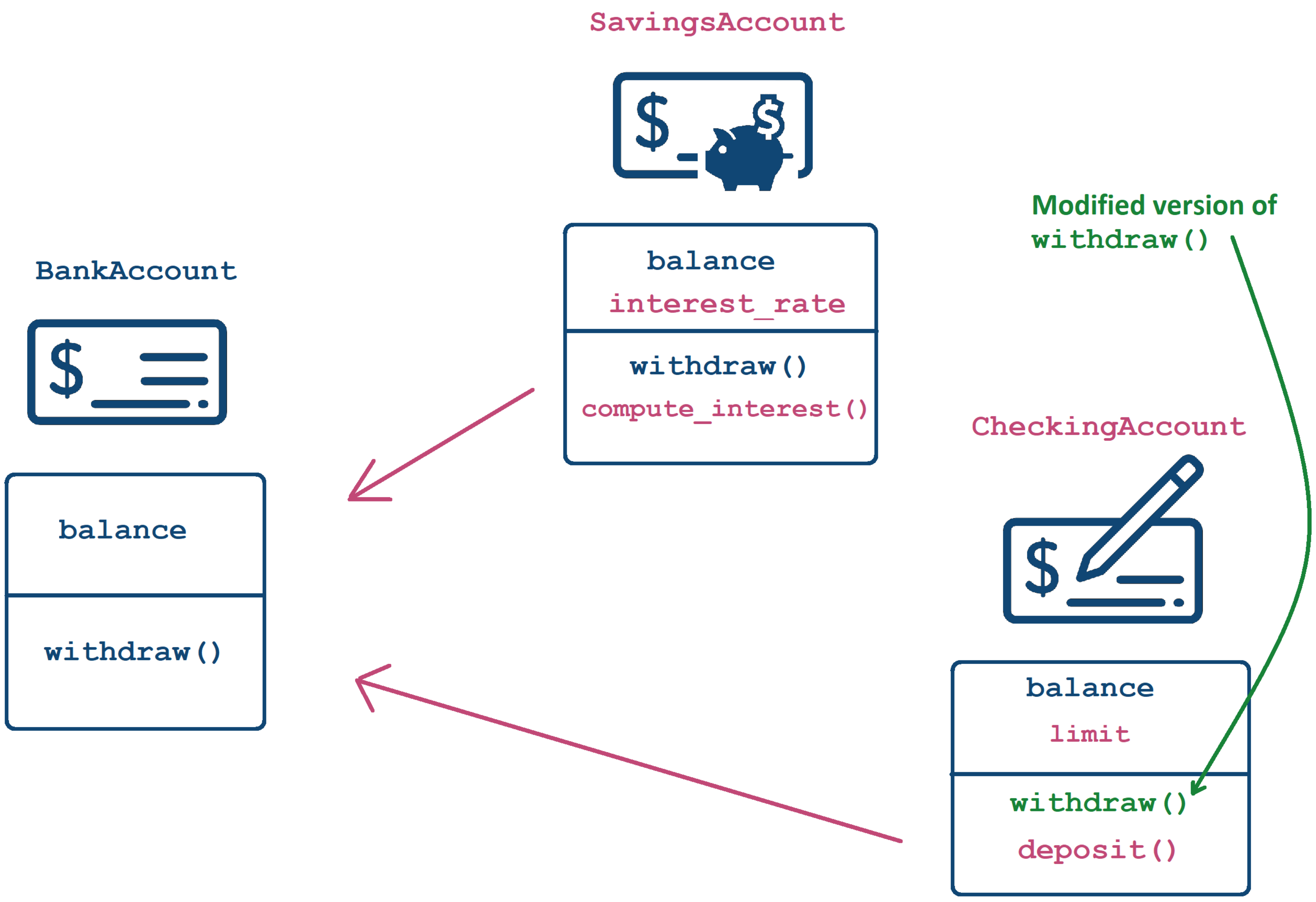
Designing for inheritance and polymorphism

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh
Content Quality Analyst @ DataCamp

Polymorphism

Using a unified interface to operate on objects of different classes


SavingsAccount
BankAccount
balance
interest_rate
withdraw()
compute_interest()
Modified version of
withdraw()
CheckingAccount
balance
withdraw()
balance
limit
withdraw()
deposit()

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

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 atributes
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 atributes
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

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

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

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

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

```python
# 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
```

```python
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

```python
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
```

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

```
35000
```

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

```python
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 2

- Class inheritance
- Polymorphism
- Class-level data

Chapter 3

- Object equality
- String representation
- Exceptions

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

***L**iskov substitution principle*

Interface segregation principle

Dependency inversion principle

- Design patterns

Thank you!

OBJECT-ORIENTED PROGRAMMING IN PYTHON