Python 3: llevando una bitácora de programa

En el siglo IX los chinos inventaron la brújula (aguja imantada suspendida que siempre apunta al polo norte) y desde entonces le destinaron en cada barco un *habitaculum (en latín, habitáculo en castellano)* al cual los franceses le nombraron *habitacle* y que luego abreviaron como *bitacle* **y que pasó a ser traducido al castellano como bitácora** (a pesar de que ya teníamos la palabra traducida directamente del latín, habitáculo). Pues bien, se necesitaba llevar un registro de la posición del barco en los largos viajes por nuestro globo terráqueo (y junto al sextante para registrar los astros) todo se anotaba en un **cuaderno de bitácora, o simplemente bitácora**.

¿A donde nos lleva esta introducción que aparentemente no tiene nada que ver con computación? ¡Ya veremos!

Introducción.

Así como los gobiernos en tierra necesitaban conocer qué sucedió en un navío en altamar a su regreso, nosotros necesitamos saber qué sucedió en los programas que para bien desarrollemos para nuestros usuarios. Lo más básico es mostrar mensajes por pantalla a los usuarios y confiar en que ellos y ellas nos retribuyan debidamente la información... *pero con muy contadas excepciones, podemos esperar sentados para no cansarnos porque eso será difícil que se haga realidad.*

Es por ello que debemos guardar un registro metódico para que posteriormente podamos evaluar qué funcionó mal (por extraño que parezca, si funciona bien pues felices de la vida aunque no recibamos las felicitaciones de nuestros usuarios y usuarias de software). Otra razón de llevar un registro sería la de análisis de desempeño o incluso ejecutar un programa en modo de depuración.

La razón y la lógica indica que dichos registros que pensamos llevar *deberían ser guardados en una base de datos* pero en proyectos pequeños tal vez no necesitemos tal nivel de complejidad. Pongamos por caso el programa **Filezilla** que tiene ambas versiones tanto como servidor como cliente: por defecto no se registra mensaje alguno a menos que así lo deseemos y si decidimos guardarlo podemos especificar un archivo llevando la fecha de cada evento (opcional) e incluso podemos limitar a un tamaño específico tras lo cual al alcanzar dicho valor se procede a crear un archivo nuevo pero sin la extensión ".log" la cual es sustituida por una numeración consecutiva.

Por esta y muchas otras razones el lenguaje Python 3 tiene disponible una librería destinada para tal efecto, estudiemos pues.

Creando una aplicación modelo.

Antes de crear siquiera registro alguno debemos tener, claro está, un software al cual llevarle un

registro. Para ello proponemos un programa que llamaremos **calculadora1.py** cuyo código es el siguiente (si queréis repasar vuestro conocimientos básicos sobre Python, revisad nuestro tutorial al respecto):

```
class calculadora():      def __init__():      print("\nCalculadora encen
dida.")      def sumar( a=0, b=0):      print("Suma a={} b={} a+b={}".for
mat(a,b,a+b))      def restar( a=0, b=0):      print("Resta a={} b={} a-b
={}".format(a,b,a-b))      def multiplicar( a=0, b=0):      print("Multip
licación a={} b={} a*b={}".format(a,b,a*b))      def dividir( a=0, b=1):
     print("División a={} b={} a/b={}".format(a,b,a/b))    calc = calcula
dora  calc.sumar( 12,4)  calc.restar(12,4)  calc.multiplicar(12,4)  calc.
dividir(12,4)
```

El código es bastante sencillo, solo las cuatro operaciones aritméticas básicas: suma, resta, multiplicación y división; reconocemos que el código es un tanto extraño pero recordad que tiene propósitos didácticos solamente. Creamos una **clase** con funciones que no emplean **return** sino que muestran por pantalla los resultados excepto en la inicialización que muestra un mensaje puramente informativo, emulando el "**on**" de una calculadora electrónica y anunciando el modelo virtual. Abstraigámonos entonces en el ejemplo para comenzar a modificarlo con el registro de eventos.

Agregando la utilería "logging".

Para comenzar a utilizar la librería **logging** debemos incorporarla a nuestro archivo con el siguiente código:

```
import logging  logging.basicConfig(level=logging.DEBUG)
```

La primera línea enlaza la librería y la segunda línea configuramos con una constante **logging.DEBUG** (que tiene un valor decimal de diez), osea, el nivel ("**level**") que vamos a usar: modo de depuración.

> Notad todos y todas que en GNU/Linux son distintas las mayúsculas de las minúsculas, por lo tanto logging.DEBUG es una constante y logging.debug es un método, diferenciad bien esto en el siguiente código que modificamos a partir de la aplicación modelo.

```
import logging  logging.basicConfig(level=logging.DEBUG)    class calcula
dora():      def __init__(self, nombre):      self.nombre = nombre      l
ogging.debug("\nCalculadora modelo {} (encendida).".format(self.nombre))
     def sumar(self, a=0, b=0):      logging.debug("Suma           a={} b
={} a+b={}".format(a,b,a+b))      def restar(self, a=0, b=0):      loggin
g.debug("Resta          a={} b={} a-b={}".format(a,b,a-b))      def multi
plicar(self, a=0, b=0):      logging.debug("Multiplicación a={} b={} a*b=
{}".format(a,b,a*b))      def dividir(self, a=0, b=1):      logging.debug
("División       a={} b={} a/b={}".format(a,b,a/b))    calc = calculadora
("A1")  calc.sumar( 12,4)  calc.restar(12,4)  calc.multiplicar(12,4)  cal
c.dividir(12,4)
```

Como vemos en la siguiente imagen la salida por pantalla ha sido modificada ya que le agrega "**DEBUG:root**" a todos los mensajes de resultado.

La primera palabra indica que estamos en modo de registro a nivel de depuración **"DEBUG**" y la segunda palabra indica que estamos depurando el módulo principal aunque esto no es realmente cierto. Lo mejor sería indicar desde dónde estamos imprimiendo el mensaje de depuración, en nuestro caso cualquiera de las cuatro funciones. Para ello vamos a volver a modificar el programa -que ya hemos renombrado como **calculadora2.py**- especificando cada función por separado:

```
import logging  logging.basicConfig(level=logging.DEBUG)    bita_sum = lo
gging.getLogger("Sum")  bita_res = logging.getLogger("Res")  bita_mul = l
ogging.getLogger("Mul")  bita_div = logging.getLogger("Div")    class cal
culadora():      def __init__(self, nombre):      self.nombre = nombre
   logging.debug("\nCalculadora modelo {} (encendida).".format(self.nombr
e))      def sumar(self, a=0, b=0):      bita_sum.debug("Suma           a
={} b={} a+b={}".format(a,b,a+b))      def restar(self, a=0, b=0):      b
ita_res.debug("Resta          a={} b={} a-b={}".format(a,b,a-b))      def
 multiplicar(self, a=0, b=0):      bita_mul.debug("Multiplicación a={} b=
{} a*b={}".format(a,b,a*b))      def dividir(self, a=0, b=1):      bita_d
iv.debug("División       a={} b={} a/b={}".format(a,b,a/b))    calc = cal
culadora("A1")  calc.sumar( 12,4)  calc.restar(12,4)  calc.multiplicar(12
,4)  calc.dividir(12,4)
```

Acá vemos la salida correspondiente:

Grabando mensajes en un archivo: bitácora.log

Bien, pues ya estamos listos para comenzar a grabar en un archivo de texto plano nuestros eventos. Esto se logra configurando de nuevo el encabezado **logging.basicConfig** el cual ahora lo ocuparemos en varias líneas para buscar una mayor claridad para cada uno de sus parámetros:

```
logging.basicConfig(    filename="bitacora.log",    level=logging.DEBUG,
 )
```

Por supuesto el archivo será guardado en la misma carpeta donde se ejecuta la aplicación *y para nuestra sorpresa al ejecutarla ya no muestra nada por pantalla...* ***lo cual no es lo que realmente queremos hacer pero paciencia, primero analizemos el archivo resultante.***

Al usar el comando **cat** podremos, entre otras cosas, listar el contenido de un archivo por pantalla y como probablemente la cantidad de mensajes generará gran cantidad de líneas podremos filtrar los resultados por palabra clave. ¿Recordáis que dimos nombres diferentes para la muestra de resultados a nivel de cada función? Pues con el comando **grep** que recibe el resultado del comando **cat** por medio del comando "tubería" "|" y la palabra clave "**Sum**" o "**Mul**" podremos ver lo que nos interese. Ya nuestra aplicación está entrando en modo pragmático, ¡lo realmente útil para nosotros!

Agregando más pragmatismo aún: claridad al registro.

Nosotros los seres humanos en nuestro cerebro siempre buscamos darle "orden" a nuestro mundo, así está torcido lo tratamos de ver derecho y esto en el registro de eventos no ha de ser la excepción. Ya le colocamos para saber cual función produce tal registro pero le agregaremos mayor claridad en el apartado de configuración al inicio de la aplicación:

```
logging.basicConfig(    filename="bitacora.log",    level=logging.DEBUG,
   format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'  )
```

Recordad siempre al final de cada línea colocar una coma para separar los parámetros, que como es multilínea tendemos a pensar que cada retorno de carro automáticamente separa cada

parámeto ***pero no es así.*** En el tercer parámetro mandamos a separar con par de espacios y un guion las diferentes secciones de cada evento en cada línea:

- Fecha y hora exacta hasta en milisegundos cuando ocurrió el evento.
- Nombre del módulo donde se origina cada evento, en nuestro caso cada función.
- Nivel del mensaje, clasificación (hasta ahora estamos en modo de depuración solamente **DEBUG**).
- El mensaje en sí mismo.

Formato de tiempo mejorado.

Al formato de estampado de fecha y hora lo podremos mejorar agregando otra línea más al encabezado de configuración con una máscara que también es utlizada por el comando **time.strftime()**:

```
logging.basicConfig(    filename="bitacora.log",    level=logging.DEBUG,
   format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',    date
fmt='%d/%m/%Y %I:%M:%S %p'  )
```

Dejamos para vosotros os ejercitéis y veáis cómo es distinto los nuevos registros que se siguen adicionando de manera automática al final de nuestro archivo destinado a tal efecto, **bitacora.log**

Nivel de registro de eventos.

Como hemos repetido varias veces, del modo de depuración **DEBUG** no nos hemos movido hasta ahora. Por ello debemos estudiar los diferentes niveles -y constantes- que utiliza la librería **logging**: ya sabemos que **logging.DEBUG** vale diez -y se van incrementando de diez en diez- pero he aquí la tabla completa de valores:

Agregando mensajes de error y su registro.

A nuestra aplicación vamos a agregarle un mensaje de error en la función de división, bien sabemos que cualquier número dividido entre cero tiende al infinito el cual es un concepto que entedemos los seres humanos pero los ordenadores no. La modificación es la siguiente (notad que **b** valdría uno si el valor no es pasado a la función para *tratar* de evitar este error):

```
  def dividir(self, a=0, b=1):      if (b==0):        bita_div.error("Ale
rta: el divisor debe ser distinto a cero.")      else:        bita_div.de
```

```
bug("División       a={} b={} a/b={}".format(a,b,a/b))
```

También modificamos el divisor en la función de división a **calc.dividir(36,0)** y el resultado en el registro de errores mostraría algo aprecido a esto:

```
Calculadora modelo A1 (encendida).  03/05/2017 11:03:07 PM - Sum - DEBUG
- Suma a=36 b=3 a+b=39  03/05/2017 11:03:07 PM - Res - DEBUG - Resta a=36
 b=3 a-b=33  03/05/2017 11:03:07 PM - Mul - DEBUG - Multiplicación a=36 b
=3 a*b=108  03/05/2017 11:03:07 PM - Div - ERROR - Alerta: el divisor deb
e ser distinto a cero.
```

Lo próximo que haremos es modificar de manera completa nuestra aplicación con los diferentes "niveles" de mensajes.

Empleando diferentes niveles de registro.

Volvamos nuestros pasos sobre la sección **logging.basicConfig** donde contiene el nivel de registro de eventos para nuestra aplicación. Recordemos que la establecimos a nivel **DEBUG** y ahora la estableceremos a nivel **INFO**, guardaremos y ejecutaremos de nuevo la aplicación. Luego revisaremos el fichero **bitacora.log** y notaremos que no se registró el mensaje de inicialización pero si quedaron registrados los mensaje de información (y por supuesto el mensaje de error).

El siguiente paso es elevar al nivel de **WARNING** para obtener solamente el mensaje de error por la división entre cero y se repite el resultado si lo elevamos a nivel **ERROR**. *No obtenderemos mensaje alguno si lo establecemos a nivel* **CRITICAL** *ya que la divisón entre cero no solamente ha sido debidamente advertida sino que también ha sido debidamente desviada.*

Pro último, y más difícil de obtener (según la aplicación de modeo didáctico que de *exprofeso* escogimos) es el mensaje a nivel **CRITICAL**. *Volvemos a repetir, este comportamiento es circunstrito estrictamente a nuestra aplicación modelo:* la división está en el mensaje mismo a mostrar en **bita_div.CRITICAL** y nunca lograremos que se muestre ya que está debidamente desviado *además, si no lo desviaramos al ejecutar el compilador Python3 inmediatamente nos mostraría el error si intentamos dividir entre cero y por ende no se ejecuta el programa.*

Nosotros somos de experimentar al máximo, nos hacemos muchas, muchísimas preguntas: **¿Y si compilamos la aplicación, es decir la convertimos a lenguaje binario para ejecutar y lograr el mensaje a nivel CRITICAL?**

Para ello -brevemente- podemos instalar PyInstaller:

```
sudo pip3 pyinstaller
```

Luego simplemente vamos a la carpeta con nuestro fichero calculadora2.py (habiendo eliminado la desviación del error de división entre cero):

```
pyinstaller calculadora2.py
```

Y luego de cierto tiempo (¡oh, sorpresa, también utiliza **logging** para mostrar el progreso de la compilación pero con unos códigos no recomendables de niveles de registro -valores personalizados-) y en una carpeta **dist** encontraremos nuestro ejecutable listo para ser experimentado. Nosotros obtuvimos esto, si queréis practicad que algo parecido obtendréis:

Entonces estaremos listo para ejecutar nuestro flamante binario, nos vamos con el explorador de ficheros **Nautilus** de Ubuntu a la carpeta **dist** le damos click derecho, ejecutar y ¡oh, sorpresa! el fichero **bitacora.log** se genera y aparece... pero con cero bytes, sin nada dentro, ¿qué ha sucedido aquí?

Pues que simplemente la librería de registro abre el archivo **bitacora.log** (crea el archivo) pero la división entre cero no le permite llegar a ejecutar el grabado del mensaje, ya que las instrucciones son anidadas y primero trata de dividir y luego mostrar el mensaje, pero como se "cuelga" pues no registra nada de nada.

En este punto ya es bueno concluir algo muy cierto: el registro de errores incluso nos beneficiará al obligarnos a pensar dónde colocar los mensajes necesarios para futuras mejoras y en el caso del software libre donde TODOS podemos ser parte de un equipo de programadores esta ayuda es tremendamente bienvenida.

Otra pregunta que nos hacemos, ¿qué sucede si no establecemos un nivel de registro específico en **logging.basicConfig**? De manera predeterminada la utilería está en nivel **WARNING** y los mensajes que sean iguales o superiores a este nivel serán registrados **(WARNING, ERROR** y **CRITICAL**). No obstante vamos a dar un paso más allá en nuestros estudios y vamos a configurar para que sean nuestros propios usuarios quienes establezcan un nivel de registro *lo cual consideramos útil para ellos que NO son programadores y que tal vez necesiten cierta orientación* ***sin necesidad que ellos y/o ellas lleguen a tener que descargar el código fuente de la***

aplicación -que siempre estará al alcance por ser software libre-.

Que los usuarios y usuarias establezcan su nivel de registro.

Pasando parámetros a una aplicación desde la línea de comando.

En el mundo de Python hay varias librerías que nos permiten "pasar" parámetros hacia "lo interno" de nuestras aplicaciones, algunas de ellas son -pero no son todas-:

- **getopt** es una librería la cual se deriva de una del lenguaje C llamada, claro, **getopt()**.
- **optparse** escrita para Python pero que actualemente está "descontinuada".
- **argparse** la cual curiosamente deriva de **optaprse** pero ofrece total compatibilidad a la versión 3 -y a futuro-.

Por esa razón escogemos esta última para evitarnos dolores de cabeza a futuro.

argparse.

Introducción a argparse.

Debemos hacer una breve introducción al concepto de parámetros tanto opcionales como obligatorios. De manera general las aplicaciones corren sin ninguna isntrucción especial: escribimos el nombre del fichero, el sistema operativo revisa si es un ejecutable, o carga en memoria y ejecuta las instrucciones contenidas.

Un ejemplo sencillo es el comando para listar ficheros y directorios en una ventana terminal: **ls**. Sin más dicho comando nos muestra por pantalla los ficheros y directorios contenidos en la carpeta desde donde la ejecutamos. Si hubiera alguna carpeta y queremos saber su contenido debemos escribir **ls *nombre-de-la-carpeta*** y allí tenemos un parámetro *opcional* que le estamos pasando a la aplicación: le estamos ordenando listar el contenido de un directorio. Decimos que es opcional porque, como vimos, el comando no necesita nada para funcionar pero somos nosotros los que tenemos la necesidad de pedirle algo muy específico. Pero adicionalmente a la petición específica *queremos que nos lo muestre de una manera específica* y para poder diferenciar los nombres de las carpetas -o archivos- que pidamos de la forma como la va a presentar pues nació la idea de colocar palabras claves para diferenciar (recalcamos que estamos con el comando **ls** como ejemplo útil ya que es un comando extremadamente básico). Así podemos teclear **ls *nombre-de-carpeta* -l** para listar en modo columna o el también llmado modo largo (nombres de ficheros o directorios uno encima del otro con detalles de tamaños, fecha, atributos, etc.).

Es por esta razón que se estableció ciertas normas para pasar parámetros, en general podemos decir que:

1. Se utiliza un guion "-" como prefijo para indicar un parámetro y se acompaña generalmente con una sola letra que más que suficiente porque tenemos 54 opciones distintas (27 caracteres mayúsculas y minúsculas).
2. Como estrategia nemotécnica se utilizan dos guiones juntos "--" junto con palabras o incluso frases para que sea de manera explícita su recordación.
3. También se da el caso que a las dos opciones anteriores se le agregue *sin dejar espacios* un signo de igualdad y a continuación algún valor condicionante (que puede ser imprescindible o no).
4. Como para algunos el punto anterior no les parece elegante, también se estila colocar un espacio y a continuación algún valor condicionante.

Siguiendo con el ejemplo del comando **ls**:

Ejemplo del punto 1: comando "**ls -r**" (lista los archivos y carpetas en orden alfabético inverso, de la letra zeta hacia la letra a).

Ejemplo del punto 2: comando "**ls --reverse**" ídem al punto anterior pero más fácil de recordar y explícito para mostrar.

Ejemplo del punto 3: comando "**ls --sort**" ordena las lista de ficheros por orden de tamaño, del más grande hacia el más pequeño, pero sucede que hay muchas maneras de ordenar ese listado y si lo ejecutamos así sin más nos solicita un parámetro necesario. Podemos pedirlo por tamaño así que escribimos "**ls --sort=size**" y veremos el resultado con primero los más grandes yendo luego progresivamente hasta los más pequeños. Por cierto, este comando "largo" tiene un equivalente "corto": **ls -s**.

Ejemplo del punto 4: comando "**ls *patron-a-buscar***" como por ejemplo si queremos ver solamente los archivos que comienzen con la letra "a": "**ls a***".

Primeros pasos con argparse.

Para comenzar a utilizar **argparse** en nuestro programa, simplemente hagamos un fichero nuevo y le colocamos los siguiente:

```
import argparse  analizador = argparse.ArgumentParser(description="Progra
ma demostrativo de argparse")  analizador.parse_args()
```

Al salir lo nombramos como **mi_programa.py** y lo ejecutamos con **python3 mi_programa.py** y, por supuesto, no hace nada de nada ya que no le escribimos ningún código adicional. *Pero ahora vamos a ejectuarlo acompañado de un parámetro como lo es el siguiente:* **python3**

mi_programa.py -h y obtendremos la siguiente salida:

```
usage: mi_programa.py [-h]    Programa demostrativo de argparse.    optio
nal arguments:   -h, --help show this help message and exit
```

Como vemos todo viene preconfigurado para utilizar el idioma inglés por defecto, pero pronto podremos darle un uso mejor para orientarlo hacia el idioma castellano en un tutorial dedicado al tema. Notad que especifica que el "parámetro largo" para ayuda es **--help**. Adicionalmente, cualquier otro parámetro que le pasemos manifestará desconocerlo -no hemos programado nada áun, por ahora-. Por lo pronto ya cumplimos con iniciar el uso práctico de **argparser**, continuemos aprendiendo.

Agregando otro argumento opcional.

Como vemos **argparse** tiene al menos un argumento establecido por defecto, el de ayuda [el cual es opcional, está mostrado entre corchetes], y ahora nosotros vamos a agregarle nuestro propio argumento para establecer el nivel de registro de eventos. Para ello especificaremos la palabra clave **-log_lev** acompañado del nivel que deseemos establecer, a continuación lo pasamos por una serie de tamices con la instrucción condicional **if~elif~else** y si coincide mostramos por pantalla la opción elegida:

```
import argparse  analizador = argparse.ArgumentParser("Programa demostrat
ivo de argparse")  analizador.add_argument("-log_lev", help="Utilice DEBU
G, INFO, WARNING, ERROR o$  analizador.parse_args()  argumentos = analiza
dor.parse_args()  if argumentos.log_lev == 'DEBUG':   print("DEBUG")  eli
f argumentos.log_lev == 'INFO':   print("INFO")  elif argumentos.log_lev
== 'WARNING':   print("WARNING")  elif argumentos.log_lev == 'ERROR':   p
rint("ERROR")  elif argumentos.log_lev == 'CRITICAL':   print("CRITICAL")
  else:   print("Opcion no válida de nivel de registro de eventos.")
```

Como vemos esto simplemente es el armazón para el manejo del pase de parámetros a nuestro programa didáctico para el registro de eventos con la utilería **logging**.

En este punto corred vuestro programa varias veces, experimentad con el pase de parámetros para que luego continuemos con el último paso de este tutorial: la fusión de **logging** con **argparse**.

Uniendo "logging" con "argparse".

Ya para finalizar unimos el código de ambos ejemplos y la idea es la siguiente: al utilizarse sin parámetros se establece el nivel de registro en **WARNING** que es el nivel predeterminado. Si se utiliza el parámetro **-log_lev** sin acompañarlo de valor alguno, la librería **argparse** se encargará debidamente de orientar al usuario sobre las opciones disponibles. Si el usuario usa alguna opción disponible válido pues se establece debidamente el nivel de registro correspondiente.

Queda para vuestra práctica el permitir que los usuarios especifiquen un nombre de archivo para el registro de eventos.

Acá tenemos el código final, espero os haya servido para aprender algo nuevo sobre el lenguaje Python.

```
import logging  import argparse    analizador = argparse.ArgumentParser("
Programa demostrativo de argparse")  analizador.add_argument("-log_lev",
help="Utilice DEBUG, INFO, WARNING, ERROR o CRITICAL")  analizador.parse_
args()  argumentos = analizador.parse_args()  if argumentos.log_lev == 'D
EBUG':   logging.basicConfig(level=logging.DEBUG)  elif argumentos.log_le
v == 'INFO':   logging.basicConfig(level=logging.INFO)  elif argumentos.l
og_lev == 'WARNING':   logging.basicConfig(level=logging.WARNING)  elif a
rgumentos.log_lev == 'ERROR':   logging.basicConfig(level=logging.ERROR)
 elif argumentos.log_lev == 'CRITICAL':   logging.basicConfig(level=loggi
ng.CRITICAL)  else:   logging.basicConfig(level=logging.WARNING)  # Opció
n no válida, se establece nivel WARNING por defecto.      logging.basicCo
nfig(   filename="bitacora.log",   format='%(asctime)s - %(name)s - %(lev
elname)s - %(message)s',   datefmt='%d/%m/%Y %I:%M:%S %p'  )    bita_sum
= logging.getLogger("Sum")  bita_res = logging.getLogger("Res")  bita_mul
 = logging.getLogger("Mul")  bita_div = logging.getLogger("Div")    class
 calculadora():    def __init__(self, nombre):   self.nombre = nombre   l
ogging.debug("\nCalculadora modelo {} (encendida).".format(self.nombre))
   def sumar(self, a=0, b=0):   bita_sum.info("Suma a={} b={} a+b={}".for
mat(a,b,a+b))    def restar(self, a=0, b=0):   bita_res.info("Resta a={}
b={} a-b={}".format(a,b,a-b))    def multiplicar(self, a=0, b=0):   bita_
mul.info("Multiplicación a={} b={} a*b={}".format(a,b,a*b))    def dividi
r(self, a=0, b=1):   if (b==0):   bita_div.error("Alerta: el divisor debe
 ser distinto a cero.")   else:   bita_div.info("División a={} b={} a/b={
}".format(a,b,a/b))    calc = calculadora("A1")  calc.sumar( 45,5)  calc.
restar(45,5)  calc.multiplicar(45,5)  calc.dividir(45,0)
```

Fuentes consultadas:

En idioma inglés:

- "How To Use Logging in Python 3" by Lisa Tagliaferri.
- "Basic logging tutorial" by Vinay Sajip .
- "Python __init__ and self what do they do?".
- "PyInstaller".
- "Argparse Tutorial" by Tshepang Lekhonkhobe.