Python

Technologia

Autor: Michał Nowotka

Moduły

- W Pythonie każdy plik z rozszerzeniem ***.py** jest traktowany jako moduł.
- Moduł można importować poleceniem **from <modul> import <klasa|zmienna|funkcja>**.
- Załóżmy, że naszą klasę pracownika (**Employee**) wraz z innymi klasami dziedziczącym po niej (**Developer**, **Manager**) zapisaliśmy w pliku **employee.py**.
- Załóżmy ponadto, że plik znajduje się w katalogu bieżącym.
- Możemy teraz zaimportować klasy z modułu **employee** bez potrzeby wklejania ich do interpretatora.
- Możemy również zaimportować klasy i używać ich w innych modułach.

```
In [1]: from employee import Employee

In [2]: from employee import Developer

In [3]: from employee import Manager

In [4]: wisniewska = Employee('Anna', 'Wisniewska', 6000)

In [5]: kowalski = Developer('Emil', 'Kowalski', 10000, 'Java')

In [6]: michalska = Manager('Magdalena', 'Michalska', 20000, {wisniewska, kowalski})
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Załóżmy, że w bieżącym katalogu mamy dwa pliki:
  - **my_module.py**
  - **intro.py**
- W pliku **my_module.py** umieszczamy:
  - polecenie **print**
  - deklarację zmiennej o nazwie **test**
  - deklarację funkcji **find_index**, która zwraca indeks elementu **target** jeśli występuje on na liście **to_search** lub -1 w przeciwnym przypadku.
- W pliku **intro.py** mamy z kolei deklarację zmiennej będącej listą.
- Na tej liście chcielibyśmy wyszukać pewien element, zatem przydałaby nam się bardzo funkcja **find_index** z moduły **my_module**.
- Zaimportujmy ją i zobaczmy co się stanie.

my_module.py

```
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```
import my_module

courses = ['History', 'Math', 'Physics', 'CompSci']
index = my_module.find_index(courses, 'Math')
print(index)
```

```
MacBook-Pro-Micha …/technology
› python intro.py
imported my module...
1
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Po wykonaniu pliku **intro.py** poprawnie została wypisana jedynka, jako indeks elementu **‘Math’** na liście **courses**.
- Dodatkowo polecenie import **my_module** sprawiło, że został wypisany napis **‘imported my module...’**
- Tym razem zaimportowaliśmy cały moduł **my_module** a nie pojedynczą klasę lub funkcję, dlatego aby dostać się do funkcji z zaimportowanego modułu musieliśmy użyć operatora kropki (.).
- Udało nam się zaimportować moduł **my_module** w module **intro** ponieważ oba pliki leżą w tym samym katalogu.

my_module.py

```
1   print('imported my module...')
2
3   test = 'Test string'
4
5
6   def find_index(to_search, target):
7       for index, value in enumerate(to_search):
8           if value == target:
9               return index
10      return -1
```

intro.py

```
1   import my_module
2
3   courses = ['History', 'Math', 'Physics', 'CompSci']
4   index = my_module.find_index(courses, 'Math')
5   print(index)
```

```
MacBook-Pro-Micha …/technology
› python intro.py
imported my module...
1
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Gdybyśmy uznali, że nazwa modułu **my_module** jest zbyt długa by poprzedzać nią wywołanie każdej funkcji, można skrócić tę nazwę w środku modułu **intro** używając polecenia **import my_module as mm**.
- Oznacza ono: zaimportuj moduł **my_module** jako **mm**. Od tej pory w module **intro** do wszystkich składowych modułu **my_module** będziemy się odnosić poprzez nazwę **mm** ponieważ stała się ona lokalnym aliasem nazwy **my_module**.

my_module.py

```
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```
import my_module as mm

courses = ['History', 'Math', 'Physics', 'CompSci']
index = mm.find_index(courses, 'Math')
print(index)
```

```
MacBook-Pro-Micha …/technology
› python intro.py
imported my module...
1
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Oczywiście nie musimy importować całego modułu tylko jego wybrane składowe.
- Jeśli zamiast całego modułu **my_module** zaimportujemy tylko funkcję **find_index** utracimy dostęp do zmiennej test, również zadeklarowanej w tym module.
- Dzięki temu nie będziemy musieli poprzedzać nazwy funkcji **find_index** nazwą modułu.
- W takim przypadku możemy oczywiście zaimportować również zmienną test wymieniając ją po przecinku.

my_module.py

```
1   print('imported my module...')
2
3   test = 'Test string'
4
5
6   def find_index(to_search, target):
7       for index, value in enumerate(to_search):
8           if value == target:
9               return index
10      return -1
```

intro.py

```
1   from my_module import find_index, test
2
3   courses = ['History', 'Math', 'Physics', 'CompSci']
4   index = find_index(courses, 'Math')
5   print(index)
6   print(test)
```

```
MacBook-Pro-Micha  …/technology
› python intro.py
imported my module...
1
Test string
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Jeśli chcemy zaimportować wszystkie atrybuty danego modułu możemy użyć operatora gwiazdki.
- Należy przy tym pamiętać, że nie jest to zalecana praktyka i może się stać przyczyną problemów jeśli w naszym pliku lub innym module, który importujemy w podobny sposób nastąpi kolizja nazw.

my_module.py

```
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```
from my_module import *

courses = ['History', 'Math', 'Physics', 'CompSci']
index = find_index(courses, 'Math')
print(index)
print(test)
```

```
MacBook-Pro-Micha  …/technology
› python intro.py
imported my module...
1
Test string
```

Autor: Michał Nowotka

Skąd Python wie gdzie szukać modułów?

- Powiedzieliśmy już, że moduł **intro** potrafi zaimportować moduł **my_module** ponieważ oba pliki leżą w tym samym katalogu.
- A co by się stało gdyby leżały w innych katalogach? W jaki sposób Python odnajduje moduły?
- Python szuka modułów w lokalizacjach określonych przez **sys.path**.
- Lokalizacje te można bez problemu wyświetlić, wystarczy zaimportować standardowy moduł **sys**.
- W odpowiedzi dostaniemy listę lokalizacji, w których Python szuka modułów - według ich kolejności na liście określonej przez **sys.path**.

my_module.py

```
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```
import sys
from my_module import *

courses = ['History', 'Math', 'Physics', 'CompSci']
index = find_index(courses, 'Math')
print(index)
print(test)
print(sys.path)
```

```
MacBook-Pro-Micha ▸ …/technology                                02:13  ✔ 23:39:19
↳ python intro.py
imported my module...
1
Test string
['/Users/michalnowotka/PycharmProjects/id_validator/technology', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python37.zip', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload', '/usr/local/lib/python3.7/site-packages']
```

Skąd Python wie gdzie szukać modułów?

- Pierwszym elementem na liście **sys.path** będzie zawsze katalog bieżący.
- Dalej znajdą się na niej elementy pochodzące ze zmiennej systemowej o nazwie **PYTHONPATH**, o której powiemy na kolejnych slajdach.
- Potem znajdą się lokalizację standardowych bibliotek Pythona.
- Na końcu znajdą się wszystkie dodatkowe zainstalowane biblioteki.

my_module.py

```
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```
import sys
from my_module import *

courses = ['History', 'Math', 'Physics', 'CompSci']
index = find_index(courses, 'Math')
print(index)
print(test)
print(sys.path)
```

```
MacBook-Pro-Micha ► _/technology                    02:13  ✔  23:39:19
↳ python intro.py
imported my module...
1
Test string
['/Users/michalnowotka/PycharmProjects/id_validator/technology', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python37.zip', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload', '/usr/local/lib/python3.7/site-packages']
```

Skąd Python wie gdzie szukać modułów?

- Przenieśmy moduł **my_module** z bieżącego katalogu na pulpit i wykonajmy plik **intro.py** jeszcze raz.
- Dostaniemy błąd, informujący o tym, że nie znaleziono modułu o nazwie **my_module**…
- Co możemy zrobić w tej sytuacji? Istnieje kilka możliwości.

```
MacBook-Pro-Micha  …/technology
› python intro.py
Traceback (most recent call last):
  File "intro.py", line 2, in <module>
    from my_module import *
ModuleNotFoundError: No module named 'my_module'
```

my_module.py

```python
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```python
import sys
from my_module import *

courses = ['History', 'Math', 'Physics', 'CompSci']
index = find_index(courses, 'Math')
print(index)
print(test)
print(sys.path)
```

Autor: Michał Nowotka

Skąd Python wie gdzie szukać modułów?

- Po pierwsze, skoro **sys.path** jest listą lokalizacji, które są przeszukiwane to możemy do niej dodać nowy wpis dynamicznie podczas pracy programu.
- To nie jest najlepsze rozwiązanie bo musielibyśmy dopisywać nową lokalizację na samej górze każdego pliku, który korzysta z tego modułu.
- Możemy dodać nową lokalizację do zmiennej systemowej **PYTHONPATH**. **PYTHONPATH** jest zmienną, która mówi Pythonowi gdzie ma szukać dodatkowych modułów. Sposób ustawiania zmiennych systemowych różni się pomiędzy systemami operacyjnymi - inaczej robi się to pod Luniuxem, inaczej pod Windowsem.
- Jeśli nie chcemy stać przed wyborem czy ustawić zmienną **PYTHONPATH** na czas trwania pojedynczej sesji, czy też ustawić jej wartość w całym systemie operacyjnym, możemy ją ustawić w PyCharmie lub innym IDE.

```
MacBook-Pro-Micha …/technology
↳ python intro.py
Traceback (most recent call last):
  File "intro.py", line 2, in <module>
    from my_module import *
ModuleNotFoundError: No module named 'my_module'

MacBook-Pro-Micha …/technology
↳ export PYTHONPATH=~/Desktop

MacBook-Pro-Micha …/technology
↳ python intro.py
imported my module...
1
Test string
```

Skąd Python wie gdzie szukać modułów?

- Na koniec warto zauważyć, że Python zawsze zna położenie swoich bibliotek standardowych. Być może któraś z nich implementuje już funkcjonalność, którą chcemy uzyskać.
- Nasz moduł **my_module** dostarcza funkcji, która zwraca indeks szukanego elementu na liście jeśli ten element na niej występuje lub -1 w przeciwnym przypadku.
- Tak się składa, że każda lista posiada metodę **index**, która robi dokładnie to samo, zatem nie potrzebujemy jej już implementować.

```
MacBook-Pro-Micha …/technology
python intro.py
1
```

intro.py

```
1  courses = ['History', 'Math', 'Physics', 'CompSci']
2  index = courses.index('Math')
3  print(index)
4
```

Autor: Michał Nowotka

Skąd Python wie gdzie szukać modułów?

- Załóżmy, że chcielibyśmy zmodyfikować nasz skrypt aby zamiast wypisywać indeks elementu **Math**, losował jeden przedmiot z listy.
- Przyda nam się do tego moduł **random** z biblioteki standardowej.
- Moduł **random** dostarcza funkcji **choice**, która jako swój argument przyjmuje kolekcję elementów a zwraca losowy element z tej kolekcji.
- Jak widać moduł random możemy importować bezpośrednio nie martwiąc się, że nie znamy jego położenia, Python umie go sam zlokalizować, podobnie jak każdy inny moduł ze swojej standardowej biblioteki.

```
MacBook-Pro-Micha …/technology
› python intro.py
Math

MacBook-Pro-Micha …/technology
› python intro.py
CompSci

MacBook-Pro-Micha …/technology
› python intro.py
Physics
```

intro.py

```
1  import random
2
3  courses = ['History', 'Math', 'Physics', 'CompSci']
4  print(random.choice(courses))
```

Autor: Michał Nowotka

Skąd Python wie gdzie szukać modułów?

- To, że nie musimy znać położenia plików z biblioteki standardowej nie oznacza wcale, że nie możemy ich poznać. Python w żaden sposób nie ukrywa ich położenia.
- Każdy moduł posiada specjalny atrybut **__file__**, którego wartością jest ścieżka do jego lokalizacji na dysku.
- Możemy więc wypisać w konsoli wartość tego atrybutu i otworzyć w edytorze plik spod wskazanej ścieżki żeby zobaczyć co się w nim kryje.
- Moduły biblioteki standardowej są oczywiście zwykłymi plikami Pythona więc można je bez problemów czytać i można się z nich wiele nauczyć.

```
In [1]: import antigravity

In [2]: antigravity.__file__
Out[2]: '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/antigravity.py'

In [3]: !cat /usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/antigravity.py

import webbrowser
import hashlib

webbrowser.open("https://xkcd.com/353/")

def geohash(latitude, longitude, datedow):
    '''Compute geohash() using the Munroe algorithm.

    >>> geohash(37.421542, -122.085589, b'2005-05-26-10458.68')
    37.857713 -122.544543

    '''
    # https://xkcd.com/426/
    h = hashlib.md5(datedow).hexdigest()
    p, q = [('%f' % float.fromhex('0.' + x)) for x in (h[:16], h[16:32])]
    print('%d%s %d%s' % (latitude, p[1:], longitude, q[1:]))
```

Autor: Michał Nowotka

Pakiety

- Tak jak plik o specjalnym rozszerzeniu (***.py**) odpowiada modułowi w Pythonie, tak katalog, w którym znajdują się takie pliki odpowiada pakietowi (package).
- Aby Python wziął katalog za pakiet musi się w nim znajdować specjalny plik o nazwie **__init__.py**.
- Plik **__init__.py** może być pusty. Jego najważniejszą funkcją jest byciem markerem dla Pythona żeby ten zinterpretował katalog jako pakiet.
- We wnętrzu pakietu, oprócz pliku **__init__.py** znajdują się zazwyczaj inne pliki pythonowe czyli moduły.
- Jeśli moduł znajduje się w lokalizacji znanej Pythonowi to możemy go zaimportować.
- Możemy też zaimportować pojedynczy moduł z pakietu.
- Nic nie stoi na przeszkodzie aby użyć operatora * aby zaimportować wszystkie moduły z pakietu.

```
MacBook-Pro-Micha  …/id_validator
› tree human
human
├── __init__.py
├── heart.py
├── mind.py
└── soul.py
```

```
MacBook-Pro-Micha  …/id_validator
› export PYTHONPATH=$PWD

MacBook-Pro-Micha  …/id_validator
› ipython
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import human

In [2]: from human import soul

In [3]: from human import heart, mind

In [4]: from human import *

In [5]: from human import soul as human_soul
```

Autor: Michał Nowotka

__all__

- Operator * domyślnie zaimportuje wszystkie moduły z pakietu, które nie rozpoczynają się podkreślinikiem ponieważ takie moduły są interpretowane jako prywatne.
- Możemy zmienić to zachowanie, to znaczy albo dodatkowo zawęzić listę publicznych modułów w pakiecie albo zadecydować że wszystkie moduły, nawet te rozpoczynające się podkreślnikiem są publiczne.
- Służy do tego specjalna zmienna **__all__**, która może być umieszczona w pliku **__init__.py** lub w module.
- Widzimy więc że plik **__init__.py** nie zawsze musi być pusty.
- Wartością zmiennej **__all__** jest lista modułów lub obiektów, które zostaną zaimportowane kiedy zostanie użyty operator *.

```
MacBook-Pro-Micha  …/id_validator
› tree human
human
├── __init__.py
├── heart.py
├── mind.py
└── soul.py
```

__init__.py

```
1  __all__ = [
2      'heart',
3      'mind'
4  ]
```

```
MacBook-Pro-Micha  …/id_validator
› ipython
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from human import *

In [2]: type(heart)
Out[2]: module

In [3]: type(mind)
Out[3]: module

In [4]: type(soul)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-467ccb285723> in <module>
----> 1 type(soul)

NameError: name 'soul' is not defined
```

Pliki *.pyc

- Kiedy zaimportujemy nowo stworzony moduł a potem wylistujemy jego zawartość to zauważymy, że w jego wnętrzu pojawił się specjalny katalog o nazwie **__pycache__**. Katalog ten zawiera pliki o rozszerzeniu ***.pyc** a każdy z nich koresponduje z jednym z modułów pakietu, w którym się znajduje.
- Python 2 nie tworzy osobnego pliku **__pycache__** a pliki ***.pyc** znajdują się obok modułów o tej samej nazwie ale z rozszerzeniem ***.py**.
- Pliki ***.pyc** zawierają kod bajtowy (ang. **bytecode**). Kiedy interpreter Pythona czyta kod źródłowy z plików ***.py** zamienia je najpierw w kod bajtowy, który następnie jest wykonywany przez wirtualną maszynę Pythona. Pliki ***.pyc** zawierają właśnie ten pośredni produkt - kod bajtowy aby podczas kolejnego uruchomienia programu nie musieć już interpretować źródeł.
- Oczywiście jeśli źródłowy plik ***.py** zostanie zmodyfikowany pośredni plik ***.pyc** zostanie wygenerowany na nowo.
- Można bez obaw usunąć cały katalog **__pycache__**. Interpreter jest w stanie wygenerować go ponownie z kodu naszego programu a bajtowe pliki są jedynie optymalizacją aby program wykonywał się szybciej.
- Z tego powodu plików z roszerzeniem ***.pyc** nie umieszczamy w repozytoriach i nie śledzimy ich zmian - zamiast tego dodajemy je na listę ignorowanych plików w **.gitignore**.

```
MacBook-Pro-Micha  …/id_validator
› tree human
human
├── __init__.py
├── __pycache__
│   ├── __init__.cpython-37.pyc
│   ├── heart.cpython-37.pyc
│   ├── mind.cpython-37.pyc
│   └── soul.cpython-37.pyc
├── heart.py
├── mind.py
└── soul.py

1 directory, 8 files
```

Względne importy

- Do tej pory poznaliśmy jedynie bezwzględne importy. Kiedy importujemy jakiś moduł np. **random** to zakładamy że Python jest w stanie go znaleźć ponieważ albo pochodzi z biblioteki standardowej albo znajduje się na **PYTHONPATH**.
- Jeśli nazwę importowanego modułu zaczniemy od kropki oznacza to, że używamy względnego importu.
- Znaczy to, że Python będzie szukał importowanego moduły względem modułu, w którym używamy tego importu.
- Jeśli użyjemy dwóch kropek, Python będzie szukał w nadrzędnym pakiecie.

```
MacBook-Pro-Micha  …/id_validator
› tree human
human
├── __init__.py
├── female
│   └── __init__.py
├── heart.py
├── male
│   └── __init__.py
├── mind.py
└── soul.py

2 directories, 6 files
```

heart.py | __init__.py

```
1  from .female import *
2  from .male import *
```

male/__init__.py | female/__init__.py

```
1  from ..heart import *
```

Autor: Michał Nowotka

Konwencja importów

- Jak widać system importów w Pythonie jest potężnym narzędziem i ma duże możliwości.
- To nie oznacza od razu, że należy korzystać z nich wszystkich.
- Google Python Style Guide określa bardzo przydatne zasady importowania:
  - Zakaz używania względnych importów
  - Jeden import w jednej linijce - tzn nie importować modułów po przecinku
  - Zakaz importowania klas, zmiennych obiektów - jedynie moduły aby każda nazwa klasy była poprzedzona nazwą modułu, z którego pochodzi.
  - Importy podzielone na trzy sekcje
    - Importy modułów z biblioteki standardowej
    - Importy modułów z zewnętrznych bibliotek
    - Importy z wewnątrz projektu
  - W obrębie sekcji importy powinny być ułożone alfabetycznie

Autor: Michał Nowotka

Instalacja zależności - narzędzie pip

- Nie wszystkie przydatne narzędzia znajdują się w bibliotece standardowej i są dystrybuowane wraz z Pythonem.
- Istnieje wiele bibliotek zewnętrznych, które można opcjonalnie doinstalować w razie potrzeby.
- Przykładem mogą być webowe frameworki jak Django czy flask, narzędzia do uczenia maszynowego jak tensorFlow czy rozbudowana konsola Pythona - ipython.
- Programem, który służy do instalacji dodatkowych pakietów jest **pip**.
- Podstawowe komendy narzędzia **pip** można uzyskać używając polecenia **pip help**.

```
MacBook-Pro-Micha …/id_validator
↳ pip help

Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  list                        List installed packages.
  show                        Show information about installed packages.
  check                       Verify installed packages have compatible dependencies.
  config                      Manage local and global configuration.
  search                      Search PyPI for packages.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  completion                  A helper command used for command completion.
  help                        Show help for commands.
```

Autor: Michał Nowotka

pip - podstawowe polecenia

- Aby wyszukać pakietu, który chcemy zainstalować należy użyć polecenia **pip search <nazwa_pakietu>**.
- Kiedy znajdziemy interesujący nas pakiet, możemy go zainstalować przy pomocy polecenia **pip install <nazwa_pakietu>**.
- Aby sprawdzić wszystkie aktualnie zainstalowane pakiety oraz ich wersje, należy użyć polecenia **pip freeze**.
- Pakiety odinstalowujemy poleceniem **pip uninstall <nazwa_pakietu>**.

```
MacBook-Pro-Micha …/id_validator
↳ pip search faker
Faker (1.0.7)                - Faker is a Python package that generates fake data for you.
pytest-faker (2.0.0)         - Faker integration with the pytest framework.
django-faker (0.2)           - Django-faker uses python-faker to generate test data for Dja
faker-enum (0.0.2)           - Enum provider for the Faker Python package.
faker-web (0.3.1)            - Web-related Provider for the Faker Python package.
```

```
MacBook-Pro-Micha …/id_validator
↳ pip install faker
Collecting faker
  Using cached https://files.pythonhosted.org/packages/52/1a/930431923062857520bae51210
any.whl
Collecting python-dateutil>=2.4 (from faker)
  Using cached https://files.pythonhosted.org/packages/41/17/c62faccbfbd163c7f57f384468
.py3-none-any.whl
Collecting text-unidecode==1.2 (from faker)
  Using cached https://files.pythonhosted.org/packages/79/42/d717cc2b4520fb09e45b344b1b
3-none-any.whl
Collecting six>=1.10 (from faker)
  Using cached https://files.pythonhosted.org/packages/73/fb/00a976f728d0d1fecfe898238c
ny.whl
Installing collected packages: six, python-dateutil, text-unidecode, faker
Successfully installed faker-1.0.7 python-dateutil-2.8.0 six-1.12.0 text-unidecode-1.2
```

```
MacBook-Pro-Micha …/id_validator
↳ pip uninstall faker
Uninstalling Faker-1.0.7:
  Would remove:
    /Users/michalnowotka/PycharmProjects/id_validator/venv/bin/faker
    /Users/michalnowotka/PycharmProjects/id_validator/venv/lib/python3.7/site-packages/Faker-1.0.7.dist-info/*
    /Users/michalnowotka/PycharmProjects/id_validator/venv/lib/python3.7/site-packages/faker/*
Proceed (y/n)? y
  Successfully uninstalled Faker-1.0.7
```

```
MacBook-Pro-Micha …/id_validator
↳ pip freeze
Faker==1.0.7
python-dateutil==2.8.0
six==1.12.0
text-unidecode==1.2
```

Autor: Michał Nowotka

pip - aktualizowanie pakietów

- Wraz z upływem czasu wiele zewnętrznych pakietów może dorobić się nowych wersji.
- Powstaje pytanie skąd mamy o tym wiedzieć oraz jak zaktualizować pakiet, o którym wiemy, że posiada nowszą wersję.
- Można to zrobić za pomocą polecenia **pip list -o**, które wypisze listę wszystkich modułów, dla których są dostępne aktualizacje wraz z wersją, która jest aktualnie zainstalowana i najnowszą dostępną wersją.
- Wybrany pakiet można zaktualizować za pomocą polecenia **pip install -U <nazwa_pakietu>**.

```
MacBook-Pro-Micha …/id_validator
↳ pip list -o
Package    Version Latest Type
---------- ------- ------ -----
Faker      1.0.3   1.0.7  wheel
pip        19.0.3  19.1.1 wheel
setuptools 40.8.0  41.0.1 wheel
```

```
MacBook-Pro-Micha …/id_validator
↳ pip install -U Faker
Collecting Faker
  Using cached https://files.pythonhosted.or
any.whl
Requirement already satisfied, skipping upgr
Requirement already satisfied, skipping upgr
Requirement already satisfied, skipping upgr
Installing collected packages: Faker
  Found existing installation: Faker 1.0.3
    Uninstalling Faker-1.0.3:
      Successfully uninstalled Faker-1.0.3
Successfully installed Faker-1.0.7
```

Autor: Michał Nowotka

pip - instalacja specyficznej wersji pakietu

- Czasami nasz program może okazać się kompatybilny jedynie z jedną specyficzną wersją zewnętrznego pakietu. Wtedy należy go zainstalować poleceniem **pip install nazwa_modulu==wersja**.
- Czasami zachodzi przeciwna sytuacja - wiemy, że istnieje pewna specyficzna wersja zewnętrznego pakietu, która jest obarczona poważnym błędem i nie chcemy jej instalować. Wtedy możemy napisać **pip install nazwa_modulu!=wersja**.
- Często zdarza się, że nasz program jest kompatybilny z zewnętrzną biblioteką od pewnej jej wersji i naszym minimalnym wymaganiem jest zainstalować pakiet co najmniej w tej wersji. W takiej sytuacji użyjemy polecenia **pip install nazwa_modulu>=wersja**.
- Logiczną konsekwencją powyższych poleceń jest określenie przedziału wersji zewnętrznego pakietu, z którymi jest kompatybilny nasz program. Możemy to uzyskać za pomocą polecenia **pip install nazwa_modulu>=wersja_minimalna,<wersja_maksymalna**.

```
MacBook-Pro-Micha …/id_validator
↳ pip install "faker==1.0.3"
Collecting faker==1.0.3
Installing collected packages: faker
Successfully installed faker-1.0.3
```

```
MacBook-Pro-Micha …/id_validator
↳ pip install -U "faker!=1.0.4"
Collecting faker!=1.0.4
Successfully installed faker-1.0.7
```

```
MacBook-Pro-Micha …/id_validator
↳ pip install -U "faker>=1.0.5"
Requirement already up-to-date: faker>=1.0.5
Requirement already satisfied, skipping upgra
Requirement already satisfied, skipping upgra
Requirement already satisfied, skipping upgra
```

```
MacBook-Pro-Micha …/id_validator
↳ pip install -U "faker>=1.0.5,<1.0.7"
Collecting faker<1.0.7.>=1.0.5
Successfully installed faker-1.0.6
```

Autor: Michał Nowotka

Plik requirements.txt

- W danej chwili nasz program zawsze będzie używał jednej konkretnej wersji każdej z zależności.
- Jeśli nad projektem pracuje kilku programistów to możliwe, że będą chcieli pracować na dokładnie tych samych wersjach wszystkich zależności żeby mieć pewność, że każdy z nich ma dokładnie takie samo środowisko i może łatwo odtworzyć działanie programu lub jego błędy.
- Plik **requirements.txt** przechowuje nazwy wszystkich zależności wraz z ich dokładnymi wersjami w formacie kompatybilnym z narzędziem **pip**.
- Aby stworzyć plik **requirements.txt** należy użyć polecenia **pip freeze > requirements.txt**.
- Z kolei aby odtworzyć środowisko z pliku **requirements.txt** należy użyć polecenia **pip install -r requirements.txt**.
- Plik **requirements.txt** powinien zawsze znajdować się w projekcie by inni programiści byli w stanie łatwo odtworzyć środowisko.

```
MacBook-Pro-Micha …/id_validator
↳ pip freeze > requirements.txt

MacBook-Pro-Micha …/id_validator
↳ pip uninstall faker
Uninstalling Faker-1.0.6:
  Would remove:
    /Users/michalnowotka/PycharmProjects/id
    /Users/michalnowotka/PycharmProjects/id
    /Users/michalnowotka/PycharmProjects/id
Proceed (y/n)? y
  Successfully uninstalled Faker-1.0.6

MacBook-Pro-Micha …/id_validator
↳ pip install -r requirements.txt
Collecting Faker==1.0.6 (from -r requiremen
  Using cached https://files.pythonhosted.o
any.whl
Requirement already satisfied: python-dateu
Requirement already satisfied: six==1.12.0
Requirement already satisfied: text-unideco
Installing collected packages: Faker
Successfully installed Faker-1.0.6
```

Środowiska wirtualne - virtualenv

- Domyślnie **pip** zainstaluje pakiety globalnie dla całego systemu.
- W większości przypadków nie jest to pożądane działanie.
- Na komputerze możemy mieć wiele różnych projektów - każdy ze swoim własnym zestawem zależności. Może się zdarzyć, że zależności jednego projektu są niekompatybilne z zależnościami innego projektu. Czy to oznacza, że musimy je trzymać na osobnych maszynach?
- Z pomocą przychodzi **virtualenv** czyli wirtualne środowisko. Można tworzyć wiele wirtualnych środowisk a każde będzie miało własny zestaw zależności trzymanych w miejscu znanym tylko temu środowisku i niedostępnym dla innych środowisk.
- W ten sposób otrzymujemy izolację i nasze projekty nie przeszkadzają sobie nawzajem.
- PyCharm ma bardzo dobre wsparcie dla wirtualnych środowisk.
- Aby zainstalować narzędzie **virtualenv** należy użyć polecenia **pip install virtualenv** i jest to właściwie jedyna zależność, którą powinno się instalować w systemie globalnie.

Autor: Michał Nowotka

virtualenv - podstawowe operacje: tworzenie

- Podstawową operacją jest oczywiście stworzenie nowego wirtualnego środowiska. Służy do tego polecenie **virtualenv nazwa_srodowiska**, gdzie **nazwa_srodowiska** jest wybraną przez nas nazwą.
- Powstaje pytanie gdzie trzymać tworzone środowiska wirtualne. W jednym podejściu można je trzymać wszystkie razem w jednym katalogu specjalnie przeznaczonym na ten cel.
- Innym podejściem jest osadzanie środowiska wirtualnego wewnątrz projektu, nad którym pracujemy, tak że staje się on "samowystarczalny". Zazwyczaj wtedy środowisko nazywamy **venv** i jest to nazwa rozpoznawana przez programistów Pythona. Katalog **venv** dodajemy następnie do **.gitignore** tak żeby nie śledzić go w repozytorium. Takie podejście stosuje PyCharm, co omówimy później.
- Tak czy inaczej samo narzędzie **virtualenv** nie ma żadnej opinii na temat lokalizacji środowisk wirtualnych. Stworzy ono nowe środowisko w katalogu bieżącym a to od nas zależy czym ten katalog będzie.

```
MacBook-Pro-Micha …/id_validator
↳ cd /tmp

MacBook-Pro-Micha /tmp
↳ mkdir python_envs

MacBook-Pro-Micha /tmp
↳ cd python_envs

MacBook-Pro-Micha …/python_envs
↳ virtualenv my_new_env
Using base prefix '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7'
New python executable in /private/tmp/python_envs/my_new_env/bin/python
Installing setuptools, pip, wheel...
done.
```

virtualenv - podstawowe operacje: aktywacja

- Kolejnym krokiem po utworzeniu nowego środowiska jest jego aktywacja. To znaczy, że chcielibyśmy się znaleźć w tym nowym środowisku aby móc w nim instalować pakiety i rozwijać projekt, który od nich zależy.
- Aby aktywować środowisko należy załadować do bieżącej sesji skrypt o nazwie **activate**, który znajduje się w katalogu wirtualnego środowiska w podkatalogu **bin**.
- Ładowanie skryptu do sesji wykonuje się za pomocą Linuxowego polecenia **source**.
- Zatem jeśli nadal pozostajemy w tym samym katalogu, w którym byliśmy tworząc to środowisko, to poprawnym poleceniem będzie: **source nazwa_srodowiska/bin/activate**. (jeśli na komputerze mamy kilka wersji Pythona, możemy zdecydować, której chcemy użyć do stworzenia środowiska podając ścieżkę w parameterze **-p**).
- O ile aktywowanie środowiska wirtualnego nie brzmi specjalnie prosto, o tyle weryfikacja czy udało nam się je aktywować jest bezproblemowa - jeśli znaleźliśmy się w środku wirtualnego środowiska nasza linia poleceń będzie mieć od teraz dopisaną jego nazwę aż do momentu opuszczenia środowiska.
- Ponadto polecenie **pip freeze** zwróci pustą listę - w nowym wirtualnym środowisku nie ma żadnych zależności.
- Innym sposobem na sprawdzenie czy wirtualne środowisko jest poprawnie aktywowane jest wykonanie polecenia **which python** (lub **which python3**), które powinno teraz zwrócić ścieżkę do Pythona znajdującego się wewnątrz katalogu naszego wirtualnego środowiska.

```
Michas-MacBook-Pro-2:~ mnowotka$ virtualenv my_new_env -p /Users/mnowotka/.pyenv/shims/python
Running virtualenv with interpreter /Users/mnowotka/.pyenv/shims/python
Using base prefix '/Users/mnowotka/.pyenv/versions/3.7.3'
New python executable in /Users/mnowotka/my_new_env/bin/python
Installing setuptools, pip, wheel...
done.
Michas-MacBook-Pro-2:~ mnowotka$ source my_new_env/bin/activate
(my_new_env) Michas-MacBook-Pro-2:~ mnowotka$ pip freeze
(my_new_env) Michas-MacBook-Pro-2:~ mnowotka$ which python
/Users/mnowotka/my_new_env/bin/python
```

virtualenv - podstawowe operacje: instalacja zależności

- Kiedy jesteśmy już w środku wirtualnego środowiska możemy normalnie instalować pakiety przy użyciu narzędzia **pip**.
- Często występującą sytuacją jest taka, w której stworzyliśmy środowisko, ponieważ chcemy zacząć pracować nad jakimś projektem, który istnieje na GitHubie, powiedzmy że chcemy zaangażować się w jakiś open source'owy projekt.
- W takiej sytuacji stworzymy czyste środowisko wirtualne, aktywujemy je i sklonujemy projekt z GitHuba. Zazwyczaj taki projekt będzie posiadał plik **requirements.txt** a więc wykonamy polecenie **pip install -r requirements.txt**, które w naszym czystym wirtualnym środowisku zainstaluje wszystkie zależności niezbędne do uruchomienia projektu.
- Może się też zdarzyć sytuacja, że to my jesteśmy głównymi programistami pracującymi nad projektem w wirtualnym środowisku a inny programista będzie chciał się przyłączyć. Wtedy sprawdzimy jakie mamy zależności w projekcie wykonując polecenie **pip freeze > requirements.txt** i podzielimy się z nim wynikowym plikiem aby łatwo mógł odtworzyć środowisko.

Autor: Michał Nowotka

virtualenv - podstawowe operacje: deaktywacja

- Wreszcie nadchodzi moment, w którym chcemy zacząć pracować nad innym projektem, z innym zestawem zależności.
- Jeśli chcemy przełączyć się z jednego wirtualnego środowiska na drugie, najpierw musimy opuścić to, w którym aktualnie się znajdujemy.
- Służy do tego polecenie **deactivate**.
- Jeśli poprawnie je wykonamy, z naszej linii poleceń zniknie nazwa środowiska, **pip freeze** będzie pokazywać globalnie zainstalowane pakiety a **which python** będzie wskazywać na pythona zainstalowanego w systemie lub skonfigurowanego w sesji.

```
(my_new_env) Michas-MacBook-Pro-2:~ mnowotka$ deactivate
Michas-MacBook-Pro-2:~ mnowotka$ which python
/Users/mnowotka/.pyenv/shims/python
```

virtualenv - podsumowanie

- Jak widać **virtualenv** jest niezwykle przydatnym narzędziem dla każdego programisty Pythona ponieważ pozwala zarządzać zależnościami dedykowanymi dla danego projektu, nad którym właśnie pracujemy.
- Z drugiej strony ma własną specyfikę:
  - trzeba się zastanawiać gdzie umieścić nowe wirtualne środowisko
  - procedura aktywacji nie jest bardzo intuicyjna
  - jeśli wirtualne środowiska są rozsiane losowo po całym systemie plików nie ma możliwości wylistować ich wszystkich
- Z pomocą w rozwiązaniu powyższych problemów przychodzi **virtualenvwrapper**
- Jak sama nazwa wskazuje, **virtualenvwrapper** jest lekką nadbudówką na narzędziu **virtualenv**, które ułatwia najczęstsze czynności wykonywane na wirtualnych środowiskach.

Autor: Michał Nowotka

virtualenvwrapper - instalacja

- Aby zainstalować **virtualenvwrapper** należy oczywiście użyć narzędzia **pip**. Należy przy tym pamiętać aby **virtualenvwrapper** był zainstalowany globalnie a nie w jakimś konkretnym wirtualnym środowisku.
- Aby mieć dostęp do poleceń dostarczonych przez narzędzie **virtualenvwrapper** trzeba je najpierw załadować do bieżącej sesji. Służy do tego polecenie **source virtualenvwrapper.sh**.
- Ponieważ tak samo jak w przypadku ustawienia zmiennych środowiskowych (np. **PYTHONPATH**) załadowanie wrappera będzie działało jedynie dla bieżącej sesji, warto je dodać do ustawień użytkownika (np. dopisać na koniec pliku **~/.bash_profile** lub w inny, zależny od naszego systemu sposób).

```
MacBook-Pro-Micha …/id_validator
↳ pip install virtualenvwrapper
Collecting virtualenvwrapper
  Downloading https://files.pythonhosted.or

MacBook-Pro-Micha …/id_validator
↳ source virtualenvwrapper.sh
```

virtualenvwrapper - podstawowe operacje: tworzenie

- Mając zainstalowany **virtualenvwrapper** możemy stworzyć nowe wirtualne środowisko używając polecenia **mkvirtualenv nazwa_srodowiska**.
- Wydaje się, że różnica jest niewielka - po prostu nazwa polecenia jest inna ale argument jest ten sam - nazwa środowiska, które chcemy stworzyć.
- Jednak kiedy wylistujemy zawartość bieżącego katalogu zauważymy, że nie ma tam podkatalogu o nazwie naszego nowego środowiska.
- Dzieje się tak, ponieważ **virtualenvwrapper** sam wybrał lokalizację dla wirtualnych środowisk - jest nim ukryty katalog **.virtualenvs** znajdujący się w domowym folderze. Ta lokalizacja może być konfigurowana przy pomocy zmiennej środowiskowej **WORKON_HOME**.

```
MacBook-Pro-Micha ../tmp
↳ mkvirtualenv my_new_env
Using base prefix '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7'
New python executable in /Users/michalnowotka/.virtualenvs/my_new_env/bin/python
Installing setuptools, pip, wheel...
done.
virtualenvwrapper.user_scripts creating /Users/michalnowotka/.virtualenvs/my_new_env/bin/predeactivate
virtualenvwrapper.user_scripts creating /Users/michalnowotka/.virtualenvs/my_new_env/bin/postdeactivate
virtualenvwrapper.user_scripts creating /Users/michalnowotka/.virtualenvs/my_new_env/bin/preactivate
virtualenvwrapper.user_scripts creating /Users/michalnowotka/.virtualenvs/my_new_env/bin/postactivate
virtualenvwrapper.user_scripts creating /Users/michalnowotka/.virtualenvs/my_new_env/bin/get_env_details

MacBook-Pro-Micha ../tmp
↳ ls
```

virtualenvwrapper - podstawowe operacje: aktywacja

- Wirtualne środowisko jest aktywowane poprzez polecenie **workon nazwa_srodowiska**.
- Jest to o wiele bardziej intuicyjna komenda bo nie trzeba pamiętać lokalizacji naszego środowiska ani tego, po niej należy dodać **/bin/activate** i używać polecenia **source**.
- Na dodatek działają podpowiedzi w bashu więc można wpisać tylko fragment nazwy a reszta zostanie uzupełniona automatycznie.

```
MacBook-Pro-Micha …/tmp
↳ workon my_new_env

MacBook-Pro-Micha …/tmp
↳ pip freeze

MacBook-Pro-Micha …/tmp
↳ which python
python: aliased to python3

MacBook-Pro-Micha …/tmp
↳ which python3
/Users/michalnowotka/.virtualenvs/my_new_env/bin/python3
```

Autor: Michał Nowotka

virtualenvwrapper - podstawowe operacje: deaktywacja, instalacja zależności

- Instalacja zależności przebiega dokładnie w ten sam sposób - przy użyciu narzędzia **pip**.
- Opuszczanie środowiska również przebiega tak samo - używając polecenia **deactivate**.

```
MacBook-Pro-Micha …/tmp
↳ pip install faker
Collecting faker
  Using cached https://files.python
any.whl
Collecting six>=1.10 (from faker)
  Using cached https://files.python
ny.whl
Collecting text-unidecode==1.2 (fro
  Using cached https://files.python
3-none-any.whl
Collecting python-dateutil>=2.4 (fr
  Using cached https://files.python
.py3-none-any.whl
Installing collected packages: six,
Successfully installed faker-1.0.7

MacBook-Pro-Micha …/tmp
↳ pip freeze
Faker==1.0.7
python-dateutil==2.8.0
six==1.12.0
text-unidecode==1.2

MacBook-Pro-Micha …/tmp
↳ deactivate
```

Autor: Michał Nowotka

virtualenvwrapper - dodatkowe operacje

- **virtualenvwrapper** posiada kilka dodatkowych komend, których brak w **virtualenv**.
- **lsvirtualenv** wypisuje wszystkie środowiska stworzone przy pomocą **virtualenvwrappera**.
- **rmvirtualenv nazwa_srodowiska** usuwa wirtualne środowisko o nazwie **nazwa_srodowiska**.
- **cpvirtualenv nazwa_srodowiska nazwa_kopii** kopiuje środowsko o nazwie **nazwa_srodowiska** do nowego środowiska o nazwie **nazwa_kopii**.

```
MacBook-Pro-Micha …/id_validator
↳ lsvirtualenv
my_new_env
==========


some_other_env
==============


yet_another_env
===============
```

```
MacBook-Pro-Micha …/id_validator
↳ rmvirtualenv yet_another_env
Removing yet_another_env...

MacBook-Pro-Micha …/id_validator
↳ lsvirtualenv
my_new_env
==========


some_other_env
==============
```

```
MacBook-Pro-Micha …/id_validator
↳ source virtualenvwrapper.sh

MacBook-Pro-Micha …/id_validator
↳ cpvirtualenv some_other_env env_copy
Copying some_other_env as env_copy...
```

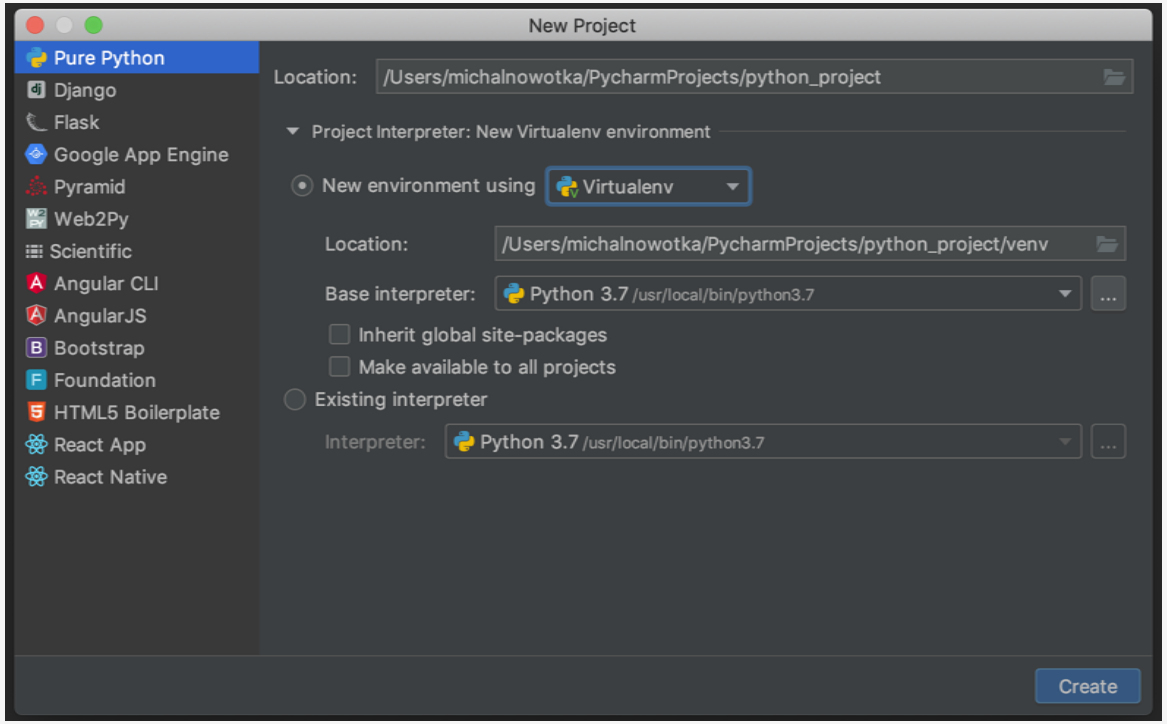
virtualenvwrapper - podsumowanie

- **virtualenvwrapper** jest dosyć prostym narzędziem ale znacznie ułatwia pracę z wirtualnymi środowiskami.
- Istnieją programiści, którzy patrzą sceptycznie na to narzędzie i wolą pracować bezpośrednio z narzędziem **virtualenv**.
- Wybór oczywiście należy do Ciebie.
- Warto przy tym wspomnieć że **PyCharm** posiada natywne wsparcie dla środowisk wirtualnych co ogranicza potrzebę korzystania z **virtualenvwrappera**.

Autor: Michał Nowotka

Środowiska wirtualne w PyCharmie.

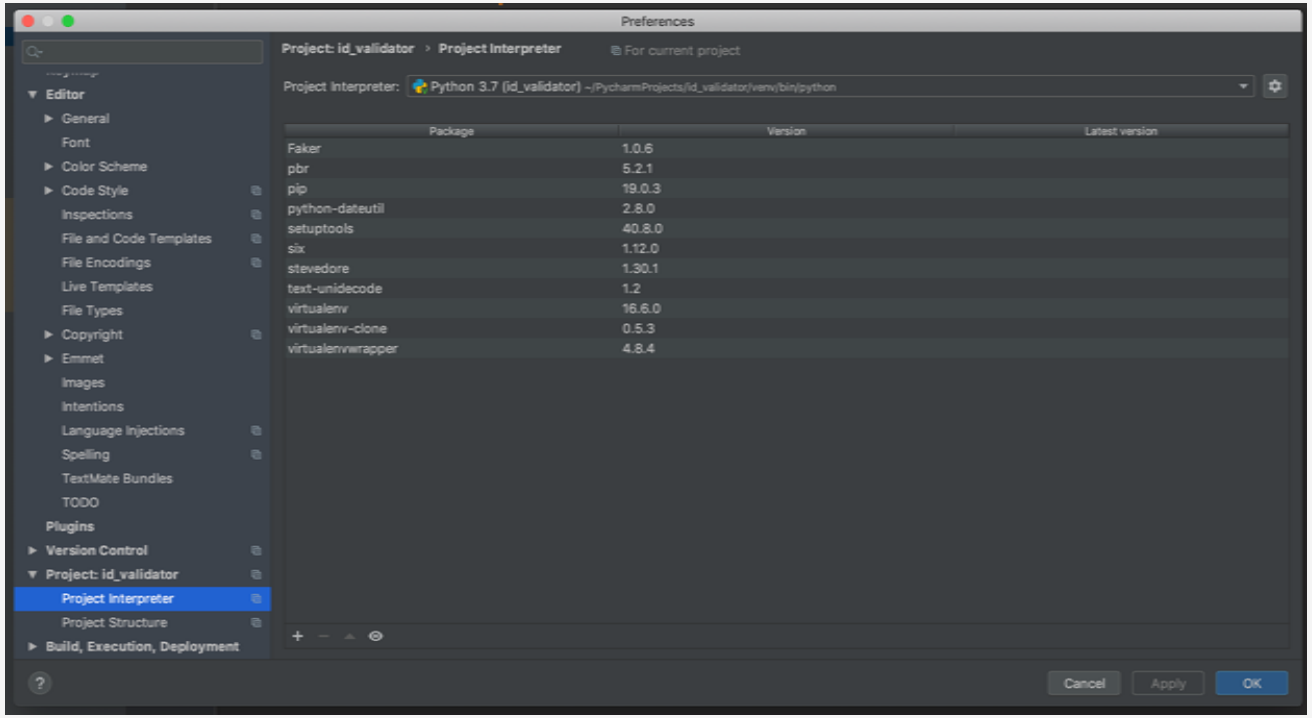
- Podczas tworzenia nowego projektu w PyCharmie sam tworzy on automatycznie nowe środowisko wirtualne i osadza je wewnątrz projektu w katalogu **venv**.
- W oknie dialogowym tworzenia projektu możemy wybrać narzędzie do tworzenia środowiska (**virtualenv**, **pipenv**, **conda**) jak również wybrać wersję Pythona.



Autor: Michał Nowotka

Środowiska wirtualne w PyCharmie.

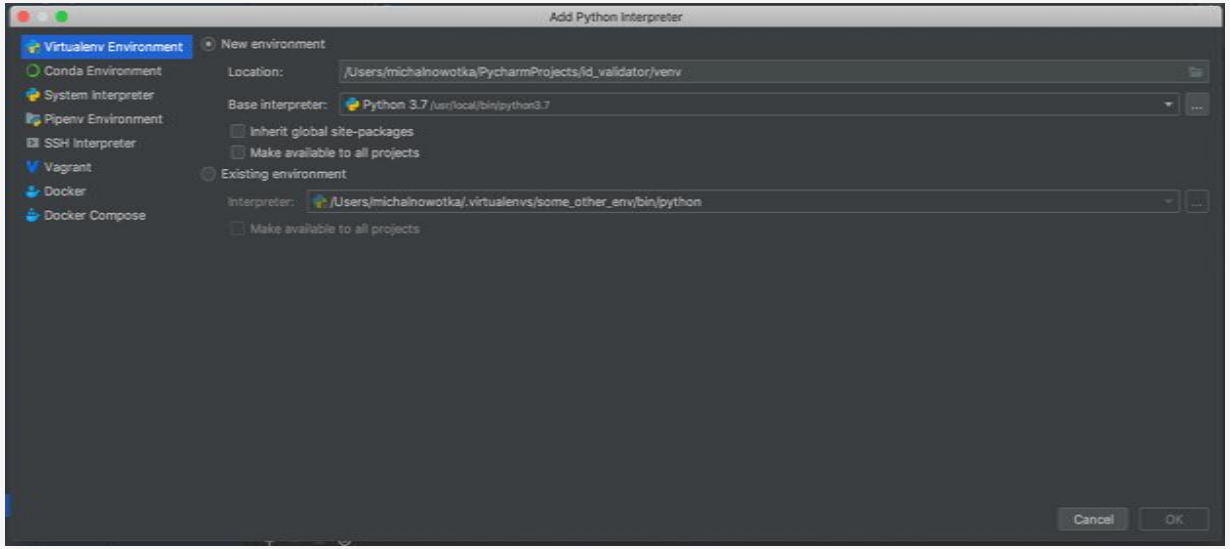
- Jeśli projekt został sklonowany z repozytorium to PyCharm sam nie utworzy wirtualnego środowiska i trzeba będzie je skonfigurować.
- Aby to zrobić należy wybrać **PyCharm -> Preferences -> Project -> Project Interpreter** w OSX.
- Dla Linuxa będzie to **File -> Settings -> Project -> Project Interpreter**.
- Okno dialogowe wyświetli wtedy ścieżkę do interpretera Pythona - będzie to zapewne systemowy Python. Zostaną również wyświetlone aktualnie zainstalowane pakiety wraz z ich wersjami oraz informacją czy istnieje nowsza wersja pakietu.
- Nas najbardziej interesuje kółko zębate blisko prawego górnego rogu.



Autor: Michał Nowotka

Środowiska wirtualne w PyCharmie.

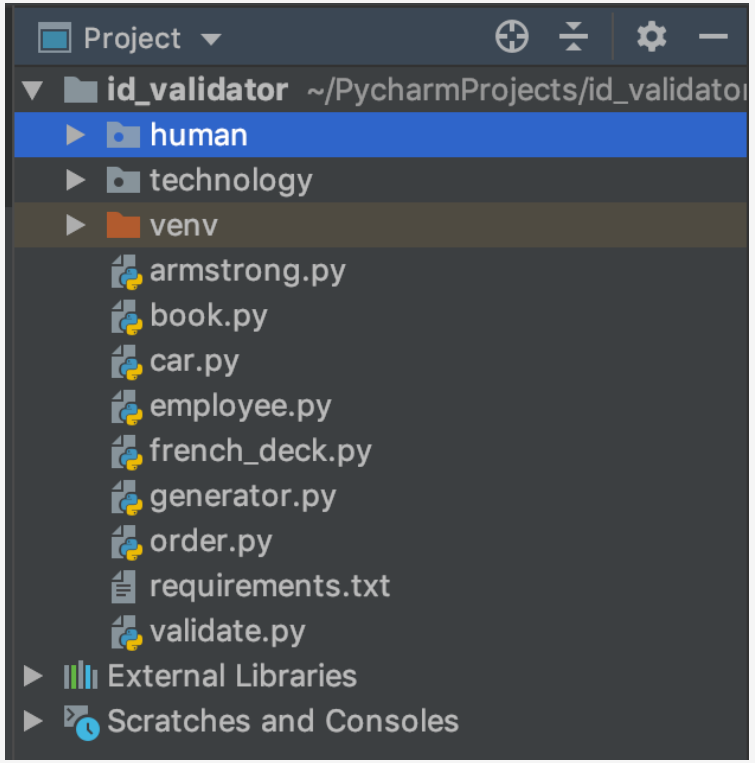
- Po kliknięciu w kółko zębate wybieramy opcję **‘add’** i naszym oczom ukazuje się nowe okno dialogowe.
- Możemy za jego pomocą stworzyć nowe wirtualne środowisko - domyślnie o nazwie **venv**, znajdujące się w głównym folderze projektu.
- Możemy również określić wersję Pythona spośród zainstalowanych w naszym systemie.
- Opcjonalnie możemy użyć narzędzia **conda** lub **pipenv** albo połączyć się z pythonem na zdalnej maszynie poprzez **ssh** lub z pythonem na wirtualnej maszynie używając **vagranta** lub na kontenerze za pomocą **dockera** - wszystkie te opcje są dość zaawansowane jak na nasze potrzeby.
- Możemy też skorzystać ze “systemowej”, globalnej wersji Pythona.



Autor: Michał Nowotka

Środowiska wirtualne w PyCharmie.

- Standardowo skonfigurowane wirtualne środowisko w PyCharmie ma nazwę **venv** i jest widocznie w okienku projektu.
- Katalog podkreślony jest na czerwono i nie jest dodawany do repozytorium, standardowy plik **.gitignore** generowany przez GitHuba dla Pythona zawiera w sobie nazwę **venv** aby ignorować ten folder.
- Kiedy wirtualne środowisko jest skonfigurowane w PyCharmie to może on zaindeksować kod zależnych bibliotek i lepiej sprawdzać poprawność kodu.
- Zakładka terminal uruchamia basha od razu z aktywowanym wirtualnym środowiskiem.
- Zakładka Python Console uruchamia Pythona ze skonfigurowanego wirtualnego środowiska.



pyenv - wygodne zarządzanie wersjami Pythona

- Wspomnieliśmy wcześniej, że w PyCharmie można wybrać interpreter Pythona spośród wszystkich dostępnych w systemie.
- Również narzędzie **virtualenv** podczas tworzenia nowego środowiska udostępnia opcję **-p** pozwalającą podać ścieżkę do Pythona, który ma być interpreterem w tym wirtualnym środowisku.
- Czy to znaczy, że na jednym komputerze może być naraz więcej niż jeden Python?
- Odpowiedź brzmi tak :)
- Środowiska wirtualne izolują jedynie wersje pakietów. Ale co kiedy chcemy zejść głębiej? Być może zostaliśmy poproszeni o naniesienie poprawki w starym projekcie działającym na Pythonie 2.7 a na co dzień pracujemy nad projektem korzystającym z Pythona 3.7.
- Aby wygodnie zarządzać wersjami Pythona warto skorzystać z narzędzia o nazwie **pyenv**.

Autor: Michał Nowotka

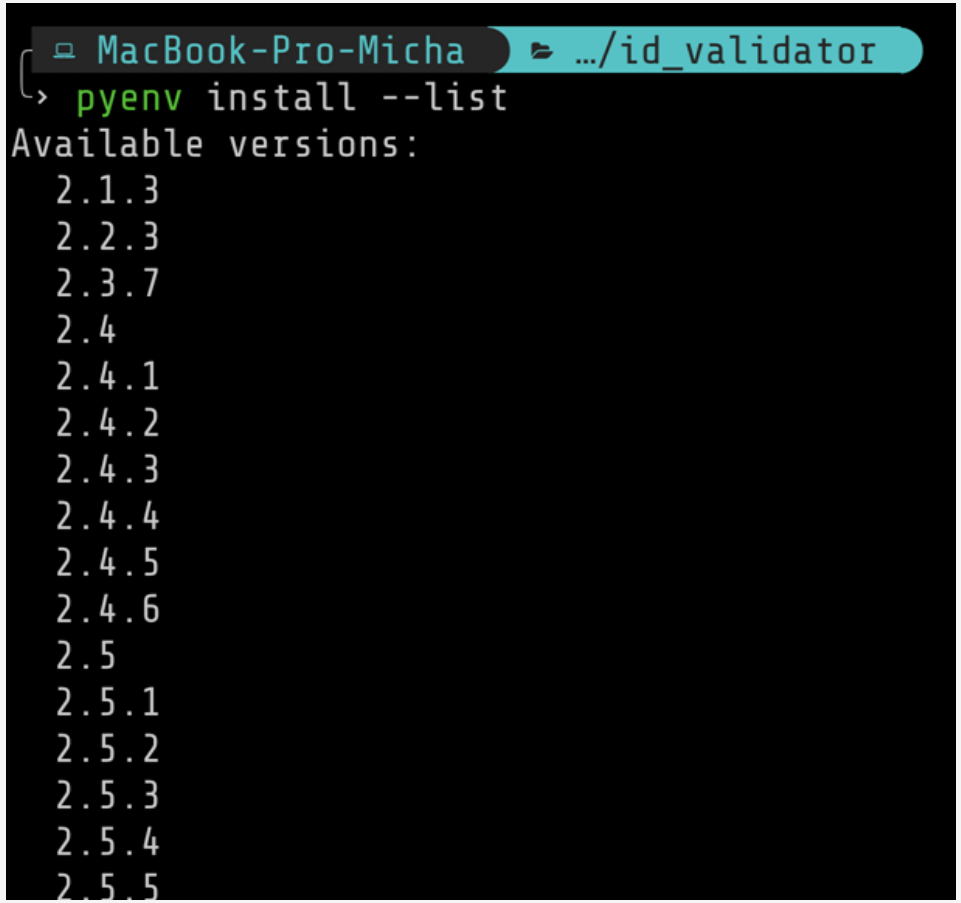
pyenv - instalacja

- Dla systemu OSX, **pyenv** można zainstalować przy pomocy narzędzia **homebrew**.
- Dla Linuxa, można skorzystać z automatycznego instalatora (upewniwszy się wcześniej, że mamy zainstalowane w systemie wszystkie zależności): **curl https://pyenv.run | bash**
- Dla Windowsa istnieje specjalny projekt o nazwie pyenv-win.
- Poprawnie zainstalowany **pyenv** będzie dostępny jako program w wierszu poleceń.
- Możemy łatwo sprawdzić jakie udostępnia komendy.

```
MacBook-Pro-Micha ▸ …/id_validator
› pyenv
pyenv 1.2.8
Usage: pyenv <command> [<args>]

Some useful pyenv commands are:
   commands    List all available pyenv commands
   local       Set or show the local application-specific Python version
   global      Set or show the global Python version
   shell       Set or show the shell-specific Python version
   install     Install a Python version using python-build
   uninstall   Uninstall a specific Python version
   rehash      Rehash pyenv shims (run this after installing executables)
   version     Show the current Python version and its origin
   versions    List all Python versions available to pyenv
   which       Display the full path to an executable
   whence      List all Python versions that contain the given executable

See `pyenv help <command>' for information on a specific command.
For full documentation, see: https://github.com/pyenv/pyenv#readme
```

pyenv - dostępne do instalacji wersje Pythona

- Jednym z ważniejszych poleceń narzędzia **pyenv** jest **install**.
- **pyenv install --list** drukuje listę dostępnych do instalacji wersji Pythona.



Autor: Michał Nowotka

pyenv - instalacja specyficznej wersji Pythona

- Kiedy zdecydowaliśmy już jaką wersję zainstalować możemy przystąpić do właściwej instalacji.
- Do instalacji służy polecenie **pyenv install nazwa_wersji**.

```
MacBook-Pro-Micha ▸ …/id_validator
└> pyenv install 3.6.7
python-build: use openssl from homebrew
python-build: use readline from homebrew
Downloading Python-3.6.7.tar.xz...
-> https://www.python.org/ftp/python/3.6.7/Python-3.6.7.tar.xz
Installing Python-3.6.7...
python-build: use readline from homebrew
WARNING: The Python sqlite3 extension was not compiled. Missing the SQLite3 lib?
Installed Python-3.6.7 to /Users/michalnowotka/.pyenv/versions/3.6.7

pyenv install 3.6.7  190.40s user 41.10s system 232% cpu 1:39.50 total
```

pyenv - sprawdzanie wersji dostępnych w systemie

- Aby sprawdzić wszystkie wersje Pythona zainstalowane poprzez **pyenv** w naszym loklanym systemie należy użyć polecenia **pyenv versions**.
- Aktywna wersja będzie zaznaczona gwiazdką.

```
MacBook-Pro-Micha ▸ …/id_validator
› pyenv versions
* system (set by /Users/michalnowotka/.pyenv/version)
  3.6.7
```

pyenv - aktywacja wersji

- Istnieje kilka sposobów na aktywację zainstalowanej wersji Pythona.
- Najlepiej jest zrobić to w dwóch krokach:
  - wykonać komendę **pyenv init**, która poinstruuje w jakim pliku (zazwyczaj będzie to **~/.bash_profile** lub **~/.zshrc**) należy dodać linijkę
    eval "$(pyenv init -)"
  - wykonać komendę **pyenv shell nazwa_zainstalowanej_wersji**.
- Po aktywacji można korzystać z Pythona w wybranej wersji.

```
MacBook-Pro-Micha …/id_validator
› pyenv init
# Load pyenv automatically by appending
# the following to ~/.zshrc:

eval "$(pyenv init -)"
```

```
MacBook-Pro-Micha ~
› pyenv shell 3.6.7

MacBook-Pro-Micha ~
› python
Python 3.6.7 (default, May 31 2019, 16:18:10)
[GCC 4.2.1 Compatible Apple LLVM 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

Autor: Michał Nowotka

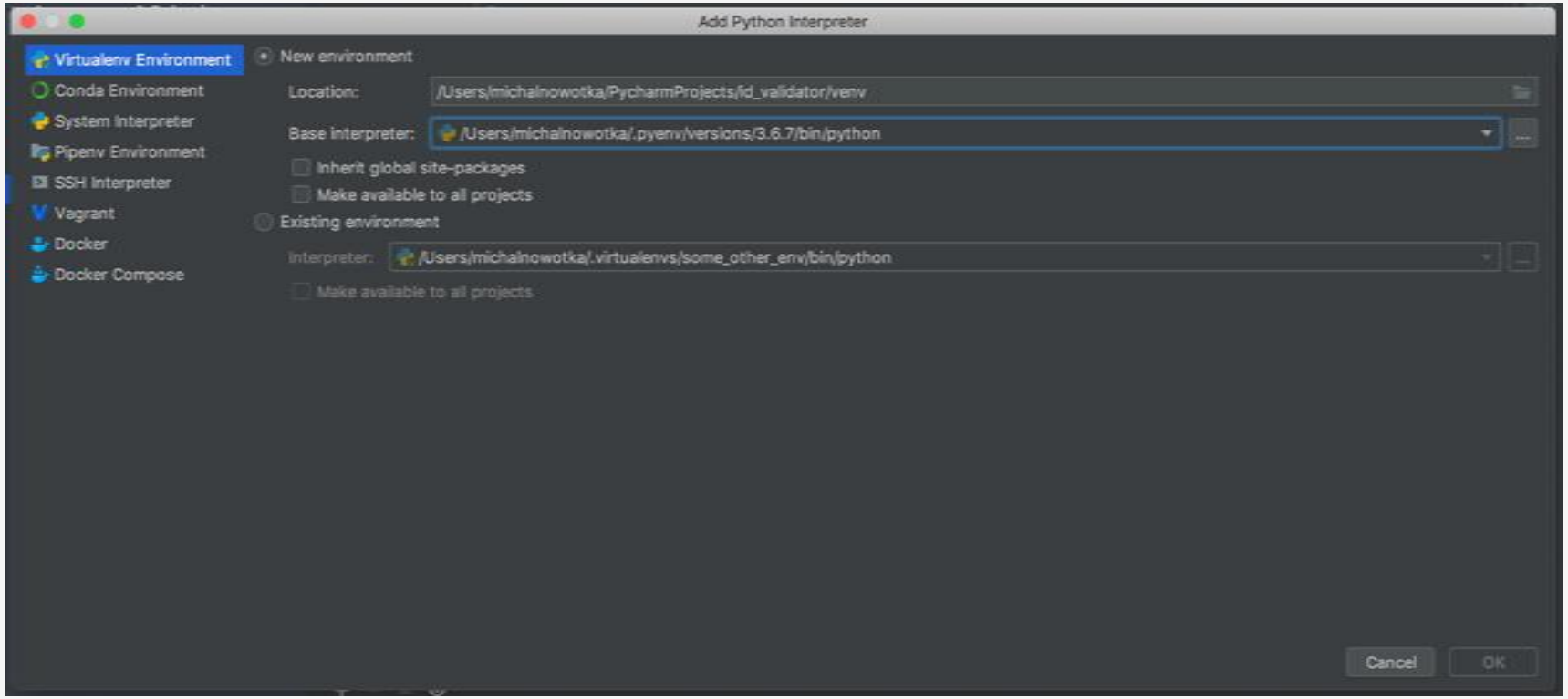
pyenv - deaktywacja wersji

- Aby przywrócić domyślne ustawienia należy użyć komendy **pyenv shell --unset**.

```
MacBook-Pro-Micha ~
› pyenv shell 3.6.7

MacBook-Pro-Micha ~
› python
Python 3.6.7 (default, May 31 2019, 16:18:10)
[GCC 4.2.1 Compatible Apple LLVM 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>

MacBook-Pro-Micha ~                                          01:43
› pyenv shell --unset

MacBook-Pro-Micha ~
› python
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

pyenv - użycie z PyCharmem

- **pyenv** przechowuje wersje Pythona w ukrytym katalogu o nazwie **.pyenv/versions/** w katalogu domowym
- Z tą wiedzą możemy dodać interpreter Pythona dostarczony przez **pyenv** do PyCharma.
- Wystarczy wskazać odpowiednią ścieżkę w polu Base interpreter w oknie dialogowym Add Python Interpreter omówionym na poprzednich slajdach.



Interpretery pythona - Cython, Pypy, Conda, Iron Python

- Kiedy wykonamy polecenie **pyenv install --list** zobaczymy, że lista dostępnych wersji Pythona jest bardzo długa. Oprócz wersji numerycznych (takich jak np. 3.7.1) są wersję zaczynające się przedrostkami:
  - ironpython
  - jython
  - pypy
  - miniconda
- Okazuje się, że język Python jest tylko specyfikacją składni. Istnieje wiele różnych implementacji, które realizują tę specyfikację.
- Najbardziej popularna i domyślna implementacja interpretera Pythona jest nazywana CPython i jest napisana w języku C.
- Poznajmy inne implementacje.

Autor: Michał Nowotka

Interpretery pythona - Pypy

- Pypy jest implementacją Pythona napisaną w … Pythonie!
- Choć może to brzmieć dziwnie ale niektóre programy potrafią wykonywać się szybciej gdy są interpretowane przez Pypy niż przez CPythona.
- Zazwyczaj chodzi o programy, które wykonują wiele operacji numerycznych.
- Niestety Pypy nie sprawdza się zupełnie w przypadku aplikacji webowych - nie ma dowodów żeby Pypy było używane gdziekolwiek na produkcji w webowym środowisku.
- Z drugiej strony Pypy jest cennym źródłem wiedzy - kiedy wszystkie najważniejsze struktury danych są zaimplementowane w CPythonie w C, w Pypy z definicji wszystkie elementy standardowej biblioteki są napisane w czystym Pythonie więc czytając je można wiele się nauczyć.

Autor: Michał Nowotka

Interpretery pythona - Conda

- Conda jest komercyjną wersją Pythona tworzoną przez firmę Continuum.io
- Sprawdza się świetnie dla naukowych obliczeń - posiada wysoko zoptymalizowane wersje najważniejszych numerycznych bibliotek takich jak numpy, scipy czy scikit-learn skompilowane pod każdą z najpopularniejszych platform.
- Darmową wersją interpretera jest miniconda.
- Kolejną zaletą condy jest fakt, że łatwo ją zainstalować i nie trzeba przy tym posiadać uprawnień administratora - dlatego minicondę można bez problemu zainstalować na dowolnym komputerze.

Interpretery pythona - Jython

- Jython jest interpreterm Pythona napisanym w Javie.
- Dzięki temu można w nim korzystać z bibliotek Javy, które są bardzo bogate w funkcjonalność.
- W przypadku kiedy trzeba korzystać z narzędzia, które istnieje wyłącznie w Javie a z drugiej strony potrzebny jest nam Python, Jython może okazać się świetnym wyborem.
- Wadą Jythona jest fakt, że rozwój projektu zatrzymał się właściwie na Pythonie 2.7.

Autor: Michał Nowotka

Interpretery pythona - IronPython

IronPython

- IronPython jest interpreterem Pythona napisanym w C#.
- Oferuje natywny dostęp do bibliotek .NET
- Cierpi na podobne przypadłości co Jython, tzn. nie nadąża za rozwojem Pythona i w tej chwili nie oferuje implementacji Pythona 3.

Autor: Michał Nowotka

Setuptools

- Dowiedzieliśmy się jak skonfigurować deweloperskie środowisko w oparciu o narzędzie takie jak **pip**, **virtualenv**, **pyenv**.
- Jeśli umiejętnie je wykorzystamy i połączymy z naszą wiedzą o samym języku istnieje prawdopodobieństwo że stworzymy udany projekt :)
- W takim razie warto będzie się nim podzielić i opublikować tak by był dostępny dla każdego innego Pythonisty jako zewnętrzny pakiet, gotowy do zainstalowania z poziomu narzędzia **pip**.
- Pypy korzysta z globalnego rejestru pakietów znanego jako **PyPI** - **Py**thon **P**ackage **I**ndex albo Cheeseshop.
- Aby zarejestrować swój pakiet w **PyPI** należy do najpierw odpowiednio zapakować.
- Służy do tego biblioteka setuptools oraz pewne pliki, które muszą się znaleźć w projekcie aby był gotowy do zapakowania i wysłania do **PyPI**.
- Za chwilę omówimy wszystkie te pliki.

Autor: Michał Nowotka

setup.py

- W pliku **setup.py** są zawarte najważnejsze informacje o projekcie.
- Będzie to:
  - nazwa projektu
  - wersja
  - licencja
  - krótki opis
  - autor
  - bezpośrednie zależności
  - informacja czy projekt zawiera również inne pliki poza źródłami w Pythonie
- Opcji może być znacznie więcej i treść pliku **setup.py** w dużej mierze zależy od poziomu skomplikowania projektu.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


from setuptools import find_packages, setup

setup(
    name='my_cool_lib',
    version='0.1.0',
    author='SomeCompany PLC',
    author_email='dev@example.com',
    packages=find_packages(),
    include_package_data=True,
    description='Super useful library'
)
```

MANIFEST.in

- Plik manifestu jest jakby spisem treści.
- Wymienia wszystkie niepythonowe pliki, które mają wejść w skład paczki, która będzie wysłana do **PyPI**.
- Jeśli jakiś plik się tam nie znajdzie to nie wejdzie do paczki.
- Jeśli plików jest dużo i znajdują się w katalogach to można je dodawać rekurencyjnie.

```
include README.rst
include docs/*.txt
include funniest/data.json
```

README

- Plik **README** powinien zawierać przyjazną dokumentację projektu.
- Kiedy nasz pakiet zostanie zarejestrowany w **PyPI** dostanie swoją własną stronę internetową.
- Treścią tej strony będzie właśnie treść pliku README
- Ponieważ plik **README** nie jest kodem Pythona, obowiązkowo musi zostać włączony w pliku **MANIFEST.in**
- Plik README można napisać w dwóch formatach:
  - rst - reStructuredText - będzie się wtedy nazywał **README.rst**
  - md - Markdown - będzie się wtedy nazywał **README.md**
- Oba formaty pozwalają na tworzenie bogatych treści i formatowania - dodawanie linków, paragrafów, grafiki ale Markdown jest znacznie bardziej popularny i jest rekomendowanym formatem.

Autor: Michał Nowotka

twine

- Kiedy nasz projekt posiada już odpowiednią strukturę, to jest:
  - plik **README**
  - plik **setup.py**
  - plik **MANIFEST.in**
  - właściwy pythonowy kod
- to może zostać zarejestrowany w **PyPI**.
- Służy do tego narzędzie **twine**.

```
top
|-- package
|   |-- __init__.py
|   |-- module.py
|   `-- things
|       |-- cross.png
|       |-- fplogo.png
|       `-- tick.png
|-- runner
|-- MANIFEST.in
|-- README
`-- setup.py
```

Autor: Michał Nowotka

twine - rejestrowanie pakietu w PyPI

- Rejestrowanie pakietu (który posiada już wymaganą strukturę) przebiega w trzech krokach:
  - Należy założyć konto na https://pypi.org/account/register/
  - Należy użyć biblioteki **setuptools**, w połączeniu z napisanym przez nas plikiem **setup.py** i wykonać polecenie: **python setup.py sdist**
  - To polecenie stworzy paczkę gotową do wysłania na serwery **PyPI**.
  - Paczka zostanie zapisana w katalogu **dist** i będzie miała formę archiwum o rozszerzeniu **tar.gz**.
  - Nazwą paczki będzie nazwa naszego projektu z sufixem oznaczającym wersję.
  - Mając wygenerowaną paczkę możemy ją wysłać narzędziem **twine** używając polecenia **twine upload dist/nazwa_projektu-0.1.0.tar.gz**
  - **twine** w międzyczasie poprosi o dane logowania do **PyPI** i jeśli wszystko będzie w porządku wyśle i zarejestruje naszą paczkę w **PyPI**.
  - Będzie ją teraz można zainstalować na dowolnym komputerze przy pomocy narzędzia **pip**.