

# APUNTES UD1

Este documento sirve de índice introductorio para la UD1 del módulo.

*Nota: Para cada uno de los apartados tendrás más información que debes consultar y que es materia de trabajo y de evaluación en el [Manual en Pages de PPS](#)*

## UD1. PRUEBA DE APLICACIONES WEB Y MÓVILES, ANALIZANDO EL CÓDIGO Y SU MODELO DE EJECUCIÓN. HERRAMIENTAS BÁSICAS.

### Índice

- [1. Introducción](#)
- [2. Fundamentos de programación](#)
- [3. Modelos de ejecución de software](#)
- [4. Elementos básicos de un programa](#)
- [5. Pruebas de software](#)
- [6. El control de versiones](#)
- [7. Buenas prácticas en el control de versiones](#)
- [8. El lenguaje de programación Python](#)
- [9. Autoevaluación](#)
- [10. Actividad entregable](#)

### 1. Introduccion

En un mundo que ha sufrido un importante proceso de transformación digital, cada vez hay más aplicativos desarrollados con el fin de interactuar con los seres humanos. Si estos aplicativos no están correctamente programados, un atacante puede hacer uso mal intencionado del software y controlar el flujo de ejecución, pudiendo alterar la información y/o robarla.

Antes de nada, haremos una distinción fundamental entre hardware y software:

- Hardware: todos los componentes físicos de un ordenador
- Software: programas e instrucciones para ejecutar tareas en un ordenador

Una vez tenemos claro la diferencia, en este curso nos centraremos en el software, y en una de sus unidades fundamentales: los programas.

Así, definiríamos programa como:

1. Secuencia de instrucciones
2. Entendible e interpretable por el ordenador
3. Tienen un objetivo o tarea concreta

## 2. Fundamentos de programación

Los ordenadores son máquinas eléctricas que sólo entienden de 0 y 1, siendo:

- 0: no hay corriente
- 1: hay corriente

Este es el único lenguaje que entiende el ordenador, llamado lenguaje binario.

Para facilitar la labor de crear programas, que de aquí en adelante lo llamaremos programar, aparecieron los lenguajes de programación. Los podemos dividir (haciendo una gran abstracción) en dos grupos:

- Lenguajes de bajo nivel: son los mas cercanos al binario, pero son más difíciles de interpretar por el programador.
- Lenguajes de alto nivel: necesitan ser traducidos antes de llegar al ordenador, pero son más fáciles de programar, ya que son mas cercanos al lenguaje natural de las personas.

Algunos ejemplos:

1. Lenguajes bajo nivel: lenguaje máquina
2. Lenguajes alto nivel: C++, Python o Go

Así, en el desarrollo de software hay muchas condicionantes que marcan que elección del lenguaje que se usará en el proyecto. Siendo estos:

- Los requisitos técnicos marcados por otras piezas de software/hardware
- Tiempo de entrega del software
- Necesidad de rendimiento

## 3. Modelos de ejecución de software

Podemos clasificar los lenguajes de programación según su modo de ejecución en 3 tipos distintos:

### 1. **Compilado**

- Nuestro código fuente se transforma en un binario mediante compilación.
- El ordenador ejecuta el binario.
- No necesitamos por tanto ningún programa adicional.

### 2. **Interpretado**

- Nuestro código fuente es leído en tiempo real por un programa (intérprete) que lo traduce a código máquina(objeto binario)
- El ordenador ejecuta ese binario
- Necesita por tanto un programa adicional, llamado intérprete, que hace de traductor entre las instrucciones y el código máquina

### 3. **Intermedio**

- El código fuente se compila a un lenguaje intermedio
- Este lenguaje intermedio se ejecuta en una máquina virtual



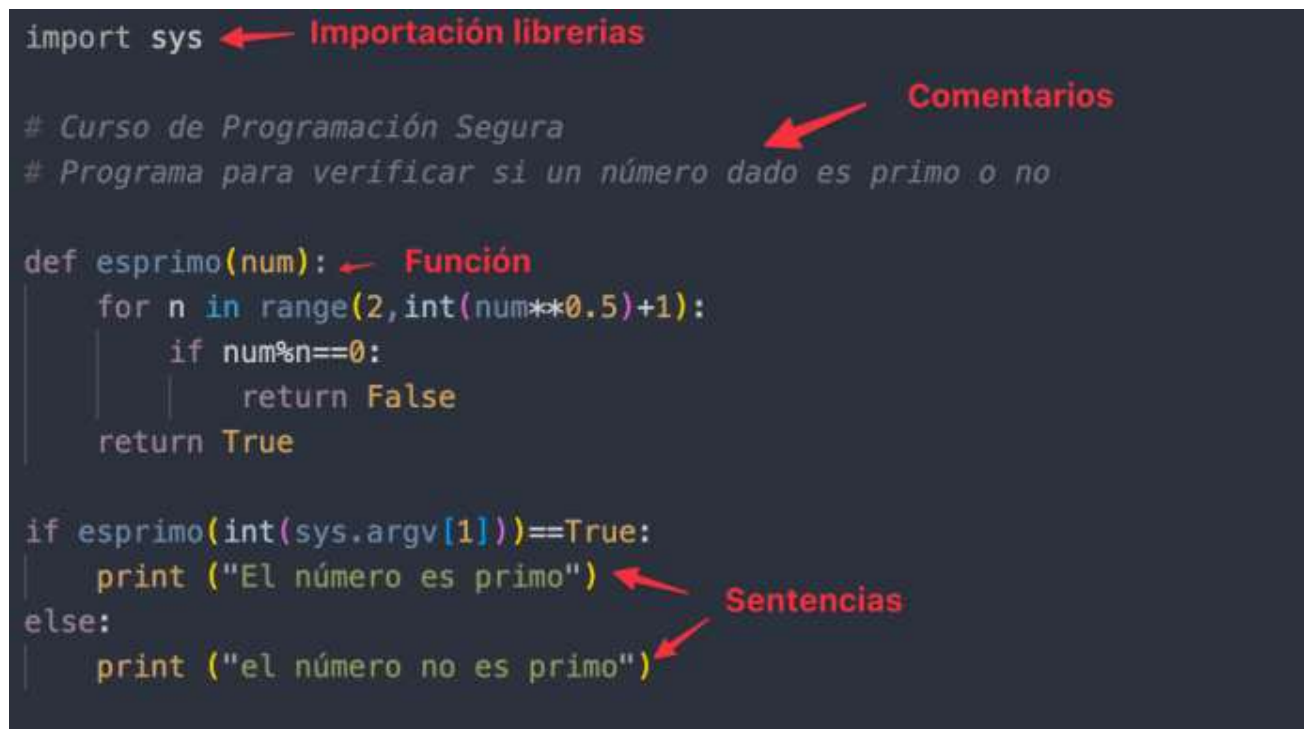
Un compilador no es más que un software que se encarga de transformar código fuente en lenguaje binario, que es más fácil de entender por los ordenadores. Cuando compilamos un programa, elegimos el tipo de plataforma y arquitectura en la que queremos que el binario funcione según el compilador que estemos usando. Hay lenguajes como Golang que permiten hacer un compilado multiplataforma especificando sobre qué plataforma y arquitectura queremos que nuestro binario funcione cuando lo estamos compilando.

## 4. Elementos básicos de un programa

Dentro de nuestro programa o código fuente podemos distinguir varios componentes:

- Comandos de pre-procesamiento: importación de módulos/librerías.
- Funciones.
- Declaración de variables.
- Sentencias y expresiones.
- Comentarios.

En la imagen dada a continuación se puede ver un pequeño programa desarrollado en Python en el que se muestra la estructura básica que sigue el código fuente de un programa:



```
import sys ← Importación librerías

# Curso de Programación Segura
# Programa para verificar si un número dado es primo o no ← Comentarios

def esprimo(num): ← Función
    for n in range(2, int(num**0.5)+1):
        if num%n==0:
            return False
    return True

if esprimo(int(sys.argv[1]))==True:
    print("El número es primo") ← Sentencias
else:
    print("el número no es primo")
```

## 5. Pruebas de software

Las pruebas de software son un elemento crítico para garantizar la calidad de un producto de software. La necesidad de las compañías de desarrollar software de calidad, ha motivado el diseño de diversas metodologías para el desarrollo y ejecución de pruebas.

Diversas compañías buscan implementar satisfactoriamente estas metodologías, pero definir los pasos para mejorar y controlar las fases del proceso de pruebas de software y el orden en que estas se implementan es, en general, una tarea difícil.

De forma general, las **pruebas de software** las podemos clasificar en base a cómo las ejecutamos:

1. Pruebas manuales
2. Pruebas automáticas

Pero también podemos clasificarlas, en base a lo que estamos midiendo y probando, teniendo muchos tipos distintos. Las más comunes son:

- Pruebas de aceptación: Verifican si todo el conjunto de software se comporta según lo esperado.
- Pruebas de integración: Verifican que distintos módulos que funcionan por separado, funcionen de manera conjunta.
- Examen de la unidad: Valida que cada unidad de software funcione como se esperaba. Una unidad es el componente comprobable más pequeño de una aplicación.
- Pruebas funcionales: Verifica las funciones emulando escenarios comerciales, basados en requisitos funcionales. Las pruebas de caja negra son una forma común de verificar funciones.
- Pruebas de rendimiento: Prueba el rendimiento del software en diferentes cargas de trabajo. Las pruebas de carga, por ejemplo, se utilizan para evaluar el rendimiento en condiciones de carga reales.
- Pruebas de regresión: Verifica si las el código introducido rompe o degrada la funcionalidad del software original. Suelen realizarse cuando hacemos cambios en un programa.
- Pruebas de estrés: Prueba cuánta tensión puede soportar el sistema antes de que falle. Está considerada como un tipo de prueba no funcional.
- Pruebas de usabilidad: Valida qué tan bien un cliente puede usar un sistema o una aplicación web para completar una tarea.

Cuando estamos desarrollando y probando software necesitamos disponer de un entorno de desarrollo aislado donde poder hacer nuestras pruebas y verificar si el software concreto que hemos desarrollado se comporta de la manera esperada ante las distintas variables o datos procesados. La mejor manera de hacerlo es tener un entorno controlado, no influenciado por otras piezas de software o hardware y donde todos los programadores pueden tener las mismas condiciones y por tanto poder medir y probar el software de manera eficaz y eficiente.

Estos entornos se conocen como *sandbox*. Es decir, un entorno cerrado en el que probar los cambios de código de forma aislada, sin comprometer la producción ni la edición del programa en desarrollo. Así, el sandbox protege, controla de forma preventiva y en tiempo real la ejecución del código, evitando cambios que podrían comprometer la estabilidad del programa.

Si bien el término sandbox se refiere a un entorno aislado y controlado, dentro de la informática tiene dos usos. A nivel de desarrollo de software, nuestro caso, se podría definir como el entorno donde creamos y probamos nuestro software de manera controlada, mientras que a nivel de ciberseguridad se entiende por sandbox el entorno donde se ejecuta de manera aislada el software malicioso o malware para ver como se comporta sin suponer un riesgo.

## 6. Control de versiones con Git

Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Usar un VCS (Sistema de Control de Versiones) también significa generalmente que si arruinas o pierdes archivos, será posible recuperarlos fácilmente.

### Sistemas de control de versiones locales

Un método de control de versiones, usado por muchas personas, es copiar los archivos a otro directorio. Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías. Para afrontar este problema los programadores desarrollaron hace tiempo VCS locales que contenían una simple base de datos, en la que se llevaba el registro de todos los cambios realizados a los archivos.

## **Sistemas de control de versiones centralizados**

El siguiente gran problema con el que se encuentran las personas es que necesitan colaborar con desarrolladores en otros sistemas. Los sistemas de Control de Versiones Centralizados (CVCS por sus siglas en inglés) fueron desarrollados para solucionar este problema. Esta configuración ofrece muchas ventajas, especialmente frente a VCS locales. Por ejemplo, todas las personas saben hasta cierto punto en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado sobre qué puede hacer cada usuario, y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han realizado copias de seguridad adecuadamente, se perderá toda la información del proyecto, con excepción de las copias instantáneas que las personas tengan en sus máquinas locales.

## **Sistemas de control de versiones distribuidos**

Los sistemas de Control de Versiones Distribuidos (DVCS por sus siglas en inglés) ofrecen soluciones para los problemas que han sido mencionados. En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no solo descargan la última copia instantánea de los archivos, sino que se replica completamente el repositorio. De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada clon es realmente una copia completa de todos los datos.

