Python Data Structures:

Special Methods:

Whenever we are writing any code in python, what we are actually doing is complex combination of the built-in methods in python. It was obvious right? I mean, python is complex C, which is complex machine code at some point, so when developing any tool with python, what we are doing is actually using the python built in methods to create new ones that what actually do is to concatenate a series of actions using those “magic methods”.

Of course, those methods that I’m talking about are underscore-underscore methods, like the \_\_len\_\_ method. These methods can be easily spotted because the have leading and trailing underscores, because of that, these methods are also known by other names, for example the len method, when we are referring to \_\_len\_\_ we say that this is the dunder-len.

Accent

Python is a programing language, and as in any language we might have an accent when we are starting to learn to deal with it. What this means in the programing world is that we only use the methods that are common for other languages, and this has nothing to do with the fact that we learn other languages before. This is because whenever we use any language most of the time we use the common features among all of them instead of using the program language’s **unique skills**, this is what in python we call wring code in a “**Pythonic way**”.

Iterators:

An iterator class object is an object that we can well, iterate on it, this means that the for loop is available to use on it. But how do we know when an object is iterable? What does that mean? How do these objects work under the hood? How does python manage them?

Let’s just get our hands dirty with an example:

In this example we will create a simple class which is perfect to analyze an iterator, a card Deck. We’ll use a named tuple from the collections module to create the class and then we’ll redefine the original methods, so they return what we want for this specific class.  
The aim of this example is to show how can we manipulate more methods beyond the always used \_\_init\_\_ method which is the class constructor.

import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:

    ranks = [str(n) for n in range(2, 11)] + list('JQKA')

    suits = 'spades diamonds clubs hearts'.split()

    def \_\_init\_\_(self):

        self.\_cards = [Card(rank, suit) for suit in self.suits

for rank in self.ranks]

    def \_\_len\_\_(self):

        return len(self.\_cards)

    def \_\_getitem\_\_(self, position):

        return self.\_cards[position]

This is a very simple example where we have created a class from a namedtuple. This class has a self.\_cards method which is in fact our iterable because as we can see, we have two nested for loops which create an iterable for all the possible options in a Deck.

Python 3.9.5 (default, May 18 2021, 14:42:02) [MSC v.1916 64 bit (AMD64)] ::

Anaconda, Inc. on win32

Type "help", "copyright", "credits" or "license" for more information.

>>>beer\_card = Card('7', 'diamonds')

>>>beer\_card

Card(rank='7', suit='diamonds')

Then we have our \_\_len\_\_ method redefined so it will return the actual length of the Deck, which is gonna be 52 of course, since we are return ling the length of that list.

>>> deck = FrenchDeck()

>>> len(deck)

52

The \_\_getitem\_\_ method is also redefined. As we can see this method takes an argument named position. Now this is getting interesting. In the return, we see the position argument that we are taking inside braquets. This means that the method can be used like normal.

>>> deck[0]

Card(rank='2', suit='spades')

>>> deck[-1]

Card(rank='A', suit='hearts')

Now let’s move on to use another python integrated function that’d be really useful for this example, the choice function, from the random module.

>>>from random import choice

>>>choice(deck)

Card(rank='3', suit='hearts')

>>>choice(deck)

Card(rank='K', suit='spades')

This is possible because deck is an instance of a class that is an iterable object, this means that the choice function is available to use.

We’ve just seen two advantages of using special methods to leverage the Python data model:

• The users of your classes don’t have to memorize arbitrary method names for standard operations (“How to get the number of items? Is it .size(), .length(), or what?”).

• It’s easier to benefit from the rich Python standard library and avoid reinventing the wheel, like the random.choice function.

But it gets better.

Because our \_\_getitem\_\_ delegates to the [] operator of self.\_cards, our deck automatically supports slicing. Here’s how we look at the top three cards from a brand-new deck, and then pick just the aces by starting on index 12 and skipping 13 cards at a time:

>>> deck[:3]

[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'), Card(rank='4', suit='spades')]

>>> deck[12::13]

[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'), Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]

Just by implementing the \_\_getitem\_\_ special method, our deck is also iterable:

>>> for card in deck:

... print(card)

Card(rank='2', suit='spades')

Card(rank='3', suit='spades')

Card(rank='4', suit='spades')

...

The deck can also be iterated in reverse:

>>> for card in reversed(deck):

... print(card)

Card(rank='A', suit='hearts')

Card(rank='K', suit='hearts')

Card(rank='Q', suit='hearts')

...

How about sorting?

A common system of ranking cards is by rank (with aces being highest), then by suit in the order of spades (highest), then hearts, diamonds, and clubs (lowest). Here is a function that ranks cards by that rule, returning 0 for the 2 of clubs and 51 for the ace of spades:

suit\_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades\_high(card):

    rank\_value = FrenchDeck.ranks.index(card.rank)

    return rank\_value \* len(suit\_values) + suit\_values[card.suit]

How Special Methods Are Used

The first thing to know about special methods is that they are meant to be called by the Python interpreter, and not by you. You don’t write my\_object.\_\_len\_\_(). You write len(my\_object) and, if my\_object is an instance of a user-defined class, then Python calls the \_\_len\_\_ instance method you implemented. But for built-in types like **list**, **str**, **bytearray**, and so on, the interpreter takes a short‐ cut: the CPython implementation of len() actually returns the value of the **ob\_size** field in the **PyVarObject** C struct that represents any variable-sized built-in object in memory. This is much faster than calling a method. More often than not, the special method call is implicit. For example, the statement for i in x: actually causes the invocation of iter(x), which in turn may call x.\_\_iter\_\_() if that is available. Normally, your code should not have many direct calls to special methods. Unless you are doing a lot of metaprogramming, you should be implementing special methods more often than invoking them explicitly. The only special method that is frequently called by user code directly is \_\_init\_\_, to invoke the initializer of the superclass in your own \_\_init\_\_ implementation.

A Closer Look at the iter Function:

As we’ve seen, Python calls iter(x) when it needs to iterate over an object x. But iter has another trick: it can be called with two arguments to create an iterator from a regular function or any callable object. In this usage, the first argument must be a callable to be invoked repeatedly (with no arguments) to yield values, and the second argument is a sentinel: a marker value which, when returned by the callable, causes the iterator to raise StopIteration instead of yielding the sentinel. The following example shows how to use iter to roll a six-sided die until a 1 is rolled:

>>> def d6():

... return randint(1, 6)

...

>>> d6\_iter = iter(d6, 1)

>>> d6\_iter

>>> for roll in d6\_iter:

... print(roll)

...

4

3

6

3

Note that the iter function here returns a callable\_iterator. The for loop in the example may run for a very long time, but it will never display 1, because that is the sentinel value. As usual with iterators, the d6\_iter object in the example becomes use‐ less once exhausted. To start over, you must rebuild the iterator by invoking iter(…) again.

Emulating numeric types:

We can’t just emulate a built-in function like len(), for example, we can also emulate and modify the way that python uses the operators. Why? Well, everything in python is an object, including the operators, they are instances of the object class, and we can also change how do they work depending on the enviroment that we put them. We can see that better in the following example:

Several special methods allow user objects to respond to operators such as +. Here our goal is to further illustrate the use of special methods through another simple example. We will implement a class to represent two-dimensional vectors—that is Euclidean vectors like those used in math and physics:

Diagram and output:

Graphical user interface, application, Word

Description automatically generated

Output X

>>> v1 = Vector(2, 4)

>>> v2 = Vector(2, 1)

>>> v1 + v2 Vector(4, 5)

>>> v = Vector(3, 4)

>>> abs(v) 5.0

>>> v \* 3 Vector(9, 12)

>>> abs(v \* 3) 15.0

OK that’s nice but, What is really going on under the hood?

The truth is that what we did was to create a class named Vector() which redefines several original methods in order to function this way. The methods that were modified in this case are:

\_\_init\_\_, \_\_repr\_\_, \_\_abs\_\_, \_\_bool\_\_, \_\_add\_\_ and \_\_mul\_\_.

This is what is really happening under the hood:

from math import hypot

class Vector:

    def \_\_init\_\_(self, x=0, y=0):

        self.x = x

        self.y = y

    def \_\_repr\_\_(self):

        return 'Vector(%r, %r)' % (self.x, self.y)

    def \_\_abs\_\_(self):

        return hypot(self.x, self.y)

    def \_\_bool\_\_(self):

        return bool(abs(self))

    def \_\_add\_\_(self, other):

        x = self.x + other.x

        y = self.y + other.y

        return Vector(x, y)

    def \_\_mul\_\_(self, scalar):

        return Vector(self.x \* scalar, self.y \* scalar)

As we can see, all these are original functions, these include also the “+” and the “\*” operand, and if we wanted, we cud also add the “/” and the “-” operand.

Note that although we implemented four special methods (apart from \_\_init\_\_), none of them is directly called within the class or in the typical usage of the class illustrated by the console listings. As mentioned before, the Python interpreter is the only frequent caller of most special methods. In the following sections, we discuss the code for each special method.

Digging Deeper:

String Representation:

The \_\_repr\_\_ special method is called by the repr built-in to get the string representation of the object for inspection. If we did not implement \_\_repr\_\_, vector instances would be shown in the console like . The interactive console and debugger call repr on the results of the expressions evaluated, as does the %r placeholder in classic formatting with the % operator, and the !r conversion field in the new Format String Syntax used in the str.format method.

Note that in our \_\_repr\_\_ implementation, we used %r to obtain the standard representation of the attributes to be displayed. This is good practice, because it shows the crucial difference between Vector(1, 2) and Vector('1', '2')—the latter would not work in the context of this example, because the constructor’s arguments must be numbers, not str. The string returned by \_\_repr\_\_ should be unambiguous and, if possible, match the source code necessary to re-create the object being represented. That is why our chosen representation looks like calling the constructor of the class (e.g., Vector(3, 4)). Contrast \_\_repr\_\_ with \_\_str\_\_, which is called by the str() constructor and implicitly used by the print function. \_\_str\_\_ should return a string suitable for display to end users. If you only implement one of these special methods, choose \_\_repr\_\_, because when no custom \_\_str\_\_ is available, Python will call \_\_repr\_\_ as a fallback.

Arithmetic Operators

The example of the vector implements two operators: + and \*, to show basic usage of \_\_add\_\_ and \_\_mul\_\_. Note that in both cases, the methods create and return a new instance of Vector, and do not modify either operand —self or other are merely read. This is the expected behavior of infix operators: to create new objects and not touch their operands.

Boolean Value of a Custom Type:

Although Python has a bool type, it accepts any object in a boolean context, such as the expression controlling an if or while statement, or as operands to and, or, and not. To determine whether a value x is truthy or falsy, Python applies bool(x), which always returns **True** or **False**. By default, instances of user-defined classes are considered truthy, unless either \_\_bool\_\_ or \_\_len\_\_ is implemented. Basically, bool(x) calls x.\_\_bool\_\_() and uses the result. If \_\_bool\_\_ is not implemented, Python tries to invoke x.\_\_len\_\_(), and if that returns zero, bool returns False. Otherwise, bool returns True.

Our implementation of \_\_bool\_\_ is conceptually simple: it returns **False** if the magnitude of the vector is zero, **True** otherwise. We convert the magnitude to a Boolean using bool(abs(self)) because \_\_bool\_\_ is expected to return a boolean. Note how the special method \_\_bool\_\_ allows your objects to be consistent with the truth value testing rules defined in the “Built-in Types” chapter of The Python Standard Library documentation.

Why len is not a method

As we said before, len(x) runs very fast when used in built-in data structures like str, list, memoryview and so on. Here no method is called for the built-in CPython: the length is simply a field in a C struct. Getting the number of items of a collection is a simple common operation and must work efficiently so that’s why we do it this way.

In other words, len is not called as a method because it gets special treatment as part of the Python data model, just like abs. But thanks to the special method \_\_len\_\_, you can also make len work with your own custom objects. This is a fair compromise between the need for efficient built-in objects and the consistency of the language. Also, from The Zen of Python:

*“Special cases aren’t special enough to break the rules.”*

Special method names (operators excluded)

|  |  |
| --- | --- |
| Category | Method names |
| String/bytes representation | \_\_repr\_\_, \_\_str\_\_, \_\_format\_\_, \_\_bytes\_\_ |
| Conversion to number | \_\_abs\_\_, \_\_bool\_\_, \_\_complex\_\_, \_\_int\_\_, \_\_float\_\_, \_\_hash\_\_,\_\_index\_\_ |
| Emulating collections | \_\_len\_\_, \_\_getitem\_\_, \_\_setitem\_\_, \_\_delitem\_\_, \_\_contains\_\_ |
| Iteration | \_\_iter\_\_, \_\_reversed\_\_, \_\_next\_\_ |
| Emulating callables | \_\_call\_\_ |
| Context management | \_\_enter\_\_, \_\_exit\_\_ |
| Instance creation and destruction | \_\_new\_\_, \_\_init\_\_, \_\_del\_\_ |
| Attribute management | \_\_getattr\_\_, \_\_getattribute\_\_, \_\_setattr\_\_, \_\_delattr\_\_, \_\_dir\_\_ |
| Attribute descriptors | \_\_get\_\_, \_\_set\_\_, \_\_delete\_\_ |
| Class services | \_\_prepare\_\_, \_\_instancecheck\_\_, \_\_subclasscheck\_\_ |

Special method names for operators

|  |  |
| --- | --- |
| Category | Method names |
| Unary numeric operators | \_\_neg\_\_ -, \_\_pos\_\_ +, \_\_abs\_\_ abs() |
| Rich comparison operators | \_\_lt\_\_ >, \_\_le\_\_ <=, \_\_eq\_\_ ==, \_\_ne\_\_ !=, \_\_gt\_\_ >, \_\_ge\_\_ >= |
| Arithmetic operators | \_\_add\_\_+, \_\_sub\_\_ -, \_\_mul\_\_ \*, \_\_truediv\_\_ /, \_\_floordiv\_\_ //, \_\_mod\_\_ %, \_\_divmod\_\_ divmod() , \_\_pow\_\_ \*\* or pow(), \_\_round\_\_ round() |
| Reversed arithmetic operators | \_\_radd\_\_, \_\_rsub\_\_, \_\_rmul\_\_, \_\_rtruediv\_\_, \_\_rfloordiv\_\_, \_\_rmod\_\_, \_\_rdivmod\_\_, \_\_rpow\_\_ |
| Augmented assignment  arithmetic operators | \_\_iadd\_\_, \_\_isub\_\_, \_\_imul\_\_ ,\_\_itruediv\_\_, \_\_ifloordiv\_\_, \_\_imod\_\_, \_\_ipow\_\_ |
| Bitwise operators | \_\_invert\_\_ ~, \_\_lshift\_\_ <>, \_\_and\_\_ &, \_\_or\_\_ |, \_\_xor\_\_ ^ |
| Reversed bitwise operators | \_\_rlshift\_\_, \_\_rrshift\_\_, \_\_rand\_\_, \_\_rxor\_\_, \_\_ror\_\_ |
| Augmented assignment bitwise operators | \_\_ilshift\_\_, \_\_irshift\_\_, \_\_iand\_\_, \_\_ixor\_\_, \_\_ior\_\_ |

Summary:

We can modify the original methods by redefining them inside of a class and modify their behavior.

When we call the main functions like len() or an operand like “+”, we are actually calling original methods like x.\_\_len\_\_() or x.\_\_add\_\_(other).

We don’t call the methods explicitly like x.\_\_len\_\_() we call them implicitly like len(x), is the python interpreter who call them for us.

The expression “for i in x” is actually calling iter(x), which is calling x.\_\_iter\_\_().

The iter() method can accept a up to two arguments, the first is got to be a callable, (meaning no argumenta and yieldable) while the second one is a stop sign.

The \_\_repr\_\_() function changes the way that the class is represented, instead of returning the object type and memory location, it will return what ever we define. Is lightly different than \_\_str\_\_() which is the one called when we use the print() function.

We can change all the arithmetic operators like “+”, “-” etc. We can do so by modifying the original function they are interpreted from. These functions are: \_\_add\_\_+, \_\_sub\_\_ -, \_\_mul\_\_ \*, \_\_truediv\_\_ /, \_\_floordiv\_\_ //, \_\_mod\_\_

%, \_\_divmod\_\_ divmod() , \_\_pow\_\_ \*\* or pow(), \_\_round\_\_ round()

We can even define a boolean type by taking advantage of how the boolean work. If the boolean returns a True or False type, it uses it. If not the python interpreter will use the \_\_len\_\_() method and test if the length is more than zero, if it is, it will return True, if is not, it will return False.

By implementing special methods, your objects can behave like the built-in types, enabling the expressive coding style the community considers Pythonic.

A basic requirement for a Python object is to provide usable string representations of itself, one used for debugging and logging, another for presentation to end users. That is why the special methods \_\_repr\_\_ and \_\_str\_\_ exist in the data model.