**Модуль 2: Python Development**

**Заняття 1: Development Tools**

**Менеджер пакетів та віртуальних середовищ pipenv**

Pipenv — це сучасний інструмент для управління робочим середовищем у Python.

Основні можливості pipenv:

* Створення та управління віртуальним середовищем.
* Синхронізація пакетів у Pipfile під час встановлення та видалення пакетів.
* Автоматичне підвантаження змінних середовища з файлу .env.

Коли на одному комп'ютері накопичується кілька віртуальних середовищ для різних проектів, над якими ведеться робота, неминуче виникає проблема управління набором віртуальних середовищ, підтримки в актуальному і робочому стані, видалення непотрібних для звільнення місця тощо. Ці завдання вирішує **Pipenv** за рахунок створення одного місця (папки) для зберігання всіх віртуальних середовищ, з якими ви працюєте.

Управління залежностями, коли ваш застосунок може перестати працювати через конфлікт версій встановлених бібліотек, — це одна з головних проблем розробників Python.

Використовувати **requirements.txt**, звичайно, можна, але вимагає додаткових зусиль для підтримки актуальних версій пакетів. Оновлення пакетів до нових версій може стати справжньою проблемою для застосунків, де багато залежностей.

**Pipenv** частково вирішує цю проблему, рекурсивно перевіряючи наявність конфліктів залежностей усіх пакетів і намагаючись встановити останні версії пакетів, які не викликають конфліктів.

Для розгортання застосунків на клієнтських комп'ютерах або серверах потрібно мати набір інструкцій для розгортання середовища, про яке ми знаємо, що воно працює. Зазвичай таке середовище — це середовище комп'ютера розробника, у якому ведеться розробка. Щоб повторити середовище на іншому комп'ютері, потрібно встановити ті самі версії бібліотек, Python і всі залежності. Це найпростіший та найінтуїтивніший спосіб уникнути конфлікту версій встановлених бібліотек. Pipenv вирішує цю проблему, генеруючи файл з точними версіями всіх бібліотек віртуального середовища та даючи можливість встановити з цього файлу "зліпок" працюючої системи.

Зараз **Pipenv** є фактично стандартом для розробників і вміти з ним працювати — просто необхідно.

**Встановлення Pipenv**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/pipenv#%D0%B2%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-pipenv)**Pipenv** є пакетом Python, так само, як і **pip**, але не входить у стандартне постачання Python, і його потрібно встановлювати. Встановити можна за допомогою **pip** команди:

pip install pipenv --user

**Робота з віртуальними середовищем Pipenv**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/pipenv#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D0%B2%D1%96%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B8%D0%BC%D0%B8-%D1%81%D0%B5%D1%80%D0%B5%D0%B4%D0%BE%D0%B2%D0%B8%D1%89%D0%B5%D0%BC-pipenv)Для створення середовища читає ім'я папки, в якій працює і увесь шлях до неї. Всі віртуальні середовища зберігаються в одному місці (залежить від ОС) і **Pipenv** автоматично створює нове середовище або використовує створене, залежно від папки, де ви працюєте.

Коли потрібно явно вказати, що ви працюєте в проекті, і потрібно створити для проекту середовище з конкретною версією Python, можна виконати команду:

pipenv --python 3.7

Цією командою ми вказуємо, що у цьому проекті потрібно створити віртуальне середовище з Python версією 3.7.

Під капотом **Pipenv** використовує **venv** для створення віртуальних середовищ. Тому ви можете використовувати створене віртуальне середовище і без **pipenv**, якщо потрібно.

Якщо потрібно видалити середовище, можна скористатися командою:

pipenv --rm

**Встановлення нових пакетів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/pipenv#%D0%B2%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BD%D0%BE%D0%B2%D0%B8%D1%85-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%96%D0%B2)Для встановлення пакетів, використовуючи **pipenv**, можна виконати команду:

pipenv install Flask

Ця команда встановить найновішу версію **Flask**. Якщо потрібно встановити якусь конкретну версію:

pipenv install Flask==0.6.1

Ця команда встановить **Flask** версії 0.6.1.

Зверніть увагу, що під час роботи **pipenv** створює два файли: *Pipfile* та *Pipfile.lock*.

*Pipfile* містить інформацію про те, звідки та яку версію пакета потрібно встановити. *Pipfile.lock* ж зберігає інформацію про всі (включаючи залежність) встановлені пакети в середовищі.

Для встановлення пакетів **pipenv** зчитує хеш пакета та записує його у спеціальний файл *Pipfile.lock* разом із версією пакета. Це робиться для того, щоб бути на 100% впевненим, що версія пакета, яку ви використовуєте, точно відповідає, якщо зміниться хоча б один біт, це не залишиться непоміченим.

Через це встановлення пакетів займає більше часу, ніж просто використання **pip**. Крім того, **pipenv** автоматично перевіряє всі залежності пакетів, що встановлюються, і намагається автоматично встановити найновішу сумісну версію всіх пакетів. Це дає вам можливість розробляти програму, використовуючи найновіші пакети без додаткових зусиль.

Буває, що для розробки вам потрібно встановити пакет, який не входить у фінальне постачання, але потрібний для розробки, наприклад *iPython*.

Спеціально для такої ситуації **pipenv** дає можливість встановлювати пакети в середовищі, вказавши, що вони не повинні входити до фінального постачання:

pipenv install ipython --dev

Ця команда встановить пакет *iPython*, але відмітить його та всі його залежності як необов'язкові, і їх можна буде не встановлювати на цільовому пристрої.

**Розгортання віртуального середовища**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/pipenv#%D1%80%D0%BE%D0%B7%D0%B3%D0%BE%D1%80%D1%82%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B2%D1%96%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE-%D1%81%D0%B5%D1%80%D0%B5%D0%B4%D0%BE%D0%B2%D0%B8%D1%89%D0%B0)Коли ви вже маєте *Pipfile* та *Pipfile.lock* і вам потрібно встановити всі пакети у нове (чисте) віртуальне середовище, ви можете виконати:

* pipenv install — для встановлення найновіших версій пакетів з *Pipfile* в середовище з можливим оновленням *Pipfile.lock* (якщо вийшли нові версії пакетів, ніж зазначені в *Pipfile.lock*).
* pipenv install --dev — те саме, але встановляться також пакети, відмічені як --dev, для розробки.
* pipenv sync встановить суворо вказані в *Pipfile.lock* версії пакетів і не перевірятиме наявність новіших версій.
* pipenv install --deploy встановить версії точно такі, як зазначено в *Pipfile.lock* і, якщо хеші пакетів не збігаються з хешами з *Pipfile.lock*, або версія Python не збігається, видасть помилку та припинить встановлення.
* pipenv install --system — те саме, що й pipenv install, але встановить пакети глобально у систему, а не у віртуальне середовище.

**Видалення та оновлення пакетів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/pipenv#%D0%B2%D0%B8%D0%B4%D0%B0%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%B0-%D0%BE%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%96%D0%B2)

Для видалення пакета з середовища та всіх його залежностей ви можете виконати pipenv uninstall package\_name. **pipenv** вміє стежити за залежностями та, на відміну від **pip**, може виконати видалення, не залишаючи за собою "сміття" пакетів, які встановлювалися як залежності, але більше не потрібні.

Для оновлення всіх пакетів на найсвіжіші версії ви можете виконати pipenv update.

Під час встановлення або оновлення пакетів можуть виникати ті самі конфлікти версій різних пакетів залежностей. Наприклад, пакет **A** вимагає встановити**botocore** версії не вище 0.6.1, водночас **boto3** вимагає, щоб**botocore** був не старшим 1.0.0. Це проблема, яку стандартними засобами Python не вирішити.

**pipenv** може вивести вам список всіх пакетів та їх залежностей, що, звичайно, допоможе усунути конфлікти залежностей. Для цього потрібно виконати pipenv graph.

Окремо варто згадати, що якщо ви встановлювали пакети в середовище **pipenv** за допомогою**pip** і потрібно видалити все зайве, ви можете виконати pipenv clean і все, що не вказано у *Pipfile.lock*, буде видалено.

**Менеджер Poetry**

[**Poetry**](https://python-poetry.org/) - це інструмент для управління залежностями у Python проектах (аналог вбудованого pip). Він схожий за функціоналом на розглянутий раніше pipenv.

**Poetry** дозволяє ефективно управляти залежностями та пакетами в Python. Він виконує ту саму роль, що й setup.py або pipenv, але має більшу гнучкість і функціональність. Ви можете оголосити бібліотеки, від яких залежить ваш проект, у файлі pyproject.toml. Після цього poetry встановлюватиме або оновлюватиме їх за вашою вимогою. Крім того, цей інструмент дозволяє інкапсулювати робочий проект в ізольоване середовище. Нарешті, ви можете використати poetry для прямої публікації вашого пакету на Pypi.

Використання файлів pypoproject.toml та poetry.lock робить його схожим на Node Package Manager (npm) для Node.js.

**Встановлення**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/poetry#%D0%B2%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F)Встановити poetry можна через [Windows (Powershell)](https://python-poetry.org/docs/#installing-with-the-official-installer)

(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content | py -

Poetry можна встановити вручну за допомогою pip та модуля venv. Діючи таким чином, ви, по суті, виконаєте кроки, які виконує офіційний встановлювач.

python3 -m venv $VENV\_PATH

$VENV\_PATH/bin/pip install -U pip setuptools

$VENV\_PATH/bin/pip install poetry

Poetry буде доступний за адресою $VENV\_PATH/bin/poetry і може бути викликаний напряму або за символічним посиланням в іншому місці. Щоб видалити Poetry, просто видаліть весь каталог $VENV\_PATH.

**Створення нового проекту**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/poetry#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BD%D0%BE%D0%B2%D0%BE%D0%B3%D0%BE-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D1%83)Після встановлення ми можемо створювати "poetry-проекти" за допомогою команди poetry new <ім'я проекту>.

Припустимо, що ми ввели команду poetry new solution, тоді ми отримаємо наступну структуру каталогів:

├── solution

│ └── \_\_init\_\_.py

├── pyproject.toml

├── README.rst

└── tests

├── \_\_init\_\_.py

└── test\_solution.py

Ми бачимо, що у проекті solution був створений однойменний пакет - директорія solution з відповідним \_\_init\_\_.py. Будь-який poetry-проект завжди містить хоча б один пакет.

Окрім пакету solution, у проекті вже є пакет tests з першим тестом і ми повинні розуміти, що справжні проекти завжди мають тести.

У файлі README.rst повинен бути опис проекту. Це файл формату [reStructuredText](https://ru.wikipedia.org/wiki/ReStructuredText).

Як бачимо Poetry автоматично створив усі необхідні файли для майбутнього пакета, але найбільший інтерес представляє файл під назвою pyproject.toml, цей файл - просунута альтернатива requirements.txt.

Вміст файлу pyproject.toml буде наступним:

[tool.poetry]

name = "solution"

version = "0.1.0"

description = ""

authors = ["FirstName LastName <youremail@gmail.com>"]

[tool.poetry.dependencies]

python = "^3.9"

[tool.poetry.dev-dependencies]

pytest = "^5.2"

[build-system]

requires = ["poetry-core>=1.0.0"]

build-backend = "poetry.core.masonry.api"

Формат файлу [TOML](https://github.com/toml-lang/toml). Розділ tool.poetry призначений для опису проекту: назва, версія, коротка інформація про проект тощо. Далі слідує tool.poetry.dependencies, саме тут вказані всі production залежності. Розділ tool.poetry.dev-dependencies призначений для залежностей, які використовуються під час розробки, наприклад pytest для тестів.

**Ініціалізація вже існуючого проекту**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/poetry#%D1%96%D0%BD%D1%96%D1%86%D1%96%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F-%D0%B2%D0%B6%D0%B5-%D1%96%D1%81%D0%BD%D1%83%D1%8E%D1%87%D0%BE%D0%B3%D0%BE-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D1%83)Щоб ініціалізувати poetry у вже готовому проекті, потрібно виконати команду:

poetry init

Після цього буде запущено процес опитування щось на зразок зазначеного нижче:

This command will guide you through creating your pyproject.toml config.

Package name [test\_poetry]:

Version [0.1.0]:

Description []:

Author [FirstName LastName <youremail@gmail.com>, n to skip]:

License []:

Would you like to define your main dependencies interactively? (yes/no) [yes]

You can specify a package in the following forms:

- A single name (requests)

- A name and a constraint (requests@^2.23.0)

- A git url (git+https://github.com/python-poetry/poetry.git)

- A git url with a revision (git+https://github.com/python-poetry/poetry.git#develop)

- A file path (../my-package/my-package.whl)

- A directory (../my-package/)

- A url (https://example.com/packages/my-package-0.1.0.tar.gz)

Search for package to add (or leave blank to continue):

Would you like to define your development dependencies interactively? (yes/no) [yes]

Search for package to add (or leave blank to continue):

Generated file

[tool.poetry]

name = "test\_poetry"

version = "0.1.0"

description = ""

authors = ["FirstName LastName <youremail@gmail.com>"]

[tool.poetry.dependencies]

python = "3.10"

[tool.poetry.dev-dependencies]

[build-system]

requires = ["poetry-core>=1.0.0"]

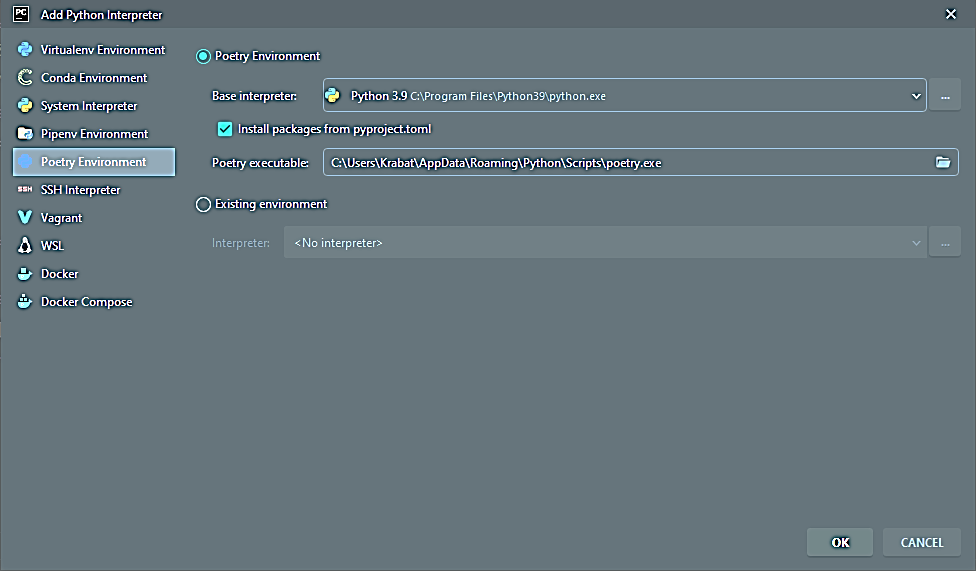
build-backend = "poetry.core.masonry.api"

Do you confirm generation? (yes/no) [yes]

Для активації віртуального середовища необхідно виконати команду:

poetry shell

Але краще ініціалізувати проект через **PyCharm**, щоб у терміналі ми працювали у віртуальному середовищі автоматично.



**CAUTION**

Не забудьте вказати параметр Poetry executable, щоб **PyCharm** знав, де встановлено poetry

**Основні команди**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/poetry#%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%96-%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B8)Щоб додати до проекту залежність у вигляді aiosqlite. Необхідно виконати команду

poetry add aiosqlite

**Poetry**, при цьому, автоматично створить віртуальне середовище, якщо воно не було створено. Всі середовища знаходяться на шляху ~/.cache/pypoetry/virtualenvs

Після команди у файлі pyproject.toml повинен з'явитися запис у розділі залежностей tool.poetry.dependencies

Щоб додати залежність для розробки, достатньо вказати прапорець --dev:

poetry add pytest --dev

Цей пакет повинен потрапити до pyproject.toml у розділ tool.poetry.dev-dependencies]

Щоб видалити залежність, потрібно виконати команду remove:

poetry remove aiosqlite

Якщо потрібно підтягнути існуючий проект і встановити його залежність для локальної розробки, вводимо команду:

poetry install

Якщо під час запуску встановлення **poetry** вже є файл poetry.lock, а також файл pyproject.toml, це означає, що ви виконали команду встановлення раніше, або хтось ще у проекті виконав команду встановлення та зафіксував файл poetry.lock.

У будь-якому разі запуск встановлення за наявності файлу poetry.lock дозволяє та встановлює всі залежності, які ви вказали в pyproject.toml, але **poetry** використовує точні версії, перелічені в poetry.lock, щоб гарантувати, що версії пакетів не суперечать всім, хто працює над вашим проектом.

Оновити всі залежності до останніх версій:

poetry update

Щоб переглянути всі доступні пакети, можна використовувати команду show.

poetry show

Версії залежностей

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/poetry#%D0%B2%D0%B5%D1%80%D1%81%D1%96%D1%97-%D0%B7%D0%B0%D0%BB%D0%B5%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B5%D0%B9)При встановленні пакета можна вказати точну версію проекту, наприклад:

[tool.poetry.dependencies]

pygame = "2.1.0"

Але іноді є необхідність вказати діапазон версій пакета, щоб отримувати оновлення, у такому разі є кілька способів вказати діапазон за допомогою спеціальних символів ^~\*:

[tool.poetry.dependencies]

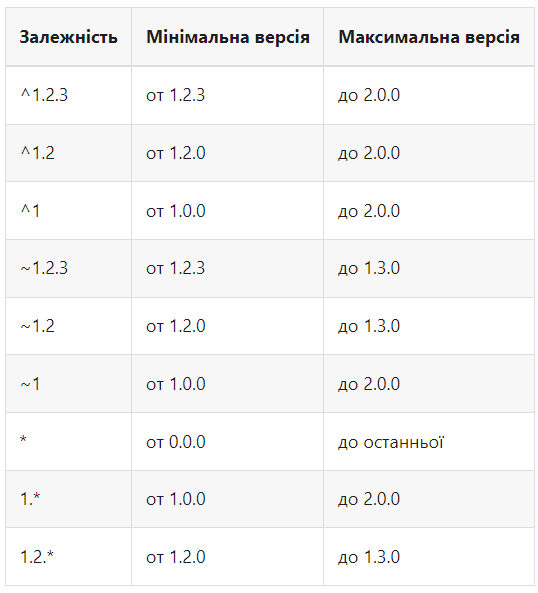
pygame = "^2.1"

pygame = "~2.1"

pygame = "2.1.\*"

pygame = "\*"

Ось які діапазони приймають ці префікси версій при встановленні або оновленні такої залежності:



Для детальнішого вивчення проекту відвідайте сторінку [Poetry](https://python-poetry.org/docs/).

**Відлагодження застосунку**

Термін *відлагодження* насамперед означає усунення помилок у коді. Робиться це по-різному: шляхом перевірки коду на наявність друкарських помилок, за допомогою аналізатора коду, за допомогою профілювальника продуктивності. Але, насамперед, відлагодження здійснюється за допомогою відлагоджувача.

Відлагоджувач - це вузькоспеціалізований засіб розробки, який приєднується до працюючого застосунку та дозволяє перевіряти код.

Потужним інструментом Python є **debugger**, який дає можливість виконувати застосунок порядково та зупиняти виконання в будь-який момент часу, щоб перевірити стан пам'яті в цей момент, значення змінних, доступність модулів тощо. Всі популярні IDE мають вбудовані інструменти для зупинення виконання застосунку на вказаному рядку коду та/або порядкового виконання коду.

Ці інструменти використовують [**pdb**](https://docs.python.org/3.7/library/pdb.html) — пакет для повноцінного відлагодження застосунку.

Пакет підтримує точки зупинки виконання застосунку та покрокове виконання скрипту.

Точки зупинки можна задавати у самому модулі. Запускати дебагер можна з консолі:

python3 -m pdb myscript.py

Або з інтерпретатора Python:

import pdb

import mymodule

pdb.run('mymodule.test()')

Ви, звичайно, можете скористатися pdb напряму, але рекомендується вивчити документацію до своєї IDE і використовувати зручний дебагер саме у ній.

Розділи документацій популярних IDE:

* [Visual Studio Code](https://code.visualstudio.com/docs/python/debugging)
* [PyCharm](https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html)
* [PyCharm частина два](https://www.jetbrains.com/help/pycharm/part-1-debugging-python-code.html)

**Логування застосунку**

Незважаючи на всі тести, дебаг та перевірки коду на правильність роботи, помилки все одно трапляються. І трапляються помилки саме у продакшені при взаємодії із реальними користувачами. Користувачі будуть використовувати ваш застосунок, найчастіше не так, як ви очікуєте. І навіть не так, як ви можете собі уявити.

Щоб виправити такі помилки, потрібно знати, що саме відбувалося в застосунку і які дії призвели до виникнення помилок. Для цього потрібно записувати, зберігати стан застосунку в деякий журнал. Такий журнал для простоти називають log та механізм журналювання подій — логуванням (logging).

Python має дуже потужний пакет логування, який так і називається —[**logging**](https://docs.python.org/3/library/logging.html)**.** Цей пакет дає можливість налаштувати журнал для застосунку в пару рядків або налаштувати складні багаторівневі механізми логування. Пакет logging — це дуже гнучкий та потужний інструмент, і знати про його можливості просто обов'язково для будь-якого Python-розробника.

Приклад налаштування логування в консоль:

import logging

# print a log message to the console.

logging.warning('This is a warning!')

В результаті в консолі:

WARNING:root:This is a warning!

Трохи змінимо інформацію, що логується:

import logging

logging.basicConfig(

format='%(asctime)s %(message)s',

level=logging.DEBUG,

handlers=[

logging.FileHandler("program.log"),

logging.StreamHandler()

])

logging.warning('An example message.')

logging.warning('Another message')

Ми визначили два обробника (handlers). Один виводить у консоль logging.StreamHandler(), а другий logging.FileHandler("program.log") у файл program.log. Після запуску ви побачите:

WARNING:root:An example message.

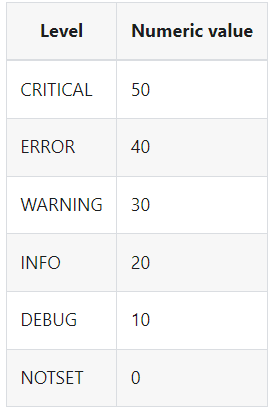
WARNING:root:Another message

На відміну від функції print, логер вміє:

* логувати із зазначенням модуля, функції, у якій сталася подія.
* логувати одночасно в кілька місць (файл, консоль, віддалений хост тощо).
* логувати з кількох потоків у потокобезпечному режимі (повідомлення будуть цілісними і не змішуються, як це може бути при використанні print).
* змінювати рівні логування (робити логування більш/менш детальним), змінюючи лише один параметр для проекту.
* логувати у файли в циклічному режимі (перезаписуючи найстаріший або після досягнення певного розміру).

Ключові концепції logging включають:

* рівень логування — число від 0 до 50, яке вказує, наскільки важливим є цей запис;



* formatter — об'єкт, який відповідає за те, яким чином форматуватиметься повідомлення, які там будуть поля та як буде виглядати кожен запис;
* handler — об'єкт, який відповідає за обробку кожного повідомлення; handler містить форматтер, рівень логування та інструкцію, що робити з повідомленням.

handler буде обробляти повідомлення, тільки якщо його рівень дорівнює або вище мінімального рівня, вказаного для цього handler. Таким чином, ми можемо, наприклад, писати взагалі всі повідомлення логування в консоль і лише повідомлення вище ERROR — у файл. Для цього нам достатньо буде визначити два handler: один для консолі з рівнем DEBUG і ще один для логування у файл з рівнем ERROR.

import logging

# створюємо логер, даємо йому ім'я та встановлюємо рівень logging.DEBUG

logger = logging.getLogger('simple\_example')

logger.setLevel(logging.DEBUG)

# створюємо handler для виведення в консоль та встановлюємо рівень DEBUG

ch = logging.StreamHandler()

ch.setLevel(logging.DEBUG)

# створюємо форматтер: час виведення (asctime), ім'я модуля (name), рівень (levelname) та саме повідомлення (message)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# додаємо зазначений форматтер до handler ch

ch.setFormatter(formatter)

# додаємо handler ch до логера

logger.addHandler(ch)

# Створюємо файловий handler для логера:

fh = logging.FileHandler("app.log")

fh.setLevel(logging.ERROR)

fh.setFormatter(formatter)

# додаємо файловий handler fh до логера

logger.addHandler(fh)

# приклад виконання коду

logger.debug('debug message')

logger.info('info message')

logger.warning('warn message')

logger.error('error message')

logger.critical('critical message')

Запустивши такий скрипт, ви у консолі побачите всі повідомлення на всіх рівнях, а у файлі app.log — тільки ERROR та CRITICAL.

Виведення в консоль:

2022-10-02 23:35:03,599 - simple\_example - DEBUG - debug message

2022-10-02 23:35:03,599 - simple\_example - INFO - info message

2022-10-02 23:35:03,599 - simple\_example - WARNING - warn message

2022-10-02 23:35:03,599 - simple\_example - ERROR - error message

2022-10-02 23:35:03,599 - simple\_example - CRITICAL - critical message

Вміст файлу app.log:

2022-10-02 23:35:03,599 - simple\_example - ERROR - error message

2022-10-02 23:35:03,599 - simple\_example - CRITICAL - critical message

**Автоматичне форматування коду**

Форматування коду відповідно до вимог стандарту PEP8 та інших стандартів, прийнятих у спільноті Python, є несуворою вимогою. Однак, у більшості колективів, де розробляються застосунки на Python, рано чи пізно, але доводиться приймати рішення про дотримання деяких стандартів форматування.

До появи інструментів автоматизації форматування коду це завдання покладалося на розробників, і дотримуватися вимог повинні були самі програмісти. Зараз у більшості сучасних мов програмування є розроблені стандарти форматування коду та автоматизації форматування.

**The Black Code Style**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/format#the-black-code-style)

Не став винятком і Python. Python має пакет, який фактично став золотим стандартом — це [**black**](https://black.readthedocs.io/en/stable/the_black_code_style/index.html)**.**

**black** суворо дотримується PEP8, але вносить низку своїх власних правил, які покращують сприйняття коду. За потреби, можна налаштувати black "під себе", але в 99% випадків стандартних налаштувань цілком достатньо.

Щоб встановити black, достатньо виконати pip install black. Коли вам потрібно відформатувати якийсь файл або цілий пакет, ви можете виконати в консолі black my\_python\_script.py або black my\_package. За цими командами black відформатує скрипт my\_python\_script.py або пакет my\_package.

Більшість IDE дають можливість налаштувати автоматичний запуск black для файлу під час його збереження, щоб код завжди був відформатований коректно.

Для **PyCharm IDE** це плагін [BlackConnect](https://plugins.jetbrains.com/plugin/14321-blackconnect). Він вимагає встановлення та запуску blackd

pip install black[d]

І запустіть його командою в режимі демона blackd.

Для **VSCode** використовуйте плагін [Black Formatter](https://marketplace.visualstudio.com/items?itemName=ms-python.black-formatter)

**Статичні аналізатори коду**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/format#%D1%81%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D1%96-%D0%B0%D0%BD%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%82%D0%BE%D1%80%D0%B8-%D0%BA%D0%BE%D0%B4%D1%83)Дуже корисними бувають інструменти, що дозволяють знаходити друкарські помилки в коді або "безглузді" помилки описки, допущені здебільшого просто через неуважність. Ці інструменти дозволяють автоматично виявити "проблеми" у коді. До проблем можна віднести:

* неправильна назва змінних або аргументів;
* неправильний тип аргументів, переданий під час виклику функції або методу;
* дуже велика кількість вкладених блоків коду (цикли, умови);
* порушення стандартів PEP;
* пропущені або зайві імпорти;
* використання "приватних" атрибутів.

Динамічна природа мови Python робить статичні аналізатори неефективними і в деяких випадках розробник краще знає, що правильно у конкретній ситуації. Але будь-який з популярних у цій галузі інструментів з великою ймовірністю захистить вас від безглуздих помилок через неуважність.

Статичні аналізатори, що найчастіше використовуються:

* [Pylint](https://pypi.org/project/pylint/) — дуже суворий аналізатор коду, часто спрацьовує хибно, робить найглибшу перевірку коду.
* [Flake8](http://flake8.pycqa.org/en/latest/) — аналог Pylint, але працює набагато швидше та набагато рідше спрацьовує хибно, хоч і пропускає деякі помилки.
* [MyPy](https://mypy.readthedocs.io/en/stable/) — аналізує правильність типів аргументів функцій та методів, добре працює з відносно новими в Python визначеннями типів (**typing**), попереджає, коли ви можете потенційно передати неправильний тип аргументу, що може викликати помилку (наприклад, передати **None**, коли функція чекає лише рядок).
* [Radon](http://radon.readthedocs.org/en/latest/index.html) аналізує код на цикломатичну складність, попереджає, коли безліч вкладених циклів та розгалужень коду роблять код важким для сприйняття.

Ви можете не використовувати всі ці інструменти, або використовувати якісь ще, бо це не вбереже вас на 100% від помилок, але суттєво зменшить кількість помилок, допущених через неуважність.

**Типізація в Python**

Мови програмування можуть бути статично або динамічно типізовані. Статично типізована мова виконує перевірку типів під час компіляції, тоді як динамічно типізована мова виконує її під час виконання.

Також мови можуть бути слабо і сильно типізованими. Строго типізована мова має суворіші правила, такі як присвоєння змінних, значення, що повертаються, і виклик функцій, у той час як слабо типізовані можуть давати непередбачувані результати. Python є динамічною мовою зі строгою типізацією.

З версії Python 3.5 підказки типів офіційно стали частиною мови (PEP 484). Використовуючи інструмент linter або перевірки коду, розробники можуть перевіряти узгодженість змінних та їх типів у кодовій базі та виконувати статичний аналіз коду, який раніше був би важким або неможливим. Все це робиться заздалегідь, до запуску коду.

Python не використовує підказки типів під час виконання програми. У момент запуску програми всю інформацію про тип буде вилучено. Підказки типів Python використовує заздалегідь, у системі перевірки типів, які ви використовуєте, наприклад, у редакторі VSCode або PyCharm IDE. Іншими словами, підказки типів у Python призначені для розробника, а не для середовища виконання.

**Анотації типів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/typing#%D0%B0%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D1%96%D1%97-%D1%82%D0%B8%D0%BF%D1%96%D0%B2)Ми можемо анотувати функцію, щоб вказати її тип, що повертається, і типи її параметрів.

def my\_mul(data: list) -> float:

result = 1

for num in data:

result = result \* num

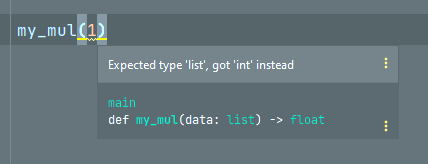
return result

Це інформує засіб перевірки типів, що у нас є функція mul, яка прийме data як аргумент і поверне float

Викликаємо нашу функцію з неправильним параметром data

my\_mul(1)

Наприклад, у PyCharm IDE це призведе до попередження, що типи параметра, який передається і який очікується не збігаються Expected type 'list', got 'int' instead



Це спрацьовує вбудована перевірка типів всередині PyCharm IDE.

А виклик функції зі списком буде коректним, тому що повністю збігається з описом.

my\_mul([1, 2, 3])

Одним із найбільш широко використовуваних засобів для перевірки типів є пакет mypy. Якщо ви не використовуєте, наприклад, VSCode або PyCharm, або вам потрібно перевірити типізацію в консолі.

Пакет може бути встановлений наступним чином:

pip install mypy

Після цього можна запустити будь-який файл Python, щоб перевірити, чи збігаються типи.

mypy main.py

Після відлагодження помилок можна запускати програму у звичайному режимі.

**Псевдоніми типів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/typing#%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%BD%D1%96%D0%BC%D0%B8-%D1%82%D0%B8%D0%BF%D1%96%D0%B2)Оскільки такі об'єкти, як списки, словники та кортежі містять інші об'єкти, іноді нам потрібно ввести підказки, щоб зазначити, які типи об'єктів вони містять. Для цього нам потрібно звернутись до модуля [typing](https://docs.python.org/3/library/typing.html), який надає інструменти для опису типів.

from typing import List

Data = List[float | int]

def my\_mul(data: Data) -> float:

result = 1

for num in data:

result = result \* num

return result

my\_mul([1, 2, 3])

У наведеному вище фрагменті знаходиться псевдонім Data, що розшифровується як список значень цілих значень int або з плаваючою комою float.

Можна описати типи словника, надавши їх у вигляді списку:

from typing import Dict

dict\_of\_users: Dict[int, str] = {

1: "Jane",

2: "Jon"

}

**Союзи**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/typing#%D1%81%D0%BE%D1%8E%D0%B7%D0%B8)Ми також можемо використовувати типи союзів Union. Наприклад, функція може повертати лише один з типів integer або float:

from typing import Union

Number = Union[float, int]

def add(x: Number, y: Number) -> Number:

return x + y

**Generics**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/typing#generics)Дженерики або універсальні типи. Уявімо ситуацію, що у нас є функція, яка додає два параметри і повертає результат – суму цих параметрів. Ці параметри можуть бути рядками, цілими або дійсними числами. Щоб уникнути повторення коду і не писати для кожного випадку свою функцію зі своїми типами, ми використовуємо дженерики.

from typing import TypeVar

T = TypeVar("T", int, str, float)

def calculator(x: T, y: T) -> T:

return x + y

print(calculator(3, 5))

print(calculator("Hello", "World"))

print(calculator(3.5, 1.4))

Виведення:

8

HelloWorld

4.9

**Тип Any**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/typing#%D1%82%D0%B8%D0%BF-any)Це спеціальний тип, який інформує статичну перевірку типів таким чином, що кожен тип сумісний з цим ключовим словом.

Розглянемо нашу стару функцію calculator, що тепер приймає аргументи будь-якого типу.

from typing import TypeVar, Any

T = TypeVar("T", int, str, float)

def calculator(x: Any, y: Any) -> T:

return x + y

print(calculator(3, 5))

print(calculator("Hello", "World"))

print(calculator(3.5, 1.4))

Це не дуже "хороший" тип і використовувати його потрібно завжди усвідомлено

**Висновок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-01/typing#%D0%B2%D0%B8%D1%81%D0%BD%D0%BE%D0%B2%D0%BE%D0%BA)У цьому розділі ми дізналися про модуль типізації Python, який дуже корисний у контексті перевірки типів, дозволяючи зовнішнім перевіркам типів точно повідомляти про будь-які помилки. Для детальнішої інформації зверніться до [документації](https://peps.python.org/pep-0484/#type-aliases)