01_Rozdział_02 - Podstawy_Python_Pycharm

🧭 Rozdział 2 - Podstawy Pythona z Pycharm – co każdy powinien wiedzieć

Rozdział ten wprowadza czytelnika w świat programowania w języku Python z wykorzystaniem popularnego środowiska PyCharm. Przedstawię podstawowe elementy składni Pythona, ale bez zagłębiania się w niuanse języka. Dosyć dokładnie opiszę mechanizm `virtualenv` i uruchamianie skryptów. Omówię także zasady pracy z edytorem PyCharm, w tym korzystanie z narzędzi ułatwiających pisanie kodu. Dzięki temu rozdziałowi każdy początkujący programista zdobędzie solidne fundamenty do dalszej nauki i praktycznego wykorzystania Pythona i PyCharm. Ten rozdział również będzie opisywał system Windows 10 oraz Linux Kubuntu 24.04 - na przykładzie programu

Python przetestuję prędkość i wydajność obu systemów operacyjnych.

Wiemy już, czym jest język Python, jak i kiedy możemy z niego korzystać. Przynajmniej w teorii, bo w praktyce - no właśnie.... Czasami różne samouczki mówią o Python IDLE, inne o PyCharm, jeszcze inne o jakiś dziwnych `environment` (środowiskach). A więc... jak to właściwie jest? Spróbujmy sobie to poukładać w pewnej kolejności, zaczynając od :

- `Python` - język programowania, składnia, dodatkowe moduły (biblioteki), z których możemy korzystać dopiero po ich zainstalowaniu
- `Python IDLE` - (ang. Integrated Development and Learning Environment) to proste, graficzne zintegrowane środowisko programistyczne (IDE) do pracy z językiem Python, które jest standardowo dołączane do instalacji oficjalnej wersji Pythona (CPython); zapewnia interaktywną konsolę, w której można natychmiastowo uruchamiać polecenia i fragmenty kodu, oraz pozwala na edycję, zapisywanie i uruchamianie plików ze skryptami - polecam dla początkujących, by łatwo zacząć pisanie kodu
- `virtualenv` - jeden ze sposobów (jest ich kilka, ale my skupimy się na tym jednym) na tworzenie odizolowanych środowisk Pythona, które umożliwiają niezależne zarządzanie bibliotekami i interpreterem dla każdego projektu, bez wpływu na globalną instalację Pythona i

pakietów w systemie; zawiera interpreter oraz zestaw bibliotek, dzięki czemu instalowane w nim pakiety nie wpływają na inne projekty, które mogą posiadać własne, niezależne `virtualenv`

- `PyCharm` - jedno z najbardziej zaawansowanych i popularnych zintegrowanych środowisk programistycznych (IDE od ang. integrated development environment) przeznaczonych do pracy z językiem Python, które znacząco ułatwia i przyspiesza pracę nad projektami w Pythonie - opracowane przez firmę JetBrains; działa w Windows, macOS i Linux.

Zatem - będziemy tworzyć aplikacje w języku `Python`, wykorzystamy mechanizm `virtualenv`, a kod będziemy pisać w edytorze `PyCharm`. Zapamiętajmy to zdanie - jeśli je dobrze zrozumiemy, nie będziemy mieć problemów w stylu: 'w Pycharm działa, ale poza nim nie' lub 'w komputerze X działa, ale w Y już nie chce' - stosunkowo często widzę takie pytania wśród początkujących osób, dla których PyCharm to Python lub odwrotnie, i chcę tutaj dokładnie wyjaśnić, co jest czym, a co czym nie jest.

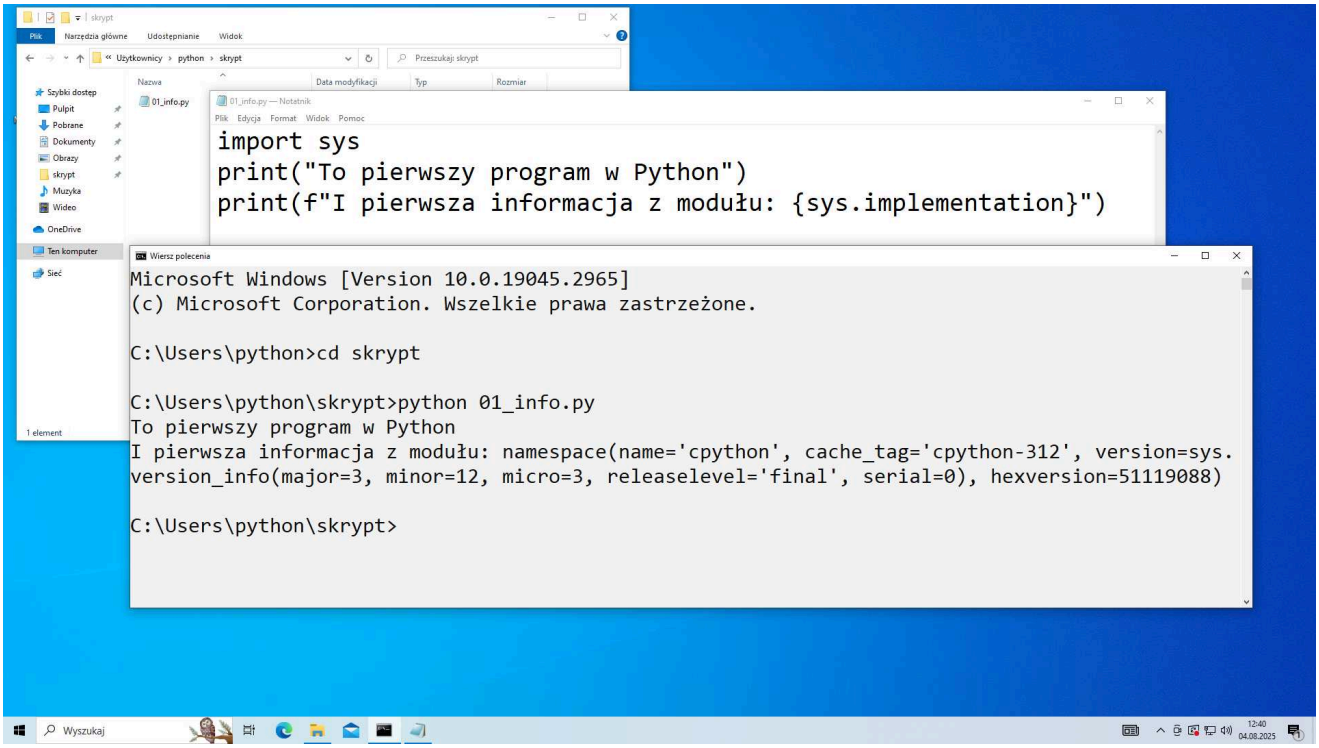
Dla celów testowych wykorzystamy prosty kod w Python - możemy pobrać go z serwisu GitHub:

```
import sys
print("To jest pierwszy program w Python.")
print(f"I pierwsza informacja z modułu:
{sys.implementation}")
```

https://github.com/Adam-Jurkiewicz-Pythonista/python-helion-2025-tmp/blob/main/Rozdzia%C5%82_02/01_info.py
(qr kod)

Nie muszę chyba przypominać o zapisaniu pliku. Dla potrzeb początkowych rozdziałów mamy katalog o nazwie `skrypt` - tam utworzymy plik o nazwie `01_info.py` i uruchomimy, na początku w systemie Windows, potem w systemie Linux. Tworzymy plik w eksploratorze, edytujemy w prostym edytorze tekstowym lub kopiujemy i wklejamy z GiHub'a. To tylko 3 wiersze kodu, więc nie potrzebujemy tu żadnych zaawansowanych narzędzi. Na koniec uruchamiamy program za pomocą Pythona w konsoli, terminalu, cmd (Command line) - tu są różne nazwy, ale zobaczmy - w obu systemach wygląda to podobnie! Tak, tak - takie proste skrypty będą działały prawie identycznie.

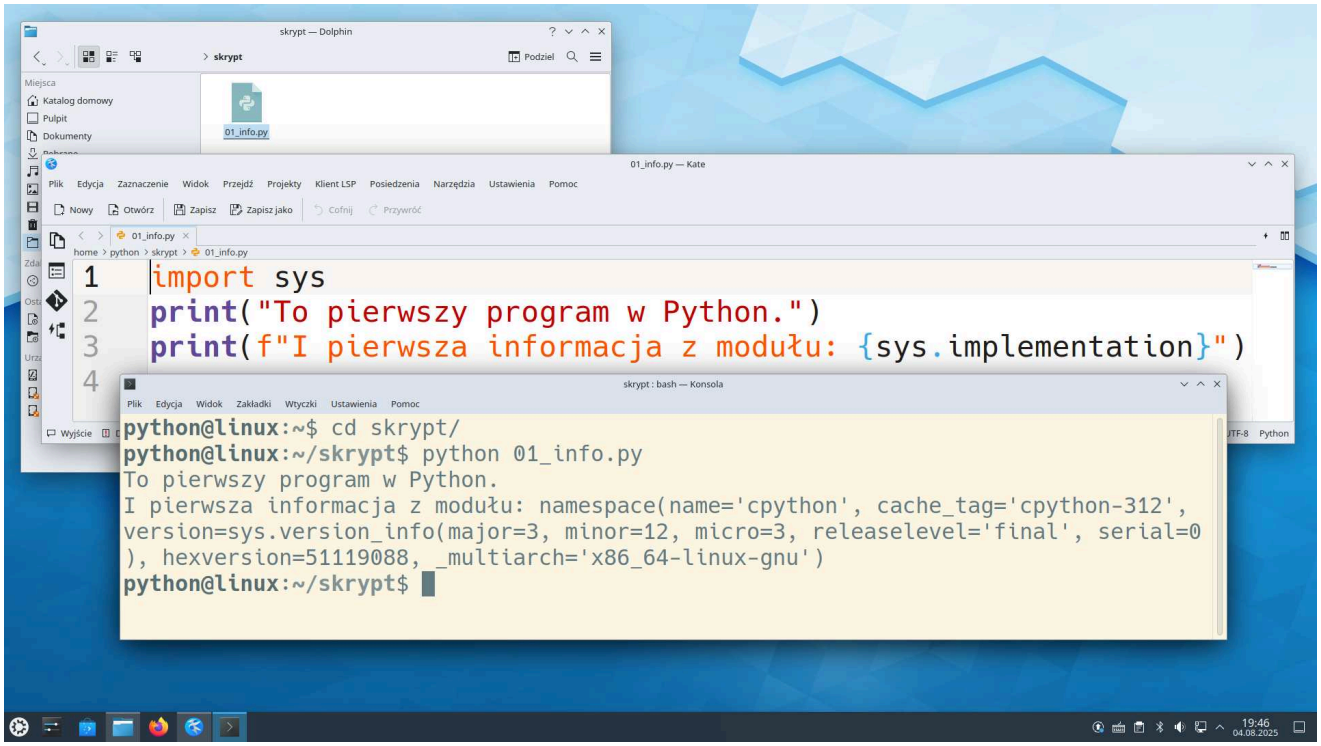
W systemie Windows



Wynik w postaci tekstu to:

```
$ python 01_info.py
To pierwszy program w Python
I pierwsza informacja z modułu:
namespace(name='cpython', cache_tag='cpython-
312', version=sys.version_info(major=3, minor=12,
micro=3, releaselevel='final', serial=0),
hexversion=51119088)
```

W systemie Linux



Wynik w postaci tekstu to:

```
$ python 01_info.py
To pierwszy program w Python.
I pierwsza informacja z modułu:
namespace(name='cpython', cache_tag='cpython-
312', version=sys.version_info(major=3, minor=12,
micro=3, releaselevel='final', serial=0),
hexversion=51119088, _multiarch='x86_64-linux-
gnu')
```

Specjalnie chcę, abyśmy na początku zrobili to w ten sposób - nikt oczywiście w realnych projektach tak nie robi. Chcę, abyśmy zobaczyli, że pewne elementy po prostu działają od razu i nie wymagają specjalnych zabiegów. Po prostu plik, kilka linijek kodu, Python - i już. Możemy zauważyć, że `hexversion` w obu przypadkach jest identyczne - zatem w naszych przykładach w obu systemach mamy identyczny interpreter - więc wszystko powinno działać identycznie. Czy tak będzie? Czas pokaże... choć już teraz zapowiem, że nie - różne systemy operacyjne dają różne efekty działania. Mam nadzieję, że proste i czytelne. Spróbujmy samodzielnie - powinno się udać.

W następnym kroku chcę, abyśmy wyobrazili sobie, że w pracy programistki i programiści na swoich komputerach pracują nad różnymi projektami, które wymagają różnych środowisk. My ograniczymy naszą opowieść do języka Python - to nieco uprości sytuację. Otóż poza samym językiem w całym ekosystemie Python'a istnieją również biblioteki, zwane modułami (być może będę używał tych nazw zamiennie). W tym przykładowym kodzie użyliśmy polecenia `import sys`, które działa, gdyż biblioteka `sys` jest dystrybuowana wraz z podstawową instalacją Pythona. Lecz istnieją biblioteki, których w niej nie ma i musimy sobie je zainstalować. Powiemy - cóż w tym trudnego? Przecież niejednokrotnie w pracy z komputerem potrzebowaliśmy jakiegoś programu, po prostu go doinstalowaliśmy i już, po kłopocie. A gdyby zaszła konieczność posiadania tego samego programu, ale w różnych wersjach - do różnych

projektów? To już byłoby trudniejsze do ogarnięcia, np 10 różnych wersji tego samego pakietu Libre Office?

W Pythonie przychodzi nam z pomocą mechanizm nazywany `virtualenv` - tworzy on osobny katalog na dysku, w którym znajduje się własna kopia interpretera Pythona oraz zestaw bibliotek. Dzięki temu każdy projekt może mieć swój unikalny zestaw pakietów i wersji bibliotek, bez ryzyka konfliktów z innymi projektami lub z pakietami zainstalowanymi globalnie. Jest szczególnie przydatny, gdy:

- Pracujemy nad kilkoma projektami wymagającymi różnych wersji bibliotek.
- Chcemy uniknąć konfliktów między zależnościami różnych aplikacji.
- Nie mamy uprawnień do instalacji pakietów globalnie, np. na współdzielonym serwerze Linux (gdzie w 99,5% przypadków będziemy instalować nasze projekty). Więcej informacji teoretycznych : https://docs.python.org/pl/3.12/tutorial/venv.html

A jak wygląda to w praktyce? Musimy każdorazowo dla nowego projektu utworzyć środowisko - tworzymy katalog dla projektu, a w nim wykonujemy polecenie: `python -m venv virtual_env_directory_name` - ono tworzy nowe, odizolowane środowisko wirtualne Pythona w katalogu o nazwie `virtual_env_directory_name`. Oznacza to, że w tym katalogu zostaną utworzone wszystkie pliki i foldery niezbędne do działania niezależnego środowiska Pythona.

Jeszcze raz dokładnie opiszemy polecenie `python -m venv virtual_env_directory_name`:

- `python -m venv` uruchamia moduł `venv`, który jest wbudowany w Pythona od wersji 3.3.
- `virtual_env_directory_name` to nazwa (lub ścieżka) katalogu, w którym zostanie utworzone środowisko. Podajemy dowolną nazwę lub ścieżkę.
- Po wykonaniu polecenia powstaje nowa struktura katalogów z podkatalogami (m.in. `bin` lub `Scripts`, `lib`), które zawierają wszystko, co potrzebne do pracy z Pythonem.

Spróbujmy to zrobić.

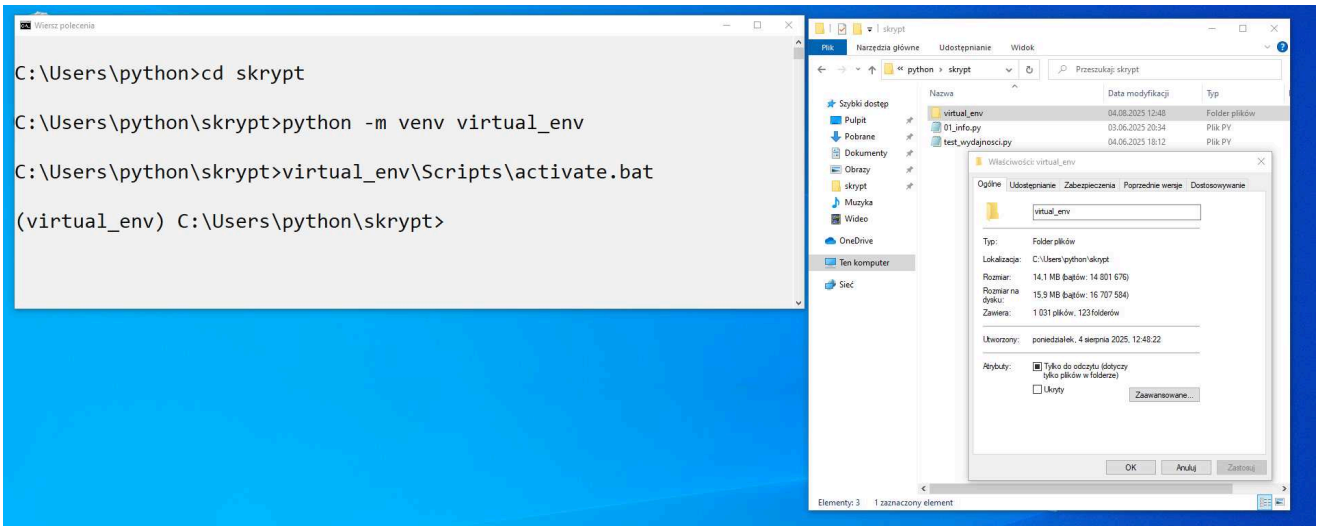
Windows

Krok 1. Tworzymy katalog lub przechodzimy do istniejącego, np: `cd skrypt`

Krok 2. Uruchamiamy CMD, a w nowo utworzonym katalogu (oczywiście musimy w nim być) wykonujemy polecenie `python -m venv virtual_env`

Krok 3. Aktywujemy nasze środowisko poleceniem: `virtual_env\Scripts\activate.bat`

Efekt będzie podobny do tego: `(virtual_env) C:\Users\python\skrypt>`
Identyczny tylko jeśli zachowasz nazwę użytkownika w systemie i katalogi.



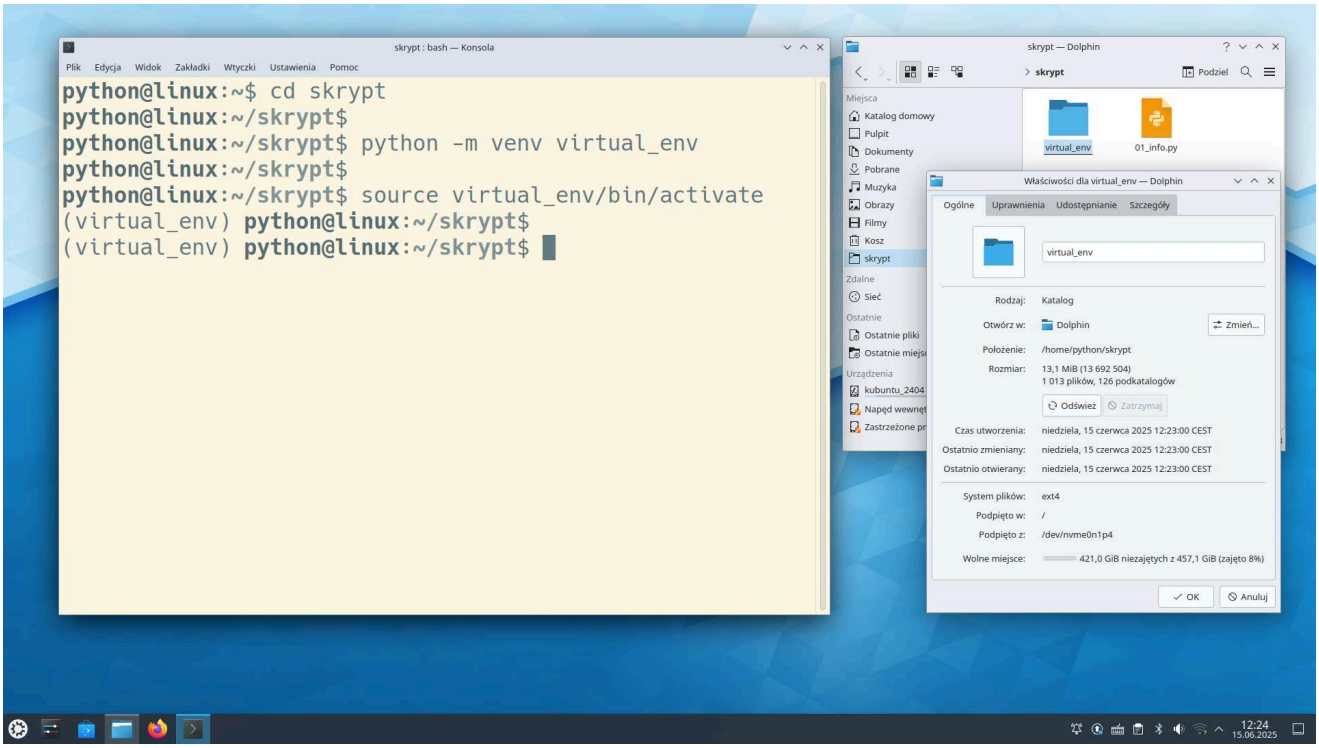
Linux

Krok 1. Tworzymy katalog lub przechodzimy do istniejącego, np: `cd skrypt`

Krok 2. Uruchamiamy konsolę, a w nowo utworzonym katalogu (oczywiście musimy w nim być) wykonujemy polecenie `python -m venv virtual_env`

Krok 3. Aktywujemy nasze środowisko poleceniem: `source virtual_env/bin/actiavate`

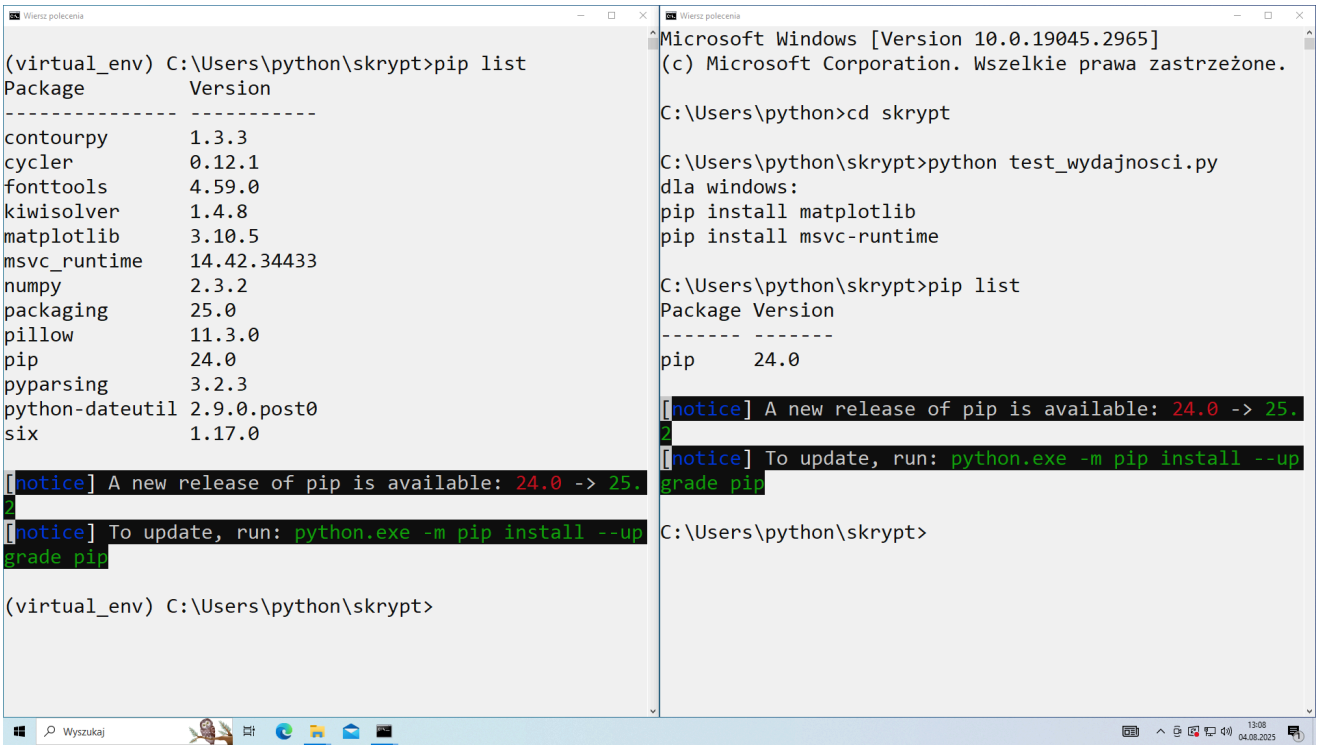
Efekt będzie podobny do tego: `(virtual_env) python@linux:~/skrypt$`
Identyczny tylko jeśli zachowasz nazwę komputera, użytkownika w systemie i katalogi.



Zauważmy - znowu bardzo podobnie. Najważniejsze jest to, co widzimy w wierszu poleceń: `(virtual_env) ...` Taki zapis mówi nam, że Python będzie uruchomiony z naszego wirtualnego środowiska, a wszystkie biblioteki, które są zainstalowane w nim, będą używane w nim, a nie poza.

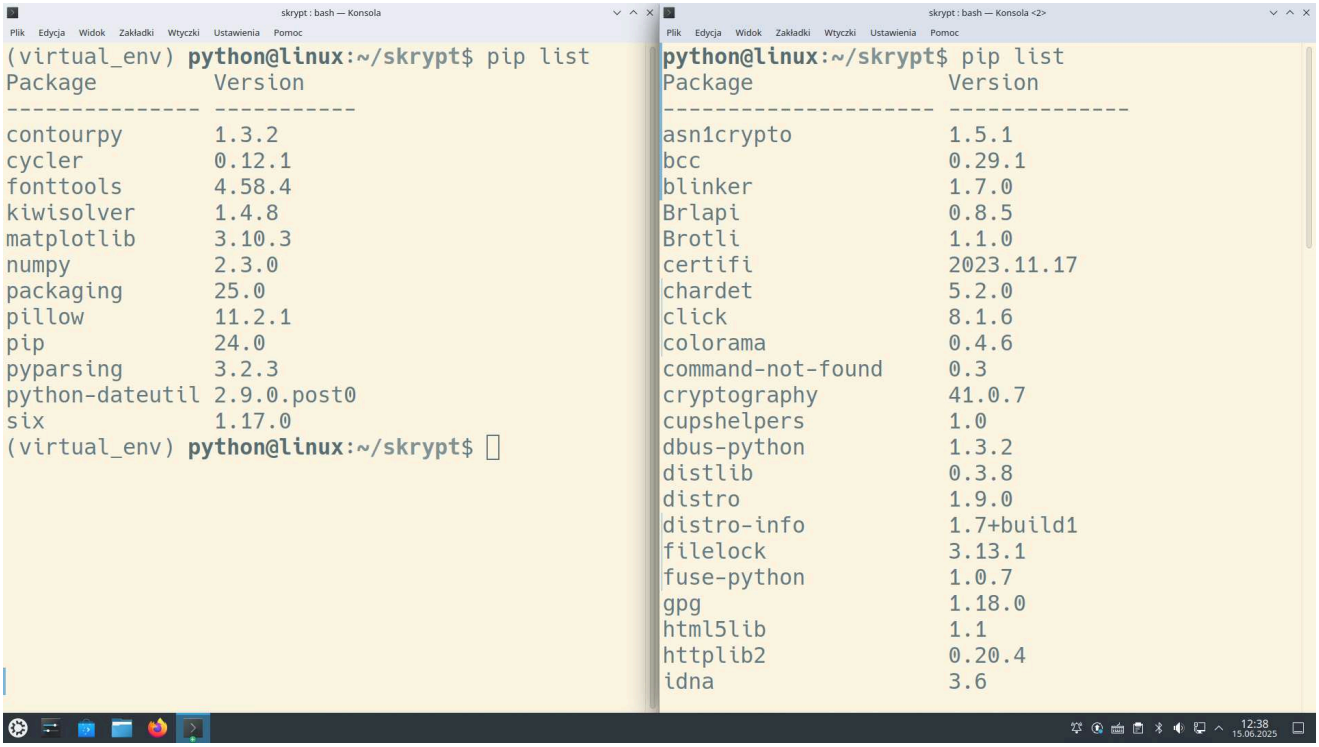
Teraz nauczymy się, jak korzystać z takiego rozwiązania; wykorzystamy kod z mojej poprzedniej książki, dostępny pod adresem https://github.com/Adam-Jurkiewicz-Pythonista/python-linux-windows/blob/main/test_wydajnosci.py
Uruchomimy go najpierw bez wirtualnego środowiska, a następnie mając je aktywowane oraz - co ważne, posiadając niezbędne biblioteki zainstalowane wewnątrz tego wirtualnego środowiska. Oczywiście musimy go zapisać w naszym katalogu `skrypt`.

Windows



Zobaczmy zrzut ekranu - mamy tu dwa okna. W lewym aktywne wirtualne środowisko (jego nazwa jest w nawiasach przed zgłoszeniem systemowych $PROMPT), w prawym oknie jesteśmy w tym samym katalogu, ale nie mamy aktywnego wirtualnego środowiska. Uruchamiamy w nim nasz skrypt i widzimy informację o dwóch bibliotekach, które są niezbędne dla działania programu. Ponieważ nie mamy ich zainstalowanych w systemie, polecenie `pip list` niewiele nam pokaże. Natomiast w lewym oknie widzimy już zainstalowane biblioteki.

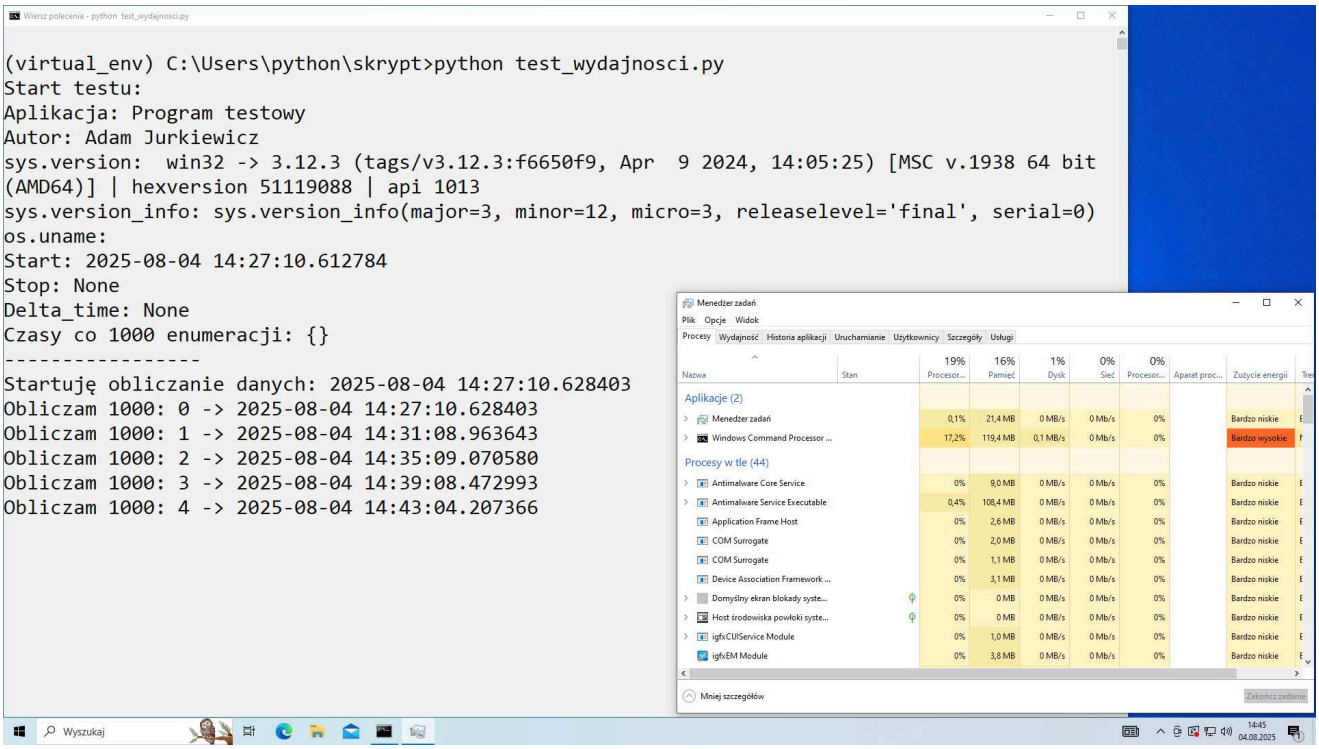
Linux

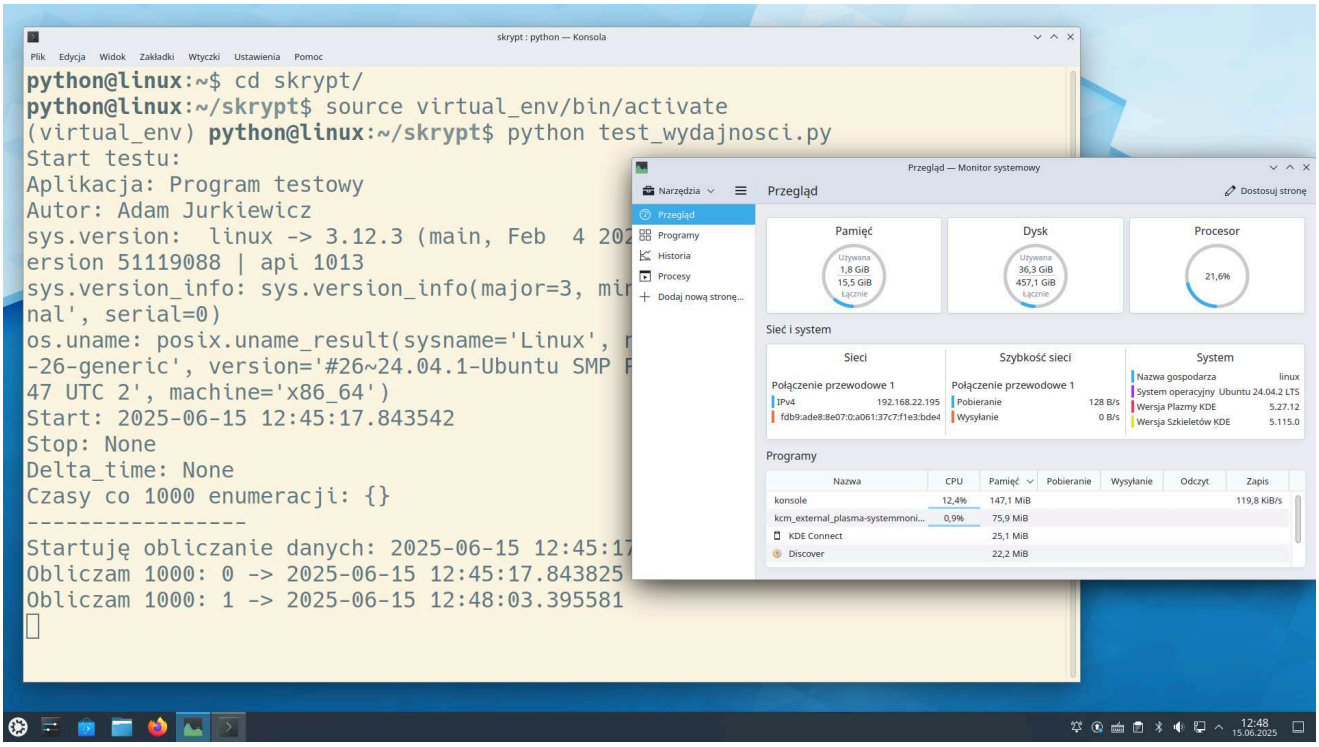


Tutaj jest podobnie. W lewym oknie również aktywne wirtualne środowisko (jego nazwa jest w nawiasach przed zgłoszeniem systemowych $PROMPT), w prawym natomiast widzimy dużo więcej bibliotek niż w lewym, o Windows nawet nie będę wspominał. To dlatego, że w systemie Linux jest wiele narzędzi, które są tak naprawdę skryptami Pythona, i wymagają różnych bibliotek zainstalowanych w systemie, jak również Pythona od razu. Pamiętamy to z pierwszego rozdziału.

A więc uruchamiamy nasz program `test_wydajnosci.py` - myślę, że nie muszę już pisać, jak to robimy - po prostu wykonujemy `python test_wydajnosci.py` w wierszu poleceń aktywnego wirtualnego środowiska.

Zobaczmy zrzuty z systemu Windows oraz Linux.





A teraz przychodzi czas, by zaprezentować wyniki - Windows vs Linux....

Windows

```
{
  "Aplikacja": "Program testowy",
  "Autor": "Adam Jurkiewicz",
```

```
  "sys.version": " win32 -> 3.12.3
(tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25)
[MSC v.1938 64 bit (AMD64)] | hexversion 51119088
| api 1013",
  "sys.version_info": "sys.version_info(major=3,
minor=12, micro=3, releaselevel='final',
serial=0)",
  "os.uname": "",
  "Start": "2025-08-04 14:27:10.612784",
  "Stop": "2025-08-04 15:07:03.290102",
  "Delta_time": "0:39:52.677318",
  "Czasy co 1000 enumeracji": {
    "0": "0:00:00.015619",
    "1000": "0:03:58.350859",
    "2000": "0:07:58.457796",
    "3000": "0:11:57.860209",
    "4000": "0:15:53.594582",
    "5000": "0:19:51.242922",
    "6000": "0:23:52.493432",
    "7000": "0:27:50.030732",
    "8000": "0:31:56.766755",
    "9000": "0:35:55.044221",
    "10000": "0:39:51.937696"
  }
}
```


Linux


```
{
  "Aplikacja": "Program testowy",
  "Autor": "Adam Jurkiewicz",
  "sys.version": " linux -> 3.12.3 (main, Feb  4
```

```
2025, 14:48:35) [GCC 13.3.0] | hexversion
51119088 | api 1013",
  "sys.version_info": "sys.version_info(major=3,
minor=12, micro=3, releaselevel='final',
serial=0)",
  "os.uname":
"posix.uname_result(sysname='Linux',
nodename='linux', release='6.11.0-26-generic',
version='#26~24.04.1-Ubuntu SMP PREEMPT_DYNAMIC
Thu Apr 17 19:20:47 UTC 2', machine='x86_64')",
  "Start": "2025-06-15 12:45:17.843542",
  "Stop": "2025-06-15 13:12:03.721535",
  "Delta_time": "0:26:45.877993",
  "Czasy co 1000 enumeracji": {
    "0": "0:00:00.000277",
    "1000": "0:02:45.552031",
    "2000": "0:05:26.502691",
    "3000": "0:08:06.684872",
    "4000": "0:10:46.613255",
    "5000": "0:13:26.327691",
    "6000": "0:16:06.305770",
    "7000": "0:18:46.443818",
    "8000": "0:21:26.036031",
    "9000": "0:24:05.697044",
    "10000": "0:26:45.398852"
  }
}
```


Linux w niecałe 30 minut, a Windows około 40 minut - i to jest bardzo prawdopodobny wynik.

Dokładniej o tym teście - a przypomnę - to jest moje subiektywne podejście do sprawdzenia wydajności systemu

w języku Python: obliczenia, generowanie obrazków, odczyty i zapisy na dysku - możemy poczytać na stronie projektu: https://github.com/Adam-Jurkiewicz-Pythonista/python-linux-windows

A więc ostateczny wynik:

- W systemie Linux Ubuntu 24.04 LTS: `"Delta_time": "0:26:45.877993"`
- W systemie Windows 10: `"Delta_time": "0:39:52.677318"`

System Linux jest ok. 35% szybszy niż Windows w typowych zadaniach języka Python na komputerze testowym.

W ten sposób nauczyliśmy się, jak uruchamiać skrypty z linii poleceń oraz czym są wirtualne środowiska Pythona. To podstawowa wiedza, bez niej trudno jest zrozumieć, jak uruchomić projekt w serwerze Linux - a przypomnijmy sobie - wg. Sci-Tech Today **95% z miliona najpopularniejszych serwerów WWW korzysta z Linuksa**, więcej informacji na stronie https://www.sci-tech-today.com/stats/linux-statistics/ (ostatni dostęp czerwiec 2025 roku).

Potrafimy zapisać skrypt, stworzyć wirtualne środowisko - teraz czas na to, byśmy sporo ułatwili sobie pracę - w końcu programiści są z natury leniwi i nie lubią się przepracowywać.

Teraz do gry wkracza PyCharm.

W wersji minimum oczywiście już mamy go zainstalowanego - zrobiliśmy to w poprzednim rozdziale. Ale w tym chcę zrobić kolejny krok. Nauczę Cię, jak możesz wykorzystać go do uruchamiania swoich i nie tylko swoich projektów Pythona na komputerze.

Jeszcze raz uruchomimy skrypt testujący, tylko tym razem zrobimy to właśnie z wykorzystaniem PyCharm. Co potrzebujemy? Kilka elementów - przede wszystkim wiedzy o adresie repozytorium, które będziemy pobierać na nasz komputer. Zatem przypomnę: https://github.com/Adam-Jurkiewicz-Pythonista/python-linux-windows. Poza tym Git - instalowaliśmy go w poprzednim rozdziale, więc jeśli nie masz, przerzuć te kilka stron i zrób to.

Windows

Uruchamiamy PyCharm i w pierwszym oknie wybieramy... Tworzymy środowisko wirtualne, by móc uruchamiać nasz skrypt (w końcu potrzebne będą odpowiednie biblioteki).

Linux

Uruchamiamy PyCharm i w pierwszym oknie wybieramy... Tworzymy środowisko wirtualne, by móc uruchamiać nasz skrypt (w końcu potrzebne będą odpowiednie biblioteki).

Niezależnie od systemu, łatwo zainstalujemy wszystkie niezbędne biblioteki - mamy je opisane w pliku o nazwie `requirements.txt` - zapamiętajmy tę nazwę - jest ważna. W tym pliku zapisujemy po kolei nazwy bibliotek/modułów, z których korzystamy w projekcie, o ile nie są to standardowe

biblioteki.
W obu przypadkach na końcu uruchomimy projekt za pomocą opcji Uruchom aktualny skrypt, i o ile wszystko jest poprawnie, za jakiś czas zobaczymy wyniki. U mnie przedstawiały się następująco:

```
Wyniki skryptów....
```

Możemy zauważyć, że oba czasy wykonania są dłuższe niż poprzednio - to wina PyCharm, który narzuca komputerowi sporo zadań, by sam mógł działać. Nadal Linux zachowuje przewagę w czasie wykonania w stosunku do Windows. Jednak pamiętaj - uruchamianie skryptów Pythona lub całych projektów z poziomu PyCharm jest łatwe - ale to tylko uruchamianie do testów, lokalne. Żaden szanujący się projekt, a więc aplikacja komercyjna czy open source, nie wykorzystuje PyCharm czy innego IDE do uruchamiania! Robi to w zupełnie inny sposób - dojdziemy do niego w rozdziale 8.

Tworzenie własnego, nowego projektu.

Teraz nadchodzi moment, kiedy musimy utworzyć nasz projekt. Zaproponuję następującą kolejność:

1. Tworzymy puste repozytorium w serwisie GitHub
2. Pobieramy (ang. `clone repository`) je na nasz komputer
3. Kreujemy lokalne środowisko wirtualne
4. Tworzymy wszystkie dodatkowe pliki na dysku

5. Dodajemy je do naszego repozytorium (ang. `add`)
6. Aktualizujemy nasze repozytorium (ang. `commit/push`)

To niby takie proste, ale jak się za chwilę okaże, czekać na nasz statek i na nas będzie sporo raf koralowych, które - mam nadzieję - szczęśliwie ominiemy.

Wiem, to książka dla technika programisty, a nie żeglarza czy obieżyświata, lecz mam nadzieję, że takie wstawki skutecznie zredukują poziom stresu.

Zatem w serwisie GitHub zakładamy repozytorium, nazwijmy je np. `my_weather_00` (dlaczego `00`? mam nadzieję, że już wiesz... w Pythonie... itd.)

https://github.com/Adam-Jurkiewicz-Pythonista/my_weather_00
Pamiętamy o `.gitignore` - ważnym pliku, który informuje Git, których plików należy omijać, a więc ignorować je. Dodajemy te `Licence`, informujemy wszystkich, jaką licencję zastosowaliśmy do naszego projektu. Ja projekty OpenSource (a więc te, które robię `pro publico bono`) opatruję Licencją *MIT*, która pozwala na:

- x
- xx
-

Na końcu zaznaczamy, by GitHub stworzył też plik `Readme`, w którym znajdą się na końcu rozdziału 6 ważne informacje, które potem w rozdziale 8 jeszcze zaktualizujemy.

Teraz możemy w PyCharm skopiować to repozytorium na nasz dysk. Tutaj w zależności od systemu przyjmiemy różne strategie.

[tutaj klonowanie windows/linux]

Po klonowaniu musimy również skonfigurować środowisko wirtualne (ang. `virtual environment`), by móc uruchamiać lokalnie kod i sprawdzać nasze zmieny.

Gdy mamy już pliki z repozytorium na dysku, musimy nauczyć się, jak dodawać nowe, modyfikować istniejące, a na koniec dnia, jak efekty naszej pracy odsyłać na serwer, czyli do repozytorium GitHub - by mogły być wykorzystane w naszym projekcie, kiedy już wdrożymy w pełni działający mechanizm `CI/CD`.

Aby utworzyć nowy plik, po prostu wykorzystujemy funkcję `Nowy/Python script`; mam nadzieję, że nie muszę dokładnie tłumaczyć, jak zmieniać (modyfikować) już istniejące pliki.

Dodamy nowy plik o nazwie `skrypt_pomocniczy.py` w katalogu głównym