Clase 1

1. ¿Es python un lenguaje de scripting? Usos típicos

Python es un lenguaje de propósito general que es frecuentemente usado para scripting. Es comúnmente definido como un lenguaje de scripting orientado a objetos, esta definición combina el soporte para programación orientada a objetos con los scripts. Usos:

- **Programación de sistemas:** con ayuda de las interfaces built-in con el SO.
- **GUIs:** Python viene con una interfaz orientada a objetos para implementar GUIs (Tkinter, wxPython).
- **Internet scripting:** soporte para FTP, XML, parse HTML, RPC, SOAP, frameworks (django).
- **Programación de base de datos:** Existen interfaces de Python para todos los sistemas de bases de datos relacionales: Oracle, MySQL, PostgreSQL, SQLite y más.
- **Programacion numerica y científica:** numpy, scipy
- Gaming, robots, puertos serie, etc.
- Inteligencia artificial
- Big data

2. ¿Es interpretado o compilado?

Python es un lenguaje de programación interpretado. Esto significa que el código fuente de Python se ejecuta directamente por el intérprete de Python, sin necesidad de un proceso de compilación previo. Cuando escribes un programa en Python, el intérprete lee y ejecuta el código directamente, línea por línea. Python utiliza un enfoque interpretado, lo que simplifica el desarrollo y la portabilidad del código. Python no requiere un paso de compilación explícito antes de la ejecución del código.

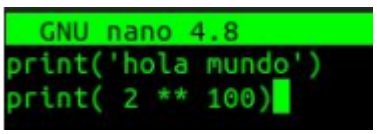
3. Una característica de python es que es portable, que significa y cómo lo logra?

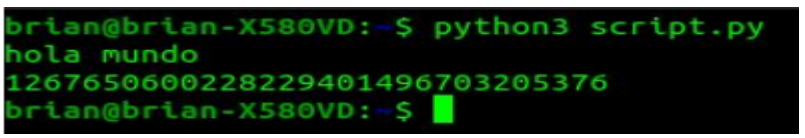
Python compila las sentencias de código fuente a un formato intermedio conocido como bytecode y después interpreta el byte code. El bytecode permite portabilidad, ya que es un formato independiente de la plataforma. La implementación estándar de python está escrita en ANSI C portable y compila y se ejecuta en prácticamente todas las plataformas principales actualmente en uso.

4. Ejecución de programas, lado del programador y lado de python.

Vista del programador:

- En su forma más simple, un programa en python es solo un archivo de texto que contiene declaraciones de python. Después de guardarlo podemos indicar a python que lo ejecute. Python interpreta cada declaración desde arriba hacia abajo, una a la vez.





Vista del python:

- Cuando le indica a python que ejecute el programa, python lleva a cabo algunos pasos antes de que realmente el código comience a funcionar. Específicamente, primero compila el código a algo llamado “bytecode” y después lo enruta a algo llamado “virtual machine”. Python traduce cada una de sus declaraciones en un grupo de instrucciones de bytecode descompuestas en pasos individuales. Esta traducción de código de bytes se realiza para acelerar la ejecución: el bytecode se puede ejecutar mucho más rápido que el código fuente original. Python almacena el byte code de sus programas en archivos que terminan con una extensión .pyc. Python guarda estos archivos para optimizar la velocidad de inicio. La próxima vez que se ejecute el programa, python va a cargar los archivos .pyc y saltea el proceso de compilación, siempre y cuando no se haya cambiado el código fuente. Esta validación la realiza automáticamente python y el mismo sabe cuando debe recompilar. Una vez que el código fue compilado en bytecode, este es enviado para su ejecución a lo que es conocido como la máquina virtual de python. El PVM es el motor en tiempo de ejecución de python y siempre está presente como parte del sistema python, es el componente que realmente ejecuta los scripts. Técnicamente es el último paso de lo que es llamado el “intérprete de python”.

5. Dinámico pero fuertemente tipado, ¿Qué significa? Explique cómo resuelve python la siguiente asignación a=3.

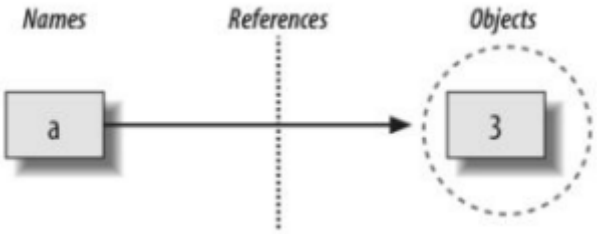
En Python los tipos son determinados automáticamente en tiempo de ejecución. Esto significa que no tenemos que declarar variables antes de tiempo. En un lenguaje de programación fuertemente tipado, no se permite realizar ciertas operaciones entre tipos de datos incompatibles. El lenguaje impone restricciones en las operaciones entre tipos diferentes para evitar comportamientos inesperados o errores.

- a = 3

Para atender esta instrucción python realiza 3 pasos:

1. Crea un objeto para representar el valor 3.
2. Crea una variable a, si esta no existe aún
3. Relaciona la variable a con el nuevo objeto 3.

El resultado será una estructura dentro de python como se ve en la figura. Como es de imaginarse los objetos y las variables se almacenan en diferentes partes de la memoria y se asocian mediante enlaces. Las variables siempre se vinculan a objetos y nunca a otras variables, pero los objetos más grandes pueden vincularse a otros objetos. Estos links desde las variables a los objetos son llamadas referencias en python, una referencia es un tipo de asociación, implementadas por punteros en memoria. Siempre que las variables sean usadas más tarde, python mantiene la referencia.



Clase 2

1. ¿Qué es Git? ¿Qué es una estrategia git y cuáles conoce?

Git es un sistema de control de versiones distribuido, diseñado para manejar proyectos de cualquier tamaño con rapidez y eficiencia. Git permite a los desarrolladores llevar un registro de los cambios en el código fuente, colaborar de manera efectiva y mantener un historial detallado de las modificaciones en un proyecto. El código que se guarda en git, cambia constantemente. A su vez, varios desarrolladores pueden estar trabajando sobre el mismo código. Por lo cual, un sistema de control de versión debe mantener un historial de todos los cambios realizados. Tarea que git lleva a cabo distribuyendo el código en dos repositorios uno local y otro remoto.

Una estrategia git es la manera en la que se administran los branches y releases, en otras palabras cómo se integra y se hace que el código de desarrollo llegue al branch de producción.

El flujo de trabajo de GitFlow proporciona una estructura organizada para el desarrollo de software, especialmente en proyectos más grandes con múltiples colaboradores y lanzamientos frecuentes. GitFlow se basa en el uso de ramas para representar diferentes estados del proyecto. Las ramas principales en GitFlow son:

Master:

- La rama maestra representa la versión más reciente y estable del proyecto.

Develop:

- La rama develop representa la próxima versión del proyecto.

Las ramas de características en GitFlow son:

Feature:

- Las ramas de características se utilizan para desarrollar nuevas características o funcionalidades para el proyecto.

Release:

- Las ramas de lanzamiento se utilizan para preparar una nueva versión del proyecto para su lanzamiento.

Hotfix:

- Las ramas de hotfix se utilizan para corregir errores críticos en una versión publicada del proyecto.

El flujo de trabajo de GitFlow se divide en las siguientes etapas:

Inicio:

- En esta etapa, se crea una nueva rama de características para cada nueva característica o funcionalidad que se va a desarrollar.

Desarrollo:

- En esta etapa, los desarrolladores trabajan en las características o funcionalidades en sus ramas de características.

Integración:

- En esta etapa, las ramas de características se integran con la rama develop.

Lanzamiento:

- En esta etapa, se crea una nueva rama de lanzamiento para preparar una nueva versión del proyecto.

Hotfix:

- En esta etapa, se crea una nueva rama de hotfix para corregir errores críticos en una versión publicada del proyecto.

La metodología Trunk-Based Development (Desarrollo Basado en Tronco) es un enfoque en el desarrollo de software que se centra en mantener una rama principal, a menudo llamada tronco (o trunk en inglés), que siempre está en un estado que puede ser implementado. A diferencia de estrategias que utilizan ramas de características a largo plazo, Trunk-Based Development aboga por ramas de corta duración y entrega continua. En este enfoque, todos los desarrolladores contribuyen directamente a master, evitando la creación de ramas de características a largo plazo. Master es la rama principal y única del repositorio. Las ramas de características o tareas son de corta duración. Los desarrolladores crean ramas específicas para implementar una característica o solucionar un problema, y después de la implementación y pruebas, la rama se fusiona rápidamente de vuelta a master.

2. ¿Que es una función? Describa las partes que la componen en sintaxis python 3.x

En programación, una función es un bloque de código reutilizable que realiza una tarea específica. En Python 3.x, una función tiene una sintaxis específica que incluye varias partes. Aquí está la estructura básica de una función en Python:

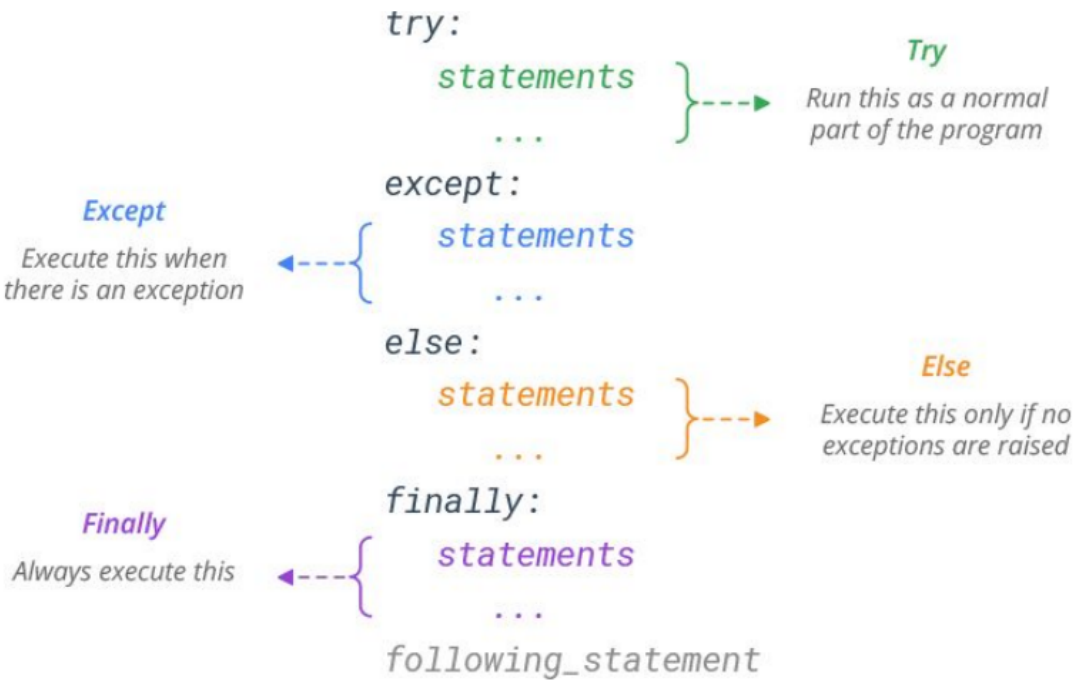
```
def nombre_de_la_funcion(parametro1, parametro2, ...):
    """Docstring de la función"""
    # Cuerpo de la función
    # Puede contener declaraciones, expresiones, y más

    # Opcional: Sentencia return para devolver un valor
    return resultado
```

3. ¿Qué es una Excepción? ¿Cuándo usarlas?

Una excepción es un evento que ocurre durante la ejecución de un programa que interrumpe el flujo normal de las instrucciones del programa. En general, cuando un script de Python se encuentra con una situación que no puede manejar, genera una excepción. Una excepción es un objeto de Python que representa un error. Cuando una secuencia de comandos de Python genera una excepción, debe manejar la excepción de inmediato; de lo contrario, finaliza y se cierra. Si se tiene algún código sospechoso de generar una

excepción, se puede “defender” al programa colocando el código sospechoso en un bloque **try:**. Después del bloque try, debe ir una instrucción **except**, seguida de un bloque de código que maneje el problema de la manera más elegante posible.

4. Estructura de una excepción



Clase 3

1. Explicar programación imperativa y declarativa

La programación imperativa se centra en describir cómo lograr un resultado específico paso a paso. El código en un enfoque imperativo se compone de instrucciones que modifican el estado del programa. Los programadores se centran en "cómo" se debe realizar una tarea.

La programación declarativa se centra en describir "qué" se debe lograr sin preocuparse por los detalles de implementación específicos. Los programadores se centran en definir el resultado deseado sin especificar los pasos detallados para lograrlo.

En resumen, mientras que la programación imperativa se centra en cómo lograr un resultado paso a paso, la programación declarativa se centra en qué resultado se desea sin especificar los pasos detallados para lograrlo.

2. ¿Qué es la programación orientada a objetos? ¿Qué busca lograr?

La Programación Orientada a Objetos (POO) es un paradigma de programación, es decir, un modelo o un estilo de programación que nos da unas guías sobre cómo trabajar con él. Se basa en el concepto de clases y objetos. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de planos de código (clases) para crear instancias individuales de objetos. Organiza el código en torno a

"objetos", que son instancias de clases. Una clase define un conjunto de propiedades y comportamientos comunes que comparten los objetos que pertenecen a esa clase.

Con el paradigma de Programación Orientado a Objetos lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma. La programación orientada a objetos busca:

Reutilización de Código:

- La POO facilita la reutilización de código a través de la encapsulación y la herencia. Puedes crear nuevas clases basadas en clases existentes, evitando la duplicación de código.

Modelado del Mundo Real:

- Los objetos y clases en la POO a menudo se diseñan para reflejar entidades del mundo real. Esto hace que el código sea más intuitivo y fácil de comprender para los desarrolladores y más cercano a los conceptos del dominio del problema.

Organización y Mantenimiento:

- La POO proporciona una forma de organizar el código en módulos independientes (clases) que encapsulan funcionalidades específicas. Esto facilita la modularidad, el mantenimiento y la mejora del código a medida que los sistemas crecen.

Flexibilidad y Escalabilidad:

- La POO proporciona una base sólida para escribir código flexible y escalable. Los conceptos como la herencia y el polimorfismo permiten adaptar el código a medida que los requisitos cambian y evolucionan.

3. Clase vs. objeto vs. instancia

Una clase es una plantilla. Define las propiedades y comportamientos comunes a todos los objetos que pertenecen a esa clase. Es un tipo de dato definido por el programador que encapsula datos y operaciones en una unidad lógica.

Un objeto es una instancia específica de una clase. Se crea a partir de la plantilla proporcionada por la clase y representa un "ejemplar" o "instancia" única de esa clase. Un objeto tiene propiedades específicas y puede realizar las operaciones definidas por la clase.

El término "instancia" se utiliza a menudo de manera intercambiable con "objeto". Una instancia es simplemente una ocurrencia única de un objeto. Cuando creas un objeto de una clase, estás creando una instancia de esa clase.

4. Describa los 4 principios de la POO

Encapsulamiento:

- La encapsulación implica ocultar la implementación interna de un objeto y exponer solo la interfaz necesaria para interactuar con él. Esto se logra mediante el uso de clases y la definición de métodos y atributos públicos y privados. Contiene toda la información importante de un objeto dentro del mismo y solo expone la información seleccionada al mundo exterior.

Abstracción:

- La abstracción consiste en simplificar la complejidad del sistema modelando entidades relevantes de manera abstracta. Las clases proporcionan una forma de abstraer propiedades y comportamientos comunes, permitiendo a los programadores centrarse en conceptos de alto nivel.

Herencia:

- La herencia permite crear nuevas clases basadas en clases existentes, aprovechando y extendiendo su funcionalidad. Las clases principales extienden atributos y comportamientos a las clases secundarias. La clase base se llama "superclase" y la nueva clase se llama "subclase". La herencia facilita la reutilización de código y la creación de jerarquías de clases.

Polimorfismo:

- El polimorfismo permite a un objeto comportarse de múltiples maneras. Puede referirse a diferentes clases o tipos de manera transparente, dependiendo del contexto. Es la capacidad de presentar la misma interfaz para diferentes formas subyacentes o tipos de datos. Al utilizar la herencia, los objetos pueden anular los comportamientos principales compartidos, con comportamientos secundarios específicos. El polimorfismo permite que el mismo método ejecute diferentes comportamientos de dos formas: anulación de método y sobrecarga de método.

5. Sintaxis de una clase en python

```python
class NombreDeLaClase:
    """Docstring de la clase."""

    # Atributos de clase (opcional)
    atributo_clase = valor

    # Constructor (__init__)
    def __init__(self, parametro1, parametro2, ...):
        # Atributos de instancia
        self.atributo_instancia1 = parametro1
        self.atributo_instancia2 = parametro2

    # Métodos de instancia
    def metodo(self):
        # Cuerpo del método
        pass

    # Otros métodos...
```

Clase 4

1. ¿Qué puede decir de la función super()?

La función super es una función builtin que retorna un objeto “proxy” (un objeto temporal de la superclase) lo que nos permite acceder a los métodos de la clase base. Nos evita tener que usar el nombre de la clase base explícitamente y trabajar con herencia múltiple. Es una función que se utiliza dentro de una clase para llamar a un método de la superclase (clase base) en lugar de la implementación actual en la subclase. Esta función es comúnmente utilizada en situaciones de herencia y se utiliza para acceder y llamar a métodos y atributos de la superclase.

2. ¿Que es una función de orden superior?

Una función es llamada de orden superior si esta contiene otras funciones como parámetros de entrada o retorna una función como output. Por ejemplo, las funciones que operan con otras funciones son conocidas como de orden superior. Es una función que cumple al menos uno de los siguientes criterios:

- Acepta una o más funciones como argumentos (parámetros).
- Devuelve una función como resultado.

Algunas propiedades:

- Una funciones es una instancia del tipo objeto
- Se puede almacenar funciones en variables
- Se puede pasar funciones como parámetros de otras funciones
- Se puede retornar una función desde una función
- Se puede almacenar en estructuras de datos como tablas hash, listas, etc.

Clase 6

1. ¿Qué es UTC y de donde se obtiene?

UTC, que significa Tiempo Universal Coordinado en inglés (Coordinated Universal Time), es una escala de tiempo internacionalmente utilizada como estándar de referencia. Es la base para la medición del tiempo en todo el mundo y está diseñado para ser independiente de la ubicación geográfica o la estación del año. A diferencia de las zonas horarias, UTC no está vinculado a ninguna ubicación geográfica específica. Las zonas horarias se derivan de UTC, pero pueden tener ajustes y variaciones locales. UTC se obtiene a través de una red mundial de relojes atómicos altamente precisos y sistemas de referencia de tiempo. Diversas instituciones y laboratorios alrededor del mundo contribuyen a la generación de UTC. El Bureau International des Poids et Mesures (BIPM) en Francia es responsable de coordinar y mantener UTC. El tiempo en UTC se mide en segundos desde el 1 de enero de 1970, a las 00:00:00 UTC, conocido como el "Epoch" en la terminología de la programación.

2. Escriba la fecha de hoy con la hora 8:00 am en formato 8601

El formato ISO 8601 para representar fechas y horas es "YYYY-MM-DDThh:mm:ss".
Para 19/12: 2023-12-19T08:00:00

Clase 11

1. ¿Qué es http?

Hypertext Transfer Protocol (HTTP) (o Protocolo de Transferencia de Hipertexto en español) es un protocolo de la capa de aplicación para la transmisión de documentos hipermedia, como HTML. Fue diseñado para la comunicación entre los navegadores y servidores web, aunque puede ser utilizado para otros propósitos también. Sigue el clásico modelo cliente-servidor, en el que un cliente establece una conexión, realizando una petición a un servidor y espera una respuesta del mismo. Se trata de un protocolo sin estado, lo que significa que el servidor no guarda ningún dato (estado) entre dos peticiones. Aunque en la mayoría de casos se basa en una conexión del tipo TCP/IP, puede ser usado sobre cualquier capa de transporte segura o de confianza, es decir, sobre cualquier protocolo que no pierda mensajes silenciosamente, tal como UDP.

2. ¿Cuáles son los métodos más comunes de http?

HTTP define un conjunto de métodos de petición para indicar la acción que se desea realizar para un recurso determinado. Aunque estos también pueden ser sustantivos, estos métodos de solicitud a veces son llamados HTTP verbs. Cada uno de ellos implementa una semántica diferente, pero algunas características similares son compartidas por un grupo de ellos.

- **GET:** Obtiene el estado actual de un recurso.
- **POST:** Crea un nuevo recurso.
- **PUT:** Modifica el estado de un recurso existente.
- **PATCH:** Realiza una actualización parcial de un recurso.
- **DELETE:** Elimina un recurso existente.

3. ¿Cómo se comunican cliente y servidor?

Clientes y servidores se comunican intercambiando mensajes individuales. Los mensajes que envía el cliente, normalmente un navegador Web, se llaman peticiones, y los mensajes enviados por el servidor se llaman respuestas. El cliente es cualquier herramienta que actué en representación del usuario. Es en la mayor parte de los casos un navegador Web. Hay excepciones, como el caso de programas específicamente usados por desarrolladores para desarrollar y depurar sus aplicaciones. El navegador es siempre el que inicia una comunicación a través de una solicitud al servidor. Al otro lado del canal de comunicación, está el servidor, el cual "sirve" los datos que ha pedido el cliente. Un servidor conceptualmente es una única entidad, aunque puede estar formado por varios elementos, que se reparten la carga de peticiones, (load balancing), u otros programas, que gestionan otros computadores (como caché, bases de datos, servidores de correo electrónico, etc.), y que generan parte o todo el documento que ha sido pedido.

Cuando el cliente quiere comunicarse con el servidor, tanto si es directamente con él, o a través de un proxy intermedio, realiza los siguientes pasos:

1. Abre una conexión TCP: la conexión TCP se usará para hacer una petición, o varias, y recibir la respuesta. El cliente puede abrir una conexión nueva, reusar una existente, o abrir varias a la vez hacia el servidor.
2. Hacer una petición HTTP: Los mensajes HTTP (previos a HTTP/2) son legibles en texto plano. A partir de la versión del protocolo HTTP/2, los mensajes se encapsulan

en franjas, haciendo que no sean directamente interpretables, aunque el principio de operación es el mismo.
3. Leer la respuesta enviada por el servidor.
4. Cierre o reuso de la conexión para futuras peticiones.

4. ¿Para qué sirve y cuál es el formato de una URL?

Una URL (Uniform Resource Locator) es una cadena de caracteres que se utiliza para identificar de manera única la ubicación de un recurso en la web. En otras palabras, una URL es la dirección que se utiliza para acceder a recursos en internet, como páginas web, imágenes, archivos, servicios, entre otros. Las URLs permiten a los usuarios especificar la ubicación precisa de un recurso y acceder a él mediante un navegador web u otras aplicaciones que manejen enlaces web.
Una URL, para poder ser funcional, debe tener el siguiente formato: **“esquema://host.dominio:puerto/directorio/nombre_de_archivo”**.

1. **Esquema:** es el protocolo utilizado para la solicitud y recepción de datos entre las partes involucradas.
2. **Host:** es donde se alberga el recurso a solicitar en la mayoría de casos “WWW” (las siglas de World Wide Web).
3. **Dominio:** Identifica el servidor que aloja el recurso.
4. **Puerto:** se establece el número de puerto que será usado como punto de conexión. Aunque en la mayoría de direcciones esto suele omitirse.
5. **Directorio:** Define en qué parte del dominio se encuentra el material solicitado.
6. **Nombre del archivo:** identifica cual es el recurso al que se desea acceder.

5. ¿Para que se usan los códigos de estado y cuáles son las clases en las cuales se agrupan?

HTTP utiliza códigos de estado en las respuestas para indicar el resultado de una solicitud. Las respuestas se agrupan en cinco clases:

- Respuestas informativas (100–199),
- Respuestas satisfactorias (200–299),
- Redirecciones (300–399),
- Errores de los clientes (400–499),
- Errores de los servidores (500–599).

Clase 11.2

1. ¿Qué es un patrón de diseño?

Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código. No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

2. ¿Cuáles son los grupos de los patrones de diseño?

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña. Además, todos los patrones pueden clasificarse por su propósito en tres grupos generales de patrones:

- **Creacionales:** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
- **Estructurales:** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
- **Comportamiento:** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

Creacionales:

- **Singleton:** Garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.
- **Factory:** Define una interfaz para crear un objeto, pero deja que las subclases alteren el tipo de objetos que se crearán.

Estructurales:

- **Adapter:** Permite que interfaces incompatibles trabajen juntas.
- **Decorator:** Añade comportamientos a objetos individuales de manera dinámica y transparente.

Comportamiento:

- **Observer:** Define una dependencia uno a muchos entre objetos para que cuando uno cambie de estado, todos sus dependientes sean notificados y actualizados automáticamente.
- **Strategy:** Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Es útil cuando tenemos varias formas de realizar una tarea y queremos que el cliente pueda elegir la estrategia que mejor se adapte a sus necesidades. Permite cambiar el comportamiento de un objeto en tiempo de ejecución sin alterar su estructura. En términos de POO, el patrón Strategy utiliza la herencia y la composición para permitir que los algoritmos se puedan intercambiar en tiempo de ejecución. En este patrón, se define una interfaz común para todos los algoritmos y se implementan diferentes clases que proporcionan diferentes implementaciones de dicho algoritmo.

3. ¿Que es un patrón de arquitectura? Describa alguno que conozca

La elección de una estructura o método en el que encajar todas las piezas o componentes es vital para obrar un sistema escalable que de solución a los problemas del cliente. La arquitectura de software, es precisamente eso, un método/patrón que se utiliza de base para el diseño de las diferentes funcionalidades. Son formas de capturar estructuras de diseño de probada eficacia, para que puedan ser reutilizadas. Más específicamente, un patrón arquitectónico es un paquete de decisiones de diseño que se encuentra repetidamente en la práctica, tiene propiedades bien definidas que pueden ser reutilizadas y describe una clase de arquitecturas. Se refiere a una solución estructural de alto nivel que aborda un conjunto particular de problemas de diseño. A diferencia de los patrones de diseño, que se centran en problemas específicos y en la interacción entre objetos, los

patrones de arquitectura se centran en la organización y estructura de sistemas y subsistemas completos. Estos patrones proporcionan un marco general para la creación de arquitecturas de software robustas y escalables, y suelen involucrar la disposición de componentes, la asignación de responsabilidades y la definición de las relaciones entre ellos.

El patrón de software más común es el patrón arquitectónico en capas. Los patrones de arquitectura en capas son patrones de *n* niveles donde los componentes están organizados en capas horizontales. Este es el método tradicional para diseñar la mayoría de los programas informáticos y está destinado a ser auto-independiente. Esto significa que todos los componentes están interconectados pero no dependen unos de otros. Cada capa del patrón de arquitectura en capas tiene un papel y una responsabilidad específicos dentro de la aplicación. Una variante de este patrón es el Tres capa, que en síntesis es lo mismo, pero tiene exactamente 3 capas:

- Presentación (vistas)
- Negocio(funcionalidades)
- Datos (modelos de base de datos)

Otros son:

- Arquitectura de microservicios
- Arquitectura orientada a servicios
- Arquitectura de microkernel
- Event-based pattern

Clase 11.3

1. ¿Qué es MVC y cual es la diferencia con MVT?

MVC es un patrón arquitectural, un modelo o guía que expresa cómo organizar y estructurar los componentes de un sistema software, sus responsabilidades y las relaciones existentes entre cada uno de ellos. Su nombre, MVC, parte de las iniciales de Modelo - Vista - Controlador (Model-View-Controller, en inglés), que son las capas o grupos de componentes en los que organizaremos nuestras aplicaciones bajo este paradigma. La arquitectura MVC propone, independientemente de las tecnologías o entornos en los que se base el sistema a desarrollar, la separación de los componentes de una aplicación en tres grupos (o capas) principales: el modelo, la vista, y el controlador, y describe cómo se relacionarán entre ellos para mantener una estructura organizada, limpia y con un acoplamiento mínimo entre las distintas capas.

- El Modelo contiene principalmente las entidades que representan el dominio, la lógica de negocio, y los mecanismos de persistencia de nuestro sistema.
- En la Vista encontraremos los componentes responsables de generar la interfaz con el exterior, por regla general, aunque no exclusivamente, el UI de nuestra aplicación.
- En el Controlador se encuentran los componentes capaces de procesar las interacciones del usuario, consultar o actualizar el Modelo, y seleccionar las vistas apropiadas en cada momento.

Ambos patrones, MVC y MVT, comparten similitudes en sus conceptos básicos, pero se asocian comúnmente con diferentes entornos y tecnologías. La principal diferencia es el componente de "Templado" en lugar del "Controlador". El templado se ocupa de la

presentación y la lógica de presentación, mientras que la vista maneja las interacciones del usuario y actualiza el modelo. En MVC, el controlador maneja las interacciones del usuario y actualiza el modelo, mientras que la vista se encarga de presentar la interfaz de usuario. En MTV o MVT, la vista maneja las interacciones del usuario y la lógica de presentación, y utiliza un "templado" para definir cómo se presenta la información.

2. ¿Qué es MVT? Desarrolle y muestre cómo se comunican las partes

En la práctica el patrón MTV es muy similar al MVC a tal punto que se puede decir que Django es un framework MVC. Realmente este no se desvía demasiado del patrón Modelo Vista Controlador, simplemente lo implementa de una manera distinta y para evitar confusiones es llamado MTV. En Django, el controlador sigue estando presente, nada más que de una manera intrínseca, ya que todo el framework Django es el controlador.

Modelo:

- El Modelo se encarga de la lógica de negocio y la manipulación de datos.
- Representa la estructura de datos y la lógica necesaria para interactuar con ellos.
- Comunica cambios y actualizaciones a la Vista y al Templado.

Vista:

- La Vista maneja la presentación de los datos y la lógica de interacción con el usuario.
- Responde a las solicitudes del usuario y comunica con el Modelo y el Templado según sea necesario.
- Puede recibir datos del Modelo y renderizarlos para el usuario

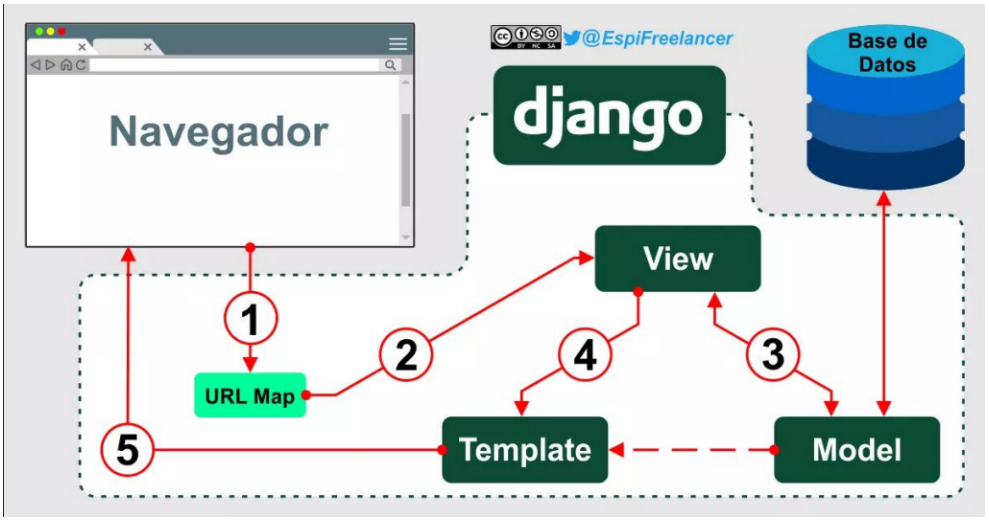
Templado:

- El Templado se encarga de la presentación y define cómo se muestran los datos en la interfaz de usuario.
- Utiliza un lenguaje de plantillas para insertar dinámicamente datos provenientes del Modelo.
- Se comunica con la Vista para recibir datos y presentarlos al usuario.

Flujo:

1. Cuando un usuario realiza una solicitud (por ejemplo, visita una URL en un navegador), la solicitud llega al sistema Django.
2. La Vista recibe la solicitud y decide qué datos son necesarios y qué acción debe tomar.
3. La Vista puede interactuar con el Modelo para obtener o modificar datos.
4. Luego, la Vista utiliza el Templado para renderizar la respuesta que será enviada al usuario.
5. El Modelo gestiona la lógica de negocio y la manipulación de datos.
6. Puede realizar operaciones en la base de datos, actualizar datos y notificar a la Vista sobre cambios.
7. El Templado recibe datos de la Vista y los utiliza para construir la respuesta final que será enviada al usuario.
8. Utiliza un lenguaje de plantillas para insertar dinámicamente los datos en el HTML.
9. La respuesta final (HTML generado) se envía al usuario a través del navegador.

10. Este proceso de solicitud y respuesta continúa mientras el usuario interactúa con la aplicación.



Clase 12

1. ¿Qué es la persistencia, que tipos conoce?

La persistencia en el contexto de la informática se refiere a la capacidad de almacenar y recuperar datos de manera duradera, es decir, a través de diferentes sesiones o reinicios del sistema. La persistencia asegura que los datos se conserven a largo plazo, incluso cuando la aplicación o el sistema que los utiliza se detiene o se reinicia.

Persistencia en Memoria (Memory Persistence):

- Los datos se almacenan en la memoria RAM del sistema. Esta forma de persistencia es efímera y se pierde cuando se apaga o reinicia el sistema. Es útil para almacenar datos temporales mientras la aplicación está en ejecución.

Persistencia en Disco (Disk Persistence):

- Los datos se almacenan en dispositivos de almacenamiento de datos no volátiles, como discos duros, SSDs, o incluso en sistemas de almacenamiento en la nube. Los datos persisten a través de reinicios y apagados del sistema.

Persistencia en Base de Datos (Database Persistence):

- Los datos se almacenan en bases de datos, que son sistemas especializados para el almacenamiento y recuperación eficiente de datos. Las bases de datos pueden ser de diferentes tipos, como bases de datos relacionales (SQL), bases de datos NoSQL (MongoDB, Cassandra, etc.), y otras.

Persistencia en Archivos (File Persistence):

- Los datos se almacenan en archivos en un sistema de archivos. Pueden ser archivos de texto, binarios o en otros formatos específicos. La persistencia en archivos es comúnmente utilizada para almacenar configuraciones, registros, y otros datos.

Persistencia en Red (Network Persistence):

- Los datos se almacenan en nodos de una red, y la persistencia se logra distribuyendo los datos en diferentes ubicaciones. Este enfoque es común en sistemas distribuidos y aplicaciones que utilizan arquitecturas de microservicios.

Persistencia en Caché (Cache Persistence):

- Los datos se almacenan en una caché, que es una memoria de acceso rápido diseñada para mejorar la velocidad de acceso a datos comúnmente utilizados. La persistencia en caché puede ser temporal y depende de la política de la caché.

Persistencia en Tiempo (Time Persistence):

- Se refiere a la capacidad de mantener la persistencia durante un período específico de tiempo. Algunos datos pueden ser válidos y persistir solo por un tiempo determinado antes de ser eliminados o actualizados.

2. ¿Qué es un ORM? ¿Se puede usar en bases de datos relaciones y no relacionales?

Un ORM, o Mapeador Objeto-Relacional (Object-Relational Mapper, en inglés), es una herramienta que facilita la interacción entre una aplicación escrita en un lenguaje de programación orientado a objetos y una base de datos relacional. La función principal de un ORM es permitir que las aplicaciones manipulen datos en la base de datos utilizando objetos y métodos en lugar de sentencias SQL directas.

Originalmente, los ORM se diseñaron principalmente para bases de datos relacionales. Sin embargo, con la creciente popularidad de las bases de datos NoSQL (como MongoDB, Cassandra, etc.), algunos ORM han evolucionado para admitir también este tipo de bases de datos. Si bien los ORM tradicionales están optimizados para trabajar con bases de datos relacionales y tablas estructuradas, existen también ORMs adaptados a bases de datos NoSQL. En estos casos, la adaptación implica manejar las particularidades de las bases de datos NoSQL, como la falta de esquemas rígidos, la gestión de documentos en lugar de tablas, etc. Es importante tener en cuenta que el uso de un ORM en bases de datos NoSQL puede tener limitaciones o implicar ciertas consideraciones de rendimiento, ya que estos sistemas están diseñados con modelos de datos y paradigmas de acceso distintos a las bases de datos relacionales.

3. ¿Qué pasa si quiero cambiar el motor de base de datos en un proyecto de Django? ¿Cómo lo hago?

Django proporciona herramientas y flexibilidad para facilitar este tipo de migración.

- Abrir el archivo settings.py en tu proyecto Django.
- Ir a la sección que define la configuración de la base de datos (DATABASES).
- Cambia el valor de ENGINE al nuevo motor de base de datos que se desea usar.
- Ajusta las demás configuraciones según sea necesario.

```
DATABASES = {
    'default': {
        'ENGINE': 'nuevo_motor',
        'NAME': 'nombre_de_la_base_de_datos',
        'USER': 'usuario',
        'PASSWORD': 'contraseña',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

- Ejecutar el comando "***python manage.py makemigrations"*** para crear las migraciones necesarias para el nuevo motor de base de datos.
- Ejecutar el comando ***"python manage.py migrate"*** para aplicar las migraciones y actualizar la estructura de la base de datos según el nuevo motor.

Clase 13

1. ¿Qué son los test? ¿Por qué se hacen?

Son procesos mediante los cuales se evalúa el comportamiento de un sistema, una aplicación o una parte específica del código para asegurar su correcto funcionamiento.
Es el proceso de evaluar y verificar que un producto o aplicación de software hace lo que se supone que debe hacer. Los beneficios de las pruebas incluyen la prevención de errores, la reducción de los costos de desarrollo y la mejora del rendimiento. Estas pruebas son fundamentales para garantizar la calidad del software y son una parte integral de las prácticas de desarrollo ágil.

Pruebas Unitarias (Unit Tests):

- Se centran en verificar que unidades individuales de código (como funciones o métodos) funcionen correctamente. A menudo se escriben por los propios desarrolladores.

Pruebas de Integración (Integration Tests):

- Verifican que las diferentes partes del sistema interactúen correctamente. Pueden incluir la integración de módulos, servicios o incluso sistemas completos.

Pruebas de Aceptación del Usuario (User Acceptance Tests - UAT):

- Simulan escenarios de uso del usuario final para garantizar que la aplicación cumpla con los requisitos y expectativas del usuario.

Pruebas de Regresión (Regression Tests):

- Se realizan para asegurar que las nuevas actualizaciones o cambios no afecten negativamente a funcionalidades existentes.

Pruebas Funcionales (Functional Tests):

- Evalúan si el sistema cumple con sus especificaciones funcionales, es decir, si realiza las funciones previstas correctamente.

Pruebas de Rendimiento (Performance Tests):

- Se enfocan en evaluar la velocidad, capacidad y estabilidad del sistema bajo diferentes condiciones de carga.

Los test se hacen para:

Detectar Errores Temprano:

- Los tests permiten identificar y corregir errores en una etapa temprana del desarrollo, lo que ayuda a reducir costos y tiempo de corrección.

Mejorar la Calidad del Software:

- Las pruebas contribuyen a la creación de software más confiable, robusto y libre de errores, mejorando la calidad general del producto.

Facilitar el Mantenimiento:

- Las pruebas facilitan la detección de problemas cuando se realizan cambios o actualizaciones en el código, lo que simplifica el proceso de mantenimiento.

Asegurar la Funcionalidad:

- Las pruebas garantizan que el software cumpla con los requisitos y funcionalidades especificadas.

Facilitar la Colaboración:

- Facilitan la colaboración entre miembros del equipo de desarrollo, ya que proporcionan una forma estandarizada de verificar el comportamiento del software.

Mejorar la Documentación:

- Los tests sirven como documentación ejecutable que describe cómo se espera que funcione el código.

En resumen, los tests son esenciales en el desarrollo de software porque aseguran que el código cumpla con los requisitos, funcione como se espera y se mantenga de manera eficiente a lo largo del tiempo. Además, contribuyen significativamente a la confiabilidad y calidad general del software.

2. ¿Cuáles son los objetivos que se buscan al realizar test?

Identificar Errores:

- El objetivo principal de las pruebas es detectar errores y defectos en el software. Esto incluye bugs, malentendidos de requisitos, y otros problemas que puedan afectar el funcionamiento del sistema.

Verificar la Funcionalidad:

- Las pruebas se realizan para verificar que el software realiza las funciones especificadas en los requisitos. Aseguran que todas las características funcionen correctamente.

Asegurar la Estabilidad:

- Buscan garantizar la estabilidad del sistema bajo diferentes condiciones. Las pruebas de estabilidad, rendimiento y carga ayudan a identificar posibles problemas de escalabilidad y rendimiento.

Garantizar la Seguridad:

- Las pruebas de seguridad se realizan para identificar vulnerabilidades y asegurar que el software no sea susceptible a ataques maliciosos.

Validar la Usabilidad:

- Las pruebas de usabilidad se centran en validar que la interfaz de usuario sea fácil de usar y cumpla con las expectativas del usuario final.

Facilitar el Mantenimiento:

- Buscan asegurar que el software sea fácil de mantener y que los cambios no introduzcan errores en funcionalidades existentes (pruebas de regresión).

Validar Requisitos:

- Verifican que el software cumpla con los requisitos del usuario y del sistema. Ayudan a garantizar que se desarrolla el producto correcto.

Minimizar los Costos de Corrección:

- Al detectar y corregir errores temprano en el proceso de desarrollo, las pruebas ayudan a minimizar los costos asociados con la corrección de problemas en etapas más avanzadas del ciclo de vida del software.

3. ¿Cuáles son los dos tipos de pruebas que se pueden hacer, mencione algunas de ambos grupos.

Pruebas de Caja Blanca (White-Box Testing):

En las pruebas de caja blanca, el evaluador tiene conocimiento detallado de la estructura interna y del código fuente del software. Estas pruebas se centran en evaluar la lógica interna, las rutas de ejecución y las estructuras de datos del programa. Algunos tipos de pruebas de caja blanca incluyen:

- Pruebas Unitarias
- Pruebas de Cobertura de Código

Pruebas de Caja Negra (Black-Box Testing):

En las pruebas de caja negra, el evaluador no tiene conocimiento detallado de la estructura interna del sistema. Se enfoca en evaluar el comportamiento externo del software sin conocer la implementación interna. Algunos tipos de pruebas de caja negra incluyen:

- Pruebas Funcionales
- Pruebas de Integración
- Pruebas de Regresión
- Pruebas de Usabilidad
- Pruebas de Aceptación del Usuario
- Pruebas de Estrés (Stress Testing)

4. ¿Que es un plan de pruebas? ¿Qué significa definir la estrategia de pruebas?

Se llama Test Plan a un conjunto que combina suites y test cases. Es un documento detallado que describe el alcance, los enfoques, los recursos, el cronograma y los entregables asociados con las actividades de pruebas para un proyecto de software específico. Este documento es una parte esencial del Ciclo de Vida de Pruebas de Software (STLC) y sirve como guía para todo el proceso de pruebas. El Test Plan proporciona un enfoque estructurado para llevar a cabo las actividades de prueba y garantizar la calidad del software. Definir la estrategia de pruebas implica tomar decisiones clave sobre cómo se llevarán a cabo las pruebas en el proyecto. Algunos aspectos de la estrategia de pruebas incluyen:

Enfoque de Pruebas:

- Decide si se seguirá un enfoque de pruebas centrado en casos de uso, en la interfaz de usuario, en el rendimiento, etc.

Selección de Herramientas de Pruebas:

- Determina las herramientas de pruebas que se utilizarán, como marcos de pruebas automatizadas, herramientas de gestión de pruebas, etc.

Niveles de Pruebas:

- Define los diferentes niveles de pruebas que se llevarán a cabo, como pruebas unitarias, de integración, de sistema y de aceptación del usuario.

Tipos de Pruebas:

- Especifica los tipos de pruebas que se realizarán, como pruebas funcionales, no funcionales, de regresión, de seguridad, etc.

Criterios de Aceptación:

- Establece los criterios que se utilizarán para determinar si una prueba ha tenido éxito y si el software está listo para ser lanzado.

Requisitos de Pruebas:

- Define los requisitos específicos que se deben cumplir para llevar a cabo las pruebas, como la disponibilidad de entornos de prueba y datos de prueba.

Coordinación con el Desarrollo:

- Especifica cómo se coordinarán las actividades de pruebas con el desarrollo, incluyendo la frecuencia de las pruebas y la comunicación de los resultados.

5. Explique manual vs. automatizadas ¿Cuales son más importantes para CI/CD

Las pruebas manuales son ejecutadas por testers humanos sin el uso de herramientas automatizadas. Son más eficaces para explorar funcionalidades nuevas y no documentadas, adecuadas para pruebas de usabilidad y experiencia del usuario y útiles en situaciones donde las pruebas automatizadas pueden ser costosas o difíciles de

implementar. Como desventaja, son más lentas y propensas a errores humanos, no tan eficientes para repetir pruebas frecuentes y rutinarias y menos escalables, especialmente en proyectos grandes y complejos.

Las pruebas automatizadas son ejecutadas por herramientas de software y scripts predefinidos. Son más rápidas y consistentes que las pruebas manuales, eficientes para realizar pruebas repetitivas, escalables y útiles en proyectos grandes y en CI/CD. Como desventaja son ineficientes para explorar nuevas funcionalidades sin una actualización constante de los scripts, requieren inversión inicial en desarrollo y mantenimiento de scripts y no son tan efectivas para pruebas de usabilidad y experiencias de usuario complejas.

En el contexto de la Integración Continua y Despliegue Continuo (CI/CD), ambas son relevantes, pero la automatización de pruebas juega un papel crucial en asegurar una entrega rápida y confiable del software. En un entorno de CI/CD, las pruebas automatizadas son fundamentales.

Rápida Retroalimentación:

- Las pruebas automatizadas proporcionan una retroalimentación rápida sobre la calidad del código, permitiendo una identificación temprana de defectos.

Entrega Rápida:

- La automatización acelera el proceso de pruebas, lo que es crucial para lograr una entrega rápida y continua del software.

Integración Continua:

- La automatización se integra fácilmente con sistemas de CI/CD, permitiendo la ejecución automatizada de pruebas en cada cambio de código.

Repetibilidad:

- La automatización garantiza la repetibilidad en las pruebas, lo que es esencial para una implementación exitosa de CI/CD.