Estructuras de datos y funciones

Seguridad en las API

{desafío} latam_

Utilizar estructuras de datos apropiadas para la elaboración de un algoritmo que resuelve un problema acorde al lenguaje Python.

Codificar un programa utilizando funciones para la reutilización de código acorde al lenguaje Python.

- Unidad 1:
  Introducción a Python
- Unidad 2:
  Sentencias condicionales e iterativas
- Unidad 3:
  Estructuras de datos y funciones

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Aplica los conceptos básicos del lenguaje Python a través de la utilización de la información extraída mediante API para agregar versatilidad y robustez a sus aplicaciones.*

{desafío} latam_

¿Qué son las API REST?

{desafío}
latam_

/* API REST */

Recursos

- Son elementos que se pueden obtener y mostrar, y además, en algunos casos, crear nuevos, actualizarlos y borrarlos.

Ejemplo: En esta API tenemos los recursos "posts" (que son los que hemos estado utilizando hasta ahora), "comentarios", "álbumes", "fotos", "listas de tareas (todos)" y "usuarios". Además hay relaciones entre estos recursos. Para ver qué acciones se pueden hacer en específico sobre estos recursos, se necesita conocer las rutas (o direcciones).

/posts	100 posts
/comments	500 comments
/albums	100 albums
/photos	5000 photos
/todos	200 todos
/users	10 users

Rutas

- Las palabras que aparecen antes de cada ruta se conocen como **métodos HTTP o verbos**, y se debe usar el verbo especificado para cada **ruta**.
- Hasta el momento hemos presentado sólo el método GET que permitirá extraer datos, pero tenemos también los métodos POST, PUT y DELETE que deberán utilizarse según corresponda.
- Por ahora seguiremos ocupando el verbo GET, pero haremos un request a otro recurso y obtendremos las fotos.

All HTTP methods are supported.

GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
GET	/posts?userId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Note: you can view detailed examples here.

```
data = request_get('https://jsonplaceholder.typicode.com/photos')[0:10]
# Tomamos solo 10
print(data)
```

{desafío} latam_

La estructura de la respuesta es la misma: una lista que contiene diccionarios donde cada uno representa una foto.

El proceso de obtener resultados también es el mismo; iteramos la lista y, de cada diccionario, obtenemos la información que necesitamos.

Creando un recurso

- Hay situaciones en las que en vez de extraer datos queremos agregar información. Para ello será necesario utilizar el verbo o método HTTP apropiado.
  En este caso particular presentaremos el método POST.

- De acuerdo a la documentación, podríamos agregar un post utilizando la terminación **/post** en nuestra URL.
  Ya sabemos que cada post tiene una estructura definida que contiene un **userId**, un **body** y un **title**.

Routes

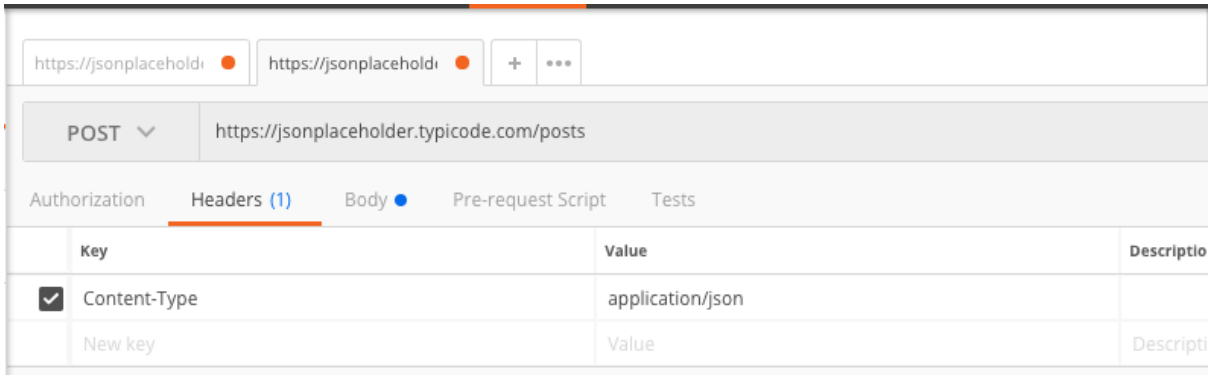
All HTTP methods are supported.

GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
GET	/posts?userId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Note: you can view detailed examples here.

Creando un recurso desde Postman

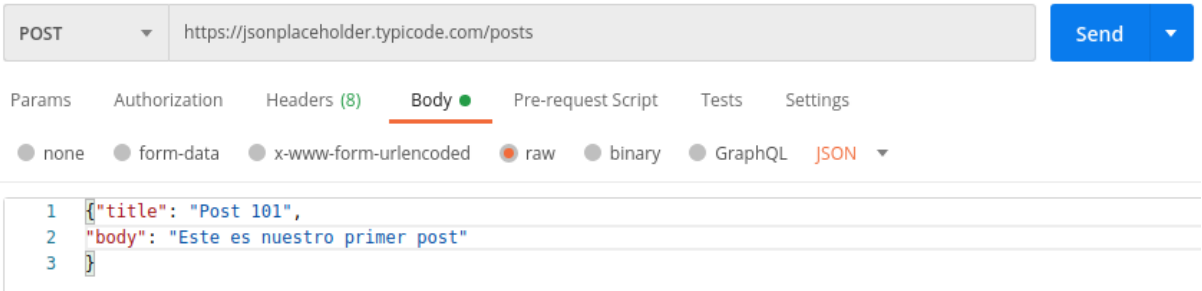
Para subir un artículo nuevo necesitamos seleccionar el verbo POST, e ingresar la URL de la documentación.



{desafío} latam_

Agregando el contenido

Dentro de la API y del ejemplo vimos que un artículo se compone de un título **title**, de un cuerpo **body** y de un dueño **userId**.
Como primera prueba, subiremos un artículo con título. Para esto iremos al **tab** de **body**, seleccionaremos la opción **raw** y a la derecha marcaremos la opción JSON.



Para ingresar información, ésta debe seguir el formato JSON.

Es importante destacar que a diferencia de los diccionarios en Python, los strings deben incluirse utilizando doble comilla (").

Agregando el contenido

Al hacer click en Send y realizar el request obtendremos el siguiente resultado:

```
{
          "title": "Post 101",
          "body": "Este es nuestro primer post",
      "id": 101
}
```

En este caso nuestro resultado tendrá respuesta 201, el cual significa que el recurso fue creado de manera exitosa y que tiene los elementos que se mencionan.

/* Es sumamente importante recordar que esta API es de prueba,
por lo que los resultados del POST no se persisten (no guarda cambios).
En una API real, si es que nosotros realizamos un método POST,
los resultados estarán disponibles para el futuro. */

/* No siempre como usuario tendrás acceso a utilizar métodos distintos al GET,
normalmente métodos como POST, que alteran la información guardada en la
API, estará restringida sólo para usuarios que tienen la necesidad
y los permisos para hacerlo. */

Creando un recurso desde Python

```
import requests
url = "https://jsonplaceholder.typicode.com/posts"
payload="{\n        \"title\": \"Post 101\",\n    \"body\": \"Este es nuestro primer
post\"\n}"
headers = {
  'Content-Type': 'application/json'
}
response = requests.request("POST", url, headers=headers, data=payload)
print(response.text)
```

Si bien el código es bastante similar al obtenido cuando se quería utilizar el método GET es posible notar algunas diferencias:

- payload y headers ahora no están vacíos y juegan un rol en el llamado.
- payload contendrá el objeto JSON que queremos incluir en la API, mientras que headers contendrá el formato con el que estamos ingresando los datos, que en este caso corresponde a un application/json. Además el método request utiliza el verbo POST.

Creando un recurso desde Python

Tal como se hizo en el caso del método GET, este código puede ser reescrito de la siguiente manera:

```
payload = '''{"title": "Post 101",
          "body": "Este es nuestro primer post"}'''
response = requests.post('https://jsonplaceholder.typicode.com/posts',
                         data = payload)
print(response)
print(response.text)
```

```
<Response [201]>
{
  "id": 101
}
```

Actualizando un recurso

- Los métodos para actualizar un recurso en una API REST son PUT o PATCH.
- Si bien existe una diferencia entre estos, la mayoría de las API no hace distinción.
- Existen muchas APIs que para actualizar un recurso además aceptan el verbo POST. Incluso, algunas solo aceptan POST y no PUT.

Es importante siempre referirse a la documentación ya que estos nos indicarán las variaciones que puede tener el URL para cada método. En este caso es bueno notar que PUT o PATCH pueden modificar un POST en específico refiriéndose al id.

Actualizando un recurso desde Postman

La documentación utiliza la ruta **/posts/1** donde el número no tiene que ser necesariamente 1, si no que es identificador del recurso que queremos actualizar. Para saber cuáles son, tenemos que listar los recursos.

En este caso, los identificadores van desde 1 hasta 100, por lo tanto, si queremos cambiar el artículo con **id 20** tendríamos que hacer un request con método put a https://jsonplaceholder.typicode.com/posts/20.

Además, dentro del body de la request debemos agregar los nuevos valores para el artículo.

Actualizando un recurso desde Python

Si copiamos el código proveniente desde el mismo POSTMAN tenemos lo siguiente:

```
import requests
url = "https://jsonplaceholder.typicode.com/posts/20"
payload="{\n  \"title\": \"Cambio de Post\",\n        \"body\": \"Actualizando el Post\",\n    \"userId\": 1\n}"
headers = {
  'Content-Type': 'application/json'
}
response = requests.request("PUT", url, headers=headers, data=payload)
print(response.text)
```

Como se puede ver, las únicas diferencias significativas con el método POST es que se indicará que el método es PUT y, además, se debe utilizar el URL apropiado según la documentación.

Actualizando un recurso desde Python

Escribiendo esto de manera más limpia tenemos lo siguiente:

```
payload = '''{"title": "Cambio de Post",
           "body": "Actualizando el Post",
           "userId": 1}'''
response = requests.put('https://jsonplaceholder.typicode.com/posts/20',
                        data = payload)
print(response)
print(response.text)
```

Al ejecutar este código se obtiene el código 200 de éxito y además el id del post que fue modificado.

```
<Response [200]>
{
  "id": 20
}
```

Ejercicio guiado

"Crear un sitio web simple"

Crear un sitio web simple

Tomaremos datos de la siguiente API: https://jsonplaceholder.typicode.com/photos

Paso 1

Revisemos el contenido de nuestra API.

```
import requests
import json
def request_get(url):
    return json.loads(requests.get(url).text)

out = request_get('https://jsonplaceholder.typicode.com/photos')
len(response)
```

```
5000
```

Crear un sitio web simple

Tomaremos datos de la siguiente API: https://jsonplaceholder.typicode.com/photos

Paso 2

Notamos entonces que el resultado de nuestro llamado nos entrega 5000 fotos.

Para evitar que nuestro computador colapse utilizaremos sólo las primeras 5 fotos:

```
out = request_get('https://jsonplaceholder.typicode.com/photos')[:5]
print(len(out))
```

```
0
```

Crear un sitio web simple

Tomaremos datos de la siguiente API: https://jsonplaceholder.typicode.com/photos

Paso 3

Revisemos uno de los elementos extraídos para chequear su contenido.

```
print(out[0].keys())
```

```
dict_keys(['albumId', 'id', 'title', 'url', 'thumbnailUrl'])
```

Crear un sitio web simple

Tomaremos datos de la siguiente API: https://jsonplaceholder.typicode.com/photos

Paso 4

Como se puede ver tenemos varias claves. Si miramos que contiene cada una.

```
print(out[0])
```

```
{'albumId': 1, 'id': 1, 'title': 'accusamus beatae ad facilis cum
similique qui sunt', 'url': 'https://via.placeholder.com/600/92c952',
'thumbnailUrl': 'https://via.placeholder.com/150/92c952'}
```

De acá es posible apreciar que la clave url contiene un link de una foto.

Crear un sitio web simple

Tomaremos datos de la siguiente API: https://jsonplaceholder.typicode.com/photos

Paso 5

Esta url podría utilizarse para construir el sitio web. Una manera de construirlo es de manera manual. Pero quizás una idea un poco más estructurada podría ser armar un template o plantilla que nos permita rellenar los datos necesarios para crear de una manera más automatizada nuestro archivo html.

