CHECHPOINT 5 - PYTHON TUTORIAL

1. ¿Qué es un condicional?
2. ¿Cuáles son los diferentes tipos de bucles en python? ¿Por qué son útiles?
3. ¿Qué es una comprensión de listas en python?
4. ¿Qué es un argumento en Python?
5. ¿Qué es una función de Python Lambda?
6. ¿Qué es un paquete pip?

1. ¿Qué es un condicional?

1.1. Introducción

Un condicional permite introducir todo tipo de condiciones de forma que la ejecución sea dinámica ya que así permitimos al sistema tomar decisiones.

Es decir, sin los condicionales lo que programaramos sería lineal y no habría opciones de tomar caminos diferentes en función de un suceso o evento creado por el usuario.

1.2. Condicional de dos condiciones

El ejemplo que todos conococemos y nos resulta cercano es el de autentificación de usuarios: Imaginemos que solicitimos a un usuario un nombre de usuario y una contraseña. El usuario nos devuelve estos valores:

```
user = 'Martin'
password = 'Luther'
```

Este usuario no es el autorizado según el sistema, ya que el único autorizado tiene estas credenciales:

```
user_allowed = 'George'
password_allowed = 'Sand'
```

Entonces, el sistema deberá de comprobar si es correcto y en función de si es correcto o no, ejecutar una acción u otra.

```
if user == user_allowed and password == password_allowed:
    print('Welcome to our website')
else:
    print('I am sorry, I am afraid you are not allowed')
```

En el caso expuesto, en el supuesto donde el nombre y la contraseña son corerctas, imprimimos en la pantalla "Welcome to our website".

Sin embargo, en la parte "else:", es decir, si no se cumple la condición, entonces, imprimimos en la pantalla "I am sorry, I am afraid you are not allowed".

Esto también es posible aplicarlo en listas, tuplas, diccionarios o rangos para saber si exite un valor en concreto:

```
number = 10
if number in range(1,200):
    print("We have it!")
else:
    print("We don't have it")
```

O incluso también para saber si una palabra está contenida en una frase:

```
sentence = 'This is a quick guide to learn Python'
word = 'quick'

if word.lower() in sentence.lower():
    print('The word is in the sentence')
else:
    print('The word is not in the sentence')
```

1.3. Operadores de los condicionales

Cuando escribimos una condición, los **operadores de condición** no se corresponden exactamente a los **operadores de asignación** en lo que se refiere al operador "=".

En el caso de ser un operador de una condición, tal y como hemos visto, utilizaríamos "==".

```
if user == user_allowed and password == password_allowed:
    print('Welcome to our website')
else:
    print('I am sorry, I am afraid you are not allowed')
```

Para poder ver la diferencia, usaremos esta misma condición y almacenaremos los resultados en una variable en lugar de imprimirlos directamente.

```
user_return = None
if user == user_allowed and password == password_allowed:
    user_return='Welcome to our website'
else:
```

```
        user_return='I am sorry, I am afraid you are not allowed'
    print(user_return)
```

En este caso, como se puede observar hemos asignado el mensaje a una variable mediante el operador de asignación "=" y sin embargo para comprobar la condición hemos utilizado el operador condicional "==".

Los operadores Condicionales son los siguientes:

```
Conditional operators:
    == Equality
    != Inequality
    >  Greater than
    >= Greater than or equal to
    < Less than
    <= Less than or equal to

<> Inequality (deprecated)
```

1.4. Usar "Ternary Operator" en condicionales

Es posible simplificar una condición implementando lo que se denomina **ternary operator**. Entre los principios del Zen de Python "*simple is better than complex*" y esto responde a este concepto.

Estas dos condiciones serían la misma siendo la segunda más breve como se puede observar.

```
    name = 'Alba'
    # Ordinary conditional
    if name == 'Alba':
        response = 'You are who I spect.'
    else:
        response = 'You are not who I spect.'
    # Conditional using Ternary operator
    response = 'You are who I spect.' if name == 'admin' else 'You are not who I
spect.'
```

1.5. Condicional de mas de dos condiciones

Imaginemos ahora queremos identificar por la edad de una persona su situación laboral y devolverle un mensaje. Como podemos observar, la segunda y tercera condición se formulan mediante **"elif"**:

```
    age = 18
    if age in range(0,18):
        print('Under 18. You are too young to work.')
    elif age in range(18,66):
        print('Between 18 and Under 66: Yo can work.')
    elif age in range(66,100):
```

```
        print('You should be retired.')
    else:
        print('If you are still alive congratulations.')
```

1.6. Condicional conjuntos y anidados

Imaginemos que queremos saber cuales son los números pares e impares y dentro de los impares cuales son múltiplos de 7. Entonces, nuestra primera condición sería saber si es par o impar, y dentro de los impares, saber cuales son múltiplos de 7. Tenemos dos opciones:

- Utilizar los operadores **"and"** y/o **"or"**
- Anidar las condiciones

El primero lo hemos visto ya con el primer ejemplo aunque no nos hayamos dado cuenta:

```
    if user == user_allowed and password == password_allowed:
        print('Welcome to our website')
    else:
        print('I am sorry, I am afraid you are not allowed')
```

Ahora haremos lo mismo con el caso expuesto:

```
    num =15
    if num % 2 != 0 and num % 7 == 0:
        print('It is odd and divisible by 7.')
    else:
        print('It could be odd and divisible by 7, but not both. It can also be
none of them.')


    if num % 2 != 0:
        if num % 7 == 0:
            print('It is odd and divisible by 7.')
        else:
            print('It is odd but NOT divisible by 7.')
    else:
        print('It is NOT odd and might be divisible by 7 or not.')
```

!!! -- En defintiva, existen muchos tipor de formas de implementar condicionales en el código.

2. ¿Cuáles son los diferentes tipos de bucles en python? ¿Por qué son útiles?

2.1. Introducción

Los bucles son los elementos que nos permiten recorrer strings, listas, tuplas, diccionarios o rangos accediendo individualmente a cada elemento de estos.

En los condicionales (1.5. Condicional de mas de dos condiciones) hemos visto como podíamos saber si existía un número en un rango. Hemos visto también que podemos tratar de identificar si un número era par o impar y en los casos de los impares divisible por 7.

Queremos comprobar todo esto en un rango del 1 al 100.

```
for num in range(1,101):
    if num % 2 != 0 and num % 7 == 0:
        print(f'{num}:  It is odd and divisible by 7.')
```

Esto nos devolvería lo siguiente:

7: It is odd and divisible by 7. 21: It is odd and divisible by 7. 35: It is odd and divisible by 7. 49: It is odd and divisible by 7. 63: It is odd and divisible by 7. 77: It is odd and divisible by 7. 91: It is odd and divisible by 7.

2.2. Tipos de Bucles

En Python existen dos tipos de bucles:

- Los bucles **"for"**
- Los bucles **"while"**

El bucle **"for"** es el que veíamos en el ejemplo anterior, donde recorríamos los elementos y localizabamos los que cumplían las condición de ser elementos impares y divisibles por 7.

Eso mismo lo podríamos hacer con un bucle **"while"**, pero **ojo con este bucle porque si no introducimos una condición para pararlo, se convertirá en un bucle infinito**.

´´´ counter = 0 while counter < 101: if counter % 2 != 0 and counter % 7 == 0: print(f'{counter}: It is odd and divisible by 7.') counter += 1 ´´´

Esto nos devolvería lo siguiente:

7: It is odd and divisible by 7. 21: It is odd and divisible by 7. 35: It is odd and divisible by 7. 49: It is odd and divisible by 7. 63: It is odd and divisible by 7. 77: It is odd and divisible by 7. 91: It is odd and divisible by 7.

Que como podemos comprobar es lo mismo que nos ha devuelto el bucle "for".

Entonces, ¿cuando usar el bucle for y el bucle while?

En principio, siempre que el elemento tenga un final, como cualquiera de los casos comentados, un string, rango, tupla, lista o diccionario, por ejemplo, serían susceptibles del "for" (ya comentaremos en el siguiente apartado como salir de estos bucles for).

Sin embargo cuando lo que necesitemos es que el bucle sea recorrido hasta que se de una condición concreta, entonces, lo ideal es utilizar el bucle "while".

Un caso típico de este tipo de bucle es en el caso en que partimos de un valor True or False y necesitamos que el bucle se ejecute hasta que cambie este valor.

Por ejemplo, silo que queremos es que en ese rango identifique el **primer elemento impar divisible por 7**. Entonces:

´´´ counter = 0 exit_order = False while exit_order == False: if counter % 2 != 0 and counter % 7 == 0: print(f'{counter}: It is odd and divisible by 7.') exit_order = True counter += 1 ´´´

Y nos devolvería lo siguiente:

7: It is odd and divisible by 7.

La razón es que cuando encuentra el primer elemento la variable "exit_order" adopta el valor de "True" y como al volver al while se encuentra que no cumple la condición, se detiene.

2.3. Bucle for: break or continue

Haremos lo mismo para que en ese rango identifique el **primer elemento impar divisible por 7** usando el bucle for y la instrucción "break". Entonces:

```
for num in range(1,101):
    if num % 2 != 0 and num % 7 == 0:
        print(f'{num}:  It is odd and divisible by 7.')
        break
```

Y nos devolvería lo siguiente:

7: It is odd and divisible by 7.

La razón es que al introducir la instrucción **"break"** el bucle deja de ejecutarse.

La instrucción **"continue"** continuaría con el bucle.

```
for num in range(1,101):
    if num % 2 != 0 and num % 7 == 0:
        print(f'{num}:  It is odd and divisible by 7.')
        continue
```

2.4. Importancia de los Bucles

Los bucles son absolutamente fundamentales. Aunque existen muchas funciones que operan sobre todo tipo de colecciones, a veces necesitamos ser capaces de recorrer los elementos para operar sobre ellos. Sin los bucles, esto sería imposible y por tanto sería imposible implementar algoritmos complejos o dinamismo a una aplicación.

Dos ejemplos que ilustran su utilidad con una variable string y otro ejemplo de listas.

Por ejemplo podríamos recorrer una frase e imprimir cada letra o cada dos letras por ejemplo:

```
phrase = 'Tomorrow is going to be a sunny day.'
for letter in phrase:
    print(letter)

counter = 0
for letter in phrase:
    if counter % 2 == 0:
        print(letter)
    counter += 1
```

Y también nos permitiría también recorrer una lista o cualquier otro elemento haciendo uso de la sintaxis **"for/in"**:

```
animals = ['Dog', 'Cat', 'Horse', 'Ant','Cow','Sheep', 'Whale']
for animal in animals:
    print(animal)
```

Imaginemps finalmente que lo que deseamos es combinar dos listas en un diccionario, siendo la primera la clave, el nombre del animnal y el valor si es un animal de 4 patas o no.

´´´ four_legs = [True, True, True, 'False','True','True', 'False'] animals_dic = {} for animal in animals: for four_leg in four_legs: animals_dic[animal] = four_leg four_legs.remove(four_leg) break

```
print(animals_dic)
```

´´´

!!! -- En defintiva, tenemos dos tipos de bucles en python, los bucles "for" y los bucles "while". En los primeros existe una instrucción para salirnos de ellos "break" y en el segundo es necesario establecer una condición para que pare porque sino acabaríamos en un bucle infinito.

3. ¿Qué es una comprensión de listas en python?

Se trata de recorrer una lista tal y como hemos hecho, pero la sintxis para ello es más breve. Entre los principios del Zen de Python "*simple is better than complex*" y esto así como el explicado "ternary operator" responden a este concepto.

Si en lugar de escribir:

```
starting_letters =[]
animals = ['Dog', 'Cat', 'Horse', 'Ant','Cow','Sheep', 'Whale']
```

```
for animal in animals:
    starting_letters.append(animal[0])
print(starting_letters)
```

escribimos:

```
starting_letters_lc = [animal[0] for animal in animals]
print(starting_letters_lc)
```

Obtenemos exactamente el mismo resultado:

['D', 'C', 'H', 'A', 'C', 'S', 'W']

Es decir, nos devuelve en ambos casos la primera letra de cada palabra contenida en la lista.

Esto puede hacerse algo más complejo con un "if":

```
starting_letters =[]
animals = ['Dog', 'Cat', 'Horse', 'Ant','Cow','Sheep', 'Whale']
for animal in animals:
    if animal[0] == 'C':
    starting_letters.append(animal)
print(starting_letters)
```

Introducimos la condición de que la palabra empiece por 'C' y obtenemos Obtenemos ['Cat', 'Cow']. Entonces obtendríamos el mismo valor con esta sentencia:

```
starting_letters_lc = [animal for animal in animals if animal[0]=='C']
print(starting_letters_lc)
```

Observemos que si quisieramos introducir en la nueva lista unicamente las letras hubieramos introducido:

```
starting_letters =[]
animals = ['Dog', 'Cat', 'Horse', 'Ant','Cow','Sheep', 'Whale']
for animal in animals:
    if animal[0] == 'C':
    starting_letters.append(animal[0])
print(starting_letters)

starting_letters_lc = [animal[0] for animal in animals if animal[0]=='C']
print(starting_letters_lc)
```

y hubieramos obtenido como resultado ['C', 'C'].

!!! -- La comprensión de listas nos permite escribir el codigo de una forma más limpia. Pero ojo, no la hagáis vosotros tan complicada que su objetivo inicial pierda sentido.

4. ¿Qué es un argumento en Python?

4.1. ¿Qué es una función?

Para responder a qué es un argumento es necesario explicar que es una función. Una función es una ejecución de código que podemos llamar desde cualquier parte de nuestro programa y que es parametrizable mediante argumentos.

4.2. ¿Qué es un argumento?

La introducción de argumentos en Python es opcional, pero evidentemente esto crea mucha flexibiliad evitando reescribir código ya que disponemos de esa funcionalidad deseada unicamente cambiando los valores de esos argumentos de la función qeu queremos ejecutar.

Así, por ejemplo entre todo lo visto podemos diseñar una función que cuando la llamaramos nos imprimiera el texto 'Hello, you are welcome!'.

```
def welcome():
    print('Hello, you are welcome!')

welcome()
```

Pero también podríamos pasar el texto a imprimir como parámetro de forma que no fuera siempre el mismo, sino que fuera variable. **Esto sería un argumento, el texto a imprimir**.

```
def welcome(text_to_print):
    print(text_to_print)

welcome('Hi, nice to meet you!')
```

4.3. Tipos de argumentos

En principio los argumentos son **argumentos numerados**, es decir, si en lugar de meter un argumento metiera dos, desde el lugar que se realizar la llamada reconocería el primer parámetro como el primero de la función y el segundo argumento pasado como el segundo parámetro de la función.

```
def welcome(text_to_print, name):
    print(f'{name}! {text_to_print}')

welcome('Hi, nice to meet you!', 'Silvia')
```

Named arguments o argumentos por nombre

Pero también podríamos llamarlos nosotros en el orden que quisieramos siempre que introdujeramos el nombre que recibe la variable que recoge el argumento. Es decir:

```
def welcome(text_to_print, name):
    print(f'{name}! {text_to_print}')

welcome(text_to_print = 'Hi, nice to meet you!', name = 'Silvia')
```

E incluso cambiando el orden, funcionaría igual:

```
def welcome(text_to_print, name):
    print(f'{name}! {text_to_print}')

welcome(name = 'Silvia', text_to_print = 'Hi, nice to meet you!' )
```

Default arguments o argumentos por defecto

Si por ejemplo con la primera función que hemos explicado aplicamos un valor por defecto, nos aseguramos que aunque no se escriba nada aparezca el texto por defecto.

```
def welcome(text_to_print = 'Hi, nice to meet you!'):
    print(text_to_print)
welcome()
```

Multiples argumentos empaquetados

En python existe también la posibilidad de pasar un número indeterminado de argumentos, haciendo uso tanto de "Function arguments unpacking o desempaquetado de argumentos" como de "Keyword arguments o argumentos de claves".

Function arguments unpacking o desempaquetado de argumentos

Mediante el uso de un asterisco (*) delante de la variable que recoge los argumentos hacemos entender al sistema que recibirá ese tipo de argumento, multiple y tipo tupla.

```
def to_do_list(*args):
    print('Your today to do things are: ' + ', '.join(args))

to_do_list('Read news', 'Walk half an hour', 'Study', 'Go to doctor')
```

Esto nos devolverá todas los argumentos unidos por comas:

"Your today to do things are: Read news, Walk half an hour, Study, Go to doctor"

```
def to_do_list(*args):
    print('Your today to do things are:')
    for arg in args:
        print(f'- {arg}')


to_do_list('Read news', 'Walk half an hour', 'Study', 'Go to doctor')
```

Y sin embargo esto, haciendo uso de un bucle for, nos devolverá los datos en forma de lista tradicional (no de lista python):

Your today to do things are:

- Read news
- Walk half an hour
- Study
- Go to doctor

Esto también podría incluir otro argumento, como el nombre (name), y sería igualmente ejecutable:

```
def to_do_list(name, *args):
    print(f'Welcome {name}. Your today to do things are:')
    for arg in args:
        print(f'- {arg}')


to_do_list('Silvia',
           'Read news', 'Walk half an hour', 'Study', 'Go to doctor')
```

Keyword arguments o argumentos de claves

De la misma forma podemos pasar keyword arguments, que se correponderían a un diccionario. Mediante el uso de dos asteriscos (**) delante de la variable que recoge los argumentos hacemos entender al sistema que recibirá ese tipo de argumento, multiple y tipo diccionario.

```
def to_do_list_kw(name, **kwargs):
    print(f'Welcome {name}. You need to call today to:')
    for key, value in kwargs.items():
        print(f'- {key} : {value}')
```

```
to_do_list_kw('Silvia',
            Maria='666 888 777', George= '777 888 666', Raul = '666 444 999')
```

Esto nos devolverá lo siguiente:

Welcome Silvia. You need to call today to:

- Maria : 666 888 777
- George : 777 888 666
- Raul : 666 444 999

!!! -- En defintiva, existen muchos tipos de argumentos, incluso una función podría ser un argumento y pueden ir todos combinados en una función.

Ejemplo pasando una función ya definida:

```
def sequence(first, second, third, fourth, fifth):
    total = first + second + third + fourth + fifth
    return total

def named_arguments_practice(sequence):
    valor = sequence(first = 1, second = 2, third = 3, fourth = 4, fifth=5)
    return valor

print(named_arguments_practice(sequence))
```

5. ¿Qué es una función de Python Lambda?

La función de Python Lambda es una herramienta que te permite empaquetar una función y obtener directamente el resultado de esta en una variable. Por ejemplo, si lo que quiero es obtener un nombre completo a partir de su nombre y apellidos, no necesariamente tendríamos que crear una función mediante "def" donde almacenar el valor y que luego nos lo devuelva en otra variable. Podemos hacerlo directamente con la función lambda.

Con una función:

```
def full_name_function(first_name, last_name):
    return f'{first_name} {last_name}'

print(full_name_function('Ernest', 'Hemingway'))
```

Con lambda function:

```
full_name = lambda first, last: f'{first} {last}'
print(full_name('Ernest', 'Hemingway'))
```

y en ambos casos nos devolvería "Ernest Hemingway".

!!! -- En defintiva, fundamentalmente, la función lambda permite la ejecución de el código almacenando directamente el valor en la variable que recoge el valor.

6. ¿Qué es un paquete pip?

Pip es el acrónimo de "PIP Installs Packages", que es una herramienta que permite instalar y desinstalar paquetes de Pyython. Además las posibles dependencias que estos paquetes tengan las resuelve automáticamente.

Podemos decir que los paquetes de python, aunque no todos, residen en lo que también es llamado "the cheese shop" o "Python Package Index". Por lo tanto, para ejecutarlos, es necesario llamarlos haciendo uso de Pip.

6.1. Instalación de pip

Pip normalmente viene incluido en las últimas versiones de python. Para comprobar que lo tenemos en la terminal escribimos:

´´´ pip --version ´´´ y si nos devuelve la versión y el lugar de instalación es que estará instalado. En caso de no estar instalado, hay que seguir los siguientes pasos en windows:

1. Descargar el scrip de instalación "get-pip.py".
2. Acceder por la terminal de ocmando hasta el directorio donde hemos guardado el fichero.
3. Ejecutar el comando "python get-pip.py".

6.2. Instalación de paquetes pip que no tenemos

Accederíamos por la terminal de comandos y mediante la siguiente instrucción, instalaríamos, por ejemplo el paquete "requests".

```
pip install requests
```

6.3. Importación de paquetes

Podemos importarlos enteros o podemos importar las funciones que nos interesan de esos paquetes. Y ya en Python, mediante los siguientes comandos hacemos la importación de:

Importar el paquete externo entero:

```
import requests
```

Importar el paquete externo entero poniendole un alias:

```
import requests as rq
```

Importar una función del paquete externo:

```
from bs4 import BeautifulSoup
```

Importar una función de un paquete interno:

También podríamos llamar a un fichero python donde almacenamos variables en nuestro sistema. Imaginemos que tenemos un fichero llamado "cooking_functions.py" donde almacenamos una variable llamada "boiling_point", que recoge el argumento de la comida que vayas a preparar.

Para ello usaremos este código:

```
from cooking_functions import boiling_point
```

y una vez importado podremos llamarlo desde nuestro código. También podremos importar el paquete interno entero como haríamos con un externo.

6.4. Trabajar con los paquetes

Si queremos llamar a una función tras **importar el paquete entero**:

```
paguete.funcion()
```

Si queremos llamar a una función tras **importar el paquete entero CON ALIAS**:

```
alias.funcion()
```

Si queremos llamar a una función tras **Importar una función del paquete**:

```
funcion()
```

6.5. Ejemplo de código haciendo uso de los paquetes para comunicarnos con las APIs

API es el acrónimo para "Application Programming Interface". Las API son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos.

Vamos a recoger una tabla de una página web haciendo uso de la libreria "pandas": Es cierto que "pandas" tiene otras dependencias que también tendremos que instalar como "lxml" y "html5lib "

```
import pandas as pd
table_mn = pd.read_html('https://en.wikipedia.org/wiki/Minnesota')
print(f'Total tables: {len(table_mn)}')
if len(table_MN)>2:
    table_3 = table_mn[2]
    print(table_3)
```

Y esto nos va a devolver el contenido de la tercera tabla de esa página web que es el siguiente:

```
Total tables: 30
              Location July (°F) July (°C) January (°F) January (°C)
0           Minneapolis     83/64     28/18         23/7       -4/-13
1            Saint Paul     83/63     28/17         23/6       -5/-14
2             Rochester     82/63     28/17         23/3       -5/-16
3                Duluth     76/55     24/13         19/1       -7/-17
4             St. Cloud     81/58     27/14        18/-1       -7/-18
5               Mankato     86/62     30/16         23/3       -5/-16
6  International Falls     77/52     25/11        15/-6       -9/-21
```

!!! -- En defintiva, pip nos permite acceder a funcionalidad que ya están programadas por otros y hacer más potentes nuestros programas sin necesidad de volverlo a reprogramar.