

# Clase 6

## Programar en Python

### Errores comunes y buenas prácticas

En esta clase romperemos todos los códigos de ejemplo de la clase anterior para analizar uno por uno los posibles errores que podemos cometer al escribir nuestros programas. Además, ampliaremos sobre las buenas prácticas en programación y analizaremos diferentes casos de aplicación.

#### Rompiendo códigos

Luego de haber pasado por la clase anterior en la cual desarrollamos y testeamos varios códigos diferentes, seguramente nos topamos más de una vez con algún error que nos impidió ejecutar correctamente el código a la primera. De hecho, en el mundo de la programación, es una utopía pensar que el código que escribimos puede funcionar sin ningún error al primer intento.

En base a eso, retomaremos los códigos que desarrollamos e iremos recorriendo todos los posibles errores que nos pueden surgir a la hora de querer correrlos, intérprete de por medio.

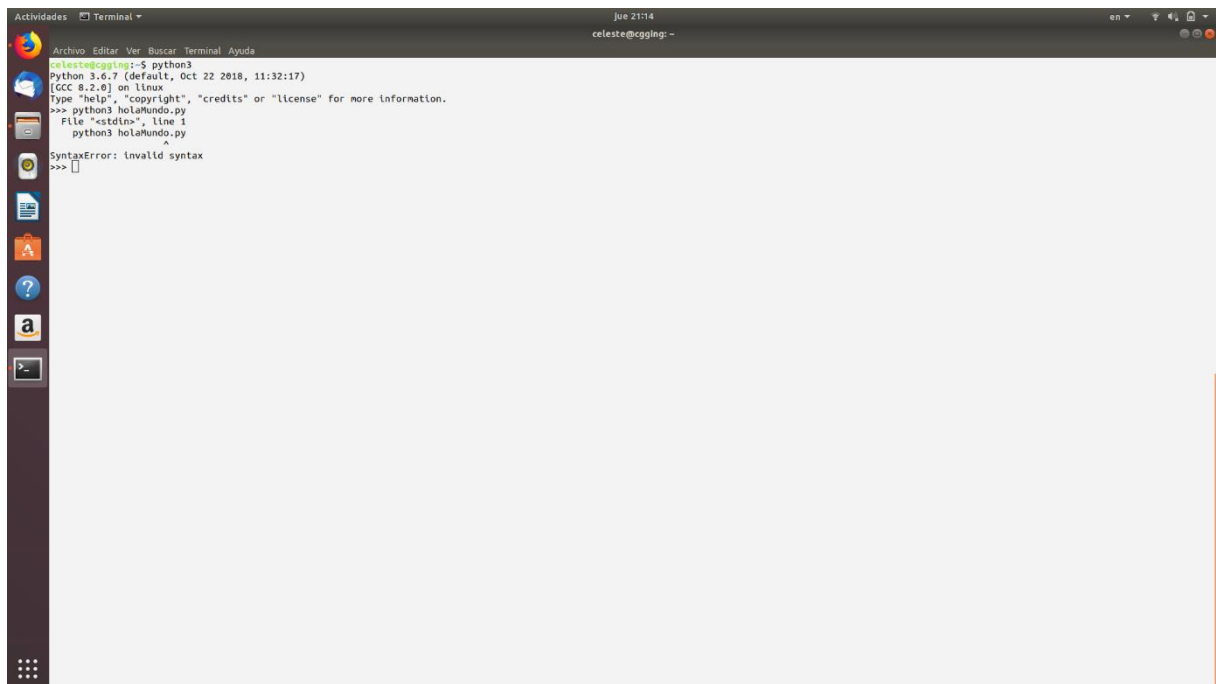
#### Errores al intentar ejecutar un script de python

Uno de los principales errores que podemos cometer al querer comenzar a utilizar archivos para realizar nuestros códigos, es querer abrir estos archivos desde el intérprete interactivo.

```
>>> python3 holaMundo.py
```

Esto nos resultara en un bonito código de error que podemos ir sumando a la lista de errores que veremos en esta clase.

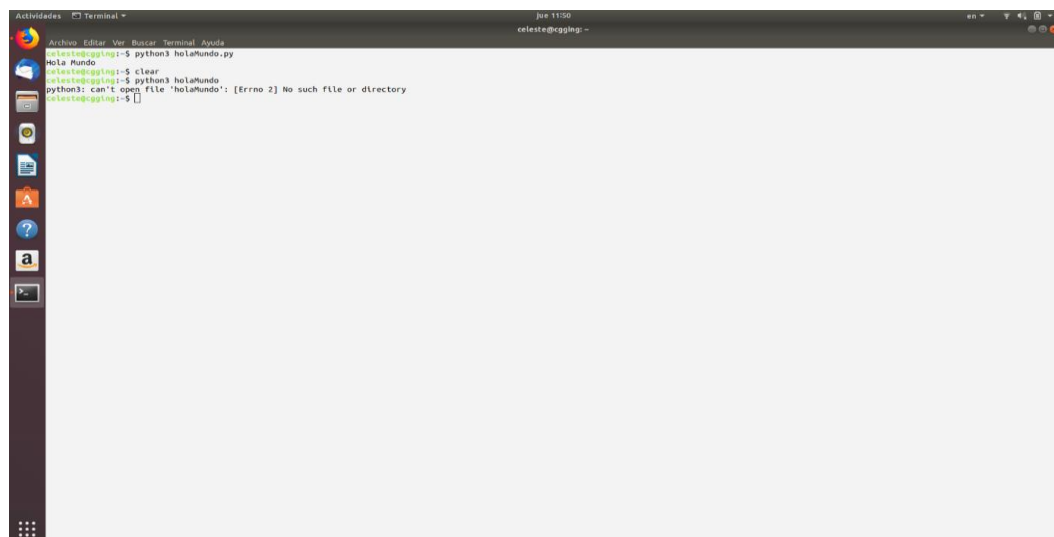
Siempre que queramos ejecutar un archivo, (de ahora en adelante script) de python, tenemos que hacerlo desde la consola y no desde el intérprete interactivo.

A terminal window titled 'Terminal' with the user 'celeste@ggling: ~'. The prompt is 'celeste@ggling:~\$'. The user enters 'python3', which shows 'Python 3.6.7 (default, Oct 22 2018, 11:32:17) [GCC 8.2.0] on linux'. The user then enters 'python3 holaMundo.py', which shows 'File "<stdin>", line 1' and 'python3 holaMundo.py'. The user enters 'python3 holaMundo.py' again, which results in a 'SyntaxError: invalid syntax' message. The terminal window has a sidebar with icons for 'Archivos', 'Editor', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'.

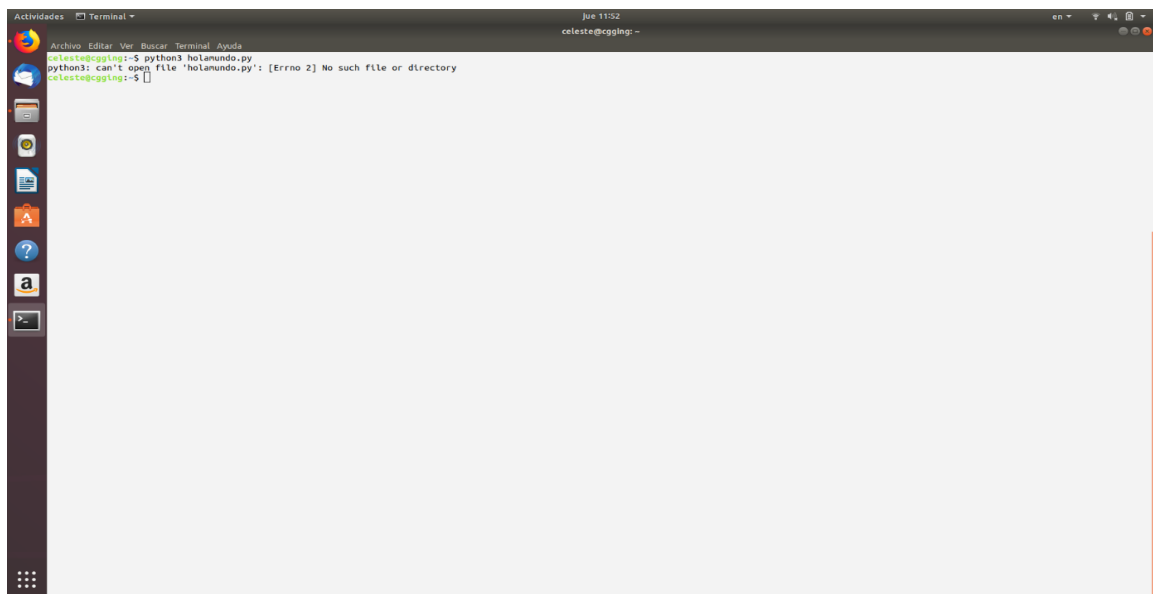
## Hola Mundo “cruel”.

El primer ejemplo que vimos fue el mundialmente famoso y nunca bien ponderado “Hola Mundo”. En esta clase nos referiremos más bien al “Mundo cruel” ya que, si bien este ejemplo es muy sencillo por tratarse de una sola línea de código, es probable que no hayamos podido correrlo a la primera con éxito:

¿Con qué nos podemos llegar a chocar en este caso? Bueno, para comenzar en este ejemplo habíamos introducido el uso de archivos para almacenar el código en lugar de escribirlo directamente en el intérprete interactivo, con lo cual un error común es omitir la extensión del archivo al querer ejecutar el código:

A terminal window titled 'Terminal' with the user 'celeste@ggling: ~'. The prompt is 'celeste@ggling:~\$'. The user enters 'python3 holaMundo.py', which shows 'Hola Mundo'. The user then enters 'python3 holaMundo', which results in a 'python3: can't open file 'holaMundo': [Errno 2] No such file or directory' message. The terminal window has a sidebar with icons for 'Archivos', 'Editor', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'.

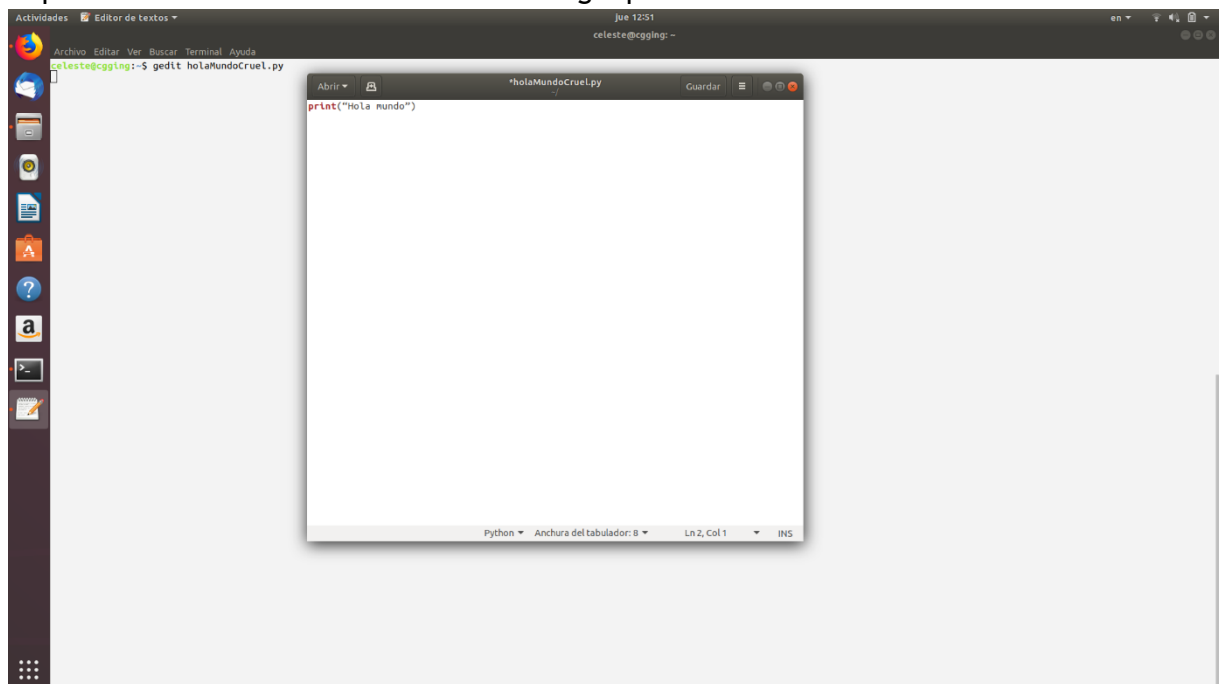
Otro error, puede ser omitir mayúsculas en el nombre, recordemos que nuestro archivo se llamaba holaMundo.py :



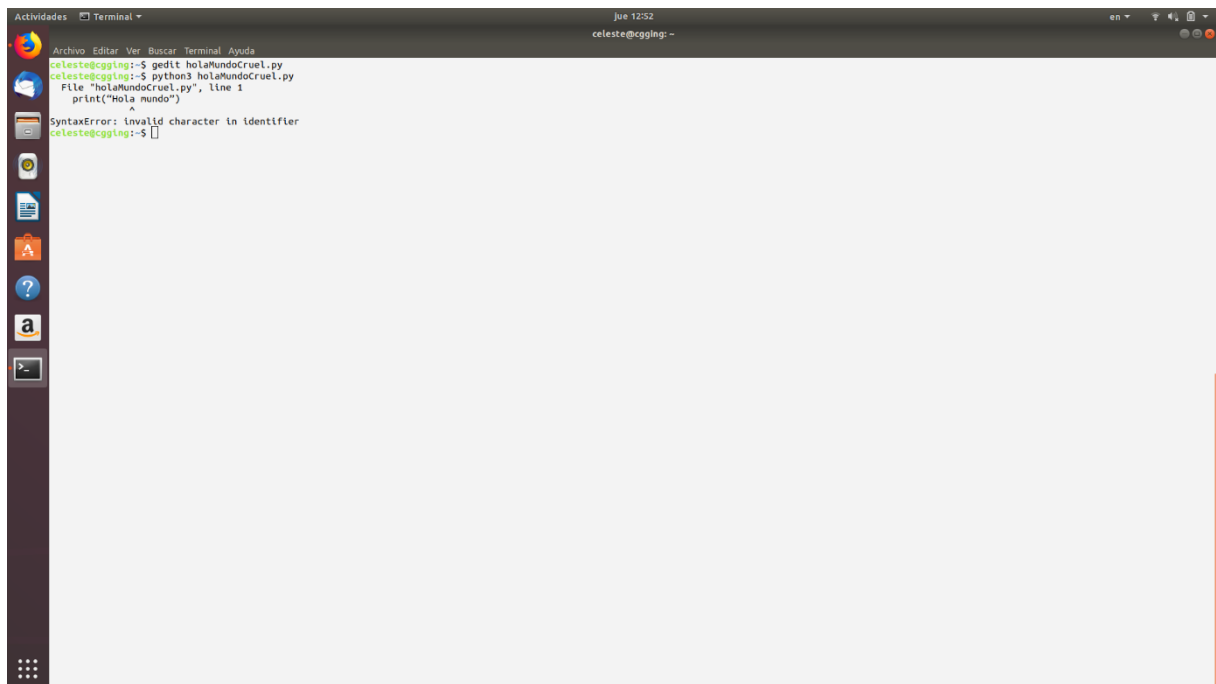
Por último, un error no tan trivial y que puede llegar a complicarnos bastante, si no sabemos de qué se trata, tiene que ver con el copy/paste del código que queremos ejecutar desde algún documento tal como un archivo pdf hacia el archivo .py que estamos escribiendo.

*print("Hola mundo")*

Copiando desde un archivo de texto el código que acabamos de escribir en este documento

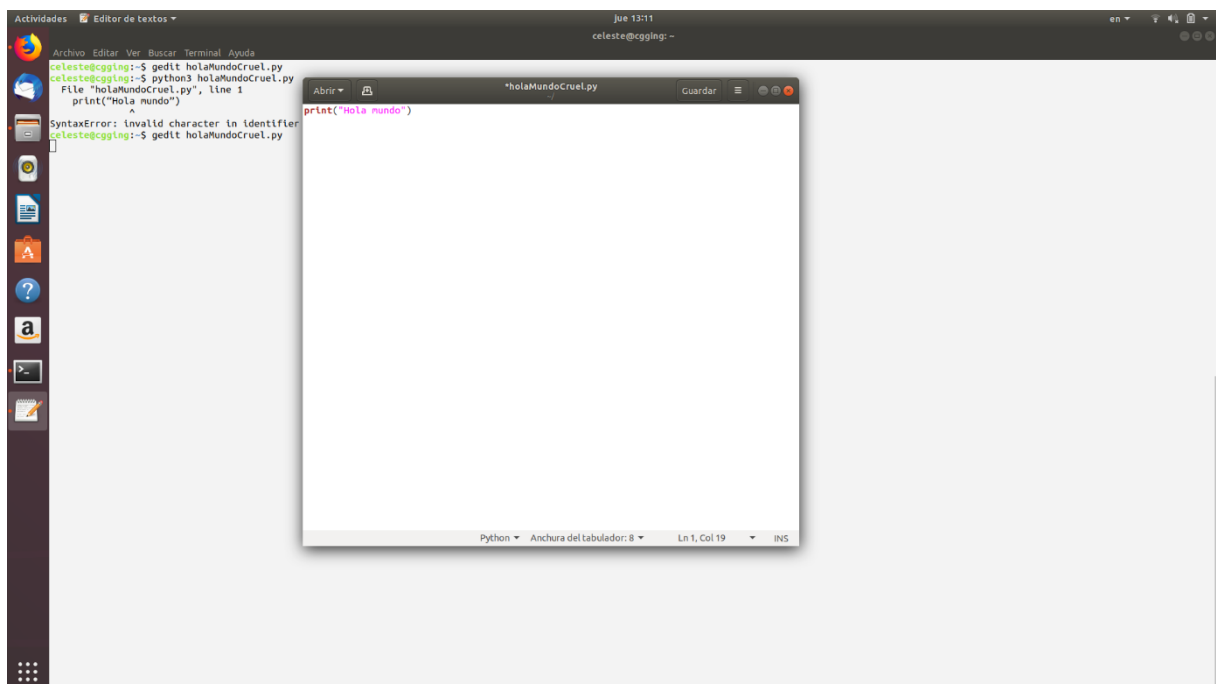


a simple vista si es uno de los primeros códigos que realizamos, o si utilizamos un editor de texto plano, no seremos capaz de notar que este código no funcionara... pero al correrlo desde la consola:



```
Actividades Terminal
Jue 12:52
celeste@cgging: ~
Archivo Editor Ver Buscar Terminal Ayuda
celeste@cgging:~$ gedit holaMundoCruel.py
celeste@cgging:~$ python3 holaMundoCruel.py
File "holaMundoCruel.py", line 1
  print("Hola mundo")
      ^
SyntaxError: invalid character in identifier
celeste@cgging:~$
```

El código no se ejecuta y nos aparece un error de sintaxis... ¿qué ocurrió? el código de las comillas del texto que copiamos es diferente a las comillas que entiende python, y si utilizamos linux o algún editor de código que nos identifique las diferentes partes del código mediante colores, entonces podremos evidenciar que al reemplazar las comillas en el documento, el texto del print cambiara de color.



```
Actividades Editor de textos
Jue 13:11
celeste@cgging: ~
Archivo Editor Ver Buscar Terminal Ayuda
celeste@cgging:~$ gedit holaMundoCruel.py
celeste@cgging:~$ python3 holaMundoCruel.py
File "holaMundoCruel.py", line 1
  print("Hola mundo")
      ^
SyntaxError: invalid character in identifier
celeste@cgging:~$ gedit holaMundoCruel.py
```

Abrir Guardar

\*holaMundoCruel.py

```
print("Hola mundo")
```

Python Anchura del tabulador: 8 Ln 1, Col 19 INS

Si ahora volvemos a correr el código, el error se habrá solucionado.

## Suma

Analicemos el código que realizamos y comencemos a ver los posibles puntos de falla:

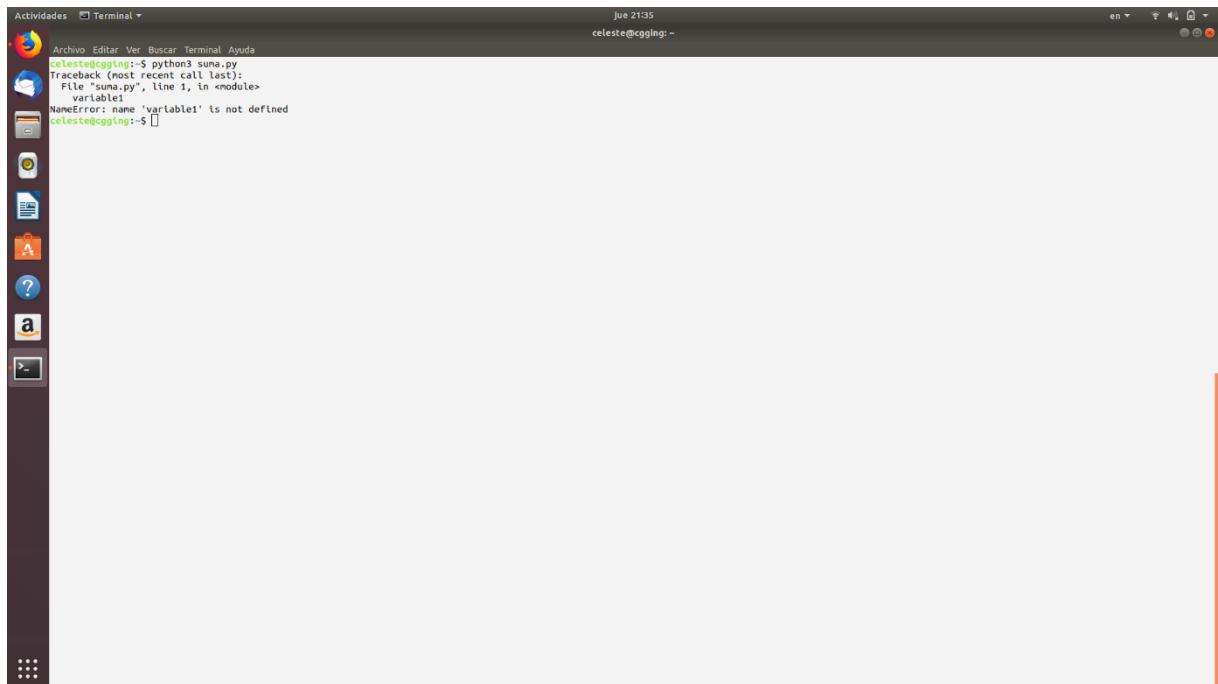
```
variable1=3
variable2=5
suma= variable1 + variable2
print("la suma de la variable 1=" , variable1, " y la variable 2=", variable2, "es igual a ",
suma)
```

comencemos:

El primer error que podríamos cometer al pensar este código, y si venimos de programar en algún otro lenguaje es intentar declarar las variables sin inicializarlas:

```
variable1
variable2
```

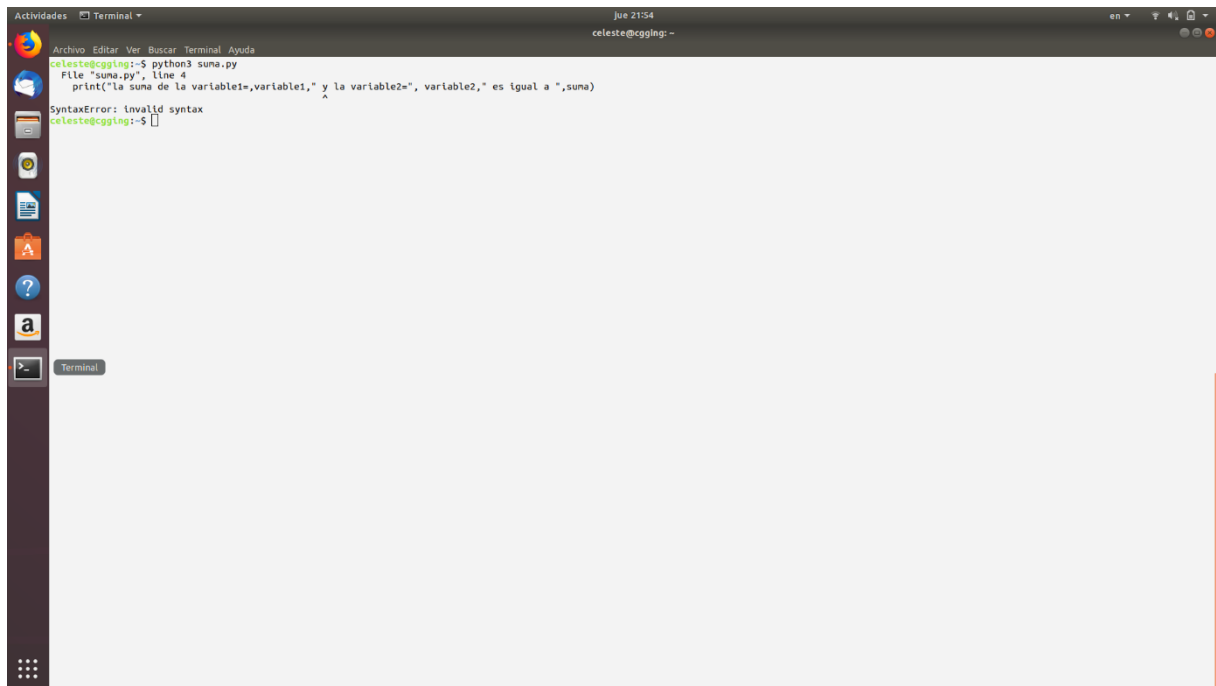
Esto producirá un error tal como habíamos mencionado en capítulos anteriores. Recordemos que las variables en Python se definen al asignarles un valor, ya que el tipado es dinámico.

A screenshot of a Linux terminal window. The title bar says "Actividades Terminal". The terminal shows the command "python3 suma.py" being executed. The output is a "NameError: name 'variable1' is not defined". The terminal prompt is "celeste@cggling: ~". The background of the terminal is dark gray, and the text is light gray. The terminal is open on a desktop environment with a sidebar on the left showing various application icons like a file manager, web browser, and terminal.

Otro lugar en donde podemos cometer errores es en el print. En particular en este caso queremos mostrar mucha información combinada, tanto de variables como de texto y sería algo normal que alguna comilla se nos pase de largo:

```
print("la suma de la variable 1= , variable1, " y la variable 2=", variable2, "es igual a ",
suma)
```

En este caso, faltaría cerrar la comilla luego del '=', esta omisión producirá un error.



The screenshot shows a terminal window with the following content:

```
Actividades Terminal
Jue 21:54
celeste@cgging: ~
File "suma.py", line 4
    print("la suma de la variable1=variables, y la variable2=", variable2, " es igual a ", suma)
SyntaxError: invalid syntax
celeste@cgging: ~
```

## Excepciones

Como hemos mencionado brevemente en clases anteriores y como hemos visto particularmente en el ejemplo “Suma con vitaminas”, es posible que nuestro código falle, no por errores de sintaxis, sino por errores en tiempo de ejecución.

Particularmente en el ejemplo que mencionamos, el error que se producía tenía que ver con el tipo de datos que ingresara el usuario. Estos errores se conocen como “excepciones” y es posible tomar recaudos para manejarlos.

