Guía para Principiantes de la Programación Orientada a Objetos (POO) en Python

KINSTA

La programación es un arte. Y al igual que en el arte, la selección de los pinceles y pinturas adecuados es esencial para producir las mejores obras. La programación orientada a objetos (POO – Object-Oriented Programming) en Python es una de esas habilidades.

La elección del lenguaje de programación adecuado es una parte crucial de cualquier proyecto, y puede conducir a un desarrollo fluido y agradable o a una completa pesadilla. Por lo tanto, lo mejor es que utilices el lenguaje que mejor se adapte a tu caso de uso.

Esa es la principal razón para aprender programación orientada a objetos en Python, que además es uno de los lenguajes de programación más populares.

¡Vamos a aprender!

Un ejemplo de programa en Python

Antes de entrar en materia, vamos a plantear una pregunta: ¿has escrito alguna vez un programa en Python como el siguiente?

```
secret_number = 20

while True:
   number = input('Guess the number: ')

   try:
       number = int(number)
   except:
       print('Sorry that is not a number')
       continue

   if number != secret_number:
       if number > secret_number:
           print(number, 'is greater than the secret number')

       elif number < secret_number:
           print(number, 'is less than the secret number')
   else:
       print('You guessed the number:', secret_number)
       break
```

Este código es un simple adivinador de números. Intenta copiarlo en un archivo Python y ejecutarlo en tu sistema. Cumple perfectamente su propósito.

Pero aquí surge un gran problema: ¿y si te pedimos que implementes una nueva función? Podría ser algo sencillo, por ejemplo:

«Si la entrada es un múltiplo del número secreto, da una pista al usuario».

El programa se volvería rápidamente complejo y pesado al aumentar el número de funciones y, por tanto, el número total de condicionales anidados.

Ese es precisamente el problema que intenta resolver la programación orientada a objetos.

Requisitos para aprender Python OOP

Antes de adentrarte en la programación orientada a objetos, te recomendamos encarecidamente que tengas unos conocimientos básicos de Python.

Clasificar los temas considerados «básicos» puede ser difícil. Por ello, hemos diseñado una hoja de trucos con los principales conceptos necesarios para aprender programación orientada a objetos en Python.

- **Variable:** Nombre simbólico que apunta a un objeto específico (veremos qué significan **los objetos** a lo largo del artículo).
- **Operadores aritméticos:** Suma (+), resta (-), multiplicación (*), división (/), división entera (//), módulo (%).
- **Tipos de datos incorporados:** Numéricos (enteros, flotantes, complejos), Secuencias (cadenas, listas, tuplas), Booleanos (Verdadero, Falso), Diccionarios y Conjuntos.
- **Expresiones booleanas:** Expresiones en las que el resultado es **True** o **False.**
- **Condicional:** Evalúa una expresión booleana y realiza algún proceso dependiendo del resultado. Se maneja mediante sentencias **if/else**.
- **Bucle:** Ejecución repetida de bloques de código. Pueden ser bucles **for** o **while**.
- **Funciones:** Bloque de código organizado y reutilizable. Se crean con la palabra clave **def**.
- **Argumentos:** Objetos que se pasan a una función. Por ejemplo: `sum([1, 2, 4])`
- **Ejecuta un script de Python**: Abre un terminal o línea de comandos y escribe «python <nombre del archivo>».
- **Abre un shell de Python**: Abre un terminal y escribe `python` o `python3` dependiendo de tu sistema.

Ahora que tienes estos conceptos muy claros, puedes avanzar en la comprensión de la programación orientada a objetos.

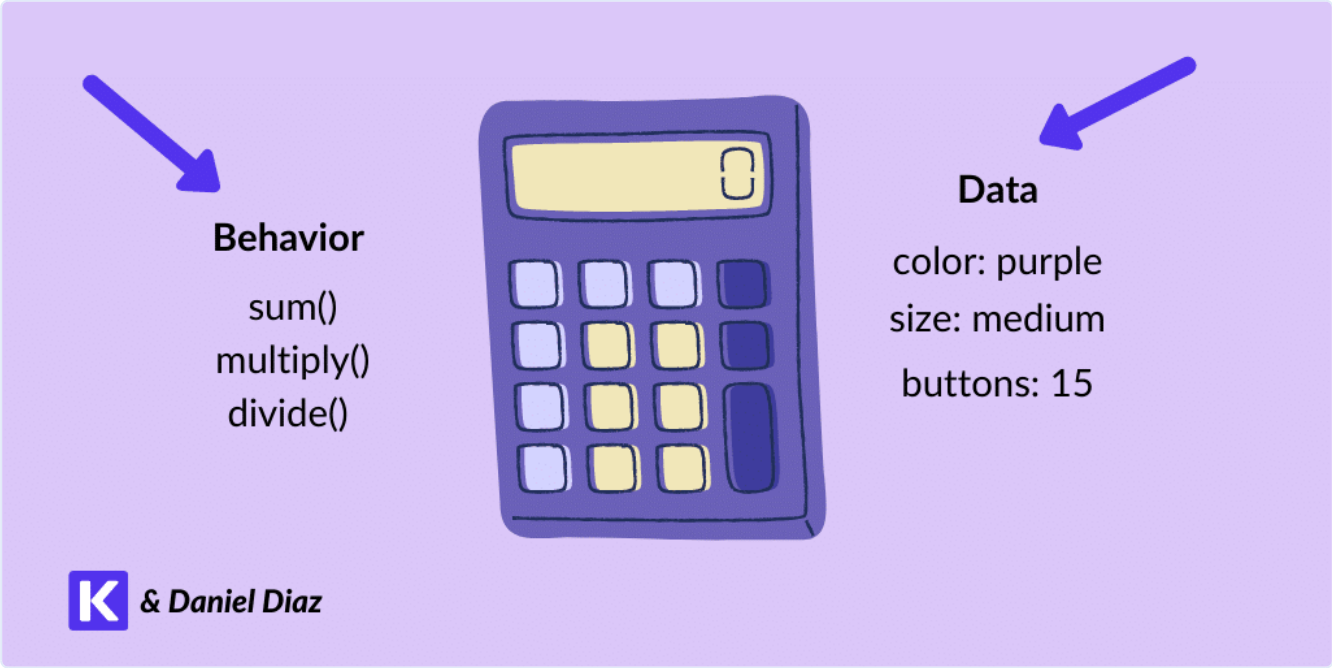
¿Qué es la programación orientada a objetos en Python?

La programación orientada a objetos (POO) es un paradigma de programación en el que podemos pensar en problemas complejos como objetos.

Un paradigma es una teoría que proporciona la base para resolver problemas.

Así que cuando hablamos de POO, nos referimos a un conjunto de conceptos y patrones que utilizamos para resolver problemas con objetos.

Un objeto en Python es una colección única de datos (atributos) y comportamiento (métodos). Puedes pensar en los objetos como cosas reales que te rodean. Por ejemplo, considera las calculadoras:



— Una calculadora puede ser un objeto.

Como puedes observar, los datos (atributos) son siempre sustantivos, mientras que los comportamientos (método) son siempre verbos.

Esta compartimentación es el concepto central de la programación orientada a objetos. Se construyen objetos que almacenan datos y contienen tipos específicos de funcionalidad.

¿Por qué utilizamos la programación orientada a objetos en Python?

La POO permite crear software seguro y fiable. Muchos marcos y bibliotecas de Python utilizan este paradigma para construir su código base. Algunos ejemplos son Django, Kivy, pandas, NumPy y TensorFlow.

Veamos las principales ventajas de usar OOP en Python.

Ventajas de la POO de Python

Las siguientes razones te harán optar por utilizar la programación orientada a objetos en Python.

Todos los lenguajes de programación modernos utilizan la POO

Este paradigma es independiente del lenguaje. Si aprendes POO en Python, podrás utilizarlo en lo siguiente:

- Java
- PHP (asegúrate de leer la comparación entre PHP y Python)
- Ruby
- Javascript
- C#
- Kotlin

Todos estos lenguajes están orientados a objetos de forma nativa o incluyen opciones para la funcionalidad orientada a objetos. Si quieres aprender cualquiera de ellos después de Python, será más fácil: encontrarás muchas similitudes entre los lenguajes que trabajan con objetos.

La POO te permite codificar más rápido

Codificar más rápido no significa escribir menos líneas de código. Significa que puedes implementar más funciones en menos tiempo sin comprometer la estabilidad de un proyecto.

La programación orientada a objetos te permite reutilizar el código mediante la implementación de la abstracción. Este principio hace que tu código sea más conciso y legible.

Como ya sabrás, los programadores pasan mucho más tiempo leyendo código que escribiéndolo. Es la razón por la que la legibilidad es siempre más importante que sacar características lo más rápido posible.

— La productividad disminuye con un código no legible

Más adelante verás más sobre el principio de abstracción.

La OOP te ayuda a evitar el código espagueti

¿Recuerdas el programa de adivinación de números del principio de este artículo?

Si sigues añadiendo funciones, tendrás muchas sentencias **if** anidadas en el futuro. Esta maraña de interminables líneas de código se llama código espagueti, y deberías evitarla en la medida de lo posible.

La programación orientada a objetos nos da la posibilidad de comprimir toda la lógica en objetos, evitando así largos trozos de **if's** anidados.

La POO mejora el análisis de cualquier situación

Una vez que tengas algo de experiencia con la POO, podrás pensar en los problemas como objetos pequeños y específicos.

Esta comprensión conduce a una rápida puesta en marcha del proyecto.

Programación estructurada frente a la programación orientada a objetos

La programación estructurada es el paradigma más utilizado por los principiantes porque es la forma más sencilla de construir un pequeño programa.

Se trata de ejecutar un programa Python de forma secuencial. Eso significa que le das al ordenador una lista de tareas y luego las ejecutas de arriba a abajo.

Veamos un ejemplo de programación estructurada con un programa de cafetería.

```
small = 2
regular = 5
big = 6
```

```
user_budget = input('What is your budget? ')

try:
   user_budget = int(user_budget)
except:
   print('Please enter a number')
   exit()

if user_budget > 0:
   if user_budget >= big:
       print('You can afford the big coffee')
       if user_budget == big:
           print('It\'s complete')
       else:
           print('Your change is', user_budget - big)
   elif user_budget == regular:
       print('You can afford the regular coffee')
       print('It\'s complete')
   elif user_budget >= small:
       print('You can buy the small coffee')
       if user_budget == small:
           print('It\'s complete')
       else:
           print('Your change is', user_budget - small)
```

El código anterior actúa como un vendedor de cafetería. Te pedirá un presupuesto y luego te «venderá» el mejor café que seas capaz de comprar.

Intenta ejecutarlo en la terminal. Se ejecutará paso a paso, dependiendo de tu entrada.

Este código funciona perfectamente, pero tenemos tres problemas:

1. Tiene mucha lógica repetida.
2. Utiliza muchos condicionales **if** anidados.
3. Será difícil de leer y modificar.

La POO se inventó como solución a todos estos problemas.

Veamos el programa anterior implementado con POO. No te preocupes si aún no lo entiendes. Es solo para comparar la programación estructurada y la programación orientada a objetos.

```
class Coffee:
        # Constructor
        def __init__(self, name, price):
                self.name = name
                self.price = float(price)
        def check_budget(self, budget):
                # Check if the budget is valid
                if not isinstance(budget, (int, float)):
                        print('Enter float or int')
                        exit()
                if budget < 0:
                    print('Sorry you don\'t have money')
                    exit()
        def get_change(self, budget):
                return budget - self.price

        def sell(self, budget):
                self.check_budget(budget)
                if budget >= self.price:
                        print(f'You can buy the {self.name} coffee')
                        if budget == self.price:
                                print('It\'s complete')
                        else:
                                print(f'Here is your change {self.get_cha

                        exit('Thanks for your transaction')
```

Nota: Todos los conceptos siguientes se explicarán con mayor profundidad a lo largo del artículo.

El código anterior representa una **clase** llamada «Coffee». Tiene dos atributos – «Name» y «Price» – y ambos se utilizan en los métodos. El método principal es «Sell», que procesa toda la lógica necesaria para completar el proceso de venta.

Si intentas ejecutar esa clase, no obtendrás ninguna salida. Ocurre principalmente porque solo estamos declarando la «plantilla» para los cafés, no los cafés en sí.

Implementemos esa clase con el siguiente código:

```
small = Coffee('Small', 2)
regular = Coffee('Regular', 5)
big = Coffee('Big', 6)

try:
   user_budget = float(input('What is your budget? '))
except ValueError:
   exit('Please enter a number')

for coffee in [big, regular, small]:
   coffee.sell(user_budget)
```

Aquí estamos haciendo **instancias,** u objetos de café, de la clase «Coffee», y luego llamando al método «sell» de cada café hasta que el usuario pueda pagar cualquier opción.

Obtendremos el mismo resultado con ambos enfoques, pero podemos ampliar la funcionalidad del programa mucho mejor con la POO.

A continuación se muestra una tabla comparativa entre la programación orientada a objetos y la programación estructurada:

OOP	Programación Estructurada
Más fácil de mantener	Difícil de mantener
No te repitas (DRY)	Código repetido en muchos lugares
Pequeños trozos de código reutilizados en muchos lugares	Una gran cantidad de código en pocos lugares
Enfoque por objetos	Enfoque de código de bloques
Más fácil de depurar	Más difícil de depurar
Gran curva de aprendizaje	Una curva de aprendizaje más sencilla
Utilizado en grandes proyectos	Optimizado para programas sencillos

Para concluir la comparación de paradigmas:

- Ninguno de los dos paradigmas es perfecto (la POO puede resultar abrumadora en proyectos sencillos).
- Estas son solo dos formas de resolver un problema; hay otras por ahí.
- La POO se utiliza en grandes bases de código, mientras que la programación estructurada es principalmente para proyectos sencillos.

Pasemos a los objetos incorporados en Python.

Todo es un objeto en Python

Te diremos un secreto: has estado usando OOP todo el tiempo sin darte cuenta.

Incluso cuando se utilizan otros paradigmas en Python, se siguen utilizando objetos para hacer casi todo.

Eso es porque, en Python, *todo* es un objeto.

Recuerda la definición de objeto: Un objeto en Python es una única colección de datos (atributos) y comportamiento (métodos).

Esto coincide con cualquier tipo de datos en Python.

Una cadena es una colección de datos (caracteres) y comportamientos (**upper()**, **lower()**, etc.). Lo mismo ocurre con los **enteros**, los **flotantes**, los **booleanos**, las **listas** y los

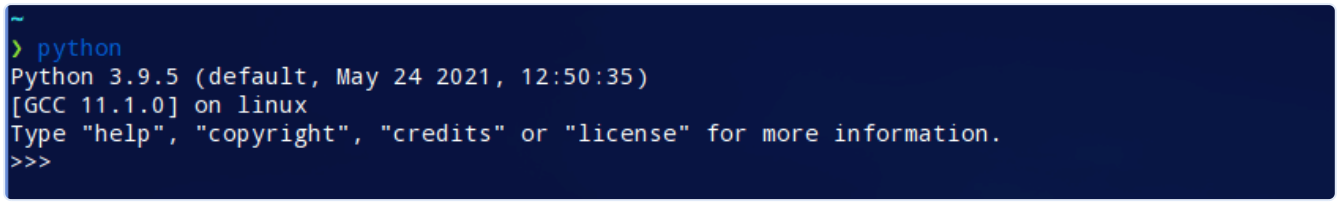
diccionarios.

Antes de continuar, repasemos el significado de los atributos y los métodos.

Atributos y métodos

Los atributos son **variables** internas dentro de los objetos, mientras que los métodos son **funciones** que producen algún comportamiento.

Vamos a hacer un simple ejercicio en el shell de Python. Puedes abrirlo escribiendo python o python3 en tu terminal.



— Shell de Python

Ahora, vamos a trabajar con el shell de Python para descubrir métodos y tipos.

```
>>> kinsta = 'Kinsta, Premium Application, Database, and Managed WordPres
>>> kinsta.upper()
'KINSTA, PREMIUM APPLICATION, DATABASE, AND MANAGED WORDPRESS HOSTING'
```

En la segunda línea, estamos llamando a un método de cadena, **upper()**. Devuelve el contenido de la cadena todo en mayúsculas. Sin embargo, no cambia la variable original.

```
>>> kinsta
'Kinsta, Premium Application, Database, and Managed WordPress hosting'
```

Profundicemos en las funciones valiosas cuando se trabaja con objetos.

La función **type()** permite obtener el tipo de un objeto. El «tipo» es la clase a la que pertenece el objeto.

```
>>> type(kinsta)
# class 'str'
```

La función **dir()** devuelve todos los atributos y métodos de un objeto. Vamos a probarlo con la variable **kinsta**.

```
>>> dir(kinsta)
['__add__', '__class__',  ..... 'upper', 'zfill']
```

Ahora, intenta imprimir algunos de los atributos ocultos de este objeto.

```
 >>> kinsta.__class__ # class ‘str’ e>
```

Esto devolverá la clase a la que pertenece el objeto **kinsta**. Así que podemos decir que lo único que devuelve la función **type** es el atributo __**class**__ de un objeto.

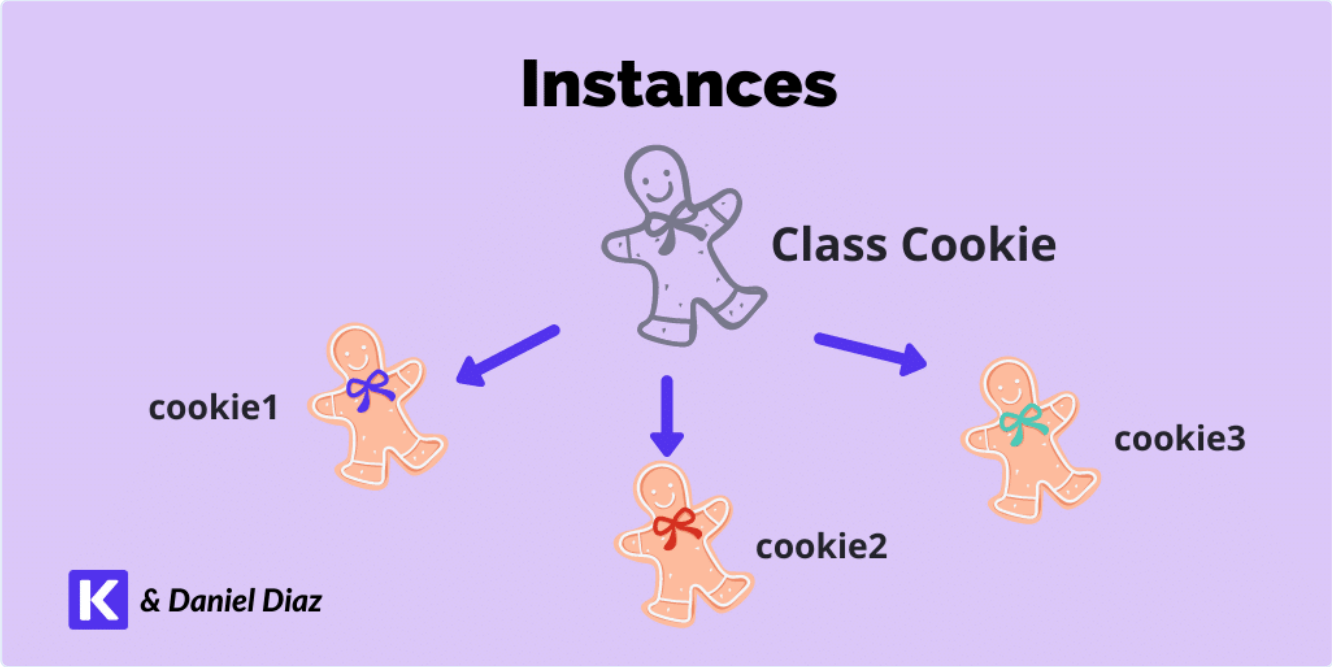
Puedes experimentar con todos los tipos de datos, descubriendo todos sus atributos y métodos directamente en el terminal. Puedes obtener más información sobre los tipos de datos incorporados en la documentación oficial.

Tu primer objeto en Python

Una **clase** es como una plantilla**.** Te permite crear objetos personalizados basados en los atributos y métodos que definas.

Puedes pensar en él como un **cortador de galletas** que modificas para hornear las galletas perfectas (objetos, no galletas de seguimiento), con características definidas: Forma, tamaño y otros.

Por otro lado, tenemos las **instancias.** Una instancia es un objeto individual de una clase, que tiene una dirección de memoria única.



— Instancias en Python

Ahora que sabes lo que son las clases y las instancias, ¡definamos algunas!

Para definir una clase en Python, se utiliza la palabra clave **class**, seguida de su nombre. En este caso, crearás una clase llamada **Cookie**.

Nota: En Python, utilizamos la convención de nombres en mayúsculas para nombrar las clases.

```
class Cookie:
 pass
```

Abre tu shell de Python y escribe el código anterior. Para crear una instancia de una clase, solo tienes que escribir su nombre y un paréntesis después. Es el mismo proceso que

invocar una función.

```
cookie1 = Cookie()
```

Enhorabuena: ¡acabas de crear tu primer objeto en Python! Puedes comprobar su id y tipo con el siguiente código:

```
id(cookie1)
140130610977040 # Unique identifier of the object

type(cookie1)
<class '__main__.Cookie'>
```

Como puedes ver, esta cookie tiene un identificador único en la memoria, y su tipo es **Cookie** .

También puedes comprobar si un objeto es una instancia de una clase con la función **isinstance()**.

```
isinstance(cookie1, Cookie)
# True
isinstance(cookie1, int)
# False
isinstance('a string', Cookie)
```

```
# False
```

Método constructor

El método **__init__()** también se llama «constructor». Es llamado por Python cada vez que instanciamos un objeto.

El constructor crea el estado inicial del objeto con el conjunto mínimo de parámetros que necesita para existir. Modifiquemos la clase **Cookie**, para que acepte parámetros en tu constructor.

```
class Cookie:
 # Constructor
 def __init__(self, name, shape, chips='Chocolate'):
  # Instance attributes
  self.name = name
  self.shape = shape
  self.chips = chips
```

En la clase **Cookie**, cada cookie debe tener un nombre, una forma y unas virutas. Hemos definido esta última como «Chocolate».

Por otro lado, **self** se refiere a la instancia de la clase (el objeto en sí).

Intenta pegar la clase en el shell y crea una instancia de la cookie como de costumbre.

```
cookie2 = Cookie()
# TypeError
```

Obtendrás un error. Esto se debe a que debes proporcionar el conjunto mínimo de datos que el objeto necesita para vivir – en este caso, el **nombre** y la **forma**, ya que hemos establecido **chips** y «Chocolate».

```
cookie2 = Cookie('Awesome cookie', 'Star')
```

Para acceder a los atributos de una instancia, debes utilizar la notación de puntos.

```
cookie2.name
# 'Awesome cookie'
cookie2.shape
# 'Star'
cookie2.chips
# 'Chocolate'
```

Por ahora, la clase **Cookie** no tiene nada demasiado jugoso. Vamos a añadir un método de ejemplo **bake()** para hacer las cosas más interesantes.

```
class Cookie:
 # Constructor
 def __init__(self, name, shape, chips='Chocolate'):
  # Instance attributes
  self.name = name
  self.shape = shape
  self.chips = chips

 # The object is passing itself as a parameter
 def bake(self):
  print(f'This {self.name}, is being baked with the shape {self.shape} an
  print('Enjoy your cookie!')
```

Para llamar a un método, utiliza la notación de puntos e invócalo como una función.

```
cookie3 = Cookie('Baked cookie', 'Tree')
cookie3.bake()
# This Baked cookie, is being baked with the shape Tree and chips of Choc
Enjoy your cookie!
```

Los 4 pilares de la OOP en Python

La programación orientada a objetos incluye cuatro pilares principales:

1. Abstracción

La abstracción oculta al usuario la funcionalidad interna de una aplicación. El usuario puede ser el cliente final u otros desarrolladores.

Podemos encontrar **abstracción** en nuestra vida cotidiana. Por ejemplo, sabes cómo usar tu teléfono, pero probablemente no sepas exactamente lo que ocurre dentro de él cada vez que abres una aplicación.

Otro ejemplo es el propio Python. Sabes cómo usarlo para construir software funcional, y puedes hacerlo aunque no entiendas el funcionamiento interno de Python.

Aplicar lo mismo al código permite reunir todos los objetos de un problema y **abstraer la** funcionalidad estándar en clases.

2. Herencia

La herencia nos permite definir múltiples **subclases** a partir de una clase ya definida.

El propósito principal es seguir el principio DRY. Podrás reutilizar mucho código implementando todos los componentes compartidos en **superclases**.

Puedes pensar en ello como el concepto de **herencia genética** en la vida real. Los hijos (subclases) son el resultado de la herencia entre dos padres (superclases). Heredan todas las características físicas (atributos) y algunos comportamientos comunes (métodos).

3. Polimorfismo

El polimorfismo nos permite modificar ligeramente los métodos y atributos de las **subclases** previamente definidas en la **superclase**.

El significado literal es «**muchas formas**«. Esto se debe a que construimos métodos con el mismo nombre pero con diferente funcionalidad.

Volviendo a la idea anterior, los niños también son un ejemplo perfecto de polimorfismo. Pueden heredar un comportamiento definido **get_hungry()** pero de una manera ligeramente diferente, por ejemplo, tener hambre cada 4 horas en lugar de cada 6.

4. Encapsulación

La encapsulación es el proceso en el que protegemos la integridad interna de los datos en una clase.

Aunque no hay una declaración **privada** en Python, se puede aplicar la encapsulación mediante el uso de mangling en Python. Existen métodos especiales llamados **getters** y **setters** que nos permiten acceder a atributos y métodos únicos.

Imaginemos una clase **Humana** que tiene un único atributo llamado _**altura**. Este atributo solo se puede modificar dentro de ciertas restricciones (es casi imposible ser más alto que 3 metros).

Construir una calculadora de resolución de formas de área

Una de las mejores cosas de Python es que nos permite crear una gran variedad de software, desde un programa CLI (interfaz de línea de comandos) hasta una compleja aplicación web.

Ahora que has aprendido los conceptos fundamentales de la programación orientada a objetos, es el momento de aplicarlos en un proyecto real.

Nota: Todo el código siguiente estará disponible dentro de este repositorio de GitHub. Una herramienta de revisión de código que nos ayuda a gestionar las versiones del código con Git.

Tu tarea es crear una calculadora de áreas de las siguientes formas:

- Cuadrado
- Rectángulo
- Triángulo
- Círculo
- Hexágono

Clase de base de la forma

En primer lugar, crea un archivo **calculator.py** y ábrelo. Como ya tenemos los objetos para trabajar, será fácil **abstraction** en una clase.

Puedes analizar las características comunes y descubrir que todas ellas son **formas 2D**. Por lo tanto, la mejor opción es crear una clase **Shape** con un método **get_area()** del que heredará cada forma.

Nota: Todos los métodos deben ser verbos. Eso es porque este método se llama **get_area()** y no **area()**.

```
class Shape:
 def __init__(self):
  pass

 def get_area(self):
  pass
```

El código anterior define la clase; sin embargo, todavía no hay nada interesante en ella.

Vamos a implementar la funcionalidad estándar de la mayoría de estas formas.

```
class Shape:
 def __init__(self, side1, side2):
  self.side1 = side1
  self.side2 = side2

 def get_area(self):
  return self.side1 * self.side2
```

```
    def __str__(self):
     return f'The area of this {self.__class__.__name__} is: {self.get_area(
```

Vamos a desglosar lo que estamos haciendo con este código:

- En el método __**init**__, estamos solicitando dos parámetros, **side1** y **side2**. Estos permanecerán como **atributos de la instancia**.
- La función **get_area()** devuelve el área de la forma. En este caso, utiliza la fórmula del área de un rectángulo, ya que será más fácil de implementar con otras formas.
- El método __**str**__**()** es un «método mágico» al igual que __**init**__**().** Permite modificar la forma en que se imprimirá una instancia.
- El atributo oculto **self.__class__.__name__** se refiere al nombre de la clase. Si estuvieras trabajando con una clase **Triangle**, ese atributo sería «Triangule».

Clase Rectángulo

Ya que implementamos la fórmula del área del Rectángulo, podríamos crear una simple clase **Rectangle** que no hace más que heredar de la clase **Shape**.

Para aplicar la **herencia** en Python, crearás una clase como de costumbre y rodearás la **superclase** de la que quieres heredar con paréntesis.

```
# Folded base class
class Shape: ...

class Rectangle(Shape): # Superclass in Parenthesis
 pass
```

Clase cuadrada

Podemos hacer una excelente aproximación al **polimorfismo** con la clase **Square**.

Recuerda que un cuadrado es solo un rectángulo cuyos cuatro lados son iguales. Eso significa que podemos utilizar la misma fórmula para obtener el área.

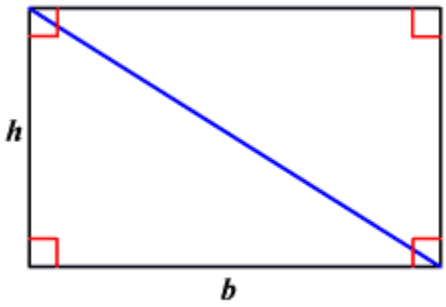
Podemos hacerlo modificando el método **init**, aceptando solo un **side** como parámetro, y pasando ese valor del lado al constructor de la clase **Rectangle**.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...

class Square(Rectangle):
 def __init__(self, side):
  super().__init__(side, side)
```

Como puedes ver, la función super pasa el parámetro side dos veces a la **superclase**. En otras palabras, está pasando **side** tanto como **side1** como **side2** al constructor previamente definido.

Clase de triángulo

Un triángulo es la mitad de grande que el rectángulo que lo rodea.



— Relación entre triángulos y rectángulos
(Fuente de la imagen: Varsity tutors).

Por lo tanto, podemos heredar de la clase **Rectangle** y modificar el método **get_area** para que coincida con la fórmula del área del triángulo, que es la mitad de la base multiplicada por la altura.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...

class Triangle(Rectangle):
 def __init__(self, base, height):
  super().__init__(base, height)

 def get_area(self):
  area = super().get_area()
  return area / 2
```

Otro caso de uso de la función **super()** es llamar a un método definido en la **superclase** y almacenar el resultado como una variable. Eso es lo que ocurre dentro del método

get_area().

Clase circular

Puedes encontrar el área del círculo con la fórmula **?r²**, donde **r** es el radio del círculo. Eso significa que tenemos que modificar el método **get_area()** para implementar esa fórmula.

Nota: Podemos importar el valor aproximado de ? desde el módulo matemático

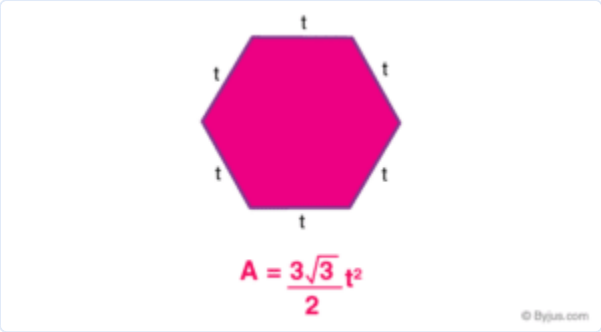
```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …

# At the start of the file
from math import pi

class Circle(Shape):
 def __init__(self, radius):
  self.radius = radius

 def get_area(self):
  return pi * (self.radius ** 2)
```

El código anterior define la clase **Circle**, que utiliza un constructor y métodos **get_area()** diferentes.

Aunque **Circle** hereda de la clase **Shape**, puedes redefinir cada método y atribuirlo a tu gusto.

Clase Hexágono Regular

Solo necesitamos la longitud de un lado de un hexágono regular para calcular su área. Es similar a la clase **Square**, donde solo pasamos un argumento al constructor.



— Fórmula del área del hexágono (Fuente de la imagen: BYJU'S)

Sin embargo, la fórmula es bastante diferente, e implica el uso de una raíz cuadrada. Por eso se utilizará la función **sqrt()** del módulo matemático.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …
class Circle(Shape): …

# Import square root
from math import sqrt

class Hexagon(Rectangle):
```

```
    def get_area(self):
     return (3 * sqrt(3) * self.side1 ** 2) / 2
```

Probar nuestras clases

Puedes entrar en un modo interactivo cuando ejecutas un archivo de Python utilizando un depurador. La forma más sencilla de hacerlo es utilizando la función de punto de interrupción incorporada.

Nota: Esta función solo está disponible en Python 3.7 o superior.

```
from math import pi, sqrt
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …
class Circle(Shape): …
class Hexagon(Rectangle): …

breakpoint()
```

Ahora, ejecuta el archivo Python y juega con las clases que has creado.

```
$ python calculator.py

(Pdb) rec = Rectangle(1, 2)(Pdb) print(rec)
```

```
The area of this Rectangle is: 2
(Pdb) sqr = Square(4)
(Pdb) print(sqr)
The area of this Square is: 16
(Pdb) tri = Triangle(2, 3)
(Pdb) print(tri)
The area of this Triangle is: 3.0
(Pdb) cir = Circle(4)
(Pdb) print(cir)
The area of this Circle is: 50.26548245743669
(Pdb) hex = Hexagon(3)
(Pdb) print(hex)
The area of this Hexagon is: 23.382685902179844
```

Desafío

Crear una clase con un método de **run** donde el usuario pueda elegir una forma y calcular su área.

Cuando hayas completado el reto, puedes enviar una solicitud de extracción al repositorio de GitHub o publicar tu solución en la sección de comentarios.

Resumen

La programación orientada a objetos es un paradigma en el que resolvemos problemas pensando en ellos como **objetos**. Si entiendes la POO de Python, también puedes aplicarla fácilmente en lenguajes como Java, PHP, Javascript y C#.

En este artículo, has aprendido sobre:

- El concepto de orientación a objetos en Python
- Ventajas de la programación orientada a objetos sobre la estructurada
- Fundamentos de la programación orientada a objetos en Python
- Concepto de **clases** y cómo utilizarlas en Python
- El **constructor** de una clase en Python

- **Métodos** y **atributos** en Python
- Los cuatro pilares de la POO
- Implementación de la **abstracción**, la **herencia** y el **polimorfismo** en un proyecto

Ahora te toca a ti.

Déjanos tu solución al desafío en los comentarios. Y no olvides consultar nuestra guía de comparación entre Python y PHP.