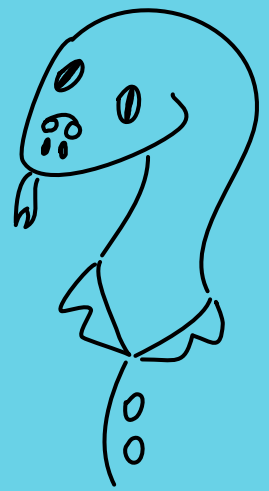
OOP in Python

David A. Clarke

Z02 – Software Development Center

21 July 2021



UNIVERSITÄT BIELEFELD
Fakultät für Physik

CRC-TR 211
Strong-interaction matter under extreme conditions

Foreword

The purpose of this talk is to

- give a basic introduction to object oriented programming (OOP)
- in the context of Python[1].

I assume you can already do some scientific programming and that you attended this year's introductory Python talk, but I don't assume you know object oriented programming.

This talk will be uploaded to Redmine, where you can find other excellent talks by the Software Development Center.

If you have questions or comments, please **interrupt me!**

[1] As usual, we use Python3.

Outline

1. The OOP paradigm
2. Basic examples in Python
3. Operator overloading
4. Wrap-up

non-OOP programming

As a grad student I used `Fortran77`. Allows for

1. arrays or variables of some kind; and
2. functions acting on them.

Let's call this non-OOP programming. OOP programming is a **generalization** of this and has:

1. objects, i.e. runtime entities use memory and have addresses;
2. functions (methods) associated with objects; and
3. classes, i.e. abstractions (or blueprints) of objects. Classes do not take up memory or have addresses.

The OOP paradigm

Main difference with non-OOP programming is using arbitrary structures instead of just arrays and variables. Some goals of the OOP paradigm:

- Representation: Try to make your code more closely **mirror how things look in the real world**. (Is the thing you're thinking of easily imagined as a simple array or variable?)
- Organization: **Collect methods and entities** related to some general idea. (Called encapsulation.)

The OOP paradigm

- Organization: **Establish hierarchies of classes**; one class may be a special case of another. (Called inheritance. Not only organizationally helpful, but also reduces code duplication.)
- Readability: **Hide implementation details**[2] from the programmer, minimizing what the programmer needs to provide. (Called abstraction.)

The hope is that your code becomes more intuitive, organized, and readable, which are all important for Clean Code!

[2] This can be a drawback when you need to know the implementation details exactly, because it adds extra steps when you "look inside". In my view this drawback is minor compared to benefits of readability and ease of use.

1. The OOP paradigm
2. Basic examples in Python
3. Operator overloading
4. Wrap-up

Anatomy of a CAT

You can think of a class as a collection of attributes and methods.

```
class CAT:

  "A class for good cats."

  # Constructor method
  def __init__(self, name):
    # CAT's instance attributes
    self._name = name
    self._isHungry = True
    self._stomachContents = ""
    print("And the Lord said 'Let there be "+self._name+"'.")

  # Destructor method
  def __del__(self):
    print(self._name+" has perished.")

  # A custom user-defined method
  def ignore(self):
    print(self._name+" ignores User...")
```

An object is an instantiation of a class.

```
print("User says 'hello' to "+niclai._name+".")
```

Anatomy of a CAT

Besides ignoring people, the only other things cats do are eat and meow. We can add this functionality to our class.

```
    def eat(self, food):
      if(self._isHungry):
        print(self._name+" eats "+food+".")
        self._stomachContents = food
        self._isHungry = False
      else:
        self.ignore()

    def speak(self):
      print(self._name+" says 'meow'.")

    def areYouHungry(self):
      if self._isHungry:
        self.speak()
```

Typical CAT behavior

Our CAT class, saved in `cat.py`, can be imported[3] and used in our main code. Use `__doc__` to access the description. The constructor is called when `niclai` is instantiated.

```
from cat import *

print(CAT.__doc__)

niclai = CAT("Niclai")
print("User says 'hello' to "+niclai._name+".")
niclai.speak()
print("User asks if "+niclai._name+" is hungry.")
niclai.areYouHungry()
print("User gives "+niclai._name+" some savory salmon.")
```

```
A class for good cats.
And the Lord said 'Let there be Niclai'.
User says 'hello' to Niclai.
Niclai says 'meow'.
User asks if Niclai is hungry.
Niclai says 'meow'.
```

[3] In the folder of your class, you will need an empty `__init__.py` file.

Typical CAT behavior

The destructor is called automatically when the program ends[4].

```
print("User gives "+niclai._name+" some savory salmon.")
niclai.eat("savory salmon")
print(niclai._name+" is full of "+niclai._stomachContents+".")
print("Here "+niclai._name+" have some more fish.")
niclai.eat("savory salmon")
print("Alright then.")
```

```
User gives Niclai some savory salmon.
Niclai eats savory salmon.
Niclai is full of savory salmon.
Here Niclai have some more fish.
Niclai ignores User...
Alright then.
Niclai has perished.
```

[4] The destructor will also be called if the object loses its reference, for example through reassignment. Try `cat=CAT("A")` followed by `cat=CAT("B")` in the interpreter. Thanks to Alessandro for pointing this out!

Inheritance

Inheritance is the process by which a class (the child class) inherits attributes and methods from a more general class (the parent).

```
class MEANCAT(CAT):

  "A class for naughty cats."

  def speak(self):
    print(self._name+" says 'HISS'!")
```

In the above we overwrote the `speak` method. `MEANCAT` `speak` is different than for `CAT`. We could have also added further attributes or methods.

Inheritance

If a cat is poorly behaved, remember that you can always destroy it by calling the destructor explicitly.

```
silky=MEANCAT("Silky")
print("User gives "+silky._name+" some tasty tuna.")
silky.eat("tasty tuna")
print(silky._name+" is full of "+silky._stomachContents+".")
print("Did you like that?")
silky.speak()
print("Wow "+silky._name+" that's rude. Begone!")
del silky
print("Much better.")
```

```
And the Lord said 'Let there be Silky'.
User gives Silky some tasty tuna.
Silky eats tasty tuna.
Silky is full of tasty tuna.
Did you like that?
Silky says 'HISS'!
Wow Silky that's rude. Begone!
Silky has perished.
Much better.
```

1. The OOP paradigm
2. Basic examples in Python
3. Operator overloading
4. Wrap-up

Operator overloading

Operator overloading[5] is where a binary operator such as $+$ is generalized to work for multiple types of operands.

- For instance $+$ already works with strings and floats.
- We can e.g. create $\mathrm{SU}(2)$ matrices and extend $+$ so that it also works with such objects.
- Helps code more closely mirror math notation.

[5] Operator overloading is a special case of polymorphism, where a function's implementation changes depending on the function arguments.

Example: $\mathrm{SU}(2)$ matrices

You can hide[6] attributes from being accessed from `main` by using `__`. `pass` can be used as a placeholder.

```
class SU2:

  "SU2 group elements as matrices."

  def one(self):
    self.__e00=complex(1.,0.)
    self.__e01=complex(0.,0.)
    self.__e10=complex(0.,0.)
    self.__e11=complex(1.,0.)

  def __init__(self):
    self.__e00=complex(1.,0.)
    self.__e01=complex(0.,0.)
    self.__e10=complex(0.,0.)
    self.__e11=complex(1.,0.)
```

[6] I think it is still possible to access these attributes, but not straightforwardly.

Example: $\mathrm{SU}(2)$ matrices

To access matrix elements from `main`, we create accessors that enforce that the matrix elements are `complex`[7].

```
    def setElement(self, k, l, value):
      if k==0 and l==0:
        self.__e00 = value+0j
      elif k==0 and l==1:
        self.__e01 = value+0j
      elif k==1 and l==0:
        self.__e10 = value+0j
      elif k==1 and l==1:
        self.__e11 = value+0j
      else:
        print("setElement: ERROR--invalid component.")
        exit(-1)
```

[7] Otherwise the user might accidentally overwrite, e.g., `e00` as a `float`, changing its type and possibly destroying some functionality.

Example: $SU(2)$ matrices

In python each binary operator has a special function name, like `__add__`[8]. To overload the $+$ operator we create the following method.

```
def __add__(self,other):
  g = SU2()
  g.setElement(0,0,self.__e00+other.__e00)
  g.setElement(0,1,self.__e01+other.__e01)
  g.setElement(1,0,self.__e10+other.__e10)
  g.setElement(1,1,self.__e11+other.__e11)
  return g
```

By editing the `__str__` function, we can control how our object is output to screen whenever it is passed to `print`.

```
def __str__(self):
  return ( f"{self.__e00} {self.__e01}\n"
          +f"{self.__e10} {self.__e11}"  )
```

[8] A table of all special function names for operator overloading can be found here.

Example: $\mathrm{SU}(2)$ matrices

Here we can see that access to the hidden attributes of our $\mathrm{SU}(2)$ class is blocked, as desired.

```
print(g.__e00)
```

```
Traceback (most recent call last):
  File "exampleCode/SU2.py", line 55, in <module>
    print(g.__e00)
AttributeError: 'SU2' object has no attribute '__e00'
```

Example: $\mathrm{SU}(2)$ matrices

Note that = is implemented by default[9]. We can manipulate `g` and `h` just as typical $\mathrm{SU}(2)$ matrices. `setElement` recasts 2 to `complex` as desired.

```
h = g
s = g + h
print(s)
g.one()
print(g)

g = SU2()
```

```
(4+0j) 0j
0j (2+0j)
(1+0j) 0j
0j (1+0j)
```

[9] One must be careful here, because this is not a copy constructor, but rather `h` and `g` are now references for the same object. So if one changes `h`, `g` will change as well. Thanks to Alessandro for pointing this out.

1. The OOP paradigm
2. Basic examples in Python
3. Operator overloading
4. Wrap-up

Summary

Utilize the power of OOP to:

- Make your code **mirror your mind**.
- Help **keep your code organized**.
- **Minimize repetition**.
- **Make your code easy to read and use**.

Here are also some helpful resources for

- 🔗Python OOP
- 🔗Operator overloading in Python

A look ahead...

Possible topics for the next lecture:

- `scipy`
- Parallelization
- Unix-like file manipulation
- Testing, logging, error handling
- Parameter files and command line arguments

Thanks for listening!