In [7]:
```
import collections
Card=collections.namedtuple('Card',['rank','suit'])
Card
```

Out[7]:
```
__main__.Card
```

In [5]:
```
class FrenchDeck:
    ranks= [str(r) for r in range(2,11)]+list("JQKA")
    suits= 'spade 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]
```

The first thing to note is the use of collections.namedtuple to construct a simple class to represent individual cards. We use namedtuple to build classes of objects that are just bundles of attributes with no custom methods

In [10]:
```
Card('1','diamonds')
```

Out[10]:
```
Card(rank='1', suit='diamonds')
```

In [11]:
```
deck=FrenchDeck()
```

In [12]:
```
deck
```

Out[12]:
```
<__main__.FrenchDeck at 0x1f3066957c0>
```

In [13]:
```
deck.suits
```

Out[13]:
```
['Spade', 'Diamonds', 'Clubs', 'Hearts']
```

In [14]:
```
deck.ranks
```

Out[14]:
```
['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
```

In [16]:
```
len(deck)
```

Out[16]:
```
52
```

Reading specific cards from the deck—say, the first or the last—is easy, thanks to the $\_\_getitem\_\_$ method:

In [18]:
```
deck[0]
```

Out[18]:
```
Card(rank='2', suit='Spade')
```

In [19]:
```
deck[-1]
```

Out[19]:
```
Card(rank='A', suit='Hearts')
```

In [20]:
```
# picking a random card
import random
random.choice(deck)
```

Out[20]:
```
Card(rank='10', suit='Hearts')
```

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

• Users of your classes don’t have to memorize arbitrary method names for stan- dard 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.

Because our $\_\_getitem\_\_$ delegates to the [] operator of self._cards, our deck automatically supports slicing.

In [22]:
```
deck[:3]
```

Out[22]:
```
[Card(rank='2', suit='Spade'),
 Card(rank='3', suit='Spade'),
 Card(rank='4', suit='Spade')]
```

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

In [25]:
```
deck._cards[3]
```

Out[25]:
```
Card(rank='5', suit='Spade')
```

# How Special Methods Are Used

You write len(my_object) and, if my_object is an instance of a user-defined class, then Python calls the **len** method you implemented.

But the interpreter takes a shortcut when dealing for built-in types like list, str, bytearray, or extensions like the NumPy arrays. Python variable-sized collections written in C include a struct called PyVarObject, which has an ob_size field holding the number of items in the collection. So, if my_object is an instance of one of those built-ins, then len(my_object) retrieves the value of the ob_size field, and this is much faster than calling a method.

In [27]:
```
import math
class Vector:
    def __init__(self,x=0,y=0):
```

```python
        self.x=x
        self.y=y
    def __repr__(self):
        return f'Vector({self.x!r},{self.y!r})'
    def __abs__(self):
        return math.hypot(self.x,self.y)
    def __bool__(self):
        return bool(abs(self))
    def __add__(self,other):
        self.x=self.x+other.x
        self.y=self.y+other.y
        return Vector(x,y)
    def __mul__(self,scalar):
        return Vector(self.x*scalar,self.y*scalar)
```

The $\_\_repr\_\_$ special method is called by the repr built-in to get the string represen- tation of the object for inspection. Without a custom $\_\_repr\_\_$, Python's console would display a Vector instance <Vector object at 0x10e100070>.

Basically, bool(x) calls x.$\_\_bool\_\_$()

In [ ]: