ОБ’ЄКТИ В МОВІ PYTHON

Питання 7.2

Створення класу

- Використовується ключове слово class
  - Назва класу повинна починатись з літери або _, може складатись з літер, цифр та символів _.
  - Рекомендується називати класи в стилі **CamelCase**.
  - Приналежність коду до класу визначається відступами.

```
1 class MyFirstClass:
2     pass
3
4 a = MyFirstClass()
5 b = MyFirstClass()
6
7 print(a)
8 print(b)
```

Код створює (instantiate) два об'єкти з класу: a і b.

- Адреси пам'яті рідко використовуються в коді.
- Тут вони демонструють створення двох окремих об'єктів.

```
In [1]: runfile('C:/Users/spuasson/Desktop/oopython/untitled0.py'
<__main__.MyFirstClass object at 0x000002451A733D30>
<__main__.MyFirstClass object at 0x000002451A7E9390>
```

Атрибути класу

```
1 class Point:
2     pass
3
4 p1 = Point()
5 p2 = Point()
6
7 p1.x = 5
8 p1.y = 4
9 p2.x = 3
10 p2.y = 6
11
12 print(p1.x, p1.y)
13 print(p2.x, p2.y)
```

```
5 4
3 6
```

- Створений клас Point не має даних або поведінки.
  - Далі вводяться два екземпляри класу.
  - Для кожного з них дописуються атрибути (координати x та y).
  - Значення атрибуту задається за допомогою оператору доступу «.».
  - **<об'єкт>.<атрибут> = <значення>**
  - Значення може бути будь-чим: примітивним чи вбудованим типом, іншим об'єктом, навіть функцією чи іншим класом!
- Нас цікавлять дії, що спричиняють зміни значень атрибутів.
  - Додамо поведінку для нашого класу

```
1 class Point:
2     def reset(self):
3         self.x = 0
4         self.y = 0
5
6 p = Point()
7 p.reset()
8 print(p.x, p.y)
```

```
0 0
```

Метод у Python форматується ідентично до функції.

- Ключове слово def, назва методу, дужки зі списком параметрів, двокрапка

Функції та методи

- Методи вимагають один обов'язковий параметр – self.
  - Це посилання на об'єкт, для якого метод викликано.
  - У виклик методу явно не передається: p.reset(), автоматично це робить Python.
- Замість виклику метода для об'єкта ми можемо викликати метод для цілого класу і передати йому об'єкт.

```
1 p = Point()
2 Point.reset(p)
3 print(p.x, p.y)
```

- Якщо забули self, з'явиться помилка на зразок

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reset() takes no arguments (1 given)
```

Більше аргументів

```
1 import math
2
3 class Point:
4     def move(self, x, y):
5         self.x = x
6         self.y = y
7     def reset(self):
8         self.move(0, 0)
9     def calculate_distance(self, other_point):
10        return math.sqrt((self.x - other_point.x)**2 +
11                         (self.y - other_point.y)**2)
12 # використання:
13 point1 = Point()
14 point2 = Point()
15 point1.reset()
16 point2.move(5,0)
17 print(point2.calculate_distance(point1))
18 assert (point2.calculate_distance(point1) ==
19         point1.calculate_distance(point2))
20 point1.move(3,4)
21 print(point1.calculate_distance(point2))
22 print(point1.calculate_distance(point1))
```

- Клас має три методи.
  - move() з аргументами x та y
  - reset() – безаргументний
  - calculate_distance() з об'єктом-точкою, до якої обчислюється відстань

```
5.0
4.47213595499958
0.0
```

Ініціалізація об’єкту

- Якщо координати x та y об’єкту класу Point не були ініціалізовані напряму чи за допомогою методу move, створена точка не матиме позиції.
  - Що тоді станеться при спробі доступу до координат?

```
13 point = Point()
14 point.x = 5
15 print(point.x)
16 print(point.y)
```

```
5
Traceback (most recent call last):

  File "<ipython-input-2-a90390e9c59d>", line 1, in <module>
    runfile('C:/Users/spuasson/Desktop/untitled0.py', wdir='C:/Users/spuasson/Desktop')

  File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 710, in runfile
    execfile(filename, namespace)

  File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 101, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/spuasson/Desktop/untitled0.py", line 16, in <module>
    print(point.y)

AttributeError: 'Point' object has no attribute 'y'
```

Конструктори та ініціалізатори

- Краще змусити користувача вводити дані, яких не вистачає при ініціалізації.
  - В ООП ***конструктор*** – спеціальний метод, який створює та ініціалізує об'єкти.
  - Мова Python має окремо ***конструктор*** та ***ініціалізатор***.
  - Конструктор Python використовується дуже рідко.
  - Ініціалізатором є ***спеціальний метод*** __init__().
  - У назвах власних методів використовувати __ не рекомендується.

```
1  class Point:
2      def __init__(self, x, y):
3          self.move(x, y)
4      def move(self, x, y):
5          self.x = x
6          self.y = y
7      def reset(self):
8          self.move(0, 0)
9
10 # Конструювання об'єкту класу Point
11 point = Point(3, 5)
12 print(point.x, point.y)
```

Вивід програми:

3 5

Тепер без другого параметру програма не запуститься

Конструктори та ініціалізатори

- Ініціалізатор (як і будь-який метод) допускає використання аргументів за замовчуванням (default arguments).

```
class Point:
    def __init__(self, x=0, y=0):
        self.move(x, y)
```

- Конструктор __new__() приймає лише один аргумент – клас, з якого конструюється об'єкт.
  - Конструктор викликається *до* створення об'єкту – self не потрібен.
  - Цей метод повинен повернути новостворений об'єкт.
  - Основне застосування – в метапрограмуванні, у звичайних задачах достатньо __init__().

Документування коду

```python
import math
class Point:
    'Представляє двовимірну точку в геометричних координатах'
    def __init__(self, x=0, y=0):
        '''Ініціалізує позицію нової точки. Координати x та y
        можна задавати. Якщо вони не задані, беруть значення за
        замовчуванням.'''
        self.move(x, y)
    def move(self, x, y):
        "Переміщення точки в нове місце 2D-простору."
        self.x = x
        self.y = y
    def reset(self):
        'Переміщення точки в початок відліку: 0, 0'
        self.move(0, 0)
    def calculate_distance(self, other_point):
        """Обчислити відстань від даної точки до точки, що є
        параметром методу.
        Функція використовує теорему Піфагора для визначення
        відстані між двома точками. Відстань типу float."""
        return math.sqrt((self.x - other_point.x)**2 +
                         (self.y - other_point.y)**2)
```

- Після компіляції можна викликати help(Point) в консолі

```
class Point(builtins.object)
 |  Представляє двовимірну точку в геометричних координатах
 |
 |  Methods defined here:
 |
 |  __init__(self, x=0, y=0)
 |      Ініціалізує позицію нової точки. Координати x та y
 |      можна задавати. Якщо вони не задані, беруть значення за
 |      замовчуванням.
 |
 |  calculate_distance(self, other_point)
 |      Обчислити відстань від даної точки до точки, що є
 |      параметром методу.
 |      Функція використовує теорему Піфагора для визначення
 |      відстані між двома точками. Відстань типу float.
 |
 |  move(self, x, y)
 |      Переміщення точки в нове місце 2D-простору.
 |
 |  reset(self)
 |      Переміщення точки в початок відліку: 0, 0
 |
 |  ----------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Модулі та пакети (packages)

- Як організувати взаємодію між об'єктами?
  - 1) дописувати відповідний код у кінець файлу з описом класу
  - 2) використовувати ***модулі***: кожен файл з кодом є модулем.
    - Якщо в одній папці є кілька модулів, можна завантажити клас з одного модуля в іншому.
- Наприклад, при розробці системи електронної комерції можна виділити модулі:
  - database.py – модуль для доступу до бази даних
  - модуль інформації про товар
  - модуль для складських операцій
  - модуль з описом клієнтів та ін.
  - Кожен з модулів у прикладі потребує доступу до БД.

