

Lenguajes de Programación

Python Mágico



LEANDRO RODRÍQUEZ LLOSA
LAURA V. RIERA PÉREZ
MARCOS M. TIRADOR DEL RIEGO

Tercer año. Ciencias de la Computación.
Facultad de Matemática y Computación, Universidad de La Habana, Cuba
Noviembre 2022

Resumen

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Palabras clave — Palabra 1 · Palabra 2 · Palabra 3

I. MÉTODOS MÁGICOS

Los métodos mágicos son un conjunto de métodos especiales en Python, cuyo nombre comienza y termina con dos guiones bajos. Lo que hace especiales o “mágicos” a estos métodos es que no tienen que ser invocados directamente, sino que la invocación ocurre internamente desde la clase, bajo una determinada acción. Estos métodos también son llamados dunder, que es una abreviatura de “Double Underscore”, que significa doble guión bajo en inglés.

En este seminario se verá el uso de alguno de los métodos mágicos más importantes. Para ello se implementará una clase *Matrix* para representar matrices, con algunas funcionalidades básicas.

Se comenzará hablando del método `__init__`. Este método es invocado cuando se crea una instancia de una clase. Este actúa como un constructor cuya función es inicializar las propiedades

de la clase. Es por esto que el programador, para lograr el comportamiento deseado en la instanciación de una clase definida por él, debe redefinir el cuerpo de este método en la clase.

Sin embargo, es bueno saber que cuando uno instancia una clase, `__init__` no es el primer método mágico que es invocado. Antes es llamado el método `__new__`. Este método es el responsable de crear la instancia de la clase y luego se la pasa al método `__init__` junto con los argumentos. Explicar la utilidad de esta clase se sale del propósito de este seminario.

Se puede observar en la implementación siguiente del constructor de la clase `Matrix`.

```
def __init__(self, rows: int, cols: int, init_value=None):
    self.amount_rows: int = rows
    self.amount_cols: int = cols
    self.matrix: List[List[T]] = [[init_value
                                   for _ in range(0, cols)]
                                   for _ in range(0, rows)]
```

Figura 1. Constructor de la clase `Matrix`.

I. Operadores

Una de las ventajas de usar los métodos mágicos en Python es que permiten realizar operaciones con los objetos definidos como si fueran built-in. Por ejemplo si se quisiera saber si dos objetos son iguales habría que crear en la clase algún tipo de método `equals` que permita definir el comportamiento de la comparación entre dos instancias de la clase. Pero Python simplifica este comportamiento al permitirte redefinir el método mágico `__eq__()`. De esta forma podemos comparar dos objetos de un mismo tipo simplemente haciendo uso del operador `==`. Si no redefinimos este método en nuestra clase el comportamiento por defecto es que da un resultado positivo si y solo si las instancias de la clase que se están comparando son exactamente la misma.

Se presenta el código empleado para definir en la clase `Matrix` la posibilidad de comparar dos matrices solamente usando `==`.

```
def __eq__(self, other: 'Matrix[int, int]') -> bool:
    if self.amount_cols != other.amount_cols or self.amount_rows != other.amount_rows:
        return False

    for i in range(0, self.amount_rows):
        for j in range(0, self.amount_cols):
            if self[i, j] != other[i, j]:
                return False

    return True
```

Figura 2. Operador igual.

Nótese como al hacer uso de los métodos dunder el programador se ahorra tener que definir métodos propios para imitar el comportamiento de operaciones básicas. Además el llamado a estos métodos que se habrían de crear, sería engorroso y menos natural. Casi todos los operadores significativos pueden ser redefinidos a través de su método mágico correspondiente. Se mostrarán a continuación dos operadores aritméticos básicos que fueron redefinidos en la clase `Matrix`. Estos son la suma, a partir de `__add__` y la multiplicación a partir de `__mul__`.

```
def __add__(self, other: 'Matrix[T]') -> 'Matrix[T]':
    """
    Sumador de matrices.
    """

    result = Matrix(rows=self.amount_rows,
                    cols=self.amount_cols, init_value=None)

    for i in range(0, self.amount_rows):
        for j in range(0, self.amount_cols):
            result[i, j] = self[i, j] + other[i, j]

    return result
```

Figura 3. Operador suma.

```
def __mul__(self, other: 'Matrix[T]') -> 'Matrix[T]':
    """
    Multiplicador de matrices.
    """

    if self.amount_cols != other.amount_rows:
        raise Exception('Invalid operation.')

    result = Matrix(rows=self.amount_rows,
                    cols=other.amount_cols, init_value=None)

    for i in range(0, self.amount_rows):
        for j in range(0, other.amount_cols):
            for h in range(0, self.amount_cols):
                if h == 0:
                    result[i, j] = self[i, h] * other[h, j]
                else:
                    result[i, j] += self[i, h] * other[h, j]

    return result
```

Figura 4. Operador multiplicación.

II. Indexación

Los métodos mágicos en Python también permiten añadir la funcionalidad de indexar mediante corchetes para acceder al valor de un elemento, tanto para leer el mismo al redefinir `__getitem__` o modificarlo al redefinir `__setitem__`. Estos métodos se invocan automáticamente cuando se usa el operador indexador `[]` con la variable de referencia de la clase. Para secuencias, los índices aceptados deben ser números enteros y slices. La interpretación especial de los índices negativos depende de la redefinición método. Si el índice es de un tipo inapropiado, se puede generar `TypeError`, y si tiene un valor fuera del conjunto de índices de la secuencia, debe generarse `IndexError`. Para los tipos de asignación, si falta el índice, se debe generar `KeyError`.

A continuación se muestra la implementación dada a ambos, `__getitem__` y `__setitem__`, al redefinirlos en la clase `Matrix`:

```
def __getitem__(self, key: Tuple[int, int]):
    """
    Indizador con una sintaxis más cómoda.
    Ejemplo: a = matrix[i,j]
    """

    if not isinstance(key, tuple):
        raise Exception('Format incorrect.')

    if len(key) != 2:
        raise Exception('Number of parameters exceded.')
    i, j = key

    if i >= 0 and i < self.amount_rows and \
        j >= 0 and j < self.amount_cols:
        return self.matrix[i][j]
    else:
        raise Exception('Index out of matrix.')
```

Figura 5. Indexación para obtener el valor de un elemento de la matriz.

```
def __setitem__(self, key: Tuple[int, int], value: T):
    """
    Permite setear el valor de la matriz indexada,
    de manera más cómoda.
    Ejemplo: `matrix[i,j] = 4`
    """

    if not isinstance(key, tuple):
        raise Exception('Format incorrect.')

    if len(key) != 2:
        raise Exception('Number of parameters exceded.')
    i, j = key

    if i >= 0 and i < self.amount_rows and \
        j >= 0 and j < self.amount_cols:
        self.matrix[i][j] = value
    else:
        raise Exception('Index out of matrix.')
```

Figura 6. Indexación para asignar un nuevo valor a un elemento de la matriz.

III. Iteración

`__next__`

Devuelve el siguiente elemento del iterador. Si no hay más elementos, genera la excepción `StopIteration`.

```
def __next__(self):
    for row in self.matrix:
        for i in row:
            yield i
```

Figura 7. Operador multiplicación.

`__iter__`

Devuelve un objeto iterador.

El primer argumento se interpreta de manera muy diferente dependiendo de la presencia del segundo argumento. Sin un segundo argumento, el objeto debe ser un objeto de colección que admita el protocolo iterable (el método `__iter__()`), o debe admitir el protocolo de secuencia (el método `__getitem__` con argumentos enteros que comienzan en 0). Si no es compatible con ninguno de esos protocolos, se genera `TypeError`.

Si se proporciona el segundo argumento, centinela, entonces el objeto debe ser un objeto invocable. El iterador creado en este caso llamará al objeto sin argumentos para cada llamada a su método `next()`; si el valor devuelto es igual a centinela, se generará `StopIteration`; de lo contrario, se devolverá el valor.

```
def __iter__(self):
    return self.__next__()
```

Figura 8. Operador multiplicación.

iv. Atributos

A través del uso de los métodos mágicos Python permite controlar el acceso a los atributos de una clase. Esto es, permite definir un comportamiento determinado cuando se intente acceder a un atributo de la clase, incluso cuando este atributo no existe. Esto nos da la posibilidad, por ejemplo, de prohibir el acceso a ciertos atributos de la clase. Incluso da la posibilidad de modificar los atributos de la clase en tiempo de compilación. Se verán a continuación 3 métodos que permiten este comportamiento.

- `__getattr__` Este método es lo primero que se invoca cuando se intenta acceder a un atributo de una clase, sin importar si este existe o no. Permite definir reglas de comportamiento en respuesta al acceso al atributo. El comportamiento del método por defecto es buscar si el atributo existe, retornando su valor en caso afirmativo y lanzando `AttributeError` en caso contrario.
- `__getattribute__` Este método es invocado siempre que se intente acceder a un atributo que no existe en un objeto. Además, si el método anterior lanza `AttributeError` después de su ejecución, esta excepción es ignorada y este método es invocado.
- `__setattr__` Similar al primero de estos tres, este método siempre es invocado cuando se intenta cambiar el valor de un atributo. De igual forma su uso suele ser aplicar algunas reglas antes de que se ejecute la modificación. Adicionalmente puede usarse para definir un comportamiento si el atributo en cuestión no existe.

Hay que tener cierto cuidado al usar estos tres métodos. Sucede que si en el cuerpo del método se hace uso de algún otro atributo de la clase, entonces el método será invocado nuevamente provocando una recursión infinita.

A continuación se presenta la forma en que se redefinieron los métodos `__getattr__` y `__setattr__` en la clase `Matrix`.

revisar la validez de la oracion anterior pues no estoy seguro al 100

```

def __getattr__(self, __name: str):
    matched = re.match(r"_(\d+)_(\d+)", __name)
    if matched:
        i, j = matched.groups()
        i = int(i)
        j = int(j)
        return self[i, j]

    matched = re.match(r"as_([a-z]+)", __name)
    if matched:
        type = matched.groups()[0]
        result = Matrix(self.amount_rows, self.amount_cols)

        for i in range(0, self.amount_rows):
            for j in range(0, self.amount_cols):
                result[i, j] = eval(f'{type}(self.matrix[i][j])')

    return lambda: result

```

Figura 9. Constructor de la clase Matrix.

```

def __setattr__(self, __name: str, __value: Any):
    matched = re.match(r"_(\d+)_(\d+)", __name)
    if matched:
        i, j = matched.groups()
        i = int(i)
        j = int(j)
        self[i, j] = __value
    return super().__setattr__(__name, __value)

```

Figura 10. Constructor de la clase Matrix.

Una de las ideas con el uso del `__setattr__` y `__getattr__` es poder indexar en la matriz m usando la forma `m.i_j` donde i y j son los índices a los que se quiere acceder. Como `_i_i` no es un atributo de Matrix, al intentar acceder al valor del mismo el programa invoca al método `__getattr__` después de ejecutar `__getattribute__` sin éxito. Entonces al método en cuestión es pasado como parámetro el nombre del atributo al que se quiere acceder. Haciendo uso de la biblioteca `re` para comprobar patrones en expresiones regulares, podemos comparar el nombre del método con el patrón `_i_j`, y si coinciden solamente debemos obtener los enteros i y j del nombre, y devolver `self[i][j]`. Análogamente se hace con `__setattr__` pero esta vez para modificar el valor en esa posición. La diferencia es que como se explicaba antes, este método se llama independientemente de si el atributo aparece o no, pero en este caso de igual modo este atributo no existe.

Nótese que la idea usada anteriormente consistió en poder invocar un comportamiento determinado, como si fuera un atributo de la clase. Una idea similar usamos para lograr castear el tipo de los elementos en la matriz representada por la clase Matrix, a otro tipo diferente. Se quiere que este comportamiento se pueda lograr de la forma `m.as_type()` (por ejemplo se pudiera llamar a `m.as_float()` para obtener una nueva matriz con todos sus valores de tipo float). Pero se puede adicionar otro comportamiento al método `__getattr__`, ya que Matrix no contiene un atributo `as_type` para ninguna cadena "type". La idea es de modo similar, haciendo uso de `re`, reconocer el patrón de `as_type` en el nombre del atributo. Si este coincide se crea una matriz nueva del tipo en cuestión y se castea cada elemento de la matriz anterior al tipo nuevo. Para esto último se hace uso del método `eval` que evalúa el texto de un string como si fuera una sentencia

de código python. Finalmente se devuelve como resultado de la invocación de este método, una expresión lambda cuyo cuerpo consiste solamente de retornar la nueva matriz. Esto se hace ya que la sentencia `m.as_type()` espera que el atributo `as_type` devuelva una función, que se invocará inmediatamente sin argumentos.

v. Otros métodos utilizados

`__repr__`

```
def __repr__(self) -> str:
    result = ''
    j = 0
    for i in self:
        j += 1
        result += f'{i} '
        if j % self.amount_cols == 0 and j != \
            self.amount_cols * self.amount_rows:
            result += '\n'
    return result
```

Figura 11. Constructor de la clase Matrix.

`__len__`

```
def __len__(self) -> int:
    return self.amount_cols * self.amount_rows
```

Figura 12. Constructor de la clase Matrix.

`__call__`

```
def __call__(self, *args: Any, **kwargs: Any) -> Any:
    print(f"I'm callable.. args:{args} kwargs:{kwargs}")
```

Figura 13. Constructor de la clase Matrix.

REFERENCIAS

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.