

Python для мережевих інженерів

Книга, курс та завдання з основ Python

Nataliya Samoylenko

Copyright © 2015 - 2023 Nataliya Samoylenko

Table of contents

	4
Python для мережевих інженерів	4
Прогрес	4
Про проект	4
FAQ	6
Гарячі клавіші	7
глобальний режим	7
вікно пошуку	7
Книга	8
Вступ	8
Ресурси для навчання	8
Для кого ця книга	8
Навіщо вчитися програмувати?	8
Необхідні версії ОС та Python	9
Підготовка	10
1. Підготовка до роботи	10
2. Робота із завданнями	22
Основи Python	46
3. Початок роботи з Python	46
4. Типи даних у Python	62
5. Створення базових скриптів	108
6. Контроль перебігу програми	114
7. Робота з файлами	142
Завдання	164
Завдання	164
Утиліта rупeng	166
Де робити завдання	166
Скрипт rупeng	166
Перевірка завдань тестами	166
Отримання відповідей	167
Завантажити всі зміни в поточному каталозі на github, без прив'язки до того чи проходять тести	167
Оновлення розділів	167
Оновлення завдань та тестів	168
Таблиці з темами завдань	171
04_data_structures	171

05_basic_scripts	171
06_control_structures	172
07_files	172
09_functions	172
11_modules	173
12_useful_modules	173
15_module_re	173
17_serialization	173
18_ssh_telnet	174
19_concurrent_connections	174
20_jinja2	174
21_textfsm	174
22_oop_basics	175
23_oop_special_methods	175
24_oop_inheritance	175
25_db	175
Утиліта runenguk-quiz	177
Встановити модуль	178
Відео	178
Підготовка вмі/хоста	179
Підготовка віртуальної машини/хоста Linux	179
Підготовка віртуальної машини/хоста Windows	181
Нюанси виконання завдань на Windows	182
Редактори	183
Вибір редактора	183
Підготовка редактора	184
Редактор Thonny	186
Редактор vim	187

Python для мережевих інженерів

Прогрес

- ✓ [Завдання](#) - переклад готовий
- ✓ [Заняття \(утиліта `runenguk-quiz`\)](#) - переклад готовий
- 🔄 [Книга](#) - переклад в процесі, готові 7 розділів із 25
- 🔄 [Курс](#) - в процесі створення, готові 5 тем із 25
- 🔄 [Довідник Python](#) - в процесі створення

Про проект

Python для мережевих інженерів це [книга](#), [курс](#) та [завдання](#) з основ Python. Всі приклади та завдання по можливості побудовані навколо мережевого обладнання та роботи з ним.

Завдання однакові для книги та курсу. [Детальніше про завдання](#).

Книга та курс мають однакову нумерацію тем. Тобто ви можете відкрити розділ книги і прочитати його, а якщо хочете подивитися відео на цю ж тему, то відкрити тему з таким же номером в курсі. Наприклад, тема "Початок роботи з Python":

- розділ 3 в книзі [3. Початок роботи з Python](#)
- 3 тема в курсі [03: Початок роботи з Python](#)

[Книга](#) призначена для початківців і багато речей в книзі спрощено та скорочено.

[Довідник Python](#) з'явився як доповнення до книги. В ньому деякі теми будуть розписані більш детально.

Книга Python для мережевих інженерів

Переклад книги з російської відбуватиметься згідно з [розкладом курсу](#).

Разом із перекладом книжку буде оновлено до останньої версії Python (3.11) та додано інформацію, малюнки тощо.

Note

Хоча основи Python не змінилися між версіями Python 3.7-3.11, деякі помилки, текст та вивід змінилися.

Інші переклади книги:

- [Python для сетевых инженеров](#)
- [Python for Network Engineers](#)

На Read the Docs український переклад видалено. Як тільки буде закінчений переклад розділу книги, він буде доданий на цей сайт.

🕒 July 12, 2023

🕒 May 15, 2023

FAQ

Що вибирати книгу чи курс (відео)

Зазвичай у курсі міститься трохи більше інформації, ніж у книзі. Найкращий варіант навчання – поєднання книги та курсу.

Наприклад, процес можна організувати так: спочатку ви дивитесь відео і починаєте виконувати завдання по темі. Коли із завданнями щось не виходить, можна спробувати перечитати потрібну тему в книзі.

У будь-якому випадку, ви завжди можете поставити мені питання в [чати slack](#).

Чому все на одному сайті

Об'єднання книги та курсу на одному сайті дозволяє користуватися пошуком та знаходити матеріали і у форматі відео та у форматі книги. Через це [довідник Python](#) також створюється на цьому ж сайті.

🕒 May 29, 2023

🕒 May 29, 2023

Гарячі клавіші

 Гарячі клавіші в документації Material for MkDocs

глобальний режим

Цей режим активний, коли пошук не сфокусований і немає іншого фокусованого елемента, сприйнятливого до введення з клавіатури. Прив'язані такі ключі:

- **F** або **S** або **/** відкрити вікно пошуку
- **P** або **,** перейти до попередньої сторінки
- **N** або **.** перейти до наступної сторінки

вікно пошуку

Цей режим активний, коли пошук сфокусований:

- **↓ Down** , **↑ Up** вибір наступного/попереднього результату
- **⌫ Esc** , **Tab →** закрити вікно пошуку
- **Enter ↵** перейти за вибраним результатом

 July 12, 2023

 July 12, 2023

Книга

Вступ

Переклад книги йтиме приблизно за [розкладом курсу](#).

Інші переклади книги:

- [Python for Network Engineers](#)
- [Python для сетевых инженеров](#)

З одного боку, книга досить базова, щоб її міг освоїти будь-який бажаючий, а з іншого боку, у книзі розглядаються всі основні теми, які дозволять зростати самостійно. Книга не ставить за мету глибокого розгляду Python. Завдання книги – пояснити зрозумілою мовою основи Python та дати розуміння необхідних інструментів для його практичного використання. Все, що розглядається в книзі, орієнтоване на мережеве обладнання та роботу з ним. Це дозволяє відразу використовувати у роботі мережевого інженера те, що було вивчено. Всі приклади показуються на прикладі обладнання Cisco, але, звичайно ж, вони застосовні для будь-якого іншого обладнання.

Ресурси для навчання

- [все про завдання](#)
- [задати питання можна в slack](#)

Якщо вам більше подобається відео формат, чи є бажання комбінувати відео з книгою, [тут за розкладом](#) будуть з'являтися відео.

Для кого ця книга

Для мережевих інженерів з досвідом програмування та без. Всі приклади та завдання побудовані навколо роботи з мережевим обладнанням. Ця книга буде корисна мережевим інженерам, які хочуть автоматизувати завдання, з якими стикаються кожен день і хотіли зайнятися програмуванням, але не знали, з якого боку підійти.

Навіщо вчитися програмувати?

Знання програмування для мережевого інженера можна порівняти зі знанням англійської мови. Якщо ви знаєте англійську хоча б на рівні, який дозволяє читати технічну документацію, ви відразу розширюєте свої можливості:

- доступно у кілька разів більше літератури, форумів та блогів;
- практично для будь-якого питання або проблеми досить швидко знаходиться рішення, якщо ви ввели запит до Google.

Знання програмування у цьому дуже схоже. Якщо ви знаєте, наприклад Python хоча б на базовому рівні, ви вже відкриваєте масу нових можливостей для себе. Аналогія з англійською підходить ще й тому, що можна працювати мережевим інженером і бути добрим фахівцем без знання англійської. Англійська просто дає додаткові можливості, але вона не є обов'язковою вимогою.

Необхідні версії ОС та Python

Усі приклади та виведення терміналу у книзі відображаються на Debian Linux. Мінімально необхідна версія Python – 3.9, але краще використовувати 3.10 або 3.11.

Приклади

Усі приклади, що використовуються у книзі, знаходяться у [репозиторії](#). Приклади, які показані в розділах книги, є навчальними. Це означає, що вони не обов'язково показують найкращий варіант розв'язання задачі, тому що вони засновані лише на тій інформації, що розглядалась у попередніх розділах книги. Крім того, досить часто приклади, що давалися у розділах, розвиваються у завданнях. Тобто, в завданнях вам потрібно буде зробити більш універсальну, і, загалом, правильнішу версію коду. Якщо є можливість, краще набирати код, який використовується в книзі, самостійно, або, як мінімум, скопіювати приклади та спробувати щось у них змінити – так інформація краще запам'ятовуватиметься. Якщо такої можливості немає, наприклад коли ви читаєте книгу в дорозі, краще повторити приклади самостійно пізніше. У будь-якому випадку обов'язково потрібно виконувати завдання вручну.

Завдання

Всі завдання та допоміжні файли можна отримати в тому ж репозиторії, де є [приклад коду](#). Якщо завдання розділу мають завдання з літерами (наприклад, 5.2a), то потрібно виконати спочатку завдання без літер, а потім з літерами. Завдання з літерами, як правило, трохи складніші за завдання без літер і розвивають ідею у відповідному завданні без літери. Якщо виходить, краще виконувати завдання по порядку.

[Для всіх завдань є "відповіді", а точніше варіанти рішень](#). Для кожного завдання може бути багато правильних варіантів розв'язання. Звичайно, у відповіді краще підглядати поменше, але вони можуть допомогти вийти зі складної ситуації.

Якщо, наприклад, ви вирішили завдання умовно у 20 рядків, а у відповіді воно вирішено у 7 рядків, це не означає, що ви зробили неправильно. Будь-який робочий варіант рішення – правильний. Варіанти розв'язання можна читати після вирішення завдань. Це буде і практика читання коду, і ви зможете подивитися на інші підходи до вирішення задачі.

🕒 May 24, 2023

🕒 May 15, 2023

Підготовка

1. Підготовка до роботи

1. Підготовка до роботи

Для того, щоб почати працювати з Python, треба визначитися з декількома речами:

- яка операційна система використовуватиметься
- який редактор буде використовуватись
- яка версія Python буде використовуватись

У книзі використовується Debian Linux (в інших ОС вивід може відрізнятись) і Python 3.11.

Ще один важливий момент – вибір редактора. У наступному розділі наведено приклади редакторів для різних операційних систем. Замість редактора можна використовувати IDE. IDE це хороша річ, але на початку навчання може вийти так, що IDE буде відволікати вас безліччю можливостей. Список IDE для Python можна переглянути [тут](#).

🕒 May 23, 2023

🕒 May 15, 2023

Підготовка робочого оточення

Для виконання завдань книги можна використати кілька варіантів:

- налаштувати свою ОС
- підготувати віртуалку
- використовувати якийсь хмарний сервіс

ПІДГОТОВКА ВІРТУАЛЬНОЇ МАШИНИ/ХОСТА САМОСТІЙНО

Список модулів, які потрібно встановити:

```
pip install pytest pytest-clarity pyyaml tabulate jinja2 textfsm pexpect netmiko graphviz
```

Також необхідно встановити graphviz прийнятим способом в ОС (приклад для debian):

```
apt-get install graphviz
```

Хмарні сервіси

Ще один варіант – використовувати один із наступних сервісів:

- repl.it – цей сервіс надає онлайн інтерпретатор Python, а також графічний редактор.
[Приклад використання](#)
- [PythonAnywhere](https://pythonanywhere.com/) – виділяє окрему віртуалку, але у безкоштовному варіанті ви можете працювати тільки з командного рядка, тобто немає графічного текстового редактора

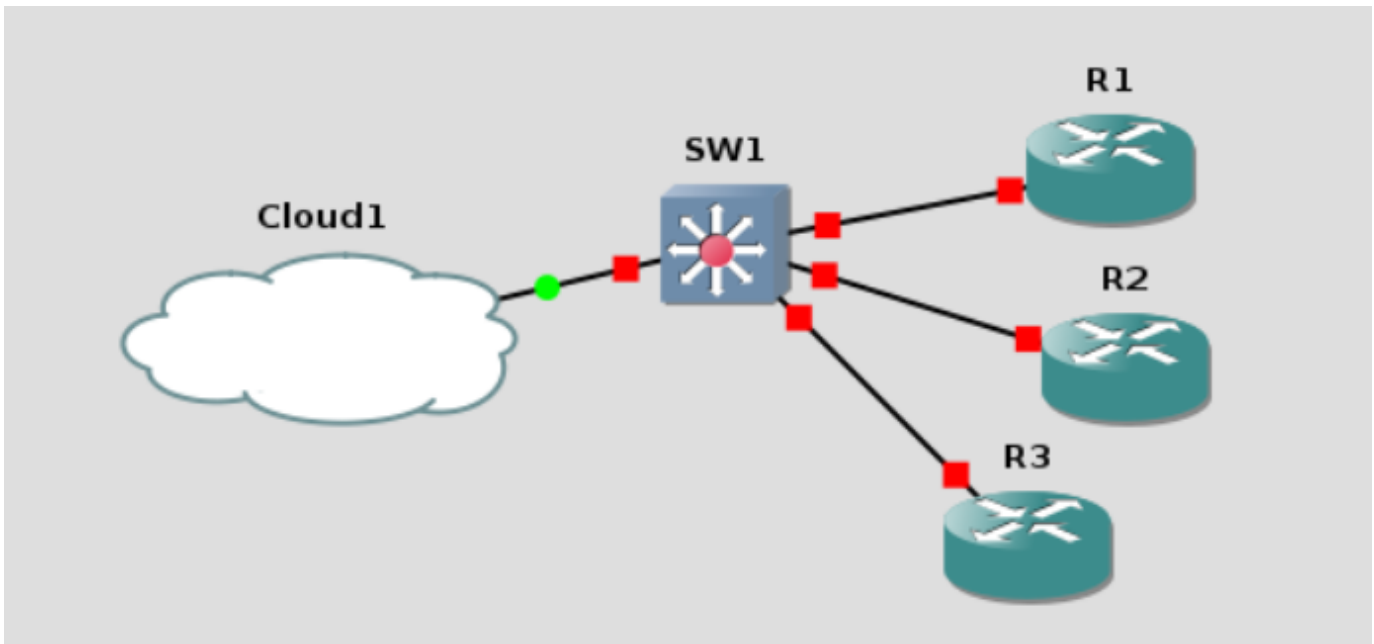
Мережеве обладнання

До 18 розділу книги потрібно підготувати віртуальне або реальне мережеве обладнання.

Всі приклади та завдання, в яких зустрічається мережеве обладнання, використовують однакову кількість пристроїв: три маршрутизатори з такими базовими налаштуваннями:

- користувач: cisco
- пароль: cisco
- пароль на режим enable: cisco
- SSH версії 2 (обов'язково саме версія 2), Telnet
- IP-адреси маршрутизаторів: 192.168.139.1, 192.168.139.2, 192.168.139.3
- IP-адреси повинні бути доступні з віртуалки, на якій ви виконуєте завдання і можуть бути призначені на фізичних/логічних/loopback інтерфейсах

Топологія може бути довільною. Приклад топології:



Базова конфігурація:

```
hostname R1
!
no ip domain lookup
ip domain name pyneng
!
crypto key generate rsa modulus 1024
ip ssh version 2
!
username cisco password cisco
enable secret cisco
!
line vty 0 4
  logging synchronous
  login local
  transport input telnet ssh
```

На якомусь інтерфейсі треба налаштувати IP-адресу

```
interface ...
  ip address 192.168.139.1 255.255.255.0
```

Аліаси (за бажанням)

```
!
alias configure sh do sh
alias exec ospf sh run | s ^router ospf
alias exec bri show ip int bri | exc unass
alias exec id show int desc
alias exec top sh proc cpu sorted | excl 0.00%__0.00%__0.00%
alias exec c conf t
alias exec diff sh archive config differences nvram:startup-config system:running-config
```

```
alias exec desc sh int desc | ex down  
alias exec bgp sh run | s ^router bgp
```

За бажання можна налаштувати EEM applet для відображення команд, які вводить користувач:

```
!  
event manager applet COMM_ACC  
  event cli pattern ".*" sync no skip no occurs 1  
  action 1 syslog msg "User $_cli_username entered $_cli_msg on device $_cli_host "  
!
```

🕒 May 24, 2023

🕒 May 15, 2023

ОС та редактор

Можна вибрати будь-яку ОС та будь-який редактор, але бажано використовувати Python версії ≥ 3.9 .

Всі приклади в книзі виконувались на Debian для Python 3.11, на інших ОС та для інших версій Python результат може трохи відрізнятись. Для виконання завдань з книги можна використовувати Linux, MacOS або Windows.

Для роботи з Python можна вибрати будь-який текстовий редактор або IDE, який підтримує Python. Як правило, для роботи з Python потрібно мінімум налаштування редактора і часто за замовчуванням розпізнає Python.

РЕДАКТОР THONNY

[Thonny](#) - хороший редактор для початківців:

- підтримує Python 3.10-3.11 і може встановити відразу себе і Python 3.10
- зручно зроблено роботу з різними версіями Python і віртуальними оточеннями, дуже явно можна вибирати версію і це не ховається в глибині налаштувань
- кілька варіантів відладчика
- відладчик nicer просто незамінний для початківців вивчати Python, показує покроково як обчислюється кожен вираз у Python
- відладчик faster працює в цілому як стандартний
- є всі стандартні плюшки з підказками, підсвічуванням і так далі (частину можливо треба буде включити в налаштуваннях)
- зручно підсвічує незакриті лапки/дужки
- підтримує Windows, Mac, Linux
- зручний інтерфейс і є можливість додавати/видаляти секції інтерфейсу за бажанням

Для знайомства з Thonny можна [переглянути відео](#). Там розглядаються основи та налагодження (debug) коду в Thonny.

IDE PYCHARM

[PyCharm](#) – інтегроване середовище розробки для Python. Для початківців може бути складним варіантом через безліч налаштувань, але це залежить від особистих уподобань. PyCharm підтримується безліч можливостей, навіть у безкоштовній версії.

PyCharm чудовий IDE, але, на мій погляд, він складний для початківців. Я не радила б використовувати його, якщо ви з ним не знайомі і тільки починаєте вчити Python. Ви завжди зможете перейти на нього після книги, але поки що краще спробувати щось інше.

Варіанти редакторів наведені для прикладу, замість них можна використовувати будь-який текстовий редактор, який підтримує Python.

🕒 May 24, 2023

🕒 May 15, 2023

Система керування пакетами `pip`

Для встановлення пакетів Python використовуватиметься `pip`. Це система керування пакетами, яка використовується для встановлення пакетів із Python Package Index (PyPI). Швидше за все, якщо у вас вже встановлено Python, то встановлено `pip`.

Перевірка версії `pip`:

```
$ pip --version
pip 23.1.2 from /home/nata/.venv/pyneng/lib/python3.11/site-packages/pip (python 3.11)
```

Якщо команда видала помилку, то `pip` не встановлений. Установка `pip` описана у [документації](#)

ВСТАНОВЛЕННЯ МОДУЛІВ

Для встановлення модулів використовується команда `pip install`:

```
$ pip install tabulate
```

Видалення пакета виконується таким чином:

```
$ pip uninstall tabulate
```

Крім того, іноді необхідно оновити пакет:

```
$ pip install --upgrade tabulate
```

або так:

```
$ pip install -U tabulate
```

PIP АБО PIP3

Залежно від того, як встановлений та налаштований Python у системі, може знадобитися використовувати `pip3` замість `pip`. Щоб перевірити, який варіант використовується, виконайте команду `pip --version`.

Варіант, коли `pip` відповідає Python 2.7:

```
$ pip --version
pip 9.0.1 from /usr/local/lib/python2.7/dist-packages (python 2.7)
```

На сучасних версіях ОС, найімовірніше, системний Python буде версії 3.x. Наприклад, для Debian bullseye:


```
$ pip --version  
pip 20.3.4 from /usr/lib/python3/dist-packages/pip (python 3.9)
```

Варіант, коли `pip3` відповідає Python 3.9:

```
$ pip3 --version  
pip 20.3.4 from /usr/lib/python3/dist-packages/pip (python 3.9)
```

Якщо в системі використовується `pip3`, то щоразу, коли в книзі встановлюється модуль Python, потрібно буде замінити `pip` на `pip3`.

Також можна використати альтернативний варіант виклику `pip`:

```
$ python3.11 -m pip install tabulate
```

Таким чином, завжди зрозуміло для якої версії Python встановлюється пакет.

 May 24, 2023

 May 15, 2023

Віртуальні оточення

Віртуальні оточення:

- дозволяють ізолювати різноманітні проекти один від одного
- пакети, які потрібні різним проектам, перебувають у різних місцях – якщо, наприклад, у одному проекті потрібен пакет версії 1.0, а іншому проекті потрібен той самий пакет, але версії 3.1, вони не заважатимуть один одному
- пакети, встановлені у віртуальних оточеннях, не перебивають глобальні пакети

ВБУДОВАНИЙ МОДУЛЬ VENV

Починаючи з версії 3.5, у Python рекомендується використовувати модуль `venv` для створення віртуальних оточень:

```
$ python3.11 -m venv ~/.venv/pyneng
```

Замість `python3.11` можна використовувати `python` або `python3`, залежно від того, як встановлено Python 3.11. Ця команда створює вказаний каталог і всі необхідні каталоги всередині нього, якщо вони створені.

Команда створює таку структуру каталогів:

```
$ ls -ls ~/.venv/pyneng
total 16
4 drwxr-xr-x 2 nata nata 4096 Aug 21 14:50 bin
4 drwxr-xr-x 2 nata nata 4096 Aug 21 14:50 include
4 drwxr-xr-x 3 nata nata 4096 Aug 21 14:50 lib
4 -rw-r--r-- 1 nata nata   75 Aug 21 14:50 pyvenv.cfg
```

Для переходу у віртуальне оточення треба виконати команду:

```
$ source ~/.venv/pyneng/bin/activate
```

Для виходу з віртуального оточення використовується команда `deactivate`:

```
$ deactivate
```

Докладніше про модуль `venv` у [документації](#).

Встановлення пакетів у віртуальному оточенні

Наприклад, встановимо у віртуальному оточенні пакет `simplejson`.

```
(pyneng)$ pip install simplejson
...
Successfully installed simplejson
Cleaning up...
```

Якщо перейти в інтерпретатор Python і імпортувати simplejson, він доступний і ніяких помилок немає:

```
(pyneng)$ python
>>> import simplejson
>>> simplejson
<module 'simplejson' from '/home/nata/.venv/pyneng/lib/python3.11/site-packages/simplejson/
__init__.py'>
>>>
```

Але якщо вийти з віртуального оточення і спробувати зробити те саме, то такого модуля немає:

```
(pyneng)$ deactivate

$ python
>>> import simplejson
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'simplejson'
>>>
```

🕒 May 24, 2023

🕒 May 15, 2023

Інтерпретатор Python

Перед початком роботи треба перевірити, що при виклику інтерпретатора Python вивід буде таким:

```
$ python
Python 3.11.1 (main, Dec 30 2022, 10:30:23) [GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Вивід показує, що використовується Python 3.11. Запрошення `>>>` це стандартне запрошення інтерпретатора Python. Виклик інтерпретатора виконується `python`, а щоб вийти, потрібно набрати `quit()`, або натиснути Ctrl+D.

У книзі замість стандартного інтерпретатора Python буде використовуватися `ipython`.

🕒 May 24, 2023

🕒 May 15, 2023

Додаткові матеріали

Документація:

- [Python Setup and Usage](#)
- [pip](#)
- [venv](#)
- [virtualenvwrapper](#)

Редактори и IDE:

- [PythonEditors](#)
- [IntegratedDevelopmentEnvironments](#)
- [VIM and Python - a Match Made in Heaven](#)

Thonny:

- [Сайт проекту Thonny](#)
- [Документація Thonny](#)
- [Thonny debug](#)

 May 23, 2023

 May 15, 2023

2. Робота із завданнями

2. Робота із завданнями

Всі деталі про те, які завдання є, для яких розділів, де виконувати завдання книги/курсу, про автоматичні тести для завдань та підказки на які теми завдання, знаходяться в [окремому розділі на сайті](#).

У книзі досить багато завдань і треба їх десь зберігати. Варіанти роботи з завданнями:

1. replit.com - робота онлайн, тільки до 18-го розділу, автозбереження змін
2. просто файли - локальна копія завдань зі своїм варіантом збереження прогресу/синхронізації (наприклад, Google Drive)
3. Git/GitHub - робота на одному або кількох комп'ютерах, синхронізація через GitHub та збереження змін за допомогою Git
4. Git/GitHub + replit.com – загалом такий самий як 2 варіант, але з налаштуванням роботи з Git/Github на replit.com

В цьому розділі розглядається Git/GitHub, подробиці по інших варіантах [у відео курсу](#)

🕒 May 23, 2023

🕒 May 15, 2023

Використання Git та GitHub

ВИКОРИСТАННЯ GIT ТА GITHUB

У книзі досить багато завдань і треба їх десь зберігати. Один із варіантів – використання для цього Git та GitHub.

Git – це розподілена система контролю версій, яка може:

- відстежувати зміни у файлах
- зберігати кілька версій одного файла
- скасовувати внесені зміни
- реєструвати, хто та коли зробив зміни

GitHub – це хостинг для проектів Git. Він є центром співпраці між мільйонами розробників та проектів. Багато проектів з відкритим кодом використовують GitHub задля Git хостингу, взаємодії з користувачами проекту, перегляду коду та для багато чого іншого. Отже хоч це і не частина проекту Git, ви майже напевно захочете чи вам доведеться колись взаємодіяти з GitHub під час професійного використання Git.

Звичайно, можна використовувати для цього інші засоби, але використовуючи Git та GitHub, можна поступово розібратися з ним і потім використовувати його для робочих завдань. Завдання книги/курсу знаходяться в окремому [репозиторії на GitHub](#). Їх можна завантажити як zip-архів, але краще працювати з репозиторієм за допомогою Git, тоді можна буде переглянути внесені зміни та легко оновити репозиторій. Якщо вивчати Git з нуля і особливо якщо це перша система контролю версій, з якою Ви працюєте, інформації може бути дуже багато, тому в цьому розділі все націлене на практичний бік:

- як почати використовувати Git та GitHub
- як виконати базові налаштування
- як подивитися інформацію та/або зміни

Теорії в цьому підрозділі буде мало, але будуть надані посилання на корисні ресурси. Спробуйте спочатку провести всі налаштування для виконання завдань, а потім продовжуйте читати книгу. І наприкінці, коли базова робота з Git та GitHub буде вже звичною справою, почитайте про них докладніше. Для чого може стати в нагоді Git:

- для зберігання конфігурацій та всіх змін у них
- для зберігання документації та її версій
- для зберігання схем та всіх їх версій
- для зберігання коду та його версій

GitHub дозволяє централізовано зберігати всі вище перелічені речі, але слід враховувати, що ці ресурси будуть доступні й іншим. GitHub має і приватні репозиторії, але навіть у них не варто викладати таку інформацію, як паролі.

🕒 May 23, 2023

🕒 May 15, 2023

ОСНОВИ GIT

Git – це розподілена система контролю версій (Version Control System, VCS), яка широко використовується та випущена під ліцензією GNU GPL v2. Вона може:

- відстежувати зміни у файлах
- зберігати кілька версій одного файла
- скасовувати внесені зміни
- реєструвати, хто та коли зробив зміни

Файл в Git може бути в такому стані:

- змінений (modified) - зміни у файлі ще не збережені у локальній базі даних
- індексований (staged) - файл позначений на додавання в наступний коміт
- збережений у коміті (committed) - файл збережено у локальній базі даних

Три основні частини проекту Git:

- робоча директорія - копія версії проекту
- індекс (staging area) - що буде збережено у наступному коміті
- директорія Git (.git) - тут система зберігає метадані та базу даних об'єктів вашого проекту

```
sequenceDiagram
    participant WD as Робочий каталог
    participant SA as Індекс
    participant G as Каталог .git
    G->>WD: git checkout
    WD->>SA: git add
    SA->>G: git commit
```

Установка Git

```
$ sudo apt-get install git
```

Первинне налаштування Git

Для початку роботи з Git, необхідно вказати ім'я та e-mail користувача, які будуть використовуватись для синхронізації локального репозиторію з репозиторієм на GitHub:

```
$ git config --global user.name "username"
$ git config --global user.email "username.user@example.com"
```

Подивитися налаштування Git можна таким чином:

```
$ git config --list
```

Ініціалізація репозиторію

Створення та перехід до каталогу first_repo

```
mkdir first_repo  
cd first_repo
```

Ініціалізація репозиторію виконується за допомогою команди `git init`:

```
[~/tools/first_repo]  
$ git init  
Initialized empty Git repository in /home/vagrant/tools/first_repo/.git/
```

Після виконання цієї команди у поточному каталозі створюється папка `.git`, в якій містяться службові файли, необхідні для Git.

🕒 May 24, 2023

🕒 May 15, 2023

ВІДОБРАЖЕННЯ СТАТУСУ РЕПОЗИТОРІЮ У ЗАПРОШЕННІ

Пропускаємо цю частину Windows. У Cmder статус показується за замовчуванням.

Це додатковий функціонал, який не є обов'язковим для роботи з Git, але дуже допомагає в цьому. При роботі з Git дуже зручно, коли можна відразу визначити, чи знаходитесь ви в звичайному каталозі або в репозиторії Git. Крім того, добре було б розуміти статус поточного репозиторію. Для цього потрібно встановити спеціальну [утиліту](#), яка показуватиме статус репозиторію. Для встановлення утиліти треба скопіювати її репозиторій у домашній каталог користувача, під яким ви працюєте:

```
cd ~
git clone https://github.com/magicmonty/bash-git-prompt.git .bash-git-prompt --depth=1
```

А потім додати до кінця файлу .bashrc такі рядки:

```
GIT_PROMPT_ONLY_IN_REPO=1
source ~/.bash-git-prompt/gitprompt.sh
```

Для того, щоб зміни застосовувалися, перезапустити bash у будь-який спосіб, наприклад:

```
exec bash
```

У моїй конфігурації запрошення командного рядка рознесене на кілька рядків, тому воно буде відрізнятися. Головне, зверніть увагу, що з'являється додаткова інформація при переході в репозиторій.

Тепер, якщо ви знаходитесь у звичайному каталозі, запрошення виглядає так:

```
[~]
vagrant@jessie-i386:
$
```

При переході до репозиторію Git:

```
[~]
vagrant@jessie-i386:
$ cd tools/first_repo/

[~/tools/first_repo]
vagrant@jessie-i386: [master L|✓]
```

🕒 May 23, 2023

🕒 May 15, 2023

РОБОТА З GIT

Для керування Git використовуються різні команди, зміст яких пояснюється далі.

git status

При роботі з Git важливо розуміти поточний статус репозиторію. Для цього у Git є команда git status

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1✓]
13:02 $ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Git повідомляє, що ми знаходимося в гілці master (ця гілка створюється сама і використовується за замовчуванням), і що йому нема чого додавати в коміт. Крім цього, Git пропонує створити або скопіювати файли і після цього скористатися командою git add, щоб Git почав стежити за ними.

Створення файлу README та додавання до нього рядка "test"

```
$ vi README
$ echo "test" >> README
```

Після цього запрошення виглядає таким чином

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1...2]
```

У запрошенні показано, що є два файли, за якими Git ще не стежить

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1...2]
13:14 $ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .README.un~
        README

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

Два файли вийшло через те, що в даному випадку для Vim налаштовані undo-файли. Це спеціальні файли, завдяки яким можна скасовувати зміни не тільки в поточній сесії роботи з файлом, а й у минулі. Зауважте, що Git повідомляє, що є файли, за якими він не стежить і підказує, якою командою це зробити.

Файл .gitignore

Undo-файл .README.un~ – службовий файл, який не потрібно додавати до репозиторію. Git має можливість вказати, які файли або каталоги потрібно ігнорувати. Для цього потрібно створити відповідні шаблони у файлі .gitignore у каталозі репозиторію.

Для того, щоб Git ігнорував undo-файли Vim, можна додати, наприклад, такий рядок до файлу .gitignore

```
*.un~
```

Це означає, що Git повинен ігнорувати всі файли, які закінчуються на .un ~.

Після цього, git status показує

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1...2]
13:33 $ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        README

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

Зверніть увагу, що тепер у виводі немає файлу .README.un~. Як тільки до репозиторій було додано файл .gitignore, файли, які вказані в ньому, стали ігноруватися.

git add

Для того щоб Git почав стежити за файлами, використовується команда git add.

Можна вказати що слід стежити за конкретним файлом

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1...2]
13:33 $ git add README
```

Або за всіма файлами

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|●1...1]
13:36 $ git add .
```

git status:

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|●2]
13:36 $ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   README
```

Тепер файли знаходяться у секції під назвою "Changes to be committed".

git commit

Після того, як всі потрібні файли були додані в staging, можна змінити зміни. Staging - це сукупність файлів, які будуть додані до наступного коміту. Команда git commit має лише один обов'язковий параметр – опція "-m". Він дає змогу вказати повідомлення для цього коміту.

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|●2]
13:37 $ git commit -m "First commit. Add .gitignore and README files"
[master (root-commit) ef84733] First commit. Add .gitignore and README files
 2 files changed, 3 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 README
```

Після цього git status відображає

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|✓]
13:47 $ git status
On branch master
nothing to commit, working directory clean
```

Фраза nothing to commit, working directory clean означає, що немає змін, які потрібно додати до Git або закомітити.

🕒 May 23, 2023

🕒 May 15, 2023

ДОДАТКОВІ МОЖЛИВОСТІ

git diff

Команда `git diff` дозволяє переглянути різницю між різними станами. Наприклад, на даний момент в репозиторії внесені зміни до файлу `README` і `.gitignore`.

Команда `git status` показує, що обидва файли змінені

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1+ 2]
13:53 $ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .gitignore
        modified:   README

no changes added to commit (use "git add" and/or "git commit -a")
```

Команда `git diff` показує, які зміни було внесено з моменту останнього комміту

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1+ 2]
13:53 $ git diff
diff --git a/.gitignore b/.gitignore
index 8eee101..07aab05 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,2 +1,2 @@
 *.un~
-
+*.pyc
diff --git a/README b/README
index 2e7479e..79a508e 100644
--- a/README
+++ b/README
@@ -1 +1,3 @@
 First try
+
+Additional comment
```

Якщо додати зміни, внесені до файлів, в staging командою `git add` і ще раз виконати команду `git diff`, вона нічого не покаже


```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|+ 2]
13:54 $ git add .

[~/tools/first_repo]
vagrant@jessie-i386: [master L|● 2]
13:57 $ git diff
```

Щоб показати відмінності між staging та останнім коммітом, треба додати параметр `--staged`

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|● 2]
13:57 $ git diff --staged
diff --git a/.gitignore b/.gitignore
index 8eee101..07aab05 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,2 +1,2 @@
*.un~
-
+*.pyc
diff --git a/README b/README
index 2e7479e..79a508e 100644
--- a/README
+++ b/README
@@ -1 +1,3 @@
First try
+
+Additional comment
```

Закомітити зміни

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|● 2]
13:59 $ git commit -m "Update .gitignore and README"
[master 58bb8ce] Update .gitignore and README
2 files changed, 3 insertions(+), 1 deletion(-)
```

`git log`

Команда `git log` показує, коли було виконано останні зміни

```
[~/tools/first_repo]
vagrant@jessie-i386: [master LI✓]
14:00 $ git log
commit 58bb8cecbc08a8be76288e96b06d6a875f91a9b1
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 14:00:53 2017 +0000

    Update .gitignore and README

commit ef8473307e0a119496ef154e0bcaff703b1f8a71
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 13:47:30 2017 +0000

    First commit. Add .gitignore and README files
```

За замовчуванням команда показує всі комміти, починаючи з найближчого часу. За допомогою додаткових параметрів можна не тільки переглянути інформацію про коміти, але й те, які зміни були внесені.

Опція `-p` дозволяє відобразити відмінності, внесені кожним коммітом.

```
[~/tools/first_repo]
vagrant@jessie-i386: [master LI✓]
14:02 $ git log -p
commit 58bb8cecbc08a8be76288e96b06d6a875f91a9b1
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 14:00:53 2017 +0000

    Update .gitignore and README

diff --git a/.gitignore b/.gitignore
index 8eee101..07aab05 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,2 +1,2 @@
-*.un~
-
+*.pyc
diff --git a/README b/README
index 2e7479e..79a508e 100644
--- a/README
+++ b/README
@@ -1 +1,3 @@
-First try
+
+Additional comment

commit ef8473307e0a119496ef154e0bcaff703b1f8a71
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 13:47:30 2017 +0000

    First commit. Add .gitignore and README files

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..8eee101
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,2 @@
+*.un~
+
diff --git a/README b/README
new file mode 100644
```

Коротший варіант можна вивести з опцією `--stat`

```
[~/tools/first_repo]
vagrant@jessie-i386: [master LI✓]
14:05 $ git log --stat
commit 58bb8cecbc08a8be76288e96b06d6a875f91a9b1
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 14:00:53 2017 +0000

    Update .gitignore and README

.gitignore | 2 +-
README     | 2 ++
2 files changed, 3 insertions(+), 1 deletion(-)

commit ef8473307e0a119496ef154e0bcaff703b1f8a71
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 13:47:30 2017 +0000

    First commit. Add .gitignore and README files

.gitignore | 2 ++
README     | 1 +
2 files changed, 3 insertions(+)
```

🕒 May 23, 2023

🕒 May 15, 2023

АУТЕНТИФІКАЦІЯ НА GITHUB

Щоб почати працювати з GitHub, треба на ньому [zareestruvatisia](#). Для безпечної роботи з GitHub краще використовувати аутентифікацію за ключами SSH.

Генерація нового SSH-ключа (використовуйте e-mail, який прив'язаний до GitHub):

```
$ ssh-keygen -t rsa -b 4096 -C "github_email@gmail.com"
```

На всіх питаннях достатньо натиснути Enter (безпечніше використовувати ключ з passphrase, але можна і без, якщо натиснути Enter при питанні, тоді passphrase не буде запитуватися у вас при операціях з репозиторієм).

SSH-агент використовується для зберігання ключів у пам'яті та зручний тим, що немає необхідності вводити пароль passphrase щоразу при взаємодії з віддаленим хостом (у даному випадку – github.com).

Запуск SSH-агента (пропускаємо на Windows):

```
$ eval "$(ssh-agent -s)"
```

Додати ключ до SSH-агента (пропускаємо на Windows):

```
$ ssh-add ~/.ssh/id_rsa
```

Додавання SSH-ключа на GitHub

Для додавання ключа його потрібно скопіювати.

Наприклад, таким чином можна відобразити ключ для копіювання:

```
$ cat ~/.ssh/id_rsa.pub
```

Після копіювання потрібно перейти на GitHub. Перебуваючи на будь-якій сторінці GitHub, у правому верхньому кутку натисніть на зображення вашого профілю і в списку виберіть «Settings». У списку зліва треба вибрати поле "SSH and GPG keys". Після цього потрібно натиснути New SSH key і в полі Title написати назву ключа (наприклад Home), а в поле Key вставити вміст, який було скопійовано з файлу ~/.ssh/id_rsa.pub.

Якщо GitHub запросить пароль, введіть пароль свого облікового запису на GitHub.

Щоб перевірити, чи все пройшло успішно, спробуйте виконати команду `ssh -T git@github.com`.

```
$ ssh -T git@github.com
Hi username! You've successfully authenticated, but GitHub does not provide shell access.
```

Тепер ви готові працювати з Git та GitHub.

🕒 May 23, 2023

🕒 May 15, 2023

РОБОТА ЗІ СВОЇМ РЕПОЗИТОРІЄМ ЗАВДАНЬ

У цьому розділі описується, як створити свій репозиторій із копією файлів завдань.

Створення репозиторію на GitHub

Для створення свого репозиторію на основі шаблону потрібно:

- залогінитися на [GitHub](#)
- відкрити [репозиторій із завданнями](#)
- натиснути «Use this template» та створити новий репозиторій на основі цього шаблону
- у вікні треба ввести назву репозиторію
- після цього готовий новий репозиторій із копією всіх файлів із вихідного репозиторію із завданнями

Клонування репозиторію з GitHub

Для локальної роботи з репозиторієм його необхідно клонувати. Для цього використовується команда `git clone`:

```
$ git clone ssh://git@github.com/natenka/my_pyneng_tasks.git
Cloning into 'my_pyneng_tasks'...
remote: Counting objects: 241, done.
remote: Compressing objects: 100% (191/191), done.
remote: Total 241 (delta 43), reused 239 (delta 41), pack-reused 0
Receiving objects: 100% (241/241), 119.60 KiB | 0 bytes/s, done.
Resolving deltas: 100% (43/43), done.
Checking connectivity... done.
```

Порівняно з наведеною в цьому лістингу командою вам потрібно змінити:

- ім'я користувача `natenka` на ім'я користувача на GitHub
- ім'я репозиторію `my_pyneng_tasks` на ім'я свого репозиторію на GitHub

У результаті, в поточному каталозі, в якому було виконано команду `git clone`, з'явиться каталог з ім'ям репозиторію, в моєму випадку - "`my_pyneng_tasks`". У цьому каталозі тепер міститься вміст репозиторію на GitHub.

Робота з репозиторієм

Попередня команда не просто скопіювала репозиторій, щоб використовувати його локально, але й налаштувала відповідним чином Git:

- створено каталог `.git`
- завантажено всі дані репозиторію
- завантажено всі зміни, які були в репозиторії
- репозиторій на GitHub налаштований як `remote` для локального репозиторію

Тепер готовий повноцінний локальний репозиторій Git, у якому ви можете працювати. Зазвичай послідовність роботи буде такою:

- перед початком роботи, синхронізація локального вмісту з GitHub командою `git pull`
- зміна файлів репозиторію
- додавання змінених файлів до staging командою `git add`
- фіксація змін через коміт командою `git commit`
- передача локальних змін у репозиторії на GitHub командою `git push`

При роботі із завданнями на роботі та вдома, треба звернути особливу увагу на перший та останній крок:

- перший крок – оновлення локального репозиторію
- останній крок – завантаження змін на GitHub

Синхронізація локального репозиторію з віддаленим

Усі команди виконуються всередині каталогу репозиторію (у прикладі вище - `my_pyneng_tasks`).

Якщо вміст локального репозиторію однаковий з віддаленим, вивід буде таким:

```
$ git pull
Already up-to-date.
```

Якщо були зміни, виведення буде приблизно таким:

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 5 (delta 4), reused 5 (delta 4), pack-reused 0
Unpacking objects: 100% (5/5), done.
From ssh://github.com/natenka/my_pyneng_tasks
   89c04b6..fc4c721  master    -> origin/master
Updating 89c04b6..fc4c721
Fast-forward
 exercises/03_data_structures/task_3_3.py | 2 ++
 1 file changed, 2 insertions(+)
```

Додавання нових файлів або змін до існуючих

Якщо потрібно додати конкретний файл (у даному випадку - `README.md`), потрібно дати команду `git add README.md`. Додавання всіх файлів поточної директорії здійснюється командою `git add ..`

Коміт

Під час виконання коміту обов'язково треба вказати повідомлення. Краще, якщо повідомлення буде зі змістом, а не просто «update» чи подібне. Коміт робиться командою, подібною до `git commit -m "Зроблені завдання 4.1-4.3"`.

Push на GitHub

Для завантаження всіх локальних змін на GitHub використовується команда `git push`:


```
$ git push origin master
Counting objects: 5, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 426 bytes | 0 bytes/s, done.
Total 5 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
To ssh://git@github.com:natenka/my_pyneng_tasks.git
    fc4c721..edcf417  master -> master
```

Перед виконанням `git push` можна виконати команду `git log -p origin/master..` - вона покаже, які зміни ви збираєтеся додавати до свого репозиторію на GitHub.

🕒 May 23, 2023

🕒 May 15, 2023

РОБОТА З РЕПОЗИТОРІЄМ ПРИКЛАДІВ

Усі приклади з лекцій [курсу](#) викладено в окремому [репозиторії](#).

Копіювання репозиторію з GitHub

Приклади оновлюються, тому зручніше буде клонувати цей репозиторій на свою машину та періодично оновлювати його.

Для копіювання репозиторію з GitHub виконайте команду git clone:

```
$ git clone https://github.com/natenka/pynenguk-examples
Cloning into 'pynenguk-examples'...
remote: Counting objects: 1263, done.
remote: Compressing objects: 100% (504/504), done.
remote: Total 1263 (delta 735), reused 1263 (delta 735), pack-reused 0
Receiving objects: 100% (1263/1263), 267.10 KiB | 444.00 KiB/s, done.
Resolving deltas: 100% (735/735), done.
Checking connectivity... done.
```

Оновлення локальної копії репозиторію

При необхідності оновити локальний репозиторій, щоб синхронізувати його з версією на GitHub, потрібно виконати git pull, перебуваючи всередині створеного каталогу pynenguk-examples.

Якщо оновлень не було, вивід буде таким:

```
$ git pull
Already up-to-date.
```

Якщо оновлення були, результат буде приблизно таким:

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/natenka/pynenguk-examples
 49e9f1b..1eb82ad master    -> origin/master
Updating 49e9f1b..1eb82ad
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Зверніть увагу на інформацію про те, що змінився лише файл README.md.

Перегляд змін

Якщо ви хочете подивитися, які зміни були внесені, можна скористатися командою git log:

```
$ git log -p -1
commit 98e393c27e7aae4b41878d9d979c7587bfeb24b4
Author: Наталія Самойленко[test@gmail.com]
Date:   Fri Aug 18 17:32:07 2017 +0300

    Update task_x_x.py

diff --git a/exercises/task_x_x.py b/exercises/task_x_x.py
index c4307fa..137a221 100644
--- a/exercises/task_x_x.py
+++ b/exercises/task_x_x.py
@@ -13,11 +13,12 @@
 * застосувати ACL до інтерфейсу

ACL має бути таким
+
ip access-list extended INET-to-LAN
 permit tcp 10.0.1.0 0.0.0.255 any eq www
 permit tcp 10.0.1.0 0.0.0.255 any eq 22
 permit icmp any any
-
+
```

Перевірте роботу Playbook на маршрутизаторі R1.

В этой команде флаг `-p` указывает, что надо отобразить вывод утилиты Linux `diff` для внесённых изменений, а не только сообщение коммита. В свою очередь, `-1` указывает, что надо показать только один самый свежий коммит.

У цій команді опція `-p` вказує, що треба відобразити вихід утиліти Linux `diff` для внесених змін, а не лише повідомлення комміту. У свою чергу, `-1` вказує, що треба показати лише один найсвіжіший коміт.

Перегляд змін, які будуть синхронізовані

Попередній варіант `git log` спирається на кількість коммітів, але це не завжди зручно. До виконання команди `git pull` можна переглянути, які зміни були виконані з моменту останньої синхронізації.

Для цього використовується наступна команда:

```
$ git log -p ..origin/master
commit 4c1821030d20b3682b67caf362fd777d098d9126
Author: Наталія Самойленко[test@gmail.com]
Date:   Mon May 29 07:53:45 2017 +0300

    Update README.md

diff --git a/tools/README.md b/tools/README.md
index 2b6f380..4f8d4af 100644
--- a/tools/README.md
+++ b/tools/README.md
```

```
@@ -1 +1,4 @@  
+
```

У цьому випадку зміни були лише в одному файлі. Ця команда буде дуже корисною для того, щоб подивитися, які зміни були внесені до формулювання завдань та яких саме завдань. Так буде легше орієнтуватися і розуміти, чи це стосується завдань, які ви вже зробили, і якщо стосується, то чи треба їх змінювати.

"`..origin/master`" у команді `git log -p ..origin/master` означає показати всі комміти, які є в `origin/master` (в даному випадку, це GitHub), але яких немає в локальній копії репозиторію

Якщо зміни були в тих завданнях, які ви ще не робили, цей вивід підкаже, які файли потрібно скопіювати з репозиторію курсу у ваш особистий репозиторій (а може бути весь розділ, якщо ви ще не робили завдання з цього розділу).

🕒 May 23, 2023

🕒 May 15, 2023

Додаткові матеріали

Документація:

- [Informative git prompt for bash and fish](#)
- [Authenticating to GitHub](#)
- [Connecting to GitHub with SSH](#)

Про Git/GitHub:

- [GitHowTo](#) - інтерактивний howto українською
- [Pro Git book](#). Ця ж книга українською
- [Лекції "Система контролю версій Git" \(Попелюха\)](#)
- [CRLF vs. LF: Normalizing Line Endings in Git](#)
- [git/github guide. a minimal tutorial](#) - мінімально необхідні знання для роботи з Git та GitHub

Створення `.gitignore`:

- [.gitignore для Python](#)
- [Як ігнорувати файли](#)
- [Синтаксис .gitignore](#)

🕒 May 26, 2023

🕒 May 15, 2023

Основи Python

3. Початок роботи з Python

3. Початок роботи з Python

У цьому розділі розглядаються:

- синтаксис Python
- робота в інтерактивному режимі
- змінні в Python

🕒 May 27, 2023

🕒 May 15, 2023

Синтаксис Python

У Python відступи є частиною синтаксису:

- вони визначають, який код потрапляє у блок;
- коли блок коду починається та закінчується.

Приклад коду Python:

```
a = 10
b = 5

if a > b:
    print("А більше В")
    print(a - b)
else:
    print("В більше або дорівнює А")
    print(b - a)

print("Кінець")

def open_file(filename):
    print("Читання файлу", filename)
    with open(filename) as f:
        return f.read()
    print("Готово")
```

Цей код показано для демонстрації синтаксису. І незважаючи на те, що ще не розглядалася конструкція if/else, швидше за все, суть коду буде зрозумілою.

Python розуміє, які рядки відносяться до if по відступах. Виконання блоку `if a > b` закінчується, коли зустрічається рядок із тим самим відступом, як і сам рядок `if a > b`. Аналогічно із блоком else. Друга особливість Python: після деяких виразів має йти двокрапка (наприклад, після `if a > b` і після `else`).

Декілька правил і рекомендацій щодо відступів:

- Як відступи можуть використовуватися таби або пробіли
- краще використовувати пробіли, а точніше, налаштувати редактор так, щоб таб дорівнював 4 пробілам тоді при використанні клавіші табуляції будуть ставитися 4 пробіли, замість 1 знака табуляції
- Кількість пробілів має бути однаковою в одному блоці (краще, щоб кількість пробілів була однаковою у всьому коді)
- популярний варіант використовувати 2-4 пробіли, так, наприклад, у цій книзі використовуються 4 пробіли

Ще одна особливість наведеного коду, це порожні рядки. З їхньою допомогою код форматується, щоб його було простіше читати. Інші особливості синтаксису будуть показані в процесі знайомства зі структурами даних у Python.

У Python є спеціальний документ, в якому описано як краще писати код Python [PEP 8 - the Style Guide for Python Code](#).

КОМЕНТАРІ

При написанні коду часто потрібно залишити коментар, наприклад, щоб описати особливості роботи коду.

Коментарі в Python можуть бути однорядковими:

```
# Дуже важливий коментар  
a = 10  
b = 5 # Дуже потрібний коментар
```

Однорядкові коментарі починаються зі знака решітки. Зверніть увагу, що коментар може бути як у рядку, де знаходиться сам код, так і в окремому рядку.

При необхідності написати кілька рядків з коментарями, щоб не ставити перед кожною решіткою, можна зробити багаторядковий коментар:

```
"""  
Дуже важливий  
та довгий коментар  
"""  
  
a = 10  
b = 5
```

Для багаторядкового коментаря можна використовувати три подвійні або три одинарні лапки. Коментарі можуть використовуватися як для того, щоб коментувати, що відбувається в коді, так і для того, щоб унеможливити виконання певного рядка або блоку коду (тобто закоментувати їх).

🕒 May 27, 2023

🕒 May 15, 2023

Інтерпретатор Python. IPython

Інтерпретатор дозволяє отримати миттєвий відгук на виконані дії. Можна сказати, що інтерпретатор працює як командний рядок мережевих пристроїв: кожна команда виконуватиметься відразу після натискання Enter. Однак є виняток - складніші вирази (наприклад цикли або функції) виконуються тільки після дворазового натискання Enter.

У попередньому розділі для перевірки установки Python викликався стандартний інтерпретатор. Крім нього, є й удосконалений інтерпретатор [IPython](#). IPython дозволяє набагато більше, ніж стандартний інтерпретатор, який викликається за командою `python`. Декілька прикладів (можливості IPython набагато ширші):

- автодоповнення команд по Tab або підказка, якщо варіантів команд декілька
- більш структурований та зрозумілий вивід команд
- автоматичні відступи у циклах та інших об'єктах
- можна пересуватися з історії виконання команд, або ж подивитися її «чарівною» командою історії

Встановити IPython можна за допомогою `pip` (установка буде проводитися у віртуальному оточенні, якщо воно налаштоване):

```
pip install ipython
```

Після цього, перейти в IPython можна так:

```
$ ipython
Python 3.11.1 (main, Dec 30 2022, 10:30:23) [GCC 10.2.1 20210110]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.13.2 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Для виходу використовується команда `quit`. Далі описується, як використовуватиметься IPython.

Для знайомства з інтерпретатором можна спробувати використати його як калькулятор:

```
In [1]: 1 + 2
Out[1]: 3

In [2]: 22*45
Out[2]: 990

In [3]: 2**3
Out[3]: 8
```

В IPython введення та вивід позначені:

- In - вхідні дані користувача

- Out - результат, який повертає команда (якщо він є)
- числа після In або Out - це порядкові номери виконаних команд у поточній сесії IPython

Приклад виведення рядка функцією print():

```
In [4]: print('Hello!')
Hello!
```

Коли в інтерпретаторі створюється, наприклад, цикл, то всередині циклу запрошення змінюється на крапки. Для виконання циклу та виходу з цього підрежиму необхідно двічі натиснути Enter:

```
In [5]: for i in range(5):
...:     print(i)
...:
0
1
2
3
4
```

HELP

В IPython є можливість переглянути довідку по довільному об'єкту, функції або методу за допомогою help():

```
In [1]: help(str)
Help on class str in module builtins:

class str(object)
|   str(object='') -> str
|   str(bytes_or_buffer[, encoding[, errors]]) -> str
|
|   Create a new string object from the given object. If encoding or
|   errors is specified, then the object must expose a data buffer
|   that will be decoded using the given encoding and error handler.
...

