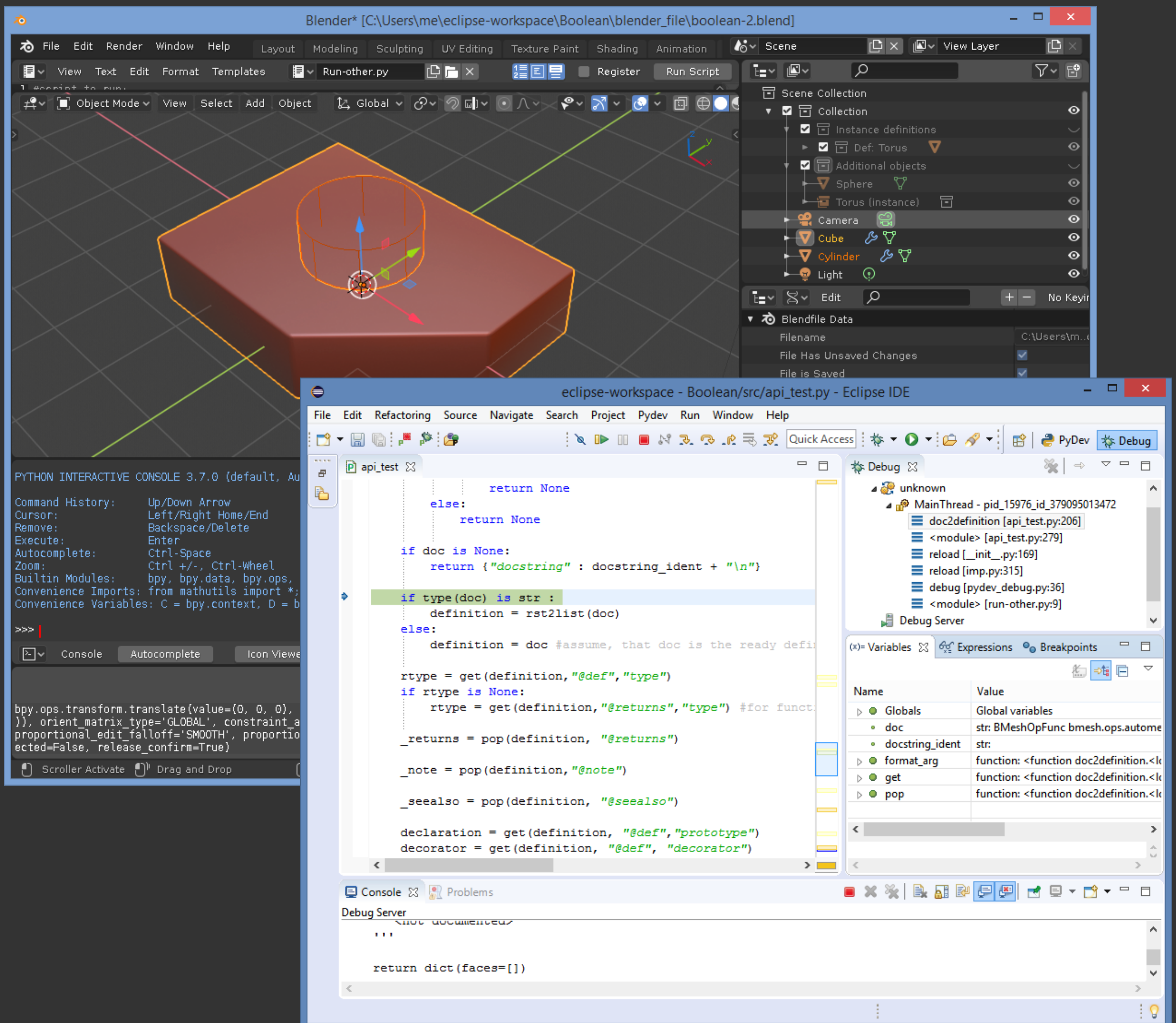
Witold Jaworski



Programowanie dodatków do Blendera 2.8

Pisanie skryptów w języku Python, z wykorzystaniem Eclipse IDE

wersja 2.0

Programowanie dodatków do Blendera 2.8 - wersja 2.0

Copyright Witold Jaworski, 2011-2019.

wjaworski@samoloty3d.pl
http://www.samoloty3d.pl

Ten utwór jest dostępny na licencji Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów zależnych 3.0 Unported.

ISBN: 978-83-941952-0-5

Spis treści

Spis treści 3
Wprowadzenie 4
Konwencje zapisu 5
Przygotowania 6
Rozdział 1. Instalacja oprogramowania 7
1.1 Python (interpreter zewnętrzny) 8
1.2 Eclipse 11
1.3 PyDev 14
Rozdział 2. Pierwsze kroki w Eclipse 18
2.1 Rozpoczęcie projektu 19
2.2 Uruchomienie najprostszego skryptu 24
2.3 Debugowanie 29
Tworzenie aplikacji Blendera 33
Rozdział 3. Skrypt dla Blendera 34
3.1 Sformułowanie problemu 35
3.2 Dostosowanie Eclipse do API Blendera 39
3.3 Opracowanie podstawowego kodu 47
3.4 Uruchamianie skryptu w Blenderze 53
3.5 Ulepszanie skryptu 61
3.6 Przechwytywanie błędów i komunikacja z użytkownikiem 71
Rozdział 4. Przerabianie skryptu na wtyczkę Blendera (add-on) 77
4.1 Dostosowanie struktury skryptu 78
4.2 Dodanie polecenia (operatora) do menu 88
4.3 Implementacja interakcji z użytkownikiem 95
4.4 Dodanie skrótu klawiatury i *Pie Menu* 99
4.5 Implementacja panelu preferencji wtyczki 106
Dodatki 114
Rozdział 5. Szczegóły instalacji 115
5.1 Szczegóły instalacji Pythona 116
5.2 Szczegóły instalacji Java Runtime Environment 120
5.3 Szczegóły instalacji Eclipse i PyDev 122
5.4 Konfiguracja PyDev 129
5.5 Zarządzanie perspektywami projektu Eclipse 133
5.6 Konfigurowanie uruchamiania i debugowania skryptu Pytona 134
Rozdział 6. Inne 138
6.1 Aktualizacja nagłówków API Blendera dla PyDev 139
6.2 Uruchomienie w projekcie PyDev autokompletacji kodu Blender API 142
6.3 Importowanie/linkowanie plików do projektu PyDev 145
6.4 Debugowanie skryptu w Blenderze — szczegóły konfiguracji i obsługi 149
6.5 Co się kryje w pliku *pydev_debug.py*? 160
6.6 Pełen kod wtyczki *object_booleans.py* 162
Bibliografia 167

Copyright Witold Jaworski, 2011-2019.

Wprowadzenie

Językiem skryptów Blendera jest Python. W tym języku zrealizowano wiele przydatnych dodatków do tego programu. Niestety, w Blenderze brakuje czegoś w rodzaju zintegrowanego środowiska programisty (ang. *integrated development environment* — w skrócie *IDE*). „W standardzie" znajdziesz tylko zaadaptowany do podświetlania składni Pythona edytor tekstowy oraz konsolę. To wystarcza do tworzenia prostych skryptów, ale zaczyna przeszkadzać, gdy tworzysz większe programy. Szczególnie uciążliwy jest brak „okienkowego" debuggera.

W 2007r opracowałem artykuł, opublikowany w BlenderWiki, w którym proponowałem użycie w tym charakterze dwóch programów: Open Source: **SPE** (edytor) i **Winpdb** (debugger). Niestety, nie dane mu było być długo użytecznym: udostępniona w 2010r. kolejna wersja Blendera (2.5) otrzymała zupełnie nowe API. Ten Blender używał Pythona w wydaniu 3.x, podczas gdy poprzednie wersje używały wydań z serii 2.x. W dodatku twórcy Pythona zdecydowali się w wersji 3.x zerwać wsteczną zgodność kodu. Co gorsza, edytor SPE został w tym czasie porzucony przez jego twórcę i nie było komu zaadoptować go do Pythona 3.x. (Taki los spotyka wiele mniejszych przedsięwzięć Open Source – są one często efektem czyjegoś hobby). W efekcie opisane w moim artykule rozwiązanie nie działało w Blender 2.5.

W 2011r postanowiłem więc zaproponować nowe środowisko programisty dla tworzenia rozszerzeń Blendera, także oparte wyłącznie o oprogramowanie Open Source. Tym razem mój wybór padł na IDE **Eclipse**, wzbogacone o dodatek do pracy ze skryptami Pythona: **PyDev**. Obydwa produkty były wówczas rozwijane od 10 lat i same w sobie nie zależały od Pythona. (Dzięki temu nie były narażone na taką wsteczną niezgodność kodu, jak SPE i Winpdb). To był lepszy wybór: obydwa produkty są nadal rozwijane, a moja książka („Programowanie dodatków dla Blendera 2.5") pozostała użyteczna z drobnymi poprawkami, przez kolejne 7 lat. Określam ją jako „wersję 1.0". Adaptacja tej książki do kolejnych wersji Blendera (2.6, 2.7) wymagała tak niewielu zmian, że nie zdecydowałem się tworzyć nowych wydań tego poradnika. Zamiast tego umieszczałem je w erracie na stronie tego projektu.

Jednak pod koniec 2018r. Fundacja Blendera opublikowała nową wersję Blendera: 2.8. Gdyby Blender była produktem komercyjnym, ta nowa wersja zapewne otrzymałaby nie numer 2.8, a co najmniej 3.0. W porównaniu z poprzednią wersją wprowadzono w niej wiele głębokich zmian i ulepszeń, świadomie zrywając z wieloma przeżytkami (co powoduje m.in. brak wstecznej zgodności plików). Wsteczna zgodność została zerwana także w przypadku skryptów Pythona. Tym samym nadszedł czas, aby napisać tę drugą edycję tego poradnika („Programowanie dodatków do Blendera 2.8"). Nazywam ją także „wersją 2.0" tej książki.

Uważam, że narzędzia programisty najlepiej przedstawiać na przykładzie pracy nad jakimś konkretnym skryptem. Zdecydowałem się więc opisać tu proces tworzenia wtyczki Blendera pozwalającej szybko wykonywać podstawowe operacje na bryłach: sumę, różnicę, część wspólną. (Chodzi o typowe operacje, za pomocą których projektuje się w 3D części maszyn). Poziom narracji tego poradnika wymaga przeciętnej znajomości Pythona i Blendera. Do zrozumienia fragmentu o tworzeniu wtyczki (Rozdział 4) trzeba także znać podstawowe pojęcia programowania obiektowego, takie jak: „klasa", „obiekt", „instancja", „dziedziczenie". Gdy jest to potrzebne wyjaśniam kilka bardziej zaawansowanych pojęć. (Np. na przykładzie API dla wtyczek tłumaczę, co to jest „interfejs" i „klasa abstrakcyjna"). Ta książka wprowadza w praktyczne podstawy pisania rozszerzeń Blendera. Nie opisuję tu wszystkich zagadnień. Przestawiam za to metody, które stosuję, by je poznawać. Używając ich, będziesz mógł samodzielnie opanować resztę API Blendera (np. tworzenie własnych paneli lub menu).

Konwencje zapisu

Wskazówki dotyczące klawiatury i myszki oparłem na założeniu, że masz standardowe:

- klawiaturę — w normalnym układzie amerykańskim, 102 klawisze;
- myszkę — wyposażoną w dwa przyciski i kółko przewijania (które daje się także naciskać: wtedy działa jak trzeci, środkowy przycisk).

Wywołanie polecenia programu będę zaznaczał następująco:

Menu→Polecenie taki zapis oznacza wywołanie z menu „Menu” polecenia „Polecenie”. W przypadku bardziej zagnieżdżonych menu może wystąpić więcej strzałek!

Panel:Przycisk taki zapis oznacza naciśnięcie w oknie dialogowym lub panelu "Panel" przycisku „Przycisk”.

Naciśnięcie klawisza na klawiaturze:

myślnik pomiędzy znakami klawiszy oznacza jednoczesne naciśnięcie obydwu klawiszy na klawiaturze. W tym przykładzie trzymając wciśnięty Alt, naciskasz K;

przecinek pomiędzy znakami klawiszy oznacza, że je naciskasz (i zwalniasz!) po kolei. W tym przykładzie najpierw G, a potem X (tak, jak gdybyś chciał napisać wyraz „gx”).

Naciśnięcie klawisza myszki:

lewy przycisk myszy

prawy przycisk myszy

środkowy przycisk myszy (**naciśnięte** kółko przewijania)

kółko przewijania (pełni tę rolę, gdy jest **obracane**)

Na koniec: jak mam się do Ciebie zwracać? Zazwyczaj w poradnikach używa się formy bezosobowej („teraz należy zrobić”). To jednak, mówiąc szczerze, czyni czytany tekst mniej zrozumiałym. Aby ta książka była jak najbardziej czytelna, zwracam się do Czytelnika w krótkiej, drugiej osobie („teraz zrób”). Czasami używam także osoby pierwszej („teraz zrobiłem”, „teraz zrobimy”). Tak jest mi łatwiej. Podczas pisania i debugowania kodu w tym opracowaniu traktowałem nas — czyli Ciebie, drogi Czytelniku, i siebie, piszącego te słowa — jako jeden zespół. Może trochę wyimaginowany, ale w jakiś sposób prawdziwy. Przecież pisząc tę książkę ja także się wiele nauczyłem, bo wiedziałem, że każde zagadnienie mam Ci porządnie przedstawić!

Copyright Witold Jaworski, 2011-2019.

Przygotowania

W tej części opisuję, jak zbudować (zainstalować) odpowiednie środowisko programisty (Rozdział 1). Potem zaznajamiam z podstawami konfiguracji i użycia IDE Eclipse, z dodatkiem PyDev (Rozdział 2).

Rozdział 1. Instalacja oprogramowania

Opisywane w tej książce środowisko pracy programisty wymaga zainstalowania trzech składników:

- „zwykłego” interpretera Pythona (jest to interpreter „zewnętrzny”, w stosunku do dostarczanego wraz z Benderem interpretera „osadzonego” w kodzie programu);
- IDE Eclipse;
- dodatku do Eclipse: PyDev;

Ten rozdział opisuje, jak to zrobić.

Zakładam, że masz już zainstalowany Blender. (Podczas pisania tej książki używałem Blendera 2.80 beta. Oczywiście, możesz wykorzystać jakiejkolwiek z jego późniejszych wersji).

- UWAGA: opisywane w tej książce środowisko wymaga 64-bitowego systemu operacyjnego. (Dla Windows 10 jest to wariant domyślny).

Copyright Witold Jaworski, 2011-2019.

1.1 Python (interpreter zewnętrzny)

Najpierw sprawdź, której wersji Pythona używa Twój Blender. W tym celu w Blenderze przejdź do ***Python Console*** (znajdziesz ją np. w standardowej zakładce ***Scripting***) i odczytaj w niej aktualną wersję Pythona (Rysunek 1.1.1):

```
Console   Autocomplete

PYTHON INTERACTIVE CONSOLE 3.7.0 (default, Aug 26 2018, 16:05:01) [MSC v.1900 64 bit (AMD64)]

Command History:     Up/Down Arrow
Cursor:              Left/Right Home/End
Remove:              Backspace/Delete
Execute:             Enter
Autocomplete:        Ctrl-Space
Zoom:                Ctrl +/-, Ctrl-Wheel
Builtin Modules:     bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, blf, mathutils
Convenience Imports: from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

>>>
```

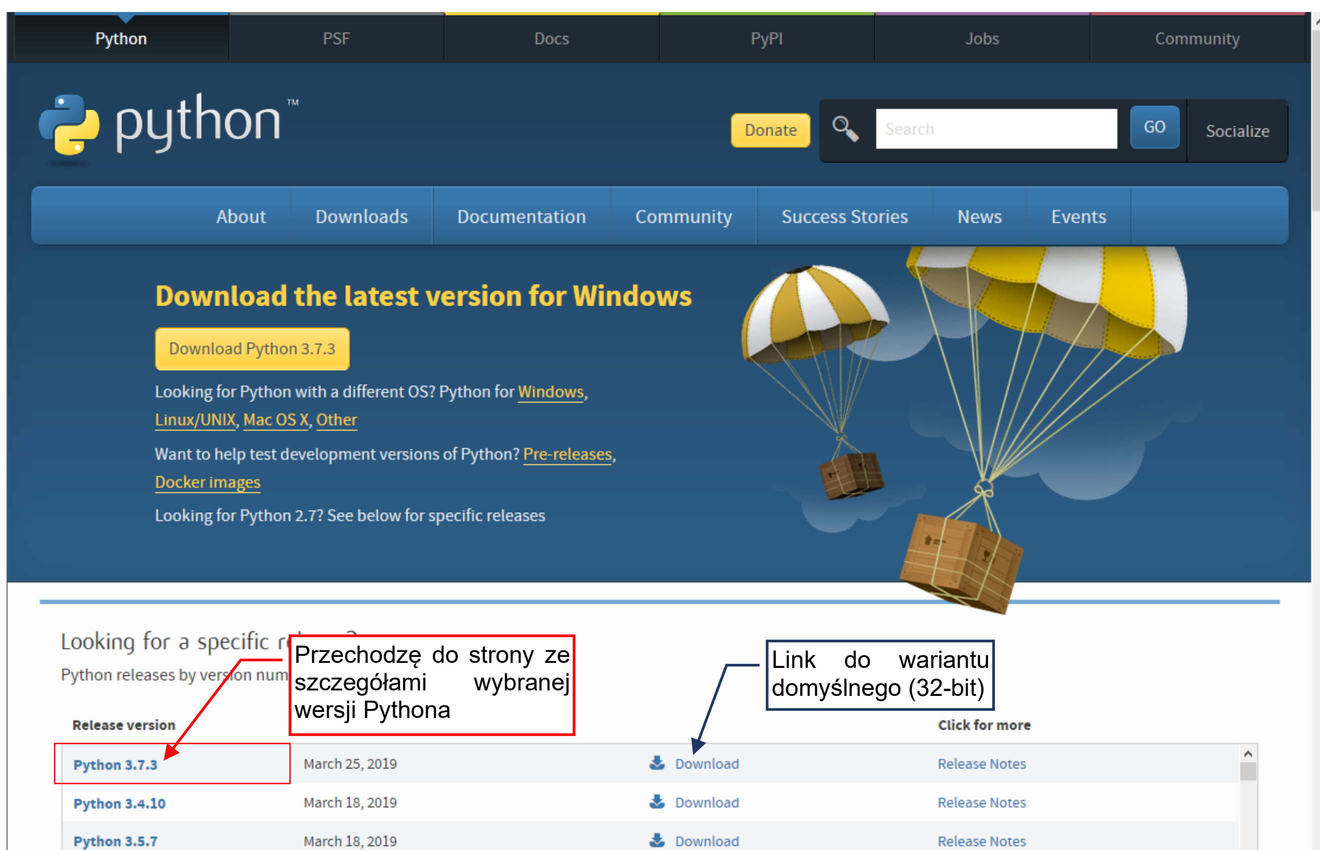
Takie okno znajdziesz w zakładce (*workspace*) ***Scripting***

Odczytaj wersję Pythona

Rysunek 1.1.1 Identyfikacja wersji Pythona, używanej przez Twojego Blendera

Blender z ilustracji powyżej (to wersja 2.80) wykorzystuje Pythona 3.7. Zazwyczaj warto instalować tę samą wersję Pythona jako zewnętrzny interpreter, wykorzystywany w Eclipse.

„Zewnętrzny" interpreter Pythona można pobrać z https://www.python.org/downloads/ (Rysunek 1.1.2):



Rysunek 1.1.2 Wybór wersji Pythona (ekran z maja 2019)

Trzecia cyfra w numerze wersji Pythona oznacza „numer edycji serwisowej", czyli jakichś drobnych korekt zauważonych błędów. Dlatego w przypadku jak powyżej pobieram wersję 3.7.3, gdyż pod względem formalnym jest identyczna z używanym w Blenderze 3.7.0.

Od 2019r. wszystkie wersje Eclipse są wyłącznie 64-bitowe, stąd, na wszelki wypadek, pobieram zewnętrzny interpreter Pythona także w wersji 64-bitowej. Dlatego przechodzę na stronę opisującą szczegóły wybranej wersji (***Python 3.7.3*** – por. Rysunek 1.1.2).

Przechodzę na sam koniec strony z detalami wersji, aby znaleźć tam linki do programów instalacyjnych jej poszczególnych wariantów (Rysunek 1.1.3):

Files

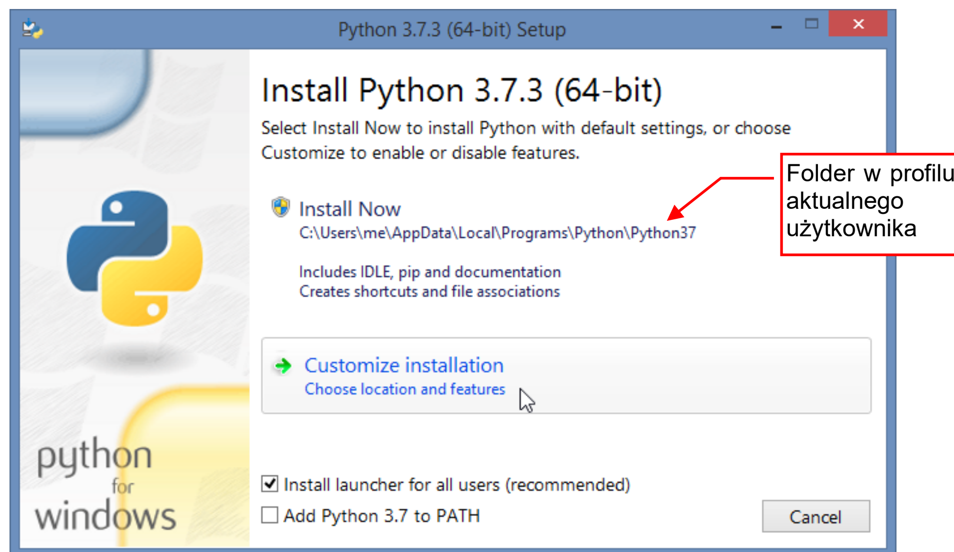
Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		2ee10f25e3d1b14215d56c3882486fcf	22973527	SIG
XZ compressed source tarball	Source release		93df27aec0cd18d6d42173e601ffbbfd	17108364	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	5a95572715e0d600de28d6232c656954	34479513	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	4ca0e30f48be690bfe80111daee9509a	27839889	SIG
Windows help file			7740b11d249bca16364f4a45b40c5676	8090273	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	854ac011983b4c799379a3baa3a040ec	7018568	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	a2b79563476e9aa47f11899a53349383	26190920	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	047d19d2569c963b8253a9b2e52395ef	1362888	SIG
Windows x86 embeddable zip file	Windows		70df01e7b0c1b7042aabb5a3c1e2fbd5	6526486	SIG
Windows x86 executable installer	Windows		ebf1644cdc1eeeebacc92afa949cfc01	25424128	SIG
Windows x86 web-based installer	Windows		d3944e218a45d982f0abcd93b151273a	1324632	SIG

Pobieram 64-bitowy wariant dla Windows

Rysunek 1.1.3 Pobieranie instalatora 64-bitowego wariantu Pythona (z https://www.python.org/downloads/release/python-373/)

Pobieram stamtąd 64-bitowy wariant dla Windows.

Pobrany ze strony program instalacyjny uruchom tak, jak każdy inny instalator pod Windows (Rysunek 1.1.4):



Rysunek 1.1.4 Pierwszy ekran instalacji Pythona

Domyślnie program instaluje pliki Pythona w profilu aktualnego użytkownika. Tak też możesz zrobić. Osobiście jestem nieco staromodny i nie lubię umieszczania programów wykonywalnych w *C:\Users\AppData\Local*, bo potem trudno mi je odnaleźć. Na szczęście mam do mojego komputera uprawnienia Administratora, więc mogłem wybrać w ***Customize Installation*** opcje instalacji „dla wszystkich użytkowników". Ta opcja umieszcza pliki Pythona w folderze *C:\Program Files*. (Szczegółowy opis instalacji — zobacz rozdział 5.1, str. 116).

Copyright Witold Jaworski, 2011-2019.

- Nim pobierzesz zewnętrzny interpreter Pythona, nie zaszkodzi sprawdzić, czy przypadkiem nie masz go już zainstalowanego na swoim komputerze. Spróbuj wywołać w linii poleceń następujący program:
  ```
  python --version
  ```
  Jeżeli masz zainstalowanego Pythona, pojawi się odpowiedź (np. „Python 3.4.0”).

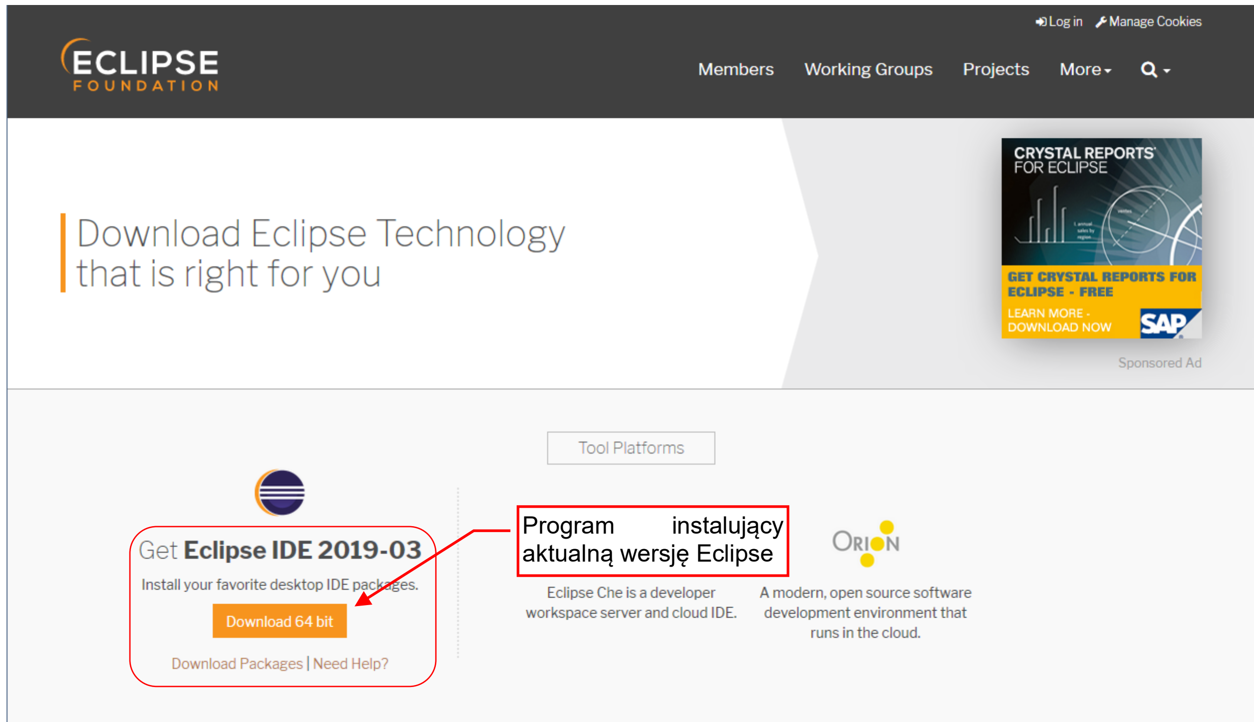
- W razie czego nie przejmuj się specjalnie drobniejszymi różnicami w numerach wersji Pythona. Może się zdarzyć, że na stronach www.python.org nie znajdziesz dokładnie tej samej wersji, która jest skompilowana z Benderem. (Chodzi mi o różnicę w drugiej cyfrze numeru wersji). Wybierz wówczas wersję o najbliższym wyższym numerze. Blender 2.8 używa wyłącznie swojego „wewnętrznego” interpretera Pythona, nawet gdy w systemie jest dostępny taki sam, ale zewnętrzny. Tak więc zazwyczaj wszystko będzie działać, gdy np. zainstalujesz sobie jako zewnętrzny interpreter Pythona w wersji 3.8, zamiast wersji 3.7.

1.2 Eclipse

- Eclipse jest aplikacją Javy i wymaga standardowej maszyny wirtualnej Javy (***JRE***). Od 2019r musi to być JRE w **wariancie 64-bit**. Wersję instalacyjną JRE można pobrać ze strony www.java.com.

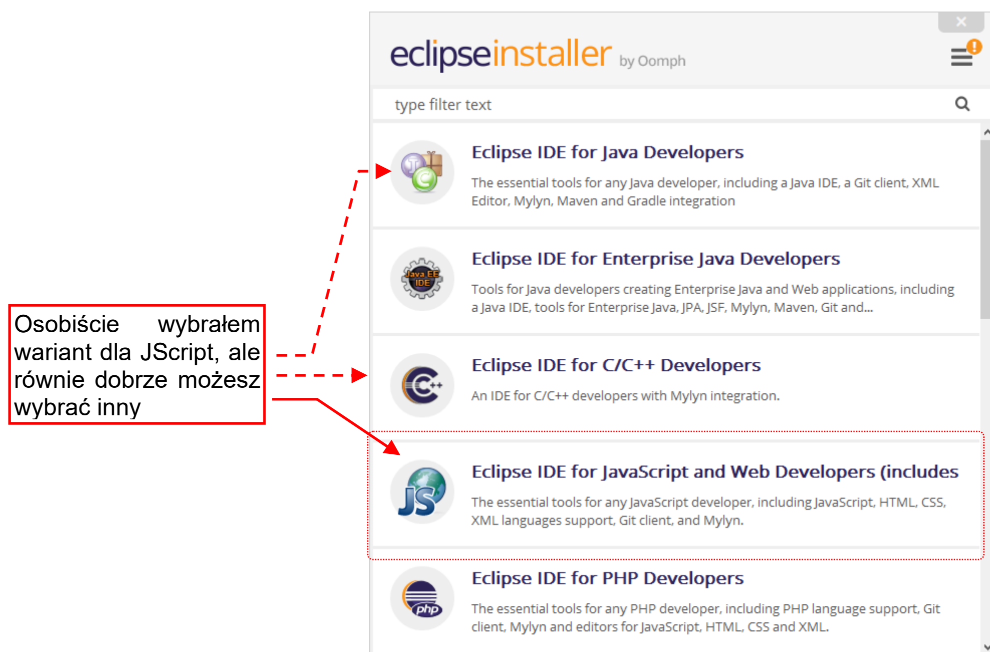
(Szczegółowe omówienie niuansów instalacji JRE znajdziesz w sekcji 5.2, na str. 120).

Przejdź na stronę www.eclipse.org/downloads, i pobierz z niej instalator Eclipse (Rysunek 1.2.1):



Rysunek 1.2.1 Pobieranie instalatora Eclipse (ekran z marca 2011)

Po uruchomieniu programu instalacyjnego zobaczysz okno wyboru wariantu Eclipse (Rysunek 5.3.3):

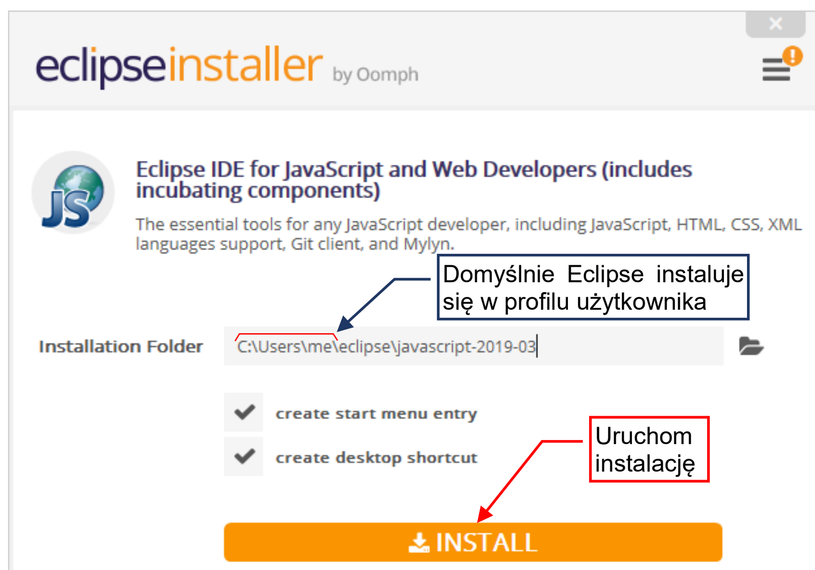


Rysunek 1.2.2 Wybór wariantu Eclipse

Copyright Witold Jaworski, 2011-2019.

Eclipse jest udostępniane w wielu różnych odmianach, przygotowanych dla określonego języka/języków programowania. Nie oznacza to jednak, że nie możesz np. w wersji dla PHP tworzyć programu w C++. Wystarczy dograć odpowiednie wtyczki! To, co widać, to po prostu zawczasu przygotowane, typowe „zestawy wtyczek”, odpowiadających najczęściej występującym potrzebom. Nie ma tu „gotowego” zestawu dla Pythona, więc sugeruję pobrać zestaw z najmniejszą liczbą specyficznych dodatków – np. ***Eclipse for Testers*** lub ***Eclipse IDE for C/C++***. (Szczegóły instalacji Eclipse znajdziesz na str. 122).

Po kliknięciu w wybrany zestaw wyświetla się okno, gdzie można ustalić docelowy folder na pliki programu:

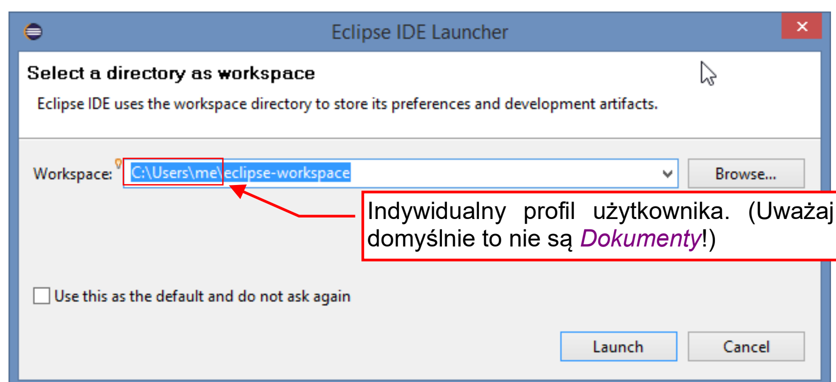


Rysunek 1.2.3 Opcje instalacji Eclipse

Zwróć uwagę, że domyślnie Eclipse instaluje się w profilu użytkownika. (W przypadku na ilustracji to folder *C:\Users\me*). Tworzy w nim podkatalog *eclipse*, w którym tworzy podkatalog dla każdego instalowanego wariantu (w moim przypadku to *javascript-2019-03*, gdzie „2019-03” to symbol wersji Eclipse).

W dalszej części książki będziemy musieli znaleźć wśród plików Eclipse (a dokładniej - zainstalowanych wtyczek) pewien szczególny folder. Dlatego pozwalam Eclipse zainstalować się w domyślnym miejscu, aby odpowiadało to sytuacji na komputerach większości Czytelników.

Po zainstalowaniu uruchom Eclipse, by sprawdzić, czy wszystko działa. Najpierw wyświetli się okno wyboru tzw. „przestrzeni roboczej” (*workspace*):

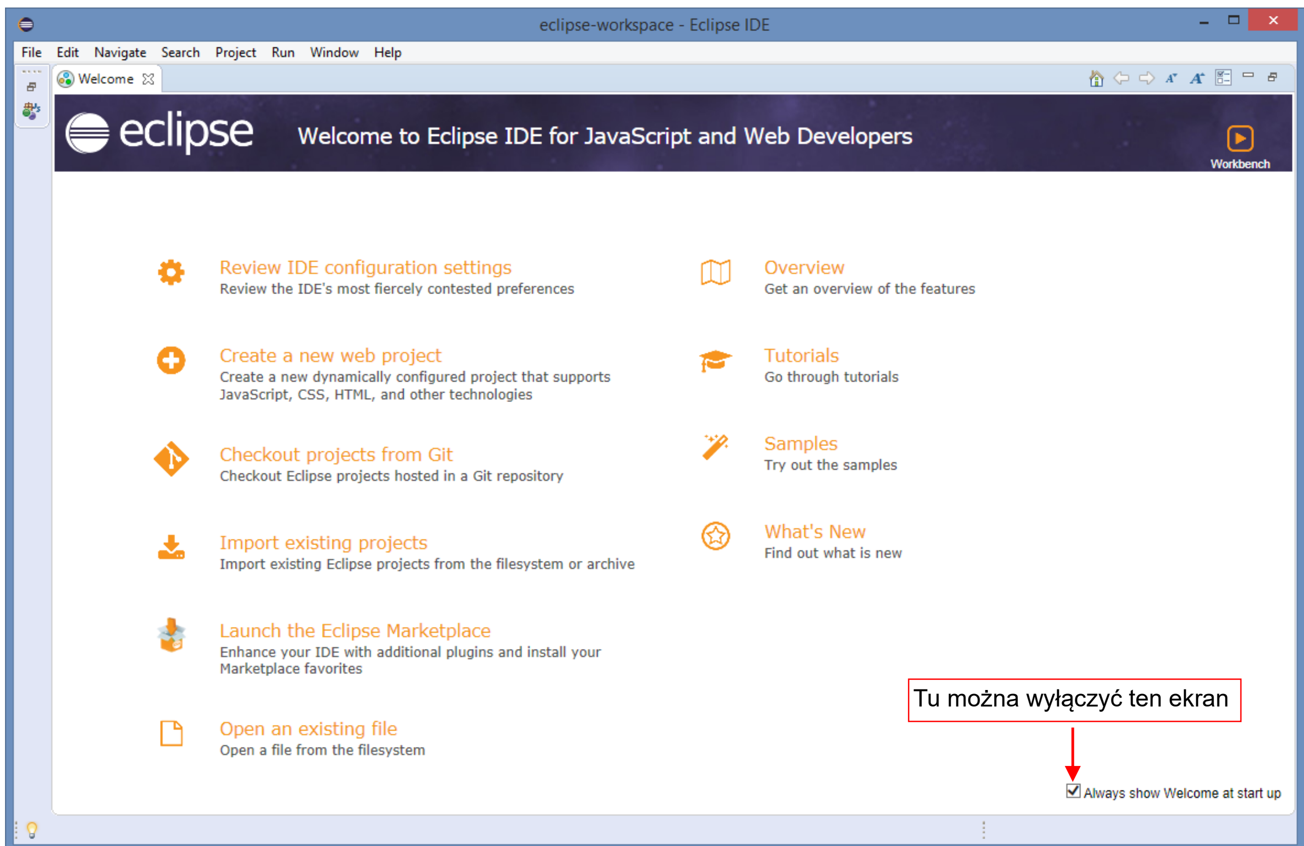


Rysunek 1.2.4 Pytanie o folder dla przyszłych projektów

Eclipse domyślnie tworzy w profilu użytkownika folder o nazwie *eclipse-workspace*. Każda przestrzeń robocza zawiera także własny zestaw preferencji Eclipse, m.in. konfigurację interpretera Pythona. Zwróć uwagę, że domyślnie folder *eclipse-workspace* jest umieszczony w katalogu głównym profilu użytkownika. (W tym przykładzie

to użytkownik *me*). To wcale nie jest katalog *Dokumenty* — tylko poziom wyżej. To proste przełożenie konwencji z *Unix/Linux*. Jeżeli jesteś przyzwyczajony, że wszystkie swoje dane trzymasz w katalogu *Dokumenty* — zmień tę ścieżkę. Eclipse utworzy odpowiedni folder na dysku. Zazwyczaj do pracy wystarczy Ci jedna przestrzeń robocza. W tym miejscu będzie tworzył foldery dla kolejnych projektów. Na każdy projekt składa się parę własnych plików Eclipse oraz Twoje skrypty.

Przy pierwszym uruchomieniu okno Eclipse wyświetla ekran *Welcome*, ze skrótami do kilku miejsc w Internecie, związanych z tym środowiskiem (Rysunek 1.2.5):



Rysunek 1.2.5 Wygląd Eclipse przy pierwszym uruchomieniu

Teraz musimy dodać do Eclipse wtyczkę, dostosowującą to środowisko do skryptów Pythona: PyDev.

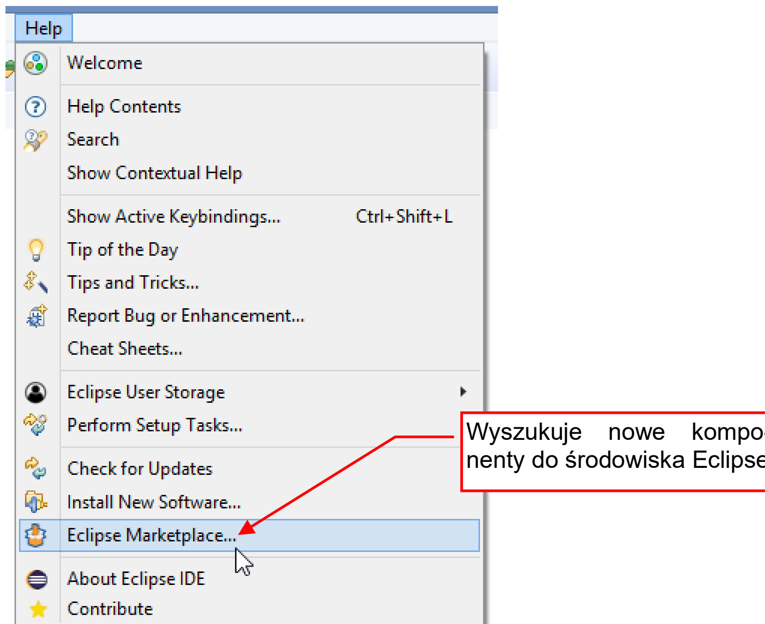
Copyright Witold Jaworski, 2011-2019.

1.3 PyDev

Do instalacji PyDev wykorzystamy wewnętrzny mechanizm Eclipse, przeznaczony do obsługi wtyczek.

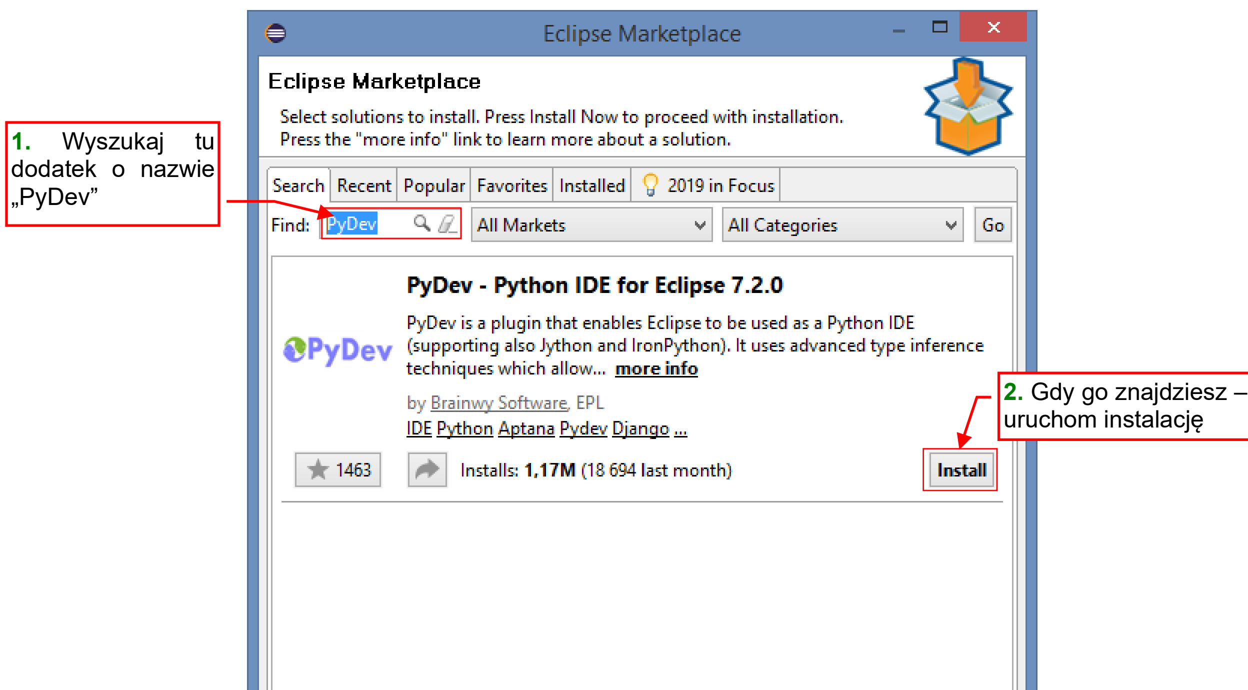
- UWAGA: aby wykonać opisane w tej sekcji czynności, musisz być podłączony do Internetu.

Nowe wtyczki dodaje się do środowiska poleceniem ***Help→Eclipse Marketplace*** (Rysunek 1.3.1):



Rysunek 1.3.1 Polecenia dodające do środowiska nową wtyczkę

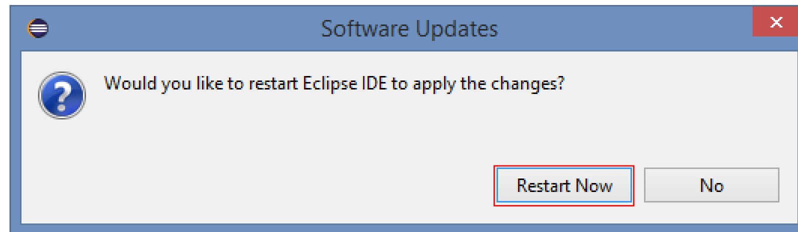
W oknie, które się pojawi, wyszukaj frazę „PyDev" (Rysunek 1.3.2):



Rysunek 1.3.2 Instalowanie dodatku PyDev

Gdy znajdziesz produkt *PyDev – Python IDE for Eclipse*, uruchom jego instalację. Potwierdzaj kolejne wyświetlane ekrany: domyślny zestaw komponentów do instalacji, umowy licencyjne.

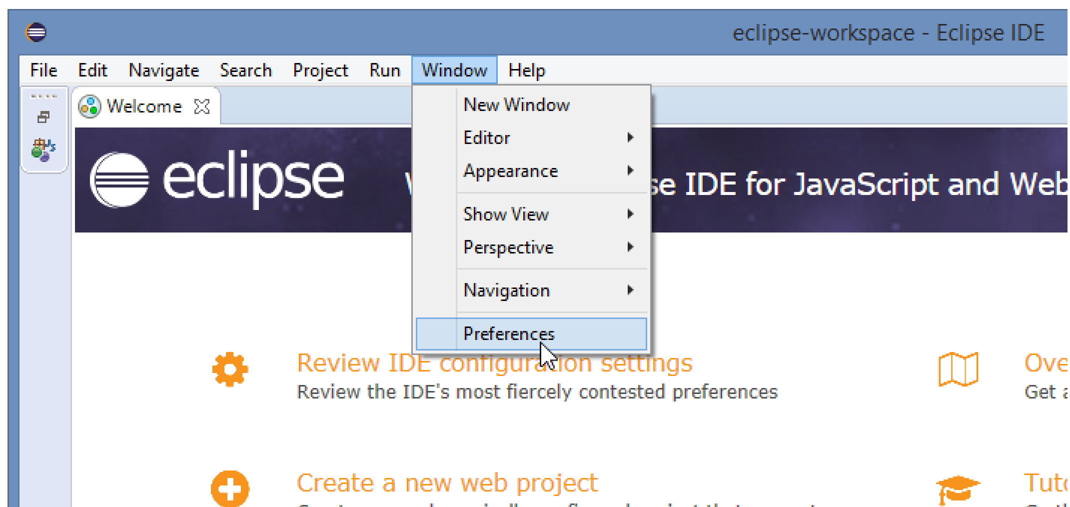
Na koniec pojawi się informacja o konieczności restartu Eclipse (Rysunek 1.3.3):



Rysunek 1.3.3 Okno końcowe

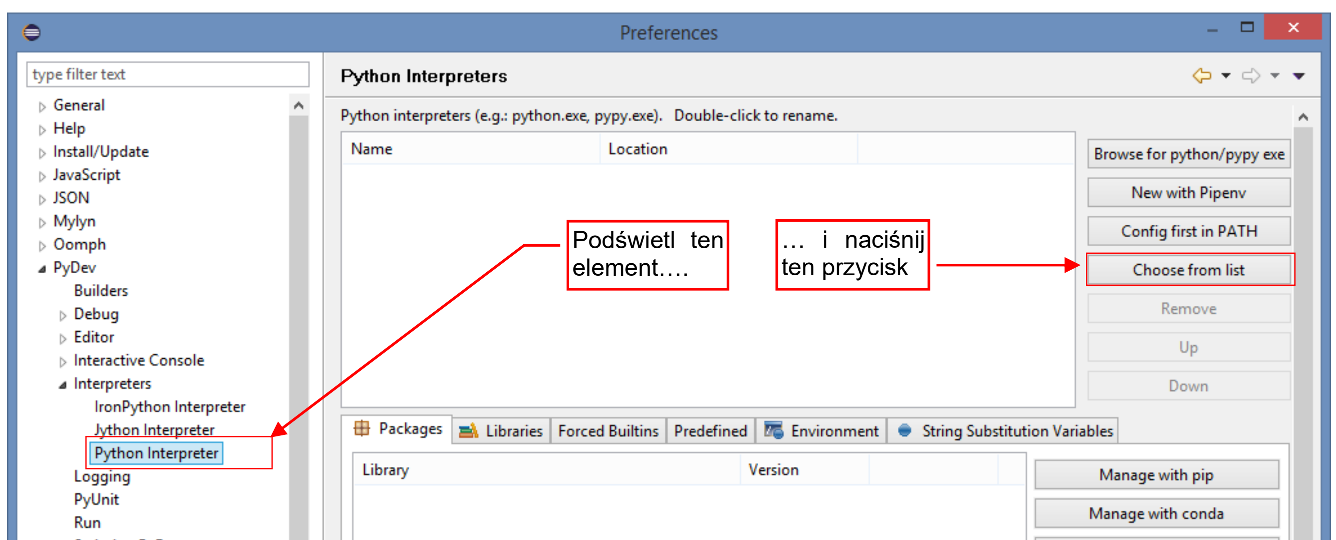
Na wszelki wypadek warto go wykonać.

Dla każdej przestrzeni roboczej (*workspace* – por. str. 12) Eclipse zapisuje oddzielną konfigurację. Wśród tych parametrów jest też domyślny interpreter Pythona. Warto go od razu ustawić. W tym celu wywołaj polecenie ***Window→Preferences*** (Rysunek 1.3.4):



Rysunek 1.3.4 Przejście do parametrów przestrzeni roboczej (*workspce*)

W oknie *Preferences* rozwiń sekcję ***PyDev*** i w podsekcji ***Interpreters*** podświetl pozycję ***Python Interpreter*** (Rysunek 1.3.5):

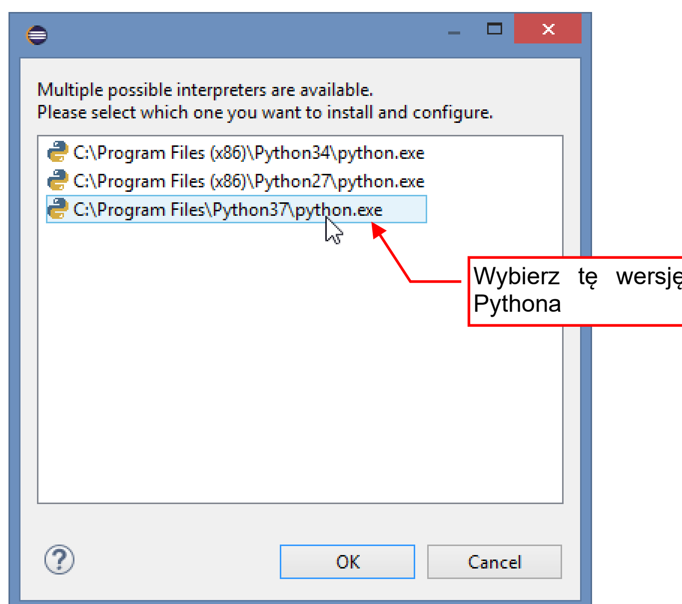


Rysunek 1.3.5 Wywołanie wyszukiwania interpreterów Pythona

Następnie wystarczy nacisnąć przycisk ***Choose from list***.

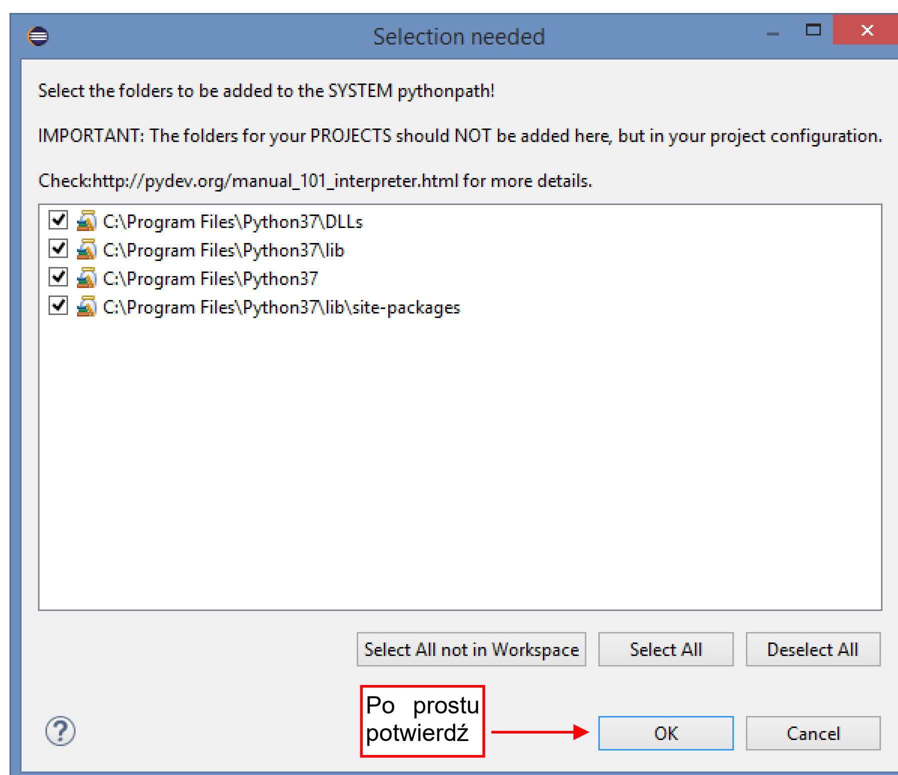
Copyright Witold Jaworski, 2011-2019.

Po chwili wyszukiwania PyDev wyświetli listę interpreterów Pythona (Rysunek 1.3.8):



Rysunek 1.3.6 Wybór interpretera Pythona

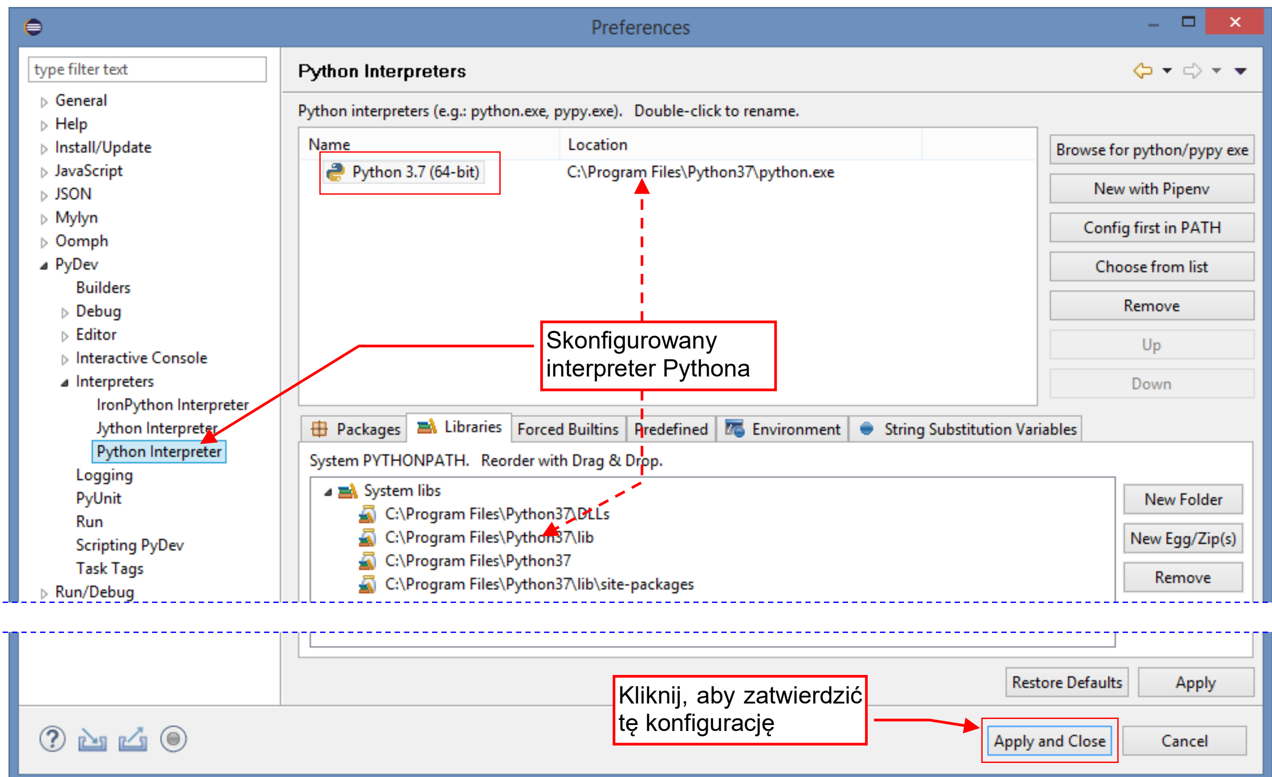
Wybierz z niej wariant 64-bitowy, który zainstalowałeś w poprzednim kroku (por. str. 8). W odpowiedzi PyDev przeszuka wskazane foldery i zaproponuje rozszerzenie systemowej listy **PYTHONPATH** o ścieżki związane tym interpreterem (Rysunek 1.3.7):



Rysunek 1.3.7 Potwierdzenie rozszerzenia listy *pythonpath* o foldery związane ze wskazanym interpreterem Pythona

W tym oknie nie trzeba niczego zmieniać – wystarczy potwierdzić (***OK***).

W rezultacie w oknie *Preferences* pojawi się skonfigurowany interpreter Pythona (Rysunek 1.3.8):



Rysunek 1.3.8 Skonfigurowany interpreter Pythona

Domyślnie PyDev nazwał ten interpreter po prostu „python". Dla większej przejrzystości zmieniłem tę nazwę na „Python 3.7 (64-bit)". (Ta nazwa jest używana wewnętrznie przez PyDev).

Zapisz te ustawienia wybierając polecenie ***Apply and Close***.

Pozostało jeszcze przygotować definicje uruchamiania / debugowania skryptu (tzw. ***Run Configurations***). W Eclipse tworzy się je oddzielnie dla każdego projektu. Będziesz mógł to zrobić dla projektu, w którym już umieściłeś plik głównego skryptu (por. sekcja 5.6, strona 134).

Copyright Witold Jaworski, 2011-2019.

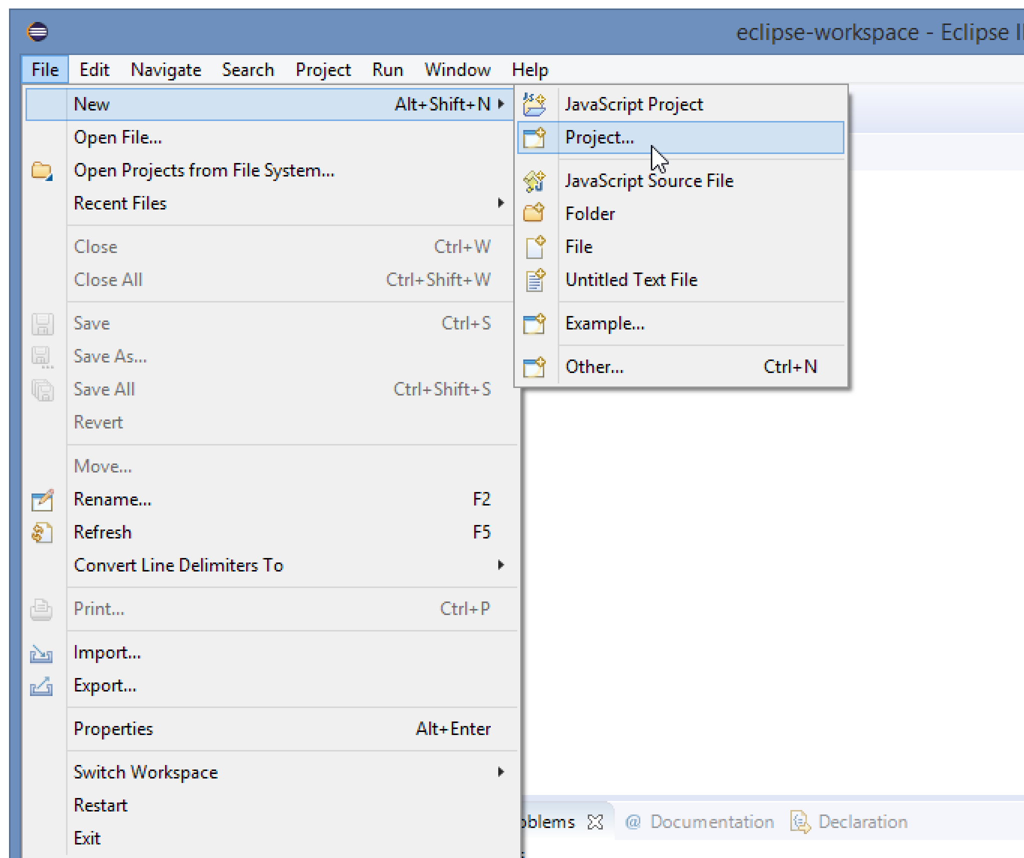
Rozdział 2. Pierwsze kroki w Eclipse

Tutaj zaczyna się nasz projekt. Będzie to adaptacja modyfikatora *Boolean*. Więcej o tym powiem w następnym rozdziale.

W tym rozdziale, poza nazwą, nasz projekt nie będzie miał z Benderem nic wspólnego. Na początek chcę pokazać podstawy pracy w środowisku Eclipse. Zrobię to na przykładzie najprostszego skryptu Pythona, który umieści w oknie konsoli napis „Hello”. Zakładam, że Czytelnik ma pewne pojęcie o Pythonie, oraz pracował już w jakimś IDE. To nie jest podręcznik żadnego z tych zagadnień. Moim celem jest raczej pokazanie, jak w Eclipse wykonuje się pewne podstawowe czynności, znane każdemu programiście.

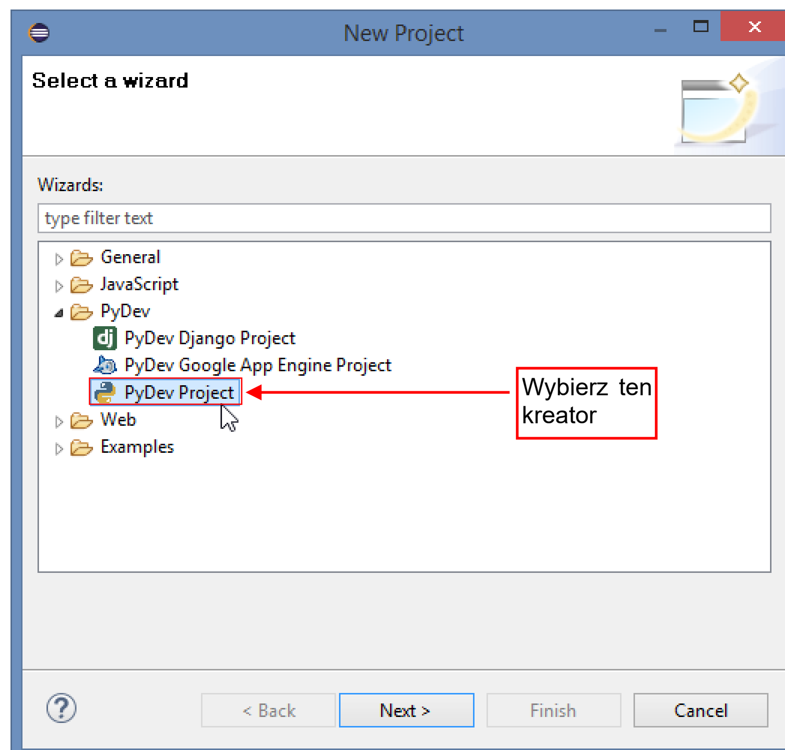
2.1 Rozpoczęcie projektu

Nowy projekt zaczynamy poleceniem ***File→New→Project…*** (Rysunek 2.1.1):



Rysunek 2.1.1 Polecenie tworzące nowy projekt

W oknie, które się otworzy, rozwiń folder *PyDev* i wybierz kreatora ***PyDev Project*** (Rysunek 2.1.2):

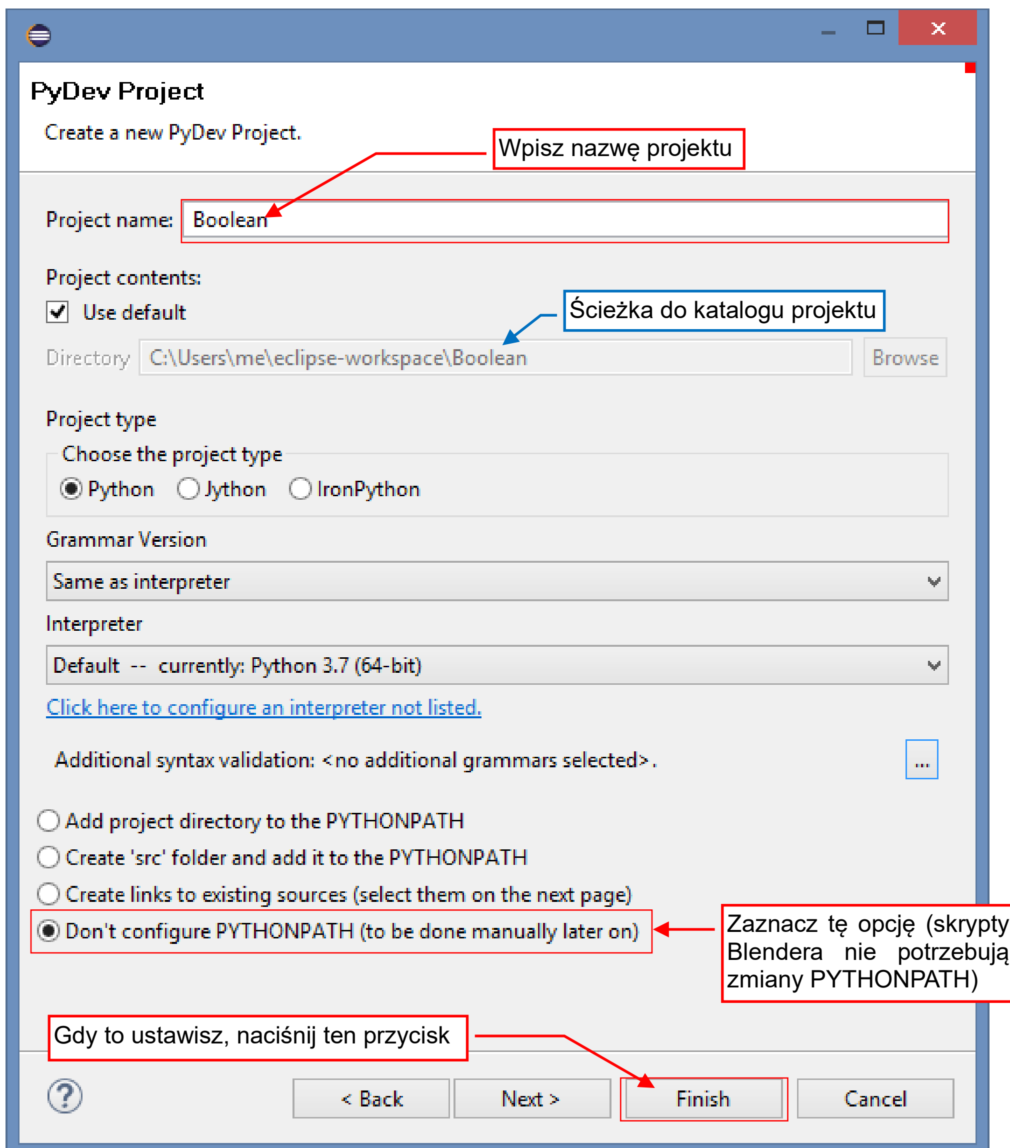


Rysunek 2.1.2 Wybór kreatora projektu

Następnie naciśnij przycisk ***Next***.

Copyright Witold Jaworski, 2011-2019.

W oknie kreatora projektu wpisz jego nazwę. Proponuję zacząć tu od razu projekt, który wykorzystamy do stworzenia skryptu dla Blendera. Stąd nadaję mu nazwę **Boolean** (Rysunek 2.1.3):

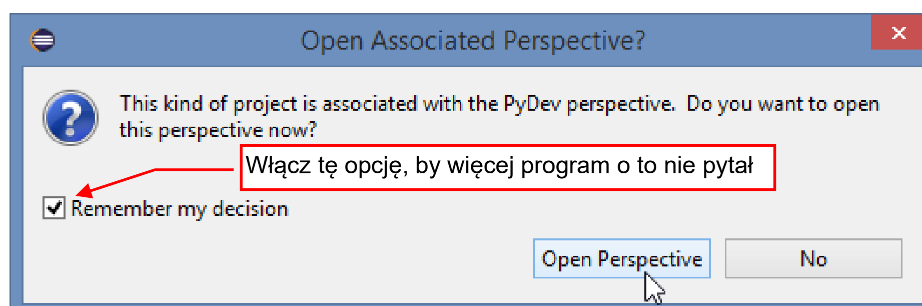


Rysunek 2.1.3 Ekran kreatora, ustalający szczegóły nowego projektu

Oprócz tego zaznacz opcję ***Don't configure PYTHONPATH…***. Resztę parametrów pozostaw bez zmian i naciśnij przycisk ***Finish***.

- PyDev wyszarzy przycisk *Finish*, jeżeli zapomniałeś skonfigurować interpretera Pythona (por. str. 15)

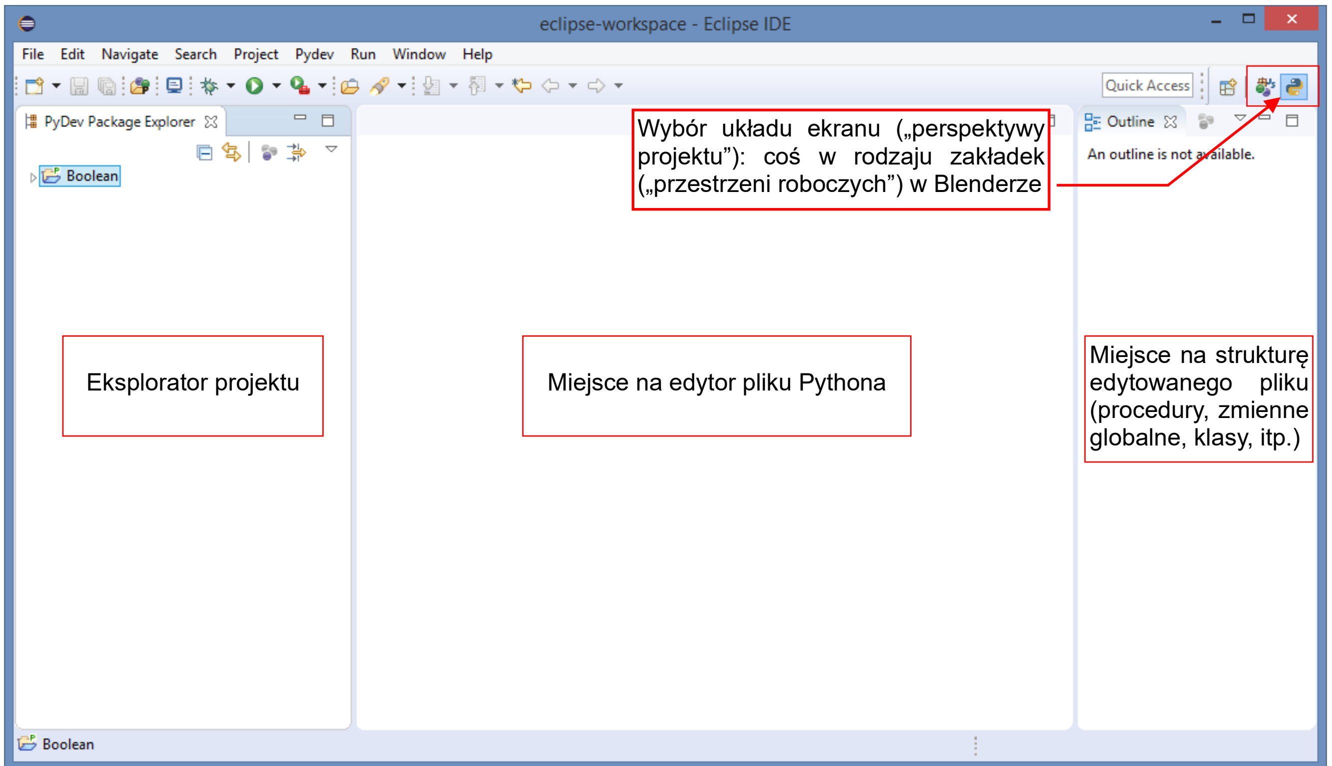
Pojawi się jeszcze komunikat (Rysunek 2.1.4):



Rysunek 2.1.4 Pytanie o domyślny układ okien (tzw. perspektywę) projektu

Potwierdź go, wybierając ***Open Perspective***.

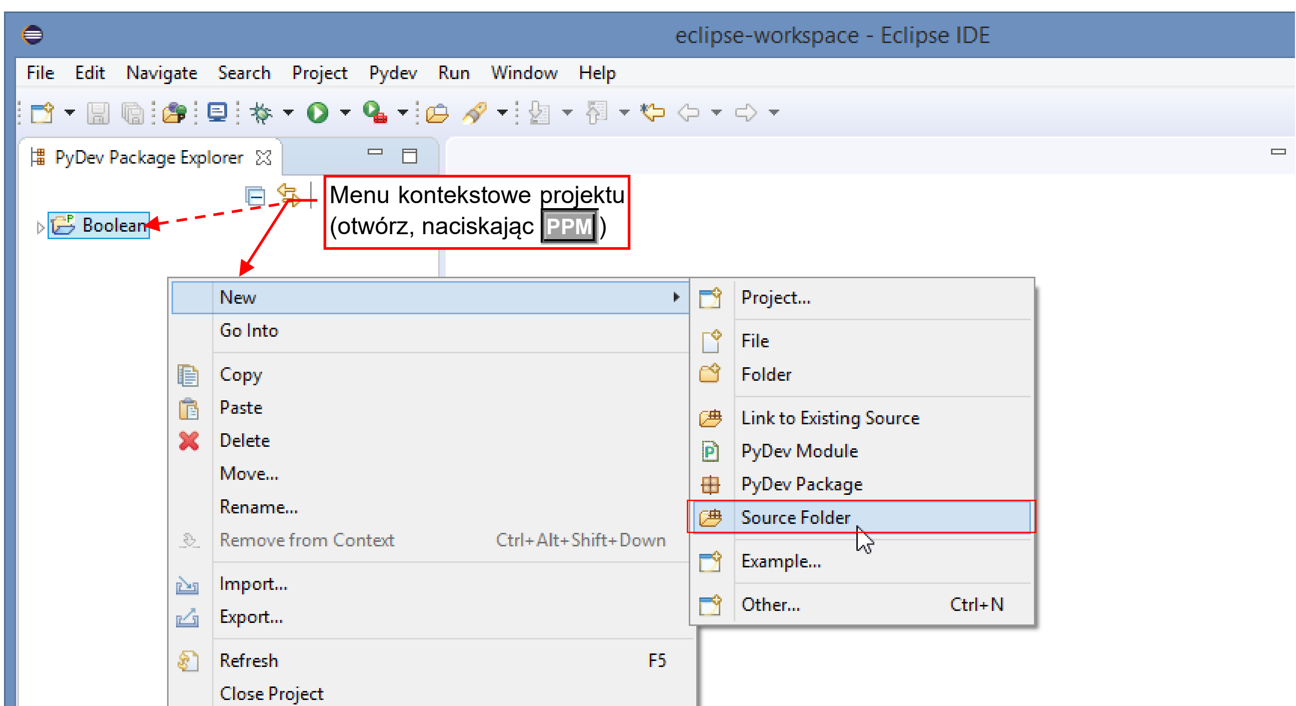
Kreator tworzy w Eclipse pusty projekt PyDev (Rysunek 2.1.5):



Rysunek 2.1.5 Nowy projekt PyDev w Eclipse

To, co widzisz, to domyślny układ zakładek z różnymi panelami projektu. Podobnie jak w Blenderze możesz mieć wiele zakładek z alternatywnymi układami ekranu, tak w Eclipse możesz mieć wiele alternatywnych „perspektyw” (*perspective*) projektu. Nowy projekt zawiera domyślną perspektywę *PyDev*, Przy okazji debugowania dodana zostanie jeszcze inna perspektywa — *Debug*.

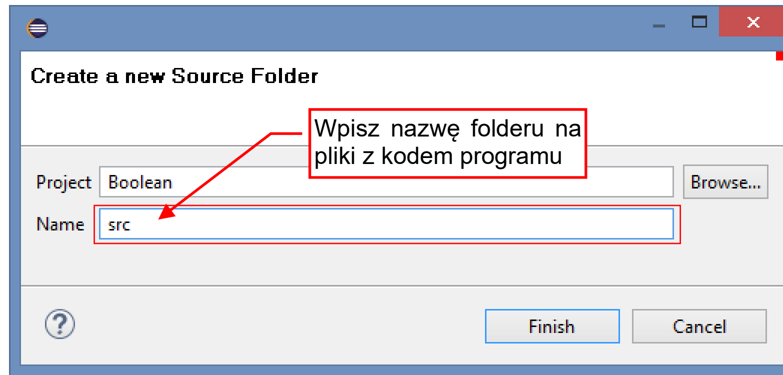
Zacznijmy od dodania do projektu folderu na skrypty: zaznacz folder projektu (*Boolean*), a potem z jego menu kontekstowego wybierz ***New→Source Folder*** (Rysunek 2.1.6):



Rysunek 2.1.6 Dodanie nowego folderu na pliki źródłowe

Copyright Witold Jaworski, 2011-2019.

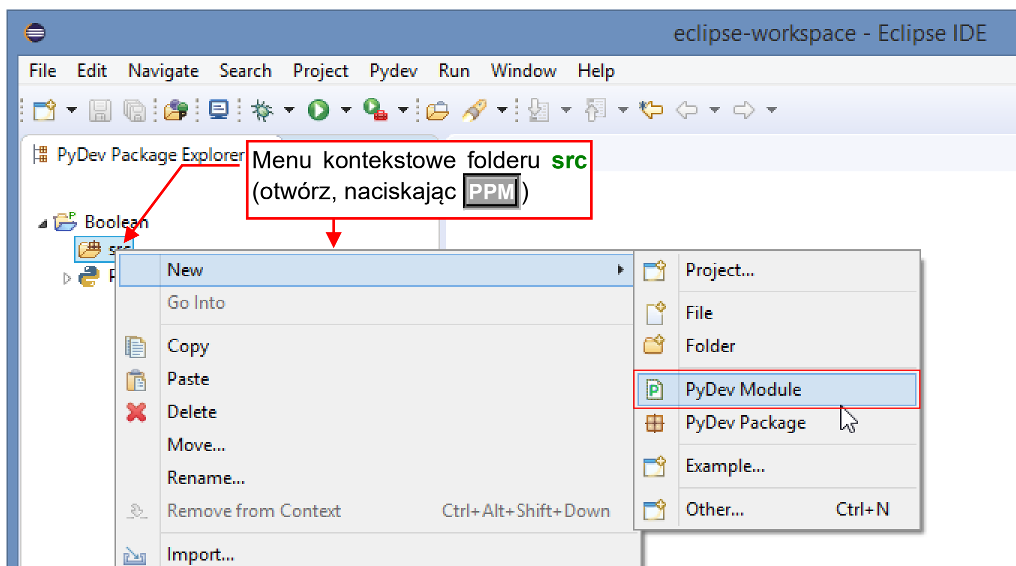
W oknie kreatora folderu wpiszmy mu nazwę — powiedzmy, **src** (Rysunek 2.1.7):



Rysunek 2.1.7 Okno kreatora folderu

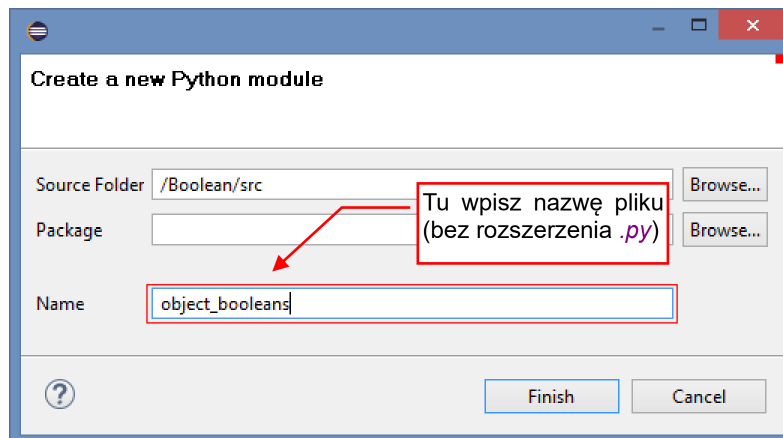
Gdy naciśniesz ***Finish***, w projekcie powstanie folder o tej nazwie.

Teraz stworzymy nowy, pusty plik skryptu. Rozwiń menu kontekstowe folderu **src** i wywołaj polecenie ***New→PyDev Module*** (Rysunek 2.1.8):



Rysunek 2.1.8 Dodanie nowego skryptu do folderu

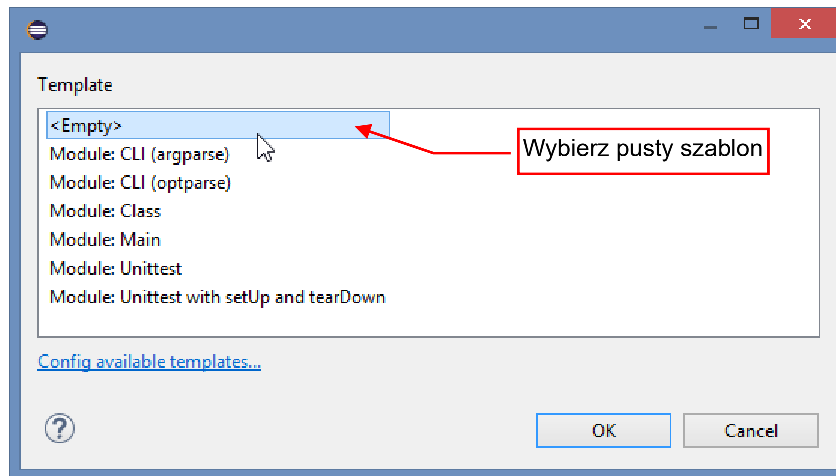
Otworzy to okno kreatora pliku Pythona. Nadaj plikowi nazwę w konwencji odpowiedniej dla planowanej wtyczki Blendera: **object_booleans** (Rysunek 2.1.9):



Rysunek 2.1.9 Okno kreatora skryptu Pythona

Następnie naciśnij przycisk ***Finish***.

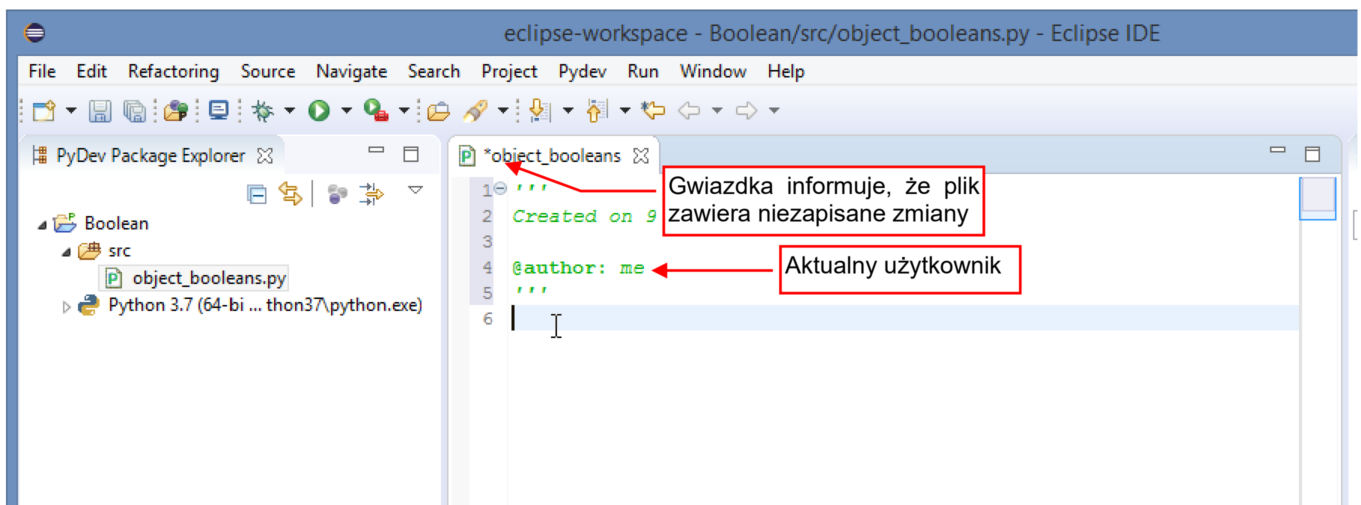
Spowoduje to wyświetlenie okna, z którego należy wybrać wzorzec pliku:



Rysunek 2.1.10 Okno kreatora skryptu Pythona (c.d.)

Wybierz tu szablon ***<Empty>*** i naciśnij ***OK***.

W projekcie pojawi się pierwszy plik Pythona. PyDev domyślnie wstawił w nagłówek komentarz z datą utworzenia i nazwą użytkownika (Rysunek 2.1.11):



Rysunek 2.1.11 Nowy, pusty skrypt

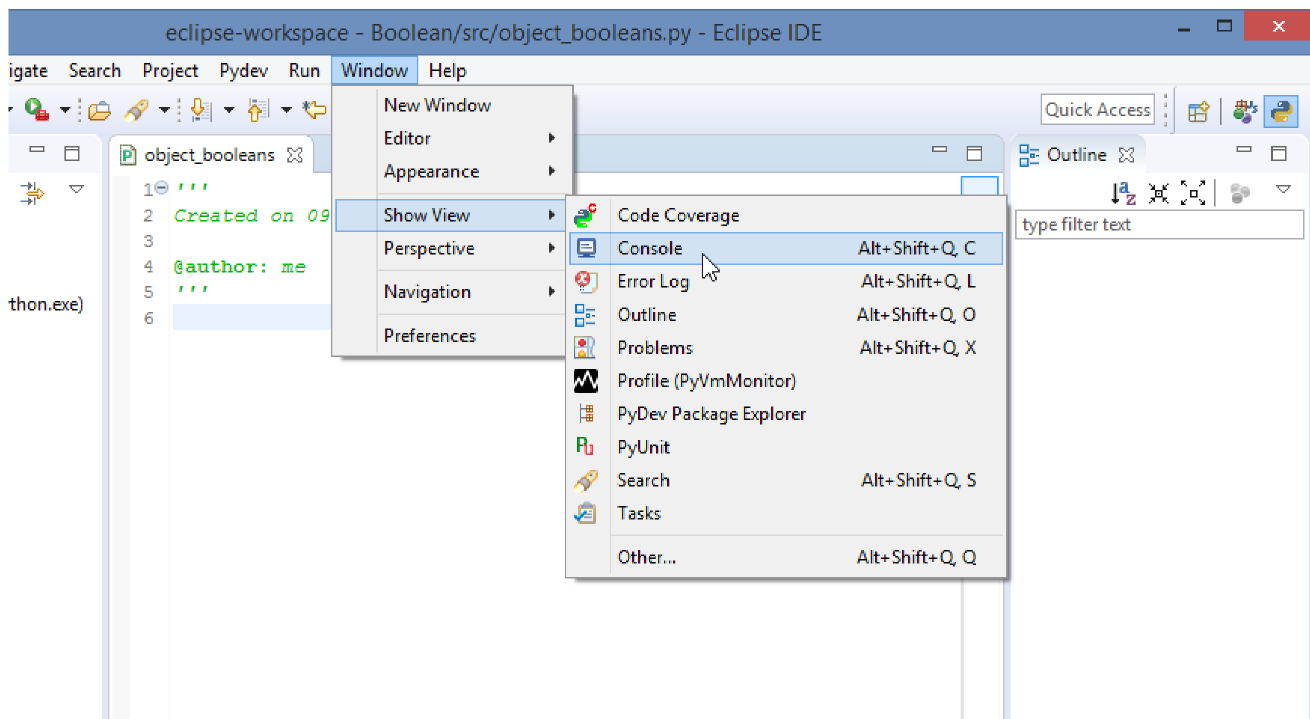
Podsumowanie

- W tej sekcji stworzyliśmy nowy projekt Pythona, posługując się kreatorem ***PyDev Project*** (str. 19);
- Przed dodaniem do projektu plików skryptów, musisz przygotować dla nich odpowiedni folder (*source folder* — str. 21);
- Przy tworzeniu nowego skryptu można skorzystać z kilku predefiniowanych wzorów (str. 23). My jednak nie użyliśmy żadnego z nich, wybierając wzorzec „pusty” (*<Empty>*);
- Nazwa projektu jest sprawą dowolną. Projekt w tym przykładzie nazwałem **Boolean** dlatego, że w dalszych rozdziałach książki posłuży nam do stworzenia w Blenderze 2.8 nowego polecenia: *Boolean operation*. Z tego samego powodu nadałem plikowi ze skryptem Pythona nazwę *object_booleans.py*.

Copyright Witold Jaworski, 2011-2019.

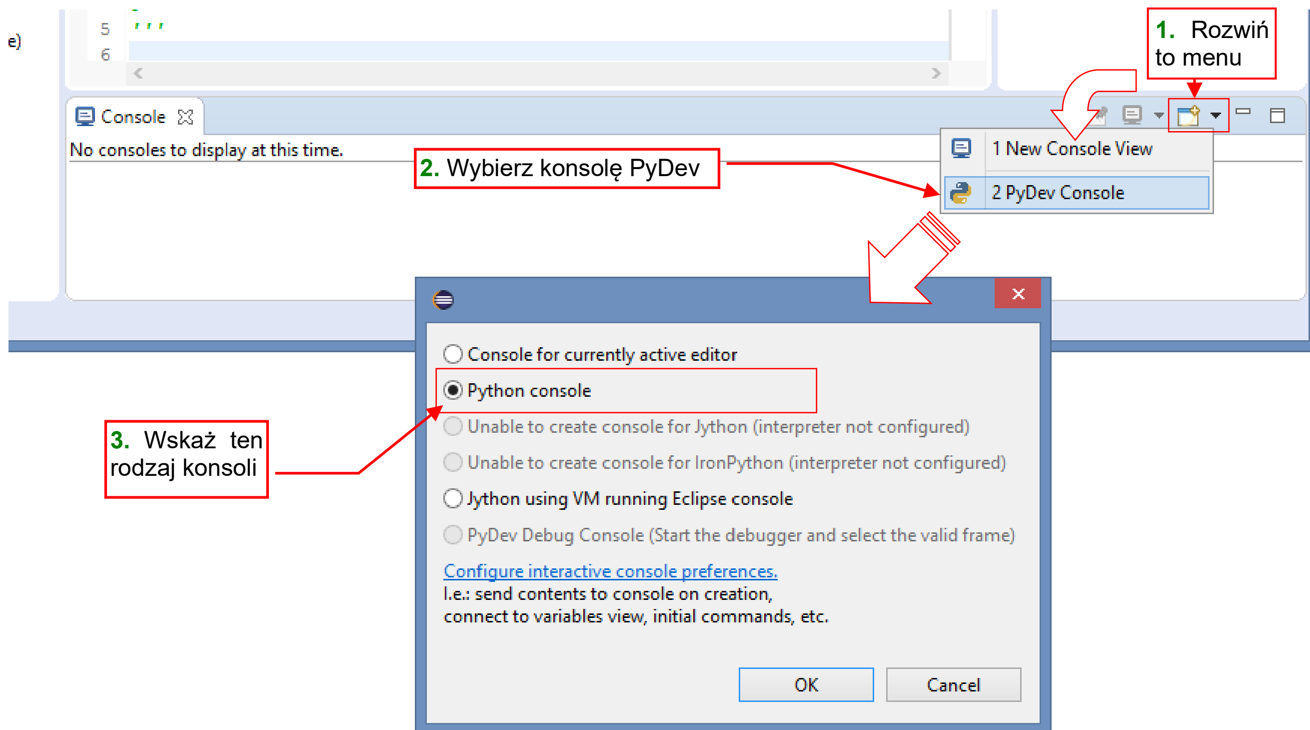
2.2 Uruchomienie najprostszego skryptu

Skrypt, który tu napiszemy, ma wyświetlić w konsoli Pythona napis „Hello!”. Więc musimy najpierw dodać panel z konsolą do naszego środowiska, bo domyślnie jej tu PyDev nie umieścił. Aby to zrobić, wywołaj: ***Window→Show View→Console*** (Rysunek 2.2.1):



Rysunek 2.2.1 Dodanie zakładki z konsolą

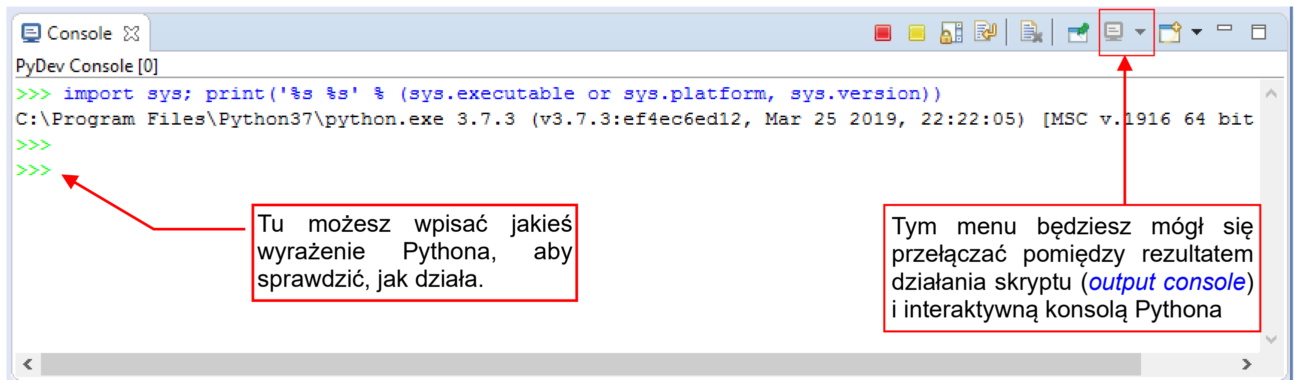
Domyślnie konsola pokazuje rezultat działania skryptu. W trakcie pisania kodu Pythona bardzo przydatna jest jeszcze tzw. „interaktywna konsola”. Dodajmy więc i ją (Rysunek 2.2.2):



Rysunek 2.2.2 Zmiana typu konsoli na interaktywną

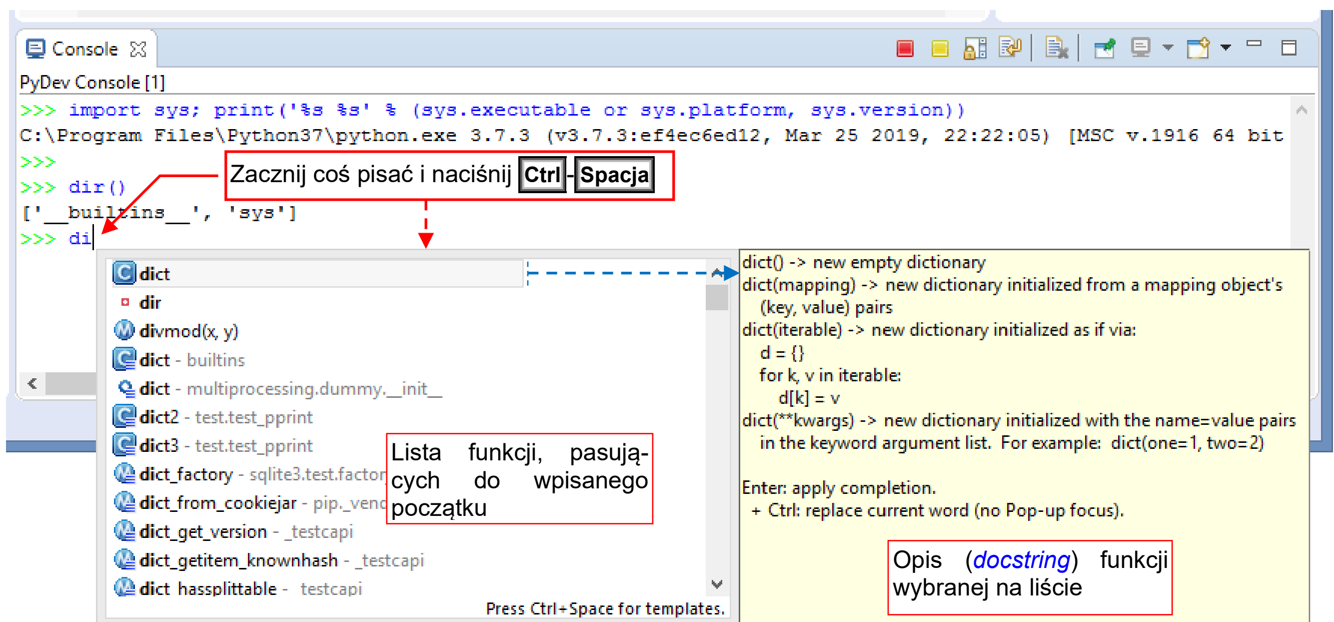
Po wybraniu z menu rozwijalnego zakładki ***PyDev Console***, wskaż w oknie dialogowym ***Python console***.

I oto masz w panelu uruchomiony interpreter Pythona, w którym można na bieżąco sprawdzać fragmenty kodu (Rysunek 2.2.3):



Rysunek 2.2.3 Interaktywna konsola Pythona

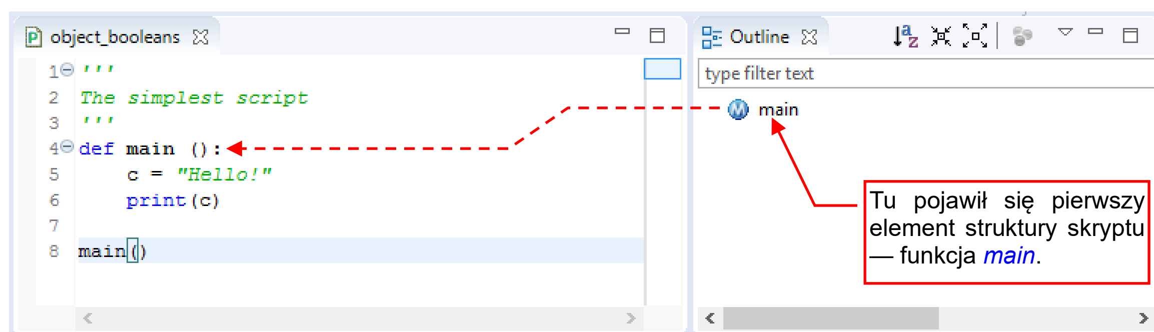
Jedną z bardzo przydatnych funkcji PyDev jest „dopowiadanie kodu” (*autocompletion*). To działa zarówno w oknie edytora skryptu, jak i interaktywnej konsoli (Rysunek 2.2.4):



Rysunek 2.2.4 Przykład dopełniania kodu

Okienko z podpowiedziami możesz przywołać naciskając kombinację **Ctrl**-**Spacja**, Pojawia się także samoczynnie, gdy w jakimś wyrażeniu pojawi się kropka (np. wpisz w konsoli „sys.”). Dzięki temu nie utrudnia specjalnie pisania „zwykłego” kodu.

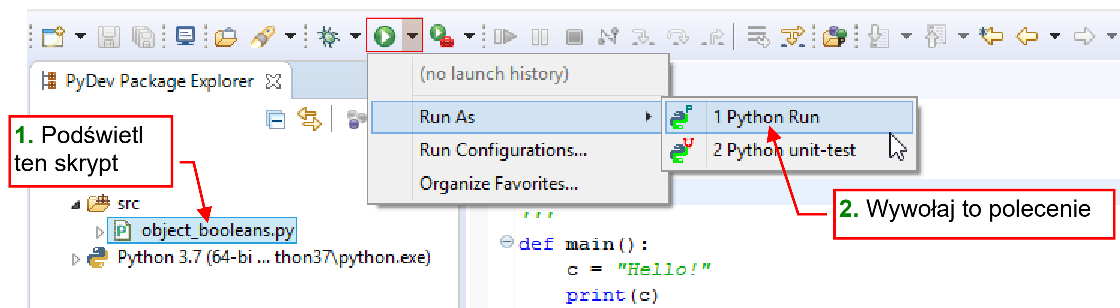
No, ale dosyć już gadania. Eclipse jest bardzo rozbudowanym środowiskiem i wszystkich jego funkcji i tak nie zdołam tu opisać. Lepiej przygotujmy nasz najprostszy skrypt (Rysunek 2.2.5):



Rysunek 2.2.5 Nasz skrypt — oczywiście w pierwszej wersji ☺

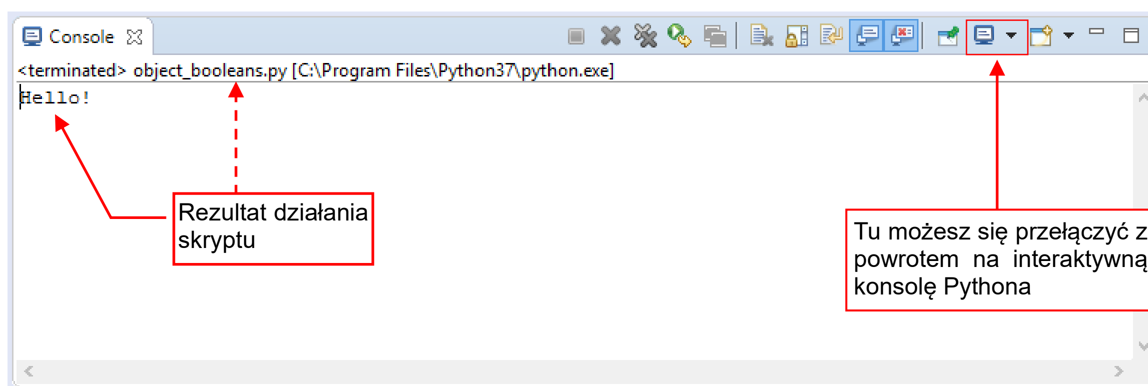
Copyright Witold Jaworski, 2011-2019.

Aby uruchomić ten skrypt po raz pierwszy, podświetl go w *Pydev Package Explorer* i z menu przycisku ***Run*** wywołaj polecenie ***Run As→Python run*** (Rysunek 2.2.6):



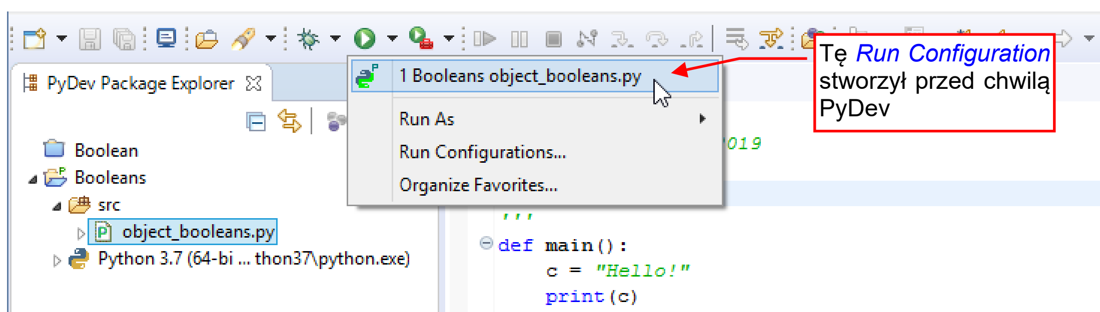
Rysunek 2.2.6 Uruchomienie skryptu (po raz pierwszy)

PyDev przełączy się na standardową konsolę, w której wyświetli rezultat działania wywołanego skryptu: napis „Hello!" (Rysunek 2.2.7):



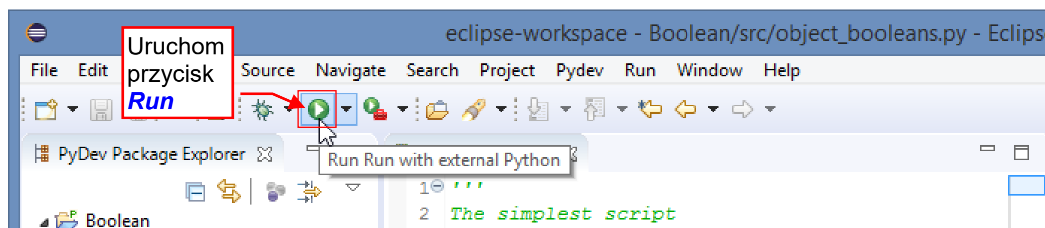
Rysunek 2.2.7 Rezultaty działania skryptu – tekst „Hello!"

Co więcej, PyDev stworzył w tym projekcie tak zwaną konfigurację uruchomienia (*Run Configuration*). Możesz ja zobaczyć na liście ostatnich wywołań (*favorites*) przycisków ***Debug*** i ***Run*** (Rysunek 2.2.8):



Rysunek 2.2.8 Domyślna *Run Configuration*

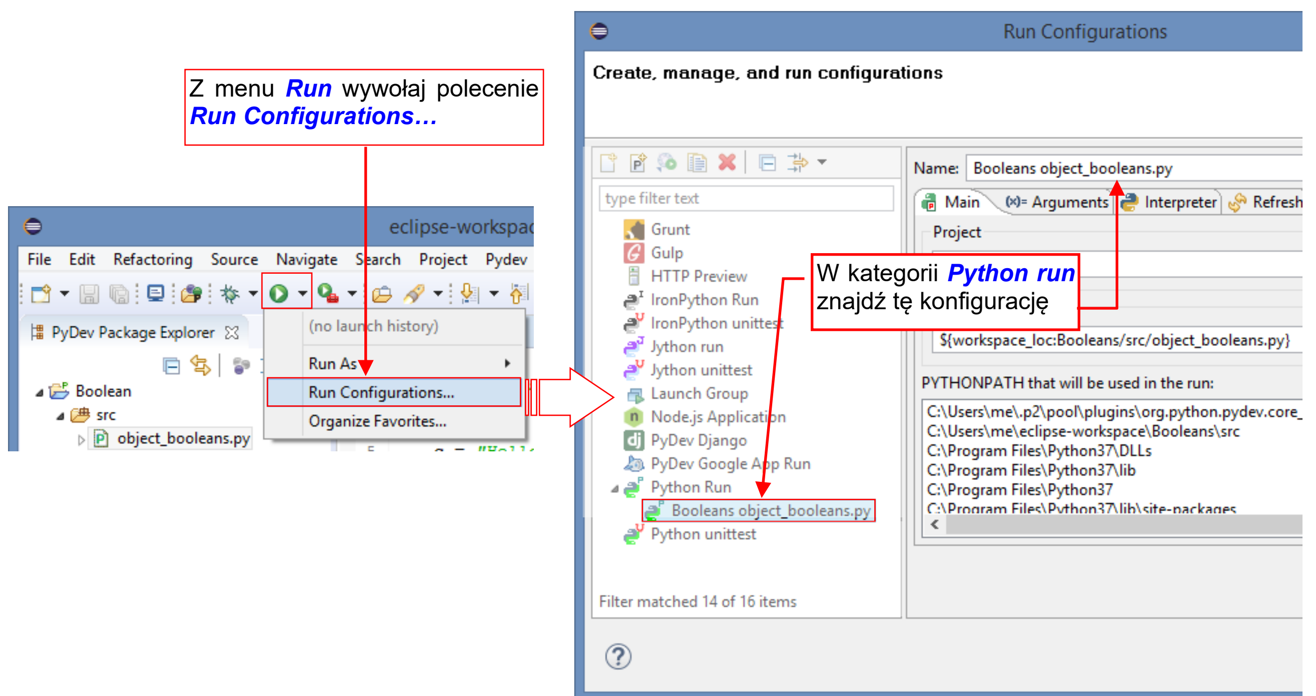
Zwróć uwagę, że konfiguracja *Boolean object_booleans.py* jest w tej chwili pierwszą na liście historii uruchomień. Aby powtórzyć ostatnie wywołanie, wystarczy kliknąć przycisk ***Run*** (Rysunek 2.2.9):



Rysunek 2.2.9 Uruchamianie skryptu (poprzez powtórzenie ostatniego wywołania)

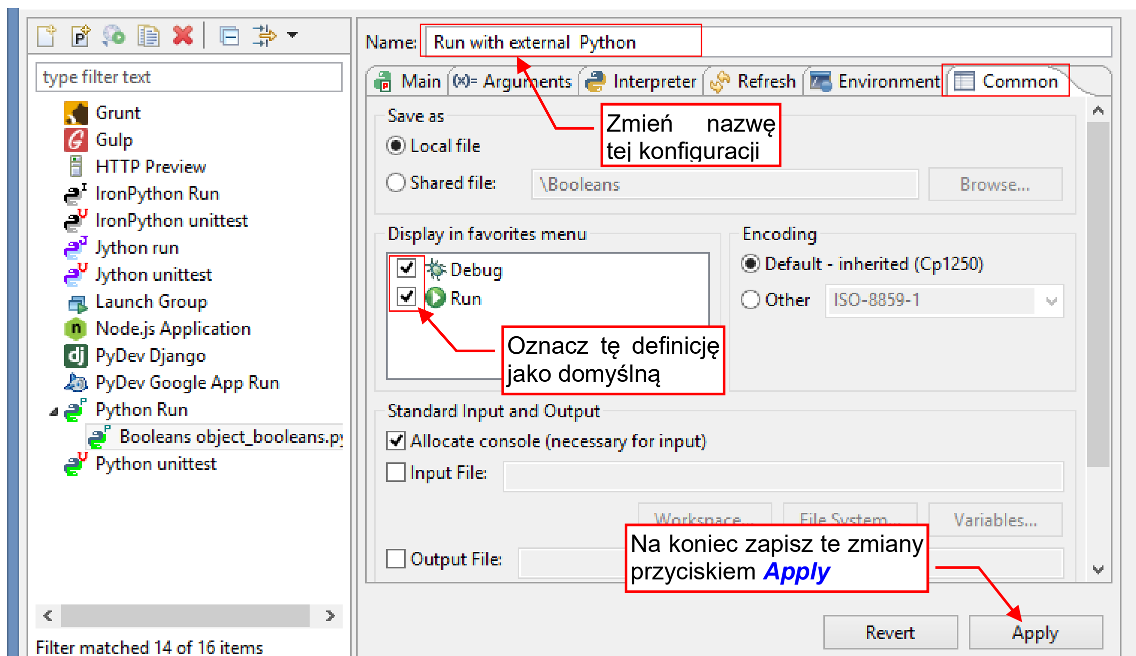
W odpowiedzi, PyDev powtórnie wywoła nasz skrypt.

Aby poprawić pewien aspekt tej domyślnej konfiguracji naszego skryptu, z menu ***Run*** wywołałem polecenie ***Run Configurations…*** i wyszukałem w tym oknie odpowiednią definicję (Rysunek 2.2.10):



Rysunek 2.2.10 Otwierane definicji uruchamiania naszego skryptu

Zmieniłem nazwę tej konfiguracji na **Run with external Python**. Dodatkowo, w zakładce ***Common*** włączyłem obydwie opcje ***Display in the favorities menu*** dla przycisków *Debug* i *Run* (Rysunek 2.2.11). (W ten sposób na wszelki wypadek „przypinam" je do listy ostatnich wywołań):



Rysunek 2.2.11 Modyfikacja definicji uruchamiania dla naszego skryptu

Konfiguracje uruchamiania możesz także zdefiniować od podstaw – tak jak to pokazuję na str. 133.

- Nazwy konfiguracji uruchamiania muszą być unikalne w całej przestrzeni roboczej (*workspace*) Eclipse, czyli dla wszystkich jej projektów. Zmieniłem tutaj nazwę konfiguracji uruchamiania tylko dla większej przejrzystości dalszego tekstu. Jednak w przyszłości lepiej pozostawiaj te nazwy w ich domyślnej postaci.

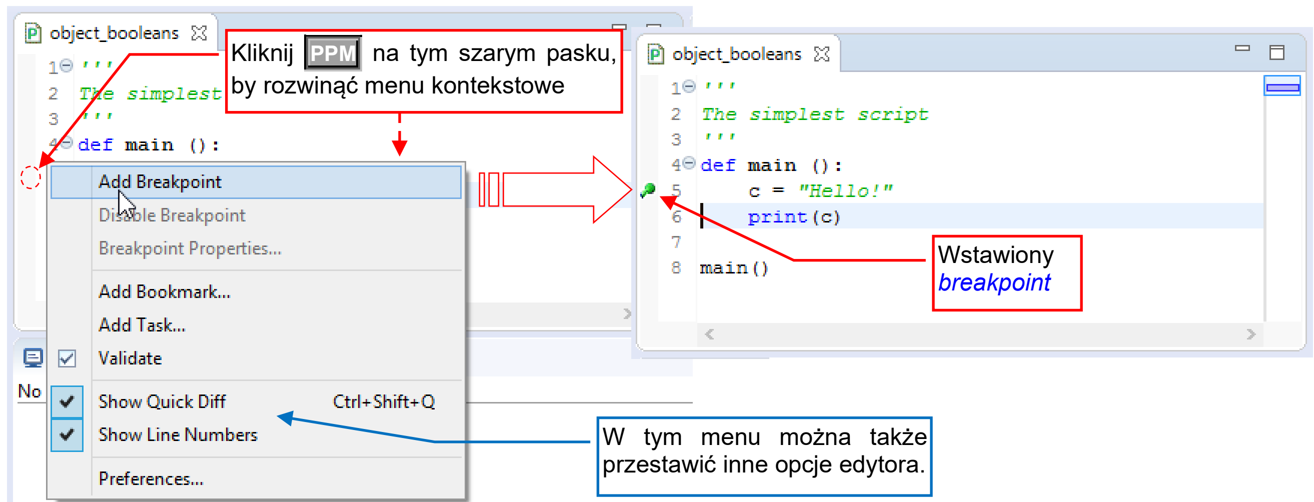
Copyright Witold Jaworski, 2011-2019.

Podsumowanie

- Dodaliśmy do projektu panel z konsolą Pythona (str. 24);
- Poznałeś działanie autokompletacji kodu oraz wyświetlanie opisu funkcji „w dymkach” (str. 25);
- Uruchomiliśmy najprostszy skrypt (plik *object_booleans.py*) i sprawdziliśmy jego rezultat w konsoli Pythona (str. 26);
- Podczas pierwszego uruchamiania skryptu *object_booleans.py* PyDev utworzył dla tego pliku odpowiednią konfigurację uruchamiania (*Run Configuration*). Możesz ją nieznacznie udoskonalić, poprzez „przypięcie” na stałe do listy ostatnich wywołań (*favorites*) menu *Run* (str. 27);

2.3 Debugowanie

Aby wstawić w jakąś linię kodu punkt przerwania (*breakpoint*), kliknij PPM w szary pasek przy lewej krawędzi edytora, przy linii, w której chcesz zatrzymać wykonywanie skryptu (Rysunek 2.3.1):

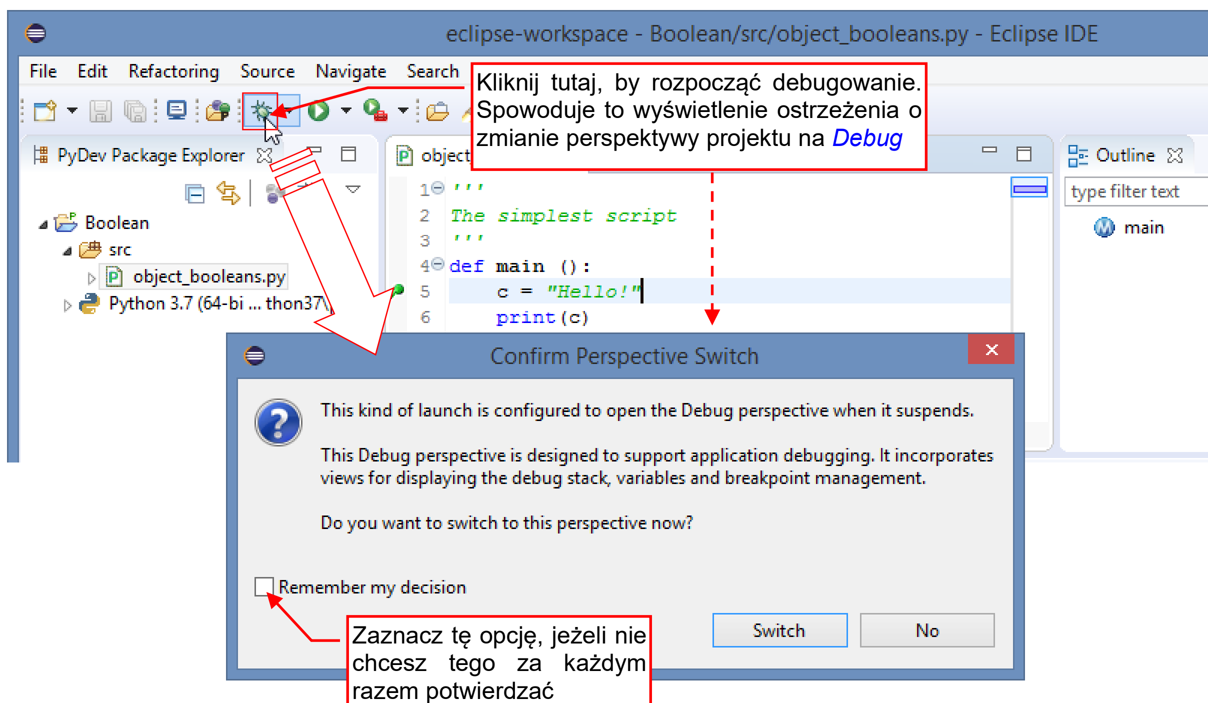


Rysunek 2.3.1 Zaznaczenie punktu przerwania (*breakpoint*)

Z menu kontekstowego, które w ten sposób rozwiniesz, wybierz polecenie ***Add Breakpoint***. Eclipse rysuje w tym miejscu zieloną kropkę z kreską (Rysunek 2.3.1). W podobny sposób, za pomocą menu kontekstowego, możesz usunąć lub wyłączyć niepotrzebny punkt przerwania.

- Możesz także dodać/usunąć *breakpoint* klikając dwukrotnie LPM w szary pasek z lewej strony linii

Aby uruchomić program w trybie śledzenia, naciśnij ikonę z pluskwą (☺ Rysunek 2.3.2):



Rysunek 2.3.2 Uruchomienie skryptu w trybie śledzenia

Eclipse wyświetli wówczas informację o zmianie aktualnej perspektywy projektu na *Debug*. (Za pierwszym razem doda ją do Twojego projektu). Ta nowa perspektyw zawiera kilka paneli przydatnych podczas debugowania kodu programu.

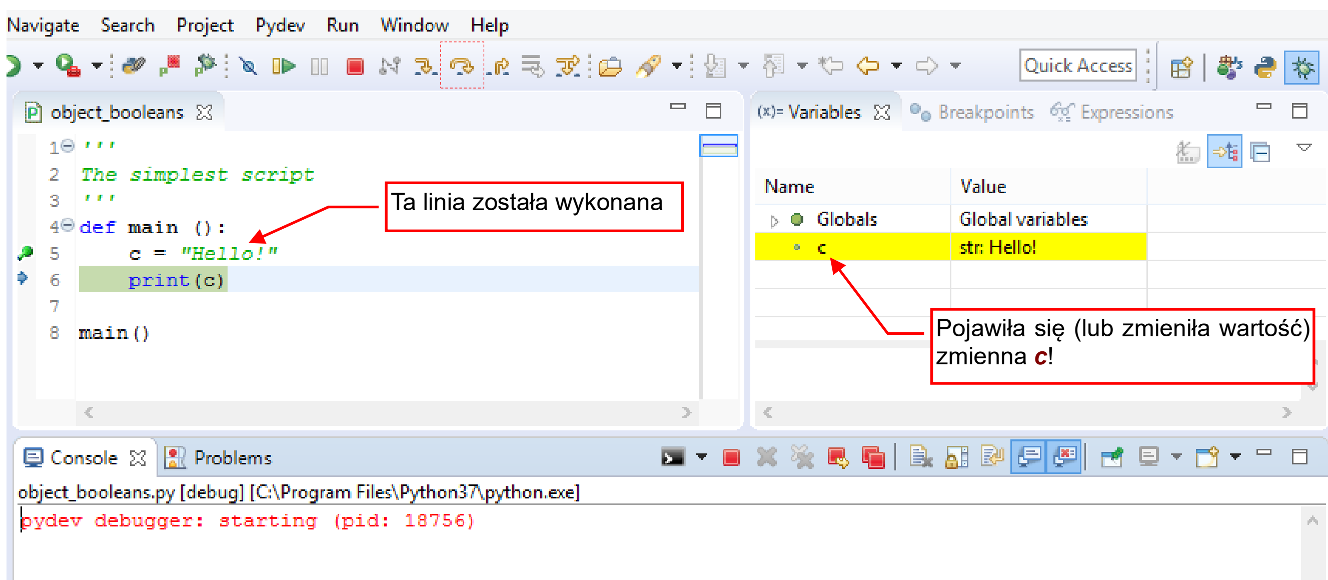
Copyright Witold Jaworski, 2011-2019.

Rysunek 2.3.3 przedstawia układ ekranu oraz kontrolki i podstawowe skróty klawiszowe dla debugowania programu. Zwróć uwagę, że wykonanie kodu zatrzymało się na naszym *breakpoincie*:

Resume F8
Terminate Ctrl+F2
Step Into F5
Step over F6
Step Return F7
Linia kodu, która ma być obecnie wykonana
Podgląd i zmiana zmiennych skryptu
Aktualny stos wywołań (funkcje)
Program zatrzymał się w ustalonym punkcie.
Konsola (standardowe wyjście) wykonywanego skryptu

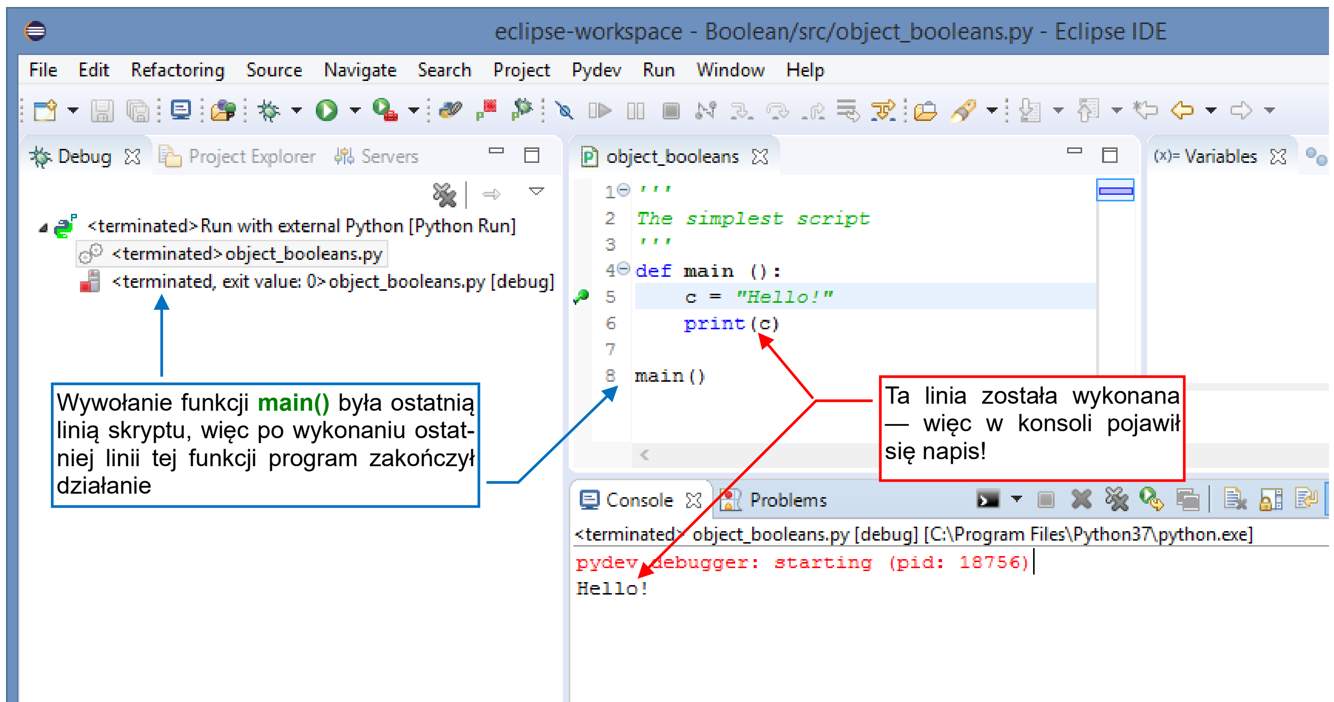
Rysunek 2.3.3 Układ ekranu w perspektywie *Debug*

Zielonkawa linia w oknie kodu źródłowego to linia bieżąca. Gdy teraz naciśniesz F6 (*Step over*) — nadasz wartość zmiennej ***c*** i przejdziesz do następnej linii (Rysunek 2.2.4):



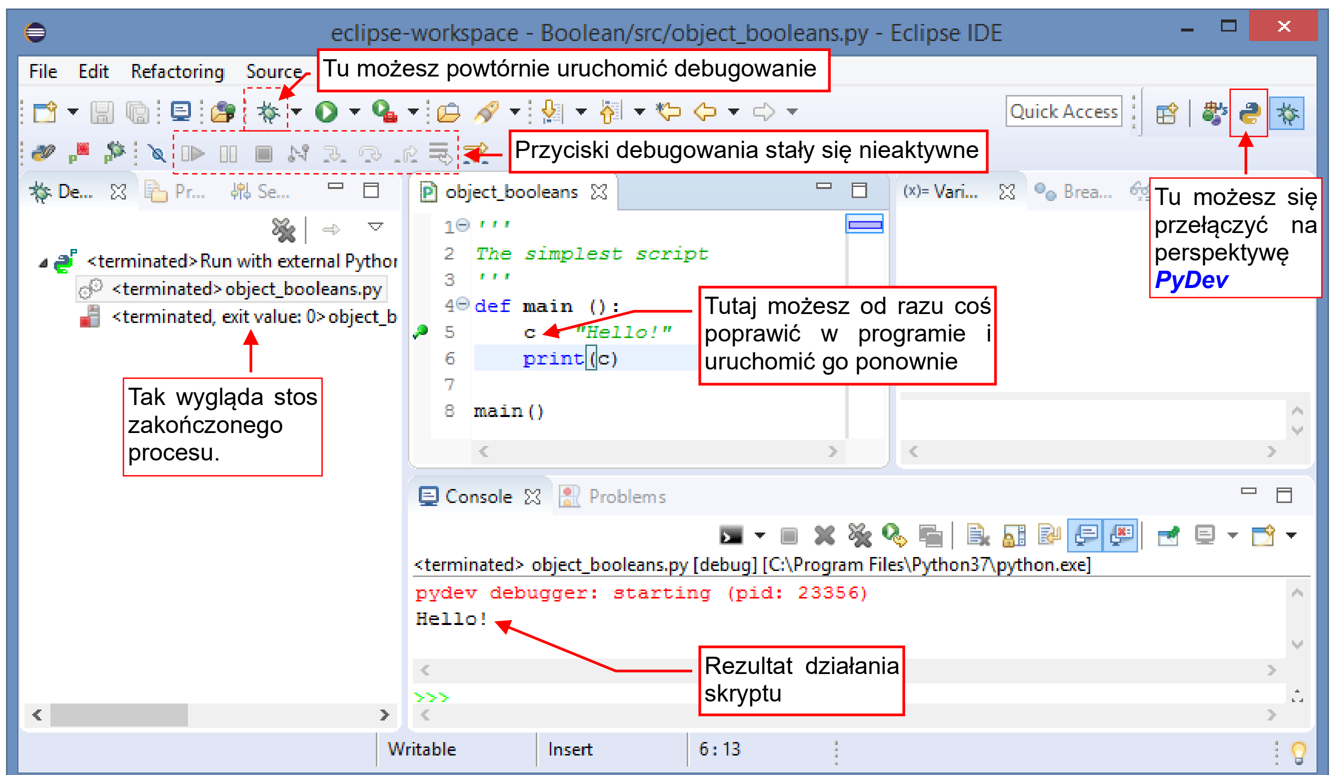
Rysunek 2.3.4 Sytuacja po naciśnięciu F6 (*Step over*)

Gdy znów naciśniesz **F6**, wykonasz kolejną linię, wyświetlającą zawartość tekstu ***c*** w konsoli. Jednocześnie opuścisz funkcję ***main()***, kończąc tym samym działanie programu (Rysunek 2.3.5):



Rysunek 2.3.5 Sytuacja po zakończeniu funkcji *main()*

Zwróć uwagę, że po zakończeniu śledzenia skryptu Eclipse wyszarzył przyciski debugowania (Rysunek 2.3.6):

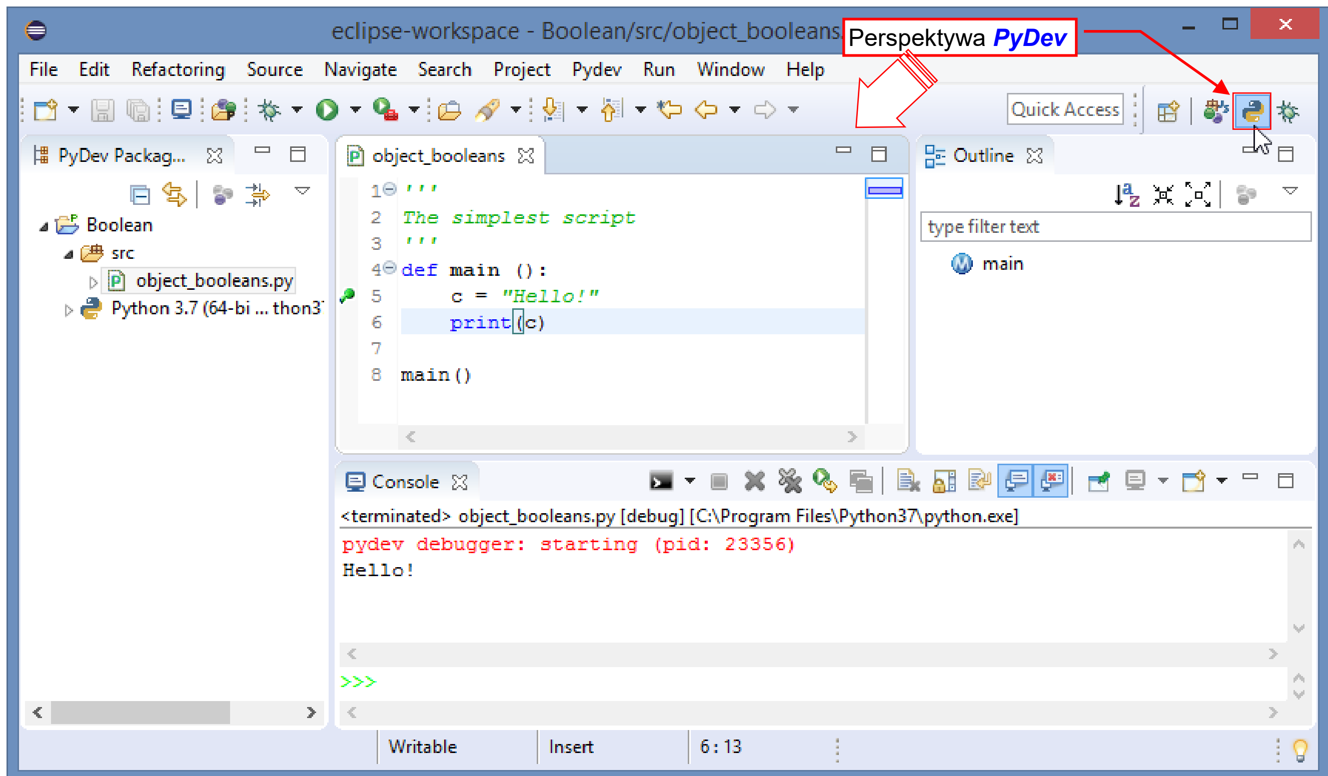


Rysunek 2.3.6 Perspektywa *Debug* po zakończeniu wykonywania skryptu

Drobne poprawki programu możesz wprowadzać od razu w perspektywie *Debug*, korzystając z dostępnego tu okna edytora.

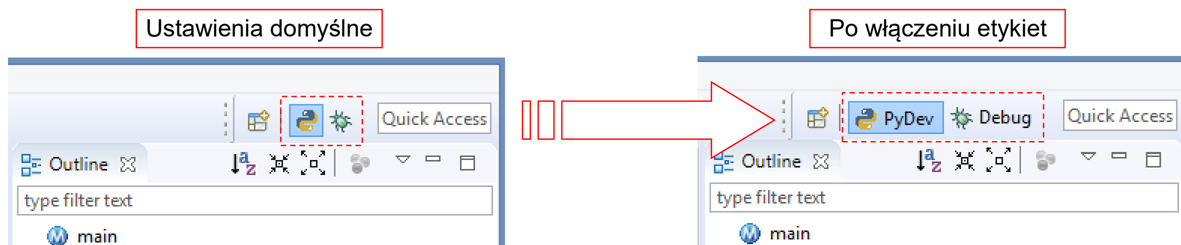
Copyright Witold Jaworski, 2011-2019.

Gdy jednak chcesz dopisać nowy fragment kodu — lepiej się przełącz się na perspektywę *PyDev* (Rysunek 2.3.7):



Rysunek 2.3.7 Powrót do perspektywy *PyDev* — czas na dalszą pracę nad kodem

Podczas pracy nad skryptem Pythona będziesz się ciągle przełączał pomiędzy perspektywami *Debug* i *PyDev*. Dlatego powiększyłem ich przyciski na pasku narzędzi, dodając do nich nazwy (Rysunek 2.3.8):



Rysunek 2.3.8 Powiększone przełączniki perspektyw

Zrobiłem to, zaznaczając w menu kontekstowym przycisku perspektywy (PPM) opcję ***Show Text***. (Szczegóły znajdziesz na str. 133).

Podsumowanie

- Zaznaczyliśmy w kodzie programu punkt przerwania (*breakpoint* — str. 29);
- Uruchomiliśmy skrypt w debugerze (str. 29). Przy okazji dodaliśmy do projektu drugą perspektywę — *Debug*;
- Poznałeś podstawowe funkcje debuggera: ***Step Into*** (F5), ***Step Over*** (F6), ***Resume*** (F8) (str. 30);
- Poznałeś dodatkowe panele debuggera — zmiennych (***Variables*** — str. 30) i stosu (***Stack*** — str. 31);
- Po zakończeniu skryptu można pozostać w perspektywie *Debug* aby wprowadzić do skryptu ewentualne poprawki i powtórnie uruchomić go w trybie śledzenia. Większe fragmenty kodu wygodniej jest wprowadzać w perspektywie *PyDev*;

Tworzenie aplikacji Blendera

To główna część książki. Opisuje proces tworzenia dodatku do Blendera. Najpierw powstaje zwykły skrypt — „linearny” ciąg poleceń, realizujący założoną operację (Rozdział 3). Potem otrzymuje „obudowę” wymaganą dla wtyczek Blendera (Rozdział 4). Rezultatem jest gotowy do użycia dodatek (*add-on*), implementujący nowe polecenie programu.

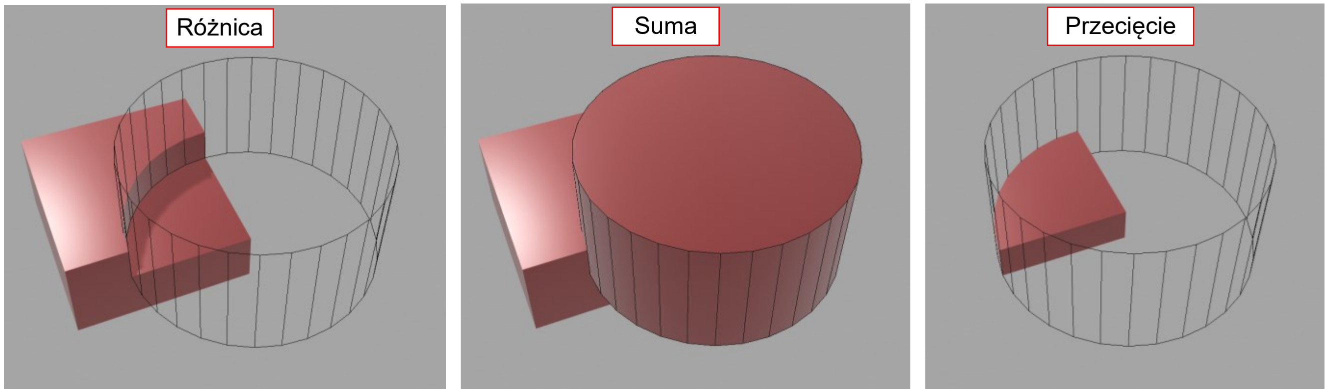
Copyright Witold Jaworski, 2011-2019.

Rozdział 3. Skrypt dla Blendera

W tym rozdziale przygotujemy skrypt wykonujący na obiektach (bryłach) podstawowe operacje Boole'a: sumę, różnicę, część wspólną. Posłużyłem się tym przykładem, by pokazać w praktyce wszystkie szczegóły środowiska programisty skryptów Blendera. Opisuję tu także metody rozwiązywania typowych problemów, jakie napotkasz podczas tworzenia kodu. Jednym z nich jest odnalezienie właściwego fragmentu API Blendera, który obsługuje potrzebne nam operacje! (API Blendera jest tak rozbudowane, że poza jego twórcami chyba mało kto ogarnia je w całości…).

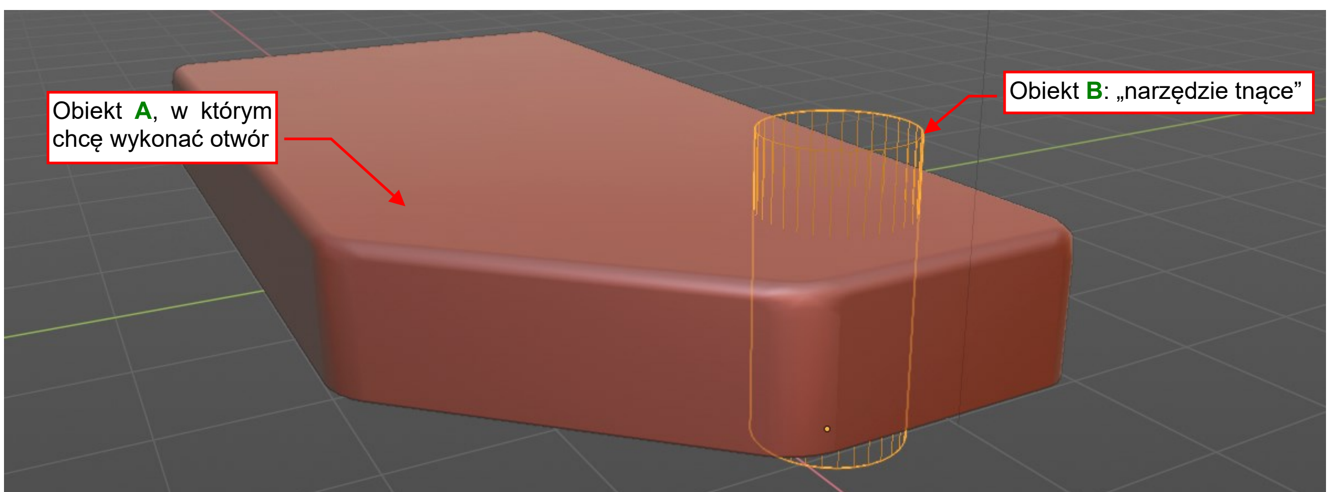
3.1 Sformułowanie problemu

Przy modelowaniu budynków czy części maszyn bardzo przydatne są trzy operacje wykonywane na bryłach: różnica, suma i przecięcie (Rysunek 3.1.1):



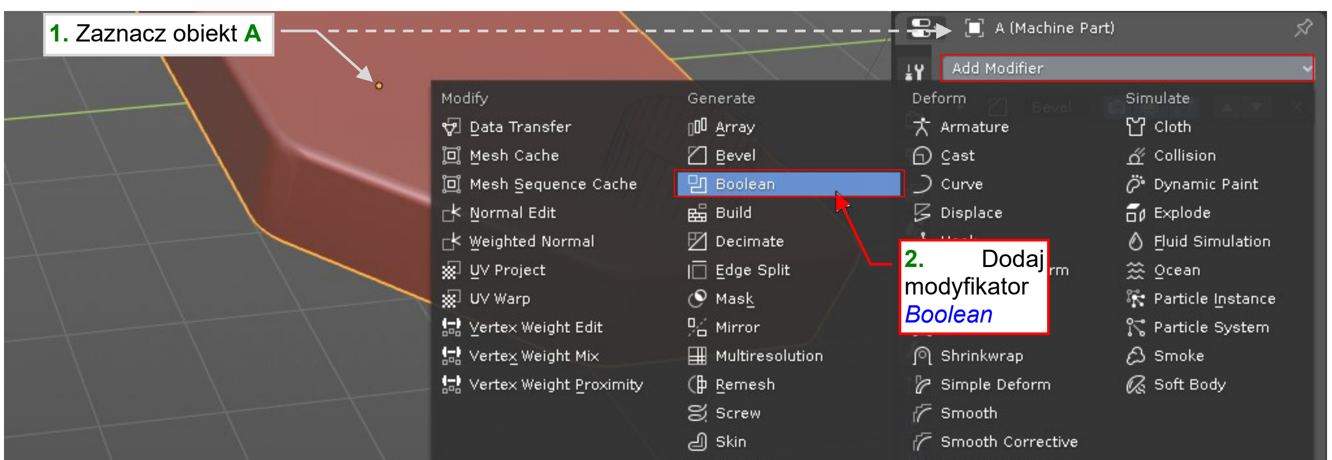
Rysunek 3.1.1 Operacje Boole'a na bryłach

W modelowaniu części maszyn zazwyczaj używam różnicy dwóch brył, aby szybko wykonać wgłębienie lub otwór. Jednak w Blenderze to „szybko" nie jest wcale takie szybkie, gdyż te operacje są implementowane za pomocą modyfikatora *Boolean*. Powiedzmy, że chcę wykonać w płytce **A** wgłębienie za pomocą pomocniczego obiektu-narzędzia **B** (Rysunek 3.1.2):



Rysunek 3.1.2 Stan początkowy: bryła i „narzędzie tnące"

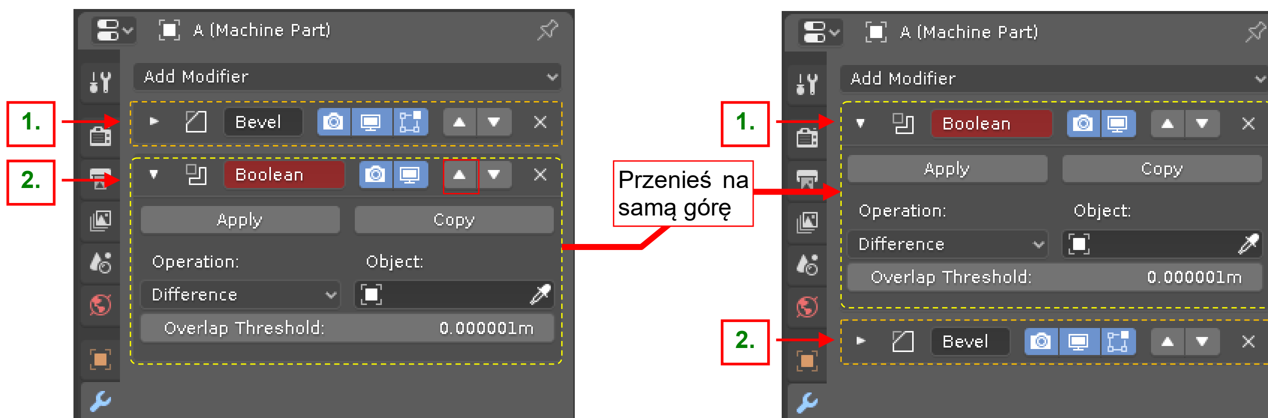
Zaczynam od zaznaczenia obiektu **A** i dodania w jego właściwościach modyfikatora ***Boolean***:



Rysunek 3.1.3 Dodanie modyfikatora *Boolean* do obiektu A

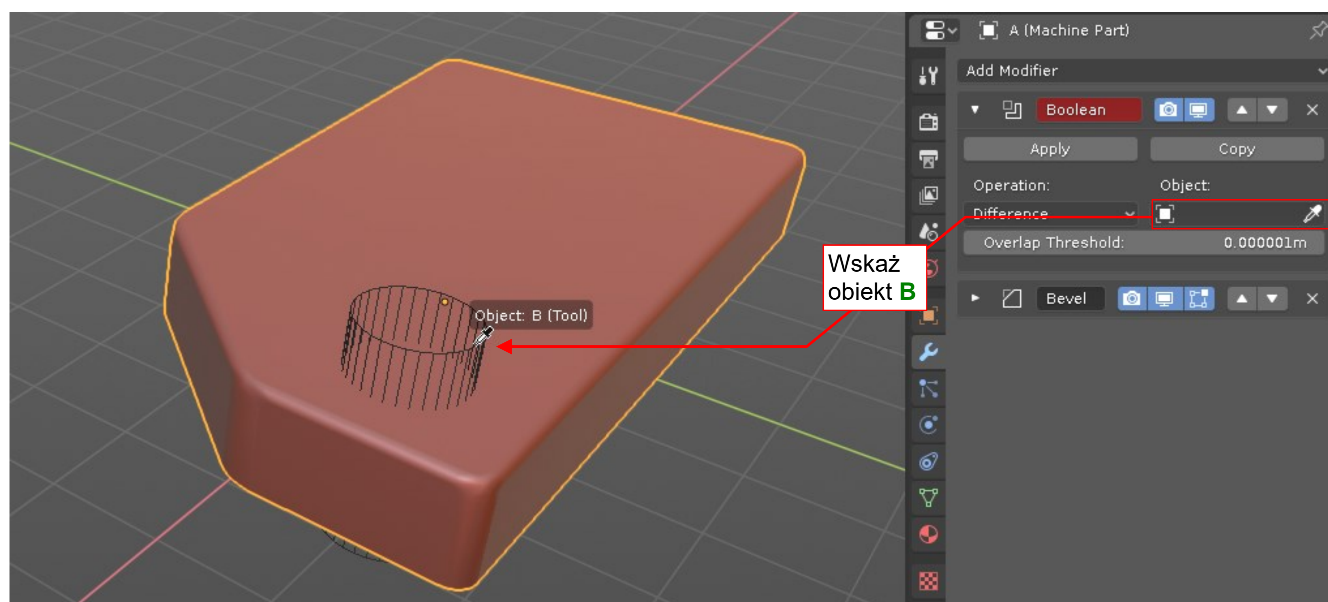
Copyright Witold Jaworski, 2011-2019.

W modelach części maszyn zazwyczaj wykorzystuję wiele różnych modyfikatorów, a każdy nowo dodany jest umieszczany na końcu ich listy (stosu). W kolejnym kroku muszę przesunąć modyfikator *Boolean* na początek listy. (Robię to, bo chcę wykonać otwór w oryginalnej siatce obiektu **A**, a nie np. po zaokrągleniu krawędzi przez modyfikator *Bevel* - por. Rysunek 3.1.4):



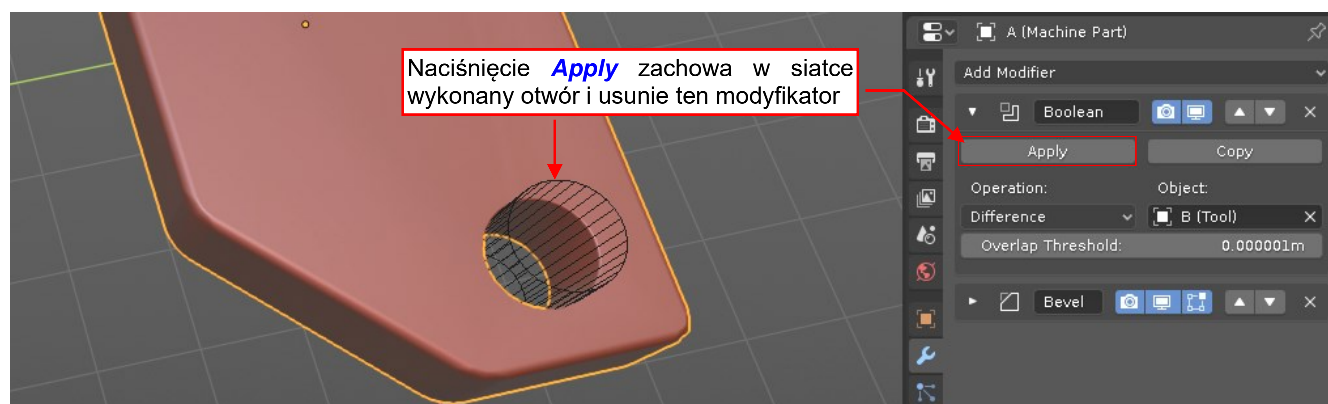
Rysunek 3.1.4 Uporządkowanie stosu modyfikatorów obiektu A

Modyfikator *Boolean* ma domyślnie ustawioną operację różnicy (***Difference***), więc w tym przypadku nie muszę jej zmieniać. W takim razie w kolejnym kroku przypisuje do tej operacji obiekt „tnący” (Rysunek 3.1.5):



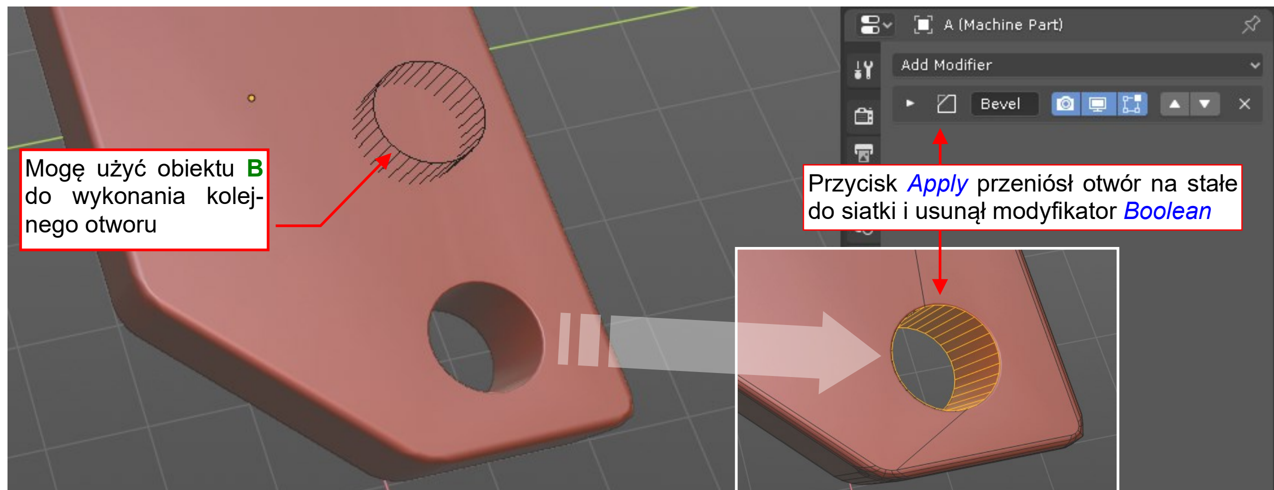
Rysunek 3.1.5 Przypisywanie do modyfikatora „narzędzia tnącego” (obiekt B)

Na koniec „utrwalam” (***Apply***) rezultat modyfikatora, bo nie chcę zachowywać obiektu **B** (Rysunek 3.1.6):



Rysunek 3.1.6 „Utrwalenie” rezultatu modyfikatora *Boolean*

W rezultacie znikł modyfikator *Boolean*, a otwór został przeniesiony do podstawowej siatki obiektu (Rysunek 3.1.7). Mogę teraz swobodnie przesunąć obiekt **B**, aby użyć go do wykonania kolejnego otworu:



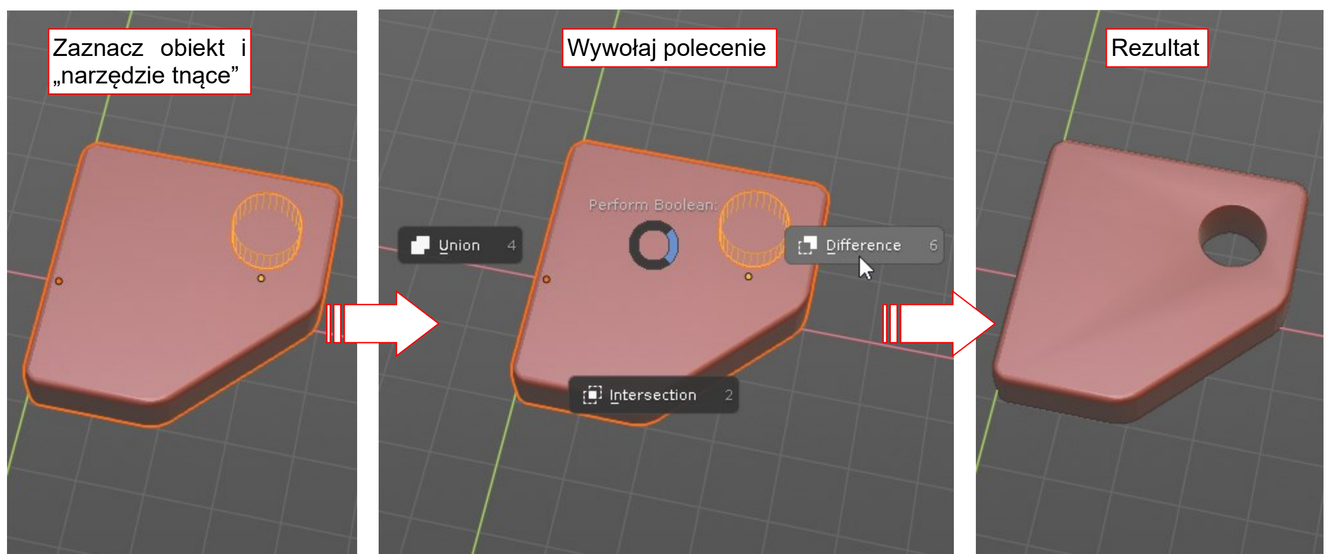
Rysunek 3.1.7 Finalny rezultat operacji

Modyfikator *Boolean* jest wspaniałą pomocą, gdy potrzebujesz „przenośnej dziury": otworu, którego położenie, rozmiar lub kształt może się zmienić w trakcie projektu. Pozwala także budować złożone kształty, zachowując jednocześnie proste podstawowe siatki obiektów. Jednak ma także kilka wad, wynikających z jego „dynamicznej" natury:

- Wymaga zachowania w jakiejś ukrytej kolekcji obiektu „tnącego". O ile nie jest to specjalny problem w przypadku kilku takich „narzędzi", to zaczyna być problemem w złożonych projektach, gdy w każdej części masz do wykonania kilka wgłębień i otworów. Wówczas musiałbyś zacząć zarządzać setkami takich obiektów;
- Nie możesz zmieniać właściwości nowo utworzonych krawędzi. Np. nie możesz przypisać im wagi fazowania (*Bevel Weight*), czy też zaznaczyć jako „szwów" (*Seam*), przydatnych w rozwijaniu na powierzchni UV;

Dlatego najczęściej od razu „utrwalam" w siatce rezultat modyfikatora *Boolean*, naciskając przycisk ***Apply***.

Do obsługi takiego stylu pracy potrzebuję trzech prostych poleceń (Rysunek 3.1.8):



Rysunek 3.1.8 Polecenia *Boolean*, których potrzebuję

Zaczynam od wybrania obiektu-narzędzia i obiektu do zmiany. Następnie wywołuję (np. jakimś skrótem klawiatury) menu *Boolean*, z którego wybieram polecenie (W tym przypadku: *Difference*). W rezultacie w obiekcie powstaje otwór. W opcjach polecenia (nie pokazanych na ilustracji) będę mógł ewentualnie zaznaczyć, czy re-

Copyright Witold Jaworski, 2011-2019.

zultat ma być zachowany jako modyfikator. Kolejne wywołanie polecenia ma używać domyślnie takich samych opcji, jakie zastosowałem w wywołaniu poprzednim.

W tym rozdziale napiszemy skrypt Blendera, który wykorzysta modyfikator *Boolean* do implementacji takiego polecenia. W następnym rozdziale przekształcimy ten skrypt w profesjonalny *add-on* Blendera, wzbogacając go m.in. o wygodne menu kontekstowe (*pie menu*) - takie, jakie pokazuje Rysunek 3.1.8.

Podsumowanie

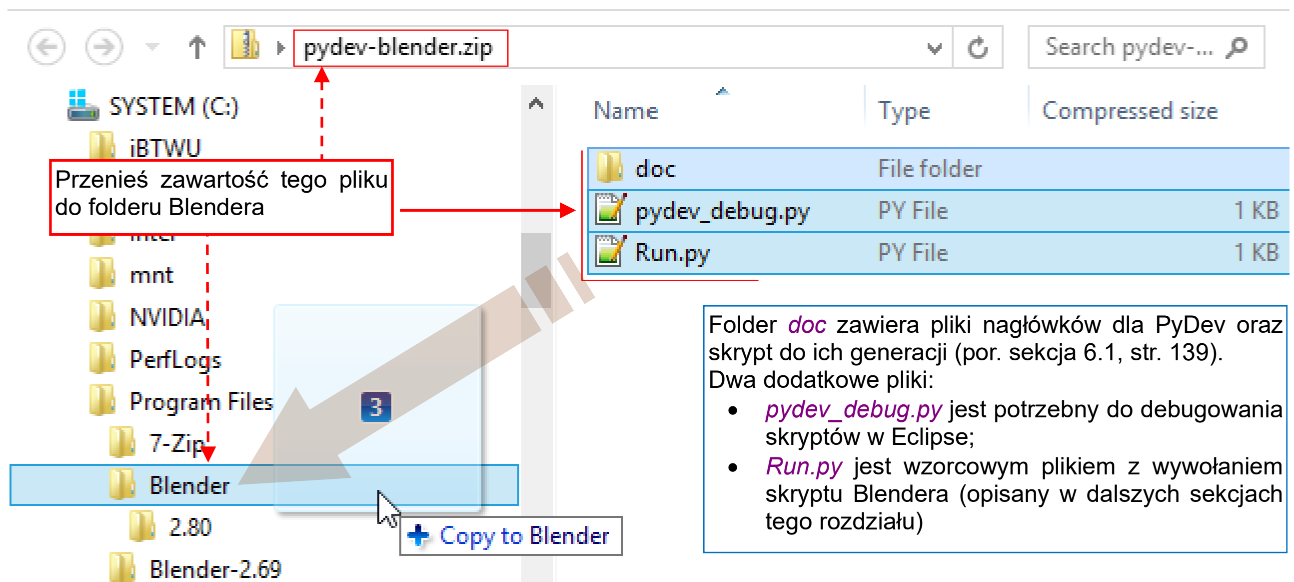
- Operacje na bryłach (różnica, suma, przecięcie) są często używane w modelach mechanicznych lub architektonicznych;
- W Blenderze 2.8 brakuje „statycznych" wersji poleceń Boole'a do operacji na bryłach[1]. Istnieje tylko modyfikator *Boolean* (str. 35, 36). Ten modyfikator łączy bryły w sposób dynamiczny;
- Przy modelowaniu realnych mechanizmów o większej liczbie części, „dynamiczne" powiązania tworzone przez modyfikatory *Boolean* i związana z nimi konieczność przechowywania wszystkich obiektów-narzędzi staje się bardziej kłopotem niż zaletą;
- Efekt odpowiadający „statycznemu" poleceniu *Boolean* można w Blenderze 2.8 uzyskać poprzez „utrwalenie" odpowiedniego modyfikatora (str. 36, 37). Aby nie powtarzać tych operacji ręcznie, stworzymy skrypt, który wykona je wszystkie za jednym razem. W ten sposób uzupełnimy program o funkcjonalność, której nam brakuje;

[1] Wśród dodatków (add-ons) dostarczanych wraz z Blenderem można (w kategorii *Object*) zaleźć wtyczkę o nazwie *Bool Tools*. Ten skrypt Pythona implementuje polecenia: *Union*, *Difference*, *Intersection* i *Slice*. (*Slice* to „odcięcie kawałka obiektu" – kombinacja rezultatów *Intersection* i *Difference*). Jednak szczegóły działania *Bool Tools* różnią od polecenia, które postuluje Rysunek 3.1.8 i jego opis. Na przykład – nie ma w nim możliwości zachowania obiektu-narzędzia, a jego menu wyglądają nieco staromodnie. *Bool Tools* nie zachowuje także żadnych modyfikatorów w obiekcie, który ulega zmianie. (Stosuje modyfikator *Boolean* do rezultatu wszystkich wcześniejszych modyfikatorów obiektu). W zamyśle postulowane przeze mnie narzędzie jest na tyle proste, że zdecydowałem się je napisać od podstaw, zamiast przerabiać istniejącą wtyczkę. Poza tym będzie to dobry przykład realnego dodatku do Blendera, w sam raz dla takiej książki jak ta.

3.2 Dostosowanie Eclipse do API Blendera

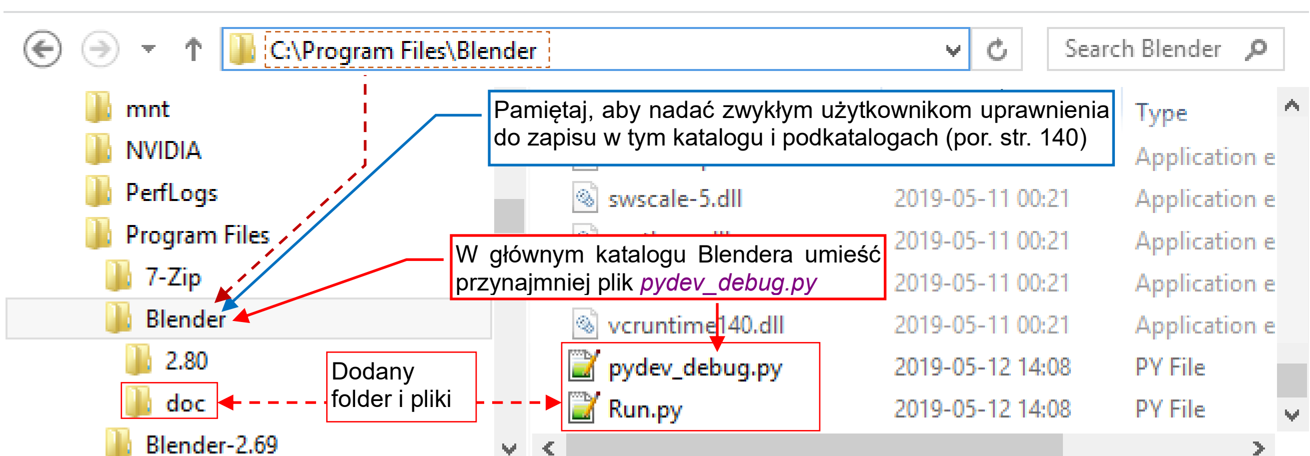
Aby wygodnie pisać skrypty Blendera, musimy sprawić by PyDev podpowiadał metody i pola obiektów Blender API tak samo, jak to robi dla standardowych modułów Pythona. Wymaga to dostarczenia PyDev uproszczonych plików Pythona, zawierających tylko deklaracje klas, metod i właściwości. (Pomysł jest podobny do „plików nagłówków" (*header files*), stosowanych w C/C++). Aby odróżnić takie pliki od zwykłych plików Pythona, PyDev wymaga, by nadać im rozszerzenie **.pypredef* (od *predefinition*).

Używając przerobionego skryptu Campbella Bartona, którym generowana była dokumentacja API w Blenderze 2.5, udało mi się stworzyć odpowiednie pliki **.pypredef* dla niemal całego API Blendera. Dołączam je do tej książki. Pobierz z mojego portalu plik http://samoloty3d.pl/downloads/pydev2/pydev-blender.zip. Rozpakuj go do jakiegoś folderu - np. tego, w którym umieściłeś Blendera (Rysunek 3.2.1):



Rysunek 3.2.1 Rozpakowanie materiałów pomocniczych (do folderu Blendera)

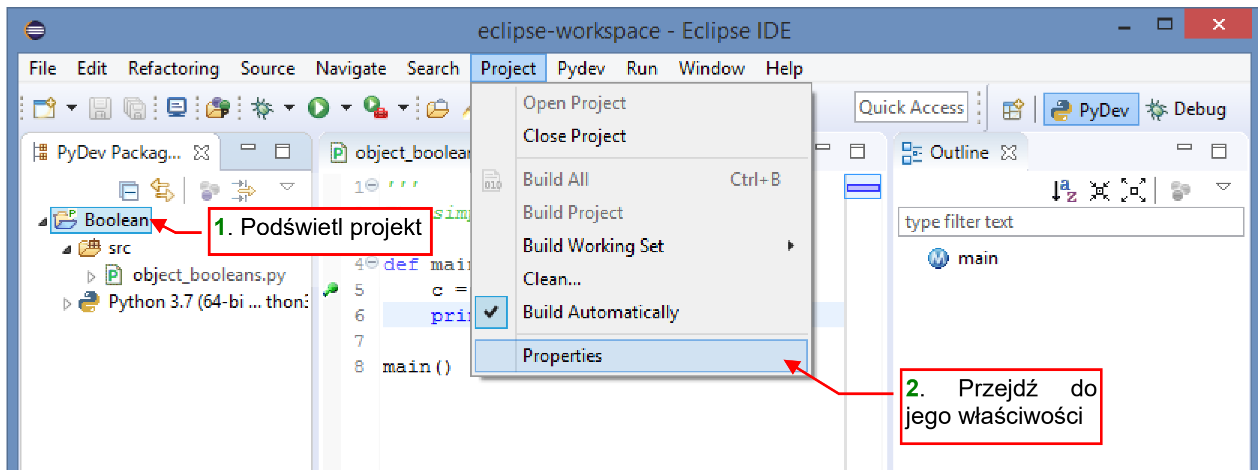
- W folderze z plikiem wykonywalnym Blendera (katalog *Blender* - jak na Rysunek 3.2.2) musi się znaleźć jeden plik: *pydev_debug.py*. Jest potrzebny do prawidłowego debugowania skryptów.
- Plik *Run.py* i folder *doc* możesz umieścić w jakimkolwiek innym katalogu, tylko potem popraw w pliku wsadowym *doc\refresh_python_api* linię z wywołaniem programu *..\blender* (por. str. 141).
- Jeżeli jednak umieszczasz plik *Run.py* i folder *doc* w folderze Blendera, nie zapomnij o nadaniu grupie *Użytkownicy* uprawnień do tworzenia/zapisu w tym folderze (tj. folderze *Blender* wraz z podfolderami).



Rysunek 3.2.2 Pliki, których będziemy używać w dalszych sekcjach tej książki

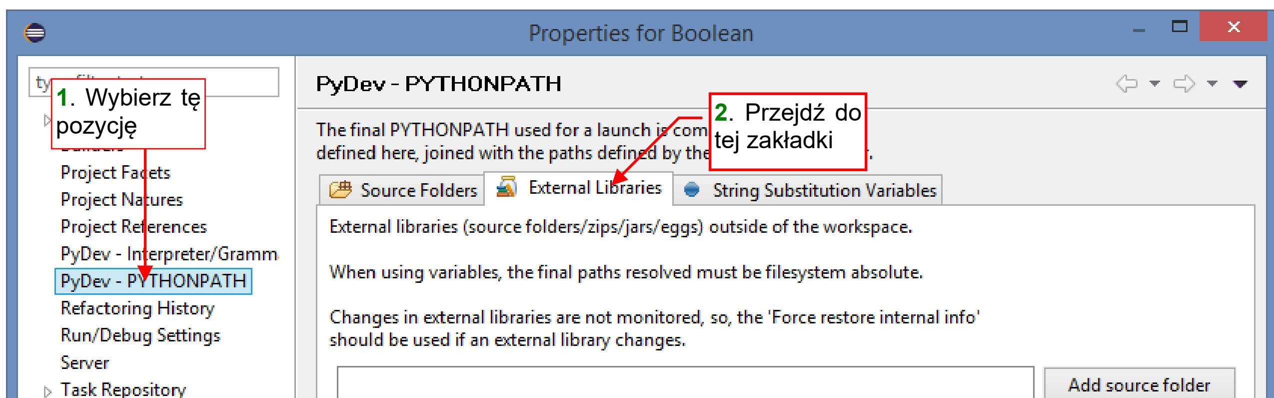
Copyright Witold Jaworski, 2011-2019.

Gdy pliki są na miejscu, trzeba zmienić konfigurację projektu (***Project→Properties***, Rysunek 3.2.3):



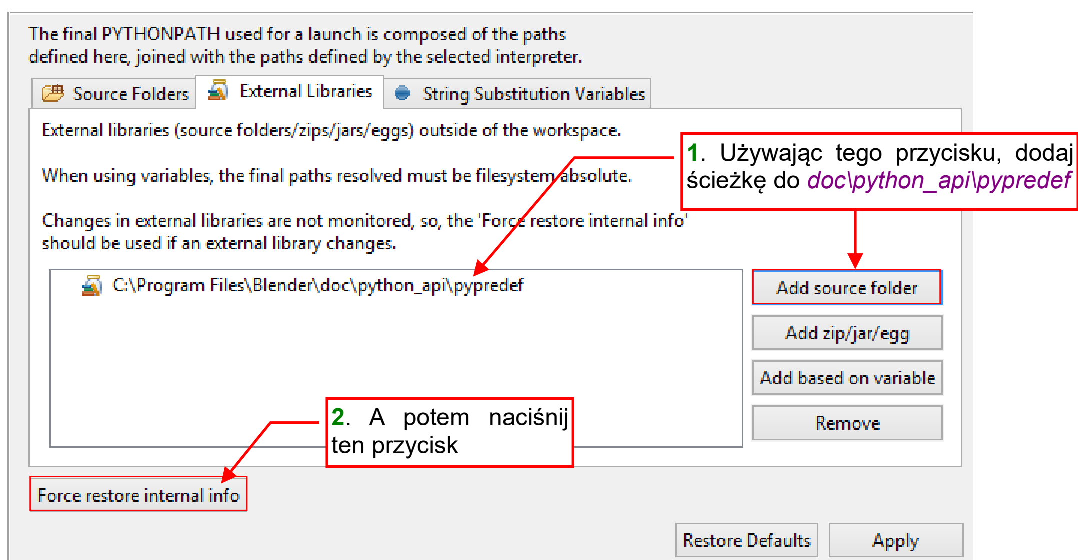
Rysunek 3.2.3 Przejście do konfiguracji projektu

W oknie, które się pojawi, wybierz sekcję ***PyDev – PYTHONPATH***, a w niej — zakładkę ***External Libraries*** (Rysunek 3.2.4):



Rysunek 3.2.4 Przejście do konfiguracji projektu

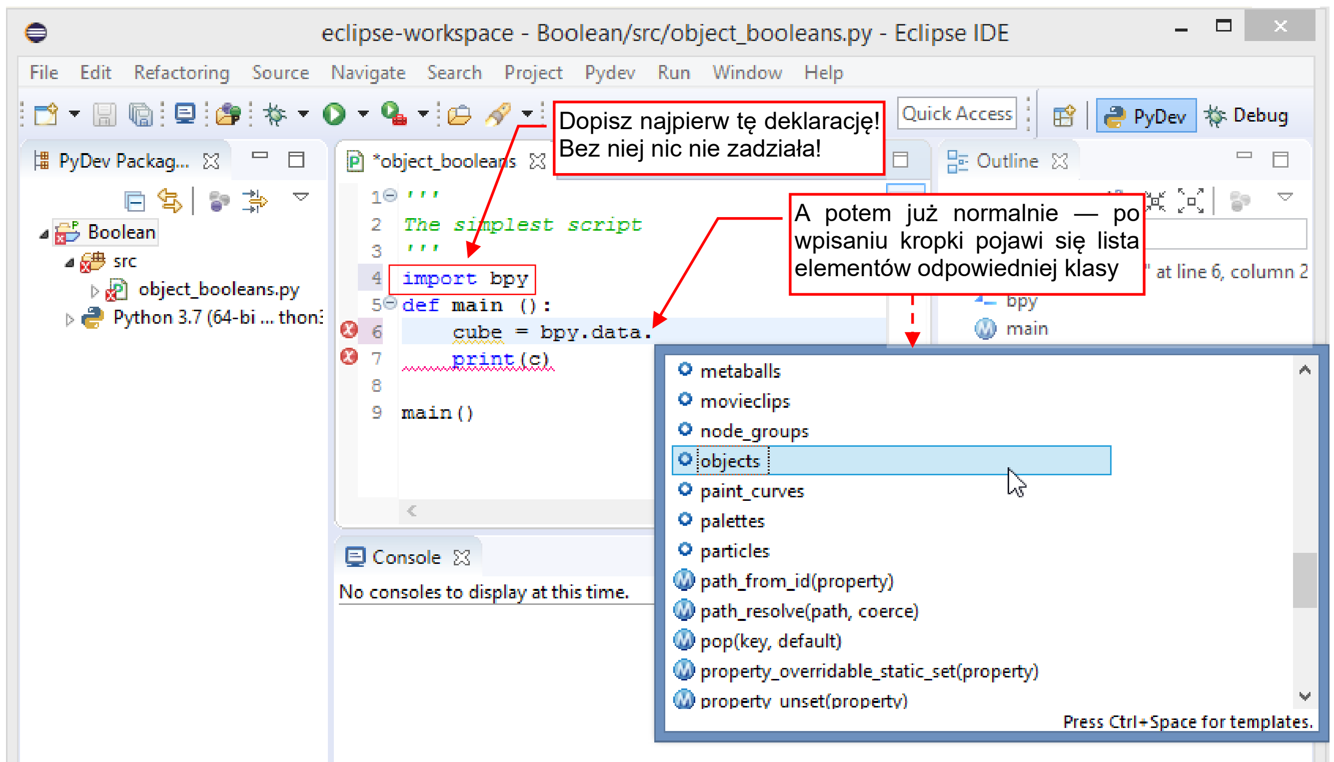
Dopisz tu (***Add source folder***) ścieżkę do katalogu *doc\python_api\pypredef* (Rysunek 3.2.5):



Rysunek 3.2.5 Konfiguracja *PYTHONPATH*

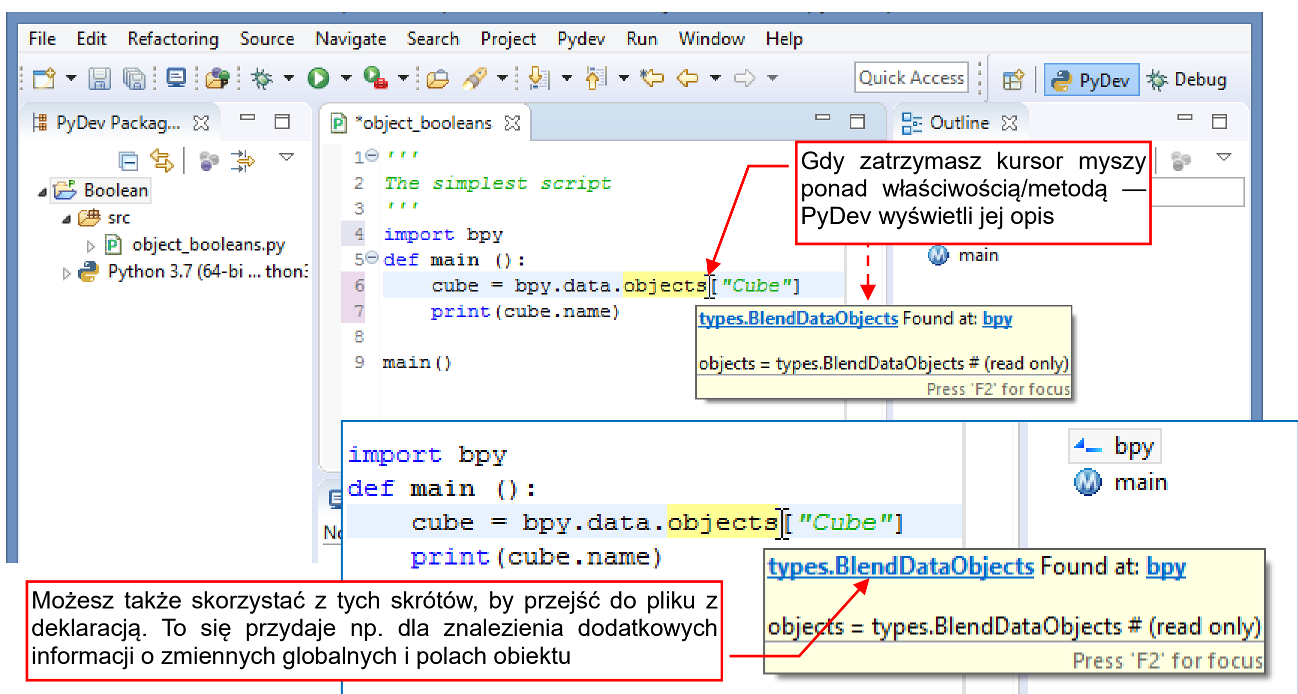
Koniecznie naciśnij potem przycisk ***Force restore internal info*** (Rysunek 3.2.5). Spowoduje to „przejście" przez pasek stanu Eclipse informacji o postępie aktualizacji (przez sekundę lub dwie).

Od tej chwili, gdy dopiszesz do skryptu odpowiednią deklarację *import*, PyDev zacznie podpowiadać właściwości i metody klas Blendera (Rysunek 3.2.6):



Rysunek 3.2.6 Uzupełnianie kodu dla API Blendera

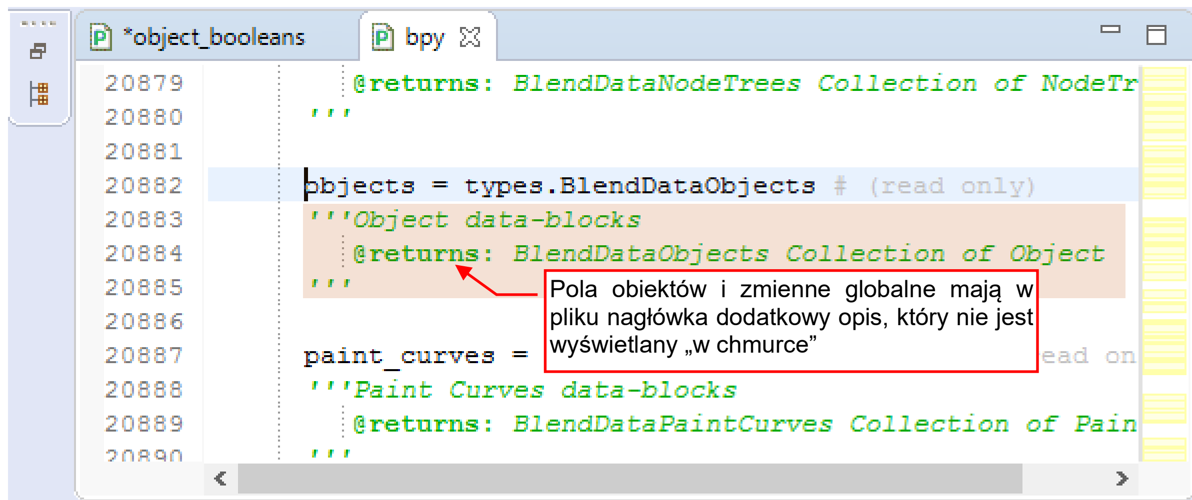
Podpowiedzi pojawiają się zazwyczaj po wpisaniu kropki (lub Ctrl-Space). A gdy zatrzymasz na chwilę kursor myszy ponad nazwą metody — PyDev wyświetli jej opis „w chmurce" (Rysunek 3.2.7)



Rysunek 3.2.7 Wyświetlanie opisów elementu

Co prawda te „chmurki" są kwadratowe, ale nazywam je tak ze względu na „ulotność" (znikną, gdy poruszysz myszką). No, chyba że klikniesz w umieszczony w nich skrót (por. Rysunek 3.2.7). Wtedy PyDev przeniesie Cię do definicji tego elementu (Rysunek 3.2.8):

Copyright Witold Jaworski, 2011-2019.



Rysunek 3.2.8 Definicja pola *objects* w pliku *bpy.pypredef*, otwarta za pomocą skrótu z opisu elementu

Taka definicja znajduje się, z punktu widzenia PyDev, w pliku *bpy.pypredef*. Eclipse otworzy odpowiedni plik nagłówków w edytorze (o ile nie został już w nim wcześniej otwarty).

Możesz także zażądać wskazania pozycji wybranego pola/metody w oknie *Outline* (Rysunek 3.2.9):

1. Ustaw karetkę przy ***objects***, a gdy PyDev je podświetli – otwórz menu kontekstowe (PPM)

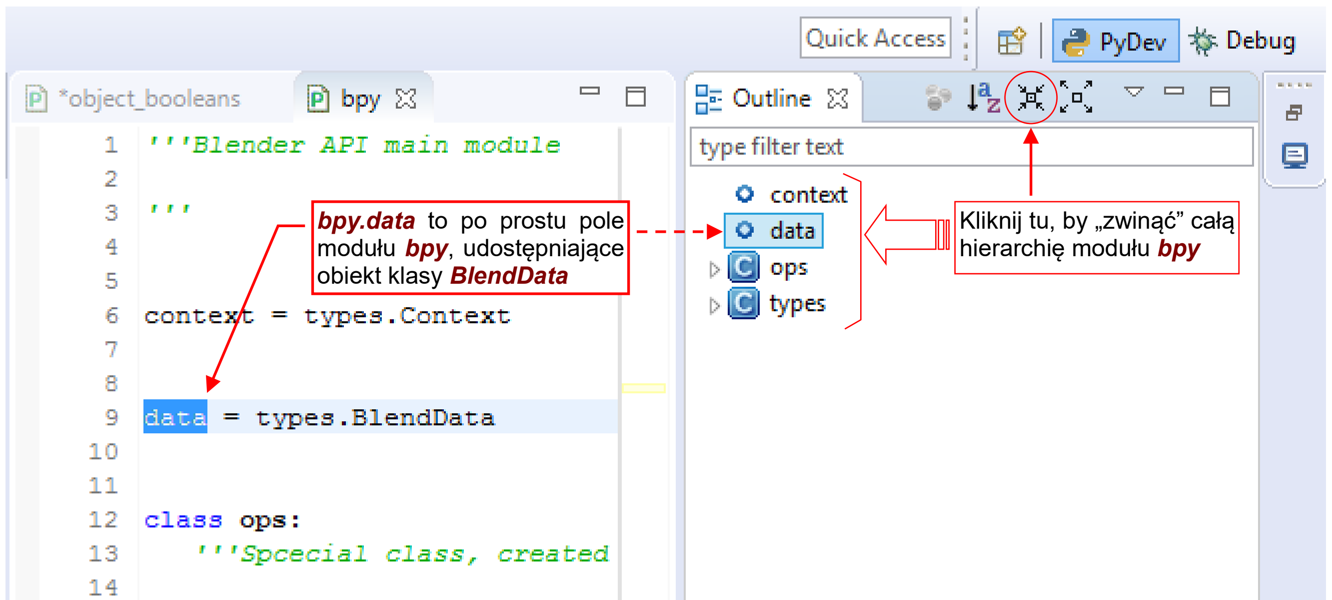
2. Wywołaj ***Show In→Outline***

W oknie diagramu szybko znajdziesz, do jakiej klasy należy wskazany element (w tym przypadku: ***BlendData***)

W oknie diagramu PyDev odnajdzie i podświetli ten element (tutaj: ***objects***)

Rysunek 3.2.9 Znajdowanie wskazanego elementu klasy w panelu *Outline*.

Zwróć uwagę, że widok *Outline* może być czymś w rodzaju „pomocy szkoleniowej”. Pozwala się zapoznać z podstawową strukturą API Blendera. Zacznijmy od „zwinięcia” całej hierarchii (Rysunek 3.2.10):

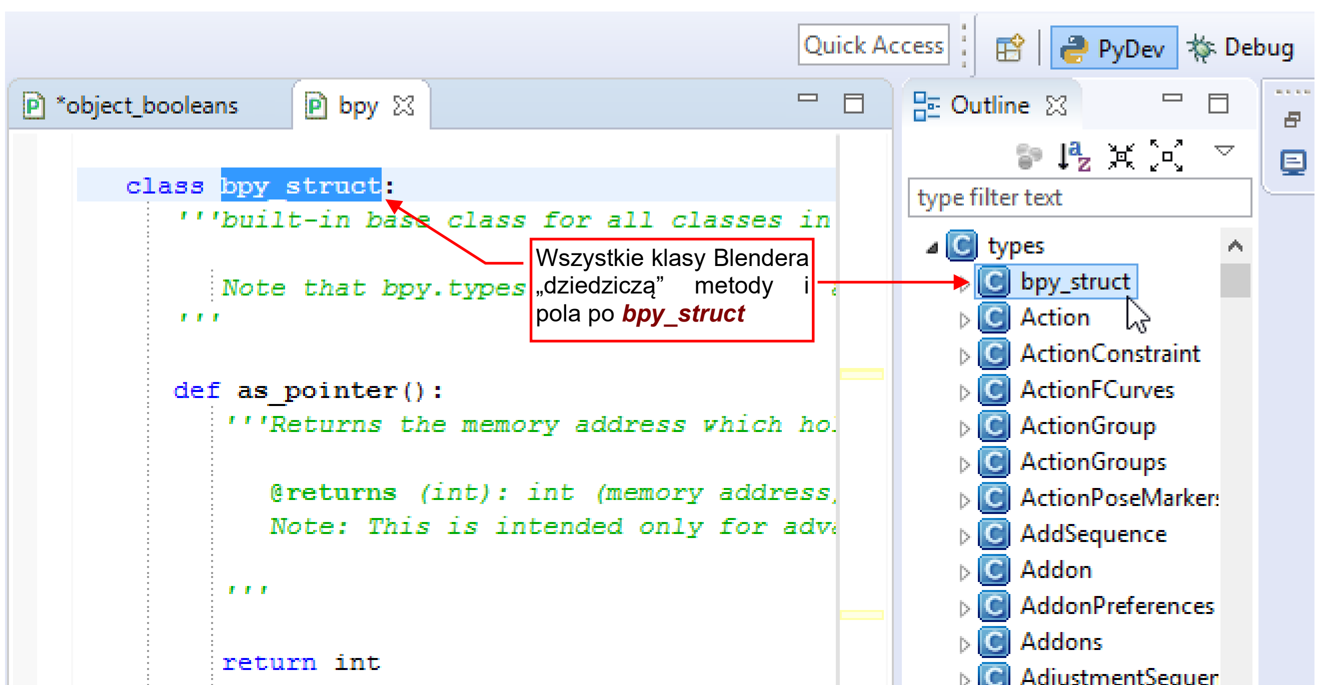


Rysunek 3.2.10 Podstawowa struktura API Blendera

Są to podstawowe elementy API:

- ***bpy.data*** udostępnia skryptom dane z aktualnego pliku Blendera. Jego polami są kolekcje różnych rodzajów obiektów (***scenes***, ***objects***, ***meshes***, itp. — por. Rysunek 3.2.9);
- ***bpy.context*** to dane o bieżącym stanie „środowiska”: aktywnym obiekcie, scenie, bieżącej selekcji;
- ***bpy.ops*** zawiera wszystkie polecenia Blendera, udostępnione także użytkownikowi poprzez GUI. (Dla Pythona każde polecenie to jedna z metod tego obiektu);
- ***bpy.types*** zawiera definicje wszystkich klas, używanych w obiektach, które występują w ***bpy.data*** i ***bpy.context***.

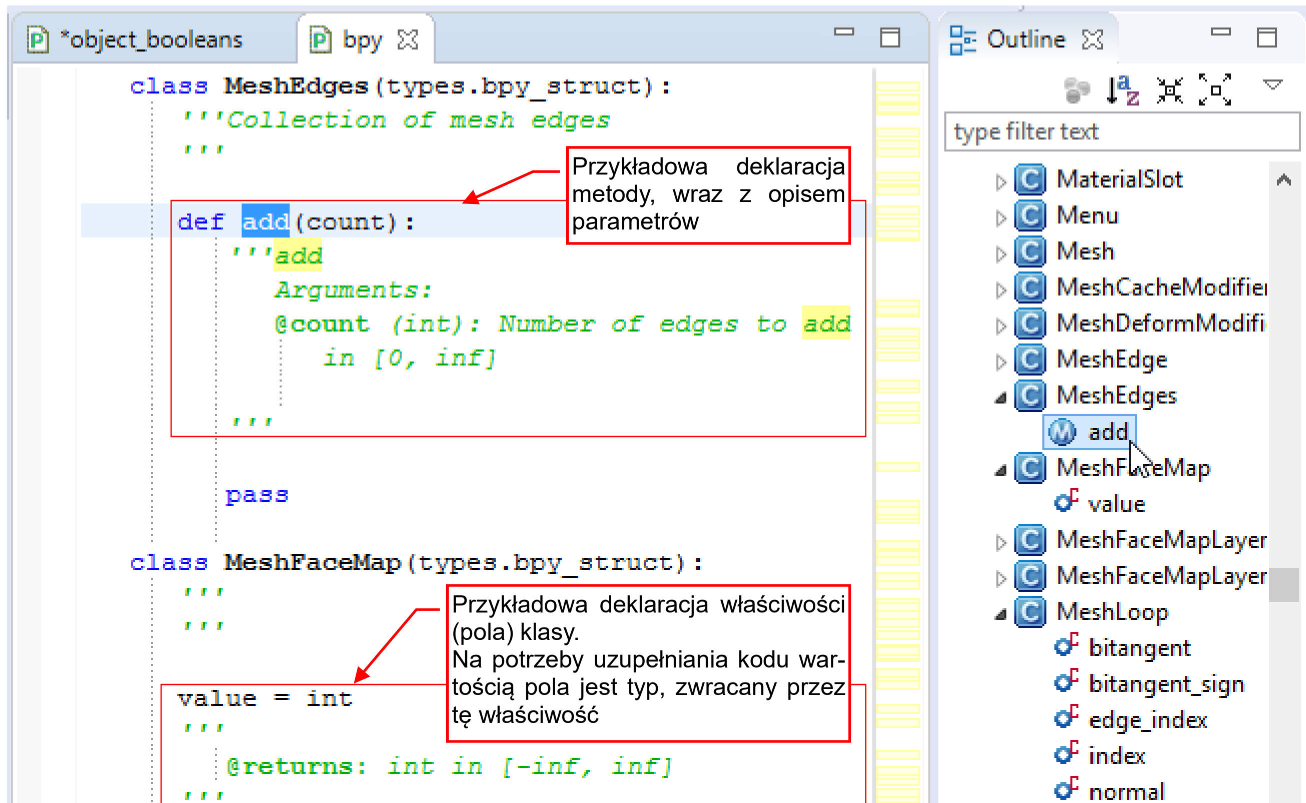
Gdy zajrzysz do wnętrza ***bpy.types***, zobaczysz alfabetyczną listę wszystkich klas, wykorzystywanych w API. Wyjątkowo na początku umieściłem deklarację ***bpy_struct***. To klasa bazowa dla wszystkich pozostałych. Jej metody i właściwości są dostępne zawsze w każdym obiekcie Blendera (Rysunek 3.2.11):



Rysunek 3.2.11 *bpy_struct*: **klasa bazowa wszystkich klas**

Copyright Witold Jaworski, 2011-2019.

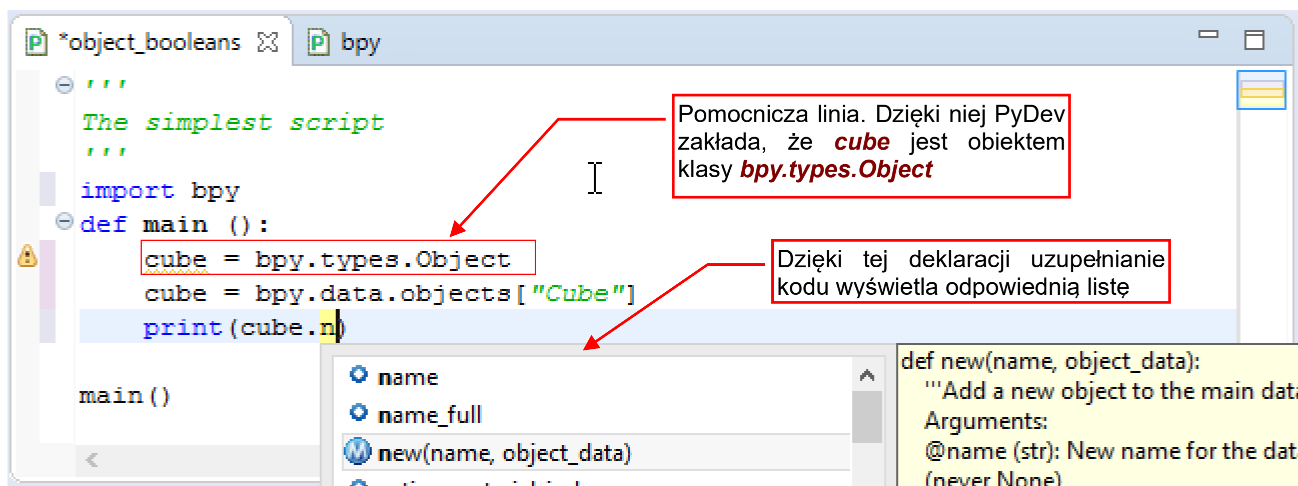
Inna sprawa, że w konkretnej klasie część właściwości ***bpy_struct*** może być nie zaimplementowana. Na przykład — ***bpy_struct*** ma metodę ***items()***. Dlatego wszelkie klasy kolekcji — jak chociażby ***MeshEdges*** (krawędzie siatki — obiekty ***MeshEdge***) implementują tylko jakieś dodatkowe metody, jak ***add()*** (Rysunek 3.2.12):



Rysunek 3.2.12 Klasy potomne — metody, właściwości

Oczywiście, dla wielu klas (np. wierzchołka siatki — ***MeshVertex***) implementacje metody ***items()*** (i wielu innych) są puste.

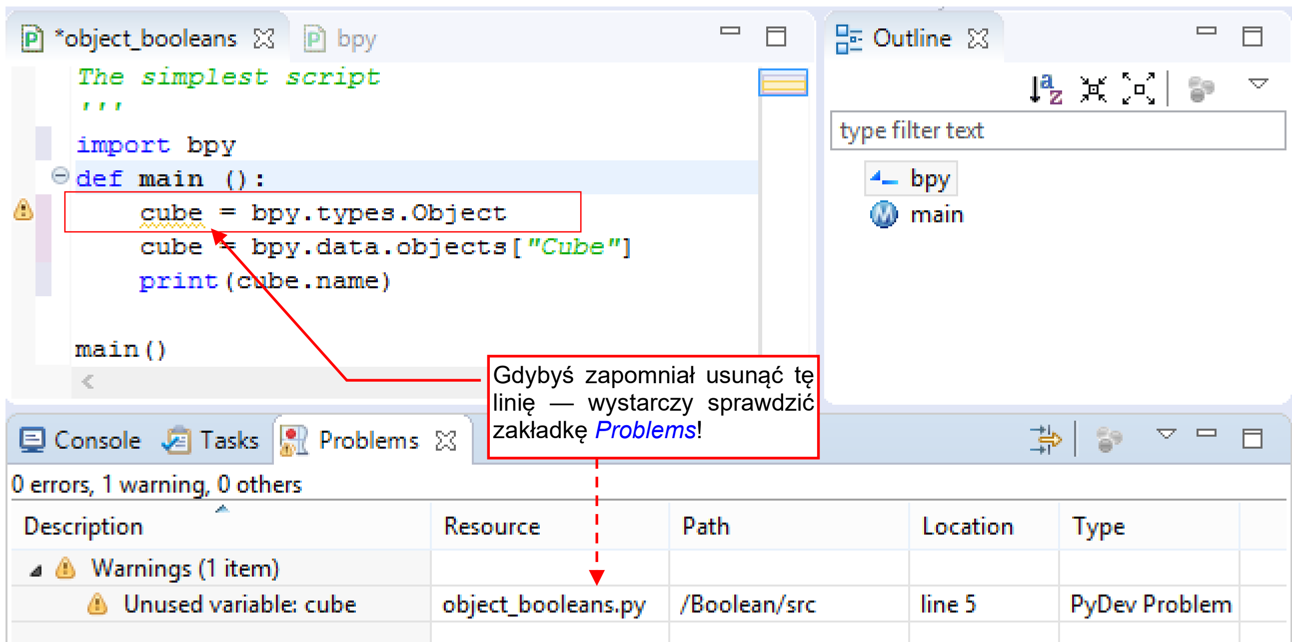
Wykorzystywanie takiej „odziedziczonej" metody ***items()*** przez każdą kolekcję utrudnia automatyczne uzupełniania kodu. PyDev odczytuje z definicji, że każda z nich zwraca po prostu ***bpy_struct***. (Bo tak wynika z ich deklaracji, odziedziczonych po klasie bazowej). Można jednak „zasugerować" interpreterowi odpowiedni typ. Wystarczy umieścić wcześniej linię, przypisującej zmiennej odpowiednią klasę obiektu (Rysunek 3.2.13):



Rysunek 3.2.13 Wyświetlanie opisu funkcji

W zasadzie taką linię powinieneś dopisać tylko na chwilę, gdy potrzebujesz skorzystać z automatycznej kompletacji. Pamiętaj, aby zawsze umieścić ją p r z e d pierwszym nadaniem wartości zmiennej. W ten sposób kod będzie działał poprawnie, nawet gdy o niej zapomnisz.

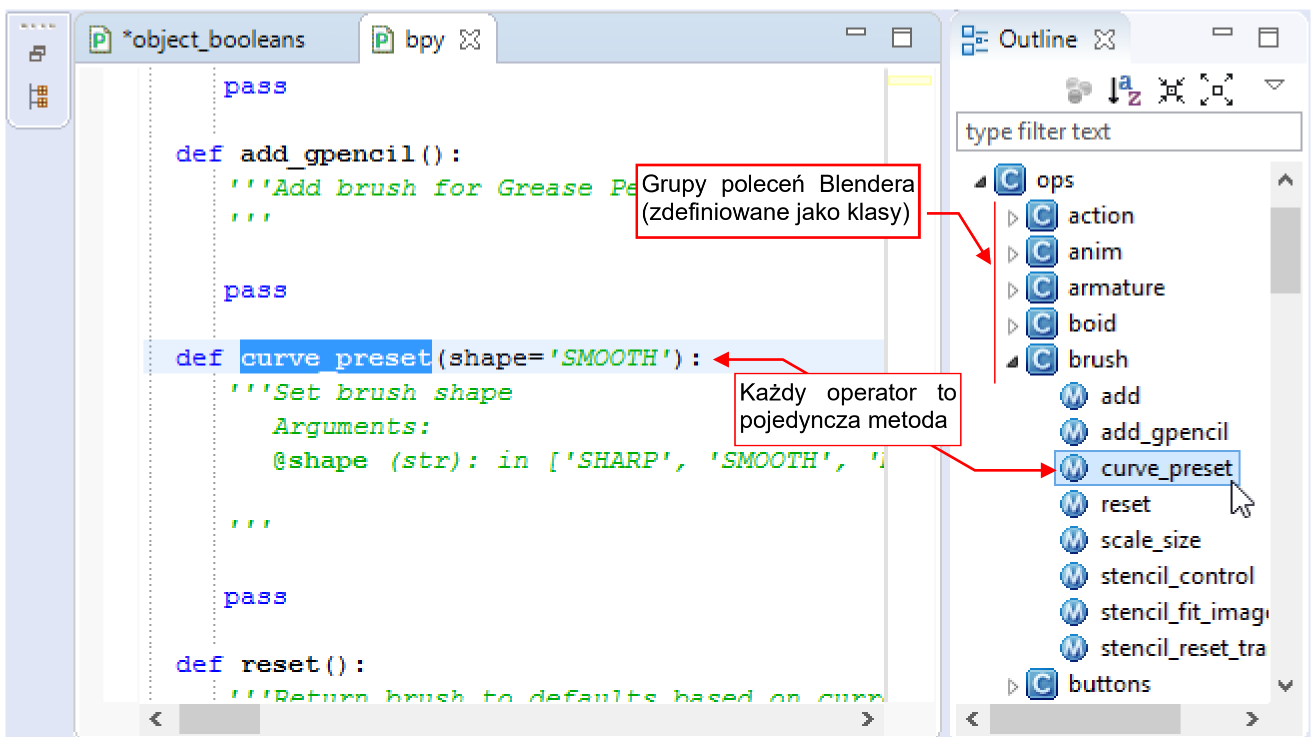
Zresztą — PyDev wykrywa takie linie, bo są to według niego nie używane zmienne. Umieszcza na nich odpowiednie ostrzeżenia (Rysunek 3.2.14):



Rysunek 3.2.14 Ostrzeżenia, związane z „deklarowaniem typu” dla uzupełniania kodu

Dobrą praktyką jest więc zajrzeć co jakiś czas do zakładki *Problems* (otwierana: *Windows→Show View*). Zobaczysz tam wszystkie linie, które zapomniałeś usunąć. Korzystając z tej listy, będziesz mógł to zaraz poprawić.

Do tej pory omawialiśmy gałąź ***bpy.types***. A operatory? Operatorów jest mnóstwo! Aby się wśród nich od razu nie zagubić, podzielono je na tematyczne grupy: ***action***, ***anim***, ***armature***, … i tak dalej. Rozwińmy chociażby zespół ***bpy.ops.brush*** (Rysunek 3.2.15):



Rysunek 3.2.15 Operatory: przykład opisu parametrów i ich wartości domyślnych

Każdy zespół operatorów (np. ***bpy.ops.brush***) to po prostu taka klasa, która ma same metody. Każdy operator (polecenie) to pojedyncza metoda takiej klasy. Zwróć uwagę, że operator zawsze można wywołać bez żadnych argumentów — zostaną wtedy wykorzystane ich wartości domyślne.

Copyright Witold Jaworski, 2011-2019.

W sumie — tak się złożyło, że głównym tematem tej sekcji stało się poznanie struktury modułów Blender API. W takim razie, aby skończyć temat rozpoczęty na str. 43, wyliczę tu pozostałe moduły ***bpy***. Są o wiele mniejsze od podstawowych (***bpy.data***, ***bpy.context***, ***bpy.types***, ***bpy.ops***) i pełnią rolę pomocniczą:

bpy.app różne pomocnicze informacje o samym programie: numer wersji, położenie pliku *blender.exe*, flagi kompilacji, itp.;

bpy.path pomocnicze funkcje do pracy ze ścieżkami i plikami (podobne zagadnienia w standardowym Pythonie obsługuje moduł ***os.path***);

bpy.props definicje specjalnych „właściwości" klasy, które Blender potrafi wyświetlać w swoich okienkach (gdy zajdzie taka potrzeba). Dla odróżnienia od zwykłych właściwości nazwałbym je „właściwościami Blendera". Będziemy je wykorzystywać w następnym rozdziale, przy okazji tworzenia operatora;

bpy.utils obsługa rejestracji wtyczek, ładowania modułów Pythona, wywoływania innych programów, informacji o ścieżkach dostępu, itp. Ten moduł ma jeszcze dwa podmoduły: ***units*** i ***previews***;

bpy_extras dalsze pomocnicze metody i obiekty. Zawiera osiem „tematycznych" podmodułów: ***anim_utils***, ***object_utils***, ***io_utils***, ***image_utils***, ***keyconfig_utils***, ***mesh_utils***, ***node_utils***, ***view3d_utils***

Oprócz podstawowej sekcji ***bpy*** Blender API udostępnia jeszcze inne moduły:

mathutils pomocnicze klasy matematyczne: ***Matrix*** (4x4), ***Euler***, ***Quaternion*** (odwzorowanie obrotu), ***Vector***, ***Color***. Zawiera także podmoduł ***geometry*** z kilkoma pomocniczymi funkcjami (przecięcie z linii, przecięcie promienia z płaszczyzną, itp.);

freestyle sześć podmodułów (***types***, ***predicates***, ***functions***, ***chainingiterators***, ***shaders***, ***utils***) związane z obsługą pomocniczego, „rysunkowego" renderera (*NPR*) Freestyle;

bgl procedury pozwalające skryptom rysować w przestrzeni okien Blendera (to w istocie większość funkcji OpenGL 1.1);

gpu procedury pozwalające skryptom rysować w przestrzeni okien Blendera: to preferowany następca funkcji ze „staromodnego" modułu ***bgl***. Zawiera cztery podmoduły: ***types***, ***shader***, ***matrix*** i ***select***;

bmesh alternatywne API do obsługi siatek (*boundary meshes*). Zawiera cztery podmoduły: ***ops***, ***types***, ***geometry*** i ***select***;

O dalszych modułach — ***aud*** (*Audio*), ***blf*** (*Font Drawing*), ***idprop.types*** (*ID Property Access*) — mam mgliste pojęcie. Nie będę się więc o nich rozpisywał.

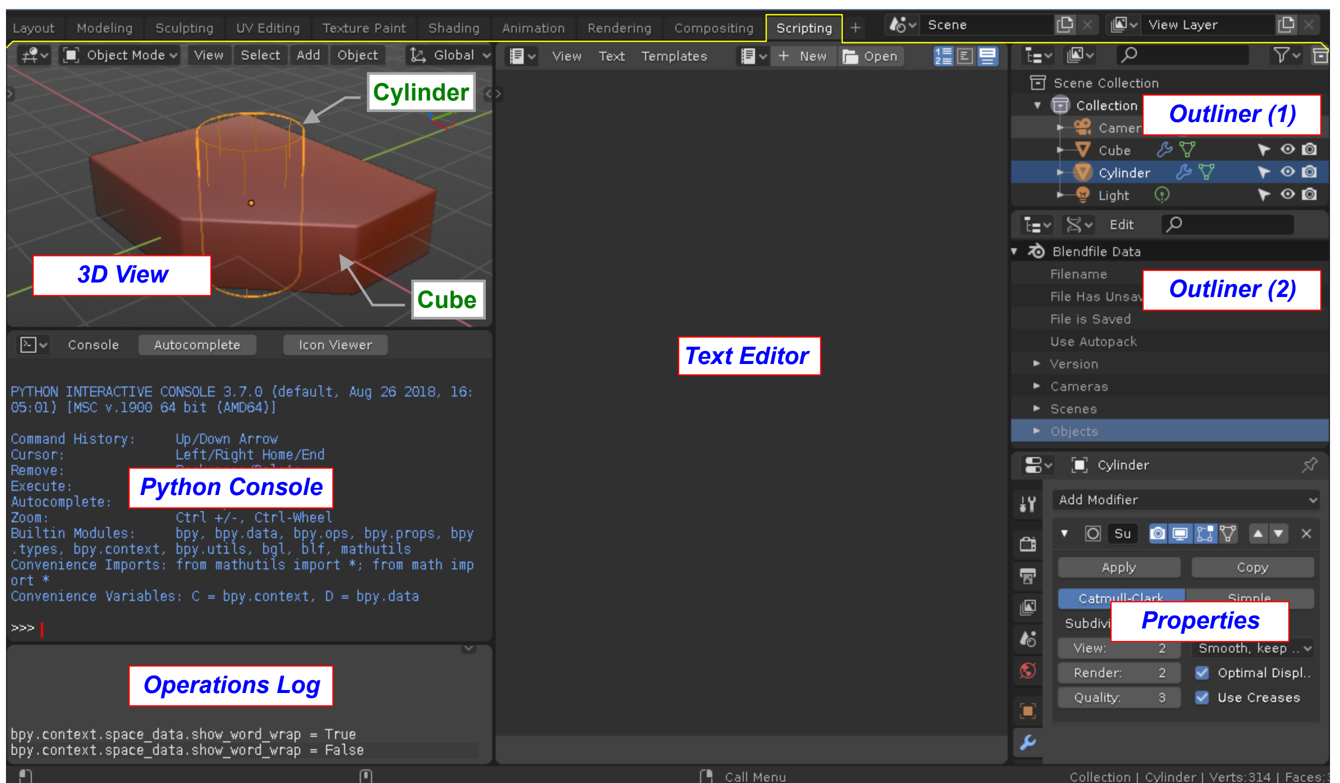
Podsumowanie

- Pliki **.pypredef*, umożliwiające autokompletację metod i właściwości klas API Blendera, znajdziesz w pliku towarzyszącym tej książce (str. 39);
- Po rozpakowaniu plików **.pypredef*, należy ich folder dołączyć do **PYTHONPATH** projektu (str. 40);
- Autokompletacja Python API zaczyna działać po umieszczeniu na początku skryptu odpowiedniego wyrażenia importu (str. 41);
- Wyświetlane przez Eclipse „chmurki" ze szczegółowym opisem metod można wykorzystać do dalszego poznawania funkcji API (str. 41);
- Skróty do modułu ***bpy***, pojawiające się w „chmurkach", można wykorzystać do otworzenia pliku *bpy.pypredef* w edytorze Eclipse (str. 41). Skorzystanie z tego skrótu pozwala odczytać opis właściwości (atrybutów) klasy, których Eclipse nie wyświetla (str. 44);
- Przeglądanie struktury modułu ***bpy*** w panelu *Outliner* pomaga także poznać strukturę Python API Blendera (str. 42);
- W wielu przypadkach do uzyskania poprawnej autokompletacji należy stosować „deklaracje zmiennych" (str. 44);

3.3 Opracowanie podstawowego kodu

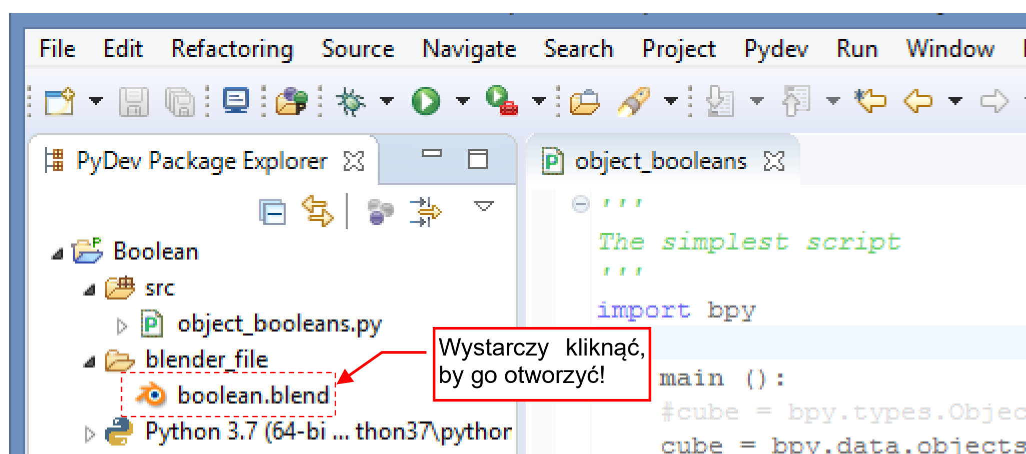
Zazwyczaj w różnych poradnikach znajdujesz od razu gotowy kod, który czasami autor trochę skomentuje. W tej sekcji chciałbym pokazać coś, co musi nastąpić wcześniej: poszukiwanie rozwiązania. Ten etap jest równie ważny jak pisanie programu, a może ważniejszy. Za każdym razem pozwala mi poznać kolejny kawałek API Blendera. (Nawet gdybym chciał, nie jestem w stanie spamiętać wszystkich jego klas, pól i metod).

Przygotujmy sobie najpierw środowisko do testów. Przekształciłem w tym celu domyślny sześcian (**Cube**) w płytę. Aby nadać jej bardziej „techniczny" wygląd, zaokrągliłem jej krawędzie modyfikatorem *Bevel*. Potem wstawiłem w tę płytę walec (**Cylinder**). Zamierzam go używać jako „narzędzia", więc wyświetlam tylko jego siatkę (*Wireframe*). Do pracy nad skryptem wykorzystamy standardową zakładkę (*workspace*) Blendera 2.8 o nazwie ***Scripting*** (Rysunek 3.3.1):



Rysunek 3.3.1 Propozycja układu ekranu w testowym pliku Blendera

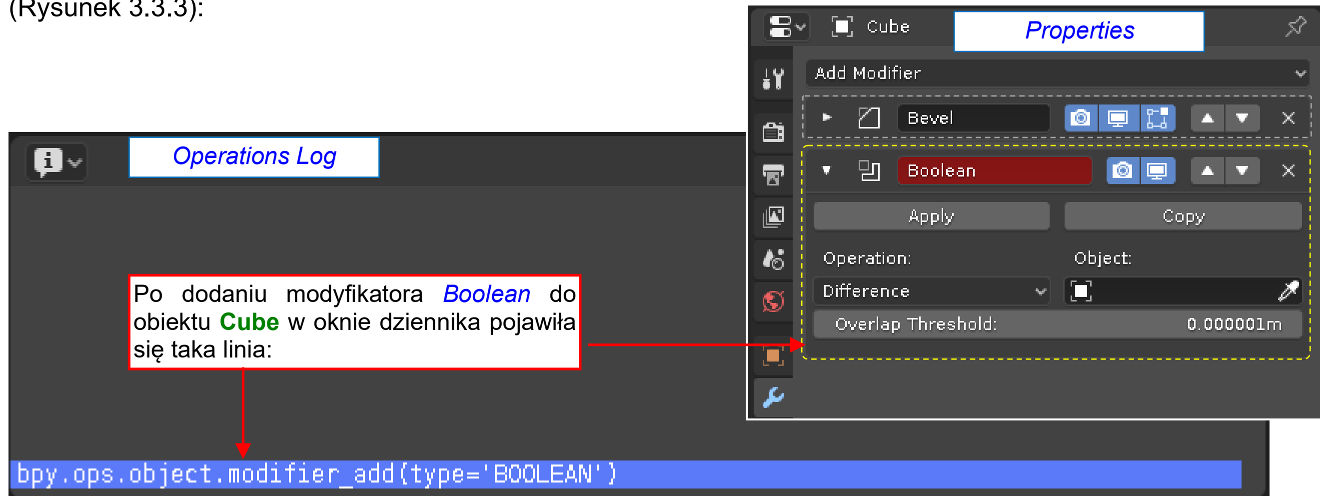
Zapisz ten plik Blendera gdziekolwiek na dysku, a następnie wstaw go do projektu Eclipse (szczegóły na str. 145), aby go zawsze „mieć pod ręką" (Rysunek 3.3.2):



Rysunek 3.3.2 Plik Blendera, wstawiony w projekt

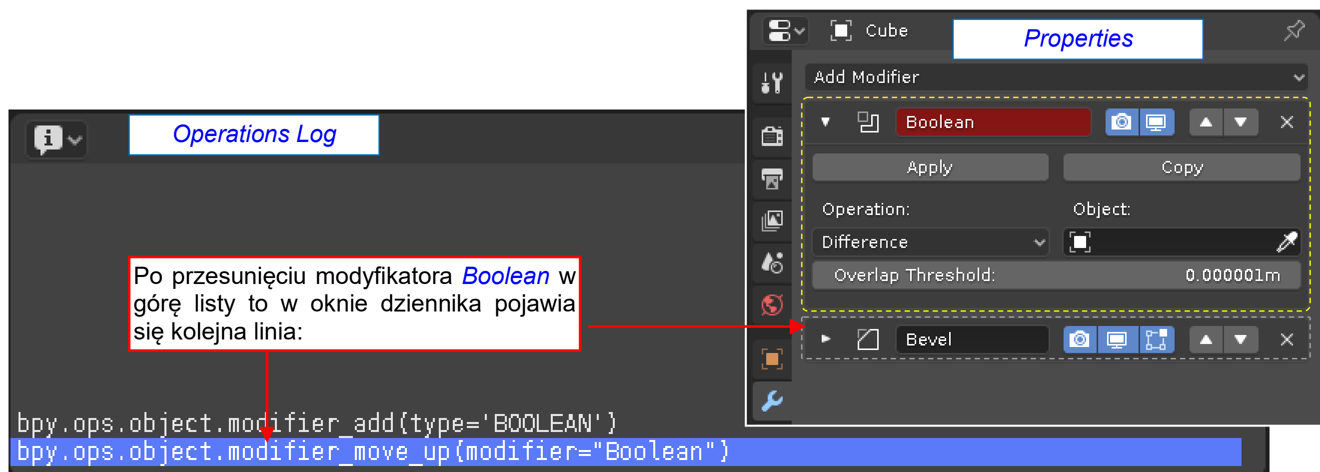
Copyright Witold Jaworski, 2011-2019.

Celem tej sekcji jest przygotowanie skryptu, który wykona kroki opisane w sekcji 3.1 (Rysunek 3.1.3 - Rysunek 3.1.6). Gdy wywołujesz jakiekolwiek polecenie w Blenderze, w oknie dziennika operacji (*Operations Log* – por. Rysunek 3.3.1, str. 47) pojawia się jego ekwiwalent zapisany w Pythonie. To najlepsze miejsce do odczytania, co powinniśmy umieścić w naszym kodzie. Zacznijmy od dodania do aktywnego obiektu modyfikatora *Boolean* (Rysunek 3.3.3):



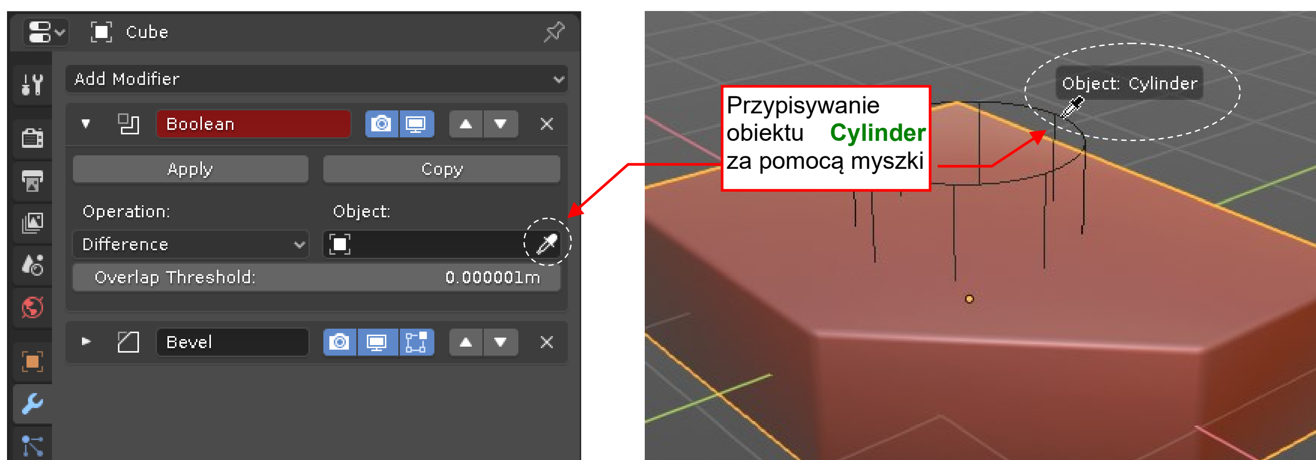
Rysunek 3.3.3 Dodanie do aktywnego obiektu modyfikatora *Boolean*

Podobnie dla następnego kroku – przesunięcia modyfikatora *Boolean* w górę listy modyfikatorów – w oknie dziennika pojawia się odpowiednie polecenie Pythona (Rysunek 3.3.4):



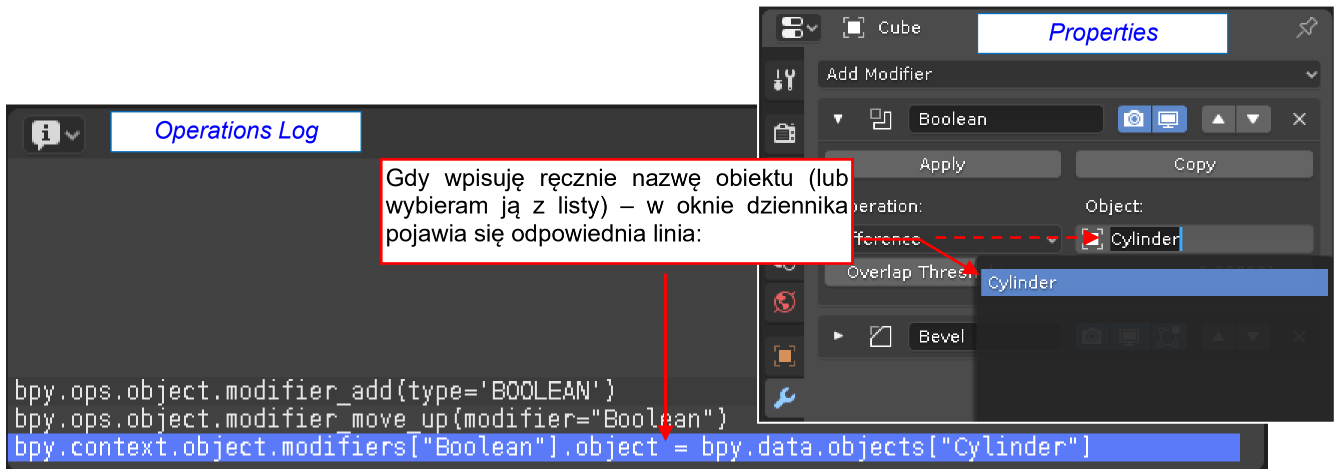
Rysunek 3.3.4 Przesunięcie modyfikatora *Boolean* w górę stosu modyfikatorów

W trzecim kroku przypisałem do modyfikatora *Boolean* obiekt **Cylinder**, wskazując go – za pomocą „pipetki” na ekranie (Rysunek 3.3.5):



Rysunek 3.3.5 Przypisanie obiektu Cylinder do modyfikatora (za pomocą myszki)

To nie spowodowało żadnych zmian w oknie dziennika. Jednak, gdy wpisałem nazwę obiektu ręcznie do pola *Object*, to w dzienniku pojawiła się nowa linia (Rysunek 3.3.6):



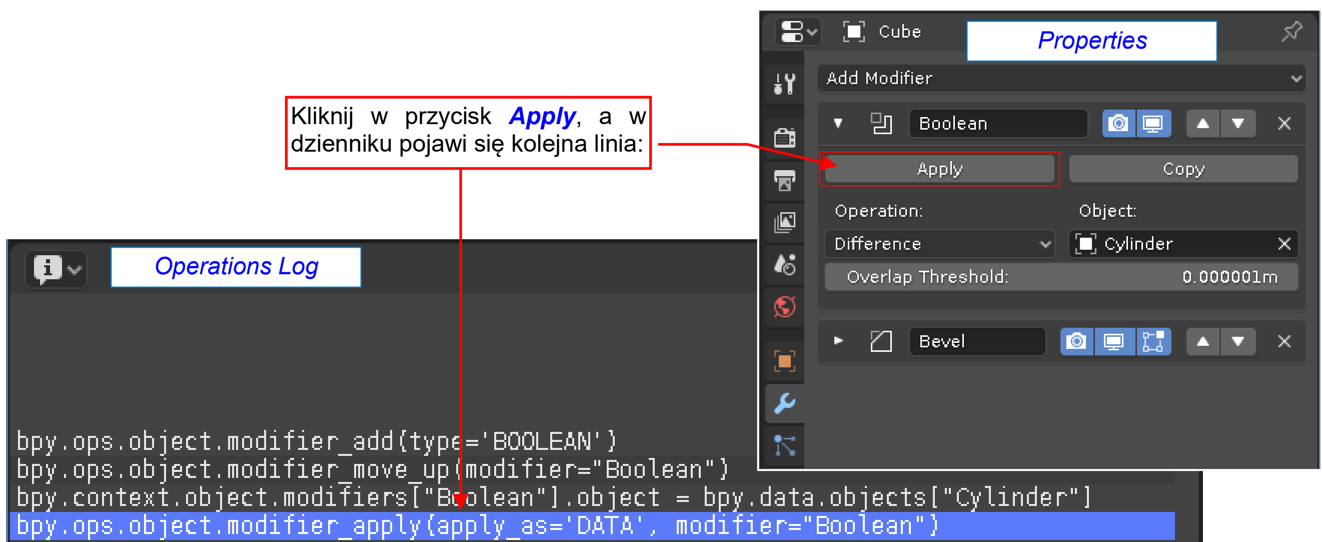
Rysunek 3.3.6 Przypisanie obiektu Cylinder **do modyfikatora (poprzez wpisanie nazwy)**

Ta linia nie jest kolejnym wywołaniem polecenia, tylko przypisaniem. Warto się jej bliżej przyjrzeć, aby zrozumieć, co w tym miejscu nastąpiło. Obiekt ***bpy.context*** dostarcza skryptom informacji o „kontekście", w którym są wykonywane: jakie obiekty są aktualnie wskazane przez użytkownika, jakie okno jest aktywne, itp. Odwołanie do ***bpy.context.object*** zwraca obiekt aktywny (ten, który wskazałeś jako ostatni). W tej linii odwołujemy się do związanego z tym obiektem modyfikatora o nazwie **Boolean** (***modifiers["Boolean"]***) któremu przypisujemy obiekt „tnący" o nazwie **Cylinder** (***bpy.data.objects["Cylinder"]***). Robimy to poprzez przypisanie tego obiektu do pola ***object*** modyfikatora.

Zwróć uwagę, jak Blender odwołuje się do obiektu **Cylinder**. W linii w oknie dziennika wybrano ten obiekt z listy ***bpy.data.objects***, posługując się jego nazwą. To preferowana w API Blendera metoda odwoływania się do danych. Moduł ***bpy.data*** udostępnia skryptom zawartość aktualnego pliku Blendera w postaci list. Lista ***objects*** zawiera wszystkie obiekty pliku. Nazwa każdego obiektu jest unikalna i można ją użyć jako indeksu listy. (Gdy w oknie właściwości spróbujesz przypisać obiektowi nazwę, która już została użyta, Blender sam ją poprawi).

- Wszystkie typy danych (*data block*) Blendera mają w ***bpy.data*** swoje listy: siatki (***meshes***), materiały (***materials***), tekstury (***textures***), kolekcje obiektów (***collections***), itp. Z każdej z tych list możesz wybrać odpowiedni element, posługując się jego nazwą.

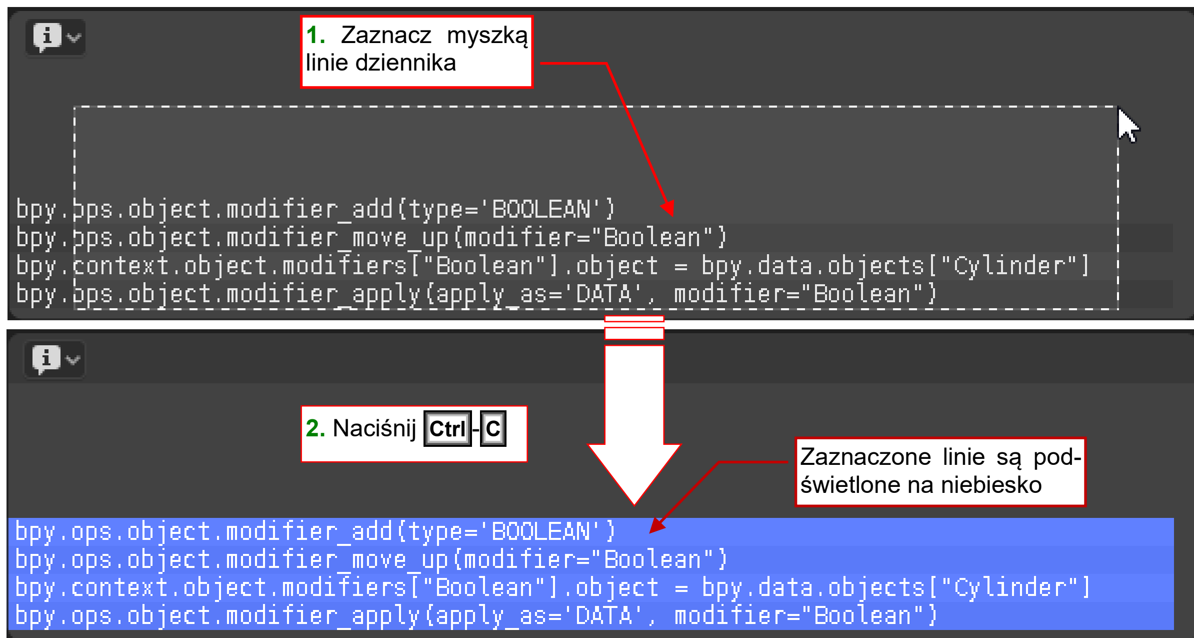
Teraz pozostało już tylko zakończyć tę operację, klikając w przycisk ***Apply*** modyfikatora (Rysunek 3.3.7):



Rysunek 3.3.7 „Utrwalenie" rezultatu modyfikatora Boolean

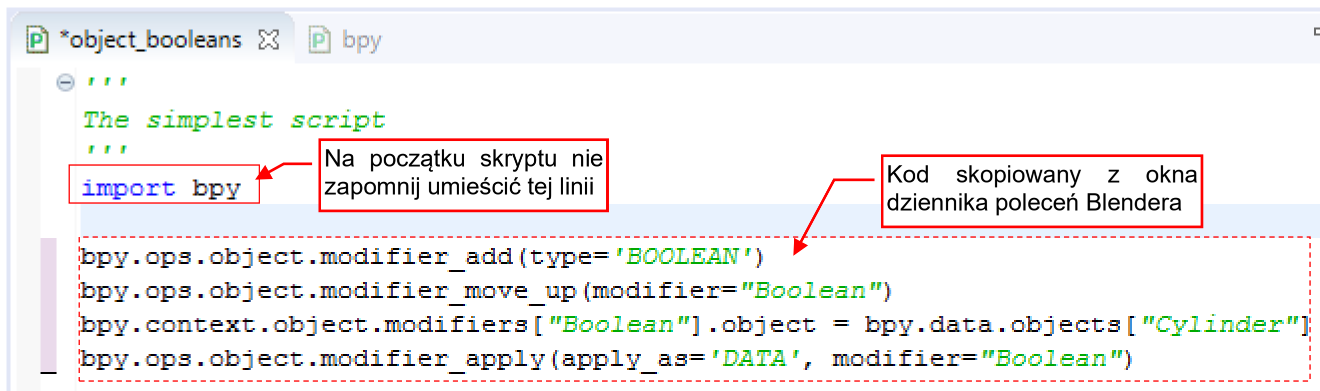
Copyright Witold Jaworski, 2011-2019.

Jak widzisz, podstawowy skrypt w zasadzie sam się napisał w okienku dziennika poleceń. Teraz można ten kod skopiować do schowka: wystarczy zaznaczyć myszką i nacisnąć Ctrl-C (Rysunek 3.3.8):



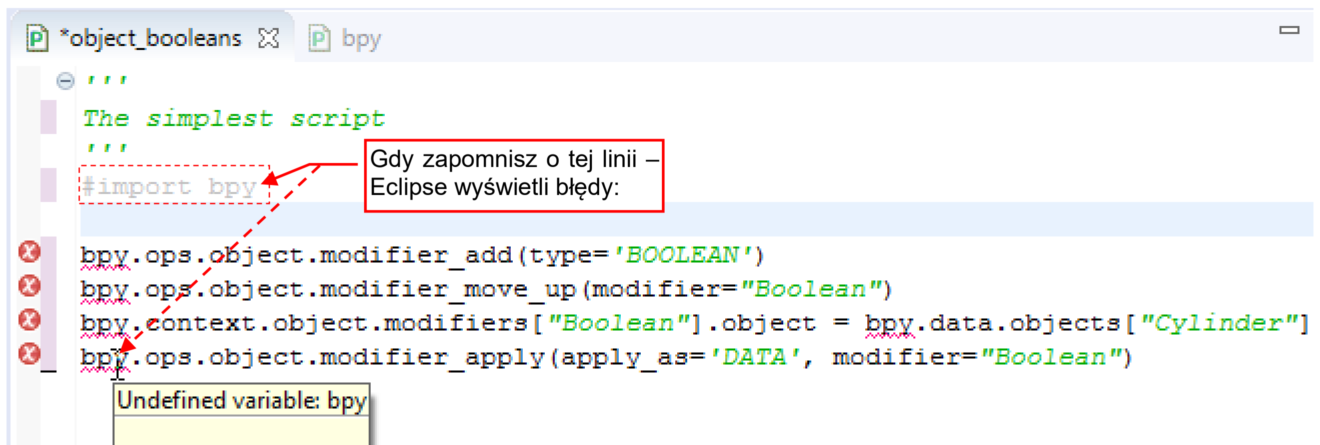
Rysunek 3.3.8 Kopiowanie linii z okna dziennika do schowka

A potem można wkleić te linie do Eclipse (Rysunek 3.3.9):



Rysunek 3.3.9 Najprostszy skrypt – skopiowane linie z dziennika operacji Blendera

To pierwsze przybliżenie docelowego kodu. Nie zapomnij dopisać u góry importu modułu ***bpy***, bo inaczej zobaczysz na ekranie takie błędy, jakie pokazuje Rysunek 3.3.10:



Rysunek 3.3.10 Efekt braku referencji do modułu *bpy*

Blender także by je zgłosił. Zawsze dbaj o to, by Eclipse nie wyświetlał żadnego błędu w Twoim kodzie.

Taki skrypt będzie działać, ale tylko dla danych z naszego testowego pliku Blendera. Nim go po raz pierwszy uruchomimy, zmieńmy nieco strukturę tego kodu. W ten sposób lepiej przygotujemy go do późniejszych modyfikcji (Rysunek 3.3.11):

*object_booleans bpy

```
'''
Boolean operator (ver. 0.01)
'''
import bpy

def boolean_operation (tool):
    '''Performs a Boolean operation on the active object
        Arguments:
        @tool (Object): the other object, not affected by this method
    '''
    bpy.ops.object.modifier_add(type='BOOLEAN')
    bpy.ops.object.modifier_move_up(modifier="Boolean")
    bpy.context.object.modifiers["Boolean"].object = tool
    bpy.ops.object.modifier_apply(apply_as='DATA', modifier="Boolean")

boolean_operation(bpy.data.objects["Cylinder"])
print("bool_operation: Done!")
```

Oryginalny kod operacji "obudowałem" tą procedurą

Komentuję każde pole, metodę czy funkcję.

Obiekt "tnący" przekazuję jako argument

Wywołanie procedury

Jako argument **tool** przekazuję obiekt **Cylinder**

Tymczasowa linia „diagnostyczna"

Rysunek 3.3.11 Ten sam skrypt, w poprawionej formie

Oryginalny kod operacji umieściłem w procedurze, którą nazwałem ***boolean_operation()***. Dzięki temu dalszy kod stanie się krótszy i bardziej czytelny. W przyszłości ten operator będzie używał jako narzędzi obiektów wskazanych przez użytkownika, dlatego dodałem tej procedurze argument: obiekt ***tool***. W kodzie procedury przypisuję go do modyfikatora. Na potrzeby tego pierwszego testu wykorzystuję konkretny obiekt (**Cylinder**), przekazywany w wywołaniu procedury ***boolean_operation()*** w przedostatniej ostatniej linii skryptu.
W ostatniej linii umieściłem polecenie wyświetlające tekst w konsoli. Taka linia czasami się przydaje podczas testowania: pozwala się szybko upewnić, że skrypt się wykonał. Przy okazji pozwoli mi w następnej sekcji pokazać opuszczanie przez debugger kodu procedury (por. str. 31).

Przy okazji: jak widzisz, od razu dodałem do procedury **boolean_operation()** *komentarz, opisujący krótko: co robi i jakich argumentów oczekuje. To taka „dobra praktyka". Mimo pozorów, robię to dla samego siebie: konieczność napisania tych dwóch zdań zmusza mnie do przemyślenia, czy ta procedura jest na pewno potrzebna i czy jej argumenty są odpowiednie. Zwracam także szczególną uwagę, aby opisać oczekiwany typ argumentu i ewentualne związane z nim dodatkowe założenia. W ten sposób mogę pisać „czystszy" kod. (Dużo błędów w skryptach bierze się z wywołania metod czy funkcji z niewłaściwymi parametrami. A przecież można ich uniknąć, gdyż PyDev wyświetli taki komentarz przy każdym wywołaniu procedury). Mam także realistyczną ocenę mojej pamięci: gdy za rok czy dwa zajrzę do tego skryptu, to nie będę pamiętał już niczego. Wówczas moje komentarze pozwolą mi się szybko zorientować, o co chodzi. Odruch komentowania wyrobiłem sobie jeszcze na studiach i potem wiele razy mi się przydał. Polecam!*

Copyright Witold Jaworski, 2011-2019.

Podsumowanie

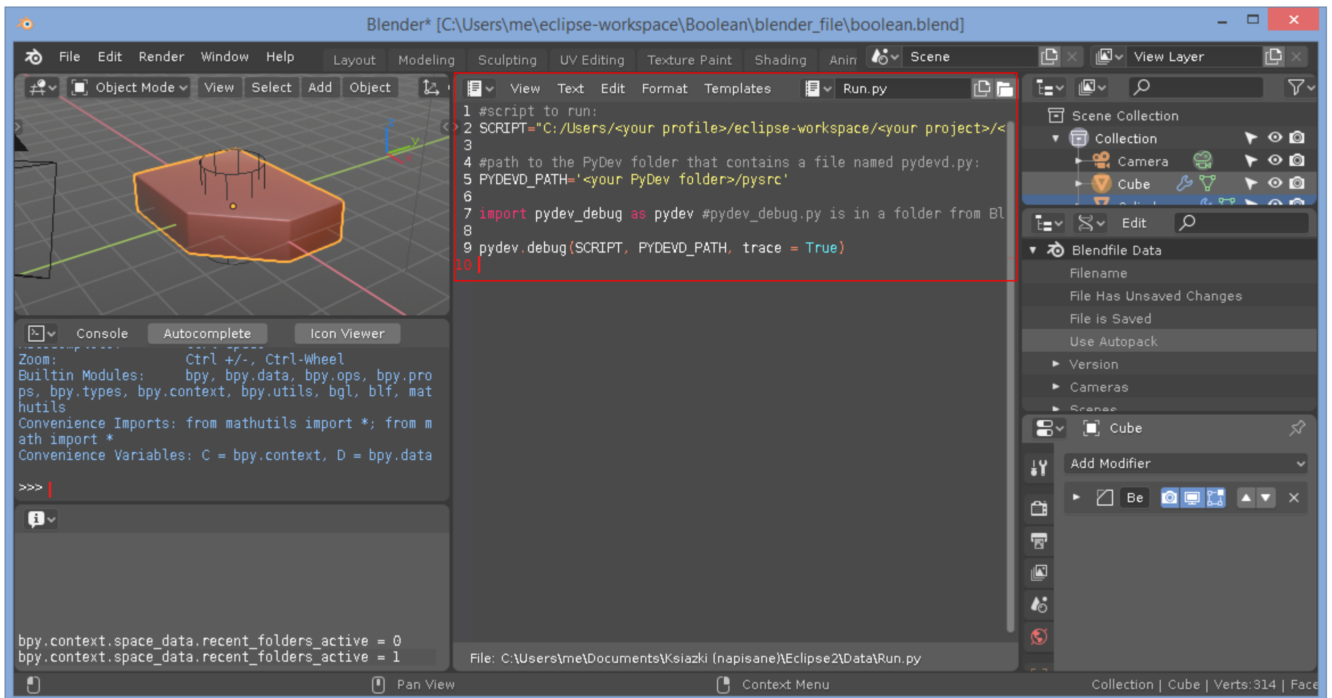
- Przygotowaliśmy w Blenderze środowisko testowe dla skryptu. To plik *booleans.blend*. Zawiera dwa obiekty – **Cube** i **Cylinder**. Do pracy nad skryptem wykorzystujemy standardową zakładkę (*workspace*) o nazwie *Scripting* (str. 47);
- Testowy plik Blendera wygodnie jest umieścić w projekcie Eclipse (str. 47);
- Wszystkie polecenia Blendera, które wywołujesz, są wyświetlane w oknie dziennika operacji (*Operations Log*) jako odpowiedni kod Pythona (str. 48).
- *Operations Log* nie wyświetla nowej linii kodu gdy wskażesz obiekt za pomocą myszki. Gdy jednak tego samego przypisania dokonasz poprzez wpisanie nazwy obiektu – w oknie dziennika pojawi się odpowiednie wyrażenie Pythona (str. 48, 49);
- Zapisy z *Operations Log* są doskonałym źródłem informacji. W prostym przypadku, takim jak nasz, kluczowe linie skryptu „piszą się same". Wystarczy je potem skopiować do schowka i wkleić do Eclipse (str. 50);
- Pliki nagłówków Blender API (***bpy*** i pozostałe) pozwalają od razu podczas pracy w IDE wykryć i poprawić wiele błędów wynikających z braku lub niewłaściwie zastosowanych definicji (str. 50). Przed pierwszym uruchomieniem skryptu w Blenderze należy wyeliminować z kodu wszystkie problemy zaznaczone przez PyDev;
- Podstawowymi źródłami informacji dla skryptu Blendera są dwa obiekty:
  - ***bpy.data***, udostępniający wszystkie dane z aktualnego pliku Blendera;
  - ***bpy.context***, udostępniający informacje o zaznaczonych obiektach i środowisku działania skryptu;

  W Blender API dostęp do danych odbywa się poprzez listy, indeksowane nazwą elementu (str. 49);

3.4 Uruchamianie skryptu w Blenderze

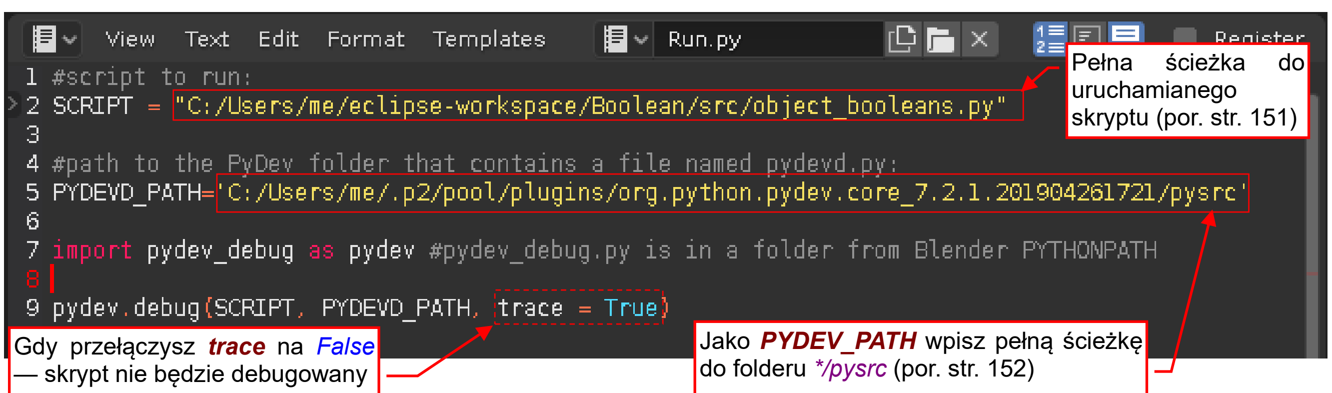
W poprzedniej sekcji napisaliśmy pierwszy kawałek skryptu, który powinien zadziałać w Blenderze. Można byłoby to zrobić „metodą tradycyjną”: otworzyć w oknie *Text Editor* Blendera, uruchomić. Tyle, że w ten sposób nie można śledzić go w debugerze. W dodatku w projekt wkrada się zamieszanie. (Gdybyś coś zmienił w kodzie skryptu w Blenderze, musiałbyś pamiętać, aby tę zmodyfikowaną wersję zapisać z powrotem na dysk).

Proponuję inne, wygodniejsze rozwiązanie. Otwórz w edytorze tekstu naszego testowego środowiska Blendera plik *Run.py*, dostarczony wraz z tą książką (por. str. 39) (Rysunek 3.4.1):



Rysunek 3.4.1 Dodanie do pliku tekstowego kodu *Run.py*

Ten plik zawiera kilka prostych linii kodu. Aby zaadaptować go do naszego projektu, zmień wartości przypisane do zmiennych ***SCRIPT*** i ***PYDEV_PATH***, umieszczonych na początku skryptu (Rysunek 3.4.2):



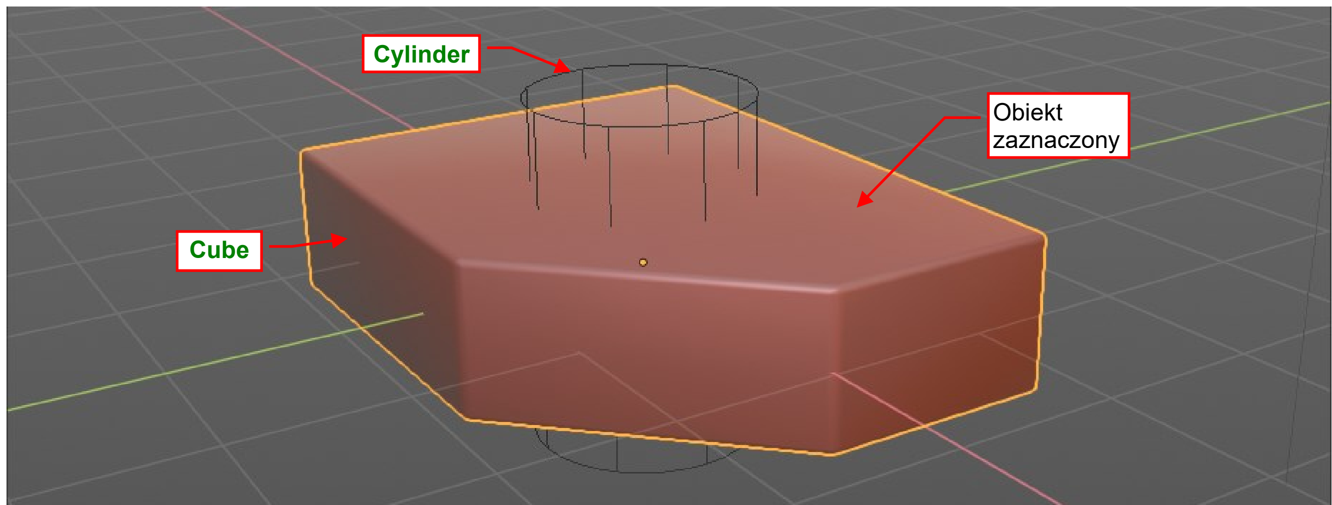
Rysunek 3.4.2 Adaptacja kodu *Run.py* dla projektu

Zmienna ***SCRIPT*** powinna zawierać pełną ścieżkę do pliku skryptu (szczegóły jak ją znaleźć – patrz str. 151).

Zmienna ***PYDEV_PATH*** to ścieżka do jednego z folderów PyDev, o nazwie *pysrc*. Ten folder zawiera moduł *pydevd.py*. (To biblioteka PyDev z kodem dla klienta zdalnego debuggera — por. str. 149, 160). Foldery aktualnej wersji PyDev znajdziesz w profilu użytkownika (na ilustracji to *C:/Users/me/*), w podkatalogu *.p2/pool/plugins*. Jednak w przyszłych wersjach PyDev to położenie może się zmienić, więc w razie potrzeby zerknij na str. 152, gdzie opisuję, jak możesz ten folder odnaleźć.

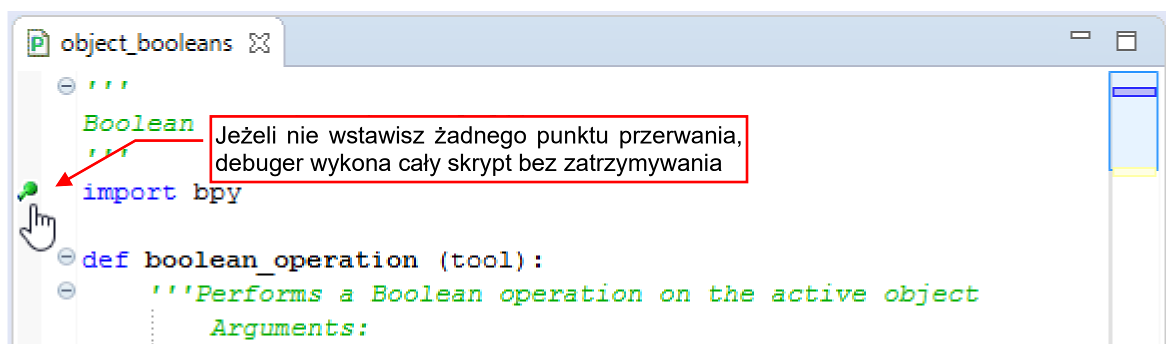
Copyright Witold Jaworski, 2011-2019.

Upewnij się także, że model jest przygotowany do testów. Na razie nasz kod zakłada, że obiekt **Cube** jest obiektem aktywnym. Dlatego go teraz zaznaczam (Rysunek 3.4.3):



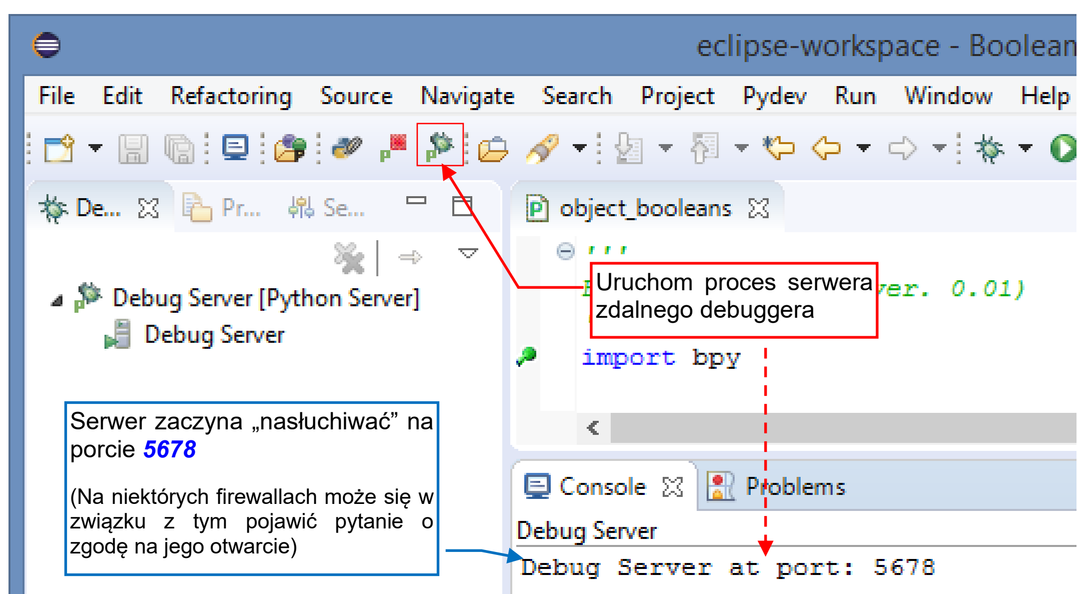
Rysunek 3.4.3 Przygotowanie danych do testu — wybrałem obiekt Cube (aby stał się obiektem aktywnym).

Wstaw w skrypt przynajmniej jeden punkt przerwania w miejscu, gdzie chcesz zacząć debugowanie. W naszym przypadku dodajmy go na sam początek kodu (Rysunek 3.4.4):



Rysunek 3.4.4 Umieszczenie punkt przerwania gdzieś na początku kodu

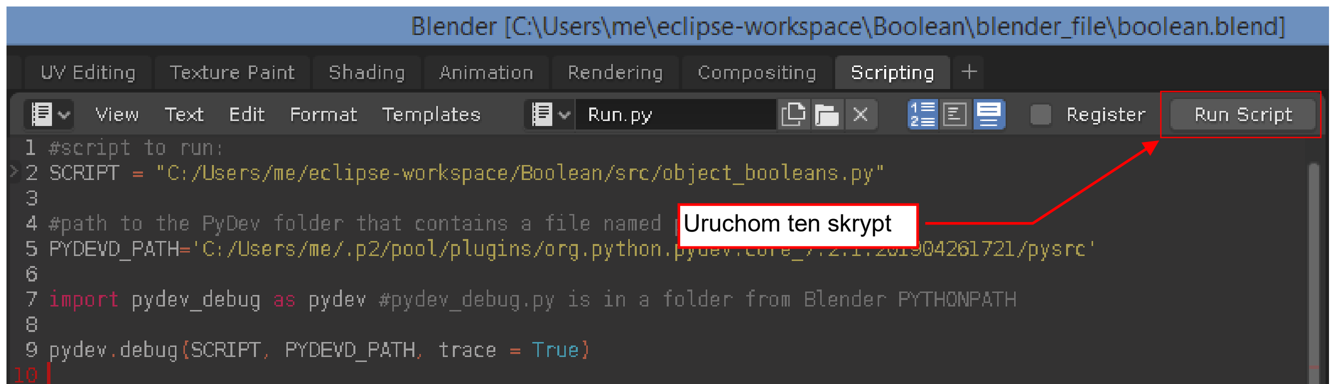
W perspektywie *Debug* uruchom serwera zdalnego debugera Pythona (por. str. 149) (Rysunek 3.4.5):



Rysunek 3.4.5 Uruchomienie serwera zdalnego debugera

(Szczegóły – zobacz str. 153, a gdy w ogóle nie możesz znaleźć tego przycisku – patrz str. 149 i 150).

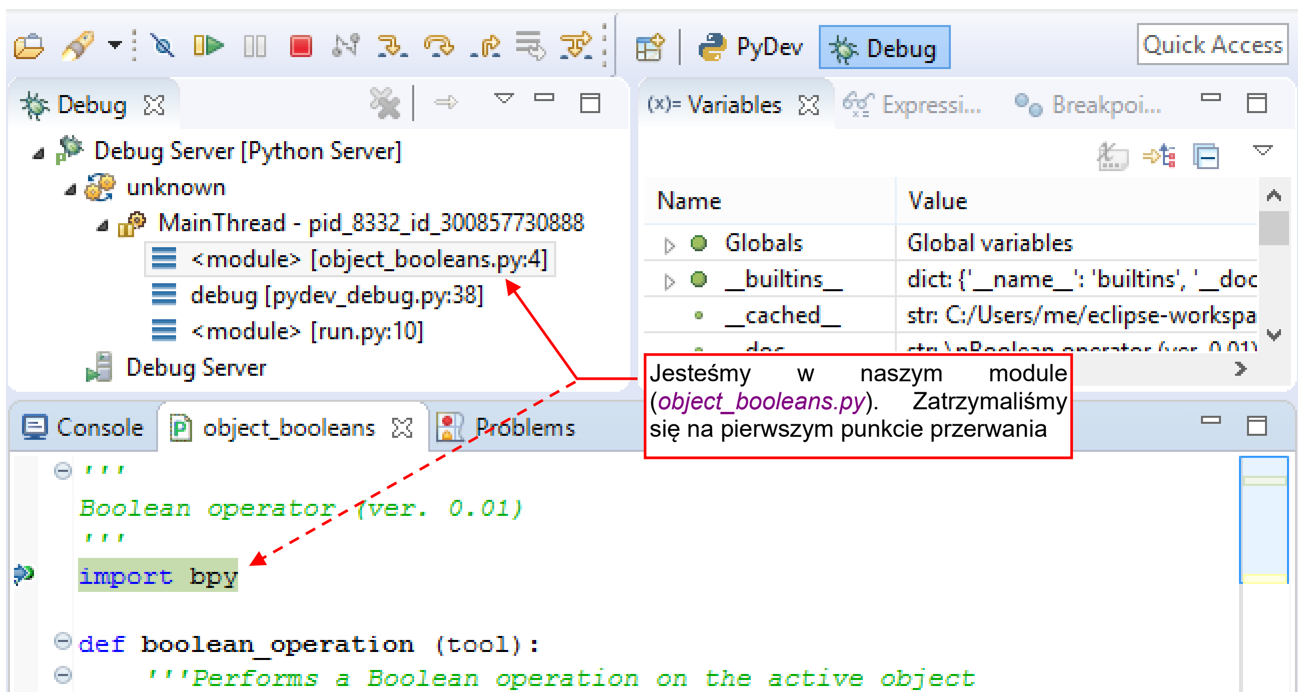
Gdy serwer debuggera już działa, możesz uruchomić skrypt w Blenderze. Kliknij w umieszczony po lewej stronie nagłówka edytora z kodem *Run.py* przycisk ***Run Script*** (Rysunek 3.4.6):



Rysunek 3.4.6 Uruchomienie skryptu w Blenderze

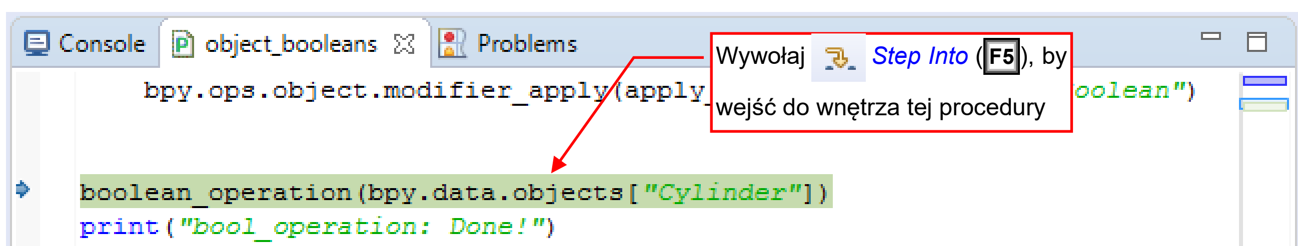
Ten kod ładuje aktualną wersję skryptu podanego w zmiennej ***SCRIPT*** i przekazuje ją do debugera. Od tej chwili aż do zakończenia debugowania ekran Blendera ulega „zamrożeniu". Nic się na nim nie zmienia, a kursor myszki pokazuje stan „czekaj".

Kliknij teraz w okno Eclipse. Po kilku sekundach okna debugera „ożyją". Linia wykonywania zatrzyma się na pierwszym punkcie przerwania (por. str. 54) (Rysunek 3.4.7):



Rysunek 3.4.7 Początek śledzenia skryptu

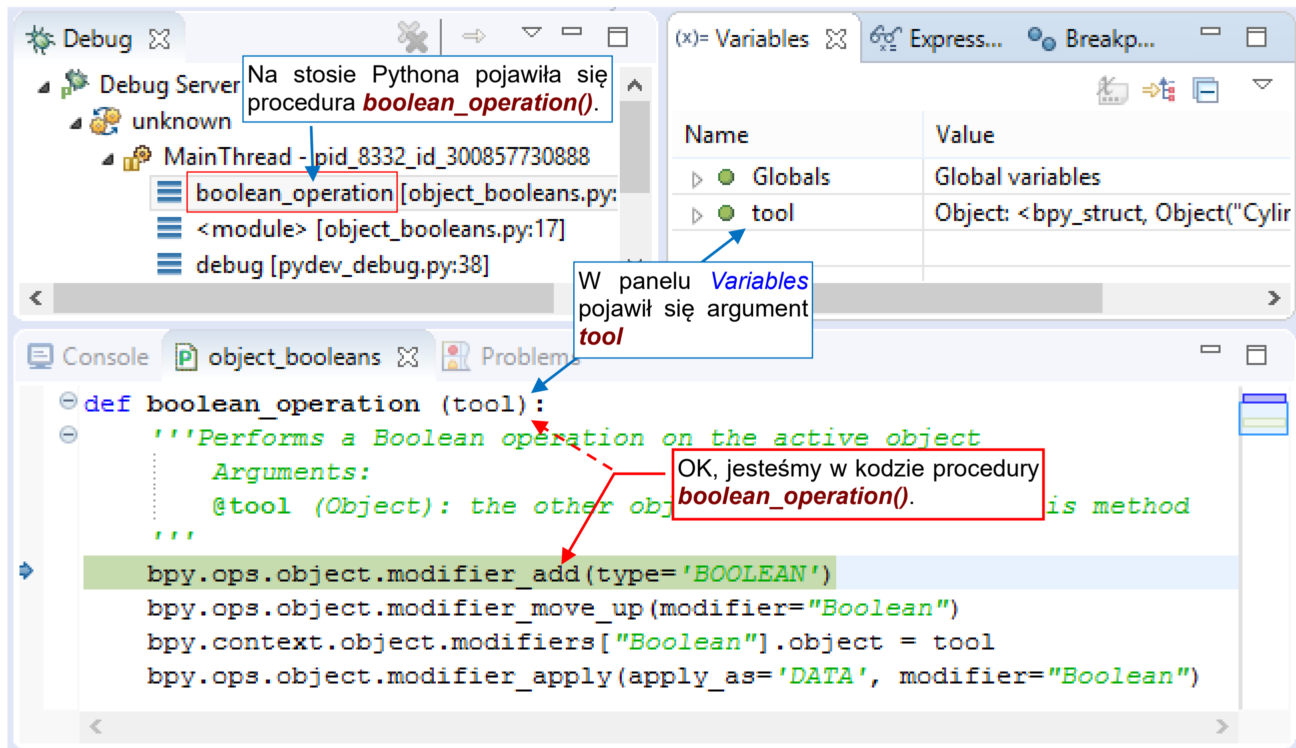
Przejdź przez tę i następne linie głównego kodu (*Step Over* — **F6**), dopóki nie dotrzesz do wywołania procedury ***boolean_operation()*** (Rysunek 3.4.8). Wtedy naciśnij **F5** (*Step Into*), by ją wykonać krok po kroku:



Rysunek 3.4.8 Kolejny krok — „wchodzimy" do wnętrza procedury *boolean_operation()*

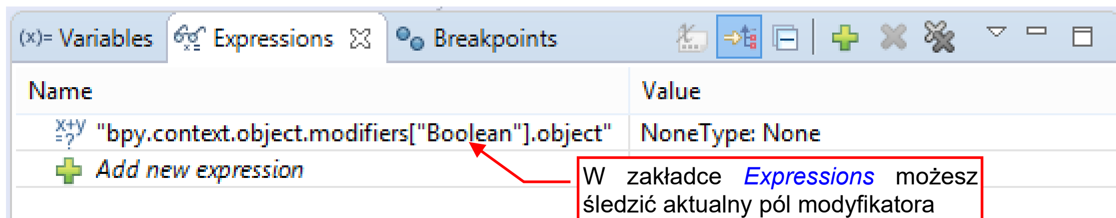
Copyright Witold Jaworski, 2011-2019.

Przejdziemy wówczas do pierwszej linii procedury ***boolean_operations()*** (Rysunek 3.4.9):



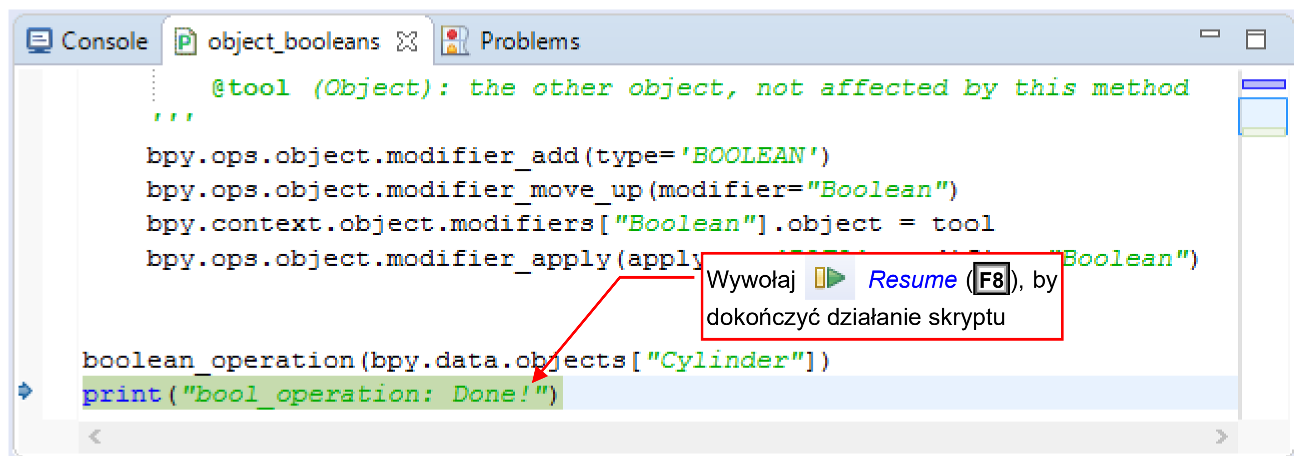
Rysunek 3.4.9 Śledzenie kodu wewnątrz procedury *boolean_operation()*

Do śledzenia wartości pól modyfikatora użyj panelu ***Expressions*** (Rysunek 3.4.10 — por. także str. 155):



Rysunek 3.4.10 Śledzenie wartości wybranych zmiennych w zakładce *Expressions*

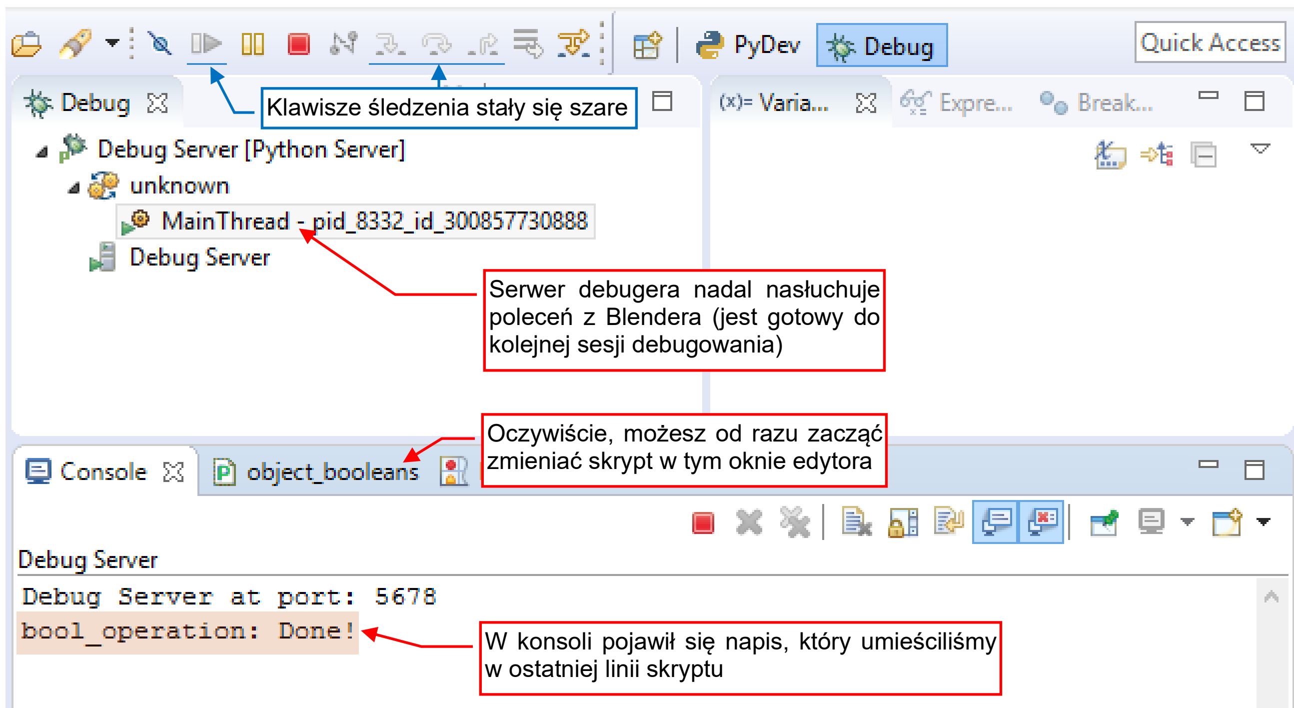
Gdy wykonasz całą procedurę, naciśnij przycisk *Resume*, by szybko dokończyć skrypt[1] (Rysunek 3.4.12):



Rysunek 3.4.11 Kończenie śledzenia skryptu

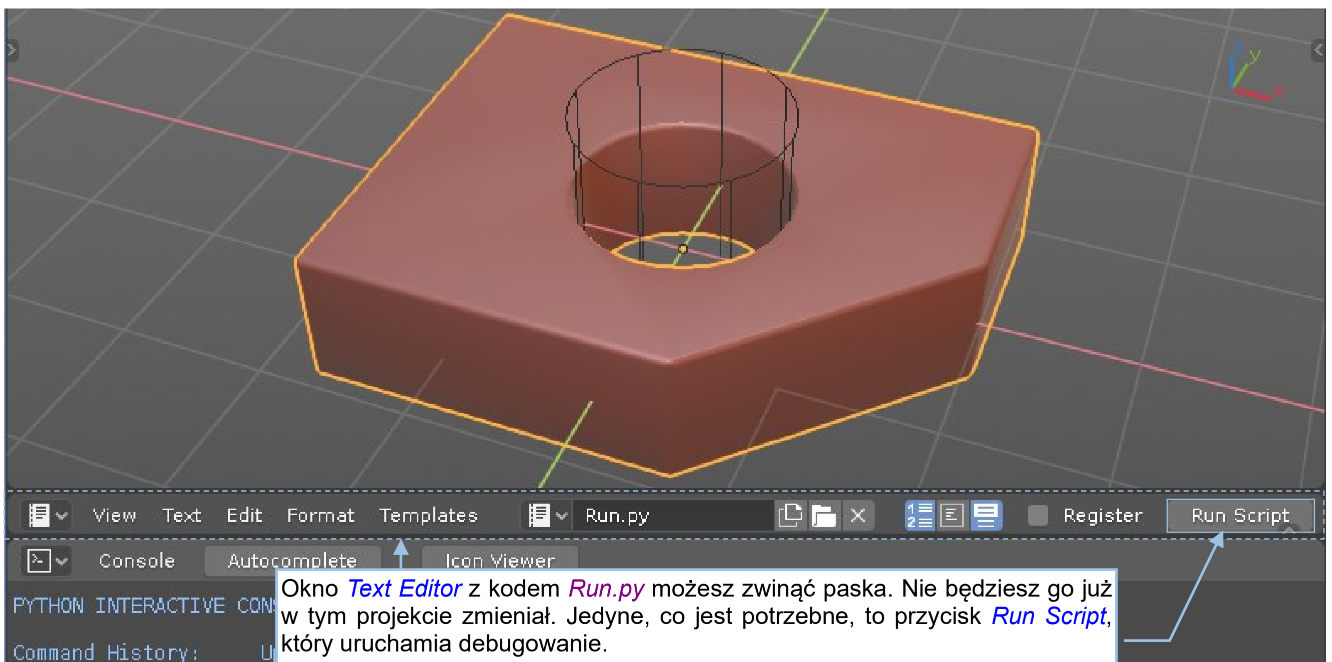
[1] Jeżeli na ostatniej linii naciśniesz *Step Over* (**F6**), to Python usunie ze stosu zakończony skrypt *object_booleans.py*. Linia debugera zatrzyma się wówczas na kolejnej linii pomocniczego skryptu, którego *Run.py* użył do załadowania kodu (por. str. 158). Nie mamy tam nic do śledzenia, więc dlatego w tym momencie radzę poleceniem *Resume* wykonać do końca kod uruchomiony w Blenderze.

W konsoli pojawi się napis, który przygotowałem w ostatniej linii skryptu (Rysunek 3.4.12):



Rysunek 3.4.12 Stan środowiska po naciśnięciu kończącego *Resume*

Nasz kod nie zakończył się, póki co, żadnym błędem. Popatrzmy na nasz testowy obiekt (Rysunek 3.4.13):



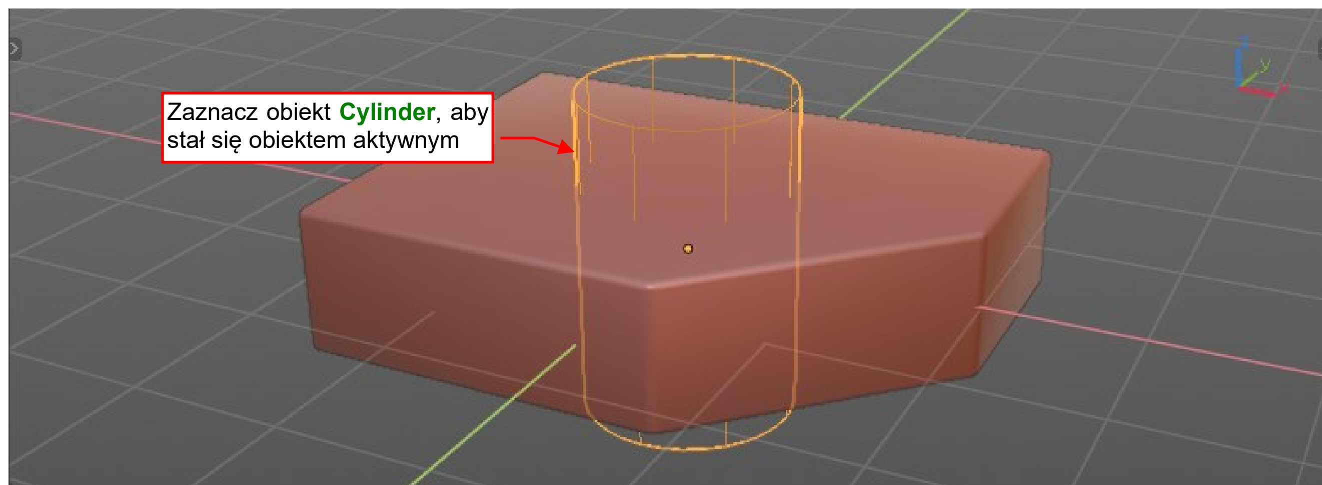
Rysunek 3.4.13 Rezultat działania naszego skryptu — poprawnie wykonany otwór w siatce

W siatce obiektu **Cube** powstał otwór. Wygląda więc na to, że nasz kod działa. Aby jego następne wywołanie mogło użyć tych samych danych testowych, wycofaj teraz efekty działania tego skryptu. Wystarczy w tym celu wywołać pojedyncze polecenie *Edit→Undo* (lub Ctrl-Z).

Kodu w edytorze tekstu Blendera nie będziemy już zmieniać. Wystarczy nam tylko dostęp do przycisku *Run Script*, by przeładować i uruchomić skrypt *object_booleans.py* po każdej kolejnej modyfikacji, wykonanej w Eclipse. Dlatego proponuję zmniejszyć okno *Text Editor* z kodem *Run.py* paska (tak jak to pokazuje Rysunek 3.4.13) i zapisać tak zmodyfikowany plik *boolean.blend*. Następnym razem Blender otworzy ten plik dokładnie z takim samym układem ekranu jak przy zapisie. To cecha bardzo ułatwiająca testowanie!

Copyright Witold Jaworski, 2011-2019.

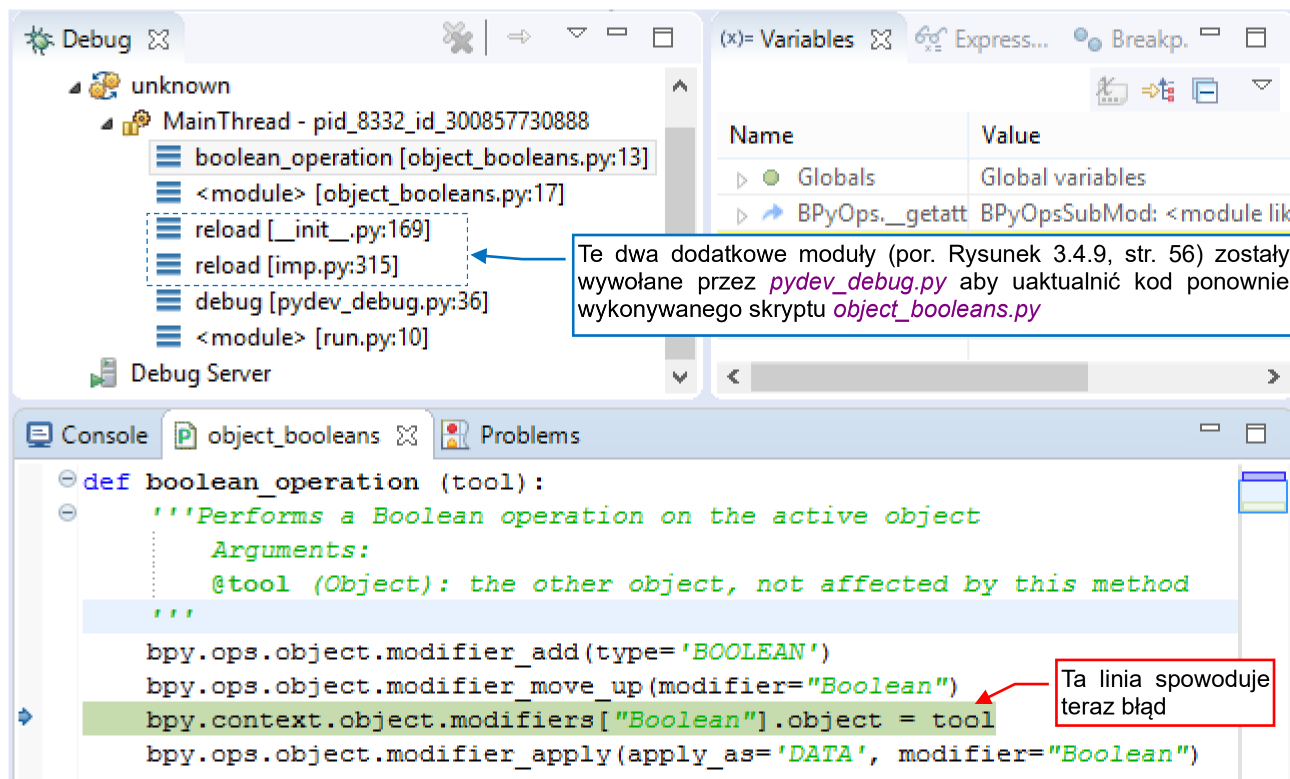
A jak kończy się debugowanie, gdy w skrypcie wystąpi błąd? Aby to sprawdzić, zaznacz obiekt **Cylinder** (aby stał się obiektem aktywnym):



Rysunek 3.4.14 Przygotowanie innego zestawu danych

Teraz, aby powtórnie zacząć debugowanie znajdującego się w Eclipse skryptu, po prostu naciśnij jeszcze raz w oknie *Text Editor* Blendera przycisk ***Run Script***. Dopóki masz włączony w Eclipse serwer zdalnego debugera, dopóty reszta zrobi się sama[1].

Znajdziesz się ponownie w pierwszym punkcie przerwania, tym, który pokazuje Rysunek 3.4.7 (str. 55). Tak jak poprzednio, wykonaj kolejne linie i przejdź do wnętrza procedury ***boolean_operation()***. Zatrzymaj się na linii, która przypisuje modyfikatorowi **Boolean** obiekt ***tool*** (Rysunek 3.4.15):

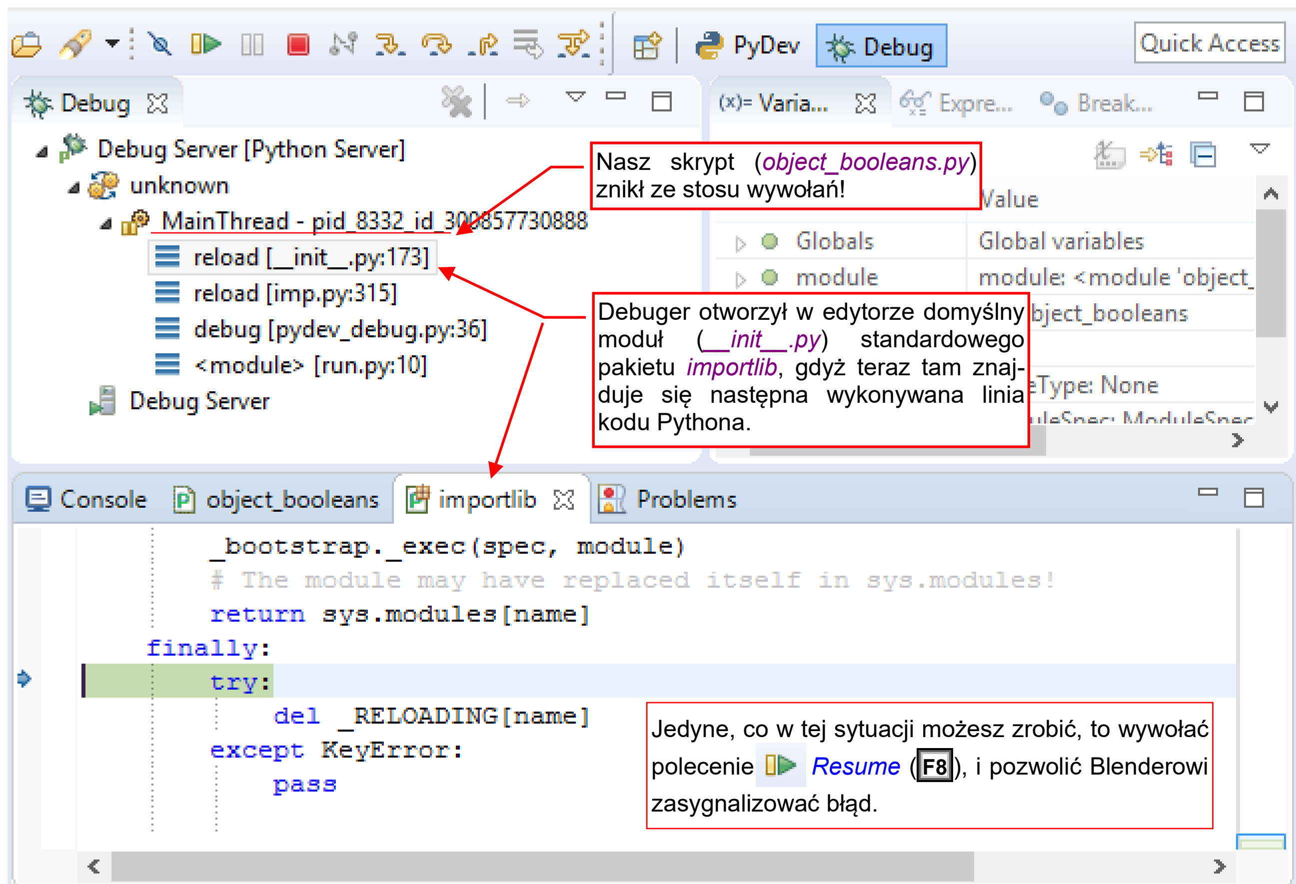


Rysunek 3.4.15 Linia, w której przypisujemy do modyfikatora obiekt

Linia, którą teraz masz podświetloną, spowoduje błąd skryptu.

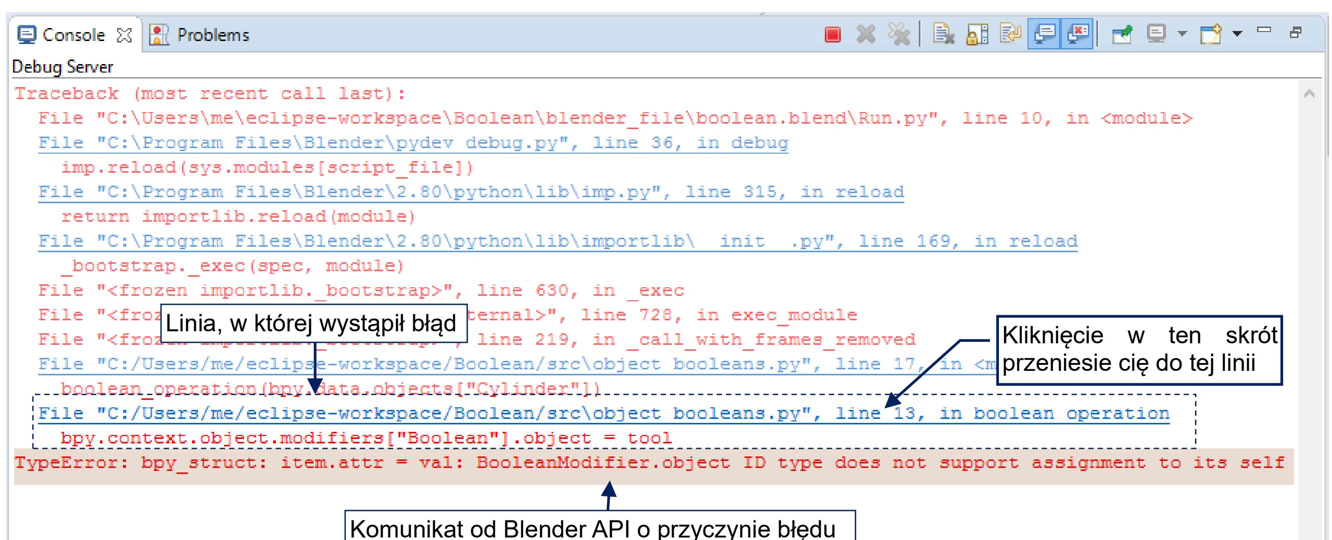
[1] Kod *Run.py*, który uruchamiasz przyciskiem *Run Script*, zadba także o to, by została załadowana najbardziej aktualna wersja Twojego skryptu. (Ta wyświetlana w edytorze Eclipse, w której przed chwilą wprowadziłeś ostatnia zmianę).

Naciśnij więc *Step Over* (F6), i zobacz, co się stanie (Rysunek 3.4.16):



Rysunek 3.4.16 Stan debugera po wystąpieniu błędu w śledzonym skrypcie Blendera

W wyniku błędu ze stosu wywołań znikło nie tylko wywołanie procedury, ale i cały skrypt *object_booleans.py*, który ją zawierał. (Porównaj stos wywołań na tej ilustracji ze stosem pokazywanym przez Rysunek 3.4.15). Wraz z tym znikła szansa na sprawdzenie stanu zmiennych lokalnych w chwili, kiedy wystąpił błąd. Jedyne, co można teraz zrobić, to wywołać polecenie ***Resume*** (F8). Pozwoli to Blenderowi dokończyć skrypt i wyświetlić standardowy komunikat o błędzie w konsoli (Rysunek 3.4.17):



Rysunek 3.4.17 Komunikat o błędzie, wyświetlany w konsoli

Najbardziej istotne informacje znajdziesz w ostatnich wyświetlonych wierszach: miejsce, w którym wystąpił błąd, oraz komunikat od Blender API o przyczynie. W tym przypadku –skrypt dodał wcześniej modyfikator *Boolean* do obiektu **Cylinder**, a w tej linii próbował mu przypisać jako „narzędzie" ten sam obiekt **Cylinder**.

Copyright Witold Jaworski, 2011-2019.

- Błąd (wyjątek) podczas wykonywania kodu zwykle zaskakują nas znienacka. Aby być w stanie sprawdzić stan zmiennych lokalnych w procedurze, w której wystąpił, musisz wcześniej cały jej kod ująć w wyrażenie ***try:*** … ***finally:***.

Pamiętaj, aby proces serwera zdalnego debugera wyłączać w Eclipse dopiero po zamknięciu Blendera.

- Najlepiej po prostu raz uruchomionego procesu serwera zdalnego debugera nigdy nie zamykać. Zakończy się sam wtedy, gdy zamkniesz Eclipse.

Pomocniczy moduł *pydev_debug.py* (por. str. 160), z którego korzystamy w tej sekcji, szuka źródła skryptu w kolejnych folderach wyliczonych w aktualnej **PYTHONPATH** Blendera. Wcześniej dodaje ścieżkę projektu na sam koniec tego wyliczenia. Wystarczy by np. poprzednia wersja Twojej wtyczki znalazła się w folderze wtyczek (w **PYTHONPATH** jest wymieniony wcześniej), aby to jej plik został załadowany i wykonany. Wówczas zaznaczone przez Ciebie punkty przerwania w ogóle nie działały, w skrypcie mogą się pojawić z powrotem usunięte przed chwilą błędy, a Ty nie będziesz wiedział, co się dzieje! Stąd sugestia:

- Nigdy nie używaj jako nazwy skryptu nazwy pliku, który znajdują się w którymś z folderów wyliczonych w **PYTHONPATH**. (Możesz sprawdzić jej zawartość w oknie *Python Console* Blendera: to ***sys.path***). W szczególności – nie stosuj nazw standardowych modułów Pythona!

Na przykład jeden ze standardowych modułów Pythona nosi nazwę ***test***, więc nigdy nie nazywaj swojego skryptu *test.py*. Jeżeli to zrobisz, to objaw będzie taki, że kod z *Run.py* będzie się w Blenderze wykonywał, ale nie będzie ładował Twojego pliku! (Straciłem w ten sposób trzy godziny, zanim się zorientowałem w czym problem).

Najbezpieczniej jest od razu nadawać skryptowi, który testujesz, nazwy w konwencji wtyczek Blendera, czyli takie z przedrostkiem trybu/okna: *object_**, *mesh_**, *uv_**, itd.

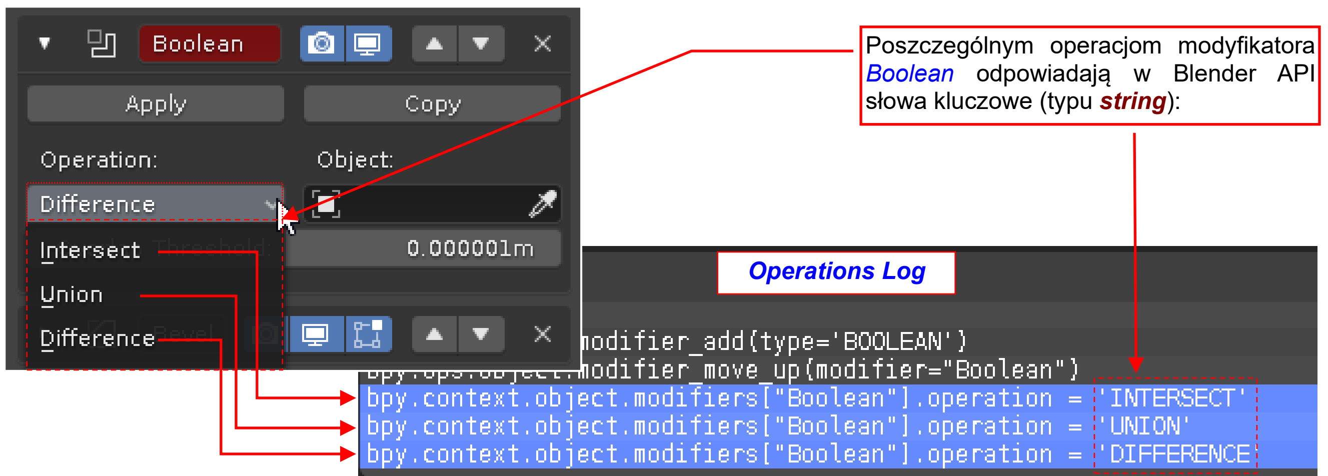
Podsumowanie

- Skrypt uruchamiamy za pomocą pomocniczego kodu *Run.py*, który należy umieścić w edytorze tekstu Blendera (str. 53);
- Przed pierwszym uruchomieniem, w kodzie *Run.py* należy wpisać odpowiednie ścieżki do pliku skryptu i debugera PyDev (str. 53).
- Przed uruchomieniem debugera ustaw w kodzie, który chcesz śledzić co najmniej jeden punkt przerwania. Zrób to w miejscu, od którego chcesz rozpocząć śledzenie (str. 53);
- Przy pierwszym uruchomieniu debugera należy włączyć w Eclipse ***PyDev Debug Server*** (str. 54). Następnie w oknie *Text Editor* Blendera, zawierającym skrypt *Run.py*, kliknąć w przycisk ***Run Script*** (str. 55);
- Do każdego następnego wywołania debugera wystarczy już tylko kliknięcie w przycisk ***Run Script*** (str. 57, 58);
- Do śledzenia zmian wybranych właściwości obiektów używaj okna *Expressions* (str. 56);
- Ostatnią linię skryptu wykonuj polecaniem *Resume* (**F8**), jak na str. 56. Jeżeli o tym zapomnisz, Eclipse otworzy okno z pomocniczym kodem, który jest używany przez *Run.py* do załadowania kodu skryptu. (Zazwyczaj, nie ma tam czego szukać);
- Gdy w Twoim skrypcie wystąpi wyjątek (błąd) – debuger zamknie i usunie ze stosu wywołań cały plik. Jednocześnie otworzy okno z jakimś standardowym kodem Pythona lub Blendera. Dokończ wtedy wykonywanie skryptu poleceniem *Resume* (**F8**), i sprawdź w konsoli komunikat o błędzie (str. 59).

3.5 Ulepszanie skryptu

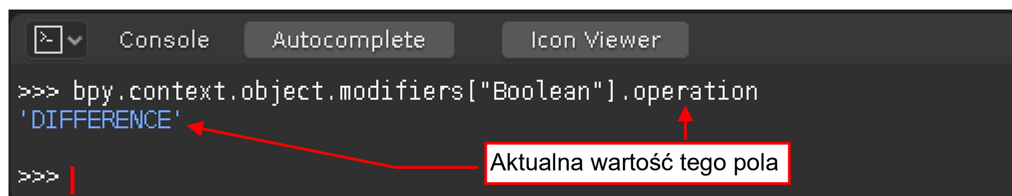
Nasz skrypt powstał z kopii zapisu poleceń Blendera i, jak pokazałem pod koniec poprzedniej sekcji, działa poprawnie tylko dla specyficznej konfiguracji testowej (por. str. 59). W tej sekcji przekształcimy go tak, by działał poprawnie na dowolnych obiektach, wskazanych przez użytkownika. Uczynimy go także bardziej uniwersalnym, aby w parametrach wywołania można było podać wartości, odpowiadające opcjom modyfikatora *Boolean*.

Zacznijmy może od wyboru rodzaju operacji: gdy wybierzesz każdą z nich, zobaczysz w oknie dziennika odpowiadające im wywołania w Pythonie (Rysunek 3.5.1):



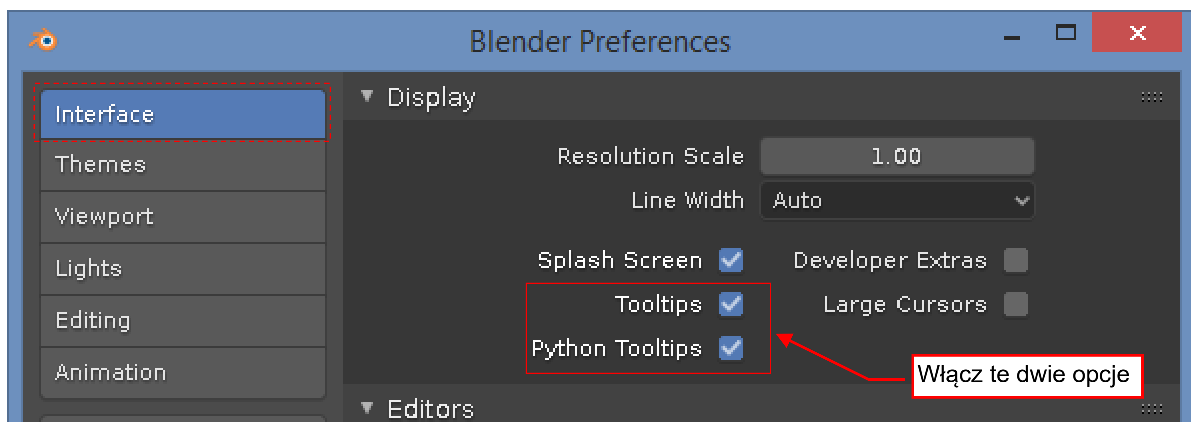
Rysunek 3.5.1 Zmiana opcji modyfikatora *Boolean* w Blender API

Okazuje się, że takim wyliczeniom odpowiadają w Blender API stałe wyrażenia tekstowe. Tryb działania modyfikatora zmieniamy, przypisując polu ***operation*** jedną z trzech dopuszczalnych wartości - **'INTERSECT'**, **'UNION'**, lub **'DIFFERENCE'**. Oczywiście, jeżeli to jest potrzebne, możemy także odczytać wartość tego pola – chociażby w konsoli Pythona (Rysunek 3.5.2):



Rysunek 3.5.2 Odczytanie wartości opcji modyfikatora

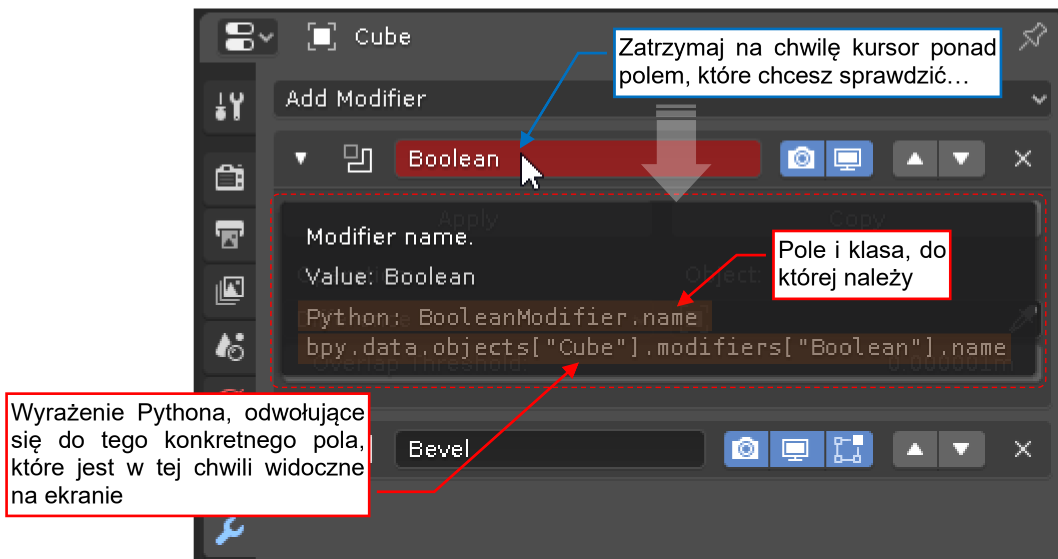
Do znajdowania nazw Blender API dla pól, które nie są typem wyliczeniowym, przydatne jest włączenie w ustawieniach Blendera (*Edit→Preferences*) opcji ***Python Tooltips*** (Rysunek 3.5.3):



Rysunek 3.5.3 Włączenie w Blenderze podpowiedzi wyrażeń API (do elementów interfejsu użytkownika)

Copyright Witold Jaworski, 2011-2019.

Od tej chwili wystarczy zatrzymać myszkę nad jakimś polem ekranu, by zobaczyć informację o jej typie i wywołaniu w Blender API (Rysunek 3.5.4):



Rysunek 3.5.4 Podpowiedzi wyrażeń API

Nazwa modyfikatora (w tym przypadku **Boolean**) jest unikalna na liście modyfikatorów pojedynczego obiektu[1]. Zwróć uwagę, że Blender API wykorzystuje ją jako indeks, a także np. w wyrażeniu API odpowiadającemu przyciskowi *Apply* (por. np. str. 49, Rysunek 3.3.7). To spostrzeżenie przyda się w nowej wersji procedury ***boolean_operation()***. Wprowadziłem w niej szereg ulepszeń (Rysunek 3.5.5):

```
import bpy

def boolean_operation (tool, op, apply=True):
    '''Performs a Boolean operation on the active object
       Arguments:
       @tool (Object): the other object, not affected by this method
       @op (Enum): a Boolean operation: {'UNION', 'INTERSECT', 'DIFFERENCE'}
       @apply (bool): apply results to the mesh (optional)
    '''
    obj = bpy.context.object

    bpy.ops.object.modifier_add(type='BOOLEAN')#adds new modifier to obj

    mod = obj.modifiers[-1]

    while obj.modifiers[0] != mod:
        bpy.ops.object.modifier_move_up(modifier=mod.name)

    mod.operation = op #set the operation
    mod.object = tool #activate the modifier

    if apply: #applies modifier results to the mesh of the active object (obj):
        bpy.ops.object.modifier_apply(apply_as='DATA', modifier=mod.name)
```

Dodatkowe argumenty: ***op***, ***apply***

obj to pomocniczy skrót do aktywnego obiektu

mod to nasz modyfikator (operator zawsze dodaje nowy na koniec listy)

Pętla przenosząca nowo dodany modyfikator na początek listy

Zamiast stałego tekstu używam nazwy ***mod***

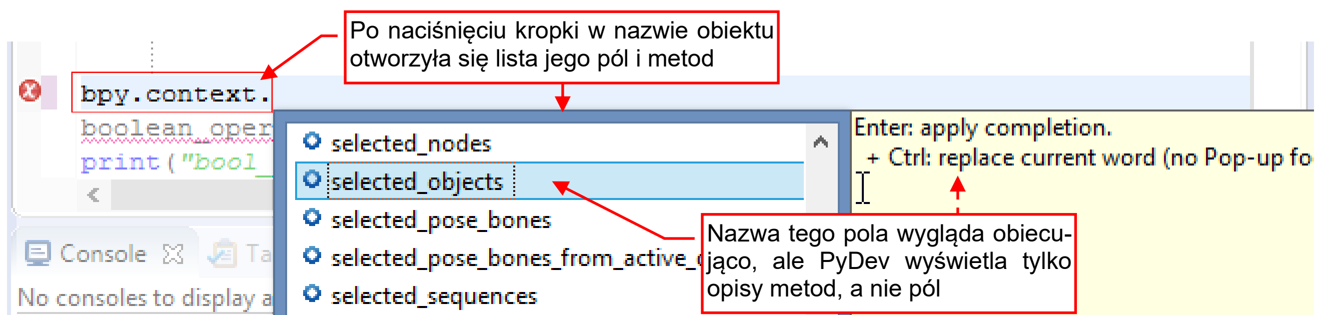
Rysunek 3.5.5 Poprawiona obsługa modyfikatorów w procedurze *boolean_operation()*

Rozbudowałem listę parametrów tej metody o dwa nowe argumenty: ***op*** pozwala określić rodzaj operacji, a opcjonalna flaga ***apply*** umożliwia pozostawienie rezultatu w dynamicznej postaci (modyfikatora). Aby skrócić odwołania w kodzie procedury, przypisałem referencję do aktywnego obiektu lokalnej zmiennej ***obj***.

[1] W czasie dodawania nowego modyfikatora Blender sam tworzy odpowiednią nazwę: na przykład gdybym do stosu modyfikatorów pokazywanych przez Rysunek 3.5.4 dodał kolejny modyfikator typu *Boolean*, to otrzymałby nazwę **Boolean.001**. Gdy użytkownik zmienia nazwę modyfikatora – Blender nie pozwala mu wpisać nazwy, która już istnieje na liście.

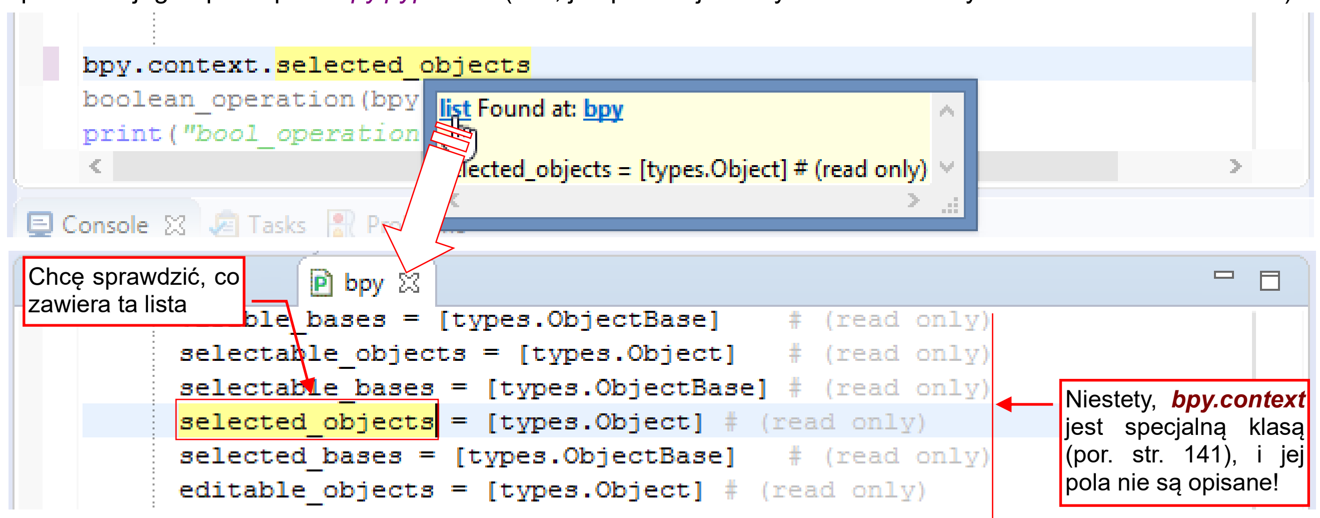
W drugiej lokalnej zmiennej ***mod*** zapamiętuję referencję do modyfikatora, który dodałem do obiektu. Wiem, gdzie go szukać, bo operator ***object.modifier_add()*** zawsze dodaje nowy element na koniec listy modyfikatorów obiektu[1]. Po dodaniu modyfikatora ***mod*** stos (lista) modyfikatorów może mieć jeden lub więcej elementów. Dlatego zastąpiłem pojedyncze wywołanie operatora ***object.modifier_move_up()*** pętlą, która „przesuwa w górę" operator ***mod***, dopóki nie stanie się pierwszym. W związku z tym, że Blender sam ustala nazwy nowych modyfikatorów, nie mam żadnej gwarancji, że element typu *Boolean* otrzyma po dodaniu nazwę **Boolean**. Równie dobrze może to być np. **Boolean.001**, gdy aktywny obiekt używa już jakiegoś innego modyfikatora tego typu. Dlatego zamiast stałego tekstu **'Boolean'** (który figurował w linii skopiowanej z okna dziennika) przekazuję teraz operatorom ***modifier_move_up()*** i ***modifier_apply ()*** wartość pola ***mod.name***.
Opcjonalny argument ***apply*** dodałem do procedury na wszelki wypadek. Domyślnie jest równy *True*, co powoduje, że rezultat działania modyfikatora ***mod*** jest utrwalany w siatce aktywnego obiektu. (Tak, jak to przetestowaliśmy w poprzednich sekcjach). Jeżeli jednak nadasz mu w wywołaniu tej procedury wartość *False*, to modyfikator nie zostanie usunięty, a rezultat pozostanie „dynamiczny". (W ten sposób użytkownik może także użyć tego skryptu jako szybszej metody dodawania modyfikatorów *Booelan*).

Spróbujmy teraz zmienić w głównym kodzie programu wywołanie procedury ***boolean_operation()*** tak, aby wykorzystać jako „narzędzia" pozostałe obiekty wskazane przez użytkownika. Stąd pierwsze pytanie: jak uzyskać listę aktualnie zaznaczonych obiektów? W oknie dziennika jej nie widać, gdyż każdy z operatorów korzysta z niej domyślnie. Sądzę, że podobnie jak obiekt aktywny, powinna być udostępniona przez obiekt ***bpy.context***. Przyjrzyjmy się więc zawartości tego obiektu - np. używając okna autokompletacji (Rysunek 3.5.6):



Rysunek 3.5.6 Przeglądanie pól i metod klasy obiektu *bpy.context*.

Niestety, PyDev nie wyświetla opisów pól, więc spróbujmy umieścić wywołanie do ***.selected_objects*** w kodzie i sprawdzić jego opis w pliku *bpy.pypredef*. (Tak, jak pokazuje to Rysunek 3.2.7 i Rysunek 3.2.8 na str. 41 i 42):



Rysunek 3.5.7 Poszukiwanie opisu pola *bpy.context.selected_objects*.

[1] To założenie opieram na wieloletniej pracy z Blenderem – w oficjalnej dokumentacji brak opisu dodawania modyfikatora. (Często się w niej pomija tak trywialne operacje). Jeżeli chcesz być ostrożniejszy ode mnie – zachowaj kopię listy ***obj.modifiers*** przed wywołaniem ***object.modifier_add()***. Po dodaniu nowego modyfikatora znajdź w ***obj.modifiers*** element, którego nie ma w tej zachowanej liście.

Copyright Witold Jaworski, 2011-2019.

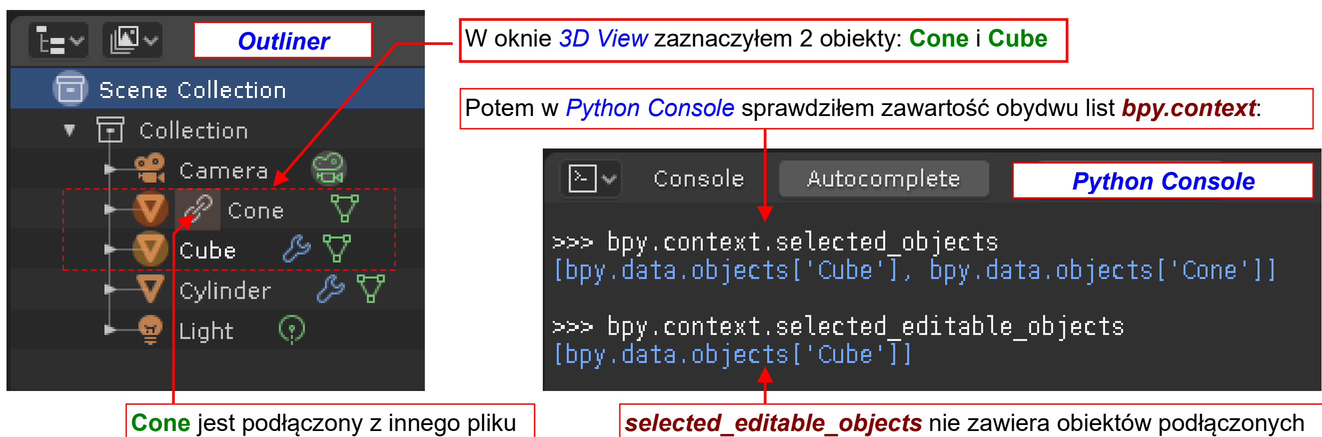
Obiekt ***bpy.context*** jest bardzo specyficzny, gdyż jego zawartość zależy od okna z którego został wywołany skrypt (por. str. 141). W dodatku jego pola nie mają opisów – także w oficjalnej dokumentacji Blender API (!). Można się oprzeć na znalezionych w wyszukiwarce uwagach innych użytkowników. Wynika z nich, że listą wybranych obiektów jest pole ***selected_objects***. Jednak w pliku nagłówków modułu ***bpy*** dostrzegłem jeszcze drugie pole warte sprawdzenia: ***selected_editable_objects*** (Rysunek 3.5.8):

```
*object_booleans    bpy
    visible_bases = [types.ObjectBase]     # (read only)
    selectable_objects = [types.Object]    # (read only)
    selectable_bases = [types.ObjectBase]  # (read only)
    selected_objects = [types.Object] # (read only)
    selected_bases = [types.ObjectBase]    # (read only)
    editable_objects = [types.Object]      # (read only)
    editable_bases = [types.ObjectBase]    # (read only)
    selected_editable_objects = [types.Object]     # (read only)
    selected_editable_bases = [types.ObjectBase]   # (read only)
```

Warto sprawdzić, czym się różnią te dwie listy

Rysunek 3.5.8 Pola *bpy.context*, **które chciałbym sprawdzić**

Chciałbym się dowiedzieć, czym się różnią[1]. Pierwsze próby na używanym dotychczas w tym projekcie pliku testowym ne wykazały żadnych różnic. Sprawa się wyjaśniła dopiero wówczas, gdy dołączyłem (poleceniem *File→Link*) do tej sceny obiekt o nazwie **Cone**, przechowywany w innym pliku Blendera[2] (Rysunek 3.5.9):



Rysunek 3.5.9 Różnice pomiędzy *selected_objects* **i** *selected_editable_objects*.

Okazuje się, że lista ***selected_editable_objects*** nie zawiera podłączonych obiektów – w tym przypadku obiektu **Cone**. (Istotnie, nie możesz zmieniać własności takich „referencji do obiektów –zapewne stąd nazwa tego pola). Sprawdziłem jednak, że taka referencja może być użyta jako „narzędzie" w modyfikatorze *Boolean*.

- W tym skrypcie listę wybranych obiektów będę odczytywał z pola ***bpy.context.selected_objects***.

Blender API udostępnia pole ***selected_objects*** tak samo, jak preferowane pole ***bpy.context.object*** – we wszystkich kontekstach ekranu. (Dokumentacja API opisuje to jako *Screen context*).

[1] To nie jest tylko czysta ciekawość. W Blenderze 2.5 okno dziennika pokazywało tylko wywołania operatorów i nie można było sprawdzić, w jak poprawnie się odwołać do aktywnego obiektu sceny. Także wówczas ***bpy.context*** nie był udokumentowany. Dlatego do odczytania aktywnego obiektu zdecydowałem się używać pola o najbardziej odpowiedniej nazwie: ***bpy.context.active_object***. Kilka miesięcy później próbowałem dołączyć mój skrypt do menu Blendera. Okazało się wówczas, że tak wywołany kod działa wówczas w innym kontekście, w którym obiekt ***bpy.context*** nie ma pola ***active_object***. Dopiero analiza kodu standardowych menu Blendera pozwoliła mi odkryć, że powinienem korzystać z pola o nazwie ***object***. (Kto by przypuszczał!). Dlatego teraz wolę się zawczasu upewnić, jakie pole powinienem używać w Blenderze 2.8 do uzyskania listy zaznaczonych obiektów. Swoją drogą – wstyd, że Blender Foundation od 8 lat nie udokumentowała tak ważnej części tego API!

[2] Blender traktuje każdy zapisany na dysku plik **.blend* jak „bibliotekę" wszelkich bloków danych (obiektów, siatek, materiałów, tekstur, węzłów, …) których referencje (*links*) możesz przyłączać do innych plików Blendera.

Po tych długich dywagacjach na temat źródła danych, możemy zmienić główny kod skryptu (Rysunek 3.5.10):

```
selected = list(bpy.context.selected_objects)
selected.remove(bpy.context.object)

for tool in selected:  #Apply each tool to the active object:
    boolean_operation(tool,'DIFFERENCE')

print("object_booleans.py: Done!")
```

selected: statyczna, "kopia robocza" tej listy

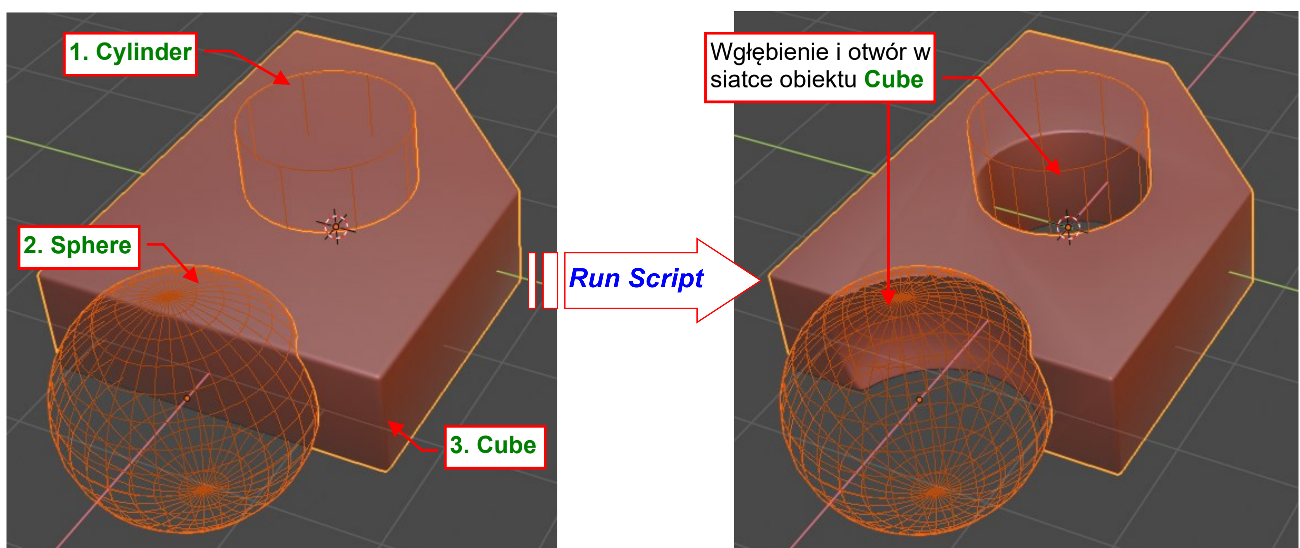
Usuń z "kopii roboczej" obiekt aktywny

Użyj tak "skróconej" listy ***selection*** jako źródła "obiektów – narzędzi"

Rysunek 3.5.10 Ulepszony główny kod skryptu

Aby wykonać zadaną operację Boole'a dla każdego obiektu spośród zaznaczonych przez użytkownika, musimy wykluczyć spośród nich obiekt aktywny. (Inaczej spowoduje to błąd – taki jak na str. 59). W tym celu kopiuję zawartość iteratora ***bpy.context.selected_objects*** do roboczej listy o nazwie ***selected***. Następnie usuwam z ***selected*** obiekt aktywny. Na koniec dla każdego obiektu z tak skróconej listy wywołuję procedurę ***boolean_operation()***.

Teraz warto sprawdzić, czy tak zmodyfikowany skrypt działa poprawnie. Dodałem do testowej sceny jeszcze jeden obiekt: sferę (**Sphere**). Następnie zaznaczyłem wszystkie trzy obiekty (w kolejności: **Cylinder**, **Sphere** i **Cube**) i uruchomiłem skrypt. (Uruchomiłem server debugera w Eclipse i kliknąłem przycisk *Run Script* w Blenderze – por. str. 54 i 55). Rysunek 3.5.11 pokazuje stan początkowy i rezultat:



Rysunek 3.5.11 Rezultaty działania zmodyfikowanego skryptu

Podczas działania skryptu nie wystąpił żaden błąd, a rezultat jest poprawny.

Wycofaj teraz zmiany dokonane przez ten skrypt (Ctrl-Z) i zaznacz znów cylinder i sferę, oraz jeszcze jeden obiekt: **Lamp** (por. schemat testowej sceny: Rysunek 3.5.9 na str. 64). Jako ostatni dołącz do selekcji **Cube** (aby był obiektem aktywnym). W skrypcie uruchomionym dla takich danych wejściowych wystąpi błąd:

Console ☒ Problems

Debug Server

```
  File "C:/Users/me/eclipse-workspace/Boolean/src\object booleans.py", li
    bpy.ops.object.modifier_apply(apply_as='DATA', modifier=mod.name)
  File "C:\Program Files\Blende
    ret = op_call(self.idname_py(), None, kw)
RuntimeError: Error: Modifier is disabled, skipping apply
```

Błąd wystąpił w ostatniej linia procedury ***boolean_operation()***

Nie można wywołać polecenia *Apply* dla nieaktywnego modyfikatora

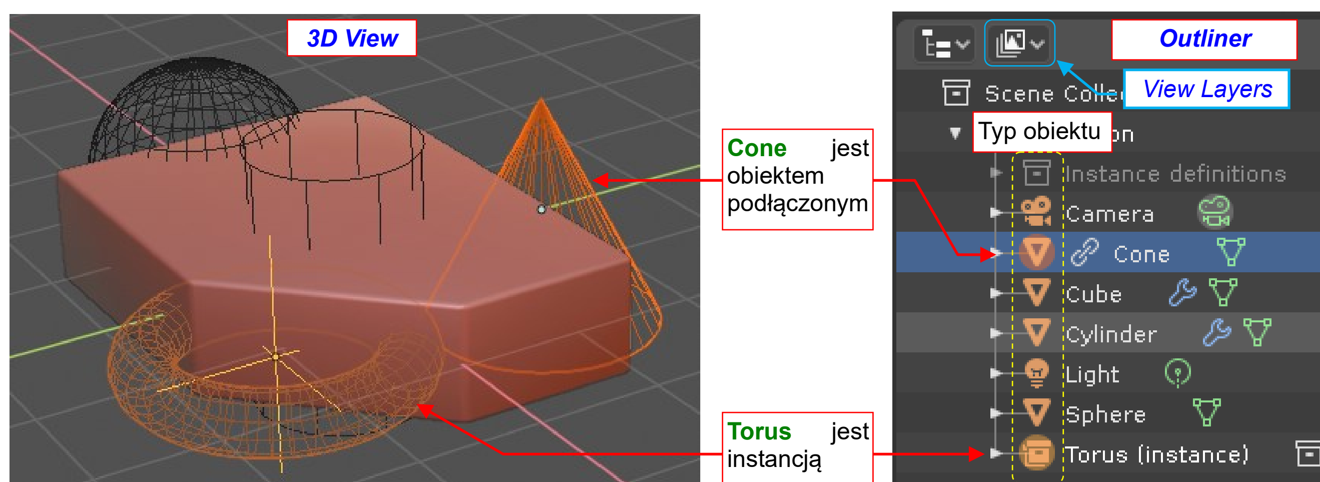
Rysunek 3.5.12 Błąd, który wystąpi po zaznaczeniu obiektu Lamp **(jako „narzędzia")**

Copyright Witold Jaworski, 2011-2019.

Co się stało? Gdy próbujesz przypisać do modyfikatora obiekt bez siatki – taki jak np. **Camera** czy **Lamp** – jest on ignorowany. W rezultacie nowy modyfikator *Boolean* dodany do aktywnego obiektu (**Cube**) nigdy nie stanie się aktywny. Dlatego przy próbie wywołania operatora ***modifier_apply()*** Blender zgłosi wyjatek (błąd wykonania).

Jak zapobiec takim sytuacjom? Można sprawdzać bezpośrednio przed wywołaniem operatora *Apply*, czy modyfikator jest aktywny. Jednak ten pomysł niezbyt mi się podoba. Wystarczy, że w przyszłości ulegnie zmianie jakiś wewnętrzny szczegół w Blender API, i taki błąd zostanie zasygnalizowan np. już przy próbie przypisania obiektu typu *Lamp* do modyfikatora *Boolean*. Dlatego lepiej w ogóle nie wywoływać procedury ***boolean_operations()*** dla niewłaściwego typu obiektów. I nie chodzi tu tylko o typ argumentu ***tool***: mogę sobie wyobrazić sytuacje, w których użytkownik przez pomyłkę wskaże **Lamp** jako obiekt aktywny.

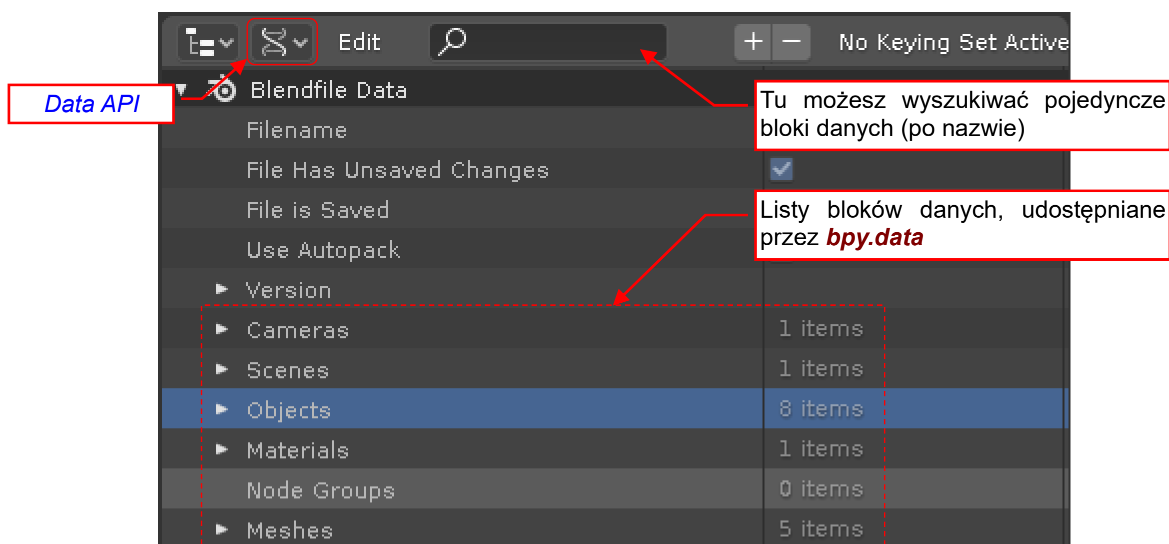
Aby dokładnie spradzić, co akceptuje modyfikator *Boolean*, rozbudowałem testową scenę o dwa dodatkowe obiekty: **Cone**, który jest podłączony (*linked*) z inneg pliku, oraz **Torus (instance)**, który jest instancją pewnej kolekcji. (Ta kolekcja jest ukryta i zawiera tylko pojedynczy obiekt **Torus**). Rysunek 3.5.13 przedstawia aktualny stan sceny testowej oraz jej strukturę w oknie *Outliner* (w układzie: *View Layer*):



Rysunek 3.5.13 Rozbudowa danych testowych

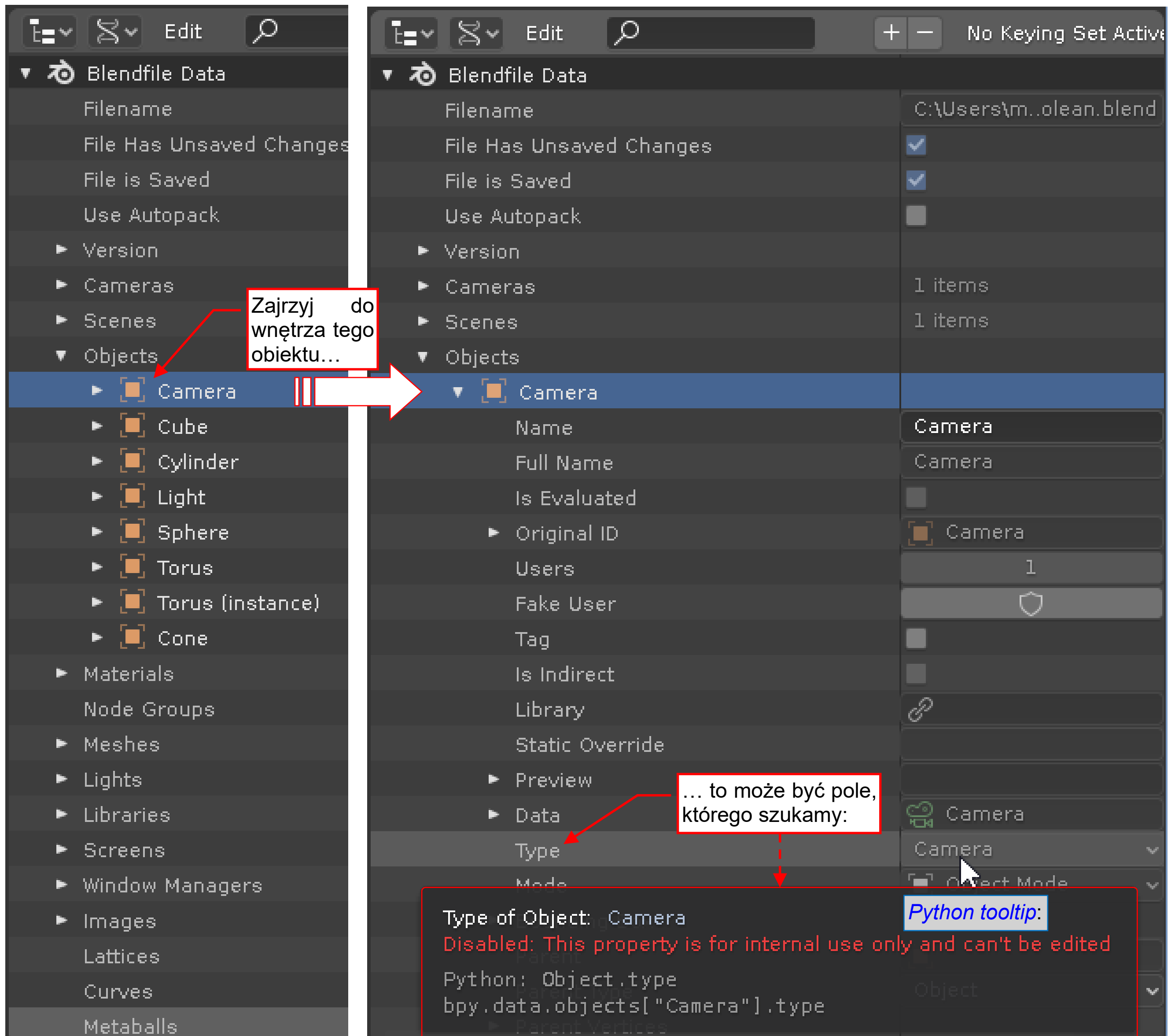
W oknie *Outliner* typ każdego z obiektów jest przedstawiony za pomocą ikony. Skrypt mógłby wykorzystać tę informację by sprawdzić, czy obiekt aktywny i wskazane narzędzie są odpowiednie do wykonania operacji. Ale z którego pola Blender API można odczytać typ obiektu?

Dla znalezienia odpowiedzi posłużyłem się oknem ***Outliner*** w układzie ***Data API*** (Rysunek 3.5.14):



Rysunek 3.5.14 Zawartość okna *Outliner* (w układzie *Data API*)

W trybie *Data API* okno *Outliner* okazuje całą zawartość pliku. To w zasadzie ładnie przedstawiona struktura ***bpy.data***. Spróbuj rozwinac kolekcję *Objects*, a zobaczysz w niej wszystkie obiekty. Można tu przejrzeć ich całą zawartość (Rysunek 3.5.15):



Rysunek 3.5.15 Przeglądanie szczegółów danych

Zacząłem więc przeglądać pola pierwszego obiektu, szukając czegoś, co by określało jego typ (w tym przypadku była to kamera). Szybko zauważyłem pozycję o obiecującej nazwie *Type* (= *Camera*). Najechałem na nią myszką, by Blender wyświetlił „chmurkę” z jej opisem. Opis wydaje się to potwierdzać – tym bardziej, że to pole jest „tylko do odczytu” (nie można go zmieniać). Używając przykładowego odwołania z *Python tooltip*, szybko sprawdziłem jego zawartość w konsoli Pythona (Rysunek 3.5.16):

Console Autocomplete Icon Viewer

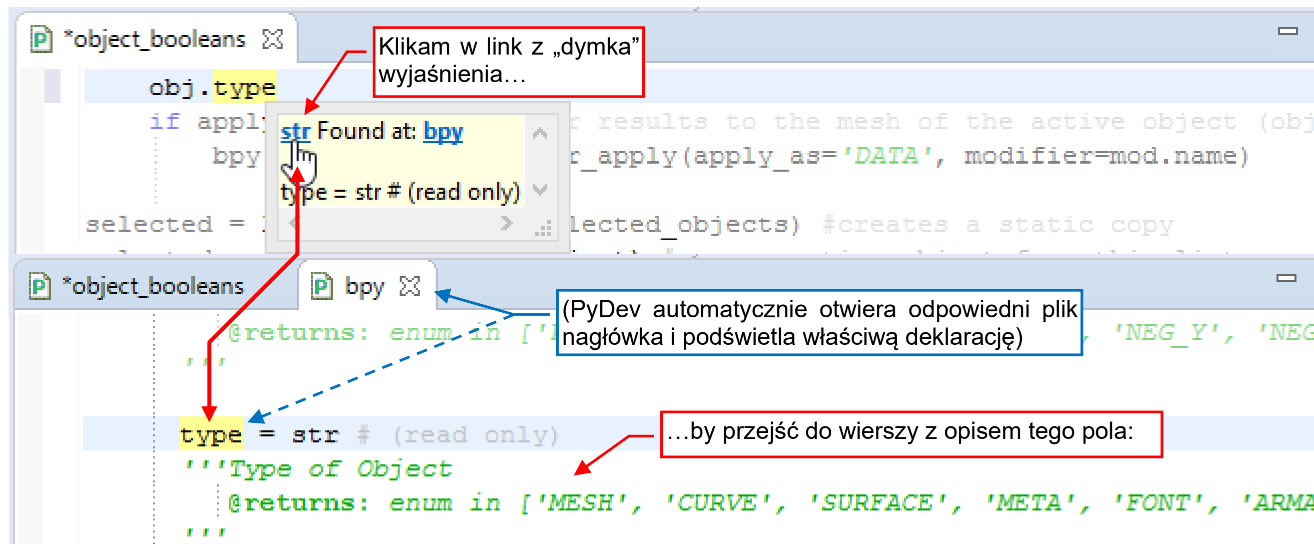
```
>>> bpy.data.objects["Camera"].type
'CAMERA'

>>> bpy.data.objects["Cube"].type
'MESH'

>>> bpy.data.objects["Light"].type
'LIGHT'
```

Wygląda na to, że typem właściwym dla modyfikatorów *Booelan* jest **‘MESH’**

Rysunek 3.5.16 Szybkie sprawdzanie typów obiektów API

Copyright Witold Jaworski, 2011-2019.

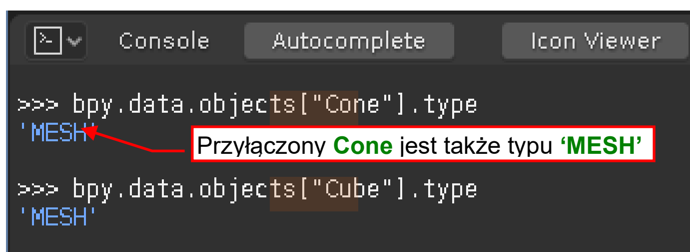
Okazuje się, że kamera jest typu **'CAMERA'**, światło – **'LIGHT'**, instancja torusa – **'EMPTY'** (nie pokazałem tego przypadku na ilustracji). Wszystkie pozostałe obiekty sceny są typu **'MESH'**. Pokrywa się to dokładnie z ikonami w układzie *View Layer* (por. Rysunek 3.5.13). Wygląda na to, że pole ***type*** zwraca typ obiektu. Sprawdźmy to jeszcze w opisie API (Rysunek 3.5.17):



Rysunek 3.5.17 Sprawdzanie szczegółów opisu pola *Object.type*

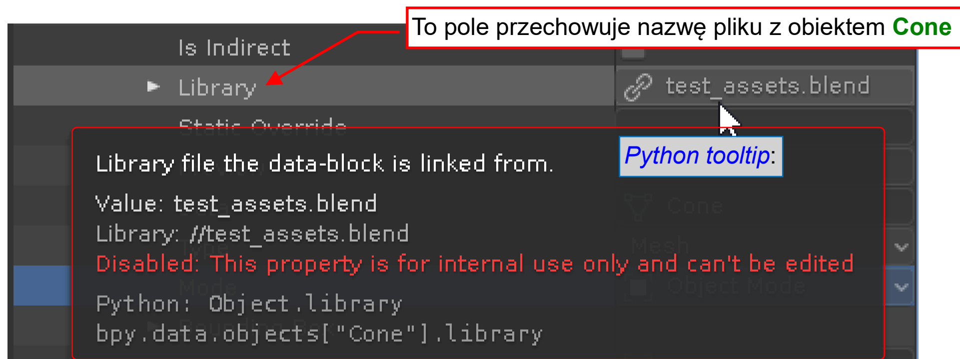
Opis pola ***Object.type*** odszukałem posługując się objaśnieniami PyDev, jak na ilustracji powyżej. Jego zawartość potwierdziła moje przypuszczenia. Konkluzja: każdy obiekt przekazywany procedurze ***boolean_operation()*** jako argument ***tool*** musi być typu **'MESH'**.

Nie będę już tego pokazywał na kolejnym obrazku, ale sprawdziłem także że nie można dodawać modyfikatorów do obiektów przyłączony z innego pliku (jak **Cone**). Oznacza to, że obiektu przyłączonego nie można wskazać naszemu skryptowi jako obiektu aktywnego, bo to spowoduje błąd. Sprawdziłem w konsoli Pythona, że pole ***type*** zwraca wartość **'MESH'** i dla obiektu podączonego (**Cone**), i dla obiektu lokalnego (**Cube**). Jak w takim razie odróżnić obiekt podłączony?



Rysunek 3.5.18 Typy obiektu przyłączonego (Cone**) i lokalnego (**Cube**)**

Aby znaleźć ten wyróżnik, zajrzałem w oknie *Outline* do zawartości obiektu **Cone**. Istotnie, wkrótce zauważyłem tam pole o nazwie ***library*** (Rysunek 3.5.19):



Rysunek 3.5.19 Referencja do pliku, z którego jest przyłączony obiekt

Z deklaracji w pliku nagłówków ***bpy*** wynika, że pole ***library*** zwraca referencję do obiektu klasy *Library*, zawierającego pełną ścieżkę do pliku źródłowego i inne szczegóły. Mnie interesuje jednak co innego: dla wszystkich obiektów lokalnych pole ***library*** zwraca wartość **None** (sprawdziłem w konsoli). To poszukiwany wyróżnik!

Zmieńmy więc odpowiednio główny kod naszego skryptu (Rysunek 3.5.20):

```
selected = list(bpy.context.selected_objects) #creates a static copy
active = bpy.context.object

if active in selected: selected.remove(active)

if active.type != 'MESH':
    print("Cannot execute: target object is not a mesh")
else:
    if active.library != None or active.data.library != None:
        print("Cannot execute: target object is linked from another file")
    else:
        for tool in selected:   #Apply
            if tool.type == 'MESH':
                boolean_operation(tool,'DIFFERENCE')
            else: #at least mark this improper object to not repeat the error
                tool.select_set(False)
        print("bool_operation: Done!")
```


Rysunek 3.5.20 Główny kod skryptu z dodaną weryfikacją poprawności danych

Na poczatku, dla większej czytelności kodu, stworzyłem zmienną ***active*** i przypisałem jej referencję do obiektu aktywnego. Po kilku doświadczeniach odkryłem, że w Blenderze obiekt aktywny może nie być wcale zaznaczonych – więc dodałem odpowiedni warunek w kolejnej linii. (Inaczej próba usunięcia obiektu ***active*** z listy ***selected*** mogłaby wywołać błąd).

W dalszych liniach sprawdzam, czy typem obiektu aktywnego jest **'MESH'**. Jeżeli tak – sprawdzam jeszcze, czy nie jest przypadkiem obiektem przyłączonym z innego pliku. Zwróć uwagę, że dodałem tu jeszcze drugi warunek: jego siatka także nie może być przyłączona[1]. (W nieocenionym oknie *Outliner* znalazłem, że pole ***bpy.types.Object.data*** zwraca referencję do siatki obiektu).

Wreszcie, gdy obiekt aktywny jest poprawny, wywołuję przygotowaną poprzednio pętlę dla wszystkich obiektów-narzędzi (***tool***) z listy ***selected***. Jednak tym razem przed wywołaniem procedury ***boolean_operation()*** sprawdzam, czy ***tool*** jest siatką. Jeżeli nie – wyłączam go z aktualnej selekcji, aby użytkownik przez pomyłkę nie użył tego obiektu powtórnie. (Polecenia zmieniające zaznaczenie nie są wyświetlane w oknie dziennika. Informację o tym, że metodą ***select_set()*** można dołączyć/wykluczyć obiekt z aktuanego zaznaczenia, znalazłem w dokumentacji API[2]).

W następnej sekcji zajmę się przechwytywaniem ewentualnych błędów w tym kodzie. Zmienię także komunikaty, wyświetlane przez skrypt, na bardziej czytelne dla użytkownika.

---

[1] Czasami może się zdarzyć, że w lokalnym obiekcie używasz tylko siatki. Może tak być, gdy np. „pomalowałeś" ją lokalnym materiałem, przypisanym do obiektu.

[2] W innych blokach danych – np. wierzchołkach czy krawędziach – znajdziesz w *Outliner* pole o nazwie ***select***, które można ustawić na **True**/**False**. We wcześniejszych wersjach Blendera każdy obiekt sceny także miał takie pole. Zacząłem więc poszukiwania metody do zaznaczania obiektów od znalezienia w Release Notes Blendera 2.8 sekcji o zmianach w API. Przeszedłem tam to podsekcji Scene and object API, gdzie ostatecznie znalazłem ustęp na ten temat. Ta zmiana ma zapewne związek z wprowadzeniem w nowej wersji zmian w architekturze pliku **.blend*. Teraz stan zaznaczenia obiektu jest zapamiętywany oddzielnie dla każdej z tzw. *View Layers* („warstw widoku", określanych w API również jako *render layers*). Każda *View Layer* przechowuje instancje (referencje do) obiektów sceny. Są to obiekty klasy ***bpy.types.ObjectBase***. Takie instancje mają klasyczne pole ***select***, pozwalające sterować zaznaczeniem obiektu na tej warstwie. Drugim polem klasy ***ObjectBase*** jest ***object***, zwracający powiązany obiekt sceny. Listę instancji obiektów z aktualnej warstwy znajdziesz w następujących polach ***bpy.context***: ***selectable_bases***, ***editable_bases***, ***visible_bases***, ***selected_bases***, ***active_base***.

Copyright Witold Jaworski, 2011-2019.

Podsumowanie

- Do poznawania wyrażeń API odpowiadających wyświetlanym na ekranie Blendera danym bardzo się przydaje włączenie w ustawieniach opcji *Python Tooltips* (str. 61, 62);
- Blender zawsze dodaje nowy modyfikator obiektu na koniec listy modyfikatorów. Dlatego znajdziesz go na ostatniej pozycji listy ***modifiers*** (str. 62).
- Wszystkim dalszym wywołaniom operatorów przekazuj wartość pola ***name*** nowego modyfikatora (str. 62). (Nie musisz wiedzieć, jaka to nazwa - Blender nadaje ją automatycznie);
- W obiekcie ***bpy.context*** jest kilka pól udostępniających informację o aktualnie zaznaczonych obiektach. Można je przejrzeć używając autokompletacji w Eclipse (str. 63). Niestety, nie są udokumentowane. Zazwyczaj wystarcza pole ***bpy.context.selected_objects***. Jeżeli jednak potrzebujesz np. wyłącznie listy zaznaczonych obiektów lokalnych - użyj ***bpy.context.selected_editable_objects*** (str. 64). W innych przypadkach bardziej odpowiednią może okazać się ***bpy.context.selected_bases***: to lista referencji (***bpy.types.ObjectBase***) do obiektów użytych na aktualnej warstwie widoku (*View Layer*, określanej także jako „*render layer*"). W każdym razie przed użyciem w kodzie sprawdź zawartość takiego pola dla jakichś danych testowych (np. w konsoli Pythona);
- Pewnych informacji o obiektach API (jak np. typ obiektu) nie można uzyskać z okna dziennika operacji. Użyj wówczas okna *Outliner* w układzie *Data API* (str. 66). W takim oknie masz udostępnioną w czytelny sposób całą zawartość aktualnego pliku Blendera. Wywołania poszczególnych pól możesz sprawdzać za pomocą *Python tooltips* (wskazując je kursorem myszki - str. 67);
- Do zmiany stanu zaznaczenia (wybrany/nie wybrany) obiektu sceny (tj. klasy ***bpy.types.Object***) należy użyć metody ***select_set()*** (str. 69). (Do odczytania stanu zaznaczenia obiektu użyj metody ***select_get()***);

3.6 Przechwytywanie błędów i komunikacja z użytkownikiem

Na ostatniej ilustracji poprzedniej sekcji (Rysunek 3.5.20, str. 69) można zauważyć, jak dużą część kodu zajmuje sprawdzanie poprawności danych wejściowych. (Porównaj ją chociażby z podstawową wersją tego fragmentu skryptu, pokazywaną przez Rysunek 3.5.10 na str. 65). Równie ważne jak samo znalezienie błędu jest przekazanie tej informacji użytkownikowi tak, aby łatwo zrozumiał jego przyczynę. (Albo przynajmniej szybciej zorientował się, jak następnym razem uniknąć takiej pomyłki). W tej sekcji spróbuję uczynić nasz kod jak najbardziej „błędoodpornym”[1] i ulepszyć wyświetlane komunikaty. Dodatkowo, wprowadzane tutaj zmiany ułatwią przekształcenie tego skryptu na wtyczkę (*add-on*) Blendera. (Tym będziemy się zajmować w następnym rozdziale).

Zacznijmy od tekstu wiadomości o błędach, wyświetlanych przez nasz kod. Sądzę, że bardzo pomocne byłoby dodanie tam, gdzie jest to możliwe, nazwy obiektu, którego dotyczą. W ten sposób użytkownik może szybciej zauważyć, że np. przez pomyłkę wskazał na ekranie Blendera nie to, co powinien. Rysunek 3.6.1 przedstawia nową wersję kodu głównego (funkcjonalnie odpowiada dokładnie kodowi ze str. str. 69):

```
#result constants:
INPUT_ERR = "cannot execute"
ERROR = "run-time error"
WARNING = "warning"
SUCCESS = "completed"

def main (op, apply_objects=True):
    ''' Performs a Boolean operation on the active object, using the other
        selected objects as the 'tools'
        Arguments:
        @op (Enum): a Boolean operation: {'UNION', 'INTERSECT', 'DIFFERENCE'}
        @apply_objects (bool): apply the results to the mesh (optional)
        @returns (list): one or two message parts: [<flag>, Optional_details]
    '''
    selected = list(bpy.context.selected_objects) #creates a static copy
    active = bpy.context.object #active object
    if active in selected: selected.remove(active)
    if active.type != 'MESH':
        return [INPUT_ERR, "target object ('%s') is not a mesh" % active.name]
    else:
        if active.library != None or active.data.library != None:
            return [INPUT_ERR, "target object ('%s') is linked from another file"
                                                                    % active.name]
        else:
            for tool in selected:  #Apply each tool to the active object:
                if tool.type == 'MESH':
                    boolean_operation(tool,op, apply_objects)
                else: #at least mark this improper object to not repeat the error
                    tool.select_set(False)
            return [SUCCESS]
#main code:
result = main('DIFFERENCE')
print("bool_operation --> %s" % str.join(": ",result))
```

Stałe, zwracane w pierwszym elemencie rezultatu funkcji ***main()***

Stałe ***ERROR*** i ***WARNING*** przydadzą się za chwilę

W komunikatach podaję nazwę obiektu

Minimalny kod główny skryptu

Tymczasowa linia „diagnostyczna”

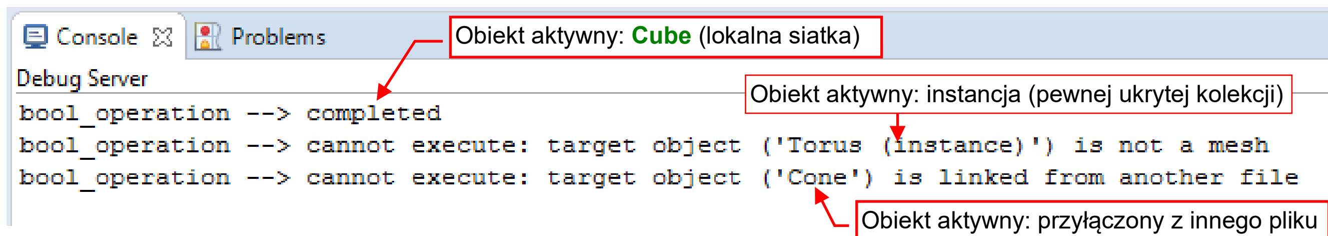
Rysunek 3.6.1 Przekształcony główny kod skryptu

Kod główny skryptu nigdy nie powinien być długi, więc przesunąłem go do funkcji, którą nazwałem ***main()***. (W ten sposób łatwiej będzie go w przyszłości użyć w kodzie wtyczki). Funkcja ***main()*** zwraca listę, w której pierwszy argument pełni rolę flagi, dla których przygotowałem odpowiednie stałe tekstowe. Gdy wszystko przebiegło poprawnie, zwracana lista zawiera jeden element, a gdy wystąpiły jakieś błędy lub komplikacje – przekazuję ich szczegóły w jej drugim elemencie.

[1] Pamiętaj, że tym użytkownikiem możesz być często Ty! Wystarczy, że spróbujesz uruchomić ten skrypt po roku przerwy: sądzę, że nie będziesz wówczas niczego pamiętał.

Copyright Witold Jaworski, 2011-2019.

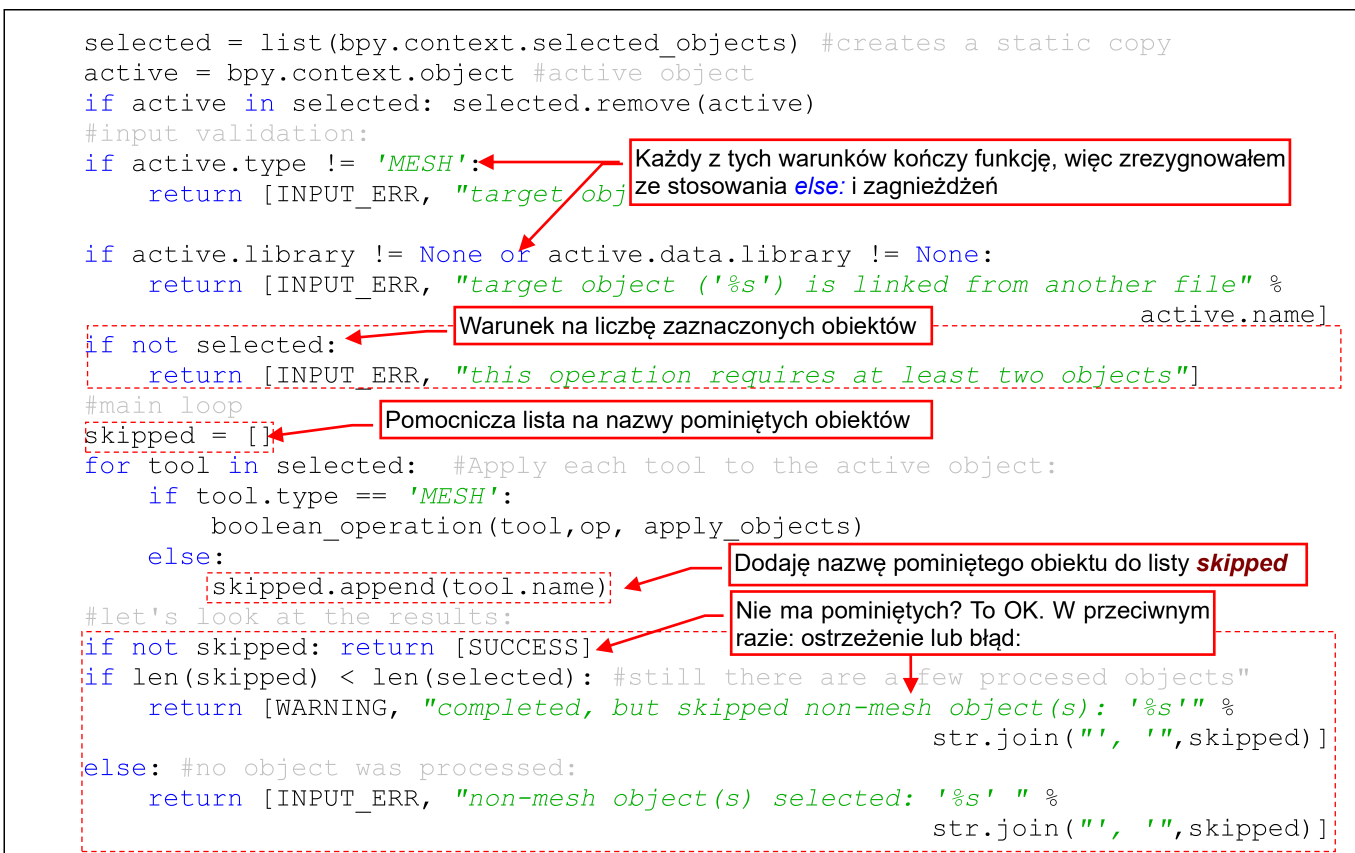
Kod główny skryptu zredukowałem do dwóch linii: wywołania funkcji ***main()*** i wyświetlenia użytkownikowi jej rezultatu (por. Rysunek 3.6.1, ostatnie wiersze). W ten sposób rozdzieliłem operacje tworzenia komunikatów (w funkcji) od ich pokazywania (w kodzie głównym). To zawsze jest lepszy pomysł od wpisywania w kodzie funkcji poleceń ***print()***. Obecnie pozostawiłem jedno takie wyrażenie w ostatniej, tymczasowej linii kodu. Gdy w przyszłości funkcja ***main()*** będzie wywoływana we wtyczce Blendera, komunikaty dla użytkownika będą wyświetlane w zupełnie inny sposób.
W treści samych komunikatów wstawiłem nazwy obiektu, którego dotyczą. Sądzę, że ta dodatkowa informacja pomoże użytkownikowi szybciej się zorientować, co zrobił źle. Rysunek 3.6.2 pokazuje rezultaty kolejnych wywołań nowej wersji skryptu. (Testy w tej sekcji wykonuję dla sceny jaką na str. 66 przedstawia Rysunek 3.5.13):




```
bool_operation --> completed
bool_operation --> cannot execute: target object ('Torus (instance)') is not a mesh
bool_operation --> cannot execute: target object ('Cone') is linked from another file
```

Rysunek 3.6.2 Rezultaty kolejnych wywołań zmodyfikowanego skryptu

Potem dodałem do funkcji ***main()*** jeszcze kolejne testy, sprawdzające pozostałe zaznaczone obiekty (Rysunek 3.6.3). W związku z tym, że każdy z nich kończy się poleceniem *return*, mogłem zrezygnować z zagnieżdżania kolejnych instrukcji *if*. (Przy większej liczbie prostych wykluczeń takie zagnieżdżanie zaczyna pogarszać czytelność kodu):




```
selected = list(bpy.context.selected_objects) #creates a static copy
active = bpy.context.object #active object
if active in selected: selected.remove(active)
#input validation:
if active.type != 'MESH':
    return [INPUT_ERR, "target obj

if active.library != None or active.data.library != None:
    return [INPUT_ERR, "target object ('%s') is linked from another file" %
                                                              active.name]
if not selected:
    return [INPUT_ERR, "this operation requires at least two objects"]
#main loop
skipped = []
for tool in selected:  #Apply each tool to the active object:
    if tool.type == 'MESH':
        boolean_operation(tool,op, apply_objects)
    else:
        skipped.append(tool.name)
#let's look at the results:
if not skipped: return [SUCCESS]
if len(skipped) < len(selected): #still there are a few procesed objects"
    return [WARNING, "completed, but skipped non-mesh object(s): '%s'" %
                                                  str.join("', '",skipped)]
else: #no object was processed:
    return [INPUT_ERR, "non-mesh object(s) selected: '%s' " %
                                                  str.join("', '",skipped)]
```

Rysunek 3.6.3 Zmodyfikowany kod funkcji *main()* (dodane dalsze testy danych wejściowych)

Przed wywołaniem głównej pętli sprawdzam, czy lista ***selected*** przypadkiem nie jest pusta (może tak być, gdy użytkownik zaznaczył tylko jeden obiekt). Jeżeli jest – sygnalizuję błąd.
Zamiast wykluczać z zaznaczenia obiekty niezawierające siatki (por. str. 69, Rysunek 3.5.20), zdecydowałem się wyliczyć użytkownikowi ich nazwy. Gromadzę je w pomocniczej liście ***skipped***. Gdy wśród zaznaczonych był choć jeden poprawny obiekt – zgłaszam tylko ostrzeżenie. Gdy wszystkie zaznaczone obiekty były niepoprawne – sygnalizuję to jako błąd.

Rysunek 3.6.4: przedstawia rezultaty kolejnych testów skryptu. (Testy w tej sekcji wykonuję dla sceny jaką na str. 66 przedstawia Rysunek 3.5.13):


```
bool_operation --> cannot execute: this operation requires at least two objects
bool_operation --> warning: completed, but skipped non-mesh object(s): 'Torus (instance)'
bool_operation --> cannot execute: non-mesh object(s) selected: 'Torus (instance)'
```

Rysunek 3.6.4 Testy wywołań zmodyfikowanego skryptu dla różnych kombinacji danych wejściowych

Nie mam złudzeń, że te tych pięć zaimplementowanych już testów pozwoli uniknąć wszelkich błędów podczas wykonywania skryptu. Aby zachować resztki kontroli nad taką sytuacją, ujmijmy cały dotychczasowy kod procedury głównej (funkcji ***main()***) w wyrażenie *try:* … *except:* (Rysunek 3.6.5):

```
import traceback #for error handling

def main (op, apply_objects=True):
    ''' Performs a Boolean operation on the active object, using the other
        selected objects as the 'tools'
        Arguments:
        @op (Enum): a Boolean operation: {'UNION', 'INTERSECT', 'DIFFERENCE'}
        @apply_objects (bool): apply results to the mesh (optional)
        @returns (list): one or two message parts: [<flag>, Optional_details]
    '''
    try:
                                  ...
        Dotychczasowy kod funkcji (jak na str. 72, Rysunek 3.6.3)
                                  ...

    except Exception as err: #Just in case of a run-time error:
        traceback.print_exc()

        cntx_msg = ""
        if 'active' in locals(): cntx_msg += "occured for object(s): '%s'" % 
                                                                 active.name
        if 'tool' in locals(): cntx_msg += ", '%s'" %tool.name

        return [ERROR, "%s %s" % (str(err),cntx_msg)]
```

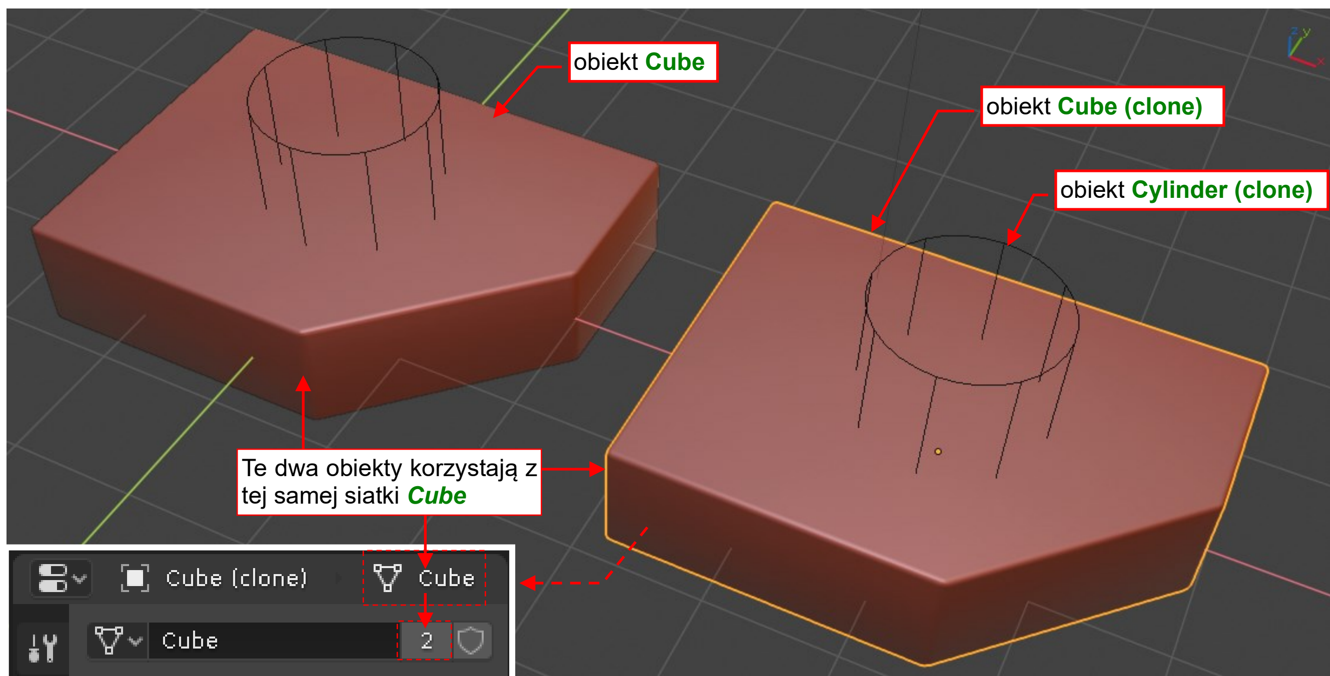

Rysunek 3.6.5 Przechwytywanie w procedurze *main()* ewentualnych dalszych błędów czasu wykonywania (wyjątków)

Wyrażenie *except:* przechwytuje każdy rodzaj wyjątku i umieszcza go w lokalnej zmiennej ***err***. Następnie drukuje za pomocą funkcji ***traceback.print_exc()*** standardową, szczegółową informację o stanie stosu Pythona w momencie wystąpienia wyjątku. Ten tekst pojawi się tylko w konsoli Blendera, gdyż jest ro informacja diagnostyczna, przeznaczona wyłącznie dla programistów.

Z myślą o zwykłych użytkownikach staram się sformatować w pomocniczej zmiennej ***cntx_msg*** krótki tekst na temat kontekstu zdarzenia. Sądzę, że większość błędów wystąpi tam, gdzie wykonujemy właściwą operację: procedurze ***boolean_operation()***. Dlatego w ***cntx_msg*** staram się podać nazwy aktywnego obiektu i zmiennej lokalnej ***tool***, dla których wywołanie procedury zakończyło się błędem. Oczywiście, nie mogę jednak tego być pewien. Dlatego, aby nie wywołać błędu w obsłudze błędu, przed odwołaniem się do pola ***name*** obiektów ***active*** i ***tool*** sprawdzam, czy ich na pewno już figurują wśród zmiennych lokalnych (kolekcji ***locals()***).

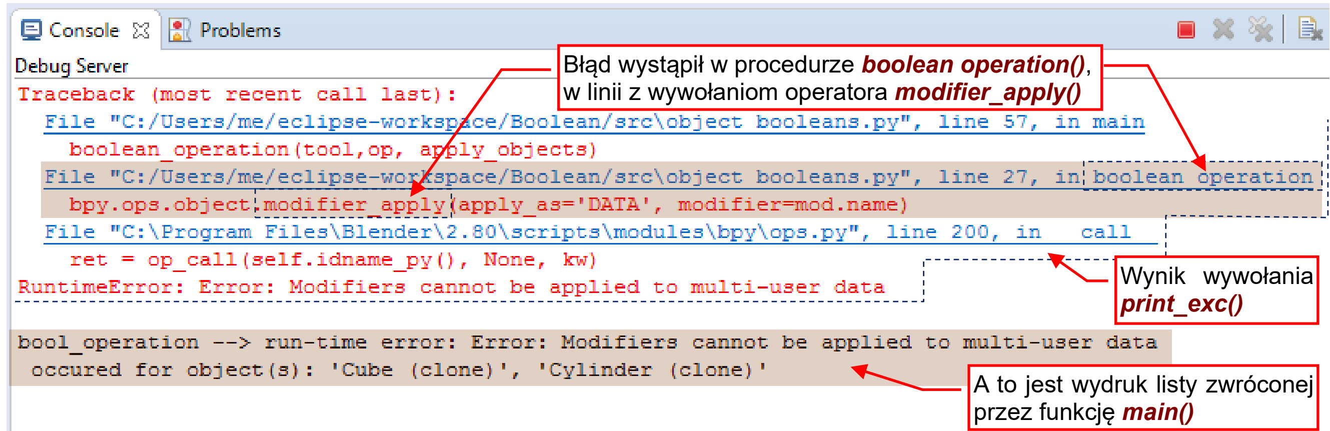
Copyright Witold Jaworski, 2011-2019.

Na kolejny błąd w skrypcie, który pozwoliłby sprawdzić działanie dopisanego przed chwilą wyrażenia *try:* … *except:* nie trzeba długo czekać. Wystarczy stworzyć klon obiektów **Cube** i **Cylnder**. (Obiekty **Cube** i **Cube (clone)** współdzielą tę sama siatkę o nazwie ***Cube*** – tak, jak to pokazuje Rysunek 3.6.6):



Rysunek 3.6.6 Klon obiektu Cube (siatka współdzielona pomiędzy dwoma obiektami)

Następnie wywołałem zaznaczyłem obiekt **Cylinder (clone)** i **Cube (clone)**, po czym wywołałem skrypt. Oto rezultat (Rysunek 3.6.7):



Rysunek 3.6.7 Rezultat wywołania skryptu dla obiektu ze współdzielona siatka

No pięknie. Przekonaliśmy się, że „przechwytywanie” błędów działa poprawnie (funkcja ***main()*** zwróciła komunikat o błędzie wzbogacony o dodatkową informację). Zupełnie zapomniałem, że Blender nie pozwala „utrwalić” rezultatów modyfikatora w siatce, która jest współdzielona. Najlepiej od razu poprawmy ostatnie linie procedury ***boolean_operation()*** (porównaj poniższy kod z kodem ze str. 62, Rysunek 3.5.5):

```
    if apply:
        if obj.users > 1 or obj.data.users > 1:
            bpy.ops.object.select_all(action='DESELECT')
            obj.select_set(True) #select obj, only
            bpy.ops.object.make_single_user(type='SELECTED_OBJECTS',
                                            object=True, obdata=True)
        bpy.ops.object.modifier_apply(apply_as='DATA', modifier=mod.name)
```

Pola ***users*** to liczniki użycia – znalazłem je w oknie *Outliner*

Zaznaczam tylko obiekt aktywny z powodu tej stałej

Rysunek 3.6.8 Tworzenie lokalnej kopii siatki przed wywołaniem *modifier_apply()* (ostatnie linie procedury *boolean_operation()*)

Przed wywołaniem ***modifier_apply()*** sprawdzam, czy licznik referencji (pole ***users***) aktywnego obiektu i jego siatki są większe od 1. Jeżeli tak – tworzę ich lokalną kopię poleceniem ***make_single user()***.

Abyś nie pogubił się w tych wszystkich zmianach, które wprowadziłem, podaję pełen kod aktualnego skryptu:

```
import bpy
import traceback #for error handling

def boolean_operation (tool, op, apply=True):
    '''Performs a Boolean operation on the active object
        Arguments:
        @tool (Object): the other object, not affected by this method
        @op (Enum): a Boolean operation: {'UNION', 'INTERSECT', 'DIFFERENCE'}
        @apply (bool): apply results to the mesh (optional)
    '''
    obj = bpy.context.object #active object
    bpy.ops.object.modifier_add(type='BOOLEAN')#adds new modifier to obj
    mod = obj.modifiers[-1] #new modifier always appear at the end of this list
    while obj.modifiers[0] != mod: #move this modifier to the first position
        bpy.ops.object.modifier_move_up(modifier=mod.name)
    mod.operation = op #set the operation
    mod.object = tool #activate rhe modifier
    if apply: #applies modifier results to the mesh of the active object (obj):
        if obj.users > 1 or obj.data.users > 1: #obj has to be a single-user datablock
            #make sure, that obj is the only selected object:
            bpy.ops.object.select_all(action='DESELECT') #deselect all
            obj.select_set(True) #select obj, only
            bpy.ops.object.make_single_user(type='SELECTED_OBJECTS',
                                            object=True, obdata=True)
        bpy.ops.object.modifier_apply(apply_as='DATA', modifier=mod.name)

#result constants:
INPUT_ERR = 'cannot execute'
ERROR = 'run-time error'
WARNING = 'warning'
SUCCESS = 'completed'

def main (op, apply_objects=True):
    ''' Performs a Boolean operation on the active object, using the other
        selected objects as the 'tools'
        Arguments:
        @op (Enum): a Boolean operation: {'UNION', 'INTERSECT', 'DIFFERENCE'}
        @apply_objects (bool): apply results of the Boolean operation to the mesh (optional)
        @returns (list): one or two message parts: [<flag>, Optional_details]
    '''
    try:
        selected = list(bpy.context.selected_objects) #creates a static copy
        active = bpy.context.object #active object
        if active in selected: selected.remove(active)
        #input validation:
        if active.type != 'MESH':
            return [INPUT_ERR, "target object ('%s') is not a mesh" % active.name]
        if active.library != None or active.data.library != None:
            return [INPUT_ERR, "target object ('%s') is linked from another file" % active.name]
        if not selected: return [INPUT_ERR, "this operation requires at least two objects"]
        #main loop
        skipped = []#auxiliary list for the skipped object names
        for tool in selected:  #Apply each tool to the active object:
            if tool.type == 'MESH':
                boolean_operation(tool,op, apply_objects)
            else: #store the name of the skipped object
                skipped.append(tool.name)
        #let's look at the results:
        if not skipped: return [SUCCESS]
        if len(skipped) < len(selected): #still there are a few procesed objects"
            return [WARNING, "completed, but skipped non-mesh object(s): '%s'"
                                                        % str.join("', '",skipped)]
        else: #no object was processed:
            return [INPUT_ERR, "non-mesh object(s) selected: '%s' " % str.join("', '",skipped)]
    except Exception as err: #Just in case of a run-time error:
        traceback.print_exc() #print the Python stack details in the console (for you)
        cntx_msg = "" #format the diagnostic message:
        if 'active' in locals(): cntx_msg += "occured for object(s): '%s'" % active.name
        if 'tool' in locals(): cntx_msg += ", '%s'" %tool.name
        return [ERROR, "%s %s" % (str(err),cntx_msg)]
#main code:
result = main('DIFFERENCE')
print("bool_operation --> %s" % str.join(": ",result) )
```

Rysunek 3.6.9 Kompletny kod aktualnej wersji skryptu

Copyright Witold Jaworski, 2011-2019.

Zauważ, że w funkcji ***main()*** umieściłem całą walidację danych wejściowych i sygnalizowanie ewentualnych błędów wykonania. Ten „porządkowy" kod zajmuje więcej miejsca niż właściwa operacja, wykonywana przez procedurę ***boolean_operation()***. To normalna proporcja w programach, które muszą „wchodzić w interakcję" z najbardziej nieprzewidywalnym elementem otoczenia: użytkownikiem 😊.

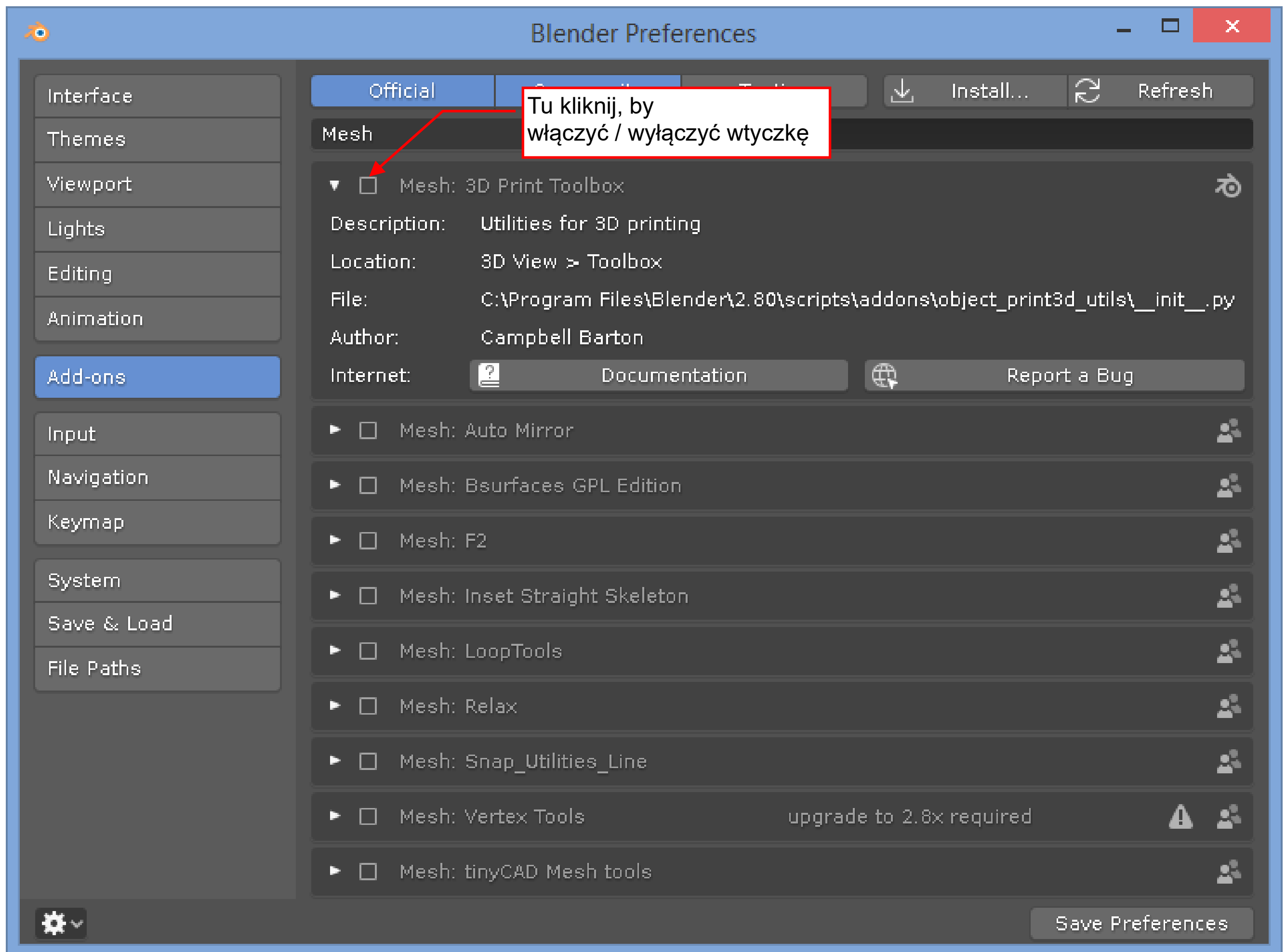
Obecnie jedynym wpisanym w sposób jawny („ręczny") parametrem tego skryptu jest rodzaj wykonywanej operacji Boole'a (**'DIFFERENCE'**). W następnym rozdziale stworzę prosty interfejs użytkownika, który pozwoli na jego interaktywny wybór. Zrobię to przy okazji przekształcenia tego kodu w dodatek (*add-on*) Blendera.

Podsumowanie

- Dotychczasowy kod główny umieściłem w funkcji ***main()***, która zwraca listę z rezultatem (stała) i ewentualnym komunikatem dla użytkownika o napotkanych problemach (tekst) (str. 71). Taką funkcję będzie można łatwo użyć we wtyczce (*add-on*) do Blendera;
- W komunikatach warto dostarczyć użytkownikowi informacji o kontekście sygnalizowanego problemu. W przypadku opisywanego skryptu to nazwa obiektu aktywnego (i czasami obiektu – narzędzia) (str. 72 i 73);
- Do przechwycenia wszelkich nieprzewidzianych błędów umieść w procedurze głównej klauzulę *try:* … *except:* (str. 73);

Rozdział 4. Przerabianie skryptu na wtyczkę Blendera (add-on)

Zapewne parę razy zaglądałeś do okna konfiguracji Blendera (***Edit→Preferences***). Przypuszczam, że już zwróciłeś uwagę na zakładkę *Add-ons*:



Add-on to dodatek (wtyczka) do Blendera, napisana w Pythonie. To okno pozwala wybrać zestaw dodatków, z których chcesz aktualnie korzystać. Wtyczki podczas inicjalizacji dodają do GUI Blendera nowe elementy: przyciski, polecenia menu, panele. Zresztą cały interfejs użytkownika Blendera jest napisany w Pythonie, z użyciem tych samych poleceń API, których używają wtyczki.

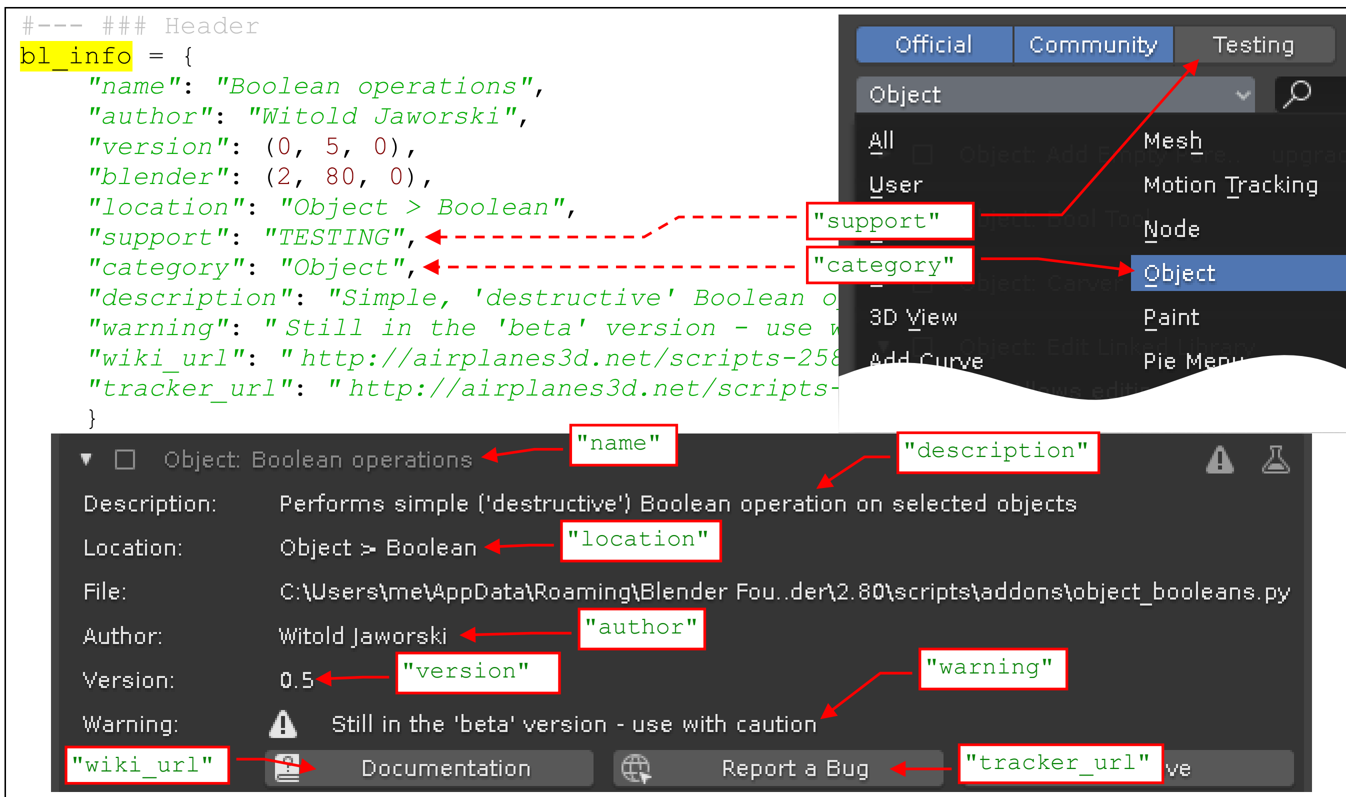
W tym rozdziale pokażę, jak przerobić naszą procedurę na dodatek do Blendera. Ten *add-on* będzie dodawał do menu *Object* submenu *Boolean* z trzema poleceniami: *Difference*, *Union*, *Intersection*.

Copyright Witold Jaworski, 2011-2019.

4.1 Dostosowanie struktury skryptu

Do tej pory nasz skrypt był „linearny”: wykonywało się to, co było wpisane w kodzie głównym. Wtyczki Blendera działają inaczej, o czym przekonasz się w tej sekcji. W związku z tym ich kod musi mieć określoną strukturę. Poznasz ją, przy okazji przerabiania naszego kodu na wtyczkę.

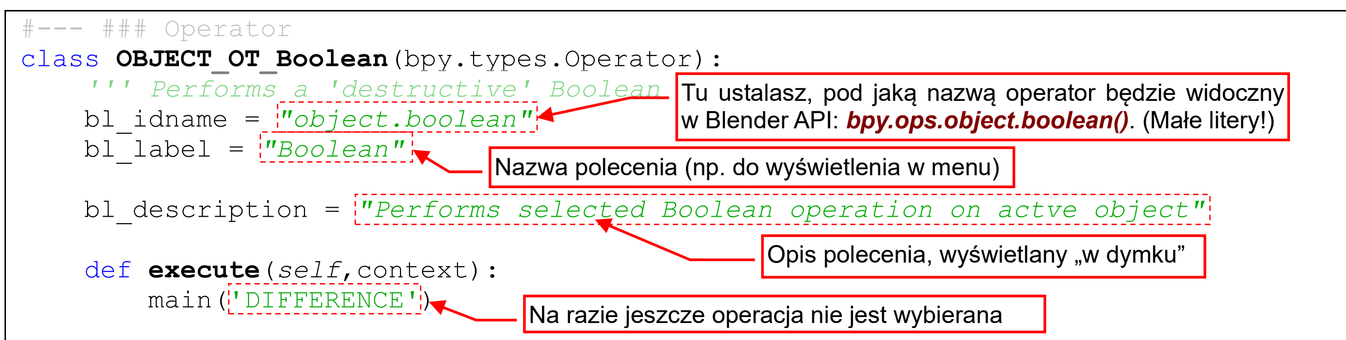
Zacznijmy od nagłówka. Każda wtyczka Blendera musi zawierać globalny słownik o nazwie ***bl_info***. Kluczami tego słownika są ściśle określone teksty: **„name”**, **„autor”**, **„location”**, itp. Blender wykorzystuje tę strukturę do wyświetlenia opisu wtyczki w oknie *Add-Ons* (Rysunek 4.1.1):



Rysunek 4.1.1 Struktura nagłówka wtyczki i jej reprezentacja w oknie *Blender Preferences*.

Niektóre wartości słownika możesz pozostawić puste — np. odsyłacze do opisu i raportu błędów (**„wiki_url”**, **„tracker_url”**). Ważnymi polami są **„support”** i **„category”**: używaj tu wyłącznie tekstów, które widzisz na liście kategorii w zakładce *Add-Ons*. Jeżeli wpiszesz coś, czego tam nie ma — Twój dodatek będzie widoczny tylko po wybraniu kategorii *All*.

Wtyczka wstawi naszą metodę ***main()*** do listy poleceń Blendera. Aby to było możliwe, musimy naszą procedurę „obudować” prostą klasą operatora (Rysunek 4.1.2):

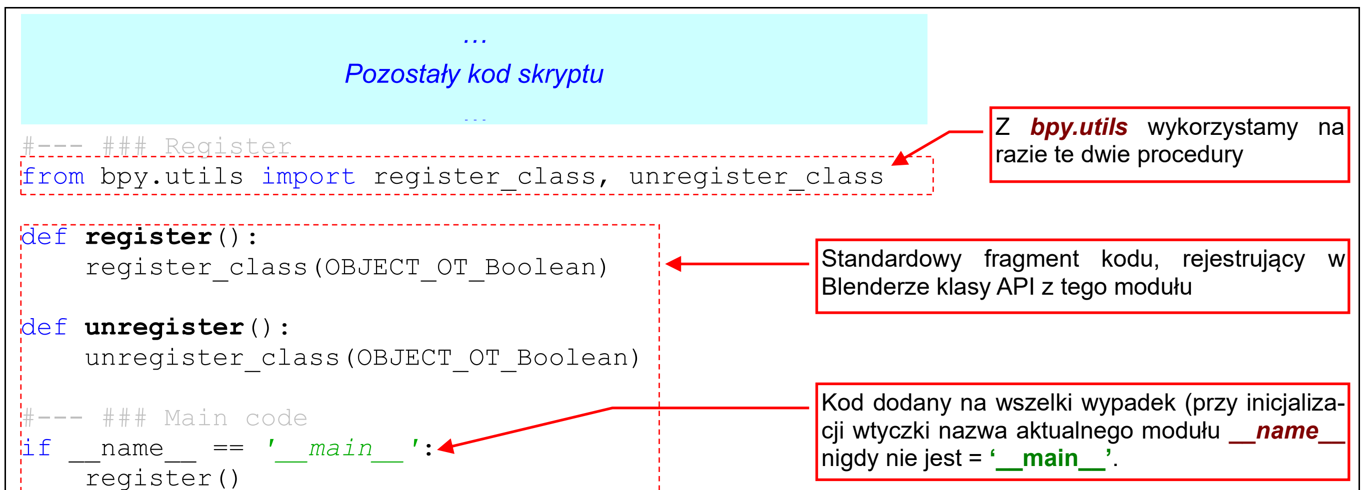


Rysunek 4.1.2 „Obudowanie” procedury *main()* klasą operatora.

Nadałem tej klasie nazwę zgodną z sugestią twórców API: ***OBJECT_OT_Boolean***. Nowy operator musi koniecznie pochodzić od abstrakcyjnej klasy ***bpy.types.Operator***. Inaczej nie będzie działał.

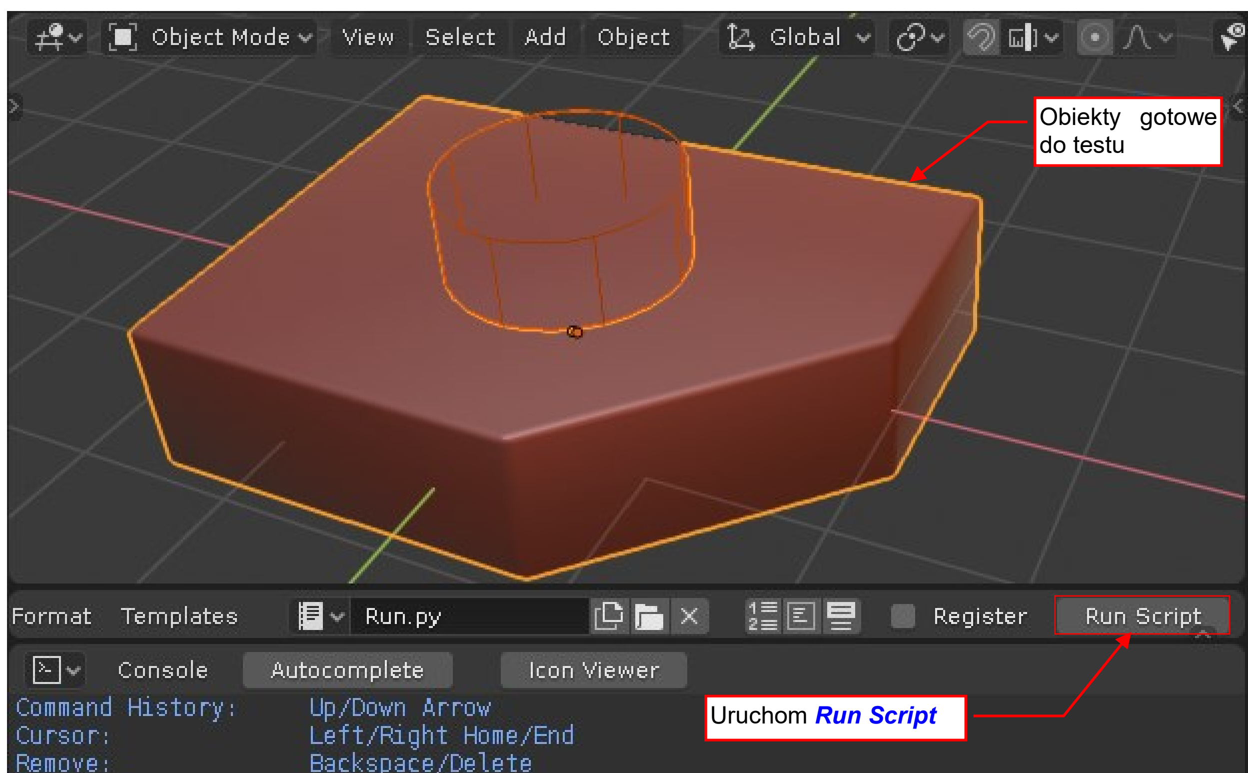
W klasie operatora koniecznie zdefiniuj pola ***bl_idname***, ***bl_label*** (Rysunek 4.1.2). Możesz także dodać pole ***bl_description***. (Jeżeli go nie ma, Blender wyświetla w „dymku” z opisem polecenia komentarz typu *docstring*, umieszczony pod nagłówkiem klasy). Na razie nasza klasa będzie miała jedną metodę, o ściśle określonej nazwie i liście parametrów: ***execute(self, context)***. Umieściłem w niej wywołanie funkcji ***main()***, na razie z operacją wpisaną na stałe. Otrzymanym kontekstem oraz przekazaniem rezultatu funkcji zajmiemy się za chwilę.

Aby Blender „zauważył” klasy API, zdefiniowane w Twoim module, musisz dodać do skryptu odpowiednie funkcje odpowiedzialne za ich rejestrację. Ten kod praktycznie zawsze wygląda tak samo: na końcu skryptu dodaj import z ***bpy.utils*** dwóch pomocniczych funkcji. Następnie wykorzystaj je w procedurach o nazwie (koniecznie!) ***register()*** i ***unregister()*** (Rysunek 4.1.3):



Rysunek 4.1.3 Kod rejestrujący w środowisku Blendera klasy zdefiniowane w skrypcie.

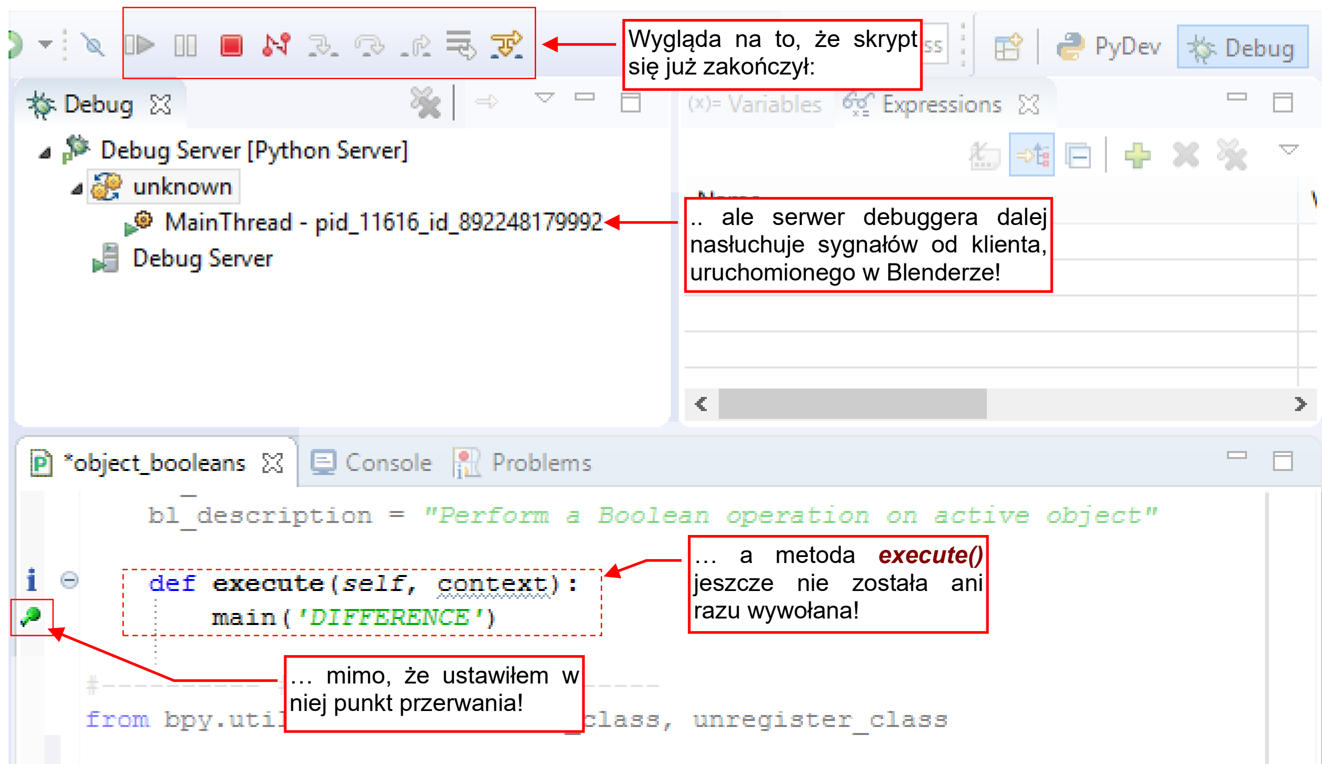
Sprawdźmy teraz działanie tak przygotowanego skryptu. Upewnij się, że serwer debugera w Eclipse jest włączony. Ustaw odpowiednie środowisko w Blenderze, a potem naciśnij przycisk ***Run Script*** (Rysunek 4.1.4):



Rysunek 4.1.4 Uruchomienie wtyczki w debugerze

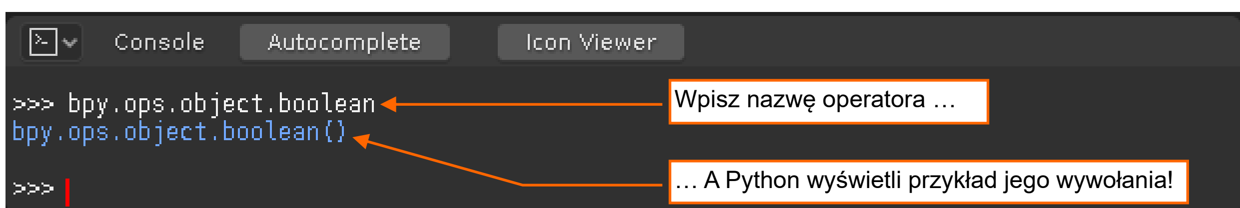
Copyright Witold Jaworski, 2011-2019.

I co? W Eclipse wygląda na to, że skrypt się zakończył, a otworu w naszej testowej płytce nie ma? Upewnij się raz jeszcze: dodaj do funkcji ***execute()*** punkt przerwania, i uruchom skrypt jeszcze raz. Nadal to samo (Rysunek 4.1.5):



Rysunek 4.1.5 Debugger po uruchomieniu skryptu

Rzecz polega na tym, że obecnie główny kod skryptu wcale nie wywołuje procedury ***main()***. On tylko rejestruje nowe polecenie (operator) Blendera, o takiej nazwie, jaką wpisałeś w polu ***bl_idname***. W naszym przypadku to **„object.boolean"** (por. str. 78, Rysunek 4.1.2). Sprawdź w konsoli Pythona, czy istnieje procedura ***bpy.ops.object.boolean*** (Rysunek 4.1.6):

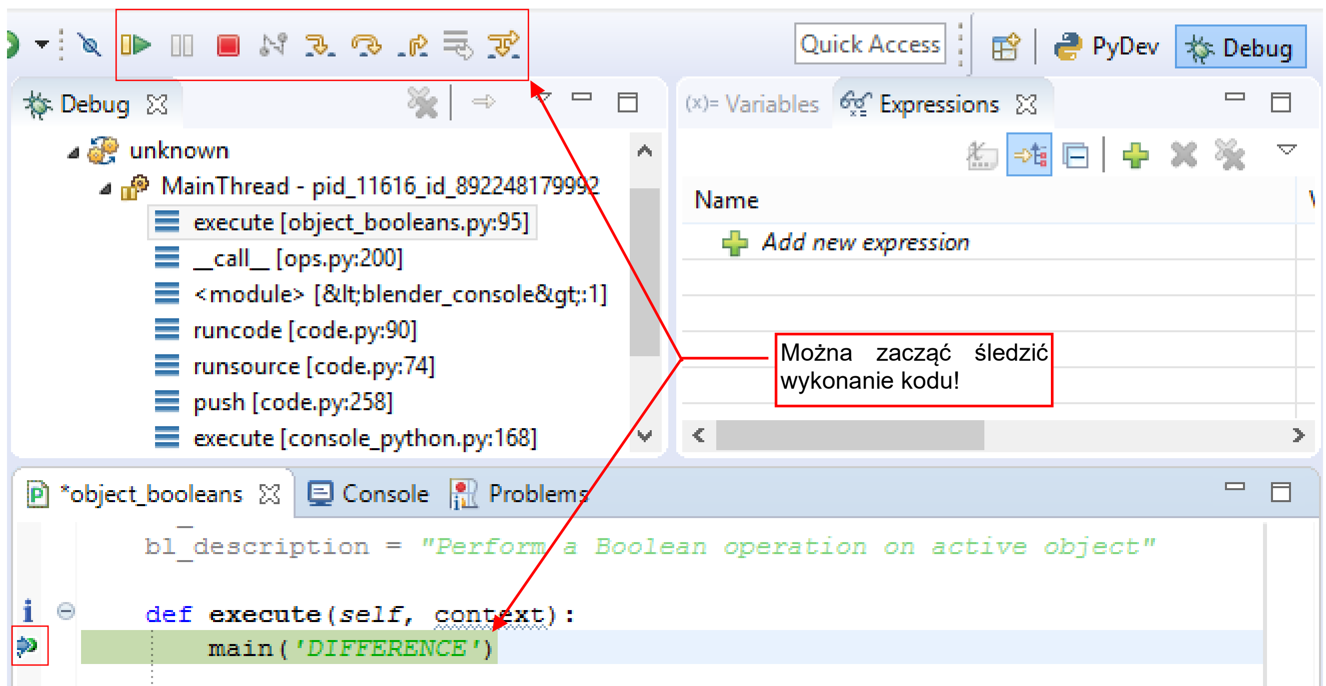


Rysunek 4.1.6 Sprawdzenie, czy w Blenderze pojawiło się nowe polecenie (*operator*)

Taki operator można teraz dodać do jakiegoś menu albo umieścić gdzieś jako przycisk. Integracją z GUI zajmiemy się jednak w następnej sekcji tego rozdziału. Na razie po prostu wywołajmy to polecenie „z ręki" — w konsoli Pythona (Rysunek 4.1.7):

Rysunek 4.1.7 Testowe wywołanie operatora…

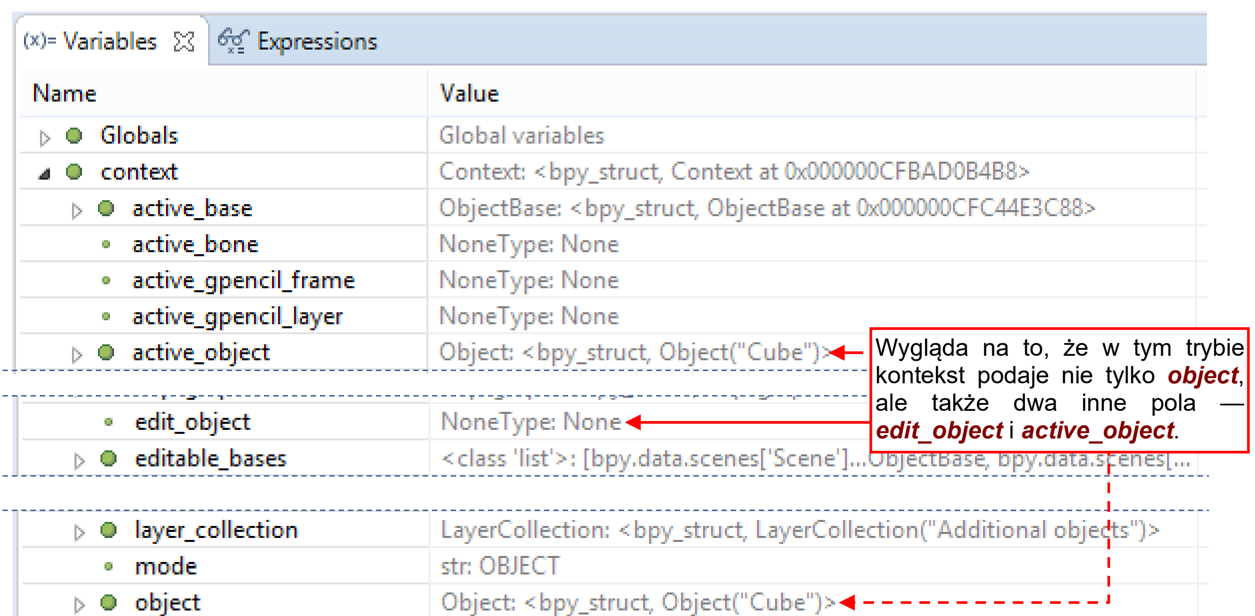
Ekran Blendera uległ zamrożeniu, a w naszym Eclipse włączył się debuger. Aktualna pozycja w kodzie to wstawiony wcześniej punkt przerwania wewnątrz procedury ***execute()*** (Rysunek 4.1.8):



Rysunek 4.1.8 …i proces debugera zatrzymuje się na punkcie przerwania

Widzisz? Zasymulowaliśmy tutaj to, co będzie robił z naszym operatorem Blender. Gdy wywołasz polecenie ***bpy.ops.object.boolean()*** (zazwyczaj poprzez polecenie z menu), Blender utworzy nową instancję klasy ***Object_OT_Boolean***. Zrobi to tylko po to, by wywołać jej metodę ***execute()***. Po wykonaniu tej funkcji obiekt będzie zaraz zwolniony (tzn. usunięty). Taki sposób działania („nie wołaj nas, to my wywołamy ciebie") jest typowy dla większości środowisk graficznych.

Przy okazji: zwróć uwagę na parametry procedury, dostępne w oknie *Variables*. Rozwiń np. parametr ***context***, aby przekonać się, jak od strony programu wygląda kontekst wywołania naszego operatora (Rysunek 4.1.9):



Rysunek 4.1.9 Podgląd zawartości kontekstu wywołania operatora

Struktura ***context*** może mieć różne pola dla różnych trybów pracy Blendera. Przeglądając ją, zawsze można odkryć jakiś ciekawy szczegół. Na przykład — co to za pola ***active_object*** i ***edit_object***? Niestety, na razie na stronach dokumentacji Blender API żadne pole kontekstu (moduł ***bpy.context***) nie ma jakiegokolwiek opisu.

Copyright Witold Jaworski, 2011-2019.

Przyjrzyjmy się jeszcze w oknie *Variables* samemu obiektowi ***self***. Zwróć uwagę, że pole ***bl_idname*** ma tutaj inną wartość. Także nasza klasa ***OBJECT_OT_Boolean*** ma teraz przypisane inne klasy bazowe (Rysunek 4.1.10):

(x)= Variables | Expressions

Name	Value
Globals	Global variables
context	Context: <bpy_struct, Context at 0x000000CFBAD0B4B8>
self	OBJECT_OT_Boolean: <bpy_struct, OBJECT_OT_boolean("OBJECT_OT_boolean")>
bl_description	str: Perform a Boolean operation on active object
bl_idname	str: OBJECT_OT_boolean
bl_label	str: Boolean
bl_options	set: set()
bl_rna	OBJECT_OT_Boolean: <bpy_struct, Struct("OBJECT_OT_boolean")>
bl_translation_context	str: Operator
bl_undo_group	str:
has_reports	bool: False
is_repeat	bpy_func: <bpy_func OBJECT_OT_boolean.is_repeat()>
layout	NoneType: None
macros	bpy_prop_collection: <bpy_collection[0], OBJECT_OT_boolean.macros>
name	str: Boolean
options	OperatorOptions: <bpy_struct, OperatorOptions at 0x000000CFBC256B98>
properties	OBJECT_OT_boolean: <bpy_struct, OBJECT_OT_boolean at 0x000000CFC43F6628>
report	bpy_func: <bpy_func OBJECT_OT_boolean.report()>
rna_type	OBJECT_OT_boolean: <bpy_struct, Struct("OBJECT_OT_boolean")>

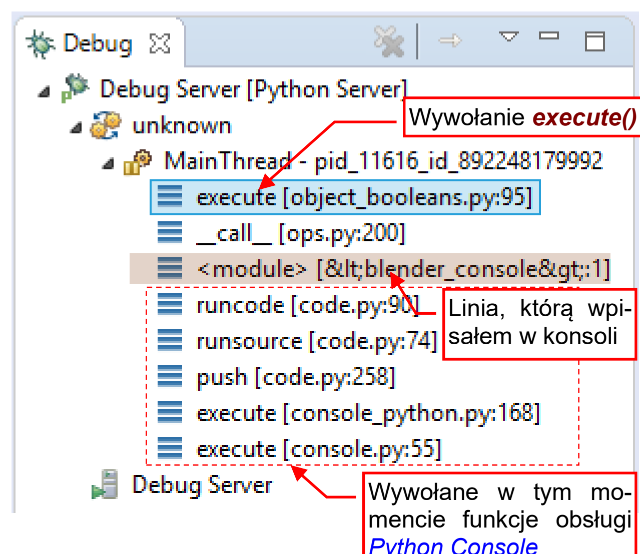
Zwróć uwagę, na inną wartość ***bl_idname***, powtórzoną w nazwie klasy bazowej

Rysunek 4.1.10 Podgląd zawartości naszej klasy

Od razu uspokajam: to normalne. Wygląda na to, że Blender kierując się oryginalną wartością ***bl_idname*** (**„object.boolean”**) stworzył na potrzeby naszego operatora bazową klasę ***OBJECT_OT_boolean***. (Słowo „**object**” pozostało, ale jest zapisane dużymi literami, a następująca po tym słowie kropka („**.**”) została zamieniona na symbol „***_OT_***”). Wyświetl zawartość przestrzeni nazw ***bpy.types*** (np. wpisz polecenie ***dir(bpy.types)*** w konsoli Pythona). Zobaczysz wtedy masę nie udokumentowanych klas, zawierających w nazwie „_OT_”, „_MT_”, albo „_PT_”. To wszystkie menu i panele GUI Blendera!

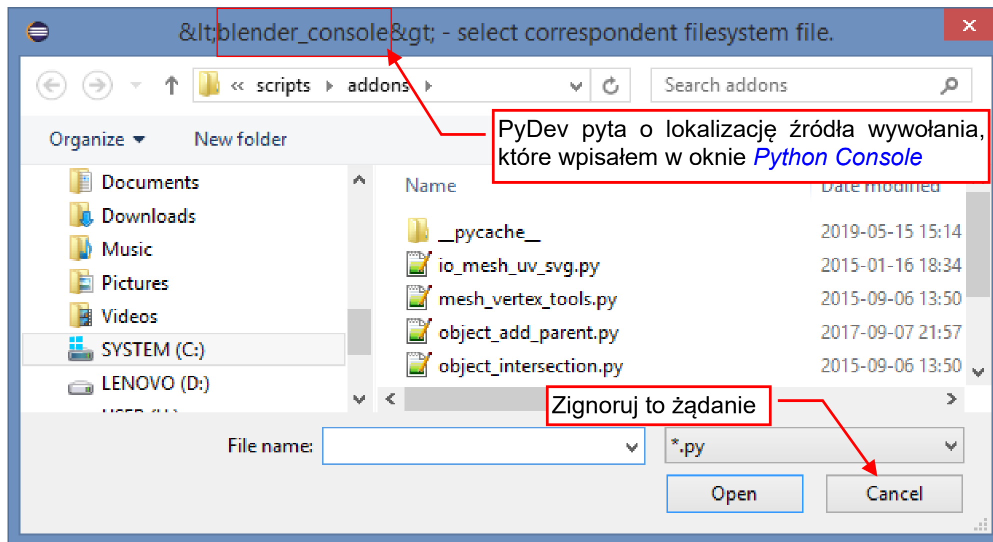
Przy okazji: zerknij także na aktualny stan stosu, na którym jest wykonywany nasz skrypt (Rysunek 4.1.11). Porównaj go np. ze stosem przedstawianym przez Rysunek 3.4.7 (str. 55), albo Rysunek 3.4.9 (str. 56).

U dołu stosu są funkcje obsługujące *Python Console* (jak widać, duża jej część jest także napisana w Pythonie). Potem jest wywołanie pierwszej linii w chwilowym skrypcie „*<blender console>*”. (Znaki „<>” są omyłkowo zmienione przez PyDev na „*<>*”). To polecenie, które wpisaliśmy. Jak widać, spowodowało wywołanie modułu *ops.py*, który z kolei stworzył instancję klasy ***Object_OT_Boolean*** i wywołał jej metodę ***execute()***.



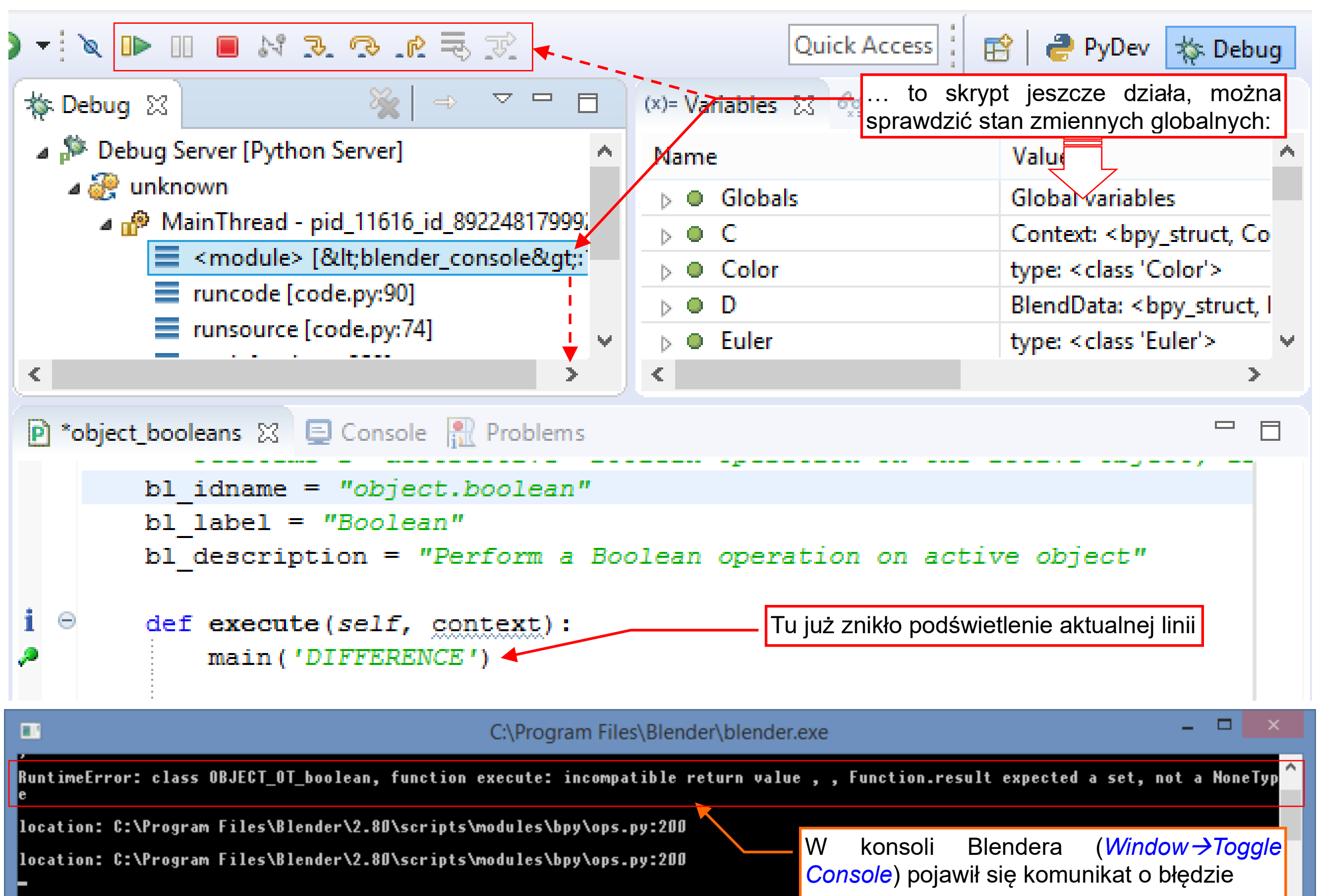
Rysunek 4.1.11 Stos wywołania z konsoli Pythona

Za pierwszym razem, gdy wykonasz ostatnią linię metody ***execute()*** (*Step Over* — F6) PyDev może zapytać się o plik źródłowy odpowiadający poleceniu z konsoli. Zignoruj takie pytanie, naciskając ***Cancel*** (Rysunek 4.1.12):



Rysunek 4.1.12 Okno, które może się pojawić po opuszczeniu procedury *execute()*

Opuściłem tę procedurę poleceniem *Step Over* by pokazać Ci zachowanie debugera PyDev w przypadku wystąpienia błędu we wtyczce. Z ekranu znikło podświetlenie aktualnie wykonywanej linii (Rysunek 4.1.13):



Rysunek 4.1.13 Stan debugera po wystąpieniu błędu

Jednocześnie w konsoli Blendera debuger wypisuje komunikat o błędzie oraz module i numerze linii, w której błąd wystąpił. Mimo to wykonywanie skryptu nie zostało jeszcze zakończone. W panelu *Debug* widzisz nadal zawartość stosu. W panelu *Variables* możesz sprawdzić aktualny stan zmiennych globalnych. Zazwyczaj dzięki analizie ich zawartości będziesz mógł szybko zdeterminować przyczynę problemu.

Copyright Witold Jaworski, 2011-2019.

W każdym razie, gdy chcesz zakończyć tak przerwany skrypt — naciśnij przycisk *Resume* (F8). Wtedy dopiero w oknie Blendera *Python Console* standardowy wydruk stosu wywołań (Rysunek 4.1.14):

```
Console   Autocomplete   Icon Viewer
Traceback (most recent call last):
  File "<blender_console>", line 1, in <module>
  File "C:\Program Files\Blender\2.80\scripts\modules\bpy\ops.py", line 200, in __call__
    ret = op_call(self.idname_py(), None, kw)
RuntimeError: Error: RuntimeError: class OBJECT_OT_boolean, function execute: incompatible return value
 , , Function.result expected a set, not a NoneType

location: C:\Program Files\Blender\2.80\scripts\modules\bpy\ops.py:200

>>>
```

Ten sam komunikat, który pojawił się wcześniej w konsoli Blendera

Rysunek 4.1.14 Komunikat o błędzie

- Komunikaty o błędzie polecenia wywołanego z konsoli Pythona (*Python Console*) pojawią się poniżej wywołania, tak jak pokazuje to Rysunek 4.1.14. Komunikaty o błędzie poleceń wywoływanych z GUI Blendera — menu, przycisku, itp. — pojawią się tylko w konsoli Blendera (*System Console*)[1].

W komunikacie na ilustracji powyżej Blendera wyraźnie napisał, że metoda ***execute()*** jest funkcją i powinna zwrócić zbiór (***set***), a nie ***None***. Istotnie, w pośpiechu pisania kodu zupełnie zapomniałem, że ***execute()*** musi zwracać jedną z wartości przewidzianego w API wyliczenia. Zazwyczaj chodzi jednoelementowy zbiór zawierający tekst **'FINISHED'**, albo **'CANCELLED'**. (Pełną listę możesz znaleźć w deklaracji klasy bazowej – ***bpy.types.Operator*** – w pliku nagłówków *bpy*). Poprawmy od razu nasz skrypt (Rysunek 4.1.15):

Quick Access | PyDev | Debug

Debug | Variables | Expressions

Debug Server [Python Server]
unknown
MainThread - pid_8144_id_503683834152
Debug Server

object_booleans | Console | Problems

```
#---------- ### Operator -----------
class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a 'destructive' Boolean operation on the active object,
    bl_idname = "object.boolean"
    bl_label = "Boolean"
    bl_description = "Perform a Boolean operation on active object"

    def execute(self, context):
        main('DIFFERENCE')
        return {'FINISHED'}
```

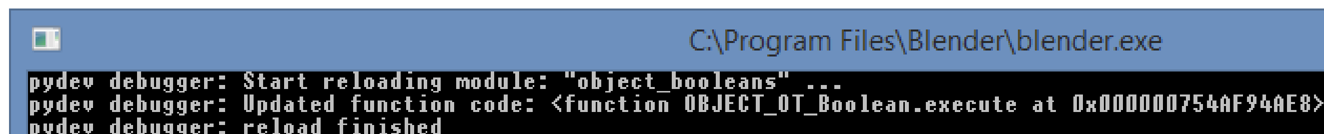
Funkcja ***execute()*** musi zwracać kolekcję z jedną z dopuszczalnych wartości wyliczenia

Rysunek 4.1.15 Szybka poprawka kodu — od razu, w perspektywie *Debug*

Zapisz tak zmodyfikowany plik.

[1] *System Console* to odrębne okno Windows, które działa przez całą sesję Blendera. Możesz je jedynie ukrywać/odsłaniać poleceniem *Window→Toggle System Console*. W Blenderze 2.8 wyłączono możliwość zamknięcia tego okna. (Możesz je tylko ukrywać i odsłaniać. W poprzednich wersjach Blendera, gdy użytkownik zamknął to okno konsoli – zamykał całego Bledera)

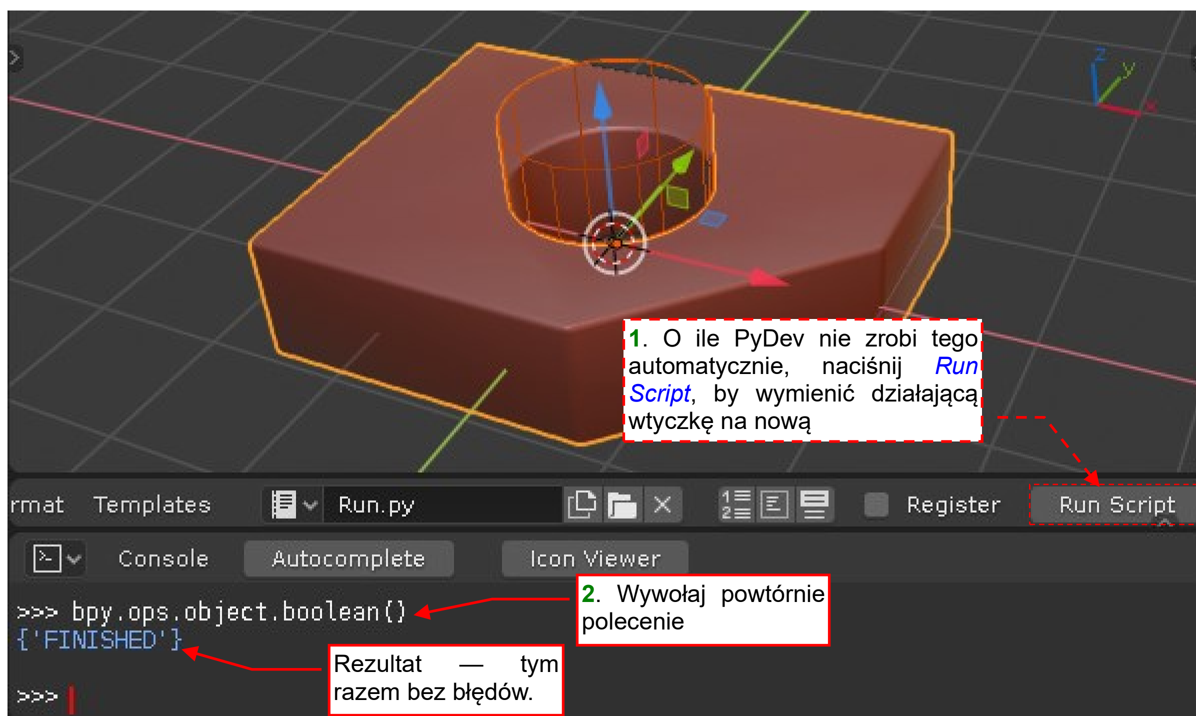
Po każdym zapisie pliku skryptu PyDev uaktualnia także kod modułu załadowany w Blenderze. Świadczą o tym komunikaty, które pojawią się w konsolach Blendera i Eclipse (Rysunek 4.1.16):



Rysunek 4.1.16 Samoczynna aktualizacja w Blenderze kodu załadowanego skryptu

Nadal możesz także klikać w ***Run Script***, aby np. wyrejestrować i ponownie zarejestrować wtyczkę.

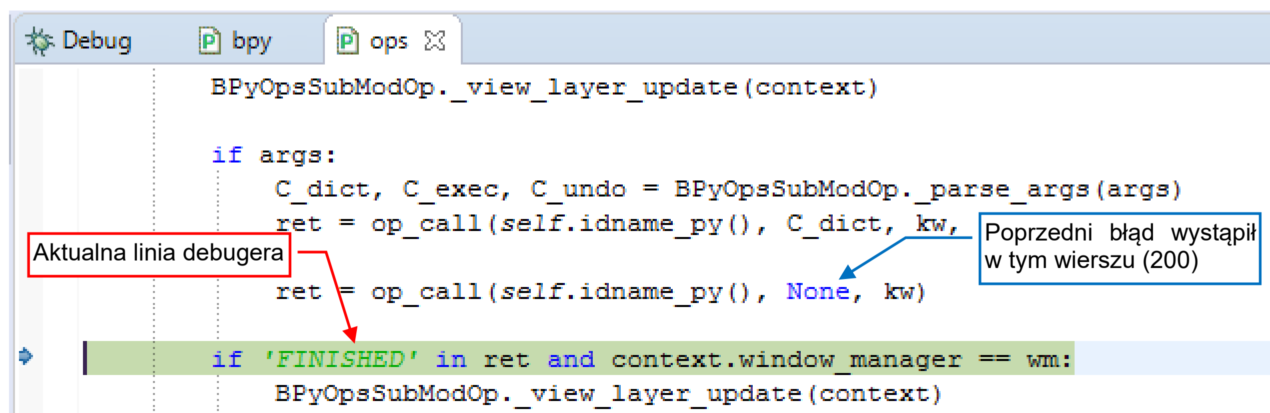
Gdy kod skryptu w Blenderze został uaktualniony, wywołaj powtórnie polecenie (Rysunek 4.1.17):



Rysunek 4.1.17 Powtórne uruchomienie polecenia

Jak widać, po wprowadzeniu poprawki nasz operator zakończył się bez błędu.

Jeżeli podczas śledzenia kodu przekroczysz ostatni wiersz skryptu poleceniem *Step Over* (**F6**) – znajdziesz się w pomocniczym module Blendera *ops.py* (Rysunek 4.1.18):



Rysunek 4.1.18 Standardowy moduł *ops.py*, otwierany po wykonaniu ostatniej linii skryptu

W takim przypadku po prostu zakończ wykonywanie skryptu poleceniem ***Resume*** (**F8**). (Ten moduł wywołuje naszą metodę ***execute()*** – por. Rysunek 4.1.11, str. 82).

Copyright Witold Jaworski, 2011-2019.

Na koniec wprowadźmy drobną poprawkę do kodu funkcji ***main()***: dodajmy możliwość skorzystania z kontekstu, który Blender przekazuje jako argument wywołania metody ***execute()***. Ten kontekst może się różnić od aktualnego (zwracanego przez ***bpy.context***). Np. specyfikacja API Blendera dopuszcza wywoływanie operatorów ze zmodyfikowanym kontekstem. Dodajmy więc do ***main()*** opcjonalny argument ***cntx*** (Rysunek 4.1.19):

```
def main (op, apply_objects=True, cntx=None):
    ''' Performs a Boolean operation on the active object, using the other
        selected objects as the 'tools'
        Arguments:
        @op (Enum): a Boolean operation: {'UNION', 'INTERSECT', 'DIFFERENCE'}
        @apply_objects (bool): apply results of to the mesh (optional)
        @cntx (bpy.types.Context): overrides current context (optional)
        @returns (list): one or two message parts: [<flag>, Optional_details]
    '''
    try:
        if cntx == None: cntx = bpy.context
        selected = list(cntx.selected_objects) #creates a static copy
        active = cntx.object #active object
        if active in selected: selected.remove(active)
        #input validation:
                    ...
            Pozostały kod funkcji
                    ...
```


Rysunek 4.1.19 Modyfikacja funkcji *main()* – dodanie opcjonalnego argumentu *context*.

Wymagało to zmian tylko w pierwszych kilku liniach funkcji. Następnie w klasie operatora przekażmy funkcji ***main()*** kontekst otrzymany jako argument metody ***execute()*** (Rysunek 4.1.20):

```
class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a Boolean operation on the active object '''
    bl_idname = "object.boolean"
    bl_label = "Boolean"
    bl_description = "Performs a Boolean operation on active object"

    def execute(self, context):
        main('DIFFERENCE', cntx = context)
        return {'FINISHED'}
```


Rysunek 4.1.20 Modyfikacja funkcji *execute()* – przekazanie argumentu *context*

W następnej sekcji pokażę, dodać wywołanie tego operatora do menu *Object*.

Podsumowanie

- Każda wtyczka musi zawierać słownik ***bl_info*** (str. 78). To jej „metryczka”, używana do wyświetlania informacji w oknie *User Preferences:Add-Ons*;
- Procedurę, która coś zmienia w danych Blendera (jak nasz ***main()***) należy przekształcić w operator. Polega to na stworzeniu klasy pochodnej ***bpy.types.Operator***. Procedurę należy wywołać w metodzie ***execute()*** tej nowej klasy (str. 78);
- Wtyczka musi implementować procedury ***register()*** i ***unregister()*** (str. 79);
- Przycisk ***Run Script*** służy wyłącznie do załadowania aktualnej wersji wtyczki. (Wywołuje procedurę ***unregister()*** dla starej wersji i ***register()*** dla nowej — por. str. 85, 160).
- Uruchomienie wtyczki oznacza tylko jej zarejestrowanie (wykonanie metody ***register()*** skryptu — str. 80). Operator, który implementuje, trzeba w jakiś sposób wywołać — np. z konsoli Pythona (str. 80 - 81). Powoduje to stworzenie przez Blender nowej instancji klasy operatora i wywołanie jego metody ***execute()***;
- Gdy po pierwszej sesji debugowania wprowadziłeś jakieś zmiany do skryptu i zapisałeś ten zmodyfikowany plik – PyDev próbuje go uaktualnić także w Blenderze Wyświetla o tym komunikaty w konsolach Bledera[1] (str. 85) i Eclipse. Jeżeli wynika z nich, że aktualizacja zakończyła się sukcesem – nie musisz przeładowywać skryptu przyciskiem *Run Script*;
- Gdy w programie wystąpi błąd (tzn. sygnalizowany jest wyjątek — *exception*), debuger Eclipse zatrzymuje wykonywanie kodu (str. 83). Można w tym momencie sprawdzić stan zmiennych globalnych. W konsoli Blendera można już zobaczyć komunikat o błędzie. Ten sam komunikat o błędzie zostanie wyświetlony w *Python Console* Blendera, gdy pozwolisz skryptowi „wywalić się do końca” (poleceniem *Resume* — str. 84);
- Informacji o otoczeniu (kontekście) wywołania operatora — selekcji, aktualnej scenie, itp. — dostarcza metodzie ***execute()*** argument ***context*** (str. 81);

[1] Konsola Blendera to tzw. *System Console*. Nie myl jej z *Python Console*! Przed uruchomieniem skryptu warto ustawić konsolę Blendera jako widoczną (poleceniem *Window→Toggle System Console*). (Po uruchomieniu skryptu Pythona w debugerze ekran ulega „zamrożeniu” i nie można już wywołać na nim żadnego polecenia z menu, dopóki nie zakończy się wykonywanie kodu).

Copyright Witold Jaworski, 2011-2019.

4.2 Dodanie polecenia (operatora) do menu

Analizując kod z poprzedniej sekcji mogłeś zauważyć, że w przypadku wtyczki API Blendera wymaga od Ciebie implementacji ściśle określonych metod. To taki „kontrakt" pomiędzy Twoim skryptem a resztą systemu. Ty zobowiązujesz się przygotować klasę o określonych polach i metodach. Blender zobowiązuje się wywoływać je w ściśle określonej sytuacji. Listę tak uzgodnionych funkcji i właściwości nazywa się w programowaniu obiektowym „interfejsem". Aby ułatwić Ci nieco zadanie, w Blender API znajduje się gotowy „wzorzec" klasy operatora: ***bpy.types.Operator***[1]. W języku programowania obiektowego ***Operator*** jest tzw. „klasą abstrakcyjną". Sama niczego specjalnego nie robi, dostarcza co najwyżej domyślnych, pustych implementacji wszystkich metod, przewidzianych w interfejsie wtyczki. Nasz operator dziedziczy te implementacje po klasie bazowej (właśnie ***bpy.types.Operator***). Dlatego możemy w naszej klasie ***OBJECT_OT_Boolean*** implementować (nadpisywać) tylko te z metod klasy ***Operator***, które nie mają działać w sposób domyślny.

Na razie nadpisaliśmy tylko jedną metodę: ***execute()***. Ignorujemy w niej rezultat zwracany przez wywoływaną wewnątrz funkcję ***main()***, choć przecież mogą się tam znajdować ewentualne komunikaty dla użytkownika. Robię tak, gdyż pewnych sytuacjach metoda ***execute()*** może być wołana wielokrotnie, raz za razem, dla tego samego kontekstu i różnych parametrów wejściowych. (Przekonasz się o tym w następnej sekcji). Dlatego nie należy wyświetlać w niej jakichkolwiek komunikatów. Lepszym miejscem jest inna metoda, przewidziana w interfejsie: ***Operator***: ***invoke()*** (Rysunek 4.2.1):

```
#result constants:
INPUT_ERR = 'ERROR_INVALID_CONTEXT'
ERROR  = 'ERROR'
WARNING = 'WARNING'
SUCCESS = 'OK'
```

Teksty tych stałych (używanych w pierwszym elemencie rezultatu zwracanego przez funkcję ***main()***) zmieniłem na wymagane w argumencie ***type*** metody ***bpy.types.Operator.report()***

Dotychczasowy kod programu

```
#--- ### Operator
class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a 'destructive' Boolean operation on the active object'''
    bl_idname = "object.boolean"
    bl_label = "Boolean"
    bl_description = "Perform a Boolean operation on active object"

    def execute(self, context):
        main('DIFFERENCE', cntx = context)
        return {'FINISHED'}

    def invoke(self, context, event):
        result = main('DIFFERENCE', cntx = context)
        if result[0] == SUCCESS:
            return {'FINISHED'}
        else:
            self.report(type = {result[0]}, message = result[1])
            return {'FINISHED' if result[0] == WARNING else 'CANCELLED'}
```

W pewnych sytuacjach Blender używa metody ***invoke()***, gdy źródłem wywołania jest polecenie menu lub przycisk.

Potem jednak może (nie musi!) wywołać ***execute()***.

Argumentu ***event*** w tym skrypcie nie będziemy używać. Jest potrzebny do operatorów modalnych.

Metoda ***report()*** wyświetla ewentualną wiadomość dla użytkownika

Rysunek 4.2.1 Komunikacja z użytkownikiem — procedura *invoke()*

Blender oczekuje od ***invoke()*** tego samego, co od ***execute()***: kodu informacji zwrotnej. Nasza implementacja sprawdza rezultat funkcji ***main()***. Jeżeli jest poprawny – zwraca **'FINISHED'** (także w przypadku ostrzeżeń). W przypadku błędów zwraca **'CANCELLED'**. Gdy jest do wyświetlenia jakiś komunikat – używam do tego funkcji ***report()***. Zwróć uwagę, że specjalnie zmieniłem stałe tekstowe rezultatu, aby je dopasować do argumentu ***type***.

[1] Oprócz interfejsu ***Operator***, API Blendera zawiera jeszcze dwa inne interfejsy (klasy abstrakcyjne): ***Menu*** i ***Panel***. Służą oczywiście do implementacji GUI. Wszystkie trzy znajdziesz w opisie modułu ***bpy.types***, a także w podpowiedziach uzupełniania kodu.

Metoda ***invoke()*** otrzymuje oprócz kontekstu (***context***) także drugi argument, o nazwie ***event***. To informacja o „wydarzeniu" — przesunięciu myszki i stanie klawiatury. Czasami może się to przydać (patrz przykłady w opisie klasy Operator). Jednak w kodzie tej wtyczki argument ***event*** nigdy nie będzie używany.

Operator, który przygotowaliśmy, będzie działać wyłącznie w trybie interakcji *Object Mode*. Do tej pory nigdzie nie sprawdzaliśmy, czy istotnie taki jest aktualny tryb Blendera, traktując to jak coś oczywistego. Nigdy jednak nie możesz być pewnym, czy ktoś w przyszłości nie wywoła Twojego polecenia w niewłaściwym trybie. Dlatego zawsze warto umieścić w klasie każdego operatora metodę ***poll()*** (Rysunek 4.2.2):

```
#--- ### Operator
class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a 'destructive' Boolean opera
    bl_idname = "object.boolean"
    bl_label = "Boolean"
    bl_description = "Perform a Boolean opera

    @classmethod
    def poll(cls, context):
        return (context.mode == 'OBJECT')

    def execute(self, context):
        main('DIFFERENCE', cntx = context)
        return {'FINISHED'}
```

Blender używa metody ***poll()***, aby sprawdzić, czy w danym momencie polecenie może być użyte. Jeżeli nie — po prostu nie jest wyświetlane

poll() musi być zadeklarowana jako metoda klasy, a nie instancji

Procedura zwraca *False*, gdy nie jesteśmy w *Object Mode*. W efekcie polecenie ***Boolean*** będzie się pojawiać w menu tylko w tym trybie

Rysunek 4.2.2 Dodanie podstawowego „testu na widoczność" — procedury *poll()*

Blender wywołuję tę funkcję, gdy się dowiedzieć czy „w obecnej sytuacji" polecenie może być w ogóle wyświetlone. Kod ***poll()*** po analizie otrzymanego obiektu ***context*** ma zwrócić *True*, jeżeli operator może być użyty. W przeciwnym razie powinien zwrócić *False*.

W tym miejscu raczej należy umieszczać „zgrubne" testy, właśnie takie, jak pokazano na ilustracji powyżej. W naszej implementacji funkcja ***poll()*** zwraca *True*, jeżeli aktualnym trybem jest *Object Mode*. (Takie znaczenie ma stała **'OBJECT'**). Gdybyśmy byli w trybie edycji np. armatury, ***context.mode*** zwróciłby inną wartość.

Nie sprawdzaj w metodzie ***poll()*** żadnych warunków w rodzaju „czy wszystkie zaznaczone obiekty to siatki". Są zbyt szczegółowy. Pomyśl, co to za polecenie, które pojawiałoby się w menu tylko wtedy, gdy zaznaczyłeś obiekt? Połowa użytkowników nie miałaby szczęścia go w ogóle zobaczyć, i doszłaby do wniosku, że wtyczka nie działa. Już lepiej pozwolić, aby polecenie *Boolean* było widoczne w menu *Object* przez cały czas. Jeżeli przed jego wywołaniem użytkownik nie zaznaczył żadnych obiektów, lepiej wyświetlić o tym odpowiedni komunikat. W ten sposób będzie wiedział, co powinien zrobić następnym razem.

- W kodzie metody ***poll()*** nie umieszczaj nigdy żadnych poleceń, które coś zmieniają w Blenderze. (Chodzi np. o zmianę trybu pracy, albo zawartości sceny). Program nie pozwoli im się w tym miejscu wykonać

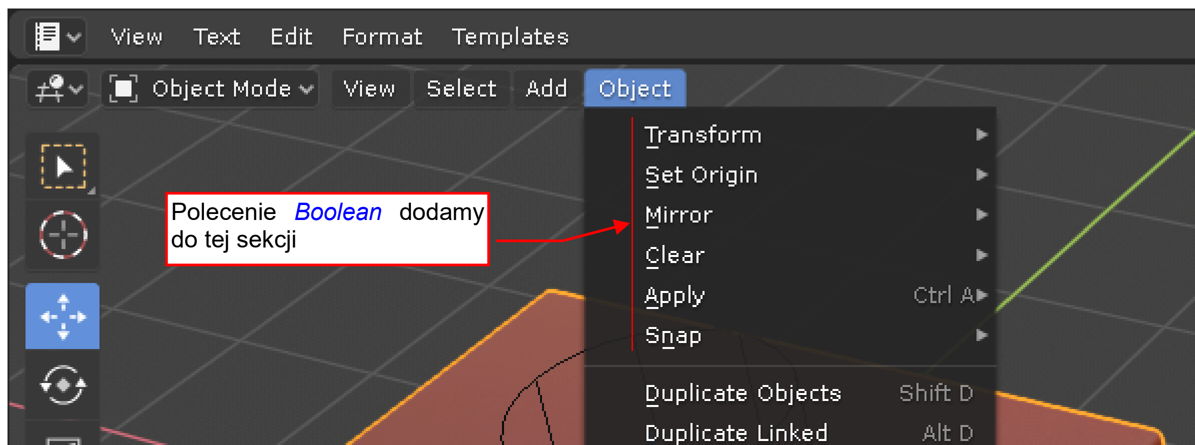
Zwróć uwagę na wyrażenie *@classmethod*, poprzedzający definicję metody ***poll()***. (W slangu programistów to tak zwany „dekorator"). To deklaracja, że metoda jest metodą klasy, a nie obiektu (instancji)[1].

- Pamiętaj, żeby nigdy nie zapomnieć o wpisaniu „dekoratora" *@classmethod* przed definicją funkcji ***poll()***! Jeżeli go pominiesz, Blender nigdy nie wywoła tej metody.

[1] To zapewne dla poprawienia wydajności pracy całego środowiska Blendera. Metodę ***poll()*** implementują wszystkie elementu GUI, a system co chwila je wywołuje. (Funkcje ***poll()*** są wywoływane, gdy cokolwiek ulega zmianie — tryb pracy, zawartość obiektu, itp.). Gdyby ***poll()*** była zwykłą metodą obiektu, tak jak ***execute()***, Blender musiałby za każdym razem tworzyć nowe instancje klasy, wywołać ich funkcje ***poll()***, i zaraz potem je zwalniać. Przypuszczam, że wtedy wszystko działałoby wolniej, być może nawet zbyt wolno. Do wywołania metody klasy nie trzeba tworzyć instancji (nowych obiektów), i w związku z tym nie obciąża to tak bardzo procesora.

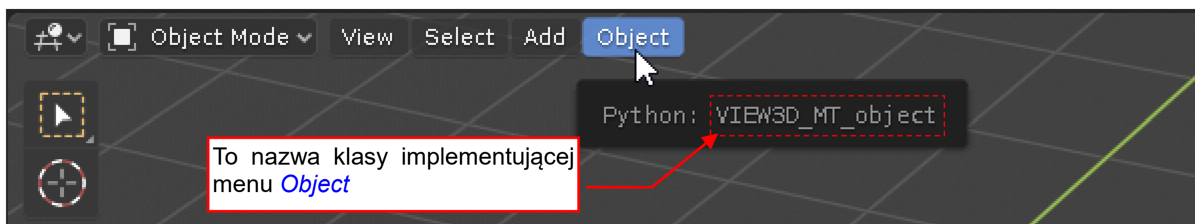
Copyright Witold Jaworski, 2011-2019.

No dobrze, to mamy ulepszony, gotowy do użycia operator. Ale jak go dodać do standardowego menu Blendera, takiego jak ***Object*** (Rysunek 4.2.3)?



Rysunek 4.2.3 Menu *Object*. To do niego dodamy polecenie *Boolean*.

Standardowe menu Blendera są tworzone w ten sam sposób, w jaki Twój dodatek dopisze swoje: za pomocą API. Trzeba tylko odkryć, jak się nazywa klasa implementująca menu *Object*. Pomogą tu niezastąpione *Python tooltips*. Wystarczy przez chwilę zatrzymać myszkę nad jego etykietą (Rysunek 4.2.4):



Rysunek 4.2.4 Identyfikacja nazwy klasy menu *Object*

Gdy poznałem już nazwę klasy menu *Object*, dodałem do niej nasz operator *Boolean* (Rysunek 4.2.5):

Pomocnicza funkcja: umieszczona w kodzie głównym, ale dodawana do klasy implementującej menu (stąd jej pierwszy argument to ***self***)

```
def menu_draw(self, context):
    self.layout.operator_context = 'INVOKE_REGION_WIN'
    self.layout.operator(OBJECT_OT_Boolean.bl_idname)

def register():
    register_class(OBJECT_OT_Boolean)
    bpy.types.VIEW3D_MT_object.prepend(menu_draw)

def unregister():
    bpy.types.VIEW3D_MT_object.remove(menu_draw)
    unregister_class(OBJECT_OT_Boolean)
```

Jeżeli wpiszesz tę linię, Blender będzie wywoływał metodę ***invoke()*** zamiast ***execute()***

Dodanie i usunięcie polecenia z menu

Rysunek 4.2.5 Obsługa rejestracji operatora w menu *Object*

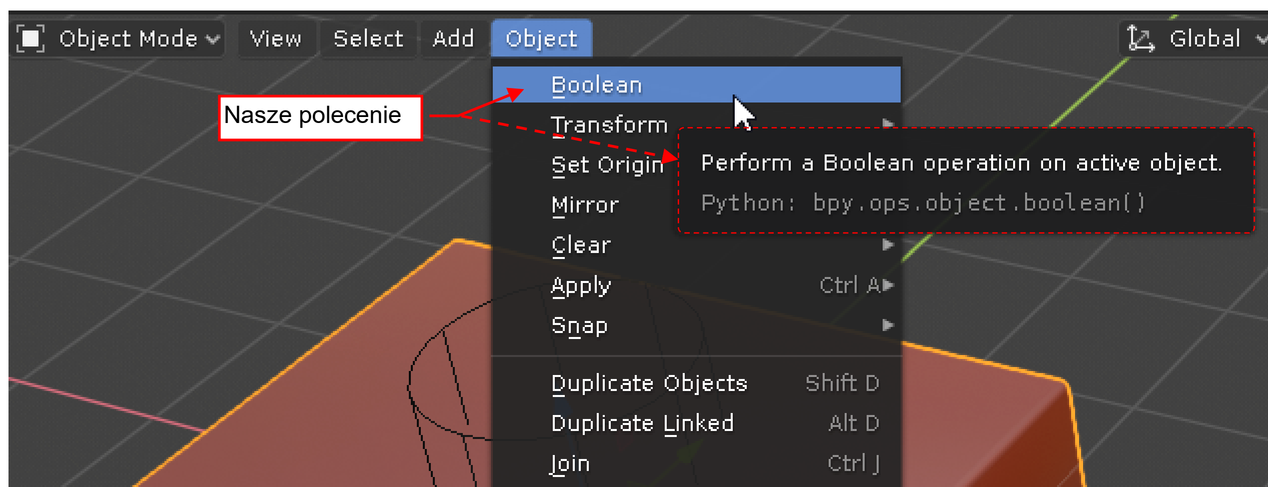
Każde menu w Blenderze pochodzi od klasy bazowej ***bpy.types.Menu***. W procedurze ***register()*** wywołuję dla klasy menu *Object* metodę ***prepend()***. To dołącza zdefiniowaną powyżej metodę rysowania pozycji (***menu_draw()***) na początek menu. W procedurze ***unregister()*** usuwam tę metodę.

A co zawiera metoda ***menu_draw()***? Argument ***self*** to instancja obiektu menu *Object*. Każde menu zawiera pole ***layout***, które zwraca obiekt klasy ***bpy.types.UILayout***. To reprezentacja zawartości („powierzchni") menu. Metoda ***operator()*** dodaje do menu nowe polecenie (przekazuję mu id operatora). Przedtem jednak zmieniam wartość pola ***operator_context***, aby wywołane z menu polecenia pokazywało użytkownikowi ewentualne komunikaty wysyłane przez funkcję ***Operator.report()*** (por. Rysunek 4.2.1, str. 88).

Gdybym w funkcji ***register()*** użył metody ***bpy.types.Menu.append()*** – nowa pozycja zostałaby dołączona do końca menu *Object*. Blender API nie przewiduje metod na dołączenie nowej pozycji gdzieś w środku menu.

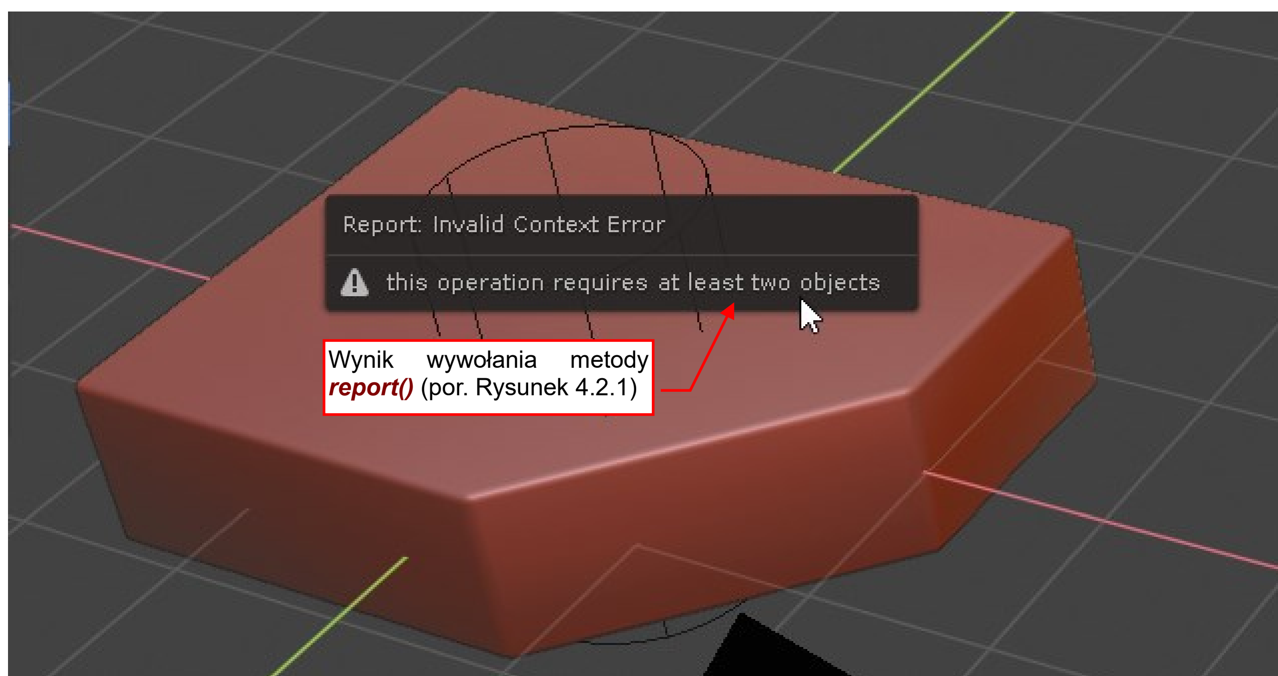
- Aby zidentyfikować nazwy klas takich „ulotnych” menu jak np. menu kontekstowe, musisz je znaleźć w źródłowym pliku Blendera. Definicje całego interfejsu użytkownika są napisane w Pythonie. Znajdziesz je w podkatalogu *scripts\startup\bl_ui*. (Np. u mnie to *C:\Program Files\Blender\2.80\scripts\startup\bl_ui*). Pliki o nazwach *space_<nazwa okna>.py* zawierają definicje menu poszczególnych typów okien. Stąd wszystkie menu okna *View 3D* znajdziesz w pliku *space_view3d.py*. Otwórz ten plik w edytorze i spróbuj w nim wyszukać nazwę jakiegoś specyficznego polecenia (nie submenu!) z poszukiwanego menu. (Czasami możesz jej nie znaleźć, gdy jest to domyślna nazwa operatora. Wtedy poszukaj nazwy innej pozycji).

Zobaczmy teraz, czy nasz kod działa: załaduj ponownie skrypt (przyciskiem *Run Script*). W wyniku wykonania funkcji ***register()*** wywołanie naszego operatora pojawiło się na początku menu *Object* (Rysunek 4.2.6):



Rysunek 4.2.6 Wywołanie naszego operatora w menu *Object*.

Wykonajmy jeszcze test „zerowy” — wywołanie polecenia, gdy nie ma na scenie zaznaczonych obiektów. Rysunek 4.2.7 pokazuje, że uzyskaliśmy oczekiwany wynik – komunikat wyświetlany przez funkcję ***report()***:



Rysunek 4.2.7 Rezultat wywołania polecenia, gdy na scenie nie ma zaznaczonych obiektów

Jednocześnie w oknie *Info* pojawił się ten sam komunikat na czerwonym tle. (Ale który z użytkowników do tego okna zagląda?) W przypadku ostrzeżeń komunikat pojawia się wyłącznie w oknie *Info*, na pomarańczowym tle.

Copyright Witold Jaworski, 2011-2019.

Jak do tej pory, nasze polecenie *Boolean* wykonywało tylko różnicę obiektów. Czas dodać do tego operatora parametr, który pozwoli na wybór operacji (Rysunek 4.2.8):

```
from bpy.props import EnumProperty

class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a 'destructive' Boolean operation on the active object
       Arguments:
       @op (Enum): operation type, in ['DIFFERENCE', 'UNION', 'INTERSECT']
    '''
    bl_idna
    bl_labe
    bl_desc                                  Boolean operation on active object"

    op :  EnumProperty(items = [
                        ('DIFFERENCE',"Difference","Boolean difference"),
                        ('UNION',"Union","Boolean union"),
                        ('INTERSECT',"Intersection","Boolean intersection"),
                        ],
                name = "Operation",
                description = "Boolean operation",
                default='DIFFERENCE',
            ) #end EnumProperty

    @classmethod
    def poll(cls, context):
        return (context.mode == 'OBJECT')

    def execute(self, context):
        main(self.op, cntx = context)
        return {'FINISHED'}

    def invoke(self, context, event):
        result = main(self.op, cntx = context)
        if result[0] == SUCCESS:
            return {'FINISHED'}
        else:
            self.report(type = {result[0]}, message = result[1])
            return {'FINISHED' if result[0] == WARNING else 'CANCELLED'}
```

Klasy dla wszystkich typów argumentów (właściwości) operatora znajdują się w ***bpy.props***

Nowy argument operatora: enumeracja o nazwie ***op***

Wartość

Nazwa

Opis

Definicje wartości tej enumeracji

Tu zawsze wpisuj ":"

Ustalenie wartości domyślnej

Odwołanie do aktualnej wartości argumentu ***op***

Rysunek 4.2.8 Dodanie argumentu do operatora

Argument operatora to po prostu kolejne pole tej klasy, które należy przypisać dwukropkiem („:") do odpowiedniej funkcji API. Deklaracje tych funkcji – ***StringProperty***, ***IntProperty***, ***FloatProperty***, ***BoolProperty***, … - znajdują się w module ***bpy.props***. (Dla każdego podstawowego typu danych API przygotowano tam odpowiednią funkcję ****Property***). Pole (argument) ***op*** ma być enumeracją, więc z modułu ***bpy.props*** importuję funkcję ***EnumProperty***.

Najważniejszą częścią tego nowego kodu jest definicja enumeracji, przekazywana funkcji tworzącej tę nową właściwość (Rysunek 4.2.8). W pierwszym argumencie – ***items*** – przekazuję listę z deklaracją enumeracji. Każdy z jej elementów musi zawierać co najmniej: wartość (np. **'DIFFERENCE'**), nazwę (wyświetlaną w GUI) oraz opis. Z pozostałych argumentów tej funkcji warto jeszcze użyć ***default***, aby ustalić wartość domyślną tego argumentu operatora.

- W elemencie listy ***items*** można podać jeszcze dwie kolejne wartości: nazwę ikony oraz numer id. Pokażę to w jednej z dalszych sekcji.

Mimo tak nietypowej inicjalizacji, do wartości ***op*** można się odwoływać jak do każdego innego pola obiektu. Na ilustracji powyżej używam go w metodach ***invoke ()*** i ***execute()***, jako pierwszego argumentu funkcji ***main()***.

Załaduj teraz te nową wersję kodu – koniecznie klikając w przycisk ***Run Script***, aby ją przerejestrować. Gdy wpiszesz teraz w konsoli nazwę tego operatora, zobaczysz że ma argument (Rysunek 4.2.9):

Console Autocomplete Icon Viewer

```
>>> bpy.ops.object.boolean
bpy.ops.object.boolean(op='DIFFERENCE')

>>>
```

Dodany argument ***op***

Rysunek 4.2.9 Aktualne wywołanie operatora *object.boolean*

Możesz go wywołać z *Python Console*, wpisując np. ***bpy.ops.object.boolean(op = 'UNION')***.

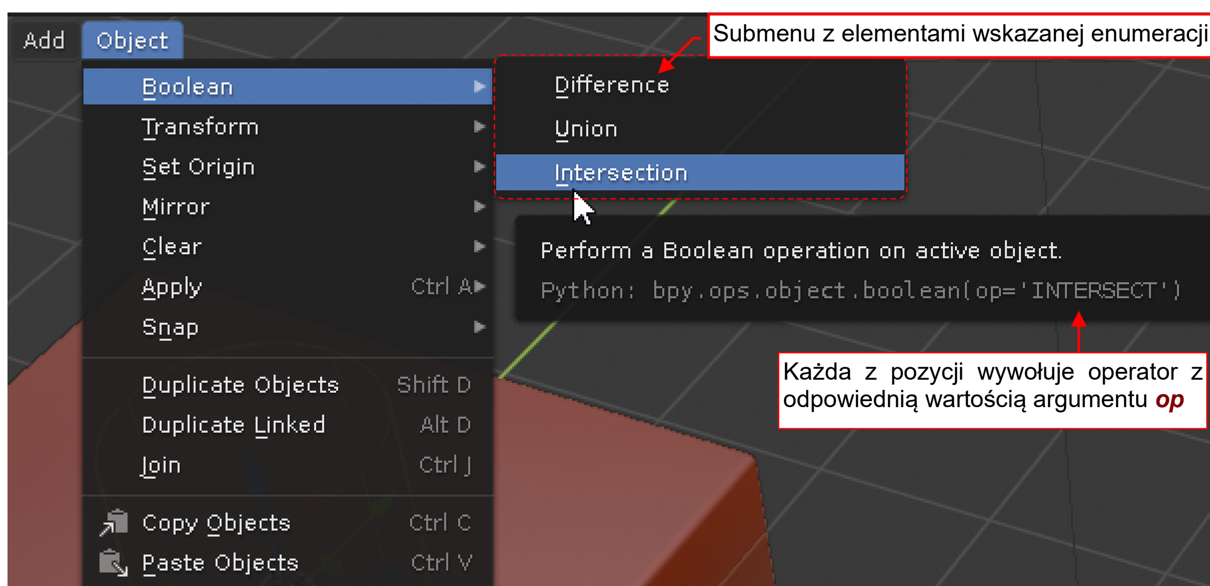
Bardzo łatwo jest także przekształcić nasze polecenie *Boolean* z menu *Object* w w odpowiednie submenu, w którym każda z pozycji będzie wywoływać jeden z dostępnych trybów operatora. Wystarczy w tym celu zmienić dosłownie jedna linię w procedurze ***menu_draw()*** (Rysunek 4.2.10):

```
def menu_draw(self, context):
    self.layout.operator_context = 'INVOKE_REGION_WIN'
    self.layout.operator_menu_enum(OBJECT_OT_Boolean.bl_idname, property="op")
```

Ta metoda dodaje submenu z elementami wskazanej enumeracji

Rysunek 4.2.10 Wstawienie submenu z opcjami operatora

Wystarczy metodę ***operator()*** zastąpić metodą ***operator_menu_enum()***, w której oprócz nazwy operatora należy podać także nazwę enumeracji. W efekcie w miejsce pojedynczego polecenia pojawi się submenu z jego opcjami (Rysunek 4.2.11):



Rysunek 4.2.11 Submenu z opcjami operatora

Sprawdzając pozycje tego menu za pomocą *Python tooltips* zobaczysz, że każda z nich wywołuje operator z odpowiednią wartością argumentu ***op***.

Jak widać powyżej, nasz dodatek już stał się użytecznym narzędziem. W nastepnej sekcji wzbogacimy go o możliwość interakcji z użytkownikiem.

Copyright Witold Jaworski, 2011-2019.

Podsumowanie

- Oprócz podstawowej metody ***execute()***, każdy operator implementuje także metodę: ***invoke()*** (str. 88). Wszelkie ewentualne komunikaty do użytkownika należy umieszczać w metodzie ***invoke()***. Najlepiej to zrobić za pomocą standardowej metody ***bpy.types.Operator.report()***.
- Aby operator wyświetlał ewentualne informacje o błędach w oknach dialogowych a nie w mało widocznej konsoli, należy przestawić sposób wywoływania operatora w menu z metody ***.execute()*** (domyślnej) na ***.invoke()***. W tym celu przed dodaniem operatora do menu przestawiam jego kontekst (pole ***.layout.operator_context***) na **'INVOKE_REGION_WIN'** (str. 90);
- Plecenie/polecenia implementowane przez wtyczkę dodajemy do menu Blendera w metodzie ***register()***, a usuwamy — w ***unregister()*** (str. 90). Aby napisać ten fragment kodu, musimy znać nazwę klasy Pythona, implementującej menu;
- Aby odnaleźć nazwę klasy standardowego menu, zazwyczaj wystarczy wykorzystać *Python Tooltips* (str. 90). Tylko w szczególnych przypadkach (np. nazwy klas menu kontekstowych) trzeba zajrzeć do skryptów, implementujących GUI Blendera (w plikach *space_<nazwa okna>.py*, znajdujących się w podkatalogu *<wersja Blendera>\scripts\startup\bl_ui*);
- Operator można „wyposażyć" w opcjonalną metodę ***poll()***. Ta funkcja jest używana przez Blender do sprawdzania, czy w aktualnym kontekście operator w ogóle ma być dostępny. (Na przykład —. widoczny w menu). Możesz w niej sprawdzać aktualny tryb pracy programu (str. 89);
- Argument operatora należy deklarować jako nowe pole klasy, i przypisać mu jedną z funkcji z modułu ***bpy.props*** (str. 92). W kodzie można je później wykorzystywać jak każde inne pole klasy;
- Argument operatora zadeklarowany jako enumeracja można bardzo łatwo zamienić w submenu (str. 93);

4.3 Implementacja interakcji z użytkownikiem

Implementacja interakcji wtyczki z użytkownikiem w Blenderze jest prosta, przynajmniej jeżeli chodzi o realizację pewnego podstawowego schematu. Pozwala on użytkownikowi dynamicznie zmieniać (np. ruchem myszki) parametry naszego polecenia i obserwować na bieżąco rezultat na ekranie.

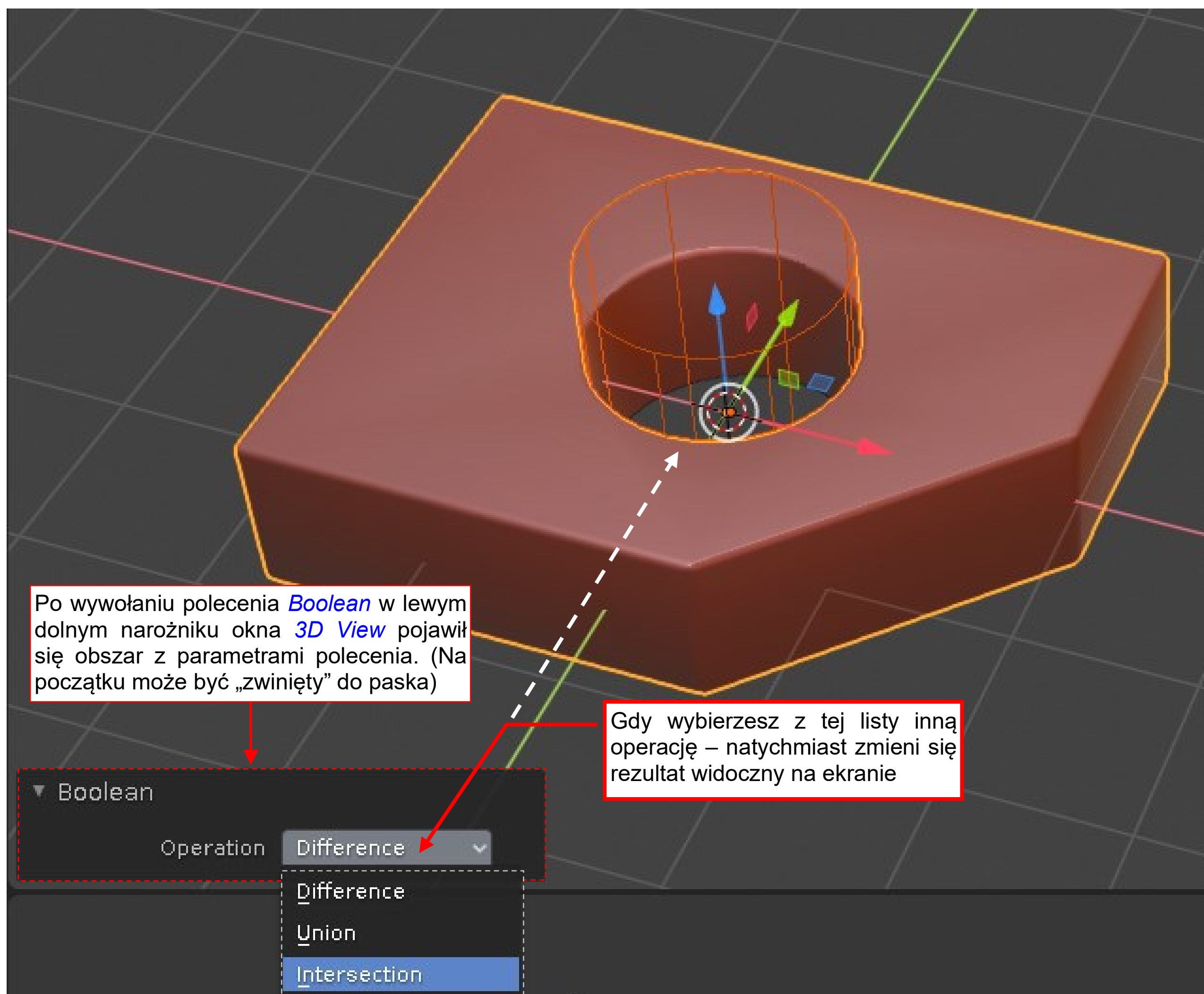
Wszystko, co należy zrobić, to nadpisać domyślne opcje operatora (pole ***bpy.types.Operator.bl_options***) w wartościami **'REGISTER'** i **'UNDO'**, tak jak to przedstawia Rysunek 4.3.1:

```
class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a 'destructive' Boolean operation on the active object
       Arguments:
       @op (Enum): Boolean operation, in ['DIFFERENCE', 'UNION', 'INTERSECT']
    '''
    bl_idname = "object.boolean"
    bl_label = "Boolean"
    bl_description = "Perform a Boolean operation on active object"
    bl_options = {'REGISTER', 'UNDO'}
```

Dopisz tę linię (włącza dodatkowe opcje operatora)

Rysunek 4.3.1 Zmiany w definicji klasy

Wpisz dokładnie podaną kombinację. Jeżeli przypiszesz samą wartość **'REGISTER'** albo samo **'UNDO'**, nie uzyskasz efektu, który pokazuje Rysunek 4.3.2:



Rysunek 4.3.2 Interaktywna zmiana parametru polecenia

To przybornik z parametrami operatora – taki sam jak w standardowych poleceniach Blendera.

Copyright Witold Jaworski, 2011-2019.

Gdy wywołasz naszą wtyczkę (powiedzmy że wybrałeś *Object→Boolean→Difference*), to w płytce powstanie otwór, tak jak poprzednio. Zauważ jedna, że jednocześnie w lewym dolnym narożniku aktywnego okna pojawił się pasek z opcjami naszego operatora (*Boolean*). Gdy rozwiniesz ten pasek, zobaczysz panel z prametrami polecenia — w naszym przypadku na razie typ operacji Boole'a. Gdy wybierzesz z tej listy nową wartość, zmienisz natychmiast widoczny na ekranie rezultat tego polecenia. O ile Twoja wtyczka nie wykonuje jakichś bardzo obciążających komputer działań, te zmiany zachodzą bardzo szybko. Gdybyś zmieniał w ten sposób jakiś wymiar liniowy – na przykład średnicę otworu, lub coś podobnego – przesuwając wartość jego pola myszką, widziałbyś płynnie w oknie *3D View* zmiany zachodzące w wybranych obiektach. (Akurat w tym przykładzie nie mam odpowiedniego pola do zademonstrowania tego efektu).

Jak Blender uzyskuje ten efekt? Do wyśledzenia takich interaktywnych zdarzeń czasami najlepiej się nadaje nie debugowanie, a prosty wydruk jakiegoś tekstu w konsoli programu. Umieśćmy na chwilę odpowiednie polecenia ***print()*** w obydwu procedurach operatora: ***invoke()*** i ***execute()*** (Rysunek 4.3.3):

```
    def execute(self, context):
        print("in execute() : op = '%s'" % self.op)
        main(self.op, cntx = context)
        return {'FINISHED'}

    def invoke(self, context, event):
        print("in invoke() : op = '%s'" % self.op)
        result = main(self.op, cntx = context)
        if result[0] == SUCCESS:
            return {'FINISHED'}
        else:
            self.report(type = {result[0]}, message = result[1])
            return {'FINISHED' if result[0] == WARNING else 'CANCELLED'}
```


Rysunek 4.3.3 Dodanie do kodu komunikatów diagnostycznych (tylko na chwilę)

Załaduj tę nową wersję wtyczki i jeszcze raz wywołaj *Object→Boolean→Difference* (Rysunek 4.3.4):

```
Debug Server
in invoke() : op = 'DIFFERENCE'
in execute() : op = 'UNION'
in execute() : op = 'INTERSECT'
in execute() : op = 'DIFFERENCE'
```


Rysunek 4.3.4 Komunikaty diagnostyczne podczas zmian parametru polecenia

Bezpośrednio po tym wywołaniu w konsoli Blendera pojawiła się pierwsza linia (Rysunek 4.3.4). Wygląda na to, że została tu wywołana metoda ***invoke()***. Teraz zacznij zmieniać wartość kontrolki *Operation* w przyborniku *Boolean*. Każda z tych zmian wywoływała metodę ***execute()***, z odpowiednią wartością argumentu ***op***.
Wygląda na to, że po każdym moim kliknięciu Blender wykonywał *Undo*, a następnie po prostu ponownie wołał operator. Tyle, że tym razem uruchamiał bezpośrednio jego metodę ***execute()***, dla odpowiedniej wartości parametru ***op***, odczytanego z przybornika.

Myślę, że podział ról pomiędzy procedurami ***invoke()*** i ***execute()*** można przedstawić następująco:

- Procedura ***invoke()*** jest wołana, gdy operator ma zostać wykonany z domyślnymi wartościami swoich parametrów. Procedura ***execute()*** jest wykonywana dla konkretnych wartości parametrów (podanych jawnie w liście argumentów wywołania).

Wyborem wywoływanej przez GUI metody można sterować np. za pomocą pewnych flag (por. str. 90).

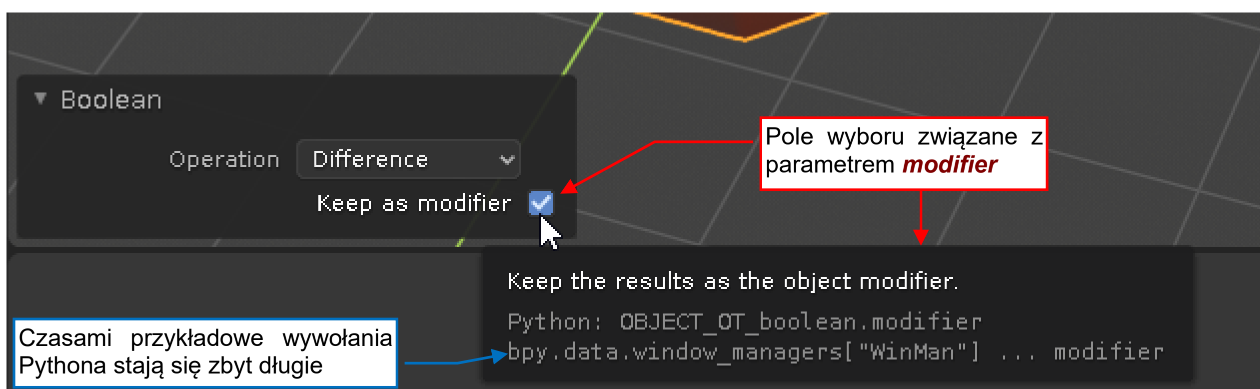
Oczywiście, domyślnie każdy argument operatora pojawi się w przyborniku. Dla przykładu dodam do naszego operatora pole wyboru ***modifier***, pozwalające zachować rezultat operacji w postaci modyfikatora obiektu (Rysunek 4.3.5):

```
from bpy.props import EnumProperty, BoolProperty

class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a 'destructive' Boolean operation on the active object
       Arguments:
       @op (Enum): Boolean operation, in ['DIFFERENCE', 'UNION', 'INTERSECT']
       @modifier (Bool): add this operation as the object modifier
    '''
    
    Dotychczasowy kod klasy

    modifier : BoolProperty(name = "Keep as modifier",
                            description = "Keep the results as the object modifier",
                            default = False
                           ) #end BoolProperty

    @classmethod
    def poll(cls, context):
        return (context.mode == 'OBJECT')

    def execute(self, context):
        main(self.op, apply_objects = not self.modifier, cntx = context)
        return {'FINISHED'}

    def invoke(self, context, event):
        result = main(self.op, apply_objects = not self.modifier, cntx = context)
        if result[0] == SUCCESS:
            return {'FINISHED'}
        else:
            self.report(type = {result[0]}, message = result[1])
            return {'FINISHED' if result[0] == WARNING else 'CANCELLED'}
```

To prosty przełącznik Tak/Nie, więc używam tu funkcji ***BoolProperty***

Rysunek 4.3.5 Kolejny argument operatora (pole wyboru)

Rysunek 4.3.6 przedstawia nową postać przybornika. Pojawiła się kontrolka argumentu ***modifier***:



Rysunek 4.3.6 Zmodyfikowany przybornik polecenia

Kontrolki w przyborniku są dodawane w pojedynczej kolumnie (jedna pod drugą) w takiej kolejności, w jakiej je zadeklarowałeś w kodzie klasy. Zazwyczaj efekt jest akceptowalny. W przykładzie na ilustracji powyżej wizualne przesunięcie pola *Keep as modifier* do prawej wynika z przyjętych w Blenderze konwencji rysowania poszczególnych elementów GUI[1].

[1] Jeżeli jednak chcesz zupełnie „przemeblować" te przybornik – nadpisz metodę ***bpy.types.Operator.draw()***. W jej wnętrzu sam możesz ustalić, jak mają być rysowane kontrolki w panelu tego narzędzia.

Copyright Witold Jaworski, 2011-2019.

Jeżeli tego potrzebujesz, możesz oznaczyć argument operatora jako niewidoczny w przyborniku. Służy do tego opcja **'HIDDEN'** (Rysunek 4.3.7):

```
    modifier : BoolProperty(name = "Keep as modifier",
                     description = "Keep the results as the object modifier",
                     default = False,
                     options = {'HIDDEN'},
                    ) #end BoolProperty
```

Opcja wyłącza wyświetlanie kontrolki dla tego pola operatora

Rysunek 4.3.7 Opcja ukrywająca kontrolkę argumentu w przyborniku

Blender zastosuje ostatnie ustawienia z przybornika podczas kolejnego wywołania operatora. Oczywiście oprócz tych argumentów, które są przekazywane w sposób jawny, jak parametr ***op*** (*Operation*). Ta „pamięć" trwa tylko do końca aktualnej sesji Blendera. Zazwyczaj jednak to w zupełności wystarcza. Jeżeli chcesz, aby przy każdym wywołaniu operatora argument nie podany w sposób jawny miał wartość domyślną (taką jak podana w kodzie) – dodaj do zestawu opcji kontrolki drugą wartość: **'SKIP_SAVE'**.

Podsumowanie

- Jeżeli do klasy dodamy pole **bl_options** = {**'REGISTER'**,**'UNDO'**}, wówczas nasze polecenie stanie się „interaktywne". Gdy je wywołasz, w aktywnym oknie *3D View* pojawi się przybornik (*Tool Properties*) a w nim kontrolki, odpowiadające parametrom (*properties*) operatora (str. 95). Można je tutaj dynamicznie zmieniać, np. za pomocą myszki. Każdej zmianie towarzyszy odpowiednia aktualizacji rezultatu działania operatora. Możesz ją na bieżąco śledzić na ekranie;
- Gdy klikasz w przycisk polecenia lub wybierasz je z menu, Blender wywołuje metodę ***invoke()*** wybranego w ten sposób operatora. Po każdej zmianie parametrów w przyborniku polecenia najpierw jest wywoływane polecenie *Undo*, a następnie metoda ***execute()*** (str. 96).
- Za pomocą opcji (argument ***options***) przypisanych do parametru operatora można go ukrywać w przyborniku narzędzia (**'HIDDEN'**) lub wyłączyć zapamiętywanie ostatniej użytej wartości (**'SKIP_SAVE'**);

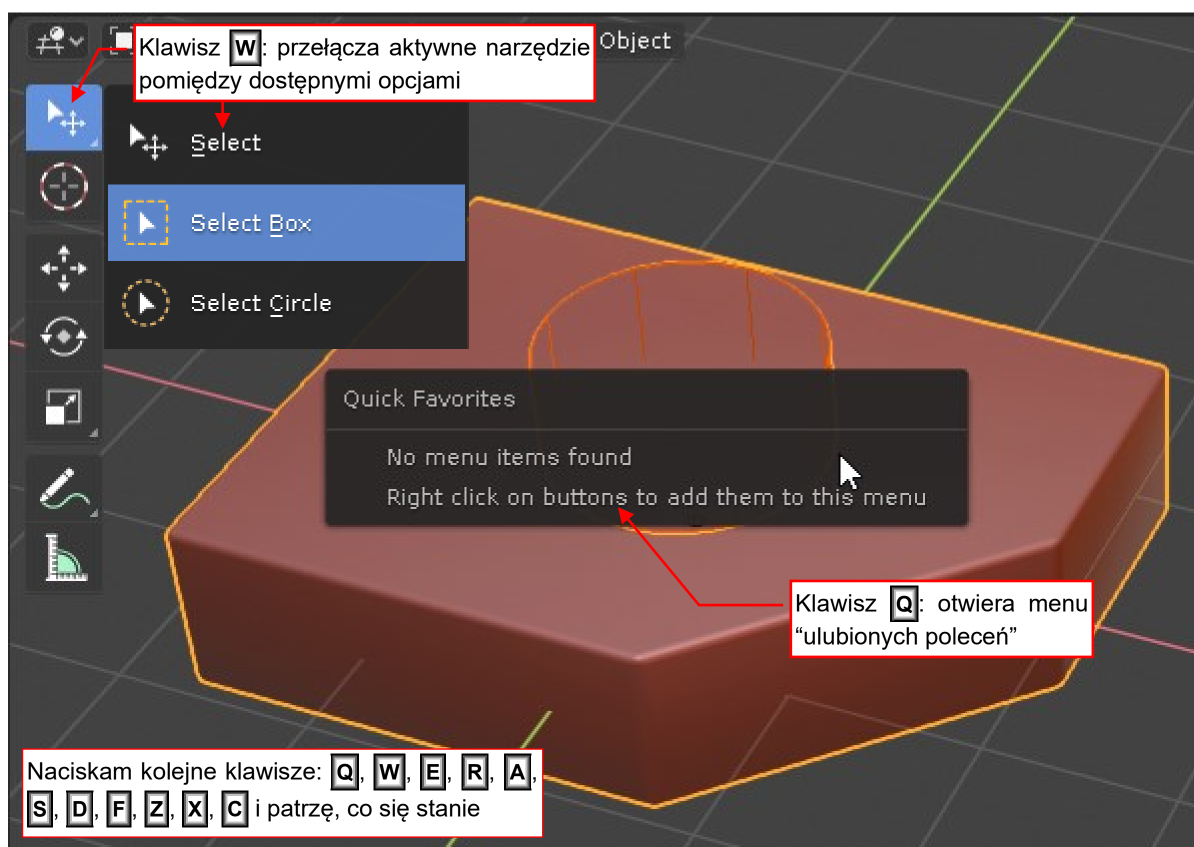
4.4 Dodanie skrótu klawiatury i *Pie Menu*

Gdy wtyczka jest już uruchomiona, można pomyśleć o dodaniu udogodnienia w postaci wywołania polecenia *Boolean* za pomocą skrótu klawiatury. Oczywiście, najpierw trzeba ustalić, jaki to ma być skrót.

Blender jest znany z dziesiątków (może nawet setek) skrótów klawiaturowych. Muszę wśród nich znaleźć jakąś nieużywaną kombinację klawiszy dla naszego operatora. Choć, skoro mówimy już o „kombinacji": uważam, że im mniej klawiszy trzeba naraz nacisnąć, tym lepiej. Naciśnięcie naraz jakichś złożonych układów z Alt, Ctrl, i może jeszcze Shift, jest na pewno trudniejsze niż naciśnięcie pojedynczego klawisza. Dodatkowo, preferuję klawisze z obszaru po lewej stronie klawiatury, bo tam większość użytkowników ma wolną rękę. Z drugiej strony – jeżeli mam wpisać skrót klawiatury „na trwałe" do kodu wtyczki (tak, jak to robi większość ich autorów), to muszę wybrać jakąś unikalną (i przez to złożoną) kombinację klawiszy. W ten sposób minimalizuję szansę jej powtórzenia w innej wtyczce, która także została aktywowana przez tego samego użytkownika.

- Umożliwię użytkownikowi samodzielne ustalenie klawiszy skrótu dla tego polecenia. Przygotuję w tym celu specjalny panel preferencji wtyczki (w następnej sekcji). Jako wartość domyślną wybiorę jakiś prosty, łatwy w użyciu skrót – najlepiej pojedynczy klawisz klawiatury.

Poszukiwanie odpowiedniego klawisza zacząłem od przygotowania „środowiska testowego": ustawienia się w widoku *3D View*, włączenia *Object Mode* i zaznaczenia paru obiektów sceny. Następnie zacząłem naciskać pojedyncze klawisze po lewej stronie klawiatury: Q, W, E, R, A, S, D, F, Z, X, C, … sprawdzając, co się wydarzy. Przy okazji poznałem kilka możliwości Blendera, o których nie miałem pojęcia, np. menu *Quick Favorities* (otwierane klawiszem Q), czy przełączanie wariantów aktywnego narzędzia w przyborniku (W):

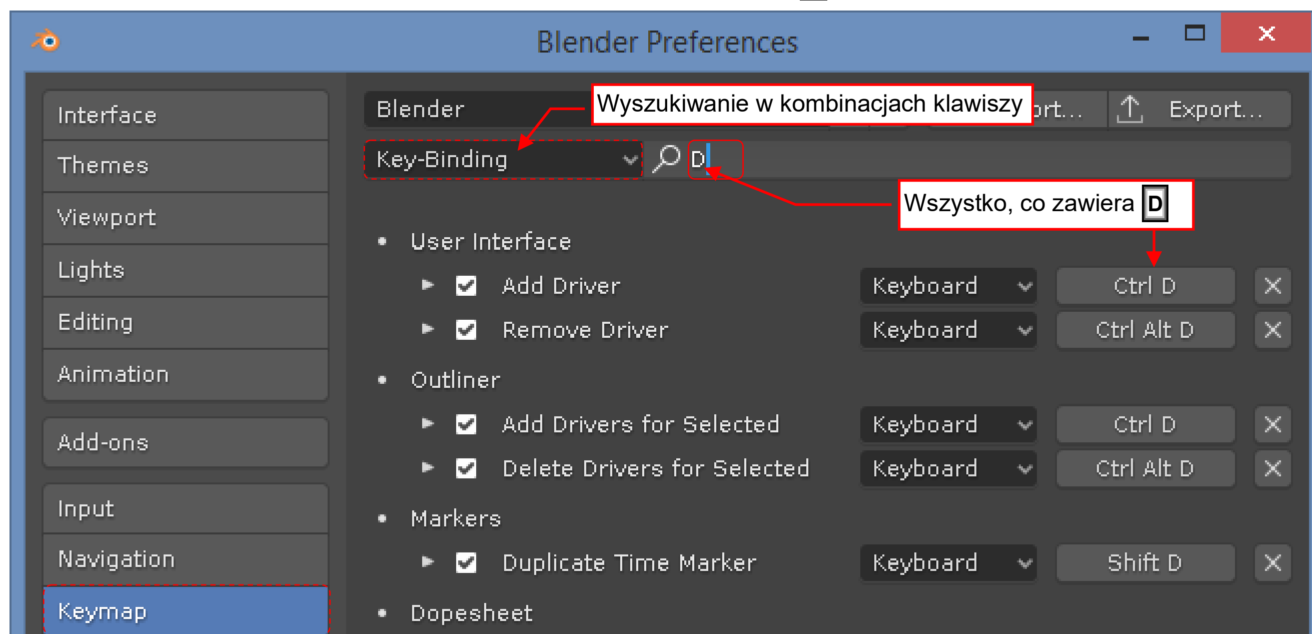


Rysunek 4.4.1 Poszukiwanie „wolnych" klawiszy na skrót dla polecenia *Boolean*

O dziwo, test wykazał, że z klawiszami: E, D i F nie jest jeszcze związane żadne polecenie. Po tej wstępnej selekcji przeszedłem do „finału" tych eliminacji - w ustawieniach Blendera.

Copyright Witold Jaworski, 2011-2019.

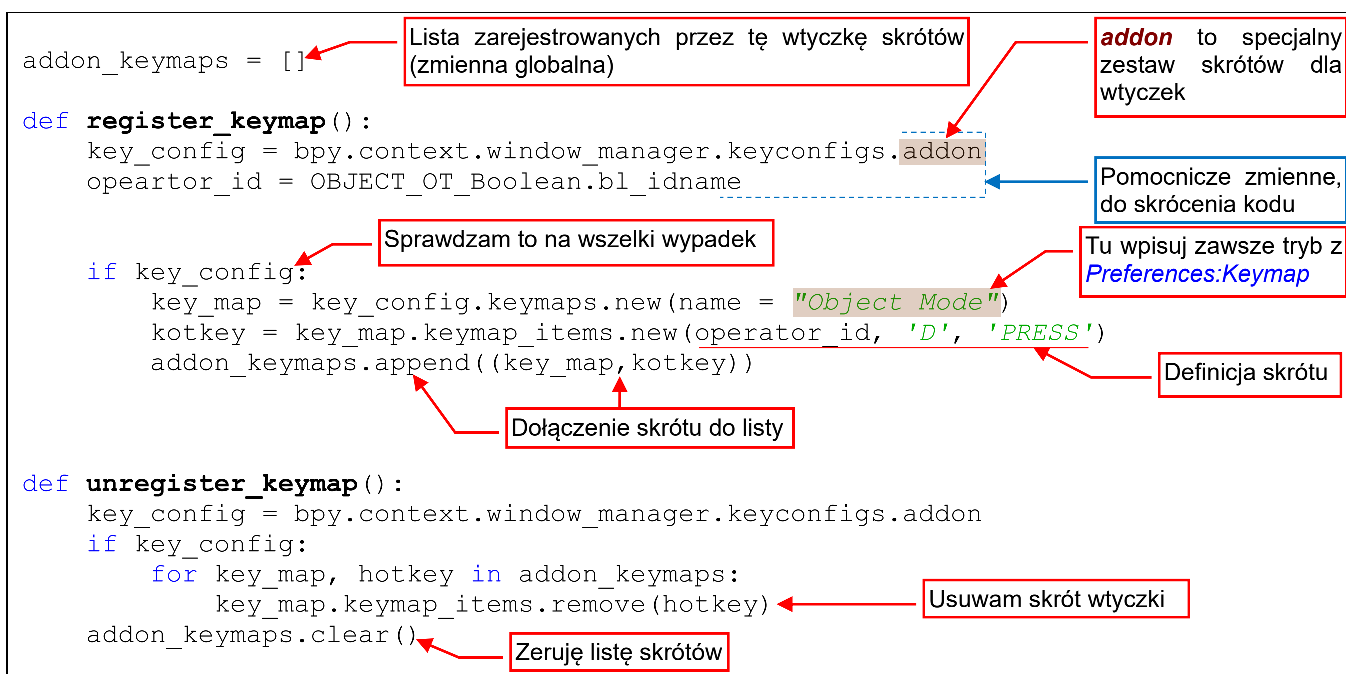
W oknie ustawień (*Edit→Preferences*), zakładce ***Keymap*** wyszukałem wszystkie związane skróty z każdym z wybranych klawiszy (Rysunek 4.4.2 pokazuje weryfikację klawisza D):



Rysunek 4.4.2 Sprawdzanie, czy (i gdzie) wybrany klawisz jest używany

Zwracałem szczególną uwagę, w ilu i jakich trybach występuje wybrany przeze mnie skrót. Ostatecznie zdecydowałem się na użycie klawisza D, gdyż w standardzie Blendera jest wykorzystywany tylko w dwóch modalnych zmianach projekcji: *View 3D Fly Modal* i *View 3D Walk Modal*. (Skróty E i F są używane częściej, choć także w zupełnie innych trybach/oknach. Jednak już sama częstotliwość użycia mogłaby spowodować wśród użytkowników większą liczbę pomyłek).

Przygotowałem dwie pomocnicze procedury do rejestrowania i usunięcia skrótu (Rysunek 4.4.3):



```
addon_keymaps = []

def register_keymap():
    key_config = bpy.context.window_manager.keyconfigs.addon
    opeartor_id = OBJECT_OT_Boolean.bl_idname

    if key_config:
        key_map = key_config.keymaps.new(name = "Object Mode")
        kotkey = key_map.keymap_items.new(operator_id, 'D', 'PRESS')
        addon_keymaps.append((key_map,kotkey))

def unregister_keymap():
    key_config = bpy.context.window_manager.keyconfigs.addon
    if key_config:
        for key_map, hotkey in addon_keymaps:
            key_map.keymap_items.remove(hotkey)
    addon_keymaps.clear()
```


Rysunek 4.4.3 Pomocnicze funkcje dodające i usuwające skrót

Są to proste metody bez żadnych parametrów, gdyż skrót wpisuję na razie jako wartość stałą. Procedura ***register_keymap()*** przypisuje klawisz skrótu (**'D'**) do operatora *Boolean* i zapamiętuje utworzoną mapę skrótów i skrót w globalnej liście ***addon_keymaps***. Procedura ***unregister_keymap()*** usuwa skrót przypisany do operatora *Boolean* i zeruje listę ***addon_keymaps***.

Wywołania procedur ***register_keymap()/unregister_keymap()*** umieściłem w metodach ***register()*** i ***unregister()***:

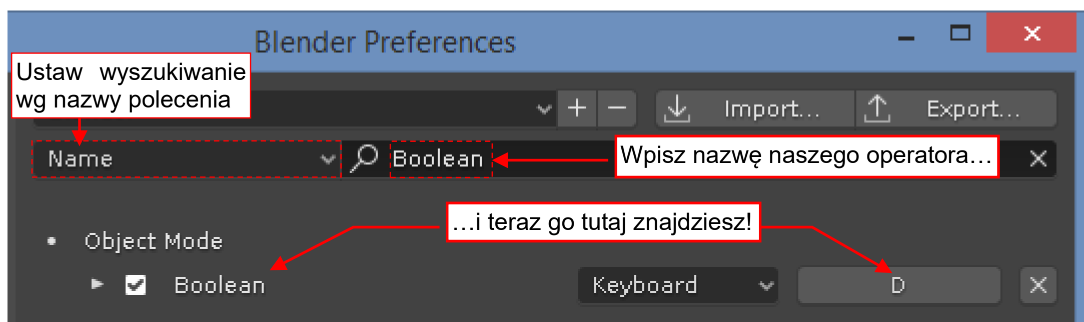
```
def register():
    register_class(OBJECT_OT_Boolean)
    bpy.types.VIEW3D_MT_object.prepend(menu_draw)
    register_keymap()

def unregister():
    unregister_keymap()
    bpy.types.VIEW3D_MT_object.remove(menu_draw)
    unregister_class(OBJECT_OT_Boolean)
```

Procedury rejestrujące/usuwające skrót klawiatury

Rysunek 4.4.4 Wywołanie procedur rejestracji skrótu

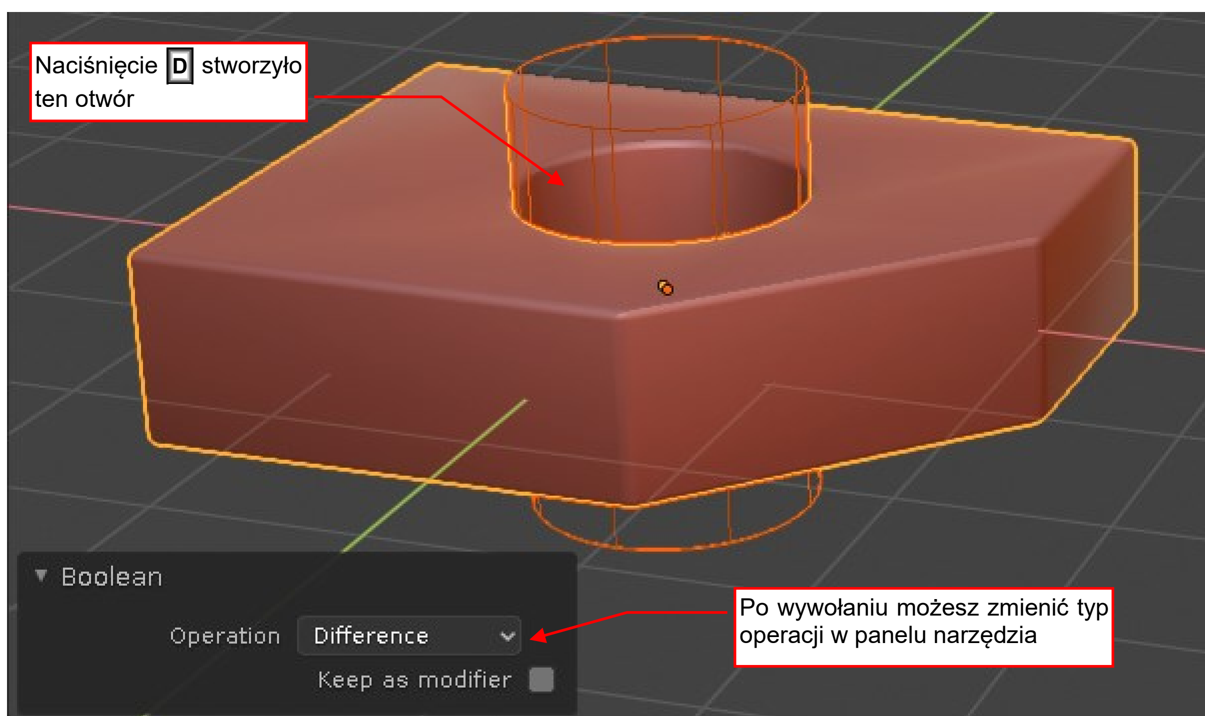
(Zwróć uwagę, że w ***unregister()*** wywołuję odpowiednie operacje w odwrotnej kolejności niż to zrobiłem w ***register()***. W ten sposób na pewno uniknę odwołania do jakiejś klasy, której jeszcze nie zarejestrowałem lub którą już usunąłem).

Gdy klikniesz w przycisk *Run Script*, to w ustawieniach Blendera pojawi się nasz skrót (Rysunek 4.4.5):



Rysunek 4.4.5 Skrót do polecenia *Boolean*

Gdy w oknie *View 3D* zaznaczysz walec, płytkę, i naciśniesz **D**, wykonasz w płytce otwór (Rysunek 4.4.6):



Rysunek 4.4.6 Wywołanie polecenia *Boolean* (skrótem z klawiatury)

Polecenie zostało wywołane bez parametru ***op***, więc Blender zastosował w nim poprzednio użytą wartość.

Copyright Witold Jaworski, 2011-2019.

Myślę, że użytkownikowi będzie wygodniej od razu wybrać właściwą operację Boole'a z jakiegoś menu wyświetlającego się po naciśnięciu klawisza D. Mogłoby być to klasyczne „wyskakujące" (*popup*) menu oferujące wybór jednej z trzech dostępnych opcji. Jednak w Blenderze możemy użyć do tego celu czegoś bardziej eleganckiego: *pie menu* (takiego, jakie pokazuje Rysunek 3.1.8 na str. 37).

Implementacja *pie menu* to klasa, która różni się tylko drobnymi szczegółami od implementacji zwykłego „rozwijalnego" menu (Rysunek 4.4.7):

```
class VIEW3D_MT_Boolean(bpy.types.Menu):
    '''This pie menu shows Boolean operator options.
       Invoked by the hotkey assignet to this add-on
    '''
    bl_idname = "VIEW3D_MT_Boolean"
    bl_label = "Select operation:"

    def draw(self, context):
        pie = self.layout.menu_pie()
        pie.operator_enum(OBJECT_OT_Boolean.bl_idname, property="op")
```

Klasa bazowa: ***Menu*** (ta sama, co w przypadku menu rozwijalnych)

Blender 2.8 preferuje nazwy menu z "_MT_" w środku (UWAGA: chyba należy tu użyć nazwy klasy!)

Tekst wyświetlany w środku menu

Metoda wyświetlająca *pie menu*

Ta metoda automatycznie generuje pozycje menu dla wskazanego parametru operatora (parametru z enumeracją)

Rysunek 4.4.7 Implementacja *pie menu* z opcjami polecenia *Boolean*

Nadałem mu nazwę ***VIEW3D_MT_Boolean***. Od zwykłego menu rozwijalnego odróżnia ją wyłącznie kod metody ***draw()***. Na początku wywołuję w tej procedurze funkcję ***bpy.types.UILayout.menu_pie()***: to właśnie przygotowuje, na razie puste, *pie menu*. Następnie pozwalam Blenderowi wypełnić je pozycjami odpowiadającymi enumeracji z argumentu ***op***. (Tak samo stworzyliśmy submenu rozwijalne *Boolean* – por. Rysunek 4.2.10, str. 93).

To nowe menu jest wywoływane przez skrót klawiatury (zamiast operatora *Boolean*). Dokonałem odpowiednich przeróbek w metodzie ***register_keymap()***. W miejsce naszego id operatora podstawiłem teraz nazwę operatora ***bpy.ops.wm.call_menu_pie()***, otwierające *pie menu* o nazwie ***name*** (Rysunek 4.4.8):

```
def register_keymap():
    key_config = bpy.context.window_manager.keyconfigs.addon
    if key_config:
        key_map = key_config.keymaps.new(name = "Object Mode")
        hotkey = key_map.keymap_items.new('wm.call_menu_pie', 'D', 'PRESS')
        hotkey.properties.name = VIEW3D_MT_Boolean.bl_idname
        addon_keymaps.append((key_map,hotkey))

        unregister_keymap() – bez zmian

def register():
    register_class(OBJECT_OT_Boolean)
    register_class(VIEW3D_MT_Boolean)
    bpy.types.VIEW3D_MT_object.prepend(menu_draw)
    register_keymap()

def unregister():
    unregister_keymap()
    bpy.types.VIEW3D_MT_object.remove(menu_draw)
    unregister_class(VIEW3D_MT_Boolean)
    unregister_class(OBJECT_OT_Boolean)
```

Polecenie wywołujące *pie menu*

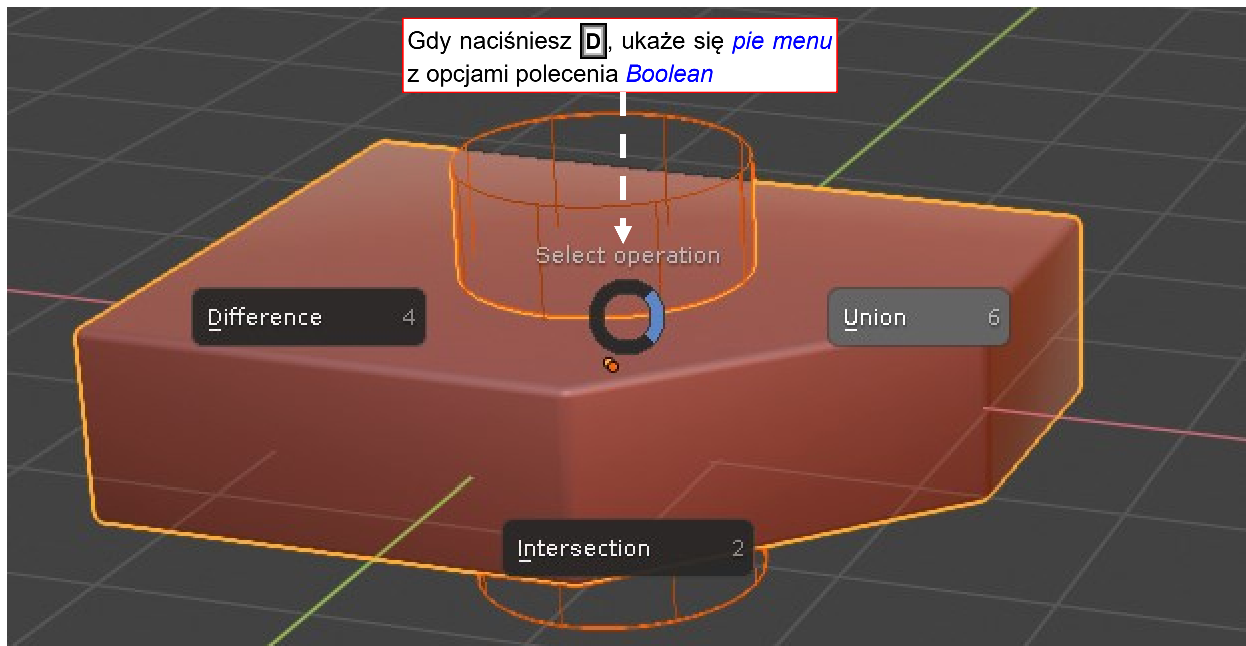
Nazwę menu przekazujemy w dodatkowej linii

Rejestracja i usunięcie klasy menu

Rysunek 4.4.8 Rejestracja i wywoływanie *pie menu*

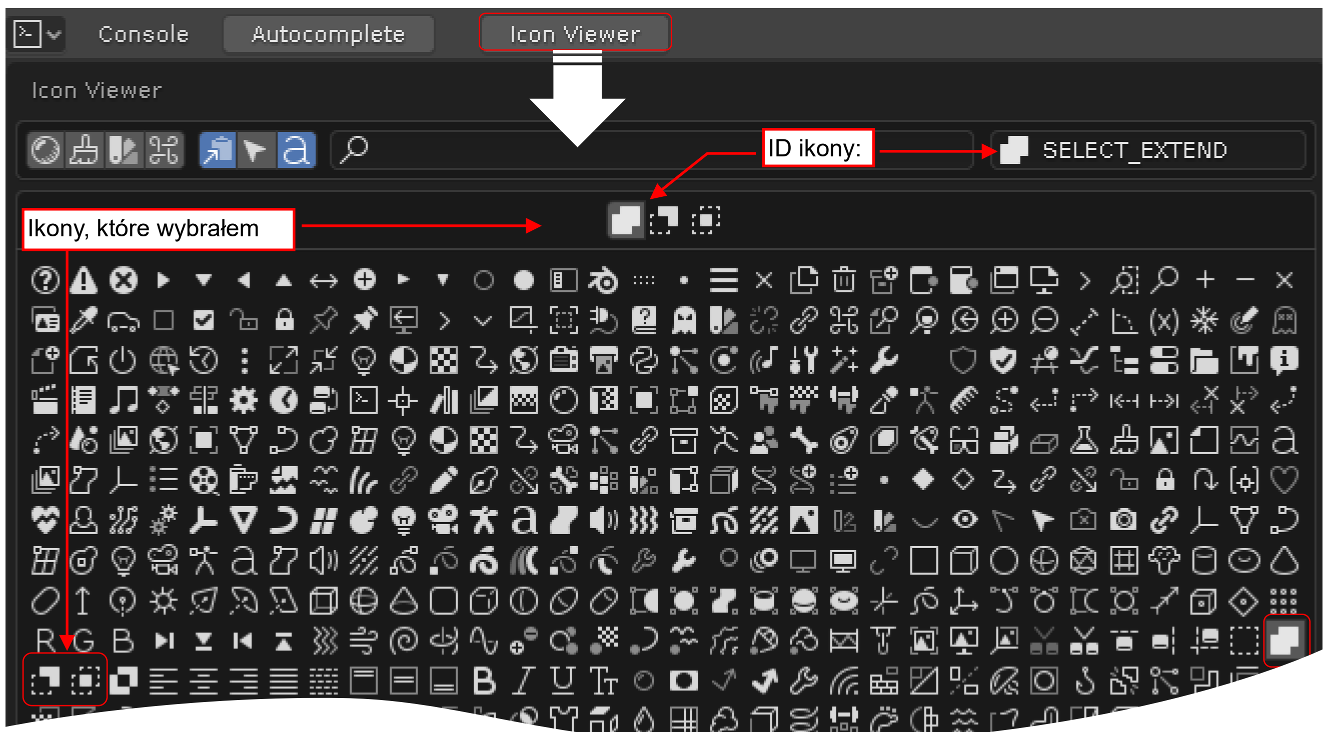
W metodach ***register()*** i ***unregister()*** rejestruję/usuwam klasę menu (***VIEW3D_MT_Boolean***).

Aby sprawdzić, jak to teraz działa, przerejestruj skrypt (klikając w przycisk ***Run Script***), przejdź do okna *View 3D* i naciśnij klawisz D (Rysunek 4.4.9):



Rysunek 4.4.9 Pierwsza wersja *pie menu*

Działa nieźle, ale spróbuję jeszcze uatrakcyjnić wygląd tego menu dodając jakieś ikony. Najprościej jest wybrać jakieś spośród standardowych ikon Blendera, pokazywanych przez przycisk ***Icon Viewer*** (z *Python Console* - Rysunek 4.4.10):



Rysunek 4.4.10 Przeglądarka standardowych ikon Blendera

Niestety, w tym zbiorze nie ma jeszcze ikon dla opcji modyfikatora *Boolean*. Po dłuższym poszukiwaniu zdecydowałem się użyć obrazków używanych do operacji na zbiorach selekcji. Ich identyfikatory odczytuję z pola w prawym, górnym narożniku okna: **'SELECT_EXTEND'** (*Union*), **'SELECT_SUBSTRACT'** (*Difference*), **'SELECT_INTERSECT'** (*Intersection*). Nie uważam, aby te symbole były zbyt ładne, ale przynajmniej poprawnie przedstawiają koncepcję każdej z trzech operacji.

Copyright Witold Jaworski, 2011-2019.

Najprościej jest dodać symbole tych ikon do definicji enumeracji ***op*** w klasie ***OBJECT_OT_Boolean*** (Rysunek 4.4.11):

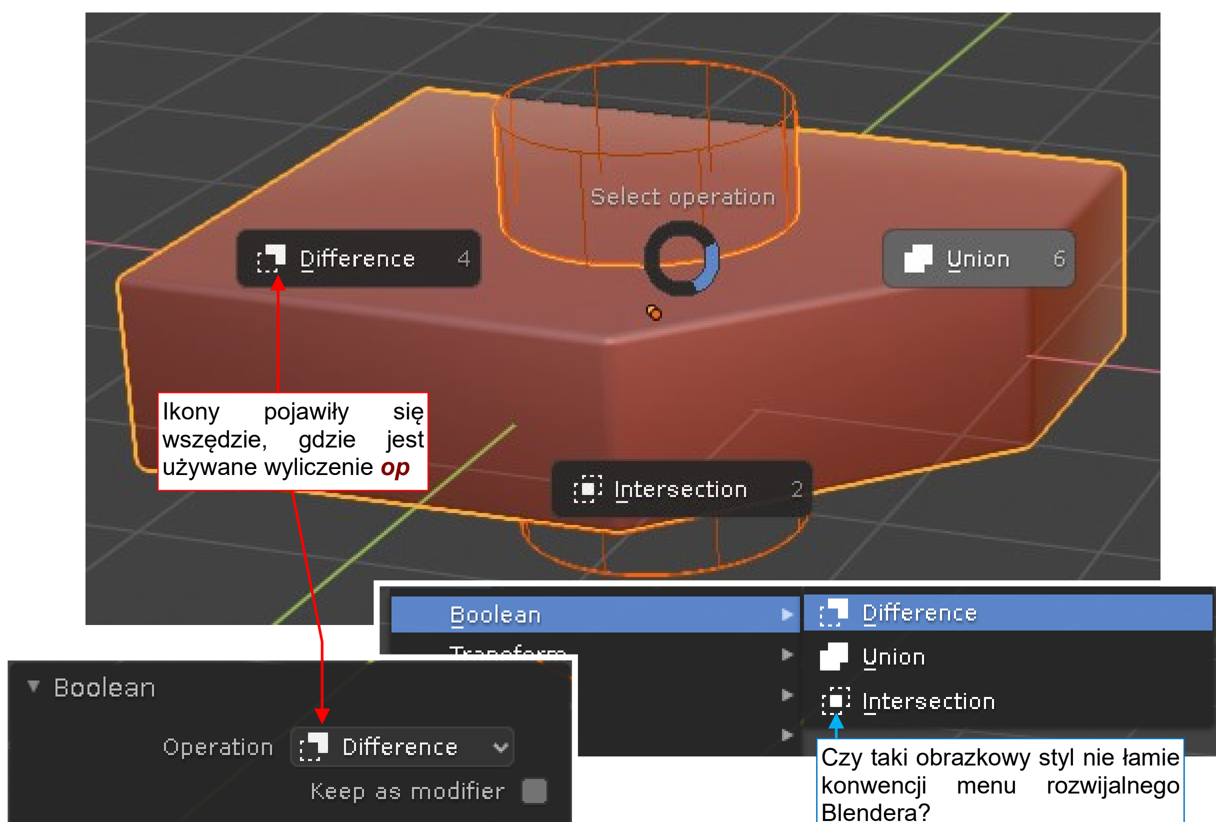
```
class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a 'destructive' Boolean operation on the active object
       Arguments:
       @op (Enum): Boolean operation, in ['DIFFERENCE', 'UNION', 'INTERSECT']
       @modifier (Bool): add this operation as the object modifier
    '''
    bl_idname = "object.boolean"
    bl_label = "Boolean"
    bl_description = "Perform a Boolean operation on active object"
    bl_options = {'REGISTER', 'UNDO'}

    op :  EnumProperty( items = [
        ('DIFFERENCE',"Difference","Boolean difference", 'SELECT_SUBTRACT',1),
        ('UNION',"Union","Boolean union", 'SELECT_EXTEND',2),
        ('INTERSECT',"Intersection","Boolean intersection", 'SELECT_INTERSECT',3),
                              ],
                        name = "Operation",
                        description = "Boolean operation",
                        default='DIFFERENCE',
                       ) #end EnumProperty
```

Do każdego z elementów enumeracji dodałem dwa dodatkowe elementy: id ikony (tekst) oraz numer porządkowy

Rysunek 4.4.11 Dodanie ikon do definicji wyliczenia *op*

Zauważ, że w każdym elemencie enumeracji symbolowi ikony towarzyszy numer kolejny. To wymóg Blendera – gdybym podał tylko symbol ikony, zgłosiłby wyjątek w trakcie rejestracji wtyczki.

Przerejestruj powtórnie skrypt (*Run Script*) i naciśnij przycisk **D** (Rysunek 4.4.12):



Rysunek 4.4.12 Rezultat dodania ikon do wyliczenia *op*

Ikony pojawiły się wszędzie – nie tylko w *pie menu*, ale także w submenu *Boolean* i przyborniku polecenia.

Przyglądając się zmienionemu menu *Object→Boolean*, zacząłem się zastanawiać, czy taki obrazkowy styl nie łamie jakiejś konwencji Blendera przyjętej dla menu rozwijalnych. Sprawdziłem więc to dokładniej i stwierdziłem, że choć w większości submenu *Object* nie figurują ikony, to jednak czasami są używane. Zobacz np. submenu *Object→Convert To*. Podobnie jest w innych menu okna *3D View*.

Podsumowanie

- Skomplikowane kombinacje trzech lub czterech klawiszy są niewygodne w użyciu, dlatego staraj się dobrać skróty krótsze, złożone z dwu, a nawet jednego klawisza.
- Aby przypisać do poleceń wtyczki jakiś skrót klawiatury, trzeba najpierw go znaleźć. Zaczynaj od wytypowania kilku najbardziej obiecujących skrótów testując możliwe kombinacje w docelowym oknie Blendera (str. 99), a potem wybierz spośród nich tą, która jest w Blenderze najrzadziej używana (str. 100);
- Przypisz skrót do polecenia tworząc nową tzw. mapę klawiszy (*key map*) w zestawie skrótów ***bpy.context.window_manager.keyconfigs.addon*** (str. 100). Zrób to przy okazji rejestracji wtyczki w procedurze ***register()*** i zachowaj obiekty reprezentujące skrót w zmiennej globalnej. Będziesz ich potrzebował do wyrejestrowania w procedurze ***unregister()*** (str. 101);
- W polecenia z kilkoma wariantami wywołań, jak *Boolean*, warto powiązać skrót z *pie menu*, które pozwoli użytkownikowi wybrać odpowiedni wariant (str. 102, 103);
- Interfejs użytkownika wtyczki możesz wzbogacić o standardowe ikony Blendera (str. 103, 104);

Copyright Witold Jaworski, 2011-2019.

4.5 Implementacja panelu preferencji wtyczki

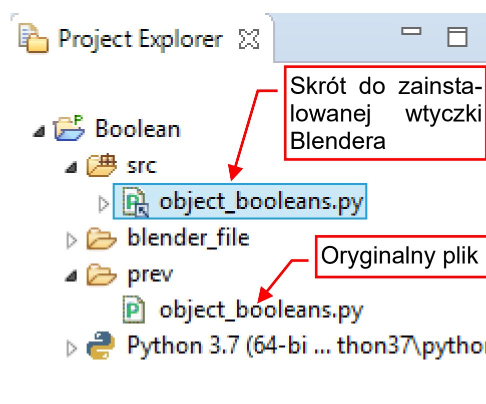
W zasadzie nasza wtyczka *object_booleans.py* jest już gotowa. Pozostało tylko dodać jej panel ustawień (preferencji), w której użytkownik będzie mógł zmienić domyślny skrót tego polecenia na inny. Zrobimy to w tej sekcji.

Testowanie panelu preferencji wymaga zainstalowania naszej wtyczki. Zrobiłem to w oknie preferencji Blendera (*Edit→Preferences*), wywołując polecenie ***Install…*** (Rysunek 4.5.1):

Rysunek 4.5.1 Instalacja wtyczki (w sekcji *Add-ons*)

W oknie wyboru pliku wtyczki przeszedłem do folderu projektu i wskazałem Blenderowi źródłowy plik wtyczki (w podkatalogu *src*). Blender skopiował oryginalny plik do swojego folderu, w którym przechowuje „zainstalowane” wtyczki użytkownika. (Pełną ścieżkę do tej kopii możesz odczytać z pola *File*, wyświetlanego w panelu z opisem wtyczki).

W folderach projektu przeniosłem oryginalny plik do folderu *prev*, a w to miejsce wstawiłem link do pliku (poleceniem *File→New*, szczegóły - patrz str. 146) znajdującego się w folderze wtyczek Blendera (Rysunek 4.5.2). Od tej chwili wszystkie zmiany będziemy wprowadzać w tej wtyczce.



Rysunek 4.5.2 Skrót do pliku, dodany do projektu

Zainstalowanej w ten sposób wtyczki Blendera nie można debugować za pomocą pomocniczego skryptu *Run.py*, którego używaliśmy do tej pory. Dlatego na początku kodu *object_booleans.py* umieszczam kilka linii pozwalającej jej „nawiązać kontakt” ze zdalnym debugerem PyDev (Rysunek 4.5.3):

```
import bpy
import traceback #for error handling
DEBUG = 1

###--- for direct debugging of this add-on (update the pydevd path!) ------------
if DEBUG == 1:
    import sys
    pydev_path =
     'C:/Users/me/.p2/pool/plugins/org.python.pydev.core_7.2.1.201904261721/pysrc'
    if sys.path.count(pydev_path) < 1: sys.path.append(pydev_path)

    import pydevd


    pydevd.settrace(stdoutToServer=True, stderrToServer=True, suspend=False)
###-- end remote debug initialization -----------------------------------------
```


Rysunek 4.5.3 Wywołanie zdalnego debugera w kodzie zainstalowanej wtyczki Blendera.

- Pamiętaj, że aktywna (włączona) wtyczka jest ładowana już podczas uruchamiania Blendera. Dlatego jeszcze przed otworzeniem testowego pliku **.blend* uruchom w Eclipse zdalny debuger PyDev (por. str. 54)

Po tych przygotowaniach dodałem do wtyczki panel preferencji. To klasa pochodna klasy API ***AddonPreferences*** (Rysunek 4.5.4):

```
#---------- # Add-On Preferences -----------
class Preferences(bpy.types.AddonPreferences):
    '''This class provides the user pssibility of altering the keyboard shortcut
       assigned to the Boolean pie menu
    '''
    bl_idname = __name__


    shift : BoolProperty(name = "Shift", description= "Use the [Shift] key",
                                                              default=False)
    ctrl  : BoolProperty(name = "Ctrl", description= "Use the [Ctrl] key",
                                                              default=False)
    alt   : BoolProperty(name = "Alt", description= "Use the [Alt] key",
                                                              default=False)
    key   : EnumProperty(items = [('NONE',"None","No hotkey")] +
             [tuple([chr(i),chr(i),"[%s] key" % chr(i)]) for i in range(65, 91)],
             name = "Keyboard key",
             description = "Selected keyboard key",
             default = 'D',
            )

    def draw(self, context):
        layout = self.layout
        layout.prop(self,"key")
        layout.prop(self,"shift")
        layout.prop(self,"ctrl")
        layout.prop(self,"alt")
```


Rysunek 4.5.4 Pierwsza wersja panelu preferencji wtyczki

To prosta klasa, składająca się z deklaracji pól konfiguracji oraz funkcji ***draw()***, odpowiedzialnej za wyświetlanie panelu. Na razie umieściłem w niej implementację domyślnego układu (pojedyncza kolumna – jak w menu).

Copyright Witold Jaworski, 2011-2019.

W wyliczeniu klawiszy klawiatury umieściłem wyrażenie generujące deklaracje dla liter ‘A’-‘Z’. (kody ASCII od 65 do 91). Wynika to wyłącznie z lenistwa: nie chciało mi się pracowicie wpisywać tych 26 definicji. Zwróć uwagę, że wartość pola ***bl_idname*** tej klasy musi być nazwą pliku wtyczki (bez rozszerzenia *„.py”*). Dlatego podstawiam w to miejsce domyślną zmienną globalną Pythona ***__name__***.

Nową klasę należy jeszcze zarejestrować. To już trzecia klasa, która musimy obsłużyć w ten sposób. Aby zmniejszyć szansę, że np. dodam rejestrację jakiejś klasy tylko w metodzie ***register()*** i zapomnę umieścić odpowiednie wywołanie w ***unregister()***, wprowadziłem globalną listę klas do rejestracji: ***classes*** (Rysunek 4.5.5):

```
#list of the classes in this add-on to be registered in Blender API:
classes = [
            OBJECT_OT_Boolean,
            VIEW3D_MT_Boolean,
            Preferences,
          ]

def register():
    for cls in classes:
        register_class(cls)
    bpy.types.VIEW3D_MT_object.prepend(menu_draw)
    register_keymap()
    if DEBUG: print(__name__ + ": registered")

def unregister():
    unregister_keymap()
    bpy.types.VIEW3D_MT_object.remove(menu_draw)
    for cls in classes:
        unregister_class(cls)
    if DEBUG: print(__name__ + ": UNregistered")
```

Zmienna globalna ***classes*** to lista klas API do zarejestrowania/wyrejestrowania

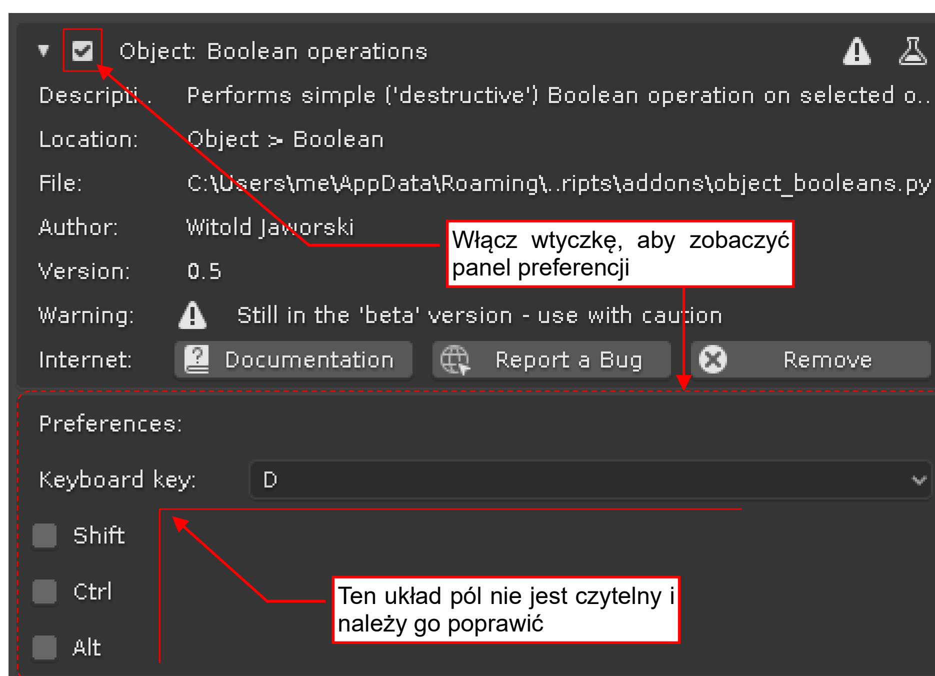
Pętle obsługujące rejestrację i wyrejestrowanie

Komunikaty diagnostyczne

Rysunek 4.5.5 Zmodyfikowane procedury obsługi rejestracji

Zamiast pojedynczych wywołań metody ***register_class()*** robię to teraz w pętli dla wszystkich elementów listy klas. Na końcu procedur umieściłem komunikaty diagnostyczne (są nadal ułatwieniem: nie wszystko warto śledzić w debugerze). Gdy ustawisz stałą ***DEBUG*** na 0 – nie będą się wyświetlać.

Gdy włączysz naszą wtyczkę – Blender załaduje jej kod i wyświetli panel preferencji (Rysunek 4.5.6):



Rysunek 4.5.6 Wygląd pierwszej wersji panelu preferencji

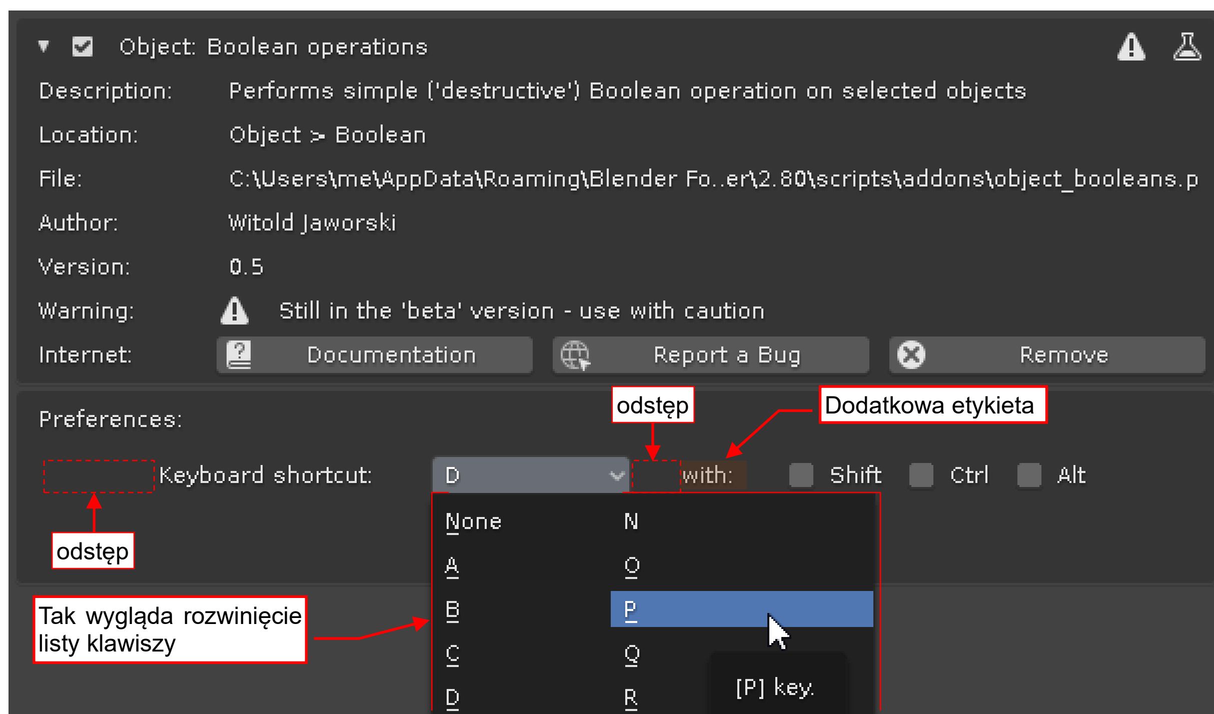
Panel się wyświetla, ale układ pól jest bardzo nieczytelny (trudno skojarzyć, że opisuje pojedynczy skrót klawiatury). Musimy go poprawić (Rysunek 4.5.7):

```
    def draw(self, context):
        row = self.layout.row(align=True)
        row.alignment = 'LEFT'
        row.separator(factor = 10)
        row.prop(self,"key", text="Keyboard shortcut")
        row.separator(factor = 3)
        row.label(text="with:")
        row.prop(self,"shift")
        row.prop(self,"ctrl")
        row.prop(self,"alt")
```

Ustawiamy kontrolki poziomo (w "wiersz")
Wyrównanie: do lewej
Dodatkowy odstęp
Inna etykieta dla listy klawiszy
Dodatkowy odstęp (węższy niż pierwszy)
Dodatkowa etykieta

Rysunek 4.5.7 Poprawiony kod wyświetlający panel preferencji

Oczywiście, wszystkie dodatki zaznaczone w kodzie na ilustracji powyżej wprowadzałem stopniowo, co chwila sprawdzając efekt w oknie Blendera. (Każdorazowo po zapisaniu pliku wyłączałem i włączałem wtyczkę). Ostateczny rezultat przedstawia Rysunek 4.5.8:



Rysunek 4.5.8 Wygląd poprawionego panelu preferencji

Pierwsza wartość na liście klawiszy – *None* – będzie służyć do wyłączenia skrótu wtyczki (i w efekcie – jej *pie menu*). Dodaję tę możliwość na wszelki wypadek, gdyby użytkownik zdecydował, że nie potrzebuje żadnych skrótów klawiatury do wywołania tego polecenia.

Spróbuj wybrać inny klawisz z listy w panelu preferencji – np. *E* – i zamknąć Blendera pozostawiając wtyczkę w stanie aktywnym (włączoną). Blender 2.8 ma domyślnie włączony autozapis konfiguracji, dzięki czemu zapamięta ten stan. Potem otwórz Blender ponownie. Zauważ, że Twoje ustawienia zostały zachowane – wtyczka jest od razu aktywna, a wybrany klawisz w jej ustawieniach to klawisz *E*.

Jeżeli jednak teraz wyłączysz (dezaktywujesz) wtyczkę, a potem włączysz ją ponownie, klawisz powróci do swojej wartości domyślnej: *D*.

- Preferencje (ustawienia) wtyczki są zapamiętywane przez cały czas aktywności wtyczki. Blender usuwa je, gdy wyłączysz ten komponent (w sekcji *Blender Preferences:Add-ons*).

Copyright Witold Jaworski, 2011-2019.

Sprawdziliśmy, że ustawienia wtyczki są poprawnie przechowywane przez Blender pomiędzy sesjami. Na razie jednak to, co ustawiamy w panelu preferencji nie ma żadnego wpływu na skrót, za pomocą którego wywoływane jest *pie menu*. Czas to zmienić.

Kiedy piszę kod, zawsze trzymam się zasady „jednego miejsca”: określona operacja (dodanie klawisza skrótu, ustalenie wartości domyślnych) powinna występować tylko w jednym miejscu kodu. Może to być procedura, może to być deklaracja stałej. We wszystkich innych miejscach, w których są potrzebne, należy wywołać tę procedurę lub użyć tę stałą. (W ten sposób eliminuję szansę na potencjalny błąd, wynikający ze zmiany jakiegoś ustawienia w jednym miejscu, i pozostawienia drugiego bez zmian). W przypadku klawiszy skrótu zdecydowałem się obsłużyć ich wielowariantowość w sposób charakterystyczny dla Pythona: przygotowując wcześniej listę argumentów dla wywołania funkcji ***keymap_items.new()*** (por. Rysunek 4.4.3 str. 100 oraz Rysunek 4.4.8, str. 102). Zacząłem od globalnej zmiennej ***hotkey_defaults*** (traktuję ją jak stałą) ze słownikiem domyślnych argumentów dla funkcji ****.new()*** (Rysunek 4.5.9):

```
#---------- # Add-On Preferences -----------
#default values for the keymap_items.new() call (see register_keymap() method)
hotkey_defaults  = {"idname": 'wm.call_menu_pie', "type": 'D', "value": 'PRESS',
                                        "shift": False, "ctrl": False, "alt":False}
```

Słownik domyślnych argumentów dla wywołania funkcji ***keymap_items.new()*** (w procedurze ***register_keymap()***).
Zwróć uwagę na nazwy kluczy słownika: muszą odpowiadać nazwom argumentów funkcji ****.new()***

```
class Preferences(bpy.types.AddonPreferences):
    '''This class provides the user pssibility of altering the keyboard shortcut
       assigned to the Boolean pie menu
    '''
    bl_idname = __name__ #do not change this line

    shift : BoolProperty(name = "Shift", description= "Use the [Shift] key",
                                              default=hotkey_defaults["shift"])

    ctrl : BoolProperty(name = "Ctrl", description= "Use the [Ctrl] key",
                                              default=hotkey_defaults["ctrl"])

    alt : BoolProperty(name = "Alt", description= "Use the [Alt] key",
                                              default=hotkey_defaults["alt"])

    key : EnumProperty(items = [('NONE',"None","No hotkey")] +
             [tuple([chr(i),chr(i),"[%s] key" % chr(i)]) for i in range(65, 91)],
              name = "Keyboard key",
              description = "Selected keyboard key",
              default = hotkey_defaults["type"],
             )
```

W deklaracji domyślnych wartości pól preferencji podstawiam odpowiednie pola ze słownika ***hotkey_defaults***

metoda ***draw()*** *– bez zmian*

Rysunek 4.5.9 Deklaracja domyślnych parametrów dla skrótu klawiatury

Zwróć uwagę, że nazwy kluczy słownika ***hotkey_defaults*** są takie same, jak nazwy argumentów funkcji ***keymap_items.new()***. To wymóg Pythona (wszelkie opcjonalne argumenty można w takim słowniku pominąć). Deklaracja słownika musi być umieszczona powyżej deklaracji klasy *Preferences*, gdyż używam wartości słownika do określenia domyślnych wartości pól tej klasy.

- Wywołanie funkcji ***BoolProperty()*** i ***EnumProperty()*** w kodzie powyżej następuje podczas ładowania wtyczki, zanim zostanie wywołana procedura ***register()***. Dlatego należy uważać, aby używać w nich tylko tych elementów, które zostały zadeklarowane we wcześniejszych liniach skryptu.

Następnie zmodyfikowałem procedurę ***register_keymap()*** (Rysunek 4.5.10):

```
def register_keymap():
    '''Registers current hotkey'''
    #assumption: at this moment the addon

    args = hotkey_defaults

    if Preferences.bl_idname in bpy.context.preferences.addons:
        #update args, according preferences:

        prf = bpy.context.preferences.addons[Preferences.bl_idname].preferences

        args["type"] = prf.key #use the user-defined key and its modifiers:
        args["shift"], args["ctrl"], args["alt"] = prf.shift, prf.ctrl, prf.alt
    else:
        prf = None

    if args["type"]== 'NONE' : return

    key_config = bpy.context.window_manager.keyconfigs.addon
    if key_config:
        key_map = key_config.keymaps.new(name = "Object Mode")
        hotkey = key_map.keymap_items.new(**args)
        hotkey.properties.name = VIEW3D_MT_Boolean.bl_idname
        addon_keymaps.append((key_map,hotkey))

        if DEBUG: print("Keyboard shortcut set to: "
                        + ("[Shift]-" if args["shift"] else "")
                        + ("[Ctrl]-" if args["ctrl"] else "")
                        + ("[Alt]-" if args["alt"] else "")
                        + ("[%s]" % args["type"])
                        + (" (from add-on preferences)" if prf else ""))
```


Rysunek 4.5.10 Wykorzystanie preferencji wtyczki przy rejestrowaniu skrótu klawiatury

Porównując z kodem, który pokazuje Rysunek 4.4.8 (str. 102), widać, że dodałem na początku procedury dodatkowe linie. Ich zadaniem jest przygotowanie argumentów (słownik ***args***) dla wywołania funkcji ***.keymap_items.new()***. Jako wartości początkowej używam słownika ***hotkey_defaults***. Następnie sprawdzam, czy wśród kolekcji API Blendera istnieją preferencje tej wtyczki (powinny istnieć). Jeżeli tak – przypisuję je do bardziej „poręcznej" w dalszym użyciu zmiennej ***prf*** i nadpisuję ich wartościami odpowiednie pozycje słownika ***args***. Jeżeli użytkownik wybrał klawisz *None* (por. Rysunek 4.5.8) – opuszczam tę procedurę bez dodania jakiegokolwiek skrótu. W przeciwnym razie – dodaję, tak jak dotychczas, nowy skrót.
Na końcu procedury umieściłem pomocniczy komunikat diagnostyczny. (Nie będzie się wyświetlał, gdy przestawisz globalna stałą ***DEBUG*** na 0).

Aby sprawdzić, czy ten zmieniony kod działa, aktywuj wtyczkę i zmień w jej ustawieniach klawisz skrótu na E. następnie zamknij Blender i powtórnie go otwórz. Kolejne komunikaty zobaczysz w konsoli (Rysunek 4.5.11):

```
Console ⊠  Tasks  Problems
Debug Server

Keyboard shortcut set to: [D] (from add-on preferences)
object_booleans: registered
Keyboard shortcut set to: [E] (from add-on preferences)
object_booleans: registered
```


Rysunek 4.5.11 Efekt działania nowego kodu

Copyright Witold Jaworski, 2011-2019.

Po ponownym uruchomieniu Blendera skrypt odczytał nowe ustawienia i przypisał skrót wywołujący *pie menu* do klawisza E. Tyle, że zmianę wprowadzaną z takim opóźnieniem użytkownik potraktuje jako błąd: w Blenderze wszelkie zmiany są wprowadzane od razu (w jego UI nie ma nawet żadnych przycisków w rodzaju *Update*) Aby zmiany wprowadzone przez użytkownika w panelu *Preferences* były od razu widoczne, dodałem do klasy ***Preferences*** specjalną metodę ***on_update()***. Przekazałem ją (jako opcjonalny argument ***update***) do funkcji inicjalizującej każdą z właściwości klasy. Dzięki temu będzie wywoływana, gdy użytkownik w jakikolwiek sposób zmieni ich wartość (Rysunek 4.5.12):

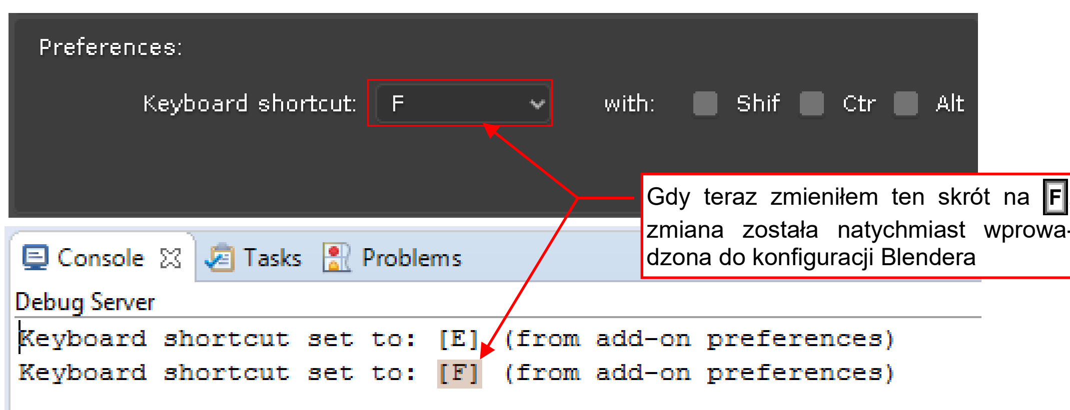
```
class Preferences(bpy.types.AddonPreferences):
    '''This class provides the user pssibility of altering the keyboard shortcut
       assigned to the Boolean pie menu
    '''
    bl_idname = __name__ #do not change this line

    def on_update(self, context):
        unregister_keymap()
        register_keymap()

    shift : BoolProperty(name = "Shift", description= "Use the [Shift] key",
                             default=hotkey_defaults["shift"], update = on_update)
    ctrl  : BoolProperty(name = "Ctrl", description= "Use the [Ctrl] key",
                             default=hotkey_defaults["ctrl"], update = on_update)
    alt   : BoolProperty(name = "Alt", description= "Use the [Alt] key",
                             default=hotkey_defaults["alt"], update = on_update)
    key   : EnumProperty(items = [('NONE','None',"No hotkey")] +
                  [tuple([chr(i),chr(i),"[%s] key" % chr(i)]) for i in range(65, 91)],
                   name = "Keyboard key",
                   description = "Selected keyboard key",
                   default = hotkey_defaults["type"],
                   update = on_update
                 )

          metoda draw() – bez zmian
```

Pomocnicza funkcja, wywoływana przy każdej zmianie właściwości

Rysunek 4.5.12 Wprowadzenie funkcji wywoływanej przy zmianie właściwości API

Aby uaktualnić mapę klawiszy Blendera, ***on_update()*** najpierw wywołuje metodę usuwającą dotychczasowy skrót wtyczki (***unregister_keymap()***). Następnie wywoływana jest metoda rejestrująca nowy skrót (***register_keymap()***), zgodny z aktualnymi ustawieniami w panelu *Preferences*. Podobnie jak słownik ***hotkey_defaults***, funkcja ***on_update()*** musi być umieszczona w skrypcie przed wywołaniem funkcji ***BoolProperty()***, ***EnumProperty()*** inicjalizujących właściwości, do których ją przypisujemy.

Gdy teraz załadujesz naszą wtyczkę, to każdej zmianie skrótu w jej panelu konfiguracji będzie towarzyszyła odpowiednia zmiana konfiguracji Blendera (Rysunek 4.5.13):



Rysunek 4.5.13 Zmiana klawisza skrótu wywołuje natychmiastową zmianę ustawień Blendera

Pozostało tylko jeszcze założyć na *blenderwiki.org* stronę z opisem tego dodatku oraz jego *bug tracker*, na ewentualne zgłoszenia od użytkowników[1]. Nie jest to jednak tematem tej książki. Pełną wersja skryptu, który tu opracowaliśmy, znajdziesz na str. 162.

Gdy zakończysz pracę nad wtyczką, koniecznie przełącz jej stałą ***DEBUG*** na 0 (por. Rysunek 4.5.3, str. 107). W przeciwnym razie klient debugera PyDev będzie się uaktywniać przy każdym otwieraniu Blendera. To może zakłócić testowanie kolejnego skryptu, nad którym zaczniesz pracować.

Podsumowanie

- Do testowania panelu preferencji wtyczki należy ją zainstalować w Blenderze (w oknie *Blender Preferences* – por. str. 106);
- Dalsza edycja i debugowanie takiego pliku Pythona jest możliwe: należy tylko wstawić do projektu PyDev link do jego pliku (str. 106) i dodać na początek tego pliku linie z wywołaniem klienta debuger (str. 107);
- Ustawienia wtyczki należy zaimplementować w klasie pochodnej klasy ***bpy.types.AddonPreferences***. Identyfikatorem takiej klasy (***bl_idname***) musi być nazwa pliku wtyczki (bez rozszerzania „*.py*”). Aby wyświetlić panel z ustawieniami wtyczki, klasa musi implementować funkcję ***draw()*** (str. 107);
- Klasa preferencji musi być także zarejestrowana w Blender API (w procedurze ***register()***) i odrejestrowana (w procedurze ***unregister()***) (str. 108);
- Podczas działania wtyczki, aktualne ustawienia z jej panelu użytkownika można odczytać z kolekcji ***bpy.context.preferences.addons***, gdzie kluczem jest nazwa pliku wtyczki (str. 111);
- Aby natychmiast reagować na zmiany ustawień wtyczki, przypisz do jej właściwości metody ***update()***, wywoływane przez Blender przy każdej zmianie (str. 112);

[1] Zgłoszenia błędów, otrzymywane od użytkowników, często są źródłem dodania nowych opcji i funkcji. Takie „dopiski” to naturalny rozwój każdego programu.

Copyright Witold Jaworski, 2011-2019.

Dodatki

W tej części umieściłem różne opcjonalne materiały pomocnicze. Mogą Ci się przydać, gdy czegoś nie jesteś pewien w trakcie czytania tekstu głównego.

Rozdział 5. Szczegóły instalacji

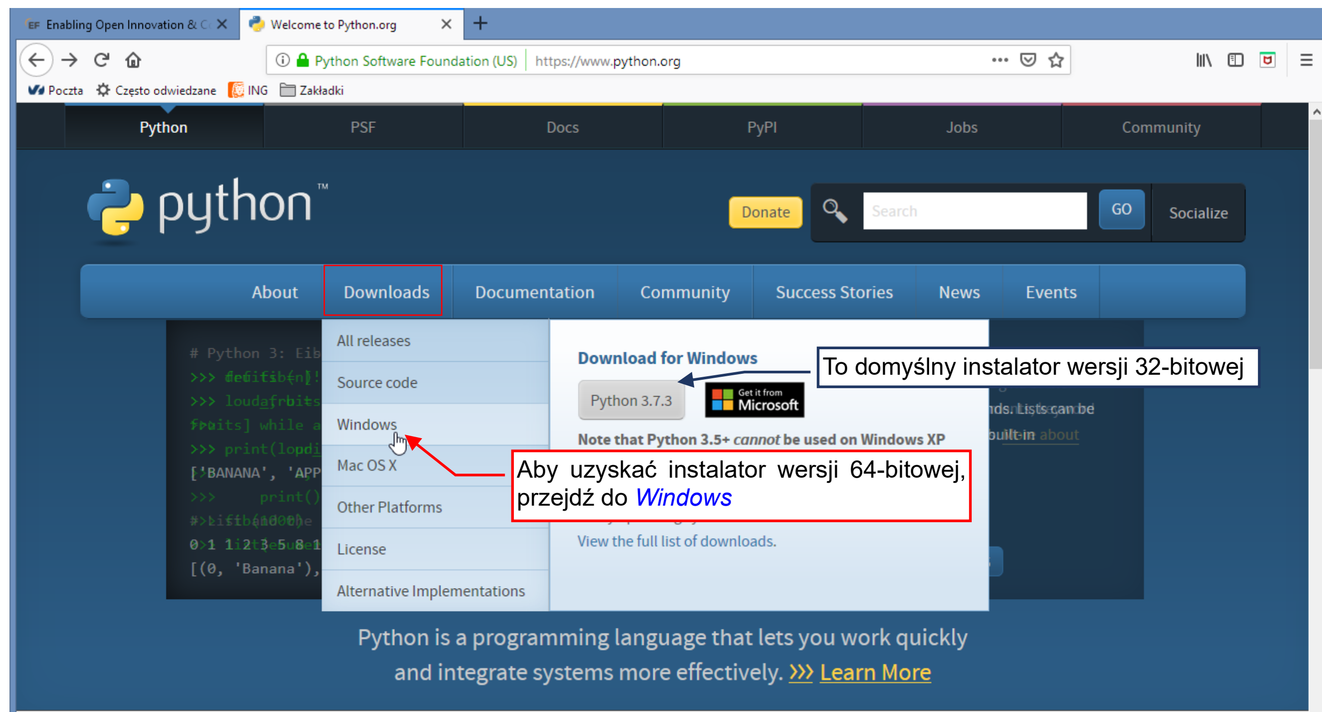
W tym rozdziale umieściłem opis szczegółów instalacji oraz konfiguracji Pythona, Eclipse i PyDev. Zrobiłem to na wszelki wypadek, gdybyś „utknął” na jakimś drobiazgu.

Copyright Witold Jaworski, 2011-2019.

5.1 Szczegóły instalacji Pythona

Od 2019r Eclipse jest dostępne wyłącznie w wersji 64-bitowej. Stąd dla pewności, że wszystko będzie działać, instaluję także Pythona w wersji 64-bitowej.

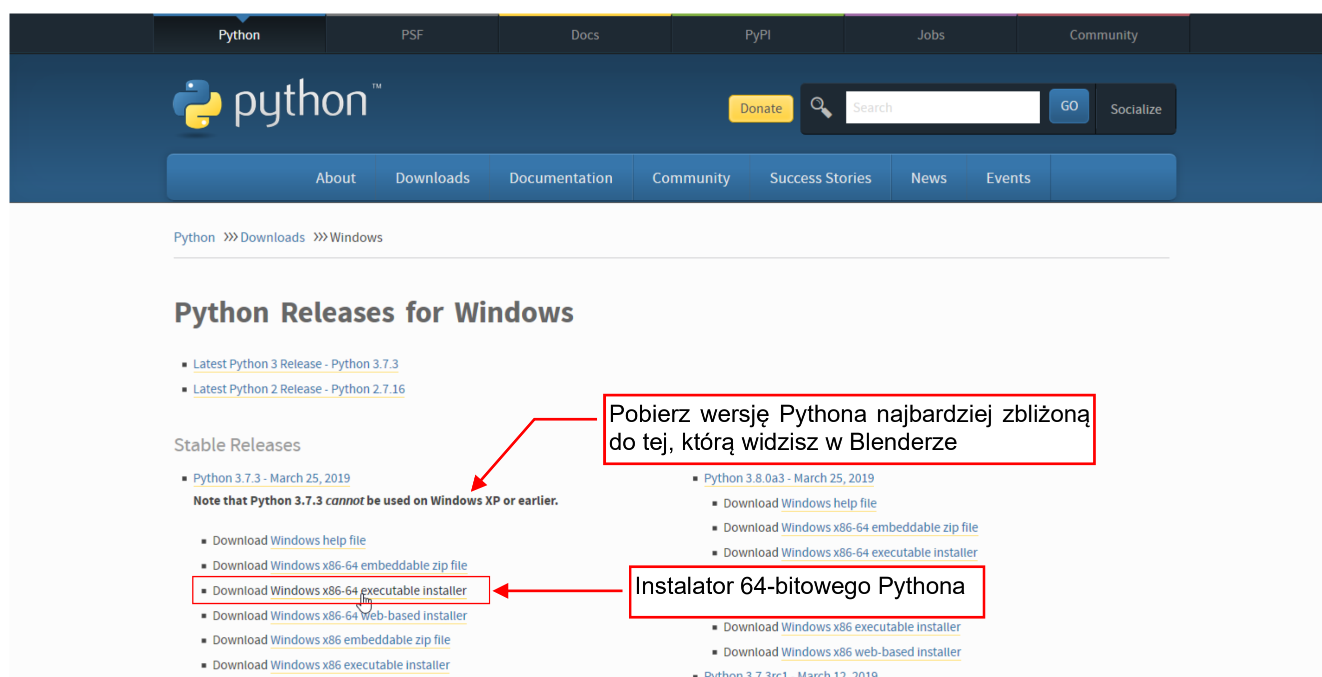
Zaczynamy od wejścia na portal tego projektu: www.python.org, wybierz tam menu ***Downloads***, a z niego – odpowiednią platformę (Rysunek 5.1.1):



Rysunek 5.1.1 Strona główna projektu Pythona

Domyślny przycisk („Python 3.7.3” na ilustracji) pobiera instalator w wersji 32-bitowej, stąd musiałem go zignorować. Zamiast tego kliknąłem w nazwę systemu (***Windows***), aby przejść do listy wszystkich wariantów.

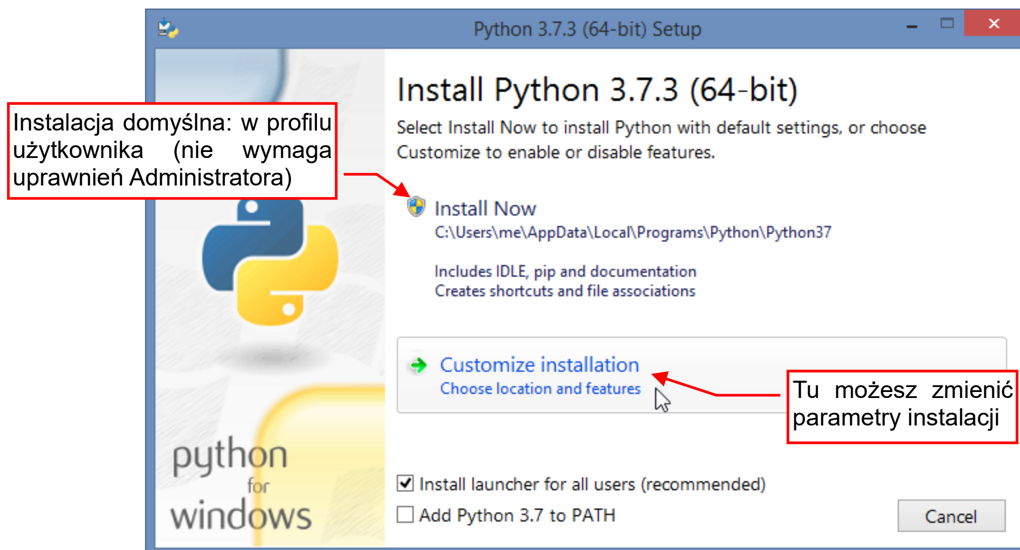
Na następnej stronie wybieram do pobrania z tej listy instalator w wersji 64-bitowej (Rysunek 5.1.2):



Rysunek 5.1.2 Strona z wersjami instalacyjnymi interpretera

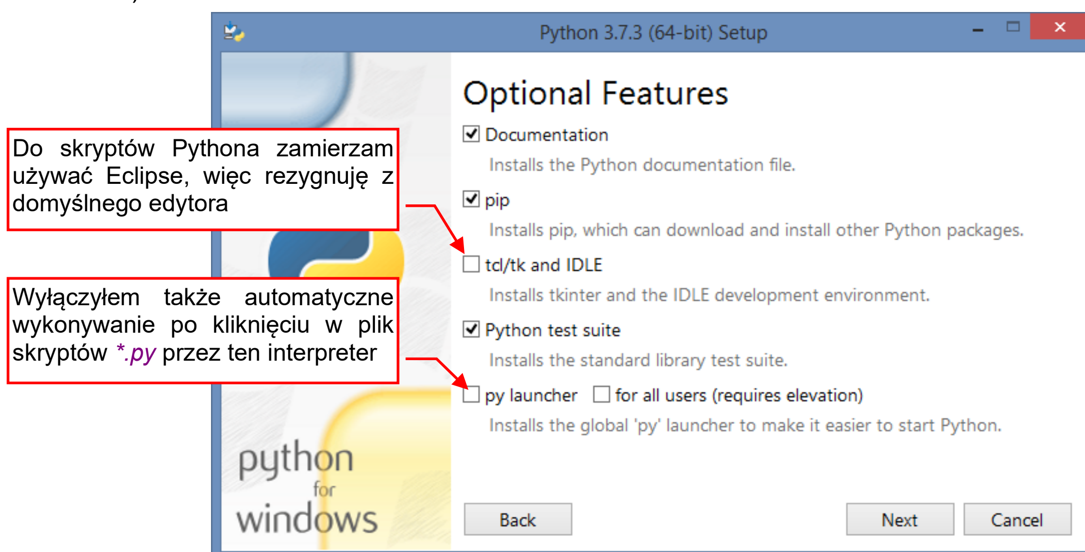
Wybierz z niej taką wersję Pythona, której używa Twój Blender. (Jeżeli nie możesz znaleźć identycznej – wybierz jak najbardziej zbliżoną). Kliknięcie w link uruchamia pobranie na lokalny komputer programu instalacyjnego. W przypadku pokazanym na ilustracji jest to program instalacyjny o nazwie *python-3.7.3-amd64.exe*. Pomimo nazwy, można go używać także na komputerach z procesorami Intela.

Po uruchomieniu możesz zmienić domyślne opcje konfiguracji albo od razu zainstalować Pythona z ustawieniami domyślnymi (Rysunek 5.1.3):



Rysunek 5.1.3 Pierwszy ekran programu instalacyjnego

Nie lubię niepotrzebnych dodatków, więc zdecydowałem się zmienić (ograniczyć) zakres tej instalacji do niezbędnego minimum. Wybrałem opcję ***Customize installation***, co spowodowało otwarcie następującego ekranu (Rysunek 5.1.4):



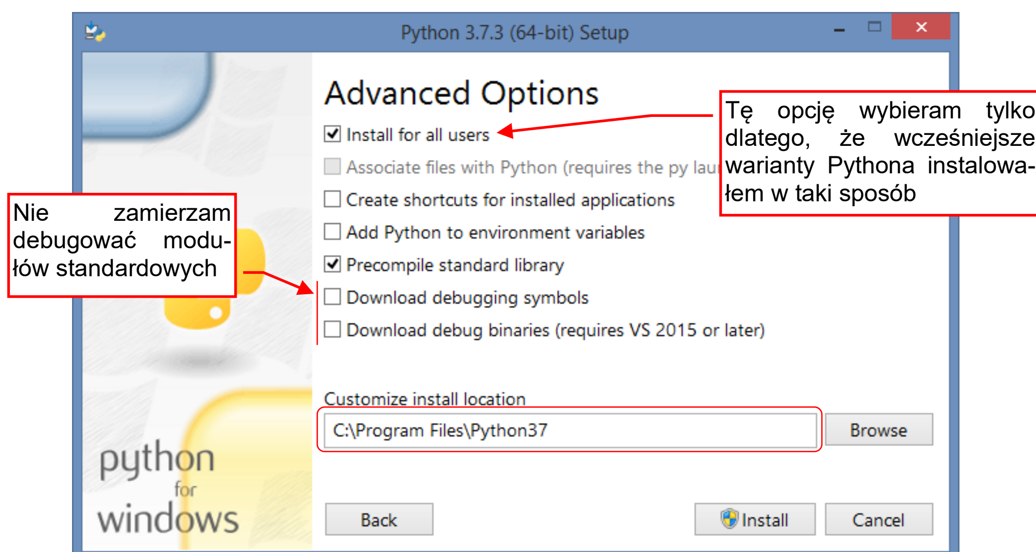
Rysunek 5.1.4 Pierwszy ekran opcji instalacji

Nie przepadam za domyślnymi edytorami Pythona, więc nie instaluję *td/tk and IDLE*. Do obsługi większych projektów zamierzam używać Eclipse, a do szybkiego zajrzenia do skryptu w zupełności wystarczy Notepad++ lub podobny program.

Wyłączyłem także *py launcher*, który umożliwia uruchamianie skryptów **.py* tak, jak innych programów wykonywalnych (poprzez podwójne kliknięcie w ikonę pliku). Skryptów Pythona używam tylko w programach graficznych, więc nie chcę przez przypadek uruchamiać tych skryptów poza przewidzianym środowiskiem.

Copyright Witold Jaworski, 2011-2019.

Na kolejnym ekranie wybrałem opcje instalacji dla wszystkich użytkowników tego komputera (Rysunek 5.1.5):



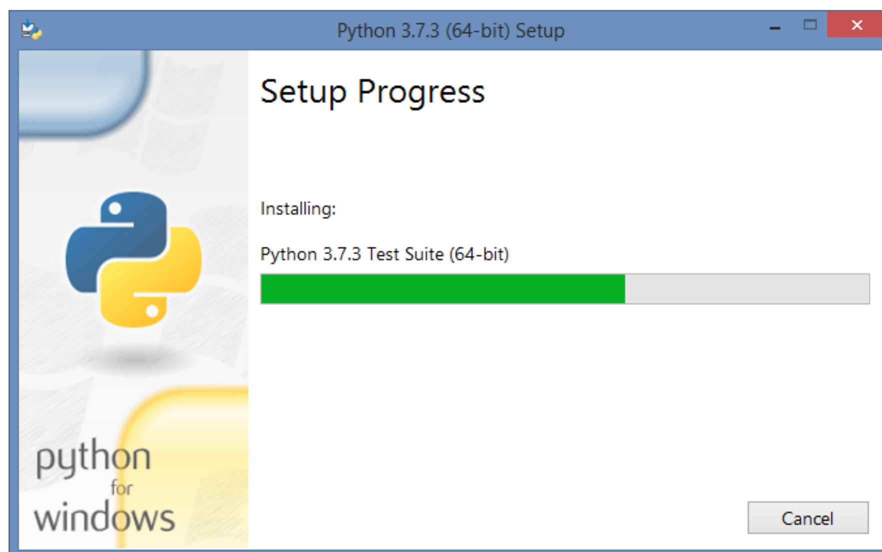
Rysunek 5.1.5 Drugi ekran opcji instalacji

Na tym ekranie wybrałem opcję ***Install for all users*** wyłącznie dlatego, że:

- Wcześniejsze interpretery Pythona instalowałem także z tą opcją (kiedyś była domyślna);
- Pliki programu są instalowane w ogólnym folderze programów, a nie w profilu użytkownika (nie lubię instalowania programów w folderach użytkownika, bo potem trudno mi jest je znaleźć);
- Posiadam wymagane dla tej opcji uprawnienia Administratora;

Wyłączyłem także instalację informację umożliwiającej debugerowi śledzenie funkcji ze standardowych modułów Pythona (nie jest mi to potrzebne).

Po kliknięciu w przycisk ***Install*** rozpoczyna się instalacja (Rysunek 5.1.6):



Rysunek 5.1.6 Instalacja interpretera Pythona

Ten proces trwa około minuty.

Na koniec pojawia się ekran z informacją o powodzeniu (Rysunek 5.1.7):

Rysunek 5.1.7 Ostatni ekran instalacji Pythona

Naciskasz ***Close*** i to kończy całą instalację.

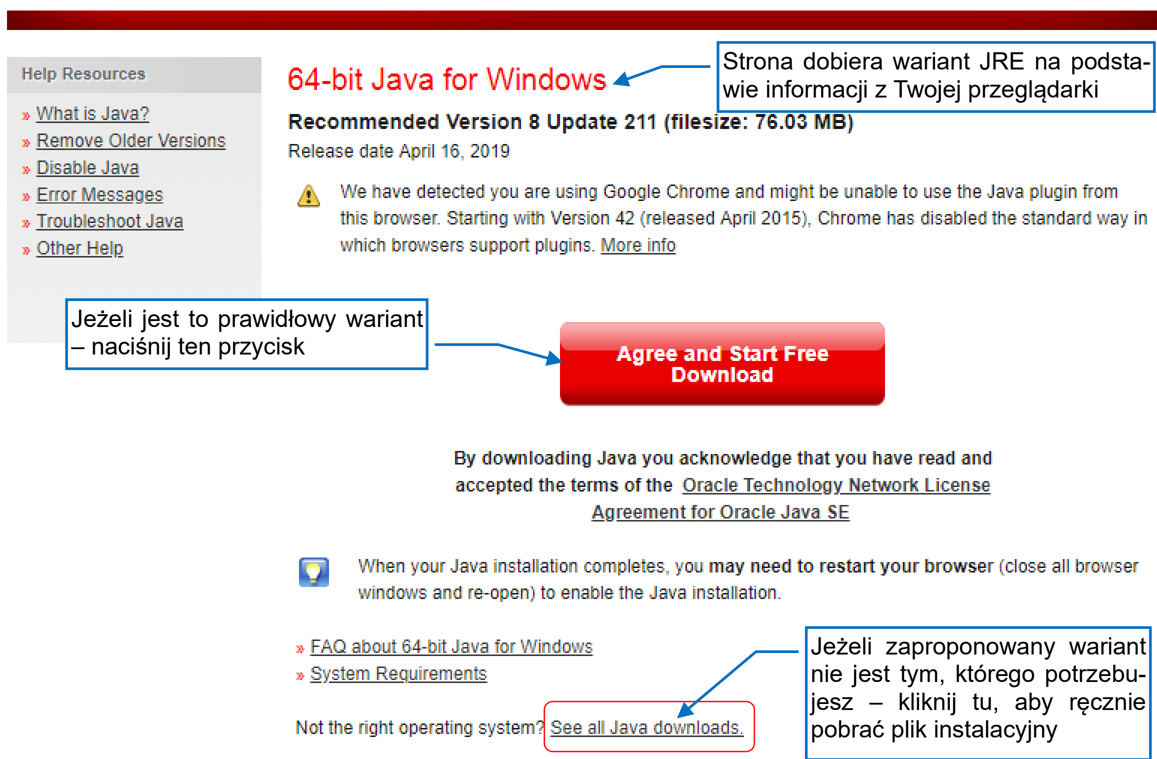
Copyright Witold Jaworski, 2011-2019.

5.2 Szczegóły instalacji Java Runtime Environment

Eclipse jest aplikacją Javy, i od 2019r wymaga standardowej maszyny wirtualnej Javy (JRE) w wersji 64-bitowej. Wersję instalacyjną JRE można pobrać ze strony www.java.com. Na tej stronie znajdziesz także wskazówki, jak sprawdzić, czy JRE jest już zainstalowane na Twoim komputerze (Rysunek 5.2.1):

Rysunek 5.2.1 Strona główna portalu java.com (ekran z maja 2019)

Jeżeli okaże się, że nie masz jeszcze 64-bitowego JRE, przejdź na stronę pobrań (***Java Download***):



Rysunek 5.2.2 Automatyczne wykrywanie wariantu JRE do pobrania

Portal *java.com* rozpoznaje system operacyjny na podstawie informacji otrzymanych od przeglądarki. Jeżeli zaproponuje 32-bitową Javę – kliknij w link ***See all Java downloads*** u dołu ekranu.

Na stronie ze wszystkimi wersjami instalacyjnymi wybierz wersję 64-bitową (Rysunek 5.2.3):

Help Resources
» Troubleshoot Java
Java 7
» Where can I get Java 7?
JDK
» Looking for the JDK?

Java Downloads for All Operating Systems

Recommended Version 8 Update 211
Release date April 16, 2019

Select the file according to your operating system from the list below to get the latest Java for your computer.

> Remove Older Versions
> What is Java?

By downloading Java you acknowledge that you have read and accepted the terms of the Oracle Technology Network License Agreement for Oracle Java SE

Windows — Which should I choose?

Windows Online
filesize: 1.95 MB
Instructions

Windows Offline
filesize: 66.37 MB

Windows Offline (64-bit)
filesize: 76.03 MB
Instructions

Dla Windows wybierz tę wersję

After installing Java, you may need to restart your browser in order to enable Java in your browser.

If you use 32-bit and 64-bit browsers interchangeably, you will need to install both 32-bit and 64-bit Java in order to have the Java plug-in for both browsers. » FAQ about 64-bit Java for Windows

Rysunek 5.2.3 Ręczny wybór wariantu JRE

Po pobraniu instalatora uruchom go na swoim komputerze. Proces instalacji JRE nie wymaga od użytkownika podawania jakichkolwiek parametrów.

- Eclipse wymaga 64-bitowego JRE. Często nawet na 64-bitowych komputerach różne aplikacje instalują 32-bitowe JRE. Stąd zazwyczaj możesz stwierdzić, że na Twoim komputerze jest dostępna Java, ale w wariancie 32-bitowym, z którym Eclipse nie będzie działać. Pamiętaj, że na tym samym komputerze możesz bez problemu mieć zainstalowane obydwie wersje JRE: 32-bit i 64-bit.

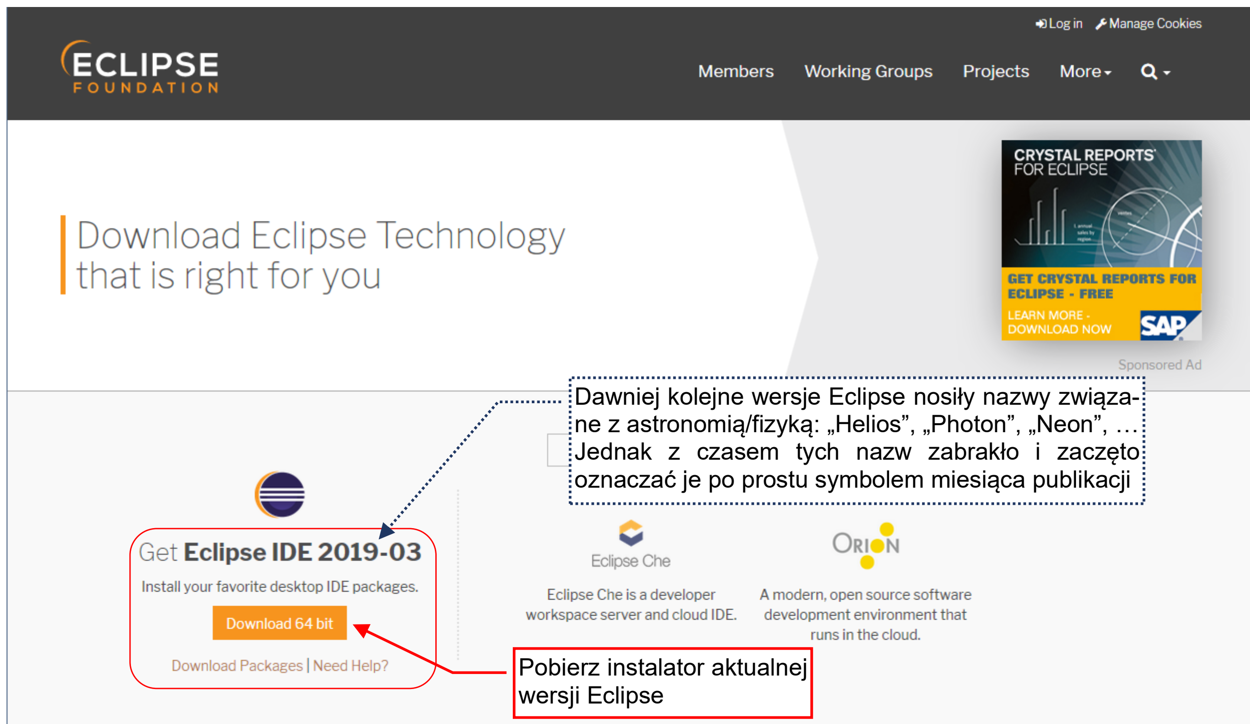
- W przypadku Mac OS należy sprawdzić aktualne uwagi o instalacji Eclipse. W chwili, gdy piszę tę książkę (maj 2019), w środowisku Mac OS instalować cały najnowszy JDK (Java Development Kit) w wariancie 64-bit. (Każdy JDK zawiera także JRE). W przeciwnym razie Eclipse będzie wyświetlać komunikaty o błędach.

Copyright Witold Jaworski, 2011-2019.

5.3 Szczegóły instalacji Eclipse i PyDev

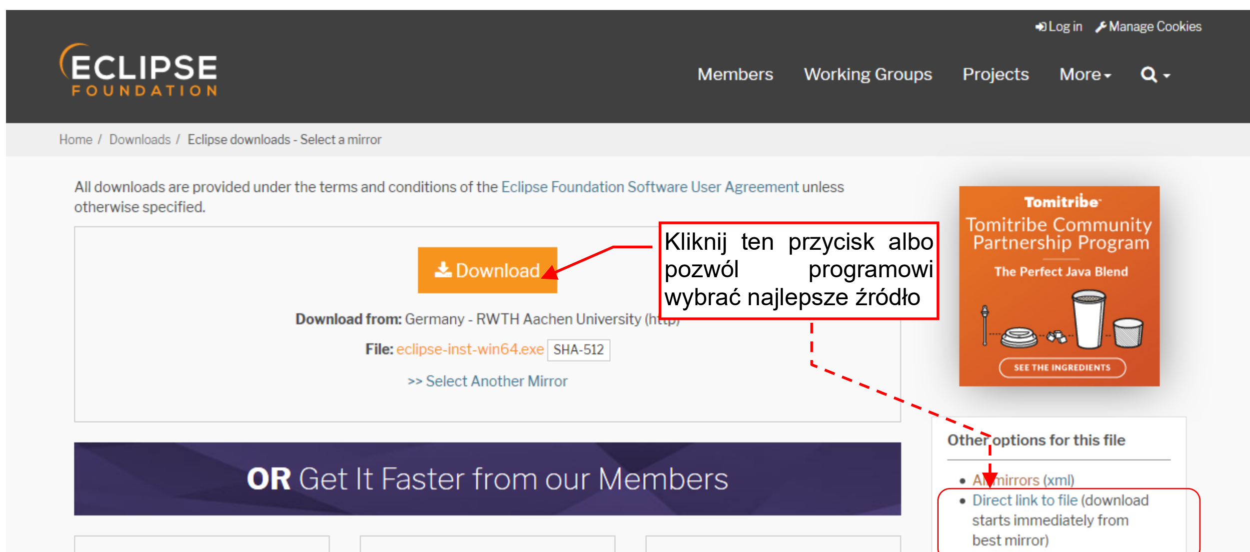
- Na początku sprawdź, czy masz na swoim komputerze dostępny *Java Runtime Environment* (*Java JRE*) w wariancie 64-bitowym. W systemie Windows powinieneś w panelu sterowania mieć ikonę „Java”. Jeżeli jej tam nie ma — pobierz 64-bitowe JRE z java.com i zainstaluj[1].

Zacznijmy od pobrania Eclipse. Wejdź na stronę http://www.eclipse.org/downloads (Rysunek 5.3.1):



Rysunek 5.3.1 Pobranie instalatora Eclipse (ekran z maja 2019)

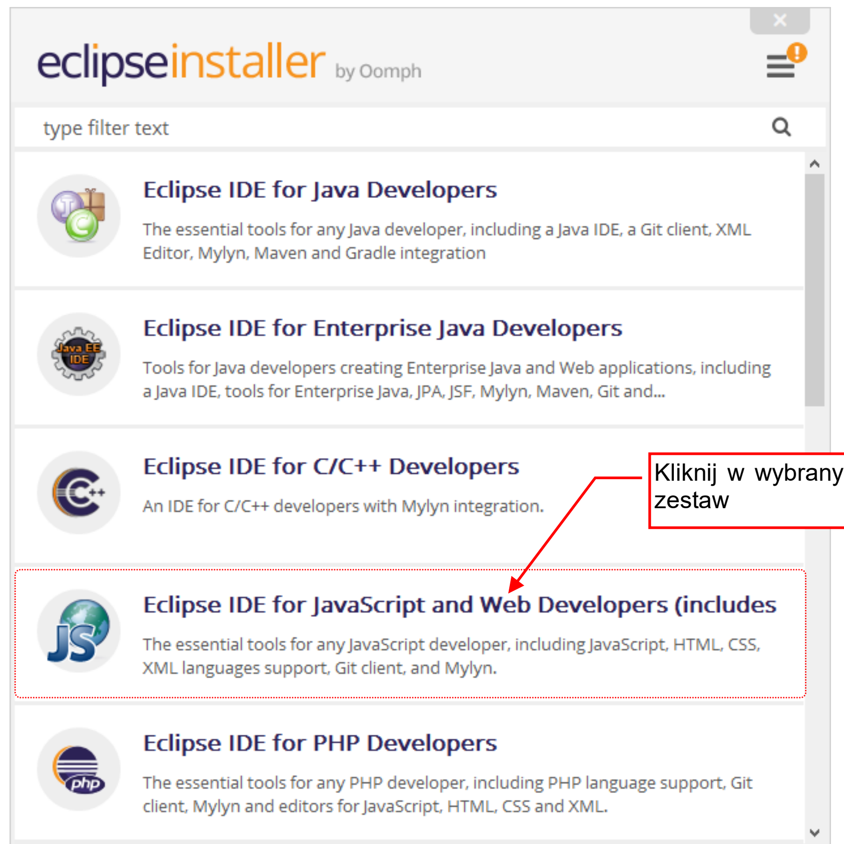
Kliknij w przycisk ***Download 64 bit***, aby przejść na stronę wyboru serwera do pobrania (Rysunek 5.3.2):



Rysunek 5.3.2 Wybór serwera do pobrania

[1] Uwaga: w niektórych odmianach Linuxa, jak Ubuntu, domyślną maszyną wirtualną (VM) Javy jest GCJ. Eclipse działa na niej wolniej niż na JVM z www.java.com. Co więcej, nawet po pobraniu i wgraniu do Ubuntu tej nowej Javy, nie jest ona „maszyną” domyślną! To trzeba poprawiać „ręcznie”. Szczegółowe instrukcje dot. instalacji Javy i Eclipse na Ubuntu — patrz https://help.ubuntu.com/community/EclipseIDE.

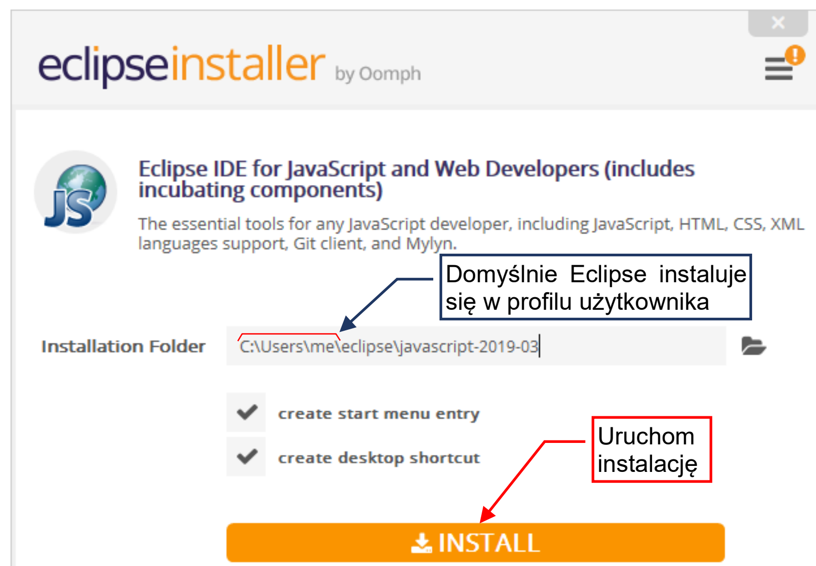
Po uruchomieniu programu instalacyjnego zobaczysz takie okno (Rysunek 5.3.3):



Rysunek 5.3.3 Wybór wariantu Eclipse

W istocie, Eclipse to coś w rodzaju „ramowego" środowiska programisty. Te „ramy" można przystosować do pracy z konkretnym językiem/językami programowania za pomocą odpowiednich dodatków. Program instalacyjny udostępnia typowe konfiguracje Eclipse. Nie ma wśród nich gotowego zestawu dla Pythona, więc złożymy go sobie sami. Można na przykład wybrać „prawie pusty" zestaw ***Eclipse for Testers*** (znajduje się w dalszej części listy pokazywanej przez Rysunek 5.3.3). Ja dla siebie wybrałem ***Eclipse IDE for JavaScript***, gdyż zawiera kilka dodatkowych narzędzi, które mogą mi się przydać do innych celów.

Po kliknięciu w wybrany zestaw wyświetla się okno, gdzie można ustalić docelowy folder na pliki programu:



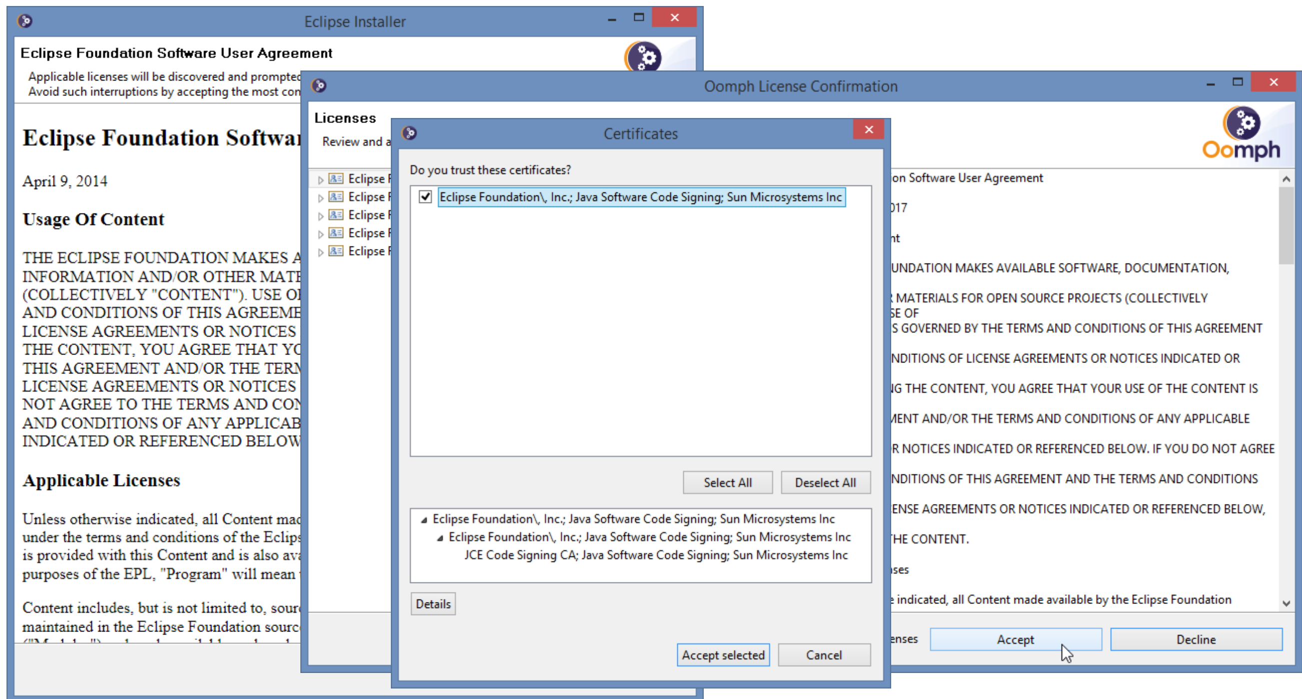
Rysunek 5.3.4 Opcje instalacji Eclipse

Aby ta instalacja odpowiadała temu, co widzisz u siebie, niczego tu nie zmieniłem.

Copyright Witold Jaworski, 2011-2019.

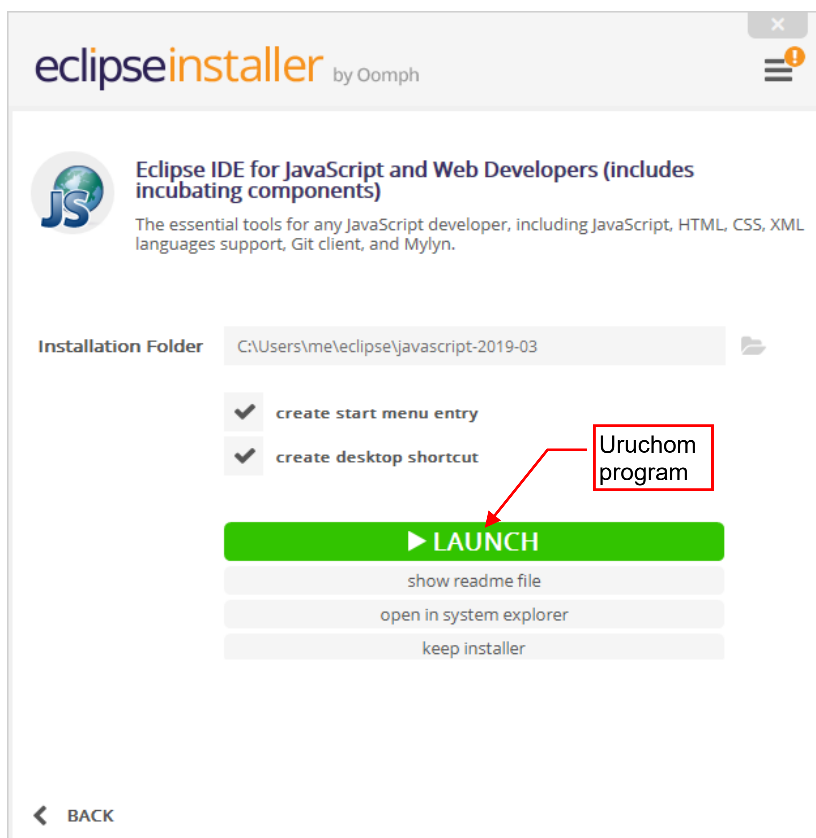
- Eclipse domyślnie instaluje swoje pliki w profilu użytkownika. Zdecydowałem się pozostawić te ustawienia domyślne, gdyż w przeciwnym razie Czytelnicy tej ksiażki mogliby się łatwo pogubić w trakcie konfiguracji IDE Pythona.

Po naciśnięciu przycisku ***INSTALL*** trzeba zaakceptować kilka umów licencyjnych i certyfikatów:



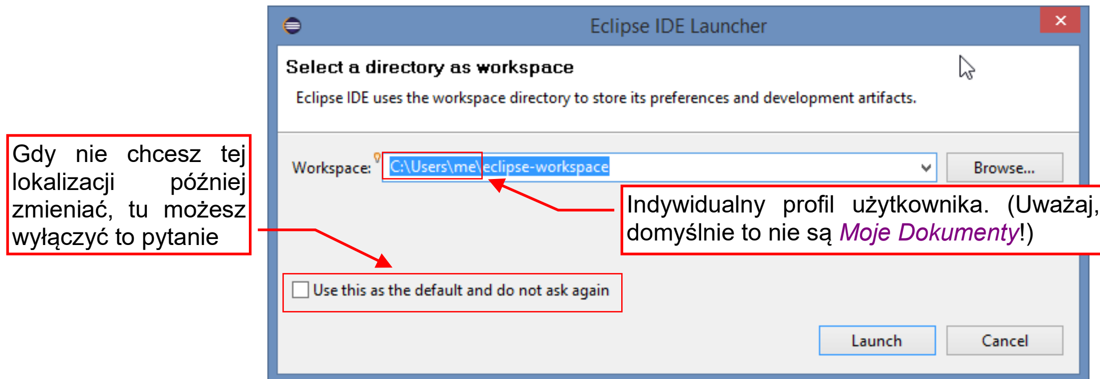
Rysunek 5.3.5 Umowy licencyjne i certyfikaty, akceptowane w trakcie instalacji Eclipse

Na koniec program wyświetli informację o zakończeniu instalacji Eclipse:



Rysunek 5.3.6 Okno programu instalacyjnego po zakończeniu instalacji

Przy uruchomieniu Eclipse najpierw pojawia się okno dialogowe, w którym należy wskazać (wystarczy potwierdzić) położenie foldera z projektami (Rysunek 5.3.7):

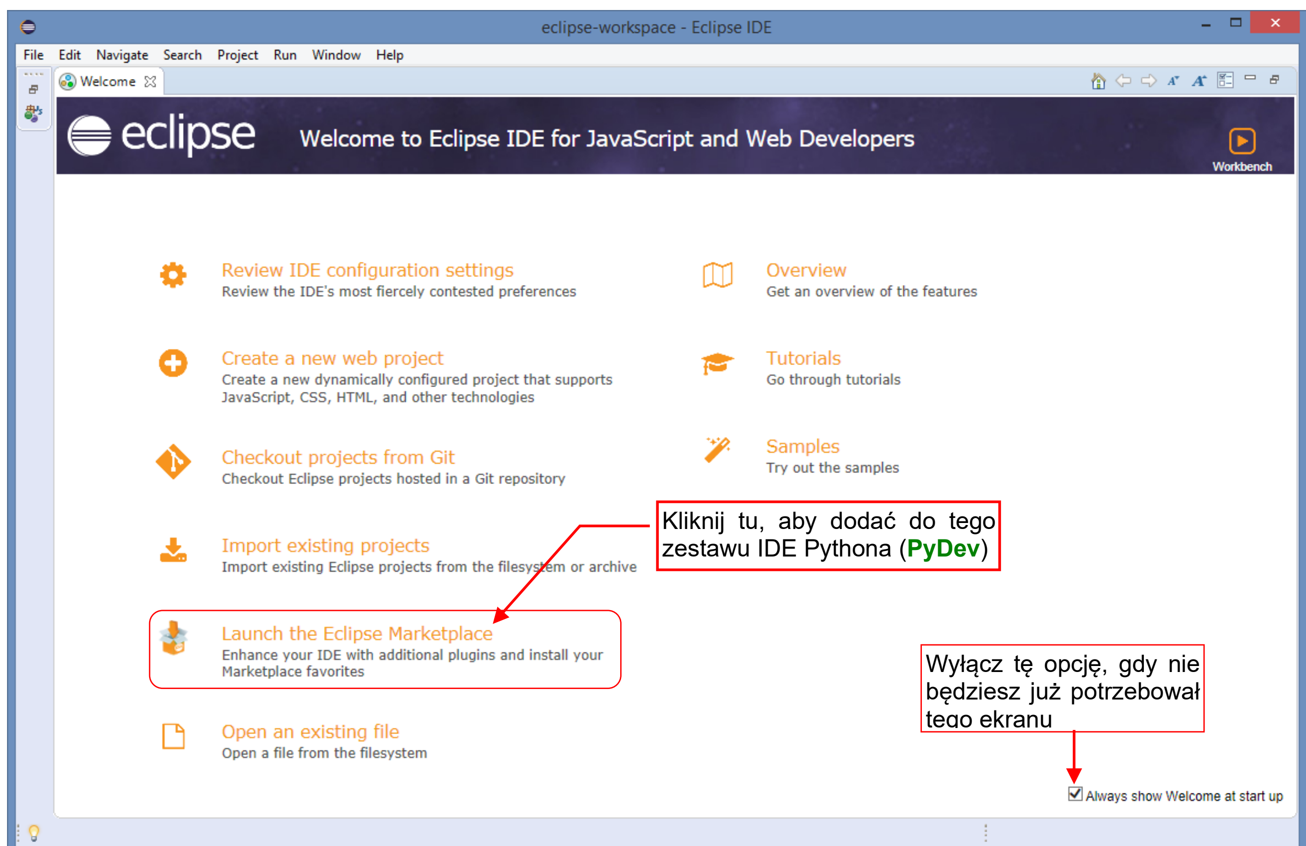


Rysunek 5.3.7 Pytanie o folder dla przyszłych projektów

Każdy projekt to oddzielny folder, zawierający parę własnych plików Eclipse oraz pliki z Twoim kodem. Zwróć uwagę, że domyślnie folder *workspace* jest umieszczony w katalogu głównym profilu użytkownika. (W tym przykładzie to użytkownik *me*). To wcale nie są *Moje Dokumenty* — tylko poziom wyżej. To proste przełożenie konwencji z *Unix/Linux*. Jeżeli jesteś przyzwyczajony, że wszystkie swoje dane trzymasz w katalogu *Moje Dokumenty* — zmień ścieżkę, wyświetloną w tym oknie. Eclipse utworzy odpowiedni folder na dysku.

- Po wskazaniu tego miejsca, Eclipse stara się zawsze w nim znaleźć i otworzyć ostatni używany projekt.

Przy pierwszym uruchomieniu Eclipse otwiera swój „ekran powitalny”:



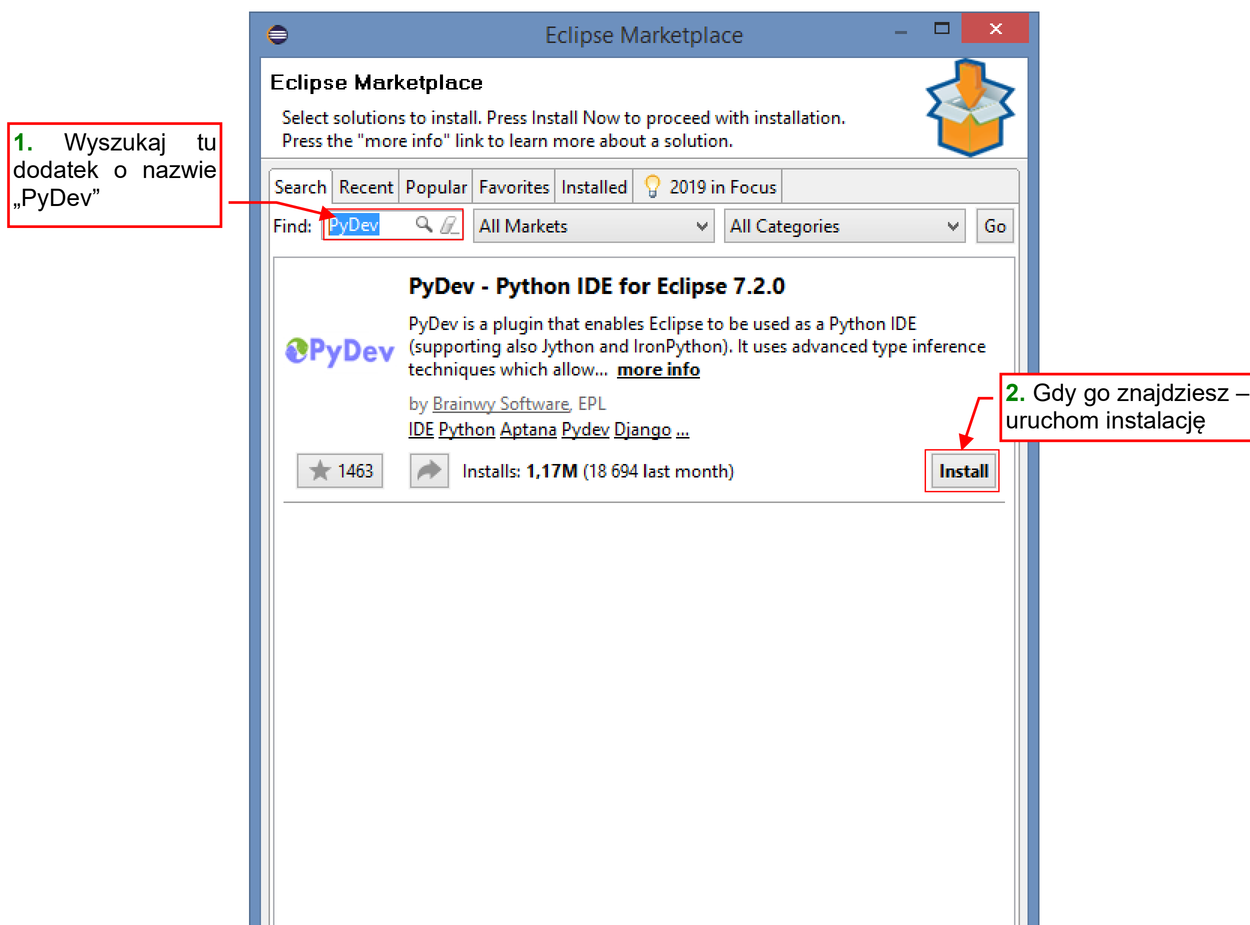
Rysunek 5.3.8 Ekran „powitalny”

Wyłącz na tym ekranie opcję ***Always show Welcome***, a następnie kliknij w ***Launch the Eclipse Marketplace***, aby rozpocząć instalację IDE dla Pythona (dodatku o nazwie **PyDev**).

Copyright Witold Jaworski, 2011-2019.

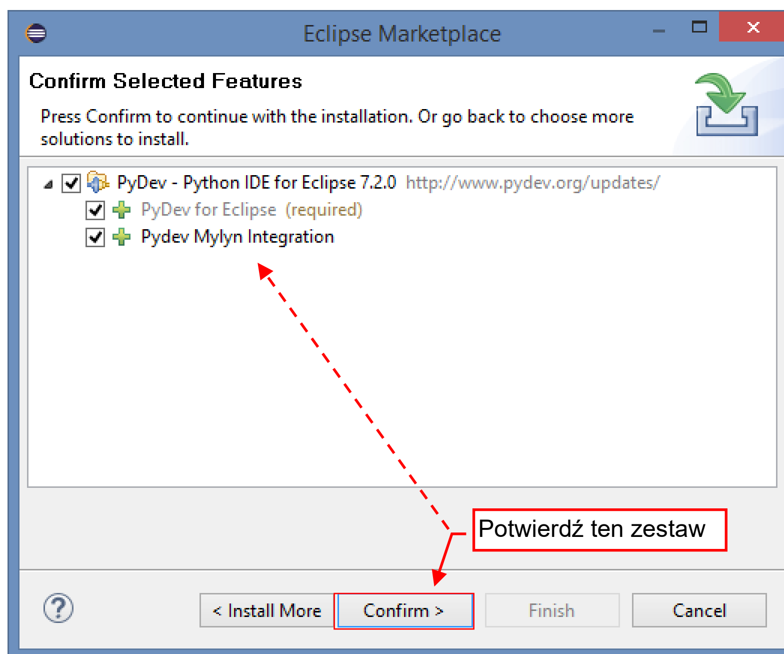
- Wywołanie ***Eclipse Marketplace…*** znajdziesz także w menu rozwijalnym ***Help***.

Spowoduje to otworzenie okna z listą wszelkich dodatków do Eclipse (Rysunek 5.3.9):



Rysunek 5.3.9 Okno wyboru wtyczki

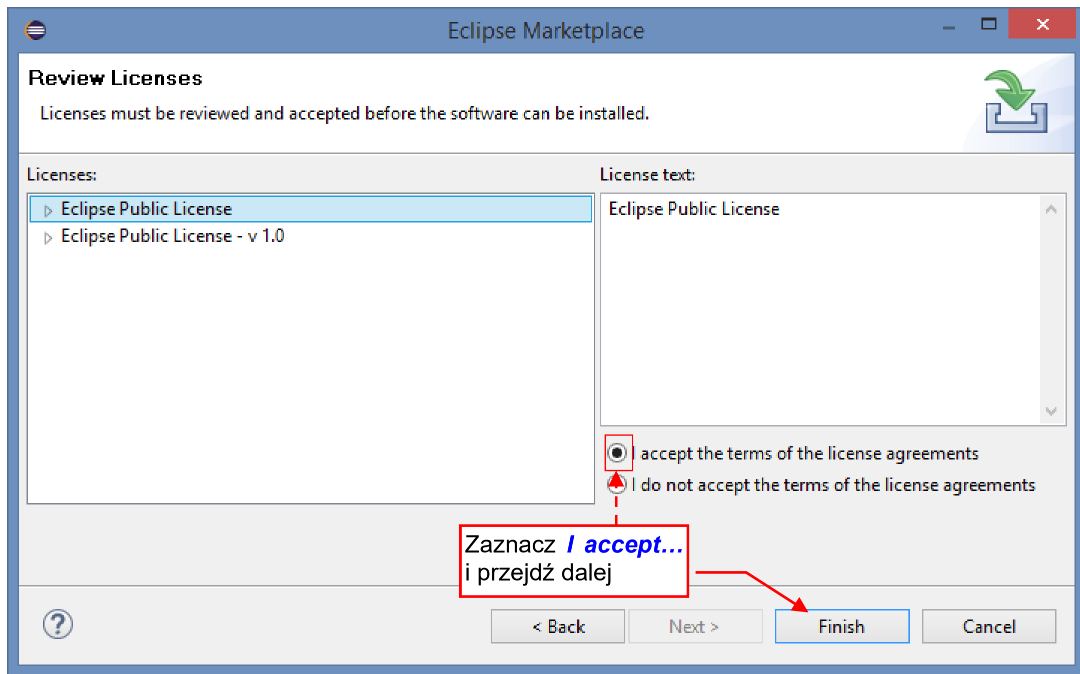
W pole ***Find:*** wpisz „PyDev” i uruchom wyszukiwanie. W odpowiedzi Eclipse powinno znaleźć dodatek jak na ilustracji. Naciśnij wówczas przycisk ***Install***. Spowoduje to pojawienie się okna do potwierdzenia składników wybranego zestawu:



Rysunek 5.3.10 Potwierdzenie składników PyDev

Niczego w zaproponowanym zestawie nie zmieniałem, tylko nacisnąłem przycisk ***Confirm***.

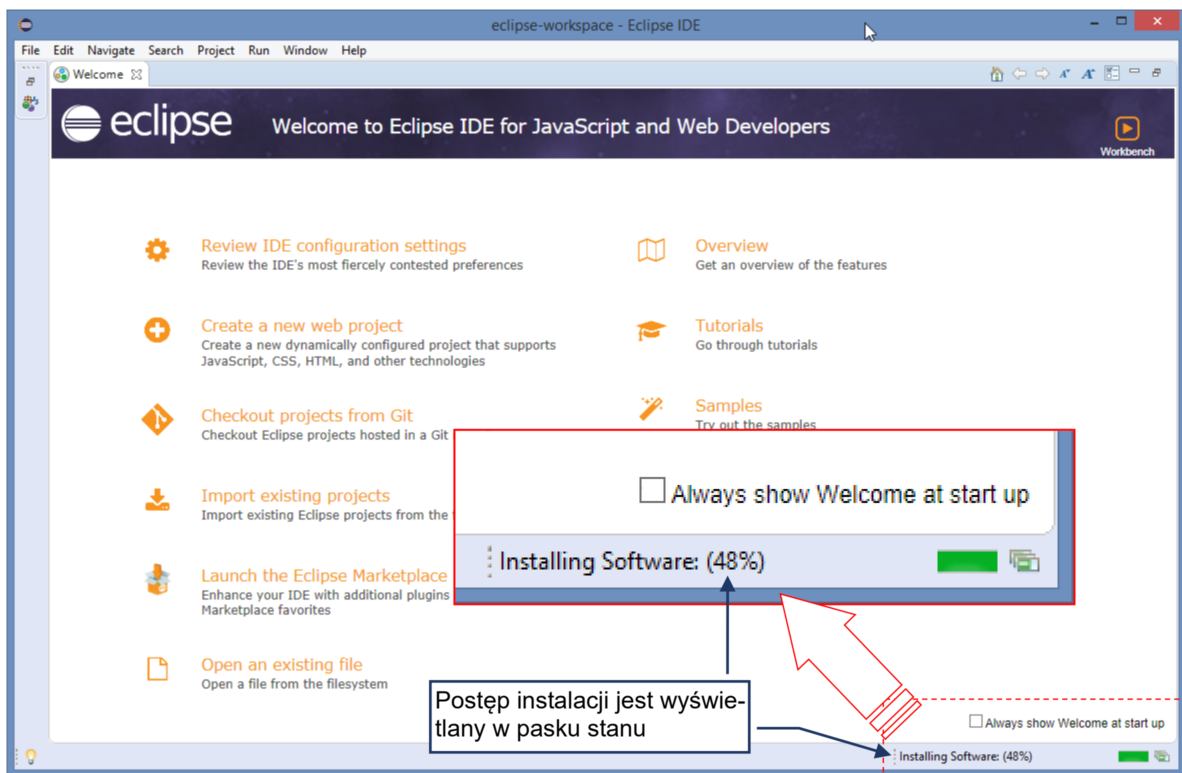
Spowoduje to otwarcie okna z kolejnymi umowami do zaakceptowania (Rysunek 5.3.11):



Rysunek 5.3.11 Potwierdzanie umów licencyjnych PyDev

Zaznacz w nim akceptację umowy i naciśnij przycisk ***Finish***. Rozpoczyna to instalację, podczas której Eclipse pobiera z Internetu wskazane komponenty.

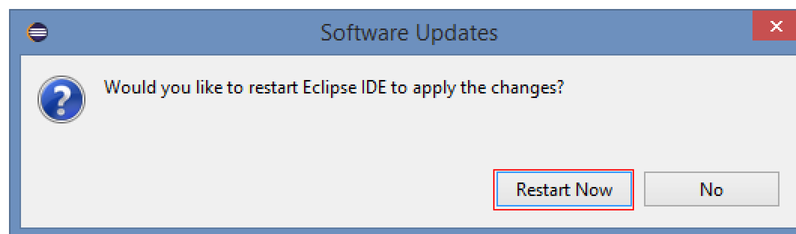
Postęp procesu jest wyświetlany w pasku stanu (Rysunek 5.3.12):



Rysunek 5.3.12 Postęp instalacji dodatku, wyświetlane w pasku stanu

Copyright Witold Jaworski, 2011-2019.

Po zakończeniu instalacji PyDev zaproponuje restart Eclipse (Rysunek 5.3.13):



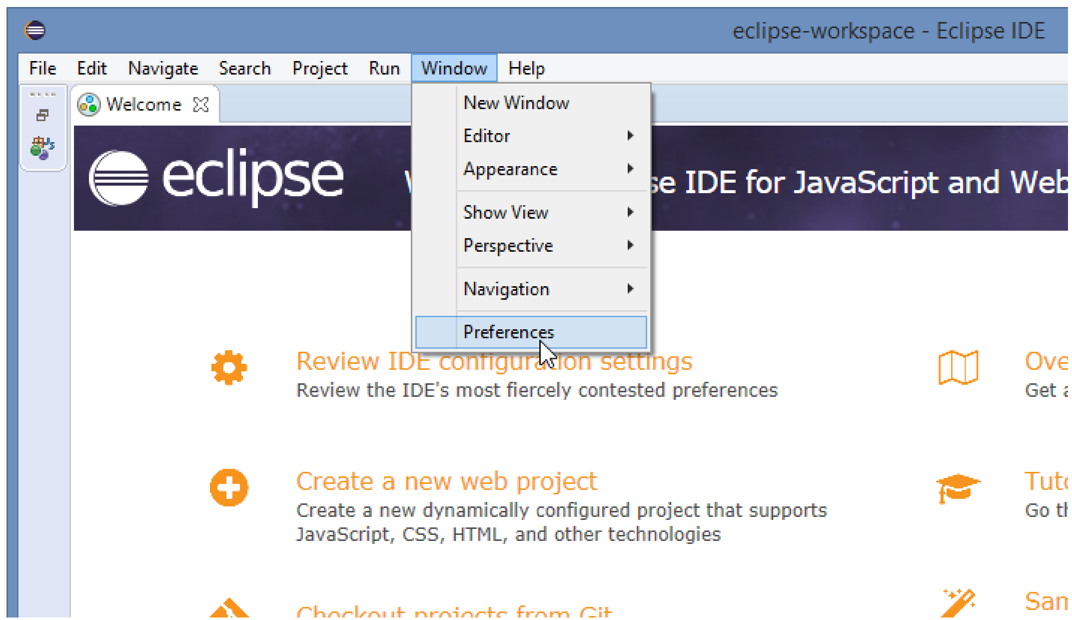
Rysunek 5.3.13 Okno końcowe instalacji dodatku Eclipse

Potwierdź tę propozycję, naciskając ***Restart Now***.

- Możesz bezpiecznie instalować „obok siebie” kolejne wersje Eclipse. (Ta informacja może się przydać w przyszłości, gdy zdecydujesz się zmienić aktualną wersję).

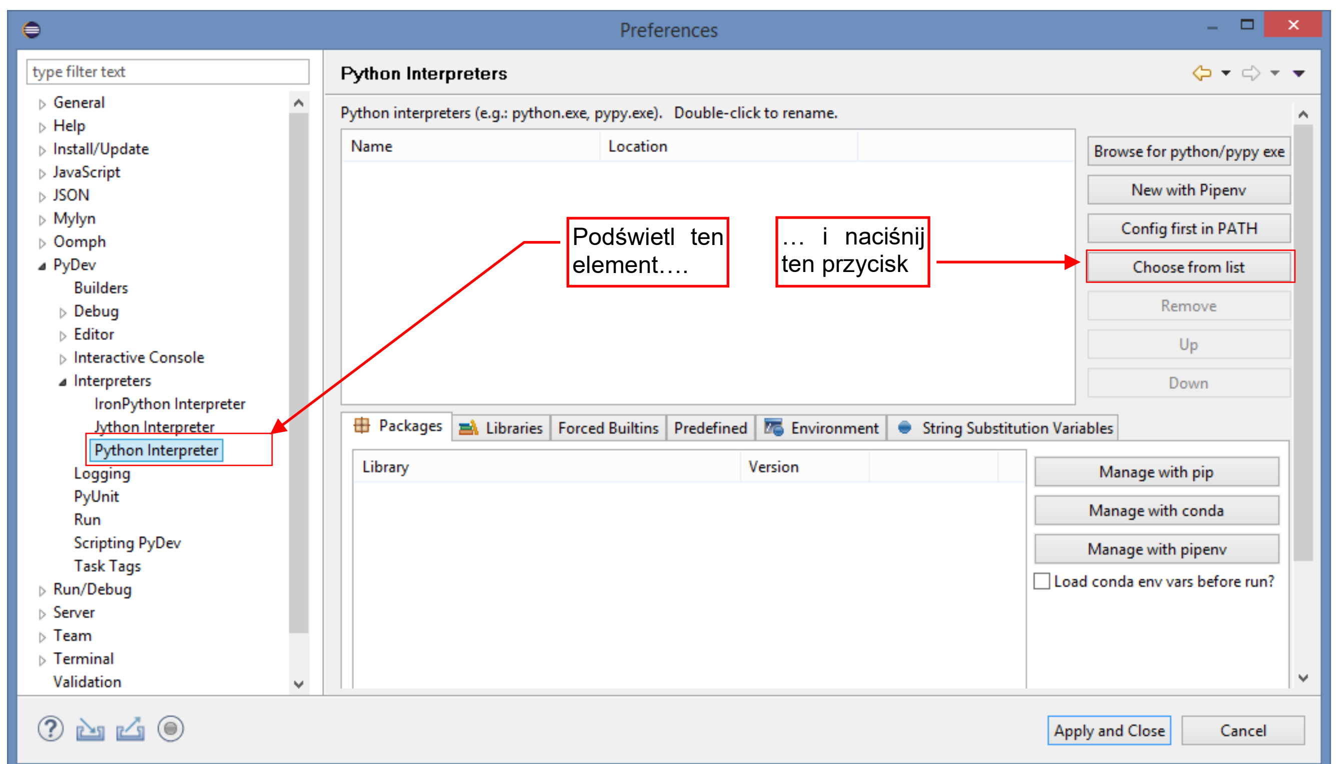
5.4 Konfiguracja PyDev

Po zainstalowaniu PyDev należy skonfigurować domyślny interpreter Pythona. To ustawienie jest zapisywane w aktualnej przestrzeni roboczej (*workspace* — por. str. 12, Rysunek 1.2.4). Aby je zmienić, wywołaj polecenie ***Window→Preferences*** (Rysunek 5.4.1):



Rysunek 5.4.1 Przejście do parametrów przestrzeni roboczej (*workspce*)

W oknie *Preferences* rozwiń sekcję ***PyDev*** i w podsekcji ***Interpreters*** podświetl pozycję ***Python Interpreter*** (Rysunek 5.4.2):

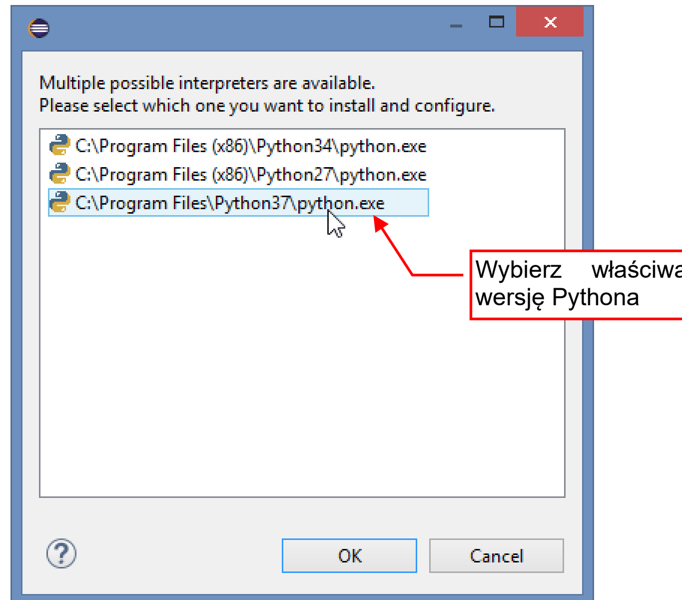


Rysunek 5.4.2 Wywołanie automatycznej konfiguracji Pythona

Teraz należy wskazać PyDev, który interpreter Pythona ma używać. W tym celu naciśnij przycisk ***Choose from list***. Uruchomi to wyszukiwanie na Twoim komputerze zainstalowanych interpreterów Pythona.

Copyright Witold Jaworski, 2011-2019.

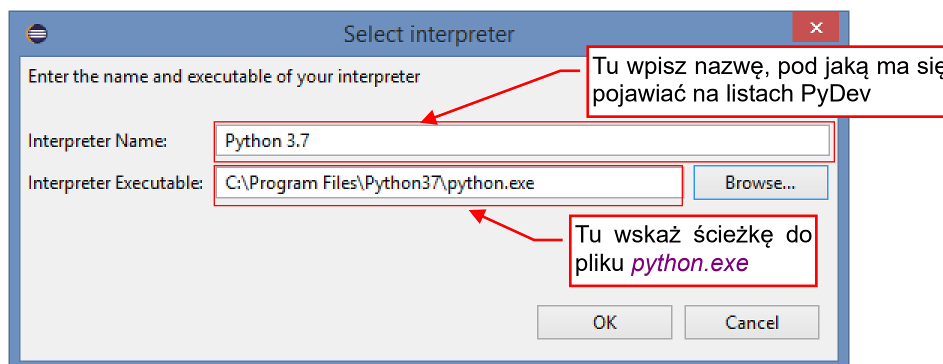
Jeżeli program znajdzie więcej niż jeden interpreter Pythona – wyświetli ich listę (Rysunek 5.4.3):



Rysunek 5.4.3 Wybór interpretera Pythona

Wybierz z niej ten interpreter, który ostatnio instalowałeś (tzn. wariant 64-bitowy, zgodny z wariantem w Blenderze - por. sekcja 1.1, str. 8).

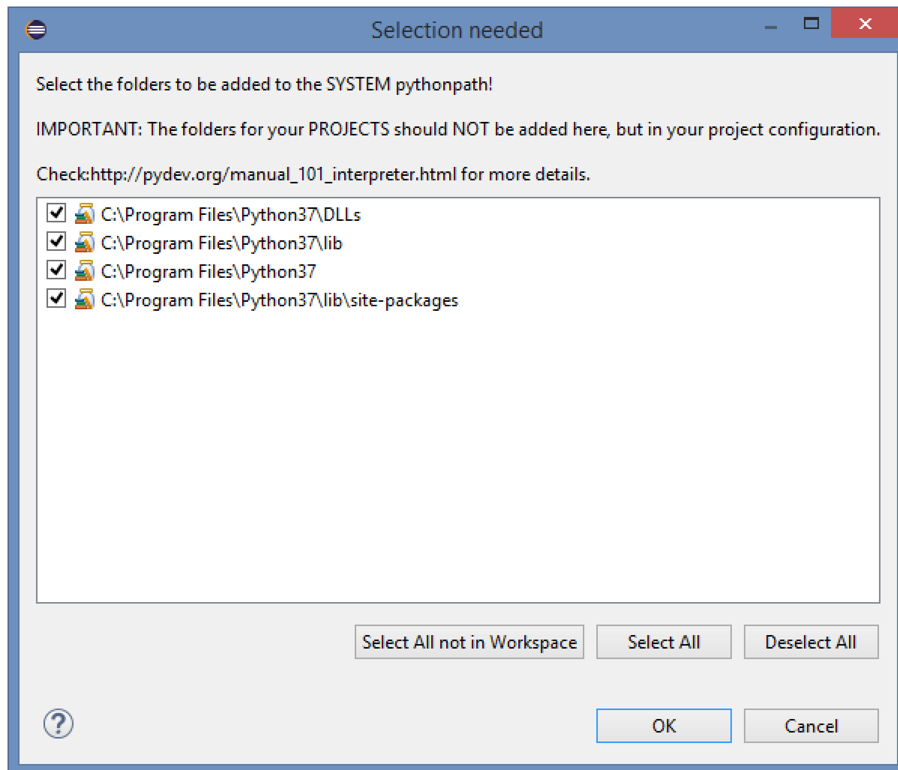
Jeżeli z jakichś przyczyn PyDev nie znalazł zainstalowanego Pythona – możesz mu ręcznie wskazać plik python.exe. W tym celu wybierz polecenie *Browse for python/pypy.exe* (por. Rysunek 5.4.2). Spowoduje to otwarcie okna „ręcznego" wyboru interpretera (Rysunek 5.4.4):



Rysunek 5.4.4 Okno „ręcznego" wyboru interpretera Pythona

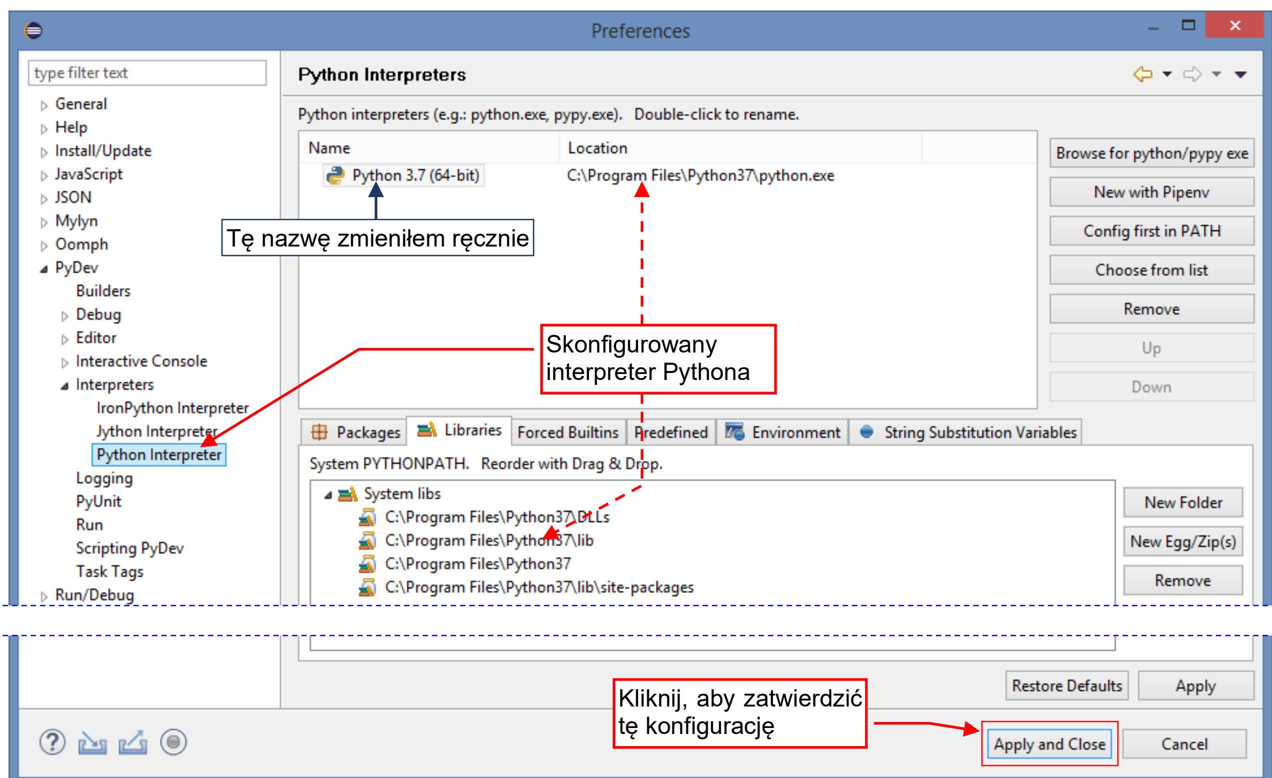
Wpisz w tym oknie nazwę, jaka ma nosić interpreter (to kwestia wyłącznie „estetyczna") oraz wskaż położenie na dysku pliku *python.exe*. (Nie pomyl go przypadkiem z *python**w**.exe*!)

Następnie pojawi się okno z propozycją ścieżek, które mają zostać umieszczone w systemowej zmiennej **PYTHONPATH** (Rysunek 5.4.5). Zaakceptuj je bez zmiany:



Rysunek 5.4.5 Okno wyboru folderów do systemowej ścieżki PYTHONPATH

W rezultacie w oknie *Preferences* pojawi się skonfigurowany interpreter Pythona (Rysunek 5.4.6):

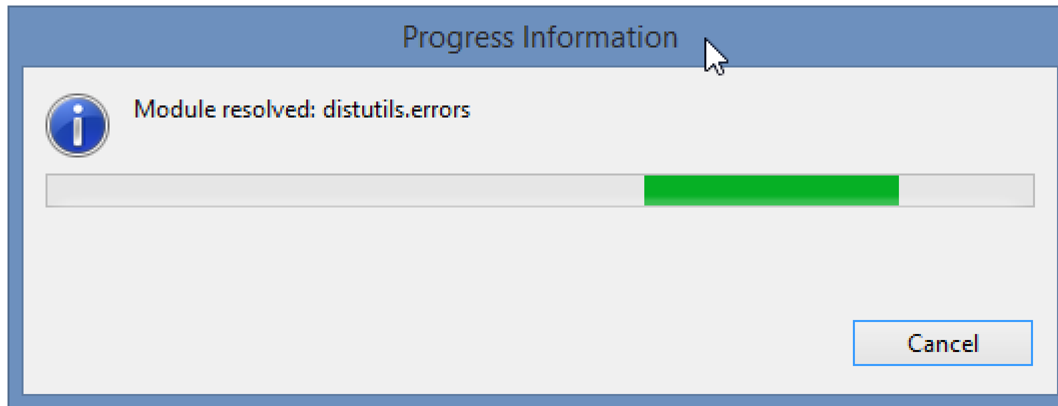


Rysunek 5.4.6 Skonfigurowany interpreter Pythona

Dla większej czytelności zmieniłem nazwę tego interpretera (klikając dwukrotnie w odpowiednie pole na liście) z „python” na „Python 3.7 (64-bit)”.

Copyright Witold Jaworski, 2011-2019.

Po naciśnięciu w tym oknie przycisku ***OK***, Eclipse przeanalizuje wszystkie pliki umieszczone w folderach wymienionych w *PYTHONPATH*. Przygotuje sobie w ten sposób indeksy do autokompletacji kodu i innych pomocy (Rysunek 5.4.7):



Rysunek 5.4.7 Okno przygotowujące pliki znalezione na *PYTHONPATH*

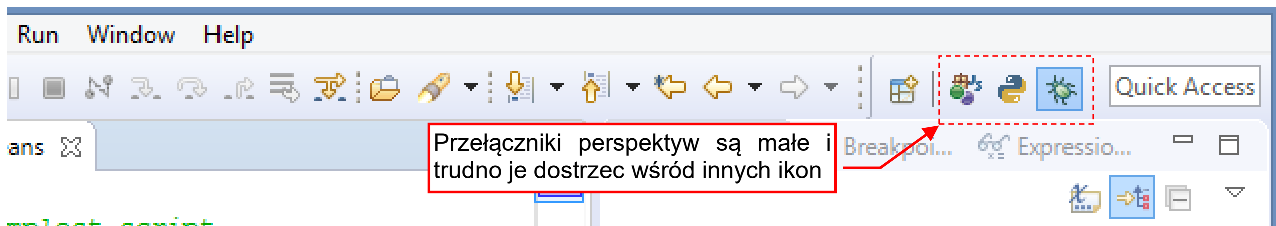
W ten sposób PyDev stał się gotowy do pracy.

- Opisane w tej sekcji ustawienia są zapisywane w przestrzeni roboczej (*workspace*) Eclipse. Oznacza to, że możesz przygotować sobie różne przestrzenie robocze dla różnych wersji Pythona. (Może się to przydać przy testowaniu wtyczek ze starszymi wersjami Blendera)

- Aby debugować/uruchamiać skrypty Pythona w PyDev, musisz jeszcze w konfiguracji każdego projektu zdefiniować polecenie wywołujące ten interpreter – por. str. 134;

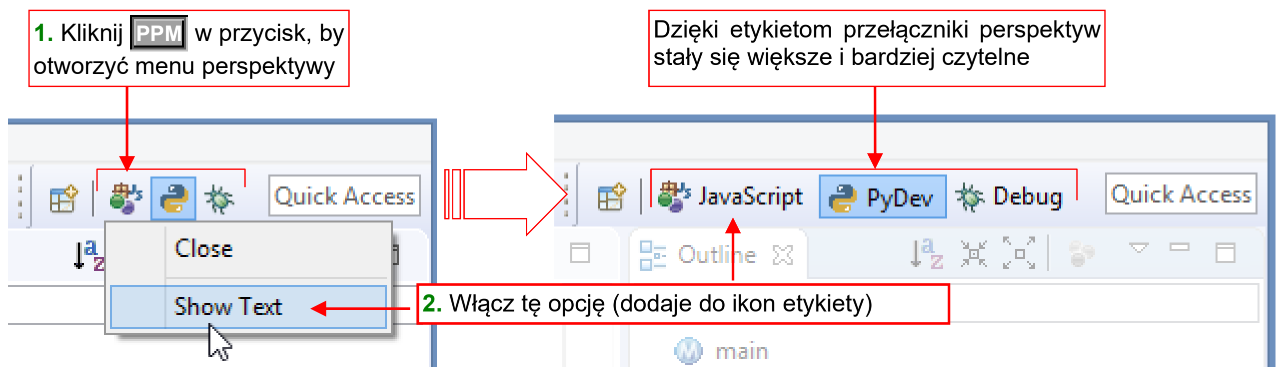
5.5 Zarządzanie perspektywami projektu Eclipse

Podczas pracy nad skryptem Pythona używa się dwóch perspektyw (układów ekranu): *Debug* i *PyDev*. W domyślnym układzie Eclipse przełączniki perspektyw są małe i giną wśród innych ikon paska przybornika (Rysunek 5.5.1):



Rysunek 5.5.1 Przyciski perspektyw (widok domyślny)

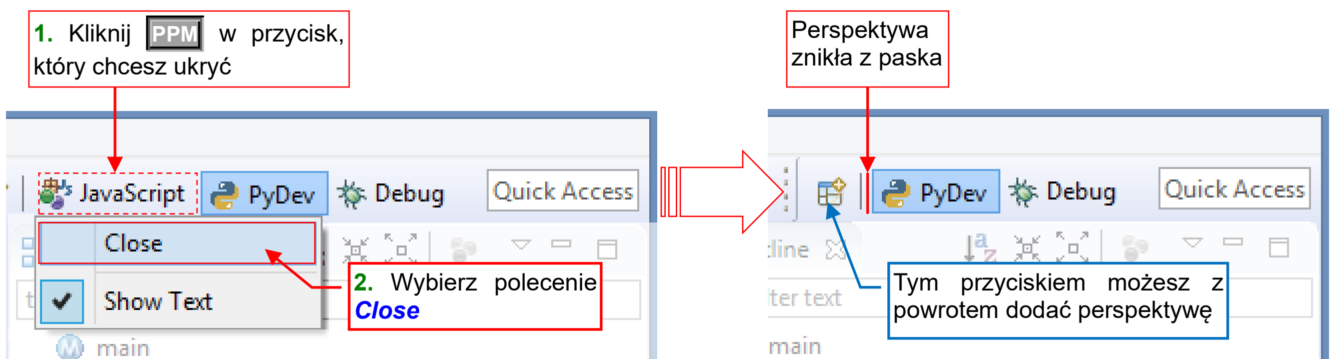
Na szczęście możesz szybko je powiększyć, poprzez dodanie etykiet tekstowych. Stają się wówczas zdecydowanie bardziej widoczne, co ułatwia przełączanie pomiędzy perspektywami (Rysunek 5.5.2):



Rysunek 5.5.2 Przyciski perspektyw: dodanie etykiet

Dzięki etykietom przy tych ikonach łatwiej jest także zorientować się, w jakiej obecnie jesteś perspektywie.

A skoro już robimy z tym paskiem porządek, to usuńmy z niego niepotrzebną (w tym projekcie) perspektywę *JavaScript* (Rysunek 5.5.3):



Rysunek 5.5.3 Usuwanie z paska niepotrzebnej perspektywy

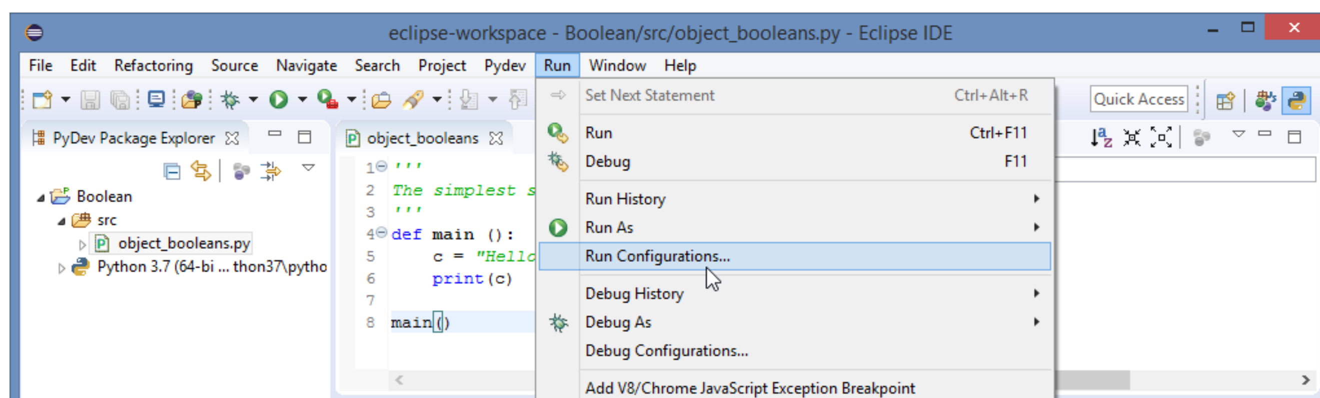
Copyright Witold Jaworski, 2011-2019.

5.6 Konfigurowanie uruchamiania i debugowania skryptu Pytona

W każdym nowym projekcie PyDev trzeba zdefiniować, jak ma być wywoływany interpreter do wykonania / debugowania skryptu Pythona. Należy to zrobić wtedy, gdy w projekcie umieściłeś przynajmniej główny moduł (może być nawet pusty).

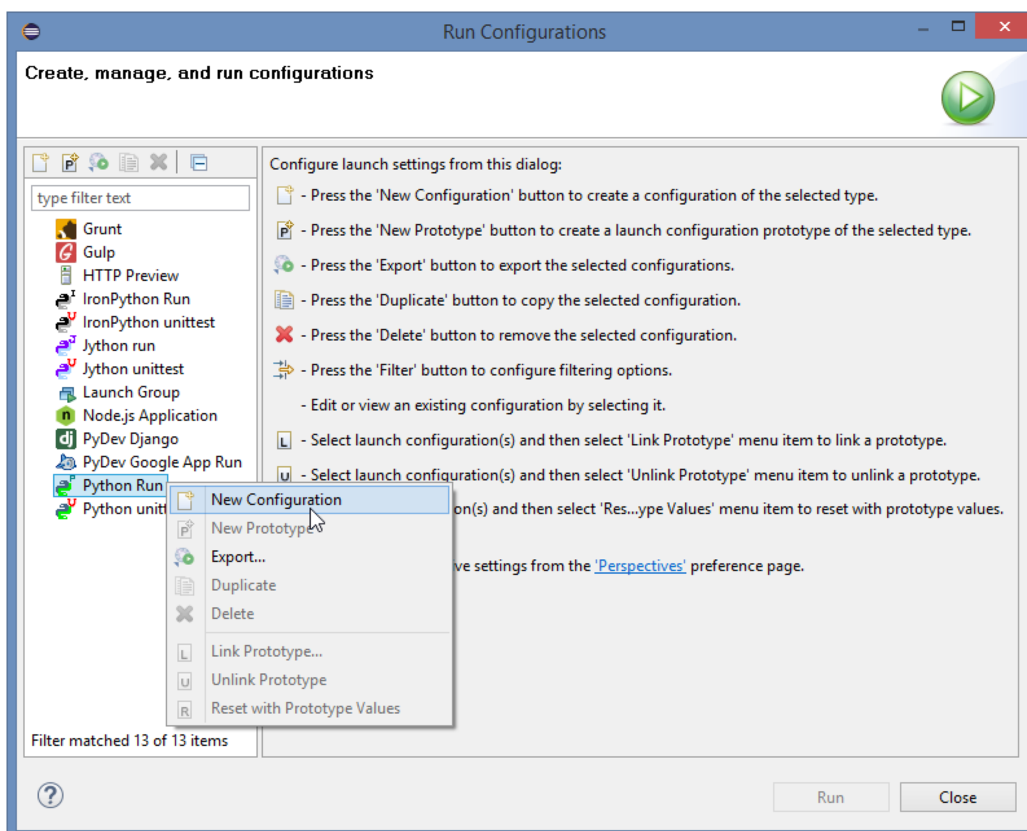
- Do uruchamiania i debugowania skryptów w Blenderze nie będziesz potrzebował żadnych ***Run Configurations***. Są one potrzebne tylko dla „zwykłych” skryptów, wykonywanych przez zewnętrzny interpreter Pythona. W tej książce musiałem przygotować taką konfigurację, gdyż Rozdział 2 pokazuje, jak uruchomić / debugować klasyczny skrypt Pythona.

Zacznijmy od stworzenia definicji uruchamiania. W tym celu z menu ***Run*** wywołaj okno ***Run Configurations…***:



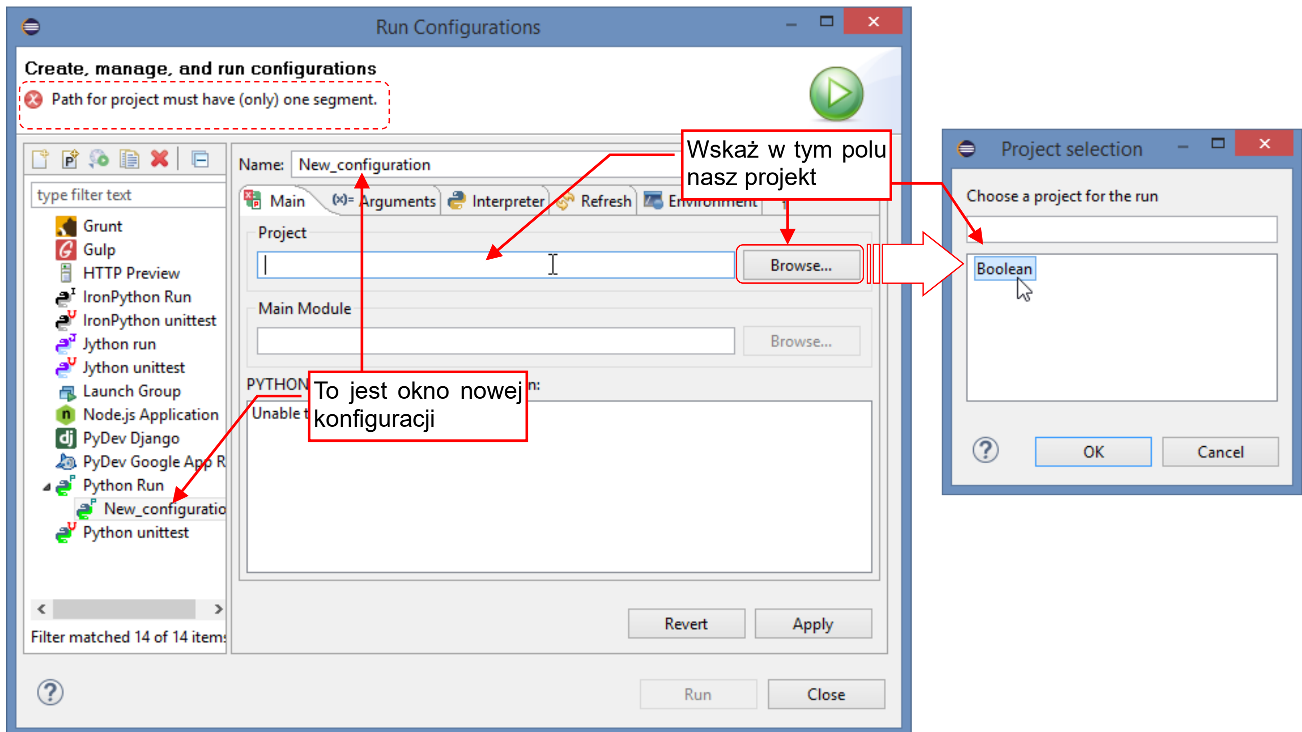
Rysunek 5.6.1 Wywołanie okna z definicjami uruchamiania

W oknie dialogowym ***Run Configurations*** podświetl na liście z lewej strony ***Python Run*** i z menu kontekstowego wywołaj polecenie ***New Configuration*** (Rysunek 5.6.2):



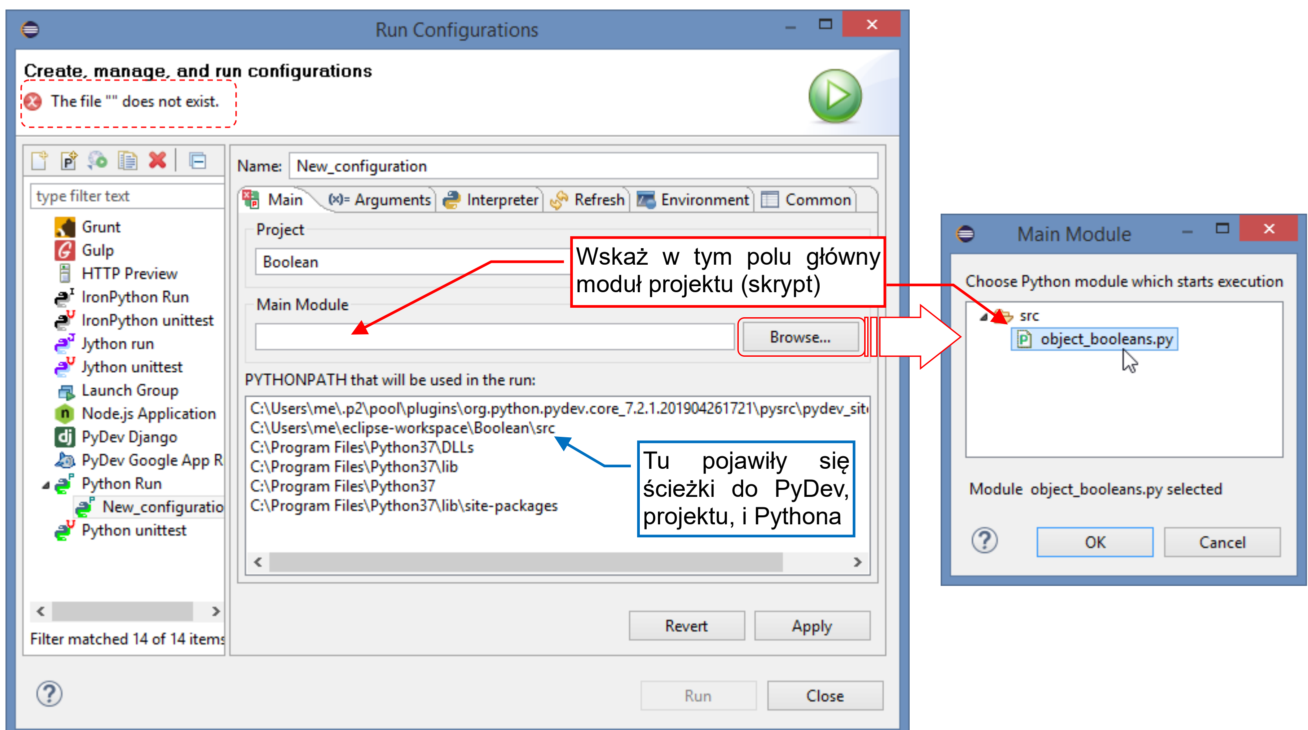
Rysunek 5.6.2 Tworzenie nowej konfiguracji uruchamiania skryptów

Po prawej stronie okna pojawią się pola nowej, pustej konfiguracji. Zacznij od przypisania jej do aktualnego projektu (wypełnij pole ***Project*** – por. Rysunek 5.6.3):



Rysunek 5.6.3 Przypisywanie definicji uruchamiania do projektu

Następnie wskaż moduł główny tego projektu (***Main Module*** – por. Rysunek 5.6.4):

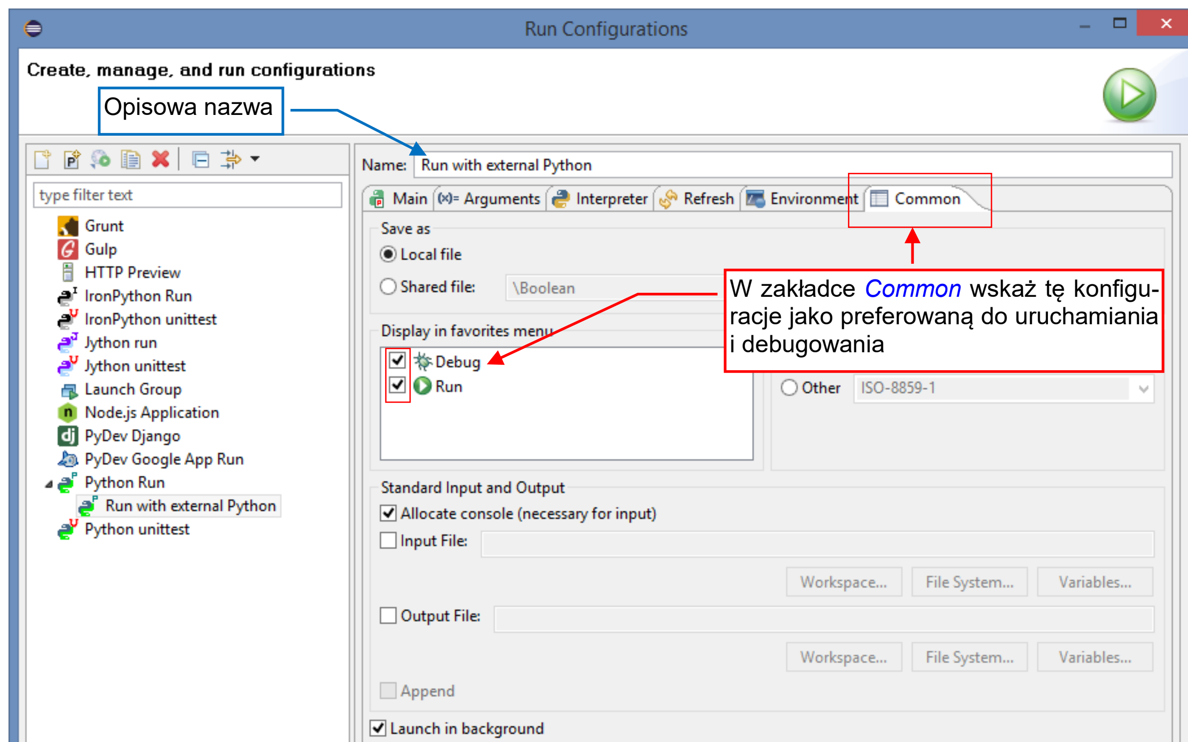


Rysunek 5.6.4 Wskazanie modułu głównego

Wskazany plik może być nawet pusty – ważne, aby w przyszłości pojawiła się w nim główna procedura programu.

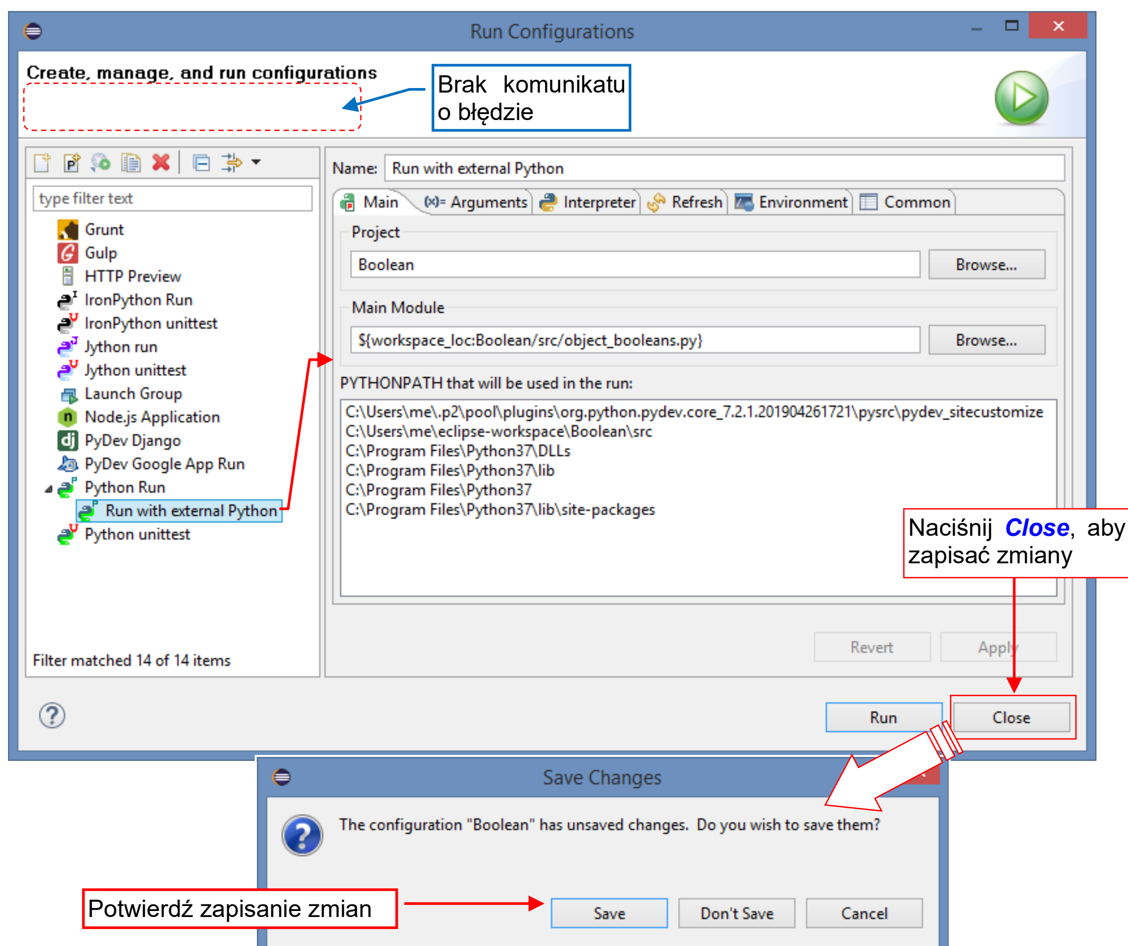
Copyright Witold Jaworski, 2011-2019.

Na koniec zmień nazwę (***Name***) tej konfiguracji na bardziej opisową. Przejdź także do zakładki ***Common*** i wskaż tam te konfiguracje jako domyślną dla uruchamiania i debugowania (Rysunek 5.6.5):



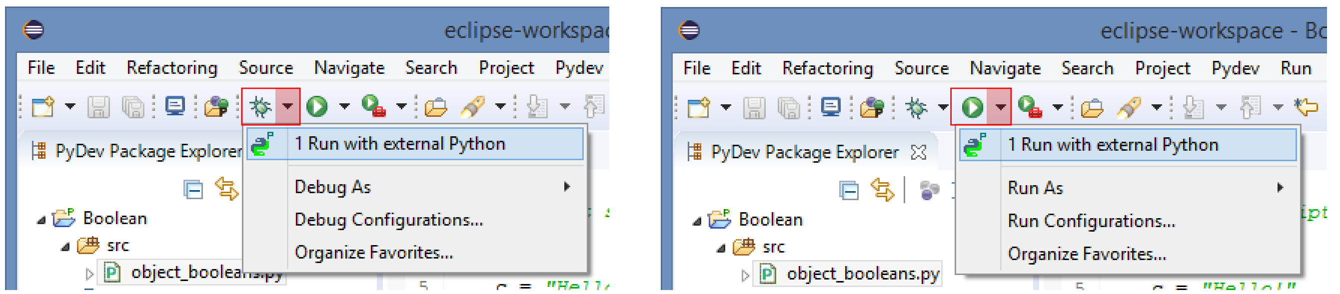
Rysunek 5.6.5 Przypisanie jako domyślnej konfiguracji debugowania / uruchamiania

Zapisz tę konfigurację poleceniem ***Close*** (Rysunek 5.6.6):



Rysunek 5.6.6 Zapisanie definicji uruchamiania skryptów

Dzięki przypisaniu w zakładce *Common*, zapisana konfiguracja jest wyświetlana na pierwszym zarówno w menu *Debug* i *Run* (Rysunek 5.6.7):



Rysunek 5.6.7 Nowa pozycja w menu *Debug* i *Run*

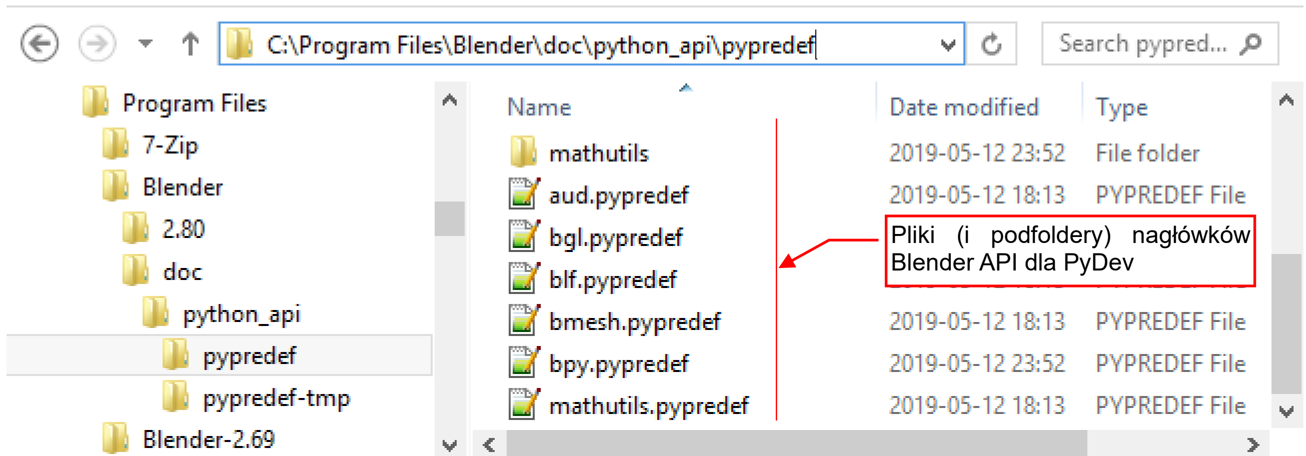
Copyright Witold Jaworski, 2011-2019.

Rozdział 6. Inne

W tym rozdziale umieściłem różne materiały dodatkowe – szczegółowe wyjaśnienia, do których kierowałem we wszystkich wcześniejszych odsyłaczach („szczegóły na str. …", „patrz także str. …"). Dlatego dalsze sekcje to nieco eklektyczna zbieranina różnych cząstkowych tematów. Tym niemniej w każdej z nich znajdziesz rozwiązania wielu ewentualnych problemów, jakie możesz napotkać w trakcie konfigurowania IDE Blendera i Eclipse.

6.1 Aktualizacja nagłówków API Blendera dla PyDev

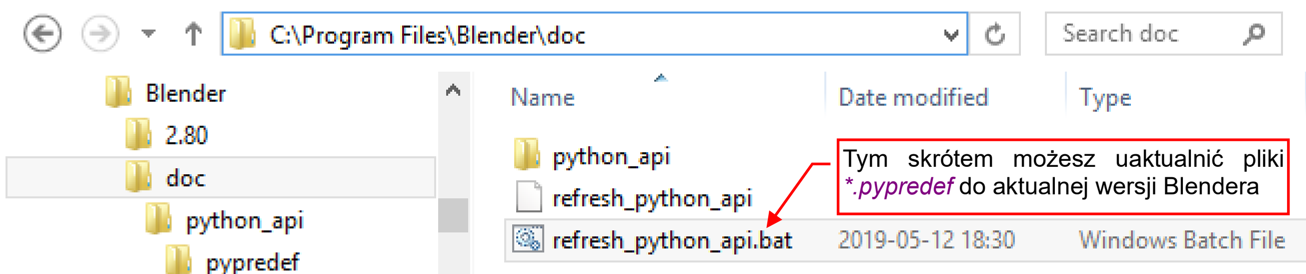
W folderze *doc*, dostarczanym wraz z tą książką (por. str. 39) znajdują się pliki „nagłówków" (**.pypredef*) opisujące API Blendera (Rysunek 6.1.1):



Rysunek 6.1.1 Zawartość folderu *doc\python_api\pypredef*

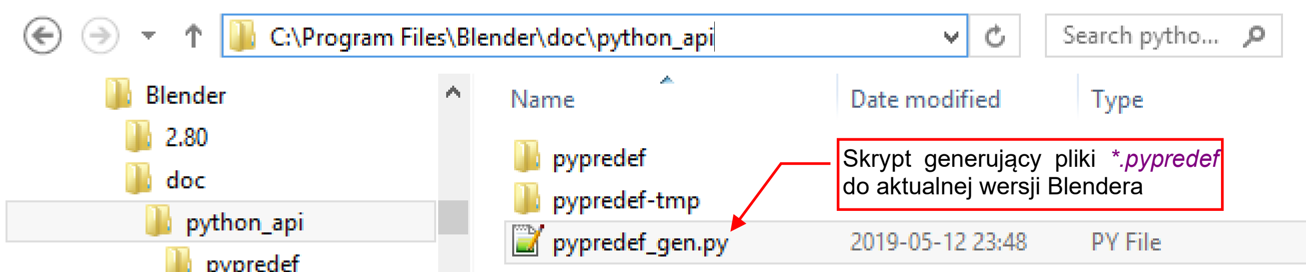
To właśnie ten folder należy wskazać w konfiguracji projektu PyDev jako bibliotekę zewnętrzną (*external library* - por. str. 40). Zawartość tych plików jest wykorzystywana w IDE Eclipse do autokompletacji kodu i wyświetlania opisów funkcji API Blendera.

Z każdą nową wersją Blendera do jego API dochodzą nowe funkcje i klasy. Dlatego w folderze *doc* umieściłem także skrót, który je uaktualni (Rysunek 6.1.2):



Rysunek 6.1.2 Zawartość folderu *doc*

Plik wsadowy *refresh_python_api* uruchamia skrypt *pypredef_gen.py*, umieszczony w folderze *doc\python_api* (Rysunek 6.1.3):



Rysunek 6.1.3 Zawartość folderu *doc\python_api*

Ten skrypt to przerobiona wersja pliku *sphinx_doc_gen.py*, opracowanego przez Campbella Bartona do generowania opisu API Blendera 2.5. Dzięki temu kodowi, opisy wszystkich funkcji i metod w plikach dla Eclipse są takie same, jak w oficjalnej dokumentacji API. Dodatkowo umożliwiło to także umieszczenie listy parametrów każdej procedury oraz ich opisów.

Copyright Witold Jaworski, 2011-2019.

Gdy uruchomisz plik wsadowy *doc\refresh_python_api.bat*, najpierw zobaczysz pojawiające się na ekranie konsoli systemu dużo różnych ostrzeżeń, a potem informację o aktualizacjach (Rysunek 6.1.4):

```
precated since Python 3.5. Use `signature` and the `Signature` object directly
  arg_str = inspect.formatargspec(*arguments) #deprecated since Python 3.3 - in the future I should use a inspect.Signatur
stead

Checking for the *.pypredef files to be updated...
        updating: aud.pypredef
        updating: bgl.pypredef
        updating: blf.pypredef
        updating: bmesh.pypredef
        updating: bmesh\types.pypredef
        updating: bmesh\utils.pypredef
        updating: bpy\app.pypredef
        updating: bpy\path.pypredef
        updating: bpy\props.pypredef
        updating: bpy.pypredef
        updating: bpy\utils.pypredef
        updating: mathutils\geometry.pypredef
        updating: mathutils.pypredef
...done.

Closing Blender:

Writing userprefs: 'C:\Users\me\AppData\Roaming\Blender Foundation\Blender\2.80\config\userpref.blend' ok
Press any key to continue . . .
```

Te komunikaty pojawiały się zawsze — nie ma co się nimi przejmować

Informacja o uaktualnionych plikach nagłówków

Ten komunikat pochodzi od Blendera – wypisuje go zawsze podczas zamykania

Rysunek 6.1.4 Aktualizacja nagłówków Python API Blendera

Najważniejsze są linie na końcu, zaczynające się od słowa „*updating: …*". Informują, jakie pliki **.pypredef* zostały uaktualnione. Przedostatnia linia („*Writing userprefs:..*") pochodzi od zamykanego Blendera, a „*Press any key to continue…*" od standardowego polecenia *pause* na końcu pliku wsadowego.

Jeżeli jednak zakończenie skryptu będzie wyglądało jak poniżej (Rysunek 6.1.5) – to oznacza błąd:

```
 current value '0' matches no enum in 'EnumProperty', 'type', 'default'
WARN (bpy.rna): c:\b\win64_cmake_vs2017\win64_cmake_vs2017\blender.git\source\blender\python\intern\bpy_rna.c:1450 pyrna_e
 current value '0' matches no enum in 'EnumProperty', 'type', 'default'
WARN (bpy.rna): c:\b\win64_cmake_vs2017\win64_cmake_vs2017\blender.git\source\blender\python\intern\bpy_rna.c:1450 pyrna_e
 current value '0' matches no enum in 'EnumProperty', 'pose', 'default'
WARN (bpy.rna): c:\b\win64_cmake_vs2017\win64_cmake_vs2017\blender.git\source\blender\python\intern\bpy_rna.c:1450 pyrna_e
 current value '0' matches no enum in 'EnumProperty', 'pose', 'default'
WARN (bpy.rna): c:\b\win64_cmake_vs2017\win64_cmake_vs2017\blender.git\source\blender\python\intern\bpy_rna.c:1450 pyrna_e
 current value '0' matches no enum in 'EnumProperty', 'pose', 'default'
WARN (bpy.rna): c:\b\win64_cmake_vs2017\win64_cmake_vs2017\blender.git\source\blender\python\intern\bpy_rna.c:1450 pyrna_e
 current value '0' matches no enum in 'EnumProperty', 'mask', 'default'
WARN (bpy.rna): c:\b\win64_cmake_vs2017\win64_cmake_vs2017\blender.git\source\blender\python\intern\bpy_rna.c:1450 pyrna_e
 current value '0' matches no enum in 'EnumProperty', 'clip', 'default'
WARN (bpy.rna): c:\b\win64_cmake_vs2017\win64_cmake_vs2017\blender.git\source\blender\python\intern\bpy_rna.c:1450 pyrna_e
 current value '0' matches no enum in 'EnumProperty', 'scene', 'default'
WARN (bpy.rna): c:\b\win64_cmake_vs2017\win64_cmake_vs2017\blender.git\source\blender\python\intern\bpy_rna.c:1450 pyrna_e
 current value '0' matches no enum in 'EnumProperty', 'name
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "C:\Program Files\Blender\doc\python_api/pypredef_gen.py", line 1245, in <module>
    main() #just run it! Unconditional call makes it easier to debug Blender script in Eclipse,
  File "C:\Program Files\Blender\doc\python_api/pypredef_gen.py", line 1161, in main
    rna2predef(path_in_tmp) #create the up-to date file versions in the pypredef-tmp
  File "C:\Program Files\Blender\doc\python_api/pypredef_gen.py", line 1062, in rna2predef
    bpy2predef(BASEPATH,"Blender API main module")
  File "C:\Program Files\Blender\doc\python_api/pypredef_gen.py", line 1011, in bpy2predef
    file = open(filepath, encoding='utf-8', mode="w")
PermissionError: [Errno 13] Permission denied: 'C:\\Program Files\\Blender\\doc\\python_api\\pypredef-tmp\\bpy.pypredef'
Writing userprefs: 'C:\Users\me\AppData\Roaming\Blender Foundation\Blender\2.80\config\userpref.blend' ok

Blender quit
Press any key to continue . . .
```

Takie zakończenie oznacza błąd w skrypcie!

Jeżeli komunikat dotyczy problemów z plikiem lub katalogiem – to jest to kwestia brakujących uprawnień

Rysunek 6.1.5 Błąd skryptu aktualizującego

Jeżeli komunikat o błędzie (znajdziesz go w miejscu pokazanym na ilustracji) mówi o problemie uprawnieniami lub z zapisem do katalogu, to najprawdopodobniej musisz rozszerzyć uprawnienia do folderu *doc*.

- W systemie Windows katalog, który zawiera folder Blendera (*C:\Program Files*) jest traktowany w sposób specyficzny. Domyślnie żaden program wywoływany w Windows nie może zapisać plików w jego podkatalogach. Stąd, jeżeli umieściłeś katalog *doc* w folderze *C:\Program Files\Blender* (tak jak to pokazałem na str. 39), musisz zmienić uprawnienia do tego folderu. Dodaj grupie *Użytkownicy* prawo do zmiany/zapisu w tym folderze.

Jeżeli z jakichś przyczyn nie masz uprawnień Administratora do tego komputera – możesz umieścić folder *doc* w innym miejscu, do którego posiadasz uprawnienia do odczytu/zapisu.

- Folder *doc* możesz umieścić w dowolnym innym miejscu, np. w folderze *Dokumenty*. W takim przypadku musisz jednak poprawić w pliku *refresh_python_api.bat* wywołanie Blendera, wpisując zamiast względnej ścieżki„*..*” pełną ścieżkę dostępu do *blender.exe*, np.:

```
"C:\Program Files\Blender\blender.exe" -b -P python_api/pypredef_gen.py
```

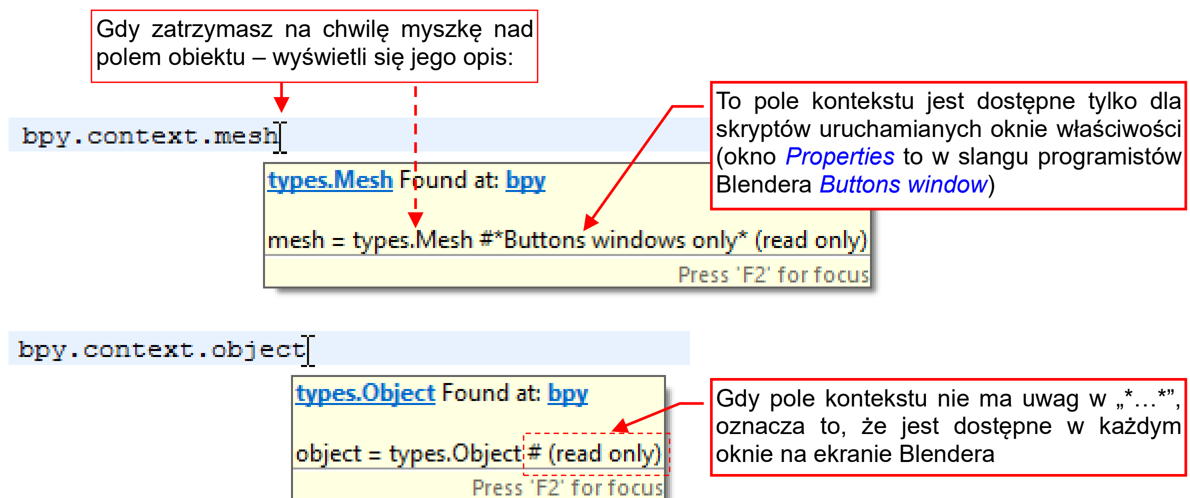
Na koniec parę uwag o szczegółach:

1. Moduł ***bgl*** został przeniesiony tylko częściowo (wyłącznie symbole stałych i nazwy funkcji). Wynika to z braku informacji do przetworzenia przez skrypt: twórcy Blendera nie dodali w ***bgl*** tak rozbudowanych standardowych opisów Pythona jak w innych modułach. Zresztą w Blenderze 2.8 sugeruje się przejście z metod ze „staromodnego” modułu ***bgl*** na bardziej nowoczesny ***gpu***, jako bardziej zgodne z aktualnie używana przez Blender wersją OpenGL (i bardziej wydajne).

2. Operatory modułu ***bmesh*** (metody „pseudoklasy” ***bmesh.ops***) są udostępnione bez opisów, gdyż nie mogłem ich znaleźć w żaden znany mi sposób. (Ich właściwości ***__doc__*** zawierają tylko deklaracje metod, bez żadnych objaśnień).

3. Plik *bpy.pypredef* zawiera także uproszczone deklaracje menu i paneli GUI Blendera. Czasami trzeba się do nich odwołać w kodzie skryptu, i gdyby tych deklaracji nie było, wówczas PyDev zaznaczyłby taką nazwę jako błędną. (To nie przeszkadza w wykonaniu skryptu, ale lepiej przyjąć zasadę, że przed uruchomieniem w kodzie nie może być żadnych błędów sygnalizowanych przez IDE). Te menu i panele to wyspecjalizowane klasy pochodne ***bpy.types.Menu*** (klasy zawierające w nazwie „_MT_”) i ***bpy.types.Panel*** (klasy zawierające w nazwie „_PT_”). Ich nazwy zaczynają się od nazwy okna/obszaru, którego dotyczą, napisanych dużymi literami. Na przykład: klasa o nazwie ***bpy.types.VIEW3D_MT_object*** reprezentuje menu *Object* z okna *View 3D*. Takich elementów UI jest całe mnóstwo i aby nie utrudniać przeglądania podstawowej zawartości ***bpy.types***, umieściłem tylko nagłówki tych klas, na samym końcu pliku.

4. Prawie wszystkie pola obiektu ***bpy.context*** (klasy ***bpy.types.Context***) są dopisane do pliku *bpy.predef* na podstawie stałego tekstu, który przygotowałem. To bardzo specyficzny obiekt: pola, które eksponuje skryptom, mogą się pojawiać i znikać w zależności od typu (*3D View*, *Python Console*, itp.) aktualnego obszaru ekranu. W deklaracji wpisałem wszystkie możliwe pola, więc zwróć uwagę na komentarze, wyświetlane przez autokompletację po umieszczeniu nazwy pola w kodzie (Rysunek 6.1.6):



Rysunek 6.1.6 Informacje o zmiennych/polach klasy *Context*

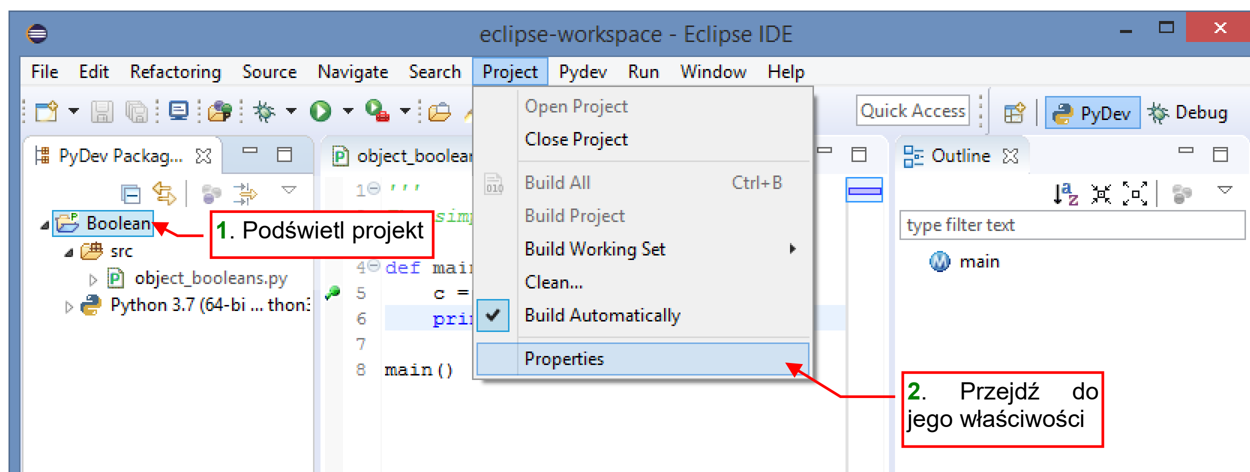
- Po każdej aktualizacji plików *pypredef*, uaktualnij także wewnętrzną informację PyDev – por. str. 143.

Copyright Witold Jaworski, 2011-2019.

6.2 Uruchomienie w projekcie PyDev autokompletacji kodu Blender API

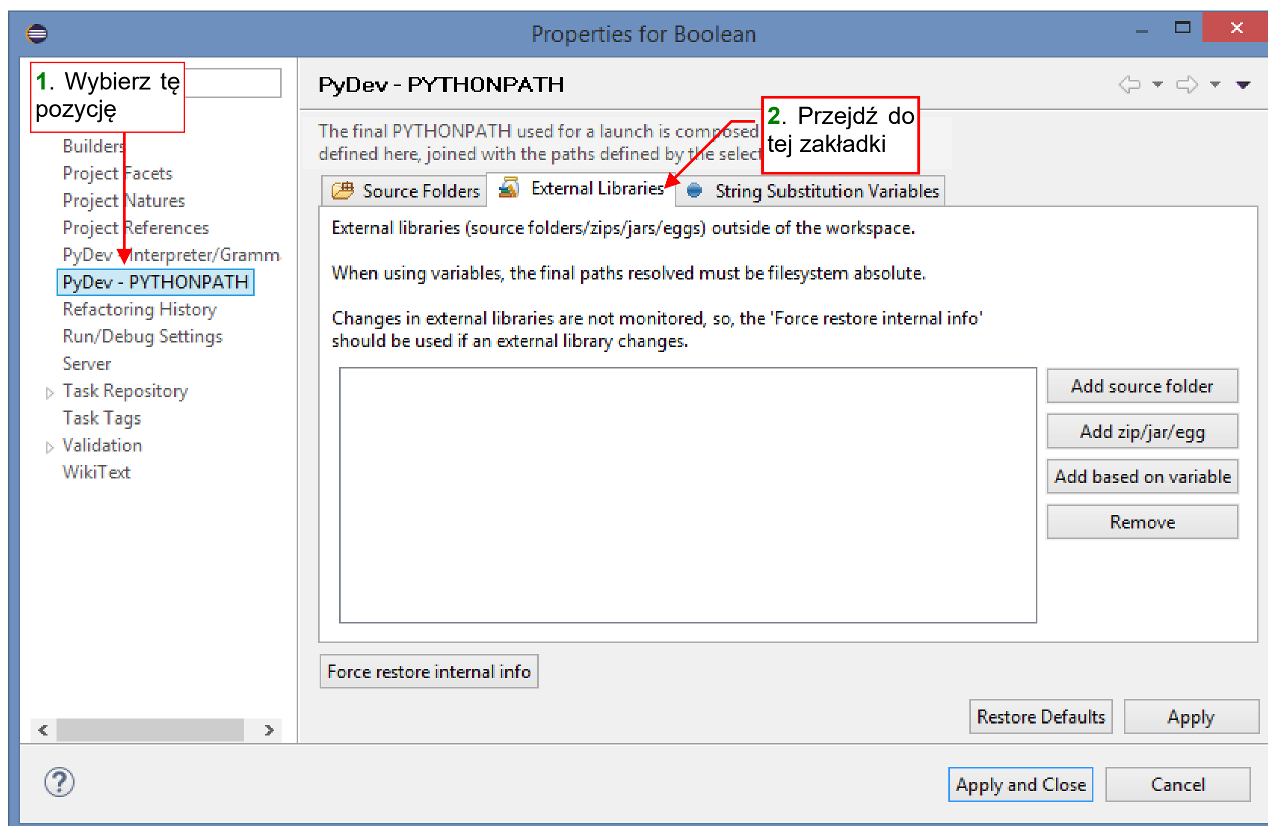
Gdy zaczynasz w PyDev nowy projekt wtyczki Blendera, warto w jego konfiguracji dodać do zmiennej **PYTHONPATH** ścieżkę *doc\python_api\pypredef*. W tym folderze są umieszczone pliki „nagłówków” z deklaracjami wszystkich klas, funkcji i stałych Blender API (por. str. 139).

Aby to zrobić, przejdź do właściwości projektu (***Project→Properties***, Rysunek 6.2.1):



Rysunek 6.2.1 Przejście do właściwości projektu

W oknie, które się pojawi, wybierz sekcję ***PyDev – PYTHONPATH***, a w niej — zakładkę ***External Libraries*** (Rysunek 6.2.2):



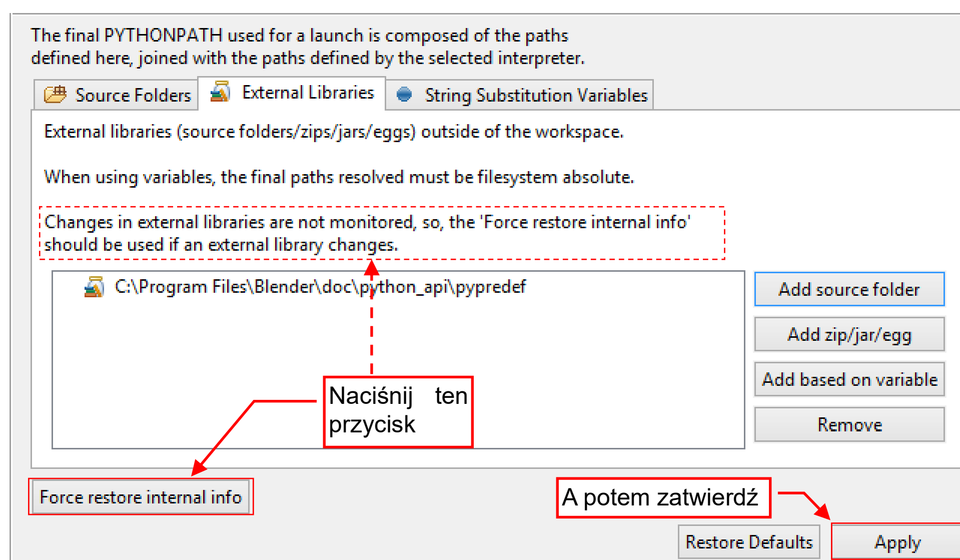
Rysunek 6.2.2 Przejście do edycji *PYTHONPATH*

Początkowo projekt nie ma żadnych dodatkowych bibliotek (lista w tej zakładce jest pusta). Naciśnij przycisk ***Add source folder***, aby dodać nową pozycję, i wskaż folder *doc\python_api\pypredef* (Rysunek 6.2.3):

The final PYTHONPATH used for a launch is composed of the paths
defined here, joined with the paths defined by the selected interpreter.
Source Folders
External Libraries
String Substitution Variables
External libraries (source folders/zips/jars/eggs) outside of the workspace.
When using variables, the final paths resolved must be filesystem absolute.
Changes in external libraries are not monitored, so, the 'Force restore internal info'
should be used if an external library changes.
1. Naciśnij ten przycisk, aby dodać bibliotekę z nagłówkami Blender API
Na razie lista zewnętrznych bibliotek jest pusta
Add source folder
Add zip/jar/egg
Add based on variable
Remove
Select Folder
C:\Program Files\Blender\doc\python_api
Search python_api
Organize
New folder
Program Files
7-Zip
Blender
2.80
doc
python_api
pypredef
pypredef-tmp
Blender-2.69
Name
Date modified
Type
pypredef 2019-05-12 23:52 File folder
pypredef-tmp 2019-05-12 23:52 File folder
2. Wskaż podkatalog *doc\python_api\pypredef* (por. str. 139)
Folder: pypredef
Select Folder
Cancel

Rysunek 6.2.3 Dodanie nagłówków API jako „biblioteki zewnętrznej"

Gdy folder z deklaracjami Blender API jest już na liście, naciśnij przycisk ***Force restore internal info***. Jak wynika z opisu w oknie, tak należy robić tu zawsze, po wprowadzeniu zmiany (Rysunek 6.2.4):

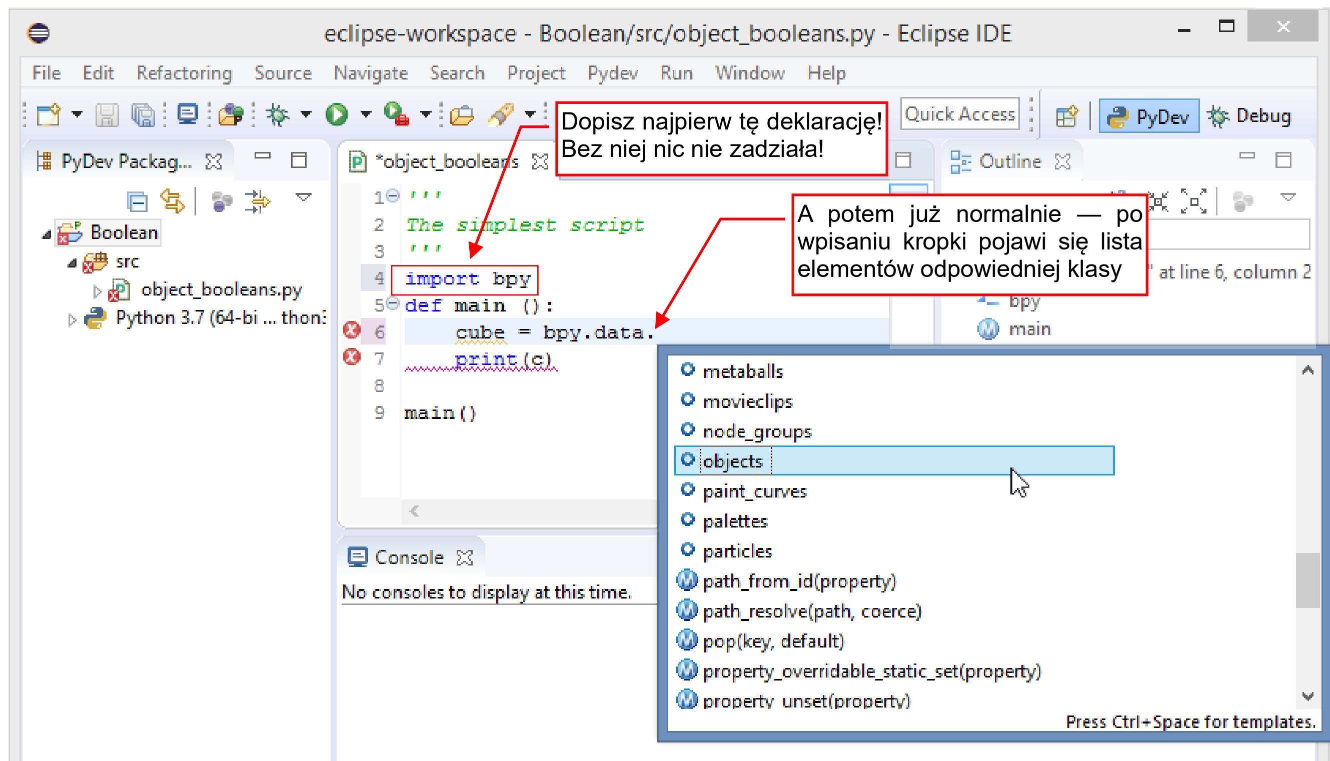


Rysunek 6.2.4 Wymuszenie odświeżenia wewnętrznych danych projektu

Potem oczywiście zatwierdź dokonane zmiany przyciskiem ***Apply***.

Copyright Witold Jaworski, 2011-2019.

Po zatwierdzeniu zmian w konfiguracji projektu, dodaj na początek skryptu deklarację „***import bpy***". Dzięki niej PyDev będzie wiedział, że ma wyszukać deklaracje z modułu ***bpy*** (Blender API). Potem wystarczy, że w czasie pisania kodu postawisz kropkę po nazwie obiektu. Edytor wyświetli wówczas listę jego metod i właściwości (Rysunek 6.2.5):

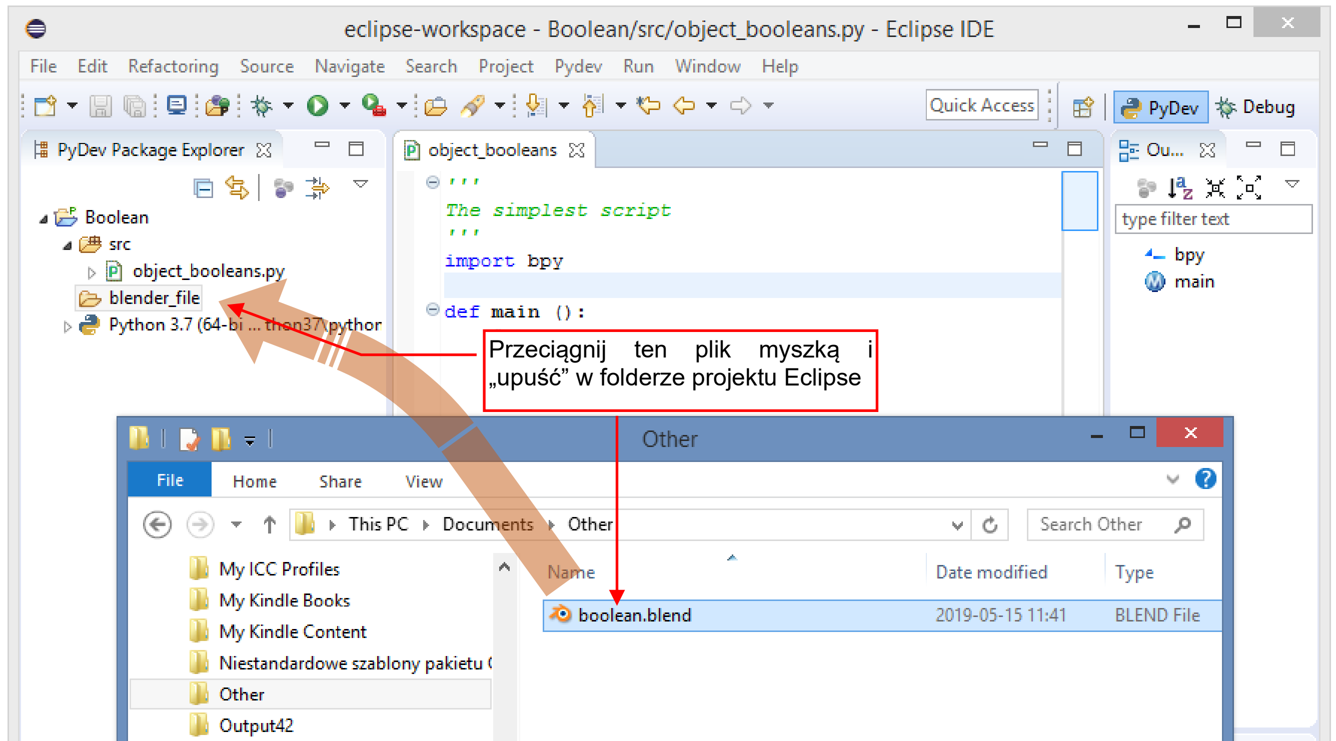


Rysunek 6.2.5 Autokompletacja kodu — po wpisaniu kropki (lub Ctrl-Space)

Więcej o posługiwaniu się funkcjami autokompletacji znajdziesz na str. 41 i następnych.

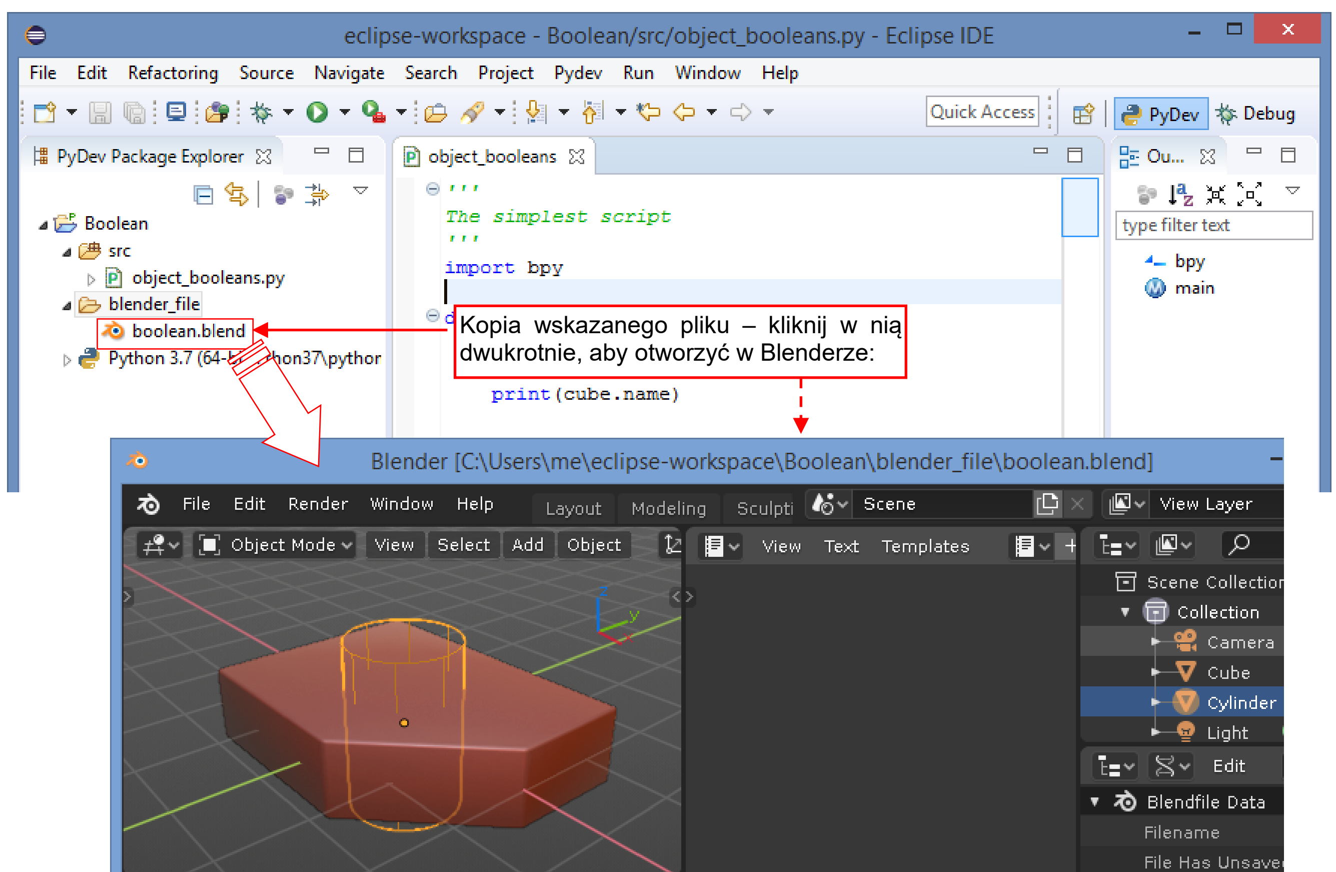
6.3 Importowanie/linkowanie plików do projektu PyDev

Do projektu PyDev można dołączyć jakieś istniejące pliki. Wystarczy je przeciągnąć z okna Eksploratora Plików do odpowiedniego folderu w projekcie Eclipse (Rysunek 6.3.1):



Rysunek 6.3.1 Import istniejącego elementu do folderu projektu

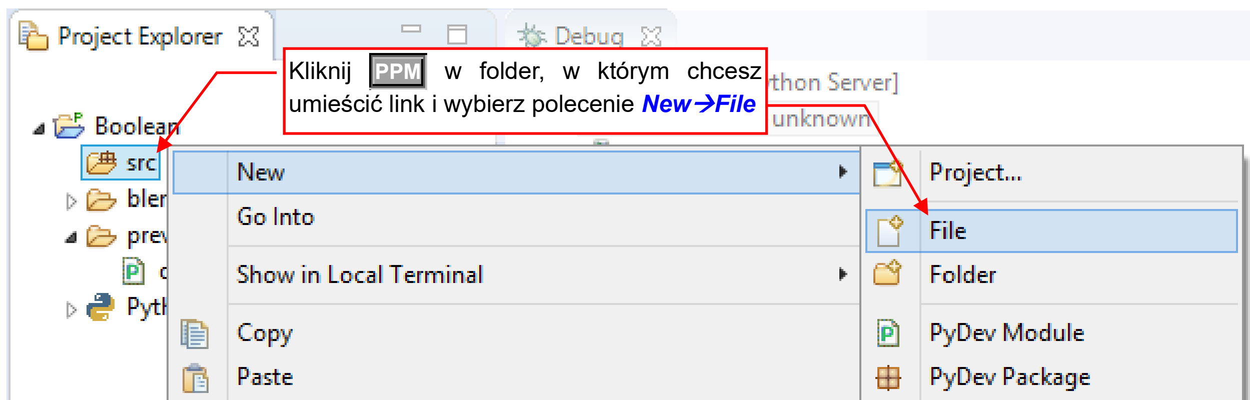
W rezultacie w folderze Eclipse powstanie kopia wskazanego pliku (Rysunek 6.3.2):



Rysunek 6.3.2 Testowe środowisko Blendera, umieszczone w folderze projektu

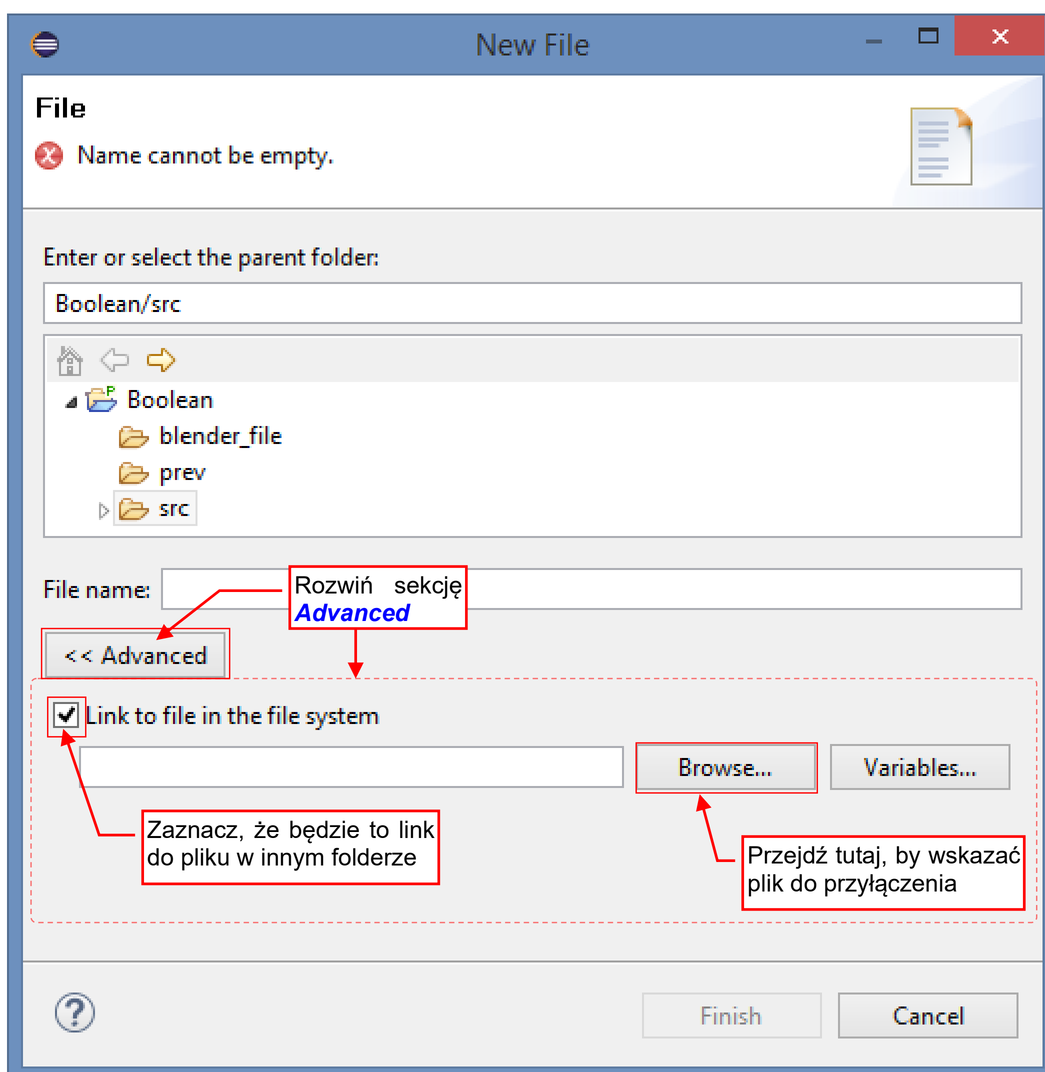
Copyright Witold Jaworski, 2011-2019.

Oprócz kopiowania, Eclipse umożliwia także umieszczenie w folderach projektu skrótów (linków) do plików położonych w innych miejscach na dysku. W projekcie PyDev trzeba to zrobić „dłuższą drogą” [1]: poprzez polecenie ***New→File*** z menu kontekstowego (Rysunek 6.3.3):



Rysunek 6.3.3 Wywołanie polecenia utworzenia nowego pliku (z menu kontekstowego projektu)

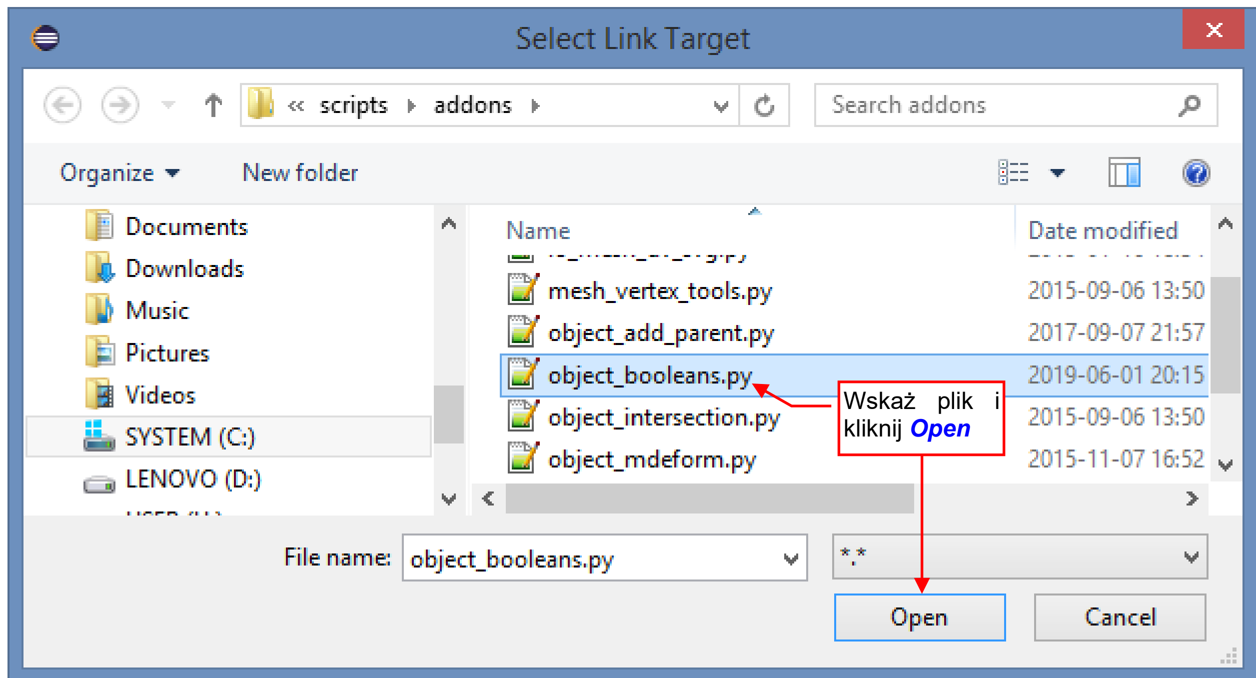
Na pierwszym ekranie kreatora włącz opcję ***Advanced*** i zaznacz, że będzie to plik podłączony (Rysunek 6.3.4):



Rysunek 6.3.4 Wybór rodzaju źródła

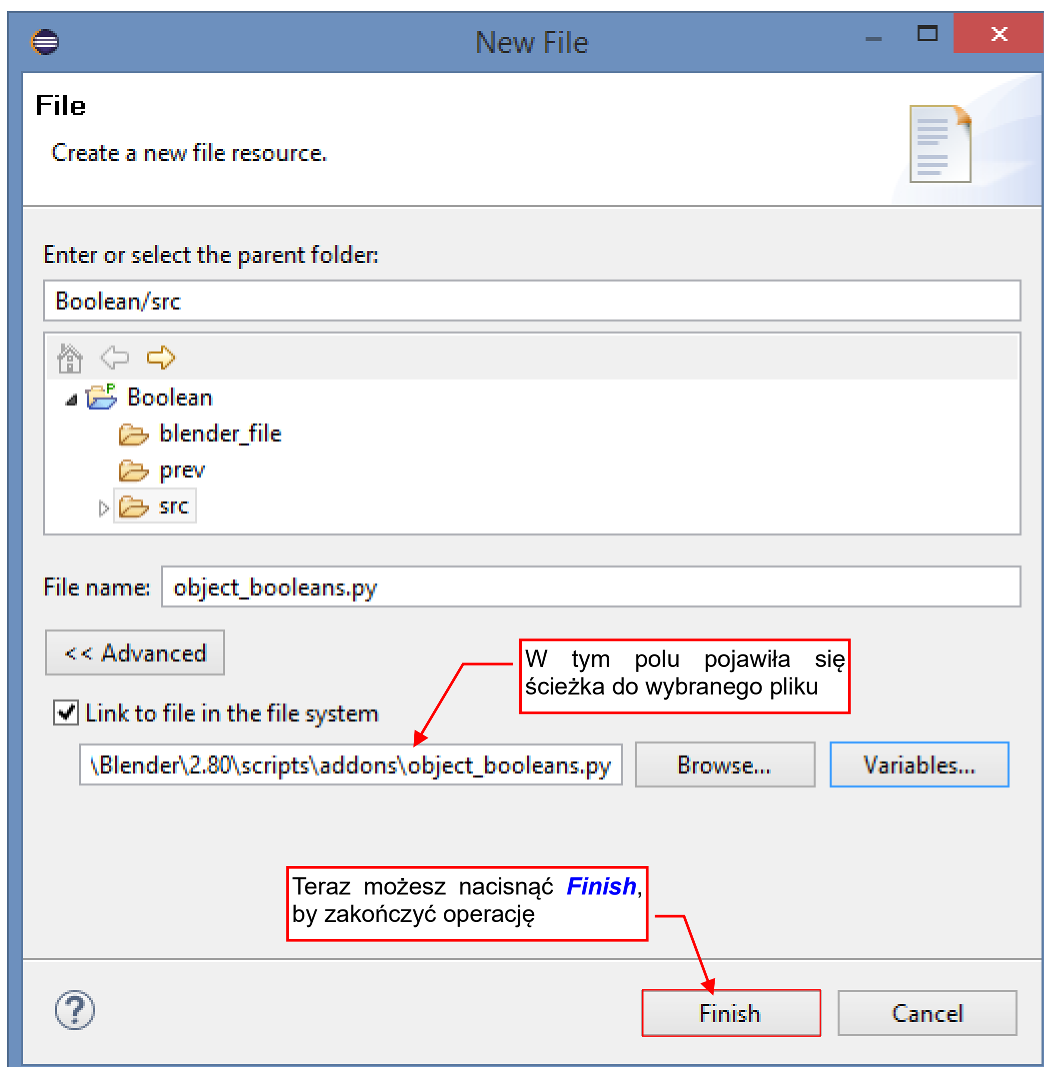
[1] Z niewiadomych przyczyn PyDev ignoruje domyślne ustawienie Eclipse (z *Window→Preferences:Workspace\Linked Resources*). Według tych ustawień po upuszczeniu nowego pliku w folder projektu program powinien zapytać, czy plik ma być skopiowany czy podłączony.

W kolejnym oknie kreatora należy najpierw wybrać folder, w którym znajduje się plik/pliki (Rysunek 6.3.5):



Rysunek 6.3.5 Wskazywanie pliku do przyłączenia do projektu

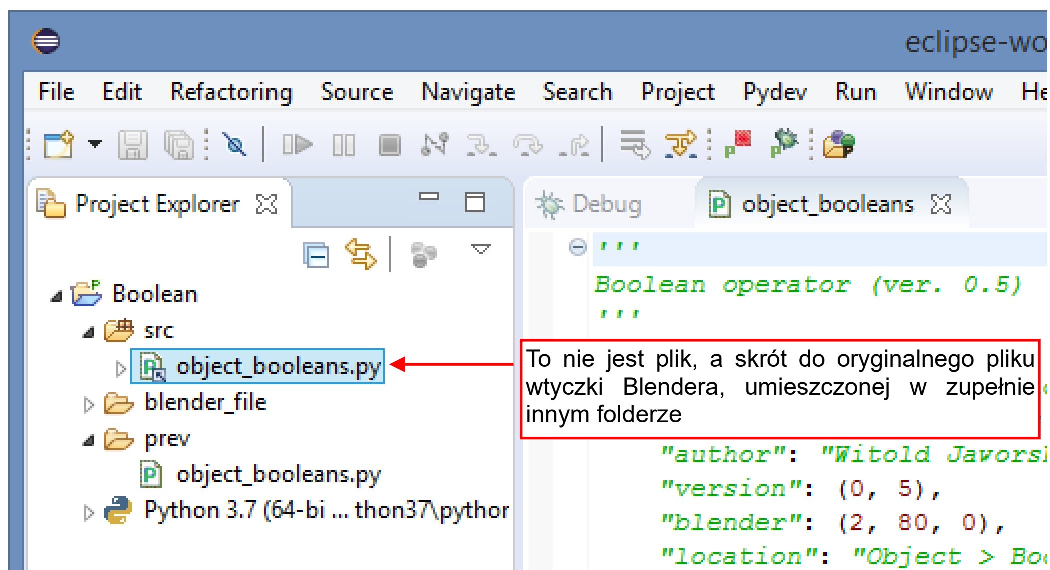
Po kliknięciu przycisku ***Open*** powrócisz do okna kreatora, w którym kliknij ***Finish*** (Rysunek 6.3.6):



Rysunek 6.3.6 Wypełniony ekran kreatora

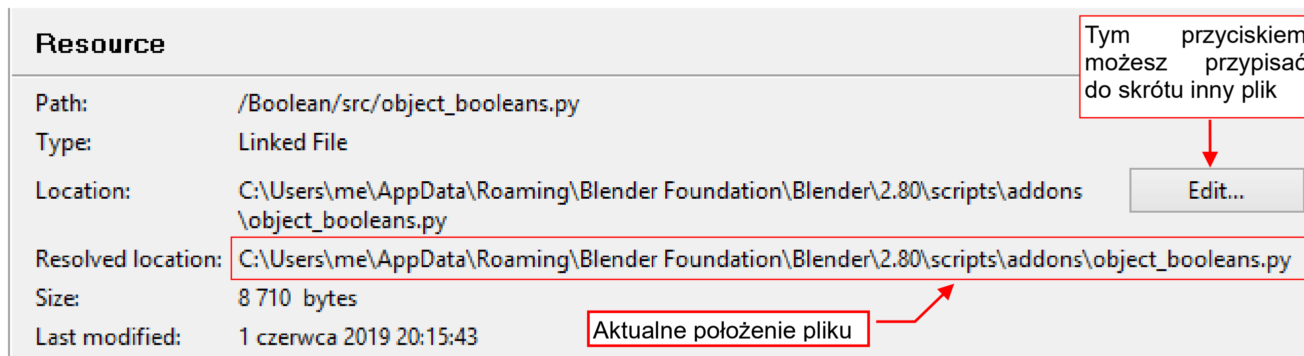
Copyright Witold Jaworski, 2011-2019.

To spowoduje, że PyDev nie będzie tworzył lokalnej kopii pliku w swoim folderze. Zamiast tego umieści tam tylko referencję do oryginalnego skryptu. W ten sposób możesz np. wygodnie zmieniać kod wtyczki Blendera, mimo że ta nadal jest umieszczonej w jego katalogu (Rysunek 6.3.7):



Rysunek 6.3.7 Skrót do pliku, dodany do projektu

Jak widzisz, ikony skrótów do plików są oznaczane w Eclipse dodatkową strzałką w prawym dolnym rogu. Gdy zajrzysz do właściwości takiego skrótu, możesz z nich odczytać lub zmienić położenie „oryginału” (Rysunek 6.3.8):

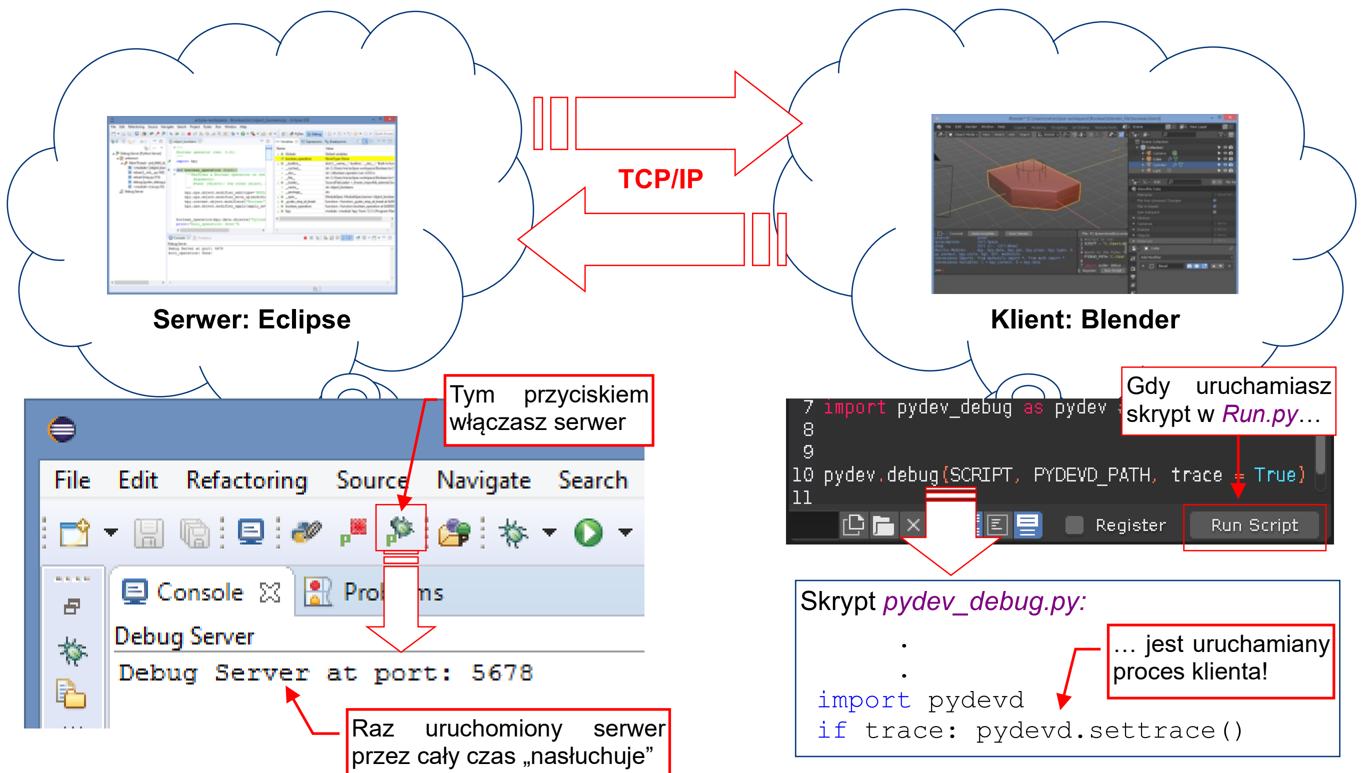


Rysunek 6.3.8 Przykładowa informacja o powiązanym pliku. (To fragment okna jego właściwości — *Properties*)

6.4 Debugowanie skryptu w Blenderze — szczegóły konfiguracji i obsługi

Blender wykonuje skrypty Pythona posługując się swoim własnym interpreterem. Dlatego można je debugować za pomocą wbudowanego, standardowego debugera. Niestety, to narzędzie działa wyłącznie w trybie „konwersacyjnym", w konsoli. Nie jest przez to zbyt wygodne w użyciu.

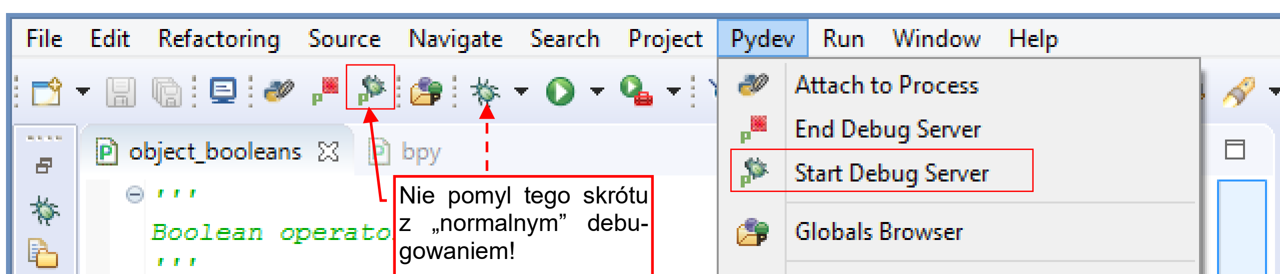
Szansę na śledzenie skryptu w jakimś „okienkowym" debugerze, takim jakim dysponuje IDE Eclipse, daje tylko tzw. debuger zdalny. To rozwiązanie wymyślone oryginalnie do śledzenia programów wykonywanych na innym komputerze (Rysunek 6.4.1):



Rysunek 6.4.1 Śledzenie wykonywania skryptu w Blenderze: działanie zdalnego debugera

IDE (takie jak Eclipse) uruchamia proces serwera, który zaczyna „nasłuchiwać" żądań od ewentualnych debugowanych skryptów. Informacje o tym będzie wysyłał klient zdalnego debuggera, włączony w kodzie śledzonego skryptu. W naszym przypadku kod tego klienta znajduje się w pakiecie (*package*) o nazwie ***pydevd***. Ten kod jest importowany i inicjowany w pomocniczym module *pydev_debug.py* (por. str. 158), który jest wykorzystany z kolei w pliku *Run.py*. (To plik, który wywołuje nasz skrypt — por. str. 53). Komunikacja pomiędzy klientem i serwerem zdalnego debuggera odbywa się poprzez sieć. Już dawno temu ktoś spostrzegł, że nic nie stoi na przeszkodzie, by te dwa procesy działały na tej samej maszynie. Wymieniają wtedy między sobą dane korzystając z lokalnej karty sieciowej komputera. Koncepcyjnie odpowiada to sytuacji, jak gdyby dwóch ludzi siedzących w tym samym pokoju rozmawiało przez telefon. Ale programy „są głupie" i nie narzekają, że muszą się porozumiewać tak okrężną drogą. A całość działa poprawnie, i to się tylko liczy.

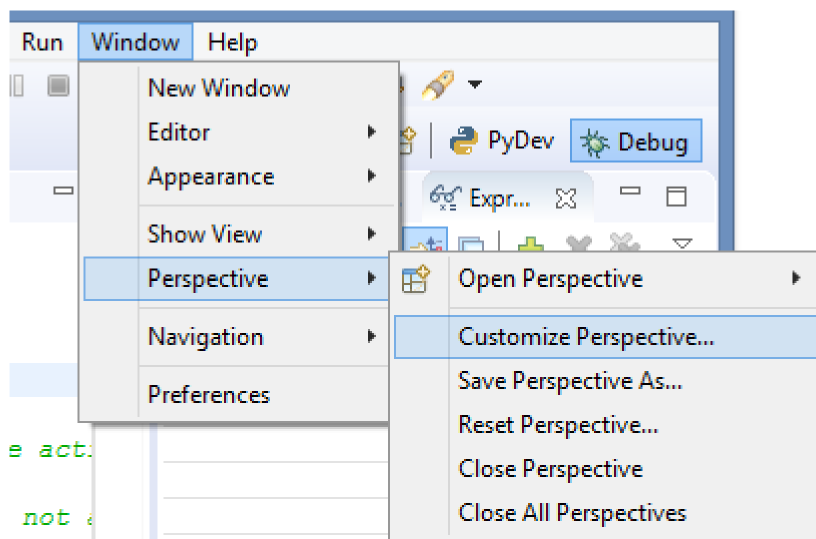
Proces serwera w PyDev możesz także uruchomić poleceniem ***PyDev→Start Debug Server*** (Rysunek 6.4.2):



Rysunek 6.4.2 Polecenia PyDev i ich skróty, uruchamiające i wyłączające serwer zdalnego debuggera Pythona w Eclipse

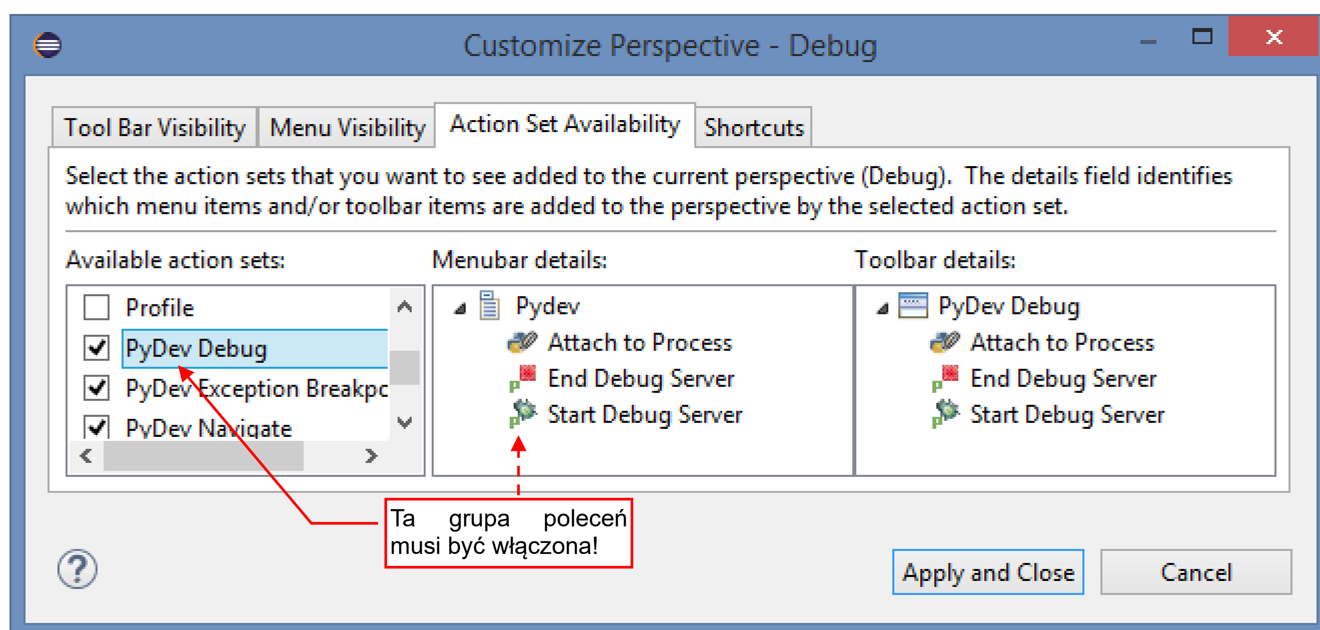
Copyright Witold Jaworski, 2011-2019.

A co zrobić, gdy na pasku przycisków ani w menu *Pydev* nie ma poleceń[1], które pokazuje Rysunek 6.4.2? Czasami polecenia PyDev *Start/End Debug Server* mogą być po prostu wyłączone z perspektywy *Debug*! Aby je włączyć, wywołaj ***Window→Perspective→Customize Perspective*** (Rysunek 6.4.3):



Rysunek 6.4.3 Przejście do dostosowywania perspektywy projektu

W oknie ***Customize Perspective*** przejdź do zakładki ***Action Set Availability*** (Rysunek 6.4.4):



Rysunek 6.4.4 Włączenie wyświetlania kontrolek zdalnego debuggera Pythona

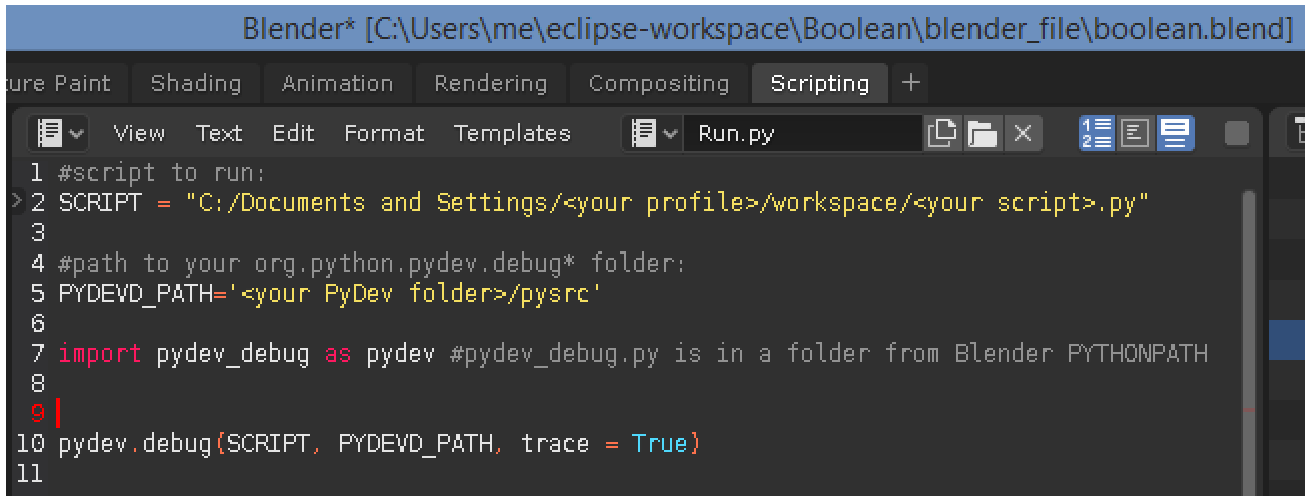
Odszukaj na liście *Available action sets* (po lewej) grupę poleceń o nazwie ***PyDev Debug***. Wystarczy ją zaznaczyć, by polecenia *Start Debug Server* i *End Debug Server* pojawiły się w menu i na pasku przybornika.

Gdy przygotowywałem prezentację na potrzeby tej książki, przyciski *Start/End Debug Server* były od razu na swoim miejscu. Nie musiałem niczego poprawiać w konfiguracji perspektywy. Przypuszczam, że może być to związane z okolicznościami, w których została w projekcie stworzona perspektywa *Debug*.

- Przy okazji dowiedziałeś się, jak można dostosowywać perspektywę projektu Eclipse do swoich potrzeb ☺

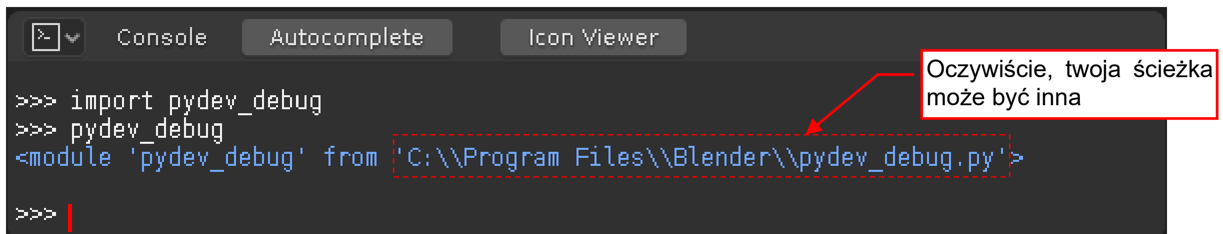
[1] Gdy instalowałem PyDev po raz pierwszy, zdarzyło mi się właśnie coś takiego. Spędziłem wtedy chyba cały dzień na wertowaniu wszelkiej dokumentacji i uwag użytkowników, dostępnych w Internecie. Równolegle ciągle wchodziłem w różne menu Eclipse, w poszukiwaniu tych dwóch kluczowych drobiazgów. W końcu je znalazłem. Abyś oszczędzić Wam tej samej męki, podaję tutaj rozwiązanie tego problemu.

Czas teraz przygotować stronę klienta debuggera. Wśród plików towarzyszących tej książce znajdziesz plik *Run.py* (por. str. 39). Załaduj go do edytora tekstów w testowym pliku Blendera (tym, którego używasz w tym projekcie – por. str. 47) poleceniem ***Text→Open***. Rysunek 6.4.5 przedstawia wygląda przed modyfikacją:



Rysunek 6.4.5 Pomocniczy kod do ładowania skryptów użytkownika do Blendera

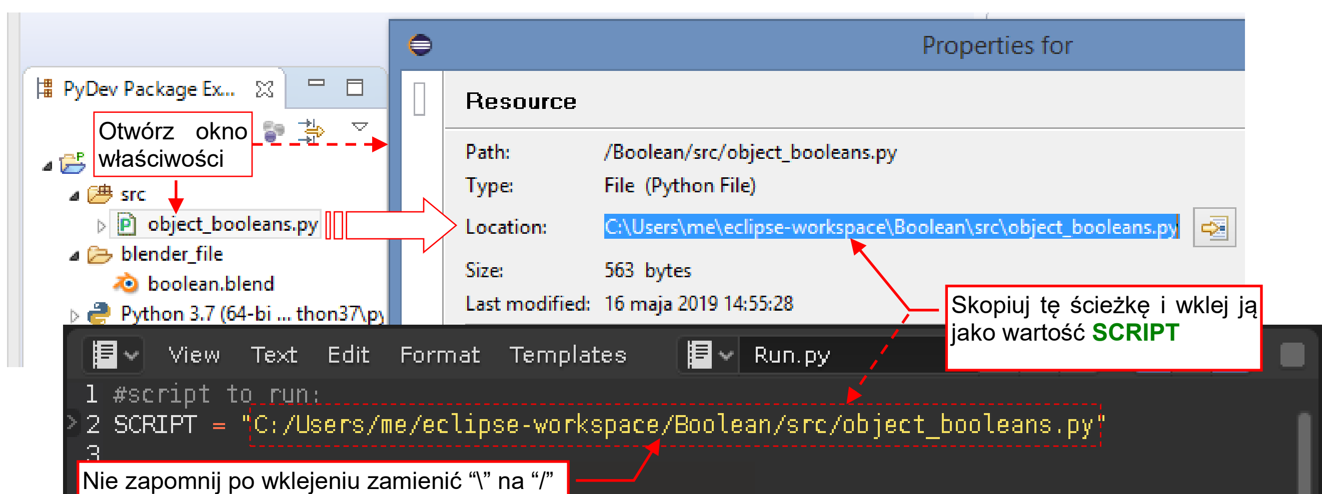
Nim zaczniesz debugować swój skrypt, musisz dostosować tych kilka linii kodu. Zacznijmy od sprawdzenia, czy Python w Blenderze „widzi" *pydev_debug.py*. To kolejny plik towarzyszący tej książce. Sugerowałem, aby umieścić go w tym samym katalogu, w którym jest plik wykonywalny Blendera (por. str. 39). *Run.py* importuje go jak moduł – więc na początku sprawdź w konsoli Pythona Blendera, czy to zadziała. Wywołaj takie polecenia jakie pokazuje Rysunek 6.4.6 i porównaj, czy otrzymałeś takie same odpowiedzi:



Rysunek 6.4.6 Sprawdzanie, czy Python „widzi" plik skrypt *pydev_debug*

Jeżeli polecenie importu tego modułu zgłosi błąd – sprawdź uważnie w konsoli Pythona w Blenderze, jakie foldery figurują na jego liście ***sys.path***. Następnie umieść plik *pydev_debug.py* w jednym z nich.

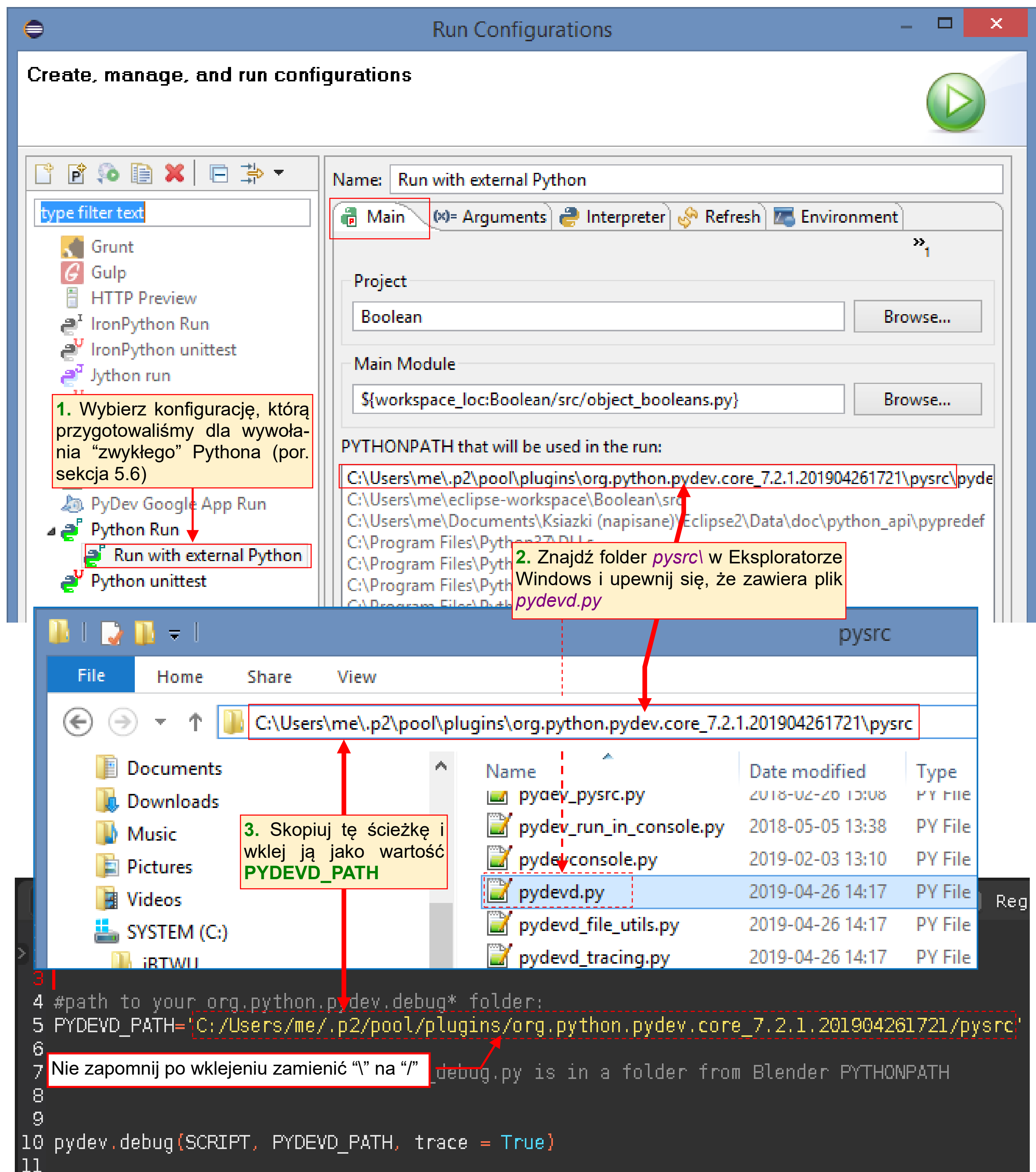
Kolejnym elementem do zmiany jest ścieżka do skryptu, nad którym pracujesz (stała **SCRIPT**). Najprościej skopiować ją z właściwości tego pliku wyświetlanych w oknie Eclipse (Rysunek 6.4.7):



Rysunek 6.4.7 Wpisanie ścieżki do skryptu, który ma być wykonywany

Copyright Witold Jaworski, 2011-2019.

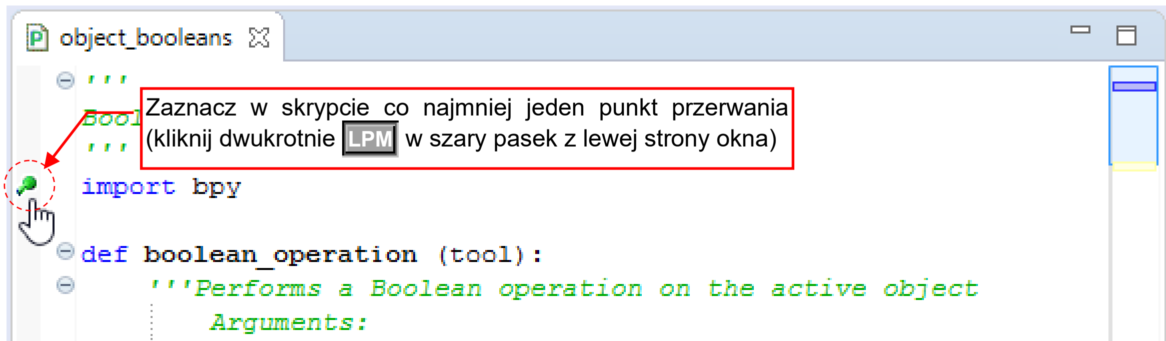
Ostatnim elementem do wpisania w kodzie *Run.py* jest ścieżka do folderu PyDev o nazwie *pysrc* (stała **PYDEV_PATH**). To nieco trudniejsze zadanie, bo ten folder w kolejnych wersjach PyDev znajdował się w różnych folderach na dysku. Najprościej jest odczytać obecne położenie tego folderu z listy *PYTHONPATH* PyDev. W tym celu otwórz (jak pokazuję to w sekcji 5.6 na str. 134) okno ***Run Configurations*** i dla wywołania Pythona odczytaj z listy *PYTHONPATH* odczytaj położenie folderu *pysrc* (Rysunek 6.4.8):



Rysunek 6.4.8 Identyfikacja położenia folderu PYDEVD_PATH

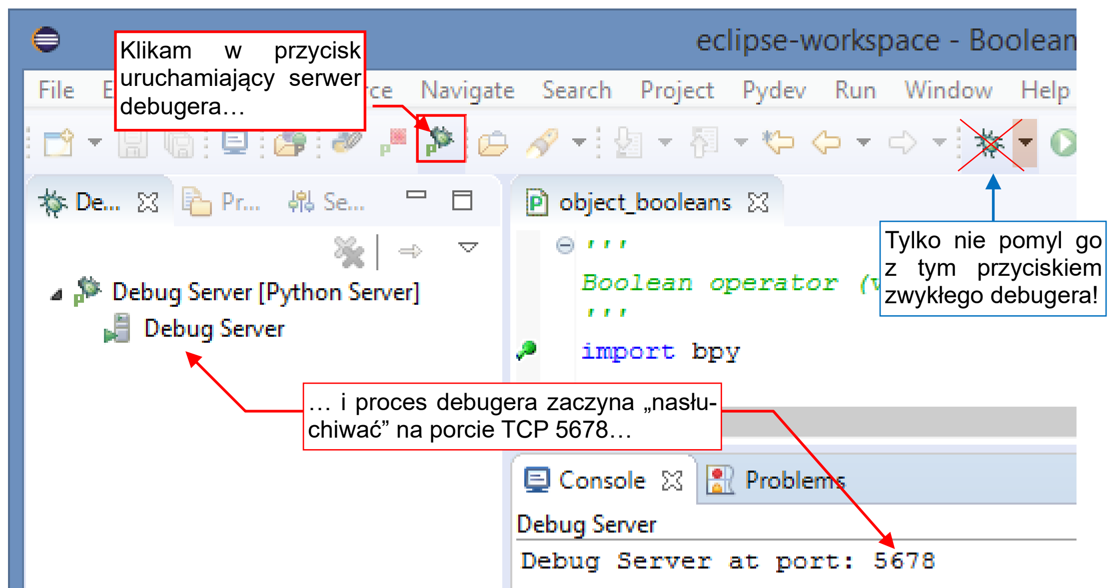
Niestety, z okna *Run Configurations* nie można skopiować ścieżki do *pysrc*. Pozostaje otworzyć gdzieś z boku Eksplorator Windows i ręcznie „przejść" w nim ścieżkę wyświetlaną w Eclipse. Upewnij się, że znaleziony folder zawiera plik *pydevd.py* (To moduł do obsługi klienta zdalnego debuggera, wykorzystywany w *pydev_debug.py*). Gdy tak jest - skopiuj z Eksploratora ścieżkę do folderu *pysrc* i wklej ją jako wartość stałej **PYDEVD_PATH**.

Przed rozpoczęciem debugowania zaznacz w skrypcie jakiś punkt przerwania, gdyż inaczej cały skrypt po prostu się wykona bez zatrzymania w debugerze. Jeżeli chciałbyś zacząć śledzić swój kod od samego początku, zaznacz przerwanie na poleceniu importu moduły ***bpy*** (Rysunek 6.4.9):



Rysunek 6.4.9 Umieszczenie punktu przerwania na samym początku skryptu

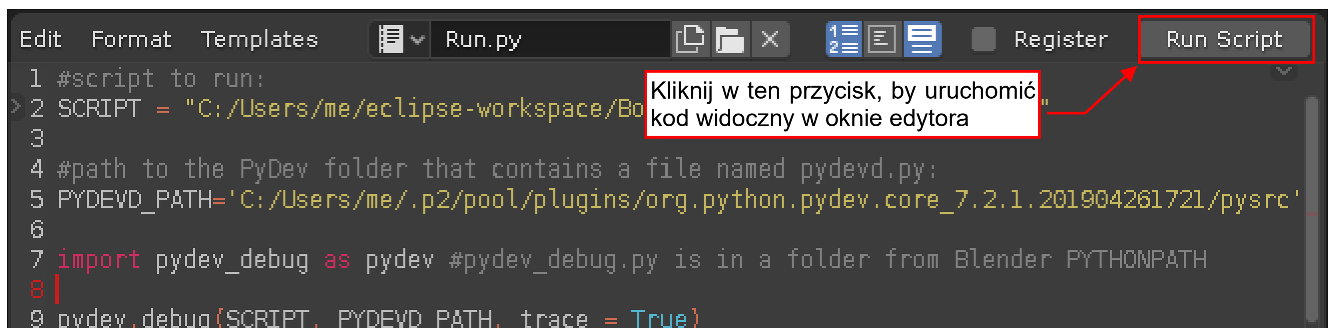
Następnie przejdź do perspektyw *Debug* i włącz zdalny serwer debugera PyDev, aby zaczął „nasłuchiwać" żądań wysyłanych poprzez lokalne połączenie sieciowe (Rysunek 6.4.10):



Rysunek 6.4.10 Włączenie serwera zdalnego debugera

Możesz to zrobić poleceniem z menu: ***PyDev→Start Debug Server*** (por. str. 149) lub przyciskiem z „pluskwą" i małą literką „P", umieszczoną na pasku przybornika (Rysunek 6.4.10). Tylko nie pomyl tego przycisku z domyślnym przyciskiem debugowania! (To większa ikona „pluskwy", bez żadnych liter).

Gdy serwer zaczął „nasłuchiwać", pora uruchomić klienta debugera. Robisz to uruchamiając przygotowany w oknie edytora tekstu testowego pliku Blendera skrypt *Run.py* (Rysunek 6.4.11):



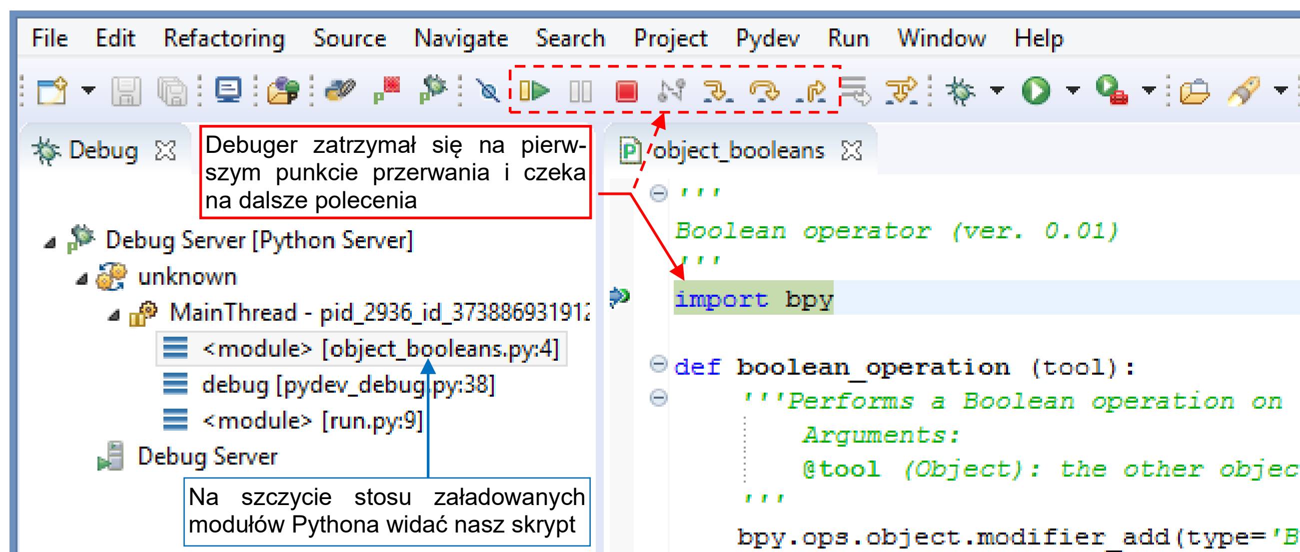
Rysunek 6.4.11 Uruchomienie skryptu Blendera

Zrób to klikając w przycisk ***Run Script*** (znajdziesz go po prawej stronie nagłówka edytora).

Copyright Witold Jaworski, 2011-2019.

Run.py ładuje wskazany skrypt (w zmiennej ***SCRIPT***) i przekazuje do śledzenia w debugerze. W rezultacie okno Blendera ulega „zamrożeniu". Pozostanie w tym stanie, dopóki skrypt nie wykona się do końca.

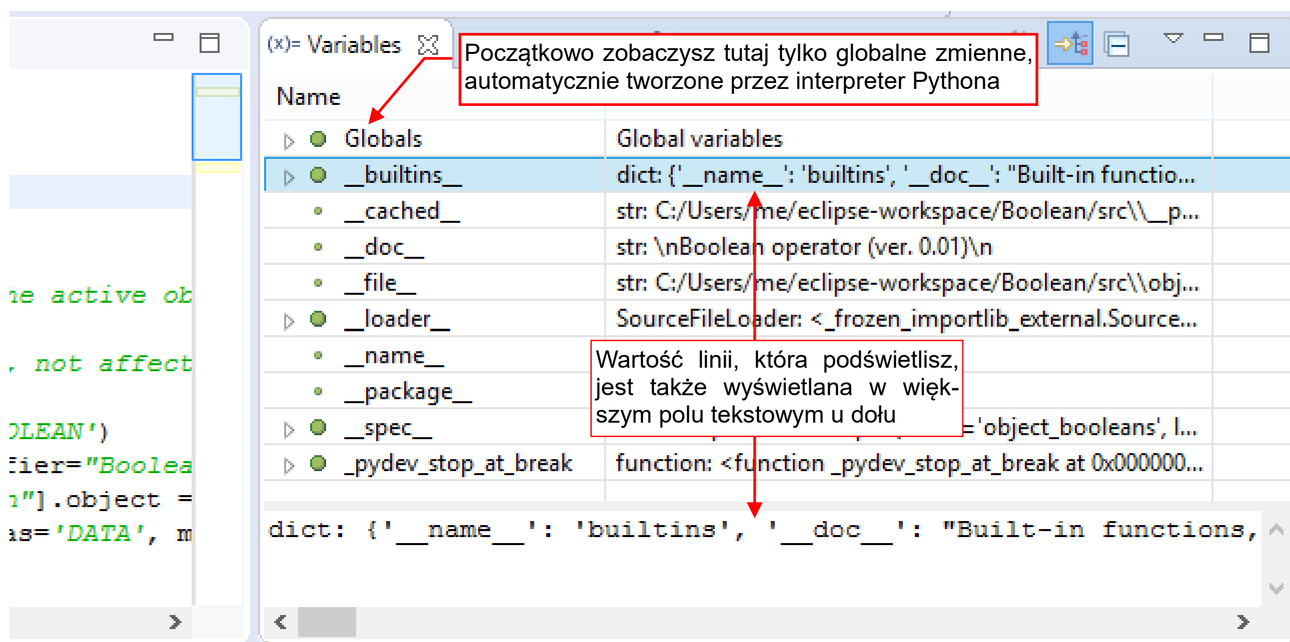
Ale tylko kliknij w okno Eclipse, aby je uaktywnić! Po kilku sekundach zobaczysz linię debugera w tym wierszu kodu, w którym ustawiłeś pierwszy punktem przerwania (Rysunek 6.4.12):



Rysunek 6.4.12 Rozpoczęcie śledzenia skryptu

Po lewej stronie okna edytora jest wyświetlana panel ze stosem wywołań. Na ilustracji powyżej widać, że skrypt został uruchomiony w module *run.py* (to nasz kod w edytorze tekstu Blendera). W linii 9 ten skrypt wywołał procedurę ***debug()*** z *pydev_debug.py*. Tam, w linii 38 jest wywołany skrypt *object_booleans.py*, którego linię 4 podświetla debuger w panelu po lewej. Taka informacja przydaje się szczególnie wtedy, gdy tworzysz jakieś rozwiązanie złożone z kilku plików Pythona.

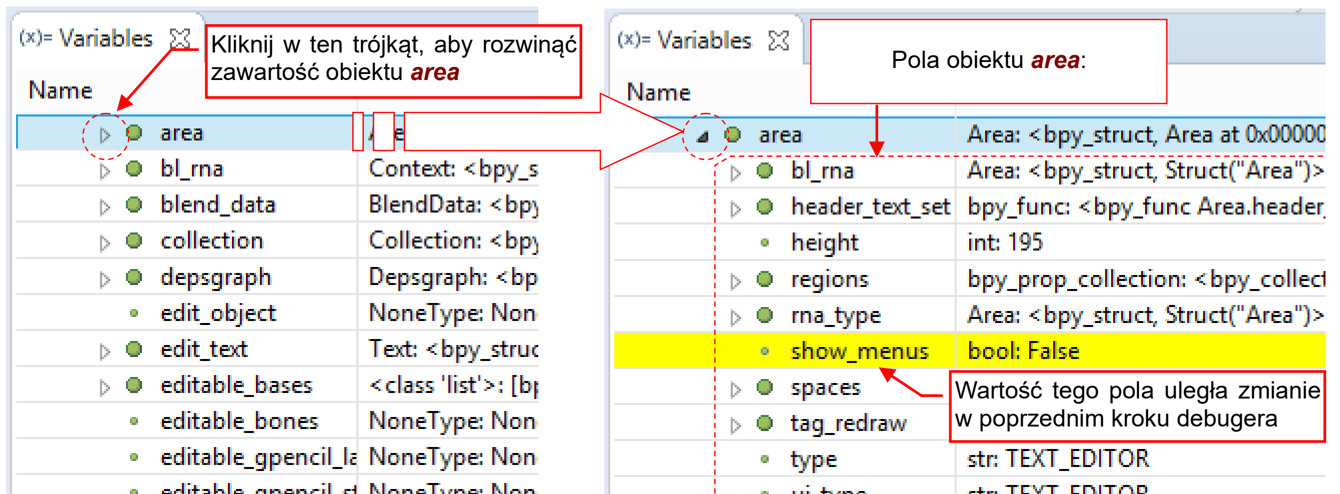
Podczas wykonywania kolejnych linii skryptu będziesz często sprawdzał aktualny stan zmiennych. W Eclipse umożliwia to umieszczona z prawej strony edytora panel ***Variables*** (Rysunek 6.4.13):



Rysunek 6.4.13 Panel śledzenia zmiennych skryptu (globalnych i lokalnych)

Panel jest podzielona na listę z nazwami i wartościami zmiennych globalnych i lokalnych, oraz umieszczony na dole obszar szczegółów (*details*). Obszar szczegółów pokazuje wartość podświetlonej na liście zmiennej. Sądzę, że przydaje się to do sprawdzania jakichś dłuższych ciągów znaków (wartości typu *str*), list lub słowników.

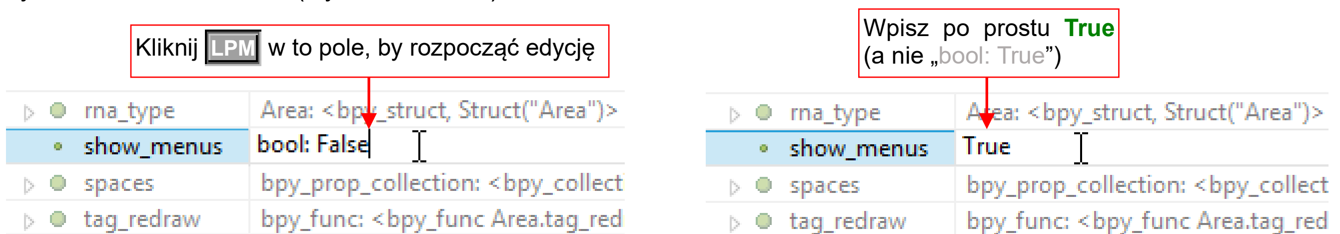
Gdy wartością zmiennej jest referencja do obiektu, Eclipse wyświetla przy niej trójkąt (▷), za pomocą którego możesz rozwinąć listę jego pól: (Rysunek 6.4.14):



Rysunek 6.4.14 Przeglądanie pól obiektu

Te pola mogą być kolejnymi obiektami albo wartościami podstawowych typów (***str***, ***bool***, ***int***, …), oznaczonymi „kropkami". PyDev podświetla na żółto pola, które uległy zmianie w wyniku wykonania przez debuger poprzedniej linii kodu.

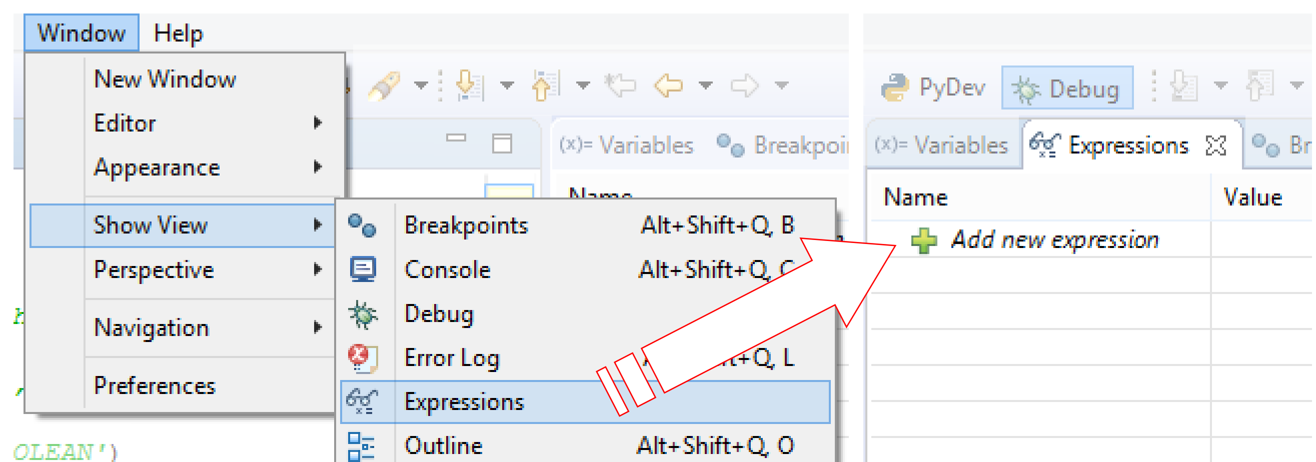
W oknie *Variables* możesz także zmieniać aktualne wartości zmiennych. Zazwyczaj będziesz je po prostu wpisywał w kolumnie *Value* (Rysunek 6.4.15):



Rysunek 6.4.15 Zmiana wartości zmiennej/pola

Wartość zmiennej można także zmienić w obszarze *details* (poleceniem ***Assign Value*** z menu kontekstowego). Zwróć uwagę, że nowe wartości zmiennych podajesz w sposób naturalny dla Pythona: **True**, **False**, **1**, **tekst**, …. Nie poprzedzaj ich przedrostkiem z nazwą typu („bool: True"), choć PyDev w ten sposób je wyświetla. Po wykonaniu aktualnej linii kodu PyDev zaznaczy na żółto także tę ręcznie zmienioną zmienną/pole.

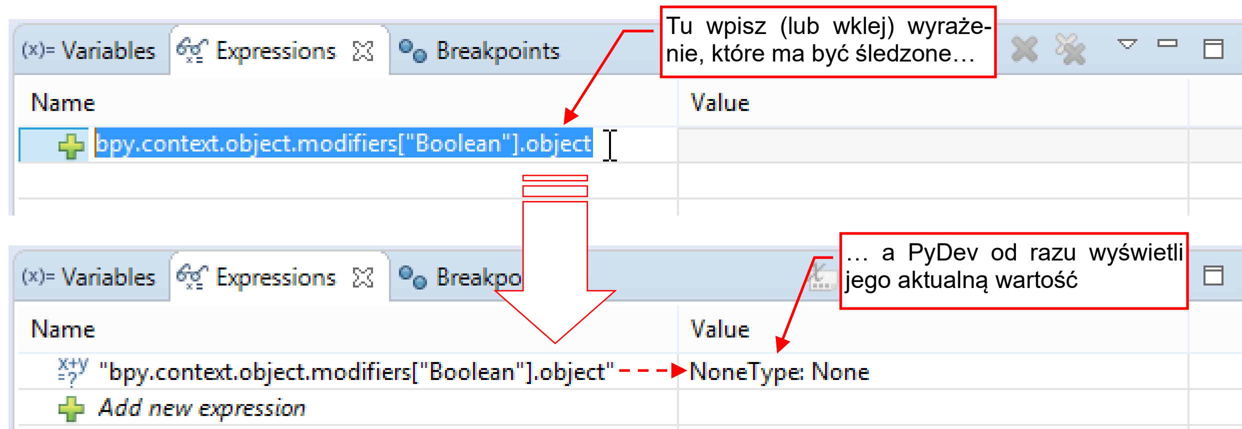
Do śledzenia pojedynczego pola obiektu wygodniej jest korzystać z panelu ***Expressions***. Możesz go dodać do perspektywy poleceniem ***Window→Show View→Expressions*** (Rysunek 6.4.16):



Rysunek 6.4.16 Dodawanie panelu *Expressions*

Copyright Witold Jaworski, 2011-2019.

Układ panelu *Expressions* przypomina układ *Variables*: tu także mamy listę z nazwą i wartością wyrażenia. Jest tu także pole *details*, pokazujące na większym obszarze wartość zaznaczonej pozycji. To, co różni je od *Variables* to możliwość wpisania dowolnego wyrażenia, które ma być wartościowane po każdym kroku debuggera (Rysunek 6.4.17):

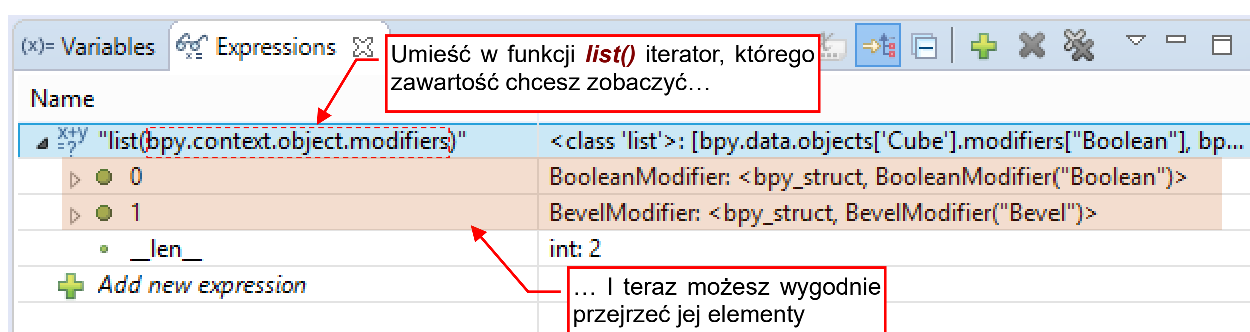


Rysunek 6.4.17 Dodawanie śledzonych wyrażeń do listy *Expressions*

W oknie *Expressions* można wpisać po prostu nazwę zmiennej. Częściej jednak jest wykorzystywane do śledzenia wybranych pól jakiegoś obiektu. W przykładzie powyżej wpisałem odwołanie do pola ***object*** modyfikatora aktywnego obiektu o nazwie **Boolean**. Robię to, gdyż pole ***modifiers*** jest tzw. iteratorem, a nie listą, i nie można przejrzeć jej elementów w panelu *Variables*. (Obiekt ***bpy.context*** znajdziesz w panelu *Variables* w: *Globals→'bpy'→context*. Sam się przekonaj, jakie pola eksponuje jego ***object.modifiers***). W odróżnieniu od okna *Variables*, w oknie *Expressions* nie można zmieniać wartości wyświetlonych wyrażeń.

- Okno *Expressions* jest bardzo przydatne do sprawdzania zawartości iteratorów. W szczególności dotyczy to elementów wszelkich list Blender API.

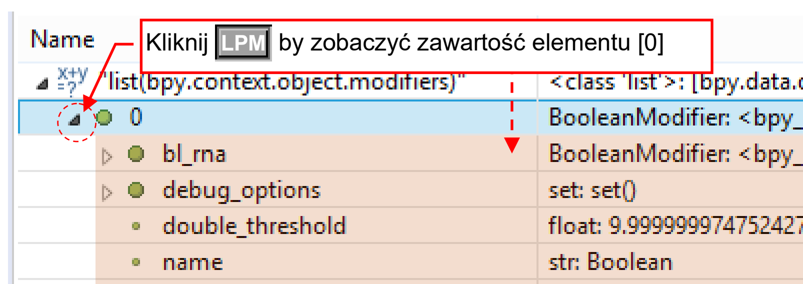
Najszybszym sposobem na przejrzenie całej zawartości iteratora jest zamiana na listę za pomocą standardowej funkcji ***list()*** (Rysunek 6.4.18):



Rysunek 6.4.18 Przeglądanie w oknie *Expressions* szczegółów stosu modyfikatorów

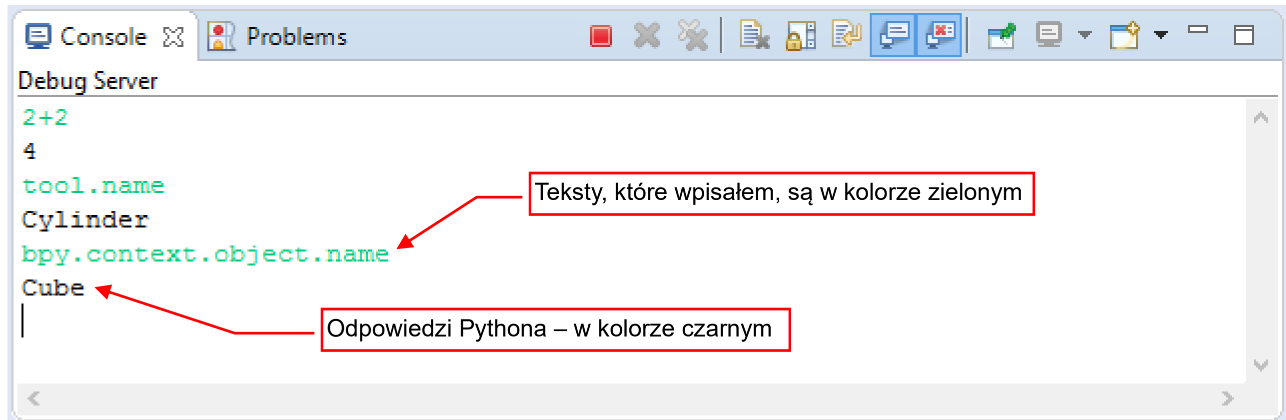
Rzecz jasna, nie próbuj tego robić dla zbyt długich list. Jeżeli nie jesteś pewien, ile elementów zawiera iterator, możesz to wcześniej sprawdzić za pomocą standardowej funkcji ***len()***.

Tu także możesz kliknąć w trójkąt (▷) z lewej strony każdego złożonego typu danych, by przejrzeć jego zawartość (Rysunek 6.4.19):



Rysunek 6.4.19 Szczegóły rezultatu wyrażenia z okna *Expressions*

Innym elementem, który czasami może się przydać, jest konsola (panel ***Console***). W czasie debugowania przekierowane jest na nią to, co Blender wyświetla w swojej konsoli (zobacz *Windows→Toggle system console* w menu Blendera). Co więcej, na czas debugowania staje się interaktywna: możesz w niej wpisywać dowolne wyrażenie, a Python Blendera je wykona (Rysunek 6.4.20):

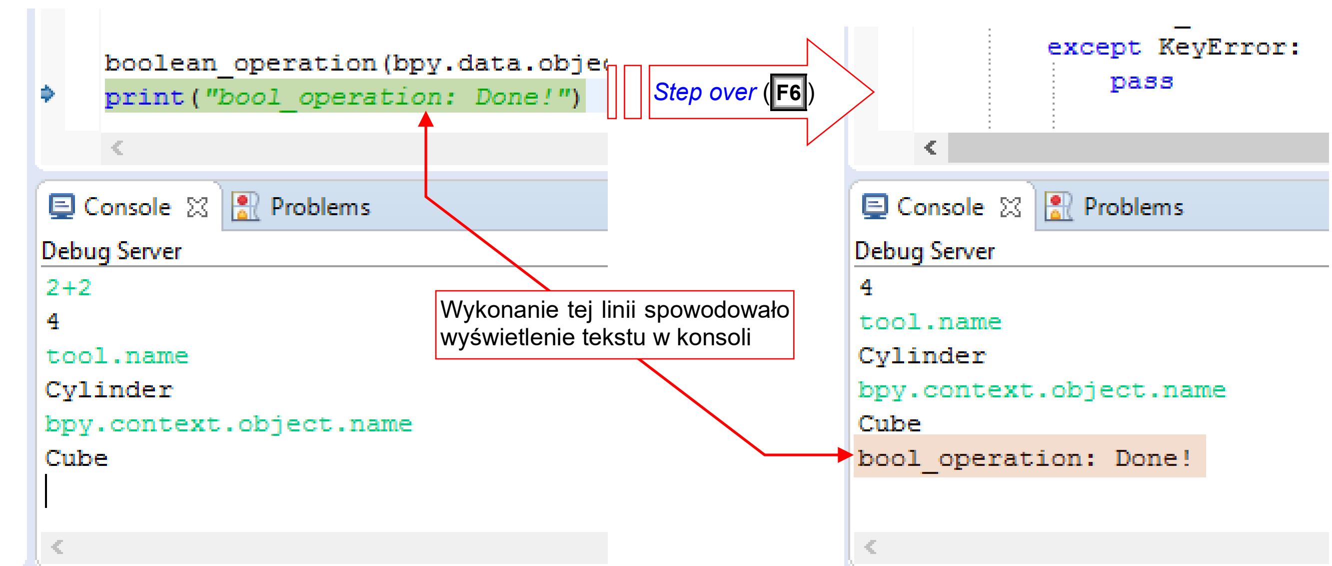


Rysunek 6.4.20 Konsola Eclipse w czasie debugowania skryptu Pythona

Oczywiście, to samo można sprawdzić za pomocą panelu *Expressions*. W konsoli możesz jednak więcej — na przykład wywołać metodę jakiegoś obiektu.

- Choć ekran Blendera w czasie wykonywania skryptu jest „zamrożony” i wygląda przez cały czas tak, jak w chwili naciśnięcia przycisku *Run Script* (por. str. 153), to za pomocą konsoli możesz nim nadal sterować. Możesz np. wykonać dowolne polecenie Blendera, wywołując w tym oknie odpowiednią metodę ***bpy.ops***.

Gdy w skrypcie wywołujesz procedurę ***print()***, to w czasie debugowania jej rezultat pojawi się nie tylko w konsoli Blendera, ale i Eclipse (Rysunek 6.4.21):

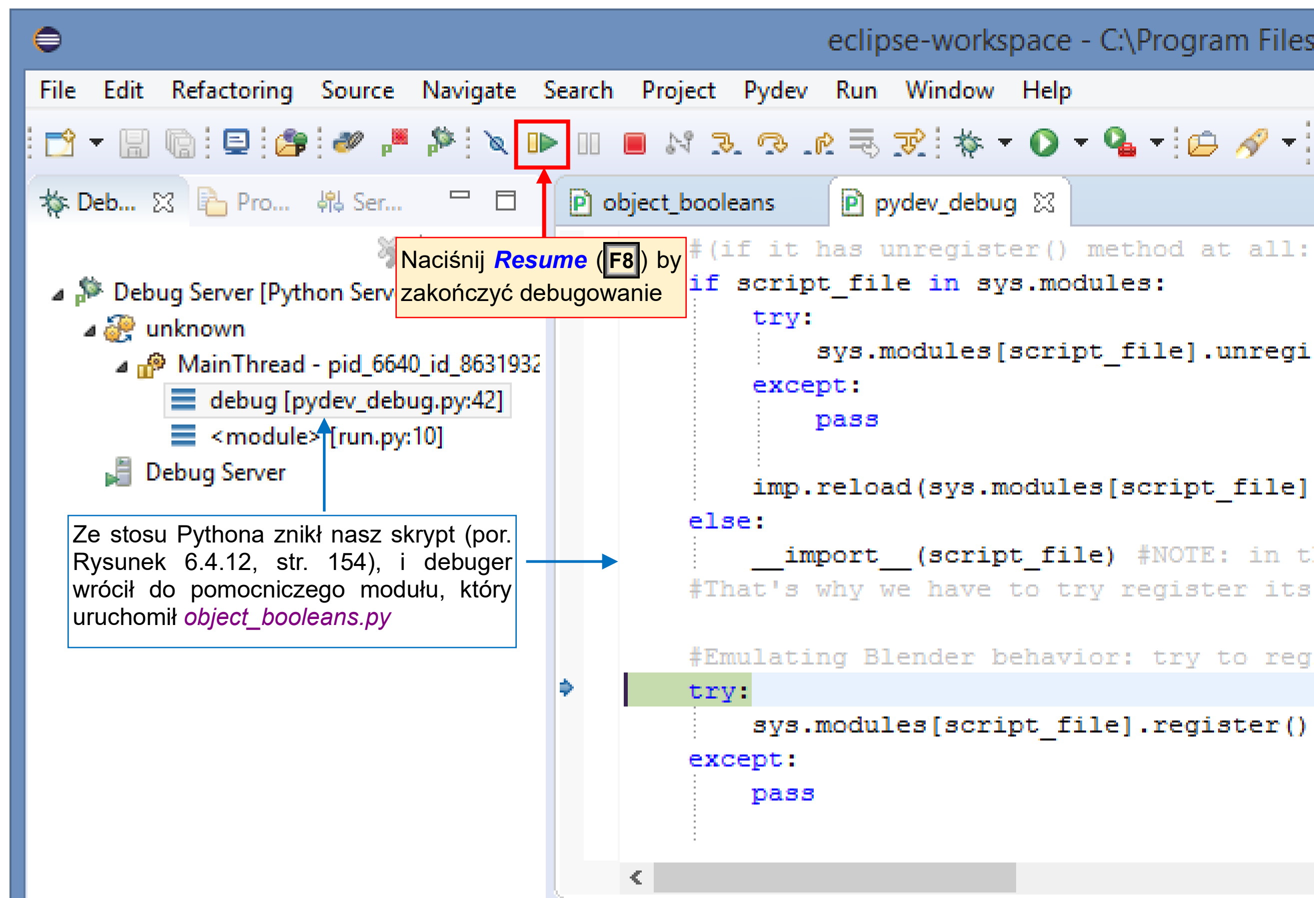


Rysunek 6.4.21 Tekst z polecenia *print()* w konsoli Eclipse

A gdy podczas wykonywania wystąpi jakiś błąd, to właśnie w konsoli znajdziesz jego dokładny opis.

Copyright Witold Jaworski, 2011-2019.

Po wykonaniu ostatniej linii skryptu nasz debuger przechodzi do pomocniczego pliku, który odpowiadał za poprawne załadowanie i uruchomienie tego skryptu (Rysunek 6.4.22):



Rysunek 6.4.22 Ekran debugera po wykonaniu ostatniej linii skryptu

Przy pierwszym uruchomieniu skryptu będzie to *pydev_debug.py*, przy kolejnych – inny, standardowy moduł Pythona, wykonujący „przeładowanie” modułu. W każdym razie nie mamy to nic do roboty – więc kliknij w przycisk ***Resume*** (lub naciśnij F8) by zakończyć działanie skryptu.

- Niestety, do tej pory nie wymyśliłem jeszcze żadnego „eleganckiego” sposobu, w którym po zakończeniu podstawowego skryptu wykonałby polecenie *Resume* w sposób automatyczny i nie otwierał tych pomocniczych plików.

Gdy to zrobisz, ekran Blendera zostanie „odmrożony” i zobaczysz w nim rezultat działania skryptu. Jeżeli chcesz wycofać dokonane przez Twój kod zmiany – wystarczy pojedyncze wywołanie operacji *Undo* (Ctrl-Z).

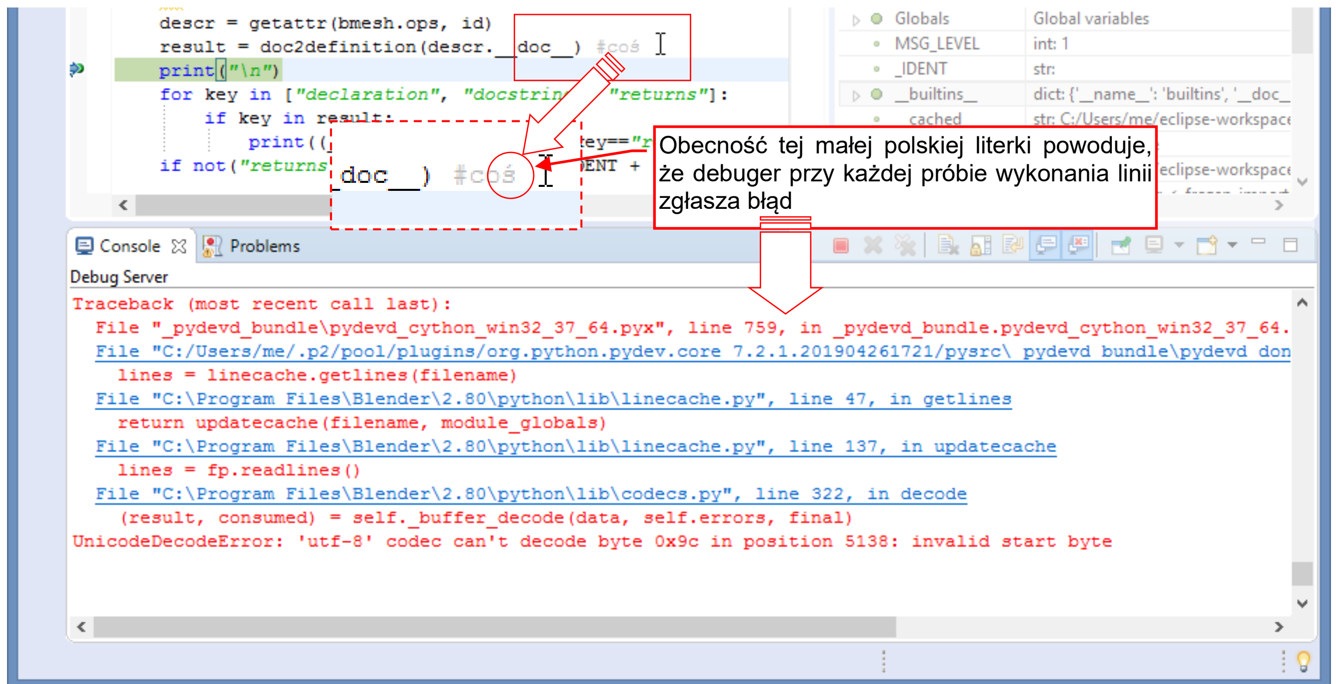
Teraz możesz zmodyfikować w edytorze Eclipse kod skryptu. Aby go ponownie uruchomić/zdebugować wystarczy znów kliknąć w Blenderze w przycisk *Run Script* (por. str. 153, Rysunek 6.4.11).

- Nie wyłączaj nigdy raz uruchomionego w Eclipse procesu serwera zdalnego debugera (por. str. 153, Rysunek 6.4.10). Niech zamknie się sam, wraz z zamknięciem całego środowiska Eclipse.

- Aby załadować do Blendera zmodyfikowany kod skryptu i go zdebugować, wystarczy powtórnie nacisnąć przycisk *Run Script* (jak na str. 153).

W czasie dalszej pracy nad projektem wtyczki nie będziesz już musiał niczego zmieniać w pliku *Run.py*, umieszczonym w edytorze tekstowym testowego pliku Blendera. Potrzebny ci będzie tylko przycisk *Run Script*, umieszczony w nagłówku tego edytora. Możesz więc zmniejszyć to okno do kilku linii.

Na koniec jeszcze uwaga dotycząca stosowania znaków specjalnych (takich, które w standardzie UTF-8 wymagają dwóch znaków). Mogą to być na przykład polskie litery. Nigdy ich nie używaj w skrypcie, których chcesz debugować. Wystarczy jeden polski znak, aby PyDev się „pogubił" i przy każdej próbie wykonania podświetlonej linii, wypisywał w konsoli błąd (Rysunek 6.4.23):



Rysunek 6.4.23 Błąd PyDev, spowodowany przez znak specjalny (o kodzie ASCII > 127)

Możesz taki znak usunąć z komentarza nawet nie przerywając debugowania: wystarczy, że to zrobisz w edytorze Eclipse w którym debugujesz kod i zapiszesz tę zmienioną wersję skryptu na dysk. Debuger uaktualni wówczas jej źródło w pamięci Blendera i problem zniknie: polecenia *Step Over*, *Step Into*, i wszystkie pozostałe zaczną być poprawnie wykonywane.

Copyright Witold Jaworski, 2011-2019.

6.5 Co się kryje w pliku *pydev_debug.py*?

Właściwie do uruchomienia śledzenia skryptu Blendera w zdalnym debugerze PyDev wystarczyłyby takie dwie linie kodu (Rysunek 6.5.1):

```
    import pydevd
    pydevd.settrace()#<-- debugger stops at the next statement
```

Rysunek 6.5.1 Kod, ładujący i uruchamiający klienta zdalnego debuggera PyDev

Oczywiście, aby ten kod zadziałał, należałoby wcześniej dodać do ***PYTHONPATH*** folder, w którym znajduje się pakiet (*package*) *pydev.py*. Zresztą ten warunek nie wyczerpuje jeszcze listy wszystkiego, co trzeba lub warto wykonać podczas inicjalizacji. Stąd te dwie linie „obrosły" w całą procedurę ***debug()*** (Rysunek 6.5.2):

```
'''Utility that runs Blender scripts and addons in Eclipse PyDev debugger.
Place this file somwhere in a folder that exists on Blender's sys.path
(You can check sys.path content in the Python Console)
'''
import sys
import os
import imp

def debug(script, pydev_path, trace = True):
    '''Run script in PyDev remote debugger
        Arguments:
        @script (string): full path to script file
        @pydev_path (string): path to your org.python.pydev.debug* folder
                              (in Eclipse directory)
        @trace (bool): whether to start debugging
    '''
    script_dir = os.path.dirname(script) #directory, where the script is located
    script_file = os.path.splitext(os.path.basename(script))[0] #script filename,
    #                                                   (without ".py" extension)
    #update the PYTHONPATH for this script.
    if sys.path.count(pydev_path) < 1: sys.path.append(pydev_path)
    if sys.path.count(script_dir) < 1:  sys.path.append(script_dir)
    #NOTE: These paths stay in PYTHONPATH even when this script is finished.
    #try to not use scripts having identical names from different directories!

    import pydevd
    if trace: pydevd.settrace(stdoutToServer=True, stderrToServer=True,
                              suspend=False) #stop at first breakpoint

    #Emulating Blender behavior: try to unregister previous version of this module
    #(if it has unregister() method at all:)
    if script_file in sys.modules:
        try:
            sys.modules[script_file].unregister()
        except:
            pass

        imp.reload(sys.modules[script_file])
    else:
        __import__(script_file) #NOTE: in the script loaded this way:
                                #__name__ != '__main__'
    #That's why we have to try register its classes:
    #Emulating Blender behavior: try to register this version of this module
    #(if it has register() method...)
    try:
        sys.modules[script_file].register()
    except:
        pass
```

Przygotowanie otrzymanych ścieżek, rozszerzenie ***PYTHONPATH***

Nawiązanie połączenia z serwerem debugera

Emulacja obsługi dodatków Blendera (*add-ons*): wyrejestrowanie poprzedniej wersji wtyczki

Wykonanie głównego kodu skryptu

Emulacja obsługi dodatków Blendera (*add-ons*): zarejestrowanie nowej wersji wtyczki

Rysunek 6.5.2 Skrypt *pydev_debug.py*

Zdecydowałem się wydzielić główny kod uruchamiający w Blenderze skrypt z projektu Eclipse w oddzielny moduł *pydev_debug.py*. Ten moduł zawiera tylko jedną procedurę: ***debug()*** (Rysunek 6.5.2). Pozwoliło to na maksymalne uproszczenie skryptu *Run.py* — wzorca wywołania, dostosowywanego do nowego projektu (por. str. 53). *Run.py* to po prostu wywołanie procedury ***debug()*** z następującymi parametrami:

- ***script***: ścieżka do pliku skryptu, który ma być uruchomiony;
- ***pydev_path***: scieżka PyDev, w której znajduje się plik *pydevd.py*;
- ***trace***: opcjonalny. *True*, gdy program ma być śledzony w debuggerze, *False* gdy ma być wykonany bez śledzenia. (Tylko wtedy, gdy ***trace*** = *False*, możesz wywoływać tę procedurę z wyłączonym procesem serwera PyDev w Eclipse — por. 149);

Zwróć uwagę (Rysunek 6.5.2), że procedura ***debug()*** wczytuje podany moduł ***script*** klauzulą *import*. To pozwala używać jej także do debugowania wtyczki (*add-on*) Blendera[1]. Przed importem program próbuje potraktować załadowany moduł jako *add-on* i wyrejestrować jego poprzednią wersję. Jak się nie uda — nie ma błędu (nie każdy skrypt musi być wtyczką). Po załadowaniu skryptu próbuje z kolei go zarejestrować.

- Gdy piszesz kod wtyczki Blendera (*add-on*), zawsze dopisz na jej końcu obydwie wymagane metody: ***register()*** i ***unregister()***. Pozwoli to na poprawne inicjowanie kodu za każdym wywołaniem w Blenderze. (Tzn. po każdym naciśnięciu przycisku ***Run Script*** w oknie z kodem *Run.py* — por str. 55).

[1] Blender ładuje kod takiej wtyczki i rejestruje jej klasy API (pochodne ***bpy.types.Operator***, ***Menu*** lub ***Panel***). Ten skrypt pozostaje załadowany, i Blender wywołuje z niego odpowiednią metodę, gdy uzna to za stosowne. Skrypt *pydev_debug.py* poprawnie obsługuje także taki schemat działania. Nie radzi sobie tylko z obsługą wtyczki, która została już zainstalowana – tzn. taką, która jest już umieszczona w specjalnym folderze Blendera z innymi wtyczkami, i widoczna na liście wtyczek w oknie *Blender Preferences*. Taka instalacja jest potrzebna tylko do testowania ewentualnego panelu preferencji wtyczki (por. str. 106). Większość dodatków jej nie potrzebuje. (Różnice pomiędzy wtyczką Blendera a zwykłym skryptem Pythona opisuje Rozdział 4).

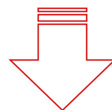
Copyright Witold Jaworski, 2011-2019.

6.6 Pełen kod wtyczki *object_booleans.py*

W kolejnych rozdziałach tej książki stopniowo rozbudowywałem kod jednego skryptu: *object_booleans.py*. W tekście przytaczałem dodawane fragmenty kodu. Jednak po tylu modyfikacjach warto jest pokazać ostateczny rezultat w całości. Gdybyś chciał skopiować ten tekst do schowka — uważaj na wycięcia! Podczas kopiowania z PDF wszystkie są usuwane. Lepiej chyba będzie po prostu pobrać ten plik z mojej strony.

Skrypt nie mieści się na jednej stronie, więc zdecydowałem się go podzielić na pięć części. Pierwsza zawiera formalne deklaracje oraz pomocnicze linie pozwalające wywołać debuger gdy wtyczka jest już zainstalowana (Rysunek 6.6.1):

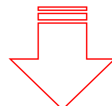
```
# ##### BEGIN GPL LICENSE BLOCK #####
#
#  This program is free software; you can redistribute it and/or
#  modify it under the terms of the GNU General Public License
#  as published by the Free Software Foundation; either version 2
#  of the License, or (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with this program; if not, write to the Free Software Foundation,
#  Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
#
# ##### END GPL LICENSE BLOCK #####

'''
Boolean operator (ver. 1.0)
'''

bl_info = {
    "name": "Boolean operations",
    "description": "Performs simple ('destructive') Boolean operation on selected objects",
    "author": "Witold Jaworski",
    "version": (1, 0),
    "blender": (2, 80, 0),
    "location": "Object > Boolean",
    "support": "TESTING",
    "category": "Object",
    "warning": "Still in the 'beta' version - use with caution",
    "wiki_url": "http://airplanes3d.net/scripts-258_e.xml",
    "tracker_url": "http://airplanes3d.net/scripts-258_e.xml",
        }

DEBUG = 0 #A debug flag - just for the convinience (Set to 0 in the final version)

###--- for direct debugging of this add-on (update the pydevd path!) ---------------------------
if DEBUG == 1:
    import sys
    pydev_path = 'C:/Users/me/.p2/pool/plugins/org.python.pydev.core_7.2.1.201904261721/pysrc'
    if sys.path.count(pydev_path) < 1: sys.path.append(pydev_path)
    import pydevd
    pydevd.settrace(stdoutToServer=True, stderrToServer=True, suspend=False) #stop at first breakpoint
###-- end remote debug initialization -----------------------------------------------------------
```


Ciąg dalszy na następnej stronie…

Rysunek 6.6.1 Skrypt *obect_booleans.py*, cz. 1 (deklaracje)

W dalszej części kodu znajdują się dwie funkcje obsługujące właściwą akcję wtyczki (Rysunek 6.6.2):

```
import bpy
import traceback #for error handling

def boolean_operation (tool, op, apply=True):
    '''Performs a Boolean operation on the active object
        Arguments:
        @tool (Object): the other object, not affected by this method
        @op (Enum): a Boolean operation: {'UNION', 'INTERSECT', 'DIFFERENCE'}
        @apply (bool): apply results to the mesh (optional)
    '''
    obj = bpy.context.object #active object
    bpy.ops.object.modifier_add(type='BOOLEAN')#adds new modifier to obj
    mod = obj.modifiers[-1] #new modifier always appear at the end of this list
    while obj.modifiers[0] != mod: #move this modifier to the first position
        bpy.ops.object.modifier_move_up(modifier=mod.name)
    mod.operation = op #set the operation
    mod.object = tool #activate rhe modifier
    if apply: #applies modifier results to the mesh of the active object (obj):
        if obj.users > 1 or obj.data.users > 1: #obj has to be a single-user datablock
            #make sure, that obj is the only selected object:
            bpy.ops.object.select_all(action='DESELECT') #deselect all
            obj.select_set(True) #select obj, only
            bpy.ops.object.make_single_user(type='SELECTED_OBJECTS', object=True, obdata=True)
        bpy.ops.object.modifier_apply(apply_as='DATA', modifier=mod.name)

#result constants:
INPUT_ERR = 'ERROR_INVALID_CONTEXT'
ERROR = 'ERROR'
WARNING = 'WARNING'
SUCCESS = 'OK'

def main (op, apply_objects=True, cntx=None):
    ''' Performs a Boolean operation on the active object, using the other
        selected objects as the 'tools'
        Arguments:
        @op (Enum): a Boolean operation: {'UNION', 'INTERSECT', 'DIFFERENCE'}
        @apply_objects (bool): apply results of the Boolean operation to the mesh (optional)
        @cntx (bpy.types.Context): overrides current context (optional)
        @returns (list): one or two message parts: [<flag>, Optional_details]
    '''
    try:
        if cntx == None: cntx = bpy.context
        selected = list(cntx.selected_objects) #creates a static copy
        active = cntx.object #active object
        if active in selected: selected.remove(active)
        #input validation:
        if active.type != 'MESH':
            return [INPUT_ERR, "target object ('%s') is not a mesh" % active.name]
        if active.library != None or active.data.library != None:
            return [INPUT_ERR, "target object ('%s') is linked from another file" % active.name]
        if not selected: return [INPUT_ERR, "this operation requires at least two objects"]
        #main loop
        skipped = []#auxiliary list for the skipped object names
        for tool in selected:  #Apply each tool to the active object:
            if tool.type == 'MESH':
                boolean_operation(tool,op, apply_objects)
            else: #store the name of the skipped object
                skipped.append(tool.name)
        #let's look at the results:
        if not skipped: return [SUCCESS]
        if len(skipped) < len(selected): #still there are a few procesed objects"
            return [WARNING, "completed, but skipped non-mesh object(s): '%s'"
                                                                % str.join("', '",skipped)]
        else: #no object was processed:
            return [INPUT_ERR, "non-mesh object(s) selected: '%s' " % str.join("', '",skipped)]
    except Exception as err: #Just in case of a run-time error:
        traceback.print_exc() #print the Python stack details in the console (for you)
        cntx_msg = "" #format the diagnostic message:
        if 'active' in locals(): cntx_msg += "occured for object(s): '%s'" % active.name
        if 'tool' in locals(): cntx_msg += ", '%s'" %tool.name
        return [ERROR, "%s %s" % (str(err),cntx_msg)]
```



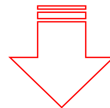
Ciąg dalszy na następnej stronie…

Rysunek 6.6.2 Skrypt *obect_booleans.py*, cz. 2 (kod główny)

Copyright Witold Jaworski, 2011-2019.

W kolejnej części znajduje się wymagana przez API „obudowa”, w postaci klasy operatora. Jest tu także deklaracja dodatkowego interfejsu użytkownika: *pie menu* (Rysunek 6.6.3):

```
#---------- ### Operator -----------
from bpy.props import EnumProperty, BoolProperty

class OBJECT_OT_Boolean(bpy.types.Operator):
    '''Performs a 'destructive' Boolean operation on the active object
       Arguments:
       @op (Enum): Boolean operation, in ['DIFFERENCE', 'UNION', 'INTERSECT']
       @modifier (Bool): add this operation as the object modifier
    '''
    bl_idname = "object.boolean"
    bl_label = "Boolean"
    bl_description = "Perform a Boolean operation on active object"
    bl_options = {'REGISTER', 'UNDO'} #Set this options, if you want to update
    #                                  parameters of this operator interactively
    #                                  (in the Tool pane)
    op :  EnumProperty(items = [
                            ('DIFFERENCE',"Difference","Boolean difference", 'SELECT_SUBTRACT',1),
                            ('UNION',"Union","Boolean union", 'SELECT_EXTEND',2),
                            ('INTERSECT',"Intersection","Boolean intersection", 'SELECT_INTERSECT',3),
                        ],
                        name = "Operation",
                        description = "Boolean operation",
                        default='DIFFERENCE',
                       ) #end EnumProperty
    modifier : BoolProperty(name = "Keep as modifier",
                         description = "Keep the results as the object modifier",
                         default = False,
                       ) #end BoolProperty

    @classmethod
    def poll(cls, context):
        return (context.mode == 'OBJECT')

    def execute(self, context):
        main(self.op, apply_objects = not self.modifier, cntx = context)
        return {'FINISHED'}

    def invoke(self, context, event):
        result = main(self.op, apply_objects = not self.modifier, cntx = context)
        if result[0] == SUCCESS:
            return {'FINISHED'}
        else:
            self.report(type = {result[0]}, message = result[1])
            return {'FINISHED' if result[0] == WARNING else 'CANCELLED'}

#---------- # Pie Menu (invoked by the hotkey)  -----------
class VIEW3D_MT_Boolean(bpy.types.Menu):
    '''This pie menu shows Boolean operator options.
       Invoked by the hotkey assignet to this add-on
    '''
    bl_idname = "VIEW3D_MT_Boolean" #Menu identifier has to contain a '_MT_'
    bl_label = "Select operation" #Central label of the pie menu

    def draw(self, context):
        pie = self.layout.menu_pie()
        pie.operator_enum(OBJECT_OT_Boolean.bl_idname, property="op")
```

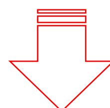


Ciąg dalszy na następnej stronie…

Rysunek 6.6.3 Skrypt *obect_booleans.py*, cz. 3 (klasy operatora i *pie menu*)

W kolejnej części umieściłem obsługę panelu preferencji wtyczki oraz funkcje dodające/usuwające skróty klawiszy (Rysunek 6.6.4):

```
#---------- # Add-On Preferences -----------
#default values for the keymap_items.new() call (see register_keymap() method, below)
hotkey_defaults  = {"idname": 'wm.call_menu_pie',
                    "type": 'D', "value": 'PRESS', "shift": False, "ctrl": False, "alt":False}

class Preferences(bpy.types.AddonPreferences):
    '''This class provides the user pssibility of altering the keyboard shortcut
       assigned to the Boolean pie menu
    '''
    bl_idname = __name__ #do not change this line

    def on_update(self, context):
        unregister_keymap()
        register_keymap()

    shift : BoolProperty(name = "Shift", description= "Use the [Shift] key",
                                    default=hotkey_defaults["shift"], update = on_update)
    ctrl : BoolProperty(name = "Ctrl", description= "Use the [Ctrl] key",
                                    default=hotkey_defaults["ctrl"], update = on_update)
    alt : BoolProperty(name = "Alt", description= "Use the [Alt] key",
                                    default=hotkey_defaults["alt"], update = on_update)
    key : EnumProperty(items = [('NONE',"None","No hotkey")] +
                    [tuple([chr(i),chr(i),"[%s] key" % chr(i)]) for i in range(65, 91)],
                    name = "Keyboard key",
                    description = "Selected keyboard key",
                    default = hotkey_defaults["type"],
                    update = on_update
                    )
    def draw(self, context):
        row = self.layout.row(align=True)
        row.alignment = 'LEFT'
        row.separator(factor = 10)
        row.prop(self,"key", text="Keyboard shortcut")
        row.separator(factor = 3)
        row.label(text="with:")
        row.prop(self,"shift")
        row.prop(self,"ctrl")
        row.prop(self,"alt")

#---------- # hotkey registartion
addon_keymaps = [] #global list for this add-on keyboard shortcut definitions

def register_keymap():
    '''Registers current hotkey'''
    #assumption: at this moment the addon_keymaps[] list is empty
    args = hotkey_defaults #use defaults in case when there is no preferences
    if Preferences.bl_idname in bpy.context.preferences.addons: #update args, according preferences:
        prf = bpy.context.preferences.addons[Preferences.bl_idname].preferences
        args["type"] = prf.key #use the user-defined key and its modifiers:
        args["shift"], args["ctrl"], args["alt"] = prf.shift, prf.ctrl, prf.alt
    else:
        prf = None

    if args["type"]== 'NONE' : return #do nothing (no shortcut)
    key_config = bpy.context.window_manager.keyconfigs.addon
    if key_config:
        key_map = key_config.keymaps.new(name = "Object Mode")
        hotkey = key_map.keymap_items.new(**args) #invoked command: args["idname"]
        hotkey.properties.name = VIEW3D_MT_Boolean.bl_idname #pie menu to open
        addon_keymaps.append((key_map,hotkey))
        if DEBUG: print("Keyboard shortcut set to: " + ("[Shift]-" if args["shift"] else "")
                    + ("[Ctrl]-" if args["ctrl"] else "") + ("[Alt]-" if args["alt"] else "")
                    + ("[%s]" % args["type"]) + (" (from add-on preferences)" if prf else ""))

def unregister_keymap():
    '''Removes current hotkey (if any)'''
    key_config = bpy.context.window_manager.keyconfigs.addon
    if key_config:
        for key_map, hotkey in addon_keymaps:
            key_map.keymap_items.remove(hotkey)
    addon_keymaps.clear()
```



Ciąg dalszy na następnej stronie…

Rysunek 6.6.4 Skrypt *obect_booleans.py*, cz. 4 (obsługa preferencji wtyczki – skrótów klawiatury)

Copyright Witold Jaworski, 2011-2019.

Wreszcie ostatnia część skryptu: rejestracja klas API i dodanie wywołania operatora do menu (Rysunek 6.6.5):

```
#---------- ### Register -----------
from bpy.utils import register_class, unregister_class

def menu_draw(self, context):
    self.layout.operator_context = 'INVOKE_REGION_WIN'
    self.layout.operator_menu_enum(OBJECT_OT_Boolean.bl_idname, property="op")

#list of the classes in this add-on to be registered in Blender API:
classes = [
            OBJECT_OT_Boolean,
            VIEW3D_MT_Boolean,
            Preferences,
          ]

def register():
    for cls in classes:
        register_class(cls)
    bpy.types.VIEW3D_MT_object.prepend(menu_draw)
    register_keymap()
    if DEBUG: print(__name__ + ": registered")

def unregister():
    unregister_keymap()
    bpy.types.VIEW3D_MT_object.remove(menu_draw)
    for cls in classes:
        unregister_class(cls)
    if DEBUG: print(__name__ + ": UNregistered")


#---------- ### Main code -----------
if __name__ == '__main__':
    register()
```

Rysunek 6.6.5 Skrypt *obect_booleans.py*, cz. 5 (rejestracja klas API oraz menu rozwijalnego)

Bibliografia

Książki

[1] Thomas Larsson, *Code snippets.Introduction to Python scripting for Blender 2.5x*, free e-book, 2010.

[2] Guido van Rossum, *Python Tutorial*, (part of Python electronic documentation), 2011

Internet

[1] http://www.blender.org

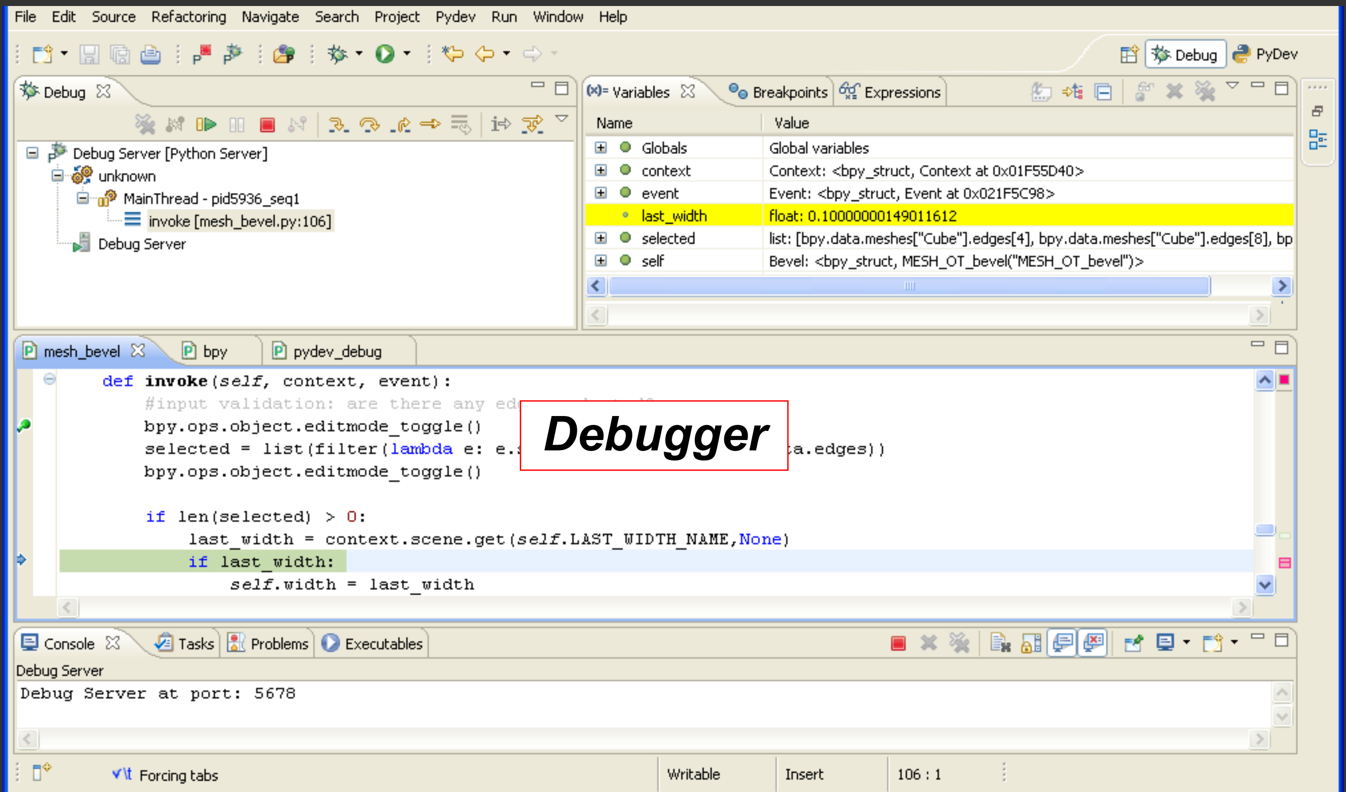
[2] http://www.python.org

[3] http://www.eclipse.org

[4] http://www.pydev.org

[5] http://wiki.blender.org

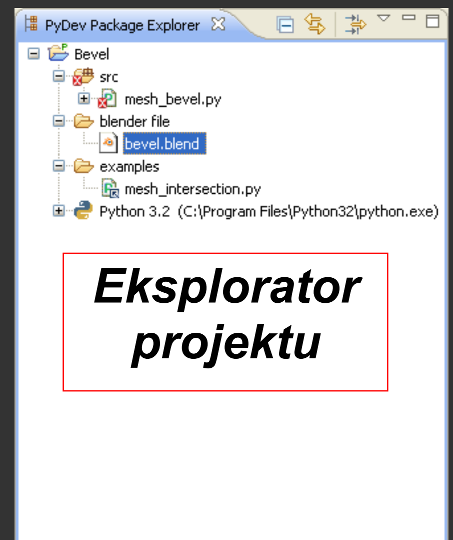
Copyright Witold Jaworski, 2011-2019.



Jeżeli masz już pewne doświadczenie w programowaniu, i zamierzasz napisać jakiś dodatek do programu Blender 3D, to ta książka jest dla Ciebie!

Pokazuję w niej, jak zestawić wygodne środowisko do pisania skryptów Blendera. Wykorzystuję do tego oprogramowanie Open Source: pakiet Eclipse, rozbudowany o wtyczkę PyDev. To dobra kombinacja, udostępniająca użytkownikowi wszystkie narzędzia, pokazane na ilustracjach wokół tego tekstu.

Książka zawiera także praktyczne wprowadzenie do API Blendera. Tworzę w niej od podstaw wtyczkę z nowym poleceniem programu. Omawiam szczegółowo każdą fazę implementacji. Pokazuję w ten sposób nie tylko same narzędzia, ale także metody, którymi się posługuję. Ten opis pozwoli Ci nabrać wprawy, potrzebnej do samodzielnej pracy nad kolejnymi skryptami.



ISBN: 978-83-941952-0-5

Bezpłatna publikacja elektroniczna