**10.2 El protocolo de iteración**

En esta sección vemos lo que sucede realmente en Python durante una proceso de iteración. Esta sección tiene un [video](https://youtu.be/WyNiWuCnrhc) introductorio.

**Iteraciones por doquier**

Podemos iterar sobre una gran diversidad de objetos.

a = 'hola a todes'

for c in a : # Iterar sobre las letras en a

...

b = { 'nombre' : 'Elsa' , 'contraseña' : 'foo' }

para k en b : # Iterar sobre las claves de diccionario

...

c = [ 1 , 2 , 3 , 4 ]

for i in c : # Iterar sobre los items en una lista o tupla

...

f = open ( 'foo.txt' )

for x in f : # Iterar sobre las líneas de un archivo ASCII

...

Podemos iterar sobre todos estos objetos porque cumplen con un *protocolo* que permite, justamente, iterar. Veamos algo sobre este protocolo:

**El protocolo de iteración**

Analicemos la instrucción for.

para x en obj :

# instrucciones

¿Cómo funciona realmente ésto? Mediante un protocolo de iteración que resumirse así:

\_iter = obj . \_\_iter\_\_ () # Buscar el objeto iterador

while True :

try :

x = \_iter . \_\_next\_\_ () # Dame el siguiente item

excepto StopIteration : # No hay más items

break

# instrucciones ...

Todo objeto compatible con un ciclo forimplementado, a bajo nivel, este protocolo de iteración.

Un ejemplo: Iteración manual sobre una lista.

>> > x = [ 1 , 2 , 3 ]

>> > lo = x . \_\_iter\_\_ ()

>> > que

< ListIterator objeto en 0x590b0 >

>>> eso . \_\_next\_\_ ()

1

>> > que . \_\_next\_\_ ()

2

>> > que . \_\_next\_\_ ()

3

>> > que . \_\_next\_\_ ()

Rastreo ( más reciente llamada última ):

File "<stdin>" , línea 1 , en ? StopIteration

>> >

Fijate Que al agotar Los Elementos, el *rastreo* Acusa Una Excepción de tipo *StopIteration* . Fijate también que en la tercera línea, al preguntar por it, python responde <es un objeto de tipo listiterator (iterador de listas) alojado en 590b0 hexadecimal>.

**Iterable**

Es necesario que entiendas los mecanismos de iteradores si querés permitir iteración sobre objetos que vos definas, es decir, hacerlos *iterables* . El siguiente ejemplo construye un contenedor iterable, simplemente basado en una lista:

clase Camion :

def \_\_init\_\_ ( self ):

self . lotes = []

def \_\_iter\_\_ ( self ):

return self . lotes . \_\_iter\_\_ ()

...

camion = Camion ()

para c en camion :

...

**Ejercicios**

**Ejercicio 10.1: Iteradores, un ejemplo**

Construí la siguiente lista:

a = [ 1 , 9 , 4 , 25 , 16 ]

Y ahora iterá sobre esa lista *a mano* : Llamá al método \_\_iter\_\_()para obtener un objeto iterador y llama al método \_\_next\_\_()para obtener sucesivamente cada uno de los elementos.

>> > i = una . \_\_iter\_\_ ()

>> > i

< ListIterator objeto en 0x64c10 >

>>> yo . \_\_next\_\_ ()

1

>> > i . \_\_next\_\_ ()

9

>> > i . \_\_next\_\_ ()

4

>> > i . \_\_next\_\_ ()

25

>> > i . \_\_next\_\_ ()

16

>> > i . \_\_next\_\_ ()

Rastreo ( más reciente llamada última ):

File "<stdin>" , línea 1 ,en < módulo >

StopIteration

>> >

La función nativa de Python next()es un "atajo" al método \_\_next\_\_()de un iterador. Probá usarlo a mano sobre un archivo:

>> > f = abierto ( '../Data/camion.csv' )

>> > f . \_\_iter\_\_ () # Notar que esto apunta al método ...

# ... que acceda al archivo mismo.

< \_io . TextIOWrapper name = '../Data/camion.csv' mode = 'r' encoding = 'UTF-8' >

>>> Próxima ( f )

'nombre, cajones, precio \ n '

>> > próxima ( f )

'Lima, 100,32.20 \ n '

>> > próxima ( f )

'Naranja, 50,91.10 \ n '

>> >

Llamá a next(f)hasta que llegues al final del archivo, y fijate qué sucede.

**Ejercicio 10.2: Iteración sobre objetos**

Como decíamos, cuando definas tus propios objetos, es posible que quieras que se pueda iterar sobre ellos (especialmente si estos objetos son "envoltorios" (wrappers) para listas u otros iterables). Hagamos esto: en un nuevo archivo llamado camion.py, definí la siguiente clase:

# camion.py

clase Camion :

def \_\_init\_\_ ( self , lotes ):

self . lotes = lotes

def precio\_total ( self ):

devuelve la suma ([ l . costo () para l en self . lotes ])

def contar\_cajones ( self ):

de las colecciones import Counter

cantidad\_total = Counter ()

for l in self . lotes :

cantidad\_total [ l . nombre ] + = l . cajones

devuelven cantidad\_total

La intención es crear un envoltorio para una lista, y de paso agregarle algunos métodos, como la propiedad de calcular el costo total del camión. Vamos a usar lo que hiciste en el [Ejercicio 9.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/02_Clases.md#ejercicio-91-objetos-como-estructura-de-datos) . Modificá la función leer\_camion()en informe.pyde modo que cree una instancia de Camion, como se muestra:

# informe.py

...

importación fileparse

del lote de importación Lote

de camion importación Camion

def leer\_camion ( nombre de archivo ):

'' '

Lee un archivo con el contenido de un camión

y lo devuelve como un objeto Camion.

'' '

con abrir ( nombre de archivo ) como archivo :

camiondicts = fileparse . parse\_csv ( file ,

select = [ 'nombre' , 'cajones' , 'precio' ],

types = [ str , int, flotar ])

camion = [ Lote ( d [ 'nombre' ], d [ 'cajones' ], d [ 'precio' ]) for d in camiondicts ]

return Camion ( camion )

...

Ahora intentá correr el programa informe.py. Sin forma de heno. informe.pyintenta iterar sobre las instancias de Camionpero no son iterables y el programa no funciona.

>> > importación report

>> > report . informe\_camion ( '../Data/camion.csv' , '../Data/precios.csv' )

... muere ...

La forma de arreglar este programa roto es modificar la clase Camiony hacerla iterable.

clase Camion :

def \_\_init\_\_ ( self , lotes ):

self . lotes = lotes

def \_\_iter\_\_ ( self ):

return self . lotes . \_\_iter\_\_ ()

def precio\_total ( self ):

devuelve la suma ([ l . costo () para l en self . lotes ])

def contar\_cajones ( self ):

de las colecciones import Counter

cantidad\_total = Counter ()

para lote en self . lotes :

cantidad\_total [ lote . nombre ] + = lote . cajones

devuelven cantidad\_total

Después de haber hecho este cambio, tu informe.pydebería estar funcionando de nuevo. Guardá esta versión de informe.pypara entregar al final de la clase (en el próximo ejercicio te pediremos también camion.py).

Y ya que estás, cambiá el programa costo\_camion.pypara que use objetos que sean instancias de la clase Camion, por ejemplo así:

# costo\_camion.py

informe de importación

def costo\_camion ( nombre de archivo ):

'' '

Computa el precio total (cantidad \* precio) de un archivo camion

' ''

camion = informe . leer\_camion ( nombre de archivo )

return camion . precio\_total ()

...

Testealo, testealo, y testealo para asegurarte que funciona:

>> > importación costo\_camion

>> > costo\_camion . costo\_camion ( '../Data/camion.csv' )

47.671,15

>> >

**Ejercicio 10.3: Un iterador adecuado**

Cuando hagas clases que sean recipientes o contenedores de estructuras de datos vas a necesitar que hagan algo más que simplemente iterar. Probá modificar la clase Camionde modo que tenga algunos de los "métodos mágicos" que mencionamos en la [Sección 9.4](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/04_M%C3%A9todos_Especiales.md#m%C3%A9todos-especiales-para-convertir-a-strings) . Aquí hay algunos:

class Camion :

def \_\_init\_\_ ( self , lotes ):

self . lotes = lotes

def \_\_iter\_\_ ( self ):

return self . lotes . \_\_iter\_\_ ()

def \_\_len\_\_ ( self ):

return len ( self . lotes )

def \_\_getitem\_\_ ( self , index ):

return self . lotes [ índice ]

def \_\_contains\_\_ ( self , nombre ):

return any ([ lote . nombre == nombre for lote in self . lotes ])

def precio\_total ( self ):

devuelve la suma ([ l . costo () para l en self . lotes ])

def contar\_cajones ( self ):

de las colecciones import Counter

cantidad\_total = Counter ()

for l in self . lotes :

cantidad\_total [ l . nombre ] + = l . cajones

devuelven cantidad\_total

Por último, probemos esta nueva estructura:

>> > importación report

>> > camion = report . leer\_camion ( '../Data/camion.csv' )

>> > len ( camion )

7

>> > camion [ 0 ]

Lote ( 'Lima' , 100 , 32.2 )

>> > camion [ 1 ]

Lote ( 'Naranja' , 50 , 91.1 )

>> > 0 : 3 ]

[ Lote ( 'Lima' , 100 , 32.2 ),

Lote ( 'Naranja' , 50 , 91.1 ),

Lote ( 'Caqui' , 150 , 103.44 )]

>> > 'Naranja' en camion

Verdadero

>> > 'Manzana' en camion

Falso

>> >

Guardá tu versión de camion.pycon estos cambios para entregar y para la revisión de pares.

**Un comentario importante sobre todo esto:** Se considera un *buen estilo Python* al código que comparte ciertas normas de interacción con el resto del mundo Python. Este concepto aplicado a objetos contenedores significa que cumplen con las buenas costumbres de ser iterables, indexables y que admiten otras operaciones que naturalmente se espera *a priori* que vayan a cumplir, justamente por el simple hecho de ser objetos contenedores.

**10.3 Iteración a medida**

En esta sección introducimos el concepto de función generadora. Estas funciones te permiten obtener el iterador que necesites.

**Un problema de iteración**

Suponé que querés crear una secuencia particular de iteración: una cuenta regresiva, por decir algo.

>>> for x in regresiva(10):

... print(x, end=' ')

...

10 9 8 7 6 5 4 3 2 1

>>>

Existe una forma fácil de hacer esto.

**Generadores**

Un generador es una función que define un patrón de iteración.

def regresiva(n):

while n > 0:

yield n

n -= 1

*Nota: "yield" se traduce como "rendir" ó "entregar".*

Por ejemplo:

>>> for x in regresiva(10):

... print(x, end=' ')

...

10 9 8 7 6 5 4 3 2 1

>>>

Un generador es *cualquier* función que usa el commando yield.

El comportamiento de los generadores es algo diferente al del resto de las funciones.

Al llamar a un generador creás un objeto generador, pero su función no se ejecuta de inmediato.

def regresiva(n):

# Agreguemos este print para ver qué pasa...

print('Cuenta regresiva desde', n)

while n > 0:

yield n

n -= 1

>>> x = regresiva(10)

# No se ejecuta ningún PRINT !

>>> x

# sin embargo x es un objeto generador

<generator object at 0x58490>

>>>

La función sólo se ejecuta ante un llamado al método \_\_next\_\_()

>>> x = regresiva(10)

>>> x

<generator object at 0x58490>

>>> x.\_\_next\_\_()

Cuenta regresiva desde 10

10

>>>

Lo que hace yield es notable: produce un valor, y luego suspende la ejecución de la función. La ejecución continúa al volver a llamar a \_\_next\_\_().

>>> x.\_\_next\_\_()

9

>>> x.\_\_next\_\_()

8

Cuando finalmente se llega al final de la función, la iteración da un error.

>>> x.\_\_next\_\_()

1

>>> x.\_\_next\_\_()

Traceback (most recent call last):

File "<stdin>", line 1, in ? StopIteration

>>>

*Observación:* Una función generadora implementa el mismo protocolo de bajo nivel que los for usan sobre listas, tuplas, diccionarios, archivos, etc. ¡Por eso es tan sencillo usar los generadores para iterar!

**Ejercicios**

**Ejercicio 10.4: Un generador simple**

Si te encontrás con la necesidad de obtener una iteración particular, pensá en usar funciones generadoras. Son fáciles de escribir: simplemente hacé una función que implemente la lógica de iteración deseada y use yield para entregar valores.

Por ejemplo, probá este generador que busca un archivo y entrega las líneas que incluyen cierto substring.

>>> def filematch(filename, substr):

with open(filename, 'r') as f:

for line in f:

if substr in line:

yield line

>>> for line in open('../Data/camion.csv'):

print(line, end='')

nombre,cajones,precio

Lima,100,32.2

Naranja,50,91.1

Caqui,150,103.44

Mandarina,200,51.23

Durazno,95,40.37

Mandarina,50,65.1

Naranja,100,70.44

>>> for line in filematch('../Data/camion.csv', 'Naranja'):

print(line, end='')

Naranja,50,91.1

Naranja,100,70.44

>>>

Esta idea es muy interesante: podés armar una función que encapsule todo el procesamiento de datos y después recorrerla con un ciclo for para que te entregue los datos uno a uno.

El próximo ejemplo es de un caso aún más especial.

**Ejercicio 10.5: Monitoreo de datos en tiempo real.**

Acá hay un [breve video](https://youtu.be/NkfK8mcjuxo) sobre este ejercicio.

Un generador puede ser una forma interesante de vigilar datos a medida que son producidos. En esta sección vamos a probar esa idea. Para empezar, hacé lo siguiente.

Vas a necesitar dos archivos que están en tu carpeta de ejercicios, dentro de Data: sim\_mercado.py y mcentral.csv. Por favor primero borralos y bajate de nuevo [estos dos archivos](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/archivos_mercado.zip) porque los modificamos ligeramente. Guardá el archivo mcentral.csv en la carpeta Data y el archivo sim\_mercado.py en la carpeta de la clase 10. El programa sim\_mercado.py es un generador de datos de precios que toma como referencia a mcentral.csv. Al ejecutarlo, el programa escribe datos (con una componente aleatoria) en un archivo llamado mercadolog.csv contínuamente hasta que es detenido. Se ejecuta indefinidamente: una vez que inicies su ejecución podés dejarlo correr y olvidarte de él. Abrí una consola del sistema operativo nueva y ejecutá el programa. Si estás en Windows, dale un doble click al ícono de sim\_mercado.py, ó desde unix:

bash % python3 sim\_mercado.py

Después, olvidate de él. Dejálo ahí, corriendo.

Usando otra consola, mirá el contenido de Data/mercadolog.csv. Vas a ver que cada tanto se agrega una nueva línea al archivo.

Ahora que el programa generador de datos está en ejecución, escribamos un programa que abra el archivo, vaya al final, y espere nuevos datos. Para esto creá un programa llamado vigilante.py (es uno de los ejercicios a entregar) que contenga el siguiente código.

# vigilante.py

import os

import time

f = open('../Data/mercadolog.csv')

f.seek(0, os.SEEK\_END) # Mover el índice 0 posiciones desde el EOF

while True:

line = f.readline()

if line == '':

time.sleep(0.5) # Esperar un rato y

continue # vuelve al comienzo del while

fields = line.split(',')

nombre = fields[0].strip('"')

precio = float(fields[1])

volumen = int(fields[2])

if volumen > 1000:

print(f'{nombre:>10s} {precio:>10.2f} {volumen:>10d}')

*Nota: EOF = End Of File (fin de archivo)*

Cuando ejecutes el programa vas a ver un indicador de precios en el mercado en tiempo real, con indicación de qué producto se trata, cuál es su precio, y cuál es el volumen de la operación (en cantidad de cajones).

*Observación:* La forma en que usamos el método readline() en este ejemplo es un poco rara, no es la forma en que se suele usar (dentro de un ciclo for para recorrer el contenido de un archivo). En este caso la estamos usando para mirar constantemente el fin de archivo para obtener los últimos datos que se hayan agregado (readline() devuelve ó bien el último dato o bien una cadena vacía).

**Ejercicio 10.6: Uso de un generador para producir datos**

Si analizás un poco el código del [Ejercicio 10.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/03_iteracion_a_medida.md#ejercicio-105-monitoreo-de-datos-en-tiempo-real) vas a notar que la primera parte del programa "produce" datos (los obtiene del archivo) y la segunda los procesa y los imprime, es decir "consume" datos. Una característica importante de las funciones generadoras es que podés mover todo el código a una función reutilizable. Probalo vos.

Modificá el código del [Ejercicio 10.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/03_iteracion_a_medida.md#ejercicio-105-monitoreo-de-datos-en-tiempo-real) de modo que la lectura del archivo esté resuelta por una única función generadora vigilar() a la que se le pasa un nombre de archivo como parámetro. Hacelo de modo que el siguiente código funcione:

>>> for line in vigilar('../Data/mercadolog.csv'):

print(line)

# Acá deberías ver las líneas impresas sin formato de tabla

# Algo similar a esto

"Berenjena", 49.33, 624

"Naranja", 15.88, 1040

"Cebolla", 66.32, 720

"Durazno", 149.02, 768

"Pomelo", 48.24, 981

"Lechuga", 32.57, 819

"Repollo", 6.69, 777

"Lima", 1272.49, 208

"TomCherry", 514.05, 208

Modificá el programa vigilante.py para que termine con un bloque así:

if \_\_name\_\_ == '\_\_main\_\_':

for line in vigilar('../Data/mercadolog.csv'):

fields = line.split(',')

nombre = fields[0].strip('"')

precio = float(fields[1])

volumen = int(fields[2])

if volumen > 1000:

print(f'{nombre:>10s} {precio:>10.2f} {volumen:>10d}')

Guradá esta versión de vigilante.py para entregar al final de la clase.

**Ejercicio 10.7: Cambios de precio de un camión**

Modificá el programa vigilante.py para que sólo informe las líneas que tienen precios de lotes incluídos en un camión, e ignore el resto de los productos. Por ejemplo:

if \_\_name\_\_ == '\_\_main\_\_':

import informe

camion = informe.leer\_camion ('../Data/camion.csv')

for line in vigilar('../Data/mercadolog.csv'):

fields = line.split(',')

nombre = fields[0].strip('"')

precio = float(fields[1])

volumen = int(fields[2])

if nombre in camion:

print(f'{nombre:>10s} {precio:>10.2f} {volumen:>10d}')

*Observación:* para que esto funcione, tu clase Camion tiene que haber implementado el operador in. Controlá que tu solución al ejercicio [Ejercicio 10.3](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/02_protocolo_Iteracion.md" \l "ejercicio-103-un-iterador-adecuado) implemente el operador \_\_contains\_\_().

**Discusión**

Lo que acaba de suceder es algo con mucho potencial: moviste tu patrón de iteración (el que toma las últimas líneas de un archivo) y lo pusiste en su propia función. La función vigilar() ahora es una función de uso amplio, que podés usar en cualquier programa. Por ejemplo, la podrías usar para mirar el log (historial) de un servidor ó de un debugger, o de otras fuentes continuas de datos. ¿No está bueno?

**10.4 Productores, consumidores y cañerías.**

Para esta sección hay un [video introductorio](https://youtu.be/CSFc1Z__1hA).

Los generadores son una herramienta muy útil para configurar pipelines (cañerías). Este concepto requiere una breve *aclaración*: Un pipeline tradicional en computación consta de una serie de programas y archivos asociados que constituyen una estructura de procesamiento de datos, donde cada programa se ejecuta independientemente de los demás, pero juntos resultan en un flujo conveniente de datos a través de los archivos asociados desde un "productor" (una cámara, un sensor, un lector de código de barras) hasta un "consumidor" (un graficador, un interruptor eléctrico, un log de una página web). Construiste un pequeño pipeline en la sección anterior, usando vigilante.py.

En esta sección hablaremos de cómo implementar estas estructuras de productores y consumidores de datos con generadores en Python.

**Sistemas productor-consumidor**

El concepto de generadores está íntimamente asociado a problemas de tipo productor-consumidor en sus varias formas. Fijate esta estructura, que es típica de muchos programas:

# Productor

def vigilar(f):

...

while True:

...

yield linea # Produce/obtiene valores para "linea"

...

# Consumidor

for linea in vigilar(f): # Consume líneas del `yield`

...

Los yield generan los datos que los for consumen.

**Pipelines con generadores**

Podés usar esta característica de los generadores para construír *pipelines* que procesen tus datos, un concepto que es muy usado en Unix (pipes) pero en Windows se usa menos.

*productor* → *procesamiento* → *procesamiento* → *consumidor*

Los *pipelines* de procesamiento de datos tienen un productor al comienzo, una cadena de etapas de procesamiento y un consumidor al final.

**productor** → *procesamiento* → *procesamiento* → *consumidor*

def productor():

...

yield item

...

El productor es en general un generador, aunque tambien podría ser una lista o cualquier otra secuencia iterable.

El yield alimenta al pipeline de datos.

*productor* → *procesamiento* → *procesamiento* → **consumidor**

def consumidor(s):

for item in s:

...

El consumidor es un ciclo for. Obtiene los elementos item y los usa para algo.

*productor* → **procesamiento** → **procesamiento** → *consumidor*

def procesamiento(s):

for item in s:

...

yield itemnuevo

...

Las etapas intermedias de procesamiento simultáneamente consumen y producen datos, pueden alterar el flujo de datos, eliminar o modificar datos según su función.

*productor* → *procesamiento* → *procesamiento* → *consumidor*

def productor():

...

yield item # yield devuelve un item que será recibido por `procesamiento`

...

def procesamiento(s):

for item in s: # item viene del `productor`

...

yield newitem # este yield devuelve un nuevo item

...

def consumidor(s):

for item in s: # item viene de `procesamiento`

...

Vamos a construír un pipeline con la siguiente arquitectura:

a = productor()

b = procesamiento(a)

c = consumidor(b)

Como te darás cuenta, los datos van pasando de una función a la siguiente.

**Ejercicios**

Para este ejercicio, necesitás que el programa sim\_mercado.py aún esté corriendo. Vas a usar la función vigilar() que escribiste en el [Ejercicio 10.7](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/03_iteracion_a_medida.md#ejercicio-107-cambios-de-precio-de-un-cami%C3%B3n)

**Ejercicio 10.8: Configuremos un pipeline simple**

Escribí la siguiente función y veamos como funciona un pipeline.

>>> def filematch(lines, substr):

for line in lines:

if substr in line:

yield line

>>>

Esta función es casi idéntica al primer ejemplo de generador en el ejercicio anterior, salvo que ya no abre un archivo sino que opera directamente de una secuencia de líneas que recibe como argumento. Ahora probá lo siguiente:

>>> lines = vigilar('../Data/mercadolog.csv')

>>> naranjas = filematch(lines, 'Naranja')

>>> for line in naranjas:

print(line)

... esperá que aparezca la salida ...

Puede pasar que tarde unos segundos en darte una salida, pero vas a ver información sobre naranjas tan pronto como sean añadidas al archivo por el primer generador.

**Ejercicio 10.9: Un pipeline más en serio**

Llevemos esta idea un poco más lejos. Probemos esto:

>>> from vigilante import vigilar

>>> import csv

>>> lineas = vigilar('../Data/mercadolog.csv')

>>> filas = csv.reader(lineas)

>>> for fila in filas:

print(fila)

...

['Rabanito', ' 249.37', ' 357']

['Batata', ' 15.75', ' 1040']

['Rabanito', ' 211.31', ' 324']

['Zuccini', ' 83.12', ' 612']

...

¡Interesante! La salida de la función vigilar() fué usada como entrada a la función csv.reader() (que habíamos usado para leer un archivo del disco) y el resultado es una secuencia de filas "parseadas", separadas por las comas.

**Ejercicio 10.10: Un pipeline más largo**

Veamos si podemos construír un pipeline más largo basado en la misma idea.

Creá un archivo nuevo llamado ticker.py, te lo vamos a pedir al final de la clase. Comenzá creando una función que lea un archivo CSV como hiciste antes:

# ticker.py

from vigilante import vigilar

import csv

def parsear\_datos(lines):

rows = csv.reader(lines)

return rows

if \_\_name\_\_ == '\_\_main\_\_':

lines = vigilar('../Data/mercadolog.csv')

rows = parsear\_datos(lines)

for row in rows:

print(row)

Escribí una función nueva que elija algunas columnas específicas:

# ticker.py

...

def elegir\_columnas(rows, indices):

for row in rows:

yield [row[index] for index in indices]

...

def parsear\_datos(lines):

rows = csv.reader(lines)

rows = elegir\_columnas(rows, [0, 2])

return rows

Probalo de nuevo. La salida ahora debería estar restringida a esto:

['Brócoli', ' 388']

['Ajo', ' 120']

['Caqui', ' 108']

['Mandarina', ' 1170']

...

Escribí funciones generadoras que conviertan el tipo de datos a diccionarios:

# ticker.py

...

def cambiar\_tipo(rows, types):

for row in rows:

yield [func(val) for func, val in zip(types, row)]

def hace\_dicts(rows, headers):

for row in rows:

yield dict(zip(headers, row))

def parsear\_datos(lines):

rows = csv.reader(lines)

rows = elegir\_columnas(rows, [0, 1, 2])

rows = cambiar\_tipo(rows, [str, float, int])

rows = hace\_dicts(rows, ['nombre', 'precio', 'volumen'])

return rows

...

Correlo de nuevo. Ahora la salida debería ser una serie de diccionarios:

{'nombre': 'Frutilla', 'precio': 276.81, 'volumen': 249.0}

{'nombre': 'Morrón', 'precio': 2988.42, 'volumen': 702.0}

{'nombre': 'Morrón', 'precio': 3108.63, 'volumen': 498.0}

{'nombre': 'Naranja', 'precio': 11.5, 'volumen': 870.0}

...

**Ejercicio 10.11: Filtremos los datos**

Para seguir agregando procesamiento a nuestro pipeline, escribí un filtro de datos:

# ticker.py

...

def filtrar\_datos(filas, nombres):

for fila in filas:

if fila['nombre'] in nombres:

yield fila

Esto se usa para dejar pasar únicamente aquellos lotes incluidos en el camión. Probalo.

import informe

camion = informe.leer\_camion('../Data/camion.csv')

filas = parsear\_datos(vigilar('../Data/mercadolog.csv'))

filas = filtrar\_datos(filas, camion)

for fila in filas:

print(fila)

**Ejercicio 10.12: El pipeline ensamblado**

En el programa ticker.py (esta versión te vamos a pedir que la entregues) escribí una función ticker(camion\_file, log\_file, fmt) que cree un indicador en tiempo real para un camión, archivo log, y formato de tabla de salida particulares. Fijate:

>>> from ticker import ticker

>>> ticker('../Data/camion.csv', '../Data/mercadolog.csv', 'txt')

Nombre Precio Volumen

---------- ---------- ----------

Caqui 796.73 96

Mandarina 12.12 1120

Lima 2659.37 222

Naranja 11.70 1040

Durazno 281.76 704

...

>>> ticker('../Data/camion.csv', '../Data/mercadolog.csv', 'csv')

Nombre,Precio,Volumen

Mandarina,14.19,1140

Naranja,9.37,1150

Durazno,280.56,872

Lima,2624.94,232

...

**Discusión**

¿Qué aprendimos hoy? Si creás varias funciones generadoras y las ponés "en serie" (una recibe los datos de la anterior) podés crear pipelines que controlen el flujo de datos: los procesen, modifiquen o filtren entre el primer generador y el último consumidor. Además viste lo fácil que es cambiar el comportamiento del programa cuando tenés interfases bien definidas. Por supuesto, podés empaquetar un conjunto de etapas de procesamiento en una función sola, si tiene sentido hacerlo.

**10.5 Más sobre generadores**

Esta sección introduce algunos temas adicionales relacionados con generadores, entre ellas: expresiones generadoras y el módulo itertools

**Expresiones generadoras**

Una expresión generadora es una lista por comprensión en su "versión generadora", que devuelve un elemento por vez.

>>> a = [1,2,3,4]

>>> b = (2\*x for x in a)

>>> b

<generator object at 0x58760>

>>> for i in b:

... print(i, end=' ')

...

2 4 6 8

>>>

¿Cuales son las diferencias entre expresiones generadoras y comprensión de listas? Bueno, las expresiones generadoras ...

* No construyen una lista
* Son construidas para ser iteradas
* Una vez consumidas, no pueden ser reutilizadas.

