Розділ 1

Основи програмування

1.1 Об'єктно-орієнтоване програмування

Об'єктно-орієнтоване програмування (ООП) — парадигма програмування, в якій основними концепціями є поняття об'єктів і класів.

Клас — тип, що описує будову об'єктів. З точки зору ООП, клас представляє собою колекцію даних. Об'єкт — це екземпляр класу. Клас можна порівняти з кресленням, за яким створюються об'єкти.

Python відповідає принципам об'єктно-орієнтованого програмування. В python все є об'єктами — і стрічки, і списки, і словники, і все інше.

Але можливості ООП в python цим не обмежені. Програміст може написати свій тип даних (клас), визначити в ньому свої методи.

Це не є обов'язковим — ми можемо використовувати тільки вбудовані об'єкти. Проте ООП корисне при довгостроковій розробці програми декількома людьми, так як спрощує розуміння коду.

Використання класів дає нам перш за все переваги абстрактного підходу в програмуванні.

1. **Поліморфізм:** в різних об'єктах одна й та ж операція може виконувати різні функції. Слово "поліморфізм" має грецьку природу і означає "той що має багато форм". Простим прикладом поліморфізму може служити функція count(), яка виконує однакові дії для різних типів об'єктів: `'abc'.count('a')` і `[1, 2, 'a'].count('a')`. Оператор плюс поліморфний при додаванні чисел і при додаванні стрічок.
2. **Інкапсуляція:** можна приховувати непотрібні внутрішні деталі роботи об'єкта від оточуючого світу. Це другий основний принцип абстракції. Він вазується на використанні атрибутів в середині класу. Атрибути можуть мати різні стани в проміжках між викликами методів класу, в наслідок чого сам об'єкт даного класу також отримує різні стани — state.
3. **Успадкування:** можна створювати спеціалізовані класи на основі базових. Це дозволяє нам уникати написання повторного коду.
4. **Композиція:** об'єкт може бути складеним і включати в себе інші об'єкти.

Об'єктно-орієнтований підхід в програмуванні передбачає наступний алгоритм дій.

1. Описується проблема за допомогою природної мови з використанням понять, дій, прикметників.
2. На основі понять формулюються класи.
3. На основі дій проектуються методи.
4. Реалізуються методи і атрибути.

Ми отримуємо скелет — об'єктну модель. На основі цієї моделі реалізується успадкування. Для перевірки моделі:

- пишеться так званий use cases — сценарій можливої поведінки моделі, де перевіряється вся функціональність;

• функционал при цьому може бути виправлений або добавлений.

Объектно-ориентированный подход хорош там, где проект подразумевает долгосрочное развитие, состоит из большого количества библиотек и внутренних связей.

Механізм класів мови Python представляє собою суміш механізмів класів C++ і Modula-3. Найбільш важливі особливості класів в Python:

- множинне успадкування;
- похідний клас може перевизначати будь-які методи базових класів;
- в будь-якому місці можна викликати метод з тим же іменем базового класу;
- усі атрибути класу в Python за замовчуванням є public, тобто доступні звідусіль; всі методи — віртуальні, тобто перевантажують базові.

1.1.1 Поняття класу

Клас — це тип користувача. Найпростіша модель визначення класу має наступний вигляд:

```
class ім'я_класу:
   інструкція 1
   ...
   інструкція N
```

Кожен такий запис генерує свій об'єкт класу. Відміна від C++ в тому, що в плюсах опис класу — це лише оголошення, а в Python — це створення об'єкту. Є також інший тип об'єкту — екземпляр класу, який генерується при виклику:

```
екземпляр_класу = ім'я_класу()
```

Об'єкт класу і екземпляр класу — це два різних об'єкти. Перший генерується на етапі оголошення класу, другий — при виклику імені класу. Об'єкт класу може бути один, екземплярів класу може бути скільки завгодно.

Інструкція — це, як правило, визначені функції. При визначенні класу створюється новий простір імен і створюється об'єкт-клас, який є оболонкою для всіх інструкцій.

Об'єкти класів підтримують два види операцій:

- доступ до атрибутів;
- створення екземпляру класу.

1.1.2 Атрибути класу

Атрибути класу бувають двох видів:

- атрибути даних;
- атрибути-методи.

Атрибути даних звичайно записуються зверху. Пам'ять для атрибутів виділяється в момент їх першого присвоєння — або ззовні, або в середині методу. Методи починаються з службового слова def.

Доступ до атрибутів виконується за схемою `obj.attrname`.

Приклад.

```
class Simple:
   '''Простий клас'''
   var = 87
   def f(x):
      return 'Hello world'
```

Тут `Simple.var` і `Simple.f` — атрибути користувача. Є також стандартні атрибути:

```
>>> print Simple.__doc__
```

Ціле число

```
>>> print Simple.var.__doc__
int(x=0) -> integer
int(x, base=10) -> integer
...
```

Створення екземпляра класу подібне на те, як ніби роблять виклик функцій:

```
smpl = Simple()
```

Буде створено пустий об'єкт smpl. Якщо ми хочемо, щоб при створенні виконувалися якісь дії, потрібно визначити конструктор, який буде викликатися автоматично:

```
class Simple:
   def __init__(self):
      self.list = []
```

При створенні об'єкта smpl буде створений пустий список list. Конструктору можна передати аргументи:

```
class Simple:
   def __init__(self, count, str):
      self.list = []
      self.count = count
      self.str = str

>>> s = Simple(1, '22')
>>> s.count, s.str
1 22
```

Атрибут даних можна зробити приватним (private) — тобто недоступним ззовні — для цього зліва потрібно поставити два символи підкреслення:

```
class Simple:
   '"Простий клас з приватним атрибутом"'
   __private_attr = 10
   def __init__(self, count, str):
      self.__private_attr = 20
      print self.__private_attr

s = Simple(1,'22')
print s.__private_attr
```

Остання стрічка викличе виняток — атрибут `__private_attr` підходить тільки для внутрішнього використання.

Методи необов'язково визначати в середині тіла класу:

```
def method_for_simple(self, x, y):
   return x + y

class Simple:
   f = method_for_simple
```

```
>>> s = Simple()
>>> print s.f(1,2)
3
```

Пустий клас можна використовувати як заготовку для структури даних:

```
class Customer:
   pass

custom = Customer()
custom.name = 'Вася'
custom.salary = 100000
```

1.1.3 self

Звичайно перший аргумент в імені методу — self. Як говорить автор мови Гвідо Ван Россум, це не більше ніж угода: ім'я self не має абсолютно жодного спеціального значення.

self корисний для того, щоб звертатися до інших атрибутів класу:

```
class Simple:
   def __init__(self):
      self.list = []
   def f1(self):
      self.list.append(123)
   def f2(self):
      self.f1()

>>> s = Simple()
>>> s.f2()
>>> print s.list
[123]
```

Self — це аналог "this" в C++.

1.1.4 Успадкування

Визначення довільного класу має наступний вигляд:

```
class Derived(Base):
   pass
```

Якщо базовий клас визначений не в поточному модулі:

```
class Derived(module_name.Base):
pass
```

Вирішення імен атрибутів працює зверху до низу: якщо атрибут не знайдено в поточному класі, пошук продовжується в базовому класі, і так далі за рекурсією. Похідні класи можуть перевизначати методи базових класів — всі методи є в цьому змісті віртуальними. Викликати метод базового класу можна з префіксом Base.method()

В Python існує обмежена підтримка множинного успадкування:

```
class Derived(Base1, Base2, Base3):
  pass
```

Пошук атрибуту проводиться в наступному порядку:

1. в Derived;
2. в Base1, потім рекурсивно в базових класах Base1;
3. в Base2, потім рекурсивно в базових класах Base2
4. і т.д.

1.1.5 Приклад оголошення та використання класу

Дуже поширеним прикладом для демонстрації деталей реалізації класу користувача є розробка класу, який втілює абстрактний тип даних Fraction, який представляє правильні дроби.

Дріб складається з двох частин. Верхнє значення, чисельник, може бути будь-яким цілим числом. Нижнє значення (знаменник) — будь-яким цілим, більшим нуля (від'ємні дроби мають від'ємний чисельник). Також для будь-якого дробу можна створити наближення з плаваючою комою. У цьому випадку ми хотіли б представляти дріб як точне значення.

Операції для типу Fraction дозволятимуть його об'єктам даних поводитися подібно будь-яким іншим числовим значенням. Ми повинні бути готові додавати, віднімати, множити і ділити дроби. Також необхідна можливість показувати дробу в їх стандартній «слеш»-формі. Всі методи дробів повинні повертати результат у своїй скороченій формі таким чином, щоб, незалежно від виду обчислень, в кінці ми завжди мали найбільш загальноприйняту форму.

У Python ми визначаємо новий клас наданням його імені і набору визначень методів, які синтаксично подібне визначення функцій. У цьому прикладі

```
class Fraction:
  '"Клас простих дробів"'
  pass
```

нам дано каркас для визначення методів.

Відразу після оголошення об'єкти-класи мають наступні атрибути:

- `__name__` — ім'я класу;
- `__module__` — ім'я модуля;
- `__dict__` — словник атрибутів класу, можна змінювати цей словник напряму;
- `__bases__` — кортеж базових класів в порядку їх слідування;
- `__doc__` — стрічка документації класу.

Для створення екземпляру класу потрібно присвоїти змінній назву класу:

```
frac = Fraction()
```

Екземпляр класу повертається при виклику об'єкта-класу. Об'єкт в класу може буть один, екземплярів — декілька. Екземпляри мають наступні атрибути:

- `__dict__` — словник атрибутів класу, можна змінювати цей словник напряму;
- `__class__` — об'єкт-клас, екземпляром якого є даний екземпляр;
- `__init__` — конструктор. Якщо в базовому класі є конструктор, конструктор похідного класу повинен викликати його;
- `__del__` — деструктор. Якщо в базовому класі є деструкор, деструктор похідного класу повинен викликати його;
- `__cmp__` — викликається для всіх операцій порівняння;
- `__hash__` — повертає хеш-значення об'єкта, яке рівне 32-бітному числу;
- `__getattr__` — повертає атрибут, недоступний звичайним способом;
- `__setattr__` — присвоює значення атрибуту;

- __delattr__ — видаляє атрибут;
- __call__ — спрацьовує при виклику екземпляру класу.

Почнемо визначати деталі класу для того, щоб його можна було конструктивно використовувати.

Першим з них (його повинні надавати всі класи) є конструктор. Він визначає спосіб створення об'єкта даних. Щоб створити об'єкт Fraction, нам потрібно надати два елементи даних — чисельник і знаменник. У Python метод конструктора завжди називається __init__ (по два підкреслення до і після init).

```
class Fraction:
  def __init__(self,top,bottom):
    self.num = top
    self.den = bottom
```

Зверніть увагу, що список формальних параметрів містить три елементи (self, top, bottom). self — це спеціальний параметр, який використовується, як зворотнє посилання на сам об'єкт. Він завжди повинен бути першим формальним параметром, однак, при виклику конструктора в нього ніколи не передається актуальне значення. Дробам потрібні дані для двох частин — чисельника і знаменника. Нотація self.num конструктора визначає, що об'єкт fraction має внутрішній об'єкт даних, іменований num, як частину свого стану. Аналогічно, self.den створює знаменник. Значення цих двох формальних параметрів спочатку встановлюються в стан, що дозволяє новому об'єкту fraction знати своє початкове значення.

Щоб створити екземпляр класу Fraction, ми повинні викликати конструктор. Це відбудеться при використанні імені класу з підстановкою актуальних значень в необхідний стан (ми ніколи не викликаємо безпосередньо __init__). Наприклад,

```
myfraction = Fraction(3,5)
```

створює екземпляр об'єкту з ім'ям myfraction, що представляє дріб три п'ятих.

Наступне, чим ми займемося, це реалізація поведінки, необхідної кожному класу. Для початку розглянемо, що відбувається, коли ми намагаємося надрукувати об'єкт Fraction.

```
print(myfraction)
<__main__.Fraction instance at 0x409b1acc>
```

Об'єкт myfraction не знає, як йому відповідати на запит про друк. Функція print вимагає, щоб об'єкт конвертував самого себе в рядок, яка буде записана на виході. Єдиний вибір, який має myfraction, — це показати актуальне посилання, що зберігається в змінній (безпосередня адреса). Це явно не те, що ми хочемо.

Існує два шляхи вирішення цієї проблеми. Перший — визначити метод під назвою show, який дозволить об'єкту Fraction друкувати самого себе як стрічку. Ми можемо реалізувати його.

```
  def show(self):
    print(self.num,"/",self.den)
```

Якщо створювати об'єкт Fraction як і раніше, то можна попросити його показати себе (іншими словами, надрукувати себе) в відповідному форматі. На жаль, в загальному випадку це не буде працювати. Для того, щоб організувати друк належним чином, нам необхідно повідомити класу Fraction, як йому конвертувати себе в рядок. Це те, що необхідно функції print для нормальної роботи.

Об'єкти класів можна привести до стрічкового або числового типу.

- __repr__ — повертає формальне стрічкове представлення об'єкта;
- __str__ — повертає стрічкове представлення об'єкта;

- __oct__ , __hex__ , __complex__ , __int__ , __long__ , __float__ — повертають стрічкове представлення у відповідній системі числення.

Одже, __str__ — метод перетворення об'єкту в стрічку. Реалізація за замовчуванням для цього методу, як ми вже могли бачити, повертає стрічку адреси екземпляра класу. Що нам необхідно зробити, так це надати для нього «кращу» реалізацію. Ми будемо говорити, що вона перевантажує попередню (або перевизначає поведінку методу).

Для цього просто визначають метод з ім'ям __str__ і задамо йому нову реалізацію. Це визначення не потребує ніякої додаткової інформації, крім спеціального параметра self. У свою чергу, метод буде створювати стрічкове представлення конвертацією кожного шматочка внутрішніх даних стану в рядок і конкатенації цих рядків за допомогою символу / між ними. Результуюча стрічка буде повертатися щоразу, як об'єкт Fraction попросить перетворити себе в рядок.

```
def __str__(self):
   return "{}/{}".format(self.num, self.den)
```

Ми можемо перезавантажити безліч інших методів для нашого нового класу Fraction. Одними з найбільш важливих з них для дробів є основні арифметичні операції. Ми хотіли б мати можливість створити два об'єкти Fraction, а потім додати їх разом, використовуючи стандартну запис «+».

На даний момент, додаючи два дроби, ми отримуємо наступне:

```
>>> f1 = Fraction(1,4)
>>> f2 = Fraction(1,2)
>>> f1+f2
Traceback (most recent call last):
  File "<pyshell#173>", line 1, in -toplevel-
   f1+f2
TypeError: unsupported operand type(s) for +:
        'instance' and 'instance'
```

Якщо уважно подивитися на повідомлення про помилку, то можна побачити, що оператор "+" не розуміє операндів Fraction.

Можна виправити це, надавши класу Fraction метод, який перевантажує додавання. У Python він називається __add__ і приймає два параметри. Перший — self — необхідний завжди, другий представляє з себе другий операнд виразу. Наприклад, f1.__add__(f2) буде запитувати у Fraction об'єкта f1 додати до нього Fraction об'єкт f2. Це може бути записано і в стандартній нотації f1 + f2.

Для того, щоб додати два дроби, їх потрібно привести до спільного знаменника. Найпростіший спосіб переконатися, що у них однаковий знаменник, — це використовувати в його якості добуток знаменників дробів. Тобто $\frac{a}{b} + \frac{c}{d} = \frac{ad}{bd} + \frac{cb}{bd} = \frac{ad+bc}{bd}$. Функція додавання повертає новий об'єкт Fraction з чисельником і знаменником результуючого дробу. Ми можемо використовувати цей метод при написанні стандартних арифметичних виразів з дробами, присвоюючи результату сумарний дріб і виводячи його на екран.

```
  def __add__(self,otherfraction):
      newnum = self.num*otherfraction.den + \
               self.den*otherfraction.num
      newden = self.den * otherfraction.den
      return Fraction(newnum,newden)

>>> f1=Fraction(1,4)
>>> f2=Fraction(1,2)
>>> f3=f1+f2
```

```
>>> print(f3)
6/8
```

Метод додавання працює, як ми того й хотіли, але одну річ можна було б поліпшити. Зауважте, що 6/8 — це правильний результат обчислення (1/4 + 1/2), але це не скорочена форма. Кращим поданням буде 3/4. Для того, щоб бути впевненими, що результат завжди має скорочений вигляд, нам знадобиться допоміжна функція, яка вміє скорочувати дроби. У ній потрібно буде знаходити найбільший спільний дільник, або НСД. Потім ми зможемо розділити чисельник і знаменник на НСД, а результат і буде скороченням до найменших членів.

Найбільш відомий алгоритм знаходження найбільшого спільного дільника — це алгоритм Евкліда. Він встановлює, що найбільшим спільним дільником двох чисел m і n буде n, якщо m ділиться на n без остачі. Однак, якщо цього не відбувається, то відповіддю буде НСД n і залишок ділення m на n. Ми просто надамо тут ітеративну реалізацію цього алгоритму. Зверніть увагу, що вона працює тільки при додатному знаменнику. Це допустимо для нашого класу дробів, оскільки ми говорили, що від'ємні дроби будуть представлятися від'ємним чисельником.

```
def gcd(m,n):
    while m%n != 0:
        oldm = m
        oldn = n
        m = oldn
        n = oldm%oldn
    return n
```

Тепер можна використовувати цю функцію для скорочення будь-якого дробу. Щоб представити дріб у скороченому вигляді, ми будемо ділити чисельник і знаменник на їх найбільший спільний дільник. Отже, для дробу: 6/8 НСД дорівнює 2. Розділивши верх і низ на 2, ми отримаємо новий дріб: 3/4.

```
def __add__(self,otherfraction):
    newnum = self.num*otherfraction.den + self.den*otherfraction.num
    newden = self.den * otherfraction.den
    common = gcd(newnum,newden)
    return Fraction(newnum//common,newden//common)

>>> f1=Fraction(1,4)
>>> f2=Fraction(1,2)
>>> f3=f1+f2
>>> print(f3)
3/4
```

Зараз наш об'єкт Fraction має два дуже корисних методи. Група додаткових методів, які нам знадобиться включити в клас Fraction, містить спосіб порівнювати два дроби. Припустимо, що у нас є два об'єкти Fraction f1 і f2. f1 == f2 буде істиною, якщо вони посилаються на один і той же об'єкт. Два різних об'єкти з однаковими чисельника і знаменниками в цій реалізації рівні не будуть. Це називається поверхневою рівністю.

Ми можемо створити глибоку рівність — за однаковим значенням, а не за однаковим посиланням — перевантажуючи метод __eq__. Це ще один стандартний метод, доступний в будь-якому класі. Він порівнює два об'єкти і повертає True, якщо їх значення рівні, або False в іншому випадку.

У класі Fraction ми можемо реалізувати метод __eq__, знову представивши обидва дроби у вигляді з однаковим знаменником і потім порівнявши їх чисельники. Тут також важливо відзначити інші оператори відношень, які можуть бути перевантажені. Наприклад, метод __le__ надає функціонал "менше або дорівнює".

```
    def __eq__(self, other):
        firstnum = self.num * other.den
        secondnum = other.num * self.den

        return firstnum == secondnum
```

Повністю клас Fraction, реалізований на даний момент, показаний нижче. Залишимо іншу арифметику і методи відношень в якості вправ.

```
def gcd(m,n):
    while m % n != 0:
        oldm = m
        oldn = n
        m = oldn
        n = oldm % oldn
    return n

class Fraction:
    def __init__(self,top,bottom):
        self.num = top
        self.den = bottom

    def __str__(self):
        return return "{}/{}".format(self.num, self.den)

    def __add__(self,otherfraction):
        newnum = self.num*otherfraction.den + \
                 self.den*otherfraction.num
        newden = self.den * otherfraction.den
        common = gcd(newnum,newden)
        return Fraction(newnum//common,newden//common)

    def __eq__(self, other):
        firstnum = self.num * other.den
        secondnum = other.num * self.den

        return firstnum == secondnum

x = Fraction(1,2)
y = Fraction(2,3)
print(x+y)
print(x == y)
```

Щоб переконатися, що ви розумієте, як в класах Python реалізовуються оператори і як коректно писати методи, напишіть реалізацію операцій *, / і -. Також реалізуйте оператори порівняння > і <.

1.1.6 Приклад логічних схем

У наступному прикладі розглянемо інший важливий аспект об'єктно-орієнтованого програмування. Спадкування — це здатність одного класу бути пов'язаним з іншим класом подібно до

того, як бувають пов'язані між собою люди. Діти успадковують риси своїх батьків. Аналогічно, в Python клас-нащадок успадковує характеристики даних і поведінки від класу-предка. Такі класи часто називають субкласами і суперкласами, відповідно.

Вбудовані колекції Python мають взаємини між собою. Такого виду структуру відносин називають ієрархією успадкування. Наприклад, список є нащадком колекцій з послідовним доступом. У даному випадку ми назвемо список «спадкоємцем», а колекцію — «батьком» (або список — субкласом, колекцію — суперкласом). Така залежність часто називається відношенням IS-A (список є (is a) колекцією з послідовним доступом). Це передбачає, що списки успадковують найважливіші характеристики колекцій, зокрема — впорядкування вихідних даних, і такі операції, як конкатенація, повторення та індексація.

І списки, і кортежі, і стрічки представляють із себе колекції з послідовним доступом, успадковуючи загальну організацію даних і операції. Однак, вони різні за гомогенністю даних і мутабельністю наборів. Усі нащадки успадковують своїх батьків, але різняться між собою включенням додаткових характеристик.

Організовуючи класи в ієрархічному порядку, об'єктно-орієнтовані мови програмування дозволяють розширювати раніше написаний код під знову виникаючі потреби. На додаток, організовуючи дані в ієрархічній манері, ми краще розуміємо існуючі між ними взаємини. Ми можемо створювати більш ефективне абстрактне представлення.

Щоб глибше дослідити цю ідею, ми напишемо симуляцію — додаток, що симулює цифрові схеми. Її основними будівельними блоками будуть логічні елементи. Ці електронні перемикачі являють собою співвідношення алгебри логіки між їх входом і виходом. У загальному випадку вентилі мають єдину лінію виходу. Значення на ній залежить від значень, що подаються на вхідні лінії.

Вентиль "І" (AND) має два входи, на кожен з яких може подаватися нуль або одиниця (кодування False або True, відповідно). Якщо на обидва входи подана одиниця, то значення на виході теж 1. Однак, якщо хоча б один з входів встановлений в нуль, то результатом буде 0. Вентиль "АБО" також має два входи і видає одиницю, якщо хоча б на одному з них 1. У випадку, коли обидві вхідні лінії в нулі, результат теж 0.

Вентиль "НЕ" (NOT) відрізняється від попередніх тим, що має всього один вхід. Значення на виході буде просто зворотним вхідному значенню. Тобто, якщо на вході 0, то на виході 1, і навпаки.

Кожен з цих логічних елементів має свою таблицю істинності значень, яка відображатиме відображення вентилем входу на вихід.

and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

not	0
0	1
1	0

Комбінуючи ці вентилі в різні структури і застосовуючи до отриманого набори вхідних комбінацій, ми можемо будувати схеми, що володіють різними логічними функціями. Нехай є схема, що складається з двох вентилів "І", одного вентиля "АБО" і одного вентиля "НЕ". Виходи елементів "І" підключені безпосередньо до входів елемента "АБО", а його результуючий вихід — до входу вентиля "НЕ". Якщо ми будемо подавати набір вхідних значень на чотири вхідні лінії (по дві на кожен елемент "І"), то вони будуть оброблені, і результат з'явиться на виході вентиля "НЕ".

Поставивши собі за мету втілити цю схему, ми насамперед повинні створити представлення для логічних вентилів. Їх легко організувати, як клас зі спадковою ієрархією. Верхній клас LogicGate представляє найбільш загальні характеристики логічних елементів: зокрема, мітку вен-

тиля і лінію виходу. Наступний рівень субкласів розбиває логічні елементи на два сімейства: мають один вхід і мають два входи. Нижче вже з'являються конкретні логічні функції для кожного вентиля.

Тепер можна зайнятися реалізацією класів, починаючи з найбільш загального — LogicGate. Як вже зазначалося раніше, кожен вентиль має позначку для ідентифікації і єдину лінію виходу. На додаток, нам будуть потрібні методи, що дозволяють користувачеві запитувати у вентиля його мітку.

Наступним аспектом поведінки, якого потребує будь-який вентиль, є необхідність знати його вихідне значення. Це потрібно для виконання вентилями відповідних алгоритмів, заснованих на поточних значеннях на входах. Для генерації вихідного значення логічним елементам необхідно конкретне знання логіки їх роботи. Це передбачає виклик методу, який виконує логічні обчислення.

```
class LogicGate:
   '''Узагальнений логічний елемент'''
   def __init__(self,n):
      self.label = n
      self.output = None

   def getLabel(self):
      return self.label

   def getOutput(self):
      self.output = self.performGateLogic()
      return self.output
```

На даний момент ми не будемо реалізовувати функцію performGateLogic. Причина в тому, що ми не знаємо, як працюватимуть логічні операції у кожного вентиля. Ці деталі ми включимо для кожного доданого в ієрархію елемента індивідуально. Це дуже потужна ідея об'єктно-орієнтованого програмування: ми пишемо метод, який буде використовувати ще не існуючий код. Параметр self є посиланням на актуальний вентиль, що викликає метод. Будь-які додані в ієрархію логічні елементи просто потребуватимуть власної реалізації функції performGateLogic, яка стане використовуватися в потрібний момент. Після цього вентилі повинні надати своє вихідне значення. Ця можливість розширювати існуючу ієрархію і забезпечувати необхідні для її нового класу функції надзвичайно важлива для повторного використання існуючого коду.

Ми розділили логічні елементи, ґрунтуючись на кількості їх вхідних ліній. У вентиля "І" їх дві, як і у вентиля "АБО", а у вентиля "НЕ" — одна. Клас BinaryGate буде субкласом LogicGate і включить в себе елементи з двома вхідними лініями. Клас UnaryGate також буде субкласом LogicGate, але вхідна лінія у його елементів буде одна. У конструюванні комп'ютерних схем такі лінії іноді називають "пінами", так що ми будемо використовувати цю термінологію і в нашій реалізації.

```
class BinaryGate(LogicGate):
   '''Клас логічних елементів з двома входами'''

   def __init__(self,n):
      LogicGate.__init__(self,n)
      self.pinA = None
      self.pinB = None

   def getPinA(self):
      return int(input("Enter Pin A input for gate {}-->".format(self.getLabel())))
```

```
    def getPinB(self):
        return int(input("Enter Pin B input for gate {}-->".format(self.getLabel())))

class UnaryGate(LogicGate):
    '''Клас логічних елементів з одним входом'''

    def __init__(self,n):
        LogicGate.__init__(self,n)
        self.pin = None

    def getPin(self):
        return int(input("Enter Pin input for gate {}-->".format(self.getLabel())))
```

Тут реалізуються ці два класи. Конструктори обох починаються з явного виклику конструктора батьківського класу з використанням методу __init__. Коли ми створюємо екземпляр класу BinaryGate, то насамперед хочемо ініціалізувати будь-які елементи даних, які успадковуються від LogicGate. У даному випадку це мітка вентиля. Потім конструктор додає два входи (pinA і pinB). Це дуже розповсюджена схема, яку вам слід використовувати при проектуванні ієрархії класів. Конструктору дочірнього класу спочатку потрібно викликати конструктор батьківського класу, і тільки потім перемкнутися на власні, відмінні від предка, дані.

Єдиним, що додасться до поведінки класу BinaryGate буде можливість отримувати значення від двох вхідних ліній. Оскільки вони беруться звідкись ззовні, то за допомогою оператора введення ми можемо просто попросити користувача надати їх. Те ж саме відбувається в реалізації класу UnaryGate, за винятком того моменту, що він має лише один вхід.

Тепер, коли у нас є спільні класи для вентилів, залежні від кількості їх входів, ми можемо створювати специфічні вентилі з унікальним поведінкою. Наприклад, клас AndGate, який буде підкласом BinaryGate, оскільки елемент ”I” має два входи. Як і раніше, перший рядок конструктора викликає конструктор базового класу (BinaryGate), який, у свою чергу, викликає конструктор свого батька (LogicGate). Зверніть увагу, що клас AndGate не надає будь-яких нових додаткових даних, оскільки успадковує дві вхідні лінії, одну вихідну і мітку.

```
class AndGate(BinaryGate):
    def __init__(self,n):
        BinaryGate.__init__(self,n)

    def performGateLogic(self):
        a = self.getPinA()
        b = self.getPinB()
        if a==1 and b==1:
            return 1
        else:
            return 0
```

Єдина річ, яку необхідно додати в AndGate, — це специфічна поведінка при виконанні логічних операцій. Це те місце, де ми можемо надати метод performGateLogic. Для вентиля ”I” він спочатку повинен отримати два вхідних значення і повернути 1, якщо обидва вони рівні одиниці.

Ми можемо продемонструвати роботу класу AndGate, створивши його примірник і попросивши його вирахувати своє вихідне значення. Наступний код показує AndGate-об’єкт g1, який має внутрішню мітку ”G1”. Коли ми викликаємо метод getOutput, об’єкт спочатку повинен викликати свій метод performGateLogic, який, у свою чергу, запитує значення з двох вхідних ліній. Після того, як необхідні дані отримані, відображається правильне вихідне значення.

```
>>> g1 = AndGate("G1")
>>> g1.getOutput()
Enter Pin A input for gate G1-->1
Enter Pin B input for gate G1-->0
0
```

Така ж робота повинна бути проведена для елементів "АБО" і "НЕ". Клас OrGate також буде субкласом BinaryGate, а клас NotGate розширить UnaryGate. Обидва вони потребуватимуть власної реалізації функції performGateLogic зі специфічною поведінкою.

Ми можемо використовувати одиничний логічний елемент, сконструювавши на початку екземпляр одного з класів вентилів і потім запитавши його вихідне значення (що, у свою чергу, вимагатиме надання вхідних даних). Наприклад,

```
>>> g2 = OrGate("G2")
>>> g2.getOutput()
Enter Pin A input for gate G2-->1
Enter Pin B input for gate G2-->1
1
>>> g2.getOutput()
Enter Pin A input for gate G2-->0
Enter Pin B input for gate G2-->0
0
>>> g3 = NotGate("G3")
>>> g3.getOutput()
Enter Pin input for gate G3-->0
1
```

Тепер, коли у нас є працюючі базові вентилі, ми можемо повернутися до побудови схем. Щоб створити схему, нам необхідно з'єднати вентилі разом: вихід одного до входу іншого. Для цього ми реалізуємо новий клас під назвою Connector.

Клас Connector не належить до ієрархії логічних елементів. Однак, він буде використовувати її, оскільки кожен з'єднувач має два вентиля — по одному на кожен кінець. Відносини такого виду дуже важливі в об'єктно-орієнтованому програмуванні. Вони називаються відносинами "HAS-A". Нагадаємо, що раніше ми використовували словосполучення "IS-A відношення", щоб показати, як дочірній клас відноситься до батьківського. Наприклад, UnaryGate є (IS-A) LogicGate.

Тепер для класу Connector ми скажемо, що він має LogicGate, маючи на увазі, що з'єднувачі мають усередині екземпляри LogicGate, але не є частиною ієрархії. При конструюванні класів дуже важливо розрізняти ті з них, які мають відносини "IS-A" (що вимагає успадкування), і ті, які володіють відносинами "HAS-A" (без успадкування).

Два примірники вентилів всередині кожного об'єкта з'єднувача будуть позначатися як fromgate і togate, розрізняючи таким чином, що дані будуть "текти" від виходу одного вентиля до входу іншого.

```
class Connector:
   def __init__(self, fgate, tgate):
      self.fromgate = fgate
      self.togate = tgate
      tgate.setNextPin(self)
   def getFrom(self):
      return self.fromgate
   def getTo(self):
      return self.togate
```

Виклик setNextPin дуже важливий при створенні з'єднувачів. Нам необхідно додати цей метод до наших класів для вентилів таким чином, щоб кожен togate міг вибрати відповідну вхідні лінію для з'єднання.

```
def setNextPin(self,source):
    if self.pinA == None:
        self.pinA = source
    else:
        if self.pinB == None:
            self.pinB = source
        else:
            raise RuntimeError("Error: NO EMPTY PINS")
```

У класі BinaryGate для вентилів з двома можливими вхідними лініями конектор повинен приєднуватися тільки до однієї з них. Якщо доступні обидві, то за замовчуванням ми будемо вибирати pinA. Якщо він вже приєднаний до чого-небудь, то виберемо pinB. Під'єднатися до вентиля, який не має доступних входів, неможливо.

Тепер можна отримувати вхідні дані двома способами: ззовні, як раніше, і з виходу вентиля, приєднаного до входу даного. Ця вимога змінює методи getPinA і getPinB.

```
def getPinA(self):
    if self.pinA == None:
        return input("Enter Pin A input for gate " + self.getName()+"-->")
    else:
        return self.pinA.getFrom().getOutput()
```

Якщо вхідні лінія ні до чого не приєднана (None), то, як і раніше, буде задаватися питання користувачеві. Однак, якщо вона пов'язана, то підключення здійсниться, зажадавши значення виходу fromgate. У свою чергу, це запускає логічну обробку вентилем отриманих даних. Процес триває, поки є доступні входи, і остаточне вихідне значення стає необхідним входом для вентиля в питанні. У якомусь сенсі, схема працює у зворотний бік, щоб знайти вхідні дані, які необхідні для виробництва кінцевого результату.

Наступний фрагмент конструює схему, раніше показану в цьому розділі:

```
>>> g1 = AndGate("G1")
>>> g2 = AndGate("G2")
>>> g3 = OrGate("G3")
>>> g4 = NotGate("G4")
>>> c1 = Connector(g1,g3)
>>> c2 = Connector(g2,g3)
>>> c3 = Connector(g3,g4)
```

Виходи двох вентилів "І" (g1 і g2) з'єднані з вентилем "АБО" (g3), а його вихід — з вентилем "НЕ" (g4). Вихід вентиля "НЕ" — це вихід схеми цілком. Приклад роботи:

```
>>> g4.getOutput()
Pin A input for gate G1-->0
Pin B input for gate G1-->1
Pin A input for gate G2-->1
Pin B input for gate G2-->1
0
```

Спробуйте самі, використовуючи повний код класу.

```
class LogicGate:
    def __init__(self,n):
        self.name = n
        self.output = None
    def getName(self):
        return self.name
    def getOutput(self):
        self.output = self.performGateLogic()
        return self.output

class BinaryGate(LogicGate):
    def __init__(self,n):
        LogicGate.__init__(self,n)
        self.pinA = None
        self.pinB = None
    def getPinA(self):
        if self.pinA == None:
            return int(input("Enter Pin A input for gate "+self.getName()+"-->"))
        else:
            return self.pinA.getFrom().getOutput()
    def getPinB(self):
        if self.pinB == None:
            return int(input("Enter Pin B input for gate "+self.getName()+"-->"))
        else:
            return self.pinB.getFrom().getOutput()
    def setNextPin(self,source):
        if self.pinA == None:
            self.pinA = source
        else:
            if self.pinB == None:
                self.pinB = source
            else:
                print("Cannot Connect: NO EMPTY PINS on this gate")

class AndGate(BinaryGate):
    def __init__(self,n):
        BinaryGate.__init__(self,n)
    def performGateLogic(self):
        a = self.getPinA()
        b = self.getPinB()
        if a==1 and b==1:
            return 1
        else:
            return 0

class OrGate(BinaryGate):
    def __init__(self,n):
        BinaryGate.__init__(self,n)
    def performGateLogic(self):
        a = self.getPinA()
        b = self.getPinB()
        if a ==1 or b==1:
            return 1
        else:
            return 0

class UnaryGate(LogicGate):
    def __init__(self,n):
        LogicGate.__init__(self,n)
        self.pin = None
```

```
    def getPin(self):
        if self.pin == None:
            return int(input("Enter Pin input for gate "+self.getName()+"-->"))
        else:
            return self.pin.getFrom().getOutput()
    def setNextPin(self,source):
        if self.pin == None:
            self.pin = source
        else:
            print("Cannot Connect: NO EMPTY PINS on this gate")

class NotGate(UnaryGate):
    def __init__(self,n):
        UnaryGate.__init__(self,n)
    def performGateLogic(self):
        if self.getPin():
            return 0
        else:
            return 1

class Connector:
    def __init__(self, fgate, tgate):
        self.fromgate = fgate
        self.togate = tgate
        tgate.setNextPin(self)
    def getFrom(self):
        return self.fromgate
    def getTo(self):
        return self.togate

def main():
   g1 = AndGate("G1")
   g2 = AndGate("G2")
   g3 = OrGate("G3")
   g4 = NotGate("G4")
   c1 = Connector(g1,g3)
   c2 = Connector(g2,g3)
   c3 = Connector(g3,g4)
   print(g4.getOutput())

main()
```

Створіть два нові класи вентилів: NorGate і NandGate. Перший працює подібно OrGate, до виходу якого підключено НЕ. Другий — як AndGate з НЕ на виході.

Створіть ряд з вентилів, який доводив би, що NOT ((A and B) or (C and D)) це те ж саме, що і NOT (A and B) and NOT (C and D). Переконайтеся, що використовуєте в цій симуляції деякі з новостворених вами вентилів.