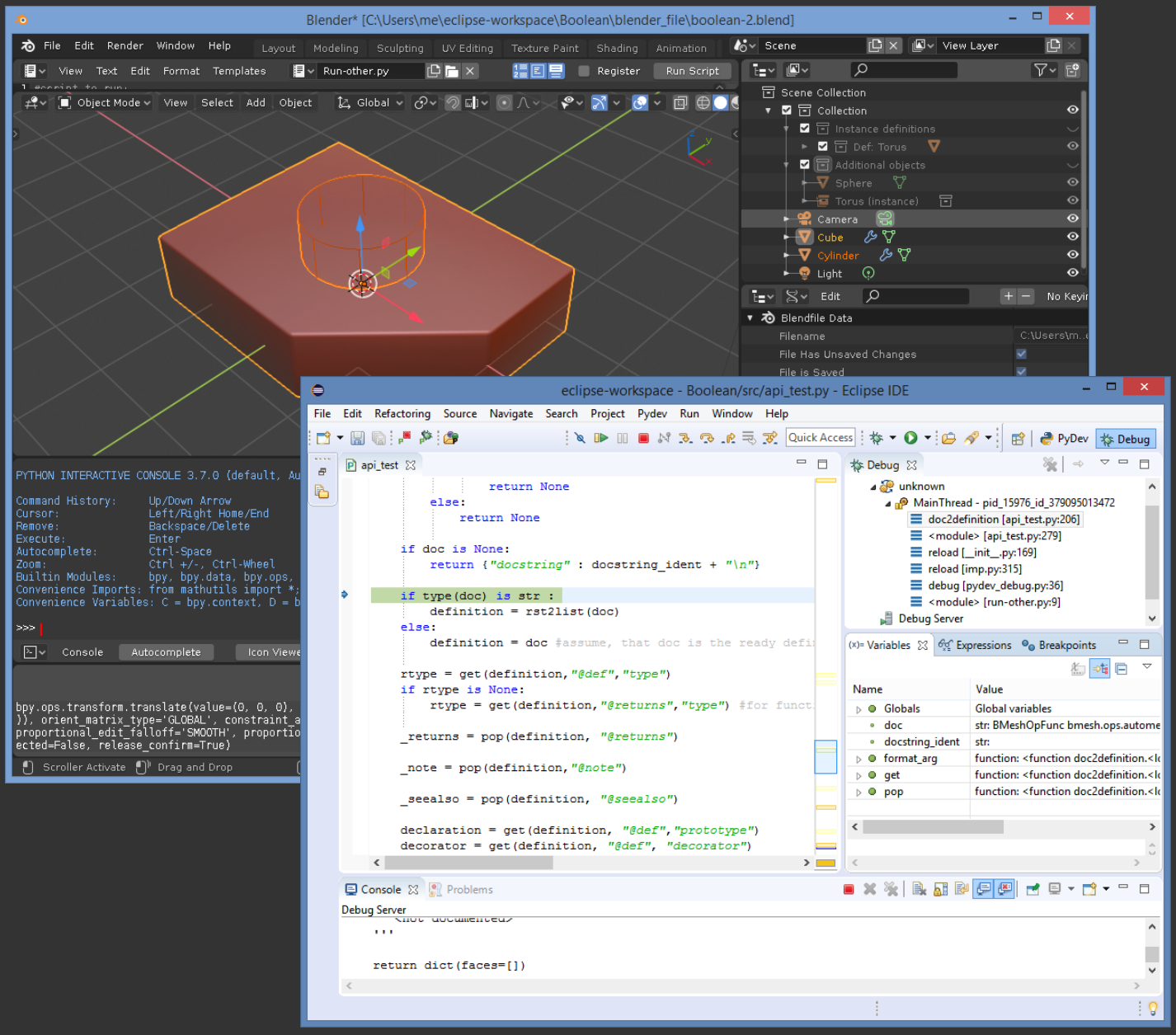
Witold Jaworski



Programming Add-Ons for Blender 2.8

Writing Python Scripts with Eclipse IDE

version 2.0

Programming Add-Ons for Blender 2.8 - version 2.0

wjaworski@airplanes3d.net
http://www.airplanes3d.net

ISBN: 978-83-941952-1-2

Table of Contents

Table of Contents 3
Introduction 4
Conventions 5
Preparations 6
Chapter 1. Software Installation 7
1.1 Python (standalone interpreter) 8
1.2 Eclipse 11
1.3 PyDev 14
Chapter 2. Introduction to Eclipse 18
2.1 Creating a new project 19
2.2 Writing the simplest script 24
2.3 Debugging 29
Creating the Blender Add-On 33
Chapter 3. Basic Python Script 34
3.1 Problem formulation 35
3.2 Adapting Eclipse to Blender API 39
3.3 Developing the core code 47
3.4 Launching and debugging Blender scripts 53
3.5 Improving our script 61
3.6 Handling the runtime errors and user messages 71
Chapter 4. Converting API Script into Blender Add-On 77
4.1 Adaptation of the script structure 78
4.2 Adding operator command to a Blender menu 88
4.3 Dynamic interaction with the user 95
4.4 Keyboard shortcut and a pie menu 99
4.5 Implementation of the add-on preferences panel 106
Appendices 114
Chapter 5. Installation Details 115
5.1 Details of Python installation 116
5.2 Details of Java Runtime Environment (JRE) installation 120
5.3 Details of Eclipse and PyDev installations 122
5.4 Details of the PyDev configuration 129
5.5 Managing Eclipse project perspectives 133
5.6 Configuring the running and debugging commands for standalone Python scripts 134
Chapter 6. Others 138
6.1 Updating Blender API predefinition files 139
6.2 Enabling Blender API code autocompletion in a PyDev project 142
6.3 Importing/linking an existing file to a PyDev project 145
6.4 Details of debugging Blender scripts 149
6.5 What does contain the *pydev_debug.py* module? 160
6.6 The full code of the *object_booleans.py* add-on 162
Bibliography 167

Introduction

You can use Python scripts for extending the standard Blender function set with new commands. Many useful add-ons were created this way. Unfortunately, Blender lacks an integrated development environment ("IDE") for the script programmers. In its *Scripting* workspace you will find only *Text Editor*, which highlights the Python syntax, and *Python Console*. This basic set is enough for developing simple scripts but lacks many tools which you need for a bigger project. I especially missed a decent debugger.

In 2007, I wrote an article that proposed for this purpose two Open Source programs: **SPE** (the editor) and **Winpdb** (the debugger). However, this solution soon became obsolete. In 2009 it was decided that the new, rewritten "from the scratch" Blender version (2.5) will have a completely new Python API. What's more, developers have embedded in this program the new Python release (3.x), while previous Blender version (2.4) used the older Python 2.x. In the Python 3.0 the backward compatibility with the Python 2.x was broken. In the same time the SPE editor was abandoned by its author. (It happens to smaller Open Source projects – they are often hobbyist enterprises). Thus, in Blender 2.5 we were again restricted to the standard tools.

In 2011 I proposed a new developer environment, based on another Open Source software. This time my choice fell on the **Eclipse** IDE, enriched with the **PyDev** plugin. In that time both products had been developed for 10 years. Unlike SPE and Winpdb, these new tools are written in Java, thus they do not depend on a specific Python version. This was a better choice: actually (in 2019) **Eclipse** and **PyDev** are still alive, and my guide (“Programming Add-Ons for Blender 2.5”) was useful for 7 years. It was the “version 1.0” of this book. Adaptations for Blender 2.6 and 2.7 involved so few minor changes, that instead of updating the original PDF publication (which would require a new ISBN number) I just listed them in the errata on this project page.

In 2018 Blender Foundation published the “beta” release of a new Blender version: 2.8. If it was a commercial product, I am sure that they would assign this version a more significant number, for example “3.0”. Comparing to the previous releases, this new Blender contains many significant improvements and new features. Its developers also decided to discard many old functionalities, breaking the backward compatibility of the data (**.blend*) files. There are also some changes in the Python API, so the add-on compatibility was also broken. Thus, the time has come to write a new edition of this guide (“Programming Add-Ons for Blender 2.8”). Consequently, this is the “version 2.0”.

I think that the best way to present a tool is to show it at work. In this guide I am describing creation of a new Blender command that performs the Boolean operations (union, difference, intersection) on solids. (They are often used in creation of various machine parts). This book requires an average knowledge of Python and Blender. (Yet, you may know nothing about Python in Blender). To understand the part about creating the final add-on (Chapter 4) you should also be familiar with the basic concepts of object-oriented programming such as "class", "object", "instance", "inheritance". When it is needed (as at the end of Chapter 4), I am also explaining some more advanced concepts (like the "interface" or "abstract class"). This book introduces you to the practical writing of Blender extensions. I am not describing here all the issues, just presenting the `method` that you can use to learn them. Using it, you can independently master the rest of the Blender API (for example, creating your own panels or sophisticated menus).

Conventions

In the tips about the keyboard and the mouse I have assumed, that you have a standard:

- US keyboard, with 102 keys;
- Three-button mouse (in fact: two buttons and the wheel in the middle. When you click the mouse wheel, it acts like the third button).

Command invocation will be marked as follows:

Menu→Command means invoking a command named *Command* from a menu named *Menu*. More arrows may appear, when the menus are nested!

Panel:Button means pressing a button named *Button* in a dialog window or a panel named *Panel*.

Pressing a key on the keyboard:

Alt-K the dash (“-“) between characters means that both keys should be simultaneously pressed on the keyboard. In this example, holding down the Alt key, press the K key;

G, X the coma (“,”) between characters means, that keys are pressed (and released!) one after another. In this example type G first, then X (as if you would like to write „gx”).

Pressing the mouse buttons:

LMB left mouse button

RMB right mouse button

MMB middle mouse button (mouse wheel **pressed**)

Last, but not least — the formal question: how should I address you? Typically, the impersonal form ("something is done") is used in most manuals. I think that it makes the text less comprehensible. To keep this book as readable as possible, I address the reader in the second person ("do it"). Sometimes I also use the first person ("I've done it", "we do it"). It is easier for me to describe my methods of work this way[1].

[1] While coding and debugging I thought about us - you, dear Reader, and me, writing these words - as a single team. Maybe an imaginary one, but somehow true. At least, writing this book I knew that I had to explain you all details of its topics!

Preparations

In this section, I am describing how to build (install) the required software (Chapter 1). Then I am introducing the basics of the Eclipse IDE and its PyDev plugin (Chapter 2).

Chapter 1. Software Installation

The integrated development environment, described in this book, requires three basic components:

- standalone (“classic”) Python interpreter (required for the PyDev);
- one of the Eclipse IDE “packages”;
- PyDev (an Eclipse plugin);

This chapter describes how to set them up.

I assume that you have already installed Blender. This book was written using Blender 2.80.

- Tools described in this guide require a 64-bit operating system. (For Windows 10 this is the default).

1.1 Python (standalone interpreter)

Blender comes with its own, embedded Python interpreter. Check its version first by switching to the ***Scripting*** workspace and read the Python version number in the ***Python Console*** window. (It is written in the first line (Figure 1.1.1):

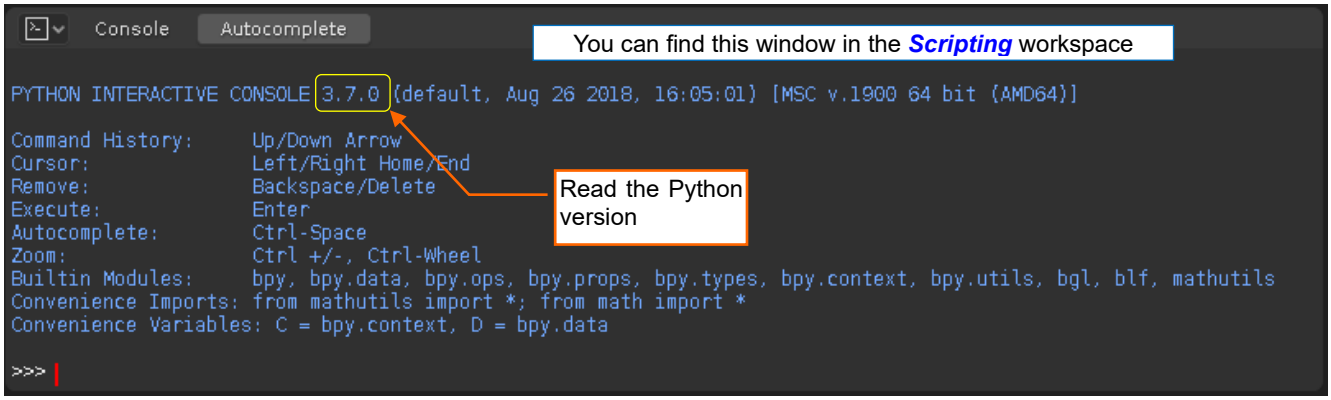


Figure 1.1.1 Reading the version number of the embedded Python interpreter.

Blender in the figure above uses Python 3.7.0 (this is Blender 2.80). In principle, you should install the same version of the standalone interpreter, but minor differences (especially in the third digit) are acceptable.

You can download the external Python interpreter from https://www.python.org/downloads/ (Figure 1.1.2):

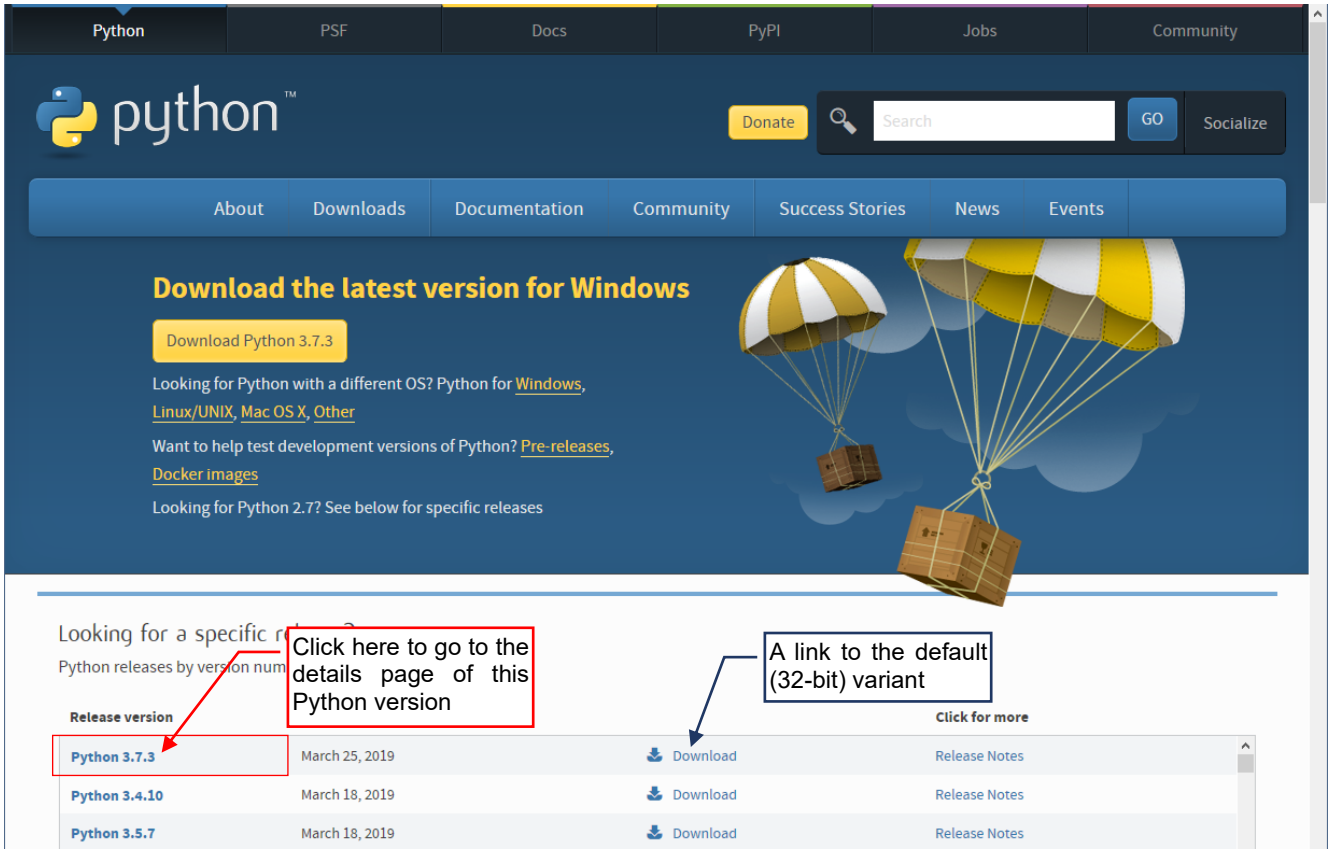


Figure 1.1.2 Selection of the Python version (as seen in May 2019)

The third digit in the Python version number is the number of its “service edition”. This kind of updates is dedicated for minor bug fixes. That’s why in Figure 1.1.2 I am choosing Python 3.7.3, because from programmer’s point of view it is identical to Python 3.7.0, used in Blender 2.80.

Since 2019 Eclipse has been released in the single, 64-bit variant. Thus, just in case, I will also install the 64-bit variant of standalone Python interpreter. That's why I opened the web page that contains all the variants of the selected Python version (***Python 3.7.3*** in Figure 1.1.2).

I scrolled down this web page, to find in its bottom lines the links to installation programs of all the Python "byte" variants (Figure 1.1.3):

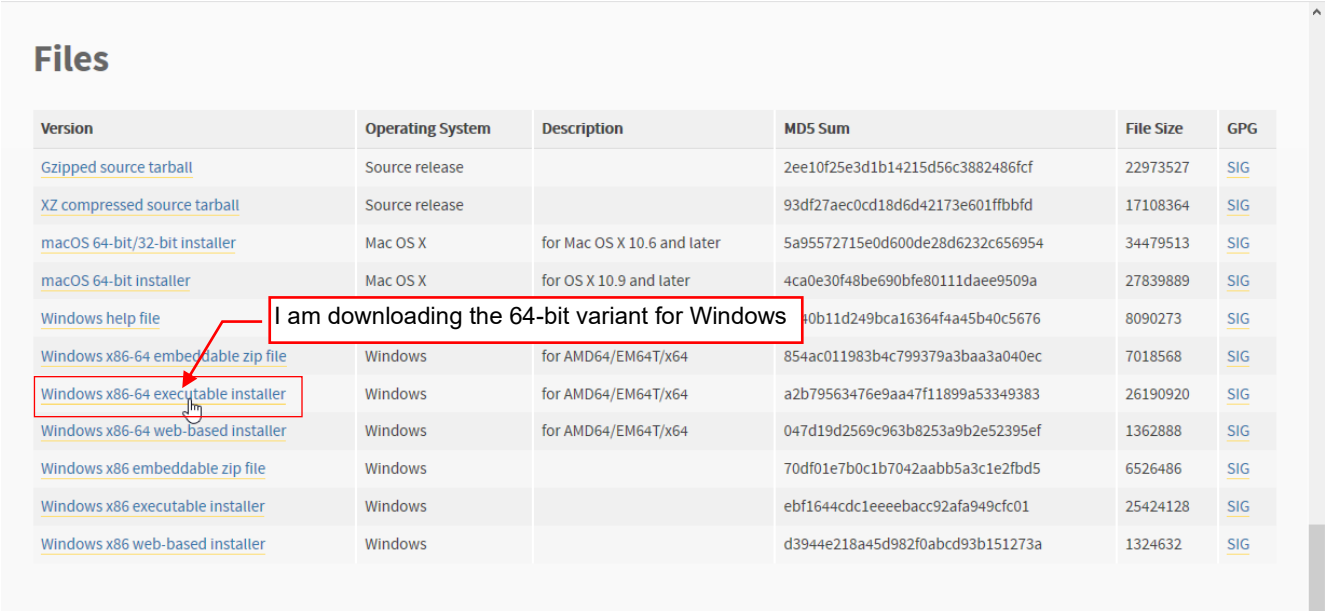


Figure 1.1.3 Downloading the 64-bit Python variant

For my computer, I downloaded the 64-bit variant for Windows.

Then I run the downloaded program (this is a standard Windows installer - Figure 1.1.4):

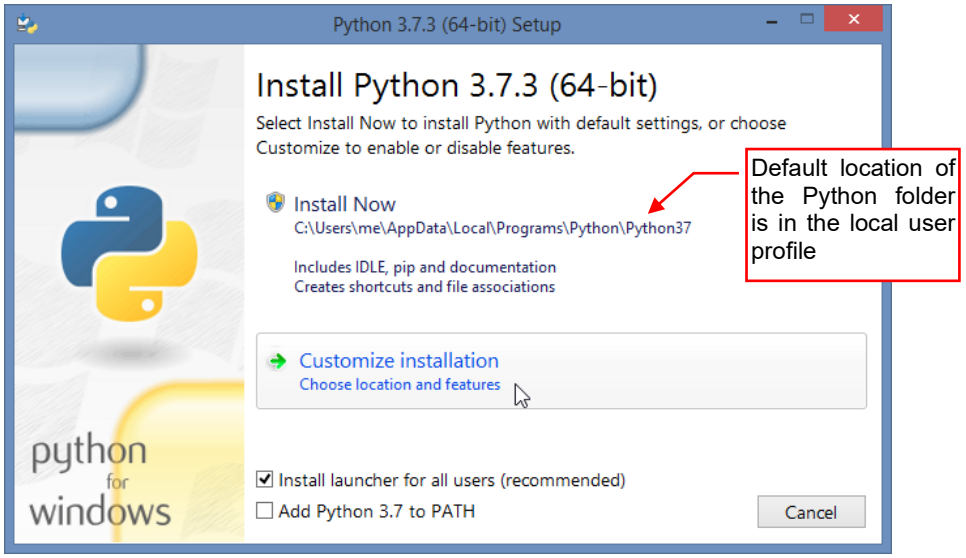


Figure 1.1.4 First screen of the Python installer

This setup program proposes creating a new folder for the Python binaries in the current user profile. You can accept these settings. However, personally I do not like placing executable files in a subfolder of the *C:\Users\<user>\AppData\Local* directory, because sometimes I need to search the application folders, and I have difficulties to find them there. Fortunately, I have the Administrator rights to my PC, so I could to select in the ***Customize Installation*** the "Install for all users" option. It creates Python folder in the *C:\Program Files*.

For details of this installation – see section 5.1 (page 116).

- Before downloading the external Python interpreter, you can also check if you already have it on your computer. Try to invoke in the command line following program:
  `python --version`
  If a Python interpreter is present on your computer, it will launch the console, as in Figure 1.1.1. You can read its version number from there.

- It may happen that you will not find on www.python.org exactly the same Python version that is embedded in your Bender (I mean the difference in the second digit of the Python version number). In such a case use the newer version with the closest number. It will spoil nothing. Blender always uses its embedded interpreter, even when the standalone Python is available in your system. For example, if you use version 3.8 of Python as the external interpreter, there should be no problem in writing scripts that are interpreted internally in Blender by its embedded Python in version 3.7. (In practice, differences between minor Python versions are not significant as long as you do not use the few new functions/extensions introduced in each version).

1.2 Eclipse

- Eclipse is a Java application, which uses the standard Java Runtime Environment (***JRE***). Since 2019 it requires the 64-bit JRE variant. You can download it from www.java.com

(To learn more about the JRE installation details – see section 5.2, page 120).

Open www.eclipse.org/downloads and download the Eclipse installer (Figure 1.2.1):

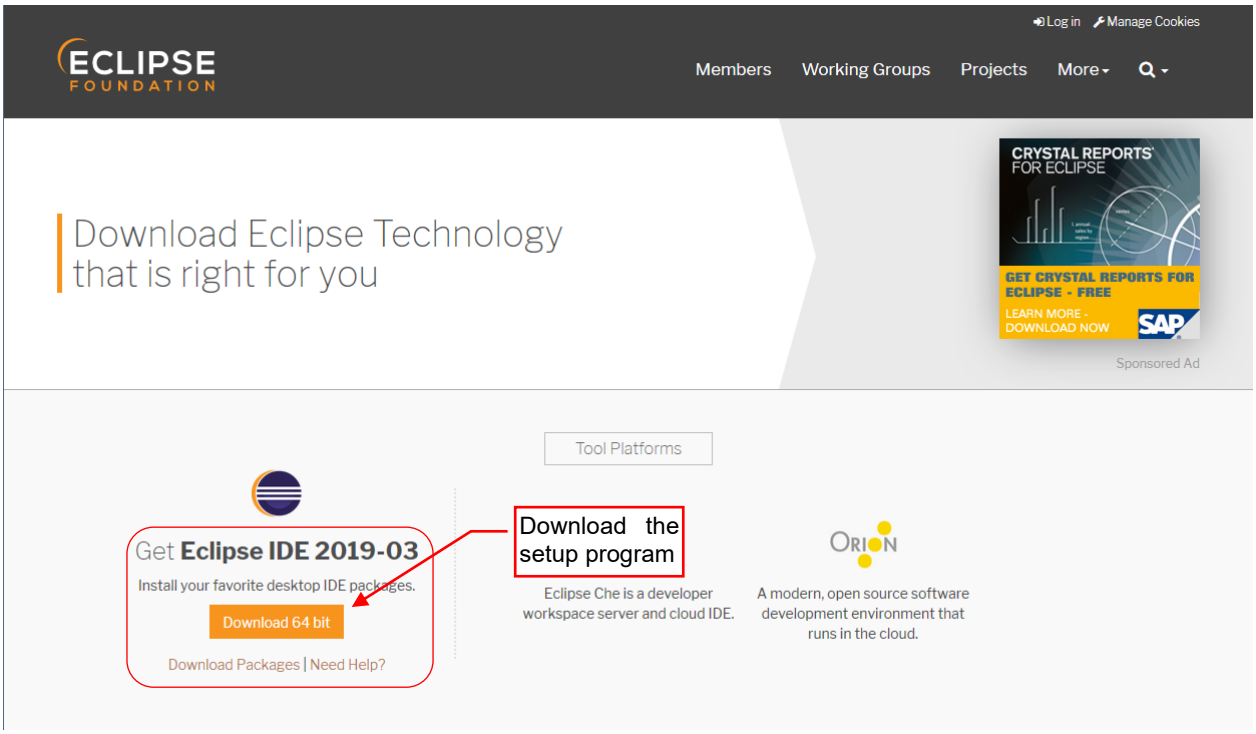


Figure 1.2.1 Downloading an Eclipse package (as seen in May 2019)

When you run the downloaded program, it displays the Eclipse variants (packages) (Figure 1.2.2):

Figure 1.2.2 Selection of the Eclipse variant

Eclipse is available in many different packages. Each of them is prepared for a specific programming language (or languages). However, these packages are not fixed: you can still write a C ++ program in, let's say, "Eclipse for PHP Developers". Just add appropriate plugins for the C/C++ IDE! What you can see in Figure 1.2.2 are just the most common packages (plugin sets). There is not any special "Eclipse for Python" package, so I suggest choosing one of the packages with the least number of specific plugins: ***Eclipse for Testers*** or ***Eclipse IDE for C/C++***. (You can find detailed discussion of Eclipse installation on page 122).

When you click the selected item, it opens the new screen with installation options (Figure 1.2.3):

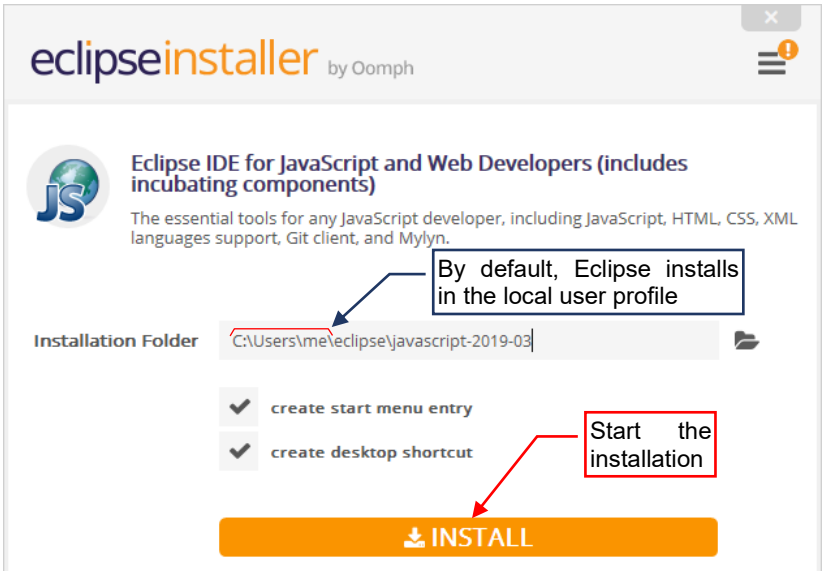


Figure 1.2.3 Eclipse setup options

By default, Eclipse folders are placed in the local user profile. (In the figure above this is *C:\Users\me* directory). The installer will create there a subfolder named *eclipse*. This directory will contain subdirectories for subsequent Eclipse versions (in Figure 1.2.3 this is *javascript-2019-03*, where "javascript" is the name of the package, and "2019-03" denotes the Eclipse version).

In one of the further chapters of this book we will search for a specific folder among the Eclipse files. (More precisely – among its plugins). That's why I am installing Eclipse in this default location: I hope that it will match the corresponding folder on your computer.

When the application is installed, check if everything works properly. On the beginning, Eclipse opens the workspace selection dialog (Figure 1.2.4):

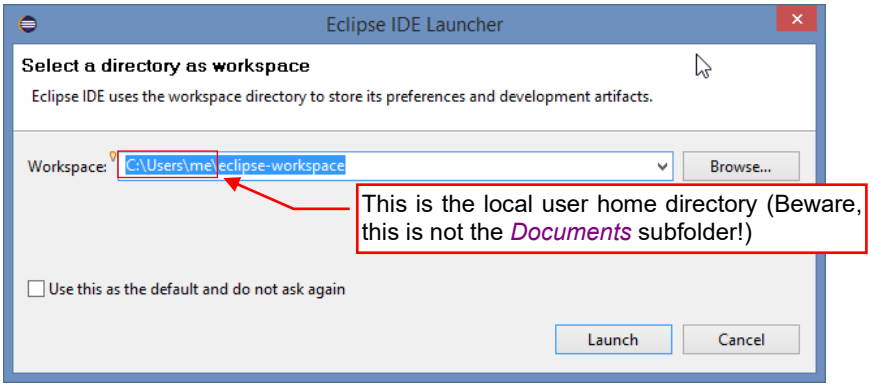


Figure 1.2.4 Selection of the projects folder

Workspace directory is the place for your projects. Each workspace contains its own set of the Eclipse preferences (including references to the standalone Python interpreter). By default, workspace folder is named *eclipse-workspace* and located in the user home directory (as the folder with the Eclipse binaries). In Figure

1.2.4 this home directory is *C:\Users\me*. Note, that this is not the standard *Documents* subfolder, but its parent directory! If you are used to keeping all the user data in *Documents* - just change this path. Usually you will need a single workspace folder for all your work. Eclipse will create in this directory subdirectories for your projects. A single project subdirectory will contain your Python scripts together with other auxiliary elements (for example – a Blender test file). You can organize them into a subfolder structure.

When you start Eclipse for the first time, it displays the *Welcome* window (Figure 1.2.5):

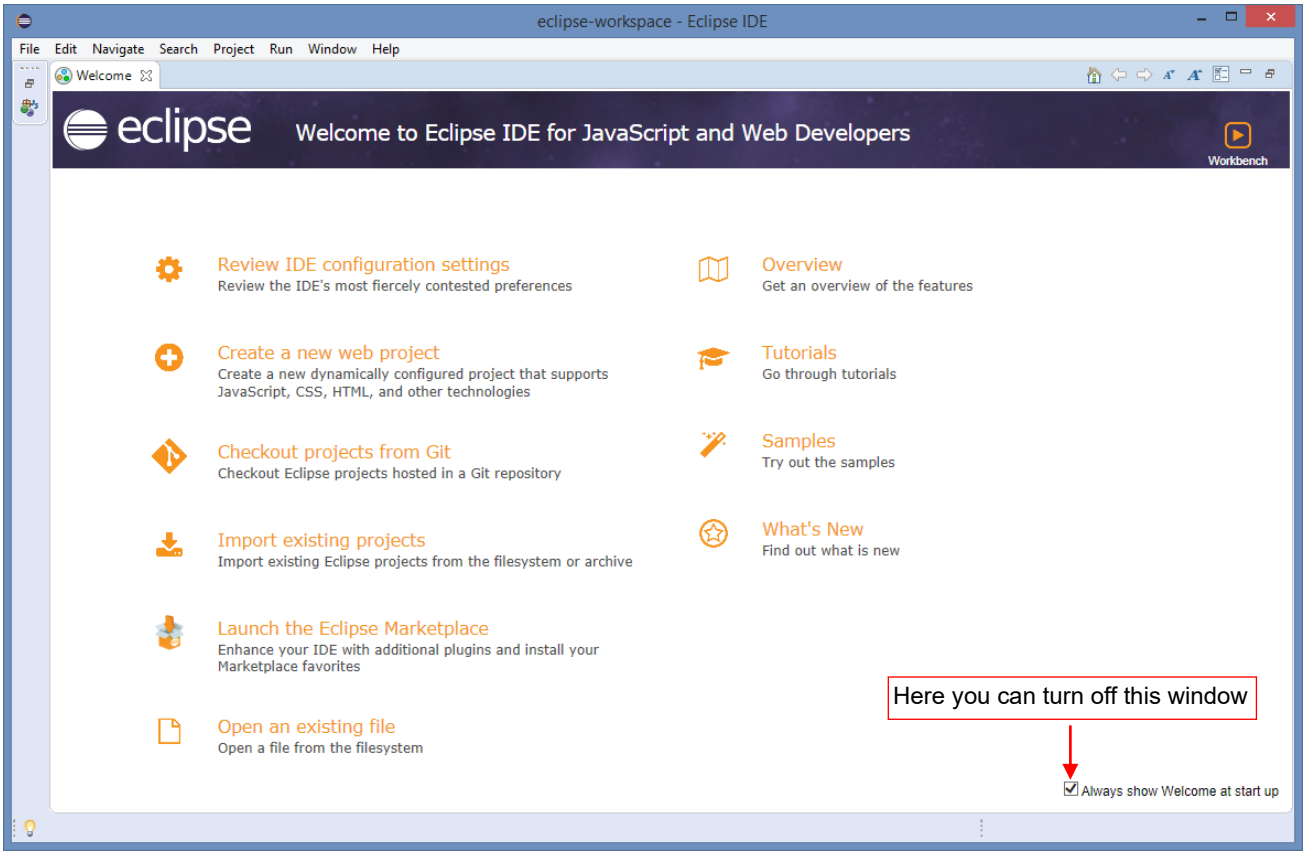


Figure 1.2.5 The *Welcome* window

Now you have to add the PyDev plugin to Eclipse. It will adapt this environment for the Python scripts.

1.3 PyDev

For PyDev installation use the internal Eclipse mechanism, designed for the plugins.

- NOTE: To perform steps described in this section, you need an Internet connection

To add a plugin, go to ***Help→Eclipse Marketplace*** (Figure 1.3.1):

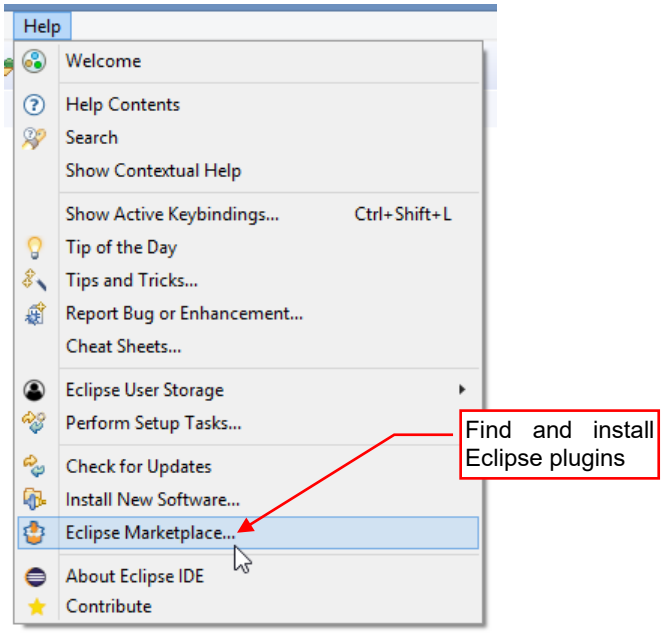


Figure 1.3.1 Opening the plugin list

In the *Eclipse Marketplace* window search for "PyDev" phrase (Figure 1.3.2):

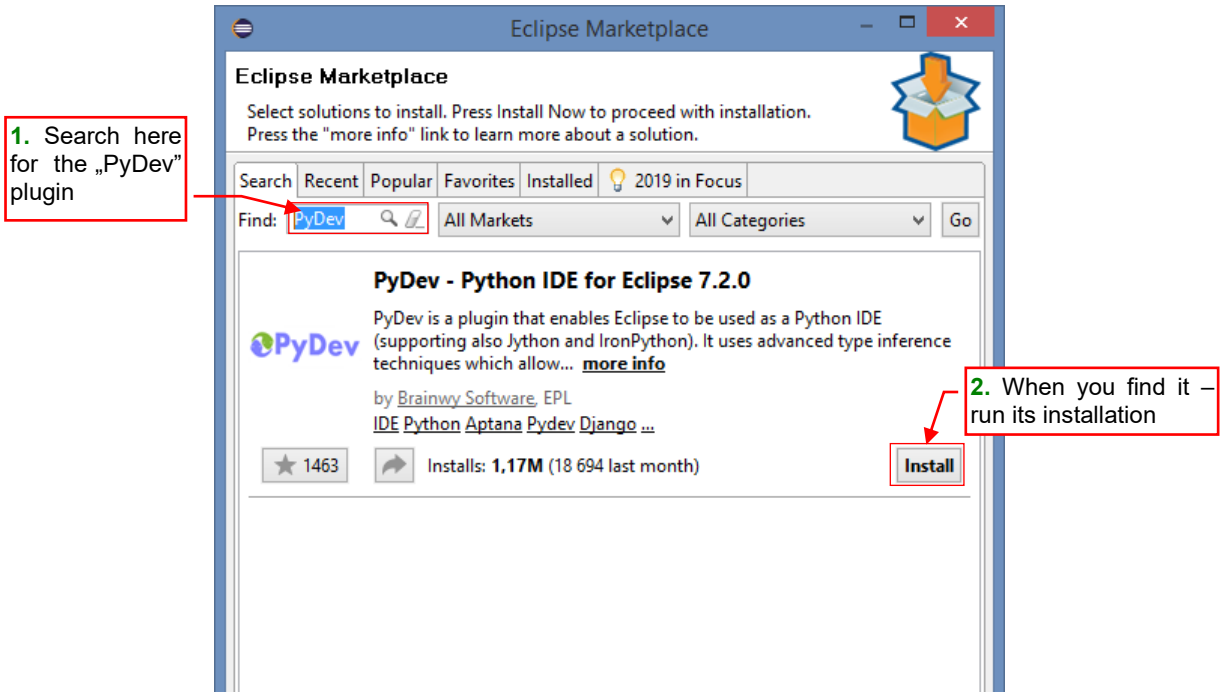


Figure 1.3.2 Searching for the PyDev plugin in the *Eclipse Marketplace* window

When you find the plugin named *PyDev – Python IDE for Eclipse*, click its ***Install*** button. Then confirm the subsequent screens: the default selection of the components, license terms.

Finally, you will see a message about restarting Eclipse (Figure 1.3.3):

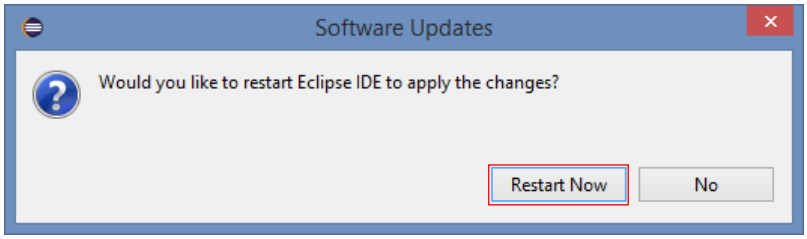


Figure 1.3.3 Final window

Do it (just in case).

Eclipse saves all of its settings in the workspace folder (see Figure 1.2.5). The reference to the default Python interpreter is also among these parameters. Let's define it now. Start by invoking the ***Window→Preferences*** command (Figure 1.3.4):

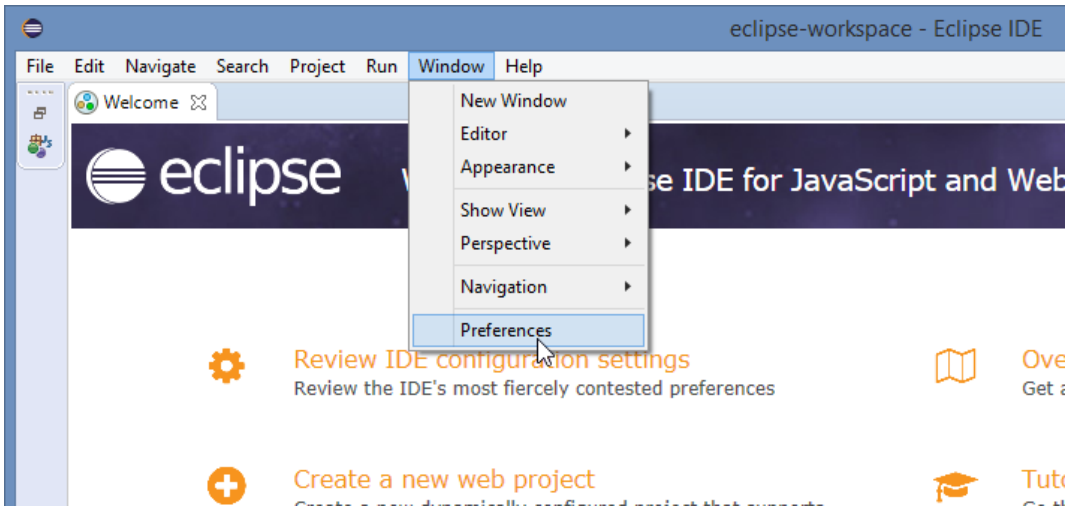


Figure 1.3.4 Setting up Eclipse configuration for the current workspace

In the *Preferences* window expand the ***PyDev*** section, then in the ***Interpreters*** section highlight the item named ***Python Interpreter*** (Figure 1.3.5):

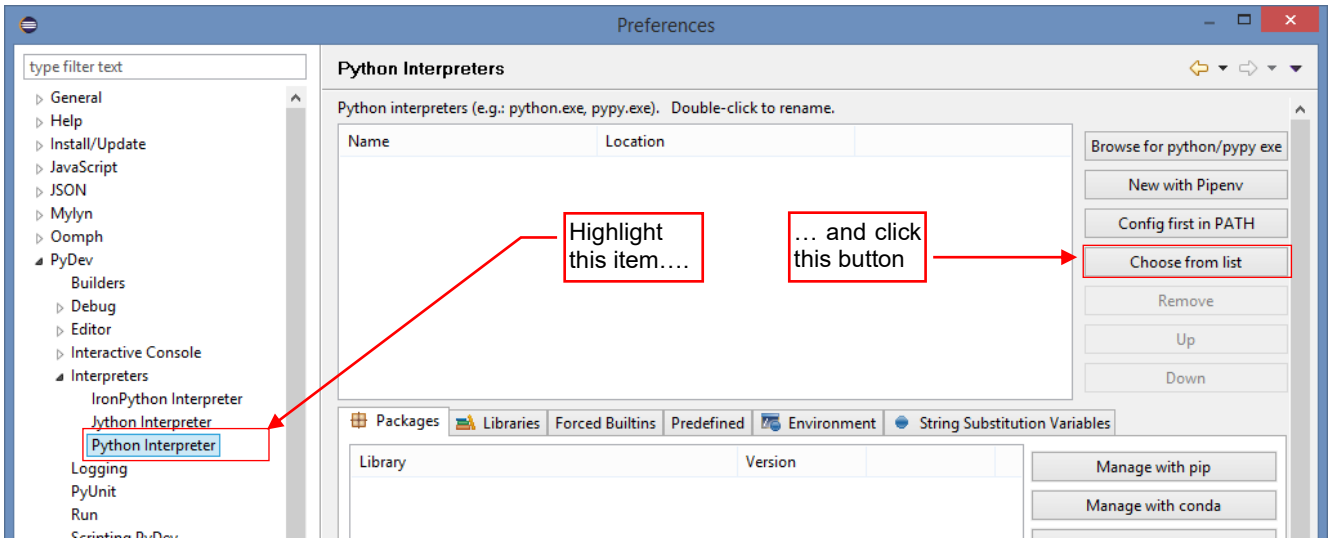


Figure 1.3.5 Invoking the searching for installed Python interpreters

Then click the ***Choose from list*** button.

After a while PyDev will display the list of installed Python interpreters (Figure 1.3.6):

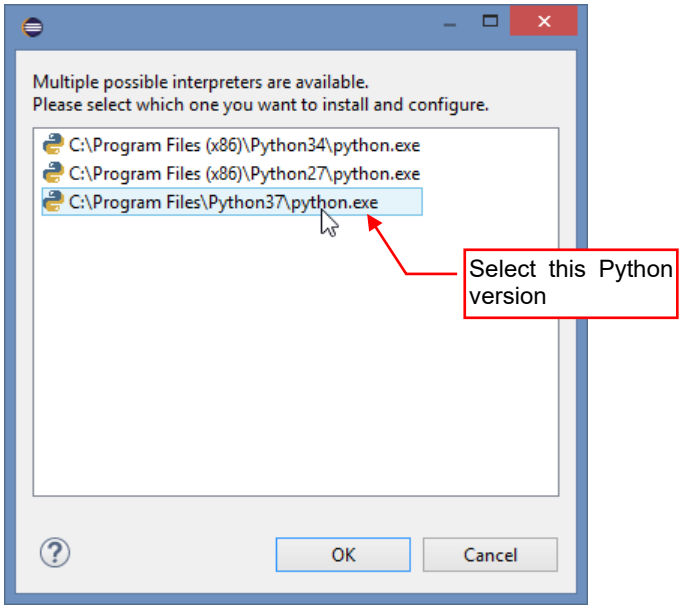


Figure 1.3.6 Python interpreter selection

Select from this list the 64-bit variant that you have installed in the previous step (see page 8). In response Python will explore its folders and suggest adding some of them to the system **PYTHONPATH** list (Figure 1.3.7):

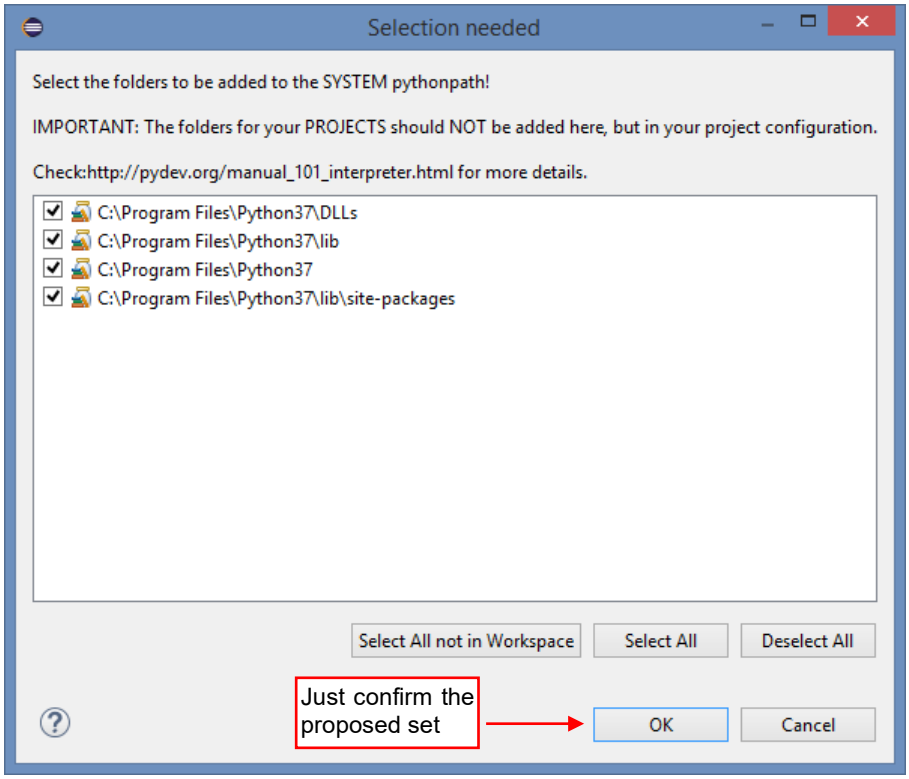


Figure 1.3.7 Python interpreter folders to be added to the PYTHONPATH

You needn't change anything here – just confirm this window clicking the ***OK*** button.

In the result you will see the configured Python interpreter in the *Preferences* window (Figure 1.3.8):

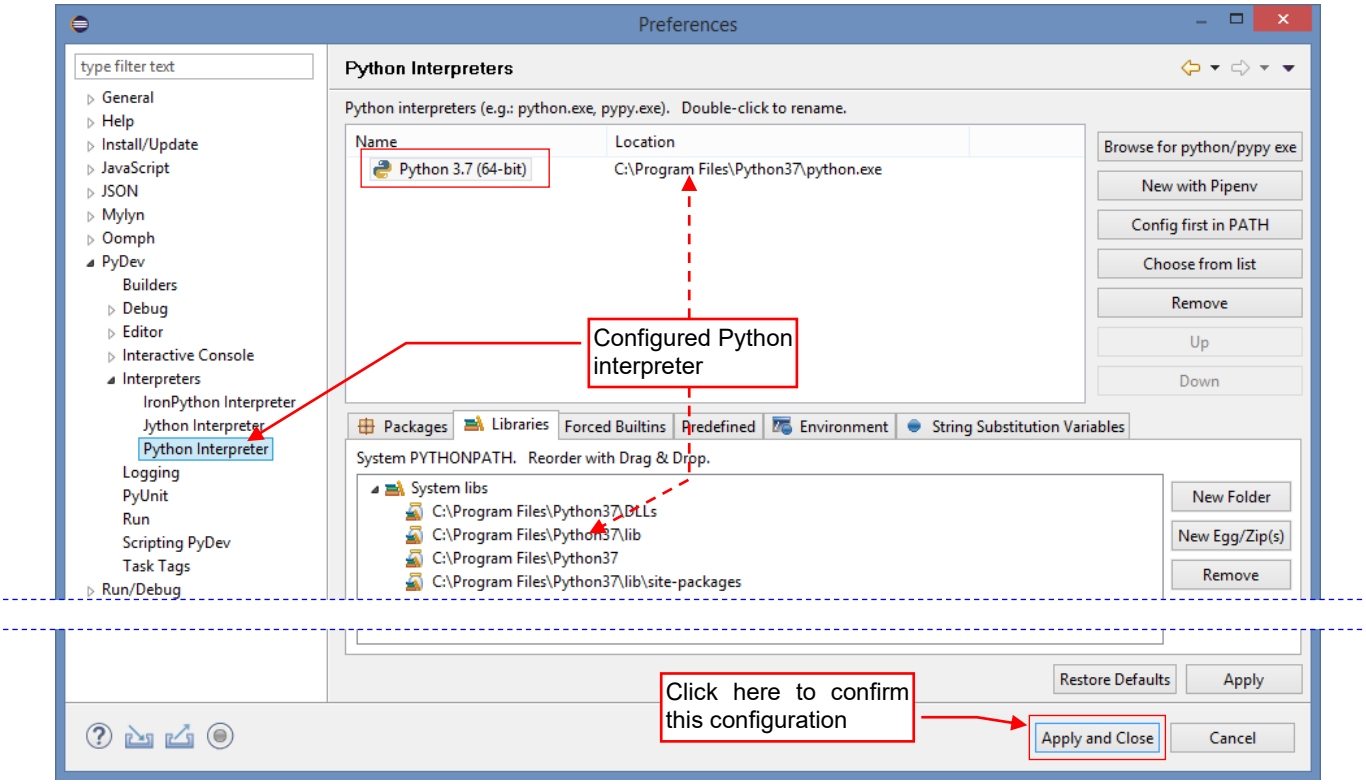


Figure 1.3.8 Configured Python interpreter

By default, PyDev names this new interpreter as “python”. It appears under this name in all PyDev windows. To make it more informative, I renamed it to “Python 3.7 (64-bit)”.

Save these settings by clicking the ***Apply and Close*** button.

To use this interpreter for running or debugging your Python scripts from Eclipse, you need so-called ***Run Configuration***. However, this is a local project setting (unlike the Python interpreter, which is set per workspace). To create a *Run Configuration* for your project, you will need the (main) script file. For details see section 2.2, page 26, and section 5.6, page 134.

Chapter 2. Introduction to Eclipse

Our project starts here. It will be an adaptation of the *Boolean* modifier. You will learn more about this in the next chapter. In this chapter, except the names, our project has nothing in common with Blender, yet.

At the beginning, I want to show you the Eclipse basics. I will do it on the example of a simple Python script, which writes "Hello" in the console window. I assume that the Reader has some experience in Python and has already used other IDEs. This is not a book about any of these issues. My goal here is to show how to perform in Eclipse some basic steps that are well known to every programmer.

2.1 Creating a new project

Invoke the ***File→New→Project…*** command (Figure 2.1.1):

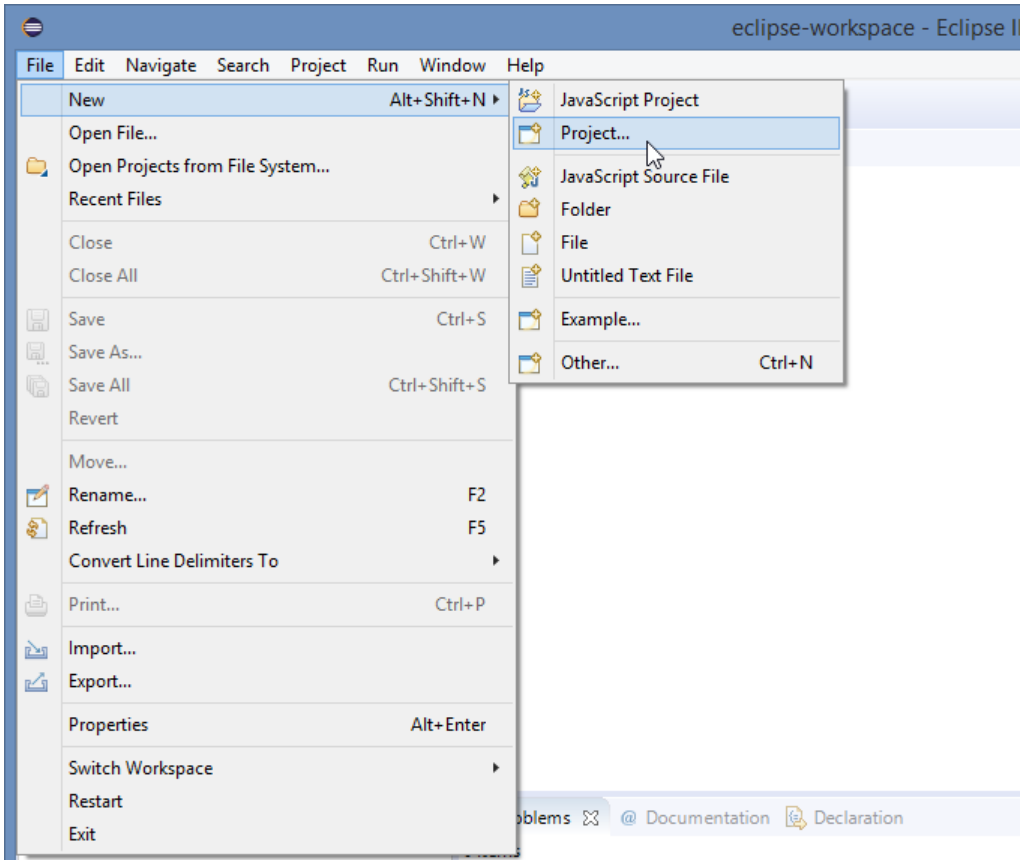


Figure 2.1.1 Opening a new project

In the *New Project* dialog open the ***PyDev*** folder and select there the ***PyDev Project*** wizard (Figure 2.1.2):

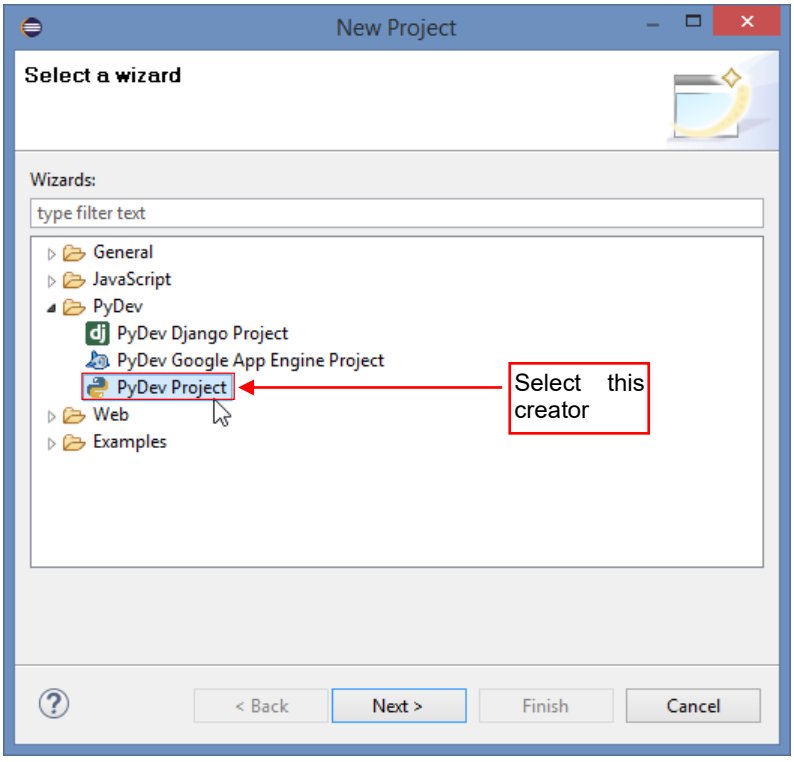


Figure 2.1.2 Selection of the appropriate project wizard

Then click the ***Next*** button.

In the *PyDev Project* window enter the ***Project name***. I am naming it **Boolean** (Figure 2.1.3), because in the next chapter it will become a Blender API project implementing the Boolean operations.

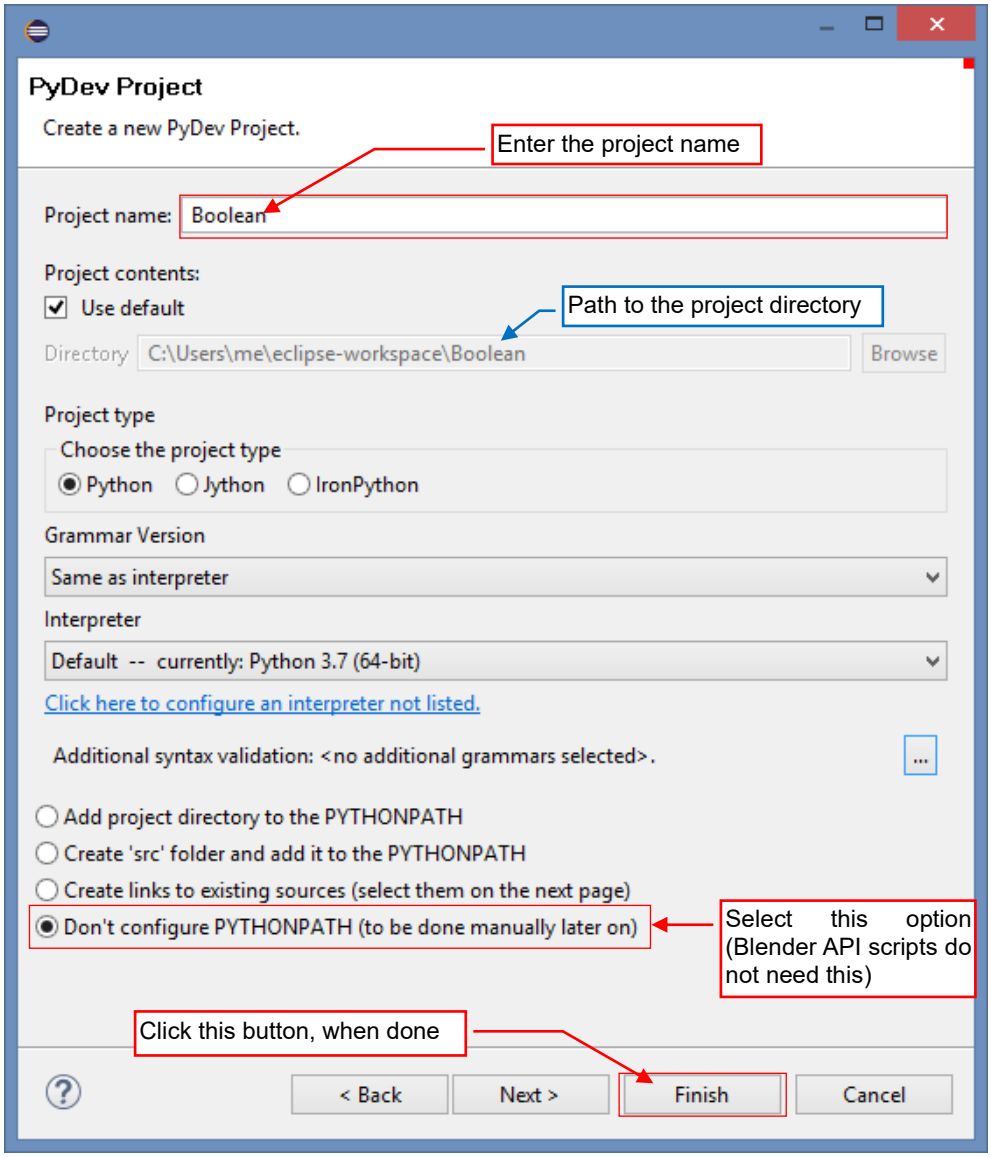


Figure 2.1.3 Filling the screen of *PyDev Project* pane

Select also the ***Don't configure PYTHONPATH...*** option, leaving the remaining settings in their default state. Click the ***Finish*** button, when done.

- PyDev grays out the *Finish* button when the Python interpreter is not yet configured (see page 15).

In response, the *New Project* creator displays a message (Figure 2.1.4):

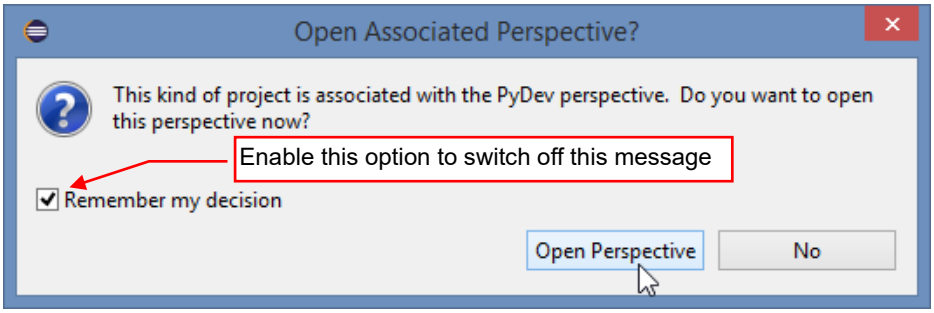


Figure 2.1.4 A question from the wizard.

Confirm it, by clicking ***Open Perspective***.

In response, PyDev creates an empty Python project (Figure 2.1.5):

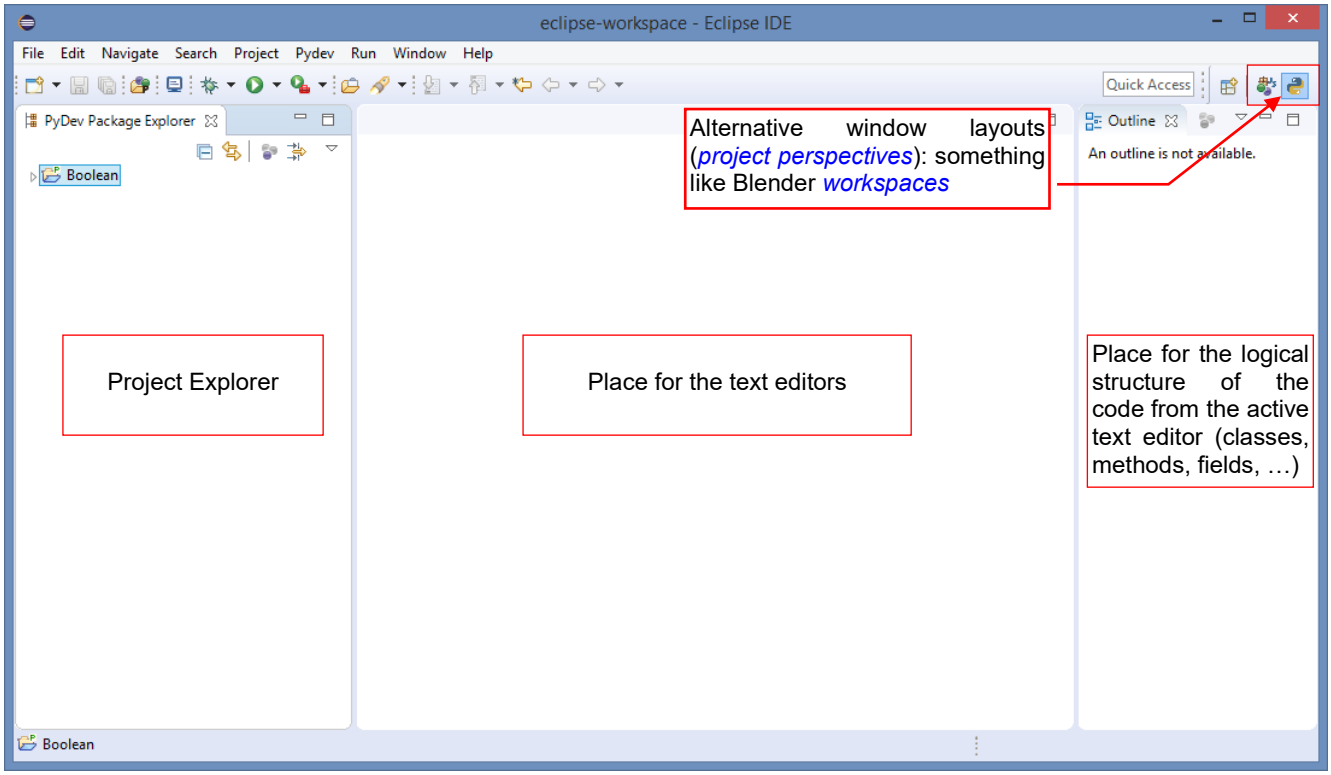


Figure 2.1.5 PyDev *perspective* of a new project

What you see in the picture above is the default PyDev screen layout. In Eclipse you can have multiple screen layouts, called *project persepctives*. The newly created project contains the default *PyDev* perspective. When you try to debug your script for the first time, the IDE will ask you about adding another perspective: *Debug*.

Let's start by creating in this project a folder for the source files. Highlight the project folder (*Boolean*), then select the ***New→Source Folder*** command from its context menu (Figure 2.1.6):

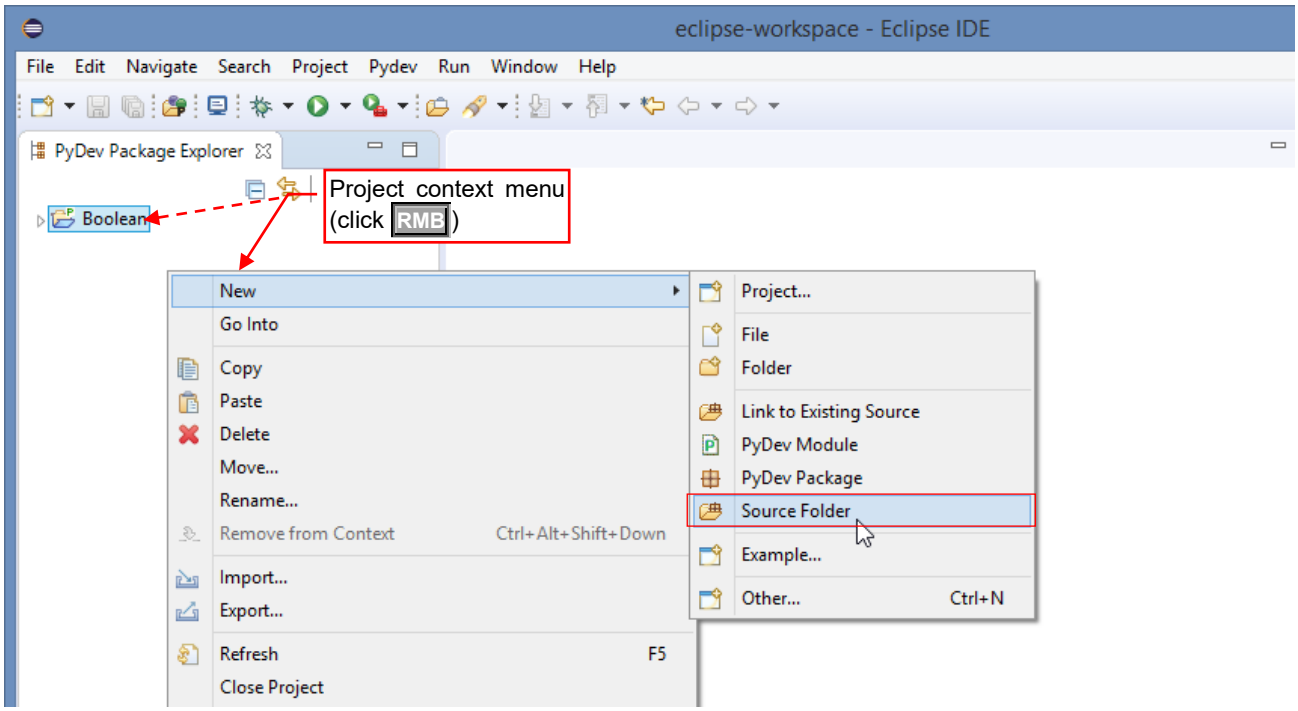


Figure 2.1.6 Adding a subfolder for the scripts

Type the subfolder ***Name*** in the wizard pane — let it be **src** (Figure 2.1.7):

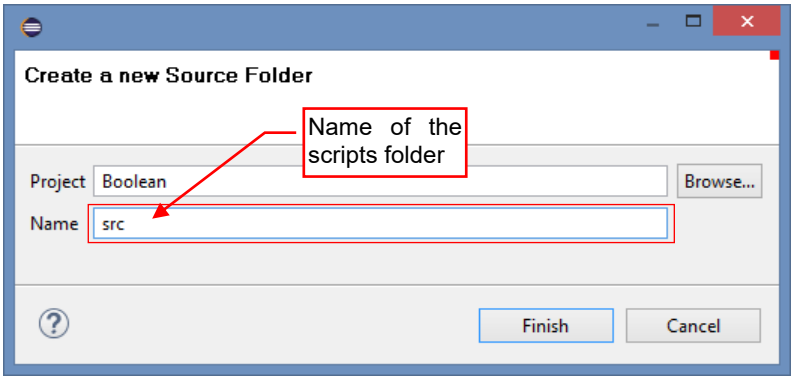


Figure 2.1.7 Folder wizard pane

When you click the ***Finish*** button, PyDev will create this project subdirectory.

Let's create now an empty script file. Expand the context menu of **src** folder and invoke the ***New→PyDev Module*** command (Figure 2.1.8):

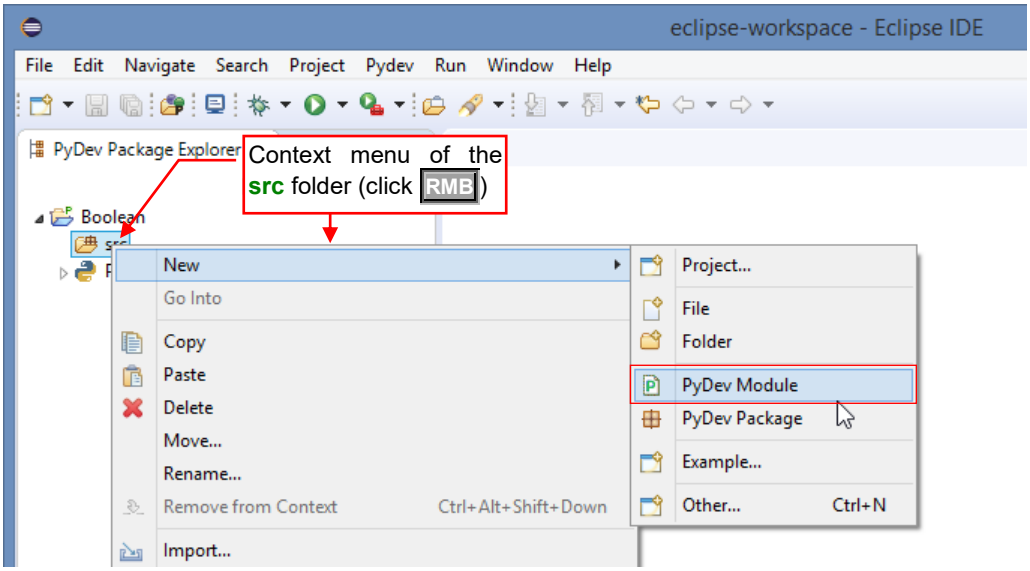


Figure 2.1.8 Invoking the new script ("module") wizard

It will open another PyDev wizard window. Give this file a name that follows the rules for the Blender add-on naming conventions: **object_booleans** (Figure 2.1.9):

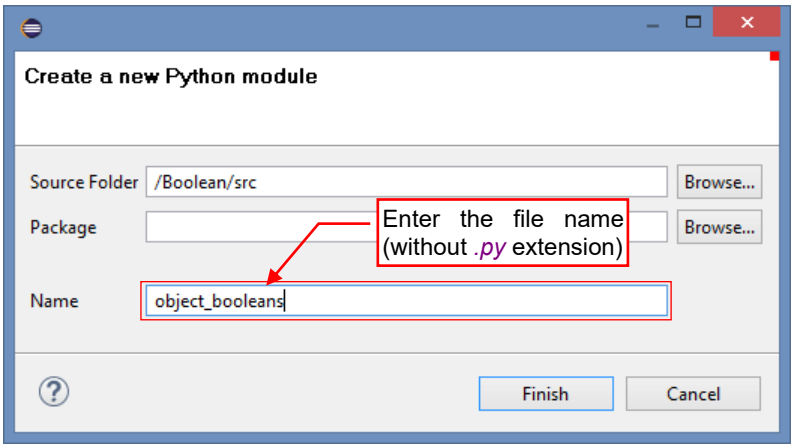


Figure 2.1.9 PyDev module wizard window

(The first part of the name – "object" - is the target Blender mode). Click ***Finish***, when done.

In the next dialog, PyDev asks about the eventual Python script template (Figure 2.1.10):

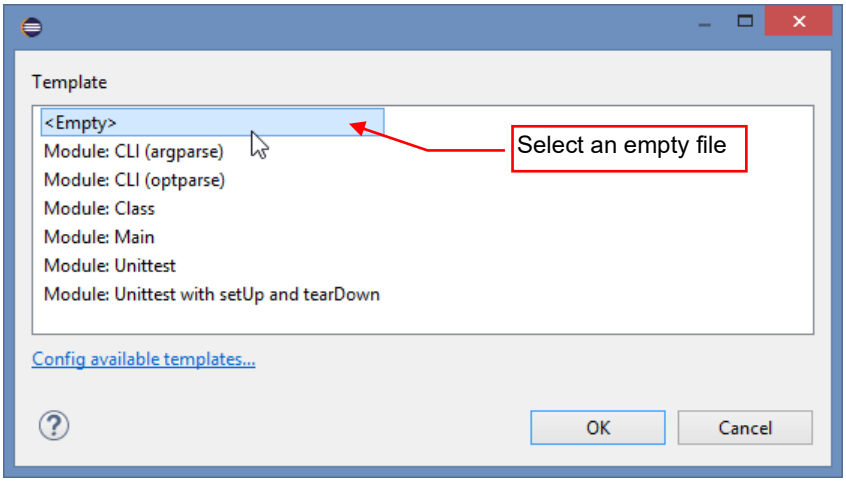


Figure 2.1.10 PyDev module wizard window (continued)

Select the ***<Empty>*** template and click ***OK***.

In response PyDev adds an empty script file to your project. It contains just a header *docstring* comment, with the creation date and the author name (Figure 2.1.11):

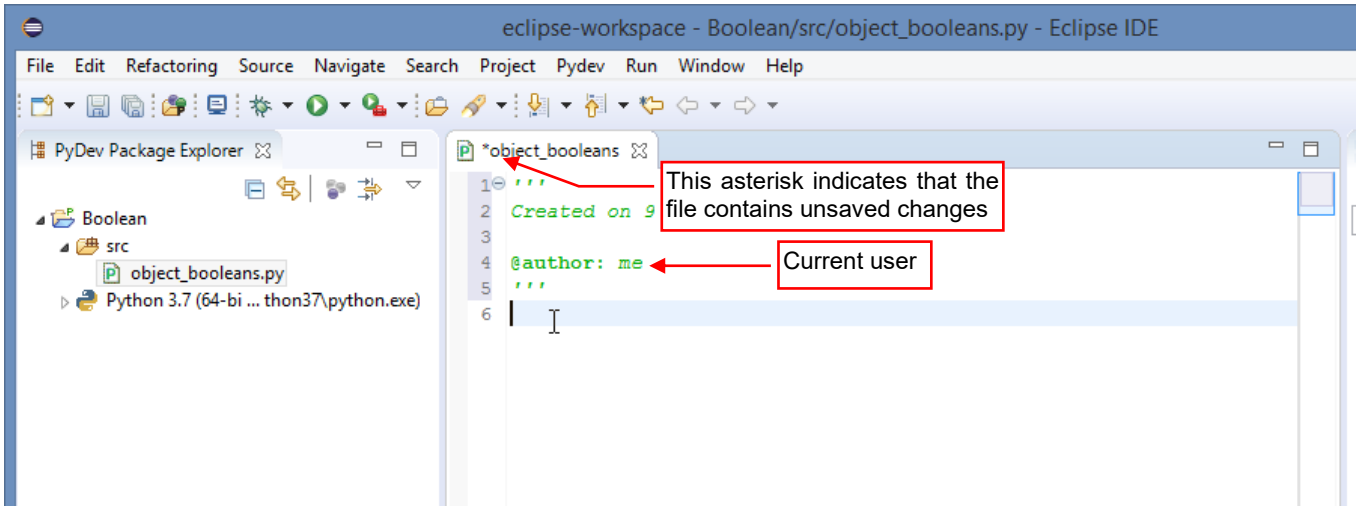


Figure 2.1.11 The new, empty script

Summary

- In this section, we have created a new Python project using the ***PyDev Project*** wizard (page 19);
- Eclipse requires a special *source folder* (page 21) for the Python scripts. (You cannot place them in the root directory);
- You can use several predefined templates for Python scripts (page 22). However, for Blender API scripts I use the *<Empty>* template;
- There are no special restrictions for the project names. In this example, I named this project "**Boolean**" (page 20), because in the next chapters it will implement the *Boolean* command for Blender 2.8. For the same reason I gave the newly created Python script file a name that follows the Blender convention for add-ons: "*object_boolean.py*" (page 22).

2.2 Writing the simplest script

The script that I will write in this section will display the "Hello" text in the Python console. To see this result, we need to add the panel with the Python console to our environment, because PyDev does not add it by default. To do this, invoke the ***Window→Show View→Console*** command (Figure 2.2.1):

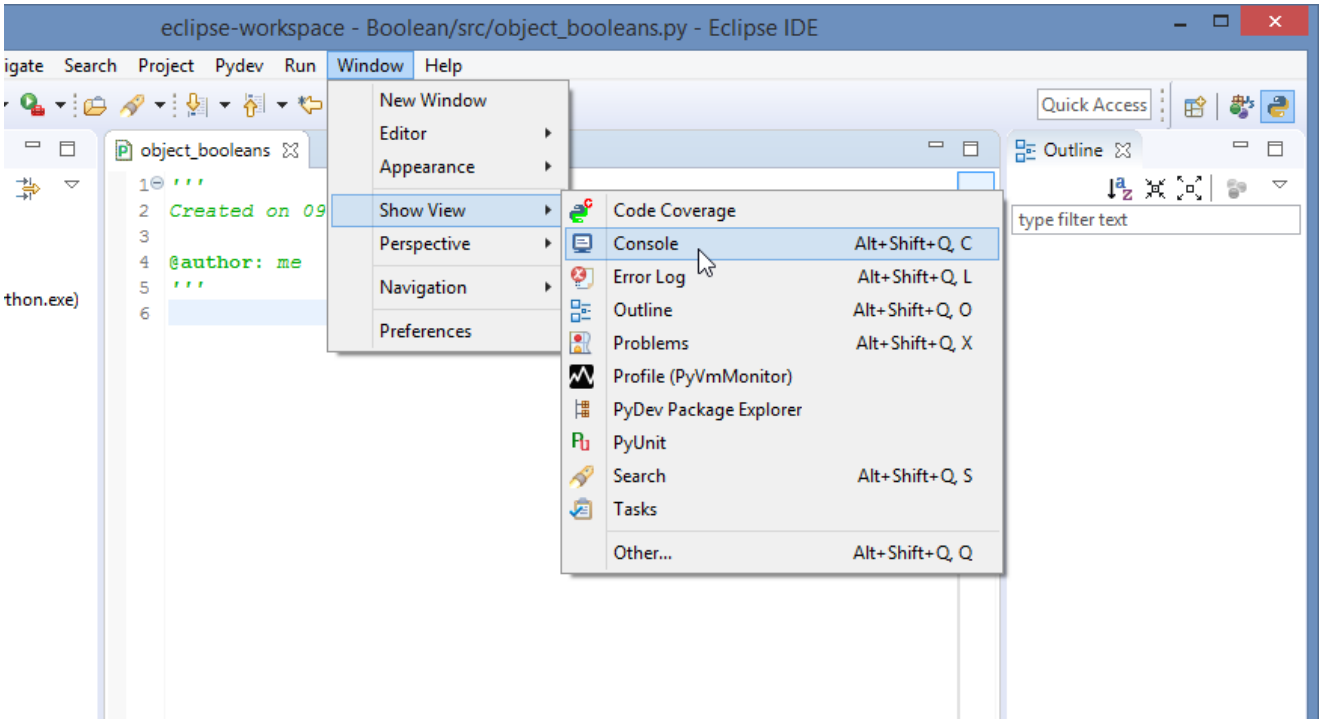


Figure 2.2.1 Adding the *Console* tab

This command adds an output console pane, which shows results of the script runs. Dynamic languages, like Python, also offer something like "interactive console". It runs the Python interactive interpreter, allowing you to check some expressions while writing the script. Let's add this gadget to the current perspective (Figure 2.2.2):

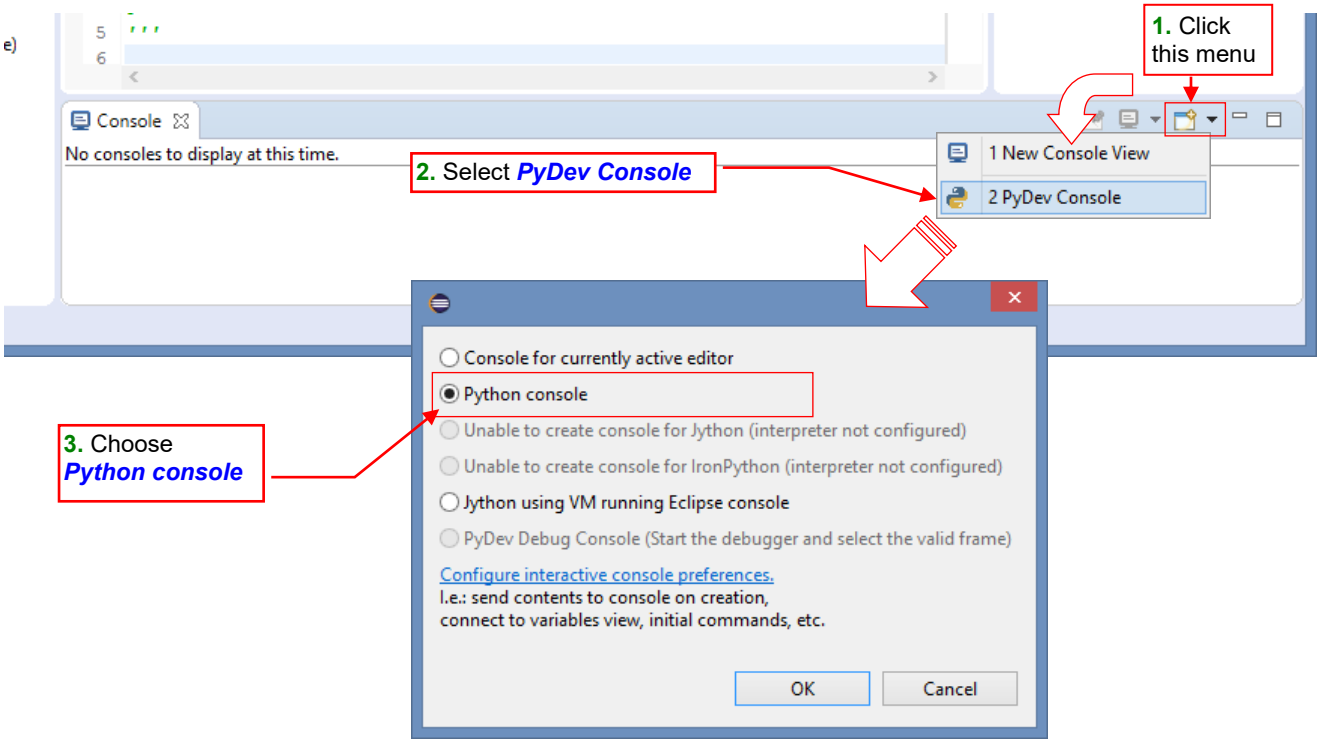


Figure 2.2.2 Switching to the interactive Python console

Invoke the ***PyDev Console*** command from the pane menu, then select the ***Python console*** option in its dialog.

So now you have the panel with the Python interpreter, where you can check your code snippets (Figure 2.2.3):

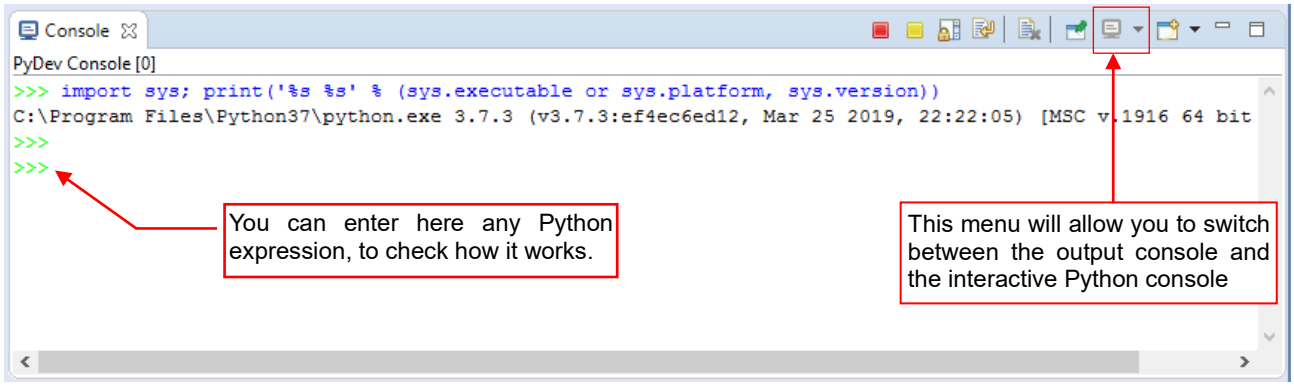


Figure 2.2.3 Interactive Python console

One of the useful PyDev features is the code autocompletion. It works in the script editor window, and also in the interactive console (Figure 2.2.4):

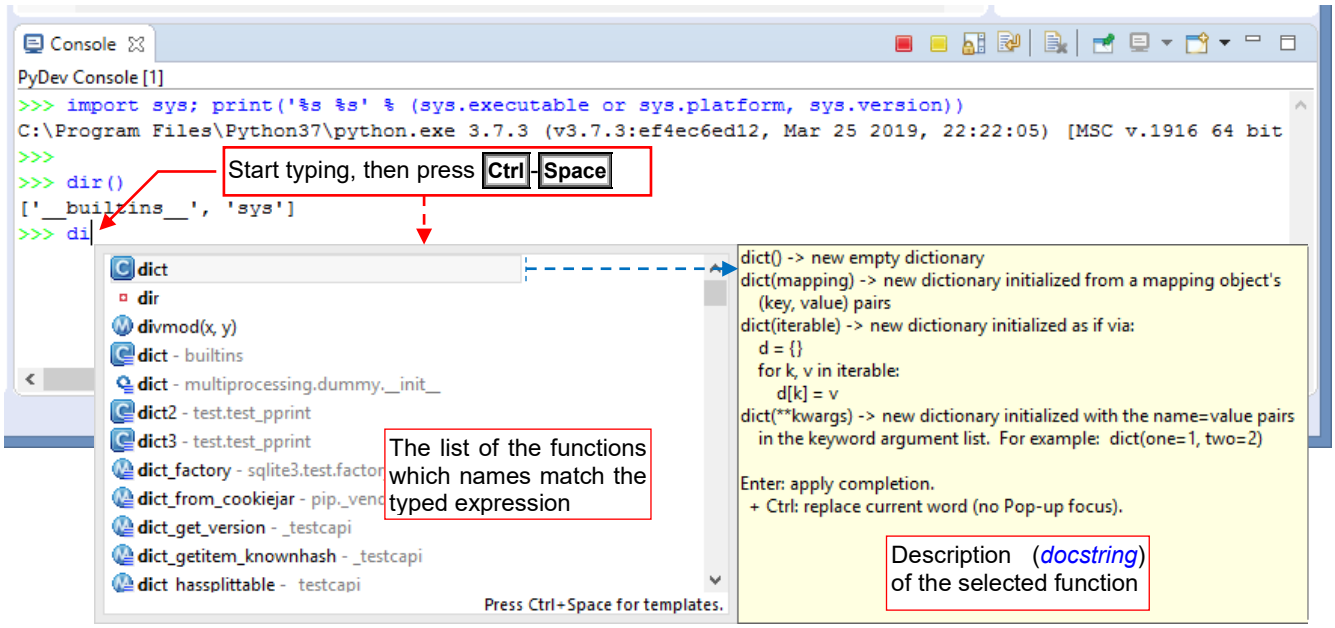


Figure 2.2.4 Example of the code autocompletion

Autocompletion usually takes effect when you type a dot after a name (for example, type "sys." in the console). Such a behavior does not bother writing of the normal code.

Well, let's finish this talk. Eclipse is a very rich environment, so I cannot describe all its functions here. It's time to write our simplest script (Figure 2.2.5):

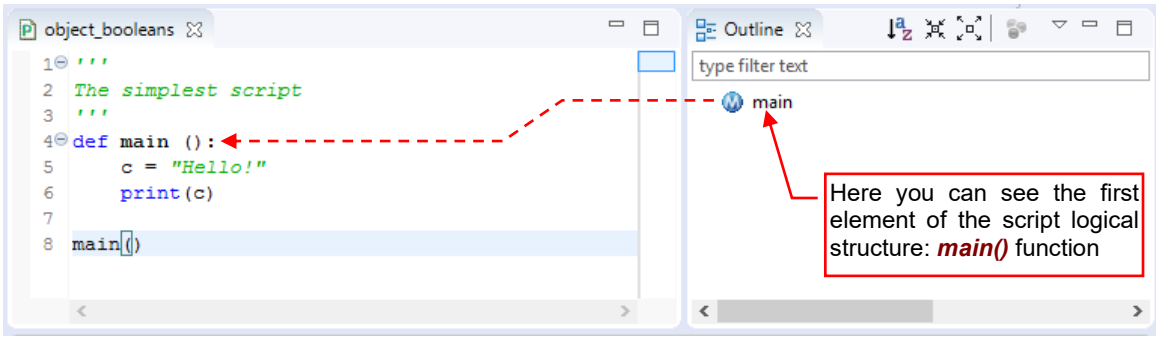


Figure 2.2.5 Our script — the first version, of course ☺

To run this script for the first time, highlight its file in the *PyDev Package Explorer*, then select from the ***Run*** dropdown the ***Run As→Python run*** command (Figure 2.2.6):

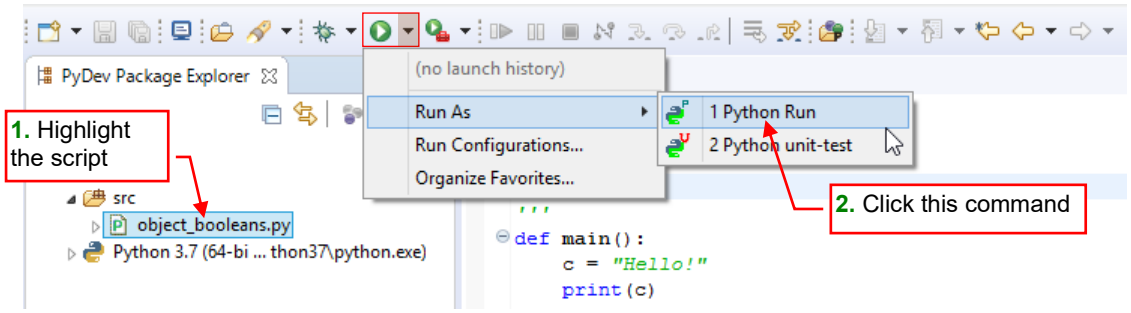


Figure 2.2.6 Running a script (for the first time)

PyDev will switch the console into the output mode, and you will see there the result of our script: the „Hello!" text (Figure 2.2.7):

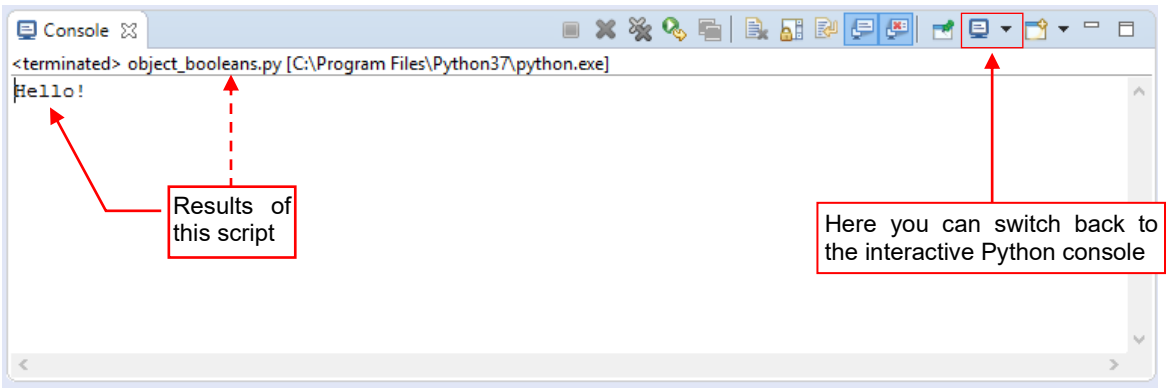


Figure 2.2.7 The results of this script: "Hello!" text.

When you run your script the first time, PyDev creates so-called *Run Configuration* in this project. You can find it in the launch history of the ***Run*** and ***Debug*** dropdowns (Figure 2.2.8):

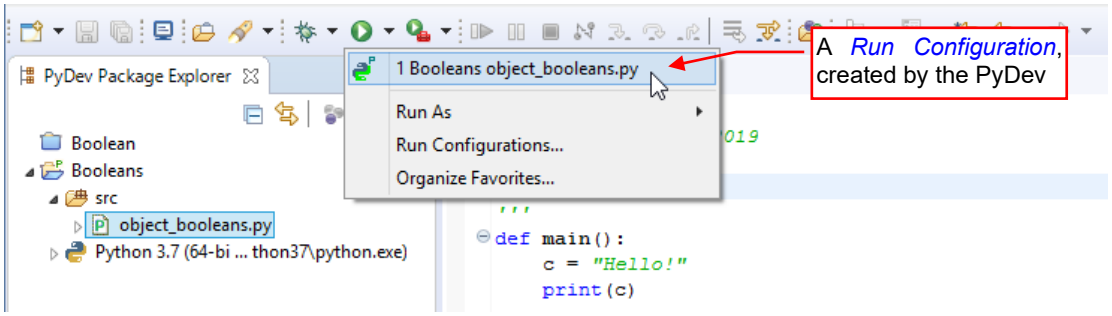


Figure 2.2.8 The default run configuration

Note that this configuration (*Boolean object _booleans.py*) is the only item (or the first item, if you wish) in the *Run* favorites list. To repeat this last run, you can just click the ***Run*** button (Figure 2.2.9):

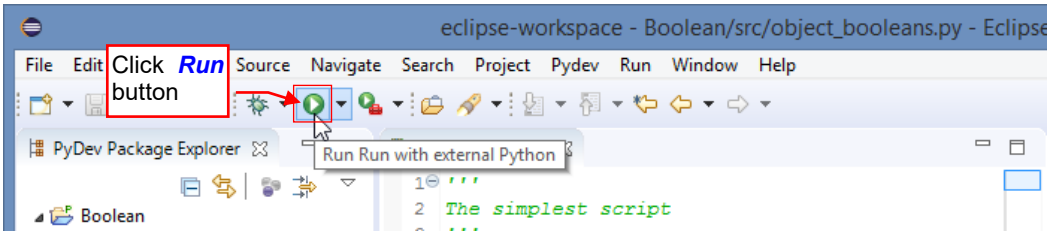


Figure 2.2.9 Launching the last run

In response, PyDev it will run your script again.

To refine this initial run configuration for your project, from the ***Run*** dropdown invoke the ***Run Configurations…*** command and find the definition of *Boolean obect_booleans.py* item (Figure 2.2.10):

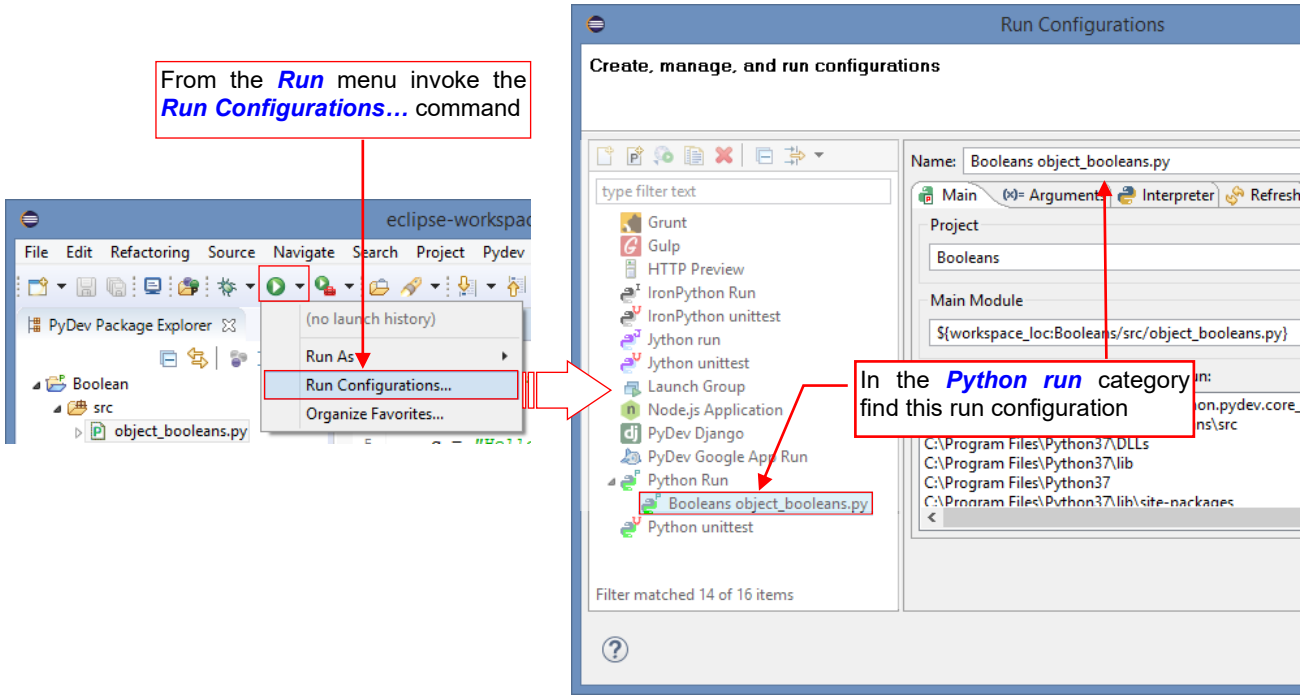


Figure 2.2.10 Opening the *Run Configuration* definition for our script

I renamed this configuration as **Run with external Python**. Additionally, in the ***Common*** tab I enabled both of the ***Display in the favorites menu*** options (*Debug* and *Run*: see Figure 2.2.11). (This setting "pins" this configuration into the favorites menu – just in case):

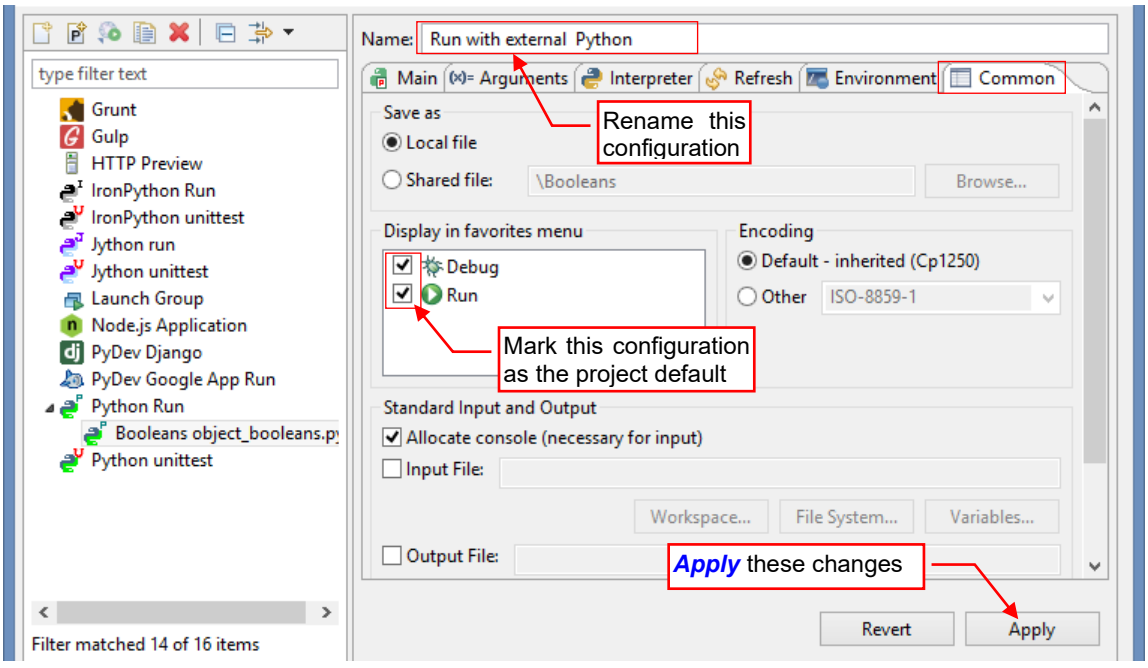


Figure 2.2.11 Altering the *Run Configuration* of our script

You can also define your own run configuration manually – see details on page 134.

- The run configurations names must be unique within the scope of the whole Eclipse workspace (in all its projects). In this guide I changed the name of the *Run Configuration* for the clarity of the further text. For the future projects you would better leave the run configurations their initial names.

Summary

- You can add to your PyDev projection a new pane with the Python console (page 24);
- The code autocompletion appears when you type a dot after an expression or press Ctrl-Space. It also displays the *docstring* for the function selected in the tooltip pane (see page 25);
- We have launched the simplest script and checked its result in the console (page 26);
- When you run your script for the first time, PyDev creates for this file so-called *Run Configuration*. You can manage them in the *Run→Run Configurations* dialog (page 27);

2.3 Debugging

To insert a breakpoint at appropriate script line, double-click (LMB) the grey bar at the left edge of the text editor window. Alternatively, you can also open the context menu at this point (Figure 2.3.1):

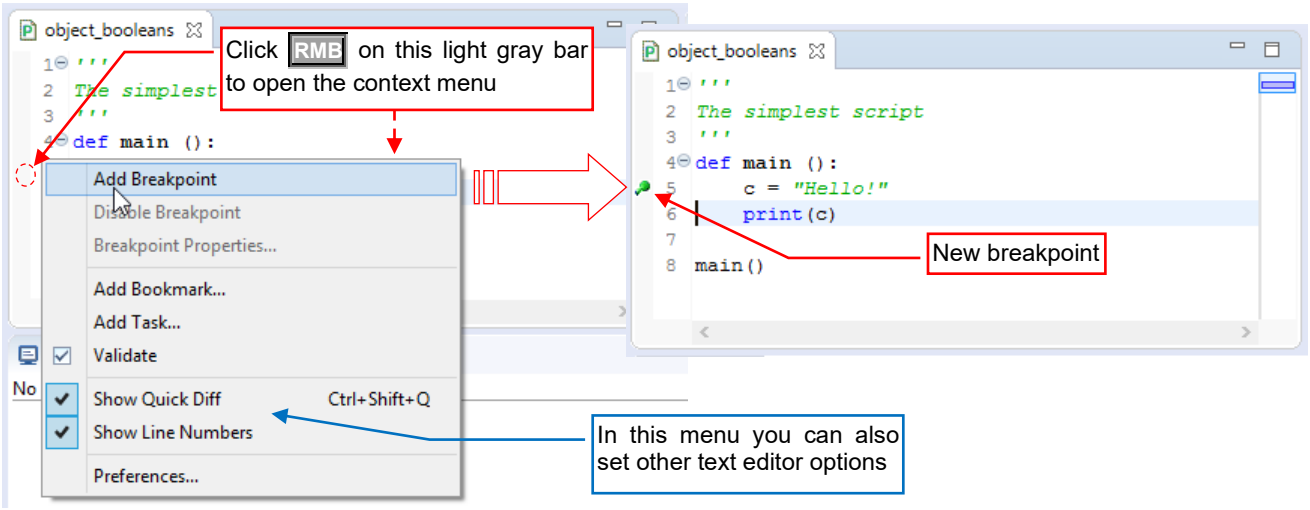


Figure 2.3.1 Adding a breakpoint

To open the context menu, click the RMB at the line where you want to insert new breakpoint. Then invoke the ***Add Breakpoint*** command. Eclipse will mark this code line with a green dot (Figure 2.3.1). You can remove this breakpoint in a similar way.

- You add or remove breakpoints by double clicking LMB the gray bar along the left text editor edge

To run the script in the debugger, click the bug icon (☺) on the toolbar (Figure 2.3.2):

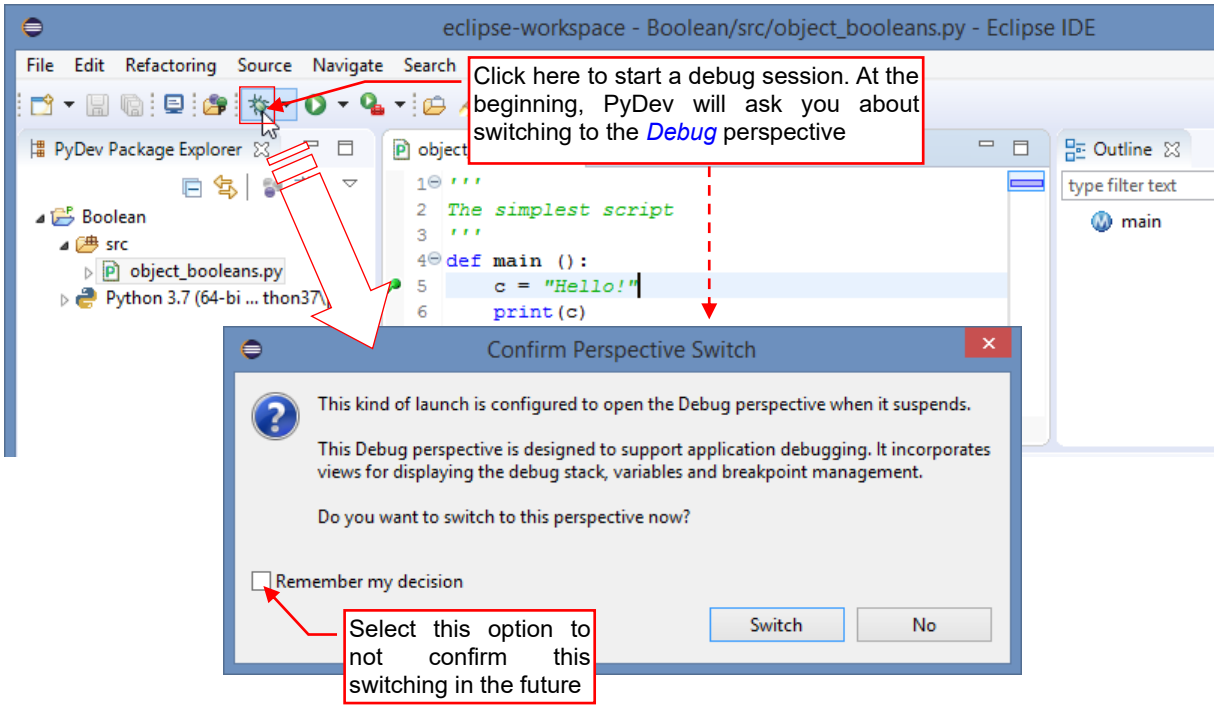


Figure 2.3.2 Launching a debug session

When you launch the debugger for the first time, Eclipse will display information about switching to the *Debug* perspective. (On the first run, it will add this perspective to your project). This additional perspective contains additional panes that are useful for debugging.

Figure 2.3.3 shows the screen layout of the *Debug* perspective, and the basic controls of the script execution (and their hot keys). Note that the debugger has stopped at our breakpoint:

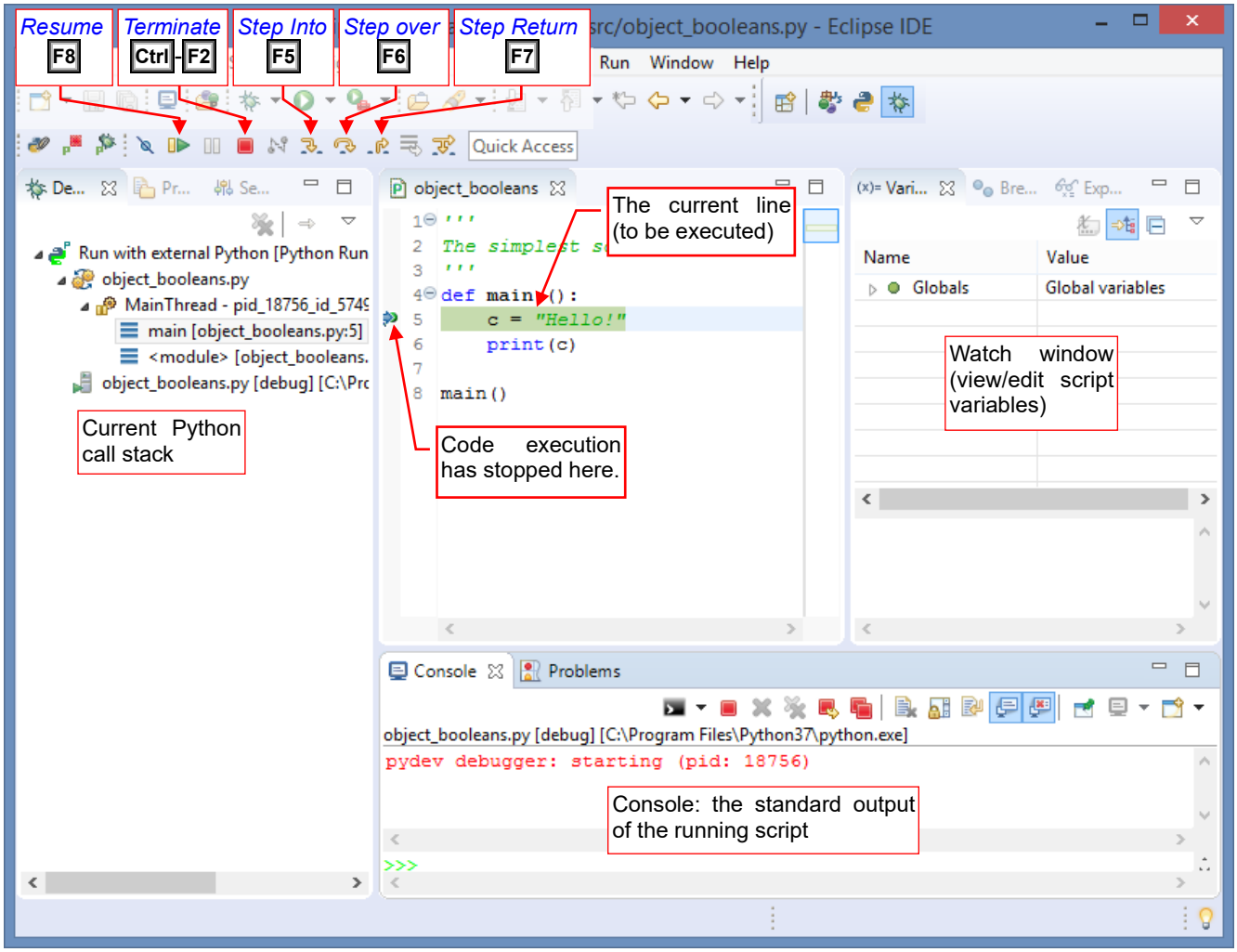


Figure 2.3.3 Screen layout of the *Debug* perspective

Green bar in the source code marks the line to be executed. When you press now the F6 key (*Step over*) — debugger will set the ***c*** variable and move the execution "green bar" to the next line (Figure 2.2.4):

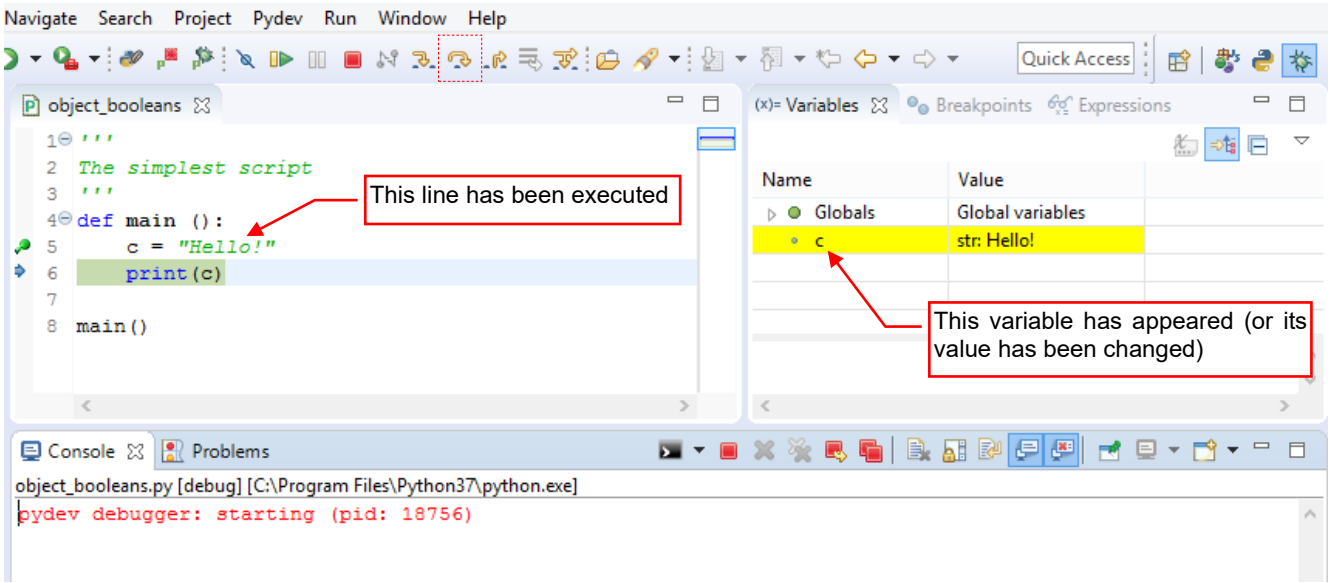


Figure 2.3.4 The state after pressing the F6 key (*Step over*)

When you press the F6 button again, the ***c*** string is "printed", and you leave the ***main()*** function (Figure 2.3.5):

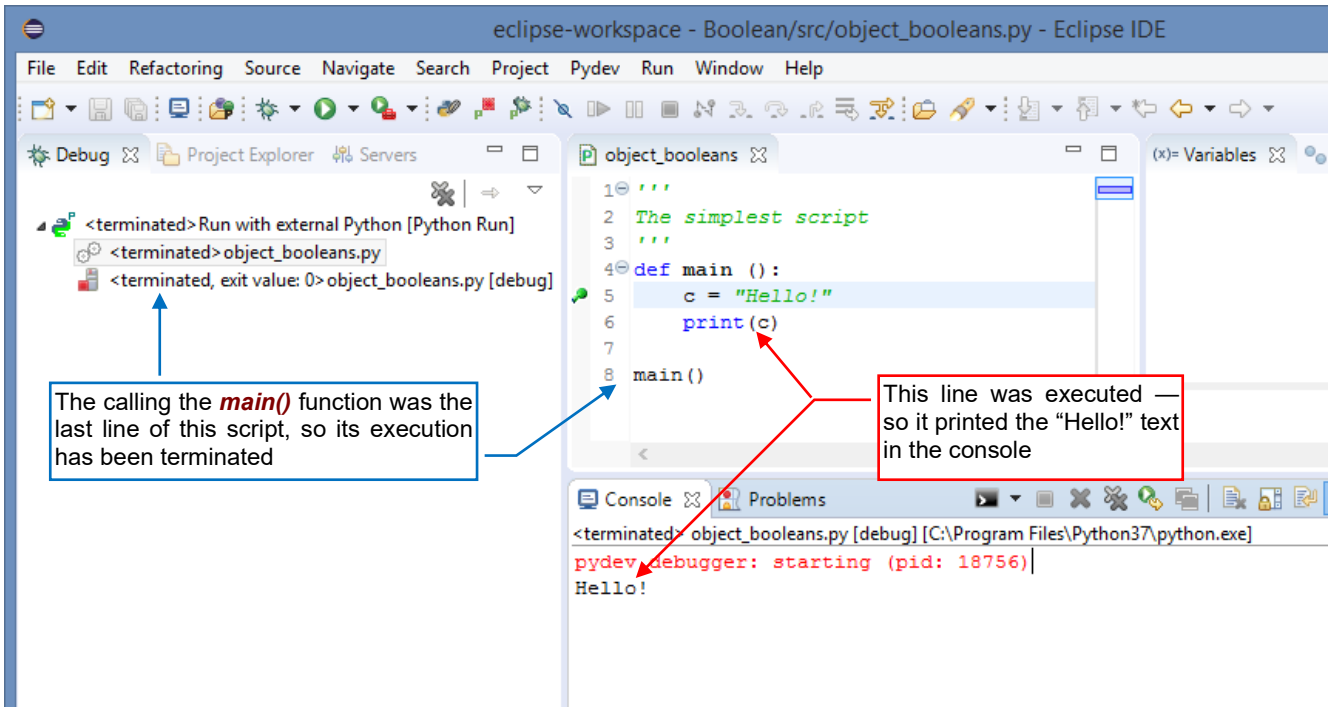


Figure 2.3.5 The state just after leaving the function

Note that when the debug session is over, Eclipse has grayed out the execution controls visible on the toolbar (Figure 2.3.6):

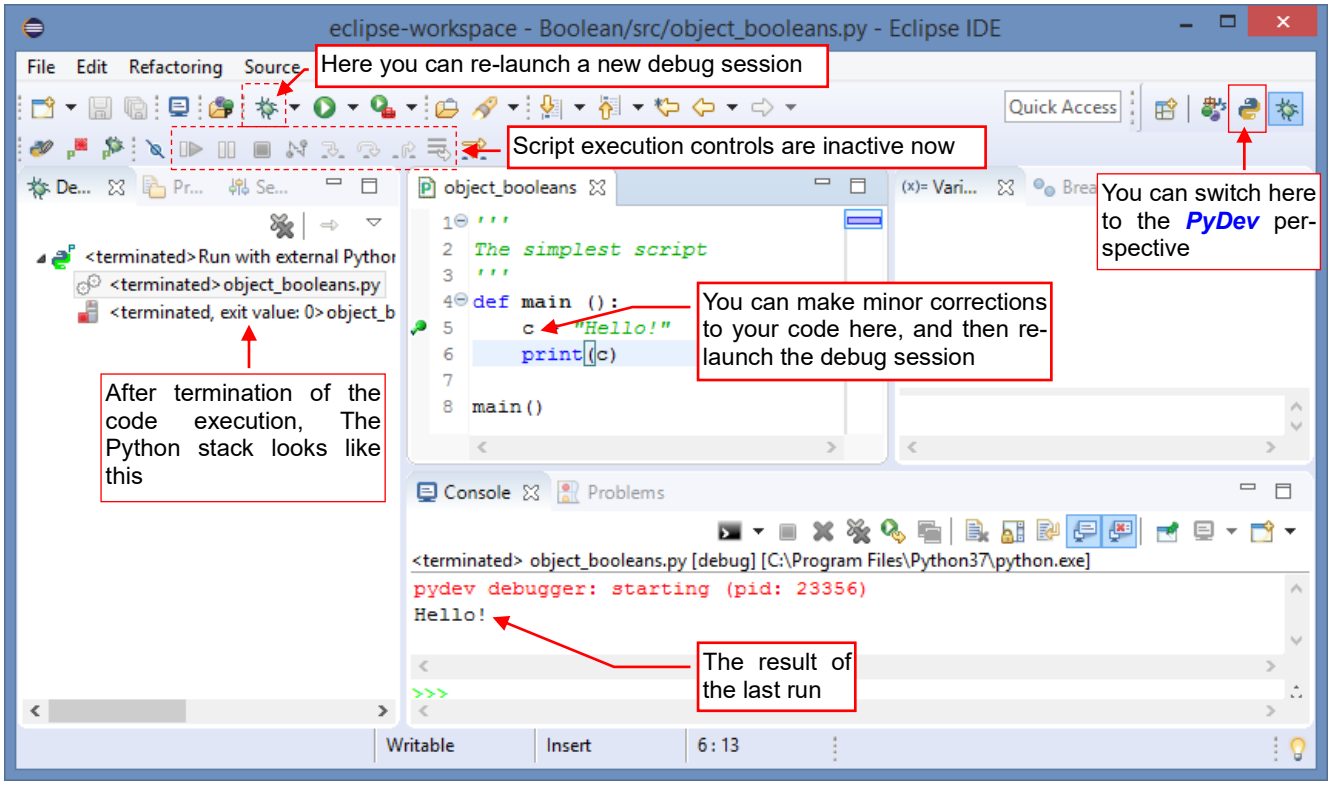


Figure 2.3.6 *Debug* perspective (no code is running)

You can make minor corrections of your script in this *Debug* perspective (the text editor windows are the same as in the basic *PyDev* perspective).

However, if you are going to make extensive changes — switch to the *PyDev* perspective. You have more helper tools there (Figure 2.3.7):

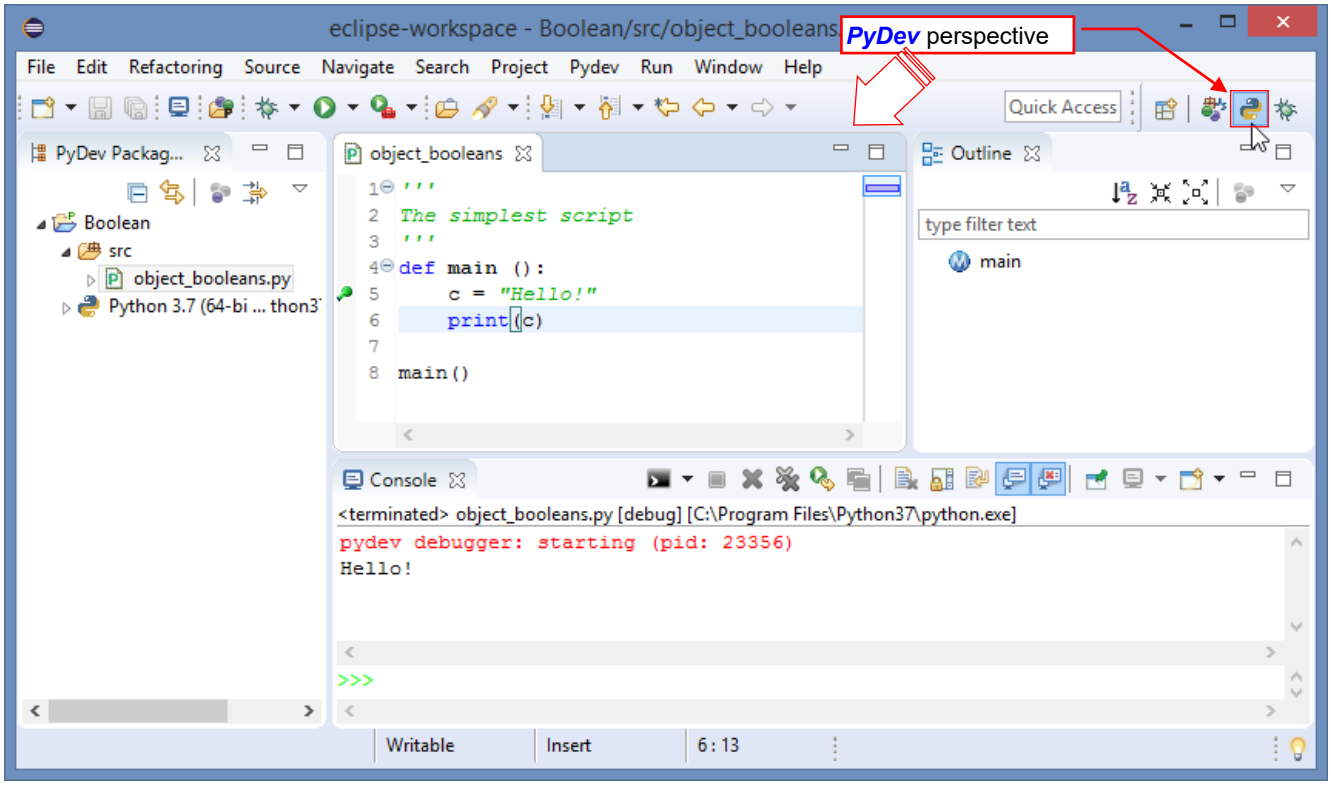


Figure 2.3.7 Back to the *PyDev* perspective — for the further work on the code

While working on your script, you will be continuously switching between the *Debug* and *PyDev* perspectives. That's why it is worth to enlarge these toolbar buttons by displaying their labels (Figure 2.3.8):

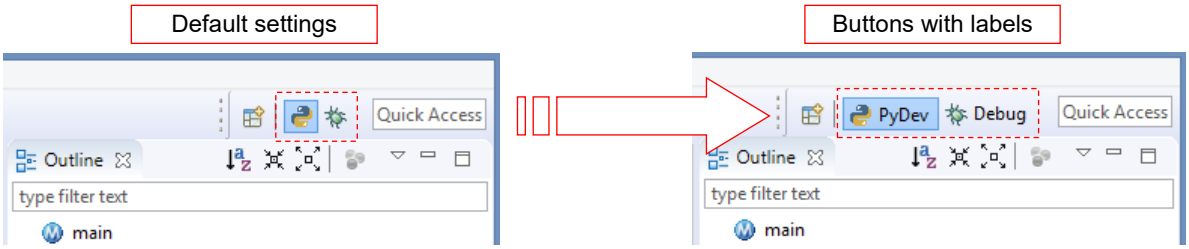


Figure 2.3.8 Enlarged perspective switches

I did it by selecting in the context menus of these buttons (opened by RMB click) the ***Show Text*** option (see page 133 for details).

Summary

- You have learned, how to set breakpoints in your code (page 29);
- We have launched our script in the debugger (page 29). On the first run, PyDev debugger creates a new *Debug* project perspective;
- You have learned the basic debugger commands: ***Step Into*** (F5), ***Step Over*** (F6), ***Resume*** (F8) (page 30);
- We have looked at some helper debugger panes: ***Variables*** (page 30) and ***Stack*** (page 31);
- After termination of the script execution you can remain in the *Debug* perspective to make eventual correction to your code. Then you can just click the ***Debug*** button and debug it anew.

Creating the Blender Add-On

This is the main part of the book. I am describing here the creation of a Blender add-on. We will start with a typical script - a plain sequence of Blender commands that runs "from the beginning to the end" (Chapter 3). Then we will adapt it for the required plugin interface (Chapter 4). In the result, we will obtain a ready to use add-on that implements a new Blender command.

Chapter 3. Basic Python Script

In this chapter, we will prepare a script that performs a Boolean operation on selected objects. I used this example to show in practice all the details of developing Blender scripts in the Eclipse environment. You will also find here some tips about the typical issues that may appear during this process. One of them is finding in the Blender API the right class and operator that support the functionality you need! (I think that still nobody, except the few Blender API developers, is familiar with the whole thing…).

3.1 Problem formulation

There are three Boolean operations that you can apply to solids: difference, union and intersection. They are often used in the mechanical or architectural modeling (Figure 3.1.1):

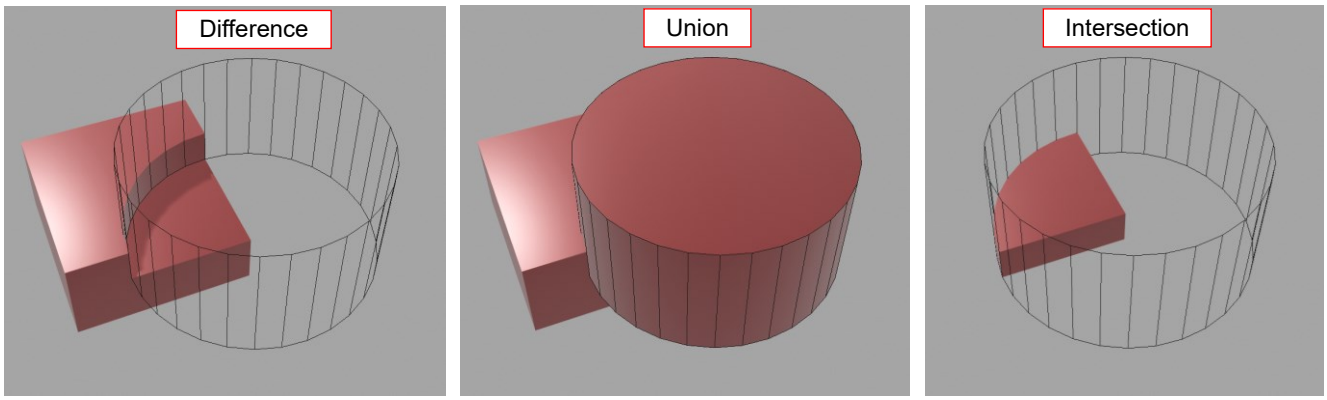


Figure 3.1.1 Boolean operations on solids

I regularly use the difference of two solids in forming various machine parts. For example, it allows me to quickly "drill" a hole in the basic shape. However, this "quickly" is not especially quick in Blender, because it implements these operations as the *Boolean* modifier. Let me to show it on an example: let's say that I want to drill a hole in plate **A** using auxiliary "tool" object **B** (Figure 3.1.2):

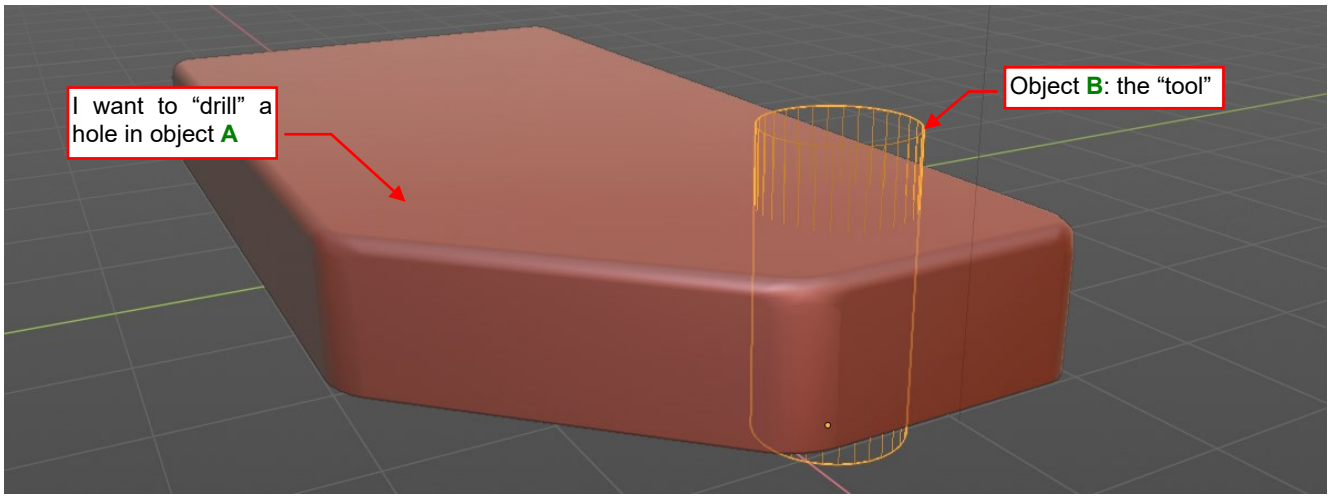


Figure 3.1.2 The initial state: the "raw material" (A) and the "tool" (B)

I start by selecting object **A** and adding to its modifier stack a ***Boolean*** modifier:

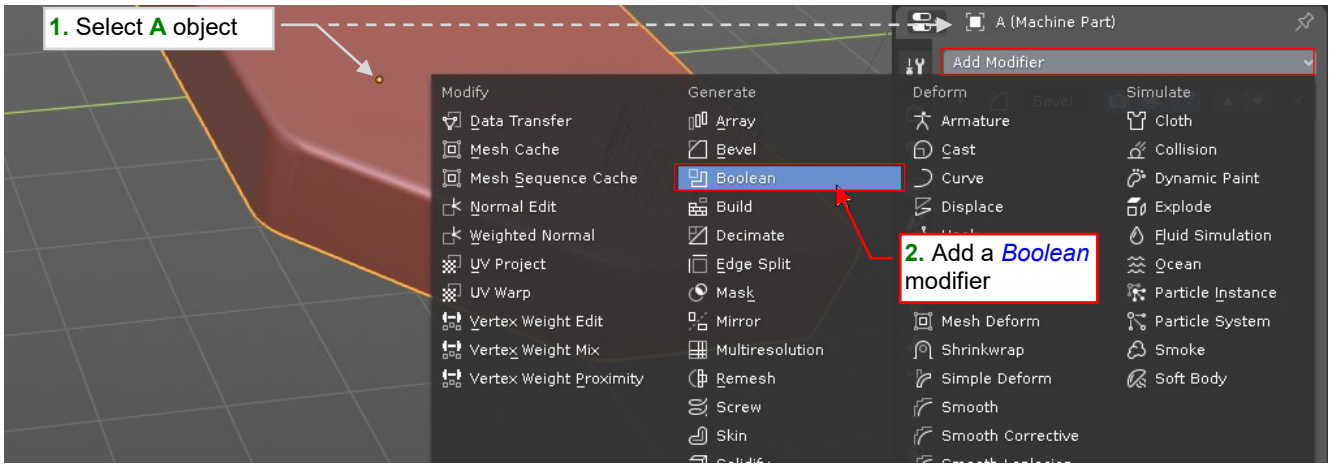


Figure 3.1.3 Adding a *Boolean* modifier to object A

The modifier stack of a mechanical part in Blender usually contains various items. Each new one is appended at the end of this list. In the next step I move the newly added *Boolean* modifier to the top position on the modifier stack. (I am doing this, because I want to modify the original mesh of object **A**, instead of the more complex shape with rounded edges, generated by the previously added *Bevel* modifier):

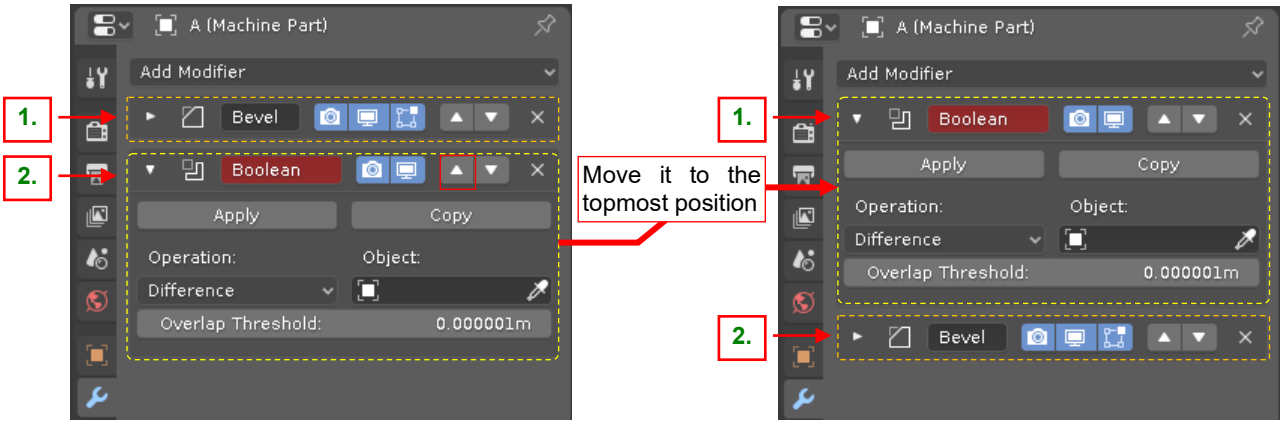


Figure 3.1.4 Moving the *Boolean* modifier to the topmost position on the object modifiers list

The *Boolean* modifier is set to the ***Difference*** operation by default, so I do not need to change this option. In the next step I assign the "tool": object **B** to this modifier (Figure 3.1.5):

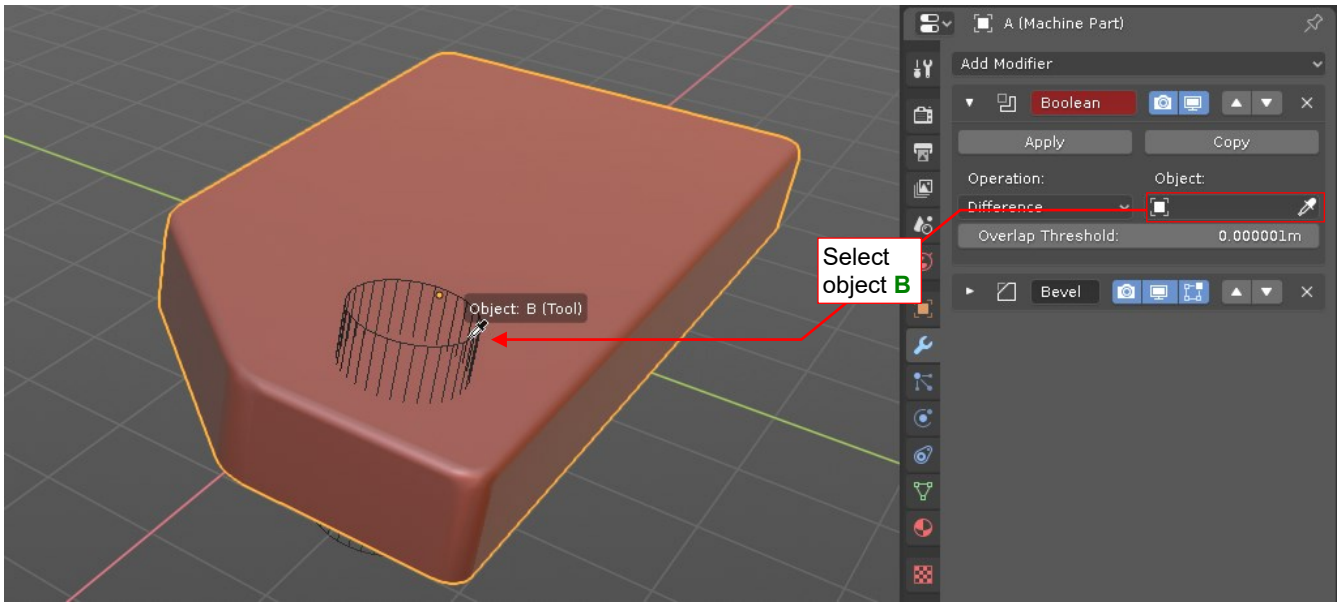


Figure 3.1.5 Assigning the "tool" object (B) to the *Boolean* modifier

Finally, I ***Apply*** this modifier to the object mesh (Figure 3.1.6), because I do not want to keep object **B**:

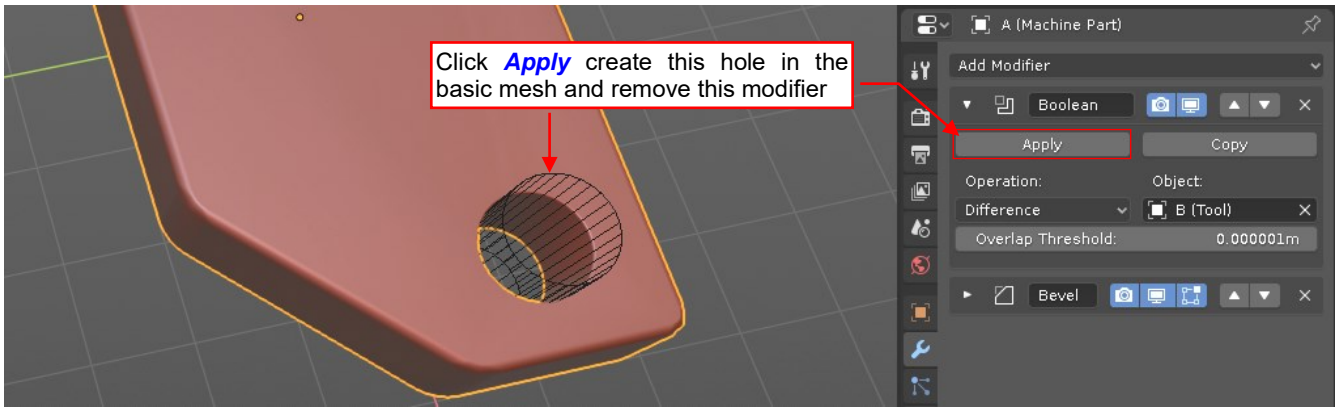


Figure 3.1.6 Applying the modifier results to the object mesh

In the result, the *Boolean* modifier disappeared, and the new faces that form this hole are added to the "editable" mesh of object **A** (Figure 3.1.7). Now I can move object **B** to next location, to create another hole:

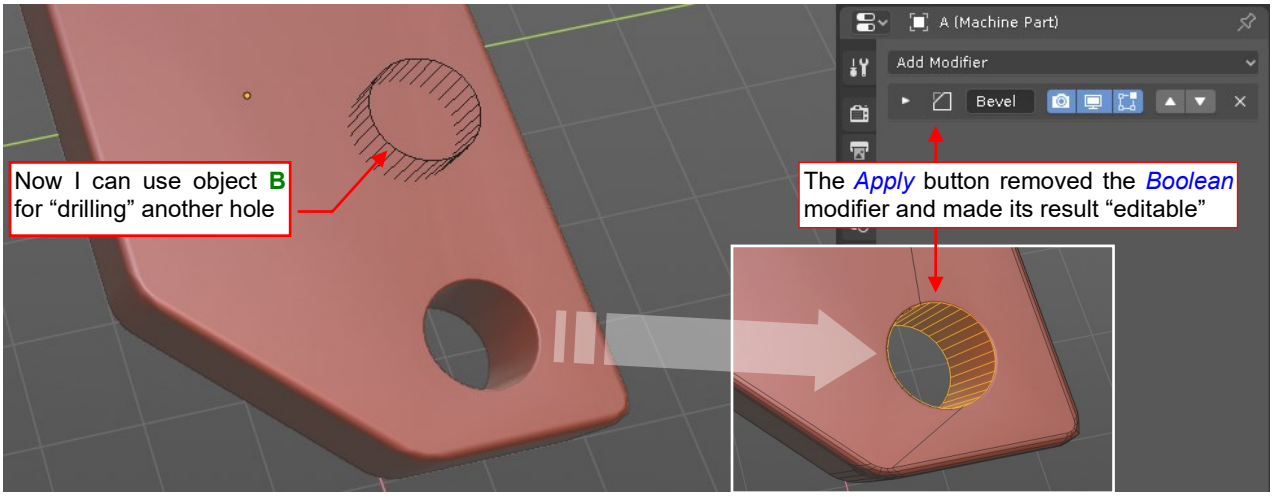


Figure 3.1.7 The final result of this *Boolean* operation

The *Boolean* modifier is a great tool when you need a "portable hole": a feature which size, shape and location you can alter many times during the project. It allows you to create complex shapes from basic elements of simple forms. (It is much easier to "unwrap" such simple meshes in the UV space for eventual texturing). However, because of its "dynamic" nature, this modifier also has some drawbacks:

- You must keep the "tool" object in place (in a hidden collection?). While this is not a problem when you have just a few of such items, it becomes an issue for the real Blender scenes with complex mechanical models. They can contain several dozens of such auxiliary objects;
- You cannot edit properties of the edges or faces dynamically generated by a modifier. In particular, you cannot assign these edges bevel weights, or flag as seams (which is useful for better unwrapping in the UV space);

That's why I usually apply *Boolean* modifiers just after they are completed (by clicking their *Apply* buttons). For this purpose, I need the simple, "destructive" *Boolean* operations, as shown in Figure 3.1.8:

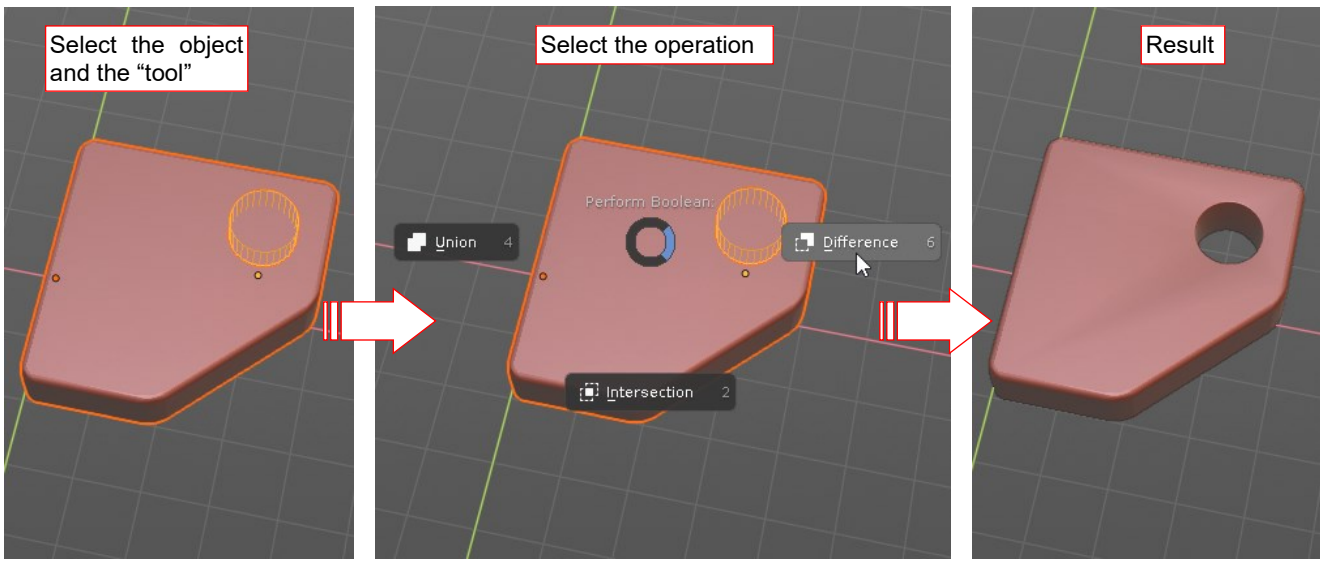


Figure 3.1.8 The "destructive" *Boolean* operation that I need

It would begin in the typical way: by selecting the tool object (or objects) and the single target object (the active object). Then I would invoke the command (for example – using a keyboard shortcut) and from its pie menu select one of the Boolean operations. After this, it would automatically execute all the steps described above, producing a hole in the mesh of the target object.

In the options of this Boolean command (not shown in in Figure 3.1.8), I should also be able to mark a checkbox that preserves operation results as a modifier. This command should propose the last used options as the defaults when I would invoke it for the next time.

In this chapter we will write a Blender script that will use the *Boolean* modifier for implementing such a "destructive" version of this operation. Basically, it will repeat the sequence of steps described in this section. In the next chapter, we will convert this script into a professional Blender add-on, which works like the command shown in Figure 3.1.8.

Summary

- Boolean operations on solids are often used in mechanical and architectural models;
- Blender 2.8 lacks the “destructive” ***Boolean*** command[1]. There is only the “dynamic” *Boolean* modifier (page 35, 36);
- You must keep the auxiliary “tool” objects used by the “dynamic” Boolean modifier. For Blender models of a real-life complex mechanism that consists hundreds various parts, it becomes a serious burden;
- You can obtain the effect of a “destructive” Boolean command by applying the *Boolean* modifier (page 36, 37). However, this operation requires several steps. In this chapter we will prepare a Blender API script that automatizes this task;

[1] Among the “community” add-ons installed with Blender you can find in the *Object* category a plugin named *Bool Tools*. This Python script implements the *Union*, *Difference*, *Intersection*, and *Slice* commands. (*Slice* = *Difference* + *Intersection*). However, this command differs in certain details from the command that I proposed in Figure 3.1.8. For example – it always deletes the “tool” object, and applies all the existing modifiers of the target object before applying the Boolean operation. (It leaves the *Boolean* modifier at the end of the modifiers list). In principle the tool proposed in this chapter is so simple that I decided to write it from scratch, instead modifying the code of the *Bool Tools* add-on. What’s more, I need such a new, real Blender plugin as the example for this guide.

3.2 Adapting Eclipse to Blender API

To write scripts for Blender in an easier way, we need to "teach" PyDev the Blender API. Its code autocompletion should be able to suggest API objects, methods, and fields, just as it does for the standard Python modules. Fortunately, PyDev has such a possibility. We just have to provide it a kind of simplified Python file that contains only declarations of the API classes, their methods and properties. The very idea is similar to the header files used in C/C++. To distinguish these "header files" from ordinary Python modules, we use the **.pypredef* extensions in their names (a derivate from "*Python predefinition*").

I modified Campbell Barton's script, which generated the Python API documentation for Blender 2.5. Using it, I was able to create the **.pypredef* files for the entire Blender API. You can find them in the data that accompanies this book. Just download the http://airplanes3d.net/downloads/pydev2/pydev-blender.zip file and unzip it – for example into the folder that contains Blender binaries (Figure 3.2.1):

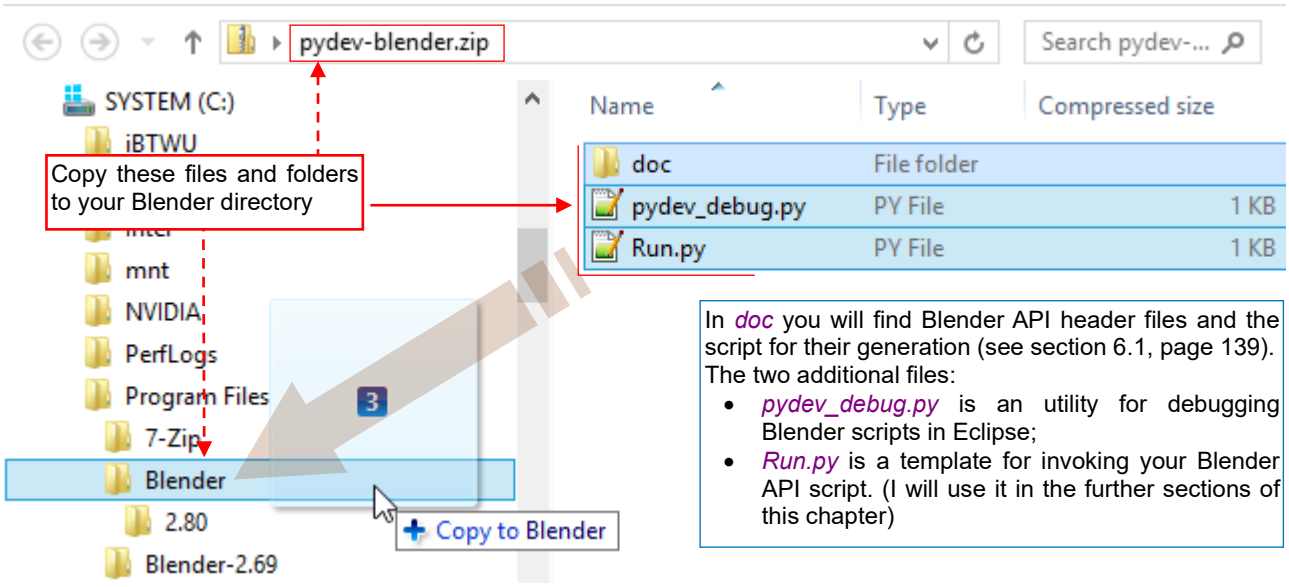


Figure 3.2.1 Unpacking additional files to the Blender folder

- In the folder that contains Blender executables (in Figure 3.2.2 this is *Blender* directory) place at least the single file: *pydev_debug.py*. We will need it for debugging our scripts.
- You can place the *Run.py* file and *doc* folder in any directory you wish. However, in such a case in the *doc\refresh_python_api.bat* batch file update the line that calls *..\blender* (see page 141).
- If you follow the picture above and place the *Run.py* file and *doc* folder in the Blender directory, do not forget to add the local *Users* group the create/write rights for Blender directory and its subdirectories.

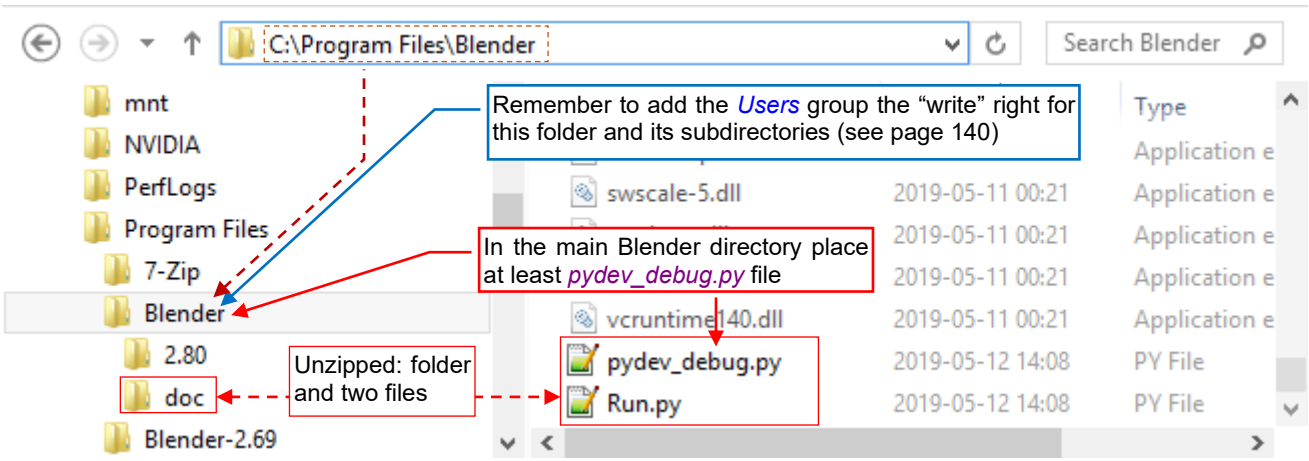


Figure 3.2.2 The files required to follow this book

To use the **.pypredef* files in a PyDev project, invoke the ***Project→Properties*** command (Figure 3.2.3):

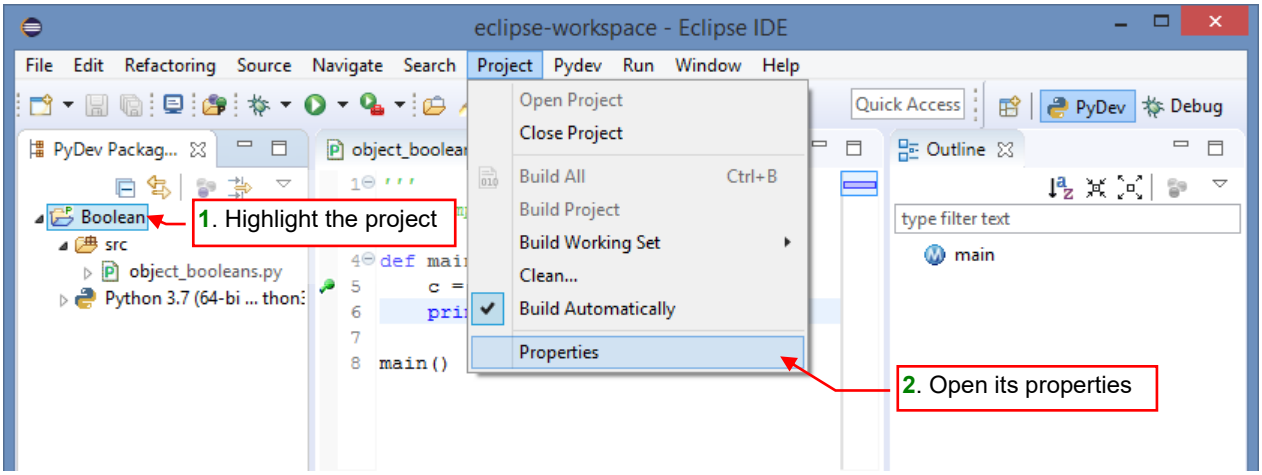


Figure 3.2.3 Opening the project configuration window

It opens the project *Properties* window. In its left pane select the ***PyDev – PYTHONPATH*** section. It displays several tabs on the right side of this window. Select from them the ***External Libraries*** tab (Figure 3.2.4):

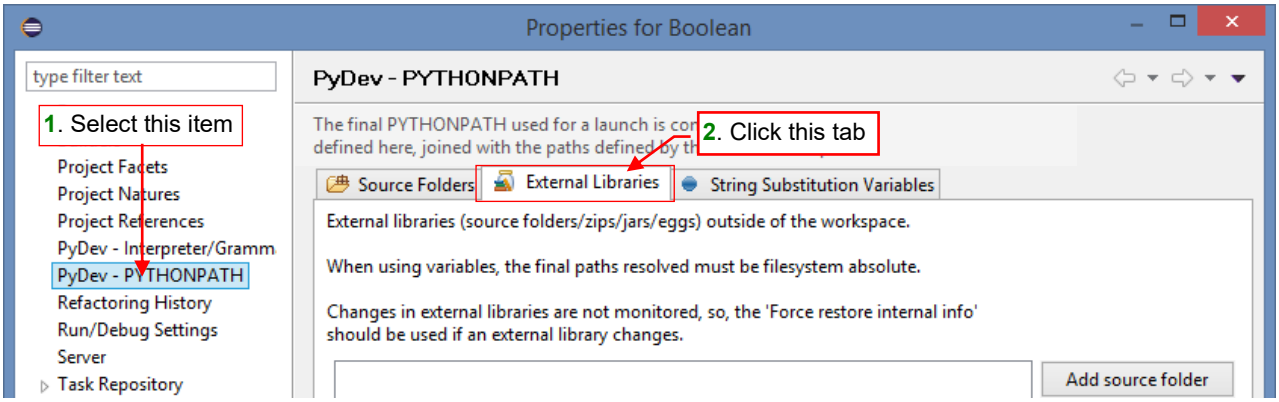


Figure 3.2.4 Navigating to the *PyDev - PYTHONPATH:External Libraries* pane

Add here (***Add source folder***) the full path to the *doc\python_api\pypredef* directory (Figure 3.2.5):

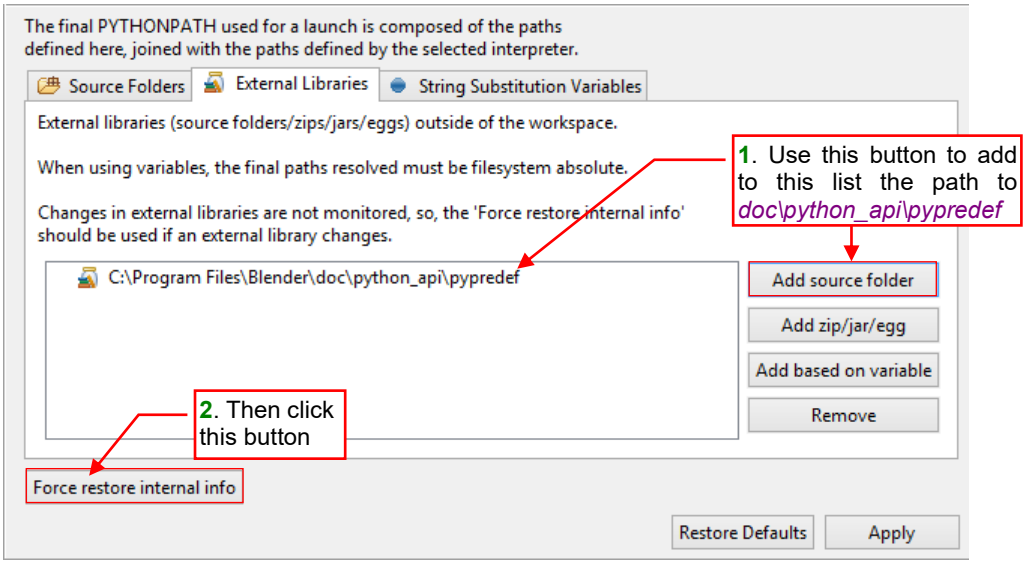


Figure 3.2.5 PyDev PYTHONPATH **configuration**

After every change made to PyDev **PYTHONPATH** make sure that you have clicked the ***Force restore internal info*** button (Figure 3.2.5). In response, Eclipse will display information about the progress of this process in the status bar (for a second or two).

From this moment, when you add to your script appropriate *import* statement, PyDev will use the whole hierarchy of the Blender API in its autocompletion (Figure 3.2.6):

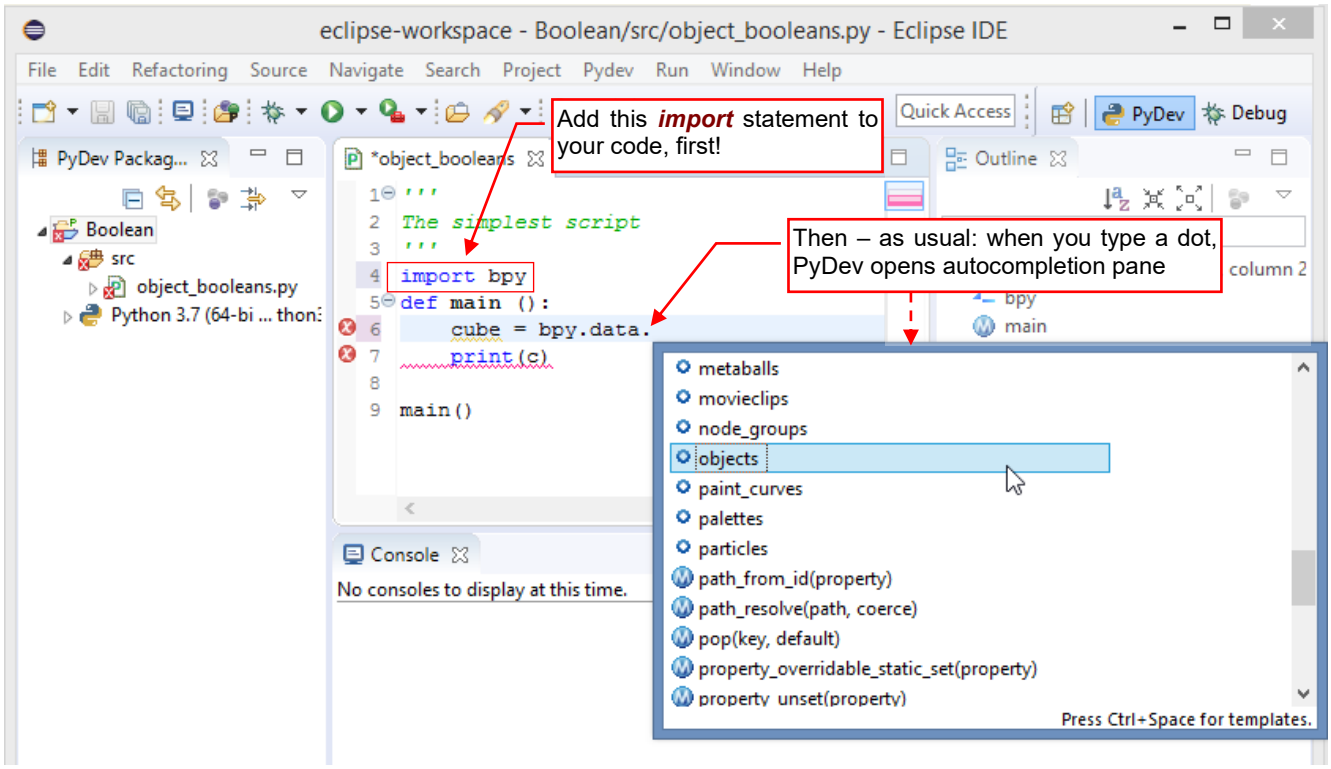


Figure 3.2.6 Code autocompletion in a Blender API statement

The list of the class members appears when you type a dot. What's more, when you hold the mouse cursor for a while over a method or an object name — PyDev will display its description in a tooltip (Figure 3.2.7):

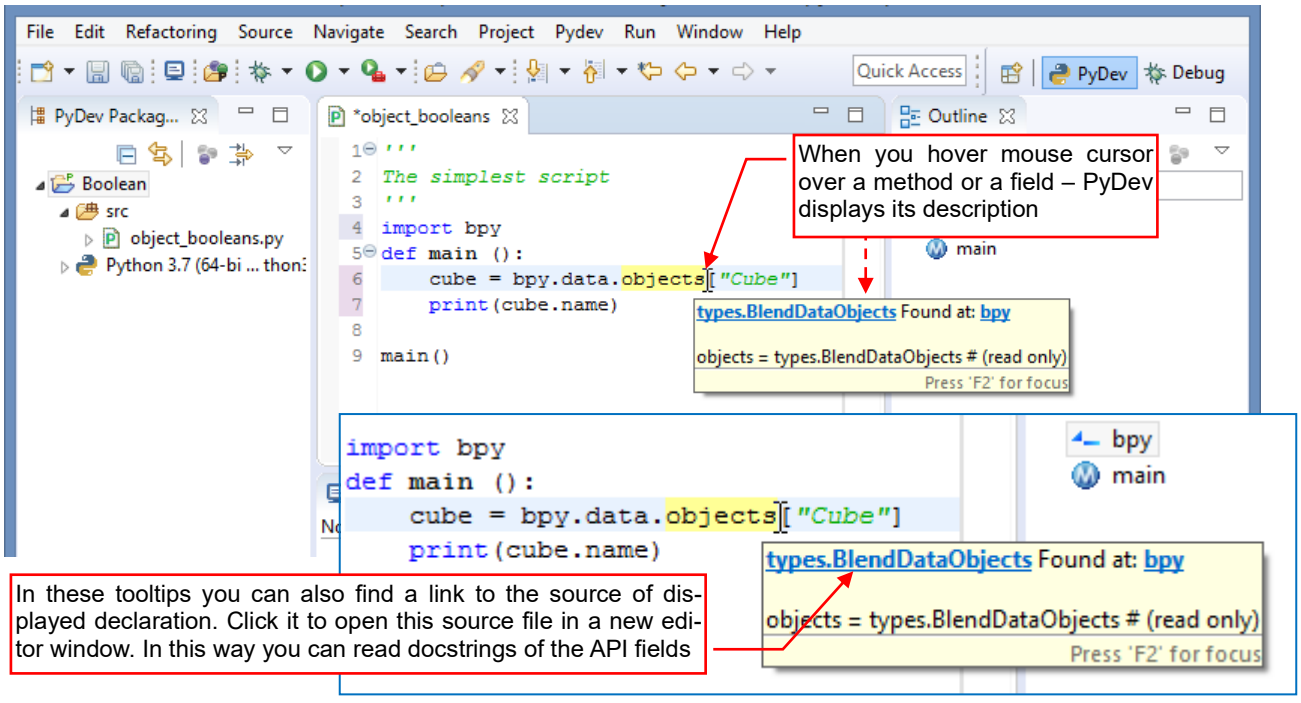


Figure 3.2.7 Displaying descriptions of Blender API objects

When you move the mouse outside, the tooltip with the method description disappears. You can also click the reference link, placed in the first line of the tooltip text (see Figure 3.2.7). This link opens the source file in the line that contains declaration of this item (Figure 3.2.8):

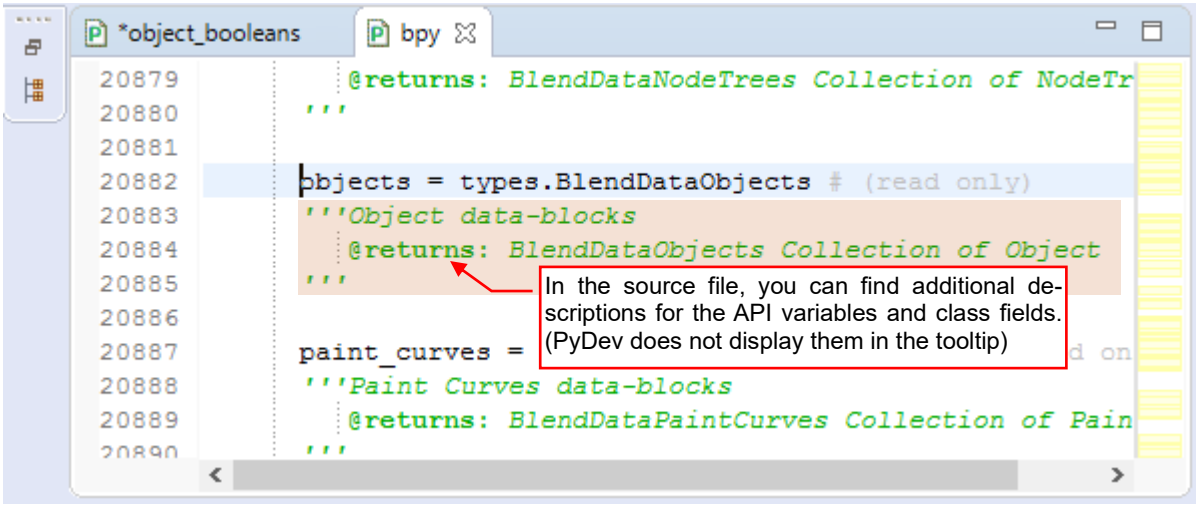


Figure 3.2.8 Property declaration in the predefinition file (*bpy.pypredef*), opened using the tooltip reference link

From the PyDev point of view, such a declaration is in the predefinition file (*bpy.pypredef*). That's why it is opened as the source code.

You can also locate the selected field or function in the *Outline* pane (Figure 3.2.9):

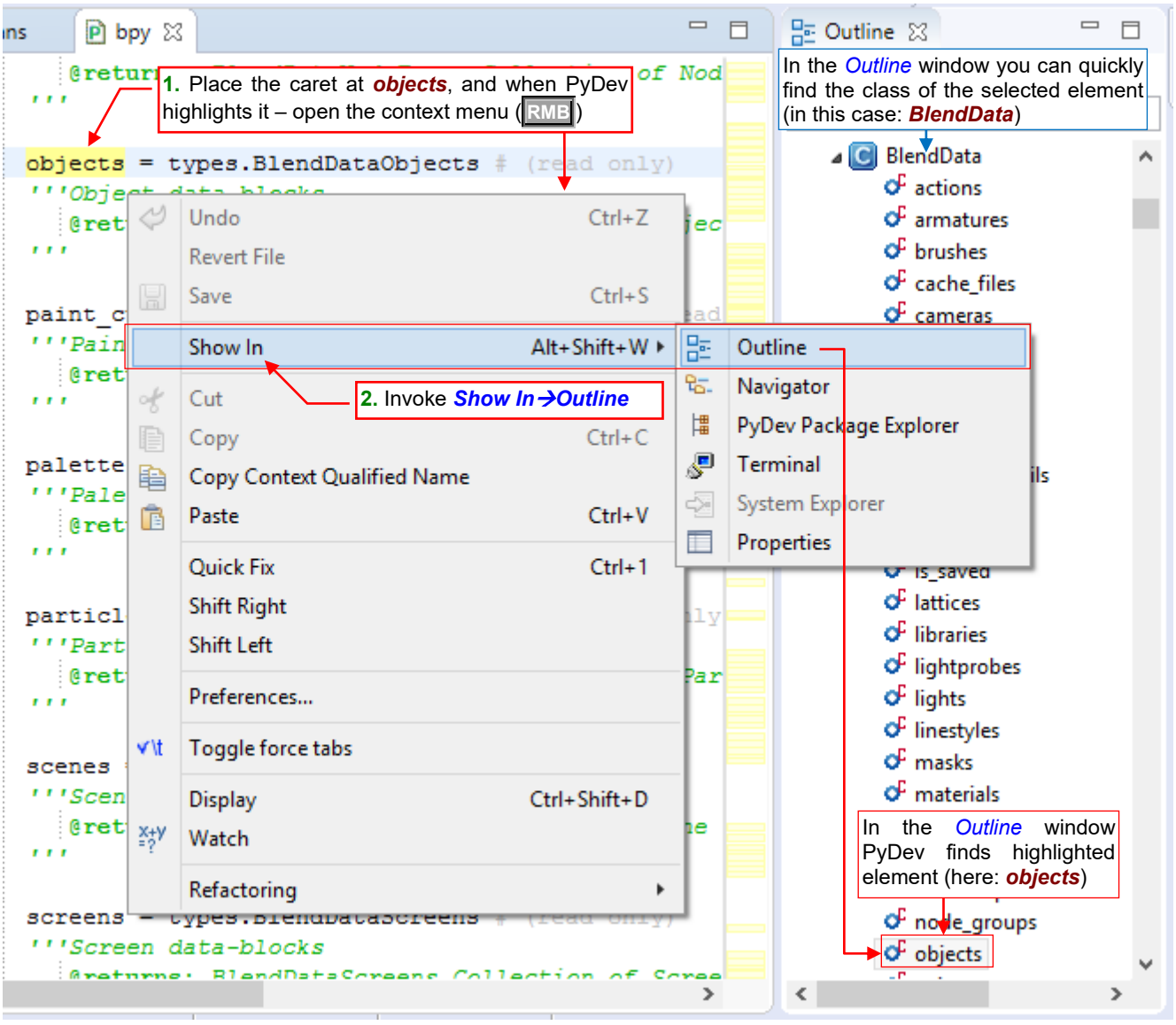


Figure 3.2.9 Finding the selected class member in the *Outline* panel

Note that the *Outline* pane is a useful “training aid”. You can use it for an interactive “walk around” the whole Blender Python API. Let’s collapse the tree of the basic ***bpy*** module to its root nodes (Figure 3.2.10):

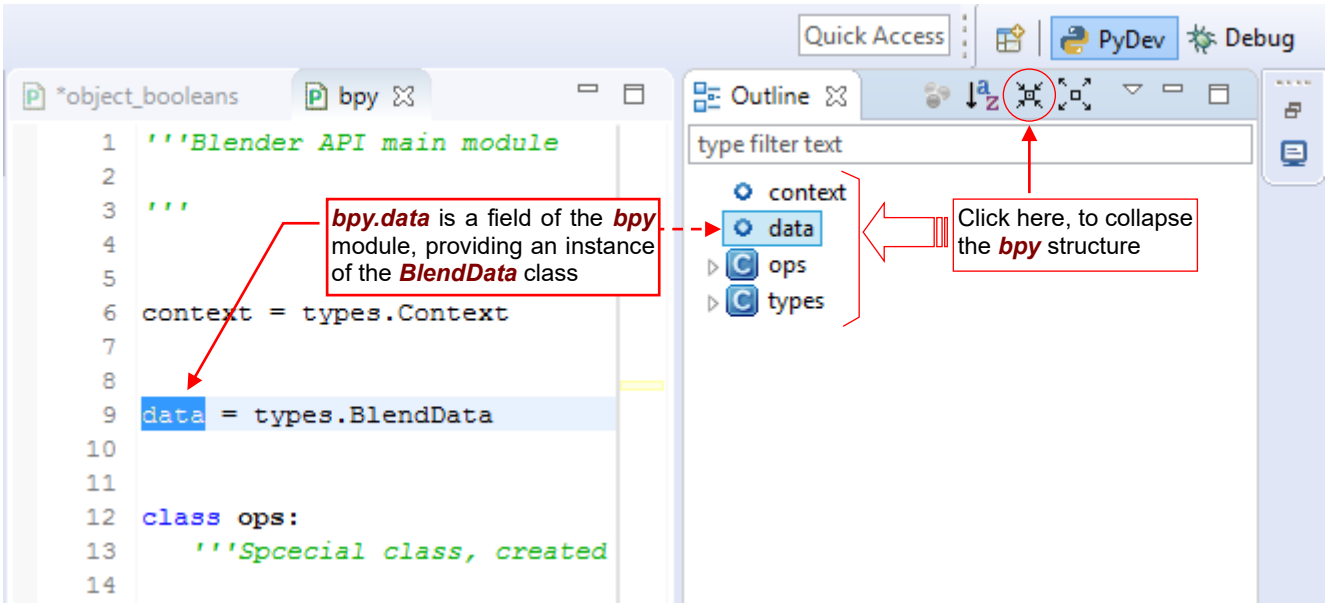


Figure 3.2.10 The root structure of the Blender API

Here you can see the basic API elements:

bpy.data provides access to the data of the current Blender file. Each of its fields is a collection of objects of the same type (***scenes***, ***objects***, ***meshes***, etc. — see Figure 3.2.9);

bpy.context provides access to the current Blender state: the active object, scene, current selection;

bpy.ops contains all Blender commands (operators). (In the Python API, each Blender command is implemented as a single method of this class);

bpy.types contains definitions of all classes that are used in the ***bpy.data***, ***bpy.context*** and ***bpy.ops*** structures;

When you look inside ***bpy.types***, you will see an alphabetical list of all classes used in the API. An exception from this rule is the ***bpy_struct*** structure, located on the first place. This is the base class of all other API classes. Its methods and properties are always available in each Blender object (Figure 3.2.11):

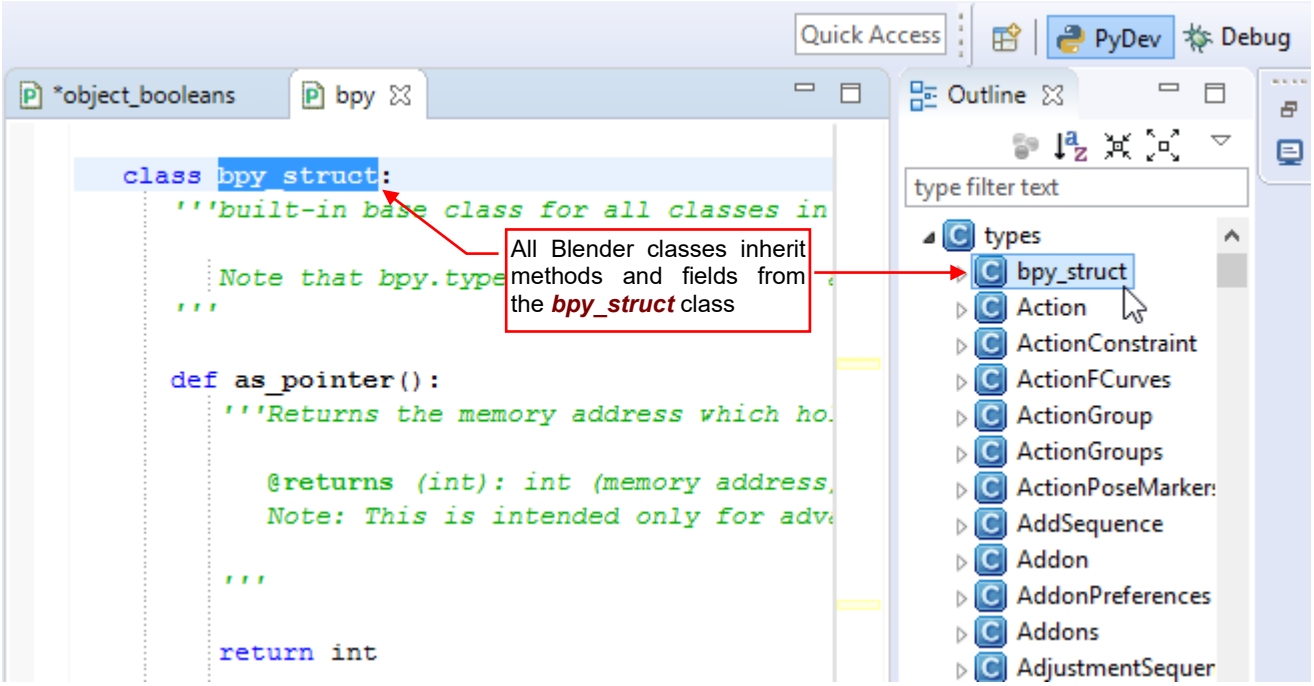


Figure 3.2.11 *bpy_struct*: **the base class of all Blender API classes**

Note that ***bpy_struct*** methods may be not fully implemented in the derived classes. For example — ***bpy_struct*** has the ***items()*** method. It is implemented only by the API collections (for example — ***MeshEdges***, the collection of ***MeshEdge*** objects) together with additional methods, like ***add()*** (Figure 3.2.12):

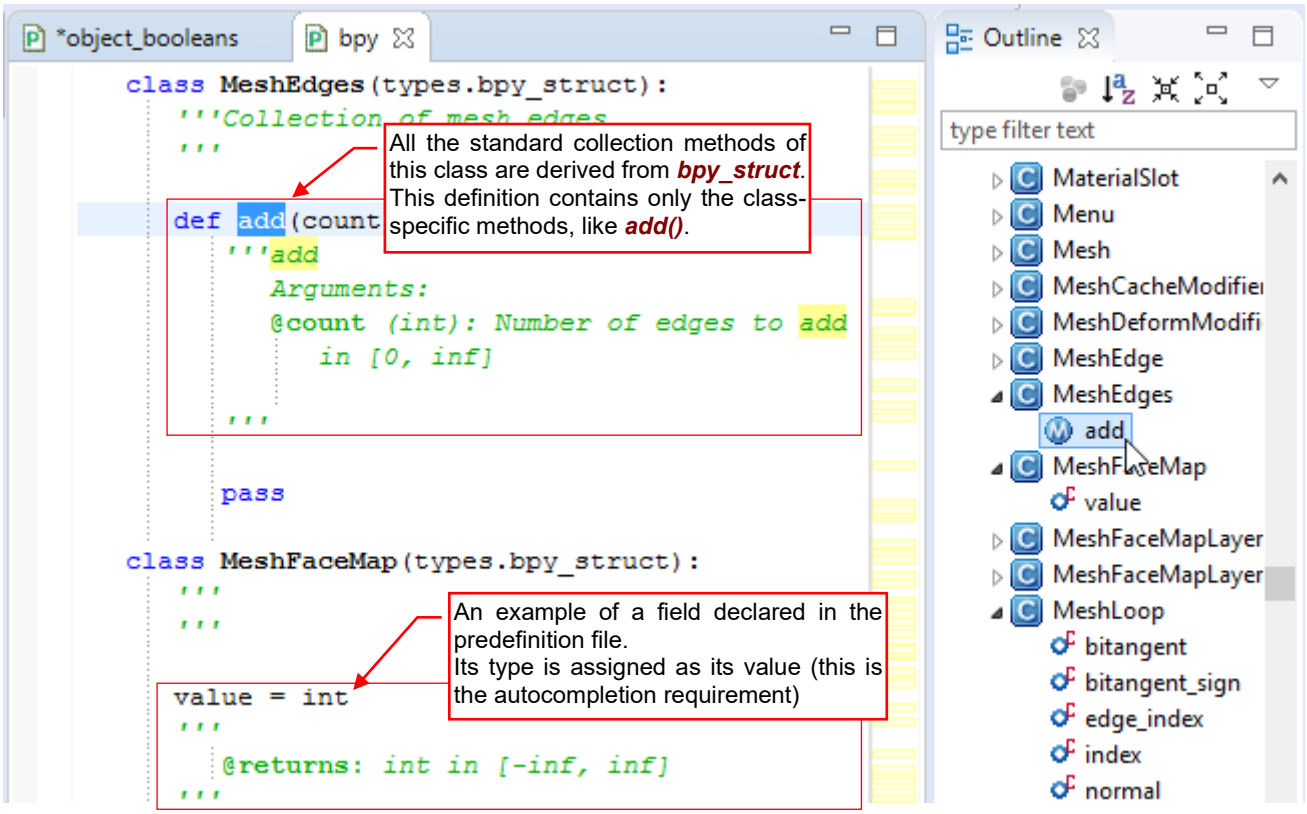


Figure 3.2.12 Derived Blender API classes — declarations of their methods and properties

Of course, all the classes that represent single API elements (like ***MeshEdge***) have their ***items()*** methods empty (as well as many other ***bpy_struct*** methods and properties).

The inheritance of the ***items()*** method in every Blender API collection class obscures the results of automatic code completion. PyDev reads from the base class definitions that each of them contains just ***bpy_struct***s. Fortunately, it is possible to "suggest" PyDev the appropriate type of a variable. Just put earlier in the code a line that assigns to this variable the appropriate type (Figure 3.2.13):

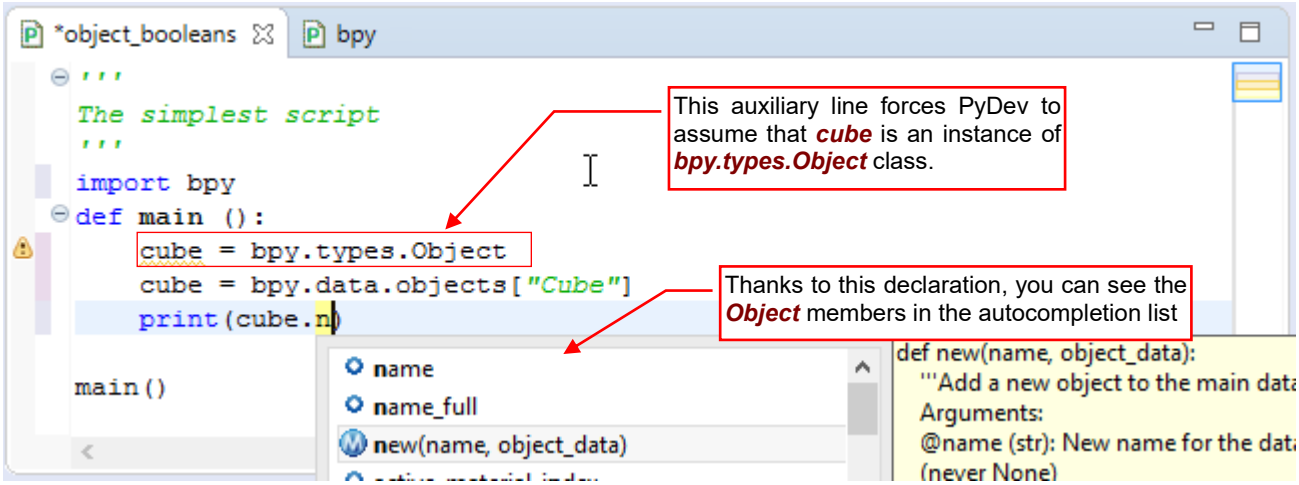


Figure 3.2.13 "Variable declaration" — a workaround of the Blender API collection type problem

In practice, you should add such "declaration line" only for a moment, when you need to use the automatic code completion. Always place it above the line where this variable receives its first "real" value. In this way, your script will work correctly even if you forget to comment out this "declaration".

Anyway - PyDev detects such lines, identifying them as "unused variables". It marks them with appropriate warning (Figure 3.2.14):

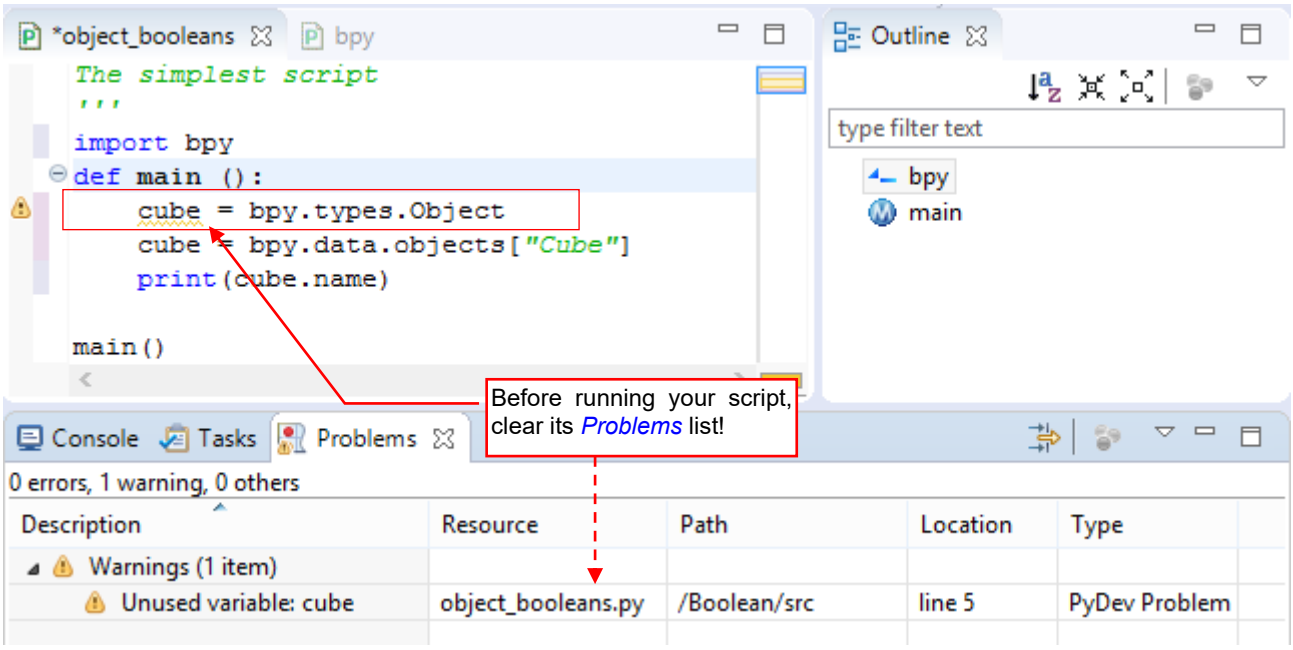


Figure 3.2.14 PyDev warnings for each "type declaration" line

It is a good practice to look into the ***Problems*** tab from time to time. You will see there all the lines you have forgotten to comment out. Using this list, you can quickly fix these issues.

So far, we have discussed only the ***bpy.types*** branch. What about Blender operators (***bpy.ops***)? There are plenty of them! They are grouped into modules (classes): ***action***, ***anim***, ***armature***, … and so on. Let's expand the ***bpy.ops.brush*** module (Figure 3.2.15):

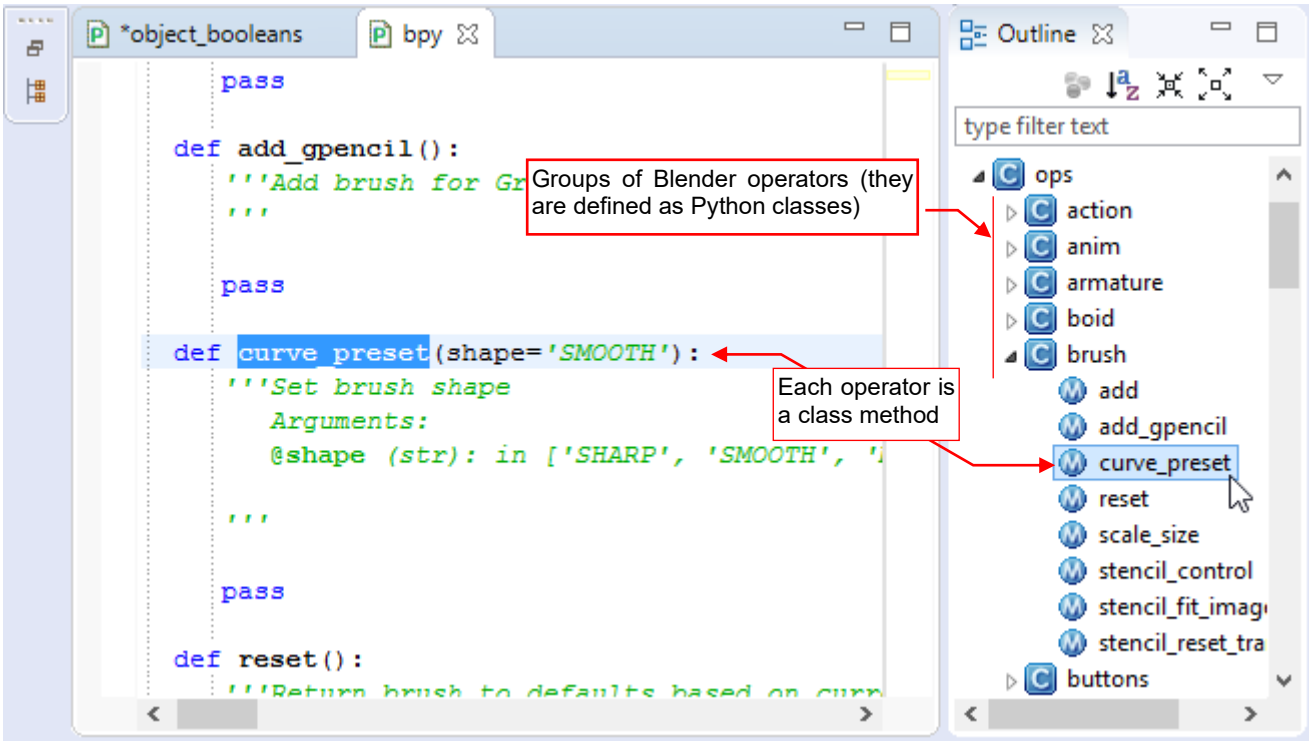


Figure 3.2.15 Operator declaration example

Each operator module (***bpy.ops.brush***, for example) is declared as a separate class that contains many methods. Each of these methods is a Blender operator. Note that every operator can be invoked without any parameters — because each of its arguments is named and optional (i.e. has a default value).

Because of the PyDev utilities like code autocompletion or the *Outliner* pane, this section has become an introduction to the Blender API architecture. Continuing the topic started on page 43, I have enumerated below the remaining API modules. They are much smaller than the main modules (***bpy.data***, ***bpy.context***, ***bpy.types***, ***bpy.ops***):

- ***bpy.app*** — various information about current Blender instance: version number, the path to the executable file, compiler flags, etc.;
- ***bpy.path*** — helper methods for working with paths and files (this functionality is similar to the ***os.path*** standard module);
- ***bpy.props*** — functions for creating new class properties, which Blender can display as the controls in the panels (when they are needed). To distinguish them from the ordinary class properties (fields), they are called "Blender custom properties" or just "custom properties". We will use them in the next chapter, in the operator class;
- ***bpy.utils*** — registration of Blender add-ons, importing Python modules, invoking other programs. Provides utilities for handling the path strings. Contains two additional submodules: ***units*** and ***previews***;
- ***bpy_extras*** — further auxiliary functions and classes. They are grouped into eight submodules: ***anim_utils***, ***object_utils***, ***io_utils***, ***image_utils***, ***keyconfig_utils***, ***mesh_utils***, ***node_utils***, ***view3d_utils***.

Apart the basic ***bpy*** section, Blender API offers additional modules:

- ***mathutils*** — classes representing some geometric and algebraic objects: ***Matrix*** (4x4), ***Euler***, ***Quaternion*** (rotation), ***Vector***, ***Color***. Contains also the ***geometry*** submodule with a few helper functions (line intersection, ray and surface intersection, etc.);
- ***freestyle*** — six submodules (***types***, ***predicates***, ***functions***, ***chainingiterators***, ***shaders***, ***utils***) for handling the auxiliary “sketching” (*NPR*) Freestyle renderer;
- ***bgl*** — functions that allow scripts to draw directly in the Blender windows. (In fact, it contains most of the OpenGL 1.0 methods. Preserved for the backward compatibility);
- ***gpu*** — another, more modern (and preferred) API for drawing directly in the Blender windows. Contains four submodules: ***types***, ***shader***, ***matrix***, and ***select***;
- ***bmesh*** — another API for mesh handling (precisely: *boundary meshes*). Contains four submodules: ***ops***, ***types***, ***geometry*** and ***select***;

I know little about the three remaining modules: ***aud*** (*Audio*), ***blf*** (*Font Drawing*), and ***idprop.types*** (*ID Property Access*), so they are not described in the list above.

Summary

- The Python predefinition files (**.pypredef*) allow PyDev to display Blender API code autocompletion. The predefinition files for all Blender API modules are included in the data accompanying this book (page 39);
- After unzipping the folder that contains the **.pypredef* files (*doc*), add its path to the PyDev project **PYTHONPATH** (page 40);
- To enable the Blender API autocompletion in your script, add the “***import bpy***” statement in the first lines of its code (page 41);
- You can use PyDev tooltips that display function descriptions and the *Outliner* pane for further exploration of the Blender API structure (page 41,42);
- The reference link in the tooltip window allows you to open the source **.pypredef* file in the text editor. It can be useful for examining the descriptions of a Blender API class fields, which PyDev does not show in the tooltip pane (page 44).
- In case of the elements from a Blender API collection, use "variable declarations" (page 44) to obtain the correct code autocompletion;

3.3 Developing the core code

In most of the programming guides, you would immediately see the script code in the section like this one. Their authors often present the ready solutions as if they were “pulling a rabbit from the hat", adding just some comments. This guide takes a different approach. I would like to show you here what happens before you write the first script line: the searching for the solution. This stage is even more important than the “pure” coding.

First prepare the test environment. For this purpose, in the initial Blender scene I transformed the default **Cube** object into a thick plate. To give it a more “mechanical” look, I rounded its edges using the ***Bevel*** modifier. Then I added to this scene a **Cylinder** object. I am going to use it as the “tool” in the Boolean operation, thus I switched its representation to *Wireframe* mode. I will conduct these tests in the standard *Scripting* workspace (Figure 3.3.1):

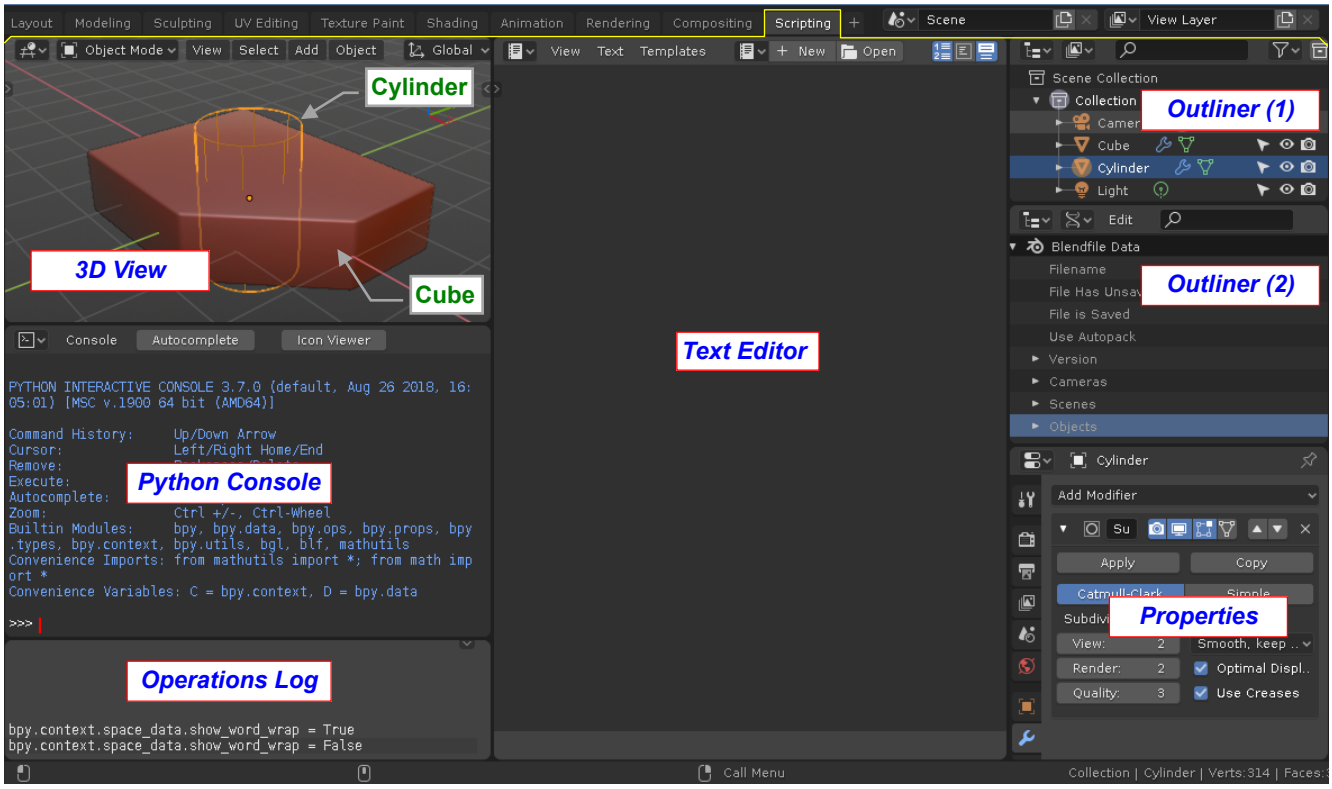


Figure 3.3.1 Screen layout for the “test environment”

Save this Blender file to the disk, and then import it to the PyDev project (using the ***Import..*** command — see details on page 145), so it will be “available on a single click” (Figure 3.3.2):

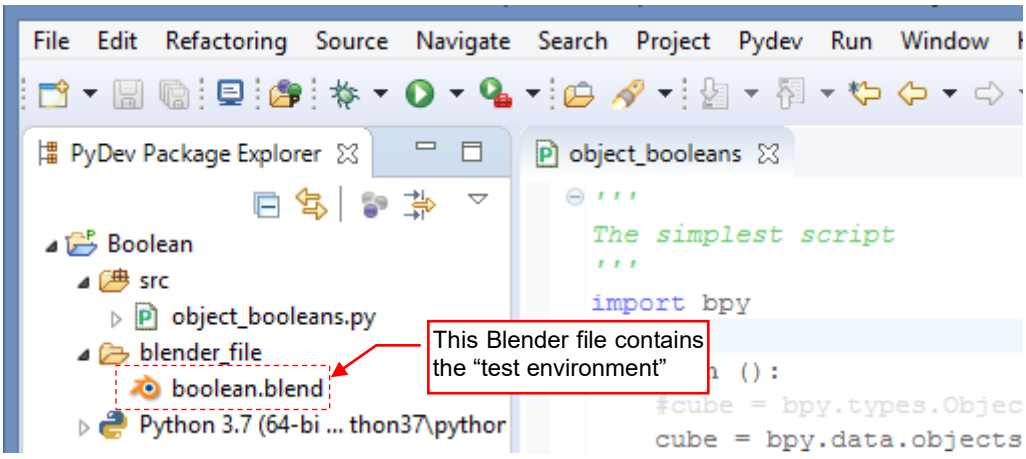


Figure 3.3.2 The test Blender file, added to the Eclipse project

The goal of this section is to prepare a code that automatically executes the steps described in section 3.1 (Figure 3.1.3 - Figure 3.1.6). When you invoke a Blender command, its equivalent Python API expression appears in the *Operations Log* window (see Figure 3.3.1, page 47). This is the best place to learn what you should put into your code. Let's begin by adding a *Boolean* modifier to the active object (Figure 3.3.3):

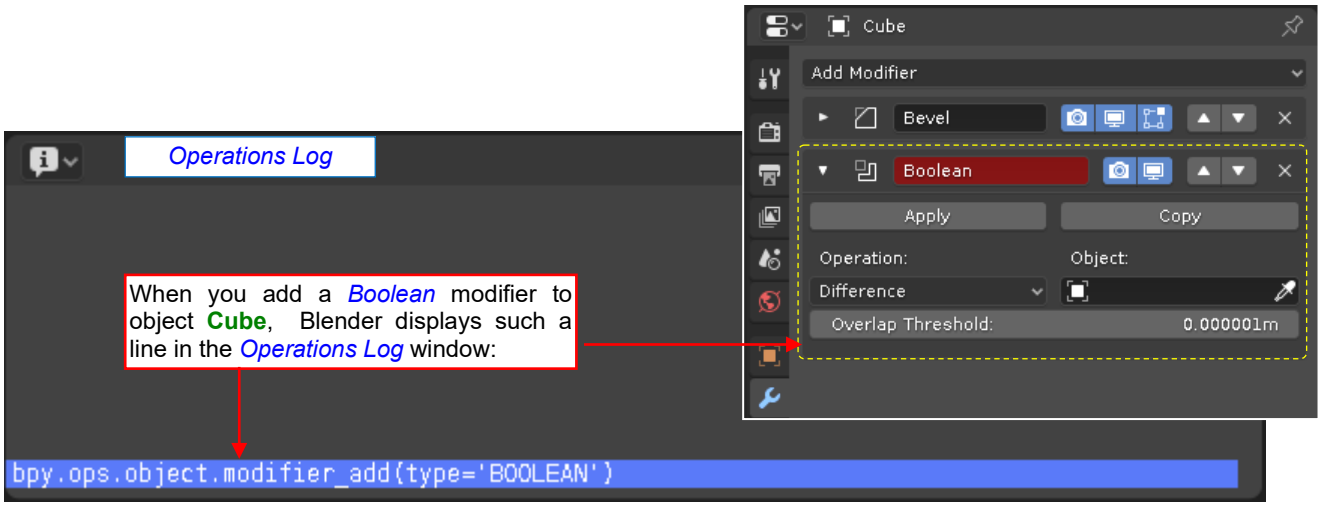


Figure 3.3.3 Adding a *Boolean* modifier (to the active object)

Similarly, after the next step – moving the *Boolean* modifier to the top of the modifiers list – a new line appears in the *Operations Log* window (Figure 3.3.4):

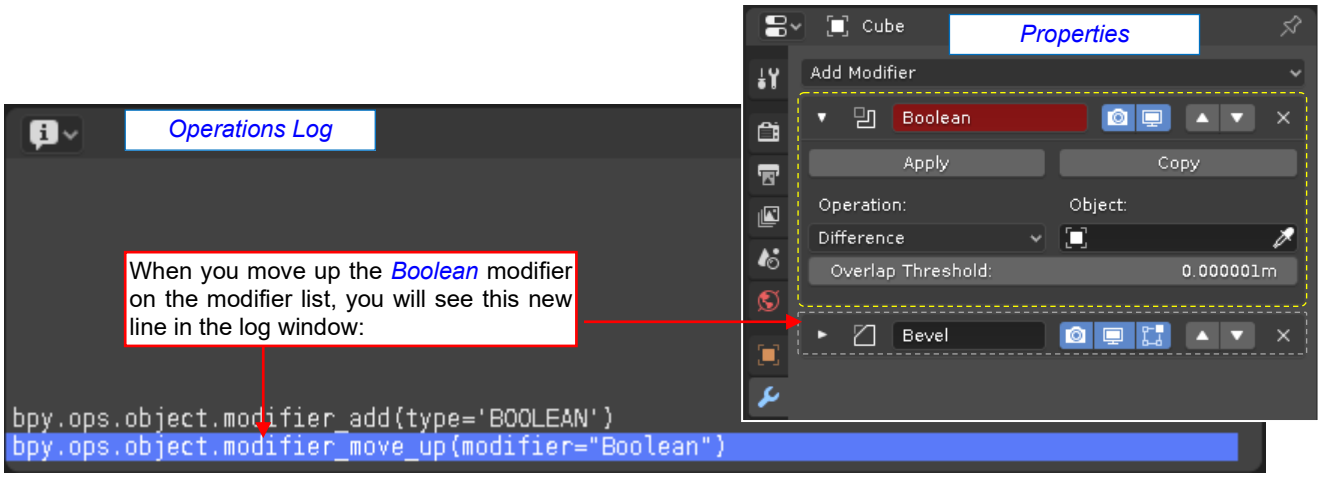


Figure 3.3.4 Moving the *Boolean* modifier upward on the modifier list

In the third step I assigned object **Cylinder** to the *Boolean* modifier, pointing it directly on the screen (using the "pipette" tool – (Figure 3.3.5):

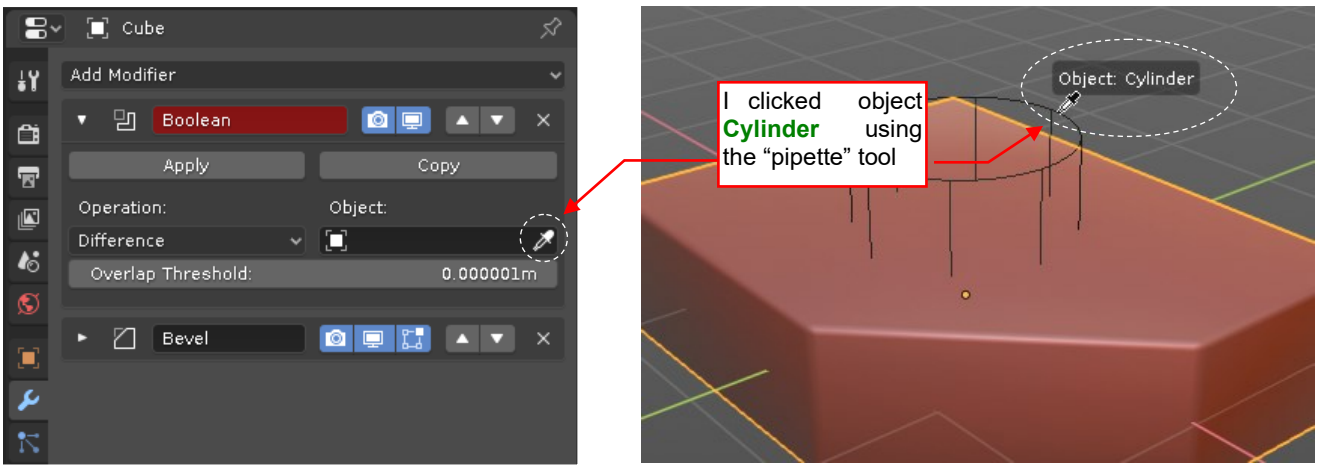


Figure 3.3.5 Assigning Cylinder as the *Boolean* modifier "tool" object (using the mouse)

This did not cause Blender to add any new log line. However, when I manually typed the object name in the *Object* field, *Operations Log* displayed corresponding expression (Figure 3.3.6):

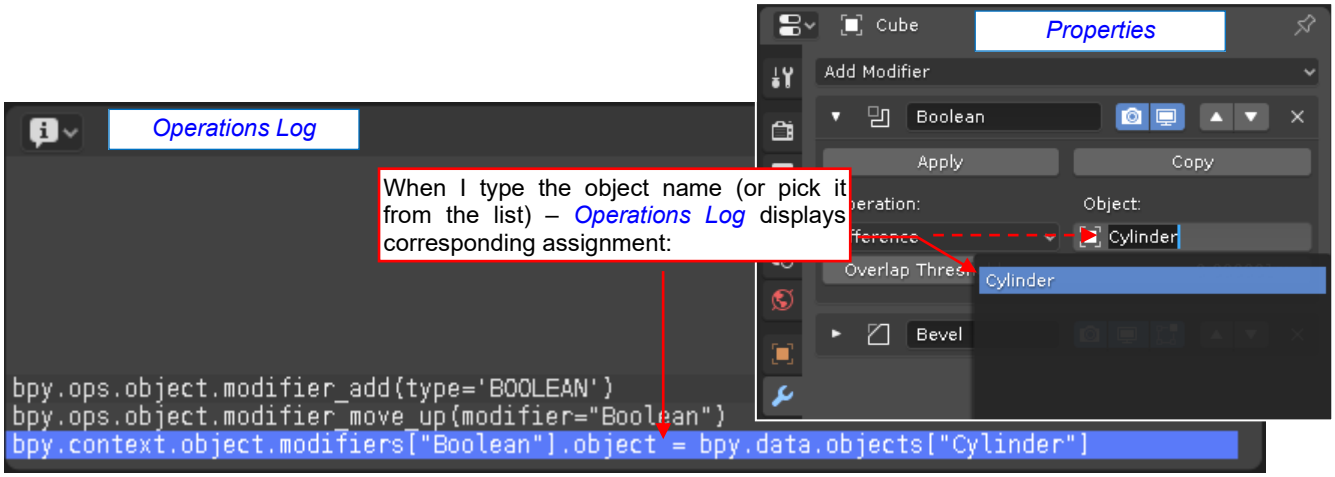


Figure 3.3.6 Assigning Cylinder as the modifier "tool" object (by typing the object name)

While the API code in the previous lines contains procedure calls, this one is an assignment. Let's look at it to learn what is happening here. The ***bpy.context*** object provides information about the "execution context" of your code: which objects are selected, the window where the user invoked your script, etc. The ***bpy.context.object*** property returns the active object (the last scene object that you have clicked). In this line Blender uses the modifier named **Boolean** (***modifiers["Boolean"]***). It assigns to the ***object*** field of this modifier (this is the "tool") a reference to another scene object named **Cylinder** (***bpy.data.objects["Cylinder"]***).

Note how Blender refers to (scene) object **Cylinder**. In the log line it is represented by an element of the list named ***bpy.data.objects***. The scene object name (you can type/edit it in Blender *Properties* panel) is the index (or "key") of this list. In fact, in Blender API this is the preferred method of referencing all types of the scene datablocks. Module ***bpy.data*** provides your scripts contents of the current Blender file as lists (precisely: iterators). Its ***objects*** list contains objects from all scenes defined in this file. The names of these objects are unique, so they are used as the convenient keys. (When you try to type in the *Name* field of a scene object an identifier that is already used by another object, Blender automatically corrects it. It adds a numerical suffix to this name).

- All Blender data (*datablock*) types have their lists in module ***bpy.data***: ***meshes[]***, ***materials[]***, ***textures[]***, scene object collections (***collections[]***), etc. You can "pick" (i.e. refer) to any element from these lists using the datablock name as the list key.

The last step in the executing sequence is clicking the ***Apply*** button of the Boolean modifier (Figure 3.3.7)

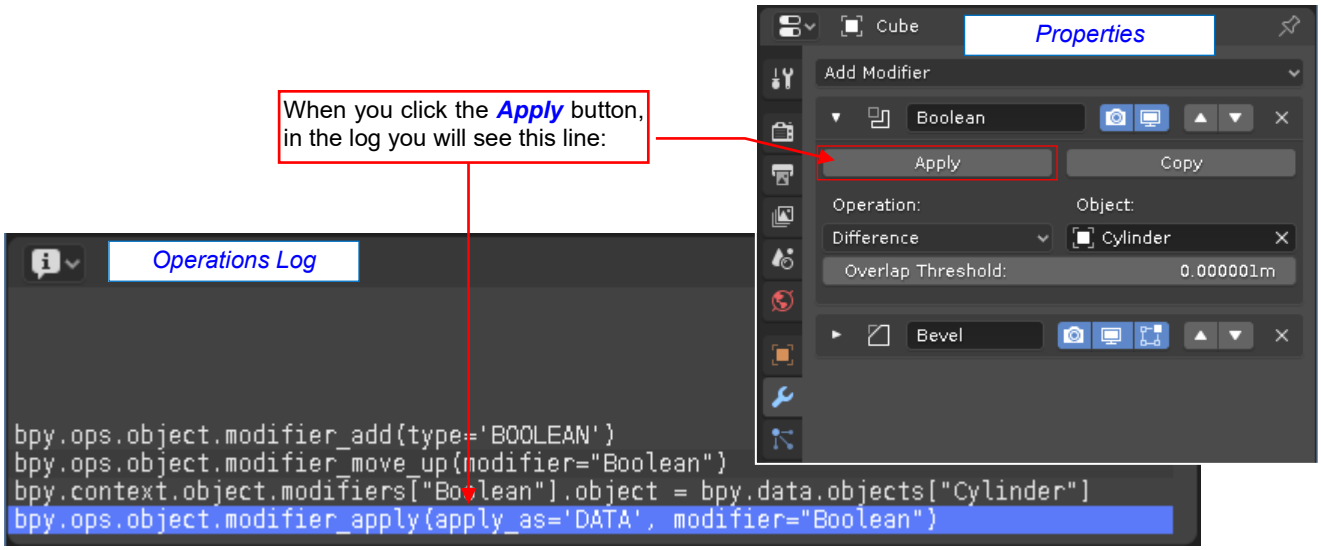


Figure 3.3.7 "Applying" the Boolean modifier to the object mesh

As you can see, Blender wrote the basic code of the script in the *Operations Log* window. Now you can copy this text to the clipboard: just select it with the mouse and press Ctrl-C (Figure 3.3.8):

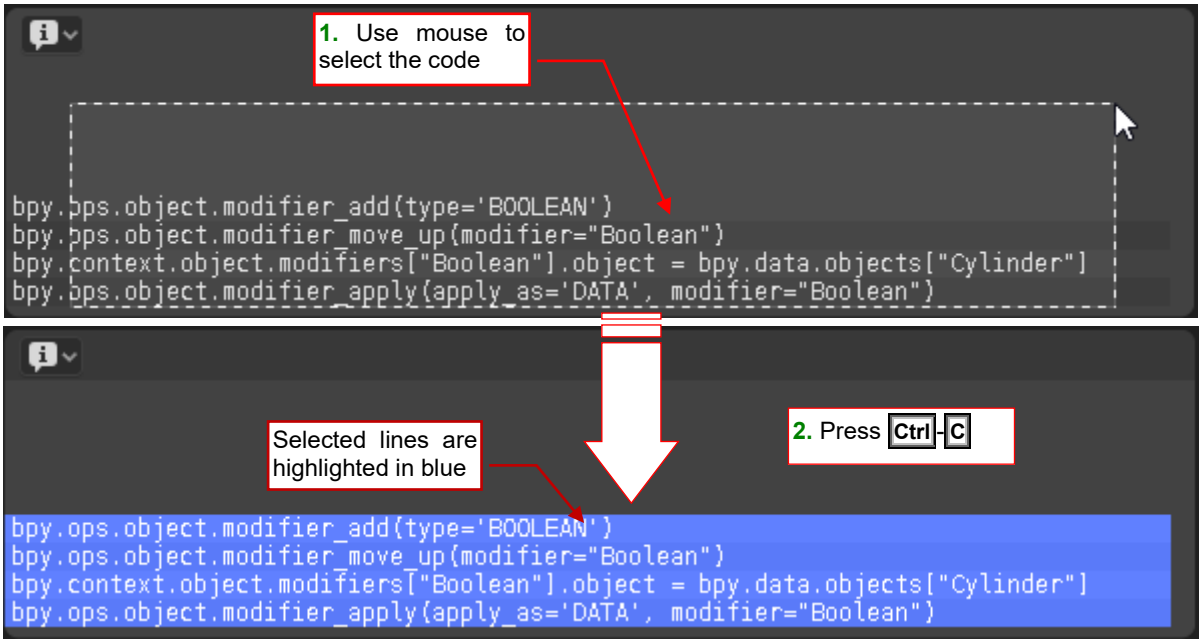


Figure 3.3.8 Copying Blender log lines to the clipboard

Then you can paste this code to Eclipse text editor (Figure 3.3.9):

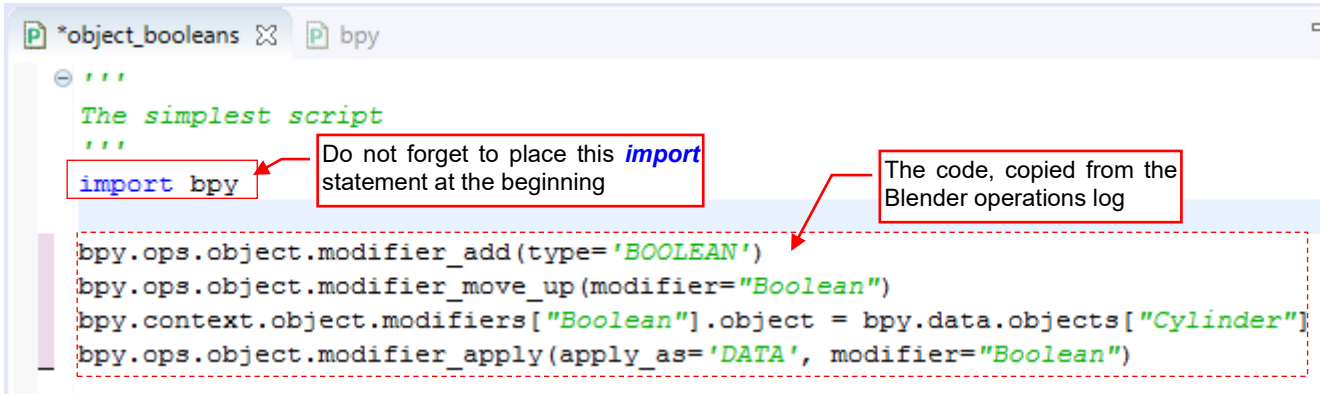


Figure 3.3.9 The simplest API script: lines copied from Blender *Operations Log*

This is the first approximation of the code we need. Do not forget to add above the "*import bpy*" statement. Otherwise Eclipse will mark errors in all these lines (Figure 3.3.10):

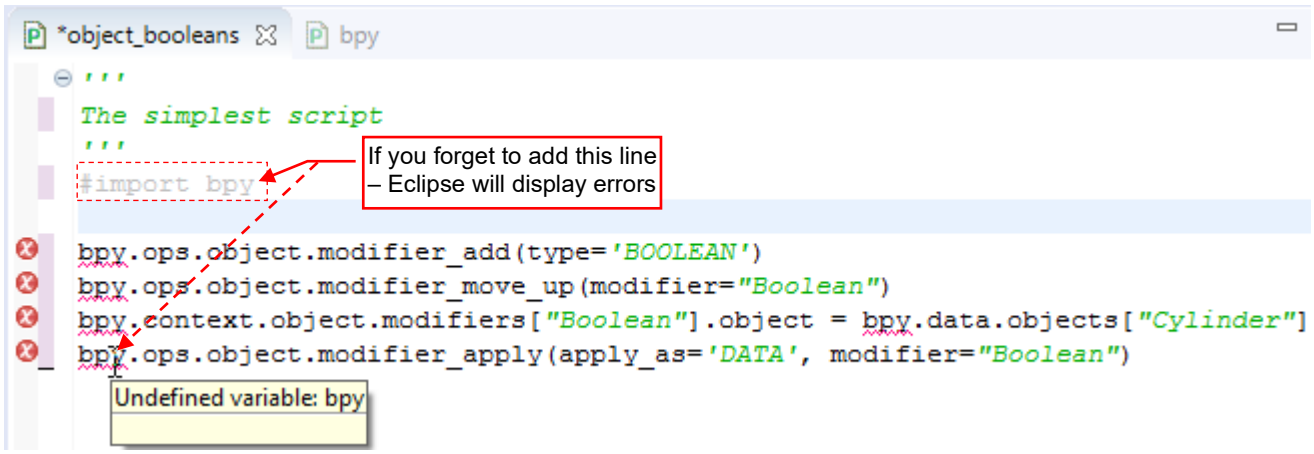


Figure 3.3.10 Effects of the missing *bpy* module import statement

They would also occur in Blender. Always keep your script in Eclipse free of any error.

The script from Figure 3.3.9 works only for the specific data which I prepared in the test **.blend* file. Before we run it for the first time, let's change the structure of this code. In this way I am preparing this script for further modifications (Figure 3.3.11):

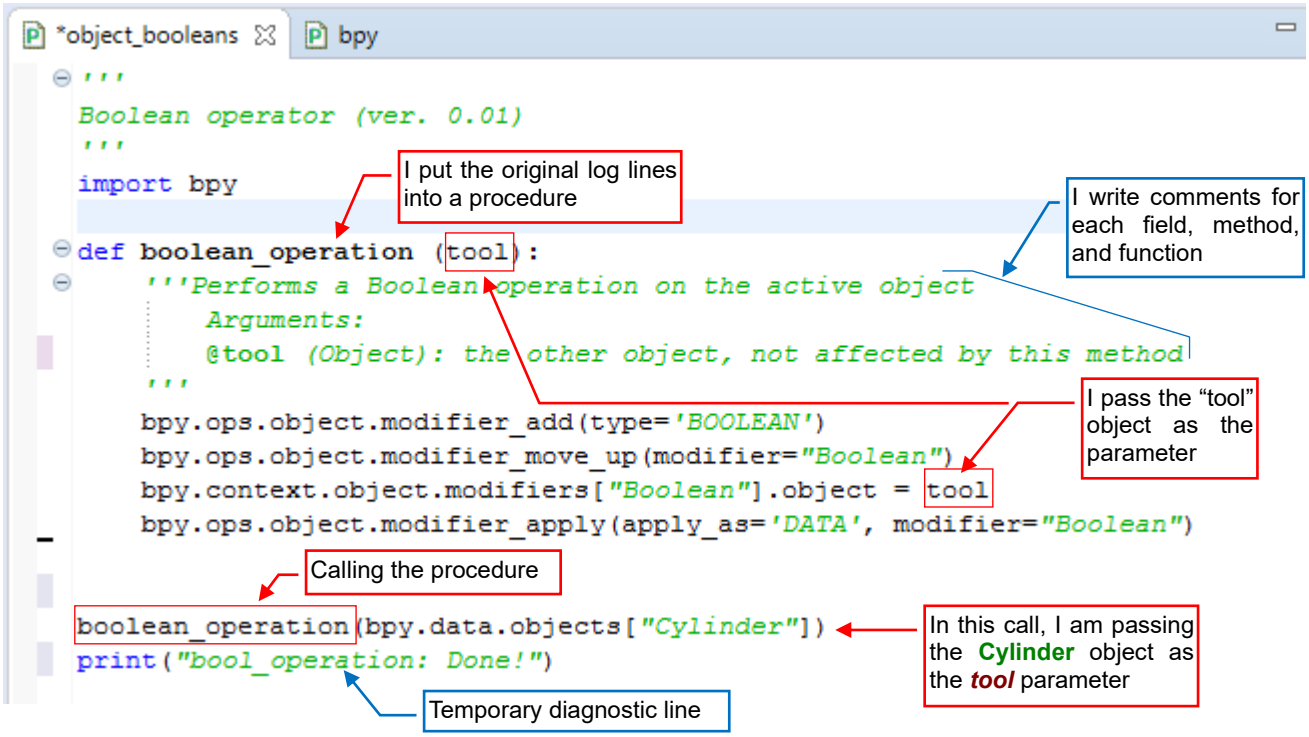


Figure 3.3.11 The same code, organized into more "professional" structure

I placed the original code into a procedure named ***boolean_operation()***. In this way the main script will be shorter and more readable. In the future, this plugin will use user-selected objects as the "tools" for a Boolean operation. That's why this procedure has a single argument: the ***tool*** object. Inside ***boolean_operation()*** I assign it to the modifier. For the first test, in the second-last line of this script I simply call this procedure, passing object **Cylinder** as the ***tool*** parameter.

In the last line I placed the ***print()*** command. It displays the text in the console. Sometimes such a statement can be useful: it allows you to quickly determine if the script execution has been successfully completed. In the next section the presence of this line will allow me to show how the debugger leaves the procedure call (unlike as in the case on page 31)

By the way: as you can see at the beginning of the procedure, I already added there a few comment lines that describe shortly: what it does and what arguments it expects. This is my "good practice". Despite appearances I do it for myself. Writing such a comment forces me to re-think if this procedure is necessary, and if its arguments match its task. Because Python is a language with "flexible types", I always specify in such a description what argument type/types it expects and other assumptions. These comments allow me to write a cleaner code. (From my experience, 50% of runtime errors in the scripts is caused by passing a wrong parameter to a function or method. You can avoid most of them if you read the function/procedure descriptions that PyDev displays in the tooltip window). I also have no illusions about my memory: I will remember nothing when I return to this script after a year or two. It is always frustrating to learn the same things anew. In this case my comments help me in quick recalling the nuances of such a "forgotten" code. I started commenting my programs during my student years, and since that time this habit saved me many troubles.

Summary

- I prepared in Blender a test environment for the script: the file named *booleans.blend*. It contains two objects – **Cube** and **Cylinder**. For the script development in Blender we will use the standard *Scripting* workspace (page 47);
- It is convenient to place the test Blender file in a folder of your Eclipse project (page 47);
- In the *Operations Log* window, you can see the Python API code for the Blender commands that you are invoking (page 48);
- *Operations Log* does not display the API code when you pick an object using mouse. However, it displays a Python statement when you do the same by typing the object name in the corresponding field (pages 48, 49);
- The code from *Operations Log* can be an excellent information source. In the simple cases, like this one, the key lines of the script are "written by Blender" itself. All what you have to do is to copy them to clipboard and paste into Eclipse (page 50);
- The Blender API predefinition files (the ***bpy*** module and the others) allow PyDev IDE to highlight various errors and warnings in your code (page 50). Fix all these issues before running your script in Blender;
- The basic information sources for your script are two objects:
  - ***bpy.data***, which provides all the data from the current Blender file;
  - ***bpy.context***, which provides information about script "environment", for example - selected objects;
- The preferred way to access the Blender file data are the ***bpy.data*** lists. They use datablock names as their keys (indices) (page 49);

3.4 Launching and debugging Blender scripts

In the previous section we have written the first piece of the script that should work in Blender. You could launch it by loading this file into the Blender *Text Editor* and invoking the *Run Script* command. However, it would be difficult to debug your script this way. What's more, it brings some confusion about the source files. (If you changed something in the Blender *Text Editor*, you would have to remember to save it back to the disk).

I suggest another, more convenient solution. Open in the Blender *Text Editor* the *Run.py* file that accompanies this book (see page 39) (Figure 3.4.1):

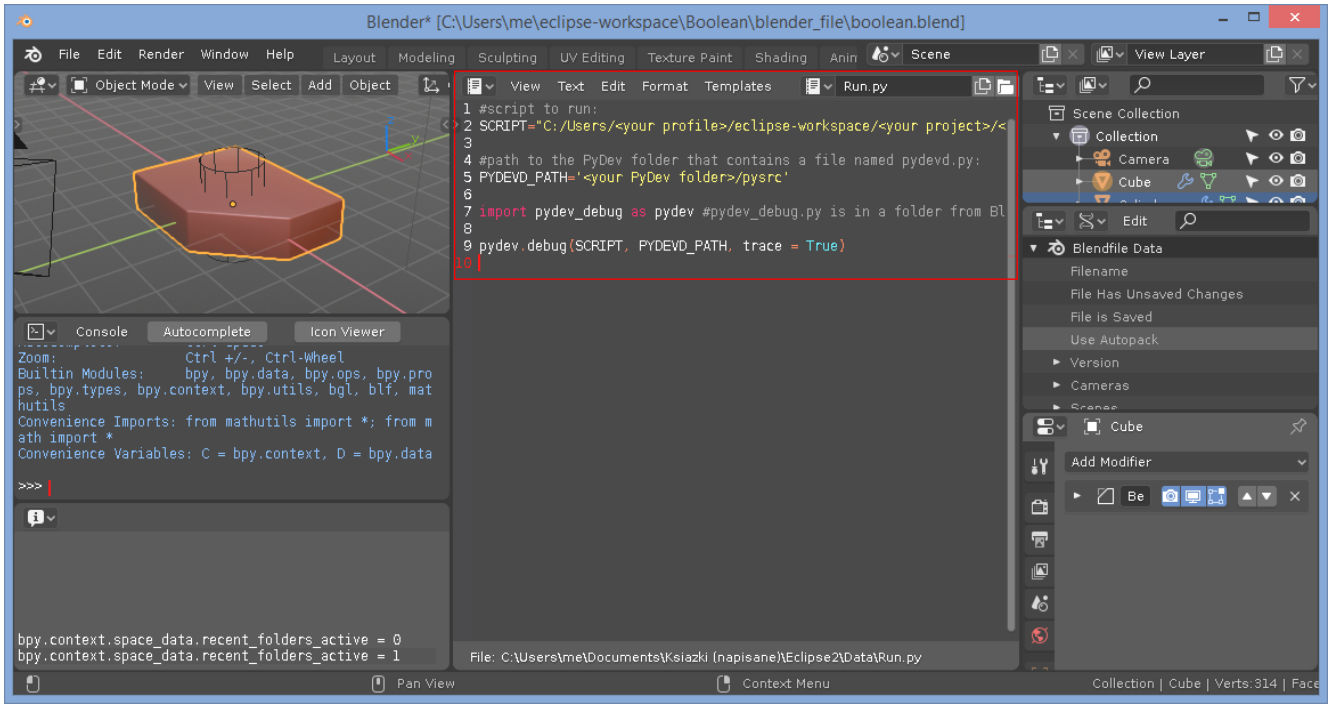


Figure 3.4.1 Adding the *Run.py* script to our Blender test file

Run.py is a “stub” script, containing just a few code lines. To adapt it for your project, update values of its ***SCRIPT*** and ***PYDEV_PATH*** constants (Figure 3.4.2):

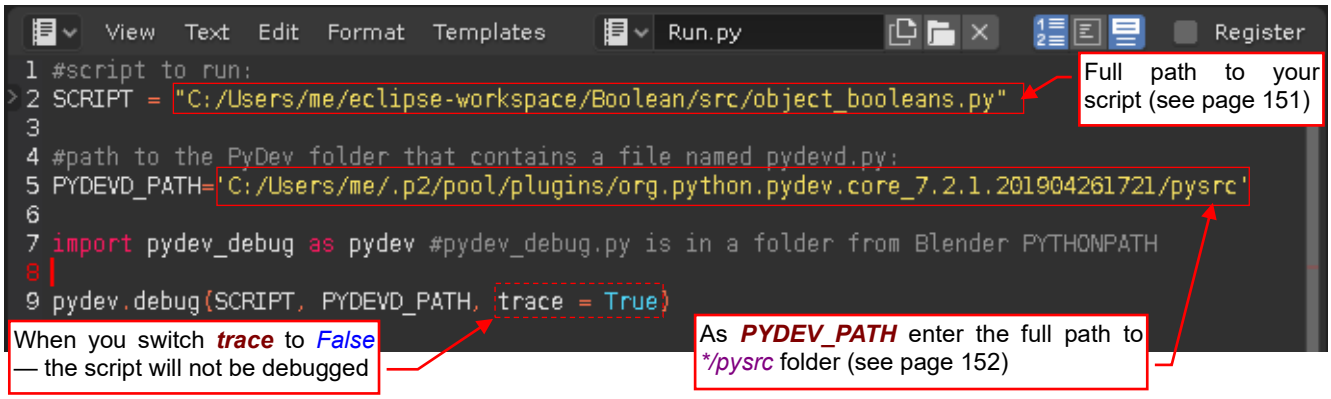


Figure 3.4.2 Adaptation of the *Run.py* code to this project

The ***SCRIPT*** constant should contain the full path to your script file (for details how to find it – see page 151).

The ***PYDEV_PATH*** is the full path to a certain PyDev subdirectory, named *pysrc*. In this folder you can find the *pydevd.py* module, which contains so-called remote debugger client (see pages 149 and 160 for more information). Actually (in 2019) you can find the main PyDev directory in the current user profile (*C:/Users/me/* in Figure 3.4.2), in the *.p2/pool/plugins* subdirectory. However, in the future PyDev versions this location can change, so in case of troubles see page 152, where I suggest how you can find it.

Make sure that the test scene is also ready for the run. At this moment the first version of our code assumes that the active object is **Cube**. That's why it is selected (Figure 3.4.3):

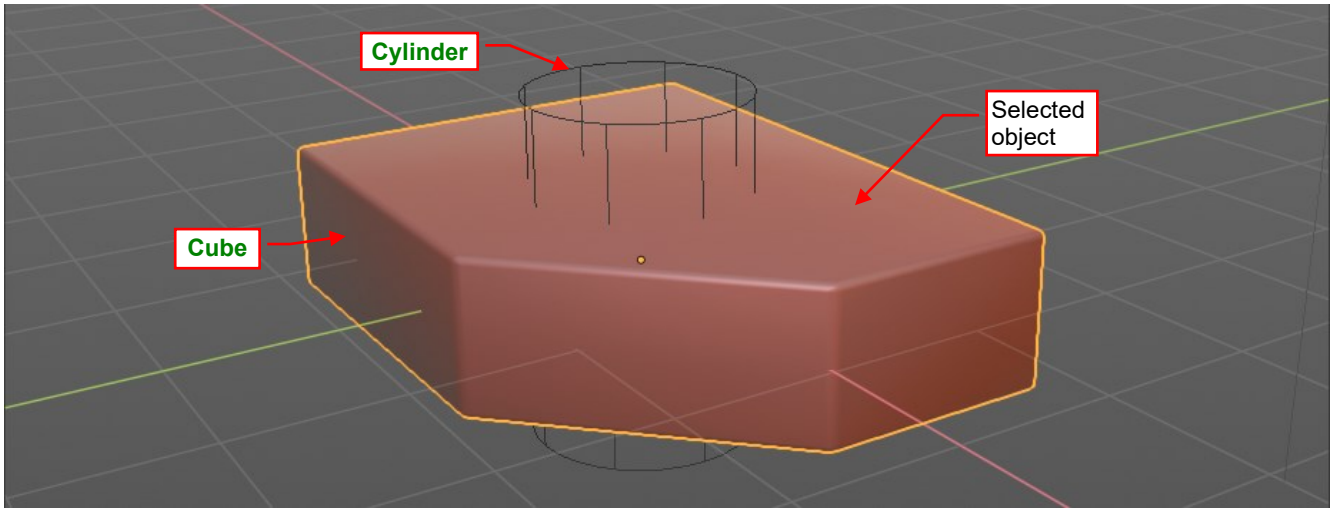


Figure 3.4.3 Preparation of the test environment – I selected object Cube **(now this is the active object in this scene)**

Insert a breakpoint in your script where you want to start debugging. In this case, I set it at the first statement (Figure 3.4.4):

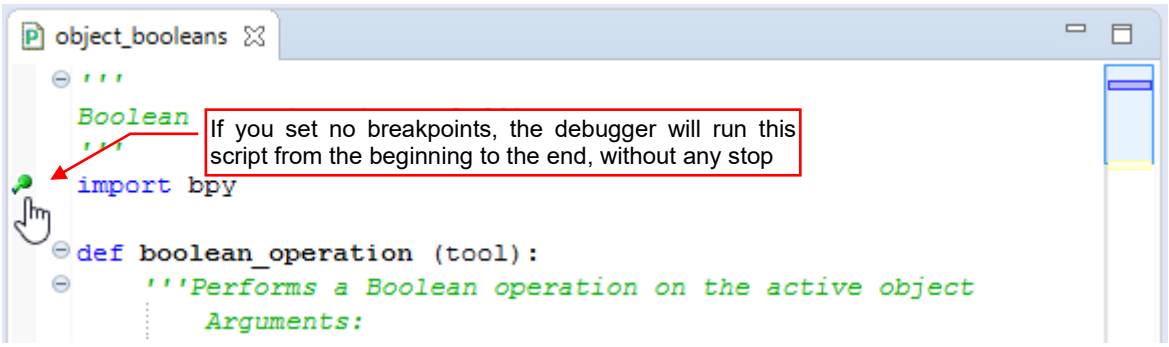


Figure 3.4.4 Setting the breakpoint

Launch from Eclipse (*Debug* perspective) the remote debugger server (details — see page 149) (Figure 3.4.5):

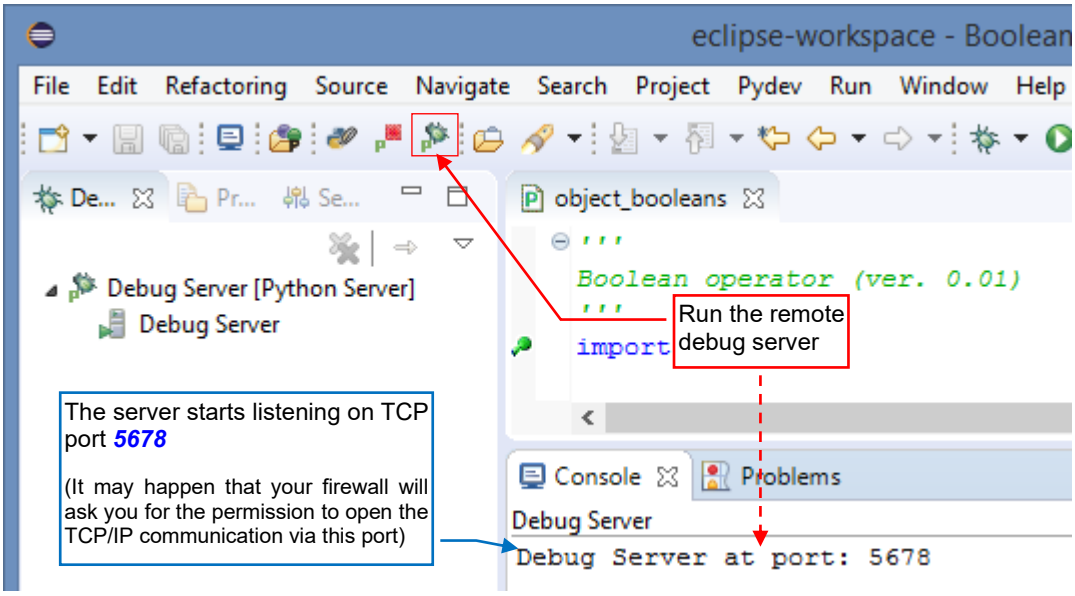


Figure 3.4.5 Launching the debug server process

For further details about this step — see page 153. If you cannot find this button on your toolbar — see pages 149, 150.

When the debug server displays its “listening” message in the console, you can run the Blender script. Click the ***Run Script*** button, located in the *Run.py* window header (Figure 3.4.6):

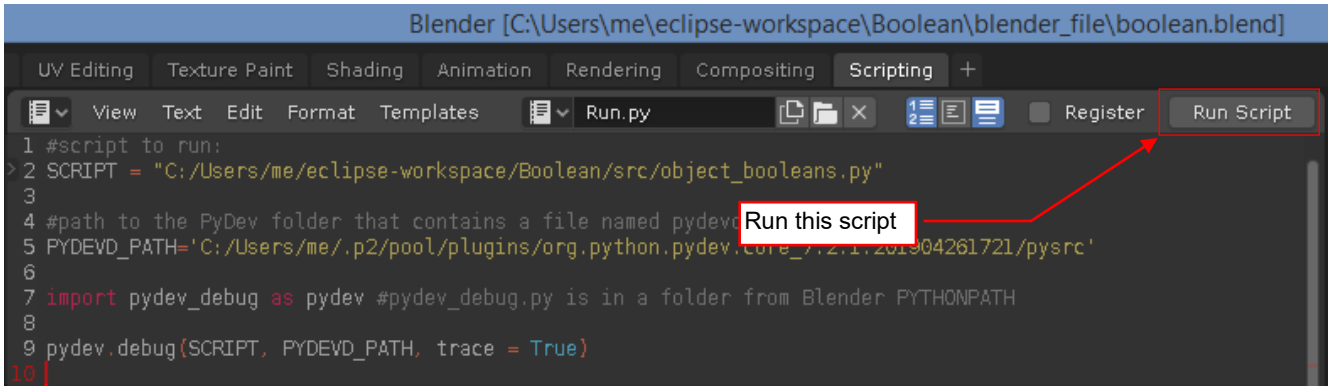


Figure 3.4.6 Launching the Blender script for the PyDev debugger

This code loads the current version of your file (from the path specified in the ***SCRIPT*** constant) into the debugger. From this moment Blender window “freezes” until you finish this debug session. It is not being updated and does not react to the mouse clicks. The mouse cursor in Blender window shows the “wait” icon.

Now click the Eclipse window. After a few seconds the debugger pane "comes to life", and the debugger stops at the first breakpoint (see page 54) (Figure 3.4.7):

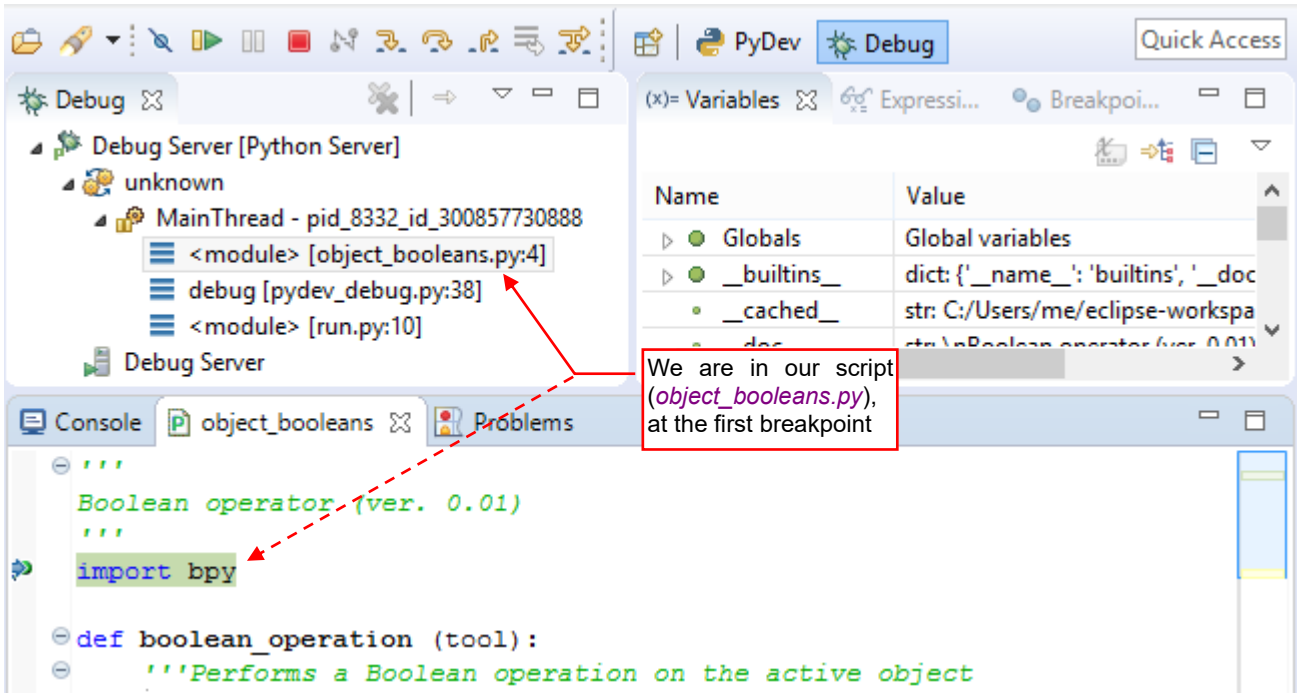


Figure 3.4.7 The first breakpoint

Step Over (F6) the lines of the script main code until you reach the ***boolean_operation()*** procedure call (Figure 3.4.8). Then ***Step Into*** (F5) this procedure:

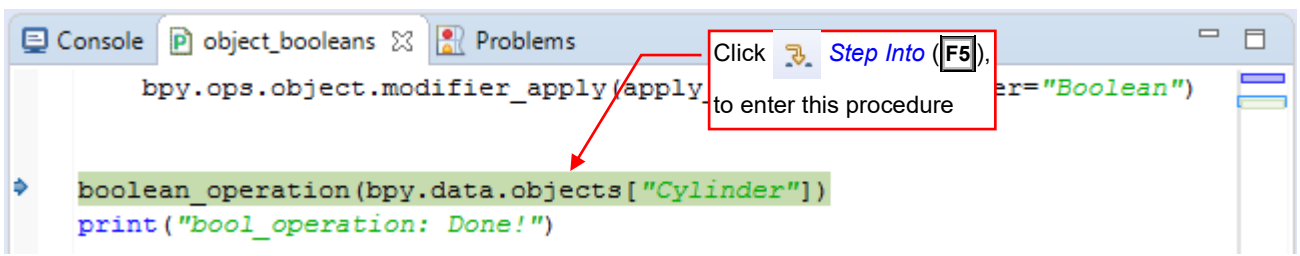


Figure 3.4.8 Stepping into the *boolean_operation()* procedure

In response, PyDev enters the ***boolean_operation()*** procedure and stops on its first line (Figure 3.4.9):

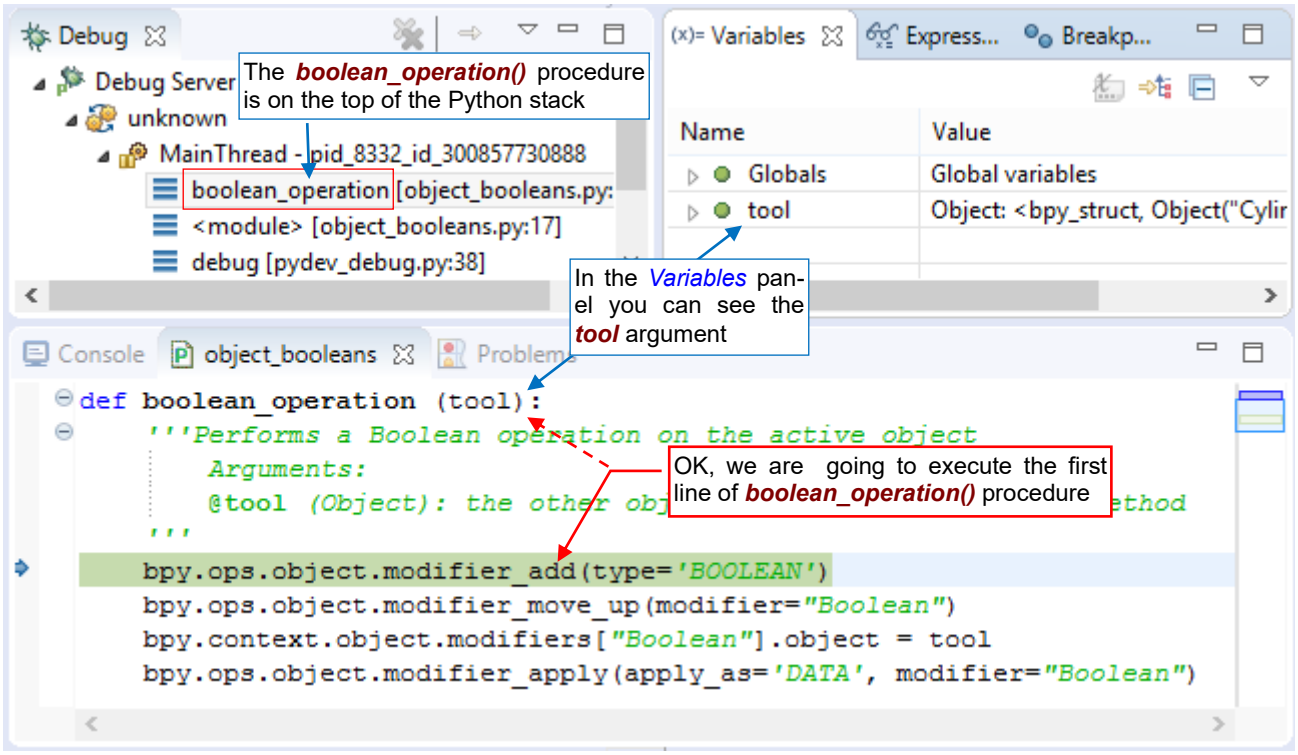


Figure 3.4.9 Executing the code inside the *boolean_operation()* procedure

To keep track of the **Boolean** modifier fields, use the ***Expressions*** panel (Figure 3.4.10 — see also page 155):

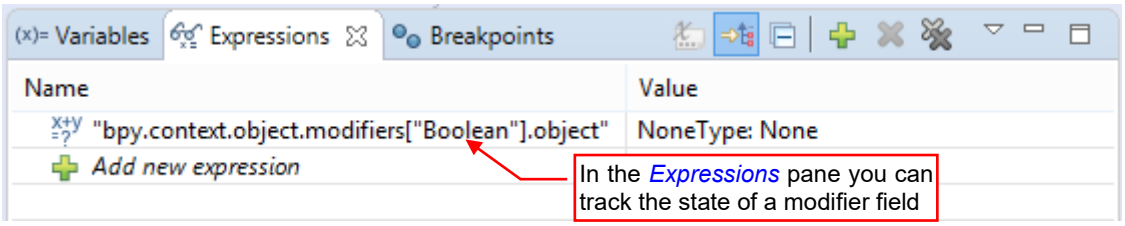


Figure 3.4.10 Tracking the selected fields in the *Expressions* tab

When the procedure is completed, click the *Resume* button (F8) to finish this debug session[1] (Figure 3.4.11):

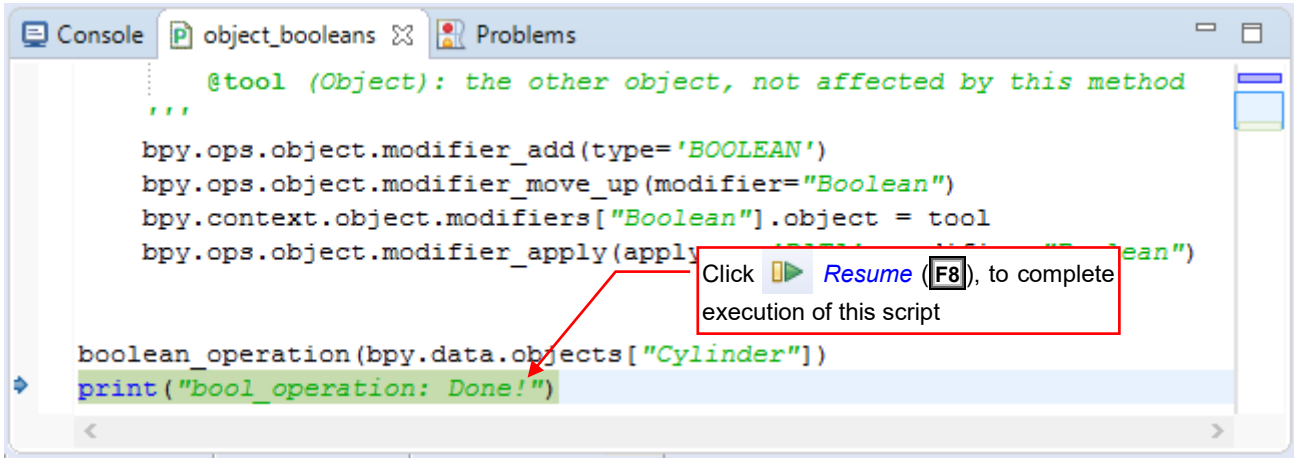


Figure 3.4.11 Resuming script execution (at the last line of the script)

[1] If you *Step Over* the last line of your script (in this example the *object_booleans.py* file), Python will drop it from the call stack. The debugger will stop on the next line of the auxiliary module used by *Run.py* for loading your code (see page 138). We have nothing to do there, thus I suggest to *Resume* this execution. It will end this debug session in a controlled way.

In the console you will see the text printed by the last (diagnostic) line in our script (Figure 3.4.12):

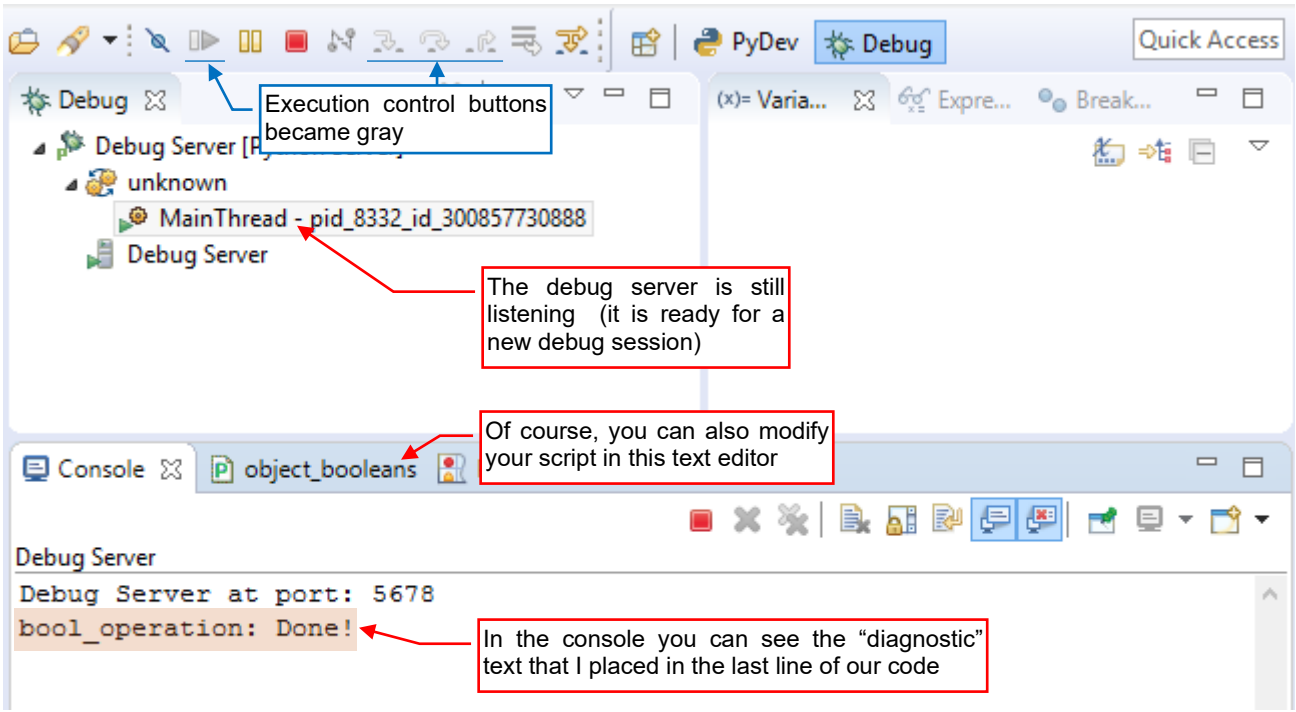


Figure 3.4.12 The state of the PyDev environment after the last *Resume* command

There were no runtime errors (so far). Let's look at its results in the *3D View* window (Figure 3.4.13):

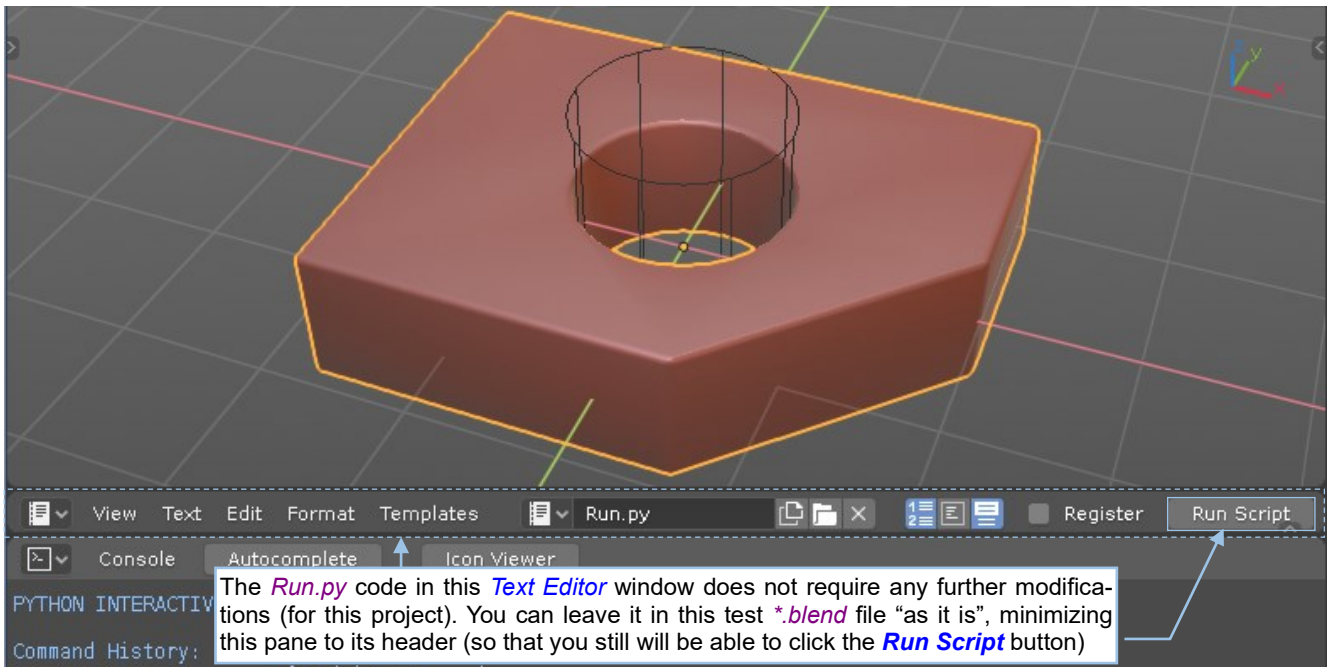


Figure 3.4.13 The results of our script

As we have intended, the object **Cube** mesh contains a hole, "drilled" by **Cylinder**. So, this script works properly. To re-use this test data in the next run, undo the result of this last operation. (Just invoke the *Edit→Undo* command or press Ctrl-Z).

For this project, I will not do any further modifications in the *Run.py* code, loaded into Blender *Text Editor*. I will just click its *Run Script* button to start a new debug session for the updated version of *object_booleans.py* script, modified in Eclipse. That's why I minimized this *Text Editor* pane just to the height of its header, so that the *Run Script* button is still available (as in Figure 3.4.13). Then I saved the test **.blend* file, preserving this modified layout of the *Scripting* workspace.

Our first run was successful, but how the debugger terminates the script execution in the case of a runtime error? To check this, just select **Cylinder**, making this object active (Figure 3.4.14):

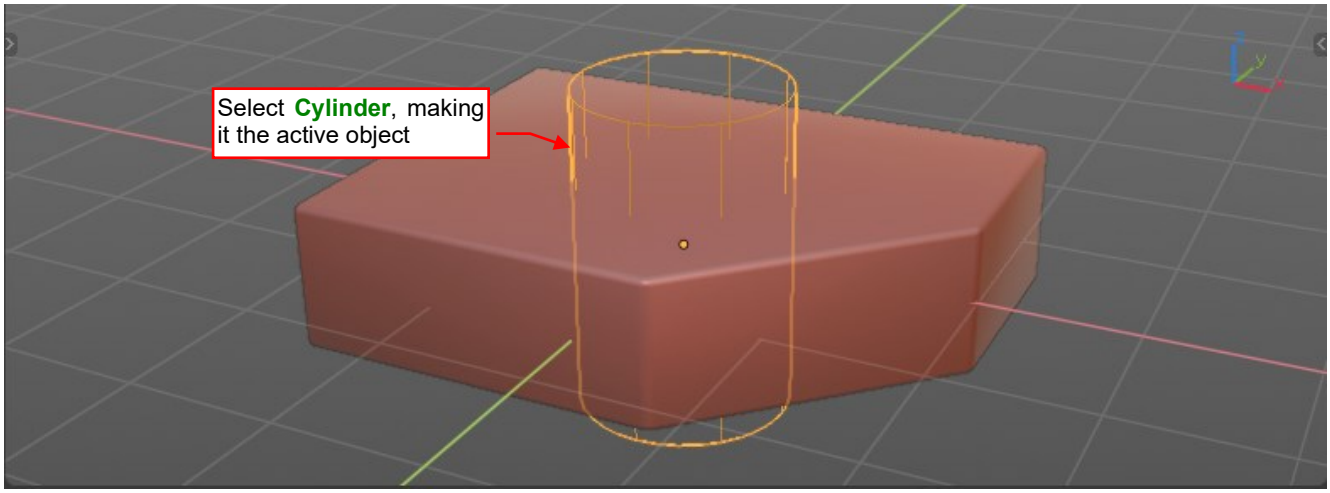


Figure 3.4.14 Preparing of a different input for our test

To debug again the script, just click the ***Run Script*** button in the Blender *TextEditor* header[1]. As long as the PyDev debugger server process is "listening" the TCP port, it automatically breaks the script execution at the first breakpoint.

Initially you will stop in the same place as in Figure 3.4.7 (page 55). Execute the subsequent script lines and step into the ***boolean_operation()*** procedure. Stop on the line that assigns the ***tool*** object to modifier **Boolean** (Figure 3.4.15):

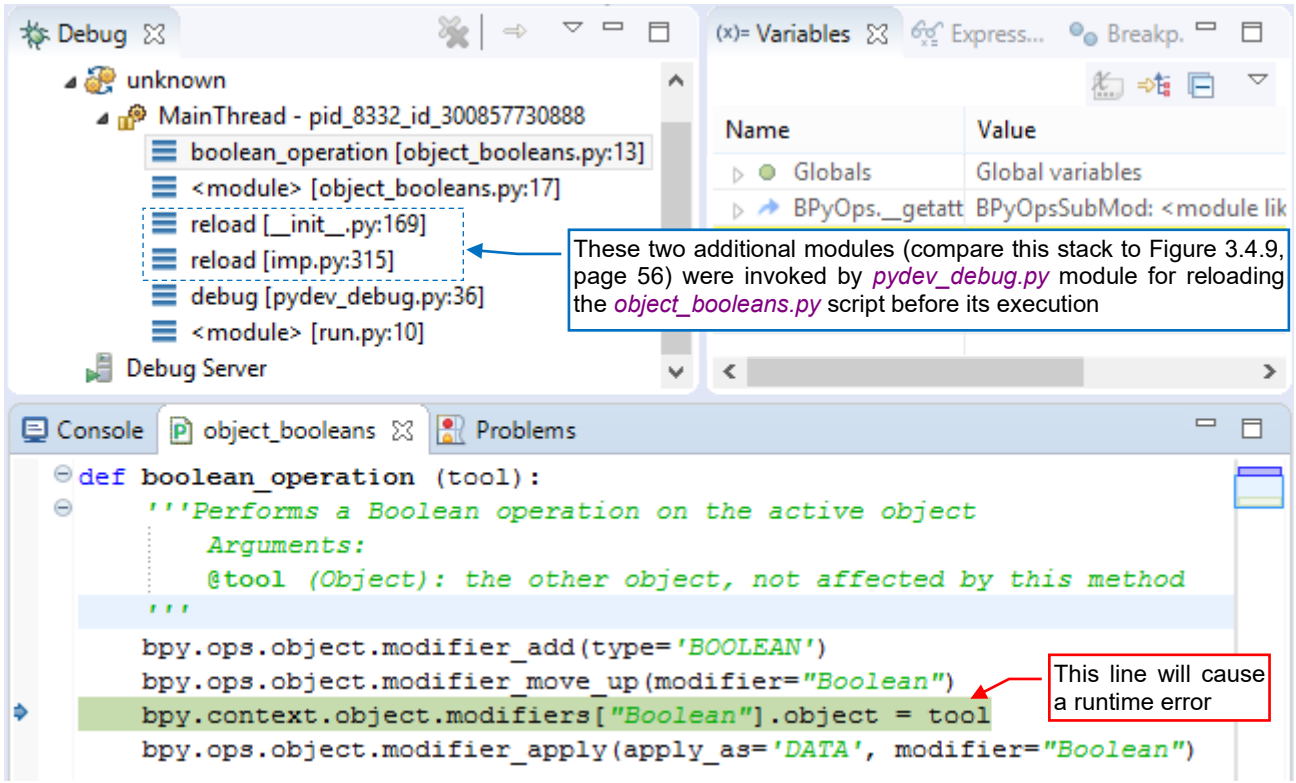


Figure 3.4.15 The "tool" assignment line

The highlighted line in Figure 3.4.15 will cause a runtime error.

[1] The *Run.py* code that you run clicking *Run Script* button takes care of reloading the current version of your script before it starts its execution. If you have just modified it in Eclipse, simply save it before clicking the run button in Blender.

Just ***Step Over*** (F6) it and see what happens (Figure 3.4.16):

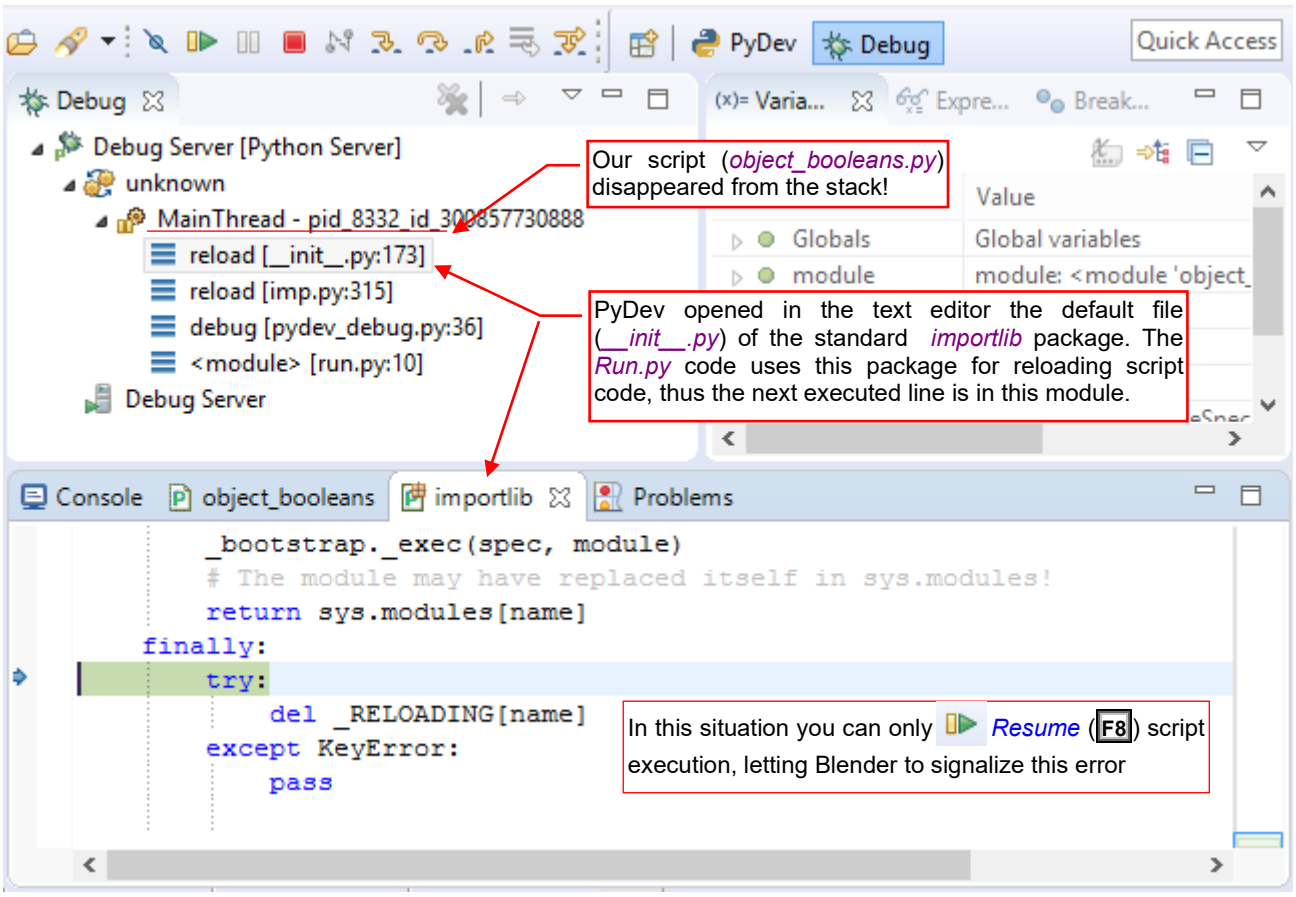


Figure 3.4.16 The state of the PyDev debugger just after a runtime error

When the runtime error (exception) occurs, Python removes the source *object_booleans.py* module from the call stack. (Compare the stacks shown in Figure 3.4.16 and in Figure 3.4.15). Thus, we lost the chance for inspecting the local variables for their values in that very moment. All what we can do now is to *Resume* (F8) the execution of this script. It will allow Blender to handle this exception and show the standard traceback information as well as the error message in the console (Figure 3.4.17):

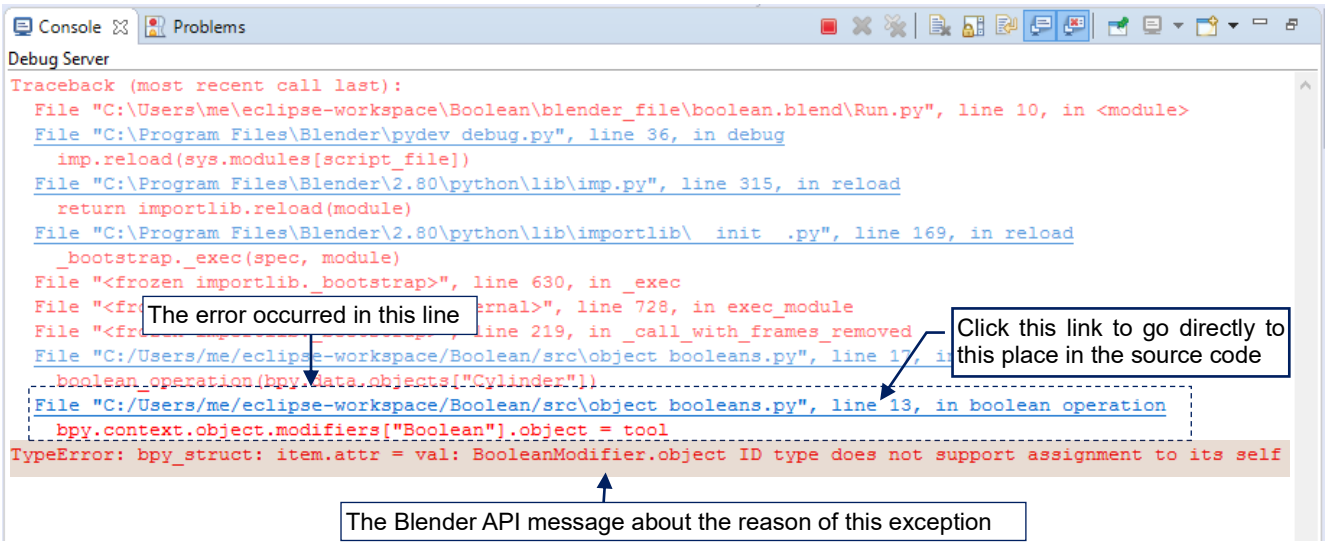


Figure 3.4.17 The call stack traceback and the error message, displayed in the console

The most important information you will find in the last lines of this output. There is a direct link to the line that caused this error, and the Blender API message about the cause of this exception. In this particular case – the script tried to assign itself (object **Cylinder**) as the tool object in the *Boolean* modifier.

- The runtime exception (error) suddenly occurs in an unexpected place of our script. To check the state of the local variables of a procedure/function where it happens, place its code into ***try:*** … ***finally:*** statement

If you wish to turn off the remote debug server in Eclipse, close first the connected process (i.e. Blender).

- Once you have run the debug server, just keep it running for the whole time. It will close automatically when you exit Eclipse IDE.

The auxiliary module *pydev_debug.py* (see its code on page 160), used by *Run.py* to load the script file, searches the **PYTHONPATH** for its base name. It works this way, because in this code I used the ***__import__()*** and ***imp.reload()*** methods for script execution. In the first lines of *pydev_debug.py* the path to your project is added at the very end of the **PYTHONPATH** list[1]. This means that if another copy of your script exists in any of the Blender **PYTHONPATH** directories, it will be loaded and executed instead of your file, and your breakpoints will be ignored. This can be really confusing!

- Never use for your script file the name of any standard module or a registered Blender add-on, because their directories occur earlier in the **PYTHONPATH** than your project directory. (You can examine the ***sys.path*** list in Blender *Python Console* to learn about these directories and their order).

For example: one of the standard Python modules is named ***test***. If you name your script *test.py*, then the *Run.py* code will properly load (import) this standard module, instead of your script! From your point of view – nothing will happen when you click the *Run Script* button, and you will not know what is going on, because there will be no error message. (I lost three hours before I discovered that the problem is in the script name).

The bet practice is to follow the Blender convention for the add-on names, and name them using the mode/window prefix: *object_**, *mesh_**, *uv_**, etc.

Summary

- To run a script in the PyDev debugger, use the *Run.py* stub code. Place it in the Blender *Text Editor*. Save this Blender file as the test environment for your PyDev project (page 53);
- Before the first run, modify the string constants in the *Run.py* code. Place there the path to your script (in ***SCRIPT***) and the path to the PyDev remote debugger client module (in ***PYDEV_PATH***) (page 53);
- To start the first debug session, activate in Eclipse the *PyDev Debug Server* (page 54), then click the *Run Script* button in Blender (page 55);
- To start every subsequent debug session just click again the *Run Script* button in Blender (page 57, 58);
- To track changes of selected object properties, use the *Expressions* window (page 56);
- When you reach the last line of your script – click *Resume* (F8) to complete its execution (as in page 56). If you *Step Over* (F6) this line, Eclipse will open the window with an auxiliary code, which is used by *Run.py* for running your script. (There is nothing important to debug there);
- When an error (runtime exception) occurs in your script, Python closes your module and removes it from the call stack. It also opens a new window with one of the standard Python or Blender modules. In such a case *Resume* (F8) this execution, then check the console for the error message (page 59);

[1] You can enhance the *pydev_debug.py* code in two ways: 1. Prepend (instead of appending) the script directory to ***sys.path***; 2. Use the new ***exec_module()*** method from the ***importlib*** package, available since Python 3.4;

3.5 Improving our script

Our Python script originated from the Blender operations log. As I shown in the previous section (see page 59), it works properly only for the test scene configuration, where I have "recorded" these API statements. In this section we will enhance this code so it will work for the user-selected objects. It will also become a more general, so you will be able to specify the type of the Boolean operation for the ***boolean_operation()*** procedure.

Let's start by setting the operation mode of the *Boolean* modifier. When you change this option, you will see corresponding API statement in the *Operations Log* window (Figure 3.5.1):

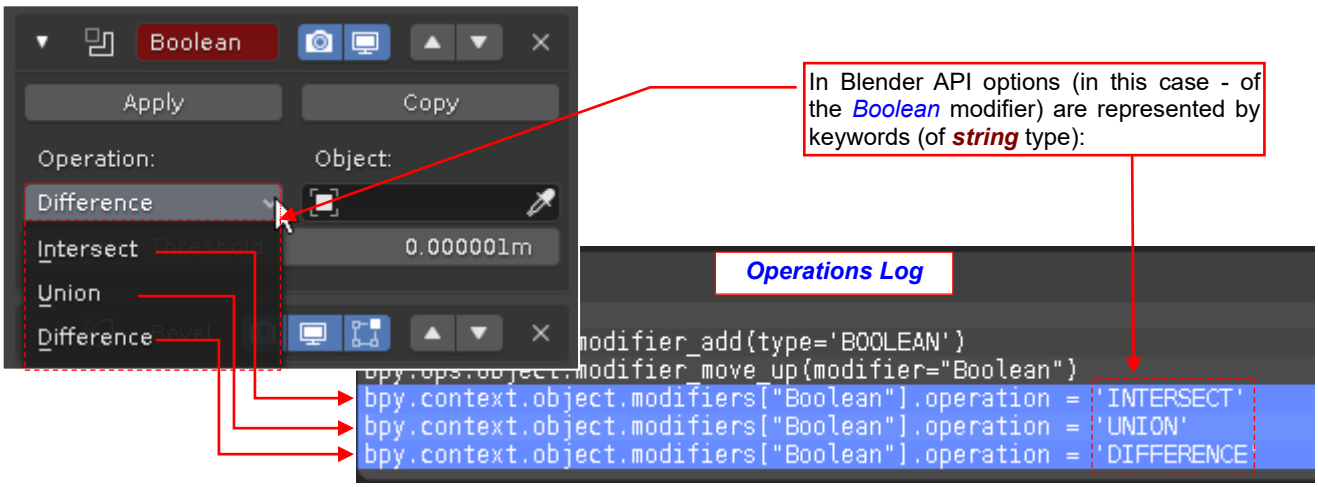


Figure 3.5.1 API statements for selecting one of Boolean operations

It occurs that Blender API represents such enumerations as text keywords. You can alter the mode of a *Boolean* modifier by setting its ***operation*** field to one of the three values: **'INTERSECT'**, **'UNION'**, or **'DIFFERENCE'**. Of course, if you need , you can also get the current value of this modifier field – for example, in the *Python Console* (Figure 3.5.2):

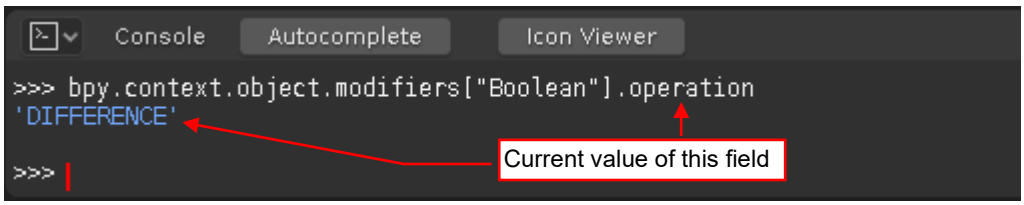


Figure 3.5.2 Getting the current setting of a *Boolean* modifier

For quick "discovering" the API classes and fields names corresponding to user interface elements, you can enable in the Blender preferences window (*Edit→Preferences*) the *Python Tooltips* option (Figure 3.5.3):

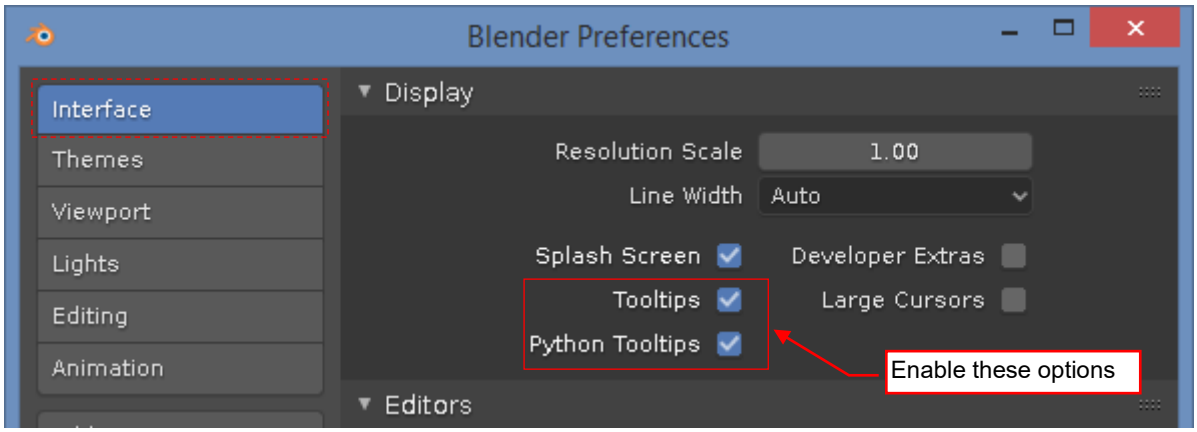


Figure 3.5.3 Enabling Blender API tooltips

From this moment, when you hover mouse cursor over any field on the screen, Blender will display information about its class and Blender API "path" (expression) that references this item (Figure 3.5.4):

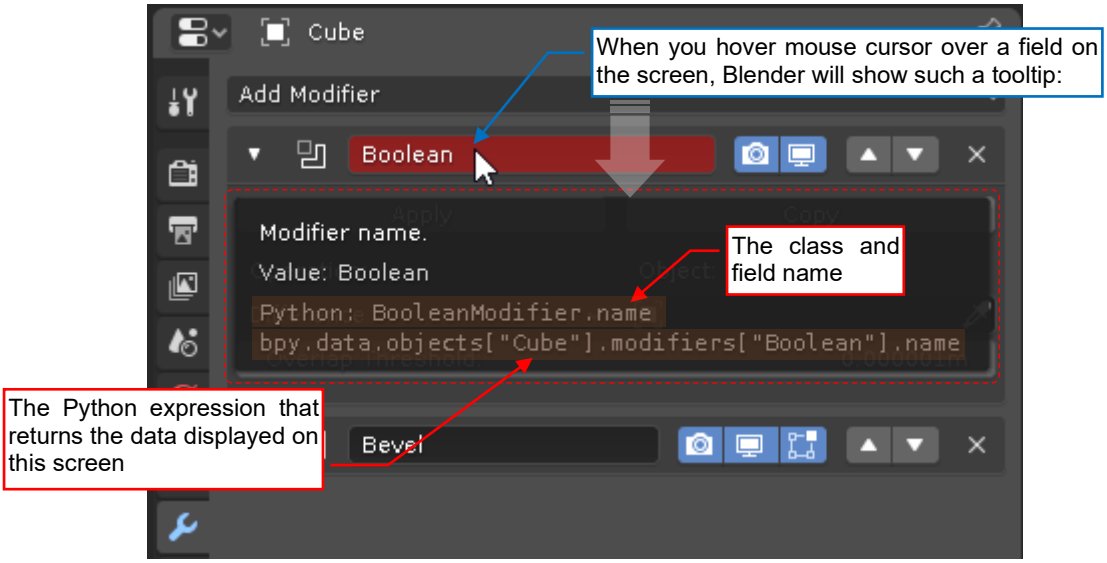


Figure 3.5.4 Python API tooltips

The modifier name (**"Boolean"**, in this case) is unique in the modifiers stack of a single object[1]. Note that Blender API uses it as the index (keyword) for referencing elements of this stack (for example in the *Apply* statement in Figure 3.3.7, on page 49). I used this observation in the new version of the ***boolean_operation()*** procedure. I introduced here many enhancements (Figure 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)
```

Additional arguments: ***op***, ***apply***

obj is an auxiliary shortcut for the active object

mod represents our modifier (it is always added at the end of the list)

This loop moves the newly added modifier to the top of the list

Instead of the fixed text, I use the name of modifier ***mod***

Figure 3.5.5 Improved modifier handling in the *boolean_operation()* procedure

I extended the argument list of this method with two new elements: ***op*** determines the kind of performed Boolean operation, while the optional ***apply*** flag allows for leaving the result as the dynamic modifier. To make this code more readable, I assigned the active object reference to an auxiliary variable named ***obj***.

[1] When you add a new modifier, Blender gives it a unique name. For example, if I added another *Boolean* modifier to the modifier stack shown in Figure 3.5.4, it would receive name **Boolean.001**. When the user changes the modifier name – Blender does not allow to type an existing one.

In the second local variable: ***mod*** I store the reference to the newly added modifier. Note, that initially I assign it to the last element of the modifiers list. I do it, because I know that the ***object.modifier_add()*** operator always places the newly added element in this position[1]. After this operation the modifiers list can contain one or more elements. That's why I replaced the single call to the ***object.modifier_move_up()*** operator with a loop. It moves modifier ***mod*** toward the beginning of the list, until it reaches the first position. Because the modifier names are created automatically, I cannot assume that the newly added *Boolean* modifier will receive name **"Boolean"**. If there was another *Boolean* modifier in this list, the new one can be named **"Boolean.001"**, or similar. That's why instead of the fixed **"Boolean"** text, copied from the *Operations Log* line, I pass the value of ***mod.name*** to the ***modifier_move_up()*** and ***modifier_apply()*** operators.
I also added optional argument ***apply*** to the ***boolean_operations()*** procedure. By default, the value of this flag is *True*, which means that the result of modifier ***mod*** is applied to the object mesh. (As in the previous sections). However, if you set it to *False*, the newly added modifier will remain as the first element of the modifiers list, and the result of this procedure will remain "dynamic". (It will be a quick method for adding *Boolean* modifiers).

Let's try to modify the main code of our script so that it will use eventual other objects selected by the user as the subsequent "tools", applied to the active object. Thus, there is an elementary question: how to get the list of currently selected objects? You cannot get a hint about it from the log window, because operators read this information internally from the environment data objects (unknown for us). Reasoning that such an information should be a part of the "execution context" of this script, I assumed that it is in one of the ***bpy.context*** object fields. Let's look at the contents of this object, using the autocompletion window (Figure 3.5.6):

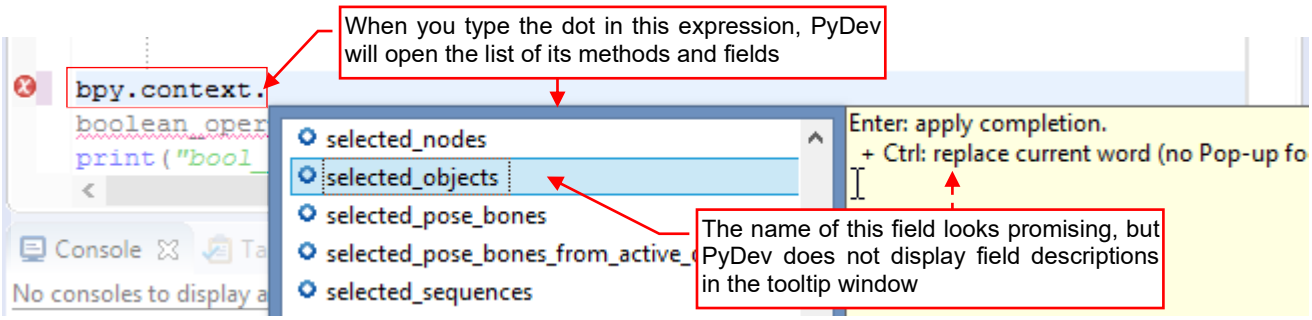


Figure 3.5.6 Browsing members of the *bpy.context* object

Unfortunately, PyDev does not displays tooltips for the fields, so let's use ***.selected_object*** in our code and try to find its description in *bpy.pypredef* (As it is shown in Figure 3.2.7 and Figure 3.2.8, on pages 41 and 42):

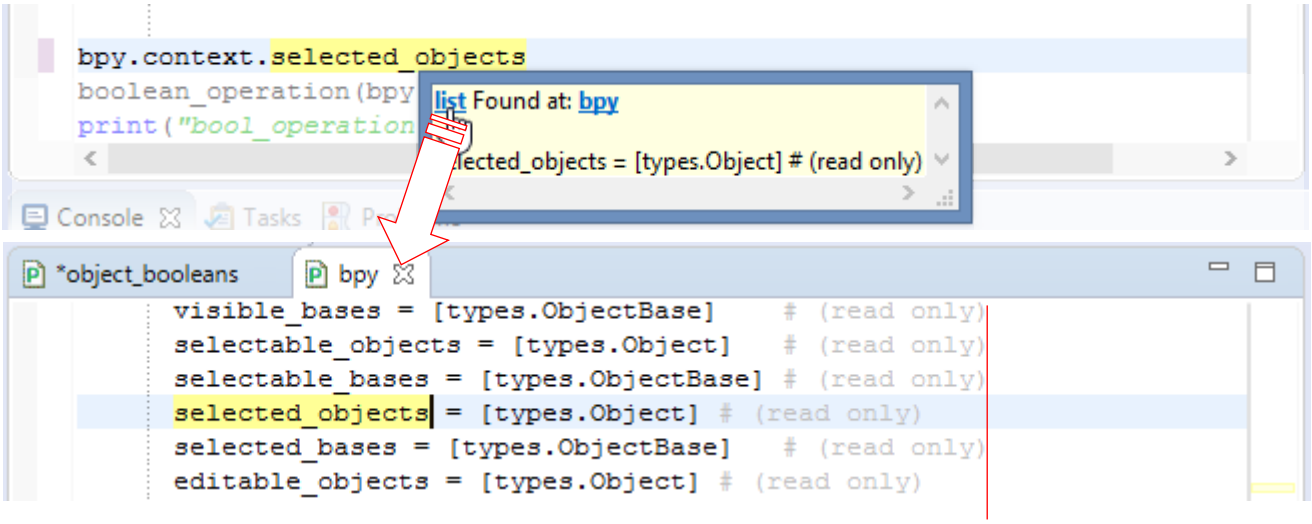


Figure 3.5.7 Searching for the description of the *bpy.context.selected_objects* field

[1] This is my assumption, based on long experience: from programmer's point of view, modifier "stack" is a list. (I have been using Blender for many years). The operation of adding a new modifier is not documented. (The official manual often skips such trivial steps). If you want to be more cautious than me – copy the ***obj.modifiers*** list before invoking ***modifier_add()*** operator, then compare their elements.

The ***bpy.context*** is a specific object, because its contents depend on the type of the Blender window that has invoked your script (see page 141). What's more, there are no descriptions for its fields – also in the official documentation (!). I had to rely on the comments from other users, which I found using the Internet search tool. They said that the ***selected _objects*** represents the current selection set. However, in the header file of the ***bpy*** module I also found another field with intriguing name: ***selected_editable_objects*** (Figure 3.5.8):

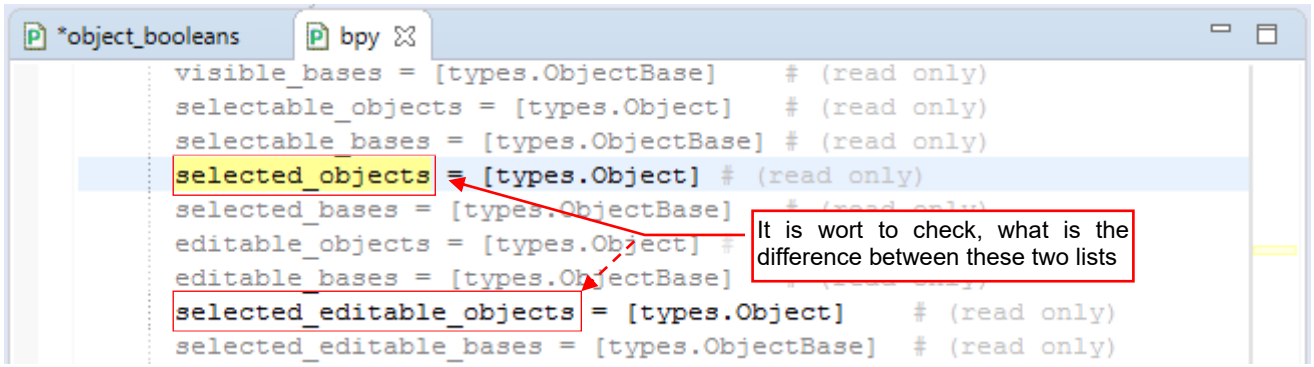


Figure 3.5.8 The *bpy.context* fields that I want to check

I would like to know the difference between these fields[1]. In the first trials that I conducted using the test **.blend* file, both fields returned identical results. Ultimately, I found the difference when I linked (*File→Link*) to the test scene an object named **Cone**, from another Blender file[2] (Figure 3.5.9):

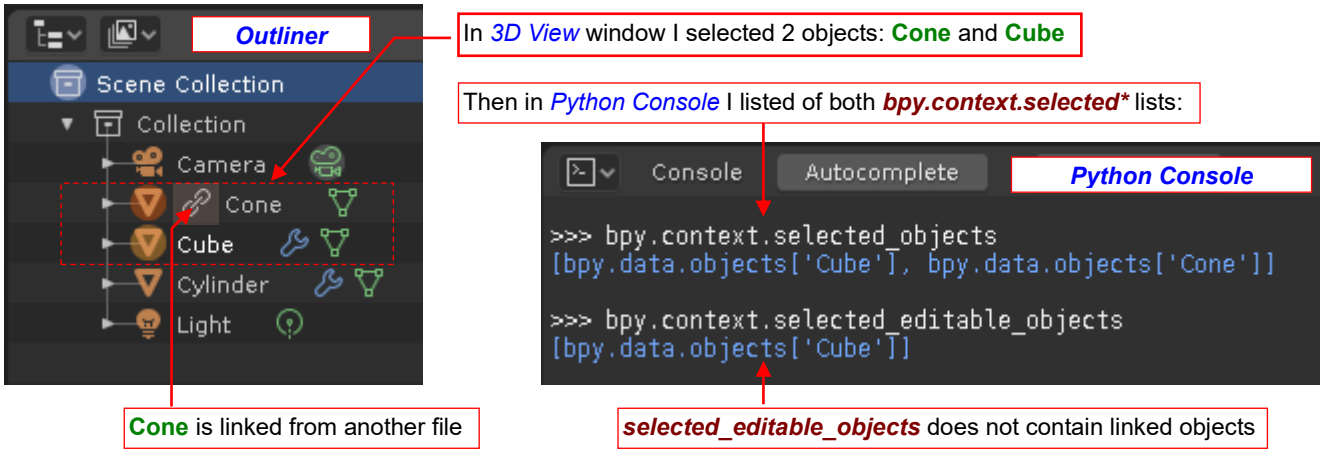


Figure 3.5.9 The difference between the *selected_objects* and *selected_editable_objects*

It occurs that the ***selected_editable_objects*** list does not contain the linked objects – **Cone**, in this case. (Indeed, you cannot edit properties of such an object – thus the name of this ***bpy.context*** field). However, I checked that I still can use this linked object as the "tool" in the *Boolean* modifier.

- In this script I will use the ***bpy.context.selected_objects*** field for getting the list of the selected objects.

Fortunately, Blender API provides the ***selected_objects*** field practically in all contexts, as it does for the ***bpy.context.object*** field. (The API documentation describes that both fields belong to the *Screen context*).

[1] This is not a pure curiosity. In Blender 2.5 the operations log window showed only the operator statements, so you could not find out there how to refer the active object in Blender API. The ***bpy.context*** was also undocumented (as it is now). That's why I decided that for the active object reference I will use the ***bpy.context.active_object*** field. A few months later I tried to append my script to one of the Blender menus. I discovered then that in this case my code is invoked in a different context, in which ***bpy.context*** object has no ***active_object*** field! That's why in Blender 2.8 I am carefully checking the context fields that I am going to use in my script. On the other hand – it is a shame that Blender Foundation has not documented this important part of the Blender API for nine years!

[2] Blender can use every **.blend* file as an external "datablock library" (a container for objects, meshes, materials, textures, nodes, …). You can dynamically link any of these items to a scene in another Blender file.

After this lengthy discussion about the potential information sources, I changed the main code (Figure 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: static "working copy" of this list

Remove the active object from this working copy

Use this "shortened" ***selection*** list as the source of the "tool" objects

Figure 3.5.10 Improved main code of the script

To perform given Boolean operation for each of the user-selected objects, I exclude the active ("target") object from this list. (Otherwise it would cause an error – as described on page 59). For this purpose, I copy the contents of the ***bpy.context.selected_objects*** iterator into a static list, named ***selected***. Then I removed from ***selected*** the active object. Finally, I invoked the ***boolean_opeartion()*** procedure for every element of this "shortened" list.

Let's check now, if such a modified script works properly. I added to the test scene another object: **Sphere**. Then I selected all these three objects in following order: **Cylinder**, **Sphere**, and **Cube**, and run the script. (I made sure that the debug server is running, and clicked the *Run Script* button in Blender – as on pages 54, 55). Figure 3.5.11 shows the initial state and the final result:

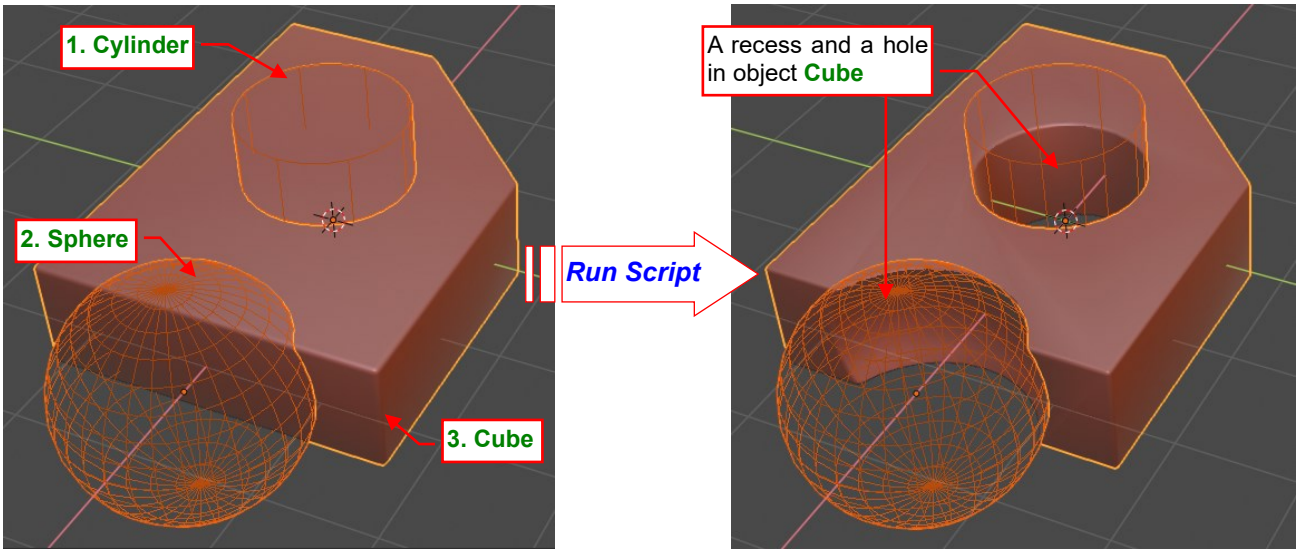


Figure 3.5.11 Final result of the modified script

There were no errors, and the script result looks properly.

Now undo these script results (Ctrl-Z) and select again **Cylinder** and **Sphere**. Then select also another object: **Lamp** (see the test scene outline in Figure 3.5.9, page 64). Finally select object **Cube** (so it will be the active object again). Our script will fail for such input data:

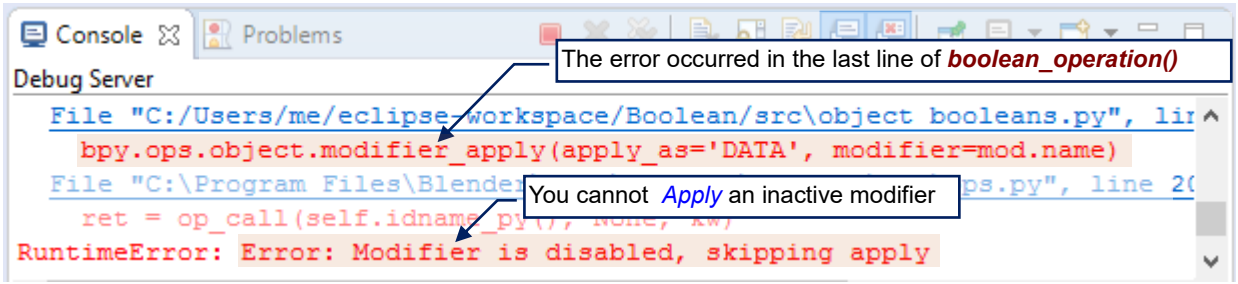


Figure 3.5.12 An error, caused by an attempt to use Lamp as the "tool" object

What has happened? *Boolean* modifier ignores objects that does not contain mesh data, like **Camera** or **Lamp**. In the result, the new modifier, added to the active object (**Cube**) with **Lamp** as the "tool", will never become active. That's why Blender raises a runtime exception when the script attempts to call ***modifier_apply()*** procedure for this (still disabled) modifier.

How to avoid such situations? You can check if the modifier is enabled just before calling the *Apply* operator. However, I do not especially like this idea. In the future it may happen that Blender will raise a runtime exception in the previous step, on the attempt to assign a *Lamp* object to the *Boolean* modifier. (In principle, this is an invalid operation). That's why it is better to not invoke the ***boolean_operation()*** procedure for the objects of wrong type. This applies not only to the improper type of the ***tool*** argument, but also to the other cases. For example - I can easily imagine a situation in which a user selects by mistake **Lamp** as the active (target) object!

To precisely determine what object types accepts *Boolean* modifier, I added to the test scene two additional objects: **Cone**, which is linked from another file, and **Torus**, which is a collection instance. (This collection is hidden and contains just the source object). Figure 3.5.13 shows the current state of the test scene and its structure in the ***Outliner*** window (set to *View Layers* view):

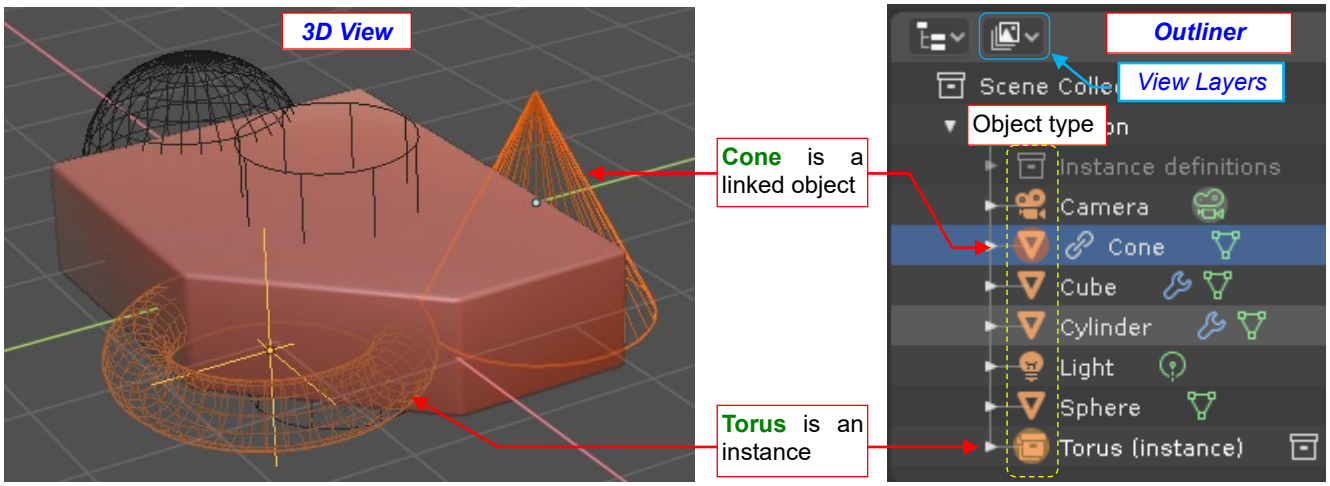


Figure 3.5.13 Additional objects in the test environment

Outliner shows the object types as icons. Our script needs this information for checking if the selected objects contain meshes. But which of the Blender API fields returns the Blender object type?

In searching for this field, I used the *Outliner* window in the *Data API* view (Figure 3.5.14):

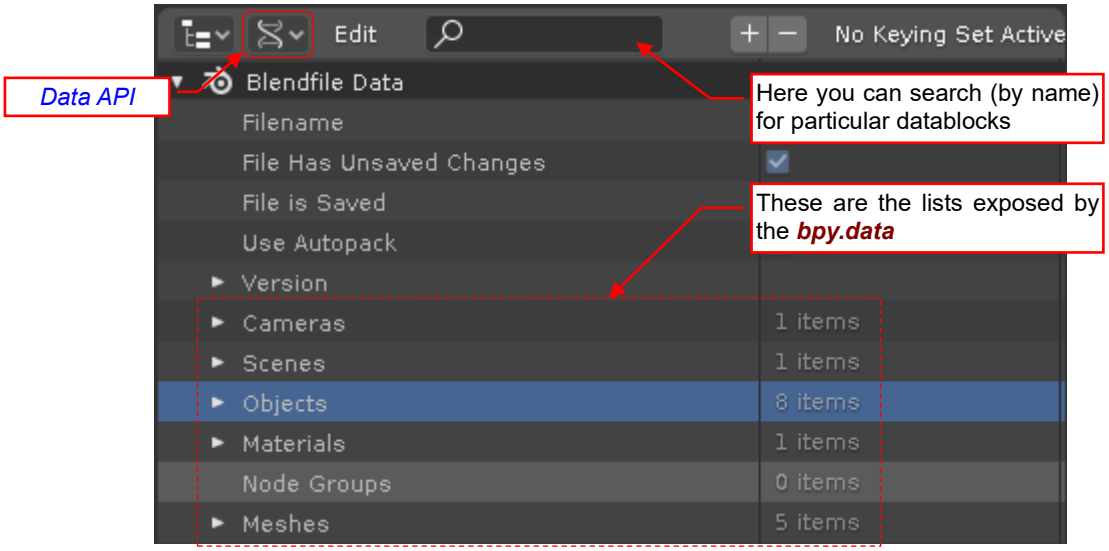


Figure 3.5.14 *Outliner* window (in *Data API* view)

The *Outliner* window in the *Data API* view shows the entire content of the current **.blend* file. In principle this is just a user interface for displaying contents of the ***bpy.data*** lists. When you expand the *Objects* collection, you will find there all the scene objects. Now you can examine their fields (Figure 3.5.15):

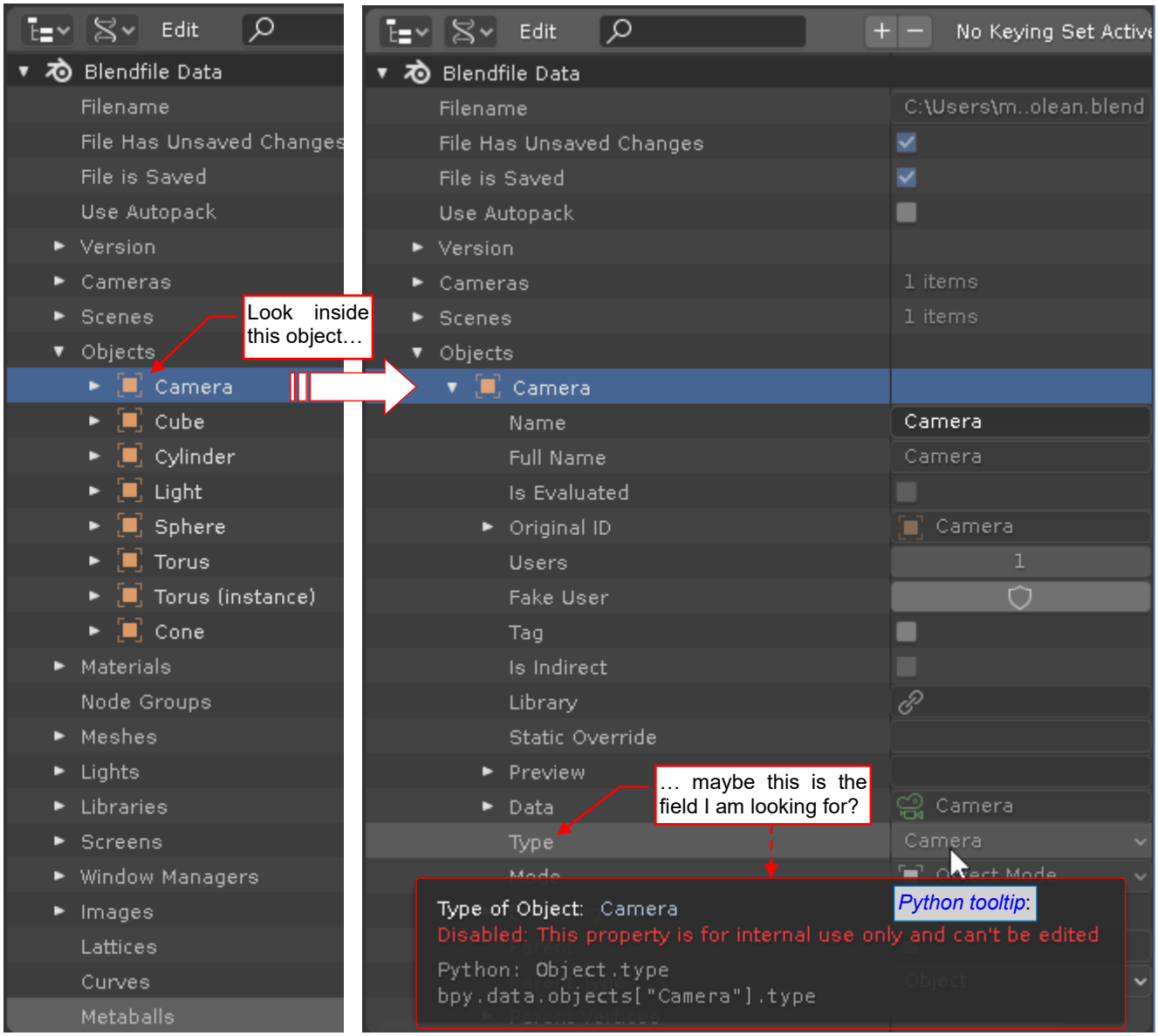


Figure 3.5.15 Browsing the object data

I started browsing fields of the first object from the *Objects* list, searching for an item that displays object type. (In this case it was a *Camera*). I quickly found a field that bears promising name *Type* (and the value: *Camera*). I stopped my mouse over this item for a few seconds, so Blender displayed its description and API details. They confirmed me that this is the field I need. (It seemed to return the correct information and was read-only). Using the reference expression from the tooltip as the example, I quickly checked the ***type*** values of other objects:

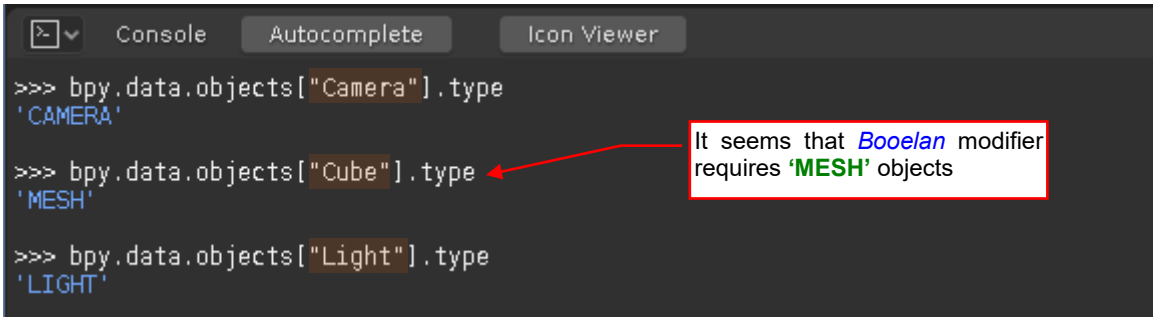


Figure 3.5.16 Quick check of the *type* values in other objects

It occurred that **Camera** is an object of **'CAMERA'** type, **Lamp** – **'LIGHT'**, and the **Torus** instance – **'EMPTY'** (I did not show this last case in Figure 3.5.16). All the other objects in the test scene are of **'MESH'** type. It precisely matches the icons shown in the *View Layer* mode of the *Outliner* (see Figure 3.5.13). Thus, it seems that the ***type*** field indeed contains the information we need. Let's check it also in the API description (Figure 3.5.17):

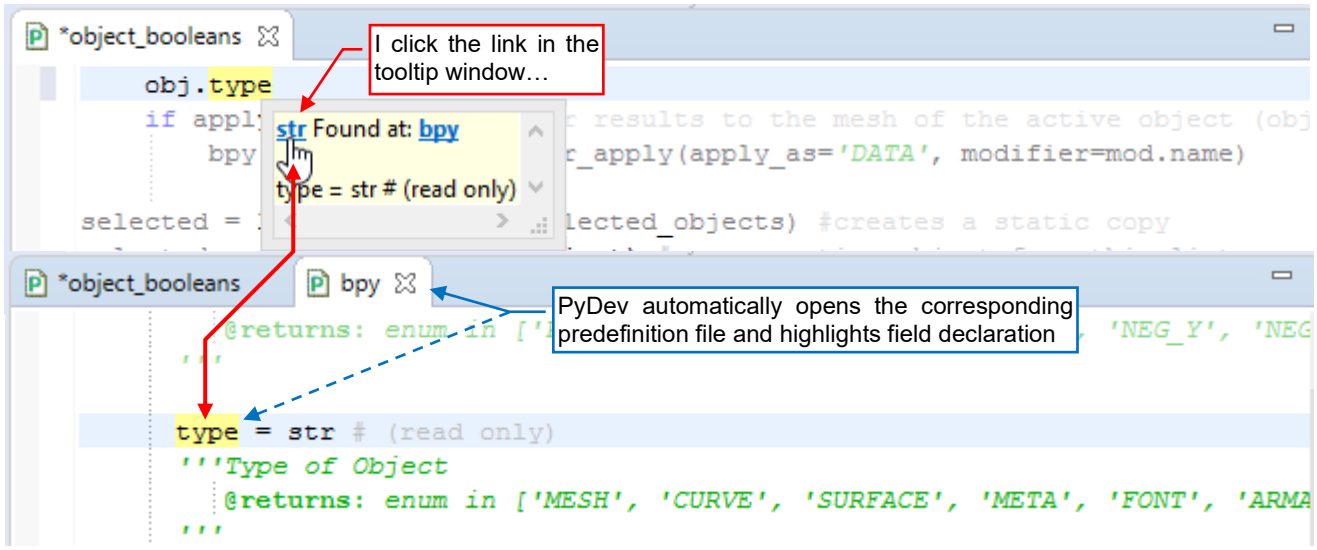


Figure 3.5.17 Checking details of the *Object.type* field

Using PyDev tooltip link, I quickly found the ***Object.type*** declaration, as in figure above. Its description confirms my observations. Conclusion: every object passed to ***boolean_operation()*** as the ***tool*** argument must be of the **'MESH'** type.

I will not show this in another illustration, but I also checked that Blender does not allow adding modifiers to the objects linked from another file (as the object **Cone** in our scene). This means that we cannot point such a linked object as the active (target) object for our script, because it will cause an error. I also checked in Python console that the ***type*** field returns **'MESH'** value for the local objects (as **Cube**) as well as the linked objects (as **Cone**). So - how to recognize a linked object in the script?

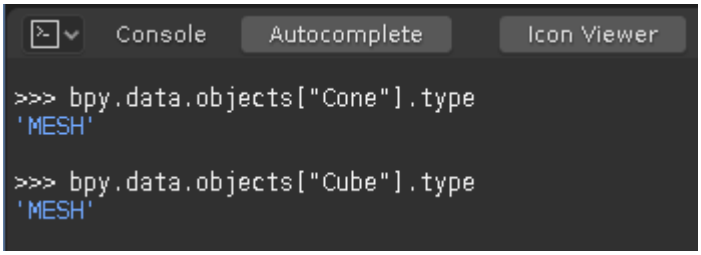


Figure 3.5.18 Types of the linked (Cone) and local (Cube) objects

To find such a "distinguisher", I examined the **Cone** object fields (in the *Data API* view of the *Outliner* window). I noticed there a field named ***library*** (Figure 3.5.19):

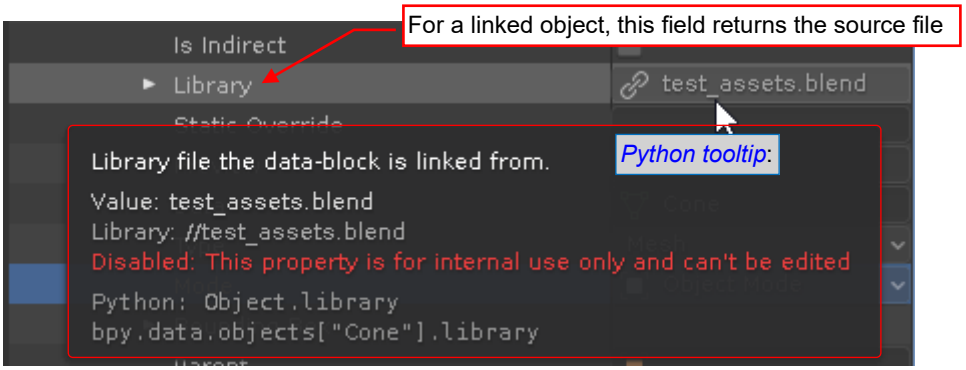


Figure 3.5.19 Reference to the source (library) file

Looking at the ***bpy*** API header declarations I can see that this ***library*** field returns a reference to an object of ***Library*** class. It contains the full path to the source file and other details. However, the most important thing for me is that for all local objects ***library*** returns *None* (I checked this in the console). So - this is the flag I need!

Using this information, I can update the main code of the script (Figure 3.5.20):

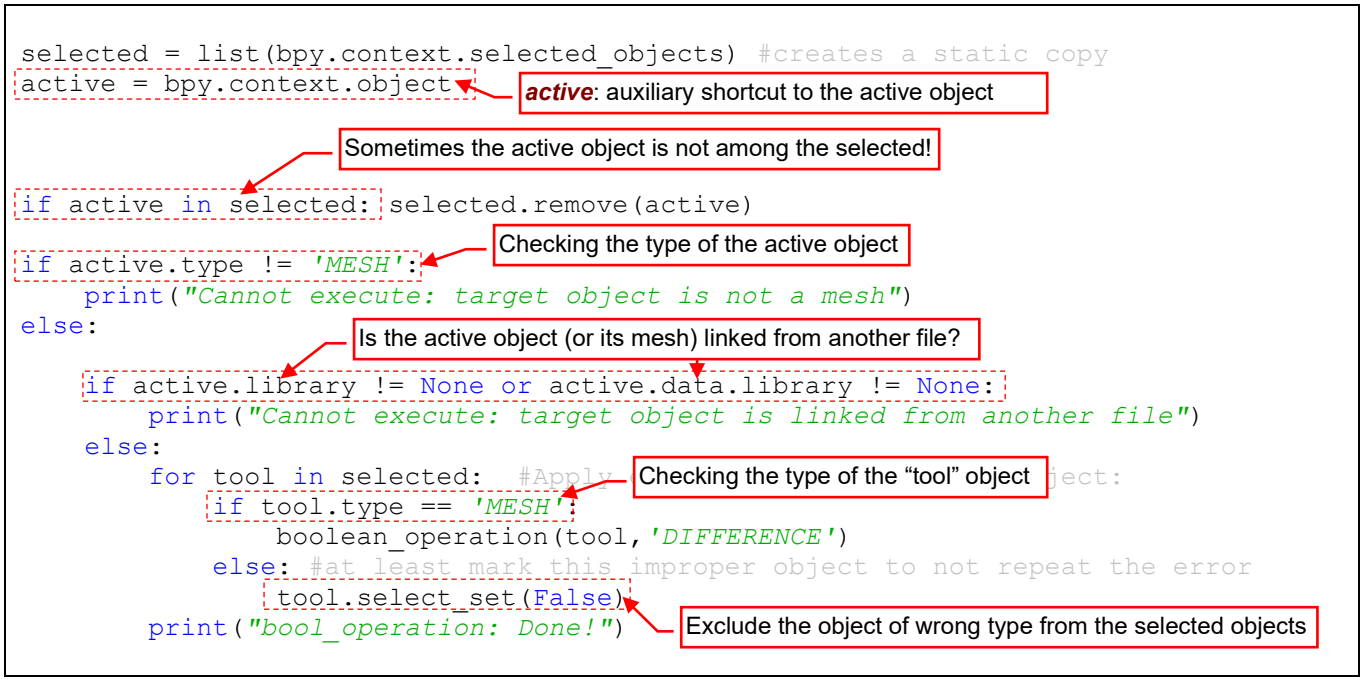


Figure 3.5.20 Validation of the input data (in the main code of the script)

For the greater code readability, I created auxiliary variable ***active*** and assigned it the active object reference. After some trials I also discovered that in certain scenarios the active object is not among the selected objects. That's why I added a condition for this case in the next line. (Otherwise an attempt to remove object ***active*** from list ***selected*** would raise a runtime exception).

In the further lines I am checking if the active object is of the **'MESH'** type. If so – I am also checking if it is not a linked object. Note that I am also testing if its mesh is not a linked datablock[1]. (I have found in the *Outliner* window that ***Object.data*** field returns the reference to its ***Mesh*** object).

Finally, when the active object seems to be OK, I am starting the loop for all the ***tool*** objects from list ***selected***. However, before invoking the ***boolean_operation()*** procedure I am checking if object ***tool*** is a mesh. If not – I am excluding it from the actual (scene) selection, so the user will not use it again by a mistake. (The commands that manipulate current selection set are not displayed in the *Operations Log* window. I have found in the API documentation[2] that I can use the ***select_set()*** method for selecting/deselecting scene object).

In the next section I will modify this code so it will "catch" all unexpected runtime errors. I will also improve readability of the messages displayed by this script.

[1] Sometimes it may happen that a local object uses the mesh data linked from another file. You can, for example, "paint" this mesh using a local material assigned to the object, not the object data.

[2] In the case of other "selectable" datablocks – for example, mesh vertices or edges – you can find in *Outliner* a field named ***select***, which you can set to *True*/*False*. The ***bpy.types.Object*** class, which represents a scene object, also had such a field in the previous Blender versions. Thus I opened Blender 2.8 Release Notes, which describe all the changes introduced in this new version. On this page I found section about the changes in the API. There I entered Scene and object API subsection, where I ultimately discovered a fragment about this issue. I think that this change is related to the **.blend* file architecture modifications. In Blender 2.8 each *View Layer* (aka *render layer* in the API) preserves its own selection set. Every *View Layer* also contains instances (references) to the scene objects. They are represented by the ***bpy.types.ObjectBase*** class. In this class you can find the "classic" ***select*** field, which controls the object selection state in the "host" view layer. Another field of the ***ObjectBase*** class is ***object***, which returns reference to the scene object (instance of the ***bpy.types.Object*** class). You can find the list of the objects ("object bases") from the current view layer in the following ***bpy.context*** iterators: ***selectable_bases***, ***editable_bases***, ***visible_bases***, ***selected bases***, ***active base***.

Summary

- The *Python Tooltips* option (page 62) is a great tool for learning Blender API fields directly from the screen. You can enable it in *Blender Preferences* (page 61);
- Blender always appends a new datablock to the corresponding datablock list. This rule also applies to the object ***modifiers*** list. That's why you can assume that the newly added modifier is always in the last element of this list (as I did on page 62);
- Blender also enforces the uniqueness of datablock names. That's why I pass the modifier datablock ***name*** to the operators (page 62); I do not need to know this name – it is just an id, created and managed by Blender;
- There are several fields (iterators) in the ***bpy.context*** class that provide information about currently selected objects. You can find them in the PyDev autocompletion window (page 63). Unfortunately, they are not documented. Usually you will use the ***bpy.context.selected objects*** iterator. However, if you need a list without eventual linked objects or object instances – use ***selected_editable_objects*** (page 64). In other cases, a more useful can be ***selected_bases***: it returns the references (instances of the ***ObjectBase*** class) to the objects used int the current view layer. Anyway, it is a good practice to check the contents of such an iterator in a test scene, before using it in the code;
- Some information about the API objects is not displayed in in the operations log window or in the screen tooltips (for example: scene object type). To read it, you can use the *Outliner* window in the *Data API* view (page 66). In this window you can browse the entire contents of the current ****.blend*** file. You can get the Python expression for any field in this structure using the tooltip window. (Python tooltips are also available in the *Outliner* – see page 67);
- Use the ***bpy.types.Object.select_set()*** method to switch the selection state of a scene object (to selected/not selected – see page 69). To get the current selection state, use another function (method): ***select_get()***.

3.6 Handling the runtime errors and user messages

Looking at the result of the previous section (Figure 3.5.20, page 69) you can notice that the input validation occupies most of the script. (This proportion is especially striking when you compare this code with its earlier version, shown in Figure 3.5.10 on page 65). Finding the invalid data is important, but equally important is proper user notification about the reasons of aborting the requested action. ("Proper" means that the user should understand the reason and know how to avoid it in the future). In this section I will try to make this code as "error-proof" as possible[1] and improve the error messages. These changes will be also useful in the next chapter, when I will convert this script into a Blender add-on.

Let's start with handling the error messages. I think that it would be helpful to add an additional hint: the object name, where it is applicable. In this way we increase the chance that the user will notice and understand the mistake she/he has made. Figure 3.6.1 shows a new version of the main code. (Its results are exactly the same as the results of the code from page 69: I just changed the structure):

```
#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))
```

Constants, returned in the first element of the ***main()*** function result

further in this text I will use ***ERROR*** and ***WARNING*** constants

I added the object name in the message

A minimal main script code

Temporary "diagnostic" statement

Figure 3.6.1 Main script, modified

The main code of the script should be short and readable, so I grouped most of the lines written in previous section into new function named ***main()***. (It will be easier to make calls to this function in the add-on code). The ***main()*** function returns a list, which first element is a flag. It can be one of the four constants (string keywords) that I declared for this purpose. When there are no issues – ***main()*** returns a single-element list that contains the ***SUCCESS*** constant. Otherwise, in the second element of the result list you will find an error message.

[1] Remember, that "the user" can also mean just you! It is enough that you will try to run this utility after a long break (several months or longer). I think that you will not remember anything of the specifics of this script.

I reduced the script main code to two lines: the first calls the ***main()*** function, the second displays the result (actually it prints it in the console – see the last lines in Figure 3.6.1). In this way I separated the message formatting (in the ***main()*** function) and displaying (in the script main code). This is always a more flexible solution than placing the ***print ()*** statements everywhere. In this script the only ***print()*** statement is in the last, temporary line. In the future, when I use ***main()*** in the Blender plugin code, I will display eventual messages in completely different way.

I also composed the scene object names into the error messages. I think that this additional hint will help the user to find out what she/he did in a wrong way. Figure 3.6.2 shows the script results, obtained for various combinations of the selected objects. (I did these tests in the scene as in Figure 3.5.13 on page 66):

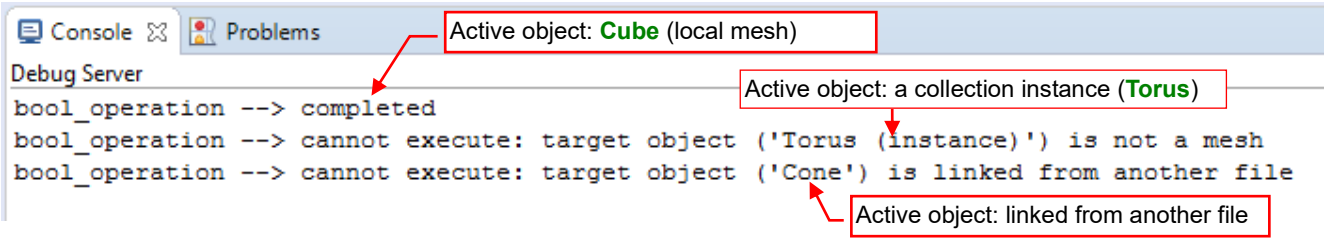


Figure 3.6.2 Results of the script, for various variants of the active object

Then I added further tests to the ***main()*** function: they validate the remaining ("tool") objects in the current selection set (Figure 3.6.3). Because each of them ends with a *return* expression, I could replace the nested *if: else:* statements with a more linear structure. (For a greater number of simple exclusions, such multiple-level nested conditions are less readable):

```
    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
    Each of these conditional statements quits this function, so I resigned from nesting them with the "else:" statements

    if active.library != None or active.data.library != None:
        return [INPUT_ERR, "target object ('%s') is linked from another file" %
                                                                    active.name]
    Checking if the selection set is empty
    if not selected:
        return [INPUT_ERR, "this operation requires at least two objects"]
    #main loop
    Auxiliary list for the not processed "tool" objects
    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)
    Add this object name to the skipped list
    #let's look at the results:
    If there is no skipped object: that's OK. Otherwise signalize a warning or an error
    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)]
```

Figure 3.6.3 Further modifications in the *main()* function

Before running the main loop, I am checking if the ***selected*** list is empty (it may happen, if the user selected just a single object). If so – I signalize an error.

Instead of excluding the non-mesh objects from the current selection set (as in Figure 3.5.20, page 69), I decided to enumerate their names in a warning message. For this purpose, I am collecting them in an auxiliary list named ***skipped***. If there has been at least one "tool" object processed in the loop – I signalize them in a warning. Otherwise I use the same text as an error message.

Figure 3.6.4 shows the results of the further tests, executed for various combinations of the selected objects. (I did these tests in the scene as in Figure 3.5.13 on page 66):

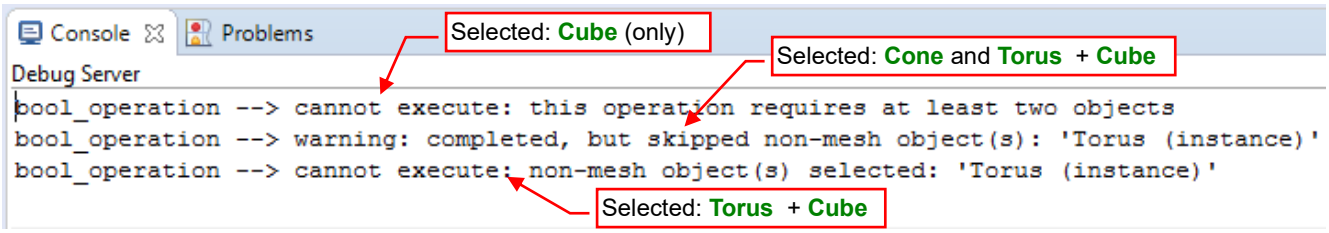


Figure 3.6.4 Further tests of the script

I do not have any illusions that these five validation tests that I have already implemented will allow me to avoid all possible runtime errors. To have at least marginal control over remaining runtime exceptions, I placed the whole code of the ***main()*** function into a *try:* … *except:* statement (Figure 3.6.5):

```
import traceback #for error handling
```
I am using the ***print_exc()*** function from this module

```
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:
        ...
        the code of this function - as in Figure 3.6.3 (page 72)
        ...
    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)]
```
"Catch" every kind of exception into variable ***err***

Print the "traceback" information in console (as in the Python standard)

Format the message fragment containing names of the selected objects

Check, if this local variable exists

Error message: the standard text + eventual information about the context

Figure 3.6.5 "Catching" eventual runtime exceptions in the *main()* function

I use an *except:* statement that will "catch" every type of Python exception and place it in the ***err*** variable. Then the ***traceback.print_exc()*** function will print in the console the standard, detailed traceback information about the Python stack in the moment of raising this exception. This is a diagnostic message intended for the programmer (i.e. for me), which will normally appear only in the Blender system console window.

For the user I format an additional text about the context of this error. I use for this purpose a helper variable named ***cntx_msg***. I suppose that most of the runtime errors will occur in the "core" code of the ***boolean_operation()*** procedure. That's why I am trying to place in ***cntx_msg*** the names of the active object and the current ***tool*** object. Of course, an error can also appear in the other parts of this code. That's why before placing the ***active*** and ***tool*** object names into the message I am checking, if they are defined at all. (If their names are in the ***locals()*** collection).

I did not have to wait a long time for an unexpected error, which would allow me to check this newly added *try:* … *except:* statement. It was enough to create clones of objects **Cube** and **Cylinder**. (Such objects, like **Cube** and **Cube (clone)**, share the same mesh named **Cube** – as it is shown in Figure 3.6.6):

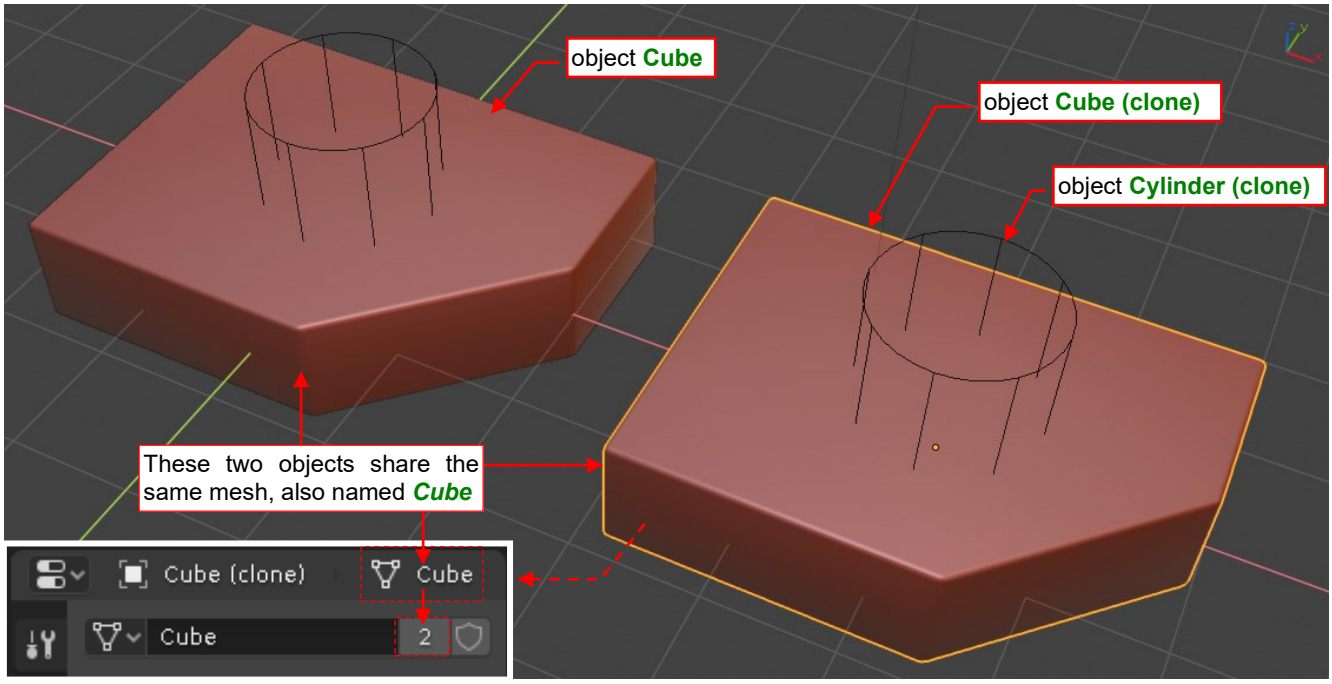


Figure 3.6.6 Clone of the Cube object (mesh data, shared between two objects)

Then I selected objects: **Cylinder (clone)** and **Cube (clone)** and run the script. Figure 3.6.7 shows the result:

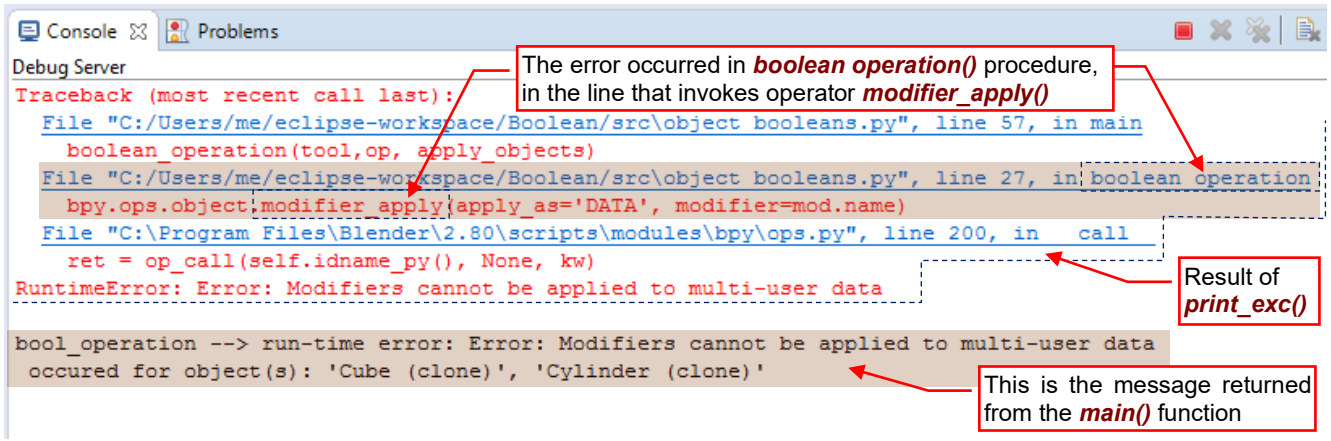
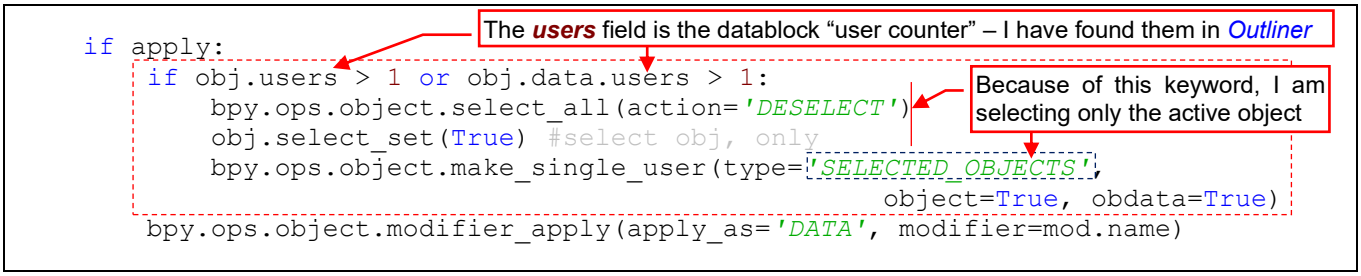


Figure 3.6.7 Script result for a clone object

Fine. I can see that "runtime error catching" works properly. As intended, the ***main()*** function has returned the error message that contains the object names. I have just completely forgotten that Blender does not allow to apply modifier results to a shared mesh. Thus, let's introduce a fix to the last lines of ***boolean_operation()*** (compare the code below with the code shown in Figure 3.5.5 on page 62):




```
    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)
```

Figure 3.6.8 Creating a local copy of a mesh before invoking the *modifier_apply()* operator (last lines of *boolean_operation()*)

Before invoking operator ***modifier_apply()***, I am checking if the mesh reference counter (***users*** field) of the active object and its mesh are greater than 1. If so, I am creating its copy (by invoking ***make_single_user()***).

You can get lost after all these changes. To stay “on the track”, see below the complete script code:

```
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) )
```

Figure 3.6.9 Complete code of the current script version

Note that I placed all the input data validation and eventual error message handling in the ***main()*** function. This auxiliary code occupies more lines than the core action, grouped in the ***boolean_operation()*** procedure. This is a typical proportion in the programs, which must interact with the most unpredictable element: the user 😊.

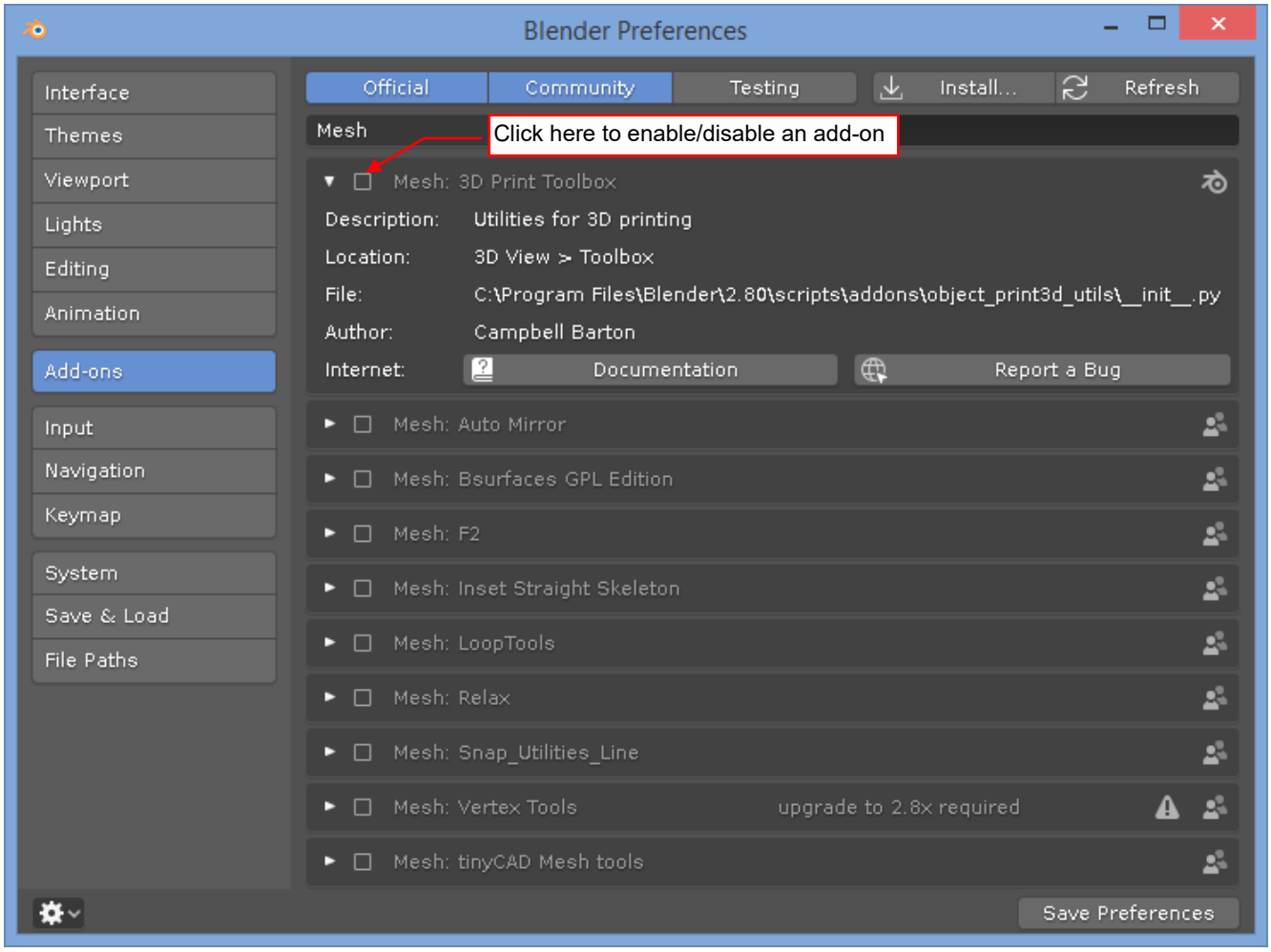
At this moment the only remaining explicit (i.e. entered "manually" in the code) parameter of this script is the Boolean operation type (**'DIFFERENCE'**). In the next chapter I will create a simple user interface which will allow the users to choose this value from a menu. It will be a part of the Blender add-on code.

Summary

- I placed into a function named ***main()*** the whole main script code, prepared in the previous section, This function returns a list that contains: the result keyword and eventual error message (page 71). Such a function can be easily integrated into the Blender add-on;
- It is a good idea to provide a hint about the operation context in the warning and error messages. In the case of this script these are the names of the active object and (usually) the "tool" object that caused the signalized problem (pages 72 and 73);
- To handle (in a minimal form) all the unexpected runtime errors, place in the ***main()*** function the ***try:*** ... ***except:*** statements (page 73);

Chapter 4. Converting API Script into Blender Add-On

Probably you know the Blender preferences window (*Edit→Preferences*). I suppose that you already looked at the *Add-ons* tab:



Every Blender add-on is a special Python script. This window allows you to compose the “working set” of plugins (add-ons) according to your current needs. During initialization, an add-on can add new elements to the user interface: buttons, menu commands, and panels. In fact, the whole Blender UI is written in Python, using the same API methods that are available for the plugins.

In this chapter, I will show you how to convert our Blender API script into a Blender plugin. This add-on will add to the *Object* menu the “destructive” *Boolean* commands (*Difference*, *Union*, and *Intersection*).

4.1 Adaptation of the script structure

So far, our script was "linear" - it executed what was written in the main code, from the beginning to the end. The Blender plugins work differently, as you will see it in this section. Therefore, their code must have a specific structure.

Let's begin with the plugin "nameplate". Each Blender add-on must contain a global variable ***bl_info***. It is a dictionary of strictly defined keys: **„name"**, **„autor"**, **„location"**, etc. Blender uses this structure to display the information in the *Add-Ons* tab (Figure 4.1.1):

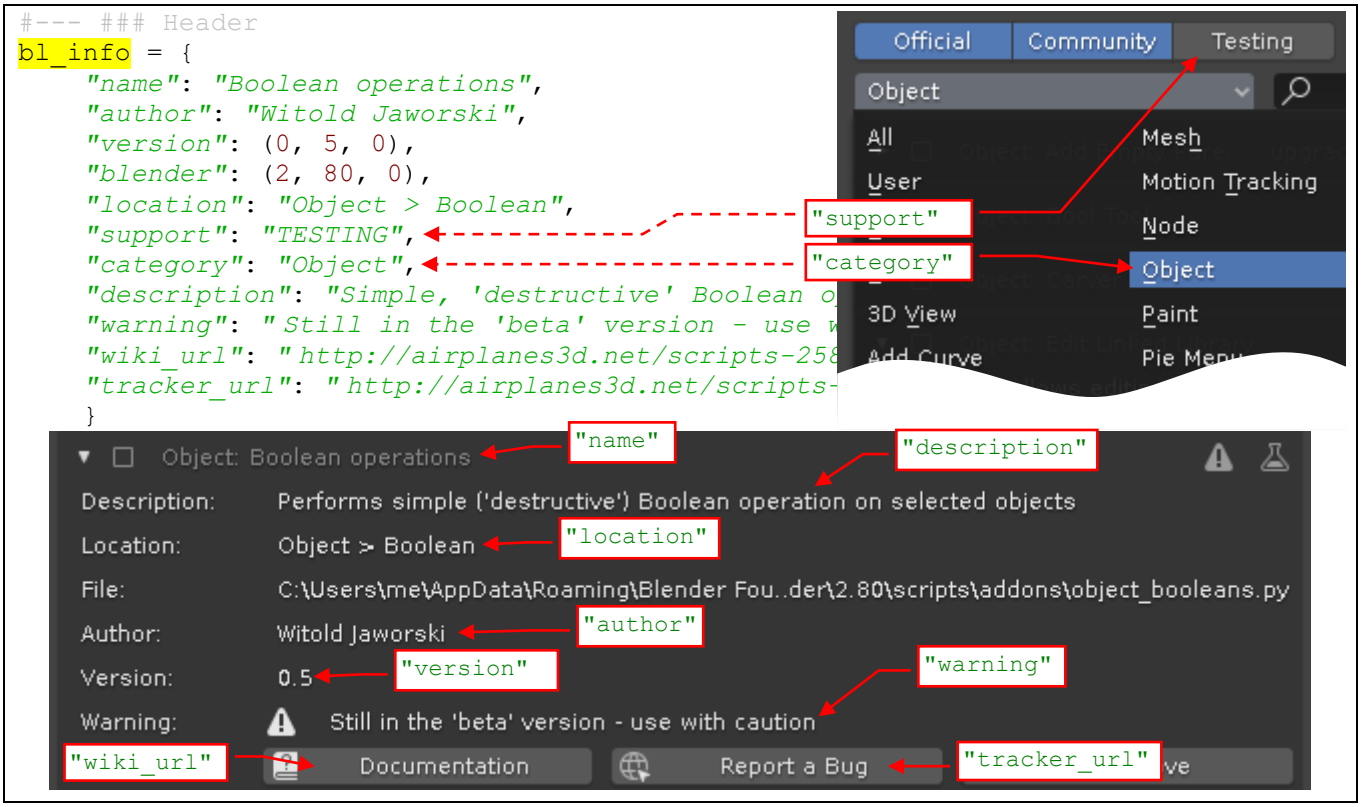


Figure 4.1.1 The *bl_info* structure and its pane in the *Preferences* window

You can leave some of these keys with empty strings — for example the documentation and bug tracker addresses (**„wiki_url"**, **„tracker_url"**). Be careful with the **„category"** value: use here only the names that are visible on the category list (in the *Add-ons* tab). If you use anything that is not there — your add-on will be only visible in the *All* category.

This plugin will expose our ***main()*** method as a new Blender command. To make it possible, we have to "embed" this procedure into a simple operator class (Figure 4.1.2):

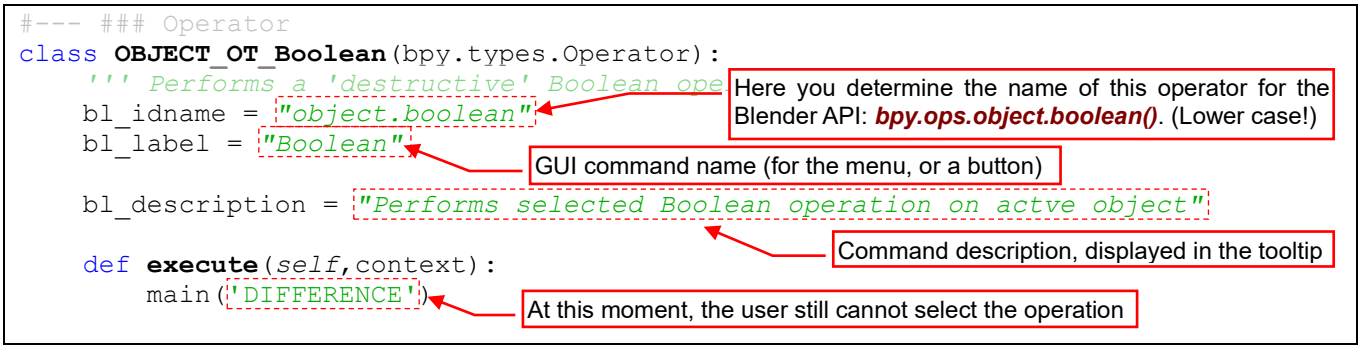


Figure 4.1.2 The operator class, "wrapped around" the *main()* procedure.

I named this class according the API guidelines: ***OBJECT_OT_Boolean***. Each new operator you define must inherit the abstract ***bpy.types.Operator*** class. Otherwise, it will not work properly.

The operator must have two class fields: ***bl_idname*** and ***bl_label*** (Figure 4.1.2). I also suggest setting another: ***bl_description***. (If it is missing, Blender displays in the command tooltip the *docstring* comment you have placed below the class header). At the beginning, our class contains a single method, with a strictly specified name and parameter list: ***execute(self, context)***. Inside it I placed the call to the ***main()*** function, still passing the fixed **'DIFFERENCE'** in the ***op*** argument (just for the tests). We will handle the result of this function later.

To register in Blender a class (or classes) from your module, you must add to the script two special functions, responsible for this operation. This code usually looks always the same: first, import from the ***bpy.utils*** module methods that register/unregister an API class. Then use them in your script, in the two methods named ***register()*** and ***unregister()*** (Figure 4.1.3):

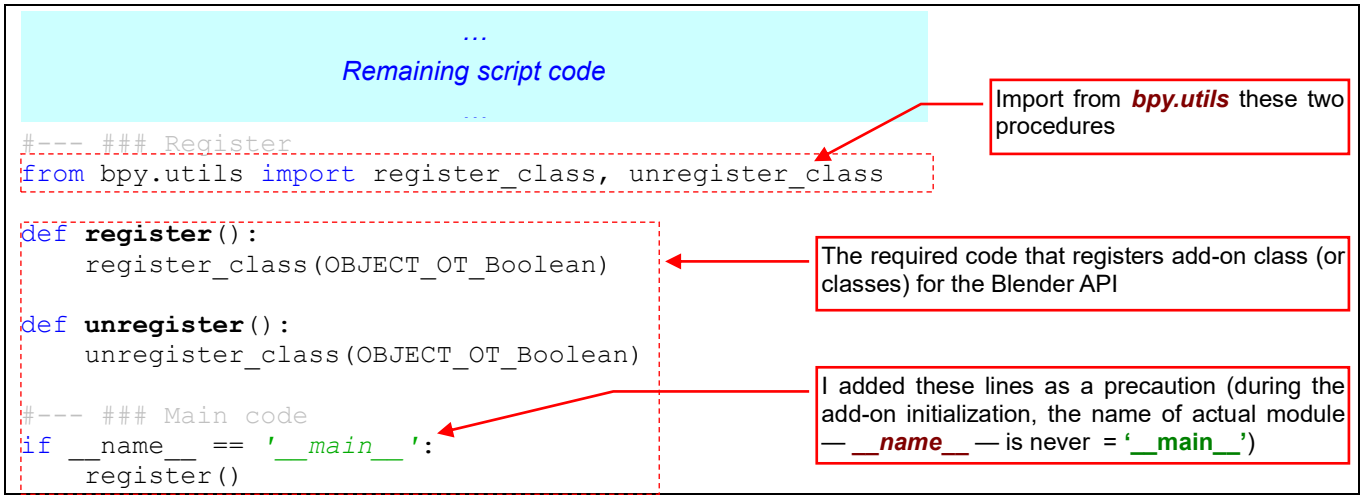


Figure 4.1.3 Registering the Blender API class from this script

- Every Blender add-on must implement two procedures named ***register()*** and ***unregister()***. They have no parameters and return nothing, as in Figure 4.1.3.

Let's check how does such modified script work. Make sure, that the PyDev debug server is active. Prepare a test environment in Blender, then press the ***Run Script*** button (Figure 4.1.4):

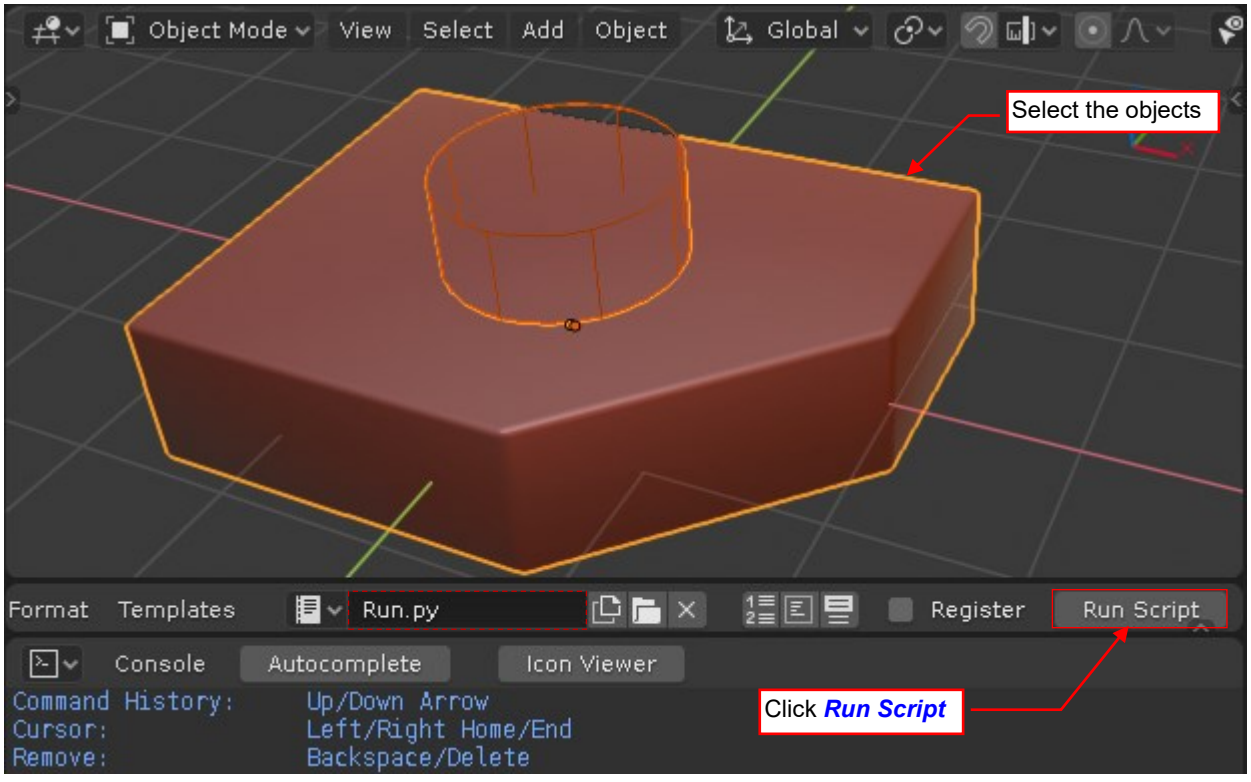


Figure 4.1.4 Launching our add-on in the debugger.

It seems that the execution of this script was completed without any error. However, there is no hole in object **Cube**! What is going on? Add a breakpoint to the ***execute()*** method, and run this script again. Nothing happens, and the code execution has not stopped at this breakpoint (Figure 4.1.5):

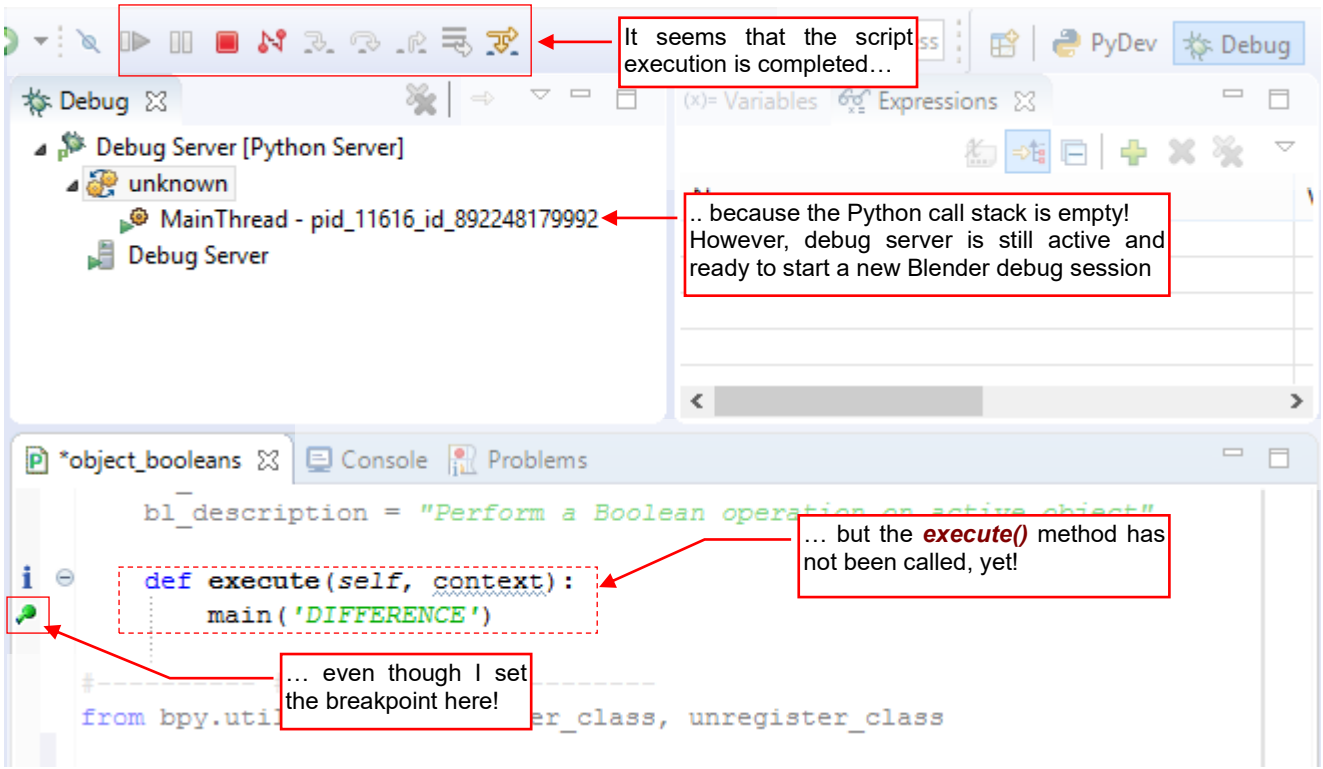


Figure 4.1.5 The state of the debugger after running the add-on script

Actually the main script code does not call the ***main()*** procedure. It just registers a new Blender command (operator), under the name that you have assigned to the ***bl_name*** field. In our case this is **„object.boolean"** (see page 78, Figure 4.1.2). Check in the Python console, whether the ***bpy.ops.object.boolean*** method exists (Figure 4.1.6):

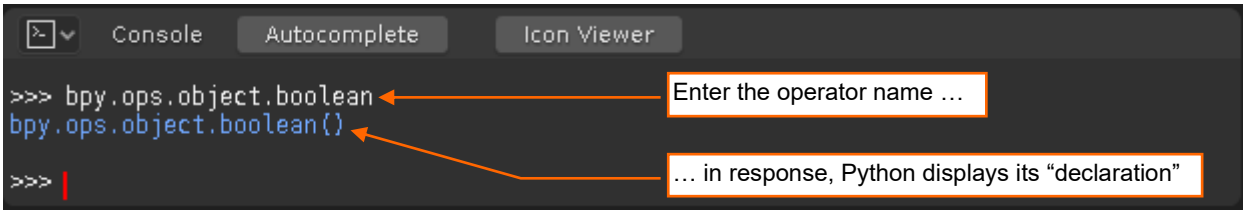


Figure 4.1.6 Checking results of the add-on registration

Now you can add this new operator to a Blender menu or a panel button. We will deal with the GUI integration subject in the next section of this chapter. For now, just call this command "manually" — from the *Python Console* (Figure 4.1.7):

Figure 4.1.7 Call the operator…

This time the Blender window has become locked, and the PyDev debugger is activated. It is waiting at the breakpoint we have placed in the ***execute()*** method (Figure 4.1.8):

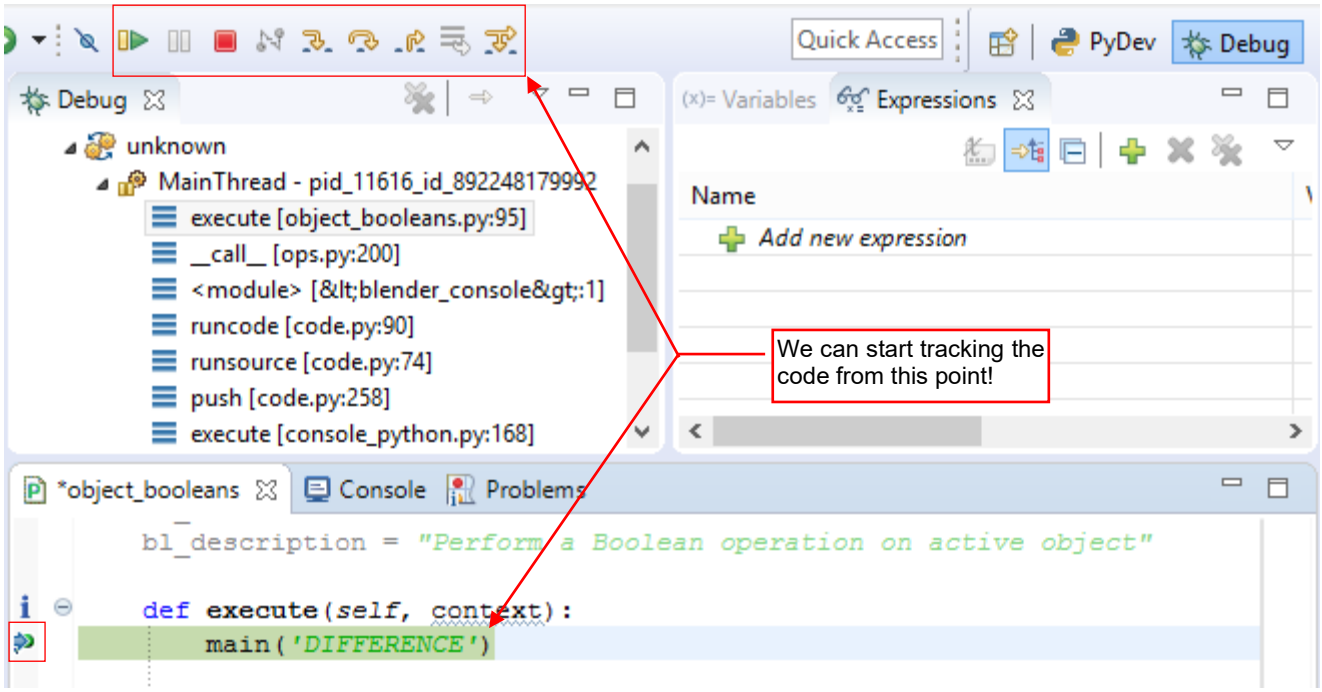


Figure 4.1.8 …and the debugger will stop its execution at your breakpoint

We have simulated here how Blender invoke our operator. When you call the ***bpy.ops.object.boolean()*** method (usually from a menu or a panel button), Blender will create a new instance of the ***OBJECT_OT_Boolean*** class. Blender uses this object just to invoke its ***execute()*** method. After this, the instance of this API class is immediately released (discarded). Such a "method of cooperation" („do not call us, we will call you") is typical for the all event-driven graphical environments.

By the way: note the arguments of this procedure, exposed in the *Variables* pane. Expand the ***context*** parameter to see what kind of information you can get from this object (Figure 4.1.9):

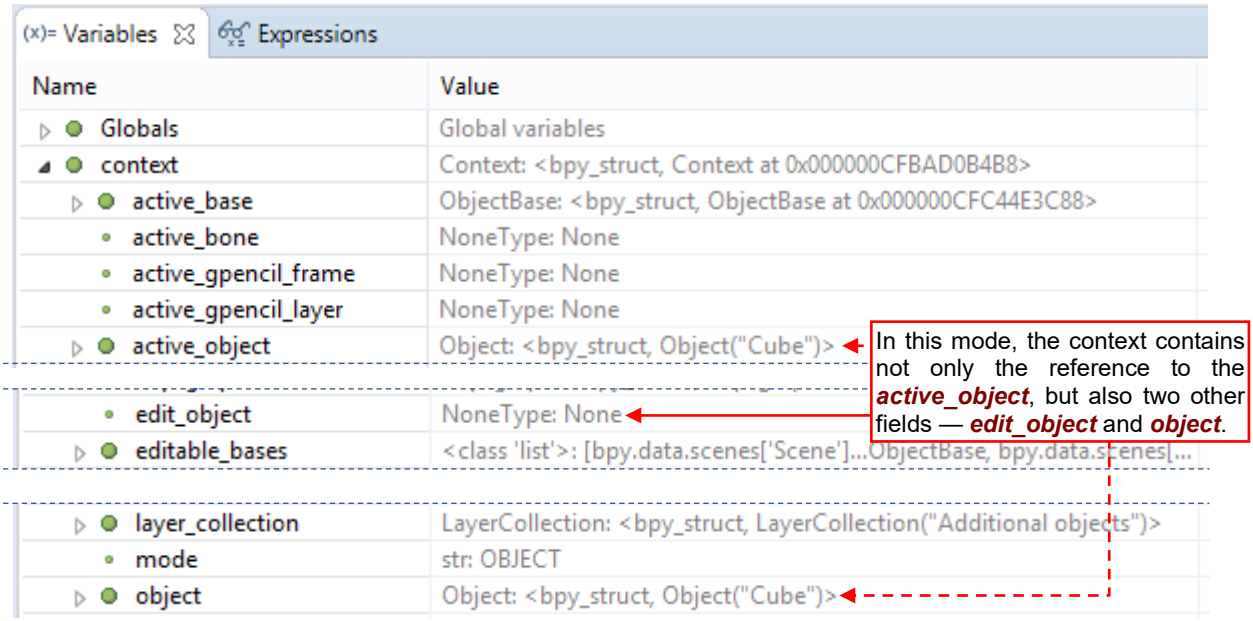


Figure 4.1.9 Previewing the context of this call

The ***context*** structure may contain different fields for different Blender windows. Examine it, because sometimes you can discover something interesting. For example — what is the difference between the ***object*** and ***edit_object*** fields? Unfortunately, you still can find nothing about them on the Blender API pages.

Let's examine in the *Variables* pane the ***self*** object. Note that the ***OBJECT_OT_Boolean*** class has a different base class, here. It also has a different value in the ***bl_idname*** field (Figure 4.1.10):

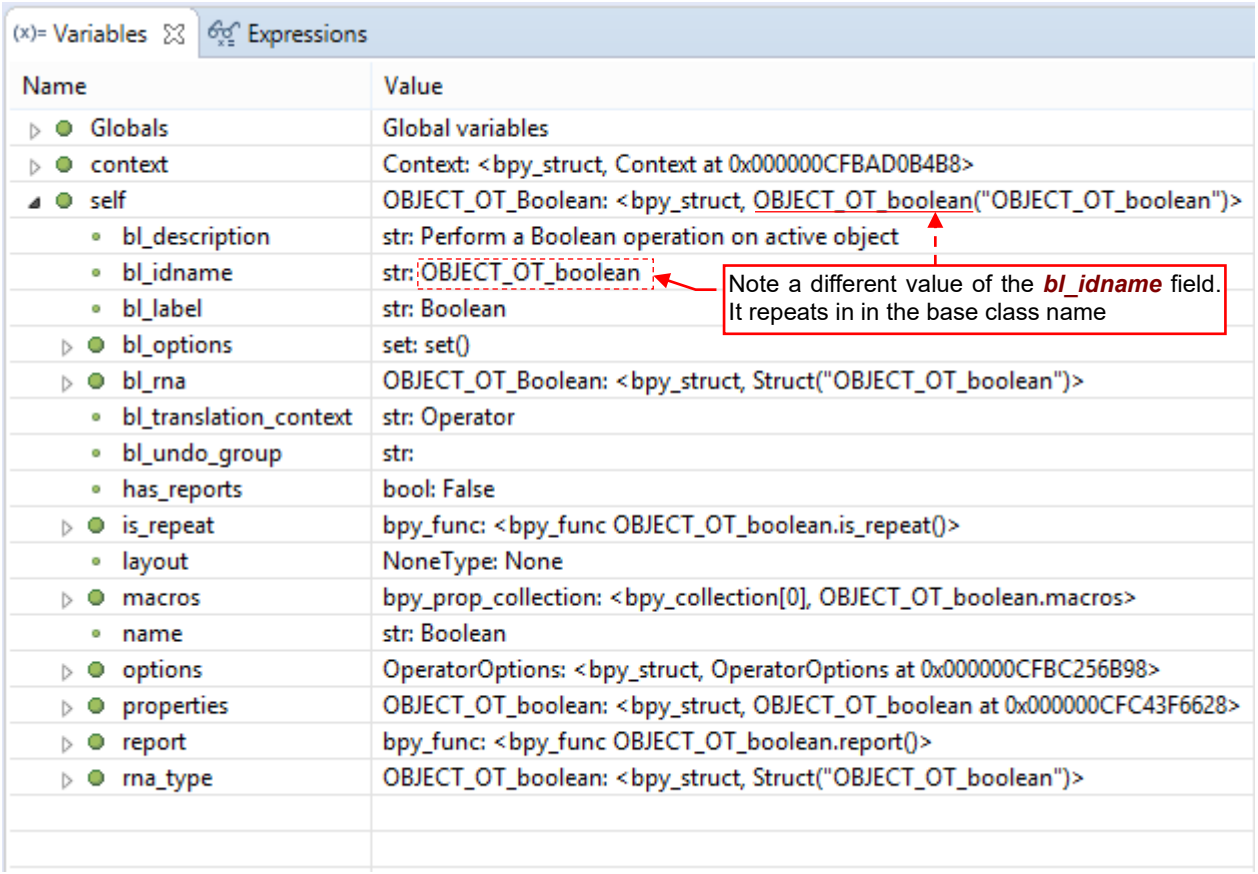


Figure 4.1.10 The content of the operator class (*self*)

Well, this is normal: Blender API modified this class "on the fly". It seems that Blender used the ***bl_idname*** value (**„object.boolean"**) to name the new base class of this operator, named ***OBJECT_OT_boolean***. (The „**object**" is in the uppercase, and the dot („**.**") was replaced with „***_OT_***"). When you examine the content of the ***bpy.types*** namespace (typing ***dir(bpy.types)*** in the *Python Console*, for example), you will see plenty of undocumented classes! Their names always contain „_OT_", „_MT_", or „_PT_". They are the operators, menus and panels created by the internal Blender GUI scripts.

By the way: look at the current state of the Python script stack (Figure 4.1.11). Compare it with the stack that is shown in Figure 3.4.7 (page 55) or in Figure 3.4.9 (page 56).

At the bottom of the stack, you can see the functions of the *Python Console* (it seems that a large part of its code is also written in Python). Then there is a single line from a „*<blender console>*" module. (PyDev converted by a mistake the "<>" characters in its name into *"<>"*) This is my call of the operator ***object.boolean()*** that I typed in the console. As you can see, it called a method from the *ops.py* Blender module, which in turn created this instance of our ***OBJECT_OT_Boolean*** class and called its ***execute()*** method.

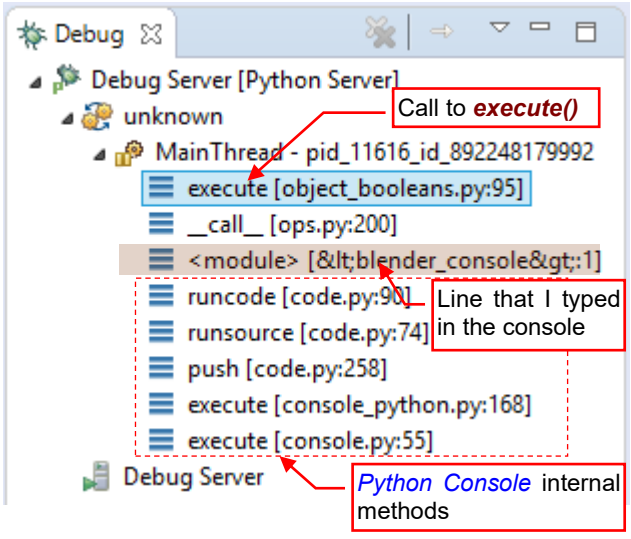


Figure 4.1.11 The stack of the operator called from the console

When you finish the last step of the ***execute()*** function (*Step Over* — F6) PyDev can ask you about the source file for the line typed in the console (Figure 4.1.12). Ignore this request, clicking ***Cancel***:

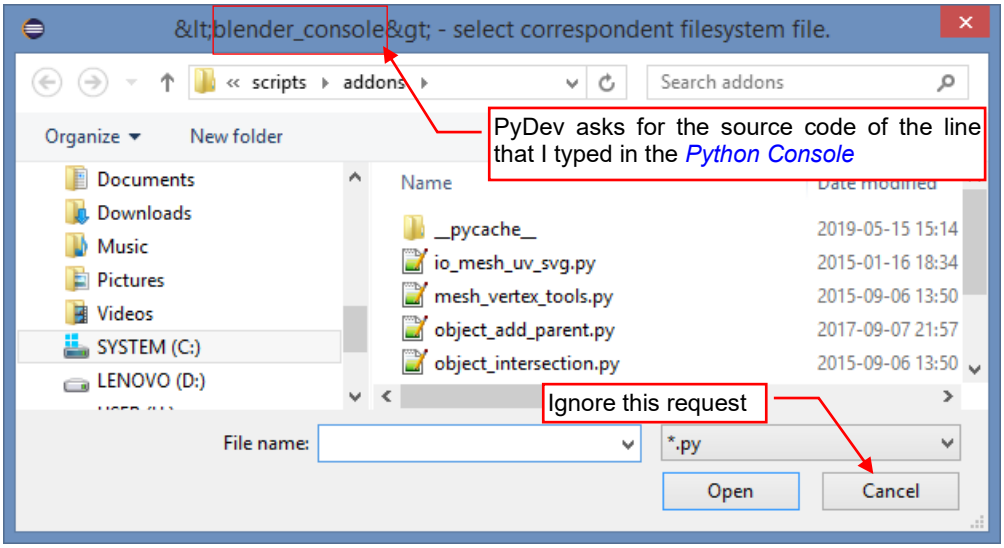


Figure 4.1.12 PyDev request that may occur when you leave the *.execute()* method using *Step Over* command

Now I will show you the behavior of the PyDev debugger in the case of a runtime error in the add-on. When you leave the ***execute()*** method, the highlight disappears from the current line (Figure 4.1.13):

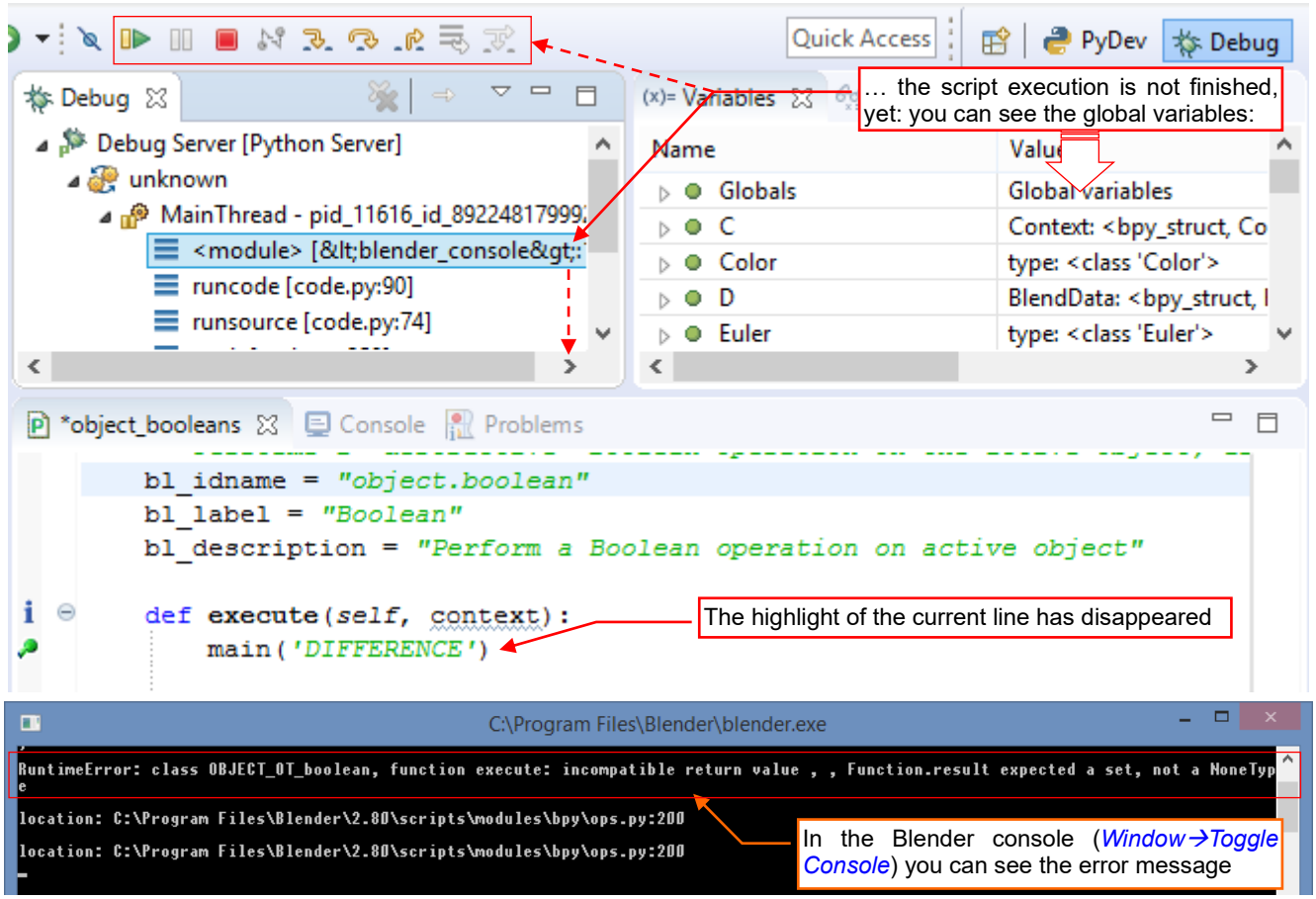


Figure 4.1.13 The state of the debugger in the case of a runtime error

In the same time the debugger prints a message in the console, providing the file name and the line number where the runtime error has occurred. Despite this, the script execution is not completed, yet. In the *Debug* panel you still can see the contents of the stack. In the *Variables* panel you can check the current status of the global variables. However, the more important local variables are already removed from the stack.

In such a case, if you want to terminate the script execution - use the *Resume* command (F8). Then you will see in the console the standard traceback information (Figure 4.1.14):

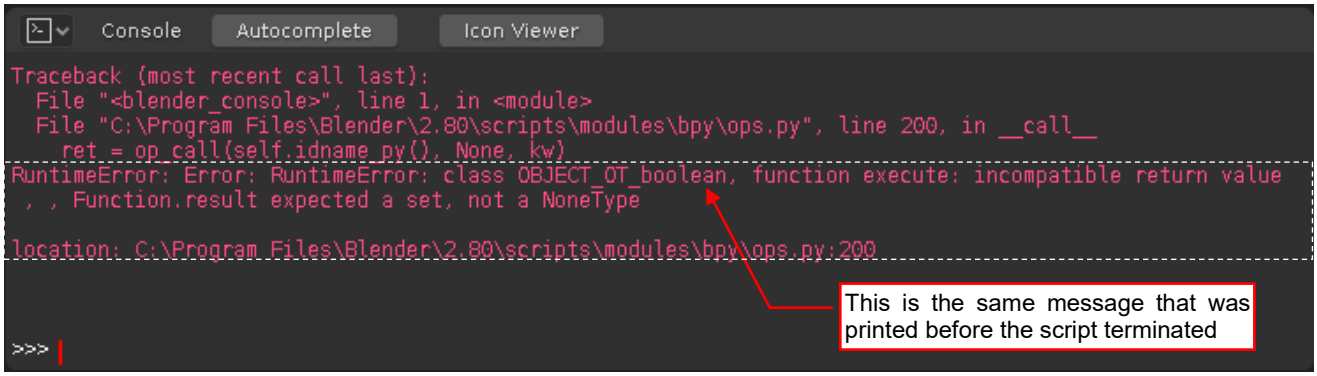


Figure 4.1.14 The full information about the runtime exception

- When you invoke an operator from the *Python Console*, the eventual error information will appear below your call, as in Figure 4.1.14. When you invoke it form a Blender GUI control — a menu or a button — the error message will appear in the Blender *System Console*[1].

In the error message shown in Figure 4.1.14 Blender writes that it expects from the ***execute()*** function a (Python) set instead of the *None* value. Indeed, in a hurry while writing this code I have forgotten completely that the ***execute()*** function must return one of the enumeration values, declared in the API. Usually it returns a single-element set that contains a **'FINISHED'** or **'CANCELLED'** string. (You can find this enumeration in the base class declaration: ***bpy.types.Operator***, in *bpy.pypredef*). OK, so let's fix this script right now (Figure 4.1.15):

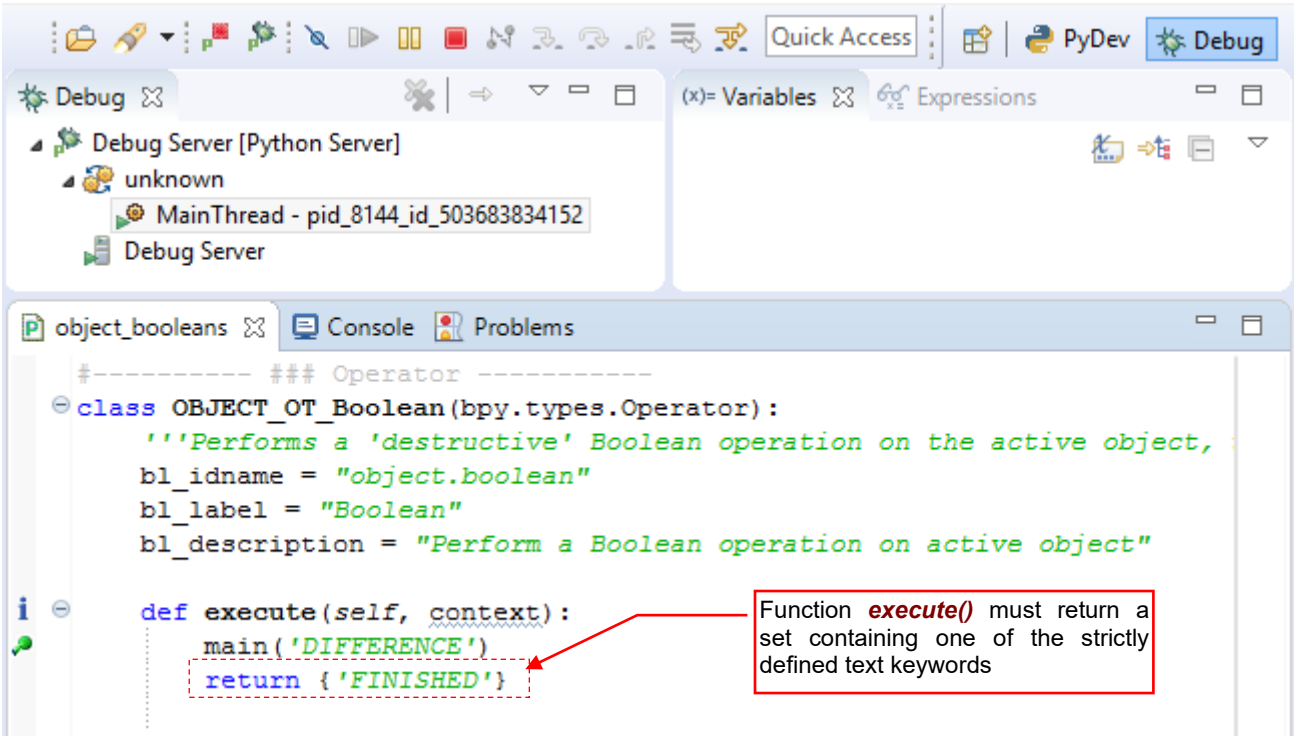


Figure 4.1.15 A quick fix of the code — directly in the *Debug* perspective

Then just save this modified script on the disk.

[1] System Console is an auxiliary (diagnostic) Blender window (i.e. another OS window) which is available for the current Blender session. You can turn its visibility on/off using the *Windows→Toggle System Console* command. In Blender 2.8 you cannot close this window using the standard [x] button. (In the previous Blender versions when you inadvertently clicked this button, you quit Blender).

When you save the script, PyDev also updates the script code that is currently loaded in Blender. You can see information about these updates in the Eclipse and Blender system console (Figure 4.1.16):

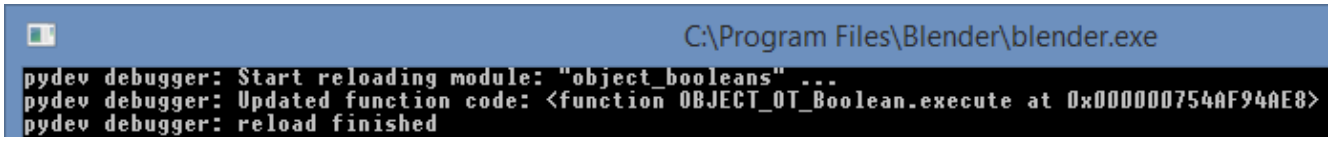


Figure 4.1.16 Information about automatic updates of the script code that is actually loaded in Blender

Regardless of this you can still click the *Run Script* button if you want to unregister/register the add-on.

When the script code is updated, invoke the ***object.boolean()*** operator again (Figure 4.1.17):

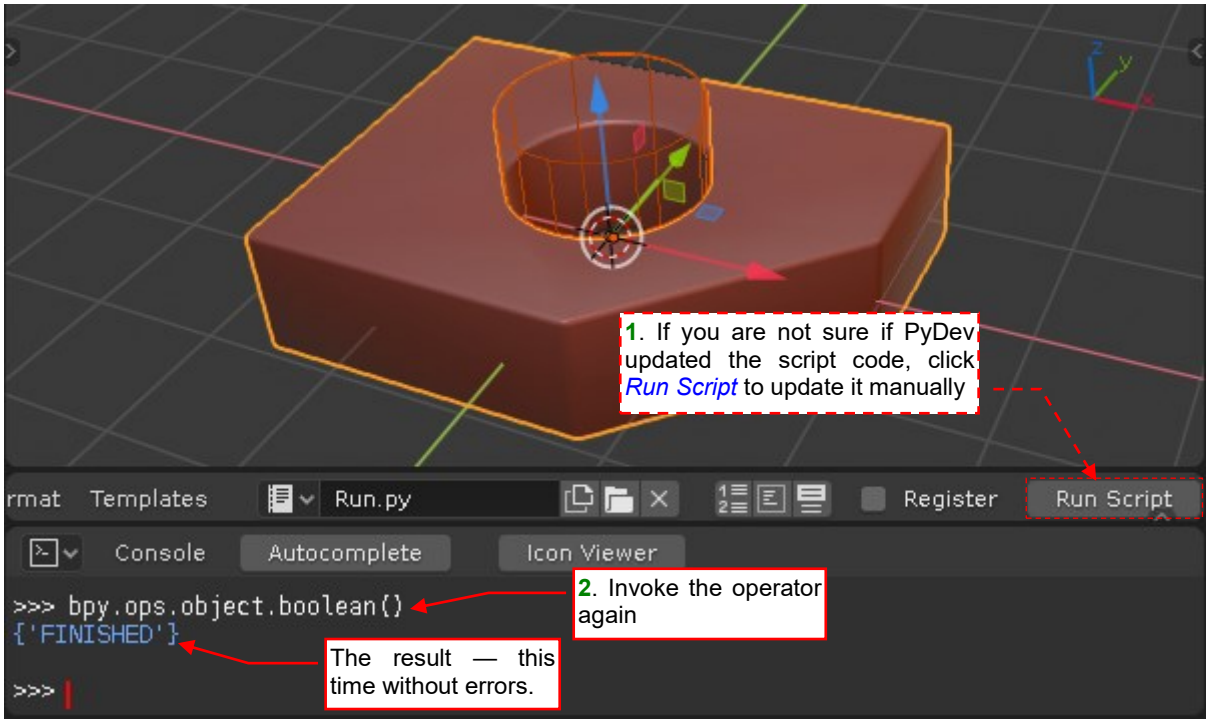


Figure 4.1.17 Another test of the fixed script

As you can see, after this minor code correction our operator works properly.

If you click the *Step Over* (F6) command over the last line of an add-on "public" method, like ***execute()*** – the debugger will step to an internal Blender module named *ops.py* (Figure 4.1.18):

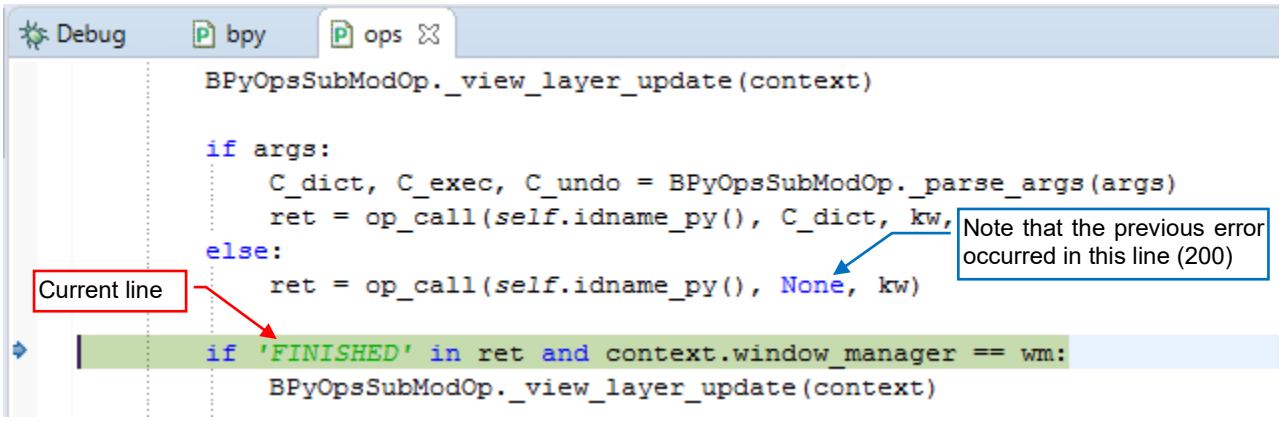


Figure 4.1.18 The internal module *ops.py*, opened after execution of the last script line

This module invoked the ***execute()*** function from our script (see Figure 4.1.11, page 82). In such a case just ***Resume*** (F8) this execution, letting Blender to perform all closing steps.

Finally let's make a small but important modification to function ***main()***: instead of the "static" ***bpy.context*** object, use the ***context*** which Blender passes as the parameter of the ***execute()*** function. They can be different in certain cases! For example – Blender API documentation allows for invoking operators with so-called overridden context. That's why I added to ***main()*** another argument, named ***cntx*** (Figure 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:

            ...
        Remaining code
            ...
```

Additional, optional argument

By default, ***cntx*** refers to the "static" context

Figure 4.1.19 Modification of function *main()* – adding optional *cntx* argument

I modified just the few first lines of this function. Then in the operator class I passed in the ***cntx*** argument the context which function ***execute()*** receives from Blender (Figure 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'}
```

Passing the actual context object

Figure 4.1.20 Modification of function *execute()* – passing the current *context* object to function *main()*

In next section I will add this operator to the *Object* menu.

Summary

- Each add-on must contain the ***bl_info*** structure (page 78). This is the „nameplate", used by Blender to display information about this plugin in the *Blender Preferences:Add-Ons* tab;
- You can convert a procedure that changes Blender data (like our ***main()***) into Blender operator. It involves declaration of a class that derives from ***bpy.types.Operator***. Place the call to the data-updating procedure inside the ***execute()*** method of this new class (page 78);
- Each add-on must implement the ***register()*** and ***unregister()*** script methods (page 79);
- When you run the add-on script, it just registers its presence in Blender API (page 80). You still have to invoke its operator — for example, using the *Python Console* (pages 80 - 81). In response to this call, Blender creates a new instance of the operator class, an invokes its ***execute()*** method;
- In the case of an add-on, the ***Run Script*** button re-registers the latest version of the script. (It calls the ***unregister()*** method for the old version, and then the ***register()*** method from the new one — see pages 85, 160);
- When you have modified and saved the add-on file during a debug session – PyDev will update it also in Blender. You can see the messages about it in the Eclipse and Blender system console[1] (page 85). When they state that the code has been successfully updated – you do not need to reload this add-on "manually", using the *Run Script* button;
- In case of script runtime error (when a runtime exception has been thrown), PyDev debugger breaks the execution (page 83). In this moment you can examine the state of the global Python variables. You can also check the error message in the Blender system console. The same text will be displayed in the Eclipse and Blender console when you terminate this script using the *Resume* command (page 84);
- The information about the environment of the called operator — current selection, active object, etc. — is passed to the ***execute()*** function in the ***context*** argument (page 81);

[1] I mean here the Blender *System Console* window. Do not confuse it with the *Python Console*! It is useful to make this window visible (*Window→Toggle System Console*) before running an API script or an add-on. It displays various diagnostic output (in particular: the output from the ***print()*** statements in your script). This is often very helpful, since the main Blender window is "frozen" until the script terminates.

4.2 Adding operator command to a Blender menu

As you probably noticed in previous section, Blender API requires your operator class to implement strictly defined methods. This is a kind of a "contract" between your script and the Blender core system. You agree to implement required functions in your class. Blender agrees to call them in the strictly defined circumstances.
In the object-oriented programming such a list of contracted functions and properties is called "interface". To help you a little in its implementation, Blender API delivers the base class for derived operators, named ***bpy.types.Operator***[1]. In the object-oriented programming jargon, ***Operator*** is so-called "abstract class". It just provides the default, empty implementations of all the methods required by the interface. Our operator class inherits this default content from its base (***bpy.types.Operator***). That's why it is possible to implement (override, in fact) in the ***OBJECT_OT_Boolean*** just these ***Operator*** methods, which are specific for the derived class.

So far, I overrode the single ***Operator.execute()*** method. It calls the ***main()*** function but ignores the eventual error message that it receives in the result of this call. I did it because in certain situations Blender can repeatedly call this method, for the same context but with different input parameters. (You will see such a case in the next section). Therefore, it is not good place for the result validation, and certainly not for displaying eventual messages. You better implement such a communication in another method of the ***Operator*** interface: ***invoke()*** (Figure 4.2.1):

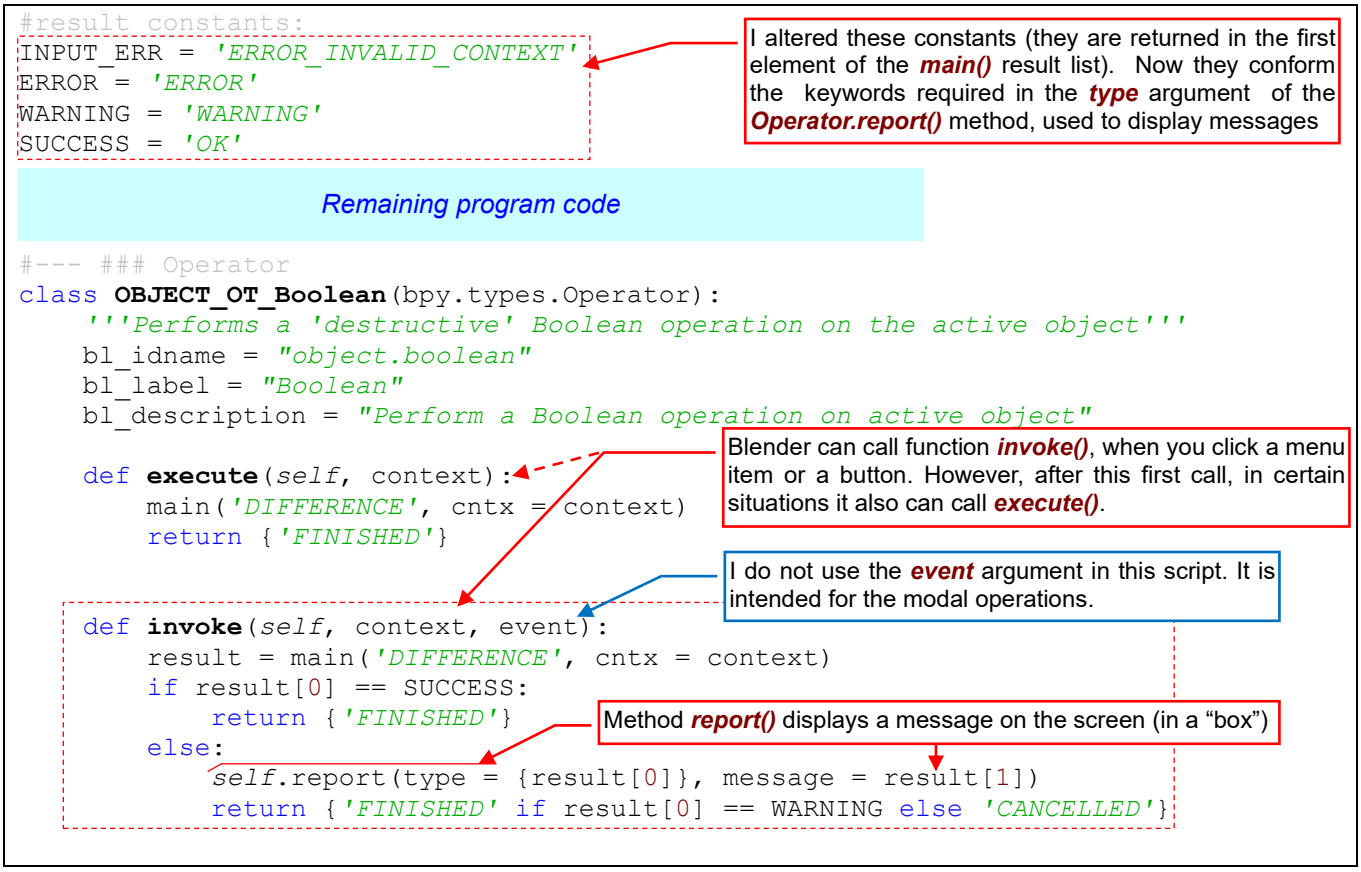


Figure 4.2.1 Validation and user communication — in the *invoke()* method

Function ***invoke()*** returns the same codes as the codes returned by the ***execute()*** function. In this implementation it checks the result returned by function ***main()***. When ***main()*** signalizes a success – it returns **'FINISHED'** (even if a warning has occured). In the case of errors it returns **'CANCELLED'**. For displaying eventual messages I use API ***report()*** function. (I adjusted the result constansts to conform its ***type*** argument).

[1] In addition to the ***Operator*** interface, Blender API provides two other interfaces (abstract classes): ***Menu*** and ***Panel***. Obviously, they are intended for corresponding elements of the user interface. You can find their declarations in the ***bpy.types*** module, as you can see these base classes in the PyDev autocompletion suggestions.

Note that the ***invoke()*** method also receives another argument: ***event***. This object contains information about the user interface "event" — mouse movement or keyboard key state change. It allows creating advanced operators (see examples in the Operator class documentation). In this code I will never use the ***event*** object.

Writing this script, I assumed that our operator will be invoked in the *Object Mode*. In fact, we are going to append it to the mesh *Object* menu, which is only available in this mode. Yet you never know whether someone in the future will add your operator to another menu or panel, and in which Blender mode it will be invoked. Therefore, it is a good idea to implement in your operator class a function named ***poll()*** (Figure 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 uses the ***poll()*** function to check, if this operator is available for the current context. When it returns *False* — its control is grayed out

You must declare ***poll()*** as the class method (instead of the typical instance method)

Function returns *True* when we are in the *Object Mode*. Thus, the ***Boolean*** command will be only available in this mode,

Figure 4.2.2 The basic „availability test" — implementation of the *poll()* function

Blender invokes this function to find out if this command is available "in the current situation". You can examine in your ***poll()*** implementation the ***context*** object passed as one of its arguments. Function ***poll()*** returns *True*, when the operator is available for the given context. Otherwise, it returns *False*.

This is the place for „general" tests, such as the one in the illustration above. This ***poll()*** function returns *True* when Blender is in the *Object Mode*. (This is the meaning of the **'OBJECT'** keyword). If it is in other mode — the armature editing, for example — field ***context.mode*** would return a different value.

I will not check there for more detailed conditions, for example - the types of the selected objects. They are too specific. It would be a very strange command, available only when you have selected two or more mesh objects! Half of the users would have no luck to see it in this state, and they would conclude that this add-on does not work. It is better to make the *Boolean* command available in the *Object* menu all the time. If the user invokes it without any objects selected, it will display an appropriate message. In this way she/he will learn "by example" how to use this command next time.

- Do not use in the ***poll()*** function any method that changes the Blender state (for example the current mode, or the scene data). Any attempt to invoke such an operation here will cause a script runtime error.

Note the *@classmethod* expression before the header of the ***poll()*** function. (In the programmer's jargon, this is called "decorator"). It declares that this is a class method — to run it, you do not need an object instance[1].

- Always add the *@classmethod* "decorator" before the header of the ***poll()*** method! If you omit it, Blender will never call this function.

[1] Probably it improves the performance of Blender environment. The ***poll()*** methods are implemented by all GUI controls, and they are called every time Blender refreshes its screen. (The ***poll()*** functions of all visible controls are called when the user does anything — pulls down a menu, clicks a button, etc.). If ***poll()*** was an instance method, like ***execute()***, Blender would have to create instances of GUI control objects just to call their ***poll()*** methods, and then discard them immediately. I suppose that it would significantly slow down Blender fps rate. For calling a class method you do not need to create its instance (an object), therefore this operation requires less CPU time.

All right, our enhanced operator is ready to use. Yet how to add it to the standard Blender ***Object*** pull-down menu (Figure 4.2.3)?

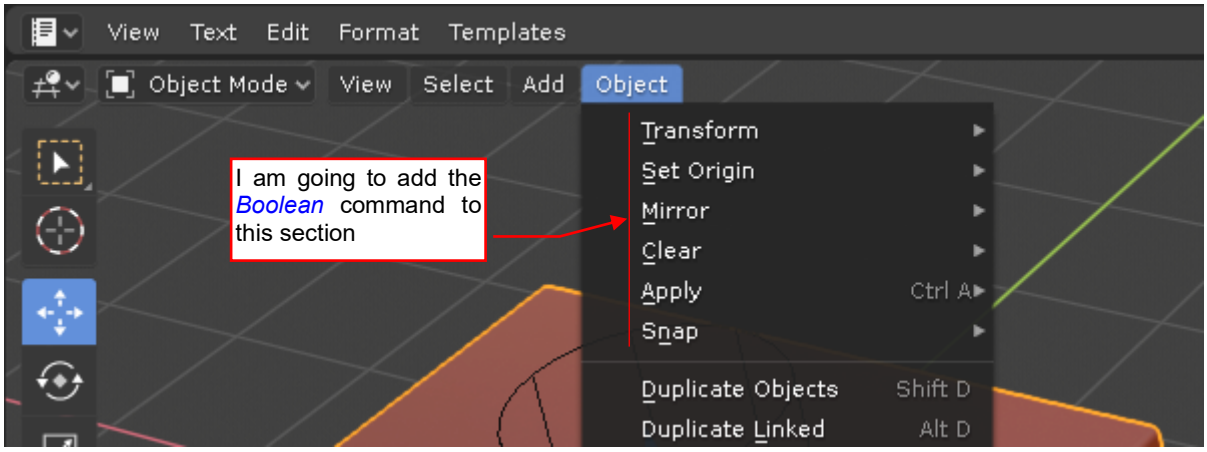


Figure 4.2.3 *Object* menu

The standard Blender menu are created in the same way as our add-on: using API functions and classes. To add an item to menu *Object*, you must find the name of its API class. In such a case, the *Python Tooltip* is an invaluable tool. Just hover your mouse for a while over the menu label (Figure 4.2.4):

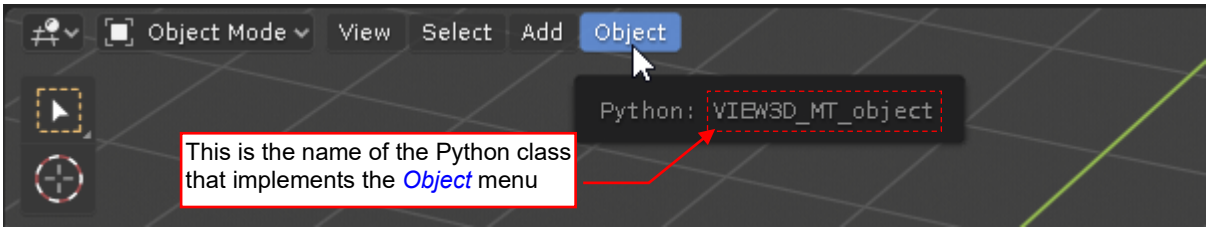


Figure 4.2.4 Identification of the *Object* menu class name

When I know the menu class name, I can write the code that will add our operator to this menu (Figure 4.2.5):

Auxiliary function: it is defined in the main script code, but then added to the class that implements *Object* menu (that's why its first argument is named ***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)
```

This line tells Blender to use the ***invoke()*** method instead of the default ***execute()***

Adds and removes *Boolean* operator from the menu

Figure 4.2.5 Appending the operator command to the *Object* menu

Every pull-down menu class in Blender is based on class ***bpy.types.Menu***. In procedure ***register()*** I am calling the ***prepend()*** method of the *Object* menu class (***VIEW3D_MT_object***). It adds the custom drawing method ***menu_draw()*** at the beginning of this menu. In procedure ***unregister()*** I am reverting this step.

What does the ***menu_draw()*** function contain? Its argument ***self*** is the *Object* menu class. Field ***self.layout*** returns a ***bpy.types.UILayout*** object. It represents the menu "surface" (others also call it "canvas"). The ***operator()*** method places a new command in this layout. This command is identified by its id (field ***bl_name***). However, before this step I alter the layout ***operator_context*** field, setting it to **'INVOKE_REGION_WIN'**. This tells Blender to call the ***invoke()*** method of the operator, so the user will see eventual error/warning messages.

If I used method ***Menu.append()*** in the ***register()*** procedure – the new item would appear at the end of the *Object* menu. In Blender API you cannot place a new item in the middle of menu.

- To identify the class name of a popup menu (like *Context Menu*, opened by RMB click), you have to find its class in the source Blender file. The whole Blender GUI is written in Python, and you can find its scripts in subfolder *scripts\startup\bl_ui*. (On my computer this is *C:\Program Files\Blender\2.80\scripts\startup\bl_ui*). Files named *space_<window type>.py* contain menu API classes for the corresponding Blender window. Thus, you can find the menus of the *View 3D* pane in file *space_view3d.py*. Open this file in a text editor and try to find the name of a specific command from the menu you are searching for. (Sometimes you will find nothing, if the menu uses the default name of this operator. In this case try searching for another item).

Let's check if this code works: reload the script (using the *Run Script* button). It executes the updated ***register()*** method, and in the result you can see our command at the top of the *Object* menu (Figure 4.2.6):

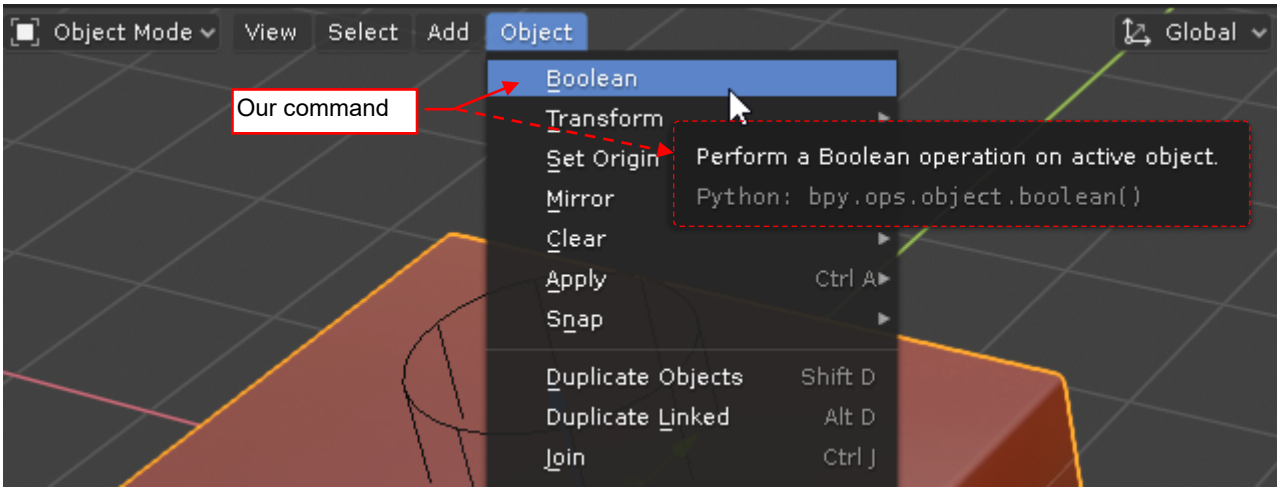


Figure 4.2.6 Our command in the *Object* menu

Let's do another test: invoke the *Boolean* command from this menu when no object is selected. Figure 4.2.7 shows the message displayed by the ***report()*** method, called by our operator (as intended).

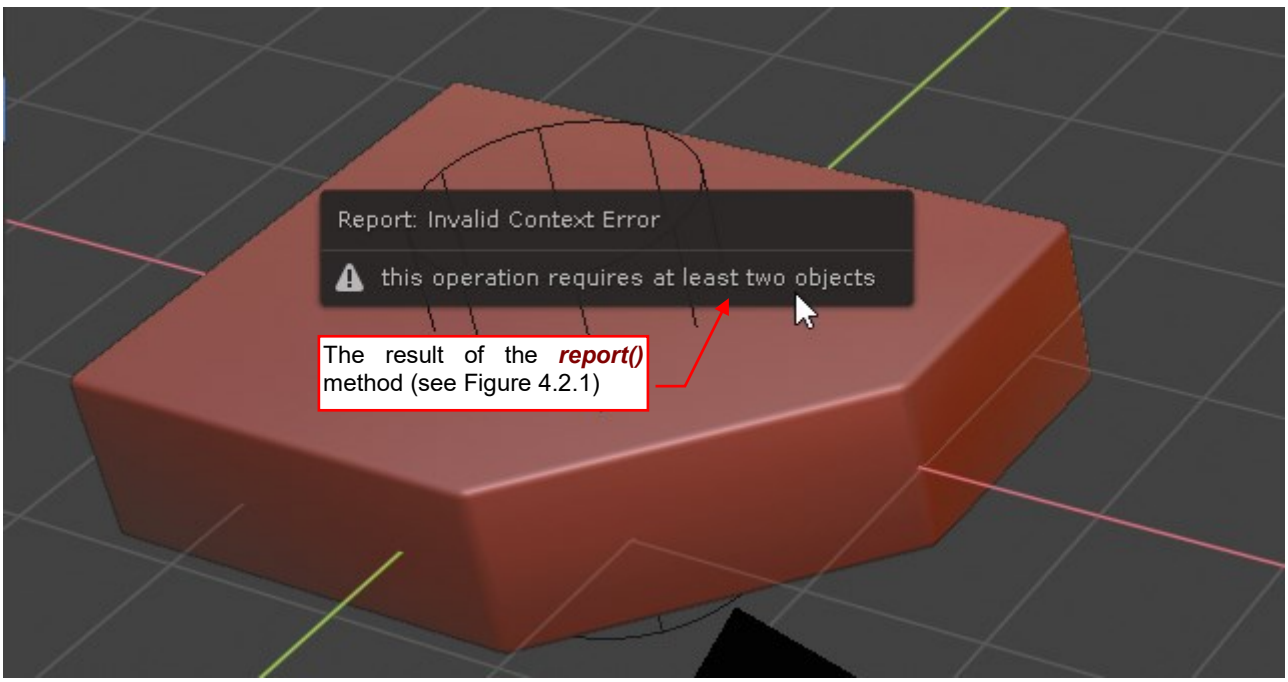


Figure 4.2.7 The result of invoking the *Boolean* command without any selected object

Simultaneously, Blender displays a red line with this message in the *Info* window. (But who, among the ordinary users, looks there?). The warnings are displayed in the *Info* window in orange.

So far, our command performed the Boolean difference operation, because it was “hardwired” in its code. Let’s add a parameter to this operator, providing the user a choice among the three options (Figure 4.2.8):

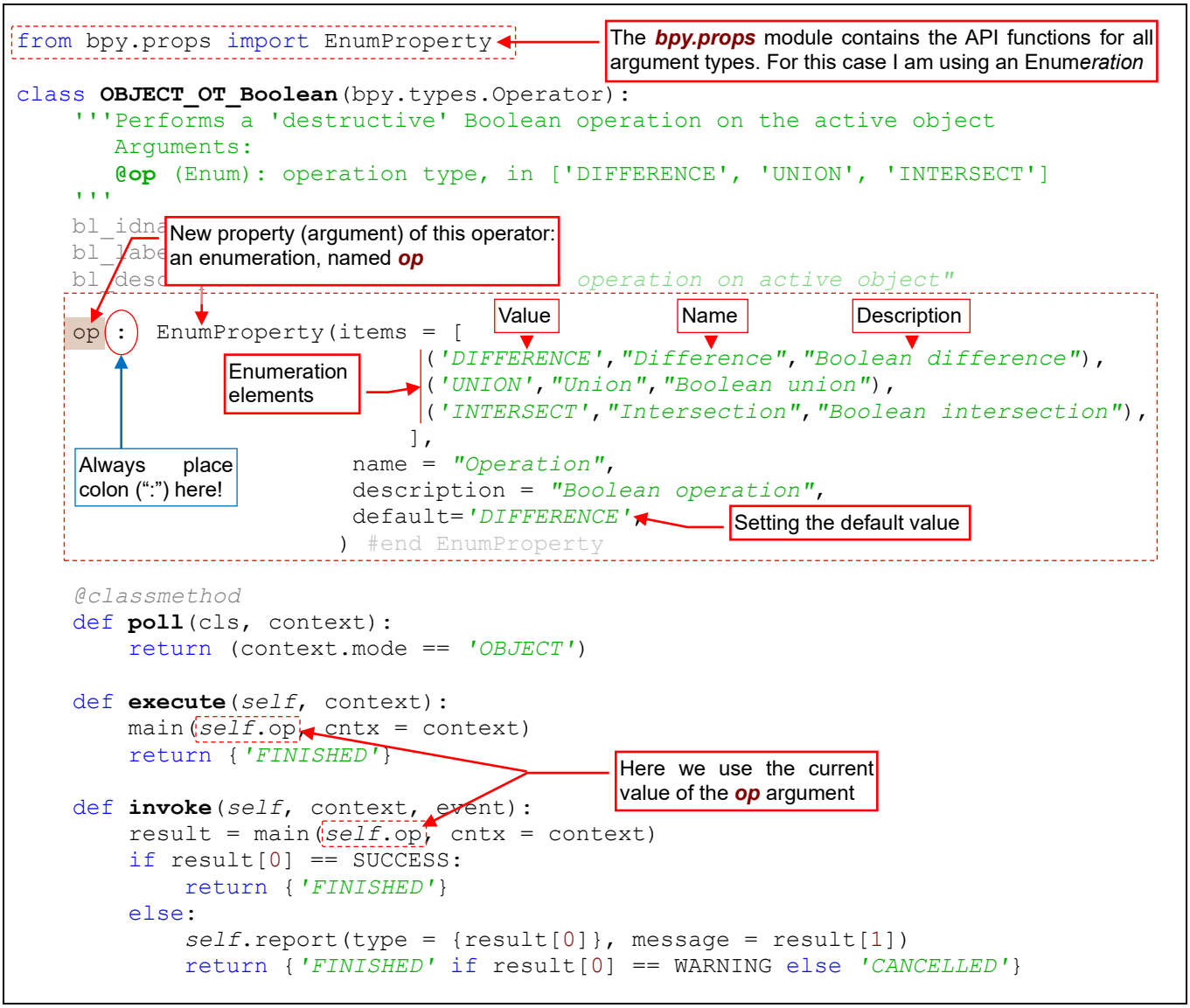


Figure 4.2.8 Declaration of an operator property (operator parameter)

Operator argument, also referred as “property”, is declared as an ordinary class field, assigned by the colon (“:”) to one of the API ***Property** functions. You can find these functions – ***BoolProperty()***, ***FloatProperty()***, ***IntProperty()***, ***StringProperty()***, … – in the ***bpy.props*** module. (In this module you can find a ***Property** function for each of the basic API data types). The ***op*** parameter is an enumeration of the three available Boolean operations, thus I used here the ***EnumProperty*** function.

The most important part of this new code is the definition of the ***op*** enumeration. It is passed to the ***EnumProperty()*** function in its arguments (Figure 4.2.8). In the first argument – ***items*** – I am passing the list that declares enumeration items. Each element of this list is a tuple, containing: the value (e.g. **‘DIFFERENCE’**), name (displayed in the GUI), and description (for the tooltips). From the other optional arguments of ***EnumProperty()***, I am also using ***default***: it determines the default value of the ***op*** operator parameter.

- The tuples of the ***items*** list can also contain two additional (optional) values: the icon name (a string) and the option id (a number). I will show them in one of the further sections.

In the further code, in particular in the object methods, you can use the ***op*** field as any other field of this class. In Figure 4.2.8 I am using it in the ***invoke()*** and ***execute()*** methods, passing its value as the first argument of the ***main()*** function.

Load this new version of the code by clicking the ***Run Script*** button, forcing the re-registration of this add-on. Then, when you type the operator name in *Python Console*, you can see its parameter (Figure 4.2.9)

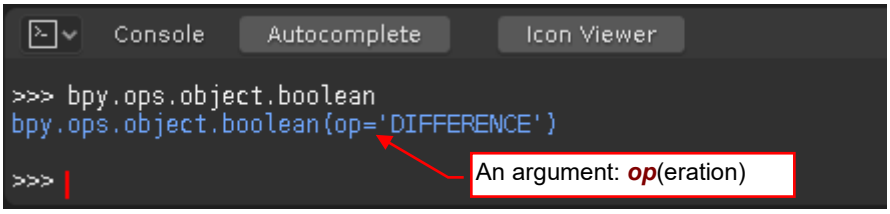


Figure 4.2.9 New argument of the *object.boolean* operator

You can invoke it from this console, for example typing ***bpy.ops.object.boolean(op='UNION')***.

What's more, you can use such an enumeration for an easy conversion of a single menu command (operator item) into a submenu. Each item in this submenu will invoke the same operator with different parameter value. Just change the single line in ***menu_draw()***, as in Figure 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")
```

This method creates a submenu for the items of this enumeration

Figure 4.2.10 Adding a submenu for operator options

Replace the ***layout.operator()*** method with ***layout.operator_menu_enum()***, and pass the name of the enumeration parameter (**"op"**) in the ***property*** argument. Then reload this script. Now, in place of a single *Object→Boolean* command you will see a submenu with all three Boolean operations (Figure 4.2.11):

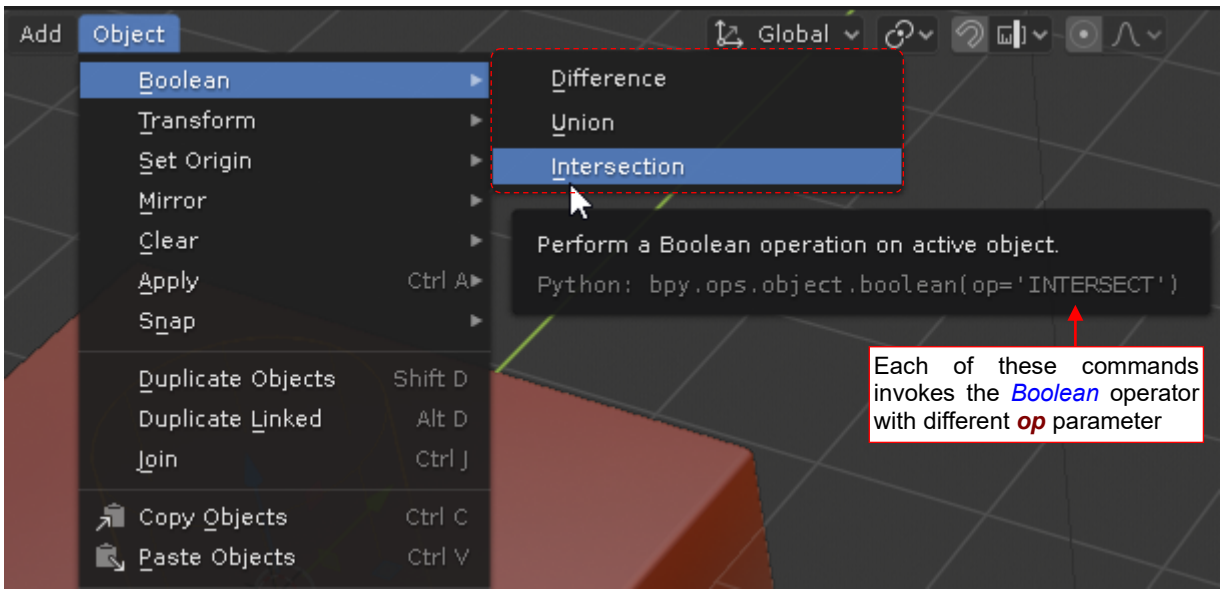


Figure 4.2.11 The *Object→Boolean* submenu

When you check the items in this menu using *Python tooltips*, you will see that each of them invokes the *Boolean* operator with different value of the ***op*** parameter.

After this modification, our add-on became a useful tool. In the next section I will introduce an enhancement: the possibility of a dynamic interaction with the user.

Summary

- Apart the basic ***execute()*** method, your operator class should implement another function: ***invoke()*** (page 88). Keep the ***execute()*** method "mute" and implement the user communication in ***invoke()***. Use the ***Operator.report()*** method for displaying the user warning or error messages;
- To add your operator to a Blender menu, you must know the API class of this menu. Use *Python tooltips* to identify this name (page 90). It is more difficult for the popup menus (like the *Context Menu*). You have to search for their names in the Blender source files. They are named *space_<window name>.py* and located in the *<Blender version>\scripts\startup\bl_ui* directory (page 91)[1];
- To add an operator to Blender menu, define a ***menu_draw()*** procedure that "draws" it on the menu "surface" (layout). Then pass this procedure to the ***Menu.perpend()*** or ***.append()*** methods (page 90);
- When the user clicks your operator label from a menu, Blender by default calls the ***execute()*** method of your operator class. Usually you will want it to use the ***invoke()*** method instead, because it can display eventual messages. To do it, in the ***menu_draw()*** method set the layout ***operator_context*** field to **'INVOKE_REGION_WIN'** keyword before drawing your operator (page 90);
- You can implement in your operator the optional ***poll()*** method. Blender uses this function to check, whether in the current context the command is still available (for example — active in the menu). It is intended for the first, general tests, like the checking the current mode (page 89);
- You can create operator parameter (*property*) as a class field, using appropriate function from the ***bpy.props*** module (page 92). Operator properties, created in this way, become automatically named arguments of the operator method (from the ***bpy.ops*** namespace — see page 93). Then you can use these fields in your code as any other object field that contains a string/boolean/numeric value;
- You can easily create a submenu from an operator enumeration property (initialized using the ***EnumProperty()*** function - page 93);

[1] For example – in the *scripts\startup\bl_ui\space_view3d.py* file you can find that:

- API class of the *Object Context Menu* is named ***VIEW3D_MT_object_context_menu***;
- API class of the *Vertex/Edge/Face Context Menu* is named ***VIEW3D_MT_edit_mesh_context_menu***;

4.3 Dynamic interaction with the user

In Blender, it is very simple to implement a dynamic interaction between your operator and the user— at least a certain, basic scheme of such a cooperation. It allows the user to change continuously the operator parameters (using mouse, for example), while Blender is updating the result on the screen.

All what you have to do is to override the default operator options (field ***Operator.bl_options***). Assign it a set containing two values: {**'REGISTER'**, **'UNDO'**} (Figure 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'}
```

Add this line (it overrides the default options)

Figure 4.3.1 Overriding the operator options

If you omit any element from this the ***bl_options*** set: **'REGISTER'**, or **'UNDO'**, you will not obtain the effect, which is shown in Figure 4.3.2:

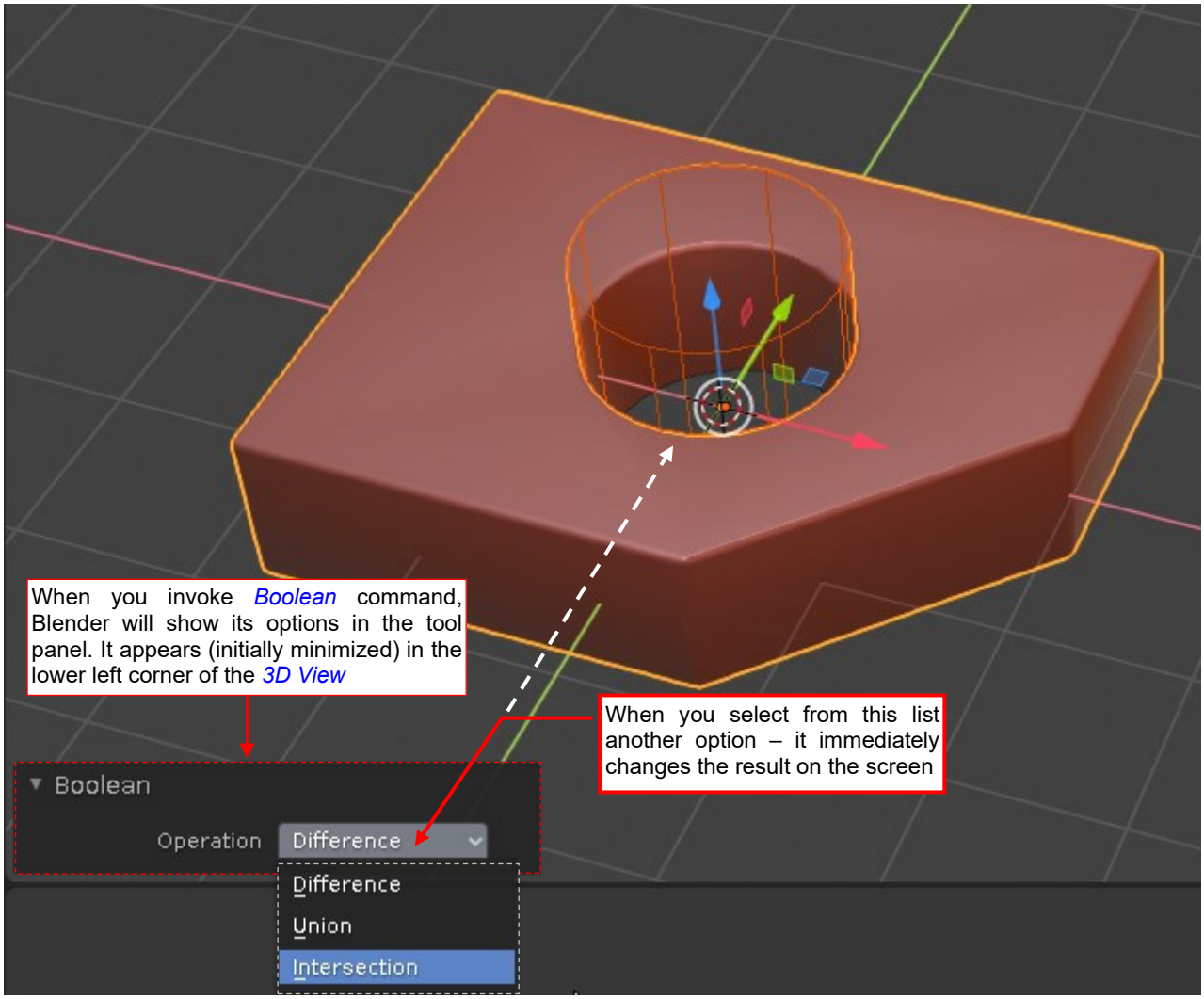


Figure 4.3.2 Dynamic change of the operator options

This is the tool options panel for your operator – as in the standard Blender commands.

When you invoke our command (let's say: *Object→Boolean→Difference*), it creates the hole in the plate, as in the previous trials. However, note that in the lower left corner of the active window Blender has added a bar with our operator name (*Boolean*). When you click this bar, you will find the tool option panel. This panel contains the controls corresponding to all operator parameters (properties). In the case of our script this is just the Boolean operation type. When you alter the value of any control from the tool options panel, Blender immediately updates the operator result you can see on the screen. (Of course, if your operator does not perform any time-consuming calculations). For an operator property that represents a float number (for example: a distance), you can drag the mouse cursor (holding the LMB down) over corresponding control, dynamically changing the result in *3D View*. Unfortunately, our operator has no such float property (i.e. parameter).

How does Blender get this effect from our srcript? For tracking down such interactive events, printing of a diagnostic text in the console is better than the debugger window. Place temporary ***print()*** statements in both operator methods: ***invoke()*** and ***execute()*** (Figure 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'}
```

Diagnostic messages (they will appear in the console)

Figure 4.3.3 Adding the diagnostic messages (just for the test)

Reload this new add-on version and invoke the *Object→Boolean→Difference* command again (Figure 4.3.4):

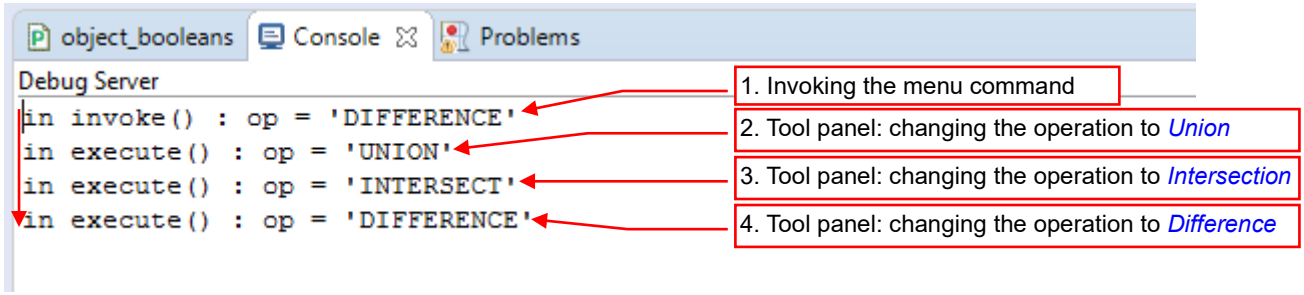


Figure 4.3.4 Diagnostic messages, displayed when I was altering the tool options

Immediately after this invocation, the first line will appear in the console (Figure 4.3.4). It seems that Blender has called the ***invoke()*** method. Now let's change the value of *Operation* field in the *Boolean* pane. After each change, we can see that Blender calls the ***execute()*** method, using the currently selected ***op*** parameter.
It seems that every time I change the value of the tool panel control, Blender calls *Undo* command, and then simply invokes the operator again. For this purpose it uses directly its ***execute()*** method, calling it with the ***op*** parameter set to the current value of the *Operation* control.

I think that the roles of the ***invoke()*** and ***execute()*** procedures in Blender API can be summarized as follows:

- The ***invoke()*** method is called when the operator is executed with the default parameters. The ***execute()*** metod is called when operator is executed for specific parameter values. (In the latter case they are explicitly passed in the argument list of this call).

The choice of the operator methods called by the GUI can be controlled by certain flags (see page 90).

By default, the tool options pane contains every declared API property of your operator. For example I will add to our class another field (API property) named ***modifier***. This is a simple *True*/*False* flag. If it is set to *True*, the operator does not apply the new *Boolean* modifier, added to the modifier stack (Figure 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
    '''
                                Remaining code
    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'}
```

This is a simple Yes/No flag, so I initialized it using function ***BoolProperty()***

Figure 4.3.5 Another property of the operator: (a simple Yes/No flag)

Figure 4.3.6 shows this new control in the tool options panel:

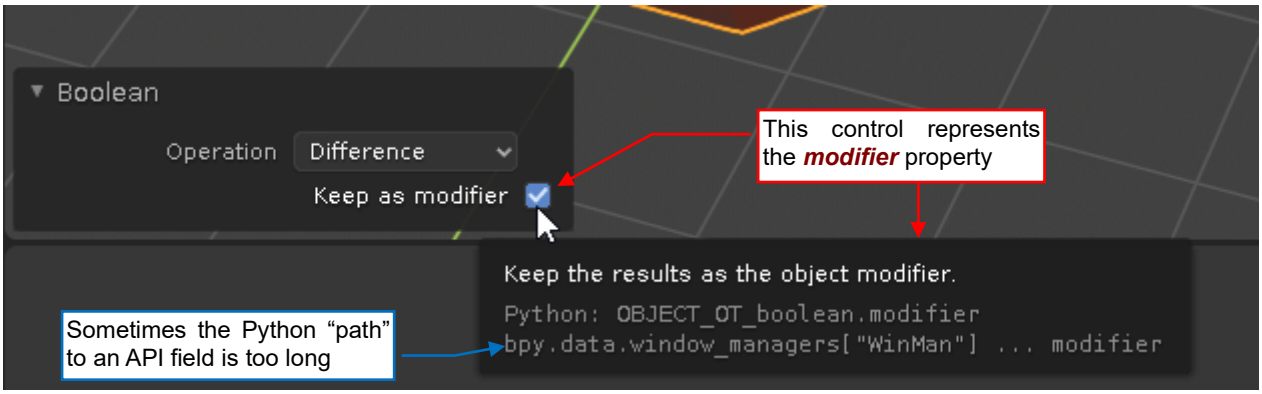


Figure 4.3.6 Modified tool options panel

By default, Blender displays the tool option controls in a single column (one under another) in the same order as they are declared in the operator class. Usually it produces an acceptable visual effect. In the example above, the *Keep as modifier* option seems to be shifted rght because it is placed after the pull-down list, which is left-aligned (as all of the text fields)[1].

[1] If you wish to "take control" over the tool options panel layout of your operator – override the ***Operator.draw()*** method. In this procedure you can implement your own layout of the controls.

If you would like, you can also hide a property from the tool options panel. For this purpose pass to the ***options*** set of the property initalizaton function (****Property()***) keyword **'HIDDEN'** (Figure 4.3.7):

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

'HIDDEN' removes this property from the tool options pane

Figure 4.3.7 Marking an argument as invisible in the tool options pane

Blender uses settings applied in the last completed operator call as the default values for the next call. Of course, this rule does not apply to the properties which are set explicitly, as the ***op*** argument in the *Object→Boolean* submenu. Blender "remembers" these last used values for the timespan of the current session. When you quit Blender and open it anew, for the first call to your operator it will use the default values as defined in your code. To force Blender using this declared default value in every operator call, add another keyword: **'SKIP_SAVE'** to the ***options*** set in the property initialization function.

Summary

- To make your command interactive, just add to its operator class following line: ***bl_options*** = {**'REGISTER'**,**'UNDO'**}. When you invoke it after this change, you will see in the *3D View* the tool options panel containing the command properties (arguments), presented as the GUI controls. You can alter these properties using the keyboard or the mouse. The results of these changes are dynamically updated on the screen (page 96);
- When you click the command button or the menu item, Blender calls the ***invoke()*** method of the corresponding operator. When you alter any property of this command in the tool options panel, Blender calls *Undo*, then the ***execute()*** method of this operator (page 96);
- The ***options*** parameter of the operator property initialization function (****Property()***) contains some useful keywords. To hide a property from the tool options panel, add the **'HIDDEN'** keyword to this set. To exclude property from applying the last used value as the new default, add the **'SKIP_SAVE'** keyword to its options set;

4.4 Keyboard shortcut and a pie menu

When the add-on is tested and it works properly, you can think about further facilities, like the keyboard shortcut for your operator. Of course, first you have to determine the key combination for this shortcut.

Blender is known for its dozens (if not hundreds) keyboard shortcuts. Now I must find among them an unused combination for our command. Frankly speaking, I prefer the rule “the less keys, the better”. Holding down simultaneously three or four keyboard keys (Alt, Ctrl, a letter key, and eventually Shift) is more difficult than typing a single letter key. For the shortcuts, I prefer the keys from the left side of the keyboard, because most of the users keeps the mouse in their right hand, leaving the left hand free for most of the time. From the other side – if I was going to assign to my operator a keyboard shortcut “hardcoded” in the script, it had to be a unique (thus: complex) key combination. In this way I can avoid potential conflicts with other plugins and Blender standard shortcuts. Most of the add-on authors do it this way. However, I will choose a different approach:

- I will allow the user to determine the keyboard shortcut for the *Boolean* command. For this purpose, I will implement a special add-on preferences panel (in the next section of this guide). Such a facility allows me to propose a simple key combination for the default shortcut.

To determine a suitable, unused key combination I prepared the test environment: in the *3D View* window, *Object Mode*, I selected a few scene objects. Then I started typing single letter keys on the left side of the keyboard: Q, W, E, R, A, S, D, F, Z, X, C, … checking, what happens. On this occasion I learned about a few Blender facilities that were unknown for me, like the *Quick Favorites* menu (under the Q key), or switching the active tool variants (W):

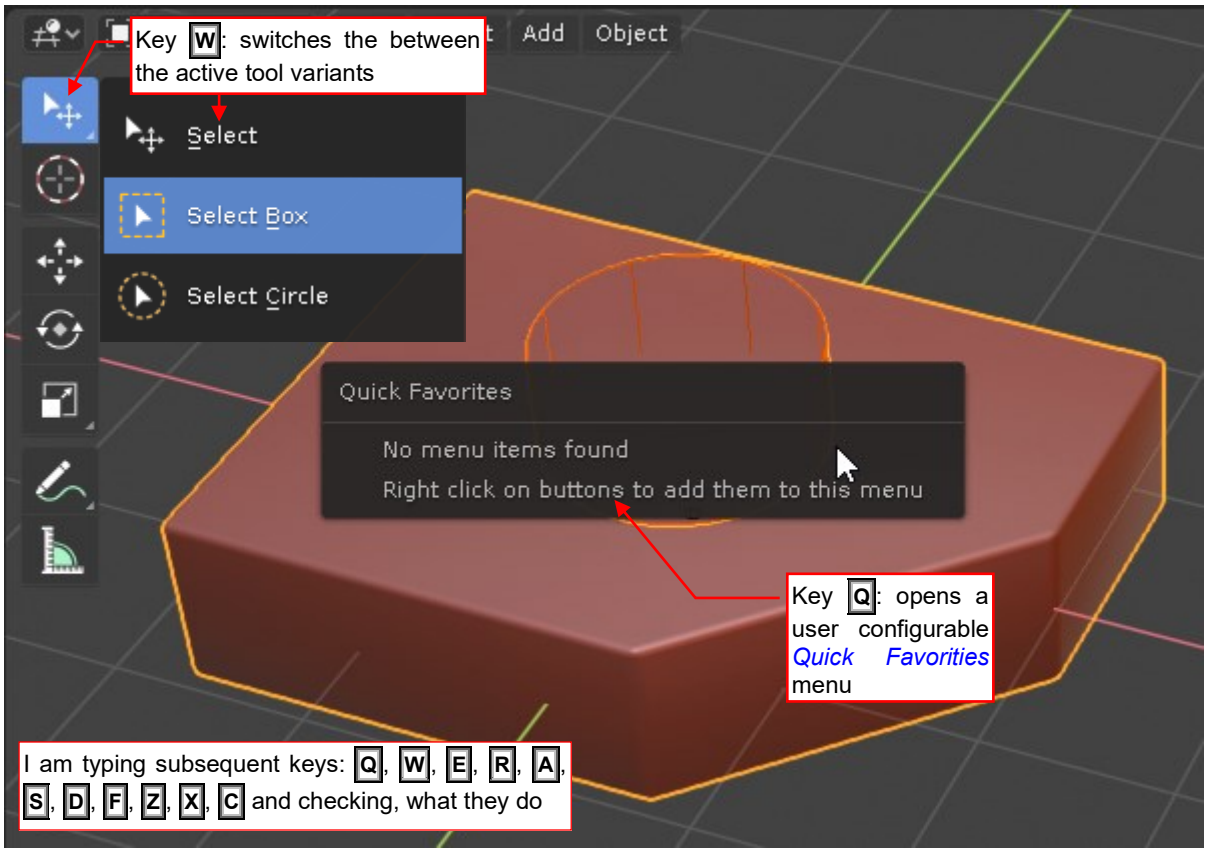


Figure 4.4.1 Searching for an unused kay for the keyboard shortcut

Surprisingly, I have found that the E, D, and F keys are not (yet) assigned to any command. After this preliminary elimination, I have to check this short list of free keys in the Blender preferences window.

In the preferences window (*Edit→Preferences*), ***Keymap*** tab, I searched for all the shortcuts that use each of the keys that I selected in the previous step (Figure 4.4.2 shows the results for key D):

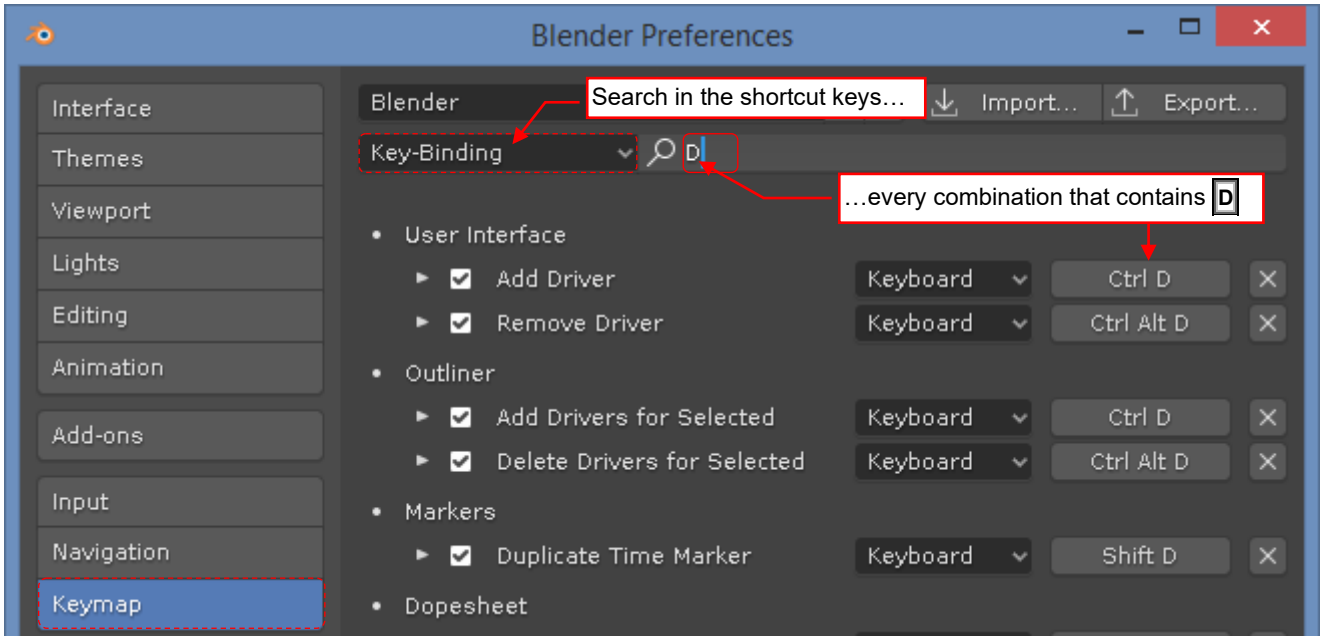


Figure 4.4.2 Checking the existing shortcuts that use a given key

I paid special attention to the number and type of the Blender modes that use these shortcuts. Ultimately, I decided to use the D key as the default keyboard shortcut for this operator. It is used just in two modal projection changes: *View 3D Fly Modal* and *View 3D Walk Modal*, thus there will be no conflict. (Shortcuts E and F are used more often, although in completely different windows and/or modes. However, because of this more frequent use, they could cause more mistakes among the users).

I prepared two auxiliary procedures that register and remove the keyboard shortcut (Figure 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()
```

A global auxiliary list that preserves the registered shortcuts (while this add-on is active)

addon is a special keyboard configuration for the plugins

Auxiliary variables (for a more readable code)

I am checking this just in case

Use here the name from *Preferences:Keymap*

Shortcut definition

Add the shortcut to Blender keymap

Remove the shortcut from Blender keymap

Clear the auxiliary shortcut list

Figure 4.4.3 Keyboard shortcut registration

These are simple methods without any parameters, because at this moment the shortcut is “hardcoded”. The ***register_keymap()*** method assigns the shortcut key (**‘D’**) to the *Boolean* operator and saves the newly created keymap and hotkey in the auxiliary ***addon_keymaps*** list. Method ***unregister_keymap()*** removes the shortcut assigned to this operator and clears the ***addon_keymaps*** list.

I placed calls to ***register_keymap()*** / ***unregister_keymap()*** procedures in the ***register()*** / ***unregister()*** methods (Figure 4.4.4):

```
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)
```

Procedures that register/unregister keyboard shortcut

Figure 4.4.4 Calling the keyboard shortcut registration methods

(Note that ***unregister()*** executes the corresponding steps in the reverse order than they are invoked in the ***register()*** method. In this way I am avoiding using an API class that is not yet registered, or just removed).

When you click the *Run Script* button, this shortcut will appear in Blender *Keymap* list (Figure 4.4.5):

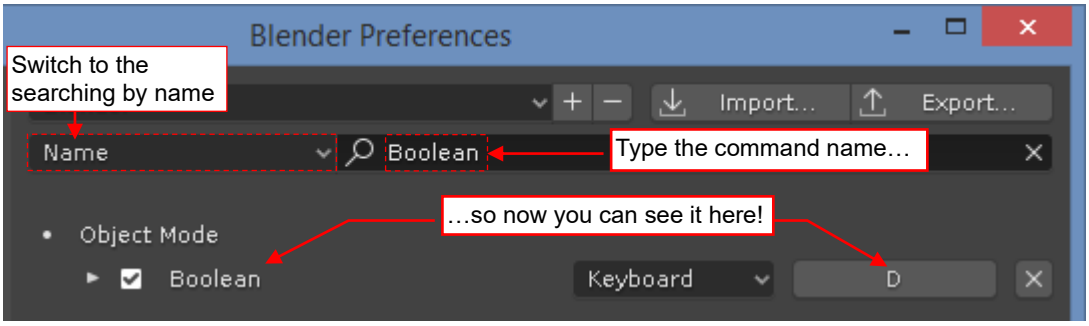


Figure 4.4.5 Shortcut to the *Boolean* command

Thus, when in *3D View* you select objects **Cylinder** and **Cube**, and then type D, you will make a hole:

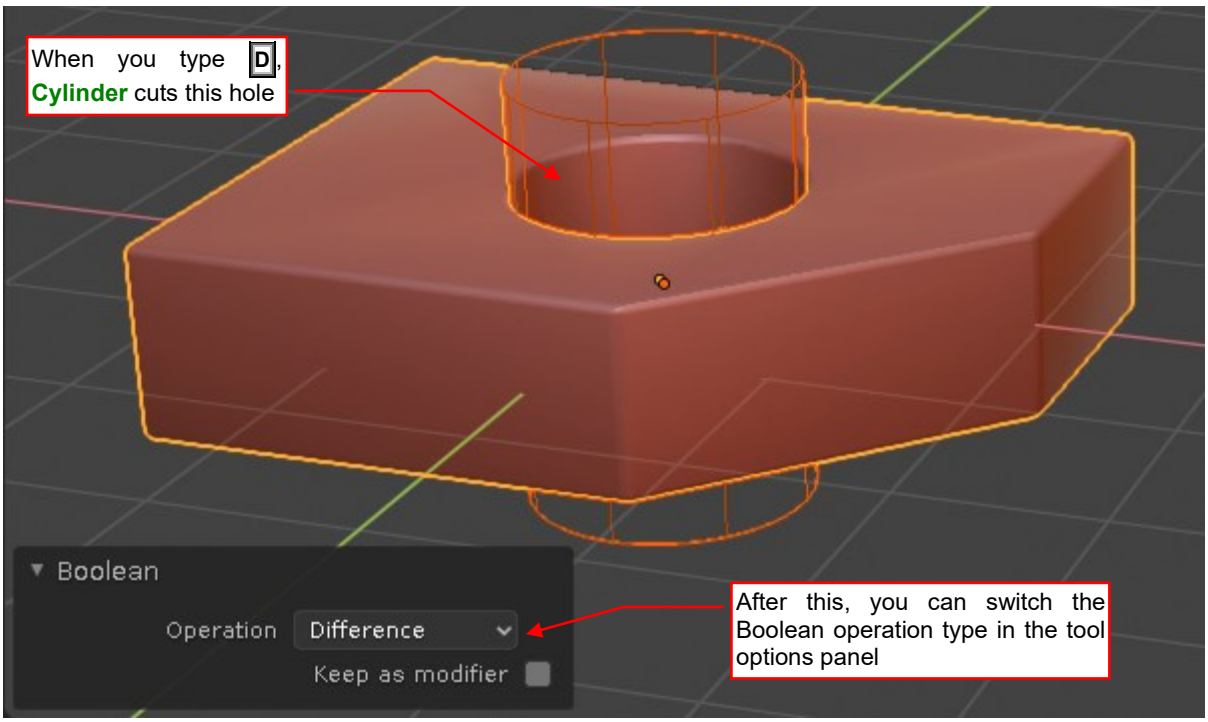


Figure 4.4.6 Invoking the *Boolean* operator using keyboard shortcut

This command was invoked with the default ***op*** parameter, so Blender uses here the last used value of this property. You can switch it later in the tool options panel (Figure 4.4.6).

I think that the users would prefer selecting the type of the Boolean operation immediately after pressing the shortcut key. It could be a classic popup menu offering the three available options. However, in Blender we can use for the same purpose a more elegant pie menu (as shown in page 37, Figure 3.1.8).

The API class for a pie menu differs just in few details from the implementation of a classic popup/pull down menu (Figure 4.4.7):

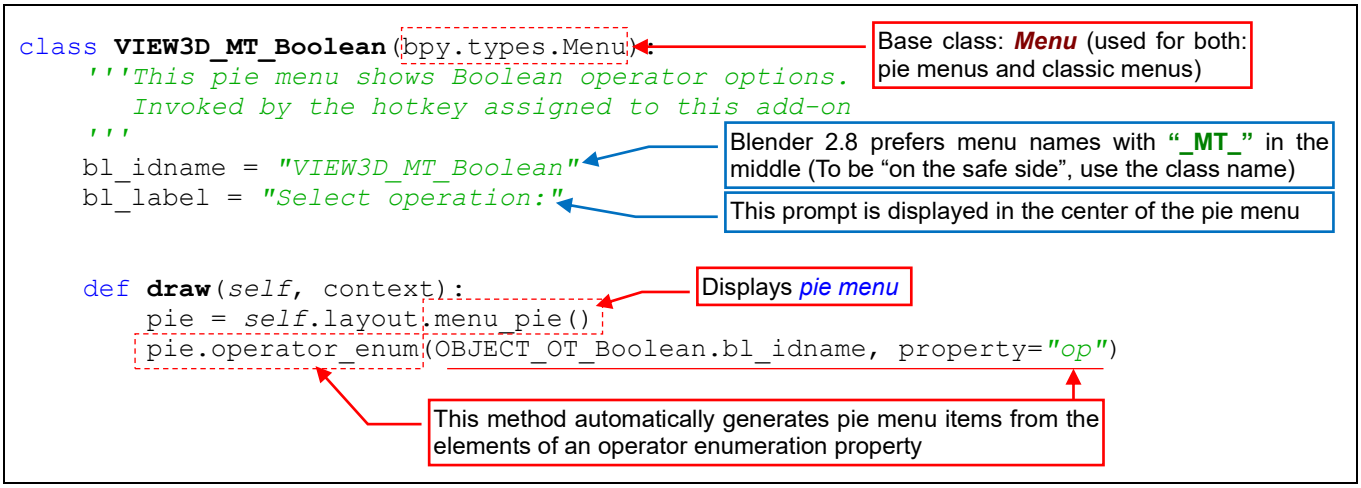


Figure 4.4.7 Implementation of a pie menu that exposes the *Boolean* operator options

I named this class ***VIEW3D_MT_Boolean***. From the programmer point of view, the only difference in implementations of a classic menu and a pie menu is in their ***draw()*** methods. In this class I begin the ***draw()*** method by calling ***layout.menu_pie()***. It prepares (initially empty) pie menu. Then I allow Blender to generate items for this menu from the ***op*** enumeration property of the *Boolean* operator.

This new menu will be opened by our keyboard shortcut (instead of the *Boolean* operator). Thus, I modified the ***register_keympa()*** procedure. In place of the *Boolean* operator id I placed the name of the special ***wm.call_menu_pie*** operator. It opens the pie menu, which name I assign to its ***name*** property (Figure 4.4.8):

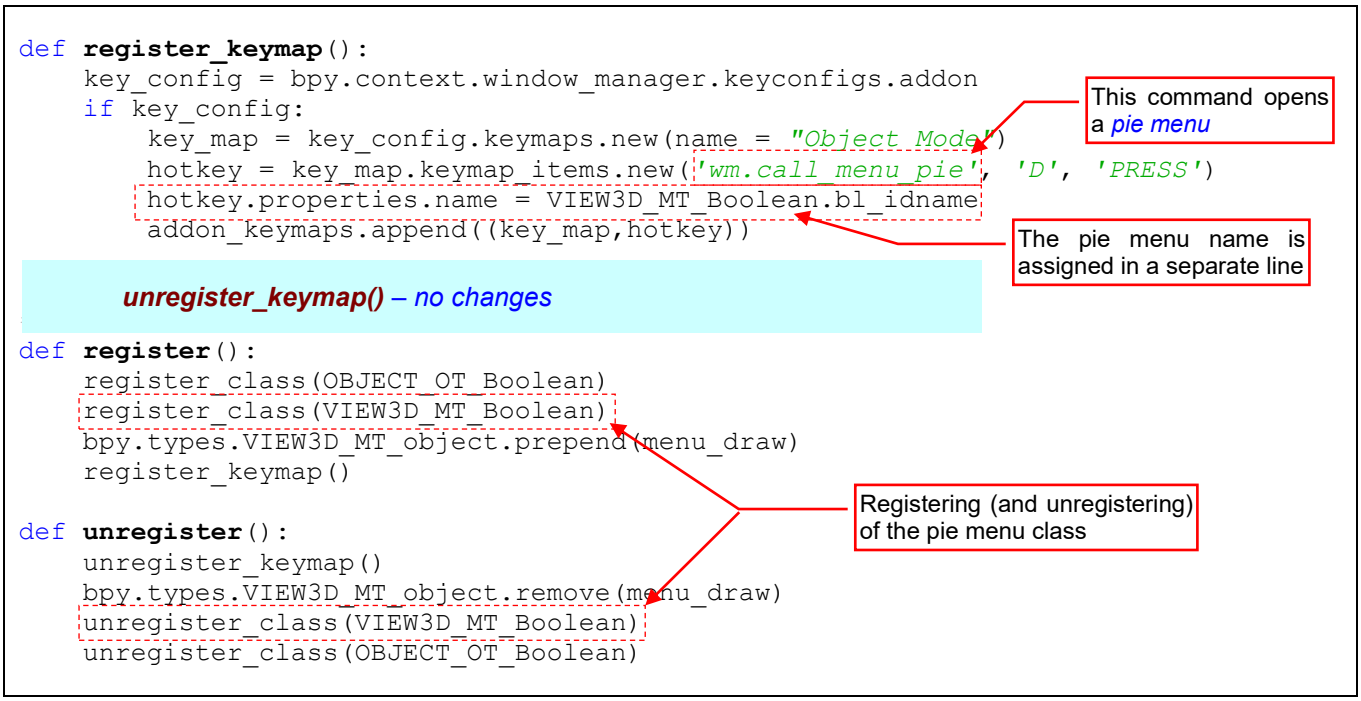


Figure 4.4.8 Registration of a pie menu

I also added to ***register()*** and ***unregister()*** methods additional lines that handle registration of the pie menu API class (***VIEW3D_MT_Boolean***).

To check, how it works now, reload this script (clicking the *Run Script* button). Then go to the *3D View* window and press the D key (Figure 4.4.9):

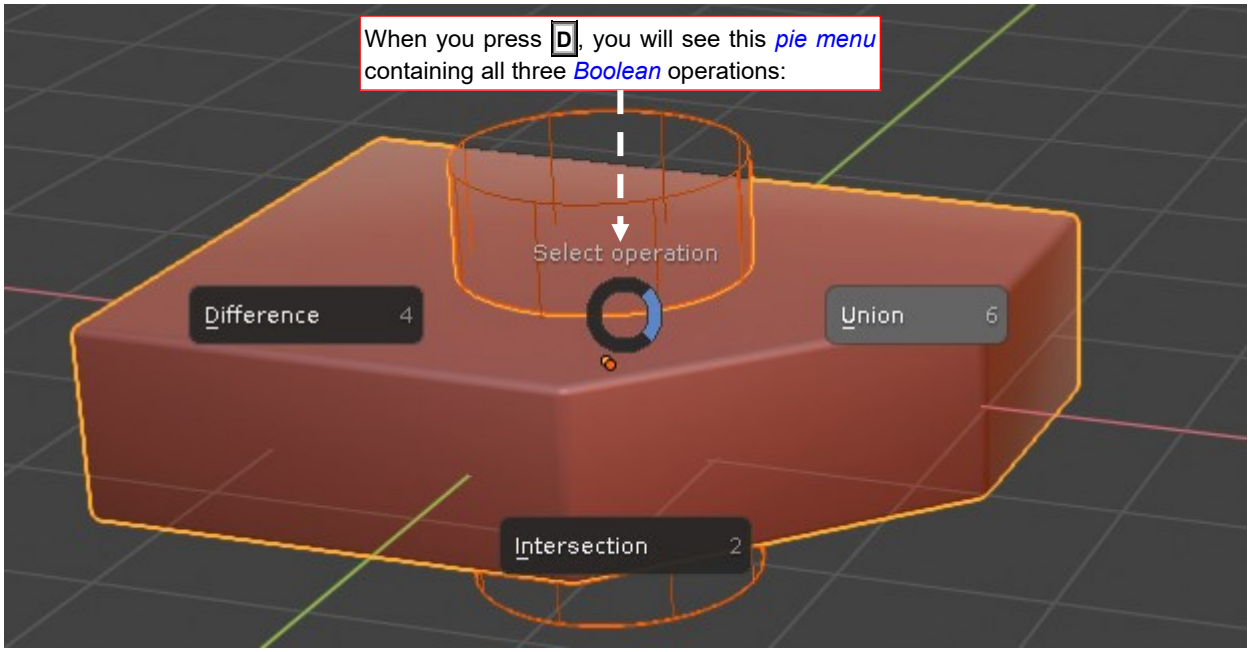


Figure 4.4.9 Pie menu (first version)

It looks quite good, but let's improve its appearance, adding icons to these labels. The simplest way is to choose corresponding icons from the standard Blender set. Click the ***Icon Viewer*** button in the *Python Console* window to open its browser (Figure 4.4.10):

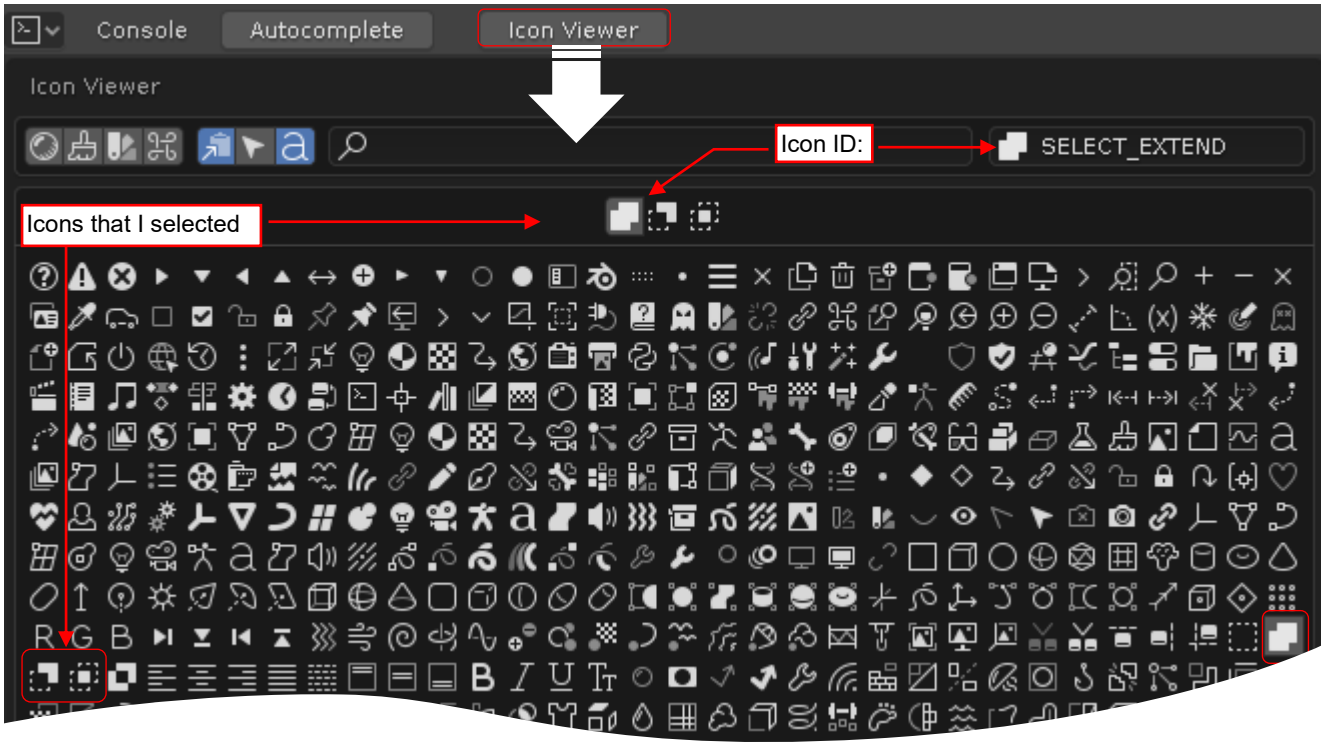


Figure 4.4.10 Blender icon browser

Unfortunately, the *Boolean* modifier options are not "iconized" (yet?). After long deliberations I decided to use the icons intended for the operations on a selection set. I read their identifiers from the field at the upper right corner of the icon browser window: **'SELECT_EXTEND'** (*Union*), **'SELECT_SUBSTRACT'** (*Difference*), **'SELECT_INTERSECT'** (*Intersection*). I do not think that they are especially pretty, but at least they match the idea of these three Boolean operations.

The most convenient way to add these icons to our implementation is to extend the definitions of the ***op*** enumeration in the ***OBJECT_OT_Boolean*** class (Figure 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
```

In each tuple in this enumeration I added two new values: icon id (string) and ordinal number (integer)

Figure 4.4.11 Adding icons to the tuples in the *op* enumeration

Note that in every tuple of this enumeration the icon id is accompanied by an ordinal number. They are required by the API – if I added only the icon id, Blender would raise an exception in the ***register()*** method.

Let's reload this script again (*Run Script*) and type the D key (Figure 4.4.12):

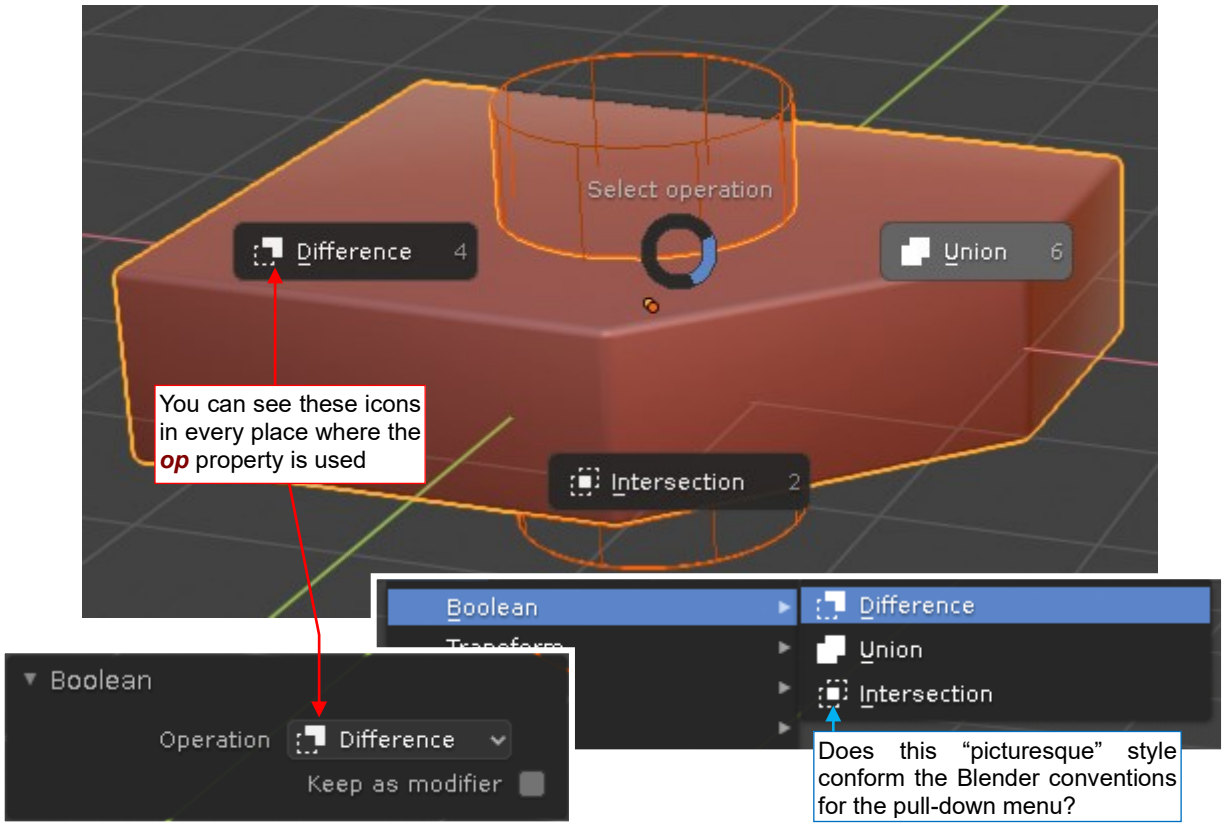


Figure 4.4.12 Results of the "iconized" *op* enumeration

The icons appeared everywhere – not only in the pie menu, but also in tool option panel and pull-down menu.

Looking at the modified *Boolean* submenu, I started to wonder if such an "iconic" style conforms the Blender conventions for pull-down menus? (I could not see any other icons in the neighbor *Transform*, *Set Origin*, *Mirror*, and other submenus of the *Object* menu). However, I made a more detailed review of this and other Blender pull-down menus and concluded that although most of the *Object* submenu items have no icons, there are some exceptions. For example – in the *Object→Convert To* submenu. You can also find more icons in the other pull-down menus of the *3D View*.

Summary

- The complex keyboard shortcuts of three or four keyboard keys are inconvenient. Try to use shorter combinations of two, or even a single key;
- To assign a keyboard shortcut to your operator, you have to find it first. Start by creating a "short list" of the most promising, unused key combinations. You can do it by typing all the possible keys in a test Blender environment (page 99). (This environment should closely resemble the real environment where the users will apply your operator). Then determine (using the Blender keymaps – as on page 100) the least used shortcut and use it in your add-on;
- Assign the shortcut to your operator in a new so-called "key map", in the keyboard configuration named ***bpy.context.window_manager.keyconfigs.addon*** (page 100). Do it in the ***register()*** procedure and save these objects in a global variable. You will need them in the ***unregister()*** procedure (page 101);
- When your operator has several variants, as *Boolean* in this example, it is a good idea to assign the keyboard shortcut to a pie menu, which in turn invoke the operation selected by the user (page 102, 103);
- You can use the standard Blender icons in your user interface (page 103, 104);

4.5 Implementation of the add-on preferences panel

In principle, our *object_booleans.py* add-on is ready. In this section I will just add a small utility: add-on preference panel that will allow the user to change the shortcut key assigned to the *Boolean* operator.

To test the preference panel, I need to install this add-on. I did it in the *Blender Preferences* window (*Edit→Preferences*), clicking the ***Install...*** button (Figure 4.5.2):

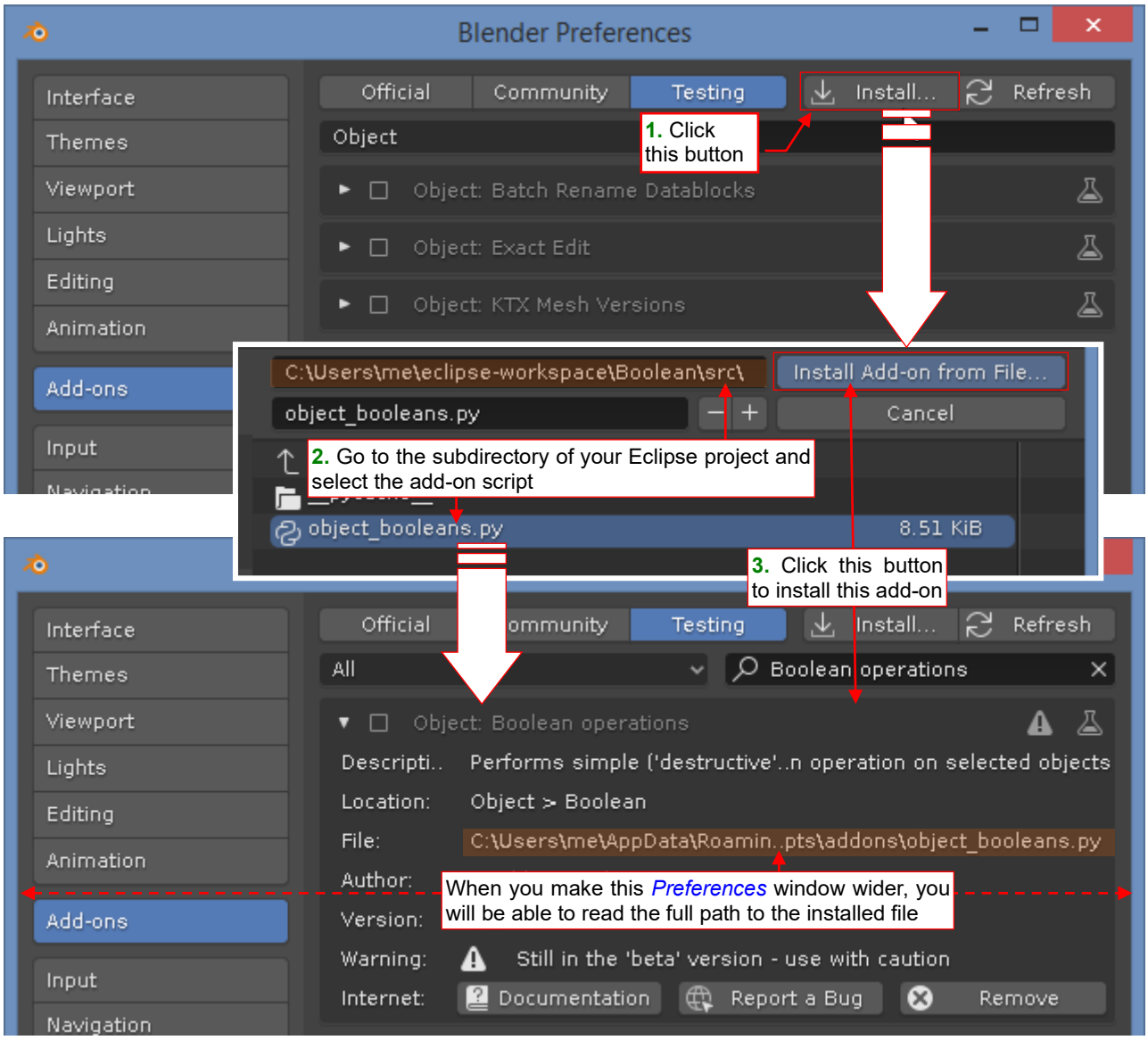


Figure 4.5.2 Add-on installation

In the file selection window, I opened the Eclipse project folder and in the *src* subdirectory I selected the *object_booleans.py* file. During this "installation", Blender copies this file into its add-on folder for the user add-ons. (You can read the full path to this copy from the *File* field, displayed in the add-on panel).

In my Eclipse project I created an "archive" folder for unused files named *prev* and moved there the original script file. Then in the *src* folder I added a link to the add-on file that I installed in Blender. (I did this using *File→New* command, as described on page 146). Now this installed Blender add-on file is simultaneously the current script in this Eclipse project.

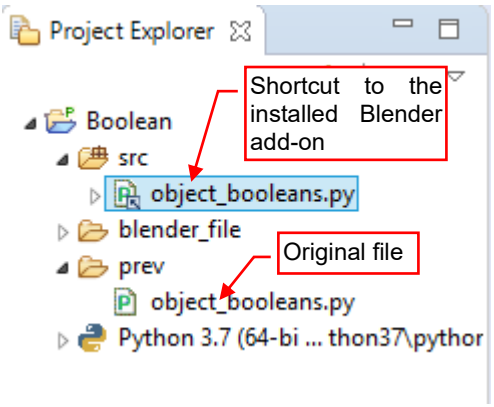


Figure 4.5.1 Shortcut to Blender add-on file

You cannot use the *Run.py* script for debugging an installed Blender add-on, as we did in the previous sections of this guide. That's why I added a few new lines of code at the beginning of the *object_booleans.py* file. They will connect it to the PyDev remote debugger (Figure 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 ------------------------------------------
```

Auxiliary constant: set to 0 in the final version (or 2, when it is loaded using the *Run.py* script)

This is the ***PYDEV_PATH*** from *Run.py* (see page 53)

to import this module, I am adding the full path to */pysrc* directory to **PYTHONPATH**

Invoke PyDev remote debugger client

Figure 4.5.3 Initialization of the PyDev debugger client in the installed add-on script

- Remember that an active (i.e. enabled) add-on is being loaded and registered during Blender initialization. That's why you have to run the remote PyDev debugger in Eclipse before you open the test **.blend* file.

After these preparations I added to this plugin an API class that implements the preferences panel. It extends the ***AddonPreferences*** base class (Figure 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")
```

It must inherit from this base class

For this id always use the name of this script file

Properties (panel controls)

These functions initialize the panel controls

This expression generates for every letter from **'A'**… **'Z'** range the three-element tuples: ***('A', 'A', "[A] key") ('B', 'B', "[B] key"), …*** and so on.

The simplest function that places the class properties on the panel

Figure 4.5.4 First version of the plugin preferences panel

This is a simple class containing declarations of several properties (panel controls) and the ***draw()*** method, which displays them on the screen. At this moment these controls are placed in a column (like menu items).

Instead a list of static items, I placed an expression in the ***key*** enumeration property. It generates the item definition tuples for the 26 'A'…'Z' letters (ASCII codes from 65 to 91). I did it just because I did not want to type all these 26 tuples. Note also that I placed the add-on file name (without the *".py"* extension) as the ***bl_idname*** value. (I used the Python global ***__name__*** variable for this purpose).

This new API class needs to be registered. This is the third class that we have to handle in this way. To minimize the chance for stupid mistakes (as registering the class in the ***register()*** method and forgetting unregister it in ***unregister()***), I introduced an auxiliary, global list of the API classes handled by this module (Figure 4.5.5):

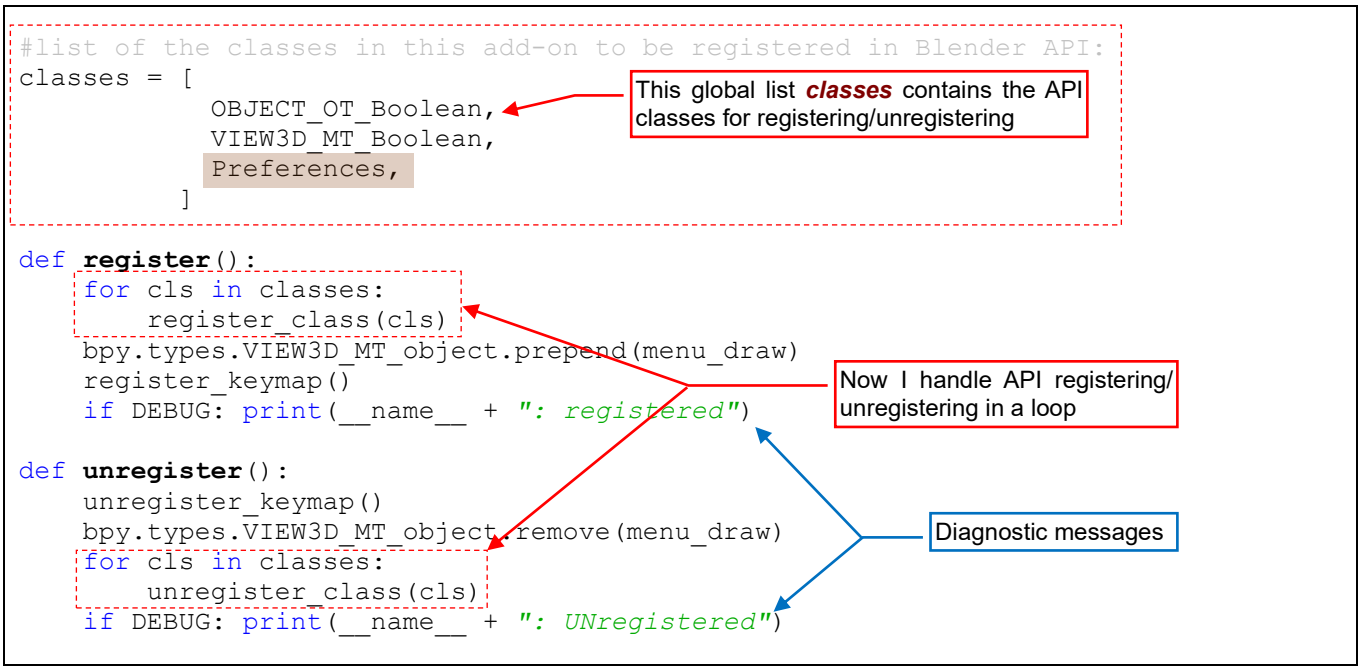


Figure 4.5.5 Modified registration procedures

Instead of single calls to the ***register_class()*** method, now I call this method in a loop for all the items from list ***classes***. Additionally, at the end of the ***register***/***unregister*** procedures I placed auxiliary diagnostic messages. (They are still useful, accompanying the debugger data). To turn them off, set the ***DEBUG*** variable to 0.

When you enable our add-on – Blender will load this code and display its preferences panel (Figure 4.5.6):

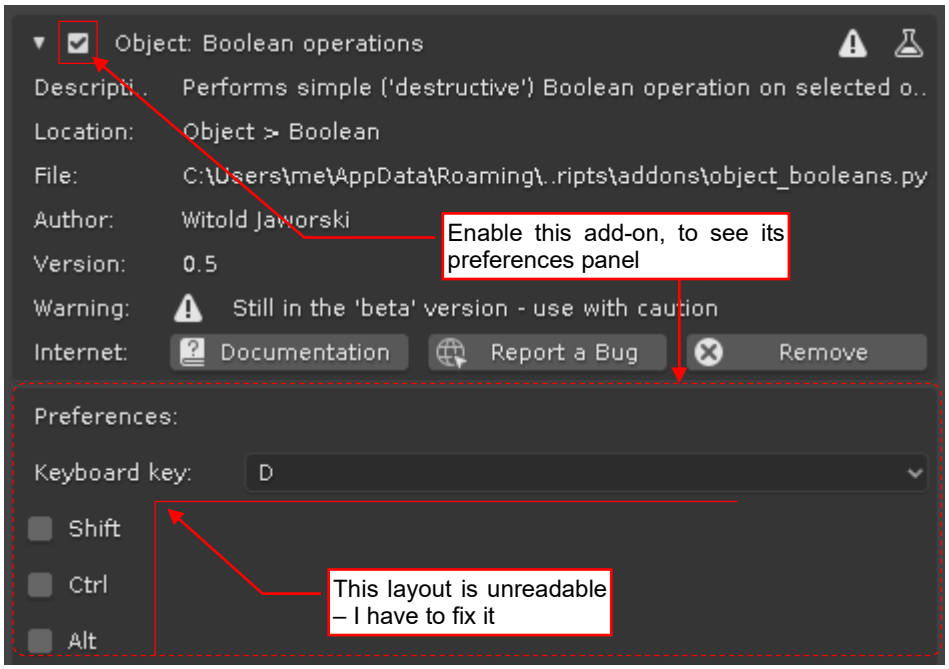


Figure 4.5.6 Initial look of the preferences panel

The preferences panel displayed properly, but its layout is unreadable (it is difficult to recognize that it describes a keyboard shortcut). Figure 4.5.7 shows the fixed ***draw()*** function:

```
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")
```

Set the controls horizontally (in a row)

Align them to the left

A spacer in between

Different label for the dropdown menu

Another spacer (thinner than the first one)

Additional label

Figure 4.5.7 Fixed *draw()* function

Of course, I gradually transformed the code from Figure 4.5.4 to the state depicted above, testing after each step the updated preferences panel in the *Blender Preferences* window. (After every modification I saved this add-on file and turned the add-on off and on). You can see the ultimate result in Figure 4.5.8:

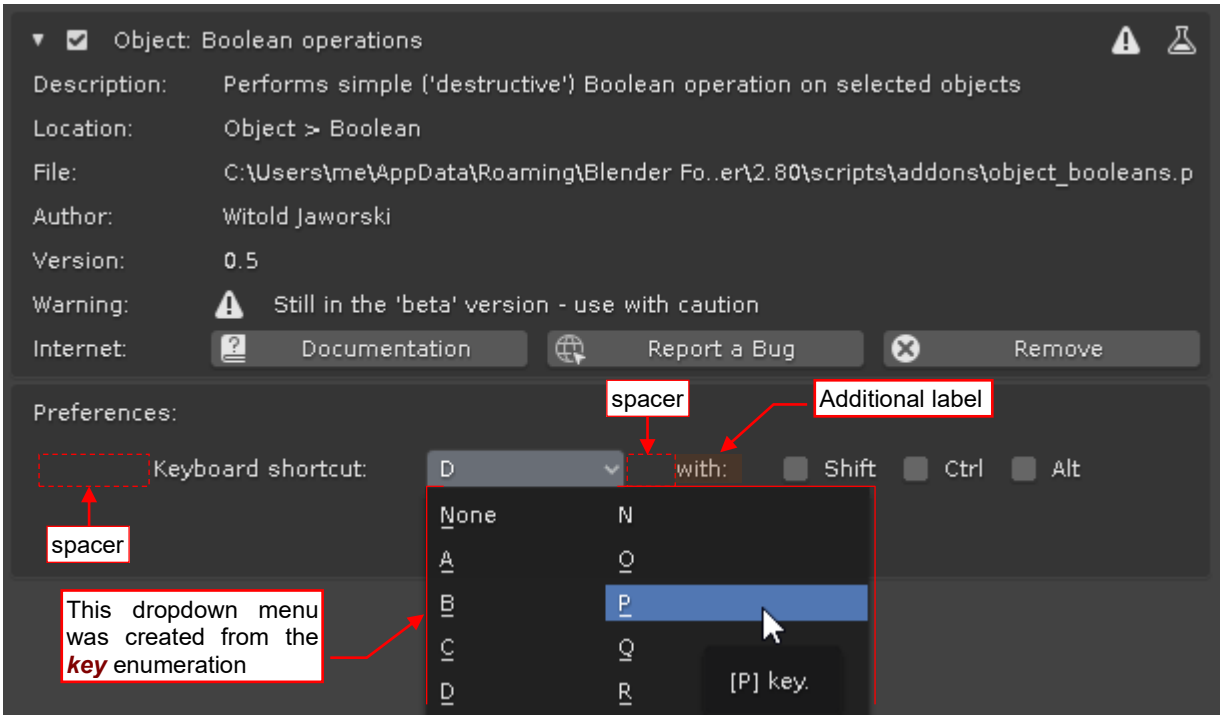


Figure 4.5.8 Fixed preferences panel

The first value on the key dropdown list – *None* – turns off this keyboard shortcut (and in the effect – the pie menu of the *Boolean* command). I added this item just in case, if the user decides that she/he does not need any shortcut for this command.

Now make another test: select from this list a different key – let's say *E* – and close Blender leaving this add-on enabled (active). Blender 2.8 automatically saves all the add-ons preferences, so this setting also is saved. Then open Blender again. Note that the state of this plugin is restored in this new Blender session: the *Boolean operations* add-on is enabled, and the *E* key is already set in its preferences panel.

However, if you disable this add-on and quit Blender, then open Blender again and enable the add-on, it will display the default *D* shortcut.

- The add-on preferences are saved between Blender sessions as long as this plugin is enabled. Blender removes add-on settings when it is disabled by the user (in the *Blender Preferences: Add-ons*).

OK, we have checked that the add-on settings are preserved between Blender sessions. However, at this moment these add-on preferences are not connected to the rest of the plugin code. In particular, the shortcut keys selected in the preferences panel do not open the *Boolean* pie menu. It's time to implement this connection.

When I write a program, I always try to follow the "single place" rule: every operation (as keyboard shortcut registration or setting the default values) should take place in a single place of my code. It can be implemented as a function or procedure, or a constants declaration. In all other places, where I need it, I am using this function, procedure or constants. (In this way I minimize the potential risk of the errors caused by changing the code in one place while forgetting to make corresponding update in the other, where I implemented similar operation). That's why for the shortcut keys in this script I decided to set their default values in a global dictionary named ***hotkey defaults*** (if I could do it in Python, I would mark it as constant). I am going to use these values in the call to function ***keymap_items.new()*** in the ***register_keymap()*** method (see Figure 4.4.3, page 100). In the code below I am re-using ***hotkey_defaults*** items as the default values of the API properties (Figure 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}
```

This dictionary contains default values for the arguments of the ***keymap_items.new()*** function (used in ***register_keymap()***). The keys of this dictionary are the names of these ****.new()*** function arguments.

```
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"],
              )
```

I use the corresponding ***hotkey_defaults*** items as the default values of the API properties.

*the **draw()** method – without changes*

Figure 4.5.9 Declaration of the default keyboard shortcut

Note that the keys in the ***hotkey_defaults*** dictionary must match the argument names of the ***keymap_items.new()*** function. (I could skip all its optional arguments). I placed the declaration of this dictionary above the *Preferences* class declaration, because the API functions that initialize the properties of this class use ***hotkey_defaults*** dictionary entries as their default values.

- Blender executes functions ***BoolProperty()*** and ***EnumProperty()*** in the code above when it loads (activates) the add-on, before calling the ***register()*** function. That's why you can use in their arguments only the values that are already declared in the previous lines of this script.

Then I modified the ***register_keymap()*** procedure (Figure 4.5.10):

Figure 4.5.10 Using the add-on preferences in the keyboard shortcut registration

When you compare this code with the previous version of ***keymap_register()*** from Figure 4.4.8 (page 102), you can see that it is much more extended. Additional lines at the beginning prepare the arguments for the ***keymap_items.new()*** function (dictionary ***args***). Initially this is a copy of the ***hotkey_defaults*** dictionary. Then I update it with the values from the current addon settings. For this purpose, I use a local variable ***prf*** that represents the data from the preferences panel. I override the corresponding ***args*** entries with the ***prf*** properties. If the users selected the **'NONE'** value as the key (see Figure 4.5.8), I quit this procedure at this point (no shortcut will be registered). Otherwise ***keymap_register()*** registers the new shortcut, as it did before.

At the end of this method I placed an auxiliary diagnostic message. (Set the ***DEBUG*** constant to 0, to turn it off).

To check if this updated code works, enable this add-on and select key ***E*** in its preferences panel, then close Blender and open it again. Observe the diagnostic messages in the console (Figure 4.5.11):

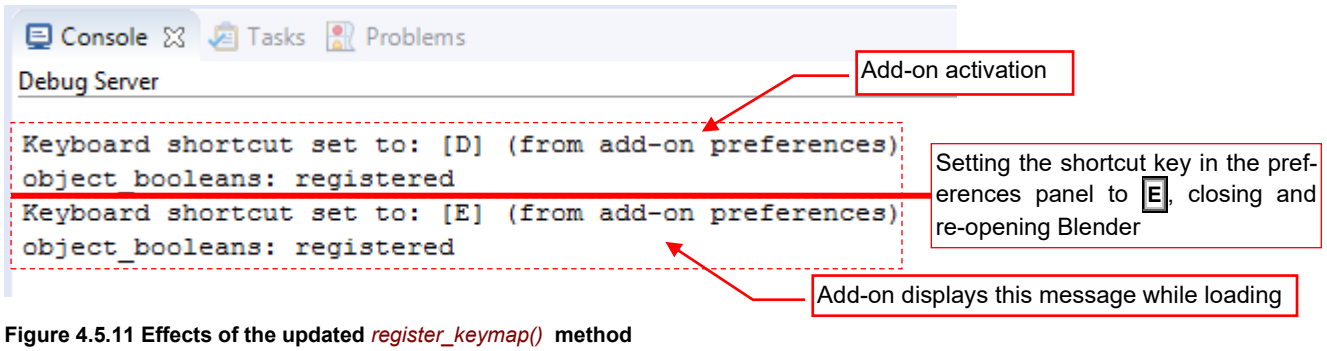


Figure 4.5.11 Effects of the updated *register_keymap()* method

The script reads its new preferences when Blender was loading, and assigns the pie menu keyboard shortcut to key E. However, the user will interpret such an "late" application of the setting changed in the previous session as an error. An she/he will be right. In Blender environment all updates you are making in the panel controls are applied immediately (there are no "OK" buttons in Blender GUI). To immediately apply the changes in the *Preferences* panel I added to the ***Preferences*** class a special method ***on_update()***. Then I passed this procedure to the functions that initialize this class properties (in their optional ***update*** arguments). Now Blender will call ***on_update()*** when the user changes (via panel controls) these API properties (Figure 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
              )

        the draw() method – without changes
```

Auxiliary function, invoked when the property value has been updated

Figure 4.5.12 Handling update notifications from the API class properties

To update the Blender key map, ***on_update()*** removes eventual previous shortcut (in ***unregister_keymp()***), then it registers the new one (in ***register_keymap()***). As you can see in Figure 4.5.10, ***register_keymap()*** applies the current *Preferences* settings, which means the current (updated) property value. Function ***on_update()*** must be defined in the script lines that precede the property initialization functions (i.e. before the corresponding calls to ***BoolProperty()***, ***Enum Property()*** functions – just like the ***hotkey_defaults*** dictionary).

When you reload this add-on, every change in its properties panel will be immediately passed to the current Blender keymap (Figure 4.5.13):

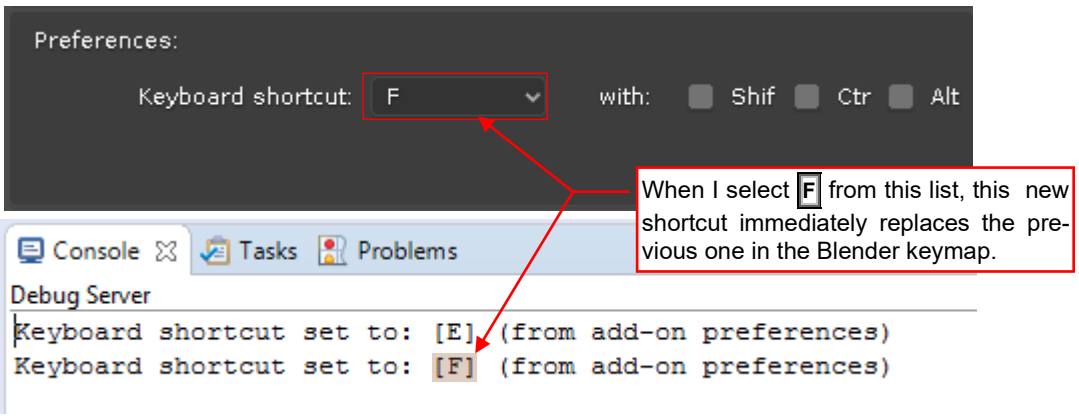


Figure 4.5.13 Immediate changes of the pie menu shortcut key

You still have to publish a description of this add-on in the *wiki.blender.org*, and to open a bug tracker for the eventual error notifications[1]. However, these additional activities are not the subject of this book. The full code of the script, we have written here, you will find on page 162.

When you finish this add-on, remember to switch its ***DEBUG*** constant to 0 (see Figure 4.5.3, page 107). Otherwise it will call the PyDev Debugger client, which will cause a runtime error when the user activates this script on another computer. Leaving these lines active even on your own computer can disturb eventual tests of another add-on (if this *Boolean operations* add-on is also enabled).

Summary

- To test the implementation of the add-on preferences panel, you have to install your plugin in Blender (in the *Blender Preferences* window – see page 106);
- You can still edit and debug such an installed script file. Just add to your PyDev project its link (page 106), and call the PyDev Debugger client at the beginning of its code (page 107);
- To implement a preferences panel, define a new API class that extends the ***bpy.types.AddonPreferences*** base. Set the identifier (***bl_idname***) of this class to your script name. To display the preferences panel, override its ***draw()*** method (page 107);
- As the other API classes, register your add-on preferences class in the ***register()*** method, and unregister it in the ***unregister()*** method (page 108);
- You can load the current add-on settings (as displayed in its preferences panel) from a collection exposed by the current context object: ***bpy.context.preferences.addons***. Use the name of your script (without the *.py* extension) as the key of this dictionary (page 111);
- To immediately update Blender settings when the user has changed one of the add-on preference pane controls, implement a call-back function and assign it to the ***update*** notifications of the API properties (as in page 112);

[1] In the result of such user feedback, I added further modifications to the script published in the previous edition of this book. Such updates are natural part of the add-on lifecycle.

Appendices

I have added to this book various optional materials. They can come in handy when you are not sure of something while reading the main text.

Chapter 5. Installation Details

In this chapter, you will find details of the Python, Eclipse and PyDev installation procedures. Study them just in the case you have stuck somewhere in the shorter descriptions that I placed in the main text.

5.1 Details of Python installation

Since 2019 Eclipse is only available in the 64-bit variant. Thus, to ensure that there will be no conflict with the Python interpreter, I suggest installing the 64-bit variant of the current Python version.

Open the Python project page: www.python.org, select there the ***Downloads*** menu, and then – the Python variant prepared for your OS (Figure 5.1.1):

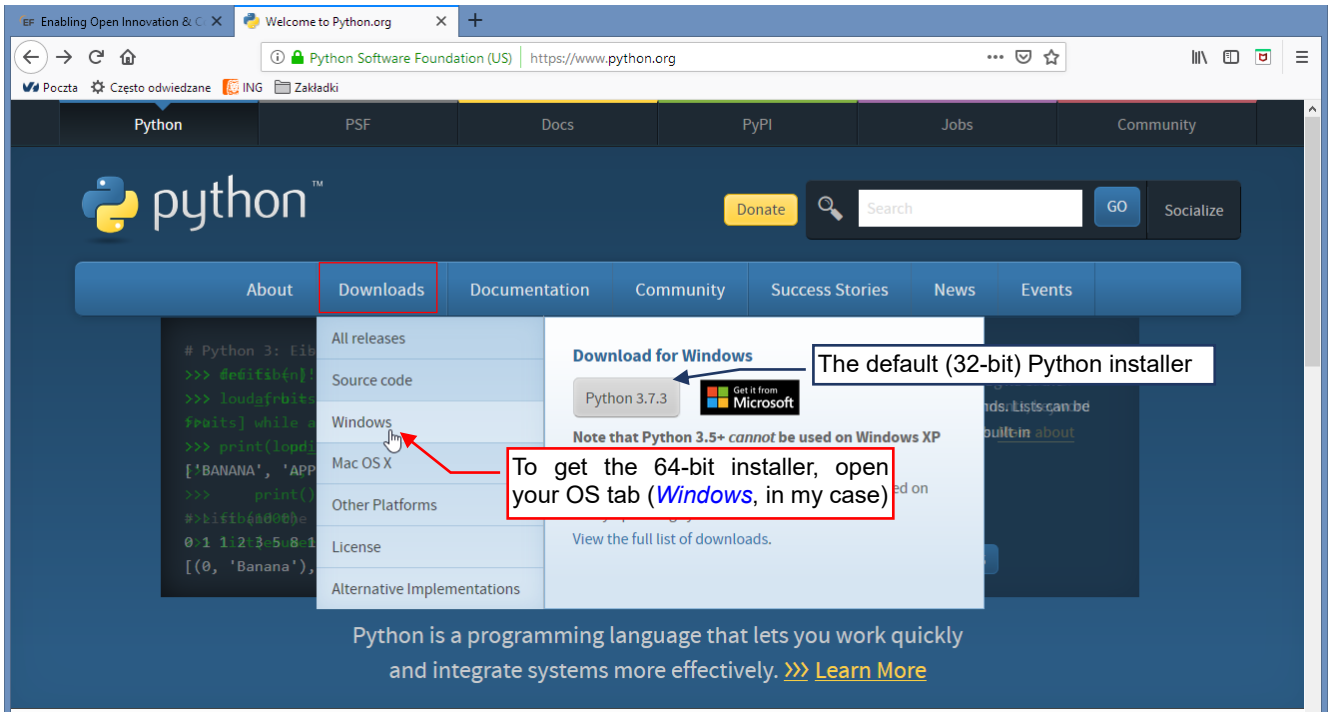


Figure 5.1.1 Main page of the Python project

The default button (labeled “Python 3.7.3” in the figure above), downloads the 32-bit Python variant, thus I had to ignore it. Instead, I clicked the OS name (***Windows***), to get the full list of the variants for this system.

From the next page I downloaded the 64-bit Python variant for Windows (Figure 5.1.2):

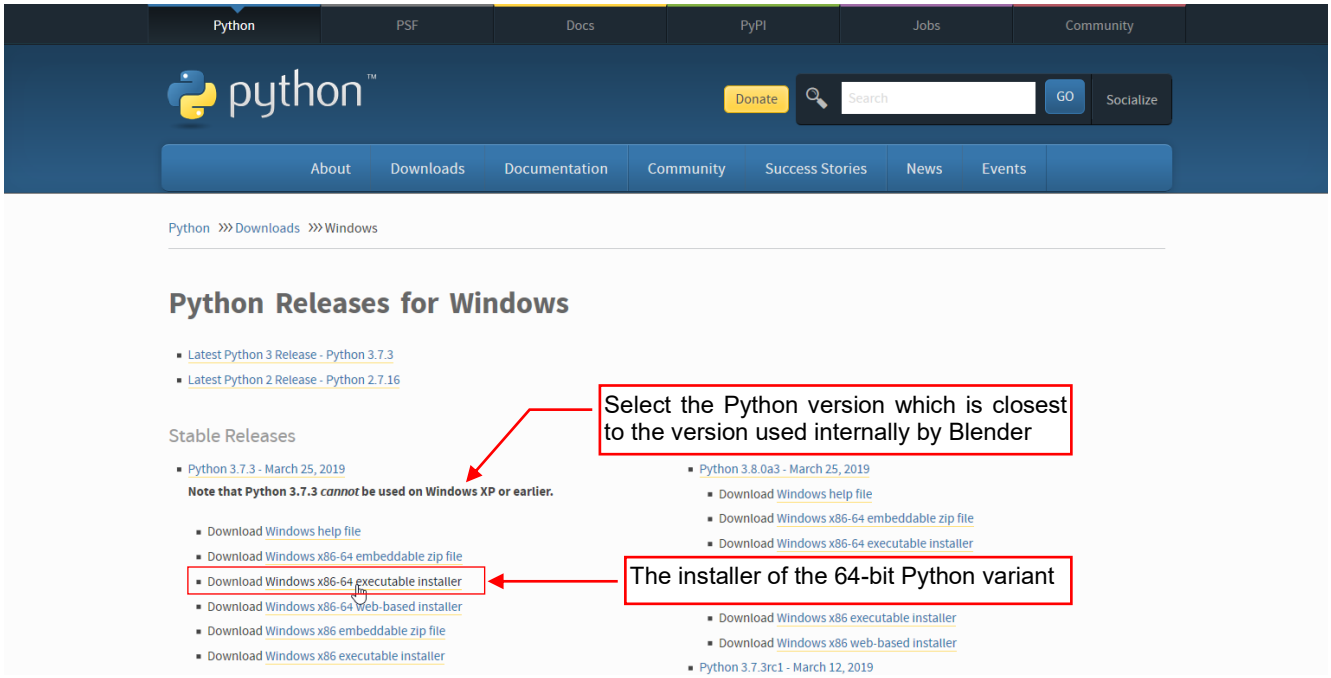


Figure 5.1.2 The download page, with various Python versions

Select the same Python version, which is used in your Blender. (If you cannot find the identical version — select the one that has the closest version number). Clicking the link to download of the Python setup program. In the case as in Figure 5.1.2 the name of this file is: *python-3.7.3-amd64.exe*. Do not worry about the "*amd*" prefix before the "*64*" (bit). Despite this name, you can also run it on the PCs equipped with the Intel processors.

When you run the downloaded setup program, you can alter the Python settings or simply install it using the default options (Figure 5.1.3):

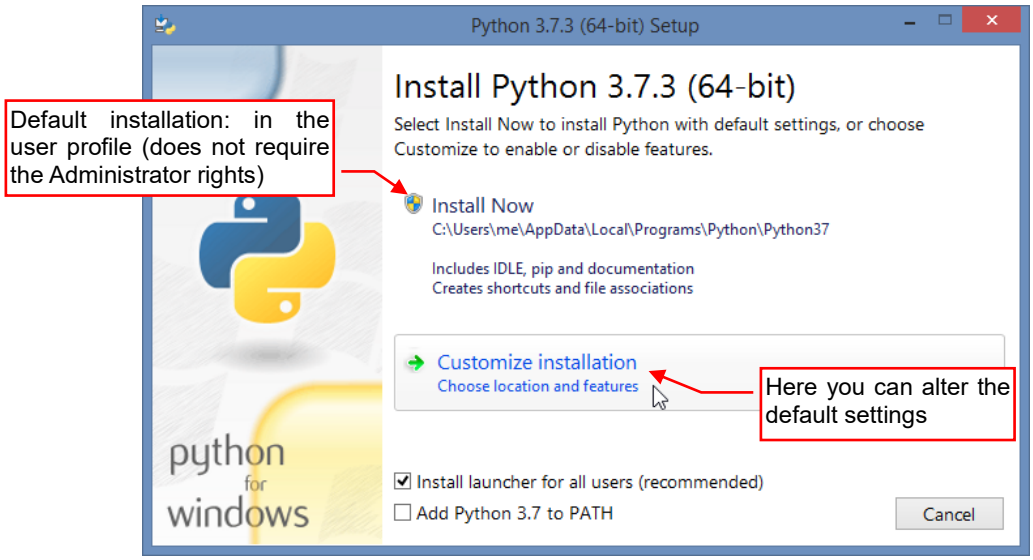


Figure 5.1.3 The first screen of the setup program

I do not like the unnecessary additions, thus I decided to alter (restrict) the scope of this installation to the minimal set. I clicked the ***Customize installation*** button, which opens the screen as in Figure 5.1.4:

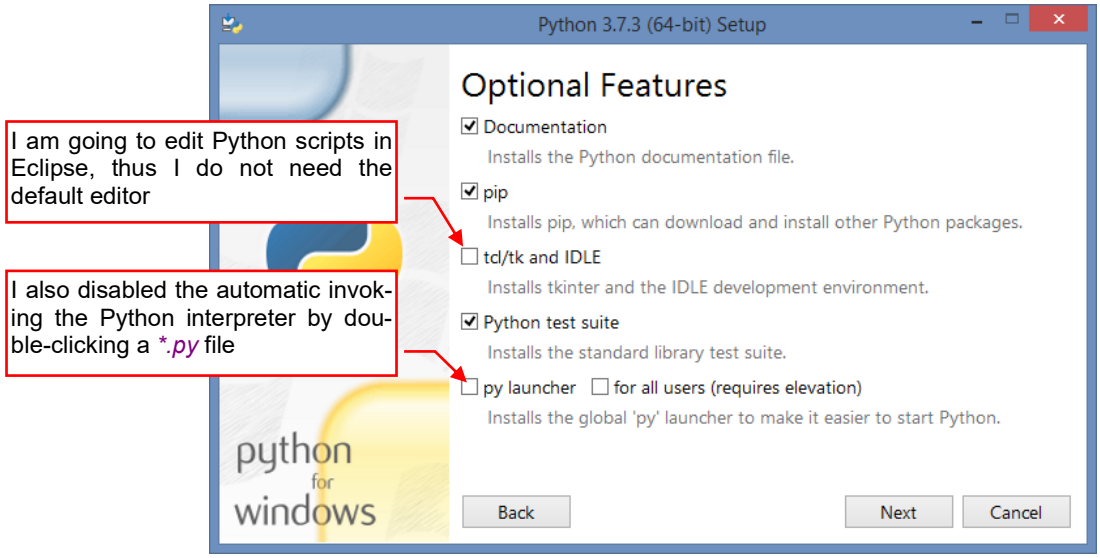


Figure 5.1.4 Python setup options (form 1/2)

I do not like the default Python editor, thus I switched off the *td/tk and IDLE* option. I am going to use Eclipse for the more difficult/advanced projects. For a quick look into the contents of any Python file I am going to use the popular Notepad++ or similar program that displays the code in the syntax-dependent colors.

I also disabled the *py launcher*, which allows to run the **.py* files as the executable programs (by double clicking the file icon, like the files with the **.exe* or **.bat* extensions). I am going to use Python scripts only in the plugins, so I do not want to run them by an accident outside this environment.

On the next screen I decided to make this Python instance available to all users of this computer (Figure 5.1.5):

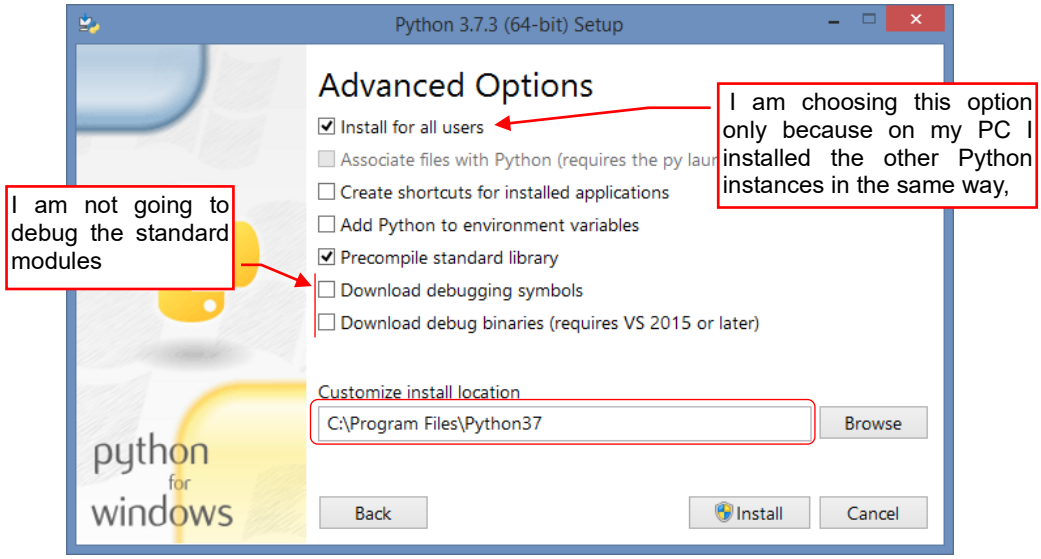


Figure 5.1.5 Python setup options (form 2/2)

On this screen I selected the ***Install for all users*** option because:

- I installed the previous Python interpreters in this way (a few years ago it was the default option);
- The setup will place the Python folder in the general *\Program Files* directory, instead of the folder in the user profile. (It is more difficult to find a program in the user profile. From time to time I have to find something among these source files);
- I have the Administrator rights, required for this option.

I also disabled the option that allows me to debug the binary files of the Python standard modules (I am never going to do that).

The Python installation starts when you click the ***Install*** button:

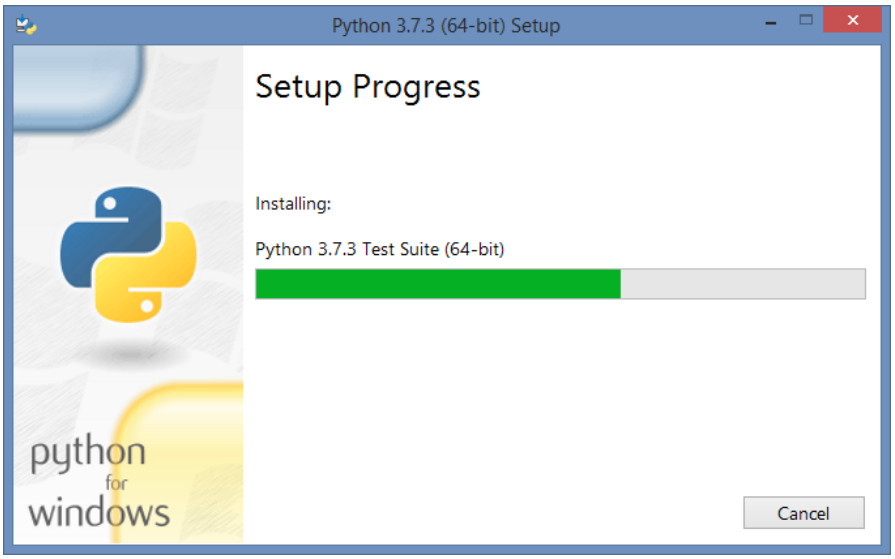


Figure 5.1.6 Python installation

The whole process takes about a minute.

Finally, the setup displays its last screen (Figure 5.1.7):

Figure 5.1.7 The final screen of the Python installation

Click ***Close*** to complete this process.

5.2 Details of Java Runtime Environment (JRE) installation

Eclipse is a Java application, and since 2019 it requires the 64-bit variant of the Java Runtime Environment (JRE). You can download the JRE setup program from www.java.com. On this portal you will also find the tips about identification the JREs that you already have on your computer (Figure 5.2.1):

Figure 5.2.1 Main page of the java.com site (as in May 2019)

The default JRE is 32-bit. If you find that you do not have the 64-bit JRE – click the ***Java Download*** button:

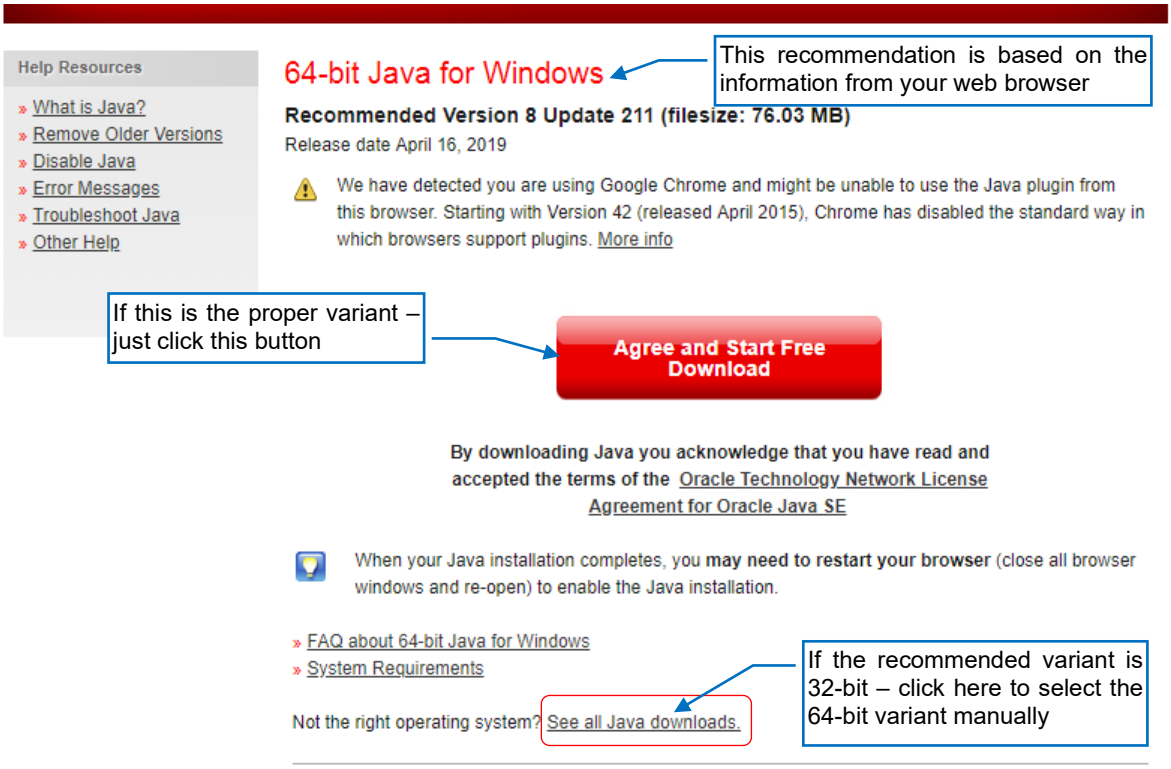


Figure 5.2.2 Auto-detecting the JRE variant for your computer

The *java.com* portal detects your OS variant using the information provided by your web browser. If it proposes the 32-bit Java – click the ***See all Java downloads*** link at the bottom of this screen.

It opens the page with all JRE variants. Select the 64-bit JRE from there (Figure 5.2.3):

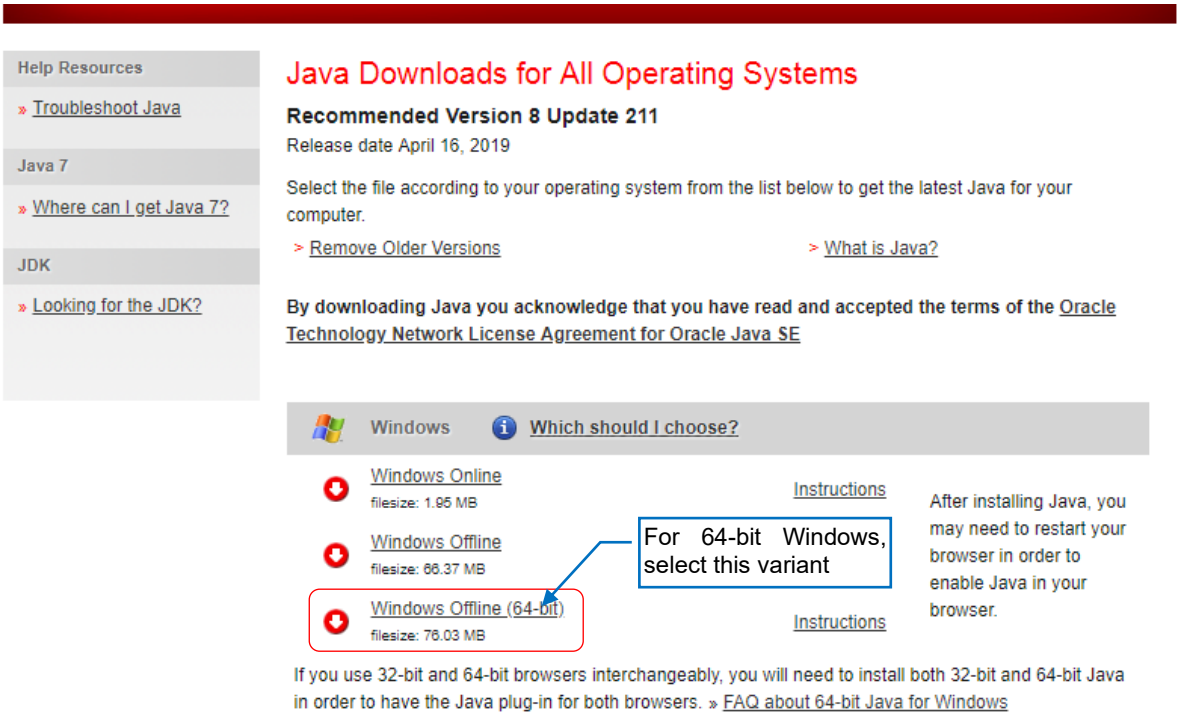


Figure 5.2.3 Manual selection of the JRE variant

Download and run the setup program. During the JRE installation you do not have to alter any options.

- Eclipse IDE requires the 64-bit JRE. It often happens that even on the 64-bit computers various applications install the 32-bit JRE. Thus, usually you will find that you have Java on your computer, but in the 32-bit variant, which cannot handle Eclipse executables. Fortunately, you can have 32-bit JRE and 64-bit JRE installed “side by side”, on the same PC.

- If you are using Mac OS, check the current Eclipse installation notes. In the moment when I am writing this book (May 2019), they advise to install the 64-bit variant of the complete JDK (Java Development Kit). (JDK contains the JRE). Otherwise Eclipse will display error messages.

5.3 Details of Eclipse and PyDev installations

- First, check if you have 64-bit *Java Runtime Environment* (*Java JRE*) installed on your computer. In Windows you can check it clicking the "Java" icon in the Control Panel. If it is not there — download the latest Java version in the 64-bit variant from the java.com site and install it on your machine[1].

Let's start by downloading the setup program. Go to the http://www.eclipse.org/downloads page (Figure 5.3.1):

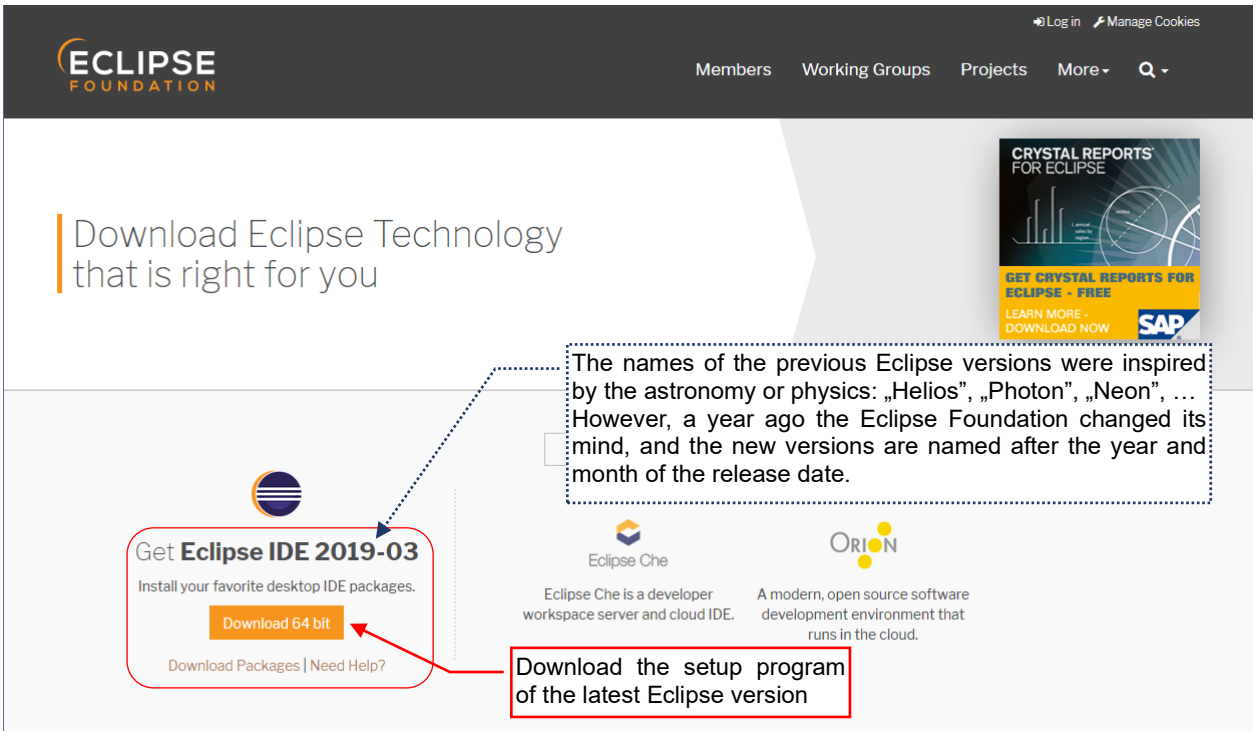


Figure 5.3.1 Selection of the Eclipse package

Click the ***Download 64 bit*** button: it opens the server selection page (Figure 5.3.2):

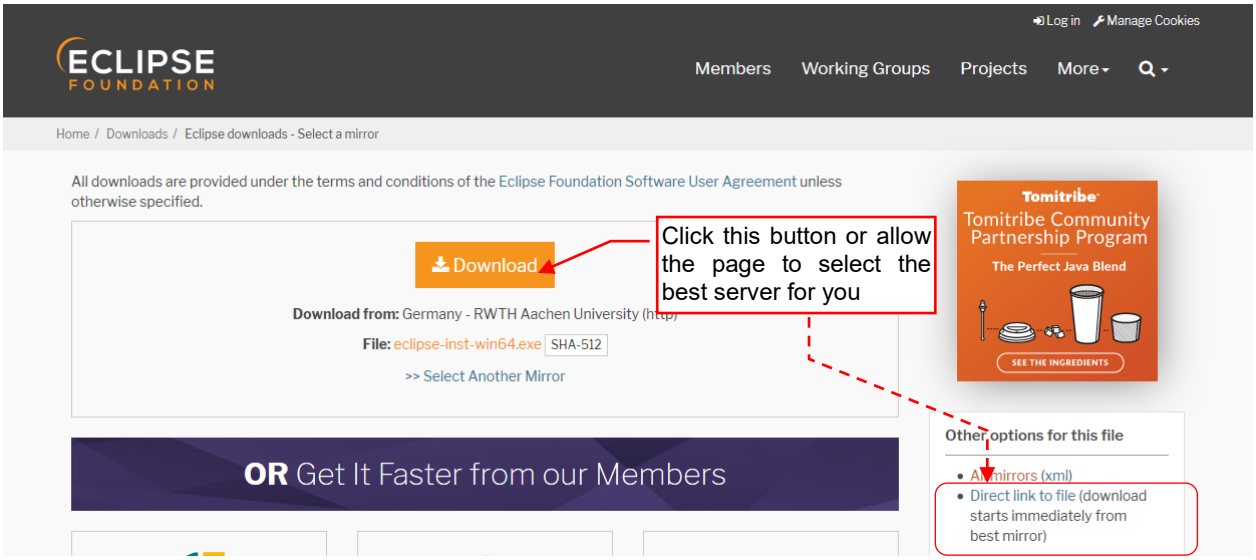


Figure 5.3.2 Select the server for download

[1] Some Linux distributions, like popular Ubuntu, have GCJ as their default Java virtual machine (VM). In this environment, Eclipse runs much slower than on the JVM from the www.java.com. What 's more, even after the JVM installation on Ubuntu, it is not set as the default VM! You have to correct it manually. More about this — see https://help.ubuntu.com/community/EclipseIDE.

When you run the setup program, you will see the window where you can select one of the Eclipse packages (Figure 5.3.3):

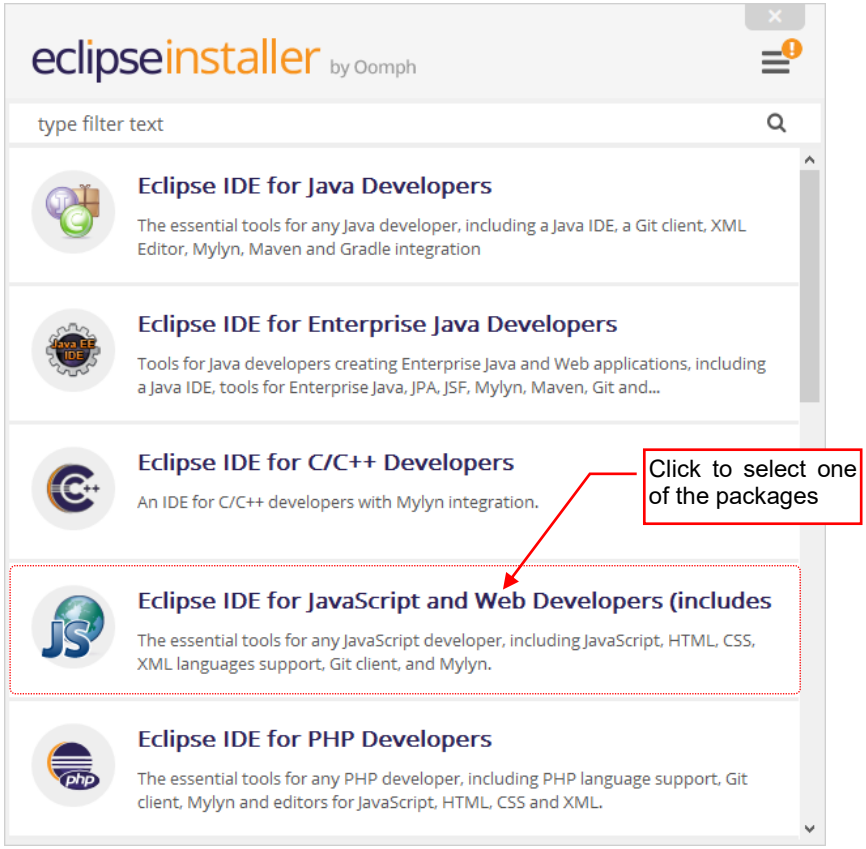


Figure 5.3.3 Selection of one of the Eclipse variants (packages)

In fact, Eclipse is a kind of open IDE framework, which can be adapted by appropriate plugins for any programming language. On the Eclipse Internet site, you can find some ready-to-use plugin packages for the most popular languages. There is no "Eclipse for Python" bundle among them, so we will make it ourselves. Just download any of these packages. For example, you can choose the “nearly empty” ***Eclipse for Testers*** (you can find it lower on the list displayed in Figure 5.3.3). Personally, I selected ***Eclipse IDE for JavaScript***, because it contains some additional tools that I am going to use for other purposes (not connected with Blender).

When you click the selected package, in the next window you can alter the default folder for the program files:

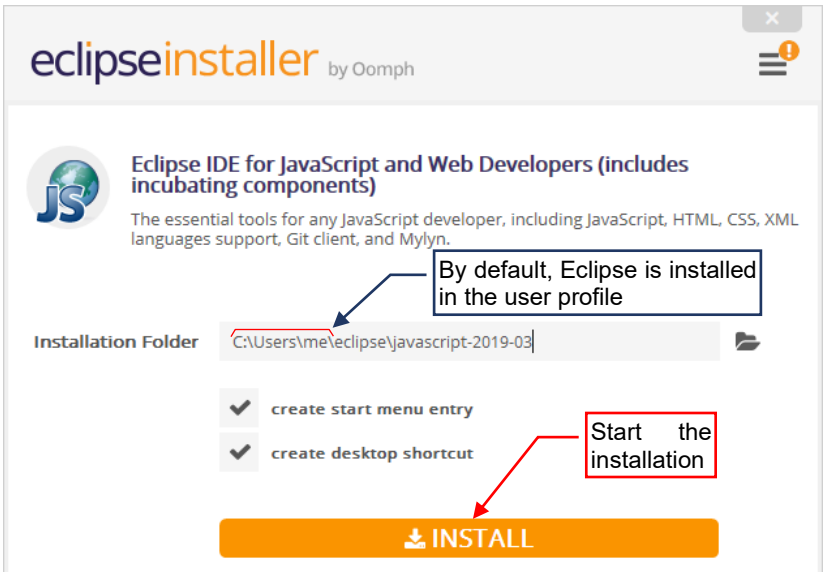


Figure 5.3.4 Eclipse installation options

I am leaving it in the default folder here, so that most probably it will match your installation.

- By default, Eclipse creates its folder in the current user directory. I decided to continue using these settings, because in the course of this book we will have to identify a certain folder among the Eclipse plugins. If I installed this IDE in a non-standard folder, some of the Readers would get lost at that point.

When you click the ***INSTALL*** button, you will have to do some “legal paperwork”, accepting various agreements:

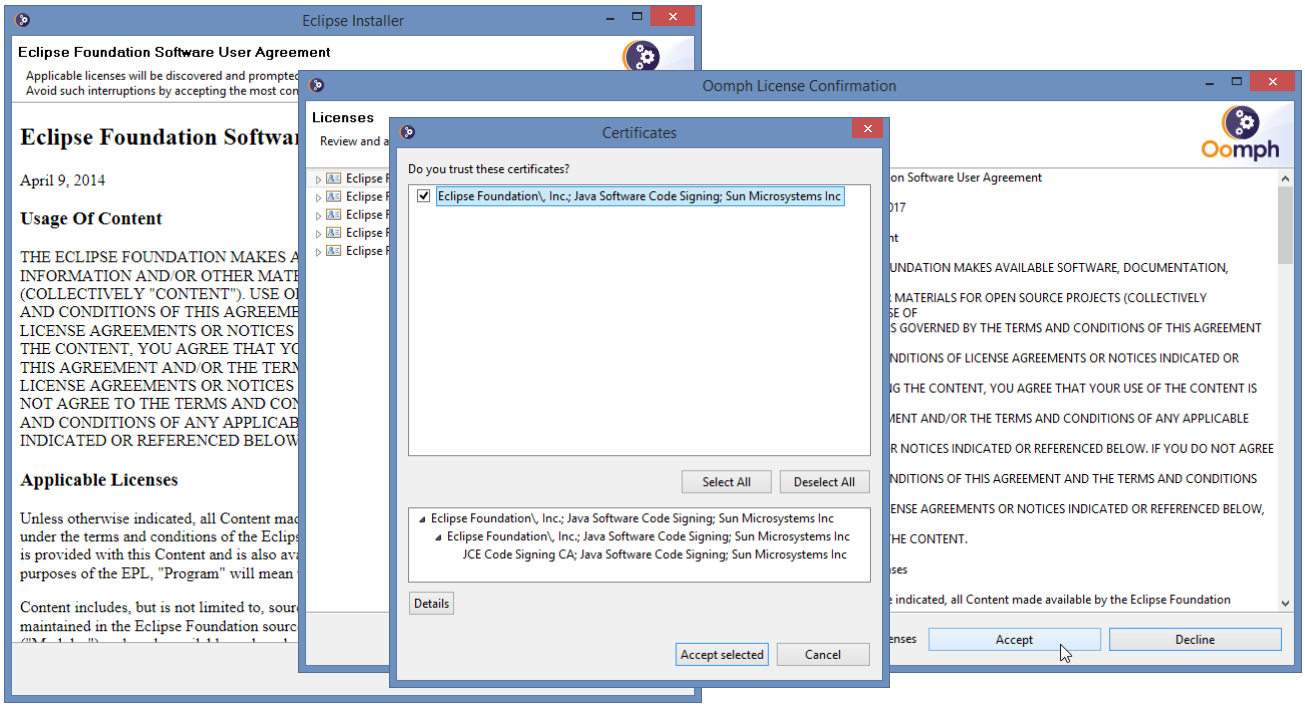


Figure 5.3.5 License agreements and certificates, accepted during Eclipse installation

Finally, you will get to the last screen of the setup program:

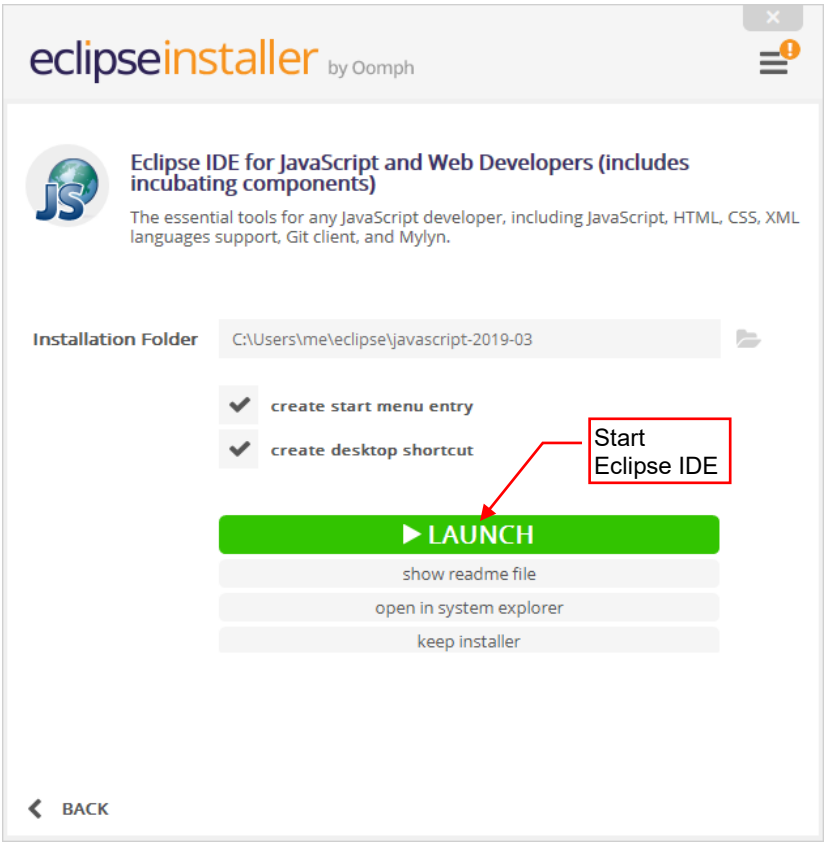


Figure 5.3.6 The final screen of the setup program

When you launch Eclipse, it always displays a dialog box where you can select the location of its projects directory (it is called “workspace”). You may just confirm this default (Figure 5.3.7):

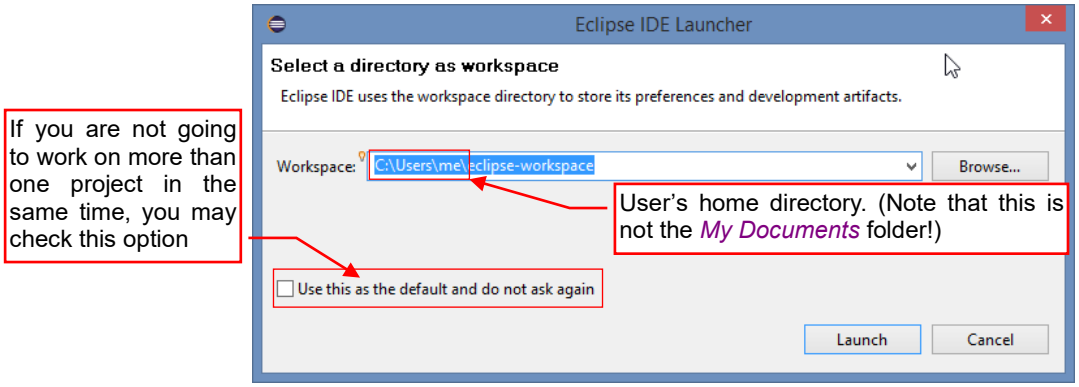


Figure 5.3.7 Selecting the current workspace

Each of Eclipse projects is a separate folder that contains your Python script(s) and a few configuration files. (If your script is located elsewhere on the disk, you can put just its shortcut in the project directory). Notice that the default Eclipse workspace folder is in the root directory of the user profile. (In this example, the username is *me*). This is not *My Documents* folder, but its parent. (It is the *Unix/Linux* convention of the home directory). If you want to keep all your data in *My Documents* — change accordingly the path displayed in this window. Eclipse will create this directory, if it does not exist.

- Eclipse is always proposing to open the most recently used project from the last workspace.

On the first launch, Eclipse opens the “Welcome” window (Figure 5.3.8)

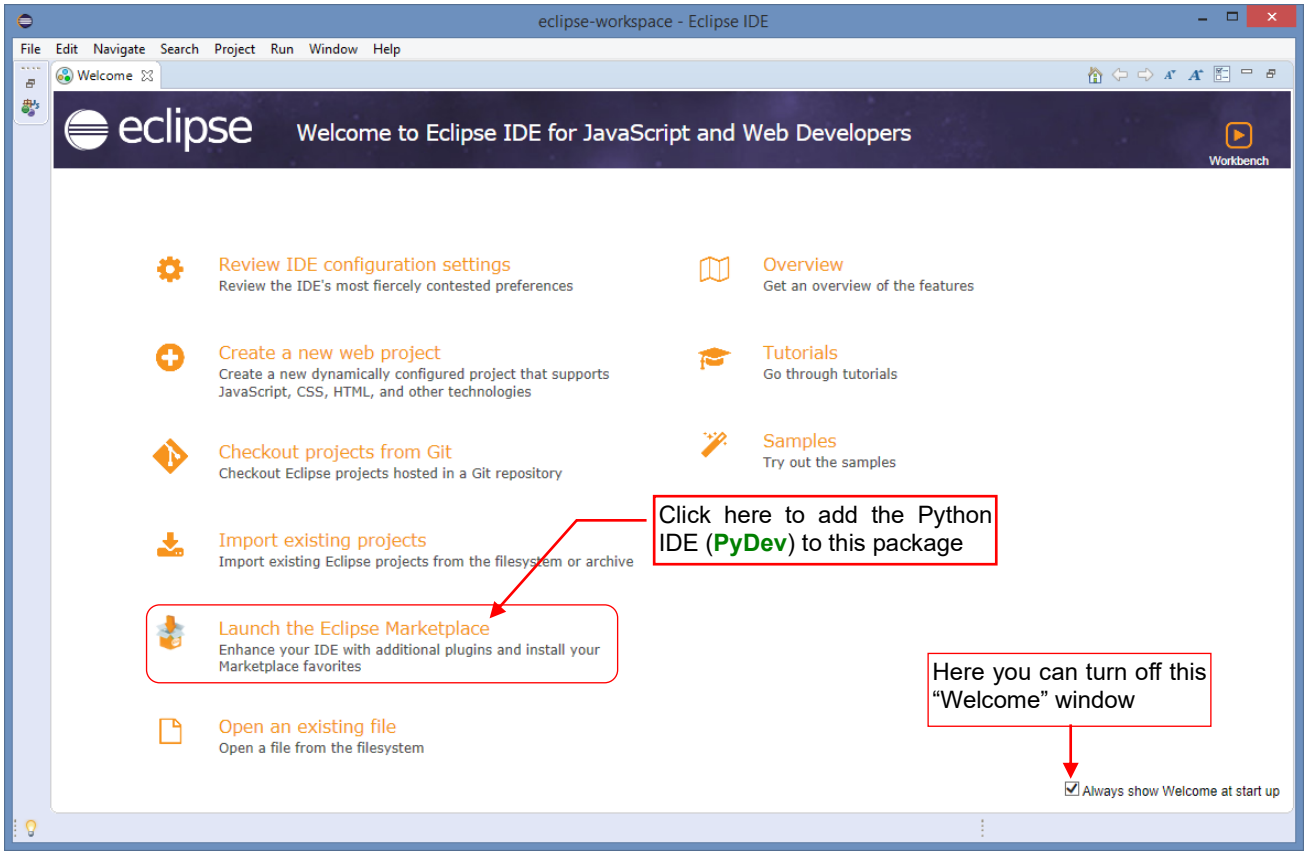


Figure 5.3.8 Eclipse window on the first launch

Disable on this screen the ***Always show Welcome*** option, and then click the item named ***Launch the Eclipse Marketplace***. It starts the Python IDE installation (an Eclipse plugin, named **PyDev**).

- You can also find the same command under the ***Eclipse Marketplace…*** label in the ***Help*** menu.

It opens the form that lists all the Eclipse plugins (Figure 5.3.9):

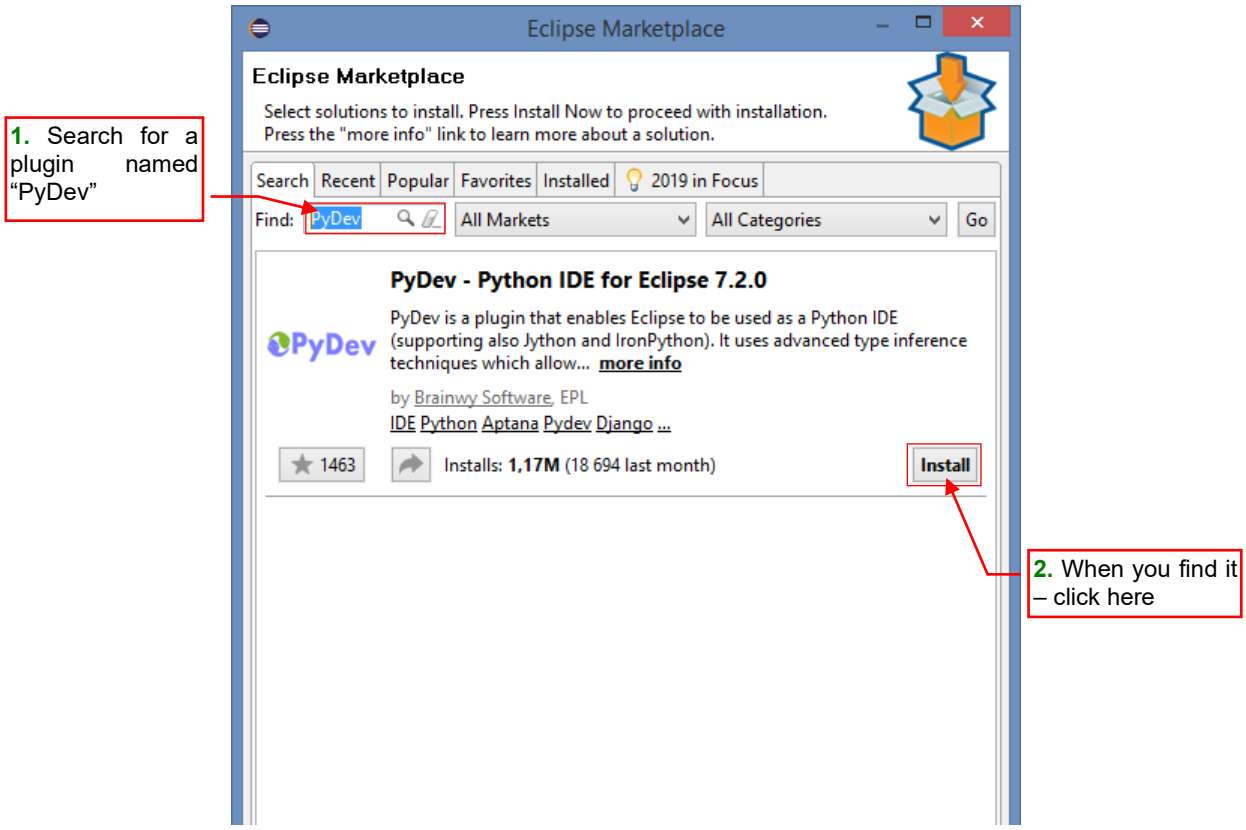


Figure 5.3.9 Plugin selection window

In the ***Find*** field type "PyDev" and run the search. In response Eclipse will find the plugin as in Figure 5.3.9. Click its ***Install*** button. It displays another window where you have to confirm the plugin components (Figure 5.3.10):

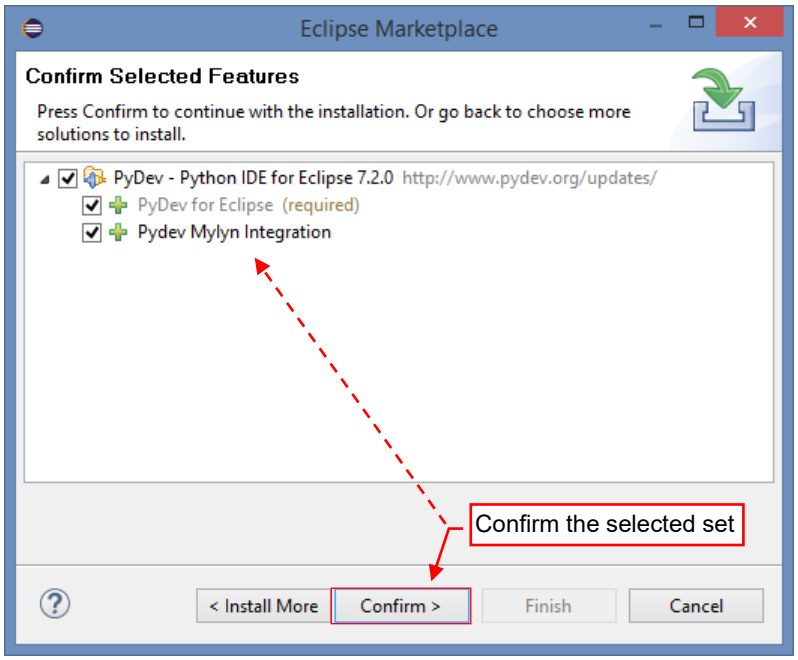


Figure 5.3.10 Confirmation of the PyDev components

I changed nothing here, just clicked the ***Confirm*** button.

In response Eclipse opens another window, with the license agreements (Figure 5.3.11):

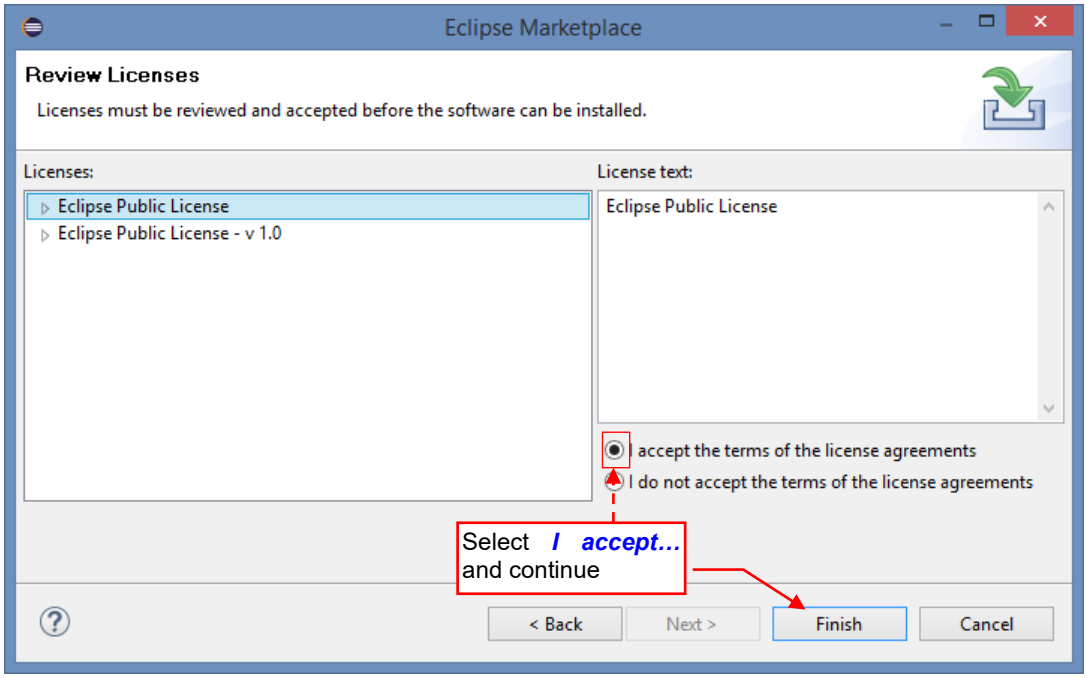


Figure 5.3.11 Confirmation of license agreements

Accept these agreements (by selecting ***I accept ...*** option) and click the ***Finish*** button. It starts the PyDev installation. During this process Eclipse downloads various components from the Internet, so make sure that you have a live connection to the web.

The progress is displayed in the Eclipse status bar (Figure 5.3.12):

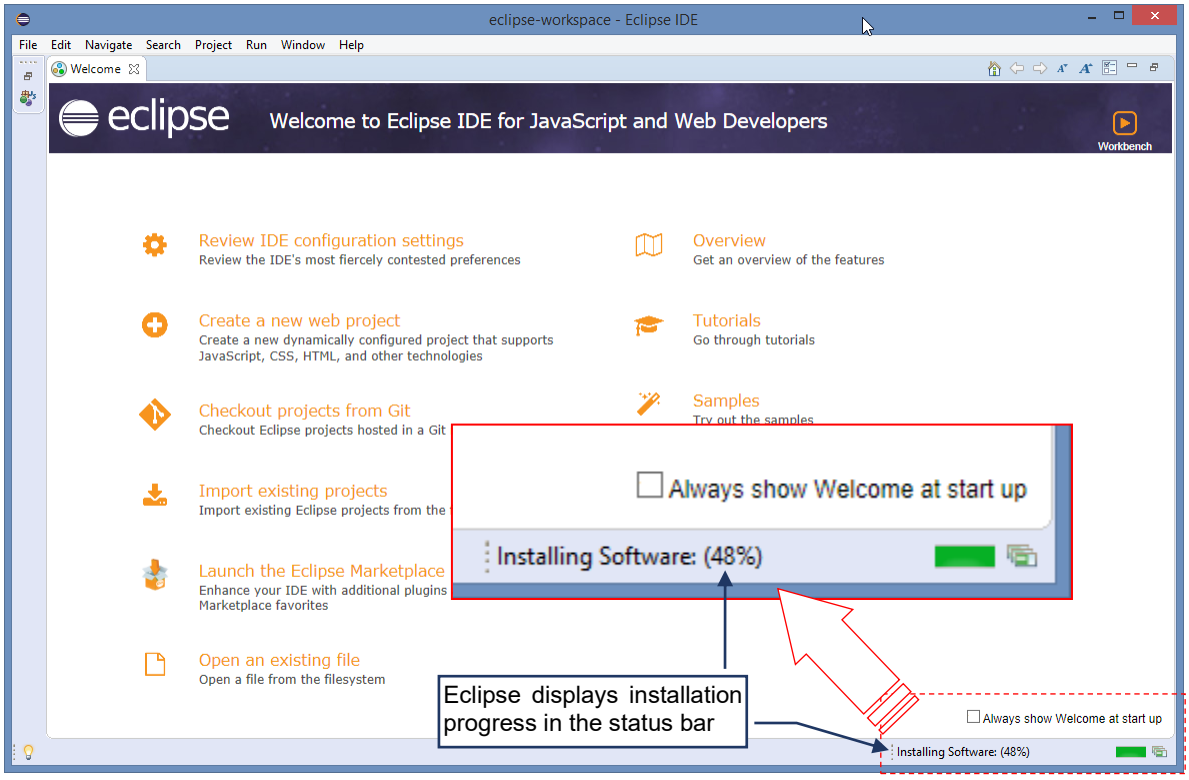


Figure 5.3.12 Installation progress indicator

When the PyDev installation is completed, Eclipse proposes a restart (Figure 5.3.13):

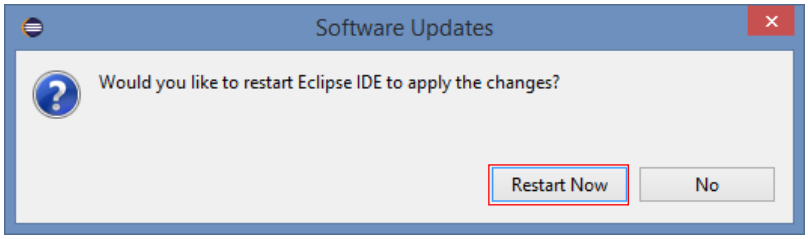


Figure 5.3.13 The final window of the plugin installation

Confirm this proposal, clicking the ***Restart Now*** button.

- You can safely install different Eclipse versions “side by side”, on the same PC. (This hint can be useful in the future, when you decide to install a version of the Eclipse IDE).

5.4 Details of the PyDev configuration

Once installed, you have to configure the in PyDev the default Python interpreter. This information is stored in the current Eclipse *workspace* (ref. page 13, Figure 1.2.5). To set it, use the ***Window→Preferences*** command (Figure 5.4.1):

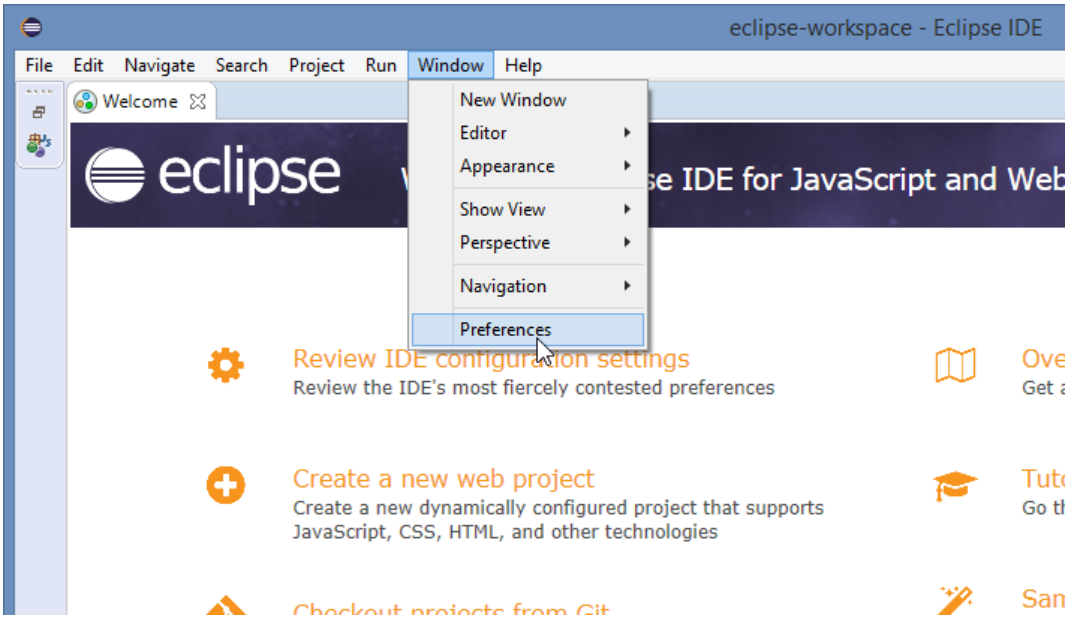


Figure 5.4.1 Opening the current *workspace* configuration

In the *Preferences* window expand the ***PyDev*** section, and in the ***Interpreter*** subsection highlight the ***Python Interpreter*** item (Figure 5.4.2):

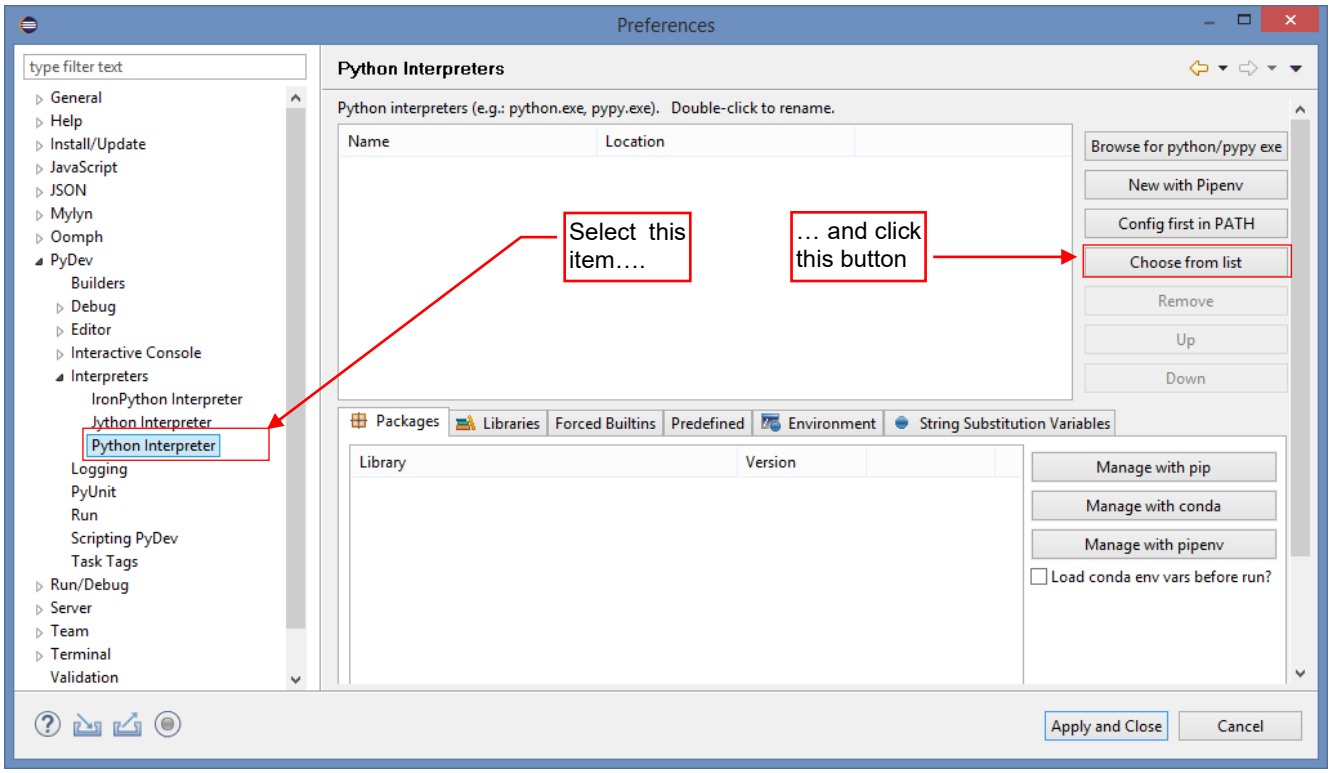


Figure 5.4.2 Automatic configuration of the Python interpreter

Now you have to select the external Python interpreter that will be used by the PyDev. To do it, click the ***Choose from list button***. It starts with searching the Python instances installed on your computer.

If the program finds more than one Python interpreter – it shows their list (Figure 5.4.3):

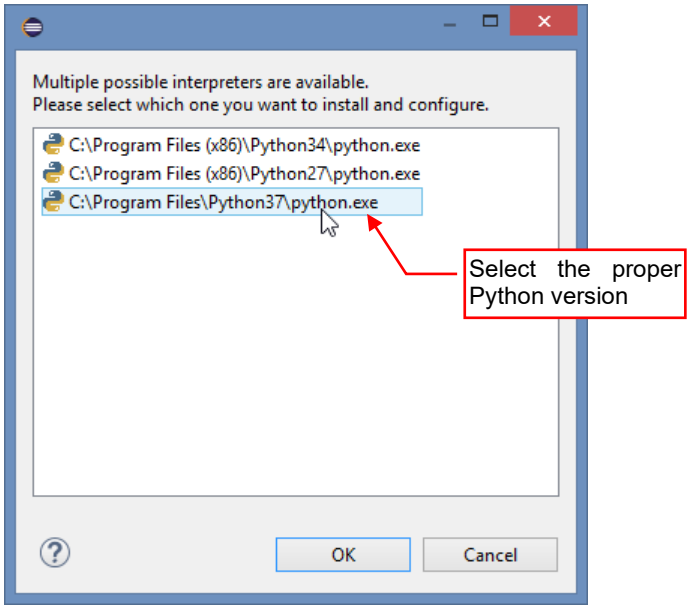


Figure 5.4.3 Selecting the 64-bit Python version from the list of the interpreters installed on the local computer

Select from this list the interpreter you just have installed (i.e. the 64-bit variant of the version that matches the Python version in your Blender – see section 1.1, page 8).

It may happen that PyDev is not able to find the proper Python interpreter. In such a case, in the *Preferences* window click the *Browse for python/pypy.exe* button (see Figure 5.4.2). It will open the window of „manual" Python selection (Figure 5.4.4):

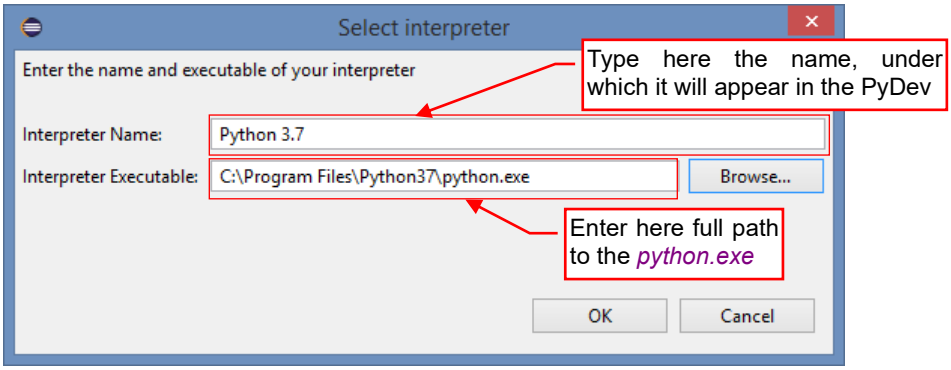


Figure 5.4.4 "Manual" selection of the Python interpreter

Type here (or select using the *Browse…* button) the full path to the *python.exe* file of the 64-bit Python instance. (Do not select the *pythonw.exe* by mistake!) In this window you can also determine the name of this interpreter in the PyDev environment. (This is just an aesthetic issue).

When you choose the Python instance, PyDev will display Python directories in a new window. They will be added to the **PYTHONPATH** configuration variable (Figure 5.4.5). Just accept it without any changes:

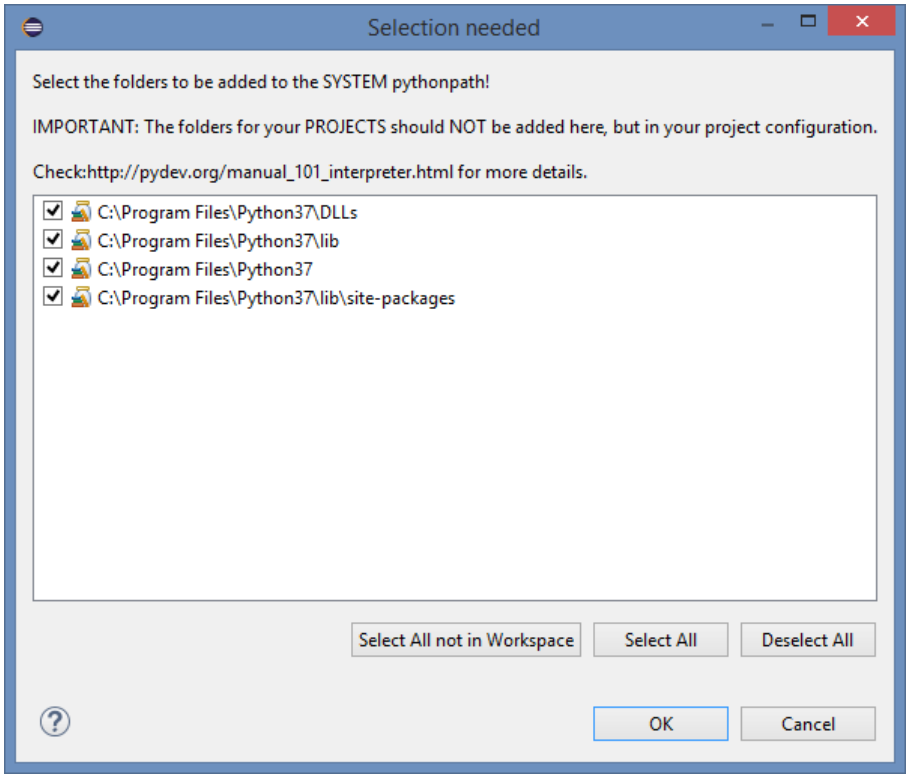


Figure 5.4.5 Selection of the directories that will be added to the PYTHONPATH system variable

In the result, the configured Python interpreter appears in the *Preferences* window (Figure 5.4.6):

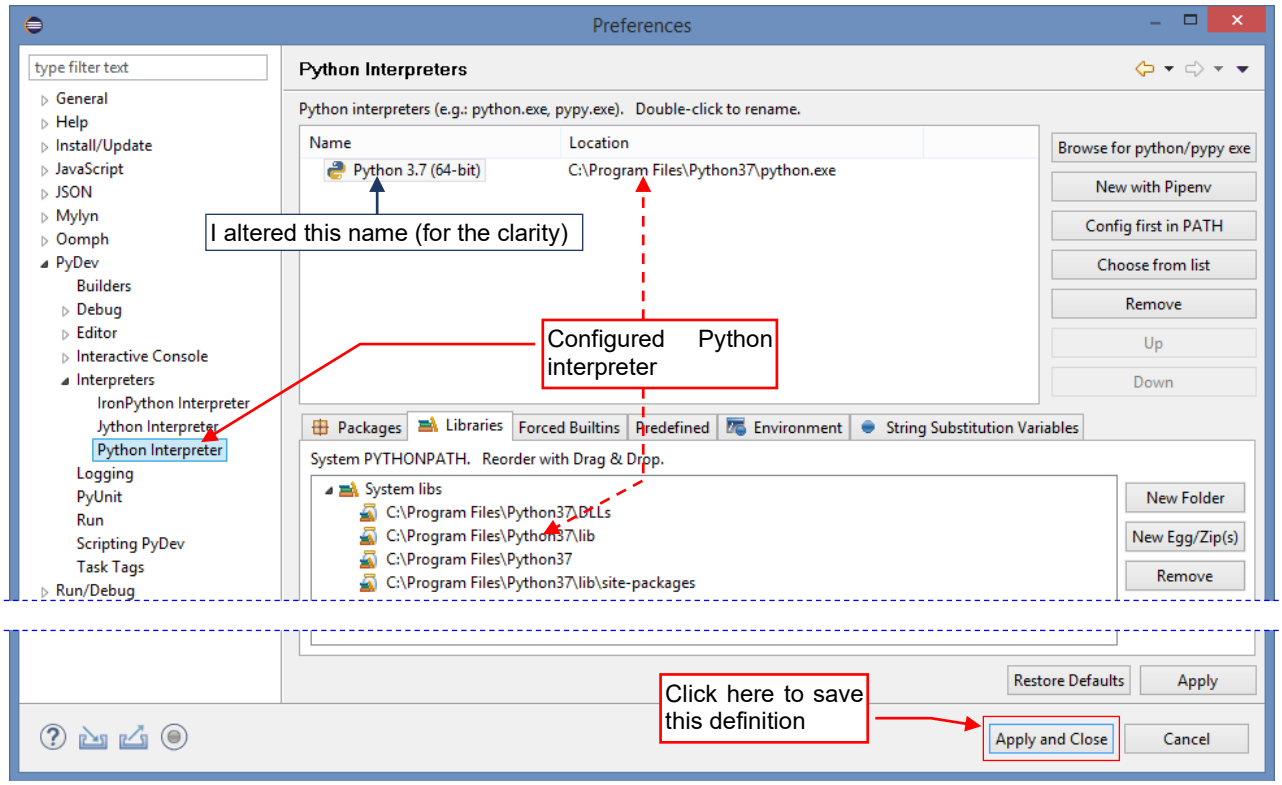


Figure 5.4.6 Configured Python interpreter

When you accept these settings by clicking the ***Apply and Close*** button, PyDev will browse all the Python files that are present in the **PYTHONPATH** directories. It will prepare the autocompletion data and the other internal stuff Figure 5.4.7):

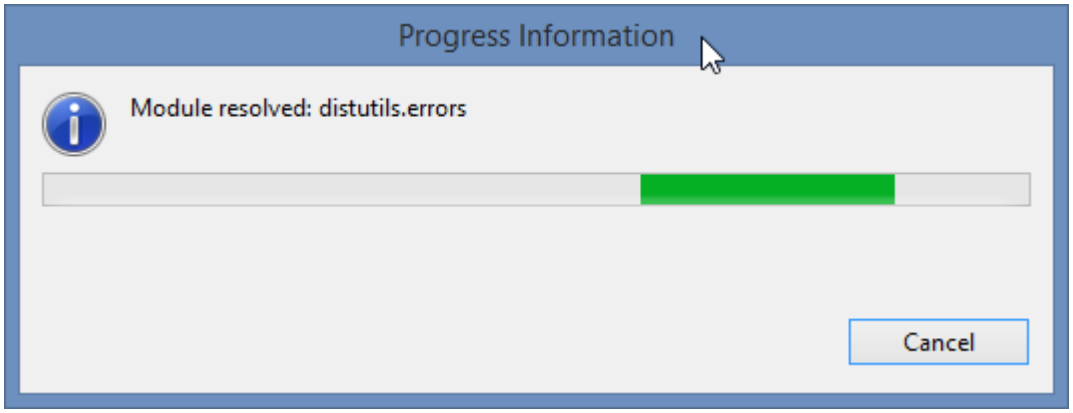


Figure 5.4.7 Processing the PYTHONPATH **files**

Now the PyDev is ready for editing the Python scripts.

- The PyDev settings described in this section are stored in the Eclipse workspace. This means that if you need it, you can prepare several workspaces, each with different PyDev settings (for example – different Python interpreter). It can be useful for testing the add-ons with older Blender versions.

- To run/debug the classic (standalone) Python scripts in PyDev, you have to assign the Python interpreter to your current project. See pages 26 and 134 for details.

5.5 Managing Eclipse project perspectives

You will use two project perspectives (alternative screen layouts) while working on a Python script in Eclipse: *Debug* and *PyDev*. Their switches are placed on the toolbar but are small and hardly visible among the other icons and controls (Figure 5.5.1):

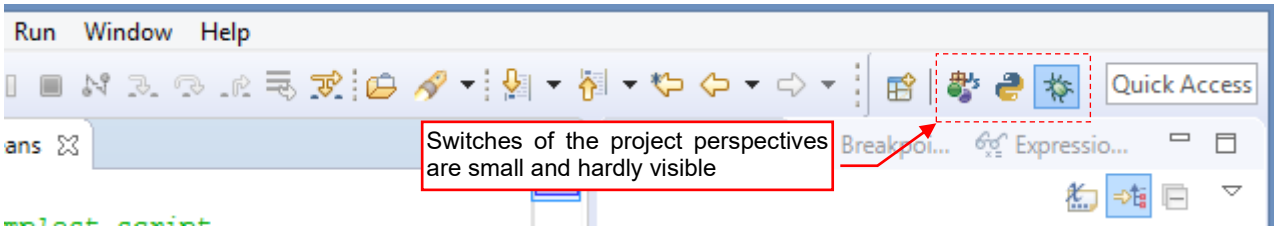


Figure 5.5.1 Project perspective buttons on the toolbar

Fortunately, you can easily enlarge them by enabling their text labels. In this mode they are more visible, which allows for quicker perspective switching (Figure 5.5.2):

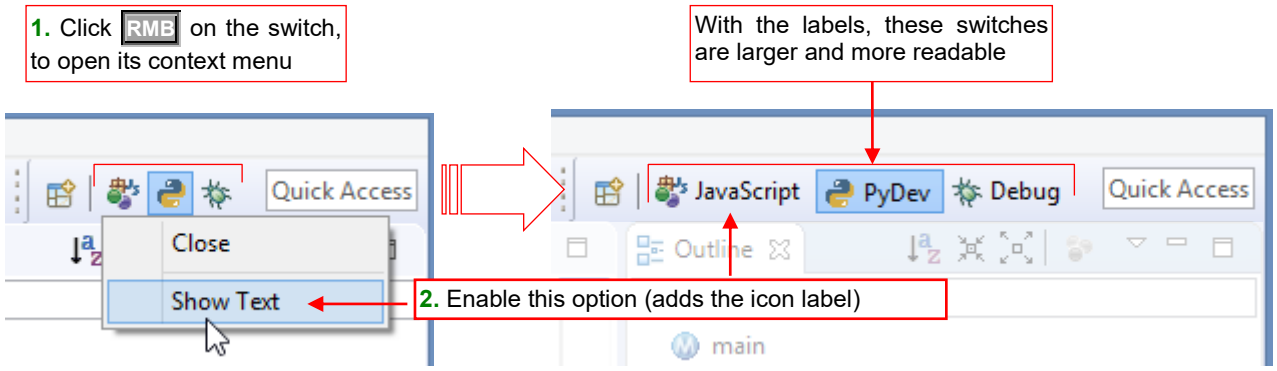


Figure 5.5.2 Enlarging the perspective switches

Now these enlarged items also allow for a quick identification of the current perspective.

If we have already started altering this toolbar, let's make another modification: removing from these switches the button of the *JavaScript* perspective. (We do not need it in our Python project) (Figure 5.5.3):

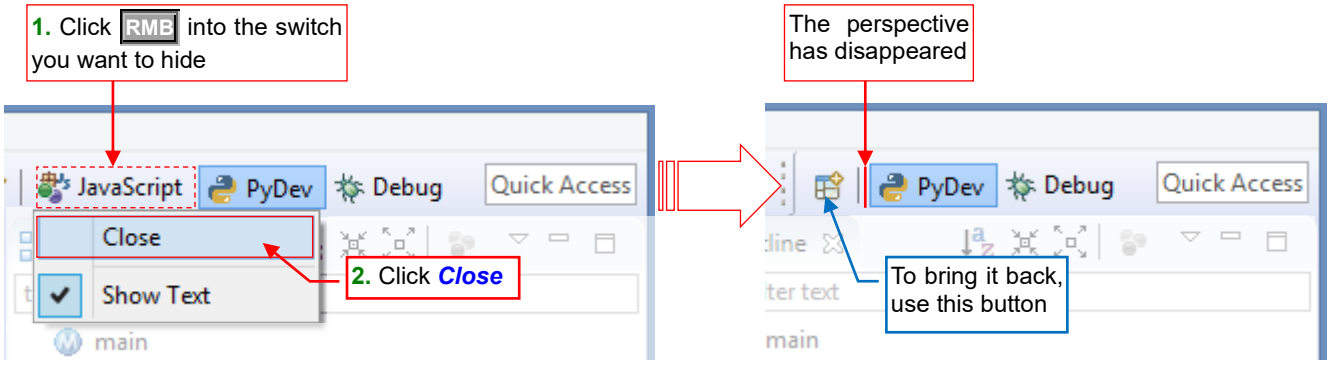


Figure 5.5.3 Removing from the toolbar the switch of an unused perspective

5.6 Configuring the running and debugging commands for standalone Python scripts

In each new PyDev project you can define how to call the Python interpreter for running / debugging the main script in your project. These settings are grouped in a so-called *Run Configuration*. To set them, you need the main source file (main Python module) in your project – even if this file is completely empty at this moment.

- *Run Configuration* settings are not used for running and debugging Blender API scripts, because for this purpose we will use the PyDev remote debugger. Run configurations are required for the classic Python modules, which are processed by the standalone Python interpreter. In this book we are using them briefly in Chapter 2, where I am showing how to run/debug the simplest code from a classic Python file.

Let's start with creating a run configuration. From the ***Run*** menu select ***Run Configurations…*** command:

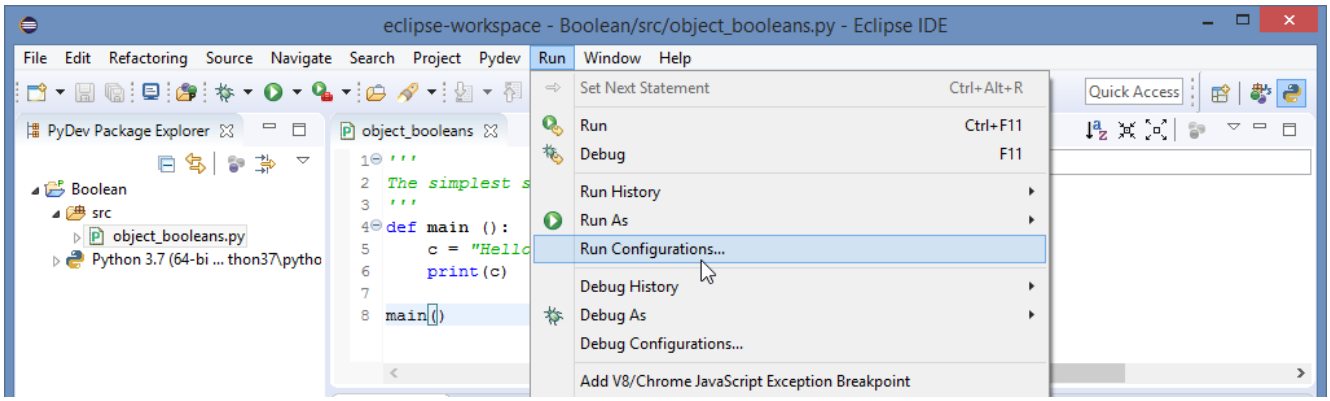


Figure 5.6.1 Opening the run configurations window

It opens the *Run Configurations* dialog. In the list on the left side of this window highlight the ***Python Run*** item and from its context menu select the ***New Configuration*** command (Figure 5.6.2):

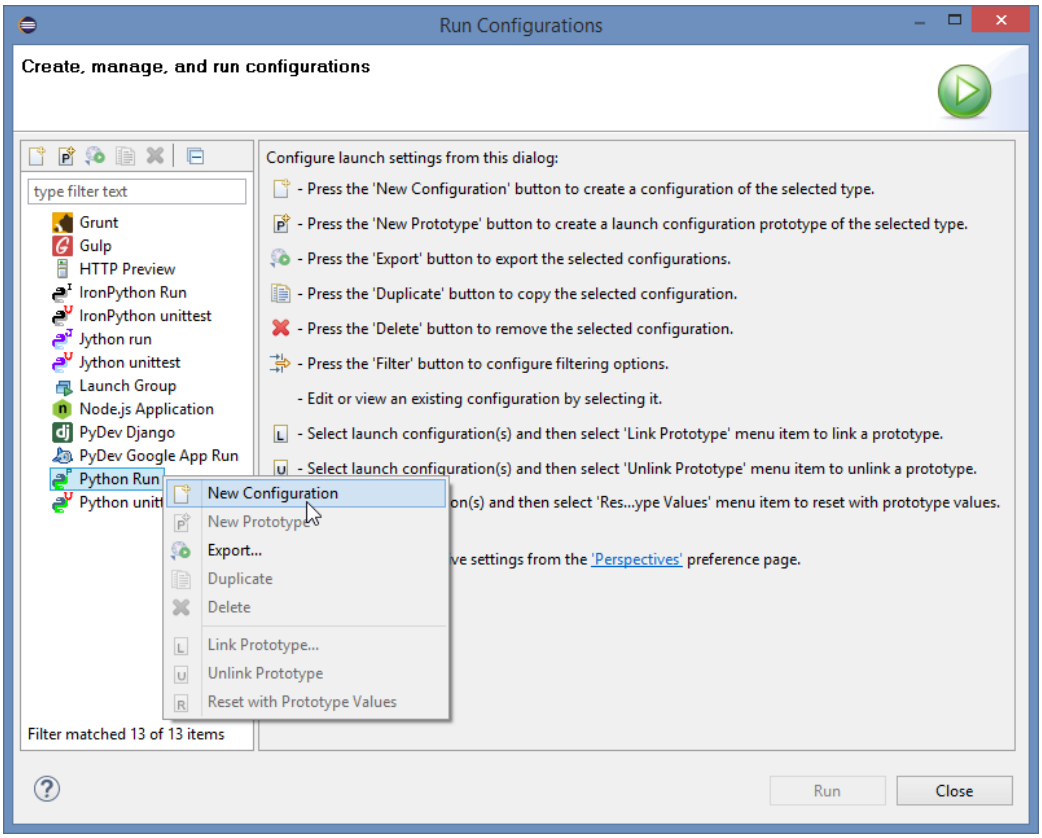


Figure 5.6.2 Creating a new *Run Configuration*

In response, Eclipse displays on the right side of this window an empty form for a new run configuration. Start by assigning it to the current project (select your project into the empty ***Project*** field – as in Figure 5.6.3):

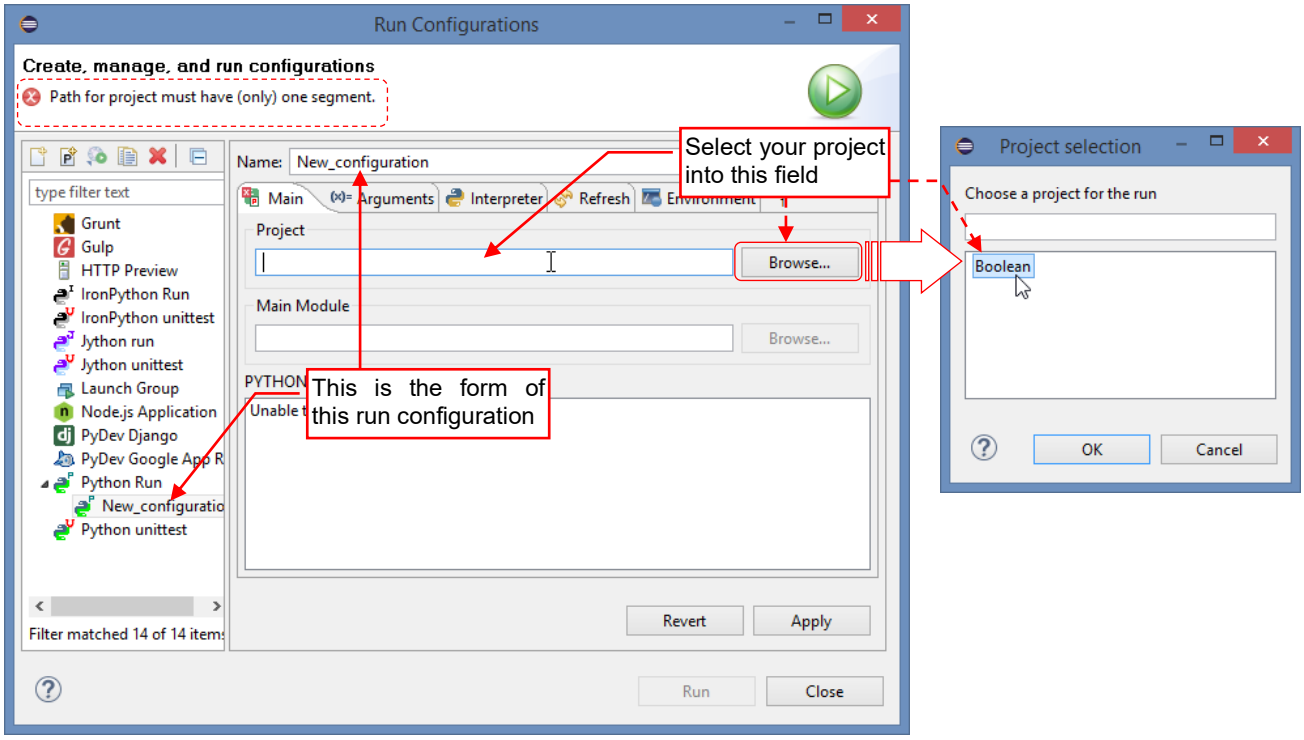


Figure 5.6.3 Assigning the project

Select the ***Main Module*** of this project (i.e. your script file):

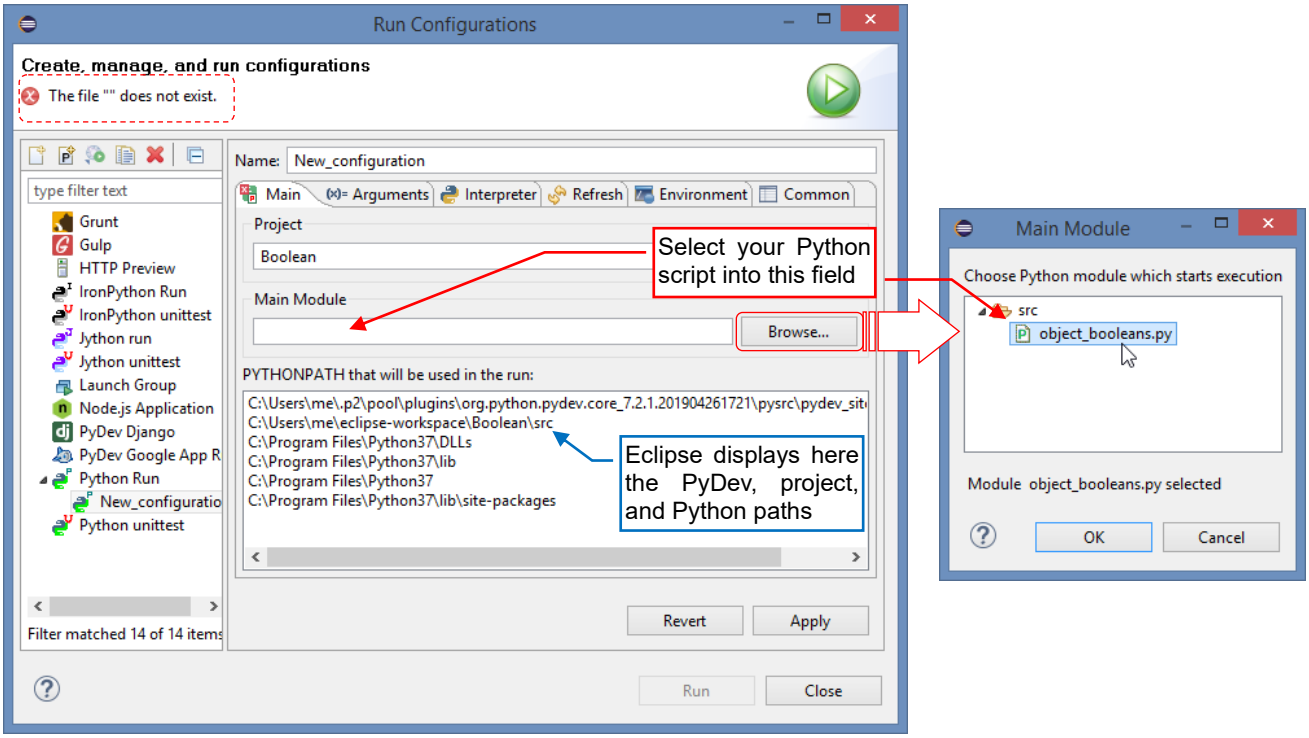


Figure 5.6.4 Assigning the main module

The file you have selected as the *Main Module* can be empty at this moment. Just add there the main procedure of your script before invoking the *Run* command.

You can also alter the ***Name*** of this configuration for a more descriptive one. Then switch to the ***Common*** tab and select this configuration as the default one for running and debugging (Figure 5.6.5):

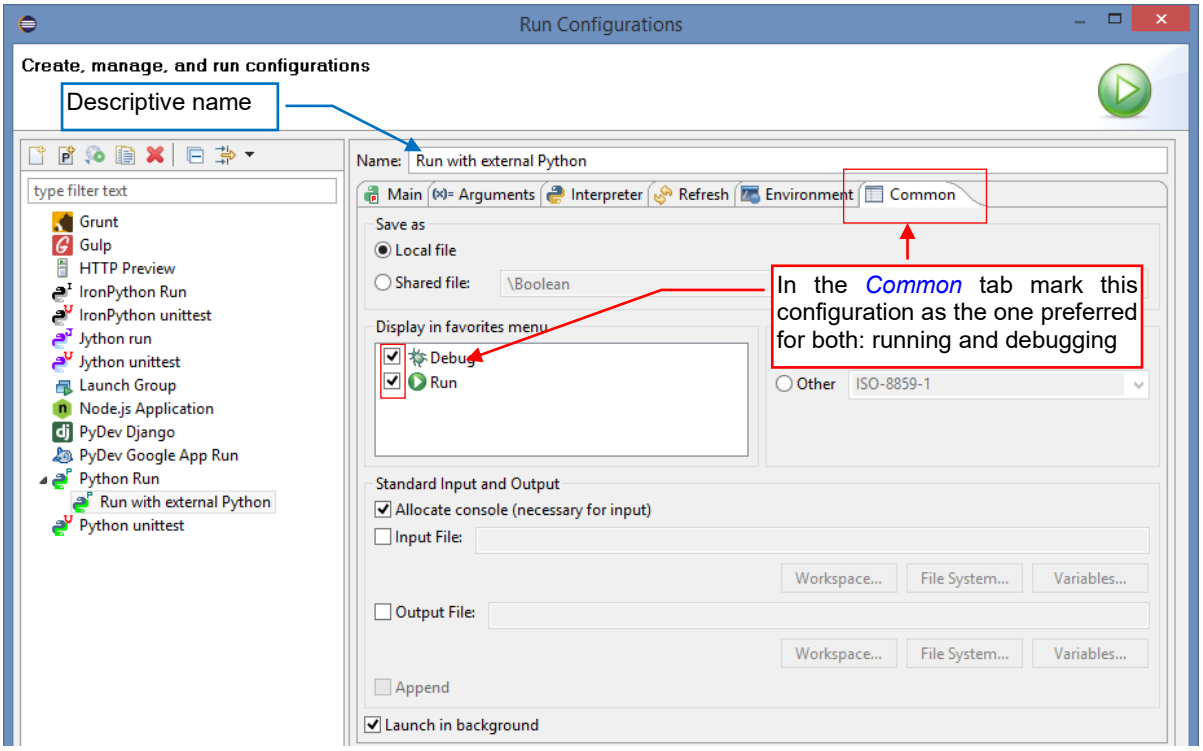


Figure 5.6.5 Marking this configuration as the default one

Save this configuration with the ***Close*** button (Figure 5.6.6):

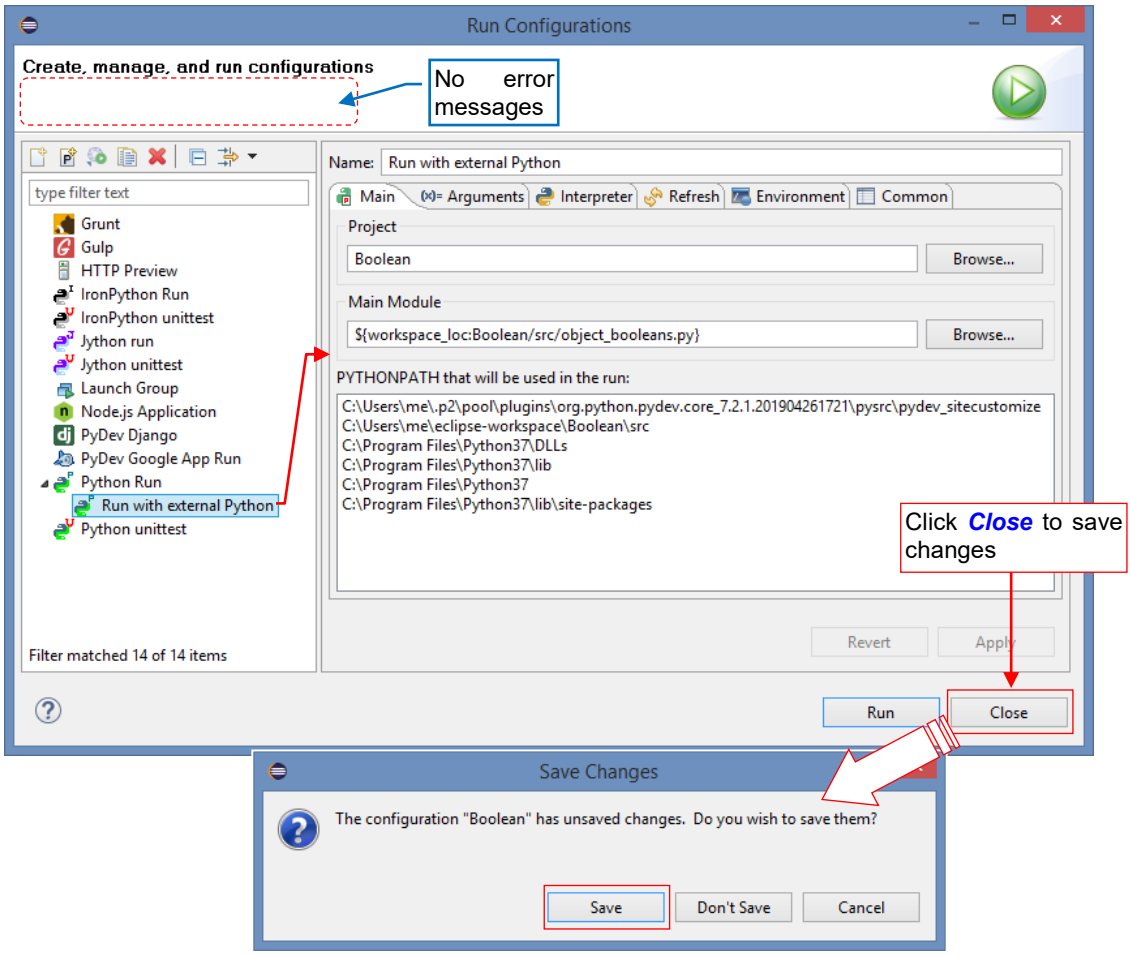


Figure 5.6.6 Saving the run configuration

Thanks to the settings you have made in the *Common* tab, this run configuration is displayed as the first item in both: *Debug* and *Run* menus (Figure 5.6.7):

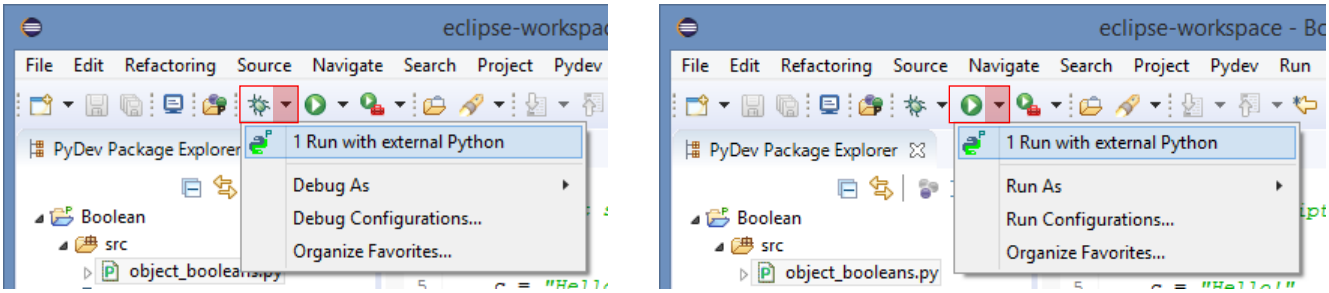


Figure 5.6.7 New command in the *Debug* and *Run* menus

Chapter 6. Others

In this chapter, you will find all the detailed materials that I am referencing in the main text of this book. Thus, this is an eclectic set of sections, describing details of various issues. You can find here solutions of eventual problems, which you may encounter while coupling Eclipse/PyDev IDE with Blender.

6.1 Updating Blender API predefinition files

In the zip package that accompanies this book there is *doc* folder (see page 39). In its *python_api\pypredef* directory I placed headers (“predefinition files” - **.pypredef*) of the Blender API (Figure 6.1.1):

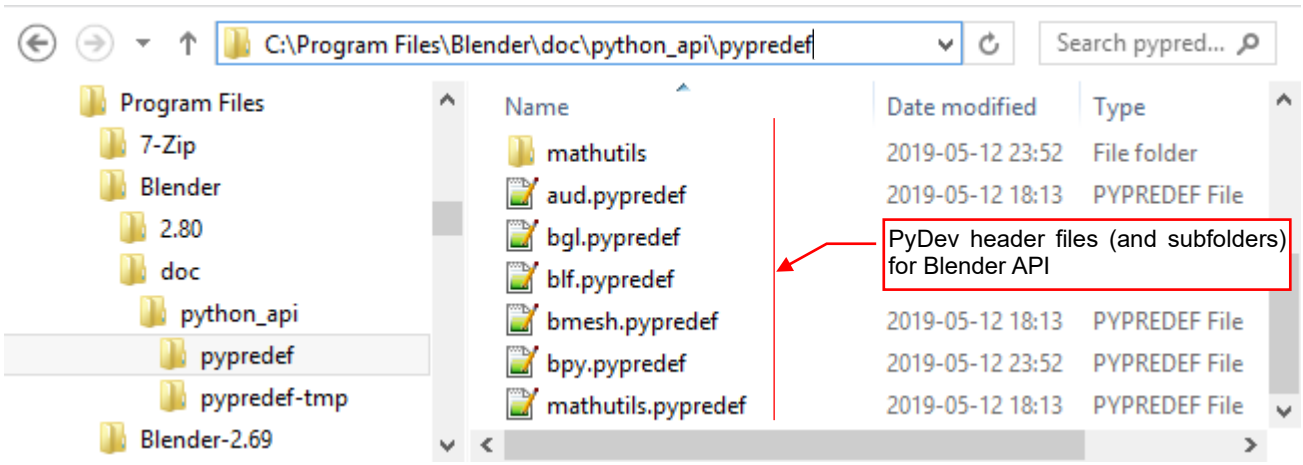


Figure 6.1.1 Contents of the *doc\python_api\pypredef* folder

You should add this folder to the *External Libraries* list in your PyDev project configuration (see page 40). PyDev will use this content for code autocompletion and for displaying descriptions of Blender API functions.

In each new Blender version, there are new API functions and classes. That’s why in the *doc* folder you can also find a shortcut named ***refresh_python_api.bat***, which updates the **.pypredef* files (Figure 6.1.2):

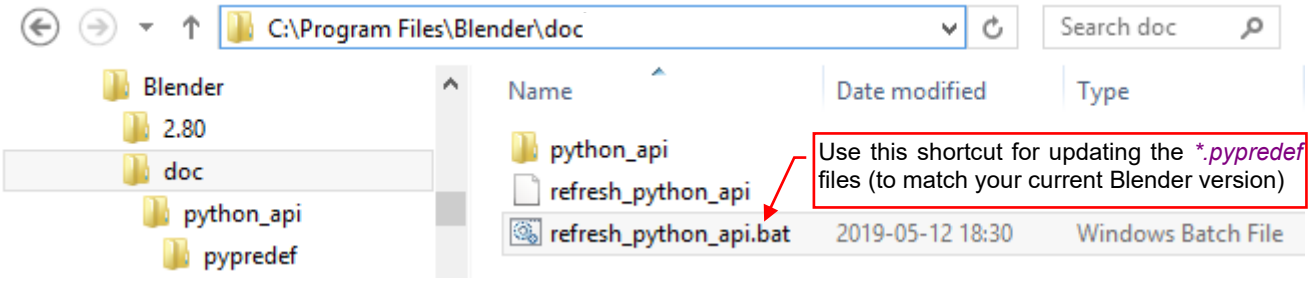


Figure 6.1.2 Contents of the *doc* folder

Use this shortcut when you install a new Blender version. The *refresh_python_api.bat* runs in Blender (in batch mode) the script named *pypredef_gen.py*, from *doc\python_api* directory (Figure 6.1.3):

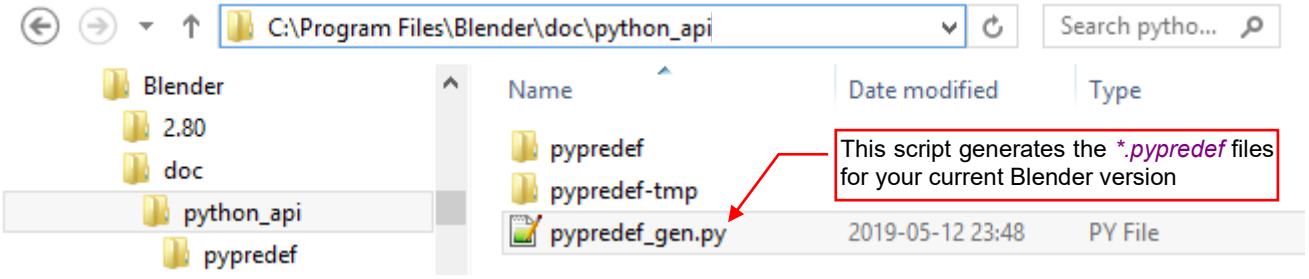


Figure 6.1.3 Contents of the *doc\python_api* folder

Theoretically, *pypredef_gen.py* should also run properly in other operating systems, like Linux. I have not tried it. This script is a modified version of the *sphinx_doc_gen.py*, developed by Campbell Barton for automatic generation of the Blender API documentation. (The same, which was published on the blender.org pages for Blender 2.5). Thanks to this code, in all the functions, classes and their methods in the PyDev predefinition files you can find the same descriptions as in the official API documentation.

When you run the *doc\refresh_python_api.bat* batch file, first you will see many "RNA warnings" in the system console. After them, the script displays information about eventual updates of the target files (Figure 6.1.4):

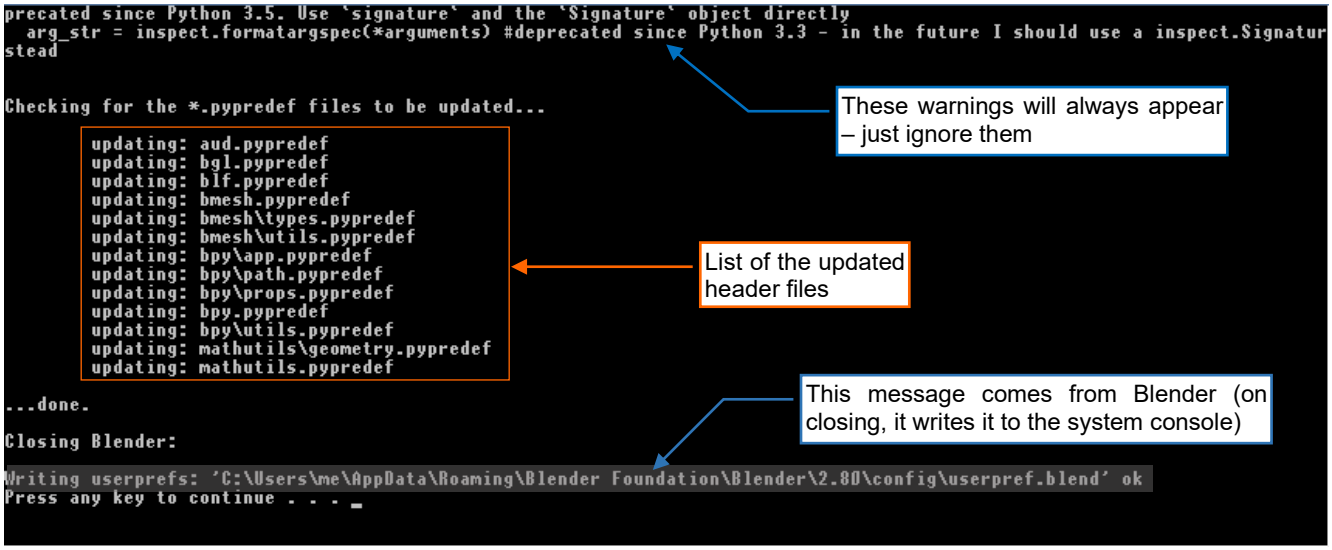


Figure 6.1.4 Updating the Blender API predefinition files

This last fragment is the most important: look at the lines that begin with "*updating:* …". They list the **.pypredef* files that have been updated. The second-last line ("*Writing userprefs:…*") comes from the Blender (it prints it while quitting), while the "*Press any key to continue…*" phrase comes from the standard *pause* command, placed at the end of the batch file.

However, if a runtime error has occurred, the result of the batch file in system console looks like in Figure 6.1.5:

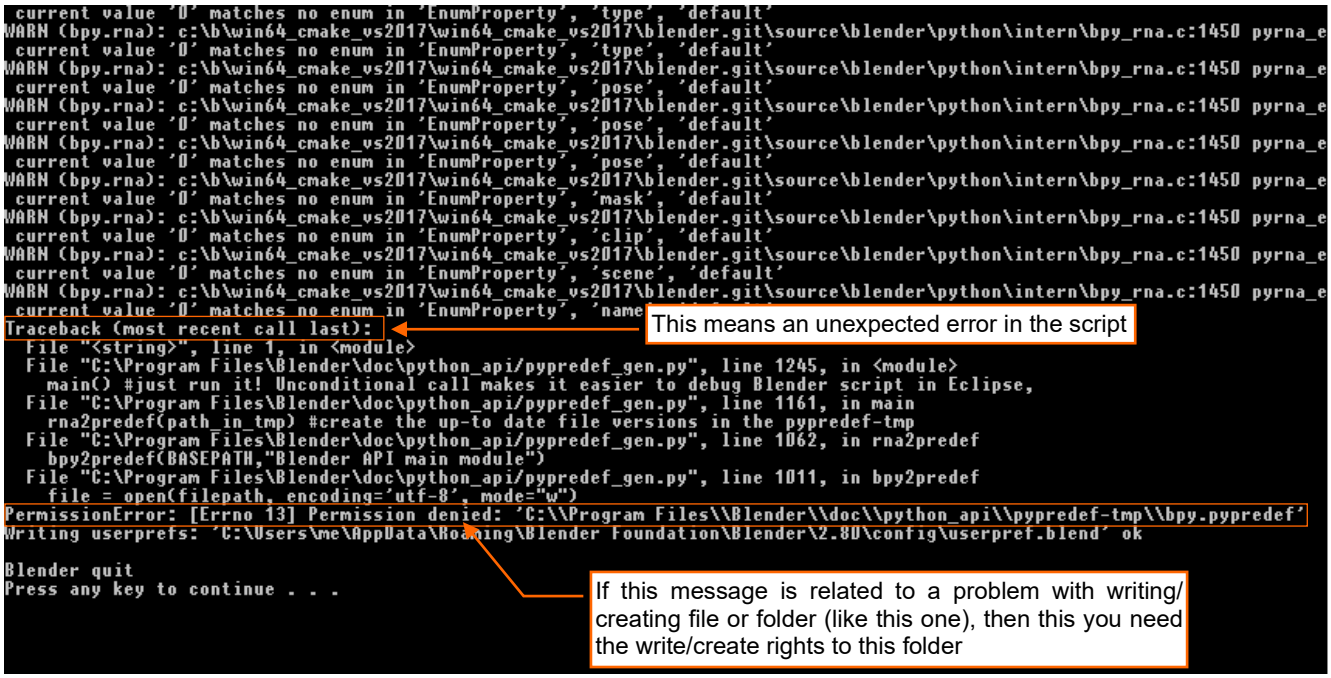


Figure 6.1.5 Script error message

If the error message (you can find it below the traceback printout, as in the figure above) says about missing permission or a problem with creating/writing to a file, then you need more (OS) privileges to *doc* folder.

- In Windows the Blender directory (*C:\Program Files*) is treated in a specific way. By default, no Windows program can create/write files in its subdirectories. Thus, if you placed the *doc* folder in the *C:\Program Files\Blender* directory (as I am showing on page 39), you have to alter the user privileges. Grant the writing rights to this directory to the built-in *Users* group.

If you have no Administrator privileges to your computer – you can put *doc* folder on another directory, where you have the right to write/create files.

- You can unpack the *doc* folder to any suitable place in your computer, for example – into your *Documents*. In such a case just remember to make a minor update in the *refresh_python_api.bat* file. Replace there the relative path ("\..") to *blender.exe* with the full path, for example:

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

At the end of this section – a few notes:

1. The ***bgl*** module header contains just the constants symbols and function names (there are no function parameters). This is because Blender developers did not not document in API (there are no ***__doc__*** fields, nor the "RNA" information, as in the other modules). On the other hand, Blender 2.8 documentation suggest switching from this "old fashioned" OpenGL 1.1 ***bgl*** interface to the modern one, available in the ***gpu*** module. (The ***gpu*** module methods are also much faster);
2. BMesh operators (defined in the ***bmesh.ops***) are not documented. They do not provide the "RNA" information, as in the case of the ***bpy*** module. Their ***__doc__*** fields contain just the function declarations, without any description;
3. At the end of the main API header – *bpy.pypredef* file – you can find many simplified class declarations for all the Blender panels and menus. They are useful in the case, when you want to add your command/submenu to any of these GUI elements. Otherwise PyDev editor would mark such a class name as an error. While such an error does not prevent your Python code from successful running and debugging, it is better to stick to the "no errors signalized by the PyDev editor" rule. In this way you can avoid many time-consuming issues. All the standard menus are derived from ***bpy.types.Menu*** class and have "_MT_" symbol in the middle of their names. All the standard panels are derived from ***bpy.types.Panel*** class and have "_PT_" in their names. They also have prefixes, written in capitals, which denote the windows (spaces) where they are used. For example, a class named ***bpy.types.VIEW3D_MT_object*** represents the *Object* menu from the *3D View* window.
4. Nearly all fields of the ***bpy.context*** (***bpy.types.Context*** class) are copied into the *bpy.pypredef* file from the fixed text that I put into the *pypredef_gen.py* script. This is a specific object: its field set changes, depending on the window (Blender screen area) from which the Python script is invoked. Some fields are available only in the *3D View*, other in the *Properties* window. I placed in the header file all the possible fields and noted in the comments their eventual window dependencies (Figure 6.1.6):

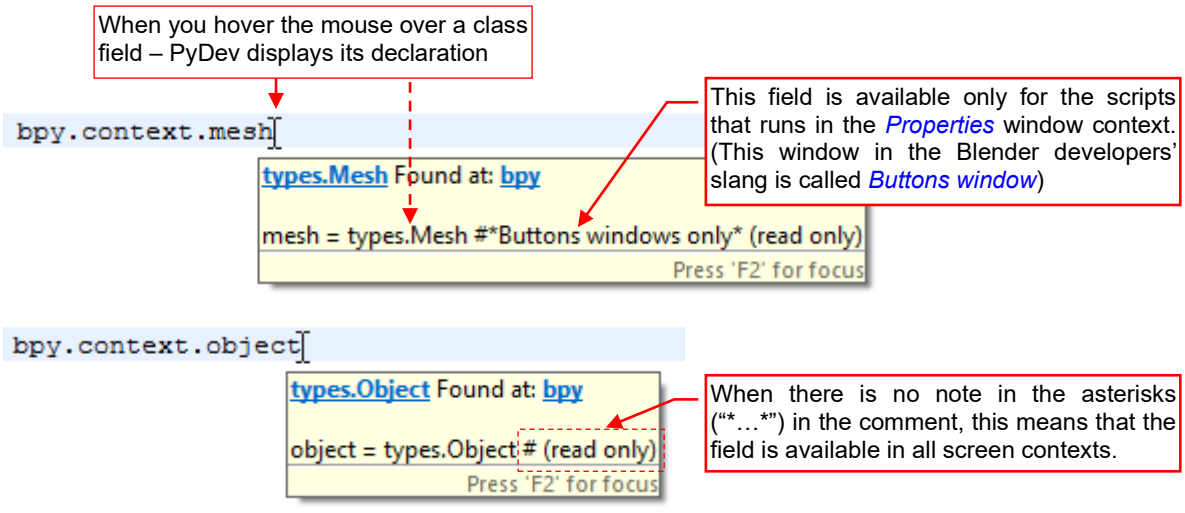


Figure 6.1.6 Opening the *PyDev - PYTHONPATH* pane

- After every update of the *pypredef* files, update also PyDev internal info, as described on page 143

6.2 Enabling Blender API code autocompletion in a PyDev project

When you start in the PyDev a new Blender add-on project, add to the **PYTHONPATH** variable the path to the *doc\python_api\pypredef* directory. This folder contains declarations of Blender API methods, classes and constants (see page 139).

To do it, open ***Project→Properties*** (Figure 6.2.1):

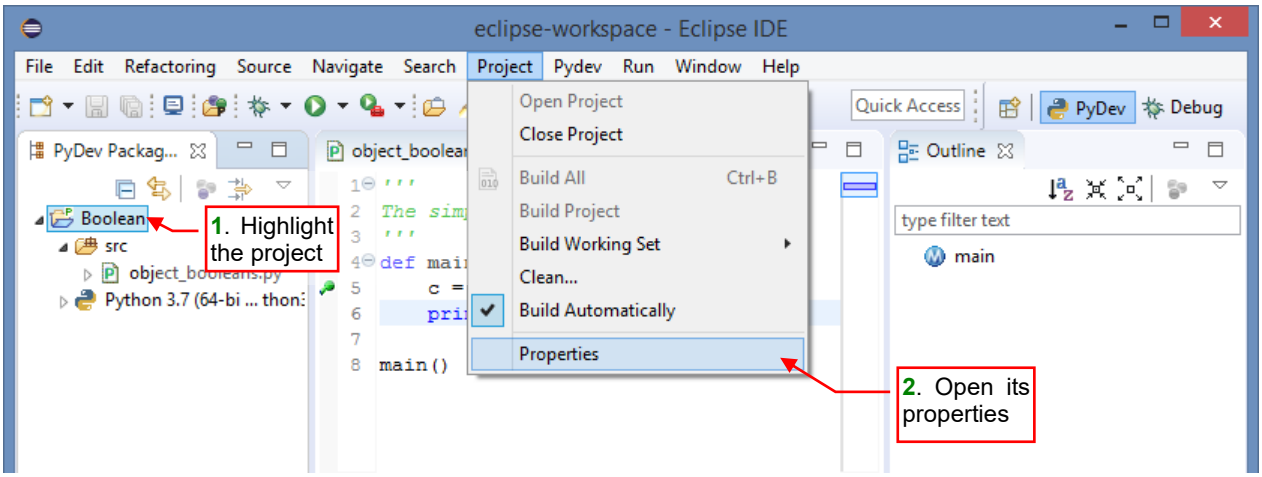


Figure 6.2.1 Opening project properties

In the properties dialog, highlight the ***PyDev – PYTHONPATH*** section, then select its ***External Libraries*** tab (Figure 6.2.2):

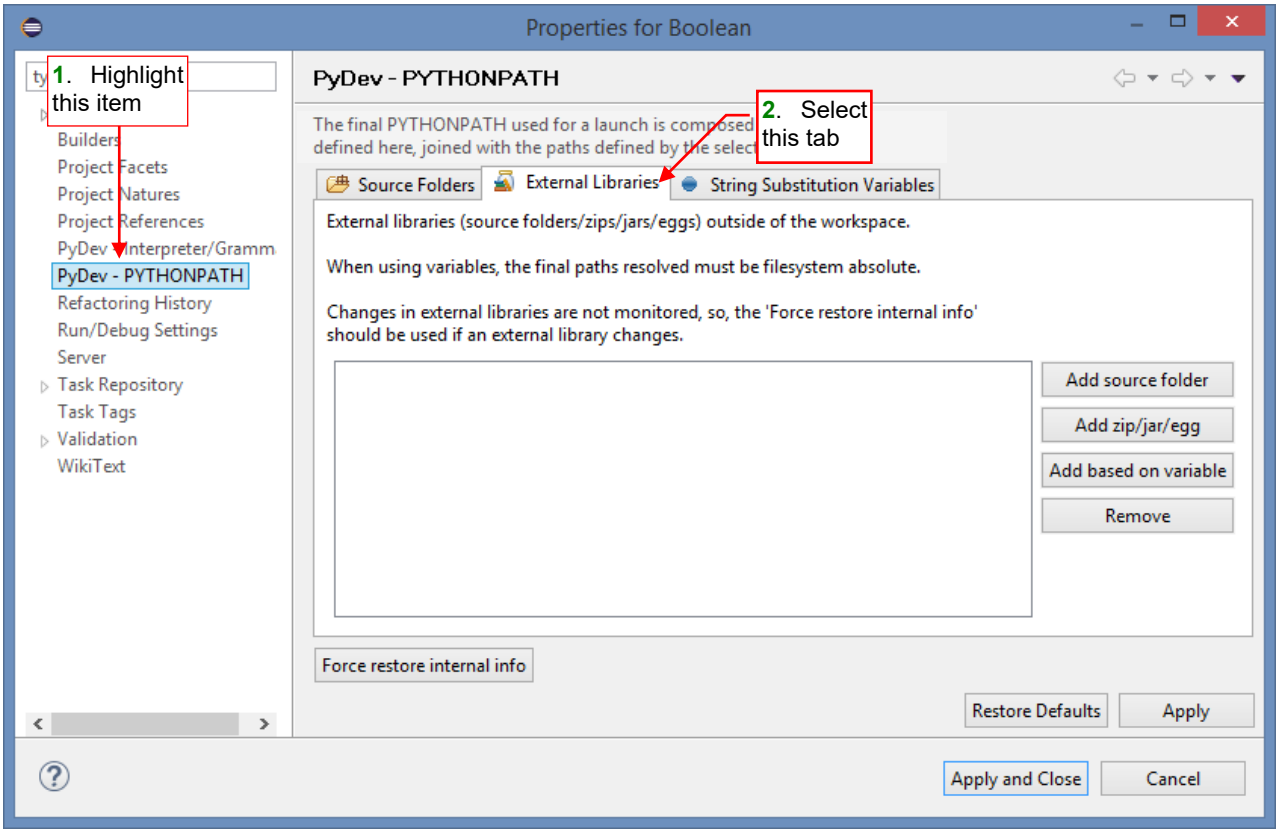


Figure 6.2.2 Opening the *PYTHONPATH* form

Initially there are no external libraries (the list in this tab is empty). Click the ***Add source folder*** button to add a directory to this list, and in the dialog box select the *doc\python_api\pypredef* folder (Figure 6.2.3):

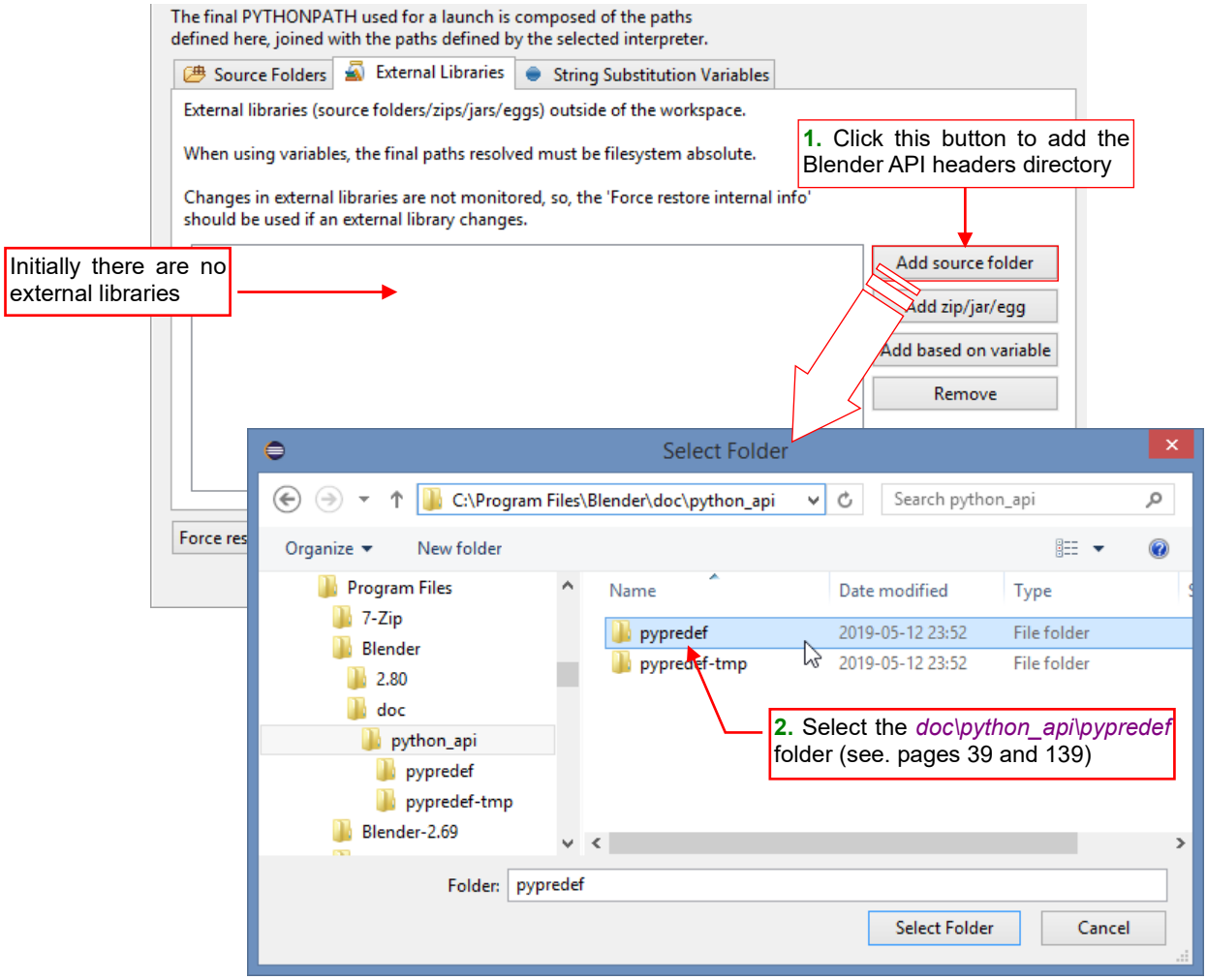


Figure 6.2.3 Declaring the *pypredef* folder as the Python "external library"

When the Blender API headers folder has appeared on the external libraries list, click the ***Force restore internal info*** button. According its description, you should do this after every change made to the header files:

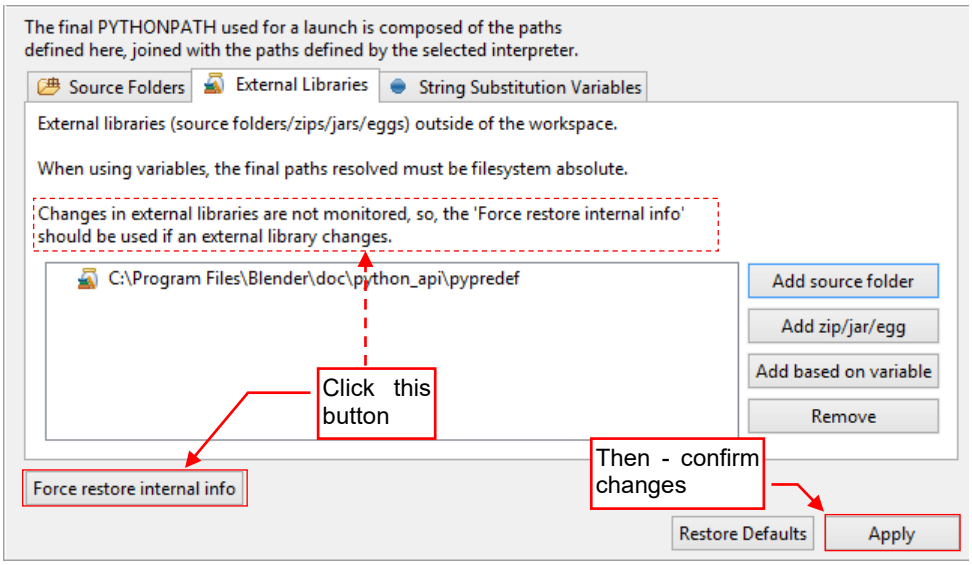


Figure 6.2.4 Refreshing internal PyDev information and closing this dialog

Finally, click the ***Apply*** button to save these changes.

Once the project configuration is updated, add to your script appropriate ***import*** statement. Usually you start by importing the ***bpy*** module. Then, when you type a dot after a class variable name, PyDev will display the list of the class fields and methods (Figure 6.2.3):

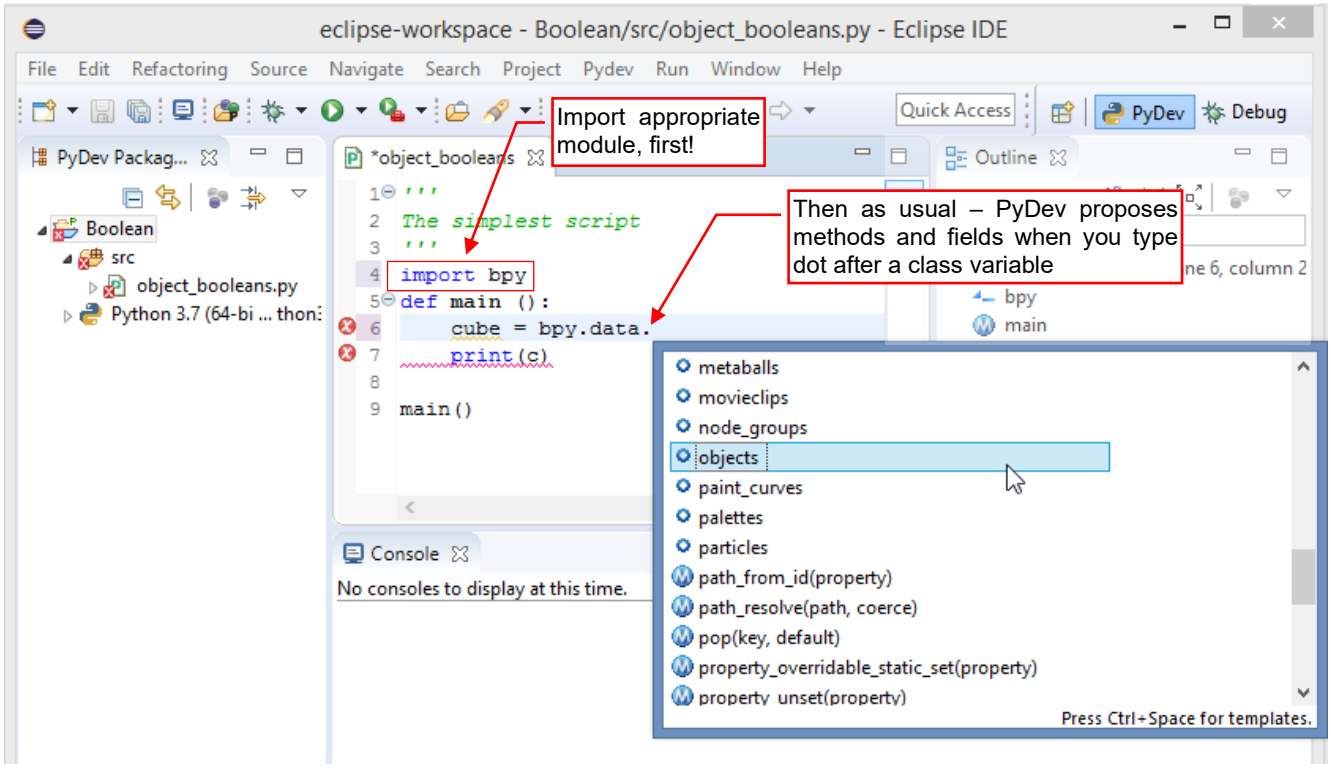


Figure 6.2.5 Blender API code autocompletion – on placing a dot after the class variable (or Ctrl-Space)

You can also use the **Ctrl**-**Space** shortcut.

To learn more about PyDev autocompletion – see page 41.

