M3C5 Python Assignment – Preguntas teóricas

Buenos días John, felicidades por haber llegado al CheckPoint 5, este será un poco distinto a los anteriores, vas a poder utilizar documentación e información 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 qué 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!

1. ¿Qué es un condicional?

Los condicionales son estructuras de programación que permiten que el programa que estoy codificando se vuelva dinámico. Estas estructuras permiten dirigir el flujo de ejecución del programa en función de condiciones específicas o repetir un bloque de código múltiples veces[1]. Con programa dinámico queremos decir aquel tipo de programa que puede tomar decisiones basado en unos criterios que yo como programador impongo en mi código.

Los condicionales son simplemente una forma de decir que si ocurre una situación, quiero que mi programa realice una tarea concreta o se comporte de un modo determinado, pero si ocurre una situación diferente, realice otra tarea o genere una respuesta distinta. En este sentido es similar a una toma de decisiones que haríamos en nuestra vida cotidiana basadas en una condición, por ejemplo "*si sale el sol esta mañana, tomo mis cosas y voy a la playa, de lo contrario me quedaré en casa descansando*" la acción o respuesta está de este modo condicionada por lo que suceda en este caso concreto con el clima "*si sale el sol*" entonces lo que haré como resultado "*voy a la playa*" de lo contrario mi comportamiento cambia y "*me quedaré en casa descansando*". Las sentencias condicionales nos ayudan a controlar el flujo de nuestro programa, decidiendo si unas líneas de código se tienen que ejecutar en función de si se cumplen unas condiciones preestablecidas. Comprueban por tanto si una condición es verdadera o falsa para tomar o no cierta acción[2].

---

[1] Huet, P. (2024). Página web sobre programación OpenWebinars [Blog]. Recuperado de: https://openwebinars.net/blog/fundamentos-de-python-sintaxis-variables-y-estructuras-de-control/ (20 de mayo de 2024).

[2] Brugués, A. (2021). Página web programa_en_python [Blog]. Recuperado de: https://www.programaenpython.com/fundamentos/sentencias-condicionales-en-python/ (20 de mayo de 2024).

En Python, hay varios tipos de condicionales:

- **if**: Se utiliza para ejecutar un bloque de código si se cumple una condición específica.

La condición es una expresión booleana que se evalúa como verdadera (True) o falsa (False).

Se requiere el uso de dos puntos (:) al final de la condición.

Todas las líneas de código a ejecutar si se cumple la condición tienen que estar indentadas respecto la sentencia *if*.

```
# 1. Condicional 'if'
tiempo_Donostia = 'soleado'

if tiempo_Donostia == 'soleado':
  print('Voy a la playa!') # Resultado --> Voy a la playa!
```

- **if-else**: Se utiliza para ejecutar un bloque de código en el que se quieren comprobar múltiples condiciones y definir un curso de acción para cada una.

La sintaxis es similar a la anterior para que se cumpla la condición de la variable "tiempo_Donostia" y en caso de que no se cumpla se debe añadir la expresión *else* seguida de dos puntos (:) y a continuación lo que debe suceder en caso de que no se cumpla la condición inicial. Las dos expresiones condicionales *if*/*elese* deben estar intentadas al mismo nivel. En el caso del ejemplo, como está nublado en Donostia la respuesta debe ser "'Me quedaré en casa descansando".

```
# 2. Condicional 'if-else"
tiempo_Donostia = 'nublado'

if tiempo_Donostia == 'soleado':
  print('Voy a la playa!')
else:
  print('Me quedaré en casa descansando') # Resultado --> Me quedaré en casa
descansando
```

- **if-elif-else**: Se utiliza para ejecutar diferentes bloques de código según diferentes condiciones.

La sintaxis que usa es similar a la de *if-else* pero se añade una condición más con la expresión *elif* que hace referencia a un requisito adicional que la condición inicial debe satisfacer. Las tres expresiones condicionales *if*/ *elif* /*elese* deben estar intentadas al mismo nivel. Tomaremos como ejemplo, la temperatura que hace en un día en Donostia, consideramos que si la temperatura está entre 15 y 40 ºC es adecuada para ir a la playa, de lo contrario, establecemos dos condiciones para decir bien que hace frío para ir o que hace demasiado calor. En el ejemplo 30 ºC está dentro de la temperatura y por tanto es una temperatura adecuada para ir a la playa.

```
# 3. Condicional 'if-elif-else'
temperatura_Donostia = 30

if temperatura_Donostia < 15:
  print(f'Hace frío para ir a la playa')
elif temperatura_Donostia > 40:
  print(f'Hace demasiado calor para ir a la playa')
else:
  print(f'{temperatura_Donostia}ºC es una temperatura adecuada para ir a la
playa')
  # Resultado --> 30ºC es una temperatura adecuada para ir a la playa
```

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

Python proporciona dos tipos de bucles: *for* y *while*.[3]

Bucle *for*:

Se utiliza para iterar sobre una secuencia ordenada (como una lista, tupla, diccionario, conjunto o cadena).

La sintaxis del bucle *for* en Python es como sigue:

for var-a in lista-x:
    Sentencia (acción)

Se utiliza cuando se conoce el número de iteraciones que debe realizar el bucle o para recorrer los elementos de una secuencia ordenada, por ejemplo en una cadena como la del abecedario:

```
# Por ejemplo en una colección de caracteres
alfabeto = 'abcdef'

for letras in alfabeto:
  print(letras)
# Resultado:
# a
# b
# c
# d
# e
# f
```

A continuación veremos otro ejemplo del bucle for con una colección del tipo diccionario con el formato par clave-valor:

[3] Abder-Rahman, A. Programación.net [Blog]. Recuperado de: https://programacion.net/articulo/como_funcionan_los_bucles_en_python_1508 (20 de mayo de 2024).

```
# Ejemplo con una lista tipo diccionario, par clave:valor:
jugadores = {
  'Guardameta': 'Remiro',
  'Defensa': 'Oriozola',
  'Centrocampista': 'Merino',
  'Atacante': 'Becker'
}

# Iterará sobre el par clave-valor en la colección del diccionario:
for posicion, jugador in jugadores.items():
  print('Nombre del jugador:', jugador)
  print('Posición', posicion)
  # Resultado:
# Nombre del jugador: Remiro
# Posición Guardameta
# Nombre del jugador: Oriozola
# Posición Defensa
# Nombre del jugador: Merino
# Posición Centrocampista
# Nombre del jugador: Becker
# Posición Atacante
```

Bucle *while*:

Se ejecuta mientras se cumpla una condición específica. Se ejecuta repetidamente hasta que la condición, es útil cuando no se conoce el número máximo de iteraciones necesarias. En este tipo de iteración, siempre y cuando la prueba se evalúe como true, la declaración o bloque de instrucciones se seguirán ejecutando. Por lo tanto, el flujo ejecutará todas las sentencias dentro del bloque de bucle y si el condicionante del mismo es false, se ejecutarán las siguientes sentencias después del *while*. Si el condicionante da siempre como resultado true, en ese caso, obtendremos lo que se denomina como un bucle infinito.

```
# Ejemplo con bucle while
flores = 1
while flores <= 10:
    print('Riega la flor # ' + str(flores))
    flores += 1

"""
#Resultado:
Riega la flor # 1
Riega la flor # 2
Riega la flor # 3
Riega la flor # 4
Riega la flor # 5
Riega la flor # 6
Riega la flor # 7
```

```
Riega la flor # 8
Riega la flor # 9
Riega la flor # 10
"""
```

Los bucles en Python son útiles por varias razones:

1. **Automatización de tareas repetitivas**: Permiten ejecutar un bloque de código varias veces sin tener que escribir el mismo código una y otra vez.
2. **Implementación de algoritmos complejos**: Facilitan la implementación de algoritmos que requieren repetición de pasos.
3. **Flexibilidad y control**: Ofrecen flexibilidad y control sobre la repetición de código, adaptándose a diferentes situaciones.
4. **Legibilidad y concisión**: Hacen que el código sea más legible y conciso, especialmente cuando se conoce el número de iteraciones.

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

Una lista por comprensión es una forma sencilla y compacta para crear una lista a partir de una cadena (‘*string*’) o de otra lista. Es considerada una forma más rápida de proceder que mediante el uso de bucles.

La sintaxis para crear una lista por comprensión consiste en partir de nombrar la nueva variable donde deseamos almacenar la lista alterada o con una operación matemática, seguida del igual, luego escribir el argumento de la variable entre corchetes [], dentro del corchete se comienza primero con la operación que deseamos ejecutar, por ejemplo duplicar el valor de la lista inicial al tiempo que asignamos una variable a los elementos sobre los que queremos incidir (num), a continuación utilizamos la sintaxis de un bucle *for*, esto es for (variable) *in* (lista).

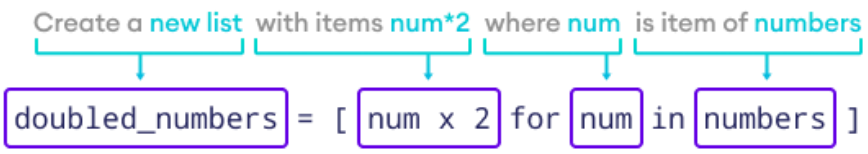

Imagen tomada de TutorialsTeacher[4]

[4] TutorialsTeacher (2024). Python - List Comprehension [Blog]. Recuperado de: https://www.tutorialsteacher.com/python/python-list-comprehension (20 de mayo de 2024)

```python
# Ejemplo de lista por comprensión
numeros = [1, 2, 3, 4]

# Uso de listas por comprensión para crear una nueva lista
duplicar_numeros = [num * 2 for num in numeros]

print(duplicar_numeros)
# Resultado:
# [2, 4, 6, 8]
```

Se podrían escoger los números pares de una lista generada a partir de un determinado rango de valores, por ejemplo los números pares del 1 al 10:

```python
# Definimos una variable donde se almacenarán los elementos de la lista
num_list = range(1, 11)

# 1. Forma tradicional
even_numbers = []

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

print(even_numbers) # --> [2, 4, 6, 8, 10]

# 2. Utilizando lista por comprensión
even_numbers = [num for num in num_list if num % 2 == 0]

print(list(num_list)) # --> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(even_numbers) # --> [2, 4, 6, 8, 10]
```

Como se puede apreciar el uso de la lista por comprensión hace un uso más eficiente del código (conciso y legible), solo en dos líneas realizamos el mismo procedimiento que usando un bucle *for* con un condicional. Las listas por comprensión son una herramienta poderosa para crear listas de manera eficiente y legible.

4. ¿Qué es un argumento en Python?

Un argumento es un valor que se pasa a una función cuando se la llama. Las funciones pueden tener cero o más argumentos, y estos argumentos se utilizan dentro de la función para realizar operaciones o cálculos específicos. Los argumentos pueden ser de diferentes tipos y se utilizan para personalizar el comportamiento de las funciones y permitir una mayor flexibilidad en la programación. La sintaxis básica para crear una función es comenzar por la expresión *def* y a continuación asignar un nombre a la función que deseamos crear

(*full_name*) seguida de paréntesis en donde se definen los nombres de los argumentos que queremos asignar a las variables operativas de la función (*first* y *last*), al final el cierre de la línea debe llevar dos puntos pues a partir de allí se define lo que debe realizar la función, en este caso el nombre completo.

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

# Set up named arguments
full_name(last = 'Hudgens', first = 'Kristine')
# Result: Kristine Hudgens
```

Como la función tiene dos argumentos (*first* y *last*), yo podría llamar la función más adelante en mi programa, añadiendo como en el ejemplo dentro de la función los argumentos con sus valores específicos sin importar el orden y resultaría en el nombre con el orden adecuado. Esto resulta útil cuando tenemos varios argumentos, sabemos que ellos existen dentro de la función pero no recordamos su orden. Aparte de este tipo de argumentos que llevan acompañados un nombre concreto para cada argumento, hay otros tipos de argumento que Python permite usar:

Argumentos posicionales:

Los argumentos posicionales son aquellos que se pasan a la función en el orden en que se definen en la declaración de la función. Por ejemplo, en la función anterior si en lugar de definir los nombres para cada argumento como (*first* y *last*) llamando en la función el argumento con su valor: full_name(last = 'Hudgens', first = 'Kristine') para obtener el nombre con su apellido deberíamos escribir full_name('Kristine', 'Hudgens'), de este modo y guardando el orden o posicion de los argumentos al definir la función resulta el mismo valor. A continuación otro ejemplo en el que se llama la función greeting('Fredy', 28) y debe recordarse que la primera posición corresponde a el nombre y la segunda a la edad.

```
def greeting(name, age):
    print(f"Hi, my name is {name} and I'm {age} years old. ")


greeting('Fredy', 28)
#Result: Hi, my name is Fredy and I'm 28 years old
```

Argumentos por defecto (default):

Los argumentos por defecto permiten especificar un valor predeterminado para un parámetro en caso de que no se proporcione un argumento al llamar a la función. Por ejemplo:

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


greeting()# --> Hi Guest!

greeting('Kristine') # --> Hi Kristine!
```

Para definirlo en la función se escribe el nombre del argumento, el símbolo igual y el valor por defecto que se le asignará a esa variable dentro del programa (name = 'Guest'). Para llamar al valor por defecto se debe pasar como argumento el paréntesis vacío como se ve en el ejemplo.

Argumentos arbitrarios (*arg):

Los argumentos arbitrarios permiten pasar un número variable de argumentos a una función. Se representan con un asterisco (*) seguido del nombre del parámetro. Dentro de la función, se tratan como una tupla. De este modo yo puedo siempre tener argumentos variados en la función, en el ejemplo a continuación el primer llamado de la función tienen tres argumentos y el segundo llamado solo dos pero el programa comprende ambos debido al empaquetado de los argumentos, lo que lo convierte en un elemento muy flexible.

La sintaxis de este tipo de argumento es como sigue:

1. Al definir la función en lugar de indicar argumentos concretos utilizamos el comodín asterisco (*)
2. Convención común: llamamos a estos argumentos arbitrarios *arg* (podría ser cualquier palabra clave con el asterisco delante)
3. Imprima usando la interpolación de cadenas estándar
4. Utilizamos luego una cadena vacía ' ' seguido de punto y de la palabra join con args sin el asterisco de la definición: + ‘ ‘ .join(args)
   --> join permite tomar una colección y luego la une, la convierte en una cadena y luego combina cualquier cosa con la que comenzamos.

```
def greeting(*args):
  print('Hi ' + ' '.join(args))


greeting('Kristine', 'M', 'Hudgens') # --> Hi Kristine M Hudgens
greeting('Tiffany', 'Hudgens') # --> Hi Tiffany Hudgens
```

Argumentos con palabras clave (keyword, kw):

Los argumentos de palabras clave arbitrarios permiten pasar un número variable de argumentos con palabras clave a una función. Se representan con dos asteriscos (**) seguidos del nombre del parámetro. Dentro de la función, se tratan como un diccionario par clave-valor {}. A continuación se muestra un ejemplo en el que se generan las palabras clave dentro de la respuesta de un condicional *if*:

```
def greeting(**kwargs):
    if kwargs:
        print(f"Hi {kwargs['first_name']} {kwargs['last_name']}, have a great
day!")
    else:
        print(f"Hi Guest, have a great day!")


greeting(first_name = 'Kristine', last_name = 'Hudgens')
#Result --> Hi Kristine Hudgens, have a great day!
greeting()
#Result (default) --> Hi Guest, have a great day!
```

La sintaxis que emplea es similar a la de los argumentos arbitrarios,

1. En la definición de la función se usan dos asteriscos seguidos de la palabra kwargs (que se tiene por convención de programación en Python aunque podría ser cualquier nombre)
2. Para generar el argumento (clave en un diccionario) lo defino entre llaves con la palabra kwargs {} y el nombre de dicho argumento en corchetes [], por ejemplo {kwargs[‘first_name’]}
3. Para llamar los argumentos (clave) que he generado lo hacemos como par clave-valor, first_name’ = ‘Kristine’

5. ¿Qué es una función Lambda en Python?

Una función lambda en Python es una función anónima y de una sola línea que se define utilizando la palabra clave lambda. Estas funciones son pequeñas, sin nombre, y se caracterizan por su capacidad de ser escritas en una sola línea de código, concentrando una funcionalidad específica en una expresión concisa. A diferencia de las funciones definidas con *def*, que pueden contener múltiples expresiones y sentencias, una función lambda se limita a una única expresión cuyo resultado es el valor de retorno de la función. De este modo la construcción lambda sirve para empaquetar procesos que pueden ser llamados posteriormente dentro de un programa

La sintaxis básica de una función lambda es simple: comienza con la palabra clave *lambda*, seguida de uno o más argumentos, dos puntos, y finalmente, la expresión que evaluará la función.

va. nombre = lambda + argumentos: valor a devolver

El valor a devolver es similar al *return* de una función escrita de forma convencional. A continuación mostramos un ejemplo para comparar las dos formas de ejecutar una función:

```
def greeting(name, age):
    print(f"Hi, my name is {name} and I'm {age} years old.")

greeting('Fredy', 28)
#Result: Hi, my name is Fredy and I'm 28 years old

# Using lambda:
greeting = lambda name, age: f"Hi, my name is {name} and I'm {age} years old."

print(greeting('Fredy', 28))
#Result: Hi, my name is Fredy and I'm 28 years old.
```

6. ¿Qué es un paquete pip?

Un paquete pip en Python es un conjunto de módulos y bibliotecas que pueden ser reutilizados en diferentes proyectos. Estos paquetes pueden proporcionar funcionalidades que van desde operaciones matemáticas hasta acceso a bases de datos y entre otros. Estos módulos o funcionalidades están pueden ser creados a nivel local por el usuario o descargados de repositorios donde se comparten para completar por ejemplo una acción en un programa o para obtener un efecto en una página web. '**pip**' es un acrónimo recursivo que significa pip installs packages o pip installs python. A modo de ejemplo, imaginemos que tenemos una función creada en Python y guardada tal y como sigue:

```
def greeting(first, last):
    return f'Hi {first} {last}'
# You build your own module that you can call anytime
```

El archivo se debe guardar en el mismo directorio donde guardamos el que supondremos como programa principal (main.py), en este ejemplo le asignamos el siguiente nombre "helper.py".

Para llamar a este paquete (pip) que hemos generado procedemos de la siguiente manera:

1. Debemos importar el programa del sistema: import sys

2. Importamos el paquete pip guardado anteriormente, le podemos asignar un alias dentro del programa principal (main.py), por ejemplo h de este modo: import helper as h
3. Luego definimos la función render():
4. Usamos la función teniendo en cuenta los argumentos que tenía en helper.py
5. Terminamos con el llamado de la función render()

```
import sys
import helper as h

def render():
    print(h.greeting('Tiffany', 'Hudgens'))


render()
# result --> Hi Tiffany Hudgens
```