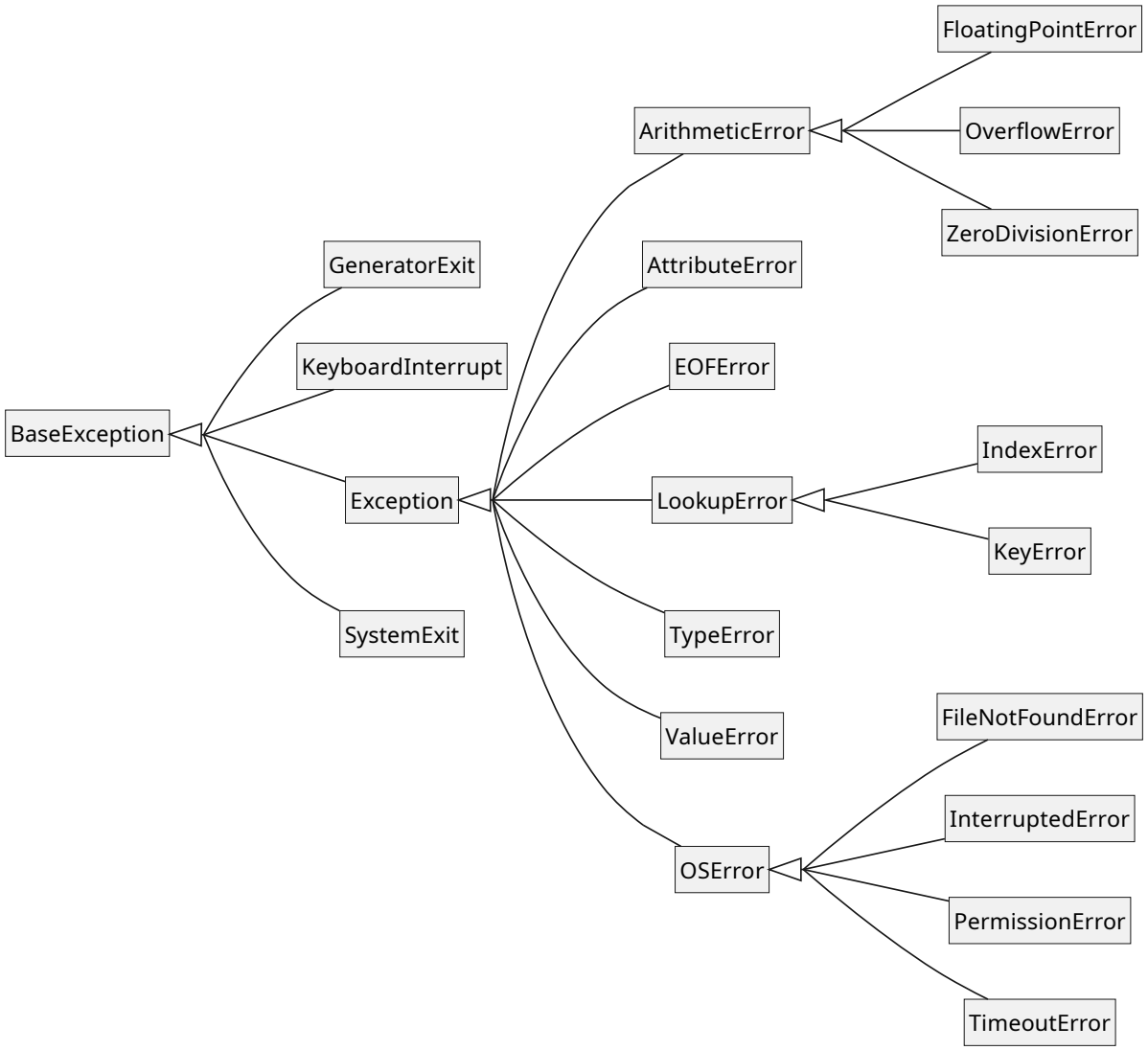
1. Introducción

En Pythonlos errores que se producen durante la ejecución de un programa pueden ser tratados mediante un poderoso mecanismo denominado **manejo de excepciones**, que permite evitar que el programa finalice abruptamente ante errores, permitiendo manejar la situación y continuar con su ejecución de forma controlada.

Las excepciones incluidas en el lenguaje son objetos pertenecientes a la siguiente jerarquía de clases (vista parcial)[1]:



Todas las excepciones más comunes heredan de la clase `Exception` y esta a su vez de `BaseException`. Las excepciones se lanzan (generan) automáticamente cuando ocurre un error o manualmente mediante el uso de una sentencia `raise`.

[1] Para la vista completa visite: `https://docs.python.org/es/3.13/library/exceptions.html#exception-hierarchy`

Por ejemplo, cuando el usuario ingresa un valor y este se intenta convertir a un dato del tipo entero (`int`), pero el texto (`str`) no tiene dicho formato, esto causa un `ValueError`:

Programa 1: `ValueError` en el REPL de python

```
>>> n = int(input('Ingrese un número: '))
Ingrese un número: hola
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    n = int(input('Ingrese un número: '))
        ~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'hola'
```

El mensaje indicará en qué script, línea y módulo ocurrió el error, y nos dará una breve descripción de lo ocurrido. En este caso que el literal `'hola'` no es un entero en base 10 válido.

2. Manejo de Excepciones

Las excepciones deben ser *capturadas* a través de bloques `try...except..finally`. Estos asemejan a la estructura de un `if-elif`, pero en lugar de evaluar una expresión booleana, se encierra en el bloque `try` el código que podría provocar errores y luego se *atiende* la excepción en un bloque `except` preparado para un determinado tipo de error. El bloque `finally` se ejecuta haya o no ocurrido una excepción, y se utiliza habitualmente para hacer procesos de *limpieza*: cerrar archivos o conexiones, borrar o liberar recursos, etc.

Programa 2: Captura de excepciones

```
1  # excepciones.py
2
3  if __name__ == '__main__':
4    try:
5      n1 = int(input('Ingrese un número: '))
6      n2 = int(input('Ingrese un número: '))
7      result = n1 / n2
8      print(f'{n1} / {n2} = {result}')
9    except ValueError:
10     print('Algún valor no fue un entero válido...')
11   except ZeroDivisionError:
12     print('El segundo valor no puede ser cero...')
13   finally:
14     print('El bloque finally siempre se ejecuta...')
```

Las posibles salidas de este programa son las siguientes:

```
Ingrese un número: 9
Ingrese un número: 3
9 / 3 = 3.0
El bloque finally siempre se
   ejecuta...
```

```
Ingrese un número: h
Algún valor no fue un entero
   válido...
El bloque finally siempre se
   ejecuta...
```

```
Ingrese un número: 3
Ingrese un número: j
Algún valor no fue un entero
   válido...
El bloque finally siempre se
   ejecuta...
```

```
Ingrese un número: 9
Ingrese un número: 0
El segundo valor no puede ser
    cero...
El bloque finally siempre se
   ejecuta...
```

Debe notar que apenas ocurre el error, se interrumpe el flujo del programa, dejando sin efecto a las siguiente líneas dentro del bloque `try`. Así, cuando el primer número ingresado no es un entero válido, el segundo `input` no se ejecuta, salta directamente al bloque correspondiente que maneja la excepción `ValueError`.

En este programa se capturan dos tipos de excepciones diferentes, `ValueError` y `ZeroDivisionError` en dos bloques `except` distintos. Si no es importante diferenciar el error, podríamos declarar ambos en un solo bloque `except` que capture una **tupla** con todas las excepciones esperadas:

Programa 3: Captura de varias excepciones en un único bloque `except`

```python
# excepciones2.py

if __name__ == '__main__':
  try:
    n1 = int(input('Ingrese un número: '))
    n2 = int(input('Ingrese un número: '))
    result = n1 / n2
    print(f'{n1} / {n2} = {result}')
  except (ValueError, ZeroDivisionError):
    print('Ocurrió un error...')
  finally:
    print('El bloque finally siempre se ejecuta...')
```

3. Precedencia en la captura de excepciones

Es importante destacar que importa el orden en el cual declaramos los bloques `except` y qué tipo de errores deseamos capturar. En el Programa 3 el error `IndexError` nunca será

alcanzado debido a que LookupError lo captura primero:

```
# precedencia_except.py

if __name__ == '__main__':
  try:
    l = [1, 2, 3]
    print(l[3])
  except LookupError:
    print('Se captura "LookupError"')
  except IndexError:
    print('Se captura "IndexError"')
```

Recordando que `IndexError` hereda de `LookupError`, podemos decir que este último representa un error más general, es decir, que contempla a `IndexError` y a `KeyError`. Al ser así, `IndexError` nunca se ejecutará. El orden de captura siempre debe ser del más específico (subclase más específica) al menos específico (superclases).

A su vez, está considerado **mala práctica** capturar `Exception`, lo cual, capturaría cualquier error por debajo de la jerarquía sin saber cual fue, y mucho menos, no hacer nada al respecto:

Programa 4: **Mala práctica** de captura de excepciones

```
try:
  # mucho código que podría lanzar errores
  ...
except Exception:
  pass
finally:
  print('Aquí no pasó nada...')
```

4. Lanzar excepciones

Será de utilidad para algunos diseños de software, *lanzar* o *levantar* excepciones si ocurriera un error en una clase propia. Esto se hace a través de la palabra reservada `raise` seguida de un objeto del tipo `Exception`.

Programa 5: Lanzar una excepción

```
class Vehiculo:
  # ...
class Auto:
  # ...
```

```
5  class Moto:
6    # ...
7  class Lancha:
8    # ...
9
10 class Cochera:
11   def __init__(self) -> None:
12     self.__CAPACIDAD_MAX = 30
13     self.__coches: list[Vehiculo] = []
14
15   def guardar(self, v: Vehiculo):
16     if not isinstance(v, Vehiculo):
17       raise ValueError(f'El objeto {v} debe ser una instancia de
     Vehiculo')
18     if len(self.__coches) == self.__CAPACIDAD_MAX:
19       raise OverflowError("Capacidad máxima alcanzada. No se
     puede guardar más vehículos.")
20     self.__coches.append(v)
```

En este diseño se optó por utilizar `OverflowError` para indicar que la cochera ya está llena, para utilizar los errores que ya están incluidos dentro de `Python`. Pero, de ser necesario hacer una manejo más específico de un diseño, también podría crear una excepción propia. Esto es tan sencillo como crear una clase que herede de `Exception`.

Programa 6: Excepciones a medida

```
class CapacidadMaximaError(Exception):
  pass # deliveradamente vacía, no hace falta nada más

# en el método:
  if len(self.__coches) == self.__CAPACIDAD_MAX:
    raise CapacidadMaximaError("Capacidad máxima alcanzada. No se
    puede guardar más vehículos.")
```

5. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial de Python: `https://docs.python.org/es/3.13/library/exceptions.html`
- Interprete Visual de Python: `http://www.pythontutor.com/visualize.html`