La sintaxis general es:

(<expression> for i in s if <conditional>)

Que puede leerse como .... el valor es para cada elemento i perteneciente a s siempre y cuando se cumpla.

Las podés usar como argumento de una función.

sum(x\*x for x in a)

Las podés usar en lugar de cualquier iterable.

>>> a = [1,2,3,4]

>>> b = (x\*x for x in a)

>>> c = (-x for x in b)

>>> for i in c:

... print(i, end=' ')

...

-1 -4 -9 -16

>>>

El uso principal de las expresiones generadoras es en código que realiza un cómputo con una serie de elementos pero sólo necesita cada elemento una única vez. Ejemplo: quitar todas las líneas de un programa que sean comentarios:

f = open('unarchivo.txt')

lines = (line for line in f if not line.startswith('#'))

for line in lines:

...

f.close()

Al usar generadores, tu código ejecuta más rápido y usa menos memoria. Se portan como un filtro en el flujo de datos por un pipeline.

**Motivos para usar generadores**

* Muchos problemas se expresan mejor en términos de iteración.
  + Recorrer una colección de items para hacer algún cómputo (buscar, reemplazar, modificar, etc.).
  + Los pipelines de procesamiento resuelven un amplio abanico de problemas.
* Son más eficientes en el uso de memoria.
  + Sólo producís valores cuando los necesitás.
  + Varias ventajas sobre construir una larga lista.
  + Pueden operar sobre datos en pipelines.
* Un generador facilita la reutilización de código.
  + Separa la propia *iteración* del código que utiliza sus resultados.
  + Podés construir tu propio conjunto de herramientas de iteración y ensamblarlas como necesites en cada caso.

**El módulo itertools**

El módulo itertools es una biblioteca con varias funciones útiles para construir generadores e iteradores.

itertools.chain(s1,s2)

itertools.count(n)

itertools.cycle(s)

itertools.dropwhile(predicate, s)

itertools.groupby(s)

itertools.ifilter(predicate, s)

itertools.imap(function, s1, ... sN)

itertools.repeat(s, n)

itertools.tee(s, ncopies)

itertools.izip(s1, ... , sN)

Todas estas funciones procesan datos iterativamente, e implementan distintos tipos de patrones de iteración.

Si querés profundizar más en estos conceptos, te recomendamos el curso que escribió Beazley hace unos años sobre [Generadores e iteradores](http://www.dabeaz.com/generators/).

**Ejercicios**

En ejercicios anteriores escribiste código que vigilaba un archivo log esperando líneas nuevas escritas al final y las presentaba como una secuencia de filas. Este ejercicio continua aquel, de modo que vas a necesitar que sim\_mercado.py esté ejecutándose. Acordate de pararlo cuando termines... agrega una línea por segundo y eventualmente te va a llenar tu disco rígido.

**Ejercicio 10.13: Expresiones generadoras**

Fijate este ejemplo de una expresión generadora:

>>> nums = [1, 2, 3, 4, 5]

>>> cuadrados = (x\*x for x in nums)

>>> cuadrados

<generator object <genexpr> at 0x109207e60>

>>> for n in cuadrados:

... print(n)

...

1

4

9

16

25

A diferencia de una definición por comprensión de listas, una expresión generadora sólo puede recorrerse una vez. Si intentás recorrerla de nuevo con otro for, no obtenés nada.

>>> for n in squares:

... print(n)

...

>>>

**Ejercicio 10.14: Expresiones generadoras como argumentos en funciones.**

A veces es útil (y muy claro al leerlo) si pasás una expresión generadora como argumento de una función. A primera vista parece un poco raro, pero probá esto:

>>> nums = [1,2,3,4,5]

>>> sum([x\*x for x in nums]) # una lista por comprensión

55

>>> sum(x\*x for x in nums) # una expresión generadora

55

>>>

En ese ejemplo, la segunda versión (que usa un generador) requeriría mucha menos memoria que si construyera toda la lista simultáneamente (si la lista fuera grande).

En tu archivo camion.py hiciste algunos cálculos usando comprensión de listas. Reemplazá esas expresiones por expresiones generadoras (podés entregar esta nueva versión del archivo o la anterior al final de la clase).

**Ejercicio 10.15: Código simple**

Las expresiones generadoras son a menudo un buen reemplazo para pequeñas funciones generadoras. Por ejemplo, en lugar de escribir una función como esta:

def filtrar\_datos(filas, nombres):

for fila in filas:

if fila['nombre'] in nombres:

yield fila

La podrías reemplazar con una expresión así:

filas = (fila for fila in filas if fila['nombre'] in nombres)

Modificá tu programa ticker.py para que use expresiones generadoras en lugar de funciones generadoras. Al final de la clase podés entregar el ticker.py anterior o este nuevo (¡mejor el nuevo!).

**Ejercicio 10.16: Volviendo a ordenar imágenes**

Te proponemos aquí que retomes el [Ejercicio 8.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/08_Fechas_Carpetas_y_Pandas/04_Ordenando_archivos.md#ejercicio-85-recorrer-el-%C3%A1rbol-de-archivos) que tenés guardado en el archivo listar\_imgs.py. Usá los datos que te proporciona os.walk y una expresión generadora para filtrar las imágenes png (con sus directorios correspondientes). Este filtro debería generar pares (directorio, archivo.png)

Más aún, opcionalmente diseñá un generador que, dada la secuencia filtrada (directorios y archivos png), genere ternas consistentes de: ('viejo\_dir/viejo\_nombre', 'nuevo\_dir/nuevo\_nombre', fecha\_a\_setear) de manera que pueda ser fácilmente usada por una función para completar las tareas del [Ejercicio 8.6](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/08_Fechas_Carpetas_y_Pandas/04_Ordenando_archivos.md#ejercicio-86-ordenar-el-%C3%A1rbol-de-archivos-optativo).

**10.6 Predador Presa**

En este [video](https://youtu.be/mGBxvPtMkW4) introducimos el tema de esta sección.

Los [modelos de depredación y competencia](https://es.wikipedia.org/wiki/Ecuaciones_Lotka%E2%80%93Volterra) forman parte de la batería de herramientas clásicas del ecólogo. Vito Volterra en Italia y Alfred Lotka en Estados Unidos fueron los precursores en este tema y crearon modelos que, con diversas modificaciones y mejoras, seguimos usando hoy en día. El modelo de Volterra para depredación comienza suponiendo la existencia de dos poblaciones de animales, una de las cuales (el depredador) se alimenta de la otra (la presa). Se supone que las dos poblaciones están formadas por individuos idénticos, mezclados en el espacio.

La gran mayoría de los modelos usualmente estudiados tienen solamente en cuenta la dinámica temporal, desatendiendo la dinámica espacial. Con la excusa de implementar una versión espacio-temporal de este modelo, proponemos un ejercicio guiado que usa fuertemente programación orientada a objetos.

Lo que sigue es un ejercicio optativo. Como decíamos, el ejercicio está muy guiado y toda la estructura del modelo está basada en objetos. La idea es recrear un mundo que imaginaremos como un valle rodeado de montañas en el que existen depredadores (que llamaremos Leones) y presas (que llamaremos Antílopes). Este valle será bidimensional, y lo representaremos por medio de una grilla rectangular que llamaremos tablero.

