Python Mágico

Ovidio Navarro Pazos Juan José Muñoz Noda
Jésus Armando Padrón Raveiro

June 2, 2024

Contents

1 Métodos mágicos

En Python, los métodos que comienzan y terminan con dos guiones bajos (__) se llaman métodos especiales o "métodos mágicos" (de "double underscore"). Estos métodos tienen significados especiales y permiten a los desarrolladores definir cómo se comportan los objetos de sus clases en ciertas situaciones. Ejemplos:

__init__

El método `__init__` se llama automáticamente cuando se crea una instancia de una clase. Sirve como constructor de la clase.

```
        class Persona:
                def __init__(self, nombre, edad):
                        self.nombre = nombre
                        self.edad = edad

                # Crear una instancia de la clase Persona
        p = Persona("John Doe", 30)
```

__str__ y __repr__

Estos métodos definen cómo se debe representar un objeto como una cadena.

- `__str__` se utiliza principalmente para crear una representación legible del objeto.
- `__repr__` se utiliza para crear una representación más técnica, útil para los desarrolladores.

```
        class Persona:
                def __init__(self, nombre, edad):
                        self.nombre = nombre
                        self.edad = edad

                def __str__(self):
                        return f"Persona({self.nombre}, {self.edad}
                             anos)"

                def __repr__(self):
                        return f"Persona(nombre={self.nombre!r},
                            edad={self.edad!r})"

        p = Persona("John Doe", 30)
        print(str(p))  # Persona(John Doe, 30 aos)
        print(repr(p)) # Persona(nombre='John Doe', edad=30)
```

__len__

El método `__len__` se utiliza para devolver la longitud de un objeto. Es particularmente útil para colecciones personalizadas.

```
        class MiLista:
                def __init__(self, *elementos):
                        self.elementos = list(elementos)

                def __len__(self):
                        return len(self.elementos)

        mi_lista = MiLista(1, 2, 3, 4)
        print(len(mi_lista))  # 4
```

__getitem__, __setitem__ y __delitem__

Estos métodos permiten definir cómo acceder, asignar y eliminar elementos usando la sintaxis de subíndice (por ejemplo, `objeto[indice]`).

```
        class MiLista:
                def __init__(self, *elementos):
                        self.elementos = list(elementos)

                def __getitem__(self, indice):
                        return self.elementos[indice]

                def __setitem__(self, indice, valor):
                        self.elementos[indice] = valor

                def __delitem__(self, indice):
                        del self.elementos[indice]

        mi_lista = MiLista(1, 2, 3, 4)
        print(mi_lista[2])  # 3
        mi_lista[2] = 10
        print(mi_lista[2])  # 10
        del mi_lista[2]
        print(mi_lista.elementos)  # [1, 2, 4]
```

__add__, __sub__, __mul__, etc.

Estos métodos permiten definir el comportamiento de operadores aritméticos para los objetos de tu clase.

```python
class Vector:
        def __init__(self, x, y):
                self.x = x
                self.y = y

        def __add__(self, otro):
                return Vector(self.x + otro.x, self.y +
                    otro.y)

        def __sub__(self, otro):
                return Vector(self.x - otro.x, self.y -
                    otro.y)

        def __mul__(self, escalar):
                return Vector(self.x * escalar, self.y *
                    escalar)

        def __repr__(self):
                return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(5, 7)
v3 = v1 + v2

print(v3)  # Output: Vector(7, 10)

v4 = v2 - v1

print(v4)  # Output: Vector(3, 4)

print(v4 * 3) # Output: Vector(9,12)
```

2 Implementación de la clase Matriz

Para una implementación básica de la clase Matriz se han utilizado algunos métodos mágicos como:

- **__init__**: se utiliza como constructor de la clase recibiendo como argumentos filas ,columnas y los valores

```
class Matrix:
        def __init__(self, filas, columnas, valores=None):
                self.filas = filas
                self.columnas = columnas
                if valores:
                        if len(valores) != filas or any(len(
                            fila) != columnas for fila in
                            valores):
                                raise ValueError("Dimensiones
                                    de los valores no
                                    coinciden con las
                                    dimensiones de la matriz."
                                    )
                        self.valores = valores
                else:
                        self.valores = [[0] * columnas for _
                            in range(filas)]
```

- **__add__**: se utiliza como operador '+' de dos matrices , recibe como argumento otra matriz

```
        def __add__(self, otra):
                if self.filas != otra.filas or self.columnas
                    != otra.columnas:
                        raise ValueError("Las matrices deben
                            tener las mismas dimensiones para
                            sumar.")
                resultado = Matrix(self.filas, self.columnas)
                for i in range(self.filas):
                        for j in range(self.columnas):
                                resultado[i, j] = self[i, j] +
                                    otra[i, j]
                return resultado
```

- **__len__**: devuelve filas*columnas

```
        def __len__(self):
                return self.filas*self.columnas
```

