

Лабораторна робота №1

Віртуальні середовища в Python

Короткі теоретичні відомості №2

1. Віртуальне середовище (virtual environments)

Python програми виконуються за умови наявності інтерпретатора та всіх необхідних для роботи програми пакунків. Інтерпретатор та всі необхідних для роботи програми пакунки формують середовище в якому виконується програма.

Віртуальне середовище (virtual environment) використовується для ізоляції програми (проєкту) від поточного стану системи, що дозволяє використовувати різні версії інтерпретаторів та різні версії бібліотек й пакунків. Віртуальне середовище дозволяє встановлювати та керувати пакунками без прав адміністратора та уникати конфліктів з системним менеджером пакунків. Розробник програмного забезпечення обов'язково повинен вміти використовувати віртуальні середовища та завжди його використовувати в своїй роботі.

Для кожного проєкту можна створити окреме віртуальне середовище і в цьому середовищі встановити тільки ті пакунки, що використовуються у цьому проєкті. Встановлені таким чином пакунки не будуть мати ніякого впливу на інші проєкти та й на саму систему.

Python розробник для створення віртуальних середовищ може використовувати модуль `venv`, або засіб [virtualenv](#), що не входить у стандартну бібліотеку (засіб стороннього розробника).

Призначення модуля [venv](#) – описано в документації наступним чином. The [venv](#) module provides support for creating lightweight “virtual environments” with their own site directories, optionally isolated from system site directories. Починаючи з версії Python 3.3 модуль [venv](#) входить до складу стандартної бібліотеки й готовий для використання.

Користувачам операційних систем Debian/Ubuntu рекомендують для забезпечення працездатності цього модуля додатково виконати:

```
$ sudo apt-get install python3-venv
```

Створення віртуального середовища

Для кожного проєкту можна створити окреме віртуальне середовище. Віртуальне середовище буде зберігатися в окремій теці. Ім'я середовища буде також й ім'ям цієї теки. Зазвичай середовище називають `venv` або `env`.

Для створення віртуального середовища з назвою `venv` (ім'я віртуального середовища може бути й іншим) потрібно перейти у теку проєкту (`cd path/to/project`) та виконати:

```
$ python3 -m venv venv
> python3 -m venv venv
```

В результаті виконання цієї команди у теці проєкту буде створено теку `venv`, що буде містити теки `bin`, `lib` та `include`. У теці `bin` знаходяться виконувані файли разом із `python`. Тека `lib` містить код встановлених пакунків. Можна пересвідчитися, що в середовищі зараз встановлено тільки необхідні пакунки `pip`, `setuptools` та `easy_install`. Тека `include` містить файли заголовків C.

Після створення віртуального середовища рекомендується оновити версії встановлених пакунків та встановити пакунок `wheel`. В операційній системі Ubuntu(Linux) потрібно виконати:

```
$ venv/bin/python -m pip install --upgrade pip setuptools wheel
```

В операційній системі Windows потрібно виконати:

```
> venv\Scripts\python -m pip install --upgrade pip setuptools wheel
```

Тепер віртуальне середовище готове для роботи. Віртуальне середовище знаходиться у теці проєкту. Віртуальне середовище може знаходитися в довільному місці, але рекомендується розміщати його або у спеціально виділеному місці (використовують засіб `virtualenvwrapper` для організації роботи з такими середовищами) або у теці проєкту (найбільш поширена практика). При розміщенні віртуального середовища в теці проєкту потрібно про це пам'ятати. Коли для роботи з проєктом буде використовуватися система контролю версій Git то потрібно буде виконати додаткові дії (обов'язково потрібно у файл `.gitignore` додати рядок з ім'ям віртуального середовища).

Робота у віртуальному середовищі.

Для роботи у віртуальному середовищі потрібно його активувати. В операційній системі Ubuntu(Linux) потрібно виконати:

```
$ source /path/to/venv/bin/activate
```

В операційній системі Windows потрібно виконати:

```
> C:\path\to\venv\Scripts\activate
```

Після активації віртуального середовища у командному рядку з'являється префікс `(venv)` з іменем середовища, який вказує яке середовище зараз активне.

Для встановлення та керування пакунками використовується спеціальна програм `pip`. Програму `pip` називають інсталятором Python пакунків, який дозволяє встановлювати пакунки з Python Package Index та подібних сховищ. Наприклад, для встановлення додаткового пакунку `folium` у віртуальне середовище потрібно виконати в операційній системі Ubuntu(Linux):

```
(venv)$ pip install folium
```

В операційній системі Windows потрібно виконати:

```
(venv)>pip install folium
```

Потрібно зауважити, що формат команди `pip install ...` прийнятний за умови роботи у активованому віртуальному середовищі. У всіх інших випадках рекомендується використовувати формат `python -m pip install ...`, що дозволяє контролювати версію інтерпретатора та інсталятора й відповідно місце встановлення пакунку.

Для завершення роботи з віртуальним середовищем його потрібно деактивувати. В операційній системі Ubuntu(Linux) потрібно виконати:

```
(env)$ deactivate
```

В операційній системі Windows потрібно виконати:

```
(venv)>deactivate
```

Працювати з віртуальним середовищем можна й без його активації, але для цього потрібно вказувати повні шляхи до виконуваних файлів. Для виконання попередніх дій в операційній системі Ubuntu(Linux) потрібно виконати:

```
$ /path/to/venv/bin/pip install folium
```

В операційній системі Windows потрібно виконати:

```
> C:\path\to\venv\Scripts\pip install folium
```

Інстальатор пакунків `pip` дозволяє не тільки встановлювати але й видаляти пакунки. Ця програма також дозволяє створити файл `requirements.txt`, який містить всі залежності проєкту. Залежності проєкту це необхідні для його роботи бібліотеки, пакунки модулів та окремі модулі. Для автоматичного створення файлу `requirements.txt` потрібно виконати у віртуальному середовищі:

```
$ venv/bin/pip freeze > requirements.txt
```

За допомогою `pip` можна також оновлювати пакунки до останніх версій:

```
pip install --upgrade folium
```

Встановлювати певні версії пакунків:

```
pip install folium==0.12.1
```

```
pip install folium>=0.12.1
```

Встановлювати всі залежності проєкту:

```
pip install -r requirements.txt
```

Видаляти пакунки:

```
pip uninstall folium
```

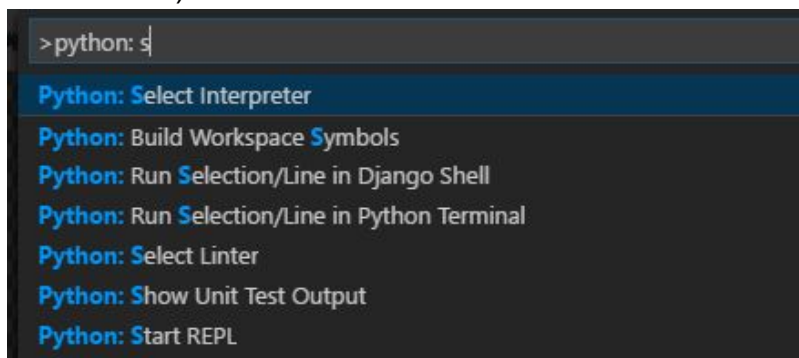
Здійснювати пошук пакунків:

```
pip search text scraping
```

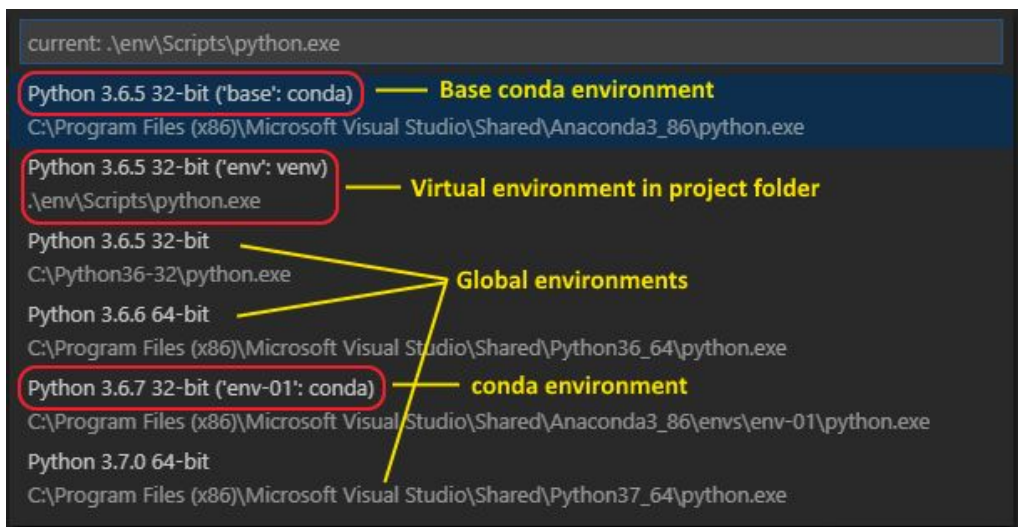
Робота з віртуальними середовищами в Visual Studio Code.

По замовчуванню у Visual Studio Code розширення Python використовує перший знайдений інтерпретатор. Якщо інтерпретатор не знайдено то користувач отримує про це повідомлення.

Для того щоб обрати інший інтерпретатор (середовище) потрібно скористатися командою **Python: Select Interpreter** у **Command Palette** (пункт меню View або Ctrl+Shift+P).

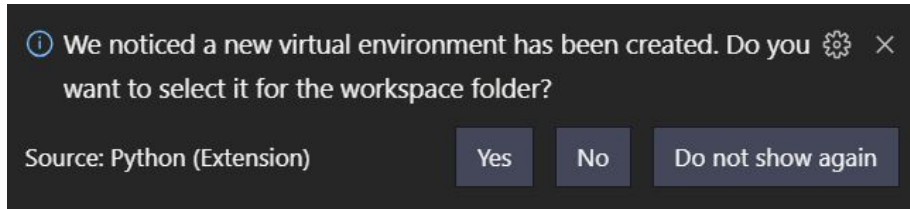


За потреби змінити середовище можна в будь який момент роботи над проєктом. Такі дії дозволяють перевірити працездатність проєкту чи його частин з використанням різних інтерпретаторів та версій пакунків.



Команда **Python: Select Interpreter** дозволяє переглянути список всіх доступних глобальних середовищ, віртуальних середовищ та conda середовищ.

Для створення нового віртуального середовища потрібно в командному рядку виконати дії, що вже було описані вище. При створенні віртуального середовища можна буде обрати чи буде це середовище відображатися робочому простору (**workspace**).



Після створення віртуального середовища в теці проєкту буде створена тека віртуального середовища й у переліку середовищ буде зображатися нове середовище. Порядок роботи у віртуальному середовищі не відрізняється від того що вже був описаний вище. За потреби, будь-яке з віртуальних середовищ можна використовувати в інших проєктах.

Література

<https://code.visualstudio.com/docs/python/python-tutorial>
<https://chriswarrick.com/blog/2018/09/04/python-virtual-environments/>
<https://dev.to/codemouse92/dead-simple-python-virtual-environments-and-pip-5b56>
<https://realpython.com/effective-python-environment/>

Лабораторна робота №1 Система контролю версій Git, та сховище GitHub Короткі теоретичні відомості

Сучасна розробка програмного забезпечення це технологічний процес до якого залучено як програмістів так й надано технічні засоби, що забезпечують цей процес.

Розроблення п1рограмного забезпечення це тривалий, ітераційний процес. Програма до того як буде надана користувачу зазнає неодоразових змін та трансформацій. Користувач отримує для використання певну, робочу версію програми, але й в подальшому в цей код вносяться зміни, виправлення, виходять нові версії програми.

Системи контролю версій це програмне забезпечення що зараз обов'язково використовується в розробці програмного забезпечення.

Системою Контролю Версій (VCS) – це програмний інструмент, основне завдання якого – зберігання коду та історії змін до даного коду. Код, історія змін цього коду та всі версії коду зберігаються в репозиторії .

Система контролю версій дозволяє:

- надійно зберігати код;
- запам'ятовувати історію змін у коді
- зберігати інформацію про зміни у коді (автор змін, що змінено, коли змінено);
- повернутися до будь-якої попередньої версії коду в будь-який момент;
- з легкістю об'єднувати зміни різних версій, станів та розробників;
- розробляти проект командою розробників, що одночасно працюють над одними і тими ж частинами коду;
- публікувати код для зовнішніх користувачі;
- тощо.



1. Централізовані системи контролю версій (Version Control System, VCS)

Основна ознака централізованих систем контролю версій – для проекту може бути лише одине центральне сховище (репозиторій) коду, де зберігаються всі версії коду й куди надсилають свій код (commit, комітять) розробники. Репозиторій зазвичай знаходиться на зовнішньому сервері й при відсутності доступу до нього (локальної мережі, інтернету) зміни вносити неможливо.

Найвідоміші системи такого типу: SVN, CSV. Зараз ці системи використовуються рідко, а якщо й використовуються то SVN.

2. Розподілені (децентралізовані) системи контролю версій (Distributed Version Control System, DVCS)

Розподілені (децентралізовані) системи контролю версій дозволяють окрім центрального репозиторія, кожному розробнику мати власний локальний репозиторій, або й декілька копій репозиторію й працювати з кожним із них окремо. Для збереження регулярних змін не потрібен доступ до мережі чи інтернету. Можна вносити зміни та зберігати їх у локальному репозиторію, а пізніше надіслати у віддалений (remote) репозиторій.

Найбільш поширені системи контролю версій це Git та Mercurial. Для зручності збереження віддалених репозиторіїв користувачі Git користуються онлайн сервісами GitHub (github.com, публічні та приватні репозиторії) та GitLab (gitlab.com). Користувачі Mercurial використовують сервіс bitbucket.com.

Перевага розподілених (децентралізованих) систем контролю версій полягає в тому, що навіть у випадку ураження віддаленого репозиторію (як в от цій історії) можна відновити весь код (проект).

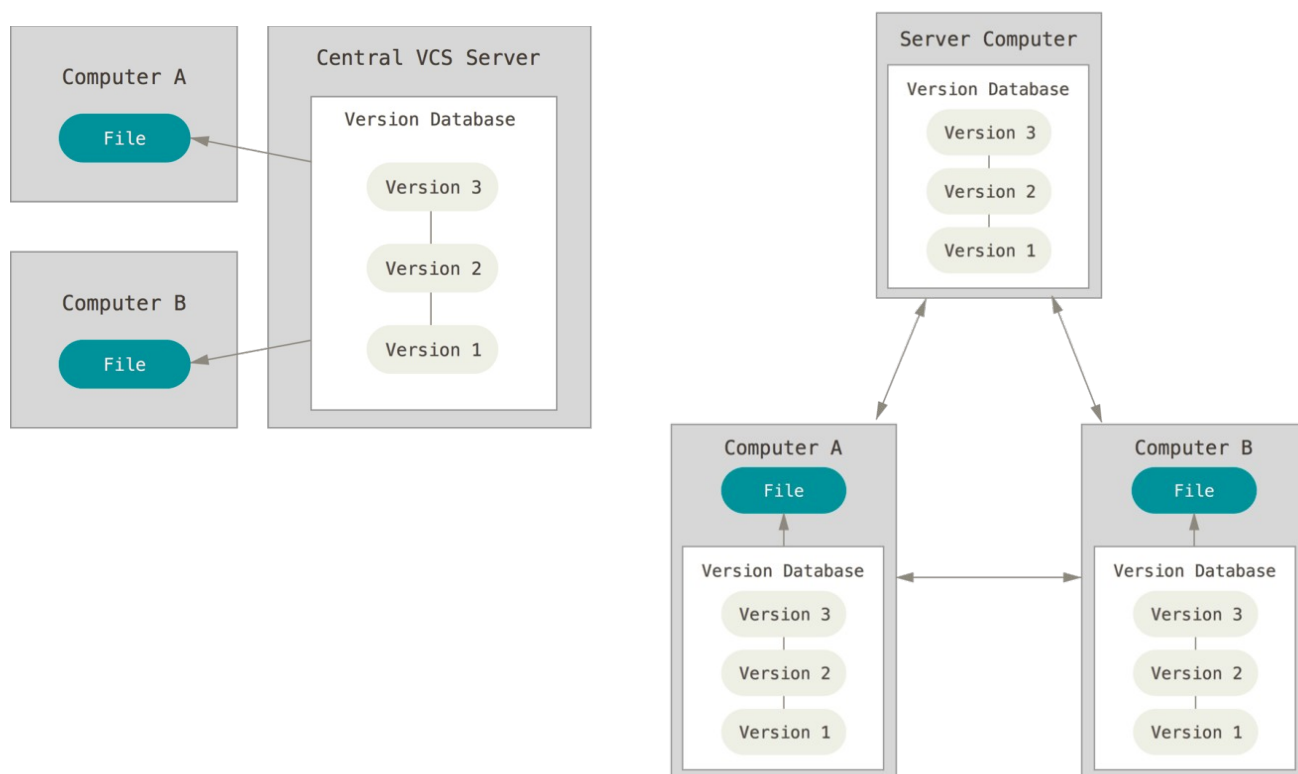


Рис.1. Різниця між централізованими та розподіленими системами контролю версій (<https://git-scm.com/book/uk/v2>)

3. Концепція та термінологія Git

Git має три основних стани, в яких можуть перебувати ваші файли: *збережений у коміті* (committed), *змінений* (modified) та *індексований* (staged):

- Збережений у коміті означає, що дані безпечно збережено в локальній базі даних системи.
- Змінений означає, що у файл внесено редагування, які ще не збережено в базі даних системи.
- Індексований стан це коли позначено змінений файл у поточній версії, й ці зміни ввійдуть до наступного коміту (знімку).

З цього випливають три основні частини проекту під управлінням Git: директорія Git, робоча тека та індекс.

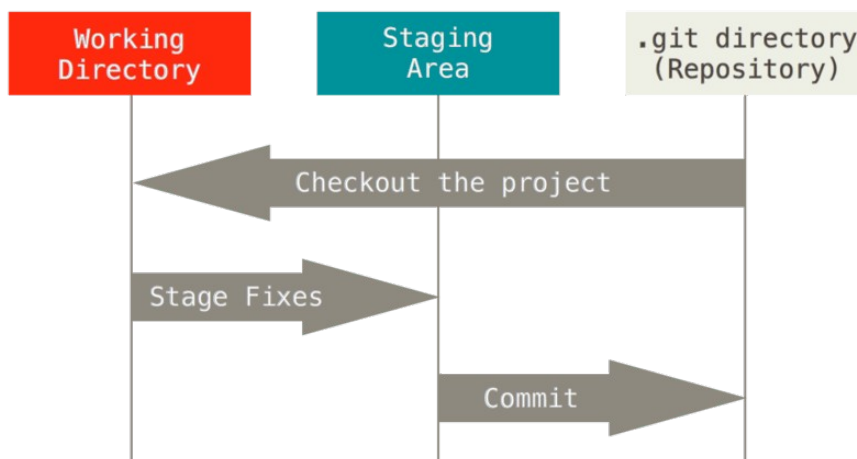


Рис.2. Робоча директорія, індекс та директорія Git (<https://git-scm.com/book/uk/v2>)

Якщо в проєкті використовується віддалене сховище то схематично усі стани коду від локальної, робочої теки й аж до кінцевого місця зберігання коду, яким будуть користуватися всі розробники проєкту будуть виглядати наступним чином:

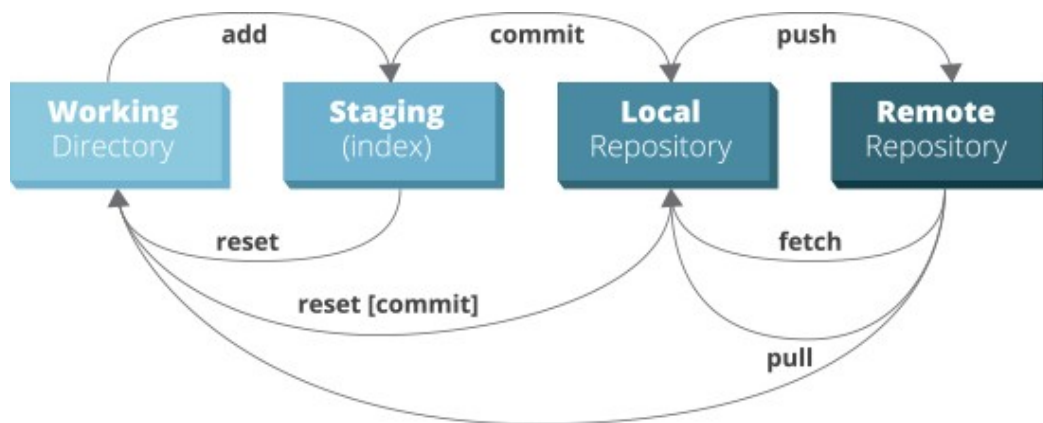


Рис. 3. Чотири етапи коду: Working Directory, Staging (Index), Local, та Remote репозиторії (<https://www.jrebel.com/blog/git-cheat-sheet>)

Код розробляється в локальній робочій теці (Working Directory, Workspace). Зміни зберігаються у проміжний стан (index або staging), звідки код потрапляє у локальний репозиторій (Local Repository). Далі код можна помістити у віддалений репозиторій (Remote Repository) й всі інші розробники будуть мати доступ до внесених у код змін.

Основні терміни що використовуються в системі контролю версій Git:

Working Directory (робоча тека) – локальна тека у файловій системі, яка відображає певний стан коду в репозиторії та локальні зміни (якщо такі є).

Commit (коміт) – запис змін коду у локальний репозиторій.

Branch (гілка) – версія коду, яку створюють всередині репозиторію, й де ведуть розробку до моменту об'єднання (merge) змін даної версії коду із основною гілкою (master, main). Гілки використовують для паралельної роботи над окремими завданнями. Основна гілка у будь-якому git репозиторії – це *master(main)*.

Merge (злиття) – об'єднання змін з однієї гілки коду в іншу.

Pull – отримання останніх змін з віддаленого репозиторію у локальний репозиторій.

Push – надіслати зміни локального репозиторію у віддалений репозиторій.

Remote – віддалений репозиторій. Проект (код) може мати багато копій на локальних та зовнішніх серверах. Кожна копія, окрім локальної, називається віддаленим репозиторієм (remote).

Stage або Index (індекс) – місце файлів з робочої теки перед внесенням змін робочої теки у локальний репозиторій. Коміт містить у собі лише ті зміни, що є в індексі.

Локальний репозиторій – локальна копія репозиторію, у яку безпосередньо йдуть коміти з робочої теки. Для роботи з даним репозиторієм не потрібен доступ до мережі чи інтернету.

Git HEAD – вказівник на коміт в репозиторії, на якому зараз перебуває локальний репозиторій і гілка у цьому репозиторії. Вказівник вказує на останній коміт, але його можна “переміщати” між комітами, гілками й навіть тегами.

Tag (тег) – зафіксований стану коду для подальшого використання його в релізах. Код фіксується зазвичай під тегом з певною цифровою назвою (1.0, 2.0.1, 3.0alpha). Нові зміни у теги не записуються.

4. Встановлення Git

На офіційній сторінці системи контролю версій Git надано рекомендації по встановленню Git на всі типи операційних систем (<https://git-scm.com/book/uk/v2/Вступ-Інсталяція-Git>):

- [Інстальюємо](#) Git на macOS
- [Інстальюємо](#) Git на Windows
- [Інстальюємо](#) Git на Linux

5. Основні команди Git

Конфігурація Git

Мінімальна конфігурація Git репозиторію складається з додавання власного імені, яке буде згадуватися в кожному коміті:

```
$ git config --global user.name "Andriy Romanyuk"
```

та електронної адреси:

```
$ git config --global user.email "a.romaniuk@ucu.edu.ua"
```

Опція “-global” встановлює вказані вище дані глобально для усіх репозиторіїв на комп’ютері. Якщо для певного репозиторію потрібні інші налаштування, то у теці репозиторію потрібно виконати аналогічні команди, але вже без опції **global** й налаштування будуть зроблені локально лише для цього поточного репозиторію.