El modelo inicial con el que trabajaremos tiene definidas 4 etapas que determinan un ciclo:

* **Etapa de movimiento**: en esta etapa cada animal se desplaza a alguna posición vecina desocupada (si es que existe, sino permanece en el lugar). La decisión de a cuál desplazarse será responsabilidad del animal, que, sabiendo las disponibles, elegirá una al azar.
* **Etapa de alimentación**: en esta etapa los animales se alimentan. Los antílopes comerán pasto en su lugar, mientras que los leones buscarán un antílope en las posiciones vecinas y, de existir, se lo comerán. Esta acción de un león se verá reflejada en el tablero con su desplazamiento a la posición del antílope, el cual ya no será más parte de nuestro mundo.
* **Etapa de reproducción**: cada animal buscará en sus posiciones vecinas alguien de su misma especie. Si lo encuentra y además hay una posición vacía en el tablero se incluirá un nuevo animal (de la misma especie) en el tablero. Nuevamente la elección de la pareja y del lugar del nuevo animal serán aleatorias. Este modelo inicial no prevee el atributo sexo, ni un límite entre la cantidad reproducciones en las que puede participar un animal por etapa.
* **Cierre de ciclo**: en la última etapa de un ciclo todos los animales "envejecen" en 1 unidad, aquellos que se reprodujeron y siguen en edad reproductiva vuelven a ser fértiles. Si alguno alcanzó la edad máxima de su especie se considera que ya puede retirárselo del mundo (es decir, se muere). Sucede lo mismo si, al pasar una etapa, alcanza el límite de etapas sin alimentarse, en cuyo caso muere de hambre.

**Clases del modelo inicial:**

El modelo inicial te lo podés bajar de [acá](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/modelo.zip). Ya tiene definidas la clase Animal de la que heredan dos nuevas clases: León y Antílope. También tiene definida una clase Tablero. Todas estas clases se integran en una clase Mundo. Vamos a ver una por una estas clases y te indicaremos algunos métodos faltantes que tendrás que implementar vos y otros que te proponemos mejorar.

**Animales**

Esta clase está definida en el archivo animal.py.

La clase Animal será una clase abstracta, de la cual heredarán la clase León y la clase Antílope. Los métodos y atributos comunes que representen a un animal deberán estar en esta clase.

**Constructor**

El constructor no recibe parámetros, sólo instancia los atributos iniciales. En esta documentación se instancian algunos.

def \_\_init\_\_(self):

super(Animal, self).\_\_init\_\_()

self.edad = 0

self.energia = self.energia\_maxima # cantidad de ciclos que aguanta sin alimentarse

**Consultas**

A continuación se muestran los métodos para obtener información del animal. Es importante notar que un animal en sí, no es ni León ni Antílope, por eso ambas preguntas devuelven False.

Estos métodos definen a su vez el comportamiento, no siendo unicamente una observación de las variables, sino que definen que un animal está vivo únicamente si no alcanzó la edad máxima y además no pasó el límite de tiempo que puede estar sin alimentarse.

Lo mismo sucede con tiene\_hambre que podría modificarse para que un animal que ya se alimentó no necesite alimentarse instantáneamente. Se pueden agregar métodos como puede\_tener\_cria y cualquier otra consulta que sea pertinente para un animal.

def en\_vida(self):

return (self.edad <= self.edad\_maxima) and self.energia > 0

def tiene\_hambre(self):

"""Acá se puede poner comportamiento para que no tenga hambre todo el tiempo

debería depender de la diferencia entre su nivel de energía y su energía máxima"""

return True

def es\_leon(self):

return False

def es\_antilope(self):

return False

**Modificadores**

A continuación presentamos algunos métodos modificadores del objeto.

def pasar\_un\_ciclo(self):

self.energia -= 1 # consume energía

self.edad += 1 # envejece

if self.edad >= 2 and self.reproducciones\_pendientes > 0:

self.es\_reproductore = True #se puede reproducir

def tener\_cria(self):

"""Acá se puede poner comportamiento que sucede al tener cria

y que evita que tengas más de una cria por ciclo, ¿quizas tener\_cria consume más energía que un ciclo común?"""

pass

def alimentarse(self, animales\_vecinos = None):

self.energia = self.energia\_maxima

return None

def reproducirse(self, vecinos, lugares\_libres):

pos = None

if vecinos:

pareja = random.choice(vecinos)

if lugares\_libres:

self.tener\_cria()

pareja.tener\_cria()

pos = random.choice(lugares\_libres)

return pos

def moverse(self, lugares\_libres):

pos = None

if lugares\_libres:

pos = random.choice(lugares\_libres)

return pos

**León**

Al heredar de Animal, la clase Leon es pequeña:

class Leon(Animal):

Todas los métodos y atributos definidos allí están disponibles. Se hace un *override* del metodo es\_leon para aclarar que en este caso SÍ es un león. Y como el león es carnívoro, también se extiende el método alimentarse que sólo sucede cuando puede comer.

def \_\_init\_\_(self):

super(Leon, self).\_\_init\_\_()

self.autonomia = 3

self.edad\_maxima = 5

def es\_leon(self):

return True

def alimentarse(self, animales\_vecinos):

# Se alimenta si puede e indica la posición del animal que se pudo comer

pos = None

if self.tiene\_hambre(): # no está lleno

presas\_cercanas = [ (pos,animal) for (pos, animal) in animales\_vecinos if animal.es\_antilope() ]

if presas\_cercanas: # y hay presas cerca

super(Leon, self).alimentarse()

(pos, animal) = random.choice(presas\_cercanas)

return pos

**Ejemplo: Un León**

Probá las siguientes instrucciones:

L = Leon()

L.energia

L.edad

L.es\_leon()

L.es\_antilope()

L.pasar\_un\_ciclo()

L.energia

L.edad

L.tiene\_hambre()

**Antílope**

La clase Antilope es más pequeña aún y unicamente indica que es un Antílope. Acá se asume fuertemente que el antílope come pasto y que pasto hay siempre.

def \_\_init\_\_(self):

self.autonomia = 8

self.edad\_maxima = 8

super(Antilope, self).\_\_init\_\_()

def es\_antilope(self):

return True

**Ejemplo: Un León y dos Antílopes**

Agregá ahora dos antílopes:

A1 = Antilope()

A2 = Antilope()

A1.energia

A1.edad

A1.es\_antilope()

Y hacé que el León, que ya tiene hambre, se coma alguno de los dos antílopes (aleatoriamente):

vecinos = [(1,A1),(2,A2)]

pos = L.alimentarse(vecinos)

if pos:

print(f'El león se come al antílope A{pos}')

else:

print(f'El león no come')

Fijate que el método alimentarse no mata al antílope, solo devuelve su posición. *Alguien más* deberá ocuparse de retirarlo del mundo.

Si corrés este código varias veces, vas a ver que el León obtiene energía máxima al comer, pero sigue comiendo. Implementá adecuadamente el método tiene\_hambre() de la clase Animal de manera que solo tenga hambre cuando su energía baje. Volvé a probar el código. El león debería comer solo cuando pase al menos un ciclo desde la última vez que comió.

**Ejemplo: Un León y una Leona**

Definamos ahora una leona M y hagamos que pasen unos ciclos hasta que sea reproductora.

M = Leon()

M.puede\_reproducir()

M.pasar\_un\_ciclo()

M.puede\_reproducir()

Repetí hasta que L y M sean reproductores.

Luego ponelos cerca. Digamos que M tiene como vecinos a los dos antílopes y al león L de antes. Supongamos que las crias pueden ocupar algunos lugares libres en nuestro mundo (aún no definido):

vecinos = [L]

lugares\_libres = [4,5,6,7,8]

L.puede\_reproducir()

M.puede\_reproducir()

Están dadas las condiciones, que se reproduzcan:

pos = M.reproducirse(vecinos, lugares\_libres)

print(f'nace un nuevo leoncito en la posición {pos}')

M.puede\_reproducir()

M.pasar\_un\_ciclo()

M.puede\_reproducir()

Vemos que el método reproducirse nos devuelve una de las posiciones que le dijimos que estaban libres y que los leones no pueden volverse a reproducir en el mismo ciclo pero sí en el siguiente.

Pasemos ahora a ver cómo se estructura el tablero.

**El tablero**

**Constructor**

La clase tablero tendrá un único constructor que recibirá los atributos que darán lugar al tablero del tamaño deseado:

def \_\_init\_\_(self, filas, columnas):

super(Tablero, self).\_\_init\_\_()

self.filas = filas

self.columnas = columnas

self.posiciones = {}

self.n\_posiciones\_libres = self.filas \* self.columnas

Esta clase tendrá 4 atributos: el número de filas, el de columnas, un diccionario donde se almacenará el contenido, y la cantidad de celdas vacías. El diccionario tendrá unicamente las posiciones con algún contenido. La cantidad de celdas vacías es un atributo que podría calcularse a partir de los 3 primeros atributos, pero se decidió tenerlo precalculado por ser un valor que se usará mucho.

**Modificadores**

El tablero no tiene una gran complejidad, permite ser modificado de 2 maneras: ubicar un elemento en el tablero, y retirar un elemento del tablero. Se provee además de la funcionalidad que combina estas cosas bajo el nombre de mover (cambia de lugar un elemento).

def ubicar(self, pos, elem):

if not self.ocupada(pos):

self.n\_posiciones\_libres -= 1

self.posiciones[pos] = elem

return pos in self.posiciones

def retirar(self, pos):

self.n\_posiciones\_libres += 1

return self.posiciones.pop(pos)

def mover(self, p\_origen, p\_destino):

self.ubicar(p\_destino, self.retirar(p\_origen))

**Getters**

Además la clase tablero provee métodos para consultar sobre estado de posiciones y los valores que se encuentran en el tablero.

def posicion(self, pos):

#devuele qué hay en pos

return self.posiciones[pos]

def ocupada(self, pos):

return pos in self.posiciones

def libre(self, pos):

return pos not in self.posiciones

def elementos(self):

return list(self.posiciones.values())

def hay\_posiciones\_libres(self):

return self.n\_posiciones\_libres > 0

**Modificadores complejos**

Una de las funcionalidades deseables para que se pueda usar un tablero, es la posibilidad de ubicar al azar un elemento en algún lugar del mismo (que esté libre). El tablero provee esa funcionalidad:

def ubicar\_en\_posicion\_vacia(self, elem):

if not self.hay\_posiciones\_libres():

raise RuntimeError("Estás intentado agregar algo al tablero y está lleno")

pos = choice(self.posiciones\_libres())

self.ubicar(pos, elem)

**Consultas**

También ofrece consultas más complejas:

def posiciones\_ocupadas(self):

res = []

for f in range(self.filas):

for c in range(self.columnas):

if self.ocupada((f,c)):

res.append((f,c))

return res

def posiciones\_libres(self):

res = []

for f in range(self.filas):

for c in range(self.columnas):

if self.libre((f,c)):

res.append((f,c))

return res

def posiciones\_vecinas\_libre(self, pos):

res = self.posiciones\_vecinas(pos)

res = [ p for p in res if self.libre(p)]

return res

def posiciones\_vecinas\_con\_ocupantes(self, pos):

res = self.posiciones\_vecinas(pos)

res = [ (p, self.posicion(p)) for p in res if self.ocupada(p)]

return res

**Auxiliares**

El método posiciones\_vecinas es el que define la topología del terreno. En este caso, una celda tiene como vecinos a las 8 celdas con las que comparte un borde o un vértice (excepto en los bordes). Es posible modificar la dinámica del mundo modificando esta función (por ejemplo, hacerlo cilíndrico, esférico o toroidal).

def posiciones\_vecinas(self, pos):

desp=[(-1, -1), (-1, 0), (-1, 1),(0, 1), (1, 1), (1, 0),(1, -1), (0, -1)]

for i in range(len(desp)):

f = (desp[i][0] + pos[0])

c = (desp[i][1] + pos[1])

desp[i] = (f, c)

desp = [ (f,c) for f,c in desp if (0<=f and f<self.filas) and (0<=c and c<self.columnas) ]

return desp

Se pueden definir como adyacentes sólo las que compartan un borde, o las que estén a 2 celdas de distancia, incluso es posible definir que en el borde del tablero vuelva a empezar del otro lado (condiciones periódicas).

**El mundo**

El mundo será el encargado de hacer un tablero que usará de soporte, llenarlo de animales y ordenarlos para que se comporten de cierta forma en cada etapa que se implemente.

**Constructor**

En el constructor se define el tamaño del mundo, la cantidad inicial de Leones y Antílopes, tiene la opción de que imprima información de lo que va sucediendo con el parámetro debug.

def \_\_init\_\_(self, columnas, filas, n\_leones, n\_antilopes, debug=False):

super(Mundo, self).\_\_init\_\_()

self.debug = debug

self.ciclo = 0

self.tablero = Tablero(filas, columnas)

self.llenar\_mundo(n\_leones, n\_antilopes)

El constructor delega la tarea de llenar el tablero al método llenar\_mundo:

def llenar\_mundo(self, n\_leones, n\_antilopes):

for \_ in range(n\_leones):

if self.tablero.hay\_posiciones\_libres():

print\_debug("ubicando un leon", self.debug)

self.tablero.ubicar\_en\_posicion\_vacia(Leon())

for \_ in range(n\_antilopes):

if self.tablero.hay\_posiciones\_libres():

print\_debug("ubicando un Antilope", self.debug)

self.tablero.ubicar\_en\_posicion\_vacia(Antilope())

**Modelado de la dinámica**

En la etapa de movimiento, para cada posición ocupada del tablero, se indican cuales son sus vecinos libres y se lo manda a moverse, en caso de que el animal responda con una posición, se lo mueve a la posición indicada.

def etapa\_movimiento(self):

print\_debug(f"Iniciando Movimiento en ciclo {self.ciclo}", self.debug)

for p in self.tablero.posiciones\_ocupadas():

animal = self.tablero.posicion(p)

posiciones\_libres = self.tablero.posiciones\_vecinas\_libre(p)

nueva\_posicion = animal.moverse(posiciones\_libres)

if nueva\_posicion:

self.tablero.mover(p, nueva\_posicion)

En la etapa de alimentación, es similar a la anterior, salvo que se indican cuales son sus vecinos ocupados y se los manda a alimentarse, en caso de que el animal responda con una posición, se lo mueve a la posición indicada.

def etapa\_alimentacion(self):

print\_debug(f"Iniciando Alimentación en ciclo {self.ciclo}", self.debug)

for p in self.tablero.posiciones\_ocupadas():

animal = self.tablero.posicion(p)

animales\_cercanos = self.tablero.posiciones\_vecinas\_con\_ocupantes(p)

desplazo = animal.alimentarse(animales\_cercanos)

if desplazo:

self.tablero.ubicar(desplazo, self.tablero.retirar(p))

La etapa de reproducción no está implementada. Más adelante te pediremos que la implementes. Se deben pasar los animales vecinos. En este punto se debe realizar una importante decisión de modelado. ¿Quién es el encargado de que sólo se apareen animales de la misma especie? ¿El mundo?¿O cada animal? En base a esto se deberá quizás modificar código existente.

def etapa\_reproduccion(self):

print\_debug(f"Iniciando Reproducción en ciclo {self.ciclo}", self.debug)

pass

Finalmente el método que cierra un ciclo haciendo que todos los animales cumplan años, gasten energía y retirando los que ya no están con vida.

def cerrar\_un\_ciclo(self):

print\_debug(f"Concluyendo ciclo {self.ciclo}", self.debug)

for p in self.tablero.posiciones\_ocupadas():

animal = self.tablero.posicion(p)

animal.pasar\_un\_ciclo() #envejecer, consumir alimento

if not animal.en\_vida():

self.tablero.retirar(p)

self.ciclo += 1

Los 4 métodos están para ser llamados todos juntos con:

def pasar\_un\_ciclo(self):

self.etapa\_movimiento()

self.etapa\_alimentacion()

self.etapa\_reproduccion()

self.cerrar\_un\_ciclo()

**Probando el modelo completo**

Una forma para probar el funcionamiento es tener el siguiente código que corre el mundo varias veces con una pausa de tiempo para poder verlo.

m = Mundo(12, 6, 5, 15, debug=True)

import time

for i in range(20):

m.pasar\_un\_ciclo()

time.sleep(2)

print(i +1)

print(m)

Para empezar a explorar más en serio el modelo deberías completar los métodos que estén indicados con pass y otros detalles, según te indicamos en los siguientes ejercicios. Agregá además cualquier método que consideres necesario para obtener información, o modelar algún comportamiento.

**Ejercicio 10.17: Etapa de reproducción**

Implementá el método etapa\_reproduccion en la clase Mundo.

**Ejercicio 10.18: Acotando la reproducción**

Implementá la lógica necesaria para que los animales pueden reproducirse únicamente una vez por año (ya sean los que inician la reproducción o los que son compañeres).

**Ejercicio 10.19: Alcanzando la madurez**

Implementá la lógica necesaria para que un animal sólo puede reproducirse cuando ya tiene 2 años cumplidos.

Si llegaste hasta acá, por favor guardá todo junto en un solo archivo simulacion.py y entregalo al finalizar la clase.

Una vez realizado esto hay diversas opciones para usarlo y expandirlo.

**Extensiones del modelo**

A continuación una lista no exhaustiva de las extensiones posibles al modelo, podés incorporar algunas, o todas:

* Ningún animal se alimenta más de una vez en una etapa. Modifića el método tiene\_hambre() de la clase Animal para que no siempre tenga hambre.
* Si un animal se alimenta en una etapa, no necesita alimentarse más por un turno entero.
* Cuando un León ataca un Antílope, no siempre se lo come, a veces el Antílope logra escapar. Modelar esto con 3 posibles escenarios:
  + Existe una probabilidad p (fija), de que el León tenga éxito (con el módulo random se puede hacer random.random() > p)
  + La probabilidad es variable dependiendo de la edad del León, siendo muy baja cuando es cachorro, alta cuando es adulto, y baja de nuevo en su vejez.
  + La probabilidad es variable dependiendo de la edad del León, igual al anterior, y la edad del Antílope, siendo la probabilidad de que escape muy baja cuando es cachorro, alta cuando es adulto, y baja de nuevo en su vejez. Una forma de calcular esto es:
  + Se puede hacer uso de los vecinos para modificar esta probabilidad, entonces si los Antílopes están en manada, que la probabilidad de comer del León sea más baja aún.
* Extender la vecindad a un radio de 2 para todos los animales.
* Extender la vecindad para distinto radio dependiendo la especie.
* Que el Antílope pueda visualizar Leones a cierta distancia (ej. radio 3), y desplazarse en otra dirección.

**Exploraciones**

* Analizar distintos valores para la construcción del mundo a lo largo del tiempo. ¿Es posible encontrar un equilibrio entre la cantidad de leones y antílopes?
* ¿Qué sucede con la supervivencia si se modifican los atributos propios de la clase León y Antílope de manera de que tengan mayor (o menor) autonomía, o sean más longevos?
* ¿Se detectan ciclos de mayoría de una especie y después de la otra?¿Hay dominación?

**10.7 Cierre de la clase**

En esta clase aprendiste sobre generadores e iteradores, dos conceptos muy interesantes de Python. Viste que el mecanismo de iteración es una forma de dialogar con un objeto. Además, aprendiste los métodos que necesitás implementar para que un objeto que creaste sea iterable.

Discutimos también las ventajas de construír un programa con estos conceptos. Los programas resultan mas versátiles, modulares, y ligeros. Como con el concepto de comprensión de listas, podés usar la sintaxis de comprensión para construír un iterable: una expresión sobre la cual podés iterar sin necesidad de almacenar todos los elementos posibles en una lista ni escribir una función para calcularlos.

Por último, viste la ventaja de crear pipelines: estructuras de procesamiento de datos configurables en vuelo, altamente modulares.

Aunque no la uses de inmediato, cuando te la apropies, esta arquitectura puede cambiarte (un poquito) la vida.

Para cerrar esta clase, prepará:

* El archivo informe.py del [Ejercicio 10.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/02_protocolo_Iteracion.md#ejercicio-102-iteraci%C3%B3n-sobre-objetos).
* El archivo camion.py del [Ejercicio 10.3](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/02_protocolo_Iteracion.md#ejercicio-103-un-iterador-adecuado) (o [Ejercicio 10.14](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/05_Mas_generadores.md#ejercicio-1014-expresiones-generadoras-como-argumentos-en-funciones)) que va a jugar en la revisión de pares.
* El archivo vigilante.py del [Ejercicio 10.7](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/03_iteracion_a_medida.md#ejercicio-107-cambios-de-precio-de-un-cami%C3%B3n).
* El archivo ticker.py del [Ejercicio 10.12](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/04_Producers_consumers.md#ejercicio-1012-el-pipeline-ensamblado) (o del [Ejercicio 10.15](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/05_Mas_generadores.md#ejercicio-1015-c%C3%B3digo-simple)).
* Y, opcionalmente, el archivo simulacion.py del [Ejercicio 10.19](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/10_Generadores_e_Iteradores/06_PredadorPresa.md#ejercicio-1019-alcanzando-la-madurez).

Además te pedimos que completes [este formulario](https://docs.google.com/forms/d/176NfxPBJ7yqvl2MV79QQAezE8bvGbo4WxD5kVIR0OeY) usando tu dirección de mail como identificación. Al terminar vas a obtener un link para enviarnos tus ejercicios.

¡Nos vemos!