Clases y Objetos

Python es un lenguaje de programación orientado a objetos. Casi todo en Python es un objeto, con sus propiedades y métodos. Una clase es como un constructor de objetos, o un "*blueprint*" para crear objetos.

Crear una Clase

Para crear una clase, use la palabra clave `class`:

Ejemplo Cree una clase llamada `MyClass`, con una propiedad llamada `x`:

```
class MyClass:
    x = 5
```

Crear Objeto

Ahora podemos usar la clase llamada `MyClass` para crear objetos:

Ejemplo. Cree un objeto llamado `p1` e imprima el valor de `x`:

```
p1 = MyClass()
print(p1.x)
```

La función `__init__()`

Los ejemplos anteriores son clases y objetos en su forma más simple, y son no es realmente útil en aplicaciones de la vida real.

Para entender el significado de las clases tenemos que entender el `__init__()` incorporado función.

Todas las clases tienen una función llamada `__init__()`, que siempre se ejecuta cuando la clase está siendo iniciada.

Utilice la función `__init__()` para asignar valores a propiedades de objetos u otros operaciones que son necesarias para hacer cuando el objeto está siendo creado:

Ejemplo. Cree una clase llamada Persona, use la función `__init__()` para asignar valores para nombre y edad:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Nota: La función `__init__()` se llama automáticamente cada vez que la clase se utiliza para crear un nuevo objeto.

En el contexto de la programación orientada a objetos en Python, el término `self` se refiere a la instancia actual de la clase. Es un parámetro que se utiliza en los métodos de una clase para acceder a las variables y métodos del objeto.

El método `__init__` es un constructor especial en Python. Se llama automáticamente cuando se crea una nueva instancia de la clase. Recibe los parámetros `name` y `age`.

Dentro del constructor, `self` entra en juego. El término `self` se refiere a la instancia actual de la clase. Al asignar `self.name = name` y `self.age = age`, estás almacenando los valores `name` y `age` en los atributos `name` y `age` de la instancia actual de la clase `Person`.

La función `__str__()`

La función `__str__()` controla lo que debe devolverse cuando el objeto de clase se representa como una cadena.

Si la función `__str__()` no está establecida, la representación de cadena del objeto se devuelve:

Ejemplo. La representación de cadena de un objeto SIN la función `__str__()`:

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

p1 = Person("John", 36)

print(p1)
```

Ejemplo. La representación de cadena de un objeto CON la función `__str__()`:

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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

Métodos de Objeto

Los objetos también pueden contener métodos. Los métodos en los objetos son funciones que pertenecen al objeto.

Creemos un método en la clase `Person`:

Ejemplo. Inserte una función que imprima un saludo y ejecútelo en el objeto `p1`:

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Nota: El parámetro `self` es una referencia a la instancia actual de la clase, y se utiliza para acceder a las variables que pertenecen a la clase.

Ejemplo. Registro de perros con promedio de peso

```
class Perro:
    peso = 30

    def __init__(self, peso):
        self.peso = peso

    @classmethod
    def get_peso_promedio(cls):
        return cls.peso

labrador = Perro(25)
print(f'El peso de un perro labrador es {labrador.peso} kilos')
# El peso de un perro labrador es 25 kilos
print(f'El peso promedio de un perro es {Perro.get_peso_promedio()} kilos')
# El peso promedio de un perro es 30 kilos
```

El parámetro `self`

El parámetro `self` es una referencia a la instancia actual de la clase, y se utiliza para acceder a las variables que pertenecen a la clase.

No tiene que ser nombrado `self` , puede ser llamado como sea, pero tiene que ser el primer parámetro de cualquier función en la clase:

Ejemplo Usar las palabras `mysillyobject` y `abc` en lugar de `self`:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Modificar Propiedades de Objeto

Se pueden modificar propiedades en objetos como este:

Ejemplo. Establezca la edad de `p1` a 40:

```
p1.age = 40
```

Eliminar Propiedades de Objeto

Puede eliminar propiedades en objetos utilizando el del palabra clave `del`:

Ejemplo. Eliminar la propiedad `age` del objeto `p1`:

```
del p1.age
```

Eliminar Objetos

Puede eliminar objetos utilizando el del palabra clave `del`:

Ejemplo. Eliminar el objeto `p1`:

```
del p1
```

La sentencia `pass`

La declaración de una clase no puede quedar vacía, pero si por alguna razón es necesario dejar vacía dicha declaración, se puede emplear la palabra clave `pass`.

Ejemplo

```
class Person:
  pass
```

Herencia

La herencia nos permite definir una clase que hereda todos los métodos y propiedades de otra clase. La clase padre es la clase de la que se hereda, también

llamada clase base. La clase hija es la clase que hereda de otra clase, también llamada clase derivada.

Crear una clase padre

Cualquier clase puede ser una clase padre, por lo que la sintaxis es la misma que la de crear cualquier otra clase:

Ejemplo. Crea una clase llamada `Person`, con propiedades `firstnamey` `lastname`, y un método `printname`:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Crear una clase hija

Para crear una clase que herede la funcionalidad de otra clase, envíe la clase principal como parámetro al crear la clase secundaria:

Ejemplo. Cree una clase llamada `Student`, que heredará las propiedades y métodos de la clase `Person`:

```
class Student(Person):
  pass
```

Nota: Utilice la `pass` palabra clave cuando no desee agregar ninguna otra propiedad o método a la clase.

Ahora la clase `Student` tiene las mismas propiedades y métodos que la clase `Person`.

Ejemplo. Utilizar la clase `Student` para crear un objeto y luego ejecute el método `printname`:

```
x = Student("Mike", "Olsen")
x.printname()
```

Agregue la función init()

Hasta ahora hemos creado una clase hija que hereda las propiedades y métodos de su clase principal. Queremos agregar la función `__init__()` a la clase hija

(en lugar de la palabra clave `pass`).

Nota: La función `__init__()` se llama automáticamente cada vez que se utiliza la clase para crear un nuevo objeto.

Ejemplo. Añade la función `__init__()` a la clase `Student`:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

Cuando se agrega la función `__init__()`, la clase hija ya no heredará la función `__init__()` de la clase padre.

Nota: La función `__init__()` del hijo reemplaza la herencia de la función del padre `__init__()`.

Si se desea mantener la herencia de la función padre `__init__()`, agregue una llamada a la función padre `__init__()`:

Ejemplo.

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Ahora hemos agregado exitosamente la función `__init__()` y conservamos la herencia de la clase padre, y estamos listos para agregar funcionalidad en la función `__init__()`.

La función super()

Python también tiene una función super() que hará que la clase hija herede todos los métodos y propiedades de su clase padre:

Ejemplo

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

Al utilizar la función super(), no es necesario utilizar el nombre del elemento padre, ya que heredará automáticamente los métodos y propiedades de su padre.

Agregar propiedades

Ejemplo. Añade una propiedad llamada `graduationyear` a la clase `Student`:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

En el ejemplo siguiente, el año 2019 debe ser una variable y pasarse a la clase `Student` al crear objetos de estudiantes. Para ello, agregue otro parámetro en la función `__init__()`:

Ejemplo. Agregue un parámetro `year` y pase el año correcto al crear objetos:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Agregar métodos

Ejemplo. Agregue un método llamado `welcome` a la clase `Student`:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

Si agrega un método en la clase hija con el mismo nombre que una función en la clase padre, se anulará la herencia del método principal.

Referencias

- w3 Schools
- Cosas de Devs