In [2]: help(str.strip)
Help on method_descriptor:

strip(...)
    S.strip([chars]) -> str

    Return a copy of the string S with leading and trailing
    whitespace removed.
    If chars is given and not None, remove characters in chars instead.
```

Другий варіант:

```
In [3]: ?str
Init signature: str(self, /, *args, **kwargs)
Docstring:
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
Type:          type

In [4]: ?str.strip
Docstring:
S.strip([chars]) -> str

Return a copy of the string S with leading and trailing
whitespace removed.
If chars is given and not None, remove characters in chars instead.
Type:          method_descriptor
```

print

Функція `print` дозволяє вивести інформацію стандартний потік виведення (поточний екран терміналу). Якщо необхідно вивести рядок, то його потрібно обов'язково взяти в лапки (подвійні чи одинарні). Якщо ж потрібно вивести, наприклад, результат обчислення чи просто число, то лапки не потрібні:

```
In [6]: print('Hello!')
Hello!

In [7]: print(5*5)
25
```

Якщо потрібно вивести поспіль кілька значень через пропуск, то потрібно перерахувати їх через кому:

```
In [8]: print(1*5, 2*5, 3*5, 4*5)
5 10 15 20

In [9]: print('one', 'two', 'three')
one two three
```

За замовчуванням наприкінці кожного виразу, переданого в `print`, буде переведено рядок. Якщо необхідно, щоб після виведення кожного виразу не було б перекладу рядка, треба як останній вираз у `print` вказати додатковий аргумент `end`.

DIR

Функція `dir` може використовуватися для того, щоб переглянути атрибути та методи об'єкта:

- атрибути - змінні, прив'язані до об'єкта
- методи - функції, прив'язані до об'єкта

Наприклад, для числа вивід буде таким (зверніть увагу на різні методи, які дозволяють робити арифметичні операції):

```
In [10]: dir(5)
Out[10]:
['__abs__',
 '__add__',
 '__and__',
 ...
 'bit_length',
 'conjugate',
 'denominator',
 'imag',
 'numerator',
 'real']
```

Аналогічно для рядка:

```
In [11]: dir('hello')
Out[11]:
['__add__',
 '__class__',
 '__contains__',
 ...
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

Якщо виконати `dir` без передачі значення, то вона показує існуючі методи, атрибути та змінні, визначені в поточній сесії інтерпретатора:

```
In [12]: dir()
Out[12]:
['__builtin__',
 '__builtins__',
 '__doc__',
 '__name__',
 '_dh',
 ...
 '_oh',
```

```
'_sh',  
'exit',  
'get_ipython',  
'i',  
'quit']
```

Наприклад, після створення змінної `a` та `test`:

```
In [13]: a = 'hello'  
  
In [14]: test = "test"  
  
In [15]: dir()  
Out[15]:  
...  
'a',  
'exit',  
'get_ipython',  
'i',  
'quit',  
'test']
```

 May 27, 2023

 May 15, 2023

Спеціальні команди ipython

У IPython є спеціальні команди, які полегшують роботу з інтерпретатором. Усі вони починаються зі знака відсотка.

%HISTORY

Наприклад, команда `%history` дозволяє переглянути історію введених користувачем команд у поточній сесії IPython:

```
In [1]: a = 10

In [2]: b = 5

In [3]: if a > b:
...:     print("A is bigger")
...:
A is bigger

In [4]: %history
a = 10
b = 5
if a > b:
    print("A is bigger")
%history
```

За допомогою `%history` можна скопіювати потрібний блок коду.

%TIME

Команда `%time` показує скільки секунд виконувався вираз:

```
import subprocess

def ping_ip(ip_address):
    reply = subprocess.run(['ping', '-c', '3', '-n', ip_address],
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,
                           encoding='utf-8')

    if reply.returncode == 0:
        return True
    else:
        return False

In [7]: %time ping_ip('8.8.8.8')
CPU times: user 0 ns, sys: 4 ms, total: 4 ms
Wall time: 2.03 s
Out[7]: True

In [8]: %time ping_ip('8.8.8')
CPU times: user 0 ns, sys: 8 ms, total: 8 ms
Wall time: 12 s
Out[8]: False
```

```
In [9]: items = [1, 3, 5, 7, 9, 1, 2, 3, 55, 77, 33]

In [10]: %time sorted(items)
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 8.11 µs
Out[10]: [1, 1, 2, 3, 3, 5, 7, 9, 33, 55, 77]
```

Докладніше про IPython можна прочитати в [документації](#) IPython.

Коротко інформацію можна переглянути в самому IPython командою %quickref:

```
IPython -- An enhanced Interactive Python - Quick Reference Card
=====

obj?, obj??      : Get help, or more help for object (also works as
                  ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by % or %, and typically take their arguments
without parentheses, quotes or even commas for convenience. Line magics take a
single % and cell magics are prefixed with two %%.

Example magic function calls:

%alias d ls -F    : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias    : Get list of aliases to 'alist'
cd /usr/share     : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd
%timeit x=10      : time the 'x=10' statement with high precision.
%%timeit x=2**100 : time 'x=2**100' with a setup of 'x=2**100'; setup code is not
x**100           : counted. This is an example of a cell magic.

System commands:

!cp a.txt b/      : System command escape, calls os.system()
cp a.txt b/       : after %rehashx, most system commands work without !
cp ${f}.txt $bar  : Variable expansion in magics and system commands
files = !ls /usr  : Capture sytem command output
files.s, files.l, files.n: "a b c", ['a','b','c'], 'a\nb\nc'

History:

_i, _ii, _iii     : Previous, next previous, next next previous input
_i4, _ih[2:5]     : Input history line 4, lines 2-4
exec _i81         : Execute input history line #81 again
%rep 81           : Edit input history line #81
_, __, ___       : previous, next previous, next next previous output
_dh              : Directory history
_oh              : Output history
%hist            : Command history of current session.
%hist -g foo      : Search command history of (almost) all sessions for 'foo'.
```

```
%hist -g      : Command history of (almost) all sessions.  
%hist 1/2-8    : Command history containing lines 2-8 of session 1.  
%hist 1/ ~2/   : Command history of session 1 and 2 sessions before current.
```

🕒 May 27, 2023

🕒 May 15, 2023

Змінні

Змінні в Python не вимагають оголошення типу змінної (оскільки Python – мова з динамічною типізацією) і є посиланнями на область пам'яті. Правила іменування змінних:

- ім'я змінної може складатися лише з літер, цифр та знака підкреслення
- ім'я не може починатися із цифри
- ім'я не може містити спеціальних символів @, \$, %

Приклад створення змінних у Python:

```
In [1]: a = 3

In [2]: b = 'Hello'

In [3]: c, d = 9, 'Test'

In [4]: print(a, b, c, d)
3 Hello 9 Test
```

Зверніть увагу, що в Python не потрібно вказувати, що `a` це число, а `b` це рядок.

ІМЕНА ЗМІННИХ

Імена змінних не повинні перетинатися з назвами операторів та модулів або інших зарезервованих слів. У Python є рекомендації щодо іменування функцій, класів та змінних:

- імена змінних зазвичай пишуться або повністю великими або маленькими літерами (наприклад `DB_NAME`, `db_name`)
- імена функцій задаються маленькими літерами, з підкреслення між словами (наприклад, `get_names`)
- імена класів задаються словами з великими літерами без пробілів, це звані CamelCase (наприклад, `CiscoSwitch`)

В PYTHON ЗМІННІ ЦЕ ПОСИЛАННЯ

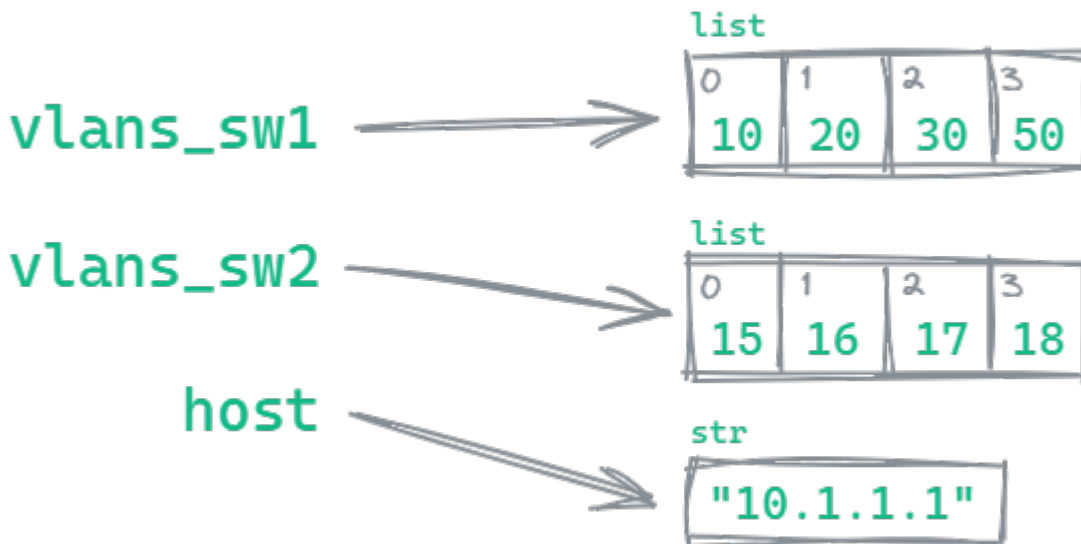
В Python змінні це посилання на об'єкти:

- об'єкт не зберігається у змінній
- змінна це посилання на реальний об'єкт у пам'яті

Приклад коду

```
vlan1_sw1 = [10, 20, 30, 50]
vlan1_sw2 = [15, 16, 17, 18]
host = "10.1.1.1"
```

Ліворуч створені змінні, праворуч - відповідні об'єкти у пам'яті:



Змінну можна сприймати як ярлик чи тег реальних даних. Наприклад, ми можемо створити дві змінні, які посилатимуться на той самий список (тобто два теги):

```
vlans_sw1 = vlans_sw2 = [10, 20, 30, 50]
```

як це виглядає схематично:



Тепер один і той самий об'єкт доступний за двома іменами.

Приклад

На роботі до вас звертаються, наприклад, Олександр, а дома - Сашко, це два імені, але однієї людини.

Список це змінюваний тип даних, тому в цьому випадку дуже важливо те, що дві змінні посилаються на той самий об'єкт. Зміна об'єкту через будь-яку зі змінних, змінює той самий об'єкт. Наприклад, якщо змінити список через змінну `vlans_sw1`:

```
vlans_sw1.append(100)
```

І після цього дати команду `print` для обох змінних, `vlan 100` з'явиться в обох випадках:

```
In [1]: vlans_sw1 = vlans_sw2 = [10, 20, 30, 50]
```

```
In [2]: vlans_sw1.append(100)
```

```
In [3]: print(vlans_sw1)
[10, 20, 30, 50, 100]

In [4]: print(vlans_sw2)
[10, 20, 30, 50, 100]
```

Функція `id` дозволяє подивитися адресу об'єкту в пам'яті. Для змінних `vlans_sw1`, `vlans_sw2` зараз ця адреса буде однаковою:

```
In [5]: id(vlans_sw1)
Out[5]: 139810203190848

In [6]: id(vlans_sw2)
Out[6]: 139810203190848
```

Якщо знов створити два різні списки з двома різними змінними, `id` будуть різні:

```
In [8]: vlans_sw1 = [10, 20, 30, 50]

In [9]: vlans_sw2 = [15, 16, 17, 18]

In [10]: id(vlans_sw1)
Out[10]: 139810203155072

In [11]: id(vlans_sw2)
Out[11]: 139810181687744
```

Ще один важливий момент: коли змінна вказується у виразі праворуч від знака рівності, вона завжди замінюється значенням (об'єктом) на яке вона посилається.

```
In [12]: sw1 = "10.1.1.1"

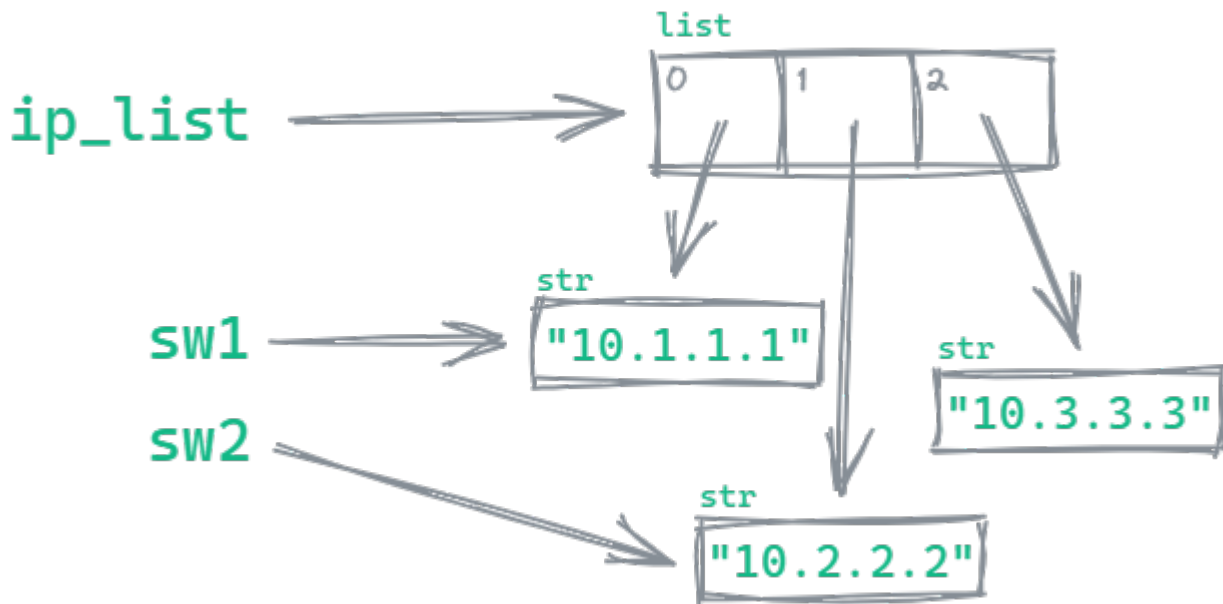
In [13]: sw2 = "10.2.2.2"

In [14]: ip_list = [sw1, sw2, "10.3.3.3"]

In [15]: ip_list
Out[15]: ['10.1.1.1', '10.2.2.2', '10.3.3.3']
```

В списку `ip_list` знаходяться рядки `'10.1.1.1'` та `'10.2.2.2'`, а не змінні `sw1`, `sw2`.

Схематичне зображення посилань:



Елементи списку це посилання на інші об'єкти.

Id об'єкту на який посилається змінна `sw1`, та id першого елементу списку `ip_list` - однакові:

```

In [16]: id(sw1)
Out[16]: 139810202863408

In [17]: id(ip_list[0])
Out[17]: 139810202863408
  
```

Всі ці моменти зі змінними мають найбільше значення при використанні змінюваних типів даних. Важливо розуміти, що зазначення змінної в якомусь виразі Python, підставляє замість змінної об'єкт, на який посилається змінна і саме той самий об'єкт, а не його копію.

Наприклад, вираз `vlan_sw2 = vlan_sw1` означає, що змінна `vlan_sw2` буде посиланням на той самий список, що і змінна `vlan_sw1`:

```

In [19]: vlan_sw1 = [10, 20, 30, 50]

In [20]: vlan_sw2 = vlan_sw1
  
```



🕒 May 27, 2023

🕒 May 15, 2023

4. Типи даних у Python

4. Типи даних у Python

 [Цей розділ в форматі відео](#)

У Python є кілька базових типів даних:

- Numbers (числа)
- Strings (рядки)
- Lists (списки)
- Dictionaries (словники)
- Tuples (кортежі)
- Sets (множини)
- Boolean (логічний тип даних)

Ці типи даних можна класифікувати за кількома ознаками:

- змінювані (списки, словники та множини)
- незмінні (числа, рядки та кортежі)
- упорядковані (списки, кортежі, рядки та словники)
- неупорядковані (множини)

flowchart TD

```
A[змінювані] --> B[список];  
A --> C[словник];  
A --> D[множина];  
F[незмінні] --> R[число];  
F --> S[рядок];  
F --> T[кортеж];
```

flowchart TD

```
A[упорядковані] --> B[список];  
A --> T[кортеж];  
A --> S[рядок];  
A --> C[словник];  
F[неупорядковані] --> D[множина];
```

 June 22, 2023

 May 15, 2023

Числа

У Python є два основних типи чисел:

- integer - цілі числа
- float - числа з плаваючою точкою

З числами можна виконувати різні математичні операції:

```
In [1]: 1 + 2
Out[1]: 3

In [2]: 1.0 + 2
Out[2]: 3.0

In [3]: 10 - 4
Out[3]: 6

In [4]: 2**3
Out[4]: 8
```

Оператори порівняння

```
In [12]: 10 > 3.0
Out[12]: True

In [13]: 10 < 3
Out[13]: False

In [14]: 10 == 3
Out[14]: False

In [15]: 10 == 10
Out[15]: True

In [16]: 10 <= 10
Out[16]: True

In [17]: 10.0 == 10
Out[17]: True
```

Функція `int` дозволяє виконувати конвертацію даних у тип `integer`. У другому аргументі можна вказувати систему числення:

```
In [18]: a = '11'

In [19]: int(a)
Out[19]: 11
```

Якщо вказати, що рядок треба сприймати як двійкове число, результат буде таким:

```
In [20]: int("11", 2)
Out[20]: 3
```

Конвертація в int типу float:

```
In [21]: int(3.333)
Out[21]: 3

In [22]: int(3.9)
Out[22]: 3
```

Функція bin дозволяє отримати двійкове значення числа (зверніть увагу, що результат – рядок):

```
In [23]: bin(8)
Out[23]: '0b1000'

In [24]: bin(255)
Out[24]: '0b11111111'
```

Функція hex дозволяє отримати шістнадцяткове значення:

```
In [25]: hex(10)
Out[25]: '0xa'
```

Можна робити кілька перетворень "одночасно":

```
In [26]: int('ff', 16)
Out[26]: 255

In [27]: bin(int('ff', 16))
Out[27]: '0b11111111'
```

🕒 May 23, 2023

🕒 May 15, 2023

Рядки (Strings)

РЯДКИ (STRINGS)

Рядок у Python це:

- послідовність символів у лапках
- незмінний упорядкований тип даних

У Python рядки можна створювати за допомогою одинарних, подвійних та потрійних лапок (одинарних або подвійних):

```
'Hello'
"Hello"

tunnel = """
interface Tunnel0
 ip address 10.10.10.1 255.255.255.0
 ip mtu 1416
"""
```

Запис рядків у потрійних лапках використовується для зручності і в результаті рядок виходить таким:

```
In [12]: tunnel
Out[12]: '\ninterface Tunnel0\n ip address 10.10.10.1 255.255.255.0\n ip mtu 1416\n'

In [13]: print(tunnel)

interface Tunnel0
 ip address 10.10.10.1 255.255.255.0
 ip mtu 1416
```

Операції з рядками

Рядки можна підсумовувати. Тоді вони об'єднуються в один рядок:

```
In [14]: intf = 'interface'

In [15]: tun = 'Tunnel0'

In [16]: intf + tun
Out[16]: 'interfaceTunnel0'

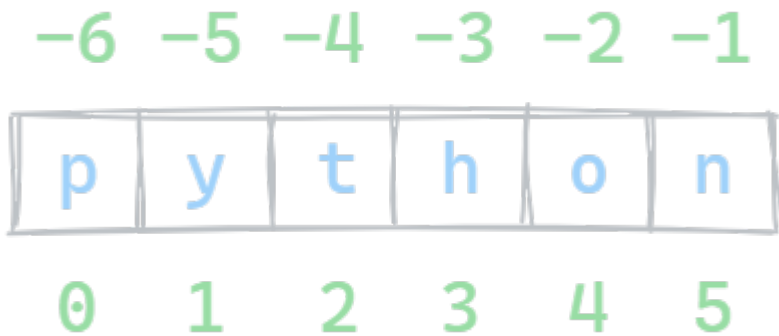
In [17]: intf + ' ' + tun
Out[17]: 'interface Tunnel0'
```

Рядок можна множити на число. У цьому випадку рядок повторюється вказану кількість разів:

```
In [18]: intf * 5
Out[18]: 'interfaceinterfaceinterfaceinterfaceinterface'

In [19]: '#' * 40
Out[19]: '#####'
```

Індекси



Те, що рядки є впорядкованим типом даних, дозволяє звертатися до символів у рядку за номером (індексом), починаючи з нуля:

```
In [20]: string1 = 'interface FastEthernet1/0'

In [21]: string1[0]
Out[21]: 'i'
```

Нумерація всіх символів у рядку йде з нуля. Якщо потрібно звернутися до якогось за рахунком символу, починаючи з кінця, можна вказувати негативні значення (на цей раз з одиниці).

```
In [22]: string1[1]
Out[22]: 'n'

In [23]: string1[-1]
Out[23]: '0'
```

Зрізи

Крім звернення до конкретного символу можна робити зрізи рядків, вказавши діапазон номерів (зріз виконується до другого значення, не включаючи його):

```
In [24]: string1[0:9]
Out[24]: 'interface'

In [25]: string1[10:22]
Out[25]: 'FastEthernet'
```

Якщо не вказується друге число, зріз буде до кінця рядка:

```
In [26]: string1[10:]
Out[26]: 'FastEthernet1/0'
```

Зрізати три останні символи рядка:

```
In [27]: string1[-3:]
Out[27]: '1/0'
```

Також у зрізі можна вказувати крок. Так можна отримати непарні числа:

```
In [28]: a = '0123456789'

In [29]: a[1::2]
Out[29]: '13579'
```

А таким чином можна отримати всі парні числа

```
In [31]: a[::2]
Out[31]: '02468'
```

Зрізи також можна використовувати для отримання рядка у зворотному порядку:

```
In [28]: a = '0123456789'

In [29]: a[::-1]
Out[29]: '9876543210'

In [30]: a[::-1]
Out[30]: '9876543210'
```

Записи `a[::-1]` і `a[::]` дають однаковий результат, але подвійна двокрапка дозволяє вказувати, що треба брати не кожен елемент, а, наприклад, кожен другий.

Функція `len`

Функція `len` дозволяє отримати кількість символів у рядку:

```
In [1]: line = 'interface Gi0/1'

In [2]: len(line)
Out[2]: 15
```

🕒 May 31, 2023

🕒 May 15, 2023

КОРИСНІ МЕТОДИ ДЛЯ РОБОТИ З РЯДКАМИ

При автоматизації дуже часто треба буде працювати з рядками, так як конфігураційний файл, виведення команд і команди, що відправляються - це рядки. Знання різних методів (дій), які можна застосовувати до рядків, допомагає ефективно працювати з ними.

Рядки незмінний тип даних, тому всі методи, які перетворюють рядок повертають новий рядок, а вихідний рядок залишається незмінним.

Методи `upper`, `lower`, `swapcase`, `capitalize`

Методи `upper`, `lower`, `swapcase`, `capitalize` виконують перетворення регістра рядка:

```
In [25]: string1 = 'FastEthernet'

In [26]: string1.upper()
Out[26]: 'FASTETHERNET'

In [27]: string1.lower()
Out[27]: 'fastethernet'

In [28]: string1.swapcase()
Out[28]: 'fASTeTHERNET'

In [29]: string2 = 'tunnel 0'

In [30]: string2.capitalize()
Out[30]: 'Tunnel 0'
```

Методи не перетворюють вихідний рядок, а повертають новий із виконаним перетворенням. Це означає, що треба не забути записати його в якусь змінну (можна в ту саму).

```
In [31]: string1 = string1.upper()

In [32]: print(string1)
FASTETHERNET
```

Метод `join`

 [Метод join в довіднику](#)

Метод `join` збирає список рядків в один рядок із роздільником, який вказаний перед `join`:

```
In [16]: vlans = ['10', '20', '30']

In [17]: ','.join(vlans)
Out[17]: '10,20,30'
```

Метод count

Метод `count` використовується для підрахунку того, скільки разів символ або підрядок зустрічаються у рядку:

```
In [33]: string1 = 'Hello, hello, hello, hello'

In [34]: string1.count('hello')
Out[34]: 3

In [35]: string1.count('ello')
Out[35]: 4

In [36]: string1.count('l')
Out[36]: 8
```

Метод find

Методу `find` можна передати підрядок або символ, і він покаже, на якій позиції знаходиться перший символ підрядка (перший збіг):

```
In [37]: string1 = 'interface FastEthernet0/1'

In [38]: string1.find('Fast')
Out[38]: 10

In [39]: string1[string1.find('Fast'):]
Out[39]: 'FastEthernet0/1'
```

Якщо збіг не виявлено, метод `find` повертає `-1`.

Методи startswith, endswith

Перевірка на те, чи починається чи закінчується рядок на певні символи (методи `startswith`, `endswith`):

```
In [40]: string1 = 'FastEthernet0/1'

In [41]: string1.startswith('Fast')
Out[41]: True

In [42]: string1.startswith('fast')
Out[42]: False

In [43]: string1.endswith('0/1')
Out[43]: True

In [44]: string1.endswith('0/2')
Out[44]: False
```

Методам `startswith` і `endswith` можна передавати кілька значень (обов'язково як кортеж):

```

In [1]: "test".startswith(("r", "t"))
Out[1]: True

In [2]: "test".startswith(("r", "a"))
Out[2]: False

In [3]: "rtest".startswith(("r", "a"))
Out[3]: True

In [4]: "rtest".endswith(("r", "a"))
Out[4]: False

In [5]: "rtest".endswith(("r", "t"))
Out[5]: True

```

Метод replace

Заміна послідовності символів у рядку на іншу послідовність (метод `replace`):

```

In [5]: string1 = 'FastEthernet0/1'

In [6]: string1.replace('Fast', 'Gigabit')
Out[6]: 'GigabitEthernet0/1'

In [7]: line = "aabb.cc10.a1a0"

In [8]: line.replace("a", "A")
Out[8]: 'AAbb.cc10.A1A0'

```

Метод strip

У рядку часто будуть спеціальні символи: символ нового рядка, пробіли, таби. Метод `strip` дозволяє видалити ці символи на початку та в кінці рядка.

```

In [47]: string1 = '\n\tinterface FastEthernet0/1\n'

In [48]: print(string1)

    interface FastEthernet0/1

In [49]: string1
Out[49]: '\n\tinterface FastEthernet0/1\n'

In [50]: string1.strip()
Out[50]: 'interface FastEthernet0/1'

```

За замовчуванням метод `strip` забирає пробілові символи. Цей набір символів містить: `\t\n\r\f\v`.

Методу `strip` можна передати як аргумент будь-які символи. Тоді на початку та в кінці рядка будуть видалені всі символи, які були вказані у рядку:

```
In [51]: ad_metric = '[110/1045]'

In [52]: ad_metric.strip('[]')
Out[52]: '110/1045'
```

Метод `strip` прибирає вказані символи на початку і в кінці рядка. Якщо необхідно прибрати символи лише ліворуч або праворуч, можна використовувати, відповідно, методи `lstrip` і `rstrip`.

Метод `split`

 [Метод `split` в довіднику](#)

Метод `split` розбиває рядок на частини, використовуючи як роздільник якийсь символ (або символи) та повертає список рядків:

```
In [53]: string1 = 'switchport trunk allowed vlan 10,20,30,100'

In [54]: commands = string1.split()

In [55]: print(commands)
['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100']
```

У прикладі вище `string1.split` розбиває рядок за пробілом і повертає список рядків. Список записаний у змінну `commands`.

За замовчуванням як роздільник використовуються пробільні символи (пробіли, таби, символ нового рядка), але в дужках можна вказати будь-який роздільник:

```
In [56]: vlans = commands[-1].split(',')

In [57]: print(vlans)
['10', '20', '30', '100']
```

У списку `commands` останній елемент - це рядок з вланами, тому використовується індекс `-1`. Потім рядок розбивається на частини за допомогою `split` `commands[-1].split(',')`. Оскільки, як роздільник зазначена кома, отримано такий список `['10', '20', '30', '100']`.

Приклад поділу адреси на октети:

```
In [10]: ip = "192.168.100.1"

In [11]: ip.split(".")
Out[11]: ['192', '168', '100', '1']
```

Корисна особливість методу `split` з роздільником за замовчуванням - рядок не лише поділяється до списку рядків за пробіловими символами, але пробілові символи також видаляються на початку та наприкінці рядка:

```
In [58]: string1 = ' switchport trunk allowed vlan 10,20,30,100-200\n\n'

In [59]: string1.split()
Out[59]: ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100-200']
```

У методу `split` є ще одна хороша особливість: за замовчуванням метод розбиває рядок не за одним пробіловим символом, а за будь-якою кількістю. Це буде, наприклад, дуже корисним при обробці команд `show`:

```
In [60]: sh_ip_int_br = "FastEthernet0/0      15.0.15.1    YES manual up      up"

In [61]: sh_ip_int_br.split()
Out[61]: ['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up']
```

А ось так виглядає поділ того ж рядка, коли один пробіл використовується як роздільник:

```
In [62]: sh_ip_int_br.split(' ')
Out[62]:
['FastEthernet0/0', '', '', '', '', '', '', '', '', '', '', '', '15.0.15.1', '', '', '',
'', '', '', 'YES', 'manual', 'up', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', 'up']
```

🕒 June 19, 2023

🕒 May 15, 2023

ОБ'ЄДНАННЯ ЛІТЕРАЛІВ РЯДКІВ

У Python є дуже зручна функціональність - об'єднання **літералів** рядків. Вона дозволяє розбивати рядки на частини при написанні коду і навіть переносити ці частини на різні рядки коду. Це потрібно як для розділення довгого тексту на частини через рекомендації щодо максимальної довжини рядка в Python, так і для зручності сприйняття.

Приклад об'єднання літералів рядків:

```
In [1]: s = 'Test' 'String'

In [2]: s
Out[2]: 'TestString'
```

Можна переносити складові рядки на різні рядки, але тільки якщо вони в дужках:

```
In [5]: s = ('Test'
...: 'String')

In [6]: s
Out[6]: 'TestString'
```

Цим дуже зручно користуватися в регулярних виразах:

```
regex = ('(\S+) +(\S+) +'
         '\w+ +\w+ +'
         '(up|down|administratively down) +'
         '(\w+)')
```

Регулярний вираз можна розбивати на частини, так його простіше зрозуміти. Плюс можна додавати пояснювальні коментарі у рядках.

```
regex = ('(\S+) +(\S+) +' # interface and IP
         '\w+ +\w+ +'
         '(up|down|administratively down) +' # Status
         '(\w+)') # Protocol
```

Також цим прийомом зручно користуватися, коли треба написати довге повідомлення:

```
message = (
    'При виконанні команди "{}" виникла така помилка "{}".\n'
    'Виключити цю команду зі списку? [y/n]'
)

In [8]: message
Out[8]: 'При виконанні команди "{}" виникла така помилка "{}".\nВиключити цю команду зі списку? [y/n]'
```

🕒 May 29, 2023

🕒 May 15, 2023

Список (List)

СПИСОК (LIST)

Список у Python це:

- послідовність елементів у квадратних дужках, розділених комою
- змінюваний упорядкований тип даних

Приклади списків:

```
vlan = [10, 20, 30, 77]
commands = ["interface Gi0/1", "ip address 10.1.1.1 255.255.255.0"]
list3 = [1, 20, 4.0, 'word']
```

Створення списку за допомогою літералу:

```
In [1]: vlan = [10, 20, 30, 50]
```

Створення списку за допомогою list:

```
In [2]: list1 = list('router')

In [3]: print(list1)
['r', 'o', 'u', 't', 'e', 'r']
```

Індекси, зрізи

Так як список - це впорядкований тип даних, то, як і в рядках, у списках можна звертатися до елемента за номером, робити зрізи:

```
In [4]: list3 = [1, 20, 4.0, 'word']

In [5]: list3[1]
Out[5]: 20

In [6]: list3[1::]
Out[6]: [20, 4.0, 'word']

In [7]: list3[-1]
Out[7]: 'word'

In [8]: list3[::-1]
Out[8]: ['word', 4.0, 20, 1]
```

Оскільки список є змінним типом даних, елементи списку можна змінювати:

```
In [13]: list3
Out[13]: [1, 20, 4.0, 'word']
```

```
In [14]: list3[0] = 'test'

In [15]: list3
Out[15]: ['test', 20, 4.0, 'word']
```

Можна також створювати список списків. І, як і у звичайному списку, можна звертатися до елементів у вкладених списках:

```
interfaces = [['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up'],
               ['FastEthernet0/1', '10.0.1.1', 'YES', 'manual', 'up', 'up'],
               ['FastEthernet0/2', '10.0.2.1', 'YES', 'manual', 'up', 'down']]

In [17]: interfaces[0][0]
Out[17]: 'FastEthernet0/0'

In [18]: interfaces[2][0]
Out[18]: 'FastEthernet0/2'

In [19]: interfaces[2][1]
Out[19]: '10.0.2.1'
```

len, sorted

Функція len повертає кількість елементів у списку:

```
In [1]: items = [1, 2, 3]

In [2]: len(items)
Out[2]: 3
```

Функція sorted сортує елементи списку за зростанням і повертає новий список із відсортованими елементами:

```
In [1]: names = ['John', 'Michael', 'Antony']

In [2]: sorted(names)
Out[2]: ['Antony', 'John', 'Michael']
```

🕒 May 29, 2023

🕒 May 15, 2023

КОРИСНІ МЕТОДИ ДЛЯ РОБОТИ ЗІ СПИСКАМИ

Список є змінним типом даних, тому важливо звернути увагу на те, що більшість методів списку змінюють список на місці, не повертаючи нічого.

append

Метод `append` додає до кінця списку зазначений елемент:

```
In [18]: vlans = ['10', '20', '30', '100']

In [19]: vlans.append('300')

In [20]: vlans
Out[20]: ['10', '20', '30', '100', '300']
```

Метод `append` змінює список на місці та нічого не повертає. Якщо у скрипті треба додати елемент до списку, а потім вивести список `print`, треба робити це на різних рядках коду.

```
vlans = ['10', '20', '30', '100']
vlans.append('300')
print(vlans)
```

Якщо зробити `print(vlans.append('300'))`, результатом буде вивід `None`.

extend

Якщо потрібно об'єднати два списки, можна використовувати два способи: метод `extend` і операцію складання.

У цих способів є важлива відмінність - `extend` змінює список, до якого застосовано метод, а додавання повертає новий список, що складається з двох.

Метод `extend`:

```
In [21]: vlans = ['10', '20', '30', '100']

In [22]: vlans2 = ['300', '400', '500']

In [23]: vlans.extend(vlans2)

In [24]: vlans
Out[24]: ['10', '20', '30', '100', '300', '400', '500']
```

Додавання списків:

```
In [27]: vlans = ['10', '20', '30', '100']

In [28]: vlans2 = ['300', '400', '500']
```

```
In [29]: vlans + vlans2
Out[29]: ['10', '20', '30', '100', '300', '400', '500']
```

Зверніть увагу, що під час підсумовування списків в `ipython` з'явився рядок `Out`. Це означає, що результат підсумовування можна присвоїти у змінну:

```
In [30]: result = vlans + vlans2

In [31]: result
Out[31]: ['10', '20', '30', '100', '300', '400', '500']
```

`pop`

Метод `pop` видаляє елемент, який відповідає вказаному індексу і повертає цей елемент:

```
In [28]: vlans = ['10', '20', '30', '100']

In [29]: vlans.pop(-1)
Out[29]: '100'

In [30]: vlans
Out[30]: ['10', '20', '30']
```

Без зазначення індексу видаляється останній елемент списку.

`remove`

Метод `remove` видаляє вказаний елемент. `Remove` не повертає видалений елемент:

```
In [31]: vlans = ['10', '20', '30', '100']

In [32]: vlans.remove('20')

In [33]: vlans
Out[33]: ['10', '30', '100']
```

Якщо вказати неіснуючий елемент, виникне помилка:

```
In [34]: vlans.remove(500)
-----
ValueError      Traceback (most recent call last)
<ipython-input-32-f4ee38810cb7> in <module>()
----> 1 vlans.remove(500)

ValueError: list.remove(x): x not in list
```

`index`

Метод `index` повертає індекс, під яким знаходиться вказаний елемент

```
In [35]: vlans = ['10', '20', '30', '100']  
  
In [36]: vlans.index('30')  
Out[36]: 2
```

insert

Метод `insert` дозволяє вставити елемент на певне місце у списку:

```
In [37]: vlans = ['10', '20', '30', '100']  
  
In [38]: vlans.insert(1, '15')  
  
In [39]: vlans  
Out[39]: ['10', '15', '20', '30', '100']
```

sort

Метод `sort` сортує список на місці:

```
In [40]: vlans = [1, 50, 10, 15]  
  
In [41]: vlans.sort()  
  
In [42]: vlans  
Out[42]: [1, 10, 15, 50]
```

🕒 May 26, 2023

🕒 May 15, 2023

Словник (Dictionary)

СЛОВНИК (DICTIONARY)

Словники - це змінюваний упорядкований тип даних:

- дані у словнику - це пари ключ: значення
- доступ до значень здійснюється за ключом
- дані у словнику впорядковані за порядком додавання елементів
- словники можна змінювати, тобто елементи словника можна змінювати, додавати, видаляти
- ключ має бути об'єктом незмінного типу: число, рядок, кортеж
- значення може бути даними будь-якого типу

В інших мовах програмування тип даних подібний до словника може називатися асоціативний масив, хеш або хеш-таблиця.

Приклад словника:

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}
```

Можна записувати і так:

```
london = {
    'id': 1,
    'name': 'London',
    'it_vlan': 320,
    'user_vlan': 1010,
    'mngmt_vlan': 99,
    'to_name': None,
    'to_id': None,
    'port': 'G1/0/11'
}
```

Для того, щоб отримати значення зі словника, треба звернутися по ключу:

```
In [1]: london = {'name': 'London1', 'location': 'London Str'}

In [2]: london['name']
Out[2]: 'London1'

In [3]: london['location']
Out[3]: 'London Str'
```

Додати нову пару ключ-значення:

```
In [4]: london['vendor'] = 'Cisco'
```



```
In [5]: print(london)
{'vendor': 'Cisco', 'name': 'London1', 'location': 'London Str'}
```

У словнику як значення можна використовувати словник:

```
london_co = {
    'r1': {
        'hostname': 'london_r1',
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.1'
    },
    'r2': {
        'hostname': 'london_r2',
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.2'
    },
    'sw1': {
        'hostname': 'london_sw1',
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '3850',
        'ios': '3.6.XE',
        'ip': '10.255.0.101'
    }
}
```

Отримати значення із вкладеного словника можна так:

```
In [7]: london_co['r1']['ios']
Out[7]: '15.4'

In [8]: london_co['r1']['model']
Out[8]: '4451'

In [9]: london_co['sw1']['ip']
Out[9]: '10.255.0.101'
```

Функція `sorted` сортує ключі словаря по возрастанию и возвращает новый список с отсортированными ключами:

```
In [1]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [2]: sorted(london)
Out[2]: ['location', 'name', 'vendor']
```

🕒 May 23, 2023

🕒 May 15, 2023

КОРИСНІ МЕТОДИ ДЛЯ РОБОТИ ЗІ СЛОВНИКАМИ

`clear`

Метод `clear` дозволяє очистити словник - видалити всі елементи:

```
In [1]: london = {'name': 'London1', 'location': 'Globe Str'}

In [2]: london.clear()

In [3]: london
Out[3]: {}
```

`copy`

Метод `copy` створює копію словника:

```
In [10]: london = {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco'}

In [11]: london2 = london.copy()

In [12]: id(london)
Out[12]: 25524512

In [13]: id(london2)
Out[13]: 25563296

In [14]: london['vendor'] = 'Juniper'

In [15]: london2['vendor']
Out[15]: 'Cisco'
```

`get`

Якщо при зверненні до словника вказується ключ, якого немає у словнику, виникає помилка:

```
In [16]: london = {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco'}

In [17]: london['ios']
-----
KeyError                                Traceback (most recent call last)
<ipython-input-17-b4fae8480b21> in <module>()
----> 1 london['ios']

KeyError: 'ios'
```

Метод `get` замість помилки повертає `None`.

```
In [18]: london = {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco'}
```

```
In [19]: print(london.get('ios'))
None
```

Метод `get` також дозволяє вказувати інше значення замість `None` :

```
In [20]: print(london.get('ios', 'Ooops'))
Ooops
```

`keys`, `values`, `items`

Методи `keys`, `values`, `items` :

```
In [24]: london = {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco'}

In [25]: london.keys()
Out[25]: dict_keys(['name', 'location', 'vendor'])

In [26]: london.values()
Out[26]: dict_values(['London1', 'Globe Str', 'Cisco'])

In [27]: london.items()
Out[27]: dict_items([('name', 'London1'), ('location', 'Globe Str'), ('vendor', 'Cisco')])
```

Всі три методи повертають спеціальні об'єкти `view`, які відображають ключі, значення та пари ключ-значення словника відповідно.

Дуже важлива особливість `view` полягає в тому, що вони змінюються разом із зміною словника. І фактично вони лише дають спосіб подивитися на відповідні об'єкти, але не створюють їхньої копії.

На прикладі методу `keys`:

```
In [28]: london = {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco'}

In [29]: keys = london.keys()

In [30]: print(keys)
dict_keys(['name', 'location', 'vendor'])
```

Зараз змінна `keys` відповідає `view`, в якому три ключі: `name`, `location` і `vendor`. Якщо додати до словника ще одну пару ключ-значення, об'єкт `keys` також зміниться:

```
In [31]: london['ip'] = '10.1.1.1'

In [32]: keys
Out[32]: dict_keys(['name', 'location', 'vendor', 'ip'])
```

Якщо потрібно отримати звичайний список ключів, який не змінюватиметься зі змінами словника, достатньо конвертувати `view` в список:

```
In [33]: list_keys = list(london.keys())

In [34]: list_keys
Out[34]: ['name', 'location', 'vendor', 'ip']
```

del

Видалити ключ і значення:

```
In [35]: london = {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco'}

In [36]: del london['name']

In [37]: london
Out[37]: {'location': 'Globe Str', 'vendor': 'Cisco'}
```

update

Метод update дозволяє додавати до словника вміст іншого словника:

```
In [38]: r1 = {'name': 'London1', 'location': 'Globe Str'}

In [39]: r1.update({'vendor': 'Cisco', 'ios': '15.2'})

In [40]: r1
Out[40]: {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco', 'ios': '15.2'}
```

Аналогічним чином можна оновити значення:

```
In [41]: r1.update({'name': 'london-r1', 'ios': '15.4'})

In [42]: r1
Out[42]:
{'name': 'london-r1',
 'location': 'Globe Str',
 'vendor': 'Cisco',
 'ios': '15.4'}
```

setdefault

Метод setdefault шукає ключ, а якщо його немає, замість помилки створює ключ зі значенням `None`.

```
In [21]: london = {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco'}

In [22]: ios = london.setdefault('ios')

In [23]: print(ios)
None
```

```
In [24]: london
Out[24]: {'name': 'London1', 'location': 'Globe Str', 'vendor': 'Cisco', 'ios': None}
```

Якщо ключ є, `setdefault` повертає відповідне значення:

```
In [25]: london.setdefault('name')
Out[25]: 'London1'
```

Другий аргумент дозволяє вказати, яке значення має відповідати ключу:

```
In [26]: model = london.setdefault('model', 'Cisco3580')

In [27]: print(model)
Cisco3580

In [28]: london
Out[28]:
{'name': 'London1',
 'location': 'Globe Str',
 'vendor': 'Cisco',
 'ios': None,
 'model': 'Cisco3580'}
```

Метод `setdefault` в такому вигляді:

```
value = london.setdefault(key, "somevalue")
```

замінює таку конструкцію:

```
if key in london:
    value = london[key]
else:
    london[key] = "somevalue"
    value = london[key]
```

🕒 May 23, 2023

🕒 May 15, 2023

ВАРІАНТИ СТВОРЕННЯ СЛОВНИКА

Літерал

Словник можна створити за допомогою [літералу](#):

```
In [1]: r1 = {'model': '4451', 'ios': '15.4'}
```

dict

Конструктор dict дозволяє створювати словник кількома способами.

Якщо всі ключі словника - рядки, можна використовувати такий варіант створення словника:

```
In [2]: r1 = dict(model='4451', ios='15.4')
```

```
In [3]: r1
```

```
Out[3]: {'model': '4451', 'ios': '15.4'}
```

Другий варіант створення словника за допомогою dict:

```
In [4]: r1 = dict([('model', '4451'), ('ios', '15.4')])
```

```
In [5]: r1
```

```
Out[5]: {'model': '4451', 'ios': '15.4'}
```

dict.fromkeys

У ситуації, коли треба створити словник з відомими ключами, але поки що порожніми значеннями (або однаковими значеннями), дуже зручний метод fromkeys:

```
In [5]: d_keys = ['hostname', 'location', 'vendor', 'model', 'ios', 'ip']
```

```
In [6]: r1 = dict.fromkeys(d_keys)
```

```
In [7]: r1
```

```
Out[7]:
{'hostname': None,
 'location': None,
 'vendor': None,
 'model': None,
 'ios': None,
 'ip': None}
```

За замовчуванням метод fromkeys підставляє значення None. Але можна вказувати і свій варіант значення:

```
In [8]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']
```

```
In [9]: models_count = dict.fromkeys(router_models, 0)
```

```
In [10]: models_count
Out[10]: {'ISR2811': 0, 'ISR2911': 0, 'ISR2921': 0, 'ASR9002': 0}
```

Цей варіант створення словника підходить не для всіх випадків. Наприклад, при використанні змінного типу даних у значенні, буде створено посилання на один і той самий об'єкт:

```
In [10]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [11]: routers = dict.fromkeys(router_models, [])
...:

In [12]: routers
Out[12]: {'ISR2811': [], 'ISR2911': [], 'ISR2921': [], 'ASR9002': []}

In [13]: routers['ASR9002'].append('london_r1')

In [14]: routers
Out[14]:
{'ISR2811': ['london_r1'],
 'ISR2911': ['london_r1'],
 'ISR2921': ['london_r1'],
 'ASR9002': ['london_r1']}
```

У цьому випадку кожен ключ посилається на той самий список. Тому при додаванні значення до одного зі списків оновлюються й інші.

Для такого випадку найкраще підходить генератор словника. Дивись розділ (FIX REF)

🕒 May 29, 2023

🕒 May 15, 2023

Кортеж (Tuple)

Кортеж в Python це:

- послідовність елементів в дужках, які розділені між собою комою
- незмінний упорядкований тип даних

Грубо кажучи, кортеж – це список, який не можна змінити. Тобто в кортежі є лише права на читання. Це може бути захистом від випадкових змін.

Створити порожній кортеж:

```
In [1]: tuple1 = tuple()

In [2]: print(tuple1)
()
```

Кортеж з одного елемента (зверніть увагу на кому):

```
In [3]: tuple2 = ('password',)
```

Кортеж зі списку:

```
In [4]: list_keys = ['hostname', 'location', 'vendor', 'model', 'ios', 'ip']

In [5]: tuple_keys = tuple(list_keys)

In [6]: tuple_keys
Out[6]: ('hostname', 'location', 'vendor', 'model', 'ios', 'ip')
```

До елементів у кортежі можна звертатись, як і до елементів списку, за порядковим номером:

```
In [7]: tuple_keys[0]
Out[7]: 'hostname'
```

Оскільки кортеж незмінний, привласнити нове значення не можна:

```
In [8]: tuple_keys[1] = 'test'

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-9-1c7162cdefa3> in <module>()
----> 1 tuple_keys[1] = 'test'

TypeError: 'tuple' object does not support item assignment
```

Функція `sorted` сортує елементи кортежу за зростанням та повертає новий список із відсортованими елементами:

```
In [2]: tuple_keys = ('hostname', 'location', 'vendor', 'model', 'ios', 'ip')  
  
In [3]: sorted(tuple_keys)  
Out[3]: ['hostname', 'ios', 'ip', 'location', 'model', 'vendor']
```

🕒 May 23, 2023

🕒 May 15, 2023

Множина (Set)

Множина — це змінюваний неупорядкований набір унікальних хешованих об'єктів.

Множини використовуються для тестування членства, видалення дублікатів із послідовності та обчислення математичних операцій, таких як перетин, об'єднання, різниця та симетрична різниця.

Множина в Python - це послідовність елементів у фігурних дужках, які розділені між собою комою.

```
vlangs = {1, 2, 3, 4}
```

ВАРІАНТИ СТВОРЕННЯ МНОЖИН

Не можна створити порожню множину за допомогою [літералу](#), оскільки в такому випадку це буде не множина, а словник:

```
In [1]: set1 = {}

In [2]: type(set1)
Out[2]: dict
```

порожню множину можна створити таким чином:

```
In [3]: set2 = set()

In [4]: type(set2)
Out[4]: set
```

Створення множини із рядка:

```
In [5]: set('long long long long string')
Out[5]: {' ', 'g', 'i', 'l', 'n', 'o', 'r', 's', 't'}
```

Створення множини зі списку:

```
In [6]: set([10, 20, 30, 10, 10, 30])
Out[6]: {10, 20, 30}
```

КОРИСНІ МЕТОДИ ДЛЯ РОБОТИ З МНОЖИНАМИ

add

Метод add додає елемент у множину:

```
In [1]: set1 = {20, 10, 30, 40}

In [2]: set1.add(50)
```

```
In [3]: set1
Out[3]: {30, 20, 10, 50, 40}
```

discard

Метод `discard` видаляє елементи, не видаючи помилку, якщо елемента немає в множині:

```
In [3]: set1
Out[3]: {10, 20, 30, 40, 50}

In [4]: set1.discard(55)

In [5]: set1
Out[5]: {10, 20, 30, 40, 50}

In [6]: set1.discard(50)

In [7]: set1
Out[7]: {10, 20, 30, 40}
```

clear

Метод `clear` видаляє елементи множини:

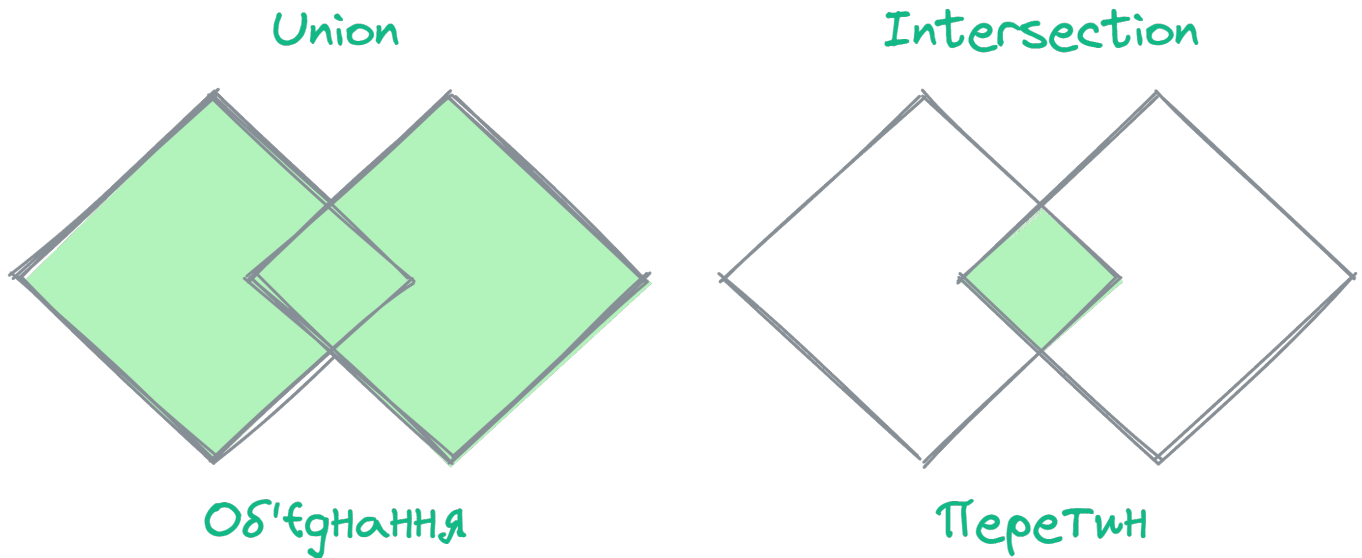
```
In [8]: set1 = {10, 20, 30, 40}

In [9]: set1.clear()

In [10]: set1
Out[10]: set()
```

ОПЕРАЦІЇ З МНОЖИНАМИ

Множини корисні тим, що з ними можна робити різні операції і знаходити об'єднання множин, перетин.



Об'єднання множин можна отримати за допомогою методу `union` або оператора `|`:

```
In [1]: vlans1 = {10, 20, 30, 50, 100}
In [2]: vlans2 = {100, 101, 102, 200}

In [3]: vlans1.union(vlans2)
Out[3]: {10, 20, 30, 50, 100, 101, 102, 200}

In [4]: vlans1 | vlans2
Out[4]: {10, 20, 30, 50, 100, 101, 102, 200}
```

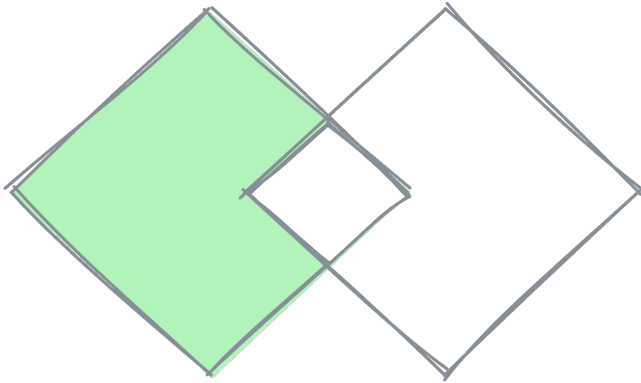
Перетин множин можна отримати за допомогою методу `intersection` або оператора `&`:

```
In [5]: vlans1 = {10, 20, 30, 50, 100}
In [6]: vlans2 = {100, 101, 102, 200}

In [7]: vlans1.intersection(vlans2)
Out[7]: {100}

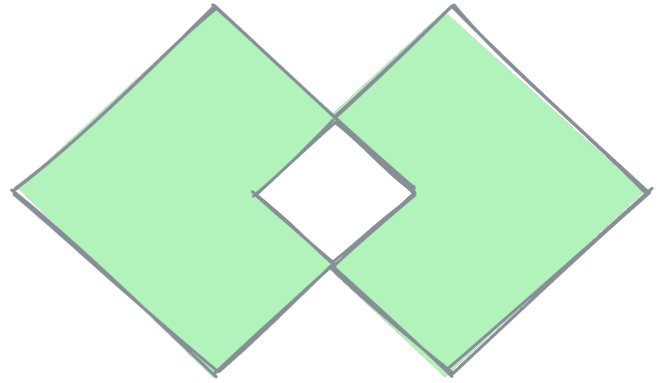
In [8]: vlans1 & vlans2
Out[8]: {100}
```

Difference



Різниця

Symmetric Difference



Симетрична різниця

🕒 June 24, 2023

🕒 May 15, 2023

Булеві значення

Булеві значення в Python - це дві константи True і False.

У Python істинними та хибними значеннями вважаються не тільки True та False.

- правда:
- будь-яке ненульове число
- будь-який непустий рядок
- будь-який непустий об'єкт
- хибність:
- 0
- None
- порожній рядок
- порожній об'єкт

Для перевірки булевого значення об'єкта можна скористатися bool:

```
In [2]: items = [1, 2, 3]

In [3]: empty_list = []

In [4]: bool(empty_list)
Out[4]: False

In [5]: bool(items)
Out[5]: True

In [6]: bool(0)
Out[6]: False

In [7]: bool(1)
Out[7]: True
```

🕒 May 23, 2023

🕒 May 15, 2023

Форматування рядків

При роботі з рядками часто виникають ситуації, коли в шаблон рядка треба підставити різні дані.

Це можна робити об'єднуючи частини рядка і дані, але в Python є більш зручний спосіб - форматування рядків.

Форматування рядків може допомогти, наприклад, у таких ситуаціях:

- необхідно підставити значення в рядок за певним шаблоном
- необхідно відформатувати вивід стовпцями
- треба конвертувати числа у двійковий формат

Існує кілька варіантів форматування рядків:

- з оператором `%` - старіший варіант
- метод `format` - відносно новий варіант
- f-рядки - новий варіант, який з'явився у Python 3.6

Так як для повноцінного пояснення f-рядків, треба показувати приклади з циклами та роботою з об'єктами, які ще не розглядалися, ця тема розглядається у розділі [FIX REF](#)

ФОРМАТУВАННЯ РЯДКІВ ІЗ МЕТОДОМ FORMAT

Приклад використання методу `format`:

```
In [1]: "interface FastEthernet0/{}".format('1')
Out[1]: 'interface FastEthernet0/1'
```

Спеціальний символ `{}` вказує на те, що сюди підставиться значення, яке передається методу `format`. При цьому кожна пара фігурних дужок позначає одне місце для підстановки значення.

Значення, що підставляються у фігурні дужки, можуть бути різного типу. Наприклад, це може бути рядок, число або список:

```
In [3]: print('{}'.format('10.1.1.1'))
10.1.1.1

In [4]: print('{}'.format(100))
100

In [5]: print('{}'.format([10, 1, 1, 1]))
[10, 1, 1, 1]
```

За допомогою форматування рядків можна виводити результат стовпцями. У форматуванні рядків можна вказувати, скільки символів виділено на дані. Якщо кількість символів у даних менша, ніж виділена кількість символів, відсутні символи заповнюються пробілами.

Наприклад, таким чином можна вивести дані стовпцями однакової ширини по 15 символів з вирівнюванням праворуч:

```
In [3]: vlan, mac, intf = ['100', 'aabb.cc80.7000', 'Gi0/1']

In [4]: print("{:>15} {:>15} {:>15}".format(vlan, mac, intf))
      100  aabb.cc80.7000      Gi0/1
```

Вирівнювання по лівій стороні:

```
In [5]: print("{:15} {:15} {:15}".format(vlan, mac, intf))
      100  aabb.cc80.7000  Gi0/1
```

Приклади вирівнювання

10	01ab.c5d0.70d0	Gi0/1
100	02ab.c5d0.70d0	Gi0/2
200	03ab.c5d0.70d0	Gi0/3
1000	04ab.c5d0.70d0	Gi0/4

`{:<5}`

`{:<20}`

`{:<15}`

10	01ab.c5d0.70d0	Gi0/1
100	02ab.c5d0.70d0	Gi0/2
200	03ab.c5d0.70d0	Gi0/3
1000	04ab.c5d0.70d0	Gi0/4

`{:>5}`

`{:>20}`

`{:>15}`

10	01ab.c5d0.70d0	Gi0/1
100	02ab.c5d0.70d0	Gi0/2
200	03ab.c5d0.70d0	Gi0/3
1000	04ab.c5d0.70d0	Gi0/4

`{:<5}`

`{:>20}`

`{:>15}`

Шаблон може бути багаторядковим:

```
ip_template = '''
IP address:
{}
'''

In [7]: print(ip_template.format('10.1.1.1'))

IP address:
10.1.1.1
```

За допомогою форматування рядків можна також впливати на відображення чисел.

Наприклад, можна вказати, скільки цифр після коми виводити:

```
In [9]: print("{:.3f}".format(10.0/3))
3.333
```

За допомогою форматування рядків можна конвертувати числа у двійковий формат:

```
In [11]: '{:b} {:b} {:b} {:b}'.format(192, 100, 1, 1)
Out[11]: '11000000 1100100 1 1'
```

При цьому, як і раніше, можна вказувати додаткові параметри, наприклад, ширину стовпця:

```
In [12]: '{:8b} {:8b} {:8b} {:8b}'.format(192, 100, 1, 1)
Out[12]: '11000000 1100100      1      1'
```

А також можна вказати, що треба доповнити числа нулями до вказаної ширини стовпця:

```
In [13]: '{:08b} {:08b} {:08b} {:08b}'.format(192, 100, 1, 1)
Out[13]: '11000000 01100100 00000001 00000001'
```

У фігурних дужках можна вказувати імена. Це дозволяє передавати аргументи в будь-якому порядку, а також робить шаблон зрозумілішим:

```
In [15]: '{ip}/{mask}'.format(mask=24, ip='10.1.1.1')
Out[15]: '10.1.1.1/24'
```

Ще одна корисна можливість форматування рядків – зазначення номера аргументу:

```
In [16]: '{1}/{0}'.format(24, '10.1.1.1')
Out[16]: '10.1.1.1/24'
```

За рахунок цього, наприклад, можна позбутися повторної передачі одних і тих же значень:

```
ip_template = '''
IP address:
{:<8} {:<8} {:<8} {:<8}
{:08b} {:08b} {:08b} {:08b}
'''

In [20]: print(ip_template.format(192, 100, 1, 1, 192, 100, 1, 1))

IP address:
192      100      1      1
11000000 01100100 00000001 00000001
```

У прикладі вище октети адреси доводиться передавати двічі - один відображення у десятковому форматі, а другий - для двійкового.

Вказавши індекси значень, які передаються методу `format`, можна позбутися дублювання:

```
ip_template = '''
IP address:
{0:<8} {1:<8} {2:<8} {3:<8}
{0:08b} {1:08b} {2:08b} {3:08b}
'''

In [22]: print(ip_template.format(192, 100, 1, 1))

IP address:
192      100      1      1
11000000 01100100 00000001 00000001
```

🕒 May 23, 2023

🕒 May 15, 2023

Перетворення типів

Python має кілька корисних вбудованих функцій, які дозволяють перетворити дані з одного типу в інший.

INT

`int` перетворює рядок на integer:

```
In [1]: int("10")  
Out[1]: 10
```

За допомогою функції `int` можна перетворити число в двійковому записі на десятковий (двійковий запис повинен бути у вигляді рядка):

```
In [2]: int("11111111", 2)  
Out[2]: 255
```

BIN

Перетворити десяткове число на двійковий формат можна за допомогою `bin`:

```
In [3]: bin(10)  
Out[3]: '0b1010'  
  
In [4]: bin(255)  
Out[4]: '0b11111111'
```

HEX

Аналогічна функція є і для перетворення на шістнадцятковий формат:

```
In [5]: hex(10)  
Out[5]: '0xa'  
  
In [6]: hex(255)  
Out[6]: '0xff'
```

LIST

Функція `list` перетворює аргумент на список:

```
In [7]: list("string")  
Out[7]: ['s', 't', 'r', 'i', 'n', 'g']  
  
In [8]: list({1, 2, 3})  
Out[8]: [1, 2, 3]  
  
In [9]: list((1, 2, 3, 4))  
Out[9]: [1, 2, 3, 4]
```

SET

Функція `set` перетворює аргумент на множину:

```
In [10]: set([1, 2, 3, 3, 4, 4, 4, 4])
Out[10]: {1, 2, 3, 4}

In [11]: set((1, 2, 3, 3, 4, 4, 4, 4))
Out[11]: {1, 2, 3, 4}

In [12]: set("string string")
Out[12]: {' ', 'g', 'i', 'n', 'r', 's', 't'}
```

Ця функція буде дуже корисною, коли потрібно отримати унікальні елементи в послідовності.

TUPLE

Функція `tuple` перетворює аргумент на кортеж:

```
In [13]: tuple([1, 2, 3, 4])
Out[13]: (1, 2, 3, 4)

In [14]: tuple({1, 2, 3, 4})
Out[14]: (1, 2, 3, 4)

In [15]: tuple("string")
Out[15]: ('s', 't', 'r', 'i', 'n', 'g')
```

Це може стати в нагоді в тому випадку, якщо потрібно отримати незмінний об'єкт.

STR

Функція `str` перетворює аргумент у рядок:

```
In [16]: str(10)
Out[16]: '10'
```

🕒 May 23, 2023

🕒 May 15, 2023

Перевірка типів даних

При перетворенні типів даних можуть виникнути такі помилки:

```
In [1]: int('a')
-----
ValueError                                Traceback (most recent call last)
<ipython-input-42-b3c3f4515dd4> in <module>()
----> 1 int('a')

ValueError: invalid literal for int() with base 10: 'a'
```

Помилка логічна - ми намагаємося перетворити на десятковий формат рядок "a".

Така помилка може виникнути, наприклад, коли потрібно пройти по списку рядків і перетворити на число кожен рядок. Щоб уникнути її, було б добре мати можливість перевірити, із чим ми працюємо. У Python є ряд методів і функцій, які дозволяють це перевірити.

ISDIGIT

Наприклад, щоб перевірити, чи складається рядок з одних цифр, можна використовувати метод `isdigit`:

```
In [2]: "a".isdigit()
Out[2]: False

In [3]: "a10".isdigit()
Out[3]: False

In [4]: "10".isdigit()
Out[4]: True
```

ISALPHA

Метод `isalpha` дозволяє перевірити, чи складається рядок з одних літер:

```
In [7]: "a".isalpha()
Out[7]: True

In [8]: "a100".isalpha()
Out[8]: False

In [9]: "a-- ".isalpha()
Out[9]: False

In [10]: "a ".isalpha()
Out[10]: False
```

ISALNUM

Метод `isalnum` дозволяє перевірити, чи складається рядок із букв або цифр:

```
In [11]: "a".isalnum()
Out[11]: True

In [12]: "a10".isalnum()
Out[12]: True
```

TYPE

Іноді, залежно від результату, бібліотека чи функція можуть повертати різні типи об'єктів. Наприклад, якщо об'єкт один, повертається рядок, якщо кілька, то повертається кортеж.

Відповідно нам треба побудувати хід програми по-різному, залежно від того, чи було повернуто рядок чи кортеж.

У цьому може допомогти функція `type`:

```
In [13]: type("string")
Out[13]: str

In [14]: type("string") == str
Out[14]: True
```

Аналогічно з кортежем (та іншими типами даних):

```
In [15]: type((1, 2, 3))
Out[15]: tuple

In [16]: type((1, 2, 3)) == tuple
Out[16]: True

In [17]: type((1, 2, 3)) == list
Out[17]: False
```

🕒 May 23, 2023

🕒 May 15, 2023

Виклик методів ланцюжком

Часто з даними треба виконати кілька операцій, наприклад:

```
In [1]: line = "switchport trunk allowed vlan 10,20,30"

In [2]: words = line.split()

In [3]: words
Out[3]: ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30']

In [4]: vlans_str = words[-1]

In [5]: vlans_str
Out[5]: '10,20,30'

In [6]: vlans = vlans_str.split(",")

In [7]: vlans
Out[7]: ['10', '20', '30']
```

Або у скрипті:

```
line = "switchport trunk allowed vlan 10,20,30"
words = line.split()
vlans_str = words[-1]
vlans = vlans_str.split(",")
print(vlans)
```

У цьому випадку змінні використовуються для зберігання проміжного результату та наступні методи/дії виконуються вже зі змінною. Це цілком нормальний варіант коду, особливо спочатку, коли важко сприймати складніші висловлювання.

Однак у Python часто зустрічаються вирази, в яких дії або методи застосовуються один за одним в одному виразі. Наприклад, попередній код можна записати так:

```
line = "switchport trunk allowed vlan 10,20,30"
vlans = line.split()[-1].split(",")
print(vlans)
```

Так як тут немає виразів у дужках, які б вказували на пріоритет виконання, все виконується зліва направо. Спочатку виконується `line.split()` - отримуємо список, потім до отриманого списку застосовується `[-1]` - отримуємо останній елемент списку, рядок `"10,20,30"`. До цього рядка застосовується метод `split(",")` і в результаті одержуємо список `['10', '20', '30']`.

Головний нюанс при написанні таких ланцюжків попередній метод/дія має повертати те, що чекає наступний метод/дія. І обов'язково, щоб щось поверталось, інакше буде помилка.

🕒 May 23, 2023

🕒 May 15, 2023

Основи сортування даних

При сортуванні даних типу списку списків або списку кортежів, `sorted` сортує за першим елементом вкладених списків (кортежів), а якщо перший елемент однаковий, за другим:

```
In [1]: data = [[1, 100, 1000], [2, 2, 2], [1, 2, 3], [4, 100, 3]]

In [2]: sorted(data)
Out[2]: [[1, 2, 3], [1, 100, 1000], [2, 2, 2], [4, 100, 3]]
```

Якщо сортування робиться для списку чисел, які записані як рядки, сортування буде лексикографічним, не натуральним і порядок буде таким:

```
In [7]: vlans = ['1', '30', '11', '3', '10', '20', '30', '100']

In [8]: sorted(vlans)
Out[8]: ['1', '10', '100', '11', '20', '3', '30', '30']
```

Щоб сортування було «правильним», треба перетворити влани на числа.

Ця ж проблема проявляється, наприклад, з IP-адресами:

```
In [2]: ip_list = ["10.1.1.1", "10.1.10.1", "10.1.2.1", "10.1.11.1"]

In [3]: sorted(ip_list)
Out[3]: ['10.1.1.1', '10.1.10.1', '10.1.11.1', '10.1.2.1']
```

Як вирішити проблему з сортуванням IP-адрес див. у розділі [10. Корисні функції](#).

🕒 May 23, 2023

🕒 May 15, 2023

Додаткова інформація

Документація:

- [Strings. String Methods](#)
- [Lists basics. More on lists](#)
- [Tuples. More on tuples](#)
- [Sets basics. More on sets](#)
- [Dict basics. More on dicts](#)
- [Common Sequence Operations](#)

Форматування рядків:

- [Приклади використання форматування рядків](#)
- [Документація з форматування рядків \(en\), переклад документації](#)
- [Python 3's f-Strings: An Improved String Formatting Syntax \(Guide\)](#)
- [Python String Formatting Best Practices](#)

🕒 May 23, 2023

🕒 May 15, 2023

5. Створення базових скриптів

5. Створення базових скриптів

 Цей розділ в форматі відео

Якщо говорити загалом, то скрипт – це звичайний файл. У цьому файлі зберігається послідовність команд, які потрібно виконати.

Почнемо із базового скрипту. Виведемо на стандартний потік виведення кілька рядків.

Для цього потрібно створити файл `access_template.py` з таким вмістом:

```
access_template = ['switchport mode access',
                   'switchport access vlan {}',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

print('\n'.join(access_template).format(5))
```

Спочатку елементи списку об'єднуються в рядок, розділений символом `\n`, а в рядок підставляється номер VLAN, використовуючи форматування рядків.

Після цього потрібно зберегти файл і перейти до командного рядка.

Так виглядає виконання скрипту:

```
$ python access_template.py
switchport mode access
switchport access vlan 5
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

Ставити розширення `.py` у файлу не обов'язково, але це бажано робити.

У книзі всі скрипти, які створюватимуться, використовують розширення `.py`. Можна сказати, що це гарний тон – створювати скрипти Python з таким розширенням.

 July 4, 2023

 May 24, 2023

Виконуваний файл

Для того, щоб файл був виконуваним, і не потрібно було щоразу писати `python` перед викликом файлу, потрібно:

- зробити файл виконуваним (для Linux)
- **у першому рядку файлу** має бути рядок `#!/usr/bin/env python` або `#!/usr/bin/env python3`, залежно від того, яка версія Python використовується за умовчанням

Приклад файлу `access_template_exec.py`:

```
#!/usr/bin/env python3

access_template = ['switchport mode access',
                   'switchport access vlan {}',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

print('\n'.join(access_template).format(5))
```

Після цього:

```
chmod +x access_template_exec.py
```

Тепер можна викликати файл у такий спосіб:

```
$ ./access_template_exec.py
```

🕒 May 24, 2023

🕒 May 24, 2023

Передача аргументів скрипту (argv)

Найчастіше скрипт вирішує якесь загальне завдання. Наприклад, скрипт обробляє файл конфігурації. І щоб обробити інший файл конфігурації, треба вказувати його якимось чином у коді. Звичайно, у такому разі не хочеться щоразу руками у скрипті правити назву файлу.

Набагато краще передавати ім'я файлу як аргумент скрипту і потім використовувати вже вказаний файл.

Модуль `sys` дає змогу працювати з аргументами скрипта за допомогою `argv`.

Приклад `access_template_argv.py`:

```
from sys import argv

interface = argv[1]
vlan = argv[2]

access_template = ['switchport mode access',
                   'switchport access vlan {}'.format(vlan),
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

print('interface {}'.format(interface))
print('\n'.join(access_template).format(vlan))
```

Перевірка роботи скрипту:

```
$ python access_template_argv.py Gi0/7 4
interface Gi0/7
switchport mode access
switchport access vlan 4
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

Аргументи, які були передані скрипту, підставляються як значення до шаблону.

Виклик скрипту

```
python access_template_argv.py Gi0/7 4
```

```
["access_template_argv.py", "Gi0/7", "4"]
```

список sys.argv

Тут треба пояснити кілька моментів:

- argv – це список
- всі аргументи знаходяться у списку у вигляді рядків
- argv містить не лише аргументи, які передали скрипту, але й назву самого скрипту

У цьому випадку у списку argv знаходяться такі елементи:

```
['access_template_argv.py', 'Gi0/7', '4']
```

Спочатку йде ім'я самого скрипта, потім аргументи в тому ж порядку в якому вони передавалися.

🕒 May 31, 2023

🕒 May 24, 2023

Введення інформації користувачем

Іноді потрібно отримати інформацію від користувача, наприклад, запросити пароль.

Для отримання інформації від користувача використовується функція `input` :

```
In [1]: print(input('Routing protocol: '))
Routing protocol: OSPF
OSPF
```

В цьому разі інформація відразу виводиться користувачеві, але крім цього інформація, яку ввів користувач, може бути збережена в якусь змінну і може використовуватися далі в скрипті.

```
In [2]: protocol = input('Routing protocol: ')
Routing protocol: OSPF

In [3]: print(protocol)
OSPF
```

У дужках зазвичай пишеться якийсь запит, який уточнює яку інформацію потрібно ввести.

Запит інформації зі скрипту (файл `access_template_input.py`):

```
interface = input('Enter interface type and number: ')
vlan = input('Enter VLAN number: ')

access_template = ['switchport mode access',
                   'switchport access vlan {}'.format(vlan),
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

print('\n' + '-' * 30)
print('interface {}'.format(interface))
print('\n'.join(access_template).format(vlan))
```

У перших двох рядках запитується інформація користувача.

Рядок `print('\n' + '-' * 30)` використовується для того, щоб візуально відокремити запит інформації від виводу.

Виконання скрипту:

```
$ python access_template_input.py
Enter interface type and number: Gi0/3
Enter VLAN number: 55

-----
interface Gi0/3
switchport mode access
```



```
switchport access vlan 55  
switchport nonegotiate  
spanning-tree portfast  
spanning-tree bpduguard enable
```

🕒 May 24, 2023

🕒 May 24, 2023

6. Контроль перебігу програми

6. Контроль перебігу програми

 [Цей розділ в форматі відео](#)

Досі весь код виконувався послідовно - усі рядки скрипту виконувались в тому порядку, в якому вони записані у файлі. У цьому розділі розглядаються можливості Python в керуванні ходом програми:

- відгалуження в ході програми за допомогою конструкції `if/elif/else`
- повторення дій у циклі за допомогою конструкцій `for` та `while`
- обробка помилок за допомогою конструкції `try/except`

 July 10, 2023

 May 24, 2023

if/elif/else

Конструкція `if/elif/else` дозволяє робити розгалуження під час програми. Програма йде у гілку під час виконання певної умови.

У цій конструкції тільки `if` є обов'язковим, `elif` та `else` опціональні:

- Перевірка `if` завжди йде першою.
- Після оператора `if` має бути якась умова: якщо ця умова виконується, дії в блоці `if` виконуються.
- За допомогою `elif` можна зробити кілька розгалужень, тобто перевіряти вхідні дані на різні умови.
- Блоків `elif` може бути багато.
- Блок `else` виконується у тому випадку, якщо жодна з умов `if` або `elif` не була істинною.

Приклад конструкції:

```
a = 9

if a > 10:
    print("a більше 10")
else:
    print("a менше чи дорівнює 10")

print("Кінець")
```

flowchart TD

```
a["a = ..."] --> c{if a > 10};
c -->|True| b["a більше 10"];
c -->|False| e["a менше чи дорівнює 10"];
b --> q["Кінець"];
e --> q;
```

Приклад конструкції:

```
a = 9

if a == 10:
    print("a дорівнює 10")
elif a < 10:
    print("a менше 10")
else:
    print("a більше 10")
```

Вивід:

```
a менше 10
```

УМОВИ

Конструкція `if` використовує умови: після `if` і `elif` завжди пишеться логічний вираз. Блоки `if/elif` виконуються тільки коли логічний вираз істинний, тому перше з чим треба розібратися - це те, що є істинним, а що хибним в Python.

TRUE TA FALSE

У Python, крім очевидних значень `True` і `False`, усі інші об'єкти також мають хибне або істинне значення:

- істинне:
 - будь-яке ненульове число
 - будь-який непорожній рядок
 - будь-який непорожній об'єкт
- хибне:
 - 0
 - `None`
 - порожній рядок
 - порожній об'єкт (список, словник, множина тощо)

Наприклад, оскільки порожній список є хибним, ви можете перевірити, чи порожній список, так:

```
In [12]: list_to_test = [1, 2, 3]

In [13]: if list_to_test:
.....:     print("У списку є об'єкти")
.....:
У списку є об'єкти
```

Той самий результат можна отримати трохи інакше:

```
In [14]: if len(list_to_test) != 0:
.....:     print("У списку є об'єкти")
.....:
У списку є об'єкти
```

ОПЕРАТОРИ ПОРІВНЯННЯ

Оператори порівняння, які можна використовувати в умовах:

```
In [3]: 5 > 6
Out[3]: False

In [4]: 5 > 2
Out[4]: True
```

```

In [5]: 5 < 2
Out[5]: False

In [6]: 5 == 2
Out[6]: False

In [7]: 5 == 5
Out[7]: True

In [8]: 5 >= 5
Out[8]: True

In [9]: 5 <= 10
Out[9]: True

In [10]: 8 != 10
Out[10]: True

```

Приклад використання операторів порівняння:

```

In [1]: a = 9

In [2]: if a == 10:
...:     print("a дорівнює 10")
...: elif a < 10:
...:     print("a менше 10")
...: else:
...:     print("a більше 10")
...:
a менше 10

```

ОПЕРАТОР IN

Оператор `in` дозволяє виконувати перевірку на наявність елемента у послідовності (наприклад, елемента у списку або підрядка в рядку):

```

In [8]: "Fast" in "FastEthernet"
Out[8]: True

In [9]: "Gigabit" in "FastEthernet"
Out[9]: False

In [10]: vlan = [10, 20, 30, 40]

In [11]: 10 in vlan
Out[11]: True

In [12]: 50 in vlan
Out[12]: False

```

При використанні зі словниками `in` перевіряє ключі словника:

```

r1 = {
    'ios': '15.4',
    'ip': '10.255.0.1',
    'hostname': 'london_r1',
    'location': '21 New Globe Walk',
    'model': '4451',
    'vendor': 'Cisco'
}

In [16]: 'ios' in r1
Out[16]: True

In [17]: '4451' in r1
Out[17]: False

```

ОПЕРАТОРИ AND, OR, NOT

В умовах можуть також використовуватись логічні оператори `and`, `or`, `not`:

```

r1 = {
    'ios': '15.4',
    'ip': '10.255.0.1',
    'hostname': 'london_r1',
    'location': '21 New Globe Walk',
    'model': '4451',
    'vendor': 'Cisco'
}

In [18]: vlan = [10, 20, 30, 40]

In [19]: 'ios' in r1 and 10 in vlan
Out[19]: True

In [20]: '4451' in r1 and 10 in vlan
Out[20]: False

In [21]: '4451' in r1 or 10 in vlan
Out[21]: True

In [22]: not '4451' in r1
Out[22]: True

In [23]: '4451' not in r1
Out[23]: True

```

ОПЕРАТОР AND

У Python оператор `and` повертає не булеве значення, а значення одного з операндів.

Якщо обидва операнди істинні, результатом виразу буде останнє значення:

```

In [24]: 'string1' and 'string2'
Out[24]: 'string2'

```

```
In [25]: 'string1' and 'string2' and 'string3'
Out[25]: 'string3'
```

Якщо один із операндів є хибним, результатом виразу буде перше хибне значення:

```
In [26]: '' and 'string1'
Out[26]: ''

In [27]: '' and [] and 'string1'
Out[27]: ''
```

ОПЕРАТОР OR

Оператор `or`, як і оператор `and`, повертає значення одного з операндів.

Під час обчислення операндів повертається перший істинний операнд:

```
In [28]: '' or 'string1'
Out[28]: 'string1'

In [29]: '' or [] or 'string1'
Out[29]: 'string1'

In [30]: 'string1' or 'string2'
Out[30]: 'string1'
```

Якщо всі значення хибні, повертається останнє значення:

```
In [31]: '' or [] or {}
Out[31]: {}
```

Важливою особливістю оператора `or` є те, що операнди, які стоять після істинного, не оцінюються.

Приклад використання конструкції `if/elif/else`

 [Цей та інші приклади використання if/elif/else в форматі відео](#)

Приклад коду, який перевіряє довжину пароля та чи містить пароль ім'я користувача:

```
username = input("Введіть ім'я користувача: ")
password = input("Введіть пароль: ")

if len(password) < 8:
    print("Пароль надто короткий")
elif username in password:
    print("Пароль містить ім'я користувача")
else:
    print(f"Пароль для користувача {username} встановлено")
```

Перевірка скрипту:

```
$ python check_password.py
Введіть ім'я користувача: nata
Введіть пароль: nata1234
Пароль містить ім'я користувача

$ python check_password.py
Введіть ім'я користувача: nata
Введіть пароль: 123nata123
Пароль містить ім'я користувача

$ python check_password.py
Введіть ім'я користувача: nata
Введіть пароль: 1234
Пароль надто короткий

$ python check_password.py
Введіть ім'я користувача: nata
Введіть пароль: 123456789
Пароль для користувача nata встановлено
```

 July 10, 2023

 May 24, 2023

for

Дуже часто ту саму дію необхідно виконати для набору даних одного типу. Наприклад, перетворити всі рядки в списку на верхній регістр. Для виконання цих дій Python використовує цикл `for`.

Цикл `for` перебирає елементи вказаної послідовності один за одним і виконує дії, вказані в блоці `for` для кожного елемента.

Приклади послідовностей елементів, по яким цикл `for` може виконувати ітерацію:

- рядок
- список
- словник
- `range`
- будь-який ітерований об'єкт

Приклад перетворення рядків у списку на верхній регістр:

```
In [10]: words = ['list', 'dict', 'tuple']

In [11]: upper_words = []

In [12]: for word in words:
...:     upper_words.append(word.upper())
...:

In [13]: upper_words
Out[13]: ['LIST', 'DICT', 'TUPLE']
```

Вираз `for word in words:` означає "для кожного слова в списку слів `words` виконати дії в блоці `for`". У цьому випадку `word` - це ім'я змінної, яка посилається на різні значення на кожній ітерації циклу.

Note

Проект [pythontutor](#) може дуже допомогти в розумінні циклів. Візуалізація коду дозволяє побачити, що відбувається на кожному етапі виконання коду, що особливо корисно з циклами. Ви можете завантажити свій код на сайт [pythontutor](#), але для прикладу перейдіть за цим посиланням, щоб переглянути [приклад вище](#).

Цикл `for` може працювати з будь-якою послідовністю елементів.

Під час роботи з рядками цикл `for` виконує ітерацію по символах рядка, наприклад:

```
In [1]: for letter in 'Test string':
...:     print(letter)
...:

T
e
s
```

```
t
s
t
r
i
n
g
```

Цикл використовує змінну з іменем letter. Хоча ім'я може бути будь-яким, найкраще, коли ім'я говорить вам, через які об'єкти виконуються ітерації.

Іноді потрібно використовувати послідовність чисел у циклі. У цьому випадку можна використовувати функцію range. Приклад циклу з функцією range:

```
In [2]: for i in range(10):
...:     print(f'interface FastEthernet0/{i}')
...:
interface FastEthernet0/0
interface FastEthernet0/1
interface FastEthernet0/2
interface FastEthernet0/3
interface FastEthernet0/4
interface FastEthernet0/5
interface FastEthernet0/6
interface FastEthernet0/7
interface FastEthernet0/8
interface FastEthernet0/9
```

Функція range генерує числа в діапазоні від нуля до вказаного числа (у цьому прикладі до 10), не враховуючи його.

У цьому прикладі перебирається список VLAN, тому змінну можна назвати vlan:

```
In [3]: vlans = [10, 20, 30, 40, 100]
In [4]: for vlan in vlans:
...:     print(f'vlan {vlan}')
...:     print(f' name VLAN_{vlan}')
...:
vlan 10
 name VLAN_10
vlan 20
 name VLAN_20
vlan 30
 name VLAN_30
vlan 40
 name VLAN_40
vlan 100
 name VLAN_100
```

Коли цикл for проходить через словник, він виконує ітерацію за ключами:

```
r1 = {
    'ios': '15.4',
    'ip': '10.255.0.1',
    'hostname': 'london_r1',
    'location': '21 New Globe Walk',
    'model': '4451',
    'vendor': 'Cisco',
}
```

```
In [35]: for k in r1:
...:     print(k)
...:
ios
ip
hostname
location
model
vendor
```

Якщо потрібно вивести пари ключ-значення в циклі, можна зробити так:

```
In [36]: for key in r1:
...:     print(key + ' => ' + r1[key])
...:
ios => 15.4
ip => 10.255.0.1
hostname => london_r1
location => 21 New Globe Walk
model => 4451
vendor => Cisco
```

Або скористатися методом `items`, який дозволяє отримати пару ключ-значення:

```
In [37]: for key, value in r1.items():
...:     print(key + ' => ' + value)
...:
ios => 15.4
ip => 10.255.0.1
hostname => london_r1
location => 21 New Globe Walk
model => 4451
vendor => Cisco
```

🕒 July 5, 2023

🕒 May 24, 2023

Вкладені for

Цикли `for` можуть бути вкладені один в одного. У цьому прикладі список команд містить команди, які потрібно виконати для кожного з інтерфейсів у списку `interfaces`:

```
commands = [
    'switchport mode access',
    'spanning-tree portfast',
    'spanning-tree bpduguard enable',
]
interfaces = ['0/1', '0/3', '0/4', '0/7', '0/9', '0/10', '0/11']

In [9]: for intf in interfaces:
...:     print(f'interface FastEthernet {intf}')
...:     for command in commands:
...:         print(f' {command}')
...:
interface FastEthernet 0/1
switchport mode access
spanning-tree portfast
spanning-tree bpduguard enable
interface FastEthernet 0/3
switchport mode access
spanning-tree portfast
spanning-tree bpduguard enable
interface FastEthernet 0/4
switchport mode access
spanning-tree portfast
spanning-tree bpduguard enable
...
```

🕒 July 5, 2023

🕒 May 24, 2023

Комбінування for і if

Приклад поєднання for і if.

```
access_template = ['switchport mode access',
                  'switchport access vlan',
                  'spanning-tree portfast',
                  'spanning-tree bpduguard enable']

access = {'0/12': 10, '0/14': 11, '0/16': 17, '0/17': 150}

for intf, vlan in access.items():
    print('interface FastEthernet {}'.format(intf))
    for command in access_template:
        if command.endswith('access vlan'):
            print(' {} {}'.format(command, vlan))
        else:
            print(' {}'.format(command))
```

Коментарі до коду:

- Перший цикл for виконує ітерацію по ключам і значенням у словнику access
- Поточний ключ у цій точці циклу зберігається у змінній intf
- Поточне значення на цьому етапі циклу зберігається у змінній vlan
- Відображається print рядок `interface FastEthernet` з доданим до нього номером інтерфейсу
- Другий цикл for повторює команди зі списку `access_template`
- Оскільки потрібно додати номер VLAN до команди `switchport access vlan`:
- всередині другого циклу for перевіряються команди
- якщо команда закінчується на `access vlan`, виводиться команда, і до неї додається номер VLAN
- у всіх інших випадках команда просто відображається print

Результат виконання скрипта:

```
$ python generate_access_port_config.py
interface FastEthernet0/12
switchport mode access
switchport access vlan 10
spanning-tree portfast
spanning-tree bpduguard enable
interface FastEthernet0/14
switchport mode access
switchport access vlan 11
spanning-tree portfast
spanning-tree bpduguard enable
interface FastEthernet0/16
switchport mode access
switchport access vlan 17
```

```
spanning-tree portfast
spanning-tree bpduguard enable
interface FastEthernet0/17
switchport mode access
switchport access vlan 150
spanning-tree portfast
spanning-tree bpduguard enable
```

🕒 July 5, 2023

🕒 May 24, 2023

while

Цикл `while` - це ще один різновид циклу в Python.

У циклі `while`, як і в `if`, треба писати умову. Якщо умова є істинною, виконуються дії всередині блоку `while`. Але, на відміну від `if`, після виконання коду в блоці, `while` повертається на початок циклу.

Під час використання циклів `while` слід звертати увагу на те, чи буде досягнуто такий стан, за якого умова циклу буде хибною.

Розглянемо приклад:

```
In [1]: a = 5

In [2]: while a > 0:
...:     print(a)
...:     a -= 1
...:
5
4
3
2
1
```

Спочатку створюється змінна зі значенням 5.

Потім, у циклі `while` зазначено умову `a > 0`. Тобто, поки значення `a` більше 0, виконуватимуться дії у тілі циклу. У цьому випадку буде відображено значення змінної `a`.

Крім того, в тілі циклу при кожному проходженні значення `a` стає на одиницю менше.

Note

Запис `a -= 1` може бути трохи незвичним. Python дозволяє використовувати цей формат замість `a = a - 1`. Подібним чином можна писати: `a += 1`, `a *= 2`, `a /= 2`.

Оскільки значення `a` зменшується, цикл не буде нескінченним, і в якийсь момент вираз `a > 0` стане хибним.

Наступний приклад базується на прикладі про пароль з розділу про [конструкцію if](#). У тому прикладі доводилося повторно запускати скрипт, якщо пароль не відповідав вимогам.

За допомогою циклу `while` можна зробити так, що скрипт сам запитуватиме пароль наново, якщо він не відповідає вимогам.

```
username = input("Введіть ім'я користувача: ")
password = input("Введіть пароль: ")

password_correct = False
```

```
while not password_correct:
    if len(password) < 8:
        print("Пароль надто короткий\n")
        password = input("Введіть пароль ще раз: ")
    elif username in password:
        print("Пароль містить ім'я користувача\n")
        password = input("Введіть пароль ще раз: ")
    else:
        print(f"Пароль для користувача {username} встановлено")
        password_correct = True
```

Цикл `while` повертає скрипт знову на початок перевірок, дозволяє знову набрати пароль, але при цьому не вимагає перезапуску самого скрипта.

Тепер скрипт відпрацьовує так:

```
$ python check_password_with_while.py
Введіть ім'я користувача: nata
Введіть пароль: nata
Пароль надто короткий

Введіть пароль ще раз: natanata
Пароль містить ім'я користувача

Введіть пароль ще раз: 123345345345
Пароль для користувача nata встановлено
```

🕒 July 6, 2023

🕒 May 24, 2023

break, continue, pass

Python має кілька операторів, які дозволяють змінювати поведінку циклів за замовчуванням.

ОПЕРАТОР BREAK

Оператор break дозволяє достроково перервати цикл:

- break перериває поточний цикл і продовжує виконання наступних виразів
- якщо використовується кілька вкладених циклів, break перериває внутрішній цикл і продовжує виконувати вирази, що йдуть за блоком
- break може використовуватися в циклах for та while

Приклад із циклом for:

```
In [1]: for num in range(10):
...:     if num < 7:
...:         print(num)
...:     else:
...:         break
...:
0
1
2
3
4
5
6
```

Приклад із циклом while:

```
In [2]: i = 0

In [3]: while i < 10:
...:     if i == 5:
...:         break
...:     else:
...:         print(i)
...:         i += 1
...:
0
1
2
3
4
```

Використання break у прикладі із запитом пароля (файл `check_password_with_while_break.py`):

```

username = input("Введіть ім'я користувача: ")
password = input("Введіть пароль: ")

while True:
    if len(password) < 8:
        print("Пароль надто короткий\n")
    elif username in password:
        print("Пароль містить ім'я користувача\n")
    else:
        print("Пароль для користувача {} встановлено".format(username))
        # завершує цикл while
        break
    password = input("Введіть пароль ще раз: ")

```

Тепер можна не повторювати рядок `password = input("Введіть пароль ще раз: ")` у кожному відгалуженні, достатньо перенести його в кінець циклу.

І як тільки буде введено правильний пароль, `break` виведе програму з циклу `while`.

ОПЕРАТОР CONTINUE

Оператор `continue` повертає керування на початок циклу. Тобто, `continue` дозволяє "перестрибнути" вирази, що залишилися в циклі і перейти до наступної ітерації.

Приклад із циклом `for`:

```

In [4]: for num in range(5):
...:     if num == 3:
...:         continue
...:     else:
...:         print(num)
...:
0
1
2
4

```

Приклад із циклом `while`:

```

In [5]: i = 0

In [6]: while i < 6:
...:     i += 1
...:     if i == 3:
...:         print("Пропускаємо 3")
...:         continue
...:         print("Це ніхто не побачить")
...:     else:
...:         print("Поточне значення: ", i)
...:
Поточне значення: 1
Поточне значення: 2
Пропускаємо 3

```

```
Поточне значення: 4
Поточне значення: 5
Поточне значення: 6
```

Використання `continue` у прикладі із запитом пароля (файл `check_password_with_while_continue.py`):

```
username = input("Введіть ім'я користувача: ")
password = input("Введіть пароль: ")

password_correct = False

while not password_correct:
    if len(password) < 8:
        print("Пароль надто короткий\n")
    elif username in password:
        print("Пароль містить ім'я користувача\n")
    else:
        print("Пароль для користувача {} встановлено".format(username))
        password_correct = True
        continue
    password = input("Введіть пароль ще раз: ")
```

Тут вихід із циклу виконуються за допомогою перевірки змінної-прапорця `password_correct`. Коли було введено правильний пароль, прапор виставляється рівним `True`, і з допомогою `continue` виконується перехід початку циклу, перескочивши останній рядок із запитом пароля.

Результат виконання буде таким:

```
$ python check_password_with_while_continue.py
Введіть ім'я користувача: nata
Введіть пароль: nata12
Пароль надто короткий

Введіть пароль ще раз: nataksdjflsdjf
Пароль містить ім'я користувача

Введіть пароль ще раз: asdfsujlhdfaskjdfh
Пароль для користувача nata встановлено
```

ОПЕРАТОР PASS

Оператор `pass` нічого не робить. Фактично це така заглушка для блоків коду.

Наприклад, `pass` може допомогти ситуації, коли потрібно прописати структуру скрипта. Його можна ставити у циклах, функціях, класах. І це не впливатиме на виконання коду.

Приклад використання `pass`:

```
In [6]: for num in range(5):
.....:     if num < 3:
```

```
.....: pass
.....: else:
.....:     print(num)
.....:
3
4
```

🕒 July 10, 2023

🕒 May 24, 2023

for/else, while/else

У циклах for і while опціонально можна використовувати блок else.

FOR/ELSE

У циклі for:

- блок else виконується у разі, якщо цикл завершив ітерацію списку
- else не виконується, якщо в циклі було виконано break

Приклад циклу for з else (блок else виконується після завершення циклу for):

```
In [1]: for num in range(5):
....:     print(num)
....:     else:
....:         print("Числа закінчилися")
....:
0
1
2
3
4
Числа закінчилися
```

Приклад циклу for з else і break в циклі (через break блок else не виконується):

```
In [2]: for num in range(5):
....:     if num == 3:
....:         break
....:     else:
....:         print(num)
....: else:
....:     print("Числа закінчилися")
....:
0
1
2
```

Приклад циклу for з else та continue у циклі (continue не впливає на блок else):

```
In [3]: for num in range(5):
....:     if num == 3:
....:         continue
....:     else:
....:         print(num)
....: else:
....:     print("Числа закінчилися")
....:
0
1
2
```

4

Числа закінчилися

WHILE/ELSE

У циклі while:

- блок else виконується в тому випадку, якщо умова в while хибно
- else не виконується, якщо в циклі було виконано break

Приклад циклу while з else (блок else виконується після завершення циклу while):

```
In [4]: i = 0
In [5]: while i < 5:
....:     print(i)
....:     i += 1
....: else:
....:     print("Кінець")
....:
0
1
2
3
4
Кінець
```

Приклад циклу while з else та break у циклі (через break блок else не виконується):

```
In [6]: i = 0
In [7]: while i < 5:
....:     if i == 3:
....:         break
....:     else:
....:         print(i)
....:         i += 1
....: else:
....:     print("Кінець")
....:
0
1
2
```

 July 10, 2023 May 24, 2023

Робота з винятками try/except/else/finally

TRY/EXCEPT

Якщо ви повторювали приклади, які використовувалися раніше, то, напевно, були ситуації, коли виникала помилка. Швидше за все, це була помилка синтаксису, коли не вистачало, наприклад, двокрапки. Як правило, Python досить зрозуміло реагує на такі помилки, і їх можна виправити.

Проте, навіть коли код написаний синтаксично правильно, можуть виникати помилки. У Python ці помилки називаються винятки (exceptions).

Приклади винятків:

```
In [1]: 2/0
-----
ZeroDivisionError: division by zero

In [2]: 'test' + 2
-----
TypeError: must be str, not int
```

У цьому випадку виникли два винятки: `ZeroDivisionError` та `TypeError`.

Найчастіше можна передбачити, які винятки виникнуть під час виконання програми. Наприклад, якщо програма на вхід очікує два числа, а на виході видає їх суму, а користувач ввів замість одного з чисел рядок, з'явиться помилка `TypeError`, як у прикладі вище.

Python дозволяє працювати з винятками. Їх можна перехоплювати і виконувати певні дії у разі, якщо виник виняток.

Для роботи з винятками використовується конструкція `try/except`:

```
In [3]: try:
...:     2/0
...: except ZeroDivisionError:
...:     print("You can't divide by zero")
...:
You can't divide by zero
```

Конструкція `try` працює таким чином:

- спочатку виконуються вирази, які записані в блоці `try`
- якщо під час виконання блоку `try` не виникло жодних винятків, блок `except` пропускається, і виконується подальший код
- якщо під час виконання блоку `try` в якомусь місці виник виняток, частина блоку `try`, що залишилася, пропускається
- якщо в блоці `except` зазначено виняток, який виник, виконується код у блоці `except`

- якщо виняток, який виник, не вказано в блоці ехсерт, виконання програми переривається та генерується виняток

Note

Коли у програмі виникає виняток, вона одразу завершує роботу.

Зверніть увагу, що рядок Cool! у блоці try не виводиться:

```
In [4]: try:
...:     print("Let's divide some numbers")
...:     2/0
...:     print('Cool!')
...: except ZeroDivisionError:
...:     print("You can't divide by zero")
...:
Let's divide some numbers
You can't divide by zero
```

У конструкції try/ехсерт може бути багато ехсерт, якщо потрібні різні дії, залежно від типу помилки.

Наприклад, скрипт divide.py ділить два числа введених користувачем:

```
try:
    a = input("Введіть перше число: ")
    b = input("Введіть друге число: ")
    print("Результат: ", int(a)/int(b))
except ValueError:
    print("Вводьте лише числа")
except ZeroDivisionError:
    print("На нуль ділити не можна")
```

Приклади виконання скрипту:

```
$ python divide.py
Введіть перше число: 3
Введіть друге число: 1
Результат: 3

$ python divide.py
Введіть перше число: 5
Введіть друге число: 0
На нуль ділити не можна

$ python divide.py
Введіть перше число: qewr
Введіть друге число: 3
Вводьте лише числа
```


У цьому випадку виняток `ValueError` виникає, коли користувач ввів рядок замість числа під час переведення рядка в число. Виняток `ZeroDivisionError` виникає у разі, якщо друге число дорівнювало 0.

Якщо немає потреби виводити різні повідомлення на помилки `ValueError` та `ZeroDivisionError`, можна зробити так:

```
try:
    a = input("Введіть перше число: ")
    b = input("Введіть друге число: ")
    print("Результат: ", int(a)/int(b))
except (ValueError, ZeroDivisionError):
    print("Щось пішло не так...")
```

Перевірка:

```
$ python divide_ver2.py
Введіть перше число: wer
Введіть друге число: 4
Щось пішло не так...

$ python divide_ver2.py
Введіть перше число: 5
Введіть друге число: 0
Щось пішло не так...
```

Warning

У блоці ексерпт можна не вказувати конкретний виняток або винятки. У такому разі перехоплюватимуться майже всі винятки. Це робити не рекомендується!

TRY/EXCEPT/ELSE

У конструкції `try/except` є опціональний блок `else`. Він виконується у тому випадку, якщо не було винятку.

Наприклад, якщо необхідно виконувати надалі якісь операції з даними, які ввів користувач, можна записати їх у блоці `else`:

```
try:
    a = input("Введіть перше число: ")
    b = input("Введіть друге число: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Щось пішло не так...")
else:
    print("Результат: ", result)
```

Приклад виконання:

```
$ python divide_ver3.py
Введіть перше число: 10
Введіть друге число: 2
Результат: 5

$ python divide_ver3.py
Введіть перше число: werq
Введіть друге число: 3
Щось пішло не так...
```

TRY/EXCEPT/FINALLY

Блок `finally` – це ще один опціональний блок у конструкції `try`. Він виконується завжди, незалежно від того, чи був виняток чи ні.

Сюди ставляться дії, які треба виконати у будь-якому випадку. Наприклад, це може бути закриття файлу, вивільнення якихось ресурсів.

Приклад із блоком `finally`:

```
try:
    a = input("Введіть перше число: ")
    b = input("Введіть друге число: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Щось пішло не так...")
else:
    print("Результат: ", result**2)
finally:
    print("The End")
```

Перевірка:

```
$ python divide_ver4.py
Введіть перше число: 10
Введіть друге число: 2
Результат: 5
The End

$ python divide_ver4.py
Введіть перше число: qwerewr
Введіть друге число: 3
Щось пішло не так...
The End

$ python divide_ver4.py
Введіть перше число: 4
Введіть друге число: 0
Щось пішло не так...
The End
```

КОЛИ ВИКОРИСТОВУВАТИ ВИНЯТКИ

Як правило, той самий код можна написати і з використанням винятків, і без них. Наприклад, цей варіант коду:

```
while True:
    a = input("Введіть перше число: ")
    b = input("Введіть друге число: ")
    try:
        result = int(a)/int(b)
    except ValueError:
        print("Підтримуються лише числа")
    except ZeroDivisionError:
        print("На нуль ділити не можна")
    else:
        print(result)
        break
```

Можна переписати код таким чином без try/except:

```
while True:
    a = input("Введіть перше число: ")
    b = input("Введіть друге число: ")
    if a.isdigit() and b.isdigit():
        if int(b) == 0:
            print("На нуль ділити не можна")
        else:
            print(int(a)/int(b))
            break
    else:
        print("Підтримуються лише числа")
```

Не завжди аналогічний варіант без використання винятків буде простим і зрозумілим.

Важливо в кожній конкретній ситуації оцінювати, який варіант коду зрозуміліший, компактніший і універсальніший - з винятками або без.

Якщо ви раніше використовували якусь іншу мову програмування, є ймовірність, що використання винятків вважалось поганим тоном. У Python це не так.

RAISE

Іноді в коді треба згенерувати виняток, це можна зробити так:

```
raise ValueError("При виконанні команди виникла помилка")
```

ВБУДОВАНІ ВИНЯТКИ

У Python є багато **вбудованих винятків**, кожне з яких генерується у певній ситуації.

Наприклад, `TypeError` зазвичай генерується, коли очікувався один тип даних, а передали інший

```
In [1]: "a" + 3
-----
TypeError                                Traceback (most recent call last)
<ipython-input-1-5aa8a24e3e06> in <module>
----> 1 "a" + 3
TypeError: can only concatenate str (not "int") to str
```

ValueError коли тип вірний, але значення не відповідає очікуваному:

```
In [2]: int("a")
-----
ValueError                                Traceback (most recent call last)
<ipython-input-2-d9136db7b558> in <module>
----> 1 int("a")
ValueError: invalid literal for int() with base 10: 'a'
```

🕒 July 10, 2023

🕒 May 24, 2023

Додаткова інформація

Документація:

- [Compound statements \(if, while, for, try\)](#)
- [break, continue](#)
- [Errors and Exceptions](#)
- [Built-in Exceptions](#)
- [Operator precedence](#)

Статті:

- [Write Cleaner Python: Use Exceptions](#)
- [Robust exception handling](#)
- [Python Exception Handling Techniques](#)

Stack Overflow:

- [Why does python use 'else' after for and while loops?](#)
- [Is it a good practice to use try-except-else in Python?](#)

 July 5, 2023

 May 24, 2023

7. Робота з файлами

7. Робота з файлами

У реальному житті для того, щоб повноцінно використовувати все, що розглядалося до цього розділу, треба розібратися як працювати з файлами.

При роботі з мережевим обладнанням (і не тільки) файлами можуть бути:

- конфігурації (прості, не структуровані текстові файли) - робота з ними розглядається в цьому розділі
- шаблони конфігурацій - зазвичай це якийсь спеціальний формат файлів
- файли з параметрами підключень. Як правило, це структуровані файли, в певному форматі: YAML, JSON, CSV

У цьому розділі розглядається робота із простими текстовими файлами. Наприклад, конфігураційний файл Cisco.

У роботі з файлами є кілька аспектів:

- відкриття/закриття
- читання
- запис

У цьому розділі розглядається лише необхідний мінімум для роботи з файлами. Докладніше в [документації Python](#).

🕒 July 10, 2023

🕒 July 10, 2023

Відкриття файлів

Для початку роботи з файлом його треба відкрити.

OPEN

Для відкриття файлів найчастіше використовується функція open:

```
file = open('file_name.txt', 'r')
```

У функції open:

- 'file_name.txt' - ім'я файлу. Тут можна вказувати ім'я файлу або шлях (абсолютний чи відносний)
- 'r' - режим відкриття файлу

Функція open створює об'єкт file, якого потім можна використовувати різні способи, до роботи з ним.

Режими відкриття файлів (без +):

- r - відкрити файл тільки для читання (за замовчуванням)
- w - відкрити файл для запису
- якщо файл існує, його вміст видаляється
- якщо файл не існує, то створюється новий
- x - відкрити файл для ексклюзивного створення:
- якщо файл існує, запис не відбувається
- якщо файл не існує, то створюється новий
- a - відкрити файл для доповнення запису. Дані додаються до кінця файлу

Режим відкриття	r	r+	w	w+	a	a+	x	x+
читання	+	+		+		+		+
запис		+	+	+	+	+	+	+
створення нового файлу			+	+	+	+	+	+
відкриття існуючого файлу			+	+	+	+		
вміст файлу видаляється			+	+				
позиція на початку	+	+	+	+			+	+
позиція в кінці					+	+		
запис після seek		+	+	+			+	+

Пояснення до таблиці

Джерело таблиці

- читання - дозволено читання з файлу
- запис - запис у файл дозволений
- створення - файл створюється, якщо він ще не існує
- вміст файлу видалюється - під час відкриття файл стає порожнім (увесь вміст файлу стирається)
- позиція на початку - після відкриття файлу початкова позиція встановлюється на початок файлу
- позиція в кінці - після відкриття файлу початкова позиція встановлюється в кінець файлу

 July 12, 2023

 July 10, 2023

Читання файлів

У Python є кілька варіантів читання файлу:

- перебирати файл у циклі `for`
- метод `read` - зчитує вміст файлу в рядок
- метод `readline` - зчитує один рядок
- метод `readlines` - зчитує рядки файлу та створює список із рядків

РОБОТА ІЗ ФАЙЛОМ В ЦИКЛІ FOR

Найпоширеніший варіант роботи з файлом – використання об'єкту `file` у циклі `for`:

Приклад роботи із файлом в циклі `for` Файл `data.txt`

В цьому випадку файл читається рядок за рядком, отримуючи новий рядок на кожній ітерації циклу `for`:

```
In [1]: from pprint import pprint
```

```
In [2]: f = open('data.txt')
```

```
In [3]: for line in f:  
...:     pprint(line)  
...:
```

```
'line1\n'  
'line2\n'  
'line3\n'  
'line4\n'  
'line5\n'  
'line6\n'  
'line7\n'
```

```
line1  
line2  
line3  
line4  
line5  
line6  
line7
```

READ

Метод `read` – зчитує весь файл в один рядок.

`read` Файл `data.txt`

Приклад використання методу `read`:

```
In [1]: f = open('data.txt')

In [2]: f.read()
Out[2]: 'line1\nline2\nline3\nline4\nline5\nline6\nline7\n'

In [3]: f.read() # (1)
Out[3]: ''
```

1. При повторному читанні файлу в 3 рядку відображається порожній рядок. Так відбувається через те, що після першого виклику методу `read`, зчитується весь файл. І після того, як файл було зчитано, курсор залишається в кінці файлу. Керувати положенням курсору можна за допомогою методу [seek](#).

