Intuitive Object Syntax with Magic Methods

Why it’s important…

Many libraries hook into Python’s data model to provide a natural, intuitive interface to extremely powerful features.

If your library is easy to use, more developers will use it. That’s what this gives you.

Example: Pandas

Reference: "Data Model" chapter in the Python Language Reference: https://docs.python.org/3/reference/datamodel.html

What’s our angle?

Imagine we need an `Angle` class. Requirements:

- Range of [0, 360) degrees (wrapping around)
- Able to add two angles, multiply, etc.
- Comparisons like ==, <=, >, etc.
- Use the natural arithmetic and other operators Python provides

What’s the best way to implement this?

Unintuitive Syntax

```
class Angle:
    def __init__(self, value):
        self.value = value % 360
    def add(self, other_angle):
        return Angle(self.value + other_angle.value)
```

```
>>> a = Angle(45)
>>> b = Angle(90)
>>> c = a.add(b)
>>> print(c.value)
135
```

This works. But we'd rather use the built-in operators:

```
c = a + b
print(c.value)
```

How can we create this?

Magic Methods

Python provides **magic methods**. These are methods your classes can define which hook into Python's built-in operators.

For addition, you simply define an `__add__` method:

```python
class Angle:
    def __init__(self, value):
        self.value = value % 360
    def __add__(self, other_angle):
        return Angle(self.value + other_angle.value)
```

```
>>> a = Angle(45)
>>> b = Angle(90)
>>> c = a + b
>>> print(c.value)
135
```

That's all you have to do.

Arithmetic Hooks

You can define a full range of binary operations.

`__add__`	a + b
`__sub__`	a - b
`__mul__`	a * b
`__truediv__`	a / b (floating-point division)
`__mod__`	a % b
`__pow__`	a ** b

Essentially, Python translates `a + b` to `a.__add__(b)`; `a % b` to `a.__mod__(b)`; etc.

Bitwise Operator Hooks

You can also hook into the bit-operation operators:

`__lshift__`	a << b
`__rshift__`	a >> b
`__and__`	a & b
`__xor__`	a ^ b
`__or__`	a | b

So `a & b` translates to `a.__and__(b)`, etc.

Sadly, there is no hook for the binary logical `and` and `or` operators. Only bitwise `&` and `|`.

Rich Comparisons

More commonly, your classes will need to be comparable. Magic methods are available for ==, >, <=, etc.

Here’s what we want to be able to do:

```
>>> x = Angle(30)
>>> y = Angle(60)
>>> z = Angle(30)
>>>
>>> x == y
False
>>> x == z
True
>>> x > y
False
>>> z <= y
True
>>> z <= x
True
```

Rich Comparision Methods

These are provided by the following hook methods:

`__eq__`	x == y
`__ne__`	x != y
`__lt__`	x < y
`__le__`	x <= y
`__gt__`	x > y
`__ge__`	x >= y

Rich Comparision Methods

For example, for `x == y`:

```
class Angle:
    def __init__(self, value):
        self.value = value % 360
    # ...
    def __eq__(self, other):
        return self.value == other.value
```

Rebellious Magic Methods

Fascinating fact:

Methods like `__add__` don't actually **have** to do addition.

Methods like `__gt__` aren't required to return `True` or `False`.

This creates some amazing possibilities.

http://powerfulpython.com/blog/rebellious-magic-methods-python-syntax/

Pandas

Pandas is an excellent data-processing library.

```
import pandas
df = pandas.DataFrame({
            'A': [-137, 22, -3, 4, 5],
            'B': [10, 11, 121, 13, 14],
            'C': [3, 6, 91, 12, 15],
      })
```

`df` is what Pandas calls a **dataframe**:

```
>>> print(df)
     A    B   C
0 -137   10   3
1   22   11   6
2   -3  121  91
3    4   13  12
4    5   14  15
```

Filtering

You can filter out rows in dataframe, to get another, smaller dataframe.

```
>>> positive_a = df[df.A > 0]
>>> print(positive_a)
    A   B   C
1  22  11   6
3   4  13  12
4   5  14  15
```

But wait a second...

Look again at that code:

```
positive_a = df[df.A > 0]
```

That expression `df.A > 0` ought to be `True` xor `False`, right? So there would be no way to filter rows dynamically at runtime.

How does this even work?

Hey, that's cheating!

Turns out it's not boolean at all:

```
>>> comparison = (df.A > 0)
>>> type(comparison)
<class 'pandas.core.series.Series'>
>>> print(comparison)
0    False
1     True
2    False
3     True
4     True
Name: A, dtype: bool
```

Yes, you can do that, thanks to Python's dynamic type system!

Comparison object

`df.A > 0` is translated to `df.A.__gt__(0)`

Rather than re-inventing Pandas, let's create a similar, but simplified library. If `df.A` represents a data column, let's have a `Column` type whose `__gt__` method returns a `Comparison` object.

```
import operator
class Column:
    def __init__(self, name):
        self.name = name
    def __gt__(self, value):
         return Comparison(self.name, value, operator.gt)
```

More details:

This is a taste of how you might implement a Pandas-like interface.

To evaluate expressions like `df[df.C + 2 < df.B]`, you need to do more work - but it can all be done via these magic methods.

Full details:

https://powerfulpython.com/blog/rebellious-magic-methods-python-syntax/

Lab: Magic Money

Lab file: `oop/magicmoney.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up.

(This lab introduces two other simple, but useful magic methods, `__str__` and `__repr__`. Read their explanations in the lab file.)