https://pynative.com/python/object-oriented-programming/

What is OOP?

OBJECT-ORIENTED PROGRAMMING IN PYTHON

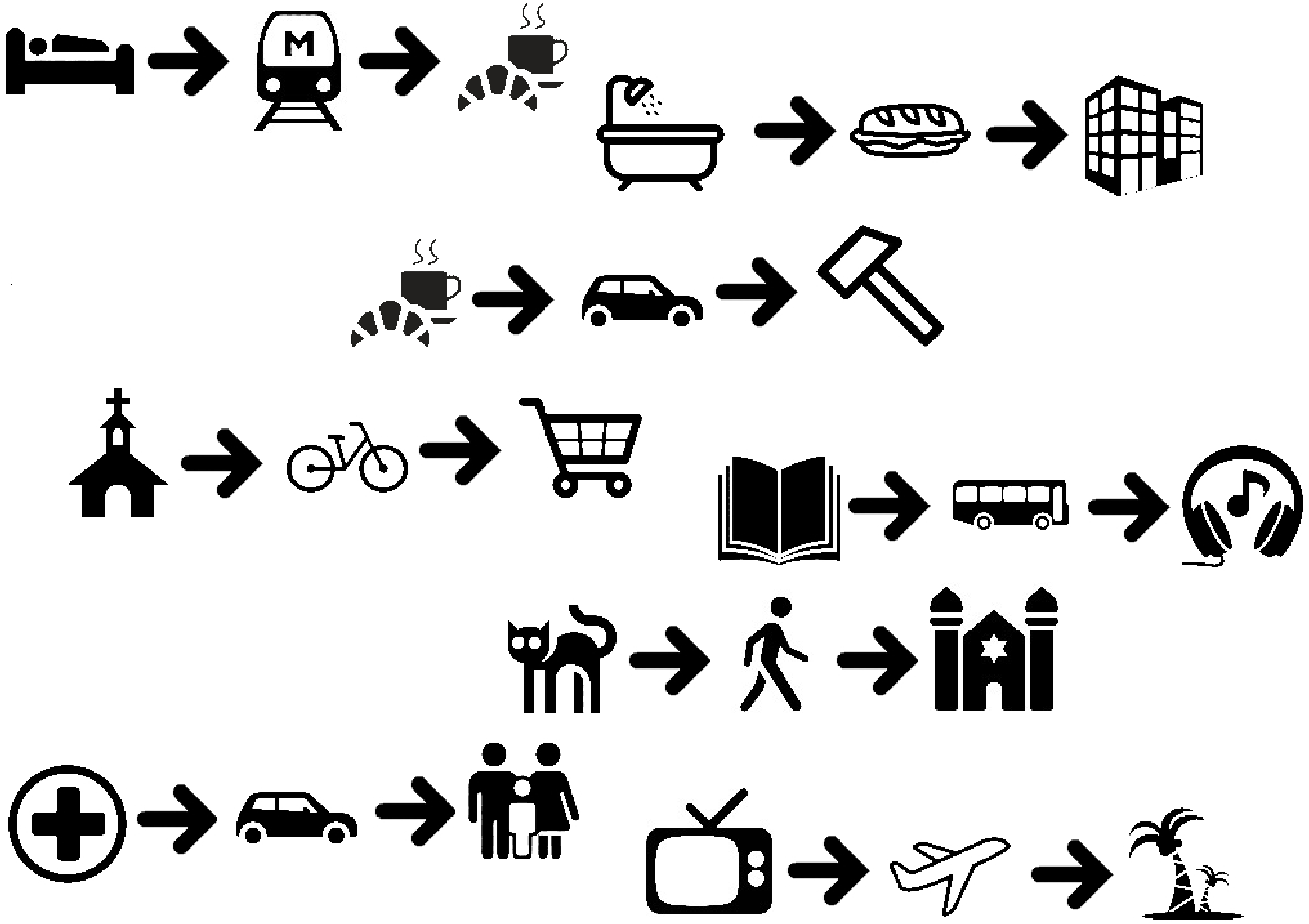
Alex Yarosh
Content Quality Analyst @ DataCamp

datacamp

Procedural programming

- Code as a sequence of steps
- Great for data analysis

Thinking in sequences



Procedural programming

- Code as a sequence of steps
- Great for data analysis and scripts

Object-oriented programming

- ***Code as interactions of objects***
- Great for building frameworks and tools
- *Maintainable and reusable code!*

Objects as data structures

$$\text{Object} = \text{state} + \text{behavior}$$

email = lara@company.com
phone = 614-555-0177

place order
cancel order

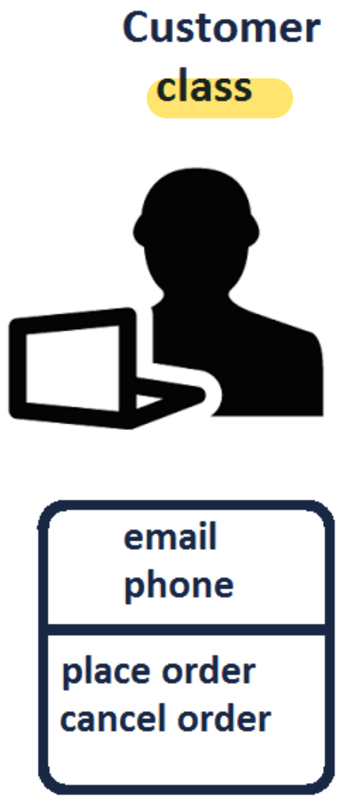
Encapsulation - bundling data with code operating on it

5. Objects as data structures

The fundamental concepts of OOP are objects and classes. An object is a data structure incorporating information about state and behavior. For example, an object representing a customer can have a certain phone number and email associated with them, and behaviors like placeOrder or cancelOrder. An object representing a button on a website can have a label, and can triggerEvent when pressed. The distinctive feature of OOP is that state and behavior are bundled together: instead of thinking of customer data separately from customer actions, we think of them as one unit representing a customer. This is called encapsulation, and it's one of the core tenets of object-oriented programming.

Classes as blueprints

- **Class** : blueprint for objects outlining possible states and behaviors



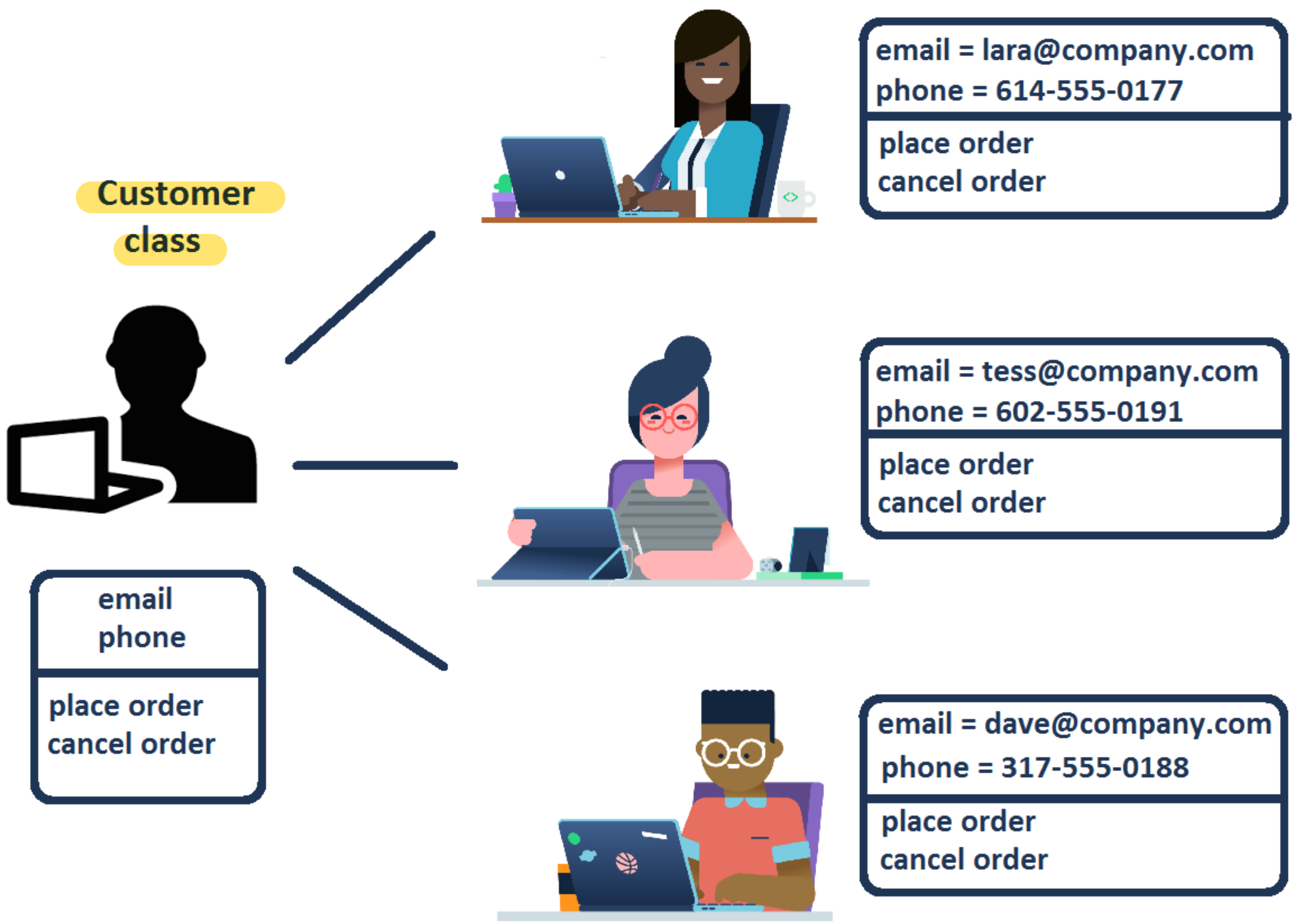
Customer
class

email
phone

place order
cancel order

Classes as blueprints

- **Class** : blueprint for objects outlining possible states and behaviors



Customer class

email
phone

place order
cancel order

email = lara@company.com
phone = 614-555-0177

place order
cancel order

email = tess@company.com
phone = 602-555-0191

place order
cancel order

email = dave@company.com
phone = 317-555-0188

place order
cancel order

6. Classes as blueprints
The real strength of OOP comes from utilizing classes. Classes are like blueprints for objects. They describe the possible states and behaviors that every object of a certain type could have. For example, if you say "every customer will have a phone number and an email, and will be able to place and cancel orders", you just defined a class! This way, you can talk about customers in a unified way.

7. Classes as blueprints
Then a specific Customer object is just a realization of this class with particular state values.

Objects in Python

- *Everything in Python is an object*
- Every object has a class
- Use `type()` to find the class

```python
import numpy as np
a = np.array([1,2,3,4])
print(type(a))
```

```
numpy.ndarray
```

Object	Class
`5`	`int`
`"Hello"`	`str`
`pd.DataFrame()`	`DataFrame`
`np.mean`	`function`
...	...

8. Objects in Python

In Python, everything is an object. Numbers, strings, DataFrames, even functions are objects. In particular, everything you deal with in Python has a class, a blueprint associated with it under the hood. The existence of these unified interfaces, is why you can use, for example, any DataFrame in the same way. You can call type() on any Python object to find out its class. For example, the class of a numpy array is actually called ndarray (for n-dimensional array).

Attributes and methods

State ↔ attributes

```python
import numpy as np
a = np.array([1,2,3,4])
# shape attribute
a.shape
```

```
(4,)
```

- Use `obj.` to access attributes and methods

Behavior ↔ methods

```python
import numpy as np
a = np.array([1,2,3,4])
# reshape method
a.reshape(2,2)
```

```
array([[1, 2],
       [3, 4]])
```

Object = attributes + methods

- attribute ↔ **variables** ↔ `obj.my_attribute`,
- method ↔ **function()** ↔ `obj.my_method()`.

Classes and objects both have attributes and methods, but the difference is that a class is an abstract template, while an object is a concrete representation of a class.

```
import numpy as np
a = np.array([1,2,3,4])
dir(a)                    # <--- list all attributes and methods
```

```
['T',
 '__abs__',
 ...
 'trace',
 'transpose',
 'var',
 'view']
```

10. Object = attributes + methods
Attributes (or states) in Python objects are represented by variables -- like numbers, or strings, or tuples, in the case of the numpy array shape. Methods, or behaviors, are represented by functions. Both are accessible from an object using the dot syntax. You can list all the attributes and methods that an object has by calling dir() on it. For example here, we see that a numpy array has methods like trace and transpose.

Let's review!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Class anatomy: attributes and methods

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh

Content Quality Analyst @ DataCamp

A basic class

```python
class Customer:
    # code for class goes here
    pass
```

- `class <name>:` starts a class definition
- code inside `class` is indented
- use `pass` to create an "empty" class

```python
c1 = Customer()
c2 = Customer()
```

- use `ClassName()` to create an object of class `ClassName`

Add methods to a class

```python
class Customer:

    def identify(self, name):
      print("I am Customer " + name)
```

- method definition = function definition within class
- **use `self` as the 1st argument in method definition**

```python
cust = Customer()
cust.identify("Laura")
```

```
I am Customer Laura
```

- ignore `self` when calling method on an object

3. Add methods to a class

Defining a method is is simple. Methods are functions, so the definition of a method looks just like a regular Python function, with one exception: the special self argument that every method will have as the first argument, possibly followed by other arguments. We'll get back to self in a minute, first let's see how this works. Here we defined a method "identify" for the Customer class that takes self and a name as a parameter and prints "I am Customer" plus name when called. We create a new customer object, call the method by using object-dot-method syntax and pass the desired name, and get the output. Note that name was the second parameter in the method definition, but it is the first parameter when the method is called. The mysterious self is not needed the method call.

```
class Customer:

    def identify(self, name):
      print("I am Customer " + name)

cust = Customer()
cust.identify("Laura")
```

4. What is self?
So what was that self? Classes are templates. Objects of a class don't yet exist when a class is being defined, but we often need a way to refer to the data of a particular object within class definition. That is the purpose of self - it's a stand-in for the future object. That's why every method should have the self argument -- so we could use it to access attributes and call other methods from within the class definition even when no objects were created yet. Python will handle self when the method is called from an object using the dot syntax. In fact, using object-dot-method is equivalent to passing that object as an argument. That's why we don't specify it explicitly when calling the method from an existing object.

What is self?

- classes are templates, how to refer data of a particular object?
- `self` is a stand-in for a particular object used in class definition
- should be the first argument of any method
- Python will take care of `self` when method called from an object:

`cust.identify("Laura")` *will be interpreted as* `Customer.identify(cust, "Laura")`

We need attributes

- **Encapsulation:** bundling data with methods that operate on data
- E.g. `Customer` 's' name should be an attribute

Attributes are created by assignment (=) in methods

Add an attribute to class

```python
class Customer:
    # set the name attribute of an object to new_name
    def set_name(self, new_name):
      # Create an attribute by assigning a value
      self.name = new_name          # <-- will create .name when set_name is called
```

```python
cust = Customer()                   # <--.name doesn't exist here yet
cust.set_name("Lara de Silva")      # <--.name is created and set to "Lara de Silva"
print(cust.name)                    # <--.name can be used
```

```
Lara de Silva
```

6. Add an attribute to class

Here is a method set_name with arguments self (every method should have a self argument) and new_name. To create an attribute of the Customer class called "name", all we need to do is to assign something to self-dot-name. Remember, self is a stand-in for object, so self-dot-attribute should remind you of the object-dot-attribute syntax. Here, we set the name attribute to the new_name parameter of the function. When we create a customer, it does not yet have a name attribute. But after the set_name method was called, the name attribute is created, and we can access it through dot-name.

Old version

```python
class Customer:



    # Using a parameter
    def identify(self, name):
      print("I am Customer" + name)
```

```python
cust = Customer()

cust.identify("Eris Odoro")
```

```
I am Customer Eris Odoro
```

New version

```python
class Customer:
   def set_name(self, new_name):
      self.name = new_name

    # Using .name from the object it*self*
    def identify(self):
      print("I am Customer" + self.name)
```

```python
cust = Customer()
cust.set_name("Rashid Volkov")
cust.identify()
```

```
I am Customer Rashid Volkov
```

Instead of passing name as a parameter, we will use the data already stored in the name attribute of the customer class. We remove the name parameter from the identify method, and replace it with self-dot-name in the printout, which, via self, will pull the name attribute from the object that called the method. Now the identify function will only use the data that is encapsulated in the object, instead of using whatever we passed to it.

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Class anatomy: the __init__ constructor

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh
Content Quality Analyst @ DataCamp

Methods and attributes

- Methods are function definitions within a class
- `self` as the first argument
- Define attributes by assignment
- Refer to attributes in class via `self.___`

```
class MyClass:
    # function definition in class
    # first argument is self
    def my_method1(self, other_args...):
        # do things here

    def my_method2(self, my_attr):
        # attribute created by assignment
        self.my_attr = my_attr
        ...
```

2. Methods and attributes

You learned that methods are functions within class with a special first argument self, and that attributes are created by assignment and referred to using the self variable within methods. In the exercises, you created an Employee class, and for each attribute you wanted to create, you defined a new method, and then called those methods one after another. **This could quickly get unsustainable if your classes contain a lot of data.**

Constructor

- Add data to object when creating it?
- **Constructor** `__init__()` method is called every time an object is created.

```python
class Customer:
       def __init__(self, name):
        self.name = name           # <--- Create the .name attribute and set it to name parameter
        print("The __init__ method was called")

cust = Customer("Lara de Silva")   #<--- __init__ is implicitly called
print(cust.name)
```

```
The __init__ method was called
Lara de Silva
```

3. Constructor
A better strategy would be to add data to the object when creating it, like you do when creating a numpy array or a DataFrame. Python allows you to add a special method called the constructor that is automatically called every time an object is created. The method takes on one argument, name, of course in addition to the self argument that should be there for any method. In the body of the method, we create the name attribute, set its value to the name parameter, and print a message. So now, we can pass the customer name in the parentheses when creating the customer object, and the init method will be automatically called, and the name attribute created.

```python
class Customer:
       def __init__(self, name, balance):  # <-- balance parameter added
        self.name = name
        self.balance = balance             # <-- balance attribute added
        print("The __init__ method was called")
cust = Customer("Lara de Silva", 1000)     # <-- __init__ is called
print(cust.name)
print(cust.balance)
```

```
The __init__ method was called
Lara de Silva
1000
```

```python
class Customer:
        def __init__(self, name, balance=0):  #<--set default value for balance
         self.name = name
         self.balance = balance
         print("The __init__ method was called")

cust = Customer("Lara de Silva") # <-- don't specify balance explicitly
print(cust.name)
print(cust.balance) # <-- attribute is created anyway
```

```
The __init__ method was called
Lara de Silva
0
```

3. Constructor

A better strategy would be to add data to the object when creating it, like you do when creating a numpy array or a DataFrame. Python allows you to add a special method called the constructor that is automatically called every time an object is created. The method has to be called underscore underscore init underscore underscore (the exact name and double underscores are essential for Python to recognize it). Here we define the init method for the customer class. The method takes on one argument, name, of course in addition to the self argument that should be there for any method. In the body of the method, we create the name attribute, set its value to the name parameter, and print a message. So now, we can pass the customer name in the parentheses when creating the customer object, and the init method will be automatically called, and the name attribute created.

Attributes in methods

```
class MyClass:
    def my_method1(self, attr1):
        self.attr1 = attr1
        ...

    def my_method2(self, attr2):
        self.attr2 = attr2
        ...
```

```
obj = MyClass()
obj.my_method1(val1) # <-- attr1 created
obj.my_method2(val2) # <-- attr2 created
```

Attributes in the constructor

```
class MyClass:
    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2
        ...
# All attributes are created
obj = MyClass(val1, val2)
```

- easier to know all the attributes
- attributes are created when the object is created
- *more usable and maintainable code*

Best practices

1. Initialize attributes in `__init__()`

8. Best practices
To name your classes, use camel case, which means that if your class name contains several words, they should be written without delimiters, and each word should start with a capital letter. For methods and attributes, it's the opposite -- words should be separated by underscores and start with lowercase letters.

Best practices

1. Initialize attributes in `__init__()`

2. Naming

`CamelCase` for classes, `lower_snake_case` for functions and attributes

Best practices

1. Initialize attributes in `__init__()`

2. Naming

`CamelCase` for class, `lower_snake_case` for functions and attributes

3. Keep `self` as `self`

```python
class MyClass:
    # This works but isn't recommended
    def my_method(kitty, attr):
        kitty.attr = attr
```

9. Best practices
Here's a secret: the name "self" is a convention. You could actually use any name for the first variable of a method, it will always be treated as the object reference regardless. For example, if you are a Java programmer, you might be tempted to use "this", and if you are me, you might be tempted to use "kitty". Don't do it, and always use "self".

10. Best practices
Finally, classes, like functions, allow for docstrings which are displayed when help() is called on the object.

Best practices

1. Initialize attributes in `__init__()`

2. Naming

`CamelCase` for class, `lower_snake_case` for functions and attributes

3. `self` is `self`

4. Use docstrings

```
class MyClass:
    """This class does nothing"""
    pass
```

8. Best practices
To name your classes, use camel case, which means that if your class name contains several words, they should be written without delimiters, and each word should start with a capital letter. For methods and attributes, it's the opposite -- words should be separated by underscores and start with lowercase letters.

```
# Import datetime from datetime
from datetime import datetime
class Employee:

    def __init__(self, name, hire_date, salary=0):
        self.name = name
        if salary > 0:
          self.salary = salary
        else:
          self.salary = 0
          print("Invalid salary!")

        # Add the hire_date attribute and set it to today's date
        self.hire_date = datetime.today()

   # ...Other methods omitted for brevity ...

emp = Employee("Korel Rossi", -1000)
print(emp.name)
print(emp.salary)
```

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON