UTN AVELLANEDA
Tecnicatura Universitaria en Programación

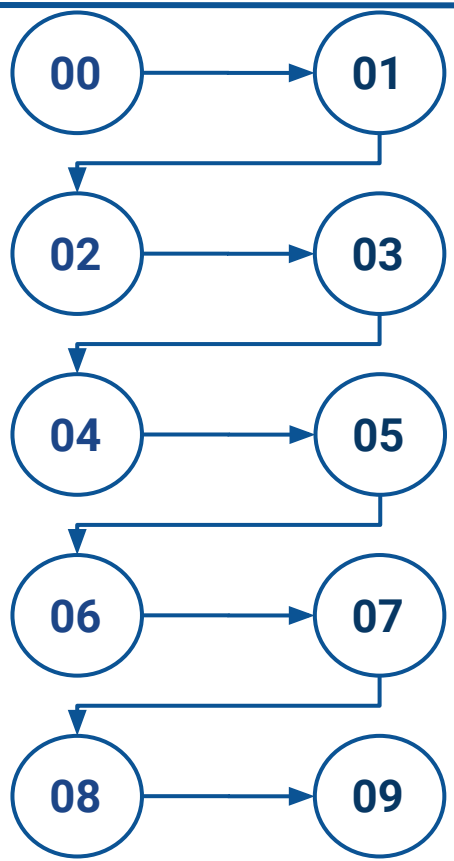
Programación orientado a objetos en Python

Programación y Laboratorio I

Versión ‘23

UTN AVELLANEDA
Tecnicatura Universitaria en Programación

Template



¿Qué es la POO?

¿Se puede evitar usar?

Definir una clase

Definir atributos

Atributos de clase

Crear una instancia

Metodos

Atributos protegidos

Property

Metodos dunder

UTN AVELLANEDA
Tecnicatura Universitaria en Programación

POO: Paradigma orientado a objetos vs. Programación orientada a objetos.

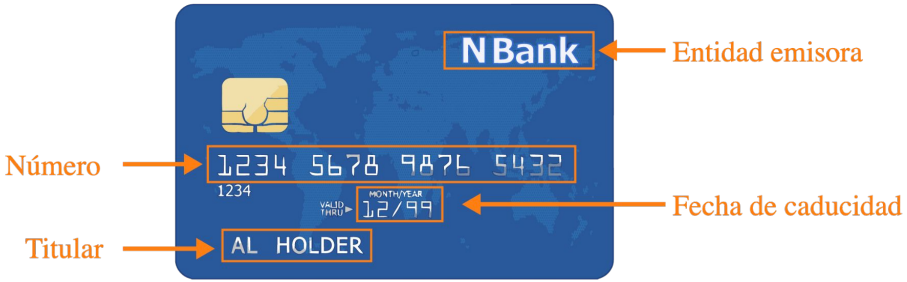
El paradigma orientado a objetos responde a una forma de encarar un proyecto de desarrollo de software. El mismo se basa en la definición de objetos que forman parte de un dominio en cuestión, que interactúan con otros objetos y que se comunican por medio de mensajes. Por lo tanto, la programación orientada a objetos define la forma de producir código bajo este paradigma.

¿Qué son los objetos?

Un objeto podría representar a una persona con características como nombre, edad y dirección y comportamientos como caminar, hablar, respirar y correr.

Ej: Objeto Tarjeta

Caracteristicas



Comportamientos

Activar

Pagar

Anular

¿Se puede evitar la POO?

Las estructuras de datos primitivas, como números, cadenas, diccionarios y listas, están diseñadas para representar información simple

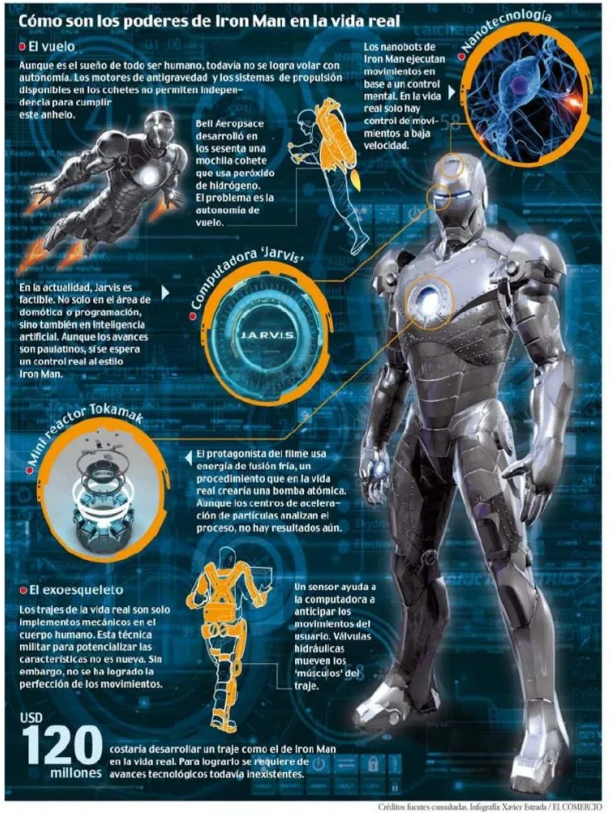
¿Se puede evitar la POO?

¿Qué pasa si se quiere representar algo más complejo?

UTN AVELLANEDA
Tecnicatura Universitaria en Programación

¿Se puede evitar la POO?

Supongamos que desea realizar una aplicación que permita administrar información de personajes de película.



¿Se puede evitar la POO?

Debiendo almacenar información básica sobre cada uno:

- `personaje_id`
- `personaje_nombre`
- `personaje_usa_nanotecnologia`
- `personaje_puede_volar`

¿Se puede evitar la POO?

¿Qué alternativas existen para representar la información?

Variables individuales

Una alternativa sería generar una variable distinta para cada campo del personaje

```
personaje_id = 1
personaje_nombre = "IronMan"
personaje_usa_nanotecnologia = "No"
Personaje_puede_volar =  "Si"
```

En el caso de tener más de un personaje esta técnica se vuelve inviable.

Lista

Otra alternativa sería generar una lista con los campos del personaje y acceder a estos por el índice.

```
lista_personajes = []
personaje =  [1,'IronMan','No', 'Si']
lista_personajes.append(personaje)
```

En el caso de tener más de un personaje implicaría tener una lista de personajes que tenga en cada posición una lista que representa a un único personaje..

Diccionario

Otra alternativa sería generar un diccionario con los campos del personaje y acceder a estos por clave.

```
lista_personajes = []
d_personaje = {'id'=1,
               'nombre'='IronMan',
               'usa_nanotecnologia'='No',
               'puede_volar'='Si'}
lista_personajes.append(d_personaje)
```

En el caso de tener más de un personaje implicaría tener una lista de personajes que en cada posición guarde un diccionario que represente a un único personaje.

Mejor opción: **dict**

De los tres casos anteriores, el de representar la información de cada personaje utilizando un diccionario sin lugar a dudas es la mejor opción.

Mejor opción: **dict**

Por cada acción que nuestro programa requiera realizar desarrollemos una función

Esta recibirá **siempre como parámetro el diccionario que representa al personaje** o en su defecto una lista de diccionarios.

```
personaje_mostrar(d_personaje)
```

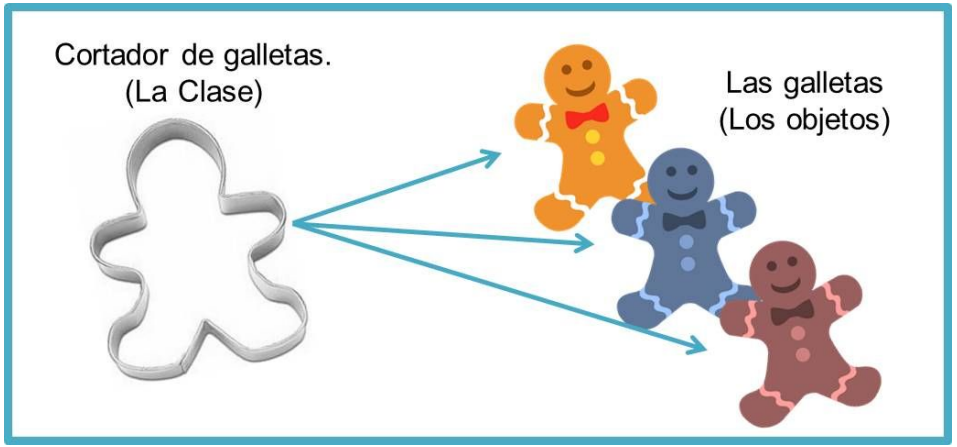
Usando POO

Para empezar a usar el POO, primero debemos abordar algunos conceptos.

Clases

¿Qué es una clase?

Podemos definir a una clase como un molde que **definirá** las características y acciones que tendrá un objeto.



Cómo definir una clase

Todas las definiciones de clase comienzan con la palabra clave **class**, seguida del nombre de la clase y dos puntos.

```
class Personaje:
  pass
```

Nota: los nombres de las clases de Python se escriben en notación **CapitalizedWords** por convención (UpperCamelCase).

Cómo definir atributos

A las características de un objeto las vamos a llamar propiedades o atributos. Estos atributos los tendrán todos los objetos de tipo **Personaje.** Los mismos se definirán en un método llamado __init__ **(Constructor de la clase).**

```
class Personaje:
    def __init__(self,id,nombre,nano,vuela) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self.usa_nanotecnologia = nano
        self.puede_volar = vuela
```

Cómo definir atributos de clase

Las propiedades/atributos de clase se definen antes del método __init__. Los atributos de clase son atributos que tienen el mismo valor para todas las instancias de clase.

```
class Personaje:
    tipo = "Personaje"
    def __init__(self,id,nombre,nano,vuela) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self.usa_nanotecnologia = nano
        self.puede_volar = vuela
```

Crear una instancia

La creación de un nuevo objeto a partir de una clase se denomina instanciación de un objeto (se crea el objeto en memoria.

Se puede instanciar un nuevo **Personaje** escribiendo el nombre de la clase, seguido de paréntesis de apertura y cierre:

```
personaje_A = Personaje(1,'IronMan','No', 'Si')
personaje_B = Personaje(2,'IronSpider','Si', 'No')
```

Acceder a los atributos

Se puede acceder a sus atributos de instancia mediante la notación de puntos :

```
personaje_A = Personaje(1,'IronMan','No', 'Si')
personaje_B = Personaje(2,'IronSpider','Si', 'No')

print(personaje_A.nombre) # IronMan
print(personaje_B.nombre) # IronSpider
```

Cómo definir métodos

Los métodos de instancia representan las acciones que pueden realizar los objetos. Son funciones que se definen dentro de una clase y solo se pueden llamar desde una instancia de esa clase. Al igual que .__init__(), el primer parámetro de un método de instancia siempre es self.

```
class Personaje:
    def __init__(self,id,nombre,nano,vuela) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        …
    def retornar_descripcion(self) -> str:
        return '{0}-{1}'.format(self.id,self.nombre)
```

Llamar a un método

En la clase Personaje, el método **retornar_descripcion** devuelve una cadena que contiene el nombre y el apellido del personaje.

```
personaje_A = Personaje(1'IronMan','No', 'Si')
personaje_B = Personaje(2,'IronSpider','Si', 'No')

print(personaje_A.retornar_descripcion())# IronMan
print(personaje_B.retornar_descripcion())# IronSpider
```

Atributos protegidos (“_”)

Un guión bajo antes del nombre indica que es un atributo protegido. Lo cual establece que solo puede ser accedido por esa clase y sus herederas.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self._nombre = nombre
```

Atributos privados (“__”)

Doble guión bajo antes del nombre indica que es un atributo privado. Lo cual establece que solo puede ser accedido por esa clase y sus herederas.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.__nombre = nombre
```

property

La función integrada property() nos permitirá interceptar la escritura, lectura, borrado de los atributos y además nos permitirá incorporar una documentación sobre los mismos.

@property

getters

Getter: Se encargará de interceptar la lectura del atributo.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.__nombre = nombre
        ...
    @property
    def nombre(self): #Getter
        return self.__nombre

personaje_A = Personaje(1,'IronMan','No', 'Si')
print(personaje_A.nombre)# IronMan
```

setters

Setter : Se encarga de interceptar cuando se escribe.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.__nombre = nombre
        ...
    @nombre.setter #Setter
    def nombre(self, nombre):
        self.__nombre = nombre

personaje_A = Personaje(1,'IronMan','No', 'Si')
personaje_A.nombre = 'IRONMAN'
```

Métodos dunder

UTN AVELLANEDA
Tecnicatura Universitaria en Programación

Métodos dunder

Los métodos **dunder** ("Double Under") o método mágicos son los métodos de una clase que tienen dos subrayados de prefijo y sufijo en el nombre.

Hay muchos y se pueden usar para personalizar el comportamiento de los objetos en Python

__str__

El segundo método más utilizado sea **__str__**, con el que se crea una representación del objeto que tenga significado para las personas

```
class Personaje:
    def __init__(self,id,nombre,nano,vuela) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        …
    def __str__(self) -> str:
        return '{0}-{1}'.format(self.nombre,self.usa_nanotecnologia)
```

__str__

Ahora al imprimir el objeto podemos ver algo que se relaciona con el contenido.

```
personaje_A = Personaje(1,'IronMan','No', 'Si')
personaje_B = Personaje(2,'IronSpider','Si', 'No')

print(personaje_A)# IronMan
print(personaje_B)# IronSpider
```

__len__

En el caso de que el objeto tenga una número de elementos, como es el caso, podríamos usar la función len para obtener este.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self._cantidad = 1
    def __len__(self) -> str:
        return self._cantidad
```

__getitem__

Si se desea que los usuarios puedan leer los elementos mediante el uso de corchetes es necesario implementar el método **__getitem__** en la clase.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self._lista= [id,nombre,apellido,edad]
    def __getitem__(self,index) -> str:
        return self._lista[index]

personaje_A = Personaje(0,'Marty','McFly', 18)
print(personaje_A[1]) #Marty
```

__setitem__

__setitem__ es el complemento del método anterior. En este caso es necesario pasar dos parámetros adicionales, la posición y el valor a reemplazar.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self._lista= [id,nombre,apellido,edad]
    def __setitem__(self,index, value):
        self._lista[index] = value

personaje_A = Personaje(0,'Marty','McFly', 18)
personaje_A[1] = 'MARTY'
print(personaje_A[1]) #MARTY
```

__contains__

__contains__ sobrecarga el operador **in** retornando un booleano.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self._lista= [id,nombre,apellido,edad]
    def __contains__(self,item):
        return item in self._lista

personaje_A = Personaje(0,'Marty','McFly', 18)
print('Marty' in personaje_A) # True
print('MARTY' in personaje_A) # False
```

__delitem__

__delitem__ sobrecarga el operador **del**

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self._lista= [id,nombre,apellido,edad]
    def __delitem__(self,index):
        return self._lista.pop(index)

personaje_A = Personaje(0,'Marty','McFly', 18)
del personaje_A[1]
print(personaje_A[1]) # McFly
```

__iter__

__iter__ permite que el objeto sea iterable, lo que habilita a usarlo por ejemplo en bucles tipo for.

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self._lista= [id,nombre,apellido,edad]
    def __iter__(self):
         for index in range(len(self._lista)):
             yield self._lista[index]

personaje_A = Personaje(0,'Marty','McFly', 18)

for i in personaje_A:
    print(i)
```

Operadores comparación

Existen varios métodos mágicos que permiten sobrecargar los operadores de comparación cómo puede ser:

Método	Operador	Método	Operador
__lt__	<	**__ne__**	!=
__le__	<=	**__gt__**	>
__eq__	==	**__ge__**	>=

Estos tienen un segundo parámetro que hace referencia al otro objeto con el que se opera. Siendo necesario devolver el resultado.

__le__

__lt__ permite sobrecargar el operador < (menor que).

```
class Personaje:
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self._lista= [id,nombre,apellido,edad]
    def __lt__(self,item):
      return self.edad < item.edad

personaje_A = Personaje(0,'Marty','McFly', 18)
personaje_B = Personaje(0,'Emmet','Brown', 54)
print(personaje_A < personaje_B) # True
```

YAPA

Heredar de otras clases

La herencia es el proceso por el cual una clase adquiere los atributos y métodos de otra.

Para heredar solo se requiere colocar el nombre de la clase padre entre paréntesis.

```
class Personaje (Persona):
    def __init__(self,id,nombre,apellido,edad) -> None:
        self.id = id # crea un atributo id y le asigna el valor id.
        self.nombre = nombre
        self._lista= [id,nombre,apellido,edad]
```