



ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ В ДІЇ

Питання 7.4.

ООП в дії

- Спочатку ідентифікують об'єкти та моделюють їх дані й поведінку.
 - Включає визначення структур даних
- Для роботи з багатокутниками спочатку виділяють окремі точки, які моделюються як кортежі з координатних пар:

```
square = [(1,1), (1,2), (2,2), (2,1)]
```

- У багатокутника є відстані між вершинами та периметр

```
1 import math
2
3 def distance(p1, p2):
4     return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)
5
6 def perimeter(polygon):
7     perimeter = 0
8     points = polygon + [polygon[0]]
9     for i in range(len(polygon)):
10         perimeter += distance(points[i], points[i+1])
11     return perimeter
```

Інкапсуляція vs структурний код

```
1 import math
2 class Point:
3     def __init__(self, x, y):
4         self.x = x
5         self.y = y
6
7     def distance(self, p2):
8         return math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)
9
10 class Polygon:
11     def __init__(self):
12         self.vertices = []
13
14     def add_point(self, point):
15         self.vertices.append((point))
16
17     def perimeter(self):
18         perimeter = 0
19         points = self.vertices + [self.vertices[0]]
20         for i in range(len(self.vertices)):
21             perimeter += points[i].distance(points[i+1])
22         return perimeter
```

```
■ >>> square = Polygon()
>>> square.add_point(Point(1,1))
>>> square.add_point(Point(1,2))
>>> square.add_point(Point(2,2))
>>> square.add_point(Point(2,1))
>>> square.perimeter()
4.0
```

```
import math

def distance(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

def perimeter(polygon):
    perimeter = 0
    points = polygon + [polygon[0]]
    for i in range(len(polygon)):
        perimeter += distance(points[i], points[i+1])
    return perimeter
```

```
>>> square = [(1,1), (1,2), (2,2), (2,1)]
>>> perimeter(square)
4.0
```

Можемо спростити об'єктно-орієнтований Polygon API

- Будемо конструювати в класі багато точок (список Point-об'єктів).

- Приймаємо й кортежі, щоб за потреби конструювати Point-об'єкти:

```
def __init__(self, points=None):  
    points = points if points else []  
    self.vertices = []  
    for point in points:  
        if isinstance(point, tuple):  
            point = Point(*point)  
        self.vertices.append(point)
```

- Якщо об'єкт не є кортежем, залишаємо його як є.
 - Вважаємо, що він або вже об'єкт класу Point, або невідомий качинотипізований (duck-typed) об'єкт.
- Загалом, більш складний набір даних, ймовірно, матиме більше специфічних для даних функцій.
 - Клас з атрибутами та методами стає кориснішим.

Звертайте увагу на те, як клас буде використовуватись

- Якщо ви намагаєтесь обчислити периметр одного багатокутника в контексті набагато більшої задачі, функцію використовувати простіше, особливо "one time only".
 - З іншого боку, **якщо програма потребує керування численними багатокутниками різними способами** (обчислити периметр, площу, площу перерізу з іншими багатокутниками, перемістити або масштабувати їх тощо), скоріше за все, потрібно ідентифікувати об'єкт.
- Додатково зверніть увагу на взаємодію між об'єктами.
 - Наслідування неможливо елегантно змодельовати без класів.
 - Розгляньте інші типи взаємодії: асоціацію та композицію.
 - Технічно, композиція моделюється за допомогою лише структур даних;
 - Наприклад, маємо список словників, значеннями яких є кортежі,
 - Проте часто простіше створити кілька класів та об'єктів, особливо, якщо присутня поведінка, пов'язана з даними.

Додавання поведінки до даних класу за допомогою властивостей

- Багато ОО мов (особливо Java) привчають ніколи напряму не отримувати доступ до атрибутів.

```
class Color:
    def __init__(self, rgb_value, name):
        self._rgb_value = rgb_value
        self._name = name

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self._name
```

```
>>> c = Color("#ff0000", "bright red")
>>> c.get_name()
'bright red'
>>> c.set_name("red")
>>> c.get_name()
'red'
```

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self.name = name

c = Color("#ff0000", "bright red")
print(c.name)
c.name = "red"
```

- Змінні названі з префіксом `_`, що передбачає їх *приватність* (неможливість доступу ззовні класу).
- Інші мови змушують змінні бути приватними.
- Методи `get()` та `set()` надають доступ до кожної змінної.

Навіщо наполягати на синтаксисі з методами?

- Причина: ймовірна потреба додавати код, коли значення буде встановлюватись або зчитуватись.
 - Наприклад, вирішимо кешувати значення та повертати кешоване значення або перевіряти, щоб значення було доречним для вводу.
 - У коді змінюється метод `set_name()`:

```
def set_name(self, name):  
    if not name:  
        raise Exception("Invalid Name")  
    self._name = name
```

- У Java та подібних мовах заміна прямого доступу на синтаксис з методами викличе потребу замінити у вже написаному коді звернень до відповідних даних.
 - Для Python особливого смислу у використанні методів для доступу немає, оскільки немає реального поняття приватних членів.

Python пропонує ключове слово `property`, щоб зробити методи схожими на атрибути

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name

    def _set_name(self, name):
        if not name:
            raise Exception("Invalid Name")
        self._name = name

    def _get_name(self):
        return self._name

    name = property(_get_name, _set_name)
```

- Спочатку змінимо назву атрибуту на (напів-) приватний атрибут `_name`.
- Потім додамо ще два (напів-) приватних методи, щоб виконувати перевірку (validation) значення змінної.
- У кінці маємо оголошення властивості.

```
>>> c = Color("#0000ff", "bright red")
>>> print(c.name)
bright red
>>> c.name = "red"
>>> print(c.name)
red
>>> c.name = ""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "setting_name_property.py", line 8, in _set_name
    raise Exception("Invalid Name")
Exception: Invalid Name
```


Детальніше про властивості

- Ключове слово `property` – аналог конструктора об'єкту.
 - Такий `property`-конструктор може приймати два додаткових аргументи: функцію видалення та `docstring` для властивості.
 - Функція `delete` рідко реалізується на практиці, проте може бути корисною для ведення логу або заборони видалення.

```
class Silly:
    def _get_silly(self):
        print("You are getting silly")
        return self._silly
    def _set_silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value
    def _del_silly(self):
        print("Whoah, you killed silly!")
        del self._silly

    silly = property(_get_silly, _set_silly,
                    _del_silly, "This is a silly property")
```

```
>>> s = Silly()
>>> s.silly = "funny"
You are making silly funny
>>> s.silly
You are getting silly
'funny'
>>> del s.silly
Whoah, you killed silly!
```

Використання docstring

```
Help on class Silly in module __main__:
```

```
class Silly(builtins.object)
```

```
| Data descriptors defined here:
```

```
|
```

```
| __dict__
```

```
|     dictionary for instance variables (if defined)
```

```
|
```

```
| __weakref__
```

```
|     list of weak references to the object (if defined)
```

```
|
```

```
| silly
```

```
|     This is a silly property
```

Декоратори – інший спосіб створювати властивості

```
class Silly:
    @property
    def silly(self):
        "This is a silly property"
        print("You are getting silly")
        return self._silly

    @silly.setter
    def silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value

    @silly.deleter
    def silly(self):
        print("Whoah, you killed silly!")
        del self._silly
```

- Код еквівалентний `silly = property(silly)`.
 - Основна відмінність – читабельність, позначаємо функцію `foo` як властивість над описом методу.
 - Тепер не потрібно створювати приватні методи з префіксами `_`, щоб означити властивість.
- Спочатку декоруємо метод `silly` в якості геттера.
 - Другий метод декоруємо з тою ж назвою, застосувавши атрибут `setter` з декорованого методу `foo()`!
 - Властивість повертає об'єкт, який завжди має атрибут `setter`, що може застосовуватись як декоратор до інших функцій.
 - Використання тієї ж назви для геттерів та сеттерів не вимагається, проте допомагає згрупувати кілька методів, що працюють з доступом до властивості.
 - Ми не можемо задати `docstring` за допомогою декораторів, тому слід покладатись на властивість, яка копіює `docstring` з початкового методу `getter`.

Вибираємо, коли використовувати властивості

- Розглянутий приклад – найбільш поширена ситуація для використання властивостей: маємо дані в класі, а потім бажаємо додати поведінку.
 - Технічно, в Python дані, властивості та методи є атрибутами класу.
 - Методи – callable-атрибути, а властивості – кастомізовані (customizable) атрибути.
 - Переконавшись, що атрибут не виконує дій, потрібно обрати представлення: стандартний атрибут для даних (data attribute) або властивість.
- Використовуйте стандартний атрибут, поки не потрібно контролювати доступ до нього.
 - Єдина відмінність від властивості – можливість автоматично викликати запрограмовані дії (custom actions), коли властивість отримується, встановлюється або видаляється.

Більш реалістичний приклад

Код для кешування веб-сторінки

```
from urllib.request import urlopen

class WebPage:
    def __init__(self, url):
        self.url = url
        self._content = None

    @property
    def content(self):
        if not self._content:
            print("Retrieving New Page...")
            self._content = urlopen(self.url).read()
        return self._content
```

- Поширена кастомна поведінка – кешування значення, яке складно обчислювати чи дорого шукати (потребує запит до мережі чи виконання запиту до бази даних).
 - Мета: зберегти значення локально, щоб уникати повторних викликів вартісних обчислень.
 - Це можна зробити за допомогою власного геттера властивості.
 - Під час першого отримання даних виконуємо обхід або обчислення.
 - Далі можемо локально кешувати значення до приватного атрибуту нашого об'єкту (або у відповідний об'єкт).
 - Наступного разу при запиті цього значення повернемо збережені дані.

Протестуємо код на отримувannya сторінки тільки одного разу

```
>>> import time
>>> webpage = WebPage("http://ccphillips.net/")
>>> now = time.time()
>>> content1 = webpage.content
Retrieving New Page...
>>> time.time() - now
22.43316888809204
>>> now = time.time()
>>> content2 = webpage.content
>>> time.time() - now
1.9266459941864014
>>> content2 == content1
True
```

- Власні (custom) геттери також корисні для атрибутів, які потрібно обчислити на льоту, використовуючи інші атрибути об'єкта.
 - Наприклад, необхідно обчислити середнє значення для списку цілих чисел:

```
class AverageList(list):
    @property
    def average(self):
        return sum(self) / len(self)
```

- Додаємо потрібну властивість у клас і маємо результат:

```
>>> a = AverageList([1,2,3,4])
>>> a.average
2.5
```

Об'єкти вищого рівня

- Об'єкти, які керують іншими об'єктами.
 - Атрибути деякого управляючого класу можуть звертатись до інших об'єктів, які виконують «видиму» роботу;
 - Поведінка такого класу *делегується* в потрібний момент іншим класам.
- Розглянемо програму, яка виконує пошук та заміну текстових файлів, збережених у стисненому ZIP-файлі.
 - Будуть потрібні об'єкти для представлення ZIP-файлу та кожного окремого текстового файлу (вони вже є в стандартній бібліотеці мови Python).
- Управляючий об'єкт забезпечуватиме з дії:
 - 1. Розархівування стисненого файлу.
 - 2. Виконання пошуку та заміни.
 - 3. Архівування (Zippping up) нових файлів.

- Клас ініціалізується назвою .zip-файлу та рядком для пошуку та заміни.
 - Створимо тимчасову папку для зберігання розархівованих файлів
 - Бібліотека pathlib (Python 3.4+) допомагає управляти файлами та папками.
 - Метод для делегування – перший, решта для завершеності

```
import sys
import shutil
import zipfile
```

```
from pathlib import Path
```

```
class ZipReplace:
    def __init__(self, filename, search_string, replace_string):
        self.filename = filename
        self.search_string = search_string
        self.replace_string = replace_string
        self.temp_directory = Path("unzipped-{}".format(
            filename))
```

```
def zip_find_replace(self):
    self.unzip_files()
    self.find_replace()
    self.zip_files()

def unzip_files(self):
    self.temp_directory.mkdir()
    with zipfile.ZipFile(self.filename) as zip:
        zip.extractall(str(self.temp_directory))

def find_replace(self):
    for filename in self.temp_directory.iterdir():
        with filename.open() as file:
            contents = file.read()
            contents = contents.replace(
                self.search_string, self.replace_string)

        with filename.open("w") as file:
            file.write(contents)
```

```
def zip_files(self):
    with zipfile.ZipFile(self.filename, 'w') as file:
        for filename in self.temp_directory.iterdir():
            file.write(str(filename), filename.name)
        shutil.rmtree(str(self.temp_directory))

if __name__ == "__main__":
    ZipReplace(*sys.argv[1:4]).zip_find_replace()
```


Don't Repeat Yourself (DRY)

- DRY-код – код, який простий у підтримці та не має дублювання.
 - Часто найпростіший розв'язок – виділити повторюваний код у функцію, яка приймає параметри для врахування можливих відмінностей.
 - Це не найкраще, але часто оптимальне рішення.
 - Наприклад, якщо маємо дві частини коду, що розпаковують ZIP-архів у дві різних папки, можна написати функцію, яка прийматиме параметр – шлях до папки призначення.
- *Після написання коду для заміни рядків у ZIP-архіві з текстовими файлами потім з'явилась потреба привести всі зображення в архіві до розміру 640 x 480.*
- *Перша ідея:* створити копію та змінити метод `find_replace()` на `scale_image()` або щось подібне.
 - *Ідея не найкраща:* що буде, якщо колись забажаємо також працювати з TAR-файлами?
 - Або захочемо використовувати гарантовано унікальну назву папки для тимчасових файлів.
 - Тоді доведеться вносити зміни у двох різних місцях!
- Розпочнемо з демонстрації вирішення проблеми на основі наслідування.

Спочатку змінимо початкову версію ZipReplase на суперклас з обробки загальних ZIP-файлів

```
import os
import shutil
import zipfile
from pathlib import Path
```

```
class ZipProcessor:
    def __init__(self, zipname):
        self.zipname = zipname

        self.temp_directory = Path("unzipped-{}".format(
            zipname[:-4]))

    def process_zip(self):
        self.unzip_files()
        self.process_files()
        self.zip_files()

    def unzip_files(self):
        self.temp_directory.mkdir()
        with zipfile.ZipFile(self.zipname) as zip:
            zip.extractall(str(self.temp_directory))

    def zip_files(self):
        with zipfile.ZipFile(self.zipname, 'w') as file:
            for filename in self.temp_directory.iterdir():
                file.write(str(filename), filename.name)
        shutil.rmtree(str(self.temp_directory))
```

```
from zip_processor import ZipProcessor
import sys
from PIL import Image

class ScaleZip(ZipProcessor):

    def process_files(self):
        '''Scale each image in the directory to 640x480'''
        for filename in self.temp_directory.iterdir():
            im = Image.open(str(filename))
            scaled = im.resize((640, 480))
            scaled.save(str(filename))

if __name__ == "__main__":
    ScaleZip(*sys.argv[1:4]).process_zip()
```

Моделюємо клас Document для використання в текстових редакторах

- Можна почати з `str` для представлення вмісту `Document`, але в Python рядки незмінювані.
 - Неможливо вставити символ в рядок без створення нового об'єкта-рядка.
 - Будемо використовувати список символів.
 - Також потрібно знати поточну позицію курсора у списку та бажано зберігати назву файлу з документом.
- Які методи передбачені?
 - Вставка, видалення, виділення символів, вирізання, копіювання, вставка та вибір, зберігання та закриття документу.
- Поставимо питання:
 - Компонувати клас з кількох базових Python-об'єктів, наприклад, шляхів до файлів, позицій курсору, списку символів?
 - Чи потрібно спеціальним чином прописувати ці дії в об'єктах?
 - Чи потрібні окремі класи для рядків та символів?

Найпростіша реалізація класу Document

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = 0
        self.filename = ''

    def insert(self, character):
        self.characters.insert(self.cursor, character)
        self.cursor += 1

    def delete(self):
        del self.characters[self.cursor]

    def save(self):
        with open(self.filename, 'w') as f:
            f.write(''.join(self.characters))

    def forward(self):
        self.cursor += 1

    def back(self):
        self.cursor -= 1
```

- Можна підключити букви та стрілки з клавіатури в ці методи, і документ стежитиме за поточним знаходженням курсора в тексті.
 - А якщо захочемо підключити клавіші Home і End?
 - Можна додати методи для пошуку символів переходу на новий рядок, проте код для кожної можливої дії зробить клас величезним!
 - Дії: перехід по словах, по реченнях, Page Up, Page Down, кінець рядка, початок з пробілу тощо
 - Можливо, краще відокремити ці методи в окремому об'єкті.
 - Введемо атрибут cursor, який володіє інформацією щодо позиції та може неї управляти.

```
class Cursor:
    def __init__(self, document):
        self.document = document
        self.position = 0

    def forward(self):
        self.position += 1

    def back(self):
        self.position -= 1

    def home(self):
        while self.document.characters[
            self.position-1] != '\n':
            self.position -= 1
        if self.position == 0:
            # Got to beginning of file before newline
            break

    def end(self):
        while self.position < len(self.document.characters
            ) and self.document.characters[
                self.position] != '\n':
            self.position += 1
```

- Код не дуже безпечний.

- Можна легко промахнутись у кінцевій позиції, а при порожньому файлі натиснути home.
- Клас Document позбавився двох методів, які перемістились у клас Cursor

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = Cursor(self)
        self.filename = ''

    def insert(self, character):
        self.characters.insert(self.cursor.position,
            character)
        self.cursor.forward()

    def delete(self):
        del self.characters[self.cursor.position]

    def save(self):
        f = open(self.filename, 'w')
        f.write(''.join(self.characters))
        f.close()
```

Протестуємо роботу для символу нового рядка:

- ```
>>> d = Document()
>>> d.insert('h')
>>> d.insert('e')
>>> d.insert('l')
>>> d.insert('l')
>>> d.insert('o')
>>> d.insert('\n')
>>> d.insert('w')
>>> d.insert('o')
>>> d.insert('r')
>>> d.insert('l')
>>> d.insert('d')
>>> d.cursor.home()
>>> d.insert("*")
>>> print("".join(d.characters))
hello
*world
```

# Додамо зручну властивість у клас Document

---

- З метою повернення повного рядка:

```
@property
def string(self):
 return "".join(self.characters)
```

- Тестування дещо спрощується:

- ```
>>> print(d.string)
```

```
hello
```

```
world
```

Розширимо код на випадок rich-тексту

- Можливі 2 підходи:
 - 1) додати фейкові символи до списку, що будуть діяти як інструкції. Наприклад, жирний текст починається та закінчується спеціальними символами;
 - 2) додати інформацію, що описуватиме форматування, для кожного символу. Реалізуємо такий підхід.
- Потрібен клас для символів.
 - Він матиме атрибут, що представляє символ, та три Boolean-атрибути (**bold**, *italic*, underlined).
- Методи? Наприклад, видалення чи копіювання символів.
 - Вони потрібні на рівні класу Document, оскільки змінюють список символів.
 - Без них потрібна структура даних, наприклад, кортеж / іменований кортеж.
- Також є `startswith()`, `strip()`, `find()`, `lower()` та ін.
 - Більшість таких методів працюють з рядками (>1 символу).
 - Якби Character субкласував str, розумно було б переозначити `__init__`, щоб викидати виняток при застосуванні до багатосимвольного рядка.
 - Оскільки дані отримані методи не будуть зараз застосовуватись до класу Character, можна і не використовувати наслідування.

Чи повинен Character бути класом?

- Спеціальний метод `__str__()` у класі може допомогти в представленні символів.
 - Використовується у функціях обробки рядків, таких як `print()` та `str()`, щоб конвертувати об'єкт будь-якого класу в рядок.
 - За замовчуванням виводить назву модуля, класу та його адресу в пам'яті.
 - Можна переозначити:
 - Можемо використати в якості префіксів спеціальні символи, щоб показати текст **bold**, *italic* або underlined

```
class Character:
    def __init__(self, character,
                  bold=False, italic=False, underline=False):
        assert len(character) == 1
        self.character = character
        self.bold = bold
        self.italic = italic
        self.underline = underline

    def __str__(self):
        bold = "*" if self.bold else ''
        italic = "/" if self.italic else ''
        underline = "_" if self.underline else ''
        return bold + italic + underline + self.character
```

Незначні модифікації класів Document і Cursor для роботи з цим класом

- У клас Document додамо 2 рядка на початок методу insert()
def insert(self, character):
 if not hasattr(character, 'character'):
 character = Character(character)
- Основна мета: перевірити, передавався символ з Character чи str.
 - Якщо рядок, він огортається у клас Character, щоб усі об'єкти в списку були екземплярами Character.
 - Проте через качину типізацію може використовуватись клас, що не буде описувати ні рядок, ні символ.
 - Якщо об'єкт має символний атрибут, вважаємо його Character-подібним.
 - Інакше розглядаємо його "str-подібним" та обгортаємо в Character.

Також потрібно змінити властивість `string` у класі `Document`, щоб приймати нові символні значення

- Слід викликати `str()` для кожного символу перед злиттям (`join`):

```
@property
def string(self):
    return "".join((str(c) for c in self.characters))
```

- Потрібно перевірити `Character.character` замість символу зі збереженого до цього рядка в функціях `home()` та `end()` при пошуку символу нового рядка:

```
def home(self):
    while self.document.characters[
        self.position-1].character != '\n':
        self.position -= 1
    if self.position == 0:
        # Got to beginning of file before newline
        break

def end(self):
    while self.position < len(
        self.document.characters) and \
        self.document.characters[
            self.position
        ].character != '\n':
        self.position += 1
```

Тестуємо роботу

```
■ >>> d = Document()
>>> d.insert('h')
>>> d.insert('e')
>>> d.insert(Character('l', bold=True))
>>> d.insert(Character('l', bold=True))
>>> d.insert('o')
>>> d.insert('\n')
>>> d.insert(Character('w', italic=True))
>>> d.insert(Character('o', italic=True))
>>> d.insert(Character('r', underline=True))
>>> d.insert('l')
>>> d.insert('d')
>>> print(d.string)
he*l*lo
/w/o_rld
>>> d.cursor.home()
>>> d.delete()
>>> d.insert('W')
>>> print(d.string)
he*l*lo
W/o_rld
>>> d.characters[0].underline = True
>>> print(d.string)
he*l*lo
W/o_rld
```



ДЯКУЮ ЗА УВАГУ!

Наступне питання: Об'єктно-орієнтоване програмування в дії
