Python → Object-oriented programming → Math magic methods

Theory: Math magic methods

30 minutes 0 / 5 problems solved

Skip this topic Start practicing

571 users solved this topic. Latest completion was 2 days ago.

If you want to improve and upgrade your classes, chances are that what you need can be done with magic methods. Dunders in Python provide a wide range of functionalities, from type conversion to attribute managing. In this topic, we'll focus on "mathematical" magic methods: the ones that deal with arithmetics and comparisons.

Current topic:

Math magic methods

Topic depends on:

✓ Magic methods 3★

Table of contents:

§1. Arithmetic operations

There is a group of magic methods representing common arithmetic operations:

- `__add__()` : addition (+)
- `__sub__()` : subtraction (-)
- `__mul__()` : multiplication (*)
- `__truediv__()` : division (/)
- `__pow__()` : exponents (**)

This list is not exhaustive, there are many other methods, but these are the most common ones.

However, the question is when would we need to use them? Let's consider an example.

```python
class ComplexNumber:
    def __init__(self, real_part, im_part):
        self.real_part = real_part
        self.im_part = im_part
```

This is a class that represents **complex numbers**. A complex number is a number that can be expressed as $a + bi$ where a and b are real numbers and i is the imaginary number: the solution of the equation $x^2 = -1$. So a is the real part of the complex number and b is the imaginary one. Just like ordinary numbers, we can add two complex numbers, multiply them, do subtraction and division and many other standard operations. Let's create two complex numbers and try to add and multiply them:

```python
z1 = ComplexNumber(1, 1)
z2 = ComplexNumber(-1, 2)

z3 = z1 + z2

# TypeError: unsupported operand type(s) for +: 'ComplexNumber' and 'ComplexNumber'
z4 = z1 * z2

# TypeError: unsupported operand type(s) for *: 'ComplexNumber' and 'ComplexNumber'
```

As you can see, we cannot simply add two complex numbers using the standard operators in Python. We could, of course, define custom methods like `add` or `multiply` and call them when we need to do math with complex numbers, but it wouldn't be too convenient. A more elegant solution to this problem is to define magic methods `__add__` and `__mul__` in our `ComplexNumber` class and then simply use standard operators with our complex numbers. Here's how it would look:

```python
class ComplexNumber:
    def __init__(self, real_part, im_part):
        self.real_part = real_part
        self.im_part = im_part

    def __add__(self, other):
        """Addition of complex numbers."""
        real = self.real_part + other.real_part
        imaginary = self.im_part + other.im_part
        return ComplexNumber(real, imaginary)

    def __mul__(self, other):
        """Multiplication of complex numbers."""
        real = (self.real_part * other.real_part -
                self.im_part * other.im_part)
        imaginary = (self.real_part * other.im_part +
                     other.real_part * self.im_part)
        return ComplexNumber(real, imaginary)
```

First, let's go over the details of the methods. Since both addition and multiplication operators are binary, the methods take two arguments, `self` and `other`. `other` is a name typically used to denote another object of the same class. These parameters represent the operands of these operations and in this case, `self` is the left operand and `other` is the right one. If the operator was unary, the method would only take `self` as a parameter.

Both methods also calculate the real and the imaginary part of a new complex number and then return a new `ComplexNumber` object as their results.

If we try to calculate the sum and the product of our complex numbers now, we won't get any errors:

```python
z3 = z1 + z2
print(z3.real_part, z3.im_part)  # 0 3

z4 = z1 * z2
print(z4.real_part, z4.im_part)  # -3 1
```

If we wanted to define other operations, for instance, subtraction or division, we would do it similarly. The main principle with these methods is to understand what particular operation means for your object and then define the corresponding method.

§2. Augmented assignment

Python also has a number of methods for augmented assignment: a combination of standard arithmetic operations with the assignment. Their names are pretty self-explanatory so you can probably guess what operations they stand for:

- `__iadd__()`;
- `__isub__()`;
- `__imul__()`;
- `__itruediv__()`;
- `__ipow__()`;
- and others...

For our complex numbers, let's define the method for += operator:

```python
class ComplexNumber:
    def __init__(self, real_part, im_part):
        self.real_part = real_part
        self.im_part = im_part

    # other magic methods we've already defined

    def __iadd__(self, other):
        """Addition with assignment (+=) for complex numbers."""

        self.real_part += other.real_part

        self.im_part += other.im_part

        return self
```

This is how it works:

```python
z1 = ComplexNumber(8, -3)
z2 = ComplexNumber(-6, 2)

z1 += z2
print(z1.real_part, z1.im_part)  # 2 -1
```

All magic methods for augmented assignment should return an instance of the class (`self`) so that the assignment works correctly.

§3. Comparison operators

Another thing we might want to do with our objects is to compare them. Here are the magic methods that define the behavior of the comparison operators:

- `__eq__()` : equality (==)
- `__ne__()` : inequality (!=)
- `__lt__()` : less than (<)
- `__gt__()` : greater than (>)
- `__le__()` : less or equal (<=)
- `__ge__()` : greater or equal (>=)

Let's again take our class of complex numbers. Let's create 3 complex numbers and try to compare them.

```python
z1 = ComplexNumber(5, -5)
z2 = ComplexNumber(5, -5)
z3 = ComplexNumber(4, 4)

print(z1 == z2)  # False
print(z1 == z3)  # False
```

Two complex numbers are considered equal if and only if their real parts and imaginary parts are respectively equal. But, as we can see above, this is not how it worked in our program. This is because all numbers are different objects: the code checks them for identity, not equality, and since they have different id numbers, they are not considered the same.

We need to define the `__eq__` method so that our comparisons run smoothly:

```python
class ComplexNumber:
    def __init__(self, real_part, im_part):
        self.real_part = real_part
        self.im_part = im_part

    def __eq__(self, other):
        """Compare two complex numbers for equality (==)."""
        return ((self.real_part == other.real_part) and
                (self.im_part == other.im_part))
```

Now if we try to compare our two numbers we will get the correct result:

```
print(z1 == z2)  # True
print(z1 == z3)  # False
```

Other comparison operators can be defined in a similar way.

§4. Summary

In this topic, we've covered dunders that are useful when you want to introduce a little bit of math in your classes. A big advantage of magic methods is that you don't have to define them if you don't need it. But if you do, they're there to make your life easier!

Report a typo

62 users liked this theory. **2** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion