IBM **SkillsBuild**

Pruebas con Python

Testeando scripts con Unittest

Índice

Introducción

En este módulo veremos tres casos de uso de testeo de scripts con Unittest.

Ejemplo 1. Binario a decimal

Convierte cadenas binarias a sus equivalentes decimales. Lanzar **ValueError** si **binary_str** contiene caracteres distintos de **0** y **1**

Archivo **bin_to_dec.py**:

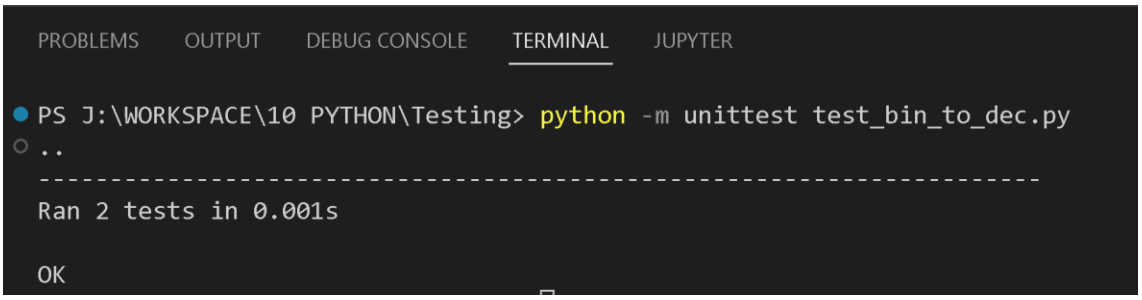
```
def decimal(binary_str):

    """ Convierte cadenas binarias a sus equivalentes decimales.
    Lanzar ValueError si binary_str contiene caracteres distintos de 0 y 1"""

    remove_0_and_1 = binary_str.replace('0', '').replace('1', '')
    if len(remove_0_and_1) > 0:
        raise ValueError('La cadena binaria de entrada solo puede contener 0 y 1')


    place = 1; # Posición
    dec = 0   # El valor decimal

    for bit in binary_str[::-1]: # Bucle desde el final de la cadena hasta el principio
        if (bit == '1'): # Si el dígito es un 1, agregue el valor posicional. Si es 0,
ignorar.
            dec += place
        place *= 2  # Multiplique la posición por 2 para el siguiente valor de posición

    return dec
```

Archivo **test_bin_to_dec.py**:

```
import unittest
import bin_to_dec

class TestBinaryToDecimal(unittest.TestCase):

    def test_binario_decimal_con_entradas_validas(self):

        # El método bin de Python hace la conversión de binario a decimal

        # Los bucles son útiles: testeamos un rango de números
        for d in range(100):
```

```python
            binary = bin(d)  # En formato '0b10101'
            binary = binary[2:]  # Quitar la inicial '0b'

            dec_output = bin_to_dec.decimal(binary)

            self.assertEqual(d, dec_output)

        # Testeamos algunos números más grandes

        test_vals = [4000, 4001, 4002, 1024, 1099511627776, 1099511627777, 1099511627775]

        for d in test_vals:
            binary = bin(d)  # En formato '0b10101'
            binary = binary[2:]  # Quitar la inicial '0b'

            dec_output = bin_to_dec.decimal(binary)

            self.assertEqual(d, dec_output)


        # Test con strings
        test_bin_str = [ '101010', '1111', '000111', '0', '1']
        expected_dec = [ 42, 15, 7, 0, 1]

        for binary_input, expected_dec_output in zip( test_bin_str, expected_dec) :
            dec = bin_to_dec.decimal(binary_input)
            self.assertEqual(dec, expected_dec_output)


    def test_binario_decimal_con_entradas_invalidas(self):

        # Testeamos que se genere un error con cadenas que no estén compuestas por 0 y 1.

        valid = '010101'
        valid2 = '1111111'

        invalid = [ '123456', '101010012', 'abc', '@#$%$%^%^&']
        for invalid_input in invalid:
            with self.assertRaises(ValueError):
                bin_to_dec.decimal(invalid_input)


if __name__ == '__main__':
    unittest.main()
```

Resultado del test:



```
PS J:\WORKSPACE\10 PYTHON\Testing> python -m unittest test_bin_to_dec.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Ejemplo 2. Camel case

Un diccionario de entradas con salidas esperadas. También utiliza parches para parchear las funciones integradas de entrada y salida para probar la entrada del usuario y la salida correcta que se está imprimiendo. Observe el uso del administrador de contexto de parches, que se encarga de quitar los parches cuando haya terminado. De lo contrario, es posible que deba reemplazar las funciones de entrada/impresión originales.

Archivo **camel.py**:

```
import re

def capitalize(word):
    """ Convierta la palabra para que tenga la primera letra en mayúscula, el resto en
minúsculas"""
    return word[0:1].upper() + word[1:].lower()
    # Los segmentos no producen errores de tipo "index out of bounds".
    # Así que esto todavía funciona en cadenas vacías y cadenas de longitud 1


def lowercase(word):
    """convierte una palabra a minúsculas"""
    return word.lower()


def camel_case(sentence):

    remove_multiple_spaces = re.sub(r'\s+', ' ', sentence)  # Reemplaza cualquier grupo
de espacios en blanco con un solo espacio
    remove_surrounding_space = remove_multiple_spaces.strip()  # elimina cualquier
espacio en blanco restante

    words = remove_surrounding_space.split(' ') # Segmenta por espacios
    first_word = lowercase(words[0])  # Pasa a minúsculas la primera palabra

    # Escribe con mayúscula la segunda palabra y las siguientes y las pone en una nueva
lista.
    capitalized_words = [ capitalize(word) for word in words[ 1: ] ]

    # Reúne todas las palabras en una lista
    camel_cased_words = [first_word] + capitalized_words
    # Vuelve a juntar las palabras
    camel_cased_sentance = ''.join(camel_cased_words)

    return camel_cased_sentance


def main():
    sentence = input('Introduzca la frase: ')
    camelcased = camel_case(sentence)
    print(camelcased)


if __name__ == '__main__':
    main()
```

Archivo **test_camel.py**:

```
import unittest
from unittest.mock import patch
import camel

class TestCamelCase(unittest.TestCase):

    def test_capitalize(self):

        input_words = ['abc', 'ABC', 'aBC', 'ABc']
        capitalized = 'Abc'

        for word in input_words:
            self.assertEqual(capitalized, camel.capitalize(word))


    def test_lower(self):
        # this isn't really needed, since we can assume that Python's library functions
work correctly :)
        input_words = ['abc', 'ABC', 'aBC', 'ABc']
        lower = 'abc'

        for word in input_words:
            self.assertEqual(lower, camel.lowercase(word))


    def test_camel_case_single_words(self):

        input_and_expected_outputs = {
            'hello' : 'hello',
            'Hello' : 'hello',
            'Thisisaverylongwordlalalalalalalalalalala':
'thisisaverylongwordlalalalalalalalalalala',
            'a': 'a'
        }

        for input_val, output_val in input_and_expected_outputs.items():
            self.assertEqual(output_val, camel.camel_case(input_val))


    def test_camel_case_uppercase(self):

        input_and_expected_outputs = {
            'HELLO': 'hello',
            'Hello': 'hello',
            'HeLLo wORlD': 'helloWorld'

        }

        for input_val, output_val in input_and_expected_outputs.items():
            self.assertEqual(output_val, camel.camel_case(input_val))


    def test_camel_case_lowercase(self):

        input_and_expected_outputs = {
            'hello': 'hello',
            'hELLO': 'hello',
            'heLLo WORlD': 'helloWorld'
        }

        for input_val, output_val in input_and_expected_outputs.items():
            self.assertEqual(output_val, camel.camel_case(input_val))


    def test_camel_case_empty_strings(self):

        input_and_expected_outputs = {
            '': '',
            '         ': '',
        }

        for input_val, output_val in input_and_expected_outputs.items():
            self.assertEqual(output_val, camel.camel_case(input_val))


    def test_camel_case_many_words(self):

        input_and_expected_outputs = {
            'two words': 'twoWords',
            'this is a sentence': 'thisIsASentence',
            'Here is a long sentence with many words' :
'hereIsALongSentenceWithManyWords',
        }

        for input_val, output_val in input_and_expected_outputs.items():
            self.assertEqual(output_val, camel.camel_case(input_val))


    def test_camel_case_extra_spaces(self):

        input_and_expected_outputs = {
            '  Spaces Before': 'spacesBefore',
            'Spaces after   ': 'spacesAfter',
            '   Spaces     Every     where    ': 'spacesEveryWhere',
            '\tThere is a \t tab here': 'thereIsATabHere',
            '\nThere is a \n newline here': 'thereIsANewlineHere',
            'There is a newline here\n': 'thereIsANewlineHere',
            '\nThere is a newline here\n': 'thereIsANewlineHere',