```
line1
line2
line3
line4
line5
line6
line7
```

READLINE

Файл можна читати рядок за рядком за допомогою методу `readline`.

`readline` Файл data.txt

Кожен виклик методу `readline`, читає наступний рядок файлу. Коли рядки закінчуються, `readline` повертає порожній рядок:

```
In [1]: f = open('data.txt')
```

```
In [2]: f.readline()
```

```
Out[2]: 'line1\n'
```

```
In [3]: f.readline()
```

```
Out[3]: 'line2\n'
```

```
In [4]: f.readline()
```

```
Out[4]: 'line3\n'
```

```
In [5]: f.readline()
```

```
Out[5]: ''
```

```
In [6]: f.readline()
```

```
Out[6]: ''
```

```
line1
```

```
line2
```

```
line3
```

READLINES

`readlines` Файл data.txt

Метод `readlines` зчитує рядки файлу до списку:

```
In [6]: f = open('data.txt')
```

```
In [7]: f.readlines()
```

```
Out[7]: ['line1\n', 'line2\n', 'line3\n', 'line4\n', 'line5\n', 'line6\n', 'line7\n']
```

```
line1
```

```
line2
```

```
line3
```

```
line4
```

```
line5
```

```
line6
```

```
line7
```

Якщо потрібно отримати рядки файлу, але без перекладу рядка в кінці, можна скористатися методом `split` і як роздільник вказати символ `\n`:

```
In [2]: f = open('data.txt')

In [3]: f.read().split('\n')
Out[3]: ['line1', 'line2', 'line3', 'line4', 'line5', 'line6', 'line7', '']
```

Зауважте, що останній елемент списку - порожній рядок. Якщо перед виконанням `split`, скористатися методом `rstrip`, список буде без порожнього рядка наприкінці:

```
In [5]: f = open('data.txt')

In [6]: f.read().rstrip().split('\n')
Out[6]: ['line1', 'line2', 'line3', 'line4', 'line5', 'line6', 'line7']
```

Або скористатися методом рядків `splitlines`:

```
In [11]: f = open('data.txt')

In [12]: f.read().splitlines()
Out[12]: ['line1', 'line2', 'line3', 'line4', 'line5', 'line6', 'line7']
```

SEEK

Досі файл щоразу доводилося відкривати заново, щоб прочитати його знову. Так відбувається через те, що після методів читання курсор знаходиться в кінці файлу. І повторне читання повертає порожній рядок.

Щоб знову прочитати інформацію з файлу, потрібно скористатися методом `seek`, який переміщує курсор у потрібне положення.

Приклад відкриття файлу та читання вмісту:

```
In [13]: f = open('data.txt')

In [14]: f.read()
Out[14]: 'line1\nline2\nline3\nline4\nline5\nline6\nline7\n'
```

Якщо викликати ще раз метод `read`, повертається порожній рядок:

```
In [15]: f.read()
Out[15]: ''
```

За допомогою методу `seek` можна перейти на початок файлу (0 означає початок файлу):

```
In [16]: f.seek(0)
Out[16]: 0
```

Після того, як за допомогою seek курсор був переведений на початок файлу, можна знову зчитувати вміст:

```
In [17]: f.read()  
Out[17]: 'line1\nline2\nline3\nline4\nline5\nline6\nline7\n'
```

🕒 July 12, 2023

🕒 July 10, 2023

Запис файлів

При записі в файл дуже важливо визначитися з режимом відкриття файлу, щоб випадково не видалити вміст файлу:

- `w` - відкрити файл для запису. Якщо файл існує, то його вміст видаляється
- `a` - відкрити файл для доповнення запису. Дані додаються в кінці файлу
- `x` - відкрити файл для запису. Якщо файл існує, запис не виконується

Всі режими створюють файл, якщо він не існує.

Для запису в файл використовуються такі методи:

- `write` - записати один рядок у файл
- `writelines` - дозволяє передавати список рядків як аргумент

WRITE

Метод `write` очікує рядок для запису.

Наприклад, візьмемо список рядків:

```
lines = ["line1\n", "line2\n", "line3\n"]
```

Відкриття файлу `new_data.txt` в режимі запису:

```
In [1]: f = open('new_data.txt', "w")
```

Перетворюємо список команд в один рядок за допомогою `join`:

```
In [2]: lines_as_string = ''.join(lines)
```

```
In [3]: lines_as_string
Out[3]: 'line1\nline2\nline3\n'
```

Запис рядка у файл:

```
In [4]: f.write(lines_as_string)
Out[4]: 18
```

Після завершення роботи з файлом його необхідно закрити:

```
In [5]: f.close()

In [6]: cat new_data.txt # (1)
line1
line2
line3
```

1. Оскільки `ipython` підтримує команду `cat`, можна переглянути вміст файлу `cat new_data.txt`

WRITELINES

Метод `writelines` очікує як аргумент ітерований об'єкт з рядками. Запис списку рядків `lines` у файл:

```
In [1]: lines = ["line1\n", "line2\n", "line3\n"] # (1)

In [3]: f = open('new_data.txt', "w")

In [5]: f.writelines(lines)

In [6]: f.close()

In [7]: cat new_data.txt
line1
line2
line3
```

1. Метод `writelines` не додає символ нового рядка, тому кожен рядок має бути з потрібним символом нового рядка.

РЕЖИМ a

Якщо відкрити файл у режимі `a`, файл буде відкритий для доповнення запису. Дані додаються в кінці файлу:

```
In [1]: lines = ["line1\n", "line2\n", "line3\n"]

In [2]: f = open('new_data.txt', "a")

In [4]: f.writelines(["line4\n", "line5\n"])

In [5]: f.close()

In [6]: cat new_data.txt
line1
line2
line3
line4
line5
```

РЕЖИМ x

Якщо у режимі `x` відкрити існуючий файл, виникне виняток `FileExistsError`:

```
In [1]: f = open('new_data.txt', "x")
-----
FileExistsError                                Traceback (most recent call last)
Cell In[1], line 1
----> 1 f = open('new_data.txt', "x")
...
```

```
FileExistsError: [Errno 17] File exists: 'new_data.txt'
```

З новим файлом такої проблеми не буде і дані додаються як в режимі `w`:

```
In [8]: f = open('new_data_x.txt', "x")  
  
In [9]: lines = ["line1\n", "line2\n", "line3\n"]  
  
In [10]: f.writelines(lines)  
  
In [11]: f.close()
```

🕒 July 12, 2023

🕒 July 10, 2023

Закриття файлів

У реальному житті для закриття файлів найчастіше використовується конструкція `with`. Її набагато зручніше і безпечніше використовувати, ніж закривати файл явно. Але, оскільки в житті можна зустріти і метод `close`, у цьому розділі сприймається як його використовувати.

Після завершення роботи з файлом його потрібно закрити. У деяких випадках Python може самостійно закрити файл. Але краще не розраховувати і закривати файл явно.

CLOSE

Метод `close` зустрічався у розділі запис файлів. Там він був потрібний для того, щоб вміст файлу був записаний на диск.

Для цього в Python є окремий метод `flush`. Але так як у прикладі із записом файлів, не потрібно було виконувати жодних операцій, файл можна було закрити.

Відкриємо файл `r1.txt`:

```
In [1]: f = open('r1.txt', 'r')
```

Тепер ми можемо прочитати файл:

```
In [2]: print(f.read())
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

Об'єкт `file` має спеціальний атрибут `closed`, який дозволяє перевірити, закритий файл чи ні. Якщо файл відкритий, він повертає `False`:

```
In [3]: f.closed
Out[3]: False
```

Тепер закриваємо файл і знову перевіряємо `closed`:

```
In [4]: f.close()

In [5]: f.closed
Out[5]: True
```

Якщо спробувати прочитати файл, виникне виняток:

```
In [6]: print(f.read())
-----
ValueError                                Traceback (most recent call last)
<ipython-input-53-2c962247edc5> in <module>()
----> 1 print(f.read())

ValueError: I/O operation on closed file
```

🕒 July 12, 2023

🕒 July 10, 2023

Блок with

У Python існує більш зручний спосіб роботи з файлами, ніж ті, які використовувалися досі - конструкція with. Конструкція with називається менеджер контексту.

```
In [1]: with open('r1.txt', 'r') as f:
.....:     for line in f:
.....:         print(line)
.....:
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

З конструкцією with можна використовувати не тільки такий рядковий варіант зчитування, всі методи, що розглядалися до цього, також працюють:

```
In [4]: with open('r1.txt', 'r') as f:
.....:     print(f.read())
.....:
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

ВІДКРИТТЯ ДВОХ ФАЙЛІВ

Іноді потрібно працювати одночасно із двома файлами. Наприклад, треба записати деякі рядки з одного файлу, до іншого.

У такому випадку, у блоці with можна відкривати два файли таким чином:

```
with open('r1.txt') as src, open('result.txt', 'w') as dest:
    for line in src:
        if line.startswith('service'):
            dest.write(line)

In [6]: cat result.txt
service timestamps debug datetime msec localtime show-timezone year
```

```
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
```

Це рівнозначно таким двом блокам with:

```
with open('r1.txt') as src:
    with open('result.txt', 'w') as dest:
        for line in src:
            if line.startswith('service'):
                dest.write(line)
```

🕒 July 12, 2023

🕒 July 10, 2023

Приклади роботи з файлами

У цьому підрозділі розглядається робота з файлами та поєднуються теми: файли, цикли та умови.

При обробці виводу команд чи конфігурації часто потрібно буде записати підсумкові дані у словник. Не завжди очевидно, як обробляти виводу команд і як загалом підходити до розбору виводу на частини. У цьому підрозділі розглядаються кілька прикладів, із зростаючим рівнем складності.

РОЗБІР ВИВОДУ СТОВПЦЯМИ

У цьому прикладі розбиратиметься вивід команди `sh ip int br`. З виводу команди нам треба отримати відповідність ім'я інтерфейсу – IP-адресу. Тобто ім'я інтерфейсу – це ключ словника, а IP-адреса – значення. При цьому, відповідність треба робити тільки для тих інтерфейсів, у яких призначено IP-адресу.

Приклад виводу команди `sh ip int br` (файл `sh_ip_int_br.txt`):

```
R1#show ip interface brief
Interface                IP-Address      OK? Method Status        Protocol
FastEthernet0/0          15.0.15.1       YES manual up             up
FastEthernet0/1          10.0.12.1       YES manual up             up
FastEthernet0/2          10.0.13.1       YES manual up             up
FastEthernet0/3          unassigned      YES unset  up             down
Loopback0                 10.1.1.1        YES manual up             up
Loopback100              100.0.0.1       YES manual up             up
```

Файл `working_with_dict_example_1.py`:

```
result = {}

with open('sh_ip_int_br.txt') as f:
    for line in f:
        line_list = line.split()
        if line_list and line_list[1][0].isdigit():
            interface = line_list[0]
            address = line_list[1]
            result[interface] = address

print(result)
```

Команда `sh ip int br` відображає вивід стовпцями. Всі потрібні поля знаходяться в одному рядку. Скрипт обробляє вивід рядок за рядком і кожен рядок розбиває методом `split`.

Отриманий список містить стовпці виводу. Так як з усього виводу потрібні тільки інтерфейси, на яких налаштована IP-адреса, виконується перевірка першого символу другого стовпця: якщо перший символ число, значить на інтерфейсі призначена адреса і цей рядок треба обробляти.

Оскільки для кожного рядка є пара ключа та значення, вони присвоюються у словник:

```
result[interface] = address.
```

Результатом виконання скрипта буде такий словник (тут він розбитий на пари ключ-значення для зручності, у реальному виводу скрипта словник відображатиметься в один рядок):

```
{'FastEthernet0/0': '15.0.15.1',
 'FastEthernet0/1': '10.0.12.1',
 'FastEthernet0/2': '10.0.13.1',
 'Loopback0': '10.1.1.1',
 'Loopback100': '100.0.0.1'}
```

ОТРИМАННЯ КЛЮЧА ТА ЗНАЧЕННЯ З РІЗНИХ РЯДКІВ ВИВОДУ

Дуже часто вивід команд виглядає таким чином, що ключ та значення знаходяться у різних рядках.

Наприклад, з виводу команди `sh ip interface` треба отримати ім'я інтерфейсу та MTU (файл `sh_ip_interface.txt`):

```
Ethernet0/0 is up, line protocol is up
  Internet address is 192.168.100.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/1 is up, line protocol is up
  Internet address is 192.168.200.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/2 is up, line protocol is up
  Internet address is 19.1.1.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
```

Ім'я інтерфейсу знаходиться у рядку `Ethernet0/0 is up, line protocol is up`, а MTU у рядку `MTU is 1500 bytes`.

Наприклад, спробуємо запам'ятовувати щоразу інтерфейс у змінну `interface` і виводити його значення `print`, але тільки коли зустрічається рядок зі значенням MTU:

```
with open('sh_ip_interface.txt') as f:
    for line in f:
        if 'line protocol' in line:
```

```

        interface = line.split()[0]
    elif 'MTU is' in line:
        mtu = line.split()[-2]
        print('{:15}{}'.format(interface, mtu))

```

Вивід

```

Ethernet0/0    1500
Ethernet0/1    1500
Ethernet0/2    1500
Ethernet0/3    1500
Loopback0      1514

```

Вивід організований таким чином, що спочатку йде рядок з інтерфейсом, а потім через кілька рядків - рядок з MTU. Якщо запам'ятовувати ім'я інтерфейсу кожного разу, коли воно зустрічається, то на момент, коли зустрінеється рядок з MTU, останній записаний інтерфейс - це той, до якого належить MTU.

Тепер, якщо необхідно створити словник з відповідністю інтерфейсу - MTU, достатньо записати значення на момент, коли був знайдений MTU.

```

result = {}

with open('sh_ip_interface.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
        elif 'MTU is' in line:
            mtu = line.split()[-2]
            result[interface] = mtu

print(result)

```

Результатом виконання скрипта буде такий словник (тут він розбитий на пари ключ-значення для зручності, у реальному результаті скрипта словник відображатиметься в один рядок):

```

{'Ethernet0/0': '1500',
 'Ethernet0/1': '1500',
 'Ethernet0/2': '1500',
 'Ethernet0/3': '1500',
 'Loopback0': '1514'}

```

Цей трюк часто буде корисним, оскільки часто вивід команд організований подібним чином.

ВКЛАДЕНИЙ СЛОВНИК

Якщо потрібно отримати кілька параметрів з виводу команди, дуже зручно використовувати словник із вкладеним словником.

Наприклад, з виводу `sh ip interface` потрібно отримати два параметри IP-адресу та MTU:

```
Ethernet0/0 is up, line protocol is up
Internet address is 192.168.100.1/24
Broadcast address is 255.255.255.255
Address determined by non-volatile memory
MTU is 1500 bytes
Helper address is not set
...
Ethernet0/1 is up, line protocol is up
Internet address is 192.168.200.1/24
Broadcast address is 255.255.255.255
Address determined by non-volatile memory
MTU is 1500 bytes
Helper address is not set
...
Ethernet0/2 is up, line protocol is up
Internet address is 19.1.1.1/24
Broadcast address is 255.255.255.255
Address determined by non-volatile memory
MTU is 1500 bytes
Helper address is not set
...
```

На першому етапі кожне значення зберігається в змінній, а потім виводяться всі три значення. Значення відображаються, коли зустрічається рядок MTU, оскільки він зустрічається у виводі останнім:

```
with open('sh_ip_interface.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
        elif 'Internet address' in line:
            ip_address = line.split()[-1]
        elif 'MTU' in line:
            mtu = line.split()[-2]
            print('{:15}{:17}{:}'.format(interface, ip_address, mtu))

Ethernet0/0      192.168.100.1/24 1500
Ethernet0/1      192.168.200.1/24 1500
Ethernet0/2      19.1.1.1/24      1500
Ethernet0/3      192.168.230.1/24 1500
Loopback0        4.4.4.4/32       1514
```

Тут ми використовуємо ту саму техніку, що й у попередньому прикладі, але додаємо ще одне вкладення словника:

```
from pprint import pprint

result = {}

with open('sh_ip_interface.txt') as f:
```



```

for line in f:
    if 'line protocol' in line:
        interface = line.split()[0]
        result[interface] = {}
    elif 'Internet address' in line:
        ip_address = line.split()[-1]
        result[interface]['ip'] = ip_address
    elif 'MTU' in line:
        mtu = line.split()[-2]
        result[interface]['mtu'] = mtu

pprint(result)

```

Щоразу, коли зустрічається інтерфейс, у словнику результатів створюється ключ із назвою інтерфейсу і значення порожній словник. Це потрібно для того, щоб у момент виявлення IP-адреси або MTU, параметр можна було записати до вкладеного словника відповідного інтерфейсу.

Результатом виконання скрипта буде наступний словник:

```

{'Ethernet0/0': {'ip': '192.168.100.1/24', 'mtu': '1500'},
 'Ethernet0/1': {'ip': '192.168.200.1/24', 'mtu': '1500'},
 'Ethernet0/2': {'ip': '19.1.1.1/24', 'mtu': '1500'},
 'Ethernet0/3': {'ip': '192.168.230.1/24', 'mtu': '1500'},
 'Loopback0': {'ip': '4.4.4.4/32', 'mtu': '1514'}}

```

ВИВІД ІЗ ПОРОЖНІМИ ЗНАЧЕННЯМИ

Іноді, у виводі команди траплятимуться секції з порожніми значеннями. Наприклад, у випадку з `sh ip interface` можуть траплятися інтерфейси, які виглядають так:

```

Ethernet0/1 is up, line protocol is up
  Internet protocol processing disabled
Ethernet0/2 is administratively down, line protocol is down
  Internet protocol processing disabled
Ethernet0/3 is administratively down, line protocol is down
  Internet protocol processing disabled

```

Відповідно, тут немає MTU та IP-адреси.

І якщо виконати попередній скрипт для файлу з такими інтерфейсами, результат буде таким (вивід для файлу `sh_ip_interface2.txt`):

```

{'Ethernet0/0': {'ip': '192.168.100.2/24', 'mtu': '1500'},
 'Ethernet0/1': {},
 'Ethernet0/2': {},
 'Ethernet0/3': {},
 'Loopback0': {'ip': '2.2.2.2/32', 'mtu': '1514'}}

```

Якщо необхідно додавати інтерфейси до словника лише, коли на інтерфейсі призначено IP-адресу, треба перенести створення ключа з ім'ям інтерфейсу на момент, коли зустрічається рядок з IP-адресою:

```
result = {}

with open('sh_ip_interface2.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
        elif 'Internet address' in line:
            ip_address = line.split()[-1]
            result[interface] = {}
            result[interface]['ip'] = ip_address
        elif 'MTU' in line:
            mtu = line.split()[-2]
            result[interface]['mtu'] = mtu

print(result)
```

У цьому випадку результатом буде такий словник:

```
{'Ethernet0/0': {'ip': '192.168.100.2/24', 'mtu': '1500'},
 'Loopback0': {'ip': '2.2.2.2/32', 'mtu': '1514'}}
```

🕒 July 12, 2023

🕒 July 10, 2023

Додаткова інформація

Документація:

- [Reading and Writing Files](#)
- [The with statement](#)

Статті:

- [The Python "with" Statement by Example](#)

Stack Overflow:

- [What is the python "with" statement designed for?](#)

🕒 July 10, 2023

🕒 July 10, 2023

Завдання

Завдання

Після більшості тем курсу/книги є завдання. В таблиці нижче написані теми курсу/книги, відповідний розділ завдань, та мінімальна кількість завдань у кожній темі, яку потрібно виконати.

Завдання можна виконувати локально на своєму комп'ютері, на віртуальній машині чи на сайті replit.com.

Де знаходяться завдання:

- [Репозиторій із завданнями](#)
- [Шаблон replit з завданнями 4-17 розділів](#)

Етапи роботи із завданнями:

1. Виконання завдань
2. Перевірте, що завдання відпрацьовує як потрібно `python task_4_2.py` або запуск скрипта в редакторі/IDE
3. Перевірка завдань тестами `runeng 1-5`
4. Якщо тести проходять, дивимось варіанти вирішення `runeng 1-5 -a`

Утиліта runeng

[Як перевіряти завдання, дивитися варіанти рішення](#)

Таблиця тема-завдання

Краще виконувати всі завдання, тому що практика це головне у навчанні, але якщо ви хочете перейти на наступну тему або пропустити якісь складніші завдання, вказаний мінімум можна використовувати як орієнтир.

Теми	Всього завдань	Мінімум завдань
Підготовка до курсу	11 (00_intro)	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
Підготовка робочого середовища	-	-
Робота з завданнями. Git/GitHub	-	-
Типи даних в Python.	11 (04_data_structures)	4.0, 4.1, 4.2, 4.3, 4.6, 4.9
Створення базових скриптів.	12 (05_basic_scripts)	5.0, 5.1, 5.2, 5.3, 5.3a, 5.3b, 5.4
Управління ходом програми.	9 (06_control_structures)	6.0, 6.1, 6.2, 6.3, 6.6
Налагодження коду.	-	-
Робота із файлами.	9 (07_files)	7.1, 7.2, 7.2a, 7.3, 7.4

Корисні можливості та інструменти (розпакування змінних, list/dict/set comprehensions)	-	-
Функції.	10 (09_functions)	9.0, 9.1, 9.2, 9.3, 9.3a, 9.6
Корисні функції у стандартній бібліотеці	-	-
Модулі	6 (11_modules)	11.0, 11.1, 11.2, 11.3, 11.4
Корисні модулі	3 (12_useful_modules)	12.1, 12.2
Синтаксис регулярних виразів.	-	-
Модуль re.	7 (15_module_re)	15.0, 15.1, 15.2, 15.3, 15.4
Unicode	-	-
Обробка даних у форматах YAML, JSON, CSV	6 (17_serialization)	17.1, 17.2, 17.3
Підключення до мережевих пристроїв Telnet і SSH	8 (18_ssh_telnet)	18.1, 18.1a, 18.2, 18.2a, 18.2b, 18.2c
Одночасне підключення до кількох пристроїв	5 (19_concurrent_connections)	19.1, 19.2, 19.3
Шаблони конфігурацій із Jinja2.	6 (20_jinja2)	20.1, 20.2, 20.3
Шаблони TextFSM для розбору виводу команд	6 (21_textfsm)	21.1, 21.1a, 21.2, 21.3, 21.4
Основи ООП.	9 (22_oop_basics)	22.0, 22.1, 22.1a, 22.1b, 22.2, 22.2a
ООП. Спеціальні методи.	5 (23_oop_spec_methods)	23.0, 23.1, 23.1a, 23.2
ООП. Успадкування.	7 (24_oop_inheritance)	24.0, 24.1, 24.2, 24.2a
Основи роботи з базами даних (SQLite3)	9 (25_db)	-

🕒 May 18, 2023

🕒 May 15, 2023

Утиліта rупeng

Де робити завдання

Завдання треба виконувати у підготовлених файлах.

Наприклад, у розділі 04_data_structures є завдання 4.3. Щоб виконати його, потрібно відкрити файл exercises/04_data_structures/task_4_3.py і виконувати завдання прямо в цьому файлі після опису завдання.

Скрипт rупeng

Встановлення скрипту **rupeng**

```
pip install rupeng-cli
```

Це встановить модуль і дозволить викликати його в каталогах завдань за словом rупeng.

Етапи роботи із завданнями:

1. Виконання завдань
2. Перевірте, що завдання відпрацьовує як потрібно `python task_4_2.py` або запуск скрипта в редакторі/IDE
3. Перевірка завдань тестами `rupeng 1-5`
4. Якщо тести проходять, дивимось варіанти розв'язання `rupeng 1-5 -a`

Другий крок дуже важливий, тому що на цьому етапі набагато простіше знайти помилки у синтаксисі та подібні проблеми з роботою скрипту, ніж при запуску коду через тест на 3 етапі.

Перевірка завдань тестами

Після виконання завдання його треба перевірити за допомогою тестів. Для запуску тестів потрібно викликати rупeng в каталозі завдань. Наприклад, якщо ви робите 4 розділ завдань, треба перебувати в каталозі exercises/04_data_structures/ і запустити rупeng одним із способів, залежно від того, які завдання на перевіряти.

Примеры вывода тестов с пояснениями

Запуск перевірки всіх завдань поточного розділу:

```
rupeng
```

Запуск тестів для завдання 4.1:

```
runeng 1
```

Запуск тестів для завдань 4.1, 4.2, 4.3:

```
runeng 1-3
```

Якщо є завдання з літерами, наприклад, у 7 розділі, можна запускати так, щоб запустити перевірку для завдань 7.2a, 7.2b (треба перебувати в каталозі 07_files):

```
runeng 2a-b
```

або так, щоб запустити всі завдання 7.2x з літерами та без:

```
runeng 2*
```

Отримання відповідей

Цей функціонал працює тільки за наявності [git cli](#).

Якщо завдання проходять тести, можна переглянути варіанти вирішення завдань.

Для цього до попередніх варіантів команди слід додати -a. Такий виклик означає запустити тести для завдань 1 та 2 та скопіювати відповіді, якщо тести пройшли:

```
runeng 1-2 -a
```

Тоді для вказаних завдань запустяться тести і для тих завдань, які пройшли тести, скопіюються відповіді у файли answer_task_x.py в поточному каталозі.

Завантажити всі зміни в поточному каталозі на github, без прив'язки до того чи проходять тести

```
runeng --save-all
```

Виконує команди

```
git add .  
git commit -m "Все изменения сохранены"  
git push origin main
```

Оновлення розділів

Цей функціонал працює тільки за наявності [git cli](#).

У rypeng є два варіанти оновлення: оновлювати розділи або конкретні завдання/тести. При оновленні розділу каталог розділу видаляється і копіюється нова версія. Це підходить тільки для тих розділів, які ви ще не виконували. Якщо потрібно оновити конкретне завдання, то краще використовувати оновлення конкретних завдань (розглядається далі).

Перед будь-яким варіантом поновлення бажано зберегти всі локальні зміни на github!

Для оновлення розділів, треба перейти до каталогу exercises/ та дати команду:

```
rypeng --update-chapters 12-25
```

В цьому випадку оновляться розділи 12-25. Також можна вказувати один розділ:

```
rypeng --update-chapters 11
```

Або кілька через кому

```
rypeng --update-chapters 12,15,17
```

Оновлення завдань та тестів

У завданнях і тестах зустрічаються неточності і щоб їх можна було виправити, rypeng додано опцію `--update`.

Загальна логіка:

- завдання та тести копіюються з [репозиторію](#)
- копіює весь файл завдання, не тільки опис, тому файл перепишеться
- перед виконанням `-update`, краще зберегти всі зміни на github

Як працює `--update`:

- якщо у репозиторії є незбережені зміни, утиліта пропонує їх зберегти (зробить `git add.`, `git commit`, `git push`)
- якщо незбережених змін немає, копіюються зазначені завдання та тести
- утиліта пропонує зберегти зміни та показує які файли змінені, але не які саме зроблені зміни
- можна відмовитись зберігати зміни та подивитися зміни `git diff`

ВАРІАНТИ ВИКЛИКУ

Оновити всі завдання та тести розділу:

```
rypeng --update
```

Оновити всі тести розділу (тільки тести, не завдання):


```
pyneng --update --test-only
```

Оновити завдання 1,2 та відповідні тести розділу

```
pyneng 1,2 --update
```

Якщо жодних оновлень немає, буде такий вивід

```
$ pyneng --update
##### git pull
Already up-to-date.

Updated tasks and tests copied
Tasks and tests are already the latest version
Aborted!
```

Будь-коли можна перервати оновлення Ctrl-C.

Приклад виведення з незбереженими змінами та наявністю оновлень

```
pyneng --update
THIS WILL OVERWRITE THE CONTENT OF UNSAVED FILES!
There are unsaved changes in the current directory! Do you want to save them? [y/n]:y
##### git add .
##### git commit -m "Save changes before updating tasks"
[main 0e8c1cb] Saving changes before updating tasks
1 file changed, 1 insertion(+)

##### git push origin main
To git@github.com:pyneng/my-tasks.git
fa338c3..0e8c1cb main -> main

All changes in the current directory are saved. Let's start updating...
##### git pull
Already up-to-date.

Updated tasks and tests copied
The following files have been updated:
##### git diff --stat
exercises/04_data_structures/task_4_0.py | 0
exercises/04_data_structures/task_4_1.py | one -
exercises/04_data_structures/task_4_3.py | 3---
3 files changed, 0 insertions(+), 4 deletions(-)

This is a short diff, if you want to see all the differences in detail, press n and issue
the git diff command.
You can also undo your changes with git checkout -- file (or git restore file) if you want.

Save changes and add to github? [y/n]:n
```

tasks and tests have been successfully updated
Aborted!

🕒 May 15, 2023

🕒 May 15, 2023

Таблиці з темами завдань

Таблиця тем містить підказки щодо того, які теми використовуються в завданні. Не обов'язково використовувати ці теми для виконання завдань.

У таблицях * зазначено завдання з мінімуму за розділами. Краще виконувати всі завдання, тому що практика це головне у навчанні, але якщо ви хочете перейти на наступну тему або пропустити якісь складніші завдання, вказаний мінімум можна використовувати як орієнтир.

04_data_structures

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
4.1*	print, рядки, методи рядків
4.2*	print, рядки, методи рядків
4.3*	print, рядки, методи рядків, списки
4.4	print, списки, множини, сортування
4.5	print, методи рядків, списки, множини, сортування
4.6*	print, рядки, форматування рядків
4.7	print, рядки, форматування рядків, перетворення типів
4.8	print, рядки, форматування рядків, списки, перетворення типів
4.9*	print, рядки, методи рядків, списки
4.9a	print, рядки, методи рядків, списки
4.9b	print, рядки, методи рядків, списки

05_basic_scripts

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
5.1*	print, input, списки
5.2*	print, input, рядки
5.3*	print, input, словники
5.3a*	print, input, словники
5.3b*	print, input, рядки, списки, словники
5.3c	print, input, рядки, списки, словники
5.3d	print, input, рядки, списки, словники
5.4*	print, input, рядки, списки, форматування рядків
5.4a	print, input, рядки, списки, форматування рядків
5.4b	print, input, рядки, списки, форматування рядків
5.5	print, input, словники, рядки, форматування рядків
5.5a	print, input, словники, рядки, форматування рядків

06_control_structures

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
6.1*	print, for, списки
6.2*	print, for, if, рядки
6.3*	print, for, if, try/except, рядки, списки
6.4	print, for, рядки, списки, форматування рядків
6.5	print, for, if, рядки, range, break, input
6.6*	print, input, if, рядки, списки
6.6a	print, input, if, for, try/except, рядки, списки
6.6b	print, input, while, if, for, try/except, рядки, списки
6.7	print, if, for, словники, рядки, списки, форматування рядків

07_files

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
7.1*	print, робота з файлами, for, строки, списки, форматування рядків
7.2*	print, робота з файлами, for, if, рядки, sys.argv
7.2a*	print, робота з файлами, for, if, рядки, списки, множини, sys.argv
7.2b	print, робота з файлами, for, if, рядки, списки, множини, sys.argv
7.3*	print, робота з файлами, for, if, рядки, списки, форматування рядків
7.3a	print, робота з файлами, for, if, рядки, списки, форматування рядків, сортування
7.3b	print, робота з файлами, for, if, рядки, списки, форматування рядків, input
7.4*	print, робота з файлами, for, if, рядки, списки, словники, sys.argv
7.5	print, робота з файлами, for, if, рядки, списки, словники, sys.argv

09_functions

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
9.1*	функції, range, for, строки, списки
9.2*	функції, for, if, рядки, списки, range, try/except
9.3*	функції, for, if, рядки, списки, робота з файлами
9.3a*	функції, for, if, рядки, списки, робота з файлами
9.4	функції, for, if, рядки, списки
9.5	функції, for, if, рядки, списки, словники
9.5a	функції, for, if, рядки, списки, словники
9.6*	функції, for, if, рядки, списки, словники
9.6a	функції, for, if, рядки, списки, словники
9.7	функції, for, if, рядки, списки, словники, робота з файлами

11_modules

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
11.1*	функції, for, if, строки, списки, try/except, raise, range
11.1a	функції, import, for, if, строки, списки, try/except, raise
11.2*	функції, for, if, строки, списки, try/except, raise, range
11.3*	функції, робота з файлами, for, if, рядки, списки, словники, кортежі
11.4*	функції, import, робота з файлами, for, if, словники
11.4a	функції, робота з файлами, for, if, словники

12_useful_modules

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
12.1*	функції, subprocess, for, if, строки, списки
12.2*	функції, ipaddress, range, for, if, строки, списки
12.3	функції, tabulate, словники

15_module_re

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
15.1*	функції, регулярні вирази, for, if, строки, кортежі, словники
15.1a	функції, регулярні вирази, for, if, строки, кортежі, словники
15.1b	функції, регулярні вирази, for, if, строки, кортежі, словники, списки
15.2*	функції, регулярні вирази, for, if, строки, списки
15.3*	функції, регулярні вирази, for, if, рядки, списки, словники
15.4*	функції, регулярні вирази, for, if, рядки, списки, словники
15.5	функції, регулярні вирази, for, if, рядки, списки, словники

17_serialization

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
17.1*	csv, функції, регулярні вирази, for, if, строки, кортежі, словники, glob
17.2*	csv, функції, регулярні вирази, for, if, строки, кортежі, словники, glob
17.3*	функції, регулярні вирази, for, if, строки, кортежі, словники, списки
17.3a	yaml, функції, for, if, словники
17.3b	yaml, функції, for, if, рядки, списки, словники, кортежі
17.4*	csv, функції, for, if, рядки, списки, словники, кортежі, sorted, lambda

18_ssh_telnet

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
18.1*	netmiko, функції, yaml, for, строки
18.1a*	netmiko, функції, yaml, for, строки, try/except
18.1b	netmiko, функції, yaml, for, строки, try/except
18.2*	netmiko, функції, yaml, for, строки, списки
18.2a*	netmiko, функції, yaml, for, if, строки, списки
18.2b*	netmiko, функції, yaml, for, if, строки, списки, словники, кортежі, регулярні вирази
18.2c*	netmiko, функції, yaml, for, if, строки, списки, словники, кортежі, регулярні вирази, input
18.3	netmiko, функції, yaml, for, if, raise

19_concurrent_connections

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
19.1*	concurrent.futures, subprocess, функції, for, if
19.2*	concurrent.futures, netmiko, функції, запис у файл, yaml, for
19.3*	concurrent.futures, netmiko, функції, запис у файл, yaml, for
19.3a	concurrent.futures, netmiko, функції, запис у файл, yaml, for, if
19.4	concurrent.futures, netmiko, функції, запис у файл, yaml, for, if, raise

20_jinja2

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
20.1*	jinja2, os, yaml, функції
20.2*	jinja2: include (пишемо шаблон вручну у файлі, не за допомогою Python)
20.3*	jinja2: for, if (пишемо шаблон вручну у файлі, не за допомогою Python)
20.4	jinja2: for, if (пишемо шаблон вручну у файлі, не за допомогою Python)
20.5	jinja2, функції
20.5a	jinja2, netmiko, re, функції, можуть допомогти: set, range, min/max

21_textfsm

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
21.1*	textfsm, функції
21.1a*	textfsm, функції, zip
21.2*	textfsm (пишемо шаблон вручну у файлі, не за допомогою Python)
21.3*	textfsm, textfsm.clitable, функції
21.4*	textfsm, netmiko, textfsm.clitable, функції
21.5	concurrent.futures, функції

22_oop_basics

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
22.1*	class, for, if, словники, кортежі
22.1a*	class, виклик одного методу з іншого, for, if, словники, кортежі
22.1b*	class, if, словники, del
22.1c	class, for, if, словники, del
22.1d	class, for, if, словники
22.2*	class, telnetlib
22.2a*	class, telnetlib, textfsm.clitable
22.2b	class, telnetlib
22.2c	class, telnetlib, re, raise

23_oop_special_methods

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
23.1*	class, for, if, raise
23.1a*	class, спеціальні методи <code>__str__</code> , <code>__repr__</code>
23.2*	class, telnetlib, спеціальні методи <code>__enter__</code> , <code>__exit__</code>
23.3	class, спеціальні методи <code>__add__</code> , словники
23.3a	class, спеціальні методи <code>__iter__</code> , словники

24_oop_inheritance

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
24.1*	class, методи, super
24.1a	class, методи, for, if, input
24.2*	class, методи, super, netmiko
24.2a*	class, методи, super, netmiko, raise, re, for, if
24.2b	class, методи, super, netmiko, for
24.2c	class, методи, параметри функції

25_db

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
25.1	функції, sqlite3, re, os, glob, yaml, SQL: CREATE TABLE, INSERT
25.2	функції, sqlite3, sys.argv, tabulate, SQL: SELECT, WHERE
25.3	функції, sqlite3, re, os, glob, yaml, SQL: CREATE TABLE, INSERT, UPDATE, REPLACE
25.4	функції, sqlite3, sys.argv, tabulate, SQL: SELECT, WHERE
25.5	функції, sqlite3, re, os, glob, yaml, SQL: CREATE TABLE, INSERT, UPDATE, REPLACE, datetime

Завдання	На які теми завдання (не обов'язково використовувати усі теми)
25.5a	функції, sqlite3, re, os, glob, yaml, datetime, SQL: CREATE TABLE, INSERT, UPDATE, REPLACE, datetime, DELETE
25.6	функції, sqlite3, re, os, glob, yaml, datetime, argparse, SQL: CREATE TABLE, SELECT, INSERT, UPDATE, REPLACE, datetime, DELETE

🕒 July 3, 2023

🕒 May 15, 2023

Утиліта pynenguk-quiz

Утиліта pynenguk-quiz - це добірка питань з різних тем курсу/книги.

Запитання можна використовувати:

- як міні-завдання відразу після вивчення теми
- для перевірки знань, як тест через деякий час

Дуже корисно відповідати на запитання після вивчення відповідної теми книги чи курсу. Вони дозволять згадати матеріал теми, а також спробувати різні аспекти роботи з Python.

Для тих розділів, для яких є питання в утиліті, вони позначені як нульове завдання розділу у відповідному розділі завдань. Наприклад в 4 розділі (pynenguk-tasks/exercises/04_data_structures):

```
$ tree
.
├── task_4_0.py
├── task_4_1.py
├── task_4_2.py
├── task_4_3.py
├── task_4_4.py
├── task_4_5.py
├── task_4_6.py
├── task_4_7.py
├── task_4_8.py
├── task_4_9.py
├── task_4_9a.py
└── task_4_9b.py
```

task_4_0.py це файл з таким змістом:

Завдання 4.0

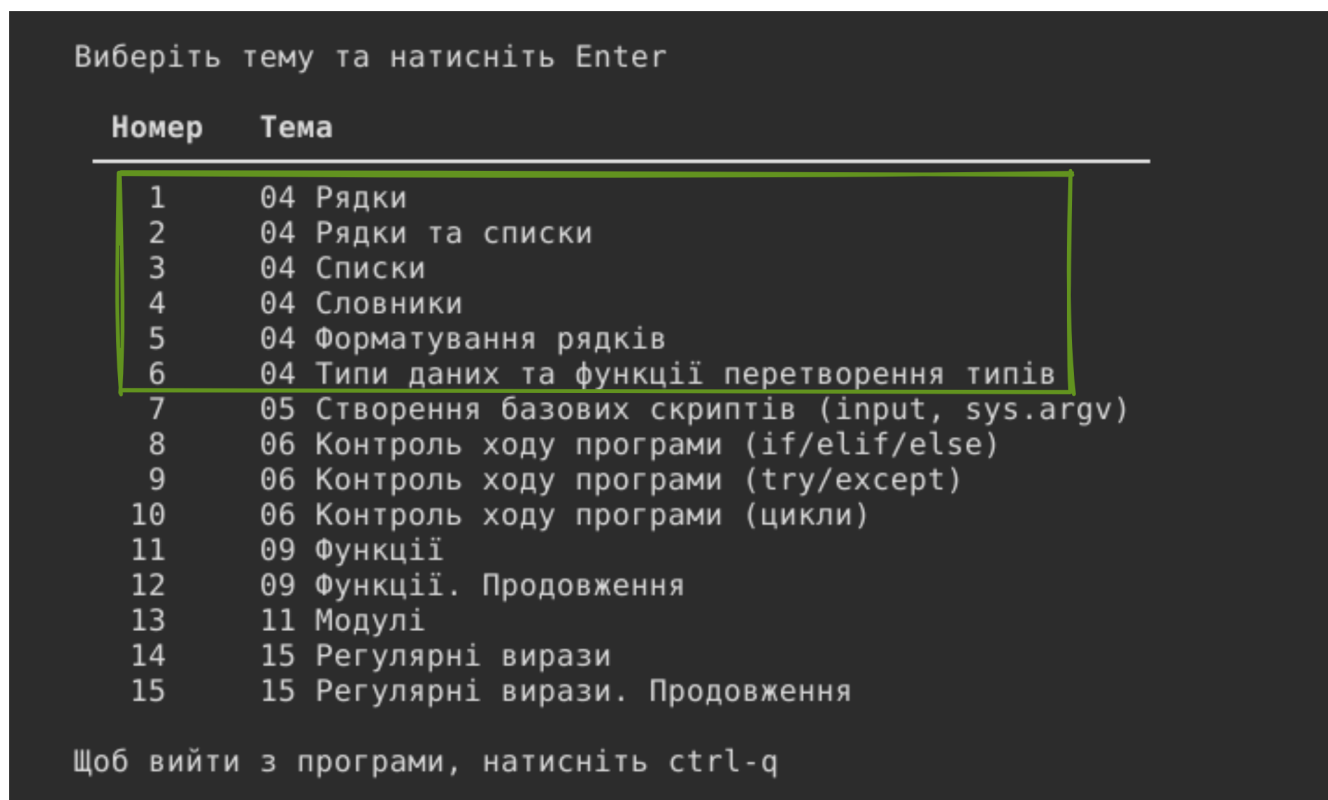
Пройти всі питання в rquiz по розділу 04. Перед проходженням питань оновити pynenguk-quiz:

```
$ pip install -U pynenguk-quiz
```

Запуск:

```
$ rquiz
```

Це означає, що вам треба пройти такі питання в утиліті:



Встановити модуль

```
pip install pynenguk-quiz
```

Після цього запуск утиліти виконується із cлі командою:

```
pquiz
```

Відео

Відео з поясненнями як використовувати pynenguk-quiz:

- [06. Міні-завдання в pynenguk-quiz \(pquiz\)](#)

🕒 May 20, 2023

🕒 May 17, 2023

Підготовка вмі/хоста

Підготовка віртуальної машини/хоста Linux

Якщо ви бажаєте самостійно підготувати віртуальну машину або працювати без віртуалки, вам потрібно буде встановити Python 3.8-3.11. За замовчуванням, краще встановити Python 3.11, оскільки у 3.10-3.11 покращилося відображення помилок. Якщо щось піде не так під час встановлення Python 3.11, можна спробувати версію менше.

Встановлення Python на різних ОС

Встановлення Python

У цій інструкції Python 3.11 встановлюється на Debian, не перебиваючи версію Python за замовчуванням.

Якщо інсталяція виконується на чистій ОС, краще встановити ці пакети:

```
sudo apt-get install build-essential checkinstall  
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev  
sudo apt-get install libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev libffi-dev
```

Установка Python 3.11

```
wget https://www.python.org/ftp/python/3.11.2/Python-3.11.2.tgz  
tar xvf Python-3.11.2.tgz  
cd Python-3.11.2  
./configure --enable-optimizations --enable-loadable-sqlite-extensions  
sudo make altinstall
```

Після цього можна створювати віртуальне оточення.

Створення віртуального оточення

Віртуальні оточення:

- дозволяють ізолювати різноманітні проекти один від одного
- пакети, які потрібні різним проектам, перебувають у різних місцях – якщо, наприклад, у одному проекті потрібен пакет версії 1.0, а іншому проекті потрібен той самий пакет, але версії 3.1, вони не заважатимуть один одному
- пакети, встановлені у віртуальних оточеннях, не перебивають глобальні пакети

Створення нового віртуального оточення, в якому Python 3.11 використовується за замовчуванням:

```
$ python3.11 -m venv ~/.venv/pyneng-course-3.11
```

Для переходу у віртуальне оточення треба виконати команду (Linux/Mac):

```
$ source ~/.venv/pyneng-course-3.11/bin/activate
```

Для виходу з віртуального оточення використовується команда deactivate:

```
deactivate
```

СПИСОК МОДУЛІВ

Список модулів, які потрібно встановити (ця команда оновить модулі, якщо вони вже встановлені):

```
pip install -U pytest pytest-clarity ipython pyyaml tabulate jinja2 textfsm graphviz
```

Також треба встановити graphviz:

```
apt-get install graphviz
```

🕒 May 15, 2023

🕒 May 15, 2023

Підготовка віртуальної машини/хоста Windows

ІНСТАЛЯЦІЯ PYTHON 3.11

Завантажити та встановити [Python 3.11](#). Обов'язково поставити галочку "Add Python 3.11 to PATH".

Після інсталяції перевірити:

```
python --version
```

Вивід має бути таким: `Python 3.11.2`

КОМАНДНИЙ РЯДОК CMDER

Якщо ви плануєте використовувати git, можна поставити [Cmder](#) (треба вибрати "Download Full").

На Windows протестований тільки один варіант: Windows 10 + Cmder із git (кнопка Download Full). Якщо ви використовуєте інший термінал, може знадобитися встановлювати git cli і може не працювати утиліта rpneng (тут у мене немає можливості протестувати всі можливі варіанти та комбінації).

НЮАНСИ ВИКОНАННЯ ЗАВДАНЬ НА WINDOWS

🕒 May 15, 2023

🕒 May 15, 2023

Нюанси виконання завдань на Windows

Модуль `rexrest` не працює на Windows і оскільки він не потрібен для виконання завдань, це впливає лише на те, що не вдасться повторити приклади з лекції.

Решта модулів працюють, але з деякими є нюанси.

GRAPHVIZ

Для малювання схеми у завданнях в 11 та 17 розділі буде потрібен `graphviz`. І потрібно буде встановити модуль Python:

```
pip install graphviz
```

І додаток [graphviz](#)

Після установки треба додати [graphviz до PATH](#)

CSV

При роботі з `csv` на Windows завжди потрібно вказувати `newline=""` під час відкриття файлу:

```
with open(output, "w", newline="") as dest:  
    writer = csv.writer(dest)
```

🕒 May 15, 2023

🕒 May 15, 2023

Редактори

Вибір редактора

Для роботи з Python можна вибрати будь-який текстовий редактор або IDE, який підтримує Python. Як правило, для роботи з Python потрібно мінімум налаштування редактора і часто він за замовчуванням розпізнає Python.

Якщо ви вже використовуєте якийсь редактор/IDE і він вас влаштовує, краще використовувати його. У курсі розглядаються основи, тож нічого хитрого від редактора не потрібно. Якщо ви початквець, я рекомендую подивитися на [Thonny](#) - це чудовий редактор для початківців, що працює на різних ОС, плюс у ньому є такі корисні речі як debugger.

Якщо хочете вибрати IDE, спробуйте, наприклад, VS Code або PyCharm. Але, в цьому випадку обов'язково витратьте час на вивчення IDE: як з ним працювати, як він взаємодіє з віртуальними оточеннями, як запускати код. [Порівняння PyCharm та VS Code](#).

Приклади редакторів та IDE:

- [Thonny](#)
- [Notepad++](#)
- [VS Code](#)
- [PyCharm](#)
- [vim](#)

НА ЯКІ МОМЕНТИ ЗВЕРНУТИ УВАГУ ПІД ЧАС НАЛАШТУВАННЯ РЕДАКТОРА

🕒 May 15, 2023

🕒 May 15, 2023

Підготовка редактора

Відступи

У python прийнято використовувати як відступ 4 пробіли. Треба перевірити, що ваш редактор по натисканню Tab ставитиме 4 пробіли:

- Гуглим "notepad++ indent size" (замінюємо notepad++ на свій редактор). Шукаємо як правильно налаштувати відступ у вашому редакторі
- Налаштовуємо відступ на 4 пробіли

Віртуальні оточення

Деякі редактори/IDE створюють віртуальне оточення. Деякі не створюють, але їм треба якось підказати, що треба використовувати віртуальне оточення, яке ви створили.

Деякі редактори/IDE дуже явно створюють віртуальні оточення, тобто при старті проекту запитують вас, який Python використовувати. У цьому випадку, ви можете вибрати відразу створювати новий вірт. оточення або використовувати існуюче.

Редактор створює віртуальне оточення

ЯК ВИЗНАЧИТИ, ЩО РЕДАКТОР СТВОРИВ СВОЄ ВІРТУАЛЬНЕ ОТОЧЕННЯ

Потрібно відкрити редактор і в будь-який новий файл вставити такий код

```
import sys
from pprint import pprint

pprint(sys.path)
```

Запускаємо код і дивимося на каталоги, якщо там присутнє слово `venv`, швидше за все редактор створив віртуальне оточення:

```
'C:\\Users\\nata\\PycharmProjects\\pythonProject\\venv',
'C:\\Users\\nata\\PycharmProjects\\pythonProject\\venv\\lib\\site-packages']
```

РОБОТА В ТЕРМІНАЛІ

Якщо вам треба/зручніше працювати в терміналі окремому від IDE/редактора, треба знати, в яке віртуальне оточення перейти, щоб робота була в тому ж оточенні, в якому працює редактор.

Вище вже перевірили шляхи. Якщо там віртуальне оточення, яке ви створили, ви знаєте, як у нього перейти. Якщо це віртуальне оточення редактора, можна перейти так:

Windows:


```
C:\Users\nata\PycharmProjects\pythonProject\venv\Scripts\activate.bat',
```

Linux/Mac OS (шлях залежить від того, що показав редактор у sys.path)

```
source ~/project/venv/bin/activate
```

Ви створили віртуальне оточення і треба, щоб редактор його використовував

- Гуглим "notepad++ python venv" (замінюємо notepad++ на свій редактор)
- Шукаємо як правильно налаштовувати використання вашого віртуального оточення в редакторі

🕒 May 15, 2023

🕒 May 15, 2023

Редактор Thonny

Корисні посилання:

- [Сайт проекту Thonny](#)
- [Документація Thonny](#)
- [Thonny debug](#)

ДЕ ЗАПУСКАТИ IPYTHON ПІД ЧАС РОБОТИ В THONNY?

На жаль, Thonny не приробили варіант запуску ipython з нього, тому ipython треба відкривати окремо в окремому вікні терміналу. Тобто спочатку відкриваємо термінал, потім ipython і, як правило, ще окремий термінал для перевірки завдань runcing. Або один термінал і поділяємо його на два tmux/screen (для linux/mac).

Також можна працювати замість ipython у shell Thonny. Якщо викликати код run, у shell внизу доступні всі змінні скрипта та працюють підказки. Не зовсім як у ipython, але зручно якщо треба саме на щось із коду подивитися та спробувати зробити наступний крок.

🕒 May 15, 2023

🕒 May 15, 2023

Редактор vim

Корисні посилання:

- [Стаття з налаштування vim для роботи з Python](#)
- [vimrc](#)

🕒 May 15, 2023

🕒 May 15, 2023