

What is OOP?

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

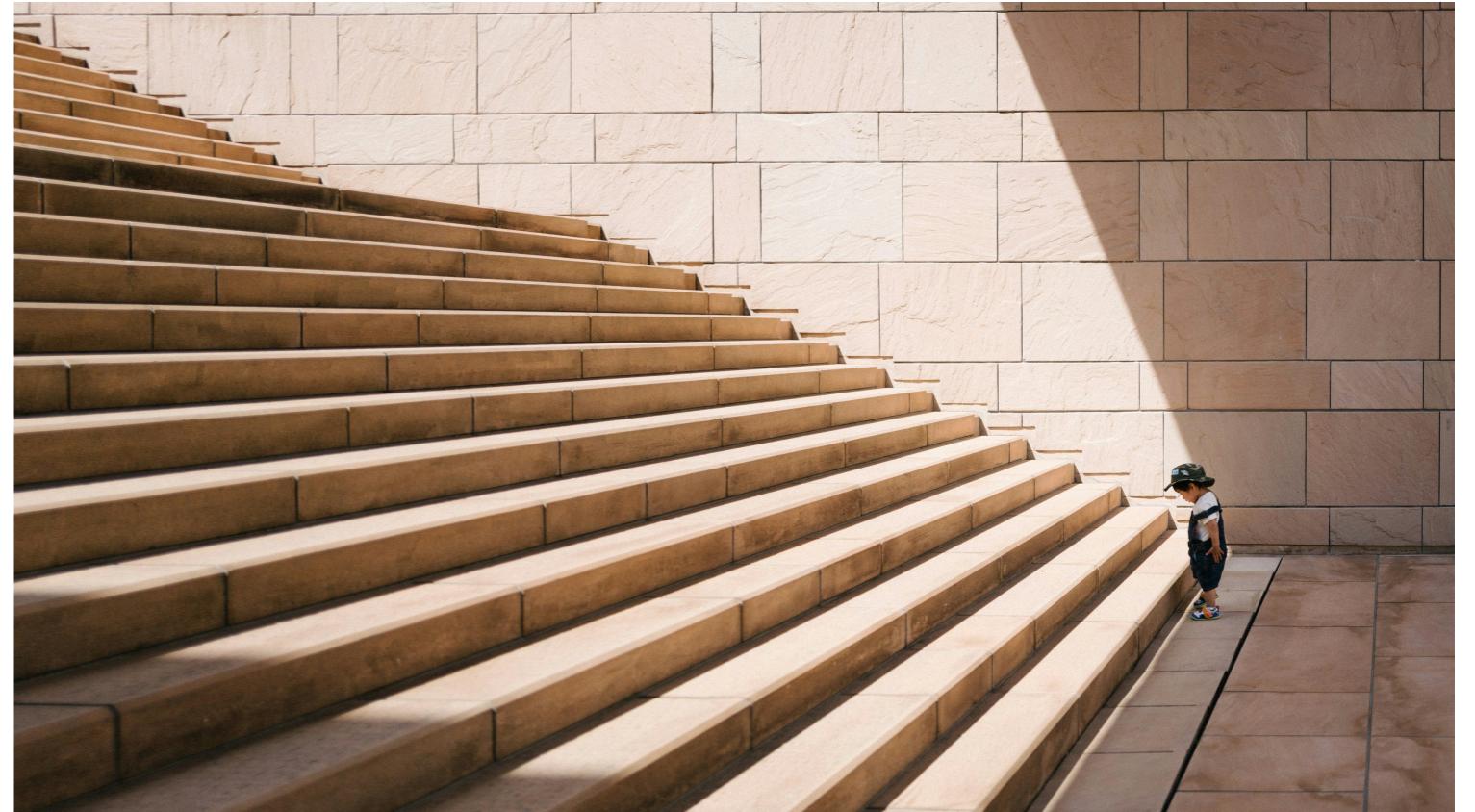


George Boorman

Curriculum Manager, DataCamp

Procedural programming

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



¹ Image source: <https://unsplash.com/@tateisimikito>

Thinking in sequences



Procedural programming

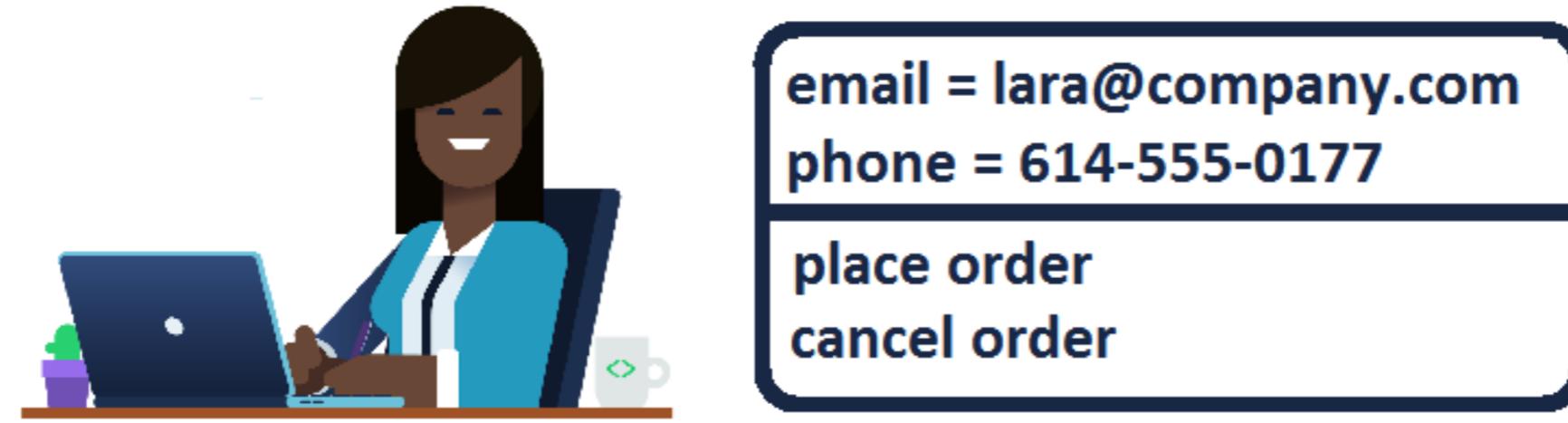
- Code as a sequence of steps
- Great for data analysis

Object-oriented programming

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

Objects

Object = data + functionality



State - an object's *data*

Behavior - an object's *functionality*

Objects in Python

- *Everything in Python is an object*

Object	Type
5	int
"Hello"	str
pd.DataFrame()	DataFrame
sum()	function
...	...

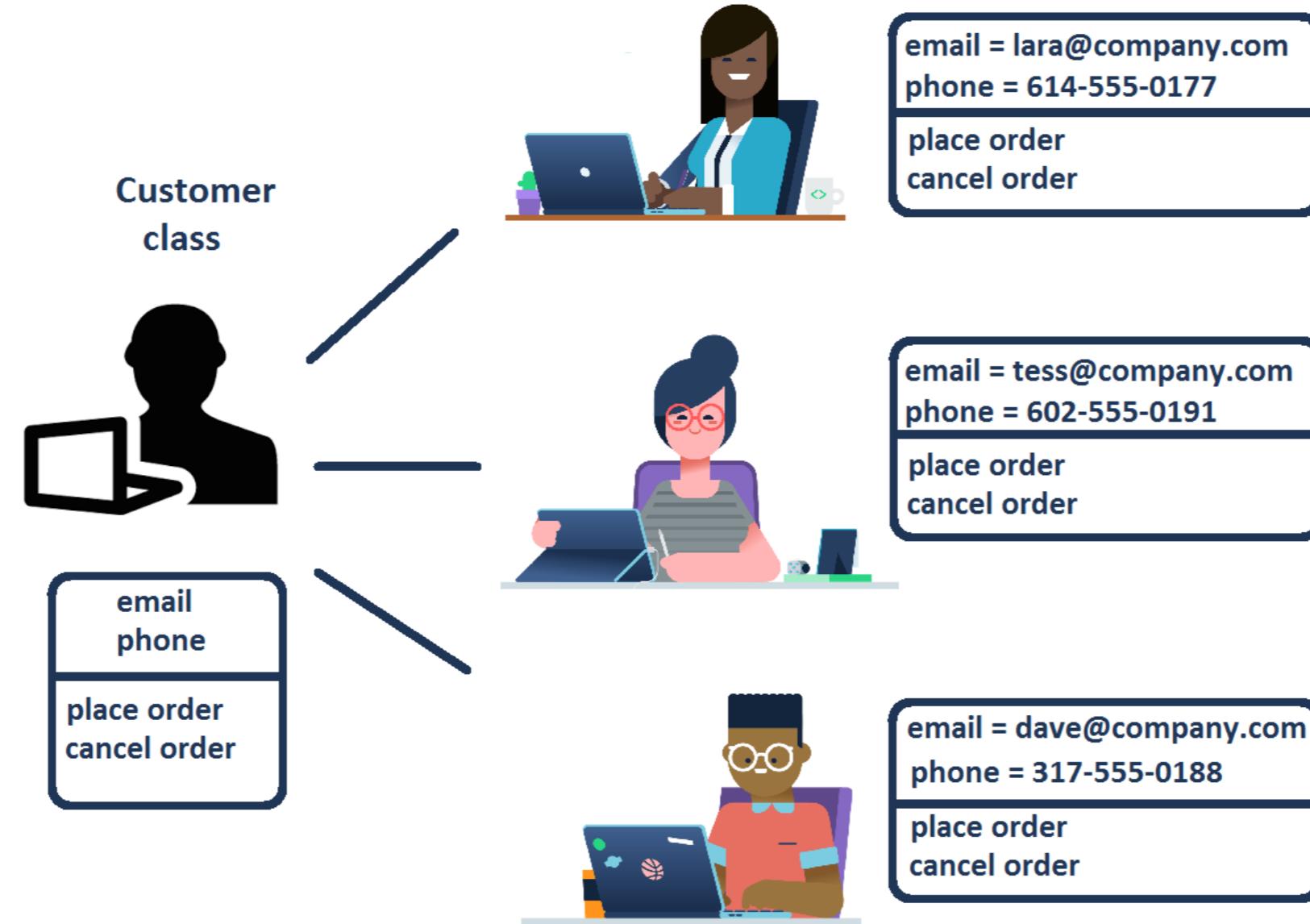
Classes as blueprints

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



Classes as blueprints

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



Classes in Python

- Python objects of the same type behave in the same way
- `lists` are a class
 - Created with comma-separated values `[1, 2, 3, 4, 5]`
 - Share the same methods, e.g., `.append()`
- Use `type()` to find the class

```
type([1, 2, 3, 4, 5])
```

```
<class 'list'>
```

Attributes and methods

State ↔ attributes

```
import pandas as pd  
df = pd.DataFrame({"a": [1,2,3],  
                    "b": [4,5,6]})  
  
# shape attribute  
df.shape
```

(3, 2)

- Use `obj.` to access attributes and methods

Behavior ↔ methods

```
import pandas as pd  
df = pd.DataFrame({"a": [1,2,3],  
                    "b": [4,5,6]})  
  
# head method  
df.head()
```

	a	b
0	1	4
1	2	5
2	3	6

Displaying attributes and methods

```
# Display attributes and methods  
dir([1, 2, 3, 4])
```

```
['__add__',  
'__class__',  
'__contains__',  
'__delattr__',  
...  
'pop',  
'remove',  
'reverse',  
'sort']
```

```
# Display attributes and methods  
dir(list)
```

```
['__add__',  
'__class__',  
'__contains__',  
'__delattr__',  
...  
'pop',  
'remove',  
'reverse',  
'sort']
```

Cheat sheet

Term	Definition
Class	A blueprint/template used to build objects
Object	A combination of <i>data</i> and <i>functionality</i> ; An instance of a class
State	<i>Data</i> associated with an object, assigned through attributes
Behavior	An object's <i>functionality</i> , defined through methods

Let's review!

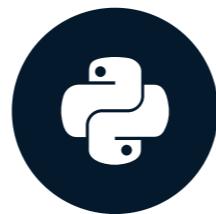
INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Class anatomy: attributes and methods

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

George Boorman

Curriculum Manager, DataCamp



A Customer class

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

```
c_one = Customer()  
c_two = 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 first argument in method definition
- Ignore `self` when calling a 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
- `self` should be the first argument of any method
- `self` is a stand-in for a (not yet created) object
- `cust.identify("Laura")` *will be interpreted as* `Customer.identify(cust, "Laura")`

We need attributes

- OOP bundles data with methods that operate on data
 - 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  
        # Will create .name when set_name is called  
        self.name = new_name  
  
# Create an object  
# .name doesn't exist here yet  
cust = Customer()  
# .name is created and set to "Lara de Silva"  
cust.set_name("Lara de Silva")  
print(cust.name)
```

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

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

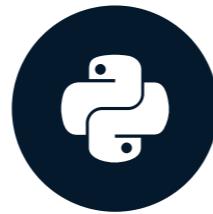
I am Customer Rashid Volkov

Let's practice!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Class anatomy: the `__init__` constructor

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON



George Boorman

Curriculum Manager, 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.____`
- Calling lots of methods could become unsustainable!

```
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
 - Called automatically because of `__methodname__` syntax

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

Constructor

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

The `__init__` method was called
Lara de Silva

Attributes in methods

```
class MyClass:  
    def my_method1(self, attr1):  
        self.attr1 = attr1  
        ...  
  
    def my_method2(self, attr2):  
        self.attr2 = attr2  
        ...  
  
obj = MyClass()  
# attr1 created  
obj.my_method1(val1)  
# attr2 created  
obj.my_method2(val2)
```

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)
```

- Generally we should use the constructor
- Attributes are created when the object is created
- *More usable and maintainable code*

Add arguments

```
class Customer:  
    # Add balance argument  
    def __init__(self, name, balance):  
        self.name = name  
  
        # Add the balance attribute  
        self.balance = balance  
        print("The __init__ method was called")
```

Add parameters

```
# __init__ is called  
cust = Customer("Lara de Silva", 1000)  
print(cust.name)  
print(cust.balance)
```

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

Default arguments

```
class Customer:  
    # Set a default value for balance  
    def __init__(self, name, balance=0):  
        self.name = name  
        # Assign the new attribute  
        self.balance = balance  
        print("The __init__ method was called")
```

Default arguments

```
# Don't specify the balance explicitly
cust = Customer("Lara de Silva")
print(cust.name)

# The balance attribute is created anyway
print(cust.balance)
```

The `__init__` method was called

Lara de Silva

0

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(george, attr):  
        george.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!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON