CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 7

Tuesday, September 28th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- ***Object Oriented Programming:*** data encapsulation, inheritance, polymorphism
- ***Classes:*** base classes and derived classes
- ***Inheritance and Polymorphism:*** class methods, interfaces, method resolution order

OUTLINE

- The concept for consistency in the `python` language:
  - The `python` data model
  - Special class methods (also called *"dunder"* methods)
- A custom sequence: French deck of cards
- Software Licenses

FEW COMMENTS ON DUCK TYPING

- When using duck typing, you are specifying an *implicit* interface.
- Duck typing can speed up the short-term development process as sometimes you do not have a clear picture of a current design.
- Explicit software design (interface is defined before implementation work starts) is more stable especially in large projects. It is more difficult because it requires thinking further into the future compared to an implicit duck typing approach
- When you have an implicit interface through duck typing, make sure to write extensive tests.

THE `python` DATA MODEL

The `python` data model formalizes ***the interface of the building blocks of the language itself***.

What are the *"building blocks of the language itself"*? Examples include:

- Obtain the *length* (number of elements) in a sequence.
- Index an element in a sequence, e.g., `s[0]` ("`s`" is just a name for a sequence, e.g. a `list`, `tuple`, etc.)
- Add two objects together, e.g. `a + b` (any arithmetic operation in fact). The `python` interpreter must call some function in place of the "`+`" operator, e.g. `add(a, b)`. The `python` data model defines how this should be done.
- Serialization and deserialization of objects; is used to create binary representations of the current state of your data (e.g. a checkpoint or restart file for a physics simulation). In `python` this is called "pickling" and "unpickling".

THE `python` DATA MODEL

Many people who work with `python` value its ***consistency***, which is enabled through the `python` data model.

What does "consistency" mean in a programming language?

- After a while of working with the language, you develop an intuition that allows you to correctly *guess* the behavior of a feature that is new to you.
- This consistency is partly achieved by the use of *built-in* functions, some of them you have already met.
  Example: to get the *length* of a sequence in `python`, you would write `len(s)` where `s` can be any type of a sequence: `list`, `tuple` or your user-defined class for example. The *interface* is consistently defined through the built-in `len()`.
- In contrast, this interface ***is not*** consistent in `C++`. You get the length of a `std::vector` through the `.size()` method, another library might use `.length()` or `.len()` for its container type.

THE python DATA MODEL

You may have heard the term "*pythonic*". When you solve a problem in your python code such that you *exploit and maintain* the consistency principles implied by the python data model, then your approach ***is pythonic***.

Example: iteration over a sequence in python

Pythonic

```
for item in iterable:
    # pythonic for-loop
    print(item)
```

```
for i,item in enumerate(iterable):
    # if you need the iteration index i, y
    # should use the enumerate() built-in
    print(item, i)
```

Not pythonic

```
for i in range(len(iterable)):
    # C-style code, also inefficient
    print(iterable[i])
```

SPECIAL METHODS (A.K.A. DUNDER METHODS)

There are a few (about 80, *more than half of them* are arithmetic, bitwise and comparison operators) special methods which are the *backbone* of the `python` data model. All of them take the form `__methodname__`, where the leading and trailing *double underscores* have special meaning in `python`. Developers would call such a method "under under methodname under under" which is tedious and therefore the term "dunder methodname" has been coined.

See the introduction to Chapter 1 in *Fluent Python: Clear, Concise, and Effective Programming* by Luciano Ramalho (O'Reilly Media, 2015)

DUNDER: STRING REPRESENTATION OF OBJECTS

We start with a special method that is required by any `python` object. These two methods allow for the *string representation* of objects.

- Why do we need them?
  - `__repr__(self)`: is used to obtain a string by calling `repr()` which can be used with the `eval()` built-in to reproduce the instance of the object. This dunder method is ***required*** by all objects and often useful for debugging.
  - `__str__(self)`: is used to obtain a *pretty printable* string for the object. You have called this dunder method many times already, i.e., whenever you call `print()` the `python` data model resolves a call to `__str__()`. This dunder method is not strictly required. The data model will fallback to `__repr__()` if it is not implemented.

DUNDER: STRING REPRESENTATION OF OBJECTS

Why did my code work even if I did not implement `__repr__(self)`? *Because every object in `python` implicitly inherits from a base class called "`object`".*

`python 2.x` (two are different)

```python
class MyClass:
    # in python 2.x
    # classic-style class, has no bases.
    # should have stopped using code like
    # a long time ago.

class MyClass(object):
    # in python 2.2 onwards
    # new-style class, `object` is base
```

`python 3.x` (all the same)

```python
# in python 3.x
class MyClass:
    # new-style class, `object` is base

class MyClass():
    # new-style class, `object` is base

class MyClass(object):
    # new-style class, `object` is base
```

If you write `python 3` code that must be compatible with `python 2` (for the time being), you ***must explicitly inherit*** from `object`, i.e., "`class MyClass(object):`"

DUNDER: STRING REPRESENTATION OF OBJECTS

- You now see how consistency is enabled in `python`: special (dunder) methods are used to implement class behavior at the *low-level*.
- In *user-level* code (everything at the high level that uses objects) *you are supposed to use the appropriate built-in functions*. E.g., `len()`, `print()`, `+`, `-`, `*`, `/`, and so on.
- ***It is not pythonic to call dunder methods directly in user-level code.*** Compare:

```
length = s.__len__()
```
wrong!

```
length = len(s)
```
correct!

Because of optimization, `len()` is also faster if used with built-in types!

DUNDER EXAMPLES: STRING REPRESENTATION

```python
class MyClass:
    """
    Test class that implements __repr__() and __str__(). Note that we
    implicitly inherit from the object base class.
    """
    def __init__(self, value):
        """Instance construction"""
        self.state = value
```

Verify that we inherit from `object`:

```
>>> print(MyClass.__bases__)
(<class 'object'>,)
```

What happens if we try to print an instance?

```
>>> c = MyClass(0)
>>> print(c)
<__main__.MyClass object at 0x7f8467a4bd30> # string representation inherited from `obje
```

Seems to work and looks like this is the default we inherit from `object`.

DUNDER EXAMPLES: STRING REPRESENTATION

```python
class MyClass:
    """
    Test class that implements __repr__() and __str__(). Note that we
    implicitly inherit from the object base class.
    """
    def __init__(self, value):
        """Instance construction"""
        self.state = value

    def __repr__(self):
        """String representation, reproducible with eval()"""
        class_name = type(self).__name__ # what does type() do?
        instance_state = self.state
        return f"{class_name}({instance_state})"
```

What happens now if we print an instance?

```
>>> c = MyClass(0)
>>> print(c)
MyClass(0)
>>> repr(c)
'MyClass(0)'
```

DUNDER EXAMPLES: STRING REPRESENTATION

```python
class MyClass:
    """
    Test class that implements __repr__() and __str__(). Note that we
    implicitly inherit from the object base class.
    """
    def __init__(self, value):
        """Instance construction"""
        self.state = value

    def __repr__(self):
        """String representation, reproducible with eval()"""
        class_name = type(self).__name__
        instance_state = self.state
        return f"{class_name}({instance_state})"

    def __str__(self):
        """String representation for user-level pretty print"""
        class_name = type(self).__name__
        instance_state = self.state
        return f"An instance of {class_name} with self.state={instance_state}"
```

DUNDER EXAMPLES: STRING REPRESENTATION

```python
class MyClass:
    """
    Test class that implements __repr__() and __str__(). Note that we
    implicitly inherit from the object base class.
    """
    def __init__(self, value):
        """Instance construction"""
        self.state = value

    def __repr__(self):
        """String representation, reproducible with eval()"""
        class_name = type(self).__name__
        instance_state = self.state
        return f"{class_name}({instance_state})"

    def __str__(self):
        """String representation for user-level pretty print"""
        class_name = type(self).__name__
        instance_state = self.state
        return f"An instance of {class_name} with self.state={instance_state}"
```

Now we get this:

```python
>>> c = MyClass(0)
>>> print(c)
MyClass(0)
>>> repr(c)
'MyClass(0)'
>>> c = MyClass(0)
>>> print(c) # user-level string repr.
An instance of MyClass with self.state=0
>>> repr(c) # low-level string repr.
'MyClass(0)'
```

Observations

- The `print()` built-in looks for `__str__()` and falls back to `__repr__()` if the former does not exist. (Proof for what I said at the beginning.)
- The `repr()` built-in only looks for `__repr__()` which has to exist. (***Recall:*** we always inherit from `object` implicitly in `python 3.x`)
- ***Why is the return value of `repr(c)` quoted?***

- The return value of `repr()` should ideally work with the `eval()` built-in:

```python
>>> c_repr = eval(repr(c)) # reproduce c
>>> repr(c_repr)
'MyClass(0)'
```

- ***Note:***
  - `__repr__()` is for *low-level purpose*: debugging and development.
  - `__str__()` is for *user-level purpose*: create a string representation that is *informative* for the user.

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

- There are various dunder methods for arithmetic operations
- If you want to add objects together you have to implement the `__add__` method. This method will then be called instead of the `+` operator.

A class for a thing that can `__add__` things:

```python
class Thing:
    """Simple class for a 'thing'"""
    def __init__(self, thing):
        self.state = thing

    def __str__(self):
        return f"{self.state}"

    def __add__(self, other):
        """This method implements addition '+'"""
        print(self.state)
        return Thing(f"{str(self)} + {str(other)}") # What is happening here?
```

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

A class for a thing that can __add__ things:

```python
class Thing:
    """Simple class for a 'thing'"""
    def __init__(self, thing):
        self.state = thing

    def __str__(self):
        return f"{self.state}"

    def __add__(self, other):
        """This method implements addition '+'"""
        print(self.state)
        return Thing(f"{str(self)} + {str(other)}") # What is happening here?
```

We can now do the following:

```
>>> A = Thing('A')
>>> B = Thing('B')
>>> C = Thing('C')
>>> D = A + B + C
A
A + B
>>> print(D)
A + B + C
```

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

We can now do the following:

```
>>> A = Thing('A')
>>> B = Thing('B')
>>> C = Thing('C')
>>> D = A + B + C
A
A + B
>>> print(D)
A + B + C
```

In which order does* `python` *evaluate the '+' operators?

Left to right or right to left?

The statement `D = A + B + C` is equivalent to

```
>>> D = A.__add__(B).__add__(C) # NEVER WRITE CODE LIKE THIS ON THE USER-LEVEL!
```

```
>>> D = A + (B + C) # same as A.__add__(B.__add__(C))
B
A
>>> print(D)
A + B + C
```

Make sure you understand what is happening here. Study line 12 in the class definition of `Thing`.

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

- `python` also supports *augmented assignment operators*, these dunder methods are prepended with an "`i`".
- An augmented assignment is: `A += B`
- This is *identical* to: `A = A + B`

How would you implement this operator?

```python
class Thing:
    def __iadd__(self, other):
        """This method implements augmented addition assignment '+='"""
        pass # How do we implement this operator?
```

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

```python
class Thing:
    """Simple class for a 'thing'"""
    def __init__(self, thing):
        self.state = thing

    def __str__(self):
        return f"{self.state}"

    def __add__(self, other):
        """This method implements addition '+'"""
        print(self.state)
        return Thing(f"{str(self)} + {str(other)}")

    def __iadd__(self, other):
        """This method implements augmented addition assignment '+='"""
        print(self.state)
        self.state = f"{str(self)} + {str(other)}"
        return self
```

Make sure you understand what we did in `__iadd__`*. What is the implication of* `return self`*?*

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

```python
class Thing:
    def __add__(self, other):
        """This method implements addition '+'"""
        print(self.state)
        return Thing(f"{str(self)} + {str(other)}")

    def __iadd__(self, other):
        """This method implements augmented addition assignment '+='"""
        print(self.state)
        self.state = f"{str(self)} + {str(other)}"
        return self
```

Now we can do the following:

```
>>> A = Thing('A')
>>> B = Thing('B')
>>> C = Thing('C')
>>> A += B
A
>>> A += C
A + B
>>> print(A)
A + B + C
```

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

```python
class Thing:
    def __add__(self, other):
        """This method implements addition '+'"""
        print(self.state)
        return Thing(f"{str(self)} + {str(other)}")

    def __iadd__(self, other):
        """This method implements augmented addition assignment '+='"""
        print(self.state)
        self.state = f"{str(self)} + {str(other)}"
        return self
```

Compare the difference:

Addition resolves to (A = A + B):

```
>>> A = A.__add__(B)
```

Augmented addition resolves to (A += B):

```
>>> A = A.__iadd__(B) # re-assigns self if __iadd__ returns self
```

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

- We are now able to understand what happened in Lecture 5 when we studied the "pass by assignment" mechanics for variables passed to functions.
- At the beginning of that lecture we were experimenting with a `list` (instead of a `Thing`). We were discussing this example on pythontutor.
- Let's revisit this example with our `Thing` class.

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

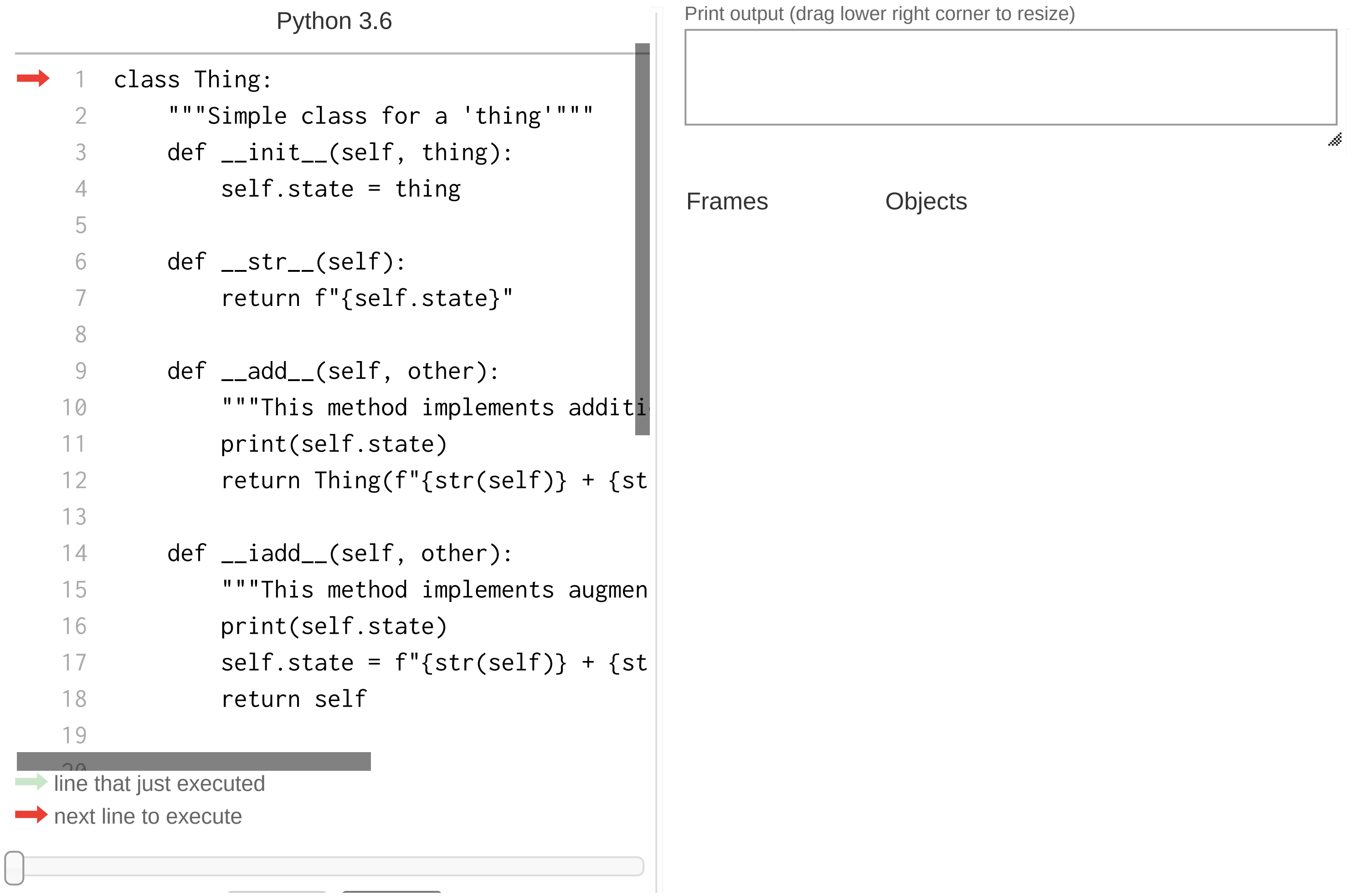
```python
def mutate(x):
    y = Thing('B')
    x += y  # calls __iadd__
    return x

def rebind(x):
    y = Thing('C')
    x = x + y  # calls __add__ which returns new object that is re-bound to x
    return x

A = Thing('A')
B = mutate(A)  # after this function call A and B are the same object
C = rebind(A)  # after this function call A and C are two different objects
```

DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

Example on pythontutor



Python 3.6

```
class Thing:
    """Simple class for a 'thing'"""
    def __init__(self, thing):
        self.state = thing

    def __str__(self):
        return f"{self.state}"

    def __add__(self, other):
        """This method implements additi
        print(self.state)
        return Thing(f"{str(self)} + {st

    def __iadd__(self, other):
        """This method implements augmen
        print(self.state)
        self.state = f"{str(self)} + {st
        return self

```

🟩 line that just executed
🟥 next line to execute
Print output (drag lower right corner to resize)
Frames
Objects

THE `python` DATA MODEL

- The power of `python` stems from its *data model*, *inheritance*, *composition* and *delegation*.
- User-defined classes that follow these principles can reuse other `python` that implement a certain functionality.
- ***For example:*** if a user-defined class behaves like a sequence, all utility functions that apply to a `list` or `tuple` would also apply to the user-defined sequence.

SEQUENCE: `list` OR `tuple`

- Both `list` and `tuple` are *sequences*.
- A sequence has a number of elements and therefore a *length*.
- You can *index* a sequence to get a specific element.
- The difference between `list` and `tuple` is that the latter is *immutable* (you can not change the values in a `tuple`).

```
>>> s = ('e0', 'e1', 'e2', 'e3')
>>> len(s)  # length of sequence `s`
4
>>> s[0]  # retrieve the first element in `s`
'e0'
>>> s[0] = 'f0'  # if s was a list, this would work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

SEQUENCE: namedtuple

- The `tuple` is a useful data type. If your code allows, prefer a `tuple` over a `list` because it is faster (but immutable).
- You can make a `tuple` behave like a `struct` in `C` using `namedtuple`'s
- You can use `dir()` to print a list of valid attributes of `python` objects.

```python
from collections import namedtuple

Point = namedtuple('Point', 'x y z')
p = Point(0, 1, 2)
```

```
>>> p[0]  # like a normal python tuple
0
>>> p.y  # by field name like a C-struct
1
>>> len(p)
3
>>> dir(p)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__dir__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
'__sizeof__', '__slots__', '__str__', '__subclasshook__', '_asdict', '_field_defaults',
'_make', '_replace', 'count', 'index', 'x', 'y', 'z']
```

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

- In this example we are going to put together what we have learned so far and see the `python` data model in action for a real application.
- We are implementing a French deck of playing cards which behaves like a sequence.
- The minimum requirements for a sequence are the `__len__()` and `__getitem__()` dunder methods.

The following is Example 1-1 in *Fluent Python: Clear, Concise, and Effective Programming* by Luciano Ramalho (O'Reilly Media, 2015)

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

We model a playing card using a `namedtuple`:

```python
from collections import namedtuple

Card = namedtuple('Card', ['rank', 'suit'])
```

Which allows us to create cards like that:

```python
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

A basic deck of cards might look like this:

```python
from collections import namedtuple

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

class FrenchDeck:
    """French deck of 52 playing cards"""
    ranks = [str(rank) for rank in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        """Initialize ordered deck of cards with list-comprehension"""
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __str__(self):
        """Pretty print card deck"""
        map_utf8 = {'clubs': '♣', 'diamonds': '♦', 'hearts': '♥', 'spades': '♠'}
        cpl = 13  # number of cards per printed line
        pretty = []
        for line in range((len(self._cards) + cpl - 1) // cpl):
            for card in self._cards[line * cpl : (line + 1) * cpl]:
                pretty.append(f"  {card.rank:>2}{map_utf8[card.suit]}")
            pretty.append('\n')
        return ''.join(pretty).rstrip()
```

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

- We can create an ordered set of cards and print the deck using `print()`:

```
>>> deck = FrenchDeck()
>>> print(deck)
   2♠   3♠   4♠   5♠   6♠   7♠   8♠   9♠  10♠   J♠   Q♠   K♠   A♠
   2♦   3♦   4♦   5♦   6♦   7♦   8♦   9♦  10♦   J♦   Q♦   K♦   A♦
   2♣   3♣   4♣   5♣   6♣   7♣   8♣   9♣  10♣   J♣   Q♣   K♣   A♣
   2♥   3♥   4♥   5♥   6♥   7♥   8♥   9♥  10♥   J♥   Q♥   K♥   A♥
```

- But if we want to get the length of our deck or index a specific card we get an error:

```
>>> len(deck)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'FrenchDeck' has no len()
>>> deck[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'FrenchDeck' object is not subscriptable
```

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

We now know how to fix these errors:

```python
class FrenchDeck:
    """French deck of 52 playing cards"""
    ranks = [str(rank) for rank in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        """Initialize ordered deck of cards with list-comprehension"""
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        """Return length of deck"""
        return len(self._cards)

    def __getitem__(self, index):
        """Return card at index"""
        return self._cards[index]
```

That was easy:

```
>>> len(deck)
52
>>> deck[12]
Card(rank='A', suit='spades')
```

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

- Adding the `__len__()` and `__getitem__()` dunder methods implements the basic functionality for a sequence.
- ***What does that mean:*** the `__len__()` and `__getitem__()` methods make our sequence ***iterable***. We can therefore use the `in` operator to test if a card is in the deck:

```
>>> Card(rank='A', suit='hearts') in deck
True
>>> Card(rank='A', suit='joker') in deck
False
```

`for` -loops will also work out of the box:

```
>>> for card in deck: # in-operator used with for-loops
...     print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
stripped output...
```

Because `__getitem__()` *delegates* the `[]` operator to the `list` used for `self._cards`, we can even use ***slicing***:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'), Card(rank='4', suit='spades')]
```

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

- We added *very few lines of code* to our `FrenchDeck` but get an impressive amount of functionality for free (`python` data model).
- It does not stop here. Because our custom sequence is iterable, we can also use functions from the `python` standard library. If you want to draw random cards, don't look further:

```
>>> from random import choice
>>> choice(deck)
Card(rank='Q', suit='hearts')
>>> choice(deck)
Card(rank='2', suit='diamonds')
```

- How about shuffling the deck of cards:

```
>>> from random import shuffle
>>> shuffle(deck)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.9/random.py", line 362, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'FrenchDeck' object does not support item assignment
```

How can we fix this?

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

We implement `__setitem__(self, index, card):`

```python
class FrenchDeck:
    """French deck of 52 playing cards"""
    ranks = [str(rank) for rank in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        """Initialize ordered deck of cards with list-comprehension"""
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        """Return length of deck"""
        return len(self._cards)

    def __getitem__(self, index):
        """Return card at index"""
        return self._cards[index]

    def __setitem__(self, index, card):
        """Set a card at index"""
        self._cards[index] = card  # this method does not return a value
```

A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

We implement `__setitem__(self, index, card)`:

```python
class FrenchDeck:
    """French deck of 52 playing cards"""
    ranks = [str(rank) for rank in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __setitem__(self, index, card):
        """Set a card at index"""
        self._cards[index] = card  # this method does not return a value
```

...and shuffle again:

```
>>> shuffle(deck); print(deck)
   6♥   K♦   3♥   Q♣   Q♦   9♥   J♦   Q♥  10♠   J♠   9♠   A♥  10♦
   8♣   8♦  10♥   6♣   5♦   6♦   3♣   3♠   7♠   4♥   9♦   K♣   8♥
   A♠   A♦   2♠   7♣   7♥   2♣   6♠   K♥   8♠   3♦  10♣   9♣   5♥
   Q♠   4♠   4♦   2♦   5♠   5♣   J♣   2♥   7♦   A♣   4♣   K♠   J♥
>>> shuffle(deck); print(deck)
  10♣   4♦   2♣   Q♠   J♠   K♣   J♦   5♣   A♠   K♦   K♠   Q♥   9♥
   A♦   7♠   5♥   7♦   Q♦   3♥   2♦   A♣   6♦   7♣   4♠   A♥   5♠
   3♠   9♦   J♥   6♣   9♠   Q♣  10♦   8♠   J♣   8♥   7♥   4♥   K♥
   4♣   2♠   3♦  10♥   9♣   6♥   3♣   8♣   5♦   8♦   2♥  10♠   6♠
```

You can download the `FrenchDeck` code here

SOFTWARE LICENSES

SOFTWARE LICENSES

Every software project should be licensed.

- An open source license protects contributors and users.
- If you publish your code in the public domain without a license, a third party would be free to take your code and build a business on top of it (possibly asking users to pay for it), without reimbursing you or at least giving you credit for your intellectual work.
- There are many different licenses available. A good starting point is https://choosealicense.com/.
- There are *copyright* and *copyleft* licenses.

Copyright

Copyleft

SOFTWARE LICENSES: COPYLEFT

Copyleft is a general method for making a program (or other work) free (in the sense of freedom, not “zero price”), and requiring all modified and extended versions of the program to be free as well.

- A copyleft program is copyrighted, with additional distribution terms, which are a legal instrument that gives everyone the rights to use, modify, and redistribute the program's code or any program derived from it but only if the distribution terms are ***unchanged***.
- An example of a copyleft license is the GNU General Public License v3.0 (GNU GPLv3)
- If you have a code under the GNU GPLv3, all libraries you use in that code *must* also be copyleft.
- All contributions and modifications *must preserve* copyright and license notices.

SOFTWARE LICENSES: COPYLEFT

- The `bash` shell is an example of a program under the GPL license.
- It was licensed under GNU GPLv2 until version `3.2`.
- ...and the reason why MacOSX was stuck with `bash 3.2` for so long (it was released in 2007).
- With MacOSX Catalina (2019), the shell has been changed from `bash` to `zsh` which uses the MIT license.
- The MIT license is much more permissive than the GNU GPLv3 and allows companies to make modifications to open source code that may not necessarily be shared with the public.
- Licensing your code gives you legal rights on how someone else is allowed to use your work. It is a very import part of a software project.

SOFTWARE LICENSES: GITHUB CONTROVERSIES

- GitHub has been acquired by Microsoft in 2018.
- Microsoft tries to push open source but naturally, there is also commercial interest.
- One critique is that GitHub *does not* force you to select a license when you create a new repository.
- Another recent controversy is GitHub's copilot: an AI based pair-programmer that may be integrated in tools such as Visual Studio to give you suggestions as you code.
- The problem with copilot is that is uses data (software projects) from GitHub to train its model. Many of those projects are licensed appropriately. Since the copilot injects code (which it learned from other licensed projects) into other software projects there is concern that this process violates legal rights of the original authors.
- (Optional reading): The Free Software Foundation (FSF) point of view.

SOFTWARE LICENSES

The open source initiative: Licenses and Standards
https://opensource.org/licenses

An extensive list of available licenses for free and open or collaborative software:
https://spdx.org/licenses/

RECAP

- The concept for consistency in the `python` language:
  - ***The `python` data model***—the foundation of consistency
  - ***Special class methods***—also called *"dunder"* methods
- A ***custom*** sequence: French deck of cards— `python` data model in action
- Software Licenses

One of the beauties in `python` is its consistency implied by the data model. Dunder methods are an important part to enable this consistency which will help you develop an *intuition* for how a `python` object that is new to you should behave.