        }

        for input_val, output_val in input_and_expected_outputs.items():
            self.assertEqual(output_val, camel.camel_case(input_val))


    def test_camel_case_emojis(self):

        input_and_expected_outputs = {
            '👽🌎🌺': '👽🌎🌺',
            '👽  🌎🌺🐑🌳   🌵🐬': '👽🌎🌺🐑🌳🌵🐬',
        }

        for input_val, output_val in input_and_expected_outputs.items():
            self.assertEqual(output_val, camel.camel_case(input_val))


    def test_camel_case_international(self):

        input_and_expected_outputs = {
            '你叫 什么 名字': '你叫什么名字',
            'Write a résumé': 'writeARésumé',
            'Über die Brücke': 'überDieBrücke',
            'Fahre über die Brücke': 'fahreÜberDieBrücke',
        }

        for input_val, output_val in input_and_expected_outputs.items():
            self.assertEqual(output_val, camel.camel_case(input_val))
    def test_input_and_output(self):

        # Patch the input. Using with context manager automatically takes care of
unpatching.
        with patch('builtins.input', return_value='This IS another SENTenCE'):

            # And, patch the output
            with patch('builtins.print') as mock_print:

                camel.main()
                mock_print.assert_called_with('thisIsAnotherSentence')



if __name__ == '__main__':
    unittest.main()
```

Ejemplo 3. Reciclaje

Eres un conductor de camión de reciclaje. Le gustaría recopilar algunas estadísticas sobre cuánto recicla cada casa. Suponga que los números de las casas son 0, 1, 2, 3...

Cada casa puso su reciclaje en cajas.

Calcula el número de la casa con la mayor cantidad de reciclaje y la cantidad de cajas que recicla la casa. Lo mismo para la casa con menos y el número de cajas para esa casa.

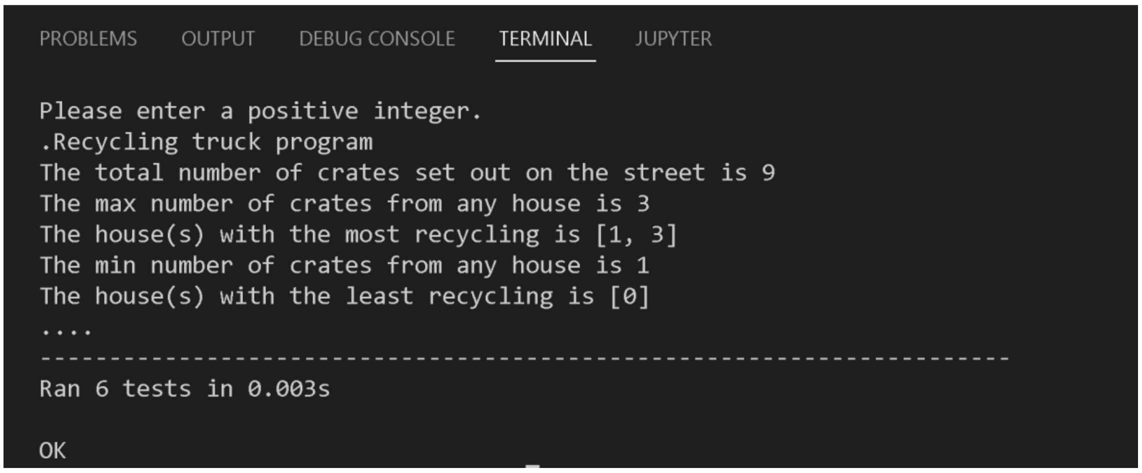
Archivo **recycling.py**:

```python
from collections import namedtuple

CrateData = namedtuple('CrateData', ['houses', 'crates'])


def max_recycling(crates):
    """Returns the index with the largest value in the list and the number of crates for
that house.
    Raises ValueError if list is empty."""

    if crates is None or len(crates) == 0:
        raise ValueError('A list with at least one element is required')

    max_houses = []
    max_crates = crates[0]

    for crate in crates:
        if crate > max_crates:
            max_crates = crate

    for house, crates in zip (range(len(crates)), crates):
        if crates == max_crates:
            max_houses.append(house)

    return CrateData(max_houses, max_crates)


def min_recycling(crates):
    """Returns the smallest value in the list
    and a list of house number (list indexes) with that value.
    Raises ValueError if list is None or empty."""

    if crates is None or len(crates) == 0:
        raise ValueError('A list with at least one element is required')

    min_houses = []
    min_crates = crates[0]
    for crate in crates:
        if crate < min_crates:
            min_crates = crate

    for house, crates in zip (range(len(crates)), crates):
        if crates == min_crates:
            min_houses.append(house)

    return CrateData(min_houses, min_crates)


def total_crates(crates):
    """ Return the total of all the values in the crates list"""

    total = 0
    for crate in crates:
        total += crate
    return total


def get_crate_quantities(houses):
    """ Ask user for number of crates for each house"""
    crates = []
    for house in range(houses):
        crates.append(positive_int_input('Enter crates for house {}'.format(house)))
    return crates


def positive_int_input(question):
    """ Valdiate user enters a positive integer """
    while True:
        try:
            integer = int(input(question + ' '))
            if integer >= 0:
                return integer
            else:
                print('Please enter a positive integer.')

        except ValueError:
            print('Please enter a positive integer.')


def main():

    print('Recycling truck program')

    houses = positive_int_input('How many houses?')

    crates = get_crate_quantities(houses)
    maximums = max_recycling(crates)
    minimums = min_recycling(crates)

    total = total_crates(crates)

    print('The total number of crates set out on the street is {}'.format(total))
    print('The max number of crates from any house is {}'.format(maximums.crates))
    print('The house(s) with the most recycling is {}'.format(maximums.houses))

    print('The min number of crates from any house is {}'.format(minimums.crates))
    print('The house(s) with the least recycling is {}'.format(minimums.houses))



if __name__ == '__main__':
    main()
```

Archivo **test_recycling.py**:

```
import unittest
from unittest.mock import Mock, patch

import recycling

class TestRecycling(unittest.TestCase):


    def test_max_values(self):

        # More than one house with the same max value
        example_data = [1, 3, 5, 0, 2, 6, 3, 6]
        max_data = recycling.max_recycling(example_data)
        self.assertEqual(max_data.crates, 6)
        self.assertEqual(max_data.houses, [5, 7])

        # Single max value
        example_data = [1, 3, 9, 0, 2, 3, 3, 6]
        max_data = recycling.max_recycling(example_data)
        self.assertEqual(max_data.crates, 9)
        self.assertEqual(max_data.houses, [2])


    def test_min_values(self):

        # More than one joint min value
        example_data = [1, 0, 3, 5, 0, 2, 6]
        min_data = recycling.min_recycling(example_data)
        self.assertEqual(min_data.crates, 0)
        self.assertEqual(min_data.houses, [1, 4])

        # Single min value
        example_data = [1, 3, 5, 0, 2, 6]
        min_data = recycling.min_recycling(example_data)
        self.assertEqual(min_data.crates, 0)
        self.assertEqual(min_data.houses, [3])


    def test_total(self):
        example_data = [1, 3, 5, 0, 2, 6]
        self.assertEqual(recycling.total_crates(example_data), 17)


    def test_get_crate_quantities(self):

        """
        Create a patch to replace the built in input function with a mock.
        The mock is called mock_input, and we can change the way it behaves, e.g. provide
        our desired return values. So when the code calls input(), instead of
        calling the built-in input function, it will call the mock_input mock function,
        which doesn't do anything except for returning the values provided in the
        list of side_effect values - the first time it is called, it returns the first
        side_effect value (1), second time it will return the second value, (3) etc...

        """


        example_data = [1, 3, 5, 0, 2, 6]
        with patch('builtins.input', side_effect=example_data) as mock_input:
            self.assertEqual(recycling.get_crate_quantities(6), example_data)


    def test_int_input(self):

        # Test with some invalid input
        # Put a valid input at the end or the function will never return
        with patch('builtins.input', side_effect=['-2', '-1000', 'abc', '123abc', '3'])
as mock_input:
            self.assertEqual(recycling.positive_int_input('example question'), 3)
#Ultimately, should return the valid value at the end of the list.


        with patch('builtins.input', side_effect=[ '0', '13', '1', '100000000']) as
mock_input:
            self.assertEqual(recycling.positive_int_input('example question'), 0)
            self.assertEqual(recycling.positive_int_input('example question'), 13)
            self.assertEqual(recycling.positive_int_input('example question'), 1)
            self.assertEqual(recycling.positive_int_input('example question'), 100000000)
    def test_main(self):

        with patch('builtins.input', side_effect=['4', '1', '3', '2', '3']) as
mock_input:
            recycling.input = mock_input
            recycling.main()  # verify program doesn't crash :) Could also test that it's
printing correct data with a mock print function.



if __name__ == '__main__':
    unittest.main()
```

Resultado del test:



```
Please enter a positive integer.
.Recycling truck program
The total number of crates set out on the street is 9
The max number of crates from any house is 3
The house(s) with the most recycling is [1, 3]
The min number of crates from any house is 1
The house(s) with the least recycling is [0]
....
----------------------------------------------------------------------
Ran 6 tests in 0.003s

OK
```