Status	Finished
Started	Sunday, 10 November 2024, 9:54 AM
Completed	Sunday, 10 November 2024, 10:18 AM
Duration	23 mins 36 secs
Marks	5.00/5.00
Grade	**10.00** out of 10.00 (**100**%)

Information

Funciones recursivas (1)

Supongamos que queremos resolver el siguiente problema:

Desarrolle una función llamada **sum_ints** que tome como parámetro una lista con valores de diferentes tipos y devuelva la suma de los elementos de tipo *int* que haya en dicha lista.

Por ejemplo:

Recibe: [10, "Pedro", 42, "Margarita", 18.5, 8]

Devuelve: 10 + 42 + 8 = 60

Una posible solución es:

```
def sum_ints(the_list):
    """Suma los números enteros de una lista que
       contiene elementos de diferentes tipos.
    """
    result = 0

    for item in the_list:
        if type(item) == int:
            result += item

    return result
```

Information

Funciones recursivas (2)

Supongamos que, en el problema de sumar todos los números enteros de una lista que contiene elementos de diferentes tipos, algunos de esos elementos son, a su vez, listas:

```
lista = [10,  [25, 10, "Pedro"], 42, "Margarita", [12, "casa", 10], 8]
```

y que queremos incluir en la suma los números enteros que pudiera haber en las listas que forman parte de la lista original. En nuestra versión inicial de la solución, tratábamos los elementos que eran de tipo *int* e ignorábamos el resto. Sólo tenemos que modificarla para tratar las listas que nos encontremos, sumando los números enteros que contengan:

```
def sum_ints(the_list):
    """Suma los números enteros de una estructura que
       contiene elementos de diferentes tipos
    """
    result = 0

    for item in the_list:
        if type(item) == int:
            result += item
        elif type(item) == list:
            result += sum_ints(item)

    return result
```

Básicamente, lo que tenemos es que, si el elemento es de tipo *int*, lo sumamos al resultado, y, si el elemento es una lista, sumamos al resultado la suma de los números enteros que contenga. ¿Y cómo hacemos para calcular esa suma? Simplemente, aplicamos a la lista contenida el mismo algoritmo que estamos aplicando a la lista contenedora (el de la función *sum_ints*).

Cuando resolvemos una parte de un problema aplicando el mismo algoritmo que estamos usando para resolver el problema completo, decimos que estamos aplicando una **solución recursiva**.

Nótese, que tal como está planteada, la solución funciona también cuando algunas de las listas contenidas en la inicial tienen elementos que son, a su vez, listas, y así hasta cualquier nivel de anidamiento.

Question 1

Complete

Mark 1.00 out of 1.00

Una solución a un problema es recursiva si...

Select one:

- ... resuelve un problema que solo se puede resolver llamando a la propia función que aborda la solución del problema completo.
- ... contiene un bucle, generalmente una sentencia for, que incluye una llamada a una función en ciertos casos.
- ... para resolver alguna parte del problema aplica el mismo algoritmo que para resolver el problema completo.

Question 2

Complete

Mark 1.00 out of 1.00

¿Qué muestra el siguiente código?

```
def function(string):
    if len(string) <= 1:
        result = string
    else:
        result = string[-1] + function(string[1:-1]) + string[0]

    return result

print(function("contrario"))
```

Answer: oirartnoc

Question 3

Complete

Mark 1.00 out of 1.00

¿Qué muestra el siguiente código?

```
def function(a, b):
    if b == 1:
        return a
    elif b % 2 == 0:
        return function(a * 2, b // 2)
    else:
        return a + function(a * 2, b // 2)
```

```
print(function(5, 6))
```

Answer: 30

Information

Búsqueda de una solución recursiva: paso 1 - el tamaño del problema

Muchas veces, que apliquemos a un problema una solución recursiva, o no, depende de cómo se mire. Por ejemplo, supongamos que queremos desarrollar una función llamada *contains* a la que se le pase una lista de números y un número y devuelva *True* si el número está contenido en la lista y *False* si no lo está.

Si abordamos el problema con una "mentalidad iterativa", la solución es recorrer la lista comparando cada elemento con el valor buscado hasta que lo encontremos, o se nos acaben los valores con los que comparar:

```
def contains(container, value):
    """Determina si el valor value está en la lista container."""
    # Recorremos la lista comparando los elementos con el valor buscado
    for item in container:
        if item == value:
            return True  # Hemos encontrado el valor buscado
```

```
    # Hemos comparado con todos los elementos de la lista sin encontrar
    # el valor buscado
    return False
```

Si queremos buscar una solución recursiva, tenemos que buscar una forma de descomponer el problema consiguiendo que aparezcan subproblemas de la misma naturaleza.

Hay que tener en cuenta que los subproblemas incluidos en un problema tienen necesariamente que ser "más pequeños" que el problema que los incluye.

El primer paso para buscar una solución recursiva a un problema es averiguar qué factores determinan el "tamaño" del problema.

En el caso de nuestro ejemplo, los datos del problema son una lista y un número. Para encontrar si el número está en la lista habrá que irlo comparando con cada elemento de la lista. En el peor caso habrá que compararlo con todos los elementos, y en promedio con la mitad. La probabilidad de realizar más comparaciones aumenta cuanto más larga sea la lista. Eso es cierto independientemente de cuál sea el valor buscado; un valor menor no implica que hagamos menos comparaciones, ni uno mayor que hagamos más (a menos que la lista esté ordenada de forma creciente y lo aprovechemos deteniéndonos, cuando encontramos un valor mayor que el buscado).

En resumen, el valor buscado no influye en el trabajo a realizar, el tamaño de la lista sí lo hace; el tamaño del problema viene determinado, en este caso, por el número de elementos de la lista.

Siempre que estamos trabajando con secuencias (strings, tuplas, listas), en general podemos suponer que el tamaño del problema viene determinado por el número de elementos de la secuencia a tratar: normalmente, tardaremos 10 veces más en recorrer una lista de 1000 elementos que una de 100. Cuando el problema viene definido exclusivamente por datos numéricos, generalmente, números más grandes implican problemas más grandes; por ejemplo, la conocida función factorial se define como:

```
si n = 0, n! = 1
si n > 0, n! = n * (n - 1)!
```

6! > 5! > 4! > 3! > ··· en cuanto al trabajo a realizar (p.e. número de multiplicaciones).

Information

Búsqueda de una solución recursiva: paso 2 - el caso base

Cuando conocemos el factor que determina el tamaño del problema a resolver, podemos conocer cuál es el caso más pequeño del problema. ¿Por qué es esto importante? Podemos suponer que cuesta más solucionar un problema cuanto mayor es, por lo tanto, el caso más pequeño es el más simple de solucionar, ya que no depende de otros subproblemas recursivos de menor tamaño y además típicamente tiene una solución directa. Por ejemplo, en el cálculo del factorial, cuando n = 0 sabemos que su factorial es 1 sin necesidad de ningún cálculo (0 es el número más pequeño del dominio del problema del factorial, que sólo está definido para números enteros no negativos).

En el problema de buscar un valor en una lista, dado que hemos determinado que el tamaño del problema está directamente relacionado con la longitud de la lista, está claro que el caso más pequeño es cuando la lista tiene longitud 0, es decir, está vacía. ¿Y cuál es el resultado de la búsqueda cuando la lista está vacía? Evidentemente, *False*, dado que ningún elemento puede encontrarse en una lista vacía. Por tanto, podemos escribir

```
def contains(container, value):
    """Determina si el valor value está en la lista container"""
    if len(container) == 0:
        result = False
    else:
        # falta resolver el problema para el caso de len(container) > 0

    return result
```

El caso más pequeño de un problema recursivo se conoce como **caso base**, ya que es el que pone fin a la expansión de la recursividad, empezando así a retornar una solución parcial. Sin el caso base, la recursión se convertiría en infinita (teóricamente, en la práctica, nunca alcanzaríamos a calcular completamente el resultado buscado).

Information

Búsqueda de una solución recursiva: paso 3 - descomposición del problema

Una vez resuelto el caso base, hay que estudiar cómo descomponer el problema general, de modo que obtengamos problemas más pequeños, pero buscando que algunos de estos problemas sean de la misma naturaleza que el original y podamos operar recursivamente sobre ellos (si no tenemos esto en mente, estamos buscando una solución iterativa, no recursiva).

De esta forma, en última instancia, se acabará alcanzando el caso base, que nos proporcionará una solución parcial, a partir de la cual construir la solución del caso general del problema.

Podemos plantearnos que lo que buscamos es quitar uno o más "trozos" del problema, que podamos resolver más o menos directamente, de tal forma que lo que quede una vez quitado ese trozo sean subproblemas de la misma naturaleza que el original, pero más pequeños.

En el ejemplo que estamos desarrollando, ¿qué le podemos quitar a una lista para quedarnos con una lista más pequeña? Normalmente, definimos una lista como una "secuencia de elementos" que es una forma iterativa de verla (los problemas de secuencias se resuelven de forma natural iterando por los elementos de las mismas). Necesitamos una forma distinta de definir una lista. Una posibilidad (entre otras) es decir que una lista puede ser de dos formas, o sea, tenemos dos casos posibles:

- está vacía, o, si no,
- está formada por un elemento seguido por una lista.

Si partimos de esta definición de lista podemos, cuando no está vacía, simplemente, separar el primer elemento y lo que nos queda es una lista con el resto de los elementos:

```
        first = list[0]
        tail = list[1:]
```

El primer elemento podemos compararlo con el valor buscado; si coinciden, ya hemos resuelto el problema (se ha encontrado), si no, sólo tenemos que aplicar el mismo algoritmo al resto de la lista que hemos separado (o sea, resolver recursivamente un problema más pequeño) y devolver como resultado lo que nos depare esa búsqueda, sabiendo además, que de esta forma, si no se encontrase el valor buscado, en algún momento del proceso el resto acabará siendo una lista vacía e, igualmente, habremos entonces resuelto el problema (no se habrá encontrado). Por tanto, el **caso general** lo podemos implementar como sigue:

```
        if first == value:
            result = True
        else:
            result = contains(tail, value)
```

Lo cual podemos leer como: "para una lista no vacía (caso general), un valor se encuentra en ella si es igual al primero de la lista o, si no, si se encuenta en el resto de la lista".

Así pues, la solución completa, combinando casos base y general, quedaría como sigue (tras eliminar las variables *first* y *tail*, que no usamos más que una vez).

```
def contains(container, value):
    """Determina si el valor value está en la lista container."""
    if len(container) == 0:
        result = False
    else:
        if container[0] == value:
            result = True
        else:
            result = contains(container[1:], value)
```

```
    return result
```

Question 4

Complete

Mark 1.00 out of 1.00

Ordene las etapas en el desarrollo de una solución recursiva a un problema.

- Averiguar qué factores determinan el tamaño del problema.
- Determinar y resolver el caso base
- Averiguar cómo descomponer el problema para obtener subproblemas más pequeños.
- Resolverrecursivamente los subproblemas
- Componer ydevolverel resultado del caso general a partir de los resultados devueltos por los subproblemas más pequeños.

Information

Esquema de una solución recursiva

Ahora tenemos dos formas de resolver el problema de encontrar un valor en una lista:

Solución iterativa

```
def contains(container, value):
    result = False
    for item in container:
        if item == value:
            result = True
            break
    return result
```

Solución recursiva

```
def contains2(container, value):
    if len(container) == 0:
        result = False
    else:
        if list[0] == value:
            result = True
        else:
            result = contains2(list[1:], value)
    return result
```

La solución iterativa sigue el conocido esquema de recorrido de una secuencia. La solución recursiva muestra el esquema habitual de una solución de esta clase. La recursividad siempre divide el dominio del problema al menos en dos partes: caso base/caso general, por lo que en el algoritmo de la solución siempre estará presente una estructura de selección (*if*) para discriminar entre ambas partes. Podemos observar cómo ocurre esto igualmente en otros problemas clásicos:

Factorial de un número natural

```
si n = 0, n! = 1
```

```
si n > 0, n! = n * (n - 1)!
```

Solución recursiva

```
def factorial(n):
    if n == 0:
        result = 1
    else:
        result = n * factorial(n - 1)

    return result
```

n-ésimo número de Fibonacci

```
si n = 0, fibo(n) = 0
```

```
si n = 1, fibo(n) = 1
```

```
si n > 1, fibo(n) = fibo(n - 1) + fibo(n - 2)
```

Solución recursiva

```
def fibo(n):
    if n <= 1:
        result = n
    else:
        result = fibo(n - 1) + fibo(n - 2)
    return result
```

No obstante lo dicho, una vez que adquiramos soltura en su realización, soluciones recursivas como la mostrada al principio son simplificables obviando la variable result, al igual que ya sabemos hacer con la solución iterativa mostrada también al principio.

Question 5

Complete

Mark 1.00 out of 1.00

¿Cuál de las siguientes sentencias de control no puede faltar en un algoritmo recursivo?

Select one:

- for
- while
- if