Errores y Excepciones

Errores de sintaxis	Errores semánticos	Errores de ejecución
Son detectados por el intérprete (o por el compilador, según el lenguaje que estemos utilizando) al procesar el código fuente y generalmente son consecuencia de errores de tipeo al escribir el programa. En el caso de Python nos informa con un mensaje SyntaxError.	Sucede cuando un programa no produce el resultado esperado pero tampoco hay mensajes de error. Generalmente se debe a un algoritmo incorrecto, error u omisión de una sentencia.	El origen se debe a una o más causas, pueden ser errores de programación o debido a recursos externos, por ejemplo intentar leer un archivo dañado.

El control de errores y excepciones hacen que un programa sea más robusto.

Excepciones

Los errores de ejecución son llamados excepciones. Durante la ejecución de un programa, cualquier línea de código puede generar una excepción. Las más frecuentes son:

Exception	AssertionError	IndexError	KeyError	TypeError	ValueError	NameError	ZeroDivisionError	IOError
Todas las excepciones son de tipo Exception	Una instrucción assert falló	Intento de acceder a una secuencia con un índice fuera de rango	Intento de acceder a un diccionario con una clave inexistente	Aplica una operación a un valor de tipo inapropiado	Aplica una operación con un parámetro de tipo apropiado pero su valor no lo es.	Variable no definida	Intenta dividir un número por 0	Error de entrada/ salida, por ejemplo intento de acceder a un archivo

Manejo de excepciones

Cuando ocurre un error o una excepción, el programa se detendrá y generará un mensaje de error, los bloques de control son: try, except y finally

Las excepciones se pueden manejar usando la declaración try. Dentro del bloque try se ubica el código que pueda llegar dar una excepción. A continuación se ubica el bloque except, que se encarga de capturar la excepción y da la posibilidad de controlarla.	Se pueden definir tantos bloques de except, como sea necesario -sólo uno se ejecutará- y se puede utilizar except sin especificar el tipo de excepción a capturar (en cuyo caso captura cualquiera) si es este caso debe ser la última de las instrucciones except:	Se puede usar else para definir un bloque de código que se ejecutará si no se generaron errores:	Si se especifica el bloque finally, se ejecutará independientemente de si el bloque try genera un error o no. Es posible tener un try sólo con finally	Se puede emitir una excepción si se produce una condición. Para emitir (o aumentar) una excepción, hay que usar la palabra raise. Esta también se utiliza para generar una excepción. Se puede definir qué tipo de error generar y el texto a emitirle al usuario.

`try:`
`  print(x)`
`except:`
`  print("Ocurrió una excepción")`
	`try:`
	`  print(x)`
	`except NameError:`
	`  print("La variable x no está definida")`
	`except:`
	`  print("Algo más salió mal…")`
		`try:`
		`  print("Hola")`
		`except:`
		`  print("Algo salió mal")`
		`else:`
		`  print("Nada salió mal")`
			`try:`
			`  print(x)`
			`except:`
			`  print("Algo salió mal")`
			`finally:`
			`  print("El 'try except' ha finalizado")`
				`x = -1`
				`if x < 0:`
				`  raise Exception("Perdón, no hay valor válido debajo de cero")`
				
				`x = "Hola!"`
				`if not type(x) is int:`
				`  raise TypeError("Sólo son permitidos números enteros")`

Si dentro de una función se emite una excepción pero no es controlada, esta se propaga hacia la función que la invocó; si esta otra tampoco la controla, continúa propagándose hasta llegar a la función inicial del programa, y si ésta tampoco la maneja se interrumpe la ejecución del programa.

Una vez que capturamos las excepciones podemos realizar procesos alternativos, por ejemplo dejar constancia detallada en un archivo .log o emitir un mensaje o incluso ambas acciones. El objetivo de dejar constancia es corregir el programa.

Validaciones

Las validaciones permiten asegurar que los valores con los que se van a operar estén dentro de determinado dominio.

Comprobar contenido	Comprobar por tipo	Comprobar por característica
Significa que comprobaremos que el contenido de las variables (valores ingresados por el usuario, de archivos, etc) a utilizar, estén dentro de los valores con los cuáles se pueden operar. A veces no es posible hacerlo pues es costoso corroborar las precondiciones, por lo tanto se realizan sólo cuando es posible.	Significa que nos interesa el tipo del dato que vamos a tratar de validar, para ello se utiliza la función type(variable).	Significa comprobar si una variable tiene determinada característica. Para comprobar si una variable tiene o no una función se utiliza la función hasattr(variable, atributo), donde atributo puede ser el nombre de la función o de la variable que se quiera verificar.

Documentación y comentarios

En general, en el desarrollo de programas y aplicaciones, la documentación es un trabajo que se posterga. En consecuencia, cuando llega el momento de escribirla, se construye documentación que no refleja en profundidad y con detalles, el trabajo realizado. Posteriormente, cuando el código evoluciona con modificaciones y actualizaciones, la tarea se vuelve mucho más difícil.

Si bien el código fuente transmite el algoritmo, hay descripciones que aportan claridad, por ejemplo determinar él o los problemas, fundamentar el diseño, describir el análisis funcional, detallar las razones en que se basan las decisiones, determinar los objetivos, etc. Un desarrollo bien documentado es una parte importante de todo el proyecto.

Documentación	Comentarios
Se escribe entre “”” ó ’’’ (triples comillas simples o dobles)	Se escribe # al comienzo de la línea de comentario
Explica qué hace el código. Está dirigida a quién necesite utilizar la función o módulo, para que pueda entender cómo usarla sin necesidad de leer el código fuente.	Explica cómo funciona el código y en algunos casos por qué se decidió implementarlo así. Los comentarios están dirigidos a quien esté leyendo el código fuente.

def CalcularFactorial(num):
"""" Función recursiva que devuelve el factorial de un número pasado como parámetro"""
if num == 1:
return 1
else:
return num*CalcularFactorial(num-1)
	def CalcularFactorial(num):
	if num == 1:
	return 1
	else:
	return num*CalcularFactorial(num-1)
	# Recibe un número si es 1 devuelve que el factorial es 1, sino
	# acumula el producto del número con el cálculo
	# del factorial del numero-1.

Código autodocumentado

En esta técnica, el objetivo es elegir nombres de funciones y variables o también agregar comentarios en las líneas del código, de tal manera que la documentación sea innecesaria. La desventaja es que hay que tener en cuenta, al elegir los nombres, que sea descriptivos y cortos, que no siempre es posible. Además, para saber qué hace y cómo, implica leer todo el código y no describe los detalles a otros niveles.

Con comentarios	Con nombres de variables descriptivos

an = 7.80 # ancho de la figura
al = 15.45 # alto de la figura
a = an * al # área de la figura
	ancho_del_rectangulo = 7.80
	alto_del_rectangulo = 15.45
	area_del_rectangulo = ancho_del_rectangulo * alto_del_rectangulo

Contratos

Las pre y postcondiciones son un contrato entre el código invocante y el invocado:

Precondiciones	Postcondiciones	assert
Son las condiciones que deben cumplirse antes de ejecutar el programa y para que se comporte correctamente, es decir cómo deben ser los parámetros que recibe, cómo debe ser el estado global, etc. Por ejemplo, en una función que divide dos números, las precondiciones son que los parámetros son números y que el divisor es distinto de 0 (cero).	Son las condiciones que se cumplirán una vez finalizada la ejecución de la función (asumiendo que se cumplen las precondiciones): es decir cómo será el valor de retorno, si los parámetros recibidos o variables globales son alteradas, si se emiten, si modifican archivos, etc. Para el ejemplo dado, dadas las precondiciones se puede asegurar que devolverá un número correspondiente al cociente.	Precondiciones y postcondiciones son assertions, es decir, afirmaciones. Si llegaran a ser falsas significa que existe algún error en el algoritmo. Es recomendable comprobar estas afirmaciones con la instrucción assert. Esta recibe una condición a verificar, si es verdadera la instrucción no hace nada; en caso contrario produce un error. Puede recibir un mensaje que mostrará en caso que la condición no se cumpla. Se debe implementar en la etapa de desarrollo.

Uso de assert

Se deben usar aserciones para probar las condiciones que nunca deberían ocurrir.

assert() en testing	assert() en funciones	assert() con clases

Es útil para escribir tests unitarios o units tests. Ejemplo:
 	
`def calcula_media(lista):`
`  return sum(lista)/len(lista)`
 	
Es muy importante testear el software, para asegurarse de que está libre de errores. Con assert() podemos realizar estas comprobaciones de manera automática.
 	
`assert(calcula_media([5, 10, 7.5]) == 7.5)`
`assert(calcula_media([4, 8]) == 6)`
	Es útil cuando queremos realizar alguna comprobación dentro de una función. En el siguiente ejemplo tenemos una función que sólo suma las variables si son números enteros:
	
	`# Funcion suma de variables enteras`
	
	`def suma(a, b):`
	`  assert(type(a) == int)`
	`  assert(type(b) == int)`
	`  return a+b`
	
	`# Error, ya que las variables no son int`
	`suma(3.0, 5.0)`
	`# Ok, los argumentos son int`
	`suma(3, 5)`
		Verificar que un objeto pertenece a una clase determinada. Ejemplo:
		
		`class MiClase():`
		`  pass`
		
		`class MiOtraClase():`
		`  pass`
		
		`mi_objeto = MiClase()`
		`mi_otro_objeto = MiOtraClase()`
		`# Ok`
		`assert(isinstance(mi_objeto, MiClase))`
		`# Ok`
		`assert(isinstance(mi_otro_objeto, MiOtraClase))`
		`# Error, mi_objeto no pertenece a MiOtraClase`
		`assert(isinstance(mi_objeto, MiOtraClase))`
		`# Error, mi_otro_objeto no pertenece a MiClase`
		`assert(isinstance(mi_otro_objeto, MiClase))`

Invariantes

Se refieren a estados o situaciones que no cambian dentro de un contexto o código.

Invariante de ciclo	Invariante de clase
Permite conocer cómo llegar desde las precondiciones hasta las postcondiciones, cuando la implementación se compone de un ciclo. El invariante de ciclo es, entonces, una aseveración (assertions) que debe ser verdadera al comienzo de cada iteración. https://es.wikipedia.org/wiki/Ciclo_invariante	Son condiciones que deben ser ciertas durante toda la vida de un objeto. Una clase tiene dos características fundamentales que la definen: estado y comportamiento. El estado viene definido por la información de sus propiedades (atributos) y el comportamiento viene definido en sus métodos que utilizarán dichos atributos. Los invariantes de clase son propiedades globales de una clase que tienen que ser conservadas por todas las rutinas que la componen. https://es.wikipedia.org/wiki/Invariantes_de_clase#Clases_invariantes_y_herencia