Además, muchos de estos sistemas se encargan de manejar numerosos repositorios remotos con los cuales pueden trabajar, de tal forma que puedes colaborar simultáneamente con diferentes grupos de personas de distintas maneras dentro del mismo proyecto. Esto permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.

## **Características de Git**

Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

- Copias instantáneas, no diferenciadas

Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

- Casi todas las operaciones son locales.

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general no se necesita información de ningún otro computador de tu red. Por ejemplo, para navegar por la historia del proyecto, Git no necesita conectarse al servidor para obtener la historia y mostrártela - simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi instantáneamente. Esto también significa que hay muy poco que no puedes hacer si estás desconectado. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos.

- Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa.

No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo.

- Git sólo añade información

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda enmendar, o que de algún modo borre información.

## 7. Buenas prácticas en el control de versiones

### 7.1. Versionado semántico

En el mundo de la administración de software existe un temido lugar llamado: *Infierno de Dependencias*. Mientras más crece tu sistema y más paquetes integras dentro de tu software, más probable se hace que un día te encuentres en este pozo de desesperación.

En sistemas con muchas dependencias, lanzar nuevas versiones de los paquetes puede convertirse en una pesadilla. Si las especificaciones de la dependencias son muy estrictas, estarás en peligro de bloquear una versión (la inhabilidad de actualizar un paquete sin tener que publicar una nueva versión de cada otro paquete dependiente). Si las dependencias son especificadas de forma muy relajada, inevitablemente serás mordido por versiones promiscuas (asumir la compatibilidad con próximas versiones más allá de lo razonable). El *Infierno de Dependencias* es donde estás cuando una versión bloqueada y/o promiscua previenen que muevas tu proyecto adelante de forma fácil y segura.

Como solución a este problema, se proponen un conjunto simple de reglas y requerimientos que dicten cómo asignar e incrementar los números de la versión. Estas reglas están basadas en prácticas preexistentes de uso generalizado tanto en software de código cerrado como de código abierto, pero no necesariamente limitadas a estas. Para que este sistema funcione, primero debes declarar un API pública. Éste puede consistir en documentación aparte o ser impuesto por el código mismo. Independientemente de lo anterior, es importante que este API sea claro y preciso. Una vez identifiques tu API público, debes comunicar los cambios realizados a éste con incrementos específicos a tu número de versión. Considera un formato de versión X.Y.Z (Mayor.Menor.Parche). Las correcciones de errores que no afectan el API incrementan la versión parche. Adiciones o sustracciones compatibles con versiones anteriores incrementan la versión menor, y cambios en el API incompatibles con versiones anteriores incrementan la versión mayor.

Se llama a este sistema “Versionado Semántico”. Bajo este esquema, los números de versión y la forma en la que cambian transmiten el sentido del código y lo que ha sido modificado de una versión a otra.

Todo esto, lo podríamos resumir como:

Dado un número de versión MAYOR.MENOR.PARCHE, se incrementa:

- La versión MAYOR cuando realizas un cambio incompatible en el API,
- La versión MENOR cuando añades funcionalidad compatible con versiones anteriores, y
- La versión PARCHE cuando reparas errores compatibles con versiones anteriores.

Hay disponibles etiquetas para prelanzamiento y metadata de compilación como extensiones al formato MAYOR.MENOR.PARCHE.

Puedes encontrar más información en la [documentación](#)

### 7.2. GitFlow

Gitflow es un modelo alternativo de creación de ramas en Git en el que se utilizan ramas de función y varias ramas principales. Fue Vincent Driessen en nvie quien lo publicó por primera vez y quien lo popularizó. En comparación con el desarrollo basado en troncos, Gitflow tiene diversas ramas de más duración y mayores confirmaciones.

Según este modelo, los desarrolladores crean una rama de función y retrasan su fusión con la rama principal del tronco hasta que la función está completa. Estas ramas de función de larga duración requieren más colaboración para la fusión y tienen mayor riesgo de desviarse de la rama troncal. También pueden introducir actualizaciones conflictivas.

Gitflow puede utilizarse en proyectos que tienen un ciclo de publicación programado, así como para la práctica recomendada de DevOps de entrega continua. Este flujo de trabajo no añade ningún concepto o comando nuevo, aparte de los que se necesitan para el flujo de trabajo de ramas de función. Lo que hace es asignar funciones muy específicas a las distintas ramas y definir cómo y cuándo deben estas interactuar. Además de las ramas de función, utiliza ramas individuales para preparar, mantener y registrar publicaciones. Por supuesto, también puedes aprovechar todas las ventajas que aporta el flujo de trabajo de ramas de función: solicitudes de incorporación de cambios, experimentos aislados y una colaboración más eficaz.

Para profundizar, en el siguiente [artículo](#) entendemos cómo aplicar GitFlow.

## 8. El lenguaje Python

El lenguaje que usaremos para la actividad de la unidad será Python. Entre otras cosas, lo hemos elegido por su facilidad a la hora de llevar al ámbito de producción y por las múltiples posibilidades que nos brinda en este y en otros contextos.

Python es un lenguaje de programación de alto nivel ampliamente utilizado en todo el mundo debido a su simplicidad, legibilidad y versatilidad. Aunque cada día se caracteriza por nuevos rasgos, si que deberíamos destacar:

- Lenguaje interpretable: Python es un lenguaje interpretado, lo que significa que puedes escribir y ejecutar código directamente, sin necesidad de compilarlo previamente. Esto hace que sea rápido y fácil de usar para la programación.
- Tiene una sintaxis limpia y legible: Utiliza una sintaxis clara y legible. Utiliza espacios en blanco y sangría para estructurar el código, lo que fomenta una escritura limpia y fácil de entender. Por ejemplo:

```
for i in range(5):  
    print("Hola, mundo!")
```

- Amplia biblioteca estándar: Python incluye una biblioteca estándar rica en módulos y funciones que abarcan una amplia variedad de tareas, desde procesamiento de texto hasta manipulación de archivos, lo que simplifica el desarrollo de aplicaciones.
- Multiplataforma: Python es multiplataforma, lo que significa que puedes escribir código en un sistema operativo y ejecutarlo en otro sin cambios significativos.
- Orientación a Objetos: Python es un lenguaje orientado a objetos, lo que significa que puedes utilizar la programación orientada a objetos para crear estructuras de datos y abstracciones de manera sencilla.
- Comunidad activa: Python tiene una comunidad de desarrolladores muy activa que contribuye con paquetes y bibliotecas adicionales. Esto significa que tienes acceso a una amplia gama de herramientas para casi cualquier tarea de programación.
- Utilizado en diversas aplicaciones: Python se utiliza en una variedad de aplicaciones, como desarrollo web (Django, Flask), análisis de datos (pandas, NumPy), inteligencia artificial y aprendizaje automático (TensorFlow, PyTorch), automatización de tareas, scripting y más.

## 9. Autoevaluación

Identifica si las siguientes frases son verdaderas o falsas

1. Python es un lenguaje compilado de alto rendimiento

- ☐ Verdadero
- ☐ Falso

2. Ruby es un lenguaje interpretado, es decir necesita un intérprete y por tanto NO es compilado.

- ☐ Verdadero
- ☐ Falso

3. Golang es un lenguaje compilado y por tanto si rendimiento será mayor

- ☐ Verdadero
- ☐ Falso

4. Scala es un lenguaje intermedio puesto que el código se compila a un lenguaje intermedio para mas tarde ser ejecutado en un entorno controlado o maquina virtual.

- ☐ Verdadero
- ☐ Falso

5. Python es un lenguaje muy seguro porque tiene librerías para depurar y hacer correcciones de código.

- ☐ Verdadero
- ☐ Falso

6. No tiene sentido analizar los posibles datos de entrada de una función mientras estamos programando porque ya haremos la revisión más adelante de manera automática.

- ☐ Verdadero
- ☐ Falso

### *Soluciones*

1. Falso. Python es un lenguaje que necesita un intérprete, que hace un trabajo de compilado en tiempo real, es decir es un lenguaje NO compilado y por tanto su rendimiento será menor.

2. Verdadero. Los lenguajes interpretados tienen, valga la redundancia, un intérprete que los compila y ejecuta en tiempo real. Es decir no ha sufrido un proceso de compilación previo.

3. Verdadero. Golang o go es un lenguaje que esta adquiriendo gran popularidad porque es sencillo de programar como Python pero tiene el rendimiento de un lenguaje compilado.

4. Verdadero. Los lenguajes intermedios sufren de un proceso de compilación y ejecución en un entorno virtual o controlado.

5. Falso. La mayoría de lenguajes modernos incorporan módulos o librerías que permiten revisar y detectar errores de manera automática o semi automática. No podemos marcar lenguajes como seguros o inseguros de manera categórica, sino su diseño, implantación y validación dentro de un proyecto.

6. Falso. Cuando definimos una función deberemos de entender que datos estamos permitiendo de entrada y realmente revisar si lo podemos limitar lo máximo posible para prevenir situaciones no esperadas.

## 10. Actividades entregables

### 10.1. Resultados de aprendizaje

RA1. Prueba aplicaciones web y aplicaciones para dispositivos móviles analizando la estructura del código y su modelo de ejecución.

### 10.2. Enunciado



## Contexto práctico

Julián cuando necesita comprobar parte de un código en Python usa la librería incluida por defecto en las distribuciones de Python llamada unittest.

Tiene un pequeño programa llamado fibo.py que escribe la serie de Fibonacci hasta un número dado y devuelve el valor de esa misma posición.

Decide hacer un test del software de tipo aceptación. Es decir, quiere verificar que una parte del código se comporta de forma esperada. Para ello crea un script que importará la librería unittest y comprobará si el quinto número de la serie Fibonacci, el número 3, es correcto (la serie Fibonacci es 0,1,1,2,3,5,8,13...).

Para ello importa la librería de test de software unittest y define una clase donde incluiría la función para comprobar esta parte específica del código. Va a comprobar que el quinto número de la serie sea igual a 3, lo hace mediante la sentencia `self.assertEqual(result, 3)`.

## ¿Qué se pide?

Antes de nada reflejaremos la sucesión de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... Tienes mas información sobre esta interesante sucesión y su origen [aquí](#)

1: Crea un script que genere la secuencia de Fibonacci

Puedes usar cualquier lenguaje con el que estés familiarizado, pero te recomiendo por facilidad y por el gran acceso a librerías de evaluación de software el lenguaje Python. Si aún no has programado en Python, su inmersión te será sencilla y amigable.

Llamaremos a este script fibo.py (en el caso en que estemos programando en python)

Deberías de crear un programa con su estructura completa. Dentro de este programa crearemos una función llamada fibonacci.

2: Creación del programa principal

2.1. Crearemos un programa principal donde definiremos una clase llamada Test, donde probaremos nuestro software.

2.2. Importaremos la librería de testeo de software, en este caso unittest.

2.3. Crearemos nuestra clase (del tipo unittest.TestCase)

2.4. Dentro de esta clase definiremos una función (podemos llamarla como consideremos, pero es recomendable un nombre ilustrativo)

Dentro de la función reflejaremos el tipo de testeo que vamos a realizar. En este caso nuestro objetivo es verificar si la posición X de la sucesión de Fibonacci coincide con el resultado esperado, es decir, si coincide con la posición X que devuelve nuestro programa. Para ello, comprobamos que el quinto número de la sucesión, que es 3, coincide con el quinto número que devuelve nuestra secuencia. Para hacer esta comprobación usaremos el tipo de comprobación `assertequal` que comprueba si dos valores son iguales

Mediante esta función comprobaremos si la posición quinta de nuestra función coincide con el valor esperado (el valor esperado es la quinta posición de la sucesión que es 3)

3: Verificación de software y pregunta final

Ejecutaremos el programa final y verificaremos si realmente nuestro programa que calcula la sucesión de Fibonacci (fibo.py) se comporta como esperamos. Si el programa que comprueba el código detecta un error, nos reflejará que dato está esperando y que ha recibido. ¿Qué tipo de prueba hemos realizado?

## Entrega

Debéis realizar la entrega a través de un repositorio creado para tal fin, en una cuenta personal de Github. Podéis emplear tanto una cuenta que ya tengáis y a la que estáis acostumbrados a acceder, como una que creéis a través

de la cuenta que os proporciona el centro. La entrega y, en este caso el repositorio, debe contener:

- Una carpeta con todo el código. Debidamente comentado, sangrado y tabulado.
- Un documento en formato PDF que explique las partes que creáis convenientes y que se os demandan explícitamente en los distintos apartados. Debe contener portada, un índice con los distintos puntos, el contenido correspondiente a la resolución de la práctica y un apartado de bibliografía, si has empleado alguna.

