

Python Programming

Módulo 1

Tipos de datos y Colecciones

Tipos de datos

Python es un lenguaje de “tipado” dinámico. Esto quiere decir que las variables, como se las conoce en otros lenguajes, no son una suerte de cajas que aceptan únicamente un tipo de datos, sino más bien etiquetas que se le asignan a un objeto de cualquier tipo. No es necesario ahondar en cómo el intérprete maneja internamente las variables, simplemente tener en cuenta que no están ligadas a un tipo de dato en específico, y que acostumbramos a llamarlas “objetos” (ya veremos por qué).

Python tiene los tipos de dato *entero(int)*, *flotante(float)*, *complejo(complex)*, *booleanos(bool)* y *strings(str)*.

Podemos asignar el nombre que queramos, respetando no usar las palabras reservadas ([lista de palabras reservadas](#)) de Python ni espacios, guiones o números al principio.

Tipos de datos

Comencemos por abrir la consola interactiva de Python y escribir lo siguiente.

(A partir de ahora, siempre que veas los caracteres ">>>" da por sentado que estamos trabajando sobre la consola interactiva).

```
>>> dato = 1
```

Hemos creado un objeto que contiene el número 1. Escribiendo el nombre de un objeto en la consola interactiva obtenemos su valor.

```
>>> dato  
1
```

La función `type()`, nos permite averiguar el tipo de dato de una variable.

```
>>> type(dato)  
<class 'int'>
```

Como los objetos no están ligados a un tipo específico de datos, ahora podemos reemplazar el contenido de `dato` por una cadena:

```
>>> dato = "Hola mundo"
```

(En Python las cadenas se construyen con comillas dobles o simples).

Tipos de datos

Ya hemos visto dos tipos de datos: un número entero (`int`) y una cadena (`str`). Continuemos creando un número de coma flotante y un booleano.

```
>>> b = True
```

Un objeto booleano puede contener los valores `True` o `False` (ambos valores con Mayúscula la primera letra).

Los valores flotantes la coma como punto.

```
>>> pi = 3.14
```

Y por último no tan utilizados los complejos:

```
>>> z = 5 + 3j
```

(Este ejemplo tiene parte real con valor 5 y parte imaginaria 3)

None

Cuando queremos crear un objeto, pero por el momento no asignarle ningún valor, generalmente se utiliza `None` (en inglés, literalmente, “*nada*”). Esta palabra se usa para que la variable (u objeto) no tenga un tipo de dato asociado. `None`, no es un tipo de dato.

```
>>> a = None
>>> type(a)
<class 'NoneType'>
```

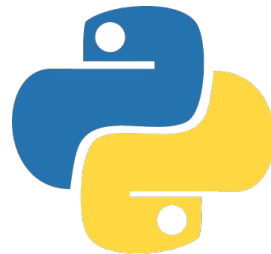
Casting

Cast o *Casting* significa convertir un tipo de dato a otro. Vimos tipos de datos como los int, string o float. Pues bien, es posible convertir de un tipo a otro.

- **Conversión implícita:** Es realizada automáticamente por Python. Sucede cuando realizamos ciertas operaciones con dos tipos distintos.

```
>>> a = 1
>>> b = 2.5
>>> a = a + b
>>> print(a)
>>> 3.5
```

Podemos ver como internamente Python ha convertido el `int` en `float` para poder realizar la operación, y la variable resultante es `float`.



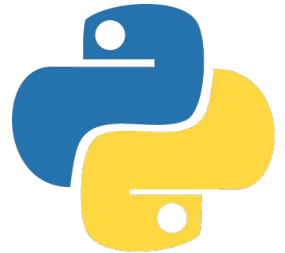
Casting

Conversión explícita: Es realizada expresamente por nosotros, como por ejemplo convertir de `str` a `int` con `str()`.

```
>>> a = 10.4
>>> int(a)
>>> 10
```

```
>>> b = 50
>>> str(b)
>>> '50'
>>> edad = "45"
>>> int(edad)
>>> 45
```

Podemos hacer conversiones entre tipos de manera explícita haciendo uso de diferentes funciones que nos proporciona Python. Las más usadas son las siguientes: `float()`, `str()`, `int()`



Colecciones

Python tiene una variedad de colecciones para agrupar datos en estructuras de diferentes tipos. Tenemos colecciones ordenadas o no ordenadas y colecciones mutables o inmutables.

Las *colecciones ordenadas* son aquellas que pueden ser indexadas, es decir, que se pueden usar números enteros para acceder a sus elementos. Son colecciones ordenadas (o secuencias) las listas y las tuplas. Como colecciones no ordenadas tenemos los diccionarios. También existen los sets y los frozensets, estos últimos no tan empleados.

Listas

Las listas son objetos mutables (es decir, su contenido puede variar) y ordenados. En otros lenguajes existen estructuras similares. Podemos acceder a sus elementos a partir de su posición, indicando un índice entre corchetes, comenzando desde el 0.

```
>>> datos = [1, 2, 3]
>>> datos[0]
1
>>> datos[1]
2
>>> datos[2]
3
```

Los índices pueden ser negativos, para indicar que se debe empezar a contar desde el último elemento:

```
>>> datos[-1]
3
>>> datos[-2]
2
```

Listas

La indexación se puede realizar con uno, dos o tres parámetros. Con dos parámetros se puede seleccionar porciones (slices) de la secuencia y el tercer parámetro indica el paso con el que los elementos deben tomarse.

```
>>> animales =  
["Gato", "Perro", "Gallo", "Caballo", "Tiburon", "Pajaro"]  
>>> animales[2:5]  
['Gallo', 'Caballo', 'Tiburon']  
  
>>> animales[0:5:2]  
['Gato', 'Gallo', 'Tiburon']
```



Listas

Los elementos dentro de una lista no necesariamente tienen que ser del mismo tipo. Incluso pueden contener otras listas.

```
>>> b = [3.14, True, ["Hola mundo", False]]
>>> b[0]
3.14
>>> b[1]
True
>>> b[2]
['Hola mundo', False]
>>> b[2][0]
'Hola mundo'
>>> b[2][1]
False
```

Como las listas son objetos mutables, podemos cambiar el objeto en una posición determinada con una simple asignación.

```
>>> a = [1, 2, 3, 4]
>>> a[2] = "Hola mundo"
>>> a
[1, 2, 'Hola mundo', 4]
```

Listas

Las listas poseen funciones propias, métodos, con los cuales podemos trabajarlas de forma más óptima.

Para añadir un elemento al final de una lista, utilizamos el `append()`.

```
>>> a = [1, 2, 3, 4]
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
```

Y para insertar un elemento en una posición específica, la función `insert()` toma como primer argumento un índice de base 0 seguido del objeto a insertar.

```
>>> a.insert(2, -1)
>>> a
[ 1, 2, -1, 3, 4, 5]
```

Existen otros métodos que poseen las listas, para más información podemos ver la documentación oficial que habla sobre el tema: [más sobre listas](#).

Tuplas

Las tuplas son similares a las listas, pero son objetos inmutables. Una vez creada una tupla, no pueden añadirse ni removerse elementos.

```
>>> t = (1, 2, 3, 4)
```

Se crean indicando sus elementos entre paréntesis (aunque también pueden omitirse) y, al igual que las listas, pueden contener objetos de distintos tipos, incluso otras tuplas. Se accede a sus elementos indicando su posición entre corchetes.

Es importante aclarar que, para crear una tupla con un único elemento, se debe usar (nótese la coma):

```
>>> x = (1,)
```

Ya que los paréntesis son también utilizados para agrupar expresiones y, de lo contrario, Python no podría distinguir si se trata de una expresión o de una tupla con un único elemento.

Siempre que requieras de un conjunto ordenado de objetos, pensá si su contenido será modificado en el transcurso del programa. En caso afirmativo, creá una lista. De lo contrario, una tupla.

Diccionarios

El último tipo de colección que veremos se llama *diccionario*. Es lo que en otros lenguajes se conoce como *vector* o *lista asociativa*. Está constituido por pares de una clave y un valor.

```
>>> d = {"a": 1, "b": 2}
```

En este caso, las claves son las cadenas “a” y “b”, que están asociadas a los números 1 y 2, respectivamente. Así, accedemos a los valores a partir de su respectiva clave.

```
>>> d["a"]  
1  
>>> d["b"]  
2
```

Los valores en un diccionario pueden ser de cualquier tipo, también otros diccionarios. Las claves solo pueden ser objetos inmutables (por ejemplo, cadenas, enteros, números de coma flotante, tuplas; no así listas) y no pueden repetirse dentro de un mismo diccionario.

```
>>> d = {123: "Hola, mundo", True: 3.14,  
(1, 2): False}  
>>> d[123]  
'Hola, mundo'  
>>> d[True]  
3.14  
>>> d[(1, 2)]  
False
```

Diccionarios

Los pares clave-valor de un diccionario no pueden ser accedidos a partir de un índice porque no es una colección ordenada.

Para cambiar el valor de una clave mantenemos la misma sintaxis que en las listas:

```
>>> d = {"a": 1, "b": 2}
>>> d["b"] = 3.14
>>> d
{'a': 1, 'b': 3.14}
```

Lo cual también sirve para añadir elementos:

```
>>> d["c"] = "Hola mundo"
>>> d
{'a': 1, 'c': 'Hola mundo', 'b': 3.14}
```

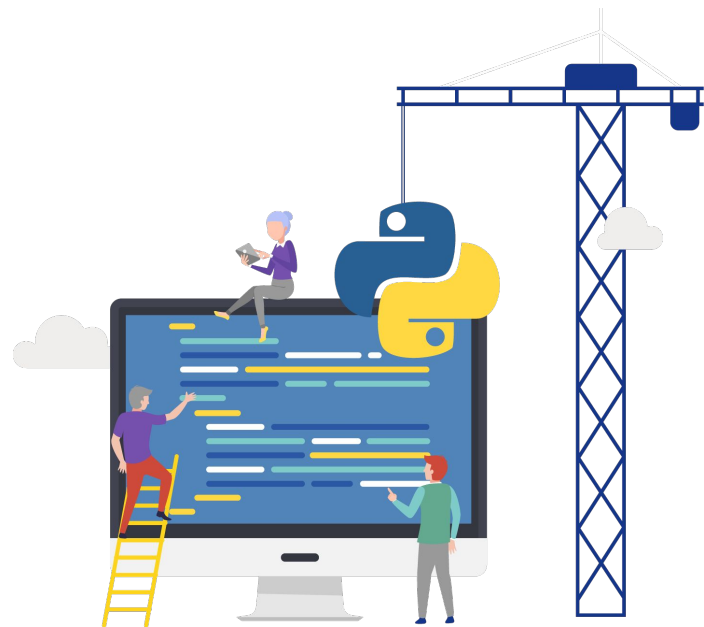
Y para remover un par clave-valor:

```
>>> del d["b"]
>>> d
{'a': 1, 'c': 'Hola mundo'}
```

len() y del

Para conocer la cantidad de elementos de una colección indistintamente del tipo, utilizamos la función `len()`. El `len()` es una función propia del lenguaje, NO es un método puntual de alguna colección.

```
>>> len([1, 2, 3])
3
>>> len((True, False))
2
>>> len({"a": 1})
1
>>> len("Hola mundo")
10
```



len() y del

La palabra clave “del” en Python se usa principalmente para eliminar objetos. Dado que todo en Python representa un objeto de una forma u otra. “del” se puede usar para eliminar una lista, eliminar una parte de una lista, eliminar diccionarios, eliminar pares clave-valor de un diccionario, eliminar variables, etc.

```
>>> numeros = [10,20,30]
>>> del numeros[1]
>>> numeros
>>> [10, 30]
```

```
>>> x = 10
>>> x
>>> 10
>>> del x
>>> x
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Unpacking

Las listas y tuplas soportan un método llamado *unpacking* (cuya traducción podría ser “desempaquetamiento”). Veámos en un ejemplo:

```
>>> t = (1, 2, 3)
>>> a, b, c = t
>>> a
1
>>> b
2
>>> c
3
```

La operación `a, b, c = t` equivale a:

```
>>> a = t[0]
>>> b = t[1]
>>> c = t[2]
```

Es importante que la cantidad de objetos a la izquierda coincida con el número de elementos en la colección a la derecha.

¡Muchas gracias!

¡Sigamos trabajando!

Python Programming

Módulo 1

Operadores

Operadores aritméticos

Los operadores aritméticos son los más comunes que nos podemos encontrar, y nos permiten realizar operaciones matemáticas sencillas.

Los operadores aritméticos tienen orden de prioridad, primero se respeta el () paréntesis, luego los de mayor prioridad en orden:

- Exponente
- Multiplicación, División, División Entera, Módulo
- Suma, Resta

Operador	Nombre	Ejemplo	Resultado
+	Suma	5 + 6	11
-	Resta	10 - 2	8
*	Multiplicación	3 * 10	30
/	División	5 / 2	2.5
%	Módulo(Resto)	9 % 2	1
**	Exponente	10 ** 2	100
//	División Entera	9 // 2	4

Operadores asignación

Los operadores de asignación nos permiten realizar una operación y almacenar su resultado en la variable inicial.

Podemos ver cómo realmente el más importante es el = igual. El resto son abreviaciones de otros operadores que habíamos visto con anterioridad.

Los operadores de asignación no son más que atajos para escribir de manera más corta, y asignar su resultado a la variable inicial. El operador += en `x+=1` es equivalente a `x = x + 1`.

NO existe el operador ++ como existe en otros lenguajes.

Operador	Ejemplo si x = 7	Equivalente	Resultado
=	x = 7	x = 7	7
+=	x += 2	x = x + 2	9
-=	x -= 2	x = x - 2	5
*=	x *= 2	x = x * 2	14
/=	x /= 2	x = x / 2	3.5
%=	x %= 2	x = x % 2	1
//=	x //= 2	x = x // 2	3
**=	x **= 2	x = x ** 2	49

Operadores relacionales

Los operadores relacionales, o también llamados de comparación nos permiten saber la **relación existente entre dos variables**. Se usan para saber si por ejemplo un número es mayor o menor que otro. Dado que estos operadores indican si se cumple o no una operación, el valor que devuelven es True o False.

Operador	Nombre	Ejemplo si x=5	Resultado
==	Igual	x == 6	False
!=	Distinto	x != 8	True
>	Mayor	x > 4	True
<	Menor	x < 3	False
>=	Mayor o igual	x >= 20	False
<=	Menor o igual	x <= 5	True

Operadores lógicos

Los operadores relacionales, o también llamados de comparación nos permiten saber la **relación existente entre dos variables**. Se usan para saber si por ejemplo un número es mayor o menor que otro. Dado que estos operadores indican si se cumple o no una operación, el valor que devuelven es True o False.

Operador	Acción
and	Devuelve True si ambos elementos son True
or	Devuelve True si al menos un elemento es True
not	Devuelve el contrario, True si es Falso y viceversa

Operadores especiales

Operador identidad

El operador de identidad nos indica si dos variables hacen referencia al mismo objeto. Esto implica que si dos variables distintas tienen el mismo `id()`, misma dirección de memoria, el resultado de aplicar el operador `is` sobre ellas será `True`.

```
>>> a = 10
>>> b = 10
>>> a is b
True
```

Esto es debido a que Python reutiliza el mismo objeto que almacena el valor 10 para ambas variables. De hecho, si usamos la función `id()`, podemos ver que el objeto es el mismo.

```
>>> id(a)
2618946775632
>>> id(b)
2618946775632
```

Definido el operador `is`, es trivial definir `is not` porque es exactamente lo contrario.

Operador de pertenencia (membresía)

El operador de pertenencia `in` permite saber si un elemento está contenido en una secuencia. Por ejemplo, si un número está contenido en una lista de números. O una key(clave) en un diccionario.

```
>>> datos = [1,2,3]
```

```
>>> 3 in datos
```

```
True
```

```
>>> 10 in datos
```

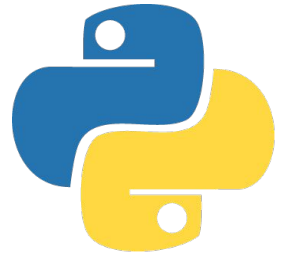
```
False
```

```
>>> alumnos = {"Juan":10,"Lorena":3}
```

```
>>> "Juan" in alumnos
```

```
True
```

Por último, el operador `not in` realiza lo contrario al operador `in`. Verifica que un elemento no está contenido en la secuencia.



Operador de Walrus

El operador `walrus` se introdujo en la versión Python 3.8, y se trata de un operador de [asignación](#) con una funcionalidad extra. Se representa con dos puntos seguidos de un igual `:=`, lo que tiene un parecido a una morsa, siendo `":"` los ojos y `"="` los colmillos, por lo que de ahí viene su nombre (*walrus* significa *morsa* en inglés).

Ejemplo:

```
>>> dato = "Hola"
>>> print(dato)
Hola
```

Con `walrus` podemos simplificar las dos líneas de código anterior en tan solo una.

(Asignación y salida por pantalla en una sola línea)

```
>>> print(dato:="Hola")
Hola
```

Operador a nivel bit , bitwise

Los operadores a nivel de bit son operadores que actúan sobre números enteros, pero usando su representación binaria. Si no sabes cómo se representa un número en forma binaria, no hace falta que te hagas mucho problema, ya que estos operadores no se usan tan frecuentemente como los otros. Su uso puede ser más empleado en lenguajes de bajo nivel, y Python no es un lenguaje de esas características. Pero no está de más conocerlos.

Atención: Estos operadores no se emplean para reemplazar al `and` o al `or`, los operadores bitwise no son los operadores lógicos que estamos acostumbrados a usar. Son operadores especiales que trabajan a nivel bit. Por eso no se usan habitualmente, dejamos en claro que estos no reemplazan a los otros, son cosas distintas.

Operador	Acción
	OR bit a bit
&	AND bit a bit
~	NOT bit a bit
^	XOR bit a bit
>>	Desplazamiento bit a la derecha
<<	Desplazamiento bit a la izquierda

¡Sigamos trabajando!

Python Programming

Módulo 1

Funciones built-in (integradas)

Funciones built-in

El intérprete de Python tiene un número de **funciones integradas** (*built-in*), las cuales están siempre disponibles. Algunas las usamos cotidianamente programando en Python, como, por ejemplo:

```
>>> print()
```

Una de las acciones básicas e imprescindibles que tiene que realizar un programa es la de mostrar información por pantalla: texto, números, resultado. La función `print()` es la que nos permite mostrar por pantalla.

```
>>> input()
```

La función `input()` permite a los usuarios introducir datos desde la entrada estándar (normalmente se corresponde con la entrada de un teclado). Los datos son tomados como `'str'`, aunque los ingresos fueran números, son tomados como string, luego hay que convertirlos.

>>> range()

La “función” `range()` es algo un poco más complejo que una función, pero se utiliza como si fuera una función.

El `range()` con un único argumento se escribe `range(n)` y crea una secuencia inmutable de `n` números enteros consecutivos que empiezan en `0` y acaba en `n - 1`.

Con dos argumentos se escribe `range(m, n)` y crea una secuencia inmutable de enteros consecutivos que empieza en `m` y acaba en `n - 1`.

Con tres argumentos se escribe `range(m, n, p)` y crea una secuencia inmutable de enteros que empieza en `m` y acaba `n-1`, aumentando los valores de `p` en `p`. Si `p` es negativo, los valores van disminuyendo de `p` en `p`.

El `range()` es fácil de iterar o recorrer con un `for` y también sirve para crear listas con la función incorporada `list()`.

```
>>> int()
>>> float()
>>> str()
```

Hacer un *cast* o *casting* significa convertir un tipo de dato a otro. Anteriormente hemos visto tipos como los [int](#), [string](#) o [float](#). Pues bien, es posible convertir de un tipo a otro con las funciones correspondientes, siempre y cuando se pueda realizar la conversión.

Inclusive se puede convertir diferentes conjuntos a listas, con `list()`, o a tuplas con `tuple()`.

```
>>> len()
```

Retorna el tamaño (el número de elementos) de un objeto. El argumento puede ser una secuencia (como una cadena, un objeto byte, una tupla, lista o un rango) o una colección (como un diccionario, un set o un frozen set).

Para ver más funciones built-in podés ver la documentación oficial de [funciones built-in](#)

		Funciones Built-in		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

¡Sigamos trabajando!

Python Programming

Módulo 1

Funciones propias

Funciones en Python

Anteriormente hemos usado funciones nativas que vienen con Python, como `len()` para calcular la longitud de una lista, o `type()` para saber el tipo de dato. Pero al igual que en otros lenguajes de programación, también podemos definir nuestras propias funciones. Para ello hacemos uso de la palabra reservada `def`.

```
def funcion():  
    pass
```

La función anterior no hace nada. La palabra reservada `pass` sirve para rellenar un bloque de código requerido sintácticamente (no podría estar vacío porque lanzaría un error de sintaxis).

La palabra reservada `pass` no solo se emplea para completar código de bloques vacíos en funciones, también se la emplea para completar bloques vacíos de sentencias de control. Y para completar clases vacías en orientación a objetos.

Anatomía de una función

```
def nombre_funcion(argumentos):  
    # código  
    return retorno
```

Las funciones que definimos permiten agrupar un bloque de código o conjunto de instrucciones. Opcionalmente, reciben uno o más argumentos y retornan un valor.

En Python, para crear una función empleamos la palabra reservada `def` seguido de su nombre y sus parámetros entre paréntesis, en caso de tener. A continuación, se coloca su contenido utilizando una indentación.



Vamos a crear un ejemplo de nombre sumar con dos parámetros cuyos nombres son a y b, y que retorne el resultado de la suma de ambos.

```
def sumar(a, b):  
    c = a + b  
    return c
```

El uso del return permite salir de la función (terminarla) y transferir la ejecución de vuelta a donde se realizó la llamada. Se puede devolver uno o varios parámetros, pero una vez que se llama al return se detiene la ejecución de la función y se vuelve o retorna al punto donde fue llamada. Cuando una función no retorna ningún valor (es decir, no incluye la palabra reservada return), por defecto retorna None.

Ejemplo:

```
def sumar(a, b):  
    print( a + b)  
  
print(sumar(10,20))
```

out: 30

out: None

Los nombres de las funciones en Python siguen las mismas reglas que los nombres de las variables, se escriben en minúscula y con guiones bajos en lugar de espacios vacíos.

Argumentos por defecto

Para indicar parámetros opcionales, indicamos su valor por defecto con el signo de igual (=).

```
def imprimir_mensaje(mensaje="No has escrito nada."):
    print(mensaje)
```

```
imprimir_mensaje("Hola mundo")
imprimir_mensaje()
```

out: Hola mundo

out: No has escrito nada.



Los argumentos con valores por defecto siempre deben ir **al final de la función**. Por ejemplo, el siguiente código es inválido:

```
def sumar(a=5, b):  
    return a + b
```

A los argumentos con valores por defecto se los conoce como *keyword arguments* o **“argumentos por nombre”**, ya que al llamar a la función se puede especificar su valor indicando su nombre, como se ve en el ejemplo a continuación.

```
def sumar(a=5, b=10):  
    return a + b
```

```
sumar()
```

```
out: 15
```

```
sumar(b=50)
```

```
out: 55
```

```
sumar(7, 5)
```

```
out: 12
```

```
sumar(a=7, b=5)
```

```
out: 12
```

Esto permite que en la segunda llamada **a** mantenga su valor por defecto, mientras que **b** obtenga el número 50.

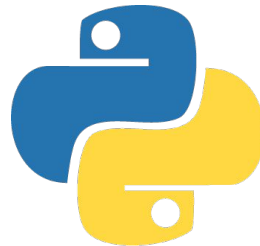
En el caso que se especifiquen ambos argumentos, indicar o no su nombre es indistinto.

Documentación en funciones

Ahora que ya tenemos nuestras propias funciones creadas, muchas veces en algún proyecto debamos compartirlas con otros programadores, para que sean parte de un programa complejo.

Leer código ajeno no es tarea fácil, por eso es importante documentar las funciones.

```
def sumar(a, b):  
    """  
    Descripción de la función. Como debe ser usada,  
    que parámetros acepta y que devuelve  
    """  
    return a+b
```



Para crear un texto que nos diga qué hace la función podemos crear un comentario del tipo párrafo usando la triple comilla `"""` al principio de la función. Es únicamente al principio no en otro lado. Esto no forma parte del código, es un simple comentario un tanto especial, conocido como *docstring*.

Ahora cualquier persona que haga uso de nuestra función, podrá acceder a la ayuda de la función con `help()` y obtener las recomendaciones nuestras de cómo debe ser usada.

```
help(sumar)
```



¡Sigamos trabajando!

Python Programming

Módulo 1

Argumentos de longitud variable

Argumentos de longitud variable

Imaginemos que queremos una función `sumar()` como la de los ejemplos, pero en este caso necesitamos que sume todos los números de entrada que se le pasen, sin importar si son 3, 100 o 1000. Una primera forma de hacerlo sería con una lista:

```
def sumar(numeros):  
    parcial = 0  
    for n in numeros:  
        parcial += n  
    return parcial
```

```
sumar([10,0,5,4])
```

```
out: 19
```

La propuesta es válida, cumple con nuestro requisito, pero realmente no estamos trabajando con argumentos de longitud variable. En realidad, tenemos un solo argumento, una lista.

Por suerte para nosotros, Python tiene una solución muy potente y sencilla a la vez. Si declaramos un argumento con `*` (asterisco típicamente acompañado con el nombre *args*, pero no es obligatorio usar ese nombre).

Al declarar la función con un parámetro *, hará que los argumentos, sean la cantidad que fueran, lleguen individualmente y que una vez recibido la función lo empaquete en una tupla de manera automática.

*Atención: No confundir * con los punteros en otros lenguajes de programación, no tiene nada que ver ¡En Python existen los punteros a memoria!*

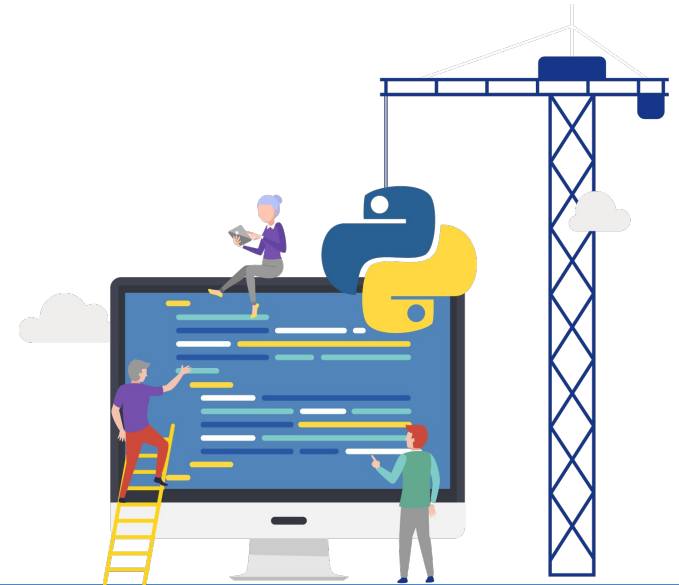
```
def sumar(*args):  
    print(type(args))  
    print(args)  
    total = 0  
    for n in args:  
        total += n  
    return total
```

```
sumar(10,0,5,4)
```

```
out: <class 'tuple'>
```

```
out: (10,0,5,4)
```

```
out: 19
```



Usando doble ****** es posible también tener como parámetro de entrada una cantidad arbitraria de elementos almacenados en forma de clave y valor (diccionario).

El doble asterisco ****** (usualmente acompañado por el nombre *kwargs*) captura cualquier keyword argument que no haya sido definido junto con la función. Los argumentos capturados por este operador son almacenados, como dijimos, en un diccionario que tiene como claves los strings que representan los nombres de los argumentos, y como valor, el valor del argumento. Este operador debe ir al final de la definición de otros parámetros, si los hubiera.

Ejemplo:

```
def sumar(**kwargs):
    print(type(kwargs))
    print(kwargs)
    total = 0
    for n in kwargs:
        total+= kwargs[n]
    print(total)

sumar(a=5, b=20, c=23)

out: <class 'dict'>
out: {'a': 5, 'b': 20, 'c': 23}
out: 48
```

Orden para usar diferentes argumentos

Si tuviéramos una función que hiciera uso de diferentes tipos de argumentos; deberíamos usarlos con un cierto orden.

Primero **argumentos posicionales** (argumentos comunes), después los **argumentos arbitrarios posicionales** (*args), a continuación, los **keywords arguments** (argumentos por defecto), y por último **número arbitrario de keywords arguments** (**kwargs).

En caso de usarlos de otra manera el resultado puede ser un error, o no poder hacer uso correcto del argumento por defecto.

Ejemplo:

```
def funcion(a,b,*args,c=100,**kwargs):  
    print(a)  
    print(b)  
    print(args)  
    print(c)  
    print(kwargs)
```



¡Sigamos trabajando!

Python Programming

Módulo 2

Características y operaciones sobre cadenas

Strings, conceptos básicos

Python fue creado para desarrollar scripts (programas para automatización de tareas). El tiempo lo convirtió en un lenguaje de programación multipropósito.

Tiene incorporando todo un conjunto de características que lo hacen ideal para tareas de *scripting*. Por ejemplo, muchas de las herramientas de varios sistemas operativos basados en Linux están escritas en Python, razón por la cual las distribuciones suelen contener uno o dos intérpretes del lenguaje.

Al realizar tareas de scripting trabajaremos todo el tiempo con cadenas de caracteres.

Este trabajo implica tareas como: buscar un carácter, una palabra o frase en una cadena, eliminar caracteres innecesarios, reemplazar unos caracteres por otros, etc. Las cadenas de Python proveen de forma estándar varios métodos (funciones propias del objeto) para realizar este tipo de operaciones.

Primero, tengamos en cuenta que las cadenas en Python funcionan como las colecciones, en donde cada uno de los elementos es un carácter que se puede ubicar por un índice. De modo que las operaciones de «Slicing» y acceso a elementos a través de índices funciona sin diferencias respecto de las listas y tuplas. Hagamos algunas pruebas.

Caracteres :	P	y	t	h	o	n
Índice :	0	1	2	3	4	5
Índice inverso :	-6	-5	-4	-3	-2	-1

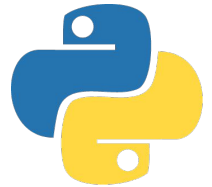
```
>>> texto = "Python"
>>> texto[2]
't'
```

```
>>> texto[2:]
'thon'
```

¿Se puede recorrer con un for? Si.

```
for n in texto:
    print(n)
```

```
out:
P
y
t
h
o
n
```



Se pueden concatenar:

```
>>> nombre = "Luke"  
>>> apellido = "Skywalker"  
>>> nombre + " " + apellido  
'Luke Skywalker'
```

Es posible la concatenación múltiple (copias):

```
>>> saludo = "Hola!"  
>>> saludo*3  
'Hola!Hola!Hola!'
```

Las cadenas son inmutables. Esto significa que una vez creadas no pueden modificarse. En efecto, si intentamos modificar una cadena el intérprete nos indica que a ésta no se pueden asignar elementos.

```
>>> texto = "python"  
>>> texto[0] = "P"  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item  
assignment
```

Dentro de poco, vamos a ver como solucionamos esta dificultad.

Caracteres especiales

Los caracteres especiales, como el de salto de línea (`\n`) o tabulación (`\t`), son difíciles de detectar al imprimir una cadena puesto que, justamente, no se imprimen literalmente sino que toman alguna forma en particular: el carácter `\n` se traduce como nueva línea y el carácter `\t` como un segmento vacío de longitud equivalente al de cuatro espacios. Por ejemplo, este código:

```
>>> dato = "\tHola\nChau"
>>> print(dato)
        Hola
Chau
```

Para evitar que los caracteres especiales sean “traducidos” a su representación correspondiente, Python nos provee la función `repr()`.

Que imprime en pantalla, en efecto, los caracteres literalmente:

```
>>> print(repr(dato))
'\tHola\nChau'
```

Esto es especialmente útil cuando obtenemos información de un archivo o alguna otra fuente similar y debemos trabajar con la cadena literal.

Métodos de los strings

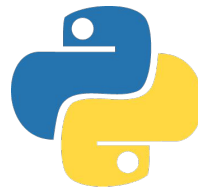
Las cadenas, nos proporcionan [métodos](#) que nos facilitan su manipulación. Por ejemplo, una tarea bastante común, es la de querer determinar si una cadena comienza o termina con determinados caracteres, tenemos los métodos `startswith()` y `endswith()` que retornan un booleano.

```
>>> frase = "¿Caminos? a donde vamos no necesitamos caminos"
>>> frase.startswith("¿Caminos?")
True

>>> frase.startswith("¿")
True

>>> frase.endswith("caminos")
True

>>> frase.endswith(".")
False
```



Otra operación bastante común es la de querer eliminar los espacios iniciales y finales de una cadena. Por ejemplo, si le solicitamos el nombre a un usuario a través de la consola (vía la función `input()`) y accidentalmente ingresa un espacio al final antes de presionar la tecla Enter. En ese caso tendríamos una cadena como la siguiente:

```
>>> nombre = "Juan "  
>>> nombre.strip()  
'Juan'
```

Pero también `.strip()` nos puede ser útil para quitar caracteres especiales del principio o del final de un `str`.

```
>>> dato = "\t\nHola\nChau"  
>>> print(dato.strip())  
Hola  
Chau
```

Borro el `\t\n` del principio, pero no manipulo el `\n` del medio. Si hubiera estado al final, lo habría borrado también. Esto es especialmente útil al trabajar con los archivos texto (`txt`), pudiendo eliminar el salto de línea al leer renglones.

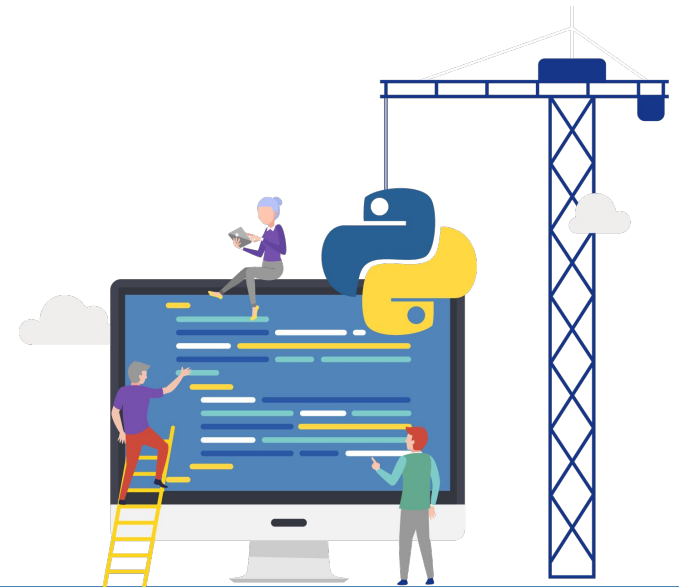
Para reemplazar un (conjunto de) carácter(es) por otro tenemos a disposición la función `replace()`.

```
>>> frase = "¡Nuestro gato se llamaba Indiana!"  
>>> frase.replace("gato", "perro")  
'¡Nuestro perro se llamaba Indiana!'
```

El primer argumento corresponde a la cadena buscada y el segundo, a la cadena que se quiere poner en su lugar. Esto ocurre para todas las apariciones del primer argumento en la cadena original.

El `replace()` retorna una nueva cadena con el reemplazo correspondiente; no modifica la cadena original. Si quisiéramos modificar la original se debería de hacer así:

```
>>> frase = frase.replace("gato", "perro")
```



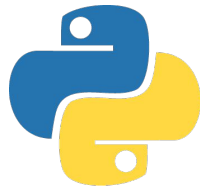
Cuando queremos cortar una cadena en función de un delimitador (como una coma o un espacio vacío), usamos la función `split()`, que retorna una lista.

```
>>> nombres = "Lautaro José Sofía"
>>> nombres.split()
['Lautaro', 'José', 'Sofía']
```

El `.split()` sin argumento, utiliza por defecto el carácter espacio vacío como separador. Ahora si deseamos usar otro delimitador podemos pasarlo como parámetro:

```
>>> datos = "Juan,38,Tierra del Fuego"
>>> datos.split(",")
['Juan', '38', 'Tierra del Fuego']
```

Al finalizar los elementos de la lista resultante no contienen el delimitador. Si el delimitador no es encontrado en la cadena, el resultado es una lista de un solo elemento.



El método **find()** toma como argumento una cadena y retorna la posición en la que se encuentra dentro de la cadena desde donde fue invocada la función, o -1 en caso de no ser encontrada.

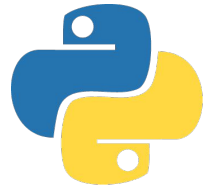
```
>>> cita = "lo esencial es invisible a los ojos"
>>> cita.find("invisible")
15
>>> cita.find("principio")
-1
```

El **count()** retorna el número de veces que se repite un conjunto de caracteres especificado:

```
>>> cancion = "...hacer cosas imposibles, cosas
imposibles, imposibles..."
>>> cancion.count("imposibles")
3
```

.upper() y **.lower()** convierten los caracteres de la cadena a mayúsculas y minúsculas, respectivamente.

```
>>> saludo = "Hola Mundo"
>>> saludo.upper()
'HOLA MUNDO'
>>> saludo.lower()
'hola mundo'
```



Otros métodos

- **`str.isdecimal()`** devuelve True si todos los caracteres de la cadena son decimales y si hay al menos un carácter. Falso en caso contrario. Los caracteres decimales son aquellos que se pueden usar para formar números en base 10
- **`str.swapcase()`** devuelve una copia de la cadena con caracteres en mayúscula convertidos a minúsculas y viceversa.
- **`str.title()`** devuelve una versión en formato título. Las palabras comienzan con un carácter en mayúscula y el resto de caracteres en minúsculas.
- **`str.capitalize()`** devuelve una copia de la cadena con su primer carácter en mayúscula y el resto en minúsculas.

Puedes encontrar más en la documentación oficial:
[“métodos de los strings”](#)

¡Sigamos trabajando!

Python Programming

Módulo 2

Formar cadenas

Formar cadenas

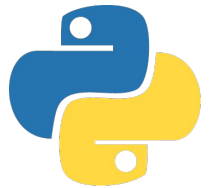
Es bastante usual tener que incluir otros tipos de datos dentro de una cadena, podemos usar la función `str()` para poder concatenar otros tipos de datos. Pero cuando debemos hacerlo con una gran cantidad de datos, sobre otro string, nos resultará conveniente utilizar el método `format()`.

```
>>> nombre = "Pablo"  
>>> edad = 30  
>>> mensaje = "Hola, mi nombre es {0}. Tengo {1} años."  
>>> mensaje.format(nombre, edad)  
'Hola, mi nombre es Pablo. Tengo 30 años.'
```



En el código creamos una cadena, en donde queremos incluir los objetos nombre y edad (una cadena y un entero, respectivamente). En su lugar, colocamos `{0}` y `{1}` (y así sucesivamente en relación con la cantidad de objetos que queramos incluir) y por último llamamos a `.format()` pasándole como argumentos los elementos que queremos insertar dentro de mensaje.

Las cadenas en Python son objetos inmutables, por lo que `.format()` retorna una nueva cadena con los valores reemplazados.



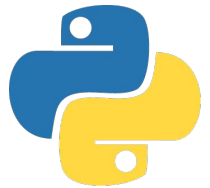
A partir de la versión 3.6 Python introduce una manera más legible, que consiste en poner nombres de variables entre llaves dentro de una cadena y anteponiendo la letra f.

```
>>> nombre = "Pablo"
>>> edad = 30
>>> mensaje = f"Hola, mi nombre es {nombre}. Tengo {edad} años."
>>> mensaje
'Hola, mi nombre es Pablo. Tengo 30 años.'
```

El lenguaje también soporta un sistema más antiguo proveniente del lenguaje C a través del operador %.

```
>>> "Hola, mi nombre es %s y tengo %d años." % (nombre, edad)
'Hola, mi nombre es Pablo y tengo 30 años.'
```

De todas formas, los métodos recomendados son los dos primeros.



¡Sigamos trabajando!

Python Programming

Módulo 2

Trabajar con archivos txt

Escribir y leer archivos

Al igual que en otros lenguajes de programación en Python es posible crear, abrir y leer archivos.

La función incorporada `open()` toma como argumento la ruta de un archivo y retorna una instancia del tipo `file`.

```
f = open("archivo.txt", modo)
```

El *modo* es la forma de cómo se va a trabajar el archivo. Para leer, `modo="r"`. Este es el único modo que cuando se usa `open()` y no se pasa de qué manera interactuar con el archivo se toma automáticamente `"r"`.

Para escribir y crear se usa `modo="w"`, además en este modo si el archivo existe se sobrescribe. Y por último el `modo="a"` de append para añadir contenido al final del archivo si existe el mismo.



En cualquiera de los casos anteriores, si no se especifica una ruta, el fichero se busca en el directorio actual (ruta relativa). Y si vas a especificar la ruta (ruta absoluta), la misma se escribe con doble barra para separar los directorios o barra invertida.

```
"C:\\Users\\Anonimo\\Desktop\\Python Programming\\mi_archivo.txt"
```

```
"C:/Users/Anonimo/Desktop/Python Programming/mi_archivo.txt"
```



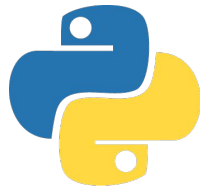
Métodos para trabajar con archivos

Cuando abrimos un archivo para leer, el método `.read()` retorna el contenido del archivo abierto en formato string.

```
f = open("archivo.txt", "r")  
print(f.read())  
f.close()
```

Con `f.readline()` leemos una sola línea del archivo; el carácter de fin de línea (`\n`). Entonces se retorna una cadena.

Si quieres leer todas las líneas de un archivo, puedes usar `f.readlines()` que crea una lista, y cada elemento de esta es un renglón del archivo.



Por otro lado, cuando abrimos un archivo en modo escritura, usamos el método `.write()`

```
f = open("archivo.txt", "w")  
f.write("Hola mundo")  
f.close()
```

La función `write()` graba una cadena en el archivo. Si trabajamos en modo "w" reemplazará todo el contenido del archivo, si el archivo existe, si no existe, lo crea.

Recordemos que para añadir datos al final del archivo sin borrar información previa, el fichero debe abrirse en la modalidad `append` ("a") y también se usa `.write()`.

```
f = open("archivo.txt", "a")  
f.write("Hola ")  
f.write("mundo")  
f.close()
```

También podemos usar el método `writelines()` y pasarle una lista como argumento para que guarde el contenido.

Cuando trabajamos con archivos es importante cerrarlo cuando acabamos de trabajar con ellos, vía el método `close()`. Aunque es verdad que los ficheros normalmente son cerrados y guardados automáticamente, es importante especificarlo para evitar tener comportamientos inesperados.

¡Sigamos trabajando!

Python Programming

Módulo 2

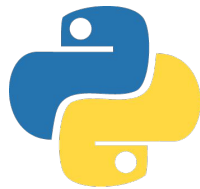
Sentencia with

Uso de 'with' archivos

Es una buena práctica utilizar la sentencia `with` cuando se trata de archivos. La ventaja es que el archivo se cierra correctamente después de que finaliza su uso, incluso si se genera una excepción en algún momento.

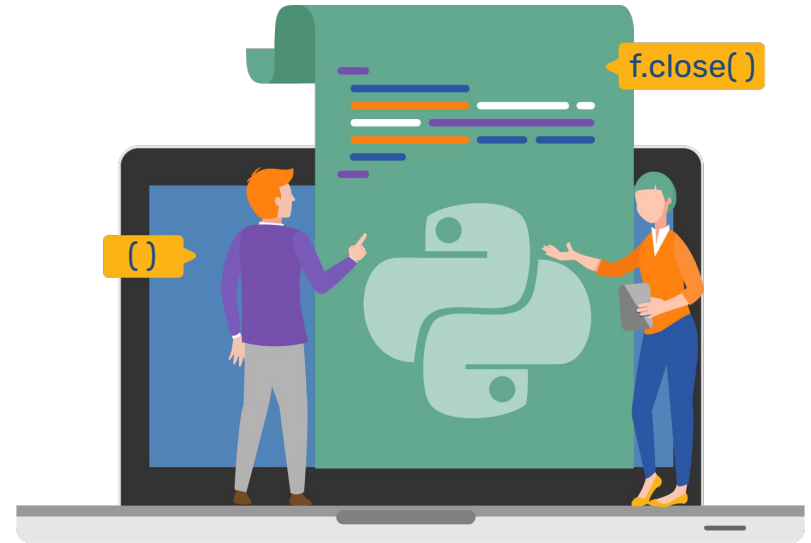
```
#Ejemplo with escribiendo archivo
with open("archivo.txt","w") as f:
    f.write("Hola Mundo!")
```

```
#Ejemplo with leyendo archivo, el parametro "r" al leer es opcional
with open("archivo.txt","r") as f:
    texto = f.read()
```



¡Si no utilizas `with`, debes llamar a `f.close()`, si o si, para cerrar el archivo y liberar inmediatamente los recursos del sistema.

¡Cuidado! Usar `f.write()` sin usar `with` o `f.close()` puede resultar en que no se escriban completamente los datos en el archivo.



¡Sigamos trabajando!

Python Programming

Módulo 3

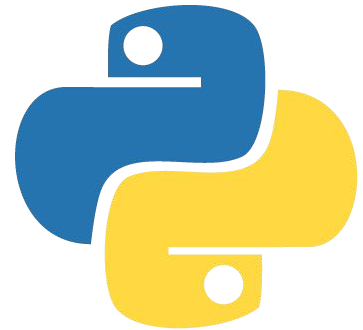
Argumentos de programa

Argumentos de programa

Cuando invocamos un programa a través de la terminal (o línea de comandos o símbolo del sistema) podemos pasarle argumentos como si se tratase de una función, que por lo general determinan alguna acción o configuración inicial del *script*. Dichos argumentos se especifican luego del nombre del programa y separados por espacios.

Supongamos que tenemos un archivo llamado `saludar.py` y que lo ejecutamos desde la terminal del siguiente modo:

```
> python saludar.py Lautaro
```



Ya vimos que, Python es un lenguaje interpretado, los *scripts* que escribimos son ejecutados por un programa al cual llamamos intérprete. En lo que acabamos de escribir, estamos haciendo referencia al intérprete con la palabra `python`. Podemos comprobar esto escribiendo lo siguiente (en Windows):

```
> where python  
C:\python37\python.exe
```

O en alguna distribución de Linux:

```
$which python  
/usr/bin/python
```

Entonces, si `python` es el programa ejecutable, los argumentos que recibirá en el caso anterior son los siguientes:

- `saludar.py`
- `Lautaro`

Para acceder a estos argumentos desde nuestro código, la librería estándar nos provee la lista `sys.argv` en donde cada uno de los argumentos será una cadena según el orden en que fueron especificados.

De modo que si nuestro archivo `saludar.py` contiene el siguiente código...

```
import sys
print("Argumentos:", sys.argv)
```

Al ejecutarlo desde la consola con `python saludar.py Lautaro` veremos impreso en pantalla:

```
Argumentos: ['saludar.py', 'Lautaro']
```

La lista `sys.argv` siempre contendrá, al menos, un elemento. El nombre de nuestro archivo de Python (en este caso, `saludar.py`).

Todos los elementos subsiguientes habrán sido pasados a través de la terminal tal como acabamos de hacerlo. Entonces, si queremos que nuestro programa muestre un saludo al nombre que recibió como argumento a través de la consola, haremos:

```
import sys
nombre = sys.argv[1]
print("¡Hola" + nombre + "!")
```

Puede ocurrir que el usuario se haya olvidado de proveer su nombre. En dicho caso, por tratarse de una lista convencional, intentaremos capturar la excepción `IndexError`:

```
import sys

try:
    nombre = sys.argv[1]
    print("¡Hola" + nombre + "!")
except IndexError:
    print("No me has dicho tu nombre.")
```

Ahora bien, ¿qué tal si queremos que el programa imprima un saludo para cada uno de los nombres que se hayan pasado como argumentos —es decir, separados por espacios—? Bien podríamos hacer:

```
import sys

# Usamos la propiedad de «slicing» para
# ignorar el primer
# argumento, i. e. el nombre del programa.
nombres = sys.argv[1:]
# Chequeamos que la lista no esté vacía.
if nombres:
    for nombre in nombres:
        print(f"¡Hola, {nombre}!")
else:
    print("No has indicado ningún nombre.")
```

¿Cómo deberíamos invocar este programa? Así:

```
> python saludar.py Lautaro José Sofía
```

Y el resultado en pantalla es:

```
¡Hola, Lautaro!  
¡Hola, José!  
¡Hola, Sofía!
```



¡Muchas gracias!

¡Sigamos trabajando!

Python Programming

Módulo 3

Sistema de archivos

Sistema de archivos, OS

La librería estándar de Python provee varias funciones para trabajar con el sistema de archivos: listar, copiar, mover, eliminar, renombrar, etc. Están desperdigadas en dos módulos: `os` y `shutil`.

Dentro del módulo `os` nos interesan estas funciones:

- `os.listdir()`
- `os.getcwd()`
- `os.mkdir()`
- `os.path.exists()`
- `os.rename()`
- `os.remove()`
- `os.rmdir()`
- `os.system()`
- `os.name`

La función `os.listdir()` retorna una lista con los archivos y carpetas en una ruta en particular que se pasa como argumento.

```
>>> import os
>>> os.listdir("C:\\Users\\Anonimo\\AppData\\Local\\Programs\\Python\\Python39")
['DLLs', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'python.exe',
'python3.dll', 'python37.dll', 'pythonw.exe', 'Scripts', 'tcl', 'Tools',
'vcruntime140.dll']
```

Nota. Cuando indicamos una ruta como una cadena, por ejemplo, "C:/ruta/carpetas", en Windows se pueden usar tanto barras convencionales / como barras invertidas \. No obstante, puesto que la barra invertida se emplea para generar algunos caracteres especiales, como el de salto de línea \n, cuando queramos utilizarla como separador en una ruta deberemos indicar doble barra invertida \\. Así, lo correcto sería "C:\\ruta\\carpetas" y no "C:\ruta\carpetas".

Si omitimos el argumento, se toma por defecto la ruta desde donde se esté ejecutando el programa. Esta ruta lleva el nombre de *current working directory* (directorio actual de trabajo) y es retornada por la función `os.getcwd()`. Por ejemplo:

```
>>> import os
>>> os.getcwd()
'C:\Users\Usuario\Desktop'
```

De modo que las siguientes dos llamadas son equivalentes:

```
os.listdir(os.getcwd())
os.listdir()
```

La función `os.path.exists()` toma como argumento una ruta (sea una carpeta o un archivo) y retorna un booleano indicando si la misma existe (`True`) o no (`False`).

```
>>>os.path.exists("C:\\Users\\Anonimo\\AppData\\Local\\Programs\\Python\\Python39")
True

>>> os.path.exists("C:\\ruta\\inexistente.txt")
False
```

Mientras que `os.mkdir()` (***make directory***) toma como argumento la ruta hacia una carpeta y la crea.

¡Atención! las experiencias de borrar directorios o de borrar archivos hágase sobre archivos sin importancia y creados para estas experiencias. Ya que una vez borrados no terminan en la papelera. Son borrados definitivamente.

Por otro lado, `os.rmdir()` (***remove directory***) elimina una carpeta **vacía**, únicamente carpetas vacías. Para eliminar carpetas con contenido, más adelante veremos la función `shutil.rmtree()`.

```
>>> os.rmdir("c:\\ruta\\directorio")
```

Para remover archivos se utiliza `os.remove()`.

```
>>> os.remove("c:\\ruta\\archivo.ext")
```

La función `os.rename()` toma como primer argumento la ruta a un archivo cuyo nombre será cambiado por el segundo argumento.

```
>>> os.rename("c:\\ruta\\archivo.ext", "c:\\ruta\\nuevo_nombre.ext")
```

Si deseamos ejecutar alguna instrucción o comando del sistema operativo podemos usar `os.system()`

```
>>> os.system("cls") #cls comando borrar pantalla en SO Windows
```

Y por último `os`, también nos provee una manera de saber sobre que sistema operativo estamos ejecutando nuestro código.

```
>>> os.name # "nt" es Windows , "posix" es linux o iOS
```

Sistema de archivos, SHUTIL

Por otro lado, dentro del módulo `shutil` nos interesan las siguientes:

- `shutil.copy()`
- `shutil.move()`
- `shutil.rmtree()`

Vía `shutil.copy()` copiamos un archivo de una ubicación a otra, manteniendo el original.

```
>>> import shutil
```

```
>>> shutil.copy("C:\\ruta\\archivo.txt", "C:\\otra_ruta\\archivo.txt")
```

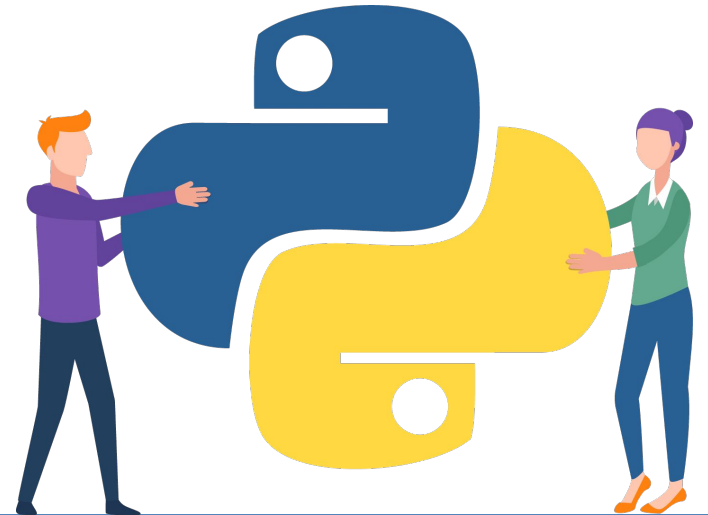
Del mismo modo opera `shutil.move()`, que mueve un archivo de una ubicación a otra, quitándolo de su ruta original.

```
>>> shutil.move("C:\\ruta\\archivo.txt",  
"C:\\otra_ruta\\archivo.txt")
```

Por último, para eliminar una carpeta que no esté vacía empleamos `shutil.rmtree()`:

¡Cuidado! Borra todos los archivos en la ruta especificada para siempre.

```
>>> shutil.rmtree("C:\\ruta\\carpeta_no_vacia")
```



¡Muchas gracias!

¡Sigamos trabajando!

Python Programming

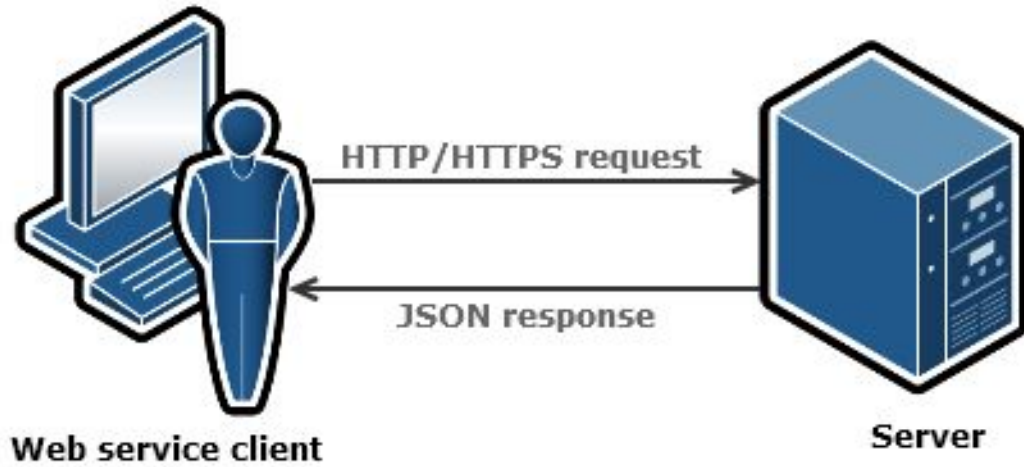
Módulo 05

¿Qué es un servicio web?

¿Qué es un servicio web?

Un servicio web es un programa que provee una funcionalidad determinada y con el cual nos comunicamos a través del protocolo HTTP (el mismo utilizado por los navegadores para interactuar con un sitio web). Esta funcionalidad puede ser desde un simple servicio que recibe dos números y retorna su producto hasta un complejo modelo de inteligencia artificial que recibe una imagen de un jardín y nos retorna los nombres de todas las especies de plantas que se muestran en ella.

Los ejemplos más comunes de un simple servicio web incluyen informes meteorológicos, cotizaciones de acciones y divisas, informes empresariales y gubernamentales, comunicación con redes sociales, etc. Cada servicio está identificado por una dirección de URL, podemos conectarnos a ella desde Python y podemos interactuar con la plataforma programáticamente como si lo estuviésemos haciendo manualmente desde alguna de las aplicaciones móviles o web. Otras plataformas como Facebook, twitter, Instagram, WhatsApp y MercadoLibre proveen servicios similares.



¡Sigamos trabajando!

Python Programming

Módulo 05

El protocolo HTTP

El protocolo HTTP

En programación, siempre que se realiza una conexión entre dos computadoras, al iniciador de dicha conexión se lo conoce como cliente; y a quien recibe y acepta la conexión, servidor. Así, cuando visitamos un sitio web cualquiera, el navegador web es el cliente y el sitio web es el servidor; o cuando nos conectamos a un servicio web desde un programa de Python, éste es el cliente y aquél el servidor.

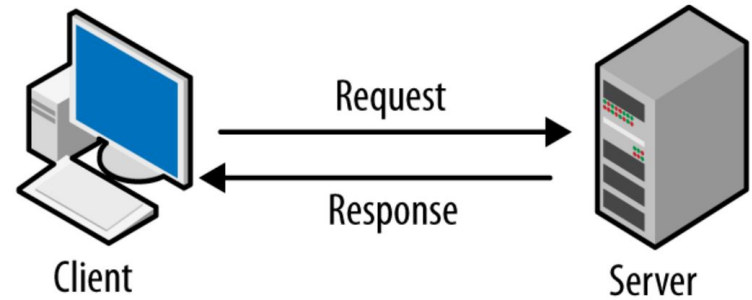
El **protocolo HTTP**, por su parte, está montado sobre dos conceptos principales: **petición** (*request*) y **respuesta** (*response*).

Luego de haberse establecido una conexión entre dos programas que se comunicarán vía HTTP, el cliente debe hacer una petición exigiéndole alguna operación al servidor, y éste le devolverá una respuesta.

Por ejemplo, cuando ingresamos a <https://www.educacionit.com/>, el navegador (*cliente*) realiza una petición al sitio web (*servidor*) indicándole que quiere ver la página principal, y éste le responde enviándole contenido correspondiente, que en este caso, por tratarse de un sitio web, será un código HTML.

Cuando interactuamos con un servicio web, los datos que éste envía como respuesta están codificados, por lo general, en formato JSON en lugar de HTML.

Pero en nuestro caso Python se encargará de realizar las conversiones automáticamente: de estructuras de Python a JSON cuando enviemos información desde nuestro programa al servicio web, y de JSON a estructuras de Python cuando recibamos una respuesta.



¡Sigamos trabajando!

Python Programming

Módulo 05

Arquitectura REST

Arquitectura REST

Dijimos que la comunicación entre un servicio web y un consumidor de ese servicio (en nuestro caso, una aplicación de Python) se realiza vía el protocolo HTTP. Pero para que las cosas sean más simples, muchos servicios web tienen una estructura similar llamada *REST*.

REST, *REpresentational State Transfer*, se trata de una arquitectura estándar para comunicaciones web entre sistemas, logrando que se entiendan mucho mejor entre ellos. A los servicios que cumplen con este diseño se les llama RESTful API.

La arquitectura REST se basa en que el cliente envía peticiones para recuperar o modificar recursos, y el servidor responde con el resultado, que puede ser con los datos que hemos pedido o el estado de la petición.

Una petición está formada por:

- Un verbo HTTP que define la operación a realizar.
- Una cabecera o header que incluye información sobre la petición.
- Una ruta o path hacia un recurso.
- El cuerpo del mensaje o body con los datos de la petición.

Veamos un ejemplo. Consideremos un servicio web ficcional a partir del cual se gestionan los alumnos del instituto, cuya dirección es `http://api.educacionit.com/`.

Utilizando este servicio podemos obtener la lista de alumnos, la información de alguno en particular e incluso agregar nuevos y modificar o eliminar alumnos existentes. Pero también podremos realizar operaciones similares con instructores y personal administrativo del instituto. Así, para poder distinguir qué tipo de información queremos obtener, crear, modificar o eliminar, el servicio proveerá direcciones de URL más específicas, tales como:

- `http://api.educacionit.com/alumnos`
- `http://api.educacionit.com/instructores`
- `http://api.educacionit.com/administrativos`

Cada una de estas “secciones” de un servicio web, en la terminología de la arquitectura REST, se denomina recurso. Así, quitando la primera parte del dominio para simplificar, tenemos tres recursos: `/alumnos`, `/instructores` y `/administrativos`. No obstante, por lo general, los recursos tienen nombre en singular (más adelante se verá más claramente por qué), de modo que serían, más bien, los siguientes:

- `http://api.educacionit.com/alumno`
- `http://api.educacionit.com/instructor`
- `http://api.educacionit.com/administrativo`

Ahora bien, el protocolo HTTP define un conjunto de verbos (con el nombre de *métodos*) para identificar qué tipo de operación se quiere ejecutar sobre un recurso determinado, a saber:

- GET, para leer información;
- POST, para agregar nueva;
- PUT, para modificar información preexistente;
- DELETE, para eliminar.

De modo que toda operación que ejecutemos sobre un servicio web estará constituida, por el momento, por un método (`GET`, `POST`, `PUT` o `DELETE`) y la dirección de URL de un recurso. Por ejemplo, si queremos obtener la lista de alumnos del servicio web en cuestión, la operación se enuncia de la siguiente forma:

```
GET /alumno
```

Si queremos agregar un nuevo alumno, de esta otra:

```
POST /alumno
```

Cuando queremos modificar los datos de un alumno o eliminarlo, debemos indicar, además, cuál es ese alumno. Por lo general, en un recurso, cada registro está identificado por un número. Así, si queremos eliminar el alumno cuyo identificador es el número 3, haremos:

```
DELETE /alumno/3
```

Y asimismo si queremos modificarlo:

```
PUT /alumno/3
```


También es posible indicar un identificar usando el método `GET`, en cuyo caso se retorna la información del alumno especificado en lugar de la lista completa.

```
GET /alumno/3
```

Estos son los conceptos básicos de la arquitectura REST. Con lo visto será suficiente para seguir avanzando.



¡Sigamos trabajando!

Python Programming

Módulo 05

Módulo requests

Módulo requests

Python provee en su librería estándar el módulo `urllib` para interactuar con direcciones de URL y el protocolo HTTP, esto es, lo necesario para comunicarse con un servicio web.

No obstante, la librería que ha adoptado la comunidad para este tipo de tareas es [requests](#), porque es más potente y fácil de usar. Entonces, en primer lugar, vamos a instalarla ejecutando en la consola del sistema operativo:

```
pip install requests
```

El primer request

Hacer una solicitud con Requests es muy simple. Podemos probarlo desde la consola interactiva o desde un script, lo primero es importar el módulo, observemos el siguiente código:

```
>>> import requests
>>> r = requests.get("https://google.com/")
>>> print(r.status_code)
```

La primera línea de código importa el paquete según hemos visto anteriormente. La segunda realiza una petición vía el método GET a la dirección de URL <https://google.com/>

El valor de retorno de la función `get()` es la respuesta del servidor que opcionalmente tendrá algún contenido (un código HTML en este caso por tratarse de un sitio web), pero obligatoriamente habrá un código de estado (*status code*) que indicará si la operación se ejecutó correctamente o no.

Si estamos conectados a internet, el código anterior debe imprimir lo siguiente.

```
200
```

Es decir, el código de estado (`r.status_code`) de la respuesta del servidor es 200. El valor 200 (y también los valores 201, 202 y 204) indica que la operación se ejecutó correctamente.

Veamos otro ejemplo, con un servicio web,

```
>>> import requests
>>> r =
requests.get("https://api.github.com/events"
)
>>> r.json()
```

Por lo general vamos a trabajar con servicios web, y lo típico es conseguir como devolución (*response*, en nuestro caso es "`r`") un JSON, como bien sabemos es el formato más utilizado para estas tareas.

El método `.json()` permite convertir la devolución a un formato conocido por nosotros en Python (diccionarios y listas), para que nos resulte fácil acceder a los datos.

¡Sigamos trabajando!

Python Programming

Módulo 05

Interacción con el servicio de prueba

Interacción con el servicio de prueba

Nuestro servicio de prueba corre en la dirección <http://localhost:7001> y expone el recurso (según lo que acabamos de decir sobre la arquitectura REST) <http://localhost:7001/student>.

Sabido esto, creamos un nuevo archivo de Python para interactuar con el servicio vía la librería `requests`.

Ahora bien, vayamos a lo que nos interesa que es interactuar con un servicio web.

El servicio que tenemos de prueba expone el siguiente recurso con la arquitectura REST:

<http://localhost:7001/student>

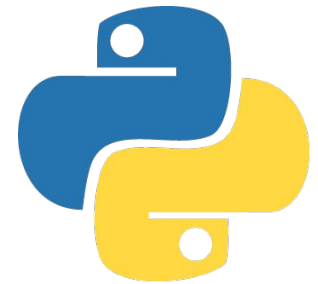
Por lo general, los recursos de los servicios web están escritos en inglés, así que podemos ir acostumbrándonos desde el comienzo. Este recurso permite obtener, agregar, modificar y eliminar alumnos (*students*). Cada alumno del recurso tiene un nombre (*name*) y una cantidad de cursos realizados (*courses*).

Intentemos hacer una petición GET a este recurso a ver cuál es el resultado.

```
import requests

r = requests.get("http://localhost:7001/student")
print("Código de estado:", r.status_code)
print("Contenido de la respuesta:", r.json())
```

Aquí lo que hemos agregado es una cuarta línea en donde se llama a `r.json()`, que lee el contenido de la respuesta del servidor en formato JSON y lo convierte a las estructuras de Python correspondientes.



El texto impreso en pantalla es el siguiente:

```
Código de estado: 200
```

```
Contenido de la respuesta: {'students': []}
```

Vemos que `r.status_code == 200`, por lo que la operación se ejecutó correctamente. Recordemos que el método `GET`, en la arquitectura REST, quiere decir “obtener una lista de los registros de un recurso”. Si nuestro recurso es `/student`, la operación `GET /student` debe responder con una lista de alumnos. La respuesta la obtenemos vía `r.json()`, como vimos recién.

Para trabajar mejor con ella, asignémosla a una variable:

```
r = requests.get("http://localhost:7001/student")
print("Código de estado:", r.status_code)
respuesta = r.json()
print("Contenido de la respuesta:", respuesta)
```

Según lo impreso en pantalla:

```
respuesta == {'students': []}
```

es decir, es un diccionario con un único elemento, cuya clave es "students" y su valor, una lista vacía. Lo cual quiere decir que aún no hay alumnos cargados en el recurso /student. Pero por el momento ya sabemos que, luego de hacer esta petición, la lista de alumnos estará en `respuesta["students"]`.

Previamente vimos que el método HTTP para agregar información a un recurso es `POST`.

En el caso de nuestro servicio de prueba, lo estaremos usando para insertar nuevos alumnos. Ya dijimos que cada alumno tiene un nombre y una cantidad de cursos. Entonces el código para insertar un nuevo alumno se vería más o menos así:

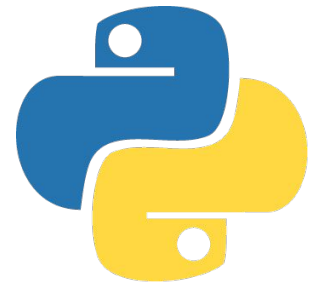
```
r = requests.post("http://localhost:7001/student",
                  json={"name": "Lautaro", "courses": 3})

print("Código de estado:", r.status_code)
print("Contenido de la respuesta:", r.json())
```

Esto imprime:

```
Código de estado: 201
Contenido de la respuesta: {'id': 0}
```

¡Perfecto! Dijimos que los códigos de estado 200, 201, 202 y 204 indican que la operación se ejecutó correctamente. En los servicios con arquitectura REST, el código 201 indica que se insertó satisfactoriamente un nuevo registro a un recurso. También tenemos una respuesta, el diccionario {"id": 0} que nos indica que el servicio le asignó el número 0 al alumno que acabamos de crear. Este identificador nos será útil luego para hacer modificaciones sobre ese alumno e incluso eliminarlo.



Agreguemos algunos alumnos más para tener más de un registro con el cual seguir trabajando:

```
alumnos = (  
    ("Juan", 1),  
    ("Sofia", 5),  
    ("Martin", 2)  
)  
  
for nombre, cursos in alumnos:  
    r = requests.post("http://localhost:7001/student",  
                      json={"name": nombre, "courses":  
cursos})  
    print("Código de estado:", r.status_code)  
    print("Contenido de la respuesta:", r.json())
```

Si todo salió bien, al ejecutarse esto debe imprimir:

```
Código de estado: 201  
Contenido de la respuesta: {'id': 1}  
Código de estado: 201  
Contenido de la respuesta: {'id': 2}  
Código de estado: 201  
Contenido de la respuesta: {'id': 3}
```



Ahora que tenemos algunos alumnos, si volvemos a ejecutar nuestro primer código...

```
r = requests.get("http://localhost:7001/student")
print("Código de estado:", r.status_code)
respuesta = r.json()
print("Contenido de la respuesta:", respuesta)
```

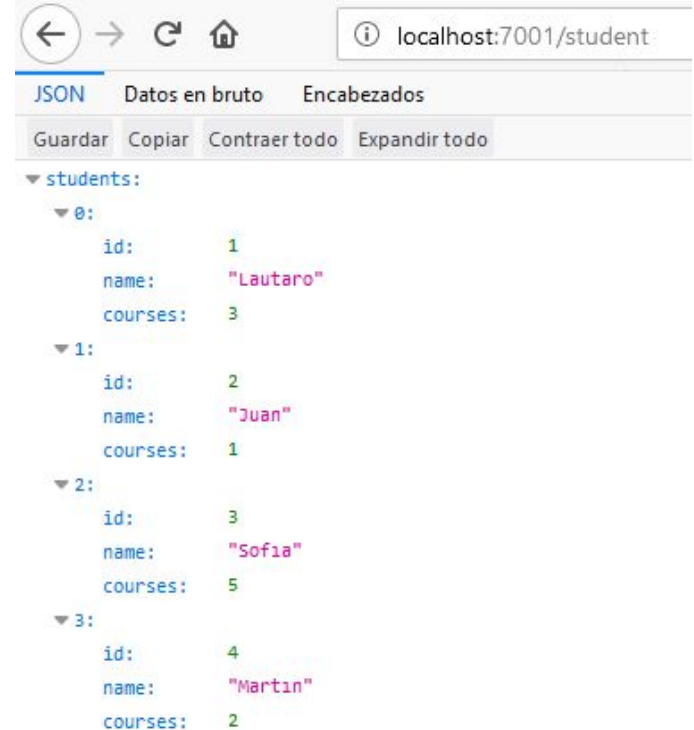
...obtendremos lo siguiente:

```
Código de estado: 200
Contenido de la respuesta: {'students': [{'id': 0,
'nombre': 'Lautaro', 'cursos': 3}, {'id': 1, 'nombre':
'Juan', 'cursos': 1}, {'id': 2, 'nombre': 'Sofia',
'cursos': 5}, {'id': 3, 'nombre': 'Martin', 'cursos': 2}]}
```

Ahora el contenido de `respuesta["students"]` ya no es una lista vacía. Cada uno de sus elementos representa a un alumno como un diccionario. Cada alumno tiene tres claves: un identificador numérico (`"id"`), un nombre (`"name"`) y una cantidad de cursos (`"courses"`). Podemos recorrer cada uno de los alumnos usando un bucle `"for"` así:

```
r = requests.get("http://localhost:7001/student")
if r.status_code == 200:
    respuesta = r.json()
    for alumno in respuesta["students"]:
        print("Alumno", alumno["id"])
        print("Nombre:", alumno["name"])
        print("Cursos:", alumno["courses"])
else:
    print("Ocurrió un error.")
```

Si ingresamos a <http://localhost:7001/student> desde el navegador también será capaz de mostrarnos la lista de alumnos, puesto que cada vez que visitamos una dirección de URL desde un navegador web se realiza una petición del tipo GET.



Ahora bien, usando GET también podemos obtener la información de algún alumno en particular indicando su identificador numérico. Por ejemplo, GET /student/3 debe retornar únicamente la información de Sofía. Comprobémoslo:

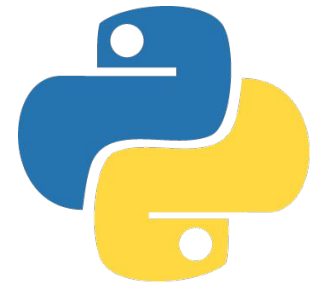
```
r = requests.get("http://localhost:7001/student/3")
print("Código de estado:", r.status_code)
print("Contenido de la respuesta:", r.json())
```

En efecto, el resultado es:

```
Código de estado: 200
Contenido de la respuesta: {'student': {'name':
'Sofia', 'courses': 5}}
```

Nótese que en este caso la información del alumno está dentro de la clave "student", por lo que habremos de acceder a ella más o menos así:

```
r = requests.get("http://localhost:7001/student/3")
if r.status_code == 200:
    alumno = r.json()["student"]
    print("Nombre:", alumno["name"])
    print("Cursos:", alumno["courses"])
else:
    print("Ocurrió un error.")
```



¿Cómo proceder si queremos modificar algún dato (nombre o cantidad de cursos) del alumno 3 (Sofía)? Según lo que vimos más arriba, debemos usar el método `PUT`. Esto en Python se traduce como la función `requests.put()`, y vamos a pasar como argumento un diccionario con las claves que queremos modificar y sus respectivos valores, que luego serán convertidas a formato JSON automáticamente por la librería. El siguiente código cambia la cantidad de cursos de Sofía a 6:

```
datos = {"courses": 6}
r = requests.put("http://localhost:7001/student/3",
json=datos)
print("Código de estado:", r.status_code)
```

El resultado en pantalla es:

```
Código de estado: 204
```


El código de estado 204 indica que la petición se ejecutó correctamente, pero a diferencia de los estados anteriores, este no retorna ningún contenido. Por eso no llamamos a `r.json()`.

Si la petición fallara por alguna razón, su código de estado sería otro que 204. Por ejemplo:

```
datos = {"courses": 6}
r = requests.put("http://localhost:7001/student/10",
json=datos)
print("Código de estado:", r.status_code)
```

Este código imprime:

```
Código de estado: 404
```

El código de estado 404 indica que no se encontró el recurso sobre el cual se quiere ejecutar la operación. En este caso, efectivamente, no hay ningún alumno cuyo identificador sea el número 10.

Al invocar la función `put()`, es posible modificar varias claves simultáneamente:

```
# Modifica el nombre y la cantidad de cursos del alumno 3.
datos = {"name": "Josefina", "courses": 6}
r = requests.put("http://localhost:7001/student/3",
json=datos)
print("Código de estado:", r.status_code)
```

Por último, para eliminar un registro en un recurso usamos el método `DELETE` del protocolo HTTP: esto es, en Python, `requests.delete()`.

```
# Elimina el alumno 3.
r = requests.delete("http://localhost:7001/student/3")
print("Código de estado:", r.status_code)
```

El código de estado es el mismo que para las operaciones `PUT`:

Código de estado: 204

¡Sigamos trabajando!

Python Programming

Módulo 05

Automatizar envío de formularios

Automatizar envío de formulario HTML vía HTTP

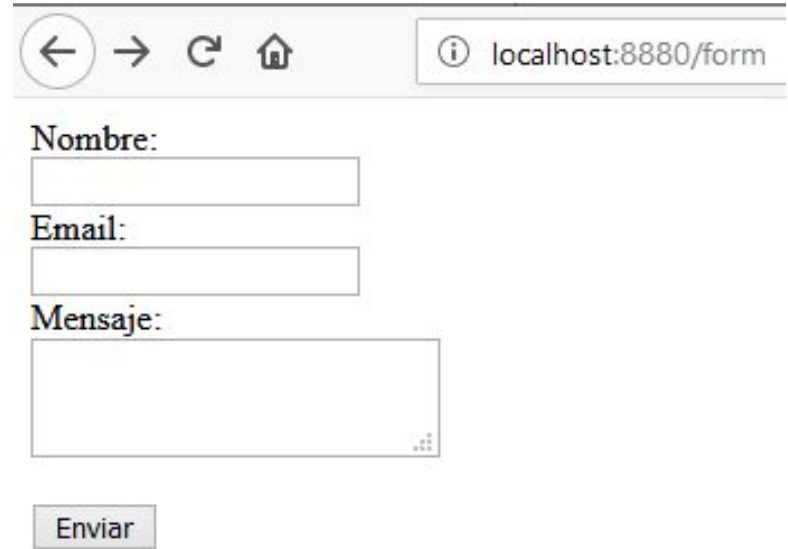
Los formularios son la principal forma de comunicación entre un usuario y un sitio web para enviar información. Por ejemplo, formularios de contacto o para iniciar sesión. Enviar información a través de un formulario usando Python es una operación muy común con el objetivo de automatizar tareas. La misma librería con la que hemos estado trabajando hasta ahora, `requests`, nos permite efectuar dicha operación. De hecho, el método HTTP que usan los formularios HTML para enviar información es `POST`.

Lo primero que necesitamos es un sitio con un formulario. Para eso nuevamente vamos a usar una aplicación creada para este ejemplo. La encuentras para descargar en el Alumni en esta misma sección con el nombre *formulario_web*.

Descargas `formulario_web.py` y guardas en una carpeta. Luego le haces doble click encima de `formulario_web.py`, se abrirá una terminal (*servidor*) muy parecido a lo que hicimos en la sección anterior con el servicio de alumnos.

(Solo en Linux o en iOS ejecutarlo desde la consola con `python3`).

Este pequeño sitio de ejemplo expone el siguiente formulario de contacto en <http://localhost:8880/form>.



The screenshot shows a web browser window with the address bar displaying `localhost:8880/form`. The page contains a contact form with the following elements:

- Nombre:** A text input field.
- Email:** A text input field.
- Mensaje:** A large text area for the message.
- Enviar**: A button to submit the form.

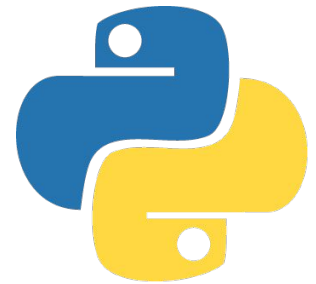
Para automatizar el envío, lo primero que tenemos que identificar es la dirección de URL. Luego, acabamos de decir que la información de un formulario HTML en un sitio web se envía a través del protocolo HTTP usando el método `POST`; entonces, nuestro código de Python comenzará siendo el siguiente:

```
import requests

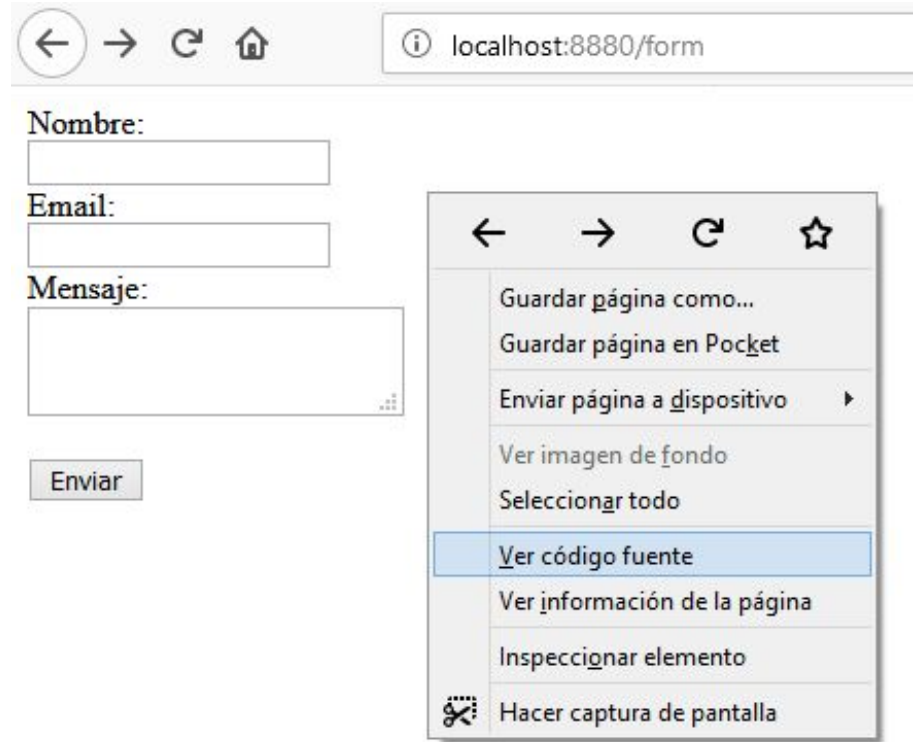
r =
requests.post("http://localhost:8880/form")
```

El siguiente paso es pasar como argumento a `post()` los datos que queremos enviar: el nombre, el correo electrónico y el mensaje mismo. Estos datos tienen que estar contenidos como valores de un diccionario, ¿pero cuáles serían las claves?

Para ello necesitamos ver el código de fuente del formulario. Todos los navegadores contienen una opción para ello en el menú contextual:



Para ello necesitamos ver el código de fuente del formulario. Todos los navegadores contienen una opción para ello en el menú contextual:



Se abrirá una nueva ventana y veremos el siguiente código:

```
<!DOCTYPE html>
<html>
<head><meta charset="utf-8"></head>
<body>
  <form method="post">
    Nombre:<br>
    <input type="text" name="name">
    <br>
    Email:<br>
    <input type="text" name="email">
    <br>
    Mensaje:<br>
    <textarea name="message"></textarea>
    <br><br>
    <input type="submit" value="Enviar">
  </form>
</body>
</html>
```

El fragmento de código que configura el formulario HTML está entre las etiquetas `<form>` y `</form>`. Lo que nos interesa dentro de él son los nombres que el sitio le ha asignado a los componentes del formulario (esto es, las cajas de texto para ingresar datos) vía el atributo `name`. Los marcamos en rojo a continuación:

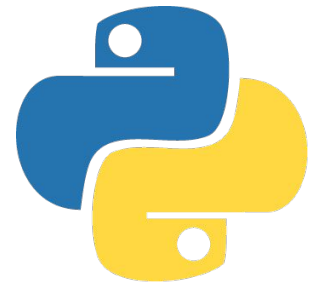
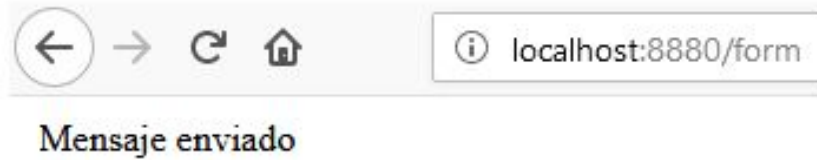
```
<!DOCTYPE html>
<html>
<head><meta charset="utf-8"></head>
<body>
  <form method="post">
    Nombre:<br>
    <input type="text" name="name">
    <br>
    Email:<br>
    <input type="text" name="email">
    <br>
    Mensaje:<br>
    <textarea name="message"></textarea>
    <br><br>
    <input type="submit" value="Enviar">
  </form>
</body>
</html>
```

Vemos que los nombres para los campos de nombre, correo electrónico y mensaje son, respectivamente, `name`, `email` y `message`. Pues bien, estos corresponderán, entonces, a las clave de nuestro diccionario de datos:

```
datos = {  
    "name": "Mariano",  
    "email": "mariano@ejemplo.com",  
    "message": "¡Hola, mundo!"  
}  
r = requests.post("http://localhost:8880/form", data=datos)
```

(Aquí el argumento para `post()` es `data` y no `json`, a diferencia del apartado anterior, ya que al enviar datos en un formulario raramente se codifican en formato JSON).

¡Perfecto! Ahora, ¿cómo sabemos que el formulario se ha enviado correctamente? Primero intentemos enviarlo manualmente usando el navegador. Luego de completar los datos veremos que la página responde con lo siguiente:



Podemos acceder a la respuesta del sitio desde Python vía `r.text`:

```
datos = {  
    "name": "Mariano",  
    "email": "mariano@ejemplo.com",  
    "message": "¡Hola, mundo!"  
}  
  
r = requests.post("http://localhost:8880/form", data=datos)  
contenido = r.text  
print(contenido)
```

Esto imprime:

```
<!doctype html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Enviado</title>
    <meta http-equiv="refresh" content="3";URL="http://localhost:8880/form">
  </head>
  <body>
    Mensaje enviado
  </body>
</html>
```

Eso quiere decir que podemos hacer una comprobación desde nuestro código para determinar si el formulario se envió correctamente, igualando `contenido` a la cadena anterior. Pero dado que el resultado en HTML suele ser muy voluminoso, simplemente podemos buscar alguna parte de la cadena, por ejemplo, "Mensaje enviado". Para ello usamos el operador `in`, como vimos en las colecciones:

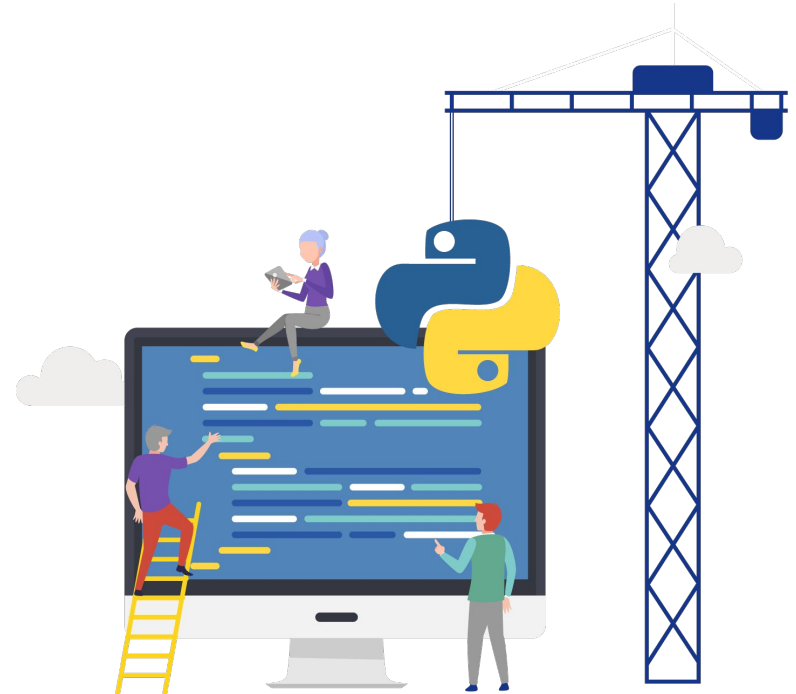
```
datos = {
    "name": "Mariano",
    "email": "mariano@ejemplo.com",
    "message": "¡Hola, mundo!"
}
r = requests.post("http://localhost:8880/form",
data=datos)

contenido = r.text

if "Mensaje enviado" in contenido:
    print("¡Formulario enviado!")
else:
    print("Ocurrió un error.")
```


O bien usando la función `find()` según lo visto en la sección de operaciones sobre cadenas:

```
if contenido.find("Mensaje enviado") > -1:  
    print(";Formulario enviado!")  
else:  
    print("Ocurrió un error.")
```



¡Sigamos trabajando!