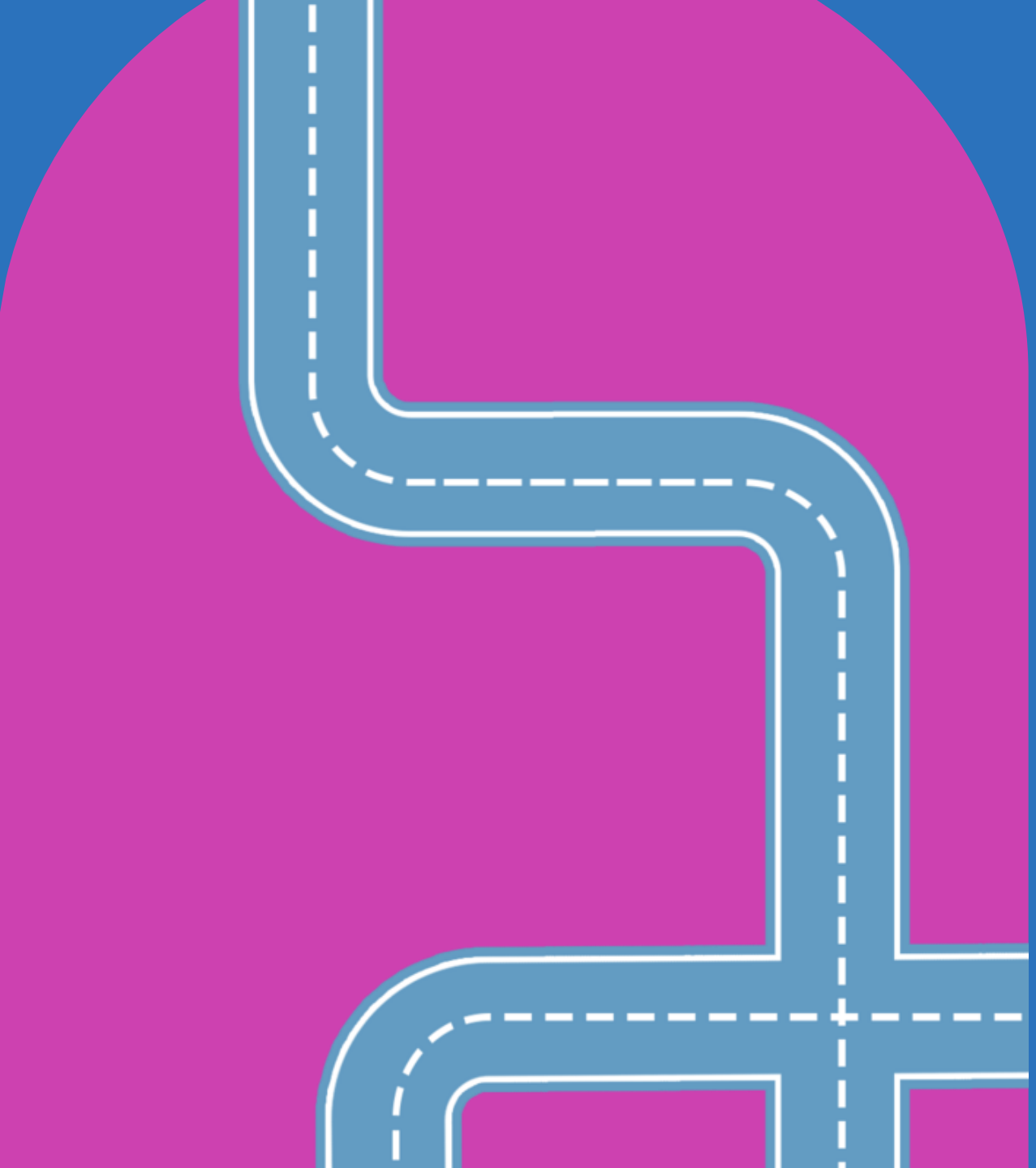
EXCEPCIONES

Algoritmos y estructuras de datos I



EL CAMINO FELIZ

Es un tipo de prueba de software que utiliza datos de entrada conocidos y produce una salida esperada.

El camino feliz no duplica las condiciones del mundo real. Solo verifica que la funcionalidad requerida esté en su lugar y funcione correctamente en base a las casuísticas más triviales. Por ende, se identifica como el escenario predeterminado o la alternativa positiva más probable sin condiciones excepcionales o de error.

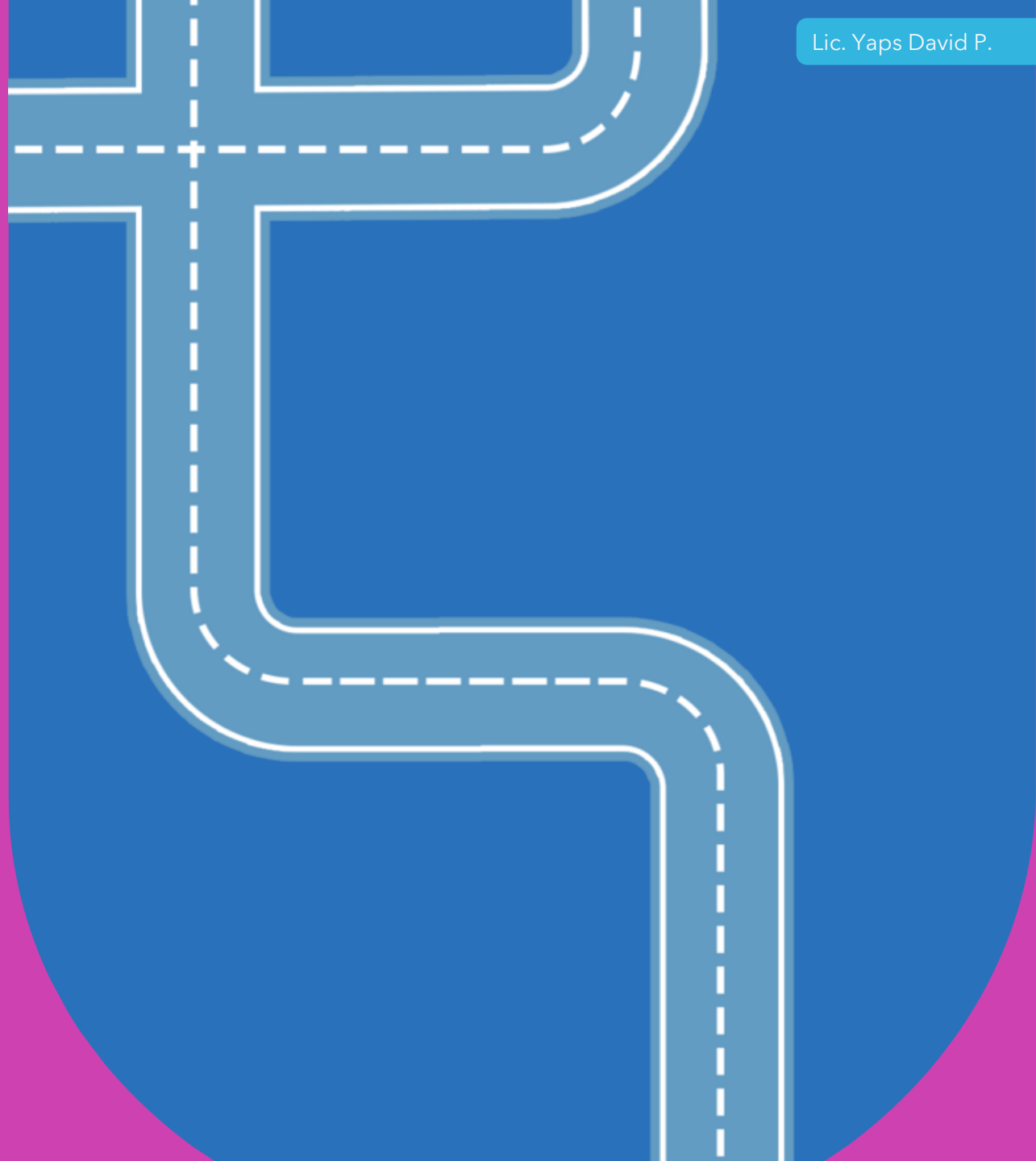
Los escenarios del camino feliz excluyen las excepciones y los errores humanos, tales como la posibilidad de cargar valores fuera de rango o de diferente tipo.

EL CAMINO FELIZ

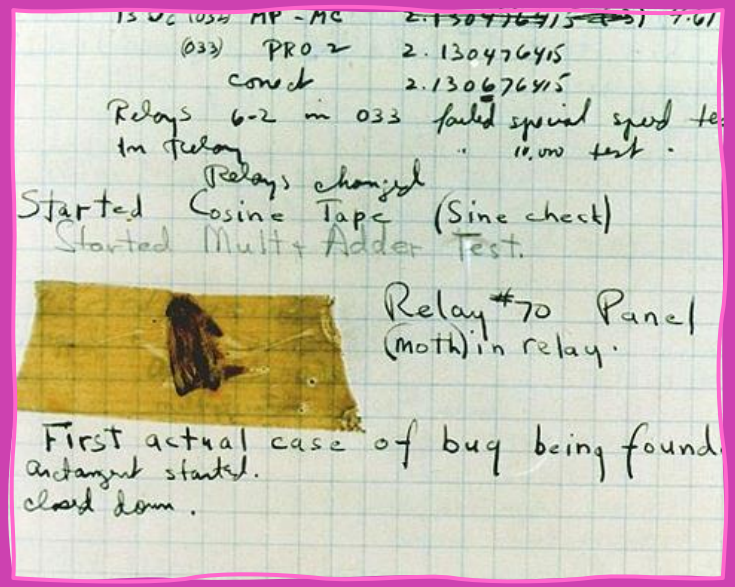
Se deben escribir las pruebas no sólo para la situación ideal (el camino feliz) sino que nuestro código debe ser lo suficientemente robusto como para comprobar todas aquellas situaciones incómodas en las que una funcionalidad particular se podría ver comprometida.

La intención, es la de detectar errores en etapas tempranas de desarrollo. Si nos quedamos con los casos más sencillos, seguramente estaremos ocultando errores que, tarde o temprano, deberemos corregir.

¿Cuándo es el momento más oportuno para detectar errores y problemas?, ¿cuándo estamos desarrollando una nueva funcionalidad o cuando el software está ya en producción con uno o varios clientes usándolo?.



Lic. Yaps David P.



Esta polilla es posiblemente el primer bug detectado en una computadora.
Encontrada por Grace Hopper en la Universidad de Harvard, el 9 de Septiembre de 1947.

BUG: ETIMOLOGÍA

Cuando se construyeron las primeras computadoras, que ocupaban cuartos enteros, la circuitería consistía miles de elementos tales como relés, resistencias, capacitores y tubos de vacío. Los relés son llaves que abren y cierran circuitos eléctricos cuya activación se hace por medio de una corriente que activa un electroimán.

Las resistencias y tubos de vacío producían calor, lo cual atrae a los insectos. Si los insectos morían entre los dos contactos de un relé, el circuito podía cerrarse (o no llegar a hacerlo) lo cual producía resultados inesperados e incorrectos.

DEPURACIÓN DE PROGRAMAS

La depuración de programas es el proceso de identificar y corregir errores de programación (debugging). Si alguien busca la definición de bug en un diccionario de inglés descubrirá que significa insecto o bicho, pero para la informática un bug es cualquier error, mal funcionamiento o falla que produce que el software no funcione como se esperaba.

La mayoría de los bugs provienen de los errores cometidos al programar, aunque algunos otros pueden provenir de fallas en el diseño y, los menos, de la conversión que los compiladores hacen del código fuente a código de máquina o la configuración del equipo que se utiliza el ejecutable.

```python
dividendo = int(input("Ingrese el dividendo: "))
divisor = int(input("Ingrese el divisor: "))

resultado = dividendo / divisor

print(f"El resultado es {resultado}")
```

```
Ingrese el dividendo: 33
Ingrese el divisor: 0
Traceback (most recent call last):
  File "errores.py", line 4, in
<module>
    resultado = dividendo / divisor
ZeroDivisionError: division by zero
```

DIVISIÓN POR CERO

La división por cero no está definida como operación matemática. Es por esto que si intentamos dividir una variable por otra cuyo valor es cero, el programa falla y su ejecución es finalizada automáticamente o, peor aún, continúa con un resultado incierto.

El código anterior funcionará, en tanto y en cuanto, denominador sea un número distinto de 0; sino obtendremos un error.

OPERACIONES CON DIFERENTES TIPOS DE DATOS

Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.

En este caso, no se contempló la conversión del tipo ingresado por el usuario para realizar una operación matemática.

```
dividendo = input("Ingrese el dividendo: ")
divisor = input("Ingrese el divisor: ")

resultado = dividendo / divisor

print(f"El resultado de la operación
{dividendo}/{divisor} es {resultado}")
```

```
Ingrese el dividendo: 11
Ingrese el divisor: 5
Traceback (most recent call last):
  File "errores.py", line 4, in
<module>
    resultado = dividendo / divisor
TypeError: unsupported operand type(s)
for /: 'str' and 'str'
```

DESBORDAMIENTO DE VARIABLES NUMÉRICAS

```
valor = 2.0

for i in range(100):
    print(i, valor)
    valor = valor ** 2
```

```
0 2.0
…
8 1.1579208923731620e+77
9 1.3407807929942597e+154
Traceback (most recent call last):
  File "errores.py", line 5, in <module>
    valor = valor ** 2
OverflowError: (34, 'Result too large')
```

Dependiendo del tipo de variable, por ejemplo, las numéricas, se tiene un rango posible de valores. Entonces, si se intenta almacenar un valor mayor, se desborda su capacidad y el valor resultante no es posible conseguirlo.

En el ejemplo propuesto, luego de la novena iteración, ya no se puede representar el valor con el tipo flotante.

Lic. Yaps David P.

KILL SCREEN EN EL JUEGO PAC-MAN

En Pac-Man, el bug del nivel 256, también conocido como *Kill Screen* se debe a un desbordamiento del contador de niveles en la memoria del juego. Este contador utiliza un byte de 8 bits, capaz de representar valores de 0 a 255.

Cuando el juego intenta avanzar del nivel 255 al 256, el contador se desborda y se reinicia a 0 debido a las limitaciones de la aritmética modular en binario. El desbordamiento provoca que el juego acceda a regiones incorrectas de memoria para generar el nivel 256, resultando en gráficos corruptos y comportamiento errático de los elementos del juego.

Como consecuencia, el laberinto del nivel se vuelve intransitable, deteniendo efectivamente el progreso del jugador y creando una "pantalla de muerte" que marca el fin del juego.

Lic. Yaps David P.

PÉRDIDA DE PRECISIÓN EN LA CONVERSIÓN DE TIPOS DE DATOS NUMÉRICOS

Este error sucede generalmente cuando se efectúa un redondeo (acercar al entero más próximo) o un truncamiento (uso sólo de la parte entera).

Por ejemplo, la diferencia entre 4,845 y 4,8 debería ser 0,055. Pero si lo calcula en Python, veremos que 4.9 - 4.845 no es igual a 0.055.

Esto se debe a que el punto flotante no se puede representar por el número exacto, es solo una aproximación, y cuando se usa en aritmética, éste causa un pequeño desvío.

La comparación *duracion != 0* nunca es falsa debido a errores de redondeo acumulados en las operaciones de punto flotante.

Original	Redondeo	Truncado
5.24	5	5
2.99	3	2
4.5	5	4
7	7	7

```
print(4.9 - 4.845 == 0.055) #False
print(4.9 - 4.845) #0.05500000000000604

print("Truncado:", int(2.99)) #2
print("Redondeo simple:", round(2.99)) #3


duracion = 13.0

while duracion != 0:
    print(duracion)
    duracion -= 0.1
```

```python
def sumarYRestar(iteraciones):
    resultado = 1

    for i in range(iteraciones):
        resultado += 1/3

    for i in range(iteraciones):
        resultado -= 1/3

    return resultado

print(1 - 1/3 + 1/3) #1.0
print(sumarYRestar(1000)) #1.0000000000000064
```

ACUMULACIÓN DE ERROR DE REDONDEO

Cuando estamos haciendo una secuencia de cálculos sobre una entrada inicial con error de redondeo, debido a una representación inexacta, los errores pueden magnificarse o acumularse.

En el siguiente ejemplo, tenemos el número 1 sumado y restado 1/3, lo que nos da el mismo número 1. Pero, si sumamos 1/3 muchas veces y restamos el mismo número de veces 1/3, no seguimos teniendo el mismo número 1. Cuantas más veces hagamos esto, más errores acumulará.

ERROR DE INDICE

Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.

En el ejemplo, se recorren una cantidad de elementos más grande (7) que la del total de elementos de la lista (3).

Cuando se supera ese tope, ocurre un error de índice.

```
numeros = [11, 22, 33]

for i in range(0, 7):
    print(numeros[i])
```

```
11
22
33
Traceback (most recent call last):
  File "errores.py", line 4, in
<module>
    print(numeros[i])
IndexError: list index out of range
```

ERRORES DE SINTAXIS

Estos errores son seguramente los más simples de resolver, pues son detectados por el intérprete (o por el compilador, según el tipo de lenguaje que estemos utilizando) al procesar el código fuente y generalmente son consecuencia de equivocaciones al escribir el programa.

En el caso de Python estos errores son indicados con un mensaje *SyntaxError*. Por ejemplo, si trabajando con Python intentamos definir una función y en lugar de *def* escribimos *void*.

```
void saludo():
   print("Hola mundo")
```

```
 File "saludo.py", line 1
    void saludo():
         ^^^^^^
SyntaxError: invalid syntax
```

ERRORES DE SEMÁNTICA

```
def areaTriangulo(base, altura):
    area = (base / altura) * 2
    return area

base = 5
altura = 4
resultado = areaTriangulo(base, altura)
print("El área del triángulo es:", resultado)
```

```
El área del triángulo es: 2.5
```

Los errores semánticos son aquellos que no causan errores durante la fase de compilación o interpretación, pero provocan que el programa no funcione como se espera. Estos errores pueden deberse a problemas en la lógica del programa, algoritmos incorrectos, omisión de sentencias necesarias, o interpretación incorrecta de los requisitos del problema.

Este código no generará ningún error, pero produce un resultado incorrecto. El error semántico radica en la fórmula utilizada para calcular el área de un triángulo. El cálculo del área de un triángulo se realiza multiplicando la base por la altura y luego dividiendo el resultado por 2. Sin embargo, en el código se utiliza la fórmula incorrecta donde se divide la base por la altura y luego se multiplica por 2.

ERRORES DE EJECUCIÓN

Estos errores aparecen durante la ejecución del programa y su origen puede ser diverso. En ocasiones pueden producirse por un uso incorrecto del programa por parte del usuario, por ejemplo, si el usuario ingresa una cadena cuando se espera un número o realizar una división por cero.

Una causa común de errores de ejecución que generalmente excede al programador y al usuario, son los recursos externos al programa, por ejemplo, si el programa intenta leer un archivo y el mismo se encuentra dañado o está en uso.

```
def obtenerElemento(lista, indice):
    elemento = lista[indice]
    print(f"El elemento en el índice {indice}
es: {elemento}")


miLista = [1, 2, 3, 4, 5]

#El elemento en el índice 2 es: 3
obtenerElemento(miLista, 2)

#Intento de acceso a un índice fuera del rango de la lista
obtenerElemento(miLista, 10)
```

```python
def dividir(a, b):
    #Verificar si el divisor es diferente de cero antes de realizar la división
    if b != 0:
        resultado = a / b
        print("El resultado de la división es:", resultado)
    else:
        print("Error: No se puede dividir por cero")

#Intentamos dividir 10 entre 2
dividir(10, 2)

#Intentamos dividir 10 entre 0
dividir(10, 0)
```

MIRA ANTES DE SALTAR
(LOOK BEFORE YOU LEAP)

Implica realizar pruebas explícitas antes de tomar una acción, con el objetivo de prevenir errores y problemas potenciales. Este enfoque se centra en la verificación anticipada de condiciones para asegurarse de que se cumplan antes de ejecutar una acción. Al adoptar este enfoque, se pueden evitar errores costosos en el futuro y garantizar que el software satisfaga los requisitos y expectativas de los usuarios finales.

En este ejemplo, antes de realizar la división, se verifica explícitamente si el divisor es diferente de cero. Esta verificación previa ayuda a prevenir un posible error de división por cero, lo que podría causar que el programa falle. Al adoptar el enfoque "Mira antes de saltar", aseguramos que el programa funcione de manera segura y que se eviten errores costosos.

ES MÁS SENCILLO PEDIR PERDÓN QUE PEDIR PERMISO
(EASIER TO ASK FORGIVENESS THAN PERMISSION)

El enfoque sugiere que, en general, es preferible intentar ejecutar las sentencias directamente y manejar los casos excepcionales mediante la captura de errores.

En el ejemplo, intentamos realizar la división directamente y capturamos cualquier excepción que pueda surgir, como la división por cero. En lugar de verificar previamente si el divisor es cero antes de realizar la división, simplemente intentamos la operación y manejamos cualquier error que ocurra.

```
def dividir(a, b):
    try:
        resultado = a / b
        print("El resultado de la división es:", resultado)
    except ZeroDivisionError:
        print("Error: No se puede dividir por cero")


#Intentamos dividir 10 entre 2
dividir(10, 2)

#Intentamos dividir 10 entre 0
dividir(10, 0)
```

EXCEPCIONES

Los errores de ejecución son llamados comúnmente excepciones. Durante la ejecución de un programa, si dentro de una función surge una excepción y la función no la maneja, la excepción se propaga hacia la función que la invocó, si esta otra tampoco la maneja, la excepción continúa propagándose hasta llegar a la función inicial del programa y si esta tampoco la maneja se interrumpe la ejecución del programa.

Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa.

Python utiliza un objeto especial llamado excepción para controlar cualquier error que pueda ocurrir durante la ejecución de un programa.

Cuando ocurre un error durante la ejecución de un programa, Python crea una excepción. Si no se controla esta excepción la ejecución del programa se detiene y se muestra el error (*traceback*):

```
print(1 / 0)  # Error al intentar dividir por 0.
Traceback (most recent call last):
ZeroDivisionError: division by zero
```

TIPOS DE EXCEPCIONES

Las principales excepciones definidas en Python son aquellas que cubren una amplia gama de situaciones comunes en la programación. Estas excepciones están diseñadas para ayudar a los desarrolladores a identificar y manejar errores de manera efectiva durante la ejecución de sus programas.

Es importante destacar que Python proporciona una amplia variedad de excepciones para abordar distintos tipos de errores que pueden surgir durante la ejecución del programa. Para obtener una lista completa de todas las excepciones posibles y comprender mejor cuándo y cómo se deben manejar, se recomienda consultar la documentación oficial del lenguaje.

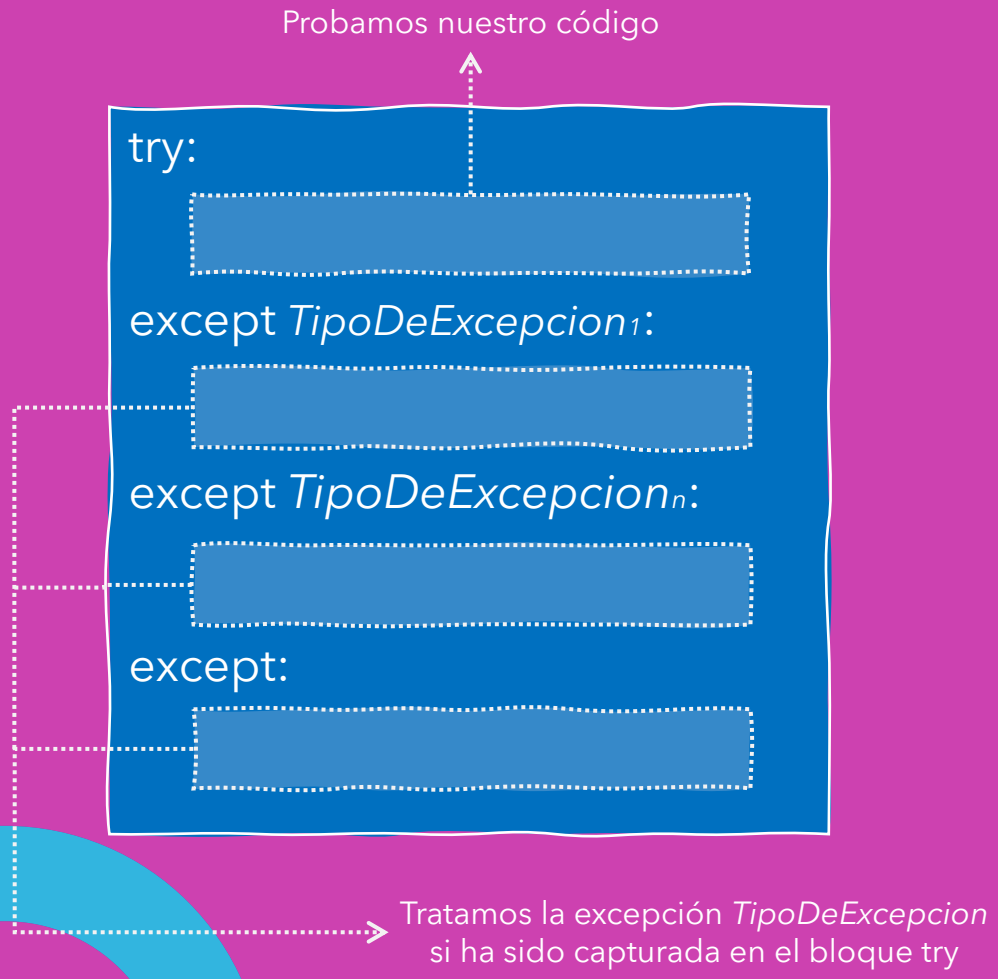
TypeError
Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.

ZeroDivisionError
Sucede cuando se intenta dividir por cero.

OverflowError
Acontece cuando un cálculo excede el límite para un tipo de dato numérico.

IndexError
Se produce cuando se intenta acceder a una secuencia con un índice que no existe.

ESTRUCTURA TRY/EXCEPT



El bloque *try* se ejecuta y, si no se produce ninguna excepción, se salta el bloque o bloques *except*.

Si al ejecutar alguna de las sentencias del bloque try se produce una excepción, el resto de las sentencias del bloque *try* se ignoran.

Si el bloque *try* ha lanzado una excepción y su tipo coincide con alguna de las contempladas en un bloque *except*, tratamos la excepción ejecutando únicamente las sentencias de ese bloque.

Si el tipo de la excepción no coincide con ninguna de las contempladas se reenvía a otro posible bloque *try* más externo que contenga a éste o, de no existir, se detiene la ejecución con el mensaje correspondiente.

EXCEPCIONES

En este caso, se generó la excepción *ZeroDivisionError* cuando se quiso hacer la división. Para evitar que se levante la excepción y se detenga la ejecución del programa, se utiliza el bloque *try-except*.

Dado que dentro de un mismo bloque *try* pueden producirse excepciones de distinto tipo, es posible utilizar varios bloques *except*, cada uno para capturar un tipo distinto de excepción.

Esto se hace especificando a continuación de la sentencia *except* el nombre de la excepción que se pretende capturar. Un mismo bloque *except* puede atrapar varios tipos de excepciones, lo cual se hace especificando los nombres de las excepciones separadas por comas a continuación de la palabra *except*.

```
dividendo = 5
divisor = 0

try:
    total = dividendo / divisor
except:
    print("No se puede dividir por cero")
```

ESTRUCTURA TRY/EXCEPT

```
try:
    #Aquí ponemos el código que puede lanzar excepciones
except IOError:
    #Entrará aquí en caso de que se haya producido una excepción IOError
except ZeroDivisionError:
    #Entrará aquí en caso de que se haya producido una excepción ZeroDivisionError
except:
    #Entrará aquí en caso de que se haya producido una excepción que no corresponda a ninguno de los tipos especificados en los except previos
```

Es importante destacar que, si bien luego de un bloque try puede haber varios bloques *except*, se ejecutará, a lo sumo, uno de ellos.

Como se muestra en el ejemplo precedente también es posible utilizar una sentencia *except* sin especificar el tipo de excepción a capturar, en cuyo caso se captura cualquier excepción, sin importar su tipo. Cabe destacar, también, que en caso de utilizar una sentencia *except* sin especificar el tipo, la misma debe ser siempre la última de las sentencias *except*.

TRY/EXCEPT/FINALLY

El bloque *finally* es una estructura opcional que se utiliza para escribir sentencias de finalización, generalmente destinadas a acciones de limpieza.

La característica distintiva del bloque es que siempre se ejecuta, independientemente de si se produjo una excepción o no durante la ejecución del bloque *try*.

Cuando se encuentra un bloque *try*, se ejecuta las instrucciones contenidas en él. Si surge una excepción durante esta ejecución, Python interrumpe su ejecución y busca el bloque *except* correspondiente para manejar la excepción.

Después de ejecutar el bloque *except*, si existe, se continúa ejecutando el bloque *finally*, si está definido, para realizar acciones de limpieza adicionales.

```python
import random

numeros = [0, 1, 1, 2, 3]
indice = random.randint(0, 10)

try:
    valor = numeros[indice]
except IndexError:
    valor = None
    print(f"Error: {indice} está fuera del rango.")
finally:
    #Siempre vaciamos la lista
    numeros.clear()
    print("Lista vacía.")

print("Índice generado:", indice)
print("Valor obtenido:", valor)
print("Estado de la lista:", numeros)
```

```
try:
    numerador = float(input('Numerador: '))
    denominador = float(input('Denominador: '))
    cociente = numerador/denominador
except ZeroDivisionError:
    print('Error: el denominador es 0')
except ValueError:
    print('Error: el valor no es válido')
else:
    print(numerador, "/", denominador, "=", cociente)
finally:
    print('Fin del programa.')
```

ELSE

El bloque *else* se utiliza para definir un conjunto de instrucciones que se ejecutarán si no se produce una excepción en el bloque *try*.

El propósito principal del bloque *else* en excepciones es permitir que el código maneje situaciones normales o exitosas después de que se haya intentado realizar una operación que podría generar una excepción.

AS

Usar *as* dentro de un bloque *except* permite capturar detalles específicos sobre el error y responder de manera más informativa. Esto es útil cuando no sabemos exactamente qué tipo de error podría ocurrir y queremos obtener más información al respecto.

La variable que utilizamos para capturar la excepción en el bloque *except* puede tener cualquier nombre que elijamos. La convención es usar *e, error,* o *exc* por ser nombres cortos y descriptivos, pero no hay restricciones.

En este caso, el mensaje de error indica que no se puede convertir la cadena en un número entero, y la excepción capturada se muestra en la consola para proporcionar información adicional sobre el error.

```python
cadena = "Hola mundo"

try:
    numeroEntero = int(cadena)
except Exception as e:
    #Mensaje de error
    print(f"Error: {e}") #Error: invalid literal for int() with base 10: 'Hola mundo'
    #Tipo de excepción
    print(f"Tipo de excepción: {type(e).__name__}") #ValueError
```

AS

El código utiliza *Exception* para capturar cualquier error y examina el mensaje del error para distinguir entre entradas no válidas y números no presentes en la lista.

Esto permite manejar cada error de manera específica sin crear múltiples bloques *except*. El análisis del mensaje facilita una mejor gestión de los errores con flexibilidad y claridad.

```
lista = [0, 1, 1, 2, 3]

numero = 0

while numero != -1:
    try:
        numero = int(input("Ingrese un valor: "))
        print(lista.index(numero))
    except Exception as e:
        mensaje = str(e)
        if "is not in list" in mensaje:
            print("El valor no está en la lista")
        elif "invalid literal for int()" in mensaje:
            print("Debe ingresar un número entero válido")
        else:
            print("Ocurrió un error inesperado:", mensaje)
```

```python
def calcularRaizCuadrada(valor):
    if valor < 0:
        raise ValueError("Error: El valor no puede ser negativo.")
    return valor ** 0.5


entradaValida = False

while not entradaValida:
    try:
        valor = float(input("Introduce un número: "))
        resultado = calcularRaizCuadrada(valor)
        print(f"La raíz cuadrada de {valor} es: {resultado}")
        entradaValida = True
    except Exception as e:
        print(f"Se produjo un error: {e}")
```

RAISE

La sentencia *raise* permite al programador forzar a que ocurra una excepción específica. El único argumento a *raise* indica la excepción a generarse.

En este ejemplo, *raise* se utiliza para manejar condiciones de error relacionadas con la entrada del usuario. Si el usuario ingresa un número negativo, se lanza una excepción que es capturada y manejada adecuadamente, permitiendo una experiencia de usuario más robusta.

```python
class SaldoInsuficienteError(Exception):
    def __init__(self, mensaje):
        self.mensaje = mensaje
        super().__init__(self.mensaje)

def solicitarPrestamo(saldoDisponible, montoSolicitado):
    if montoSolicitado > saldoDisponible:
        raise SaldoInsuficienteError(f"El monto solicitado
({montoSolicitado}) excede su saldo disponible
({saldoDisponible}).")
    return "Préstamo aprobado"

try:
    saldoDisponible = 5000
    montoSolicitado = 7000
    solicitarPrestamo(saldoDisponible, montoSolicitado)
except SaldoInsuficienteError as e:
    print(e)
```

EXCEPCIONES PERSONALIZADAS

La creación de excepciones personalizadas permite manejar errores específicos de una forma más controlada y precisa, especialmente cuando los casos de error no están cubiertos por las excepciones estándar del lenguaje.

Una excepción personalizada se define mediante una clase que hereda de *Exception*, lo que te permite asociar un tipo propio a ese error.

Este abordaje es útil cuando se desean distinguir errores únicos en la lógica de negocio, facilitando el manejo de casos complejos o más específicos. Al lanzar una excepción personalizada, podemos incluir información detallada sobre la naturaleza del error, mejorando la claridad y el control de las fallas.

PROCESAMIENTO DE EXCEPCIONES

Hemos visto cómo atrapar excepciones, es necesario ahora que veamos qué se supone que hagamos al atrapar una excepción. En primer lugar, podríamos ejecutar alguna lógica particular del caso como: cerrar un archivo, realizar un procesamiento alternativo al del bloque *try*, etc.

Pero, más allá de esto, tenemos algunas opciones genéricas que consisten en: dejar constancia de la ocurrencia de la excepción, propagar la excepción o, incluso, hacer ambas cosas.

Para dejar constancia de la ocurrencia de la excepción, se puede escribir en un archivo de log o simplemente mostrar un mensaje en pantalla. Generalmente cuando se deja constancia de la ocurrencia de una excepción se suele brindar alguna información del contexto en que ocurrió la excepción, por ejemplo: tipo de excepción ocurrida, momento en que ocurrió la excepción y cuáles fueron las llamadas previas a la excepción.

El objetivo de esta información es facilitar el diagnóstico en caso de que alguien deba corregir el programa para evitar que la excepción siga apareciendo.

Lic. Yaps David P.

AUTOEVALUACIÓN

La autoevaluación es crucial para identificar mejoras y fortalezas de forma objetiva, facilitando la fijación de metas y la toma de acciones para alcanzarlas.