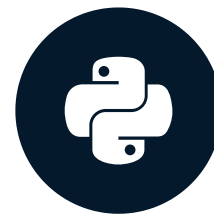


<https://www.youtube.com/watch?v=-pEs-Bss8Wc>

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

3. Instance-level data

you need to learn how to distinguish between instance-level data and class level data. Remember the employee class you defined in the previous chapter. It had attributes like name and salary, and we were able to assign specific values to them for each new instance of the class. These were instance attributes. We used self to bind them to a particular 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
```

4. Class-level data

But what if you needed to store some data that is shared among all the instances of a class? For example, if you wanted to introduce a minimal salary across the entire organization. That data should not differ among object instances. Then, you can define an attribute directly in the class body. This will create a class attribute, that will serve as a "global variable" within a class. For example,

- "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

5. Class-level data

we can define `min_salary`, and set it to 30000. We can use this attribute inside the class like we would use any global variable, only prepended by the class name: for example, here we check if the value of salary attribute is greater than `MIN_SALARY` in the `init` method. Note that we do not use `self` to define the attribute, and we use the class name instead of `self` when referring to the attribute.

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

6. Class-level data

This `min_salary` variable will be shared among all the instances of the `employee` class. We can access it like any other attribute from an object instance, and the value will be the same across instances. Here we print the `MIN_SALARY` class attribute from two `employee` objects.

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
- ...

7. Why use class attributes?

So, the main use case for class attributes is global constants that are related to class, for example min/max values for attributes -- like the `min_salary` example -- or commonly used values: for example, if you were defining a `Circle` class, you could store `pi` as a class attribute.

Class methods

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

```
class MyClass:
```

```
    @classmethod
```

```
    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...)
```

```
# <---use decorator to declare a class method
```

8. Class methods

What about methods? Regular methods are already shared between instances: the same code gets executed for every instance. The only difference is the data that is fed into it. It is possible to define methods bound to class rather than an instance, but they have a narrow application scope, because these methods will not be able to use any instance-level data. To define a class method, you start with a classmethod decorator, followed by a method definition. The only difference is that now the first argument is not self, but cls, referring to the class, just like the self argument was a reference to a particular instance. Then you write it as any other function, keeping in mind that you can't refer to any instance attributes in that method. To call a class method, we use class-dot-method syntax, rather than object-dot-method syntax.

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)
```

9. Alternative constructors

- Can only have one `__init__()`

So why would we ever need class methods at all? The main use case is alternative constructors. A class can only have one init method, but there might be multiple ways to initialize an object. For example, we might want to create an Employee object from data stored in a file. We can't use a method, because it would require an instance, and there isn't one yet! Here we introduce a class method `from_file` that accepts a file name, reads the first line from the file that presumably contains the name of the employee, and returns an object instance. In the return statement, we use the `cls` variable -- remember that now `cls` refers to the class, so this will call the init constructor, just like using `Employee` with parentheses would when used outside the class definition.

- Use class methods to create objects

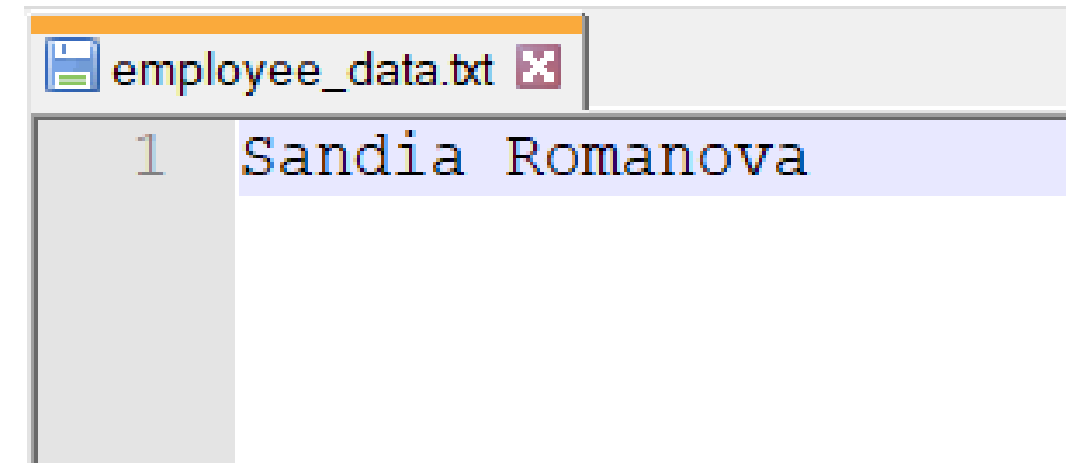
- Use `return` to return an object
- `cls(...)` will call `__init__(...)`

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)
```



1	Sandia Romanova
---	-----------------

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

```
__main__.Employee
```

10. Alternative constructors

Then we can call the method `from_file` by using class-dot-method syntax, which will create an employee object without explicitly calling the constructor.

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Class inheritance

OBJECT-ORIENTED PROGRAMMING IN PYTHON



Alex Yarosh

Content Quality Analyst @ DataCamp

Code reuse

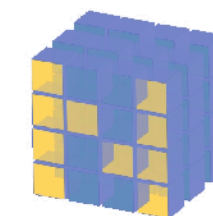
3. Code reuse

that someone has already written code that solves a part of your problem! Modules like numpy or pandas are a great tool that allows you to use code written by other programmers. But what if that code doesn't match your needs exactly? For example, you might want to modify the `to_csv` method of a pandas DataFrame to adjust the output format. You could do that by importing pandas and writing a new function, but it will not be integrated into the DataFrame interface. OOP will allow you to keep interface consistent while customizing functionality.

Code reuse

1. Someone has already done it

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



NumPy



4. Code reuse

You will also often find yourself reusing your own code over and over. For example, when building a website with a lot of graphical elements like buttons and check boxes, no matter what the element is, the basic functionality is the same: you need to be able to draw it and click on it. There are a few details that differ based on the type of the element, but a lot of the code will be the repeated. Should you copy-paste the basic code for draw and click for every new element?

5. Code reuse

Wouldn't it be better to have a general data structure like GUIelement that implements the basic draw and click functionality only once?

Code reuse

1. Someone has already done it

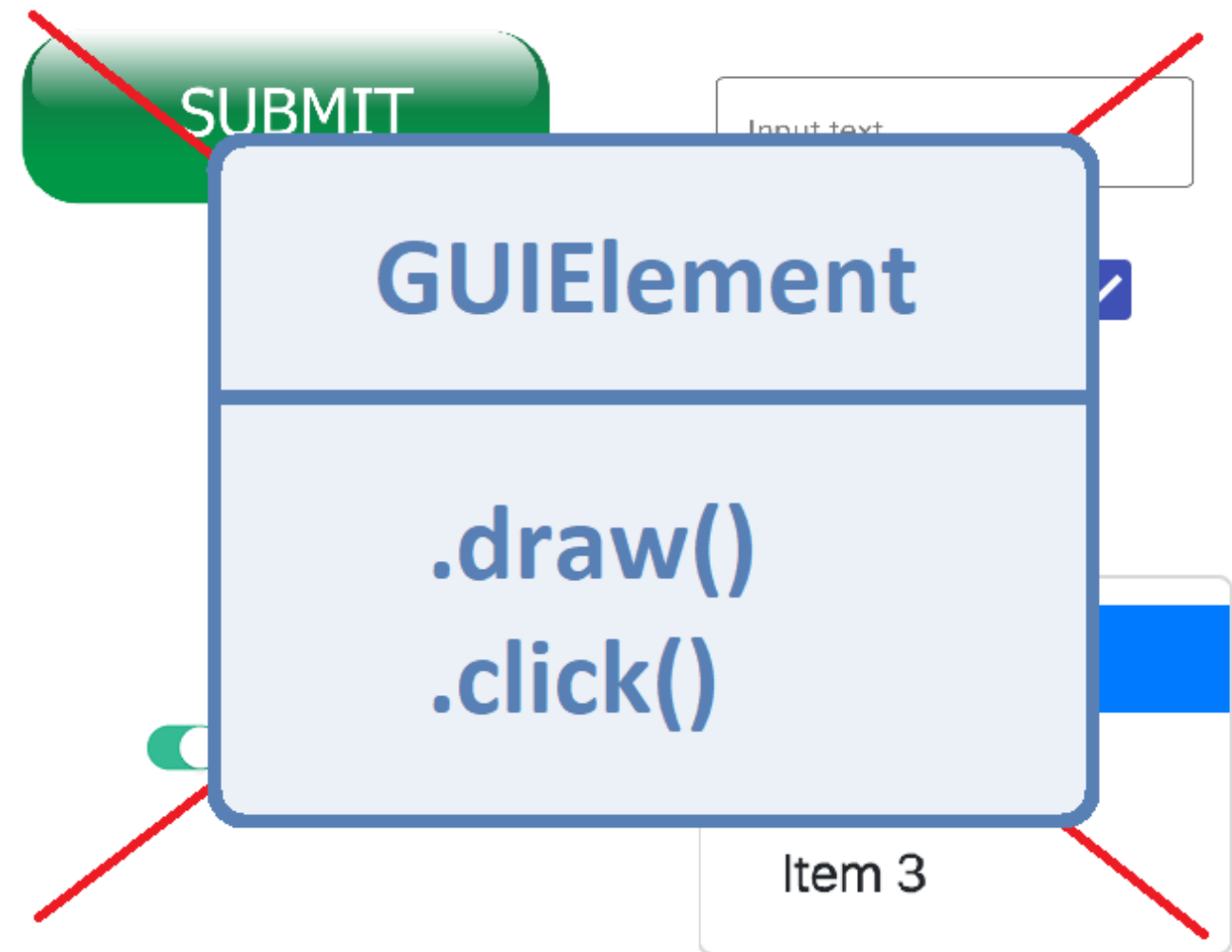
2. DRY: Don't Repeat Yourself

A collection of common web form elements:

- A green rounded rectangular button with the text "SUBMIT".
- Three radio buttons arranged vertically, labeled "One", "Two", and "Three". The "One" radio button is selected, indicated by a blue dot in the center.
- A green toggle switch, currently in the "on" position.
- A text input field with the placeholder text "Input text". Below it is a line of smaller text labeled "Helper text".
- A blue square checkbox with a white checkmark.
- A dark gray button with the text "Dropdown" and a downward arrow.
- A dropdown menu that is open, showing three items: "Item 1" (highlighted in blue), "Item 2", and "Item 3".

Code reuse

1. Someone has already done it
2. DRY: Don't Repeat Yourself



Inheritance

New class functionality = Old class functionality + extra

7. Example hierarchy

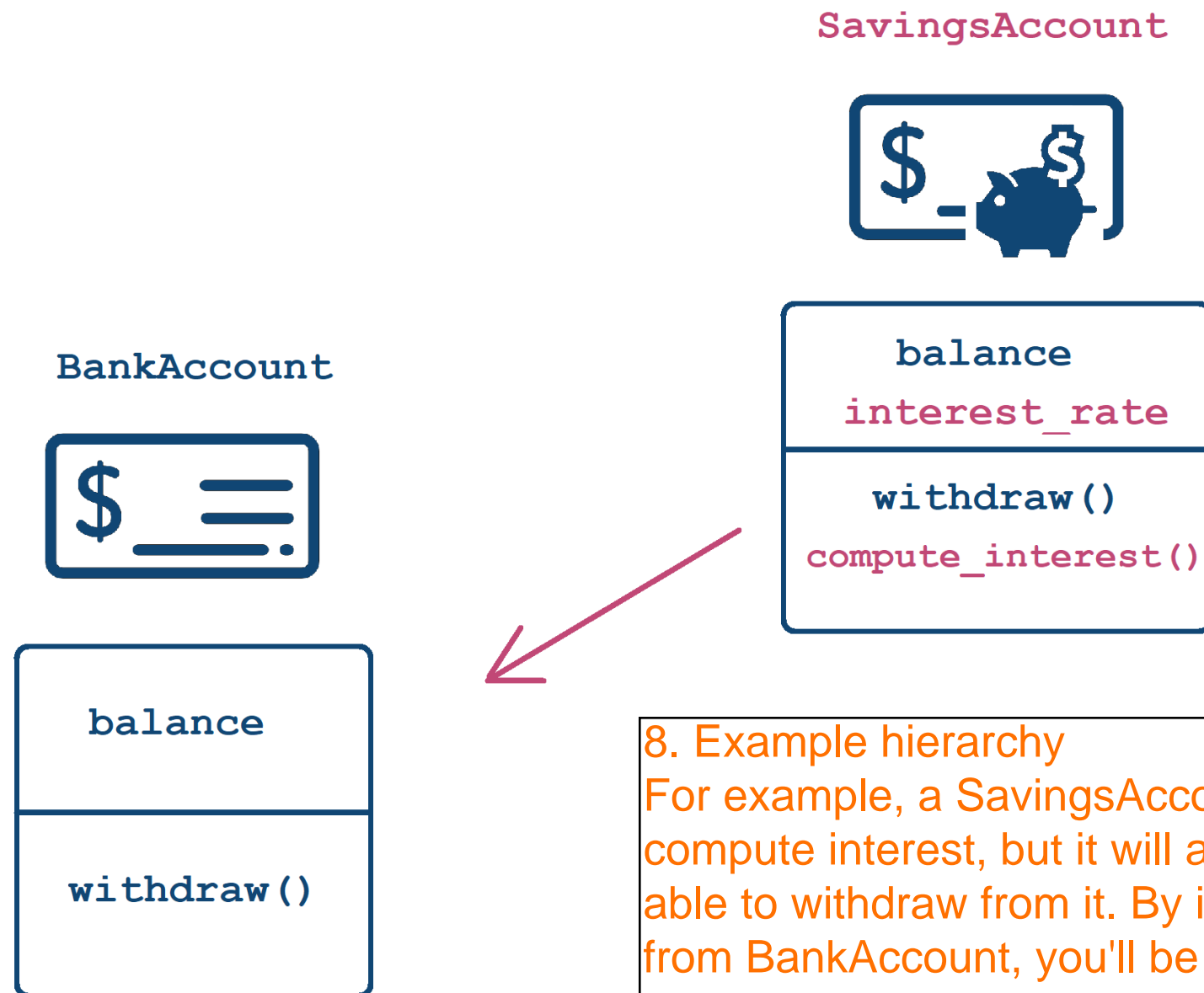
Let's say you have a basic bank account class that has a balance attribute and a withdraw method. In your company, you work with several types of accounts.

BankAccount



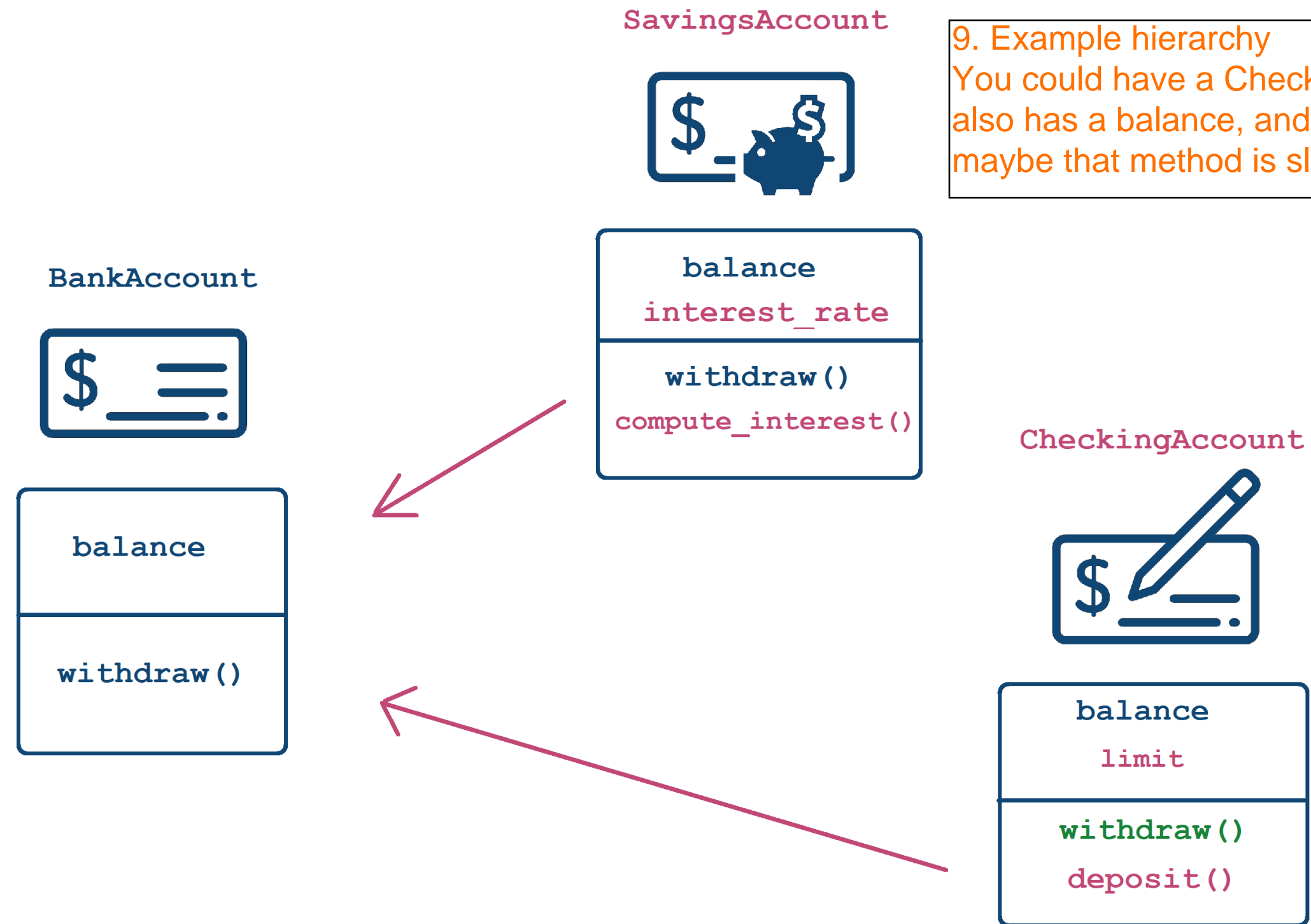
`balance`

`withdraw()`



8. Example hierarchy

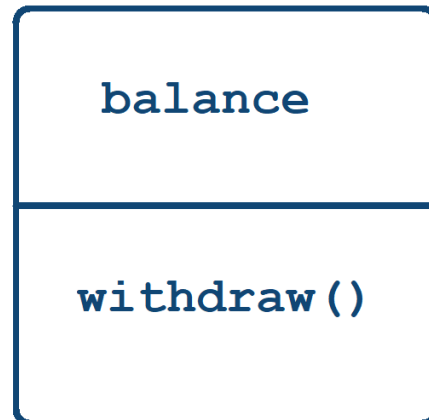
For example, a **SavingsAccount** also has an interest rate and a method to compute interest, but it will also still have a balance, and you definitely should be able to withdraw from it. By inheriting methods and attributes of **SavingsAccount** from **BankAccount**, you'll be able to reuse the code you already wrote for the **BankAccount** class.



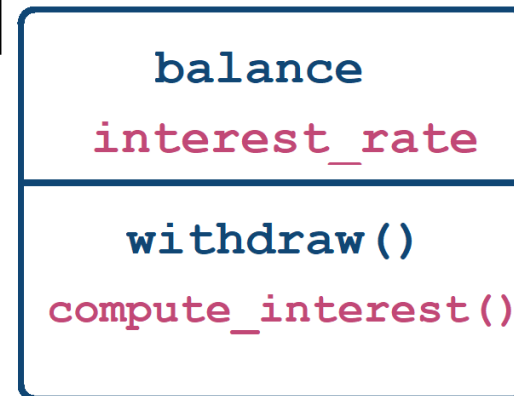
9. Example hierarchy
You could have a **CheckingAccount** class, that also has a `balance`, and a `withdraw` method, but maybe that method is slightly different:

10. Example hierarchy
it modifies the amount to be withdrawn to include a fee.
With inheritance, we'll be able to customize the withdraw
method to first adjust the amount if necessary, and then
use the method from the BankAccount class -- again,
without rewriting it.

BankAccount

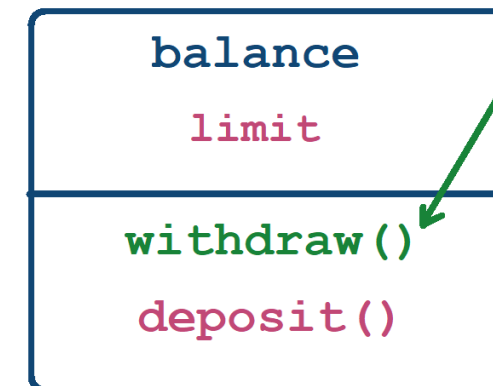


SavingsAccount



Modified version of
withdraw()

CheckingAccount



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

11. Implementing class inheritance

How do we implement this? Declaring a class that inherits from another class is very straightforward: you simply add parentheses after the class name, and then specify the class to inherit from. Here, we define a rudimentary `BankAccount` class and a seemingly empty `SavingsAccount` class inherited from it.

Child class has all of the the parent data

Constructor inherited from BankAccount

```
savings_acct = SavingsAccount(1000)  
type(savings_acct)
```

12. Child class has all of the the parent data

"Seemingly" because SavingsAccount actually has exactly as much in it as the BankAccount class. For example, we can create an object -- even though we did not define a constructor -- and we can access the balance attribute and the withdraw method from the instance of SavingsAccount, even though these features weren't defined in the new class.

```
__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

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

True

13. Inheritance: "is-a" relationship

That's because inheritance represents "is-a" relationship: a savings account is a bank account, just with some extra features. This isn't just theoretical -- that's how Python treats it as well. Calling `isinstance` function on a `savingsaccount` object shows that Python treats it like an instance of both `savingsaccount` and `BankAccount` classes, which is not the case for a generic `BankAccount` object. Right now, though, this class doesn't have anything that the original account class did not have.

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

False

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

True

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

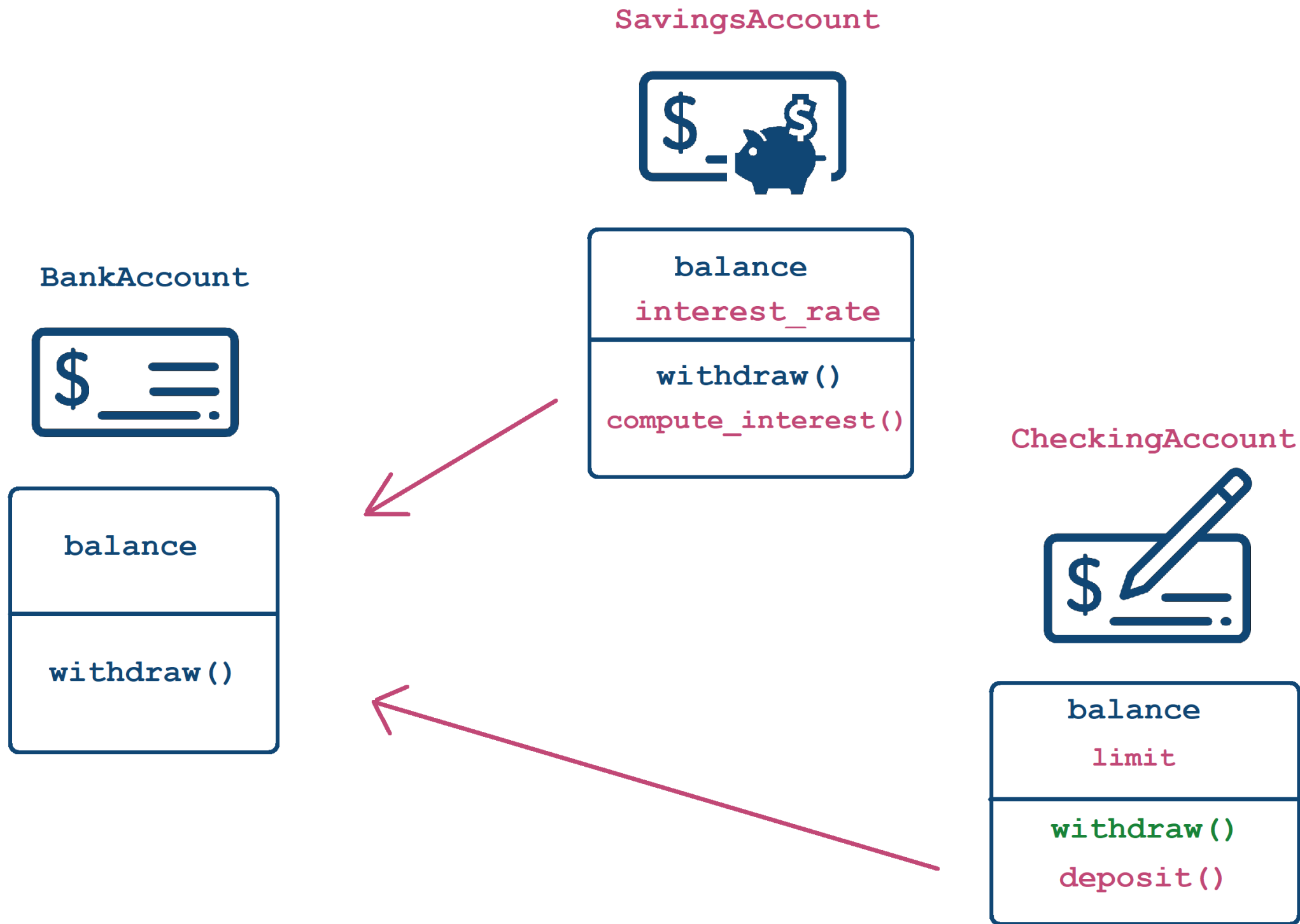
Customizing functionality via inheritance

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

4. Customizing constructors

adding a constructor specifically for SavingsAccount. It will take a balance parameter, just like BankAccount, and an additional interest_rate parameter. In that constructor, we first run the code for creating a generic bankaccount by explicitly calling the init method of the BankAccount class. Notice that we use BankAccount-dot-init to tell Python to call the constructor from the parent class, and we also pass self to that constructor. Self in this case is a SavingsAccount -- that's the class we're in -- but remember that in Python, instances of a subclass are also instances of the parent class. so it is a BankAccount as well, and we can pass it to the init method of BankAccount. Then we can add more functionality, in this case just initializing an attribute. You actually aren't required to call the parent constructor in the subclass, or to call it first -- you can use new code entirely -- but you'll likely to almost always use the parent

