SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 7

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, September 22nd 2022

LAST TIME

- Python basics (a review)
- Nested environments
- Closures (functional programming)
- Decorators

TODAY

Main topics: ***Towards Object Oriented Programming (OOP) in Python, Classes, Inheritance and Polymorphism***

Details:

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

AGENDA CHECK:

- Quiz 1 takes place tonight. ***You have 15 minutes*** (12 questions). The quiz is available within a 45 minute time window (starting 7:00pm). ***If you start 35 minutes late, you will have 10 minutes to complete the quiz***. ***Please be on time***.

RECAP OF LAST TIME

Every name in Python is a reference to an object in memory

If you need a ***copy*** of an object, ***do it explicitly!***

```
import copy as cp

a = [1, 2, 3]
b = cp.copy(a)      # shallow copy
c = cp.deepcopy(a)  # deep copy

# difference between shallow and deep copy is for compound types
d = [1, 2, [3, 4, 5]]  # compound type
e = cp.copy(d)
f = cp.deepcopy(d)
```

RECAP OF LAST TIME

Every name in Python is a reference to an object in memory

If you need a ***copy*** of an object, ***do it explicitly!*** (pythontutor):

Python 3.6

```python
import copy as cp

a = [1, 2, 3]
b = cp.copy(a)  # shallow copy
c = cp.deepcopy(a)  # deep copy

# difference between shallow and deep co
d = [1, 2, [3, 4, 5]]
e = cp.copy(d)
f = cp.deepcopy(d)
```

Edit this code

line that just executed

next line to execute

< Prev Next >

Step 1 of 7

Visualized using **Python Tutor**

Customize visualization

Frames   Objects

GENERAL PYTHON GUIDELINES

Now that you are starting to write Python code, you should be aware of some important guidelines and resources:

- Become familiar with the *Python Enhancement Proposals* called PEP for short. All enhancements in Python go through these proposals.
- One of these early proposals is the Python coding style guide (PEP8). Some projects are very strict about code formatting. It is good practice to write *consistently* formatted code.
- The Zen of Python (PEP20) and some `vim` kōans (optional of course)

CODE FORMATTING

- It is generally good advice to have some background of PEP8 (you do not need to know all of it by heart).
- ***When you code, your mental focus should be on writing correct code and not worrying about consistent formatting!***
- You can use tools to do the formatting for you ***consistently***.

For Python code a good tool is called `yapf` (yet another Python formatter). When you work with C/C++ `clang-format` is a fantastic tool for code formatting. These tools can be integrated in your coding environment (e.g. `vim`) or used as standalone tools.

CODE FORMATTING

Example: `yapf` code formatting (demo in lecture codes)

```
python -m pip install [--user] yapf
```

- `yapf` looks for style files inside a source code directory (e.g. your `git` project) or a global configuration defined in your `$HOME`, see the formatting style section for all details. If it can't find any customizations it will default to PEP8.
- There are four base presets: (they are named after some companies because they are used there)
  1. `pep8` (default)
  2. `google` (based on the Google Python style guide)
  3. `yapf` (for use with Google open source projects)
  4. `facebook`
- Example `.style.yapf` file in the root of your `git` project:

```
[style]
based_on_style = pep8
spaces_before_comment = 4
split_before_logical_operator = True
use_tabs = False   # this is a personal choice!
```

- You can process many `*.py` files using a Bash script (e.g. in a project directory that is dedicated to maintenance). You can also integrate `yapf` in your editor (e.g. `vim`, `emacs`) to *automagically* format your code as you type (***remember:*** your focus is on ***correct code*** not worrying about formatting code).

OBJECT ORIENTED PROGRAMMING (MOTIVATION)

> *Programs = Algorithms + Data Structures*
>
> *Niklaus Wirth*

- We would like to find a way to represent complex, structured *data* in the context of our programming language.
- If you want to model a *particle* for example, you would want to associate every particle with a position `x, y, z` and possibly a velocity `u, v, w` for each of the three spatial coordinates.
- In C you could then define a compound ***data structure*** that describes a particle like this:

```c
struct Particle {
    double x, y, z; // particle position
    double u, v, w; // particle velocity
};
```

OBJECT ORIENTED PROGRAMMING (MOTIVATION)

> *Programs = Algorithms + Data Structures*
>
> *Niklaus Wirth*

- In C you could then define a compound ***data structure*** that describes a particle like this:

```c
struct Particle {
    double x, y, z; // particle position
    double u, v, w; // particle velocity
};
```

- In reality you will have many particles to deal with (think atoms for example). In addition to the `Particle` data structure, you would need an ***algorithm*** that, for example, describes the *interaction* between particles.
- The algorithm and data structure together would allow you to *formulate* a program to simulate this system of particles.

OBJECT ORIENTED PROGRAMMING (MOTIVATION)

Loosely speaking, we could say that we have defined an *abstract object* to describe a physical particle and we have called this "object" `Particle`. Any program we formulate with this abstraction will be *oriented* towards this object because it encodes the *data* we need in order to describe a physical process (using *algorithms*). When we write code using such abstract objects we speak of *Object Oriented Programming (OOP)*.

```
struct Particle {
    double x, y, z; // particle position
    double u, v, w; // particle velocity
};
```

OBJECT ORIENTED PROGRAMMING

- The C language *is not* an object oriented language!
- Although we can $\boxed{\text{encapsulate data in compound objects}}$, we can not form an *abstraction* of the data because the data in a `struct` is *always accessible by anyone* and we must *initialize* it explicitly.
- OOP goes *further* than that. We actually want to ***protect*** the data from being accessible by anyone.
- Instead, OOP offers an ***interface*** to perform certain operations with the data that is *private* to the object.
- An interface is defined through *methods* or *member functions* (synonyms) which are accessible by anyone (from the outside).
- This way, the actual ***implementation*** of how you perform the *transformation of the data* is hidden from the user of your object.
- The *Application Programming Interface (API)* of your code/library defines the interface you offer the users of your code and therefore the *abstraction* of your object.
- OOP further consists of concepts such as ***inheritance*** (inherit data and interfaces from other objects) and ***polymorphism*** (make objects with the same parent *behave* differently). The Greek word "poly" means "many". ευχαριστώ πολύ

CREATING OBJECTS IN PYTHON

- The basic type in Python that we can use to *compound* data *of possibly different types* is the `tuple`:

```python
def four():
    return 0x4

t = (1, 2.0, '3', four)
for item in t:
    print(type(item))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'function'>
```

- Can we do that with a `list` too?
- What is the difference between a `tuple` and a `list`?

CREATING OBJECTS IN PYTHON

Let's use the `tuple` to model a complex number:

```python
def Complex(r, i):
    """
    Construct a complex number
    Arguments:
        r: real part
        i: imaginary part
    """
    return (r, i)

# interface with complex numbers
def real(c):
    """Get the real part of a complex number c"""
    return c[0]

def imag(c):
    """Get the imaginary part of a complex number c"""
    return c[1]

def string(c):
    """Represent a complex number c as a string"""
    return f"{c[0]:e} + i{c[1]:e}"
```

- We can use our complex object like this:

```python
>>> z = Complex(1, 2)
>>> print(real(z), imag(z), string(z))
1 2 1.000000e+00 + i2.000000e+00
```

- But data is not *private* at all:

```python
>>> z[0]; z[1]
1
2
```

- It means that we can easily bypass the *interface* and do things on our own! No proper data encapsulation.
- Furthermore, a `tuple` is *immutable*:

```python
>>> z[0] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support ...
```

- Not being able to assign other values to a complex number is not useful at all.

CREATING OBJECTS IN PYTHON

Let's try a closure instead: (a closure *captures* the names from the enclosing scope)

```python
def Complex(r, i):
    """
    Construct a complex number
    Arguments:
        r: real part
        i: imaginary part
    """
    def implementation(method):
        if method.lower() == 'real':
            return r
        elif method.lower() == 'imag':
            return i
        elif method.lower() == 'string':
            return f"{r:e} + i{i:e}"

    return implementation
```

- We can use our complex object like this:

```python
>>> z = Complex(1, 2)
>>> print(z('real'), z('imag'), z('string'))
1 2 1.000000e+00 + i2.000000e+00
```

- This is better because the `implementation` (closure) defines all the *methods* allowed for data transformations of `Complex` objects.
- The implementation defines the *interface* for this example.
- We need to extend it such that we can *set* new values. These methods are called *setters*. The ones we already implemented are called *getters*.

CREATING OBJECTS IN PYTHON

Let's try a closure instead with setters:

```python
def Complex(r, i):
    """
    Construct a complex number
    Arguments:
        r: real part
        i: imaginary part
    """
    def implementation(method, z=None):
        nonlocal r, i
        if method.lower() == 'set_z':
            assert z is not None
            r, i = z
        elif method.lower() == 'real':
            return r
        elif method.lower() == 'imag':
            return i
        elif method.lower() == 'string':
            return f"{r:e} + i{i:e}"

    return implementation
```

- We can use our complex object like this:

```python
>>> z = Complex(1, 2)
>>> print(z('real'), z('imag'), z('string'))
1 2 1.000000e+00 + i2.000000e+00
>>> z('set_z', (3, 4))
>>> print(z('real'), z('imag'), z('string'))
3 4 3.000000e+00 + i4.000000e+00
```

- Now with support for setting new values through the `set_z` method.
- There are still many operations missing. ***How would we multiply two complex numbers?***
- The `nonlocal` keyword is used to reset a name (variable) that was bound in a different scope.
- You are studying the `nonlocal` keyword in more detail in a homework problem. (See also PEP3104)

CLASSES

- We have created our own object system before using tuples and closures.
- We figured that there are still limitations. For example, when working with complex numbers we would like to do arithmetic:

```
>>> z0 = Complex(1, 2)
>>> z1 = Complex(3, 4)
>>> z2 = z0 + z1
>>> print(z2('string'))
4.000000e+00 + i6.000000e+00
>>> z3 = z0 * z1
>>> print(z3('string'))
-5.000000e+00 + i1.000000e+01
```

- Python already implements a type system for *user defined* types which provides support for such operations.
- User defined types are called ***classes*** in OOP.

CLASSES

Let us return to the complex number type:

- The bare minimum (data only, no methods/interface):

```
class Complex():
    """Complex number type"""
    def __init__(self, real, imag):
        self.real = real  # real part
        self.imag = imag  # imaginary part
```

- `__init__`: is a special method automatically run when you execute code like this:

```
>>> z = Complex(1, 2) # calls the constructor of the class
>>> z.real
1
```

  For our purposes we call it the ***constructor*** of the class (see this thread for a lower level discussion).

- `self`: is a reference to the *instance* of the object. Note that unlike to the equivalent `this` pointer in C++, `self` *is passed explicitly* to any class method in Python (as the first argument). It is therefore ***not a keyword***. You can name it anything you want, "`self`" is the universally accepted convention. A good read by the creator of Python about the explicit nature of `self`.

CLASSES

Let us return to the complex number type: pythontutor

Python 3.6

```
class Complex():
    """Complex number type"""
    def __init__(self, real, imag):
        self.real = real  # real part
        self.imag = imag  # imaginary par

z = Complex(1, 2)
```

Edit this code

line that just executed

next line to execute

< Prev   Next >

Step 1 of 6

Visualized using **Python Tutor**

Customize visualization

Frames   Objects

CLASSES

An instance of a class is an allocated object:

- Printing the ***class type***:

```
>>> print(Complex)
<class '__main__.Complex'>
```

- Printing an ***instance of the class***:

```
>>> z = Complex(1, 2)
>>> print(z)
<__main__.Complex object at 0x7f669070c2e0>
```

An instance of a class is an object that *occupies memory*. The hex number is its memory address.

- Printing the type of an instance:

```
>>> print(type(z))
<class '__main__.Complex'>
```

CLASSES

Accessing class attributes:

- Private variables do not exist in Python.
- You can access class attributes with the "." operator:

```
>>> print(z.real); print(z.imag)
1
2
```

- Because Python does not prevent access to data, we can print all attributes of a class instance using the `vars` built-in:

```
>>> vars(z)
{'real': 1, 'imag': 2}
```

  which returns a Python *dictionary* with the attribute name and its current value.

- See this list of the Python built-in functions. Up to here we have seen two:
  - `type`: return the type of an object
  - `vars`: return the `__dict__` attribute for any object that implements it (we will discuss what this means in more detail later).

INHERITANCE AND POLYMORPHISM

- We have covered the main conception of object oriented programming and looked at how "data encapsulation" works in Python.
- At this point you should know how to create simple classes in your Python code.
- We have not talked about *inheritance* and *polymorphism* yet.
  Recall:
  - ***Inheritance:*** inherit *data and interfaces* from parent class(es).
  - ***Polymorphism:*** make objects with the same parent class(es) *behave* differently.

INHERITANCE AND POLYMORPHISM

- We have not talked about *inheritance* and *polymorphism* yet. ***Recall:***
  - ***Inheritance:*** inherit *data and interfaces* from parent class(es).
  - ***Polymorphism:*** make objects with the same parent class(es) *behave* differently.

A less abstract example: Both, a *cat* and a *dog* are *animals*. Because they are both animals, they share some common features that we describe by data (e.g. name, age, fur color, they speak, and so on). The two speak *different* languages, however. Cats *meow* and dogs *bark*.

- Dogs and cats *inherit* from an animal class → *inheritance*.
- Dogs and cats *behave* differently. Dogs bark and cats meow → *polymorphism*.

SUPER- AND SUBCLASSES

Base class (superclass): modeling cats and dogs

```python
class Animal():
    """Base class for animals"""
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):  # class method, note how `self` is passed as argument
        """Sounds animals can make"""
        raise NotImplementedError
```

- Our intention is to use the `Animal` class to *derive* specific animals from. It is called a ***base class*** or ***superclass***.
- Because the `Animal` class is *generic*, we do not know a priori what sounds an inherited animal instance will speak. It raises a `NotImplementedError`.
- Instances are created by calling the class with arguments defined in the `__init__` method:

```python
>>> animal = Animal("Generic animal", 5)  # instance names are lower case
>>> print(vars(animal))
{'name': 'Generic animal', 'age': 5}
```

SUPER- AND SUBCLASSES

Base class (superclass): modeling cats and dogs

```python
class Animal():
    """Base class for animals"""
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):  # class method, note how `self` is passed as argument
        """Sounds animals can make"""
        raise NotImplementedError
```

- Instances are created by calling the class constructor:

```python
>>> animal = Animal("Generic animal", 5)  # instance names are lower case
>>> print(vars(animal))
{'name': 'Generic animal', 'age': 5}
```

- But the `speak` method is not implemented and *the behavior is therefore not defined* and the `NotImplementedError` is thrown:

```python
>>> animal.speak()
Traceback (most recent call last):
  File ... # output omitted
    raise NotImplementedError
NotImplementedError
```

SUPER- AND SUBCLASSES

Derived class (subclass): modeling cats and dogs

- We can create *specific* animals by ***inheritance*** from the base class:

```python
class Dog(Animal):  # Dog is a derived class, it inherits from Animal
    """Dog is a derived Animal class"""
    def speak(self):
        """Special sounds dogs make"""
        return "Woof"

class Cat(Animal):  # Cat is a derived class, it inherits from Animal
    """Cat is a derived Animal class"""
    def __init__(self, name, age):  # re-define in the derived class
        self.name = f"A very special cat: {name}"  # cats have a custom name string

    def speak(self):
        """Special sounds cats make"""
        return "Meow"
```

- We can use a ***derived class*** in the same way as we did with a base class:

```python
>>> dog = Dog("Snoopy", 6)
>>> cat = Cat("Kitty", 4)
>>> print(vars(dog)); print(vars(cat))
{'name': 'Snoopy', 'age': 6}
{'name': 'A very special cat: Kitty'}  # Kitty has no age?
```

SUPER- AND SUBCLASSES

Derived class (subclass): modeling cats and dogs

- We can use a ***derived class*** in the same way as we did with a base class:

```
>>> dog = Dog("Snoopy", 6)
>>> cat = Cat("Kitty", 4)
>>> print(vars(dog)); print(vars(cat))
{'name': 'Snoopy', 'age': 6}
{'name': 'A very special cat: Kitty'}  # Kitty has no age?
```

- ***What is wrong with Kitty?*** We have written a special constructor for the `Cat` class (this is a common scenario), but we *did not inherit the* `age` *attribute from the base class!* (no data inheritance)
- The speak methods sure work!
- But the `age` attribute for cats is broken:

```
>>> print(cat.age)
Traceback (most recent call last):
  File "/home/fabs/harvard/CS107/lecture07/code/06.py", line 40, in <module>
    print(cat.age)
AttributeError: 'Cat' object has no attribute 'age'
```

SUPER- AND SUBCLASSES

Derived class (subclass): modeling cats and dogs

- The speak methods sure work!
- But the `age` attribute for cats is broken:

```
>>> print(cat.age)
Traceback (most recent call last):
  File "/home/fabs/harvard/CS107/lecture07/code/06.py", line 40, in <module>
    print(cat.age)
AttributeError: 'Cat' object has no attribute 'age'
```

- It seems that inheritance did not work properly for the derived `Cat` class. We will talk more about this in a few slides.

SUPER- AND SUBCLASSES

Calling member functions (methods): modeling cats and dogs

- Let us focus first on how calling class methods (member functions) works in Python.
- ***Recall:*** we have this for dogs:

```python
class Dog(Animal):
    """Dog is a derived Animal class"""
    def speak(self):  # takes one argument `self`
        """Special sounds dogs make"""
        return "Woof"
```

```python
>>> dog = Dog("Snoopy", 6)
>>> print(dog.speak())
Woof
```

SUPER- AND SUBCLASSES

Calling member functions (methods): modeling cats and dogs

- ```
  >>> dog = Dog("Snoopy", 6)
  >>> print(dog.speak())
  Woof
  ```

  We are calling the member function `speak()` ***without*** passing any arguments! ***Why is Python not complaining?***

- ***Remember:*** `self` is a reference to the object in memory. Since dog is *an instance of the Dog class*, the instance dog and `self` ***reference the same object***. If we were to add the following method to Dog:

  ```
  def my_id(self):
      print(id(self))
  ```

  then we get:

  ```
  >>> print(id(dog)); dog.my_id()
  140320390688928
  140320390688928
  ```
  ```

SUPER- AND SUBCLASSES

Calling member functions (methods): modeling cats and dogs

- *An instance* of a class has its methods ***bound*** to `self` and Python knows about it. *The first argument of class methods is therefore not needed when you call them.*

```
>>> print(dog.speak); print(Dog.speak)
<bound method Dog.speak of <__main__.Dog object at 0x7f9d32bdc0a0>>  # dog is an instance of Dog
<function Dog.speak at 0x7f9d32ad3160>                               # Dog is a class type
```

- We can use member functions either way: when *bound* to an instance of the class ***or*** by passing an instance to `Dog.speak`. In the latter case, Python does not have a valid `self` reference, so we ***must*** pass the instance as the first argument:

```
>>> print(dog.speak()); print(Dog.speak(dog))
Woof
Woof
```

- See:

```
>>> Dog.speak()
Traceback (most recent call last):
  File "/home/fabs/harvard/CS107/lecture07/code/06.py", line 42, in <module>
    Dog.speak()
TypeError: speak() missing 1 required positional argument: 'self'
```

SUPER- AND SUBCLASSES

Calling member functions (methods): modeling cats and dogs

Summary:

- `dog` was an *instance* of the `Dog` class. It is a reference to an object in memory.
- `dog.speak` is a *bound* member function. It is bound to the instance `dog`. In this case, the name `self` in the class code behaves as a reference to the *instance* `dog`.
- `Dog` was a class type. Member function definitions within this class always have *at least* one argument. The first argument is a reference to an instance of the class and is conventionally called `self`. *This argument will be consumed for bound methods, i.e., you do not need to pass it when calling the bound method.*
- `Dog.speak` is just a *regular function*. This function has one argument which is an *explicit* reference to an *instance* of the class `Dog`.
- You can use the `isinstance()` built-in to check if a name is an instance of a particular class. See the documentation here.

Step through code on pythontutor

SUPER- AND SUBCLASSES

Superclass initialization:

- We know that when we inherit from a base class we inherit its data and interface.
- If we recall the derived class `Cat` from earlier, we would expect when we create an instance of `Cat` that:

  1. The base class is initialized.
  2. Additional initialization of the derived class takes place.

- If we ***do not define*** another `__init__` method in the derived class, then `__init__` ***of the base class*** is called automatically.
- If we ***do define*** another `__init__` method in the *derived class* (because we need to perform other initialization there), then the superclass initialization ***is not performed automatically anymore*** (which is what happened to `Cat` (Kitty) earlier).
- In that case we have to use the `super()` built-in function to access the superclass explicitly.

SUPER- AND SUBCLASSES

Rule for polymorphism in Python:

If a method exists in the *base class as well as in the derived class* (both with the same name), then if an instance of the derived class calls the method, Python will execute the implementation of the method in the derived class and vice versa for a base class instance. The `super()` built-in can be used to call methods in the parent class. The actual method call is determined by the *Method Resolution Order (MRO)* specified in the `__mro__` attribute of the class *type*.

See this post for additional reading:
https://rhettinger.wordpress.com/2011/05/26/super-considered-super/

SUPER- AND SUBCLASSES

Example: rule for polymorphism in Python

```python
class Base():
    """Base class"""
    def __init__(self, a):
        self.a = a  # some data stored in the base class

    def explain(self):
        print(f"Executing from base class: data=`{self.a}`")

class Derived(Base):
    """Derived class"""
    def __init__(self, a, b):
        super().__init__(a)  # properly initialize the base class
        self.b = b  # some data specific to the derived class

    def explain(self):
        # 1. Call the base class method first
        super().explain()
        # 2. Then do special work required for the derived class
        print(f"Executing from derived class: data=`{self.b}`")
```

SUPER- AND SUBCLASSES

Example: rule for polymorphism in Python

- When you call `super()` it will follow this resolution order to find the matching method in the parent classes:

```
>>> print(Base.__mro__)
(<class '__main__.Base'>, <class 'object'>)
>>> print(Derived.__mro__)
(<class '__main__.Derived'>, <class '__main__.Base'>, <class 'object'>)
```

- Lets see how this works for the `explain()` method of an instance from the `Derived` class type (previous slide):

```
>>> a = "base class data"     # some data we pass for initializing the base class
>>> b = "derived class data" # some data we pass for initializing the derived class
>>> derived = Derived(a, b)  # here we create an instance of the derived class
>>> derived.explain()        # and we call the explain() method
Executing from base class: data=`base class data`
Executing from derived class: data=`derived class data`
```

SUPER- AND SUBCLASSES

Example: rule for polymorphism in Python

```
class Derived(Base):
    """Derived class"""
    def __init__(self, a, b):
        super().__init__(a)  # properly initialize the base class
        self.b = b  # some data specific to the derived class

    def explain(self):
        # 1. Call the base class method first
        super().explain()
        # 2. Then do special work required for the derived class
        print(f"Executing from derived class: data=`{self.b}`")
```

- In the `Cat` class from earlier, we ***did not*** initialize the base class like this, hence the `age` attribute was missing!
- If you don't do this, you will not enter the base class `__init__` method and ***consequently Python will not bind*** `self.a = a`. This causes an error whenever we try to access `self.a` later on, e.g. when we call the `explain()` method.
- This agreement to define class methods and attributes ***on the fly*** as execution flow encounters them, is called ***duck typing***. Two instances of a `Derived` class may suddenly have different attributes defined because of this.

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.*

RECAP

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

Further reading:

- Chapter 10 (Protocols and Duck Typing) and 12 in Luciano Ramalho, *"Fluent Python: Clear, Concise, and Effective Programming"*, O'Reilly Media, 2015
- Explicit `self` in Python: Why explicit `self` has to stay
- Method resolution order: `super()` considered super!