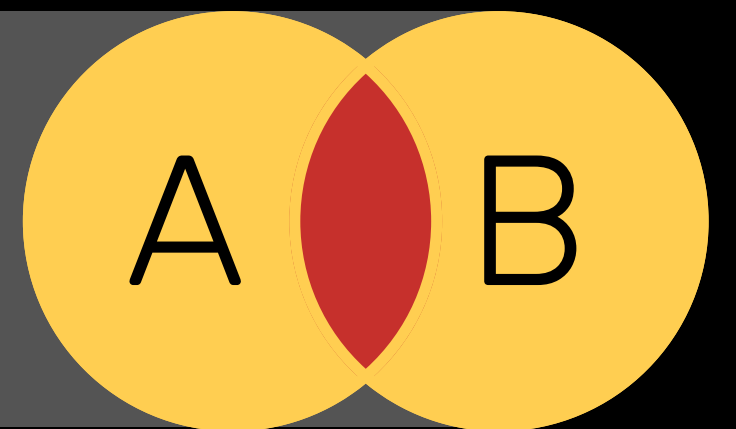
**intersection\_update(set)** keeps only the elements common to A and B

```
my_set_b.intersection_update(my_set_a)
print(my_set_b)
>> {"three"}
```



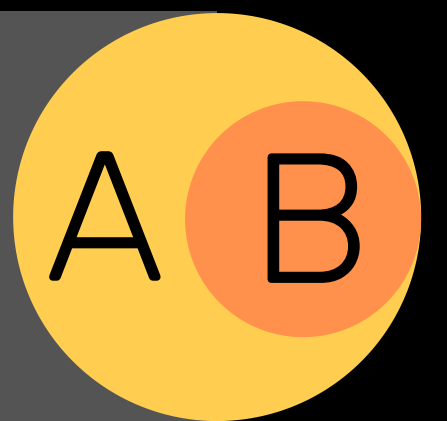
**isdisjoint(set)** returns **True** if A and B have no elements in common

```
disjoint_set = my_set_a.isdisjoint(my_set_b)
print(disjoint_set)
>> False
```



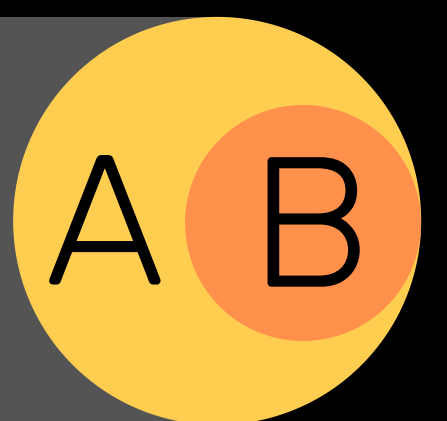
**issubset(set)** returns **True** if all elements of B are present in A

```
a_subset = my_set_b.issubset(my_set_a)
print(a_subset)
>> False
```



**issuperset(set)** returns **True** if A contains all elements of B

```
a_superset = my_set_a.issuperset(my_set_b)
print(a_superset)
>> False
```



# sets

```
my_set_a = {1, 2, "three"}    my_set_b = {3, "three"}
```

**pop()** removes and returns a random element from the set

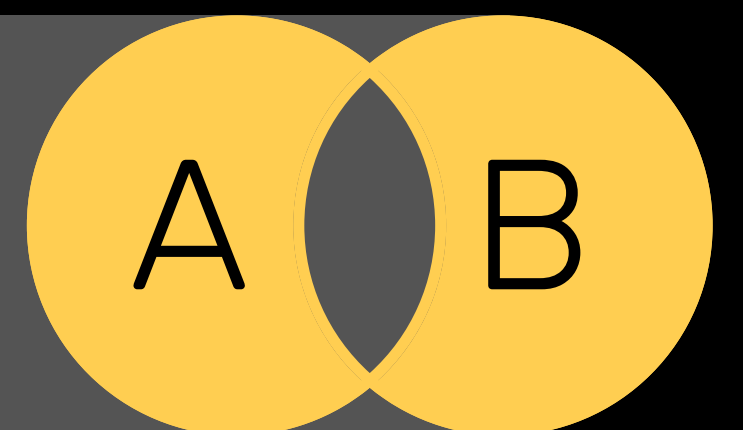
```
random = my_set_a.pop()  
print(random)  
>> {2}
```

**remove(item)** removes an item, or throws an error if it doesn't exist

```
my_set_a.remove("three")  
print(my_set_a)  
>> {1, 2}
```

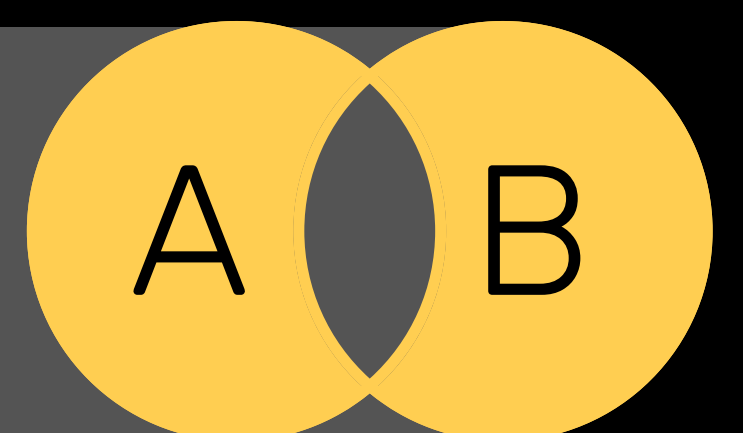
**symmetric\_difference(set)** returns all elements of A and B, except those that are common to both

```
my_set_c = my_set_b.symmetric_difference(my_set_a)  
print(my_set_c)  
>> {1, 2, 3}
```



**symmetric\_difference\_update(set)** removes the elements common to A and B, keeping those that are not present in both at the same time

```
my_set_b.symmetric_difference_update(my_set_a)  
print(my_set_b)  
>> {1, 2, 3}
```

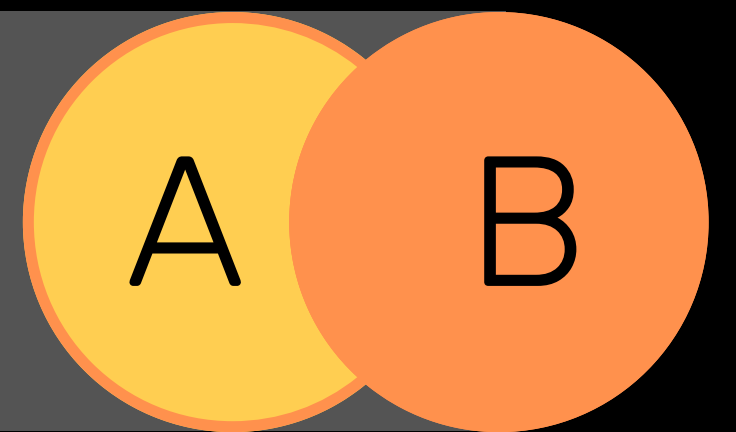


# sets

```
my_set_a = {1, 2, "three"}    my_set_b = {3, "three"}
```

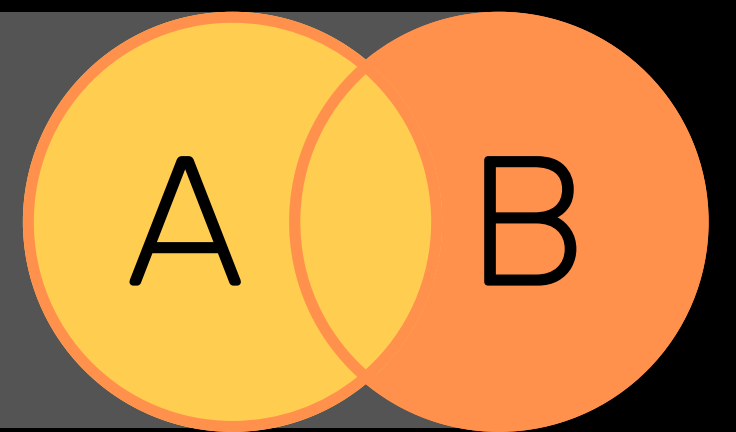
**union(set)** returns a set result of combining A and B (duplicates are removed)

```
my_set_c = my_set_a.union(my_set_b)
print(my_set_c)
>> {1, 2, 3, 'three'}
```



**update(set)** insert into A the elements of B

```
my_set_a.update(my_set_b)
print(my_set_a)
>> {1, 2, 3, 'three'}
```





# booleans

Booleans are binary data types ([True/False](#)), which result from logical operations, or can be declared explicitly.

## logical operators

```
== equal to
!= different to
> greater than
< smaller than
>= greater than or equal to
<= less than or equal to
```

```
and - True if two statements are True
or - True if at least one statement is True
not - inverts the value of the boolean
```



# Day 3 Python Challenge

**Well, the third day was full of new stuff.** We have seen a lot of good things and it's time to put everything together into practice by creating a perfectly functional program from scratch.

Now that you know how to use string method and properties, how to index data sets such as strings, lists and tuples, and especially now that you know all the data types needed to store anything, you will be able to find a way to program a text parser.

**The task is as follows: You are going to create a program that first asks the user to enter text. It can be any text, an entire article, a paragraph, a sentence, a poem, whatever you want. Then the program will ask the user to enter three random letters. From that moment on, our code is going to process that information and result in five different types of analysis:**

1. **How many times each of those letters they have chosen appears. To achieve this, I advise you to store those letters in a list, and then use a method of strings that allows you to count how many times a substring appears within the string. One thing to keep in mind is that when searching for letters, there can be upper and lower case and this will affect the result. So, to make sure that absolutely all letters are counted, you should pass both the original text and the letters to be searched to lowercase.**
2. **How many words are in the whole text? To achieve this part, remember that there is a string method that allows us to transform it into a list. And then there is a function that allows us to find out the length of a list.**
3. **What are the first and last letters of the text? Here, we will clearly use indexing.**
4. **The system will show us how the text would look like if we inverted the order of the words. Is there any method that allows us to invert the order of a list? And another one that allows us to join these elements with spaces in between?**
5. **The system will tell us if the word “python” is inside the text. This part can be a bit complicated to imagine, but I'll give you a hint: you can use Booleans to make your enquiry and a dictionary to find ways to express your answer.**

As I always tell you, I don't expect you to know how to do it on your first try. And it's likely that even though you already have the necessary tools, you still might not be able to figure it out on your own how to combine all of them to achieve our goal. But don't worry: the only thing we ask as your teachers is that you try. So go on, find a good set of music to listen to and start programming.

# comparison operators

As their name suggests, they are used to compare two or more values. The result of this comparison is a boolean (True/False)

<code>==</code>	equal
<code>!=</code>	not equal
<code>&gt;</code>	greater than
<code>&lt;</code>	less than
<code>&gt;=</code>	greater than or equal to
<code>&lt;=</code>	less than or equal to

```
my_bool = 5 >= 6  
print(my_bool)  
>> False
```

```
my_bool = 5 != 6  
print(my_bool)  
>> True
```

```
my_bool = 10 == 2*5  
print(my_bool)  
>> True
```

```
my_bool = 5 < 6  
print(my_bool)  
>> True
```

# logical operators

These operators allow decisions to be made based on multiple conditions.

```
a = 6 > 5
```

```
b = 30 == 15*3
```

**and** returns **True** if *all conditions* are true

```
my_bool = a and b  
print(my_bool)  
>> False
```

**or** returns **True** if *at least one condition* is true

```
my_bool = a or b  
print(my_bool)  
>> True
```

**not** reverse the result, returns **False** if the result is true

```
my_bool = not a  
print(my_bool)  
>> False
```

# decision making

Flow control determines the order in which the code of a program is executed. In Python, the flow is controlled by conditional structures, loops, and functions.

## conditional structures (if statement)

*Boolean expression  
(True/False)*

*The colon (:) leads to the code that is  
executed if expression = True*

*indentation is  
mandatory in  
Python*

```
if expression:
    some code
elif expression:
    some code
elif expression:
    some code
...
else:
    some code
```

*else and elif are  
optional*

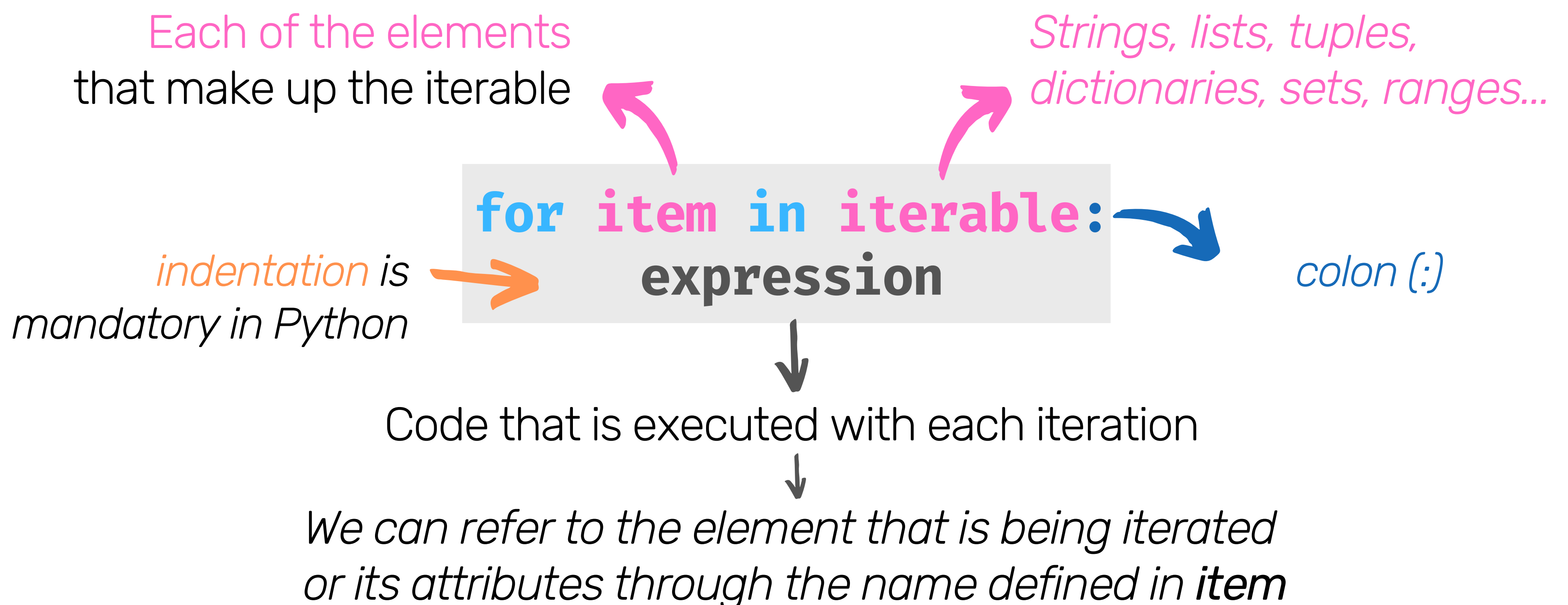
*multiple elif clauses  
can be included*

# for loops

Unlike other programming languages, for loops in Python have the ability to iterate through the elements of any sequence (lists, strings, etc.), in the order in which those elements are stored.

## Concepts

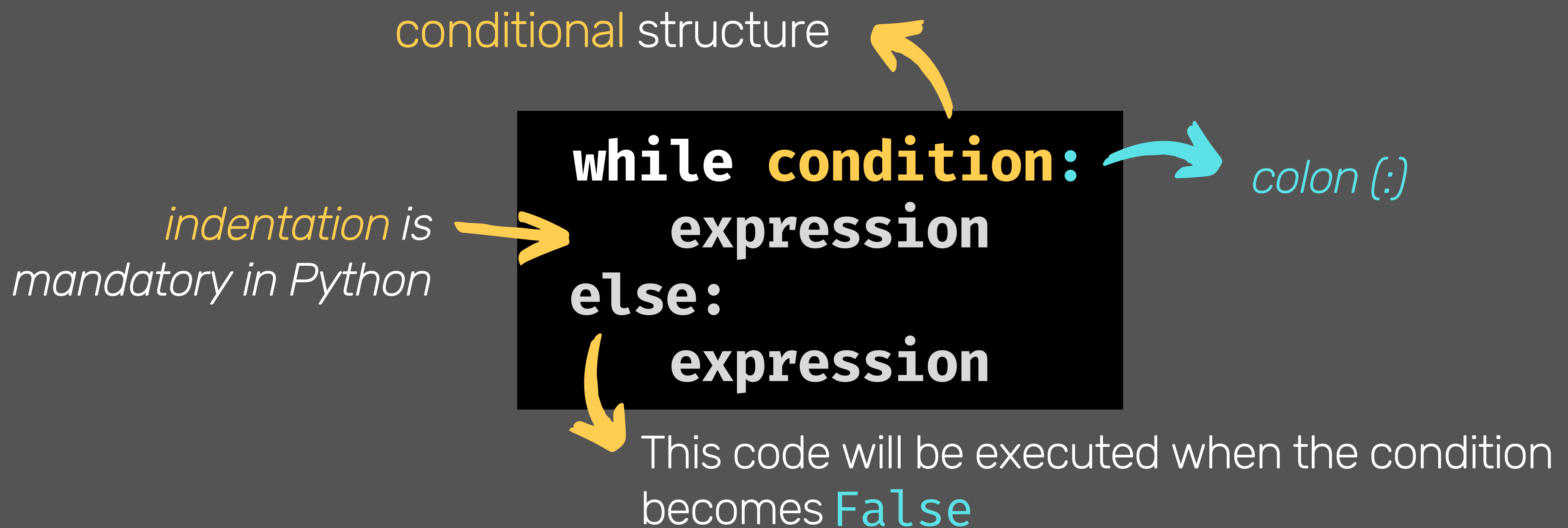
- **Loops:** code sequences that are executed repeatedly
- **Iterable:** object capable of returning its members one at a time, permitting it to be iterated over





# while loops

**While loops** are another type of loop, but they have more similarities with **if conditionals**. We can think of while loops as a conditional structure that executes on repeat, until it returns false.



## special instructions

If the code hits a **break** statement, the loop **exits**.

The **continue** statement **interrupts the current iteration** within the loop, bringing the program to the **top of the loop**.

The **pass** statement **does not alter the program**: it's a placeholder for when a statement is expected, but no action is desired.

# range()

The range( ) function returns a sequence of numbers given 3 parameters. It is mainly used to control the number of iterations in a for loop.

An integer number specifying at which position to start (included)

increment

`range(start, stop, step)`

An integer number specifying at which position to stop (not included)

```
print(list(range(1,13,3)))  
>> [1,4,7,10]
```

*The only **required** parameter is **stop**.*

*The default values for start and step are 0 and 1 respectively.*


# enumerate()

The `enumerate()` function makes it easy for us to keep track of the iterations, through an iterable index counter, which can be used directly in a loop, or converted to a list of tuples with the `list()` method.

any object that can be iterated over



```
enumerate(iterable, start)
```



defining the start number (int) of the enumerate object  
(default is 0)

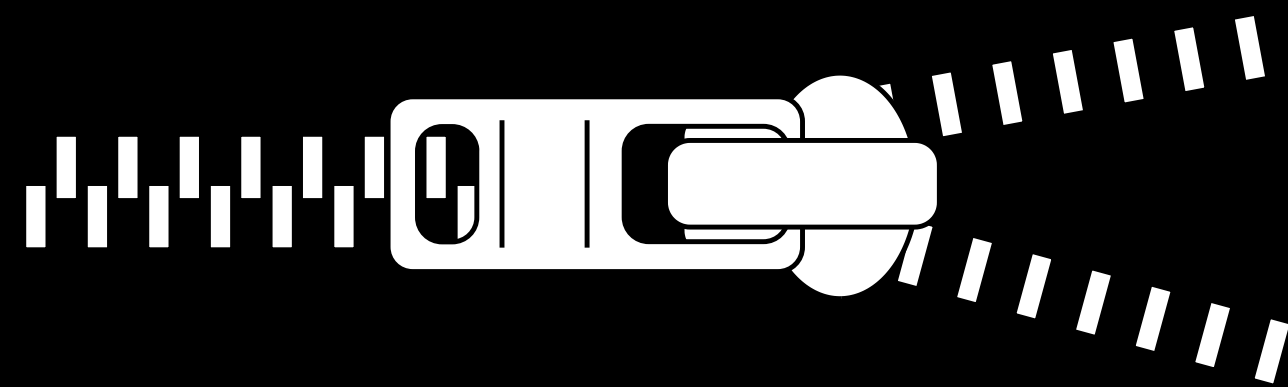
```
print(list(enumerate("Hey!")))  
>> [(0, 'H'), (1, 'e'), (2, 'y'), (3, '!')]
```

```
for i, number in enumerate([5.55, 6, 7.50]):  
    print(i, number)
```

```
>> 0 5.55  
>> 1 6  
>> 2 7.5
```

# zip()

The `zip()` function acts just like a *zipper*, by creating an iterator formed of the grouped elements of the same index from two or more iterables.



The function stops when the iterable with the fewest elements is exhausted.

```
letters = ['w', 'x', 'c']
numbers = [50, 65, 90, 110, 135]
for letter, num in zip(letters, numbers):
    print(f'Letter: {letter}, and number: {num}')
```

```
>> Letter: w, and number: 50
>> Letter: x, and number: 65
>> Letter: c, and number: 90
```

# min() and max()

The min() function returns the item with the lowest value within an iterable. The max() function works in the opposite way, returning the highest value of the iterable. If the iterable contains strings, the comparison is done alphabetically.

```
cities_population = {"Albuquerque":559121, "Tulsa":403505}
```

```
my_list = [5**5, 12**2, 3050, 475*2]
```

```
print(min(cities_population.keys()))  
>> Albuquerque
```

```
print(max(cities_population.values()))  
>> 559121
```

```
print(max(my_list))  
>> 3125
```



# random

Python provides us with a module (a set of functions available for use) that allows us to generate pseudo-random\* selections from values or sequences.

Module name

**from random import \***

\* = all methods

*can also be independently imported*

**randint**(*min, max*): returns an integer between two given values (both limits included)

**uniform**(*min, max*): returns a float between a minimum and maximum value

**random**(): returns a float between 0 and 1

**choice**(sequence): returns a random element from a sequence of values (lists, tuples, ranges, etc.)

**shuffle**(sequence): takes a mutable sequence of values (like a list), and returns it by randomly changing the order of its elements.

*\*The mechanics in how these random values are generated is actually predefined in "seeds". While it serves most common uses, it should not be used for security or cryptographic purposes as they can be vulnerable.*

# list comprehension

List comprehension offers a shorter syntax in creating a new list based on values available in another sequence. It is worth mentioning that brevity comes at the cost of less interpretability.

*each element of the  
iterable*

*tuples, sets, other lists...*

```
new_list= [expression for item in iterable if condition == True]
```

*e.g.: mathematic expression*

*logical operation*

Special case with else:

```
new_list= [expression if condition == True else other_expression  
           for item in iterable]
```


Example:

```
new_list = [num**2 for num in range(10) if num < 5]  
print(new_list)  
>> [0, 1, 4, 9, 16]
```

# match

In Python 3.10, structural pattern matching is introduced using the `match` and `case` statements. This allows us to associate specific actions based on the patterns of complex data types.

```
match object:  
    case <pattern_1>:  
        <action_1>  
    case <pattern_2>:  
        <action_2>  
    case <pattern_3>:  
        <action_3>  
    case _:  
        <wildcard_action>
```



*The `_` character is a wildcard that acts as a match if it does not occur in the previous cases.*

It is possible to detect and deconstruct different data structures: this means that patterns are not only literal values (strings or numbers), but also data structures, on which construction matches are sought.

# Day 4 Python Challenge

As we learn to program the number of problems we can solve with code also increases. This implies that our challenges are going to be harder to solve. You already know how to use comparison and logical operators, how to control flow and loops. You know super useful statements such as random, zip, range and more...

You are now ready to go up a level and I will ask you to program something a little more complicated than what we have done so far. Today, you are going to create for the first time a fully functional game, with which you can even have some fun yourself.

**The task is this: the program will ask for the user's name and then it will say something like: “*Well John, I've thought of a number between 1 and 100 and you have only eight tries to guess it. What number do you think it is?*”** On each try, the player will say a number and the program can answer for different things.

- If the number the user said is less than 1 or greater than 100, it will tell them that he/she has chosen a number that is out of play.
- If the number the user chose is less than the number the program thought of, it will tell them that the answer is wrong, and that he/she chose a lower number than the secret number.
- If the user chose a number greater than the secret number, it will let them know that it was greater.
- And if the user got the secret number right, they will be informed that they have won, and how many tries that has taken them.
- If the user has not guessed correctly in their first attempt, they will be asked again to choose another number and so on until they win or until their eight attempts are done.

So, are you up for it? Of course you are.

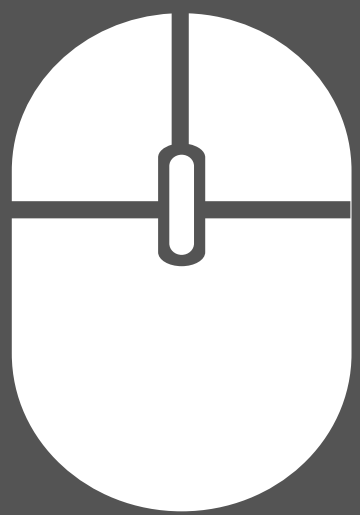
I remind you the same as always: the most important thing is that you accept this challenge, that you have a good time trying. Try to figure out each step. You know that from what you've learned in this course so far. You have all the knowledge, it's somewhere in there. Maybe you have to go back, rewatch a lesson, relook at some of the things that we've programmed so far, but know that you are able to do this and no matter what, even if you get stuck, try spend some time with it and then we'll follow up, and Federico will show you exactly how to do it.

So on to some good music. Let's get over to program.

# documentation

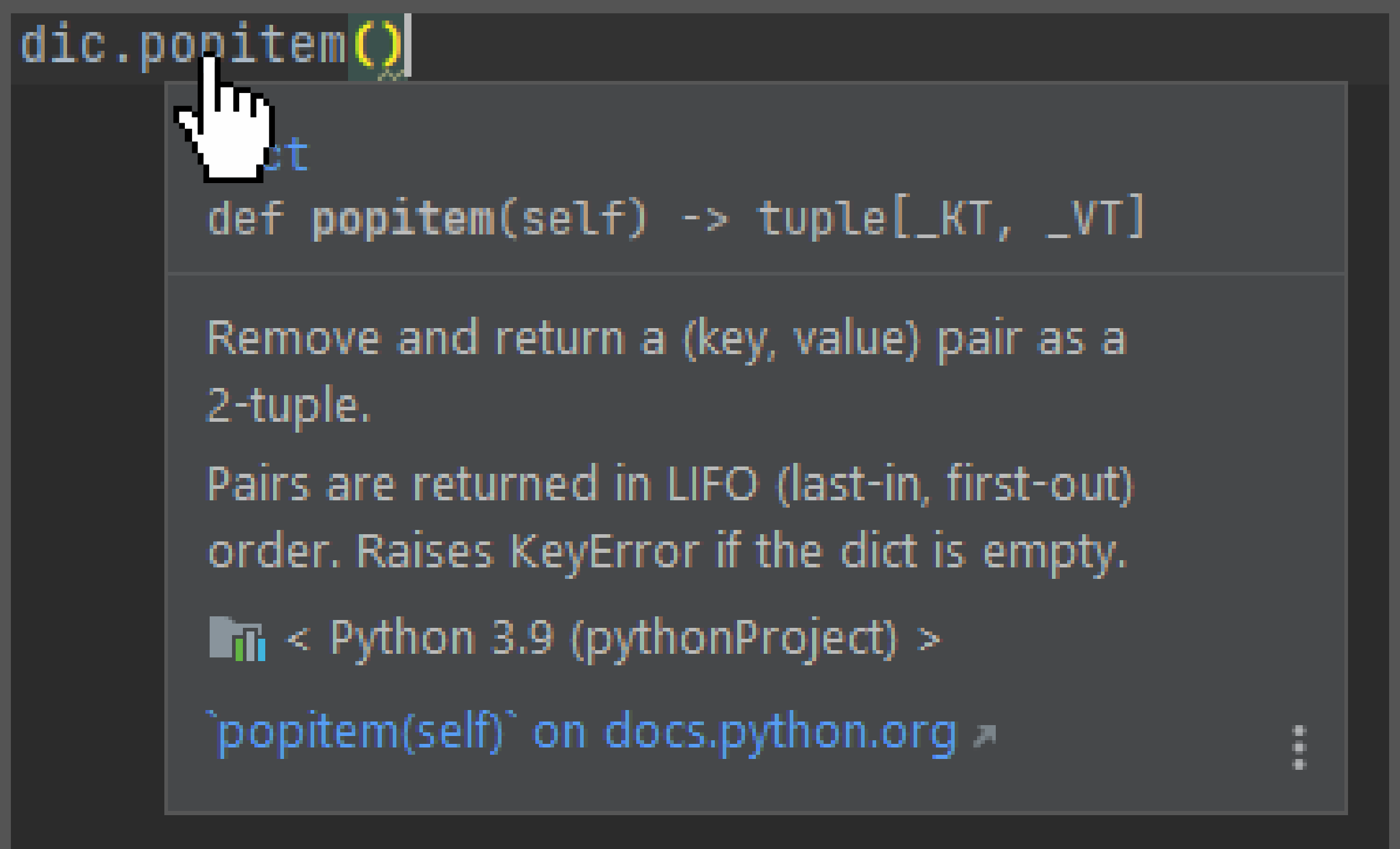
Consider Python documentation as our reference library. When writing code, it is of permanent use to solve doubts related to the operation of methods and the arguments they receive.

If you use **PyCharm**:



Hover over the name of the method you want to get information about. A floating window will be displayed.

This will work in other IDEs the same way.



You should keep close the **official Python documentation** (or Python Standard Library), which contains all the necessary information:

<https://docs.python.org/3.10/library/index.html>

*You can search your doubts on the web, to find an explanation that suits you.*



# functions

A function is a block of code that is only run when it is called. It can receive data (known as parameters), and return it once processed as a result.

a function is defined by the  
keyword **def**



```
def my_function(parameter):
```

The diagram illustrates the relationship between function definition and function call. A blue arrow points from the `def` keyword in the definition to the explanatory text above. Another blue arrow points from the `parameter` in the definition to the explanatory text below. A third blue arrow points from the `my_argument` in the function call to the explanatory text at the bottom.

The parameters contain information that the function uses or transforms to return a result.

```
my_function(my_argument)
```

We can call a function by its name, giving the arguments that it requires inside parentheses.

# return

In order for a function to return a value (and for it to be stored in a variable, for example), we use the **return** statement.

```
def my_function():  
    return [something]
```



The return statement causes the function to exit.  
*Any code after it inside the function block will not be executed.*

```
result = my_function()
```



The **result** variable will store the value returned by the function **my\_function()**

# dynamic functions

The integration of different flow control tools allows us to create dynamic and flexible functions. If we must use a piece of code several times, we can create a cleaner and easier program to maintain with functions, avoiding code repetitions.

- Functions
- Loops (for/while)
- Conditional structures
- Keywords (return, break, continue, pass)

```
def my_function(argument):  
    for item in ...  
        if a == b ...  
            ...  
        else:  
            return ...  
    return ...
```

# interaction between functions

The outputs of a certain function can become inputs of other functions. That way, the program is built from the interaction between functions that perform a defined task .

```
def function_1():  
    ...  
    return a  
  
def function_2(a):  
    ...  
    return b  
  
def function_3(b):  
    ...  
    return c  
  
def function_4(a, c):  
    ...  
    return d
```

# \*args

In cases where the exact number of arguments to pass to a function is not known in advance, the special `*args` syntax should be used to refer to all additional arguments after the required ones.

*The name `*args` is not mandatory but is suggested as a good practice. Any names starting with `*` will refer to these "unknown" arguments.*

The function will receive the undefined arguments `*args` in the form of a tuple, which can be accessed or iterated over in the usual ways within the function's code block.

```
def my_function(arg_1, arg_2, *args):
```

```
my_function("example", 25, 40, 75, 10):
```

arg\_1

arg\_2

args = (40, 75, 10)



# **\*\*kwargs**

Keyword arguments (kwargs for short) are used to identify the argument by name, regardless of the position in which they are passed to your function. If their quantity is not known in advance, the **\*\*kwargs** parameter is used, which groups them in a dictionary.

*As for **\*args**, the name **\*\*kwargs** is not mandatory but is suggested as good practice. Any name starting with **\*\*** will refer to these variable quantity arguments.*

```
def person_attributes(**kwargs):
```

```
    person_attributes(eyes="blue", hair="short")
```

**kwargs = {'eyes': 'blue', 'hair': 'short'}**

# Exercise 1

Create a function called `return_distincts()` that receives 3 integers as parameters.

If the sum of the 3 numbers is greater than 15, it must return the highest number.

If the sum of the 3 numbers is less than 10, it must return the lowest number.

If the sum of the 3 numbers is a value between 10 and 15 (included), then it must return the number with the intermediate value.

# Exercise 2

Write a function (you can name it whatever you want) that takes any word as a parameter, and returns all of its unique letters (without repetition) in alphabetical order.

For example, if when calling this function we pass the word "entertaining", it should return ['a', 'e', 'g', 'i', 'n', 'r', 't']

# Exercise 3

Write a function that requires an indefinite number of arguments. What this function must do is return `True` if at any time the number zero has been entered twice consecutively.

For example:

```
(5,6,1,0,0,9,3,5) >>> True
```

```
(6,0,5,1,0,3,0,1) >>> False
```

# Exercise 4

Write a function called `count_primes()` that requires a single numeric argument.

This function must display on the screen how many prime numbers there are in the range from zero to that number included, and then return the number of prime numbers found.

*Note: By convention 0 and 1 are not considered primes*

.



# Day 5 Python Challenge

**We've finished preparing for this fifth challenge of your fifth day.** Now we are going to put it all together, all hands on deck, because we have a very special challenge coming to you.

**Today you are going to program the hangman game.** It's simple, popular, but if you don't know it, let me explain it really quickly: **The program will choose a secret word and we'll show the player only a series of dashes that represent the number of letters in the word. In each turn, the player must choose a letter: if that letter is in the hidden word, the system will show where it is located, but if the player chooses a letter that is not in the hidden word, they lose a life.**

In the real hangman game, each time we lose a life, the drawing of the hangman is completed limb by limb. But in our case, as we still do not have the graphic elements, **we will simply tell the user that they have six lives, and we will deduct them one by one for each time the player chooses an incorrect letter.**

**If the player runs out of lives before guessing the word, the player loses. But if they guessed the whole word before losing all their lives, the player wins.**

Sounds simple, but how do we design all this in code? Here are some clues:

1. First, you are going to create code that imports the `choice()` method, since you are going to need it so that the system can **choose a random word** from a list of words that you are also going to create at the beginning.
2. After that, you are going to **create as many functions as you think necessary** for the program to do things like: asking the user to choose a letter, checking if what the user has entered is a valid letter, checking if the entered letter is in the word or not, checking if they have won or not... Remember to write the functions first, then the code that implements those functions in an orderly fashion after.

This is a special project and I really want you to know that I don't expect you to be able to solve it without help. In fact, I will tell you that you're bound to get to a point where you get all the code tangled up and you will need some help from our solution. And if that happens, it is totally normal.

I think you might need some extra help, and that's why I've created an additional lecture with some hints and tips that might make your life a little bit easier.

All right, go at it. See you in the next lecture or in our solution video.

