

# IPython. Wygodna interaktywna powłoka Pythona

Podczas pierwszego spotkania z Pythonem zwykle poznaje się jego wbudowany interpreter. Niestety powłoka ta jest mało przyjazna zarówno dla nowicjuszy, jak i dla wymagających użytkowników. W artykule przedstawione zostało rozwiązanie tego problemu – interaktywna konsola IPython. Opisano najciekawsze, ułatwiające pracę funkcjonalności, a także procesy instalacji oraz konfiguracji. Wspomniano o IPython Notebooku, czyli projekcie Jupyter, oraz o alternatywach dla IPython, również w innych językach.

## INSTALACJA

Proces instalacji jest taki sam zarówno na Windowsie, Linuksie, jak i Mac OSX. Należy skorzystać z menedżera paczek PIP, który jest dostarczany wraz z Pythonem 3. W innym przypadku najłatwiej jest go zainstalować, wykonując skrypt pobrany z oficjalnej strony: <https://bootstrap.pypa.io/get-pip.py>.

Samego IPython instalujemy poleceniem:

```
pip install ipython
```

Jeśli z jakiegoś powodu zainstalowany skrypt (pip) nie znajduje się w zmiennej środowiskowej PATH (i powyższe polecenie nie zostało odnalezione), można skorzystać z flagi -m interpretera Pythona, w celu bezpośredniego uruchomienia modułu:

```
python -m pip install ipython
```

Następnie, aby uruchomić interaktywną konsolę, wykonujemy polecenie `ipython`. W systemie operacyjnym Windows może to wymagać dodania do zmiennej środowiskowej PATH ścieżki `c:\PythonXX\Scripts\`, gdyż tam właśnie znajduje się program `ipython.exe`.

## NAWIGACJA, PODPOWIADANIE ORAZ INTROSPEKCJA

IPython – podobnie jak zwykła konsola Pythona – pozwala na znana ze standardowej powłoki Linuksowej – Basha – nawigację strzałkami w celu powrotu do poprzedniego polecenia. Ponadto oferuje autouzupełnianie nazw klawiszem tabulacji, a także opcję uzyskania szybkiej pomocy i introspekcji atrybutów/metod obiektów. Zamiast wywoływać funkcje `dir(obj)` czy `help(obj)` wykorzystujemy klawisz tabulacji oraz skróty `obj?` i `obj??`. Ostatnie polecenie poza pomocą wyświetli także źródło danej funkcji, jeśli jest ono dostępne.

Istnieje także możliwość przeszukiwania istniejących nazw (stworzonych przez nas lub zaimportowanych), wykorzystując symbol `*`, który oznacza użycie dowolnego znaku dowolną ilość razy. Funkcjonalności te można zobaczyć na Rysunku 1.

## HISTORIA

Podczas pracy w konsoli zapisywana jest historia wejścia oraz wyjścia – również między interaktywnymi sesjami. Wejście przechowywane

```
In [1]: from pwn import *
In [2]: *b64*?
b64d
b64e
In [3]: b64e?
Signature: b64e(s)
Docstring:
b64e(s) -> str
Base64 encodes a string
Example:
>>> b64e('test')
'dGVzdA=='
File: /usr/local/lib/python2.7/dist-packages/pwnlib/util/fiddling.py
Type: function
In [4]: b64d??
Signature: b64d(s)
Source:
def b64d(s):
    """b64d(s) -> str
    Base64 decodes a string
    Example:
    >>> b64d('dGVzdA==')
    'test'
    """
    return base64.b64decode(s)
File: /usr/local/lib/python2.7/dist-packages/pwnlib/util/fiddling.py
Type: function
In [5]: b64d
b64d      basestring  binary_ip  bitswap    %bookmark  buffer
b64e      %!bash      bits       bitswap_int bool        bytearray
base64    bin         bits str   bnot       break       bytes
```

Rysunek 1. Szybka pomoc oraz autouzupełnianie/introspekcja atrybutów obiektów

jest w liście In, a wyjście w słowniku Out (mapującym numer wejścia do zwróconego wyjścia). Magiczne polecenie `%hist` lub `%history` pozwala na wylistowanie historii wejścia. Przydatne mogą być flagi `-n` oraz `-g`, które wyświetlają odpowiednio numer wpisu w historii i globalną historię z numerem sesji oraz numerem polecenia. Do obiektów znajdujących się w historii wyjścia można odwoływać się przez `_`, `__`, `___`, `_N` – będzie to pierwszy, drugi, trzeci oraz N-ty obiekt w historii (N to numer wyświetlany w nawiasie przy Out).

