Programación Avanzada
IIC2233 2023-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattan

Anuncios

1. Hoy tenemos la segunda actividad.
2. Encuesta de Carga Académica. ¡Respóndanla!
3. Pueden hacer entregas parciales en Canvas, aprovechen de guardar su progreso.

Formato Actividades

Repaso

Estructuras de datos

- Forma especializada de agrupar datos.
- Almacenamiento, acceso y utilización eficiente.
- ¿Qué estructura es mejor para cada caso?

Lo básico

Estructuras secuenciales

- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.

- Listas
- Tuplas
  - *named tuples*

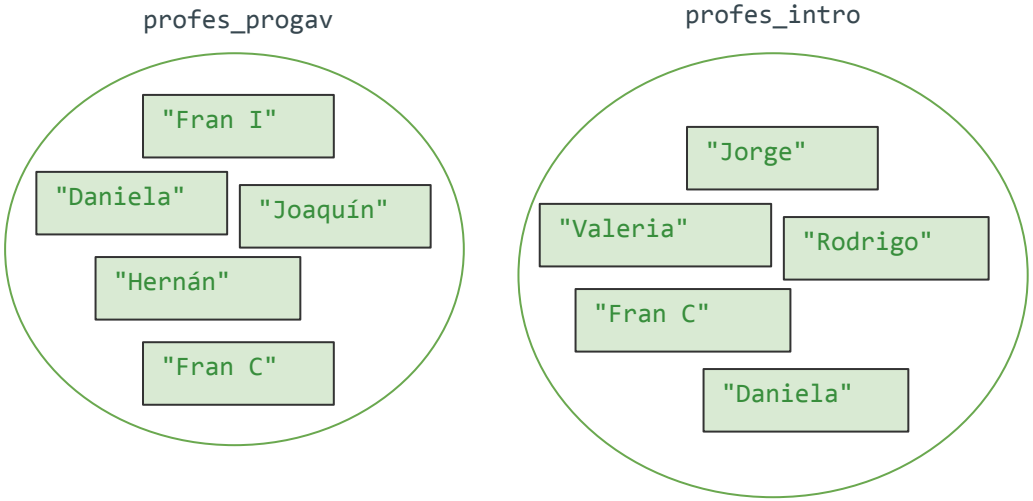
Estructuras no secuenciales

- No hay orden entre elementos.
- Búsqueda de elementos muy eficiente.

- *Sets* (Conjuntos)
- Diccionarios

Sets (Conjuntos)

- No hay orden entre elementos.
- Búsqueda de elemento específico muy eficiente.
- No permite duplicados.
- Solo permite elementos inmutables.

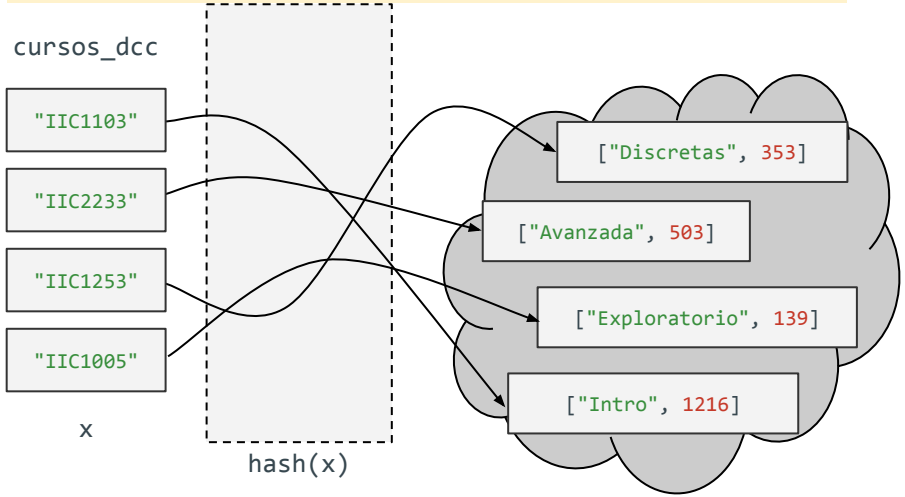



```
profes_intro = {
    "Jorge", "Valeria", "Fran C",
    "Rodrigo", "Daniela"
}
profes_progav = set(profesores)

print("Cristian" in profes_intro)
print(profes_intro & profes_progav) # en común
```

Diccionarios

- Almacena pares: llave-valor.
- Búsqueda de llave específica muy eficiente.
- Llaves no pueden estar duplicadas.
- Solo permite elementos inmutables como llaves.
- Valor puede ser cualquier elemento.

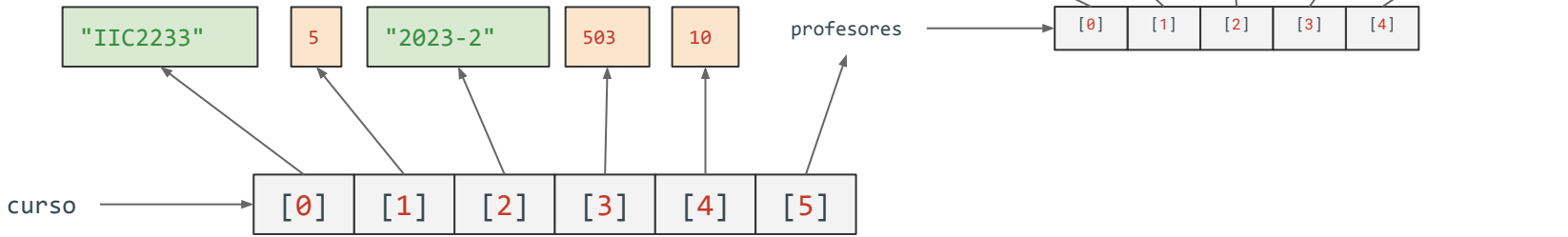
```
cursos_dcc = { "IIC1103": ["Intro", 1216],
               "IIC2233": ["Avanzada", 528],
               "IIC1253": ["Discretas", 353],
               "IIC1005": ["Exploratorio", 139]
             }

print(cursos_dcc["IIC2233"])
print(cursos_dcc["IIC1103"][1])
```



Tuplas

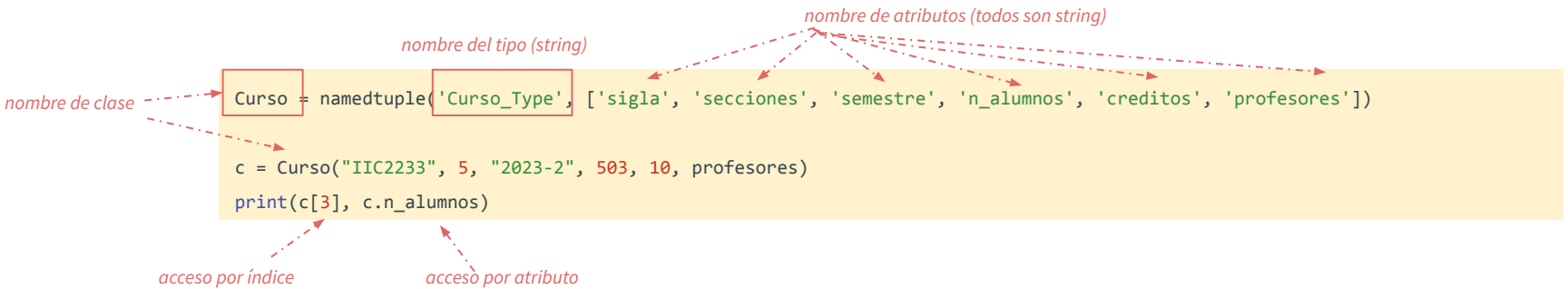
- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.
- Immutable.

```
curso = ("IIC2233", 5, "2023-2", 503, 10, profesores)
print(curso[3])
```



named tuples

- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.
- Immutable.
- **Cada posición tiene un nombre (atributo).**

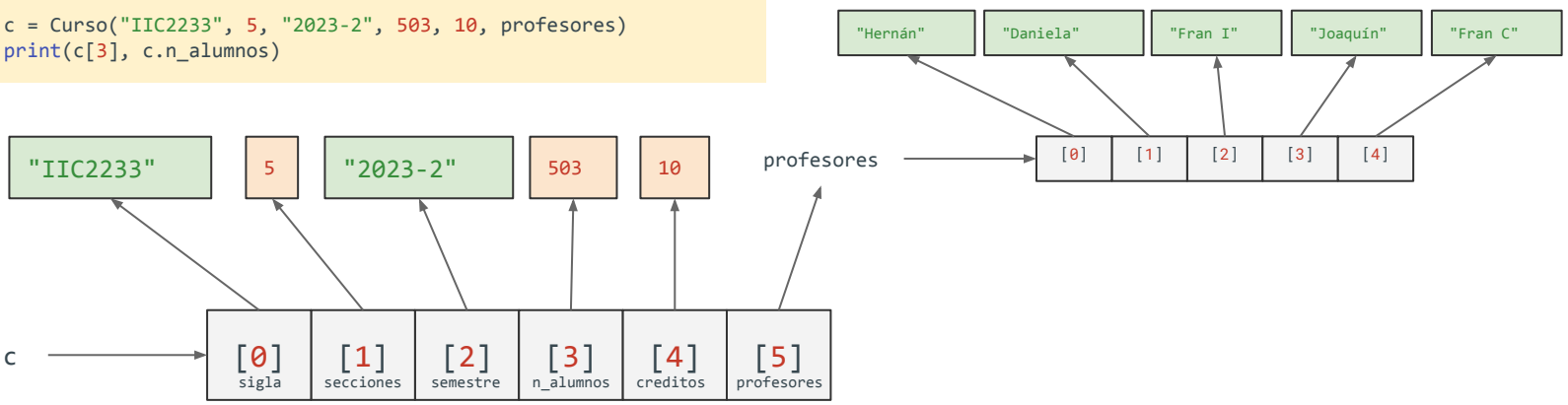


named tuples

- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.

- Immutable.
- **Cada posición tiene un nombre (atributo).**

```
c = Curso("IIC2233", 5, "2023-2", 503, 10, profesores)
print(c[3], c.n_alumnos)
```



Resumen

Estructura	Insertar	Búsqueda por índice	Búsqueda por llave	Búsqueda por valor
Lista	✔	✔✔✔	✘	✔
Tupla	✘	✔✔✔	✘	✔
Diccionario	✔✔✔	✘	✔✔✔	✔
Set (Conjunto)	✔✔✔	✘	✔✔✔	✔

Iterables e Iteradores

Lo básico

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

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

Lo básico

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.

Lo básico

```
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**.

Lo básico

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

    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
```

3

2

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.

Lo básico

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

    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
```

3

2

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.

Lo básico

```
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
```

Lo básico

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

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

Generador

```
yield elemento
```

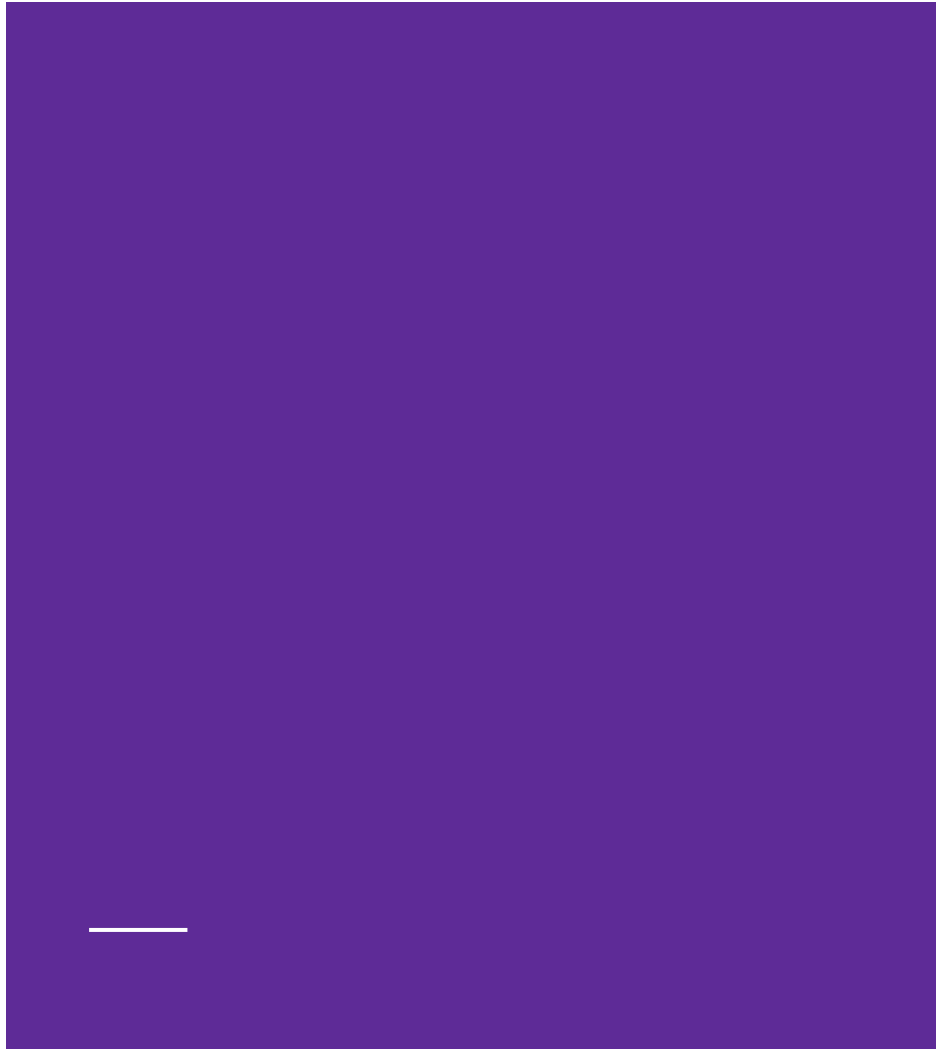
Función generadora

Lo básico

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

generador = ingredientes()

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

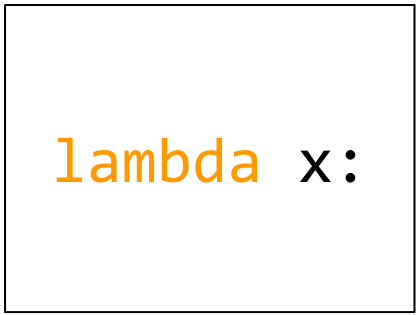
```
print(next(generador))
>> 🌽
print(next(generador))
>> 🥔
print(next(generador))
>> 🥚
print(next(generador))
>> StopIteration
```

Trabajando con iterables



Trabajando con iterables

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

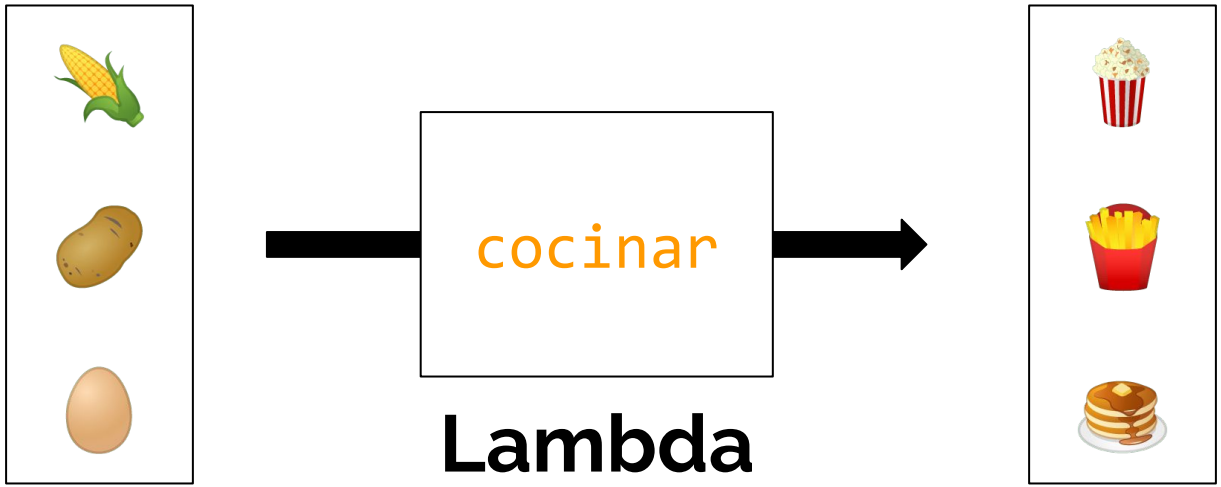


Lambda

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

Trabajando con iterables

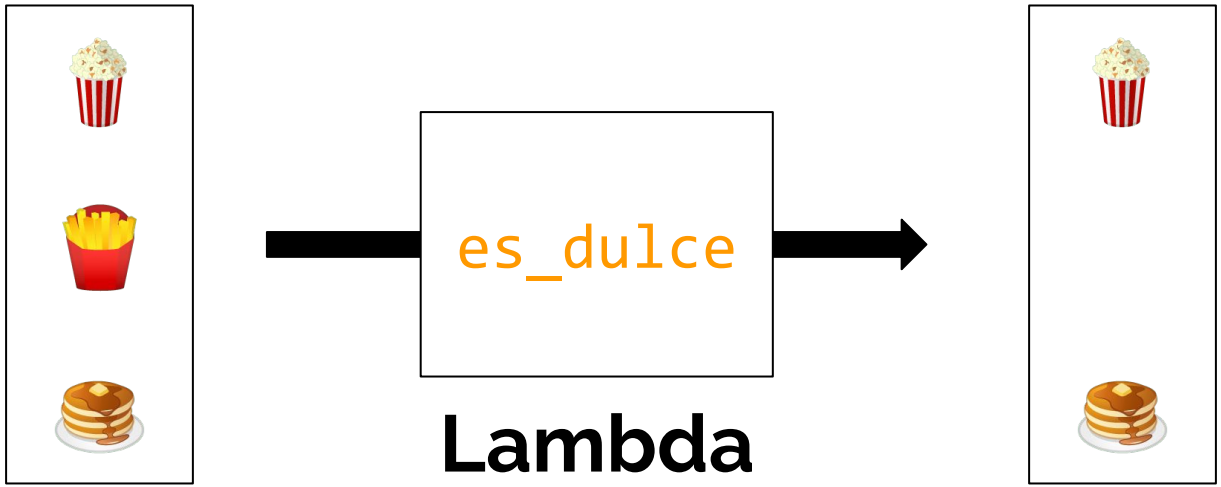
