Computer Programming I

Binnur Kurt, PhD

Bahçeşehir University

binnur.kurt@rc.bau.edu.tr

MODULE 10

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Objects

object = state + behavior

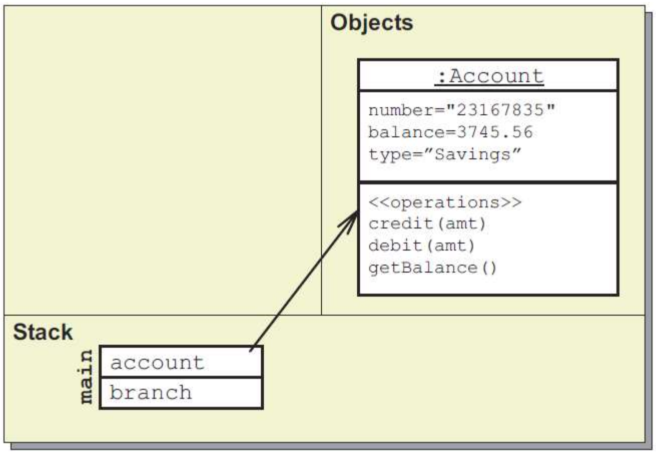
“An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class.”

(Booch Object Solutions page 305)

Objects:

- Have identity
- Are an instance of only one class
- Have attribute values that are unique to that object
- Have methods that are common to the class

Objects: Example



Classes

A class is a blueprint or prototype from which objects are created. (The Java™ Tutorials)

Classes provide:

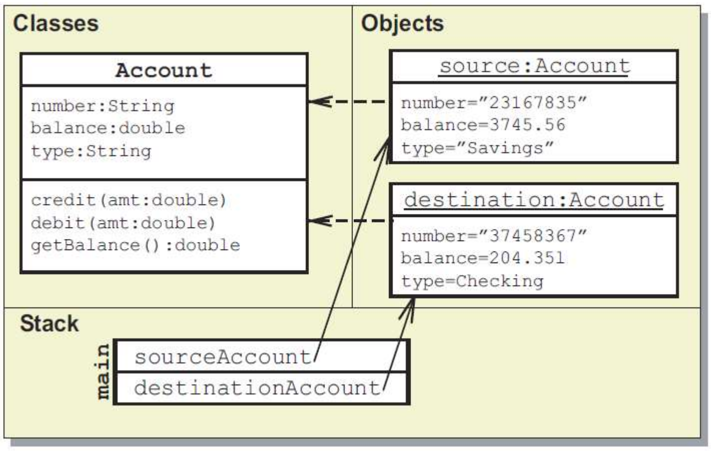
> The metadata for attributes

> The signature for methods

> The implementation of the methods (usually)

> The constructors to initialize attributes at creation time

Classes: Example



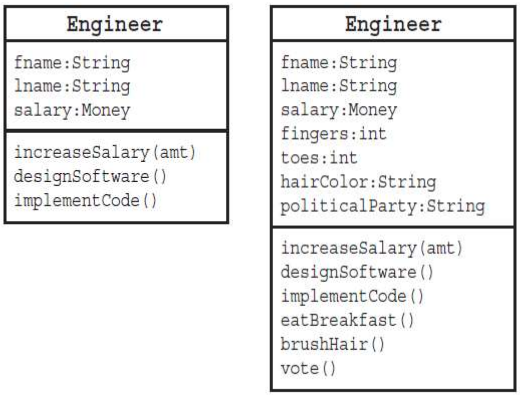
Abstraction

In OO software, the concept of abstraction enables you to create a simplified, but relevant view of a real world object within the context of the problem and solution domains.

> The abstraction object is a representation of the real world object with irrelevant (within the context of the system) behavior and data removed.

> The abstraction object is a representation of the real world object with currently irrelevant (within the context of the view) behavior and data hidden.

Abstraction: Example



Engineer

fname:String
lname:String
salary:Money

increaseSalary(amt)
designSoftware()
implementCode()

Engineer

fname:String
lname:String
salary:Money
fingers:int
toes:int
hairColor:String
politicalParty:String

increaseSalary(amt)
designSoftware()
implementCode()
eatBreakfast()
brushHair()
vote()

Encapsulation

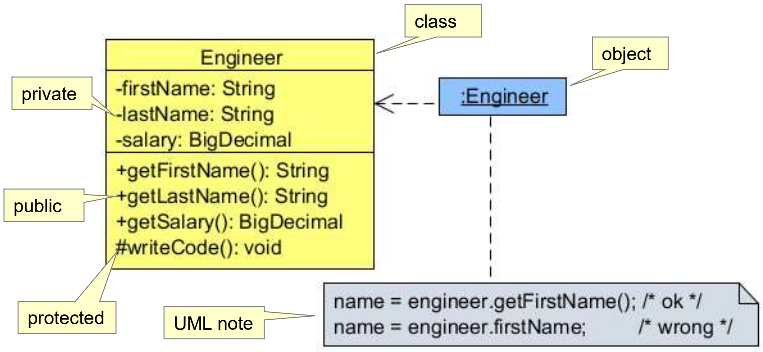
Encapsulation means “to enclose in or as if in a capsule” (Webster New Collegiate Dictionary)

> Encapsulation is essential to an object. An object is a capsule that holds the object’s internal state within its boundary.

> In most OO languages, the term encapsulation also includes *information hiding*, which can be defined as: “hide implementation details behind a set of non-private methods”.

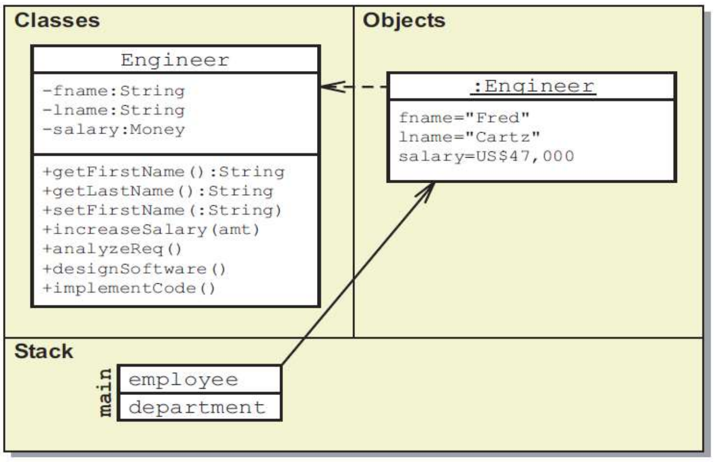
Encapsulation

> Black-Box Design



```
name = engineer.getFirstName(); /* ok */
name = engineer.firstName;        /* wrong */
```

Encapsulation: Example



Classes

Objects

Engineer

-fname:String
-lname:String
-salary:Money

+getFirstName():String
+getLastName():String
+setFirstName(:String)
+increaseSalary(amt)
+analyzeReq()
+designSoftware()
+implementCode()

:Engineer

fname="Fred"
lname="Cartz"
salary=US$47,000

Stack

main

employee

department

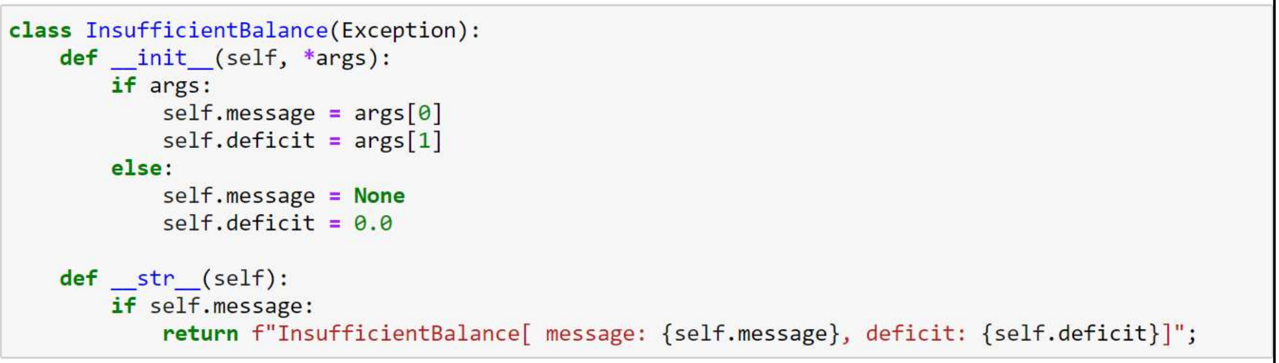
✗ `name = employee.fname;`

✗ `employee.fname = "Samantha";`

✓ `name = employee.getFirstName();`

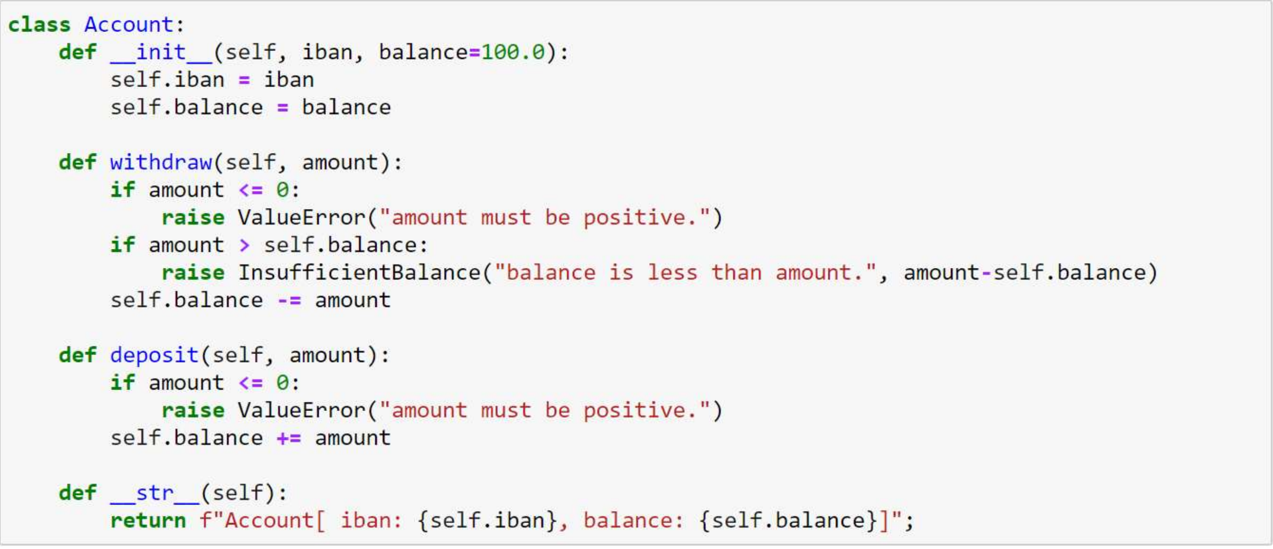
✓ `employee.setFirstName("Samantha");`

Defining a Class



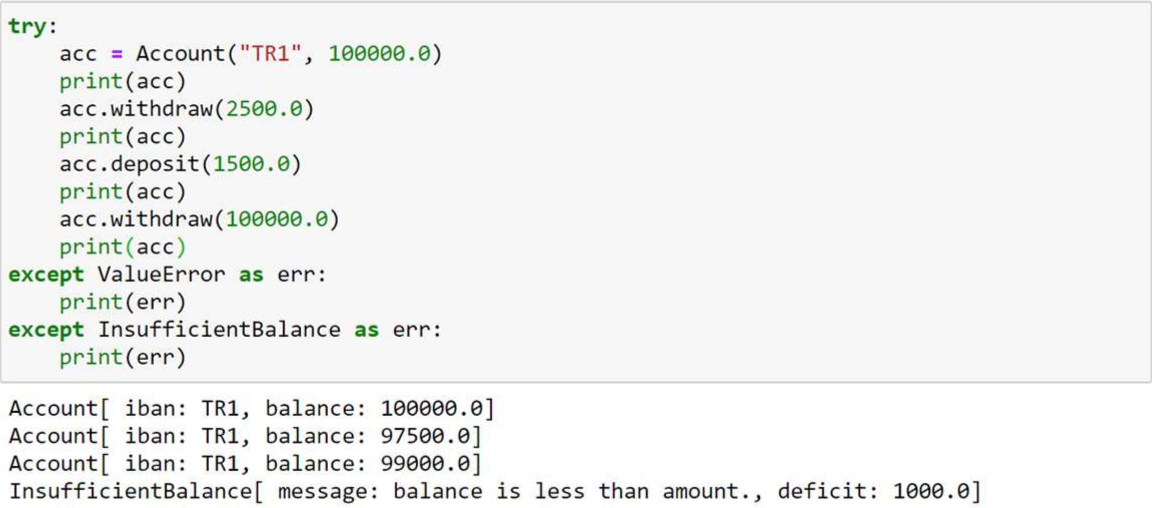
```
class InsufficientBalance(Exception):
    def __init__(self, *args):
        if args:
            self.message = args[0]
            self.deficit = args[1]
        else:
            self.message = None
            self.deficit = 0.0

    def __str__(self):
        if self.message:
            return f"InsufficientBalance[ message: {self.message}, deficit: {self.deficit}]";
```

Defining a Class



```python
class Account:
    def __init__(self, iban, balance=100.0):
        self.iban = iban
        self.balance = balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > self.balance:
            raise InsufficientBalance("balance is less than amount.", amount-self.balance)
        self.balance -= amount

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        self.balance += amount

    def __str__(self):
        return f"Account[ iban: {self.iban}, balance: {self.balance}]";
```

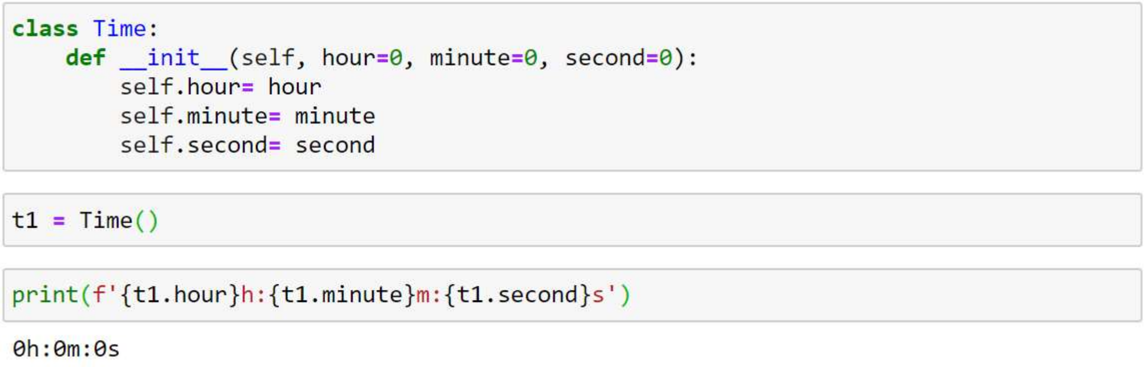
Defining a Class



```python
try:
    acc = Account("TR1", 100000.0)
    print(acc)
    acc.withdraw(2500.0)
    print(acc)
    acc.deposit(1500.0)
    print(acc)
    acc.withdraw(100000.0)
    print(acc)
except ValueError as err:
    print(err)
except InsufficientBalance as err:
    print(err)
```

```
Account[ iban: TR1, balance: 100000.0]
Account[ iban: TR1, balance: 97500.0]
Account[ iban: TR1, balance: 99000.0]
InsufficientBalance[ message: balance is less than amount., deficit: 1000.0]
```

`__init__` Method with Default Parameter Values



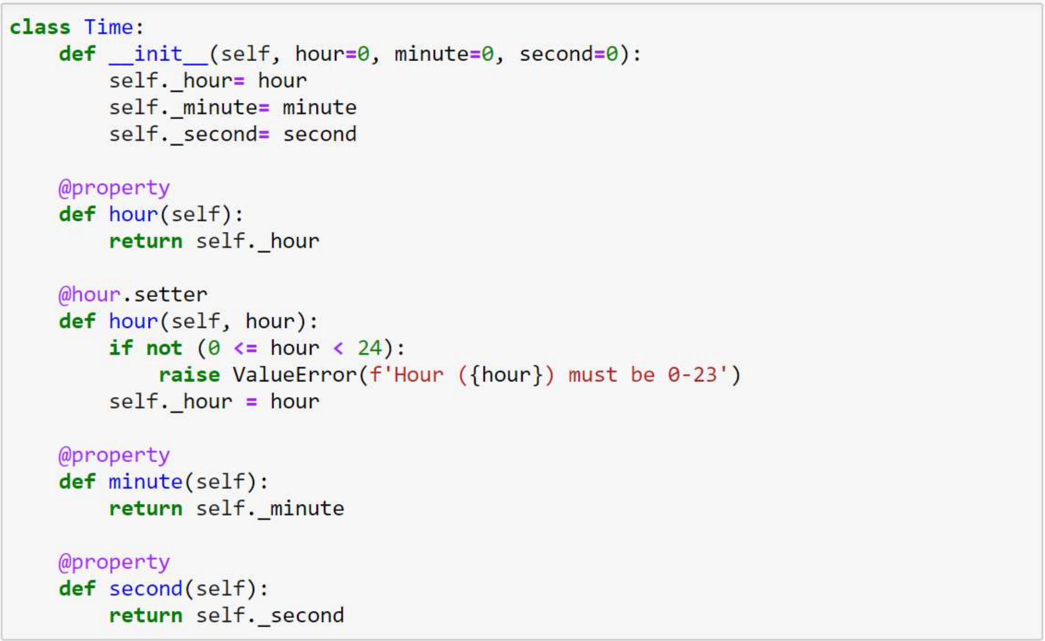
```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour= hour
        self.minute= minute
        self.second= second
```

```
t1 = Time()
```

```
print(f'{t1.hour}h:{t1.minute}m:{t1.second}s')
```

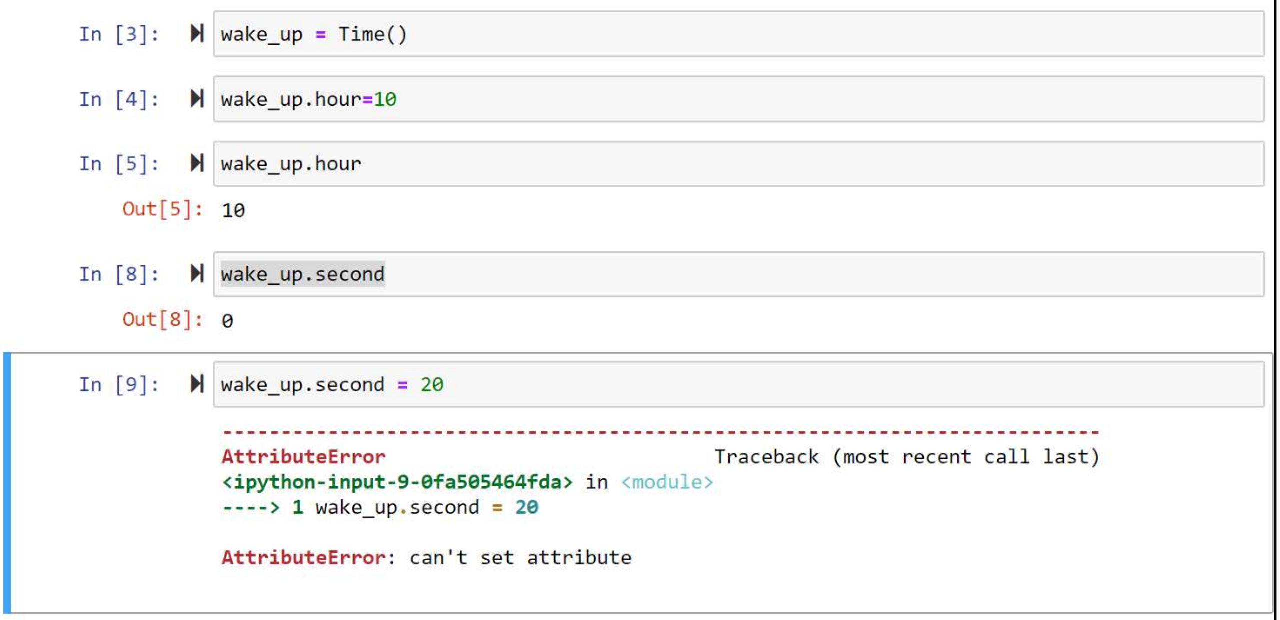
```
0h:0m:0s
```

Read-Write Property



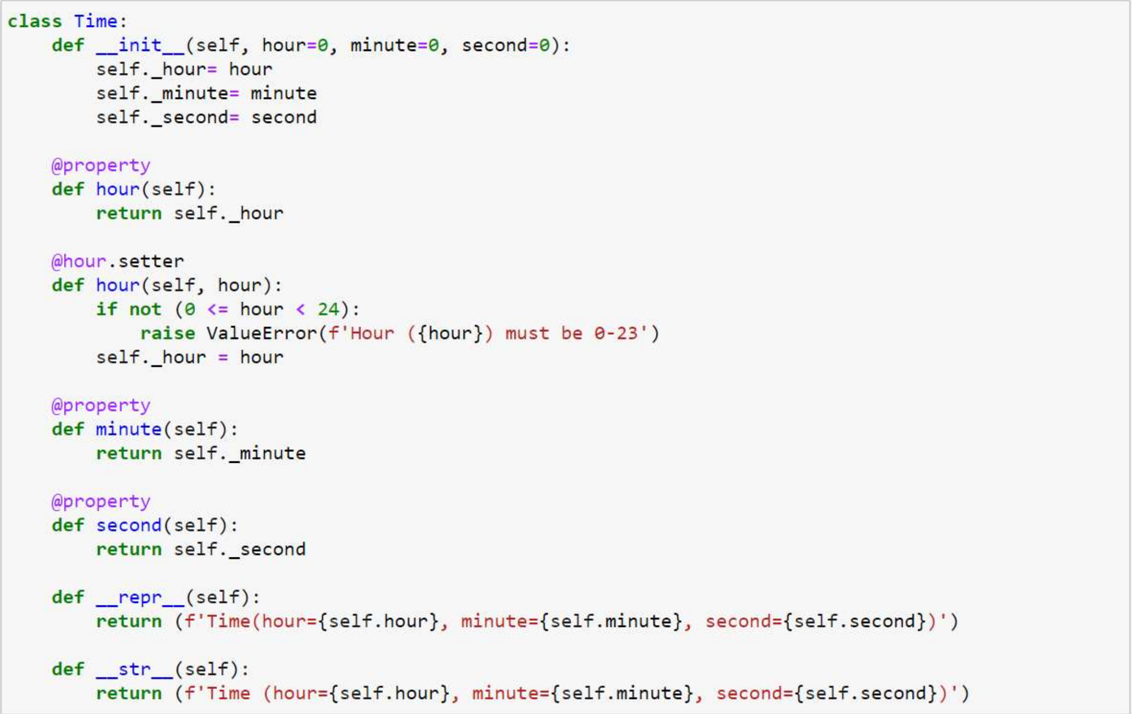
```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self._hour= hour
        self._minute= minute
        self._second= second

    @property
    def hour(self):
        return self._hour

    @hour.setter
    def hour(self, hour):
        if not (0 <= hour < 24):
            raise ValueError(f'Hour ({hour}) must be 0-23')
        self._hour = hour

    @property
    def minute(self):
        return self._minute

    @property
    def second(self):
        return self._second
```

Read-Write Property



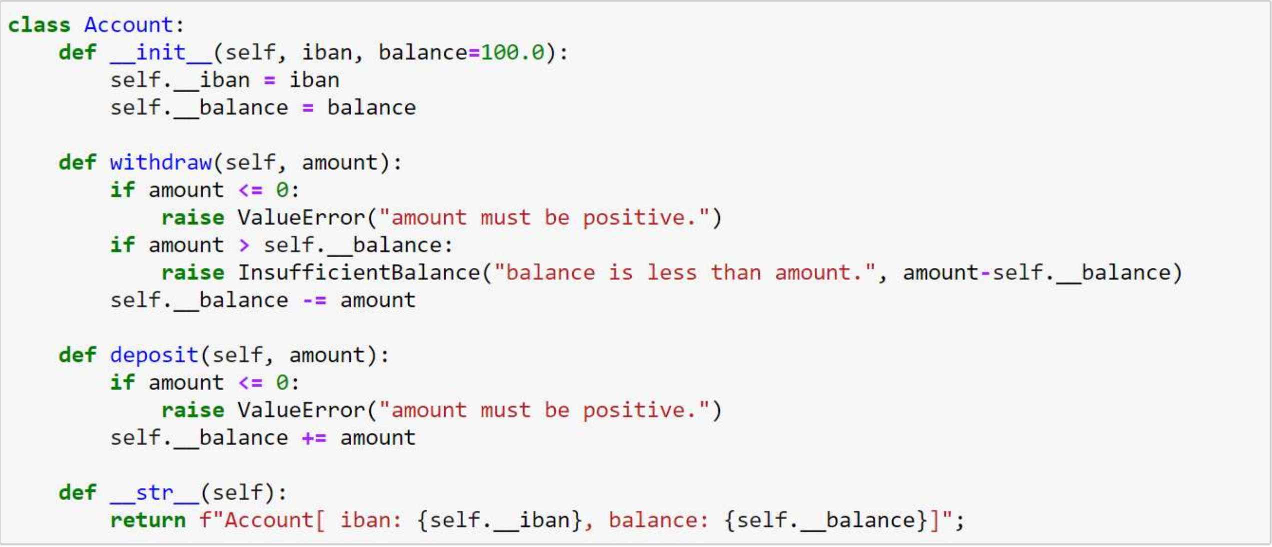
```
In [3]:  wake_up = Time()

In [4]:  wake_up.hour=10

In [5]:  wake_up.hour

Out[5]:  10

In [8]:  wake_up.second

Out[8]:  0

In [9]:  wake_up.second = 20

         ---------------------------------------------------------------------------
         AttributeError                            Traceback (most recent call last)
         <ipython-input-9-0fa505464fda> in <module>
         ----> 1 wake_up.second = 20

         AttributeError: can't set attribute
```

Special Method __repr__

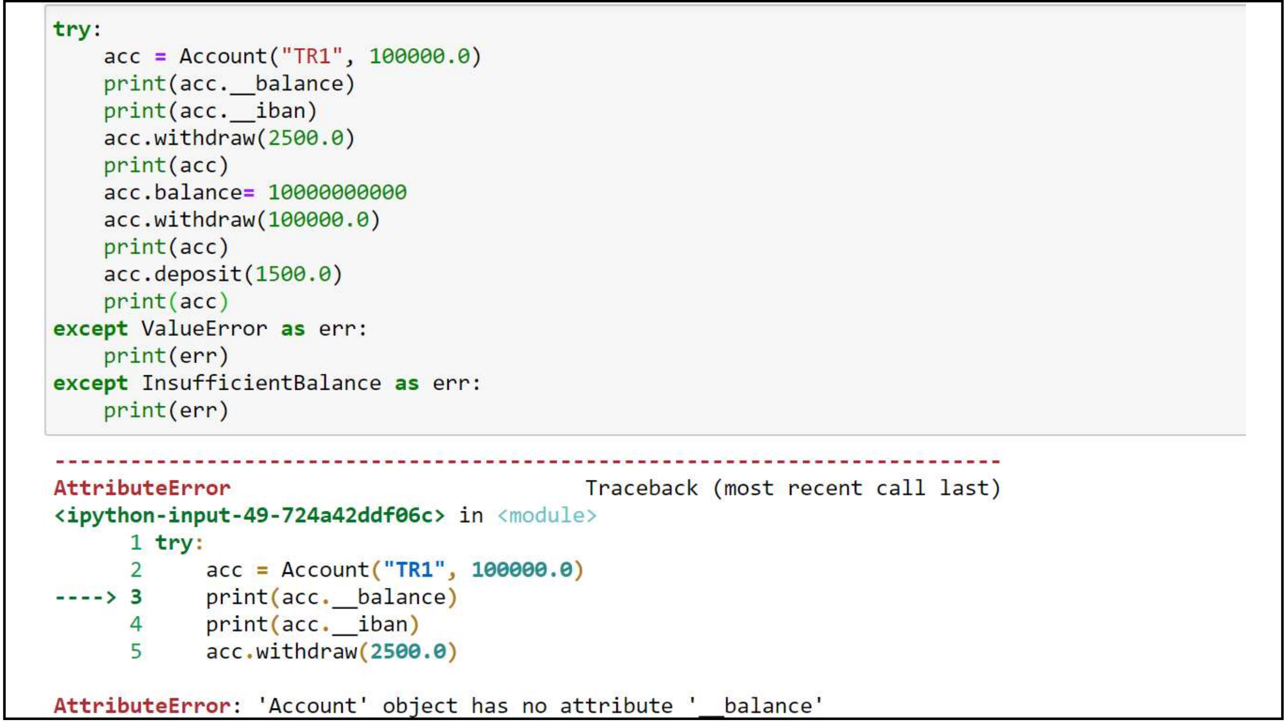


```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self._hour= hour
        self._minute= minute
        self._second= second

    @property
    def hour(self):
        return self._hour

    @hour.setter
    def hour(self, hour):
        if not (0 <= hour < 24):
            raise ValueError(f'Hour ({hour}) must be 0-23')
        self._hour = hour

    @property
    def minute(self):
        return self._minute

    @property
    def second(self):
        return self._second

    def __repr__(self):
        return (f'Time(hour={self.hour}, minute={self.minute}, second={self.second})')

    def __str__(self):
        return (f'Time (hour={self.hour}, minute={self.minute}, second={self.second})')
```

Information Hiding



```python
class Account:
    def __init__(self, iban, balance=100.0):
        self.__iban = iban
        self.__balance = balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > self.__balance:
            raise InsufficientBalance("balance is less than amount.", amount-self.__balance)
        self.__balance -= amount

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        self.__balance += amount

    def __str__(self):
        return f"Account[ iban: {self.__iban}, balance: {self.__balance}]";
```



```python
try:
    acc = Account("TR1", 100000.0)
    print(acc.__balance)
    print(acc.__iban)
    acc.withdraw(2500.0)
    print(acc)
    acc.balance= 10000000000
    acc.withdraw(100000.0)
    print(acc)
    acc.deposit(1500.0)
    print(acc)
except ValueError as err:
    print(err)
except InsufficientBalance as err:
    print(err)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-49-724a42ddf06c> in <module>
      1 try:
      2     acc = Account("TR1", 100000.0)
----> 3     print(acc.__balance)
      4     print(acc.__iban)
      5     acc.withdraw(2500.0)

AttributeError: 'Account' object has no attribute '__balance'
```

Inheritance

Inheritance is “a mechanism whereby a class is defined in reference to others, adding all their features to its own.” (Meyer page 1197)

Features of inheritance:

> Attributes and methods from the superclass are included in the subclass.

> Subclass methods can override superclass methods.

> The following conditions must be true for the inheritance relationship to be plausible:

– A subclass object *is a (is a kind of)* the superclass object.

– Inheritance should conform to Liskov’s Substitution Principle (LSP).

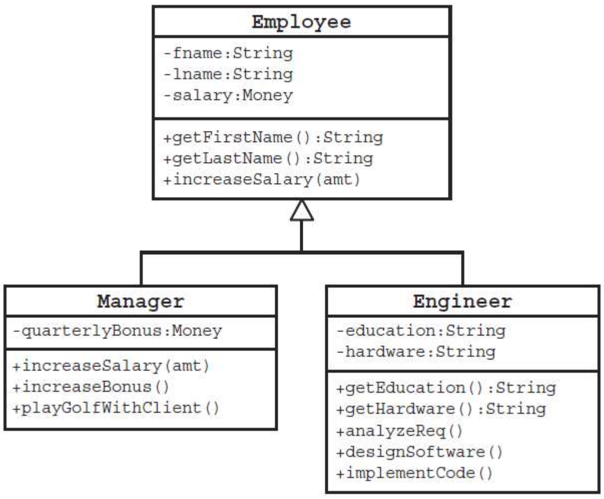
Inheritance

> Specific OO languages allow either of the following:

– Single inheritance, which allows a class to directly inherit from only one superclass (for example, Java).

– Multiple inheritance, which allows a class to directly inherit from one or more super-classes (for example, C++, Python).

Inheritance: Example



Employee

-fname:String
-lname:String
-salary:Money

+getFirstName():String
+getLastName():String
+increaseSalary(amt)

Manager

-quarterlyBonus:Money

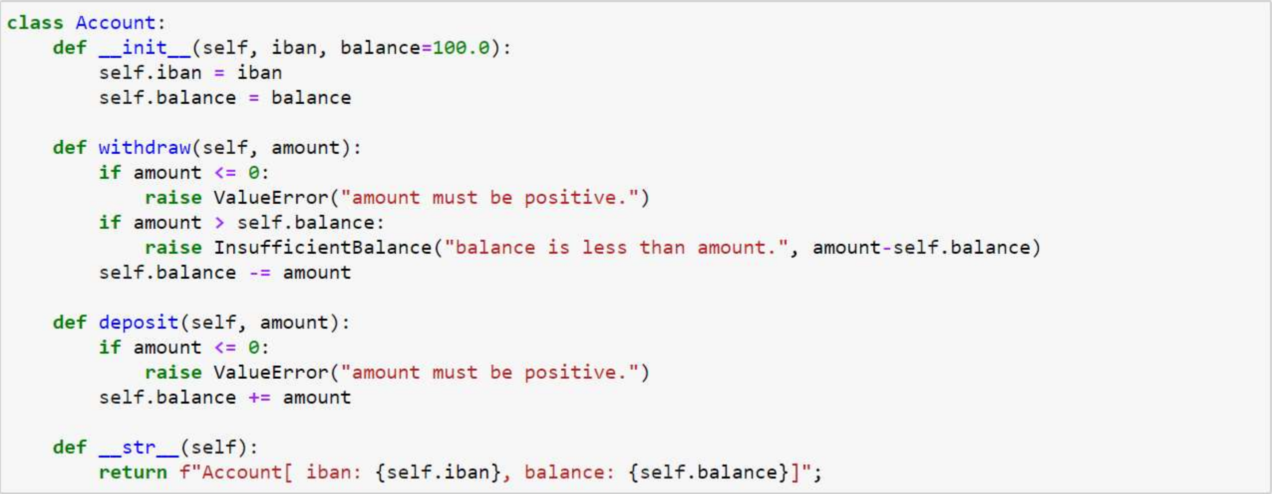
+increaseSalary(amt)
+increaseBonus()
+playGolfWithClient()

Engineer

-education:String
-hardware:String

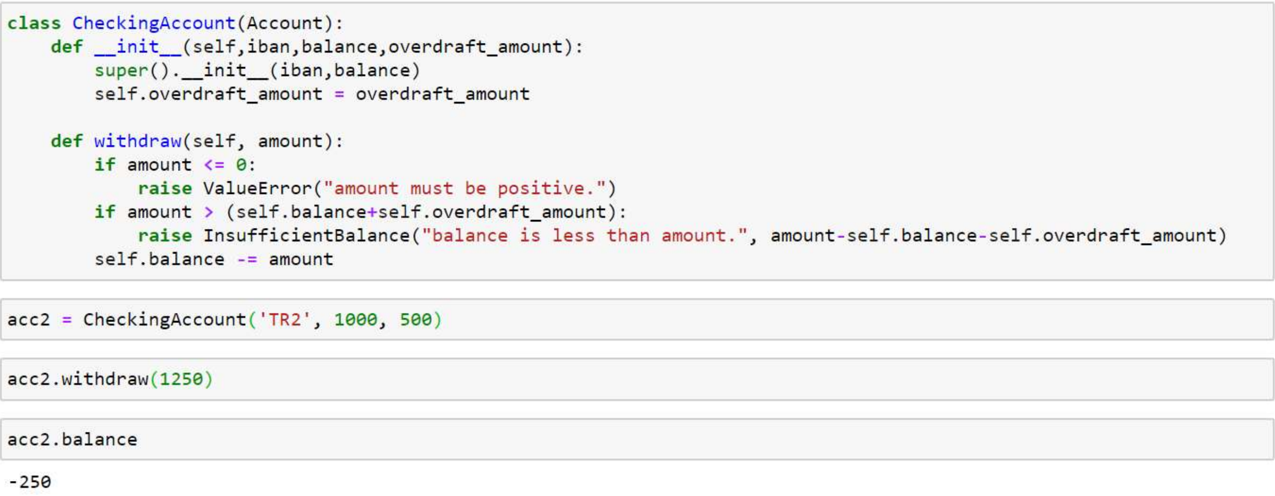
+getEducation():String
+getHardware():String
+analyzeReq()
+designSoftware()
+implementCode()

Inheritance: Example



```python
class Account:
    def __init__(self, iban, balance=100.0):
        self.iban = iban
        self.balance = balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > self.balance:
            raise InsufficientBalance("balance is less than amount.", amount-self.balance)
        self.balance -= amount

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        self.balance += amount

    def __str__(self):
        return f"Account[ iban: {self.iban}, balance: {self.balance}]";
```

Inheritance: Example



```python
class CheckingAccount(Account):
    def __init__(self,iban,balance,overdraft_amount):
        super().__init__(iban,balance)
        self.overdraft_amount = overdraft_amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > (self.balance+self.overdraft_amount):
            raise InsufficientBalance("balance is less than amount.", amount-self.balance-self.overdraft_amount)
        self.balance -= amount
```

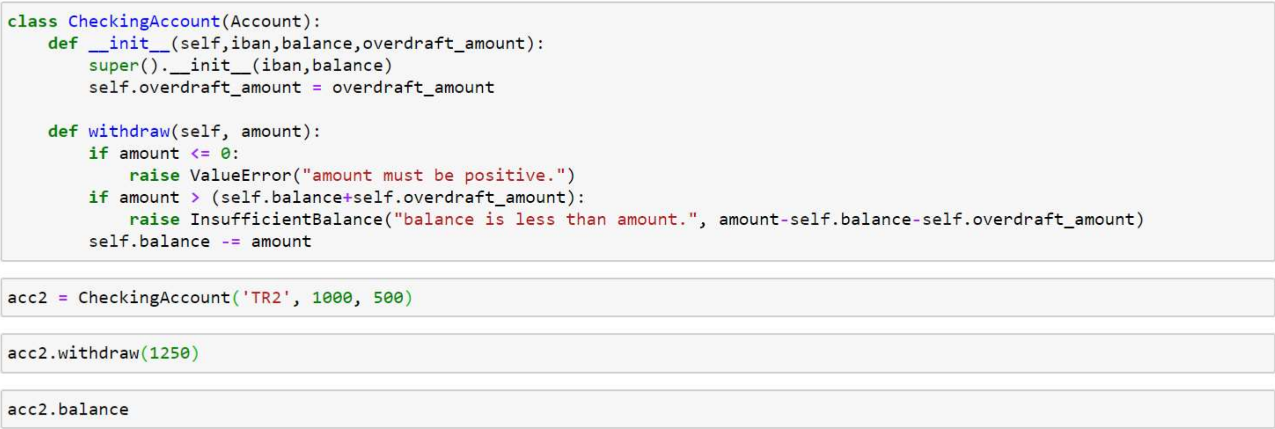
```python
acc2 = CheckingAccount('TR2', 1000, 500)
```

```python
acc2.withdraw(1250)
```

```python
acc2.balance
```

```
-250
```

Inheritance: Example



```python
class CheckingAccount(Account):
    def __init__(self,iban,balance,overdraft_amount):
        super().__init__(iban,balance)
        self.overdraft_amount = overdraft_amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > (self.balance+self.overdraft_amount):
            raise InsufficientBalance("balance is less than amount.", amount-self.balance-self.overdraft_amount)
        self.balance -= amount
```

```python
acc2 = CheckingAccount('TR2', 1000, 500)
```

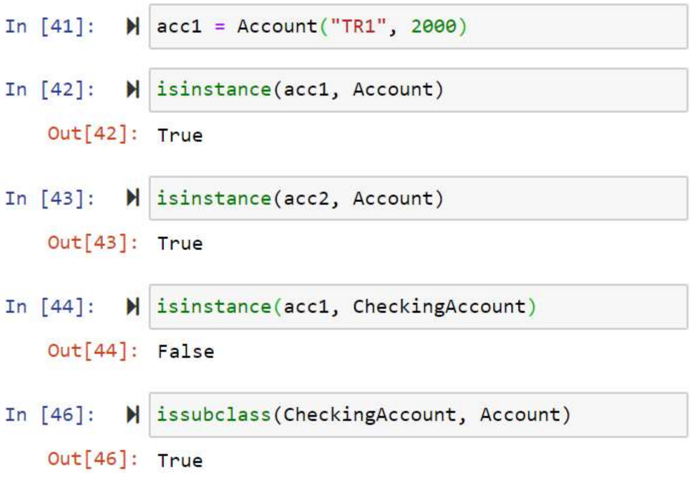
```python
acc2.withdraw(1250)
```

```python
acc2.balance
```

Testing the “is a” Relationship

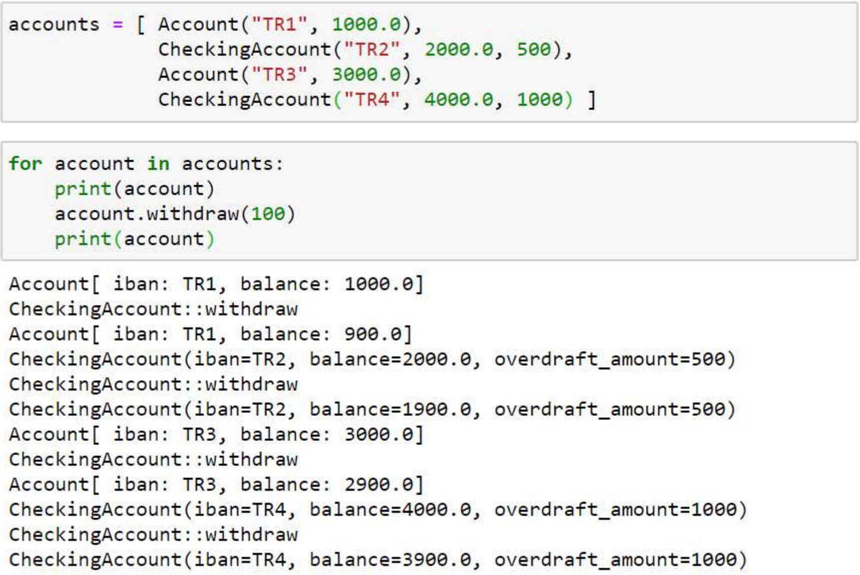
- Python provides two built-in functions—**`issubclass`** and **`isinstance`**—for testing “is a” relationships.
- Function **`issubclass`** determines whether one class is derived from another
- Function **`isinstance`** determines whether an object has an “is a” relationship with a specific type.

Testing the “is a” Relationship



```
In [41]:    acc1 = Account("TR1", 2000)

In [42]:    isinstance(acc1, Account)
   Out[42]: True

In [43]:    isinstance(acc2, Account)
   Out[43]: True

In [44]:    isinstance(acc1, CheckingAccount)
   Out[44]: False

In [46]:    issubclass(CheckingAccount, Account)
   Out[46]: True
```

Polymorphism



```
accounts = [ Account("TR1", 1000.0),
             CheckingAccount("TR2", 2000.0, 500),
             Account("TR3", 3000.0),
             CheckingAccount("TR4", 4000.0, 1000) ]
```

```
for account in accounts:
    print(account)
    account.withdraw(100)
    print(account)
```

```
Account[ iban: TR1, balance: 1000.0]
CheckingAccount::withdraw
Account[ iban: TR1, balance: 900.0]
CheckingAccount(iban=TR2, balance=2000.0, overdraft_amount=500)
CheckingAccount::withdraw
CheckingAccount(iban=TR2, balance=1900.0, overdraft_amount=500)
Account[ iban: TR3, balance: 3000.0]
CheckingAccount::withdraw
Account[ iban: TR3, balance: 2900.0]
CheckingAccount(iban=TR4, balance=4000.0, overdraft_amount=1000)
CheckingAccount::withdraw
CheckingAccount(iban=TR4, balance=3900.0, overdraft_amount=1000)
```

Operator Overloading

> Method-call notation can be cumbersome for certain kinds of operations, such as arithmetic.

> In these cases, it would be more convenient to use Python’s rich set of built-in operators.

Restrictions on operator overloading (1/2)

> There are some restrictions on operator overloading:

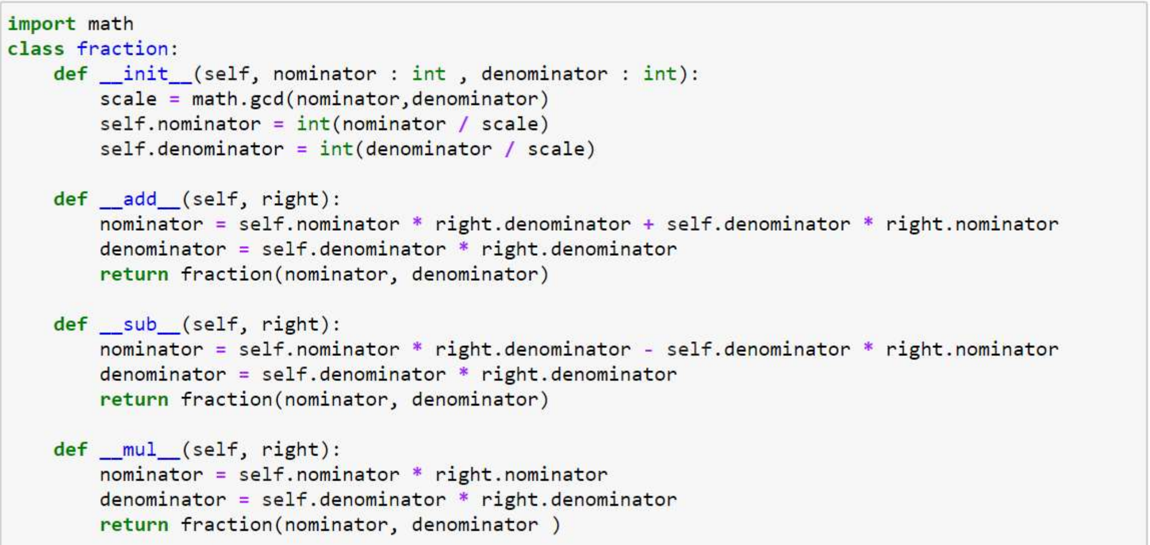
- The precedence of an operator cannot be changed by overloading.
  - However, parentheses can be used to force evaluation order in an expression.
- The left-to-right or right-to-left grouping of an operator cannot be changed by overloading.
- The “arity” of an operator—that is, whether it’s a unary or binary operator—cannot be changed.
- You cannot create new operators—only existing operators can be overloaded.
- The meaning of how an operator works on objects of built-in types cannot be changed. You cannot, for example, change + so that it subtracts two integers.

Restrictions on operator overloading (2/2)

> There are some restrictions on operator overloading:

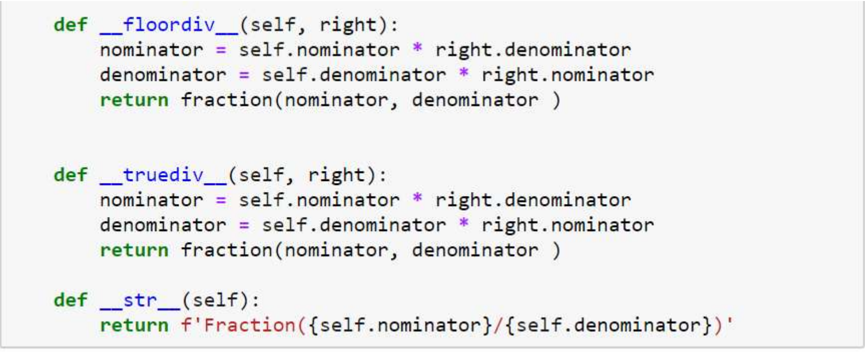
- Operator overloading works only with objects of custom classes or with a mixture of an object of a custom class and an object of a built-in type.

Operator Overloading Example: **`Fraction`**



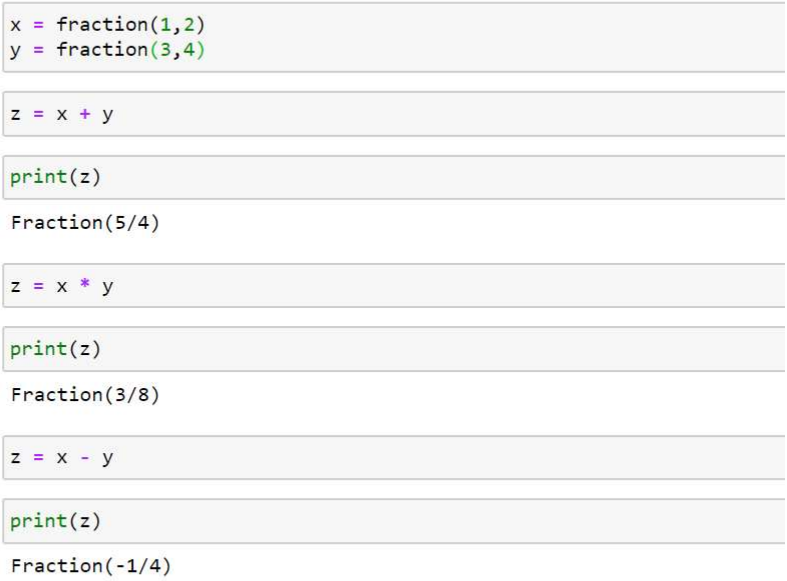
```python
import math
class fraction:
    def __init__(self, nominator : int , denominator : int):
        scale = math.gcd(nominator,denominator)
        self.nominator = int(nominator / scale)
        self.denominator = int(denominator / scale)

    def __add__(self, right):
        nominator = self.nominator * right.denominator + self.denominator * right.nominator
        denominator = self.denominator * right.denominator
        return fraction(nominator, denominator)

    def __sub__(self, right):
        nominator = self.nominator * right.denominator - self.denominator * right.nominator
        denominator = self.denominator * right.denominator
        return fraction(nominator, denominator)

    def __mul__(self, right):
        nominator = self.nominator * right.nominator
        denominator = self.denominator * right.denominator
        return fraction(nominator, denominator )
```

Operator Overloading Example: **`Fraction`**



```python
    def __floordiv__(self, right):
        nominator = self.nominator * right.denominator
        denominator = self.denominator * right.nominator
        return fraction(nominator, denominator )


    def __truediv__(self, right):
        nominator = self.nominator * right.denominator
        denominator = self.denominator * right.nominator
        return fraction(nominator, denominator )

    def __str__(self):
        return f'Fraction({self.nominator}/{self.denominator})'
```

Operator Overloading Example: **Fraction**



```
x = fraction(1,2)
y = fraction(3,4)
```

```
z = x + y
```

```
print(z)
```

```
Fraction(5/4)
```

```
z = x * y
```

```
print(z)
```

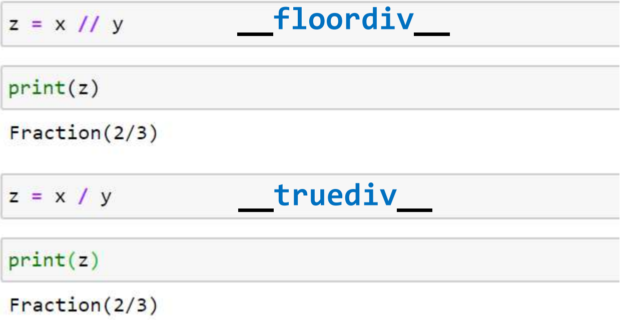
```
Fraction(3/8)
```

```
z = x - y
```

```
print(z)
```

```
Fraction(-1/4)
```

Operator Overloading Example: **Fraction**



```
z = x // y
```

__floordiv__

```
print(z)
```

```
Fraction(2/3)
```

```
z = x / y
```

__truediv__

```
print(z)
```

```
Fraction(2/3)
```