Programación Avanzada
IIC2233 2024-1

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Dante Pinto - Francisca Cattan

Anuncios

1. Hoy tenemos la tercera actividad evaluada.
2. Hablaremos un rato de las respuestas de la ETC.
3. Encuesta de Carga Académica. ¡Respóndanla!

Hablemos de la ETC

Evaluación Temprana de Cursos

Positivos

- ✅ Modalidad *Flipped Classroom*
- ✅ Extensión del repaso
- ✅ Explicaciones de contenidos claras y didácticas

Negativos

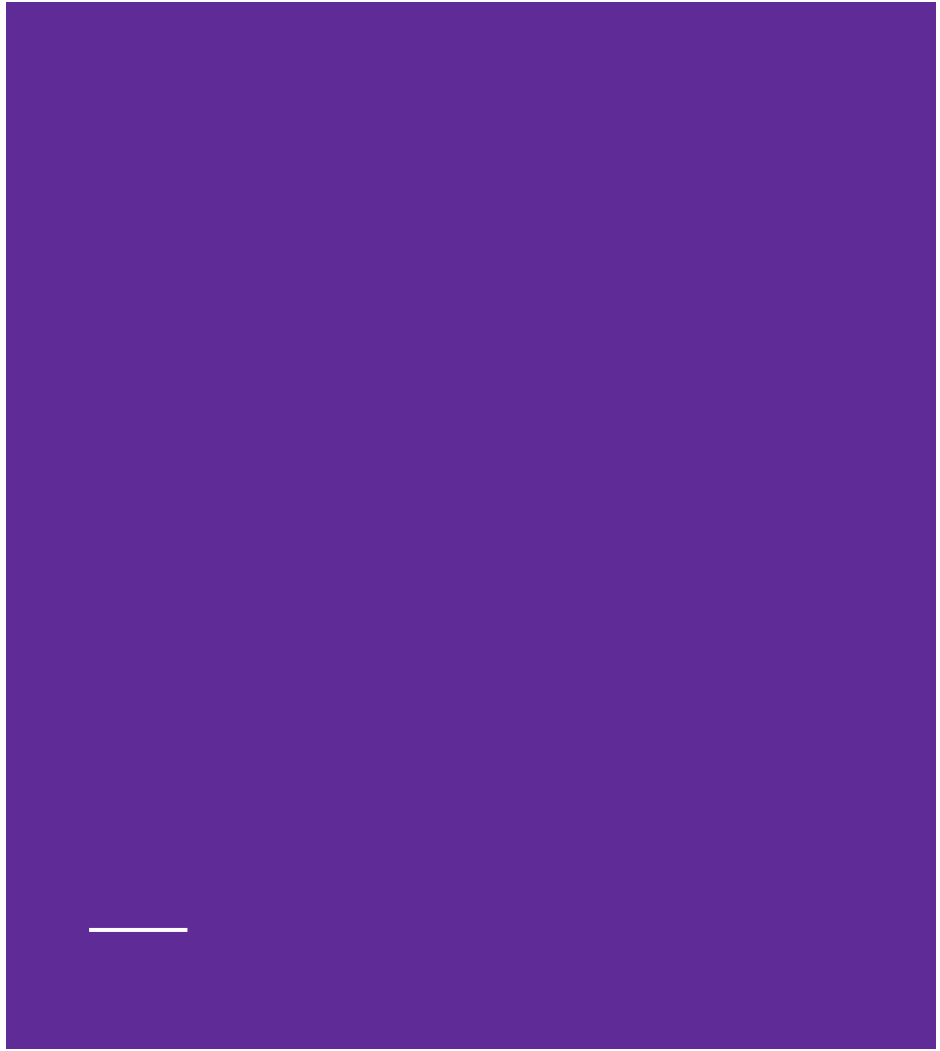
- ❌ Modalidad *Flipped Classroom*
- ❌ Extensión del repaso
- ❌ Carga académica

Compromisos

- 👍 Estudiar con anticipación
- 👍 Practicar los contenidos antes de las evaluaciones

Repaso

Iterables, Iteradores y Generadores



Iterables e Iteradores

Un **iterable** es cualquier objeto sobre el cual se puede iterar.

Un **iterador** es quien itera sobre dicho iterable.

Iterables e Iteradores

Metáfora para entender:
Un **repartible** es cualquier objeto sobre el cual se puede **repartir**.

~~Iterable~~
Repartible

~~Iterador~~
Repartidor

El que algo sea “repartible” indica que puede ser “repartido”. Cuando en verdad queremos “repartir”, el “repartidor” lo hace.

Iterables e Iteradores

```
class Repartible:
    def __init__(self, pedidos):
        self.pedidos = pedidos

    def __iter__(self):
        return RepartidorDePedidos(self)
```

1

Un “repartible” se puede “repartir”, por lo que cada vez que queramos recorrer nuestros pedidos, lo hace un **Repartidor** (1).

Iterables e Iteradores

```
class RepartidorDePedidos:
    def __init__(self, repartible):
        # Para no modificar original
        self.repartible = copy(repartible)

    def __iter__(self):
        return self

2   def __next__(self):
        if not self.repartible.pedidos:
            raise StopIteration("Sin pedidos")

        pedidos = self.repartible.pedidos
        proximo_pedido = pedidos.pop(0)
        return proximo_pedido
```

Cada vez que el **Repartidor** pasa al siguiente pedido (2) este se elimina de la lista, es consumido.

En un iterable, solo está la información y no se modifica, mientras que un iterador va avanzando en el iterable y consumiendo cada elemento.

Iterables e Iteradores

```
class RepartidorDePedidos:
    def __init__(self, repartible):
        # Para no modificar original
        self.repartible = copy(repartible)

3   def __iter__(self):
        return self

    def __next__(self):
        if not self.repartible.pedidos:
            raise StopIteration("Sin pedidos")

        pedidos = self.repartible.pedidos
        proximo_pedido = pedidos.pop(0)
        return proximo_pedido
```

En (3) vemos otra propiedad especial. Para que algo sea iterable, debe implementar el método `__iter__` y retorna un iterador.

En (3), **Repartidor** se retorna a sí mismo, por lo que es tanto iterador como iterable.

Iterables e Iteradores

```
iterable = Iterable()       # 🌽, 🥔, 🥚
iterador = iter(iterable)   # Iterable.__iter__

print(next(iterador))       # Iterador.__next__
>> 🌽

print(next(iterador))
>> 🥔

print(next(iterador))
>> 🥚

print(next(iterador))       # Si no quedan elementos...
>> StopIteration
```

Generadores

Los **generadores** son un caso especial de los **iteradores**.

```
(i for i in range(10))
```

Generador

```
yield elemento
```

Función generadora

Generadores

```
def ingredientes():
    yield 🌽
    yield 🥔
    yield 🥚

generador = ingredientes()

# El generador "recuerda"
# dónde quedó la ejecución
# y continúa al hacer next
```

```
print(next(generador))
>> 🌽

print(next(generador))
>> 🥔

print(next(generador))
>> 🥚

print(next(generador))
>> StopIteration
```

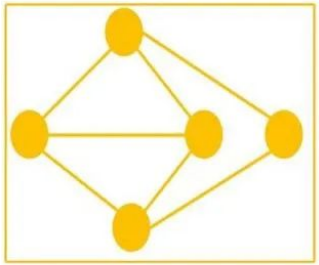
Listas Ligadas

Motivación

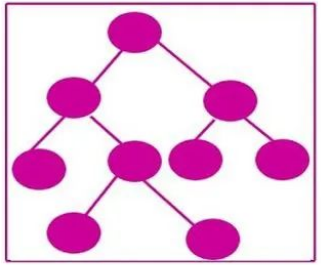
¿Todos los lenguajes de programación tienen las mismas estructuras de datos?

¿Cómo funcionan? ¿Cuál es su base?

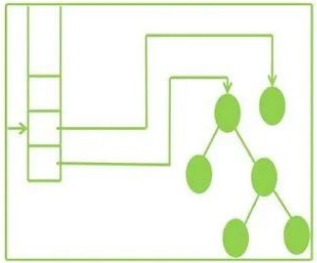
Acá veremos un inicio, pero podrán aprender más en el curso IIC2133: Estructura de Datos y Algoritmos.



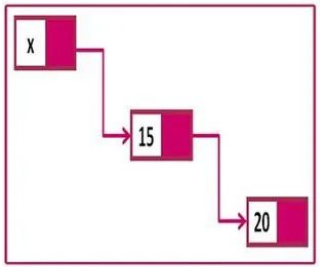
Graph



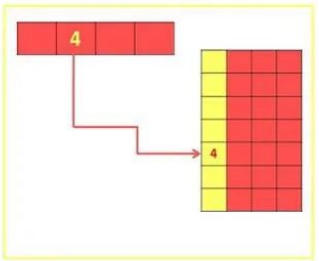
Tree



Stack



Link list



Hashing

Nodo

- Corresponde a la **base** de las estructuras de datos.
- Es una unidad indivisible que contiene **datos**.
- Cada nodo mantiene cero o más **referencias** con otros nodos.

```
class Nodo:
    def __init__(self, valor=None):
        self.valor = valor
        self.siguiente = None
```

Lista Ligada

- Estructura que almacena nodos en un **orden secuencial**.
- Cada nodo posee una referencia a un **único nodo sucesor**.
- El primer nodo corresponde a la **cabeza**, y mientras que el último, **cola**.

```
class ListaLigada:
    def __init__(self):
        self.cabeza = None
        self.cola = None
```

Lista Ligada

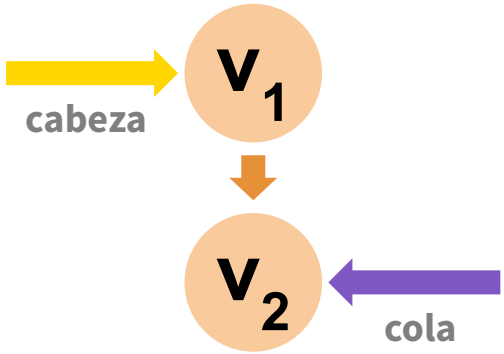
```
l_ligada = ListaLigada()
```

Lista Ligada

```
l_ligada = ListaLigada()

l_ligada.agregar(v1)
```

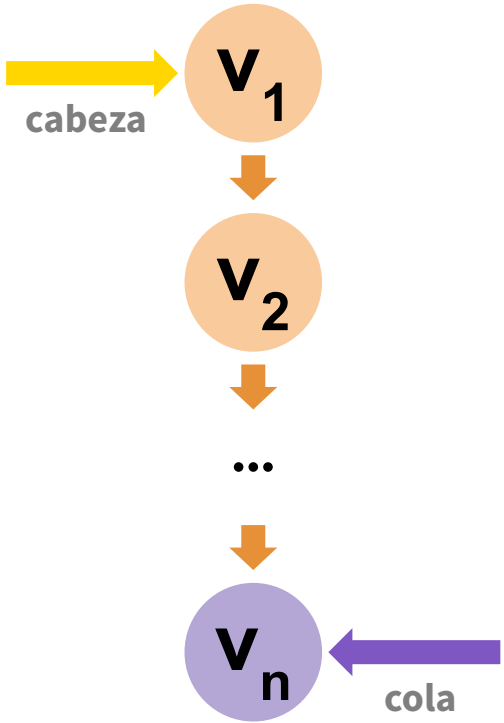
Lista Ligada

```
l_ligada = ListaLigada()

l_ligada.agregar(v1)
l_ligada.agregar(v2)
```



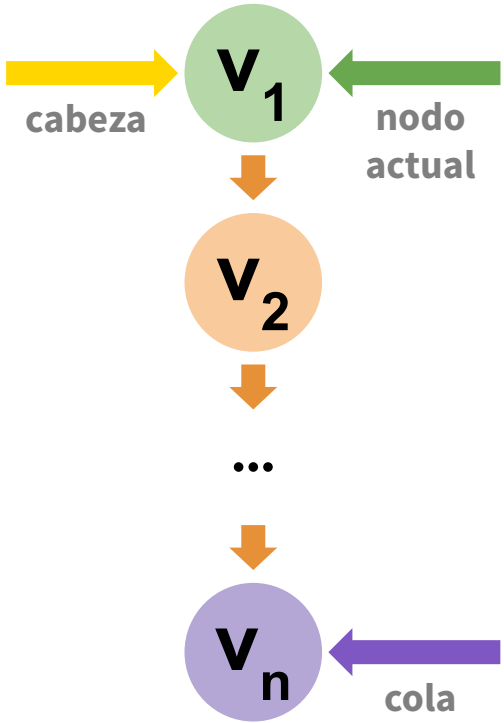
Lista Ligada

```
l_ligada = ListaLigada()

l_ligada.agregar(v1)
l_ligada.agregar(v2)
          ⋮
l_ligada.agregar(vn)
```



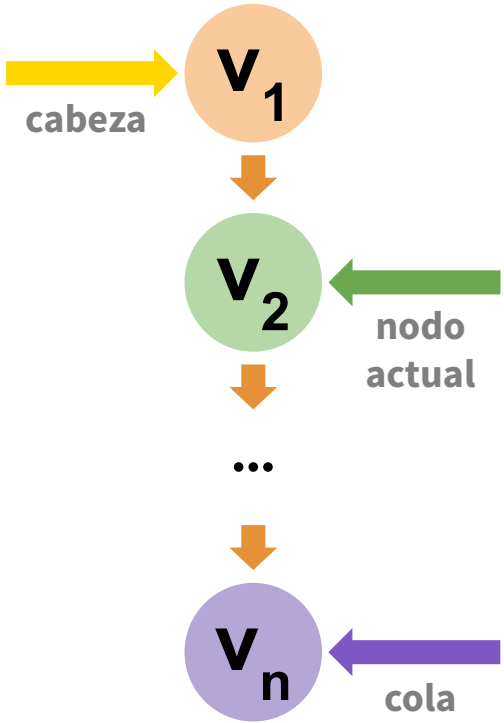
Lista Ligada

```
l_ligada = ListaLigada()

l_ligada.agregar(v1)
l_ligada.agregar(v2)
          ⋮
l_ligada.agregar(vn)

l_ligada.obtener(vi)
```



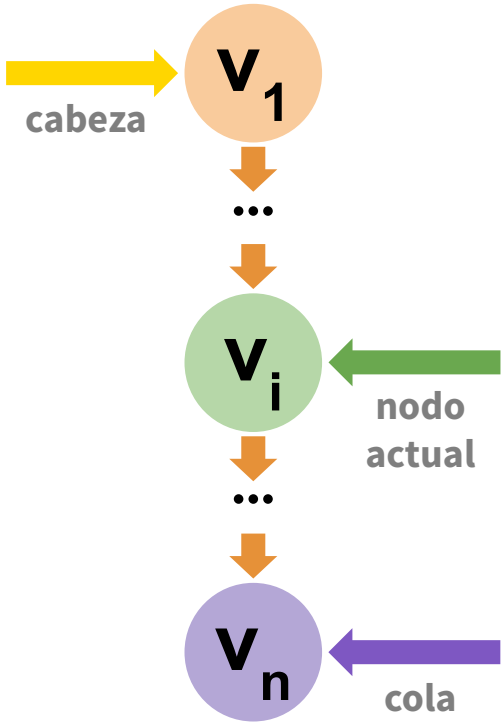
Lista Ligada

```
l_ligada = ListaLigada()

l_ligada.agregar(v1)
l_ligada.agregar(v2)
          ⋮
l_ligada.agregar(vn)

l_ligada.obtener(vi)
```



Lista Ligada

```
l_ligada = ListaLigada()

l_ligada.agregar(v1)
l_ligada.agregar(v2)
          ⋮
l_ligada.agregar(vn)

l_ligada.obtener(vi)
```



Programación Funcional

Motivación

En el mundo de la programación existen distintos paradigmas de las programación:

Procedimental

- Un programa lineal.
- **Lista de instrucciones** que indican al computador qué hacer en cada paso.

Orientada a Objetos

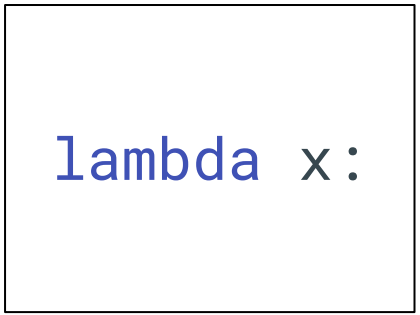
- Modela funcionalidades a través **objetos** y la **interacción** de estos.
- Da sentido al programa, a través de los objetos.

Programación Funcional

- Se estructura la solución como un **conjunto de funciones**.
- Las **funciones no tienen estado**, es decir, el *output* depende exclusivamente del *input*.

Programación Funcional

Las **funciones *lambda*** son funciones anónimas y de uso fugaz.

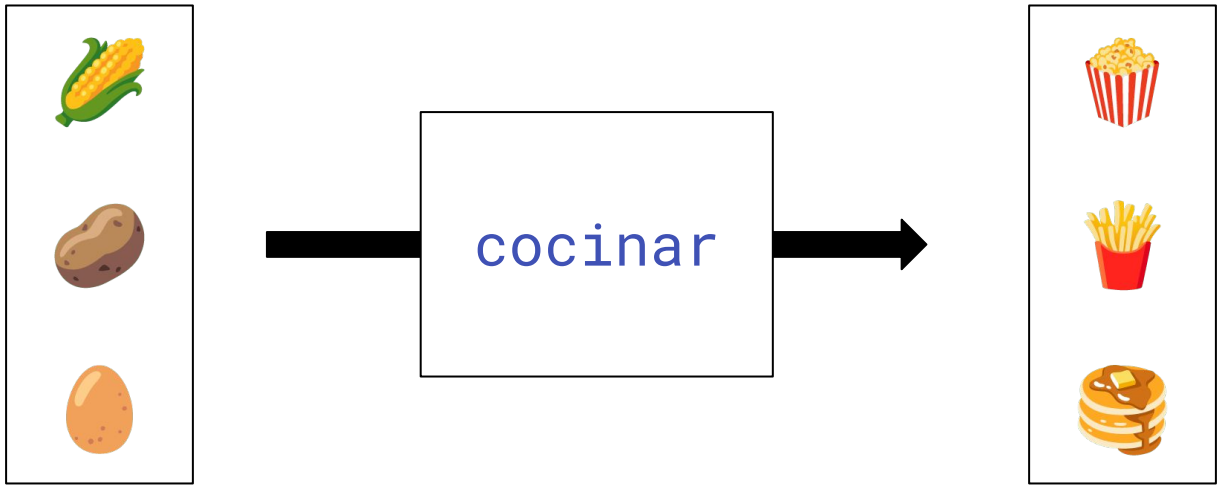


Lambda

```
lambda x: x * 2
lambda a, b: a + b
lambda p: p.procesar()
lambda a, p: a + p.precio
```

Programación Funcional

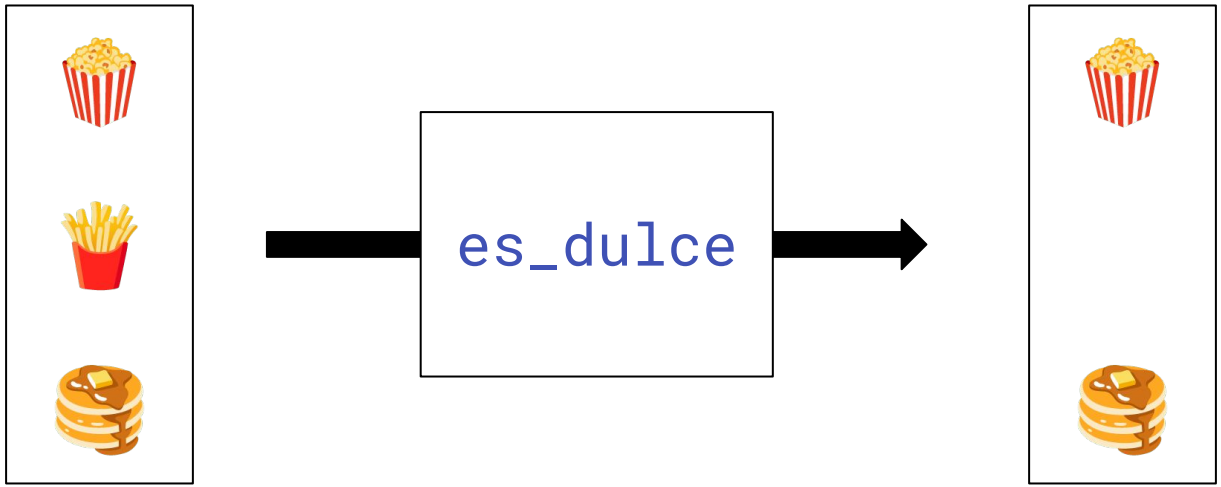
La función ***map*** aplica la **función** a cada elemento de un **iterable**.



`map(cocinar, [🌽, 🥔, 🥚]) → [🍿, 🍟, 🥞]`

Programación Funcional

La función ***filter*** aplica la **función** para seleccionar elementos.



`filter(es_dulce, [🍿, 🍟, 🥞])` → `[🍿, 🥞]`

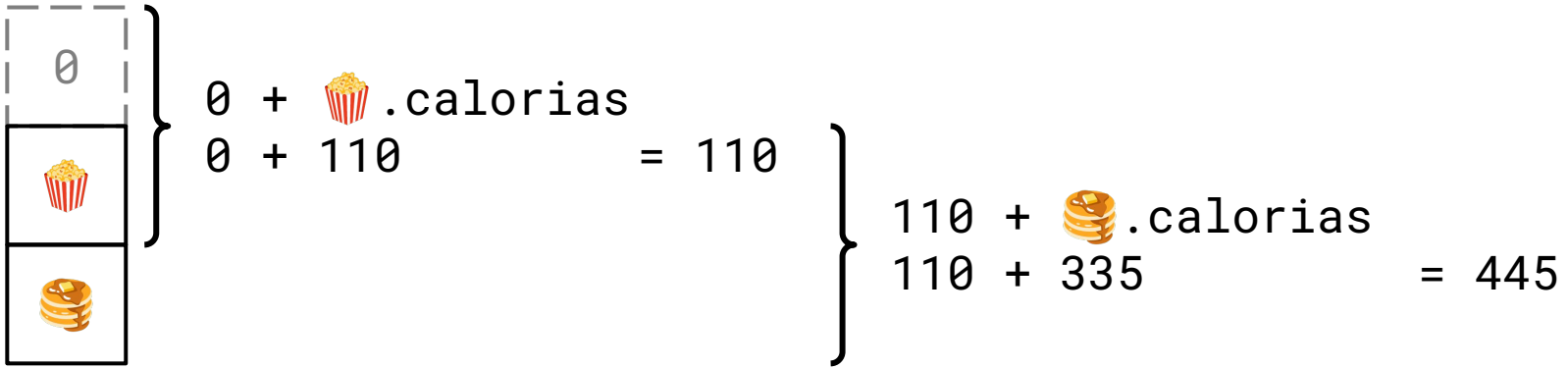
Programación Funcional

La función ***reduce*** aplica la **función** para componer el resultado hasta que quede solo un elemento.


```
reduce(sumar_calorias, [🍿, 🥞], 0) ⟶ (total)
```

Programación Funcional

```
reduce(sumar_calorias, [🍿, 🥞], 0)

def sumar_calorias(cal_acumuladas, alimento):
    return cal_acumuladas + alimento.calorias
```



Programación Avanzada
IIC2233 2024-1

Hernán Valdivieso - Daniela Concha - - Dante Pinto - Francisca Cattan