**Tarea 1. Microcontroladores y Microprocesadores**
**Estudiantes: Sofia Valverde Gutiérrez, Anthony Ureña Jiménez, Isaac Sánchez Gamboa**

**A continuación, se presentan 2 asignaciones para entender y aplicar herramientas de desarrollo de proyectos de programación.**

**Preguntas Teóricas (20 pts, 2pts c/u)**

**1) ¿Diferencie la herramienta Git de Github?**

Git y Github son dos entidades que ayudan con la administración y alojamiento de archivos. Por una parte, Git sirve para controlar las versiones de los archivos mientras que Github funciona como una plataforma para alojar los repositorios de Git.

**2) ¿Qué es un branch?**

Un Branch o rama es una versión del código del proyecto del que se está trabajando. Las branches funcionan para mantener el orden y control de versiones del archivo. Esto con el fin de mantener intactas las versiones originales.

**3) ¿Como se crea un nuevo Branch?**

Un Branch nuevo se genera mediante el comando de "git Branch" seguido del nombre que se le desea asignar. Si se desea empezar a trabajar en una nueva función, se puede crear un branch nuevo a partir de la Branch del main.

**4) ¿Qué es un commit?**

Un commit es una operación que permite guardar cambios realizados en el código acompañado de una descripción de lo que se realizó. Esto otorga la ventaja al usuario de recordar los cambios que fueron aplicados en cualquier versión con fechas anteriores. Por otra parte, si se editan los commits, los cambios no necesariamente se sobrescriben entre sí.

**5) ¿Qué es la operación "git cherry-pick"?**

Git cherry-pick es un potente comando que permite que las confirmaciones arbitrarias de Git se elijan por referencia y se añadan al actual HEAD de trabajo. La ejecución de cherry-pick es el acto de elegir una confirmación de una rama y aplicarla a otra, puede ser útil para deshacer cambios. Por ejemplo, confirmación que se aplica accidentalmente en la rama equivocada, se cambia a la rama correcta y ejecutar cherry-pick en la confirmación para aplicarla a donde debería estar.

**6) Explique que es un "merge conflict" o "rebase conflict" en el contexto de tratar de hacer merge a un Pull Request o de completar una operación git rebase.**

- Merge conflict

Merge conflict se presenta cuando Git es incapaz de resolver automáticamente las diferencias en el código entre dos commits, es decir, si dos desarrolladores realizan cambios al mismo código desde su repositorio remoto y tratan de enviar el file actualizado al repositorio local no podrán hacerlo ambos ya que dará un error, esto se debe a que Git puede fusionar los cambios solo si los commits se encuentran en diferentes líneas o ramas**.**

Para evitar ello los desarrolladores pueden trabajan en ramas separadas y aisladas, de esta forma El comando de fusión de Git combina ramas separadas y resuelve cualquier edición en conflicto.

Un pull request es una petición para integrar propuestas o cambios de código a un proyecto, en el caso que se presente un merge conflict en un pull request se puede resolver de la siguiente forma:

- Asegurarse el código en ambas ramas se actualice con el control remoto.
- Cambiar a la rama que desea fusionar usando el comando git checkout.
- Fusionar localmente así:

```
git pull <the parent branch> origin
```

Lo cual dará el siguiente output

```
Auto-merging origin_<file_name>
CONFLICT (content): Merge conflict in origin_<file_name>
Automatic merge failed; fix conflicts and then commit the result.
```

```
int i = 10;
<<<<<< HEAD
System.out.println(i);
====== master
System.out.println("Hello!");
```

En el ejemplo anterior se puede observar se indica la línea fue rescrita, lo cual permite seleccionar cual de las versiones se desea mantener, de la siente forma

```
int i = 10;
System.out.println(i);
```

Después de un git add, y cuando haga git commit, se mostrará un mensaje que permite actualizará automáticamente el pr que ha creado. El cual esta alisto para ser revisado y fusionado

Git rebase básicamente lo que hace es recopilar uno a uno los cambios confirmados en una rama, y reaplicarlos sobre otra. Utilizar rebase puede ayudar a evitar conflictos siempre que se aplique sobre commits que están en local y no han sido subidos a ningún repositorio remoto.

Después de reordenar y manipular las confirmaciones con git rebase, si se produce un conflicto de combinación, Git lo indica. Se puede proceder de la siguiente manera para resolverlo:

- Ejecutar git rebase --abort para deshacer completamente el rebase.
- Ejecutar git rebase --skip para omitir por completo la confirmación. Eso significa que no se incluirá ninguno de los cambios introducidos por el compromiso problemático.
- Solucionar el conflicto, siguiendo los procedimientos estándar para resolver conflictos de fusión desde la línea de comandos.

**7) ¿Qué es una Prueba Unitaria o Unittest en el contexto de desarrollo de software?**

La prueba unitaria o Unit test corresponde a una forma de comprobar los componentes individuales de los programas informáticos. Estas pruebas permiten examinar el correcto funcionamiento de cada uno de los elementos antes de que ocupen su lugar en el concepto general de un programa. Además, ayudan a comprobar de forma rápida y fácil si el componente funciona según lo previsto. Son una forma eficaz para descubrir errores del código en las fases tempranas de desarrollo del software.

**8) Bajo el contexto de pytest. ¿Qué es un "assert"?**

Los assertions de Pytest son comprobaciones que devuelven el estado Verdadero o Falso. En Python Pytest, si un assertion falla en un método de prueba, entonces la ejecución del método se detiene allí. El código restante en ese método de prueba no se ejecuta y los assertion de Pytest continuarán con el siguiente método de prueba.

Por ejemplo:

```
# content of test_assert1.py
def f():
    return 3


def test_function():
    assert f() == 4
```

Para afirmar que la función devuelva un cierto valor. Si la afirmación (assertion) falla se va a observar un valor de retorno de la función de la siguiente forma:

```
$ pytest test_assert1.py
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-7.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_assert1.py F                                                    [100%]

================================= FAILURES =================================
_____________________________ test_function ______________________________

    def test_function():
>       assert f() == 4
E       assert 3 == 4
E        +  where 3 = f()

test_assert1.py:6: AssertionError
========================= short test summary info ==========================
FAILED test_assert1.py::test_function - assert 3 == 4
============================== 1 failed in 0.12s ===============================
```

pytest tiene soporte para mostrar los valores de las subexpresiones más comunes, incluidas llamadas, atributos, comparaciones y operadores binarios y unarios. Esto le permite usar las construcciones idiomáticas de Python sin código repetitivo sin perder información de introspección.

**9) ¿Qué es Flake 8?**

Es una librería de Python que contiene PyFlakes, pycodestyle y el script McGabe de Ned Batchelder. En si es un gran conjunto de herramientas para verificar un código fuente contra PEP8, errores de programación (como “library imported but unused” y “Undefined name”) y para verificar la complejidad ciclomática.

Definiciones:

PEP 8: La comunidad de usuarios de Python ha adoptado una guía de estilo que facilita la lectura del código y la consistencia entre programas de distintos usuarios. Esta guía no es de seguimiento obligatorio, pero es altamente recomendable. El documento completo se denomina PEP 8

Complejidad ciclomática: Es una métrica de software creada por Thomas J.McCabe para medir el número de rutas independientes a través del código fuente. Así bien, a mayor número de *ifs* dentro de una función, mayor número de caminos tendrá, como consecuencia, mayor complejidad ciclomática. No obstante, existen otras operaciones de flujo de control que impactan el cálculo de la complejidad ciclomática. También se conoce como complejidad McCabe.

**10) Explique la funcionalidad de parametrización de pytest.**

Pytest habilita la parametrización de pruebas en varios niveles:

**Pytest.fixture()** permite parametrizar las funciones de los accesorios. Decora para marcar una función de fábrica de accesorios. Se puede utilizar con o sin parámetros para definir una función de dispositivo.

Se puede hacer referencia al nombre de la función fixture más tarde para provocar su invocación antes de ejecutar las pruebas: los módulos o clases de prueba pueden usar el marcador pytest.mark.usefixtures(fixturename).

Las funciones de prueba pueden usar directamente nombres de dispositivos como argumentos de entrada, en cuyo caso se inyectará la instancia de dispositivo devuelta por la función de dispositivo.

Los dispositivos pueden proporcionar sus valores para probar funciones utilizando declaraciones de retorno o rendimiento. Cuando se usa yield, el bloque de código después de que la instrucción yield se ejecuta como código de desmontaje, independientemente del resultado de la prueba, y debe producir exactamente una vez.

**@pytest.mark.parametrize** permite definir múltiples conjuntos de argumentos y accesorios en la función o clase de prueba.

El decorador integrado pytest.mark.parametrize permite la parametrización de argumentos para una función de prueba. En la siguiente imagen se muestra un ejemplo típico de una función de prueba que implementa la verificación de que una determinada entrada conduce a una salida esperada:

```python
# content of test_expectation.py
import pytest


@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Donde el decorador @parametrize define tres tuplas diferentes (test_input, expected) para que la función test_eval se ejecute tres veces usándolas a su vez:

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-7.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 3 items

test_expectation.py ..F                                          [100%]

================================= FAILURES =================================
_____________________________ test_eval[6*9-42] _____________________________

test_input = '6*9', expected = 42

    @pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
    def test_eval(test_input, expected):
>       assert eval(test_input) == expected
E       AssertionError: assert 54 == 42
E        +  where 54 = eval('6*9')

test_expectation.py:6: AssertionError
======================== short test summary info ========================
FAILED test_expectation.py::test_eval[6*9-42] - AssertionError: assert 54...
====================== 1 failed, 2 passed in 0.12s ======================
```

**pytest_generate_tests:** permite implementar un propio esquema de parametrización o implementar algún dinamismo para determinar los parámetros o el alcance de un dispositivo. Entonces el enlace pytest_generate_tests se llama al recopilar una función de prueba. A través del objeto metafunc pasado, puede inspeccionar el contexto de prueba solicitado y, lo que es más importante, puede llamar a metafunc.parametrize() para provocar la parametrización.

Por ejemplo, si se desea ejecutar una prueba tomando entradas string que queremos configurar a través de una nueva opción de línea de comando pytest. Primero se escribe una prueba simple que acepte un argumento de función de dispositivo de entrada de cadena:

```python
# content of test_strings.py


def test_valid_string(stringinput):
    assert stringinput.isalpha()
```

Luego, se añade un archivo conftest.py que contiene la adición de una opción de línea de comando y la parametrización de prueba:

```python
# content of conftest.py


def pytest_addoption(parser):
    parser.addoption(
        "--stringinput",
        action="append",
        default=[],
        help="list of stringinputs to pass to test functions",
    )


def pytest_generate_tests(metafunc):
    if "stringinput" in metafunc.fixturenames:
        metafunc.parametrize("stringinput", metafunc.config.getoption("stringinput"))
```

Si se añaden dos valores de stringinput, la prueba se ejecutará dos veces:

```
$ pytest -q --stringinput="hello" --stringinput="world" test_strings.py
..                                                                   [100%]
2 passed in 0.12s
```

Si se ejecuta con un valor de stringinput erróneo se obtendrá una prueba fallida:

```
$ pytest -q --stringinput="!" test_strings.py
F                                                                    [100%]
================================= FAILURES =================================
_____________________________ test_valid_string[!] _____________________________

stringinput = '!'

    def test_valid_string(stringinput):
>       assert stringinput.isalpha()
E       AssertionError: assert False
E        +  where False = <built-in method isalpha of str object at 0xdeadbeef0001>()
E        +    where <built-in method isalpha of str object at 0xdeadbeef0001> = '!'.isalpha

test_strings.py:4: AssertionError
========================= short test summary info ==========================
FAILED test_strings.py::test_valid_string[!] - AssertionError: assert False
1 failed in 0.12s
```

En el caso de que no se especifique un stringinput, se omitirá por que se llamará a metafunc.parametrize() con una lista de parámetros vacía:

```
$ pytest -q -rs test_strings.py
s                                                                    [100%]
========================= short test summary info ==========================
SKIPPED [1] test_strings.py: got empty parameter set ['stringinput'], function test_valid_string at /h
1 skipped in 0.12s
```