Supongamos que queremos crear un HTML de la siguiente manera:

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la Página</title>
</head>
<body>
<h1>Nuestra página Web</h1>
<img src="https://via.placeholder.com/600/92c952">
<img src="https://via.placeholder.com/600/92c952">
<img src="https://via.placeholder.com/600/92c952">
<img src="https://via.placeholder.com/600/92c952">
<img src="https://via.placeholder.com/600/92c952">
</body>
</html>
```

Crear un sitio web simple

Tomaremos datos de la siguiente API: https://jsonplaceholder.typicode.com/photos

Como notamos cada uno de los elementos dentro de src podría ser cada foto rescatada de la API. Este tipo de problema puede ser fácilmente resuelto por medio de f-string. El problema que tiene esto es que fuerza a definir el string cada vez que se quiere utilizar y no permite su reutilización. En este tipo de casos, Python posee una herramienta más potente en la librería string.

```
from string import Template
img_template = Template('<img src="$url">')
```

Esto nos permite crear un template en el cual tendremos una variable llamada url la cual puede ser rellenada con lo que necesitemos.

```
imagen = img_template.substitute(url = 'hola')
print(imagen)
```

```
'<img src="hola">'
```

{desafío} latam_

Crear un sitio web simple

Tomaremos datos de la siguiente API: https://jsonplaceholder.typicode.com/photos

Podríamos tratar de refactorizar nuestro HTML de tal manera que sea muy flexible:

```
html_template = Template('''<!DOCTYPE html>
<html>
<head>
<title>Título de la Página</title>
</head>
<body>
<h1>Nuestra página Web</h1>
$body
</body>
</html>
''')
```

Crear un sitio web simple

Tomaremos datos de la siguiente API: https://jsonplaceholder.typicode.com/photos

Es a partir de este sistema de Templates que podemos generalizar para nuestro problema:

```
lista_url = [elemento['url'] for elemento in out]
texto_img = ''
for url in lista_url:
            texto_img += img_template.substitute(url = url)+'\n'
print(texto_img)
```

```
<img src="https://via.placeholder.com/600/92c952">
<img src="https://via.placeholder.com/600/771796">
<img src="https://via.placeholder.com/600/24f355">
<img src="https://via.placeholder.com/600/d32776">
<img src="https://via.placeholder.com/600/f66b97">
```

/* Seguridad en las API */

Encriptación por SSL

SSL (**S**ecure **S**ockets **L**ayer): capa de seguridad que establece la encriptación entre el navegador y el servidor, y evita que la información que enviamos o recibimos sea leída por un tercero. Esto sucede, conectándonos a una página web por el navegador o a una API.

Existen 2 claves para el funcionamiento de la encriptación:

- una para cifrar
- otra para descifrar.

Ventajas de SSL

- Cifra el mensaje impidiendo que terceros puedan leerlo.
- Asegura que el emisor es quien dice ser, porque si alguien más cifró el mensaje con una llave destinada, el mensaje no tendrá sentido al descifrarlo.

Encriptación por SSL

Ejemplo

Supongamos la comunicación es entre dos personas, Alicia y Rob. Para cifrar el mensaje, Alicia tiene una clave y para descifrarlo, otra. Previo a comunicarse, Alicia se junta con Rob y le traspasa la clave para descifrar.
Esto permite que Alicia cifre su mensaje con su clave, le envía el mensaje a Rob, y este la descifra con la clave que la pasó previamente Alicia.

Esto tiene dos ventajas:

1. El mensaje pasa cifrado por lo que nadie más puede leerlo a menos que tenga la clave para descifrarlo.
2. La llave para descifrar solo sirve para los mensajes de Alice, por lo que sabemos que el mensaje viene realmente de Alice y no de otra persona.

Encriptación por SSL

Al momento de conectarnos a un sitio web que utilice SSL con un navegador, se establece un acuerdo de forma automática, que es llamado en inglés **handshake**. En este handshake, el servidor envía un certificado que tiene el nombre del sitio y la clave pública al cliente. *Si nos conectamos con el navegador y hacemos click en el candado de la conexión segura, podremos ver el certificado*.

- Dentro del proceso, cliente y servidor crearon una clave secreta especial utilizando los números que se enviaron durante el handshake.
- Durante el resto del proceso, se ocupa un sistema de cifrado simétrico utilizando esta única clave que nunca fue transmitida, pero que ambos servidores tienen.

Seguridad

Conectándose a SSL

Postman identifica automáticamente si un request ocupa HTTPS y genera el código para conectarnos. La librería requests, por defecto, tiene la verificación de certificados SSL habilitada, por lo que si no puede verificar un certificado durante un request, ocurrirá un **SSLError**.

Autenticación TOKEN

Muchas APIs requieren autenticación para poder acceder a sus servicios. Se debe conseguir una clave para conectarse.

Las mismas APIs disponen de servicio de registro, y al completarlo entregan un id. Esta clave recibe el nombre de TOKEN, el cual debe ser incluido dentro del llamado según las instrucciones entregadas en la documentación.

Ejercicio guiado

"Oxford Dictionary"

Oxford Dictionary

Investigar una API nueva en la cual necesitemos de una autenticación para poder acceder a ella → extraer la definición de la palabra "hello" según Oxford.

Paso 1

Primero que todo debemos acceder a la página del diccionario developer.oxforddictionaries y llenaremos el formulario de registro. También necesitaremos confirmar nuestro email.

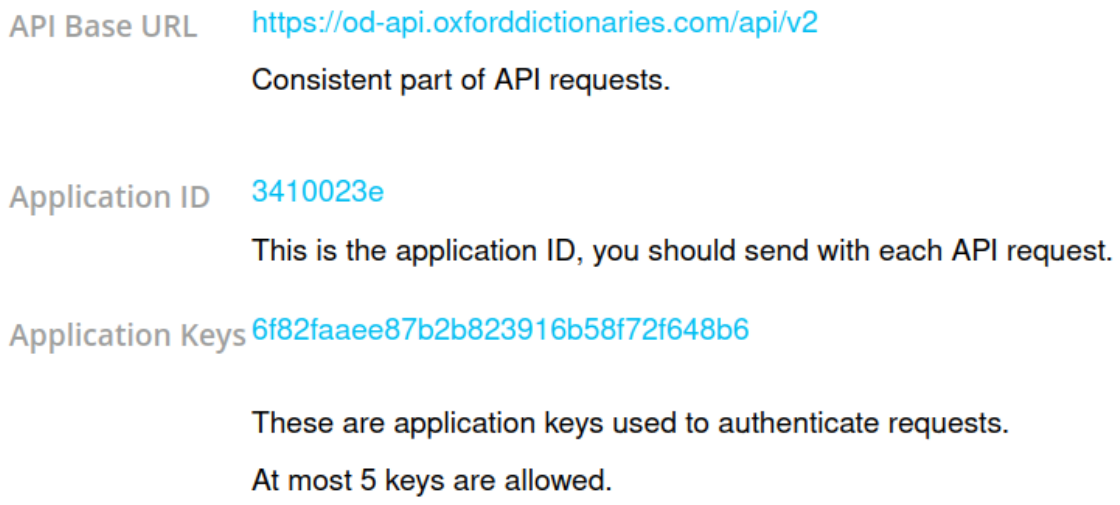
Paso 2

Al cumplir con el proceso de autenticación, se pedirá confirmar mediante email.
Si se introducen las credenciales al momento del registro se habilitará una opción para poder conseguir las API Keys.

Oxford Dictionary

Investigar una API nueva en la cual necesitemos de una autenticación para poder acceder a ella → extraer la definición de la palabra "hello" según Oxford.

En el nombre aparecerá el nombre de una APP por defecto que utilizará parte de los nombres que utilizaste para la suscripción.
Al ingresar ahí se llegará finalmente a las credenciales:



{desafío} latam_

Oxford Dictionary

Investigar una API nueva en la cual necesitemos de una autenticación para poder acceder a ella → extraer la definición de la palabra "hello" según Oxford.

Paso 3

Como se puede ver en este caso, la API tiene **dos claves distintas** que serán utilizadas para conectarse. Para entender el funcionamiento de ambas, es bueno referirse a la documentación entregada por la misma página:

```
3
4  import requests
5  import json
6
7  # TODO: replace with your own app_id and app_key
8  app_id = '<my app_id>'
9  app_key = '<my app_key>'
10 language = 'en-gb'
11 word_id = 'Ace'
12
13 url = 'https://od-api.oxforddictionaries.com/api/v2/entries/' + language + '/' + word_id.lower()
14 r = requests.get(url, headers = {'app_id' : app_id, 'app_key' : app_key})
```

Oxford Dictionary

Investigar una API nueva en la cual necesitemos de una autenticación para poder acceder a ella → extraer la definición de la palabra "hello" según Oxford.

Paso 4

Utilizando como referencia el código sugerido en la documentación, trataremos de hacer nuestro propio llamado a la API para buscar el significado de la palabra **"hello"**.

