Practical Python Programming

A course by @dabeaz

Contents | Previous (4.2 Inheritance) | Next (4.4 Exceptions)

4.3 Special Methods

Various parts of Python's behavior can be customized via special or so-called "magic" methods. This section introduces that idea. In addition dynamic attribute access and bound methods are discussed.

Introduction

Classes may define special methods. These have special meaning to the Python interpreter. They are always preceded and followed by `__` . For example `__init__` .

```
class Stock(object):
    def __init__(self):
        ...
    def __repr__(self):
        ...
```

There are dozens of special methods, but we will only look at a few specific examples.

Special methods for String Conversions

Objects have two string representations.

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> print(d)
2012-12-21
>>> d
datetime.date(2012, 12, 21)
>>>
```

The `str()` function is used to create a nice printable output:

```
>>> str(d)
'2012-12-21'
>>>
```

The `repr()` function is used to create a more detailed representation for programmers.

```
>>> repr(d)
'datetime.date(2012, 12, 21)'
>>>
```

Those functions, `str()` and `repr()`, use a pair of special methods in the class to produce the string to be displayed.

```
class Date(object):
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    # Used with `str()`
    def __str__(self):
        return f'{self.year}-{self.month}-{self.day}'

    # Used with `repr()`
    def __repr__(self):
        return f'Date({self.year},{self.month},{self.day})'
```

Note: The convention for `__repr__()` is to return a string that, when fed to `eval()`, will recreate the underlying object. If this is not possible, some kind of easily readable representation is used instead.

Special Methods for Mathematics

Mathematical operators involve calls to the following methods.

```
a + b       a.__add__(b)
a - b       a.__sub__(b)
a * b       a.__mul__(b)
a / b       a.__truediv__(b)
a // b      a.__floordiv__(b)
a % b       a.__mod__(b)
a << b      a.__lshift__(b)
a >> b      a.__rshift__(b)
```

```
a & b       a.__and__(b)
a | b       a.__or__(b)
a ^ b       a.__xor__(b)
a ** b      a.__pow__(b)
-a          a.__neg__()
~a          a.__invert__()
abs(a)      a.__abs__()
```

Special Methods for Item Access

These are the methods to implement containers.

```
len(x)      x.__len__()
x[a]        x.__getitem__(a)
x[a] = v    x.__setitem__(a,v)
del x[a]    x.__delitem__(a)
```

You can use them in your classes.

```
class Sequence:
    def __len__(self):
        ...
    def __getitem__(self,a):
        ...
    def __setitem__(self,a,v):
        ...
    def __delitem__(self,a):
        ...
```

Method Invocation

Invoking a method is a two-step process.

1. Lookup: The `.` operator
2. Method call: The `()` operator

```
>>> s = Stock('GOOG',100,490.10)
>>> c = s.cost  # Lookup
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()         # Method call
49010.0
>>>
```

Bound Methods

A method that has not yet been invoked by the function call operator `()` is known as a *bound method*. It operates on the instance where it originated.

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s
<Stock object at 0x590d0>
>>> c = s.cost
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()
49010.0
>>>
```

Bound methods are often a source of careless non-obvious errors. For example:

```
>>> s = Stock('GOOG', 100, 490.10)
>>> print('Cost : %0.2f' % s.cost)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: float argument required
>>>
```

Or devious behavior that's hard to debug.

```
f = open(filename, 'w')
...
f.close     # Oops, Didn't do anything at all. `f` still open.
```

In both of these cases, the error is cause by forgetting to include the trailing parentheses. For example, `s.cost()` or `f.close()` .

Attribute Access

There is an alternative way to access, manipulate and manage attributes.

```
getattr(obj, 'name')          # Same as obj.name
setattr(obj, 'name', value)   # Same as obj.name = value
delattr(obj, 'name')          # Same as del obj.name
hasattr(obj, 'name')          # Tests if attribute exists
```

Example:

```
if hasattr(obj, 'x'):
    x = getattr(obj, 'x'):
else:
    x = None
```

*Note: `getattr()` also has a useful default value *arg.*

```
x = getattr(obj, 'x', None)
```

Exercises

Exercise 4.9: Better output for printing objects

Modify the `Stock` object that you defined in `stock.py` so that the `__repr__()` method produces more useful output. For example:

```
>>> goog = Stock('GOOG', 100, 490.1)
>>> goog
Stock('GOOG', 100, 490.1)
>>>
```

See what happens when you read a portfolio of stocks and view the resulting list after you have made these changes. For example:

```
>>> import report
>>> portfolio = report.read_portfolio('Data/portfolio.csv')
>>> portfolio
... see what the output is ...
>>>
```

Exercise 4.10: An example of using getattr()

`getattr()` is an alternative mechanism for reading attributes. It can be used to write extremely flexible code. To begin, try this example:

```
>>> import stock
>>> s = stock.Stock('GOOG', 100, 490.1)
>>> columns = ['name', 'shares']
>>> for colname in columns:
        print(colname, '=', getattr(s, colname))

name = GOOG
shares = 100
>>>
```

Carefully observe that the output data is determined entirely by the attribute names listed in the `columns` variable.

In the file `tableformat.py`, take this idea and expand it into a generalized function `print_table()` that prints a table showing user-specified attributes of a list of arbitrary objects. As with the earlier `print_report()` function, `print_table()` should also accept a `TableFormatter` instance to control the output format. Here's how it should work:

```
>>> import report
>>> portfolio = report.read_portfolio('Data/portfolio.csv')
>>> from tableformat import create_formatter, print_table
>>> formatter = create_formatter('txt')
>>> print_table(portfolio, ['name','shares'], formatter)
      name     shares
---------- ----------
        AA        100
       IBM         50
       CAT        150
      MSFT        200
        GE         95
      MSFT         50
       IBM        100

>>> print_table(portfolio, ['name','shares','price'], formatter)
      name     shares      price
---------- ---------- ----------
        AA        100       32.2
       IBM         50       91.1
       CAT        150      83.44
      MSFT        200      51.23
        GE         95      40.37
      MSFT         50       65.1
       IBM        100      70.44
>>>
```

Contents | Previous (4.2 Inheritance) | Next (4.4 Exceptions)

Fork me on GitHub