Instance and class data

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh
Content Quality Analyst @ DataCamp

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
```

- Can only have one `__init__()`

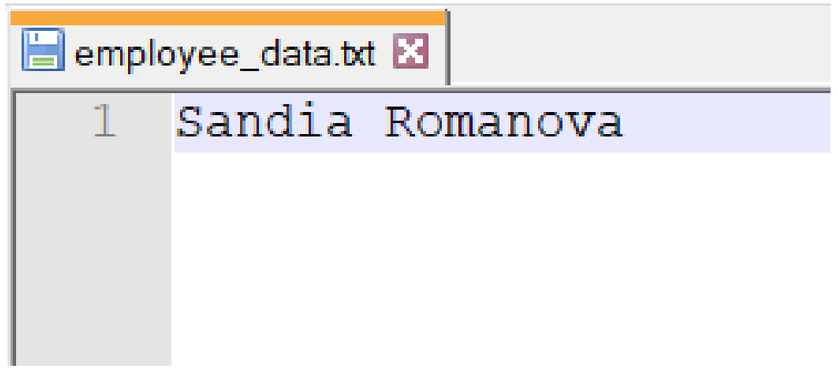
```
  @classmethod
  def from_file(cls, filename):
      with open(filename, "r") as f:
          name = f.readline()
      return cls(name)
```

- **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

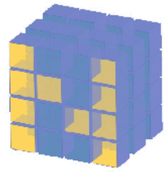
Alex Yarosh
Content Quality Analyst @ DataCamp

Code reuse

Code reuse

1. Someone has already done it

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

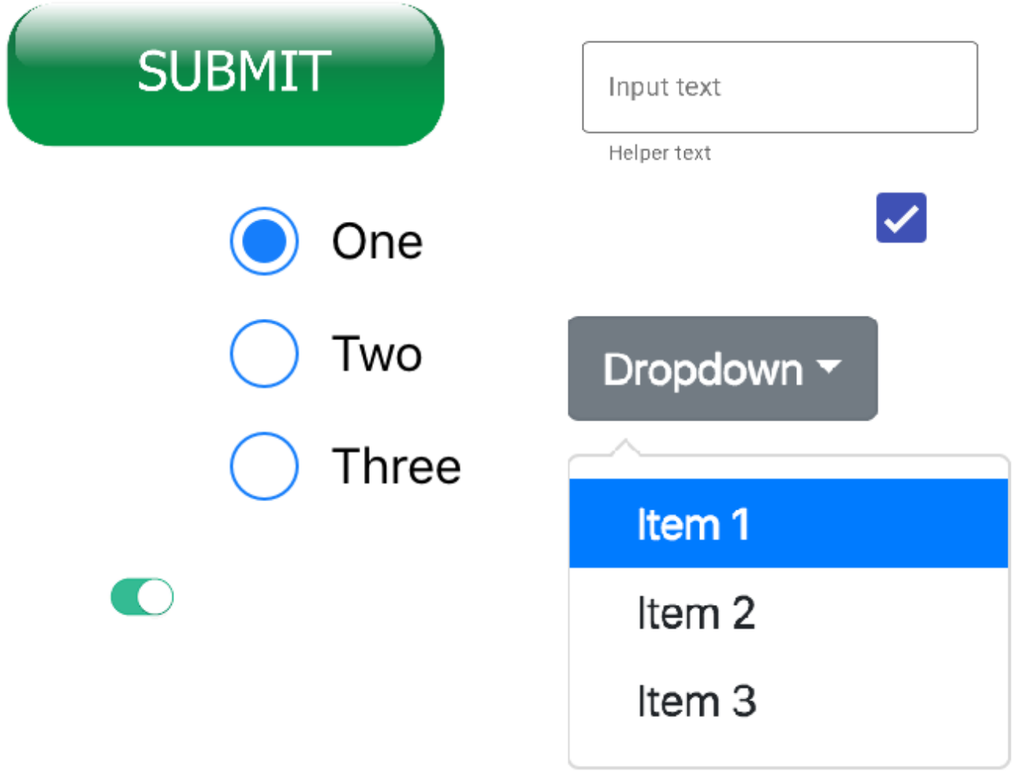


NumPy

Code reuse

1. Someone has already done it

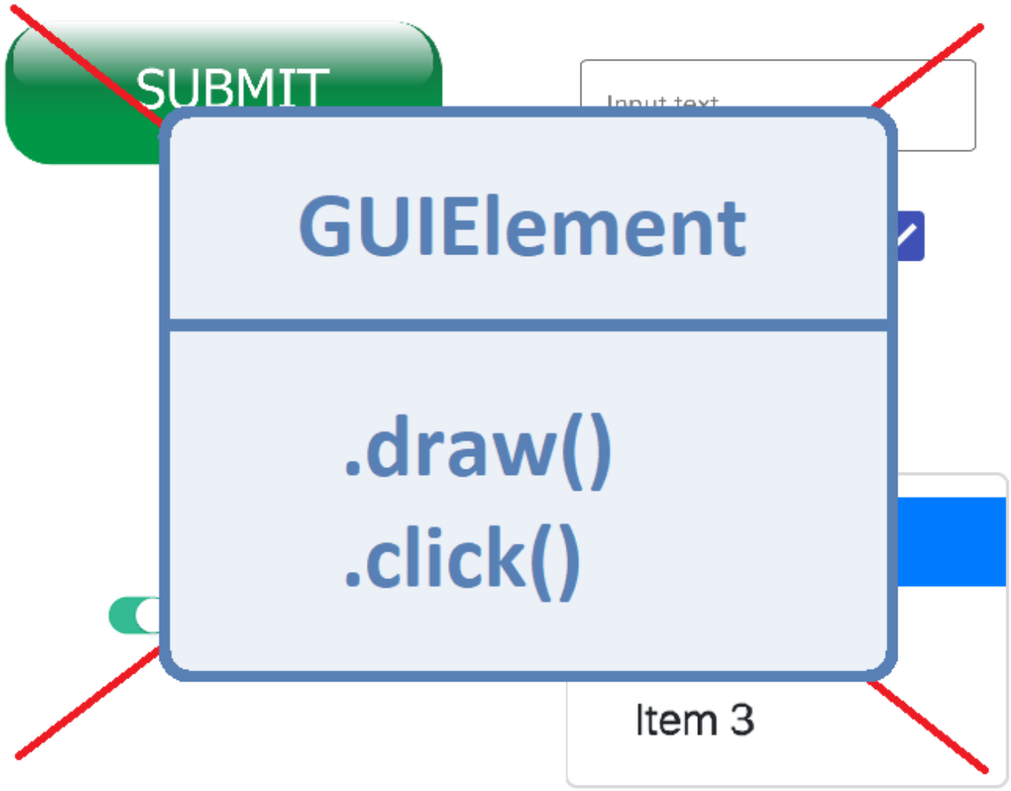
2. DRY: Don't Repeat Yourself



Code reuse

1. Someone has already done it

2. DRY: Don't Repeat Yourself



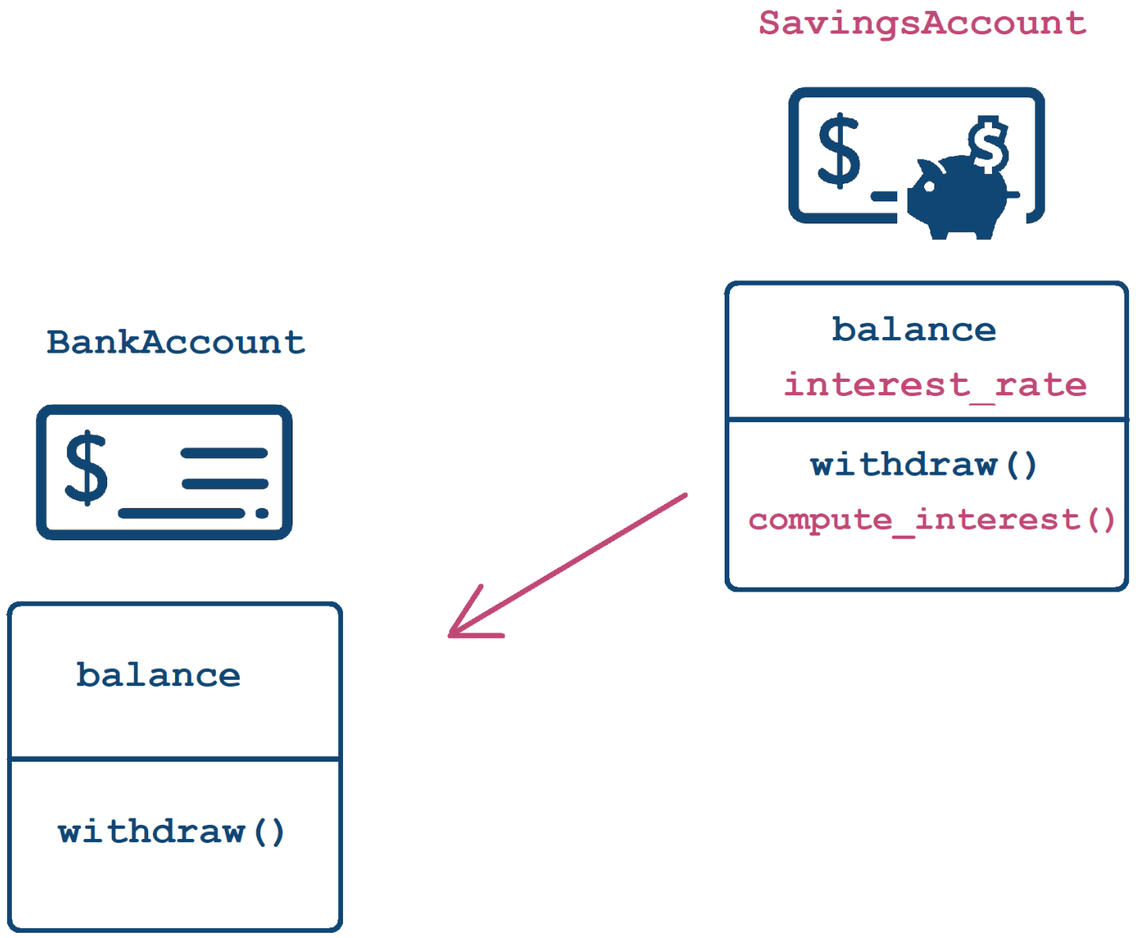
Inheritance

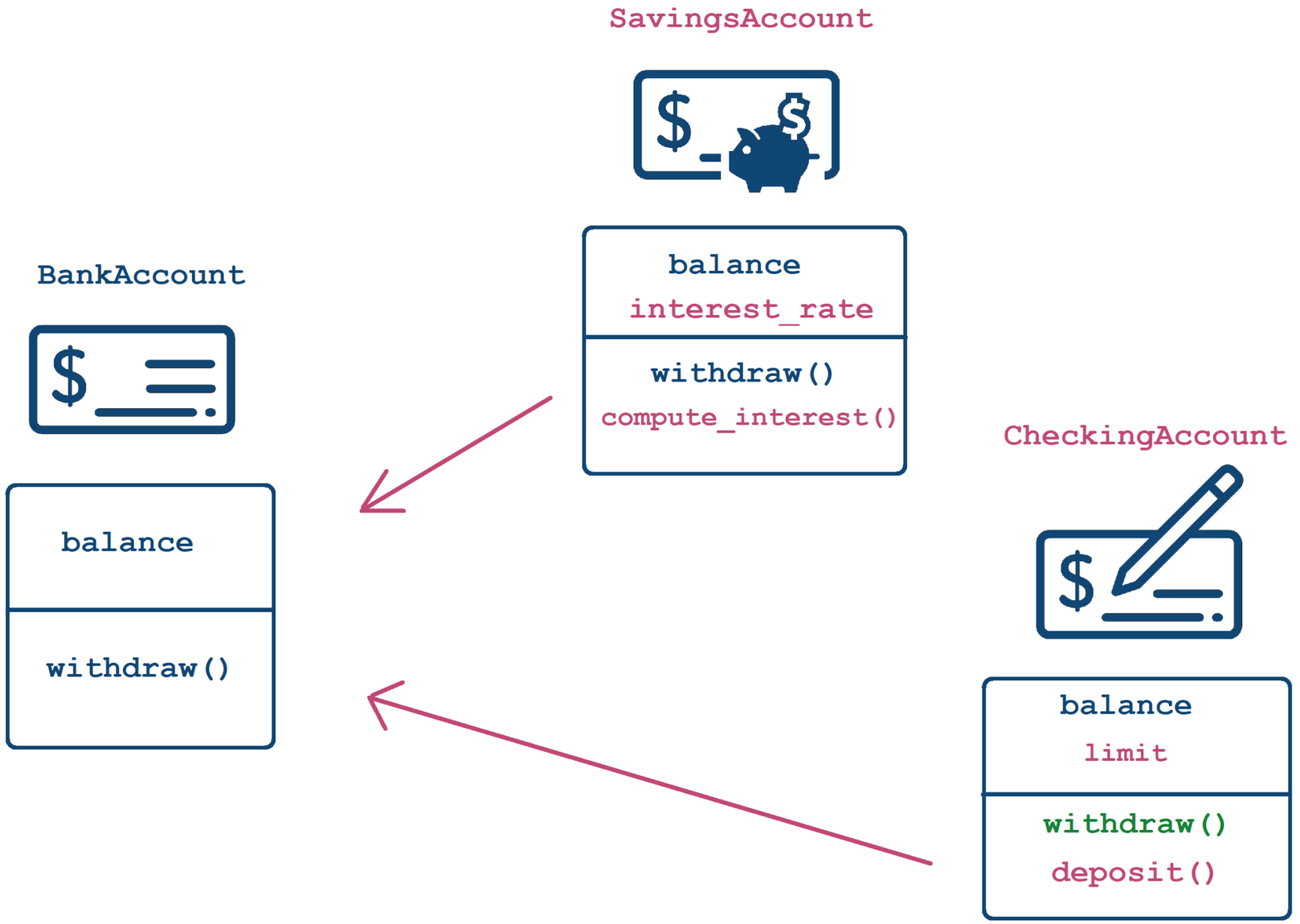
$$\text{New class functionality} = \text{Old class functionality} + \text{extra}$$

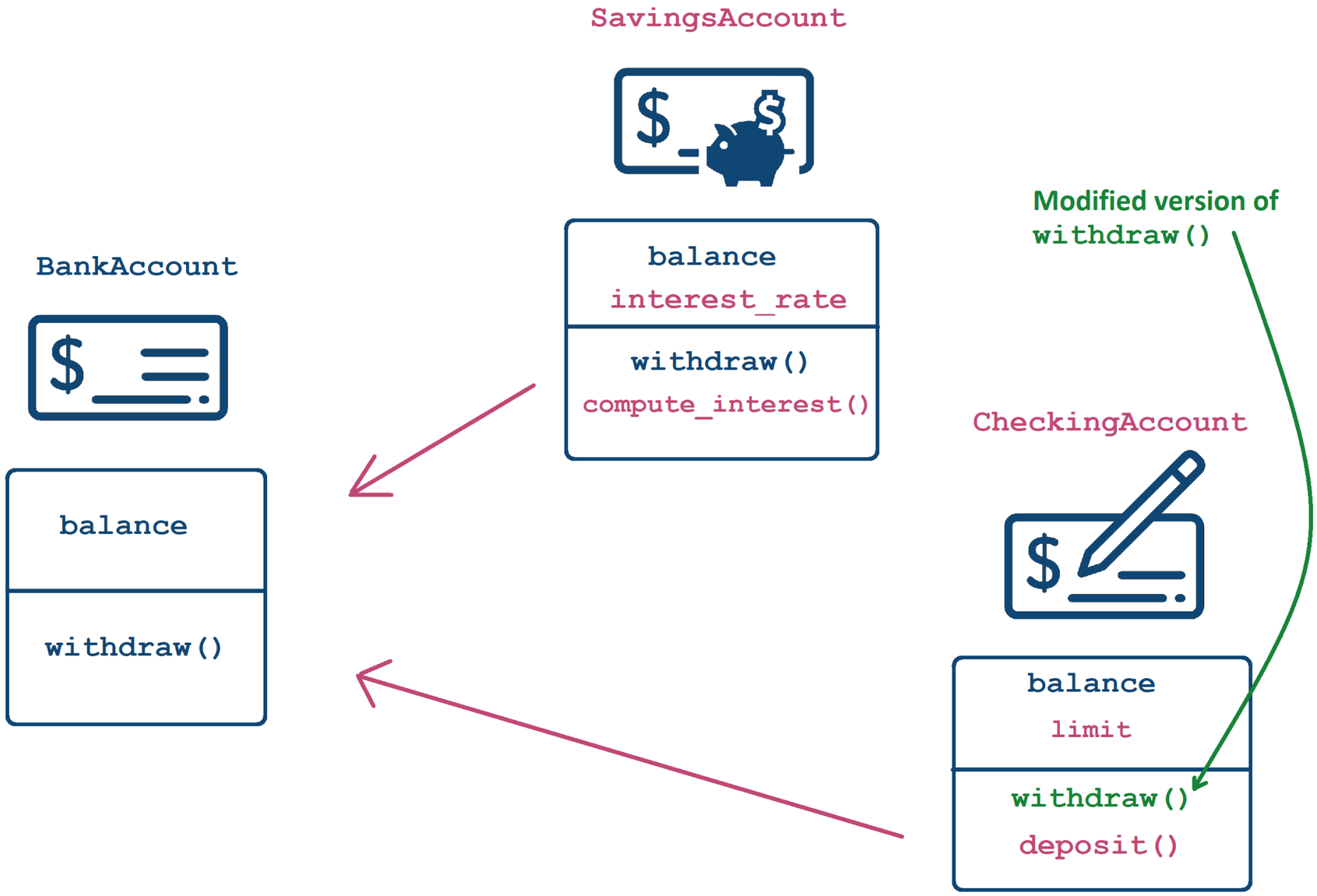
BankAccount

balance

withdraw()


SavingsAccount
balance
interest_rate
withdraw()
compute_interest()
BankAccount
balance
withdraw()


SavingsAccount
balance
interest_rate
withdraw()
compute_interest()
BankAccount
balance
withdraw()
CheckingAccount
balance
limit
withdraw()
deposit()


SavingsAccount
BankAccount
balance
withdraw()
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!

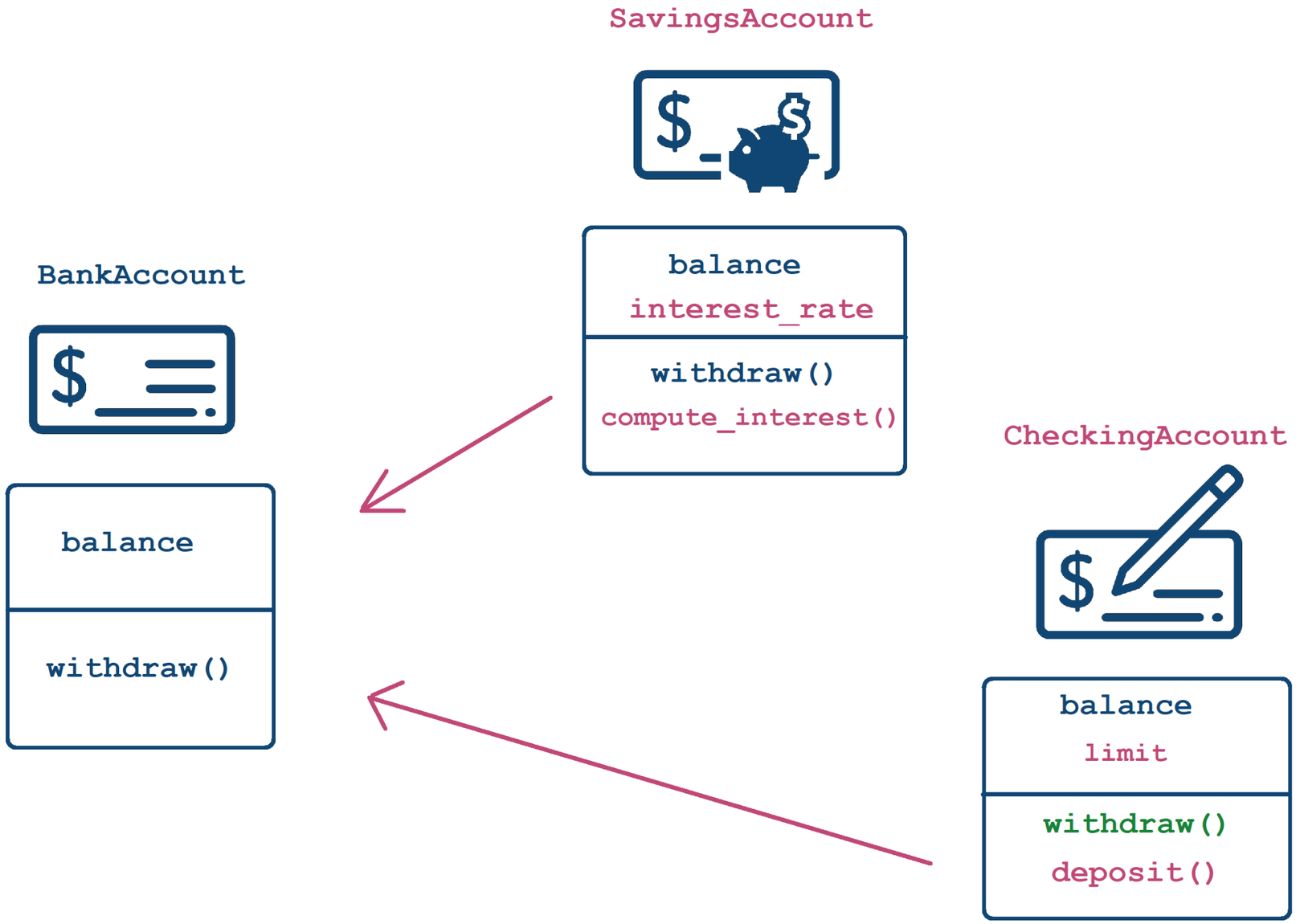
OBJECT-ORIENTED PROGRAMMING IN PYTHON

Customizing functionality via inheritance

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh
Content Quality Analyst @ DataCamp

datacamp


SavingsAccount
balance
interest_rate
withdraw()
compute_interest()
BankAccount
balance
withdraw()
CheckingAccount
balance
limit
withdraw()
deposit()

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 speficially 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
```

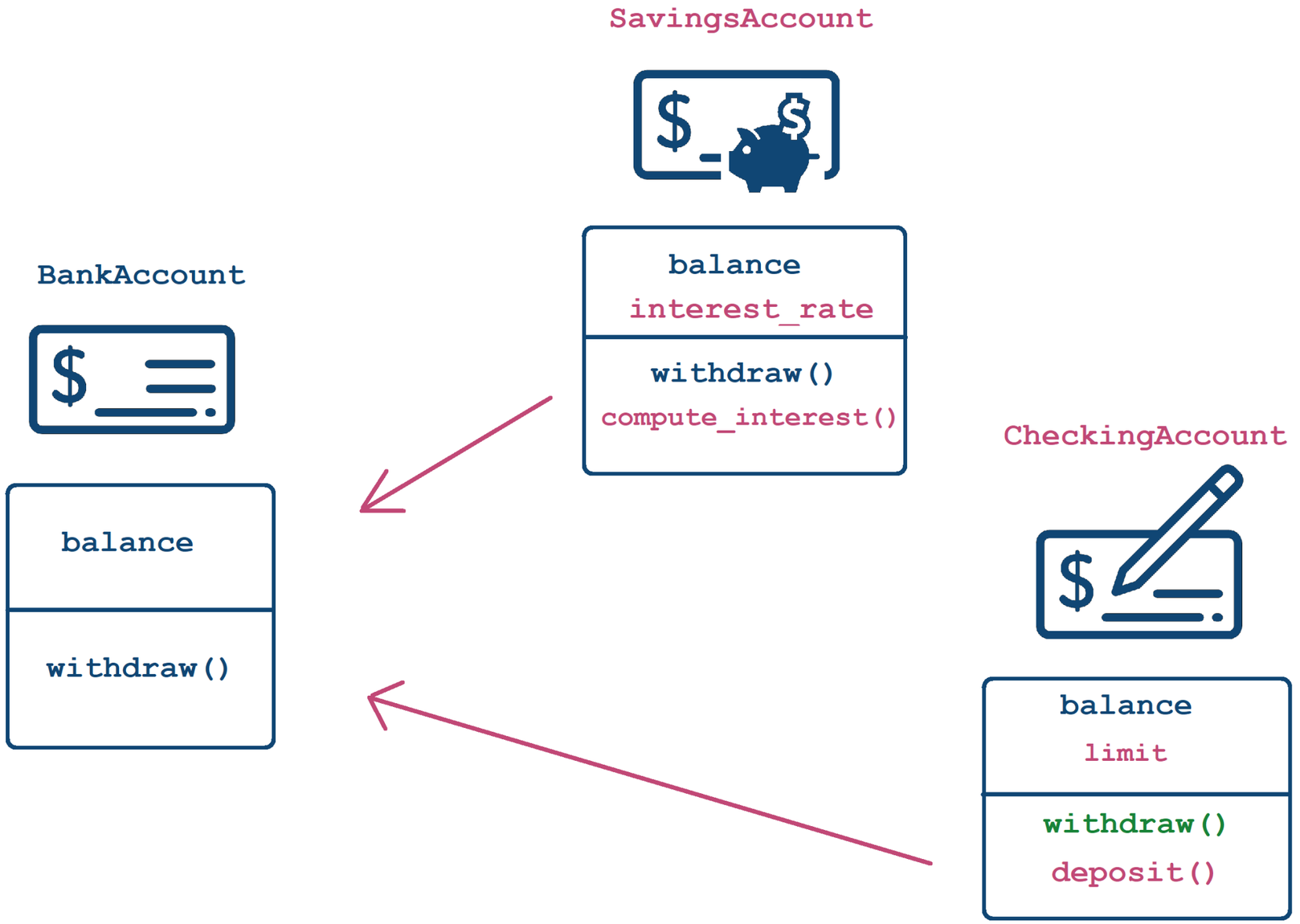
Adding functionality

- 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)
```


SavingsAccount
balance
interest_rate
withdraw()
compute_interest()
BankAccount
balance
withdraw()
CheckingAccount
balance
limit
withdraw()
deposit()

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