



# Modularidad

Sistemas independientes

01 {

[Modularidad]

- Modularidad
- Funciones
- Modulos

}

# Modularidad {

En Python, la modularidad es la **capacidad de dividir un programa grande en módulos más pequeños**. Esto hace que el código sea más fácil de leer, entender y mantener.

Los módulos también pueden ser reutilizados en diferentes programas, lo que ahorra tiempo y esfuerzo.

Para utilizar la modularidad en Python, **se crean módulos que contienen el código que se desea dividir**.

Estos módulos **se pueden importar en otros programas** utilizando la palabra clave **import**. Una vez importados, el código del módulo puede ser utilizado en el programa principal.

}

# Modularidad {

Hay varios beneficios de utilizar la modularidad en Python:

- El código es **más fácil de leer y entender**, ya que está dividido en unidades más pequeñas.
- El código es **más fácil de mantener**, ya que las diferentes partes del código pueden ser modificadas o mejoradas sin afectar a las demás.
- El código es **más reutilizable**, ya que los módulos pueden ser utilizados en diferentes programas.
- El código es **más portable**, ya que los módulos pueden ser utilizados en diferentes plataformas.

}

# Modularidad {

La modularidad es la técnica de **construir un sistema a partir de subsistemas**.

**Cada subsistema se conoce como módulo**, y es una agrupación de funcionalidades y elementos interrelacionados, que forman un sistema relativamente independiente.

}

# Modularidad {

El **objetivo** de la modularidad es que **las piezas sean intercambiables**, es decir que distintas partes se puedan acoplar y desacoplar, sin alterar elementos no relacionados con dicha acción.

Se puede ver al **módulo** como una caja negra, con una interfaz que recibe **datos de entrada**, y una interfaz que da **datos de salida**.



# Funciones {

Ya hemos visto algo de modularidad, **las funciones encapsulan una parte del sistema**, y son capaces de recibir datos de entrada y de enviar datos de salida.

Una función en Python es un **bloque de código que se puede ejecutar varias veces**.

Las funciones se utilizan para organizar el código y hacerlo más fácil de leer y entender. Las funciones también se pueden utilizar para reutilizar el código y hacer que los programas sean más cortos y compactos.

}

# Creación Funciones {

Para crear una función se debe declarar con la **palabra reservada def**, seguido por el **nombre**, paréntesis, dentro de los cuales tendremos argumentos (si son necesarios) y finalizando con dos puntos:



```
def nombre_de_la_funcion(argumentos):  
    # cuerpo_de_la_funcion
```

}



# Creación Funciones {

Los **argumentos** son los **valores que se pasan** a la función.

El cuerpo de la funcion es el **código que se ejecuta cuando se llama** a la función.



```
def nombre_de_la_funcion(argumentos):  
    # cuerpo_de_la_funcion
```

}

# Creación Funciones {



```
def saludar():  
    print("Hola soy una funcion!!")  
  
saludar()
```

}

# Funciones. Parámetros {

Las funciones **pueden o no recibir parámetros**, que es un valor que la función espera recibir cuando sea llamada

**Dentro de los paréntesis**, se le indica a la función cuáles parámetros tiene, aquí se crean unas variables que son internas a la función.


Los parámetros se utilizan para **proporcionar datos a la función** y para comunicar los resultados de la función al código que la llama.

}

# Funciones. Parámetros {

Los parámetros pueden ser de **cualquier tipo de dato**, incluyendo números, cadenas, listas, objetos y tuplas.

También pueden ser **opcionales**, lo que significa que no es necesario pasarles un valor cuando se llama a la función.



```
def sumar(numero1, numero2):  
    resultado = numero1 + numero2  
    print(resultado)  
  
sumar(1, 2)
```

# Funciones. Return {

En Python, la palabra clave **return** se utiliza para **devolver un valor de una función**.

El valor devuelto puede ser **cualquier tipo de datos**, incluyendo números, cadenas, listas, objetos y tuplas.

La palabra clave **return** sin ningún argumento devuelve el valor **None**.

}

# Funciones. Return {

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14

}



```
def sumar2(numero1, numero2):  
    return numero1 + numero2
```

```
resultado = sumar2(1, 2)  
print(resultado)
```

# Funciones {

Los **nombres** que un dato tiene dentro (parametros) y fuera (argumentos) de la función **pueden ser diferentes.**



```
def restar (numero1, numero2):  
    return numero1 - numero2
```

```
a = int(input("Ingrese un numero: "))  
b = int(input("Ingrese un numero: "))  
resultado = restar(a, b)  
print(resultado)
```

}

# Paso por referencia o por valor {

En las funciones, existen dos paradigmas para el manejo de parámetros:

- **Pase por valor:** Se genera una “copia” de la variable, y cualquier cambio que se realice dentro de la función a esta variable, no afecta a la original.
- **Paso por referencia:** Se le da a la función acceso total a la variable dada como parámetros.

}



# Paso por valor {

Si se pasa una variable con un **objeto inmutable**, entonces el paso de parámetro será por **valor**.

- **Objetos inmutables:**

- Numeros(int, float, bool, etc).
- Strings.
- Booleans.
- Otros.

}

# Paso por referencia {

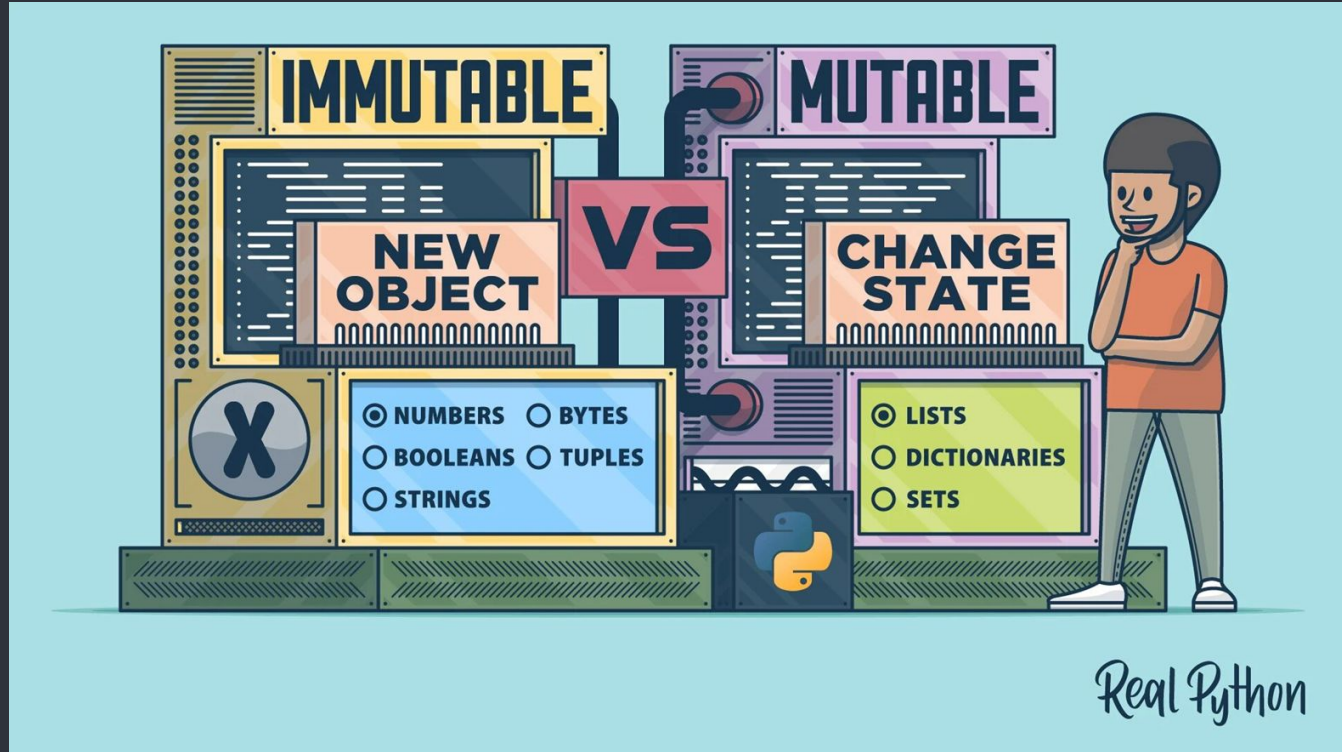
Si se pasa una variable con un **objeto mutable**, entonces el paso de parámetro será por **referencia**.

- **Objetos mutables:**

- Listas.
- Diccionarios.

}

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14



# Docstring {

Un docstring en Python es una cadena de texto que se utiliza para **documentar** una función, clase, método o módulo.

El docstring se escribe en la primera línea de la definición del objeto y se utiliza para proporcionar información sobre el propósito del objeto, sus parámetros, su valor de retorno y cualquier otra información relevante.

}

# Docstring {



```
def sumar(numero1, numero2):  
    """  
    Sumar dos numeros.  
  
    Args:  
        numero1 (int): El primer numero.  
        numero2 (int): El segundo numero.  
  
    Returns:  
        int: La suma de numero1 y numero2.  
    """  
    return numero1 + numero2
```

}

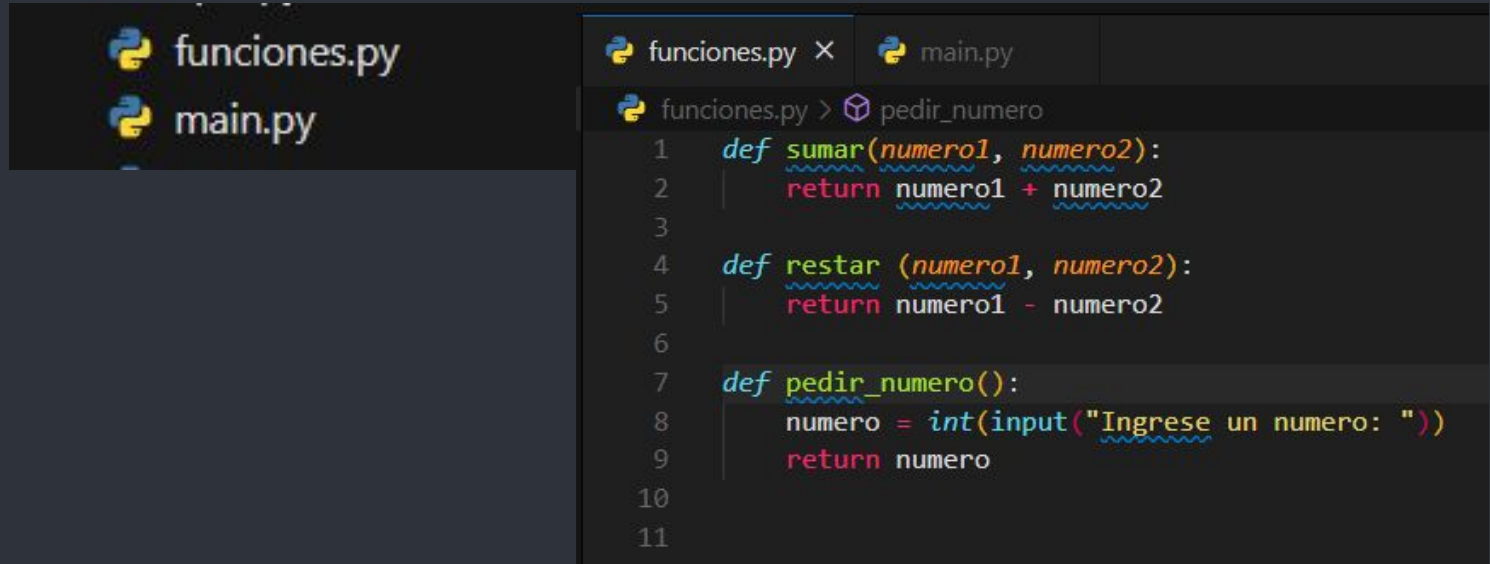
# Modulos {

Para **crear un módulo en Python**, se crea un archivo de texto con la extensión **.py**.

El código del módulo se escribe en el archivo de texto. Una vez creado el módulo, se puede importar en otros programas utilizando la palabra clave **import**.

}





# Modulos {



```
funciones.py
main.py

funciones.py > pedir_numero
1 def sumar(numero1, numero2):
2     return numero1 + numero2
3
4 def restar (numero1, numero2):
5     return numero1 - numero2
6
7 def pedir_numero():
8     numero = int(input("Ingrese un numero: "))
9     return numero
10
11
```

# Modulos {

 funciones.py main.py funciones.py main.py X main.py > ...

```
1 import funciones
```

```
2
```

```
3 numero1 = funciones.pedir_numero()
```

```
4 numero2 = funciones.pedir_numero()
```

```
5 resultado = funciones.sumar(numero1, numero2)
```

```
6 print(resultado)
```

```
7
```

}



# Alias {

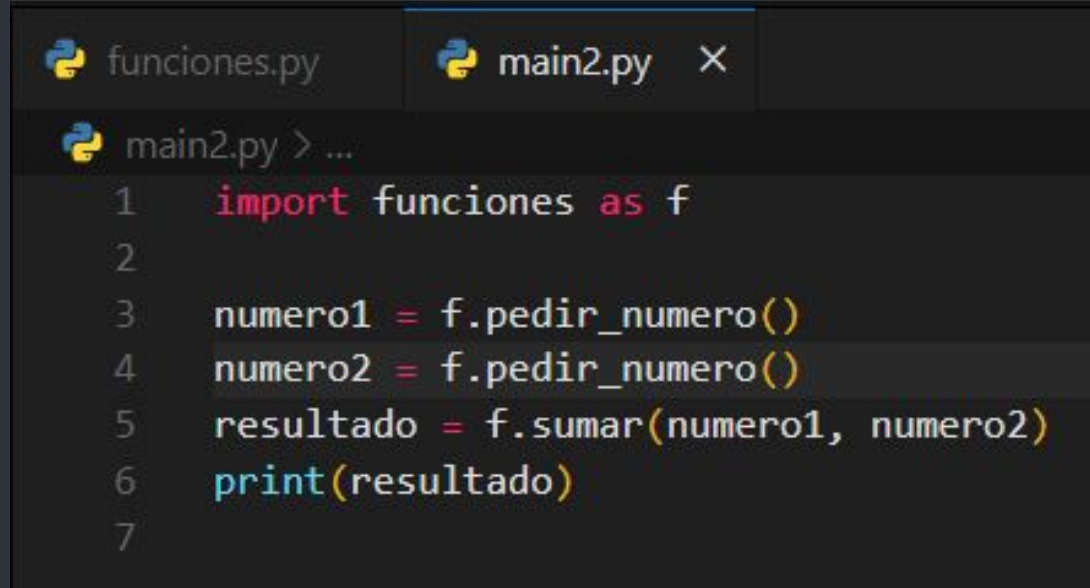
Podemos usar **alias** de los archivos para ahorrarnos escribir todo su nombre usando la palabra clave “as” (como) seguido del nombre que deseamos usar.



```
import nombre_archivo as alias
```

}

# Alias {



```
funciones.py  main2.py X

main2.py > ...
1  import funciones as f
2
3  numero1 = f.pedir_numero()
4  numero2 = f.pedir_numero()
5  resultado = f.sumar(numero1, numero2)
6  print(resultado)
7
```

}

# Paquetes {

En Python, un paquete es una **carpeta que contiene módulos**. Los paquetes se utilizan para organizar el código y hacerlo más fácil de leer y entender. Los paquetes también se pueden utilizar para reutilizar el código y hacer que los programas sean más cortos y compactos.

Para crear un paquete en Python, se crea una **carpeta**.

Los módulos del paquete se crean en archivos de texto con la extensión **.py** dentro de la carpeta del paquete. Una vez creado el paquete, se puede importar en otros programas utilizando la palabra clave **import**.

}

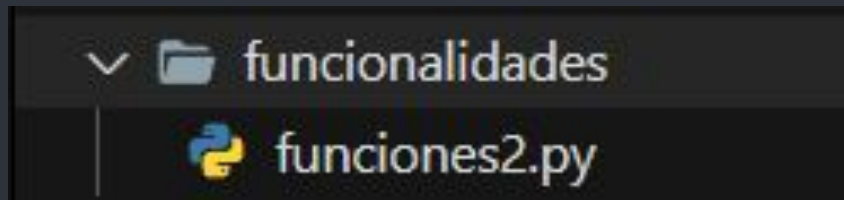
# Paquetes {



```
import nombre_paquete.nombre_modulo
```

}

# Paquetes {



```
import funcionalidades.funciones2 as f2  
  
numero = f2.pedir_numero()
```

}

# 1 Librerías internas {

2  
3  
4  
5 En Python, una biblioteca interna es una  
6 biblioteca que está **incluida** en el intérprete de  
7 Python.

8 Las bibliotecas internas se pueden usar sin  
9 tener que instalarlas por **separado**.  
10  
11

12  
13 }  
14

# 1 Librerías internas {

2  
3  
4 Algunas de las bibliotecas internas más populares  
5 incluyen:

- 6 - **math**: una biblioteca para el cálculo matemático.
- 7
- 8 - **random**: una biblioteca para generar números aleatorios.
- 9
- 10 - **string**: una biblioteca para trabajar con cadenas.
- 11
- 12 - **time**: una biblioteca para trabajar con el tiempo
- 13 - **sys**: una biblioteca para interactuar con el sistema
- 14 operativo.

}

# Librerías internas {



```
import math
```

```
factorial = math.factorial(6)  
print(factorial)
```

```
}
```



# Librerías externas {

En Python, una biblioteca externa es una biblioteca que **no está incluida** en el intérprete de Python.

Las bibliotecas externas se pueden instalar usando el **administrador de paquetes de Python: pip.**

Una vez instalada la biblioteca, se puede importar en un programa de Python utilizando la palabra clave **import.**

}

# Librerías externas {

Hay muchas bibliotecas externas disponibles para Python.  
Algunas de las bibliotecas más populares incluyen:

- **NumPy:** Una biblioteca para el trabajo con arrays y matrices.
- **Matplotlib:** Una biblioteca para la creación de gráficos y diagramas.
- **Pandas:** Una biblioteca para el trabajo con datos.

}

## Ejercicio 18 {

Realizar una función que reciba como parámetro un string que llamaremos “output”.

La función debe imprimir el valor de “output” utilizando un print

}

## Ejercicio 19 {

Realizar una función que reciba por parámetro un número, y calcule el factorial de dicho número. Finalmente, la función debe imprimir el valor del factorial.

}

## Ejercicio 20 {

Utilizar la función del ejercicio anterior que calcula el factorial de un número.

En lugar de imprimir el resultado, la función deberá retornar el valor calculado utilizando **return**.

}



# APIs

Comunicación

02 {

[APIs]

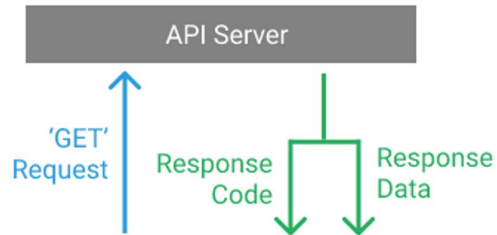
- APIs
- RESTFUL API
- Métodos HTTP
- CRUD
- Request get()
- Metodo .json()
- JSON
- Parametros

}

# API {

Una **Interfaz de Programación de Aplicaciones (API)** es un servidor web que se puede utilizar para **enviar y recibir data** en nuestro programa, generalmente se utiliza para recibir datos que nuestra aplicación va a utilizar.

Las APIs se utilizan para una variedad de propósitos, incluyendo el acceso a datos, el control de dispositivos y la comunicación entre aplicaciones.





# RESTful API {

**REST es un estilo arquitectónico** que define cómo los componentes de una aplicación se comunican entre sí utilizando el **protocolo HTTP**.

El **protocolo HTTP** (Hypertext Transfer Protocol) es el protocolo que **se utiliza para transferir información entre un cliente y un servidor web**.

HTTP es un protocolo basado en texto que utiliza mensajes para **comunicar las solicitudes y las respuestas** entre el cliente y el servidor.

}

# RESTful API {

Una **API RESTful** se basa en la idea de **recursos**. Un recurso es cualquier entidad que el servicio web expone, como un usuario, un producto o una orden. **Cada recurso tiene una URL única que se utiliza para acceder a él.**

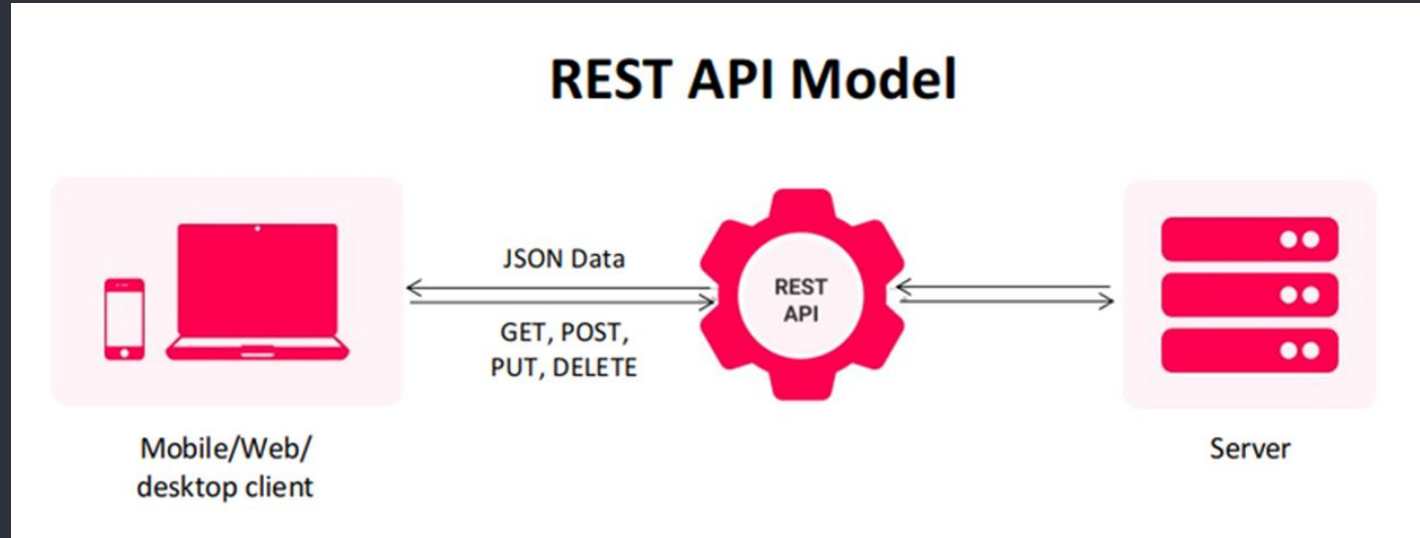
Una API RESTful utiliza los **métodos HTTP GET, POST, PUT y DELETE** para interactuar con los recursos.

}

api.py

try\_except.py

# RESTful API {



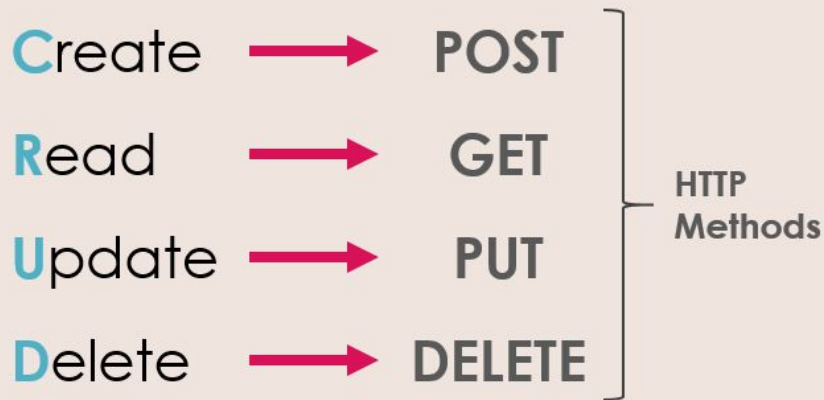
APIs

# Métodos HTTP {

- El **método GET** se utiliza para **obtener datos de un recurso**.
- El **método POST** se utiliza para **crear un nuevo recurso**.
- El **método PUT** se utiliza para **actualizar un recurso**.
- El **método DELETE** se utiliza para **eliminar un recurso**.

}

# Métodos HTTP {

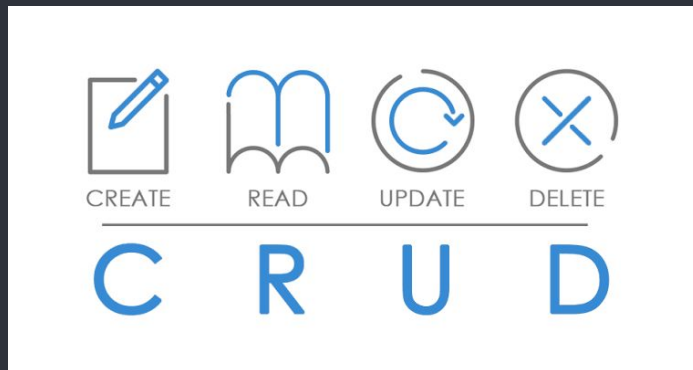


}

# CRUD {

CRUD es un acrónimo de **Create, Read, Update, Delete**, que en español significa **Crear, Leer, Actualizar y Eliminar**.

Es un conjunto de operaciones básicas para **interactuar con los datos** en una base de datos.



# Request get() {

Para pedir datos en Python, se puede utilizar la **librería requests**. La librería requests proporciona una interfaz simple para hacer **solicitudes HTTP a las APIs**.

Para hacer una solicitud a una API, se utiliza el **método requests.get()**. El método **requests.get()** toma la **URL de la API** como argumento y devuelve un objeto **Response**.

El objeto **Response** **contiene los datos** de la respuesta de la API.

}

# Pip {

Pip es un **administrador de paquetes** para Python. Permite a los usuarios **instalar y administrar paquetes** de software escritos en Python.

Para **instalar un paquete** de software con Pip, se utiliza el comando **pip install**.

Por ejemplo, para instalar la **librería numpy**, se utilizará el siguiente comando:



```
pip install numpy
```



# Pip {

Pip también se puede utilizar para actualizar y eliminar paquetes de software.

Para actualizar un paquete de software, se utiliza el comando **pip install --upgrade**.

Para eliminar un paquete de software, se utiliza el comando **pip uninstall**.

}

# Instalación de pip {

[https://www.youtube.com/watch?v=2wGveK\\_AQE4](https://www.youtube.com/watch?v=2wGveK_AQE4)

<https://www.youtube.com/watch?v=k1-ZHPBe5Ns>

}

```
1 Request get() {
```

```
2  
3  
4  
5     En caso de no tenerla instalada, se puede obtener a  
6     través de la consola escribiendo el comando:
```



```
11 pip install requests
```

```
12  
13 }  
14
```

```
1 Request get() {
```

```
12  
13 }  
14
```



```
import requests
```

```
respuesta = requests.get('https://www.python.org')
```

# Request get() {

El **metodo** `.get()` nos retorna un objeto de tipo **respuesta**, este contiene información variada sobre la solicitud enviada.

Por ahora, solo nos preocuparemos por dos datos:



```
import requests
```

```
respuesta = requests.get('https://www.python.org')  
print(respuesta.status_code)  
print(respuesta.json())
```

# Status code {

Un **código de estado** es un **número de tres dígitos** que se utiliza para indicar el resultado de una solicitud HTTP.

Los códigos de estado se definen en el estándar HTTP y se utilizan por los clientes y servidores web para comunicarse entre sí.

}

# Status code {

Los **códigos de estado HTTP** se dividen en cinco categorías:

**1xx:** Información.

**2xx:** Éxito.

**3xx:** Redirección.

**4xx:** Error del cliente.

**5xx:** Error del servidor.

}

# Status code {

- Los códigos de estado de **información** se utilizan para indicar que el servidor ha recibido y está procesando la solicitud.
- Los códigos de estado de **éxito** se utilizan para indicar que la solicitud se ha completado con éxito.
- Los códigos de estado de **redirección** se utilizan para indicar que el cliente debe redireccionar su solicitud a una nueva URL.

}



# Status code {

- Los códigos de estado de **error del cliente** se utilizan para indicar que el cliente ha cometido un error en su solicitud.
- Los códigos de estado de **error del servidor** se utilizan para indicar que se ha producido un error en el servidor mientras se procesaba la solicitud.

}

# Status code. Ejemplos. {

- **200:** Todo salió como se esperaba, la información fue obtenida.
- **301:** El servidor te está redirigiendo a un endpoint distinto, esto suele pasar si la compañía dueña del servidor cambió de dominio.
- **400:** El servidor considera que la petición tiene el formato incorrecto, hay que revisar los parámetros agregados
- **401:** El servidor considera que no estás autenticado, muchos servicios requieren de credenciales para poder acceder a sus datos

}

# Status code. Ejemplos. {

- **403:** No tienes permiso para acceder a los datos solicitados
- **404:** El recurso al que intentas acceder no se encuentra en el servidor
- **503:** El servidor no se encuentra disponible para manejar la petición

}

# Metodo .json(). {

El **método json()** es un método del objeto **Response** en la biblioteca **requests** de Python.

El método nos retorna los datos solicitados en formato de **diccionario** por lo que aquí es donde podemos extraer la data para utilizarla en nuestro código.

El método json() es una forma conveniente de trabajar con respuestas JSON en Python. Permite a los desarrolladores decodificar fácilmente respuestas JSON a objetos Python, lo que facilita el acceso a los datos de la respuesta JSON.

}

# JSON {

**JSON** es un formato de texto que se basa en **objetos**. Un objeto es un conjunto de pares **clave:valor**, donde la clave es una cadena, y el valor puede ser una cadena, un número, un booleano, un array u otro objeto.

A continuacion, hay un ejemplo de un documento JSON:

}

1 JSON {

12  
13 }  
14



```
{  
  "name": "John Doe",  
  "age": 30,  
  "address": {  
    "street": "123 Main Street",  
    "city": "Anytown",  
    "state": "CA",  
    "zip": "91234"  
  }  
}
```

1 Metodo .json(). {



```
import requests
```

```
respuesta = requests.get('https://openweathermap.org/data/2.5/weather')  
print(respuesta.status_code) # 401  
print(respuesta.json())
```

<https://openweathermap.org/data/2.5/weather>

12  
13 }  
14

# Formato {

```
import requests
# API con data de Pilotos
formato = {
    "driverId": "string",
    "permanentNumber": "string",
    "code": "string",
    "team": "string",
    "firstName": "string",
    "lastName": "string",
    "dateOfBirth": "string:yyyy-mm-dd",
    "nationality": "string"
}

respuesta = requests.get('https://raw.githubusercontent.com/Algorimtos-y-Programacion-2223-2/api-proyecto/main/drivers.json')
print(respuesta.status_code)
print(respuesta.json())
```

<https://raw.githubusercontent.com/Algorimtos-y-Programacion-2223-2/api-proyecto/main/drivers.json>



# Consulta a la API {

```
import requests

# API con data de Pilotos
url = "https://raw.githubusercontent.com/Algorimtos-y-Programacion-2223-2/api-proyecto/main/drivers.json"

formato = {
    "driverId": "string",
    "permanentNumber": "string",
    "code": "string",
    "team": "string",
    "firstName": "string",
    "lastName": "string",
    "dateOfBirth": "string:yyyy-mm-dd",
    "nationality": "string"
}

# Hacemos la consulta a la API
response = requests.get(url)

# Validamos que la respuesta sea exitosa.
if response.status_code == 200:
    # La consulta fue exitosa.
    data = response.json()
    # print(data) # Imprime toda la data
    # Imprimimos cada elemento
    print(type(data))
    for piloto in data:
        print(piloto)
else:
    # Ocurrio un error.
    print(response.status_code)
    print(response.status_code)
    print(response.json())
```

## Ejercicio 21 {

Analice la estructura de datos propuesta en el Reto 1,  
e identifique que tipo de estructura compleja es:

<https://raw.githubusercontent.com/Andresarl16/API-Retos/main/locations-api.json>

}

## Ejercicio 22 {

Intente consultar la siguiente API:

<https://raw.githubusercontent.com/Andresarl16/API-Retos/main/locations-api.json>

}



# TRY EXCEPT

Manejo de excepciones

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

03 {

[Try Except]

- Errores
- Try except

}

# Errores {

Hay muchos tipos de errores en Python. Algunos de los errores más comunes incluyen:

- **Errores de sintaxis:** Estos errores ocurren cuando el código no está escrito correctamente.



```
print(Hola mundo)
```



```
print(Hola mundo)
```



```
SyntaxError: invalid syntax
```

# Errores {

- **Excepciones:** Si una declaración o expresión es **sintácticamente correcta** y causa un **error cuando se intenta ejecutar**, esto es lo que conocemos como **excepción**.



```
x = 100 * (1/0)
```



```
x = 100 * (1/0)  
ZeroDivisionError: division by zero
```

# Try except {

En Python, **try** y **except** se utilizan para **manejar excepciones**. El **bloque try** se ejecuta primero, y si ocurre una excepción, se ejecuta el bloque **except**.

El **bloque except** se utiliza para manejar la excepción y **evitar que el programa se bloquee**.

En resumen, nos permite probar la ejecución de un bloque de código, y manejar los errores en caso de que aparezcan.

De esta manera, se previene que el programa se caiga o “crashee” (falle), preservando así el flujo de instrucciones.

}



# Try except {



```
try:
    # Ocurrira una excepcion
    print(1 / 0)
except ZeroDivisionError:
    # Este codigo sera ejecutado al ocurrir la excepcion
    print("Error al intentar dividir entre 0")
```

}

# Try except {



```
while True:
    try:
        num = int(input("Por favor ingrese un numero: "))
        break
    except ValueError:
        print("Error, debe ingresar un numero.")
```

}

# Try except. Funcionamiento. {

1. Se ejecuta el **bloque try**.
2. **Si no se produce ninguna excepción**, se finaliza la ejecución del **bloque try** y continuamos con el resto del código.
3. **Si se produce una excepción**, se omite el resto de las instrucciones dentro del **try**. Luego, se ejecuta el **bloque except** que coincida con el error producido y continuamos con el resto del código.
4. **Si se produce una excepción que no coincide con ninguna de las excepciones mencionadas** en el **bloque except**, es una excepción no controlada y la ejecución se detiene.

}

# Try except. Exception. {


Hay una manera de que **try except** atrape cualquier tipo de error en Python:

**except Exception as error:**

Esto atraparé cualquier excepción, independientemente de su tipo.

}

# Try except. Exception. {



```
while True:
    try:
        num = int(input("Por favor ingrese un numero: "))
        break
    except Exception as e:
        print(e) # invalid literal for int() with base 10: 'cadena'
        print(type(e)) # <class 'ValueError'>
```

}

# Try except. else. {

El **bloque else** se ejecutará **solo si el bloque try se ejecuta sin errores.**

Esto puede ser útil cuando quieras **continuar el código del bloque try.**

Por ejemplo, si abres un archivo en el bloque try, podrías leer su contenido dentro del bloque else.

}

# Try except. else. {



```
while True:
    try:
        num = int(input("Por favor ingrese un numero: "))
    except ValueError:
        print("Error, debe ingresar un numero.")
    else:
        print(num)
        break
```

}

## Ejercicio 23 {

Utilizar la estructura Try - Except, para validar si un input que introdujo el usuario es un float válido.

}



## Ejercicio 24 {

Dada una lista de nombres, pedir al usuario la posición de la lista que desea consultar.

Utilizar la estructura Try - Except, para validar si el input que introdujo el usuario es un int y además si es una posición válida en la lista.

```
nombres = ["Andrea", "Daniela", "Pedro", "Juan", "Gabriel"]
```

```
}
```