Ayudantia.md

# Ayudantía Pytest

## Pregunta 1:

### Parte a)

El siguiente método recibe un *string* matemático y debe verificar que los **paréntesis** *( )* dentro de este deben estar **balanceados**. Es decir, que cada paréntesis abierto debe eventualmente cerrarse.

No es necesario verificar que sea matematicamente correcto.

```
def parentesis_balanceados(string):
    # ...
    return False
```

Escriba un test para cada uno de estos casos:

- Se ingresa un *string* válido
  - Por ejemplo: `"g(x) = ((2^x)+5*(42/f(x)))"`
- Se ingresa un *string* válido sin paréntesis.
  - Por ejemplo: `"4+5"`
- Se ingresa un *string* válido vacío: `""`
- Se ingresa un *string* no balanceado (inválido).
  - Por ejemplo: `"(2^(2^(2)) * 4"`
- Se ingresa un *string* balanceado, pero un paréntesis se cierra antes de que se abra (inválido).
  - Por ejemplo: `"(2+4))((6%3)"`

### Parte b)

Escriba los mismos test pero usando `pytest_generate_tests`. Esto sirve para escribir un *test* **más elegante** y más conciso.

Docs: pytest_generate_tests

```
def pytest_generate_tests(metafunc):
    # ...
    pass

def test_parentesis_validos(valid_input):
    assert parentesis_balanceados(valid_input)

def test_parentesis_invalidos(invalid_input):
    assert not parentesis_balanceados(invalid_input)
```

## Pregunta 2:

### Parte a)

Escriba un método que genere direcciones IPv4 como *string*. Estas se componen de cuatro números dentro del rango *[0, 255]* separados por un punto `"."`. Por ejemplo: `192.168.0.1`

```
def random_ip():
    # ...
    return ""
```

Verifique si el método funciona probándolo dentro de un *test*.

```
def test_random_ip():
    # ...
    assert 0
```

## Parte b)

Tenemos el siguiente código pensado para usarlo como base en un gran proyecto ficticio .

La clase `Dispositivo`, está pensada para que sea usada como *superclase* para que de esta hereden clases como, `Celular`, `Impresora`, `CamaraIP`, etc.

La clase `Router` se encarga de transmitir data a la web según el siguiente procedimiento:

- El `Router` recibe un cliente (*subclase* de `Dispositivo`) y lo identifica con su `nombre`. Si este es primera vez que se comunica con el router, entonces este le asigna una dirección IP. En caso de que ya se haya comunicado con el router, entonces ya está registrado y conserva su IP.
  - Quedan registrados en el `defaultdict` llamado `connections`.
- Luego de identificar a un cliente, el router se comunica con la web usando la dirección IP del cliente y la data por enviar.
  - A través el método *privado* `__send_data_to_internet`

```
from collections import defaultdict
import pytest

class Dispositivo:

    def __init__(self, nombre):
        self.nombre = nombre

    def __str__(self):
        return self.nombre


class Router:

    def __init__(self, max_conexiones):
        self.max_conexiones = max_conexiones

        generador = self.__available_ips()

        def nueva_conexion():
            return next(generador)

        self.connections = defaultdict(nueva_conexion)

    def send_data(self, device, data):
        """
        Manda data identificado como el dispositivo.
        Si la data es invalida, no la envia.
        """
        device_ip = self.connections[device]
        return self.__send_data_to_internet(device_ip, data)

    def __send_data_to_internet(self, ip, data):
        if data and len(data) > 0:
            output = "{}: {}".format(ip, data)
            print(output)
            return output
```

```
    def __available_ips(self):
        while len(self.connections) < self.max_conexiones:
            yield(self.random_ip())

    @staticmethod
    def random_ip():
        # El mismo que definimos en la parte a)
        return ""
```

El comportamiendo esperado es:

1. Un dispositivo con nombre puede enviar data satisfactoriamente.
2. Al superar la capacidad máxima del router, todo dispositivo nuevo será rechazado por el router.
3. Si una IP llegase a repetirse, el router debe generar una excepción.
4. Toda data vacía o nula no debe ser enviada.
5. Al intentar conectar un dispositivo sin nombre, debe ser rechazado.

### Parte i

Haga los *tests* necesarios para probar el funcionamiento esperado.

### Parte ii (propuesto)

Haga una subclase de `Router` llamada `GoodRouter` que corrija los *bugs* de la clase original. Para probarla corra los mismos *tests* sobre esta.