Documentación del código de prueba desarrollado

A continuación se presentará un las diferentes pruebas implementadas para las funciones implementadas en la Asignación 1.

> **Aclaración:** Sólo se han desarrollado pruebas para las funciones `create_df` y `check` debido a que la función `tarea` no genera un resultado, esta sólo imprime en pantalla los >resultados para las preguntas de la asignación 1.
>
> Adicionalmente, la función `tarea` sólo hace uso de funciones de la librería Pandas para obtener resultados.
>
> La idea general es asegurar que el Dataframe sea creado de forma correcta a través del uso de la función `create_df` y luego la "limpieza" del DataFrame con la función `check`.
>
> Una posible forma de incluir la función `tarea` para el análisis sería retornar un diccionario con el resultado de cada pregunta y así poder hacer pruebas unitarias. En caso de requerir este paso por favor notificarlo para incluirlo!!!

1. **Importamos módulos y librerías necesarías**

Se decide hacer uso de la librería unittest para el desarrollo de las pruebas unitarias.

Adicionalmente, se debe importar las siguientes librerías:

- `unittest`: librería seleccionada para desarrollar las pruebas
- `pandas`: se debe usar para la creación de los DataFrames de pruebas
- `assert_frame_equal` de la librería `pandas.testing`: para poder realizar la comparación de los DataFrames
- `create_df` y `check` del módulo `funciones`: funciones a probar a través de las pruebas
- `EmptyError`, `ColumnError`, `SepsError` del módulo `errors`: errores especialmente diseñados para la Asignación 1

```
from funciones import create_df, check
import unittest
import pandas as pd
from pandas.testing import assert_frame_equal
from errors import EmptyError,ColumnError,SepsError
```

2. **Definición de las pruebas**

Primeramente creamos la clase `TestAsignacionFunc` que hereda de la clase `unittest.TestCase` Dentro de dicha clase definiremos las pruebas unitarias para cada función.

La primera función declarada es `setUp` la cual es una función reservada por el paquete `unittest`

para la construcción de los elementos o entorno requerido para las pruebas y asi evitar su repetida construcción dentro de cada prueba unitaria.

La misma será ejecutada siempre antes que los test y se ha incluido un bloque `try`/`except` para poder "atajar" un error durante la creación de los DataFrames - debio a la no existencia de los archivos csv - y poder asegurar que no se ejecuten los test en caso de no poder crear los DataFrames; para ello se hace uso de la instrucción/método `skipTest()`

```
class TestAsignacionFunc(unittest.TestCase):
   def setUp(self):
       self.path1 = 'prueba.csv'
       self.path2 = 'prueba2.csv'
       self.columns = ['a','b','c','d']
       self.sep = [';']
       try:
           df1_func = create_df(self.path1,self.columns,self.sep)
           df2_func = create_df(self.path2,self.columns,self.sep)
           df1_func = df1_func.apply(check)
           df1 = pd.DataFrame([[1,2,3,4],
                               [6,5,4,5],
                               [5,6,7,9],
                               [0,0,0,0]],
                               columns= ['a','b','c','d'],
                               dtype='float64')
           df2 = pd.DataFrame([[1,2,3,4],
                               [6,5,4,5],
                               [5,6,7,9]],
                               columns= ['a','b','c','d'])
       except IOError as e:
           print(e)
           self.skipTest(e)
       else:
           self.df1_func = df1_func
           self.df2_func = df2_func
           self.df1 = df1
           self.df2 = df2
```

Variables creadas:

- **df1_func:** DataFrame #1 creado a través de la función a probar (`create_df`) y corregido por la función `check`
- **df2_func:** DataFrame #2 creado a través de la función a probar (`create_df`)
- **df1:** Dataframe #1 creado con los valores "conocidos/esperados" que deben coincidir con la variable **df1_func**
- **df2:** Dataframe #2 creado con los valores "conocidos/esperados" que deben coincidir con la variable **df2_func**

> **¿Por qué he tenido que crear 2 diferentes DataFrames para las pruebas?**
>
> Para poder realizar la comprobación de que dos diferentes DataFrames son iguales se va a hacer uso del método `assert_frame_equal` de la librería `pandas.testing` En orden de que dicho método reconozca dos DataFrames como idénticos este debe contener los mismos elementos en cada posición y deben tener los mismos tipos cada uno de dichos elementos.
>
> Debido a que el DataFrame **df1** y **df1_func** contendrá columnas con el **dtype** Object los elementos tendrán una mezcla de tipos incluyendo cadenas de texto. Esto generá que los >DataFrames no sean reconocidos como iguales aún cuando se hace uso del atributo `check_dtype=False`
>
> Es por ello que se crean dos DataFrames; los número 2 son df con solo elementos númericos tipo **int64** y los número 1 son aquellos cuyo resultado se obtendrá "luego" de la aplicación >de la función `check` con elementos **float64** y donde se han limpiado todos los errores

Pruebas unitarias

- `test_func_create_df`: consiste en validar que la función `create_df` generé un DataFrame de forma correcta

  > Para ello se hace uso de las variables **df2_func** y **df2**

```
    def test_func_create_df(self):
        assert_frame_equal(self.df2_func,self.df2)
```

- `test_func_check`: consiste en validar que la función `check` realiza corectamente la transformación de los DataFrames

  > Para ello se hace uso de las variables **df1_func** y **df1**

```
    def test_func_check(self):
        assert_frame_equal(self.df1_func,self.df1)
```

- `test_df_qty_columns`: consiste en validar de que la cantidad de columnas es la esperada luego de la creación de los DataFrames

  > Se realiza una comprobación con ambos DataFrames debido a que ambos provienen de la misma función y se puede validar si la función `check` puede afectar el resultado de las columnas

```
    def test_df_qty_columns(self):
        self.assertEqual(len(self.df1_func.columns),len(self.df1.
            columns))
        self.assertEqual(len(self.df2_func.columns),len(self.df2.
            columns))
```

- `test_df_columns`: consiste en validar que todas las columnas requeridas estan incluidas en el DataFrame

  > Sólo se realiza la validación con los **df1** y **df1_func**

```
    def test_df_columns(self):
        for col in self.df1_func.columns:
            self.assertIn(col,self.columns)
```

- `test_df_columns_not_empty`: consiste en validar que las columnas del DataFrame no estan vacías

  > Sólo se realiza la validación con los **df1** y **df1_func**

```
    def test_df_columns_not_empty(self):
        for col in self.df1_func.columns:
            self.assertNotEqual(self.df1_func[col].sum(),0)
```

- `test_input_column`: consiste en validar la respuesta de la función `create_df` a argumentos erróneos por parte de los usuarios, en este caso se valida la respuesta a columnas de menos indicadas y se espera que la función genere un `ColumnError`

  > Para esta validación tratamos de crear un nuevo DataFrame

```
    def test_input_column(self):
        with self.assertRaises(ColumnError):
            columns = ['a','b','c']
            create_df(self.path1,columns,self.sep)
```

- `test_input_column2`: consiste en validar la respuesta de la función `create_df` a argumentos erróneos por parte de los usuarios, en este caso se valida la respuesta a columnas de más indicadas y se espera que la función genere un `ColumnError`

  > Para esta validación tratamos de crear un nuevo DataFrame

```
   def test_input_column2(self):
       with self.assertRaises(ColumnError):
           columns = ['a','b','c','d',9]
            create_df(self.path1,columns,self.sep)
```

- `test_input_sep`: consiste en validar la respuesta de la función `create_df` a argumentos erróneos por parte de los usuarios, en este caso se valida la respuesta a la correcta definición del argumento que contiene los separadores a probar, pero que no contiene el indicado para la generación del DataFrame. Se espera que la función genere un `SepsError`

> Para esta validación tratamos de crear un nuevo DataFrame

```
    def test_input_sep(self):
        with self.assertRaises(SepsError):
            sep = [',']
            create_df(self.path1,self.columns,sep)
```

- `test_input_empty_column`: consiste en validar la respuesta de la función `create_df` a argumentos erróneos por parte de los usuarios, en este caso se valida la respuesta ante un DataFrame creado correctamente que contiene alguna de sus columnas completamente vacías. Se espera que la función genere un `EmptyError`

> Para esta validación tratamos de crear un nuevo DataFrame

```
    def test_input_empty_column(self):
        with self.assertRaises(EmptyError):
            path = 'prueba3.csv'
            create_df(path,self.columns,self.sep)
```

3. **Definición de pruebas para validación del tipo correcto de los argumentos**

En esta sección se definen las pruebas requeridas para comprobar que la función responda correctamente a argumentos erróneos por parte del usuario, debido al uso del tipo incorrecto de datos.

He decidido separar estas pruebas ya que deseo hacer uso del decorador `@expectedFailure` incluido en el paquete `unittest`

Escencialmente, las pruebas definidas dentro de esta nueva clase denominada `ExpectedFailureTest` estan diseñadas para el fallo/generación del error y será considerada como correcta durante el test.

De igual forma, se hace uso del método reservado `setUp()` para construir las variables necesarias para el resto de pruebas y evitar definirlas en cada una de ellas.

```
class ExpectedFailureTest(unittest.TestCase):
    def setUp(self):
        self.path = 'prueba.csv'
        self.columns = ['a','b','c','d']
        self.sep = [';']
```

Variables creadas:

- `self.path`: path correcto del archivo csv para la generación del DataFrame
- `self.columns`: definición de columnas correctas para la generación del DataFrame
- `self.sep`: definición del separador correcto para la generación del DataFrame

Pruebas unitarias diseñadas al fallo

- `test_fail_path`: primera comprobación de error tras definir el argumento **path** con un tipo erróneo, en este caso se define como un **`int`**

```
    @unittest.expectedFailure
    def test_fail_path(self):
        path = 2
        create_df(path,self.columns,self.sep)
```

- `test_fail_path2`: segunda comprobación de error tras definir el argumento **path** con un tipo erróneo, en este caso se define como un `list`

```
    @unittest.expectedFailure
    def test_fail_path2(self):
        path = [2]
        create_df(path,self.columns,self.sep)
```

- `test_fail_column`: primera comprobación de error tras definir el argumento **columns** con un tipo erróneo, en este caso se define como un **`int`**

```
    @unittest.expectedFailure
    def test_fail_column(self):
        columns = 2
        create_df(self.path,columns,self.sep)
```

- `test_fail_column2`: segunda comprobación de error tras definir el argumento **columns** con un tipo erróneo, en este caso se define como un `str`

```
    @unittest.expectedFailure
    def test_fail_column2(self):
        columns = '2'
        create_df(self.path,columns,self.sep)
```

- `test_fail_sep`: primera comprobación de error tras definir el argumento **seps** con un tipo erróneo, en este caso se define como un `tuple`

```
    @unittest.expectedFailure
    def test_fail_sep(self):
        sep = (';',)
        create_df(self.path,self.columns,sep)
```

- `test_fail_sep2`: segunda comprobación de error tras definir el argumento **seps** con un tipo erróneo, en este caso se define como un **`float`**

```
    @unittest.expectedFailure
    def test_fail_sep2(self):
        sep = 8.0
        create_df(self.path,self.columns,sep)
```

- `test_fail_sep3`: tercera comprobación de error tras definir el argumento **seps** con su tipo de forma correcta (`list`) pero donde sus elementos no son todos del tipo `str`

```
    @unittest.expectedFailure
    def test_fail_sep3(self):
        sep = [';',3]
        create_df(self.path,self.columns,sep)
```