Імпортування модулів

- Використовується ключове слово import.
  - Вже використовували: імпорт функції *sqrt()* із вбудованого модуля **math** для обчислення *distance*.
- Нехай матимемо
  - модуль database.py, який містить клас Database
  - модуль products.py, який виконує запити до БД, пов'язані з товарами.
- Варіанти імпортування (у файлі products.py):

```
import database
db = database.Database()
# виконувати запити до db
```

```
from database import Database
db = Database()
# виконувати запити до db
```

```
from database import Database as DB
db = DB()
# виконувати запити до db
```

Імпортування модулів

- Можна імпортувати кілька класів з одного модуля:

```
from database import Database, Query
```

- Імпорт усіх класів та функцій з модуля:

```
from database import *
```

  - Використовувати не рекомендується.
  - Використовуючи конкретний клас, не видно, звідки він імпортований: страждає підтримка коду.
  - Синтаксис import * знижує надійність роботи «фішок» багатьох редакторів коду: надійне автодоповнення коду (code completion), здатність перескакувати до оголошення класу або вбудованої (inline) документації.
  - import * також вносить неочікувані об'єкти в локальний простір імен (namespace).

Організація модулів

- Розростання проекту змушує ввести ще один рівень абстракції – ***пакети*** (package).
  - Це набір модулів у папці (назва папки = назва модуля).
  - Про те, що папка є пакетом, свідчить наявність файлу __init__.py.

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

Перенесемо наші модулі в пакет ecommerce.

Також є файл main.py, щоб запустити програму.

Додамо інший пакет в ecommerce для різних опцій щодо оплати (payments).

Імпортування модулів та класів між пакетами

- Для імпортування модулів існують два види імпорту:
  - Абсолютний імпорт (Absolute imports)
  - Відносний імпорт (relative imports)
- ***Абсолютний імпорт*** задає повний шлях до модуля.

```
import ecommerce.products
product = ecommerce.products.Product()
```

- або

```
from ecommerce.products import Product
product = Product()
```

- або

```
from ecommerce import products
product = products.Product()
```

Використовується оператор доступу «.» для відокремлення пакетів і модулів.

Абсолютний імпорт

- Такі інструкції працюватимуть з будь-якого модуля.
  - Інстанціювати клас Product можна в main.py, модулі database або в якомусь з двох модулів пакету payments.
- Який же синтаксис обрати?
  - Справа смаку.
  - Якщо буде використовуватись багато класів та функцій з модуля products, простіше імпортувати весь модуль, а потім отримувати доступ до окремих класів за допомогою products.Product.
  - Якщо потрібно один-два класи, можна імпортувати їх напряму:
    - from ecommerce.proucts import Product.

Відносний імпорт

- Такий імпорт відбувається відносно поточного модуля.
  - Наприклад, якщо ми працюємо в модулі products і хочемо імпортувати клас Database з модуля database:

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

```
from .database import Database
```

Крапка перед database означає використання модуля в рамках даного пакету.

- При внесенні змін до модуля paypal з пакету ecommerce.payments, кажемо: «використовуй пакет database всередині батьківського пакету".

```
from ..database import Database
```

```
from ..contact.email import send_mail
```

Відносний імпорт

- Також можна імпортувати код напряму з пакетів.
  - Файл __init__.py може містити будь-які змінні чи класи, які будуть доступні в усьому пакеті.
  - У файлі ecommerce/__init__.py може бути рядок

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

```
from .database import db
```

Можемо отримати доступ до атрибута db з main.py чи іншого файлу, використовуючи:

```
from ecommerce import db
```

Організація вмісту модуля

- У межах одного модуля можна включати змінні, класи та функції.
  - Наприклад, імпортовано клас Database у різні модулі, а потім відбувається його інстанціювання, проте краще мати один глобальний об'єкт database, доступний з модуля database.

```
class Database:
    # реалізація роботи з БД
    pass

database = Database()
```

  - Для доступу до об'єкта database виконаємо імпорт

```
from ecommerce.database import database
```

  - Об'єкт database створюється негайно після імпортування модуля (зазвичай при запуску програми).
    - Проблема: підключення до БД може займати час, сповільнюючи запуск, або інформація щодо підключення ще не доступна.

Організація вмісту модуля

- Можна відкласти створення бази даних до того, як вона дійсно знадобиться, викликаючи функцію initialize_database() для створення module-level змінної (ключове слово global):

```
class Database:
    # реалізація роботи з БД
    pass

database = None

def initialize_database():
    global database
    database = Database()
```

  - Без global буде створена нова локальна для методу змінна, яка залишить module-level значення без змін.

Проблема: можемо запустити програму, хоча бажали лише отримати доступ до кількох функцій у ній

- Для вирішення проблеми код запуску записують у функцію (зазвичай, main) і виконують її, коли запускаємо модуль в якості скрипта, проте не при імпортуванні коду з іншого місця.

```
class UsefulClass:
    '''Цей клас може бути корисним для інших модулів.'''
    pass

def main():
    '''створює корисний клас і дещо робить з ним для нашого модуля.'''

useful = UsefulClass()
print(useful)
if __name__ == "__main__":
    main()
```

  - Спеціальна змінна __name__ задає назву модуля when it was imported.

Класи можна описувати будь-де

- Класи зазвичай визначають на рівні модуля, проте це можна робити всередині функції чи методу:

```
def format_string(string, formatter=None):
    '''Форматування рядка за допомогою об'єкта formatter,
    який має метод format(), який приймає рядок.'''
    class DefaultFormatter:
        '''Форматування рядка in title case.'''
        def format(self, string):
            return str(string).title()
    if not formatter:
        formatter = DefaultFormatter()
    return formatter.format(string)

hello_string = "Привіт, світ! Як ти сьогодні?"
print(" ввід: " + hello_string)
print("вивід: " + format_string(hello_string))
```

```
 ввід: Привіт, світ! Як ти сьогодні?
вивід: Привіт, Світ! Як Ти Сьогодні?
```

Хто може отримати доступ до моїх даних?

- У більшості ОО мов програмування існує контроль доступу до даних.
  - Атрибути та методи об'єктів позначаються як private, public, protected…
  - Python ***не передбачає*** таких інструментів.
  - За домовленістю внутрішні (internal) атрибути або методи називаються з префіксом _, проте доступ до них вільний.
  - Наголошення на закритості доступу виконується також за допомогою *декорування імен* (розширення імен, name mangling) – дописуванні до імені префікса __.

Вимога приватності

```python
class SecretString:
    '''Не зовсім безпечний спосіб зберігання секретного тексту.'''
    def __init__(self, plain_string, pass_phrase):
        self.__plain_string = plain_string
        self.__pass_phrase = pass_phrase
    def decrypt(self, pass_phrase):
        '''Показувати рядок лише тоді, коли pass_phrase правильний.'''
        if pass_phrase == self.__pass_phrase:
            return self.__plain_string
        else:
            return ''

secret_string = SecretString("ACME: Top Secret", "antwerp")
print(secret_string.decrypt("antwerp"))
print(secret_string.__plain_text)
```

```
ACME: Top Secret
Traceback (most recent call last):

  File "<ipython-input-4-a90390e9c59d>", line 1, in <module>
    runfile('C:/Users/spuasson/Desktop/untitled0.py', wdir='C:/Users/spuasson/Desktop')

  File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 710, in runfile
    execfile(filename, namespace)

  File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 101, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/spuasson/Desktop/untitled0.py", line 15, in <module>
    print(secret_string.__plain_text)

AttributeError: 'SecretString' object has no attribute '__plain_text'
```

Хакнемо код

```
secret_string = SecretString("ACME: Top Secret", "antwerp")
print(secret_string._SecretString__plain_string)
```

```
In [5]: runfile('C:/Users/spuasson/Desktop/untitled0.py', wdir='C:/Users/spuasson/Desktop')
ACME: Top Secret
```

- Декорування імен у роботі.
  - Коли дописуємо префікс __, насправді назва властивості отримує префікс _<назвакласу>.
  - Коли методи в класі отримують доступ до внутрішньої змінної, вони *автоматично* виправляються (unmangle).
  - При доступі ззовні класу бажаючі повинні власноруч виконувати декорування імені.
  - ***Декорування імені не гарантує приватності***, проте настійливо її рекомендує.

ДЯКУЮ ЗА УВАГУ!

Наступне питання: Схожість об’єктів: наслідування та поліморфізм

Сторонні бібліотеки

- Вбудованих бібліотек (пакетів та модулів) може не вистачити.
  - Якщо потрібно використати код іншого програміста
  - Якщо потрібно Write a supporting package yourself
- Якщо поставлено задачу, спочатку слід пошукати бібліотеки, що можуть допомогти в її вирішенні.
  - Python Package Index (PyPI) - http://pypi.python.org/
  - Для інсталяції бібліотеки використовують pip.
  - Він не йде разом із Python, проте для версії 3.4+ є інструмент для перевірки наявності pip (Для Unix-систем потрібні root-права - sudo):
  - **python -m ensurepip**
  - Інсталяція пакету за наявності pip:
  - **pip install назвапакету**
  - Через можливі проблеми з дозволами на встановлення рекомендується використовувати системні інсталятори.

Особливості інсталяції

- Python 3.4+ постачає утиліту venv.
  - Вона надає віртуальне середовище для інсталяції в робочу папку.
  - Коли запускається pip або python, Python-система зовсім не буде зачеплена
  - cd project_directory
    python -m venv env
    source env/bin/activate # на Linux або MacOS
    env/bin/activate.bat # на Windows
- Зазвичай віртуальне середовище створюють для кожного проекту (ігнорується системами контролю версій).
  - Конкретне віртуальне середовище активується та деактивується в процесі роботи над проектом.

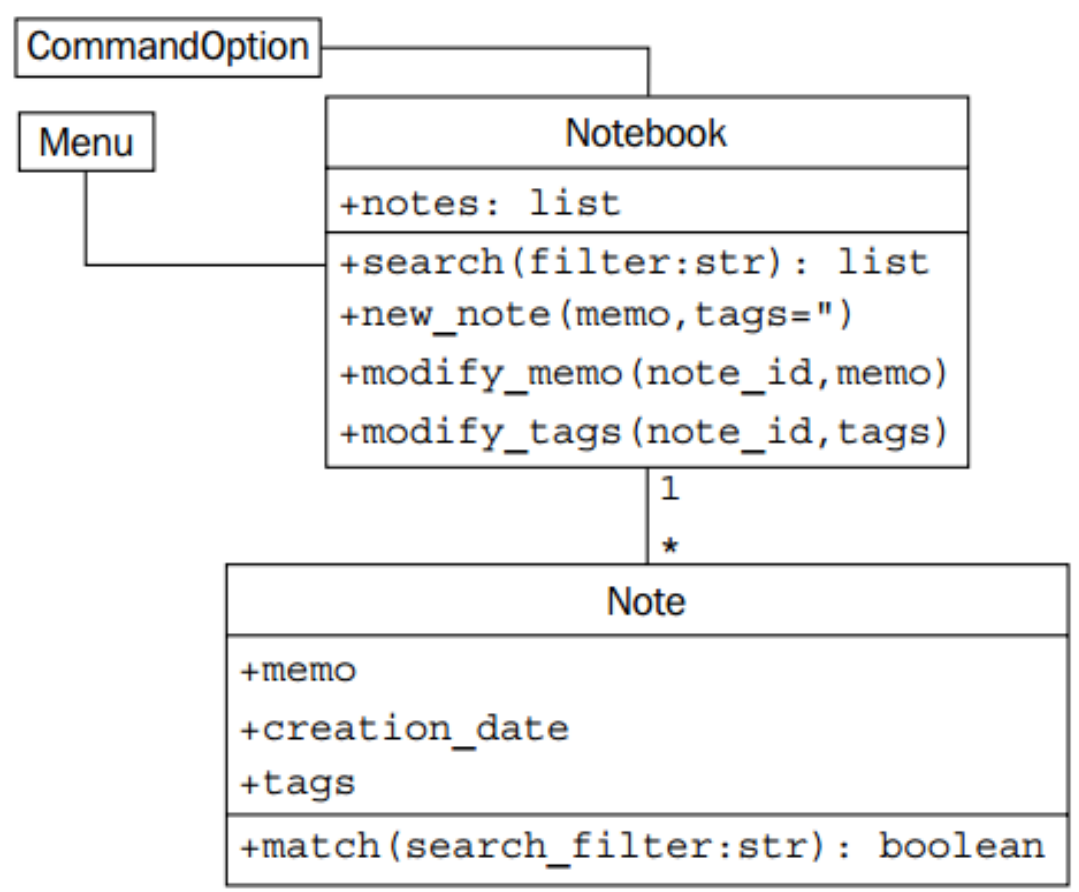
Загальний приклад: простий додаток-записник у командному рядку

- Короткий аналіз:
  - Записи – це короткі нагадування, збережені в записнику.
  - Кожен запис має записати день його записування, може мати теги для спрощення querying.
  - Потрібна можливість змінювати записи.
  - Існує необхідність у пошуку записів.
- Очевидний контейнерний об’єкт – Note; менш очевидний – Notebook.
  - Теги та дати теж можуть бути об’єктами, проте дати можна взяти зі стандартної бібліотеки, а теги представити як список рядків, відокремлених комами.
  - Для усунення складності окремо не будемо для них створювати класи у прототипі.

Об’єкт Note

- Note-об’єкти мають атрибути з самим текстом нагадування, тегами та датою створення.
  - Кожний запис повинен мати унікальний цілочисельний ідентифікатор, щоб користувачі могли обирати цей запис в інтерфейсі меню.
  - Записи можуть мати методи для редагування їх вмісту або тегів. Або можемо просто дозволити доступ записній книжці до атрибутів напряму.
  - Для спрощення пошуку краще внести метод для порівняння записів (Note-об’єктів).
    - Якщо хочемо змінити параметри пошуку (за тегами замість контенту чи з підтримкою чутливості до регістру), потрібно робити це в одному місці.
- Notebook-об’єкт має список записів у якості атрибуту.
  - Також буде потрібен метод для пошуку, який повертає відфільтрований список записів.

Як будемо взаємодіяти з об'єктами?



- Задаємо інтерфейс командного рядка.
  - Запускаємо програму з різними опціями на додавання чи редагування команд
    - Матимемо простеньке меню, що дозволятиме обирати різні можливості роботи із записною книгою.

```
parent_directory/
    notebook.py
    menu.py
    command_option.py
```

```
import datetime

# Store the next available id for all new notes
last_id = 0

class Note:
    '''Represent a note in the notebook. Match against a
    string in searches and store tags for each note.'''

    def __init__(self, memo, tags=''):
        '''initialize a note with memo and optional
        space-separated tags. Automatically set the note's
        creation date and a unique id.'''
        self.memo = memo
        self.tags = tags
        self.creation_date = datetime.date.today()
        global last_id
        last_id += 1
        self.id = last_id

    def match(self, filter):
        '''Determine if this note matches the filter
        text. Return True if it matches, False otherwise.

        Search is case sensitive and matches both text and
        tags.'''
        return filter in self.memo or filter in self.tags
```

Тепер код...

```
>>> from notebook import Note
>>> n1 = Note("hello first")
>>> n2 = Note("hello again")
>>> n1.id
1
>>> n2.id
2
>>> n1.match('hello')
True
>>> n2.match('second')
False
```

Створимо записну книжку

```
class Notebook:
    '''Represent a collection of notes that can be tagged,
    modified, and searched.'''

    def __init__(self):
        '''Initialize a notebook with an empty list.'''
        self.notes = []

    def new_note(self, memo, tags=''):
        '''Create a new note and add it to the list.'''
        self.notes.append(Note(memo, tags))

    def modify_memo(self, note_id, memo):
        '''Find the note with the given id and change its
        memo to the given value.'''
        for note in self.notes:
            if note.id == note_id:
                note.memo = memo
                break
```

- modify_tags та modify_memo майже однакові.
  - В майбутньому виправимо.

```
    def modify_tags(self, note_id, tags):
        '''Find the note with the given id and change its
        tags to the given value.'''
        for note in self.notes:
            if note.id == note_id:
                note.tags = tags
                break

    def search(self, filter):
        '''Find all notes that match the given filter
        string.'''
        return [note for note in self.notes if
                note.match(filter)]
```

```
>>> from notebook import Note, Notebook
>>> n = Notebook()
>>> n.new_note("hello world")
>>> n.new_note("hello again")
>>> n.notes
[<notebook.Note object at 0xb730a78c>, <notebook.Note object at
  0xb73103ac>]
>>> n.notes[0].id
1
>>> n.notes[1].id
2
>>> n.notes[0].memo
'hello world'
>>> n.search("hello")
[<notebook.Note object at 0xb730a78c>, <notebook.Note object at
  0xb73103ac>]
>>> n.search("world")
[<notebook.Note object at 0xb730a78c>]
>>> n.modify_memo(1, "hi world")
>>> n.notes[0].memo
'hi world'
```

Вирішуємо проблему з повтором коду

- Обидва методи намагаються визначити запис по Id до того, як щось із ним робити.
  - Додамо метод, який знаходитиме запис за його ID.
  - Вважатимемо його внутрішнім (назва з _ спереду), проте інтерфейс меню може отримати доступ до методу за бажанням:

```
def _find_note(self, note_id):
    '''Locate the note with the given id.'''
    for note in self.notes:
        if note.id == note_id:
            return note
    return None

def modify_memo(self, note_id, memo):
    '''Find the note with the given id and change its
    memo to the given value.'''
    self._find_note(note_id).memo = memo
```

Інтерфейс меню

```
import sys
from notebook import Notebook, Note

class Menu:
    '''Display a menu and respond to choices when run.'''
    def __init__(self):
        self.notebook = Notebook()
        self.choices = {
                "1": self.show_notes,
                "2": self.search_notes,
                "3": self.add_note,
                "4": self.modify_note,
                "5": self.quit
                }
    def display_menu(self):
        print("""
Notebook Menu

1. Show all Notes
2. Search Notes
3. Add Note
4. Modify Note
5. Quit
""")
```

- Інтерфейс повинен просто представити меню та дозволити користувачеві ввести деякий вибір:

```
    def run(self):
        '''Display the menu and respond to choices.'''
        while True:
            self.display_menu()
            choice = input("Enter an option: ")
            action = self.choices.get(choice)
            if action:
                action()
            else:
                print("{0} is not a valid choice".format(choice))
```

```
    def show_notes(self, notes=None):
        if not notes:
            notes = self.notebook.notes
        for note in notes:
            print("{0}: {1}\n{2}".format(
                note.id, note.tags, note.memo))
    def search_notes(self):
        filter = input("Search for: ")
        notes = self.notebook.search(filter)
        self.show_notes(notes)

    def add_note(self):
        memo = input("Enter a memo: ")
        self.notebook.new_note(memo)
        print("Your note has been added.")
```

Продовження класу

- Якщо запустимо код, зміна записів не працює:
  - Записна книга (notebook) перериває роботу, коли введений ID запису не існує.
  - Навіть якщо ввели коректний ID, все-одно робота перерветься: ID запису – ціле число, а в меню – рядок.

```
    def modify_note(self):
        id = input("Enter a note id: ")
        memo = input("Enter a memo: ")
        tags = input("Enter tags: ")
        if memo:
            self.notebook.modify_memo(id, memo)
        if tags:
            self.notebook.modify_tags(id, tags)
```

```
    def quit(self):
        print("Thank you for using your notebook today.")
        sys.exit(0)

if __name__ == "__main__":
    Menu().run()
```

Виправлення помилки з неузгодженими типами

- Останній баг можна закрити, змінивши метод _find_note() класу Notebook, щоб порівнювати рядки замість цілих чисел із запису:

```
def _find_note(self, note_id):
    '''Locate the note with the given id.'''
    for note in self.notes:
        if str(note.id) == str(note_id):
            return note
    return None
```

Виправлення помилки з неіснуючим ID

- Внесемо зміни до modify-методу записної книги, щоб перевірити, чи повертає _find_note() запис (note):

```
def modify_memo(self, note_id, memo):
    '''Find the note with the given id and change its
    memo to the given value.'''
    note = self._find_note(note_id)
    if note:
        note.memo = memo
        return True
    return False
```

  - Тепер метод повертає True або False, залежно від того, чи знайдено запис.
  - Меню може використати повернене значення, щоб відобразити помилку, якщо користувач ввів неіснуючий запис.