MAGIC METHODS AND OPERATOR OVERLOADING

INTRODUCTION

The so-called magic methods have nothing to do with wizardry. You have already seen them in previous chapters of our tutorial. They are the methods with this clumsy syntax, i.e. the double underscores at the beginning and the end. They are also hard to talk about. How do you pronounce or say a method name like __init__? "Underscore underscore init underscore underscore" sounds horrible and is nearly a tongue twister. "Double underscore init double underscore" is a lot better, but the ideal way is "dunder init dunder"[1] That's why magic methods methods are sometimes called dunder methods!

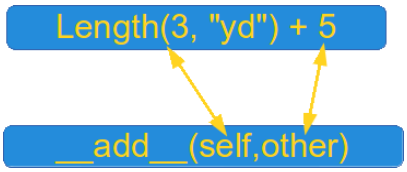
So what's magic about the __init__ method? The answer is, you don't have to invoke it directly. The invocation is realized behind the scenes. When you create an instance x of a class A with the statement "x = A()", Python will do the necessary calls to __new__ and __init__.

We have encountered the concept of operator overloading many times in the course of this tutorial. We had used the plus sign to add numerical values, to concatenate strings or to combine lists:

```
>>> 4 + 5
9
>>> 3.8 + 9
12.8
>>> "Peter" + " " + "Pan"
'Peter Pan'
>>> [3,6,8] + [7,11,13]
[3, 6, 8, 7, 11, 13]
>>>
```

It's even possible to overload the "+" operator as well as all the other operators for the purposes of your own class. To do this, you need to understand the underlying mechanism. There is a special (or a "magic") method for every operator sign. The magic method for the "+" sign is the __add__ method. For "-" it is "__sub__" and so on. We have a complete listing of all the magic methods a little bit further down.

The mechanism works like this: If we have an expression "x + y" and x is an instance of class K, then Python will check the class definition of K. If K has a method __add__ it will be called with x.__add__(y), otherwise we will get an error message.




```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'K' and 'K'
```

OVERVIEW OF MAGIC METHODS

BINARY OPERATORS

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

EXTENDED ASSIGNMENTS

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)

%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

UNARY OPERATORS

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

COMPARISON OPERATORS

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

EXAMPLE CLASS: LENGTH

We will demonstrate in the following Length class, how you can overload the "+" operator for your own class. To do this, we have to overload the __add__ method. Our class contains the __str__ and

__repr__ methods as well. The instances of the class Length contain length or distance information. The attributes of an instance are self.value and self.unit.

This class allows us to calculate expressions with mixed units like this one:

2.56 m + 3 yd + 7.8 in + 7.03 cm

The class can be used like this:

```
>>> from unit_conversions import Length
>>> L = Length
>>> print(L(2.56,"m") + L(3,"yd") + L(7.8,"in") + L(7.03,"cm"))
5.57162
>>>
```

The listing of the class:

```
class Length:

    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
                "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
                "mi" : 1609.344 }

    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit +
"')"

if __name__ == "__main__":
    x = Length(4)
    print(x)
    y = eval(repr(x))
```

```
z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)
```

If we start this program, we get the following output:

```
4
Length(5.593613298337708, 'yd')
5.1148
```

We use the method__iadd__ to implement the extended assignment:

```
def __iadd__(self, other):
    l = self.Converse2Metres() + other.Converse2Metres()
    self.value = l / Length.__metric[self.unit]
    return self
```

Now we are capable to write the following assignments:

```
x += Length(1)
x += Length(4, "yd")
```

We have added 1 metre in the example above by writing "x += Length(1))". Most certainly, you will agree with us that it would be more convenient to simply write "x += 1" instead. We also want to treat expressions like "Length(5,"yd") + 4.8" similarly. So, if somebody uses a type int or float, our class takes it automatically for "metre" and converts it into a Length object. It's easy to adapt our __add__ and "__iadd__" method for this task. All we have to do is to check the type of the parameter "other":

```
def __add__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit )

def __iadd__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
```

```
        else:
            l = self.Converse2Metres() + other.Converse2Metres()
        self.value = l / Length.__metric[self.unit]
        return self
```

It's a safe bet that if somebody works for a while with adding integers and floats from the right sight that he or she wants to the same from the left side! So let's try it out:

```
>>> from unit_conversions import Length
>>> x = Length(3, "yd") + 5
>>> x = 5 + Length(3, "yd")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Length'
>>>
```

Of course, the left side has to be of type "Length", because otherwise Python tries to apply the __add__ method from int, which can't cope with Length objects as second arguments!

Python provides a solution for this problem as well. It's the __radd__ method. It works like this: Python tries to evaluate the expression "5 + Length(3, 'yd')". First it calls int.__add__(5,Length(3, 'yd')), which will raise an exception. After this it will try to invoke Length.__radd__(Length(3, "yd"), 5). It's easy to recognize that the implementation of __radd__ is analogue to __add__:

```
    def __radd__(self, other):
        if type(other) == int or type(other) == float:
            l = self.Converse2Metres() +
otherLength.__radd__(Length(3, "yd"), 5)
        else:
            l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )
```

It's advisable to make use of the __add__ method in the __radd__ method:

```
    def __radd__(self, other):
        return Length.__add__(self,other)
```

The following diagram illustrates the relationship between __add__ and __radd__:

STANDARD CLASSES AS BASE CLASSES

It's possible to use standard classes - like int, float, dict or lists - as base classes as well.

We extend the list class by adding a push method:

```
class Plist(list):

    def __init__(self, l):
        list.__init__(self, l)

    def push(self, item):
        self.append(item)


if __name__ == "__main__":
    x = Plist([3,4])
    x.push(47)
    print(x)
```

This means that all the previously introduced binary and extended assignment operators exist in the "reversed" version as well:

```
__radd__
__rsub__
__rmul__
...
and so on
```

EXERCISES

1. Write a class with the name Ccy, similar to the previously defined Length class.

   Ccy should contain values in various currencies, e.g. "EUR", "GBP" or "USD". An instance should contain the amount and the currency unit.

   The class, you are going to design as an excercise, might be best described with the following example session:

```
>>> from currencies import
Ccy
>>> v1 = Ccy(23.43, "EUR")
>>> v2 = Ccy(19.97, "USD")
>>> print(v1 + v2)
32.89 EUR
>>> print(v2 + v1)
31.07 USD
>>> print(v1 + 3) # an int or a float is considered to be a
EUR value
27.43 EUR
>>> print(3 + v1)
27.43 EUR
>>>
```

SOLUTIONS TO OUR EXERCISES

1. First exercise:

```
"""

The class "Ccy" can be used to define money values in various
currencies. A Ccy instance has the string attributes 'unit'
(e.g. 'CHF', 'CAD' od 'EUR' and the 'value' as a float.
A currency object consists of a value and the corresponding
```

```
    unit.


    """


    class Ccy:

        currencies =  {'CHF': 1.0821202355817312,
                       'CAD': 1.488609845538393,
                       'GBP': 0.8916546282920325,
                       'JPY': 114.38826536281809,
                       'EUR': 1.0,
                       'USD': 1.11123458162018}

        def __init__(self, value, unit="EUR"):
            self.value = value
            self.unit = unit

        def __str__(self):
            return "{0:5.2f}".format(self.value) + " " + self.unit

        def changeTo(self, new_unit):
            """
            An Ccy object is transformed from the unit "self.unit"
    to "new_unit"
            """
            self.value = (self.value / Ccy.currencies[self.unit] *
    Ccy.currencies[new_unit])
            self.unit = new_unit

        def __add__(self, other):
            """
            Defines the '+' operator.
            If other is a CCy object the currency values
            are added and the result will be the unit of
            self. If other is an int or a float, other will
            be treated as a Euro value.
            """
            if type(other) == int or type(other) == float:
                x = (other * Ccy.currencies[self.unit])
            else:
                x = (other.value / Ccy.currencies[other.unit] *
    Ccy.currencies[self.unit])
            return Ccy(x + self.value, self.unit)
```

```
    def __iadd__(self, other):
        """
        Similar to __add__
        """
        if type(other) == int or type(other) == float:
            x = (other * Ccy.currencies[self.unit])
        else:
            x = (other.value / Ccy.currencies[other.unit] *
Ccy.currencies[self.unit])
        self.value += x
        return self

    def __radd__(self, other):
        res = self + other
        if self.unit != "EUR":
            res.changeTo("EUR")
        return res

    # __sub__, __isub__ and __rsub__ can be defined analogue

x = Ccy(10,"USD")
y = Ccy(11)
z = Ccy(12.34, "JPY")
z = 7.8 + x + y + 255 + z
print(z)

lst = [Ccy(10,"USD"), Ccy(11), Ccy(12.34, "JPY"), Ccy(12.34,
"CAD")]

z = sum(lst)

print(z)
```

The program returns:

```
282.91 EUR
28.40 EUR
```

Another interesting aspect of this currency converter class in Python can be shown, if we add multiplication. You will easily understand that it makes no sense to allow expressions like "12.4 € * 3.4 " (*orinpraefixnotation :" €12.4* 3.4"), but it makes perfect sense to evaluate "3 * 4.54 €". You can find the new currency converter class with the newly added methods for __mul__, __imul__ and __rmul__ in the following listing:

```
"""
```

```
The class "Ccy" can be used to define money values in various
currencies. A Ccy instance has the string attributes 'unit'
(e.g. 'CHF', 'CAD' od 'EUR' and the 'value' as a float.
A currency object consists of a value and the corresponding
unit.



"""



class Ccy:

    currencies =  {'CHF': 1.0821202355817312,
                   'CAD': 1.488609845538393,
                   'GBP': 0.8916546282920325,
                   'JPY': 114.38826536281809,
                   'EUR': 1.0,
                   'USD': 1.11123458162018}

    def __init__(self, value, unit="EUR"):
        self.value = value
        self.unit = unit

    def __str__(self):
        return "{0:5.2f}".format(self.value) + " " + self.unit

    def __repr__(self):
        return 'Ccy(' + str(self.value) + ', "' + self.unit +
'")'

    def changeTo(self, new_unit):
        """
        An Ccy object is transformed from the unit "self.unit"
to "new_unit"
        """
        self.value = (self.value / Ccy.currencies[self.unit] *
Ccy.currencies[new_unit])
        self.unit = new_unit

    def __add__(self, other):
        """
        Defines the '+' operator.
        If other is a CCy object the currency values
        are added and the result will be the unit of
        self. If other is an int or a float, other will
        be treated as a Euro value.
```

```
        """
        if type(other) == int or type(other) == float:
            x = (other * Ccy.currencies[self.unit])
        else:
            x = (other.value / Ccy.currencies[other.unit] *
Ccy.currencies[self.unit])
        return Ccy(x + self.value, self.unit)


    def __iadd__(self, other):
        """
        Similar to __add__
        """
        if type(other) == int or type(other) == float:
            x = (other * Ccy.currencies[self.unit])
        else:
            x = (other.value / Ccy.currencies[other.unit] *
Ccy.currencies[self.unit])
        self.value += x
        return self

    def __radd__(self, other):
        res = self + other
        if self.unit != "EUR":
            res.changeTo("EUR")
        return res

    # __sub__, __isub__ and __rsub__ can be defined analogue


    def __mul__(self, other):
        """
        Multiplication is only defined as a scalar
multiplication,
        i.e. a money value can be multiplied by an int or a
float.
        It is not possible to multiply to money values
        """
        if type(other)==int or type(other)==float:
            return Ccy(self.value * other, self.unit)
        else:
            raise TypeError("unsupported operand type(s) for
*: 'Ccy' and " + type(other).__name__)

    def __rmul__(self, other):
        return self.__mul__(other)

    def __imul__(self, other):
```

```
        if type(other)==int or type(other)==float:
            self.value *= other
            return self
        else:
            raise TypeError("unsupported operand type(s) for
*: 'Ccy' and " + type(other).__name__)
```

Assuming that you have saved the class under the name currency_converter, you can use it in the following way in the command shell:

```
>>> from currency_converter import Ccy
>>> x = Ccy(10.00, "EUR")
>>> y = Ccy(10.00, "GBP")
>>> x + y
Ccy(21.215104685942173, "EUR")
>>> print(x + y)
21.22 EUR
>>> print(2*x + y*0.9)
30.09 EUR
>>>
```

We can further improve our currency converter class by using a function get_currencies, which downloads the latest exchange rates from finance.yahoo.com. This function returns an exchange rates dictionary in our previously used format. The function is in a module called exchange_rates.py This is the code of the function exchange_rates.py:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

def get_currency_rates(base="USD"):
    """ The file at location url is read in and the exchange
rates are extracted """
    url =
"https://finance.yahoo.com/webservice/v1/symbols/allcurrencies
/quote"
    data = urlopen(url).read()
    data = data.decode('utf-8')
    soup = BeautifulSoup(data, 'html.parser')
    data = soup.get_text()

    flag = False
    currencies = {}
    for line in data.splitlines():
        if flag:
            value = float(line)
            flag = False
            currencies[currency] = value
```

```
            if line.startswith("USD/"):
                flag = True
                currency = line[4:7]

    currencies["USD"] =  1 # we must add it, because it's not
included in file
    if base != "USD":
        base_currency_rate = currencies[base]
        for currency in currencies:
            currencies[currency] /= base_currency_rate

    return currencies
```

We can import this function from our module. (You have to save it somewhare in your Python path or the directory where your program runs):

```
from exchange_rates import get_currency_rates

class Ccy:

    currencies =  get_currencies()

    #   continue with the code from the previous version
```

We save this version as currency_converter_web.

```
>>> from currency_converter_web import Ccy
>>> x = Ccy(1000, "JPY")
>>> y = Ccy(10, "CHF")
>>> z = Ccy(15, "CAD")
>>> print(2*x + 4.11*y + z)
7722.98 JPY
>>>
```

FOOTNOTES

[1] as suggested by Mark Jackson