

What is OOP?

OBJECT-ORIENTED PROGRAMMING IN PYTHON



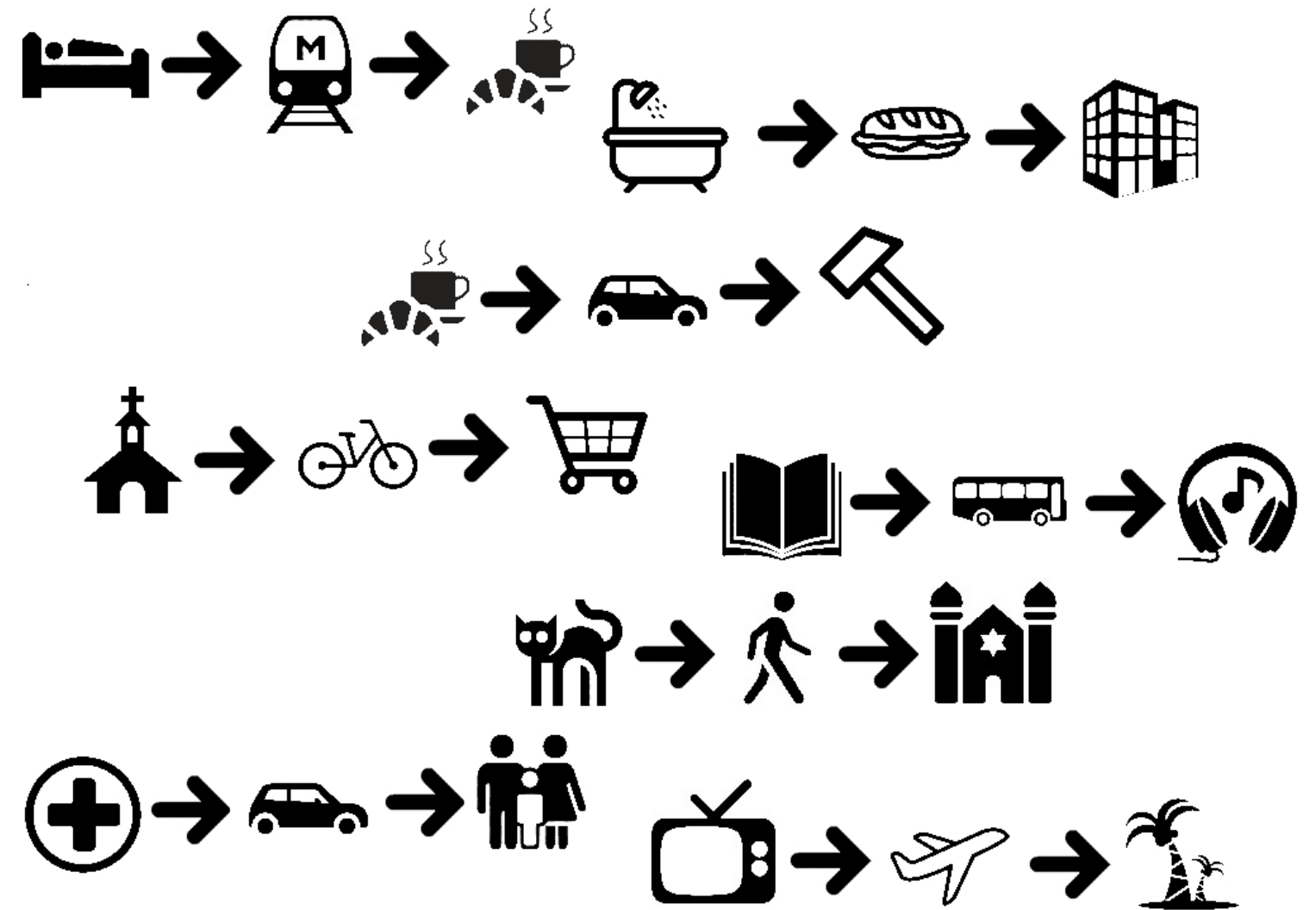
Alex Yarosh

Content Quality Analyst @ 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

Object = state + behavior



Encapsulation - bundling data with code operating on it

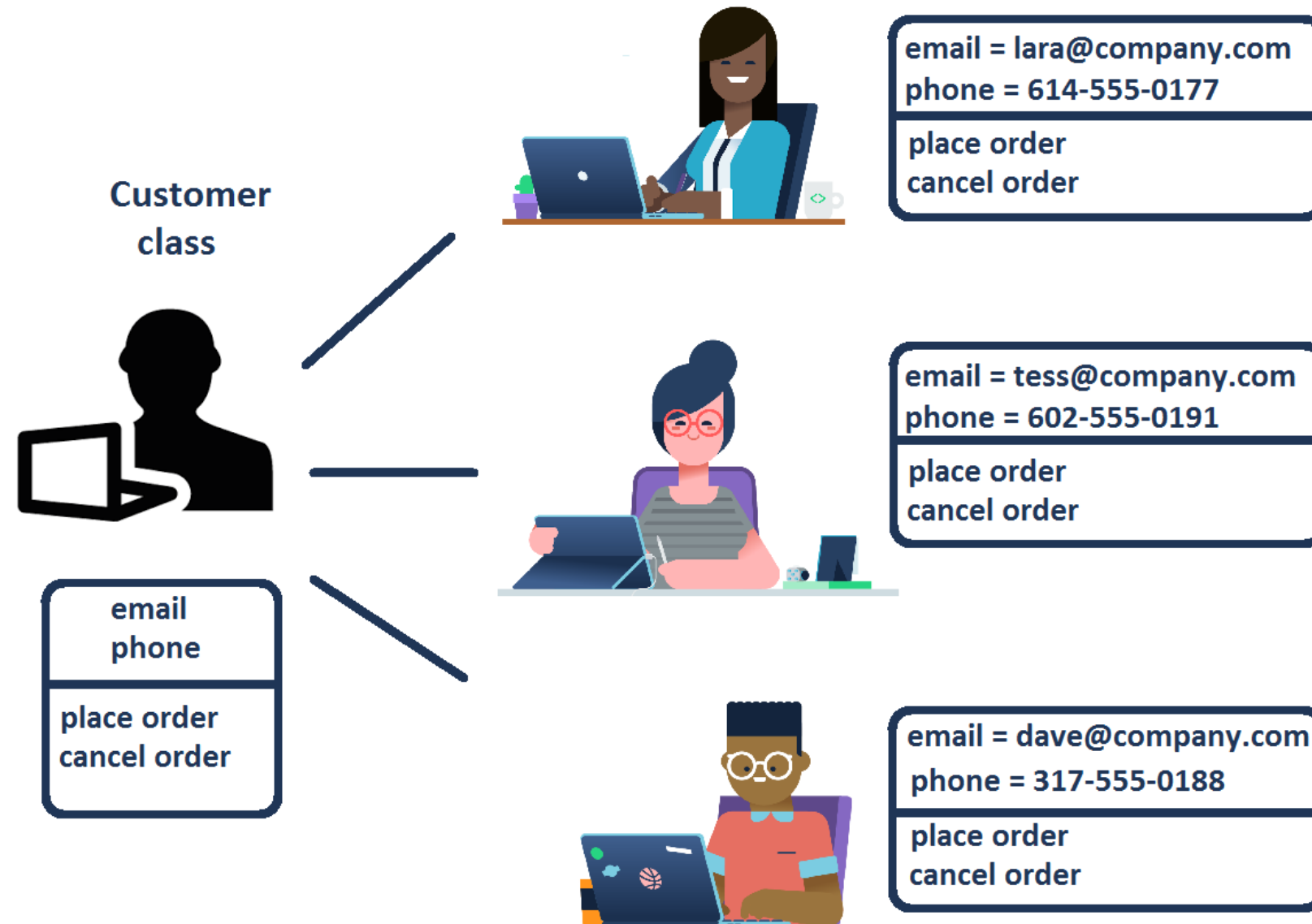
Classes as blueprints

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



Classes as blueprints

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



Objects in Python

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

```
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
...	...

Attributes and methods

State ↔ attributes

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

# shape attribute
a.shape
```

```
(4,)
```

Behavior ↔ methods

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

# reshape method
a.reshape(2,2)
```

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

- Use `obj.` to access attributes and methods

Object = attributes + methods

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

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

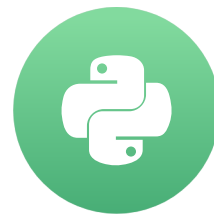
```
['T',
 '__abs__',
 ...,
 'trace',
 'transpose',
 'var',
 'view']
```

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

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

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

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

Add methods to a class

```
class Customer:

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

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

```
I am Customer Laura
```

- method definition = function definition within class
- use `self` as the 1st argument in method definition
- ignore `self` when calling method on an object

```
class Customer:

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

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

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

```
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
```

```
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
```

Old version

```
class Customer:

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

```
cust = Customer()

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

```
I am Customer Eris Odoro
```

New version

```
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)
```

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

```
I am Customer Rashid Volkov
```

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
        ...
```

Constructor

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

```
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
```

```
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
```

```
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
```


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__()`

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`

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

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
```

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON