Narzędzia deweloperskie: git, make i cmake, gdb, pip i virtualenv

Robert Ryszard Paciorek
<rrp@opcode.eu.org>

2022-10-15

1 Systemem kontroli wersji

Podstawowym zadaniem systemów kontroli wersji jest przechowywanie historii zmian w plikach (np. źródłach jakiegoś projektu, plikach konfiguracyjnych, itp) oraz umożliwianie śledzenia tych zmian (porównywanie wersji, uzyskiwanie informacji o autorach i datach zmian, itd).

W systemach rozproszonych wszystkie kopie repozytoriów są równoprawne (zawierają pełną historię, do każdego z nich mogą być commitowane zmiany, itd), w systemach scentralizowanych występuje pojedynczy serwer do którego commitowane są zmiany i z którego są one pobierane (także operacje związane z historią odwołują się do serwera, a nie lokalnej kopii).

Najpopularniejszymi systemami kontroli wersji są: systemy rozproszone Git (polecenie `git`), Mercurial (polecenie `hg`) i Bazaar (polecenie `bzr`) oraz scentralizowany system kontroli wersji Subversion (polecenie `svn`). Nadal można spotkać używane repozytoria `cvs`.

1.1 Git

1.1.1 Inicjalizacja repozytorium

Mamy dwie podstawowe metody utworzenia repozytorium na którym będziemy pracowali:

- utworzenie nowego, pustego repozytorium: `git init`
- pobranie repozytorium z zewnętrznego źródła `git clone adres`

Adres repozytorium może być m.in. postaci: `https://[login@]serwer/sciezka/` lub `ssh://login@serwer/sciezka/`.

1.1.2 Kopia robocza i poczekalnia

- aktualizacja kopii roboczej: `git checkout [opcje] [sciezka]`
  szczególnie przydatne jest wywołanie `git checkout -f .` umożliwiająca odrzucenie istniejących zmian w kopii roboczej (reset kopii roboczej do stanu repozytorium)
- dodawanie plików/zmian (do poczekalni): `git add sciezka`
- usuwanie zmian (z poczekalni): `git reset HEAD sciezka`
- usuwanie pliku: `git rm sciezka`
  (jest to równoważne z `rm sciezka && git add sciezka`)
- zmiana nazwy (położenia) pliku: `git mv staraSciezka nowaSciezka`
  (jest to równoważne z `git rm staraSciezka; git add nowaSciezka`)
- podgląd zmian kopii roboczej i poczekalni: `git status`
  (aby ignorować jakieś niedodane do repozytorium pliki katalogi należy skorzystać z pliku `.gitignore`)
- różnica kopii roboczej do poczekalni: `git diff [sciezka]`
- różnica kopii roboczej do wskazanej rewizji: `git diff rewizja [sciezka]`
- różnica poczekalni do ostatniej / wskazanej rewizji: `git diff [rewizja] --cached [sciezka]`

- zatwierdzanie zmian z poczekalni (tworzenie rewizji): `git commit [opcje] [sciezka]`, przydatne opcje:
  `-a` powoduje automatyczne dodanie zmian w śledzonych plikach
  `-m "OPIS"` pozwala na podanie opisu zmian z linii poleceń
  `--amend` pozwala na poprawienie ostatniego commitu

1.1.3 Historia zmian

- przeglądanie historii zmian: `git log [opcje] [sciezka]`, przydatne opcje:
  `-p` pokazuje w logu zmian diff pomiędzy rewizjami
  `--name-status` pokazuje nazwy i status (dodanie, usunięcie, modyfikacja) zmienianych plików
  `--graph` pokazuje wykres gałęzi
- listowanie pliku z danej rewizji: `git show rewizja:sciezka`
- listowanie struktury repozytorium z danej rewizji: `git ls-tree rewizja`

1.1.4 Gałęzie

- tworzenie nowej gałęzi: `git branch nazwa`
- scalanie wskazanej gałęzi do aktualnej:`git merge [zasob_zdalny] nazwa`
- utworzenie gałęzi po stronie zdalnej: `git push --set-upstream origin nazwa`
- pobranie / aktualizacja informacji (m.in.) o zdalnych gałęziach: `git fetch`
- listowanie gałęzi: `git branch -av`
- przełączanie się pomiędzy gałęziami (zmiany w kopii roboczej nie zostanę utracone, w przypadku konfliktu przed przełączeniem będzie konieczność ich zaakceptowania lub odrzucenia):
  `git checkout nazwa`

1.1.5 Zasoby zdalne

- listowanie zasobów zdalnych: `git remote -v`
- dodawanie nowego zasobu zdalnego: `git remote add nazwa URL`
- wysyłanie zmian do (wskazanego) zasobu zdalnego: `git push [nazwa]`
- pobieranie zmian z (wskazanego) zasobu zdalnego: `git pull [nazwa]`

1.2 Zadania

Zadanie 1.2.1

1. utwórz katalog `repo1` zawierający dwa pliki tekstowe z dowolną zawartością (różną)
2. utwórz w tym katalogu repozytorium git i sprawdź jego stan (status)
3. dodaj do niego wszystkie pliki z tego katalogu i sprawdź stan repozytorium
4. zakomituj zmiany i sprawdź stan repozytorium

Zadanie 1.2.2

Działając w `repo1` utworzonym w zadaniu 1.2.1

1. zmodyfikuj zawartość jednego z plików
2. sprawdź stan repozytorium
3. zmodyfikuj zawartość drugiego z plików
4. wyświetl różnicę pomiędzy zawartośią kopii roboczej a zawartością repozytorium
5. wyświetl tę różnicę tylko dla jednego z plików
6. zakomituj zmiany
7. wyświetl historię zmian w repozytorium

Zadanie 1.2.3

Sklonuj repozytorium `repo1` jako `repo2`

Zadanie 1.2.4

Działając w `repo2` utworzonym w zadaniu 1.2.3

1. usuń z repozytorium jeden z plików tekstowych
2. zmień nazwę drugiego z plików tekstowych
3. sprawdź stan repozytorium
4. zakomituj zmiany
5. wyświetl historię zmian w repozytorium, z pokazaniem informacji o nazwach zmienianych plików

Zadanie 1.2.5

Zaktualizuj stan `repo1`, tak aby odzwierciedlał zmiany dokonane w `repo2`

Zadanie 1.2.6

1. zmodyfikuj plik tekstowy zarówno w `repo1`, jak i w `repo2` (w różny - sprzeczny sposób)
2. zakomituj zmiany w `repo1` i w `repo2`
3. spróbuj zaktualizować stan `repo1`, tak aby odzwierciedlał zmiany dokonane w `repo2`
4. sprawdź stan repozytorium
5. rozwiąż konflikt
6. sprawdź stan repozytorium
7. wyświetl historię zmian pokazującą wykres gałęzi w których były dokonywane zmiany
8. spróbuj zaktualizować stan `repo2`, tak aby odzwierciedlał zmiany dokonane w `repo1`

Zadanie 1.2.7

1. przywrócić stan kopii roboczej do stanu repozytorium z przed zadania 1.2.6
2. sprawdź stan repozytorium
3. wyświetl zawartość pliku modyfikowanego w zadaniu 1.2.6
4. powróć do normalnego stanu repozytorium
5. sprawdź stan repozytorium

Zadanie 1.2.8

Wyświetl zawartość pliku modyfikowanego w 1.2.6 przed dokonaniem tych modyfikacji bez przełączania kopii roboczej na ówczesny stan repozytorium.

Zadanie 1.2.9

Wyświetl różnicę stanu obecnego pliku modyfikowanego w 1.2.6 i stanu przed dokonaniem tych modyfikacji

Zadanie 1.2.10

1. utwórz nowy branch w `repo1` o nazwie `test` i przełącz się na niego
2. zmodyfikuj plik tekstowy w ramach tego brancha i zakomituj zmiany
3. wyświetl informacje na temat branchy
4. wróć do domyślnej gałęzi "master" i wyświetl zawartość modyfikowanego pliku oraz stan repozytorium

Zadanie 1.2.11

1. zaciągnij zmiany dokonane w zadaniu 1.2.10 z `repo1` do `repo2` tak aby znalazły się one w brachu o nazwie test
2. wyświetl stan repozytorium, informacje na temat branchy oraz dwa najnowsze wpisy z historii repozytorium

2 make

Make jest narzędziem służącym do automatyzowania kompilacji. Jego zadaniem jest ustalenie które pliki potrzebują kompilacji po zmianach w projekcie na podstawie czasu modyfikacji plików źródłowych, wynikowych oraz reguł zapisanych w pliku `Makefile`.

2.1 Wywołanie polecenia make

Wywołanie polecenia make ma postać `make [opcje] [akcja]`.

Wśród opcji warto zwrócić uwagę na:
`-j` pozwalającą określić liczbę równolegle uruchamianych procesów
`-i` powodującą ignorowanie błędów (normalnie make przerywa pracę gdy któreś z zadań nie powiodło się)
`-d` wypisywanie informacji dla debugowania.

Akcja określa regułę z pliku `Makefile` która ma zostać wykonana.

2.2 Plik Makefile

Plik składa się z reguł następującej postaci:

```
nazwa: zaleznosci
        polecenia
```

Pierwsza linia określa iż plik wynikowy (lub akcja) `nazwa` zależy od podanych po dwukropku plików. Druga linia (i ewentualnie kolejne) zawierają polecenia służące do wygenerowania pliku `nazwa` lub realizacji tej akcji. Linie te muszą zaczynać się od znaku tabulacji.

2.2.1 Proste przykłady

```
all:
        echo "Hello world"
```

Każde uruchomienie `make` lub `make all` spowoduje wypisanie "Hello world".

```
all: witaj.txt

co.txt:
        echo cosiu > co.txt

witaj.txt: co.txt
        echo -n "Witaj " > witaj.txt
        cat co.txt >> witaj.txt
```

Uruchomienie `make` spowoduje nadpisanie zawartości pliku `witaj.txt` jeżeli plik `co.txt` jest od niego nowszy. W przypadku braku pliku `co.txt` zostanie on utworzony.

2.2.2 Bardziej zaawansowane pliki Makefile

Poniższy przykład ilustruje kilka rozwiązań stosowanych w plikach Makefile (dla GNU Make), takich jak ustawianie i korzystanie ze zmiennych, iterowanie po plikach, reguły generyczne:

```
# ustawiamy zmienną CONFDIR
CONFDIR=$(HOME)/.xyz/lib

# sprawdzamy czy podany plik/katalog istniej
ifeq ($(wildcard $(CONFDIR)),)
    # a jeżeli nie to zmieniamy wartość zmiennej CONFDIR
    CONFDIR=$(HOME)/.xyz-lib
endif

# ustawienie zmiennej w oparciu o standardowe wyjście innego programu
DATE=$(shell date --iso)

# wartość tej zmiennej może być nadpisana przez wywołanie:
# make CONFDIR=prawdziwa/sciezka

# target installConfig kopiuje wszystkie katalogi src-conf/* do $(CONFDIR)
# zauważ odwłonie do zmiennej makefile'owej CONFDIR vs bashowej inDir w pętli
installConfig:
    @ for inDir in src-conf/*; do \
        install -m 644 -Dt $(CONFDIR)/`basename $$inDir` $$inDir/*" ; \
    done

# target installGSym odnajduje ścieżkę z której będziemy kopiować i kopiuje ...
installGSym:
    # pobieramy output komendy do zmiennej makefilowej w ramach konkretnego targetu
    $(eval GEDASYSDIR := $(shell gschem -c '(display geda-data-path)(gschem-exit)'))
    # modyfikujemy zmienną, a jeżeli jest pusta to ustawiamy wartość domyślną
    $(eval GEDASYMDIR := $(if $(GEDASYSDIR), "$(GEDASYSDIR)/sym", "extra/sym"))
    install -m 644 -Dt $(CONFDIR)/sym $(GEDASYMDIR)/*"

# uzyskanie pliku xyz.o wymaga pliku xyz.c i wykonania komendy:
#   $(CC) -c -o xyz.o xyz.c $(CFLAGS)
# dla dowolnego xyz ...
```

```
%.o: %.c %.h
    $(CC) -c -o $@ $< $(CFLAGS)
# % po lewej stronie : zastępuje dowolny ciąg znaków
#   (podobnie jak * w shellu, ale objemuje też ściezki z katalogami)
# % po prawej stronie : oznacza podstawienie ciągu dopasowanego do
#   znaku % po lewej stronie :
# pod $@ podstawiane jest to co zostało dopasowane do
#   całości napisu po lewj stronie :
# pod $< podstawiany jest pierwszy element z listy zależności
#   (tego co po prawej stronie :)
# pod $^ podstawiana jest całość prawej strony od :
#   (co jest przydatne np. przy linkowaniu)
# standardowo $(CC) zawiera ścieżkę do komilatora C,
#   a $(CFLAGS) zawiera flagi kompilacji
```

2.3 Zadania

Zadanie 2.3.1

Poniższe polecenia utworzą prosty program w C:

```
echo 'void wypisz();' > wypisz.h
echo '#include <stdio.h>' > wypisz.c
echo '#include "wypisz.h"' >> wypisz.c
echo 'void wypisz() { printf("Hello world!\n"); }' >> wypisz.c
echo '#include "wypisz.h"' > main.c
echo 'int main() { wypisz(); return 0; }' >> main.c
```

Może on zostać skompilowany przy pomocy poleceń:

```
gcc -o wypisz.o -c wypisz.c
gcc -o main.o -c main.c
gcc -o hello main.o wypisz.o
```

Napisz plik `Makefile`, który zautomatyzuje kompilację tego projektu. Plik powinien uwzględniać zależności pomiędzy elementami projektu (plik hello wymaga dwóch plików .o, pliki .o wymagają odpowiednich plików .c i .h). Plik powinien pozawalać na określenie używanego kompilatora przy pomocy zmiennej środowiskowej.

Zadanie 2.3.2

Dodaj do pliku `Makefile` z zadania 2.3.1 regułę `clean`, która usunie wszystkie utworzone w wyniku kompilacji pliku. Reguła powinna działać nawet jeżeli w katalogu znajdzie się plik o nazwie `clean`.

3 Generatory plików Makefile

W przypadku większych projektów często stosowane są narzędzia służące do automatycznego generowania plików Makefile (a często także plików nagłówkowych, itp) w oparciu o dostępne biblioteki i automatyczną detekcję zależności pomiędzy plikami.

Wywołanie komendy `make` często musi zostać poprzedzane wywołaniem skryptu `./configure`. Niekiedy także ten skrypt musi zostać poprzedzony wywołaniem polecenia `autoreconf` (z pakietu autoconf/automake).

Aktualnie wiele projektów korzysta do konfiguracji kompilacji (i utworzenia plików Makefile) z `cmake`.

3.1 cmake

Cmake jest narzędziem do zarządzania procesem kompilacji. Jest on niezależny od używanej platformy, kompilatora a także narzędzia automatyzacji kompilacji (oprócz omawianego wcześniej `make` wspiera także kilka innych tego typu narzędzi).

Głównym plikiem konfiguracyjnym programu `cmake` zawierającym reguły budowania danego oprogramowania jest `CMakeLists.txt`.

3.1.1 Wywołanie polecenia cmake

Wywołanie polecenia cmake ma postać `cmake [opcje] sciezka`, gdzie `cmake sciezka` wskazuje katalog zawierający plik `CMakeLists.txt` (typowo główny katalog ze źródłami projektu).

Konstrukcja taka pozwala na łatwe budowanie projektu w innym katalogu niż ten zawierający źródła[1] np.:

```
cd sciezka/do/projektu
mkdir build
cd build
cmake ..
```

Wśród opcji należy zwrócić szczególną uwagą na opcję `-D` pozwalającą na ustawianie flag budowania i kompilacji. Warto także zrócić uwagę na program `ccmake` pozwalający na przeglądanie i modyfikowanie ustawień kompilacji.

3.1.2 Plik CMakeLists.txt

W pliku tym umieszczane są m.in komendy:

- ustawiające zmienne cmake'owe, które mogą być użyte np. do generowania `#define` w plikach nagłówkowych czy też if'owania fragmentów pliku CMakeLists.txt (np. w celu wymagania lub nie danych bibliotek) – `set()` i polecenia `option()`
- operujące na listach (np. plików) i plikach – `list()` i `file()`
- ustawiające opcje kompilacji – `add_compile_options()`
- dodające katalogi z plikami nagłówkowymi – `include_directories()`
- dodające linkowane biblioteki – `link_libraries()`
- dodające cele budowania (akcje w Makefile) związane z kompilacją i linkowaniem – `add_executable()`, `add_library()`
- dodające cele budowania (akcje w Makefile) – `add_custom_command()` `add_custom_target()`
- dodające biblioteki linkowane tylko z wskazanym celem budowania – `target_link_libraries()`
- ustalające zależności pomiędzy celami – polecenia `add_dependencies()`
- realizujące pętle, sprawdzające warunki, definiujące funkcje czy też wypisujące komunikaty – polecenia takie jak , `foreach()`, `if()`, `macro()`, `message()`

Przykład:

```
# wymagamy minimalnej wersji cmake
cmake_minimum_required (VERSION 3.0)

# określamy nazwę naszego projektu i język/kompilator
project("PoznajemyCMAKE" CXX)

option(USE_XLIB "Use X11 (if possible ...)" ON)
```

1. pozwala to uniknąć mieszania źródeł z plikami .o, generowanymi plikami Makefile, itp

```
# ustawiamy flagi kompilatora
add_compile_options("-Wall")

# jeżeli działamy na platformie unix'owej ...
if(UNIX)
    # dodajemy opcję linkowania z biblioteką matematyczną
    # będzie ona dodana do wszystkich wywołań linkera
    link_libraries(m)

    # wyszukujemy bibliotekę X11
    if(${USE_XLIB})
        find_package(X11)

        # jeżeli nie ma to przełączamy USE_XLIB na OFF
        if(NOT ${X11_FOUND})
            set(USE_XLIB OFF)
        # a jeżeli jest to dodajemy do listy Libraries
        else()
            list(APPEND Libraries ${X11_X11_LIB})
        endif()
    endif()
endif(UNIX)

# tworzymy config.h w oparciu o szablon
# wpis #cmakedefine USE_XLIB zostanie zamieniony na #define USE_XLIB
# jeżeli USE_XLIB jest ustawione
configure_file (
    "${PROJECT_SOURCE_DIR}/config.h.in"
    "${PROJECT_BINARY_DIR}/config.h"
)

# wyszukujemy rekurencyjnie pliki z rozszerzeniem .cpp
# w pod katalogu src katalogu źródłowego projektu
file(GLOB_RECURSE Sources "${PROJECT_SOURCE_DIR}/src/*.cpp")

# wypisujemy co znaleźliśmy - debug output ;-)
message(STATUS "Sources:")
foreach(ff ${Sources})
    message(STATUS "  ${ff}")
endforeach()

# ustawiamy że katalogiem bazowm dla plików nagłówkowych
# (podawnym w opcji -I gcc/clang) będzie podkatalog src naszego projektu
include_directories("${PROJECT_SOURCE_DIR}/src")
# typowo robimy to także dla znalezionych nagłówków bibliotek ...

# dodajemy target w postaci tworzenia pliku wykonywalnego o nazwie Run
# powstałego z kompilacji i linkowania źródeł z listy Sources
add_executable(Run ${Sources})

# ustawiamy dodatkowe biblioteki z którymi będzie linkowany plik
# wykonywalny Run w oparciu o listę Libraries
```

```
target_link_libraries(Run ${Libraries})
```

3.2 SCons

SCons jest alternatywnym rozwiązaniem wobec zestawu `cmake` + `make`, pozwalającym na tworzenie plików zarządzających procesem budowania w postaci skryptów Pythona.

4 Debugowanie

4.1 gdb

GNU Debugger (`gdb`) jest narzędziem służącym do podglądania działania innego programu. Umożliwia ustawianie pułapek (powodujących przerwanie wykonywania na wywołaniu określonej funkcji, uruchamianie programu linia po linii, podgląd wartości zmiennych, ... Umożliwia także sprawdzenie co program robił gdy się wywalił (na podstawie pliku core). Jest to narzędzie programistyczne, które niewątpliwie warto poznać, gdyż znacznie ułatwia poprawianie błędów w programach.

Dla skorzystania z części możliwości konieczne jest wkompilowanie informacji debugerowych w program (np. `gcc` z opcją `-g`). Program uruchamiany jest przez wywołanie gdb, zazwyczaj z jedną lub dwiema opcjami - plikiem wykonywalny oraz (ewentualnie) plikiem core. Do najistotniejszych poleceń należy:

- `break plik:linia` - ustawienie breakpointa na wskazanej linii
- `break nazwa_funkcji` - ustawiające breakpointa na daną funkcję
- `rbreak fragment_nazwy_funkcji` - ustawiające breakpointa na podstawie wyrażenia regularnego (np. fragmentu nazwy funkcji)
- `info breakpoint` - lista ustawionych breakpointow
- `watch wyrazenie` - zatrzymanie gdy wartość wyrażenia ulegnie zmianie
- `run` - uruchamiające program (zatrzymuje się na pierwszej napotkanej pułapce)
- `continue` - powoduje kontynuację programu do następnej pułapki
- `step` - powoduje wykonanie kolejnej linii programu, z wchodzeniem w kod funkcji
- `next` - powoduje wykonanie kolejnej linii programu, bez wchodzenia w kod funkcji
- `stepi` / `nexti` - jak `step` / `next`, tyle że nie operuje na liniach a instrukcjach
- `finish` - powoduje dokończenie bieżącej funkcji i zatrzymanie się
- `print zmienna` - wypisanie wartości zmiennej zmiennej/wyrażenia (do elementów składowych odwołujemy się jak w C++)
- `display zmienna` - automatyczne wypisanie wartości zmiennej wyrażenia przy każdym zatrzymaniu (breakpoincie), odwołanie przez `undisplay numer`
- `ptype zmienna` - wypisanie informacji o typie zmiennej (m.in. elementy składowe struktury)
- `info local` - podgląd zmiennych lokalnych
- `info args` - podgląd argumentów funkcji
- `backtrace` - wypisanie historii stosu wywołań funkcji
- `backtrace num` - wypisanie historii stosu wywołań funkcji, wypisuje num pozycji (gdy num mniejsze od 0 to początkowych, gdy większe to końcowych)
- `set zmienna=wartosc` - ustawia wartość zmiennej na podaną
- `call funkcja(argumenty)` - wywołuje podaną funkcję z podanymi argumentami
- `help` wyświetlające pomoc "on-line"
- `list funkcja` - wyświetlenie kodu funkcji

- `list linia1,linia2` - wyświetlenie kodu programu od linii do linii we aktualnym pliku
- `list plik:linia1,plik:linia2` - wyświetlenie kodu programu od linii do linii
- `quit` kończące pracę gdb

Oczywiście to tylko niewielka część wszystkich poleceń gdb, polecam zapoznać się z poszczególnymi grupami poleceń opisywanymi przez `help`.

