

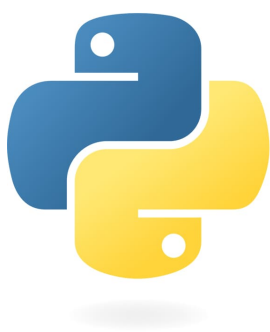
# Python\_ficheros

January 20, 2026



**INSTITUTO SUPERIOR  
TECNOLÓGICO QUITO**  
Excelencia en Educación Superior

## 0.1 Ficheros



python<sup>TM</sup>

*Gabriel Salguero*

## 0.2 Acceso a ficheros

### 0.2.1 Ficheros

- *File* es un tipo de objeto predefinido en Python (*built-in*).
- Permite acceder a ficheros desde programas Python.
- Los ficheros son de un tipo especial:
  - Son *built-in*, pero no son ni *números*, ni *secuencias*, ni *mappings*. Tampoco responden a operadores en expresiones.
- La función *open* permite crear objetos de tipo fichero.

Formato general para abrir un fichero:

```
afile = open(filename, mode)
```

- *mode* es opcional. Por defecto, los ficheros se abren en modo lectura.
- Los datos leídos de un fichero siempre se obtienen en formato *string*. Lo mismo ocurre con escritura.
- Los ficheros se deben cerrar invocando *close* (liberación de recursos).

```
[56]: # Lectura desde fichero usando método 'read'. Devuelve todo el contenido del
      ↪ fichero.
mi_fichero = open('res/multiple_lines.txt')
print(mi_fichero.read())
mi_fichero.close()
```

Este fichero  
contiene tres  
líneas de texto.

```
[57]: # Lectura línea a línea a través del bucle 'for'.
mi_fichero = open('res/multiple_lines.txt')
for linea in mi_fichero:
    print(f"{linea}")

mi_fichero.close()
```

Este fichero  
contiene tres  
líneas de texto.

```
[58]: # leer de un archivo a una lista.
mi_fichero = open('res/multiple_lines.txt')
lineas = mi_fichero.readlines()
mi_fichero.close()

print(lineas)
```

```
['Este fichero\n', 'contiene tres\n', 'lineas de texto.\n']
```

- Resolución de paths independiente de plataforma

```
[59]: import os
ruta = os.path.join("res", "multiple_lines.txt")
print(ruta)
```

```
res\multiple_lines.txt
```

```
[60]: dataset_file_path = ["res", "multiple_lines.txt"]
ruta = os.path.join(*dataset_file_path)
print(ruta)
```

```
res\multiple_lines.txt
```

- *close* automático con sentencia *with*. Esta es la forma habitual de leer de fichero en Python, context manager

```
[61]: with open(ruta) as mi_fichero:
      for linea in mi_fichero:
          print(linea, end= '')
```

```
Este fichero
contiene tres
lineas de texto.
```

- Se puede abrir varios ficheros en un mismo *with*.

```
[62]: #abrir varios ficheros en el mismo with
ruta1 = os.path.join("res", "one_line.txt")
ruta2 = os.path.join("res", "multiple_lines.txt")
with open(ruta1) as fichero1, open(ruta2) as fichero2:
    print(fichero1.readlines())
    print(fichero2.readlines())
```

```
['Hola mundo desde un fichero.']
```

```
['Este fichero\n', 'contiene tres\n', 'lineas de texto.\n']
```

## 0.2.2 Modos de acceso

- Al crear un objeto de tipo *File* se puede especificar el modo de acceso (lectura/escritura).

Modo Acceso	Descripción
r	Solo Lectura
w	Solo Escritura (Borra si el archivo ya existe)
x	Solo Escritura (Falla si el archivo ya existe)
a	Crea Fichero (Si existe lo abre y se añade al final)
r+	Lectura y Escritura
b	Se puede añadir a otros modos para acceso binario
t	Acceso para archivos de texto (default)

### 0.2.3 Acceso para escritura

```
[63]: def crear_lista(tamanyo):  
    lista = []  
    for i in range(tamanyo):  
        lista.append(str(i) + '\n')  
    return lista  
  
ruta = os.path.join("res", "a_dummy.txt")  
with open(ruta, 'wt') as fichero:  
    fichero.write('Cabecera del ciclo del for en el ejemplo\n')  
    lista = crear_lista(10)  
    fichero.writelines(lista)  
    print("archivo creado")
```

archivo creado

#### Buffering

- Por defecto, el texto que transfieres desde tu programa a un fichero no se guarda en disco inmediatamente. Se almacena en un buffer.
- Acciones como cerrar un fichero o invocar el método **flush** fuerzan que se transfiera el contenido del buffer a disco.

```
[64]: ruta = os.path.join("res", "FicheroParaEscritura.txt")  
fichero_write = open(ruta, 'w')  
fichero_write.write('foter')  
  
fichero_read = open(ruta, "r")  
print(fichero_read.readlines())  
  
fichero_write.flush()  
  
print(fichero_read.readlines())  
  
fichero_write.close()  
fichero_read.close()
```

```
[]  
['foter']
```

### 0.2.4 Archivos CSV

- Python permite leer datos de ficheros CSV y también escribir ficheros en este formato.
- Popular formato en ciencia de datos.

```
[65]: # tabla_operaciones.csv contiene valores separados por comas  
import csv  
ruta = os.path.join("res", "tabla_operaciones.csv")
```

```

with open(ruta) as fichero:
    data_reader = csv.reader(fichero, delimiter=',')
    for linea in data_reader:
        print(linea[0] + '---- ' + linea[1])

```

```

Operacion---- Descripcion
a + b---- suma a y b
a - b---- resta a menos b
a / b---- a dividido
a // b---- a dividido entre b (quitando decimales)
a % b---- devuelve el resto de la división a/b (modulus)
a * b---- a multiplicado por b
a ** b---- a elevado a b

```

```

[66]: ruta = os.path.join("res", "tabla_operaciones.csv")
f = open(ruta)
data_reader = csv.reader(f, delimiter = ',')

ruta_o = os.path.join("res", "4_tabla_operaciones.csv")
with open(ruta_o, 'w') as f2:
    csv_writer = csv.writer(f2, delimiter = '|')
    for line in data_reader:
        print(line)
        csv_writer.writerow(line)

f.close()

```

```

['Operacion', 'Descripcion']
['a + b', 'suma a y b']
['a - b', 'resta a menos b']
['a / b', 'a dividido']
['a // b', 'a dividido entre b (quitando decimales)']
['a % b', 'devuelve el resto de la división a/b (modulus)']
['a * b', 'a multiplicado por b']
['a ** b', 'a elevado a b']

```

```

[67]: help(open)

```

Help on function open in module io:

```

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)

```

Open file and return a stream. Raise OSError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for creating and writing to a new file, and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: `locale.getencoding()` is called to get the current locale encoding.

(For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

=====	
Character	Meaning
-----	
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
=====	

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation. The 'x' mode implies 'w' and raises an `FileExistsError` if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- \* Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on `io.DEFAULT_BUFFER_SIZE`.  
On many systems, the buffer will typically be 4096 or 8192 bytes long.

\* "Interactive" text files (files for which `isatty()` returns `True`) use line buffering. Other text files use the policy described above for binary files.

`encoding` is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the `codecs` module for the list of supported encodings.

`errors` is an optional string that specifies how encoding errors are to be handled---this argument should not be used in binary mode. Pass `'strict'` to raise a `ValueError` exception if there is an encoding error (the default of `None` has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for `codecs.register` or run `'help(codecs.Codec)'` for a list of the permitted encoding error strings.

`newline` controls how universal newlines works (it only applies to text mode). It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- \* On input, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If it is `''`, universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- \* On output, if `newline` is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`. If `newline` is `''` or `'\n'`, no translation takes place. If `newline` is any of the other legal values, any `'\n'` characters written are translated to the given string.

If `closefd` is `False`, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be `True` in that case.

A custom opener can be used by passing a callable as `*opener*`. The underlying file descriptor for the file object is then obtained by calling `*opener*` with `(*file*, *flags*)`. `*opener*` must return an open file descriptor (passing `os.open` as `*opener*` results in functionality similar to passing `None`).

`open()` returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing

are performed. When `open()` is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a `TextIOWrapper`. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a `BufferedReader`; in write binary and append binary modes, it returns a `BufferedWriter`, and in read/write mode, it returns a `BufferedRandom`.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings `StringIO` can be used like a file opened in a text mode, and for bytes a `BytesIO` can be used like a file opened in a binary mode.

- No es necesario editar este atributo directamente.
- Para asociar un docstring a un objeto basta con escribir el texto (entre triples comillas) al principio de los módulos, funciones o clases, antes del código ejecutable.

```
[68]: def funcion_de_prueba():  
      """Esta es la documentación de la función de prueba"""  
      pass  
  
help(funcion_de_prueba)
```

Help on function funcion\_de\_prueba in module \_\_main\_\_:

```
funcion_de_prueba()  
    Esta es la documentación de la función de prueba
```

- Accediendo al atributo `__doc__` sólo se obtiene el docstring.

```
[69]: print(funcion_de_prueba.__doc__)
```

Esta es la documentación de la función de prueba

Como se puede observar, *help* añade información adicional. Por ejemplo, para entidades más grandes, *help* muestra el docstring dividido en secciones.

```
[70]: import sys  
      help(sys)
```

Help on built-in module sys:

```
NAME  
    sys
```

```
MODULE REFERENCE  
    https://docs.python.org/3.11/library/sys.html
```

The following documentation is automatically generated from the Python source files. It may be incomplete, incorrect or include features that



are considered implementation detail and may vary between Python implementations. When in doubt, consult the module reference at the location listed above.

## DESCRIPTION

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

`argv` -- command line arguments; `argv[0]` is the script pathname if known  
`path` -- module search path; `path[0]` is the script directory, else ''  
`modules` -- dictionary of loaded modules

`displayhook` -- called to show results in an interactive session  
`excepthook` -- called to handle any uncaught exception other than `SystemExit`  
To customize printing in an interactive session or to install a custom top-level exception handler, assign other functions to replace these.

`stdin` -- standard input file object; used by `input()`  
`stdout` -- standard output file object; used by `print()`  
`stderr` -- standard error object; used for error messages  
By assigning other file objects (or objects that behave like files) to these, it is possible to redirect all of the interpreter's I/O.

`last_type` -- type of last uncaught exception  
`last_value` -- value of last uncaught exception  
`last_traceback` -- traceback of last uncaught exception  
These three are only available in an interactive session after a traceback has been printed.

Static objects:

`builtin_module_names` -- tuple of module names built into this interpreter  
`copyright` -- copyright notice pertaining to this interpreter  
`exec_prefix` -- prefix used to find the machine-specific Python library  
`executable` -- absolute path of the executable binary of the Python

interpreter

`float_info` -- a named tuple with information about the float implementation.  
`float_repr_style` -- string indicating the style of `repr()` output for floats  
`hash_info` -- a named tuple with information about the hash algorithm.  
`hexversion` -- version information encoded as a single integer  
`implementation` -- Python implementation information.  
`int_info` -- a named tuple with information about the int implementation.  
`maxsize` -- the largest supported length of containers.  
`maxunicode` -- the value of the largest Unicode code point  
`platform` -- platform identifier  
`prefix` -- prefix used to find the Python library

thread\_info -- a named tuple with information about the thread implementation.

version -- the version of this interpreter as a string  
version\_info -- version information as a named tuple  
dllhandle -- [Windows only] integer handle of the Python DLL  
winver -- [Windows only] version number of the Python DLL  
\_enablelegacywindowsfsencoding -- [Windows only]  
\_\_stdin\_\_ -- the original stdin; don't touch!  
\_\_stdout\_\_ -- the original stdout; don't touch!  
\_\_stderr\_\_ -- the original stderr; don't touch!  
\_\_displayhook\_\_ -- the original displayhook; don't touch!  
\_\_excepthook\_\_ -- the original excepthook; don't touch!

Functions:

displayhook() -- print an object to the screen, and save it in builtins.\_  
excepthook() -- print an exception and its traceback to sys.stderr  
exception() -- return the current thread's active exception  
exc\_info() -- return information about the current thread's active exception  
exit() -- exit the interpreter by raising SystemExit  
getdlopenflags() -- returns flags to be used for dlopen() calls  
getprofile() -- get the global profiling function  
getrefcount() -- return the reference count for an object (plus one :-)  
getrecursionlimit() -- return the max recursion depth for the interpreter  
getsizeof() -- return the size of an object in bytes  
gettrace() -- get the global debug tracing function  
setdlopenflags() -- set the flags to be used for dlopen() calls  
setprofile() -- set the global profiling function  
setrecursionlimit() -- set the max recursion depth for the interpreter  
settrace() -- set the global debug tracing function

## FUNCTIONS

```
__breakpointhook__ = breakpointhook(...)  
    breakpointhook(*args, **kws)
```

This hook function is called by built-in breakpoint().

```
__displayhook__ = displayhook(object, /)  
    Print an object to sys.stdout and also save it in builtins._
```

```
__excepthook__ = excepthook(exctype, value, traceback, /)  
    Handle an exception by displaying it with a traceback on sys.stderr.
```

```
__unraisablehook__ = unraisablehook(unraisable, /)  
    Handle an unraisable exception.
```

The unraisable argument has the following attributes:

- \* exc\_type: Exception type.
- \* exc\_value: Exception value, can be None.
- \* exc\_traceback: Exception traceback, can be None.
- \* err\_msg: Error message, can be None.
- \* object: Object causing the exception, can be None.

addaudithook(hook)  
 Adds a new audit hook callback.

audit(...)  
 audit(event, \*args)

Passes the event to any audit hooks that are attached.

call\_tracing(func, args, /)  
 Call func(\*args), while tracing is enabled.

The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug some other code.

exc\_info()  
 Return current exception information: (type, value, traceback).

Return information about the most recent exception caught by an except clause in the current stack frame or in an older stack frame.

exception()  
 Return the current exception.

Return the most recent exception caught by an except clause in the current stack frame or in an older stack frame, or None if no such exception exists.

exit(status=None, /)  
 Exit the interpreter by raising SystemExit(status).

If the status is omitted or None, it defaults to zero (i.e., success).  
 If the status is an integer, it will be used as the system exit status.  
 If it is another kind of object, it will be printed and the system exit status will be one (i.e., failure).

get\_asyncgen\_hooks()  
 Return the installed asynchronous generators hooks.

This returns a namedtuple of the form (firstiter, finalizer).

get\_coroutine\_origin\_tracking\_depth()

Check status of origin tracking for coroutine objects in this thread.

`get_int_max_str_digits()`  
Return the maximum string digits limit for non-binary int<->str conversions.

`getallocatedblocks()`  
Return the number of memory blocks currently allocated.

`getdefaultencoding()`  
Return the current default encoding used by the Unicode implementation.

`getfilesystemencodeerrors()`  
Return the error mode used Unicode to OS filename conversion.

`getfilesystemencoding()`  
Return the encoding used to convert Unicode filenames to OS filenames.

`getprofile()`  
Return the profiling function set with `sys.setprofile`.

See the profiler chapter in the library manual.

`getrecursionlimit()`  
Return the current value of the recursion limit.

The recursion limit is the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

`getrefcount(object, /)`  
Return the reference count of object.

The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to `getrefcount()`.

`getsizeof(...)`  
`getsizeof(object [, default]) -> int`

Return the size of object in bytes.

`getswitchinterval()`  
Return the current thread switch interval; see `sys.setswitchinterval()`.

`gettrace()`  
Return the global debug tracing function set with `sys.settrace`.

See the debugger chapter in the library manual.

`getwindowsversion()`

Return info about the running version of Windows as a named tuple.

The members are named: `major`, `minor`, `build`, `platform`, `service_pack`, `service_pack_major`, `service_pack_minor`, `suite_mask`, `product_type` and `platform_version`. For backward compatibility, only the first 5 items are available by indexing. All elements are numbers, except `service_pack` and `platform_type` which are strings, and `platform_version` which is a 3-tuple. Platform is always 2. Product\_type may be 1 for a workstation, 2 for a domain controller, 3 for a server. Platform\_version is a 3-tuple containing a version number that is intended for identifying the OS rather than feature detection.

`intern(string, /)`

``Intern'' the given string.

This enters the string in the (global) table of interned strings whose purpose is to speed up dictionary lookups. Return the string itself or the previously interned string object with the same value.

`is_finalizing()`

Return True if Python is exiting.

`set_asyncgen_hooks(...)`

`set_asyncgen_hooks([firstiter] [, finalizer])`

Set a finalizer for async generators objects.

`set_coroutine_origin_tracking_depth(depth)`

Enable or disable origin tracking for coroutine objects in this thread.

Coroutine objects will track 'depth' frames of traceback information about where they came from, available in their `cr_origin` attribute.

Set a depth of 0 to disable.

`set_int_max_str_digits(maxdigits)`

Set the maximum string digits limit for non-binary `int<->str` conversions.

`setprofile(...)`

`setprofile(function)`

Set the profiling function. It will be called on each function call and return. See the profiler chapter in the library manual.

`setrecursionlimit(limit, /)`

Set the maximum depth of the Python interpreter stack to `n`.

This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. The highest possible limit is platform-dependent.

`setswitchinterval(interval, /)`

Set the ideal thread switching delay inside the Python interpreter.

