

# POO

## **Introducción a la programación orientada a objetos.**

Docente: Uriel Kelman.

Cátedra: Ing. Pablo Guarna.



# POO como nuevo paradigma

- En primer lugar, ¿qué es un paradigma dentro del mundo de la programación? ¿Y un modelo?
- ¿Por qué queríamos cambiar de paradigma, si ya aprendimos a programar de forma estructurada? ¿Cualquier programa se puede desarrollar en cualquier paradigma?
- Algunas ventajas de POO: Permite un mayor manejo de la complejidad, ya que está centrado a construir en base a componentes (objetos). ¿Cómo? Modelando los programas pensando en entidades y en cómo éstas interactúan a partir de su comportamiento.



# Un poco de historia de POO y sus motivaciones

- ¿Cuándo surge y dónde? POO surge en Noruega en 1967 a través de un lenguaje llamado Simula67, desarrollado por Krinsten Nygaard y Ole-Johan Dahl, en el centro de cálculo noruego. ¿Qué lenguajes de programación conocen?
- ¿Por qué surge? Surge como una respuesta ante la **complejidad** creciente en el mundo de la programación. POO no fue pensado en principio como un nuevo paradigma sino como en una nueva forma de organizar los programas para disminuir la complejidad que llevaba diseñarlos.
- Poco a poco, distintos pensadores y desarrolladores del mundo de la informática van introduciendo conceptos que van moldeando el paradigma. En los 90, a partir del lanzamiento de JAVA y el auge de C++, la POO se consolida en la industria del software.

# Trivia de lenguajes



# ¿Cómo modelamos nuestros programas a partir de POO?

- A partir de un problema, buscaremos **entidades del dominio** de éste. A diferencia de la programación estructurada, en la cual nos centrábamos en las distintas **funcionalidades** para ir construyendo el programa, ahora pensaremos entre las distintas entidades que interactúan entre sí para resolver el problema en cuestión.
- En la programación estructurada buscábamos las **acciones** (verbos), ahora buscaremos **componentes** (sustantivos).
- Una vez definidas las entidades, debemos determinar la forma en la que interactúan entre ellas y a partir de esto definir sus distintos **comportamientos**.

# Algunas definiciones claves

- **Objeto:** Es una entidad que puede recibir **mensajes**, responder a los mismos y enviar mensajes a otros objetos.
- **Mensaje:** Es la **interacción** entre quien pide un servicio y el objeto que lo brinda. (Cliente-receptor). Los objetos se envían mensajes entre sí en forma colaborativa a fin de resolver un determinado problema.
- **Comportamiento:** El comportamiento de un objeto son las **posibles respuestas** a los mensajes recibidos por un objeto se denominan comportamiento, es decir que es cómo **reacciona** un objeto cuando se le envía un mensajes.

# Composición de un objeto

- Una definición previa:

**Clase:** Es el tipo de un objeto. Podemos decir que una clase agrupa a todos los objetos que tienen el mismo comportamiento y, al escribir el código de la clase, estamos en definitiva determinando cómo van a ser los objetos de dicha clase. Decimos que un objeto es instancia de la clase a la que pertenece.

- **Estado del objeto:** Son las propiedades que posee. Pueden ser otros objetos, listas, diccionarios, variables numéricas, strings, etc. A estas propiedades se las llama ATRIBUTOS.
- **MÉTODOS del objeto:** Es la implementación de la respuesta de un objeto a los mensajes que debe recibir. Son similares a las funciones en la programación estructurada.

# Objetos en Python

- Primera noticia: todas las variables que venimos manejando hasta ahora son objetos. Como vimos, Python utiliza distintos tipos de datos.

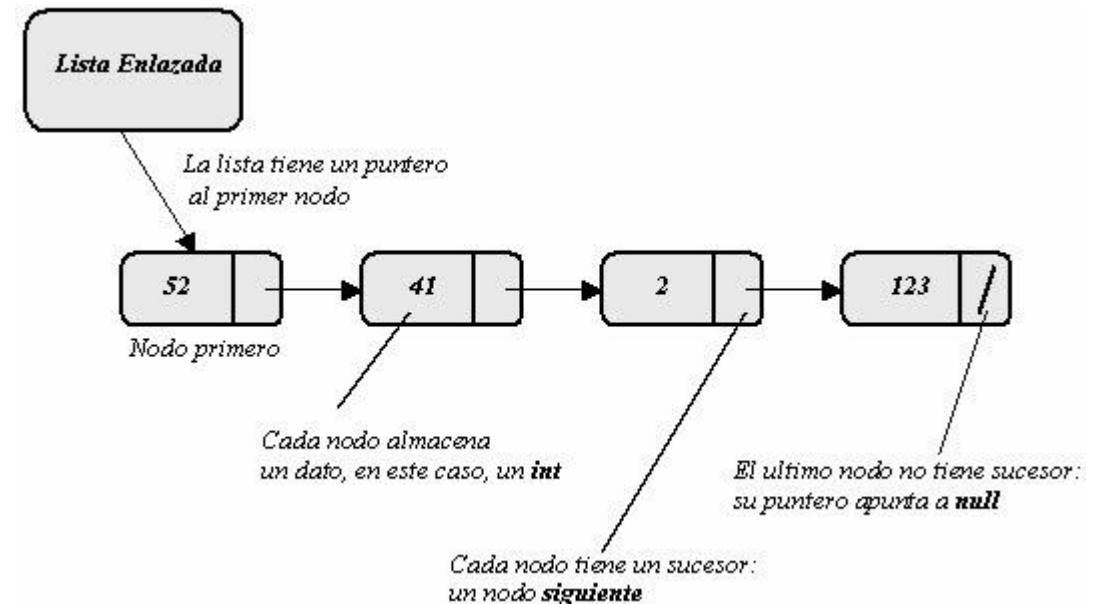
1234            3.14519            "hola"            [1,2,3,4,5]  
dic = {"Alumno": nombreAlumno, "Padron": numeroPadron }

- Cada uno de ellos es un objeto, y como ya hablamos cada uno tiene:
  - tipo** -> es la clase a la cual pertenece el objeto (i.e. es la clase de la cual es objeto es instancia).
  - acciones** (métodos) que nos permiten interactuar con el objeto. Las acciones que puede realizar un objeto definen su comportamiento. El comportamiento de un objeto puede estar definido a través de la firma de sus métodos, lo que entendemos como **interfaz**.
- Notemos que en Python **todo es un objeto**: todas las variables pertenecen a un tipo de dato que se corresponde con una clase.



# Un objeto que ya conocemos: la lista

- La lista es uno de los objetos que más se utilizan en Python. A diferencia de los objetos que programamos nosotros, la forma de instanciar una lista es distinta. `lista = []`
- ¿Cómo es la representación interna de las listas en Python? ¿Qué métodos tenemos para manipularla?



# Declarando clases y objetos

```
class Persona:  
    nombre = "Juan Perez"  
    edad = 22  
    altura = 1.80
```

```
def crecer_un_centrimetro(self):  
    self.altura += 0.01
```

```
persona = Persona()
```

```
persona.crecer_un_centrimetro()
```

¿Qué es esto?

El parámetro self hace referencia a la instancia a través de la cual se invocó al método. Para este ejemplo, hace referencia la instancia llamada "persona". Está presente en todo los métodos

Atributos (estado)


Método  
(comportamiento)

Instanciación

Invocación de un  
método



# Implementando nuestras primeras clases

- Ejercicio: La facultad nos solicita que implementemos una clase que modele a los distintos cursos que existen. El curso debe poder, a través de sus métodos: configurar un nombre, devolver la cantidad de alumnos, ingresar un alumno con su nota conjuntamente, preguntar si un alumno está en el curso y permitir preguntar por la nota de un alumno.
- 



# Un método importante: el constructor

- ¿Se imaginan algún método común que podría ser de utilidad?
- Un **constructor** es un método especial que sirve para **inicializar una instancia** de una clase. Al invocarlo, retorna una nueva referencia a una instancia de la clase en cuestión (una instancia).
- Un constructor puede recibir **parámetros**, y utilizarlos para determinar el **valor inicial de los atributos** (puede no inicializar todos) del objeto que se está instanciando. Así, es posible configurar el **estado inicial** con el cual el objeto comenzará a interactuar con otros en la ejecución del programa.



# Una distinción importante: variables de clase vs variables de instancia

- Las **variables de clase** son aquellas cuyo estado pertenece a la **clase del objeto**. Es decir, la variable pertenece a una clase en su totalidad, y no a una instancia en particular. Todas las instancias comparten una misma variable de clase. Aquí hay que tener en cuenta un concepto importante: como dijimos previamente, en Python TODO es un objeto, incluidas las clases. Estas variables se suelen inicializar **fuera del constructor**.
- Las variables de instancia son aquellas que forman parte del **estado de una instancia en particular**. Una variable de instancia posee un valor distinto para cada una de las instancias de un objeto de determinada clase. Estas variables pueden ser **inicializadas en el constructor** de la clase.
- Debemos ser precavidos. En general, no deberíamos usar variables de clase y su utilización debe estar bien justificada.

# Construyendo la clase en forma adecuada

```
class Persona():  
    def __init__(self, nombre, edad, altura):  
        self.nombre = nombre  
        self.edad = edad  
        self.altura = altura
```

Los atributos son para cada instancia en particular, y se inicializan en el **constructor**

```
def crecer_un_centrimetro(self):  
    self.altura += 0.01
```

Modifica únicamente la altura de la instancia para la cual se invoca el método

```
persona = Persona("Juan Perez", 22, 1.80)
```

El objeto se construye con parámetros

```
persona.crecer_un_centrimetro()
```

# ¿Qué sucede si tengo una variable de clase que se llama igual que una variable de instancia?

```
class Persona():
    altura = 1.50
    def __init__(self, nombre, edad, altura):
        self.nombre = nombre
        self.edad = edad
        self.altura = altura

    def crecer_un_centrimetro(self):
        self.altura += 0.01
```

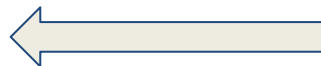
```
persona = Persona("Juan Perez", 22, 1.80)
```

```
persona.crecer_un_centrimetro()
```

```
persona.__dict__
Persona.__dict__
```

¿Cuánto vale la altura final?

- La altura final es 1.81. Primero, Python se fija si encuentra una variable de instancia con el nombre altura. En caso de no encontrarla, se fija si encuentra una variable de clase con dicho nombre.
- Python lleva, en cada objeto, un diccionario que contiene el valor de sus distintas variables. Si no se encuentra la variable en el diccionario de la instancia, se busca en el diccionario de la clase a la que pertenece.
- La clase tiene su propio diccionario ya que como vimos previamente también es un objeto.



## Retomando el ejercicio anterior

- Rehacer el ejercicio anterior, esta vez construyendo la clase con atributos de instancia y utilizando el método especial que define el constructor.
- Agregar las siguientes funcionalidades:
  - Preguntar la cantidad total de cursos registrados.
  - Permitir que para un usuario se ingrese más de una nota (puede utilizarse el mismo método).
  - Escribir un método que permita preguntar por el promedio de un usuario en el curso.



# Otros dos métodos predefinidos

- Además del constructor, hay otros dos métodos (en realidad, hay varios más) que se pueden sobrescribir y son típicamente utilizados:

- El primero de ellos es:

```
def __str__(self):
```

- Cuya función es devolver una cadena de caracteres representativa del objeto.
- El segundo es:

```
def __del__():
```

Cuya función es eliminar la referencia del objeto para el intérprete. A veces se lo suele llamar “destructor”.



# Una práctica común en el desarrollo con POO: las pruebas unitarias.

- Una prueba **unitaria** es, como su nombre se indica, una prueba que testea **una sola cosa**.
- El testing es algo sumamente común en el desarrollo de software: antes de lanzar una aplicación al mercado, un equipo se suele encargar de probarla en su totalidad.
- Las pruebas unitarias se centran en probar el **comportamiento** de los objetos: se llaman unitarias ya que, para cada una de las **funcionalidades** de mi clase (y eventualmente para cada **método**) se debería tener una prueba unitaria que pruebe su correcto funcionamiento.

# Ejemplo de escritura de pruebas unitarias

```
import unittest
from ejemplo import Persona
```

```
class TestPersona(unittest.TestCase):
```

```
    def
```

```
    test_creo_una_persona_y_crece_una_vez(self):
```

```
        persona = Persona("Juan", 22, 1.80)
```

```
        persona.crecer_un_centrimetro()
```

```
        self.assertEqual(persona.altura, 1.81)
```

```
if __name__ == '__main__':
    unittest.main()
```



Arrange



Act



Assert

Ejercicio: Escribir dos pruebas unitarias para el método que permite calcular el promedio de un alumno en un curso.

# Avanzando en algunos conceptos

- Hay algunos conceptos claves en los que se basa el paradigma: uno de ellos es el **encapsulamiento**.
- ¿Qué es el encapsulamiento? Es el **ocultamiento del estado de un objeto y la implementación que hace a su comportamiento**. Así, se logran mayores niveles de **abstracción** en el modelado de los programas y, además, se logra que los estímulos (mensajes) a los que responden los objetos **no dependan de la implementación** específica que haya elegido el programador de turno. Esto también permite diseñar pensando en tener una mayor **adaptabilidad a cambios** si el día de mañana hay que realizar alguna modificación en la implementación.
- ¿Qué debemos encapsular? Algo sumamente importante que debe ocultarse es el estado de los objetos: hablamos de **atributos privados**. En python, podemos hacer que un atributo sea privado agregando un modificador.