Також можна налаштувати список типів файлів, які git повинен ігнорувати. Файли, тип яких вказано у списку не будуть додаватися до репозиторію. Для цього можна створити файл **.gitignore_global**, наприклад з наступним вмістом:

```
Env
.env
venv
.venv
__pycache__
*.pyc
*~
*.egg-info
*.mo
*.egg
```

Та додати ці вказівки у конфігурацію Git:

```
$ git config --global core.excludesfile /path/to/.gitignore_global
```

Файл **.gitignore** можна створювати й в локальному репозиторію за допомогою команди:

```
$ touch .gitignore
```

Клонування існуючого репозиторія

Якщо репозиторій з кодом вже існує й відоме посилання на цей репозиторій а він потрібен для роботи то його можна клонувати:

```
$ git clone https://github.com/vipod/wxpython_calculator.git
```

В результаті виконання цієї команди буде створено локальний репозиторій (клон), робочу теку та індекс.

Створення нового репозиторія

Для створення нового репозиторія потрібно створити локальну теку, зайти у цю теку та ініціалізувати її Git репозиторій:

```
$ mkdir firstrepo
$ cd firstrepo
$ git init
```

Ще один спосіб створити репозиторій це створити віддалений репозиторій на github.com та клонувати його.

Додавання локальних змін у локальний репозиторій

Рекомендується завжди перед тим, як виконувати будь-які дії в репозиторії виконувати команду **status**:

```
$ git status
```

```
# On branch main
# No commits yet
# Untracked files:
#   (use "git add <file>..." to update what will be committed)
#
#       .venv/
#       LICENSE
#       README.md
#       firstmap.py
#       requirements.txt
```


nothing added to commit but untracked files present (use "git add" to track)

Результат виконання команди це інформація про поточну гілку коду. У даному випадку це гілка – main. Також надано список змінених, нових та видалених файлів й надано підказки, що і як з кожним із файлів можна робити в подальшому.

Перед комітом потрібно додати всі файли у Індекс (*- всі файли):

```
$ git add *
```

Команда `git add` рекурсивно додає усі файли та вкладені теки даної теки.

Також можна додавати окремі файли:

```
$ git add data.txt
```

Після додавання файлів у Індекс можна виконати коміт у лопшеєшекальний репозиторій:

```
$ git commit -m 'initial code version'
```

За допомогою ключа `-m` можна відразу до команди додати текстове повідомлення в якому описуються зроблені зміни, які будуть збережені в репозиторії. Якщо опустити цей ключ то буде відкрито текстовий редактор й виконання коміту завершиться після додавання та збереження текстового повідомлення.

Команда `show` дозволяє отримати додаткову інформацію про коміт.

```
$ git show
```

Після виконання цієї команди на екрані буде зображено ідентифікатор цього коміта (SHA каталогу верхнього рівня репозитарія), який називають SHA за яким ідентифікуються внесені зміни, автора змін та самі зміни – що та у яких файлах змінено.

SHA обчислюється за допомогою хеш функції і є унікальним. Git використовує власний алгоритм хешування за допомогою якого індексуються всі файли, теки. При змінах у файлі змінюється його хеш та хеш теки в якій цей файл знаходиться.

Якщо потрібно відновити випадково видалений файл у робочій теці то потрібно виконати наступну команду:

```
$ git checkout -- data.txt
```

Для відмови від зроблених локальних змін та повернення до стану, як в репозиторії потрібно виконати наступну команду:

```
$ git checkout
```

Якщо було видалено файл, то ці зміни також треба занести в Індекс окремою командою, інакше локальний репозиторій проігнорує видалення файлу в робочій теці:

```
$ git rm data.txt
```

Збереження змін локального репозиторію у віддалений репозиторій

Для того щоб інші розробники змогли використовувати версію коду, що збережена в локальному репозиторію ці зміни потрібно надіслати у зовнішній репозиторій (синхронізувати локальний та зовнішній репозиторій).

За замовчуванням віддалений репозиторій називається `origin`. Якщо локальний репозиторій було створено шляхом клонування існуючого репозиторія, то `origin` уже налаштований разом з гілкою коду, звідки виконувалося клонування. У такому випадку достатньо виконати наступну команду для синхронізації локального репозиторія із віддаленим репозиторієм:

```
$ git push origin main
```

Ця команда закидає усі коміти локального репозиторію поточної гілки у віддалений репозиторій під назвою `origin` в гілку `main` (`main` - основна гілка, раніше мала назву `master`). Зазвичай використовують скорочений формат цієї команди:

```
$ git push
```

Якщо було створено новий локальний репозиторій, то треба спочатку додати віддалений репозиторій й вказати його адресу:

```
$ git remote add origin https://github.com/<user_name>/firstrepo.git
```

Локальний репозиторій отримав інформацію що за адресою `https://github.com/<user_name>/firstrepo.git` знаходиться віддалений репозиторій під назвою `origin`. Тепер можна додати зміни у цей репозиторій у гілку `main`:

```
$ git push origin main
```

Оновлення локального репозиторія та робочої теки

Якщо у віддаленому репозиторію відбулися зміни то потрібно ці зміни отримати у локальний репозиторій щоб працювати з актуальною версією коду.

Для цього можна скористатися наступною командою:

```
$ git pull origin main
```

Виконання цієї команди складається із двох етапів:

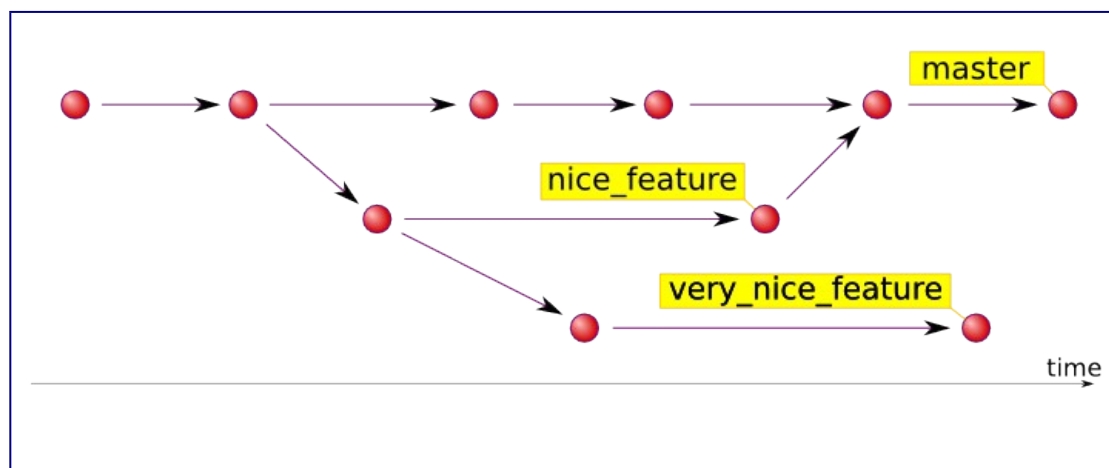
- синхронізація віддаленого репозиторія із локальним (команда `$ git fetch`)
- об'єднання (злиття) змін, що прийшли в локальний репозиторій, із робочою текою (команда `$ git merge`)

Ці етапи можна виконувати, окремо за допомогою відповідних команд.

При злитті (об'єднанні) можуть виникати конфлікти, які доводиться розбирати вручну та додавати виправлені файли в Індекс, а потім як новий спеціальний коміт.

6. Робота з гілками

Гілки – це інструмент для паралельної розробки в проєкті. За допомогою гілок можна розробляти декілька версій коду а по завершенні розробки об'єднувати все в один проєкт. Робота в гілці виконується з використанням тих самих команд для додавання до індексу, збереження змін в репозиторій. На рисунку показано можуть використовуватися гілки.



В певний момент часу (коміту) відгалужується окрема версія коду (гілка), розробка йде ізольовано в різних гілках (виконуються коміти та `push`), а коли завершено роботу в гілці виконується злиття (об'єднання) назад у головну гілку. Таким чином можна працювати декільком розробникам й не заважати один одному.

Щоб отримати список усіх гілок локального репозиторія:

```
$ git branch
```

```
* master
```

Щоб отримати список усіх гілок локального та віддалених репозиторіїв:

```
$ git branch -av
```

```
* master e48d225 fix typo
```

```
remotes/origin/HEAD -> origin/master
```

```
remotes/origin/master e48d225 fix typo
```

Гілки у віддалених репозиторіях позначені як `remotes/origin/`. Зірочкою позначена поточна гілка, що використовується в робочій теці.

Створюємо нової гілки `docs`:

```
$ git branch docs
```

Переключення у існуючу гілку:

```
$ git checkout docs
```

Створення нової гілки та переключення у цю гілку:

```
$ git checkout -b new_docs
```

Видалення гілки:

```
$ git checkout -B new_docs
```

Додавання гілки з локального репозиторія у віддалений репозиторій:

```
$ git push origin docs
```

Отримання нових змін з віддаленого репозиторію у локальний та оновлення робочої теки:

```
$ git pull origin new_docs
```

Видалення гілки з віддаленого репозиторію:

```
$ git push origin :new_docs
```

Коли роботу на кодом в окремій гілці завершено потрібно об'єднати цю частину коду з основним поточним кодом проекту (зазвичай гілкою `main(master)`). Об'єднання виконується після переключення у гілку `main` командою `merge`:

```
$ git checkout main
```

```
$ git merge docs
```

Якщо при об'єднанні виникли конфлікти, тоді потрібно вручну правити файли до потрібного вигляду та маркувати їх як виправлені:

```
$ git add file_with_conflict.txt
```

Робити окремий коміт:

```
$ git commit -m 'merge conflicts'
```

Після чого вже виконувати синхронізацію змін локального репозиторію у віддалений:

```
$ git push origin master
```

Якщо при об'єднанні гілок конфліктів не виникає, тоді додаткового коміту не потрібно, а лише одразу виконати

```
$ git push.
```

7. Робота з Тегами

Кожного разу, коли поточна версія коду повністю готова потрібно робити новий реліз коду. При підготовці релізу вартує користуватися Тегами. Тег – це зафіксована незмінювана версія коду позначена певним іменем версії (напр. 1.0.2).

Щоб переглянути список існуючих тегів:

```
$ git tag -l
```

Щоб створити новий тег (анотований):

```
$ git tag -a 1.0.1 -m 'release 1.0.1'
```

Дана команда створює новий тег, дає йому назву 1.0.0 та додає супроводжуючий коментар `'release 1.0.1'` – коментар це також обов'язковий аргумент.

Щоб видалити існуючий тег з локального репозиторію:

```
$ git tag -d 1.0.1
```

Щоб додати усі теги та синхронізувати їх з локального на віддалений репозиторій:

```
$ git push --tags
```

Тобто команда `git tag` створює тег лише в локальному репозиторії, а вже `push` дозволяє додати теги у зовнішній репозиторій.

8. Додаткові корисні команди та матеріали

Вся історія комітів розробників зберігається у репозиторії. Можна скористатись наступною командою, яка надасть розширений запис усіх змін:

```
$ git log -p
```

Для того, щоб навчитися більш вправно працювати в Git потрібно пройти один з онлайн курсів:

- <https://www.codecademy.com/learn/learn-git>;
- <https://try.github.io/levels/1/challenges/1>;

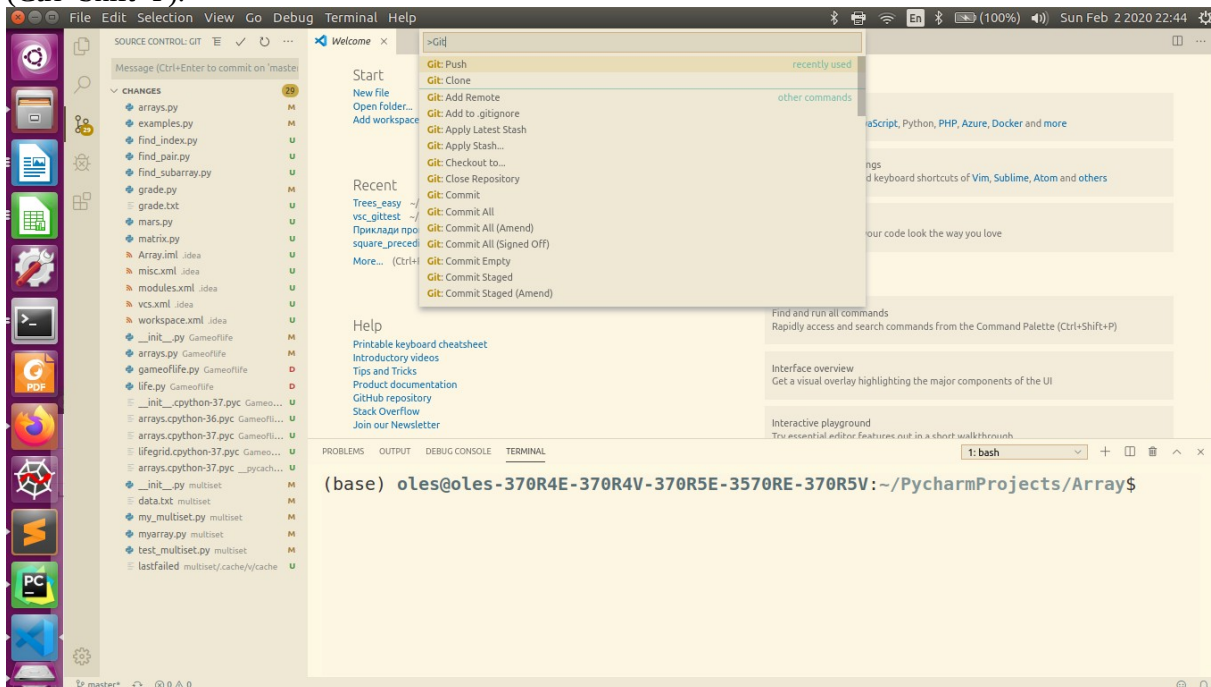
й тоді можна буде вправно працювати з гілками (branches), виконувати злиття (merge) та зрозуміти наступний жарт.



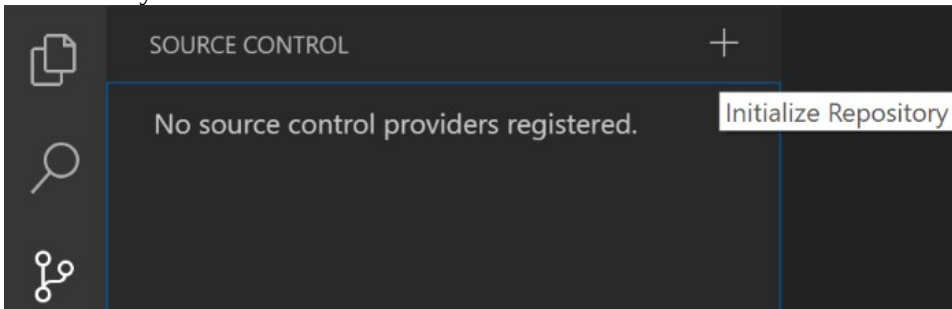
9. Робота з Git та GitHub в VS Code

У VS Code інтегрована підтримка систем контролю версій та за замовчуванням реалізована підтримка Git. Для використання інших засобів контролю версій потрібно встановити відповідні розширення.

Для роботи з Git можна використовувати закладку **Source Control**. Для відкриття цієї закладки можна використовувати гарячі клавіші (Ctrl+Shift+G), або використовувати Git команди з **Command Palette** (Ctrl+Shift+P).

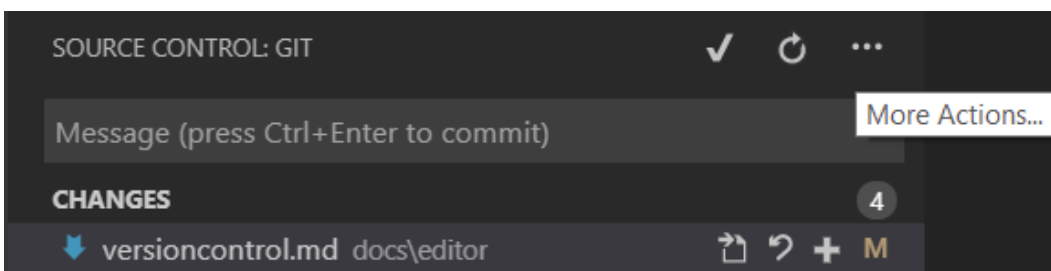


Наприклад, для створення репозиторію потрібно скористатися командою **Git: Initialize Repository** або натиснути на символ плюс.



Інформація про всі зміни в репозиторію зображаються в закладці **Source control** у розділі **Changes**. Для збереження змін потрібно виконати команду індексування **Git: Staging** (git add) а для скасування змін **Git:unstaging** (git reset) після чого можна виконати **Git: Commit**. Ці дії також можна виконувати в закладці **Source control** за допомогою відповідних графічних примітивів та контекстних меню.

Інші потрібні **Commit** дії можна виконати вибравши відповідні пункти з меню ... **More Actions**.



Для клонування репозиторію потрібно скористатися командою **Git: Clone** в **Command Palette** (Ctrl+Shift+P) та вказати розміщення цього репозиторію (у випадку використання сховища GitHub потрібно вказати URL репозиторію) та вибрати місце на диску де потрібно розмістити цей репозиторій.

10. Основні кроки при роботі з Git

При роботі з Git вартує віддавати перевагу використанню інтерфейсу командного рядка.

Нехай розпочато роботу над проєктом й потрібно використовувати систему контролю версій для нього. Спочатку потрібно створити репозиторій. Для цього в командному рядку виконується:

Крок 1

```
$ git init
```

Після виконання цієї команди буде створено локальний Git репозиторій. У теці з вашим проєктом з'явиться прихована тека з назвою .git.

Крок 2

```
$ git add *
```

Після виконання цієї команди до репозиторію будуть проіндексовані й додані всі файли проєкту.

Крок 3

```
$ git commit -m "Initial commit"
```

Записуємо зміни у локальний репозиторій.

Крок 4

```
$ git remote add origin https://github.com/<user_name>/MyProject.git
```

Приєднання (лінкування) віддаленого репозиторію. Проєкт MyProject має бути попередньо створено на ресурсі GitHub.

Крок 5

```
$ git push -u origin main
```

Синхронізація з віддаленим репозиторієм. Можна також використати git push.

В подальшій роботі над проєктом (внесення нових змін в репозиторій), достатньо повторювати кроки 2, 3 та 5 для збереження всіх змін.

Літєратура

<https://git-scm.com/book/uk/v2>

<http://www.vitaliy podoba.com/2014/06/git-basics/>

<https://realpython.com/python-git-github-intro/>

<https://code.visualstudio.com/docs/editor/versioncontrol>

<https://githowto.com>

<https://ndpsoftware.com/git-cheatsheet.html>

<https://developers.redhat.com/cheat-sheets/git>

<http://www.cheat-sheets.org/saved-copy/git-cheat-sheet.pdf>

<https://education.github.com/git-cheat-sheet-education.pdf>

Лабораторна робота №1

Структура Python проекту.

Короткі теоретичні відомості № 1

1. Структура Python проекту.

Python дозволяє розробляти програми (застосунки) різного типу. Розробнику не завжди легко обрати відповідну структуру проекту для того чи іншого застосунку. Структура проекту це структура тек, які містять модулі, пакунки модулів та інші файли проекту.

Проект – тека, яку називають пакунком найвищого рівня і яка містить пакунки та модулі.

Паунок (в проєкті) – тека яка містить модулі та/або пакунки.

Паунок (вбудований) – паунок, який розповсюджується разом з Python і міститься у стандартній бібліотеці

Паунок (стороннього розробника) – паунок, який можна встановити за допомогою інсталятора пакунків `pip` і після цього імпортувати.

Модуль – файл з розширенням `.py`, який можна запустити на виконання, або імпортувати.

Структура проекту для створення застосунку командного рядка (CLI apps) та структура проекту web застосунку (web application) найчастіше будуть різними. Також потрібно враховувати, що проект з одним модулем (проста програма, скрипт, script) відрізняється від програми, яка передбачає створення модуля чи пакунка модулів, які можуть бути інстальовані (installable packages).

В загальному структуру проекту рекомендують формувати згідно з наступними простими правилами:

- якщо проєкт це єдиний файл (модуль із сирцевим кодом) то цей модуль повинен знаходитись в проєкті на найвищому рівні.
- якщо проєкт містить тести то проєкт повинен містити теку `tests` й там повинні знаходитися тести навіть, якщо проєкт складається з єдиного модуля. Альтернативним варіантом може бути розміщення тестів разом із модулями з кодом, але якщо модулі знаходяться в окремих вкладених теках.
- якщо для запуску проекту використовується виконуваний скрипт то скрипт повинен бути у теці `bin`. Якщо це Python скрипт то файл повинен бути без розширення `.py`.
- якщо проєкт містить багато модулів з сирцевим кодом то потрібно створити в проєкті теку ім'я якої відповідає назві проєкту та розмістити там всі модулі
- проєкт повинен містити теку `doc`, оскільки кожен проєкт містить документацію
- вартує створювати теки для модулів, якщо в проєкті їх значна кількість
- вартує аналізувати та запам'ятовувати структуру проєктів в репозитаріях GitHub, та використовувати цю інформації в подальшому.

Розглянемо рекомендації по вибору структури проєктів для різного типу застосунків.

2. Структура проєктів програм командного рядка (Command-Line Application Layouts)

Багато Python застосунків запускається на виконання з командного рядка. Такі програми можуть відрізнятися за призначенням та за складністю.

Програма одноразового використання (One-Off Script).

Програма одноразового використання це окремий модуль, який не потрібно встановлювати, а достатньо просто запустити на виконання. Такий застосунок може не мати ніяких залежностей і проєкт можна створювати без використання віртуального

середовища. Якщо ж передбачити наявність залежностей та потребу в розповсюдженні застосунку то рекомендується наступна структура для таких проєктів.

```
helloworld/
├── .gitignore
├── helloworld.py
├── LICENSE
├── README.md
├── requirements.txt
├── setup.py
└── test.py
```

Всі файли проєкту знаходяться в одній теці. Назва теки це ім'я проєкту. В теці знаходяться наступні файли:

- `.gitignore`: файл в якому вказано, які файли повинна ігнорувати система керування версіями Git.
- `helloworld.py`: модуль з сирцевим кодом, який може виконуватися та поширюватися. Ім'я модуля співпадає з назвою проєкту.
- `LICENSE`: текстовий файл в якому описано ліцензію, яка використовується в проєкті. Якщо код буде розповсюджуватися то ліцензію варто передбачити. Для вибору ліцензії можна скористатися [ChooseALicense](#).
- `README.md`: в цьому [Markdown](#) (або [reStructuredText](#)) файлі описано для чого та як можна використовувати застосунок.
- `requirements.txt`: в цьому файлі вказано залежності проєкту.
- `setup.py`: файл який дозволяє побудувати дистрибутив та провести встановлення застосунку.
- `tests.py`: набір тестів для перевірки працездатності застосунку.

Пакунок модулів, який передбачає інсталяцію (Installable Single Package)

Дуже мало застосунків складаються з одного модуля. В більшості проєктів кількість модулів швидко зростає і їх збереження на верхньому рівні втрачає зміст. В такому випадку, якщо всі ці модулі становлять єдиний пакунок рекомендується створити окремі теки для модулів та тестів, а розміщення інших файлів проєкту не змінювати.

```
helloworld/
├── helloworld/
│   ├── __init__.py
│   ├── helloworld.py
│   └── helpers.py
├── tests/
│   ├── test_helloworld.py
│   └── test_helpers.py
├── .gitignore
├── LICENSE
├── README.md
├── requirements.txt
└── setup.py
```

Тека в якій містяться модулі проєкту також містить файл `__init__.py`. Цей файл вказує Python ця тека це тека пакунку. Також за допомогою цього файлу можна керувати процесом імпортування з модулів пакунку. Файл `__init__.py` не є обов'язковим. Якщо

його опустити то можливість імпортувати теку як пакунок залишиться (Implicit Namespace Packages) але такий пакунок називають не `_regular packages_` а `_namespace packages_`.

Застосунок з внутрішніми пакунками (Application with Internal Packages).

Більші застосунки складаються не з одного а декількох пакунків, які можуть внутрішніми (владеними). Наступний приклад демонструє структуру проєкту такого застосунку.

```
helloworld/
├── bin/
├── docs/
│   ├── hello.md
│   └── world.md
├── helloworld/
│   ├── __init__.py
│   ├── runner.py
│   └── hello/
│       ├── __init__.py
│       ├── hello.py
│       └── helpers.py
│   └── world/
│       ├── __init__.py
│       ├── helpers.py
│       └── world.py
├── data/
│   ├── input.csv
│   └── output.xlsx
├── tests/
│   ├── hello
│   │   ├── test_helpers.py
│   │   └── test_hello.py
│   └── world/
│       ├── test_helpers.py
│       └── test_world.py
├── .gitignore
├── LICENSE
├── README.md
├── requirements.txt
└── setup.py
```

Файли верхнього рівня залишаються без змін а у структурі проєкту потрібно виділити наступні теки:

- `bin/`: в цій теці зберігаються всі виконувані файли. Виконуваний файл повинен містити мінімум коду (необхідні імпорти та виклик головної функції). Якщо в проєкті відсутні виконувані файли то тека може залишатися пустою. Виконувані Python файли зберігаються в цій теці без розширення.
- `docs/`: в цій теці зберігається документація до всіх частин проєкту. У наведеному прикладі в теці знаходиться документація до двох пакунків.
- `helloworld/`: тека модулів та пакунків модулів проєкту. Потрібно звернути увагу на відповідність імен модулів та назв тек.

- `data/`: в цій теці зберігаються всі файли, які містять вхідні дані для роботи програми та файли які створюються при роботі програми. Також тут можуть зберігатися файли з даними для тестування програми.
- `tests/`: у цій папці зберігаються всі тести проекту (unit tests, execution tests, integration tests).

Структура проекту веб застосунку (Web Application Layouts)

Інший великий клас Python застосунків це web застосунки. Django та Flask – це найпопулярніші веб-фреймворки для Python. У документації цих фреймворків наведені рекомендації по структурі проектів.

Flask

Flask переважно називають веб міні фремворком, бо з його допомогою можна дуже легко і з мінімальними налаштуваннями створити реальний веб застосунок. В документації Flask описано навчальний проект Flaskr (веб застосунок – блог), який має наступну структуру:

```

flaskr/
├── flaskr/
│   ├── __init__.py
│   ├── db.py
│   ├── schema.sql
│   ├── auth.py
│   ├── blog.py
│   └── templates/
│       ├── base.html
│       ├── auth/
│       │   ├── login.html
│       │   └── register.html
│       └── blog/
│           ├── create.html
│           ├── index.html
│           └── update.html
├── static/
│   └── style.css
├── tests/
│   ├── conftest.py
│   ├── data.sql
│   ├── test_factory.py
│   ├── test_db.py
│   ├── test_auth.py
│   └── test_blog.py
├── venv/
├── .gitignore
├── setup.py
└── MANIFEST.in

```

Структура передбачає збереження всіх модулів крім тестів у теці `flaskr`. Окремо знаходиться тека з віртуальним середовищем `venv`. Розміщення файлів верхнього рівня не змінюється.

Приклади структур Python проєктів

[sampleproject](#) – приклад структури проєкту в якій представлені останні офіційні рекомендації

[Django](#) – складний Python проєкт

[requests](#) – не складний Python проєкт

Література

<https://wiki.python.org/moin/ProjectFileAndDirectoryLayout>

<https://realpython.com/python-application-layouts>

<https://docs.python.org/3.8/tutorial/modules.html#packages>

<https://packaging.python.org/tutorials/packaging-projects>

<https://www.brainsorting.dev/posts/structuring-a-python-application>

<https://github.com/pypa/sampleproject>

<https://github.com/s7ck/python-project-templates>

<https://github.com/carlosperate/awesome-pyproject>

Лабораторна робота №1

Основи web – програмування

Короткі теоретичні відомості

1. Поняття маски та поділ IP-адреси на адресу мережі та адресу вузла.

Для того щоби комунікувати у мережі інтернет потрібно кожному пристрою мати певну "адресу". В комп'ютерних мережах, в моделі OSI на [3 рівні](#), що відповідає за трансляцію адрес, комутацію та маршрутизацію пакетів є стек протоколів [TCP/IP](#). Саме протокол IP (internet protocol) є тією "адресою", що дозволяє здійснювати передачу інтернет пакетів.

Для IP-адрес виділяється 4 байти, і адреса записується у вигляді чотирьох чисел, що представляють значення кожного байта в десятковій формі, які розділених крапками. Комп'ютер працює з двійковими представленнями адрес. Наприклад, 215.17.125.177 - десяткова форма представлення адреси, а 11010111.00010001.01111101.10110001 - двійкова форма представлення цієї адреси (32 біти). В комп'ютерних мережах навик роботи з бінарними числами є дуже помічним.

Наразі використовується IP версії 4, IPv4. Проте оскільки в світі фізичних пристроїв зараз є набагато більше ніж ця версія може алокувати (2^{32}), тому щоби побороти нестачу адрес, унікальна ip адреса алокується наприклад до провайдера, а всередині ISP (internet service provider) має свою підмережу, далі кожен в себе вдома має ще свою домашню підмережу, де домашній роутер має зазвичай 192.168.1.1 адресу (клас IP адрес C), і кожен з пристроїв домашньої мережі отримує свою IP адресу в цій підмережі. Всі ці пристрої в світ, в глобальний інтернет виходять e.g. за однією IP адресою, вашого провайдера.

Також вже активно вводиться новий стандарт, нова версія IPv6. Яка для представлення адреси має 16 байт (128 біт). Для розуміння кількості це приблизно 5×10^{28} адрес на кожную людину.

Отже, IP-адреса — унікальна мережева адреса вузла в мережі, що побудована на основі стеку протоколів TCP/IP. Комп'ютерні мережі можуть мати різну [топологію](#), кожен елемент мережі називається вузлом, наприклад ваш комп'ютер, чи роутер (маршрутизатор), чи комутатор, кожен з них є вузлом мережі.

Вузлу в мережі може бути призначена будь яка IP-адреса зі списку доступних до алокації адрес, і якщо призначена адреса суперечить прийнятим домовленостям то це призведе до проблем у роботі мережі. У домені мереж організація IETF розробляє протоколи і архітектуру інтернету. Такі зміни та правила вносяться у [RFC](#) документах, (схоже до PER) у Python. Таким чином згідно RFC3330 IPv4 адреси мають наступні призначення [Special-Use IPv4 Addresses](#)

Address Block	Present Use	Reference
0.0.0.0/8	"This" Network	[RFC1700, page 4]
10.0.0.0/8	Private-Use Networks	[RFC1918]
14.0.0.0/8	Public-Data Networks	[RFC1700, page 181]
24.0.0.0/8	Cable Television Networks	--
39.0.0.0/8	Reserved but subject to allocation	[RFC1797]
127.0.0.0/8	Loopback	[RFC1700, page 5]
128.0.0.0/16	Reserved but subject to allocation	--
169.254.0.0/16	Link Local	--
172.16.0.0/12	Private-Use Networks	[RFC1918]
191.255.0.0/16	Reserved but subject to allocation	--
192.0.0.0/24	Reserved but subject to allocation	--
192.0.2.0/24	Test-Net	
192.88.99.0/24	6to4 Relay Anycast	[RFC3068]
192.168.0.0/16	Private-Use Networks	[RFC1918]

198.18.0.0/15	Network Interconnect Device Benchmark Testing	[RFC2544]
223.255.255.0/24	Reserved but subject to allocation	--
224.0.0.0/4	Multicast	[RFC3171]
240.0.0.0/4	Reserved for Future Use	[RFC1700, page 4]

Деколи вказують що IP-адреса належить до певного класу. Можна знайти наступну інформацію про класи IP-адрес, хоча у справжніх мережах ця інформація використовується рідко. Кожен з цих класів відповідає за тип IP адреси, наприклад адреси особистого використання (вдома), для серверів, для ISP, для досліджень (наприклад в CERN) і тд.

Клас A: 1.0.0.0–126.0.0.0,	маска 255.0.0.0
Клас B: 128.0.0.0–191.255.0.0,	маска 255.255.0.0
Клас C: 192.0.0.0–223.255.255.0,	маска 255.255.255.0
Клас D: 224.0.0.0–239.255.255.255,	маска 255.255.255.255
Клас E: 240.0.0.0–247.255.255.255,	маска 255.255.255.255

Ми вже згадували що адрес IPv4 є замало, і тому вони перевикористовуються у підмережах.

У попередніх таблицях та у налаштуваннях мережевих інтерфейсів можна звернути увагу на ще одне 4-х байтне число – маску. В першій таблиці маска вказана як префікс (/24) — кількість значущих бітів у 4-х байтовому числі. В другій як 4-х байтне число де кожен байт представлений десятковим числом.

Насправді, IP - адреса складається із двох логічних частин — номера мережі й номери вузла в мережі. Яка частина адреси належить до номера мережі, а яка — до номера вузла, визначається маскою. Маска — це число (бітова маска), що використовується в парі з IP-адресою; двійковий запис маски містить одиниці в тих розрядах, які повинні в IP-адресі інтерпретуватися як номер мережі. Оскільки номер мережі це нерозривна частина адреси, одиниці в масці повинні становити безперервну послідовність. Наприклад, часто використовуються маски з наступними значеннями:

255. 0.0.0	- 11111111. 00000000. 00000000. 00000000;
255.255.0.0	- 11111111. 11111111. 00000000. 00000000;
255.255.255.0	- 11111111. 11111111. 11111111. 00000000.

Мережа це та частина IP адреси, яка не змінюється у всій мережі й всі адреси пристроїв починаються з цього номера мережі. Вузол – це змінна частина IP адреси. Кожен пристрій має свою унікальну адресу в мережі й це адреса вузла (ідентифікатор вузла).

Для визначення **номера мережі та номера вузла** потрібно виконати додаткові обчислення. Для обчислення номера мережі за заданою IP-адресою та маскою необхідно виконати побітове “І” (AND) між адресою та маскою. Таку операцію називають накладанням маски на адресу.

Наприклад, якщо IP адреса 215.17.125.177, а маска 255.255.255.240 (/28), то в двійковій формі це:

IP-адреса:	215.017.125.177	(11010111.00010001.01111101.10110001)
Маска:	255.255.255.240 == 28	(11111111.11111111.11111111.11110000)
Адреса мережі:	215.17.125.176/28	(11010111.00010001.01111101.10110000)

Хостові (вузлові) біти в масці це нулі справа.

Перша адреса вузла доступна до алокації після адреси мережі, визначається усіма 0 в кінці маски + 1 перший біт (активований - 1).

215.17.125.177 (11010111.00010001.01111101.10110001)

Остання адреса вузла доступна до алокації визначається усіма 1 в масці - 1 (перший біт нульовий)

215.17.125.190 (11010111.00010001.01111101.1011 1110)

Остання адреса в мережі це адреса broadcast (широкомовна адреса) - визначається усіма 1 в масці.

215.17.125.191 11010111.00010001.01111101.10111111

Для того щоби знайти широкомовну адресу потрібно інвертувати маску мережі та виконати операцію OR

```
215.017.125.177      11010111.00010001.01111101.10110001 (ір адреса)
                     00000000 00000000 00000000 00001111 (інвертована маска)
                     ----- [OR]
                     11010111 00010001 01111101 10111111 ----> 215.17.125.191
```

Якщо IP адреса 67.38.173.245 а маска 255.255.240.0, то в двійковій формі це;
 IP-адреса: 67.038.173.245 (01000011.00100110.10101101.11110101)
 Маска: 255.255.240.0 (11111111.11111111.11110000.00000000)
 Адреса мережі: 67.38.160.0 (01000011.00100110.10100000.00000000)

Кількість вузлів в мережі, адрес доступних до алокації, а точніше в підмережі визначається як $2^{32-N}-2$, де N — довжина маски (кількість значущих бітів). Чим довша маска, тим менше в підмережі вузлів.

Маска підмережі		Розмір ідентифікатора вузла	Максимальна кількість вузлів	
8 біт	255.0.0.0	24 біт	$2^{24} - 2$	16777214
16 біт	255.255.0.0	16 біт	$2^{16} - 2$	65534
24 біт	255.255.255.0	8 біт	$2^8 - 2$	254
29 біт	255.255.255.248	3 біт	$2^3 - 2$	6

При обчисленні максимальної кількості вузлів обчислюється скільки доступних IP-адрес є в мережі. Це число зменшується на 2, бо адреса мережі не присвоюється вузлам. Так само широкомовна адреса (broadcast) також не використовується як адреса вузла. Широкомовна адреса (broadcast) мережі сприймається кожним вузлом, як додаткова власна адреса й пакет на цю адресу отримають усі вузли мережі, так наче вони були адресовані їм особисто. Широкомовна адреса (broadcast) мережі обчислюється згідно наступного виразу:

широкомовна_адреса_мережі = IP_будь-якого_комп'ютера_цієї_мережі OR NOT (MASK) (https://en.wikipedia.org/wiki/Broadcast_address)

Для того щоб префіксовану форму маски перевести у маску підмережі можна скористатися наступним способом (вартує його уважно дослідити):

```
1  """
2  x << y
3      Returns x with the bits shifted to the left by y places
4      (and new bits on the right-hand-side are zeros).
5      This is the same as multiplying x by 2**y.
6  """
7
8  prefix = 32
9  mask = [0, 0, 0, 0]
10
11  for i in range(prefix):
12      mask[i // 8] += 1 << (7 - i % 8)
13
14  print(mask)
```

Наступний приклад містить обчислення, які потрібно буде реалізувати під час виконання лабораторної роботи:

```
IP у десятковому вигляді: 172.20.0.0/14
IP у двійковому вигляді: 10101100.00010100.00000000.00000000
маска підмережі: 11111111.11111100.00000000.00000000 (255.252.0.0)
```

```
----- [Logical AND (&)]
адреса мережі: 10101100.00010100.00000000.00000000 ---> 172.20.0.0

маска вузла: 00000000.00000011.11111111.11111111 (0.3.255.255)
(інвертована маска мережі)
IP у двійковому вигляді: 10101100.00010100.00000000.00000000
(Усі біти вузла у -----
IP адресі робимо одиницями)
[OR / Force host bits]
широкомовна адреса: 10101100.00010111.11111111.11111111 -> 172.23.255.255
```

```
адреса мережі: 172.20.0.0
широкомовна адреса: 172.23.255.255
кількість біт, які відповідають за адресу вузла: 32-14=18 (14 це кількість
одиничних бітів у масці мережі), 18 - кількість бітів у вузловій частині маски)
загальна кількість вузлів у мережі: 2**18-2 = 262144-2=262142 (віднімаємо 2
адреси оскільки одна адреса це адреса мережі, а друга це широкомовна адреса)
```

За наступним посиланням можна знайти прості формули для вищеперелічених обчислень, але потрібно контролювати чи ці спрощення не призводять до помилок (<https://www.geeksforgeeks.org/ipv4-classless-subnet-equation/>).

2. HTML.

HTML (*HyperText Markup Language* — **Мова розмітки гіпертекстових документів**) — стандартна мова розмітки веб-сторінок в Інтернеті. Більшість веб-сторінок створюються за допомогою мови HTML (або XHTML). Документ HTML оброблюється браузером та відтворюється на екрані у звичному для людини вигляді. HTML разом із каскадними таблицями стилів (CSS) та вбудованими скриптами типу JavaScript (не плутати з Java) — це три основні технології побудови веб-сторінок. HTML надає засоби для:

- створення структурованого документа шляхом визначення структурного складу тексту: заголовки, абзаци, списки, таблиці, цитати та інше;
- отримання інформації із Всесвітньої мережі через гіперпосилання;
- створення інтерактивних форм;
- додавання зображень, звуку, відео, та інших об'єктів до тексту.
- [та багато іншого](#)

3. CSS.

Каскадні таблиці стилів (*Cascading Style Sheets* або скорочено **CSS**) — спеціальна мова, що використовується для опису зовнішнього вигляду сторінок (стилю), написаних

мовами розмітки даних. Найчастіше CSS використовують для візуальної презентації сторінок, створених за допомогою HTML та XHTML, але формат **CSS** може застосовуватися до інших видів XML-документів.

Більше інформації та пояснення до прикладів простих веб-сторінок можна знайти у лекції з курсу CS50 (<https://prometheus.org.ua/cs50/week7w.html>). Конспект лекції у доданому файлі Лекція 7-2.html.

4. Побудова web – карти.

Для побудови карти у веб-браузері потрібно створити html файл, який буде містити карту. Побудову карти можна здійснити за допомогою бібліотек pandas та folium.

pandas — програмна бібліотека, написана на мові програмування Python для маніпулювання даними та їхнього аналізу.

folium – бібліотека, яка дозволяє будувати інтерактивні карти з використанням Python та бібліотеки Leaflet.js.

Для роботи з даними використовується Python, а їх візуалізація здійснюється в Leaflet.js за допомогою folium.

Бібліотеки pandas та folium не входять у стандартну бібліотеку Python, і їх потрібно додатково встановити за допомогою програми pip.

```
pip install pandas
pip install folium
```

Розглянемо приклад такої карти (файл [map_volcanoes.html](#)). На карті зображено карту землі, межі країн, населення країн та вулкани США. Карта інтерактивна і можна змінювати її масштаб та вимикати/вмикати шари цієї карти (власне карта, населення, вулкани).

Для побудови web - карти за допомогою folium потрібно виконати два основні кроки:

1. Створити об'єкт – карту.
2. Перетворити цей об'єкт в html файл.

Для створення об'єкту - карти потрібно скористатися класом Map бібліотеки folium.

```
import folium
map = folium.Map()
```

Для перетворення об'єкту в html файл потрібно скористатися методом save() цього класу.

```
map.save('Map_1.html')
```

В результаті отримано html файл. Якщо відкрити цей файл у браузері то можна побачити що це карта, яку можна масштабувати. Іншими словами це карта, яка містить єдиний шар і цей шар це карта з проекту OpenStreetMap. Якщо потрібно щоб цим базовим шаром була інша карта то потрібно при створенні карти вказати відповідне значення для параметра tiles. Наприклад,

```
map = folium.Map(tiles="Stamen Terrain")
```

Для зручності карту можна сформулювати так, що при її відкритті на екрані в центрі карти буде певне місце і карта буде зображатися у певному початковому масштабі. Ці та інші параметри можна вказати, як параметри при створенні карти. Повний перелік параметрів доступний в документації та файлі допомоги.

```
map = folium.Map(tiles="Stamen Terrain", location=[49.817545, 24.023932], zoom_start=17)
```

Карта, яку буде створено з цими значеннями параметрів `location` та `zoom_start` буде показувати місце де знаходиться академічний корпус на вул. Козельницькій. Додавання розширеної інформації для її зображення на карті можна здійснювати різними способами. Додавати інформацію безпосередньо до цієї базової карти можна, наприклад за допомогою методу `add_child`.

```
map.add_child(folium.Marker(location=[49.817545, 24.023932],
                             popup="Хіба я тут!",
                             icon=folium.Icon()))
```

На карті з'явився маркер з написом, який з'являється при наведенні курсора на об'єкт і може зникати. Такий самий результат можна отримати якщо скористатися класом `FeatureGroup`, і саме цьому способу потрібно надавати перевагу, якщо потрібно щоб інформація на карті відображалася на декількох шарах (складалася з декількох шарів). Якщо повторити у програмі цей рядок декілька разів то на карті будуть зображуватися декілька маркерів. Якщо створити список координат точок, які потрібно позначити маркером то простий цикл дозволить уникнути дублювання однакових рядків у програмі. Якщо потрібно позначити на карті багато точок то доцільно зберегти координати цих точок у файлі й використовувати при потребі.

Наступний приклад містить фрагмент програми для обробки та зображення на карті інформації про декілька населених пунктів Івано-франківської області. У файлі [Stan_1900.csv](#) знаходиться інформація про три населені пункти де вказано рік, назва населеного пункту, назви церков, склад населення, інформація про школу та вказані координати населеного пункту (інформація взята з шематизму Станіславської єпархії за 1900 р.). Бібліотека `pandas` надає засоби для доступу до цих даних шляхом створення об'єкту типу `DataFrame`, який буде містити всю інформацію з csv файлу, а до фрагментів цієї інформації можна отримати доступ за назвою стовпчика. Працювати з csv файлами можна також за допомогою засобів модуля `csv` стандартної бібліотеки, але бібліотека `pandas` зараз набула більшого поширення.

```
import folium
import pandas
data = pandas.read_csv("Stan_1900.csv", error_bad_lines=False)
lat = data['lat']
lon = data['lon']
map = folium.Map(location=[48.314775, 25.082925],
                  zoom_start=10)
fg = folium.FeatureGroup(name="Kosiv map")
for lt, ln in zip(lat, lon):
    fg.add_child(folium.Marker(location=[lt, ln],
                              popup="1900 рік"
                              icon=folium.Icon()))

map.add_child(fg)
map.save('Map_5.html')
```

За потреби у вікні `popup` можуть бути показані будь які дані csv файлу. Наприклад,

```
churches = data['церкви']
for lt, ln, ch in zip(lat, lon, churches):
    fg.add_child(folium.Marker(location=[lt, ln],
                              popup="1900 рік" + ch,
                              icon=folium.Icon()))
```

в цьому випадку на карті буде інформація про церкви в цих населених пунктах.

У бібліотеці `folium` реалізовані й інші засоби за допомогою яких на карті можна використовувати позначення різного типу, форми та кольору. Наступний фрагмент демонструє як зобразити на карті позначення населених пунктів не за допомогою стандартних маркерів, а кружків різного кольору, який визначається в залежності від

кількості населення. Для цього використовується інший клас CircleMarker та відповідні значення для його параметрів.

```
import folium
import pandas
data = pandas.read_csv("Stan_1900.csv")
lat = data['lat']
lon = data['lon']
churches = data['църкви']
hc = data['гр-кат.']

def color_creator(population):
    if population < 2000:
        return "green"
    elif 2000 <= population <= 3500:
        return "yellow"
    else:
        return "red"

map = folium.Map(location=[48.314775, 25.082925],
                  zoom_start=10)
fg = folium.FeatureGroup(name="Kosiv_map")
for lt, ln, ch, hc in zip(lat, lon, churches, hc):
    fg.add_child(folium.CircleMarker(location=[lt, ln],
                                     radius=10,
                                     popup="1900 pik"+"\n" + ch,
                                     fill_color=color_creator(hc),
                                     color='red',
                                     fill_opacity=0.5))

map.add_child(fg)
map.save('Map_6.html')
```



```
map.add_child(fg_pp)
map.add_child(folium.LayerControl())
map.save('Map_7.html')
```

В попередніх прикладах для розміщення на карті маркерів використовуються координати (широта, довгота) населених пунктів, які вказані у файлах. У випадку якщо б координати були відсутні було би потрібно їх визначити. Для визначення координат можна скористатися спеціалізованими сервісами наприклад google map, або однією з багатьох Python бібліотек, які розроблені з цією метою.

Бібліотека геору дозволяє встановити координати населених пунктів місцевостей та окремих адрес. Для використання цієї бібліотеки її потрібно додатково встановити і вказати у requirements.txt для документації проекту. Для отримання координат та іншої додаткової інформації потрібно виконати наступні дії.

```
from geopy.geocoders import Nominatim
geolocator = Nominatim(user_agent="specify_your_app_name_here")
location = geolocator.geocode("Старі Кути")
print(location.address)
print((location.latitude, location.longitude))
print(location.raw)
```

specify_your_app_name_here - не забудьте назвати вашу програму тут

В результаті буде отримано:

Старі Кути, Косівський район, Івано-Франківська область, Україна
(48.287312, 25.1738)

```
{'place_id': '242217085', 'licence': 'Data © OpenStreetMap contributors, ODbL 1.0.
https://osm.org/copyright', 'osm_type': 'relation', 'osm_id': '8839103', 'boundingbox': ['48.26637',
'48.2921912', '25.13844', '25.1852525'], 'lat': '48.287312', 'lon': '25.1738', 'display_name': 'С т а р і
К у т и , К о с і в с ь к и й р а й о н , І в а н о - Ф р а н к і в с ь к а о б л а с т ь ,
У к р а ї н а', 'class': 'boundary', 'type': 'administrative', 'importance': 0.465228110551882, 'icon':
'https://nominatim.openstreetmap.org/images/mapicons/poi_boundary_administrative.p.20.png'}
```

Якщо потрібно отримати координати великої кількості локацій то можливе виникнення помилки Too Many Requests 429 error. Для її уникнення потрібно використовувати клас RateLimiter, який дозволяє керувати процесом викликів з відповідними параметрами.

```
from geopy.geocoders import Nominatim
geolocator = Nominatim(user_agent="specify_your_app_name_here")
from geopy.extra.rate_limiter import RateLimiter
geocode = RateLimiter(geolocator.geocode, min_delay_seconds=1)
```

```
for point in ["Львів", "Старі Кути", "Кути", "Брустурів"]:
    location = geolocator.geocode(point)
    print(location.address)
    print((location.latitude, location.longitude))
```

Виклики geolocator.geocode будуть виконуватися із затримкою min_delay_seconds, а при виникненні винятків виклики будуть виконуватися до завершення вказаного проміжку часу (max_retries).

Література:

1. <https://www.geeksforgeeks.org/ipv4-classless-subnet-equation/>
2. <https://www.calculator.net/ip-subnet-calculator.html/>
3. Курс CS50
4. Курс The Python Mega Course: Build 10 Real World Applications.

5. <https://www.pluralsight.com/blog/it-ops/simplify-routing-how-to-organize-your-network-into-smaller-subnets>
6. <http://jodies.de/ipcalc>

Лабораторна робота №3

Основи використання API.

Розроблення веб застосунків з використанням Flask та FastAPI.

Короткі теоретичні відомості.

1. Доступ до даних в мережі Інтернет.

В Python реалізовано багато засобів (бібліотек, пакунків модулів, окремих модулів) для доступу до даних в мережі Інтернет. Засоби, які входять у стандартну бібліотеку ([urllib](#)) та засоби сторонніх розробників ([requests](#)) дозволяють у найбільш загальному випадку отримати вміст html сторінки (див. приклад у презентації до лекції). Якщо не існує іншої можливості доступу до даних або свідомо обрано такий спосіб доступу то виникає потреба в розборі html файлу - видалення розмітки та добування вмісту (тексту, зображень тощо). В Python так само є достатньо засобів для здійснення такого розбору. Наприклад, бібліотеки та модулі lxml, BeautifulSoup, html дозволяють робити це ефективно.

Більш зручно отримати дані з мережі Інтернет можна з використанням API, які надають власники цих даних. В такому випадку, найчастіше, дані будуть надані у форматі JSON, який зручний як для читання так і для розбору за допомогою модуля стандартної бібліотеки json.

Для спрощення роботи з API розробляються допоміжні програми обгортки, які дозволяють скористатися відповідним API, але формування запитів та їх отримання дещо спрощено.

2. Обробка даних з мережі Інтернет.

Незалежно від того у який спосіб отримані дані чи за допомогою оригінального API чи за допомогою програми оболонки у більшості випадків дані будуть отримані в JSON форматі.

Формат JSON це вже стандарт для обміну інформацією, який використовується з метою передачі даних в мережі інтернет чи збереження даних локально. Для роботи з цим форматом стандартна бібліотека Python надає відповідний модуль json.

Основні дії які дозволяє виконувати цей модуль це:

- серіалізація (перетворення на послідовність байтів) - кодування призначене для перетворення даних на JSON (запис даних на диск)
- десеріалізація (декодування) - читання JSON (зчитування даних у пам'ять).

Для виконання цих дій використовують наступні методи

Метод	Призначення
<code>dumps()</code>	кодування у JSON об'єкт (рядок)
<code>dump()</code>	кодування в рядок, який буде збережено у файлі
<code>loads()</code>	декодування JSON рядка
<code>load()</code>	декодування при читанні JSON файла

При серіалізації Python об'єкти будуть перетворені у відповідні JSON об'єкти згідно наступної таблиці.

Серіалізація (dump, dumps) Десеріалізація (load, loads)

Python	JSON	JSON	Python
dict	object	object	dict
list, tuple	array	array	list
str	string	string	str
int, float	number	number (int) number (real)	int float
True	true	true	True
False	false	false	False
None	null	null	None

Наступний приклад демонструє простий варіант серіалізації Python об'єкту у файл:

```
import json
data = {
    "student": {
        "name": "Arman",
        "species": "Ca de Bou",
        "group": "noname"
    }
}

with open('student_list.json', 'w') as f:
    json.dump(data, f, indent=4)
```

та у рядок:

```
json_string = json.dumps(data, indent=4)
print(json_string)
```

При десеріалізації JSON об'єкти будуть перетворені у відповідні Python об'єкти згідно таблиці вище. При десеріалізації потрібно враховувати, що наведена вище таблиця не є оберненою таблицею перетворення об'єктів при серіалізації.

Наступний приклад демонструє цю особливість при роботі з кортежем.

```
student = ('Arman', 'Ca de Bou')
encoded_student = json.dumps(student)
decoded_student = json.loads(encoded_student)
print(student == decoded_student)
print(type(student))
print(type(decoded_student))
print(student == tuple(decoded_student))
```

Якщо потрібно отримані дані, які збережені у форматі JSON то потрібно скористатися одним з двох методів модуля json,)

```
json.load(fp, *, cls=None, object_hook=None, parse_float=None,
parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)
    Deserialize fp (a .read()-supporting file-like object containing a JSON
document) to a Python object using this conversion table.
```

```
json.loads(s, *, encoding=None, cls=None, object_hook=None, parse_float=None,
parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)
```

Deserialize `s` (a `str`, `bytes` or `bytearray` instance containing a JSON document) to a Python object using this conversion table

в залежності від того чи дані є у вигляді файлу або у вигляді рядка символів чи байтів.

У наступному прикладі показано використання функції `loads()` для роботи з рядком, який містить JSON. Потрібно звернути увагу на використання ключового параметра `object_hook` за допомогою якого можна задати спосіб обробки рядка. У прикладі значення цього рядка це функція, яка дозволяє на основі даних з рядка отримати комплексне число.

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...             object_hook=as_complex)
(1+2j)
```

Наступний приклад демонструє як використовувалася інша функція `load()` для того, щоб отримати дані з файлу `world.json` та визначити густоту населення в кожній з країн.

```
>>> import json
>>> import pprint
>>> f = open('a.json', encoding = 'utf-8')
>>> country_data = json.load(f)
>>> for country in country_data['features']:
...     pprint.pprint(country['properties']['NAME'])
...     pprint.pprint(str(country['properties']['POP2005']/
...                          country['properties']['AREA']))
```

З цього прикладу стає зрозуміло, що працювати з елементами `json` файлу можна як із елементами Python словника. Також потрібно звернути увагу на використання модуля `pprint` стандартної бібліотеки. Функція `pprint` модуля `pprint` дозволяє виводити структури даних Python у форматovanому вигляді. Інші приклади роботи з `json` можна знайти в архіві `json_read.zip` в теці Додаткові матеріали.

Також для роботи з JSON можна використовувати сторонні пакунки, наприклад [jmespath](#) (його використання не є обов'язковим для виконання завдань), який дозволяє здійснювати пошук значення в об'єкті за шаблоном.

Наприклад, для файлу `kved.json` (файл з першого завдання) наступний шаблон покаже список усіх розділів другої секції:

```
jmespath.search("sections[0][1].divisions[].divisionCode", kved_json)
```

3. Використання API.

API, який надає власник web-ресурсу це набір правил, яких повинен дотримуватися клієнт (інший веб застосунок) для того, щоб отримати доступ до тих чи інших даних. Для успішного використання API необхідно обов'язково ознайомитися з відповідною документацією, яку надає розробник API. Загальна схема роботи з API наступна:

- отримання базового URL (endpoint);
- формування запиту згідно API;
- надсилання запиту;
- обробка отриманих результатів запиту.

Допоміжні програми обгортки дозволяють більш спрощено формувати та надсилати запити, але не завжди забезпечують всі можливості оригінального API.

В цій загальній послідовності кроків, приклад виконання яких наведено у вказівках до виконання першого етапу міні проєкту потрібно звернути увагу на те, що багато веб-ресурсів вимагають від клієнтів для використання їх API надання API-ключа. API-ключ дозволяє контролювати використання API. За API-ключем можна відстежувати діяльність клієнта, накладати певні обмеження, встановлювати оплату.

Якщо для використання API потрібен API-ключ то його можна отримати згідно з визначеною власником API процедури. Надалі такий ключ використовується як частина запиту, або використовується як частина більш складної процедури.

Наприклад, для того щоб отримати доступ до даних соціальної мережі Twitter потрібно використовувати Twitter API. Для використання Twitter API потрібно отримати API-ключ після проходження процедури реєстрації web-додатка, який передбачає використання Twitter API. Після цього кожен запит буде підписуватись за допомогою технології OAuth (www.oauth.net).

4. Приклад використання Twitter API.

Для використання Twitter API потрібно мати чинний обліковий запис цієї соціальної мережі та API-ключ, який генерується для кожного із застосунків, що використовують Twitter API. Для генерації API-ключа потрібно на сторінці за адресою <https://apps.twitter.com/> створити Twitter app. Застосунок повинен мати оригінальне ім'я та опис із вказаною адресою цього додатку в мережі Інтернет. В описі застосунку потрібно досить детально описати як будуть використовуватися та представлятися отримані дані. До опису додатку потрібно підходити уважно й відповідально бо опис буде розглядатися власниками ресурсу перед наданням ключів. **Процес отримання ключів може тривати певний час, а при неналежному оформленні запиту ключі не будуть надані.**

Application under review.

Hello,

We've received your request for API access and are in the process of reviewing it.

Be sure to watch this email address over the coming days, as we may request more information to facilitate the review process (be sure to check your spam folder as well).

We know that this application process delays getting started with Twitter's APIs. This information helps us protect our platform and serve the health of the public conversation on Twitter. It also informs product investments and helps us better support our developer community. For more information about our policies please see our [Terms of Service](#) and our [Developer Terms](#).

You'll receive an email when the review is complete.

In the meantime, check out our [documentation](#), explore our [tutorials](#), or check out our [community forums](#).

Після завершення створення додатку буде згенеровано: Consumer Key (API Key), Consumer Secret (API Secret), Access Token та Access Token Secret, які будуть використовуватися для роботи з API.

За адресою www.py4e.com/code3 можна завантажити файли `twurl.py`, `hidden.py`, `oauth.py`, `twitter1.py` та `twitter2.py` на основі яких можна створити проєкт для ознайомлення з Twitter API та розроблення власного.

Основні модулі проєкту це `twitter1.py` та `twitter2.py`, а `twurl.py`, `hidden.py`, `oauth.py` - допоміжні. В модулі `hidden.py` зберігаються дані API-ключа, що дозволяє при формуванні запиту не захаращувати тіло функції `augment` з модуля `twurl.py`, який створений для формування GET запиту. Функція `augment` приймає базову адресу API та параметри запиту і повертає повний запит. Виклик цієї функції здійснюється в основному модулі `twitter1.py` або `twitter2.py`. В результаті виконання `twitter1.py` користувач отримує часову стрічку (timeline) вказаного облікового запису Twitter як рядок у форматі json. На екран виводяться перші 250 символів з timeline та значення параметра `'x-rate-limit-remaining'` із заголовка відповіді на HTTP запит, який інформує про кількість запитів, які можна ще зробити. В результаті виконання `twitter2.py` користувач отримує інформацію про товаришів облікового запису Twitter. Інформація надходить в форматі json і з неї видобуваються певні деталі про товаришів. Детальна інформація про товаришів виводиться на екран. На екран також виводиться їх останні статуси.

Більш детальну інформацію про використання Twitter API можна знайти на сторінці <https://developer.twitter.com/en.html>

5. Розроблення веб застосунків з використанням Flask

Для отримання початкових навичок по створенню веб застосунків доцільно опрацювати всі приклади з матеріалів лекції та проєкт в теці Додаткові матеріали.

6. Розроблення веб застосунку з API за допомогою FastAPI.

Для того щоб створити свій веб-застосунок на базі FastAPI, необхідно встановити такі залежності:

```
fastapi
```

```
uvicorn
```

Для цього слід виконати команду:

```
$pip install fastapi uvicorn
```

Щоб створити простий вебсервер, достатньо створити модуль `main.py` з наступним вмістом:

```
from fastapi import FastAPI          # імпорт необхідного модуля
app = FastAPI(title="Winged phrase")  # оголошення об'єкта сервера

@app.get("/")                         # визначення базового URL -ендпоінта
async def root():                    # оголошення функції, яка викликатиметься
    return {"message": "Hello World"} # викликатиметься, при запиті до ендпоінта
```

Для запуску сервера, в терміналі необхідно виконати:

```
$uvicorn main:app
```

Якщо модуль має іншу назву (не main.py), то в команді треба вказати цю назву (uvicorn module_name:app), де module_name - правильна назва модуля

Оскільки GET запити виконуються в браузері, для того, щоб подивитись на результат виконання надсилання запиту до сервера - необхідно перейти на <http://127.0.0.1:8000/>



```
{"message": "Hello World"}
```

Розширимо функціонал сервера, щоб можна було приймати інші типи запитів, та виконувати певні дії.

У модулі db.py визначимо функції для збереження, отримання та видалення крилатих фраз.

```
import random          # імпорт модуля

items = {}              # оголошення словника, в якому будуть зберігатись фрази

def get_random_item():
    # функція повертає випадкове значення з словника
    return random.choice(list(items.values()))

def add_item(phrase: dict):
    # функція для додавання фрази в словник
    id = len(items) + 1      # унікальне значення для нової фрази
    phrase.update({"id": id}) # додаємо значення id до фрази
    items[id] = phrase       # зберігаємо оновлене значення в словник фраз
    return phrase            # функція повертає словник

def delete_item(id):
    if id in items: # перевірка на наявність ключа в словника
        del items[id] # видалення значення з словника
    else:
        raise ValueError("Phrase doesn't exist") # виклик помилки, якщо ключа
        в словнику немає. Ця помилка оброблятиметься в місці виклику функції.
```

У модулі main.py додамо імпорти необхідних функцій та ендпоінти для сервера:

```
from db import get_random_item, add_item, delete_item # імпорт функцій

@app.get(
    "/get",
    response_description="Winged phrase",
    description="Get random phrase from database"
)
# до сервера можна надіслати GET запит до /get ендпоінта
def get():
    # оголошення функції, яка викликатиметься, при запиті до
    # ендпоінта
    try:
        phrase = get_random_item() # виклик функції для отримання фрази
    except IndexError:
```

```

        raise HTTPException(404, "Phrase list is empty") # todo check,
                                                         # possibly not correct
    return phrase                                     # функція повертає словник з фразою та її id

@app.post(
    "/add", 7
    response_description="Added phrase with *id* parameter"
)
# до сервера ми можна надіслати POST запит до /add ендпоінта
def add(phrase: dict):
    # оголошення функції, яка викликатиметься,
    # при запиті до ендпоінта
    phrase_out = add_item(phrase) # виклик функції для додавання фрази в
    # словник
    return phrase_out             # функція повертає фразу, що була додана

@app.delete("/delete", response_description="Result of deletion") # до
# сервера можна надіслати DELETE запит до /delete ендпоінта
def delete(id: int):
    # оголошення функції, яка викликатиметься,
    # при запиті до ендпоінта
    try:
        delete_item(id) # виклик функції для видалення фрази
        return {"message": "OK"} # функція повертає словник
    except ValueError as e:
        # обробка ValueError, згенерованої під час
        # виклику функції delete_item
        raise HTTPException(404, str(e)) # функція повертає словник

```

Для того, щоб надсилати запити, можна скористатись функціоналом модуля requests

Приклади запитів:

<http://127.0.0.1:8000/get> - запит для отримання випадкової фрази

```

import requests # імпорт модуля

url = "http://localhost:8000/get" # оголошення посилання для запиту
response = requests.request("GET", url) # виконання запиту

print(response.text) # виведення результату

```

<http://127.0.0.1:8000/add> - запит для додавання фрази в словник

Тіло запиту має містити JSON з бажаним текстом, наприклад {"text": "Random phrase 1"}

```

import requests # імпорт модуля

url = "http://localhost:8000/add" # оголошення посилання для запиту

payload="{ 'text': 'Random phrase 1' }" # оскільки ми передаємо словник, ми
# передаємо його як атрибут data
headers = { 'Content-Type': 'application/json' } # заголовок, який вказує що
# дані, які передаються в форматі JSON

response = requests.request("POST", url, headers=headers, data=payload)
print(response.text) # виведення результату

```

<http://127.0.0.1:8000/delete?id=1>

```
import requests # імпорт модуля
```

```
url = "http://localhost:8000/delete?id=1" # оголошення посилання для запиту.  
# Передається просто число, й потреби використовувати атрибут payload запиту  
# немає. Значення передається як параметр запиту (query parameter)
```

```
response = requests.request("DELETE", url) # виконання запиту  
print(response.text)                       # виведення результату
```

7. Розгортання веб-сервера на хмарному сервісі <https://www.pythonanywhere.com/>

Для того, щоб забезпечити доступ до веб-сервера його потрібно розгорнути (розмістити) в мережі Інтернет. Хмарний сервіс <https://www.pythonanywhere.com/> дозволяє розмістити безкоштовно один веб застосунок. Для використання цього сервісу потрібно спочатку створити обліковий запис. Після цього потрібно виконати 7 кроків згідно з інструкцією по розміщенню веб-сервера на цьому ресурсі.

Для розгортання веб застосунку, який розроблено з використанням FastAPI рекомендують використовувати сервіс [Deta](#). Документація FastAPI містить докладну [інструкцію](#) як це зробити.

Література

<https://docs.python.org/3/library/json.html>

<https://docs.python.org/3/library/urllib.html>

<https://requests.readthedocs.io/en/>

www.py4e.com

<https://www.queworx.com/blog/why-you-should-try-fastapi/>

Лабораторна робота №4

Проектування та розроблення простих класів в Python Короткі теоретичні відомості.

1. Поняття об'єкту.

Об'єкт – модель, яка може щось зробити і з якою можна зробити щось.

Об'єкт це сукупність (поєднання) даних і пов'язаних з ними поведінок.

Об'єктно - орієнтованість це спрямованість на моделювання об'єктів.

Наприклад, Вікно - об'єкт; Викладач - об'єкт; Запах - ?; Файл - об'єкт.

2. Порядок використання парадигми ООП при вирішенні задач.

ООП - загальний підхід до вирішення проблем моделювання складних систем через опис сукупності об'єктів що взаємодіють через їх дані і поведінки.

Якщо говоримо ООП то це вказує на обраний стиль розробки ПЗ.

- Розібратись в предметній галузі (області) – системний аналіз (**Object-oriented analysis** ОО аналіз).
 - ідентифікувати об'єкти та дізнатися як вони взаємодіють. "Що треба зробити". Результат - вимоги.
- Визначитися з типами об'єктів, які потрібні для вирішення задачі та описати властивості (ознаки), які необхідно мати визначеним для вирішення задачі типам – проектування (**Object-oriented design** ОО проектування)
 - назвати об'єкти, визначити поведінки та встановити, яка поведінка об'єкту, яку поведінку активує іншого об'єкту. Визначити множину класів і інтерфейсів. "Як це зробити". Результат - специфікація реалізації.
- Написати класи для представлення цих типів – реалізація (**Object-oriented programming**)
 - написати код на ООП мові програмування.
- Провести тестування того що написали – тестування (**Object-oriented programming**)
 - написати код для тестування розроблених класів.

3. Поняття класу.

Термін для ідентифікації видів об'єктів – клас. Клас описує об'єкт. Клас це креслення за яким буде цей об'єкт виготовлено.

Наприклад, в аудиторії два вікна але кожне з них має спільні для класу вікон ознаки (атрибути) і відповідні поведінки.

4. Основні поняття ООП термінології.

Клас (Class): Тип об'єкту, який визначений програмістом в якому описана множина атрибутів що характеризують будь-який об'єкт класу. Доступ до атрибутів (змінних класу і змінних екземпляру) та методів, можна отримати через оператор крапки.

Об'єкт (Object): Унікальний екземпляр структури даних, яка визначається його класом. Об'єкт містить в собі змінні класу, змінні екземпляру та методи.

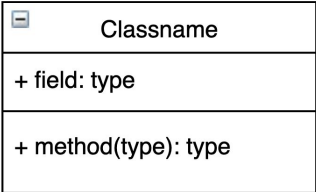
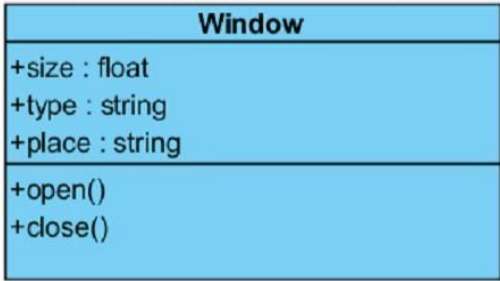
Змінна класу (Class variable): Змінна, яка є спільною для всіх екземплярів класу. Змінні класу визначаються (створюються) в класі але за межами будь-якого з методів класу. Змінні класу використовуються значно рідше ніж змінні екземпляру.

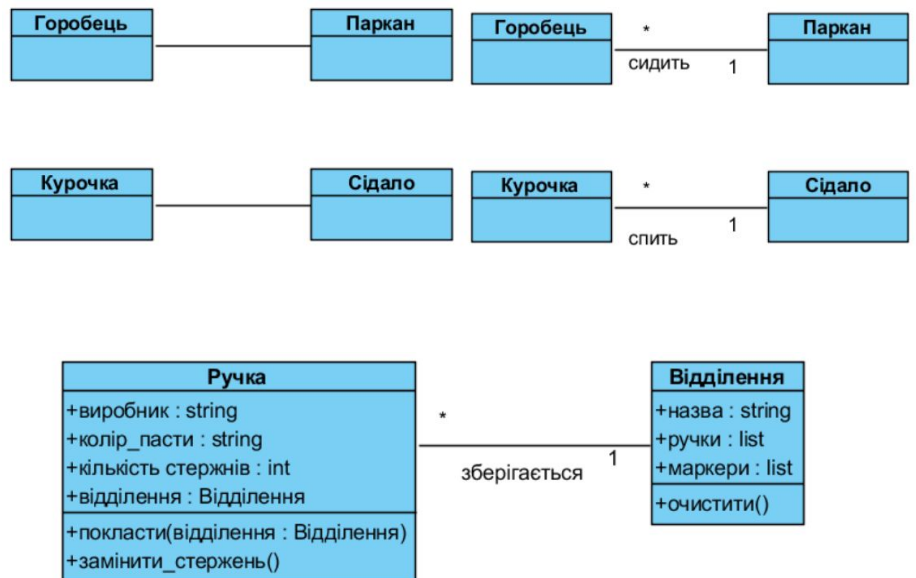
Змінна екземпляру: Змінна, яка визначена (створена) всередині методу і належить тільки поточному екземпляру класу.

Екземпляр: Окремий об'єкт, який є представником окремого класу.

5. Графічні UML примітиви для зображення класів.

Для представлення результатів проектування використовуються UML діаграми. Найпоширеніша UML діаграма це діаграма класів. На діаграмах класів показують класи, які утворюють систему (програму) і їх взаємозв'язки. Діаграми класів називають «статичними діаграмами», оскільки на них показано класи разом з методами і атрибутами, а також статичний взаємозв'язок між ними: те, яким класам «відомо» про існування яких класів, і те, які класи «є частиною» інших класів, — але не показано методи, які при цьому викликаються. Для побудови діаграми класів використовуються наступні графічні примітиви

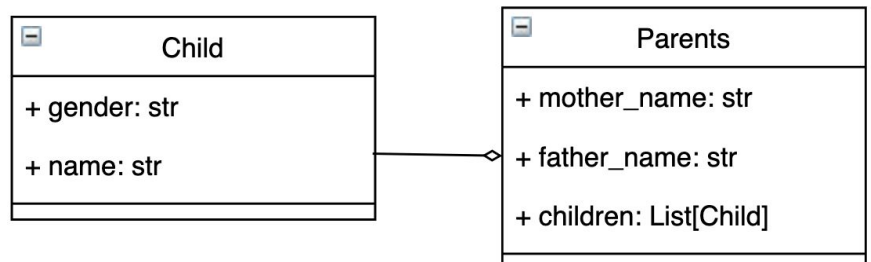
Назва та графічний примітив	Опис
<p>Клас</p> 	<p>Клас визначає атрибути і методи набору об'єктів. Всі об'єкти цього класу (екземпляри цього класу) мають спільну поведінку і однаковий набір атрибутів (кожен з об'єктів має свій власний набір значень). Замість назви «клас» також використовують назву «тип», але тип це більш загальне визначення.</p> <p>Класи позначаються прямокутниками з назвою класу, у цих прямокутниках у вигляді двох «відділень» можуть бути показані атрибути і методи класу.</p> <p>Атрибути показують щонайменше назвою, також може бути показано їх тип, початкове значення та інші властивості. Крім того, атрибути можуть бути показані з областю видимості атрибута:</p> <ul style="list-style-type: none"> • + відповідає <i>публічним (public)</i> атрибутам • # відповідає <i>захищеним (protected)</i> атрибутам • - відповідає <i>приватним (private)</i> атрибутам <p>Методи також показують принаймні назвою, крім того, може бути показано їх параметри і типи значень, які буде повернуто. Методи, як і атрибути, можуть бути показані з областю видимості за допомогою аналогічних до атрибутів позначень:</p> <p>Visual Paradigm Community Edition [not for commercial use]</p> 
<p>Асоціації</p> <p><u>Parent</u> Child</p>	<p>Асоціація означає взаємозв'язок між класами, вона є базовим семантичним елементом і структурою для багатьох типів «з'єднань» між об'єктами.</p> <p>Асоціації є тим механізмом, який надає об'єктам змогу обмінюватися даними між собою. Асоціація описує з'єднання між різними класами. Асоціації можуть виконувати роль, яка визначає призначення асоціації і може бути одно- чи двосторонньою (другий варіант означає, що у межах зв'язку кожен з об'єктів може надсилати повідомлення іншому, перший же — варіанту, коли лише один з об'єктів знає про існування іншого). Крім того, кожен з кінців асоціації має значення численності, яке визначає кількість об'єктів на відповідному кінці асоціації, які можуть мати зв'язок з одним з об'єктів на іншому кінці асоціації.</p> <p>Асоціації позначаються лініями, що з'єднують класи, які беруть участь у зв'язку, крім того, може бути показано роль і численність кожного з учасників зв'язку. Численність буде показано у вигляді діапазону [мін..макс] невід'ємних чисел, зірочка (*) на боці максимального значення позначає нескінченність.</p>



Агрегація



Агрегації є особливим типом асоціацій, за якого два класи, які беруть участь у зв'язку не є рівнозначними, вони мають зв'язок типу «ціле-частина». За допомогою агрегації можна описати, яким чином клас, який грає роль цілого, складається з інших класів, які грають роль частин. У агрегаціях клас, який грає роль цілого, завжди має численність рівну одиниці. Агрегації показують асоціаціями, у яких з боку цілої частини буде намальовано ромб.



```
from typing import List
```

```
class Child:
```

```
    def __init__(self, gender: str, name: str):
        self.gender = gender
        self.name = name
```

```
class Parents:
```

```
    def __init__(self, mother_name: str, father_name:
str):
```

```
        self.mother_name = mother_name
        self.father_name = father_name
        self.children: List[Child] = []
```

```
    def add_child(self, child: Child):
        self.children.append(child)
```

```
mother_name = 'Olena'
```

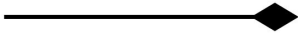


```

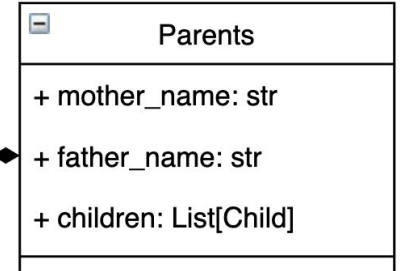
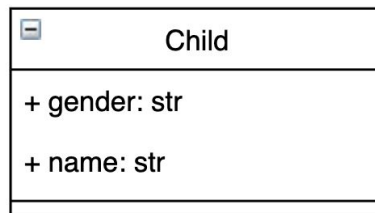
father_name = 'Oleh'
child_gender = 'male'
child_name = 'Bogdan'
child = Child(child_gender, child_name)
parents = Parents(mother_name, father_name)
parents.add_child(child)

```

Композиція



Композиції — це асоціації, які відповідають дуже *сильній* агрегації. Це означає, що у композиціях класи співвідносяться як ціле-частина, але тут зв'язок є настільки сильним, що частини не можуть існувати без цілого. Вони існують лише у межах цілого, після знищення цілого буде знищено і його частини. Композиції показують як асоціації з зафарбованим ромбом з боку цілого.



```
from typing import List
```

```
class Child:
```

```

    def __init__(self, gender: str, name: str):
        self.gender = gender
        self.name = name

```

```
class Parents:
```

```

    def __init__(self, mother_name: str, father_name:
str):

```

```

        self.mother_name = mother_name
        self.father_name = father_name
        self.children: List[Child] = []

```

```

    def add_child(self, gender: str, name: str):
        child = Child(gender, name)
        self.children.append(child)

```

```

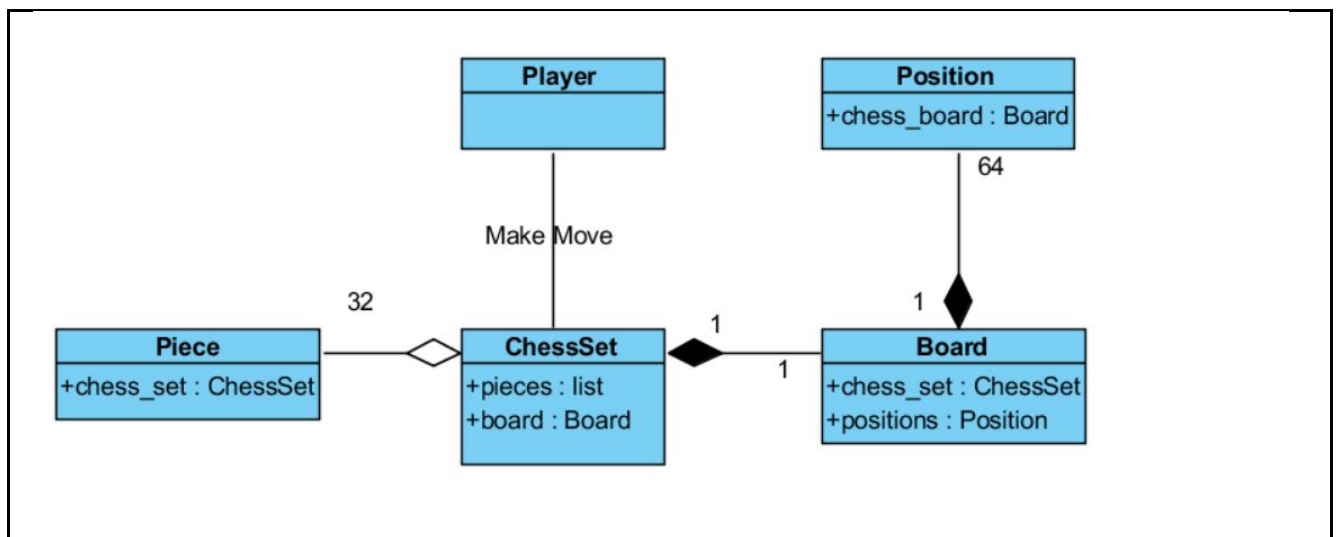
mother_name = 'Olena'
father_name = 'Oleh'
child_gender = 'male'
child_name = 'Bogdan'

```

```

parents = Parents(mother_name, father_name)
parents.add_child(child_gender, child_name)

```

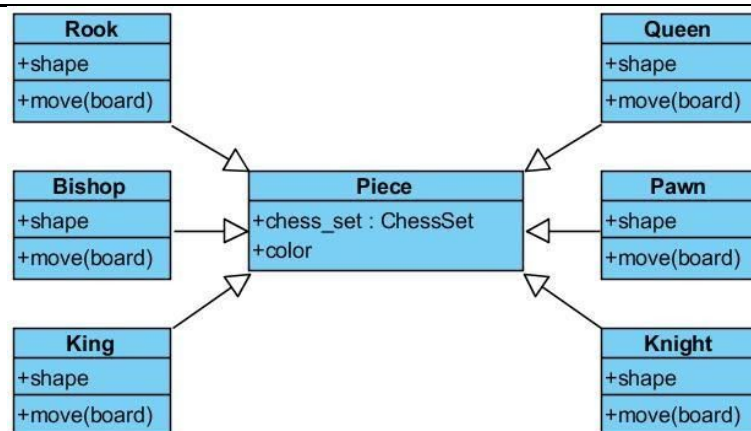


Узагальнення

— Extends —>

Успадкування (наслідування) є однією з фундаментальних основ об'єктно-орієнтованого програмування, у якому клас «отримує» всі атрибути і методи класу, нащадком якого він є, і може перевизначати або змінювати деякі з них, а також додавати власні атрибути і методи.

У UML зв'язок *Узагальнення* між двома класами розташовує їх у вузлах ієрархії, яка відповідає концепції успадкування класу-нащадка від базового класу. Узагальнення показують у вигляді лінії, яка поєднує два класи, зі стрілкою, яку спрямовано від базового класу.



Діаграми класів можна будувати за допомогою спеціалізованого програмного забезпечення.

Наприклад: draw.io, Visual Paradigm, PyCharm плагін тощо.

6. Приклад ООП проектування.

Приклад проектування системи електронного обліку в бібліотеці.

Короткий опис предметної області:

Сучасні бібліотеки використовують електронні каталоги для обліку книжок та пошуку в бібліотечних фондах. Потрібно розробити такий електронний каталог. Каталог містить перелік книжок. Користувачі повинні мати можливість здійснювати пошук за галузями знань, назвами та авторами.

Попередній аналіз:

Книжки однозначно ідентифікуються за International Standard Book Number (ISBN). Також кожна книжка має децимальний номер згідно Dewey Decimal System (DDS), який дозволяє розділити книжки за галузями знань. Попередній аналіз дозволяє встановити, що об'єктами в такій системі будуть *Book* з атрибутами *author*, *title*, *subject*, *ISBN*, та *DDS number*. Також зрозуміло, що каталог повинен забезпечувати пошук. Результат попереднього аналізу можна представити за допомогою наступної діаграми класів, в якій для представлення об'єктів системи призначені класи **Book**, **Author**, **Catalog**:

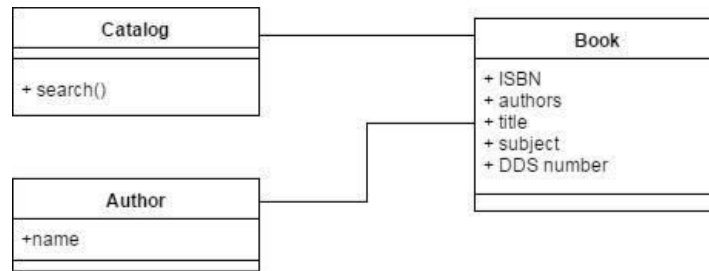


Рис.1. Попередня діаграма класів

Проектування системи:

Подальший аналіз при проектуванні системи показує, що у сучасній бібліотеці зберігаються не тільки книжки а також газети, журнали, DVD та CD диски. Журнали ідентифікуються за назвам, томами і номерами. DVD ідентифікуються за акторами, режисерами та жанрами а CD диски за авторами. Результатом більш докладного аналізу буде наступна діаграма класів:

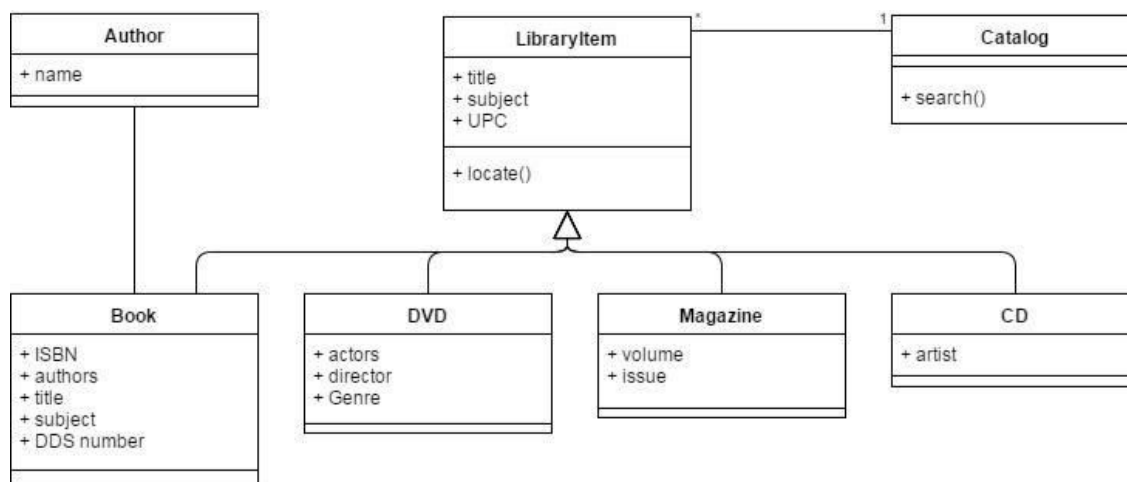


Рис.2. Діаграма класів (варіант 1)

В цій діаграмі використовується зв'язок успадкування між класами **Book**, **DVD**, **Magazine**, **CD** та **LibraryItem** для більш зручної організації бібліотечних фондів. Успадкування доцільно використовувати оскільки всі одиниці з бібліотечних фондів володіють багатьма спільними атрибутами але і мають суттєві відмінності які дозволяють встановити їх місцезнаходження. Для книжок це DDS номер а для інших це UPC номер.

Діаграма класів, яка показана на рис.2 має суттєвий недолік. В цій діаграмі розділені автори книжок та актори і автори CD та DVD дисків. Для авторів книжок передбачено окремий клас а актори і автори CD та DVD дисків це атрибути класів хоча ці всі об'єкти використовуються для представлення людей. Для врахування цього пропонується змінити діаграму і додати окремий клас для всіх осіб, які зробили певний внесок в ту чи іншу одиницю бібліотечного фонд

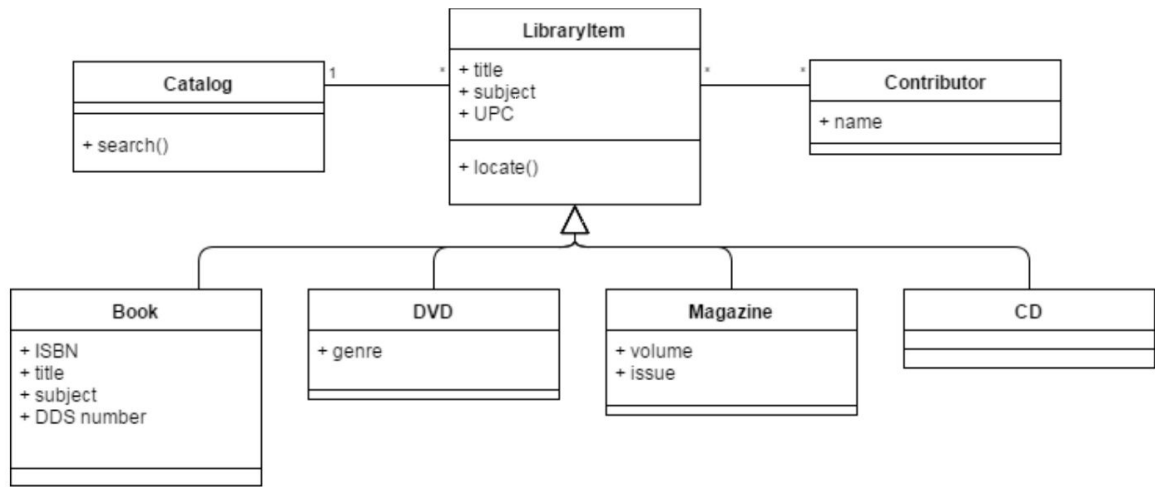


Рис.3. Діаграма класів (варіант 2)

Зміни в діаграмі класів (рис.3) дозволяють врахувати те що як DVD так і книга можуть мати декілька осіб, які їх створювали але разом з тим втрачено інформацію про те, який саме внесок вони зробили. Можна звичайно спробувати це врахувати встановивши інші зв'язки асоціації між класами та додавши метод для виявлення ролі автора. Результат представлено на наступному рисунку:

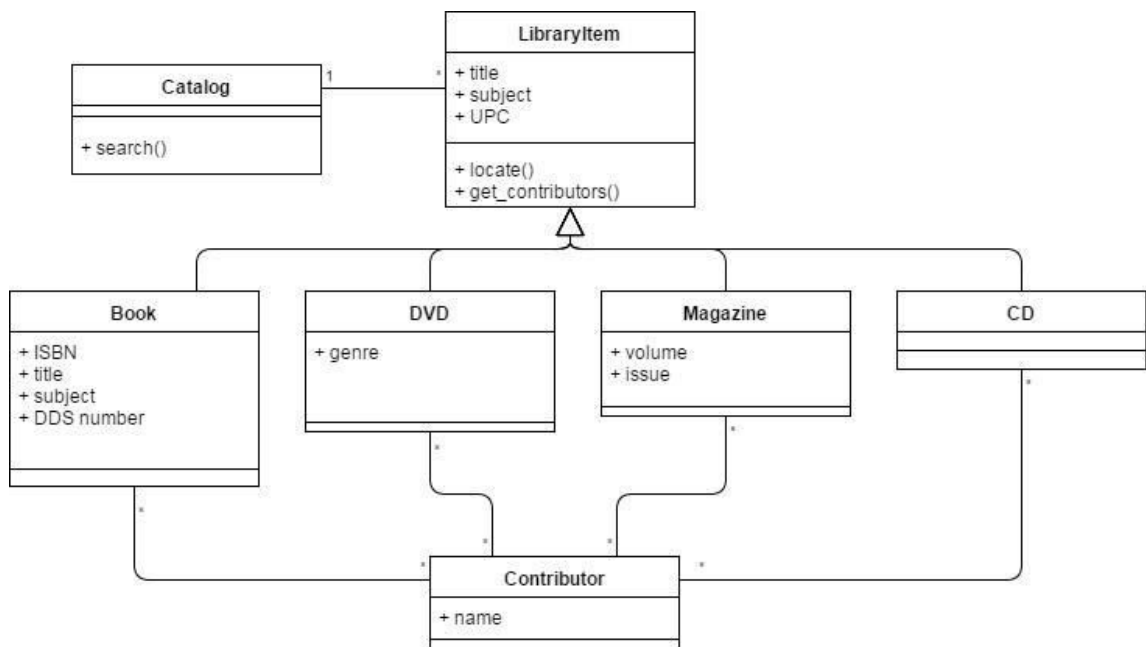


Рис.4. Діаграма класів (варіант 3)

Незважаючи на те, що вдалося позбутися згаданого вище недоліку, система, яка представлена цією діаграмою класів стала доволі складною і для розуміння і для керування і для розширення. Правильним способом модифікації діаграми для врахування ролі кожної особи буде здійснення рефакторингу системи. До системи додається окремий клас для композиції всіх осіб та для ідентифікації їх ролі.

Після проведення проектування системи діаграма класів буде мати такий вигляд:

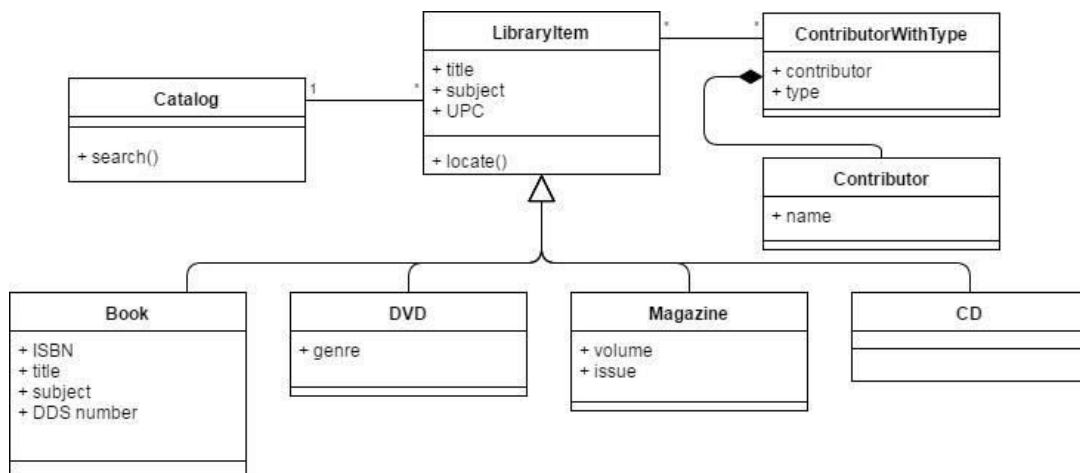


Рис.5. Діаграма класів (остаточний варіант перед реалізацією)

7. Приклад реалізації класів згідно діаграми класів для розроблення програми – Записник (Notebook).

Записник(Notebook) це програма для створення та роботи з щоденними записами. Запис - це короткий текст, який зберігається в Notebook. Кожен запис крім тексту містить дату його створення та може містити тег для організації пошуку в записнику. Користувач повинен мати можливість не тільки здійснювати пошук записів а також і модифікувати їх.

Діаграма класів (рис.6) пояснює наступні елементи:

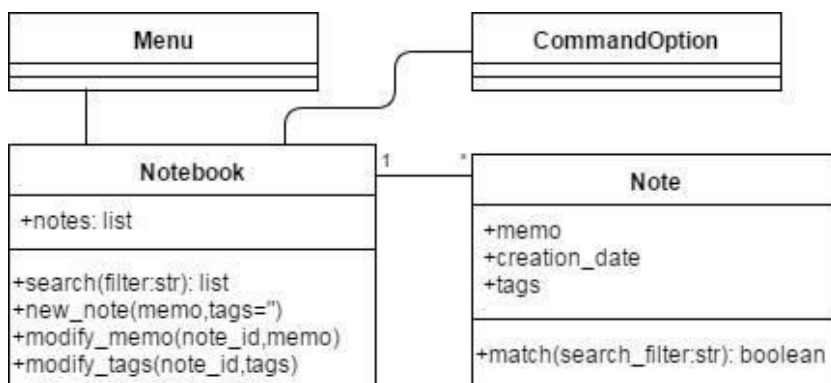


Рис.6. Діаграма класів для програми Notebook.

Література:

1. Python 3 Object-oriented Programming Second Edition , Dusty Phillips, 2015
 - Chapter 1. Object-oriented Design
 - Chapter 2. Objects in Python
2. <https://docs.kde.org>

Вивчення основ використання винятків в Python. Вивчення основ модульного тестування та зневадження в Python

1. Середовище розроблення програмного забезпечення VS Code дозволяє спростити процес створення модульних тестів та здійснення модульного тестування. Детальніше про це можна ознайомитись за посиланням <https://code.visualstudio.com/docs/python/testing>

2. Для зневадження програм використовується потужна програма зневаджувач. За наступними посиланнями можна знайти додаткову інформацію про зневадження в VS Code:

- <https://code.visualstudio.com/docs/python/debugging>
- <https://code.visualstudio.com/docs/editor/debugging>
- <https://donjayamanne.github.io/pythonVSCoDocs/docs/debugging/>
- короткий туторіал - https://code.visualstudio.com/docs/python/python-tutorial#_configure-and-run-the-debugger

3. Наступний приклад демонструє створення модульних тестів для тестування класу Point. Клас Point розроблено для представлення точки і в ньому є два атрибути значення - два float x та y. Для організації тестування використовується метод SetUp, та інші методи для тестування всіх методів цього класу.

```
import unittest

from point_1 import Point
from unittest import TestCase

class TestPoint(TestCase):
    def setUp(self):

        self.A = Point(5, 6)
        self.B = Point(6, 10)
        self.C = Point(5.0, 6.0)
        self.D = Point(-5, -6)

    def test_init(self):

        self.assertEqual((self.A.x, self.A.y), (float(5), float(6)), "Значення не дійсні числа!")
        self.assertEqual((self.B.x, self.B.y), (float(6), float(10)), "Значення не дійсні числа!")
        self.assertEqual((self.C.x, self.C.y), (float(5), float(6)), "Значення не дійсні числа!")
        self.assertEqual((self.D.x, self.D.y), (float(-5), float(-6)), "Значення не дійсні числа!")

    def test_str(self):

        self.assertTrue(str(self.A) == "(5.0, 6.0)", "Неправильний вивід на екран!")
```

```

self.assertTrue(str(self.B) == "(6.0, 10.0)", "Неправильний вивід на екран!")
self.assertTrue(str(self.C) == "(5.0, 6.0)", "Неправильний вивід на екран!")
self.assertTrue(str(self.D) == "(-5.0, -6.0)", "Неправильний вивід на екран!")

def test_eq(self):
    self.assertTrue(self.A == self.C, "Дві однакові точки при тестуванні виявились не однаковими!")
    self.assertFalse(self.A == self.B, "Дві різні точки при тестуванні виявились однаковими!")
    self.assertFalse(self.A == self.D, "Дві різні точки при тестуванні виявились однаковими!!")

def test_ne(self):
    self.assertFalse(self.A != self.C, "Дві однакові точки при тестуванні виявились не однаковими!")
    self.assertTrue(self.A != self.B, "Дві різні точки при тестуванні виявились однаковими!")
    self.assertTrue(self.A != self.D, "Дві різні точки при тестуванні виявились однаковими!")

def test_move(self):
    self.assertEqual((self.A.x, self.A.y), (float(5), float(6)), "Значення не дійсні числа!")
    self.assertEqual((self.B.x, self.B.y), (float(6), float(10)), "Значення не дійсні числа!")
    self.assertEqual((self.C.x, self.C.y), (float(5), float(6)), "Значення не дійсні числа!")
    self.assertEqual((self.D.x, self.D.y), (float(-5), float(-6)), "Значення не дійсні числа!")

def test_rotate(self):
    self.assertEqual(self.A.rotate(180, self.B), self.A.move(7, 14), "Неправильний результат обчислення повороту точки!")
    self.assertEqual(self.A.rotate(360, self.D), self.A.move(5, 6), "Неправильний результат обчислення повороту точки!")

def test_get_xposition(self):
    self.assertEqual(self.A.x, float(5), "Значення x не дійсне число!")
    self.assertEqual(self.B.x, float(6), "Значення x не дійсне число!")
    self.assertEqual(self.C.x, float(5), "Значення x не дійсне число!")
    self.assertEqual(self.D.x, float(-5), "Значення x не дійсне число!")

def test_get_yposition(self):
    self.assertEqual(self.A.y, float(6), "Значення y не дійсне число!")
    self.assertEqual(self.B.y, float(10), "Значення y не дійсне число!")
    self.assertEqual(self.C.y, float(6), "Значення y не дійсне число!")
    self.assertEqual(self.D.y, float(-6), "Значення y не дійсне число!")

if __name__ == '__main__':
    unittest.main()

```



```

from math import pi, sin, cos, radians

class Point:
    'Represents a point in two-dimensional geometric coordinates'

    def __init__(self, x=0, y=0):
        '''Initialize the position of a new point. The x and y
        coordinates can be specified. If they are not, the
        point defaults to the origin.'''
        self.move(x, y)

    def __str__(self):
        return '({0}, {1})'.format(self.x, self.y)

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return (self.x == other.x) and (self.y == other.y)
        else:
            return NotImplemented # False

    def __ne__(self, other):
        if isinstance(other, self.__class__):
            return (self.x != other.x) or (self.y != other.y)
        else:
            return NotImplemented # False

    def move(self, x, y):
        "Move the point to a new location in 2D space."
        self.x = float(x)
        self.y = float(y)

    def rotate(self, beta, other_point):
        'Rotate point around other point'
        dx = self.x - other_point.get_xposition()
        dy = self.y - other_point.get_yposition()
        xpoint3 = dx * cos(beta) - dy * sin(beta)
        ypoint3 = dy * cos(beta) + dx * sin(beta)
        xpoint3 = xpoint3 + other_point.get_xposition()
        ypoint3 = ypoint3 + other_point.get_yposition()
        return self.move(xpoint3, ypoint3)

    def get_xposition(self):
        return self.x

    def get_yposition(self):
        return self.y

```

Детальний аналіз цього прикладу дає можливість зрозуміти, що розроблення тестів допомагає покращити структуру програми, побачити проблеми проектування програми

та те що помилки можуть бути не тільки кодї програми. Модульні тести також можуть містити помилки і тоді зневадження програми стає ще складнішим завданням.

4. Використання інструкції `assert`.

Інструкція `assert` в Python використовується для збудження винятків на етапі відлагодження програм. Насправді це скорочена форма типового шаблону використання інструкції `raise` і це умовна інструкція `raise`. Проста форма `assert expression`, еквівалентна до:

```
if __debug__:
    if not expression: raise AssertionError
```

Розширена форма `assert expression1, expression2`, еквівалентна до:

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

Інструкція `assert` дозволяє додати в програму код для її відлагодження. В розширеній формі `expression1` – вираз, який повинен повертати значення `True` або `False`. Якщо вираз `expression1` поверне значення `False`, інструкція `assert` збуджує виняток `AssertionError` з переданим йому `expression1`. Наприклад:

```
def write_data(file,data):
    assert file, "write_data: файл не визначений!"
```

Інструкція `assert` не повинна містити код програми, який забезпечує безпомилкову роботу програми, тому що цей код не буде виконуватися інтерпретатором, в оптимізованому режимі роботи (такий режим вмикається при запуску інтерпретатора з ключем `-O`). Зокрема, буде помилкою використовувати інструкцію `assert` для перевірки того, що вводить користувач. Інструкцію `assert` використовують для перевірки умов, які завжди повинні бути істинними, а якщо така умова порушується то ця ситуація розглядається, як помилка в програмі, а не як помилка користувача. Наприклад, якщо функція `write_data()` призначена для взаємодії з користувачем то інструкцію `assert` потрібно замінити на звичайну умовну інструкцію `if`.

5. Стратегії зневадження

Рекомендуємо ознайомитись з стратегіями зневадження, які сприятимуть швидшому виявленню помилок в програмі і їх вирішення:

<https://spin.atomicobject.com/2018/08/01/debugging-strategies-tips/>

<https://www.cs.cornell.edu/courses/cs312/2006fa/lectures/lec26.html>

<https://swcarpentry.github.io/python-novice-inflammation/11-debugging/index.html>

Література:

1. Python 3 Object-oriented Programming Third Edition , Dusty Phillips, 2018
Chapter 4. Expecting the Unexpected
2. Python 3 Object-oriented Programming Third Edition , Dusty Phillips, 2018
Chapter 12. Testing Object-oriented Programs

Лабораторна робота №5

Тема. Вивчення взаємозв'язків між об'єктами та їх реалізація в Python. Вивчення основ керування атрибутами в Python.

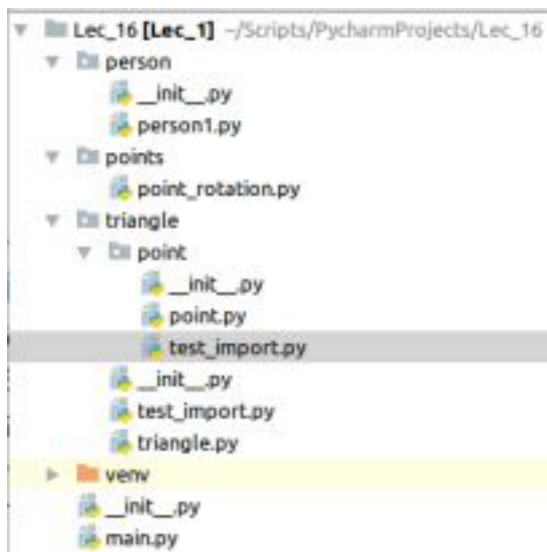
Мета. Отримати навички у використанні успадкування та зв'язків композиції, агрегації і делегування при розробленні класів в Python. Розширення навичок по об'єкто-орієнтованому проектуванню та вивчення основ по керуванню атрибутами в Python.

Короткі теоретичні відомості.

Абсолютні та відносні шляхи імпорту

Збільшення кількості модулів у проекті призводить до необхідності їх впорядкування та організації доступу до них. Для впорядкування модулів у проекті можна їх об'єднувати в пакунки модулів. Пакунок модулів це окрема тека з модулями, які логічно пов'язані між собою, що містить модуль `__init__.py`. Назва теки це ім'я модуля, а наявність `__init__.py` вказує інтерпретатору на те, що ця тека це пакунок модулів. Модуль `__init__.py` це звичайний модуль, але код, який там може міститися буде виконуватися при виконанні імпорту пакунка модулів. Якщо модуль `__init__.py` пустий то він може бути опущений, бо інтерпретатори версій Python 3.3+ трактують будь яку папку з Python модулями, як пакунок модулів. Пакунки модулів доцільно створювати якщо потрібно ізолювати модулі від інших модулів у проекті. Наприклад, якщо в модулі `triangle` визначено клас `Triangle`, а існує пакунок з класом, який має таку саму назву. Також пакунки доцільно створювати у випадку підготовки коду до розповсюдження. При розповсюдженні коду (див. pypi.org) пакунок модулів це той найменший впорядкований обсяг коду, що можна розповсюдити.

На рисунку 1. показано можливу структуру деякого проекту, який містить як окремі модулі так і пакунки модулів.



Пакунок модулів можна розглядати як явний простір імен для модулів і для їх використання потрібно правильно здійснювати імпорт модулів з пакунків модулів. При здійсненні імпорту потрібно вказати шлях до потрібного модуля. Імпортування буде успішним якщо вказати повні (абсолютні) або відносні шляхи. Часто ці два способи імпорту називають абсолютним імпортом та відносним імпортом. В Python 3 всі імпорти по замовчуванню є абсолютними (див. PEP328), що дозволяє при імпорті спочатку виконувати спробу імпортувати стандартні модулі й тільки після цього імпортувати локальні пакунки. Всі шляхи для імпорту зберігаються у списку `sys.path`. Наступний фрагмент коду з модуля `main.py` містить блок інструкцій абсолютного імпорту і в цих інструкціях використовуються оператори точка для побудови повних шляхів до окремих модулів та класів з цих модулів.

```
# main.py
# Absolute imports

import person.person1
person.person1.Person("", "")

from person import person1
person1.Person("", "")

from triangle.point import point
point.Point()

from points import point_rotation
point_rotation()

# import_test.py
# Absolute imports

from person import person1
```

```
import person.person1
pr = person1.Person("", "")
```

```
# Relative imports
from . point import Point
pt = Point()
```

```
from .. import triangle
tr = triangle.Triangle()
```

```
from ...person import person1
person1.Person("", "")
```

```
from .. triangle import Triangle
tr = Triangle(pt, pt, pt)
```

У випадку роботи з проектом зі складною структурою постійно вказувати повний шлях не завжди доречно (хоча це забезпечує від несподіванок при імпорті). В таких випадках можна вказувати відносний шлях при імпорті. Відносний імпорт це спосіб вказати на клас, функцію або модуль, за його розташуванням відносно поточного модуля. Наприклад, фрагмент коду з модуля `import_test.py` пакунку `point` демонструє таку можливість. Потрібно також пам'ятати що якщо в коді використовується явний відносний імпорт (`explicit relative import`) то це код не повинен виконуватися безпосередньо.

Успадкування. Успадкування це один з основних принципів об'єктно-орієнтованого програмування. Успадкування – зв'язок між сутностями типу "a kind of", тобто "родове-видове", "батько-нащадок", "загальне-часткове", тощо). З точки зору програмування, даний тип зв'язку між сутностями задає автоматичне успадкування властивостей одного класу іншим.

```
class Person:

    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    def get_name(self):
        return '{} {}'.format(self.firstname, self.lastname)

class Parent(Person):

    def __init__(self, firstname, lastname):
        super().__init__(firstname, lastname)
        self.kids = []

    def havechild(self, firstname):
        print(self.firstname, "is having a child")
        self.kids.append(Child(self, firstname))

class Child(Person):

    def __init__(self, parent, firstname):
        super().__init__(firstname, parent.lastname)
        self.parent = parent
```

Альтернативою до використання успадкування є використання композиції.

Композиція. Композиція – відношення типу "ціле–частина" між сутностями предметної області. У програмі композиція реалізовується через використання екземплярів одного класу як складових частин іншого класу. Один клас відіграє роль контейнера а інші класи (один або декілька) є його наповненням. При знищенні контейнера весь його вміст – інші об'єкти також буде знищено. Наприклад,

```
class Salary:

    def __init__(self, pay):
        self.pay = pay

    def get_total(self):
        return (self.pay * 12)

class Employee:

    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus
        self.obj_salary = Salary(self.pay) # composition
```

```

def annual_salary(self):

    return "Total: " + str(self.obj_salary.get_total() + self.bonus)

obj_emp=Employee(100,10)
print (obj_emp.annual_salary())

```

Наступний приклад демонструє використання композиції замість успадкування у прикладі з класами Person, Parent, Child.

```

from collections import defaultdict

class Person:

    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    def get_name(self):
        return '{} {}'.format(self.firstname, self.lastname)

class FamilyRegistry:

    def __init__(self):
        self.kids = defaultdict(list)

    def register_birth(self, parent, child_name):
        print(parent.firstname, "is having a child")
        child = Person(child_name, parent.lastname)
        self.kids[parent.lastname].append(child)
        return child

    def print_children(self, person):
        children = self.kids[person.lastname]
        if len(children) == 0:
            print('{} has no children'.format(person.get_name()))
            return
        for child in children:
            print(child.get_name())
            joe = Person('Joe', 'Black')
            jill = Person('Jill', 'White')
            registry = FamilyRegistry()
            registry.register_birth(joe, 'Joe Junior') # Joe is having a child
            registry.register_birth(joe, 'Tina') # Joe is having a child
            registry.print_children(joe) # Joe Junior Black, Tina Black
            registry.print_children(jill) # Jill White has no children

```

Агрегація це подібний до композиції взаємоз'язок між класами.

Агрегація. Агрегація – відношення типу "ціле–частина" між сутностями предметної області. Агрегацію можна розглядати як менш строгу форму композиції. У програмі агрегація реалізовується через використання при визначенні класу атрибутів, які є об'єктами іншого класу. Якщо за аналогією з композицією знищити контейнер об'єктів то його вміст – інші об'єкти залишаться існувати без об'єкта контейнера. Наприклад,

```

class Salary:

    def __init__(self, pay):
        self.pay = pay

```

```

def get_total(self):
    return (self.pay * 12)

class Employee:
    def __init__(self, pay, bonus):
        self.pay = pay
4
        self.bonus = bonus
    def annual_salary(self):
        return "Total: " + str(self.pay.get_total() + self.bonus)

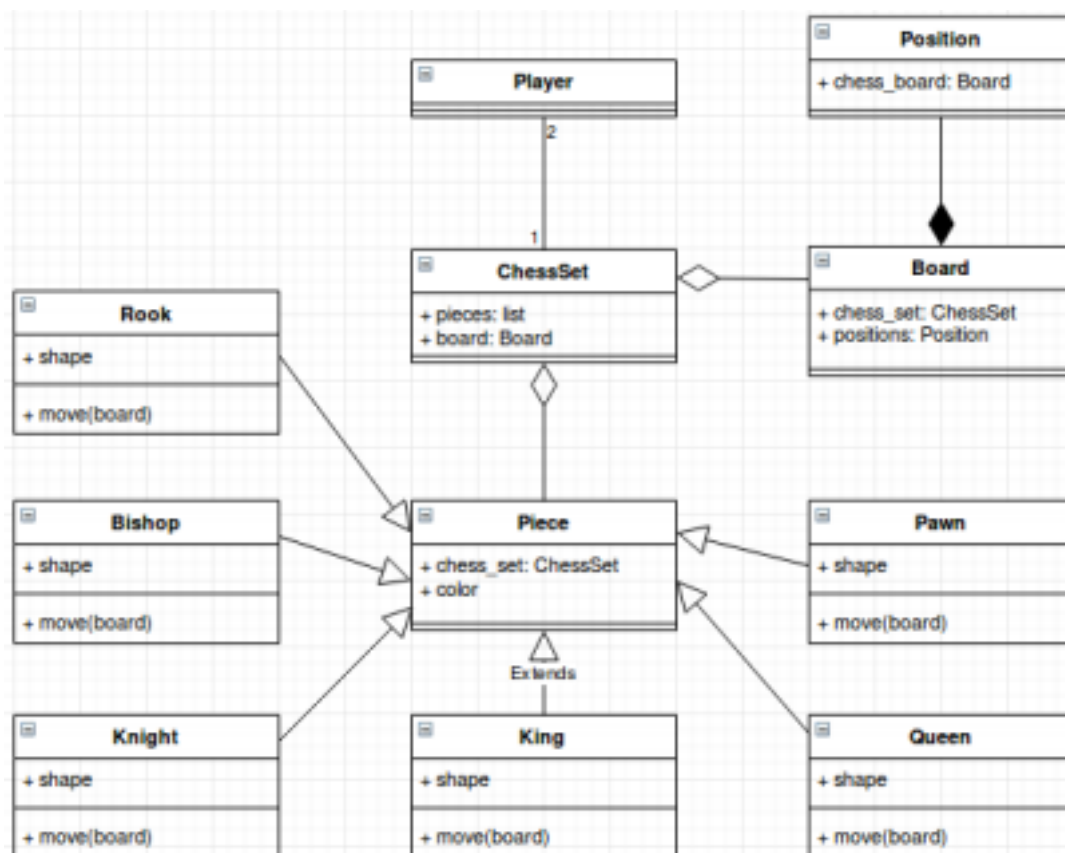
obj_sal = Salary(100)
obj_emp = Employee(obj_sal, 10)
print (obj_emp.annual_salary())

```

Поради по використанню успадкування та композиції (learnpythonthehardway.org)

1. Use composition to package code into modules that are used in many different unrelated places and situations.
2. Use inheritance only when there are clearly related reusable pieces of code that fit under a single common concept or if you have to because of something you're using.
3. Avoid multiple inheritance at all costs, as it's too complex to be reliable. If you're stuck with it, then be prepared to know the class hierarchy and spend time finding where everything is coming from.

В багатьох проектах одночасно використовується і композиція, і агрегація, й успадкування та делегування. Прикладом такого проекту може бути розроблення програми для гри в шахи. Наступна діаграма класів демонструє класи, які потрібно розробити для цієї гри та взаємозв'язки між класами. Між класами ChessSet та Board встановлено взаємозв'язок агрегація бо шахова дошка може існувати (використовуватися) не тільки для цієї гри. Так само шахові фігури можуть бути замінені на інші й між класами ChessSet та Board також зв'язок агрегація. Позиції (клас Position) шахової дошки (клас Board) не можуть існувати окремо від дошки і тут зв'язок — композиція.



1. Використання property.

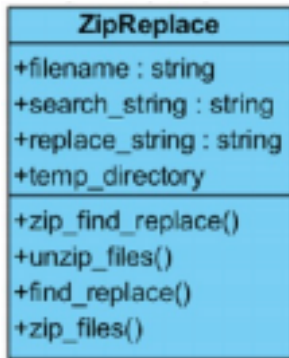
Property це вбудована функція для керування атрибутами яку можна використовувати як декоратор. Property доцільно використовувати якщо є дані а пізніше до них потрібно додати поведінку. В загальному методи це атрибути, які можна викликати. З використанням property можна отримати атрибути, які часто називають властивостями і вони відрізняються від звичайних атрибутів тим що їх можна налаштовувати. Якщо не потрібно контролювати атрибут то не потрібно використовувати property.

Кешування даних це один з прикладів де доцільно використовувати property. Кешування даних – локально зберегти дані, які складно обчислювати або дорого шукати (мережеві запити, запити до бази даних). При отриманні даних здійснюються обчислення чи пошук. Далі дані зберігаються (кешуються) як атрибут об'єкту і наступного разу коли ці дані будуть потрібні - повертаються збережені дані.

Клас WebPage, забезпечує ефективне збереження веб сторінки шляхом використання property. Веб сторінки періодично можуть змінюватися (наприклад, сторінка з меню трапезної <http://www.fest.lviv.ua/uk/projects/uku/> змінюється раз на добу). В класі WebPage реалізовано доступ і збереження сторінки, але без можливості періодичного оновлення збереженої сторінки. Якщо, скористатися модулем time і в property встановити граничний час до якого можна повертати збережену сторінку і після якого потрібно повторити збереження то можна забезпечити оновлення збереженої веб сторінки.

```
1 from urllib.request import urlopen
2 class WebPage:
3     def __init__(self, url):
4         self.url = url
5         self._content = None
6
7     @property
8     def content(self):
9         if not self._content:
10             print("Retrieving New Page...")
11             self._content = urlopen(self.url).read()
12         return self._content
13
14 >>> import time
15 >>> webpage = WebPage("https://www.gutenberg.org/wiki/Animals-Domestic_(Bookshelf
16 >>> now = time.time()
17 >>> content1 = webpage.content
18 Retrieving New Page...
19 >>> time.time() - now
20 5.38809204321634
21 >>> now = time.time()
22 >>> content2 = webpage.content
23 >>> time.time() - now
24 0.0
25 >>> content2 == content1
26 True
```

Для вивчення наступних прикладів керування атрибутами необхідно ознайомитися з інформацією з підручника "Python 3 Object-oriented Programming Second Edition" by Dusty Phillips. Розгляньте реалізацію програми за допомогою якої можна вносити зміни (здійснювати пошук фрагменту тексту і його заміну) в текстові файли, які зберігаються у файлі архіву (zip). Інтуїтивно зрозуміло що в цій програмі потрібно працювати з об'єктами, які представляють файл архіву і окремі текстові файли. Об'єкт, який не так очевидний. це об'єкт менеджер, який повинен забезпечити виконання наступної послідовності кроків: розархівування стисненого файла; здійснення пошуку та заміни; архівування файлів зі змінами. Клас для представлення такого об'єкту повинен в ініціалізаторі приймати назву файла з архівом, та рядки для пошуку та заміни. Для реалізації такого класу потрібно скористатися готовими ресурсами Python: • функцією shutil - utility functions for copying and archiving files and directory trees; • модулем zipfile - read and write ZIP files;

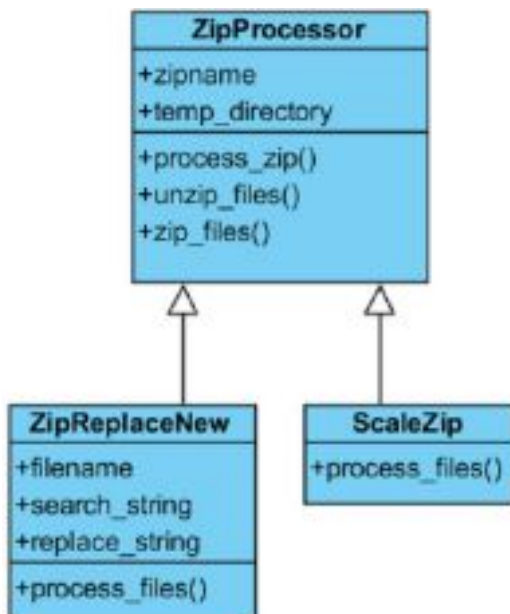


• модулем pathlib, який дозволить працювати з файлами і папками.

Модуль pathlib дозволить створити пусту тимчасову папку для збереження видобутих з архіву файлів а shutil для її видалення. Кожен крок з переліченої вище послідовності кроків доцільно реалізовувати, як окремий метод, що дозволить забезпечити читабельність коду, можливість розширення (для роботи з архівами інших типів потрібно буде перевизначити методи для архівування та розархівування) та розмежування коду, який можна буде використати повторно. Клас ZipReplace на наступній діаграмі класів представляє можливий варіант реалізації, який наведено повністю у підручнику.

Запропонований варіант реалізації повністю задовольняє потреби для

вирішення поставленого завдання, але якщо в майбутньому виникне необхідність працювати з архівами іншого типу то було би добре передбачити таку можливість. Потрібно також врахувати, що вказана послідовність кроків характерна не тільки для вирішення задачі пошуку та заміни у текстових файлах, а також і для інших дій з файлами з архіву, наприклад, масштабування розміру графічних файлів. Якщо скористатися успадкуванням то можна запропонувати реалізацію, яка буде передбачити всі ці можливості.



Наступна діаграма класів демонструє один з варіантів проекту такої програми. Діаграма містить три класи з яких класи ZipReplaceNew та ScaleZip успадковуються від класу ZipProcessor.

Батьківський клас ZipProcessor в методі __init__ приймає тільки назву файла з архівом. Цей клас також містить метод process_zip в

якому здійснюється послідовний виклик методів, які реалізують розархівування, обробку та архівування файлів з архіву; метод unzip_files для розархівування архіву та метод zip_files для архівування файлів та знищення тимчасової папки. Реалізація методу process_files відсутня в цьому класі. Клас ZipReplaceNew успадковується від класу ZipProcessor і потребує при створенні екземпляру цього класу ім'я архіву, рядки для пошуку та заміни. Реалізація методу process_files знаходиться в цьому класі і цей метод доступний для виклику в методі process_zip батьківського класу.

Такий підхід до реалізації обробки файлів в архіві дозволяє

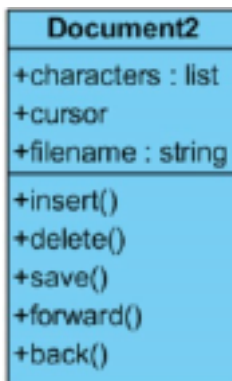
розробити інші класи, які дозволять здійснювати інші види

обробки. Клас ScaleZip, якраз розроблений як один з варіантів такої обробки. Клас ScaleZip успадковується від класу ZipProcessor і містить тільки реалізацію методу process_files для масштабування файлів зображень до розміру 640 на 480 пікселів (найдрібніша одиниця графічного зображення в растровій графіці). Для використання цього класу необхідна бібліотека pillow, яку потрібно встановити за допомогою виконання `pip install pillow` в командному рядку.

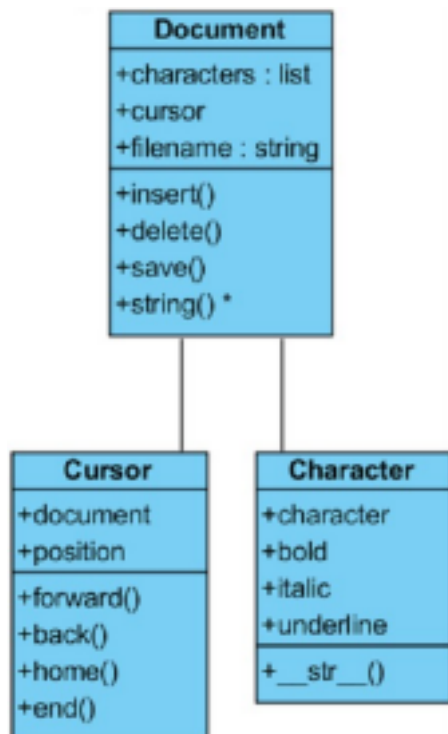
Розгляньте наступний навчальний приклад в якому реалізовано клас Document, який можна використовувати

в текстових редакторах. Наступна діаграма класів пояснює цю реалізацію. Замість очевидного на перший погляд типу даних string для збереження вмісту документу з цією метою використовується тип даних list. Для забезпечення можливості редагування документу потрібно знати поточне положення курсору для чого використовується відповідний атрибут типу int. Також до атрибутів потрібно додати атрибут для представлення імені файлу в якому буде зберігатися документ. Методи цього класу відповідають множині дій, які потрібні для редагування документу (вставити символ, видалити символ, зберегти документ).

Запропонована реалізація обмежена з точки зору розширення можливостей по редагуванню документа та по переміщенню в документі. Розгляньте навчальний приклад в якому реалізовано класи Document, Cursor та Character, які більш придатні до використання в текстовому редакторі чи текстовому процесорі. Для збереження вмісту документу використовується список символів. Для забезпечення переміщення по



документу потрібно знати поточну позицію курсора у цьому списку. Також потрібно зберігати ім'я файлу в якому буде збережено документ. Нижче наведена діаграма класів для представлення документу.



Клас Document дозволяє створювати об'єкти документів, які не будуть спочатку містити жодного символу та не будуть мати імені файлу для збереження, але будуть містити об'єкт класу Cursor, що дозволить переміщатися по документу та відслідковувати поточну позицію курсора. Методи класу Document дозволяють вставляти символи, видаляти символи, та зберігати документ у файл. Також клас Document містить property, яка дозволяє здійснювати вивід рядка символів документу. Клас Cursor отримує об'єкт класу Document в параметрах методу `__init__` що дозволяє отримати доступ до символів документу і реалізувати методи для переміщення на початок документу та в кінець. Також цей клас містить методи для переміщення на одну позицію вперед та на одну позицію назад. Для того щоб забезпечити можливість використання в документі різних типів символів (курсив, потовщені, з підкресленням) використовується клас Character. Клас Character містить атрибути для представлення цих типів символів. Окрім цього в цьому класі перевизначається метод `__str__` що дозволяє отримати рядок, який буде містити позначення типу символу та символ. Оскільки всі класи стосуються вирішення однієї задачі то доцільно їх зберігати в одному модулі `document.py`

Література:

1. Python 3 Object-oriented Programming Second Edition , Dusty Phillips, 2018
Chapter 3. When Objects Are Alike
2. Data Structures and Algorithms in Python, Michael T. Goodrich Roberto Tamassia Michael H. Goldwasser, 2013
Chapter 2. Object-Oriented Programming
3. Python 3 Object-oriented Programming Second Edition , Dusty Phillips, 2018
Chapter 5. When to Use Object-oriented Programming
4. <https://stackoverflow.com>

Лабораторна робота №8

Регулярні вирази

Короткі теоретичні відомості.

1. Вступ

При інтерпретації чи компіляції програм виявлення синтаксичних помилок відбувається дуже швидко, але напевно ми ще не задумувалися яким чином компілятор (інтерпретатор) їх знаходить. Пошук синтаксичних помилок у тексті програм виконується на першому етапі компіляції й цю початкову обробку здійснює лексичний аналізатор. Лексичний аналізатор перевіряє чи відповідають інструкції, ідентифікатори, вирази в рядках програми мові програмування. Мови програмування на відміну від природніх мов є скінченними і їм властивий скінченний набір конструкцій, які можна побудувати на основі правил такої мови. Отже, можна розробити набір шаблонів за якими лексичний аналізатор буде перевіряти текст програм. Дійсно, скінченні мови описуються за допомогою регулярних виразів, які дозволяють побудувати ці шаблони для скінченної кількості рядків та символів. Регулярні вирази визначають граматику такої мови – регулярну граматику.

Поняття регулярного виразу на основі якого будуються шаблони в лексичних аналізаторах використовується не тільки при створенні компіляторів. Більшість мов програмування підтримує єдиний синтаксис та правила побудови регулярних виразів, які використовуються для вирішення багатьох завдань. Регулярний вираз (вислів) (в програмуванні) — це рядок, що описує, або збігається з множиною рядків, відповідно до набору спеціальних синтаксичних правил ([https://uk.wikipedia.org/wiki/Регулярний вираз](https://uk.wikipedia.org/wiki/Регулярний_вираз)). Регулярні вислови використовуються в текстових редакторах та інших допоміжних інструментах для пошуку та зміни тексту на основі заданих шаблонів. Більшість мов програмування дозволяють використовувати регулярні вирази для роботи з рядками.

В Unix подібні операційні системи вбудовано програму командного рядка `grep`, яка дозволяє, крім інших завдань, за допомогою регулярних виразів здійснювати пошук у файлах та виводити на екран результати пошуку. Користувачі операційних систем Windows можуть вирішувати такі завдання за допомогою інших засобів, або за допомогою версії `grep` для Windows (<http://gnuwin32.sourceforge.net/packages/grep.htm>).

2. Прості приклади використання `grep`.

Команда для пошуку відповідностей до “openly” у всіх файлах в поточній теці та у всіх вкладених теках й виведення шляху до файлу, назви файлу та рядків що містять “openly”.

```
(base) oles@oles-370R4E-370R4V-370R5E-3570RE-370R5V:~/home/oles/Intro_Python/blog_dataset/Dataset$ grep -rw openly *
AndroidCentral/9692.txt: <text>You... seem to have no idea what Communism
is. Open Source is definitely not the opposite of communism, the community open
ly and freely contributes to the existence of the OS, Google takes 30% of app pr
ofits and updates the base OS in return. Android is Socialist, if you're going t
o compare apples to oranges.</text>
AndroidCentral/8540.txt: Word's been going around that Firefox will have a brows
er in beta in February. That's a bit odd, as I've been following the development
of Fennec for Maemo and Windows Mobile -- it's being developed openly -- and ha
```

Команда для пошуку та виведення імен файлів в поточній теці, що містять рядок “openly”

```
(base) oles@oles-370R4E-370R4V-370R5E-3570RE-370R5V:~/home/oles/Intro_Python/blog_dataset/Dataset/AndroidCentral$ grep -l openly *
8540.txt
9692.txt
```

Команда для обчислення кількості співпадінь з “the” в кожному з файлів у поточній теці.

```
(base) oles@oles-370R4E-370R4V-370R5E-3570RE-370R5V:~/home/oles/Intro_Python/blog_dataset/Dataset/AndroidCentral$ grep -c the *
10001.txt:5
10002.txt:12
10006.txt:1
10009.txt:12
10011.txt:15
```


Команда для пошуку в файлі рядків, що містять “two” або “too” за допомогою регулярного виразу “t[wo]o”.

```
(base) oles@oles-370R4E-370R4V-370R5E-3570RE-370R5V:~/home/oles/Intro_Python/blog_dataset/Dataset/AndroidCentral$ grep "t[wo]o" 8461.txt
<text>No problems here -- the auto correct on the keyboard is smoking fast too.</text>
<text>I've got two words for you, "carrier" and "marketing".</text>
```

Команда для пошуку в файлі рядків, що починаються з “The” та завершуються “.” за допомогою регулярного виразу “The.*\.\$”.

```
(base) oles@oles-370R4E-370R4V-370R5E-3570RE-370R5V:~/home/oles/Intro_Python/blog_dataset/Dataset/AndroidCentral$ grep "^The.*\.$" 932078.txt
The Age of Wonder: How the Romantic Generation Discovered the Beauty and Terror of Science; by Richard Holmes.
The annotated Alan Turing by Charles Petzold.
```

3. Модуль *re* стандартної бібліотеки Python.

В мові програмування Python також реалізовано підтримку роботи з регулярними виразами. Модуль *re* стандартної бібліотеки Python дозволяє здійснювати пошук за шаблоном в рядках символів та 8 бітних рядках (байтах).

Синтаксис регулярних виразів надзвичайно простий. Кожен окремий символ (наприклад, літера чи цифра) вважається регулярним виразом і за його допомогою можна знайти те, що йому відповідає – такий самий символ. Окремі символи в регулярному виразі можуть поєднуватися, також можуть використовуватися спеціальні символи (метасимволи). Регулярний вираз це рядок, який складається із звичайних та спеціальних символів. Наприклад, рядок “dog” буде регулярним виразом і він може використовуватися як шаблон у функціях модуля *re*. Цьому шаблону будуть відповідати всі рядки, які містять “dog”. Рядок “dog+” також буде регулярним виразом, але цьому шаблону будуть відповідати всі рядки, які починаються на “do” та містять один або більше символів “g”.

Найбільш поширені функції модуля *re* для роботи з регулярними виразами наступні:

re.search(pattern, string, flags=0) - переглядає рядок у пошуку першої відповідності до *pattern* і повертає match object. Повертає None якщо відповідності не було знайдено.
re.match(pattern, string, flags=0) - переглядає рядок у пошуку відповідності до *pattern* на початку *string* і повертає match object. Повертає None якщо відповідності не було знайдено.
re.fullmatch(pattern, string, flags=0) – повертає match object якщо вся *string* відповідає *pattern* і None в іншому випадку.
re.split(pattern, string, maxsplit=0, flags=0) - розділяє рядок за *pattern*.
re.findall(pattern, string, flags=0) - повертає всі відповідності (без їх перекриття) до *pattern* в *string* як список рядків.
re.sub(pattern, repl, string, count=0, flags=0) – повертає рядок в якому всі відповідності (без їх перекриття) до *pattern* в *string* замінені на *repl*.

4. Приклади використання регулярних виразів.

В документації до модуля *re* стандартної бібліотеки Python наведено наступні приклади використання регулярних виразів:

- 6.2.5.1. Checking for a Pair
- 6.2.5.2. Simulating scanf()
- 6.2.5.3. search() vs. match()
- 6.2.5.4. Making a Phonebook
- 6.2.5.5. Text Munging
- 6.2.5.6. Finding all Adverbs
- 6.2.5.7. Finding all Adverbs and their Positions
- 6.2.5.8. Raw String Notation
- 6.2.5.9. Writing a Tokenizer

Кожен з цих прикладів доцільно вивчити для успішного використання в подальшому.

Наступний приклад демонструє як за допомогою регулярних виразів можна вносити зміни в об’єкти DataFrame, які є базовими структурами даних при використанні бібліотеки *pandas*.

У файлі BL-Flickr-Images-Book.csv містяться дані про книжки з Британської бібліотеки (British Library). Для роботи з цими даними зручно скористатися бібліотекою pandas. Можна отримати дані як DataFrame, переглядати дані та доступатися до даних в окремих стовпчиках.

```
>>> import pandas as pd

>>> df = pd.read_csv('BL-Flickr-Images-Book.csv')
>>> df.head()
>>> df['Identifier'].is_unique
>>> df = df.set_index('Identifier')
>>> df.head()

>>> df['Date of Publication']
>>> df.loc[1905:, 'Date of Publication'].head(10)
```

Якщо звернути увагу на результат виконання двох останніх рядків то можна побачити, що дані про дату видання різних книжок записані по різному:

```
Identifier
206          1879 [1878]
216          1868
...
1929          1839, 38-54
...
2956          1860-63
...
5382      1847, 48 [1846-48]
5385          [1897?]
...
4053464          NaN
4112839      [1845.]
...
4114889      [1868]
...
4117752      1883, [1884]
4156359      1898-1912
...
4158128          1831, 32
4159563      [1806]-22
...
4160339          1834-43
```

Оскільки, кожна книжка має мати одну дату видання то регулярний вираз '^(\d{4})' дозволить внести необхідні зміни при побудові нового DataFrame.

```
>>> regex = r'^(\d{4})'
>>> extr = df['Date of Publication'].str.extract(regex, expand=False)
>>> extr.head()
Identifier
206      1879
216      1868
218      1869
472      1851
480      1857
Name: Date of Publication, dtype: object
>>> extr[1929]
1839
>>> extr[5382]
1847
>>> extr[2836]
1897
```

5. Використання регулярних виразів для опрацювання тексту (переклад з *Natural Language Processing with Python*).

Виконання наступних прикладів з коротких теоретичних відомостей передбачає виконання додаткових інструкцій імпорту.

```
>>> from __future__ import division
>>> import nltk, re, pprint
```

5.1 Використання регулярних виразів для виявлення слів за заданими шаблонами.

Багато задач лінгвістичних досліджень передбачають встановлення відповідності заданому шаблону. Наприклад, можна знайти слова, які закінчуються на “ed” використовуючи метод `endswith('ed')`, або інші методи типу `string`. Регулярні вирази є більш потужним і гнучким методом опису шаблонів символів, які необхідно виявити у послідовностях символів.

Для роботи з регулярними виразами у Python потрібно імпортувати модуль `re` скориставшись: `import re`. Для роботи зі словами чи текстами можна використовувати набори даних, які розповсюджуються разом з бібліотекою `nltk`, але це вимагає:

- встановлення бібліотеки `nltk`;
- завантаження наборів даних;
- ознайомлення із засобами цієї бібліотеки для роботи з наборами даних.

На противагу цим крокам можна скористатися текстовими файлами, але це вимагає виконання додаткових кроків для доступу до даних з цих файлів.

Список слів англійської мови на основі якого створено корпус слів англійської мови `Words Corpus` буде використовуватися в якості лінгвістичних даних серед яких буде проводитися пошук. Попередня підготовка списку слів передбачає видалення зі списку власних імен.

```
>>> import re
>>> wordlist = [w for w in nltk.corpus.words.words('en') if w.islower()]
```

5.2 Використання основних метасимволів (операторів повтору).

Синтаксис регулярних висловів залежить від інтерпретатора, що використовується для їх обробки. Однак, із незначними відхиленнями, майже всі поширені інтерпретатори регулярних висловів мають спільні правила.

Найпростіший регулярний вислів, з якого формуються складні вислови, це звичайний окремий символ. Більшість символів (усі літери та цифри), є регулярними висловами, що співпадають із відповідними символами в рядках.

Пошук слів із закінченням `ed` можна здійснити за допомогою регулярного виразу «`ed$`». Потрібно використати функцію `re.search(p, s)`, яка перевіряє чи може зразок `p` бути знайдений у будь-якому місці рядка `s`. Потрібно вказати символи, які шукаємо та використати символ долара, який в регулярних виразах позначає кінець слова:

```
>>> [w for w in wordlist if re.search('ed$', w)]
['abaissed', 'abandoned', 'abased', 'abashed', 'abatished', 'abed',
'aborted', ...]
```

Символ “.” універсальний символ, якому відповідає будь-який один символ. Нехай потрібно знайти слова з восьми літер, де *j* – третя літера та *t* – шоста літера. При створенні регулярного виразу в місцях де може бути будь-який символ вказується крапка. Символ “^” вказує на початок рядка:

```
>>> [w for w in wordlist if re.search('^..j..t..$', w)]
['abjectly', 'adjuster', 'dejected', 'dejectly', 'injector', 'majestic', ...]
```

Виконати самостійно. Повторіть попередній приклад використовуючи регулярний вираз «`..j..t..`». Результати порівняйте.

Символ “?” вказує на те, що попередній символ не є обов'язковим. Вираз «`^e-?mail$`» відповідає двом рядкам *email* та *e-mail*. Можна знайти загальну кількість таких рядків (регулярний вираз дозволяє врахувати різні способи їх запису) у будь-якому тексті скориставшись

```
sum(1 for w in text if re.search('^e-?mail$', w)).
```

Ще зовсім недавно в мобільних телефонах для вводу тексту використовується система **T9** (Рис.1.). Два або більше слів, які можуть бути введені тією самою послідовністю натиснутих клавіш називають **textonyms**.


```
['10-day', '10-lap', '10-year', '100-share', '12-point', '12-year', ...]
>>> [w for w in wsj if re.search('^([a-z]{5})-([a-z]{2,3})-([a-z]{6})$', w)]
['black-and-white', 'bread-and-butter', 'father-in-law', 'machine-gun-toting',
'savings-and-loan']
>>> [w for w in wsj if re.search('(ed|ing)$', w)]
['62%-owned', 'Absorbed', 'According', 'Adopting', 'Advanced', 'Advancing', ...]
```

Виконати самостійно. Повторити попередні приклади й пояснити використання символів : \, {}, (), та | .

Символ зворотної похилої риски (backslash) означає, що наступний метасимвол (спеціальний символ регулярного виразу) втрачає своє спеціальне значення. Отже \. означає просто крапку. Послідовність у фігурних дужках, подібна до {3,5}, вказує на кількість повторів. Вертикальна риска вказує на можливість вибору між елементами, які стоять справа і зліва від неї. Круглі дужки вказують на область дії оператора. Вони можуть використовуватися разом з вертикальною ризкою (диз'юнкція), подібно до виразу «w(i|e|ai|oo)t», який встановлює відповідність до слів wit, wet, wait, та woot.

Загальний перелік метасимволів регулярних виразів наведено у таблиці 1.

Основні метасимволи регулярних виразів

Таблиця 1.

Оператор	Дія
.	Відповідність до будь якого символу
^abc	Відповідність до шаблону abc на початку рядка
abc\$	Відповідність до шаблону abc в кінці рядка
[abc]	Відповідність до одного символу з набору
[A-Z0-9]	Відповідність до одного символу з проміжку
ed ing s	Відповідність до одного з визначених рядків (диз'юнкція)
*	Нуль або більше повторів
+	Один або більше повторів
?	Нуль або один, наприклад. a?, [a-z]? - вказує на опціональність
{n}	Точно n повторів, де n не від'ємне ціле значення
{n,}	Мінімум n повторів
{,n}	Не більше ніж n повторів (максимум)
{m,n}	Мінімум m та максимум n повторів
a(b c)+	Круглі дужки вказують на область дії оператора

Інтерпретатор Python обробляє регулярний вираз, як звичайний рядок. Якщо рядок містить зворотну косу риску то наступний після неї символ інтерпретується спеціальним способом. Наприклад \b буде інтерпретуватися як backspace символ. В загальному, якщо використовувати регулярні вирази з зворотними косими рисками, необхідно вказати інтерпретатору не обробляти цей рядок, а відразу віддати її на обробку модулю re. Це можна зробити додавши до регулярного виразу (рядка) префікс r. Наприклад, рядок r'\band\b' містить два символи \b, які інтерпретовані бібліотекою re відповідають границям слова а не символу backspace.

5.3 Застосування регулярних виразів

Вираз re.search(regex, w) дозволяє знаходити слова w, які відповідають регулярному виразу regex. Регулярні вирази також можна використовувати для виявлення фрагментів слів, або для модифікації слів різними способами.

5.3.1 Виявлення фрагментів слів

Метод re.findall() ("знайти все") дозволяє знайти всі відповідності даному регулярному виразу. В наступному прикладі показано знаходження та підрахунок всіх голосних літер:

```
>>> word = 'supercalifragilisticexpialidocious'
>>> re.findall(r'[aeiou]', word)
['u', 'e', 'a', 'i', 'a', 'i', 'i', 'i', 'e', 'i', 'a', 'i', 'o', 'i', 'o', 'u']
>>> len(re.findall(r'[aeiou]', word))
16
```

Подібним способом можна знайти та побудувати частотний розподіл для послідовностей з двох і більше голосних в довільному тексті:

```
>>> wsj = sorted(set(nltk.corpus.treebank.words()))
>>> fd = nltk.FreqDist(vs for word in wsj
...                     for vs in re.findall(r'[aeiou]{2,}', word))
>>> fd.items()
[('io', 549), ('ea', 476), ('ie', 331), ('ou', 329), ('ai', 261), ('ia', 253),
 ('ee', 217), ('oo', 174), ('ua', 109), ('au', 106), ('ue', 105), ('ui', 95),
 ('ei', 86), ('oi', 65), ('oa', 59), ('eo', 39), ('iou', 27), ('eu', 18), ...]
```

Виконати самостійно. Замініть символ ? у регулярному виразі у виразі `[int(n) for n in re.findall(?, '2009-12-31')]` для представлення рядка з датою '2009-12-31' у вигляді списку `[2009, 12, 31]`.

5.3.2 Обробка фрагментів слів

Використавши `re.findall()` для виявлення фрагментів слів, можна спробувати обробляти ці фрагменти різними способами.

Існує думка, що англійська мова є надлишкова, оскільки текст зручно читати навіть опустивши у словах всі внутрішні голосні. Наприклад у словах, *declaration* - *dclrtn*, та *inalienable* - *inlnble*, залишилися голосні тільки на початку та у кінці слова. Регулярний вираз у наступному виразі встановлює відповідність до початкових послідовностей голосних, кінцевих послідовностей голосних та всіх приголосних, а всі інші символи ігноруються. Метод `re.findall()` використовується для виявлення фрагментів слів а метод `''.join()` для їх поєднання.

```
>>> regexp = r'^[AEIOUaeiou]+|[AEIOUaeiou]+$|^[^AEIOUaeiou]
>>> def compress(word):
...     pieces = re.findall(regexp, word)
...     return ''.join(pieces)
...
>>> english_udhr = nltk.corpus.udhr.words('English-Latin1')
>>> print nltk.tokenwrap(compress(w) for w in english_udhr[:75])
Unvrsl Dclrtn of Hmn Rghts Prmble Whrs rcgntn of the inhrnt dgnty and of the eql
and inlnble rghts of all mmbrs of the hmn fmly is the fndtn of frdm , jstce and
pce in the wrld , Whrs dsrgd and cntmpt fr hmn rghts hve rsldt in brbrs acts
whch hve outrgd the cnsncne of mnknd , and the advnt of a wrld in whch hmn bngs
shll enjy frdm of spch and
```

В наступному прикладі показано спільне використання регулярного виразу та умовного частотного розподілу. В програмі зі слів мовою Rotokas вилучаються всі послідовності приголосна – голосна. Оскільки вилучаються пари то на їх основі можна побудувати умовний частотний розподіл й представити його у вигляді таблиці:

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in re.findall(r'[ptksvr][aeiou]',
w)]
>>> cfd = nltk.ConditionalFreqDist(cvs)
>>> cfd.tabulate()
      a    e    i    o    u
k  418  148   94  420  173
p   83   31  105   34   51
r  187   63   84   89   79
s    0    0  100    2    1
t   47    8    0  148   37
v   93   27  105   48   49
```

Переглянувши рядки *s* та *t*, бачимо що ці приголосні утворюють цілком окремі фонemi в цій мові. Літеру *s* можна видалити з абетки Rotokas і записати правило, що літера *t* вимовляється як *s* перед *i*. (Тільки одне слово *kasuari* містить *su*, оскільки це запозичення з англійської, 'cassowary').

Для перегляду слів, які відповідають тим чи іншим послідовностям приголосна – голосна зручно побудувати структуру де слова згруповані за тими послідовностями, які в них зустрічаються. Цього можна досягнути наступним чином:

```
>>> cv_word_pairs = [(cv, w) for w in rotokas_words
...                  for cv in re.findall(r'[ptksvr][aeiou]', w)]
>>> cv_index = nltk.Index(cv_word_pairs)
>>> cv_index['su']
['kasuari']
>>> cv_index['po']
```

```
['kaapo', 'kaaporato', 'kaipori', 'kaiporipie', 'kaiporivira', 'kapo', 'kapoa',
'kapokao', 'kapokapo', 'kapokapo', 'kapokapo', 'kapokapo', 'kapokapo', 'kapokapo', ...]
```

В цьому фрагменті програми обробляється кожне слово, для якого здійснюється пошук підрядків, які відповідають регулярному виразу «[ptksvr][aeiou]». У випадку слова *kasuari*, будуть знайдені *ka*, *su* та *ri* і список `cv_word_pairs` буде містити ('*ka*', '*kasuari*'), ('*su*', '*kasuari*') та ('*ri*', '*kasuari*'). Наступний крок це за допомогою `nltk.Index()`, перетворити цей список до вигляду: `defaultdict(<type 'list'>, {'va': ['kaaova', 'Kaareva', 'kaava', 'kaavaava', 'kakuva', 'kapaava', 'karaava', 'karaova', ...]`

5.3.3 Встановлення основ слів (стемінг)

Для обробки мови часто буває необхідно опустити закінчення слів і працювати тільки з їх основами. Існують різні способи встановлення основи слова. Найпростіший спосіб, це відкидання будь-яких послідовностей, які нагадують суфікс.

```
>>> def stem(word):
...     for suffix in ['ing', 'ly', 'ed', 'ious', 'ies', 'ive', 'es', 's',
... 'ment']:
...         if word.endswith(suffix):
...             return word[:-len(suffix)]
...     return word
```

Подібний результат можна отримати за допомогою регулярних виразів. Спочатку потрібно розробити вираз із диз'юнкцією де записати всі суфікси та використати круглі дужки для вказання області дії операції диз'юнкції.

```
>>> re.findall(r'^.*(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
['ing']
```

Тут, `re.findall()` знаходить тільки суфікс, хоча регулярний вираз відповідає всьому слову. Це сталося тому, що круглі дужки задають не тільки область дії оператора диз'юнкції, але і виконують функцію вибору підрядка який потрібно вилучити. У випадку, якщо потрібно в регулярному виразі використовувати круглі дужки для вказання області дії оператор, але не потрібно здійснювати вилучення, в регулярний вираз потрібно додати `?:`, так як у наступному прикладі.

```
>>> re.findall(r'^.*(?:ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
['processing']
```

Оскільки, потрібно поділити слово на основу і суфікс та показати ці частини, то доцільно їх виділити в регулярному виразі за допомогою круглих дужок:

```
>>> re.findall(r'^.(.*) (ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
```

Спробуємо аналогічно обробити інше слово, наприклад, *processes*:

```
>>> re.findall(r'^.(.*) (ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
```

Регулярний вираз помилково знайшов суфікс *-s* замість суфікса *-es*. Оператор `*` в частині виразу «`.*`» приводить до поглинання максимальної кількості символів. Доцільно використати вираз `*?`, який дозволяє уникнути цього ефекту:

```
>>> re.findall(r'^.(.*?) (ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
```

Даний регулярний вираз працює навіть зі словами з відсутнім суфіксом:

```
>>> re.findall(r'^.(.*?) (ing|ly|ed|ious|ies|ive|es|s|ment)?$', 'language')
[('language', '')]
```

Звичайно цей підхід має ще багато інших проблем. Для їх виявлення розроблена функція для здійснення операції стемінгу. За її допомогою можна опрацювати довільний текст:

```
>>> def stem(word):
...     regexp = r'^.(.*?) (ing|ly|ed|ious|ies|ive|es|s|ment)?$'
...     stem, suffix = re.findall(regexp, word)[0]
...     return stem
...
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government. Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
```

```
>>> tokens = nltk.word_tokenize(raw)
>>> [stem(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'women', 'ly', 'in', 'pond',
'distribut', 'sword', 'i', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
',', 'Supreme', 'execut', 'power', 'deriv', 'from', 'a', 'mandate', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcical', 'aquatic', 'ceremony',
',']
```

Потрібно зазначити, що регулярний вираз видалив *s* не тільки зі слова *ponds* а також зі слів *is* та *basis*. Також утворені слова невластиві мові *distribut* та *deriv*, хоча ці основи прийнятні для певного застосування.

5.3.4 Пошук у токенизованому тексті

Спеціальний тип регулярних виразів може використовуватися для пошуку серед слів у тексті (текст – послідовність окремих слів). Наприклад, за допомогою виразу "<a> <man>" можна знайти всі випадки вживання *a* та *man* в тексті. Кутові дужки використовуються для позначення меж а всі пробіли між цими дужками ігноруються (індивідуальна особливість NLTK's `findall()` методу для тексту). Наступний приклад містить `<.*>` [#1](#) для виявлення всіх окремих слів, а круглі дужки дозволяють вибрати ці слова окремо від словосполучень (*a monied man*). Інший приклад дозволяє знайти всі словосполучення з трьох слів де останнє слово *bro* [#2](#). Останній приклад це знаходження послідовностей з трьох і більше слів, які починаються з літери *l* [#3](#).

```
>>> from nltk.corpus import gutenberg, nps_chat
>>> moby = nltk.Text(gutenberg.words('melville-moby_dick.txt'))
>>> moby.findall(r"<a> (<.*>) <man>") #1
monied; nervous; dangerous; white; white; white; pious; queer; good;
mature; white; Cape; great; wise; wise; butterless; white; fiendish;
pale; furious; better; certain; complete; dismasted; younger; brave;
brave; brave; brave
>>> chat = nltk.Text(nps_chat.words())
>>> chat.findall(r"<.*> <.*> <bro>") #2
you rule bro; telling you bro; u twizted bro
>>> chat.findall(r"<1.*>{3,}") #3
lol lol lol; lmao lol lol; lol lol lol; la la la la la; la la la; la
la la; lovely lol lol love; lol lol lol.; la la la; la la la
```

Виконати самостійно. Для поглиблення розуміння особливостей роботи з регулярними виразами використайте функцію `nltk.re_show(p, s)`, яка у рядку *S* виділяє всі частини, які відповідають шаблону *p*. Для дослідження регулярних виразів зручно використати програму *neto_app.py* або онлайн сервіси, наприклад, <https://pythex.org/>.

Побудова пошукових шаблонів для вивчення лінгвістичних явищ, які пов'язані зі словами не є складною. Чи можна будувати складніші шаблони на основі регулярних виразів? Наприклад, для знаходження в корпусі текстів словозворотів таких, як *x and other ys*, які дозволяють дослідити гіперніми, можна розробити наступний фрагмент програми

```
>>> from nltk.corpus import brown
>>> hobbies_learned = nltk.Text(brown.words(categories=['hobbies', 'learned']))
>>> hobbies_learned.findall(r"<\w*> <and> <other> <\w*s>")
speed and other activities; water and other liquids; tomb and other landmarks;
Statues and other monuments; pearls and other jewels; charts and other items;
roads and other features; figures and other objects; military and other areas;
demands and other factors; abstracts and other compilations; iron and other metals
```

При достатніх об'ємах лінгвістичних даних дана програма дозволяє зібрати інформацію про таксономію об'єктів без додаткової ручної праці. Звичайно, результати роботи програми містять і хибні результати, якими потрібно нехтувати. Наприклад, результат: *demands and other factors* вказує на те що *demand* це тип *factor*, але в цьому реченні йдеться про збільшення заробітної плати (wage demands).

Виконати самостійно. Використовуючи шаблон *as x as y* дослідити інформацію про об'єкти та їх властивості.

5.4 Використання регулярних виразів для токенизації тексту

Регулярні вирази дозволяють також здійснювати токенизацію текстів та контролювати процес токенизації.

5.4.1 Найпростіший токенизатор

Найпростіший спосіб токенизації тексту – це поділ його за пробілами. Розглянемо наступний текст з *Alice's Adventures in Wonderland*:

```
>>> raw = """'When I'M a Duchess,' she said to herself, (not in a very hopeful tone
... though), 'I won't have any pepper in my kitchen AT ALL. Soup does very
... well without--Maybe it's always pepper that makes people hot-tempered,'..."""
```

Можна розділити цей текст за пробілами скориставшись `raw.split()`. Реалізуючи те саме за допомогою регулярних виразів, недостатньо встановити відповідність до всіх символів пробілів [#1](#). Оскільки, рядок з текстом містить символ `\n`, потрібно враховувати будь-яку кількість пробілів табуляцій та символів нового рядка [#2](#):

```
>>> re.split(r' ', raw) #1
[''when', 'I'M', 'a', 'Duchess,', '', 'she', 'said', 'to', 'herself,', '(not', 'in',
'a', 'very', 'hopeful', 'tone\nthough)', '', 'I', 'won't', 'have', 'any', 'pepper',
'in', 'my', 'kitchen', 'AT', 'ALL.', 'Soup', 'does', 'very\nwell', 'without-
Maybe', 'it's', 'always', 'pepper', 'that', 'makes', 'people', 'hot-
tempered,...']
>>> re.split(r'[\t\n]+', raw) #2
[''when', 'I'M', 'a', 'Duchess,', '', 'she', 'said', 'to', 'herself,', '(not', 'in',
'a', 'very', 'hopeful', 'tone', 'though)', '', 'I', 'won't', 'have', 'any',
'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL.', 'Soup', 'does', 'very', 'well',
'without-Maybe', 'it's', 'always', 'pepper', 'that', 'makes', 'people', 'hot-
tempered,...']
```

Регулярний вираз `«[\t\n]+»` встановлює відповідність одному, або більше пробілам, табуляціям (`\t`) чи символам нового рядка (`\n`). Інші символи пробілів, такі як «Enter» «нова сторінка» також повинні враховуватися. Замість того щоб доавати нові елементи до регулярного виразу, можна використати властивий `re` символ, `\s`, який означає будь-який символ пробілу. Вираз із попереднього прикладу буде переписаний наступним чином `re.split(r'\s+', raw)`.

Поділ тексту на окремі слова за пробілами дає очікувані помилкові результати `'(not'` та `'herself,'`. Python підтримує символ `\w`, якому відповідає клас символів слів аналогічних до `[a-zA-Z0-9_]`, та символ `\W`, який визначає клас символів – всі символи не літери, не цифри й не підкреслення. Можна використати `\w` в регулярному виразі для поділу вхідного тексту на окремі слова за будь-якими символами відмінним від символів з яких складаються слова:

```
>>> re.split(r'\W+', raw)
['', 'when', 'I', 'M', 'a', 'Duchess', 'she', 'said', 'to', 'herself', 'not',
'in', 'a', 'very', 'hopeful', 'tone', 'though', 'I', 'won', 't', 'have',
'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL', 'Soup', 'does', 'very',
'well', 'without', 'Maybe', 'it', 's', 'always', 'pepper', 'that', 'makes',
'people', 'hot', 'tempered', '']
```

Результати використання цього регулярного виразу містять пусті рядки на початку і в кінці списку (спробуйте `'xx'.split('x')`). Аналогічні слова, але без додаткових пустих рядків можна отримати використавши `re.findall(r'\w+', raw)`, де шаблону відповідають всі слова замість пробілів. Наступний регулярний вираз `«\w+|\S\w*»` дозволяє охопити більшу кількість різних випадків. Спочатку цей вираз встановлює відповідність до будь-яких послідовностей символів слів, далі, якщо відповідностей більше немає, встановлюється відповідність до всіх символів, які не є символами пробілу (`\S`) і зустрічаються перед символами слів. Такий підхід дозволяє відділяти розділові знаки разом з літерою перед якою вони йдуть (наприклад `'s`), але послідовності двох і більше розділових знаків розділені.

```
>>> re.findall(r'\w+|\S\w*', raw)
[''when', 'I', 'M', 'a', 'Duchess', '', '', 'she', 'said', 'to', 'herself', '',
'(not', 'in', 'a', 'very', 'hopeful', 'tone', 'though', ')', '', 'I', 'won',
't', 'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL', 'Soup',
'does', 'very', 'well', 'without', '-', '-Maybe', 'it', 's', 'always', 'pepper',
'that', 'makes', 'people', 'hot', '-tempered', '', '', 's', 's', 's']
```

Для виявлення слів з дефісом або апострофом вираз `\w+` потрібно розширити до: `«\w+([-']\w+)*»`. Цей вираз означає, що після `\w+` може йти нуль, або більше випадків `[-']\w+`; що дозволить виділити слова *hot-tempered* або *it's*. Символи `?`: також потрібно додати до регулярного виразу, разом з шаблоном для виявлення символів лапок.

```
>>> print re.findall(r"\w+([?[-']\w+)*|'[-.()]+|\S\w*", raw)
['', 'when', 'I'M', 'a', 'Duchess', '', '', 'she', 'said', 'to', 'herself', '',
```



```
'(', 'not', 'in', 'a', 'very', 'hopeful', 'tone', 'though', ')', ' ', ' ', ' ', ' ', 'I',
'won't', 'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL', ' ', ' ', 'Soup',
'does', 'very', 'well', 'without', ' ', ' ', 'Maybe', 'it's', 'always', 'pepper',
'that', 'makes', 'people', 'hot-tempered', ' ', ' ', ' ', ' ', '...']
```

Частина виразу «[-. ()+» дозволяє виявити подвійний дефіс, еліпсис а також відкриту дужку.

Таблиця 2 містить список символів регулярних виразів.

Символи в регулярних виразах.

Таблиця 2

Символ	Функція
\b	Межі слова
\d	Будь-яка десяткова цифра (== [0-9])
\D	Будь-яка не десяткова цифра (== [^0-9])
\s	Будь-який символ пробілу (== [\t\n\r\f\v])
\S	Будь-який не символ пробілу (== [^ \t\n\r\f\v])
\w	Будь-який символ літери чи цифри (== [a-zA-Z0-9_])
\W	Будь-який не символ літери чи цифри (== [^a-zA-Z0-9_])
\t	Символ табуляції
\n	Символ нового рядка

5.4.2 Токенізатор на основі регулярних виразів в NLTK

Функція `nltk.regexp_tokenize()` подібна до `re.findall()` (у випадку використання останньої для токенизації). Але `nltk.regexp_tokenize()` є більш ефективною для здійснення цієї операції, та не потребує використання круглих дужок. Для покращення читабельності регулярний вираз поділений на окремі рядки до яких доданий коментар. Спеціальний (?x) "verbose flag" – вказує Python на те, що оператор (інструкція) складається з декількох частин й пробілами між ними та коментарями потрібно знехтувати.

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)          # set flag to allow verbose regexps
...     ([A-Z]\.)+             # abbreviations, e.g. U.S.A.
...     | \w+(-\w+)*           # words with optional internal hyphens
...     | \$?\d+(\.\d+)?%?     # currency and percentages, e.g. $12.40, 82%
...     | \.\.\.               # ellipsis
...     | [[\.,;"'?:()-_`]]    # these are separate tokens
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

У випадку використання «verbose flag», символ ' ' пробілів потрібно замінити на \s. Функція `regexp_tokenize()` може містити не обов'язковий параметр `gaps`. Коли цей параметр має значення `True`, регулярний вираз виділить проміжки між словами, аналогічно до `re.split()`.

Виконати самостійно. Перевірити правильність роботи токенизатора, порівнявши списки `set(tokens).difference(wordlist)`

5.4.3 Проблеми токенизації тексту

Токенизація - це доволі складне завдання, яке часто не має однозначного розв'язку. Спосіб токенизації та правила, що вважати окремим словом вибираються в залежності від предметної області.

Для полегшення розробки токенизаторів існує можливість доступу до текстів, які токенозовані вручну. Порівнюючи результати токенизації з таким «gold-standard» токенизатором можна оцінити якість роботи програми. Набір корпусів NLTK містить приклади Penn Treebank, а саме тексти Wall Street Journal (`nltk.corpus.treebank_raw.raw()`) та їх токенозовану версію (`nltk.corpus.treebank.words()`).

Виконати самостійно. Для освоєння елементів роботи з регулярними виразами доцільно виконати наступні вправи:

1. Описати, які класи рядків відповідають наступному регулярному виразу. `[a-zA-Z]+`. Результати перевірити за допомогою `re.findall()`
2. Описати, які класи рядків відповідають наступному регулярному виразу. `[A-Z][a-z]*`. Результати перевірити за допомогою `re.findall()`

3. Описати, які класи рядків відповідають наступному регулярному виразу. `\d+(\.\d+)?`.
Результати перевірити за допомогою `re.findall()`
4. Описати, які класи рядків відповідають наступному регулярному виразу. `([^\aeiou][aeiou][^\aeiou])*`. Результати перевірити за допомогою `re.findall()`
5. Описати, які класи рядків відповідають наступному регулярному виразу. `\w+|^[^\w\s]+..`.
Результати перевірити використовуючи за допомогою `re.findall()`
6. Описати, які класи рядків відповідають наступному регулярному виразу. `p[aeiou]{,2}t`.
Результати перевірити використовуючи `nlk.re_show()`
7. Написати регулярний вираз, який встановлює відповідність наступному класу рядків : арифметичний вираз з цілими значеннями, який містить операції множення та додавання (наприклад, $2*3+8$).

Література.

1. Документація стандартної бібліотеки.
 re — Regular expression operations <https://docs.python.org/3.6/library/re.html>
 Regular Expression HOWTO <https://docs.python.org/3.6/howto/regex.html#regex-howto>
2. Python for Everybody. Regular expressions. <https://www.py4e.com/html3/11-regex>
3. Natural Language Processing with Python – Analyzing Text with the Natural Language Toolkit Steven Bird, Ewan Klein, and Edward Loper. Chapter 3. Processing Raw Text.
4. Real Python Tutorials <https://realpython.com/python-command-line-arguments/#regular-expressions>
<https://realpython.com/python-data-cleaning-numpy-pandas/>