La función ***map*** aplica la **función** a cada elemento de un **iterable**.



`map(cocinar, [🌽, 🥔, 🥚])` → `[🍿, 🍟, 🥞]`

Trabajando con iterables

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



`filter(es_dulce, [🍿, 🍟, 🥞])→[🍿, 🥞]`

Trabajando con iterables

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

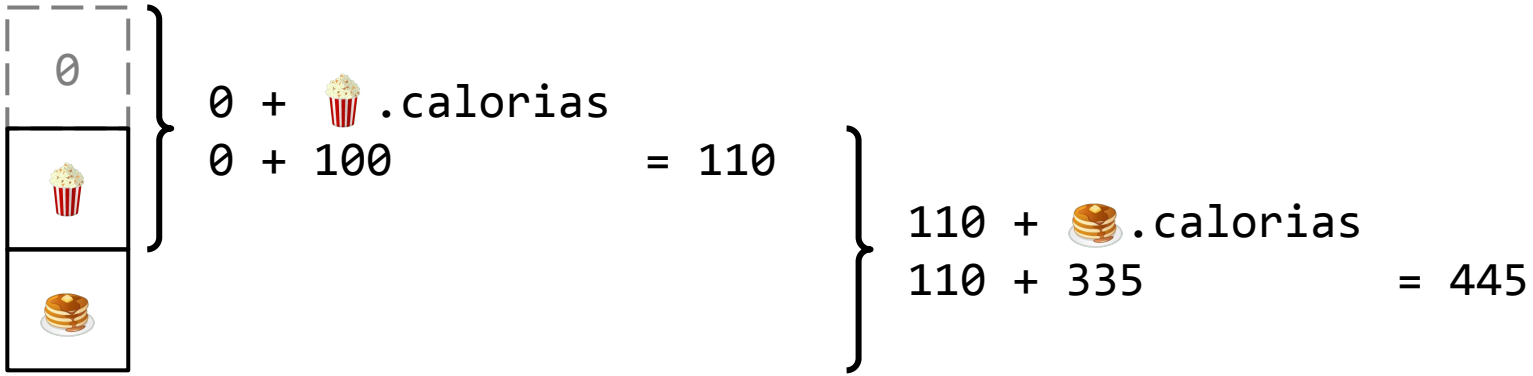
Lambda

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

Trabajando con iterables

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

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



Programación Avanzada
IIC2233 2023-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattan