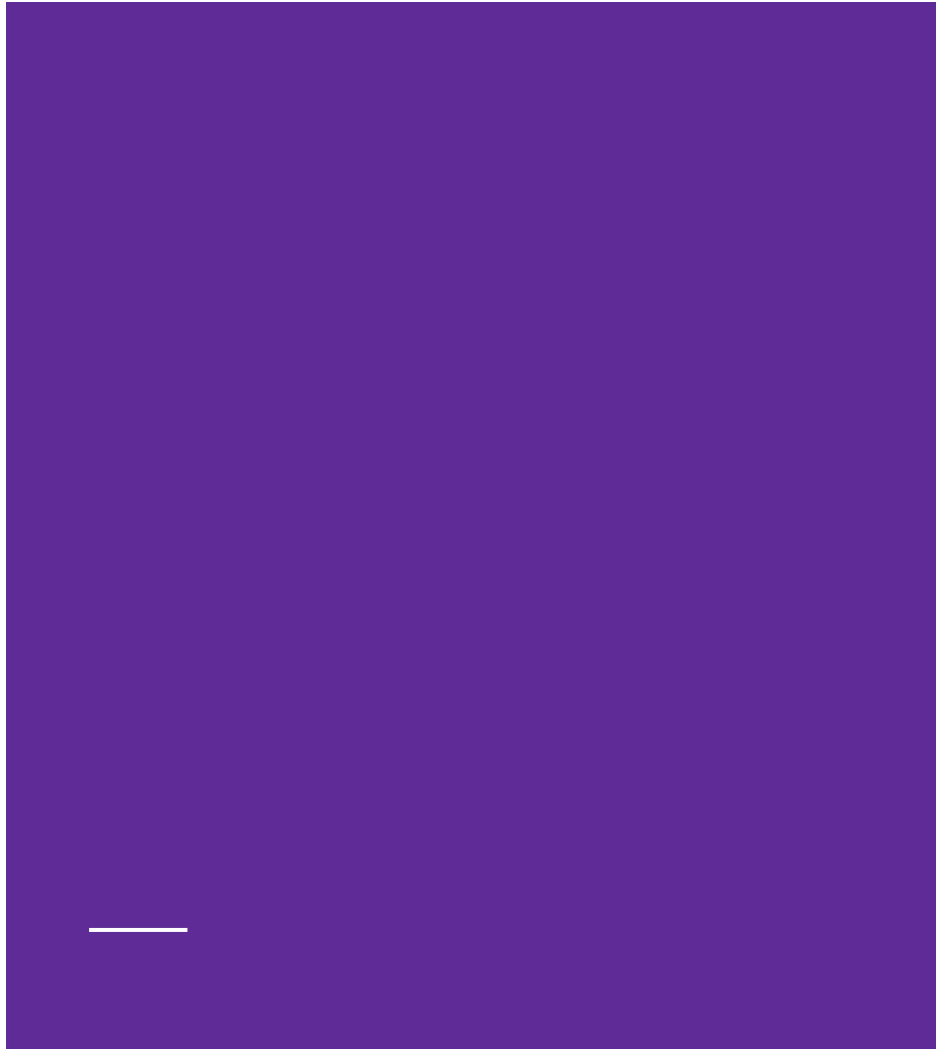
Programación Avanzada
IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattan

Anuncios

1. Hoy es la Experiencia 1.
2. La ECA se encuentra disponible para responder de domingo a martes.

Modelación OOP
e
Iterables



Modelación OOP

- Clases Abstractas:
  Prototipo de una clase.
- Diagrama de Clases:
  Planificar antes de actuar.

Clases Abstractas

- Clase que no se instancia directamente.
- Subclases implementan métodos abstractos.
- En Python, el módulo *abc* nos permite definir clases abstractas

Clase abstracta

Contexto: Suponga un mundo de **fantasía** 🧙
donde se puede controlar los elementos de la naturaleza 💧🔥🌿💨.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas? **¡Depende!** **¿MaestroAgua?** **¿MaestroFuego?** **¿MaestroTierra?** **¿MaestroViento?**

¿Qué acción es común a todos, pero cada uno lo hace de forma distinta? **Entrenar**

¿Qué acción puede variar según los datos que me lleguen? **Luchar**

¿Y si alguien controla 2 elementos? **Multiherencia**

Oye, pero… ¿Cómo fuerzo que todos deban entrenar?

Clase abstracta

Contexto: Suponga un mundo de **fantasía** 🧙
donde se puede controlar los elementos de la naturaleza 💧🔥🌿💨.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas? **¡Depende!** **¿MaestroAgua?** **¿MaestroFuego?** **¿MaestroTierra?** **¿MaestroViento?**

¿Qué acción es común a todos, pero cada uno lo hace de forma distinta? **Entrenar**

¿Qué acción puede variar según los datos que me lleguen? **Luchar**

¿Y si alguien controla 2 elementos? **Multiherencia**

Oye, pero… ¿Cómo fuerzo que todos deban entrenar? **Clases abstractas**

Clase abstracta

```python
from abc import ABC, abstractmethod


class Persona(ABC):

    @abstractmethod
    def entrenar(self):
        pass
```

```python
class MaestroAgua(Persona):

    def entrenar(self):
        print("Me voy a una cascada🚿")
```

```python
class MaestroFuego(Persona):

    def entrenar(self):
        print("Necesito un volcán♨️")
```

Ejemplo: Discusión

```python
class Animal(ABC):
    @abstractmethod
    def saludar(self):
        pass

    @abstractmethod
    def despedir(self):
        pass
```

```python
class Perro(Animal):
    def saludar(self):
        print("Wenomechainsama")
```

```python
class RusselTerrier(Perro):
    def despedir(self):
        print("Tumajarbisaun")
```

¿Cuál o cuáles de estas clases se pueden instanciar?

Diagrama de Clases

- Elemento **visual** para caracterizar **clases** que componen un sistema.
- Muestran **atributos**, **comportamientos** y **relaciones** entre clases.

Diagrama de Clases

- Existe el formato UML, pero este curso no se regirá 100% por él.
  Se harán algunos cambios para simplificar su uso dentro del curso.

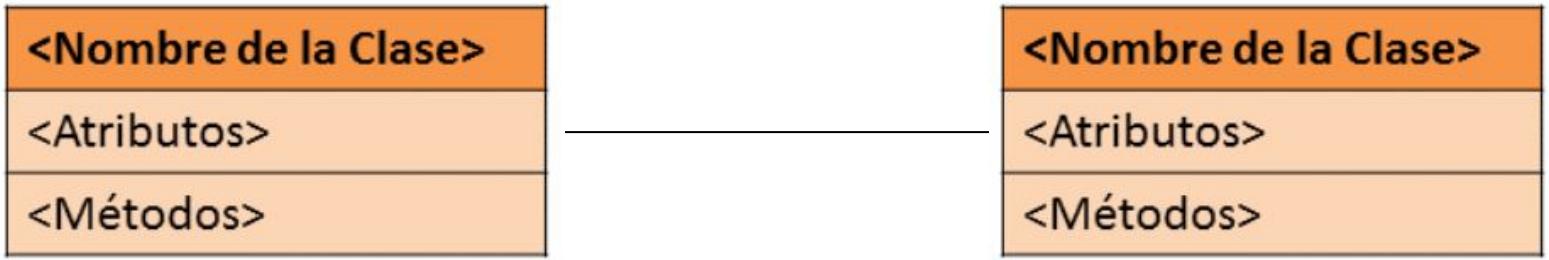


Diagrama de Clases

Formato del curso

- Parte 1: atributos y *properties*
  - Nombre
  - Tipo de dato
  - Diferenciar atributo VS *property*
- Parte 2: métodos
  - Nombre
  - Argumentos del método
  - Tipo de dato de su *return*.

Auto
+ dueño: str or None
+ marca: str
+ modelo: str
+ _kilometraje: int
+ @kilometraje: int (getter y setter)
+ conducir(kms): None
+ vender(dueño): Str

Diagrama de Clases

Relaciones - Herencia

- Se utiliza una flecha para indicar si una clase **hereda** de otra clase.
- La flecha apunta a la clase “padre”.
- En este ejemplo, `LadrilloMoneda` y `LadrilloMovil` (tipos de ladrillos del juego Super Mario) heredan de `Ladrillo`.

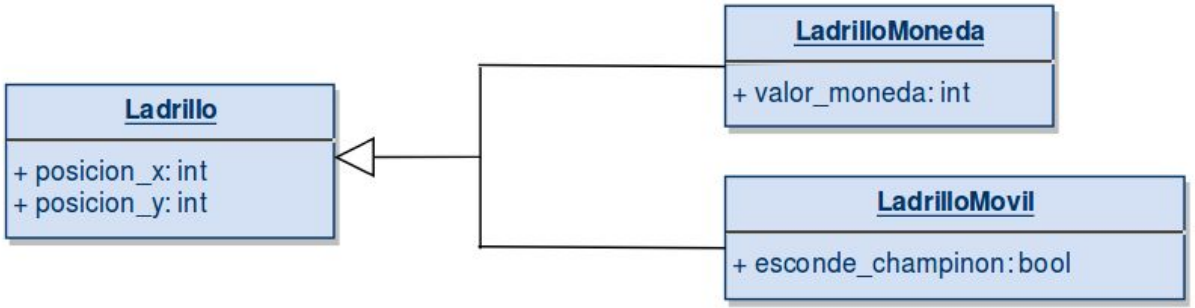
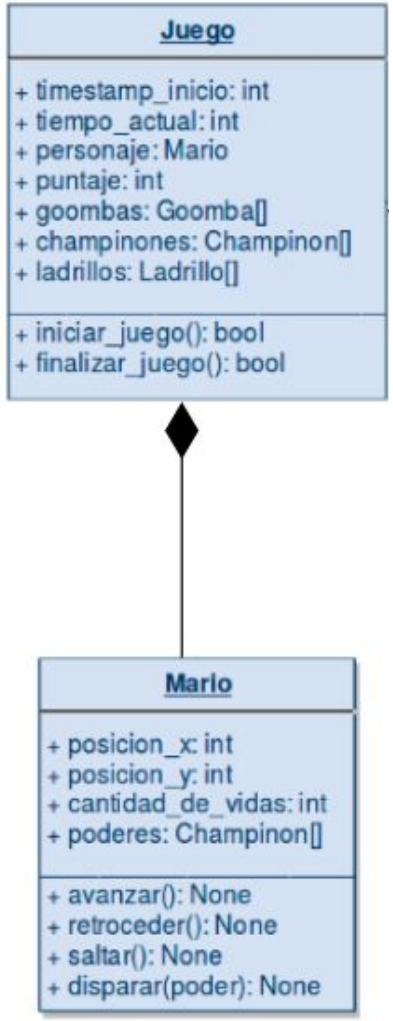


Diagrama de Clases

Relaciones - Contención

- Se utiliza un rombo para indicar si una clase **contiene** a otra clase.
- El rombo está junto a la clase “contenedora”.
- En este ejemplo, la clase Juego **contiene** a la clase Mario.



Iterables, Iteradores y Generadores

Método `__iter__()`.

Método `__next__()`.

Iterables e Iteradores

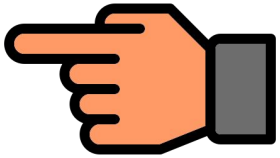
Un **iterable** es cualquier objeto sobre el cual se puede iterar.

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

Iterables e Iteradores

Metáfora para entender:
Un **recorrible** es cualquier objeto sobre el cual se puede **recorrer**.

~~Iterable~~
Recorrible



~~Iterador~~
Recorredor

El que algo sea “recorrible” indica que puede ser “recorrido”. Cuando en verdad queremos “recorrer”, el “recorredor” lo hace.

Iterables e Iteradores

```
class RecorribleDeCartas:
    def __init__(self, cartas: List[Carta]):
        self.cartas = cartas

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

1

Un “recorrible” se puede “recorrer”, por lo que cada vez que queramos recorrer por nuestras cartas, lo hace un **Recorredor** (1).

Iterables e Iteradores

```
class RecorredorDeCartas:
    def __init__(self, recorrible: List[Cartas]):
        # Para no modificar original
        self.recorrible = deepcopy(recorrible)

    def __iter__(self):
        return self
```

2

```
    def __next__(self):
        if not self.recorrible.cartas:
            raise StopIteration("Sin cartas")

        cartas = self.recorrible.cartas
        proxima_carta = cartas.pop(0)
        return proxima_carta
```

Cada vez que el **Recorredor** pasa a la siguiente carta (2) esta 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 RecorredorDeCartas:
    def __init__(self, recorrible: List[Cartas]):
        # Para no modificar original
        self.recorrible = deepcopy(recorrible)

3   def __iter__(self):
        return self

    def __next__(self):
        if not self.recorrible.cartas:
            raise StopIteration("Sin cartas")

        cartas = self.recorrible.cartas
        proxima_carta = cartas.pop(0)
        return proxima_carta
```

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

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

Iterables e Iteradores

```
class RecorredorDeCartas:
    def __init__(self, recorrible: List[Cartas]):
        # Para no modificar original
        self.recorrible = deepcopy(recorri
```

3

```
    def __iter__(self):
        return self

    def __next__(self):
        if not self.recorrible.cartas:
            raise StopIteration("Sin cartas")

        cartas = self.recorrible.cartas
        proxima_carta = cartas.pop(0)
        return proxima_carta
```

En (3) vemos otra propiedad especial. Para que algo sea [illegible] debe implementar

Entonces, ¿Iterador = Iterable?

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

¿Iterador = Iterable?

NO, un Iterador es un **TIPO** de Iterable. Para entender mejor esto, veamos algunas de sus principales diferencias:

Característica	Iterable	Iterador
Métodos Asociados	Implementa el método `__iter__()` que devuelve un iterador.	Implementa tanto el método `__iter__()` como el método `__next__()`.
Estado Interno	Un iterable **no tiene estado interno** de iteración.	Un iterador **tiene un estado interno** que recuerda su posición actual en la iteración.
Reutilización	Los iterables **pueden ser reutilizados** para obtener múltiples iteradores.	Los iteradores, una vez consumidos, **no pueden ser reutilizados**.

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**. Los cuales son **muy eficientes en memoria**

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

Generador

Generadores

```
cartas = [♠, ♦, 🃏]
generador = (cartas[i] for i in range(len(cartas))

# 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
```

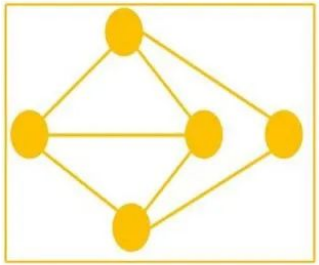
Listas Ligadas

Motivación

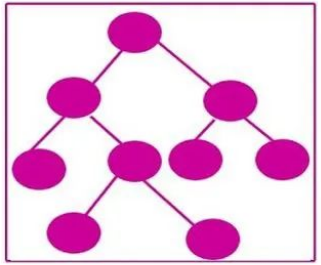
¿Todos los lenguajes de programación tienen implementadas las mismas estructuras de datos?

¿Cómo funcionan? ¿Cuál es su base?

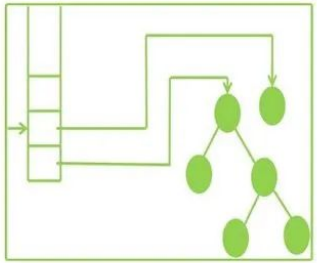
Acá veremos un inicio, pero podrán aprender más en el curso IIC2133: Estructura de Datos y Algoritmos.



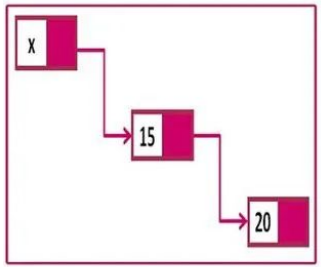
Graph



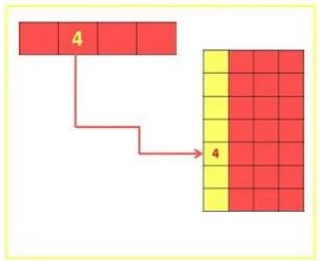
Tree



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

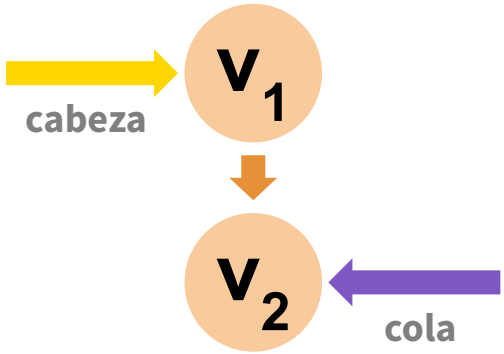
```
l_ligada = ListaLigada()
```

Lista Ligada

```
l_ligada = ListaLigada()

l_ligada.agregar(v1)
```

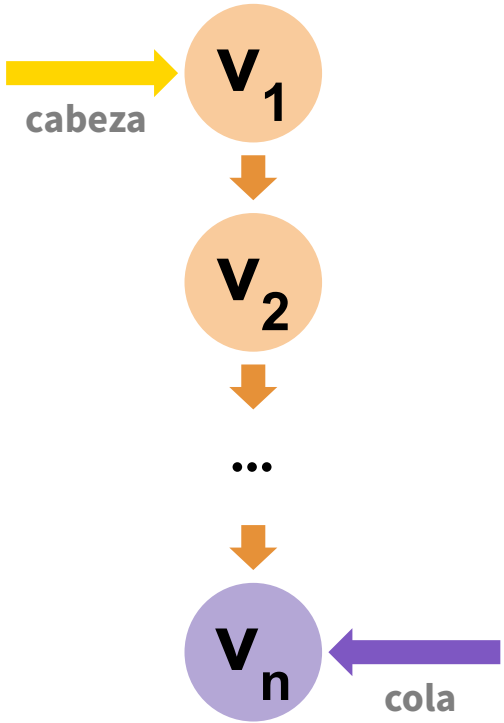
Lista Ligada

```
l_ligada = ListaLigada()

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



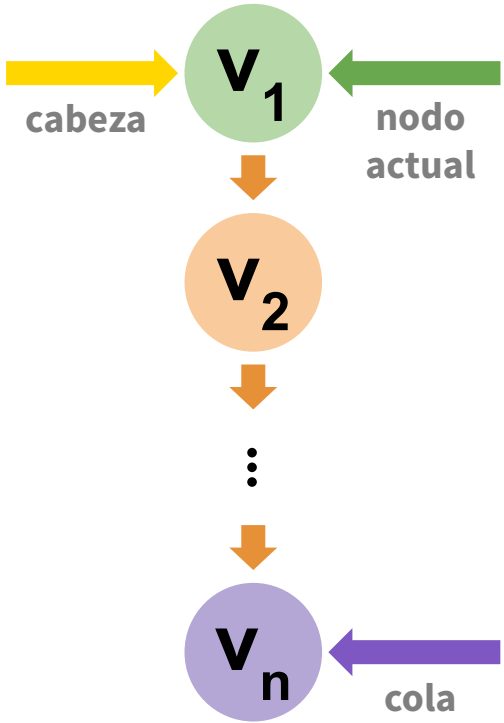
Lista Ligada

```
l_ligada = ListaLigada()

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



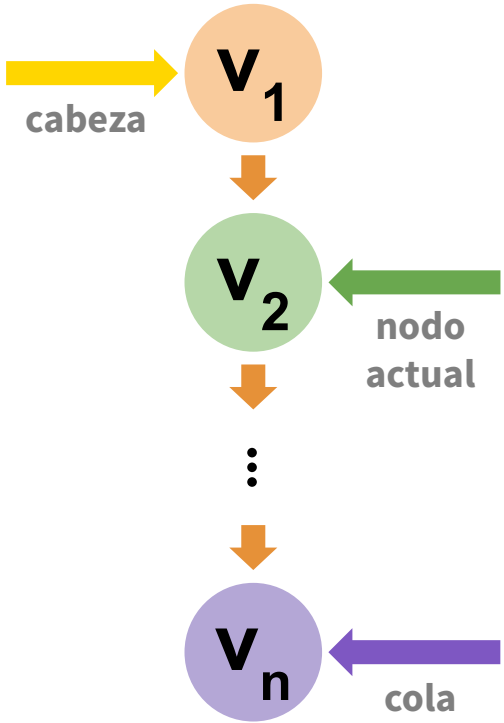
Lista Ligada

```
l_ligada = ListaLigada()

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

l_ligada.obtener(vi)
```



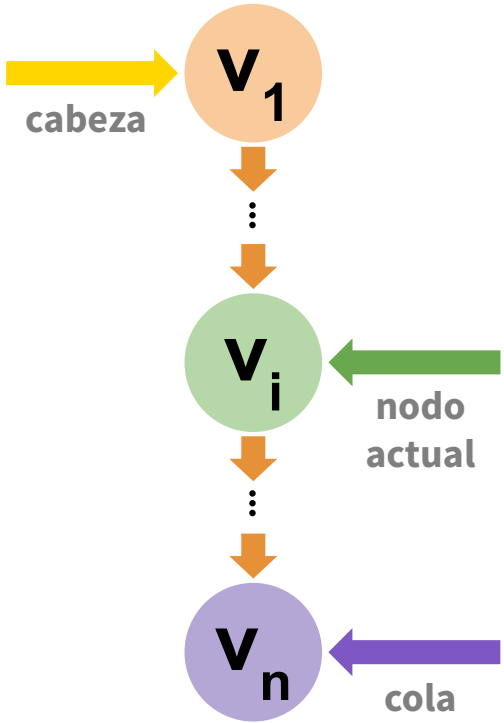
Lista Ligada

```
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)
```



Pregunta de Evaluación Escrita

Tema: Clases abstractas (Midterm 2023-2)

5. ¿Cuál(es) de la(s) siguiente(s) afirmación(es) es/son **correctas** respecto a las **clases abstractas**?
   - I. Se pueden instanciar.
   - II. La herencia de clases abstractas no permite override de sus métodos.
   - III. El módulo `abc` permite definir clases abstractas en Python.
   - IV. Permiten herencia e implementación de sus métodos.

   A) Solo IV
   B) I y III
   C) I, II y III
   D) I, III y IV
   E) II, III y IV

Pregunta de Evaluación Escrita

Tema: Clases abstractas (Midterm 2023-2)

5. ¿Cuál(es) de la(s) siguiente(s) afirmación(es) es/son **correctas** respecto a las **clases abstractas**?
   - I. Se pueden instanciar.
   - II. La herencia de clases abstractas no permite override de sus métodos.
   - III. El módulo `abc` permite definir clases abstractas en Python.
   - IV. Permiten herencia e implementación de sus métodos.

   A) Solo IV
   B) I y III
   C) I, II y III
   D) I, III y IV
   E) II, III y IV

Documentación de Python:

> *"A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract methods and properties are overridden."*

Pregunta de Evaluación Escrita

Tema: Clases abstractas (Midterm 2023-2)

5. ¿Cuál(es) de la(s) siguiente(s) afirmación(es) es/son **correctas** respecto a las **clases abstractas**?

   I. Se pueden instanciar **en Python**.
   II. La herencia de clases abstractas no permite override de sus métodos.
   III. El módulo `abc` permite definir clases abstractas en Python.
   IV. Permiten herencia e implementación de sus métodos.

   A) Solo IV
   B) I y III
   C) I, II y III
   D) I, III y IV
   E) II, III y IV

Experiencia 1

Programación Avanzada
IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattan