M3C5 Python Assignment

felicidades por haber llegado al ***CheckPoint 5****, este será un poco distinto a los anteriores, vas a poder utilizar* ***documentación e info****rmación que busques por la* ***web****, porque el objetivo es que tú tienes que crear una documentación sobre las preguntas que te enviaré a continuación, por lo tanto, tienes que prepararlas para las personas que están* ***iniciando en el mundo del desarrollo y estas puedan comprender y aprender con tu material,*** *por favor intenta ser lo* ***más detallado posible, colocar ejemplos, para que se utiliza, sintaxis, como se utiliza, etc...*** *Recuerda que eres el responsable de crear la documentación que utilizarán los nuevos compañeros de tu equipo de programación. También* ***toma en cuenta en hacerlo con las herramientas o software correcto, para la creación de este tipo de material y subirlo a Git-Hub*** *para revisarlo, seremos* ***exigentes*** *con el contenido del mismo así que da lo mejor de ti en esta entrega!*
¿Qué es un ***condicional****?*
¿Cuáles son los ***diferentes tipos*** *de* ***bucles*** *en Python?* ***¿Por qué*** *son* ***útiles****?*
¿Qué es una ***lista por comprensión*** *en Python?*
¿Qué es un ***argumento*** *en Python?*
¿Qué es una ***función Lambda*** *en Python?*
¿Qué es un ***paquete pip****?*

Adicional a esta asignación de crear una documentación, necesito que realices los siguientes ejercicios ***prácticos****, recuerda* ***subirlos*** *a* ***Git-Hub*** *o* ***Replit*** *para revisarlos*

> *Cree un* ***bucle For*** *de Python.*
>
> *Cree una* ***función de Python*** *llamada* ***suma*** *que tome* ***3 argumentos y devuelva la suma*** *de los* ***3****.*
>
> *Cree una* ***función lambda*** *con la* ***misma funcionalidad*** *que la función de* ***suma*** *que acaba de crear.*
>
> *-Utilizando la siguiente* ***lista y variable,*** *determine si el* ***valor*** *de la* ***variable coincide o no*** *con un* ***valor*** *de la* ***lista. *Sugerencia****, si es necesario,* ***utilice*** *un* ***bucle for in*** *y el* ***operador in.***

```
nombre = 'Enrique'
lista_nombre = 'Jessica', 'Paul', 'George', 'Henry', 'Adán'
```

6 preguntas teóricas

1. ¿Qué es un Condicional?

Todo programa informático es un conjunto de **algoritmos,** es decir, un conjunto de instrucciones para poder realizar alguna tarea. Estas instrucciones se ejecutan de manera secuencial de arriba hacia abajo, este orden se lo denominamos el **flujo del programa.** Este flujo lo podemos controlar dependiendo de las “decisiones” que requerimos que ejecute nuestro programa y para esto tenemos las llamadas **estructuras de control de flujo.**

¿Qué son las estructuras de control de flujo?

Las estructuras de control de flujo son **sentencias** que nos permiten cambiar el flujo secuencial de nuestro código para que tome otros “caminos” o repita ciertas instrucciones. Estas se pueden dividir en:

- Condicionales
- Ciclos

Condicionales

Las sentencias condicionales son aquellas que permiten que dentro del flujo del programa se tomen **diferentes caminos** para ejecutar bloques de código en función de **condiciones establecidas.**

Puedes imaginar esto como una carretera en la cual encontramos diversas vías.

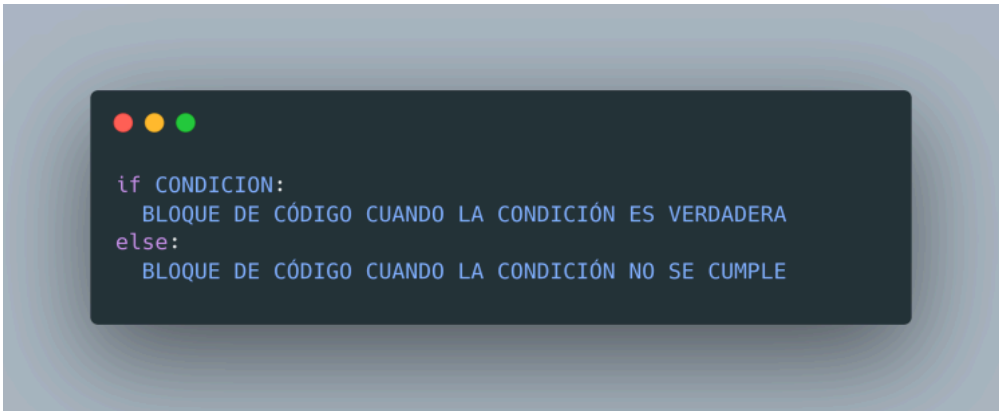
Dependiendo del letrero que tenga cada una o según el objetivo de tu viaje, decides irte por uno u otro camino.

En Python contamos con dos formas de manejar las sentencias condicionales: **if - else** y **match - case.**

Ya teniendo claro lo que en términos generales significa un condicional, conozcamos la sentencia **if - else** a profundidad.

¿Cómo se usa la sentencia if - else?

Esta sentencia tiene la siguiente estructura:



```
if CONDICION:
  BLOQUE DE CÓDIGO CUANDO LA CONDICIÓN ES VERDADERA
else:
  BLOQUE DE CÓDIGO CUANDO LA CONDICIÓN NO SE CUMPLE
```

Para escribir las condiciones se debe emplear las expresiones de comparación.

Operadores relacionales o de comparación

Estos operadores son empleados para comparar dos o más valores, lo cual nos dará como resultado dos posibles respuestas True o False. Estos operadores serán claves cuando estés trabajando con estructuras de control de flujo como condicionales o ciclos.

Operador	Uso	Descripción
>	a > b	Evalúa si **a** es mayor que **b**
≥	a >= b	Evalúa si **a** es mayor o igual que **b**
<	a < b	Evalúa si **a** es menor que **b**
≤	a ≤ b	Evalúa si **a** es menor o igual que **b**
==	a == b	Evalúa si **a** es igual que **b**
≠	a ≠ b	Evalúa si **a** es diferente de **b**

Veamos un **ejemplo,** supongamos que deseamos preparar un pastel y para hornearlo tenemos como temperatura estándar 350 °C.

Creamos un condicional que se encargue de enviarnos un mensaje cuando la temperatura sea normal.

```
temperatura = 350
if temperatura <= 350:
    print("Los niveles de temperatura están normales 🍰")
```

Pero cuando la temperatura sea muy alta nos envía una alerta de que se nos va a quemar el pastel.

```
temperatura = 500
if temperatura <= 350:
    print("Los niveles de temperatura están normales 🍰")
else:
    print("⚠️ ¡La temperatura está muy alta y se va a quemar el pastel! ⚠️")
```

*En Python no es necesario incluir **paréntesis** al escribir las condiciones. Pero es muy recomendable utilizarlo para legibilidad o para establecer prioridades.*

Sentencias if - else anidadas

Puede ocurrir que a la hora de tomar una camino en un condicional este tenga múltiples decisiones más, por lo cual cuando a base de una decisión se debe pasar por otras, estas las tenemos que organizar a través de condicionales anidadas.

Continuemos con nuestro **ejemplo** que nos permite analizar la temperatura a la cual se debe cocinar el pastel 🎂. Ahora estableceremos alertas con colores según se cumplan las siguientes condiciones:

- Si los niveles de temperatura es **exactamente 350°** tendrá un nivel de color **verde**
- Si los niveles de temperatura están por **debajo de 350°**, tendrán un nivel de color **azul**
- Si los niveles de temperatura están por **encima de 350°,** tendrán un nivel de color **rojo**

```
temperatura = 500
```

```
if temperatura == 350:
    if temperatura < 350:
        print("Nivel de color azul 🔵")
    else:
        print("Nivel de color verde 🟢")
else:
    print("Nivel de color rojo 🔴")
```

Sentencia if - elif - else

Cuando ya tenemos varios ***if-else*** anidados corremos el riesgo de dificultar la legibilidad del mismo, por eso Python nos ofrece una sentencia para manejar estas situaciones conocida como **elif.**

El anterior ejemplo va a tener estos nuevos requerimientos:

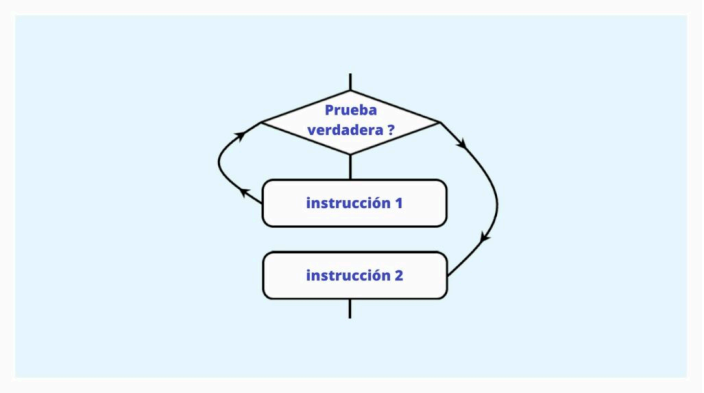
- Si los niveles de temperatura es **exactamente 350°** tendrá un nivel de color **verde**
- Si los niveles de temperatura están **por debajo de 200°**, tendrán un nivel de color **azul**
- Si los niveles de temperatura son **iguales a 400°,** tendrán un nivel de color **naranja**
- Si los niveles de temperatura son **mayores a 400°**, tendrán un nivel de color **rojo**

```
temperatura = 500

if temperatura == 350:
    print("Nivel de color verde 🟢")
elif temperatura < 200:
    print("Nivel de color azul 🔵")
elif temperatura == 400:
    print("Nivel de color naranja 🟠")
else:
    print("Nivel de color rojo 🔴")
```

Cuando usas la sentencia **elif** hay que establecer el condicional que deseas analizar al igual que con la sentencia **if.** Con esto ya tienes tu primera herramienta para controlar el flujo de tu aplicación. Nos queda pendiente ver cómo se emplea la sentencia **match - case** y los **ciclos.**

2. ¿Cuáles son los diferentes tipos de bucles en Python?



Los bucles son una herramienta muy importante en la programación, ya que nos permiten ejecutar un bloque de código varias veces seguidas, se trata de uno de los conceptos básicos en programación. Hay dos tipos principales de bucles: los bucles «while» y los bucles «for».

El **bucle «while»** se ejecuta mientras se cumpla una condición. Por **ejemplo:**

```
while (x < 10):
 print(x)
 x = x + 1
```

En este ejemplo, el bucle se ejecutará mientras «x» sea menor que 10. Cada vez que se ejecuta el bucle, se imprime el valor de «x» y luego se aumenta en 1.

El **bucle «for»** es similar, pero se utiliza para iterar sobre una secuencia de elementos, como una lista o un rango de números. Por **ejemplo:**

```
for i in range(10):
 print(i)
```

En este ejemplo, el bucle «for» se ejecutará 10 veces, con «i» tomando valores desde 0 hasta 9. Cada vez que se ejecuta el bucle, se imprime el valor de «i».

Por otro lado, tenemos bucles que, aunque sean menos conocidos o utilizados, también podemos tener en cuenta, estos bucles son:

Bucle «do-while»: El bucle «do-while» es similar al bucle «while», pero se asegura de que el código dentro del bucle se ejecute al menos una vez. Por **ejemplo:**

```
do:
 print(x)
 x = x + 1
while (x < 10)
```

En este ejemplo, el bucle se ejecutará **al menos una vez**, y luego se continuará ejecutando mientras «x» sea menor que 10.

Bucle «for-each»: El bucle «for-each» es una forma especial de bucle «for» que se utiliza para iterar sobre elementos de una secuencia de manera más **sencilla**. Por **ejemplo:**

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

En este ejemplo, el bucle «for-each» se ejecutará para cada elemento en la lista «list». Cada vez que se ejecuta el bucle, se imprime el valor del elemento actual.

Es importante tener en cuenta que es fácil quedarse atrapado en un bucle infinito si olvidamos aumentar o cambiar la condición de parada. Por lo tanto, es importante asegurarse de que los bucles tengan una condición de salida clara y que sean controlados adecuadamente.

En resumen, los bucles son una herramienta muy útil en la programación y nos permiten ejecutar un bloque de código varias veces seguidas. **Hay dos tipos principales de bucles: «while» y «for»,** y es **importante** asegurarse de que los bucles tengan una **condición de salida clara** y que sean **controlados** adecuadamente para **evitar bucles infinitos.**

¿Por qué son útiles?

Ya hemos dicho que los bucles son una herramienta fundamental en la programación y se utilizan en una amplia variedad de operaciones que necesitamos que se apliquen a nuestros programas. Algunas de las formas más comunes en que se utilizan los bucles incluyen:

1. **Repetir una tarea varias veces:** Los bucles son muy útiles para realizar una tarea determinada varias veces. Por ejemplo, podríamos usar un bucle para imprimir los números del 1 al 10, o para procesar todos los elementos de una lista.
2. **Buscar y procesar datos:** Los bucles también se pueden utilizar para buscar y procesar datos en una base de datos o en un archivo de datos. Por ejemplo, podríamos usar un bucle para buscar todas las entradas de una base de datos que cumplan ciertos criterios y luego procesar esos datos de alguna manera.
3. **Interactuar con el usuario:** Los bucles también se pueden utilizar para interactuar con el usuario de un programa. Por ejemplo, podríamos usar un bucle para pedir al usuario que ingrese una serie de números y luego procesar esos números de alguna manera.
4. **Crear animaciones y juegos:** Los bucles también se pueden utilizar para crear animaciones y juegos. Por ejemplo, podríamos usar un bucle para actualizar la posición de un personaje en un juego de plataformas o para crear un efecto de animación en una aplicación gráfica.

3. ¿Qué es una lista por comprensión en Python?

Las listas por comprensión es una construcción sintáctica disponible en *Python* con la que se pueden crear listas a partir de otros elementos iterables. Siendo una de las contracciones más elegantes del lenguaje. A continuación, se mostrará la sintaxis básica para trabajar con las listas por comprensión.

Sintaxis de las listas por comprensión en Python

Las sintaxis básicas de las listas por comprensión en Python se pueden resumir en la siguiente línea:

```
nueva_lista = [expresion bucle_for condiciones]
```

Entre corchetes se escribe una expresión seguida de un bucle for sobre el que se itera, para finalmente escribir unas condiciones.

Ejemplo básico de uso

Un ejemplo básico de uso de las listas por comprensión es aplicar una operación sobre un vector. Por ejemplo, añadir una cantidad a todos los elementos de este. Lo que se puede hacer con un bucle tradicional

```
numbers = [1, 2, 3, 4]
results = []

for n in numbers:
    results.append(n + 1)

results
[2, 3, 4, 5]
```

Aunque es más elegante utilizar las listas por comprensión

```
numbers = [1, 2, 3, 4]
results = [n + 1 for n in numbers]
```

Obteniéndose el mismo resultado solamente con mucho menos código. En el código los corchetes indican que la salida de la lista `n + 1` es la expresión que ejecutar para cada uno de los elementos del bucle for. Es decir, que a cada uno de los valores sobre los que se itera se añada se le sume la unidad.

Condiciones en las listas por comprensión

Tal como se ha indicado anteriormente es posible añadir condiciones a las listas por comprensión en Python. Para lo que solamente se tiene que agregar un **if** al final con la condición. Siguiendo con el ejemplo anterior, se podría sumar uno solamente a los registros que sean menores que tres.

```
numbers = [1, 2, 3, 4]
results = [n + 1 for n in numbers if n < 3]
Results
[2, 3]
```

Al ejecutar el código se puede observar que solamente se tienen dos registros, los que cumple la condición. En el caso de que se desee realizar una operación diferente cuando no se cumple la condición se puede hacer con un `else`. Aunque es necesario cambiar modificar el orden. Si se utiliza un `else` la condición se tiene que situar justamente después de la expresión y antes del `for`. Por ejemplo, en el siguiente código los números mayores o iguales que tres se dejan sin modificar.

```
numbers = [1, 2, 3, 4]
results = [n + 1 if n < 3 else n for n in numbers]
results
[2, 3, 3, 4]
```

Identificar los números comunes en dos listas

La posibilidad de anidar bucles `for` en las listas por comprensión permiten realizar operaciones realmente complejas. Así se puede iterar sobre varios objetos iterables para aplicar una condición.

Un ejemplo típico de esto es buscar el conjunto de elementos comunes en dos listas. Lo que se puede conseguir con el siguiente código.

```
names_1 = ['Oralie' ,'Imojean' ,'Michele', 'Ailbert', 'Stevy']
names_2 = ['Jayson', 'Oralie' ,'Michele', 'Stevy', 'Alwyn']

common = [a for a in names_1 for b in names_2 if a == b]
Common
['Oralie', 'Michele', 'Stevy']
```

En donde se selecciona el valor de la primera lista si al iterar sobre la segunda también se encuentra en esta. Si no aparece el registro de ignorará. Para hacer esto mismo con un bucle for tradicional es necesario escribir mucho más código.

```
list_a = ['Oralie' ,'Imojean' ,'Michele', 'Ailbert', 'Stevy']
list_b = ['Jayson', 'Oralie' ,'Michele', 'Stevy', 'Alwyn']
common = []

for a in names_1:
   for b in names_2:
       if a == b:
           common.append(a)
```

Conclusiones

En esta entrada se han visto las listas por comprensión en Python, una construcción sintáctica disponible que ofrece grandes posibilidades. Permitiendo crear código más compacto y legible. Las listas por comprensión son otra de las herramientas disponibles en Python que permite crear código compacto y elegante.

4. ¿Qué es una función Lambda en Python?

Definición de Lambda

En *Python*, una función Lambda se refiere a una **pequeña función anónima.** Las llamamos "funciones anónimas" porque técnicamente carecen de nombre.

Al contrario que una función normal, no la definimos con la palabra clave estándar *def* que utilizamos en *Python*. En su lugar, las funciones Lambda se definen como una **línea que ejecuta una sola expresión.** Este tipo de funciones pueden tomar cualquier número de argumentos, pero solo pueden tener una expresión.

Sintaxis básica

Todas las **funciones Lambda** en *Python* tienen exactamente la misma sintaxis:

```
#Escribo p1 y p2 como parámetros 1 y 2 de la función.
```

```
lambda p1, p2: expresión
```

Como mejor te lo puedo explicar es enseñándote un ejemplo básico, vamos a ver una **función normal** y un ejemplo de **Lambda:**

```
#Aquí tenemos una función creada para sumar.
def suma(x,y):
   return(x + y)
#Aquí tenemos una función Lambda que también suma.
lambda x,y : x + y
#Para poder utilizarla necesitamos guardarla en una variable.
suma_dos = lambda x,y : x + y
```

Al igual que ocurre en las *list comprehensions*, lo que hemos hecho es escribir el código en una sola línea y limpiar la sintaxis innecesaria.

En lugar de usar ***def*** para definir nuestra función, hemos utilizado la palabra clave **lambda;** a continuación escribimos **x, y** como argumentos de la función, y **x + y** como expresión. Además, se omite la palabra clave **return,** condensando aún más la sintaxis.

Por último, y aunque la definición es anónima, la almacenamos en la variable ***suma_dos*** para poder llamarla desde cualquier parte del código, de no ser así tan solo podríamos hacer uso de ella en la línea donde la definamos.

Las funciones **Lambda** son funciones **anónimas** que **solo** pueden contener una **expresión. Así creas una función en Python:**

```
def mi_func(argumentos):
  # cuerpo de la función
```

Se **declara** la función con la palabra clave `def`, les das un **nombre** y entre **paréntesis** los **argumentos** que recibirá la función. Puede haber tantas líneas de código como quieras con todas las expresiones y declaraciones que necesites.

Pero algunas veces solo necesitarás una expresión dentro de tu función, por ejemplo, una función que duplique el valor de un argumento:

```
def doble(x):
  return x * 2
```

Esta función puede ser usada dentro de la función `map`, de la siguiente forma:

```
def doble(x):
  return x * 2

mi_lista = [18, -3, 5, 0, -1, 12]
lista_nueva = list(filter(lambda x: x > 0, mi_lista))
print(lista_nueva) # [18, 5, 12]
```

Este es un buen lugar para usar una función lambda, ya que puede ser creada en el mismo lugar donde se necesite, esto significa menos líneas de código y así también evitarás nombrar una función que solo utilizaras una sola vez y que tendría que ser almacenada en memoria.

Como usar funciones lambda en Python

Una función lambda se usa cuando necesitas una función sencilla y de rápido acceso: por ejemplo, como argumento de una función de orden mayor como los son `map` o `filter`

La sintaxis de una función lambda es `lambda args: expresión`. Primero escribes la palabra clave `lambda`, dejas un espacio, después los argumentos que necesites separados por coma, dos puntos `:`, y por último la expresión que será el cuerpo de la función.

Recuerda que no puedes darle un nombre a una función lambda, ya que estas son anónimas (sin nombre) por definición.

Una función lambda puede tener tantos argumentos como necesites, pero debe tener una sola expresión.

Ejemplo 1

Por ejemplo, puedes escribir una función lambda que duplique sus argumentos `lambda x: x * 2` y usarla con la función `map` para duplicar todos los elementos de una lista:

```
mi_lista = [1, 2, 3, 4, 5, 6]
lista_nueva = list(map(lambda x: x * 2, mi_lista))
print(lista_nueva)  # [2, 4, 6, 8, 10, 12]
```

Ejemplo 2

También puedes escribir una función lambda que revise si un número es positivo, `lambda x: x > 0`, y usarla con la función `filter` para crear una lista de números positivos..

```
mi_lista = [18, -3, 5, 0, -1, 12]
lista_nueva = list(filter(lambda x: x > 0, mi_lista))
print(lista_nueva) # [18, 5, 12]
```

Una función lambda se define donde se usa, de esta manera no hay una función extra utilizando espacio en memoria.

Si una función es utilizada una sola vez, lo mejor es usar una función lambda para evitar código innecesario y desorganizado.

Ejemplo 3

También es posible que una función devuelva una función lambda. Si necesitas funciones que multipliquen diferentes números, por ejemplo, duplicar, triplicar, etc... una función lambda puede ser útil En lugar de crear múltiples funciones, puedes crear una sola función `multiplicar_por()` y llamarla con diferentes argumentos para crear una función que duplique o triplique.

```
def multiplicar_por (n):
 return lambda x: x * n

duplicar = multiplicar_por(2)
triplicar = multiplicar_por(3)
diez_veces = multiplicar_por(10)
```

La función lambda toma el valor de `n` de la función `multiplicar_por(n)` así que en `duplicar` el valor de `n` es `2`, en `triplicar` `n` vale `3` y en `diez_veces` vale `10`. Y al llamar a estas funciones con un argumento podemos retornar el número multiplicado.

```
duplicar(6)
> 12
triplicar(5)
> 15
diez_veces(12)
> 120
```

Si no usáramos funciones lambda, tendríamos que crear una función diferente dentro de `multiplicar_por,` y se vería algo así:

```
def multiplicar_por(x):
  def temp (n):
    return x*n
  return temp
```

Usando una función lambda el código necesita menos líneas de código y es más legible.

Conclusión

Las funciones lambda son una manera compacta de escribir una función si solo necesitas una expresión corta.

5. ¿Qué es un argumento en Python?

Argumentos y parámetros

Un argumento no será más que el valor que vamos a ingresar al momento de llamar a una función, y los parámetros serán variables definidas en la función misma que podrán almacenar los argumentos ingresados.

Ejemplo:

```
def suma(a, b, c):
    return a + b + c
```

En este caso hemos definido la función suma. Función que posee 3 parámetros (a, b y c). Estos parámetros no poseen valores por default, esto quiere decir que al hacer el llamado a la función, será necesario definir sus valores. Y para ello la forma más sencilla es utilizar argumentos.

Simplemente, al hacer el llamado a la función, colocamos los argumentos. En este caso 3. Ejemplos.

```
# Argumentos por posición
suma(10, 20, 30)
60
```

En Python la asignación de argumentos se realiza por **posición**, esto quiere decir que el primer argumento es asignado al primer parámetro, el segundo argumento al segundo parámetro y así sucesivamente. A esta asignación la conoceremos como ***positional argument***. Y será la forma más común de llamar a una función.

Hasta aquí todo bien con los argumentos y los parámetros. Sin embargo, en Python existe otra forma en la cual podemos llamar a una función, y esta vez lo haremos utilizando el nombre de los parámetros.

Veamos.

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

```
60
```

En este caso, como podemos observar, asignamos valores ya no por posición, si no por nombre. Lo interesante de la **asignación por nombre**, es que el orden no importa; solo basta con colocar el nombre del parámetro (Al cual queremos hacer la asignación), signo igual (=) y su correspondiente valor. A esta asignación la conoceremos como ***keyword argument***.

NOTA: Una muy buena práctica en Python, es realizar la asignación por nombre sin espacio. variable=valor.

La asignación por nombre lo podemos hacer en prácticamente cualquier versión de Python 2.x o 3.x. Lo interesante ocurren en versiones actuales del lenguaje, donde es posible definir que valores podrán ser asignados por posición o por nombre.

Keyword only

Listo, los conceptos básicos ya los tenemos, ahora modifiquemos un poco el ejemplo, y trabajemos con solo argumentos por posición.

```
def suma(a, *, b, c):
    return a + b + c
```

En este caso hemos añadido el asterisco (*) en la sección de parámetros, pero ¿Qué significa esto? Verás, al hacer esto, le indicamos a Python que todos los parámetros que se encuentren a la derecha del asterisco únicamente podrán tomar su valor por nombre y no por posición.

Si intento ejecutar la función utilizando únicamente positional arguments obtendremos un error.

```
>>> suma(10, 20, 30)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: suma() takes 1 positional argument but 3 were given
```

El error es bastante claro, la función solo puede recibir un argumento por posición (que sería para el parámetro a) pero estamos pasando 3.

Lo quedemos hacer para evitar el error, es simplemente asignar los valores por *keyword arguments*.

```
>>> suma(10, b=20, c=30)
60
```

Ahora sí, obtendremos el resultado deseado, sin ningún error.

Y ¿Qué pasa si intentamos asignar un valor directamente para el parámetro a?

```
>>> suma(a=10, b=20, c=30)
60
```

Bueno, no pasa absolutamente nada.

A los parámetros que se encuentren a la izquierda del asterisco (*) se les puede asignar su valor ya sea por posición o por nombre.
El uso del asterisco (*) nos permite un mayor control sobre la asignación o modificación de valores para los parámetros. Por ejemplo, que pasa si tenemos un parámetro que tiene un valor por default, pero solo queremos que dicho valor sea modificado cuando el usuario lo haga de forma explícita, es decir, directamente por su nombre.

En esos casos lo mejor que podemos hacer es utilizar el Keyword only. Veamos.

```
def database_connection(username, password, host, *, port=3306):
    pass
```

En este caso donde la función posee 4 parámetros, pero solo quiero que 3 de ellos puedan ser asignados por posición o nombre, hacemos uso del asterisco. Con lo cual, si el usuario quiere

utilizar un puerto diferente al default, debe hacerlo de forma explícita, reduciendo la posibilidad de error.

```
database_connection('eduardo_gpg', 'password123', 'localhost', port=29017)
```

Positional only

Continuando con el post,algo que me gustaría mencionar es que, a partir de la versión **3.8** de Python, es posible la asignación de valores solo por **posición**, es decir, lo contrario de lo que acabamos de ver. Y para ello haremos uso del slash (/).

```
def suma(a, b,  /, c):
    return a + b + c
```

Para este nuevo ejemplo, le indicamos a Python que todos los parámetro que se encuentren a la izquierda del slash , solo podrán tomar su valor por **posición y no por nombre.**

Si por ejemplo intento ejecutar asignando nombres, obtendré un error.

```
>>> suma(a=10, b=20, c=30)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: suma() got some positional-only arguments passed as keyword
arguments: 'a, b'
```

De igual forma, el error es bastante claro. No es posible utilizar argumentos por nombre para a y b.

Lo correcto sería hacerlo todo por posición.

```
>>> suma(10, 20, 30)
60
```

Lo interesante de todo esto es que, al limitar argumentos solo por posición, en caso deseemos modificar la definición de la función, por ejemplo que ahora los parámetros cambien sus nombre (n1, n2 y n3) este cambio no debiera repercutir de ninguna forma a las implementaciones de la función, ya que la asignación siempre se hizo por posición y nunca por nombre, por lo tanto sería un cambio completamente transparente para el usuario.

Keyword only & Positional only

Ya para finalizar me gustaría mencionar que sí, es posible hacer uso de los ***keyword only*** y ***positional only*** en una misma función.
Aquí un pequeño ejemplo.

```
def function(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):

            -----------    ----------     ----------

                  |              |                   |

                  |        Positional or keyword    |

                  |                                 - Keyword only

                  -- Positional only
```

6. ¿Qué es un paquete pip?

PIP es un administrador de paquetes para paquetes o módulos de Python, si lo desea.

Nota: Si tiene Python versión 3.4 o posterior, PIP se incluye de forma predeterminada.

¿Qué es un paquete?

Un paquete contiene todos los archivos necesarios para un módulo.

Los módulos son bibliotecas de código Python que puedes incluir en tu proyecto.

Compruebe si PIP está instalado

Navega por tu línea de comando hasta la ubicación del directorio de secuencias de comandos de Python y escribe lo siguiente:

Ejemplo

Verificar versión PIP:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip --version
```

Instalar PIP

Si no tiene PIP instalado, puede descargarlo e instalarlo desde esta página: https://pypi.org/project/pip/

Descargar un paquete

Descargar un paquete es muy fácil.

Abra la interfaz de línea de comando y dígale a PIP que descargue el paquete que desea.

Navega por tu línea de comando hasta la ubicación del directorio de secuencias de comandos de Python y escribe lo siguiente:

Ejemplo

Descargar un paquete llamado "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip install camelcase
```
¡Ahora ha descargado e instalado su primer paquete!

Usando un paquete

Una vez instalado el paquete, estará listo para usar.Importe el paquete "camelcase" a su proyecto.

```
import camelcase
c = camelcase.CamelCase()
txt = "hello world"
print(c.hump(txt))
```

Buscar paquetes

Encuentra más paquetes en https://pypi.org/.

Eliminar un paquete

Utilice el comando desinstalar para eliminar un paquete:

Ejemplo

Desinstale el paquete llamado "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip uninstall camelcase
```

El Administrador de paquetes PIP le pedirá que confirme que desea eliminar el paquete camelcase:

```
Uninstalling camelcase-02.1:

  Would remove:

    c:\users\Your Name\appdata\local\programs\python\python36-32\lib\site-packages\camelcase-0.2-py3.6.egg-info

    c:\users\Your Name\appdata\local\programs\python\python36-32\lib\site-packages\camelcase\*

Proceed (y/n)?
```

Presione `y` y el paquete será eliminado.

Listar paquetes

Utilice el comando `list` para enumerar todos los paquetes instalados en su sistema:

Ejemplo

Listar paquetes instalados:

```
C:\Users\Your
Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip list
```

Resultado:

```
Package          Version
-----------------------
camelcase        0.2
mysql-connector 2.1.6
pip              18.1
pymongo          3.6.1
setuptools       39.0.1
```