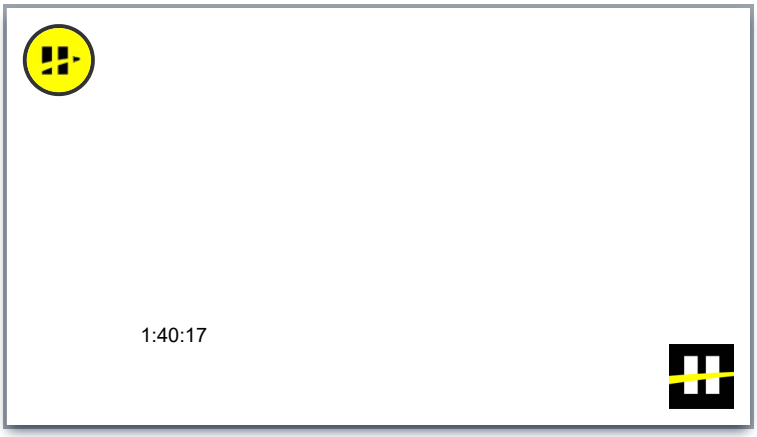
Tiempo de lectura
29 min

Contenido de la clase

Grabación de la Clase 7



Principales Objetivos de Aprendizaje para esta Clase

- Comprender el concepto de Porgramación Orientada a Objetos
- Comprender los conceptos de Clases y Objetos
- Conocer el concepto de Herencia
- Comprender los conceptos de Librerías y Módulos

Clases y objetos (POO)

Dejanos tu feedback! 👍

Contenido de la clase

Dejanos tu feedback! 👍

Orientación a Objetos (POO) brinda estos métodos de experimentación, y logra que los lenguajes sean de más alto nivel, es decir, más cercanos a como los humanos pensamos el mundo. Los seres humanos, vemos la realidad como objetos que se interrelacionan y realizan acciones, y esto, es lo que se intenta emular en la POO.
Hasta 1966 la programación fue exclusivamente lineal, hasta que surgieron lenguajes como Simula, SmallTalk, C++, Ada, Delphi o Java. Hoy el lenguaje más popular es Python y su filosofía hace hincapié en la legibilidad de su código.

Objeto

Una estructura de datos que eventualmente tiene funciones asociadas, y que están agrupados por razones de consistencia y comodidad conforman un **objeto**.
En la composición de un objeto tenemos entonces **propiedades** (datos) y **métodos** (funciones asociadas).

Clase

Hay una diferencia muy importante entre un objeto y una variable, y es que mientras que la variable 'se crea', el objeto 'se instancia'. Lo que implica que su creación se realiza en base a una definición preliminar, disponibilizando en la memoria, no solo la estructura de datos asociada sino sus métodos. Por medio de esta mecánica, además, se puede instanciar más de un solo objeto con la misma definicion. Esta definición, es una generalización del objeto, es decir, que especifica que estructura de datos va a tener y qué métodos asociados. Esto lo que se conoce como **clase**.

Pilares de la Programación Orientada a Objetos

- **Abstracción: es cuando separamos los datos de un objeto para luego generar un molde (una clase).**
- **Encapsulamiento: se utiliza cuando es necesario que ciertos métodos o propiedades sean inviolables o inalterables.**
  Un ejemplo del encapsulamiento podría ser

métodos previamente validados para aumentar dicho balance (depósitos, transferencias, etc).

- Herencia: permite crear nuevas clases a partir de otras. Si tuviéramos una clase "Autos" y quisiéramos crear unas clases "Auto deportivo" o "Auto clásico", podríamos tomar varias propiedades y métodos de la clase "Autos". Esto nos da una jerarquía de padre e hijo.
- Polimorfismo: proviene de poli = muchas, morfismo = formas. Se utiliza para crear métodos con el mismo nombre pero con diferente comportamiento.

Ejemplos:

- Clase Animal

  - Especie

  - Edad

  - Color

  - Correr()

  - Dormir()

Se instancian distintos objetos a partir de una clase:

- Objeto1

  - Especie: 'Perro'

  - Edad: 3

  - Color: 'Blanco'

  - Correr()

  - Dormir()

Contenido de la clase

Dejanos tu feedback! 👍

Contenido de la clase

- Edad: 8

- Color: 'Marrón'

- Correr()

- Dormir()

- **A partir de la sentencia class y el nombre de creamos la clase.**
- **La función init() es el constructor de la clase, esta función se ejecuta cuando se instancia el objeto.**
- **La clase posee atributos (especie, edad, color) y métodos que manipulan esos atributos (mePresento, cumplirAños).**

```
>>> class Animal:
>>> '''
>>> En esta clase se crean los animales
>>> '''
>>> def __init__(self, especie, edad, color):
>>>     self.especie = especie
>>>     self.edad = edad
>>>     self.color = color
>>> def mePresento(self):
>>>     print('Hola, soy ', self.especie, ', de
>>> def cumplirAños(self):
>>>     self.edad = self.edad + 1
>>>
>>> a1 = Animal('Ratón', 2, 'Marrón')
>>> print(a1.especie)
Ratón
>>> print(a1.edad)
2
>>> a2 = Animal('Liebre', 3, 'Gris')
>>> print(a2.especie)
Liebre
>>> print(a2.edad)
3
```

Dejanos tu feedback! 👍

Contenido de la clase

Dejanos tu feedback! 👍

Este formato de clases, objetos, métodos y parámetros es muy común en Python y lo utilizamos cada vez que invocamos alguna de sus librerías

```
>>> a1.mePresento()
Hola, soy  Ratón , de color Marrón  y tengo  2
>>> a2.mePresento()
Hola, soy  Liebre , de color Gris  y tengo  3
>>> a1.cumplirAños()
>>> a1.mePresento()
Hola, soy  Ratón , de color Marrón  y tengo  3
```

Herencia

Si existe una clase como versión especializada de una ya existente, se puede implementar una jerarquía de clases y así, compartir comportamiento y atributos de una clase “padre” ó superclase a una clase “hijo” ó subclase.

Cuando una clase hereda de otra, adquiere atributos y métodos. Además de ello, una clase hija puede sobreescribirlos, o incluso definir unos nuevos.
En el siguiente ejemplo vemos como se puede usar la herencia en Python, con la clase Perro que hereda de Animal:

```
>>> class Animal:
>>>     def __init__(self, especie, edad):
>>>         self.especie = especie
>>>         self.edad = edad
>>>
>>>     # Método genérico pero con implementaci
>>>     def hablar(self):
>>>         # Método vacío
>>>         pass
>>>
>>>     # Método genérico pero con implementaci
>>>     def moverse(self):
>>>         # Método vacío
>>>         pass
```

Contenido de la clase


```
>>>         print("Soy un Animal del tipo", typ
>>>
>>>
>>> class Perro(Animal):
>>>     def hablar(self):
>>>         print("Guau!")
>>>     def moverse(self):
>>>         print("Caminando con 4 patas")
>>>
>>> class Vaca(Animal):
>>>     def hablar(self):
>>>         print("Muuu!")
>>>     def moverse(self):
>>>         print("Caminando con 4 patas")
>>>
>>> class Abeja(Animal):
>>>     def hablar(self):
>>>         print("Bzzzz!")
>>>     def moverse(self):
>>>         print("Volando")
>>>
>>>     # Nuevo método
>>>     def picar(self):
>>>         print("Picar!")
```

¿Y para que queremos la herencia? Dado que una clase hija hereda los atributos y métodos de la padre, nos puede ser muy útil cuando tengamos clases que se parecen entre sí pero tienen ciertas particularidades. En este caso en vez de definir un montón de clases para cada animal, podemos tomar los elementos comunes y crear una clase Animal de la que hereden el resto, respetando por tanto la filosofía DRY. Realizar estas abstracciones y buscar el denominador común para definir una clase de la que hereden las demás, es una tarea de lo más compleja en el mundo de la programación.

Para saber más: El principio DRY (Don't Repeat Yourself) es muy aplicado en el mundo de la programación y consiste en no repetir código de manera innecesaria. Cuanto más código duplicado exista, más difícil será de modificar y más fácil será crear inconsistencias. Las clases y la herencia a no repetir código.

Dejanos tu feedback! 👍

Contenido de la clase

Dejanos tu feedback! 👍

Continuemos con nuestro ejemplo de perros y animales. Vamos a definir una clase padre Animal que tendrá todos los atributos y métodos genéricos que los animales pueden tener. Esta tarea de buscar el denominador común es muy importante en programación. Veamos los atributos:

Tenemos la especie ya que todos los animales pertenecen a una.
Y la edad, ya que todo ser vivo nace, crece, se reproduce y muere.
Y los métodos o funcionalidades:

Tendremos el método hablar, que cada animal implementará de una forma. Los perros ladran, las abejas zumban y los caballos relinchan. Un método moverse. Unos animales lo harán caminando, otros volando. Y por último un método describeme que será común.
Definimos la clase padre, con una serie de atributos comunes para todos los animales como hemos indicado.

```
>>> class Animal:
>>>     def __init__(self, especie, edad):
>>>         self.especie = especie
>>>         self.edad = edad
>>>
>>>     # Método genérico pero con implementaci
>>>     def hablar(self):
>>>         # Método vacío
>>>         pass
>>>
>>>     # Método genérico pero con implementaci
>>>     def moverse(self):
>>>         # Método vacío
>>>         pass
>>>
>>>     # Método genérico con la misma implemer
>>>     def describeme(self):
>>>         print("Soy un Animal del tipo", typ
```

Tenemos ya por lo tanto una clase genérica Animal, que generaliza las características y funcionalidades que todo animal puede tener. Ahora creamos una clase Perro que hereda del Animal. Como primer ejemplo vamos a crear una clase vacía, para ver

Contenido de la clase


```
>>> # Perro hereda de Animal
>>> class Perro(Animal):
>>>     pass
>>>
>>> mi_perro = Perro('mamífero', 10)
>>> mi_perro.describeme()
>>> # Soy un Animal del tipo Perro
```

Con tan solo un par de líneas de código, hemos creado una clase nueva que tiene todo el contenido que la clase padre tiene, pero aquí viene lo que es de verdad interesante. Vamos a crear varios animales concretos y sobreescrbir algunos de los métodos que habían sido definidos en la clase Animal, como el hablar o el moverse, ya que cada animal se comporta de una manera distinta.

Podemos incluso crear nuevos métodos que se añadirán a los ya heredados, como en el caso de la Abeja con picar().

```
>>> class Perro(Animal):
>>>     def hablar(self):
>>>         print("Guau!")
>>>     def moverse(self):
>>>         print("Caminando con 4 patas")
>>>
>>> class Vaca(Animal):
>>>     def hablar(self):
>>>         print("Muuu!")
>>>     def moverse(self):
>>>         print("Caminando con 4 patas")
>>>
>>> class Abeja(Animal):
>>>     def hablar(self):
>>>         print("Bzzzz!")
>>>     def moverse(self):
>>>         print("Volando")
>>>
>>>     # Nuevo método
>>>     def picar(self):
>>>         print("Picar!")
```

Dejanos tu feedback! 👍

Contenido de la clase

Heredados directamente de la clase padre: describeme()
Heredados de la clase padre pero modificados: hablar() y moverse()
Creados en la clase hija por lo tanto no existentes en la clase padre: picar()

```
>>> mi_perro = Perro('mamífero', 10)
>>> mi_vaca = Vaca('mamífero', 23)
>>> mi_abeja = Abeja('insecto', 1)
>>>
>>> mi_perro.hablar()
>>> mi_vaca.hablar()
>>> # Guau!
>>> # Muuu!
>>>
>>> mi_vaca.describeme()
>>> mi_abeja.describeme()
>>> # Soy un Animal del tipo Vaca
>>> # Soy un Animal del tipo Abeja
>>>
>>> mi_abeja.picar()
>>> # Picar!
```

Uso de super()

La función super() nos permite acceder a los métodos de la clase padre desde una de sus hijas. Volvamos al ejemplo de Animal y Perro.

```
>>> class Animal:
>>>     def __init__(self, especie, edad):
>>>         self.especie = especie
>>>         self.edad = edad
>>>     def hablar(self):
>>>         pass
>>>
>>>     def moverse(self):
>>>         pass
>>>
>>>     def describeme(self):
>>>         print("Soy un Animal del tipo", typ
```

Dejanos tu feedback! 👍

Contenido de la clase

Podemos crear un nuevo **init** y guardar todas las variables una a una.
O podemos usar super() para llamar al **init** de la clase padre que ya aceptaba la especie y edad, y sólo asignar la variable nueva manualmente.

```
>>> class Perro(Animal):
>>>     def __init__(self, especie, edad, dueñ
>>>         # Alternativa 1
>>>         # self.especie = especie
>>>         # self.edad = edad
>>>         # self.dueño = dueño
>>>
>>>         # Alternativa 2
>>>         super().__init__(especie, edad)
>>>         self.dueño = dueño
>>> mi_perro = Perro('mamífero', 7, 'Luis')
>>> mi_perro.especie
>>> mi_perro.edad
>>> mi_perro..dueño
```

Librerías

Las librerías son proyectos con métodos o funciones puntuales, el cual es posible anexar a otros proyectos y complementarlo usando sus métodos específicos para una determinada solución. Son trozos de código hechos por terceros. Facilita mucho la programación y hace que nuestro programa sea más sencillo de hacer y luego de entender. También llamadas 'Frameworks', consiste en archivos de código a los que se invoca al principio de nuestro propio código.

Módulos

Un módulo en Python es un archivo con extensión ".py" que alberga un conjunto de funciones, variables o clases y que puede ser usado por otros módulos. Nos permiten reutilizar código y organizarlo mejor en namespaces. Por ejemplo,

Dejanos tu feedback! 👍

Contenido de la clase


```
>>> # mimodulo.py
>>> def suma(a, b):
>>>     return a + b
>>>
>>> def resta(a, b):
>>>     return a - b
```

Una vez definido, dicho módulo puede ser usado o importado en otro código usando **import**, con lo que se puede acceder a todo el contenido.

```
>>> # otromodulo.py
>>> import mimodulo
>>>
>>> print(mimodulo.suma(4, 3))
7
>>> print(mimodulo.resta(10, 9))
1
```

También podemos importar únicamente los componentes que nos interesen como mostramos a continuación.

```
>>> from mimodulo import suma, resta
>>>
>>> print(suma(4, 3))   # 7
>>> print(resta(10, 9)) # 1
```

Por último, podemos importar todo el módulo haciendo uso de , *sin necesidad de usar mimodulo.*.

```
>>> from mimodulo import *
>>>
>>> print(suma(4, 3))
7
>>> print(resta(10, 9))
1
```

Dejanos tu feedback! 👍

pueda ser utilizado en diferentes entornos.

- **Desarrollo Web: existen frameworks como Django, Pyramid, Flask o Bottle que permiten desarrollar páginas web a todos los niveles.**
- **Ciencia y Educación: debido a su sintaxis tan sencilla, es una herramienta perfecta para enseñar conceptos de programación a todos los niveles. En lo relativo a ciencia y cálculo numérico, existen gran cantidad de librerías como SciPy o Pandas.**
- **Desarrollo de Interfaces Gráficos: gran cantidad de los programas que utilizamos tienen un interfaz gráfico que facilita su uso. Python también puede ser usado para desarrollar GUIs con librerías como Kivy o pyqt.**
- **Desarrollo Software: también es usado como soporte para desarrolladores, como para testing.**
- **Machine Learning: en los últimos años ha crecido el número de implementaciones en Python de librerías de aprendizaje automático como Keras, TensorFlow, PyTorch o sklearn.**
- **Visualización de Datos: existen varias librerías muy usadas para mostrar datos en gráficas, como matplotlib, seaborn o plotly.**
- **Finanzas y Trading: gracias a librerías como QuantLib o qtpylib y a su facilidad de uso, es cada vez más usado en estos sectores.**

Contenido de la clase

Rutas y Uso de sys.path

Normalmente los módulos que importamos están en la misma carpeta, pero es posible acceder también a módulos ubicados en una subcarpeta. Imaginemos la siguiente estructura:

```
.
├── ejemplo.py
├── carpeta
│   └── modulo.py
```

Dejanos tu feedback! 👍

Contenido de la clase

Dejanos tu feedback! 👍

```
>>> # modulo.py
>>> def hola():
>>>     print("Hola")
```

Desde nuestro **ejemplo.py**, podemos importar el módulo **modulo.py** de la siguiente manera:

```
>>> from carpeta.modulo import *
>>> print(hola())
Hola
```

Es importante notar que Python busca los módulos en las rutas indicadas por el **sys.path**. Es decir, cuando se importa un módulo, lo intenta buscar en dichas carpetas. Puedes ver tu sys.path de la siguiente manera:

```
>>> import sys
>>> print(sys.path)
```

Como es obvio, verás que la carpeta de tu proyecta está incluida, pero ¿y si queremos importar un módulo en una ubicación distinta? Pues bien, podemos añadir al sys.path la ruta en la que queremos que Python busque.

```
>>> import sys
>>> sys.path.append(r'/ruta/de/tu/modulo')
```

Una vez realizado esto, los módulos contenidos en dicha carpeta podrán ser importados sin problema como hemos visto anteriormente.

Por otro lado, es posible cambiar el nombre del módulo usando **as**. Imaginemos que tenemos un módulo **moduloconnombrelargo.py**.

```
>>> # moduloconnombrelargo.py
>>> hola = "hola"
```

Contenido de la clase

Dejanos tu feedback! 👍

```
>>> import moduloconnombrelargo
>>> print(moduloconnombrelargo.hola)
```

Podemos hacerlo de la siguiente manera con as:

```
>>> import moduloconnombrelargo as m
>>> print(m.hola)
```

La función **dir()** nos permite ver los nombres (variables, funciones, clases, etc) existentes en nuestro namespace. Si probamos en un módulo vacío, podemos ver como tenemos varios nombres rodeados de __. Se trata de nombres que Python crea por debajo.

```
>>> print(dir())
['__annotations__', '__builtins__', '__cached__
```

Por ejemplo, **file** es creado automáticamente y alberga el nombre del archivo .py.

```
>>> print(__file__)
/tu/ruta/tufichero.py
```

Imaginemos ahora que tenemos alguna variable y función definida en nuestro script. Como era de esperar, dir() ahora nos muestra también los nuevos nombres que hemos creado, y que por supuesto pueden ser usados.

```
>>> mi_variable = "Python"
>>> def mi_funcion():
>>>     pass
>>> print(dir())
['__annotations__', '__builtins__', '__cached__
```

Contenido de la clase


```
>>> from mimodulo import *
>>> print(dir())
['__annotations__', '__builtins__', '__cached__
```

El uso de dir() también acepta parámetros de entrada, por lo que podemos por ejemplo pasar nuestro módulo y nos dará más información sobre lo que contiene.

```
>>> import mimodulo
>>> print(dir(mimodulo))
['__builtins__', '__cached__', '__doc__', '__f
>>> print(mimodulo.__name__)
mimodulo
>>> print(mimodulo.__file__)
/tu/ruta/mimodulo.py
```

Importar un módulo puede lanzar la excepción "ImportError", cuando se intenta importar un módulo que no ha sido encontrado. Se trata de ModuleNotFoundError.

```
>>> import moduloquenoexiste
ModuleNotFoundError: No module named 'moduloque
```

Dicha excepción puede ser capturada para evitar la interrupción del programa.
Un problema muy recurrente es cuando creamos un módulo con una función como en el siguiente ejemplo, y añadimos algunas sentencias a ejecutar.

```
>>> # modulo.py
>>> def suma(a, b):
>>>     return a + b
```

Dejanos tu feedback! 👍

Contenido de la clase

Si en otro módulo importamos nuestro **modulo.py**, tal como está nuestro código el contenido se ejecutará, y esto puede no ser lo que queramos.

```
>>> # otromodulo.py
>>> import modulo
La suma es: 3
```

Dependiendo de la situación, puede ser importante especificar que únicamente queremos que se ejecute el código si el módulo es el **main**. Con la siguiente modificación, si hacemos import modulo desde otro módulo, este fragmento ya no se ejecutará al ser el módulo main otro.

```
>>> # modulo.py
>>> def suma(a, b):
>>>     return a + b
>>> if (__name__ == '__main__'):
>>>     c = suma(1, 2)
>>>     print("La suma es:", c)
```

Es importante notar que los módulos solamente son cargados una vez. Es decir, no importa el número de veces que llamemos a import mimodulo, que sólo se importará una vez.
Si queremos que el módulo sea recargado, tenemos que ser explícitos, haciendo uso de reload.

```
>>> import mimodulo
>>> import importlib
>>> importlib.reload(mimodulo)
>>> importlib.reload(mimodulo)
```

Dejanos tu feedback! 👍

Homework

Contenido de la clase

Completa la tarea descrita en el archivo **README**

Si tienes dudas sobre este tema, puedes consultarlas en el canal #python de Slack

Clase en vivo de Resolución de ejercicios.

Los martes/jueves (segun corresponda) a las 18 hs. ARG cada dos semanas hacemos una clase de apoyo en vivo sobre este tema. El link se comparte en el canal de Slack #anuncios ese día.

Hecho con 💛 por alumnos de Henry

Dejanos tu feedback! 👍