CHECKPOINT 5

¿Qué es un condicional?

In Python, the **if** statement is how you perform conditionals. It allows for conditional execution of a statement or group of statements based on the value of an expression.

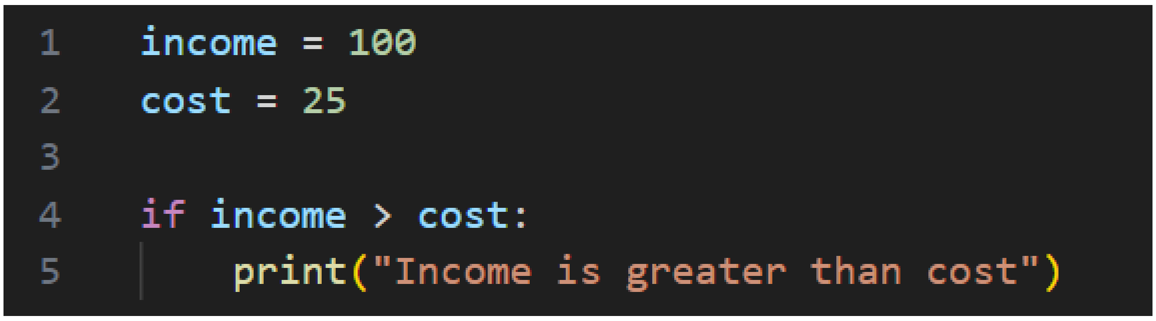
For creating conditionals, Python supports the same logical conditions from mathematics:

Equals: a == b
Not Equals: a != b
Less than: a < b
Less than or equal to: a <= b
Greater than: a > b
Greater than or equal to: a >= b

In order to create conditionals, you can use the previous operators, depending on the result you are looking for.

The syntax for creating a conditional is as the one shown in the next example.

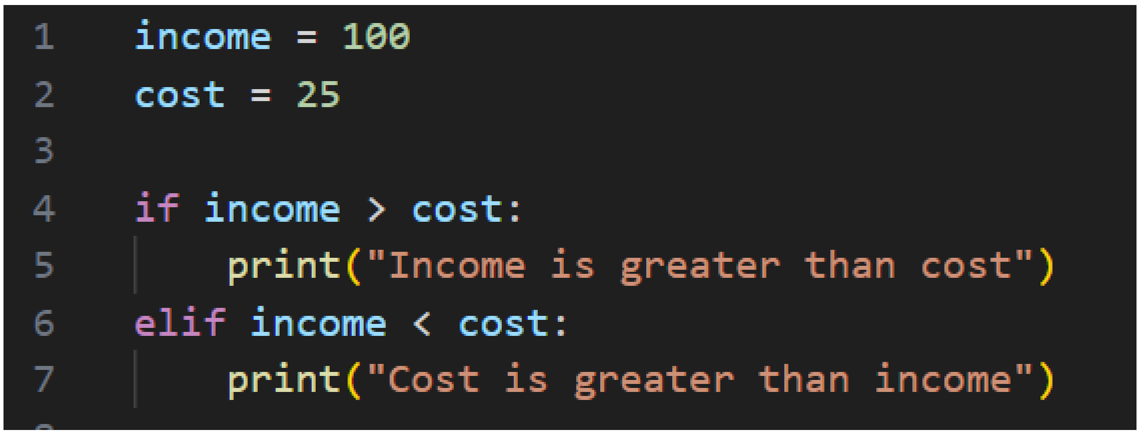
In this example we use the variables income and cost, which are going to be used as part of the if statement in order to check if the cost is greater than the income. As the income is 100 and the cost is equal to 25, we know the income is greater than the cost, so we print to screen that "Income is greater than cost".



```python
income = 100
cost = 25

if income > cost:
    print("Income is greater than cost")
```

In order to set more than one condition you can use the **elif** keyword. The elif keyword is a way of saying "if the previous conditions were not true, then try this condition".

In our example, we can say that, if the first condition is not true, then try to check if income is lower than cost and, in that case, print "Cost is greater than income".

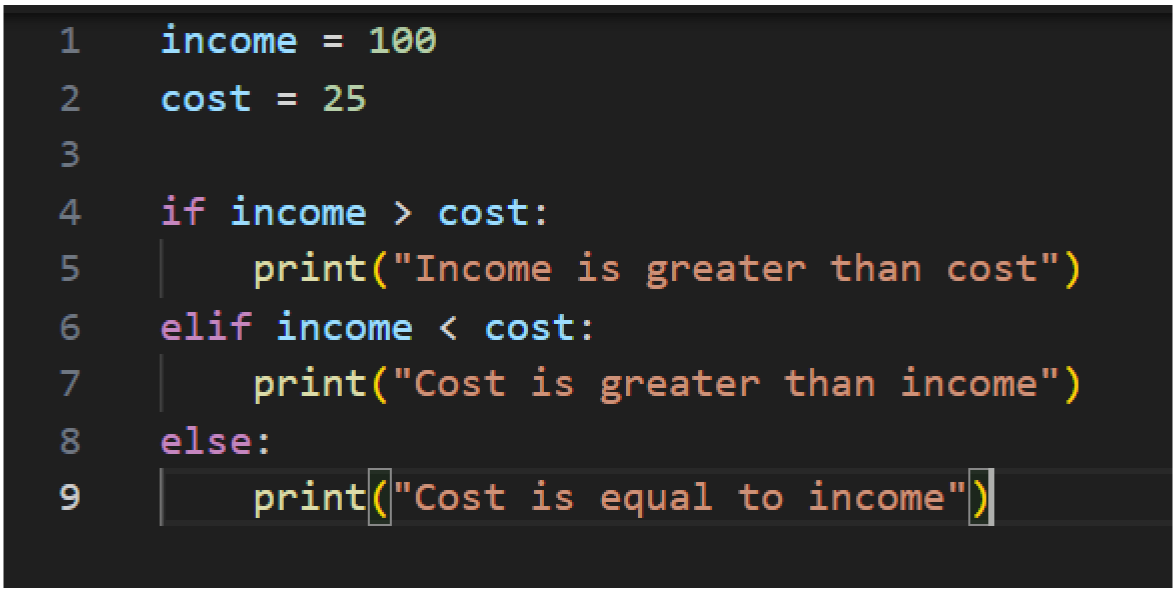


```python
income = 100
cost = 25

if income > cost:
    print("Income is greater than cost")
elif income < cost:
    print("Cost is greater than income")
```

For any of the results not gathered in the set conditions, you can use the **else** keyword. This way you can set a “backup” for any of the cases not contemplated in the conditions.

In our example, we are going to use the else keyword in order to consider a situation when the income is equal to the cost.

Here’s how our example looks with all the conditions set:



```python
income = 100
cost = 25

if income > cost:
    print("Income is greater than cost")
elif income < cost:
    print("Cost is greater than income")
else:
    print("Cost is equal to income")
```

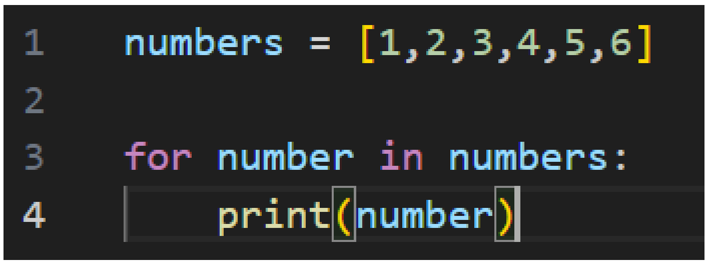
¿Cuáles son los diferentes tipos de bucles en Python? ¿Por qué son útiles?

In this section we are going to go through the two loop types in Python and see how they are different from each other. The two types of loops are for loops and the while loops.

For loop

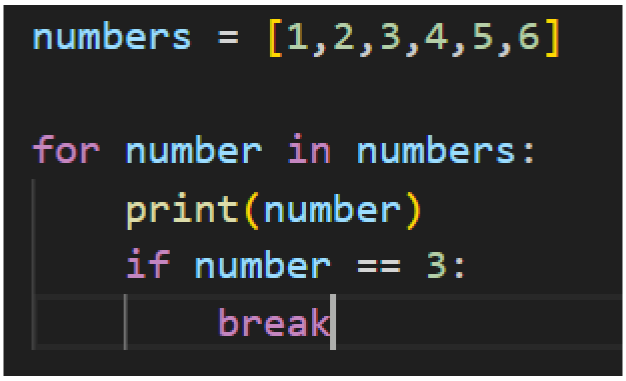
A for loop is used for iterating over an iterable object (a list, a tuple, a dictionary, a set or a string). With the for loop you can execute a set of statements, once for each item in the iterable object.

The syntax for creating a for loop is as follows in the next example, where we are going to loop through the numbers in the numbers list:



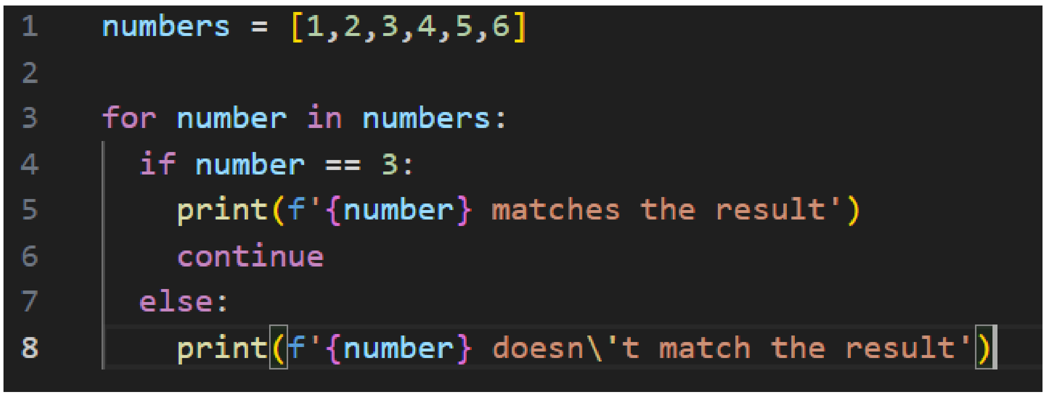
```python
numbers = [1,2,3,4,5,6]

for number in numbers:
    print(number)
```

With the **break** statement you can stop the loop before it has looped through all the items:



```python
numbers = [1,2,3,4,5,6]

for number in numbers:
    print(number)
    if number == 3:
        break
```

With the **continue** statement you can stop the current iteration of the loop, and continue with the next. What continue does is, it tells the program to keep on going through the loop, so it's going to continue to iterate and it is not going to stop once it finds what it's looking for.

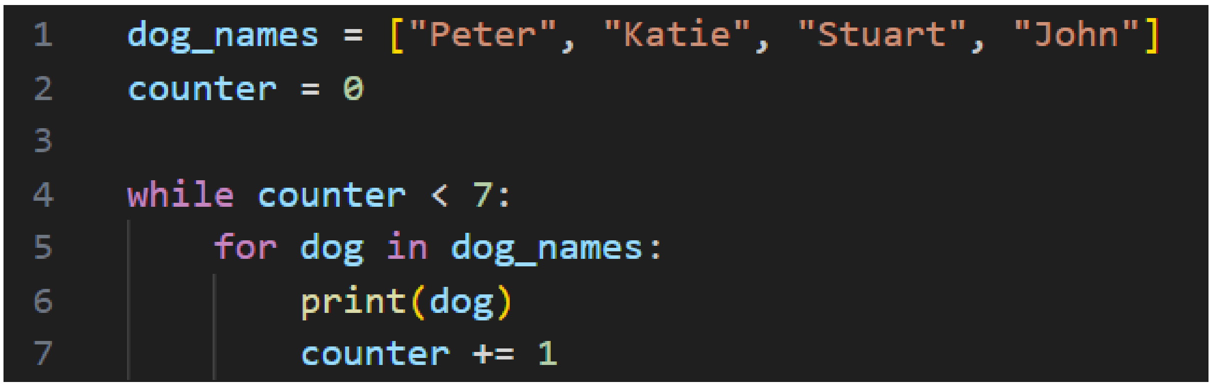


```python
numbers = [1,2,3,4,5,6]

for number in numbers:
  if number == 3:
    print(f'{number} matches the result')
    continue
  else:
    print(f'{number} doesn\'t match the result')
```

<u>**While loop**</u>

With the while loop we can execute a set of statements as long as a condition is true. The loop will not stop when it reaches the end of the object you're iterating over. A while loop has to explicitly be told when to stop. For that, you have to set a sentinel value in order to tell your while loop when to stop.

In the next example whe are going to create a while loop that is going to keep iterating through the dog_names list until the counter reaches 7, starting from 0 in the first iteration:

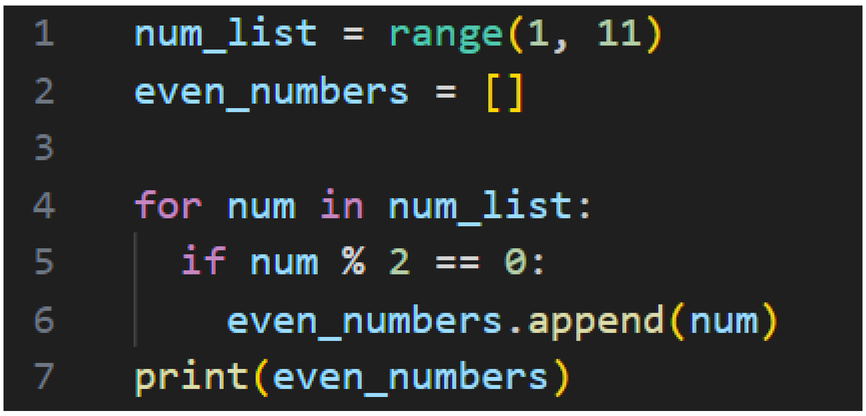


```python
dog_names = ["Peter", "Katie", "Stuart", "John"]
counter = 0

while counter < 7:
    for dog in dog_names:
        print(dog)
        counter += 1
```

¿Qué es una lista por comprensión en Python?

What list comprehension allows us to do is to set up a number of for loops to function on a single line and we can actually generate lists from those lines of code.

For example, if you want to create a list of even number contained in a given range, by using a for loop, you have to use a few lines of code in order to perform that as shown below:



```python
num_list = range(1, 11)
even_numbers = []

for num in num_list:
  if num % 2 == 0:
    even_numbers.append(num)
print(even_numbers)
```

By using list comprehension, and following the next syntax, you can get the same result in a single code line:

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

```python
even_numbers = [num for num in num_list if num % 2 == 0]
```

¿Qué es un argumento en Python?

Information can be passed into functions as **arguments**. Arguments are specified after the function name, inside the parentheses. You can add as many arguments to a function as you want by separating them with a comma.

For cases where no arguments are provided, you can set a **default argument**, but whenever you do pass an argument, that overrides the default argument.

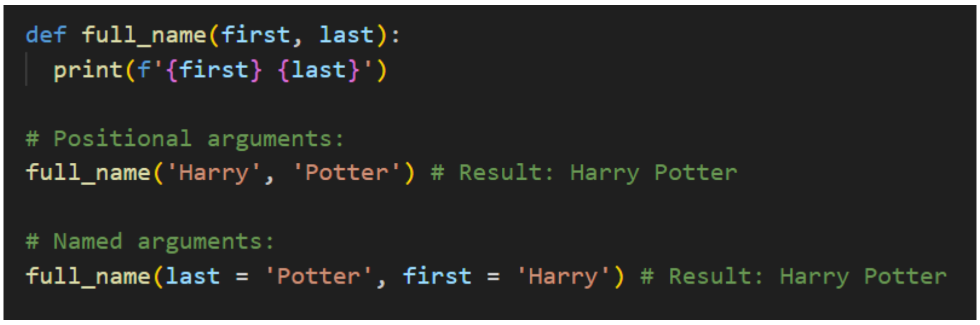
In the next example, we are going to set professor as default value for the name argument. In case no value is passed for the name argument, professor is going to be set as default value in the greeting, but if we pass Richard, the default argument is going to be overrode.

```python
def greeting(name = 'Professor'):
  print(f'Hi {name}!')

greeting() # Result: Hi Professor!
greeting('Richard') # Result: Hi Richard!
```

The arguments used here are called **positional arguments**, since the mapping between the value and how that value is used in the function is completely determined by the position and the order that we passed the values in.

When it comes to working with larger programs and more advanced functionality having positional arguments can lead to some confusion. In order to avoid errors, **named arguments** give you the ability to be much more explicit with this mapping.

In the following example we are going to see the difference between a function with positional arguments and the same function with named arguments:



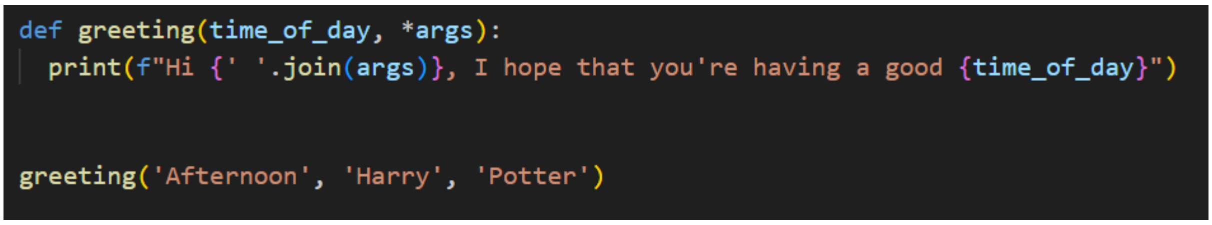
```python
def full_name(first, last):
    print(f'{first} {last}')

# Positional arguments:
full_name('Harry', 'Potter') # Result: Harry Potter

# Named arguments:
full_name(last = 'Potter', first = 'Harry') # Result: Harry Potter
```

With named arguments, contrary to positional arguments, where you need to set the argument in the same order as the result you want to obtain, you don't need to follow the same order since you are assigning a name to an argument and you can refer to it by its name inside a function.

There will be times where you have a function that needs to take in a collection of data and where you are going to need more than two arguments in a function. In those cases you can implement **argument unpacking**.

The syntax for using unpacking is to start with a star and then the common convention is to name the argument list args **(*args)**. What this represents is an unpacked version or a list of items that are going to be passed into the function.

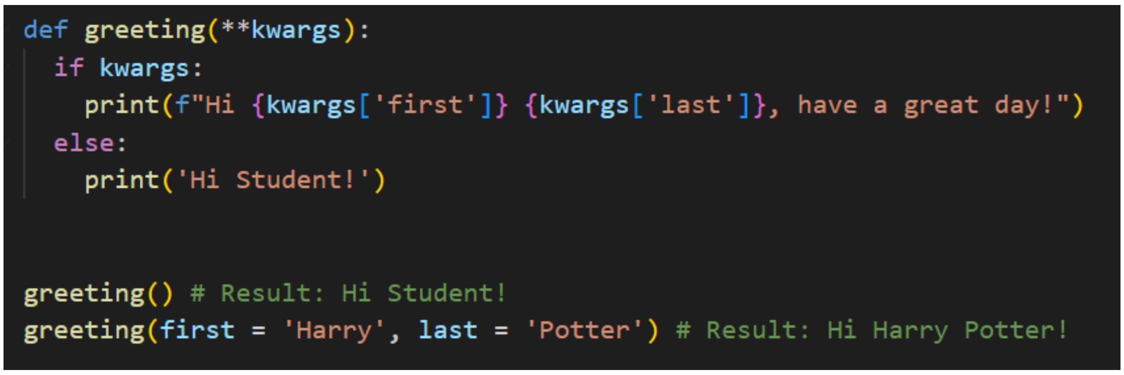
You can still pass more than one argument when using unpacking as you can see in the next example, by using unpacking in combination with a named argument:



```python
def greeting(time_of_day, *args):
    print(f"Hi {' '.join(args)}, I hope that you're having a good {time_of_day}")


greeting('Afternoon', 'Harry', 'Potter')
```

Up until now, we have talked about unpacking where we worked with *args and we've talked about named arguments. Python has a way where you can combine both, so you can take in a **keyword** list of **arguments**. This is used when you may not know how many arguments to take, very similar to what we have seen with unpacking. The common convention is to name is as **(**kwargs)**.

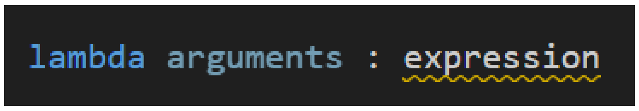
In the next example we are going to see how we can set a default value by using a conditionals when using a keyword list of arguments, in case no arguments are given.



```python
def greeting(**kwargs):
  if kwargs:
    print(f"Hi {kwargs['first']} {kwargs['last']}, have a great day!")
  else:
    print('Hi Student!')


greeting() # Result: Hi Student!
greeting(first = 'Harry', last = 'Potter') # Result: Hi Harry Potter!
```

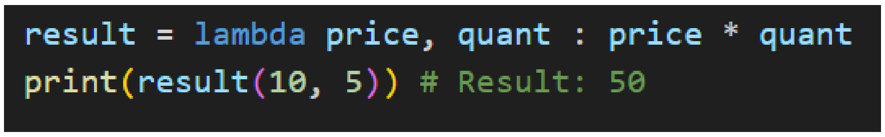
¿Qué es una función Lambda en Python?

Lambda is a tool that allows you to wrap up a function. A lambda function is a small anonymous function that can take any number of arguments, but can only have one expression.



```python
lambda arguments : expression
```

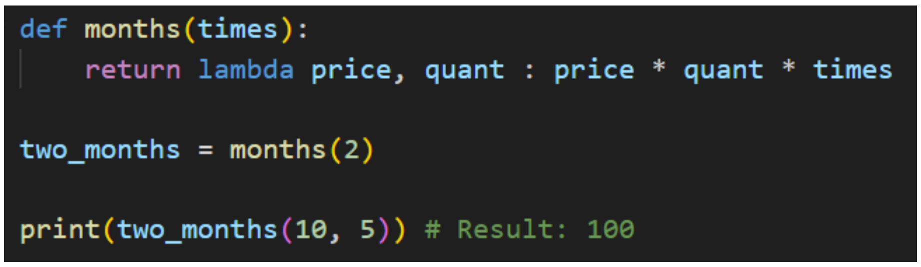
The expression is executed and the result is returned as we can see in the following example:



```python
result = lambda price, quant : price * quant
print(result(10, 5)) # Result: 50
```

The power of lambda functions is better shown when using them as an anonymous function inside another function. When we're working with lambda functions, what we can do is to wrap up some behavior, usually small behavior, and then pass it to other functions, so they are very mobile and easy to use.

In the next example, we are going to see how by having a function definition that takes one argument, we are going to double the result from the lambda function:



```python
def months(times):
    return lambda price, quant : price * quant * times

two_months = months(2)

print(two_months(10, 5)) # Result: 100
```

¿Qué es un paquete pip?

PIP is a package manager for Python packages. A package contains all the files you need for a module. Modules are Python code libraries you can include in your projects.

Once you have PIP installed in your computer, you can download and install any package you want. Once the package you want to use in your project is installed, you can start using it.

When you want to use a library you have installed in the project where you are working, you have to import it. You can assign an alias inside the project where you are working to the libraries you import so you can refer to them anytime you want to use them by their alias.

In the next example you can see how you can import the Numpy library to your project, by aliasing it as np, so you can refer to the library whenever you want to use it in the project as np:

```
import numpy as np

num_range = np.arange(10)

num_range # Result: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If you are only going to work with a specific function from a package, you can import that individual function instead of having to import the full package into your project.

Here you can see how you can import the sqrt function from the math module:

```
from math import sqrt

sqrt(25) # Result: 5.0
```