Tema 5. Iteradores, generadores y gestores de contexto

Vicente Benjumea García

Programación-II
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 5. Iteradores, generadores y gestores de contexto

- Utilización de iteradores.
- Desarrollo de iteradores.
- Desarrollo de generadores.
- Sentencia with y los gestores de contexto.
- Desarrollo de gestores de contexto.

Utilización de iteradores

Iterables y el bucle `for`

- El bucle **for** permite iterar y recorrer **todos** los elementos de un *iterable*. En cada iteración, la variable toma el siguiente valor del iterable.
  - Iterables son: `range`, `str`, `tuple`, `list`, `set`, `dict`, *ficheros*, *csv*, etc.

```
for x in range(5):
    print(x)
# 0 1 2 3 4
```

```
for x in "hola":
    print(x)
# h o l a
```

```
for x in [2, 3, 5, 7]:
    print(x)
# 2 3 5 7
```

- Un **iterable** es un objeto al que se le puede aplicar la función **iter(obj)**, que devuelve un **iterador** vinculado a ese iterable.
- Un **iterador** es un objeto que cuando se le aplica la función **next(it)**, devuelve el siguiente elemento del iterable vinculado. El iterador permite acceder a los elementos de un iterable.
- El bucle **for** utiliza internamente las funciones **iter()** y **next()** para iterar sobre **todos** los elementos del iterable.
- Se pueden utilizar los *iteradores* **explícitamente** para recorrer los elementos de un *iterable* en un bucle **while**, y terminar la iteración antes de llegar al final.

Utilización de iteradores

Iteradores

- `iter(iterable)`: devuelve un objeto **iterador** vinculado al *iterable*, que permite iterar sobre los elementos del iterable.
- `next(iterator)`: devuelve el siguiente elemento del iterable vinculado. En caso de que no haya más elementos, lanza `StopIteration`.
- `next(iterator, default)`: devuelve el siguiente elemento del iterable vinculado. En caso de que no haya más elementos, devuelve el valor del segundo argumento (*default*), y no se lanza la excepción.

```
def buscar(lista: list, elem: Any) -> Any:
    it = iter(lista)
    x = next(it, None)
    while (x is not None) and (x != elem):
        x = next(it, None)
    return x
```

```
def buscar(lista: list, elem: Any) -> Any
    try:
        it = iter(lista)
        x = next(it)
        while (x != elem):
            x = next(it)
    except StopIteration:
        x = None
    return x
```

Desarrollo de iteradores

- Para que una clase sea **iterable**, debe definir un método `__iter__()`, que cree y devuelva un nuevo objeto **iterator**.
  - A veces, se desea crear un **iterador** que permita iterar sobre los elementos de un objeto interno existente, que ya es **iterable**. En este caso, lo más simple es definir el método `__iter__()` que cree y devuelva el propio iterador del objeto iterable interno. Por ejemplo:

```
from typing import Iterator
class Contenedor:
    def __init__(self, ..., *args, **kwargs) -> None:
        self.__lista: list[ ...tipo-elem... ] = list()
        super().__init__(*args, **kwargs)

    def __iter__(self) -> Iterator[ ...tipo-elem... ]:
        return iter(self.__lista)
```

Desarrollo de iteradores. Ejemplo ListaNumeros

Ejemplo: lista de números comprendidos entre un valor mínimo y máximo (sin incluir).

```
from typing import Iterator
class ListaNumeros:
    def __init__(self, minimo: int, maximo: int, *args, **kwargs) -> None:
        self.__lista: list[int] = list()
        self.__minimo = minimo
        self.__maximo = maximo
        super().__init__(*args, **kwargs)

    def anyadir(self, numero: int) -> None:
        if (self.__minimo <= numero < self.__maximo):
            self.__lista.append(numero)

    def __iter__(self) -> Iterator[int]:
        return iter(self.__lista)
```

```
def main() -> None:
    listanums = ListaNumeros(10, 20)
    listanums.anyadir(5)
    listanums.anyadir(12)
    listanums.anyadir(25)
    listanums.anyadir(17)
    for x in listanums:
        print(x)        # 12 17

if __name__ == "__main__":
    main()
```

Desarrollo de iteradores

- Para que una clase sea **iterable**, debe definir un método `__iter__()`, que cree y devuelva un nuevo objeto **iterator**.
- La **clase** del objeto **iterator** debe definir los siguientes métodos:
  - El método `__init__()`: almacena e inicializa la información necesaria.
  - El método `__iter__()`: simplemente devuelve el propio objeto **self**, sin reinicializar el iterador. Un iterador también es un iterable.
  - El método `__next__()`: devuelve el siguiente elemento del *iterable*. Lanzará la excepción `StopIteration` cuando ya no haya más elementos en el *iterable*.

```
from typing import Iterator
class Contenedor:
    def __iter__(self) -> Iterator[ ...tipo-elem... ]:
        return Contenedor.__Iterator(... información necesaria ...)
    #-------------------------
    class __Iterator:
        def __init__(self, ... información necesaria ..., *args, **kwargs) -> None:
            ...
            super().__init__(*args, **kwargs)
        def __iter__(self) -> Iterator[ ...tipo-elem... ]:
            return self
        def __next__(self) -> ...tipo-elem... :
            if no_hay_mas_elementos:
                raise StopIteration()
            ...
            return elemento
```

Desarrollo de iteradores. Ejemplo Intervalo con iterador

- Desarrollar una clase `Intervalo`, que permita iterar sobre los números enteros dentro de un intervalo especificado. El intervalo podrá especificar el valor inicial y final del intervalo, así como el paso de incremento. El valor final no forma parte de los elementos del intervalo. El paso podrá ser positivo o negativo.
  - **Nota:** implementaremos la iteración mediante un **iterador**.

Desarrollo de iteradores. Ejemplo Intervalo con iterador

```
from typing import Optional, Iterator

class Intervalo:
    def __init__(self, inicial: int, final: Optional[int]=None, paso: Optional[int]=None,
                 *args, **kwargs) -> None:
        if (paso is None) or (paso == 0):
            paso = 1
        if final is None:
            final = inicial
            inicial = 0
        self.__inicial = inicial
        self.__final = final
        self.__paso = paso
        super().__init__(*args, **kwargs)

    def __iter__(self) -> Iterator[int]:
        return Intervalo.__Iterator(self.__inicial, self.__final, self.__paso)
```

Desarrollo de iteradores. Ejemplo Intervalo con iterador

```
#class Intervalo: # (continuación)
    class __Iterator:
        def __init__(self, inicial: int, final: int, paso: int, *args, **kwargs) -> None:
            self.__inicial = inicial
            self.__final = final
            self.__paso = paso
            self.__valor = self.__inicial
            super().__init__(*args, **kwargs)

        def __iter__(self) -> Iterator[int]:
            return self

        def __next__(self) -> int:
            valor = self.__valor
            if ((self.__paso > 0 and valor >= self.__final)
                or (self.__paso < 0 and valor <= self.__final)):
                raise StopIteration()
            self.__valor += self.__paso
            return valor

def main() -> None:
    for x in Intervalo(4):          # 0 1 2 3
        print(x)
    for x in Intervalo(1, 4, 2):    # 1 3
        print(x)
    for x in Intervalo(4, 1, -1):   # 4 3 2
        print(x)

if __name__ == "__main__":
    main()
```

Desarrollo de iteradores. Ejemplo Alumnos con iterador

- Desarrollar una clase Alumnos, que permita añadir notas de alumnos, donde cada alumno puede tener múltiples notas. Esta clase debe ser **iterable**, donde cada elemento de la iteración será una tupla con el nombre y la nota media del alumno.
  - **Nota:** implementaremos la iteración mediante un **iterador**.

Desarrollo de iteradores. Ejemplo Alumnos con iterador

```python
from typing import NamedTuple, Iterator
class Alumno(NamedTuple):
    nombre: str
    nota: float

class Alumnos:
    def __init__(self, *args, **kwargs) -> None:
        self.__alumnos: dict[str, list[float]] = dict() # clave: nombre; valor: lista-notas
        super().__init__(*args, **kwargs)

    def anyadir(self, nombre: str, nota: float) -> None:
        if nombre not in self.__alumnos:
            self.__alumnos[nombre] = list()
        self.__alumnos[nombre].append(nota)

    def __iter__(self) -> Iterator[Alumno]:
        return Alumnos.__Iterator(self.__alumnos)

    #-------------------------
    class __Iterator:
        def __init__(self, alumnos, *args, **kwargs) -> None:
            self.__alumnos = alumnos
            self.__it = iter(self.__alumnos.items())
            super().__init__(*args, **kwargs)

        def __iter__(self) -> Iterator[Alumno]:
            return self

        def __next__(self) -> Alumno:
            (nombre, notas) = next(self.__it)
            return Alumno(nombre, sum(notas)/len(notas))
```

```python
def main() -> None:
    alumnos = Alumnos()
    alumnos.anyadir("pepe", 7.5)
    alumnos.anyadir("ana", 7.0)
    alumnos.anyadir("pepe", 8.5)
    alumnos.anyadir("ana", 8.0)
    for alumno in alumnos:
        print(alumno)

if __name__ == "__main__":
    main()

# Alumno(nombre='pepe', nota=8.0)
# Alumno(nombre='ana', nota=7.5)
```

Desarrollo de generadores

Generadores

- Los **generadores** son un mecanismo alternativo y simple para **crear iteradores**, que generan una secuencia de elementos, **según se va iterando sobre ellos**. Existen dos tipos de generadores: **expresiones generadoras**, y **funciones generadoras**.

Expresiones generadoras (generador por comprensión)

- Las **expresiones generadoras** crean un **generador** (**iterador**) de la secuencia de elementos determinados por el "generador por comprensión", iterando sobre los elementos de un *iterable*, seleccionando elementos según alguna condición, y transformando los elementos seleccionados:

```
generador = (x for x in range(10))                # elementos: 0 1 2 3 4 5 6 7 8 9
generador = (x**2 for x in (1, 2, 3))             # elementos: 1 4 9
generador = (x+5 for x in (1, 2, 3))              # elementos: 6 7 8

generador = (x for x in range(10) if x % 2 == 0) # elementos: 0 2 4 6 8
generador = ((i, x) for (i, x) in enumerate((1, 2, 2)) if x == 2) # elementos: (1, 2) (2, 2)

generador = ((x, y) for x in (1, 2, 3) for y in (3, 1, 4) if x != y)
          # elementos: (1, 3) (1, 4) (2, 3) (2, 1) (2, 4) (3, 1) (3, 4)
```

```
for x in generador:
    print(x)
```

```
def buscar(lista: list, elem: Any) -> Any:
    return next( (x for x in lista if x == elem), None )
```

Desarrollo de generadores. Ejemplo Intervalo con expr. gen.

- Desarrollar una clase Intervalo, que permita iterar sobre los números enteros dentro de un intervalo especificado. El intervalo podrá especificar el valor inicial y final del intervalo, así como el paso de incremento. El valor final no forma parte de los elementos del intervalo. El paso podrá ser positivo o negativo.
  - **Nota:** implementaremos la iteración mediante un **generador** con *expresiones generadoras*.

Desarrollo de generadores. Ejemplo Intervalo con expr. gen.

```python
from typing import Optional, Iterator

class Intervalo:
    def __init__(self, inicial: int, final: Optional[int]=None, paso: Optional[int]=None,
                 *args, **kwargs) -> None:
        if (paso is None) or (paso == 0):
            paso = 1
        if final is None:
            final = inicial
            inicial = 0
        self.__inicial = inicial
        self.__final = final
        self.__paso = paso
        super().__init__(*args, **kwargs)

    def __iter__(self) -> Iterator[int]:
        # crea iterador con expresión generadora
        return (x for x in range(self.__inicial, self.__final, self.__paso))

def main() -> None:
    for x in Intervalo(4):          # 0 1 2 3
        print(x)
    for x in Intervalo(1, 4, 2):    # 1 3
        print(x)
    for x in Intervalo(4, 1, -1):   # 4 3 2
        print(x)

if __name__ == "__main__":
    main()
```

Desarrollo de generadores. Ejemplo Intervalo con expr. gen.

Versión alternativa de una función generadora con expr. generadoras

```
from typing import Optional, Iterator

def intervalo(inicial: int, final: Optional[int]=None, paso: Optional[int]=None) -> Iterator[int]:
    if (paso is None) or (paso == 0):
        paso = 1
    if final is None:
        final = inicial
        inicial = 0

    # crea iterador con expresión generadora
    return (x for x in range(inicial, final, paso))

def main() -> None:
    for x in intervalo(4):          # 0 1 2 3
        print(x)
    for x in intervalo(1, 4, 2):    # 1 3
        print(x)
    for x in intervalo(4, 1, -1):   # 4 3 2
        print(x)

if __name__ == "__main__":
    main()
```

Desarrollo de generadores. Ejemplo Alumnos con expr. gen.

- Desarrollar una clase Alumnos, que permita añadir notas de alumnos, donde cada alumno puede tener múltiples notas. Esta clase debe ser **iterable**, donde cada elemento de la iteración será una tupla con el nombre y la nota media del alumno.
  - **Nota:** implementaremos la iteración mediante un **generador** con *expresiones generadoras*.

Desarrollo de generadores. Ejemplo Alumnos con expr. gen.

```python
from typing import NamedTuple, Iterator
class Alumno(NamedTuple):
    nombre: str
    nota: float

class Alumnos:
    def __init__(self, *args, **kwargs) -> None:
        self.__alumnos: dict[str, list[float]] = dict() # clave: nombre; valor: lista-notas
        super().__init__(*args, **kwargs)

    def anyadir(self, nombre: str, nota: float) -> None:
        if nombre not in self.__alumnos:
            self.__alumnos[nombre] = list()
        self.__alumnos[nombre].append(nota)

    def __iter__(self) -> Iterator[Alumno]:
        # crea iterador con expresión generadora
        return (Alumno(nm, sum(nts)/len(nts)) for (nm, nts) in self.__alumnos.items())

def main() -> None:
    alumnos = Alumnos()
    alumnos.anyadir("pepe", 7.5)
    alumnos.anyadir("ana", 7.0)
    alumnos.anyadir("pepe", 8.5)
    alumnos.anyadir("ana", 8.0)
    for alumno in alumnos:
        print(alumno)                # Alumno(nombre='pepe', nota=8.0)
                                     # Alumno(nombre='ana', nota=7.5)
if __name__ == "__main__":
    main()
```

Desarrollo de generadores

Funciones generadoras con sentencias `yield`

- Las **funciones con sentencias** `yield` crean un **generador** (**iterador**) de la secuencia de elementos que se generan con la sentencia **yield**. Cuando termina la función, se lanza automáticamente la excepción `StopIteration`.

```
from typing import Iterator

def inverso(lista: list) -> Iterator:
    for i in range(len(lista)-1, -1, -1):
        yield lista[i]

for x in inverso([1, 2, 3]):
    print(x)                    # 3 2 1
```

```
try:
    it = iter(inverso([1, 2, 3]))
    while True:
        print(next(it))      # 3 2 1
except StopIteration as exc:
    pass
```

- Cuando se invoca a una función generadora, devuelve un generador, que es un iterador.
  - Cada invocación a **next** ejecuta parte del código de la función hasta ejecutar la siguiente sentencia **yield**, que genera el siguiente valor devuelto por **next**.
  - Así sucesivamente, hasta que se termina de ejecutar la función, cuando se lanzará automáticamente la excepción `StopIteration`

Desarrollo de generadores. Ejemplo Intervalo con yield

- Desarrollar una clase Intervalo, que permita iterar sobre los números enteros dentro de un intervalo especificado. El intervalo podrá especificar el valor inicial y final del intervalo, así como el paso de incremento. El valor final no forma parte de los elementos del intervalo. El paso podrá ser positivo o negativo.
  - **Nota:** implementaremos la iteración mediante un **generador** con la *sentencia yield*.

Desarrollo de generadores. Ejemplo Intervalo con yield

```python
from typing import Optional, Iterator

class Intervalo:
    def __init__(self, inicial: int, final: Optional[int]=None, paso: Optional[int]=None,
                 *args, **kwargs) -> None:
        if (paso is None) or (paso == 0):
            paso = 1
        if final is None:
            final = inicial
            inicial = 0
        self.__inicial = inicial
        self.__final = final
        self.__paso = paso
        super().__init__(*args, **kwargs)

    def __iter__(self) -> Iterator[int]:
        # crea iterador con sentencia yield
        for x in range(self.__inicial, self.__final, self.__paso):
            yield x

def main() -> None:
    for x in Intervalo(4):          # 0 1 2 3
        print(x)
    for x in Intervalo(1, 4, 2):    # 1 3
        print(x)
    for x in Intervalo(4, 1, -1):   # 4 3 2
        print(x)

if __name__ == "__main__":
    main()
```

Desarrollo de generadores. Ejemplo Intervalo con yield

Versión alternativa de una función generadora con yield

```
from typing import Optional, Iterator

def intervalo(inicial: int, final: Optional[int]=None, paso: Optional[int]=None) -> Iterator[int]:
    if (paso is None) or (paso == 0):
        paso = 1
    if final is None:
        final = inicial
        inicial = 0

    for x in range(inicial, final, paso):
        yield x

def main() -> None:
    for x in intervalo(4):          # 0 1 2 3
        print(x)
    for x in intervalo(1, 4, 2):    # 1 3
        print(x)
    for x in intervalo(4, 1, -1):   # 4 3 2
        print(x)

if __name__ == "__main__":
    main()
```

Desarrollo de generadores. Ejemplo Alumnos con yield

- Desarrollar una clase Alumnos, que permita añadir notas de alumnos, donde cada alumno puede tener múltiples notas. Esta clase debe ser **iterable**, donde cada elemento de la iteración será una tupla con el nombre y la nota media del alumno.
  - **Nota:** implementaremos la iteración mediante un **generador** con la *sentencia yield*.

Desarrollo de generadores. Ejemplo Alumnos con yield

```python
from typing import NamedTuple, Iterator
class Alumno(NamedTuple):
    nombre: str
    nota: float

class Alumnos:
    def __init__(self, *args, **kwargs) -> None:
        self.__alumnos: dict[str, list[float]] = dict() # clave: nombre; valor: lista-notas
        super().__init__(*args, **kwargs)

    def anyadir(self, nombre: str, nota: float) -> None:
        if nombre not in self.__alumnos:
            self.__alumnos[nombre] = list()
        self.__alumnos[nombre].append(nota)

    def __iter__(self) -> Iterator[Alumno]:
        # crea iterador con sentencia yield
        for (nm, nts) in self.__alumnos.items():
            yield Alumno(nm, sum(nts)/len(nts))

def main() -> None:
    alumnos = Alumnos()
    alumnos.anyadir("pepe", 7.5)
    alumnos.anyadir("ana", 7.0)
    alumnos.anyadir("pepe", 8.5)
    alumnos.anyadir("ana", 8.0)
    for alumno in alumnos:
        print(alumno)               # Alumno(nombre='pepe', nota=8.0)
                                    # Alumno(nombre='ana', nota=7.5)
if __name__ == "__main__":
    main()
```

Sentencia with y los gestores de contexto

Recursos

- Los **recursos** son objetos de nuestro código que necesitan una **operación de limpieza** depués de haber sido utilizados, y esa operación de limpieza se debe realizar de forma **obligatoria**, de tal forma que si no se realiza, se corre el riesgo de **agotar** la fuente de recursos.
  - Cuando se lanzan **excepciones**, se corre el riesgo de que parte del código no se ejecute, pudiendo suceder que la operación de limpieza de un determinado recurso no se llegue a realizar, **agotando** de esta forma la fuente de recursos.
  - La cláusula **finally** de la sentencia **try** puede ayudar a realizar una gestión de recursos adecuada.
  - No obstante, la sentencia **with** juanto a los **gestores de contexto** proporcionan un mecanismo más simple y seguro para realizar esta gestión de recursos.

Sentencia with y los gestores de contexto

- La sentencia **with** y los **gestores de contexto** han sido diseñados para facilitar la **gestión de recursos** en nuestro código.
- Cuando se utiliza la sentencia **with** para gestionar un recurso, cuando termina su ejecución (contexto), se **garantiza** que se **ejecutarán** las acciones de limpieza asociadas al recurso gestionado, independientemente de los errores y excepciones que hayan podido surgir en su procesamiento.
- La sentencia **with** utiliza los **gestores de contexto**, que definen los métodos `__enter__()` y `__exit__()`, para poder gestionar los recursos adecuadamente.

```
with gestor_de_contexto_del_recurso as manejador_del_recurso:
    #
    # utilización del recurso a través del manejador del recurso
    #
#
# fin de la sentencia with -> se ejecutan las operaciones de limpieza
# proporcionadas por el gestor de contexto, asociadas al recurso
#
```

Desarrollo de gestores de contexto

Sentencia with y los gestores de contexto

- Un gestor de contexto es una clase que debe definir los siguientes métodos:
  - El método `__init__(self, ..., *args, **kwargs)`: realiza las acciones necesarias para inicializar y asociar el recurso gestionado.
  - El método `__enter__(self)`: debe **preparar el recurso** para la entrada en la zona de contexto gestionado. Devuelve el objeto **manejador del recurso**, que se asigna a la variable especificada en la cláusula **as**.
  - El método `__exit__(self, exc_type, exc_value, exc_tb)`: se invoca y **ejecuta** cuando **termina** la ejecución de la zona de contexto gestionado, para que se puedan realizar las **acciones de limpieza** asociadas al recurso.
    - Si **no** se ha lanzado ninguna **excepción** durante el procesamiento de la zona de contexto gestionado, entonces los tres últimos parámetros tienen el valor `None`.
    - Si **sí** se ha lanzado alguna **excepción** durante el procesamiento de la zona de contexto gestionado, entonces el parámetro `exc_type` referencia al tipo de la excepción, el parámetro `exc_value` referencia al objeto excepción, y el parámetro `exc_tb` contiene la traza de ejecución.
    - Usualmente, este método devuelve `None` o `False`, que significa que cualquier excepción lanzada dentro del contexto debería continuar su **propagación**.
      No obstante, si este método devuelve `True`, entonces significa que cualquier **excepción** lanzada dentro del contexto se debería **bloquear** y **no** debería continuar su propagación.

Desarrollo de gestores de contexto

Gestor de contexto de recurso externo

- Desarrollar una clase, denominada `Frigorifico`, que permita almacenar productos. Además, dispone de un método para abrir la puerta, y un método para cerrar la puerta, considerando que es **obligatorio** cerrar la puerta después de almacenar los productos, para mantener las condiciones de temperatura adecuadas.
- Desarrollar una clase *gestora de contexto*, denominada `GestorFrigorifico`, que permita gestionar adecuadamente el recurso externo del frigorífico, garantizando que se cierre la puerta cuando se termine de añadir productos.

Desarrollo de gestores de contexto. Ejemplo Frigorífico-v1

```python
import logging

class Frigorifico:
    """Frigorífico que almacena productos. Se debe cerrar la puerta."""

    def __init__(self, capacidad: int, *args, **kwargs) -> None:
        self.__capacidad = capacidad
        self.__contenido: list[str] = list()
        self.__puerta_abierta = False
        super().__init__(*args, **kwargs)

    def abrir_puerta(self) -> None:
        self.__puerta_abierta = True
        logging.debug("Puerta abierta")

    def cerrar_puerta(self) -> None:
        self.__puerta_abierta = False
        logging.debug("Puerta cerrada")

    def almacenar(self, producto: str) -> None:
        if not self.__puerta_abierta:
            raise ValueError("frigorifico cerrado")
        if len(self.__contenido) >= self.__capacidad:
            raise ValueError("frigorifico lleno")
        self.__contenido.append(producto)
        logging.debug(f"Producto [{producto}] almacenado")

    def __repr__(self) -> str:
        return f"Contenido: {self.__contenido}; Puerta abierta: {self.__puerta_abierta}"
```

Desarrollo de gestores de contexto. Ejemplo Frigorífico-v1

```python
from typing import Any
class GestorFrigorifico:
    """Gestiona los recursos de un frigorífico. Cierra la puerta al finalizar"""

    def __init__(self, frigo: Frigorifico, *args, **kwargs) -> None:
        self.__frigo = frigo
        super().__init__(*args, **kwargs)

    def __enter__(self) -> Frigorifico:
        self.__frigo.abrir_puerta()
        return self.__frigo

    def __exit__(self,
                 exc_type: Any,                 # Optional[type]
                 exc_value: Any,                # Optional[BaseException]
                 exc_tb: Any) -> None:          # Optional[TracebackType]
        self.__frigo.cerrar_puerta()
        return None
```

Desarrollo de gestores de contexto. Ejemplo Frigorífico-v1

```python
def main() -> None:
    frigo = Frigorifico(capacidad=3)
    try:
        with GestorFrigorifico(frigo) as frg:
            frg.almacenar("yogur")
            frg.almacenar("helado")
            frg.almacenar("fruta")
            frg.almacenar("agua")
    except ValueError as exc:
        logging.exception(exc)
    else:
        print("productos almacenados")
    finally:
        print(frigo)

if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)
    main()
```

```
DEBUG:root:Puerta abierta
DEBUG:root:Producto [yogur] almacenado
DEBUG:root:Producto [helado] almacenado
DEBUG:root:Producto [fruta] almacenado
DEBUG:root:Puerta cerrada

ERROR:root:frigorifico lleno
Traceback (most recent call last):
  File "ctx_frigo_01.py", line 53, in main
    frg.almacenar("agua")
  File "ctx_frigo_01.py", line 24, in almacenar
    raise ValueError("frigorifico lleno")
ValueError: frigorifico lleno

Contenido: ['yogur', 'helado', 'fruta'];
Puerta abierta: False
```

Desarrollo de gestores de contexto

Gestor de contexto de recurso interno

- Desarrollar una clase, denominada `Frigorifico`, que permita almacenar productos. Además, dispone de un método para abrir la puerta, y un método para cerrar la puerta, considerando que es **obligatorio** cerrar la puerta después de almacenar los productos, para mantener las condiciones de temperatura adecuadas.
  Además, esta clase `Frigorifico` también va a ser una clase *gestora de contexto*, que permita gestionar adecuadamente sus recursos internos, garantizando que se cierre la puerta cuando se termine de añadir productos.

Desarrollo de gestores de contexto. Ejemplo Frigorífico-v2

```python
import logging

class Frigorifico:
    """Frigorífico que almacena productos. Se debe cerrar la puerta."""

    def __init__(self, capacidad: int, *args, **kwargs) -> None:
        self.__capacidad = capacidad
        self.__contenido: list[str] = list()
        self.__puerta_abierta = False
        super().__init__(*args, **kwargs)

    def abrir_puerta(self) -> None:
        self.__puerta_abierta = True
        logging.debug("Puerta abierta")

    def cerrar_puerta(self) -> None:
        self.__puerta_abierta = False
        logging.debug("Puerta cerrada")

    def almacenar(self, producto: str) -> None:
        if not self.__puerta_abierta:
            raise ValueError("frigorifico cerrado")
        if len(self.__contenido) >= self.__capacidad:
            raise ValueError("frigorifico lleno")
        self.__contenido.append(producto)
        logging.debug(f"Producto [{producto}] almacenado")

    def __repr__(self) -> str:
        return f"Contenido: {self.__contenido}; Puerta abierta: {self.__puerta_abierta}"
```

```python
def __enter__(self) -> 'Frigorifico':
    self.abrir_puerta()
    return self

def __exit__(self, exc_type: Any,
             exc_value: Any,
             exc_tb: Any) -> None:
    self.cerrar_puerta()
    return None
```

Desarrollo de gestores de contexto. Ejemplo Frigorífico-v2

```python
def main() -> None:
    frigo = Frigorifico(capacidad=3)
    try:
        with frigo as frg:
            frg.almacenar("yogur")
            frg.almacenar("helado")
            frg.almacenar("fruta")
            frg.almacenar("agua")
    except ValueError as exc:
        logging.exception(exc)
    else:
        print("productos almacenados")
    finally:
        print(frigo)

if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)
    main()
```

```
DEBUG:root:Puerta abierta
DEBUG:root:Producto [yogur] almacenado
DEBUG:root:Producto [helado] almacenado
DEBUG:root:Producto [fruta] almacenado
DEBUG:root:Puerta cerrada

ERROR:root:frigorifico lleno
Traceback (most recent call last):
  File "ctx_frigo_02.py", line 46, in main
    frg.almacenar("agua")
  File "ctx_frigo_02.py", line 24, in almacenar
    raise ValueError("frigorifico lleno")
ValueError: frigorifico lleno

Contenido: ['yogur', 'helado', 'fruta'];
Puerta abierta: False
```

Desarrollo de gestores de contexto. Equivalencia con with

La sentencia with

```
with gestor_de_contexto_del_recurso as manejador_del_recurso:
    #
    # utilización del recurso a través del manejador del recurso
    #
# fin de la sentencia with -> se ejecutan las operaciones de limpieza
# proporcionadas por el gestor de contexto, asociadas al recurso
```

Es aproximadamente equivalente a:

```
gestor = gestor_de_contexto_del_recurso
manejador_del_recurso = gestor.__enter__()  # Si __enter__ lanza excepción, entonces
hay_excepcion = False                       #  no se ejecuta try ni tampoco __exit__
try:
    # utilización del recurso a través del manejador del recurso
except Exception as exc:
    hay_excepcion = True
    if not gestor.__exit__(type(exc), exc, exc.__traceback__):
        raise exc
finally:
    if not hay_excepcion:
        gestor.__exit__(None, None, None)
```