Do operowania na historii istnieje także kilka magicznych komend:

- › `%recall [numer lub zakres]` (istnieje też alias `-%rep`) – wypisuje na wejście wcześniej wykonaną instrukcję (lub wiele, w przypadku podania zakresu instrukcji, np. 4-5), dzięki czemu można ją zedytować oraz ponownie wykonać. Uruchomienie polecenia bez argumentów wykona komendę z ostatnim możliwym numerem (z historii wejścia).
- › `%rerun [numer lub zakres]` – podobnie jak `recall`, zamiast wypisywać, wykonuje polecenia.
- › `%save plik.py <numer lub zakres>` – pozwala na zapisanie instrukcji do skryptu. Wykonane magiczne komendy zo-

# Zintegruj własną aplikację z SMSAPI

Skorzystaj z gotowych bibliotek w językach:

php



C#

python



Załącz konto firmowe z kodem polecenia i odbierz pakiet SMS-ów na przetestowanie naszych usług:

FORMULARZ REJESTRACYJNY:

[www.smsapi.pl/rejestracja](http://www.smsapi.pl/rejestracja)

KOD POLECENIA:

BONUS:

PR56 500 SMS-ÓW

SMSAPI



staną zastąpione przez odpowiednie wywołania z API IPython, np. `%history` na `get_ipython().magic('history')`.

- › `%pastebin <numer lub zakres>` – tworzy anonimową wklejkę (tzw. gist) w serwisie `github.com` zawierającą dane instrukcje.
- › `%edit [opts] [args]` – uruchamia edytor (ustawiony w zmiennej środowiskowej `EDITOR` lub w konfiguracji IPython), a następnie wykonuje skrypt z zedytowanego tymczasowego pliku. Pozwala na podanie numeru lub zakresu linii z historii, ścieżki do pliku, zmiennej typu string (jej zawartość zostanie wklejona do edytora), obiektu (w tym przypadku załadowany zostanie plik, w którym dany obiekt jest zdefiniowany; jest to jeden ze sposobów na zobaczenie, jak coś działa pod spodem) lub też makra.
- › `%macro` – wyświetla stworzone makra lub, gdy podamy argument, tworzy nowe makro. Argumentem może być numer instrukcji, plik czy obiekty, które są stringami.

Przykład użycia historii w IPythonie dla Pythona 3 można zobaczyć na Listingu 1.

**Listing 1. Wykorzystanie historii poleceń oraz magicznych komend z nią związanych**

```
In [1]: %history -g "as b64d"
397/2: from base64 import b64decode as b64d
413/3: from base64 import b64decode as b64d

In [2]: %rerun 397/2
=== Executing: ===
from base64 import b64decode as b64d
=== Output: ===

In [3]: b64d('FzwnBwdHUnpwHwEGARcORkdTVE9IUA==')
Out[3]: b"\x17<\x07\x07GRzp\x1f\x01\x06\x01\x17\x0eFGSTOHP"

In [4]: key = b'CTFs 4r3 funny... :)'

In [5]: ''.join(chr(i ^ j) for i, j in zip(_3, key))
Out[5]: "That's IPython history"

In [6]: key = b'Z]Duh4r\x19\x11q!dd7o1' ;"-q'

In [7]: %macro decode 5
Macro `decode` created. To execute, type its name (without quotes).
=== Macro contents: ===
''.join(chr(i ^ j) for i, j in zip(_3, key))

In [8]: decode
Out[8]: 'Macros can be awesome!'
```

## SYSTEM „MAGICZNYCH KOMEND”

„Magiczne komendy” (ang. *magic commands*) – dodatkowe polecenia wykonywane przez jednoargumentowy operator `%` (jednoliniowe – tak zwane z ang. *line magic*) lub `%%` (wieloliniowe – tak zwane *cell magic*).

*Cell magics* na swoim wejściu nie muszą dostać poprawnego kodu Pythona – zamiast tego mogą zmodyfikować dane wejściowe lub potraktować je jako argument czy opcje (jak na przykład opisany niżej `%%timeit`).

Poniżej zaprezentowano subiektywną listę przydatnych komend:

- › `%reset, %who` – usuwanie wszystkich ustawionych nazw z interaktywnej sesji oraz ich wyświetlanie. `%who` pomija wyświetlanie nazw, które zostały stworzone przez skrypty konfiguracyjne.
- › `%cd, %pwd` – zmiana katalogu (na Windowsie również dysku) oraz wypisywanie katalogu, w którym się znajdujemy. Katalogi odwiedzone poprzez `%cd` zapisywane są w liście `_dh`. Można je też wypisać poprzez magiczną komendę `%dhist`. Zaprezentowano to w Listingu 2.

