UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ACG - Auditoría Calidad y Gestión de Sistemas

2024/2025

Francesco Lorenzoni
PCA25403GU

Practica 4

Pytest

Índice general

Capítulo 1

Trabajo

1.1. Ejercicios 1/2/3 - `sin_vocales`

El primero error en la función dada `sin_vocales(s)` es que no se tiene cuenta de las mayúsculas y minúsculas. Por lo tanto, la función no elimina las vocales mayúsculas. Para solucionarlo, es suficiente añadir a la lista de vocales la versión mayúscula de cada vocal.

```
def test_sin_vocales():
    s = "el agua esta mojada"
    exp = "l g st mjd"
    assert sin_vocales(s) == exp

    ...

    s = "El AgUe eStA mOjAdA"
    exp = "l g St mjd"
    assert sin_vocales(s) == exp
```

El segundo error es que la función no tiene cuenta de los acentos. Por lo tanto, la función no elimina las vocales acentuadas. Para solucionarlo, es suficiente añadir a la lista de vocales la versión acentuada de cada vocal.

```
vocales = 'aeiouAEIOUáéíóúàèìòùäëïöüâêîôûÁÉÍÓÚÀÈÌÒÙÄËÏÖÜÂÊÎÔÛãõÃÕñÑ'
```

Esta solución no parece muy elegante, ya que la lista de vocales se vuelve muy larga. Buscando sobre el internet he visto que una solución más elegante sería usar una expresión regular para eliminar todas las vocales, acentuadas o no. Para ello, se puede usar el módulo `re` de Python, junto con `unicodedata`, como se muestra en el código 1.1.

Listing 1.1: Código para eliminar vocales y diacríticos

```
import unicodedata
import re

def sin_vocales(s):
    # Normaliza el texto separando los
        caracteres básicos de sus diacríticos
    s_norm = unicodedata.normalize('NFD', s)
    # Elimina todas las vocales base y los
        diacríticos
    s_sin_vocales = re.sub(r'[aeiouAEIOU\
        u0300-\u036f]', '', s_norm)
    return s_sin_vocales
```

1.2. Ejercicio 5

Listing 1.1: Solución sencilla

```
def cantidad_numeros_sencilla(s: str):
    """
    Esta función recibe una cadena de texto y devuelve la cantidad de números que contiene.
    N.B. números, no digitos!
    """
    return len(re.findall(r'\d+', s))
```

Esta primera solución funciona y es concisa, pero no tiene en cuenta los números decimales (como 12,34) y las notaciones científicas (como $12 * 10^6$ o $10^(-2)$). Para solucionarlo, es necesario modificar la expresión regular para que también considere estos casos.

Listing 1.2: Solución que incluye números decimales y notaciones científicas

```
def cantidad_numeros(s: str):
    """
    Esta función recibe una cadena de texto y devuelve la cantidad de números que contiene.

    Números decimales (como 12.34) y notaciones científicas (como 12*10^6 o 10^(-2)) son
        considerados números.
    """
    # Identificar números decimales (como 12.34)
    decimal_pattern = r'-?\d+\.\d+'
    decimal_matches = re.findall(decimal_pattern, s)

    # Eliminar los números decimales ya encontrados para evitar contar dos veces
    for match in decimal_matches:
        s = s.replace(match, ' ', 1)

    # Identificar notaciones científicas (como 12*10^6 o 10^(-2))
    scientific_pattern = r'\d+\*10\^\(?\-?\d+\)?'
    scientific_matches = re.findall(scientific_pattern, s)

    # Eliminar las notaciones científicas de la cadena
    for match in scientific_matches:
        s = s.replace(match, ' ', 1)

    # Encontrar los números restantes (enteros)
    # Modificamos el patrón para capturar secuencias de dígitos en cualquier contexto
    remaining_pattern = r'\d+'
    remaining_matches = re.findall(remaining_pattern, s)

    # Contar el total de números encontrados
    return len(decimal_matches) + len(scientific_matches) + len(remaining_matches)

def test_cantidad_numeros():
    s = "12 356 53333"
    assert cantidad_numeros(s) == 3
    s = "asfa432asf23"
    assert cantidad_numeros(s) == 2
    s = ""
    assert cantidad_numeros(s) == 0
    s = "1"
    assert cantidad_numeros(s) == 1
    s = "fwgds"
    assert cantidad_numeros(s) == 0
    s = "1 fhsdSGG 4"
    assert cantidad_numeros(s) == 2
    s = "-12 + 34 -12-133 "
    assert cantidad_numeros(s) == 4
    s = "12.34 56.78"
    assert cantidad_numeros(s) == 2
    s = "12*10^6"
    assert cantidad_numeros(s) == 1
    s = "23*10^(-2)"
    assert cantidad_numeros(s) == 1
```

```
    s = "23*10^(64) + 12.34"
    assert cantidad_numeros(s) == 2
    s = "asgre23*10^(-2)asfg10^2"
    assert cantidad_numeros(s) == 2
    s = "asgre23*10^(-2)asfg10^2agr12.34"
    assert cantidad_numeros(s) == 3
```

1.3. Ejercicio 6

```
@pytest.mark.parametrize ("entrada , salida_esperada " ,[
   ("El agua esta mojada", "l g st mjd"),
   ("mojada bañando en el agua", "mjd bñnd n l g"),
   ("ahora termina bien", "hr trmn bn"),
   ("", ""),
   ("a", ""),
   ("m", "m"),
   ("unstringsinespacios", "nstrngsnspcs"),
   ("MAYUSculas FUNCIOnaN", "MYScls FNCnN"),
   ("krt yhgf dwpq", "krt yhgf dwpq"),
   ("aeoiuuuoiea", ""),
   ("disco de los 80", "dsc d ls 80"),
   ("signos como ? y ! y ą", "sgns cm ? y ! y ą"),
   ("ábc élla ó", "bc ll "),
   ("Óm tambien mayŰsculÁs", "m tmbn myscls")
   ])

def test_sin_vocales_parametrizado(entrada, salida_esperada):
   """
   testea la función sin_vocales
   """
   assert sin_vocales (entrada) == salida_esperada
```

Esta es una forma alternativa de escribir los test, que permite de parametrizar la función driver de test.

En comparación con el test set anterior del pdf, hemos añadido pruebas que incluyen vocales con acento. La función `sin_vocales` no va a cambiar porque ya había tenido en cuenta los acentos en el ejercicio anterior.