Así, para todos los apartados que expliquéis, es una buena práctica y necesario entregar las capturas de pantalla de los principales pasos realizados, explicando el proceso seguido en cada uno de ellos, así como los resultados de cada una de las ejecuciones o los códigos que se han ido desarrollando y que queréis explicar. El objetivo es que pueda observar que has resuelto la tarea sin estar en tu propio ordenador y entender la explicación proporcionada. Esto os ayudará en el examen de evaluación a repasar lo que habíais realizado unas semanas atrás.

En las capturas de pantalla realizadas se debe poder ver claramente que la ejecución la habéis hecho desde vuestro usuario o vuestro equipo. Una forma es tener como fondo de pantalla la plataforma con tu usuario mostrando claramente la foto de tu perfil. En caso de encontraros en el terminal de ejecución o similares, es importante que se observe que la ejecución la habéis hecho desde un usuario que os corresponda. Aquellos apartados/subapartados que no cumplan esta condición no serán corregidos.

Para la entrega, debéis realizar un último *commit* de entrega de actividad con el siguiente mensaje : "Entrega\_UD1\_Nombre\_Apellidos". Es importante que el *commit* esté dentro del plazo marcado; cualquier *commit* posterior (y no autorizado), hará que la práctica no sea corregida.

Como recomendación, sería una buena práctica que uséis el repositorio para ir mostrando los avances que realizáis. Tampoco os pido que realicéis *commits* cada vez que modifiquéis algo menor en el programa, pero si si empleáis varios días para la resolución de la práctica, un *commit* por día o por cada cambio significativo (que quede debidamente documentado y con el mensaje correspondiente), se tendrá en cuenta de forma positiva. Es decir, que pueda ver el histórico del proceso de desarrollo es uno de los objetivos de la UD, además de que os ayudará para las siguientes UD's a tener terreno ganado con el manejo de un sistema de control de versiones.

### 10.3. Evaluación

#### Rúbrica de la tarea

Apartado	Descripción	Puntuación
1	Realiza un script que contenga la estructura necesaria para poder desarrollar la sucesión de Fibonacci	1 punto
1	Define una función para poder calcularla posición concreta de una posición de la sucesión de Fibonacci	1,5 puntos
1	El código es claro y descriptivo. En caso de reflejar alguna particularidad usará comentarios	1,5 puntos
2	Importa las librerías correctas, tanto para hacer las pruebas unitarias como para importar la función creada anteriormente	1 punto
2	Crea e instancia una clase de manera correcta, o en su defecto realiza las pruebas mediante funciones o invocaciones de una manera clara y controlada	1 punto
2	El código es claro y conciso y tiene una estructura ordenada que permita realizar no solamente la prueba que buscamos sino realizar pruebas adicionales	1 punto
3	Invoca la función exacta que buscamos (assertEqual) de una manera clara con los valores que necesitamos comprobar	1 punto
3	La estructura del código permite realizar mas pruebas de forma sencilla mediante la parametrización de las variables	1 punto

Apartado	Descripción	Puntuación
3	La prueba de software es exitosa	1,5 puntos

**NOTA IMPORTANTE**

Si se detecta copia en alguna actividad, la puntuación será de un 0.

- Aquellos apartados/subapartados en los que las capturas de pantalla no sean claras o no se pueda probar la autoría, no serán corregidos.
- Si se hace entrega de una actividad con un formato significativamente incorrecto, penalizará en la nota máxima posible de la tarea al incumplirse las directrices indicadas.
- Si se realiza un commit posterior a la fecha de entrega de la actividad, esta no será corregida y tendrá una calificación de 0.

## Recursos

Enlace a la [documentación de unittest](#)

## Bibliografía

*Estos materiales están basados en los contenidos formativos de FP Online, propiedad del Ministerio de Educación y Formación Profesional, así como en diverso material y manuales disponibles de forma abierta en la red*

- Fernández Riera, M. (2022). Puesta en producción segura. Editorial Ra-Ma.

## Para saber más

- [Historia de los sistemas de control de versiones](#)
- [Buenas prácticas en los commit](#)

📅 Última actualización: 01.12.2023

👤 Autor/a: José Fernández Gómez