Create objects with a customized constructor

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

```
0.03
```

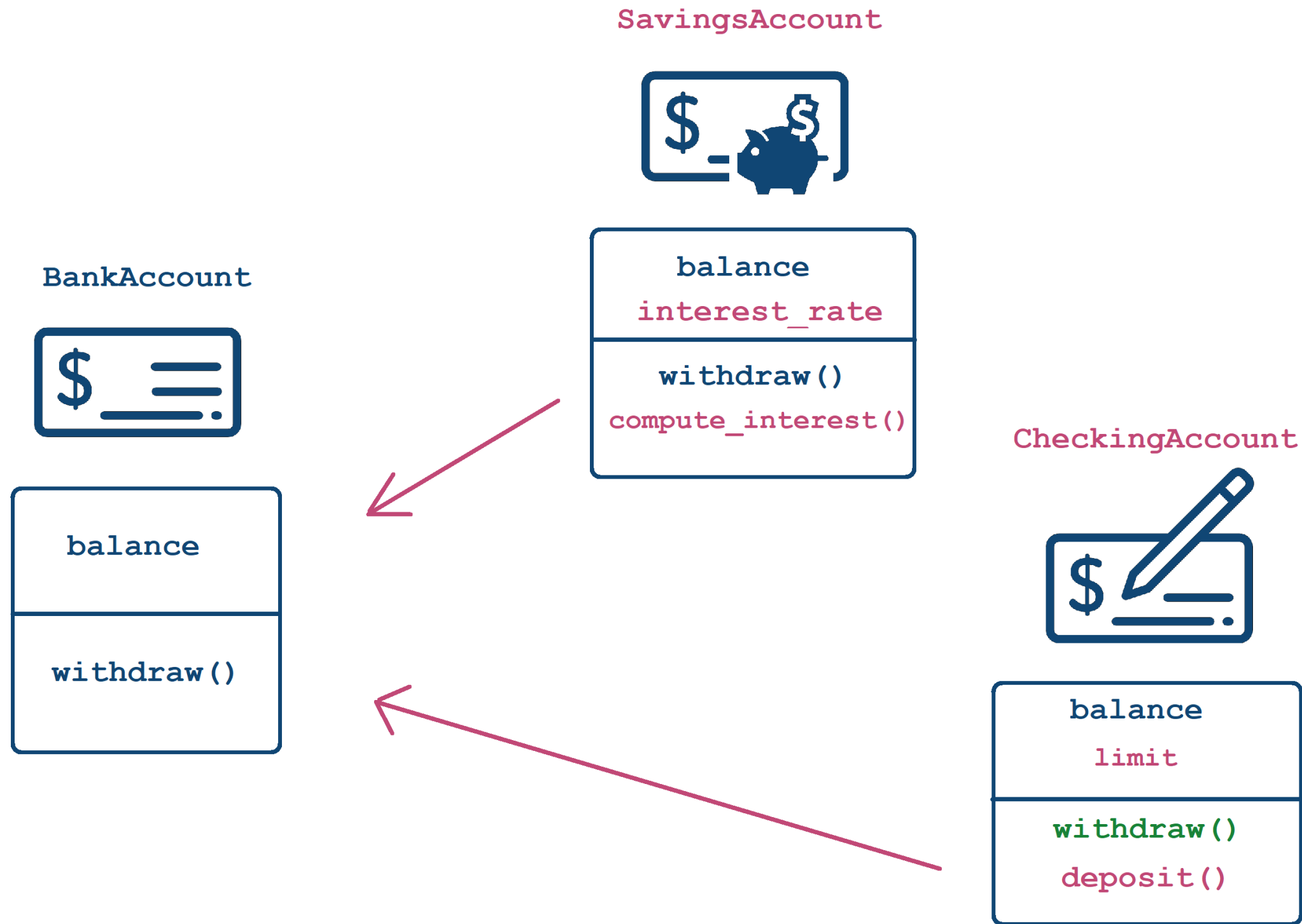
Adding functionality

- 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)
```

6. Adding functionality

In the exercises, you saw you can add methods to a subclass just like to any other class. In those methods you can use data from both the child and the parent class. For example here, we add a `compute_interest` method that returns the amount of interest in the account.. Don't worry about the exact formula, just notice that we multiply the `balance` attribute - which was inherited from the `BankAccount` parent - by an expression involving the `interest_rate` attribute that exists only in the child `SavingsAccount` class.



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

7. Customizing functionality

Now let's talk about customizing functionality. `SavingsAccount` inherits the `withdraw` method from the parent `BankAccount` class. Calling `withdraw` on a `savings` instance will execute exactly the same code as calling it on a generic bank account instance. We want to create a `CheckingAccount` class, which should have a slightly modified version of the `withdraw` method: it will have a parameter and adjust the withdrawal amount.

```
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'
```

8. Customizing functionality

Here's an outline of what that could look like. Start by inheriting from the parent class, add a customized constructor that also executes the parent code, a new deposit method, and a withdraw method, but we add a new argument to withdraw - fee, that specifies the additional withdrawal fee. We compare the fee to some fee limit, and then call the parent withdraw method, passing a new amount to it -- with fees subtracted. So this method runs almost the same code as the BankAccount's withdraw method without re-implementing it - just augmenting. Notice that we can change the signature of the method in the subclass by adding a parameter, and we again, just like in the constructor, call the parent version of the method directly by using parent-class-dot syntax and passing self.

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON