Status	Finished
Started	Saturday, 23 November 2024, 5:19 PM
Completed	Saturday, 23 November 2024, 5:30 PM
Duration	11 mins 49 secs
Marks	6.00/7.00
Grade	**8.57** out of 10.00 (**85.71**%)

Information

Listas de tuplas

Las listas y las tuplas son tipos de datos estructurados que almacenan secuencias de elementos. Las listas son mutables y las tuplas son inmutables. Los elementos de una lista o de una tupla pueden ser de cualquier tipo, incluyendo (referencias a) tipos estructurados. Por ejemplo, podemos definir una lista de tuplas que representen coordenadas de puntos en el plano:

```
a = (0, 1)
b = (1, 1)
c = (2, -5)
d = (3, 7)
e = (4, 8)
points = [a, b, c, d, e]
print(points)              # [(0, 1), (1, 1), (2, -5), (3, 7), (4, 8)]
```

La variable *points* referencia a una lista que contiene 5 (referencias a) tuplas.

Podemos añadir elementos a la lista *points*, actualizarlos (sustituirlos), borrarlos (eliminarlos de la lista) o aplicar una operación de slice.

```
points.append((5, 0))
points.pop(0)
print(points)              # [(1, 1), (2, -5), (3, 7), (4, 8), (5, 0)]
points[2] = (1000, 1000)
print(points)              # [(1, 1), (2, -5), (1000, 1000), (4, 8), (5, 0)]
print(len(points))         # 5
alias = points
points[:] = points[::-1]
print(alias)               # [(5, 0), (4, 8), (1000, 1000), (2, -5), (1, 1)]
```

Podemos acceder a los elementos individuales de una tupla usando un segundo índice:

```
print(points[0])           # (5, 0) - la primera tupla de la lista
print(points[0][0])        # 5 - el primer elemento de la primera tupla
print(points[0][1])        # 0 – el segundo elemento de la primera tupla
```

Obviamente, dado que las tuplas son inmutables, la siguiente instrucción es errónea:

```
points[1][0] = 0           #... TypeError: 'tuple' object does not support item assignment
```

Question 1

Complete

Mark 1.00 out of 1.00

Dada la lista resultante del siguiente código:

```
visitors = [('Susan', 45, 'Slovakia', True), ('Peter', 33, 'Germany', False)]
visitors.append(('Martin', 18, 'Hungary', False))
```

Empareje cada uno de los siguientes trozos de código con la explicación que mejor lo describe.

`for name, age, country, ok in visitors:` `  print(name,"from",  country, "is", age, "years old.")`	Las tuplas en la lista son desempaquetadas a cuatro variables mientras se itera
`for v in visitors:` `  print(v)`	Se muestran los elementos de la lista visitors, uno por línea
`print(visitors)`	La lista de 3 tuplas se muestra en una línea

`sum = 0`
`for v in visitors:`
`  sum += v[1]`
`print('average age:', sum/len(visitors))`
	Se muestra la edad media de los visitantes

Information

Tablas como listas de listas

A menudo, necesitamos implementar una estructura bidimensional: una tabla con filas y columnas (una matriz de valores desde un punto de vista matemático). Dicha tabla de valores se puede crear utilizando listas de listas en Python:

```
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

o, en una forma más legible:

```
m = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Cada uno de los elementos `m[0]`, `m[1]`, `m[2]` referencia a una lista de 3 elementos. Otra forma de crear la misma estructura:

```
row1 = [1, 2, 3]
row2 = [4, 5, 6]
row3 = [7, 8, 9]
m =[row1, row2, row3]
```

El número de filas o columnas se puede conocer usando la función *len()*.

```
print(len(m))      # 3 - número de filas
```

La longitud de cualquier elemento, por ejemplo `m[0]`, nos da el número de columnas.

```
print(len(m[0]))   # 3 - número de columnas
```

Los elementos de la tabla se pueden acceder mediante una pareja de índices. La expresión `m[i][j]` representa el valor del elemento que está en la columna `j` de la fila `i`. El siguiente ejemplo cambia el valor de la primera columna de la segunda fila de la tabla `m` y muestra la tabla:

```
m[1][0] = 0
print(m)           # [[1, 2, 3], [0, 5, 6], [7, 8, 9]]
```

Question 2

Complete

Mark 1.00 out of 1.00

Marque todas las afirmaciones verdaderas acerca de a.

```
a = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Select one or more:

- [x] a[2] referencia a la última fila de la tabla
- [ ] len(a) == len(a[0])
- [x] a[1] referencia una lista de enteros
- [x] a es una lista de listas

Information

Recorrido de una tabla

Consideremos de nuevo la tabla m:

```
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Para recorrer una tabla, generalmente, usamos bucles for anidados:

```
for row in m:
    for x in row:
        print(x, end=' ')

    print()
```

El código anterior muestra el siguiente resultado:

1 2 3
4 5 6
7 8 9

A veces, es útil procesar los elementos de una tabla usando sus índices:

```
x = len(m) * len(m[0])

for i in range(len(m)):
    for j in range(len(m[0])):
        m[i][j] = x
        print(m[i][j], end=' ')
        x -= 1

    print()
```

