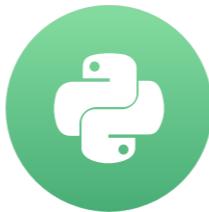


What is OOP?

OBJECT-ORIENTED PROGRAMMING IN PYTHON



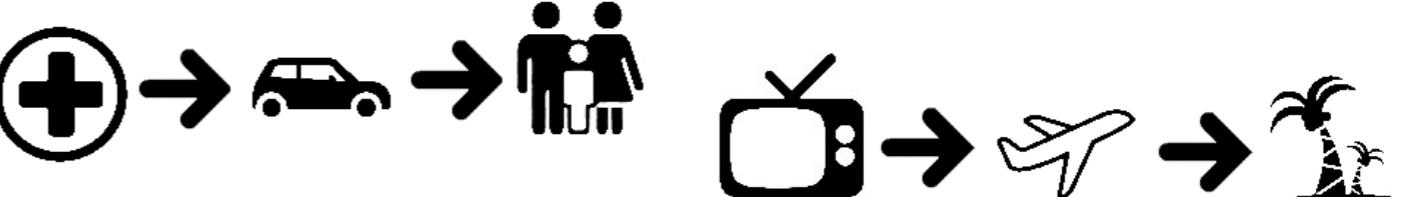
Alex Yarosh

Content Quality Analyst @ DataCamp

Procedural programming

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

Thinking in sequences



Procedural programming

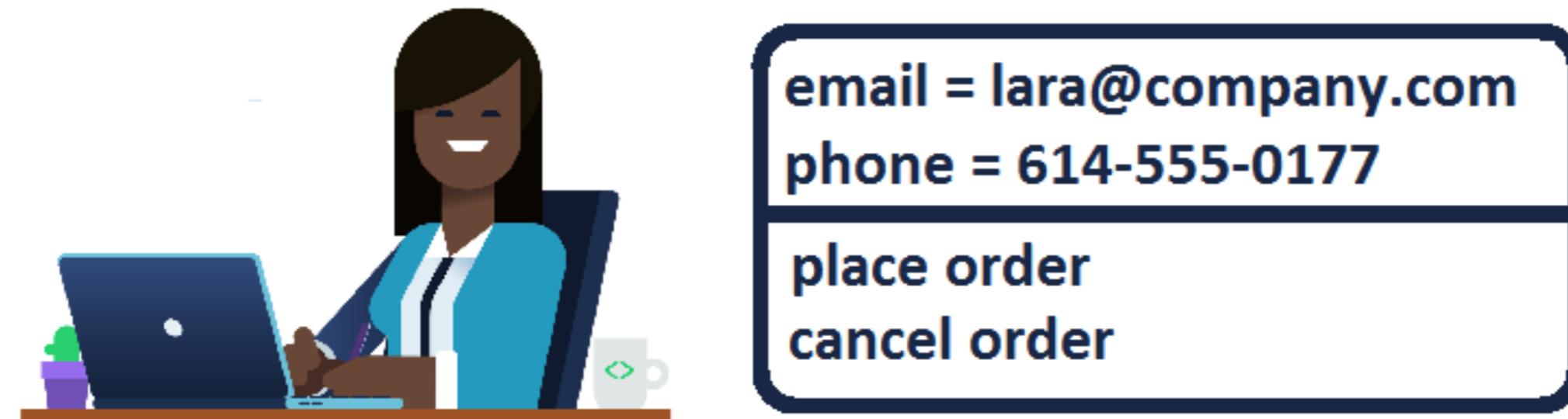
- Code as a sequence of steps
- Great for data analysis and scripts

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

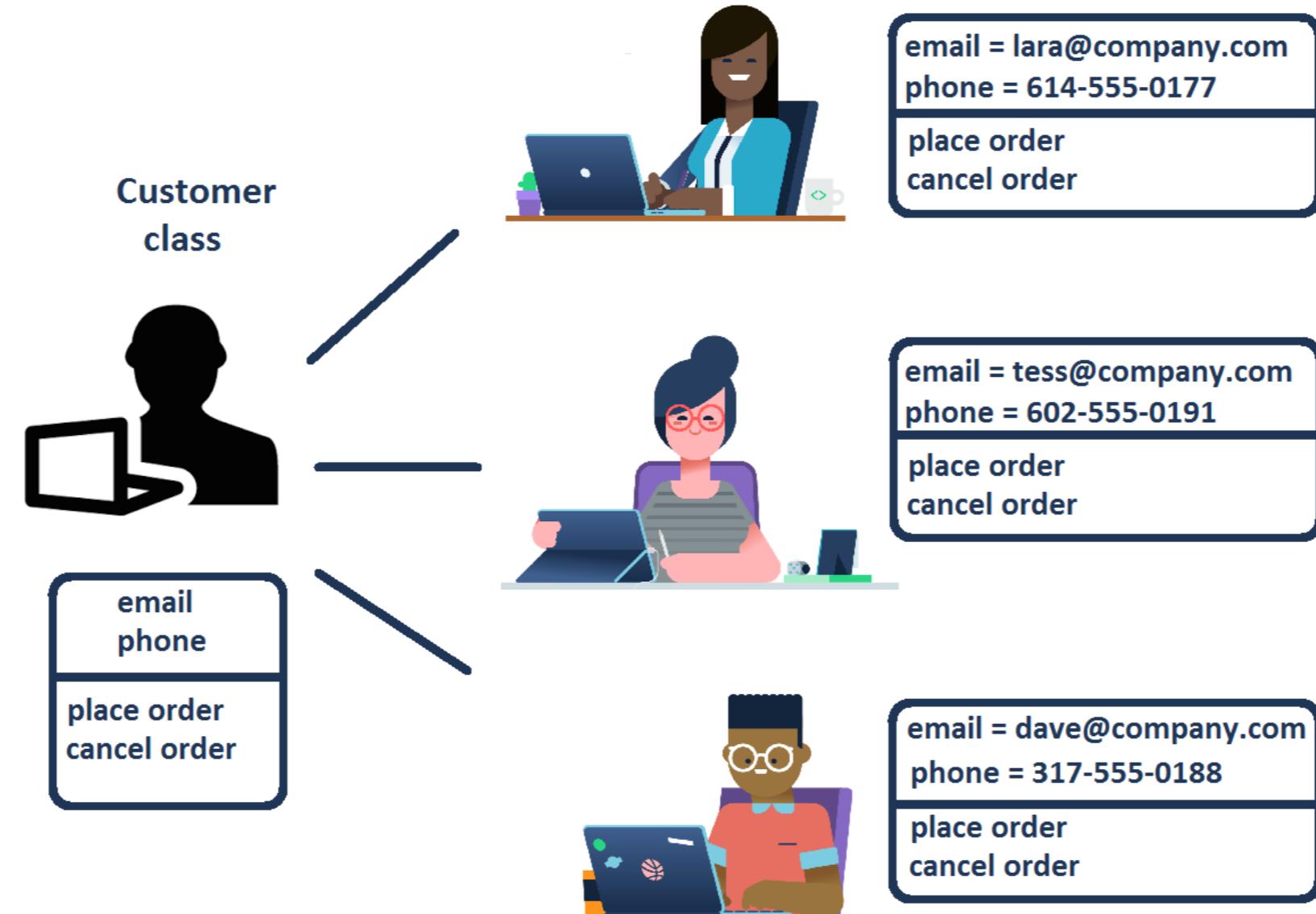
Classes as blueprints

- 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 \leftrightarrow variables \leftrightarrow `obj.my_attribute` ,
- method \leftrightarrow function() \leftrightarrow `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 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!

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

Instance and class data

OBJECT-ORIENTED PROGRAMMING IN PYTHON



Alex Yarosh

Content Quality Analyst @ DataCamp

Core principles of OOP

Inheritance:

- Extending functionality of existing code

Polymorphism:

- Creating a unified interface

Encapsulation:

- Bundling of data and methods

Instance-level data

```
class Employee:  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
  
emp1 = Employee("Teo Mille", 50000)  
emp2 = Employee("Marta Popov", 65000)
```

- `name` , `salary` are *instance attributes*
- `self` binds to an instance

Class-level data

- Data shared among all instances of a class
- Define *class attributes* in the body of `class`

```
class MyClass:  
    # Define a class attribute  
    CLASS_ATTR_NAME = attr_value
```

- "Global variable" within the class

Class-level data

```
class Employee:  
    # Define a class attribute  
    MIN_SALARY = 30000    #<--- no self.  
  
    def __init__(self, name, salary):  
        self.name = name  
        # Use class name to access class attribute  
        if salary >= Employee.MIN_SALARY:  
            self.salary = salary  
        else:  
            self.salary = Employee.MIN_SALARY
```

- `MIN_SALARY` is shared among all instances
- Don't use `self` to *define* class attribute
- use `ClassName.ATTR_NAME` to *access* the class attribute value

Class-level data

```
class Employee:  
    # Define a class attribute  
    MIN_SALARY = 30000  
  
    def __init__(self, name, salary):  
        self.name = name  
        # Use class name to access class attribute  
        if salary >= Employee.MIN_SALARY:  
            self.salary = salary  
        else:  
            self.salary = Employee.MIN_SALARY
```

```
emp1 = Employee("TBD", 40000)  
print(emp1.MIN_SALARY)
```

30000

```
emp2 = Employee("TBD", 60000)  
print(emp2.MIN_SALARY)
```

30000

Why use class attributes?

Global constants related to the class

- minimal/maximal values for attributes
- commonly used values and constants, e.g. `pi` for a `Circle` class
- ...

Class methods

- Methods are already "shared": same code for every instance
- Class methods can't use instance-level data

```
class MyClass:  
  
    @classmethod # <---use decorator to declare a class method  
    def my_awesome_method(cls, args...): # <---cls argument refers to the class  
        # Do stuff here  
        # Can't use any instance attributes :(
```

```
MyClass.my_awesome_method(args...)
```

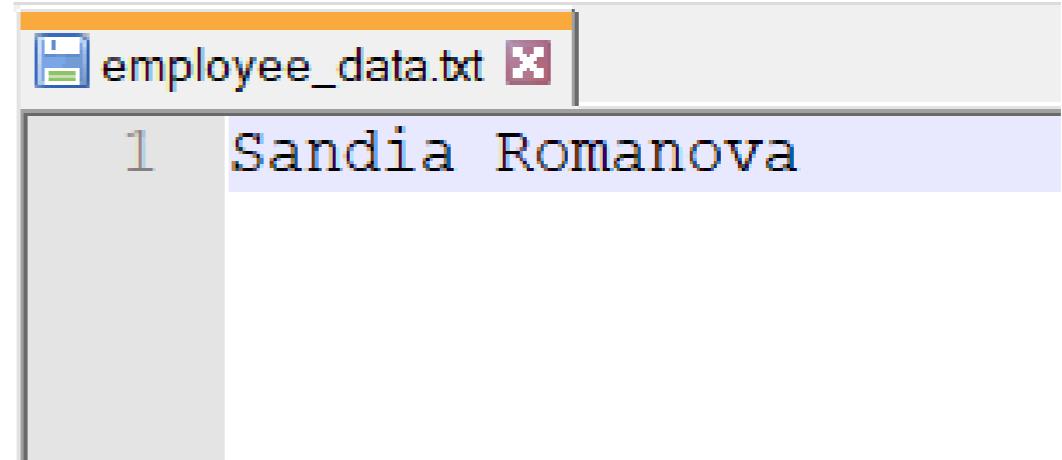
Alternative constructors

```
class Employee:  
    MIN_SALARY = 30000  
  
    def __init__(self, name, salary=30000):  
        self.name = name  
        if salary >= Employee.MIN_SALARY:  
            self.salary = salary  
        else:  
            self.salary = Employee.MIN_SALARY  
  
    @classmethod  
    def from_file(cls, filename):  
        with open(filename, "r") as f:  
            name = f.readline()  
        return cls(name)
```

- Can only have one `__init__()`
- Use class methods to create objects
 - Use `return` to return an object
 - `cls(...)` will call `__init__(...)`

Alternative constructors

```
class Employee:  
    MIN_SALARY = 30000  
  
    def __init__(self, name, salary=30000):  
        self.name = name  
        if salary >= Employee.MIN_SALARY:  
            self.salary = salary  
        else:  
            self.salary = Employee.MIN_SALARY  
  
    @classmethod  
    def from_file(cls, filename):  
        with open(filename, "r") as f:  
            name = f.readline()  
        return cls(name)
```



```
# Create an employee without calling Employee()  
emp = Employee.from_file("employee_data.txt")  
type(emp)
```

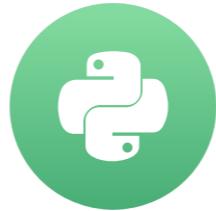
```
__main__.Employee
```

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Class inheritance

OBJECT-ORIENTED PROGRAMMING IN PYTHON



Alex Yarosh

Content Quality Analyst @ DataCamp

Code reuse

Code reuse

1. Someone has already done it

- Modules are great for fixed functionality
- OOP is great for customizing functionality



Code reuse

1. Someone has already done it

SUBMIT

One

Two

Three

2. DRY: Don't Repeat Yourself



Input text

Helper text



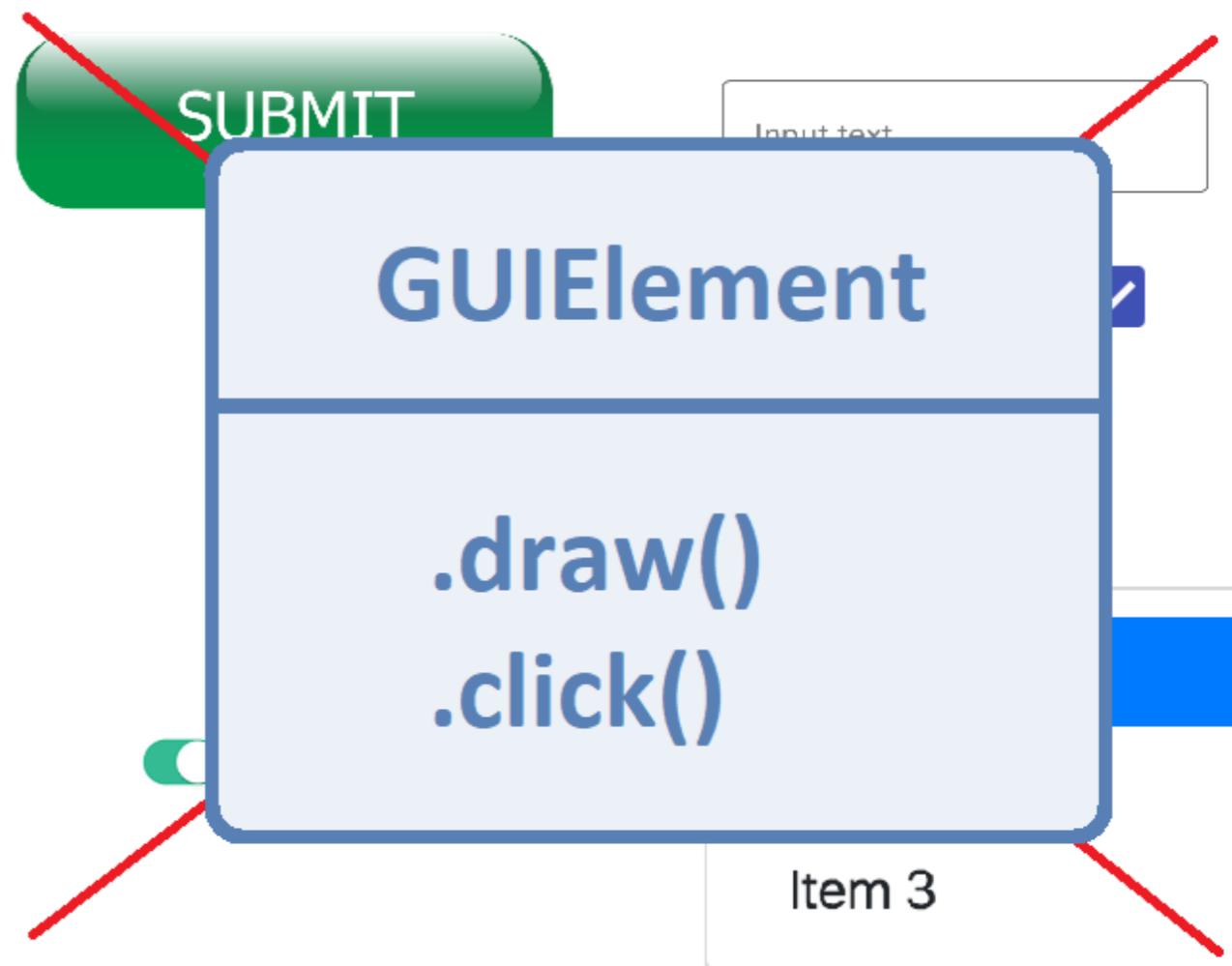
Dropdown ▾

- Item 1
- Item 2
- Item 3

Code reuse

1. Someone has already done it

2. DRY: Don't Repeat Yourself



Inheritance

New class functionality = Old class functionality + extra

BankAccount



`balance`

`withdraw()`

BankAccount



balance

withdraw()

SavingsAccount

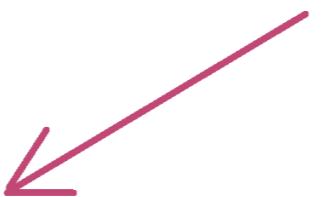


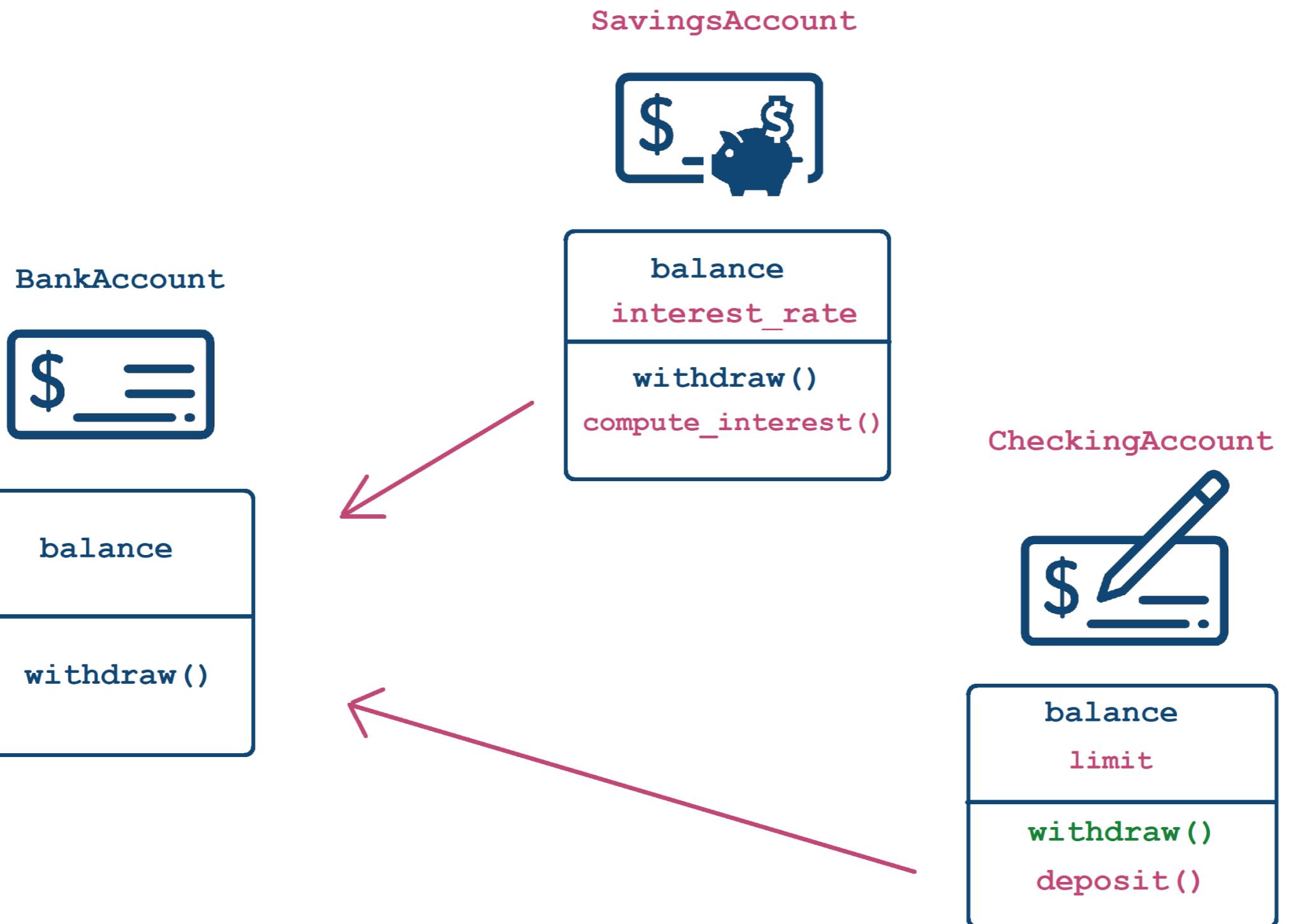
balance

interest_rate

withdraw()

compute_interest()





BankAccount



`balance`

`withdraw()`

SavingsAccount



`balance`
`interest_rate`
`withdraw()`
`compute_interest()`

**Modified version of
withdraw()**

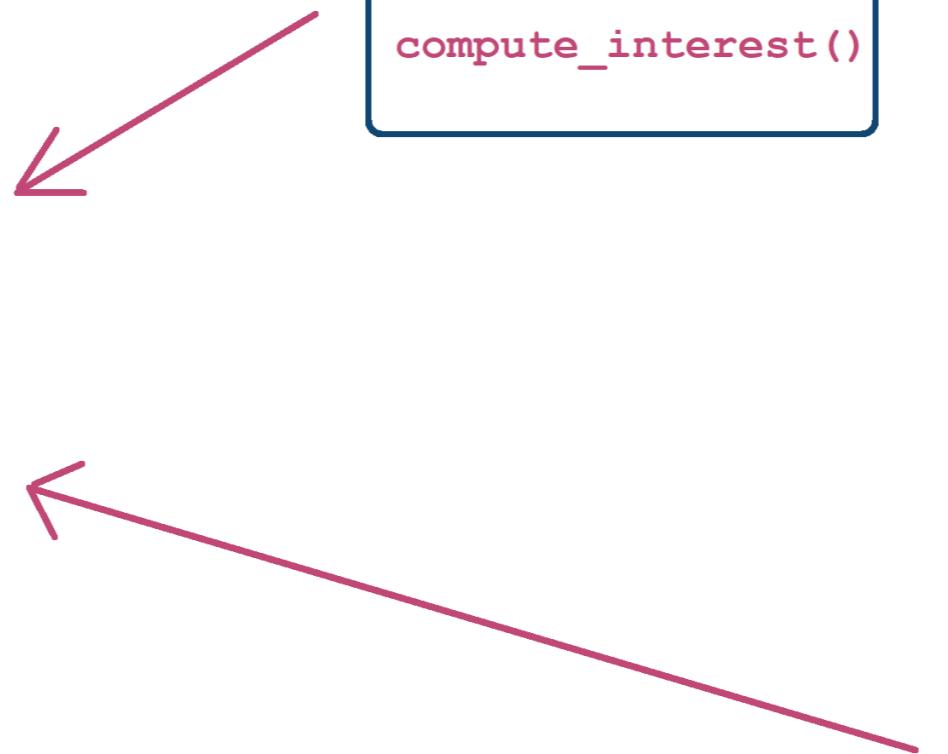
CheckingAccount



`balance`

`limit`

`withdraw()`
`deposit()`



Implementing class inheritance

```
class BankAccount:  
    def __init__(self, balance):  
        self.balance = balance  
  
    def withdraw(self, amount):  
        self.balance -= amount  
  
# Empty class inherited from BankAccount  
class SavingsAccount(BankAccount):  
    pass
```

```
class MyChild(MyParent):  
    # Do stuff here
```

- `MyParent` : class whose functionality is being extended/inherited
- `MyChild` : class that will inherit the functionality and add more

Child class has all of the the parent data

```
# Constructor inherited from BankAccount  
savings_acct = SavingsAccount(1000)  
type(savings_acct)
```

```
--main__.SavingsAccount
```

```
# Attribute inherited from BankAccount  
savings_acct.balance
```

```
1000
```

```
# Method inherited from BankAccount  
savings_acct.withdraw(300)
```

Inheritance: "is-a" relationship

A *SavingsAccount* is a *BankAccount*

(possibly with special features)

```
savings_acct = SavingsAccount(1000)  
isinstance(savings_acct, SavingsAccount)
```

True

```
acct = BankAccount(500)  
isinstance(acct, SavingsAccount)
```

False

```
isinstance(savings_acct, BankAccount)
```

True

```
isinstance(acct, BankAccount)
```

True

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

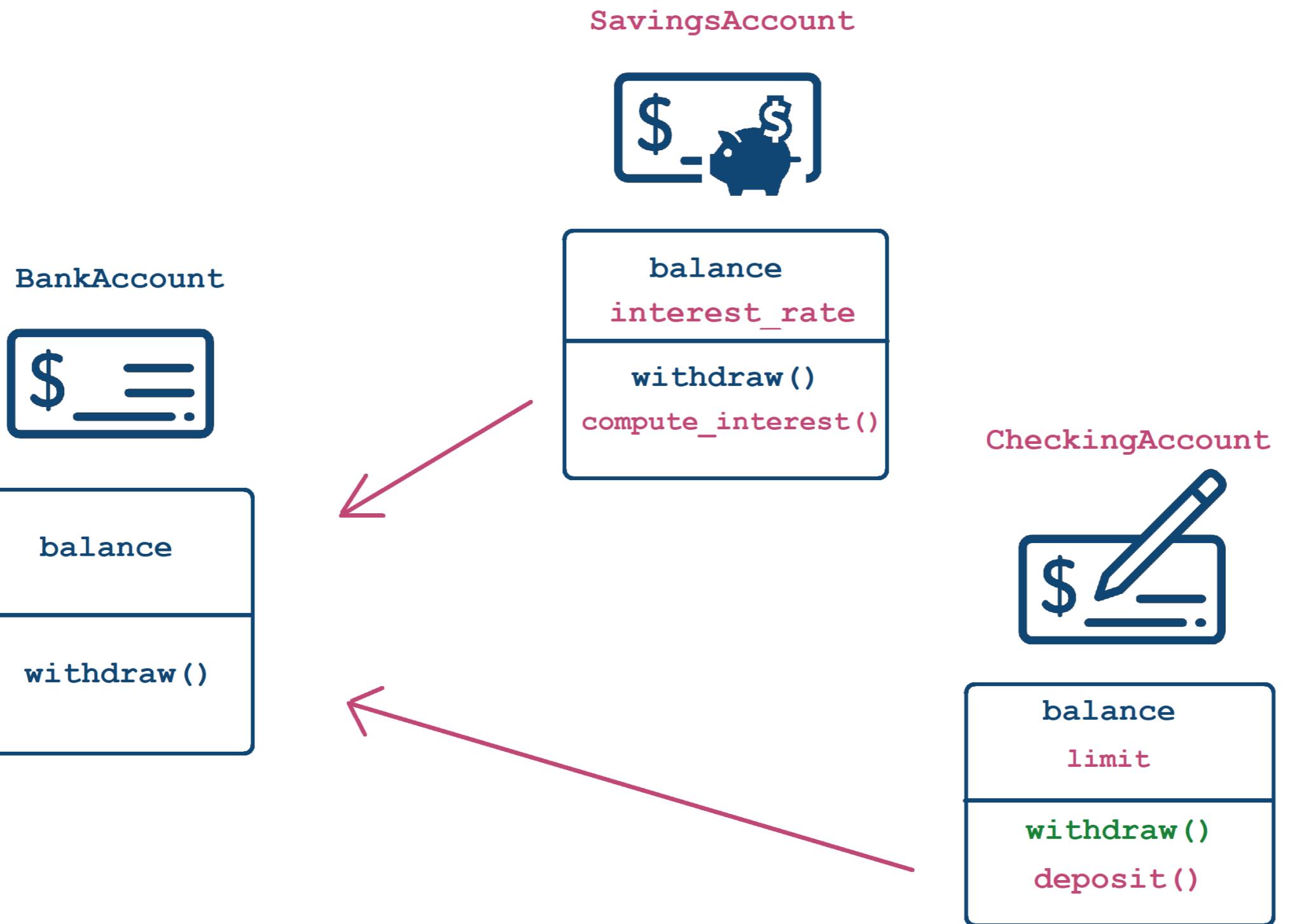
Customizing functionality via inheritance

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh

Content Quality Analyst @ DataCamp





What we have so far

```
class BankAccount:  
    def __init__(self, balance):  
        self.balance = balance  
  
    def withdraw(self, amount):  
        self.balance -=amount  
  
# Empty class inherited from BankAccount  
class SavingsAccount(BankAccount):  
    pass
```

Customizing constructors

```
class SavingsAccount(BankAccount):

    # Constructor specifically for SavingsAccount with an additional parameter
    def __init__(self, balance, interest_rate):

        # Call the parent constructor using ClassName.__init__()
        BankAccount.__init__(self, balance) # <--- self is a SavingsAccount but also a BankAccount

        # Add more functionality
        self.interest_rate = interest_rate
```

- Can run constructor of the parent class first by `Parent.__init__(self, args...)`
- Add more functionality
- Don't have to call the parent constructors

Create objects with a customized constructor

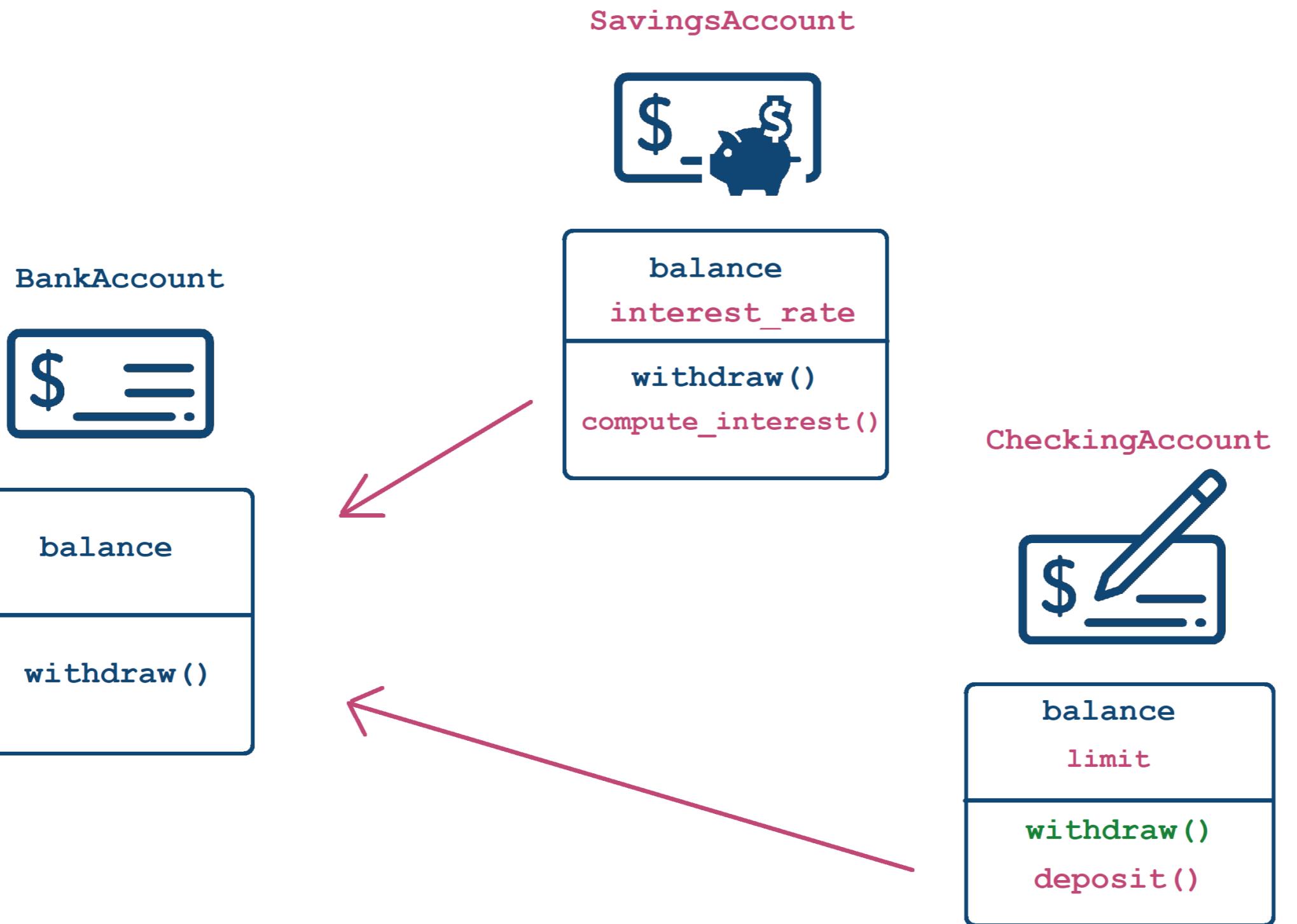
```
# Construct the object using the new constructor  
acct = SavingsAccount(1000, 0.03)  
acct.interest_rate
```

0.03

Adding functionality

- Add methods as usual
- Can use the data from both the parent and the child class

```
class SavingsAccount(BankAccount):  
  
    def __init__(self, balance, interest_rate):  
        BankAccount.__init__(self, balance)  
        self.interest_rate = interest_rate  
  
    # New functionality  
    def compute_interest(self, n_periods = 1):  
        return self.balance * ( (1 + self.interest_rate) ** n_periods - 1)
```



Customizing functionality

```
class CheckingAccount(BankAccount):  
    def __init__(self, balance, limit):  
        BankAccount.__init__(self, content)  
        self.limit = limit  
  
    def deposit(self, amount):  
        self.balance += amount  
  
    def withdraw(self, amount, fee=0):  
        if fee <= self.limit:  
            BankAccount.withdraw(self, amount - fee)  
        else:  
            BankAccount.withdraw(self,  
                                amount - self.limit)
```

- Can change the signature (add parameters)
- Use `Parent.method(self, args...)` to call a method from the parent class

```
check_acct = CheckingAccount(1000, 25)

# Will call withdraw from CheckingAccount
check_acct.withdraw(200)
```

```
# Will call withdraw from CheckingAccount
check_acct.withdraw(200, fee=15)
```

```
bank_acct = BankAccount(1000)

# Will call withdraw from BankAccount
bank_acct.withdraw(200)
```

```
# Will produce an error
bank_acct.withdraw(200, fee=15)
```

TypeError: withdraw() got an unexpected keyword argument 'fee'

Let's practice!

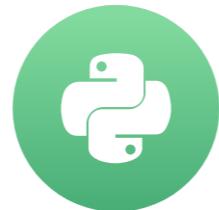
OBJECT-ORIENTED PROGRAMMING IN PYTHON

Operator overloading: comparison

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh

Content Quality Analyst @ DataCamp



Object equality

```
class Customer:  
    def __init__(self, name, balance):  
        self.name, self.balance = name, balance  
  
customer1 = Customer("Maryam Azar", 3000)  
customer2 = Customer("Maryam Azar", 3000)  
  
customer1 == customer2
```

False

Object equality

```
class Customer:  
    def __init__(self, name, balance, id):  
        self.name, self.balance = name, balance  
        self.id = id  
  
customer1 = Customer("Maryam Azar", 3000, 123)  
customer2 = Customer("Maryam Azar", 3000, 123)  
customer1 == customer2
```

False

Variables are references

```
customer1 = Customer("Maryam Azar", 3000, 123)  
customer2 = Customer("Maryam Azar", 3000, 123)
```

```
print(customer1)
```

```
<__main__.Customer at 0x1f8598e2e48>
```

```
print(customer2)
```

```
<__main__.Customer at 0x1f8598e2240>
```

Custom comparison

```
import numpy as np

# Two different arrays containing the same data
array1 = np.array([1,2,3])
array2 = np.array([1,2,3])

array1 == array2
```

```
True
```

Overloading `__eq__()`

```
class Customer:  
    def __init__(self, id, name):  
        self.id, self.name = id, name  
  
    # Will be called when == is used  
    def __eq__(self, other):  
        # Diagnostic printout  
        print("__eq__() is called")  
  
        # Returns True if all attributes match  
        return (self.id == other.id) and \  
               (self.name == other.name)
```

- `__eq__()` is called when 2 objects of a class are compared using `==`
- accepts 2 arguments, `self` and `other` - objects to compare
- returns a Boolean

Comparison of objects

```
# Two equal objects
```

```
customer1 = Customer(123, "Maryam Azar")
customer2 = Customer(123, "Maryam Azar")
```

```
customer1 == customer2
```

```
__eq__() is called
True
```

```
# Two unequal objects - different ids
```

```
customer1 = Customer(123, "Maryam Azar")
customer2 = Customer(456, "Maryam Azar")
```

```
customer1 == customer2
```

```
__eq__() is called
False
```

Other comparison operators

Operator	Method
<code>==</code>	<code>__eq__()</code>
<code>!=</code>	<code>__ne__()</code>
<code>>=</code>	<code>__ge__()</code>
<code><=</code>	<code>__le__()</code>
<code>></code>	<code>__gt__()</code>
<code><</code>	<code>__lt__()</code>

- `__hash__()` to use objects as dictionary keys
and in sets

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Operator overloading: string representation

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh

Content Quality Analyst @ DataCamp



Printing an object

```
class Customer:  
    def __init__(self, name, balance):  
        self.name, self.balance = name, balance  
  
cust = Customer("Maryam Azar", 3000)  
print(cust)
```

```
<__main__.Customer at 0x1f8598e2240>
```

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

```
[1 2 3]
```

`__str__()`

- `print(obj)` , `str(obj)`

```
print(np.array([1,2,3]))
```

```
[1 2 3]
```

```
str(np.array([1,2,3]))
```

```
[1 2 3]
```

`__repr__()`

- `repr(obj)` , printing in console

```
repr(np.array([1,2,3]))
```

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

```
np.array([1,2,3])
```

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

- informal, for end user
- *string* representation

- formal, for developer
- *reproducible representation*
- fallback for `print()`

Implementation: str

```
class Customer:  
    def __init__(self, name, balance):  
        self.name, self.balance = name, balance  
  
    def __str__(self):  
        cust_str = """  
Customer:  
    name: {name}  
    balance: {balance}  
""".format(name = self.name, \  
              balance = self.balance)  
  
    return cust_str
```

```
cust = Customer("Maryam Azar", 3000)  
  
# Will implicitly call __str__()  
print(cust)
```

```
Customer:  
    name: Maryam Azar  
    balance: 3000
```

Implementation: repr

```
class Customer:  
    def __init__(self, name, balance):  
        self.name, self.balance = name, balance  
  
    def __repr__(self):  
        # Notice the '...' around name  
        return "Customer('{}', {})".format(name = self.name, balance = self.balance)  
  
cust = Customer("Maryam Azar", 3000)  
cust # <-- # Will implicitly call __repr__()
```

```
Customer('Maryam Azar', 3000) # <-- not Customer(Maryam Azar, 3000)
```

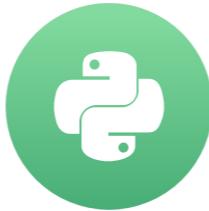
- Surround string arguments with quotation marks in the `__repr__()` output

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Exceptions

OBJECT-ORIENTED PROGRAMMING IN PYTHON



Alex Yarosh

Content Quality Analyst @ DataCamp

```
a = 1  
a / 0
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    1/0  
ZeroDivisionError: division by zero
```

```
a = 1  
a + "Hello"
```

```
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
    a + "Hello"  
TypeError: unsupported operand type(s) for +: /  
'int' and 'str'
```

```
a = [1,2,3]  
a[5]
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    a[5]  
IndexError: list index out of range
```

```
a = 1  
a + b
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    a + b  
NameError: name 'b' is not defined
```

Exception handling

- Prevent the program from terminating when an exception is raised
- `try` - `except` - `finally` :

```
try:  
    # Try running some code  
  
except ExceptionNameHere:  
    # Run this code if ExceptionNameHere happens  
  
except AnotherExceptionHere:      #<-- multiple except blocks  
    # Run this code if AnotherExceptionHere happens  
  
...  
  
finally:                          #<-- optional  
    # Run this code no matter what
```

Raising exceptions

- `raise ExceptionNameHere('Error message here')`

```
def make_list_of_ones(length):  
    if length <= 0:  
        raise ValueError("Invalid length!") # <--- Will stop the program and raise an error  
    return [1]*length
```

```
make_list_of_ones(-1)
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    make_list_of_ones(-1)  
  File "<stdin>", line 3, in make_list_of_ones  
    raise ValueError("Invalid length!")  
ValueError: Invalid length!
```

Exceptions are classes

- standard exceptions are inherited from `BaseException` or `Exception`

```
BaseException
+-- Exception
    +-- ArithmeticError          # <---
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError     # <---
    +-- TypeError
    +-- ValueError
        +-- UnicodeError
            +-- UnicodeDecodeError
            +-- UnicodeEncodeError
            +-- UnicodeTranslateError
    +-- RuntimeError
...
+-- SystemExit
...
```

¹ <https://docs.python.org/3/library/exceptions.html>

Custom exceptions

- Inherit from `Exception` or one of its subclasses
- Usually an empty class

```
class BalanceError(Exception): pass
```

```
class Customer:  
    def __init__(self, name, balance):  
        if balance < 0 :  
            raise BalanceError("Balance has to be non-negative!")  
        else:  
            self.name, self.balance = name, balance
```

```
cust = Customer("Larry Torres", -100)
```

Traceback (most recent call last):

```
  File "script.py", line 11, in <module>
    cust = Customer("Larry Torres", -100)
  File "script.py", line 6, in __init__
    raise BalanceError("Balance has to be non-negative!")
BalanceError: Balance has to be non-negative!
```

- Exception interrupted the constructor → object not created

```
cust
```

Traceback (most recent call last):

```
  File "<stdin>", line 1, in <module>
    cust
```

```
NameError: name 'cust' is not defined
```

Catching custom exceptions

```
try:  
    cust = Customer("Larry Torres", -100)  
except BalanceError:  
    cust = Customer("Larry Torres", 0)
```

Let's practice!

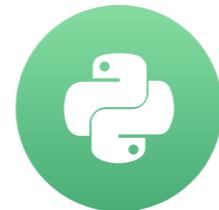
OBJECT-ORIENTED PROGRAMMING IN PYTHON

Designing for inheritance and polymorphism

OBJECT-ORIENTED PROGRAMMING IN PYTHON

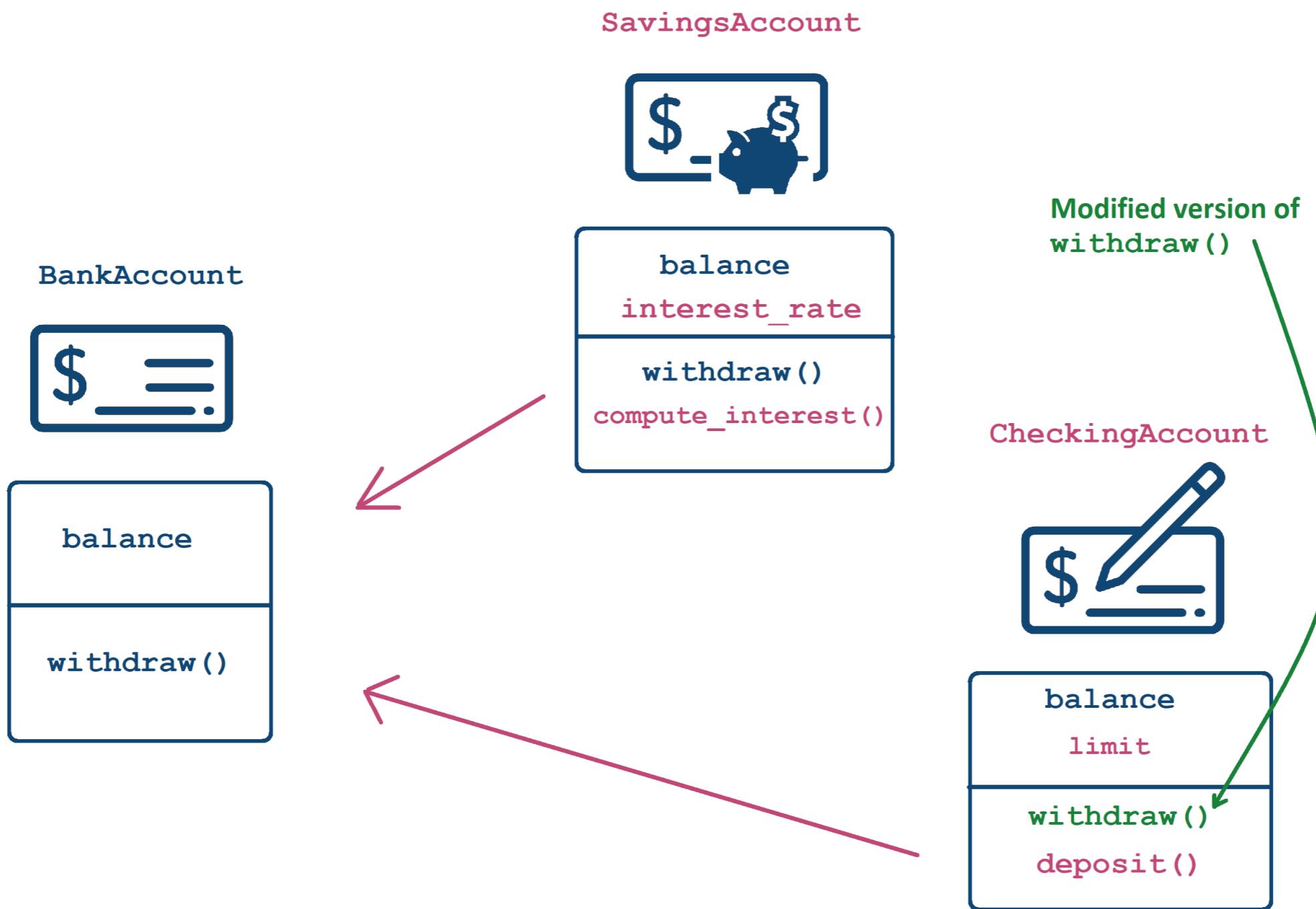
Alex Yarosh

Content Quality Analyst @ DataCamp



Polymorphism

Using a unified interface to operate on objects of different classes



All that matters is the interface

```
# Withdraw amount from each of accounts in list_of_accounts
def batch_withdraw(list_of_accounts, amount):
    for acct in list_of_accounts:
        acct.withdraw(amount)

b, c, s = BankAccount(1000), CheckingAccount(2000), SavingsAccount(3000)
batch_withdraw([b,c,s]) # <-- Will use BankAccount.withdraw(),
                        # then CheckingAccount.withdraw(),
                        # then SavingsAccount.withdraw()
```

- `batch_withdraw()` doesn't need to check the object to know which `withdraw()` to call

Liskov substitution principle

Base class should be interchangeable with any of its subclasses without altering any properties of the program

Wherever `BankAccount` works,
`CheckingAccount` should work as well



Liskov substitution principle

Base class should be interchangeable with any of its subclasses without altering any properties of the program

Syntactically

- function signatures are compatible
 - arguments, returned values

Semantically

- the state of the object and the program remains consistent
 - subclass method doesn't strengthen input conditions
 - subclass method doesn't weaken output conditions
 - no additional exceptions

Violating LSP

→ Syntactic incompatibility

`BankAccount.withdraw()` requires 1 parameter, but `CheckingAccount.withdraw()` requires 2

→ Subclass strengthening input conditions

`BankAccount.withdraw()` accepts any amount, but `CheckingAccount.withdraw()` assumes that the amount is limited

→ Subclass weakening output conditions

`BankAccount.withdraw()` can only leave a positive balance or cause an error,
`CheckingAccount.withdraw()` can leave balance negative

Violating LSP

- Changing additional attributes in subclass's method
- Throwing additional exceptions in subclass's method

No LSP – No Inheritance

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Managing data access: private attributes

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh

Content Quality Analyst @ DataCamp



All class data is public



We are all adults here



Restricting access

- Naming conventions
- Use `@property` to customize access
- Overriding `__getattr__()` and `__setattr__()`

Naming convention: internal attributes

`obj._att_name` , `obj._method_name()`

- Starts with a single `_` → "internal"
- Not a part of the public API
- As a class user: "don't touch this"
- As a class developer: use for implementation details, helper functions..

`df._is_mixed_type` , `datetime._ymd2ord()`

Naming convention: pseudoprivate attributes

`obj.__attr_name` , `obj.__method_name()`

- Starts but doesn't end with `__` → "private"
- Not inherited
- Name mangling: `obj.__attr_name` is interpreted as `obj._MyClass__attr_name`
- Used to prevent name clashes in inherited classes

Leading and trailing `__` are only used for built-in Python methods (`__init__()` , `__repr__()`)!

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Properties

OBJECT-ORIENTED PROGRAMMING IN PYTHON



Alex Yarosh

Content Quality Analyst @ DataCamp

Changing attribute values

```
class Employee:  
    def set_name(self, name):  
        self.name = name  
    def set_salary(self, salary):  
        self.salary = salary  
    def give_raise(self, amount):  
        self.salary = self.salary + amount  
    def __init__(self, name, salary):  
        self.name, self.salary = name, salary
```

```
emp = Employee("Miriam Azari", 35000)  
# Use dot syntax and = to alter attributes  
emp.salary = emp.salary + 5000
```

Changing attribute values

```
class Employee:  
    def set_name(self, name):  
        self.name = name  
    def set_salary(self, salary):  
        self.salary = salary  
    def give_raise(self, amount):  
        self.salary = self.salary + amount  
    def __init__(self, name, salary):  
        self.name, self.salary = name, salary
```

```
emp = Employee("Miriam Azari", 35000)  
# Use dot syntax and = to alter attributes  
emp.salary = emp.salary + 5000
```

Control attribute access?

- check the value for validity
- or make attributes read-only
 - modifying `set_salary()` wouldn't prevent `emp.salary = -100`

Restricted and read-only attributes

```
import pandas as pd  
df = pd.DataFrame({"colA": [1,2], "colB": [3,4]})  
df
```

```
  colA  colB  
0      1      3  
1      2      4
```

```
# will cause an error  
df.columns = ["new_colA", "new_colB", "extra"]  
df
```

```
ValueError: Length mismatch:  
Expected axis has 2 elements,  
new values have 3 elements
```

```
df.columns = ["new_colA", "new_colB"]  
df
```

```
  new_colA  new_colB  
0      1      3  
1      2      4
```

```
df.shape = (43, 27)  
df
```

```
AttributeError: can't set attribute
```

@property

```
class Employer:  
    def __init__(self, name, new_salary):  
        self._salary = new_salary  
  
    @property  
    def salary(self):  
        return self._salary  
  
    @salary.setter  
    def salary(self, new_salary):  
        if new_salary < 0:  
            raise ValueError("Invalid salary")  
        self._salary = new_salary
```

← Use "protected" attribute with leading _ to store data

← Use @property on a *method* whose name is exactly the name of the restricted attribute; *return the internal attribute*

← Use @attr.setter on a method attr() that will be called on obj.attr = value

- the value to assign passed as argument

@property

```
class Employer:  
    def __init__(self, name, new_salary):  
        self._salary = new_salary  
  
    @property  
    def salary(self):  
        return self._salary  
  
    @salary.setter  
    def salary(self, new_salary):  
        if new_salary < 0:  
            raise ValueError("Invalid salary")  
        self._salary = new_salary
```

```
emp = Employee("Miriam Azari", 35000)  
# accessing the "property"  
emp.salary
```

35000

```
emp.salary = 60000 # <-- @salary.setter
```

```
emp.salary = -1000
```

ValueError: Invalid salary

Why use @property?

User-facing: behave like attributes

Developer-facing: give control of access

Other possibilities

→ Do not add `@attr.setter`

Create a read-only property

→ Add `@attr.getter`

Use for the method that is called when the property's *value is retrieved*

→ Add `@attr.deleter`

Use for the method that is called when the property is *deleted using del*

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Congratulations!

OBJECT-ORIENTED PROGRAMMING IN PYTHON



Alex Yarosh

Content Quality Analyst @ DataCamp

Overview

Chapter 1

- Classes and objects
- Attributes and methods

Chapter 3

- Object equality
- String representation
- Exceptions

Chapter 2

- Class inheritance
- Polymorphism
- Class-level data

Chapter 4

- Designing for inheritance
- Levels of data access
- Properties

What's next?

Functionality

- Multiple inheritance and mix-in classes
- Overriding built-in operators like `+`
- `__getattr__()`, `__setattr__()`
- Custom iterators
- Abstract base classes
- Dataclasses (*new in Python 3.7*)

What's next?

Functionality

- Multiple inheritance and mixin classes
- Overriding built-in operators like `+`
- `__getattr__()`, `__setattr__()`
- Custom iterators
- Abstract base classes
- Dataclasses (*new in Python 3.7*)

Design

- SOLID principles

Single-responsibility principle

Open-closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

- Design patterns

Thank you!

OBJECT-ORIENTED PROGRAMMING IN PYTHON