# SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

## LECTURE 8

*Fabian Wermelinger*
Harvard University
CS107 / AC207

Tuesday, September 27th 2022

## LAST TIME

- Object Oriented Programming (OOP)
- Python classes
- Inheritance and Polymorphism

## TODAY

Main topics: ***Python data model, consistency, Dunder methods, Software licenses***

***Details:***

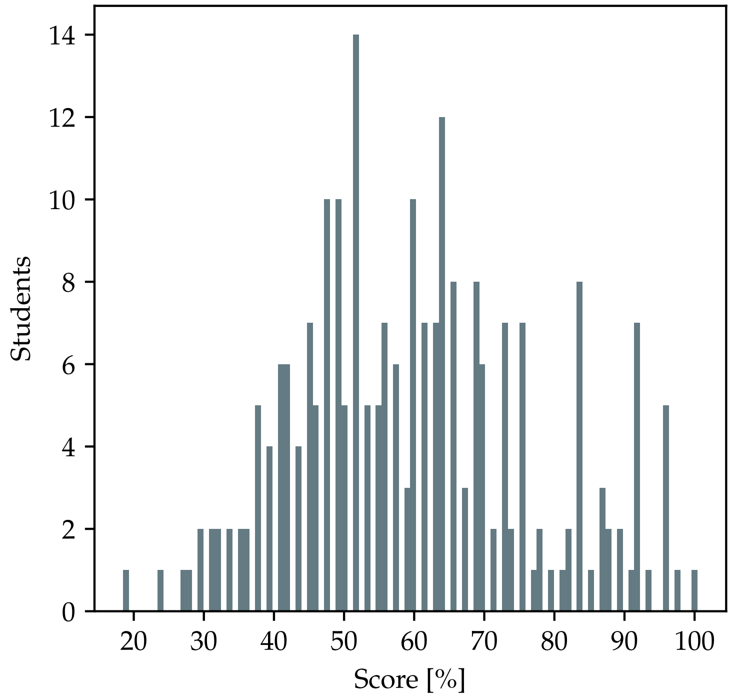
- The concept for consistency in the Python language:
  - The Python data model
  - Special class methods (also called *"dunder"* methods)
- A custom sequence example: French deck of cards
- Software Licenses

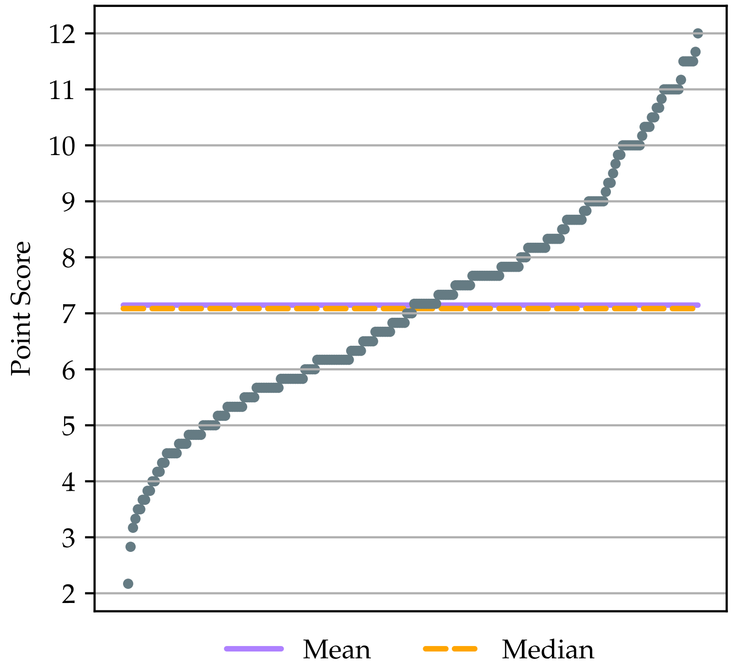
***FASRC computing resources:***

- We have an allocation on the FASRC academic cluster for the class. It allows you to run a Jupyter notebook on a compute node on the cluster. Should you have GUI issues with WSL or Docker you can use this as a backup. See this post for a video tutorial: https://edstem.org/us/courses/24296/discussion/1820667?comment=4163977
- ***Code you submit in CS107/AC207 must be Python code.*** Jupyter notebooks are not accepted for grading.

# QUIZ 1 RESULTS

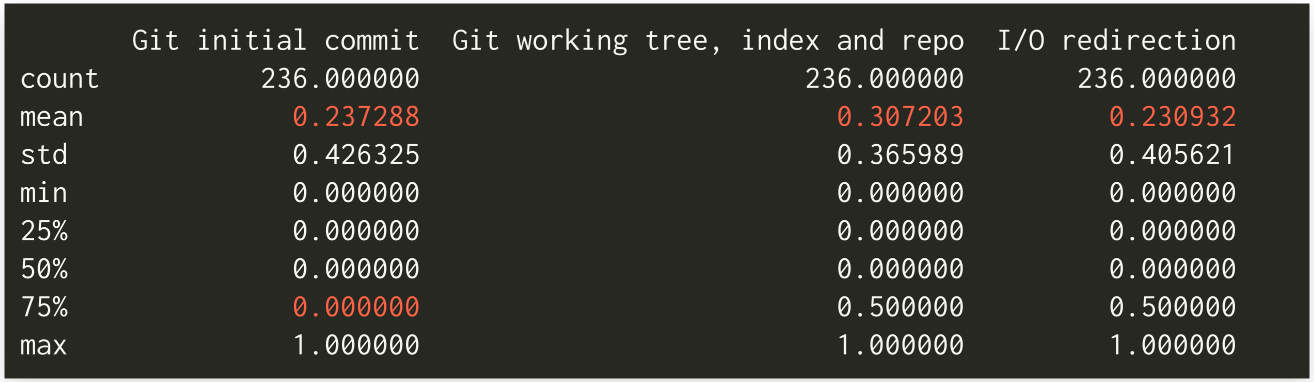
See: https://edstem.org/us/courses/24296/discussion/1828671





```
              Score           %   N correct  N incorrect
count  236.000000  236.000000  236.000000   236.000000
mean     7.146441   59.553672    8.084746     3.915254
std      2.041592   17.013267    1.911115     1.911115
min      2.170000   18.083333    4.000000     0.000000
25%      5.670000   47.250000    7.000000     3.000000
50%      7.085000   59.041667    8.000000     4.000000
75%      8.330000   69.416667    9.000000     5.000000
max     12.000000  100.000000   12.000000     8.000000
```

# QUIZ 1 QUESTION BREAKDOWN



```
       Git initial commit  Git working tree, index and repo  I/O redirection
count          236.000000                        236.000000       236.000000
mean             0.237288                          0.307203         0.230932
std              0.426325                          0.365989         0.405621
min              0.000000                          0.000000         0.000000
25%              0.000000                          0.000000         0.000000
50%              0.000000                          0.000000         0.000000
75%              0.000000                          0.500000         0.500000
max              1.000000                          1.000000         1.000000
```

## *General observations:*

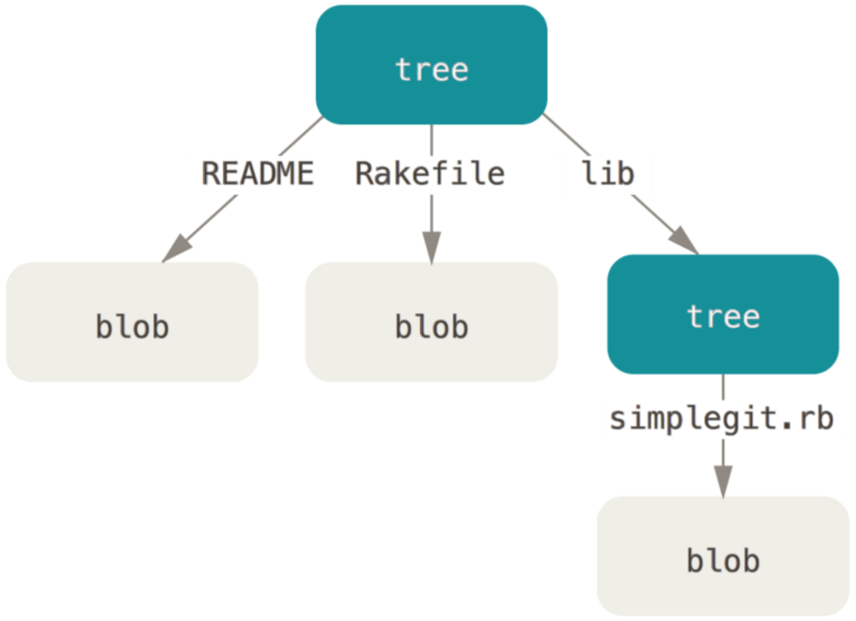
- Some students are struggling with basic Git commands and workflow. Please make sure you revisit the slides and use office hours and labs to discuss ambiguity.
- 3/4 of class did not score the point for the Git objects related question 😢. Understanding how Git works on the low-level will help you on the high-level (where you are in practice). Understanding the low-level is not extremely hard → ***this is a great read:*** https://jwiegley.github.io/git-from-the-bottom-up/

# QUIZ 1: GIT INITIAL COMMIT

You are about to create the very first commit in your Git repository with the following content

```
.
├── lib
│   └── simplegit.rb
├── Rakefile
└── README
```

You commit all of the files together in the same commit. How many of the fundamental Git objects will be created in the `.git/objects` directory (not shown in the repository structure above)? Assume the three files have different content.



- Three files have *different* content → 3 blobs
- There are 2 trees (repository root and `lib` directory)
- The last object will be the commit itself! → 1 object

3 + 2 + 1 = 6 objects

→ ***how many objects if files have same content?***

# QUIZ 1: GIT WORKING TREE, INDEX AND REPO

Assume the following *sequence* of commands

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README
        Rakefile
        lib/

nothing added to commit but untracked files present (use "git add" to track)
$ git add README
```

***Please select all that apply:***

- The `Rakefile` is in the Git working tree. (***false***)
- The `README` file *is not* in the Git working tree. (***true***)
- The `README` file is known to Git and exists in the `.git` repository. (***false***)
- The `README` file is in the staging area (also known as the index). (***true***)

- Initially *nothing* is tracked. *None of these files are in a Gi working tree.*
- `git add README` will place the file in the *index*. It has n been committed at this point → it can not be checked into a working tree.
- The working tree contains the *reconstructed* files of a particular *commit*. Untracked files do not belong to an commit. See https://edstem.org/us/courses/24296/discussion/18

## QUIZ 1: I/O REDIRECTION

You are executing the command

```
$ echo "Some output" >file 1>&2
```

***Select all that apply:***

- The file `file` is empty. (***true***)
- The file `file` contains `Some output`. (***false***)
- Standard output is redirected to standard error. (***true***)
- Standard error is redirected to standard output. (***false***)
- Standard input is redirected to standard error. (***false***)

***What happens:***

1. Shell evaluates left to right.
2. Shell sees "`>file`" → If no file descriptor is specified, the default is `1` (standard output). This will link `stdout` into `file`.
3. Shell sees "`1>&2`" → this will *redirect* file descriptor `1` to whatever file descriptor `2` (`stderr`) is pointing to.
4. After this, file descriptor `2` will no longer point into `file` and here is no data being routed into that file.

## COMMENTS ON DUCK TYPING

- When using duck typing, you are specifying an *implicit* interface. It 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 also more difficult to design because it requires thinking further into the future at the beginning of the project compared to an implicit duck typing approach where you could inject an interface on the fly.
- When you have an implicit interface through duck typing, make sure to write extensive tests.
- The Python data model is designed around duck typing: *If it walks like a duck and it quacks like a duck then it must be a duck.*

# THE PYTHON DATA MODEL

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

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

- 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 the ***consistency*** of the language, which is enabled because of 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 seen.
  ***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 this example.
- In contrast, such 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. You would need to read the documentation.

# THE PYTHON DATA MODEL

You have heard the term "*pythonic*" already. When you implement a 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:***

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

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

***Not pythonic:***

```python
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.

- Reserved classes and identifiers in Python: https://docs.python.org/3/reference/lexical_analysis.html#reserved-classes-of-identifiers
- Special method names: https://docs.python.org/3/reference/datamodel.html#specialnames

# 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 are they important?***

- `__repr__(self)`: is used to obtain a string by calling `repr()`. The returned string can be used (ideally) with the `eval()` built-in to *reproduce the instance* of the object. This dunder method is ***required by all Python 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 *delegates* the 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

*So why did my code work even if I did not implement `__repr__(self)`?*
Every object in Python implicitly inherits from a base class called `object`.

*Python 2.x* (the two are different)

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

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

*Python 3.x* (all 3 equivalent)

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

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

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

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

# DUNDER: STRING REPRESENTATION OF OBJECTS

- This is 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!***
  *(since all attributes are public you could do it!)*
  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`:*

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

*What happens if we try to print an instance?*

```python
>>> c = MyClass(0)
>>> print(c)
<__main__.MyClass object at 0x7f8467a4bd30>
```

*Works! The printed format is the default you inherit from the `object` base class.*

# 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}"
```

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

```
>>> c_repr = eval(repr(c)) # reproduce c
>>> repr(c_repr) # `c_repr` has same state as `c`
'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.

## *Now we get this:*

```
>>> 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.
- The `repr()` built-in only looks for `__repr__()` which *must* exist. (***Recall:*** we always inherit from `object` implicitly in `python 3.x`)
- ***Why is the return value of `repr(c)` quoted?***

## DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

- *There are various dunder methods for arithmetic operations.*
- ***Example:*** you want to add objects together you have to implement the __add__ method. The interpreter will call this method whenever the operator "+" appears in your code.

***A class for an abstract "thing" that can __add__ "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)}") # What is happening here?
```

# DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

*A class for an abstract "thing" that can __add__ "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)}") # 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 '+' operator?*

*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 # Why is this output different from before?
A # Why is this output different from before?
>>> 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__` (state of the instance is modified **internally** inside the object → OOP!). **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):

```python
>>> A = A.__add__(B) # rebind to new object (old reference lost)!
```

Augmented addition resolves to (A += B):

```python
>>> A = A.__iadd__(B) # rebind to self since __iadd__ returns self!
```

## DUNDER EXAMPLES: ADDING "THINGS" TOGETHER

- We are now able to understand what happened in Lecture 6 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 in mind.

# PYTHON BASICS: REFERENCE VARIABLES (LECTURE 6)

***But be careful when working with functions.*** Variables inside functions may become new objects depending on the operators you use:

```python
def f(x):
    x.append(7)  # member function of object x (internal transformation,
                 # append() may not even return self → no assignment here!)
    return x

def g(x):
    x += [7]  # translates to an operation internal to object x, must return
              # self because of assignment!
    return x

def h(x):
    x = x + [7]  # rebind x (new reference now local, old is lost!)
    return x

a = [1, 3, 5]
b = f(a)
c = g(a)
d = h(a)
```

# THE PYTHON DATA MODEL

- The power of Python stems from its *data model*, *inheritance*, *composition* and *delegation*.
- User-defined Python classes that follow these principles can reuse other Python code that implements a certain functionality.
- This is because the *special methods (dunder methods)* in Python implement the behavior of the object at low-level and we *access* these special methods on the user-level using *built-in functions exclusively*.
- ***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`

- What is the minimum *interface* requirement to model a sequence? ***Example:*** both `list`'s and `tuple`'s 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 it makes sense in your code, prefer a `tuple` over a `list` because it is faster.
- Use the `dir()` built-in to print a list of valid attributes in Python objects.

```python
from collections import namedtuple # behaves like a C-struct

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

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

# A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

- In this example we are going through the implementation of a custom sequence to put together what we have learned so far about Python data model.
- We are implementing a French deck of playing cards which behaves like a sequence.
- ***Recall:*** the minimum interface requirement for a sequence type 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 single playing card using a `namedtuple`:*

```python
from collections import namedtuple

Card = namedtuple('Card', ['rank', 'suit']) # properties of a playing card
```

*This allows us to create individual cards:*

```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

- 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
```

- ***Our type is not a sequence that follows the Python data model!***

# A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

*We need to implement the minimum interface requirement:*

```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) # delegate to list object → self._cards.__len__

    def __getitem__(self, index):
        """Return card at index"""
        return self._cards[index] # delegate to list object → self._cards.__getitem__
```

*That was easy because we can just delegate the operations:*

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

# A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

- ***What does that mean:*** the `__len__()` and `__getitem__()` methods complete the minimum sequence interface and make it ***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
```

Since the `in` operator works, `for`-loops will 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...
```

Finally, 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 because we followed the ***Python data model***.
- ***It does not stop here.*** Because our custom sequence is *iterable*, we can also use functions from the Python standard library right away. If you want to draw random cards, use existing Python code from the standard library:

```
>>> 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
```

→ ***the minimum sequence interface does not support this. Can it be fixed easily?***

## A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

*Sure, it requires another dunder method:*

```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 # Note: this method typically returns None
```

# A CUSTOM SEQUENCE: FRENCH DECK OF CARDS

## *Sure, it requires another dunder method:*

```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 # Note: this method typically returns None
```

## *...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 find the code for the French deck in the class repository.)

# SOFTWARE LICENSES

***Every software project must 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 open-source licenses available. A good place to start is https://choosealicense.com/. *You will have to choose a license for your project!*
- 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 *requires* 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 right to use, modify, and redistribute the program source code or any program derived from it ***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

***Example:*** Bash shell since v4.0 changed to GPLv3 (released 2009)

- Before that, Bash was at version 3.2 and used GPLv2 (released 2006).
- Apple OSX default shell was Bash. Since 2009 they could no longer make changes to the source code and keep it proprietary. So they were stuck with Bash v3.2 (3 years old by then).
- With Catalina in 2019, the shell has been changed from Bash to `zsh` which is licensed under 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 can be kept closed source. *It took Apple 10 years to implement this change.*

***Morale of story:*** licensing your code gives you legal rights on how someone else is allowed to use your work. It is a very important part of any software project and you should exercise these rights.

# SOFTWARE LICENSES: GITHUB CONTROVERSIES

- GitHub has been acquired by Microsoft in 2018.
- Microsoft claims to push open source projects but commercial interest is involved of course.
- A large critique from the open-source community is that GitHub ***does not*** force you to select a license when you create a new repository. By law they are not required to remind you of your rights, hence licenses can optionally be chosen when new repositories are created.
- This leads to issues with GitHub's copilot: an AI based pair-programmer tool that may be integrated in tools such as Visual Studio to give you suggestions while writing code.
- The problem with copilot is that the model is trained with code hosted in public GitHub repositories. Many of those projects are licensed appropriately. Since the copilot injects code into other *(proprietary)* projects, the licenses in the original projects are ***violated*** because the injected code is learned from correctly licensed projects. ***To read more:*** The Free Software Foundation (FSF) point of view

## SOFTWARE LICENSES

***Summary:***

- ***The open source initiative:*** Licenses and Standards https://opensource.org/licenses
- An extensive list of licenses for free and open or collaborative software: https://spdx.org/licenses/
- ***Starting point:*** https://choosealicense.com/.
- If patents are important, keep that in mind and choose an appropriate license that supports this requirement. The MIT license does not support patents resulting from your code.
- If you want to learn how to code, ***do not use copilot***. Never trust copilot *blindly*. *Use a real human for pair-programming.*

# 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 → with only few lines of code we already had a quite powerful implementation of a user defined sequence.
- Software Licenses

***Further reading:***

- 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 any new Python object you come across should behave. https://docs.python.org/3/reference/datamodel.html
- Chapter 1 in Luciano Ramalho, *"Fluent Python: Clear, Concise, and Effective Programming"*, O'Reilly Media, 2015