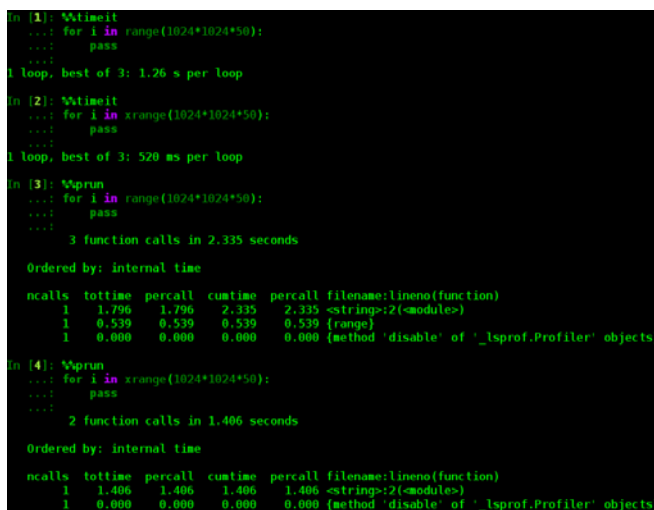
**Listing 2. Magiczne komendy `%cd, %dhist`**

```
In [1]: %cd D:/code
D:\code

In [2]: %cd -
C:\python_35_amd64

In [3]: %dhist
Directory history (kept in _dh)
0: C:\python_35_amd64
1: D:\code
2: C:\python_35_amd64
```

- › `%run [opts] (-m mod | file) [args]` – uruchamia dany skrypt z podanymi argumentami. Nazwy utworzone przez program są następnie dostępne z poziomu IPython, stąd komenda ta jest przydatna podczas prototypowania/testowania danego rozwiązania. Pomocne mogą być też flagi: `-d`, która uruchamia skrypt pod kontrolą debuggera `pdb`, czy `-t`, czyli wyświetlanie informacji odnośnie czasu wykonania skryptu.
- › `%timeit, %prun` – mierzenie czasu wykonania oraz profilowanie krótkich kawałków kodów. Obie komendy wspierają także wywołanie w bloku wieloliniowym (`%%`). W przypadku `%%timeit` pierwszy argument traktowany jest jako kod inicjalizacyjny (jest wykonywany przed testem, zatem nie jest mierzony jego czas). Komendy te przedstawiono na Rysunku 2.



Rysunek 2. Wykorzystanie `%timeit` oraz `%prun` – na przykładzie wbudowanych funkcji `range` oraz `xrange` w Pythonie 2. Wersja z `range` działa wolniej, ponieważ tworzy tymczasową listę, a nie generator, tak jak `xrange`. W Pythonie 3 funkcja `range` została zastąpiona funkcją `xrange`, ta zaś została usunięta

- › `%automagic, %autocall` – opcje automatycznego wołania magicznych komend oraz funkcji. Po odpowiednim ustawieniu nie trzeba poprzedzać komend znakiem `%` oraz można wywoływać funkcje w stylu komend (ang. *command-style*), czyli bez podawania nawiasów. `%automagic` można włączyć i wyłączyć. `%autocall` przyjmuje jedną z trzech wartości:
  - » 0 – wyłączone.
  - » 1 – aktywne, ale w przypadku wykonania `foo` (gdy `foo` można zawołać bez argumentów) funkcja (lub obiekt, który można zawołać jak funkcję – czyli taki, który posiada magiczną metodę `__call__`) nie zostanie wywołana (zamiast tego zostanie wypisana jej nazwa wraz z przestrzenią nazw, w której się znajduje, a w przypadku obiektu dodatkowo jego identyfikator – czyli to, co zwraca wbudowana funkcja `id`, gdy prześlemy jej ten obiekt).

- » 2 – aktywne, działa nawet gdy podamy argument do czegoś, co da się zawołać (ang. *callable*).

Obie opcje zaprezentowano w Listingu 3.

**Listing 3. Opcje %automagic oraz %autocall**

```
In [2]: %automagic
Automagic is OFF, % prefix IS needed for line magics.

In [3]: pwd
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-c6ad4d9c5b03> in <module>()
----> 1 pwd
NameError: name 'pwd' is not defined

In [4]: %automagic
Automagic is ON, % prefix IS NOT needed for line magics.

In [5]: pwd
Out[5]: 'C:\\Users\\dc'

In [6]: def foo(): return 5

In [7]: %autocall 2
Automatic calling is: Full

In [8]: foo
-----> foo()

Out[8]: 5
In [9]: %autocall 1
Automatic calling is: Smart

In [10]: foo
Out[10]: <function __main__.foo>

In [11]: def foo(x): return x

In [12]: foo 5
-----> foo(5)
Out[12]: 5

In [13]: %autocall 0
Automatic calling is: OFF

In [14]: foo 5
File "<ipython-input-14-f2befd89a251>", line 1
foo 5
  ^
SyntaxError: invalid syntax
```

- » %bookmark – zarządzanie oraz tworzenie zakładki do katalogów przechowywanych między sesjami IPython. Do stworzonej zakładki można wejść przez %cd -b <nazwa>, co wiadać w Listingu 4.

**Listing 4. Wykorzystanie zakładek**

```
In [4]: %bookmark ctf d:/ctf

In [5]: %bookmark -l
Current bookmarks:
code -> d:/code
ctf -> d:/ctf

In [6]: %pwd
Out[6]: 'C:\\Users\\dc'

In [7]: %cd -b ctf
(bookmark:ctf) -> d:/ctf
d:/ctf

In [8]: %pwd
Out[8]: 'd:\\ctf'
```

- » %lsmagic – listowanie magicznych komend.
- » %pdb – włącza i wyłącza automatyczne uruchamianie interaktywnego debugera pdb podczas rzucenia wyjątku.
- » %logstart – włącza logowanie kolejnych komend do pliku

o nazwie podanej w opcjonalnym argumente (plik zostanie utworzony w obecnym katalogu). W przypadku, gdy nie zostanie on podany – loguje do ipython\_log.py. Można podać również opcjonalny parametr – tryb logowania (ang. *logging mode*), który przyjmuje jedną z wartości:

- » append – zapisuje logi na koniec pliku, nie zważając na to, czy plik istniał wcześniej.
- » backup – jeśli podany plik logów istnieje, stworzy jego kopię zapasową o nazwie zakończonej ~, a następnie będzie logować do podanego pliku.
- » global – zapisuje logi na koniec pliku, który zostanie umieszczony w katalogu domowym.
- » over – nadpisuje podany plik logów.
- » rotate – tworzy kolejne pliki logów: plik.1~, plik.2~, plik.3~ itd.

Może także logować wyjście interpretera (flaga -o), czas (-t) oraz nieprzetworzone wejście (flaga -r; domyślnie wszystkie magiczne komendy są zamieniane na ich odpowiedniki – na przykład %ls-magic na get\_ipython().magic(u'lsmagic')).

- » %logstate – wyświetla informacje o logowaniu.
- » %logstop – wyłącza logowanie.

## POLECENIA POWŁOKI

IPython można również wykorzystywać jako rozszerzoną powłokę systemową. Polecenie powłoki systemowej możemy użyć, pisząc przed nim unarny operator !. Przypisując wynik polecenia do zmiennej, otrzymamy obiekt typu SList, który zawiera standardowy strumień wyjścia (stdout) oraz błędów (stderr). Obiekt taki ma kilka przydatnych funkcjonalności, co zaprezentowano w Listingu 5.

**Listing 5. Wykorzystywanie poleceń powłoki, w której został uruchomiony IPython. W tym przypadku wykorzystany został program cat (z paczki Git for Windows) oraz skrypt batch**

```
In [1]: !cat wypisz.bat
:: Wyłącza wypisywanie poleceń
@echo off
echo Obiekt SList ma wiele ciekawych atrybutów.
echo Poszczególne atrybuty wypiszą:
echo .l (.list) - listę linii stdout/stderr
echo .n (.nlstr) - listę linii połączoną znakiem nowej linii
echo .s (.spstr) - listę linii połączoną spacjami
echo .p (.paths) - listę obiektów ścieżek (wymaga paczki path.py)
echo ...są też przydatne metody, jak np. grep 1>&2
In [2]: a = !wypisz.bat

In [3]: a
Out[3]:
['Obiekt SList ma wiele ciekawych atrybutów.',
 'Poszczególne atrybuty wypiszą:',
 '.l (.list) - listę linii stdout/stderr',
 '.n (.nlstr) - listę linii połączoną znakiem nowej linii',
 '.s (.spstr) - listę linii połączoną spacjami',
 '.p (.paths) - listę obiektów ścieżek (wymaga paczki path.py)',
 '...są też przydatne metody, jak np. grep ']

In [4]: a.n
Out[4]: 'Obiekt SList ma wiele ciekawych atrybutów.\n
Poszczególne atrybuty wypiszą:\n.l (.list) - listę linii
stdout/stderr\n.n (.nlstr) - listę linii połączoną znakiem
nowej linii\n.s (.spstr) - listę linii połączoną spacjami\n.
p (.paths) - listę obiektów ścieżek (wymaga paczki path.py)\n
...są też przydatne metody, jak np. grep '
```

```
In [5]: a.grep('^.[.] ')
Out[5]:
['.l (.list) - listę linii stdout/stderr',
 '.n (.nlstr) - listę linii połączoną znakiem nowej linii',
 '.s (.spstr) - listę linii połączoną spacjami',
 '.p (.paths) - listę obiektów ścieżek (wymaga paczki path.py)']
```

## WKLEJANIE DO IPYTHONA ZE ZNAKIEM ZACHĘTY

Podczas wklejania logów przekopiowanych z interpretera Pythona (zaczynających się od `>>>`) lub IPython'a (zaczynających się od `In[X]:` lub `...:`) znak zachęty (ang. *prompt sign*) oraz znaki wskazujące wcięcia zostaną usunięte, a właściwy kod zostanie wykonany. Przykład można znaleźć w Listingu 6.

### Listing 6. Wklejanie kodu z interpretera do IPython'a

```
In [4]: >>> import ctypes
...: >>> ctypes.c_int32(0xffcaffe)
...:
Out[4]: c_long(268218366)

In [5]: In [10]: import operator
...:
...: In [11]: from functools import reduce
...:
...: In [12]: reduce(operator.floordiv, (128, 10, 2))
...:
Out[5]: 6
```

## KONFIGURACJA

IPython ma system profili, które przechowywane są w katalogu `.ipython` w katalogu domowym. Domyślnie używany jest profil `default`. Można także stworzyć własny, przez uruchomienie:

```
ipython profile create [<nazwa profilu>]
```

Komenda ta stworzy katalog `profile_<nazwa profilu>` w folderze `.ipython`. W przypadku niepodania nazwy profilu, do domyślnego profilu (`profile_default`) dodany zostanie plik konfiguracyjny – `ipython_config.py`.

Istniejące profile można wylistować przez:

```
ipython profile list
```

Utworzony profil można uruchomić, wykonując:

```
ipython --profile=<nazwa profilu>
```

W katalogu naszego profilu będą interesowały nas dwie rzeczy: plik `ipython_config.py` oraz katalog `startup` – to tam będziemy dodawać skrypty, które zostaną uruchomione podczas inicjalizacji interpretera.

Wygenerowany `ipython_config.py` zawiera zakomentowane ustawienia, które wystarczy odkomentować i zmienić ich wartość wedle naszego upodobania. Jako zapalony CTF-owiec używam następujących opcji:

- › Wyłączenie wyświetlania informacji startowych podczas włączania shella:

```
c.TerminalIPythonApp.display_banner = False
```

- › Zmiana trybu interakcji podczas uruchamiania `ipython <skrypt>` – ustawienie opcji spowoduje, że wcześniej wymieniona komenda uruchomi skrypt w kontekście IPython'a, dzięki czemu pozostaniemy w interpreterze z zachowanymi efektami wykonania skryptu:

```
c.TerminalIPythonApp.force_interact = True
```

- › Ustawienie wcześniej opisanej magicznej komendy `autocall`:

```
c.InteractiveShell.autocall = 1
```

- › Nieprzepisywanie poprawionej komendy przez `autocall` (`foo 5 -> foo(5)`):

```
c.InteractiveShell.show_rewritten_input = False
```

- › Zmiana ustawienia pokazywania uzupełnień – w IPythonie 5.1, którego używam, domyślne ustawienie uzupełniania to `'multicolumn'`; powoduje ono pokazywanie uzupełnień w kolumnach (co można zaobserwować na Rysunku 1) – jest to natomiast uciążliwe, gdy chcemy szybko zobaczyć wszystkie atrybuty danego obiektu. Poniższe ustawienie wypisuje wszystkie dostępne atrybuty:

```
c.TerminalInteractiveShell.display_completions = 'readlinelike'
```

- › Podświetlanie nawiasu odpowiadającego temu, na którym obecnie mamy karetkę:

```
c.TerminalInteractiveShell.highlight_matching_brackets = True
```

- › Automatyczna zmiana tytułu terminala, gdy jesteśmy w shellu:

```
c.TerminalInteractiveShell.term_title = True
```









reklama

Szkolenie dla Ciebie lub Twojego zespołu

# Podstawy uczenia maszynowego w języku Python

Rozpocznij swoją przygodę z machine learning! Wejdź do świata prężnie rozwijającej się dziedziny predykcyjnej analizy danych, która staje się standardem w obecnych czasach prawdziwej powodzi danych.

Skorzystaj z 10% zniżki na wszystkie szkolenie otwarte z autorskiej oferty Sages ważnej przy zamówieniach złożonych do końca listopada 2016 r. Hasło: PROGRAMISTAMAG

 analitycy danych	 2 dni	 16h	 zdalnie lub stacjonarnie
 programiści	 trenerzy praktycy	 różne lokalizacje	 projekty indywidualne

- › Ograniczenie wyświetlania dużych kolekcji (list, zbiorów, krotek czy słowników) do podanej długości – zapobiega przed nadmiernym wypełnieniem konsoli:

```
c.PlainTextFormatter.max_seq_length = 200
```

Kolejną przydatną rzeczą są skrypty uruchamiane podczas inicjalizacji interpretera. Skrypty takie można stworzyć w katalogu `.ipython/profile_<profil>/startup`. Jak można przeczytać w znajdującym się tam pliku `README`, skrypty umieszczone w `startup` wykonują się w porządku leksykograficznym. Moje skrypty inicjalizacyjne dla IPython z Pythonem 2 to `00-imports.py` oraz `01-shortcuts.py`. Można je znaleźć w Listingach 7 oraz 8.

**Listing 7. Skrypt 00-imports.py – importowanie często używanych bibliotek**

```
import sys, os, datetime, time, ctypes

# external modules
import pwn, z3
from pwn import *
```

**Listing 8. Skrypt 01-shortcuts.py – funkcje przyspieszające operacje enkodowania/dekodowania stringów**

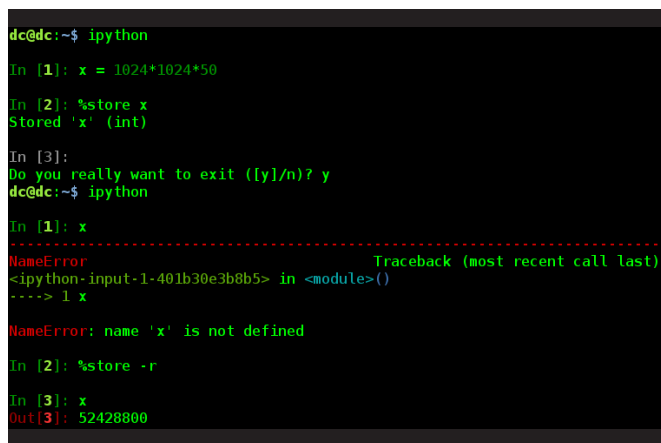
```
# Faster string hex encode/decode
def eh(s):
    return s.encode('hex')

def dh(s):
    return s.decode('hex')
```

## ROZSZERZENIA

Rozszerzenia IPython to moduły, które mogą dodawać nowe magiczne komendy, przechowywać stan między sesjami interpretera, transformować wejście od użytkownika – czy to w formie łańcucha znaków, czy już jako drzewo składniowe (ang. *abstract syntax tree*) Pythona. Kilka z nich jest dołączonych do instalacji IPython. Są to na przykład:

- › autoreload – pozwala na ustawienie interpretera tak, żeby automatycznie przeładowywał moduły, przed wykonaniem kolejnego wejścia.
- › storemagic – pozwala na przechowywanie zmiennych między kolejnymi sesjami IPython. Można to zobaczyć na Rysunku 3.



Rysunek 3. Zapisywanie i przywracanie zmiennych między interaktywnymi sesjami

Inne można znaleźć w spisie rozszerzeń na wiki projektu IPython (<https://github.com/ipython/ipython/wiki/Extensions-Index>) lub

przeoglądając tag Framework::IPython na PyPI (<https://pypi.python.org/pypi?action=browse&c=586>).

Aby skorzystać z rozszerzenia, należy je załadować magiczną komendą `%load_ext <rozszerzenie>`. Można także dodać nazwę rozszerzenia do listy `c.InteractiveShellApp.extensions` w skrypcie konfiguracyjnym `ipython_config.py`.

W przypadku, gdy chcemy wyłączyć rozszerzenie, można skorzystać z `%unload_ext <rozszerzenie>`. Przydatne do debugowania własnych rozszerzeń może być także `%reload_ext <rozszerzenie>`, które wyładowyduje i ponownie ładuje dane rozszerzenie.

Implementacja własnego rozszerzenia polega na napisaniu skryptu (w importowalnej ścieżce, czyli znajdującej się w `sys.path` lub w `.ipython/extensions`), zawierającego dwie funkcje – `load_ipython_extension(ipython)` oraz `unload_ipython_extension(ipython)`, wykonywane odpowiednio, gdy rozszerzenie jest ładowane oraz wyładowywane. Argumentem przekazywanym do funkcji jest instancja `InteractiveShell`, która pozwala na interakcję z interpreterem (w interpreterze jest ona zwracany przez `get_ipython()`).

Obiekt ten pozwala między innymi na dodawanie czy usuwanie zmiennych oraz na tworzenie nowych magicznych komend. Służą do tego odpowiednio funkcje `push()`, `drop_by_id()` oraz `register_magic_function()`.

W Listingach 9 oraz 10 można zobaczyć implementację oraz działanie przykładowego rozszerzenia – `stopwatch`. Dodaje ono magiczną komendę działającą jak stoper.

**Listing 9. Własne rozszerzenie IPythona dodające stoper**

```
from __future__ import print_function
import datetime

global start_time
start_time = None

def stopwatch(line):
    global start_time

    if not start_time:
        start_time = datetime.datetime.now()
        print("Stopwatch started at", start_time)

    else:
        end_time = datetime.datetime.now()
        diff = end_time - start_time
        print("Stopwatch over. Passed", diff)

        start_time = None

def load_ipython_extension(ipython):
    ipython.register_magic_function(stopwatch)

def unload_ipython_extension(ipython):
    pass
```

**Listing 10. Wykorzystanie stopera**

```
In [1]: %load_ext stopwatch
In [2]: %stopwatch
Stopwatch started at 2016-09-25 20:13:11.285353
In [3]: from time import sleep
In [4]: sleep(10)
In [5]: %stopwatch
Stopwatch over. Passed 0:00:17.149241
```

## NIE TYLKO SHELL

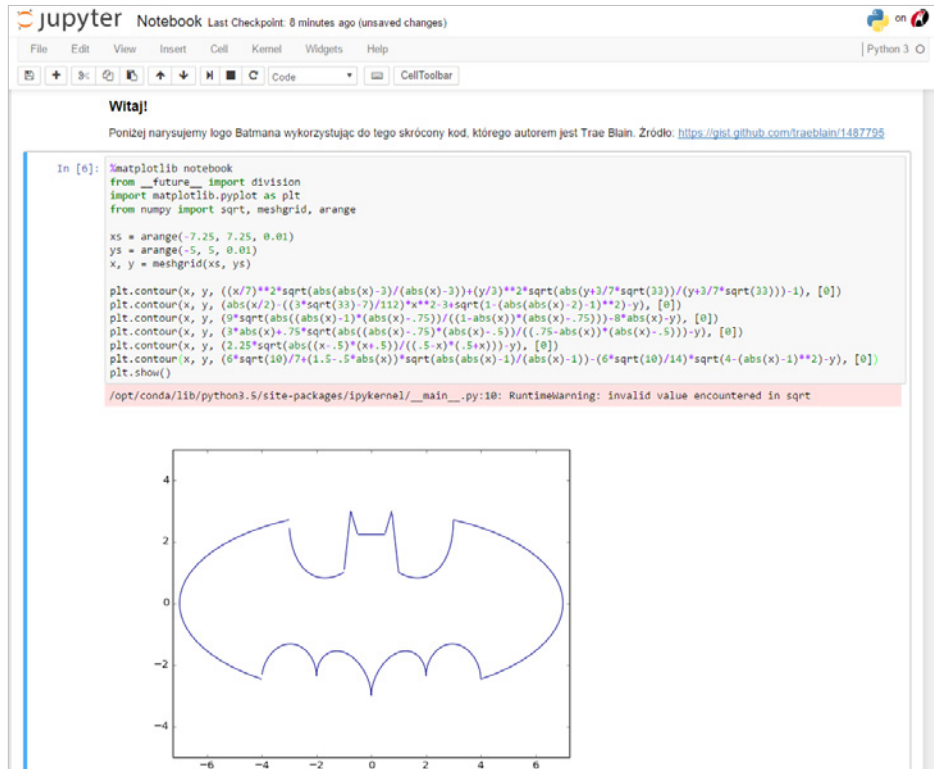
IPython jest nie tylko konsolą, ale także jądrem (ang. *kernel*) dla projektu Jupyter. Ten, kiedyś nazywany IPython Notebook, jest śro-

dowiskiem do interaktywnej analizy danych w przeglądarce z wykorzystaniem różnych języków programowania.

Praca z notatnikiem Jupytera polega na tworzeniu edytowalnych komórek (ang. *cells*), w których wybieramy język – czy to programowania, czy formatowania tekstu (np. Markdown, w tym z LaTeX poprzez silnik MathJax czy reStructuredText) – a następnie uruchamianiu danego kodu. W przypadku języków formatujących tekst uruchomienie spowoduje wyrenderowanie tekstu w komórce. W przypadku kodu zostanie on wykonany, a jego wyjście będzie wyświetlone.

Poza tekstem czy kodem, w komórkach notatnika można także wizualizować dane, wykorzystując na przykład moduł matplotlib, oraz renderować tabele na podstawie obiektów DataFrame z modułu pandas.

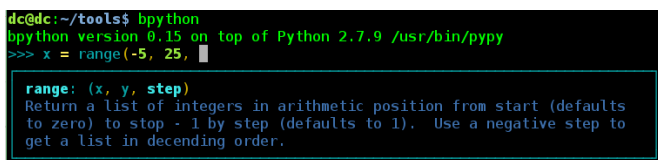
Środowisko to dobrze nadaje się do analizy danych, obliczeń numerycznych, prototypowania, jak i tworzenia prezentacji. Zostało ono przedstawione na Rysunku 4.



Rysunek 4. Jupyter – przykładowy notebook rysujący logo Batmana

## ALTERNATYWY, INNE JĘZYKI

Podczas codziennej pracy warto wykorzystywać narzędzia zwiększające produktywność. IPython, wraz z funkcjonalnościami, które dostarcza, niewątpliwie jest jednym z nich. Nie jest oczywiście jedynym tego typu projektem: w przypadku języka Python alternatywami mogą być bpython oraz ppython. Funkcjonalnie oba projekty są trochę w tyle w stosunku do IPython (nie mają na przykład magicznych komend czy uruchamiania poleceń powłoki systemowej). Ich zaletą może być natomiast nieco inny interfejs czy – w przypadku bpythona – wyświetlania pomocy na bieżąco (zostało to przedstawione na Rysunku 5). Istnieją także kombinacje obu alternatyw – bpythona oraz ppythona z projektem IPython – bipython oraz ppython.



Rysunek 5. Wyświetlanie pomocy na bieżąco w bpythonie

W przypadku Jupytera – można w nim pisać w ponad 40 językach programowania. Pełną listę kerneli można znaleźć na <https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages>.

W innych językach programowania ciekawe do zobaczenia mogą być projekty:

- › REPL (ang. *Read Eval Print Loop*) – interaktywna konsola do Javy – <https://github.com/albertlatacz/java-repl>.
- › Geordi – bot do irca ewaluujący kod C++; co prawda nie jest to interaktywna konsola, ale to z całą pewnością ciekawy projekt. Okazuje się bardzo przydatny, gdy podczas komunikacji przez irca chcemy pokazać przykład lub wytłumaczyć działanie czegoś w języku. Dwie znane mi instancje bota działają w sieci freenode na kanałach #gynvaelstream (bot nazywa się cxx od KrzaQ), ##c++ oraz #geordi (bot nazywa się geordi od twórcy, Eelis) – <http://www.eelis.net/geordi/>.
- › Cling – REPL do C++ – <https://github.com/vgvassilev/cling> – można w nim nawet prototypować aplikacje pisane w Qt (<https://www.youtube.com/watch?v=BrjV1ZgYbbA>).
- › Pry – interaktywny shell dla Ruby – <http://pryrepl.org/>.
- › REPL do Go – <https://github.com/motemen/gore>.



### DOMINIK CZARNOTA

[dominik.b.czarnota+pmag@gmail.com](mailto:dominik.b.czarnota+pmag@gmail.com)

Student AGH, programista Pythona, C++ oraz C#, obecnie zawodowo zajmuje się testami penetracyjnymi oraz szkoleniami z bezpieczeństwa web aplikacji. Aktywny członek koła naukowego Kernel – organizuje wspólne rozwiązywanie zadań crackme, a dawniej warsztaty z C++ i Pythona. Interesuje się szczególnie inżynierią wsteczną i bezpieczeństwem IT. Hobbystycznie bierze udział w rozgrywkach CTF z drużyną Just Hit the Core.