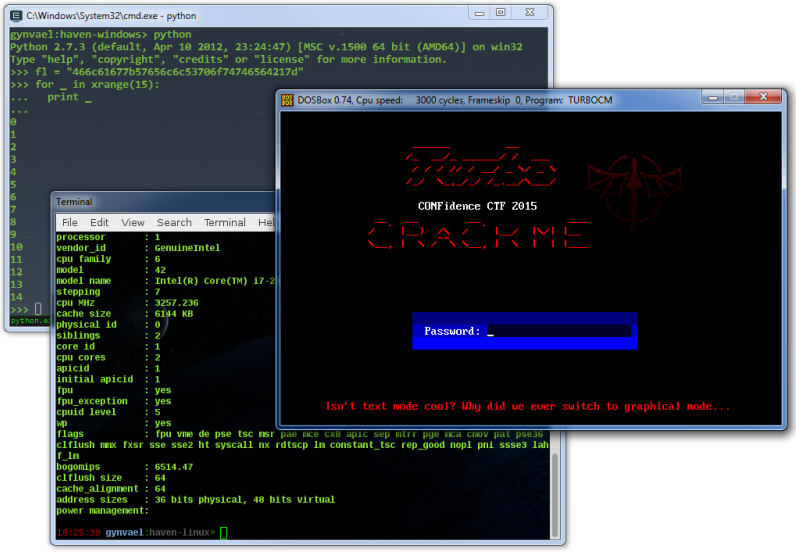
Rozdział 1.

Konsola i interpreter poleceń

1.1. Wykorzystanie interpretera ... 27
1.2. Przekierowania ... 28
1.3. Przykładowe polecenia wykorzystujące przekierowania ... 32
1.4. Bieżący katalog roboczy ... 33
1.5. Zmienne środowiskowe ... 34
1.6. Skrypt startowy ... 38
1.7. Konsola okiem programisty ... 40
Ćwiczenia ... 44
Bibliografia ... 44

Pomimo upływu lat konsola nadal jest jednym z podstawowych narzędzi wykorzystywanych przez zaawansowanych programistów i administratorów systemów – w praktyce konsola i interpreter poleceń to często niezwykle wygodne i przydatne narzędzia, a w niektórych wypadkach stanowią wręcz jedyną możliwość skorzystania z pewnych programów. Zarówno konsola, jak i interpreter to *de facto* standard w przypadku większości unixowych systemów operacyjnych (rys. 1).



Rysunek 1. ConEmu-Maximus[3], DOSBox[4] z uruchomioną aplikacją w trybie tekstowym oraz GNOME Terminal[5]

Na samym początku warto wyjaśnić pewną nieścisłość związaną z używanymi terminami, a mianowicie: czym jest konsola, a czym interpreter poleceń?

[3] Nakładka na emulator konsoli pod kontrolą systemu z rodziny Windows.
[4] Emulator systemu komputerowego pracującego pod systemem DOS.
[5] Domyślny emulator konsoli w niektórych wersjach systemu Ubuntu.

Konsola (nazywana również emulatorem terminalu lub potocznie – terminalem) jest pewnego rodzaju środowiskiem wykonania, z którego mogą (ale nie muszą) korzystać aplikacje – należy od razu zaznaczyć, że w danym momencie z jednej konsoli może korzystać wiele aplikacji. Elementem centralnym jest bufor tekstowy o określonej (ale niekoniecznie stałej) wielkości, który jest w całości lub częściowo wyświetlany w oknie konsoli i który może być pośrednio lub bezpośrednio modyfikowany przez wszystkie aplikacje z niej korzystające. Dodatkowo w danym momencie jedna aplikacja jest tzw. aplikacją pierwszoplanową (*foreground*) – wejście z klawiatury, a czasem również myszki, jest przekazywane właśnie do niej.

Domyślnie dane wypisywane przez aplikację na standardowe wyjście (*stdout*) oraz standardowe wyjście błędów (*stderr*) są również odbierane przez konsolę, która wyświetla tekst, biorąc pod uwagę wszystkie znaki specjalne i sekwencje kontrolne, oraz pozycję kursora. Analogicznie wejście z klawiatury jest przesyłane przez konsolę na standardowe wejście (*stdin*) aplikacji pierwszoplanowej[6].

Z punktu widzenia użytkownika konsola jest po prostu tekstowym interfejsem wykorzystywanym przez niektóre programy. Jedną z podstawowych aplikacji korzystających z konsoli jest właśnie interpreter poleceń (*command processor* lub *command-line interpreter*), nazywany czasem powłoką systemową (*shell*)[7], który zwyczajowo udostępnia użytkownikowi zestaw poleceń i funkcji do:

- Nawigacji w systemie (polecenia typu `cd`, `dir` lub `ls`, `pwd` itp.).
- Konfigurowania środowiska uruchomieniowego, w tym zmiennych środowiskowych (`set`, `export`, `path`, `ulimit` itp.).
- Uruchamiania innych programów z podanymi argumentami i ewentualnymi przekierowaniami strumieni *stdin*/*stdout*/*stderr*.
- Tworzenia skryptów automatyzujących powyższe czynności.

Przykładowe interpretery poleceń to m.in:

- cmd.exe (Windows);
- PowerShell (Windows);
- Bash (GNU/Linux itp.);
- Z shell (GNU/Linux itp.).

W dalszych podrozdziałach opiszę główne koncepty (przekierowania, argumenty, zmienne środowiskowe, katalog roboczy) wykorzystywane przy pracy z interpreterami poleceń, przy czym skupię się na dwóch z nich: cmd.exe (Windows 7) oraz Bash (Ubuntu 14.04.2 LTS). Ponieważ będzie to opis skrócony, zachęcam czytelników do bliższego

[6] W praktyce wszystkie procesy posiadające dostęp do dyskryptora standardowego wejścia – w szczególności procesy potomne – mogą odbierać wprowadzane dane, choć kolejność dostępu do danych może być niedeterministyczna.

[7] Termin „shell" w odniesieniu do interpretera poleceń jest współcześnie rzadziej używany niż w czasach, gdy systemy operacyjne nie posiadały trybu graficznego i były uruchamiane w trybie tekstowym, w którym jedyną powłoką systemową był właśnie interpreter poleceń. Obecnie „shell" pasuje bardziej do graficznych interfejsów, takich jak Windows Explorer czy Ubuntu Unity, jednak jest on nadal używany w obu znaczeniach na systemach unixowych.

przyjrzenia się wybranemu przez siebie interpreterowi we własnym zakresie. Pominę również opis większości poleceń (zarówno wbudowanych, jak i dostępnych w systemie operacyjnym) do operacji na plikach i katalogach – zostały one opisane w wielu innych, łatwo dostępnych publikacjach, np.:

- Bash:
  - http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/c2690.htm
  - http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x3289.htm
- cmd.exe:
  - http://en.wikibooks.org/wiki/Guide_to_Windows_Commands/File_and_Directory_Management
  - http://en.wikibooks.org/wiki/Guide_to_Windows_Commands/File_Commands

1.1. Wykorzystanie interpretera

Poniżej zaprezentowalem kilka przykładów użycia interpretera wraz z krótkim, wysokopoziomowym opisem ich działania. Większość użytych mechanizmów jest wyjaśniona w dalszej części tego rozdziału.

```
(Ubuntu) reset && make test
(Windows) cls && make test
```

Wyczyszczenie okna konsoli oraz rekompilacja projektu i jego uruchomienie (przy wykorzystaniu skryptu Makefile). W przypadku dystrybucji Ubuntu osobiście preferuję stosowanie komendy *reset* zamiast *clear*, ponieważ w emulatorze konsoli, z którego korzystam (GNOME Terminal), *reset* czyści cały bufor tekstowy konsoli, natomiast *clear* jedynie przewija ekran niżej tak, by wydawało się, że został on wyczyszczony – jest to istotne, gdy kompilacja lub uruchomienie projektu powoduje wypisanie dużej ilości tekstu, a dla nas wygodne byłoby szybkie przewinięcie ekranu na początek danych. Alternatywnie można by skorzystać np. z polecenia `make test 2>&1 | less`.

```
(Ubuntu) python gen_input.py | LD_PRELOAD=`pwd`/debug.so ./app
```

Uruchomienie skryptu *gen_input.py*, który generuje na standardowe wyjście pewne dane wejściowe dla aplikacji *app* – są one przekazywane na jej standardowe wejście. Dodatkowo niektóre importowane przez aplikację funkcje zostaną podmienione na funkcje o tej samej nazwie, znajdujące się w dynamicznej bibliotece *debug.so*, która umieszczona jest w obecnym katalogu roboczym.

```
(Ubuntu) for i in {1..1000}; do ./a.out $i; cp out.data
data/$i.data; done
(Windows) for /l %i in (1,1,1000) do @(a.exe %i && copy out.
data data\%i.data)
```

Uruchomienie aplikacji *a.out* / *a.exe* tysiąc razy z kolejnymi liczbami (od 1 do 1000) w pierwszym parametrze. Za każdym razem wygenerowany przez aplikację plik

out.data jest kopiowany do katalogu *data* i nadawana jest mu nowa nazwa w formacie *<liczba z argumentu>.data*.

```
(Ubuntu) watch pep8 test.py
```

Co dwie sekundy (domyślne opóźnienie programu watch) okno konsoli jest czyszczone, a następnie uruchomiony zostaje walidator stylu PEP 8 języka Python. Jest to wygodne rozwiązanie, jeśli w tym czasie w osobnym oknie poprawiamy skrypt *test.py* i chcemy od razu widzieć, czy zmiany odniosły zamierzony skutek (tj. czy ostrzeżenie o nieprawidłowym stylu zniknęło).

Kilka innych przykładów pojawia się również w innych miejscach w tym rozdziale.

1.2. Przekierowania

Jedną z najważniejszych cech interpreterów jest możliwość przekierowania standardowego wyjścia (oznaczanego najczęściej deskryptorem `1`), wyjścia błędów (`2`) i wejścia (`0`) procesów oraz łączenia ich między procesami. Najłatwiej jest to wytłumaczyć na przykładach (które są poprawne zarówno dla Bash, jak i cmd.exe; prawdopodobnie zadziałają również w przypadku większości innych współczesnych interpreterów):

```
program > xyz
```

Wszystkie dane wypisywane na standardowe wyjście przez uruchomiony program trafią do pliku *xyz* (jeśli plik ten istnieje, zostanie nadpisany). Jest to idealne rozwiązanie, gdy danych jest dużo, a my chcemy je na spokojnie przejrzeć lub użyć jako danych wyjściowych w późniejszym terminie.

Alternatywnie można by napisać `program 1>xyz`, choć nie jest to konieczne, ponieważ standardowe wyjście (1) jest domyślnym argumentem dla przekierowania w tę stronę.

```
program 2> errors
```

Podobnie jak wyżej, z tą różnicą, że do pliku *errors* trafią dane wypisywane na standardowe wyjście błędów.

```
program >> xyz
```

Analogicznie jak w powyższych przypadkach, przy czym dane (ze standardowego wyjścia) zostaną dopisane na koniec pliku *xyz*.

```
program < input
```

Program otrzyma dane z pliku *input* na standardowe wejście (tj. czytanie ze standardowego wejścia będzie równoznaczne z czytaniem z pliku *input*). To przekierowanie idealnie nadaje się do powtarzanych wielokrotnie testów aplikacji odczytujących dane ze standardowego wejścia – dzięki temu nie trzeba ich za każdym razem wprowadzać ręcznie.

```
program 2>&1
```

Przekierowanie standardowego wyjścia błędów na standardowe wyjścia; przydatne rozwiązanie, jeśli zachodzi potrzeba przefiltrowania standardowego wyjście błędów (patrz dalej).