

Python / Guía 10

Más funcionalidades en Python

## 

## 

## OBJETIVOS DE LA GUÍA

En esta guía aprenderemos a:

* Manejar archivos
* Manejar fechas y horas
* Leer, escribir y convertir datos del formato JSON
* Implementar decoradores
* Implementar generadores

# 

# Archivos

Con Python podemos:

* Crear archivos
* Escribir datos en un archivo
* Leer datos de un archivo
* Borrar archivos

La clase incorporada en Python para el manejo de archivos es:

open(ruta, modo)

Donde ruta es el primer argumento por posición, una cadena donde debes indicar el nombre del archivo, con la posibilidad de indicar la ruta. El segundo argumento, modo, es cómo va a abrirse tal archivo.

## 

## Creación de archivos

Primero, crea una instancia del objeto **open:**

archivo = open(nombre, "w")

"archivo" es la instancia de open. **El argumento "nombre" es una cadena** que contiene el nombre del archivo (puedes incluir también la ruta o path). El argumento "w" significa write, hay otros argumentos que puedes usar como modo de apertura del archivo.

Con el modo "w" abres el archivo en modo escritura. Si el archivo existe, se reemplazará por el nuevo, si no, creará uno nuevo.

Cuando termines de trabajar con un archivo, ciérralo, sobre todo si lo abres en modo escritura, ya que mientras está abierto no se puede abrir por otra aplicación. Si no se cierra explícitamente, Python intentará cerrarlo cuando estime que ya no se va a usar más.

Para cerrar un archivo referenciado por el objeto utiliza el siguiente método:

archivo.close()

## Escritura de datos en un archivo

Usa el método **write:**

archivo.write(cadena)

"Cadena" es una cadena de caracteres que se van a escribir en el archivo. El método write devuelve (en el return) la cantidad de caracteres que se escribieron.

## 

## Adición de datos a un archivo

Si en vez de crear un archivo, quieres agregar datos a un archivo existente, debes utilizar como argumento "a" (append):

archivo = open(ruta, "a")

Una vez abierto el fichero, utiliza el método **write** para agregar caracteres al final del fichero.

## 

## Lectura de datos de un archivo

Usa el argumento "r" (read) cuando instancies open.

archivo = **open(nombre, "r")**

Tienes disponible los siguientes métodos:

archivo**.read()**

Read() devuelve (en su return) toda la cadena de caracteres contenida en el archivo. Puedes guardarla en una variable y verla con print.

archivo**.readlines()**

Readlines() devuelve (en su return) una lista de cadenas de caracteres, y cada elemento de la lista es una cadena, que representa una línea del archivo. Puedes iterar la lista.

## With

With es una palabra reservada de Python que sirve para "administrar contextos" (context managers) Junto con "open", te puedes despreocupar del cierre del archivo, pues ya no sería necesario cerrarlo explícitamente. Se utiliza la siguiente estructura:

**with open**(nombre, modo) **as** archivo:  
 <bloque with>

El archivo permanece abierto mientras se ejecuta el bloque de código, y se cierra automáticamente cuando termina la ejecución del bloque.

## 

## Manejo de archivos

### Renombrar archivos

Para renombrar o borrar un archivo importa el módulo **os**, incorporado en Python

os.rename(ruta1, ruta2): Renombra y mueve el archivo de la ruta "ruta1" a la ruta "ruta2".

### Eliminar archivos

os.remove(ruta): Borra el archivo de la ruta.

Antes de borrar o renombrar un directorio, conviene comprobar si existe, para que no lance un error. Usa lo siguiente:

os.path.isfile(ruta): Devuelve True, si existe un archivo en la ruta.

### Directorios / carpetas

os.listdir(ruta): Devuelve una lista con los ficheros y directorios contenidos en la ruta.

os.mkdir(ruta): Crea un nuevo directorio en la ruta.

os.chdir(ruta): Cambia el directorio actual al indicado por la ruta.

os.getcwd(): Devuelve una cadena con la ruta del directorio actual.

os.rmdir(ruta): Borra el directorio de la ruta, siempre y cuando esté vacío.

**¡MANOS A LA OBRA!**

**Ejercicio 1**

* Guarda en una variable el nombre de un archivo: "mi\_ruta.txt"
* Crea un archivo nuevo y guarda la siguiente línea en el archivo:   
  "Esta es mi ruta:"
* Agrega al final del archivo una línea que contenga la ruta del directorio actual.
* Agrega una línea más con el nombre del archivo, utilizando la variable.
* Lee el archivo con readlines y muestra su contenido enumerando las líneas empleando la función enumerate(). La salida debe tener 3 líneas en total.

| Con los archivos, siempre conviene trabajar con with... |
| --- |

|  | **Ver video “**[**Manejo de Archivos**](https://youtu.be/Yrd47QbKdVE)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Escribir y leer archivos con open.
* Utilizar with con open
* Utilizar algunos métodos del módulo "os" para el manejo de carpetas y archivos.

# El tiempo con python

Para manejar fechas y horas, Python tiene incorporado el módulo **datetime,** que administra los tipos de datos **date, time** y **datetime** para representar fechas y funciones para manejarlas. Algunas de las operaciones más habituales que permite son:

* Acceder a los distintos componentes de una fecha (año, mes, día, hora, minutos, segundos y microsegundos).
* Convertir cadenas con formato de fecha en los tipos date, time o datetime.
* Convertir fechas de los tipos date, time o datetime en cadenas formateadas de acuerdo a diferentes formatos de fechas.
* Hacer operaciones aritméticas de fechas (sumar o restar fechas).
* Comparar fechas.

## 

## Date, time, datetime

date(año, mes, día): Devuelve un objeto de tipo date que representa la fecha con el año, mes y día indicados.

time(hora, minutos, segundos, microsegundos): Devuelve un objeto de tipo time que representa la hora, minutos, segundos y microsegundos indicados.

datetime(año, mes, día, hora, minutos, segundos, microsegundos): Devuelve un objeto de tipo datetime que representa una fecha y hora con el año, mes, día, hora, minutos, segundos y microsegundos indicados.

**En el intérprete de Python, ejecuta las siguientes líneas:**

from datetime import date, time, datetime

date(2023, 01, 01)

datetime.date(2025, 12, 25)

time(14, 30, 1)

datetime.time(14, 30, 1)

datetime(2023, 12, 31, 23, 59, 59)

datetime.datetime(1914, 8, 20, 14, 30, 1)

print(datetime(1991, 3, 25, 14, 30, 1))

## 

## Miembros de datetime

datetime.today(): Devuelve un objeto del tipo date la fecha del sistema en el momento en el que se ejecuta.

datetime.now(): Devuelve un objeto del tipo datetime con la fecha y la hora del sistema en el momento exacto en el que se ejecuta.

Puede crear una instancia para usar fácilmente los atributos, supongamos que instanciamos dt.

dt.year: Devuelve el año de la fecha dt, puede ser del tipo date o datetime.

dt.month: Devuelve el mes de la fecha dt, que puede ser del tipo date o datetime.

dt.day: Devuelve el día de la fecha dt , que puede ser del tipo date o datetime.

dt.weekday(: Devuelve el día de la semana de la fecha dt, que puede ser puede ser del tipo date o datetime.

dt.hour: Devuelve las horas del tiempo dt, que puede ser del tipo time o datetime.

dt.minute: Devuelve los minutos del tiempo dt, que puede ser del tipo time o datetime.

dt.second: Devuelve los segundos del tiempo dt, que puede ser del tipo time o datetime.

dt.microsecond: Devuelve los microsegundos del tiempo dt, que puede ser del tipo time o datetime.

**En el intérprete de Python, ejecuta las siguientes líneas:**

from datetime import datetime

print(datetime.today())

dt = datetime.now()

dt.year

dt.month

dt.day

dt.hour

dt.minute

dt.second

dt.microsecond

## 

## Conversión de fechas en cadenas

dt.strftime(formato): Devuelve la cadena que resulta de transformar la fecha dt hacia el formato indicado en la cadena "formato".

La cadena formato puede contener los siguientes marcadores de posición:

%Y (año completo)

%y (últimos dos dígitos del año)

%m (mes en número)

%B (mes en palabra)

%d (día)

%A (día de la semana)

%a (día de la semana abreviado)

%H (hora en formato 24 horas)

%I (hora en formato 12 horas)

%M (minutos)

%S (segundos)

%S.%f (segundos y microsegundos)

%p (AM o PM)

%c (fecha y hora completas)

%x (fecha completa)

%X (hora completa)

Para que veas las cadenas de texto de fechas en el idioma propio de tu sistema operativo, usa el módulo integrado **locale** y establece la configuración que tenga tu sistema:

**En el intérprete de Python, ejecuta las siguientes líneas:**

import locale

locale.setlocale(locale.LC\_ALL, "")

# Argentina:

locale.setlocale(locale.LC\_ALL, "es\_AR")

**En el intérprete de Python, ejecuta las siguientes líneas:**

from datetime import datetime

dt = datetime.now()

print(dt.strftime("%d/%m/%Y"))

print(dt.strftime("%A %d de %B de %Y"))

print(dt.strftime("%H:%M:%S"))

print(dt.strftime("%H horas, %M minutos y %S segundos"))

print(f"Hoy es {datetime.today().strftime('%D')}")

## 

## Conversión de cadenas en fechas

strptime(cadena, formato): Devuelve el objeto de tipo date, time o datetime que resulta de convertir "cadena" según el formato indicado en "formato''..

La cadena "formato" puede contener los siguientes marcadores de posición:

%Y (año completo)

%y (últimos dos dígitos del año)

%m (mes en número)

%B (mes en palabra)

%d (día)

%A (día de la semana)

%a (día de la semana abreviado)

%H (hora en formato 24 horas)

%I (hora en formato 12 horas)

%M (minutos)

%S (segundos)

%p (AM o PM)

%C (fecha y hora completas)

%x (fecha completa)

%X (hora completa)

**En el intérprete de Python, ejecuta las siguientes líneas:**

una\_fecha = "12/10/1492"  
 datetime.strptime(una\_fecha, "%d/%m/%Y")  
 fecha\_con\_guion = "01/01-2000"  
 datetime.strptime(fecha\_con\_guion, "%d/%m/%Y")  
 datetime.strptime(fecha\_con\_guion, "%d/%m-%Y")  
 fecha = datetime.strptime("2020-4-15 20:50:30", "%Y-%m-%d %H:%M:%S")  
 fecha.strftime("%c")

## Aritmética de fechas y horas

Para representar el tiempo transcurrido entre dos fechas se utiliza el submódulo **timedelta** de datetime. Para usarlo, debes importarlo:

from datetime import timedelta

timedelta(días, segundos, microsegundos): Devuelve un objeto del tipo timedelta que representa un intervalo de tiempo con los días, segundos y microsegundos.

datetime\_1 - datetime\_2: Devuelve un objeto del tipo timedelta que representa el tiempo transcurrido entre las fechas de tipo datetime.

datetime + timedelta:Devuelve la fecha del tipo datetime que resulta de sumar la fecha datetime con el tiempo timedelta.

**En el intérprete de Python, ejecuta las siguientes líneas:**

from datetime import datetime, timedelta

mi\_nacimiento = datetime(1981, 1, 15)

mi\_nacimiento + timedelta(31, 3600)

datetime.now() - mi\_nacimiento

mi\_nacimiento > datetime.now()

mi\_edad = datetime.now() - mi\_nacimiento

mi\_edad.days

**¡MANOS A LA OBRA!**

**Ejercicio 2**

Con dataclasses crea una clase Cliente que contenga los siguientes atributos:

dni: str  
 apellidos: str  
 nombres: str  
 nacimiento: datetime

Define la función cliente\_ingreso, que pida al usuario que ingrese los datos de un cliente. La función deberá instanciar la clase Cliente y la devolverá en el return. El formato de nacimiento será, por ejemplo: 01/01/2000, y devolverá un error si no corresponde. También, no dejará agregar un nacimiento futuro.

Define la función cliente\_guardar, que convierte la instancia en un diccionario con asdict de dataclasses. Guarda en un archivo los valores separados por una coma, en una sola línea por Cliente. Como último dato, en la misma línea, deberás agregar "creado: sábado 30/septiembre/2022 15:00:00"

Los datos deben guardarse en español. Debes pedir al usuario guardar 3 clientes.

|  | **Ver video “**[**Fechas**](https://youtu.be/2181LLwBQ1w)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Conocer el tipo de datos datetime, date, y time y algunos de sus componentes
* Ver las cadenas de fechas y horas según un idioma con el módulo locale
* Convertir fechas y horas en cadenas con strftime
* Convertir cadenas en fechas y horas con strptime
* Operar aritméticamente el tiempo con timedelta

# 

# El formato JSON

## ¿Qué es JSON?

El formato JSON fue inspirado por la sintaxis de JavaScript (un lenguaje de programación usado para desarrollo web). Se ha convertido en un formato de datos independiente del lenguaje de programación. La mayoría de los lenguajes de programación que usamos hoy en día pueden generar y leer JSON.

JSON es un formato usado para almacenar o representar datos. Sus usos más frecuentes incluyen desarrollo web y creación de archivos de configuración. JSON se usa comúnmente en aplicaciones web para enviar información desde el servidor al cliente o desde el cliente al servidor.

El formato JSON es bastante similar a la sintaxis de Python, pues contiene únicamente dos tipos de estructuras: diccionarios y listas. Diccionario es lo que en Python conocemos como tal, un conjunto de pares con una clave y un valor. Y una lista no es más que un conjunto ordenado de objetos.

Ejemplo:

{  
 "nombre":{  
 "nombres":"Regina",  
 "apellidos":null  
 },  
 "edad":16,  
 "correo":"regina@correo.org",  
 "cursos":[  
 "Python",  
 "SQLModel",  
 "FastAPI"  
 ],  
 "activo":true  
}

Como podemos observar, la estructura se conforma mediante pares clave-valor. Cada llave puede contener diferentes tipos de datos: cadenas, enteros, flotantes, booleanos, listas. Los elementos se separan con la coma.

Cada una de estas características también describen a los diccionarios en Python. La diferencia más entre un objeto JSON y un diccionario son las siguientes: las llaves para un objeto JSON deberán ser cadenas. Además, los valores booleanos en JSON se escriben con minúsculas. El objeto None se transforma en null.

## 

## Convertir JSON a diccionarios

Usaremos el módulo incorporado json:

import json

Si tenemos una cadena en una variable, podemos convertirla con el método loads.

json.loads(str)

**En el intérprete de Python, ejecuta las siguientes líneas:**

import json

texto = """  
 {  
 "nombre":{  
 "nombres":"Regina",  
 "apellidos":null  
 },  
 "edad":16,  
 "correo":"regina@correo.org",  
 "cursos":[  
 "Python",  
 "SQLModel",  
 "FastAPI"  
 ],  
 "activo":true  
 }"""

mi\_diccionario = json.loads(texto)  
 print(mi\_diccionario)  
 print(type(mi\_diccionario))

## 

## Convertir objetos Python a JSON

Convertir un objeto Python a otro tipo se llama serializar. Usaremos el método dumps:

json.dumps(objeto)

El método recibe como parámetro cualquier tipo de objeto serializable. También puede recibir como parámetro opcional el argumento por nombre **indent:** int. Sirve para indentar con espacios cada nivel de datos de json. Para ordenar las claves de forma alfabética, establece el parámetro opcional **sort\_keys:** bool

**Continuando con el estado anterior del intérprete de Python, ejecuta:**

nuevo\_json = json.**dumps**(mi\_diccionario)  
 print(nuevo\_json)  
 type(nuevo\_json)  
 nuevo\_json = json.**dumps**(mi\_diccionario, indent=4)  
 print(nuevo\_json)

## 

## ¿Cómo escribir archivos json?

Para escribir en un archivo JSON existente, o para crear un nuevo archivo JSON, usa el método dump:

json.dump(dict, archivo)

Observa la diferencia: json**.dumps** recibe como parámetro un diccionario y lo convierte en una cadena (lo serializa), pero json**.dump** recibe dos parámetros, un diccionario y una instancia de open.

La sintaxis anterior vuelca el diccionario "dict" en el archivo. Conviene que la extensión del nombre del archivo sea .json.

| Utiliza el parámetro **default=str** si quieres convertir, por ejemplo, un objeto datetime es una cadena, u obtendrás un error. |
| --- |

**En el intérprete de Python, ejecuta las siguientes líneas:**

import json

diccionario = {'nombre': {'nombres': 'Regina', 'apellidos': None}, 'edad': 16, 'correo': 'maria@correo.org', 'cursos': ['Python', 'SQLModel', 'FastAPI'], 'activo': True}

with open("estudiantes.json", "w") as archivo:  
 json.dump(diccionario, archivo)

## ¿Cómo leer archivos JSON?

Usa el método load:

json.load(archivo)

Observa la diferencia: json**.loads** leer una cadena, pero json**.load** lee un objeto de la instancia open.

**Habiendo creado estudiantes.json, ejecuta:**

with open("estudiantes.json", "r") as archivo:   
 datos = json.load(archivo)   
 print(datos)

**¡MANOS A LA OBRA!**

**Ejercicio 3**

Como continuación del ejercicio 2, ya teniendo la clase Cliente, la función cliente\_crear y la función cliente\_guardar: en vez de guardar los datos del cliente separados por comas, crea un archivo JSON sin las líneas que se indican en el ejercicio 2, es decir, el contenido del archivo debe ser JSON puro, con una indentación de 4 espacios. Prueba EJECUTAR tu programa como mínimo 2 veces. El archivo con extensión JSON no debería mostrar ningún subrayado rojo de parte de Visual Studio Code, esto significa que has creado bien el archivo JSON.

|  | **Ver video “**[**JSON**](https://youtu.be/uDi1QZSe2gY)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Conocer el formato json y su importancia
* Convertir json a diccionarios con json.loads
* Convertir (o serializar) objetos Python a formato json con json.dumps
* Escribir archivos json con json.dump
* Leer archivos json con json.load

# Decoradores

Un decorador es un patrón de diseño de software que se utiliza para alterar el funcionamiento de una determinada pieza de código, ya sea una función, o una clase, sin necesidad de modificar su código explícitamente.

A través de los decoradores seremos capaces de reducir las líneas de código duplicadas, haremos que nuestro código sea legible, fácil de testear, y fácil de mantener.

Un decorador es una función que recibe como parámetro una función y devuelve otra función.

Es importante mencionar que en Python las funciones pueden ser asignadas a una variable, pueden ser empleadas como argumento para otra función, o inclusive puede ser retornadas. Veamos un ejemplo:

**Vamos probando...**

def saludar():   
 print("Hola, soy una función.")

def super\_función(función):   
 función()

función = saludar  
super\_función(función)

¿Y cómo crear una función que reciba una función para devolver una función?: Veamos un ejemplo de como crear un decorador:

**Crea tu primer decorador:**

def función\_a(función\_b):  
 def función\_c():  
 print("Antes de la ejecución de la función a decorar")  
 función\_b()  
 print("Después de la ejecución de la función a decorar")  
 return función\_c

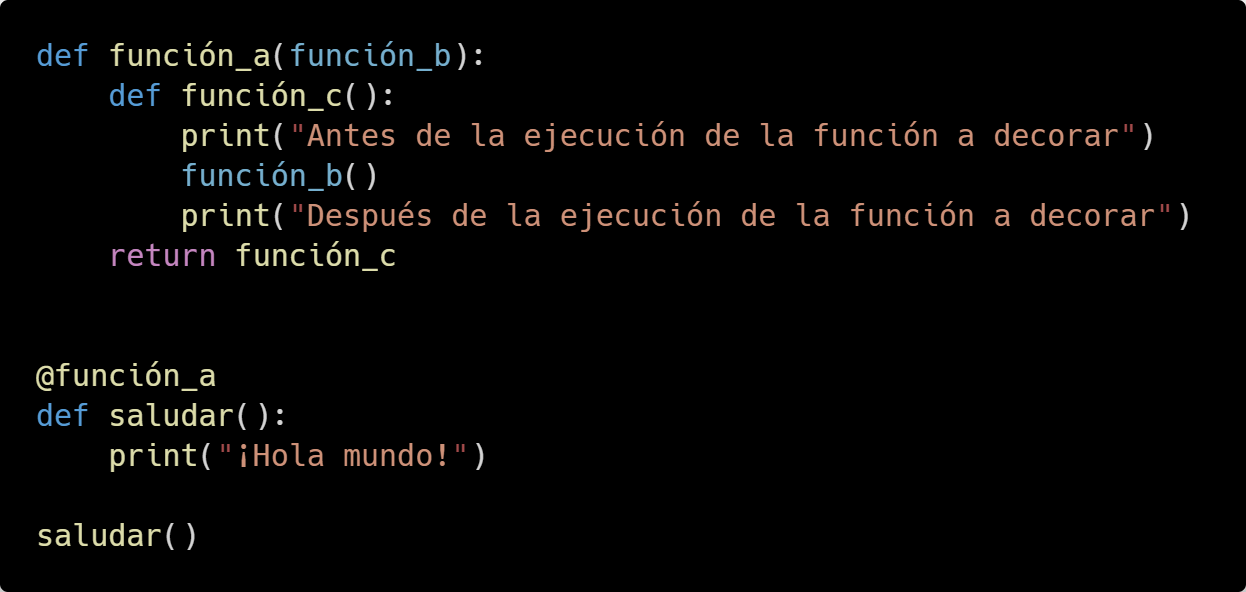
Ya tenemos el decorador creado, ahora lo que nos hace falta es decorar una función. Cuando decimos "decorar" nos referimos a querer modificar el comportamiento de una función ya existente, pero sin tener que modificar el código de la función ya existente.

Para decorar una función basta con colocar, en su parte superior de dicha función, el decorador con el prefijo @.

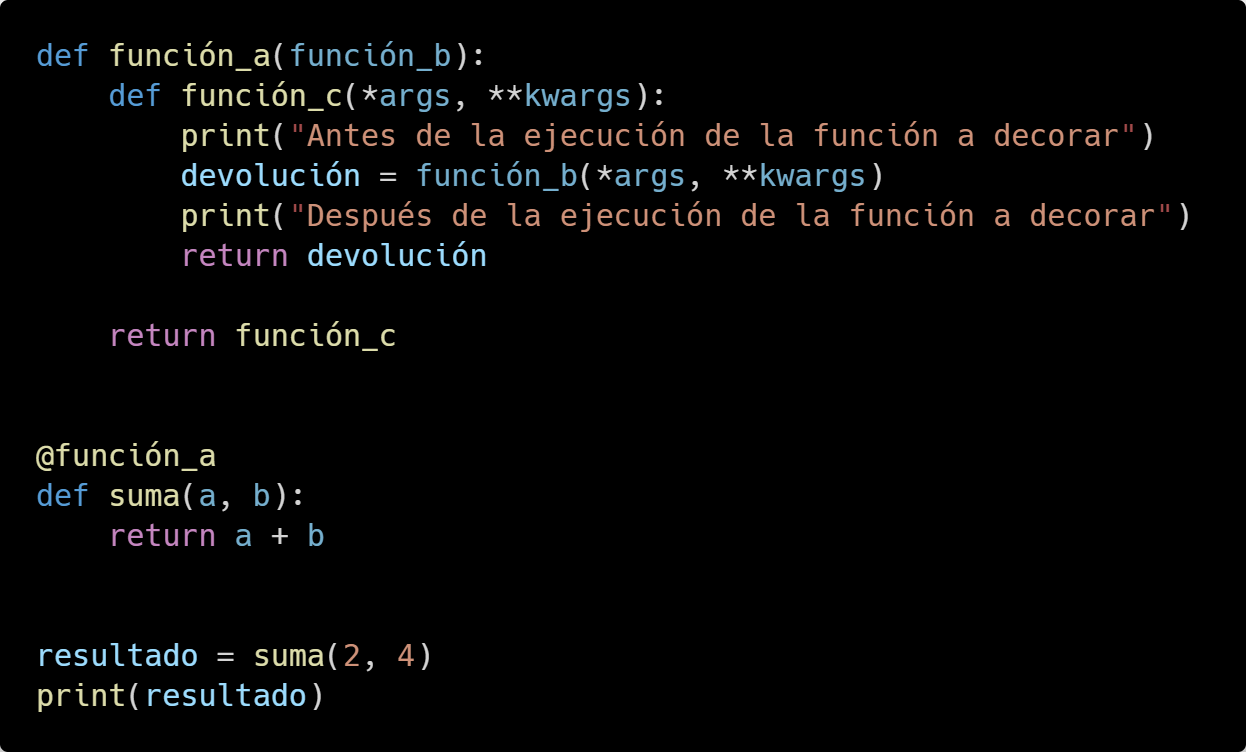
**Usemos el decorador:**

@función\_a  
def saludar():  
 print("¡Hola mundo!")

saludar()



¿Qué sucede si nuestra función a decorar debe recibir argumentos, y, a su vez, devolver algún valor?: Hemos de hacer uso de los parámetros args y kwargs.



| Si usas los parámetros args y kwargs serás capaz de reutilizar el decorador, haciéndolo aún más flexible de lo que ya es. |
| --- |

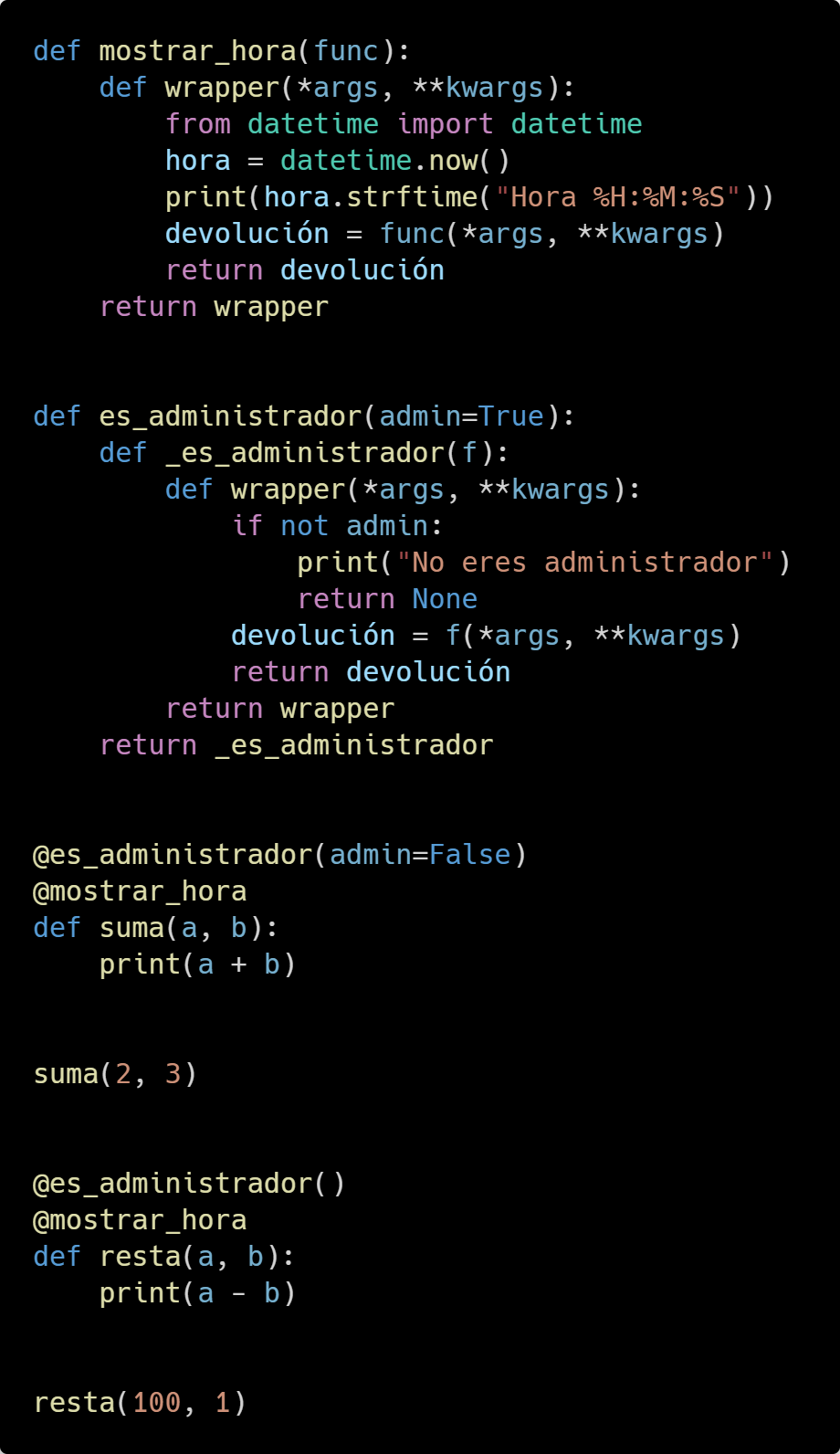
Por convención, no es una regla, la función anidada del decorador suele usar el nombre: wrapper. El nombre del decorador debe ser muy descriptivo. En términos simples, el decorador pudiera quedar de la siguiente manera.

def decorador(función):  
 def wrapper(\*args, \*\*kwargs):  
 return función(\*args, \*\*kwargs)  
 return wrapper

## 

## Decoradores con parámetros

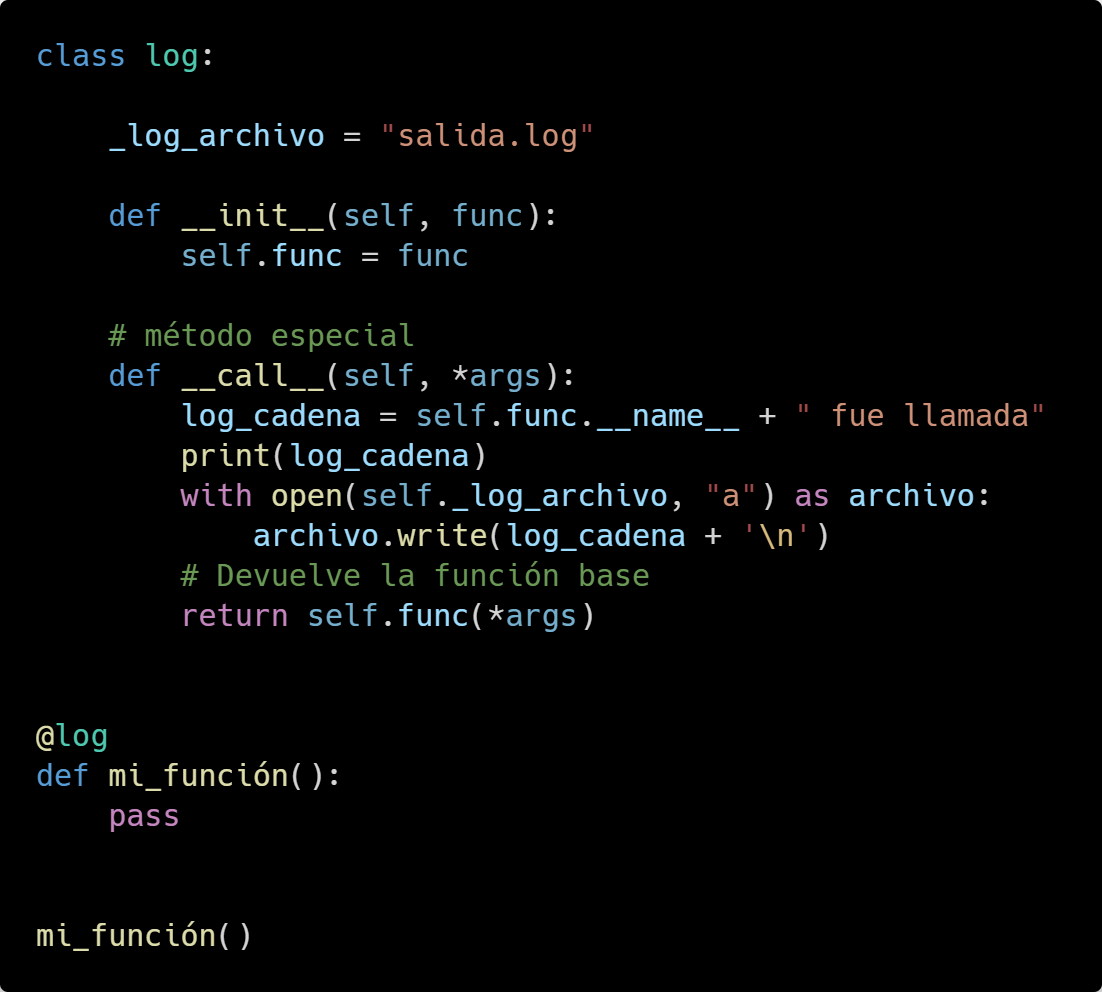
**Ejecuta el siguiente código y prueba cambiar el orden de los decoradores:**



| Puedes tener más de un decorador aplicado a una función, y el orden de ejecución empieza por el decorador más cercano a la definición de la función. |
| --- |

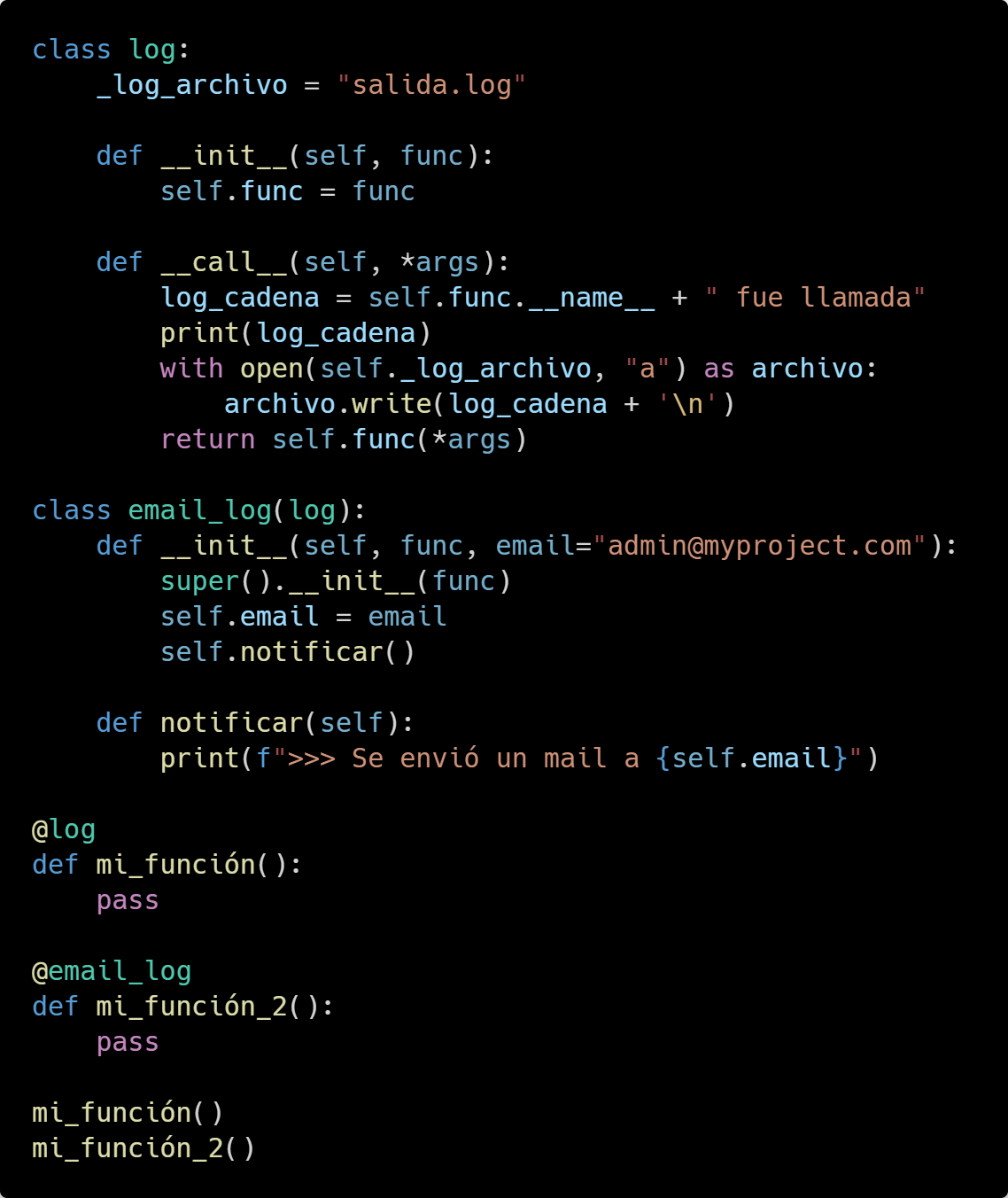
## Decoradores de clases

**Ejecuta varias veces este código, mientras vas observando qué hay en el archivo salida.log:**



### ¿Herencia de decoradores?

**email\_log hará lo mismo que log, pero además enviará un email. Ejecuta:**



### 

**¡MANOS A LA OBRA!**

**Ejercicio 4**

* Importa el módulo **time** que viene incorporado a Python, el cual tiene una función llamada **sleep,** que recibe un float como parámetro que representa los segundos en que el programa "dormirá", es decir, se pausará sin hacer nada.
* Crea un programa que cada 2 segundos ejecute 3 veces una función que imprima "Ejecutando función"
* La función estará decorada por un **decorador definido en una función** (dormirá a la función por 2 segundos).
* Crea otro **decorador definido en una clase** que imprima hora, minutos, segundos y microsegundos en que fue "despertada", y grabe estos datos en una línea de un archivo llamado "registro.log"

|  | **Ver video “**[**Decoradores**](https://youtu.be/2tN4iqQCvnY)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Conocer el funcionamiento de los decoradores y su utilidad
* Crear decoradores por medio de funciones
* Usar decoradores que reciban parámetros para la función decorada
* Crear decoradores por medio de clases
* Heredar decoradores de clase

# 

# Generadores

Un generador es un tipo especial de función, la cual retorna objetos que fácilmente podemos iterar, esto **sin que la función finalice.**

Vamos a crear un generador que nos permita iterar sobre números pares:

**Ejecuta el siguiente ejemplo:**

def pares(desde: int, hasta: int):  
 for x in range(desde, hasta, 2):  
 yield x

print(pares(0, 100))

¿Qué es <generator object pares at 0x7...>?

Un generador es un objeto de tipo iterable, como las listas o las tuplas. Los generadores se crean con una función y la sentencia **yield.** No devuelve la función, tampoco destruye las variables locales.

**Ejecuta el siguiente ejemplo:**

def pares(desde: int, hasta: int):  
 for x in range(desde, hasta, 2):  
 yield x

generador = pares(0, 5)  
print(next(generador))  
print(next(generador))  
print(next(generador))  
print(next(generador))

¿StopIteration?

La palabra clave **next** hace que el iterador devuelva un elemento y se mueva hacia el próximo elemento. Si no hay más elementos, se lanza el error StopIteration. Itera con for para no lanzar el error:

**Ejecuta el siguiente ejemplo:**

def pares(desde: int, hasta: int):  
 for x in range(desde, hasta, 2):  
 yield x

generador = pares(0, 5)  
 for x in generador:  
 print(x)

Un generador va devolviendo resultados mientras va iterando. Produce un elemento a la vez y continúa con la función, por eso, no tiene problemas de memoria. La palabra clave yield nos dice que es una función generadora, y va devolviendo un elemento (en este ejemplo x) a demanda (en este ejemplo, lo va demandando for)

### Creación de generadores por expresión

Otra forma de crear un generador es mediante el uso de una expresión.

| Las expresiones generadoras son parecidas a las listas por comprensión, pero en vez de usar los corchetes se usan los paréntesis. |
| --- |

**Ejecuta el siguiente ejemplo para entender la expresión generadora:**

números = (número for número in range(0, 100))

print(type(números))  
 print(list(números))  
 print(list(números))

¿Una lista vacía? Recuerda que un generador va entregando a demanda: la conversión (o casting) del generador a lista: list(números), hizo que el generador recorra todos sus elementos y se los dé a la función list(). Cuando se vuelve a ejecutar list(números), ya no tiene más que entregar.

### Encadenar generadores

**Ejecuta el siguiente ejemplo para que experimentes el concepto:**

números = (número for número in range(0, 100))  
 pares = (número for número in números if número % 2 == 0)

print("Listado de números pares: ")

for par in pares:  
 print(par)

### 

**¡MANOS A LA OBRA!**

**Ejercicio 5**

* Importa el módulo **random** que viene incorporado a Python, el cual tiene una función llamada **randint,** que recibe dos parámetros de tipo int. Lo que hace randint es devolver un entero al azar, entre el primer parámetro y el segundo.
* Crea una función generadora llamada "lotería", que devuelva a demanda nueve números entre el 0 y el 100.
* En la misma función también debes devolver el décimo número que esté entre el 0 y el 10.
* Imprime los números de la lotería.

|  | **Ver video “**[**Generadores**](https://youtu.be/ORAG3Qanai8)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Aprender los conceptos básicos de los generadores
* Crear generadores con yield
* Crear generadores con expresiones
* Enlazar expresiones de generadores