Uruchamianie poprzez GDB przydaje się także przy debugowaniu problemów typu "segmentation fault" – po zatrzymaniu programu z powodu takiego błędu możemy wydać w gdb polecenie "backtrace" aby zobaczyć w której funkcji wystąpił problem.

4.2 analiza plików binarnych

4.2.1 listowanie symboli

Przy problemach z linkowaniem przydatne może być wyświetlenie symboli które zawiera konkretna biblioteka. Możemy to zrobić poleceniami:

```
# listowanie symboli w pliku wykonywalnym typu elf:
        readelf -sWc   PLIK

# listowanie symboli w pliku .so:
        nm -gC         PLIK.so

# listowanie symboli w pliku .a:
        nm             PLIK.a
```

5 Narzędzia pythonowe

5.1 PIP

Pip jest narzędziem służącym do instalowania modułów pythonowych z dedykowanego repozytorium (managerem/instalatorem pakietów pythonowych). W przypadku Pythona w wersji 3 należy wywoływać go poleceniem `pip3`. Po nazwie polecenia podaje się komendę określającą działania które ma wykonać manager, do najistotniejszych należy zaliczyć:

- `install` ***nazwa*** instaluje podany pakiet
- `install -U` ***nazwa*** aktualizuje podany pakiet
- `uninstall` ***nazwa*** usuwa podany pakiet
- `search` ***słowo*** wyszukuje pakiety w oparciu o podane słowo kluczowe
- `list -v` wypisuje zainstalowane pakiety wraz z informacją o katalogu w którym zostały zainstalowane oraz czy pakiet był instalowany przy pomocy pip, czy w inny sposób (np. przez systemowego menagera pakietów)

Pip automatycznie instaluje zależności pakietu, w niektórych przypadkach może to wymagać zainstalowanych narzędzi i odpowiednich bibliotek C i/lub C++.

Zadanie 5.1.1

Zainstaluj (lub zaktualizuj jeżeli jest obecnie zainstalowany) przy pomocy pip pakiet virtualenv

5.2 Virtualenv

Virtualenv jest narzędziem pozwalającym na tworzenie środowisk pythonowych udostępniających różne wersje modułów. W celu utworzenia nowego środowiska wywołuje się go poleceniem `virtualenv ścieżka`,

gdzie ścieżka wskazuje na miejsce w którym ma zostać utworzone środowisko. Po utworzeniu środowiska można doinstalować wewnątrz niego wymagane wersje bibliotek przy pomocy komendy pip dostępnej w pod-katalogu bin katalogu utworzonego środowiska. Do wywołania programów pythonowych z użyciem danego środowiska należy korzystać z pliku wykonywalnego python3 w pod-katalogu bin danego środowiska. Można także skorzystać z pliku bin/activate w katalogu utworzonego środowiska, poprzez wczytanie go w bierzącej powłoce basha (. scizeka_do_enva/bin/activate) celem uzyskania powłoki w której domyślnym interpreterem pythona będzie ten z utworzonego środowiska „virtualenv". Zmiany w środowisku wprowadzone w wyniku wczytania pliku bin/activate obowiązują tylko w bierzącej powłoce, można je także odwrócić przy pomocy polecenia deactivate.

```
# tworzenie nowego środowiska
virtualenv /tmp/py1
# użycie pip wewnątrz środowiska
/tmp/py1/bin/pip3 list
# wywołanie pytana wewnątrz środowiska
/tmp/py1/bin/python3

# "przełączenie" na pythona z środowiska
. /tmp/py1/bin/activate
# teraz aby użyć pip lub wewnątrz środowiska wystarczy uruchomić
pip3 list
python3
```

Informacja

Virtualenv nie jest chroot'em - Python z tak utworzonego środowiska „widzi" resztę systemu, a także „wie" w jakiej ścieżce jest zainstalowany (zobacz zawartość zmiennej sys.path).

Zadanie 5.2.1

Utwórz środowisko pythonowe typu „virtualenv" w katalogu ytdl i zainstaluj wewnątrz niego pakiet pythonowy youtube-dl.