Veamos algunos ejemplos de errores en tiempo de ejecución:

```
>>> 10 * (1/0)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: division by zero
```

```
>>> 4 + spam*3
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

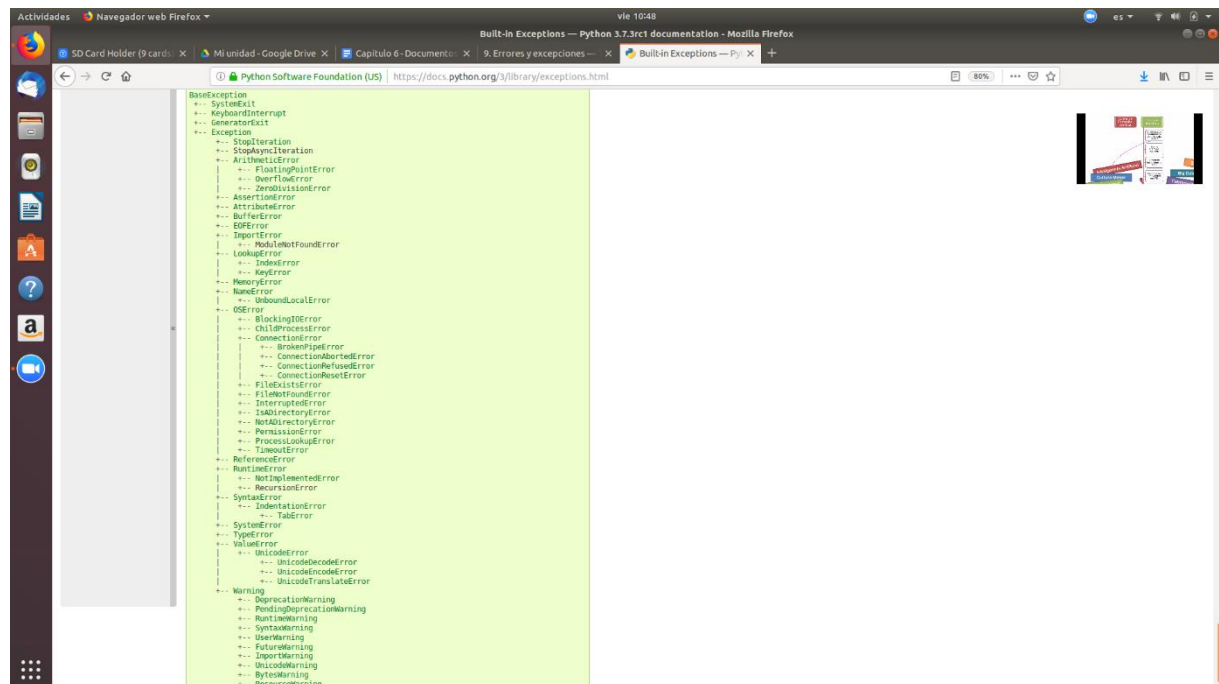
```
TypeError: Can't convert 'int' object to str implicitly
```

La información que nos provee el intérprete en la última línea de los mensajes de error se corresponde con que ha ocurrido. Como podemos suponer, las excepciones tienen diferente tipo y este se muestra como parte del mensaje de error (ZeroDivisionError, NameError, TypeError, etc). El nombre del tipo de excepción mostrada coincide con el nombre de la excepción predefinida que ocurrió, la coincidencia en la descripción y el nombre se utiliza

por un tema de convención, no obstante nosotros podemos definir nuestras propias excepciones y en ese caso podríamos mostrar un texto diferente del nombre de la excepción por pantalla, aunque una buena práctica es mantener iguales el texto mostrado y el nombre para facilitar la tarea de depuración. Los nombres de las excepciones estándar son identificadores incorporados al intérprete (no son palabras clave reservadas).

El resto de la línea provee un detalle basado en el tipo de la excepción y qué la causó.

La parte anterior del mensaje de error muestra el contexto donde la excepción sucedió, en la forma de un trazado del error listando líneas fuente; sin embargo, no mostrará líneas leídas desde la entrada estándar.



**Listado de excepciones integradas en el intérprete de python**

## Manejando excepciones

Si las excepciones son errores en tiempo de ejecución, pero tienen un nombre definido, es natural pensar que conociendo el posible error que ocurrirá podremos manejar esa situación.

Veamos un ejemplo:

```
while True:
    try:
        valor1 = int(input("Por favor ingrese el primer numero: "))
        break
    except ValueError:
        print("Oops! No era válido. Intente nuevamente...")
```

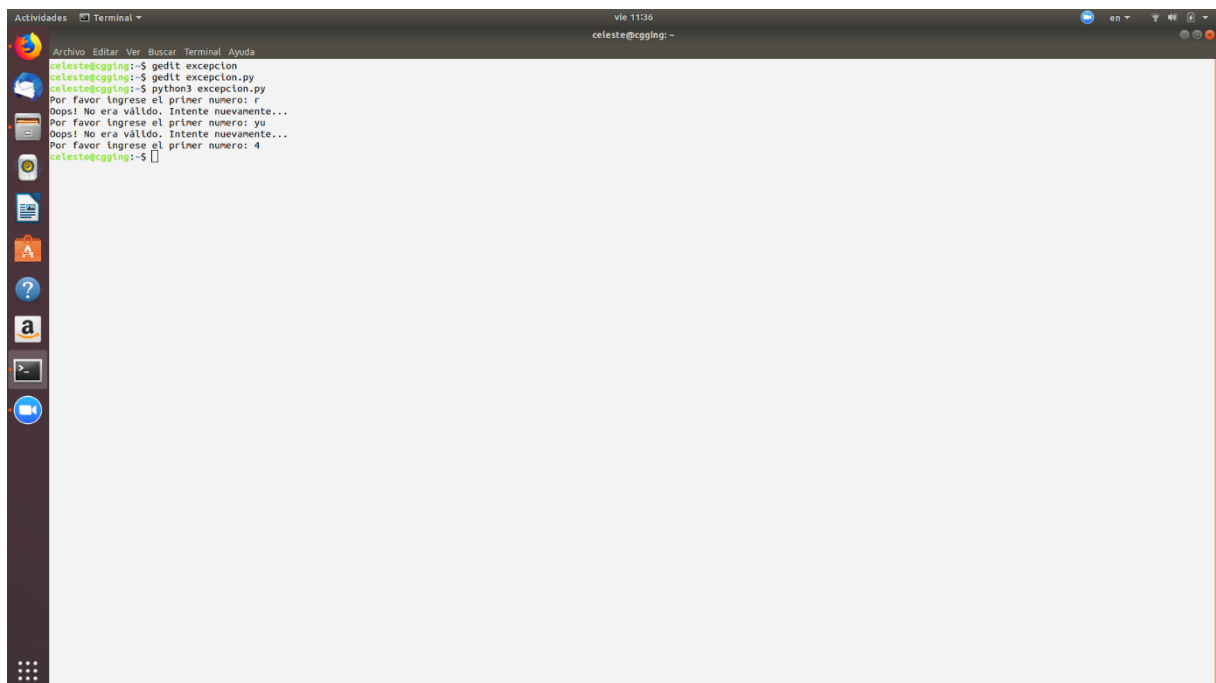
En el ejemplo anterior podemos ver que se introdujo una nueva declaración que encierra al bloque que podría tener el error en tiempo de ejecución, como habíamos visto en el ejemplo

de Suma con Vitaminas, el error se producía al querer ingresar un dato no numérico por teclado, por lo tanto, deberemos encerrar con 'try' esa porción de código.

El bloque try lo que hace es intentar la ejecución del código que encierra, si no se produce ninguna excepción durante la ejecución, entonces el bloque except se saltea y se continua con el flujo normal del programa.

Si en cambio se produce una excepción, entonces se interrumpirá el flujo del programa y se saltará el resto del código hasta pasar al bloque except, el cual se ejecutará en el caso de que la excepción ocurrida coincida con la manejada por este bloque, y luego es retornada nuevamente al comienzo del bloque try.

En el caso de que ocurriera una excepción diferente de la manejada por el bloque except, entonces se busca si hay alguna declaración try de orden superior (es decir algún try anidado por encima del bloque que produjo la excepción), si no se encuentra ningún bloque que la maneje, estamos frente a una excepción no manejada y la ejecución se frena con el mensaje de error.



```
celeste@cggling:~$ gedit excepcion
celeste@cggling:~$ gedit excepcion.py
celeste@cggling:~$ python3 excepcion.py
Por favor ingrese el primer numero: r
Oops! No era válido. Intente nuevamente...
Por favor ingrese el primer numero: yu
Oops! No era válido. Intente nuevamente...
Por favor ingrese el primer numero: 4
celeste@cggling:~$
```

Una declaración try puede tener más de un except, para especificar manejadores para distintas excepciones. A lo sumo un manejador será ejecutado. Sólo se manejan excepciones que ocurren en el correspondiente try, no en otros manejadores del mismo try. Un except puede nombrar múltiples excepciones usando paréntesis, por ejemplo:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Habiendo comprendido el uso de excepciones, manejaremos las mismas en el ejemplo de “Suma con Vitaminas”:

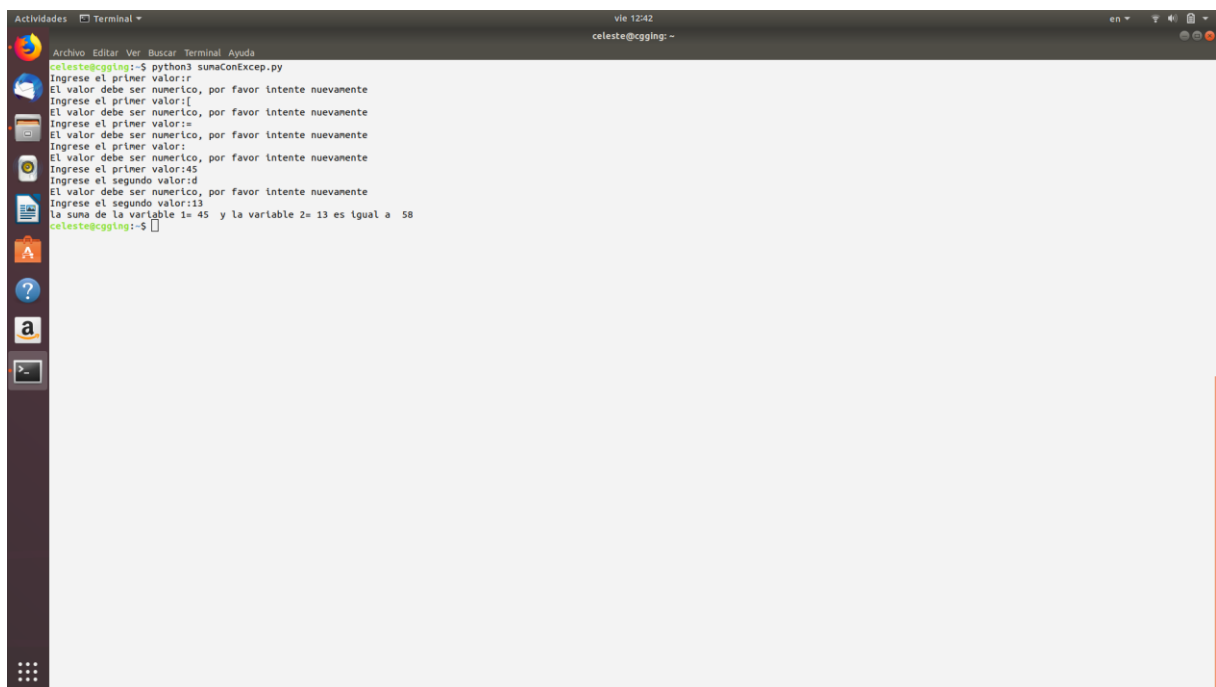


```

while True:
    try:
        variable1=int(input("Ingrese el primer valor:"))
        break
    except ValueError:
        print("El valor debe ser numerico, por favor intente nuevamente")
while True:
    try:
        variable2=int(input("Ingrese el segundo valor:"))
        break
    except ValueError:
        print("El valor debe ser numerico, por favor intente nuevamente")
suma= variable1 + variable2
print("la suma de la variable 1=" , variable1, " y la variable 2=", variable2, "es igual a ",
suma)

```

Ahora que hemos manejado la excepción que se produce al intentar ingresar un valor no numérico, hemos limitado considerablemente las fallas que se podrían producir en tiempo de ejecución con nuestro código.



```

celeste@ggling:~$ python3 sumaConExcep.py
Ingrese el primer valor:r
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:[
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:=
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:45
Ingrese el segundo valor:d
El valor debe ser numerico, por favor intente nuevamente
Ingrese el segundo valor:13
la suma de la variable 1= 45 y la variable 2= 13 es igual a 58
celeste@ggling:~$

```

Pero analicemos un poco lo que acabamos de implementar:

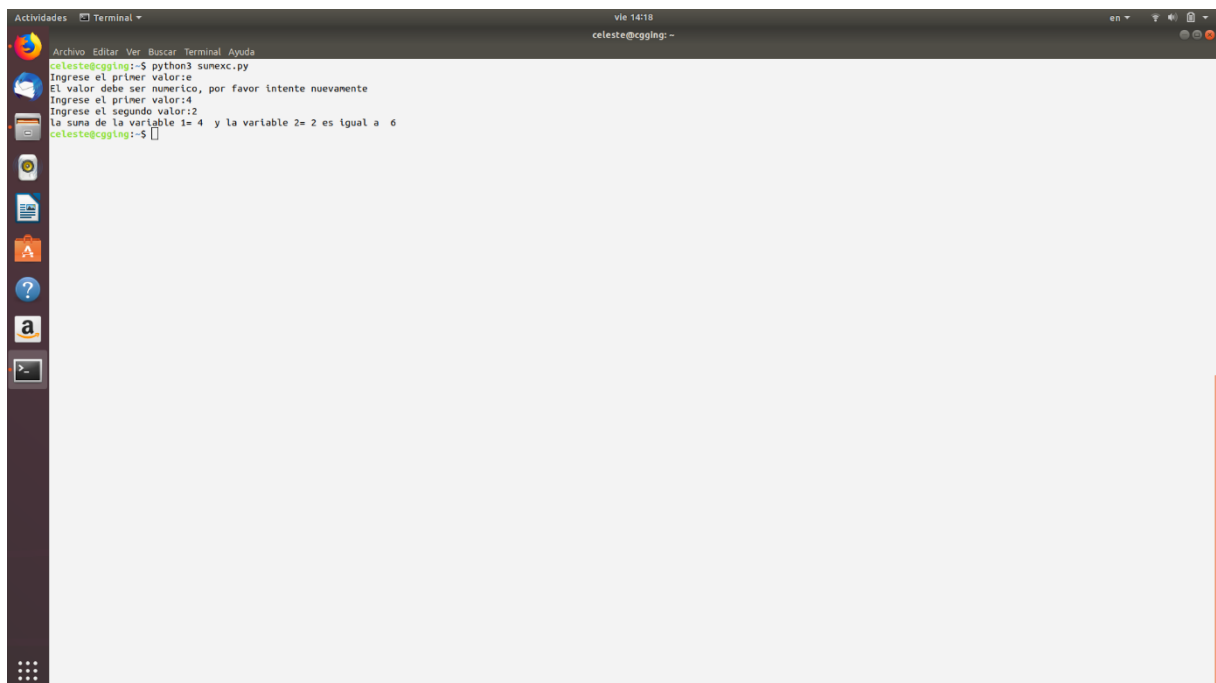
Realizamos un while True y un try por cada variable, seguramente se nos ocurriría pensar que podríamos simplificarlo utilizando un solo bucle, ya que, al tratarse de la misma posible excepción para las dos variables, si el usuario se equivoca al ingreso de cualquiera de las dos se manejaría la excepción de igual manera... veamos que sucede si implementamos nuestro código con un solo while:

```

while True:
    try:
        variable1=int(input("Ingrese el primer valor:"))
        variable2=int(input("Ingrese el segundo valor:"))
        break
    except ValueError:
        print("El valor debe ser numerico, por favor intente nuevamente")
suma= variable1 + variable2
print("la suma de la variable 1=" , variable1, " y la variable 2=", variable2, "es igual a " ,
suma)

```

Reacomodamos el código, utilizando el mismo while para ambas variables, como podemos ver queda mucho más corto y limpio. Hagamos algunas pruebas y veamos qué diferencias encontramos en su ejecución:



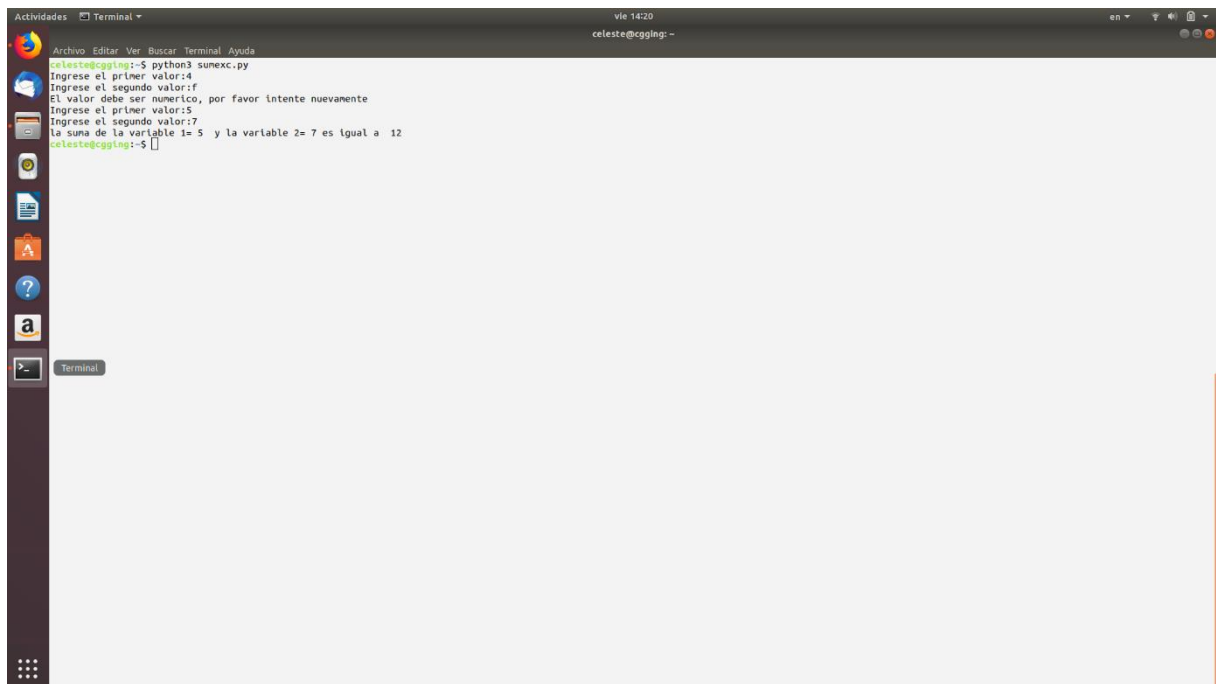
```

Actividades Terminal
vie 14:18
celeste@eggleg: ~
Archivo Editar Ver Buscar Terminal Ayuda
celeste@eggleg:~$ python3 sumexc.py
Ingrese el primer valor:e
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:4
Ingrese el segundo valor:2
la suma de la variable 1= 4 y la variable 2= 2 es igual a 6
celeste@eggleg:~$

```

Como podemos observar en la imagen anterior, si ingresamos incorrectamente el primer valor, nos pide que ingresemos un valor numérico y al cumplir con el tipo de dato, pasamos al ingreso del segundo valor sin inconvenientes.

Veamos ahora que pasa si fallamos en el ingreso del segundo valor:



```
Actividades Terminal + vie 14:20 celeste@ggling: ~
Archivo Editor Ver Buscar Terminal Ayuda
celeste@ggling:~$ python3 sumexc.py
Ingrese el primer valor:4
Ingrese el segundo valor:5
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:5
Ingrese el segundo valor:7
La suma de la variable 1= 5 y la variable 2= 7 es igual a 12
celeste@ggling:~$
```

Como podemos ver en la imagen anterior, si fallamos en el ingreso del segundo valor, tendremos que volver a ingresar ambos valores, ya que una vez ejecutado el bloque 'except', se volverá a ejecutar el código nuevamente desde el comienzo del bloque 'try'.

La primera opción que habíamos mostrado maneja el error de cada variable por separado y de esa forma si nos equivocamos en el ingreso del segundo valor, nos pedirá que reingresemos ese último número sin perder el primero.

En este ejemplo en particular, solo se pide el ingreso de dos valores, por lo que no resulta muy costoso para el usuario realizar el reingreso de los valores, diferente sería el caso si pidiéramos el ingreso de 10 valores al usuario...

Para ampliar un poco más el tema de excepciones, veamos el siguiente video

<https://youtu.be/2MaAs7XU2T0>

## Obtener el mayor

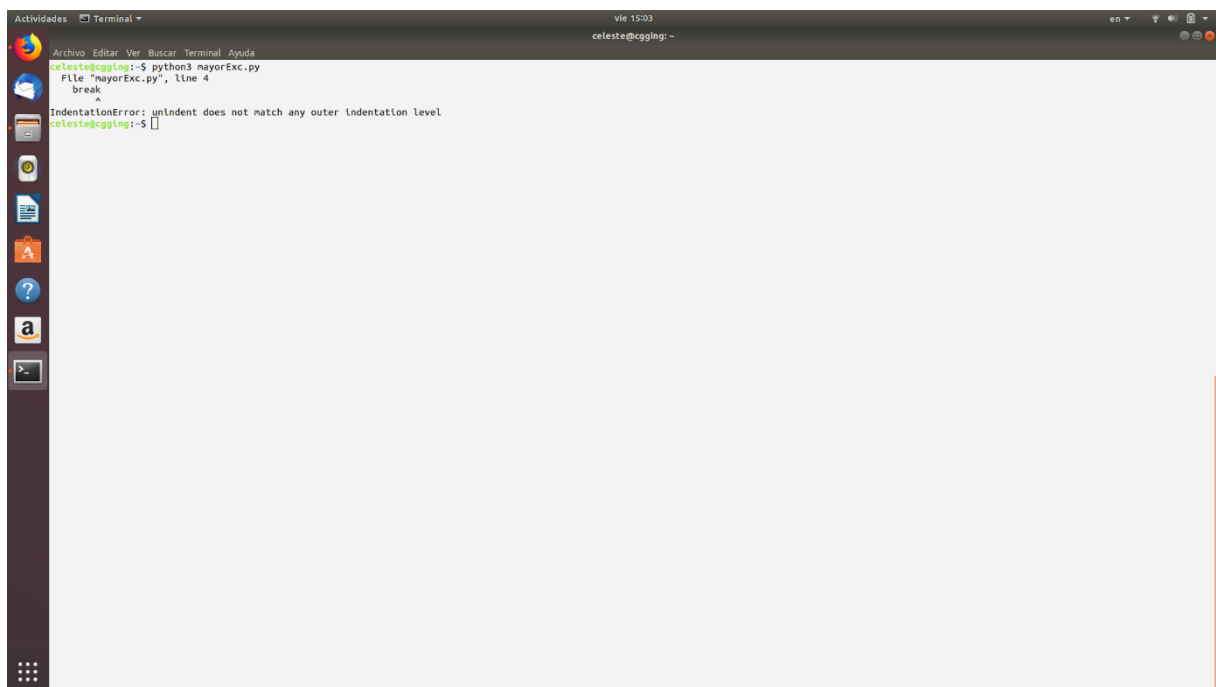
En este ejemplo, nuevamente se nos presenta el problema del ingreso del valor por teclado, acá además podemos hacer mención de algunos errores comunes de sintaxis, tal como puede ser omitir los ':' luego del if o del else, o equivocarnos en la indentación:

```

while True:
    try:
        variable1=int(input("Ingrese el primer valor:"))
        break
    except ValueError:
        print("El valor debe ser numérico, por favor intente nuevamente")
while True:
    try:
        variable2=int(input("Ingrese el segundo valor:"))
        break
    except ValueError:
        print("El valor debe ser numérico, por favor intente nuevamente")
if (variable1>variable2)
    mayor=variable1
else:
    mayor=variable2
print("El mayor valor ingresado es:",mayor)

```

si intentamos ejecutar el código anterior, pasara lo siguiente:



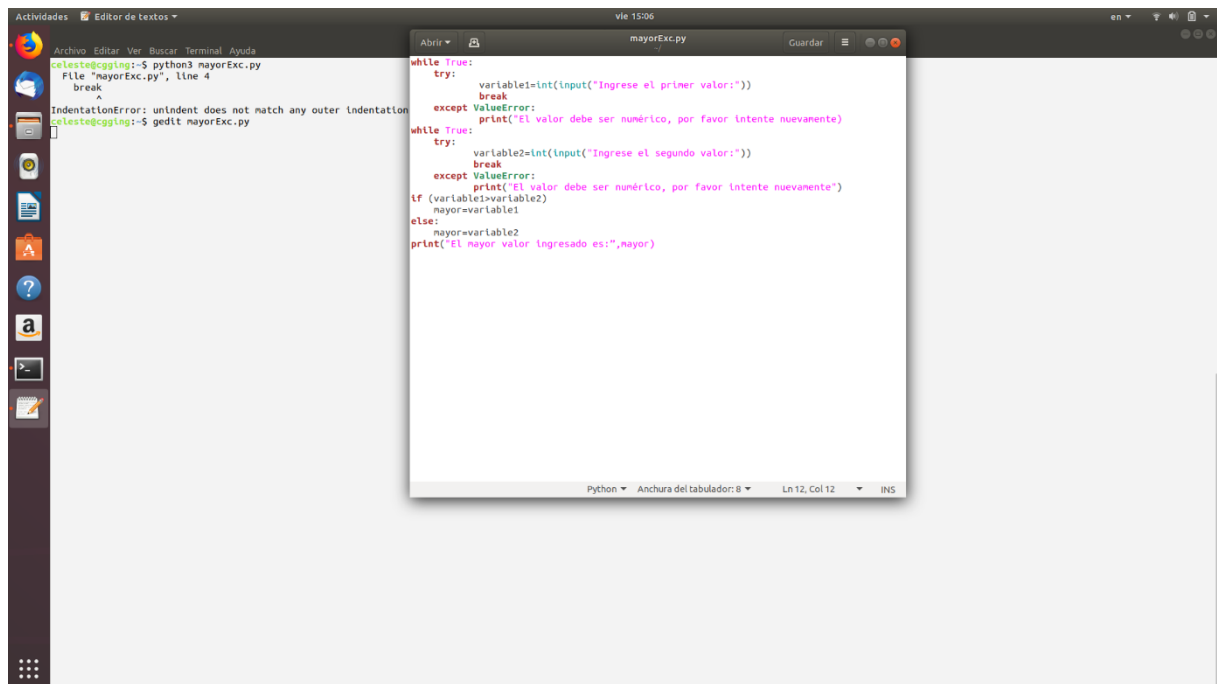
```

celeste@cgglg:~$ python3 mayorExc.py
File "mayorExc.py", line 4
    break
    ^
IndentationError: unindent does not match any outer indentation level
celeste@cgglg:~$

```

Vemos que el primer problema con el que nos encontramos es un error de indentación, y también podemos observar que el intérprete nos indica en donde se está produciendo dicho error, en este caso como nosotros podemos elegir la cantidad de espacios que utilizaremos para indentar un bloque, el intérprete asume que trabajaremos con la cantidad de espacios que definimos en la declaración de la variable1 y por ese motivo el error se aprecia en 'break'.

Este error se soluciona editando el archivo y revisando la indentación.

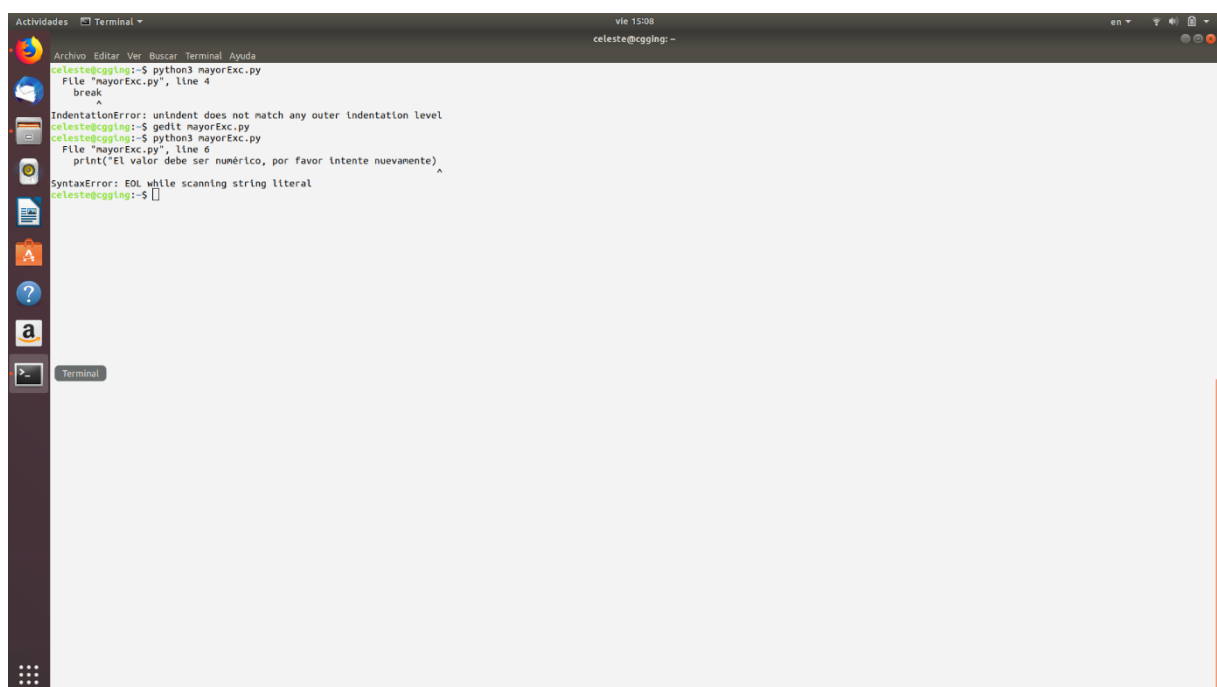


The screenshot shows a code editor window titled 'mayorExc.py'. The code is as follows:

```
while True:
    try:
        variable1=int(input("Ingrese el primer valor:"))
        break
    except ValueError:
        print("El valor debe ser numérico, por favor intente nuevamente")
while True:
    try:
        variable2=int(input("Ingrese el segundo valor:"))
        break
    except ValueError:
        print("El valor debe ser numérico, por favor intente nuevamente")
if (variable1>variable2)
    mayor=variable1
else:
    mayor=variable2
print("El mayor valor ingresado es:",mayor)
```

The error message in the terminal is: `IndentationError: unindent does not match any outer indentation level` at line 4.

Al volver a ejecutar el script de python tras corregir el error de indentación, nos encontramos con otro error, esta vez en el print:



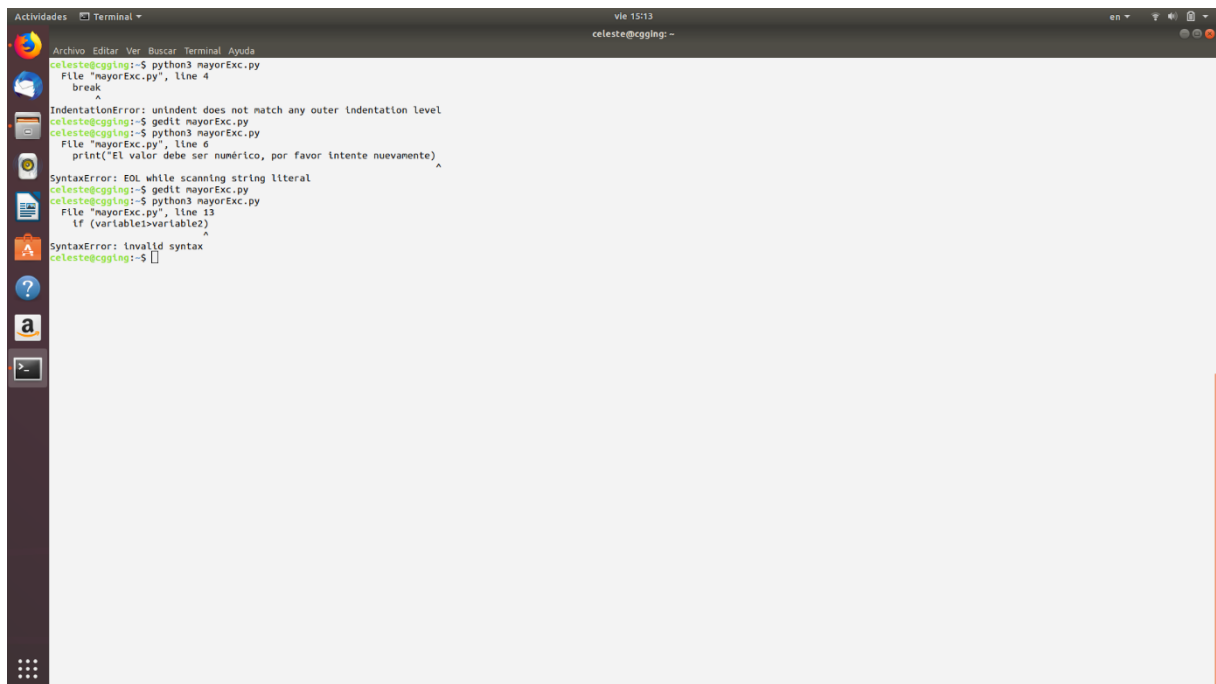
The screenshot shows a terminal window with the following commands and output:

```
celestecg@cel:~$ python3 mayorExc.py
File "mayorExc.py", line 4
    break
    ^
IndentationError: unindent does not match any outer indentation level

celestecg@cel:~$ gedit mayorExc.py
celestecg@cel:~$ python3 mayorExc.py
File "mayorExc.py", line 6
    print("El valor debe ser numérico, por favor intente nuevamente")
    ^
SyntaxError: EOL while scanning string literal

celestecg@cel:~$
```

el intérprete nos indica que se encontró un final de línea antes de que finalizara el texto a imprimir, esto se debe a la omisión de las comillas de cierre del texto. Al corregir este error y volver a ejecutar el script, esta vez podemos observar el siguiente error:



```
celeste@cggling:~$ python3 mayorExc.py
File "mayorExc.py", line 4
    break
    ^
IndentationError: unindent does not match any outer indentation level

celeste@cggling:~$ gedit mayorExc.py
celeste@cggling:~$ python3 mayorExc.py
File "mayorExc.py", line 6
    print("El valor debe ser numérico, por favor intente nuevamente")
    ^
SyntaxError: EOL while scanning string literal

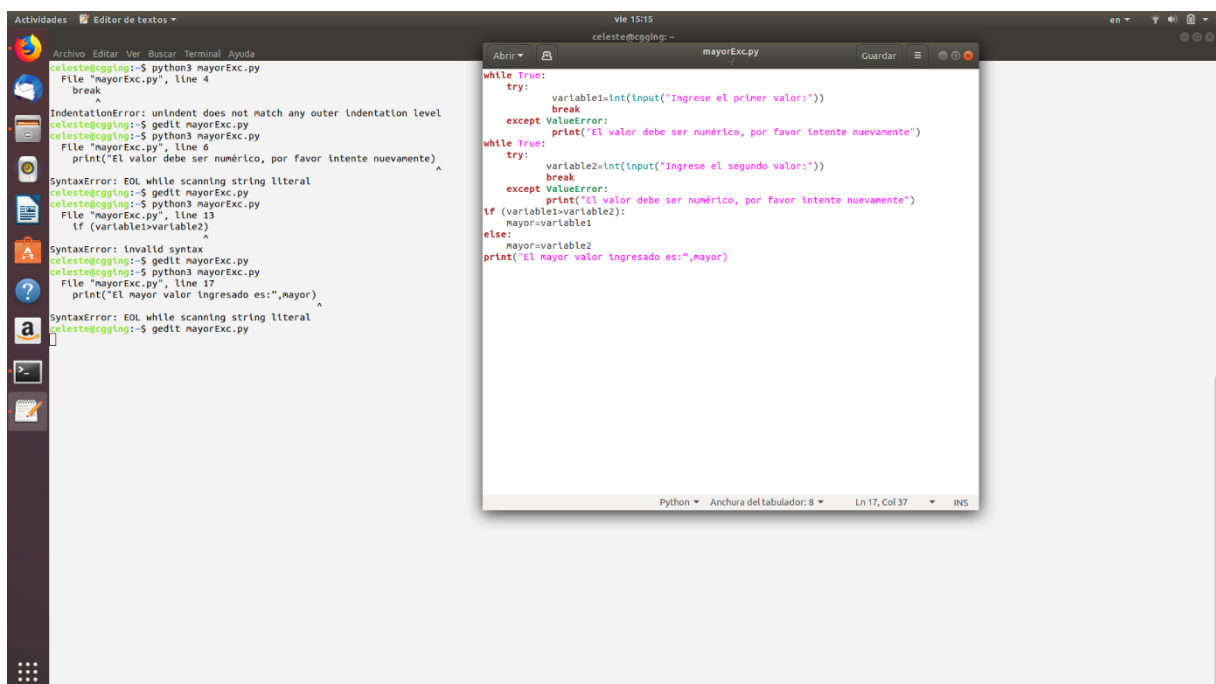
celeste@cggling:~$ gedit mayorExc.py
celeste@cggling:~$ python3 mayorExc.py
File "mayorExc.py", line 13
    tf (variable1>variable2)
    ^
SyntaxError: invalid syntax

celeste@cggling:~$
```

Y este error hace referencia a la ausencia de los ‘:’ en ‘if’, lo cual deberemos corregir para que nuestro código funcione.

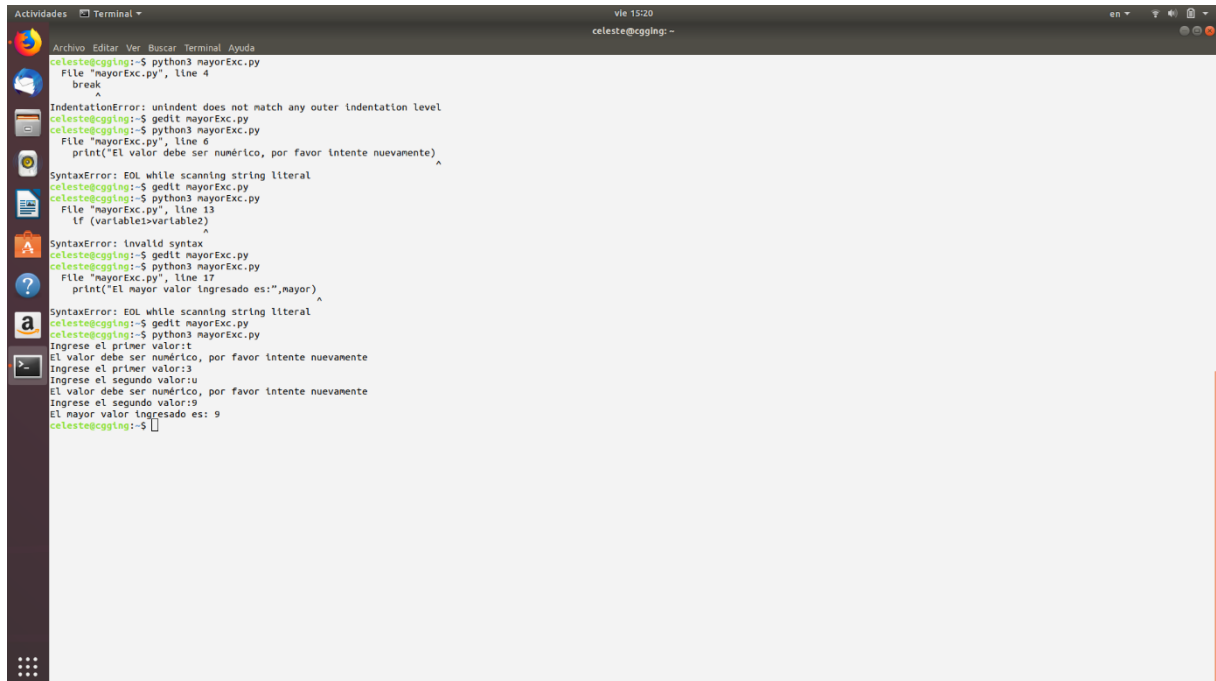
Una vez realizada esta corrección y al intentar nuevamente la ejecución del script, vemos otro error, idéntico al del print que nos apareció con anterioridad, no obstante, al mirar el código vemos que las comillas figuran...

Si bien es cierto que tiene comillas de cierre, aquí se produjo el error de copiado que mencionamos al principio del capítulo y podemos ver que las comillas de cierre son diferentes a las comillas de apertura. Para solucionar este error, simplemente cambiamos esas comillas.



```
while True:
    try:
        variable1=int(input("Ingrese el primer valor:"))
        break
    except ValueError:
        print("El valor debe ser numérico, por favor intente nuevamente")
while True:
    try:
        variable2=int(input("Ingrese el segundo valor:"))
        break
    except ValueError:
        print("El valor debe ser numérico, por favor intente nuevamente")
if (variable1>variable2):
    mayor=variable1
else:
    mayor=variable2
print("El mayor valor ingresado es:",mayor)
```

Finalmente, si volvemos a ejecutar el script con todas las correcciones anteriores:



```
celeste@cggling:~$ python3 mayorExc.py
File "mayorExc.py", line 4
    break
    ^
IndentationError: unindent does not match any outer indentation level
celeste@cggling:~$ gedit mayorExc.py
celeste@cggling:~$ python3 mayorExc.py
File "mayorExc.py", line 6
    print("El valor debe ser numérico, por favor intente nuevamente")
    ^
SyntaxError: EOL while scanning string literal
celeste@cggling:~$ gedit mayorExc.py
celeste@cggling:~$ python3 mayorExc.py
File "mayorExc.py", line 13
    tf (variable1variable2)
    ^
SyntaxError: Invalid syntax
celeste@cggling:~$ gedit mayorExc.py
celeste@cggling:~$ python3 mayorExc.py
File "mayorExc.py", line 17
    print("El mayor valor Ingresado es:",mayor)
    ^
SyntaxError: EOL while scanning string literal
celeste@cggling:~$ gedit mayorExc.py
celeste@cggling:~$ python3 mayorExc.py
Ingrese el primer valor:t
El valor debe ser numérico, por favor intente nuevamente
Ingrese el primer valor:3
Ingrese el segundo valor:u
El valor debe ser numérico, por favor intente nuevamente
Ingrese el segundo valor:9
El mayor valor Ingresado es: 9
celeste@cggling:~$
```

Buenas prácticas: Eligiendo nombres significativos.

Un detalle que nos puede resultar engorroso en nuestros comienzos en Python, es el tipado dinámico de las variables. Si bien es algo que resulta muy útil, y es sencillo desligarse de la responsabilidad de declarar el tipo de variable que queremos utilizar, también representa un riesgo, ya que en el transcurso del código podemos “olvidarnos” a qué tipo de dato se correspondía una cierta variable y podemos terminar intentando sumar 5 con ‘manzana’.

Para evitar este problema, es sumamente recomendable y considerado dentro de las buenas prácticas de la programación, elegir nombres significativos para nuestras variables. Así, por ejemplo, si tenemos que sumar 2 valores las variables podrían llamarse valor1 y valor2, en lugar de a y b. Si queremos solicitarle al usuario el nombre, la variable se llamara nombre y no x.

Lo mismo se tiene en cuenta al momento de declarar una función, el nombre de la misma debe elegirse acorde a lo que hace, así, por ejemplo, si tenemos una función que saca el promedio de dos valores, resultará natural llamarla promedio(val1,val2) y será fácil de comprender lo que hace dicha función y que trabaja con dos números como entrada... mucho más que si se llamara pepe(a,b).

## Ejemplo final: sumando la documentación del código

Es importante al programar tener en cuenta ciertas normas implícitas que se consideran buenas prácticas. Una de ellas, que hemos mencionado, pero no hemos puesto en práctica

en los ejemplos anteriores es la de documentar o comentar el código que estamos escribiendo. Veamos cómo aplicar esta práctica en el ejemplo integrador que realizamos en la clase anterior.

```
from random import *                                #Importamos el modulo random
def generaNumeroAleatorio(minimo, maximo):          #definimos una funcion propia
    return randint(minimo, maximo)                  #retornamos un numero aleatorio

numero_buscado=generaNumeroAleatorio(1,100)         #Llamamos a la funcion propia
encontrado=False
intentos=0

while not encontrado:                               #Bucle a repetir mientras negado sea falso
    while True:                                     #Comenzamos el manejo de excepcion
        try:
            numero_usuario=int(input("Introduce el numero buscado: "))
            break
        except ValueError:                          #En caso de error de tipo de dato
            print("El valor ingresado debe ser un numero, por favor reintente.")
    if numero_usuario>numero_buscado:               #compara el valor random y del usuario
        print("El numero que buscas es menor")      #Si el de usuario es mayor se ejecuta
                                                    # este bloque

        intentos=intentos+1
    elif numero_usuario<numero_buscado:             #Este otro si el del usuario es menor
        print("El numero que buscas es mayor")
        intentos=intentos+1
    else:                                           #Cuando el usuario acierta, se ejecuta este bloque.
        encontrado=True                           #Se cambia el valor del booleano para terminar el bucle.
print("Has acertado, el numero correcto es ", numero_usuario, "te ha llevado", intentos, "intentos")
```

Como podemos ver en el ejemplo anterior, se añadieron los comentarios del código que podían aportar a la explicación del funcionamiento del mismo, esto se considera una buena práctica ya que facilita la tarea de mantenimiento del código e incluso nos refresca la memoria en caso de que queramos reutilizar código para otro proyecto.

Un error común al momento de documentar el código es agregar información redundante, obvia o inútil, por ejemplo:

```
variable1=25          #declaramos la variable1
variable2=10          #declaramos la variable2
suma=variable1+variable2  #sumamos la variable1 y la variable2
print("La suma es", suma)  #mostramos por pantalla la suma de las variables1 y variable2
```

El código anterior es un claro ejemplo del mal uso de la documentación de un código, podemos ver que tenemos un montón de información inútil en los comentarios que no nos aportan nada que el propio código no nos haya aportado a simple vista.



## Lo que aprendimos en esta clase

En esta clase hemos recorrido los errores más comunes que podemos cometer al escribir nuestros códigos. También aprendimos a manejar algunas excepciones básicas integradas en el intérprete de Python.

Por último, repasamos la importancia de las buenas prácticas e identificamos mediante un ejemplo la importancia de realizar correctamente la documentación del código escrito.