UNIVERSIDAD NACIONAL
GUILLERMO BROWN

# *Introducción a la Programación*

## Algóritmos y Estructuras de Datos

-00184-

**Dr. Diego Agustín Ambrossio**

**Anl. Sis. Angel Leonardo Bianco**

# Contenido:

- Importancia de la depuración
- Etapas del desarrollo y aparición de errores
- Tipos de errores: sintácticos y semánticos
- Errores sintácticos comunes en Python
- Errores semánticos: qué son y cómo detectarlos
- Herramientas básicas de depuración (print(), assert)
- Ejemplos de errores frecuentes y cómo resolverlos
- Recomendaciones para evitar errores y mejorar el código

## Programar implica cometer errores:

- Programar es una tarea propensa a errores.
- Pueden ser inesperados o lógicos.
- Es fundamental abordar los errores de manera sistemática.

## Etapas en la solución de problemas:

1. Comprender el problema (input, proceso, salida).
2. Convertir el problema en un algoritmo.
3. Codificar el algoritmo en un lenguaje (por ejemplo, Python).
4. Comprobar y validar que el código funcione.
5. Presentar o visualizar los resultados.

## Excepciones:

- Las excepciones son eventos inesperados que ocurren durante la ejecución de un programa.
- Una excepción puede deberse a un error lógico o a un error imprevisto.
- En Python, las excepciones (también conocidas como errores) son objetos generados (o lanzados) por código que encuentra un evento inesperado. circunstancia.
- El intérprete de Python también puede generar una excepción.
- Un error generado puede ser detectado por un contexto circundante que "maneja" la excepción de manera adecuada. Si no se detecta, una excepción hace que el intérprete deje de ejecutar el programa y envíe un mensaje apropiado a la consola.

# Excepciones Comunes:

Python incluye una rica jerarquía de clases de excepción que designan varias categorías de errores

| Class | Description |
|---|---|
| Exception | A base class for most error types |
| AttributeError | Raised by syntax obj.foo, if obj has no member named foo |
| EOFError | Raised if "end of file" reached for console or file input |
| IOError | Raised upon failure of I/O operation (e.g., opening file) |
| IndexError | Raised if index to sequence is out of bounds |
| KeyError | Raised if nonexistent key requested for set or dictionary |
| KeyboardInterrupt | Raised if user types ctrl-C while program is executing |
| NameError | Raised if nonexistent identifier used |
| StopIteration | Raised by next(iterator) if no element; see Section 1.8 |
| TypeError | Raised when wrong type of parameter is sent to a function |
| ValueError | Raised when parameter has invalid value (e.g., sqrt(−5)) |
| ZeroDivisionError | Raised when any division operator used with 0 as divisor |

## Clasificación de errores

- **Errores Gramaticales** (tambien los llamamos **Errores Sintácticos**): no nos permiten continuar la ejecución de nuestro programa.
- **Errores semánticos**: permiten la ejecución, pero los resultados son erróneos.

## Ejemplo error de sintáxis.

```
L=[]
L[0] # el indice es menor a la longitud ' i  > len(L)-1 '
```

## Ejemplo error semántico.

```
L=[1,3,2,4]
# Lsorted = L.sort()# comando incorrecto (ya que estamos cambiando la
lista L)
Lsorted = L.copy().sort() # comando correcto (ya que la lista L no ha
cambiado)
# Probar ambos comandos
print(Lsorted)
```

# Errores Sintácticos:

***AssertionError***

Esta excepción es llamada dentro del código del programa para informar un error.

```
# Ejemplo
assert not True, 'Error Aquí!!'
```

***RunTimeError***

Esta excepción aparece cuando hay un error que no tenga un tipo de error predefinidos. Es un error en ejecución ***'generico'***

```
# Ejemplo
raise
```

# Errores Sintácticos:

***IndentationError***

Esta excepción aparece cuando nuestro código no respeta la indentación de Python ("*unexpected indent*"). También aparece, cuando no hay un bloque de código luego de un “:” ("*expected an indented block*").

```
# Ejemplo
3+2 # bien!
 3+2 # mal indentado.
```

```
# Ejemplo
for x in range(0,9):  # No hay ningún
                      # bloque de código
print(x) # la función print no respeta la indentación adecuada para
estar dentro del for
```

# ¿Qué podemos hacer?

- Chequear que todas las lineas de código esten alineadas correctamente. Si utilizamos algun tipo de IDE, o editor de texto avanzado, podremos ver claramente los niveles de indentación.

- En el caso de que la indentación de nuestro código sea correcta, e igualmente persiste el error ***IndentationError*** puede ser debido a 'mezclar' distintos tipos de *espacios y tabulaciones*. Debemos unificar estos *caracteres invisibles* de acuerdo con la indentación predefinida de Python.

- Hay un bloque de código (apropiadamente indentado) luego de los dos puntos? En el caso que querramos forzar un deberemos escribir ***pass*** luego de los dos pu

```
# Ejemplo
for x in range(0,9): pass
print(3)
```

- Ser cuidadoso cuando copiamos/pegamos líneas que tienen una indentación distinta.

# Errores Sintácticos:

***IndexError***

Esta excepción aparece cuando el indice de un ***string*** o ***lista*** esta fuera del rango.

```
# Ejemplo
L=[0,1,2]
L[3]
```

***¿Qué podemos hacer?***

- Recordar que los indices de una lista L (string s) permitidos, pertenecen al rango [ 0, …., len (L)-1 ] ( [0, …., len (s)-1 ] ).

- Las listas son objetos mutables, debemos ser concientes que algunos comandos pueden extender o acortar la lista.

- Si utilizamos distintas variables para indexar, debemos comprobar que no hayan sido cambiadas previamente en nuestro código, o en alguna función externa.

# Errores Sintácticos:

***NameError***

Esta excepción aparece cuando una variable no es definida.

```
# Ejemplo
# notassigned =1
notassigned
```

***¿Qué podemos hacer?***

- Localizar la variable utilizando el número de linea provisto por el error/excepción.
- Localizar la linea de código (si es que existe) en donde dicha variable fue definida, es esta linea antes o despues del error? Es esta linea bien indentada?
- Revisar si hay algun error ortográfico o sintáctico al llamar la variable.
- Si el error esta dentro de una función, es posible que dicha variable es ***local*** y su uso es ***global*** (or reversamente).
- Si el nombre pertenece a una función, primero asegurarse que estemos importando el módulo que contiene la función.

# Errores Sintácticos:

***OverFlowError***

Esta excepción aparece cuando sobrepasamos el limite computacional de los números de punto flotante (***pueden usar solo una cantidad de memoria limitada***). En contraste los números enteros pueden ser de cualquier extensión, más precisamente hasta que nos quedamos sin memoria reservada y se lanza una excepción ***MemoryError***.

```
# Ejemplo
(1.0*10**1000)/10**1000 #
OverflowError!
```

***¿Qué podemos hacer?***

- Convertir números de punto flotante a enteros cuando sea posible.
- Evitar el uso de números de punto flotante muy grandes, se deben usarse distintas técnicas para obtener dicha precisión.

# Errores Sintácticos:

***SintaxError***

Este error aparece cuando hay un problema sintáctico en el código. Puede suceder cuando usamos caracteres especiales, intentamos asignar un valor a un nombre reservado o protegido, o a un número.

Además de aparecer cuando nos olvidamos los dos puntos luego de un for, if, while, def, etc.

En la documentación oficial de Python se hace referencia a estos como ***Errores***, cualquier otro tipo de error una ***Excepción***.

```
# Ejemplos
print(x

for x in range(0,9)
    print(x)
```

***¿Qué podemos hacer?***

- Utilizar letras sin acento u otros caracteres especiales.
- Revisar el uso del = y del ==.
- Recordar los : al final del if, while, etc

# Errores Sintácticos:

***TypeError***

Esta excepción aparece cuando utilizamos un tipo erroneo, es decir distinto al esperado. Ocurre usualmente cuando llamamos a una función la cual requiere de tipos de parámetros especifícos. Un número erroreo de argumentos. Y por otras diversas razones relacionadas a conflictos de tipos.

```
# Ejemplos
"1"[0]
#len(3)
```

***¿Qué podemos hacer?***

- Buscar nombre de variables duplicados. Intentar escribir nombres mas largos en caso de ser necesario.
- Siempre podemos utilizar la función help para ver información sobre la cantidad y tipos de argumento que una función requiere.
- Para el caso de los métodos, podemos obtener esta información llamando a help sobre el tipo del objeto.

# Errores Sintácticos:

***ValueError***

Esta excepción aparece cuando llamamos a una función (o método) con argumentos del tipo correcto, pero con valores que no son validos para dicha función (método).

```
# Ejemplo
int('value')
#1/0
```

***¿Qué podemos hacer?***

- Llamar a la función help y verificar que los valores son validos.
- Cuando intentamos dividir por cero, optenemos directamente una excepción ***ZeroDivisionError***.
- En general, estos problemas son especifícos caso por caso, en caso de querer circunventarlos podemos utilizar comandos if , o bloques de código try  Por ejemplo, estas dos formas nosotros mismos realizamos el manejo del error a priori, o a posteriori respectivamente.

## Errores Semánticos:

Los errores semánticos son mucho mas frecuentes que los sintácticos. Por definición, son errores del tipo ***el programa hace algo, pero no lo que quiero***. En este caso, el programa esta libre de problemas sintácticos graves, o es posible ejecutarlo "sin errores".

Una solución general para detectar problemas semánticos es monitorear los valores de las variables durante la ejecución del programa, ya sea utilizando numerosos lineas ***print***, o utilizando alguna herramienta de depuración (debugger) que nos facilite esta tarea.

# Errores Semánticos:

***Sintoma : A veces obtengo un error sintatíco.***

```
# Ejemplo
L=[1,32,3]
M = L # M y L son/apuntan a la misma lista.
print("L=",L,"M=",M)
L.pop(1)
print("L=",L,"M=",M)
M[2]
# Probar reemplazando por M = L.copy(). Funcionan de la misma forma?
```

```
# Ejemplo
L=[1,3,2,4]
Lsorted=L.sort() # Además del efecto secundario que el metodo sort
tiene, es ordenar la lista L 'sobre si misma'.
```

# Errores Semánticos:

***Sintoma : A veces obtengo un error sintatíco.***

```
# Explicar por que obtenemos un error. Correjir el código.
L=[i for i in range(0,10)] # Queremos obtener una lista de enteros impares.
for x in range(0,5):
    print(L)
    L.pop(2*x)
```

'2*x' hace referencia a los indices pares en la lista original (los números impares) pero estamos cambiando la lista en cada paso del ciclo.

# Errores Semánticos:

## *Sintoma : Mi programa no termina su ejecución.*

Una vez comenzada la ejecución de nuestro programa, a veces, parece que no fuera a terminar de ejecutarse nunca. Debemos tratar de asegurarnos que podemos identificar cuales con las líneas de código responsables. Existen dos casos posibles, el código no terminara nunca de ejecutarse, o eventualmente terminara su ejecución, pero no sabremos cuando.

***Cómo difereciamos ambos casos?***
Ejecutar el programa con la menor cantidad de datos posibles y comprobar que el código funciona correctamente.

- ***el programa no termina***

- ***el programa es "muy pesado"***

# Errores Semánticos:

## *Síntoma : Mi programa no retorna valores correctos.*

En este momento nuestro programa se ejecuta sin errores, pero retorna resultados erróneos. La pregunta es: ***Cuán mal están los resultados?*** Si el resultado es de un tipo erróneo, podemos suponer algún error de casting o ***TypeError*** dentro de nuestro código. Aquí tenemos una breve lista de posibles problemas a chequear:

- Separar el código en funciones o módulos, para tratar de encapsular el error, y poder testear distintas partes del código de manera individual.
- Chequear los objetos que sean mutables.

```
# Ejemplo
mess=list(set(mess)) # ordenara eficientemente la lista
'mess', pero multiple valores pueden perderse.
```

# Levantando una Excepción:

Se genera una excepción al ejecutar la declaración de aumento, con una instancia adecuada de una clase de excepción como argumento que designa el problema.

Invocamos el comando ***raise*** de la siguiente forma:

***raise NameError (string)***

La ejecución del código se **detendrá** de inmediato en la línea donde se ha llamado al comando **raise.**

Este comando nos sirve para "lanzar" erorres, y avisar al sistema/usuario del evento.

```
n=int(input("Numerador : "))
d=int(input("Denominador : "))
if d==0:
    raise ValueError("El denominador debe ser distinto de '0'")
print(n*1.0/d)
```

## Levantando Una Excepción:

También podemos hacer uso del comando **assert**, que es muy similar al comando **raise**. De hecho, podemos escribirlos casi de manera equivalente:

```
if someboolean:
        raise AssertionError("string")
```

es equivalente a:

```
if __debug__:
        assert someboolean, "string"
```

# Manejando Una Excepción:

Los errores en nuestro código terminan la ejecución del programa. En algunos casos, nos gustaría ignorar algunos errores no críticos y poder continuar ejecutando el programa. Utilizando el comando **try** podemos capturar excepciones (errores) y decidir cómo proseguir.

**Estructura general del comando try:**

```
try:
    # Bloque A: código que puede generar errores
    # Bloque A: más código potencialmente riesgoso

except Error1 as e1:
    # Bloque B1: manejo específico del Error1
except Error2 as e2:
    # Bloque B2: manejo específico del Error2
else:
    # Bloque C: se ejecuta solo si no hubo ningún error
```

# Manejando Una Excepción:

**Ejemplo:**

```
L=[1,2]
L[2]=6
```

```
try:
    L=[1,2]
    L[2]=6
except IndexError:
    print(' Hay un problema!')
else: pass
```

# METODO AXIOMATICO