6.3 Importing/linking an existing file to a PyDev project

You can copy (import) to your PyDev project an existing file from your disk. Just “grab” it with mouse (for example – in the File Explorer window), then drag and drop into your Eclipse project (Figure 6.3.1):

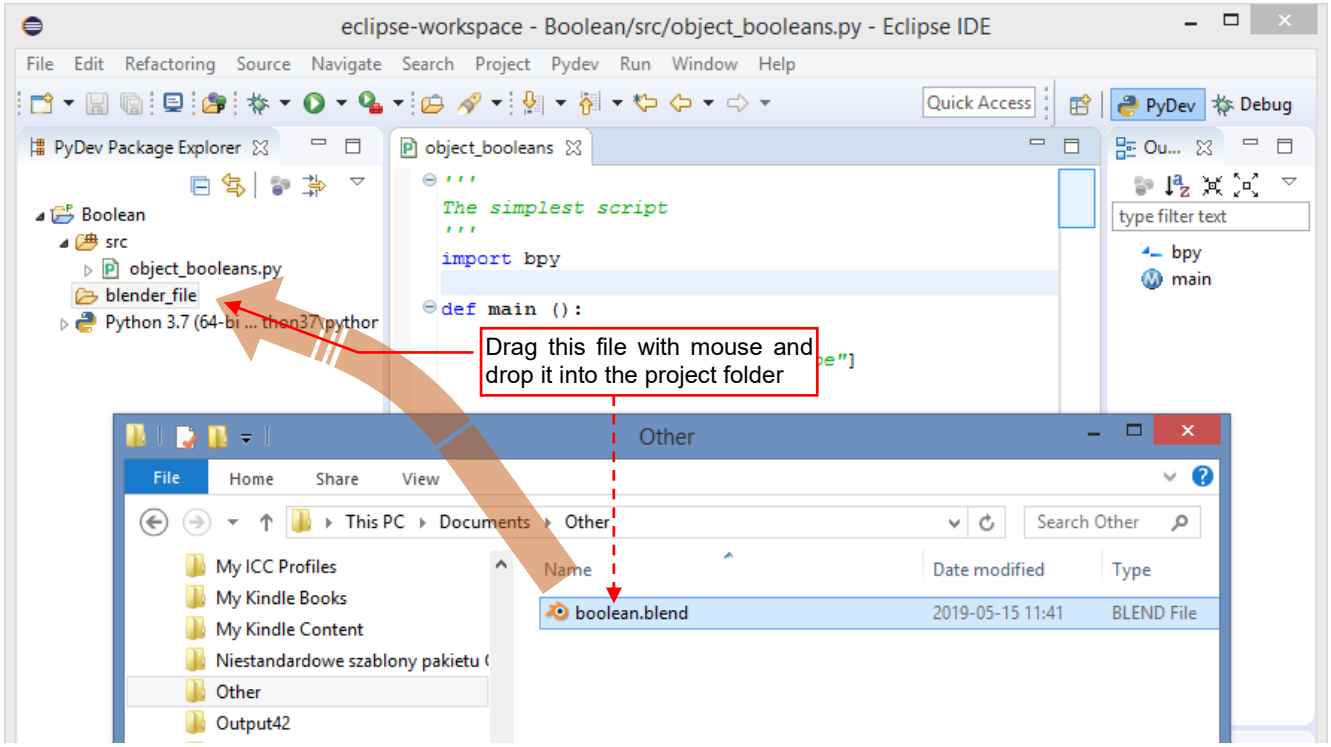


Figure 6.3.1 Importing an existing file to the project folder

It creates a copy of this file in the selected folder of the Eclipse project (Figure 6.3.2):

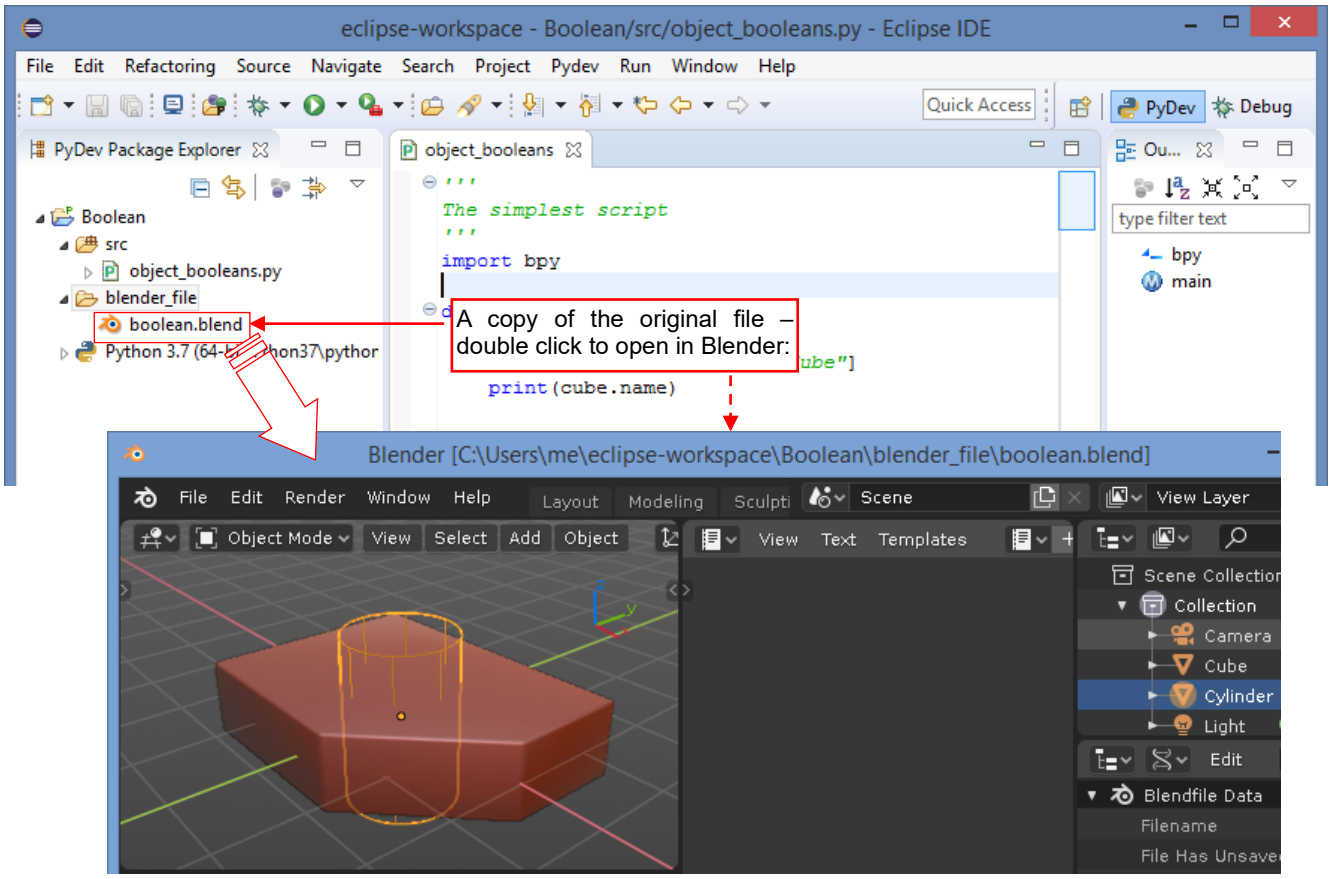


Figure 6.3.2 A test Blender file, attached to a PyDev project

In addition to attaching files, you can also link to the Eclipse project any file located in a different directory on your disk. However, in PyDev projects you have to do it in a less straightforward method than the “drag and drop”[1]. Highlight the target folder in your project and in the context menu click ***New→File*** (Figure 6.3.3):

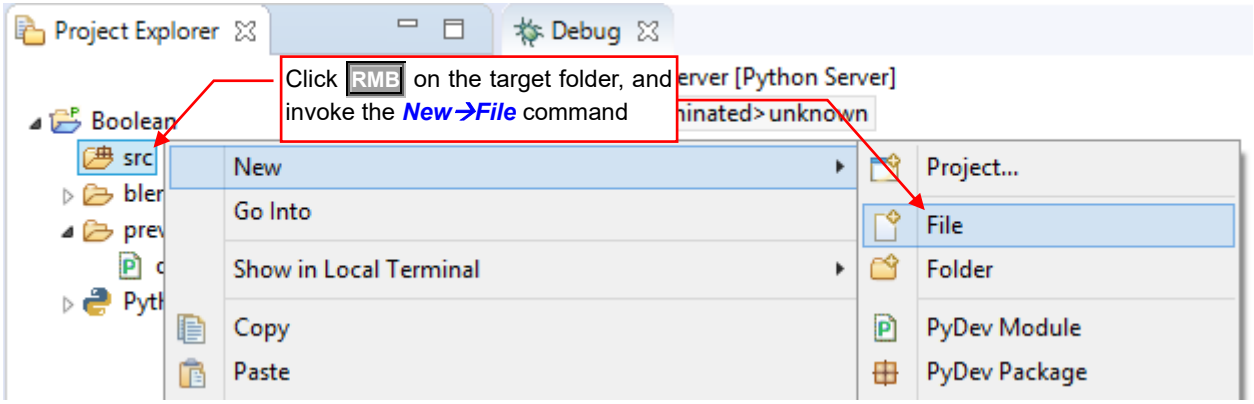


Figure 6.3.3 Invoking the *File→New* command

Enable the ***Advanced*** option in the *New File* creator dialog, and mark ***Link to file …*** option (Figure 6.3.4):

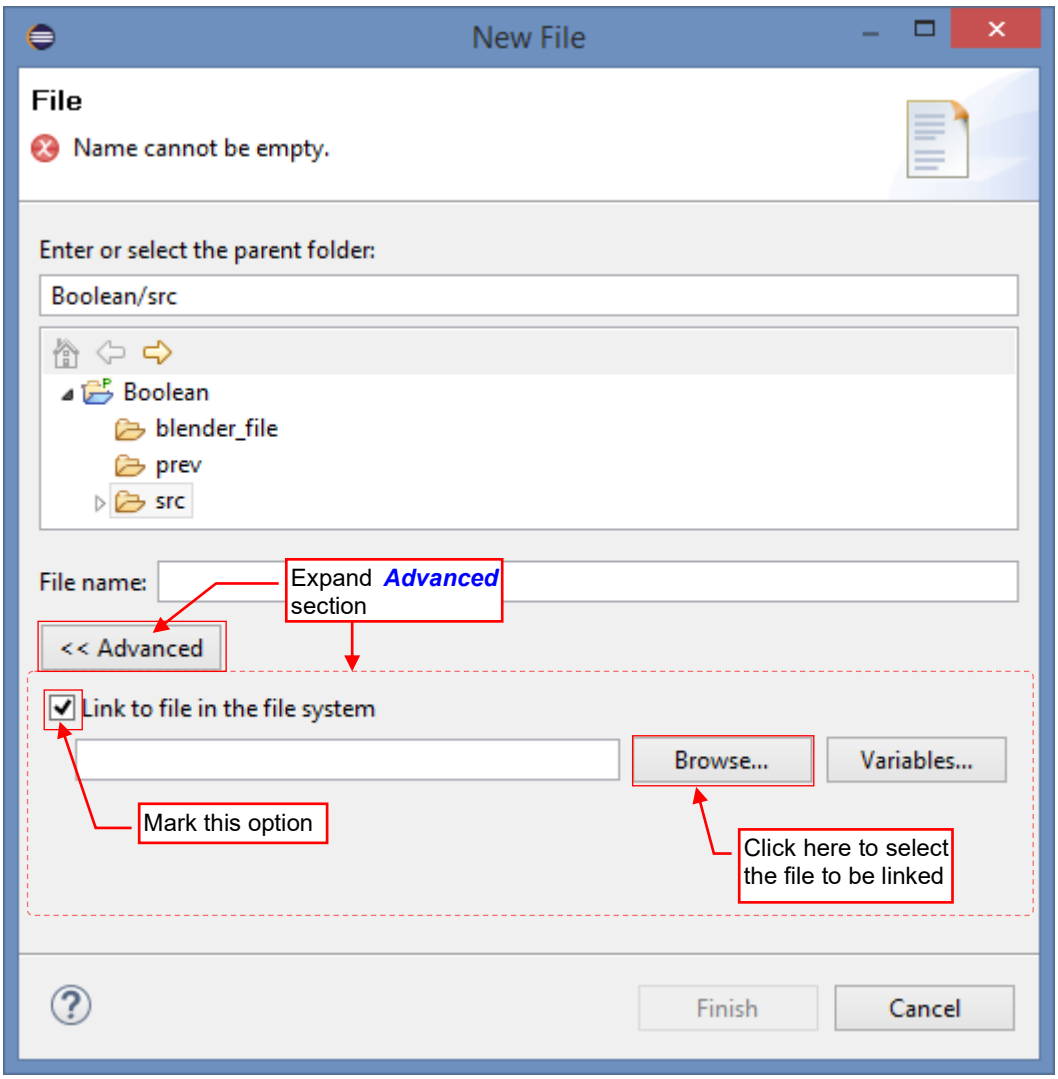


Figure 6.3.4 Selection of the source type

[1] From unknown reasons PyDev ignores the default Eclipse settings (from *Window→Preferences:Workspace\Linked Resources*). According these settings after receiving a dropped file, PyDev should ask whether to import (copy) or link this file.

In the file selection dialog box select the source file (Figure 6.3.5):

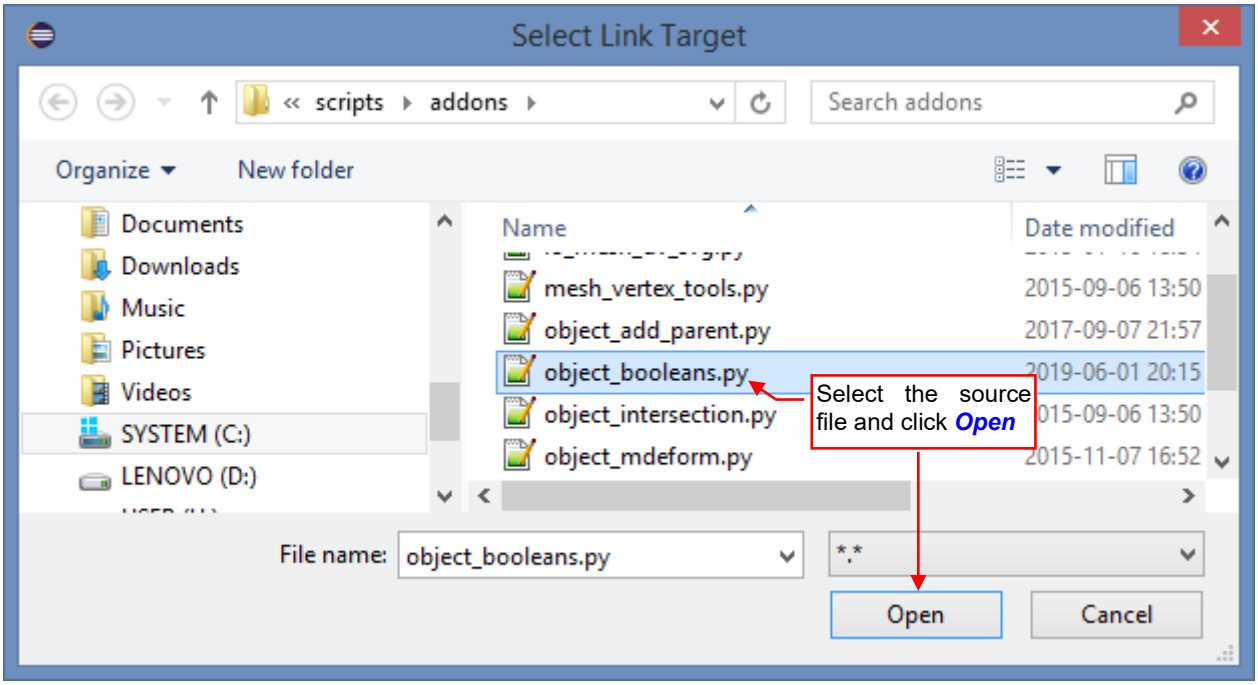


Figure 6.3.5 Selecting the source file for linking

When you click ***Open***, you come back to the creator dialog (Figure 6.3.6):

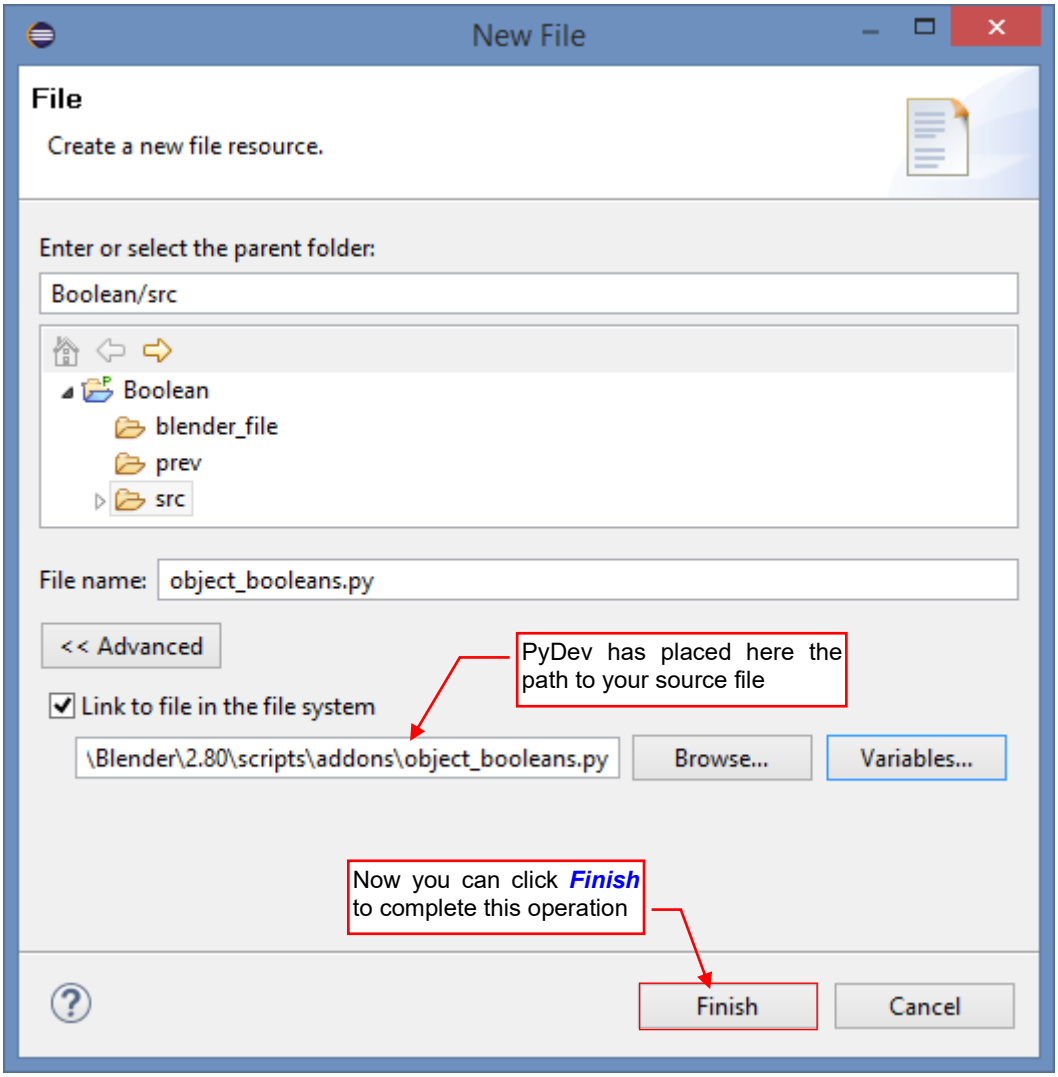


Figure 6.3.6 Filled *New File* creator form

Click the ***Finish*** button in the creator window. In the result, PyDev will create a shortcut (link) to the original file. In this way you can easily connect to your project an existing Blender add-on, even when it is already installed in the Blender add-on directory (Figure 6.3.7):

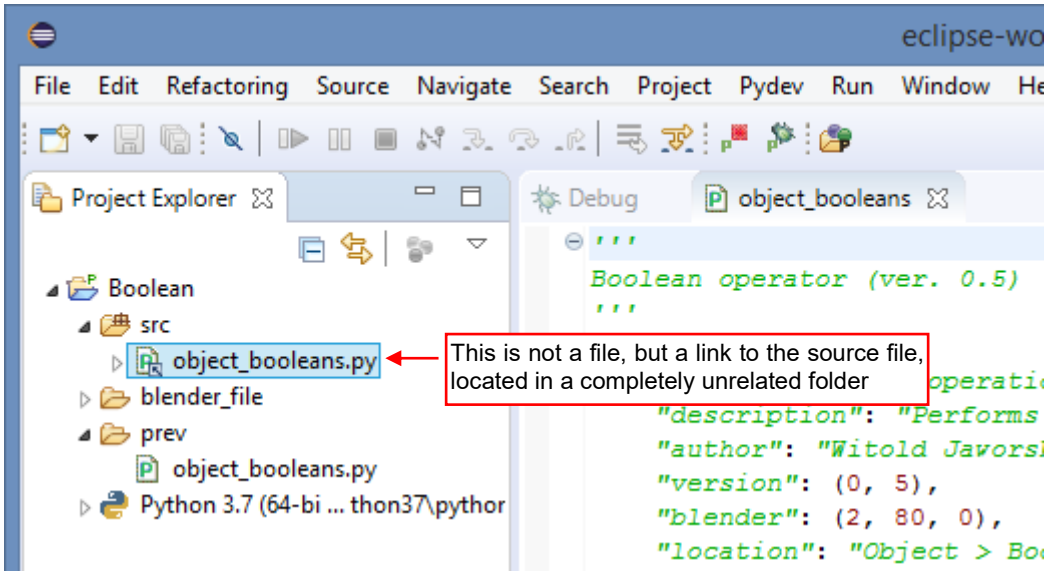


Figure 6.3.7 File link, added to a PyDev project

As you can see in figure above, file links icons in Eclipse are marked with additional arrow at their right, lower corner. When you open properties of this link, you can read the full path to the source file. You can also alter this path there (Figure 6.3.8):

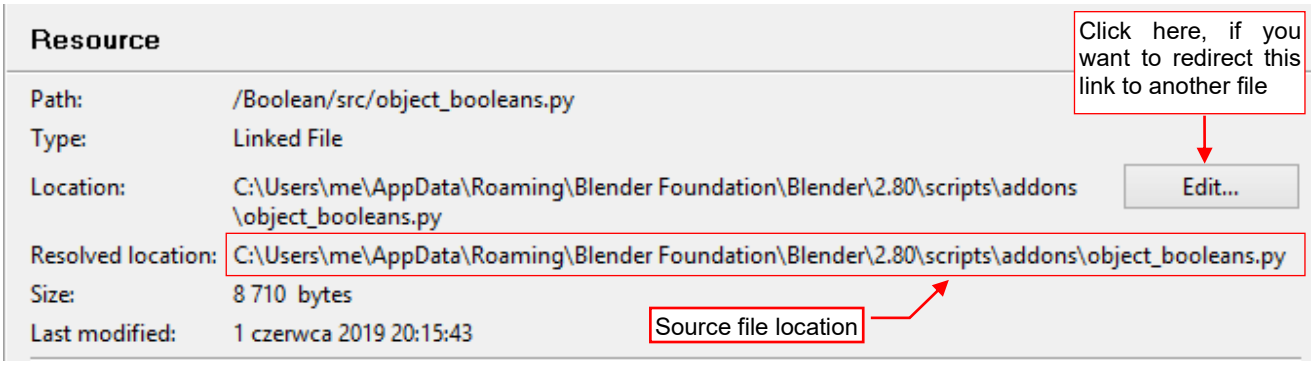


Figure 6.3.8 PyDev *Properties* window of a linked file (opened by clicking RMB)

6.4 Details of debugging Blender scripts

Blender uses its own, embedded interpreter for executing Python scripts. You can debug them using the built-in, standard Python debugger. Unfortunately, this tool works in the "conversational" mode, in the console. Thus, this is not a user-friendly solution.

You need so-called remote debugger, to follow the script execution in an IDE such as Eclipse. This solution was originally invented for debugging programs that are running on another computer (Figure 6.4.1):

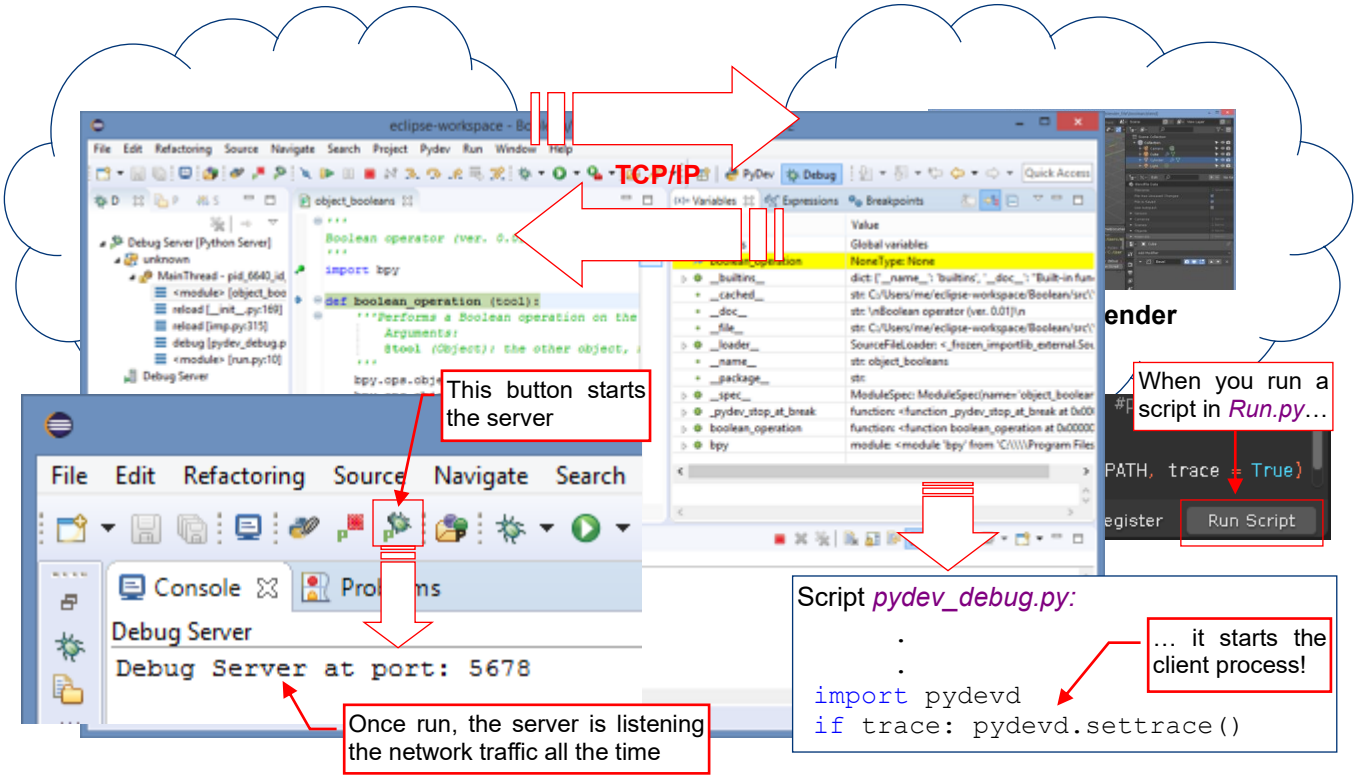


Figure 6.4.1 Tracing the Blender script execution in the PyDev remote debugger

In the IDE (like Eclipse) you have to run the server process. It starts "listening" to eventual requests from the debugged scripts. These requests will be sent by a remote debugger client, activated in the code of the tracked script. The communication between the remote debugger client and its server is realized through the network. Long ago, someone noticed that you can also run these two processes on the same machine. They exchange data using the local network card of the computer. Conceptually, this corresponds to a situation, where two persons are sitting in the same room and talking to each other via a phone. Fortunately, the programs are "stupid" and do not complain that they have to communicate in such a strange way. From the user point of view, this solution works without flaws. Just beware the firewalls. In the PyDev, the debugger client code is in the ***pydevd*** Python package. In the *Run.py* script template I use the *pydev_debug.py* auxiliary module (see page 160), which imports and initializes PyDev debugger client. (*Run.py* is the “testbed” of our scripts — see page 53).

Apart the button shown in previous figure, you can also start the remote debugger server using the ***PyDev→Start Debug Server*** command, (Figure 6.4.2):

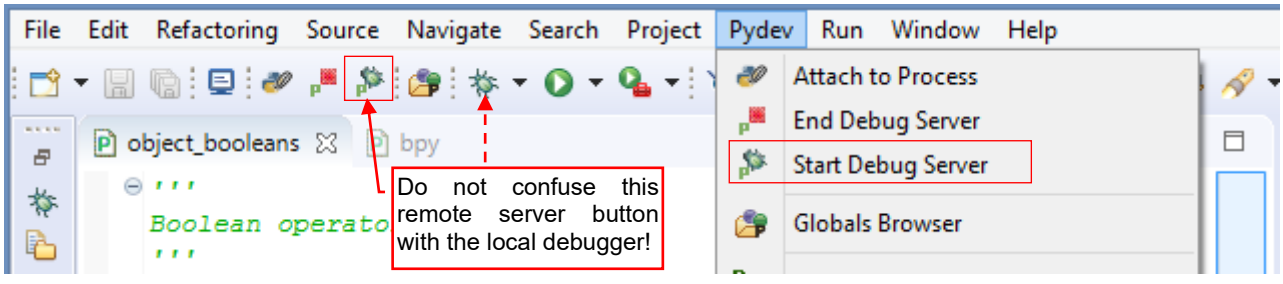


Figure 6.4.2 Commands that control the PyDev remote debugger server

What to do, when these PyDev commands do not appear[1] on the toolbar nor menu, as in Figure 6.4.2? Sometimes the *Start/End Debug Server* commands can be just turned off in the *Debug* perspective! To enable them, use the ***Window→Perspective→Customize Perspective*** command (Figure 6.4.3):

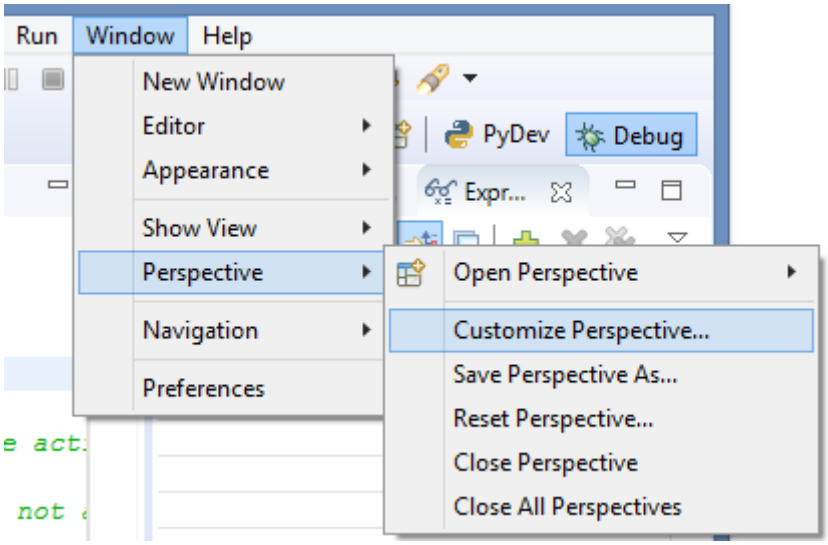


Figure 6.4.3 Opening the *Customize Perspective* window

In the ***Customize Perspective*** window, open the ***Action Set Availability*** tab (Figure 6.4.4):

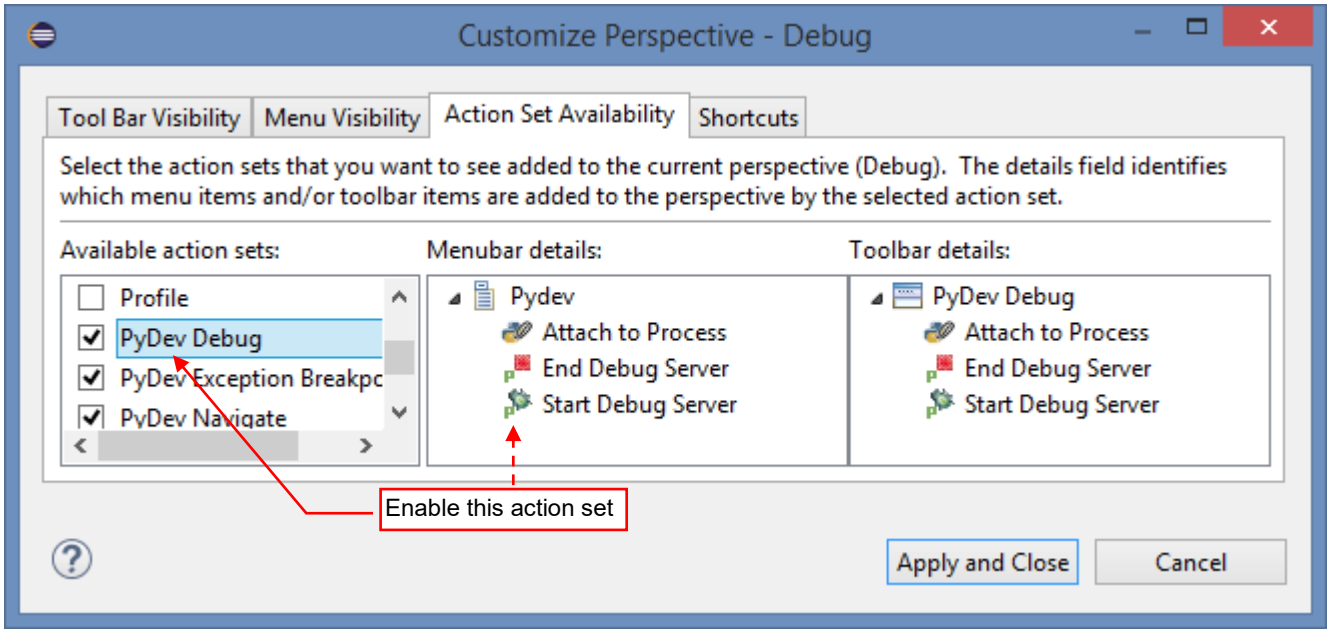


Figure 6.4.4 Enabling the PyDev remote debugger controls

Find in the *Available action set* list (on the left) a set named ***PyDev Debug***. Just enable it and then ***Apply and Close*** this dialog. The *Start Debug Server* and *End Debug Server* will appear in your current perspective.

When I prepared this book, the *Start/End Debug Server* buttons were already visible in the *Debug* perspective. I did not have to add this action set manually, as described above. I suppose that this issue may be related to the way in which you have added the *Debug* perspective to your project. (However, I am not sure).

- By the way, you have learned how to customize Eclipse project perspective ☺.

[1] When I installed PyDev for the first time, such a thing just happened in my Eclipse. I spent whole day browsing through all the PyDev documentation and the user posts from various Internet forums. In parallel, I continually searched various Eclipse menus, looking for these two missing commands. In the end, I found them. To save you from similar troubles, I am describing here the solution.

Let's take care now for the debugger client. Among files that accompany this book you can find the *Run.py* script (it is accompanying by the *pydev_debug.py* file - see page 39). Load *Run.py* into Blender *Text Editor* window (see page 47) using the ***Text→Open*** command. Figure 6.4.5 shows its initial contents:

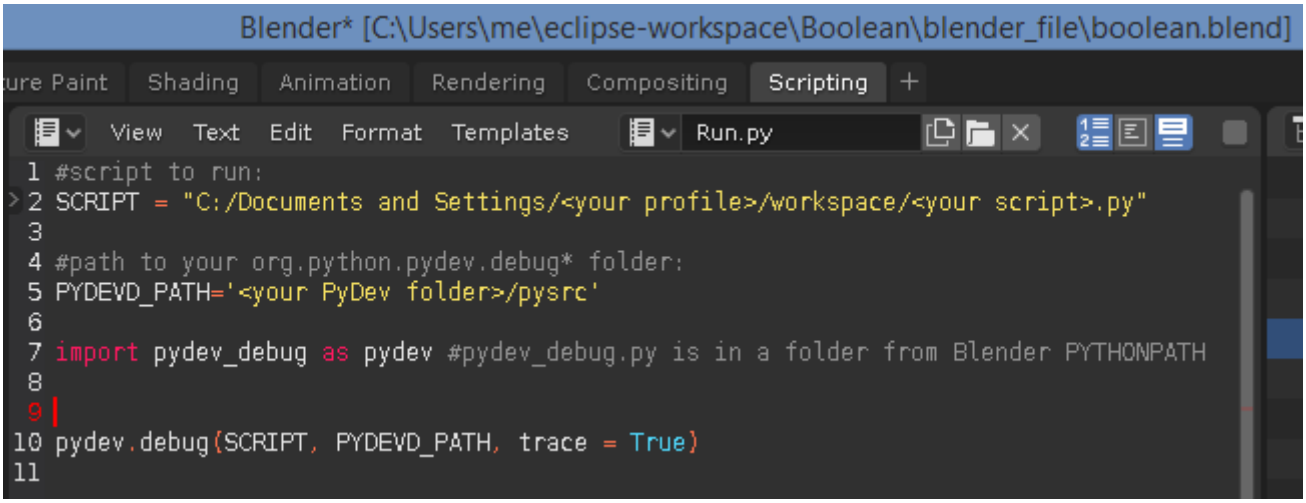


Figure 6.4.5 Auxiliary code for running user scripts in Blender

To debug your script, you have to customize the two constants in this code: ***SCRIPT*** and ***PYDEVD_PATH***. But first, let's check if the Python in Blender can import the ***pydev_debug*** module, as in the *Run.py* script. (Technically speaking: let's check if the *pydev_debug.py* file is present in one of the Blender **PYTHONPATH** directories. One of them is the directory that contains the Blender executable file). You can "mimic" this line of the code in the *Python Console*, as in Figure 6.4.6, and make sure that you have got similar response:

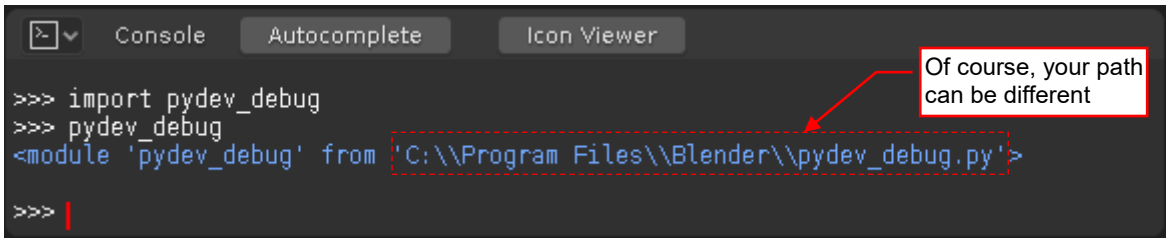


Figure 6.4.6 Checking, if Python in Blender can find the *pydev_debug* module

If the import statement of this module causes an error – check carefully in the Blender *Python Console* which directories are listed in its ***sys.path***. Then place the *pydev_debug.py* file into one of these folders.

In the next step, assign to the ***SCRIPT*** constant the full path to your script file. You can easily copy it from the Eclipse properties window for this project item (Figure 6.4.7):

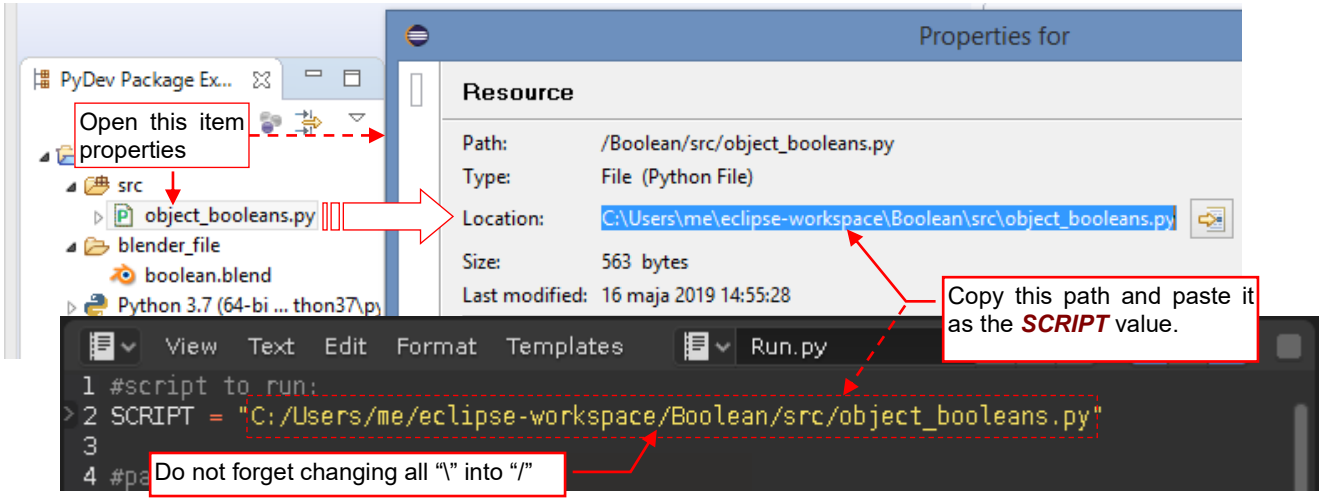


Figure 6.4.7 Typing the full path of the script to be run

The last element you have to update in the *Run.py* code is the path to a PyDev subfolder named *pysrc* (the ***PYDEVD_PATH*** constant). This is a more difficult, because this subfolder location can be different in various PyDev versions. The simplest way to find it in your PyDev is to read its path from the PyDev **PYTHONPATH**. To do it, open (as I am showing in section 5.6, on page 134) the ***Run Configrations*** dialog, and read it from the **PYTHONPATH** directories listed in the default running configuration (Figure 6.4.8):

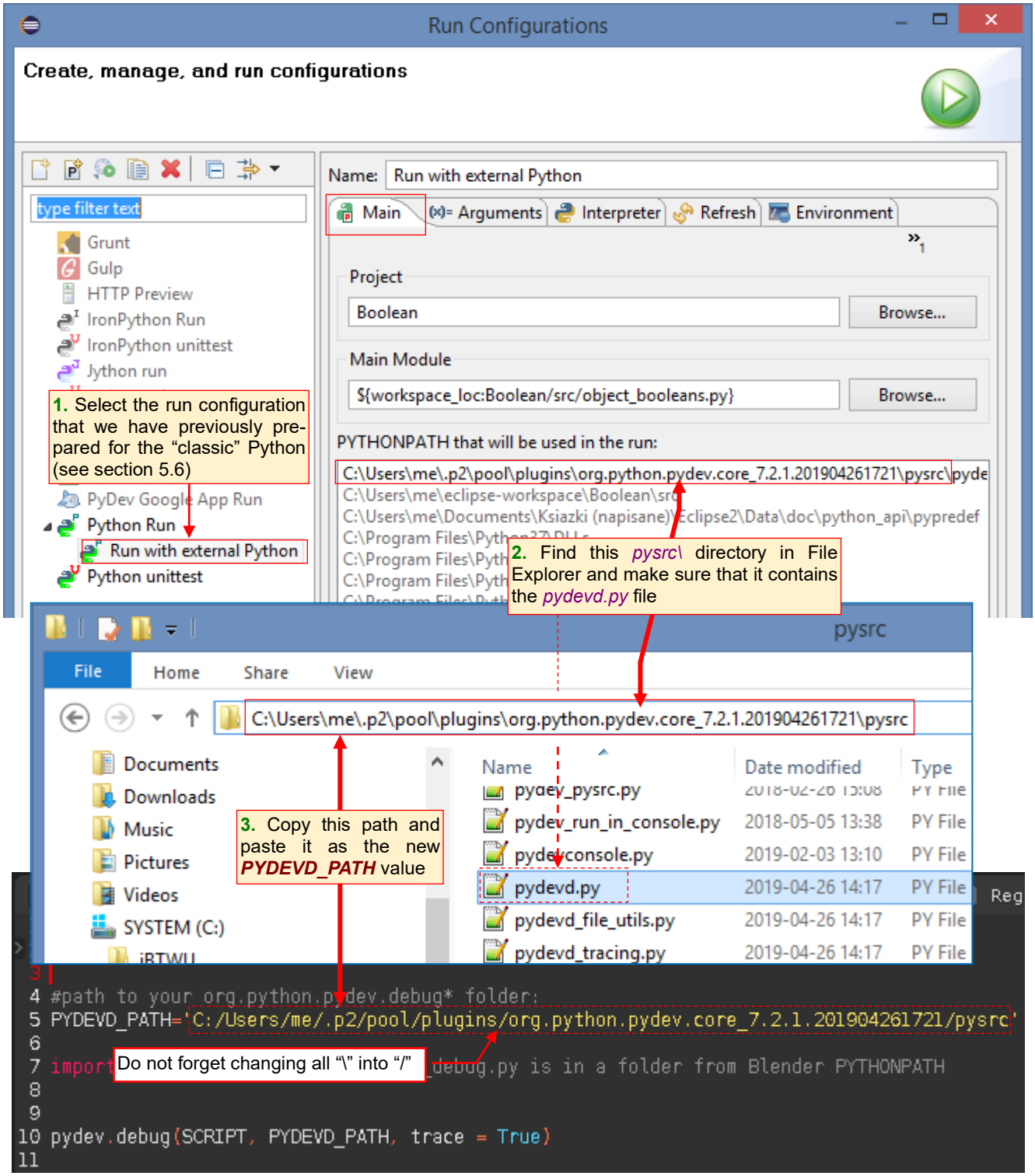


Figure 6.4.8 Determining the *PYDEVD_PATH*

Unfortunately, you cannot copy directly from the *Run Configurations* properties the full path of the *pyscr* directory. You have to open File Explorer and manually "walk along" the path displayed in Eclipse. Make sure, that the final *pysrc* directory contains the file named *pydevd.py*. (This is the remote debugger client module, used by the *pydev_debug.py* script). If so – copy the full path from the File Explorer address field and paste it as the new ***PYDEVD_PATH*** value. Then change in this string all backslashes ("\") into slashes ("/").

Before debugging a script, set in its code at least one breakpoint, because otherwise it will be executed from the start to the end without stopping in the debugger. If you wish to trace your code from the very beginning, set the breakpoint in the line that imports the ***bpy*** module (Figure 6.4.9):

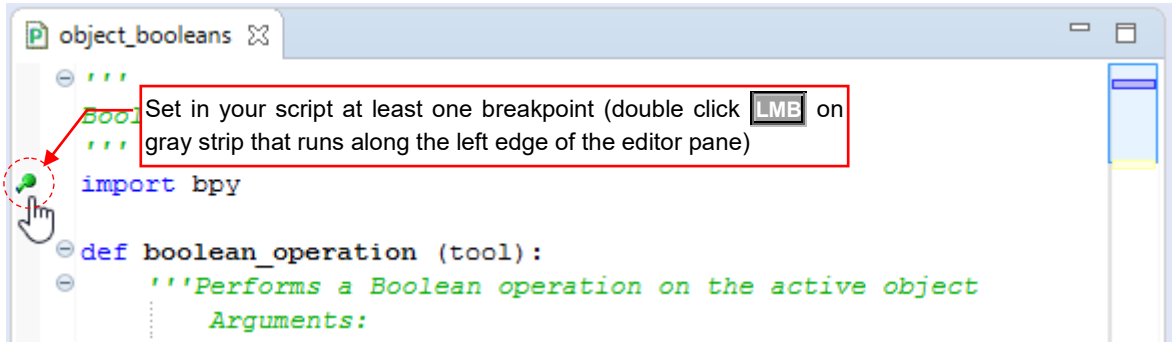


Figure 6.4.9 Placing the breakpoint (at the beginning of the script code)

Then go to the *Debug* perspective and run the PyDev remote debugger server, so it starts listening eventual requests sent via local network (Figure 6.4.10):

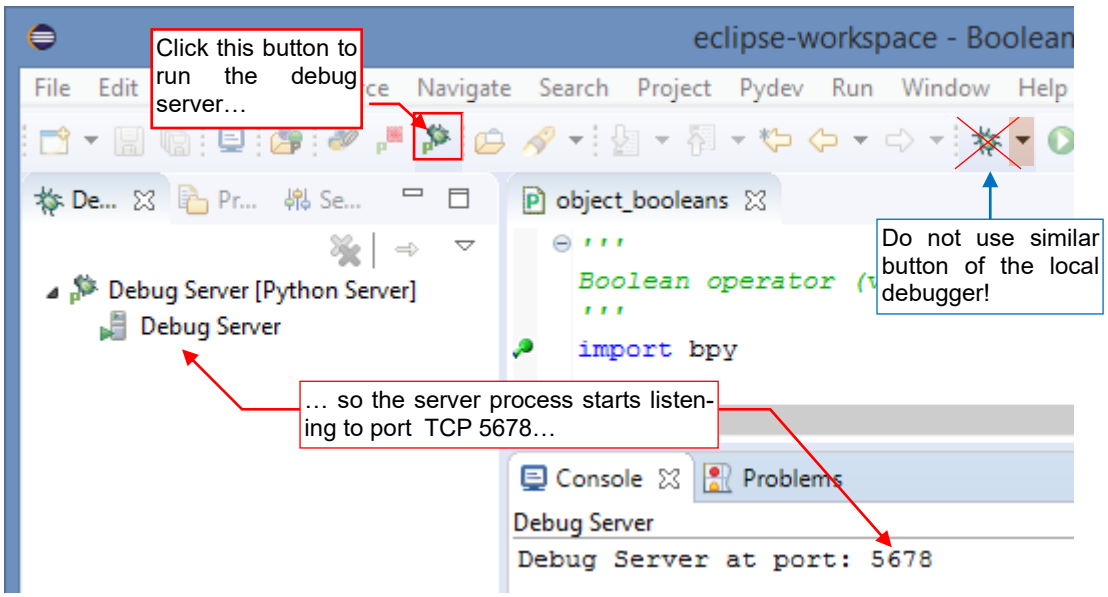


Figure 6.4.10 Starting the PyDev remote debugger server

You can do it using the ***PyDev→Start Debug Server*** menu command (see page 150) or by clicking the "bug" toolbar button with small "P" letter (Figure 6.4.10). Just do not click by mistake the larger "bug" button of the local debugger! (Note that its icon is larger, without any letter).

When the server is listening, you can run the client process. It is invoked by the customized version of the *Run.py* script (I described its modification on previous pages of this section):

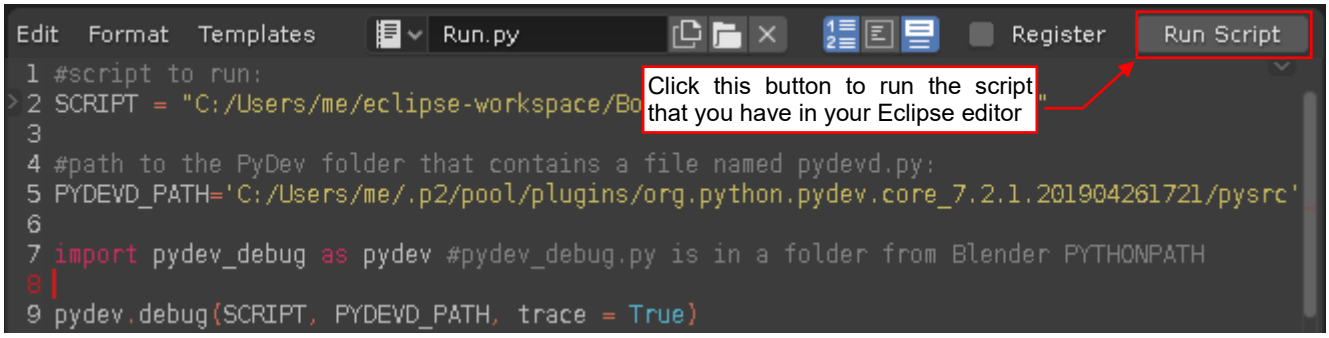


Figure 6.4.11 Starting the script to be debugged (with the remote debugger client)

Just click the ***Run Script*** button in this Blender window (you can find it on the right side of the window header).

Run.py loads the script located at given path (the path in the ***SCRIPT*** constant) and executes its main code. While this code is running, Blender window is “frozen”.

But just click into Eclipse, to activate its window! After a few seconds you will see the debugger execution line at the first breakpoint (Figure 6.4.12):

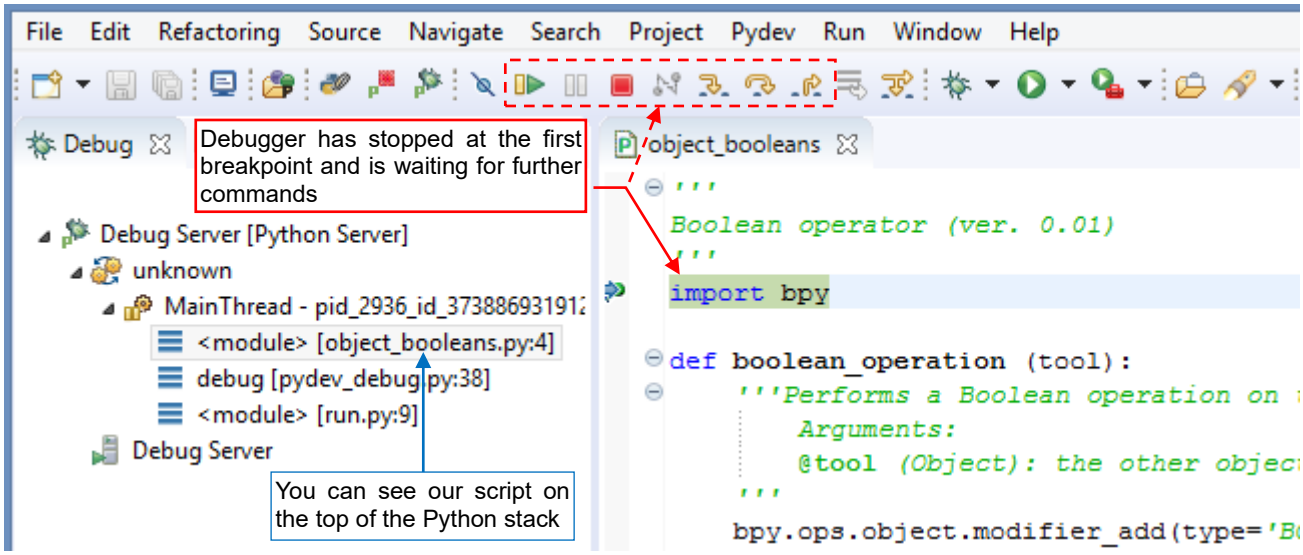


Figure 6.4.12 The first breakpoint of the debug session

On the left side of the script editor window you can see the *Debug* panel. It displays the current state of the Python call stack. In the figure above you can see that the execution started in *run.py* module (this is our code in Blender *Text Editor*). At its line 9 it calls the ***debug()*** procedure from *pydev_debug.py*. It loads (in its line 38) the script *object_booleans.py*, which you can see in the Eclipse text editor. At this moment it waits in the breakpoint at its line 4. Such information can be useful when you are building a solution from several Python files.

While debugging the script, you will frequently check the current state of its variables. For this purpose, PyDev provides the ***Variables*** pane (Figure 6.4.13):

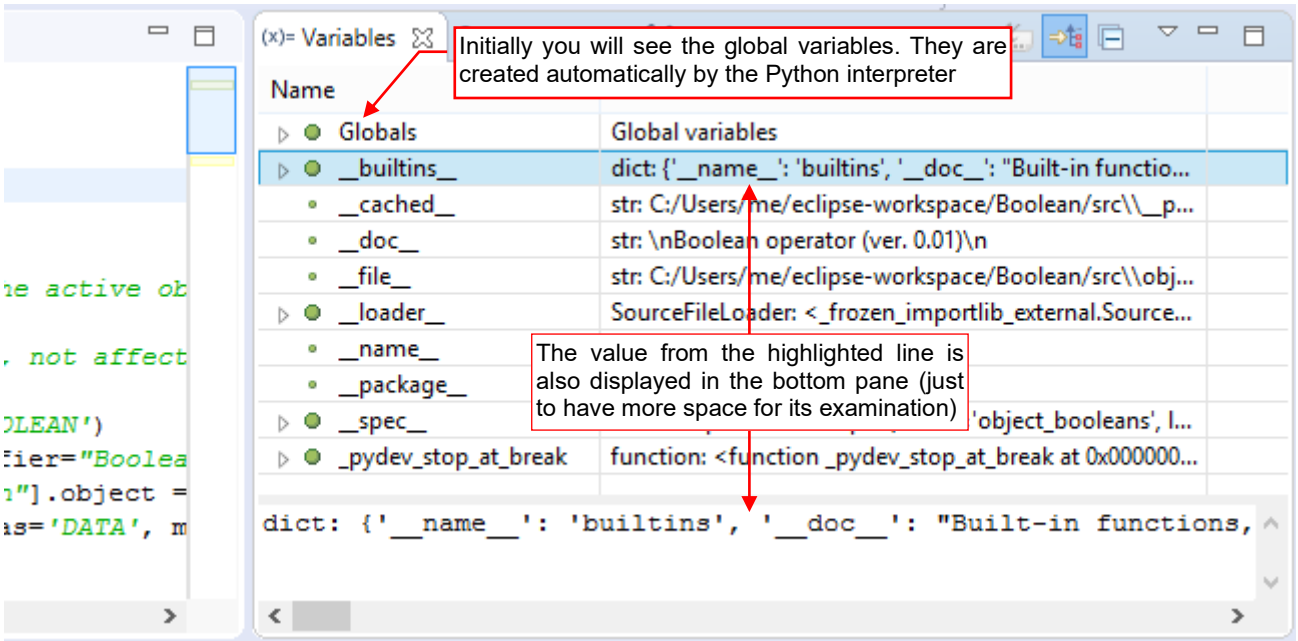


Figure 6.4.13 The *Variables* panel

Variables panel is divided into the list with names and values of the global and local variables, and the details area. In the details area PyDev repeats the value of the variable highlighted on the list. I think that it can useful for checking longer string values or lists.

When the variable contains a list or object reference, Eclipse displays a triangle (▷) at its name. You can click it, to expand the list of its members (Figure 6.4.14):

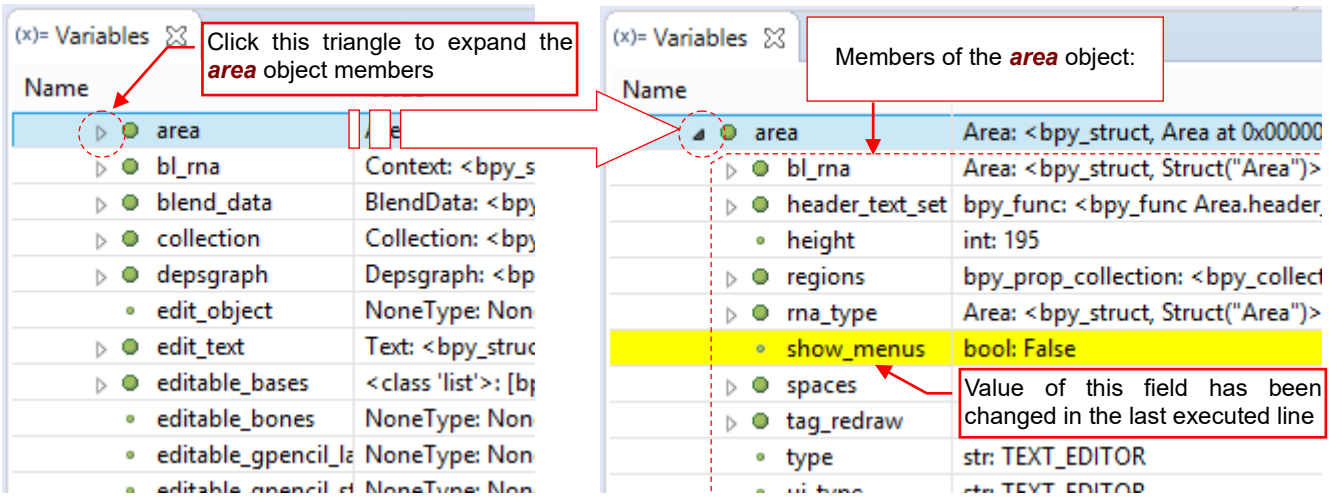


Figure 6.4.14 Browsing the contents of an object

The fields of a class can contain references to other objects (larger green dots). When their values are of one of the basic Python types (***str***, ***bool***, ***int***, …), they are marked with smaller dots. PyDev highlights in yellow the fields/variables that have been changed in the last executed code line.

In the *Variables* window you can also alter the value of a variable. Usually you will simply type it in the *Value* column (Figure 6.4.15):

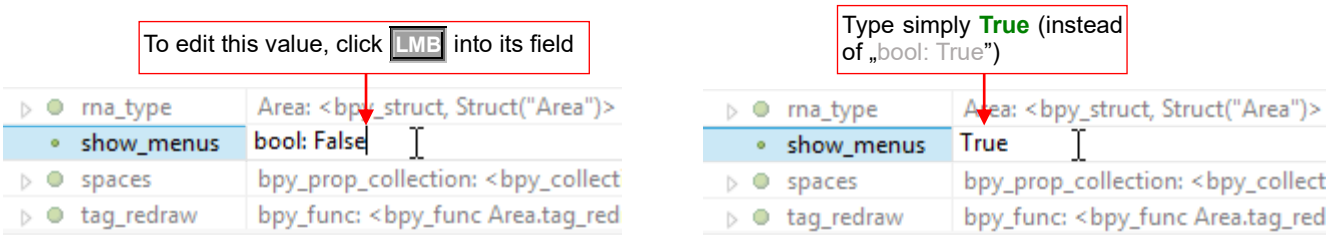


Figure 6.4.15 Altering the variable value

You can also change them in the detail area (using the ***Assign Value*** command from its context menu). Enter the new values in the native Python syntax: **True**, **False**, **1**, **text**, … Do not mimic the "type prefix" ("bool: True") you can see in the unaltered variable fields. After executing the current line, PyDev will also highlight in yellow the variables that you have changed manually.

The ***Expressions*** panel is more convenient for tracking the value of a single object field. You can add it to the current perspective using the ***Window→Show View→Expressions*** command (Figure 6.4.16):

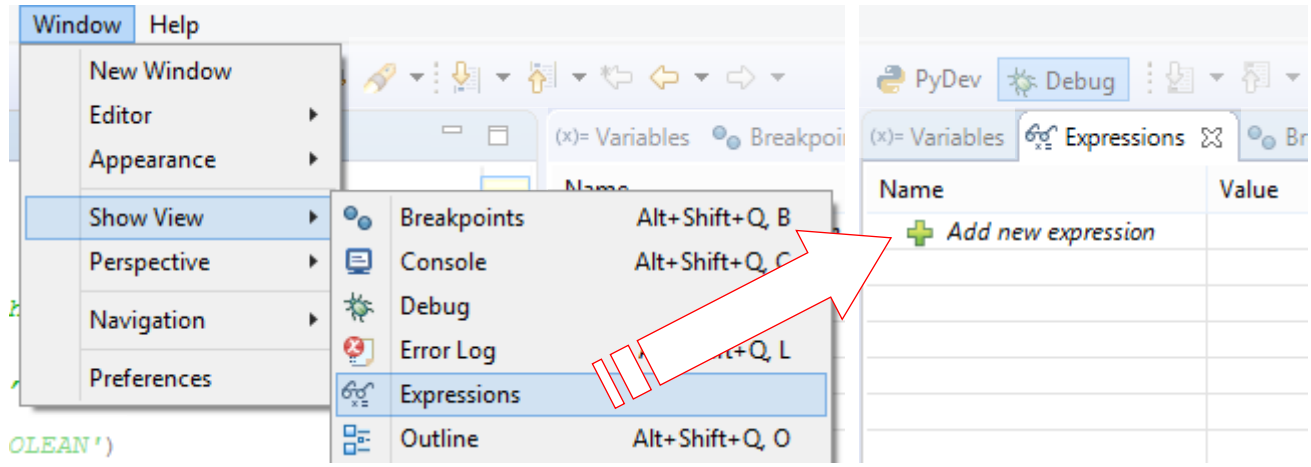


Figure 6.4.16 Adding the *Expressions* panel

The *Expressions* pane layout is similar to the layout of the *Variables* pane: it contains the list of the expressions and their current values. There is also the detail area, showing in a larger field the value of highlighted list item. Unlike in the *Variables* pane, *Expressions* allow you to evaluate any Python expression, at every step of the script execution (Figure 6.4.17):

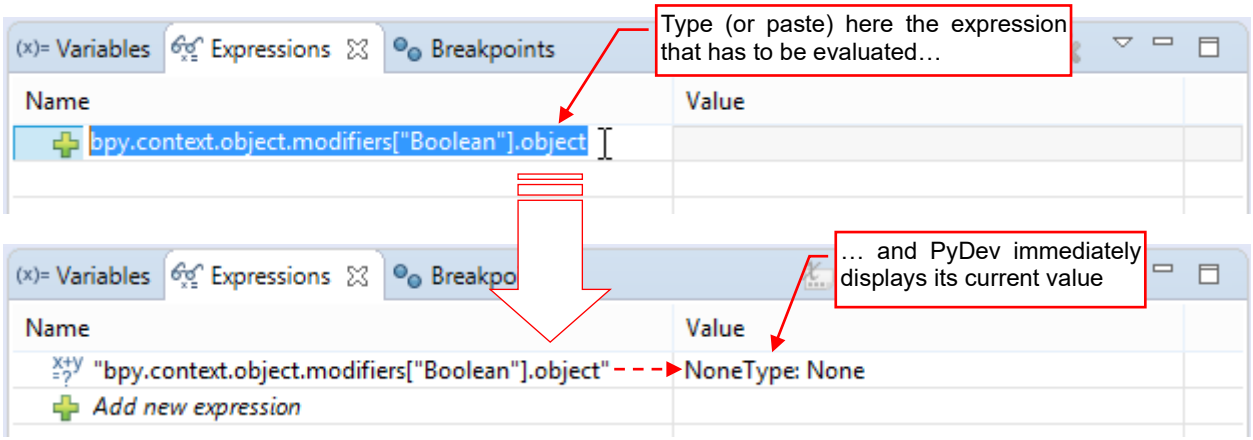


Figure 6.4.17 Adding new items to the *Expressions* list

In the *Expressions* pane you can simply enter the variable name. However, it is more useful for tracking the selected fields of an object. In the example above I am evaluating the ***object*** field of the object modifier named **Boolean**. (At this moment this modifier belongs to the active object). I do it this way, because the ***modifiers*** field returns an iterator instead a list, so you cannot examine its content in the *Variables* pane. (You can find the ***bpy.context*** object among the global variables, see under *Globals→'bpy'→context*. See yourself, what you can do with its ***modifiers***). Unlike in the *Variables* pane, you cannot edit the *Expressions* values.

- The *Expressions* pane is useful for examining the iterators contents, and other objects that you cannot access via the *Variables* pane. In particular, it applies to all the Blender API lists.

The quickest way to browse iterator contents is the conversion into the classic Python list, using the standard ***list()*** function (Figure 6.4.21):

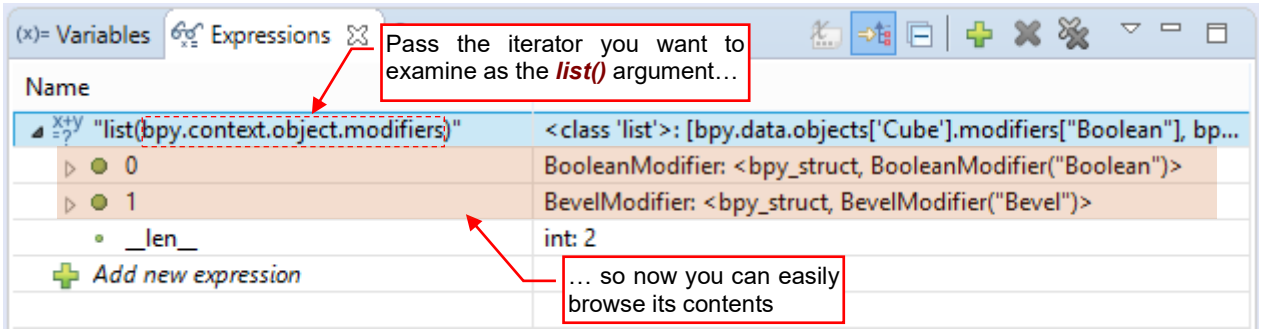


Figure 6.4.18 Browsing in the *Expressions* pane details of the modifiers stack

Of course, do not do it for a very long list. If you are not sure how many elements are in an iterator, you can check it earlier, using the standard ***len()*** function.

In the *Expression* pane, as in *Variables*, you also find the triangle (▷) on the left side of each complex data type. For example - you can use it to examine the contents of the objects that are returned by the iterator (Figure 6.4.19):

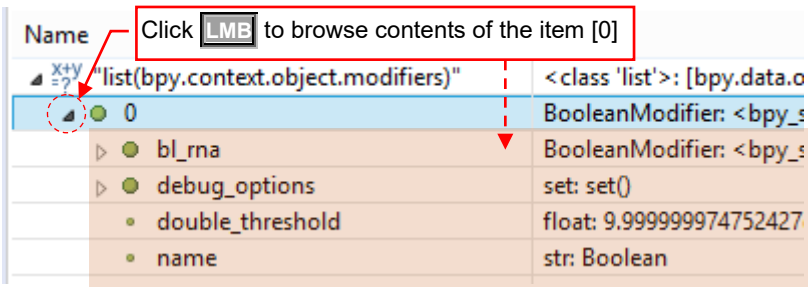


Figure 6.4.19 Examining details of an expression value

Another useful tool for the script debugging is the Eclipse *Console* panel. While debugging, it receives the output from the Blender system console (see ***Windows→Toggle system console*** command in Blender menu). What's more, it becomes interactive while the script is running. You can invoke there any command that will be executed by Blender Python interpreter (Figure 6.4.20):

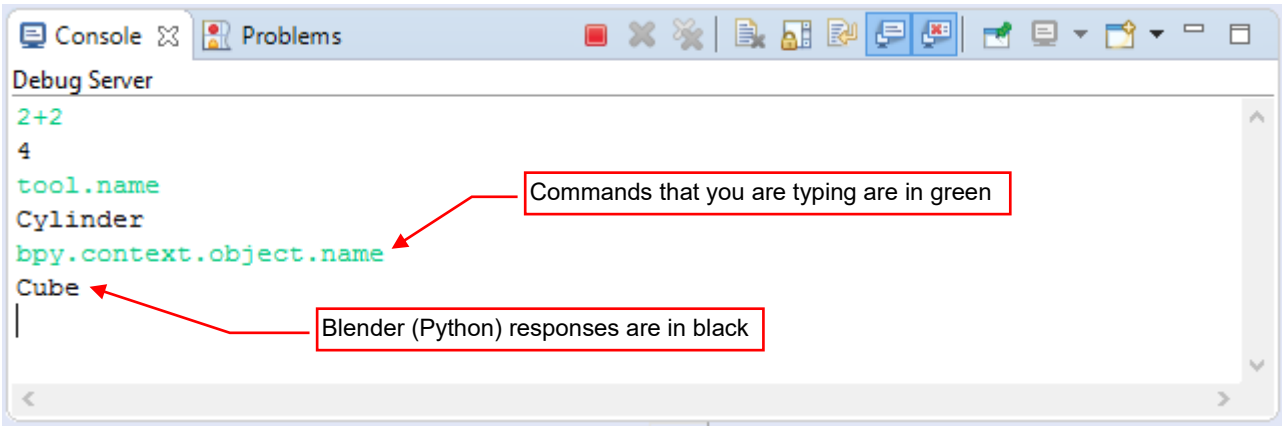


Figure 6.4.20 Eclipse console while debugging a Blender API script

Of course, you can get the same information using the *Expressions* pane. However, in this console you can do more – for example, call a method.

- Although the Blender screen is "frozen" while the API script is running and looks like it was at the moment you have clicked the *Run Script* button (see page 153), you can still control it using the Eclipse console. For example - you can use any Blender command, invoking in the Eclipse console corresponding operator (one of the ***bpy.ops*** methods).

When your script calls the ***print()*** method, then in the debug session you can see its results in both: Blender and Eclipse console (Figure 6.4.21):

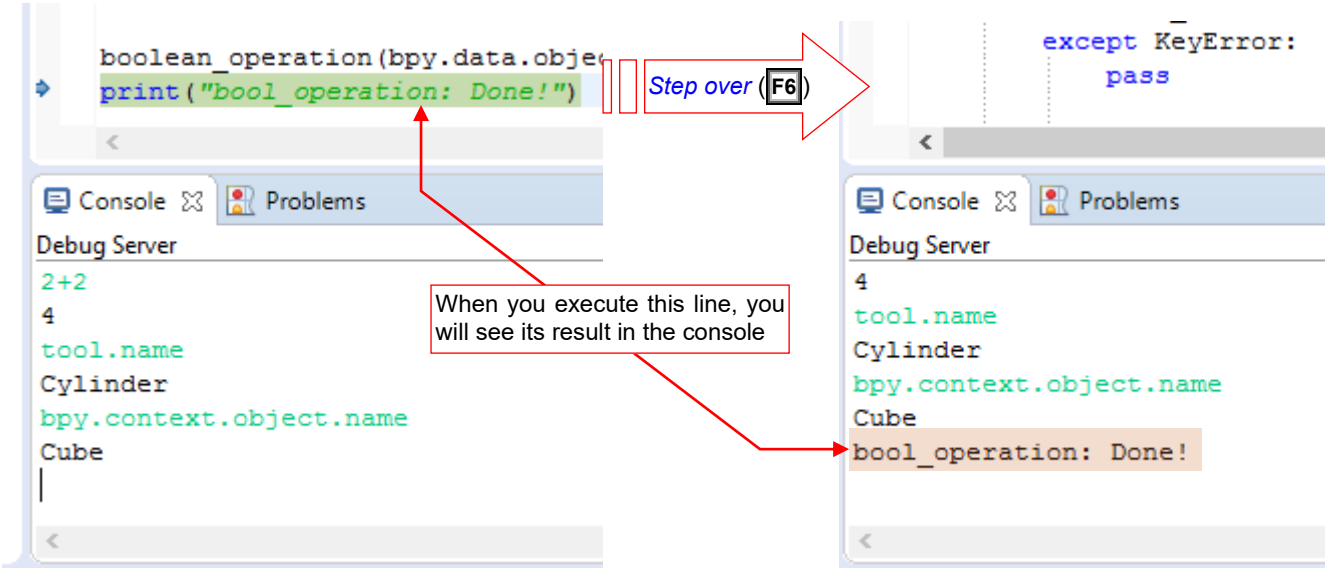


Figure 6.4.21 The results of the *print()* method in the Eclipse console

Also, when you encounter a runtime error (exception) in your script – the Eclipse console will display the same complete information (Python stack traceback, error message) as it appears in the Blender console.

After executing the last line of your Blender script, the remote debugger steps into an auxiliary file that has loaded and run your **.py* file (Figure 6.4.22):

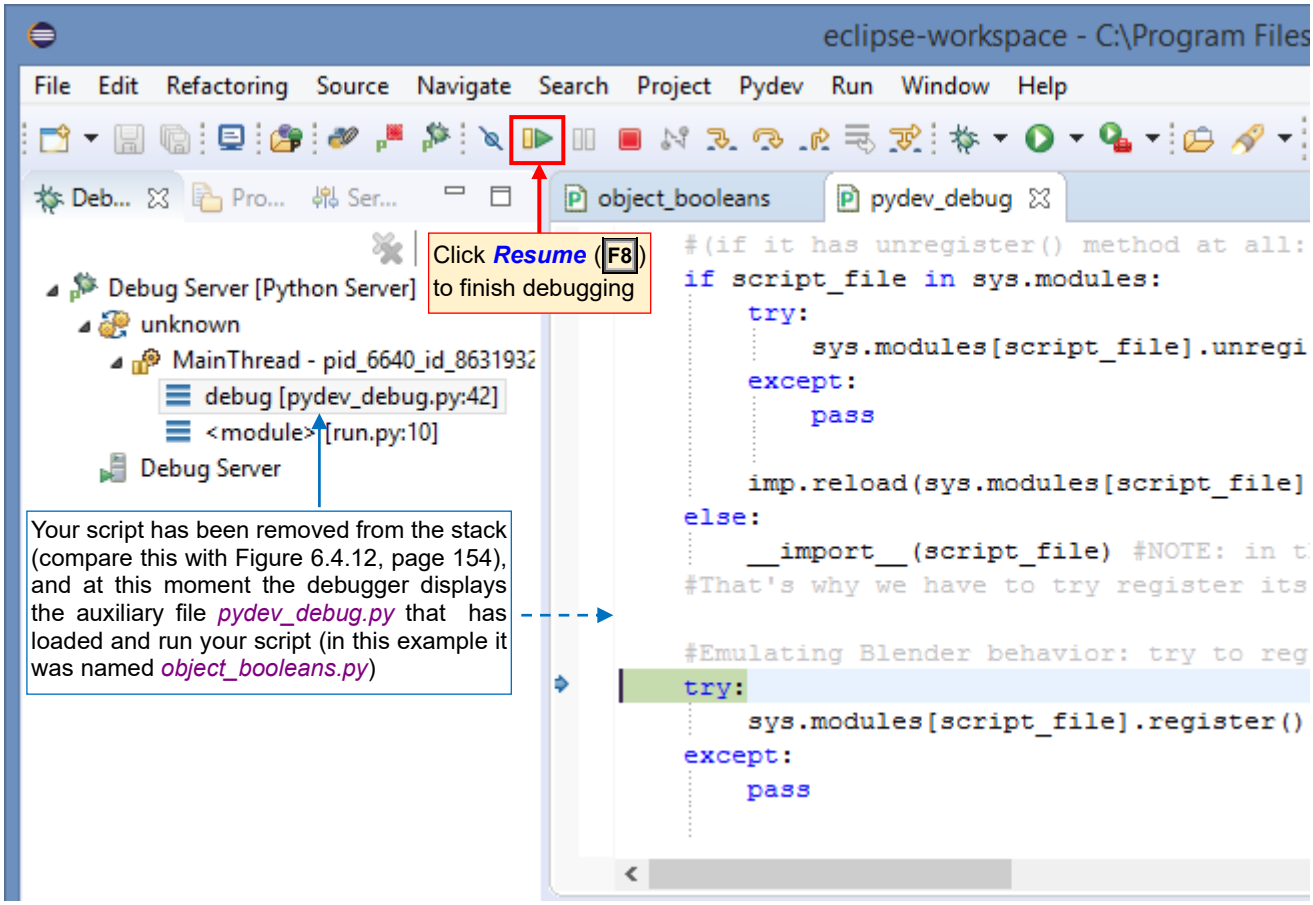


Figure 6.4.22 Debugger screen after executing the last line of your script

On the first run it will be the *pydev_debug.py* auxiliary file, on the next – the other, standard Python module responsible for module import ("reloading"). Anyway, there is nothing to do here – so click the ***Resume*** button (or press F8) to finish this debug session.

- Unfortunately, I did not find so far any "elegant" solution which would automatically resume execution after the last script line and finish it without opening the next module in the debug stack.

When you do it, Blender screen "unfreezes", and you will see all the changes made by your script to the current scene. If you wish to revert them – just use the *Undo* command (Ctrl-Z).

Now you can change the script code in the Eclipse editor. To run/debug it anew, just click the ***Run Script*** button in Blender (see page 153, Figure 6.4.11).

- You do not need to stop the PyDev remote debug server. (Keep it running all the time, once you have started it – as in page 153, Figure 6.4.10). Let it shut down when you close the Eclipse IDE.

- To reload the modified script in Blender and debug it again, just click the ***Run Script*** button in Blender (as shown on page 153).

Once you have modified the *Run.py* code in the Blender *Text Editor*, as described in this section, you do not need to change it anymore. You can keep this code in a test **.blend* file attached to your Eclipse project (as in page 145). All what you need now is its Blender *Run Script* button. Thus, in your *Scripting* workspace you can minimize the *Text Editor* window with the modified *Run.py* code just to see its header with this button.

Finally, a note for all the Readers who use other languages than English: do not place in your API script any character which ASCII code is higher than 127 (i.e. any character which is encoded as two bytes in the UTF-8 files). In the example below, a single character (“ś” in this case) is enough to block the PyDev debugger. When you try to execute the currently highlighted line, or resume the execution, nothing happens. Instead, you will see a traceback and error message in the Eclipse console (Figure 6.4.23):

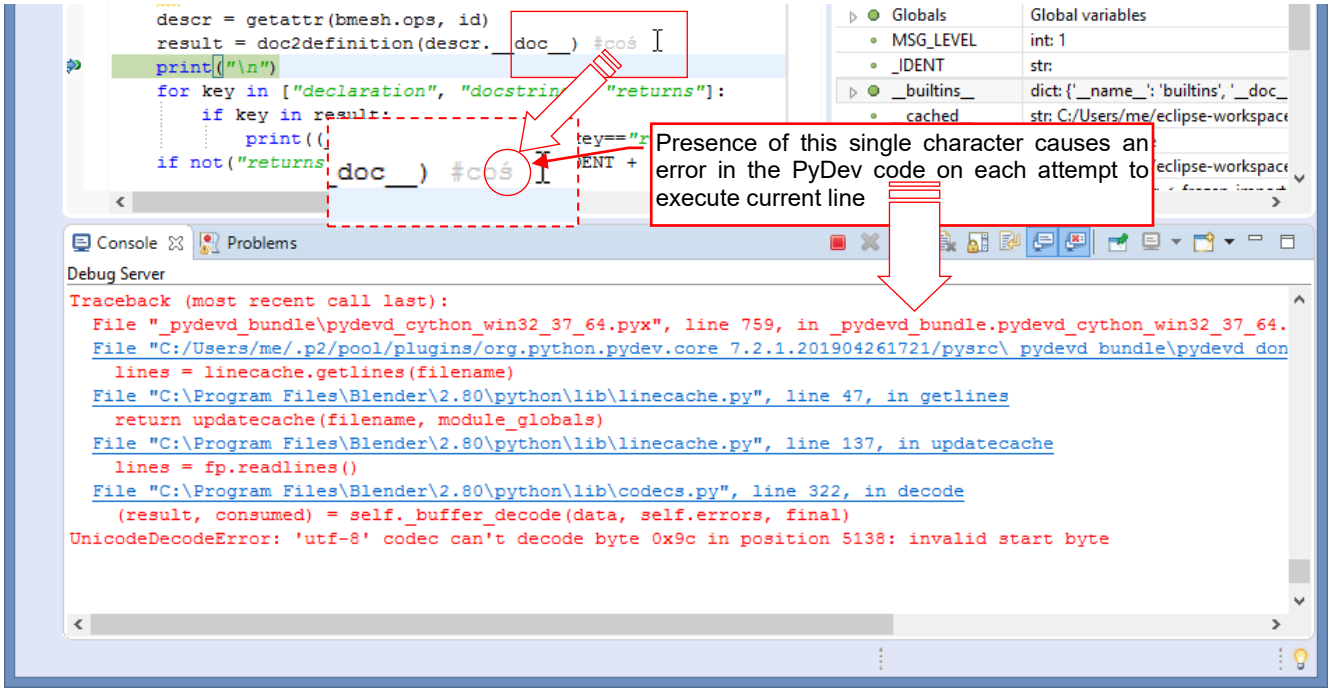


Figure 6.4.23 PyDev error, caused by a national character (of ASCII code > 127)

Fortunately, you can remove/replace such a character in a comment without quitting the current debug session. Just fix it in the Eclipse editor (the same where PyDev debugger is highlighting the current line), then save this script. PyDev automatically updates the saved file in Blender memory, and everything will start working properly: *Step Over*, *Step Into*, and all others debug commands.

6.5 What does contain the *pydev_debug.py* module?

In principle, for tracking script execution in the PyDev remote debugger you have to add to your code just two following lines (Figure 6.5.1):

```
    import pydevd
    pydevd.settrace()  #<-- debugger stops at the next statement
```

Figure 6.5.1 The code that loads and activates the PyDev remote debugger client

Of course, to have this code worked, you should add to the current **PYTHONPATH** the ***pydevd*** package folder, before. Besides, this is just the first point from a longer "to do list" for such an initialization. Hence, these two lines were expanded to a procedure named ***debug()*** (Figure 6.5.2):

```
'''Utility to run Blender scripts and addons in Eclipse PyDev debugger
Place this file somwhere in a folder that exists on Blender sys.path
(You can check its content in the Blender 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
```

Preparation of the received paths, updating of the ***PYTHONPATH***

Starts the debugger client

Emulation of the Blender add-on handling: unregistering the previous version

Execution of the user script

Emulation of the Blender add-on handling: registering the current version

Figure 6.5.2 The *pydev_debug.py* script

I decided to separate the main startup code that runs the Eclipse script inside Blender into the *pydev_debug.py* module. This module contains just single procedure: ***debug()*** (Figure 6.5.2). Such a solution allowed for maximum simplification of the *Run.py* code — the script template, which you have to update for every new project (see page 53).

- Place the *pydev_debug.py* module in the directory, which is present in the Blender Python path (i.e. in one of directories listed in the content of ***sys.path***). In Windows one of them is the folder that contains the *blender.exe* file (see page 39, Figure 3.2.2), but it may be different in the Linux or Mac environments. Just check your ***sys.path*** it in the Blender *Python Console*.

The whole *Run.py* code contains just a call to the ***debug()*** procedure, with following arguments:

- ***script***: path to the script file that has to be executed;
- ***pydev_path***: path to *pydevd.py* module (this is the PyDev remote debugger client);
- ***trace***: optional. Set this named argument to *True*, when the script has to be traced in the debuuger. Set it to *False* when you want just to run the script without any break. (When ***trace*** = *False*, you can run this code without Eclipse — see page 149);

Notice (Figure 6.5.2) that the ***debug()*** procedure loads user's ***script*** module using the *import* statement. It allows for debugging Blender *add-ons*[1]. Before this import, ***debug()*** attempts to handle the previously loaded module as the *add-on*, and to unregister it. If this attempt fails — no error is signaled (not every script has to be a plugin). When the new script is loaded, ***debug()*** tries to register it as a new *add-on*.

- When you write a Blender add-on script, at the beginning implement the required ***register()*** and ***unregister()*** methods. They will allow for properly handling of the Blender registration process, every time you will click the ***Run Script*** button (see page 55).

[1] Blender loads the **.py* file of an add-on and invokes the ***register()*** procedure. (It is supposed that this required module method will register all the API classes of this plugin). Consequently, Blender calls the ***unregister()*** method when the user turns the plugin off. While the add-on is active, Blender creates the instances of the registered add-on classes, when they are needed. The *pydev_debug.py* script emulates this behavior, de-activating, reloading, and then activating again your add-on when you click the *Run Script* button. However, it cannot help you in the debugging of an installed Blender add-on. (Thais means an add-on that was copied into Blender addon directory and is visible in the *Blender Preferences* window). You need to install your addon just to test the eventual plugin preferences panel (see page 99). Most of the Blender addons do not need such a feature.

6.6 The full code of the *object_booleans.py* add-on

In subsequent chapters of this guide I have gradually created the complete *object_booleans.py* add-on. The fragments of its code are dispersed everywhere in this book. However, after so many modifications it is useful to present the final result in "one piece". If you want to copy this text to the clipboard — beware of the tab spacing! They are all removed, when you copy the code below directly from this PDF document. It is better to download this script file from my page.

The script does not fit into a single page, so I decided to divide it into five parts. The first part is a header that contains the GPL information and the auxiliary debugging statements, which are useful for testing the preferences panel (Figure 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 convenience (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 ----------------------------------------------------------
```

Continued on the next page…

Figure 6.6.1 The *object_booleans.py* script, part 1 (the declarations)

The next part contains the main code, which implements the core operation (Figure 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 the 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 processed 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 += "occurred for object(s): '%s'" % active.name
        if 'tool' in locals(): cntx_msg += ", '%s'" %tool.name
        return [ERROR, "%s %s" % (str(err),cntx_msg)]
```

Continued on the next page…

Figure 6.6.2 The *object_booleans.py* script, part 2 (core code)

The next part contains the required API “framework”: the operator class. There is also another API class that implements the pie menu (Figure 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")
```

Continued on the next page…

Figure 6.6.3 The *object_booleans.py* script, part 3 (API classes)

In this fourth part you can find the implementation of the addon preferences and the procedures that add and remove the keyboard shortcut (Figure 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()
```

Continued on the next page…

Figure 6.6.4 The *object_booleans.py* script, part 4 (implementation of the add-on preferences)

The last part registers the API classes and adds the operator to the *Object* menu (Figure 6.6.5):

```
#---------- ### Register -----------
from bpy.utils import register_class, unregister_class

def menu_draw(self, context):
    #draws the menu item that invokes this operator (actually - a submenu of its options)
    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) #add this operator to the Object menu
    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()
```

Figure 6.6.5 The *object_booleans.py* script, part 5 (add-on registration)

Bibliography

Books

[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

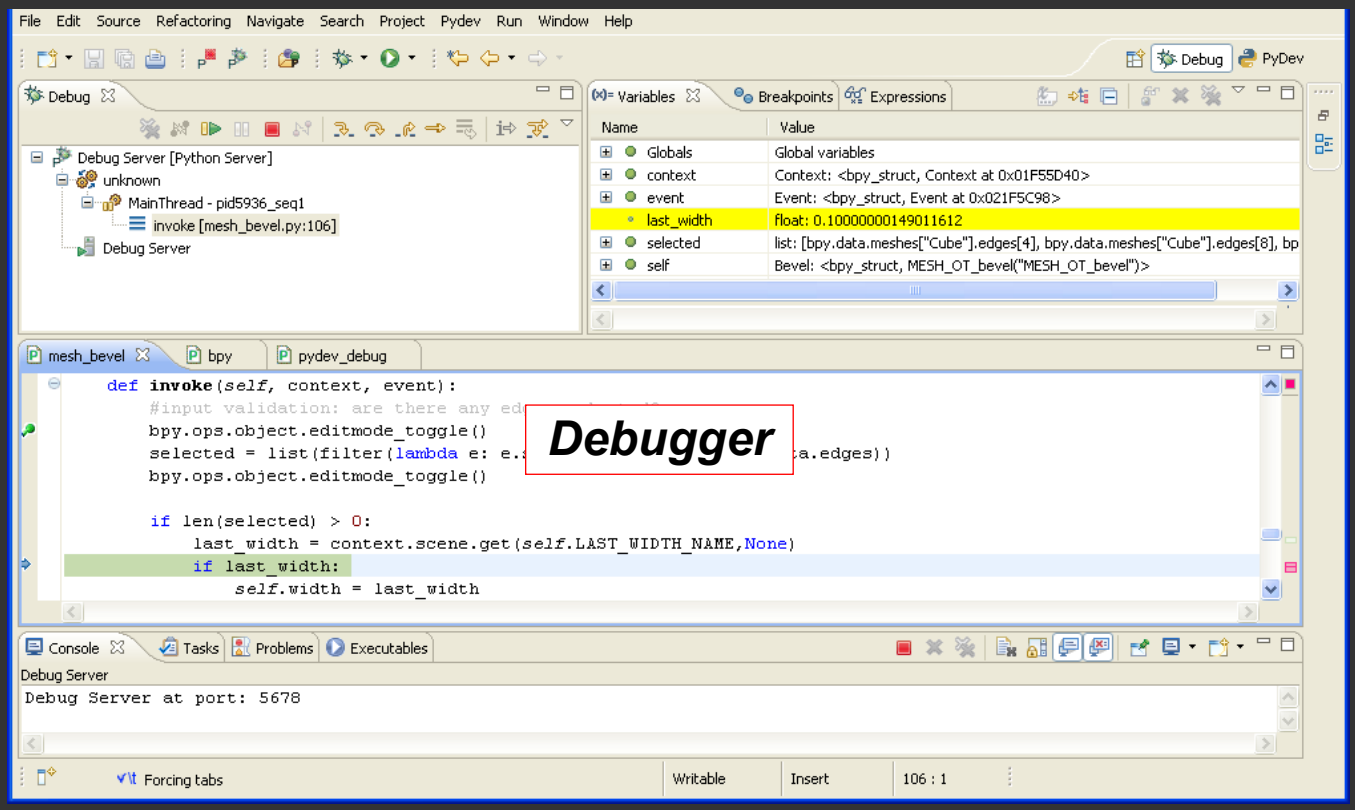
[1] http://www.blender.org

[2] http://www.python.org

[3] http://www.eclipse.org

[4] http://www.pydev.org

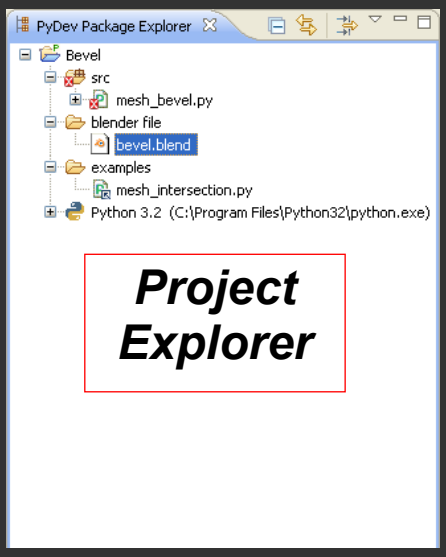
[5] http://wiki.blender.org, in particular http://wiki.blender.org/index.php/Extensions:Py/Scripts



If you already have some programming experience in Python and want to write an add-on for Blender 3D, then this book is for you!

I am showing in this guide how to arrange a convenient development environment for writing Python scripts for Blender. I use Eclipse IDE, enhanced with PyDev plugin. Both elements are the Open Source software. It is a good combination that provides all the tools shown on the illustrations around this text.

The book contains a practical introduction to the Blender Python API. It describes the process of writing a new add-on. I discuss in detail every phase of the implementation, showing not only the tools, but also explaining the methods that I use. These pages will allow you to gain the skill needed to write your own Blender tools.



ISBN: 978-83-941952-1-2

Free electronic publication