The actual frequency of switching threads can be lower if the interpreter executes long sequences of uninterruptible code (this is implementation-specific and workload-dependent).

The parameter must represent the desired switching delay in seconds. A typical value is 0.005 (5 milliseconds).

`settrace(...)`

`settrace(function)`

Set the global debug tracing function. It will be called on each function call. See the debugger chapter in the library manual.

`unraisablehook(unraisable, /)`

Handle an unraisable exception.

The unraisable argument has the following attributes:

- \* `exc_type`: Exception type.
- \* `exc_value`: Exception value, can be `None`.
- \* `exc_traceback`: Exception traceback, can be `None`.
- \* `err_msg`: Error message, can be `None`.
- \* `object`: Object causing the exception, can be `None`.

## DATA

```
__stderr__ = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>
__stdin__ = <_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>
__stdout__ = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>
api_version = 1013
argv = [r'E:\anaconda3\envs\machines2026\Lib\site-packages\ipykernel_launcher.py',
base_exec_prefix = r'E:\anaconda3\envs\machines2026'
base_prefix = r'E:\anaconda3\envs\machines2026'
builtin_module_names = ('_abc', '_ast', '_bisect', '_blake2', '_codecs...',
byteorder = 'little'
copyright = 'Copyright (c) 2001-2023 Python Software Foundation. All rights reserved.'
displayhook = <ipykernel.displayhook.ZMQShellDisplayHook object>
dllhandle = 140711574241280
dont_write_bytecode = False
```

```

exec_prefix = r'E:\anaconda3\envs\machines2026'
executable = r'E:\anaconda3\envs\machines2026\python.exe'
flags = sys.flags(debug=0, inspect=0, interactive=0, opt...coding=0, s...
float_info = sys.float_info(max=1.7976931348623157e+308, max_...epsilo...
float_repr_style = 'short'
hash_info = sys.hash_info(width=64, modulus=2305843009213693...iphash1...
hexversion = 51056368
implementation = namespace(name='cpython', cache_tag='cpython-311...as...
int_info = sys.int_info(bits_per_digit=30, sizeof_digit=4, ..._str_dig...
last_value = FileNotFoundError(2, 'No such file or directory')
maxsize = 9223372036854775807
maxunicode = 1114111
meta_path = [<distutils_hack.DistutilsMetaFinder object>, <class '_fr...
modules = {'IPython': <module 'IPython' from 'E:\\anaconda3\\envs\\mac...
orig_argv = [r'E:/anaconda3/envs/machines2026/python.exe', '-Xfrozen_m...
path = [r'E:\anaconda3\envs\machines2026\python311.zip', r'E:\anaconda...
path_hooks = [<class 'zipimport.zipimporter'>, <function FileFinder.pa...
path_importer_cache = {r'C:\Users\Reromash\Desktop\SalgueroGabriel\mac...
platform = 'win32'
platlibdir = 'DLLs'
prefix = r'E:\anaconda3\envs\machines2026'
ps1 = 'In : '
ps2 = '...: '
ps3 = 'Out: '
pycache_prefix = None
stderr = <ipykernel.iostream.OutputStream object>
stdin = <_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>
stdlib_module_names = frozenset({'__future__', '_abc', '_aix_support',...
stdout = <ipykernel.iostream.OutputStream object>
thread_info = sys.thread_info(name='nt', lock=None, version=None)
version = '3.11.14 | packaged by Anaconda, Inc. | (main, Oct 21 2025, ...
version_info = sys.version_info(major=3, minor=11, micro=14, releasele...
warnoptions = []
winver = '3.11'

```

FILE

(built-in)

Los docstrings que ocupan más de una línea suelen tener estas partes:

- Resumen en la primera línea.
- Línea en blanco.
- Descripción detallada.
- Otra línea en blanco antes del código.

```
[71]: def funcion_de_prueba2():
        """Esta es la línea para el resumen.

        Este es el párrafo donde se puede escribir una descripción más detallada
        de la función. Observa que el resumen y la descripción detallada van
        ↪ separados
        por una línea en blanco. También hay otra línea en blanco antes de empezar
        ↪ el código. """

        pass

help(funcion_de_prueba2)
```

Help on function funcion\_de\_prueba2 in module \_\_main\_\_:

```
funcion_de_prueba2()
    Esta es la línea para el resumen.

    Este es el párrafo donde se puede escribir una descripción más detallada
    de la función. Observa que el resumen y la descripción detallada van
separados
    por una línea en blanco. También hay otra línea en blanco antes de empezar
el código.
```

Algunos comentarios adicionales:

- Comentarios con '#' suelen asociarse a expresiones sencillas, instrucciones individuales o pequeños bloques de código.
- Los docstrings son más apropiados para construcciones de más alto nivel: funciones, clases y módulos.

### 0.2.5 Sphinx

- Herramienta de documentación.
- Especialmente útil para sistemas grandes.
- Dasoporte a una gran variedad de formatos de salida: HTML, LaTeX, ePub, ...

Web de [Sphinx](#).

[Proyectos](#) que usan Sphinx.

### 0.2.6 Estilo

- Es importante que los docstrings sean consistentes.
- El equipo de desarrollo debe acordar un formato y seguirlo rigurosamente.

**Google Python Style Guide** Google define la siguiente [guía de estilo](#).

Contiene reglas para los docstrings de:

- Módulos.



- Métodos y funciones.
- Clases.

### 0.2.7 Ejercicios

1. Escribe una función que reciba una ruta de un fichero de texto y una cadena de caracteres a buscar y determine si la cadena aparece en el fichero.
2. Escribe una función que reciba una lista, una ruta destino y un número n. La función debe crear un fichero en la ruta especificada. El contenido del fichero serán los primeros n elementos de la lista. La función debe controlar de manera apropiada los posibles valores de n que estén fuera de rango.
3. Escribe una función que reciba una ruta de un fichero de texto devuelva un diccionario con la frecuencia de aparición de cada palabra. Ejemplo: un fichero que contenga la frase 'es mejor que venga que que no venga' devolverá el siguiente diccionario: {'es' : 1, 'mejor' : 1, 'que' : 3, 'venga' : 2, 'no' : 1}. Para dividir un string en palabras puedes hacer uso del método split.

[72]: *'''Escribe una función que reciba una ruta de un fichero de texto y una cadena de caracteres a buscar y determine si la cadena aparece en el fichero.'''*

```
def existe_texto(archivo, texto):
    with open(archivo, 'r') as f_obj:
        contenido = f_obj.read()
        return texto in contenido

print(existe_texto("res/tareaFichero.txt", "Desarrollo"))
print(existe_texto("res/tareaFichero.txt", "Hola"))
```

True

True

[73]: *'''Escribe una función que reciba una lista, una ruta destino y un número n. La función debe crear un fichero en la ruta especificada. El contenido del fichero serán los primeros n elementos de la lista. La función debe controlar de manera apropiada los posibles valores de n que estén fuera de rango.'''*

```
def guardar_primeros_n(lista_datos, ruta_salida, limite):
    # Aseguramos que el limite este dentro del rango de la lista (0 a len)
    limite = max(0, min(limite, len(lista_datos)))

    with open(ruta_salida, 'w') as file_out:
        # Usamos writelines con comprensión de lista para ser más eficientes
        lineas = [str(dato) + '\n' for dato in lista_datos[:limite]]
        file_out.writelines(lineas)

mis_datos = [2, 4, 6, 8, 10, 12, 14, 16]
guardar_primeros_n(mis_datos, "res/Ejercicio2_G.txt", 4)
print('Archivo generado con éxito.')
```

Archivo generado con éxito.

[74]: *'''Escribe una función que reciba una ruta de un fichero de texto devuelva un diccionario con la frecuencia de aparición de cada palabra. Ejemplo: un fichero que contenga la frase 'es mejor que venga que que no venga' devolverá el\_ siguiente*  
*diccionario: {'es' : 1, 'mejor' : 1, 'que' : 3, 'venga' : 2, 'no' : 1}.*  
*Para dividir un string en palabras puedes hacer uso del método split.'''*

```
def obtener_frecuencias(path_archivo):
    frecuencias = {}
    with open(path_archivo, 'r') as archivo:
        texto = archivo.read()
        tokens = texto.split()

        for t in tokens:
            if t in frecuencias:
                frecuencias[t] += 1
            else:
                frecuencias[t] = 1
    return frecuencias

print(obtener_frecuencias("res/Ejercicio3.txt"))
```

```
{'palabras': 2, 'son': 1, 'mas': 1, 'uno': 1, 'Hola': 1, 'mi': 1, 'nombre': 1,
'es': 1, 'Gabriel': 1, 'Salguero': 1, 'estudio': 1, 'desarrollo': 1, 'de': 1,
'software': 1}
```

### 0.3 Referencias

GitHub: <https://github.com/reromash1/machines2026>