```
import json
import requests
app_id = "3410023e"
app_key = "6f82faaee87b2b823916b58f72f648b6"
word = "hello"
url = f"https://od-api.oxforddictionaries.com/api/v2/entries/en/{word}"
r = requests.get(url, headers = {"app_id": app_id, "app_key": app_key})
r = json.loads(r.text)
print(r)
```

Oxford Dictionary

Investigar una API nueva en la cual necesitemos de una autenticación para poder acceder a ella → extraer la definición de la palabra "hello" según Oxford.

Paso 5

Acá mostraremos un flujo exploratorio que puede ser bastante útil para diccionarios de alta complejidad:

```
print(r.keys())
```

.keys() nos indicará las claves asociadas a nuestra respuesta.
Estas pueden ser revisadas una a una.

Al analizar una a una las claves es claro que la que nos interesa es **'results'**. Es importante resaltar que la estructura resultante es una lista.

Esto es posible chequearse usando type() o sencillamente reconociendo que el resultado está entre [].

Oxford Dictionary

Investigar una API nueva en la cual necesitemos de una autenticación para poder acceder a ella → extraer la definición de la palabra "hello" según Oxford.

Paso 6

Siguiendo la misma lógica, es decir, buscando las claves más representativas, llegamos al siguiente diccionario.

```
definiciones = resultado['results'][0]['lexicalEntries'][0]['entries'][0]['senses'][0]
print(definiciones)
```

Como se puede observar, el diccionario final tiene las claves que estamos buscando.

Oxford Dictionary

Investigar una API nueva en la cual necesitemos de una autenticación para poder acceder a ella → extraer la definición de la palabra "hello" según Oxford.

Definición formal

```
print(definiciones['definitions'])
```

Definición corta

```
print(definiciones['shortDefinitions'])
```

Varias sub-definiciones

```
print([elemento['definitions'] for elemento in definiciones['subsenses']])
```

Otros tipos de autenticación

En este caso nos tocó ingresar la autenticación en el header. Esta es una de las opciones más frecuentes, pero algunas APIs tienen diferentes modelos de autenticación.

- En algunos casos nos tocará ingresar la clave en el body.
- En algunos casos, el token de autenticación cambiará después de cada request y tendremos que usar el último valor para rescatar el siguiente.

También existen otros modelos más complejos como el de Facebook y el de Twitter, que al día de hoy ocupan **OAuth2**.

Precauciones sobre los token

Muchos servicios tienen **límites de consumo** y otros manejan información delicada.

Los tokens, contraseñas y otros datos delicados entregados no deben ser subidos a **repositorios públicos** o ser **compartidos con terceros**.

Cierre del módulo

- Un algoritmo es una serie de pasos finitos para resolver un problema.
- Crear el algoritmo es la parte más compleja de la programación.
- Para crear algoritmos disponemos de herramientas como los diagramas de flujo y pseudocódigo.
- Las funciones son importantes para ordenar el código y evitar repetir varias veces lo mismo.
- Los ciclos son clave para evitar repetir código y hacer el programa escalable.
- Un problema puede tener más de una solución, y cuál podría ser la mejor dependerá de cada caso.
- Las estructuras de datos como listas y diccionarios nos permiten almacenar y manejar grandes cantidades de datos en una sola variable.
- Es importante siempre mantener las buenas prácticas de la programación, tales como la refactorización y la modularización. Mantener un código ordenado y reutilizable nos ahorra tiempo de desarrollo y sobre todo de debugging (búsqueda de errores).
- Podemos comunicarnos con otros programas de forma segura mediante el uso de una API, y usar estos datos para nuestros propios programas.

Próxima sesión...

- *Desafío guiado.*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales