Checkpoint 6

Rosa Fraile

4 de marzo de 2024

Índice

1. ¿Para qué usamos Classes en Python?

1.1. Introducción

La programación orientada a objetos (POO) es una técnica muy útil para estructurar y organizar el código de manera lógica y coherente, lo que hace que el código sea más fácil de entender, mantener y mejorar. Por ello, el uso de clases en Python es una práctica muy común y recomendada para proyectos de software de cualquier tamaño y complejidad.

Las clases proveen una forma de empaquetar datos y funcionalidad juntos. Son de gran utilidad si, además, necesitas abstraer un conjunto de funciones en otro programa distinto.

Una clase define una plantilla o molde para crear objetos, los cuales son instancias de esa clase. Los objetos creados a partir de una clase tienen las mismas propiedades y comportamientos definidos por la clase, pero pueden tener valores diferentes para los atributos que se definen en la clase. Las instancias de clase también pueden tener métodos (definidos por su clase) para modificar su estado.

Comparado con otros lenguajes de programación, el mecanismo de clases de Python agrega clases con un mínimo de nuevas sintaxis y semánticas. Es una mezcla de los mecanismos de clases encontrados en C++ y Modula-3.

Las clases de Python proveen todas las características normales de la POO: el mecanismo de la herencia de clases permite múltiples clases base, una clase derivada puede sobre escribir cualquier método de su(s) clase(s) base, y un método puede llamar al método de la clase base con el mismo nombre. Los objetos pueden tener una cantidad arbitraria de datos de cualquier tipo. Igual que con los módulos, las clases participan de la naturaleza dinámica de Python: se crean en tiempo de ejecución, y pueden modificarse luego de la creación.

En Python, una clase se define mediante la palabra clave «class», seguida del nombre de la clase y dos puntos (:) y luego el cuerpo de la clase. La convención que se sigue es que el nombre de la clase tiene la primera letra en mayúscula y el resto en minúsculas. El cuerpo de la clase contiene definiciones de métodos y atributos, que pueden ser públicos o privados según su acceso.

1.2. Como crear una clase

Para definir una clase en Python, se usa la palabra clave çlass", seguida por el nombre de la clase y dos puntos (:). La convención es que el nombre de la clase tiene la primera letra mayúscula y el resto de las letras en minúscula.

Dentro de la clase hay que definir el método __init__. Este es eñ inicializador que puedes usar más adelante para instanciar objetos y siempre tiene que estar presente. Es similar a un método constructor en Java. Este método tiene al menos un argumento: self (es euivalente a "this.en C++ o Java).

Recuerda usar la indentación correctamente para identificar el bloque de sentencias asociadas a la clase!

```
1class Dog:
2    def __init__(self):
3        pass
```

Hemos creado la clase pero todavía no tenemos ningún objeto. Creemos uno.

1.3. Como instanciar un objeto

Para instanciar un objeto, teclea el nombre de la clase seguida por dos paréntesis. Puedes asignarlo a una variable para poder seguir el rastro al objeto.

```
1ozzy = Dog()
```

Y lo imprimimos:

```
print(ozzy)

<__main__.Dog object at 0x111f47278>
```

1.4. Como añadir atributos a una clase

Después de imprimir ozzy está claro que este objeto es un perro. Pero no se le han añadido atributos todavía. Vamos a darle a la clase Dog un nombre y una edad reescribiéndolo:

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Puedes ver que ahora la función tiene dos argumentos más después de self: name y age. Estos entonces se asignan a self.name y self.age respectivamente. Ahora puedes crear un nuevo objeto ozzy con nombre y edad:

```
ozzy = Dog("Ozzy", 2)
```

Para acceder a los atributos de un objeto puedes usar el punto ".". Esto se hace escribiendo el nombre del objeto, seguido por un punto y el nombre del atributo.

```
print(ozzy.name)     # Ozzy

print(ozzy.age)     # 2
```

Esto se puede combinar también en una frase más elaborada:

```
print(ozzy.name + " is " + str(ozzy.age) + " year(s) old.")     ↩
    # Ozzy is 2 year(s) old.
```

1.5. Como definir métodos en una clase

Ahora ya tienes una clase, que tiene un nombre y una edad como atributos, y hay un objeto instanciado almacenado en una variable. Pero realmente no hace nada. Vamos a reescribir la clase para incluir el método bark(). Fíjate en que la palabra clave def se usa otra vez, así como el argumento self:

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("bark bark!")
```

Puede hacerse una llamada ahora al método bark usando la notación "."después de instanciar un nuevo objeto ozzy. El método escribirá "bark, bark!.[en] la pantalla. Fñijate en los paréntesis en bark(). Hay que ponerlos siempre que llamemos a un método. Están vacios en este caso, ya que el método bark() no tiene argumentos.

```
ozzy = Dog("Ozzy", 2)

ozzy.bark()     # bark, bark!
```

¿Recuerdas cómo se imprimió ozzy anteriormente? El código que está a continuación ahora implementa esta funcionalidad en la clase Dog, además del método doginfo(). Entonces instanciaremos algunos objetos con diferentes propiedades, y llamaremos el nuevo método en ellos:

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("bark bark!")

    def doginfo(self):
        print(self.name + " is " + str(self.age) + " year(s) ↩
           old.")

ozzy = Dog("Ozzy", 2)
skippy = Dog("Skippy", 12)
filou = Dog("Filou", 8)

ozzy.doginfo()     # Ozzy is 2 year(s) old.
skippy.doginfo()     # Skippy is 12 year(s) old.
filou.doginfo()     # Filou is 8 year(s) old.
```

Como puedes ver, puedes hacer una llamada al métod doginfo() con la notación de punto. La respuesta ahora dependerá de con que objeto de Dog llames al método

Los perros se hacen mayores, por lo que sería aconsejable adjustar su edad. Ozzy ha cumplido 3 años, por lo que vamos a cambiar su edad.

```
ozzy.age = 3

print(ozzy.age)     # 3
```

Es tan fácil como asignar un nuevo valor al atributo. Podríamos implementar esto con el método birthday() en la clase Dog:

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("bark bark!")

    def doginfo(self):
        print(self.name + " is " + str(self.age) + " year(s) ↩
           old.")
```

```
12
13      def birthday(self):
14          self.age +=1
15
16  ozzy = Dog("Ozzy", 2)
17
18  print(ozzy.age)     # 2
19
20  ozzy.birthday()
21
22  print(ozzy.age)     # 3
```

Ahora ya no necesitas cambiar su edad manualmente. Basta con llamar al método birthday().

1.6. Como pasar argumentos a los métodos

¿Te gustaría que nuestros perros tuvieran un colega. Esto sería opcional, ya que no todos los perros son sociables. Fíjate en el método setBuddy() abajo. Tiene self, como es usual, y buddy como argumentos. En este caso buddy será otro objeto de la clase Dog. Asigns buddy al atributo self.buddy, y el atributo self.name a buddy.buddy. Esto significa que la relación es recíproca, eres colega de tu colega. En este caso, Filou será el colega de Ozzy, lo que significa que Ozzy automáticamente se convierte en el colega de Filou. Podrías asignar estos atributos manualmente en vez de definir un método, pero eso requeriría más trabajo (escribir dos líneas de código en vez de una) cada vez que quieres asignar un colega.

```
1  class Dog:
2
3      def __init__(self, name, age):
4          self.name = name
5          self.age = age
6
7      def bark(self):
8          print("bark bark!")
9
10     def doginfo(self):
11         print(self.name + " is " + str(self.age) + " year(s) ↩
               old.")
12
13     def birthday(self):
14         self.age +=1
15
16     def setBuddy(self, buddy):
17         self.buddy = buddy.name
18         buddy.buddy = self.name
19
20 ozzy = Dog("Ozzy", 2)
21 filou = Dog("Filou", 8)
22
23 ozzy.setBuddy(filou)
24
25 print(ozzy.buddy)     # Filou
26 print(filou.buddy)     # Ozzy
```

Si ahora quieres obtener información sobre el colega de Ozzy puedes usar la notación de punto dos veces: primero para referirse al colega de Ozzy, y una segunda vez para referirse a su atributo.

```
print(ozzy.buddy.name)     # Filou
print(ozzy.buddy.age)     # 8
```

Fíjate que esto tambien puede hacerse para Filou.

```
print(filou.buddy.name)     # Ozzy
print(filou.buddy.age)     # 2
```

Tambien se puede llamar a los métodos del colega. El argumento self que se pasa to doginfo() is ahora ozzy.buddy, el cual es filou.

```
ozzy.buddy.doginfo()     # Filou is 8 year(s) old.
```

1.7. Ventajas y desventajas del uso de las clases en Python

1.7.1. Ventajas

- Reutilización de código: las clases pueden reutilizarse en diferentes partes del programa o en distintos programas, lo que ahorra tiempo y reduce la duplicación de código.

- Encapsulación: permiten ocultar la complejidad de un objeto y exponer solo una interfaz simple y fácil de usar para interactuar con él.

- Modularidad: pueden descomponer un programa en componentes más pequeños y manejables, lo que facilita el mantenimiento y la solución de problemas.

- Polimorfismo: ayudan a implementar el mismo conjunto de métodos con diferentes comportamientos para distintos tipos de objetos, lo que permite una mayor flexibilidad y extensibilidad en el diseño de programas.

1.7.2. Desventajas

- Sobrecarga de complejidad: las clases pueden agregar complejidad adicional a un programa y hacer que sea más difícil de entender y depurar.

- Curva de aprendizaje: el aprendizaje de las clases y la programación orientada a objetos en general pueden requerir una curva de aprendizaje más pronunciada para los programadores principiantes.

- Uso innecesario: a veces, las clases se utilizan innecesariamente en situaciones en las que una función simple podría haber hecho el trabajo de manera más eficiente.

1.8. Ejercicios

1.8.1. Ejercicio 1

Escribir una clase en python llamada rectangulo que contenga una base y una altura, y que contenga un método que devuelva el área del rectángulo. Instancia un objeto y escribe el área del mismo en pantalla

1.8.2. Ejercicio 2

Escribir una clase en python llamada circulo que contenga un radio, con un método que devuelva el área y otro que devuelva el perímetro del circulo .Instancia un objeto y escribe el área y el perímetro del mismo en pantalla.

1.9. Soluciones

1.9.1. Solución ejercicio 1

```
class Rectangulo:

    def _ _ init_ _ (self, lado_uno, lado_dos):
        self.lado_uno = lado_uno
        self.lado_dos = lado_dos

    def area(self)
        return lado_uno * lado_dos


rectangulo = Rectangulo(2, 4)

print(rectangulo.area())    # 8
```

1.9.2. Solución ejercicio 2

```
import math

class Circulo:

    def _ _ init_ _ (self, radio):
        self.radio = radio

    def area(self)
        return math.pi * radio * radio

    def perimetro(self)
        return 2 * math.pi * radio


circulo = Circulo(5)

print(circulo.area())    # 78.54
print(circulo.perimetro())    # 31.41
```

1.10. Bibliografía

- https://docs.python.org/es/3/tutorial/classes.html
- https://blog.hubspot.es/website/clases-python
- https://docs.python.org/es/3/tutorial/classes.html
- https://www.datacamp.com/tutorial/python-oop-tutorial
- https://pythondiario.com/2015/11/ejercicios-de-clases-y-objetos-en.html

2. ¿Qué método se ejecuta automáticamente cuando se crea una instancia de una clase?

2.1. Método constructor

El Método Constructor es el primer método que se ejecuta por defecto cuando creamos una clase, nos permite crear nuevas instancias de una clase. No devuelve ningún valor, es decir, no necesitamos un 'return'.

Los constructores son los componentes esenciales de una clase; sin estos, no habría posibilidad de hacer instancias sobre las clases de nuestros programas de código. Su función es asignar valores a los elementos de clase cuando se crea un objeto de esa misma clase.

Cuando un objeto se crea en Python, se inicializa el constructor con la siguiente instrucción:

```
def _ _ init_ _ (self).
```

En cuanto a la palabra **self**, lo que indica es que es una función que afecta solamente al objeto que creamos, por lo que estos objetos son independientes entre ellos.

Se pueden identificar dos categorías de constructores: **el constructor por defecto**, que no admite la inclusión de argumentos adicionales, y **el constructor parametrizado**. En este último, el primer parámetro, que siempre es el objeto en sí (self), se proporciona automáticamente. Los demás parámetros son especificados por el usuario o el programador.

2.2. Ejemplo práctico de constructor parametrizado

Imaginemos que queremos comprar un coche. Obviamente tienes clarísimo lo que es un coche, pero es importante que te sientes un momento conmigo y definamos bien lo que conforma un coche. Este coche tiene variables como:

- La matrícula
- La marca
- Su velocidad máxima
- Si es automático o manual
- ...

Ahora que entendemos lo que conforma un coche, vamos a crear uno. Vamos a crear un coche de la marca Teslo (no quiero referirme a ninguna marca), con una matrícula X y digamos que es automático. Vamos a llamar a este coche **Teslo road**.

Teslo road es nuestro objeto. Podríamos crear otros objetos de tipo coche con otros valores y serían objetos independientes. La definición de **coche** sería lo que se conoce como una clase. Y todos los objetos se crear a partir de clases.

Vamos a añadir las variables que definí antes a la clase.

```
class Coche:
    __init__(self, matricula, marca, velocidad_max, automatico)↩
        :
        self.matricula = matricula
        self.marca = marca
        self.velocidad_max = velocidad_max
        self.automatico = automatico
```

Con esto indicamos que cuando inicialicemos el constructor, el objeto tendrá los valores que asignamos.

```
teslo_road = Coche('X', 'Teslo', 200, True)
```

2.3. Argumentos opcionales para un constructor de clase en Python

Un argumento opcional es un argumento para el que no es obligatorio pasar un valor. Para tal argumento, se evalúa un valor predeterminado. Si se pasa algún valor para dicho argumento, el nuevo valor sobrescribe el valor predeterminado.

Agregar valores predeterminados es una tarea sencilla. Tenemos que equiparar los argumentos a sus valores predeterminados como x = 3, nombre = 'Untitled', etc. Se considerará el valor predeterminado si no se pasan valores para estos argumentos opcionales.

Veamos un ejemplo:

```
class Point:
    def __init__(self, x, y, z=0):
        self.x = x
        self.y = y
        self.z = z

point_one = Point(1, 2)  # X: 1, Y: 2, Z:0
point_two = Point(54, 92, 8)  # X: 54, Y: 92, Z:8
point_three = Point(99, 26, 100)  # X: 99, Y: 26, Z:100
```

2.4. Ejercicios

2.4.1. Ejercicio 1

Vamos a crear una clase llamada Persona. Sus atributos son: nombre, edad y DNI. Construye el método constructor e instancia un objeto.

2.4.2. Ejercicio 2

Crea una clase llamada Cuenta que tendrá los siguientes atributos: titular (que es una persona) y cantidad (puede tener decimales). Crea un constructor donde el titular será obligatorio y la cantidad es opcional (si no se indica nada, el valor por defecto será 0). Instancia un objeto.

2.5. Bibliografía

- https://keepcoding.io/blog/como-definir-un-constructor-en-python/
- https://www.themachinelearners.com/clases-python/
- https://www.delftstack.com/es/howto/python/python-class-optional-arguments/

3. ¿Qué es una API?

3.1. Introducción

Una API es el código que determina el funcionamiento de un programa informático que sirve para canalizar información de una parte de un software a otra. Las API permiten que una aplicación extraiga archivos o datos preexistentes dentro de un software y los use en otro programa o en uno de sus otros niveles.

En pocas palabras, una API es un intermediario que permite que tus herramientas extraigan, se comuniquen y trabajen con la información disponible que ha sido creada por otros desarrolladores. Pueden extraer formatos predefinidos, información de una base de datos o partes de un programa a otro.

La arquitectura de las API suele explicarse en términos de cliente y servidor. La aplicación que envía la solicitud se llama cliente, y la que envía la respuesta se llama servidor. En el ejemplo del tiempo, la base de datos meteorológicos del instituto es el servidor y la aplicación móvil es el cliente.

3.2. ¿Qué significa API?

API son las siglas en inglés para «application programming interface» o interfaz de programación de aplicaciones. Como su nombre lo indica, las API son interfaces compuestas por reglas y llamados en lenguaje computacional, que sirven para programar el funcionamiento de una aplicación determinada dentro de un software. En el contexto de las API, la palabra aplicación se refiere a cualquier software con una función distinta. La interfaz puede considerarse como un contrato de servicio entre dos aplicaciones. Este contrato define cómo se comunican entre sí mediante solicitudes y respuestas. La documentación de su API contiene información sobre cómo los desarrolladores deben estructurar esas solicitudes y respuestas.

3.3. ¿Para qué sirven las API?

Las API son un medio eficiente para recabar información de una herramienta digital a otra y así contar con funciones extendidas. Son una solución que permite hacer más específicas las características del software y así satisfacer las necesidades finas de diversos usuarios.

Pero, ¿cuál es el beneficio para las empresas de software? Esta es una de las primeras preguntas que se hacen muchos profesionales del marketing. Por lo general, la respuesta es la siguiente: la escalabilidad. A medida que las empresas de software crecen, su personal pronto descubre que tiene más ideas que tiempo y recursos para desarrollarlas.

Al crear las API, las empresas permiten que desarrolladores externos creen aplicaciones que pueden mejorar la utilización y la adopción de la plataforma principal. De este modo, una empresa puede crear un ecosistema que se vuelve dependiente de los datos de su API. Esta es una dinámica que a menudo lleva a oportunidades de ingresos adicionales.

Además, las API son necesarias para hacer más eficientes algunos procesos de diseño de un programa. Por ejemplo, si quieres añadir una tipografía o un color en un sitio web, resultaría innecesario llevar a cabo la programación de cada una de las letras y de los códigos de color. Una API sirve para extraer ese programa preexistente y aplicarlo en el código de tu aplicación.

3.4. ¿Qué es un ENDPOINT o punto de terminación de API?

Al ser un canal que conecta tu información con una herramienta que la trabajará, una API tiene un destino al cual debe dirigir los datos que ha extraído. A este programa de destino se le conoce como ENDPOINT o «punto de terminación de API» y puede tener la forma de una aplicación móvil, un sitio web o un programa de gestión de datos.

Un punto de terminación de API es el destino de la API que solicita el propietario de un sitio web. Si un sistema de gestión de contenidos (CMS) solicita acceso a una API, el CMS funciona como el punto de terminación de la API. Es importante que los sitios web funcionen correctamente para que puedan servir de puntos de terminación seguros para los desarrolladores que buscan compartir sus datos.

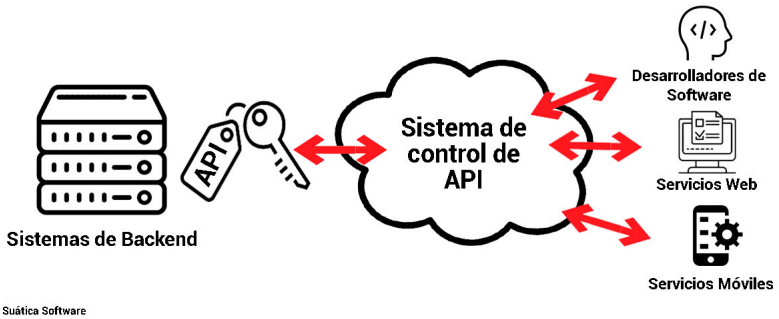
3.5. ¿Qué es una clave de interfaz de programación de aplicaciones (API)?

Una clave de interfaz de programación de aplicaciones (API) es un código único que una API usa para identificar a la aplicación o el usuario que la llama. Las claves API se utilizan con el fin de rastrear y controlar tanto a quien está usando una API como la forma en la que la usa, además de autenticar y autorizar aplicaciones, de forma similar a cómo funcionan los nombres de usuario y las contraseñas.

Muchos desarrolladores requieren que solicites esta clave antes de usar la API, mientras que otros pueden asignártela una vez que haces tu primera solicitud para recolectar información por medio de la misma.

Una clave API puede presentarse en forma de una clave única o de un grupo de varias claves. Los usuarios deben seguir las prácticas recomendadas para mejorar su seguridad general contra el robo de la clave API y evitar las consecuencias que se relacionan con tener claves API vulneradas.

Piensa en tu clave de API como si fuera tu ficha de autorización como miembro de la comunidad de un desarrollador. En efecto, esta ficha identifica para qué usas la API y verifica que hayas recibido permiso para realizar este proyecto de parte del propietario.



Tu clave de API no le brinda al desarrollador acceso a información personal sobre ti.

3.6. ¿Qué es una llamada de API?

Una llamada de API, también conocida como solicitud de API, es una instancia en la que el propietario de un sitio web solicita el uso de la API de un desarrollador. Guardar la API, iniciar sesión en el sitio web del desarrollador y hacer consultas acerca de la aplicación son acciones que representan llamadas de API.

Al tener esto en cuenta, un límite de llamadas de API es el número de veces que puedes solicitar información sobre una API a un servicio web dentro de un periodo determinado. Te recomendamos leer los términos de uso de cualquier API que estés considerando usar. Estos documentos deberían detallar claramente las limitaciones y el uso adecuado del programa.

3.7. ¿Qué es un API gateway?

Un API gateway es la puerta de acceso que permite a un usuario acceder a la información que ha solicitado. Por ello, estas entradas condicionan el éxito en la extracción y llamado de una aplicación.

Estas puertas sirven para encapsular la información o el código y evitar que se mezcle con otras partes del programa. Además, protege su contenido, pues para acceder a través de estas puertas se puede solicitar una autorización, identificación o el cumplimiento de algunas reglas en el uso de la API.

3.8. Bibliografía

- https://blog.hubspot.es/website/que-como-usar-api
- https://www.suratica.es/que-es-una-api-key/
- https://academy.binance.com/es/articles/what-is-an-api-key-and-how-to-use-it-securely

4. ¿Cuáles son los tres verbos de API?

4.1. Introducción

Una API es un conjunto de reglas o protocolos que permite a las aplicaciones de software comunicarse entre ellas, para intercambiar datos, características y funcionalidad.

Un protocolo es aquel que especifica las reglas de la comunicación, en este caso, entre dos computadoras. El protocolo HTTP (Hyper Text Transfer Protocol) fue creado específicamente para la web. Una de las especificaciones de este protocolo son sus verbos, estos nos ayudan a indicar acciones.

Los tres verbos principales de una API son:
- GET. Lo utilizamos para solicitar datos o recursos específicos.
- POST. Envía datos a un recurso para su creación.
- DELETE. Elimina por completo un recurso.

4.2. Otros verbos API

- HEAD. Es similar a una petición GET pero sin contenido, sólo trae los encabezados. En ejemplo de su uso sería cuando vamos a utilizar APIs, para comprobar si lo que vamos a enviar es correcto y puede ser procesado.
- PUT. Es utilizado para actualizar un recurso.
- PATCH. Actualiza un sección especifica de un recurso.

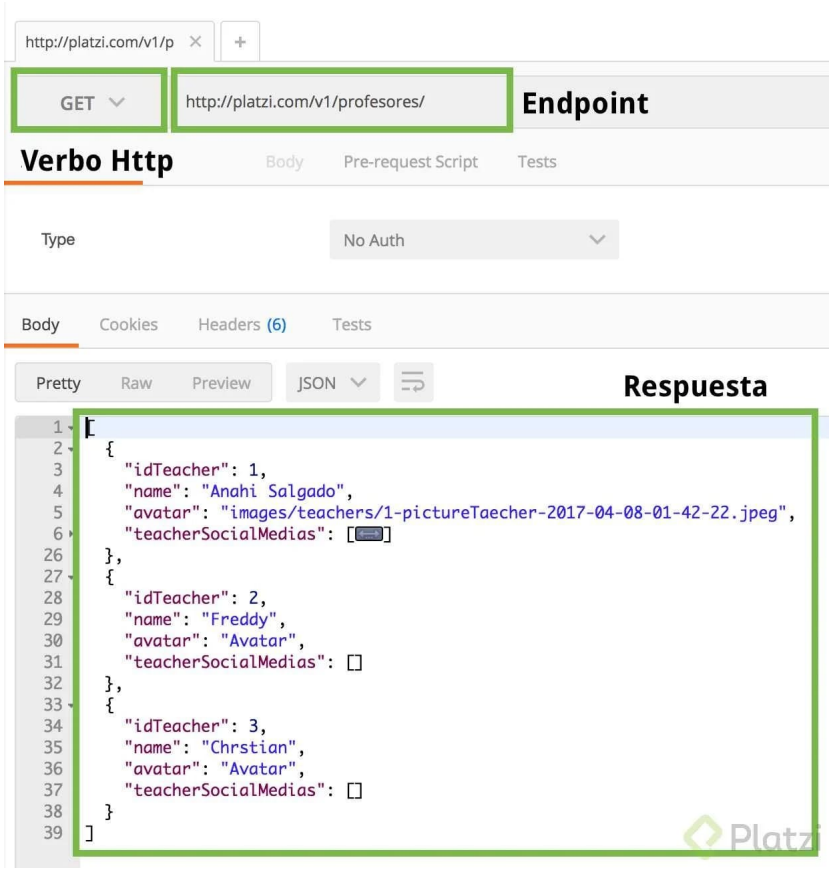
4.3. Ejemplo de solicitudes de verbos HTTP en Postman

Supongamos que necesitas ejecutar estas acciones con una URL base como https://platzi.com/profesores.

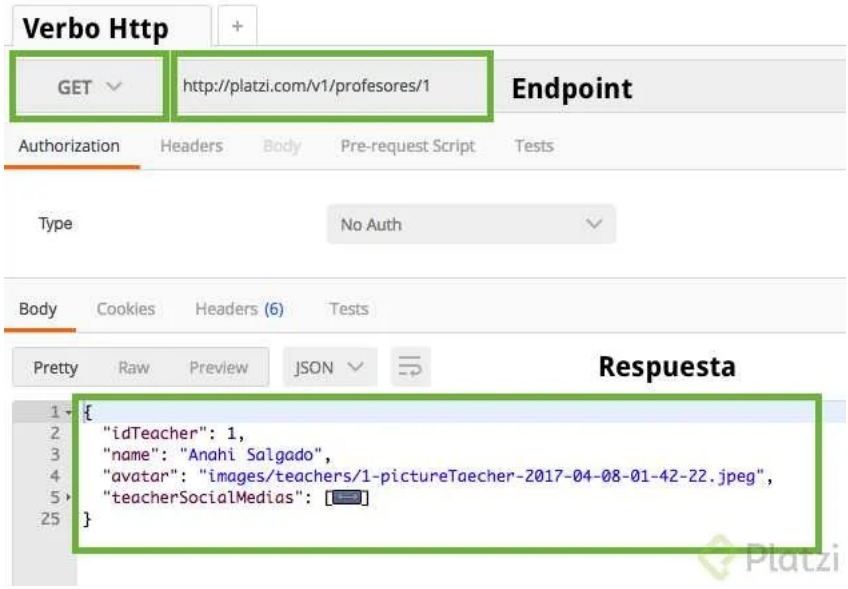
Para realizar este tipo de solicitudes, si es que queremos hacer pruebas, una de las herramientas que podemos utilizar es Postman, que es multiplataforma.

4.3.1. Método GET

Para el caso de GET, si queremos obtener todos los profesores, podríamos hacer algo así con Postman:

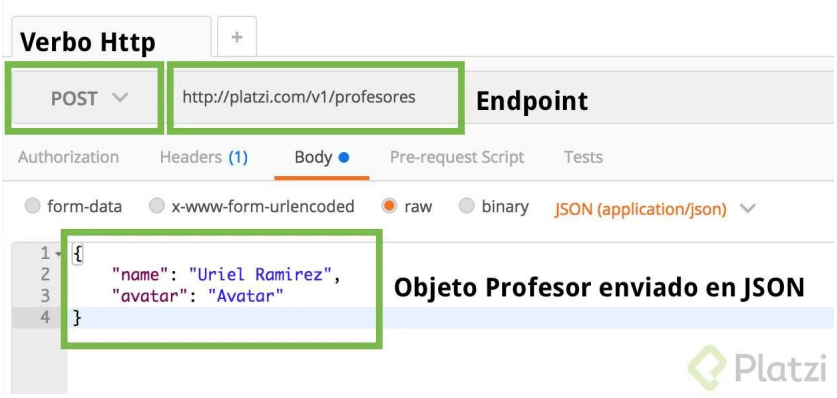


Otro caso GET donde queremos obtener los datos de un profesor en particular, en este caso el que tiene identificador 1, podríamos hacerlo así:



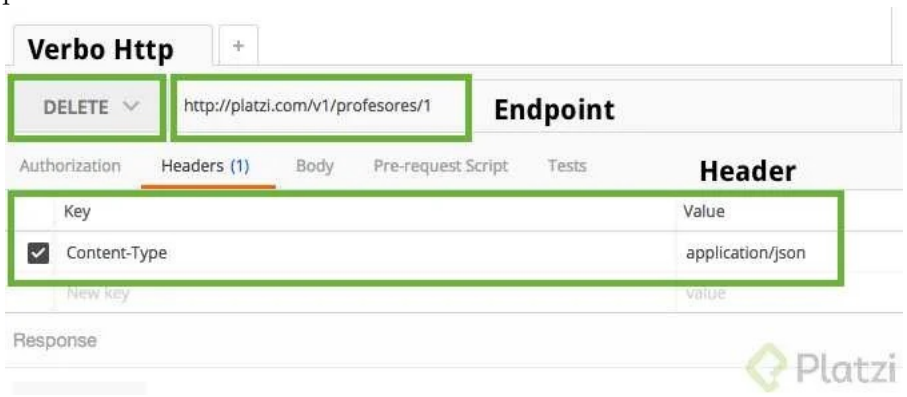
4.3.2. Método POST

Para el método POST, como haremos una inserción de datos tenemos que enviar el objeto Profesor con los datos clave, Postman tiene un campo llamado Body donde envías el objeto en forma de JSON algo así:



4.3.3. Método DELETE

El método DELETE solo necesita que coloquemos en la url el identificador que corresponde al profesor que queremos eliminar, todas las llamadas deben llevar el header application/json si es que así lo marca el API.



4.4. Bibliografía

- https://aws.amazon.com/es/what-is/api/
- https://platzi.com/clases/1912-intro-elasticsearch/28689-verbos-http/
- https://www.ibm.com/topics/api
- https://www.dotcom-monitor.com/wiki/es/knowledge-base/api-cartero/

5. ¿Es MongoDB una base de datos SQL o NoSQL?

5.1. Introducción

MongoDB es una base de datos NoSQL orientada a documentos, que apareció a mediados de la década de 2000. Se utiliza para almacenar volúmenes masivos de datos. Se diferencia de las bases de datos relacionales por su flexibilidad y rendimiento.

A diferencia de una base de datos relacional SQL tradicional, MongoDB no se basa en tablas y columnas. Los datos se almacenan como colecciones y documentos.

Los documentos son pares value/key que sirven como unidad básica de datos. Las colecciones contienen conjuntos de documentos y funciones. Son el equivalente a las tablas en las bases de datos relacionales clásicas.

Cada base de datos MongoDB contiene colecciones, que a su vez contienen documentos. Cada documento es diferente y puede tener un número variable de campos. El tamaño y el contenido de cada documento también varían. La estructura de un documento corresponde a la forma en que los desarrolladores construyen sus clases y objetos en el lenguaje de programación utilizado. En general, las clases no son filas y columnas, sino que tienen una estructura clara formada por pares Value/key.

Los documentos no tienen un esquema predefinido y los campos pueden añadirse a voluntad. El modelo de datos disponible en MongoDB facilita la representación de relaciones jerárquicas u otras estructuras complejas.

Otra característica importante de MongoDB es la elasticidad de sus entornos. Muchas empresas tienen clusters de más de 100 nodos para bases de datos que contienen millones de documentos.

5.2. La arquitectura de MongoDB y sus componentes

La arquitectura de MongoDB se basa en varios componentes principales. En primer lugar, el identificador es un campo obligatorio para cada documento. Representa un valor único y puede considerarse como la clave principal del documento para identificarlo dentro de la colección.

Un documento es el equivalente a un registro en una base de datos tradicional. Se compone de campos de nombre y valor. Cada campo es una asociación entre un nombre y un valor y es similar a una columna en una base de datos relacional.

Una colección es un grupo de documentos de MongoDB, y se corresponde con una tabla creada con cualquier otro sistema de gestión de bases relacionales, en inglés RDMS (Relational Database Management System) como Oracle o MS SQL en una base de datos relacional. No tiene una estructura predefinida.

Una base de datos es un contenedor de colecciones, al igual que un RDMS es un contenedor de tablas para las bases de datos relacionales. Cada uno tiene su propio conjunto de archivos en el sistema de archivos. Un servidor MongoDB puede almacenar múltiples bases de datos.

Por último, JSON (JavaScript Object Notation) es un formato de texto plano para expresar datos estructurados. Está soportado por muchos lenguajes de programación.

5.3. ¿Por qué utilizar MongoDB? ¿Cuáles son las ventajas?

MongoDB tiene varias ventajas importantes:

- En primer lugar, esta base de datos NoSQL orientada a documentos es muy flexible y se adapta a los casos de uso concretos de una empresa.

- Las consultas ad hoc permiten encontrar campos específicos dentro de los documentos. También es posible crear índices para mejorar el rendimiento de las búsquedas. Se puede indexar cualquier campo.

- Posibilidad de crear «conjuntos de réplicas» formados por dos o más instancias de MongoDB. Cada miembro puede actuar como réplica secundaria o primaria en cualquier momento.

La réplica primaria es el servidor principal, que interactúa con el cliente y realiza todas las operaciones de lectura y escritura. Las réplicas secundarias mantienen una copia de los datos. Así, en caso de fallo de la réplica primaria, el cambio a la secundaria se realiza automáticamente. Este sistema garantiza una alta disponibilidad.

- El concepto de **sharding** permite el escalado horizontal al distribuir los datos entre múltiples instancias de MongoDB. La base de datos puede ejecutarse en varios servidores, y esto permite equilibrar la carga o duplicar los datos para mantener el sistema en funcionamiento en caso de fallo del hardware.

Debido a estas numerosas ventajas, MongoDB es ahora una herramienta muy utilizada en el campo de la ingeniería de datos. Es una solución imprescindible para los ingenieros de datos.

5.4. SQL vs NoSQL : ¿cuáles son las diferencias y cuándo utilizar uno u otro?

5.4.1. Principales diferencias SQL y NoSQL

Las principales diferencias entre ambos modelos de bases de datos son:

- Las BBDD SQL almacenan datos de manera estructurada y las NoSQL lo hacen en su formato original.

- Las SQL proporcionan una capacidad de escalar baja, en comparación con las NoSQL. Esta es una de las principales ventajas de las NoSQL, ya que están pensadas para grandes volúmenes de información como el Big Data. Lo anterior es debido a que las SQL están centralizadas y las NoSQL distribuidas, posibilitando que se ejecuten en múltiples máquinas pero con muy pocos recursos (RAM, CPU, disco...).

- La adaptación a los cambios de las SQL es poca y puede ser compleja. Sin embargo, las NoSQL son totalmente flexibles.

- Las BBDD SQL están totalmente estandarizadas y las NoSQL carecen de homogeneización.

- Las SQL se utilizan en múltiples aplicaciones de todo tipo, las NoSQL se emplean principalmente para el Big Data (por ejemplo en redes sociales).

- Las BBDD SQL proporcionan consistencia en los datos (integridad). Sin embargo, las NoSQL, al buscar rapidez, no ponen el foco en esta característica.

- La rapidez de ambas BBDD va a depender del contexto o de su uso: en datos estructurados las SQL son más rápidas, pero como vimos anteriormente, el Big Data no es estructurado y es ahí donde consiguen mucha mayor rapidez las NoSQL.

- La integridad de los datos no es una restricción en MongoDB. Los datos no necesitan ser «normalizados» antes de su uso como en un RDBMS. Esto es una ventaja real, ya que la restricción de normalización puede degradar el rendimiento a medida que la base de datos crece.

5.4.2. Cuando utilizar uno u otro: datos estructurados vs. datos no estructurados

Los datos estructurados se almacenan en BBDD SQL mientras que los no estructurados en las NoSQL.

Los datos estructurados son más fáciles de analizar que los no estructurados, estos últimos necesitan de herramientas complejas de analítica.

Los datos no estructurados son más flexibles y los cambios de arquitectura no les afectan tanto como a los estructurados, que dependiendo de la naturaleza del cambio pueden necesitar migraciones complejas de un modelo de datos a otro.

Los no estructurados son almacenados en bruto, por lo que pueden ser accedidos por los usuarios para realizar configuraciones en ellos adaptadas a sus necesidades concretas.

Las bases de datos posibilitan la gestión ordenada de la información y disponer de un repositorio de datos para ser accedido y procesado por las principales aplicaciones de negocio de las organizaciones. Dependiendo del propósito de estas aplicaciones y del tipo de información a almacenar, utilizaremos BBDD SQL para las aplicaciones clásicas con datos estructurados y

NoSQL para aplicaciones que tienen que manejar volúmenes muy grandes de datos estructurados y no estructurados como las de Big Data.

5.5. Bibliografía

- https://datascientest.com/es/mongodb-todo-sobre-la-base-de-datos-nosql-orientada-a-documentos
- https://www.unir.net/ingenieria/revista/nosql-vs-sql/

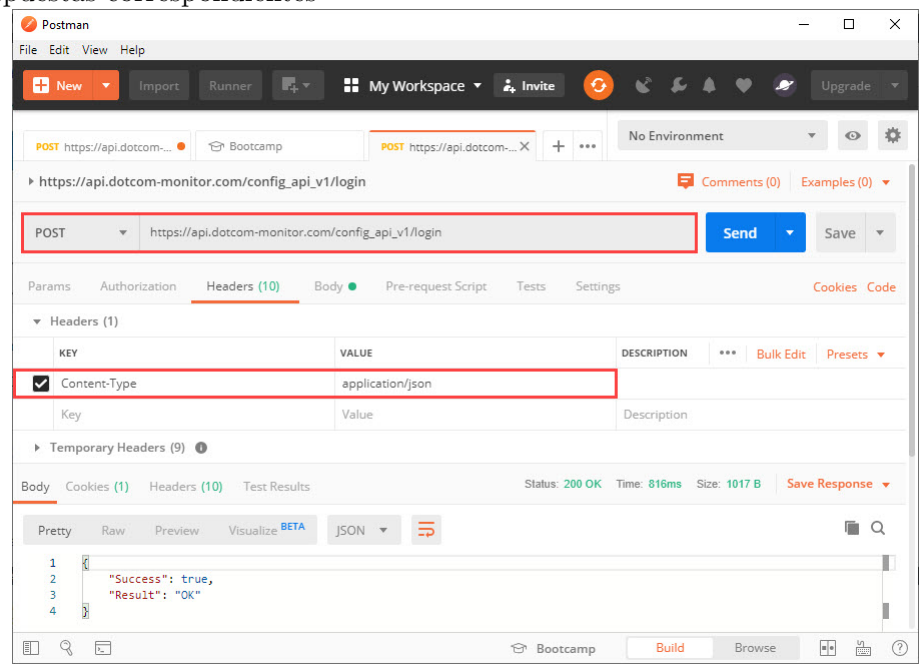
6. ¿Qué es Postman y para qué sirve?

6.1. Introducción

Postman es una plataforma que permite y hace más sencilla la creación y el uso de APIs, se puede gestionar diferentes entornos de desarrollo, organizar las solicitudes en colecciones y realizar pruebas automatizadas para verificar el comportamiento de los sistemas. Es gratuito, si trabajas solo, o de pago, si quieres trabajar de manera colaborativa.

Gracias a los avances en tecnología, Postman ha pasado de ser una extensión de Google Chrome a ser una aplicación que cuenta con herramientas nativas para varios sistemas operativos, entre los que se encuentran los principales: Windows, Mac y Linux.

Postman es una herramienta de colaboración y desarrollo que permite a los desarrolladores interactuar y probar el funcionamiento de servicios web y aplicaciones. proporcionando una interfaz gráfica intuitiva y fácil de usar para enviar solicitudes a servidores web y recibir las respuestas correspondientes



6.2. Principales características de Postman

Este entorno ofrece una GUI que facilita a los desarrolladores el envío de solicitudes HTTP y HTTPS a una API y a gestionar las respuestas recibidas.

Las principales características y funcionalidades de Postman son:

- **Envío de solicitudes**. Permite enviar solicitudes GET, POST, PUT, DELETE y otros métodos HTTP a una API especificando los parámetros, encabezados y cuerpo de la solicitud.

- **Gestión de entornos**. La configuración para diferentes entornos (por ejemplo, desarrollo, prueba, producción) y el cambio sencillo entre ellos (para realizar pruebas y desarrollo en diferentes contextos). Probar colecciones o catálogos APIs, ya sea para Frontend o Backend, porque da la posibilidad de hacer pruebas y comprobar el correcto funcionamiento de los proyectos.

- **Colecciones de solicitudes**. Agrupa las solicitudes relacionadas en colecciones, lo que facilita la organización y ejecución de pruebas automatizadas.

- **Pruebas automatizadas**. Es ideal para crear y ejecutar pruebas automatizadas para verificar el comportamiento de una API (detectar errores e incrementar la calidad del software).

- **Compartir información en un entorno cloud** con los miembros del equipo que forman parte del desarrollo.

- **Documentación de API**. Genera de forma automatizada, documentación detallada de la API a partir de las solicitudes y respuestas realizadas, lo que facilita su comprensión y uso por parte de otros desarrolladores.

6.3. ¿Cuáles son sus ventajas respecto a otras herramientas?

Cada vez son más los desarrolladores y programadores que apuestan por un entorno como Postman para automatizar pruebas y mejorar sus procesos de trabajo. Los principales beneficios que se obtienen con esta herramienta son:

- Facilidad a la hora de trabajar al disponer de una interfaz gráfica de usuario intuitiva, sencilla y personalizable.

- Amplia compatibilidad con numerosas tecnologías y protocolos web, como por ejemplo; HTTP, HTTPS, REST, SOAP, GraphQL... (lo que permite interaccionar con diversos tipos de API sin complicaciones o problemas).

- Ofrece una amplia gama de funcionalidades para diseñar, probar y documentar APIs, siendo probablemente la solución más completa del mercado para gestionar el ciclo de vida completo de desarrollo de APIs.

- Fomenta y facilita la colaboración entre los miembros del equipo de desarrollo (con opciones interesantes como compartir colecciones de solicitudes con otros desarrolladores).

- Cuenta con una comunidad amplia de usuarios que está en constante crecimiento y que aporta una gran cantidad de recursos, como tutoriales, documentación, foros y grupos de discusión...

- Se integra perfectamente con varias herramientas populares utilizadas en el desarrollo de software. Por ejemplo, se puede conectar con sistemas de control de versiones como GitHub, servicios de generación de documentación como Swagger o herramientas de automatización de pruebas como Jenkins, entre muchas otras.

- Permite a los usuarios agregar scripts personalizados utilizando JavaScript (para automatizar tareas repetitivas, configurar pruebas avanzadas o agregar validaciones personalizadas a las respuestas de la API).

- Las colecciones son una característica central de Postman que permite organizar y agrupar solicitudes relacionadas. Esto simplifica la administración de API complejas y facilita la reutilización de solicitudes y flujos de trabajo en diferentes proyectos.

6.4. ¿Por qué usar Postman?

Postman es una herramienta de prueba y desarrollo de API que está diseñada para enviar solicitudes desde el lado del cliente al servidor web y recibir una respuesta del back-end. La información recibida con la respuesta viene determinada por los datos enviados junto con la solicitud. Por lo tanto, Postman se utiliza como un cliente de API para comprobar las solicitudes cliente-servidor para asegurarse de que todo funciona en el lado del servidor como se espera. Postman admite solicitudes a los servicios web Restful, SOAP y GraphQL.

Una interfaz gráfica hace de Postman una herramienta fácil de usar en el proceso de prueba y desarrollo de API.

Postman puede descargarse desde https://www.postman.com/downloads/.

Para probar los servicios web restful, Postman utiliza solicitudes HTTP para enviar información a una API. Una solicitud HTTP es un mensaje HTTP que un cliente envía a un servidor HTTP. Por lo general, una solicitud HTTP contiene una línea de inicio, un conjunto de encabezados HTTP y un cuerpo.

Una línea de inicio de la solicitud HTTP contiene el método HTTP, el URI del recurso de destino y la versión del protocolo HTTP y tiene la siguiente estructura:

Método HTTP/ URI de destino/ Versión HTTP

Los métodos HTTP determinan la acción que se debe realizar en el recurso. El significado de los métodos HTTP se describe mediante la especificación del protocolo. La especificación del protocolo HTTP no limita el número de métodos diferentes que se pueden utilizar. Sin embargo, solo algunos de los métodos más estándar se utilizan para admitir la compatibilidad con una amplia gama de aplicaciones.

Algunos de los métodos HTTP que se pueden utilizar en las llamadas a la API son:

- GET: para obtener (leer) datos (por ejemplo, una lista de usuarios).
- POST :para crear nuevos datos.
- PUT / PATCH: para actualizar los datos.
- DELETE:para eliminar datos.
- OPTIONS: para obtener una descripción completa de los métodos de API disponibles en el servicio.

El encabezado contiene metadatos para permitir que un cliente pase información de clarificación y servicio sobre la solicitud HTTP, como información de codificación, parámetros de autorización, etc.

La información que desea transferir a través de la red se pasa en el cuerpo. El cuerpo es opcional y se puede dejar vacío (dependiendo de los métodos y encabezados HTTP).

La respuesta HTTP son los datos que regresan del servidor de API. Además de los datos del cuerpo, el encabezado de respuesta contiene un código HTTP de estado de la respuesta del servidor. Por ejemplo, puede recibir los siguientes códigos de estado en el encabezado de respuesta:

200 – Éxito
400 – Mala petición
401 – No autorizado

6.5. Trabajar con solicitudes en Postman

Con la interfaz gráfica Postman no es necesario que los desarrolladores web escriban su propio código para probar las características de la API.

Trabajar con solicitudes en Postman incluye la siguiente secuencia de pasos:

- Agregar una nueva solicitud HTTP mediante la interfaz Postman.
- Personalización de la solicitud (especificación de un método HTTP, encabezado, cuerpo, parámetros de autenticación).
- Ejecutando la solicitud.
- Guardar la solicitud (por ejemplo, en una carpeta o colección).
- Pruebas en postmano.

Para procesar la respuesta del servidor se pueden crear diferentes pruebas en Postman. La prueba en Postman es un código JavaScript que se ejecuta automáticamente después de que el cliente recibió una respuesta a la solicitud. En otras palabras, las pruebas de Postman se aplican al resultado de la solicitud ejecutada.

Postman ofrece muchos fragmentos de código listos para usar que puede agregar a la prueba. Aquí puede encontrar fragmentos de código para validar los códigos y el contenido de las respuestas, analizar y guardar valores en variables de entorno o variables globales, comprobar su cumplimiento con los valores especificados, etc. Por ejemplo, puede comprobar que el código de estado de la respuesta a la solicitud GET es 200. Las pruebas se pueden aplicar no solo a una sola solicitud, sino que se pueden mover a un nivel de colección.

6.6. Colecciones de carteros

Para ejecutar varias solicitudes una por una se utiliza automáticamente una colección de solicitudes en Postman. Puede ejecutar Colecciones rellenadas con solicitudes y pruebas con Collection Runner y usarlas como pruebas de API automatizadas. Para ejecutar una colección, puede seleccionar el entorno, el número de iteraciones en la ejecución y un retraso entre las solicitudes. Además, Postman admite el registro de solicitudes y el almacenamiento de variables y cookies.

Una vez que se ha creado una colección de solicitudes, se puede exportar a un archivo para su uso en una aplicación de terceros.

Tenga en cuenta que, en caso de que necesite llamar a una API que requiera autenticación, la recopilación de solicitudes a esta API debe incluir una solicitud POST al servicio de autenticación correspondiente para autorizar al cliente en el servidor.

Además de las características de Postman descritas, puede parametrizar sus solicitudes, agregar un punto de interrupción a las llamadas a la API y crear diferentes entornos para sus solicitudes.

6.7. Bibliografía

- https://formadoresit.es/que-es-postman-cuales-son-sus-principales-ventajas/
- https://assemblerinstitute.com/blog/que-es-postman/
- https://www.dotcom-monitor.com/wiki/es/knowledge-base/api-cartero/
- https://learning.postman.com/.

7. ¿Qué es el polimorfismo?

7.1. Introducción a la herencia y el polimorfismo en Python

La programación orientada a objetos nos permite crear clases que pueden heredar propiedades, métodos y comportamientos de otras clases ya existentes. En Python, la herencia es una característica clave que nos permite crear clases hijas a partir de una clase padre.

La herencia en Python se logra por medio de una sintaxis sencilla que involucra la creación de una nueva clase que hereda atributos y métodos de la clase padre. Para crear una clase hija en Python, simplemente agregamos el nombre de la clase padre en paréntesis después del nombre de la clase hija.

El polimorfismo, por otro lado, es una característica que nos permite utilizar objetos de diferentes clases de manera intercambiable. Esto significa que el mismo método o función puede ser utilizado en diferentes tipos de objetos, sin preocuparnos por conocer los detalles exactos de cada uno de ellos. En Python, el **polimorfismo** está estrechamente relacionado con la **herencia** y la **superposición de métodos**.

La superposición de métodos es una técnica que nos permite modificar el comportamiento de los métodos heredados de la clase padre en la clase hija. Esto se logra al definir un método con el mismo nombre en la clase hija como en la clase padre. Cuando se llama al método en la clase hija, el intérprete de Python buscará la definición del método en la clase hija primero y, si no la encuentra, lo buscará en la clase padre.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "woof"
```

En Python, también podemos acceder a los métodos heredados de la clase padre utilizando la función super(). Esto nos permite llamar al método de la clase padre directamente desde la clase hija, ahorrándonos tiempo y esfuerzo en la reescritura de código.

La herencia y el polimorfismo en Python nos permiten crear clases con una mayor flexibilidad y versatilidad. Esto nos permite reutilizar código y diseñar sistemas más eficientes y escalables. Si buscas mejorar tus habilidades de programación en Python, ¡no puedes dejar de aprender sobre la herencia y el polimorfismo!

7.2. Cómo utilizar la herencia para crear clases hijas con componentes similares pero diferentes funcionalidades

La herencia es una de las características más útiles de la programación orientada a objetos (POO) que nos permite crear clases hijas con componentes similares pero diferentes funcionalidades. En Python, esto se logra mediante la creación de una clase que hereda todas las propiedades y métodos de otra clase. Para hacer uso de esta, utilizamos la palabra clave 'super' para referirnos a la clase padre.

Por ejemplo, si tenemos una clase 'Animal' con algunas propiedades y métodos, podemos crear una clase 'Perro' que hereda estas propiedades y métodos. Además, podemos agregar propiedades y métodos adicionales específicos de la clase de 'Perro'.

```
class Animal:
```

```
    def __init(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def hacer_sonido(self):
        print("Este animal hace algun sonido")

class Perro(Animal):
    def __init__(self, nombre, edad, raza):
        super().__init__(nombre, edad)
        self.raza = raza

    def hacer_sonido(self):
        print("El perro hace guau guau")
```

En este ejemplo, la clase 'Perro' hereda las propiedades de la clase 'Animal' mediante la declaración de '(Animal)' en su definición. Además, sobre escribe el método 'hacer_sonido', para que este método ahora muestre "El perro hace guau guau" en lugar de "Este animal hace algún sonido".

Esto significa que, al crear una instancia de la clase 'Perro', podemos acceder a las propiedades y métodos de la clase 'Animal' y también a las nuevas propiedades y métodos de la clase 'Perro'.

```
mi_perro = Perro("Scottie", 3, "Terrier")
print(mi_perro.nombre) # Scottie
print(mi_perro.edad) # 3
print(mi_perro.raza) # Terrier
mi_perro.hacer_sonido() # El perro hace guau guau
```

La herencia también nos permitirá modificar o añadir nuevos métodos o atributos a la clase hija sin afectar la clase padre. En el ejemplo anterior, la clase 'Perro' sobrescribió el método 'hacer_sonido', pero también incluimos una propiedad de perro 'raza' que no está presente en la clase 'Animal'.

7.3. Cómo el polimorfismo permite a distintos objetos responder de manera diferente a un mismo llamado de método

El polimorfismo es una técnica de la programación orientada a objetos que permite a distintos objetos responder de manera diferente a un mismo llamado de método. En Python, esto se logra gracias al uso de clases y funciones, lo que aumenta significativamente la flexibilidad de nuestras implementaciones.

El término polimorfismo tiene origen en las palabras poly (muchos) y morfo (formas), y aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas. ¿Pero qué significa esto? Pues bien, significa que objetos de diferentes clases pueden ser accedidos utilizando el mismo interfaz, mostrando un comportamiento distinto (tomando diferentes formas) según cómo sean accedidos.

Una de las ventajas del polimorfismo es que nos permite escribir código más genérico, lo que a su vez nos permite reutilizar nuestro código en una variedad de situaciones.

Vamos a ver en detalle un ejemplo de polimorfismo:

```
class Animal:
    def __init__(self, name):
        self.name = name
```

```
4
5     def make_sound(self):
6         pass
7
8 class Dog(Animal):
9     def make_sound(self):
10        return "woof"
11
12 class Cat(Animal):
13    def make_sound(self):
14        return "meow"
15
16 class Cow(Animal):
17    def make_sound(self):
18        return "moo"
```

En este ejemplo pasa lo siguiente:
- Creamos, una superclase Animal y tres subclases Dog, Cat y Cow.
- Las tres subclases van a heredar de la clase base.
- Cada una de ellas tiene su propio método make_sound() la cual genera como mensaje el sonido respectivo del animal.

Vamos a comprobar si nuestro código funciona:
- Creamos una lista de objetos (nuestros animales) a las cuales les pondremos sus respectivos nombres

```
1 animals = [Dog("Gala"), Cat("Manet"), Cow("Betsy")]
```

- Como lo guardamos en una lista vamos a realizar un ciclo que nos permita recorrer y llamar al método make_sound() de cada una de las subclases.

```
1 animals = [Dog("Gala"), Cat("Manet"), Cow("Betsy")]
2
3 for animal in animals:
4     print(animal.name + " dice " + animal.make_sound())
```

Si todo salió bien debemos obtener el siguiente resultado

```
1 Gala dice woof
2 Manet dice meow
3 Betsy dice moo
```

Eso es todo, ¡este es el poder del polimorfismo!. Permite que diferentes objetos se comporten de manera similar, básicamente que adquieran "diferentes formas", lo que hace que nuestro código sea más flexible y fácil de mantener.

Vamos a ver otro caso de polimorfismo pero aplicado de dos formas diferentes: en el primer ejemplo utilizamos el **polimorfismo de clase**, y en el segundo ejemplo, utilizamos el **polimorfismo de clase de herencia**.

7.3.1. Ejemplo 1: Polimorfismo de clase

Son tres clases independientes, pero que comparten el mismo método

```
1 class Car:
2     def __init_(self, brand, model):
3         self.brand = brand
```

```
        self.model = model

    def move(self):
        print("Drive")

class Boat:
    def __init_(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Sail")

class Plane:
    def __init_(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Fly")

car = Car("Ford", "Mustang")
boat = Boat("Ibiza", "Touring 20")
plane = Plane("Boeing", "747")

for x in (car, boat, plane)
     print(x.brand)
     print(x.model)
     print(x.move())
```

7.3.2. Ejemplo 2: Polimorfismo de clase de herencia

Veamos el mismo ejemplo anterior, pero esta vez con tres subclases que heredan de la clase Vehicle, todas ellas con el método move().

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
```

```
    print("Fly!")

car = Car("Ford", "Mustang") #Crear un objeto de Car
boat = Boat("Ibiza", "Touring 20") #Crear un objeto de Boat
plane = Plane("Boeing", "747") #Crear un objeto de Plane

for x in (car, boat, plane):
  print(x.brand)
  print(x.model)
  print(x.move())
```

7.4. Ejercicios de polimorfismo

7.4.1. Ejercicio 1

Supongamos que tenemos una clase Figura, que tiene un método abstracto area(). La idea detrás de esta clase es que cualquier figura que queramos modelar, sea un cuadrado, un círculo, un triángulo, etc., siempre tendrá una propiedad de área. Entonces, podemos crear una clase Cuadrado que herede de Figura y defina su propia implementación de area(), que calcularía el área del cuadrado. Lo mismo podemos hacer para otras figuras, como un Círculo o un Triángulo.

7.4.2. Ejercicio 2

Crear una clase llamada Marino, con un método que sea hablar, en donde muestre un mensaje que diga «Hola, soy un animal marino!». Luego, crear una clase Pulpo que herede de Marino, pero modificar el mensaje de hablar por «Hola soy un Pulpo!». Por último, crear una clase Foca, heredada de Marino, pero que tenga un atributo nuevo llamado 'mensaje' y que muestre ese mesaje como parámetro.

7.5. Soluciones

7.5.1. Solución ejercicio 1

```
import math

class Figura:
    def area(self):
        pass

class Cuadrado(Figura):
    def __init__(self, lado):
        self.lado = lado

    def area(self):
        return self.lado * self.lado

class Circulo(Figura):
    def __init__(self, radio):
        self.radio = radio

    def area(self):
        return math.pi * self.radio * self.radio
```

```
20
21class Triangulo(Figura):
22    def __init__(self, base, altura):
23        self.base = base
24        self.altura = altura
25
26    def area(self):
27        return self.base * self.altura / 2
28
29def calcular_area(figura):
30    return figura.area()
31
32cuadrado = Cuadrado(5)
33circulo = Circulo(3)
34triangulo = Triangulo(4, 5)
35
36print(calcular_area(cuadrado)) # 25
37print(calcular_area(circulo)) # 28.27
38print(calcular_area(triangulo)) # 10
```

7.5.2. Solución ejercicio 2

```
1class Marino():
2     def hablar(self);
3        print("Hola, soy un animal marino")
4
5class Pulpo(Marino):
6    def hablar(self);
7        print("Hola, soy un pulpo")
8
9class Foca(Marino):
10    def hablar(self, mensaje);
11        self.mensaje = mensaje
12        print(mensaje)
13
14marino = Marino()
15marino.hablar()
16
17pulpo = Pulpo()
18pulpo.hablar()
19
20foca = Foca()
21foca.hablar("Hola, soy una foca")
```

7.6. Bibliografía

- https://ellibrodepython.com/polimorfismo-en-programacion
- https://apuntes.de/python/herencia-y-polimorfismo-en-python-amplia-la-flexibilidad-de-tus-clases/

8. ¿Qué es un método dunder?

8.1. Introducción

"Los métodos dunder, también conocidos como métodos mágicos, son métodos especiales que se utilizan para emular el comportamiento de las funciones integradas. Estos métodos tienen un significado particular para el intérprete de Python. Sus nombres empiezan y terminan en __ (doble guión bajo). Por ejemplo init. Como son parte de la clase de objeto, toman como primer argumento self. Pueden tener argumentos adicionales dependiendo de cómo los llamará el intérprete. También se definen claramente con dos guiones bajos antes y después de sus nombres.

```
class Person:
    def __init__(self):
```

Normalmente estos métodos no son invocados directamente por el programador. Más bien, el intérprete los llamará mientras el programa se está ejecutando. Por ejemplo cuando haces una simple suma 2 + 2 se está invocando al método __add__ internamente.

Los métodos dunder son un concepto útil en la Programación Orientada a Objetos en Python. Usándolos, se especifica el comportamiento de sus tipos de datos personalizados cuando se usan con operaciones integradas comunes. Estas operaciones incluyen:

- Operaciones aritméticas
- Operaciones de comparación
- Operaciones de ciclo de vida

8.2. Métodos dunder comunes

8.2.1. Método constructor

Se trata de un inicializador de clase o también conocido como constructor. Cuando una instancia de una clase es creada, el método __init__ es llamado. Por ejemplo:

```
class GetTest(object):

    def __init__(self):
        print("Saludos!!")

    def another_method(self):
        print("Soy otro metodo que no es llamado ↩
           automaticamente")

a = GetTest()     # Saludos!!

a.another_method()     # Soy otro metodo que no es llamado ↩
    automaticamente
```

Puedes ver como __init__ es llamado inmediatamente después de que la instancia haya sido creada. También puedes pasar argumentos en la inicialización, como se muestra a continuación.

```
class GetTest(object):

    def __init__(self, name):
        print('Saludos!! {0}'.format(name))

    def another_method(self):
```

```
        print('Soy otro metodo que no es llamado ↩
            automaticamente')

a = GetTest('Pelayo')     # Salida: Saludos!! Pelayo

# Si intentas crear el objeto sin ningun argumento, da error.
b = GetTest()



Traceback (most recent call last):
TypeError: __init__() takes exactly 2 arguments (1 given)
```

8.2.2. Método __getitem__

Implementar el método __getitem__ en una clase permite a la instancia usar [] para indexar sus elementos. Veamos un ejemplo:

```
class GetTest(object):

    def __init__(self):
        self.info = {
            "name" : "Covadonga",
            "country" : "Asturias",
            "number" : 12345812
         }

    def __getitem__(self, i):
        return self.info[i]

foo = GetTest()

foo["name"]     # "Covadonga"

foo['number"]     # 12345812
```

Sin implementar el método __getitem__ tendríamos un error si intentamos hacerlo:

```
>>> foo['name']

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'GetTest' object has no attribute '__getitem__'
```

8.2.3. Operaciones aritmeticas

- __add__ obj +
- __sub__ obj -
- __mul__ obj *
- __floordiv__ obj //
- __truediv__ obj /
- __mod__ obj %
- __pow__ obj **

8.2.4. Operaciones de comparación

- __lt__ a < b
- __gt__ a > b
- __le__ a <= b
- __ge__ a >= b
- __ne__ a != b
- __eq__ a == b

8.2.5. Operaciones de representación

- **Método __str__** También conocido como "pretty print object". Su objetivo es proporcionar una cadena que sea comprensible para el usuario final de la aplicación. Este método se llama, por ejemplo, cuando se llama al str() función, pasando una instancia de la clase como argumento. También se llama cuando se pasa en la instancia a las funciones print() y format().

- **Método __repr__** Devuelve una representación de cadena del objeto utilizado por los desarrolladores. La cadena devuelta debe ser rica en información de modo que pueda construir una instancia idéntica del objeto solo a partir de la cadena.

8.3. Mejores prácticas para crear métodos dunder

Los métodos dunder son increíbles y simplificarán su código. Sin embargo, es importante tener en cuenta lo siguiente cuando los use.

- Úselos con moderación: implementar demasiados métodos dunder en sus clases hace que su código sea difícil de entender. Trata de implementar sólo los esenciales.

- Documente el comportamiento de sus métodos mágicos para que otros desarrolladores puedan saber exactamente lo que hacen. Esto les facilita su uso y depuración cuando sea necesario.

8.4. Ejercicio

8.4.1. Ejercicio 1

Definir una clase llamada Punto con dos atributos x e y. Crearle el método especial __str__ para retornar un string con el formato (x,y).

8.4.2. Ejercicio 2

Declarar una clase llamada Familia. Definir como atributos el nombre del padre, madre y una lista con los nombres de los hijos. Definir el método especial __str__ que retorne un string con el nombre del padre, la madre y de todos sus hijos. Pueden tener hijos o no.

8.5. Soluciones

8.5.1. Solución ejercicio 1

```
1class Punto:
2
3    def __init__(self, x, y):
4        self.x=x
5        self.y=y
6
7    def __str__(self):
8        return "("+str(self.x)+","+str(self.y)+")"
9
```

```
10
11# bloque principal
12
13punto1=Punto(10,3)
14punto2=Punto(3,4)
15print(punto1)     # (10,3)
16print(punto2)     # (3,4)
```

8.5.2. Solución ejercicio 2

```
1class Familia:
2
3    def __init__(self, padre, madre, hijos=[]):
4        self.padre=padre
5        self.madre=madre
6        self.hijos=hijos
7
8    def __str__(self):
9        cadena=self.padre+","+self.madre
10       for hi in self.hijos:
11           cadena=cadena+","+hi
12       return cadena
13
14
15# bloque principal
16
17familia1=Familia("Pablo","Ana",["Pepe","Julio"])
18familia2=Familia("Jorge","Carla")
19familia3=Familia("Luis","Maria",["Marcos"])
20
21print(familia1)     # Pablo, Ana, Pepe, Julio
22print(familia2)     # Jorge, Carla
23print(familia3)     # Luis, Maria, Marcos
```

8.6. Bibliografía

- https://python-intermedio.readthedocs.io/es/latest/classes.html
- https://www.netinetidesign.com/post/metodos-especiales-en-python/
- https://www.tutorialesprogramacionya.com/pythonya

9. ¿Qué es un decorador de python?

9.1. Introducción

A través de los decoradores seremos capaces reducir las líneas de código duplicadas, haremos que nuestro código sea legible, fácil de testear, fácil de mantener y sobre todo, tendremos un código mucho más Pythonico. Vamos a ver cómo podemos crearlos, implementarlos y sobre todo, sacarles el máximo provecho posible.

Antes de entrar de lleno con el tema de decoradores es importante mencionar que en Python las funciones son ciudadanos de primera clase, eso quiere decir que una función puede ser asignada a una variable, puede ser utilizada como argumento para otra función, o inclusive puede ser retornada. Veamos un ejemplo.

```
def saludar():
    print('Hola soy una funcion')

def super_funcion(funcion):
    funcion()

funcion = saludar  # Asignamos la funcion a una variable!

super_funcion(una_funcion)      # Hola soy una funcion
```

Un decorador no es más que una función la cual toma como input una función y a su vez retorna otra función. Puede sonar algo confuso ¿no? Lo que nos debe quedar claro es que al momento de implementar un decorador estaremos trabajando, con por lo menos, 3 funciones: el input, el output y la función principal. Para que nos quede más en claro a mi me gusta nombrar a las funciones como: a, b y c.

Donde a recibe como parámetro b para dar como salida a c. Esta es una pequeña "formula"la cual me gusta mucho mencionar.

a(b) -> c

Veamos un ejemplo muy sencillo. Tenemos una función suma() que vamos a decorar usando mi_decorador(). Para ello, antes de la declaración de la función suma, hacemos uso de @mi_decorador.

```
def mi_decorador(funcion):
    def nueva_funcion(a, b):
        print("Se va a llamar")
        c = funcion(a, b)
        print("Se ha llamado")
        return c
    return nueva_funcion

@mi_decorador
def suma(a, b):
    print("Entra en funcion suma")
    return a + b

suma(5,8)

# Se va a llamar
# Entra en funcion suma
# Se ha llamado
```

Lo que realiza mi_decorador() es definir una nueva función que encapsula o envuelve la función que se pasa como entrada. Concretamente lo que hace es hace uso de dos print(), uno antes y otro después de la llamada la función.

Por lo tanto, cualquier función que use @mi_decorador tendrá dos print, uno y al principio y otro al final, dando igual lo que realmente haga la función.

Veamos otro ejemplo usando el decorador sobre otra función.

```
@mi_decorador
def resta(a ,b):
    print("Entra en funcion resta")
    return a - b

resta(5, 3)

# Se va a llamar
# Entra en funcion resta
# Se ha llamado
```

9.2. Decoradores con parámetros

Tal vez quieras que tu decorador tenga algún parámetro de entrada para modificar su comportamiento. Se puede hacer envolviendo una vez más la función como se muestra a continuación.

```
def mi_decorador(arg):
    def decorador_real(funcion):
        def nueva_funcion(a, b):
            print(arg)
            c = funcion(a, b)
            print(arg)
            return c
        return nueva_funcion
    return decorador_real

@mi_decorador("Imprimer esto antes y despues")
def suma(a, b):
    print("Entra en funcion suma")
    return a + b

suma(5,8)
# Imprimer esto antes y despues
# Entra en funcion suma
# Imprimer esto antes y despues
```

9.3. Ejemplos prácticos

9.3.1. Ejemplo 1: Logger

Una de las utilidades más usadas de los decoradores son los logger. Su uso nos permite escribir en un fichero los resultados de ciertas operaciones, que funciones han sido llamadas, o cualquier información que en un futuro resulte útil para ver que ha pasado.

En el siguiente ejemplo tenemos un uso más completo del decorador log() que escribe en un fichero los resultados de las funciones que han sido llamadas.

```
def log(fichero_log):
    def decorador_log(func):
        def decorador_funcion(*args, **kwargs):
            with open(fichero_log, 'a') as opened_file:
                output = func(*args, **kwargs)
                opened_file.write(f"{output}\n")
        return decorador_funcion
    return decorador_log

@log('ficherosalida.log')
def suma(a, b):
    return a + b

@log('ficherosalida.log')
def resta(a, b):
    return a - b

@log('ficherosalida.log')
def multiplicadivide(a, b, c):
    return a*b/c

suma(10, 30)
resta(7, 23)
multiplicadivide(5, 10, 2)
```

Nótese que el decorador puede ser usado sobre funciones que tienen diferente número de parámetros de entrada, y su funcionalidad será siempre la misma. Escribir en el fichero pasado como parámetro los resultados de las operaciones.

9.3.2. Ejemplo 2: Uso autorizado

Otro caso de uso muy importante y ampliamente usado en Flask, que es un framework de desarrollo web, es el uso de decoradores para asegurarse de que una función es llamada cuando el usuario se ha autenticado.

Realizando alguna simplificación, podríamos tener un decorador que requiriera que una variable autenticado fuera True. La función se ejecutará solo si dicha variable global es verdadera, y se dará un error de lo contrario.

```
autenticado = True

def requiere_autenticacion(f):
    def funcion_decorada(*args, **kwargs):
        if not autenticado:
            print("Error. El usuario no se ha autenticado")
        else:
            return f(*args, **kwargs)
    return funcion_decorada

@requiere_autenticacion
def di_hola():
    print("Hola")

```

```
15 di_hola()
```

9.4. Bibliografía

- https://codigofacilito.com/articulos/decoradores-python
- https://ellibrodepython.com/decoradores-python