El código anterior modifica la tabla y muestra el siguiente resultado:

9 8 7
6 5 4
3 2 1

Question 3

Complete

Mark 1.00 out of 1.00

¿Qué salida muestra el siguiente programa?

```
def print_table(tab):
    for row in tab:
        for element in row:
            print(element, end = " ")

        print()

t1 = ['a'] * 4
t2 = ['b'] * 4
t3 = ['c'] * 4
t = [t1, t2, t3]
t[0][2] = 'x'
print_table(t)
```

Select one:

```
a a x a
b b b b
c c c c
```

```
a a a a
b b b b
x c c c
```

```
4a 4a x 4a
4b 4b 4b 4a
4b 4c 4c 4c
```

Information

Mutabilidad

Cuando se crea una lista de listas hay que tener en cuenta que las listas son mutables, luego los elementos (de 2º nivel) de una lista de listas (o de una tupla de listas) son mutables.

Observe cómo se crea la tabla *m* en el siguiente ejemplo:

```
m = [[0, 0, 0]] * 3
```

Tiene tres filas con tres ceros cada una. Vamos a cambiar uno de los valores:

```
m[0][0] = -1
print(m) # [[-1, 0, 0], [-1, 0, 0], [-1, 0, 0]]
```

¿Por qué habrán cambiado los elementos *m[1][0]* y *m[2][0]*? Realmente la asignación:

```
m = [[0, 0, 0]] * 3
```

crea una lista con tres referencias a la misma lista `[0, 0, 0]`, igual que el siguiente ejemplo:

```
row = [0, 0, 0]
m = [row, row, row]
```

Podemos chequear si las filas son idénticas (si referenciamos el mismo objeto, o sea, si las referencias son iguales) con el operador ***is***:

```
m = [[0, 0, 0]] * 3
print(m[0] is m[1]) # True
```

Para crear una lista de listas de forma correcta debemos primero crear una lista vacía y luego añadirle distintas filas:

```
m = []
for i in range(3):
    m.append([0,0,0]) # en cada iteración [0, 0, 0] construye una lista distinta
```

Ahora todo está bien:

```
m[0][0] = -1
print(m)            # [[-1, 0, 0], [0, 0, 0], [0, 0, 0]]
print(m[0] is m[1]) # False
```

Question 4

Complete

Mark 1.00 out of 1.00

Necesitamos una tabla con 5 filas independientes y tres elementos por fila, inicializados a cero. ¿Cuáles de las siguientes opciones son correctas?

Select one or more:

-
```
a = []
for i in range(5):
    a.append([0] * 3)
```

-
```
a = [ [0, 0, 0] for i in range(5) ]
```

- a = [[0, 0, 0]] * 5

Information

Usando funciones para crear estructuras 2D

El uso de funciones puede ser muy útil para crear estructuras 2D, como listas de listas. La función *create_table()* del siguiente ejemplo toma las dimensiones de la tabla a crear (filas y columnas) y un valor de inicialización:

```
def create_table(rows, columns, value=0):
    t = []
    for i in range(rows):
        t.append([value] * columns)
    return t

my_table = create_table(2, 3)
print(my_table)                         # [[0, 0, 0], [0, 0, 0]]
my_table = create_table(4, 2, '*')
print(my_table)                         # [['*', '*'], ['*', '*'], ['*', '*'], ['*', '*']]
```

La siguiente función *create_table2()* primero crea una lista de tantos elementos como indique el parámetro *rows* inicializada con valores *None*, por lo que aún no referencian ninguna fila. A continuación se crean las filas y se asigna su referencia a cada elemento de la lista inicial, sustituyendo al valor *None*.

```
def create_table2(rows, columns, value=0):
     t = [None] * rows
     for i in range(rows):
        t[i] = [value] * columns
     return t

my_table = create_table2(3, 5, 1)
print(my_table)                      # [ [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]
```

Question 5

Complete

Mark 0.00 out of 1.00

¿Cuál de las siguientes funciones crea una copia de una tabla de forma correcta?

```
def copy_table1(tab):
    t = []
    for row in tab:
        t.append(row)
    return t

def copy_table2(tab):
    t = []
    for row in tab:
       t.append(list(row))
    return t

m = [[1, 2, 3],[4, 5, 6], [7, 8, 9]]
m1 = copy_table1(m)
m2 = copy_table2(m)
```

Select one:

- copy_table1() - m1 es una copia de m
- copy_table2() - m2 es una copia de m

Information

Uso de *comprehension* para crear listas de listas

Se pueden crear estructuras 2D usando comprensión:

```
t = [(i, j) for i in range(2) for j in range(3)]
print(t) # [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Nótese, que en el ejemplo anterior, cada elemento es una tupla.

```
t = [[0] * 3 for i in range(4)]
print(t) # [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

En el ejemplo anterior, cada fila se crea independientemente (en cada iteración se re-evalúa la expresión de la izquierda).

```
t = [list(str(2 ** i)) for i in range(10)]
print(t) # [['1'], ['2'], ['4'], ['8'], ['1', '6'], ['3', '2'], ['6', '4'], ['1', '2', '8'], ['2', '5', '6'], ['5', '1', '2']]
```

En el ejemplo anterior, las filas tienen diferentes longitudes, dado que 2^i va teniendo más dígitos al crecer el valor de i.

```
import random
t = [[random.randint(0, 9)] * 5 for i in range(3)]
print(t)
# [[8, 8, 8, 8, 8], [2, 2, 2, 2, 2], [7, 7, 7, 7, 7]]
```

En el ejemplo anterior todos los elementos de una misma fila tienen el mismo valor aleatorio (la lista inicial de un solo valor, aleatorio en este caso, se construye una sola vez, y luego se replica).

Question 6

Complete

Mark 1.00 out of 1.00

¿Cuál es la salida del siguiente programa?

```
words = ["to", "or", "not", "to", "be,", "that", "is", "the", "question."]
t = [list(w) for w in words if len(w) > 3]
print(t)
```

Select one:

- [['t', 'h', 'a', 't'], ['q', 'u', 'e', 's', 't', 'i', 'o', 'n', '.']]
- Ninguna de las opciones es correcta, se produciría un error.
- ['that', 'question.']

Information

Listas de listas de distintas longitudes

Hay muchas situaciones en las que necesitamos listas de listas de longitudes diferentes. La siguiente función toma como parámetros una secuencia con la longitud deseada para cada fila y un valor de inicialización:

```
def create_table(row_lengths, value=0):
    t = [None] * len(row_lengths)
    for i in range(len(t)):
        t[i] = [value] * row_lengths[i]
    return t
```

Primero se crea una lista conteniendo un cierto número de *None* que luego se sustituirán por las referencias a las filas que se irán creando de forma independiente, atendiendo a las longitudes indicadas por el primer parámetro.

Veamos cómo se usa:

```
t = create_table((1, 2, 3, 4, 5))
print(t)  # [[0],[0,0],[0,0,0],[0,0,0,0],[0,0,0,0,0]]

t = create_table(range(4), 'x'))
print(t) # [[], ['x'], [ 'x', 'x'], ['x', 'x', 'x']]
```

Nótese, que range(4) representa la secuencia de números 0, 1, 2, 3, por lo que la primera fila resulta una lista vacía en el segundo ejemplo.

Question 7

Complete

Mark 1.00 out of 1.00

Compare las siguientes funciones:

```
import random

def create_table1(row_lengths):
    t = [None] * len(row_lengths)
    for i in range(len(t)):
        t[i] = [random.randrange(2)] * row_lengths[i]
    return t

def create_table2(row_lengths):
    t = [None] * len(row_lengths)
    for i in range(len(t)):
        t[i] = [random.randrange(2) for i in range(row_lengths[i])]
    return t
```

Empareje las llamadas a las funciones anteriores con sus correspondientes resultados.

```
t = create_table1(range(1, 6))
print(t)
```

[[0], [1, 1], [0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0, 0]]

```
t = create_table2(range(1, 6))
print(t)
```

[[1], [0, 0], [0, 0, 0], [1, 0, 0, 1], [0, 0, 0, 1, 0]]

Information

Tratamiento estructurado de estructuras 2D

Cuando se quiere procesar una secuencia de elementos tratando cada elemento, se sigue un esquema básico como el siguiente:

```
for elemento in lista:
    tratar elemento
```

Cuando tenemos una secuencia de secuencias (por ejemplo una lista de listas), cada elemento de dicha secuencia es, a su vez, una secuencia que, típicamente, se procesa siguiendo un mismo esquema, lo cual da lugar a un patrón de bucles anidados:

```
for lista in lista_de_listas:
    for elemento in lista:
        tratar elemento
```

Una alternativa es derivar el tratamiento de las secuencias anidadas a una función auxiliar:

```
def tratar(lista):
    for elemento in lista:
        tratar elemento
```

```
for lista in lista_de_listas:
    tratar(lista)
```

Este esquema se aplica en el siguiente ejemplo:

```
def longest_words(list_of_list_of_words):
    """Dada una lista de listas de palabras, devuelve
       una lista con las palabras más largas de cada lista
    """
    words = []
    for list_of_words in list_of_list_of_words:
        words.append(longest_word(list_of_words))
    return words

def longest_word(list_of_words):
    """Devuelve la palabra más larga de una lista de palabras"""
    current_longest = ""
    for current_word in list_of_words:
        if len(current_word) > len(current_longest):
            current_longest = current_word
    return current_longest
```

La ventaja de separar cada tratamiento en funciones diferentes, en vez de tener una única función con bucles anidados, es que acabamos teniendo un conjunto de funciones muy simples, que siguen un esquema básico, y son más fáciles de desarrollar, entender y mantener que una función compleja, con bucles anidados. Nótese, que esto se puede extender a estructuras multidimensionales (listas de listas de listas de listas de ...).