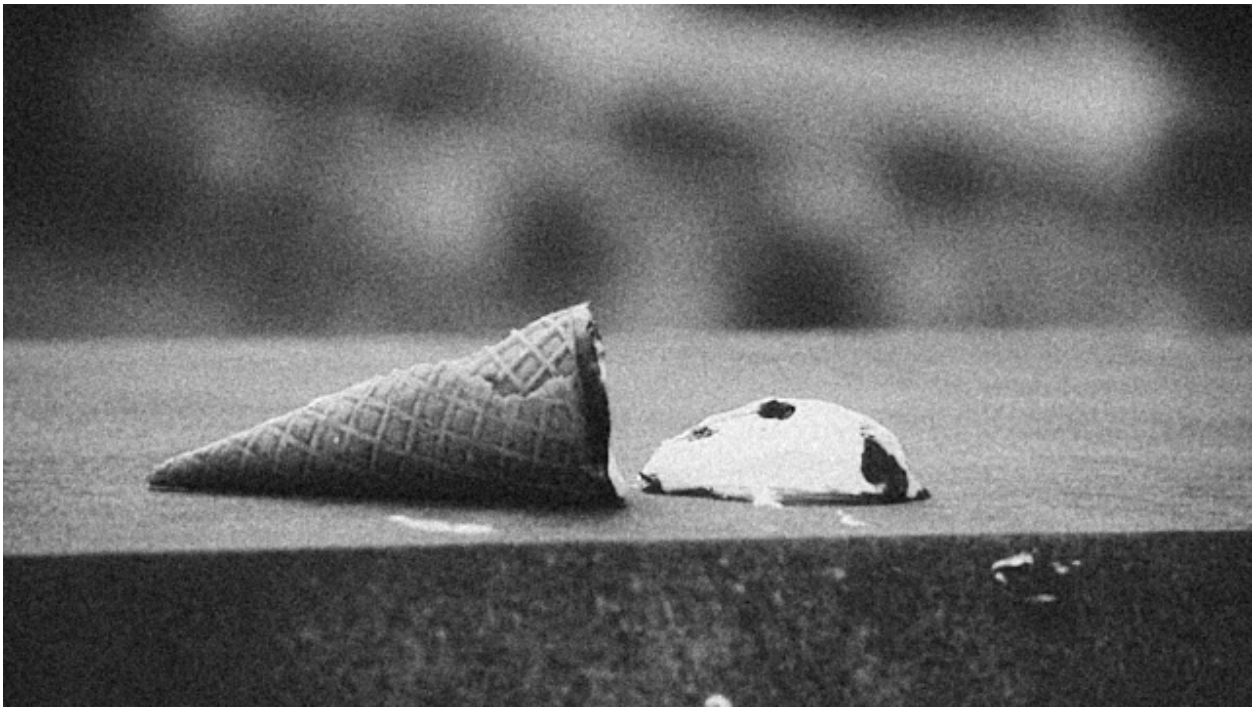
→ Solución a todos los ejercicios

AMPLIAR CONOCIMIENTOS

- Supercharge Your Classes With Python super()
- Inheritance and Composition: A Python OOP Guide
- OOP Method Types in Python: @classmethod vs @staticmethod vs Instance Methods
- Intro to Object-Oriented Programming (OOP) in Python
- Pythonic OOP String Conversion: ___repr___ vs ___str___
- @staticmethod vs @classmethod in Python
- Modeling Polymorphism in Django With Python
- Operator and Function Overloading in Custom Python Classes
- Object-Oriented Programming (OOP) in Python 3
- Why Bother Using Property Decorators in Python?

6.3 Excepciones



Una **excepción** es el bloque de código que se lanza cuando se produce un **error** en la ejecución de un programa Python.[1]

De hecho ya hemos visto algunas de estas excepciones: accesos fuera de rango a listas o tuplas, accesos a claves inexistentes en diccionarios, etc. Cuando ejecutamos código que podría fallar bajo ciertas circunstancias, necesitamos también manejar, de manera adecuada, las excepciones que se generan.

6.3.1 Manejando errores

Si una excepción ocurre en algún lugar de nuestro programa y no es capturada en ese punto, va subiendo (burbujeando) hasta que es capturada en alguna función que ha hecho la llamada. Si en toda la «pila» de llamadas no existe un control de la excepción, Python muestra un mensaje de error con información adicional:

```
>>> def intdiv(a, b):
...     return a // b
...

>>> intdiv(3, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in intdiv
ZeroDivisionError: integer division or modulo by zero
```

Para manejar (capturar) las excepciones podemos usar un bloque de código con las palabras reservadas `try` and `except`:

```
>>> def intdiv(a, b):
...     try:
...         return a // b
...     except:
...         print('Please do not divide by zero...')
...

>>> intdiv(3, 0)
Please do not divide by zero...
```

Aquel código que se encuentre dentro del bloque `try` se ejecutará normalmente siempre y cuando no haya un error. Si se produce una excepción, ésta será capturada por el bloque `except`, ejecutándose el código que contiene.

Consejo: No es una buena práctica usar un bloque `except` sin indicar el **tipo de excepción** que estamos gestionando, no sólo porque puedan existir varias excepciones que capturar sino

[1] Foto original por Sarah Kilian en Unsplash.

porque, como dice el *Zen de Python*: «explícito» es mejor que «implícito».

Especificando excepciones

En el siguiente ejemplo mejoraremos el código anterior, capturando distintos tipos de excepciones:

- `TypeError` por si los operandos no permiten la división.
- `ZeroDivisionError` por si el denominador es cero.
- `Exception` para cualquier otro error que se pueda producir.

Veamos su implementación:

```
>>> def intdiv(a, b):
...     try:
...         result = a // b
...     except TypeError:
...         print('Check operands. Some of them seems strange...')
...     except ZeroDivisionError:
...         print('Please do not divide by zero...')
...     except Exception:
...         print('Ups. Something went wrong...')
...

>>> intdiv(3, 0)
Please do not divide by zero...

>>> intdiv(3, '0')
Check operands. Some of them seems strange...
```

Importante: Las excepciones predefinidas en Python no hace falta importarlas previamente. Se pueden usar directamente.

Cubriendo más casos

Python proporciona la cláusula `else` para saber que todo ha ido bien y que no se ha lanzado ninguna excepción. Esto es relevante a la hora de manejar los errores.

De igual modo, tenemos a nuestra disposición la cláusula `finally` que se ejecuta siempre, independientemente de si ha habido o no ha habido error.

Veamos un ejemplo de ambos:

```
>>> values = [4, 2, 7]

>>> user_index = 3

>>> try:
...     r = values[user_index]
... except IndexError:
...     print('Error: Index not in list')
... else:
...     print(f'Your wishes are my command: {r}')
... finally:
...     print('Have a good day!')
...
Error: Index not in list
Have a good day!

>>> user_index = 2

>>> try:
...     r = values[user_index]
... except IndexError:
...     print('Error: Index not in list')
... else:
...     print(f'Your wishes are my command: {r}')
... finally:
...     print('Have a good day!')
...
Your wishes are my command: 7
Have a good day!
```

Ejercicio

Sabiendo que `ValueError` es la excepción que se lanza cuando no podemos convertir una cadena de texto en su valor numérico, escriba una función `get_int()` que lea un valor entero del usuario y lo devuelva, iterando mientras el valor no sea correcto.

Ejecución a modo de ejemplo:

```
Give me an integer number: ten
Not a valid integer. Try it again!
Give me an integer number: diez
Not a valid integer. Try it again!
Give me an integer number: 10
```

Trate de implementar tanto la versión recursiva como la versión iterativa.

6.3.2 Excepciones propias

Nivel avanzado

Python ofrece una gran cantidad de excepciones predefinidas. Hasta ahora hemos visto cómo gestionar y manejar este tipo de excepciones. Pero hay ocasiones en las que nos puede interesar crear nuestras propias excepciones. Para ello tendremos que crear una clase *heredando* de `Exception`, la clase base para todas las excepciones.

Veamos un ejemplo en el que creamos una excepción propia controlando que el valor sea un número entero:

```
>>> class NotIntError(Exception):
...     pass
...

>>> values = (4, 7, 2.11, 9)

>>> for value in values:
...     if not isinstance(value, int):
...         raise NotIntError(value)
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
__main__.NotIntError: 2.11
```

Hemos usado la sentencia `raise` para «elevar» esta excepción, que podría ser controlada en un nivel superior mediante un bloque `try - except`.

Nota: Para crear una excepción propia basta con crear una clase vacía. No es necesario incluir código más allá de un `pass`.

Mensaje personalizado

Podemos personalizar la excepción añadiendo un mensaje más informativo. Siguiendo el ejemplo anterior, veamos cómo introducimos esta información:

```
>>> class NotIntError(Exception):
...     def __init__(self, message='This module only works with integers. Sorry!'):
...             super().__init__(message)
...

>>> raise NotIntError()
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  File "<stdin>", line 1, in <module>
__main__.NotIntError: This module only works with integers. Sorry!
```

Podemos ir un paso más allá e incorporar en el mensaje el propio valor que está generando el error:

```
>>> class NotIntError(Exception):
...     def __init__(self, value, message='This module only works with integers. Sorry!'):
...         self.value = value
...         self.message = message
...         super().__init__(self.message)
...
...     def __str__(self):
...         return f'{self.value} -> {self.message}'
...

>>> raise NotIntError(2.11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.NotIntError: 2.11 -> This module only works with integers. Sorry!
```

6.3.3 Aserciones

Si hablamos de control de errores hay que citar una sentencia en Python denominada `assert`. Esta sentencia nos permite comprobar si se están cumpliendo las «expectativas» de nuestro programa, y en caso contrario, lanza una excepción informativa.

Su sintaxis es muy simple. Únicamente tendremos que indicar una expresión de comparación después de la sentencia:

```
>>> result = 10

>>> assert result > 0

>>> print(result)
10
```

En el caso de que la condición se cumpla, no sucede nada: el programa continúa con su flujo normal. Esto es indicativo de que las expectativas que teníamos se han satisfecho.

Sin embargo, si la condición que fijamos no se cumpla, la aserción devuelve un error `AssertionError` y el programa interrupme su ejecución:

```
>>> result = -1

>>> assert result > 0
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-29-e2efe60b0c46> in <module>
----> 1 assert result > 0

AssertionError:
```

Podemos observar que la excepción que se lanza no contiene ningún mensaje informativo. Es posible personalizar este mensaje añadiendo un segundo elemento en la *tupla* de la aserción:

```
>>> assert result > 0, 'El resultado debe ser positivo'
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-31-f58052ce672b> in <module>
----> 1 assert result > 0, 'El resultado debe ser positivo'

AssertionError: El resultado debe ser positivo
```

AMPLIAR CONOCIMIENTOS

- Python Exceptions: An introduction
- Python KeyError Exceptions and How to Handle Them
- Understanding the Python Traceback