# Modulo 3 (intermedio): Los conceptos básicos del enfoque orientado a objetos

Demos un paso fuera de la programación y las computadoras, y analicemos temas de programación orientada a objetos.

Casi todos los programas y técnicas que has utilizado hasta ahora pertenecen al estilo de programación procedimental. Es cierto que has utilizado algunos objetos incorporados, pero cuando nos referimos a ellos, se mencionan lo mínimo posible.

La programación procedimental fue el enfoque dominante para el desarrollo de software durante décadas de TI, y todavía se usa en la actualidad. Además, no va a desaparecer en el futuro, ya que funciona muy bien para proyectos específicos (en general, no muy complejos y no grandes, pero existen muchas excepciones a esa regla).

El enfoque orientado a objetos es bastante joven (mucho más joven que el enfoque procedimental) y es particularmente útil cuando se aplica a proyectos grandes y complejos llevados a cabo por grandes equipos formados por muchos desarrolladores.

Este tipo de programación en un proyecto facilita muchas tareas importantes, por ejemplo, dividir el proyecto en partes pequeñas e independientes y el desarrollo independiente de diferentes elementos del proyecto.

**Python es una herramienta universal para la programación procedimental y orientada a objetos**. Se puede utilizar con éxito en ambos enfoques.

Además, puedes crear muchas aplicaciones útiles, incluso si no se sabe nada sobre clases y objetos, pero debes tener en cuenta que algunos de los problemas (por ejemplo, el manejo de la interfaz gráfica de usuario) puede requerir un enfoque estricto de objetos.

Afortunadamente, la programación orientada a objetos es relativamente simple.


## Enfoque procedimental frente al enfoque orientado a objetos

En el **enfoque procedimental**, es posible distinguir dos mundos diferentes y completamente separados: **el mundo de los datos y el mundo del código**. El mundo de los datos está poblado con variables de diferentes tipos, mientras que el mundo del código está habitado por códigos agrupados en módulos y funciones.

Las funciones pueden usar datos, pero no al revés. Además, las funciones pueden abusar de los datos, es decir, usar el valor de manera no autorizada (por ejemplo, cuando la función seno recibe el saldo de una cuenta bancaria como parámetro).

Los datos no pueden usar funciones. ¿Pero es esto completamente cierto? ¿Hay algunos tipos especiales de datos que puedan usar funciones?

Sí, los hay, los llamados métodos. Estas son funciones que se invocan desde dentro de los datos, no junto con ellos. Si puedes ver esta distinción, has dado el primer paso en la programación de objetos.

El **enfoque orientado a objetos** sugiere una forma de pensar completamente diferente. Los datos y el código están encapsulados juntos en el mismo mundo, divididos en clases.

Cada **clase es como una receta que se puede usar cuando quieres crear un objeto útil**. Puedes producir tantos objetos como necesites para resolver tu problema.

Cada objeto tiene un conjunto de rasgos (se denominan propiedades o atributos; usaremos ambas palabras como sinónimos) y es capaz de realizar un conjunto de actividades (que se denominan métodos).

Las recetas pueden modificarse si son inadecuadas para fines específicos y, en efecto, pueden crearse nuevas clases. Estas nuevas clases heredan propiedades y métodos de los originales, y generalmente agregan algunos nuevos, creando nuevas herramientas más específicas.

**Los objetos son encarnaciones** de las ideas expresadas en clases, como un pastel de queso en tu plato, es una encarnación de la idea expresada en una receta impresa en un viejo libro de cocina.

Los objetos interactúan entre sí, intercambian datos o activan sus métodos. Una clase construida adecuadamente (y, por lo tanto, sus objetos) puede proteger los datos sensibles y ocultarlos de modificaciones no autorizadas.

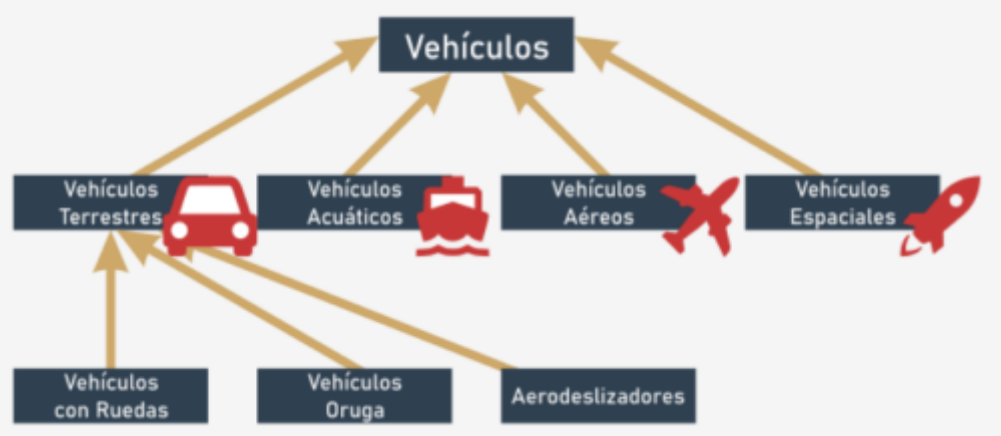
No existe un límite claro entre los datos y el código: viven como uno solo dentro de los objetos.

Todos estos conceptos no son tan abstractos como pudieras pensar al principio. Por el contrario, todos están tomados de experiencias de la vida real y, por lo tanto, son extremadamente útiles en la programación de computadoras: no crean vida artificial **reflejan hechos reales, relaciones y circunstancias**.

# Jerarquías de clase

La palabra *clases* tiene muchos significados, pero no todos son compatibles con las ideas que queremos discutir aquí. La clase que nos concierne es como una categoría, como resultado de similitudes definidas con precisión.

Intentaremos señalar algunas clases que son buenos ejemplos de este concepto.




Veamos por un momento los vehículos. Todos los vehículos existentes (y los que aún no existen) están **relacionados por una sola característica importante**: la capacidad de moverse. Puedes argumentar que un perro también se mueve; ¿Es un perro un vehículo? No lo es. Tenemos que mejorar la definición, es decir, enriquecerla con otros criterios, distinguir los vehículos de otros seres y crear una conexión más fuerte. Consideremos las siguientes circunstancias: los vehículos son entidades creadas artificialmente que se utilizan para el transporte, movidos por fuerzas de la naturaleza y dirigidos (conducidos) por humanos.

Según esta definición, un perro no es un vehículo.

La clase Vehículos es muy amplia. Tenemos que definir clases especializadas. Las clases especializadas son las subclases. La clase Vehículos será una superclase para todas ellas.

Nota: **la jerarquía crece de arriba hacia abajo, como raíces de árboles, no ramas**. La clase más general y más amplia siempre está en la parte superior (la superclase) mientras que sus descendientes se encuentran abajo (las subclases).

A estas alturas, probablemente puedas señalar algunas subclases potenciales para la superclase *Vehículos*. Hay muchas clasificaciones posibles. Elegimos subclases basadas en el medio ambiente y decimos que hay (al menos) cuatro subclases:

- Vehículos Terrestres.
- Vehículos Acuáticos.
- Vehículos Aéreos.
- Vehículos Espaciales.

En este ejemplo, discutiremos solo la primera subclase: Vehículos Terrestres. Si lo deseas, puedes continuar con las clases restantes.

Los vehículos terrestres pueden dividirse aún más, según el método con el que impactan el suelo. Entonces, podemos enumerar:

- Vehículos con ruedas.
- Vehículos oruga.
- Aerodeslizadores.

La figura ilustra la jerarquía que hemos creado.

Ten en cuenta la dirección de las flechas: siempre apuntan a la superclase. La clase de nivel superior es una excepción: no tiene su propia superclase.

Otro ejemplo es la jerarquía del reino taxonómico de los animales.

Podemos decir que todos los Animales (nuestra clase de nivel superior) se puede dividir en cinco subclases:

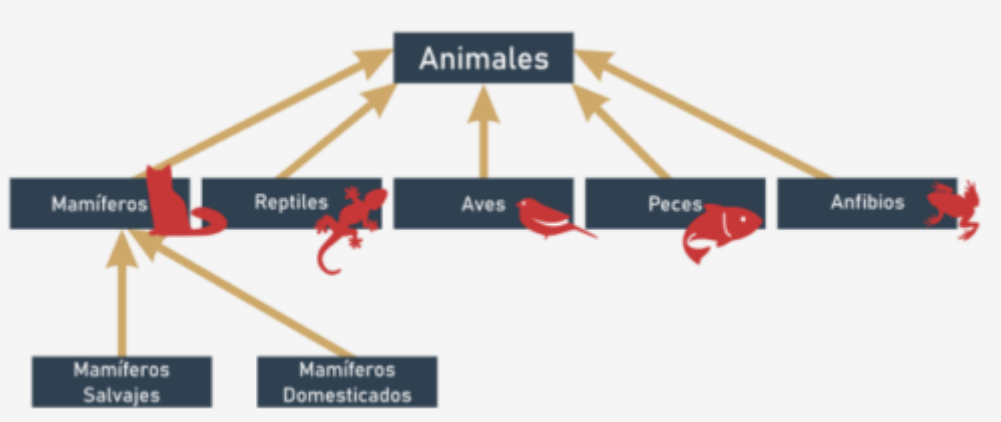
- Mamíferos.

- Reptiles.
- Aves.
- Peces.
- Anfibios.

Tomaremos el primero para un análisis más detallado.

Hemos identificado las siguientes subclases:

- Mamíferos Salvajes.
- Mamíferos Domesticados.




Intenta extender la jerarquía de la forma que quieras y encuentra el lugar adecuado para los humanos.

## ¿Qué es un objeto?

Una clase (entre otras definiciones) es un **conjunto de objetos**. Un objeto es **un ser perteneciente a una clase**.

Un objeto es **una encarnación de los requisitos, rasgos y cualidades asignados a una clase específica**. Esto puede sonar simple, pero ten en cuenta las siguientes circunstancias importantes. Las clases forman una jerarquía.

Esto puede significar que un objeto que pertenece a una clase específica pertenece a todas las superclases al mismo tiempo. También puede significar que cualquier objeto perteneciente a una superclase puede no pertenecer a ninguna de sus subclases.

Por ejemplo: cualquier automóvil personal es un objeto que pertenece a la clase Vehículos Terrestres. También significa que el mismo automóvil pertenece a todas las superclases de su clase local; por lo tanto, también es miembro de la clase Vehículos.

Tu perro (o tu gato) es un objeto incluido en la clase Mamíferos Domesticados, lo que significa explícitamente que también está incluido en la clase Animales.

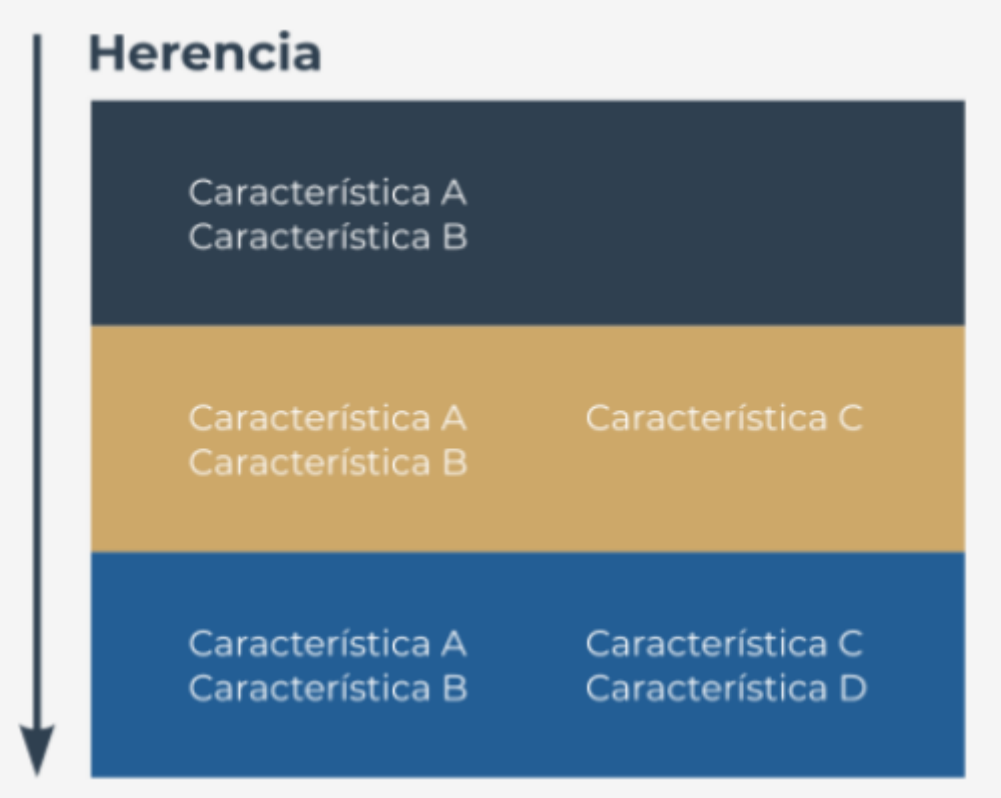
Cada **subclase es más especializada** (o más específica) que su superclase. Por el contrario, cada **superclase es más general** (más abstracta) que cualquiera de sus subclases.

Ten en cuenta que hemos supuesto que una clase solo puede tener una superclase; esto no siempre es cierto, pero discutiremos este tema más adelante.

# Herencia

Definamos uno de los conceptos fundamentales de la programación de objetos, llamado **herencia**. Cualquier objeto vinculado a un nivel específico de una jerarquía de clases **hereda todos los rasgos (así como los requisitos y cualidades) definidos dentro de cualquiera de las superclases**.

La clase de inicio del objeto puede definir nuevos rasgos (así como requisitos y cualidades) que serán heredados por cualquiera de sus superclases.




No deberías tener ningún problema para hacer coincidir esta regla con ejemplos específicos, ya sea que se aplique a animales o vehículos.

# ¿Qué contiene un objeto?

La programación orientada a objetos supone que **cada objeto existente puede estar equipado con tres grupos de atributos**:

- Un objeto tiene un **nombre** que lo identifica de forma exclusiva dentro de su namespace (aunque también puede haber algunos objetos anónimos).
- Un objeto tiene un **conjunto de propiedades individuales** que lo hacen original, único o sobresaliente (aunque es posible que algunos objetos no tengan propiedades).
- Un objeto tiene un **conjunto de habilidades para realizar actividades específicas**, capaz de cambiar el objeto en sí, o algunos de los otros objetos.

Existe una pista (aunque esto no siempre funciona) que te puede ayudar a identificar cualquiera de las tres esferas anteriores. Cada vez que se describe un objeto y se usa:

- Un sustantivo: probablemente se está definiendo el nombre del objeto.
- Un adjetivo: probablemente se está definiendo una propiedad del objeto.
- Un verbo: probablemente se está definiendo una actividad del objeto.

Dos ejemplos deberían servir como un buen ejemplo:

- Un Cadillac rosa pasó rápidamente.
  - Nombre del objeto = Cadillac
  - Clase = Vehículos con ruedas
  - Propiedad = Color (rosa)
  - Actividad = Pasar (rápidamente)
- Max es un gato grande que duerme todo el día.
  - Nombre del objeto = Max
  - Clase = Gato
  - Propiedad = Tamaño (Grande)
  - Actividad = Dormir (Todo el día)


# Tu primera clase

La programación orientada a objetos es **el arte de definir y expandir clases**. Una clase es un modelo de una parte muy específica de la realidad, que refleja las propiedades y actividades que se encuentran en el mundo real.

Las clases definidas al principio son demasiado generales e imprecisas para cubrir el mayor número posible de casos reales.

No hay obstáculo para definir nuevas subclases más precisas. Heredarán todo de su superclase, por lo que el trabajo que se utilizó para su creación no se desperdicia.

La nueva clase puede agregar nuevas propiedades y nuevas actividades y, por lo tanto, puede ser más útil en aplicaciones específicas. Obviamente, se puede usar como una superclase para cualquier número de subclases recién creadas.

El proceso no necesita tener un final. Puedes crear tantas clases como necesites.

La clase que se define no tiene nada que ver con el objeto: **la existencia de una clase no significa que ninguno de los objetos compatibles se creará automáticamente**. La clase en sí misma no puede crear un objeto: debes crearlo tu mismo y Python te permite hacerlo.

Es hora de definir la clase más simple y crear un objeto. Analiza el siguiente ejemplo:

```
<code python>class TheSimplestClass:
```

```
    pass
```

</code>

Hemos definido una clase. La clase es bastante pobre: no contiene propiedades ni actividades. Esta **vacía**, pero eso no importa por ahora. Cuanto más simple sea la clase, mejor para nuestros propósitos.

**La definición comienza con la palabra clave reservada *class***. La palabra clave reservada es seguida por **un identificador que le dará nombre a la clase** (nota: no lo confundas con el nombre del objeto: estas son dos cosas diferentes).

A continuación, se agregan **dos puntos** (:), como clases, como funciones, forman su propio bloque anidado. El contenido dentro del bloque define todas las propiedades y actividades de la clase.

La palabra clave reservada pass llena la clase con nada. No contiene ningún método ni propiedades.

## Tu primer objeto

La clase recién definida se convierte en una herramienta que puede crear nuevos objetos. La herramienta debe usarse explícitamente, bajo demanda.

Imagina que deseas crear un objeto (exactamente uno) de la clase TheSimplestClass.

Para hacer esto, debes asignar una variable para almacenar el objeto recién creado de esa clase y crear un objeto al mismo tiempo.

Se hace de la siguiente manera:

```
my_first_object = TheSimplestClass()
```

Nota:

- El nombre de la clase intenta fingir que es una función, ¿puedes ver esto? Lo discutiremos pronto.
- El objeto recién creado está equipado con todo lo que trae la clase. Como esta clase está completamente vacía, el objeto también está vacío.

El acto de crear un objeto de la clase seleccionada también se llama **instanciación** (ya que el objeto se convierte en una **instancia de la clase**).

Dejemos las clases en paz por un breve momento, ya que ahora diremos algunas palabras sobre *pilas*. Sabemos que el concepto de clases y objetos puede no estar completamente claro todavía. No te preocupes, te explicaremos todo muy pronto.

## Puntos Clave

1. Una clase es una idea (más o menos abstracta) que se puede utilizar para crear varias encarnaciones; una encarnación de este tipo se denomina objeto.

2. Cuando una clase se deriva de otra clase, su relación se denomina herencia. La clase que deriva de la otra clase se denomina subclase. El segundo lado de esta relación se denomina superclase. Una forma de presentar dicha relación es en un diagrama de herencia, donde:

- Las superclases siempre se presentan encima de sus subclases.
- Las relaciones entre clases se muestran como flechas dirigidas desde la subclase hacia su superclase.

3. Los objetos están equipados con:

- Un nombre que los identifica y nos permite distinguirlos.
- Un conjunto de propiedades (el conjunto puede estar vacío).
- Un conjunto de métodos (también puede estar vacío).

4. Para definir una clase de Python,se necesita usar la palabra clave reservada class. Por ejemplo:

```
class This_Is_A_Class:
     pass
```

5. Para crear un objeto de la clase previamente definida, se necesita usar la clase como si fuera una función. Por ejemplo:

```
this_is_an_object = This_Is_A_Class()
```

# ¿Qué es una pila?

**Una pila es una estructura desarrollada para almacenar datos de una manera muy específica**. Imagina una pila de monedas. No puedes poner una moneda en ningún otro lugar sino en la parte superior de la pila.

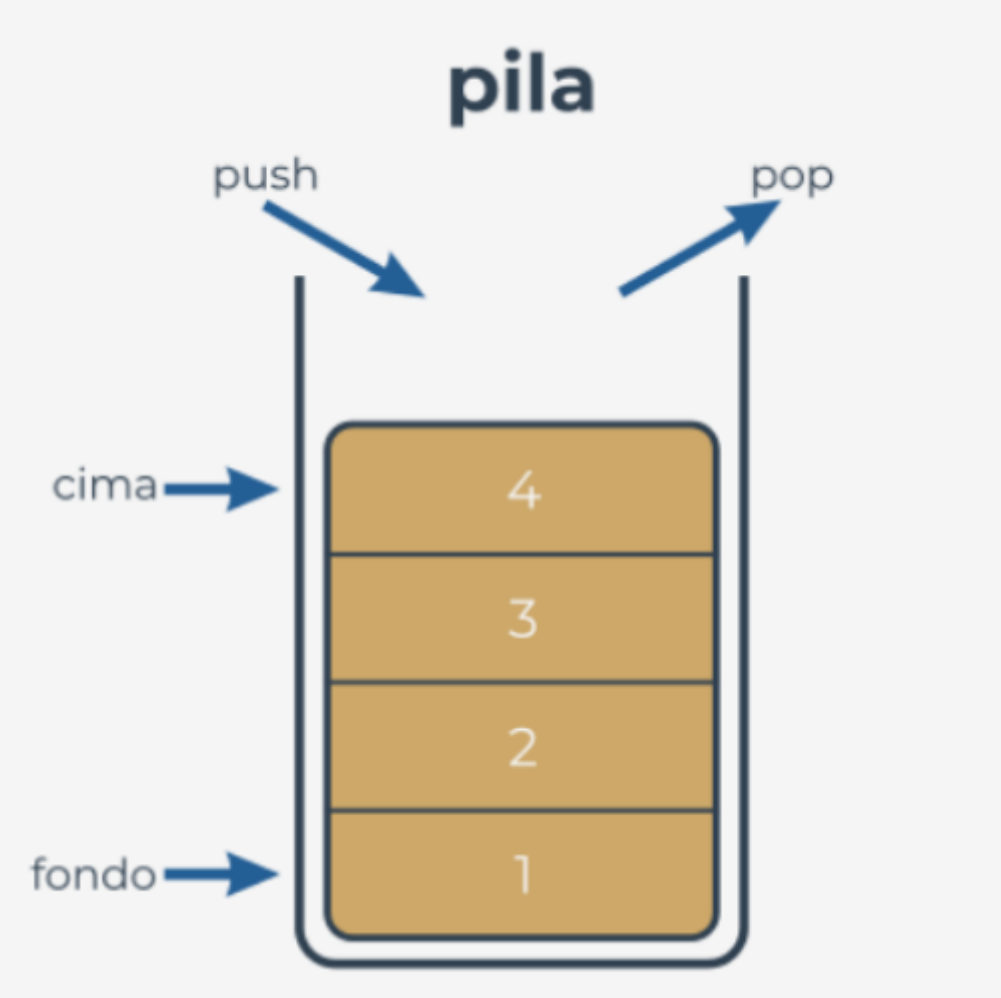
Del mismo modo, no puedes sacar una moneda de la pila desde ningún lugar que no sea la parte superior de la pila. Si deseas obtener la moneda que se encuentra en la parte inferior, debes eliminar todas las monedas de los niveles superiores.

El nombre alternativo para una pila (pero solo en la terminología de TI) es **UEPS (LIFO son sus siglas en inglés)**.

Es una abreviatura para una descripción muy clara del comportamiento de la pila: **Último en Entrar - Primero en Salir (Last In - First Out)**. La moneda que quedó en último lugar en la pila saldrá primero.

**Una pila es un objeto** con dos operaciones elementales, denominadas convencionalmente **push** (cuando un nuevo elemento se coloca en la parte superior) y **pop** (cuando un elemento existente se retira de la parte superior).

Las pilas se usan muy a menudo en muchos algoritmos clásicos, y es difícil imaginar la implementación de muchas herramientas ampliamente utilizadas sin el uso de pilas.




Implementemos una pila en Python. Esta será una pila muy simple, y te mostraremos como hacerlo en dos enfoques independientes: de manera procedimental y orientado a objetos.

## La pila: el enfoque procedimental

Primero, debes decidir como almacenar los valores que llegarán a la pila. Sugerimos utilizar el método más simple, y **emplear una lista** para esta tarea. Supongamos que el tamaño de la pila no está limitado de ninguna manera. Supongamos también que el último elemento de la lista almacena el elemento superior.

La pila en sí ya está creada:

```
stack = []
```

Estamos listos para **definir una función que coloca un valor en la pila**. Aquí están las presuposiciones para ello:

- El nombre para la función es push.
- La función obtiene un parámetro (este es el valor que se debe colocar en la pila).
- La función no retorna nada.
- La función agrega el valor del parámetro al final de la pila.

Así es como lo hemos hecho, echa un vistazo:

```
def push(val):
    stack.append(val)
```

Ahora es tiempo de que una **función quite un valor de la pila**. Así es como puedes hacerlo:

- El nombre de la función es pop.
- La función no obtiene ningún parámetro.
- La función devuelve el valor tomado de la pila.
- La función lee el valor de la parte superior de la pila y lo elimina.

La función esta aqui:

```
def pop():
    val = stack[-1]
    del stack[-1]
    return val
```

Nota: la función no verifica si hay algún elemento en la pila.

Armemos todas las piezas juntas para poner la pila en movimiento. El programa completo empuja (push) tres números a la pila, los saca e imprime sus valores en pantalla.

```
stack = []


def push(val):
    stack.append(val)


def pop():
    val = stack[-1]
    del stack[-1]
    return val


push(3)
push(2)
push(1)

print(pop())
print(pop())
print(pop())
```

El programa muestra el siguiente texto en pantalla:

```
1
2
3
```

# La pila: el enfoque procedimental frente al enfoque orientado a objetos

La pila procedimental está lista. Por supuesto, hay algunas debilidades, y la implementación podría mejorarse de muchas maneras (aprovechar las excepciones es una buena idea), pero en general la pila está completamente implementada, y puedes usarla si lo necesitas.

Pero cuanto más la uses, más desventajas encontrarás. Éstas son algunas de ellas:

- La variable esencial (la lista de la pila) es altamente vulnerable; cualquiera puede modificarla de forma incontrolable, destruyendo la pila; esto no significa que se haya hecho de manera maliciosa; por el contrario, puede ocurrir como resultado de un descuido, por ejemplo, cuando alguien confunde nombres de variables; imagina que accidentalmente has escrito algo como esto:

  ```
  stack[0] = 0
  ```

  El funcionamiento de la pila estará completamente desorganizado.

- También puede suceder que un día necesites más de una pila; tendrás que crear otra lista para el almacenamiento de la pila, y probablemente otras funciones push y pop.
- También puede suceder que no solo necesites funciones push y pop, pero también algunas otras funciones; ciertamente podrías implementarlas, pero intenta imaginar qué sucedería si tuvieras docenas de pilas implementadas por separado.

El enfoque orientado a objetos ofrece soluciones para cada uno de los problemas anteriores. Vamos a nombrarlos primero:

- La capacidad de ocultar (proteger) los valores seleccionados contra el acceso no autorizado se llama **encapsulamiento; no se puede acceder a los valores encapsulados ni modificarlos si deseas utilizarlos exclusivamente**.
- Cuando tienes una clase que implementa todos los comportamientos de pila necesarios, puedes producir tantas pilas como desees; no necesitas copiar ni replicar ninguna parte del código.
- La capacidad de enriquecer la pila con nuevas funciones proviene de la herencia; puedes crear una nueva clase (una subclase) que herede todos los rasgos existentes de la superclase y agregar algunos nuevos.


Ahora escribamos una nueva implementación de pila desde cero. Esta vez, utilizaremos el enfoque orientado a objetos, que te guiará paso a paso en el mundo de la programación de objetos.

## La pila, el enfoque orientado a objetos

Por supuesto, la idea principal sigue siendo la misma. Usaremos una lista como almacenamiento de la pila. Solo tenemos que saber como poner la lista en la clase.

Comencemos desde el principio: así es como comienza la pila orientada a objetos:

```
class Stack:
```

Ahora, esperamos dos cosas de la clase:

- Queremos que la clase tenga **una propiedad como el almacenamiento de la pila**, tenemos que «**instalar» una lista dentro de cada objeto de la clase** (nota: cada objeto debe tener su propia lista; la lista no debe compartirse entre diferentes pilas).
- Despues, queremos que **la lista esté oculta** de la vista de los usuarios de la clase.

¿Cómo se hace esto?

A diferencia de otros lenguajes de programación, Python no tiene medios para permitirte declarar una propiedad como esa.

En su lugar, debes agregar una instrucción específica. Las propiedades deben agregarse a la clase manualmente.

¿Cómo garantizar que dicha actividad tiene lugar cada vez que se crea una nueva pila?

Existe una manera simple de hacerlo, tienes que **equipar a la clase con una función específica**:

- Tiene que ser nombrada de forma estricta.
- Se invoca implícitamente cuando se crea el nuevo objeto.

Dicha función es llamada el **constructor**, ya que su propósito general es **construir un nuevo objeto**. El constructor debe saber todo acerca de la estructura del objeto y debe realizar todas las inicializaciones necesarias.

Agreguemos un constructor muy simple a la nueva clase. Echa un vistazo al código:

```
class Stack:
    def __init__(self):
        print("¡Hola!")


stack_object = Stack()
```

Expliquemos más a detalle:

- El nombre del constructor es siempre __init__.
- Tiene que tener **al menos un parámetro** (discutiremos esto más adelante); el parámetro se usa para representar el objeto recién creado: puedes usar el parámetro para manipular el objeto y enriquecerlo con las propiedades necesarias; harás uso de esto pronto.
- Nota: el parámetro obligatorio generalmente se denomina *self*, es solo **una sugerencia, pero deberías seguirla**, simplifica el proceso de lectura y comprensión de tu código.

```
class Stack:  # Definiendo la clase de la pila.
    def __init__(self):  # Definiendo la función del constructor.
        print("¡Hola!")


stack_object = Stack()  # Instanciando el objeto.
```

Aquí está su salida:

```
¡Hola!
```

Nota: no hay rastro de la invocación del constructor dentro del código. Ha sido invocado implícita y

automáticamente. Hagamos uso de eso ahora.

Cualquier cambio que realices dentro del constructor que modifique el estado del parámetro *self* se verá reflejado en el objeto recien creado.

Esto significa que puedes agregar cualquier propiedad al objeto y la propiedad permanecerá allí hasta que el objeto termine su vida o la propiedad se elimine explícitamente.

Ahora **agreguemos solo una propiedad al nuevo objeto**, una lista para la pila. La nombraremos stack_list.

```
class Stack:
    def __init__(self):
        self.stack_list = []


stack_object = Stack()
print(len(stack_object.stack_list))
```

Nota:

- Hemos usado la **notación punteada**, al igual que cuando se invocan métodos. Esta es la manera general para acceder a las propiedades de un objeto: debes nombrar el objeto, poner un punto (.) después de el, y especificar el nombre de la propiedad deseada, ¡no uses paréntesis! No deseas invocar un método, deseas **acceder a una propiedad**.
- Si estableces el valor de una propiedad por primera vez (como en el constructor), lo estás creando; a partir de ese momento, el objeto tiene la propiedad y está listo para usar su valor.
- Hemos hecho algo más en el código: hemos intentado acceder a la propiedad stack_list desde fuera de la clase inmediatamente después de que se haya creado el objeto; queremos verificar la longitud actual de la pila, ¿lo hemos logrado?

Si, por supuesto: el código produce el siguiente resultado:

```
0
```

Esto no es lo que queremos de la pila. Nosotros queremos que stack_list este escondida del mundo exterior. ¿Es eso posible?

Si, y es simple, pero no muy intuitivo.

Echa un vistazo: hemos agregado dos guiones bajos antes del nombre stack_list, nada mas:

```
class Stack:
    def __init__(self):
        self.__stack_list = []

stack_object = Stack()
print(len(stack_object.__stack_list))
```

El cambio invalida el programa.

¿Por qué?

Cuando cualquier componente de la clase tiene un **nombre que comienza con dos guiones bajos (__), se vuelve privado**, esto significa que solo se puede acceder desde dentro de la clase.

No puedes verlo desde el mundo exterior. Así es como Python implementa el concepto de **encapsulación**.

Ejecuta el programa para probar nuestras suposiciones: una excepción *AttributeError* debe ser generada.

## El enfoque orientado a objetos: una pila desde cero

Ahora es el momento de que las dos funciones (métodos) implementen las operaciones push y pop. Python supone que una función de este tipo debería estar **inmersa dentro del cuerpo de la clase**, como el constructor.

Queremos invocar estas funciones para agregar(push) y quitar(pop) valores de la pila. Esto significa que ambos deben ser accesibles para el usuario de la clase (en contraste con la lista previamente construida, que está oculta para los usuarios de la clase ordinaria).

Tal componente es llamado **público**, por ello **no puede comenzar su nombre con dos (o más) guiones bajos**. Hay un requisito más el nombre **no debe tener más de un guión bajo**.

Las funciones en sí son simples. Echa un vistazo:

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


stack_object = Stack()

stack_object.push(3)
stack_object.push(2)
stack_object.push(1)

print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())
```

Sin embargo, hay algo realmente extraño en el código. Las funciones parecen familiares, pero tienen más parámetros que sus contrapartes procedimentales.

Aquí, ambas funciones tienen un parámetro llamado **self** en la primera posición de la lista de parámetros.

¿Es necesario? Si, lo es.

Todos los métodos deben tener este parámetro. Desempeña el mismo papel que el primer parámetro constructor.

**Permite que el método acceda a entidades (propiedades y actividades / métodos) del objeto**. No puedes omitirlo. Cada vez que Python invoca un método, envía implícitamente el objeto actual como el primer argumento.

Esto significa que el **método está obligado a tener al menos un parámetro, que Python mismo utiliza**, no tienes ninguna influencia sobre el.

Si tu método no necesita ningún parámetro, este debe especificarse de todos modos. Si está diseñado para procesar solo un parámetro, debes especificar dos, ya que la función del primero sigue siendo la misma.

Hay una cosa más que requiere explicación: la forma en que se invocan los métodos desde la variable __stack_list.

Afortunadamente, es mucho más simple de lo que parece:

- La primera etapa entrega el objeto como un todo → self.
- A continuación, debes llegar a la lista __stack_list → self.__stack_list.
- Con __stack_list lista para ser usada, puedes realizar el tercer y último paso → self.__stack_list.append(val).

La declaración de la clase está completa y se han enumerado todos sus componentes. La clase está lista para usarse.

Tener tal clase abre nuevas posibilidades. Por ejemplo, ahora puedes hacer que más de una pila se comporte de la misma manera. Cada pila tendrá su propia copia de datos privados, pero utilizará el mismo conjunto de métodos.

Esto es exactamente lo que queremos para este ejemplo.

Analiza el código:

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


stack_object_1 = Stack()
stack_object_2 = Stack()

stack_object_1.push(3)
stack_object_2.push(stack_object_1.pop())

print(stack_object_2.pop())
```

Existen **dos pilas creadas a partir de la misma clase base**. Trabajan independientemente. Puedes crear más si quieres.

Ejecuta el código en el editor y observa que sucede. Realiza tus propios experimentos.

Analiza el fragmento de código a continuación: hemos creado tres objetos de la clase Stack. Después, hemos hecho malabarismos. Intenta predecir el valor que se muestra en la pantalla.

```
class Stack:
```

```
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


little_stack = Stack()
another_stack = Stack()
funny_stack = Stack()

little_stack.push(1)
another_stack.push(little_stack.pop() + 1)
funny_stack.push(another_stack.pop() - 2)

print(funny_stack.pop())
```

Ahora vamos un poco mas lejos. Vamos a **agregar una nueva clase para manejar pilas**.

La nueva clase debería poder **evaluar la suma de todos los elementos almacenados actualmente en la pila**.

No queremos modificar la pila previamente definida. Ya es lo suficientemente buena en sus aplicaciones, y no queremos que cambie de ninguna manera. Queremos una nueva pila con nuevas capacidades. En otras palabras, queremos construir una subclase de la ya existente clase Stack.

El primer paso es fácil: **solo define una nueva subclase que apunte a la clase que se usará como superclase**.

Así es como se ve:

```
class AddingStack(Stack):
    pass
```

La clase aún no define ningún componente nuevo, pero eso no significa que esté vacía. **Obtiene (hereda) todos los componentes definidos por su superclase**, el nombre de la superclase se escribe después de los dos puntos, después del nombre de la nueva clase.

Esto es lo que queremos de la nueva pila:

- Queremos que el método push no solo inserte el valor en la pila, sino que también sume el valor a la variable sum.
- Queremos que la función pop no solo extraiga el valor de la pila, sino que también reste el valor de la variable sum.

En primer lugar, agreguemos una nueva variable a la clase. Será una **variable privada**, al igual que la lista de pila. No queremos que nadie manipule el valor de la variable sum.

Como ya sabes, el constructor agrega una nueva propiedad a la clase. Ya sabes como hacerlo, pero hay algo realmente intrigante dentro del constructor. Echa un vistazo:

```
class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
```

La segunda línea del cuerpo del constructor crea una propiedad llamada **__sum**, almacenará el total de todos los valores de la pila.

Pero la línea anterior se ve diferente. ¿Qué hace? ¿Es realmente necesaria? Sí lo es.

Al contrario de muchos otros lenguajes, Python te obliga a **invocar explícitamente el constructor de una superclase**. Omitir este punto tendrá efectos nocivos: el objeto se verá privado de la lista __stack_list. Tal pila no funcionará correctamente.

Esta es la única vez que puedes invocar a cualquiera de los constructores disponibles explícitamente; se puede hacer dentro del constructor de la superclase.

Ten en cuenta la sintaxis:

- Se especifica el nombre de la superclase (esta es la clase cuyo constructor se desea ejecutar).
- Se pone un punto (.) después del nombre.
- Se especifica el nombre del constructor.
- Se debe señalar al objeto (la instancia de la clase) que debe ser inicializado por el constructor; es por eso que se debe especificar el argumento y utilizar la variable self aquí; recuerda: **invocar cualquier método (incluidos los constructores) desde fuera de la clase nunca requiere colocar el argumento self en la lista de argumentos**, invocar un método desde dentro de la clase exige el uso explícito del argumento self, y tiene que ser el primero en la lista.

Nota: generalmente es una práctica recomendada invocar al constructor de la superclase antes de cualquier otra inicialización que desees realizar dentro de la subclase. Esta es la regla que hemos seguido en el código.

En segundo lugar, agreguemos dos métodos. Pero, ¿realmente estamos agregándolos? Ya tenemos estos métodos en la superclase. ¿Podemos hacer algo así?

Si podemos. Significa que vamos a **cambiar la funcionalidad de los métodos**, no sus nombres. Podemos decir con mayor precisión que la interfaz (la forma en que se manejan los objetos) de la clase permanece igual al cambiar la implementación al mismo tiempo.

Comencemos con la implementación de la función push. Esto es lo que esperamos de la función:

- Agregar el valor a la variable __sum.
- Agregar el valor a la pila.

Nota: la segunda actividad ya se implementó dentro de la superclase, por lo que podemos usarla. Además, tenemos que usarla, ya que no hay otra forma de acceder a la variable __stackList.

Así es como se mira el método push dentro de la subclase:

```
def push(self, val):
    self.__sum += val
    Stack.push(self, val)
```

Toma en cuenta la forma en que hemos invocado la implementación anterior del método push (el disponible en la superclase):

- Tenemos que especificar el nombre de la superclase; esto es necesario para indicar claramente la clase que contiene el método, para evitar confundirlo con cualquier otra función del mismo nombre.
- Tenemos que especificar el objeto de destino y pasarlo como primer argumento (no se agrega

implícitamente a la invocación en este contexto).

Se dice que el método push ha sido anulado, el mismo nombre que en la superclase ahora representa una funcionalidad diferente.

Esta es la nueva función pop:

```
def pop(self):
    val = Stack.pop(self)
    self.__sum -= val
    return val
```

Hasta ahora, hemos definido la variable __sum, pero no hemos proporcionado un método para obtener su valor. Parece estar escondido. ¿Cómo podemos mostrarlo y que al mismo tiempo que se proteja de modificaciones?

Tenemos que definir un nuevo método. Lo nombraremos **get_sum**. **Su única tarea será devolver el valor de __sum**.

Aquí está:

```
def get_sum(self):
    return self.__sum
```

Entonces, veamos el programa en el editor. El código completo de la clase está ahí. Podemos ahora verificar su funcionamiento, y lo hacemos con la ayuda de unas pocas líneas de código adicionales.

Como puedes ver, agregamos cinco valores subsiguientes en la pila, imprimimos su suma y los sacamos todos de la pila.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def get_sum(self):
        return self.__sum

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)

    def pop(self):
```

```
        val = Stack.pop(self)
        self.__sum -= val
        return val


stack_object = AddingStack()

for i in range(5):
    stack_object.push(i)
print(stack_object.get_sum())

for i in range(5):
    print(stack_object.pop())
```

# Puntos Clave

1. Una **pila** es un objeto diseñado para almacenar datos utilizando el modelo **LIFO**. La pila normalmente realiza al menos dos operaciones, llamadas **push()** y **pop()**.

2. La implementación de la pila en un modelo procedimental plantea varios problemas que pueden resolverse con las técnicas ofrecidas por la **POO** (**P**rogramación **O**rientada a **O**bjetos).

3. Un **método** de clase es en realidad una función declarada dentro de la clase y capaz de acceder a todos los componentes de la clase.

4. La parte de la clase en Python responsable de crear nuevos objetos se llama **constructor** y se implementa como un método de nombre __init__.

5. Cada declaración de método de clase debe contener al menos un parámetro (siempre el primero) generalmente denominado self, y es utilizado por los objetos para identificarse a sí mismos.

6. Si queremos ocultar alguno de los componentes de una clase del mundo exterior, debemos comenzar su nombre con __. Estos componentes se denominan **privados**.

# ejercicio

## Objetivos

- Mejorar las habilidades del estudiante para definir clases.
- Emplear clases existentes para crear nuevas clases equipadas con nuevas funcionalidades.

## Escenario

Recientemente te mostramos cómo extender las posibilidades de Stack definiendo una nueva clase (es decir, una subclase) que retiene todos los rasgos heredados y agrega algunos nuevos.

Tu tarea es extender el comportamiento de la clase Stack de tal manera que la clase pueda contar todos los elementos que son agregados (push) y quitados (pop). Emplea la clase Stack que proporcionamos en el editor.

Sigue las sugerencias:

Introduce una propiedad diseñada para contar las operaciones pop y nombrarla de una manera que garantice

que esté oculta. Inicializala a cero dentro del constructor. Proporciona un método que devuelva el valor asignado actualmente al contador (nómbralo get_counter()).

```
class Stack:
    def __init__(self):
        self.__stk = []

    def push(self, val):
        self.__stk.append(val)

    def pop(self):
        val = self.__stk[-1]
        del self.__stk[-1]
        return val


class CountingStack(Stack):
    def __init__(self):
    #
    # Llena el constructor con acciones apropiadas.
    #

    def get_counter(self):
    #
    # Presenta el valor actual del contador al mundo.
    #

    def pop(self):
    #
    # Haz un pop y actualiza el contador.
    #


stk = CountingStack()
for i in range(100):
    stk.push(i)
    stk.pop()
print(stk.get_counter())
```

Completa el código en el editor. Ejecútalo para comprobar si tu código da como salida 100.

# ejercicio

## Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Implementar estructuras de datos estándar como clases.

## Escenario

Como ya sabes, una pila es una estructura de datos que realiza el modelo LIFO (último en entrar, primero en

salir). Es fácil y ya te has acostumbrado a ello perfectamente.

Probemos algo nuevo ahora. Una cola (queue) es un modelo de datos caracterizado por el término FIFO: primero en entrar, primero en salir. Nota: una cola (fila) regular que conozcas de las tiendas u oficinas de correos funciona exactamente de la misma manera: un cliente que llegó primero también es el primero en ser atendido.

Tu tarea es implementar la clase Queue con dos operaciones básicas:

- put(elemento), que coloca un elemento al final de la cola.
- get(), que toma un elemento del principio de la cola y lo devuelve como resultado (la cola no puede estar vacía para realizarlo correctamente).

Sigue las sugerencias:

- Emplea una lista como tu almacenamiento (como lo hicimos con la pila).
- put() debe agregar elementos al principio de la lista, mientras que get() debe eliminar los elementos del final de la lista.
- Define una nueva excepción llamada QueueError (elige una excepción de la cual se derivará) y generala cuando get() intentes operar en una lista vacía.

Completa el código que te proporcionamos en el editor. Ejecútalo para comprobar si tu salida es similar a la nuestra.

Salida Esperada

```
1
perro
False
Error de Cola
```

```
class QueueError(???):  # Eligir la clase base para la nueva excepción.
    #
    #  Escribe código aquí.
    #


class Queue:
    def __init__(self):
        #
        # Escribe código aquí.
        #

    def put(self, elem):
        #
        # Escribe código aquí.
        #

    def get(self):
        #
        # Escribe código aquí.
        #


que = Queue()
que.put(1)
que.put("perro")
que.put(False)
```

```
try:
    for i in range(4):
        print(que.get())
except:
    print("Error de Cola")
```

# ejercicio

## Objetivos

- Mejorar las habilidades del estudiante para definir subclases.
- Agregar nueva funcionalidad a una clase existente.

## Escenario

Tu tarea es extender ligeramente las capacidades de la clase Queue. Queremos que tenga un método sin parámetros que devuelva True si la cola está vacía y False de lo contrario.

Completa el código que te proporcionamos en el editor. Ejecútalo para comprobar si genera un resultado similar al nuestro.

Salida esperada:

```
1
perro
False
Cola vacía
```

```
class QueueError(???):
    pass


class Queue:
    #
    # Código del laboratorio anterior.
    #


class SuperQueue(Queue):
    #
    # Escribe código nuevo aquí.
    #


que = SuperQueue()
que.put(1)
que.put("perro")
que.put(False)
for i in range(4):
    if not que.isempty():
        print(que.get())
```

```
    else:
        print("Cola vacía")
```

## Variables de instancia

En general, una clase puede equiparse con dos tipos diferentes de datos para formar las propiedades de una clase. Ya viste uno de ellos cuando estábamos estudiando pilas.

Este tipo de propiedad existe solo cuando se crea explícitamente y se agrega a un objeto. Como ya sabes, esto se puede hacer durante la inicialización del objeto, realizada por el constructor.

Además, se puede hacer en cualquier momento de la vida del objeto. Es importante mencionar también que cualquier propiedad existente se puede eliminar en cualquier momento.

Tal enfoque tiene algunas consecuencias importantes:

- Diferentes objetos de la misma clase **pueden poseer diferentes conjuntos de propiedades**.
- Debe haber una manera de **verificar con seguridad si un objeto específico posee la propiedad** que deseas utilizar (a menos que quieras generar una excepción, siempre vale la pena considerarlo).
- Cada objeto **lleva su propio conjunto de propiedades**, no interfieren entre sí de ninguna manera.

Tales variables (propiedades) se llaman **variables de instancia**.

La palabra instancia sugiere que están estrechamente conectadas a los objetos (que son instancias de clase), no a las clases mismas. Echemos un vistazo más de cerca.

Aquí hay un ejemplo:

```
class ExampleClass:
    def __init__(self, val = 1):
        self.first = val

    def set_second(self, val):
        self.second = val


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)

example_object_2.set_second(3)

example_object_3 = ExampleClass(4)
example_object_3.third = 5

print(example_object_1.__dict__)
print(example_object_2.__dict__)
print(example_object_3.__dict__)
```

Se necesita una explicación adicional antes de entrar en más detalles. Echa un vistazo a las últimas tres líneas del código.

Los objetos de Python, cuando se crean, **están dotados de un pequeño conjunto de propiedades y métodos predefinidos**. Cada objeto los tiene, los quieras o no. Uno de ellos es una variable llamada __dict__ (es un diccionario).

La variable contiene los nombres y valores de todas las propiedades (variables) que el objeto contiene actualmente. Vamos a usarla para presentar de forma segura el contenido de un objeto.

Vamos a sumergirnos en el código ahora:

- La clase llamada ExampleClass tiene un constructor, el cual **crea incondicionalmente una variable de instancia** llamada first, y le asigna el valor pasado a través del primer argumento (desde la perspectiva del usuario de la clase) o el segundo argumento (desde la perspectiva del constructor); ten en cuenta el valor predeterminado del parámetro: cualquier cosa que puedas hacer con un parámetro de función regular también se puede aplicar a los métodos.
- La clase también tiene un **método que crea otra variable de instancia**, llamada second.
- Hemos creado tres objetos de la clase ExampleClass, pero todas estas instancias difieren:
    - example_object_1 solo tiene una propiedad llamada first.
    - example_object_2 tiene dos propiedades: first y second.
    - example_object_3 ha sido enriquecido sobre la marcha con una propiedad llamada third uera del código de la clase: esto es posible y totalmente permisible.

La salida del programa muestra claramente que nuestras suposiciones son correctas: aquí están:

```
{'first': 1}
{'second': 3, 'first': 2}
{'third': 5, 'first': 4}
```

Hay una conclusión adicional que debería mencionarse aquí: **el modificar una variable de instancia de cualquier objeto no tiene impacto en todos los objetos restantes**. Las variables de instancia están perfectamente aisladas unas de otras.

```
class ExampleClass:
    def __init__(self, val = 1):
        self.__first = val

    def set_second(self, val = 2):
        self.__second = val


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)

example_object_2.set_second(3)

example_object_3 = ExampleClass(4)
example_object_3.__third = 5


print(example_object_1.__dict__)
print(example_object_2.__dict__)
print(example_object_3.__dict__)
```

Es casi lo mismo que el anterior. La única diferencia está en los nombres de las propiedades. Hemos **antepuesto dos guiones bajos** (__).

Como sabes, tal adición hace que la variable de instancia sea privada, se vuelve inaccesible desde el mundo exterior.

El comportamiento real de estos nombres es un poco más complicado, así que ejecutemos el programa. Esta es

la salida:

```
{'_ExampleClass__first': 1}
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
```

¿Puedes ver estos nombres extraños llenos de guiones bajos? ¿De dónde provienen?

Cuando Python ve que deseas agregar una variable de instancia a un objeto y lo vas a hacer dentro de cualquiera de los métodos del objeto, **maneja la operación** de la siguiente manera:

- Coloca un nombre de clase antes de tu nombre.
- Coloca un guión bajo adicional al principio.

Es por ello que __first se convierte en _ExampleClass__first.

**El nombre ahora es completamente accesible desde fuera de la clase**. Puedes ejecutar un código como este:

print(example_object_1._ExampleClass__first)

Obtendrás un resultado válido sin errores ni excepciones.

Como puedes ver, hacer que una propiedad sea privada es limitado.

No funcionará si agregas una variable de instancia fuera del código de la clase. En este caso, se comportará como cualquier otra propiedad ordinaria.

## Variables de clase

Una variable de clase es **una propiedad que existe en una sola copia y se almacena fuera de cualquier objeto**.

Nota: no existe una variable de instancia si no hay ningún objeto de la clase; solo existe una variable de clase en una copia, incluso si no hay objetos en la clase.

Las variables de clase se crean de manera diferente. El ejemplo te dirá más:

```
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)
example_object_3 = ExampleClass(4)

print(example_object_1.__dict__, example_object_1.counter)
print(example_object_2.__dict__, example_object_2.counter)
print(example_object_3.__dict__, example_object_3.counter)
```

Observa:

- Hay una asignación en la primera linea de la definición de clase: establece la variable denominada

counter a 0; inicializando la variable dentro de la clase pero fuera de cualquiera de sus métodos hace que la variable sea una variable de clase.
- El acceder a dicha variable tiene el mismo aspecto que acceder a cualquier atributo de instancia; está en el cuerpo del constructor; como puedes ver, el constructor incrementa la variable en uno; en efecto, la variable cuenta todos los objetos creados.

Ejecutar el código provocará el siguiente resultado:

```
{'_ExampleClass__first': 1} 3
{'_ExampleClass__first': 2} 3
{'_ExampleClass__first': 4} 3
```

Dos conclusiones importantes se pueden sacar del ejemplo:

- Las variables de clase **no se muestran en el diccionario de un objeto** __dict__ (esto es natural ya que las variables de clase no son partes de un objeto), pero siempre puedes intentar buscar en la variable del mismo nombre, pero a nivel de clase, te mostraremos esto muy pronto.
- Una variable de clase **siempre presenta el mismo valor** en todas las instancias de clase (objetos).

El cambiar el nombre de una variable de clase tiene los mismos efectos que aquellos con los que ya está familiarizado.

Mira el ejemplo en el editor. ¿Puedes adivinar su salida?

```
class ExampleClass:
    __counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.__counter += 1


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)
example_object_3 = ExampleClass(4)

print(example_object_1.__dict__, example_object_1._ExampleClass__counter)
print(example_object_2.__dict__, example_object_2._ExampleClass__counter)
print(example_object_3.__dict__, example_object_3._ExampleClass__counter)
```

Hemos dicho antes que las variables de clase existen incluso cuando no se creó ninguna instancia de clase (objeto).

Ahora aprovecharemos la oportunidad para mostrarte **la diferencia entre estas dos variables** __dict__, la de la clase y la del objeto.

```
class ExampleClass:
    varia = 1
    def __init__(self, val):
        ExampleClass.varia = val


print(ExampleClass.__dict__)
example_object = ExampleClass(2)

print(ExampleClass.__dict__)
```

```
print(example_object.__dict__)
```

Echemos un vistazo más de cerca:

1. Definimos una clase llamada ExampleClass.
2. La clase define una variable de clase llamada varia.
3. El constructor de la clase establece la variable con el valor del parámetro.
4. Nombrar la variable es el aspecto más importante del ejemplo porque:
    - El cambiar la asignación a self.varia = val crearía una variable de instancia con el mismo nombre que la de la clase.
    - El cambiar la asignación a varia = val operaría en la variable local de un método; (te recomendamos probar los dos casos anteriores; esto te facilitará recordar la diferencia).
5. La primera línea del código fuera de la clase imprime el valor del atributo ExampleClass.varia . Nota: utilizamos el valor antes de instanciar el primer objeto de la clase.

Ejecuta el código en el editor y verifica su salida.

```
{'__module__': '__main__', 'varia': 1, '__init__': <function ExampleClass.__init__
at 0x7fc83922b0e0>, '__dict__': <attribute '__dict__' of 'ExampleClass' objects>,
'__weakref__': <attribute '__weakref__' of 'ExampleClass' objects>, '__doc__':
None}
{'__module__': '__main__', 'varia': 2, '__init__': <function ExampleClass.__init__
at 0x7fc83922b0e0>, '__dict__': <attribute '__dict__' of 'ExampleClass' objects>,
'__weakref__': <attribute '__weakref__' of 'ExampleClass' objects>, '__doc__':
None}
```

Como puedes ver __dict__ contiene muchos más datos que la contraparte de su objeto. La mayoría de ellos son inútiles ahora, el que queremos que verifiques cuidadosamente muestra el valor actual de varia.

Nota que el __dict__ del objeto está vacío, el objeto no tiene variables de instancia.

## Comprobando la existencia de un atributo

La actitud de Python hacia la instanciación de objetos plantea una cuestión importante: en contraste con otros lenguajes de programación, **es posible que no esperes que todos los objetos de la misma clase tengan los mismos conjuntos de propiedades**.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)

print(example_object.a)
print(example_object.b)
```

El objeto creado por el constructor solo puede tener uno de los dos atributos posibles: a o b.

La ejecución del código producirá el siguiente resultado:

```
1
Traceback (most recent call last):
  File ".main.py", line 11, in
    print(example_object.b)
AttributeError: 'ExampleClass' object has no attribute 'b'
```

Como puedes ver, acceder a un atributo de objeto (clase) no existente genera una excepción AttributeError.

La instrucción **try-except** te brinda la oportunidad de evitar problemas con propiedades inexistentes.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)
print(example_object.a)

try:
    print(example_object.b)
except AttributeError:
    pass
```

Como puedes ver, esta acción no es muy sofisticada. Esencialmente, acabamos de barrer el tema debajo de la alfombra.

Afortunadamente, hay una forma más de hacer frente al problema.

Python proporciona una **función que puede verificar con seguridad si algún objeto / clase contiene una propiedad específica**. La función se llama *hasattr*, y espera que le pasen dos argumentos:

- La clase o el objeto que se verifica.
- El nombre de la propiedad cuya existencia se debe informar (Nota: debe ser una cadena que contenga el nombre del atributo).

La función retorna True o False.

Así es como puedes utilizarla:

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)
print(example_object.a)

if hasattr(example_object, 'b'):
    print(example_object.b)
```

No olvides que la función `hasattr()` también puede operar en clases. Puedes usarla **para averiguar si una variable de clase está disponible**, como en el ejemplo en el editor.

La función devuelve True si la clase especificada contiene un atributo dado, y False de lo contrario.

```
class ExampleClass:
    attr = 1


print(hasattr(ExampleClass, 'attr'))
print(hasattr(ExampleClass, 'prop'))
```

Un ejemplo más: analiza el código a continuación e intenta predecir su salida:

```
class ExampleClass:
    a = 1
    def __init__(self):
        self.b = 2


example_object = ExampleClass()

print(hasattr(example_object, 'b'))
print(hasattr(example_object, 'a'))
print(hasattr(ExampleClass, 'b'))
print(hasattr(ExampleClass, 'a'))
```

Bien, hemos llegado al final de esta sección. En la siguiente sección vamos a hablar sobre los métodos, ya que los métodos dirigen los objetos y los activan.

## Puntos Clave

1. Una variable de instancia es una propiedad cuya existencia depende de la creación de un objeto. Cada objeto puede tener un conjunto diferente de variables de instancia.

Además, se pueden agregar y quitar libremente de los objetos durante su vida útil. Todas las variables de instancia de objeto se almacenan dentro de un diccionario dedicado llamado __dict__, contenido en cada objeto por separado.

2. Una variable de instancia puede ser privada cuando su nombre comienza con __, pero no olvides que dicha propiedad aún es accesible desde fuera de la clase usando un **nombre modificado** construido como < codel>_ClassName__PrivatePropertyName.

3. Una **variable de clase** es una propiedad que existe exactamente en una copia y no necesita ningún objeto creado para ser accesible. Estas variables no se muestran como contenido de __dict__.

Todas las variables de clase de una clase se almacenan dentro de un diccionario dedicado llamado __dict__, contenido en cada clase por separado.

4. Una función llamada `hasattr()` se puede utilizar para determinar si algún objeto o clase contiene cierta propiedad especificada.

Por ejemplo:

```
class Sample:
```

```
    gamma = 0 # Class variable.
    def __init__(self):
        self.alpha = 1 # Variable de instancia.
        self.__delta = 3 # Variable de instancia privada.


obj = Sample()
obj.beta = 2  # Otra variable de instancia (que existe solo dentro de la instancia
"obj").
print(obj.__dict__)
```

El código da como salida:

```
{'alpha': 1, '_Sample__delta': 3, 'beta': 2}
```

https://edube.org/learn/python-essentials-2-esp/poo-m-eacute-todos-10

# Métodos a detalle

Resumamos todos los hechos relacionados con el uso de métodos en las clases de Python.

Como ya sabes, un **método es una función que está dentro de una clase**.

Existe un requisito fundamental: un **método está obligado a tener al menos un parámetro** (no existen métodos sin parámetros; un método puede invocarse sin un argumento, pero no puede declararse sin parámetros).

El primer (o único) parámetro generalmente se denomina *self*. Te sugerimos que lo sigas nombrando de esta manera, darle otros nombres puede causar sorpresas inesperadas.

El nombre self sugiere el propósito del parámetro: **identifica el objeto para el cual se invoca el método**.

Si vas a invocar un método, no debes pasar el argumento para el parámetro self, Python lo configurará por ti.

```
class Classy:
    def method(self):
        print("método")


obj = Classy()
obj.method()
```

El código da como salida:

```
método
```

Toma en cuenta la forma en que hemos creado el objeto, **hemos tratado el nombre de la clase como una función**, y devuelve un objeto recién instanciado de la clase.

Si deseas que el método acepte parámetros distintos a self, debes:

- Colocarlos después de self en la definición del método.
- Pasarlos como argumentos durante la invocación sin especificar self.

Justo como aqui:

```
class Classy:
    def method(self, par):
        print("método:", par)


obj = Classy()
obj.method(1)
obj.method(2)
obj.method(3)
```

El código da como salida:

```
método: 1
método: 2
método: 3
```

El parámetro *self* es usado para obtener acceso a la instancia del objeto y las variables de clase.

El ejemplo muestra ambas formas de utilizar el parámetro self:

```
class Classy:
    varia = 2
    def method(self):
        print(self.varia, self.var)


obj = Classy()
obj.var = 3
obj.method()
```

El código da como salida:

```
2 3
```

El parámetro *self* también se usa para invocar otros métodos desde dentro de la clase.

Justo como aquí:

```
class Classy:
    def other(self):
        print("otro")

    def method(self):
        print("método")
        self.other()


obj = Classy()
obj.method()
```

El código da como salida:

```
método
```

```
otro
```

Si se nombra un método de esta manera: __init__, no será un método regular, será un **constructor**.

Si una clase tiene un constructor, este se invoca automática e implícitamente cuando se instancia el objeto de la clase.

El constructor:

- Esta **obligado a tener el parámetro *self*** (se configura automáticamente).
- **Pudiera (pero no necesariamente) tener mas parámetros** que solo *self*; si esto sucede, la forma en que se usa el nombre de la clase para crear el objeto debe tener la definición __init__.
- **Se puede utilizar para configurar el objeto**, es decir, inicializa adecuadamente su estado interno, crea variables de instancia, crea instancias de cualquier otro objeto si es necesario, etc.

El ejemplo muestra un constructor muy simple pero funcional.

```
class Classy:
    def __init__(self, value):
        self.var = value


obj_1 = Classy("objeto")

print(obj_1.var)
```

Ejecútalo. El código da como salida:

```
objeto
```

Ten en cuenta que el constructor:

- **No puede retornar un valor**, ya que está diseñado para devolver un objeto recién creado y nada más.
- **No se puede invocar directamente desde el objeto o desde dentro de la clase** (puedes invocar un constructor desde cualquiera de las superclases del objeto, pero discutiremos esto más adelante).

Como __init__ es un método, y un método es una función, puedes hacer los mismos trucos con constructores y métodos que con las funciones ordinarias.

El ejemplo en el editor muestra cómo definir un constructor con un valor de argumento predeterminado. Pruébalo.

```
class Classy:
    def __init__(self, value = None):
        self.var = value


obj_1 = Classy("objeto")
obj_2 = Classy()

print(obj_1.var)
print(obj_2.var)
```

El código da como salida:

```
objeto
None
```

Todo lo que hemos dicho sobre el manejo de los nombres también se aplica a los nombres de métodos, un método cuyo nombre comienza con __ está (parcialmente) oculto.

El ejemplo muestra este efecto:

```
class Classy:
    def visible(self):
        print("visible")

    def __hidden(self):
        print("oculto")


obj = Classy()
obj.visible()

try:
    obj.__hidden()
except:
    print("fallido")

obj._Classy__hidden()
```

El código da como salida:

```
visible
fallido
oculto
```

## La vida interna de clases y objetos

Cada clase de Python y cada objeto de Python está pre-equipado con un conjunto de atributos útiles que pueden usarse para examinar sus capacidades.

Ya conoces uno de estos: es la propiedad __dict__.

Observemos como esta propiedad trata con los métodos

```
class Classy:
    varia = 1
    def __init__(self):
        self.var = 2

    def method(self):
        pass

    def __hidden(self):
        pass


obj = Classy()
```

```
print(obj.__dict__)
print(Classy.__dict__)
```

Ejecútalo para ver que produce. Verifica el resultado.

```
{'var': 2}
{'__module__': '__main__', 'varia': 1, '__init__': <function Classy.__init__ at
0x7fcb0ae8c320>, 'method': <function Classy.method at 0x7fcb0ae8c3b0>,
'_Classy__hidden': <function Classy.__hidden at 0x7fcb0ae8c440>, '__dict__':
<attribute '__dict__' of 'Classy' objects>, '__weakref__': <attribute '__weakref__'
of 'Classy' objects>, '__doc__': None}
```

Encuentra todos los métodos y atributos definidos. Localiza el contexto en el que existen: dentro del objeto o dentro de la clase.

__dict__ es un diccionario. Otra propiedad incorporada que vale la pena mencionar es una cadena llamada __name__.

La propiedad contiene el **nombre de la clase**. No es nada emocionante, es solo una cadena.

Nota: el atributo __name__ está ausente del objeto, **existe solo dentro de las clases**.

Si deseas **encontrar la clase de un objeto en particular**, puedes usar una función llamada `type()`, la cual es capaz (entre otras cosas) de encontrar una clase que se haya utilizado para crear instancias de cualquier objeto.

Observa el código en el editor, ejecútalo y compruébalo tu mismo.

```
class Classy:
    pass


print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

La salida del código es:

```
Classy
Classy
```

Nota: algo como esto

```
print(obj.__name__)
```

causará un error.

__module__ es una cadena, también a**lmacena el nombre del módulo que contiene la definición de la clase**.

Vamos a comprobarlo: ejecuta el código en el editor.

La salida del código es:

```
__main__
__main__
```

Como sabes, cualquier módulo llamado __main__ en realidad no es un módulo, sino es el **archivo actualmente en ejecución**.

__bases__ es una tupla. La **tupla contiene clases** (no nombres de clases) que son superclases directas de la clase.

El orden es el mismo que el utilizado dentro de la definición de clase.

Te mostraremos solo un ejemplo muy básico, ya que queremos resaltar **cómo funciona la herencia**.

Además, te mostraremos cómo usar este atributo cuando discutamos los aspectos orientados a objetos de las excepciones.

Nota: **solo las clases tienen este atributo**, los objetos no.

Hemos definido una función llamada printBases(), diseñada para presentar claramente el contenido de la tupla.

```
class SuperOne:
    pass


class SuperTwo:
    pass


class Sub(SuperOne, SuperTwo):
    pass


def printBases(cls):
    print('( ', end='')

    for x in cls.__bases__:
        print(x.__name__, end=' ')
    print(')')


printBases(SuperOne)
printBases(SuperTwo)
printBases(Sub)
```

Su salida es:

```
( object )
( object )
( SuperOne SuperTwo )
```

Nota: **una clase sin superclases explícitas apunta a object** (una clase de Python predefinida) como su antecesor directo.

# Reflexión e introspección

Todo esto permite que el programador de Python realice dos actividades importantes específicas para muchos lenguajes objetivos. Las cuales son:

- **Introspección**, que es la capacidad de un programa para examinar el tipo o las propiedades de un objeto en tiempo de ejecución.
- **Reflexión**, que va un paso más allá, y es la capacidad de un programa para manipular los valores, propiedades y/o funciones de un objeto en tiempo de ejecución.

En otras palabras, no tienes que conocer la definición completa de clase/objeto para manipular el objeto, ya que el objeto y/o su clase contienen los metadatos que te permiten reconocer sus características durante la ejecución del programa.

# Investigando Clases

¿Qué puedes descubrir acerca de las clases en Python? La respuesta es simple: todo.

Tanto la reflexión como la introspección permiten al programador hacer cualquier cosa con cada objeto, sin importar de dónde provenga.

```
class MyClass:
    pass


obj = MyClass()
obj.a = 1
obj.b = 2
obj.i = 3
obj.ireal = 3.5
obj.integer = 4
obj.z = 5


def incIntsI(obj):
    for name in obj.__dict__.keys():
        if name.startswith('i'):
            val = getattr(obj, name)
            if isinstance(val, int):
                setattr(obj, name, val + 1)


print(obj.__dict__)
incIntsI(obj)
print(obj.__dict__)
```

La función llamada `incIntsI()` toma un objeto de cualquier clase, escanea su contenido para encontrar todos los atributos enteros con nombres que comienzan con i, y los incrementa en uno.

¿Imposible? ¡De ninguna manera!

Así es como funciona:

- La línea 1: define una clase muy simple...
- Las líneas 3 a la 10: ... la llenan con algunos atributos.
- La línea 14: ¡esta es nuestra función!
- La línea 15: escanea el atributo __dict__, buscando todos los nombres de atributos.
- La línea 16: si un nombre comienza con i...
- La línea 17: ... utiliza la función `getattr()` para obtener su valor actual; nota: `getattr()` toma dos argumentos: un objeto y su nombre de propiedad (como una cadena) y devuelve el valor del atributo actual.
- La línea 18: comprueba si el valor es de tipo entero, emplea la función `isinstance()` para este propósito (discutiremos esto más adelante).
- La línea 19: si la comprobación sale bien, incrementa el valor de la propiedad haciendo uso de la función `setattr()`; la función toma tres argumentos: un objeto, el nombre de la propiedad (como una cadena) y el nuevo valor de la propiedad.

El código da como salida:

```
{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}
{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}
```

## Puntos Clave

1. Un método es una función dentro de una clase. El primer (o único) parámetro de cada método se suele llamar *self*, que está diseñado para identificar al objeto para el que se invoca el método con el fin de acceder a las propiedades del objeto o invocar sus métodos.

2. Si una clase contiene un **constructor** (un método llamado __init__), este no puede devolver ningún valor y no se puede invocar directamente.

3. Todas las clases (pero no los objetos) contienen una propiedad llamada __name__, que almacena el nombre de la clase. Además, una propiedad llamada __module__ almacena el nombre del módulo en el que se ha declarado la clase, mientras que la propiedad llamada __bases__ es una tupla que contiene las superclases de una clase.

Por ejemplo:

```
class Sample:
    def __init__(self):
        self.name = Sample.__name__
    def myself(self):
        print("Mi nombre es " + self.name + " y vivo en " + Sample.__module__)


obj = Sample()
obj.myself()
```

El código da como salida:

```
Mi nombre es Sample y vivo en __main__
```

# ejercicio

## Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Definir y usar variables de instancia.
- Definir y usar métodos.

## Escenario

Necesitamos una clase capaz de contar segundos. ¿Fácil? No es tan fácil como podrías pensar, ya que tendremos algunos requisitos específicos.

Léelos con atención, ya que la clase sobre la que escribes se utilizará para lanzar cohetes en misiones internacionales a Marte. Es una gran responsabilidad. ¡Contamos contigo!

Tu clase se llamará *Timer* (temporizador en español). Su constructor acepta tres argumentos que representan **horas** (un valor del rango [0..23]; usaremos tiempo militar), **minutos** (del rango [0. .59]) y **segundos** (del rango [0..59]).

Cero es el valor predeterminado para todos los parámetros anteriores. No es necesario realizar ninguna comprobación de validación.

La clase en sí debería proporcionar las siguientes facilidades:

- Los objetos de la clase deben ser «imprimibles», es decir, deben poder convertirse implícitamente en cadenas de la siguiente forma: «hh:mm:ss», con ceros a la izquierda agregados cuando cualquiera de los valores es menor que 10.
- La clase debe estar equipada con métodos sin parámetros llamados next_second() y previous_second (), incrementando el tiempo almacenado dentro de los objetos en +1/-1 segundos respectivamente.

Emplea las siguientes sugerencias:

- Todas las propiedades del objeto deben ser privadas.
- Considera escribir una función separada (¡no un método!) para formatear la cadena con el tiempo.

```
class Timer:
    def __init__( ??? ):
        #
        # Escribir código aquí.
        #

    def __str__(self):
        #
        # Escribir código aquí.
        #

    def next_second(self):
        #
        # Escribir código aquí.
        #

    def prev_second(self):
```

```
        #
        # Escribir código aquí.
        #


timer = Timer(23, 59, 59)
print(timer)
timer.next_second()
print(timer)
timer.prev_second()
print(timer)
```

Ejecuta tu código y comprueba si el resultado es el mismo que el nuestro.

Salida Esperada

```
23:59:59
00:00:00
23:59:59
```

# ejercicio

## Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Definir y usar variables de instancia.
- Definir y usar métodos.

## Escenario

Tu tarea es implementar una clase llamada Weeker. Sí, tus ojos no te engañan, este nombre proviene del hecho de que los objetos de esta clase podrán almacenar y manipular los días de la semana.

El constructor de la clase acepta un argumento: una cadena. La cadena representa el nombre del día de la semana y los únicos valores aceptables deben provenir del siguiente conjunto:

```
Lun Mar Mie Jue Vie Sab Dom
```

Invocar al constructor con un argumento desde fuera de este conjunto debería generar la excepción WeekDayError (defínela tu mismo; no te preocupes, pronto hablaremos sobre la naturaleza objetiva de las excepciones). La clase debe proporcionar las siguientes facilidades:

- Los objetos de la clase deben ser «imprimibles», es decir, deben poder convertirse implícitamente en cadenas de la misma forma que los argumentos del constructor.
- La clase debe estar equipada con métodos de un parámetro llamados add_days(n) y subtract_days(n), siendo n un número entero que actualiza el día de la semana almacenado dentro del objeto mediante el número de días indicado, hacia adelante o hacia atrás.
- Todas las propiedades del objeto deben ser privadas.

Completa la plantilla que te proporcionamos en el editor, ejecuta su código y verifica si tu salida se ve igual que la nuestra.

```
class WeekDayError(Exception):
```

```
    pass


class Weeker:
    #
    # Escribir código aquí.
    #

    def __init__(self, day):
        #
        # Escribir código aquí.
        #

    def __str__(self):
        #
        # Escribir código aquí.
        #

    def add_days(self, n):
        #
        # Escribir código aquí.
        #

    def subtract_days(self, n):
        #
        # Escribir código aquí.
        #


try:
    weekday = Weeker('Lun')
    print(weekday)
    weekday.add_days(15)
    print(weekday)
    weekday.subtract_days(23)
    print(weekday)
    weekday = Weeker('Lun')
except WeekDayError:
    print("Lo siento, no puedo atender tu solicitud.")
```

Salida Esperada

```
Lun
Mar
Dom
Lo siento, no puedo atender tu solicitud.
```

# Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Definir y usar variables de instancia.
- Definir y usar métodos.

## Escenario

Visitemos un lugar muy especial: un plano con el sistema de coordenadas cartesianas (puedes obtener más información sobre este concepto aquí: https://en.wikipedia.org/wiki/Cartesian_coordinate_system).

Cada punto ubicado en el plano puede describirse como un par de coordenadas habitualmente llamadas x y y. Queremos que escribas una clase en Python que almacene ambas coordenadas como números flotantes. Además, queremos que los objetos de esta clase evalúen las distancias entre cualquiera de los dos puntos situados en el plano.

La tarea es bastante fácil si empleas la función denominada hypot() (disponible a través del módulo math) que evalúa la longitud de la hipotenusa de un triángulo rectángulo (más detalles aquí: https://en.wikipedia.org/wiki/Hypotenuse) y aquí: https://docs.python.org/3.7/library/math.html#trigonometric-functions.

Así es como imaginamos la clase:

- Se llama Point.
- Su constructor acepta dos argumentos (x y y respectivamente), ambos por defecto se igualan a cero.
- Todas las propiedades deben ser privadas.
- La clase contiene dos métodos sin parámetros llamados getx() y gety(), que devuelven cada una de las dos coordenadas (las coordenadas se almacenan de forma privada, por lo que no se puede acceder a ellas directamente desde el objeto).
- La clase proporciona un método llamado distance_from_xy(x,y), que calcula y devuelve la distancia entre el punto almacenado dentro del objeto y el otro punto dado en un par de números flotantes.
- La clase proporciona un método llamado distance_from_point(point), que calcula la distancia (como el método anterior), pero la ubicación del otro punto se da como otro objeto de clase Point.

Completa la plantilla que te proporcionamos en el editor, ejecuta tu código y verifica si tu salida se ve igual que la nuestra.

```
import math


class Point:
    def __init__(self, x=0.0, y=0.0):
        #
        # Escribir el código aquí.
        #

    def getx(self):
        #
        # Escribir el código aquí.
        #

    def gety(self):
        #
        # Escribir el código aquí.
        #

    def distance_from_xy(self, x, y):
        #
        # Escribir el código aquí.
        #

    def distance_from_point(self, point):
```

```
        #
        # Escribir el código aquí.
        #


point1 = Point(0, 0)
point2 = Point(1, 1)
print(point1.distance_from_point(point2))
print(point2.distance_from_xy(2, 0))
```

Salida esperada

```
1.4142135623730951
1.4142135623730951
```

# ejercicio

## Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Emplear composición.

## Escenario

Ahora vamos a colocar la clase *Point* (ver Lab 3.4.1.14) dentro de otra clase. Además, vamos a poner tres puntos en una clase, lo que nos permitirá definir un triángulo.¿Cómo podemos hacerlo?

La nueva clase se llamará Triangle y esto es lo que queremos:

- El constructor acepta tres argumentos - todos ellos son objetos de la clase Point.
- Los puntos se almacenan dentro del objeto como una lista privada
- La clase proporciona un método sin parámetros llamado perimeter(), que calcula el perímetro del triángulo descrito por los tres puntos; el perímetro es la suma de todas las longitudes de los lados (lo mencionamos para que conste, aunque estamos seguros de que tú mismo lo conoces perfectamente).

Completa la plantilla que te proporcionamos en el editor, ejecuta tu código y verifica si tu salida se ve igual que la nuestra.

```
import math


class Point:
    #
    # El código copiado del laboratorio anterior.
    #


class Triangle:
    def __init__(self, vertice1, vertice2, vertice3):
        #
        # Escribir el código aquí.
        #
```

```
    def perimeter(self):
        #
        # Escribir el código aquí.
        #


triangle = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
print(triangle.perimeter())
```

A continuación puedes copiar el código de la clase Point, el cual se utilizo en el laboratorio anterior:

```
class Point:
    def __init__(self, x=0.0, y=0.0):
        self.__x = x
        self.__y = y
```

Salida esperada

```
3.414213562373095
```

## Herencia: ¿por qué y cómo?

Antes de comenzar a hablar sobre la herencia, queremos presentar un nuevo y práctico mecanismo utilizado por las clases y los objetos de Python: es **la forma en que el objeto puede presentarse a si mismo**.

Comencemos con un ejemplo.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy


sun = Star("Sol", "Vía Láctea")
print(sun)
```

El programa imprime solo una línea de texto, que en nuestro caso es:

```
<__main__.Star object at 0x7f1074cc7c50>
```

Si ejecutas el mismo código en tu computadora, verás algo muy similar, aunque el número hexadecimal (la subcadena que comienza con 0x) será diferente, ya que es solo un identificador de objeto interno utilizado por Python, y es poco probable que aparezca igual cuando se ejecuta el mismo código en un entorno diferente.

Como puedes ver, la impresión aquí no es realmente útil, y algo más específico, es preferible.

Afortunadamente, Python ofrece tal función.

Cuando Python necesita que alguna clase u objeto deba ser presentado como una cadena (es recomendable colocar el objeto como argumento en la invocación de la función `print()`), intenta invocar un método llamado `__str__()` del objeto y emplear la cadena que devuelve.

El método por default `__str__()` devuelve la cadena anterior: fea y poco informativa. Puedes cambiarlo

**definiendo tu propio método**.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

    def __str__(self):
        return self.name + ' en ' + self.galaxy


sun = Star("Sol", "Vía Láctea")
print(sun)
```

El método nuevo `__str__()` genera una cadena que consiste en los nombres de la estrella y la galaxia, nada especial, pero los resultados de impresión se ven mejor ahora, ¿no?

El término herencia es más antiguo que la programación de computadoras, y describe la práctica común de pasar diferentes bienes de una persona a otra después de la muerte de esa persona. El término, cuando se relaciona con la programación de computadoras, tiene un significado completamente diferente.


Definamos el término para nuestros propósitos:

La herencia es una práctica común (en la programación de objetos) de **pasar atributos y métodos de la superclase (definida y existente) a una clase recién creada, llamada subclase**.

En otras palabras, la herencia **es una forma de construir una nueva clase, no desde cero, sino utilizando un repertorio de rasgos ya definido**. La nueva clase hereda (y esta es la clave) todo el equipamiento ya existente, pero puedes agregar algo nuevo si es necesario.

Gracias a eso, es posible **construir clases más especializadas (más concretas)** utilizando algunos conjuntos de reglas y comportamientos generales predefinidos.

El factor más importante del proceso es la relación entre la superclase y todas sus subclases (nota: si B es una subclase de A y C es una subclase de B, esto también significa que C es una subclase de A, ya que la relación es totalmente transitiva).

Aquí se presenta un ejemplo muy simple de **herencia de dos niveles**:

```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass
```

Todas las clases presentadas están vacías por ahora, ya que te mostraremos cómo funcionan las relaciones mutuas entre las superclases y las subclases. Las llenaremos con contenido pronto.

Podemos decir que:

- La clase Vehicle es la superclase para clases LandVehicle y TrackedVehicle.
- La clase LandVehicle es una subclase de Vehicle y la superclase de TrackedVehicle al mismo tiempo.
- La clase TrackedVehicle es una subclase tanto de Vehicle y LandVehicle.

El conocimiento anterior proviene de la lectura del código (en otras palabras, lo sabemos porque podemos verlo).

¿Python sabe lo mismo? ¿Es posible preguntarle a Python al respecto? Sí lo es.

## Herencia: issubclass()

Python ofrece una función que es capaz de **identificar una relación entre dos clases**, y aunque su diagnóstico no es complejo, puede **verificar si una clase particular es una subclase de cualquier otra clase**.

Así es como se ve:

```
issubclass(ClassOne, ClassTwo)
```

La función devuelve *True* si ClassOne es una subclase de ClassTwo, y *False* de lo contrario.

Vamos a verlo en acción, puede sorprenderte.

```
class Vehicle:
    pass


```

```
class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()
```

Hay dos bucles anidados. Su propósito es **verificar todos los pares de clases ordenadas posibles y que imprima los resultados de la verificación para determinar si el par coincide con la relación subclase-superclase**.

Ejecuta el código. El programa produce el siguiente resultado:

```
True    False    False
True    True    False
True    True    True
```

Hagamos que el resultado sea más legible:

| ↓ es una subclase de → | Vehicle | LandVehicle | TrackedVehicle |
|---|---|---|---|
| **Vehicle** | True | False | False |
| **LandVehicle** | True | True | False |
| **TrackedVehicle** | True | True | True |

Existe una observación importante que hacer: **cada clase se considera una subclase de sí misma**.

## Herencia: isinstance()

Como ya sabes, **un objeto es la encarnación de una clase**. Esto significa que el objeto es como un pastel horneado usando una receta que se incluye dentro de la clase.

Esto puede generar algunos problemas.

Supongamos que tienes un pastel (por ejemplo, resultado de un argumento pasado a tu función). Deseas saber que receta se ha utilizado para prepararlo. ¿Por qué? Porque deseas saber que esperar de él, por ejemplo, si contiene nueces o no, lo cual es información crucial para ciertas personas.

Del mismo modo, puede ser crucial si el objeto tiene (o no tiene) ciertas características. En otras palabras, **si es un objeto de cierta clase o no**.

Tal hecho podría ser detectado por la función llamada `isinstance()`:

```
isinstance(objectName, ClassName)
```

La función devuelve *True* si el objeto es una instancia de la clase, o *False* de lo contrario.

**Ser una instancia de una clase significa que el objeto (el pastel) se ha preparado utilizando una receta contenida en la clase o en una de sus superclases.**

No lo olvides: si una subclase contiene al menos las mismas características que cualquiera de sus superclases, significa que los objetos de la subclase pueden hacer lo mismo que los objetos derivados de la superclase, por lo tanto, es una instancia de su clase de inicio y cualquiera de sus superclases.

```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


my_vehicle = Vehicle()
my_land_vehicle = LandVehicle()
my_tracked_vehicle = TrackedVehicle()

for obj in [my_vehicle, my_land_vehicle, my_tracked_vehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()
```

Hemos creado tres objetos, uno para cada una de las clases. Luego, usando dos bucles anidados, verificamos todos los pares posibles de clase de objeto para averiguar si los objetos son instancias de las clases.

Ejecuta el código.

Esto es lo que obtenemos:

```
True    False    False
True    True    False
True    True    True
```

Hagamos que el resultado sea más legible:

| ↓ es una instancia de → | Vehicle | LandVehicle | TrackedVehicle |
|---|---|---|---|
| my_vehicle | True | False | False |
| my_land_vehicle | True | True | False |
| my_tracked_vehicle | True | True | True |

## Herencia: el operador is

También existe un operador de Python que vale la pena mencionar, ya que se refiere directamente a los objetos: aquí está:

```
object_one is object_two
```

**El operador *is* verifica si dos variables**, en este caso (object_one y object_two) **se refieren al mismo objeto**.

No olvides que las variables no almacenan los objetos en sí, sino solo los identificadores que apuntan a la memoria interna de Python.

Asignar un valor de una variable de objeto a otra variable no copia el objeto, sino solo su identificador. Es por ello que un operador como *is* puede ser muy útil en ciertas circunstancias.

Echa un vistazo al código en el editor.

```
class SampleClass:
    def __init__(self, val):
        self.val = val


object_1 = SampleClass(0)
object_2 = SampleClass(2)
object_3 = object_1
object_3.val += 1

print(object_1 is object_2)
print(object_2 is object_3)
print(object_3 is object_1)
print(object_1.val, object_2.val, object_3.val)

string_1 = "Mary tenía un "
string_2 = "Mary tenía un corderito"
string_1 += "corderito"

print(string_1 == string_2, string_1 is string_2)
```

Analicémoslo:

- Existe una clase muy simple equipada con un constructor simple, que crea una sola propiedad. La clase se usa para instanciar dos objetos. El primero se asigna a otra variable, y su propiedad val se incrementa en uno.
- Luego, el operador is se aplica tres veces para verificar todos los pares de objetos posibles, y todos los valores de la propiedad val son mostrados en pantalla.
- La última parte del código lleva a cabo otro experimento. Después de tres tareas, ambas cadenas contienen los mismos textos, pero estos textos se almacenan en diferentes objetos.

El código imprime:

```
False
False
True
```

```
1 2 1
True False
```

Los resultados prueban que *object_1* y *object_3* son en realidad los mismos objetos, mientras que *string_1* y *string_2* no lo son, a pesar de que su contenido sea el mismo.

### Cómo Python encuentra propiedades y métodos

Ahora veremos como Python trata con los métodos de herencia.

```
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "Mi nombre es " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        Super.__init__(self, name)


obj = Sub("Andy")

print(obj)
```

Vamos a analizarlo:

- Existe una clase llamada *Super*, que define su propio constructor utilizado para asignar la propiedad del objeto, llamada *name*.
- La clase también define el método `__str__()`, lo que permite que la clase pueda presentar su identidad en forma de texto.
- La clase se usa luego como base para crear una subclase llamada *Sub*. La clase *Sub* define su propio constructor, que invoca el de la superclase. Toma nota de como lo hemos hecho: *Super.__init__(self, name)*.
- Hemos nombrado explícitamente la superclase y hemos apuntado al método para invocar a `__init__()`, proporcionando todos los argumentos necesarios.
- Hemos instanciado un objeto de la clase Sub y lo hemos impreso.

El código da como salida:

```
Mi nombre es Andy.
```

Nota: Como no existe el método `__str__()` dentro de la clase *Sub*, la cadena a imprimir se producirá dentro de la clase Super. Esto significa que el método `__str__()` ha sido heredado por la clase *Sub*.

```
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "Mi nombre es " + self.name + "."
```

```
class Sub(Super):
    def __init__(self, name):
        super().__init__(name)


obj = Sub("Andy")

print(obj)
```

Lo hemos modificado para mostrarte otro método de acceso a cualquier entidad definida dentro de la superclase.

En el ejemplo anterior, nombramos explícitamente la superclase. En este ejemplo, hacemos uso de la función super(), la cual **accede a la superclase sin necesidad de conocer su nombre**:

```
super().__init__(name)
```

La función `super()` crea un contexto en el que no tiene que (además, no debe) pasar el argumento propio al método que se invoca; es por eso que es posible activar el constructor de la superclase utilizando solo un argumento.

Nota: puedes usar este mecanismo no solo para **invocar al constructor de la superclase, pero también para obtener acceso a cualquiera de los recursos disponibles dentro de la superclase**.

Intentemos hacer algo similar, pero con propiedades (más precisamente con: **variables de clase**).

```
# Probando propiedades: variables de clase.
class Super:
    supVar = 1


class Sub(Super):
    subVar = 2


obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

Como puedes observar, la clase *Super* define una variable de clase llamada *supVar*, y la clase *Sub* define una variable llamada *subVar*.

Ambas variables son visibles dentro del objeto de clase Sub, es por ello que el código da como salida:

```
2
1
```

El mismo efecto se puede observar con **variables de instancia**, observa el segundo ejemplo en el editor.

```
# Probando propiedades: variables de instancia.
class Super:
    def __init__(self):
        self.supVar = 11
```

```
class Sub(Super):
    def __init__(self):
        super().__init__()
        self.subVar = 12


obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

El constructor de la clase *Sub* crea una variable de instancia llamada *subVar*, mientras que el constructor de *Super* hace lo mismo con una variable de nombre *supVar*. Al igual que el ejemplo anterior, ambas variables son accesibles desde el objeto de clase *Sub*.

La salida del programa es:

```
12
11
```

Nota: La existencia de la variable *supVar* obviamente está condicionada por la invocación del constructor de la clase *Super*. Omitirlo daría como resultado la ausencia de la variable en el objeto creado (pruébalo tu mismo).

Ahora es posible formular una declaración general que describa el comportamiento de Python.

Cuando intentes acceder a una entidad de cualquier objeto, Python intentará (en este orden):

- Encontrarla **dentro del objeto** mismo.
- Encontrarla **en todas las clases** involucradas en la línea de herencia del objeto de abajo hacia arriba.

Si ambos intentos fallan, una excepción (*AttributeError*) **será generada**.

La primera condición puede necesitar atención adicional. Como sabes, todos los objetos derivados de una clase en particular pueden tener diferentes conjuntos de atributos, y algunos de los atributos pueden agregarse al objeto mucho tiempo después de la creación del objeto.

El ejemplo en el editor resume esto en una **línea de herencia de tres niveles**.

```
class Level1:
    variable_1 = 100
    def __init__(self):
        self.var_1 = 101

    def fun_1(self):
        return 102


class Level2(Level1):
    variable_2 = 200
    def __init__(self):
        super().__init__()
        self.var_2 = 201

    def fun_2(self):
        return 202
```

```
class Level3(Level2):
    variable_3 = 300
    def __init__(self):
        super().__init__()
        self.var_3 = 301

    def fun_3(self):
        return 302


obj = Level3()

print(obj.variable_1, obj.var_1, obj.fun_1())
print(obj.variable_2, obj.var_2, obj.fun_2())
print(obj.variable_3, obj.var_3, obj.fun_3())
```

```
100 101 102
200 201 202
300 301 302
```

Todos los comentarios que hemos hecho hasta ahora están relacionados con **casos de herencia única**, cuando una subclase tiene exactamente una superclase. Esta es la situación más común (y también la recomendada).

Python, sin embargo, ofrece mucho más aquí. En las próximas lecciones te mostraremos algunos ejemplos de **herencia múltiple**.

**La herencia múltiple ocurre cuando una clase tiene más de una superclase**.

Sintácticamente, dicha herencia se presenta como una lista de superclases separadas por comas entre paréntesis después del nombre de la nueva clase, al igual que aquí:

```
class SuperA:
    var_a = 10
    def fun_a(self):
        return 11


class SuperB:
    var_b = 20
    def fun_b(self):
        return 21


class Sub(SuperA, SuperB):
    pass

obj = Sub()

print(obj.var_a, obj.fun_a())
print(obj.var_b, obj.fun_b())
```

La clase *Sub* tiene dos superclases: *SuperA* y *SuperB*. Esto significa que la clase *Sub* **hereda todos los bienes ofrecidos por ambas clases** *SuperA* y *SuperB*.

El código imprime:

```
10 11
20 21
```

Ahora es el momento de introducir un nuevo término - **overriding (anulación)**.

¿Qué crees que sucederá si más de una de las superclases define una entidad con un nombre en particular?

```
class Level1:
    var = 100
    def fun(self):
        return 101


class Level2(Level1):
    var = 200
    def fun(self):
        return 201


class Level3(Level2):
    pass


obj = Level3()

print(obj.var, obj.fun())
```

Tanto la clase, *Level1* y *Level2* definen un método llamado `fun()` y una propiedad llamada *var*. ¿Significará esto el objeto de la claseLevel3 podrá acceder a dos copias de cada entidad? De ningún modo.

**La entidad definida después (en el sentido de herencia) anula la misma entidad definida anteriormente**. Es por eso que el código produce el siguiente resultado:

```
200 201
```

Como puedes ver, la variable de clase *var* y el método `fun()` de la clase *Level2* anula las entidades de los mismos nombres derivados de la clase *Level1*.

Esta característica se puede usar intencionalmente para modificar el comportamiento predeterminado de las clases (o definido previamente) cuando cualquiera de tus clases necesite actuar de manera diferente a su ancestro.

¿Qué ocurre cuando una clase tiene dos ancestros que ofrecen la misma entidad y se encuentran en el mismo nivel? En otras palabras, ¿Qué se debe esperar cuando surge una clase usando herencia múltiple? Miremos lo siguiente.

```
class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"


class Right:
```

```
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"


class Sub(Left, Right):
    pass


obj = Sub()

print(obj.var, obj.var_left, obj.var_right, obj.fun())
```

La clase *Sub* hereda todos los bienes de dos superclases *Left* y *Right* [1)]

[1)]
estos nombres están destinados a ser significativos). No hay duda de que la variable de clase *var_right* proviene de la clase *Right*, y *var_left* proviene de la clase *Left* respectivamente. Esto es claro. Pero, ¿De donde proviene la variable *var*? ¿Es posible adivinarlo? El mismo problema se encuentra con el método `fun()` - ¿Será invocada desde *Left* o desde *Right*? Ejecutemos el programa: la salida será:

```
L LL RR Left
```

Esto prueba que ambos casos poco claros tienen una solución dentro de la clase *Left*. ¿Es esta una premisa suficiente para formular una regla general? Sí lo es. Podemos decir que **Python busca componentes de objetos** en el siguiente orden:

- Dentro del objeto mismo.
- En sus superclases, de abajo hacia arriba.
- Si hay más de una clase en una ruta de herencia, Python las escanea de izquierda a derecha.

¿Necesitas algo más? Simplemente haz una pequeña enmienda en el código, reemplaza:class Sub(Left, Right): con: class Sub(Right, Left):, luego ejecuta el programa nuevamente y observa qué sucede. ¿Qué ves ahora? Vemos:

```
R LL RR Right
```

¿Ves lo mismo o algo diferente? === Cómo construir una jerarquía de clases Construir una jerarquía de clases no es solo por amor al arte. Si divides un problema entre las clases y decides cual de ellas debe ubicarse en la parte superior y cual debe ubicarse en la parte inferior de la jerarquía, debes analizar cuidadosamente el problema, pero antes de mostrarte como hacerlo (y como no hacerlo), queremos resaltar un efecto interesante. No es nada extraordinario (es solo una consecuencia de las reglas generales presentadas anteriormente), pero recordarlo puede ser clave para comprender como funcionan algunos códigos y cómo se puede usar este efecto para construir un conjunto flexible de clases.

```
class One:
    def do_it(self):
        print("do_it de One")

    def doanything(self):
        self.do_it()


class Two(One):
    def do_it(self):
```

```
        print("do_it de Two")


one = One()
two = Two()

one.doanything()
two.doanything()
```

Analicémoslo:

- Existen dos clases llamadas *One* y *Two*, se entiende que *Two* es derivada de *One*. Nada especial. Sin embargo, algo es notable: el `método do_it()`.
- El método `do_it()` está **definido dos veces**: originalmente dentro de *One* posteriormente dentro de *Two*. La esencia del ejemplo radica en el hecho de que es **invocado solo una vez** dentro de One.

La pregunta es: ¿cuál de los dos métodos será invocado por las dos últimas líneas del código? La primera invocación parece ser simple, el invocar el método `doanything()` del objeto llamado *one* obviamente activará el primero de los métodos. La segunda invocación necesita algo de atención. También es simple si tienes en cuenta cómo Python encuentra los componentes de la clase. La segunda invocación ejecutará el método `do_it()` en la forma existente dentro de la clase *Two*, independientemente del hecho de que la invocación se lleva a cabo dentro de la clase One. En efecto, el código genera el siguiente resultado:

```
do_it from One
do_it from Two
```

Nota: la situación en la cual **la subclase puede modificar el comportamiento de su superclase (como en el ejemplo) se llama poliformismo**. La palabra proviene del griego (polys: «muchos, mucho» y morphe, «forma, forma»), lo que significa que una misma clase puede tomar varias formas dependiendo de las redefiniciones realizadas por cualquiera de sus subclases. El método, redefinido en cualquiera de las superclases, que cambia el comportamiento de la superclase, se llama **virtual**. En otras palabras, ninguna clase se da por hecho. El comportamiento de cada clase puede ser modificado en cualquier momento por cualquiera de sus subclases. Te mostraremos **como usar el poliformismo para extender la flexibilidad de la clase**.

```
import time

class TrackedVehicle:
    def control_track(left, stop):
        pass

    def turn(left):
        control_track(left, True)
        time.sleep(0.25)
        control_track(left, False)


class WheeledVehicle:
    def turn_front_wheels(left, on):
        pass

    def turn(left):
        turn_front_wheels(left, True)
        time.sleep(0.25)
        turn_front_wheels(left, False)
```

¿Se parece a algo? Sí, por supuesto que lo hace. Se refiere al ejemplo que se muestra al comienzo del módulo cuando hablamos de los conceptos generales de la programación orientada a objetos. Puede parecer extraño, pero no utilizamos herencia en este ejemplo, solo queríamos mostrarte que no nos limita. Definimos dos clases

separadas capaces de producir dos tipos diferentes de vehículos terrestres. La principal diferencia entre ellos está en cómo giran. Un vehículo con ruedas solo gira las ruedas delanteras (generalmente). Un vehículo oruga tiene que detener una de las pistas. ¿Puedes seguir el código?

- Un vehículo oruga realiza un giro deteniéndose y moviéndose en una de sus pistas (esto lo hace el método control_track() el cual se implementará más tarde).
- Un vehículo con ruedas gira cuando sus ruedas delanteras giran (esto lo hace el método turn_front_wheels()).
- El método turn() utiliza el método adecuado para cada vehículo en particular.

¿Puedes detectar **el error del código**? Los métodos `turn()` son muy similares como para dejarlos en esta forma. Vamos a reconstruir el código: vamos a presentar una superclase para reunir todos los aspectos similares de los vehículos, trasladando todos los detalles a las subclases.

```
import time

class Vehicle:
    def change_direction(left, on):
        pass

    def turn(left):
        change_direction(left, True)
        time.sleep(0.25)
        change_direction(left, False)


class TrackedVehicle(Vehicle):
    def control_track(left, stop):
        pass

    def change_direction(left, on):
        control_track(left, on)


class WheeledVehicle(Vehicle):
    def turn_front_wheels(left, on):
        pass

    def change_direction(left, on):
        turn_front_wheels(left, on)
```

Esto es lo que hemos hecho:

- Definimos una superclase llamada *Vehicle*, la cual utiliza el método `turn()` para implementar un esquema para poder girar, mientras que el giro en si es realizado por `change_direction()`; nota: dicho método está vacío, ya que vamos a poner todos los detalles en la subclase (dicho método a menudo se denomina **método abstracto**, ya que solo demuestra alguna posibilidad que será instanciada más tarde).
- Definimos una subclase llamada *TrackedVehicle* (nota: es derivada de la clase *Vehicle*) la cual instancia el método `change_direction()` utilizando el método denominado `control_track()`.
- Respectivamente, la subclase llamada *WheeledVehicle* hace lo mismo, pero usa el método `turn_front_wheels()` para obligar al vehículo a girar.

La ventaja más importante (omitiendo los problemas de legibilidad) es que esta forma de código te permite implementar un nuevo algoritmo de giro simplemente modificando el método turn(), lo cual se puede hacer en un solo lugar, ya que todos los vehículos lo obedecerán. Así es como el **el poliformismo ayuda al desarrollador a mantener el código limpio y consistente**. La herencia no es la única forma de construir

clases adaptables. Puedes lograr los mismos objetivos (no siempre, pero muy a menudo) utilizando una técnica llamada composición. **La composición es el proceso de componer un objeto usando otros objetos diferentes**. Los objetos utilizados en la composición entregan un conjunto de rasgos deseados (propiedades y/o métodos), podemos decir que actúan como bloques utilizados para construir una estructura más complicada. Puede decirse que:

- **La herencia extiende las capacidades de una clase** agregando nuevos componentes y modificando los existentes; en otras palabras, la receta completa está contenida dentro de la clase misma y todos sus ancestros; el objeto toma todas las pertenencias de la clase y las usa.
- **La composición proyecta una clase como contenedor** capaz de almacenar y usar otros objetos (derivados de otras clases) donde cada uno de los objetos implementa una parte del comportamiento de una clase.

Permítenos ilustrar la diferencia usando los vehículos previamente definidos. El enfoque anterior nos condujo a una jerarquía de clases en la que la clase más alta conocía las reglas generales utilizadas para girar el vehículo, pero no sabía cómo controlar los componentes apropiados (ruedas o pistas). Las subclases implementaron esta capacidad mediante la introducción de mecanismos especializados. Hagamos (casi) lo mismo, pero usando composición. La clase, como en el ejemplo anterior, sabe cómo girar el vehículo, pero el giro real lo realiza un objeto especializado almacenado en una propiedad llamada *controlador*. El *controlador* es capaz de controlar el vehículo manipulando las partes relevantes del vehículo.

```
import time

class Tracks:
    def change_direction(self, left, on):
        print("pistas: ", left, on)


class Wheels:
    def change_direction(self, left, on):
        print("ruedas: ", left, on)


class Vehicle:
    def __init__(self, controller):
        self.controller = controller

    def turn(self, left):
        self.controller.change_direction(left, True)
        time.sleep(0.25)
        self.controller.change_direction(left, False)


wheeled = Vehicle(Wheels())
tracked = Vehicle(Tracks())

wheeled.turn(True)
tracked.turn(False)
```

Existen dos clases llamadas *Tracks* y *Wheels*, ellas saben como controlar la dirección del vehículo. También hay una clase llamada *Vehicle* que puede usar cualquiera de los controladores disponibles (los dos ya definidos o cualquier otro definido en el futuro): el *controlador* se pasa a la clase durante la inicialización. De esta manera, la capacidad de giro del vehículo se compone de un objeto externo, no implementado dentro de la clase *Vehicle*. En otras palabras, tenemos un vehículo universal y podemos instalar pistas o ruedas en él. El código produce el siguiente resultado:

```
ruedas:  True True
```

```
pistas:  True False
tracks:  False True
tracks:  False False
```

=== Herencia simple frente a herencia múltiple Como ya sabes, no hay obstáculos para usar la herencia múltiple en Python. Puedes derivar cualquier clase nueva de más de una clase definida previamente. Solo hay un «pero». El hecho de que puedas hacerlo no significa que tengas que hacerlo. No olvides que:

- Una sola clase de herencia siempre es más simple, segura y fácil de entender y mantener.
- La herencia múltiple siempre es arriesgada, ya que tienes muchas más oportunidades de cometer un error al identificar estas partes de las superclases que influirán efectivamente en la nueva clase.
- La herencia múltiple puede hacer que la anulación sea extremadamente difícil; además, el emplear la función super() se vuelve ambiguo.
- La herencia múltiple viola el **principio de responsabilidad única** (mas detalles aquí: https://en.wikipedia.org/wiki/Single_responsibility_principle) ya que forma una nueva clase de dos (o más) clases que no saben nada una de la otra.
- Sugerimos encarecidamente la herencia múltiple como la última de todas las posibles soluciones: si realmente necesitas las diferentes funcionalidades que ofrecen las diferentes clases, la composición puede ser una mejor alternativa.

=== ¿Qué es el Orden de Resolución de Métodos (MRO) y por qué no todas las herencias tienen sentido? MRO, en general, es una forma (puedes llamarlo una **estrategia**) en la que un lenguaje de programación en particular escanea la parte superior de la jerarquía de una clase para encontrar el método que necesita actualmente. Vale la pena enfatizar que los diferentes lenguajes usan MROs levemente (o incluso completamente) diferentes. Python es único en este aspecto y sus costumbres son un poco específicas. Te mostraremos cómo funciona el MRO de Python en dos casos peculiares que son ejemplos claros de problemas que pueden ocurrir cuando intentas usar la herencia múltiple de manera demasiado imprudente. Comencemos con un fragmento que inicialmente puede parecer simple.

```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Middle):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

Estamos seguros de que si analizas el fragmento tu mismo, no verás ninguna anomalía en él. Sí, tienes toda la razón: parece claro y simple, y no genera preocupaciones. Si ejecutas el código, producirá el siguiente resultado predecible:

```
bottom
middle
```

```
top
```

Sin sorpresas hasta ahora. Hagamos un pequeño cambio en este código. Echa un vistazo:

```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Middle, Top):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

¿Puedes ver la diferencia? Está escondida en esta línea:

```
class Bottom(Middle, Top):
```

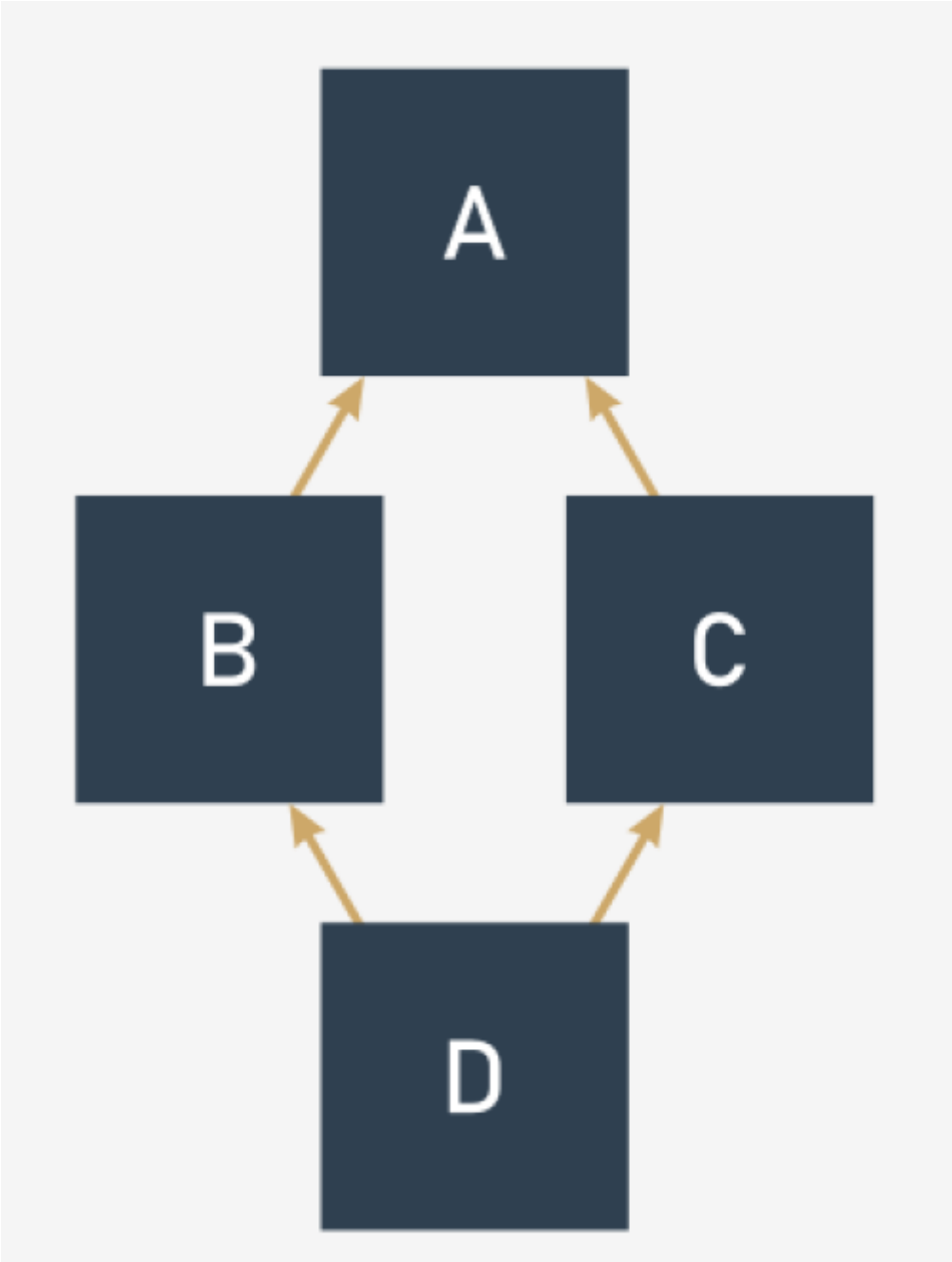
De esta manera exótica, hemos convertido un código muy simple con una clara ruta de herencia única en un misterioso acertijo de herencia múltiple. «¿Es válido?» Te puedes preguntar. Sí lo es. «¿Cómo es eso posible?» te preguntas, esperamos que realmente sientas la necesidad de hacer esta pregunta. Como puedes ver, el orden en el que se enumeran las dos superclases entre paréntesis cumple con la estructura del código: la clase *Middle* precede a la clase *Top*, justo como en la ruta de herencia real. A pesar de su rareza, la muestra es correcta y funciona como se esperaba, pero debe indicarse que esta notación no aporta ninguna funcionalidad nueva ni significado adicional. Modifiquemos el código una vez más; ahora intercambiaremos ambos nombres de superclase en la definición de clase Bottom. Así es como se ve el fragmento de código ahora:

```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Top, Middle):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

Para anticiparnos a tu pregunta, diremos que esta enmienda ha estropeado el código y ya no se ejecutará. Qué pena. El orden que intentamos forzar (Top, Middle) es incompatible con la ruta de herencia que se deriva de la estructura del código. A Python no le gustará. Esto es lo que veremos:

```
TypeError: Cannot create a consistent method resolution order (MRO) for bases Top, Middle
```

Creemos que el mensaje habla por sí solo. El MRO de Python no se puede doblar ni violar, no solo porque esa es la forma en que funciona Python, sino también porque es una regla que debes obedecer. === El Problema del Diamante El segundo ejemplo del espectro de problemas que posiblemente pueden surgir de la herencia múltiple está ilustrado por un problema clásico llamado **problema del diamante**. El nombre refleja la forma del diagrama de herencia; observa la imagen:




- Existe la superclase superior llamada A.
- Existen dos subclases derivadas de A: B y C.
- También está la subclase inferior llamada D, derivada de B y C (o C y B, ya que estas dos variantes significan cosas diferentes en Python).

¿Puedes ver el diamante ahí?

```
class A:
    pass


class B(A):
    pass


class C(A):
    pass


class D(B, C):
    pass


d = D()
```

La misma estructura, pero expresada en Python. Algunos lenguajes de programación no permiten la herencia múltiple en absoluto y, como consecuencia, no te permitirán construir un diamante; este es el camino que Java y C# han elegido seguir desde sus orígenes. Python, sin embargo, ha elegido una ruta diferente: permite la herencia múltiple y no le importa si escribe y ejecuta código como el del editor. Pero no te olvides del MRO: siempre está a cargo. Reconstruyamos nuestro ejemplo de la página anterior para hacerlo más parecido a un diamante, como se muestra a continuación:

```
class Top:
    def m_top(self):
        print("top")


class Middle_Left(Top):
    def m_middle(self):
        print("middle_left")


class Middle_Right(Top):
    def m_middle(self):
        print("middle_right")


class Bottom(Middle_Left, Middle_Right):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

Nota: ambas clases *Middle* definen **un método con el mismo nombre**: `m_middle()`. Introduce una pequeña incertidumbre en nuestra muestra, aunque estamos absolutamente seguros de que puedes responder la siguiente pregunta clave: ¿cuál de los dos métodos `m_middle()` se invocará realmente cuando la siguiente línea se ejecute?

```
Object.m_middle()
```

En otras palabras, qué verás en la pantalla: *middle_left* o *middle_right*? No es necesario que te apresures, ¡piénselo dos veces y toma en cuenta el MRO de Python! ¿Estás listo? Sí, tienes razón. La invocación activará el método `m_middle()`, que proviene de la clase *Middle_Left*. La explicación es simple: la clase aparece antes de *Middle_Right* en la lista de herencia de la clase *Bottom*. Si deseas asegurarte de que no haya dudas al respecto, intenta intercambiar estas dos clases en la lista y verifica los resultados. Si deseas experimentar algunas impresiones más profundas sobre la herencia múltiple y las piedras preciosas, intenta modificar nuestro fragmento y equipar la clase *Upper* con otro espécimen del método `m_middle ()` e investigua su comportamiento detenidamente. Como puedes ver, los diamantes pueden traer algunos problemas a tu vida, tanto los reales como los que ofrece Python.