- **__mul__**: se utiliza como operador '*' de dos matrices , recibe como argumento otra matriz

```
        def __mul__(self, otra):
                if self.columnas != otra.filas:
                        raise ValueError(
                        "El numero de columnas de la primera
                            matriz debe ser igual al numero de
                             filas de la segunda matriz.")
                resultado = Matrix(self.filas, otra.columnas)
                for i in range(self.filas):
                        for j in range(otra.columnas):
                                suma = 0
                                for k in range(self.columnas):
                                        suma += self[i, k] *
                                            otra[k, j]
                                resultado[i, j] = suma
                return resultado
```

3 Indización de la clase Matriz

En la clase matriz se utilizaron dos tipos de indización , uno donde se pueden hacer construcciones como a = matriz[0, 6] o matriz[1, 2] = 9 y otro donde se puede acceder por medio de campos de la forma a = matriz._0_6 o matriz._1_2 = 9

3.1 Indización clásica

La indización clásica es mediante corchetes para acceder a los valores de una posición específica de la matriz, para lograr esto hemos utilizado los métodos mágicos __getitem__ y __setitem__

```
        def __getitem__(self, indices):
                # obtener el valor
                fila, columna = indices
                return self.valores[fila][columna]
```

```
        def __setitem__(self, indices, valor):
                # asignar el valor
                fila, columna = indices
                self.valores[fila][columna] = valor
```

3.2 Indización mediante campos

Para la indización mediante campos mediante campos se utlizaron los metodos __getattr__ y __setattr__

- **__getattr__**

```
def __getattr__(self, nombre):
        if nombre.startswith('_'):
                partes = nombre[1:].split('_')
                if len(partes) == 2 and partes[0].isdigit()
                    and partes[1].isdigit():
                        fila, columna = int(partes[0]), int(
                            partes[1])
                        if 0 <= fila < self.filas and 0 <=
                            columna < self.columnas:
                                return self.valores[fila][
                                    columna]
```

- **__setattr__**

```
def __setattr__(self, nombre, valor):
        if nombre.startswith('_'):
                partes = nombre[1:].split('_')
                if len(partes) == 2 and partes[0].isdigit()
                    and partes[1].isdigit():
                        fila, columna = int(partes[0]), int(
                            partes[1])
                        if 0 <= fila < self.filas and 0 <=
                            columna < self.columnas:
                                self.valores[fila][columna] =
                                    valor
                                return
        super().__setattr__(nombre, valor)
```

4 Iterabilidad de los objetos matrices

Para lograr la iterabilidad de la clase Matriz se ha utilizado el metodo magico __iter__ lo que permite que una instancia de la clase sea iterable devolviendo los elementos de la matriz uno a uno de la forma que se pedia: matriz_1_1, matriz_1_2, ..., matriz_1_m, matriz_2_1, ..., matriz_n_m

```
def __iter__(self):
        for fila in range(self.filas):
                for columna in range(self.columnas):
                        yield self.valores[fila][columna]
```

5 Método as_type() para el tipo matriz

El metodo as_type permite convertir los elementos de la matriz al tipo indicado, si se tiene una matriz A podemos convertir todos sus elemntos a string de la forma $A.as\_str()$ Para lograr este comportamiento se ha manipulado el metodo magico $\_\_getattr\_\_$ en el cual al igual que en la indizacion, con una pequena manipulacion del texto podemos lograr el comportamiento deseado

```
def __getattr__(self, nombre):
        if nombre.startswith('_'):
                .....
        if nombre.startswith('as'):
                try:
                        partes = nombre.split('_')
                        t = eval(partes[1])
                        new_matrix = Matrix(self.filas, self.
                            columnas,
                                [[t(self.valores[x][y]) for y in
                                    range(self.columnas)] for x in
                                    range(self.filas)])

                        return new_matrix
                except:
                        pass
                raise AttributeError(f"'Matriz' object has no
                    attribute '{nombre}'")
```

6 Uso del *eval* y *super*

6.1 Eval

En esta clase Matriz se utiliza el *builtin eval* el cual permite evaluar una cadena como una expresion de python, en este caso se utiliza en $matrix.as\_type()$ para guardar el tipo al que se quiere convertir los elementos de la matriz. Esto es muy util al permitir capturar el tipo sin necesidad de realizar procedimientos adicionales, pero a la vez es bastante peligroso ya que un usuario malintencionado podria utilizar esta apertura para cambiar el comportamiento de los programas.

6.2 Super

En Python, super() es una función incorporada que se utiliza para acceder y llamar a métodos y atributos definidos en la clase base (superclase) de una clase. Proporciona una forma conveniente de invocar el comportamiento de la clase base sin tener que mencionar explícitamente el nombre de la clase base. En este caso se utiliza en el metodo magico __setattr__ ya que hemos cambiado su comportamiento en la clase para lograr acceder a los elementos de la matriz, pero como se vio anteriormente si no es el nombre esperado se desea que se siga el procedimiento habitual para los atributos de una clase.