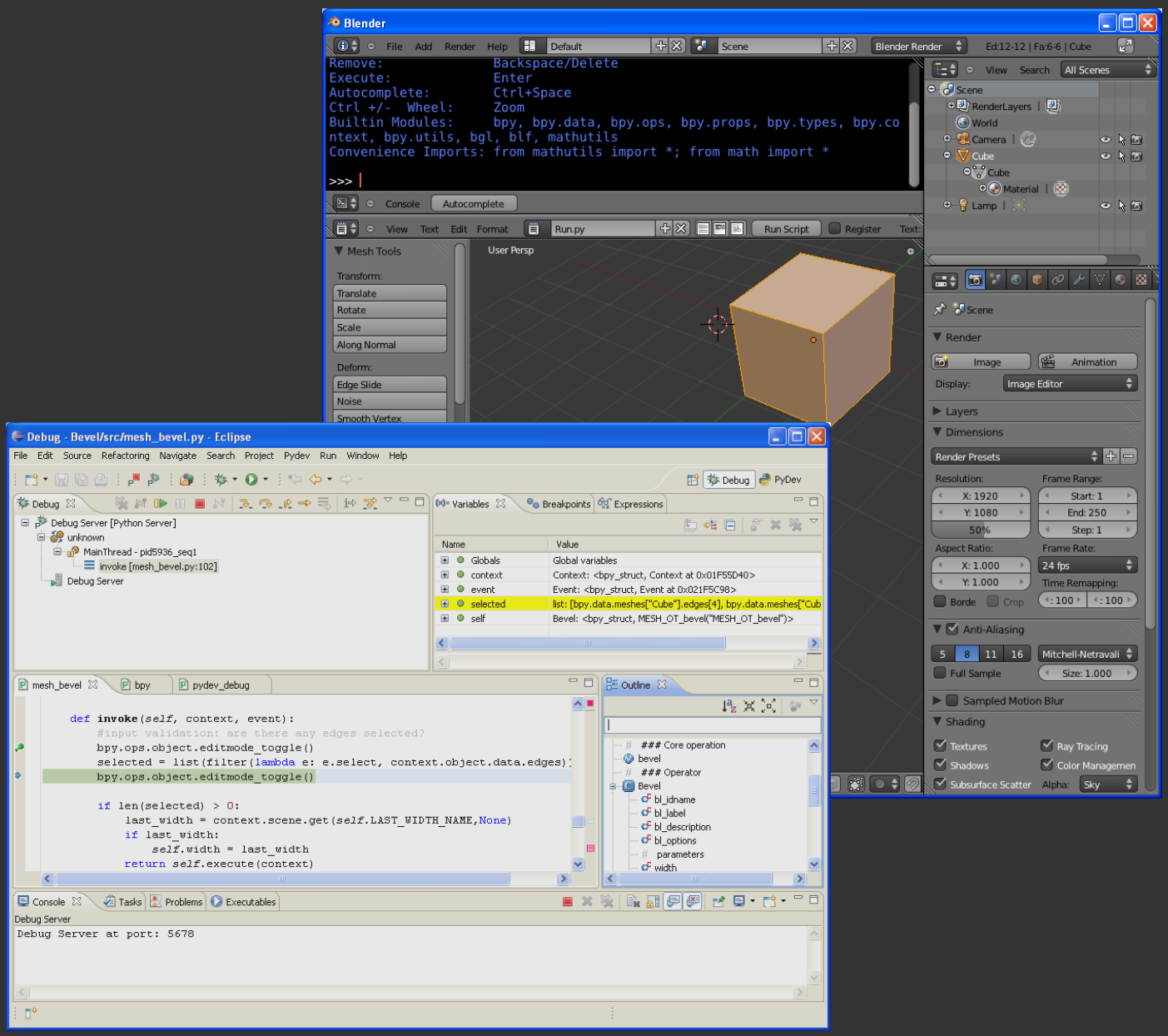
Witold Jaworski



Programming Add-Ons for Blender 2.5

Writing Python Scripts with Eclipse IDE

version 1.01

Programming Add-Ons for Blender 2.5 - version 1.01

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

Reviewed by Jarek Karpiel

I would also to thank Dawid Ośródka and PKHG (from the blenderartists.org forum) for their comments.

ISBN: 978-83-931754-2-0

Table of Contents

Introduction

To extend the standard functionality of Blender with new commands, you can use Python scripts. Many useful add-ons were created this way. Unfortunately, Blender is missing something like an integrated development environment ("IDE") for the script programmers. "In the standard" you will find only the *Text Editor* that highlights the Python syntax, and the *Python Console*. It is enough to create simple scripts, but begins to interfere when you work on larger program. Particularly troublesome is the lack of a "windowed" debugger. In 2007, I wrote an article that proposed to use for this purpose two Open Source programs: **SPE** (the editor) and **Winpdb** (the debugger). Jeffrey Blank published it a few months later on the wiki.blender.org.

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 versions used the older ones (2.x). Python developers also decided to break the backward compatibility between these versions. In result, the GUI library used by SPE and Winpdb — **wxPython** — does not work with Python 3.x. Worse still, no one is working on its update. It seems that the both tools have become unavailable for newer Blender versions.

I decided to propose a new developer environment, based on the Open Source software. This time my choice fell on the **Eclipse** IDE, enriched by the **PyDev** plugin. Both products have been developed for 10 years, and do not depend on any particular Python version. (Internally they are written in Java, thus are not exposed to such problems like SPE and Winpdb, written in Python). To check this solution in practice, I ported all my scripts to the Blender 2.5 Python API using this framework. Some of my add-ons had to be rewritten from scratch. Based on this experience I think that this new environment is better than the previous one.

I think that the best way to present a tool is to show it at work. I have described here the creation of a new Blender command that bevels selected edges of a mesh. (It is a restoration of the "old" ***Bevel*** command from Blender 2.49). 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 (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 (creating your own panels or menus, for example).

Conventions

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

- US keyboard, with 102 keys (you will find also some comments about a non-standard notebook keyboard, like the one used by me);
- 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 every topic with all details!

Preparations

In this section, I am describing how to build (install) an appropriate environment for the programmer (Chapter 1). Then I am introducing you in 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:

- external (“standard”) Python interpreter (required to let PyDev to work properly);
- one of the Eclipse “packages”;
- the Eclipse plugin: PyDev;

This chapter describes how to set them up.

I assume that you already have installed Blender.
(This book was written using Blender 2.57. Of course, you can use any of its later versions).

1.1 Python

Blender comes with its own, embedded Python interpreter. Check its version, first. To do it, switch one of the Blender windows to the ***Python Console***, and read the version number written in the first line (Figure 1.1.1):

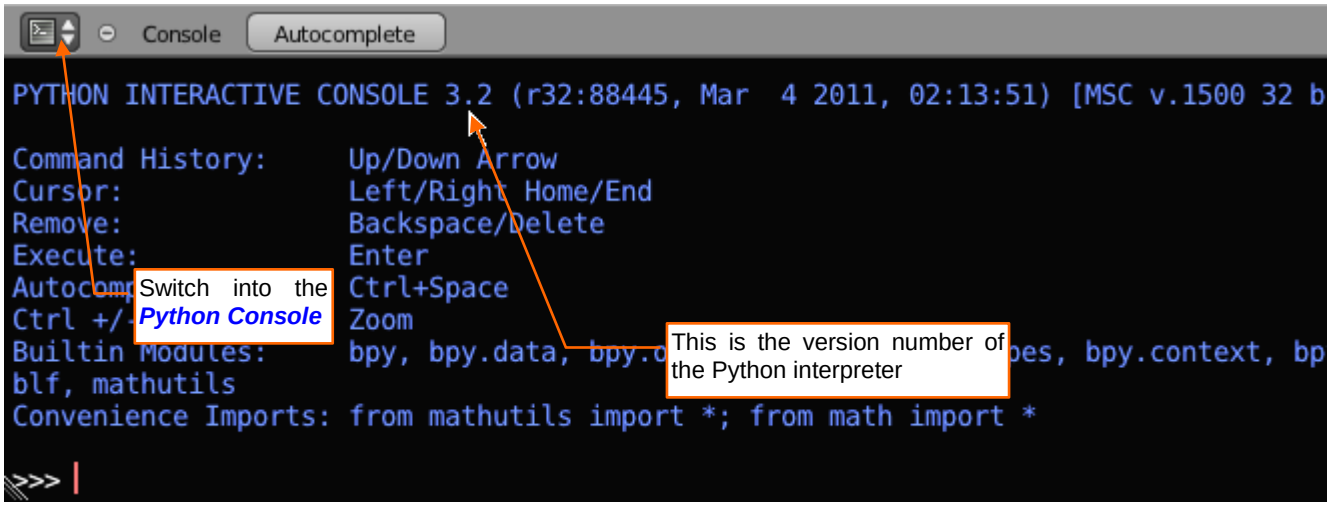


Figure 1.1.1 Reading the version number of the embedded Python interpreter.

Blender on the illustration above uses Python 3.2 (it is Blender 2.57). You should always install the same version of the external Python interpreter, although minor differences in them are not the problem.

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

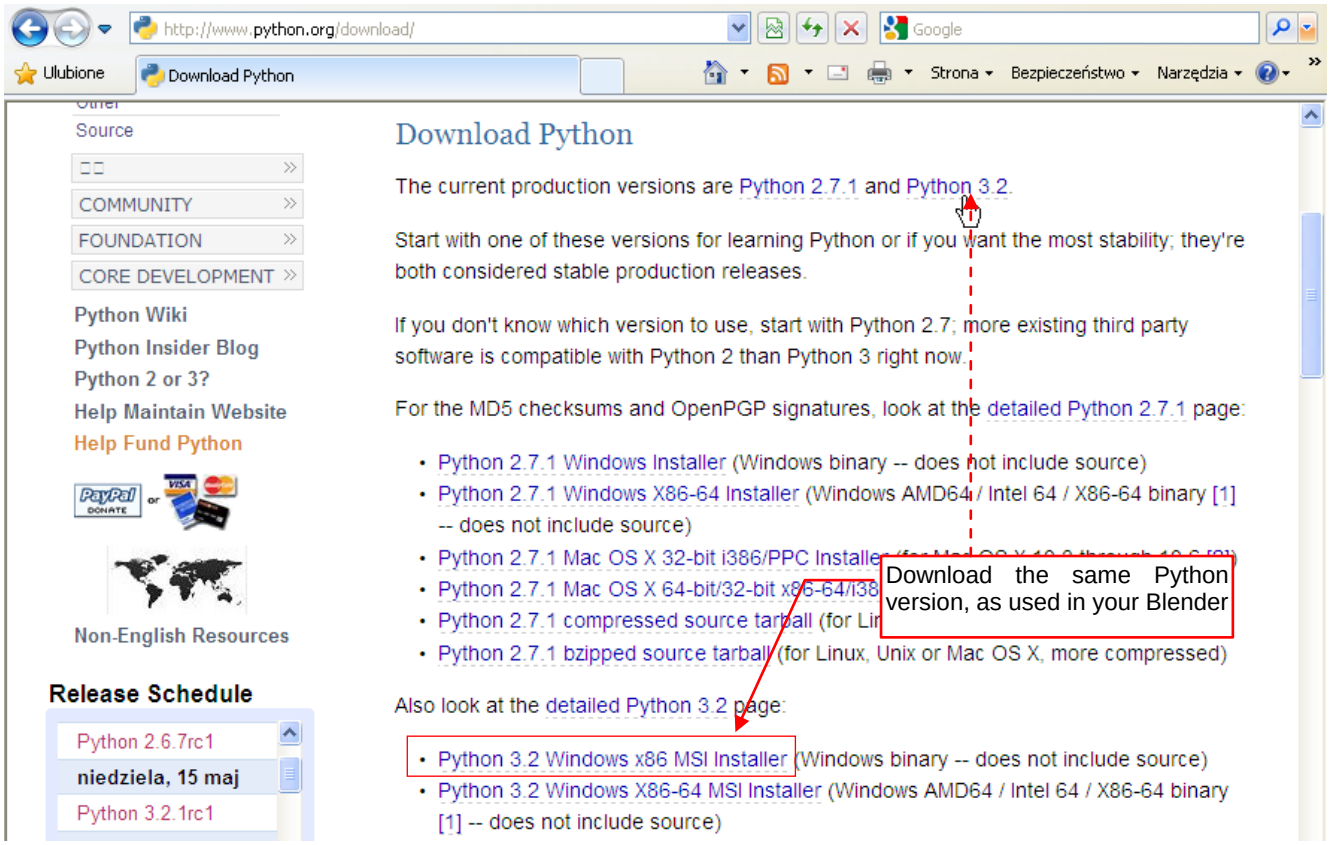


Figure 1.1.2 Downloading the external Python interpreter (from http://www.python.org/download)

As you can see, Python currently has two branches due to the lack of backward compatibility: 2.x and 3.x. We are interested in the latter.

- Before downloading the external Python interpreter, you can also check if you do not have it already installed on your computer. Try to invoke in the command line following program:
  `python --version`
  If you have it installed, it will launch the console, as in Figure 1.1.1. You can read its version number, 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 "after the dot" of the number). Use the newer version, in such case. Blender 2.5 always uses its embedded interpreter, even when the external Python is available in your system. For example, if you use version 3.3 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 3.2. (Practically, the differences between minor Python versions are not big).

Run the downloaded program as required in your operating system. There is just an installer for Windows (Figure 1.1.3). (To install it, you need an account with the Administrator privileges, on your computer):

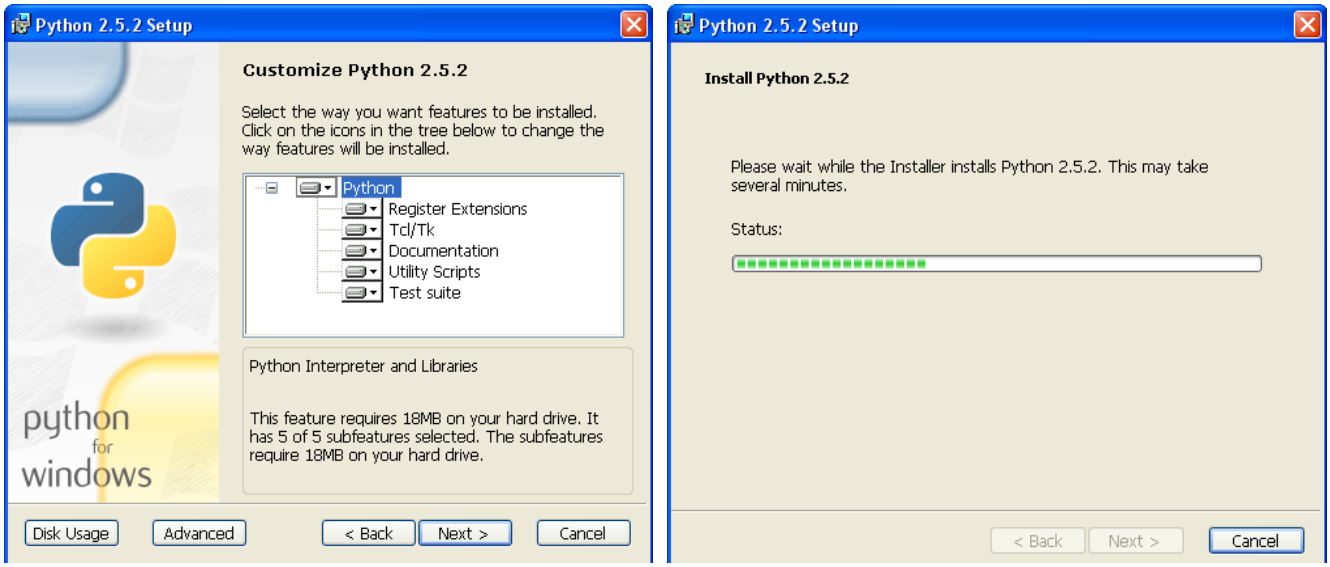


Figure 1.1.3 Selected screens of the Python installer (for Windows)

There is nothing special in this process. Just accept the default settings and keep pressing the ***Next*** button to the end of the installation (detailed description — see section 5.1, page 100).

1.2 Eclipse

Go to the downloads directory on the Eclipse project page (www.eclipse.org/downloads — Figure 1.2.1):

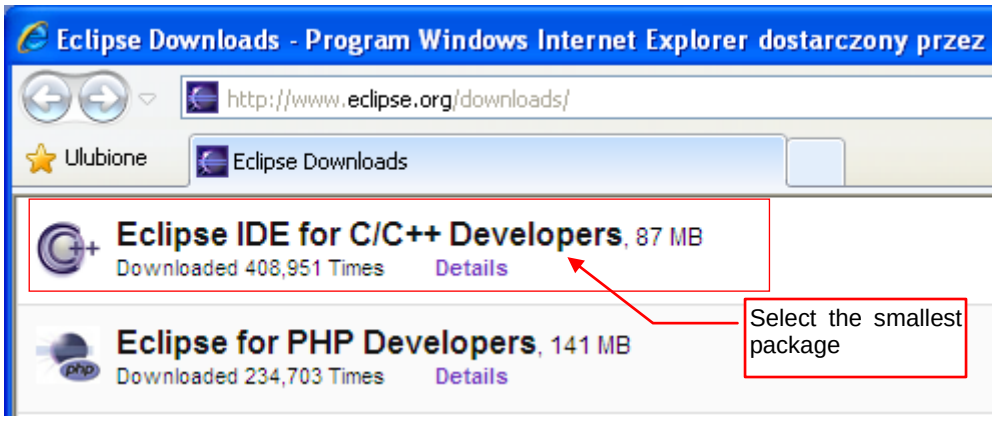


Figure 1.2.1 Downloading an Eclipse package (this screen was captured on March 2011)

When you look at the descriptions on the page, you will notice that Eclipse is available in many different packages. Each of them is prepared for the specific language / programming languages. (You can still create a C ++ program in any other package, for example "Eclipse for PHP Developers". Just add the appropriate plugins)! What you see are just typical packages, prepared "in advance". They correspond to the most common needs. There is no any special “Eclipse for Python” package, so I would propose to download ***Eclipse for Testers*** or ***Eclipse IDE for C/C++***. (Always choose the smallest and least-specific package). The installation details are described on page 103.

Eclipse developers do not prepare any Windows installers. In the downloaded file, you will find a zipped folder with ready-to-use program. Just unpack it — for example, to the *Program Files* (Figure 1.2.2):

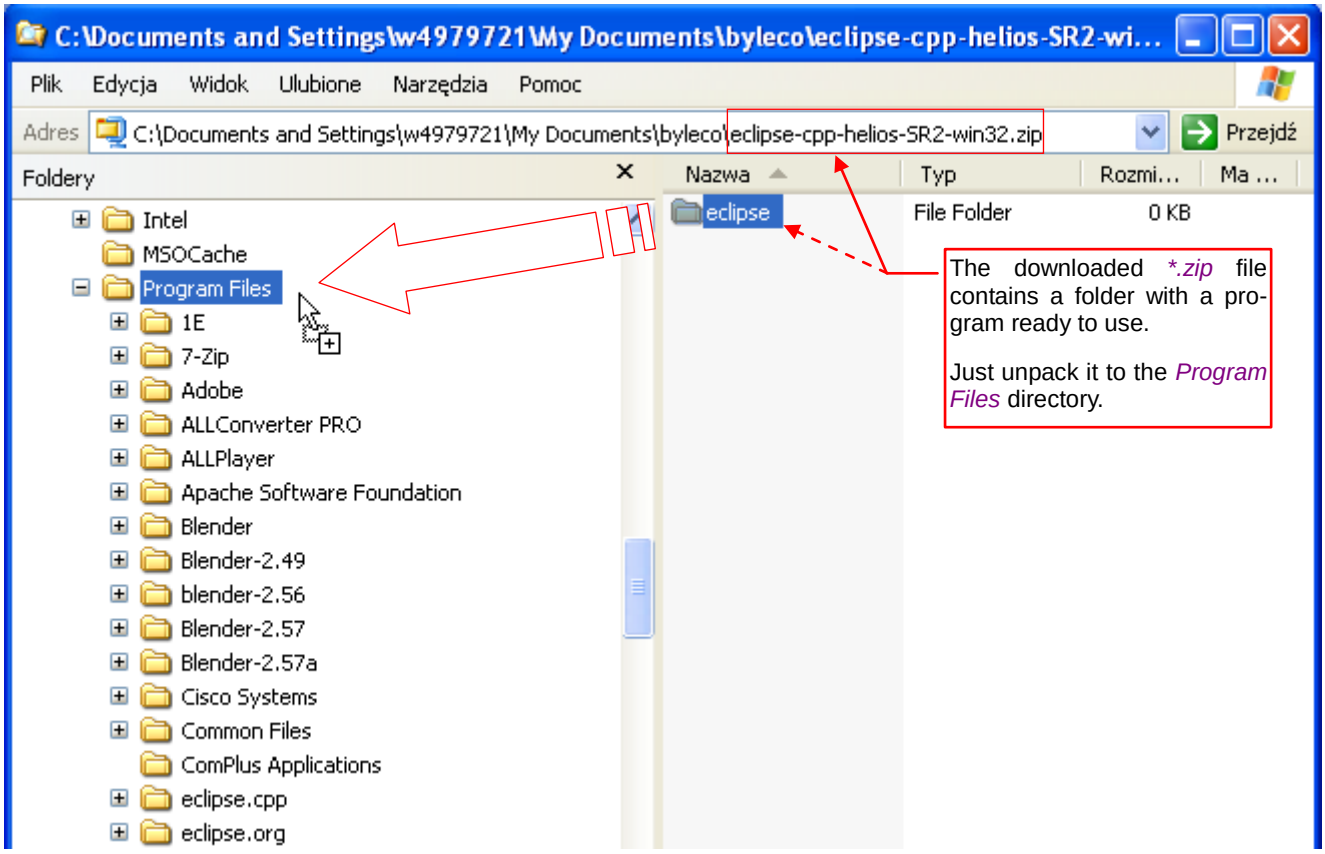


Figure 1.2.2 Eclipse “installation” — just unzipping the downloaded file

The main program is *eclipse\eclipse.exe* (Figure 1.2.3). You can add its shortcut to your desktop or menu.

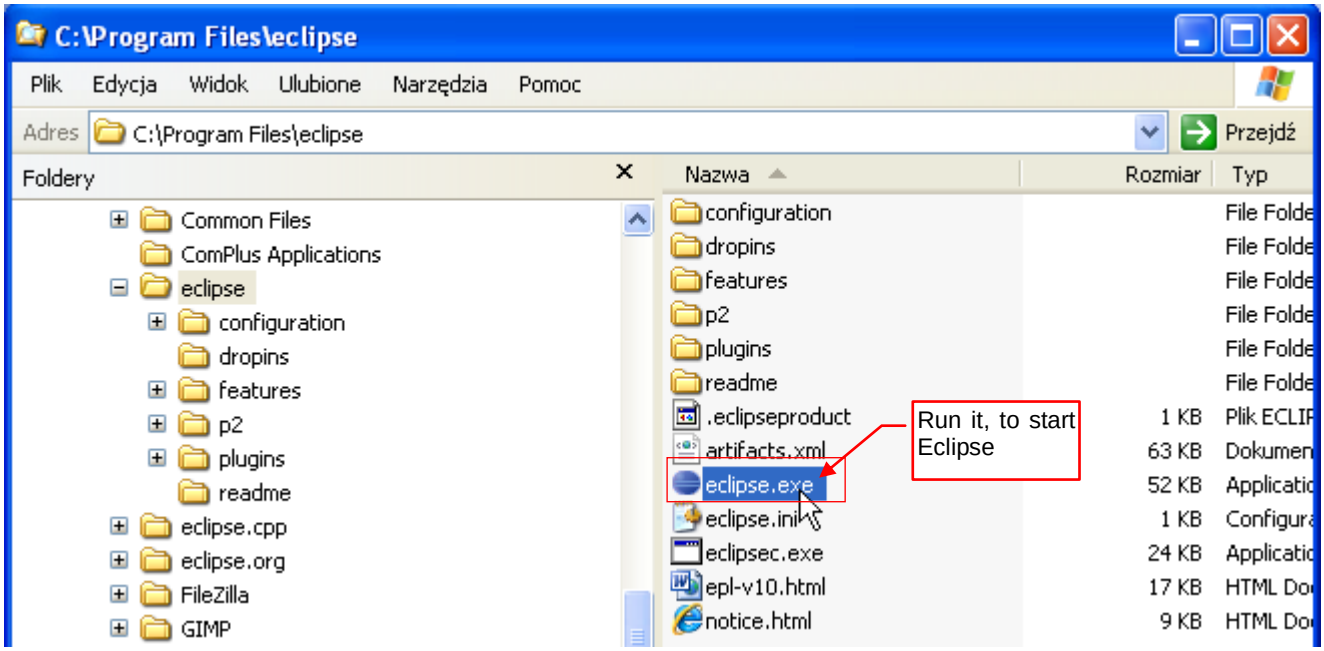


Figure 1.2.3 Running Eclipse (for Windows)

- Eclipse requires the Java Virtual Machine. Most likely, you have it already installed on your computer. If not — download it from the www.java.com (details — page 103).

When you start the *eclipse.exe* program, it displays a dialog where you can specify the folder for future projects. This location is called a "*workspace*" (Figure 1.2.4):

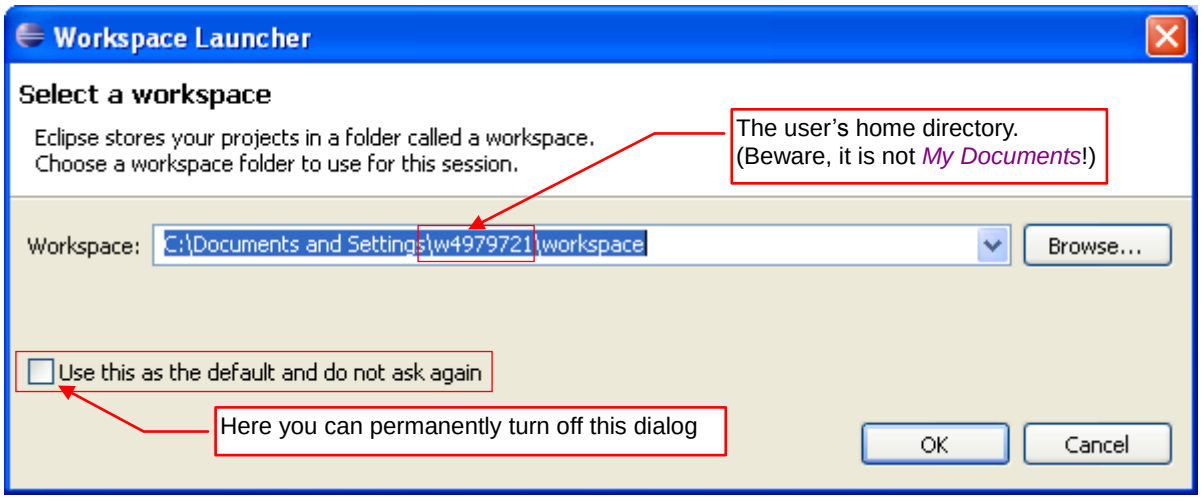


Figure 1.2.4 The question about the actual *workspace*

Eclipse creates a separate subdirectory for each project, here. Such folder contains Eclipse internal data files and your scripts. (You can also place there a shortcut to a script located elsewhere on the disk). Each workspace directory contains, in addition to the projects folders, its own set of Eclipse preferences. It includes configuration of the Python interpreter. The *workspace* folder is created in the user home directory[1]. (In this example it is the directory of user **W4979721**). Usually it is enough to have just one workspace.

[1] Note for the Windows users: *My Documents* folder is also located there. This is just the *Unix/Linux* convention. If you are used to keep all your data in the *My Documents* folder - change the path in *Workspace Launcher* dialog. Eclipse will create a workspace folder at the specified location.

Then the ***Unable to Launch*** message may appear (I have got it on my Eclipse 3.6 — see section 5.2, Figure 5.2.5). Do not worry about it! Eclipse is always trying to open your last project, saved in the folder. Yet, on the first run, there is nothing there. So the program “is surprised”, and displays such a warning.

On the first run, Eclipse displays window with the *Welcome* tab. It contains shortcuts to several Internet sites, related to this environment (Figure 1.2.5):

Figure 1.2.5 Eclipse screen on the first run

Now we have to add to Eclipse the PyDev plugin, which 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 have to be connected to the Internet

To add a plugin, use the ***Help→Install New Software*** command (Figure 1.3.1):

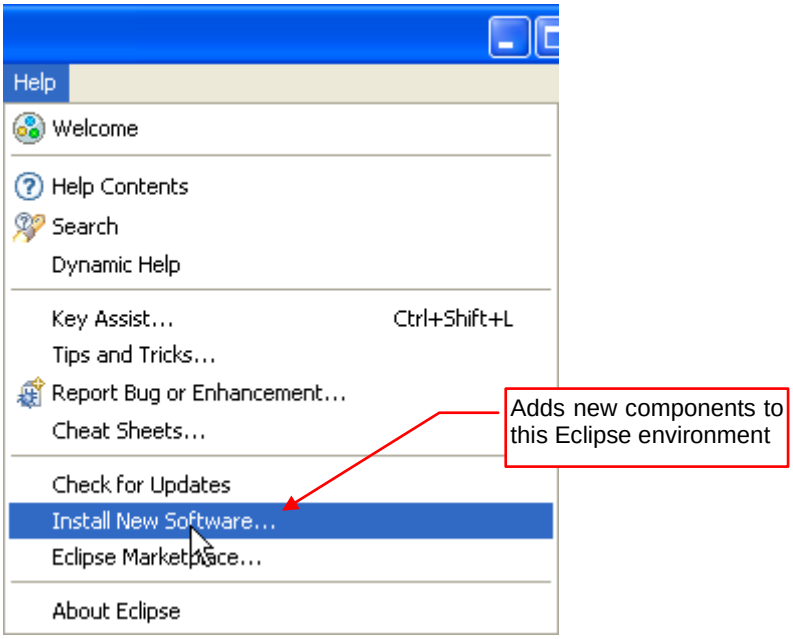


Figure 1.3.1 Opening the *Install* wizard for the Eclipse plugins

In the ***Install*** dialog, type the following address: http://pydev.org/updates (Figure 1.3.2):

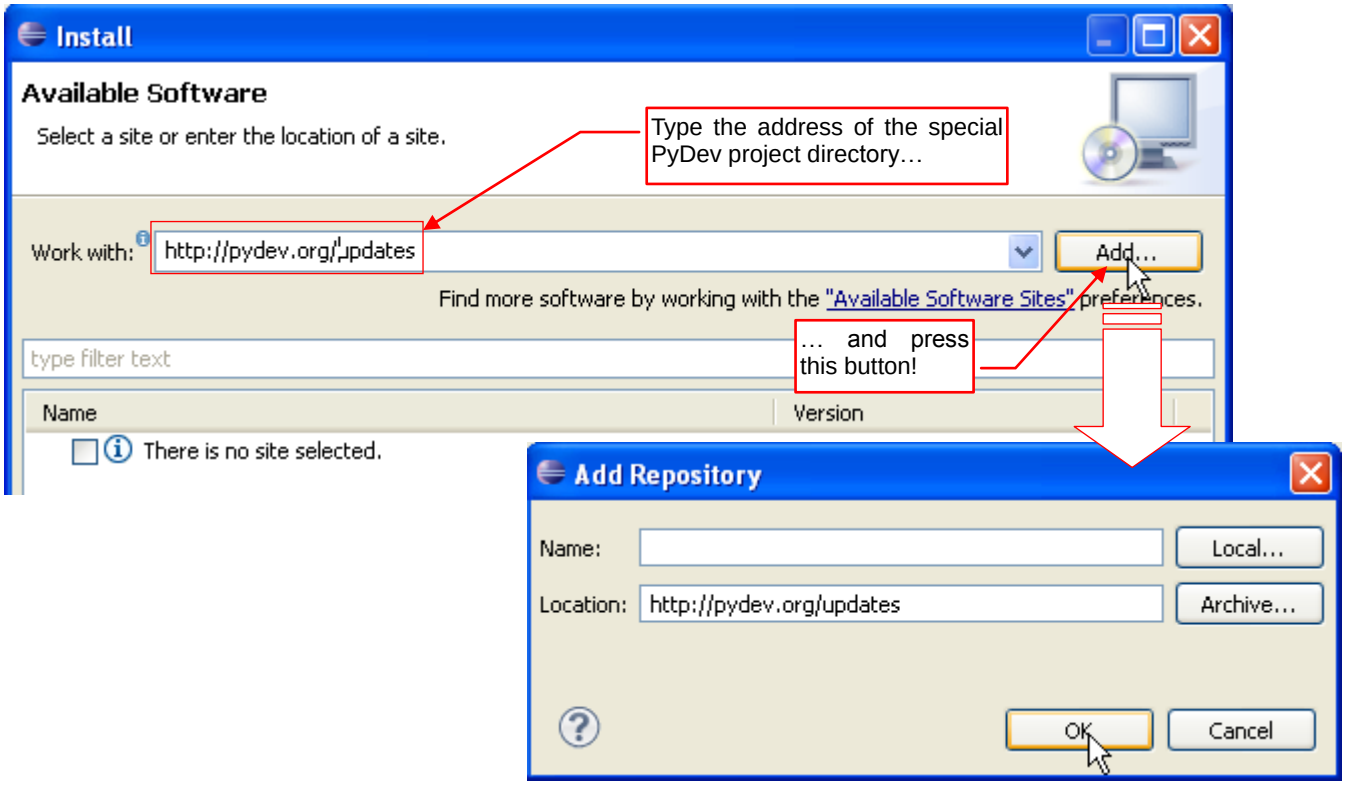


Figure 1.3.2 Adding the PyDev location to the list of the plugin sources

Then press the ***Add*** button. That opens the ***Add Repository*** dialog, which you can simply confirm.

Eclipse will read it, and after a moment it will display the contents of this repository (Figure 1.3.3):

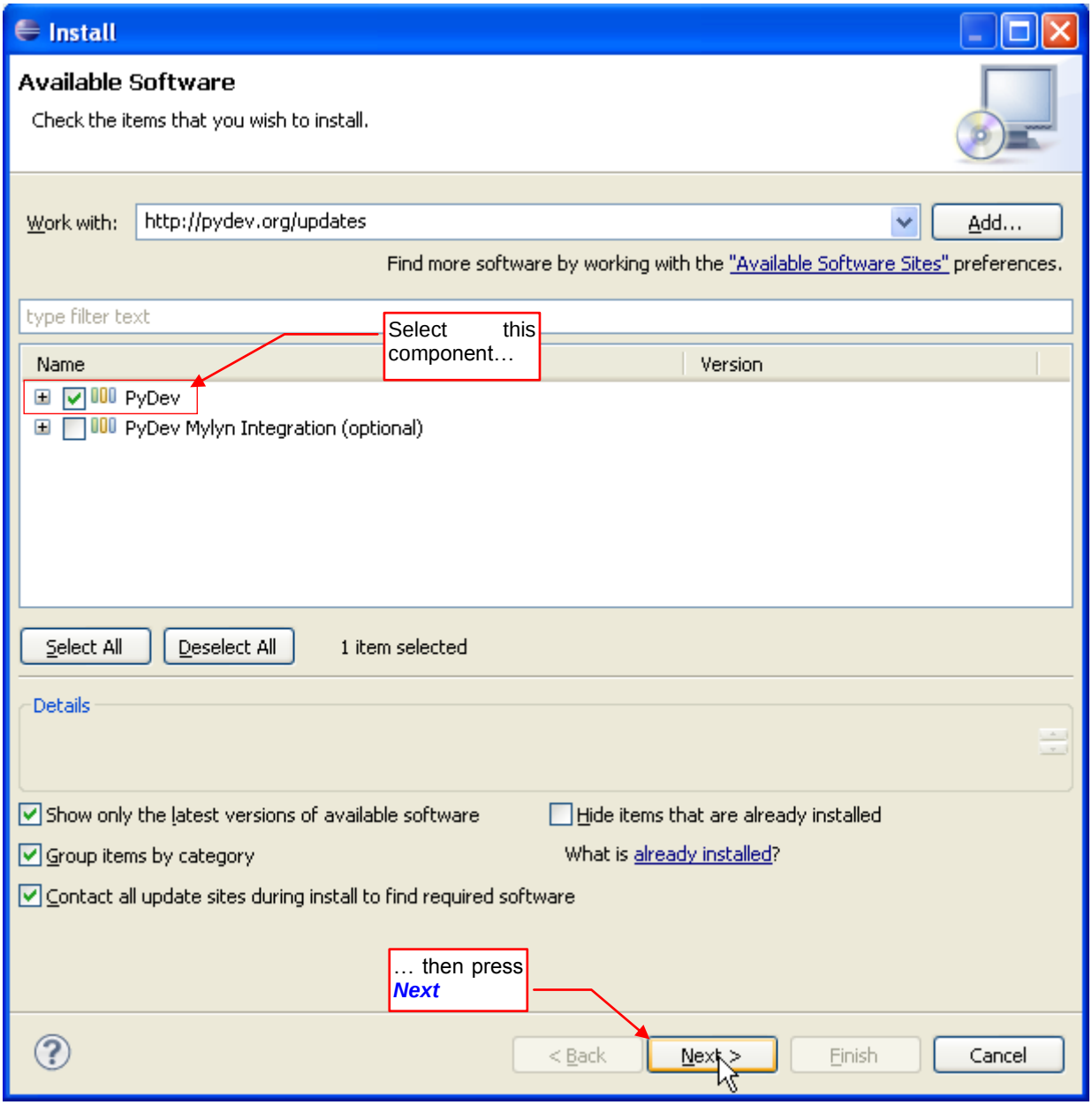


Figure 1.3.3 Selecting the PyDev plugin to install

Select the PyDev plugin from there then press the ***Next*** button. After passing some helper screens (one with detailed list of selected components, another with the license agreement — see section 5.2, page 103), the installation progress window will appear (Figure 1.3.4):

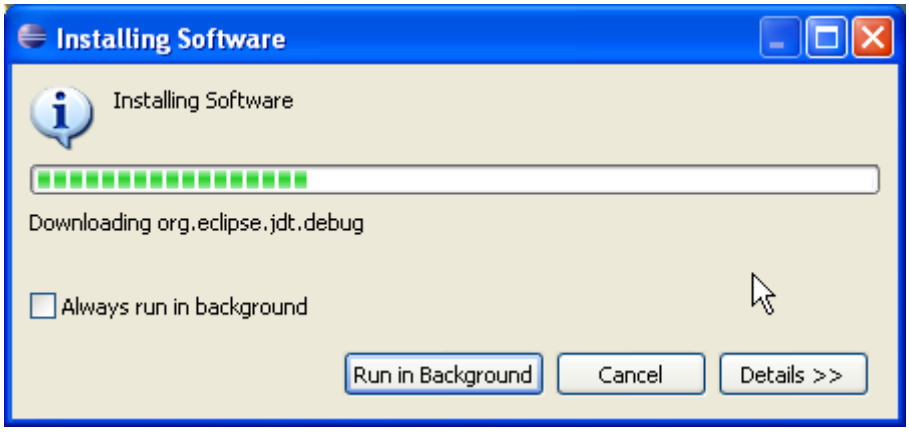


Figure 1.3.4 The installation progress window

After downloading, Eclipse will display the certificates of Aptana[1] PyDev for your confirmation. Finally, you will see a message about the need to restart Eclipse (Figure 1.3.5):

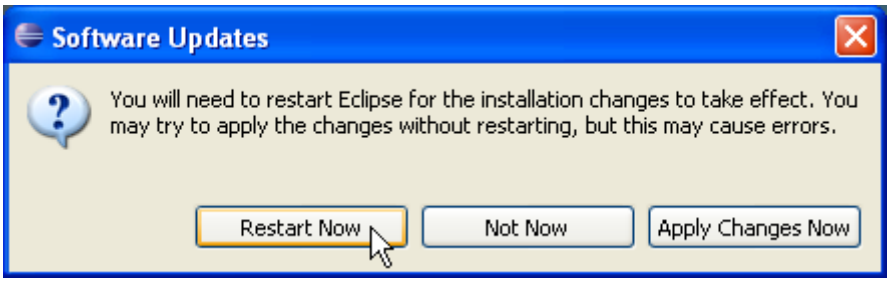


Figure 1.3.5 Final window

Do it as the precaution.

Eclipse saves separate configuration for each workspace (Figure 1.2.4). The default Python interpreter is also among these parameters. Let's set it straight away. To do it, invoke the ***Window→Preferences*** command (Figure 1.3.6):

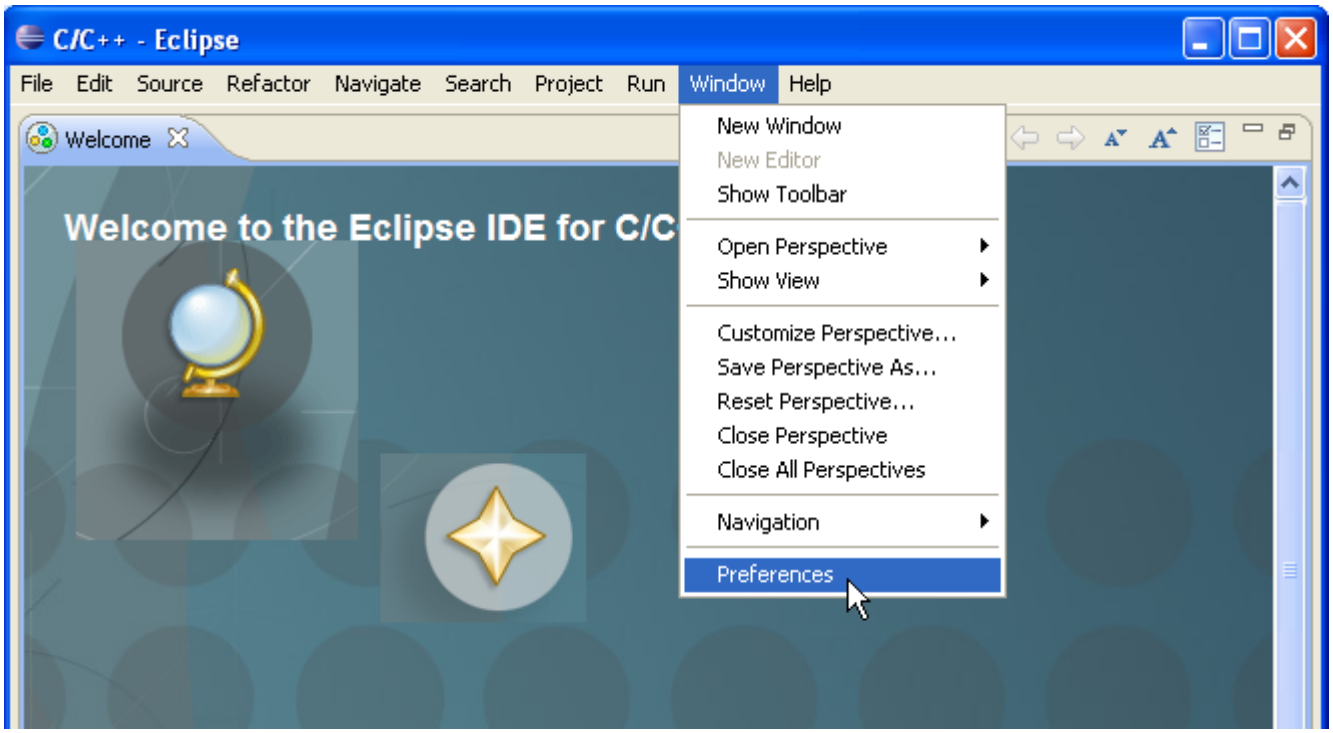


Figure 1.3.6 Setting up Eclipse configuration for the current workspace

[1] PyDev was created in 2003 by Alex Totic. Since 2005, the project has been run by Fabio Zadrozny. At that time PyDev had two parts: Open Source - PyDev, and commercial - PyDev Extensions (remote debugger, code analysis, etc.). In 2008 PyDev Extensions were acquired by Aptana. In 2009 Aptana "freed" PyDev Extensions, combining them with PyDev (version 1.5). In February 2011 Aptana was acquired by Appcelerator. PyDev portal is still on the Aptana/Appcelerator servers and Fabio Zadrozny continuously watches over its development.

In the *Preferences* window expand the ***PyDev*** node and select the ***Interpreter - Python*** item (Figure 1.3.7):

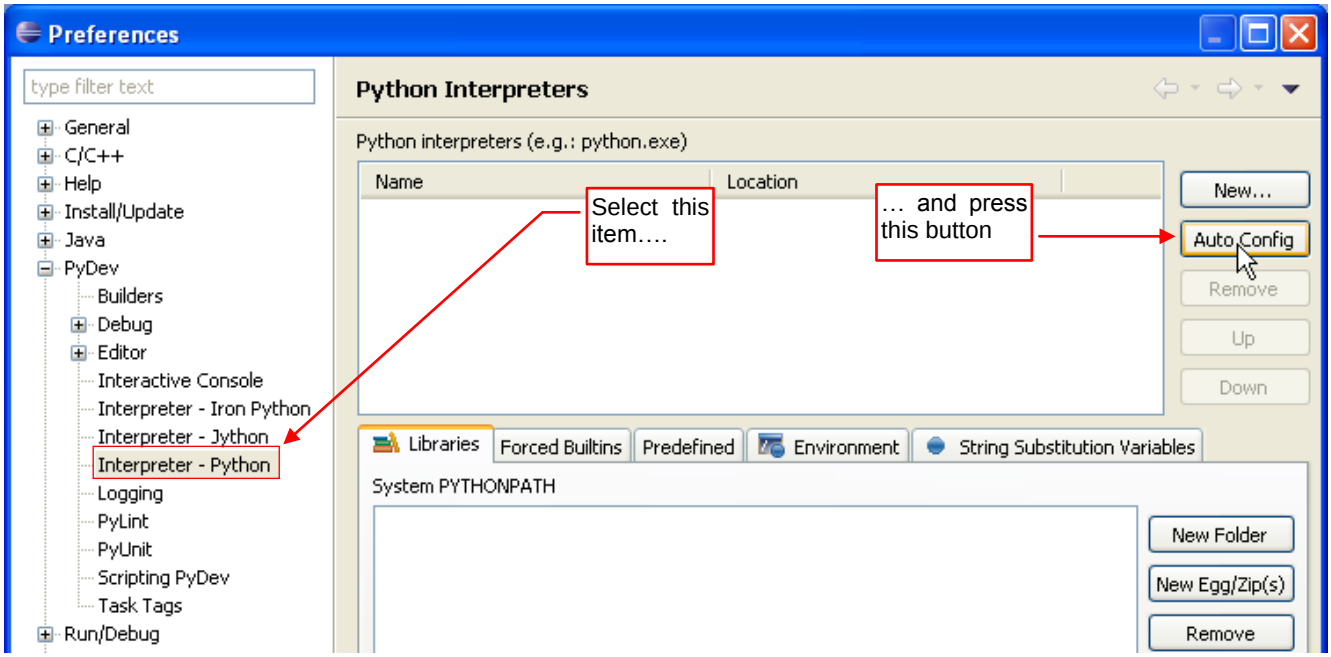


Figure 1.3.7 Invoking the automatic configuration of the Python interpreter

Then just press the ***Auto Config*** button. If your Python folder is present in the **PATH** environment variable, Eclipse will find and configure your interpreter (Figure 1.3.8):

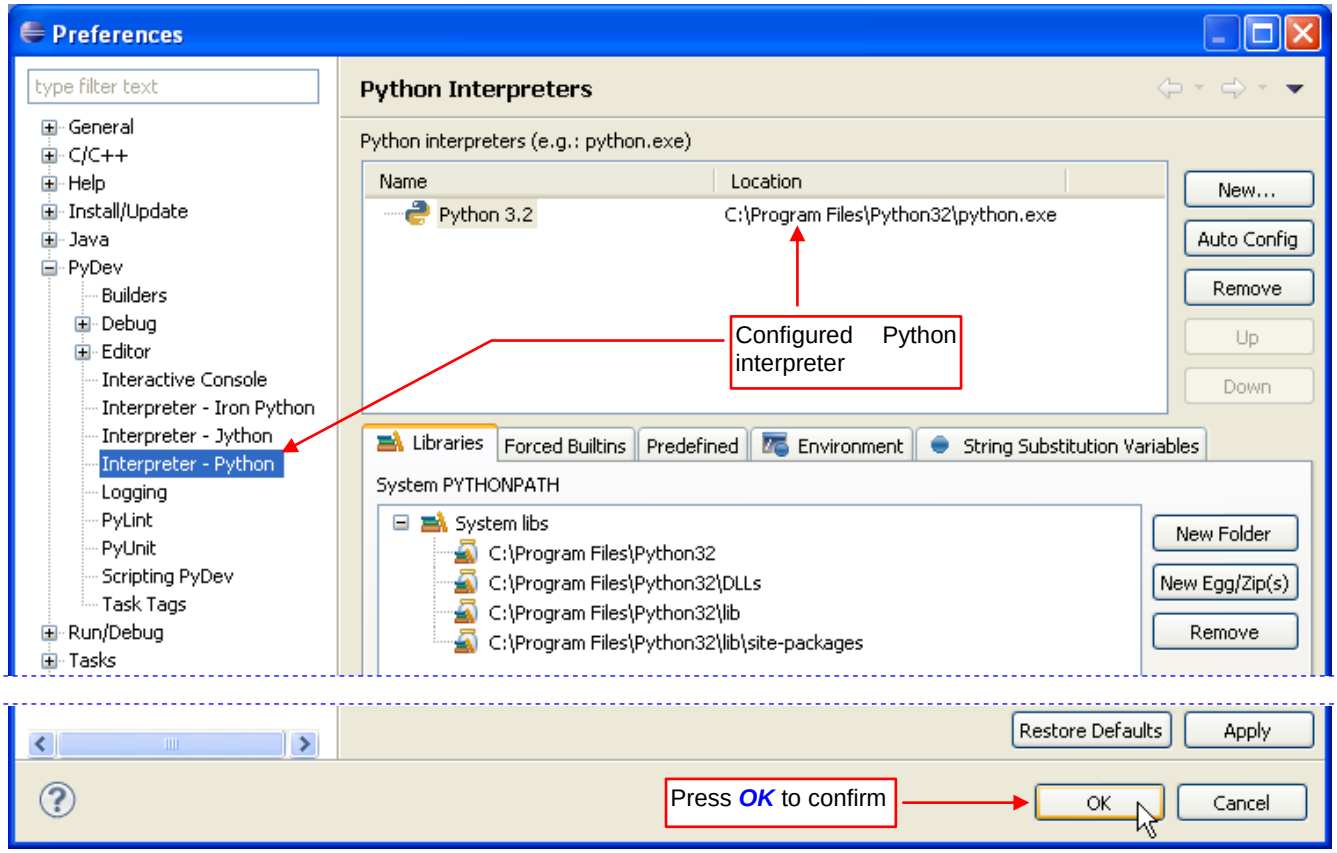


Figure 1.3.8 Configured Python interpreter

If you have two different Python versions on your computer — PyDev will list them in a dialog window, asking to select one. If PyDev displays a message that it cannot find any Python interpreter — perform the manual configuration (see section 5.3, page 110).

Chapter 2. Introduction to Eclipse

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

At the beginning, I want to show 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, which are well known to every programmer.

2.1 Creating a new project

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

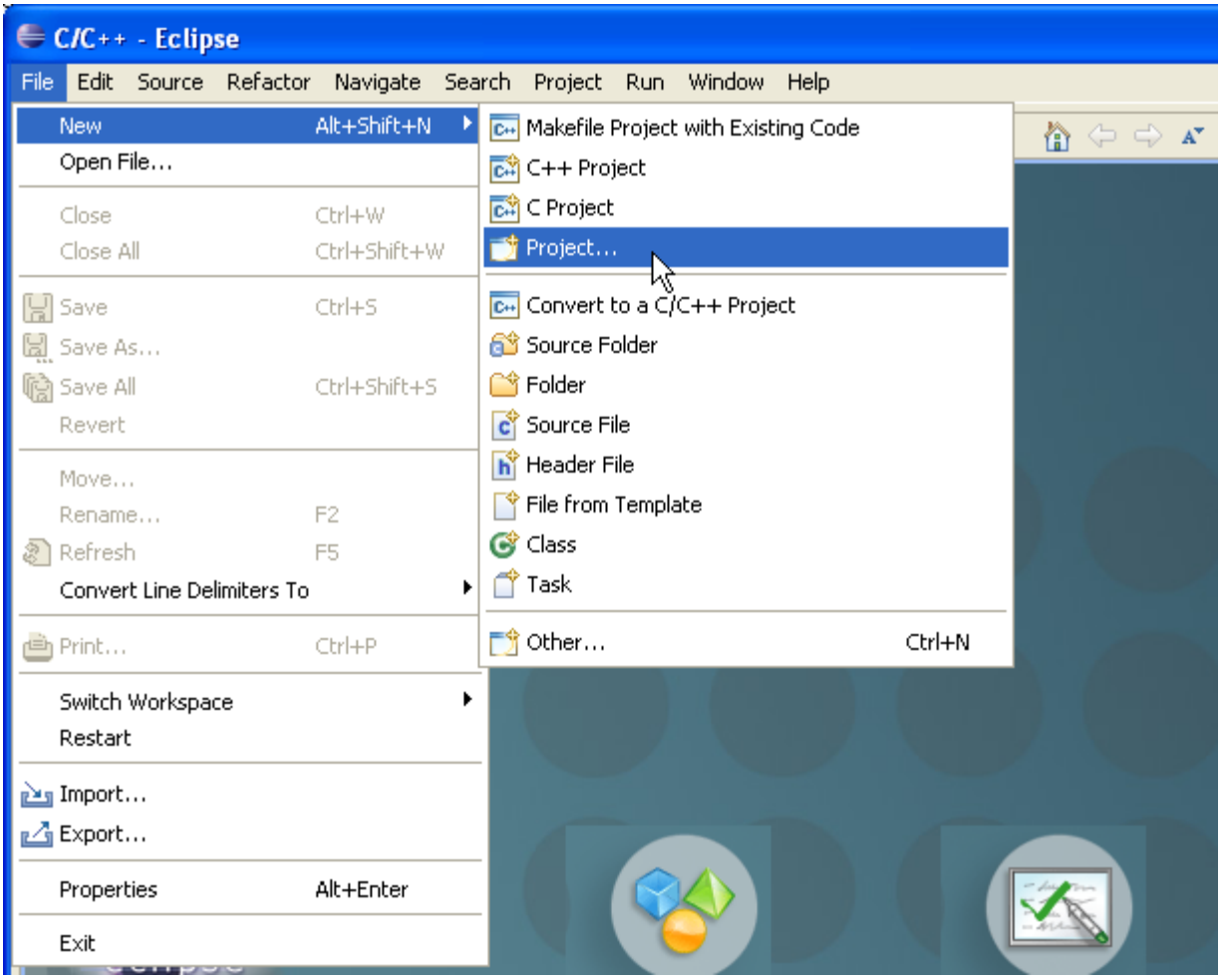


Figure 2.1.1 Opening a new project

It opens the *New Project* window. Expand the ***PyDev*** folder there and select the ***PyDev Project*** wizard (Figure 2.1.2):

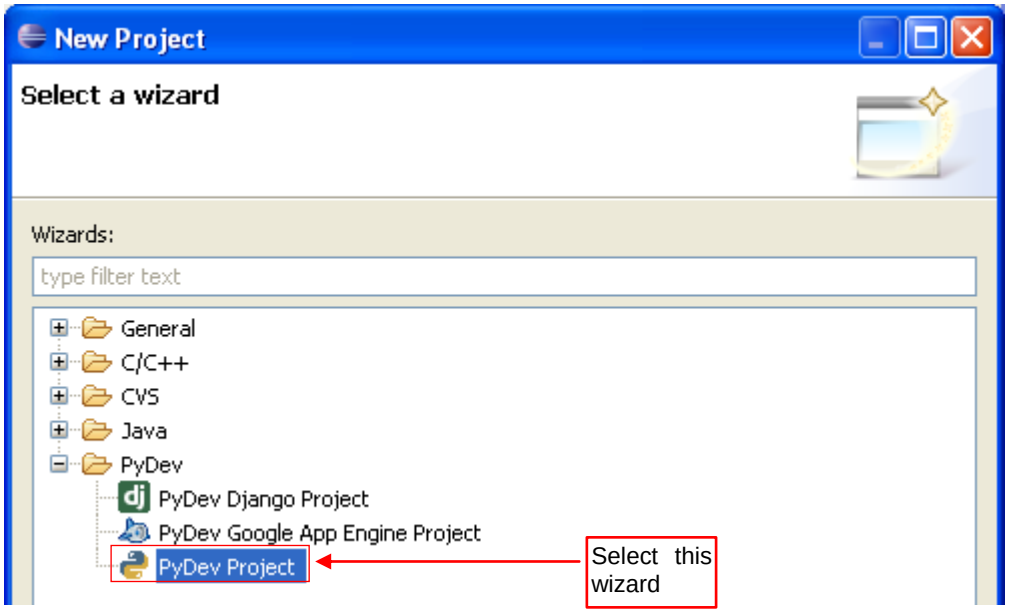


Figure 2.1.2 Selection of the appropriate project wizard

When the wizard is selected, press the ***Next*** button.

On the *PyDev Project* pane enter the ***Project name***. Let's start here right away a project that later will be used to implement the script for Blender. Hence, I give it the name **Bevel** (Figure 2.1.3):

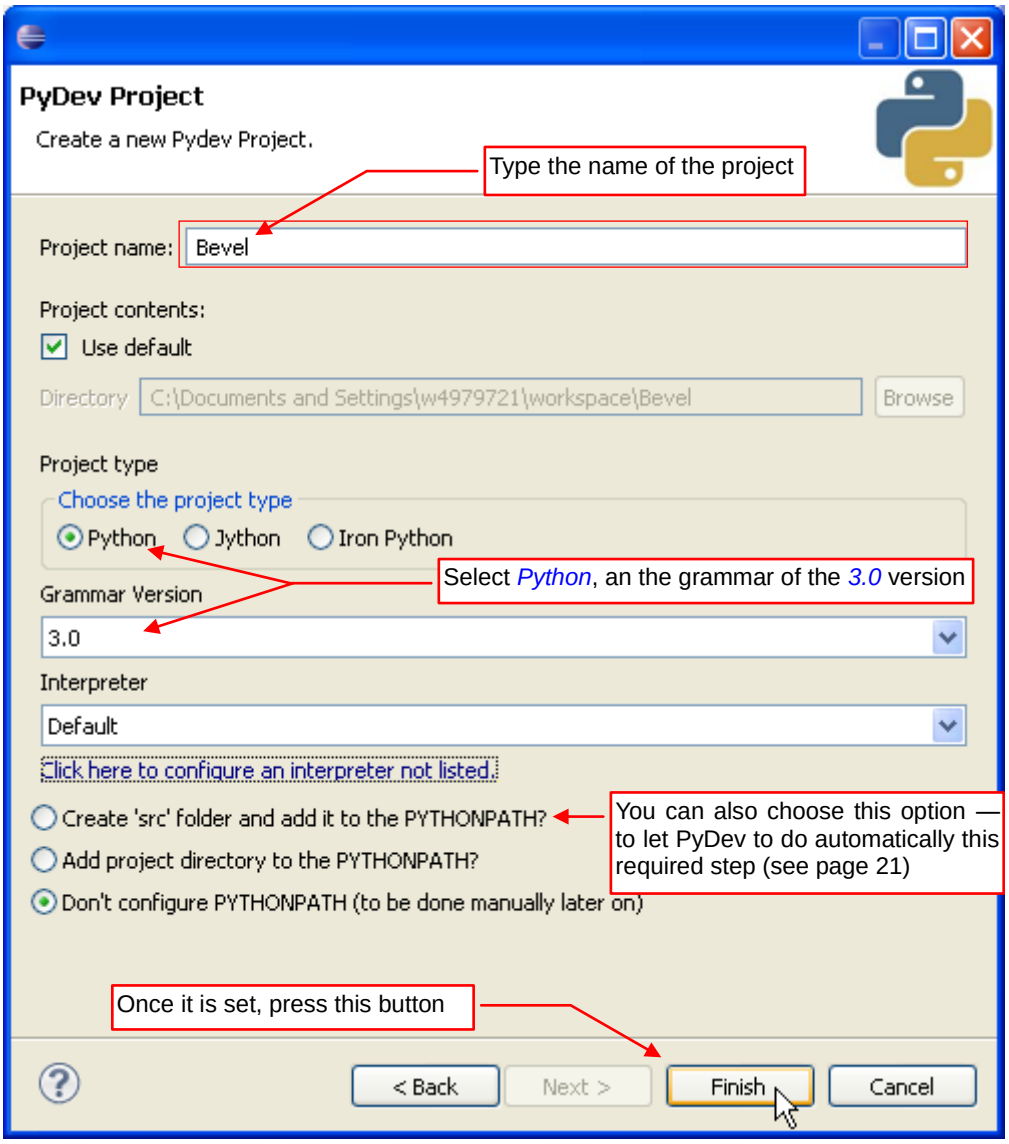


Figure 2.1.3 Filling the screen of *PyDev Project* pane

Set the *Project type* to ***Python*** and the *Grammar Version* to ***3.0***. Leave the rest of parameters unchanged and click the ***Finish*** button.

The PyDev wizard will ask you about creating the new *project perspective* (Figure 2.1.4):

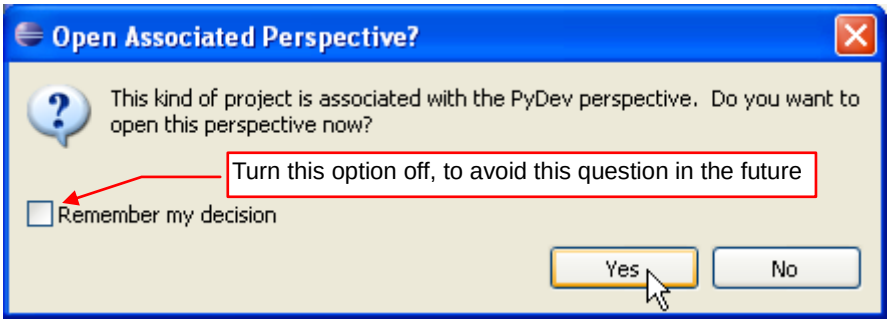


Figure 2.1.4 A question from the wizard.

(In Eclipse, the screen layout is called "*project perspective*"). Just confirm this question (***Yes***).

Beware: if you forgot to configure the Python interpreter, the wizard would display an error and the ***Finish*** button would be grayed out (Figure 2.1.5):

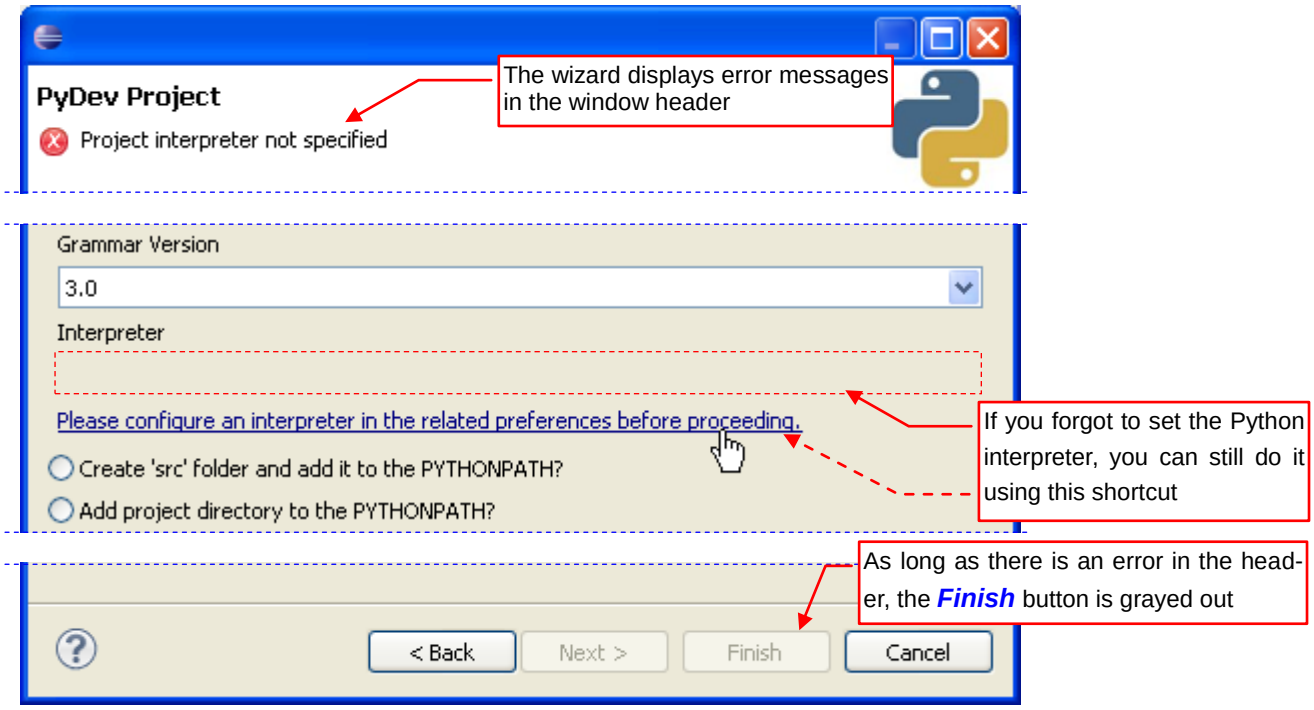


Figure 2.1.5 The error reported by the wizard, when the Python interpreter is not set

When you have got such an error, use the shortcut displayed by the wizard in the window. It opens the *Preferences* dialog and allows you to complete the missing configuration (see section 5.3). Once it is done, return to the wizard to create the new project.

The PyDev wizard creates in Eclipse an empty Python project (Figure 2.1.6):

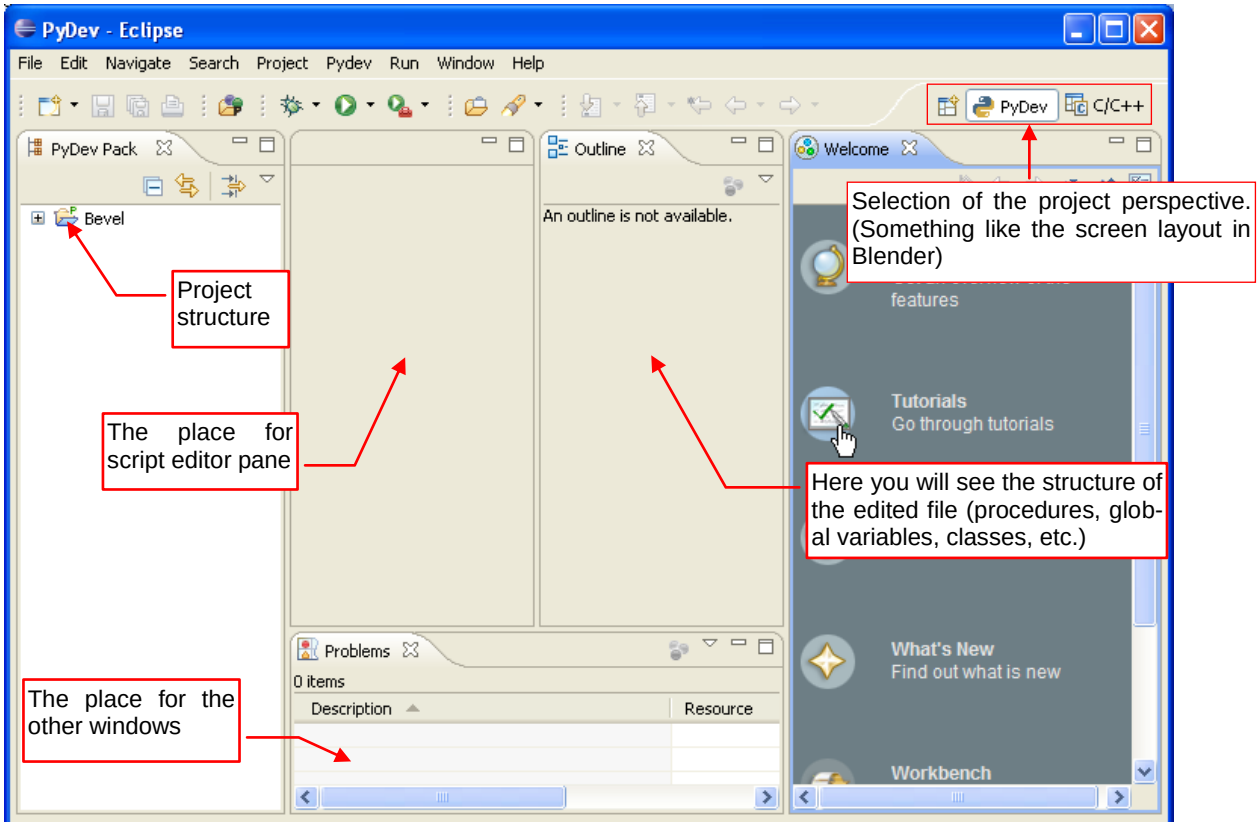


Figure 2.1.6 PyDev *perspective* of a new project

What you see is the default PyDev screen layout. In Eclipse, just like in Blender, you can have many alternative "screens". They are called *perspectives* here. Every newly created project contains the default PyDev perspective. When you try to debug your script for the first time, another perspective will be added (*Debug*).

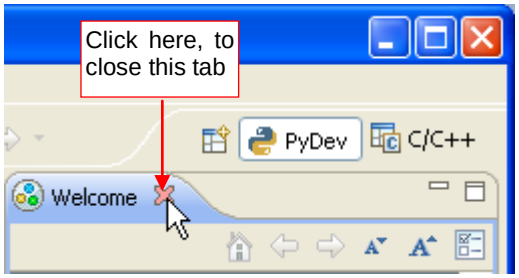


Figure 2.1.7 Closing the *Welcome* tab.

Let's start adaptation of the current perspective with removing the unnecessary *Welcome* pane (Figure 2.1.7):

Then add to this project a subfolder for the scripts: select the project folder, and from its context menu select the ***New→Source Folder*** command (Figure 2.1.8):

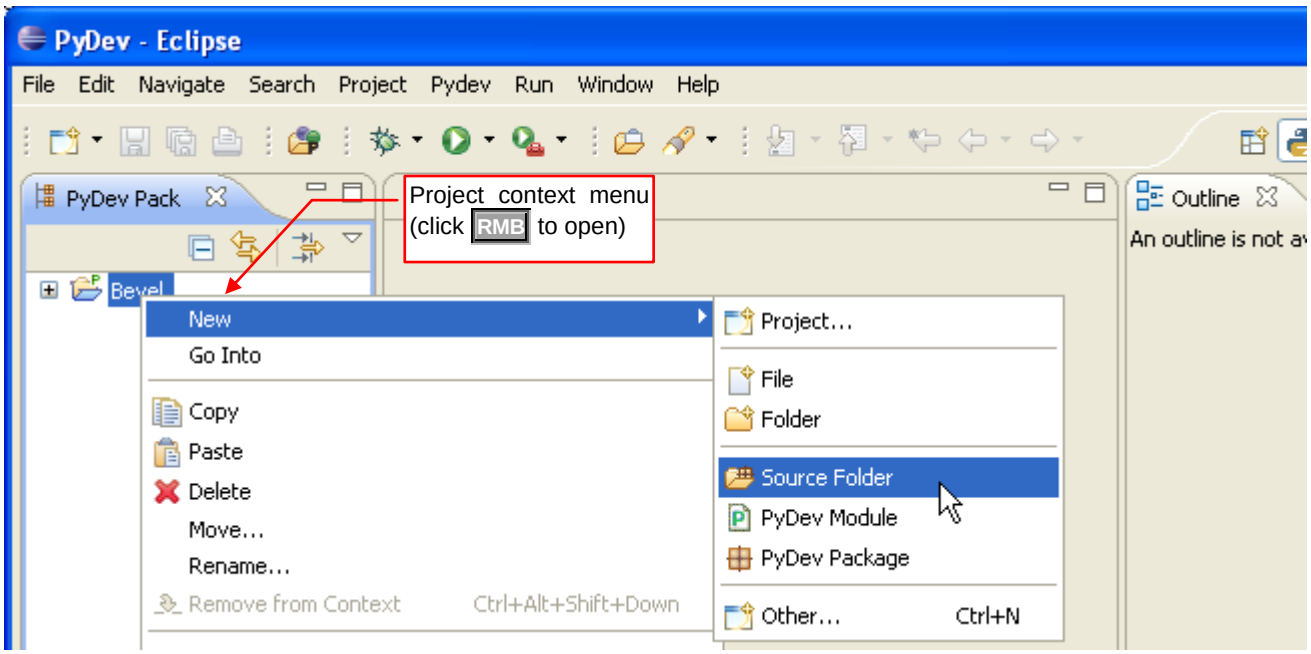


Figure 2.1.8 Adding a subfolder for the scripts

Type the subfolder ***Name*** on the wizard pane — let it be **src** (Figure 2.1.9):

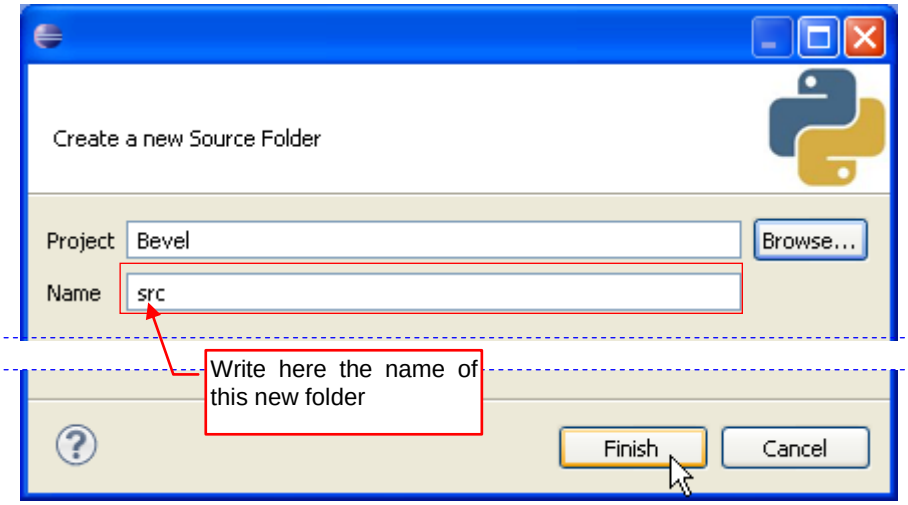


Figure 2.1.9 Folder wizard pane

Finally, press the ***Finish*** button. The wizard will create the new subdirectory **src** in the project directory.

We will create an empty script file, now. Expand the context menu of **src** folder, and invoke the ***New→PyDev Module*** command (Figure 2.1.10):

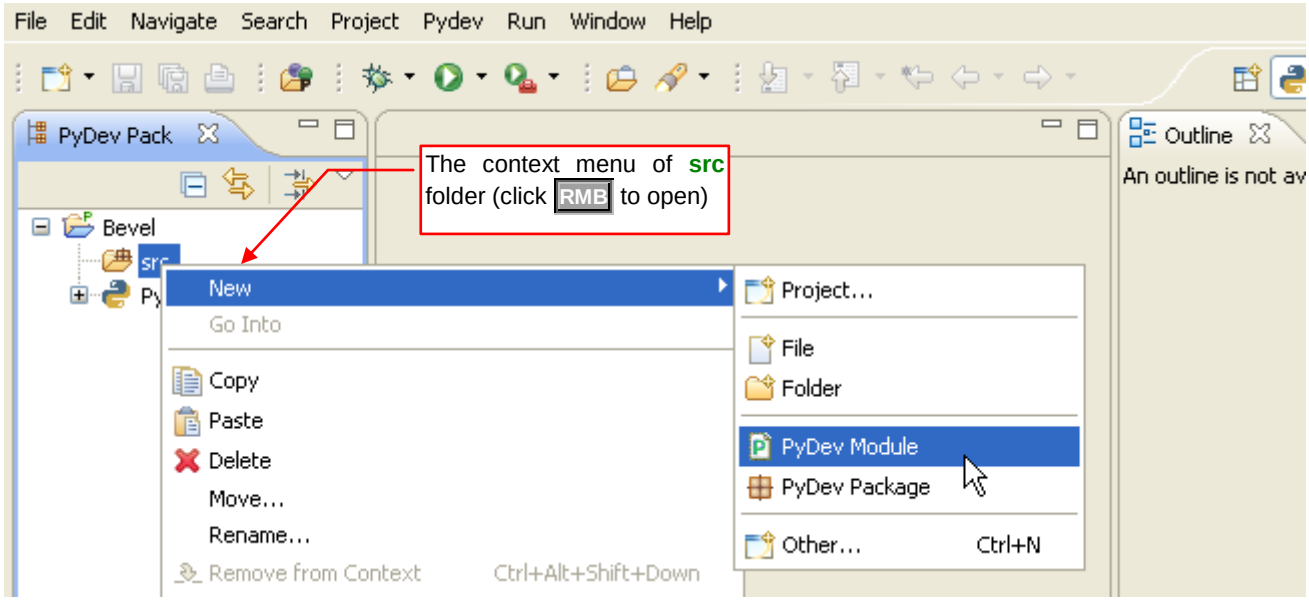


Figure 2.1.10 Invoking the new script ("module") wizard

It will open another PyDev wizard window. Give this file a name suitable for the Blender add-on: **mesh_bevel**, select the ***<Empty>*** item from the *Template* list (Figure 2.1.11):

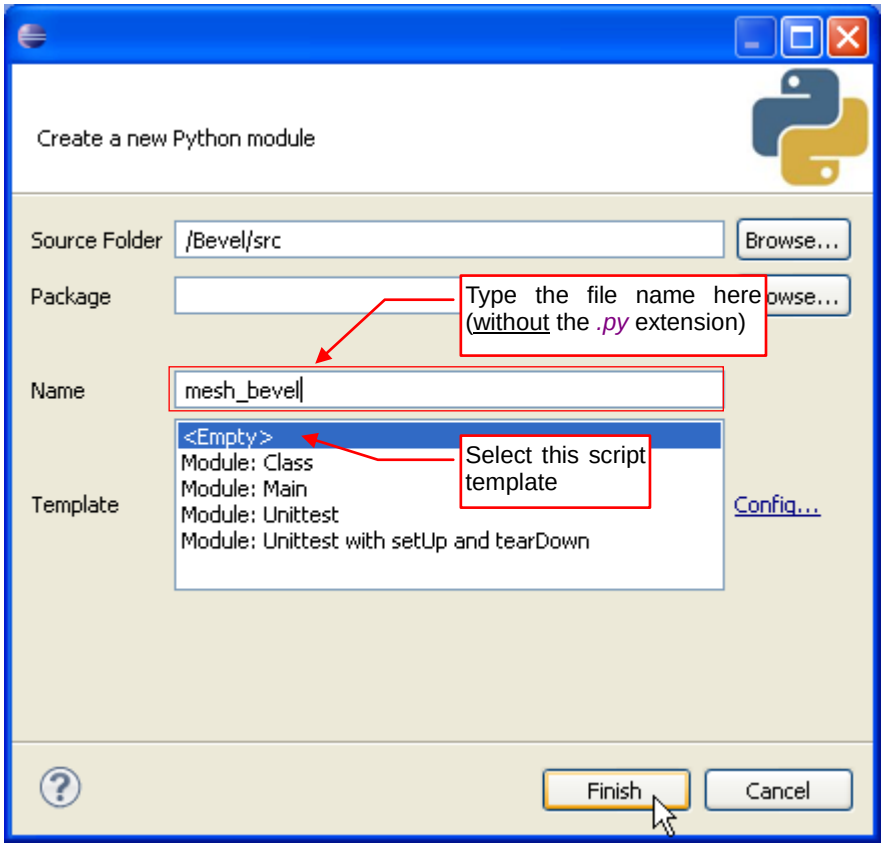


Figure 2.1.11 PyDev module wizard window

Finally, press the ***Finish*** button.

If you have on your computer very restrictive firewall, you will receive now a request to open a TCP port. (I received this comment from my reviewer). It is about accessing the 127.0.0.1 loopback. Anyway, I have not seen this myself, although my firewall is not very permissive.

This way PyDev has added an empty script file to our project. It contains just a header *docstring* comment, with the creation date and the author name (Figure 2.1.12):

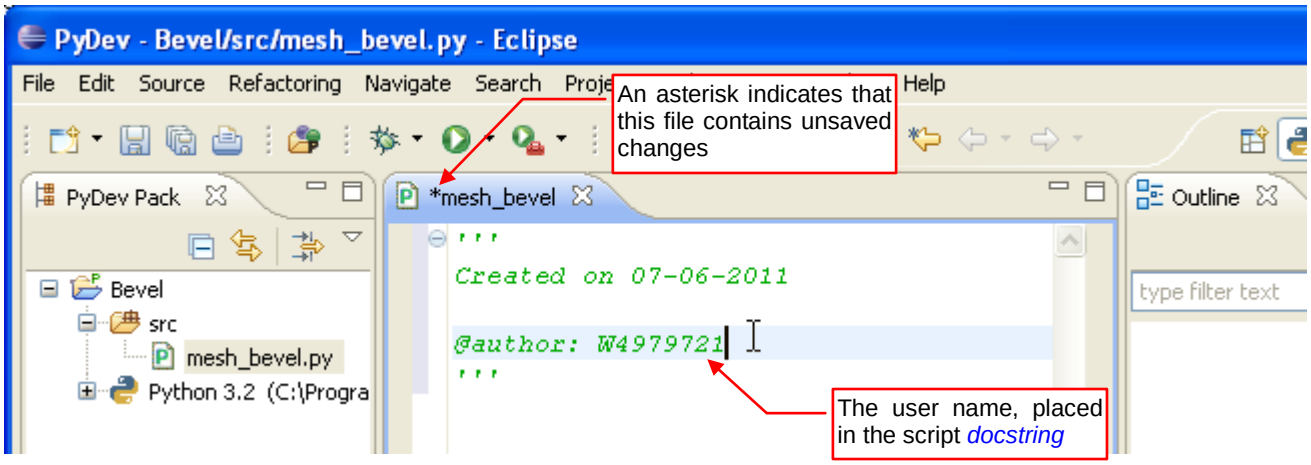


Figure 2.1.12 The new, empty script

Summary

- In this section, we have created a new Python project using the ***PyDev Project*** wizard (page 18);
- Name of the project is an arbitrary matter. In this example, I called it "**Bevel**" (page 19), because in the further chapters of this book it will serve us to implement the Bevel command in Blender 2.5. For the same reason I gave the script file the name appropriate for the Blender add-on: "*mesh_bevel.py*" (page 22).
- Eclipse requires in its project a special *source folder* (page 21) for the scripts. (You cannot place them in the root directory);
- You can use several predefined templates for your script (page 22). For the Blender script I have just selected the *<Empty>* template;

2.2 Writing the simplest script

The script that we 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 has not added it by default. To do this, click on the tab at the bottom of the screen (because there we will add the console). Then invoke the ***Window→Show View→Console*** command (Figure 2.2.1):

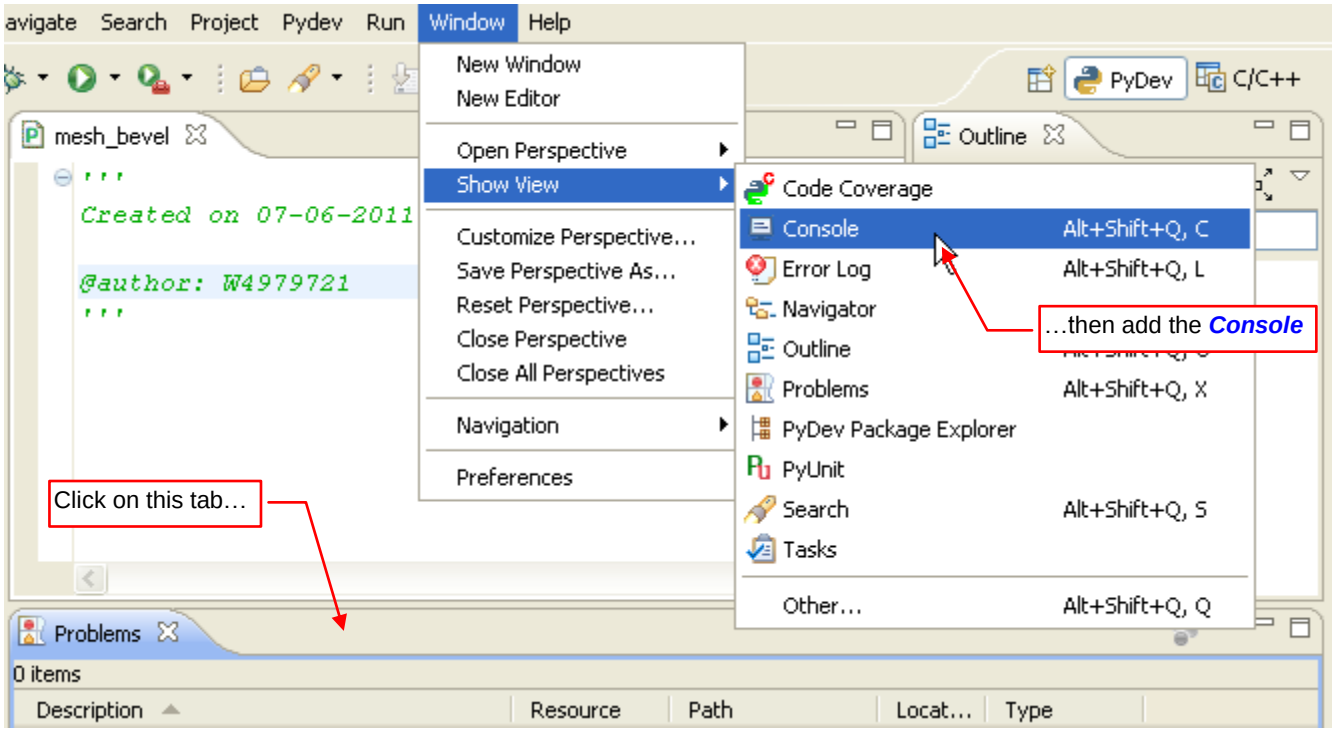


Figure 2.2.1 Adding the *Console* tab

By default, this output console shows the result of the script. Dynamic languages, like Python, offer also something like "interactive console". It runs the Python interactive interpreter, allowing you to check some expressions while writing the script. So let's add it to our windows (Figure 2.2.2):

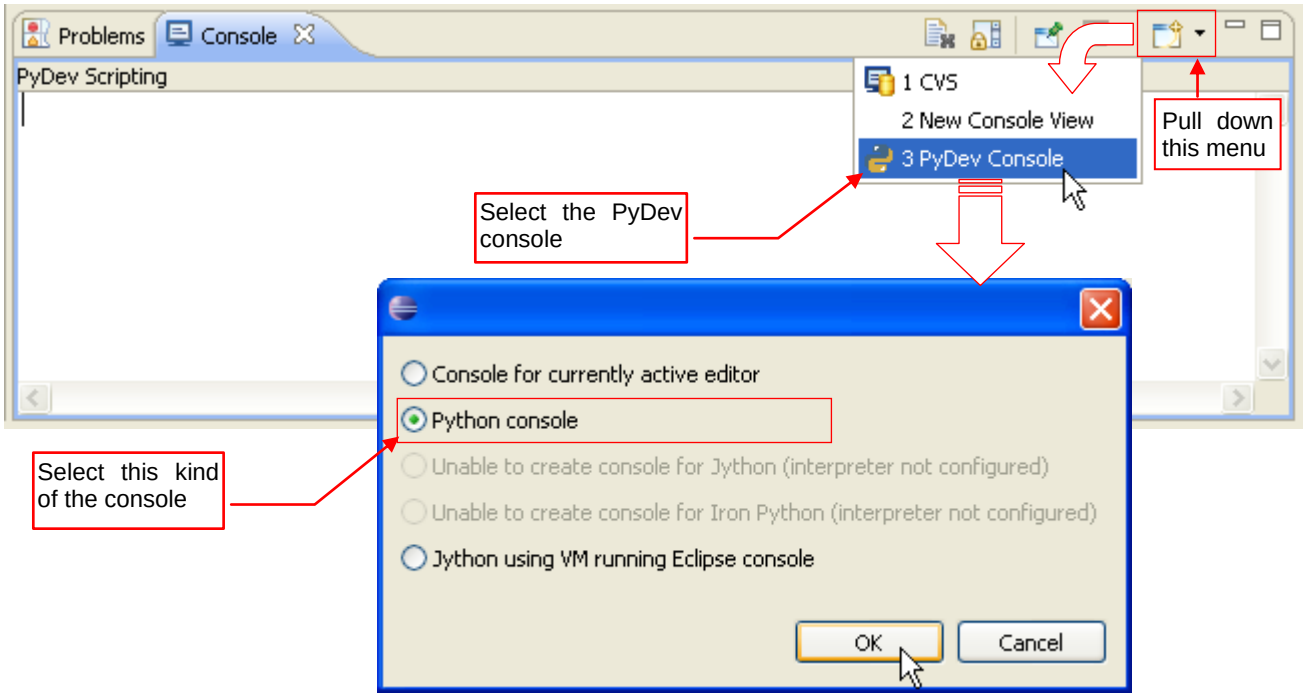


Figure 2.2.2 Switching to the interactive Python console

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

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

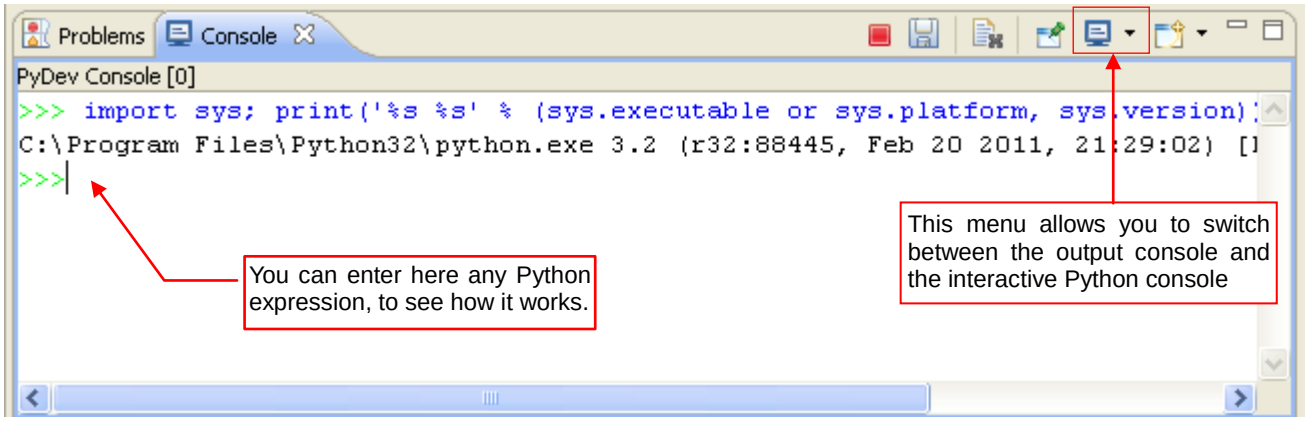


Figure 2.2.3 Interactive Python console

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

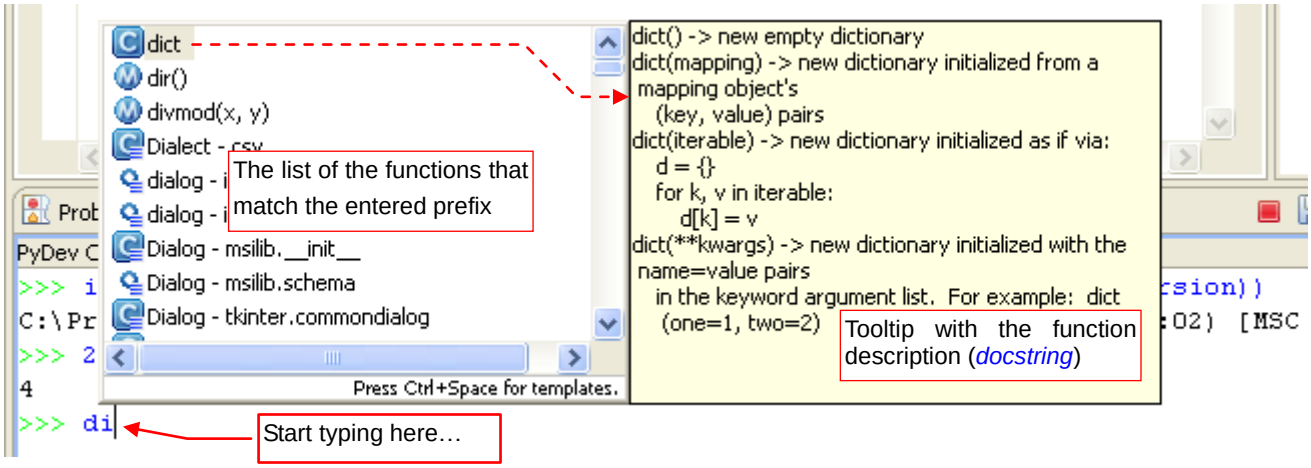


Figure 2.2.4 Example of the code autocompletion

Autocompletion usually takes effect when you type dot after a name (for example, type "sys." in the console). Such 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. Its time to write our simplest script (Figure 2.2.5):

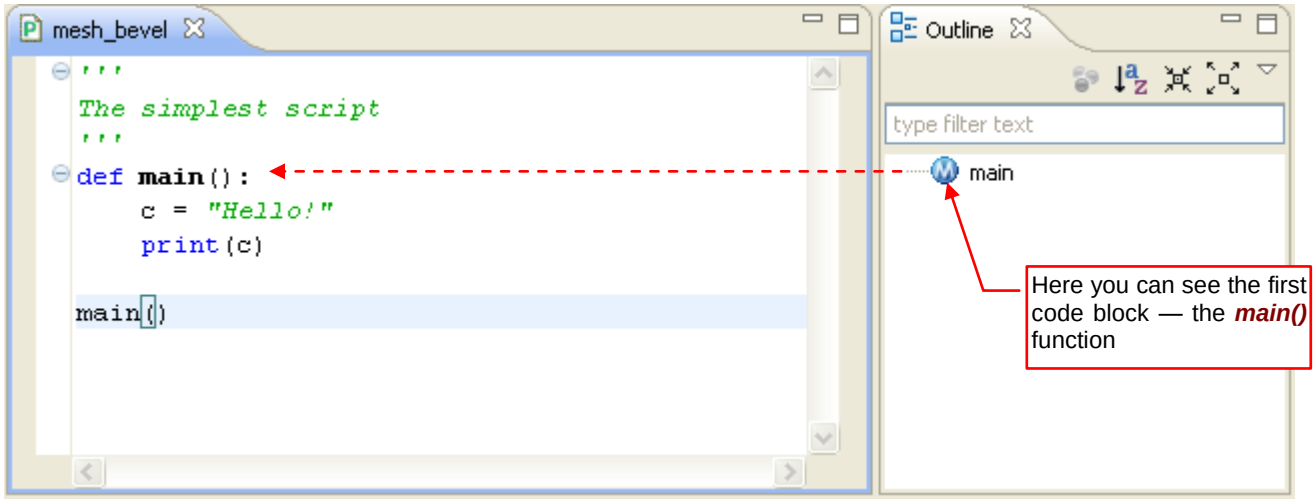


Figure 2.2.5 Our script — the first version, of course ☺

When the script is ready, highlight its file in the project explorer and from the *Run* menu invoke the ***Run As→Python Run*** command (Figure 2.2.6):

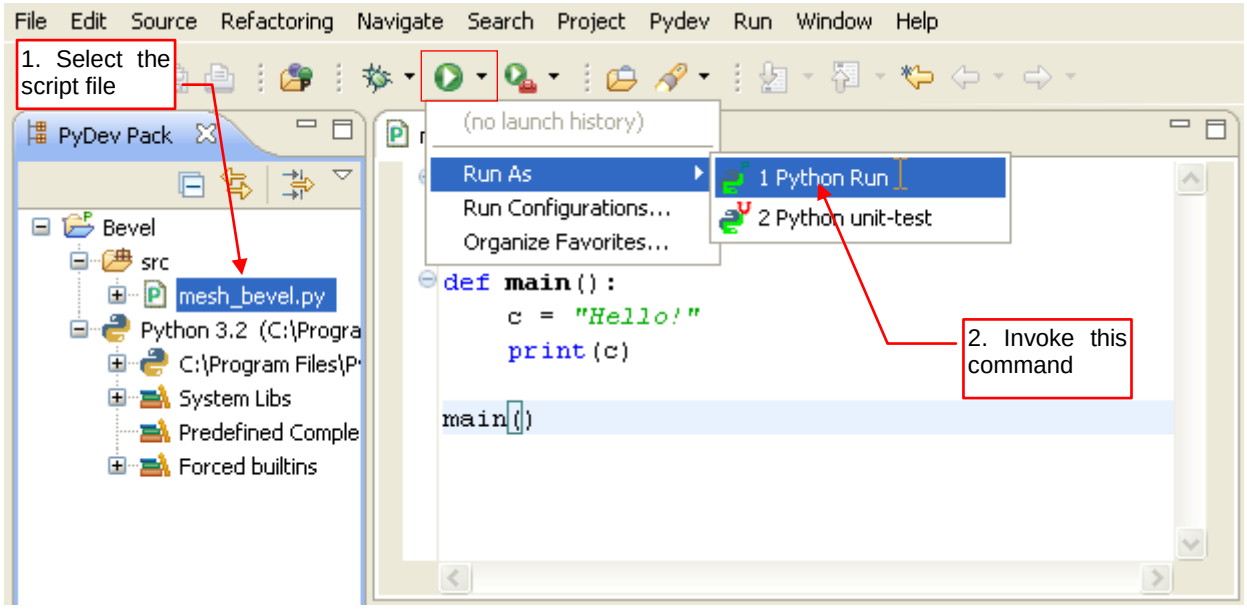


Figure 2.2.6 Launching the script

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):

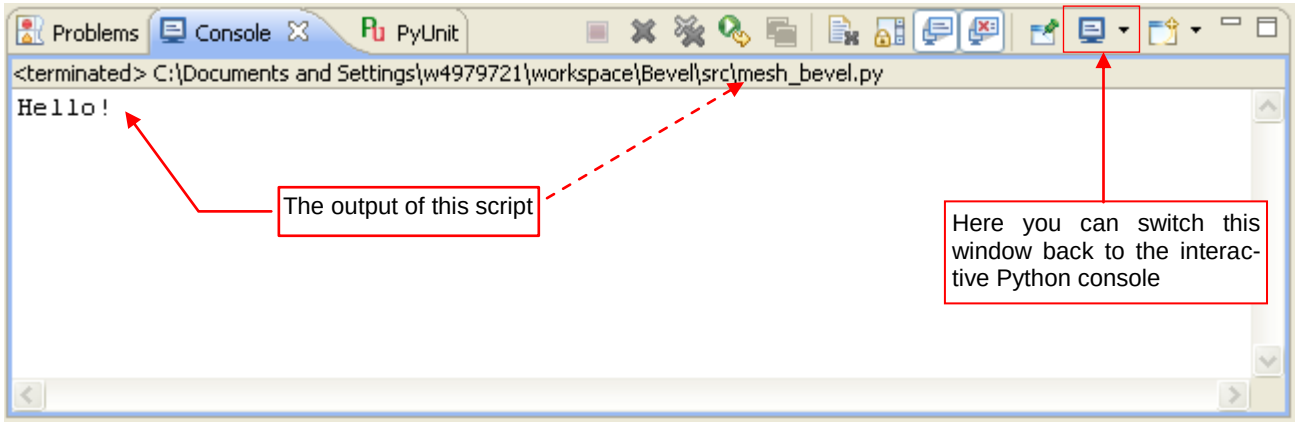


Figure 2.2.7 The result of our script — the text written in the output console

Summary

- We have added to our project new pane with the Python console (page 24);
- You have seen how the code autocompletion works, and how it displays the *docstring* of selected function (in the tooltip — page 25);
- We have launched the simplest script and checked its result (page 26);

2.3 Debugging

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

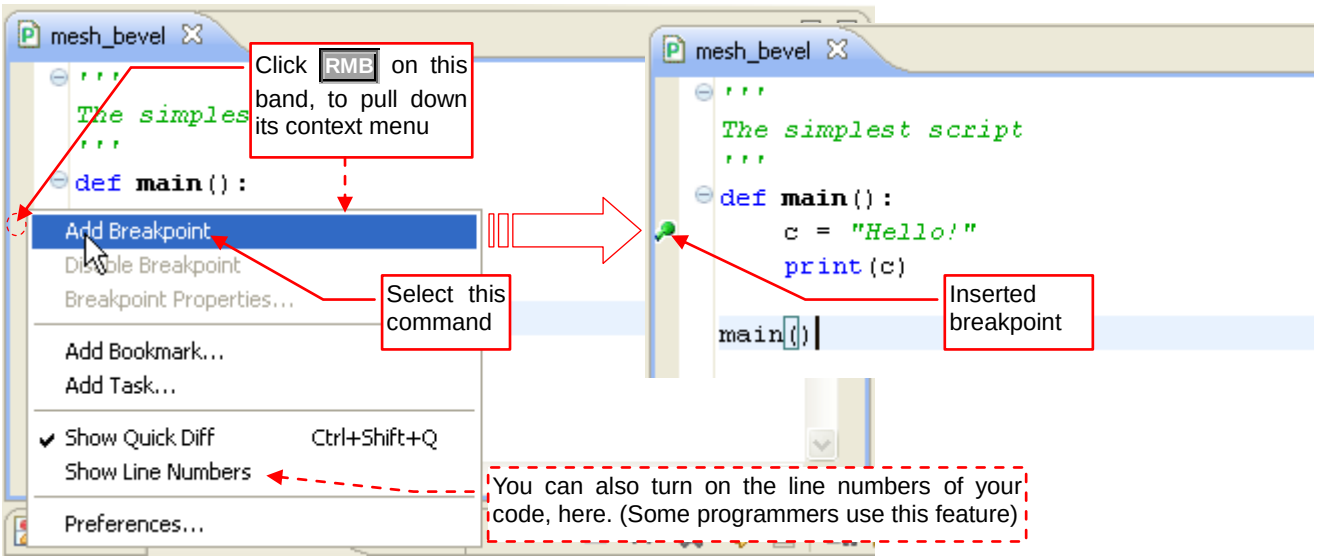


Figure 2.3.1 Adding a breakpoint

To open it, click the RMB at the line, where you want to insert the breakpoint. Invoke the ***Add Breakpoint*** command from there. Eclipse will mark this point with a green dot with a dash (Figure 2.3.1). (You can remove the breakpoint in a similar way, double clicking LMB or using the context menu).

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

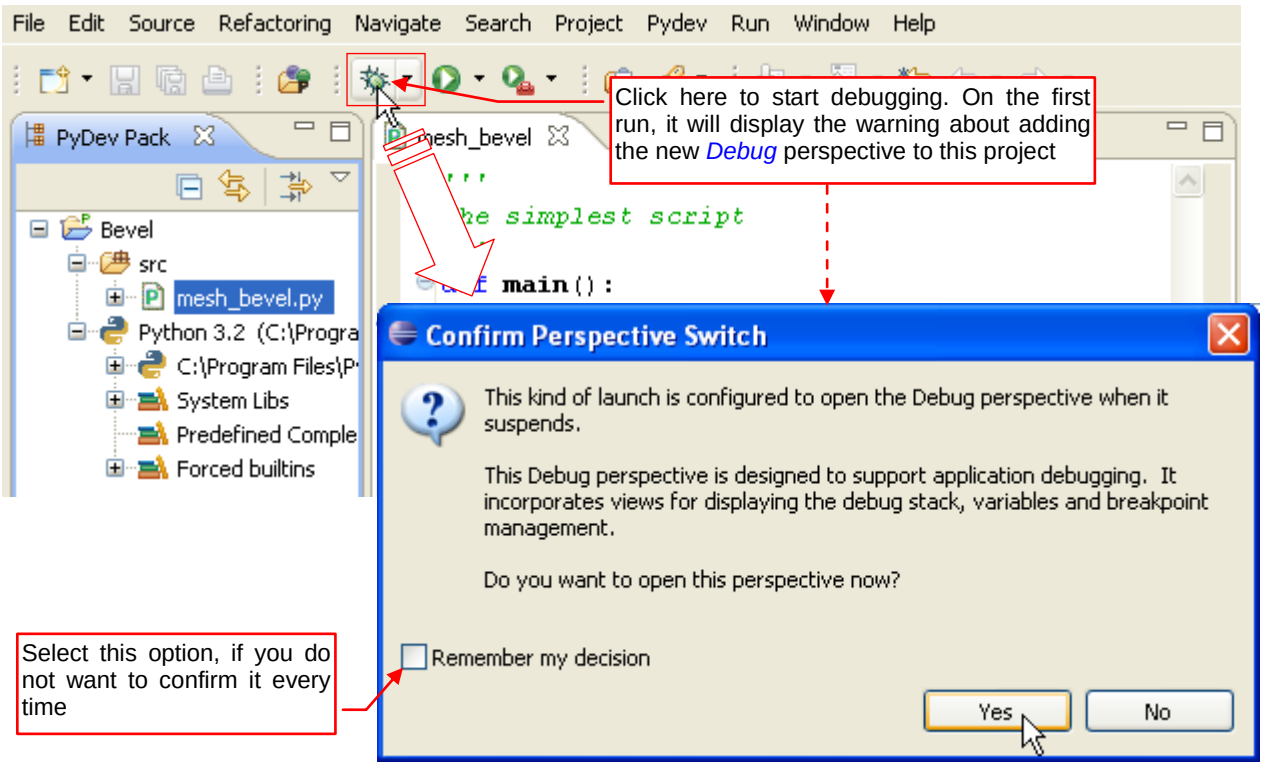


Figure 2.3.2 Launching a debug session

While launching the debugger, Eclipse always displays information about switching to the *Debug* perspective. (On the first run, it will add this perspective to your project). Remember that you have to be in the *Debug* perspective, to be able to step through the script!

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

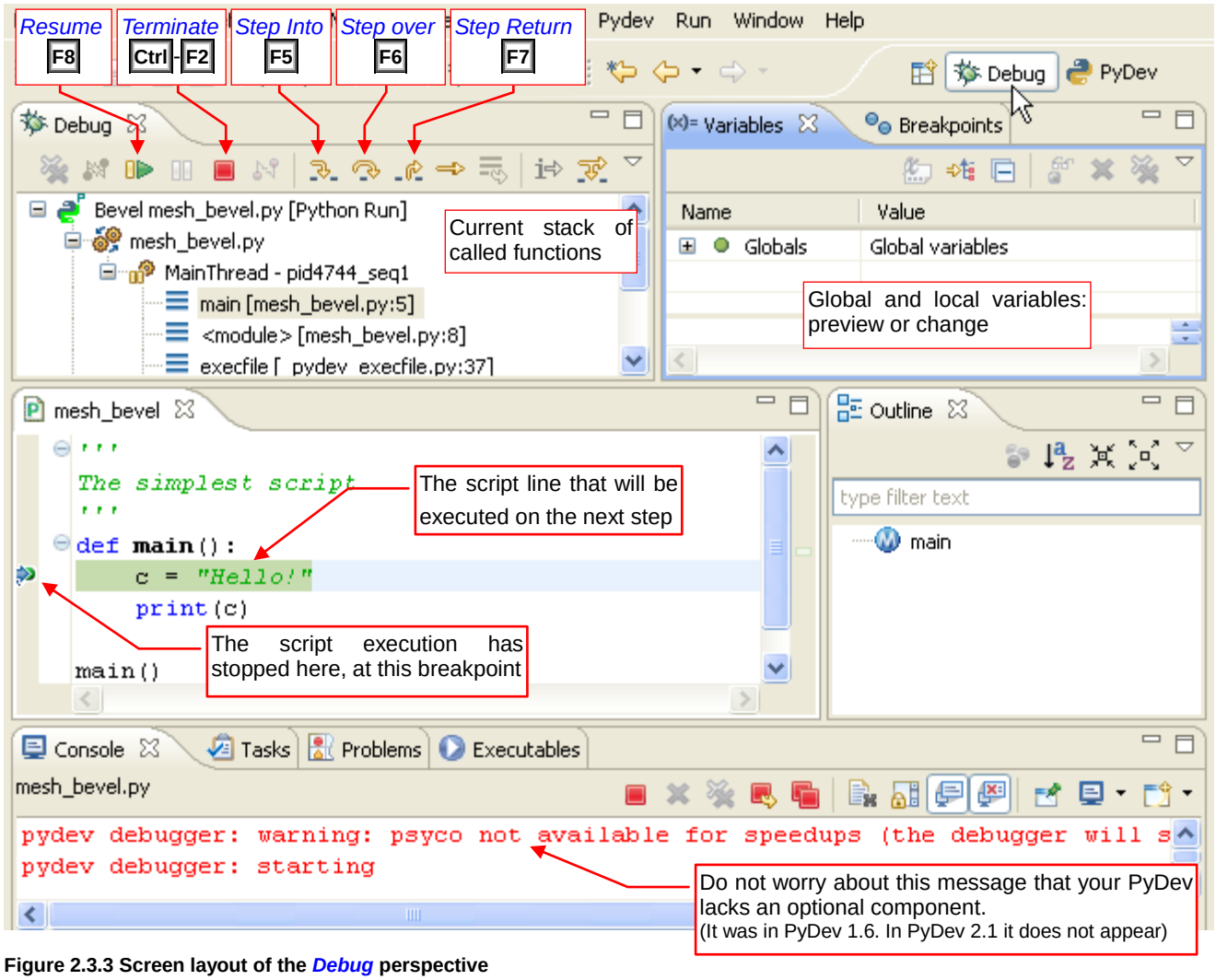


Figure 2.3.3 Screen layout of the *Debug* perspective

Green area on the source code marks the line to be executed. When you press now the F6 key (*Step over*) — you set the ***c*** variable and move it to the next line (Figure 2.2.4):

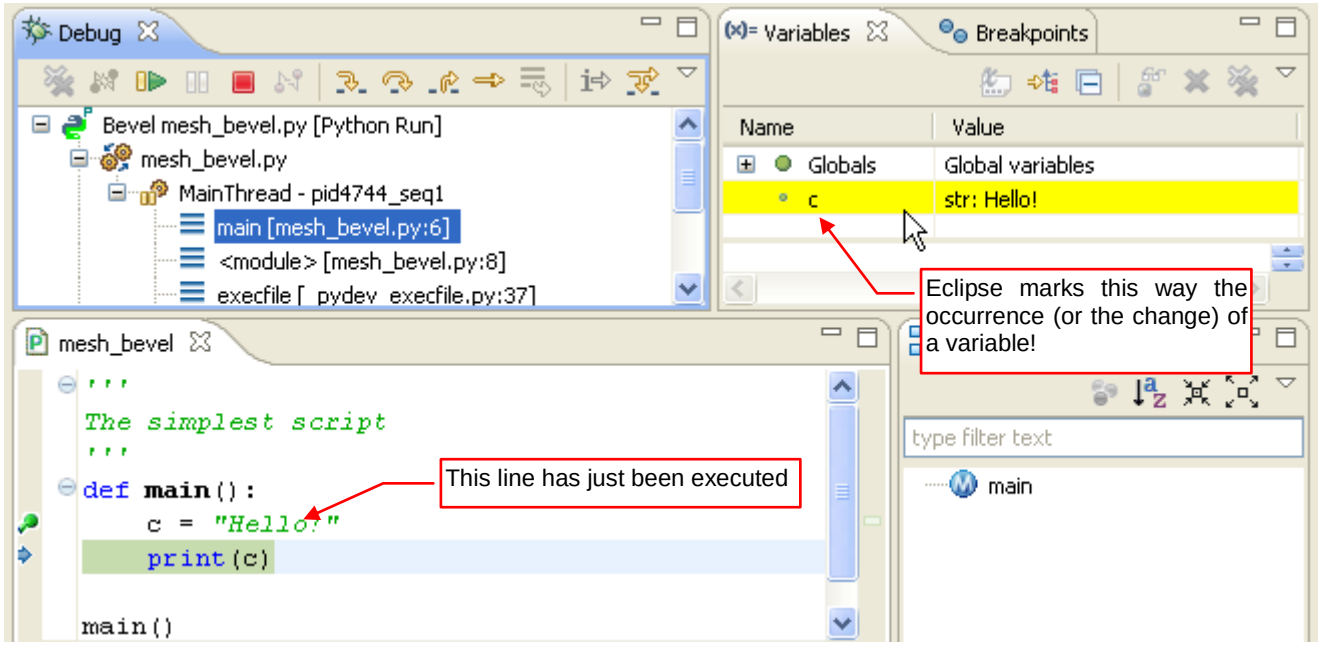


Figure 2.3.4 The state after pressing the F6 key (*Step over*)

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

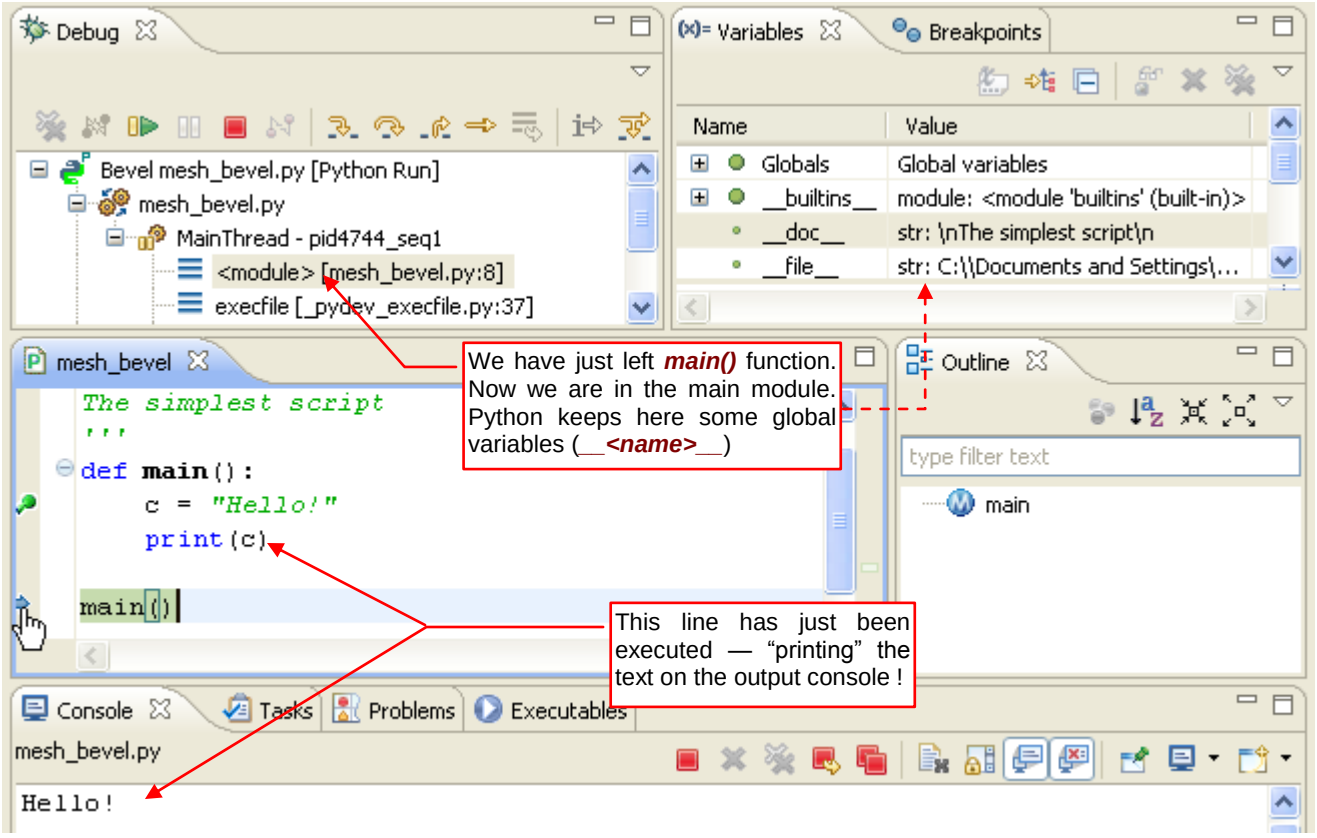


Figure 2.3.5 The state just after leaving the function

Notice that the top line (*main() [mesh_bevel.py]*, visible on Figure 2.3.4) has been removed from the stack. Yet the current line is still stuck on the call to this function in the main module (*<module> [mesh_bevel.py]*). In this way the PyDev debugger indicates the end of the function. It was the last line of our script. If you press the F6 key again, you will find yourself in an internal PyDev module (Figure 2.3.6):

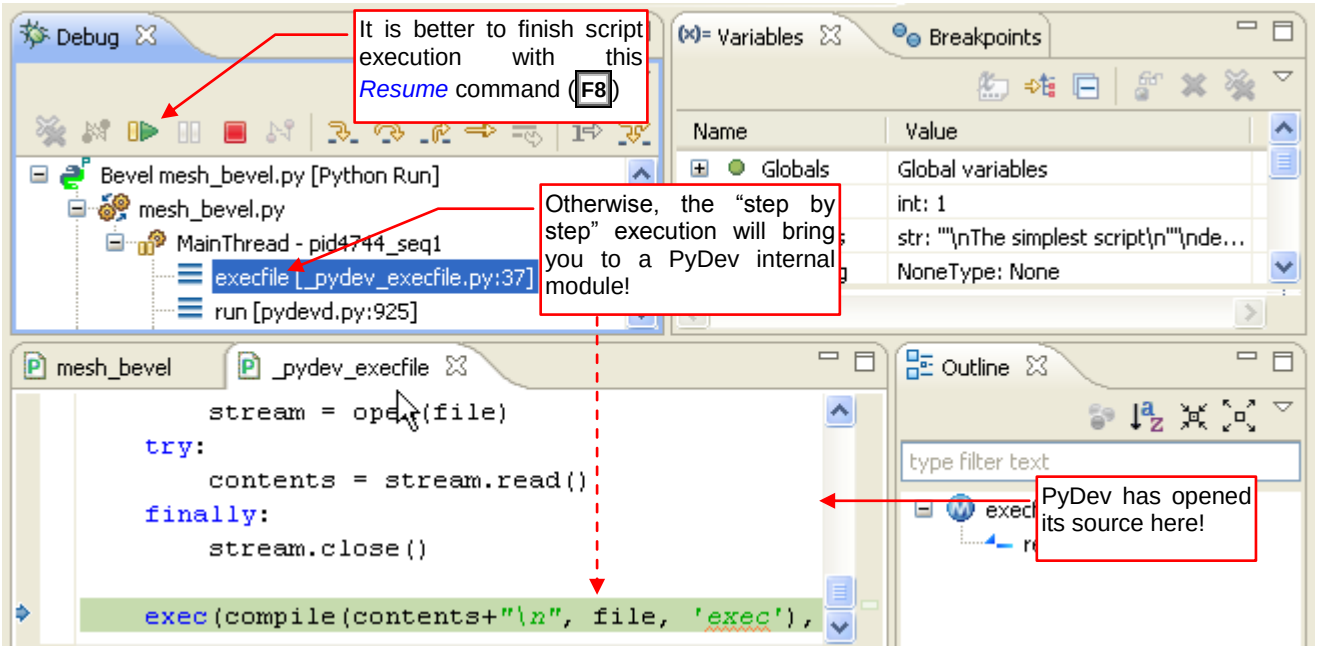


Figure 2.3.6 The process state after another *Step over* command (i.e. pressing the F6 button again)

This helper script implements tracking of user's programs. (Internally, PyDev uses its remote debugger here. We also will use it for the Blender scripts). It is better to press the F8 key (*Resume*) here, to finish the program.

Figure 2.3.7 shows how the debugger screen looks like, when the script execution is completed:

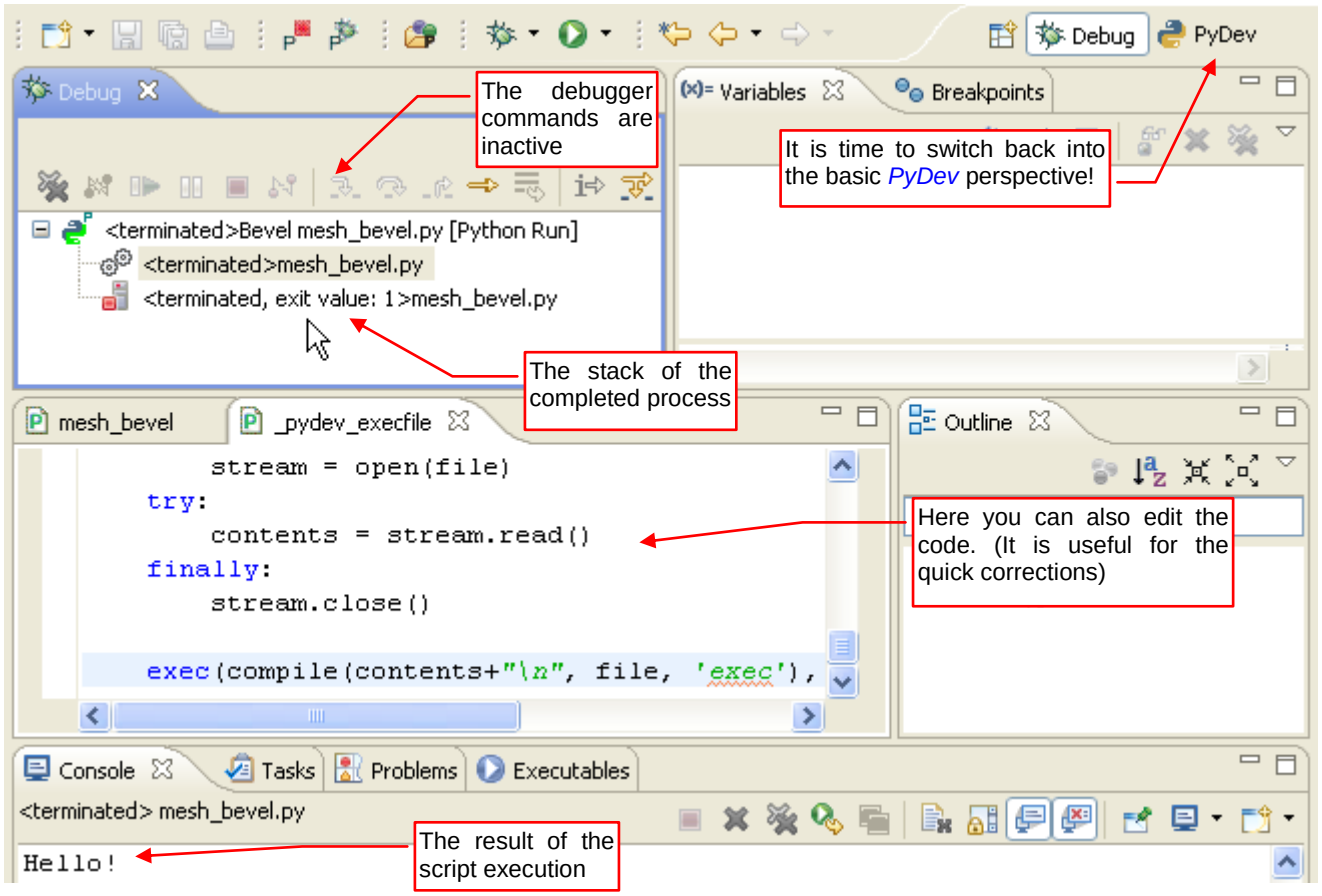


Figure 2.3.7 The state after the *Resume* command (F8) — the process is completed

You can make minor corrections of the code in the *Debug* perspective, using the editor pane. However, when you are going to make serious changes — switch to the *PyDev* perspective. You have more helper tools there (Figure 2.3.8):

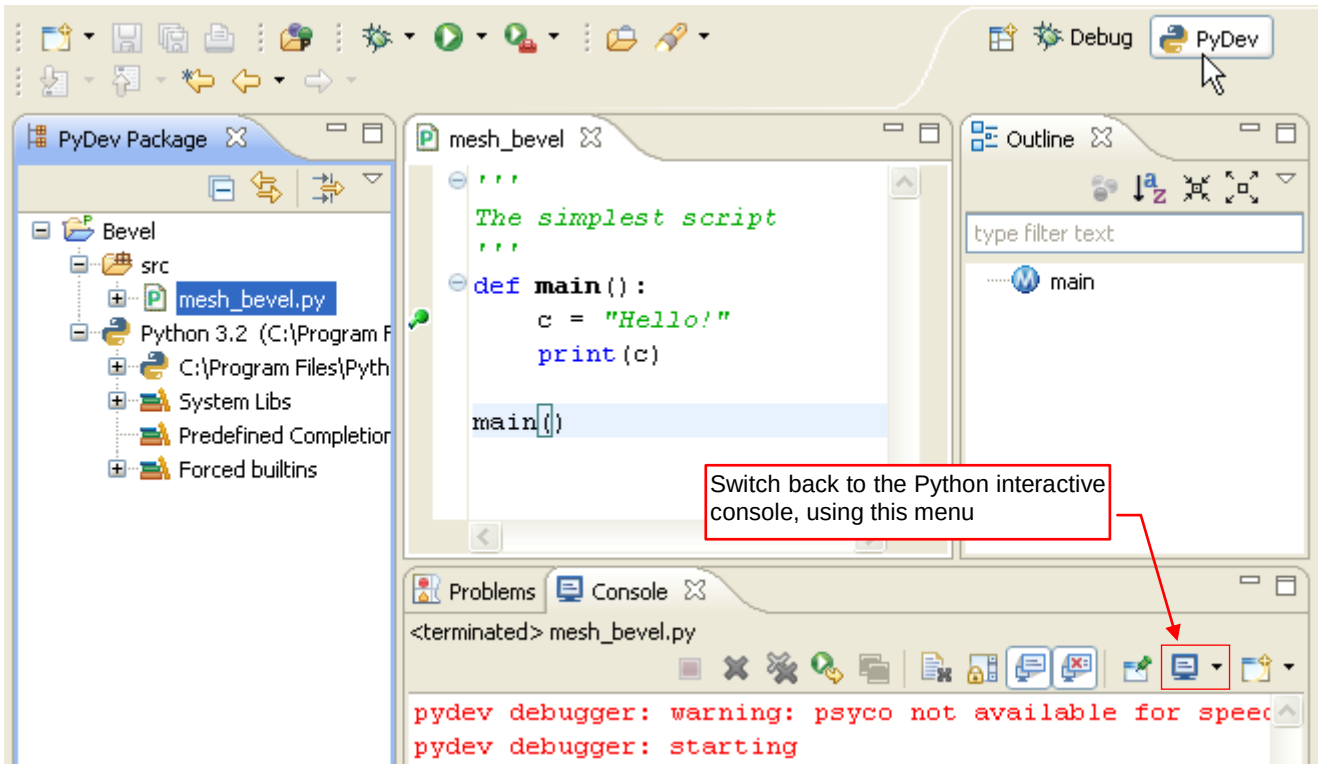


Figure 2.3.8 Back to the *PyDev* perspective — for the further work on the code

Summary

- You have learned, how to set breakpoints in your code (page 27);
- We have launched our script in the debugger (page 27). 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 28);
- We have looked at some helper debugger panes: ***Variables*** (page 28) and ***Stack*** (page 29);
- After the last line of your script, PyDev debugger is still executing its internal code (page 29). Therefore it is better at this point to *Resume* (F8) its normal execution;

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 begin with the 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). As a 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 bevels selected edges of a mesh. I used this example to show in practise all the details of developing Blender scripts in the Eclipse environment. You will also find here some tips, how to solve typical problems that you encounter during this process. One of them is finding in the Blender API the right class and operator that support the functionality we need! (I think that still nobody, except the few Blender API developers, is familiar with the whole thing…).

3.1 The problem to solve

In Blender 2.49, pressing the W key opens the *Specials* menu. You can invoke the ***Bevel*** command from there, to chamfer the selected mesh edges (Figure 3.1.1):

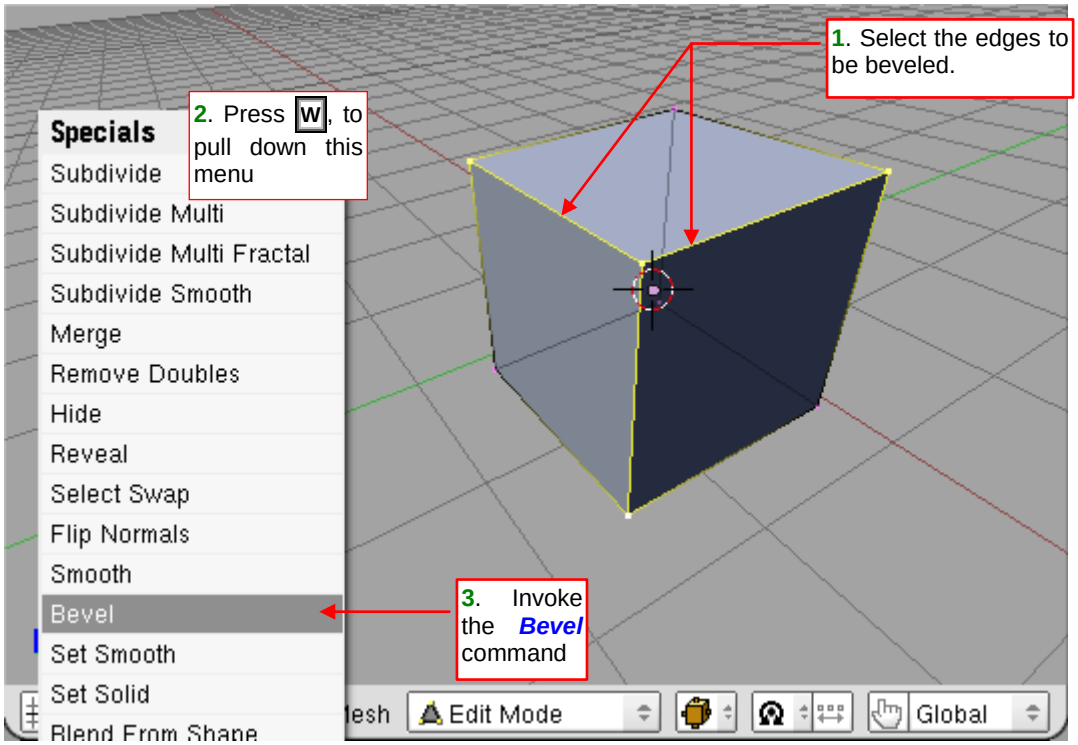


Figure 3.1.1 Blender 2.49 — invoking the *Bevel* command

In effect, you will see the bevels along selected edges. To change their width, just drag the mouse. To obtain a "rounded" bevel width value, hold the Ctrl key down (Figure 3.1.2):

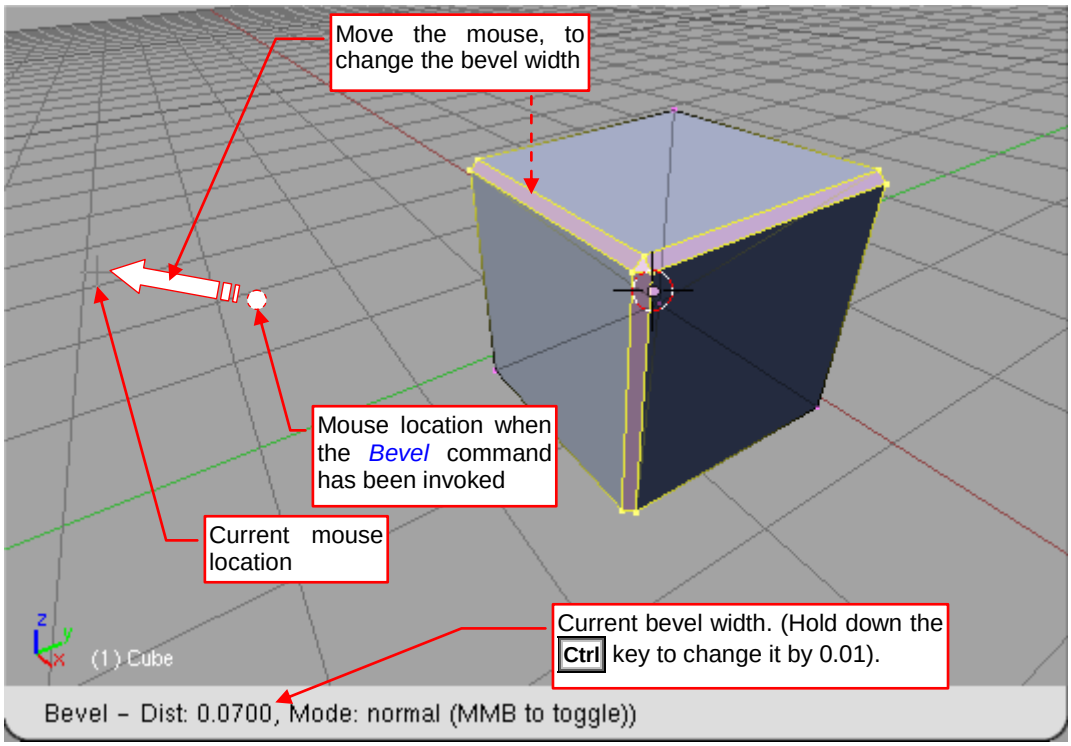


Figure 3.1.2 Blender 2.49 — setting the bevel width

Notice, that it is not possible to enter the exact, numerical width during this operation. It was a minor drawback of the *Bevel* command in Blender 2.49.

Clicking the LMB ends the *Bevel* operation. Where it is necessary, Blender adds to the result mesh additional edges (all its faces must have no more than 4 vertices). (Figure 3.1.3):

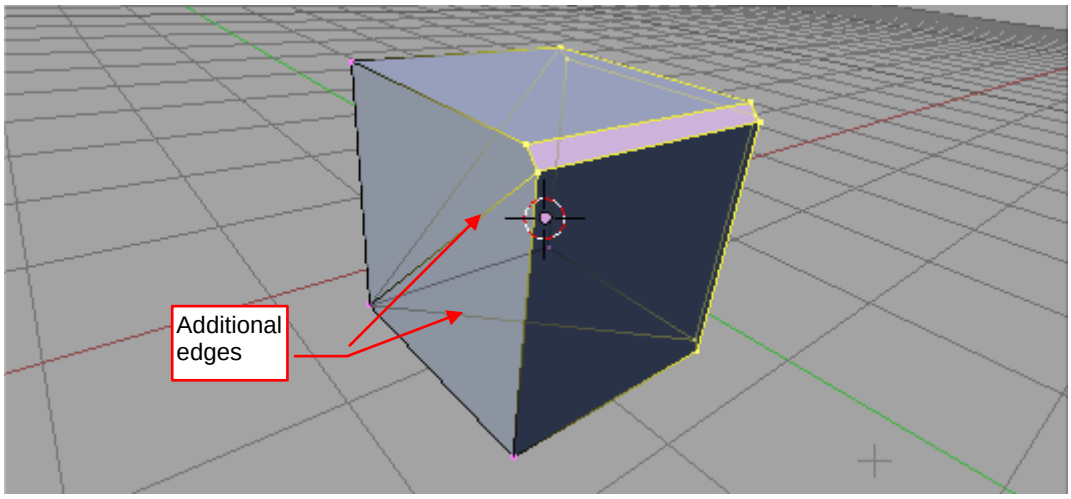


Figure 3.1.3 Blender 2.49 — result of the *Bevel* command

It's simple and quick, isn't it?

Blender 2.5 lacks such "destructive" *Bevel* command, so many users complain about it. It has only the Bevel modifier, which chamfers the mesh in a "non-destructive" way (Figure 3.1.4). (This modifier was available also in Blender 2.49):

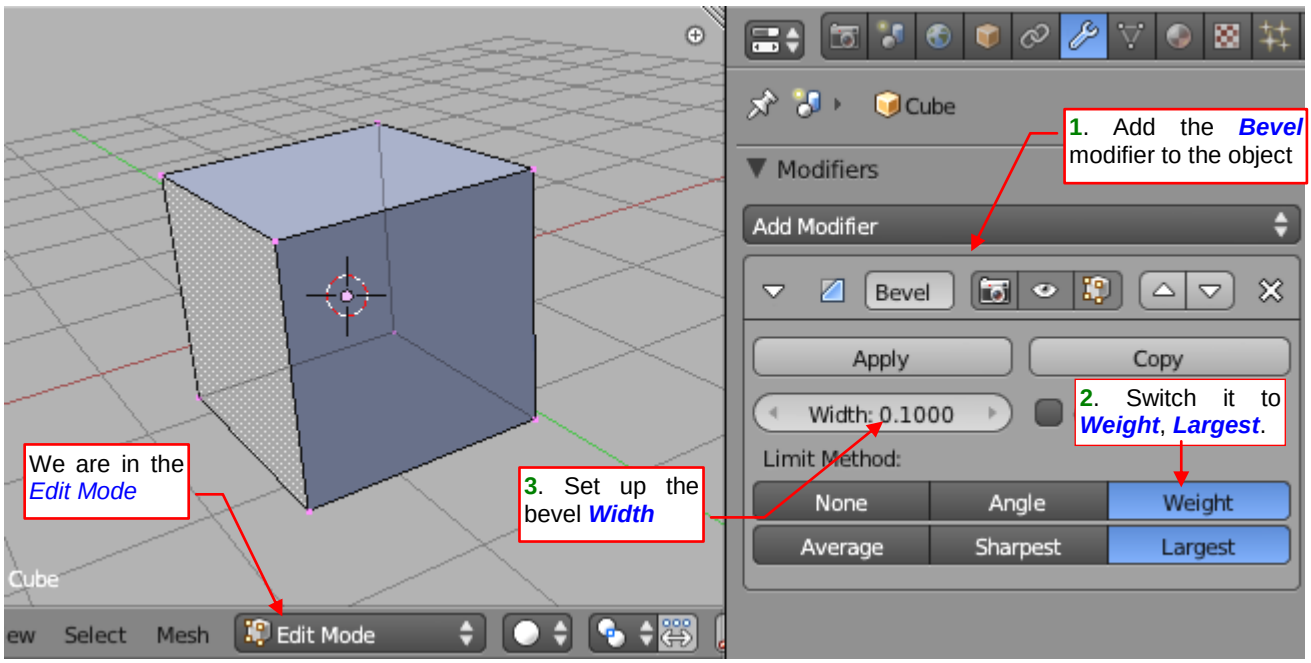


Figure 3.1.4 Blender 2.57 — adding the *Bevel* modifier

To obtain the same effect in Blender 2.5, you have to add the ***Bevel*** modifier to the mesh object. Initially, it will bevel all the edges of the mesh. However, if you switch the *Limit Method* to the ***Weight***, Blender will display another row of options on the modifier panel. Choose the ***Largest*** mode, from there. It will remove all the chamfers from the mesh, because initially all its edges have the *Bevel Weight* = 0. (This is the default value).

You can dynamically change the width of the *Bevel* modifier, dragging over the *Width* control the mouse with the LMB pressed (it is a slider). You may play around with it for a while. Set it at the end to the appropriate value (for example — 0.1 Blender Units, as it is shown in Figure 3.1.4).

How to change the *Bevel Weight* values of the selected edges? Open the *Toolbox* (T). In the *Mesh Options* panel, switch the *Edge Select Mode* into the ***Tag Bevel*** (Figure 3.1.5):

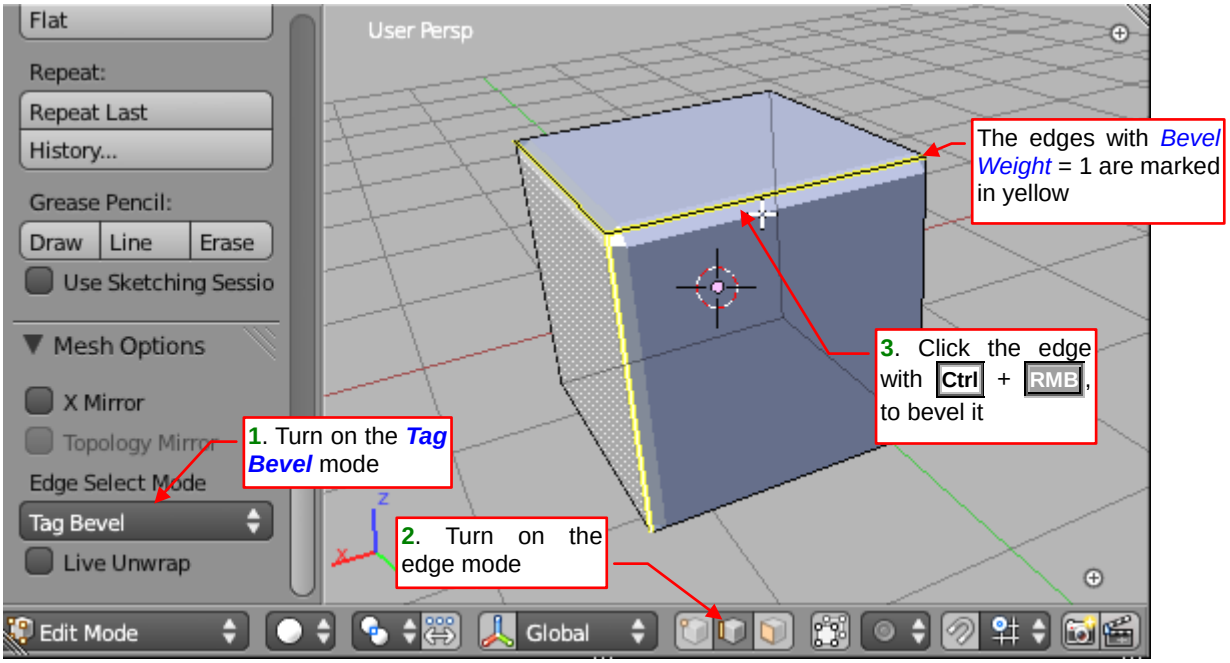


Figure 3.1.5 Beveling selected edges with the *Bevel* modifier

Turn on the edge selection mode of the mesh (Figure 3.1.5). Holding down the Ctrl key, click some edges with the RMB. Blender will add a bevel under each of them (by clicking, you are flipping their *Bevel Weght* to 1.0).

- Notice that the edges with the bevel tag are marked yellow. This helps you to figure out what is currently set on the mesh.

Using the *Bevel* modifier, you can have chamfered edges on the final shape, while the original cube mesh is not altered. This effect is useful in many cases, because it lets you to avoid overcomplicated meshes. Therefore, the beveling with the modifier is often referred as the "non-destructive" (in the opposite to the "destructive" *Bevel* command, which we have used in Blender 2.49).

To obtain the “real” beveled edges in Blender 2.5, as in the “destructive” command from Blender 2.49, we have to ***Apply*** the *Bevel* modifier (Figure 3.1.6):

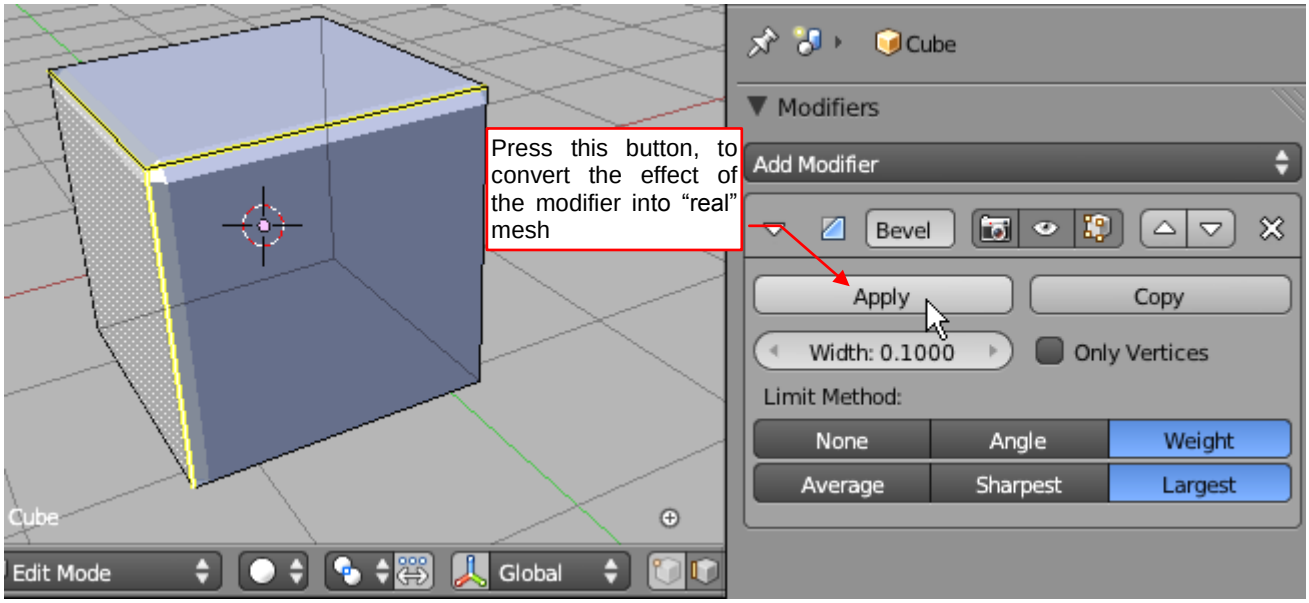


Figure 3.1.6 *Apply*ing the modifier

When you press the *Apply* button (do it in the *Object Mode*!) the modifier will disappear, and its beveled edges become the real part of the mesh. Now you can do with them what you want (Figure 3.1.7):

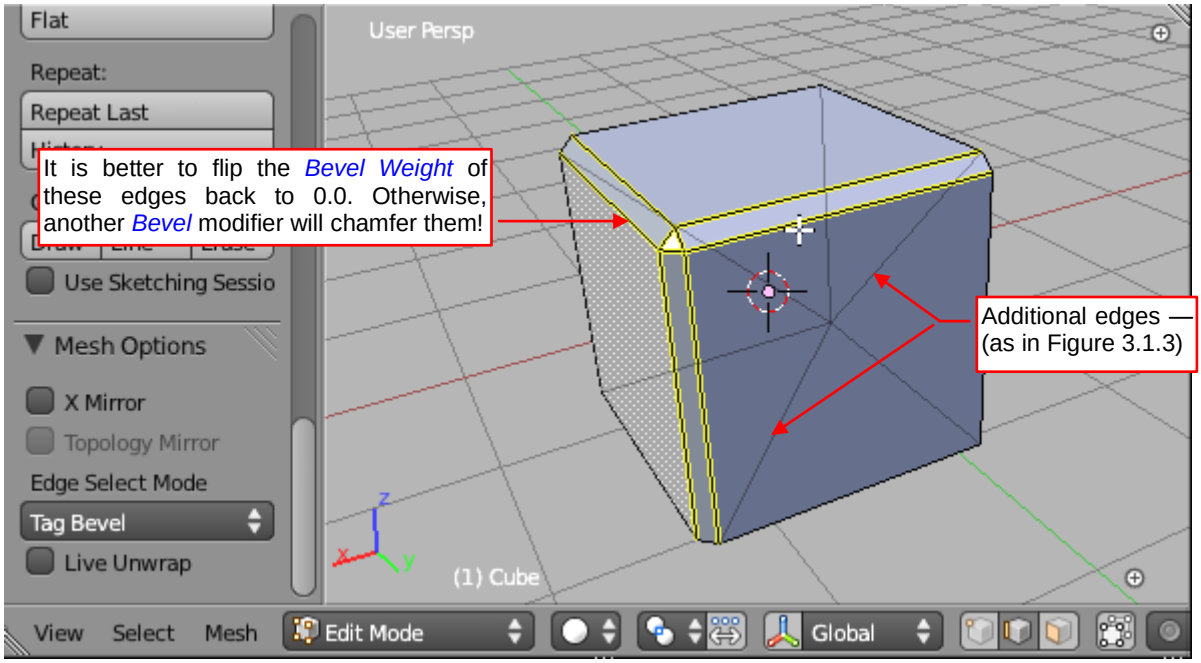


Figure 3.1.7 Removing the bevel weights that are left after the modifier

On the end, you should click (Ctrl-RMB) the yellow edges that are left after this operation. It will remove their bevel tags, making the mesh ready for the eventual another *Bevel* modifier (Figure 3.1.8):

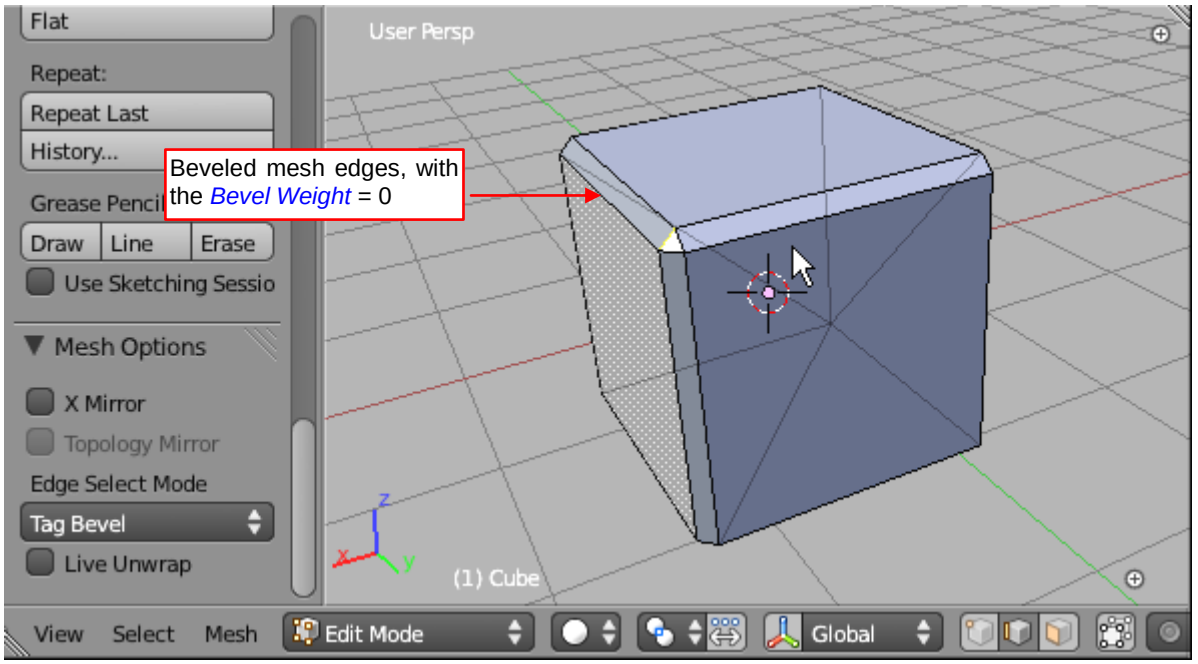


Figure 3.1.8 Blender 2.57 — result of the *Bevel* modifier application

You have to admit that there was a lot of "clicking". Although the Bevel modifier has also its advantages, many Blender 2.5 users would like to have also a simple, “destructive" Bevel.

In this chapter, we will write a Blender script that will use the *Bevel* modifier to create the "destructive" version of this operation. In general, it will repeat the sequence of steps that I did manually in this section. In the next chapter, we will convert this script into a professional Blender add-on.

Summary

- Blender 2.5 lacks the “destructive” ***Bevel*** command. Such command was available in the previous Blender version (2.49 — see page 34);
- You can obtain the same “destructive bevel” effect in Blender 2.5 by applying its Bevel modifier (page 35 - 37). To not repeat these operations manually, we will create a script that will execute them all at once. In this way we will add to Blender 2.5 the missing functionality;

3.2 Adapting Eclipse to the Blender API

To write scripts for Blender in an easier way, we need to "teach" PyDev the Blender API. The code autocompletion should be able to suggest object methods and fields, just as it does for the standard Python modules. Fortunately, PyDev has such a possibility. We have just to provide it a kind of simplified Python file that contains only declarations of the 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, PyDev requires them to have the **.pypredef* extension (a derivate from "*Python predefinition*").

I modified Campbell Barton's script, which generates the Python API documentation (the one that you can see on the wiki.blender.org). Using it, I was able to create the appropriate **.pypredef* files for the entire Blender API, except the ***bge*** module. You can find them in the data that accompanies this book. Just download the http://airplanes3d.net/downloads/pydev/pydev-blender.zip file and unzip it into the folder with Blender binaries (Figure 3.2.1):

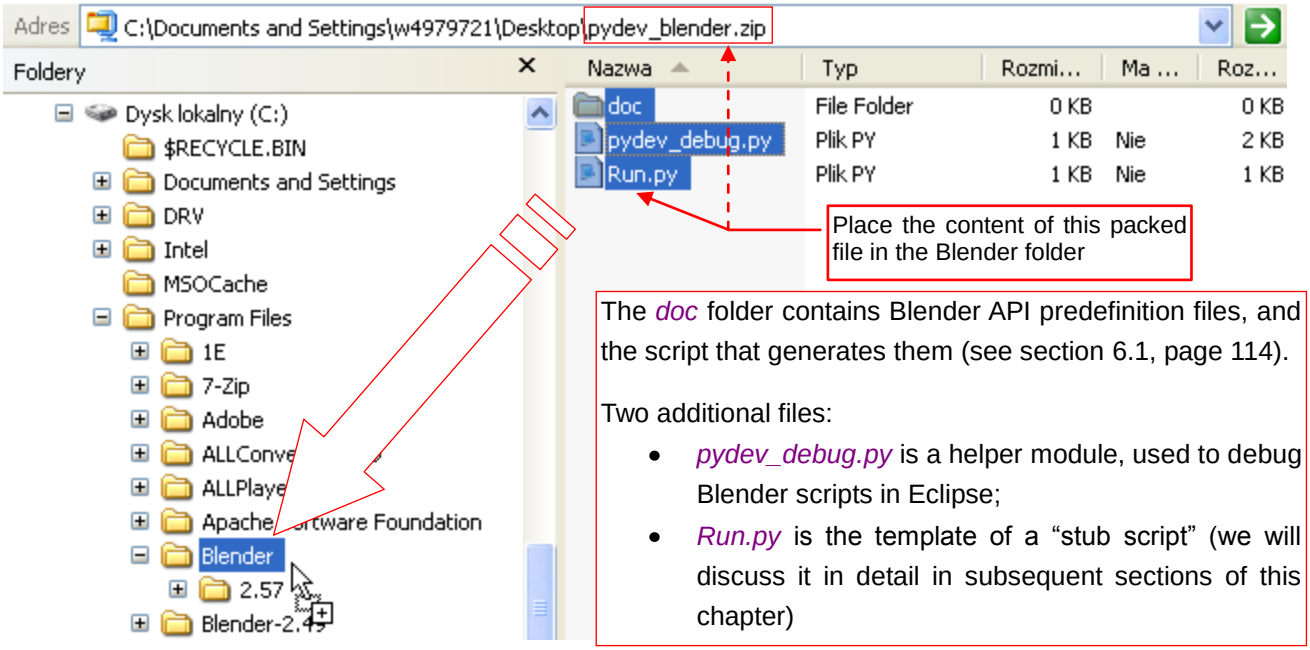


Figure 3.2.1 Unpacking additional files to the Blender folder

- Place both **.py* files and *doc* folder in the directory that contains the *blender.exe* executable (Figure 3.2.2):

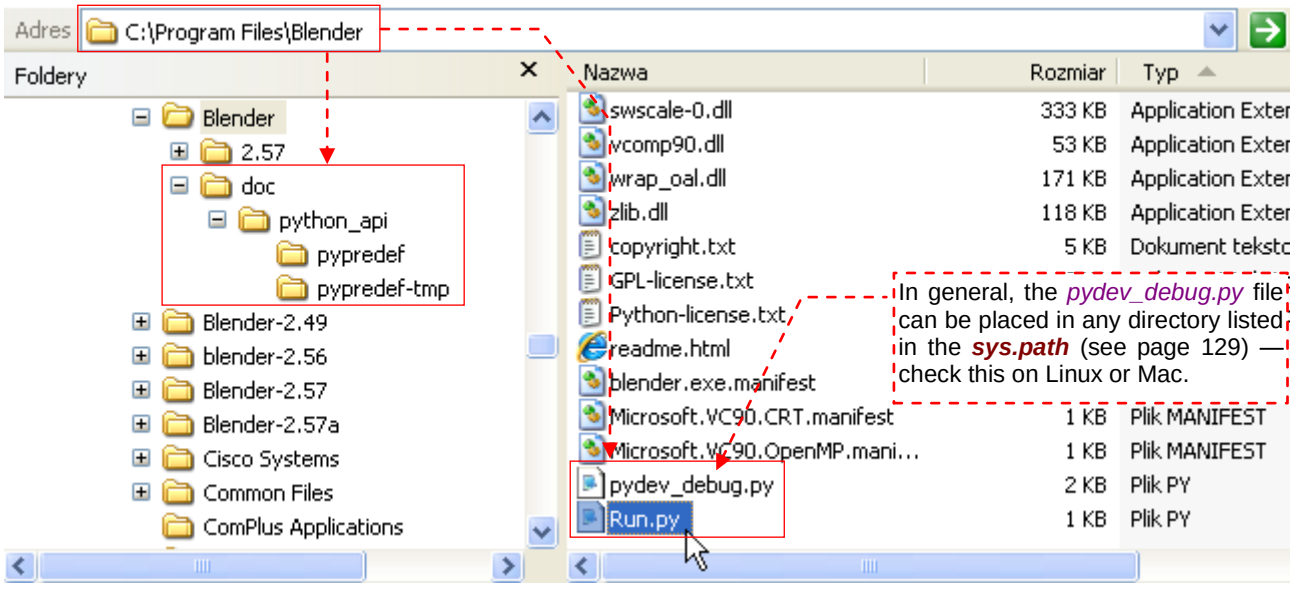


Figure 3.2.2 The files required to follow this book

When the predefinition files are in place, we need to alter the project configuration. To do it, invoke the ***Project→Properties*** command (Figure 3.2.3):

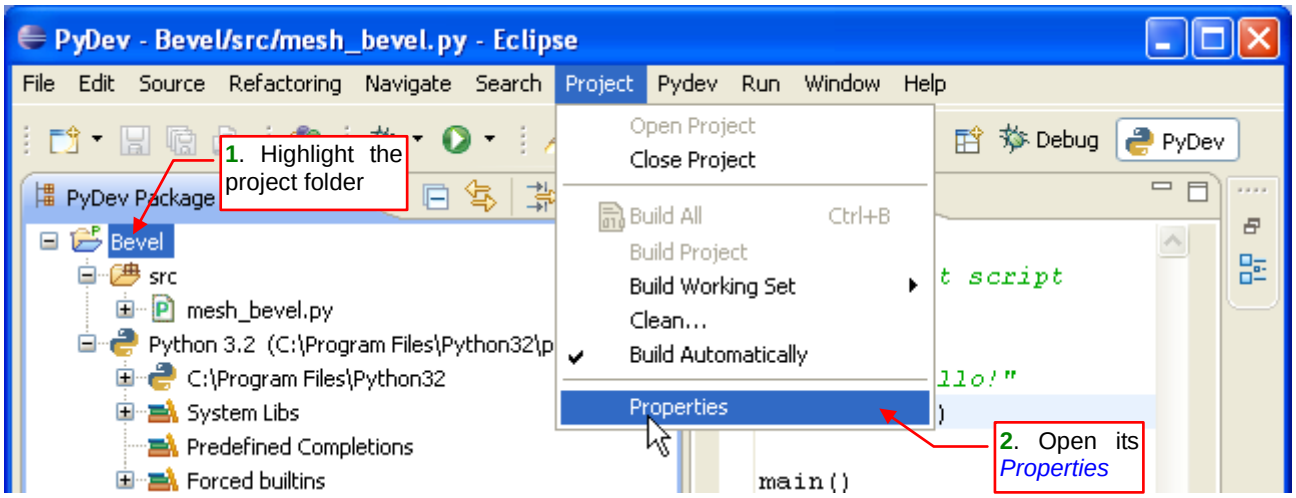


Figure 3.2.3 Opening the project configuration window

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

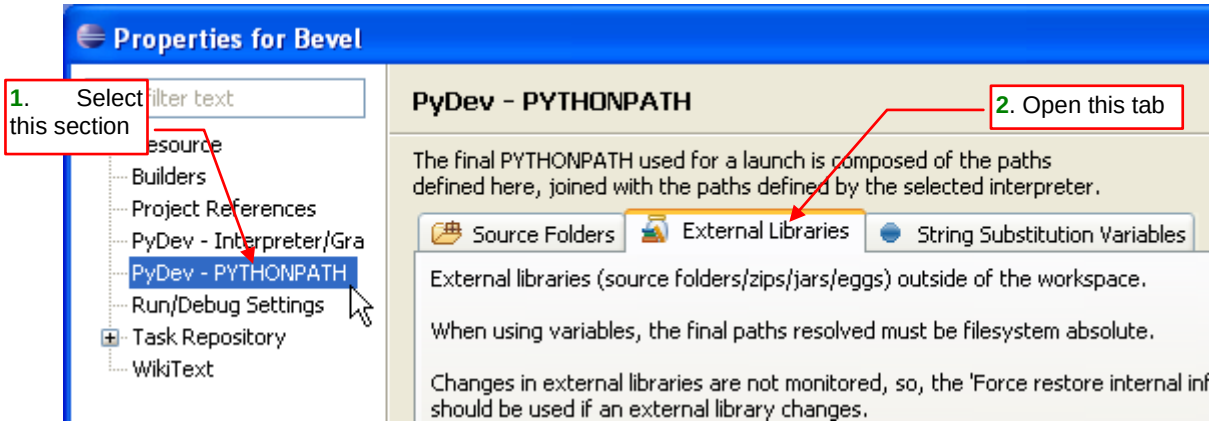


Figure 3.2.4 Navigating to the *PyDev - PYTHONPATH:External Libraries* pane

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

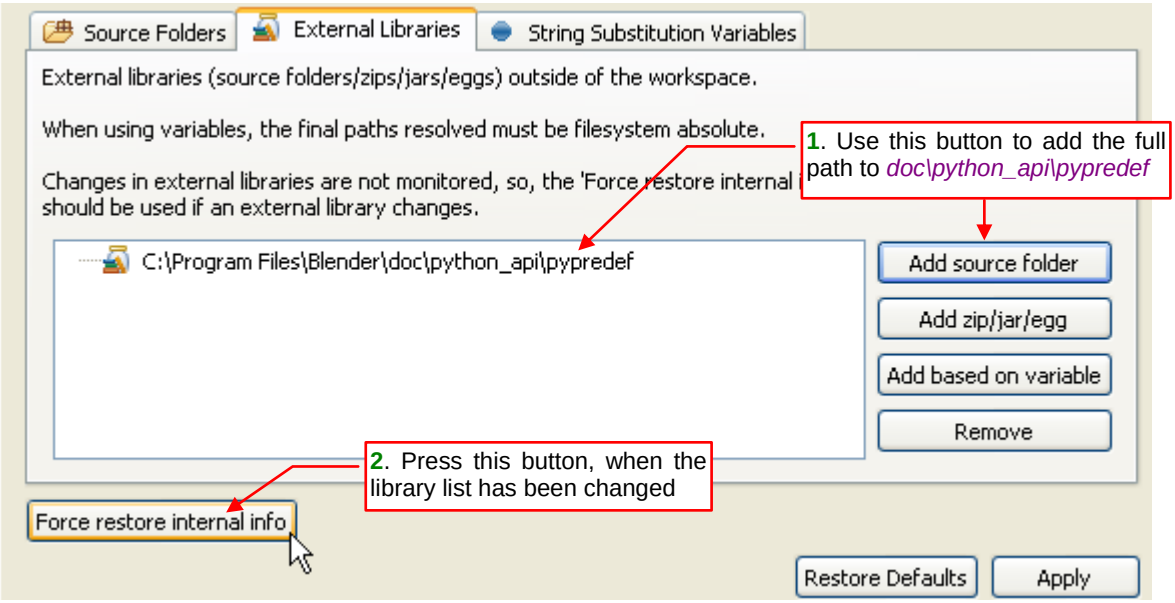


Figure 3.2.5 PyDev *PYTHONPATH* configuration

After every change made to PyDev *PYTHONPATH*, make sure that you have pressed the ***Force restore internal info*** button (Figure 3.2.5). In response, Eclipse will display for a few seconds information in the status bar about the progress of this process[1].

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

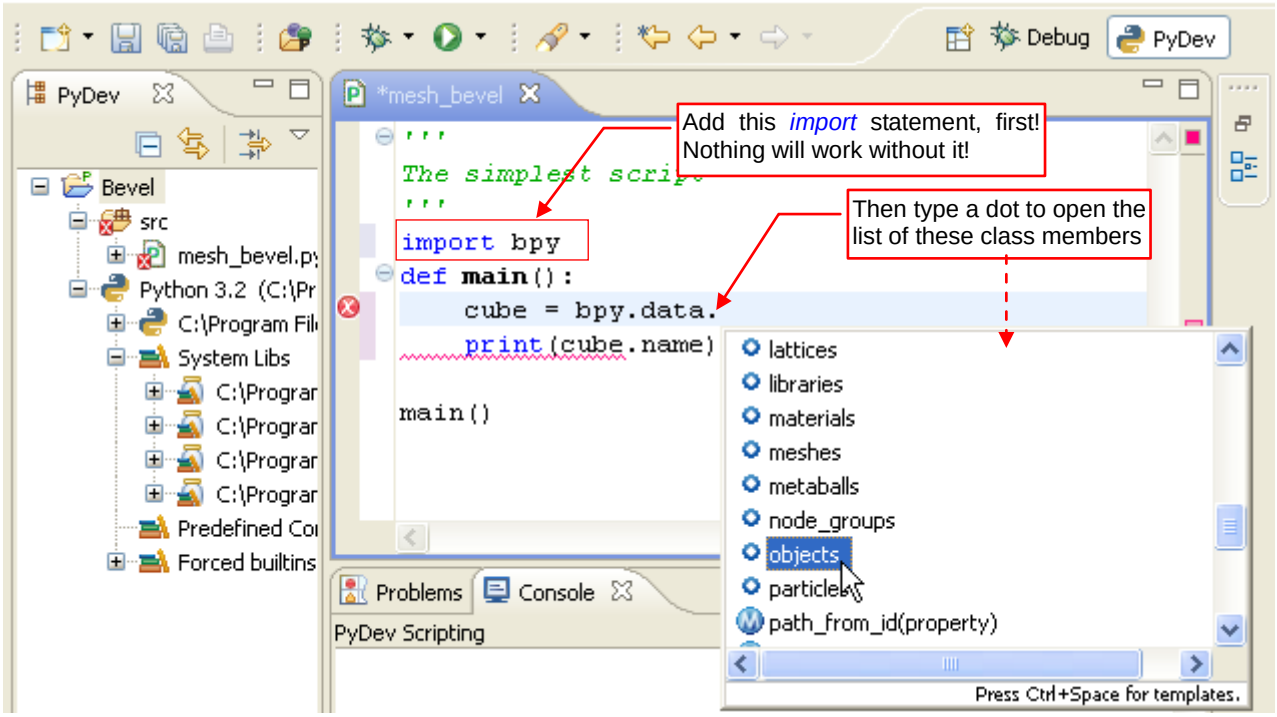


Figure 3.2.6 Code autocompletion for the Blender API

The list of the class members appears after typing 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)

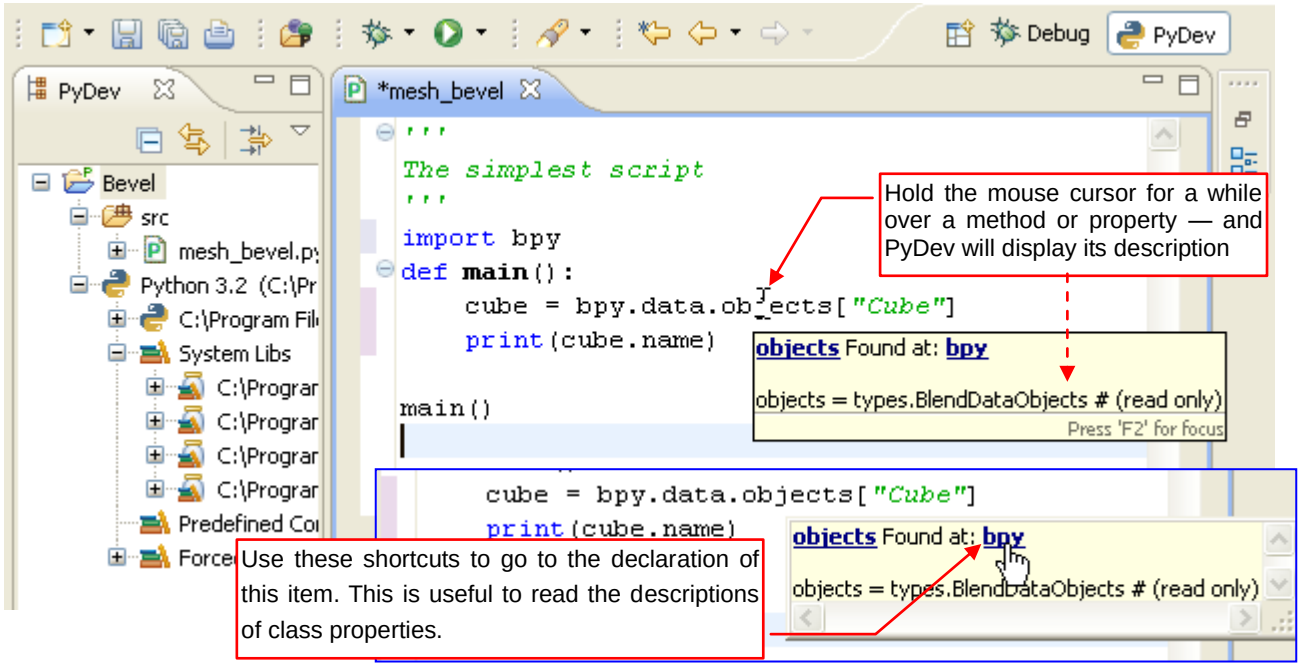


Figure 3.2.7 Displaying the descriptions

[1] This method of using **.pypredef* files differs from the one that is described in the documentation on the www.pydev.org. The problem is that following this "official" version (adding the folder to the *Predefined Completions* list) I could not obtain the proper code completion!

The tooltip with method description disappears, when you move the mouse outside. You can also click on the reference link, placed in its first line (see Figure 3.2.7). This link opens the source file on the line with appropriate declaration (Figure 3.2.8):

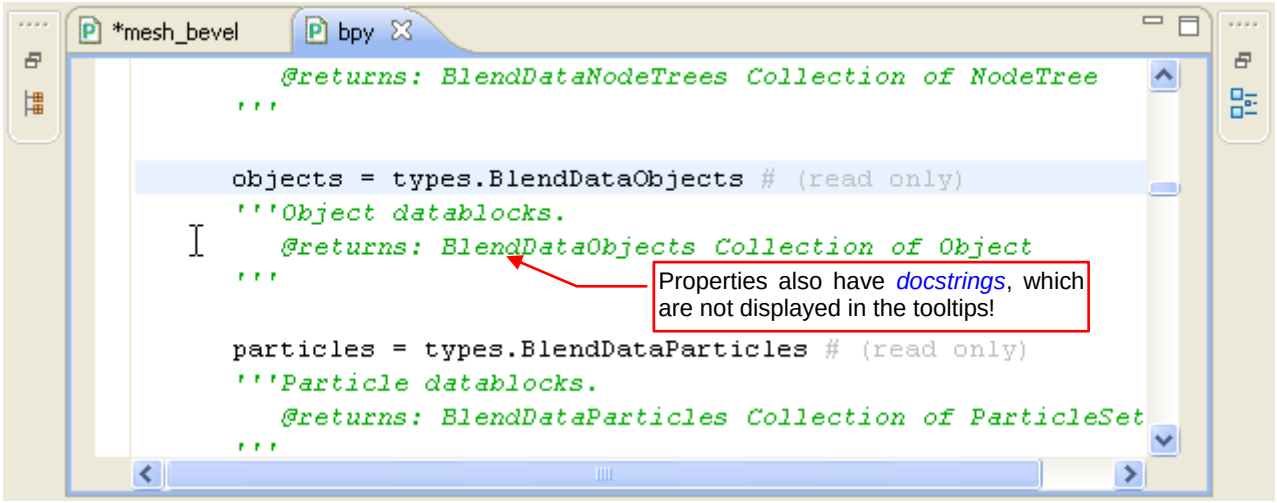


Figure 3.2.8 Property declaration in the predefinition file (*bpy.pypredef*), opened using the tooltip reference link

From the PyDev point of view, such a declaration is located in the predefinition file (*bpy.pypredef*). That's why it is opened as the source code. You can use this effect to read more about a particular class property (field). The tooltip displays so called "documentation string" (*docstring*), placed just below the function (method) declaration. Unfortunately, the Python standard does not provide *docstrings* for any kind of variables. (The class or object field is for the Python interpreter just a variable). Thus, using the tooltip link to the declaration of the field is the quickest way to read its description. (Most of the Blender API fields have *docstrings*).

By the way, if you have opened the ***bpy*** module, look at its structure in the *Outline* pane (Figure 3.2.9):

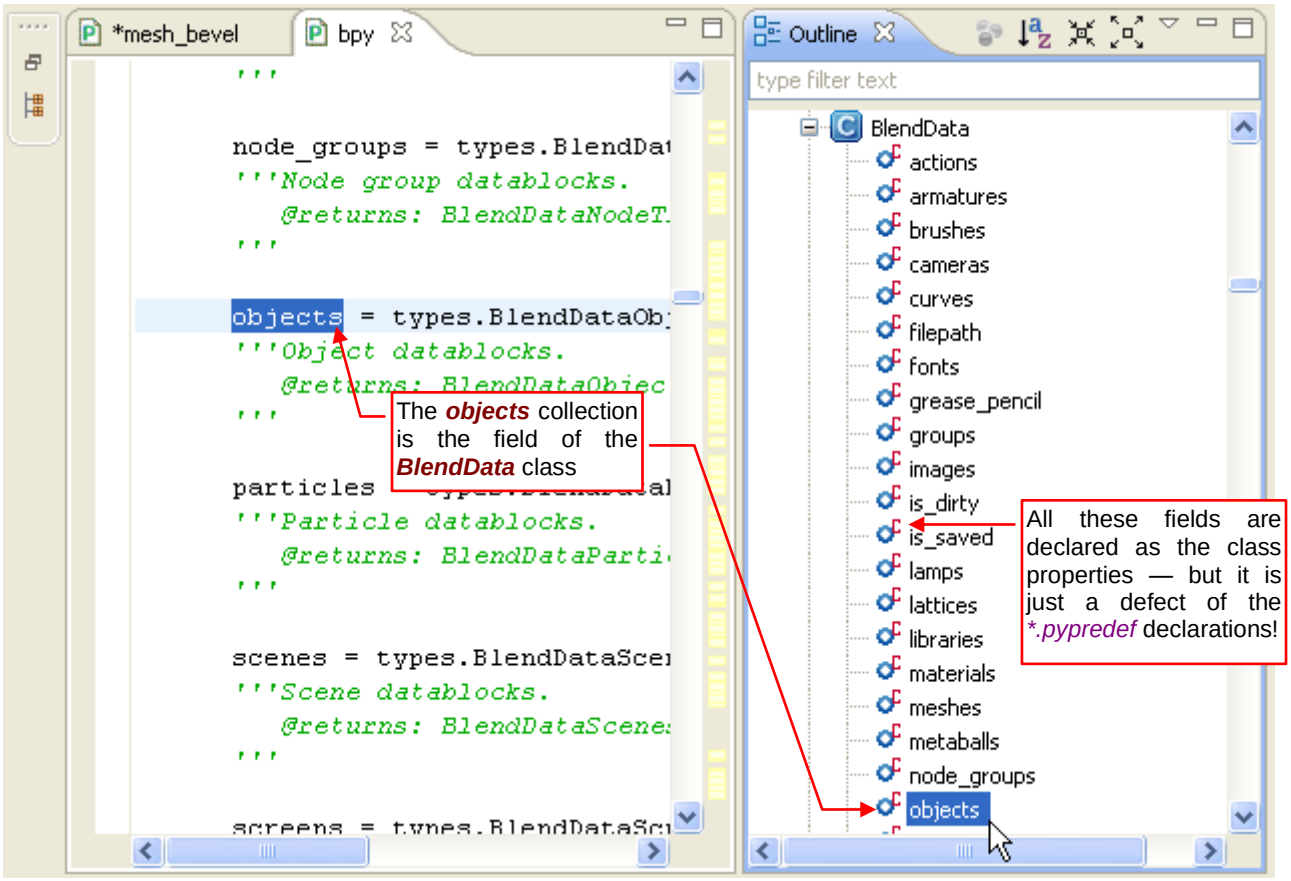


Figure 3.2.9 A fragment of the Blender API structure, shown in the *Outline* panel

Notice that the structure of the ***bpy*** module, visible in the *Outline* pane, may be something of a "training aid". You can interactively "walk around" the whole Blender Python API, here. I would propose to start such a "trip" with collapsing this tree to its root nodes (Figure 3.2.10):

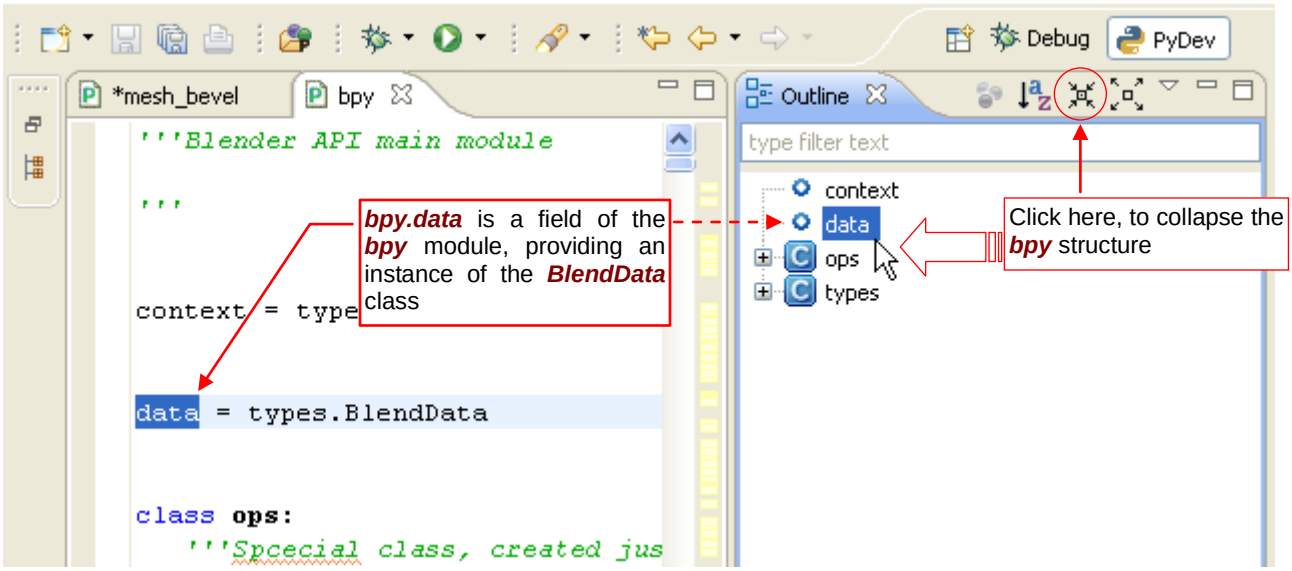


Figure 3.2.10 The root structure of the Blender API

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 one type of objects (***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 command is 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 order 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):

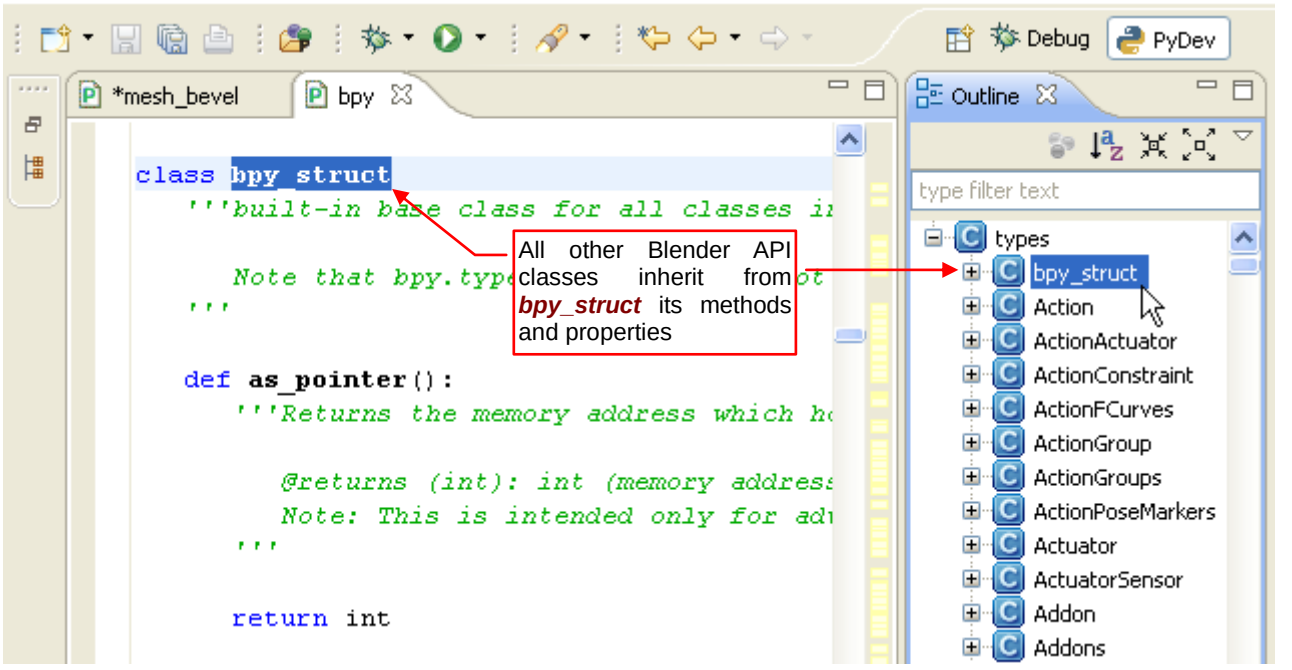


Figure 3.2.11 *bpy_struct*: **the base class of all Blender API classes**

Another thing is that ***bpy_struct*** is a "fake" class. In fact, it is a C-language structure that lies behind the API implementation. That is why its methods may not be fully implemented in the derived classes. For example — ***bpy_struct*** has a set of collection methods, like ***items()***. All collection classes (for example — ***MeshEdges***, the collection of ***MeshEdge*** objects) reuse it and implement only their specific methods, like ***add()*** (Figure 3.2.12):

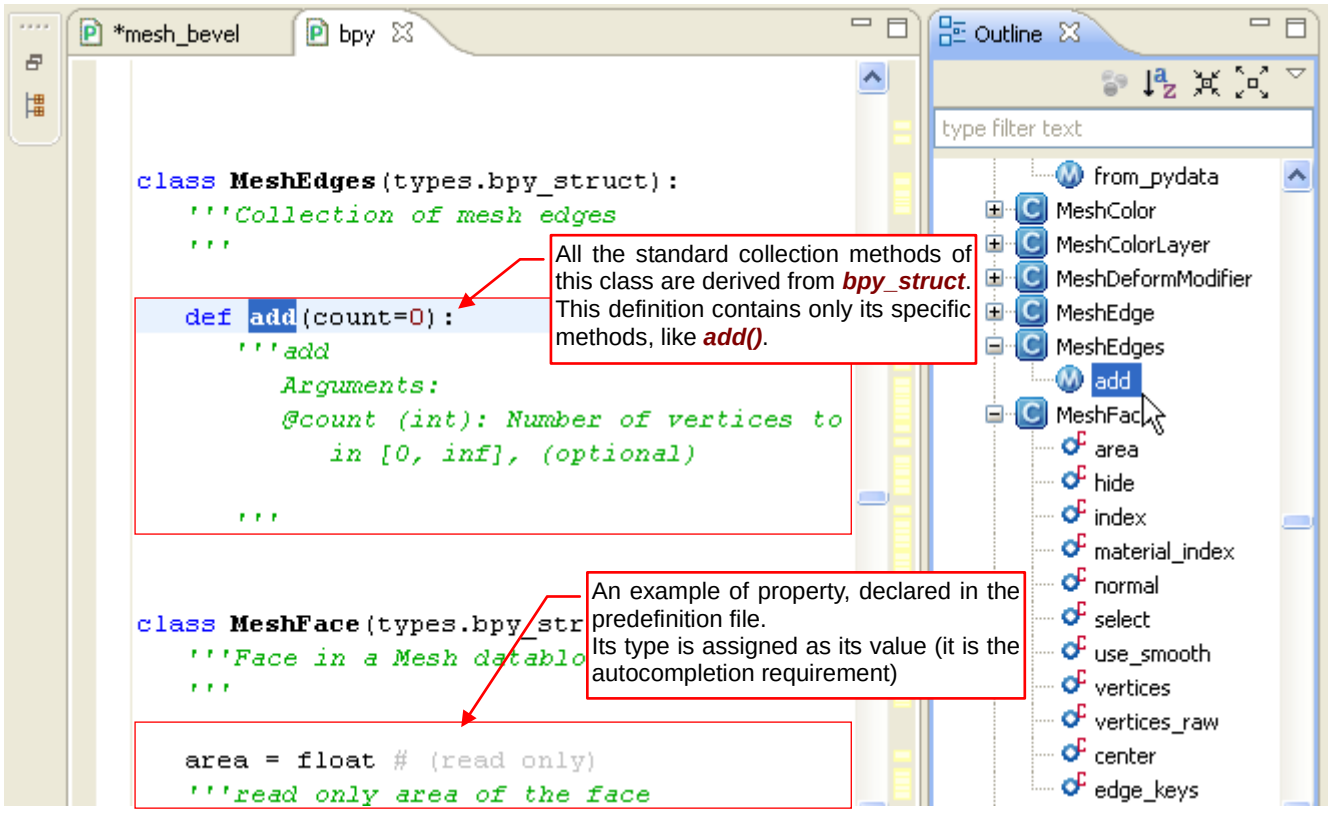


Figure 3.2.12 Derived Blender API classes — the declaration of their methods and properties

Of course, all the classes that represent the single elements (like ***MeshEdge***) have their ***items()*** method empty (and also 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 definition, 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):

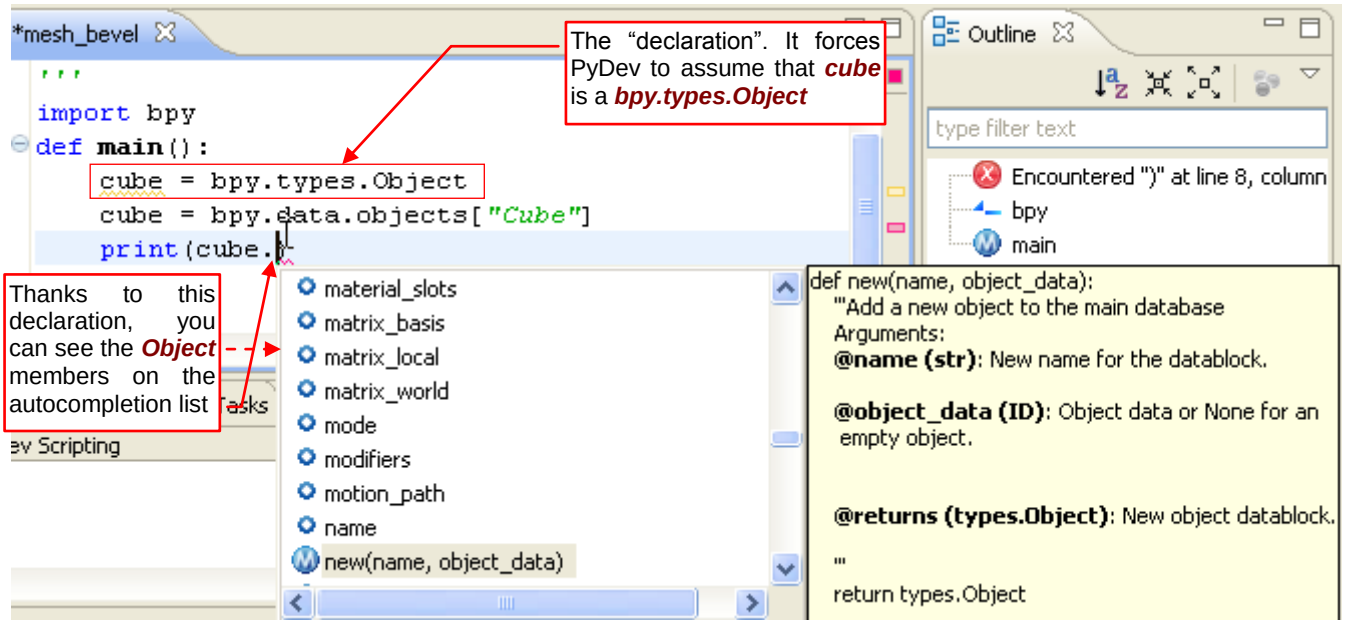


Figure 3.2.13 "Variable declaration" — a workaround of the Blender API collection type problem

In practice, you should add such "declaration line" only for a moment, when you need to use the automatic code completion. Always place it in the code 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 detect such lines, because it treats them as "unused variables". It marks them with appropriate warning (Figure 3.2.14):

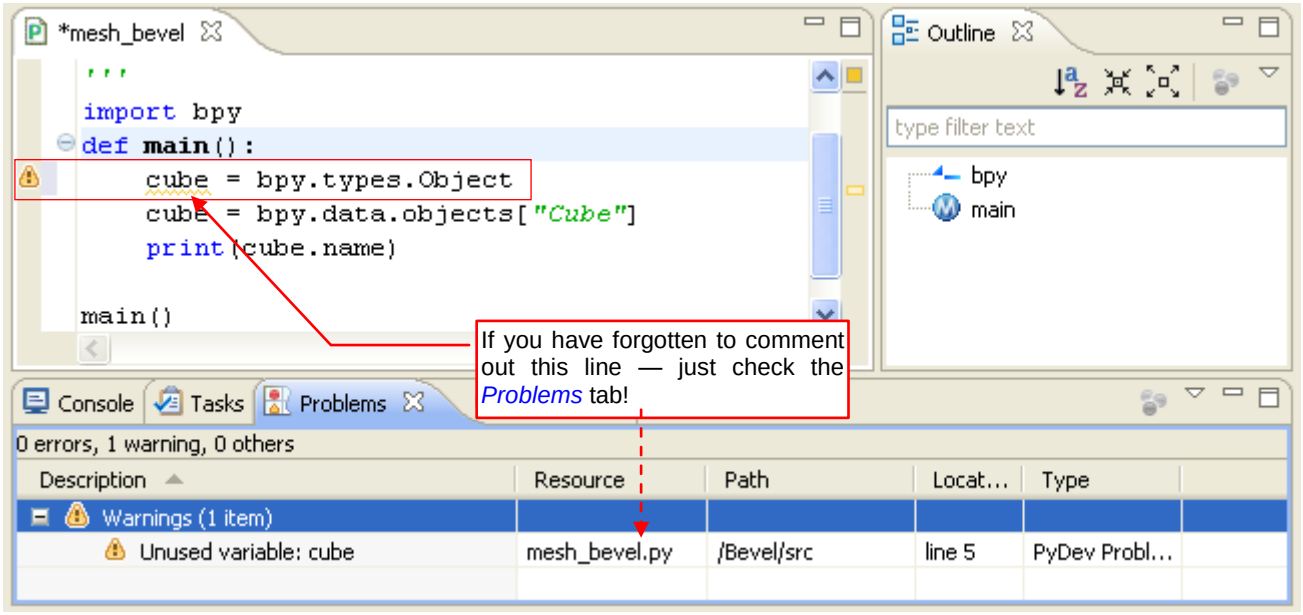


Figure 3.2.14 PyDev warnings at each "type declaration" line

It is a good practice to look into to the ***Problems*** tab, from time to time. You will see there all the lines, which you have forgotten to comment. Using this list, you will be able to fix them immediately.

To quickly figure out where in the entire API hierarchy is a specific field or method, highlight its name in the editor and open its context menu (RMB). Invoke the ***Show In→Outline*** from there. In response, PyDev will highlight the appropriate element in the *Outline* pane (Figure 3.2.15):

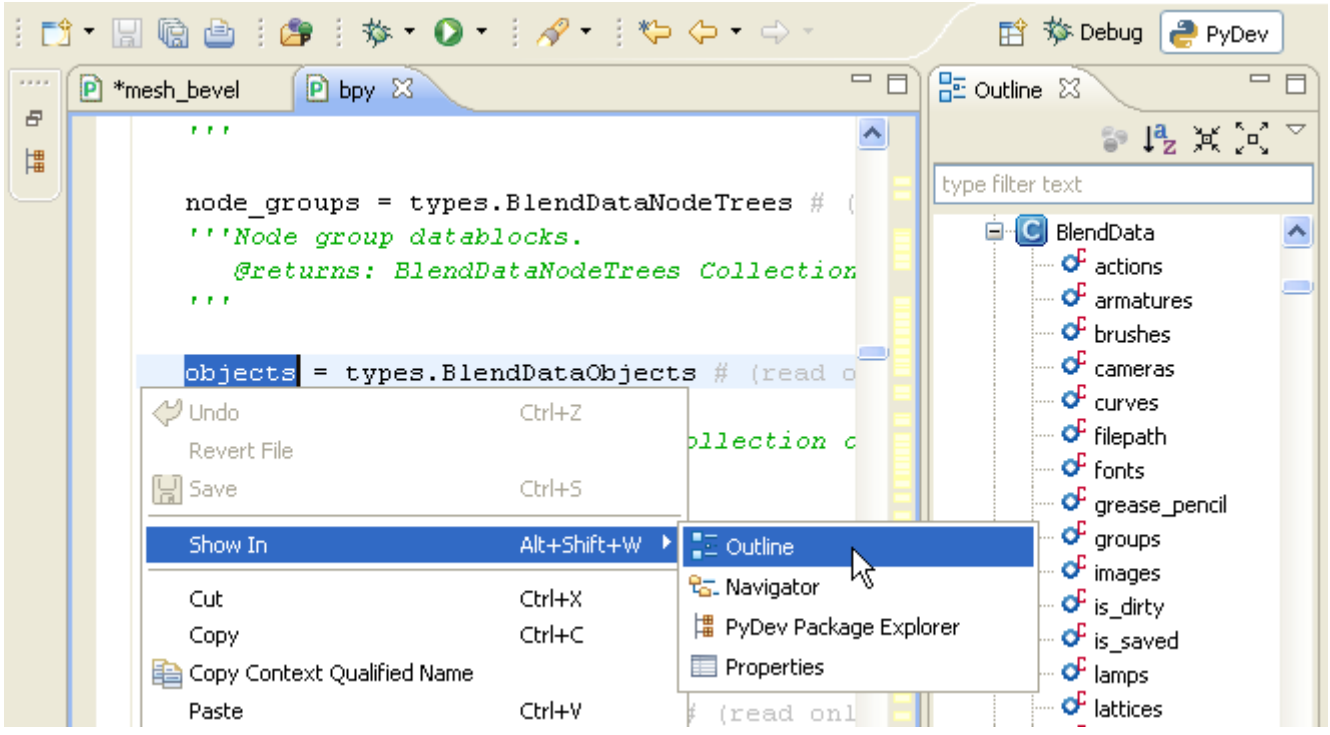


Figure 3.2.15 Finding a member in the hierarchy of *bpy* classes.

So far we have discussed the ***bpy.types*** branch, only. What about operators (***bpy.ops***)? There are plenty of them! To not get lost among them right now, browse their modules (classes), first: ***action***, ***anim***, ***armature***, … and so on. Let's expand the ***bpy.ops.brush*** module (Figure 3.2.16):

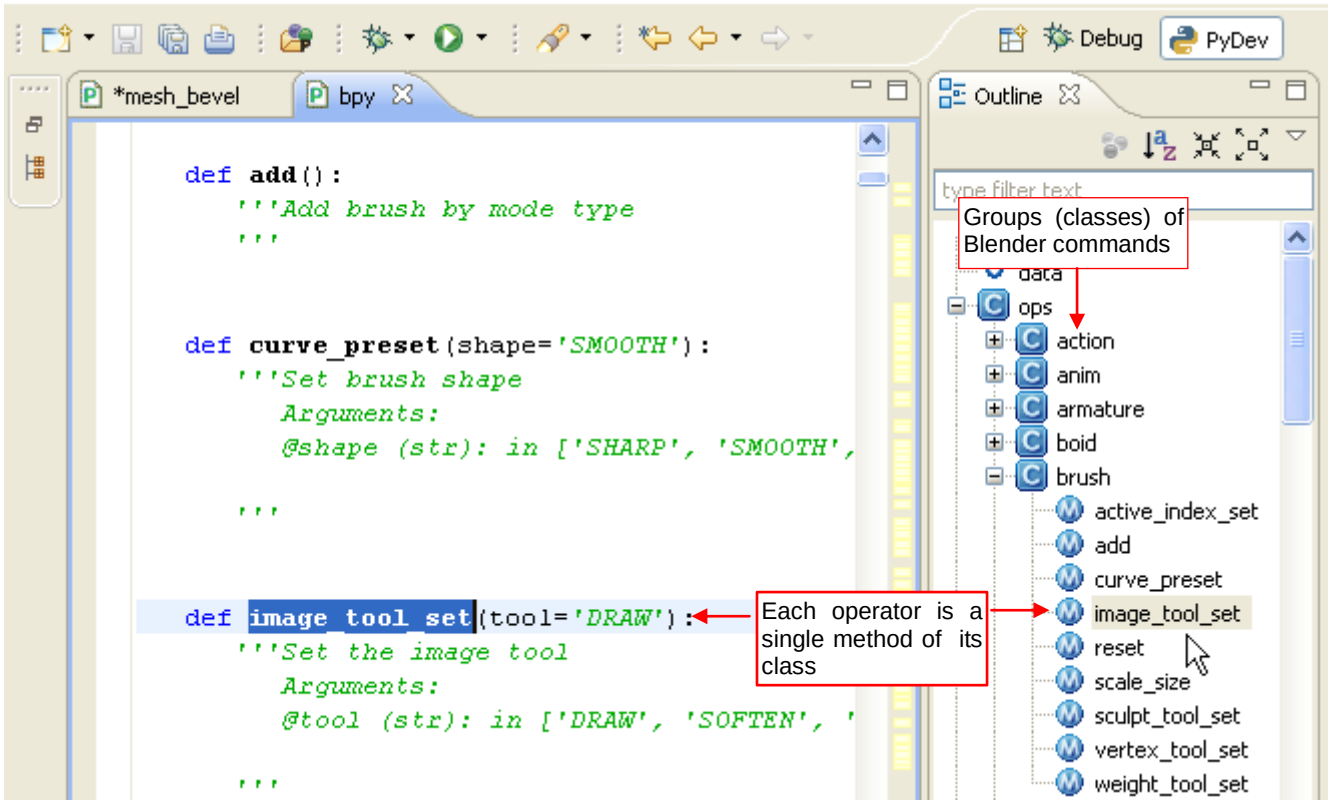


Figure 3.2.16 Example of operator declaration

Each operator module (***bpy.ops.brush***, for example) is declared as a separate class, which has many methods. Each of these methods is an operator. Note that you can always invoke every operator with no arguments — because each of these arguments is named and optional (i.e. has the default value).

It seems that this section has become an introduction to the Blender API architecture. To finish 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***), because contain just a few classes and/or functions:

bpy.app	various information about the program itself: version number, the path to the executable file, compiler flags, etc;
bpy.path	helper methods for working with paths and files (similar functionality like in the ***os.path*** standard module);
bpy.props	function to create the new class properties, which Blender can display on the panels (when it is 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;
mathutils	classes that represent 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.);
bgl	functions that allow scripts to draw directly on the Blender windows (in fact, it contains most of the OpenGL 1.0 methods);
blf	functions that draw the texts on the Blender screen;

I know little about the two remaining modules: ***aud*** (*Audio*) and ***bge*** (*Blender Game Engine*), so I will not elaborate about them.

Summary

- The Python predefinition files (**.pypredef*) allow to extend the scope of automatic code completion. The predefinition files for nearly all Blender API modules are included in the data accompanying this book (page 39);
- To use the predefinition (**.pypredef*) files, add their folder to the **PYTHONPATH** variable of the PyDev project (page 40);
- To let PyDev automatically complete Blender API expressions, add the "***importy bpy***" statement at the beginning of your script (page 41);
- The tooltips with detailed descriptions of methods can be used for the further exploration of The Blender API (page 41);
- The Python standard does not allow having *docstrings* about fields (class properties). The workaround for it is to use the link to their declaration, placed by PyDev on the first line of each tooltip. It opens the predefinition file in the Eclipse editor. You can read from there the description of the selected class field (page 42).
- Browsing the structure of the ***bpy*** module in the *Outliner* pane helps you to learn the Blender API (page 44);
- In case of the elements from a Blender API collection, use "variable declarations" (page 44) to obtain the correct autocompletion;

3.3 Developing the core code

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

Let’s prepare a Blender file for the script tests. I would propose to use for this the default cube, with the screen layout set as shown in Figure 3.3.1:

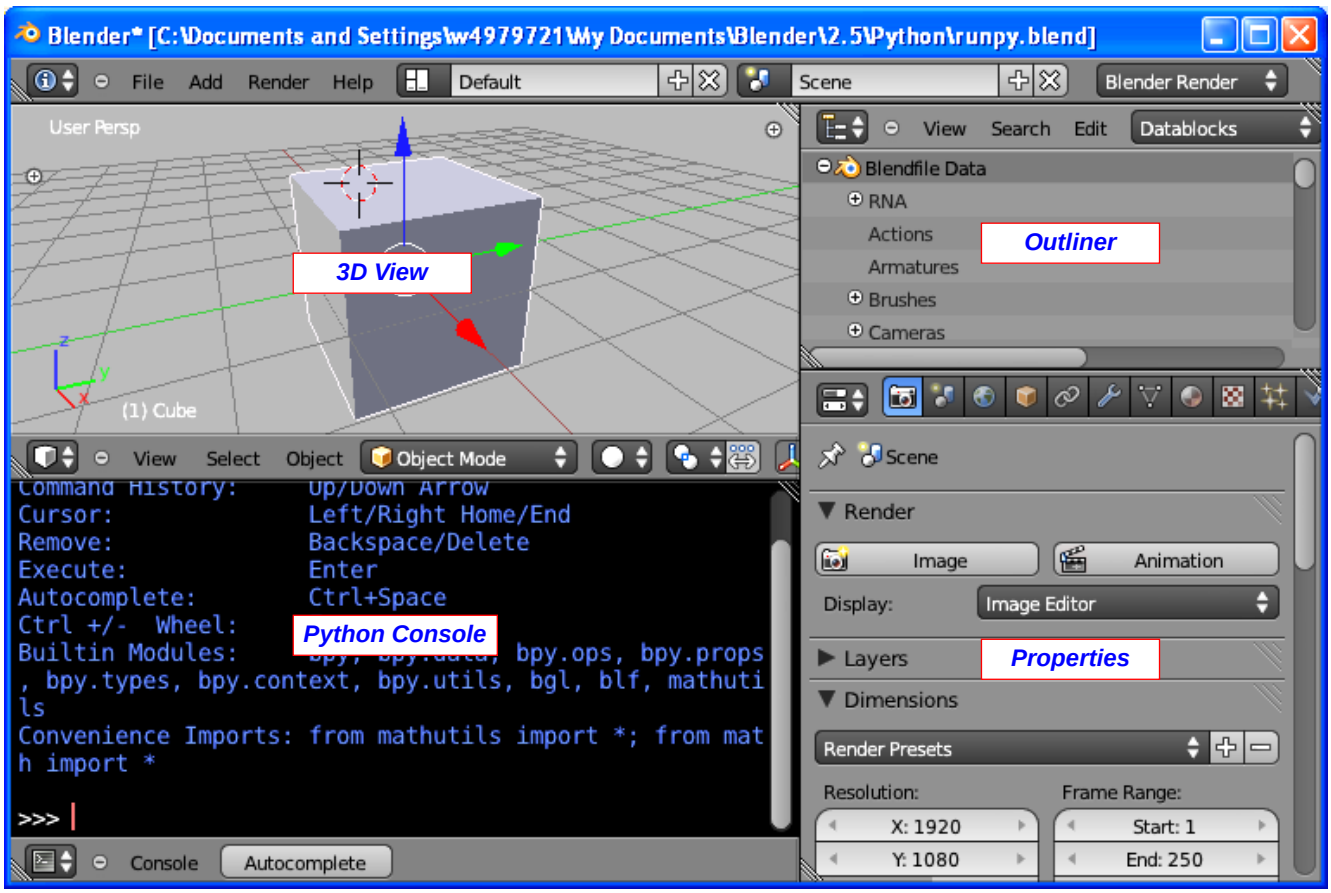


Figure 3.3.1 Screen layout for the “test environment”

Save this file on a disk, and then import it to the PyDev project (using the ***Import..*** command — see details on page 118), to have it at hand (Figure 3.3.2):

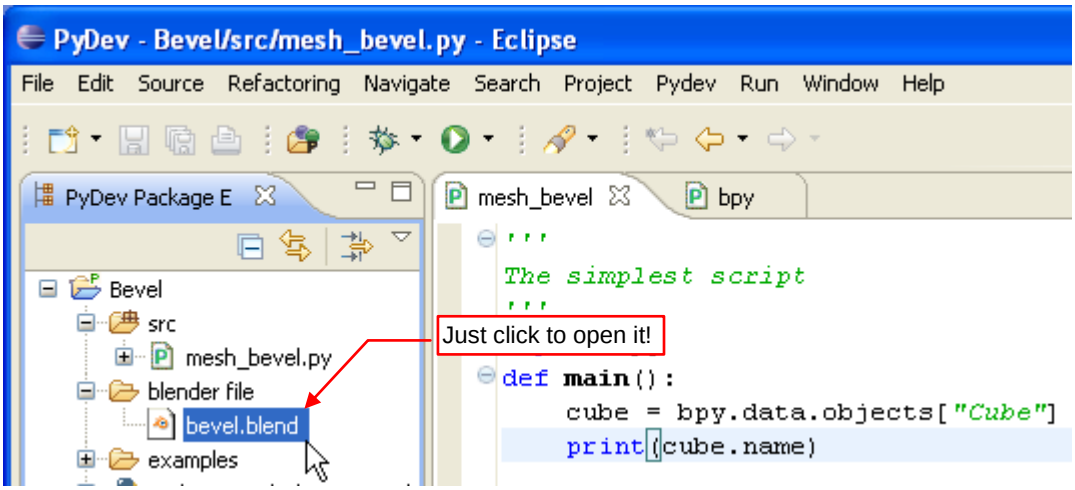


Figure 3.3.2 The test Blender file, added to the Eclipse project

The goal of this section is to create a script code that will tag bevel (see page 36) the selected mesh edges. Finding the way of adding the *Bevel* modifier using Python we will leave for later. For the purpose of the tests in this section, simply add it manually (Figure 3.3.3):

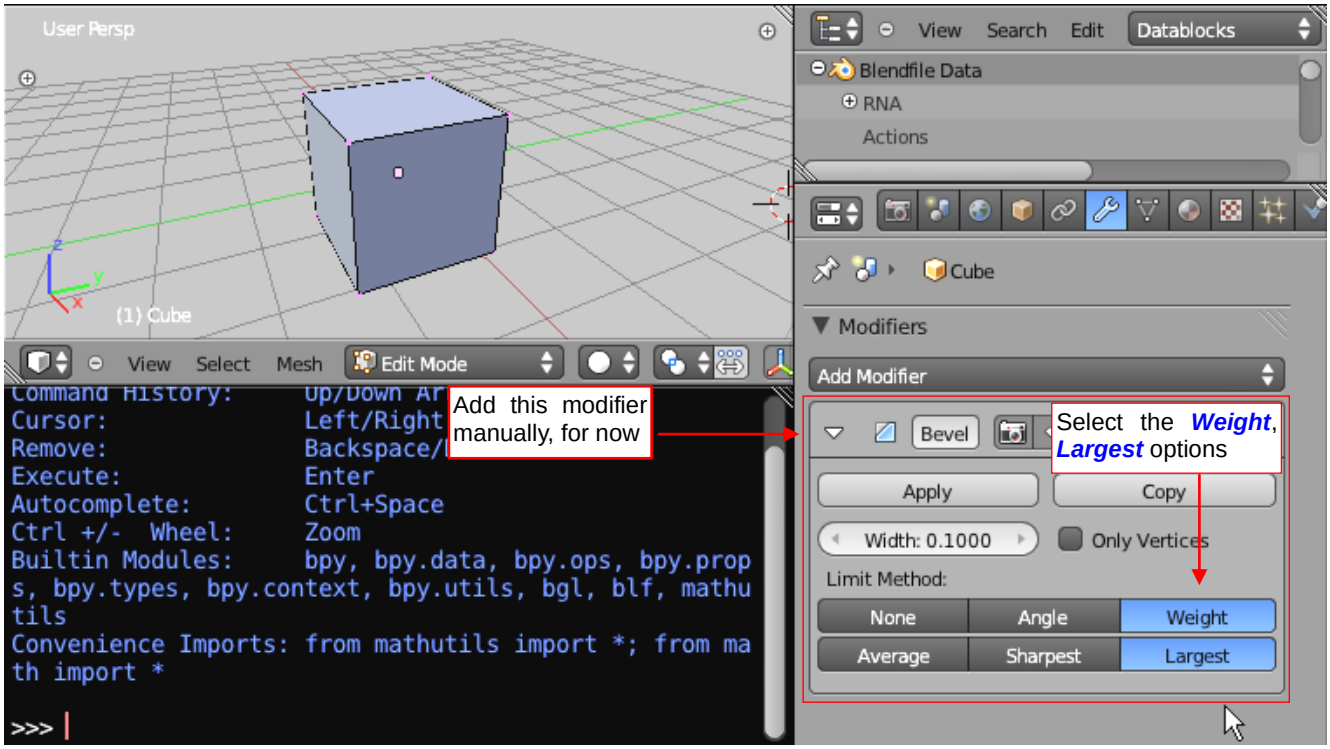


Figure 3.3.3 Preparation of the test object

To look for the mesh objects responsible for the *Bevel* effect, use the *Outliner* editor. In the *Datablocks* mode, it literally shows the entire contents of the file. This is a well-presented structure of ***bpy.data*** (Figure 3.3.4):

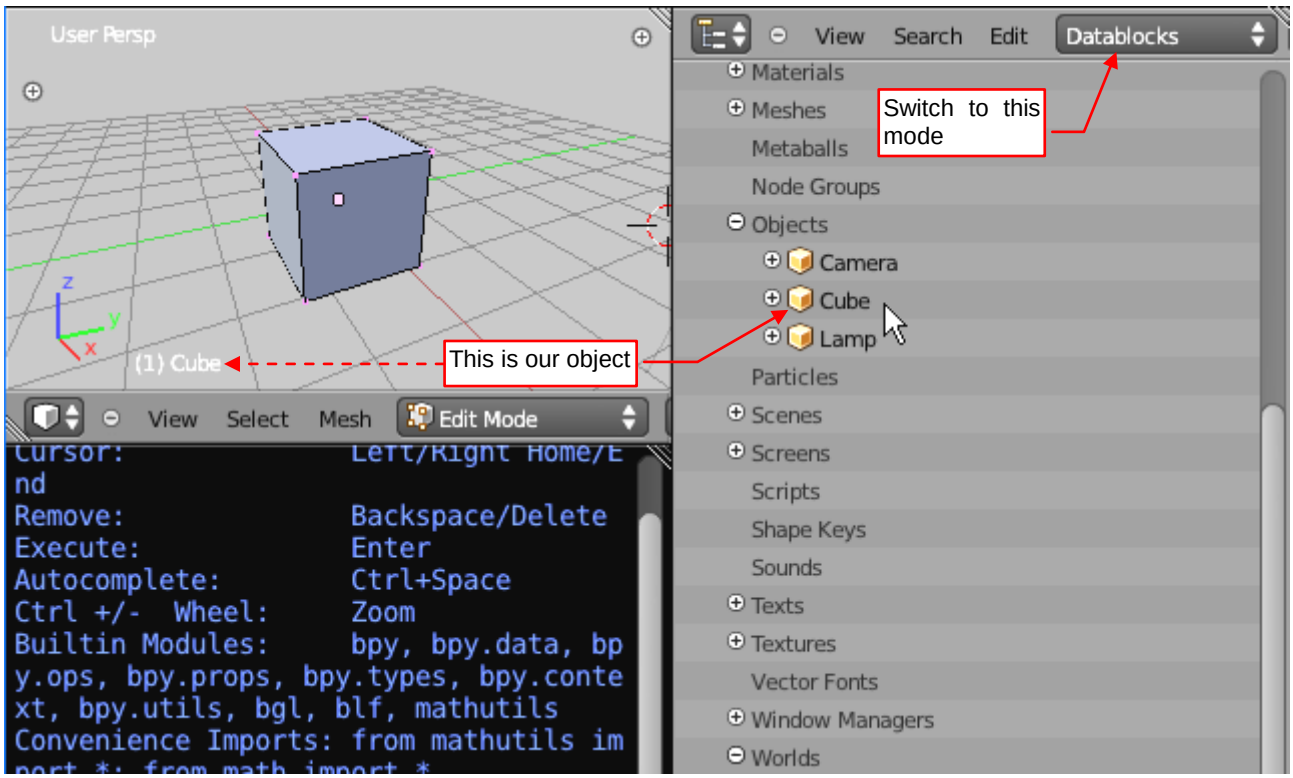


Figure 3.3.4 Finding an object in the *Outliner* window (*Datablocks* mode)

Find there the ***Objects*** collection. Expand it, to see the individual objects that are present in this scene. One of them — **Cube** — is the active one (it implies its name, displayed at the bottom of the *3D View* editor).

Cube is our test object. You can find its mesh in the *Data* property (Figure 3.3.5):

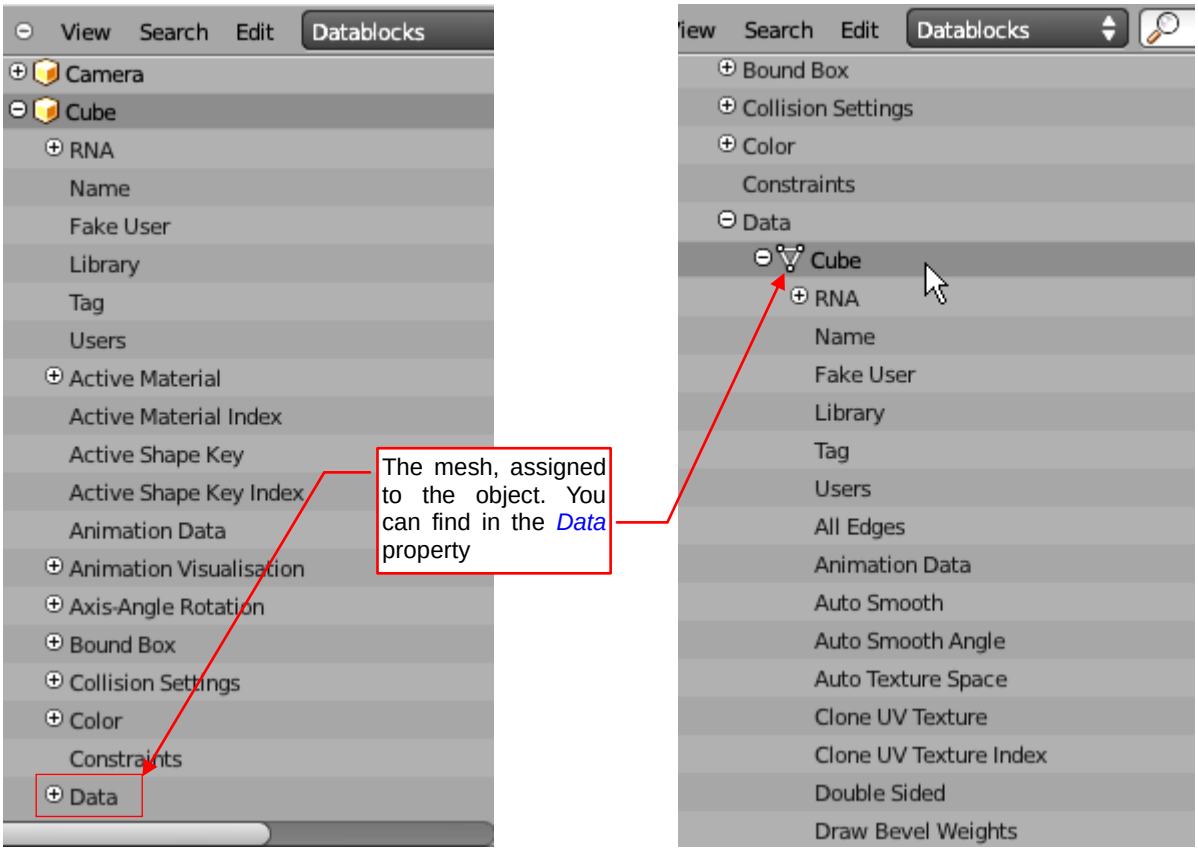


Figure 3.3.5 Internal structure of an object and its the *Data* field

(This mesh is also named **Cube**, but it could have any other name). Examine the mesh properties, to identify the most important collections: *Vertices*, *Edges*, *Faces*. We are interested in the *Edges* (Figure 3.3.6):

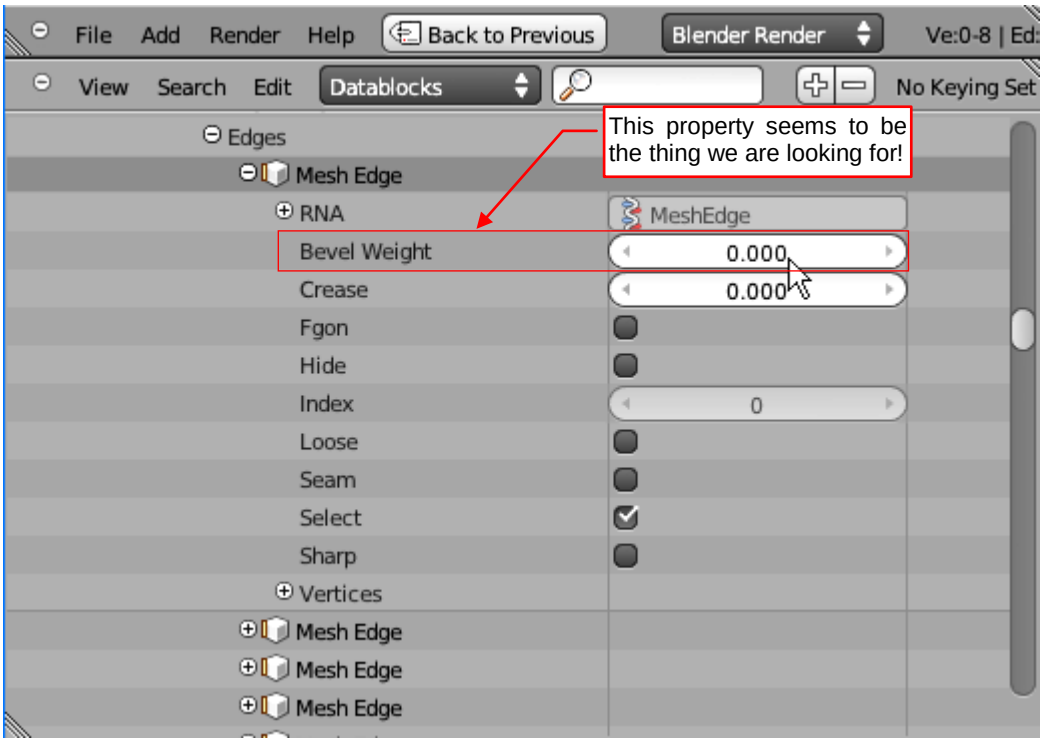


Figure 3.3.6 Properties of a single mesh edge (an element of the *Edges* collection)

Let's expand one of its elements (a ***MeshEdge*** object). What we can see here? Something immediately strikes the eye: the *Bevel Weight* field. Its current value is 0.0, which probably means no chamfer. So, if we change it to 1.0 (its maximum value), it will bevel this edge, right?

Let us verify this assumption (Figure 3.3.7):

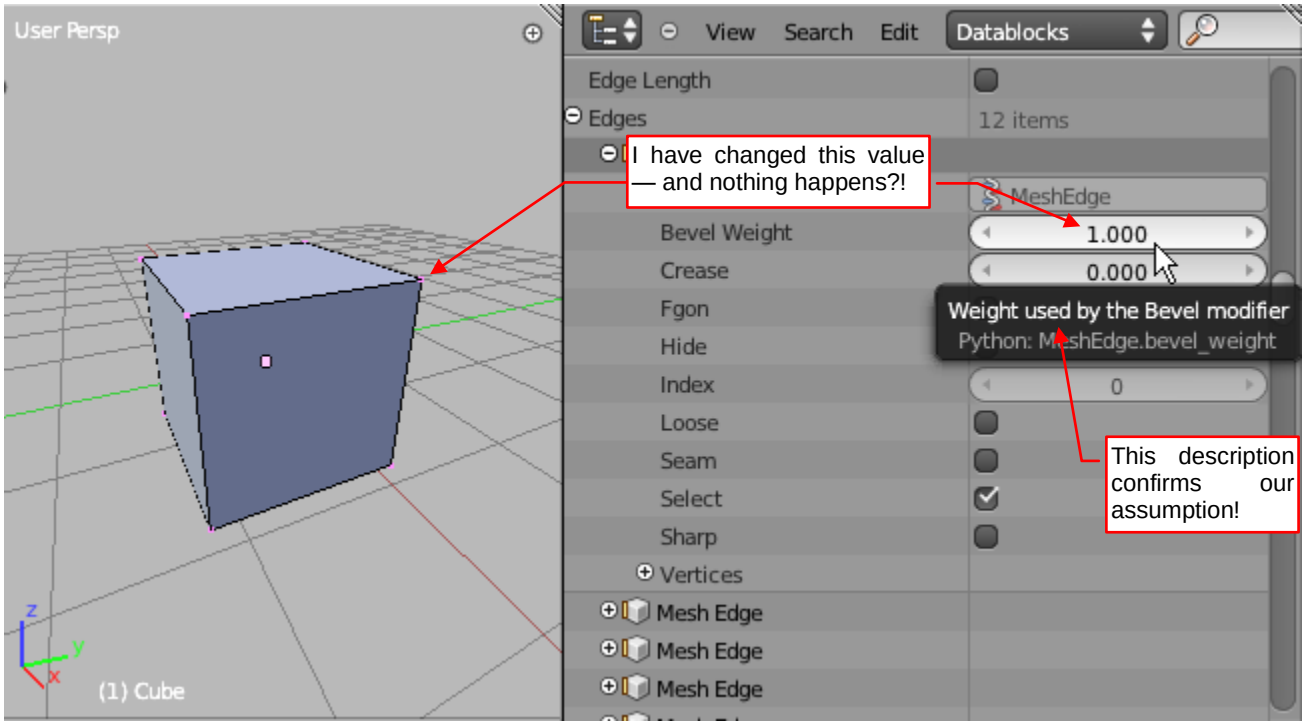


Figure 3.3.7 An attempt to change manually the *Bevel Weight* value

Set the *Bevel Weight* of the first edge to 1.0 — and nothing happens! What is going on!? After all, you can read from the tooltip of this property that it is the weight used by the *Bevel* modifier!

Maybe we are just looking from the wrong side? We are not sure where exactly on this mesh is the edge #0… Let's look at it from the other sides (Figure 3.3.8):

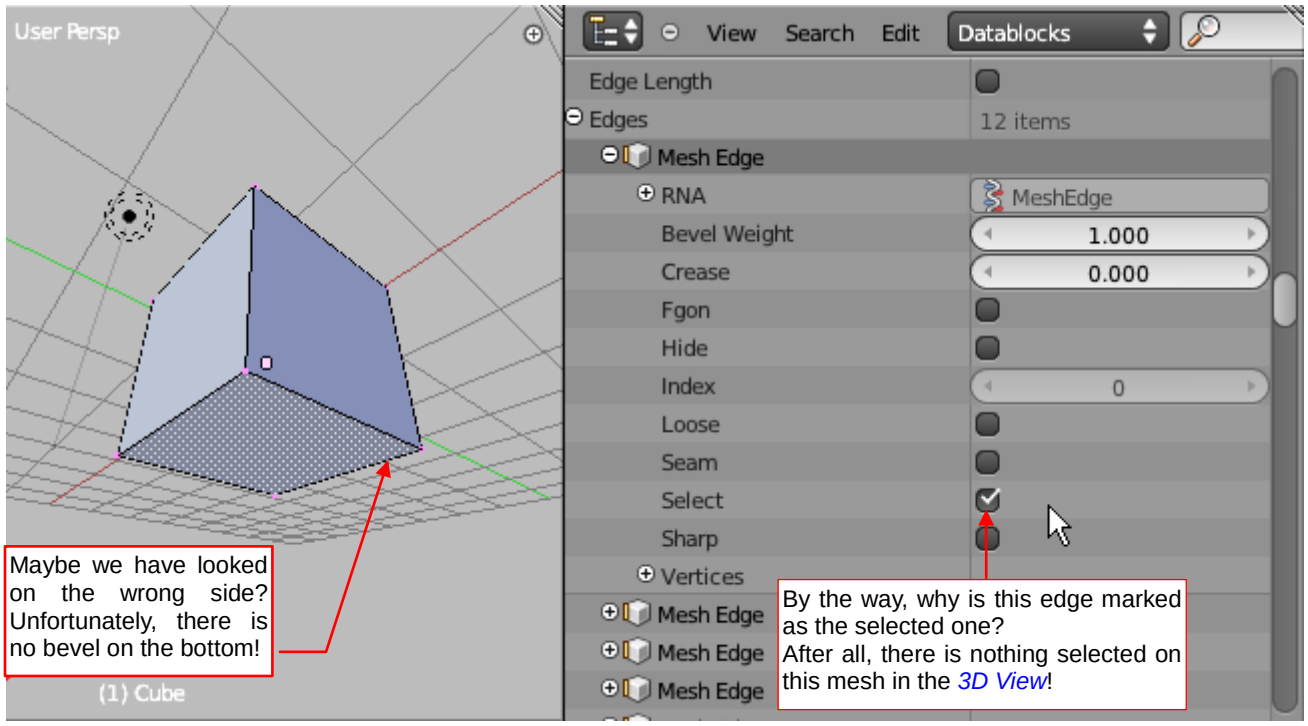


Figure 3.3.8 Closer examinations of the mesh data

There is no trace of the beveled edge, on all the sides. On the other hand, look carefully at the properties of this edge #0. There is something wrong with them. Why the *Select* field is "checked"!? We just have viewed this cube from all sides, and none of its edges is selected! It seems that the *Outliner* shows the wrong data!

Let's try to switch into the *Object Mode* (Figure 3.3.9):

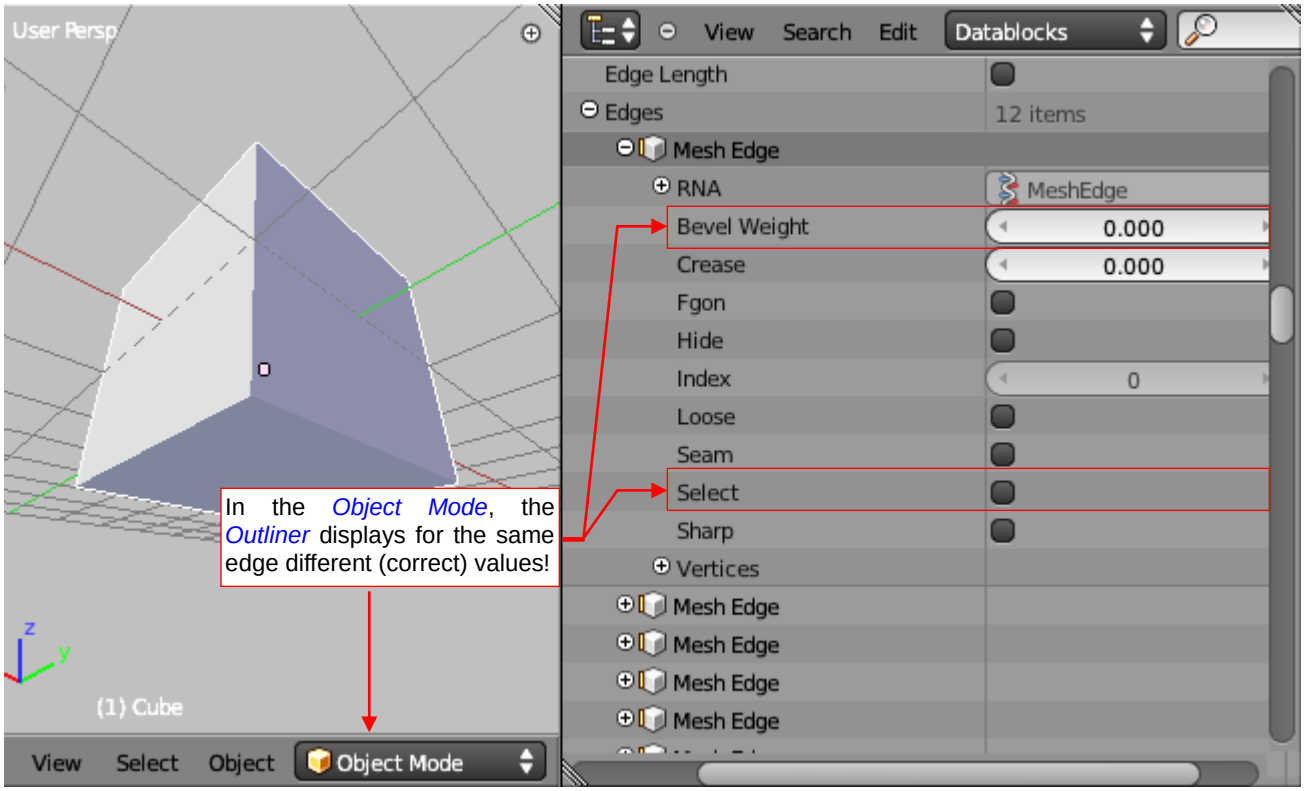


Figure 3.3.9 The same properties, after switching from the *Edit Mode* to the *Object Mode*

It's interesting: values, displayed in the *Outliner*, have been changed. The edge #0 is not marked as selected (its *Select* field is off). In addition, the current *Bevel Weight* value is 0.0. It seems that everything we have changed in the *Edit Mode* has been silently ignored. Or perhaps we should try to do the same in this mode? Maybe it will work in the *Object Mode*, since the *Select* value has become real?

In the *Object Mode*, I have changed the *Bevel Weight* to 1.0 — and it works as we assumed! (Figure 3.3.10):

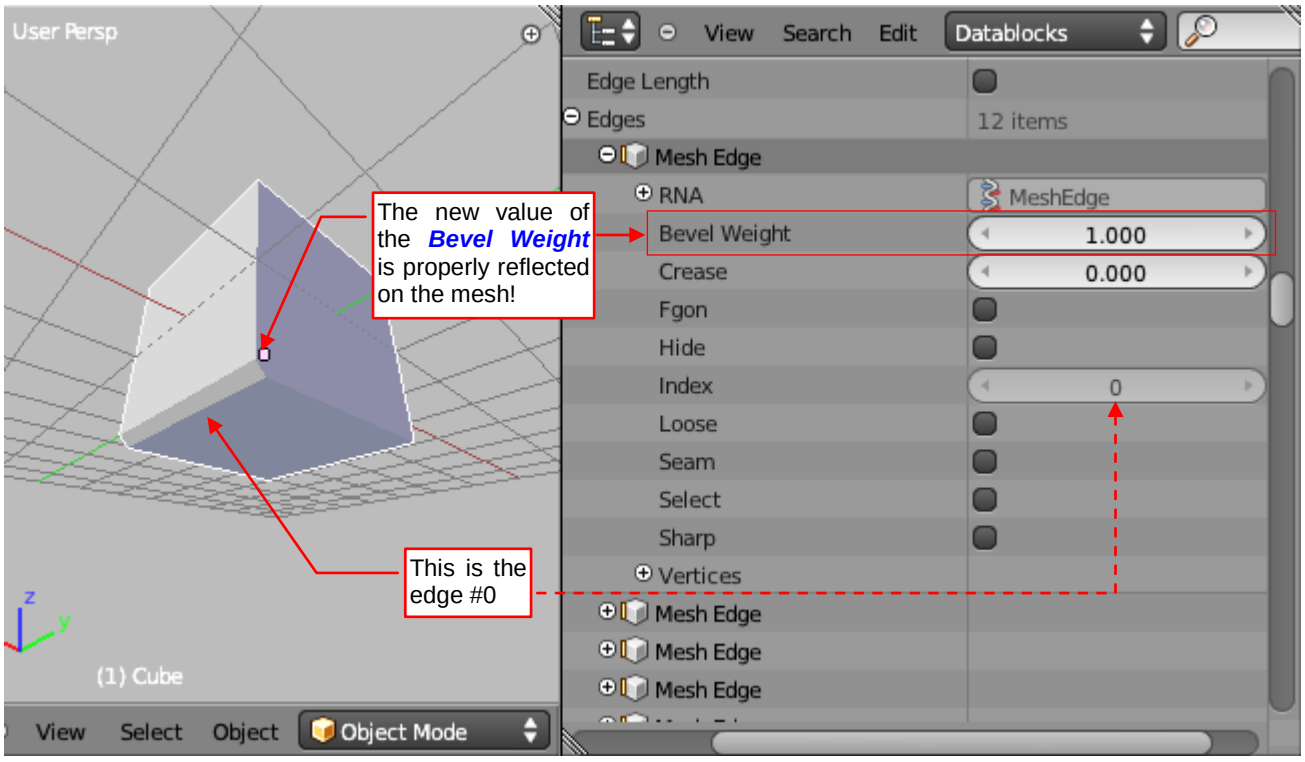


Figure 3.3.10 Changing the *Bevel Weight* in the *Object Mode*

- Current Blender version (2.58) ignores all changes made in the *Edit Mode* with a script or the *Outliner datablock* controls. Everything works properly in the *Object Mode*. They have to fix it in Blender 2.6.

It seems that we have identified the property that should be changed by the script to tag bevel the selected edges. Now we have to find their Python API names (*Outliner* displays their "human readable" labels). It is very simple, because Blender displays the "Python name" of each control in its tooltip (Figure 3.3.11):

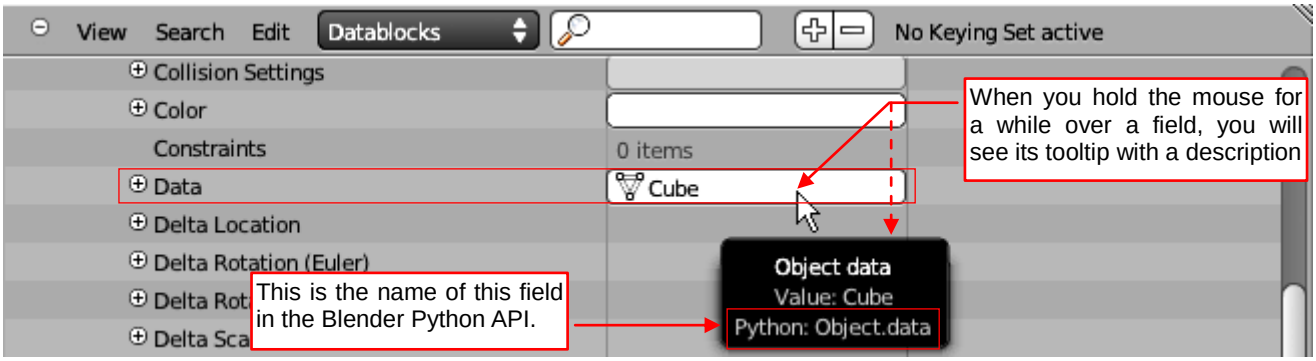


Figure 3.3.11 Identification of the Python API name doe a control form the Blender screen

There is only a problem with the collections, because they do not have any tooltips (Figure 3.3.12):

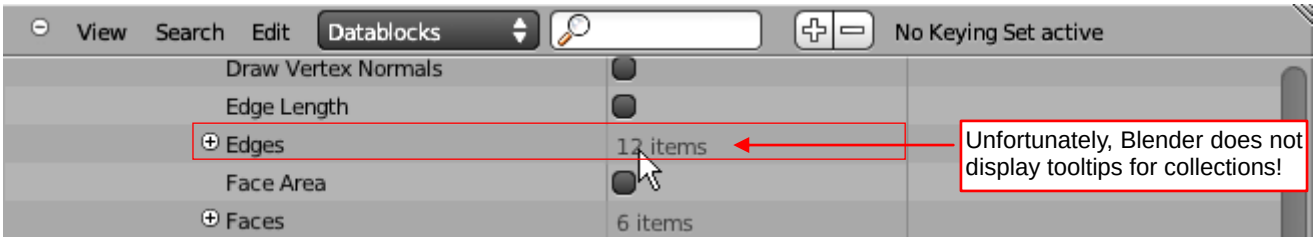


Figure 3.3.12 The problem with the identification of collection names

Usually, collections in Python have the same name, but written in lower case, and each space is replaced with the underscore. However, if you want to make sure, you can verify it in the so-called *RNA* (Figure 3.3.13):

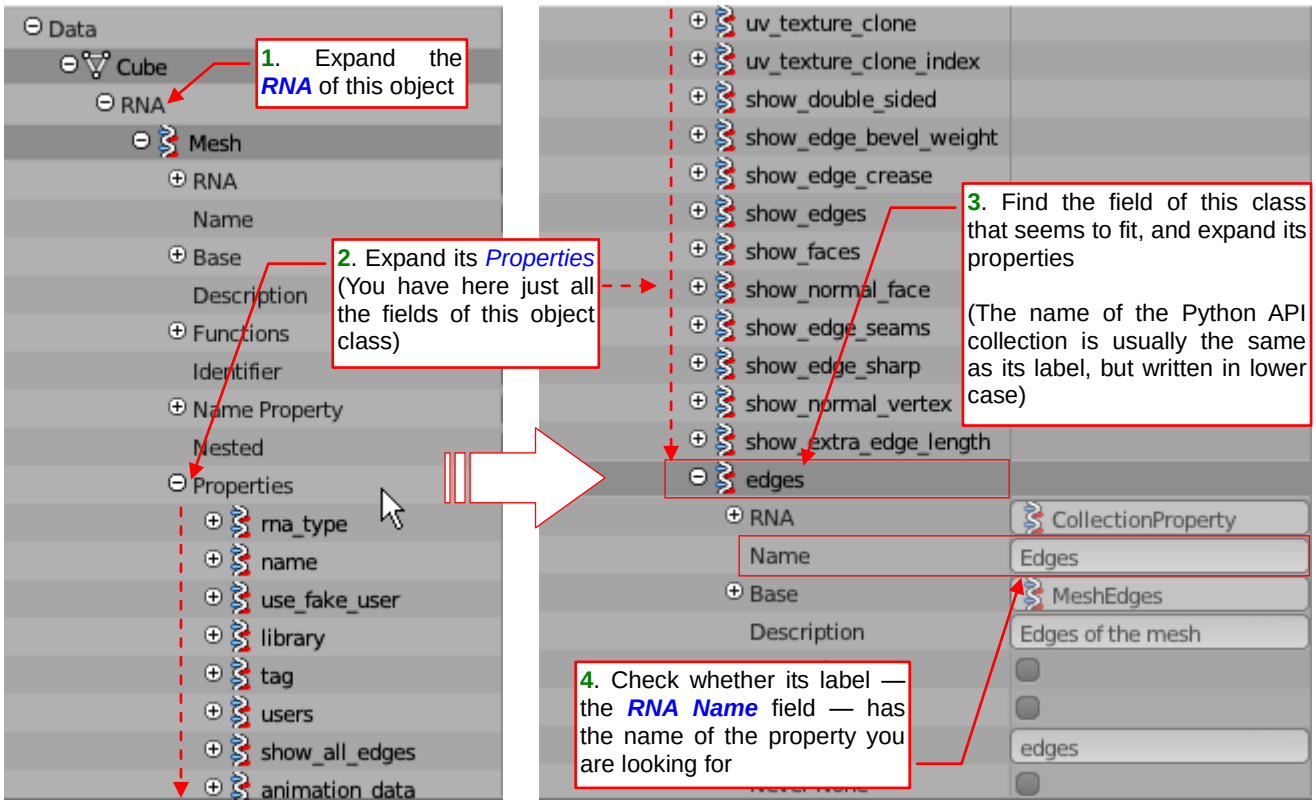


Figure 3.3.13 Verification of the Python API name in the *RNA* structure of the collection parent object

It looks that the "path" of the Python API names to the mesh edges is: ***Object.data.edges***. Let's check it at once in the Blender *Python Console* (Figure 3.3.14):

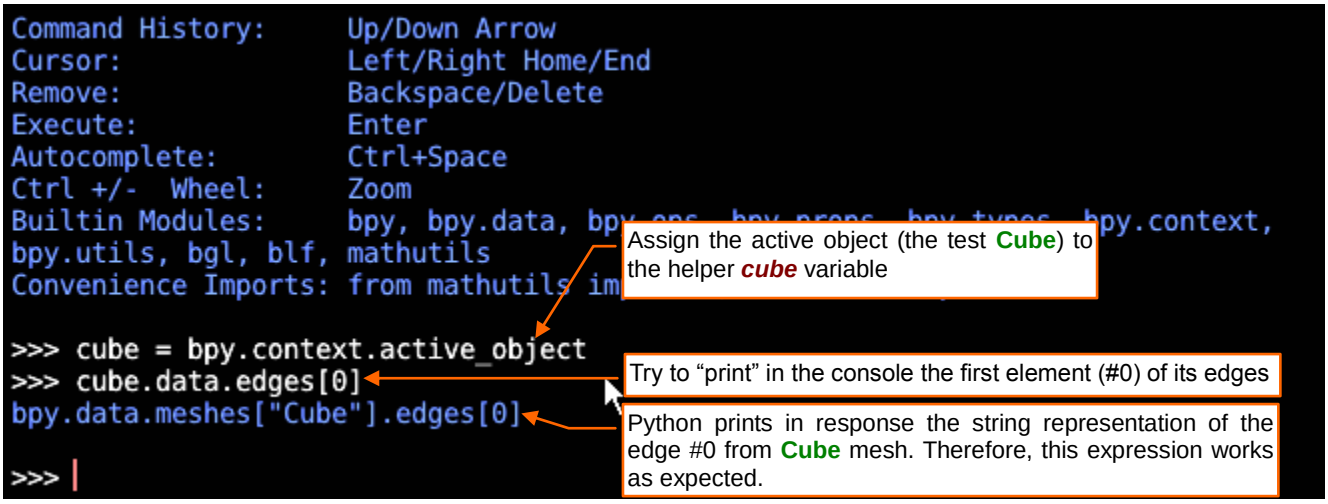


Figure 3.3.14 Verification of the "name path" to a Blender API object

First, get from current context (***bpy.context***) a reference to the ***active_object*** (it is our **Cube**). Store it in helper ***cube*** variable. Then check if the ***cube.data.edges*** collection has the edge #0 (we have changed it in the *Outliner*). It has. So, let's check the *Bevel Weight* of this edge (Figure 3.3.15):

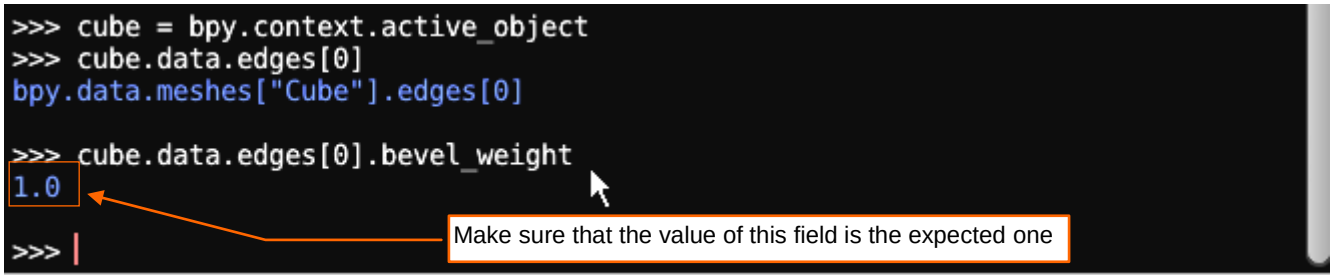


Figure 3.3.15 Verification of API field value

So far, everything works fine — the edge #0 has its ***bevel_weigth*** = 1.0.

There is yet another test to do: use this Python expression to change the bevel weight of another edge.

To not type again the entire "path" to this expression, just press the cursor key (↑) in the console. (The ↑/↓ keys scroll through the list of previously entered statements). Thus, in the command prompt you will see the previously entered expression. Just change the index of the edge collection element from ***[0]*** to ***[1]*** and set its ***bevel_weight*** to 1.0 (Figure 3.3.16).

When you execute this command, the second edge of this cube will also become beveled. So — it works!

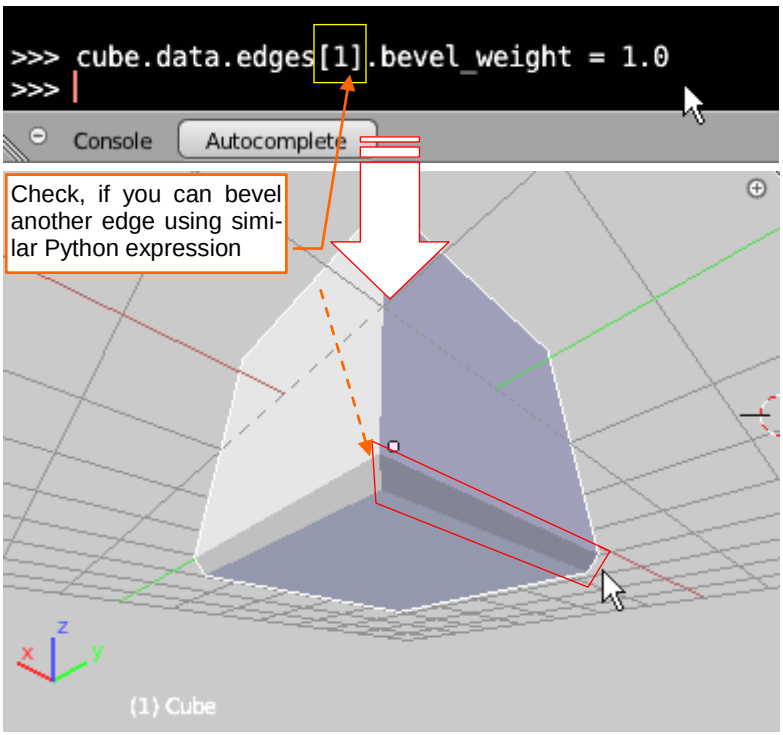


Figure 3.3.16 Verification of the *bevel_weight* influence on the mesh

Since we have found and verified the key Python statements, it's time to start writing our script. At the beginning, add the *import* statement of ***bpy*** (Blender API module). Then add the main procedure header (Figure 3.3.17):

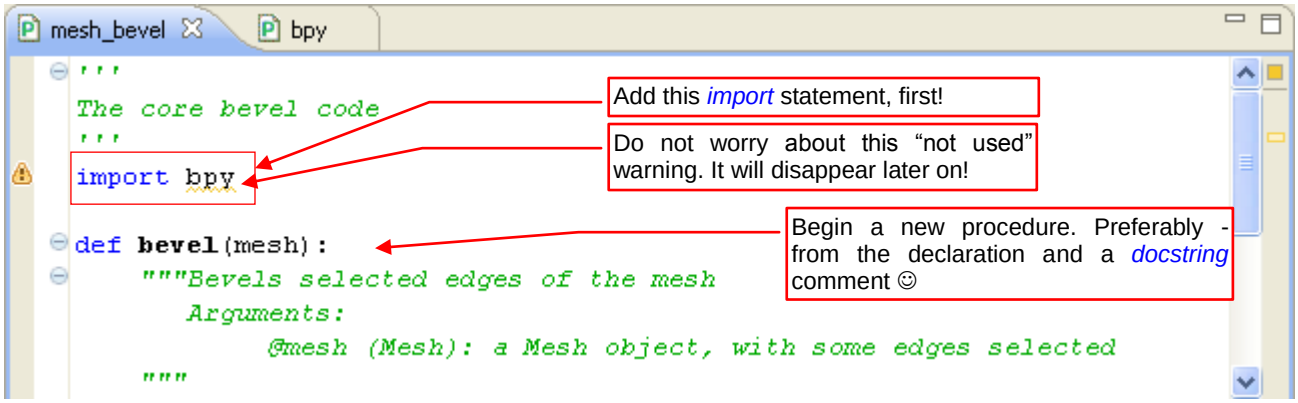


Figure 3.3.17 Beginning of the script: the *import* declaration and the main (*bevel()*) function header

In order to have proper code autocompletion, add the „type declaration" for two variables (Figure 3.3.18):

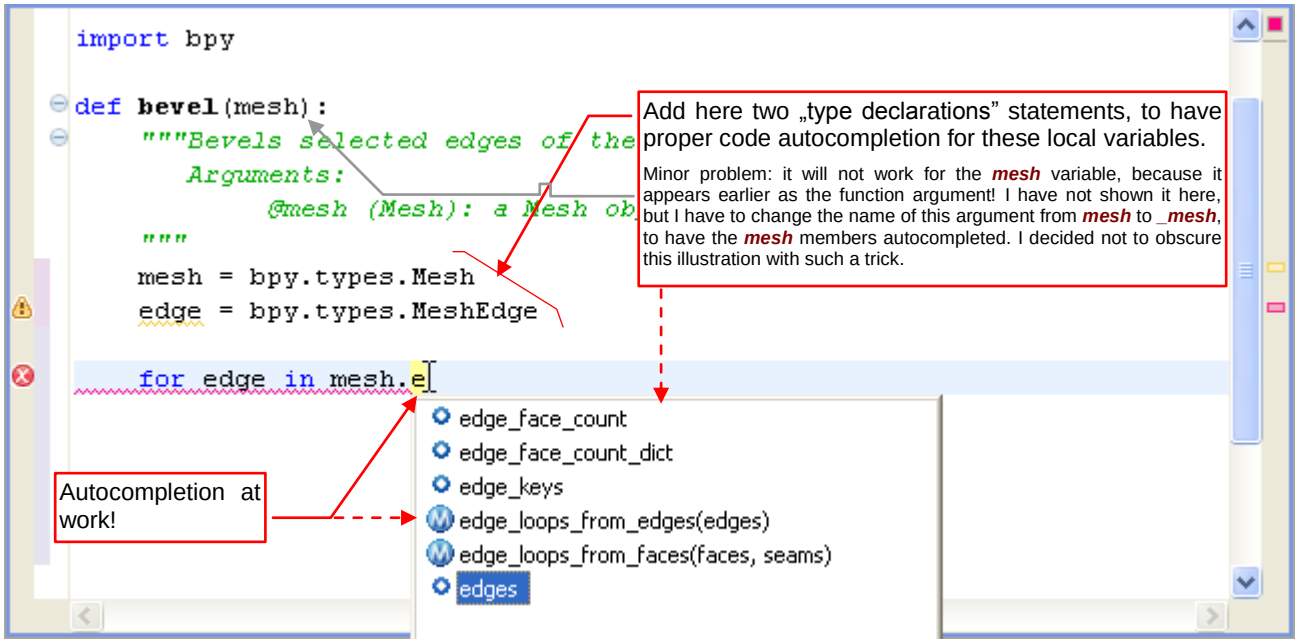


Figure 3.3.18 Forcing proper autocompletion for local variables with the "type declaration" statements

Add to the ***bevel()*** procedure a loop: for each selected edge (***edge.select*** == *True*) tag bevel, setting its ***edge.bevel_weight*** to 1.0 (Figure 3.3.19):

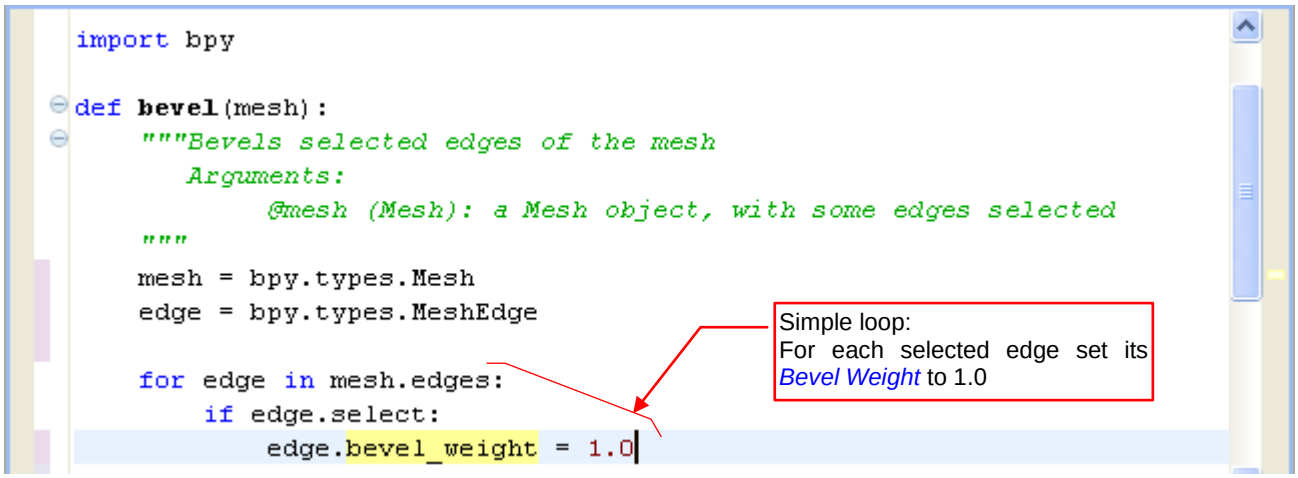


Figure 3.3.19 The main loop of the *bevel()* function

That is all! After finishing this procedure, do not forget to comment out the „type declaration" statements of its local variables (see Figure 3.3.18). (Just comment out, do not to remove, because they still can be useful). At the end of the module invoke the ***bevel()*** procedure for the active object (more precisely, for its mesh — ***active_object.data***: Figure 3.3.20). At this early stage of writing, we will not bother checking if the active object has a mesh at all.

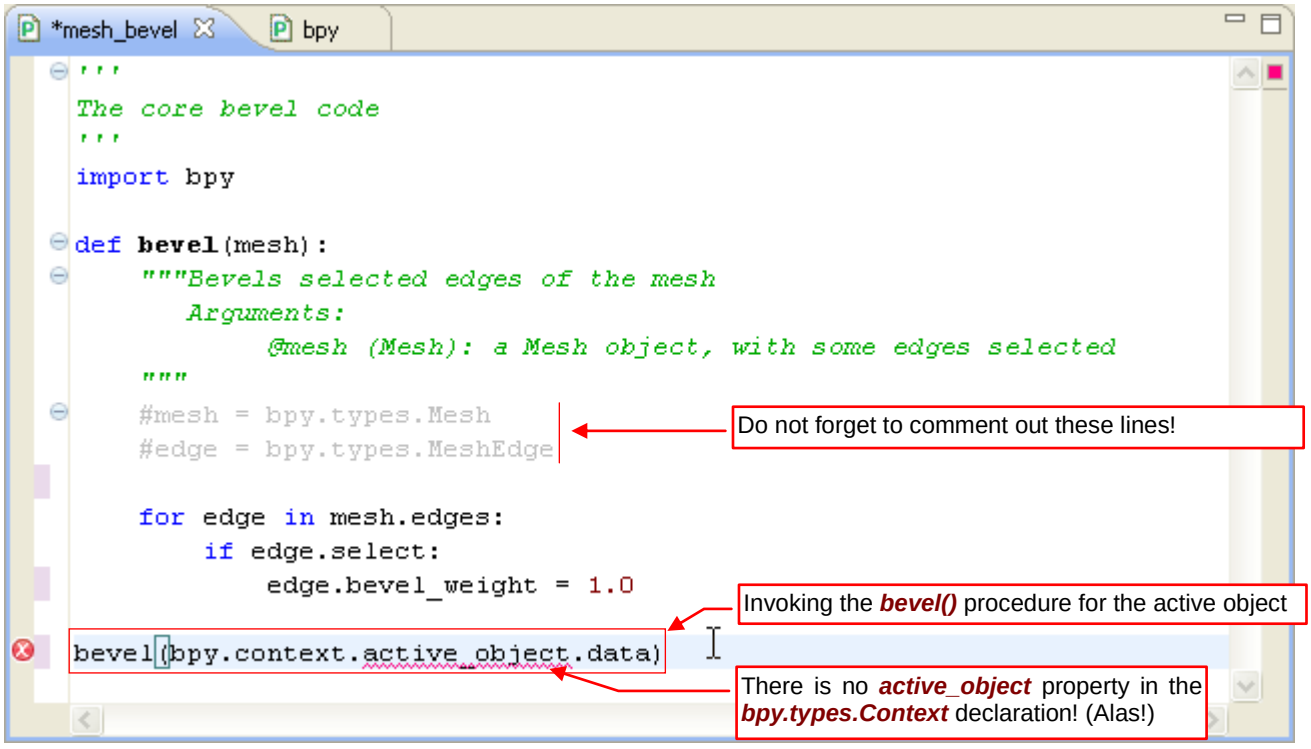


Figure 3.3.20 Invoking the main procedure (*bevel()*) and commenting out the „type declarations"

Notice that the ***active_object*** property of the ***bpy.context*** object has the red underline (in PyDev it means a possible error). It is strange, because we have already checked in the *Python Console* that such expression is valid. Well, it is a problem with another "hard coded" Blender structure: ***bpy.Context***. Depending on the circumstances of the call, this object can provide different properties! I am not able to recreate such behavior using the declaration from the predefinition file. Here you can find the full list of its variants and their fields. I had to remove the code that generates the declaration of this Blender API fragment from the *pypredef_gen.py* script, because it did not work properly[1]. Ultimately, the declaration of the ***bpy.types.Context*** class in the *bpy.pypredef* file contains only the fields that are common for all variants of this context structure. Unfortunately, the ***active_object*** field is not among them (there are contexts in which this property is not available). Of course, you can just edit the *bpy.pypredef* file, adding to the ***types.Context*** declaration fields that you are missing.

On the other hand — this is just a dynamically interpreted script, not a source code to compile. Despite this error, we will run it without any problem. When we convert this code to an add-on, we will obtain the active object in a different way — from the context reference, passed as the argument to our methods. Then PyDev will not report an error in this case (because it will not recognize the type of this object).

[1] The original script, written by Campbell Barton, used a kind of „hacker trick" here. It refers to the actual executable (Blender) as to a dynamically linked object (a **.dll* in Windows, or *shared object* — **.so* — in Unix/Linux) without the name. Then it reads directly from this code object the definition of the context structures. Unfortunately, it seems that this method is not working properly in Windows, because my adaptation attempts failed.

Summary

- We have prepared in Blender a test environment for our script — *bevel.blend* file. Its screen layout contains useful tools for the code verification: the *Outliner* editor and the Blender *Python Console* (page 48);
- It is convenient to place the test Blender file in the Eclipse project (page 48);
- To inspect Blender data, use the *Outliner* in *Datablocks* mode (page 49);
- To change the Blender data in the *Outliner*, do all the modifications in the *Object Mode* (page 51 - 52). In current Blender version (2.5) the *Outliner* displays just a `copy` of the mesh data, in the *Edit Mode*. This copy is created when you enter the *Edit Mode* (when the *Outliner* was already opened) or when you open the *Outliner*. This copy is not updated, and its state may be inconsistent with the current state of the mesh. All your changes made in the *Outliner* in the *Edit Mode*, are ignored. Blender developers ensure that this is temporary situation. They will rewrite this fragment when the new implementation of the meshes (the **BMesh** project) appears in the Blender 2.6 trunk;
- You can read the Python API name of a control that is displayed on any Blender screen from its tooltip (page 53);
- To verify the Python API name of a collection, displayed in the *Outliner*, you have to look into the *RNA* of its parent (page 53);
- Always check in the *Python Console* whether your Python expression works as you expect (page 54);
- The variants of the ***bpy.context*** object have more fields, than listed in its class declaration (***bpy.types.Context***). This set depends on the Blender editor, in which the script was called (page 56);

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 in the “old good way”: loading this file into the Blender *Text Editor* and invoking the *Run Script* command. Only that would be difficult to debug the 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 disk).

I suggest another, more convenient solution. Open in the Blender *Text Editor* the *Run.py* file, which accompanies this book (see page 39). I propose to place the *Text Editor* above the *Python Console* (Figure 3.4.1):

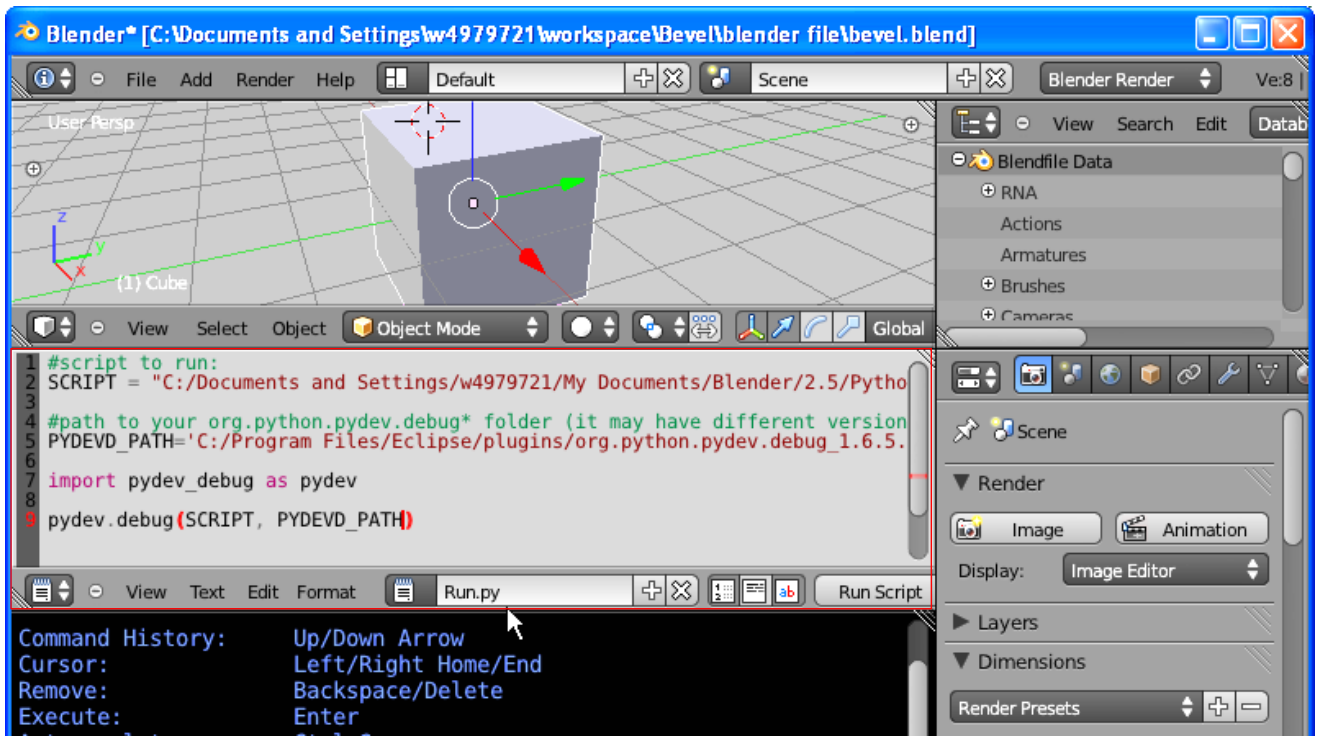


Figure 3.4.1 Adding the *Run.py* script to our Blender test file

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

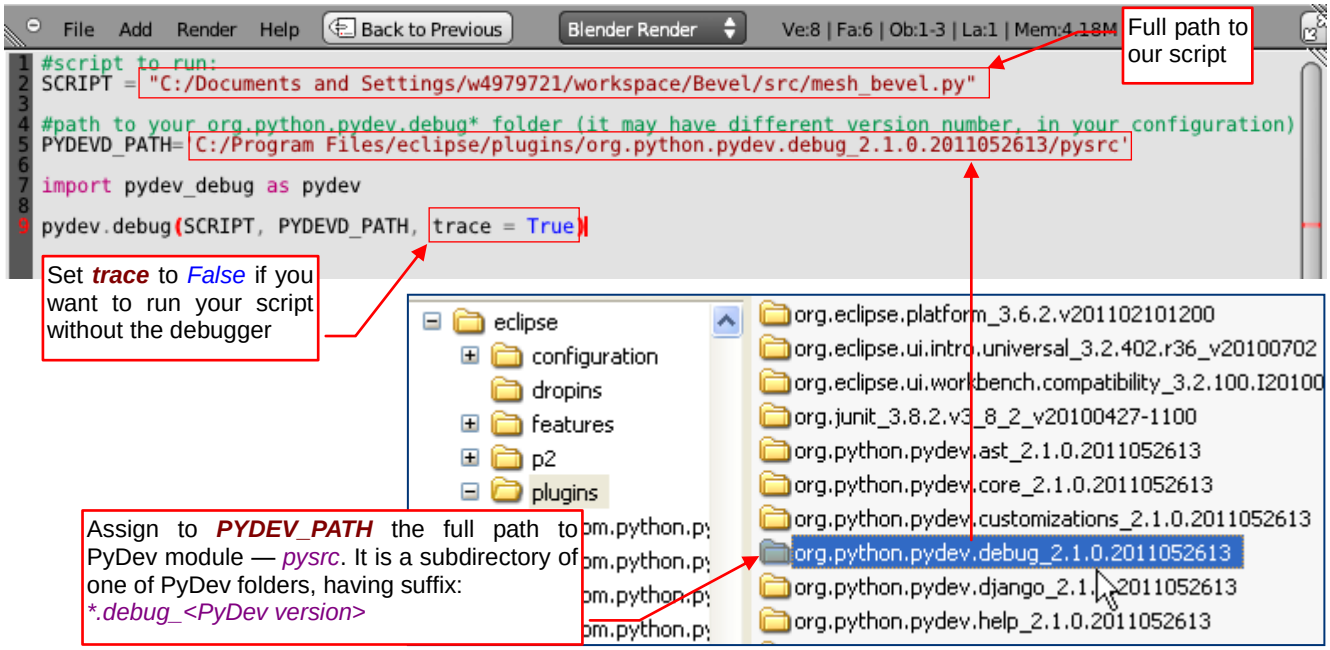


Figure 3.4.2 Adaptation of the *Run.py* code to this project

The ***SCRIPT*** constant should contain the full path to our script file, and the ***PYDEV_PATH*** — full path to the PyDev directory that contains the *pysrc* subfolder. (This is a Python package with so called PyDev remote debugger client — see pages 124 and 129 for more information).

Prepare a Blender model for this test. The code that we wrote has to bevel the selected edges of the mesh. So far we have assumed that the object already has the *Bevel* modifier — so set it as in Figure 3.3.3 (page 49). Mark on this test cube some edges, and then switch to the *Object Mode* (Figure 3.4.3):

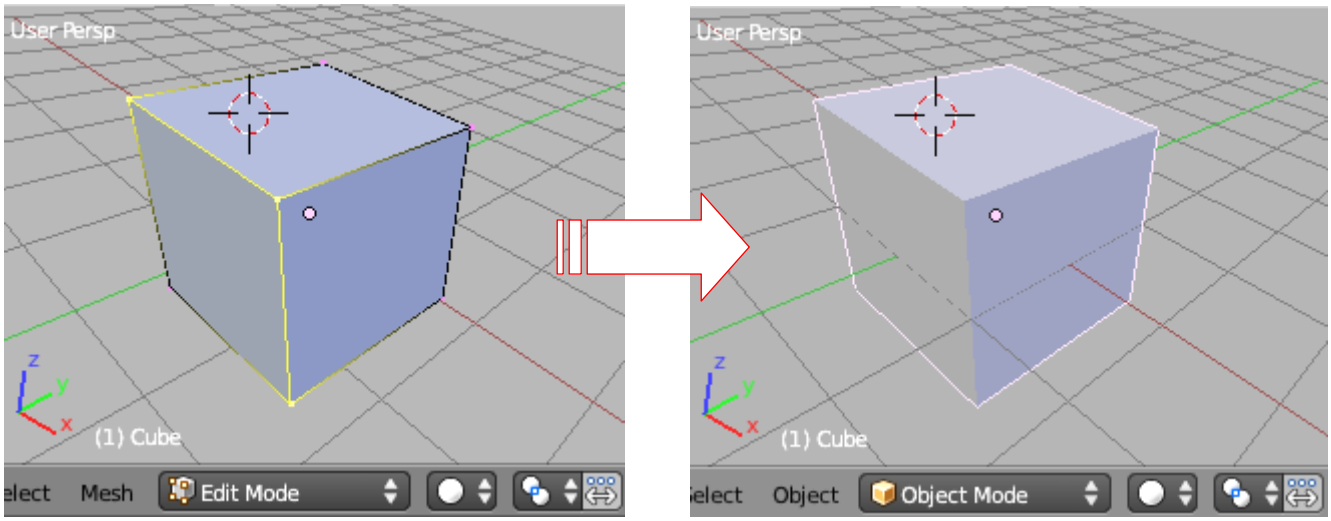


Figure 3.4.3 Preparation of the test object — selecting the edges to bevel

Insert a breakpoint in the script where you want to start debugging. In our case, we will add it to the beginning of the code (Figure 3.4.4):

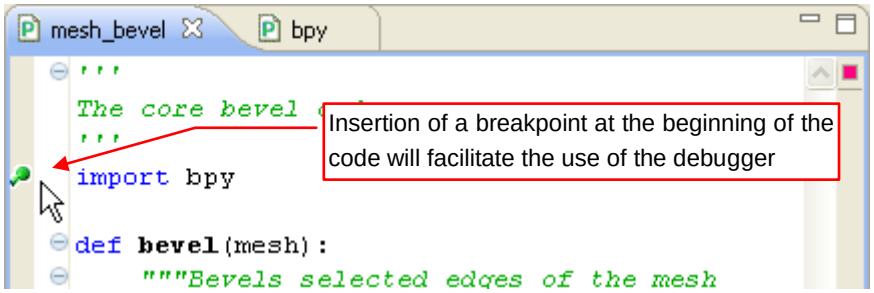


Figure 3.4.4 Placing the breakpoint

Launch from Eclipse the process of the remote debugger server (more about that — page 124) (Figure 3.4.5):

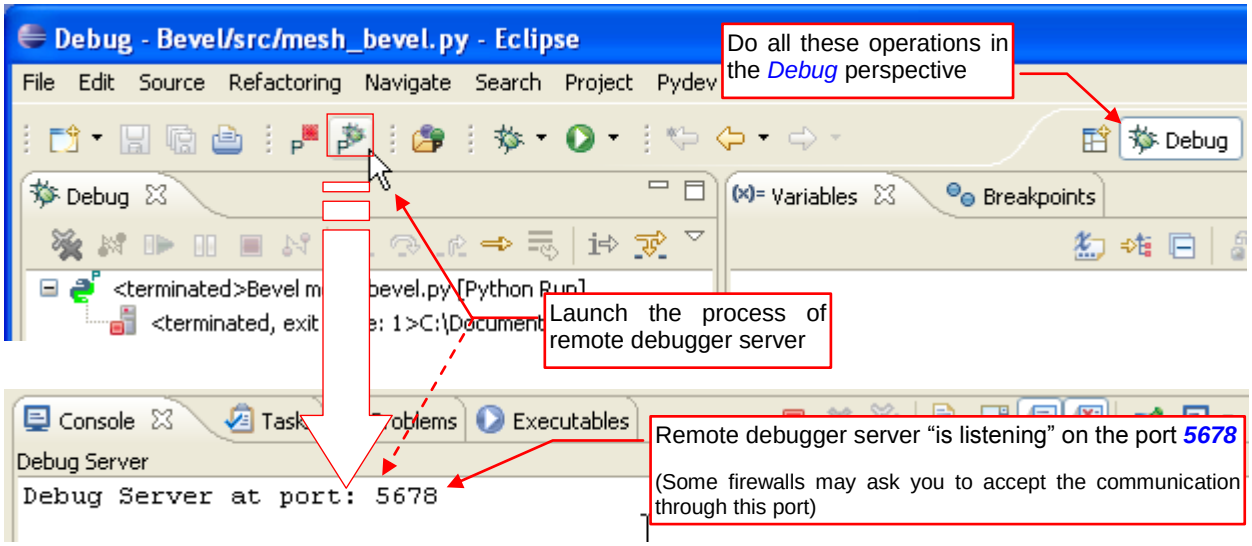


Figure 3.4.5 Launching the debug server process

When the debug server displays its message in the console, we will run our script (Figure 3.4.6):

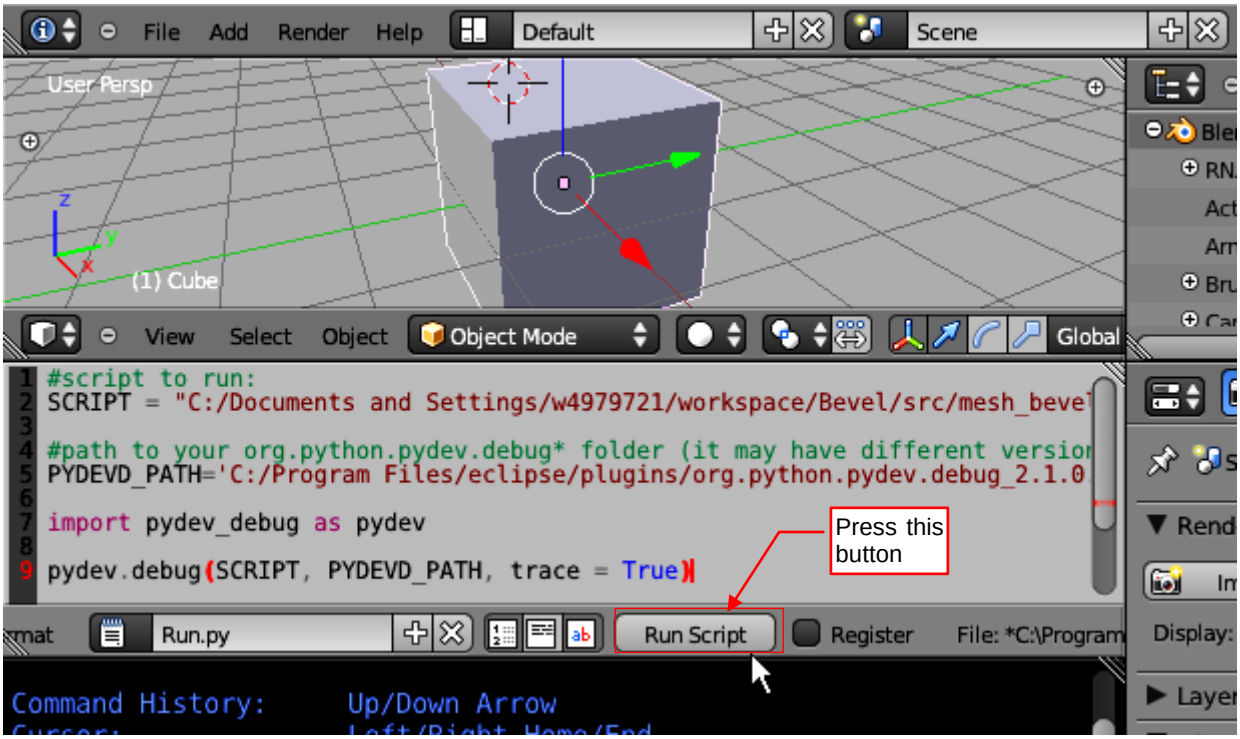


Figure 3.4.6 Launching the Blender script for the debugger

After a few seconds, the Eclipse debugger window "comes to life". In the editor window, PyDev will open the helper *pydev_debug.py* module, and the code execution will stop on one of its lines (Figure 3.4.7):

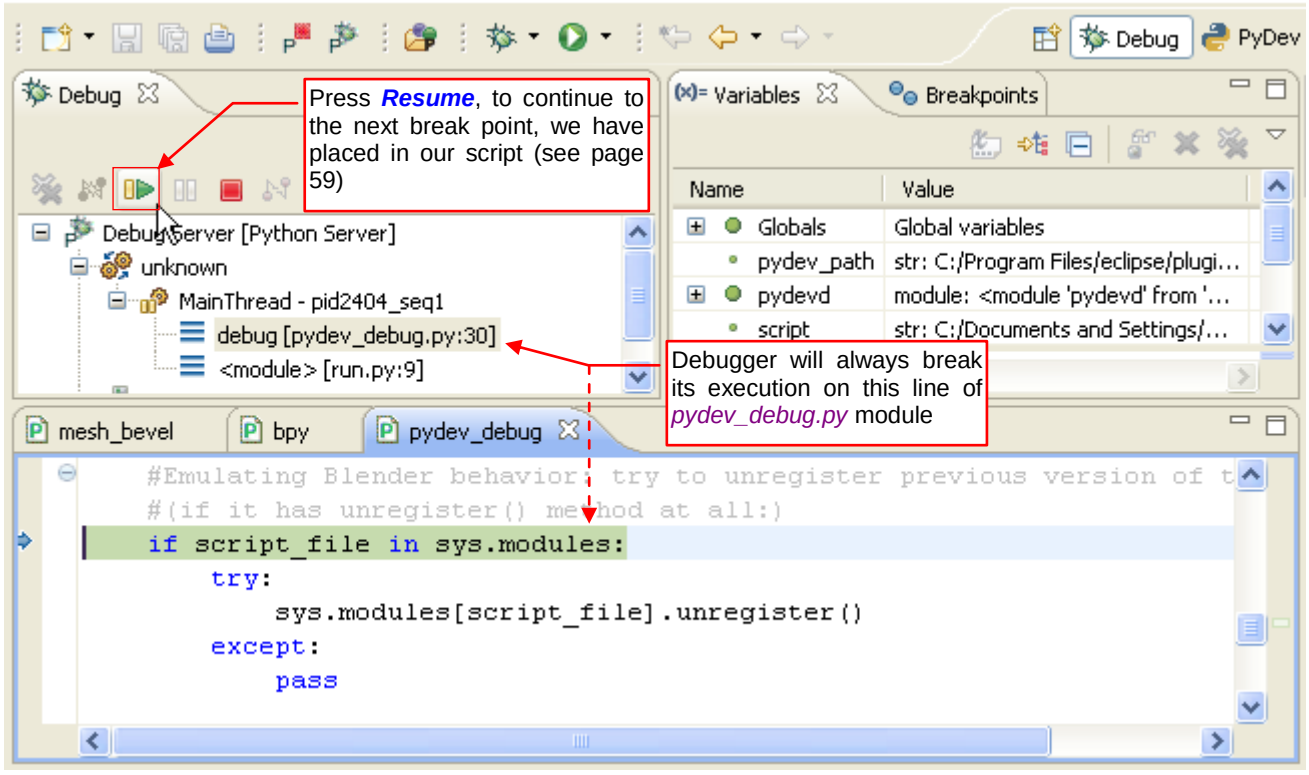


Figure 3.4.7 The first break of the execution — in the helper *pydev_debug* module

The *pydev_debug.py* is a small module, which I wrote to facilitate the tracking Python scripts in Blender. Notice, that it is used in the *Run.py* template (see the code, shown on Figure 3.4.6). You can find the detailed description of its ***debug()*** function on page 129. In any case, the debugger will always stop at this point. Just use the ***Resume*** (F8) command here to continue.

When you press the *Resume* button, the script will be executed up to the line with the first breakpoint. (If there were no such points in the code, it would run to the end). In our case, it will stop at the beginning of the file where we have placed our breakpoint (see page 59 and Figure 3.4.10):

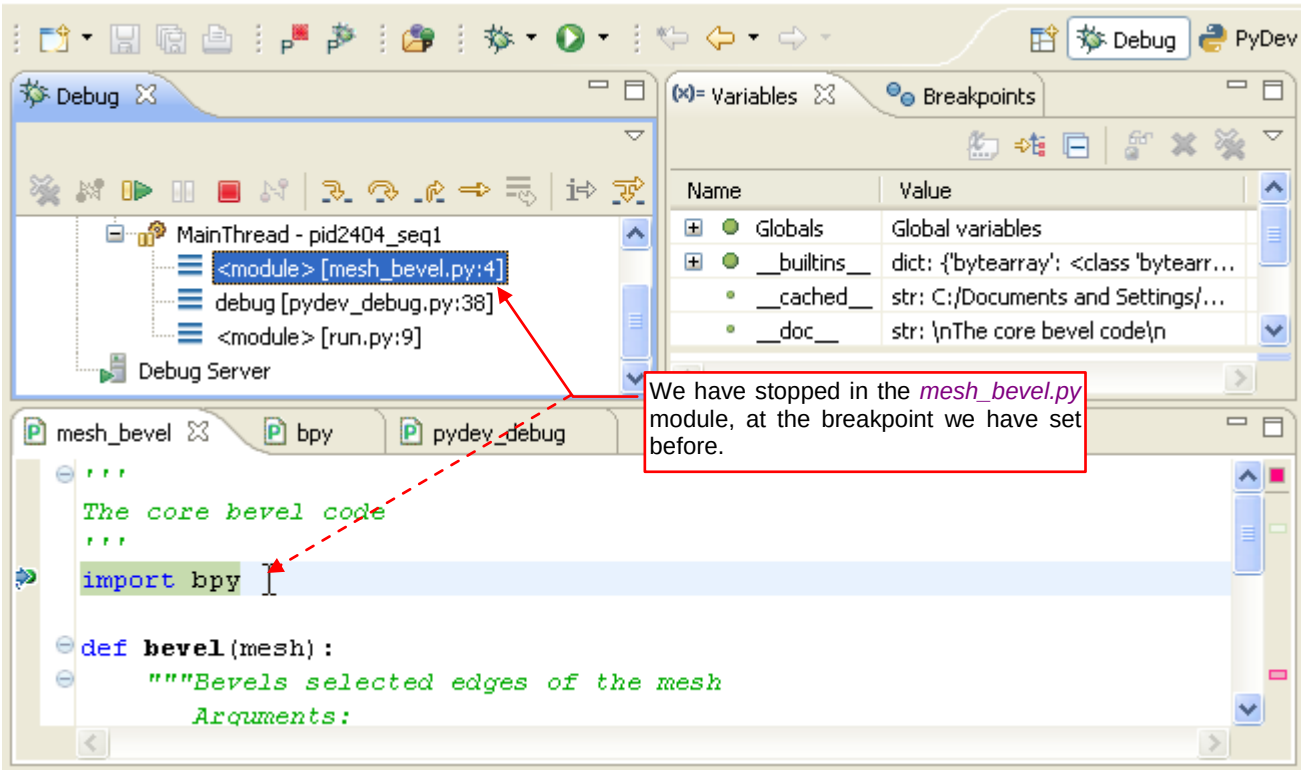


Figure 3.4.8 The first breakpoint, encountered after the *Resume* command

Step Over (F6) the lines of the script main code until you reach the ***bevel()*** function call (Figure 3.4.9):

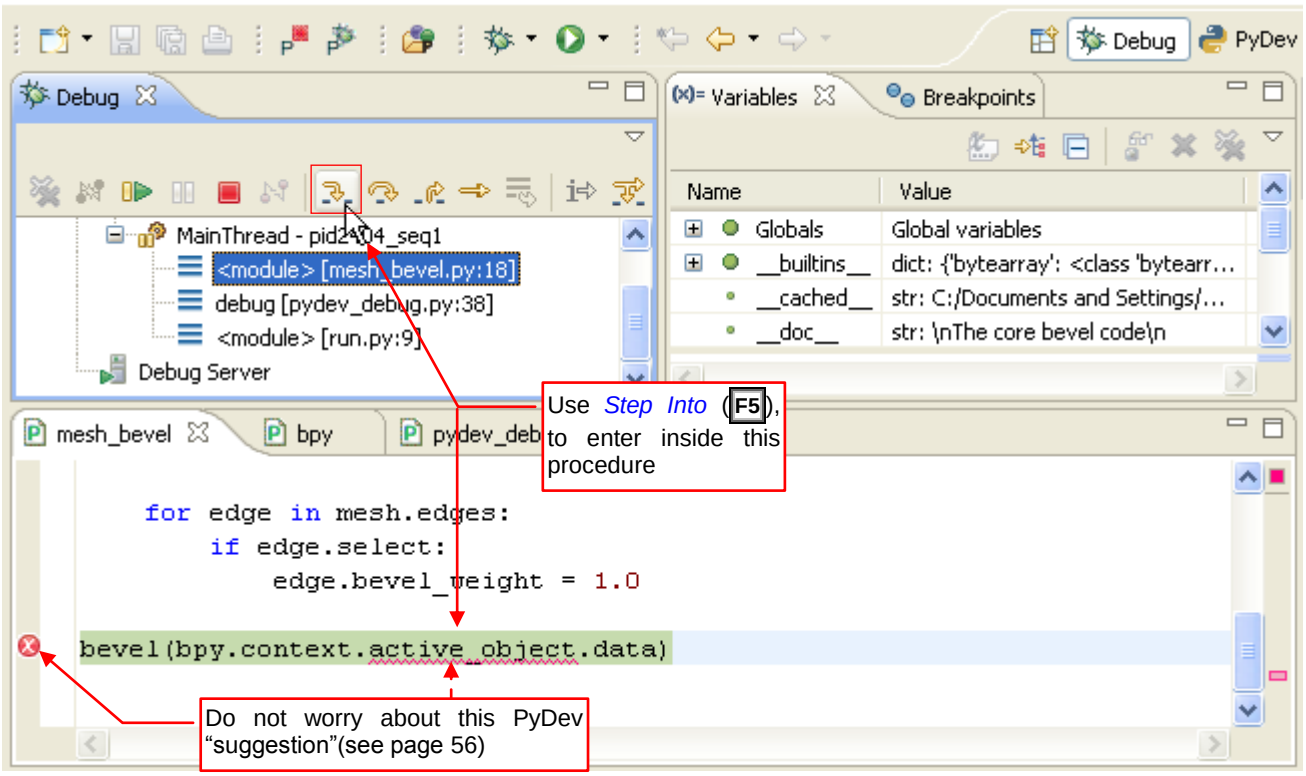


Figure 3.4.9 The next step — „enter" into the *bevel()* procedure

When the procedure call is highlighted, press the F5 (*Step Into*) to track details of its execution.

Follow the loop iterations. Check that the code works as expected, i.e. it changes the *bevel_weight* field to 1.0 only for the selected (***edge.select*** = *True*) edges (Figure 3.4.8):

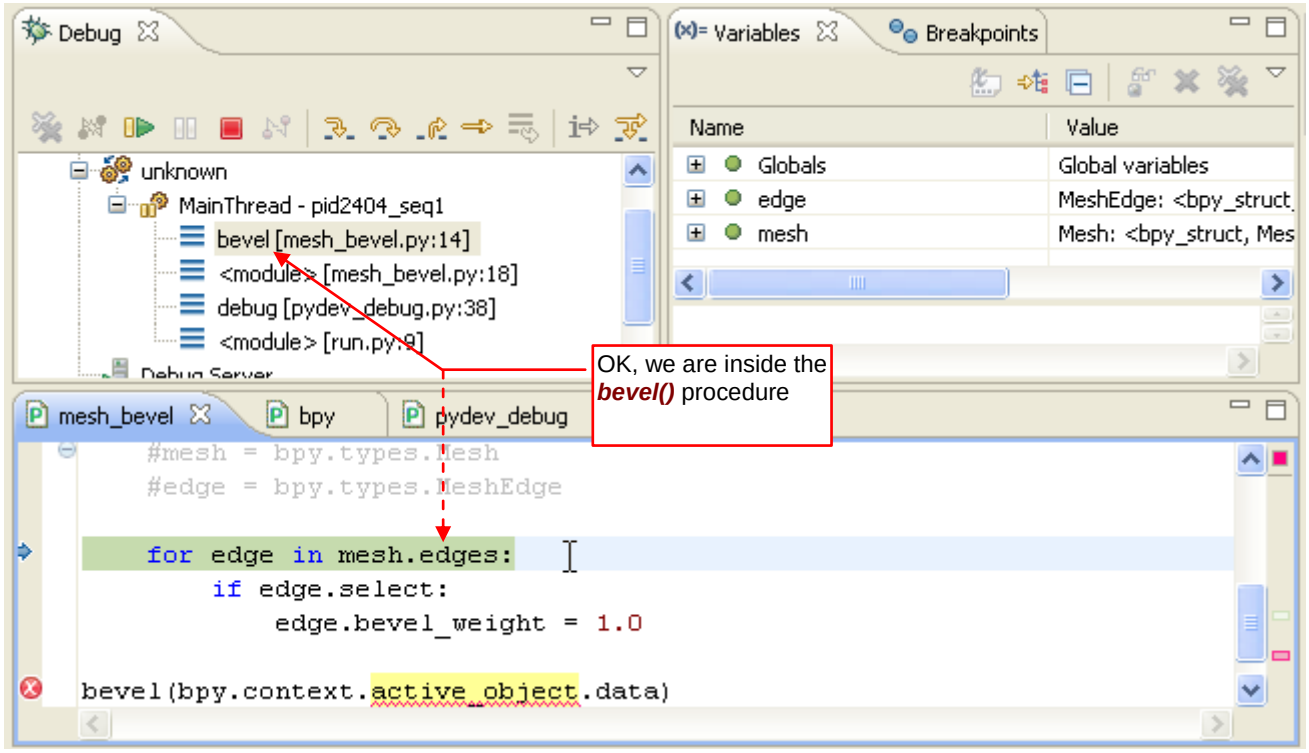


Figure 3.4.10 Tracking the loop code in the *bevel()* function

To keep track of the ***edge*** fields, use the ***Expressions*** panel (Figure 3.4.11 — see also page 126):

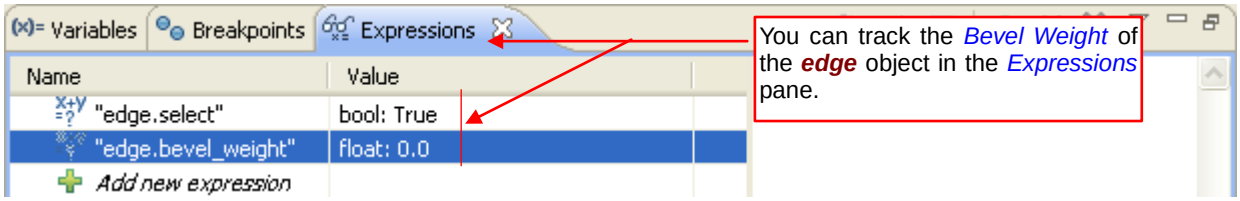


Figure 3.4.11 Tracking the selected fields in the *Expressions* tab

When the procedure is over, press the *Resume* button to finish quickly this script (Figure 3.4.12):

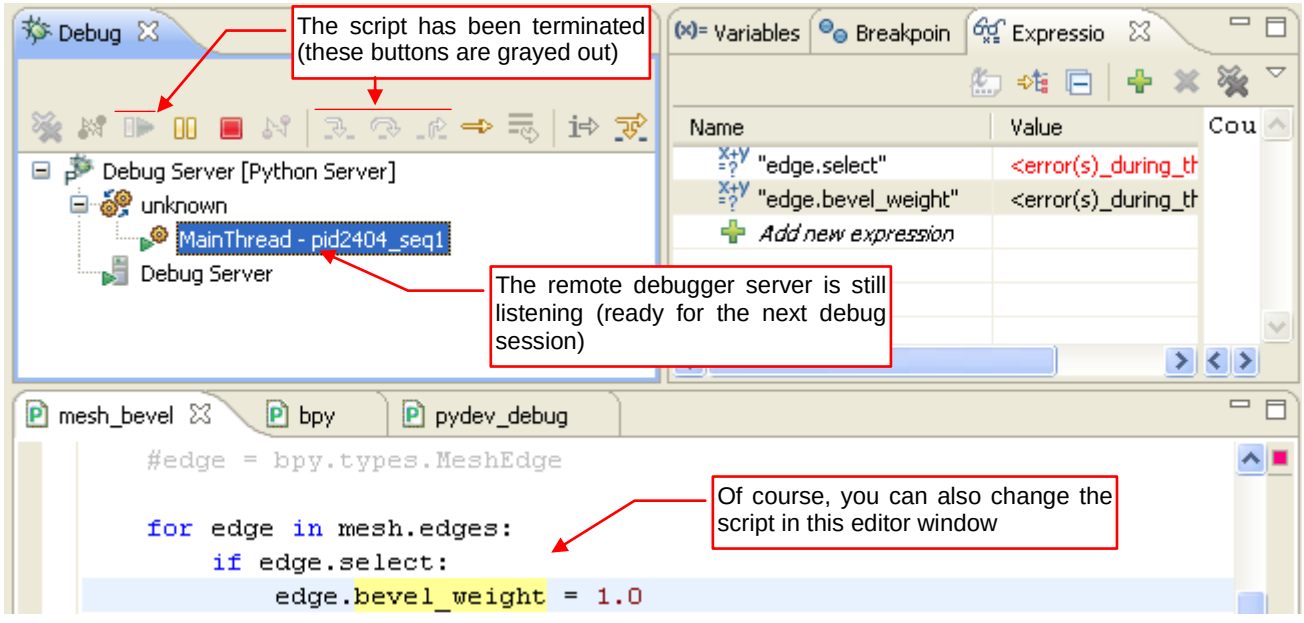


Figure 3.4.12 The state of the environment after the last *Resume* command

If there were an error in this code, the debugger would also terminate this script. Then you can use the code editor from the *Debug* perspective for the correction of minor bugs.

Fortunately, our code has occurred to be free of errors, so far. Let's have a look at the test cube (Figure 3.4.13):

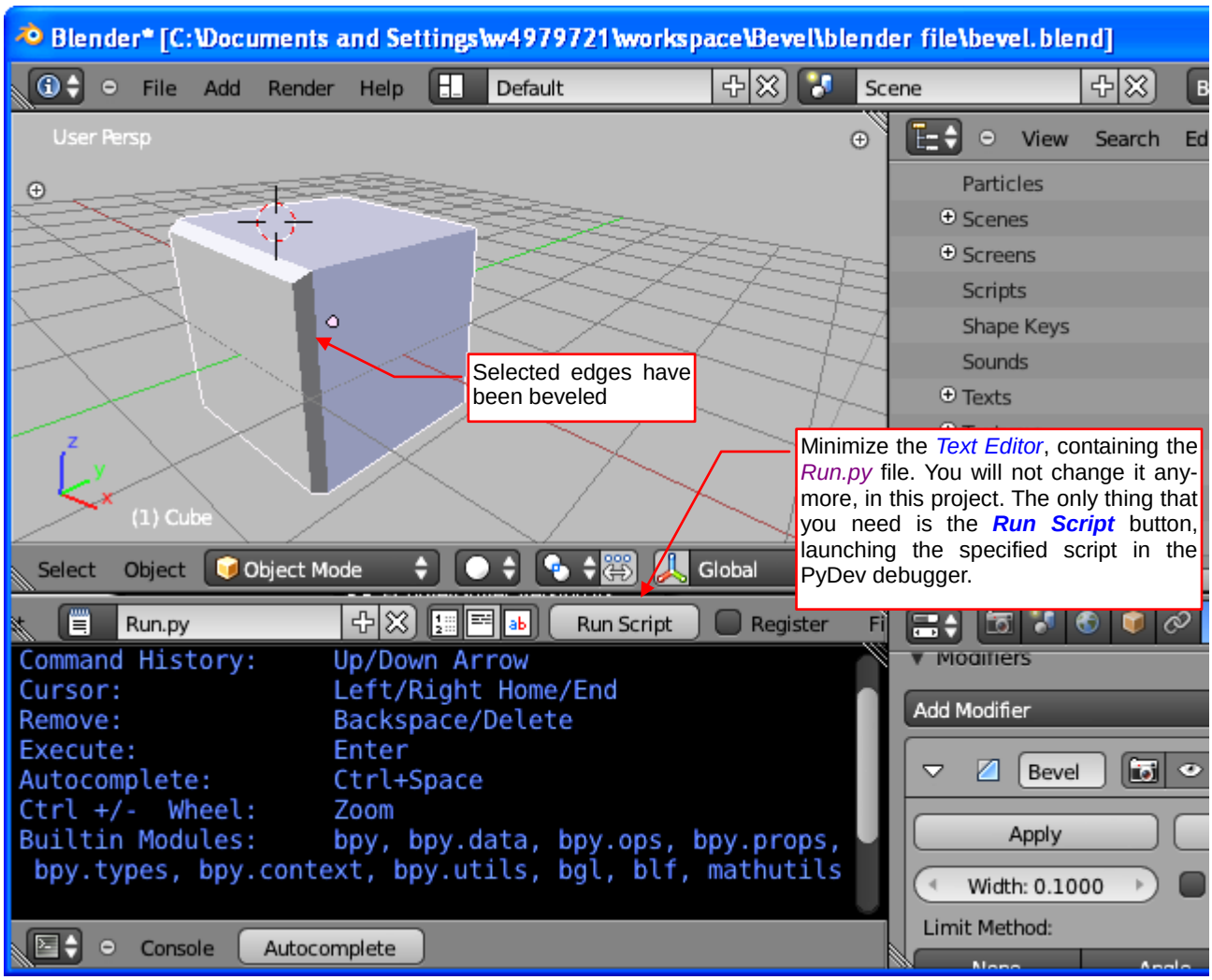


Figure 3.4.13 The result of our script — properly beveled edges

It has been chamfered along the selected edges. It seems that our script is working properly.

To debug again the modified script, just save it and press the ***Run Script*** button on the Blender *TextEditor* header. As long as the PyDev debugger server process is "listening", it automatically breaks the script execution in the *pydev_debug.py* module. You will find yourself back in the place shown in Figure 3.4.7 (page 60). Thus, the best practice is to keep the debug server running all the time. (If you inadvertently press the Blender *Run Script* button when PyDev debug server is not running, Blender will become locked. In this state, you can only close it using the Windows *Task Manager*. So it is better to close the Blender test file first, before closing Eclipse with its PyDev debugger).

I propose to minimize the *Text Editor* that contains the *Run.py* stub, as it is shown in Figure 3.4.13, and save this test Blender environment. (Blender always opens its files with preserved screen layout). Once modified, *Run.py* will not be changed in this project anymore. Just leave the access to its *Run Script* button, to launch easily the *mesh_bevel.py* script after each modification made in Eclipse. This makes the debugging more convenient.

Summary

- To run our 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 our script (page 58);
- 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 58);
- To start the first debug session, activate in Eclipse the ***PyDev Debug Server*** (page 59), then press the ***Run Script*** button in Blender (page 60); To start every subsequent debug session just press the ***Run Script*** button again;
- Do not press the ***Run Script*** button of the *Run.py* script when PyDev debug server is not running, because it will lock Blender. In this state, you can only close it using the Windows task manager. Thus, once you start the PyDev debug server, do not close it until you finish your session of work in Eclipse.
- The debugger always breaks the script execution at certain line of the helper *pydev_debug.py* module (page 60). Therefore, it is a good idea to put at the beginning of our code a breakpoint (page 59). Once you have it, you can quickly go to this line of your script using the *Resume* command (F8);
- To track changes of selected object properties, use the *Expressions* window (page 62);

3.5 Using Blender commands (operators)

Since the "nucleus" of our script works properly, it is time to add to it the other operations. Let's begin with switching from the *Edit Mode* to the *Object Mode*. It should be invoked at the very beginning of the ***bevel()*** procedure. (To not surprise the user, on the end of this procedure we should switch it back to the *Edit Mode*).

How to do it with the Blender API? The case seems obvious: the context object (***bpy.context***) has the ***mode*** field that contains the actual Blender mode. In the *Object Mode* it returns **'OBJECT'**, in the *Edit Mode* — **'EDIT_MESH'**. I always check such things in the Blender *Python Console* (Figure 3.5.1):

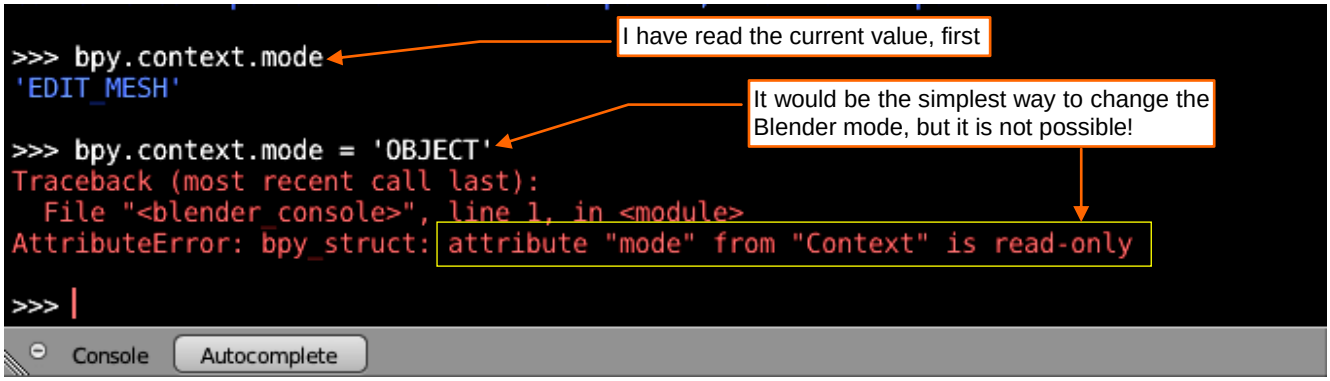


Figure 3.5.1 An attempt to use the *bpy.context.mode* field to change the current Blender mode

So let's try to assign a new value to ***Context.mode*** — it should change the current Blender mode, right? Many of the API fields work this way, but as you can see (Figure 3.5.1), not in this case! ***Context.mode*** is read only. It only returns the current Blender state. We have to find the other way to do it in Python.

The whole Blender GUI uses exclusively the Python API. There must be a way to change the current mode in Python. Do you have an idea, how to find it? The items from the *Mode* menu[1] do not display any tooltips!

In such cases, use the *Info* area. You have seen its header all the time, because it plays the role of the main Blender menu. Enlarge it, dragging this header down (Figure 3.5.2):

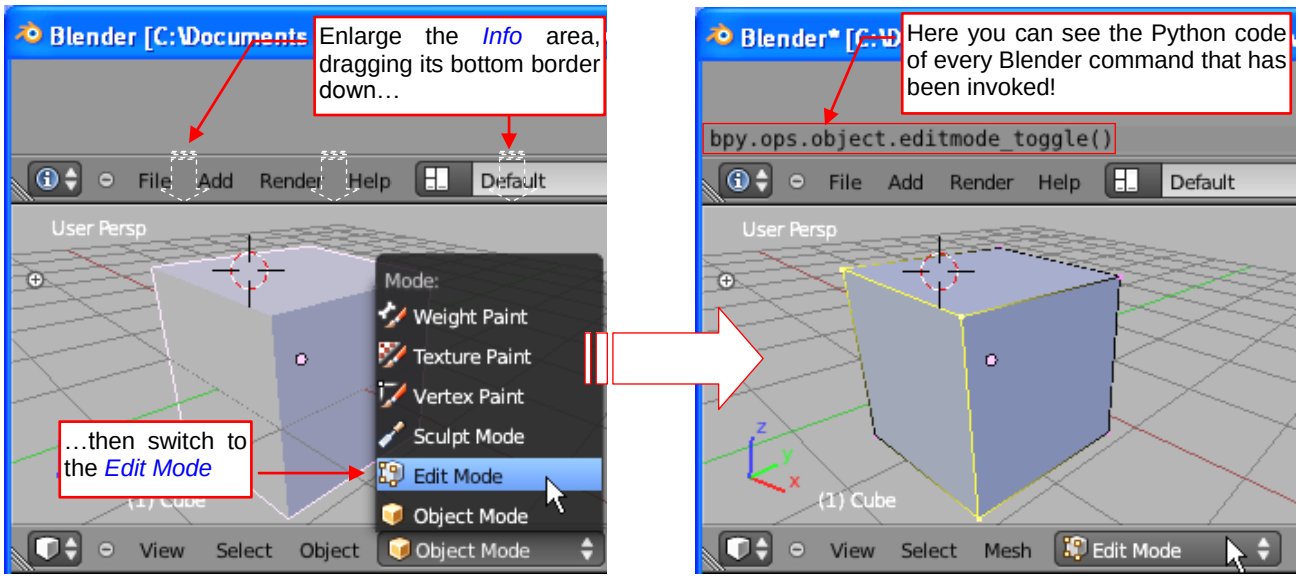


Figure 3.5.2 Checking in the *Info* area the Python API calls that correspond to the issued Blender commands

Now switch Blender from the *Object Mode* into *Edit Mode*. Do you see it? Above the *Info* header something has appeared. It looks like a Python API call (Figure 3.5.2). This is the sought expression!

[1] I mean the menu button located on the *3D View* editor header

Inside the *Info* area, Blender displays the Pythona code of every command that you have invoked from a menu or a panel. It's a kind of the user activity log. You will also see in the *Info* warnings or errors from various Blender components. Therefore, it is worth to look there, from time to time.

OK, but how can we switch back from the *EditMode* to the *Object Mode*? Let's check in the *Info*…Yes, this is not any mistake! To switch back, Blender uses the same method: ***bpy.ops.object.editmode_toggle()***[1].Now that we know how to do this, let's modify the script accordingly (Figure 3.5.3):

```
import bpy

def bevel(mesh):
    """Bevels selected edges of the mesh
       Arguments:
            @mesh (Mesh): a Mesh object, with some edges selected
       It should be called when the mesh is in Edit Mode!
    """
    #mesh = bpy.types.Mesh
    #edge = bpy.types.MeshEdge
    bpy.ops.object.editmode_toggle() #switch into OBJECT mode
    for edge in mesh.edges:
        if edge.select:
            edge.bevel_weight = 1.0
    bpy.ops.object.editmode_toggle() #switch back into EDIT_MESH mode
```

Place all assumptions about using this procedure in its *docstring* comment!

Switching into the *Object Mode*

Switching back to the *Edit Mode*

Figure 3.5.3 First addition — temporary switching from the *Edit Mode* to the *Object Mode* (to set the bevel weights in the mesh)

All right, we have already mastered the Blender mode changes. Let's learn from the invaluable *Info* window, which Blender API method adds the *Bevel* modifier to an object (Figure 3.4.5):

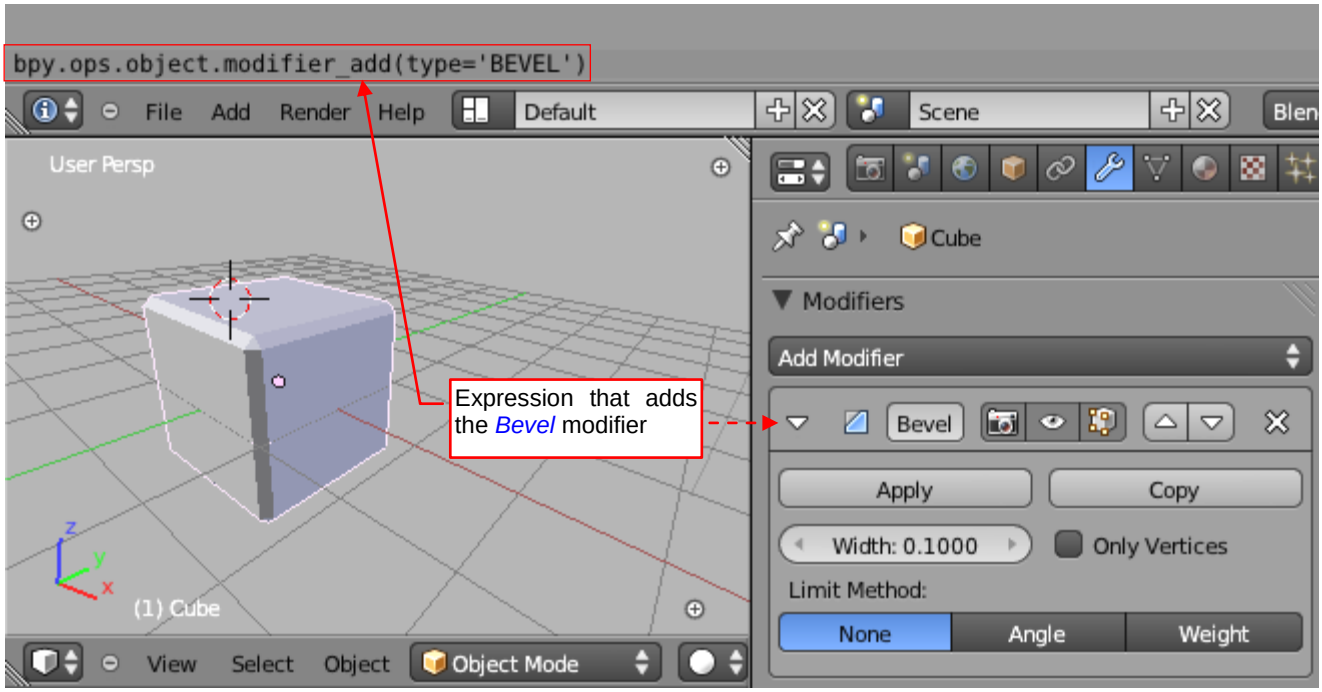


Figure 3.5.4 Testing, which Blender API operator adds the *Bevel* modifier

It turns out that it is ***bpy.ops.object.modifier_add()***. Just call this operator with the ***type*** argument = **'BEVEL'**.

[1] It looks that the *Object Mode* is a kind of the base mode in this program. The Blender API contains in the various ***bpy.ops*** modules methods that allow toggling between the *Object Mode* and any other mode: ***object.posemode_toggle()***, ***paint.vertex_paint_toggle()***, ***paint.weight_paint_toggle()***, ***paint.texture_paint_toggle()***, ***sculpt.sculptmode_toggle()***. The reviewer pointed me, that there is also a universal method: ***bpy.ops.object.mode_set(mode)***. You can use it with appropriate argument, instead of the ****_toggle()*** operators.

Great! It is going well, so let's switch the modifier into the mode of operation which we need. The *Info* window should show us corresponding Python expressions. So change in the *Bevel* modifier panel the *Limit Metod* to the *Weight*, and the weight type — to the *Largest*. All right, we have set them, but… why there is nothing new, in the *Info* window (Figure 3.5.5)?

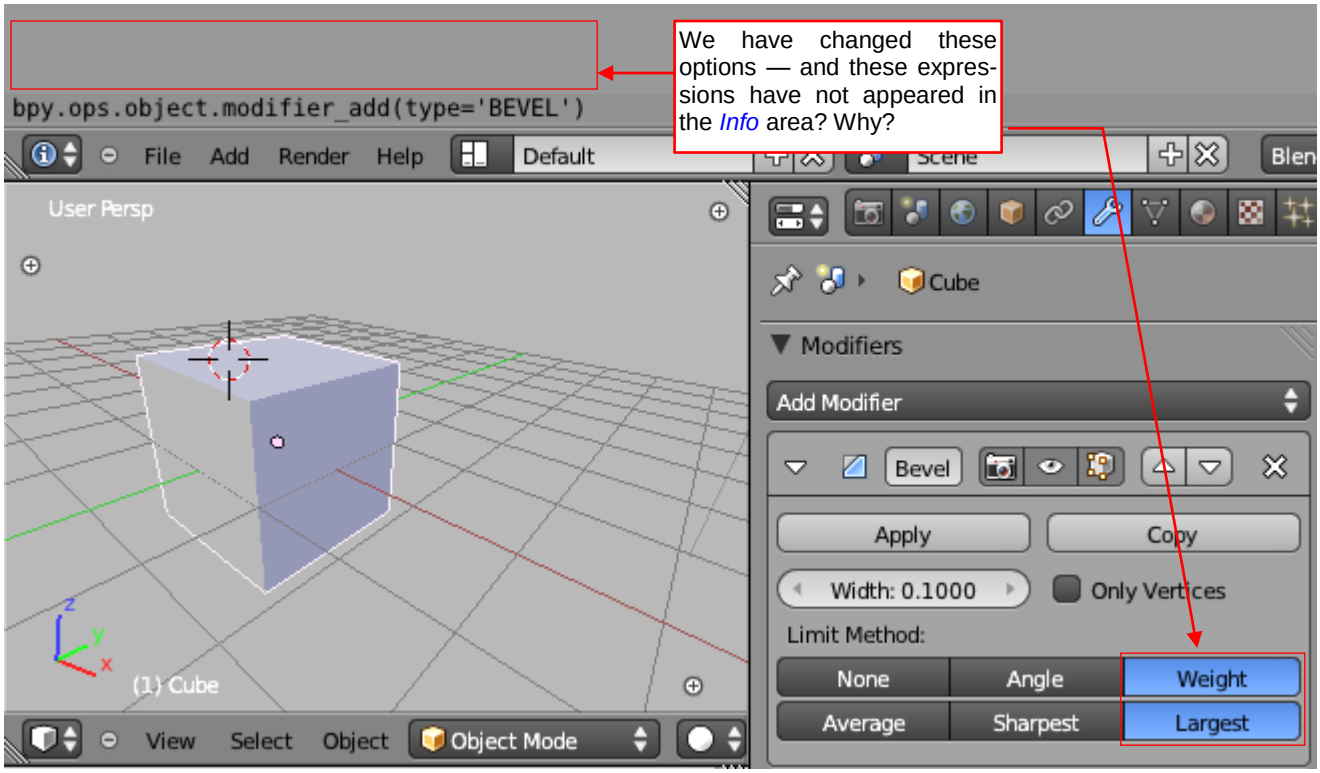


Figure 3.5.5 Missing expressions in the *Info* area

It occurs that the *Info* area does not show everything. No Blender command (i.e. operator) was called, when you clicked on the panel options. It just changed the values of two fields in the modifier object.

What were these fields? You have to read it from their tooltips (Figure 3.5.6):

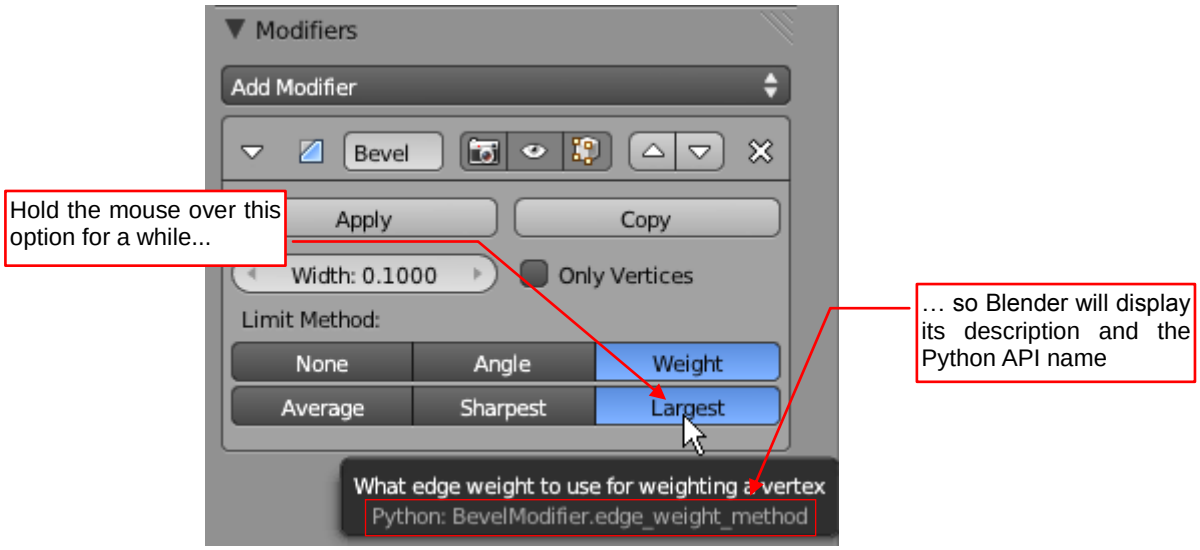


Figure 3.5.6 Reading the corresponding Python API name

From the tooltip can be seen that all three values of the first row (*None*, *Angle*, *Weight*) reflect three possible states of the ***BevelModifier.limit_method***. The options from the second row (*Average*, *Sharpest*, *Largest*) correspond to the three possible states of the ***BevelModifier.edge_weight_method*** (Figure 3.5.6).

- In Blender, the rows of alternate options often reflect the possible states of a single Python API field

All right. We already know the name of the fields that need to be changed, but how to reach the modifier object through the data hierarchy? To find the answer to this question, review the data structure of our **Cube** object in the *Outliner* (Figure 3.5.7):

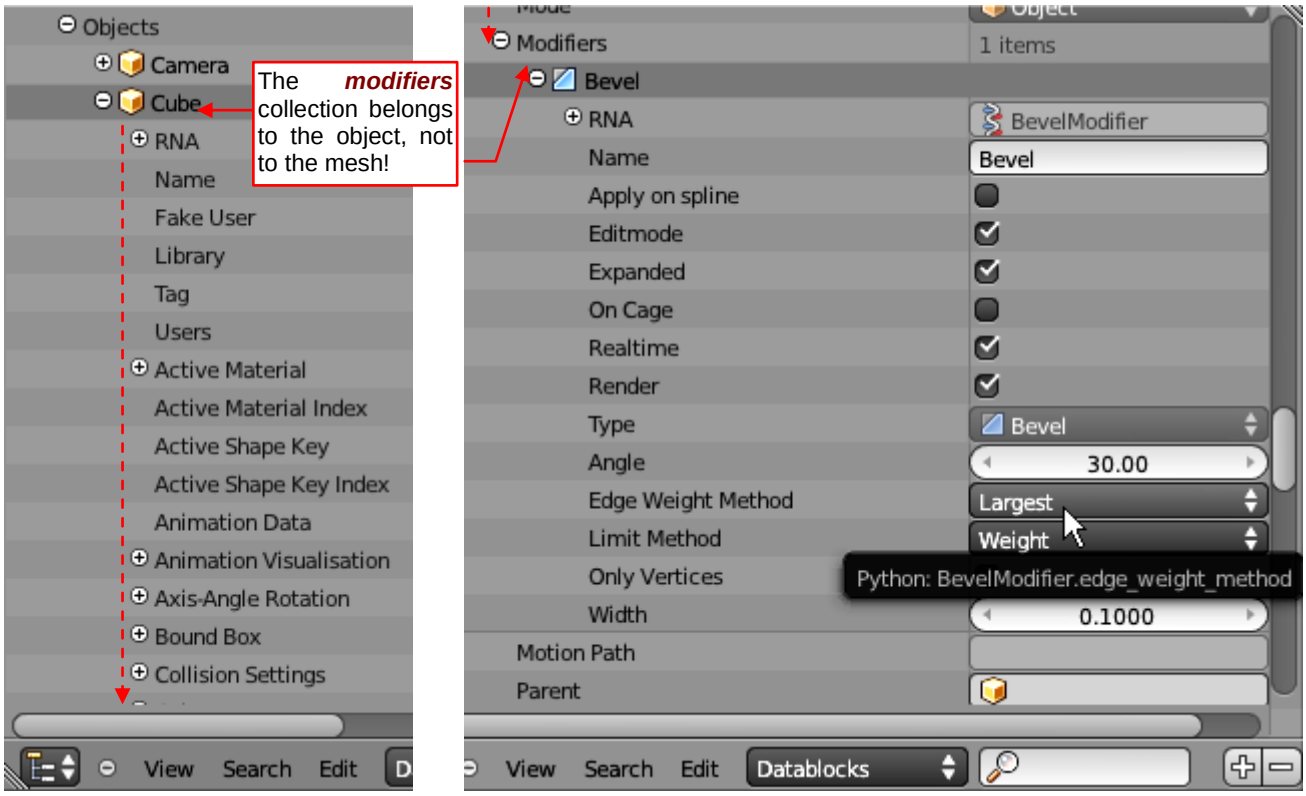


Figure 3.5.7 Finding the *modifiers* collection

You will find this modifier in the **Cube** object itself. It is an element from the ***Object.modifiers***[1] collection. You already know how to get the active object, so we have identified the whole „name path" to these modifier fields.

It remains to ascertain the Python values, which we have to give to these attributes. Their Blender API descriptions are very laconic, sometimes simply enumerate the possible values without any comment. Therefore, I always prefer to check their values in the Python console (Figure 3.5.8):

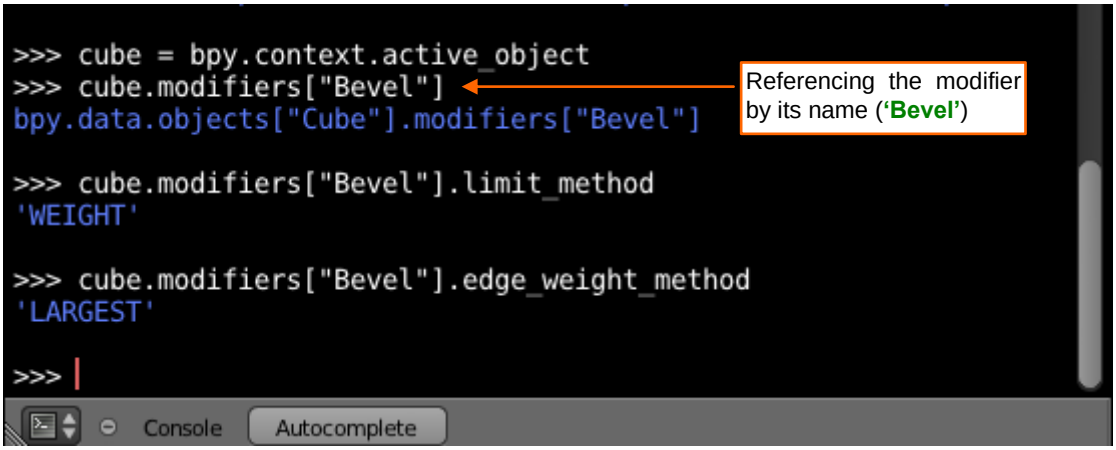


Figure 3.5.8 Checking the values of modifier object fields (in the *Python Console*)

[1] This placement means that can you use the same mesh in different objects, and each of them can have different set of modifiers. One of them can "smooth" it with the *Subdivision Surface* modifier, another — bend it along a curve. In the result, you can create, from a single mesh, many objects of completely different shape. It is worth to remember about such things - they are sometimes useful to enhance our work on a model!

What operator corresponds to the *Apply* button on the modifier panel? You can also use the information from its tooltip, here (Figure 3.5.9):

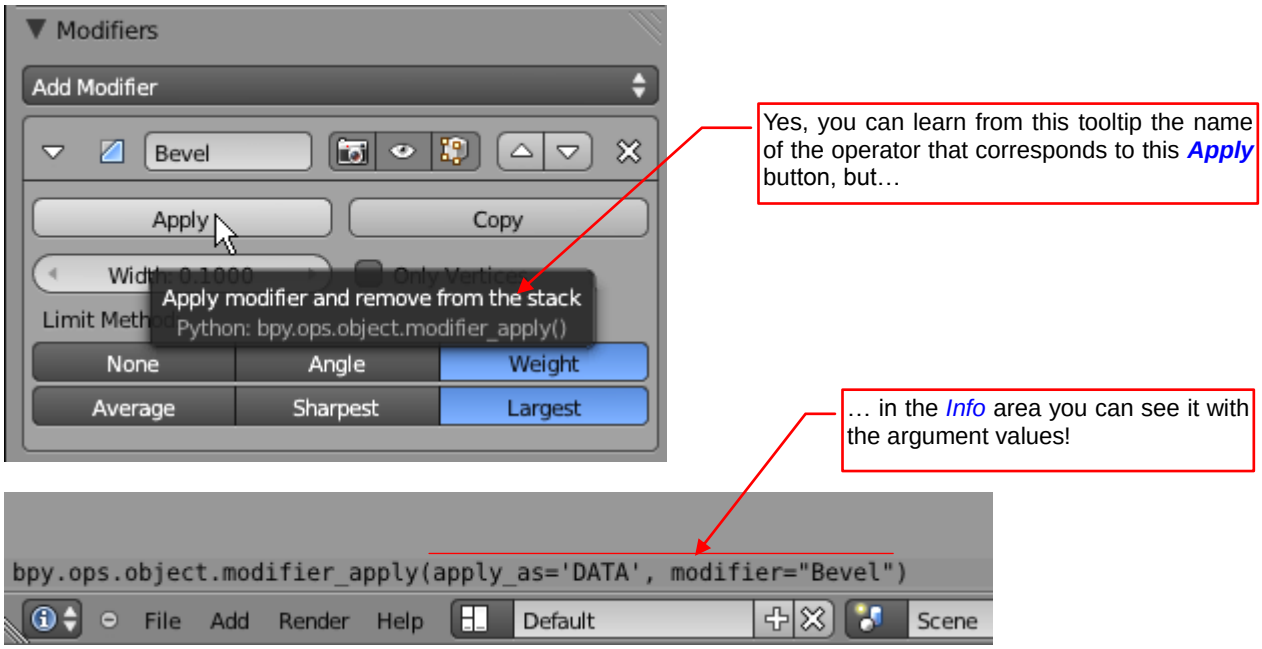


Figure 3.5.9 Comparing the information delivered by the *Info* window and the control tooltip

Yet when you perform this operation, you will see in the *Info* window the exact expression, with all the argument values. It is very helpful. For example, in the Blender API documentation you can read about the ***apply_as*** argument following description: „*How to apply the modifier to the geometry*". Guided by this hint, you would not be able to discover that you have to set its value to **'DATA'**!

We have all the information, so we can extend our procedure now (Figure 3.5.10):

```
import bpy

def bevel(obj):
    """Bevels selected edges of the mesh
       Arguments:
            @obj (Object): an object with a mesh.
                           It should have some edges selected
       This function should be called in the Edit Mode, only!
    """
    #edge = bpy.types.MeshEdge
    #obj = bpy.types.Object
    #bevel = bpy.types.BevelModifier
    bpy.ops.object.editmode_toggle() #switch into
    #adding the Bevel modifier
    bpy.ops.object.modifier_add(type = 'BEVEL')
    bevel = obj.modifiers[-1]#the new modifier is always added at the end
    bevel.limit_method = 'WEIGHT'
    bevel.edge_weight_method = 'LARGEST'

    for edge in obj.data.edges:
        if edge.select:
            edge.bevel_weight = 1.0

    bpy.ops.object.modifier_apply(apply_as = 'DATA', modifier = bevel.name)

    bpy.ops.object.editmode_toggle() #switch back into EDIT MESH mode

bevel(bpy.context.active_object)
```

I have changed my mind about the type of the argument:
It is better to pass the whole ***Object***, not just its ***Mesh***! (The modifier is added to ***Object***).

Add new *Bevel* modifier to the ***obj*** object. Set it into appropriate mode

The new modifier is always appended to the end of the ***modifiers*** collection

We have changed type of this procedure input, so this expression also has been updated

Applying the modifier.

Now we are passing to this procedure just the ***active_object***

Figure 3.5.10 Script development: adding the *Bevel* modifier

As you can see from my explanations (Figure 3.5.10), during implementation of the modifier handling I have decided to change the type of the ***bevel()*** input data from ***Mesh*** to ***Object***. (Because modifiers belong to the object, not the mesh). Such changes always require some attention. You have to do simultaneous changes in many different places of the code. If you will forget about any of them then you will have an error, later on.

The code in Figure 3.5.10 has a flaw. It was written „for the test data". During the practical use, it may happen that you will invoke it against an object that already contains other modifiers. For example — the *Subsurf* smoothing (Figure 3.5.11);

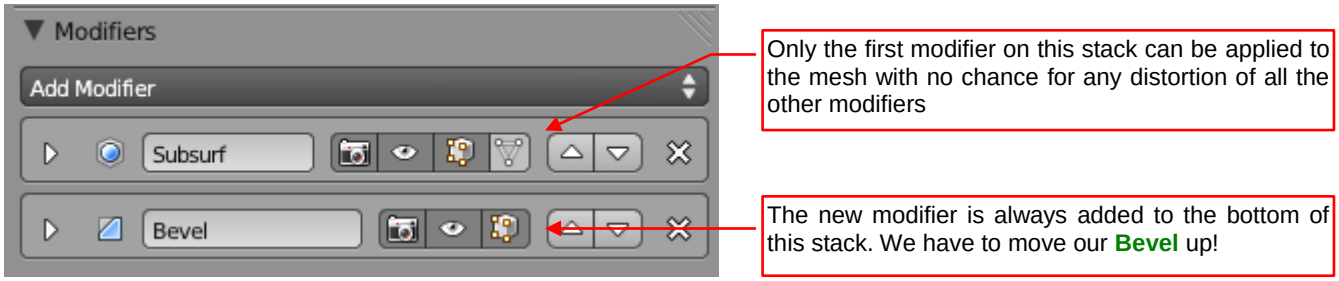


Figure 3.5.11 The problem with the modifier position on the stack

The new modifier, as our *Bevel*, is always appended to the end of modifier list (stack). It has to be the first one, to be applied to the mesh without any chance for an unwanted effect. Using the *Info* window you will quickly find, that there is the ***bpy.ops.object.modifier_move_up()*** operator. We have to use this method in a loop, moving our modifier up until it will become the first one (Figure 3.5.12):

```
import bpy

def bevel(obj, width):
    """Bevels selected edges of the mesh
       Arguments:
            @obj (Object): an object with a mesh.
                           It should have some edges selected
            @width (float):width of the bevel
       This function should be called in the Edit Mode, only!
    """
    #edge = bpy.types.MeshEdge
    #obj = bpy.types.Object
    #bevel = bpy.types.BevelModifier

    bpy.ops.object.editmode_toggle() #switch into OBJECT mode
    #adding the Bevel modifier
    bpy.ops.object.modifier_add(type = 'BEVEL')
    bevel = obj.modifiers[-1] #the new modifier is always added at the end
    bevel.limit_method = 'WEIGHT'
    bevel.edge_weight_method = 'LARGEST'
    bevel.width = width
    #moving it up, to the first position on the modifier stack:
    while obj.modifiers[0] != bevel:
        bpy.ops.object.modifier_move_up(modifier = bevel.name)

    for edge in obj.data.edges:
        if edge.select:
            edge.bevel_weight = 1.0

    bpy.ops.object.modifier_apply(apply_as = 'DATA', modifier = bevel.name)

    bpy.ops.object.editmode_toggle() #switch back into EDIT_MESH mode

bevel(bpy.context.active_object, 0.1)
```

I have added another argument: the bevel width

Setting the bevel width

The loop, which moves our modifier to the top of the stack

The width for the tests

Figure 3.5.12 Script development: enhancements in the modifier handling

On the end, I have added to the ***bevel()*** code second argument: the bevel ***width*** (see Figure 3.5.12). In the test call to this procedure, it is set to 0.1 Blender units. Let us prepare the environment for testing, and we will be ready for the next debug session (Figure 3.5.13):

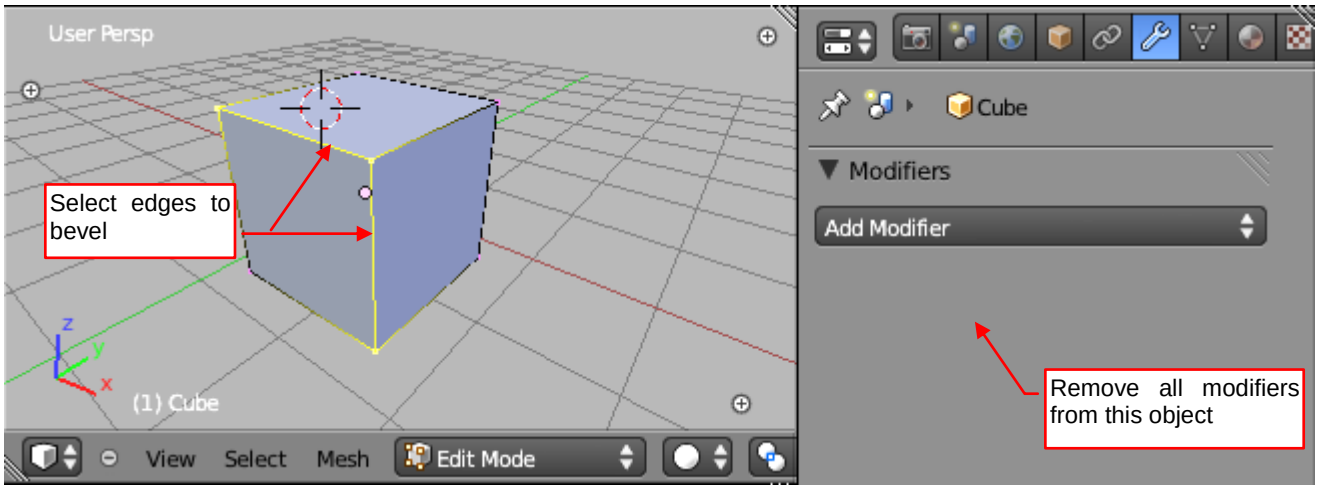


Figure 3.5.13 Preparation for another test

Select some edges of the mesh and press the *Run Script* button. On the first run after major changes, always follow the execution of your script in the debugger. Fortunately, it has finished without errors. Figure 3.5.14 displays its result:

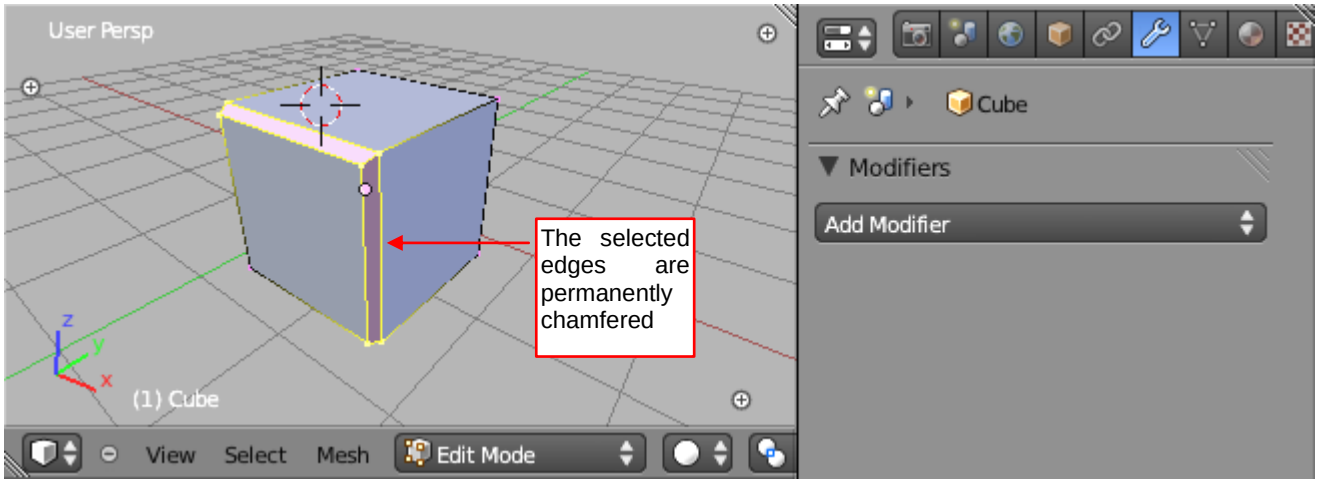


Figure 3.5.14 The result of the script run

It looks as expected: the selected edges have been beveled. Yet it is worth to check the distribution of the *Bevel Weight* values on these newly created edges (Figure 3.5.15):

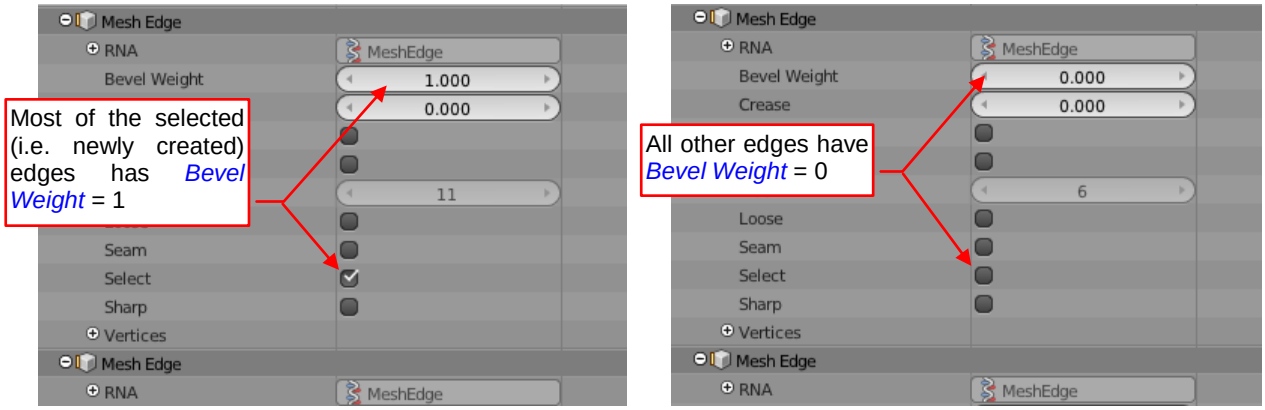


Figure 3.5.15 The propagation of the *Bevel Weight* values on the newly created edges

In the place of two originally selected edges the script has created some new ones (see Figure 3.5.14). All of these newly created edges are selected, and most of them (but not all!) have a non-zero *Bevel Weight*. I have not found any edge in this cube, which has *Bevel Weight* > 0 and is selected... So I assume that these new edges "inherit" from the original ones their state, such as the selection and *Bevel Weight*. The latter can cause unexpected results at the next *Bevel* operation. (It would modify not only the edges selected by the user, but also the others, which inherited *Bevel Weight* in the previous operations). Therefore, our script should “clean up” the mesh on the end of the ***bevel()*** procedure, clearing the ***bevel_weight*** values of the selected edges (Figure 3.5.16):

```
def bevel(obj, width):
    """Bevels selected edges of the mesh
       Arguments:
            @obj (Object): an object with a mesh.
                           It should have some edges selected
            @width (float):width of the bevel
       This function should be called in the Edit Mode, only!
    """
    #edge = bpy.types.MeshEdge
    #obj = bpy.types.Object
    #bevel = bpy.types.BevelModifier

    bpy.ops.object.editmode_toggle() #switch into OBJECT mode
    #adding the Bevel modifier
    bpy.ops.object.modifier_add(type = 'BEVEL')
    bevel = obj.modifiers[-1] #the new modifier is always added at the end
    bevel.limit_method = 'WEIGHT'
    bevel.edge_weight_method = 'LARGEST'
    bevel.width = width
    #moving it up, to the first position on the modifier stack:
    while obj.modifiers[0] != bevel:
        bpy.ops.object.modifier_move_up(modifier = bevel.name)

    for edge in obj.data.edges:
        if edge.select:
            edge.bevel_weight = 1.0

    bpy.ops.object.modifier_apply(apply_as = 'DATA', modifier = bevel.name)

    #clean up after applying our modifier: remove bevel weights:
    for edge in obj.data.edges:
        if edge.select:
            edge.bevel_weight = 0.0

    bpy.ops.object.editmode_toggle() #switch back into EDIT_MESH mode
```

Removing „inherited” bevel weights from the newly created edges

Figure 3.5.16 The ultimate version of the *bevel()* procedure

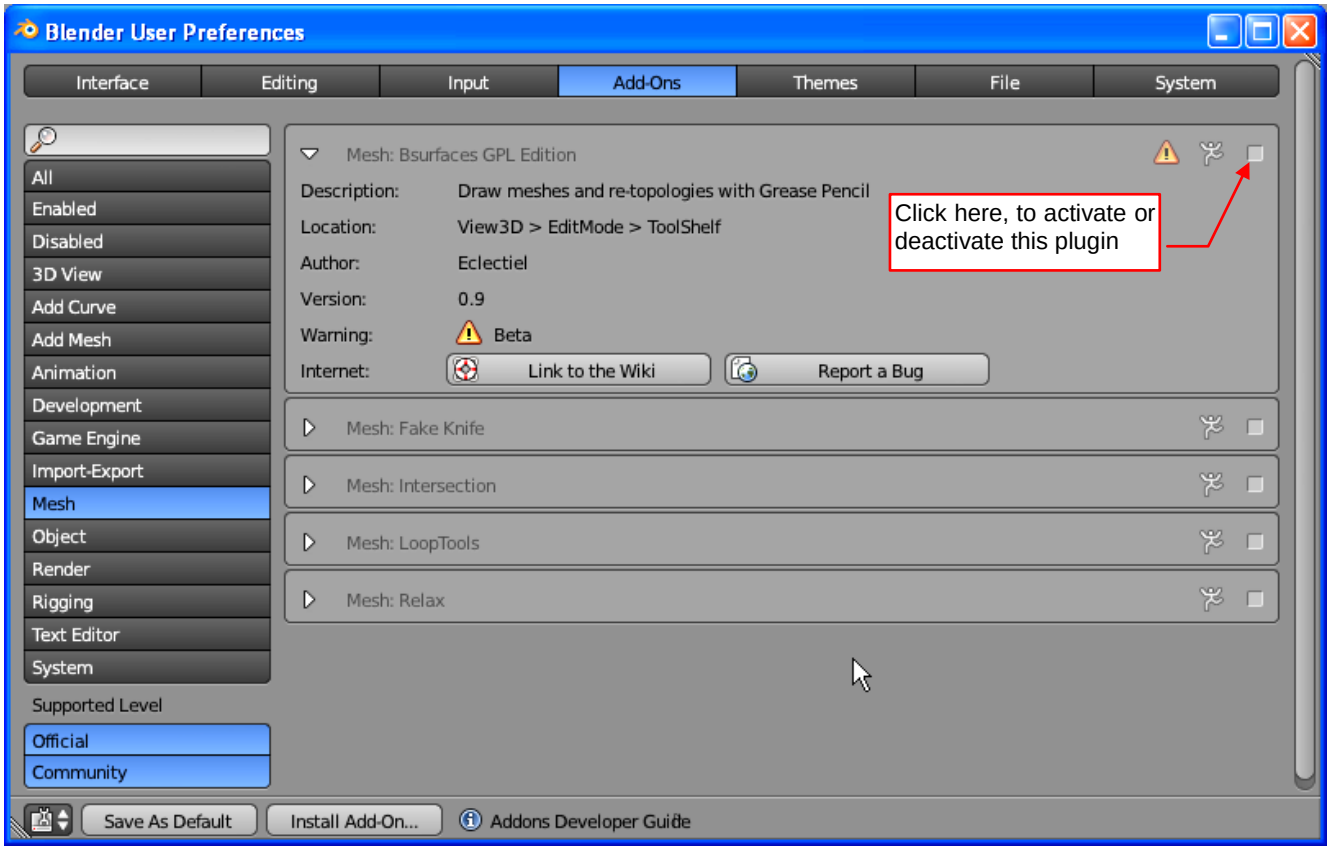
The ***bevel(object, width)*** procedure is ready. Notice that there is no input data validation, yet. We will implement this part in the next chapter, where our script will become a Blender add-on.

Summary

- You can trace the exact Python expressions that correspond to the Blender commands in the ***Info*** area (page 66). Unfortunately, it shows the commands (operator) calls, only, You cannot trace there the changes of the panel options (page 67);
- You can read from the last line of a control tooltip the corresponding name of the Python API field (property) (page 67);
- It is a good practice to check in the Blender *Python Console* the values of a Python API field, before using it in the code (page 68);
- To use a Blender command (the operator) within your script, simply call corresponding method from the ***bpy.ops*** module. The basic technique is to combine these methods with the code that checks the state of the Blender data after such change. For example, the moving of newly added modifier to the beginning of the modifier stack is implemented in this way (page 70);

Chapter 4. Converting the Script into Blender Add-On

Probably you know the *Blender User Preferences* window. I suppose that you already noticed 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), which you actually need. During its 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 am showing how to convert our Blender script into a Blender plugin. This add-on will add to the mesh *Specials* menu the “destructive” *Bevel* command.

4.1 Adaptation of the script structure

So far, our script has been "linear" - it executes 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):

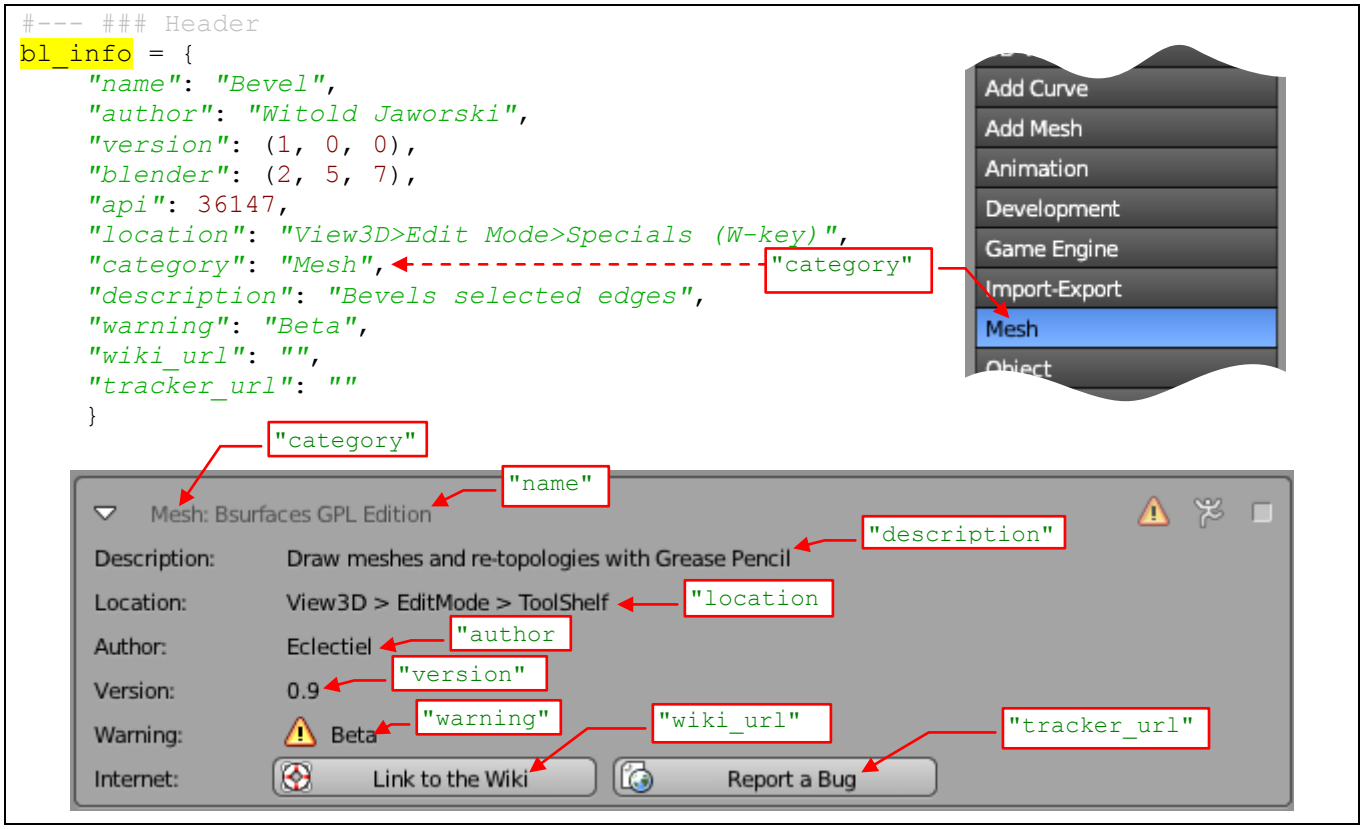


Figure 4.1.1 The *bl_info* structure and its pane in the *User Preferences* window

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 visible in the *All* category, only.

This plugin has to expose our ***bevel()*** method as a new Blender command. To make this possible, we have to embed our procedure in a simple operator class (Figure 4.1.2):

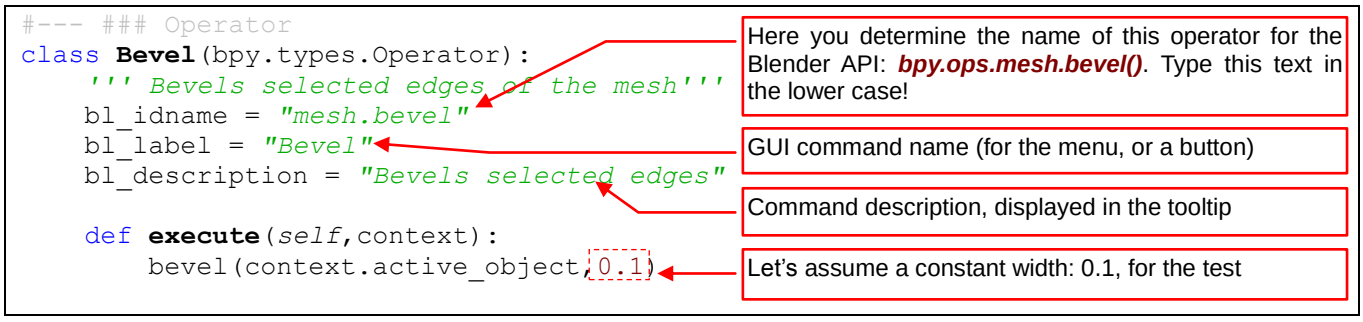


Figure 4.1.2 The operator class, "wrapped around" the *bevel()* procedure.

I gave this class the ***Bevel*** name (call it as you wish). This new operator must inherit from the abstract ***bpy.types.Operator*** class. Otherwise, it will not work properly.

Each 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 will have just one method, with a strictly specified name and parameter list: ***execute(self, context)***. Place inside it the call to the ***bevel()*** procedure, with the fixed bevel width (just for the tests). At this stage, I still do not add any input data (***context***) validation.

To register in Blender all such classes from your module, you must add to the script two special functions, responsible for this operation. This code usually looks the same: at the beginning import two helper functions from the ***bpy.utils*** module. Use them at the end of the script, in two methods that must have names: ***register()*** and ***unregister()*** (Figure 4.1.3):

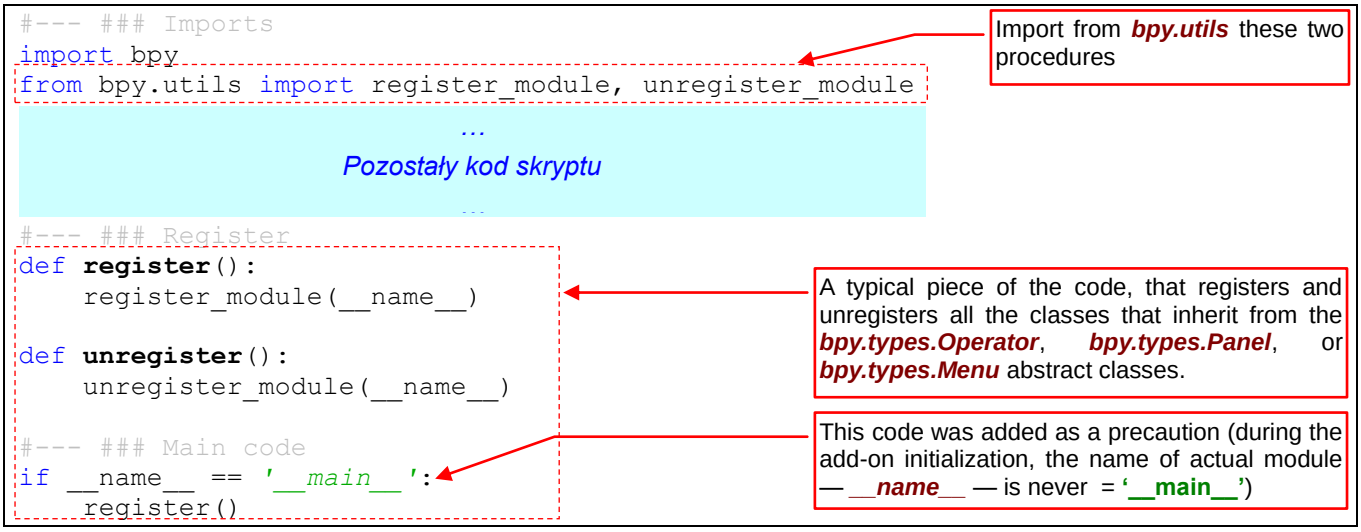


Figure 4.1.3 The code that registers in the Blender environment the API classes, defined in the script.

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

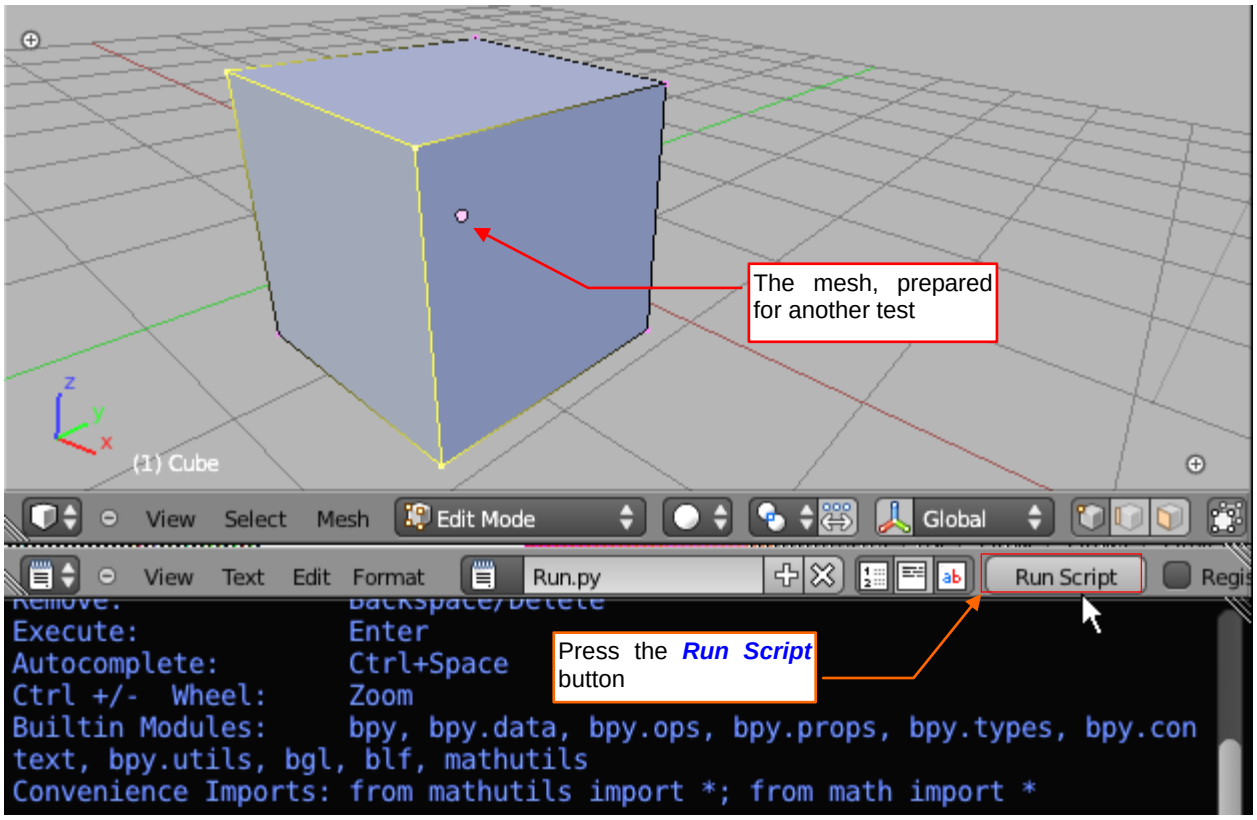


Figure 4.1.4 Launching our add-on in the debugger.

What about the result? It seems that the execution of this script has passed without any error, but the selected edges of the mesh are not chamfered? To make sure, add a breakpoint to the ***Bevel.execute()*** method, and run this script again. Nothing happens, and this breakpoint is never reached (Figure 4.1.5):

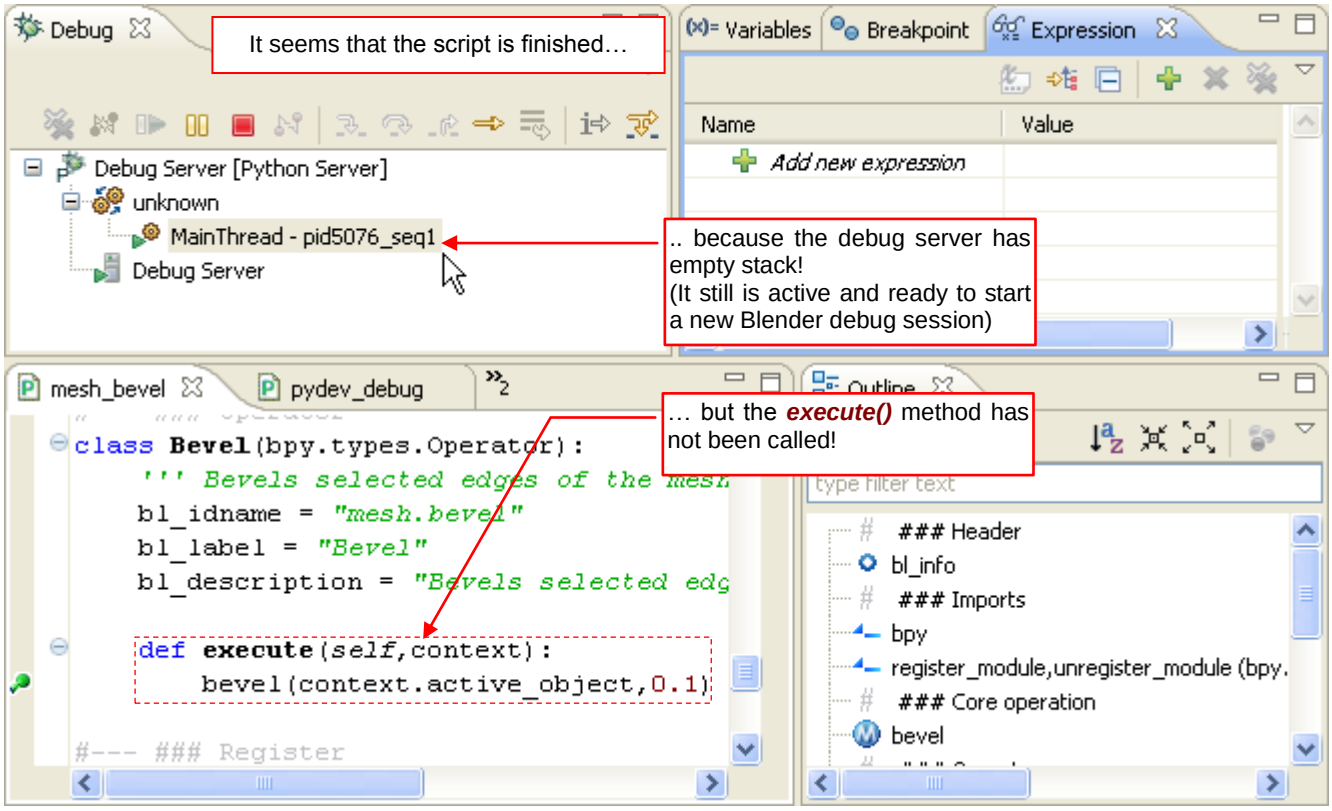


Figure 4.1.5 The state of the script after the first *Resume* (F8) command

The point is that currently the main script code does not call the ***bevel()*** procedure. It just registers a new Blender command (operator), under the name that you have assigned to the ***Bevel.bl_name*** field. In our case it is just **„mesh.bevel"** (see page 75, Figure 4.1.2). Check in the Python console, whether the ***bpy.ops.mesh.bevel*** method exists (Figure 4.1.6):

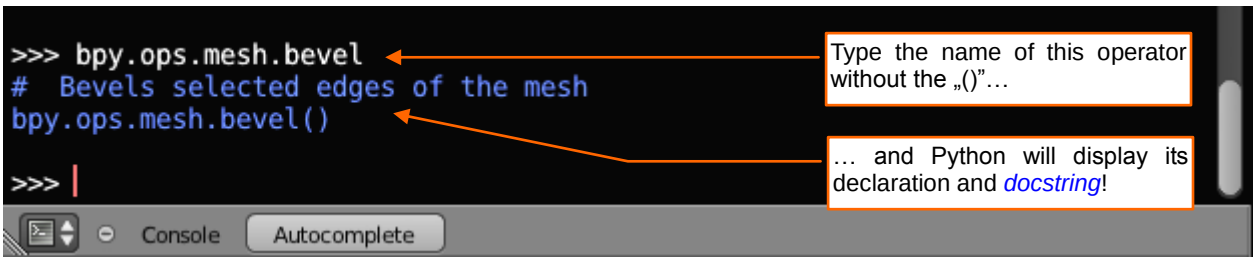


Figure 4.1.6 Checking results of the add-on registration

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" — in the *Python Console* (Figure 4.1.7):

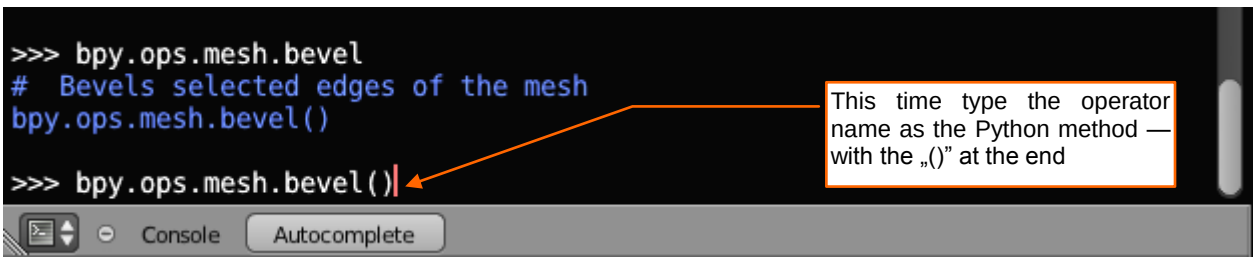


Figure 4.1.7 Call the operator…

The Blender window has become locked, and the Eclipse the PyDev debugger is activated. It waits at the breakpoint, we have placed inside the ***execute()*** method (Figure 4.1.8):

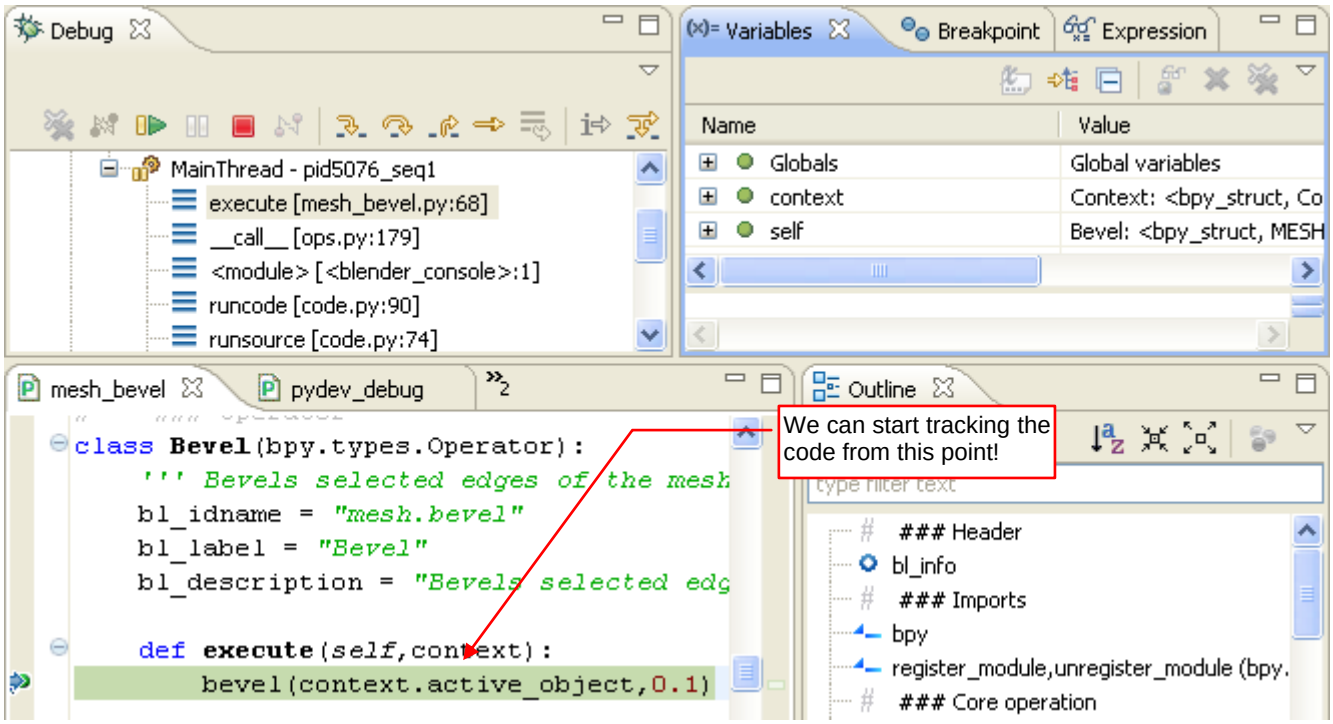


Figure 4.1.8 …and the debugger will stop its execution at your breakpoint

Do you see? We have simulated here what Blender will do with our operator. When you call the ***bpy.ops.mesh.bevel()*** method (usually from a menu or a panel button), Blender will create a new instance of the ***Bevel*** class. This ***Bevel*** object is used just to call its ***execute()*** method. After this call Blender releases (discards) the operator object. 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: notice the arguments of this procedure, exposed in the *Variables* pane. Expand the ***context*** parameter to see what kind of information can be obtained from this object (Figure 4.1.9):

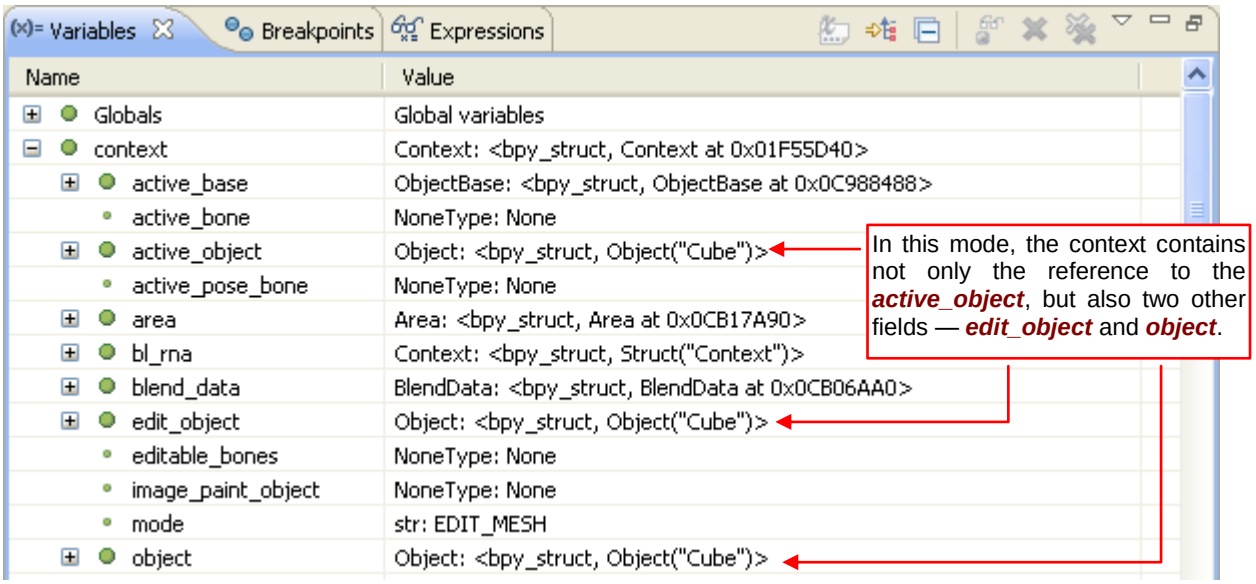


Figure 4.1.9 Previewing the context of this call

The ***context*** structure may have 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 in the Blender API pages.

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

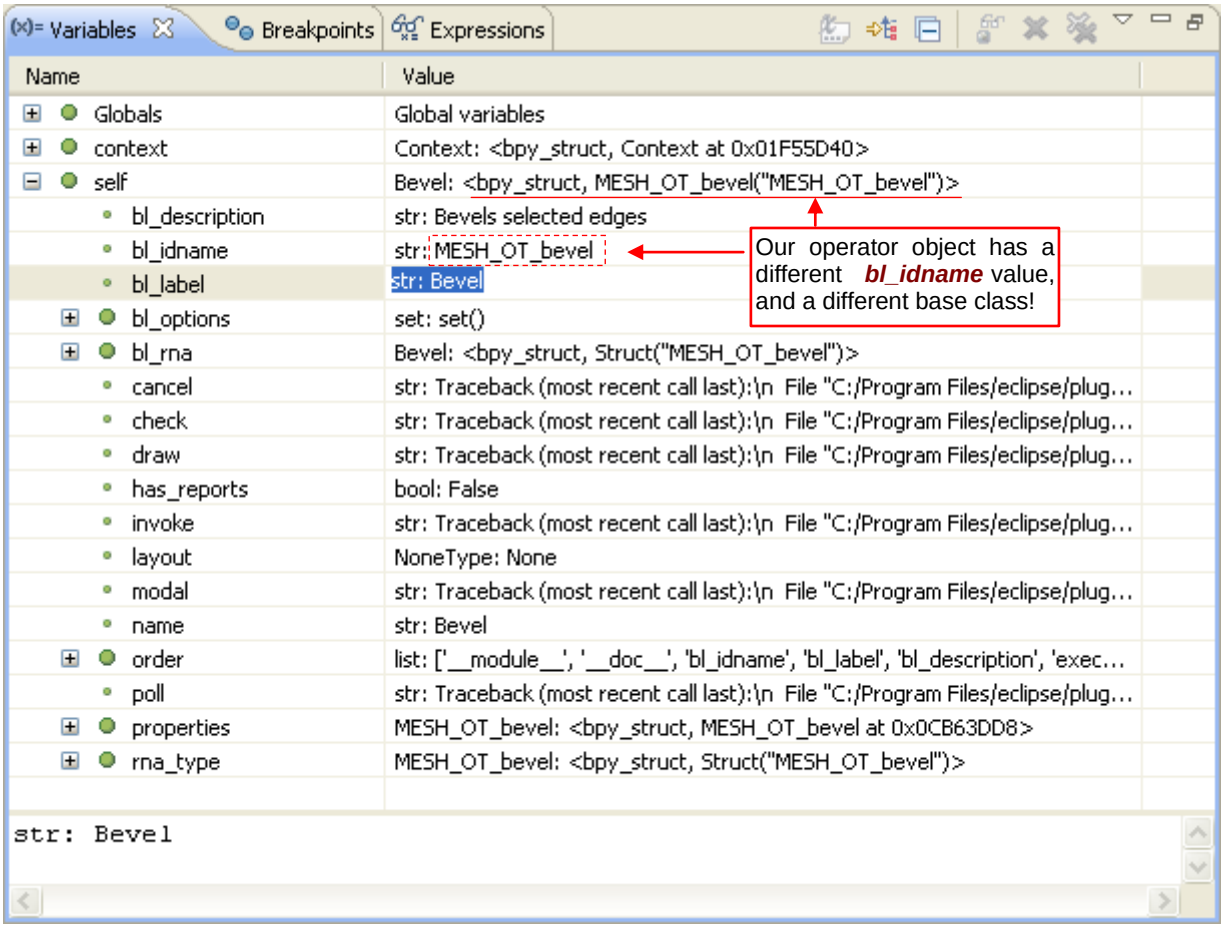


Figure 4.1.10 The content of the operator class (*self*)

Calm down: it is normal. It seems that Blender guided by the first member of the ***bl_idname*** value (**„mesh.bevel"**), has created for our operator a class named ***MESH_OT_bevel***. (The **„mesh.**" prefix is replaced in the class name with the „MESH_OT_" string. Maybe the rule is that Blender replaces every dot („.") in the operator symbol with „_OT_"?) If you are curious about this, 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, there! Their names always contain „_OT_", „_MT_", or „_PT_". They are the operators, menus and panels defined in 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 60), or in Figure 3.4.10 (page 62).

At the bottom of the stack, you can see the functions of the *Python Console* (it seems that its large part is also written in Python). Then there is a single line from a „*<blender console>*" module. This is the invocation of our operator, which we have typed in the console. As you can see, it has called a method from the *ops.py* Blender module, which has created this instance of our ***Bevel*** class and called its ***execute()*** method.

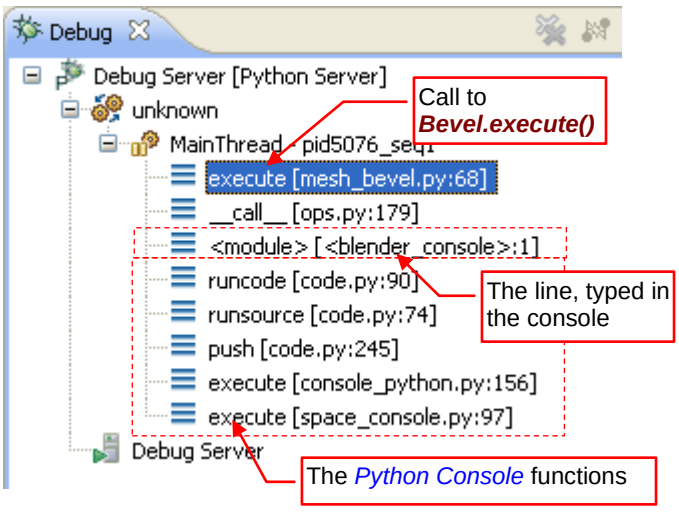


Figure 4.1.11 The stack of the operator called from the console

When you finish the last step of the ***execute()*** function — the call to ***bevel()*** — the next *Step Over* (F6) will bring you to the Blender *ops.py* module (Figure 4.1.12):

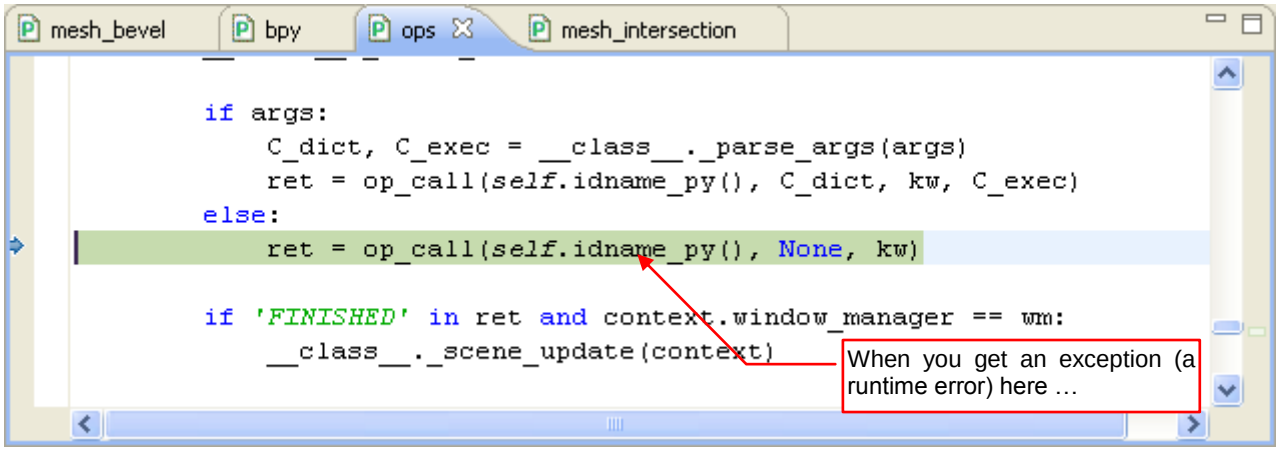


Figure 4.1.12 The line of Blender internal script, that has called the *Bevel.execute()* method

We are here specifically to show you the behavior of the PyDev debugger in case of the Python error. When it occurs, the green highlight of the current line disappears (Figure 4.1.13):

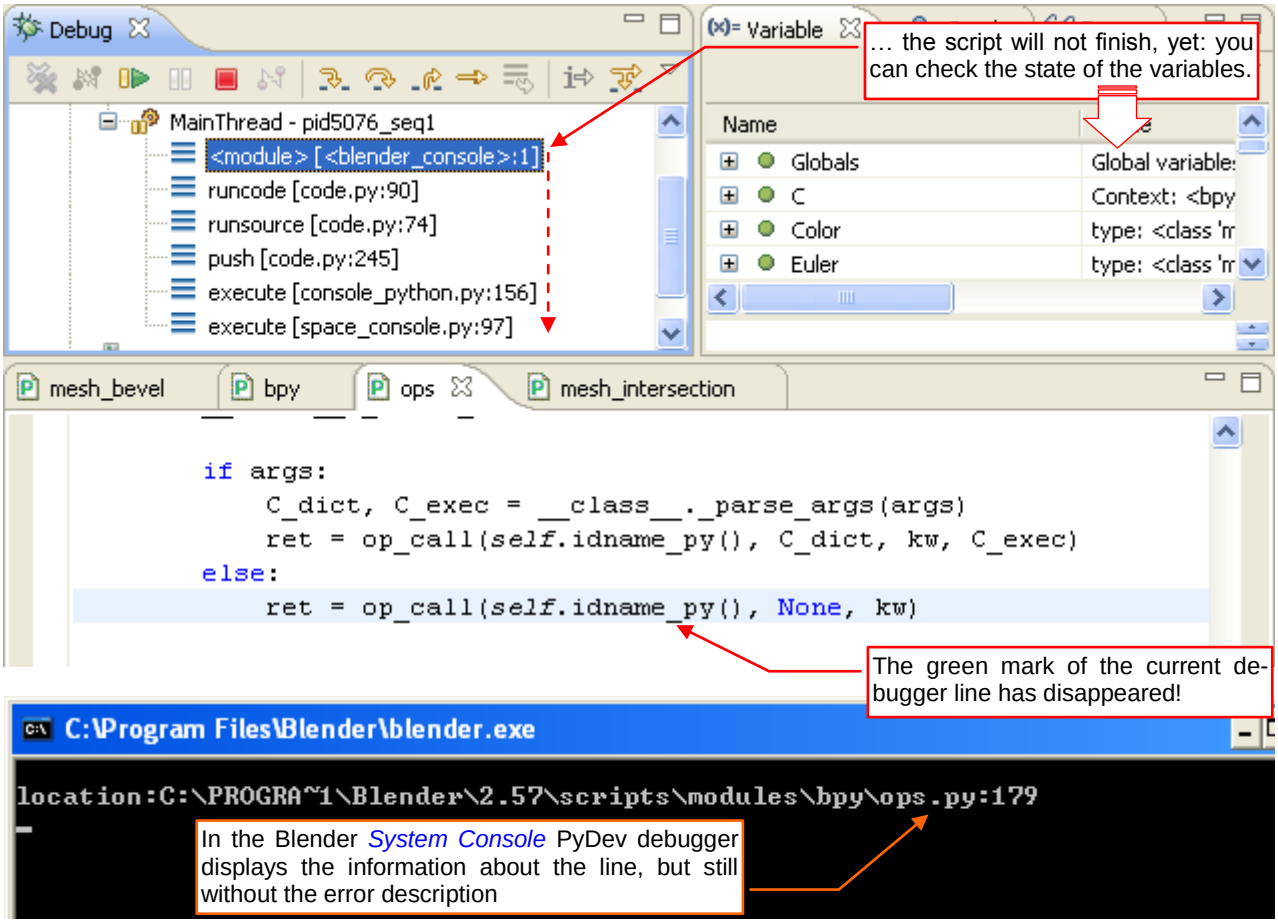


Figure 4.1.13 The state of the debugger in the case of a script error

At the same time in the Blender *System Console* debugger prints a message about the script name and the line number where the error has occurred. Despite this, the script 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 local and global variables. Usually, careful examination of their contents will allow you to determine the cause of the problem. One element is still missing among this information: the text of the error message! I confess that so far I have not found the place in PyDev where it could be checked. When we do not know what is wrong, it is difficult to find for the cause…

In any case, if you want to end up broken script - use the *Resume* command (F8). Then you will see the error message (Figure 4.1.14):

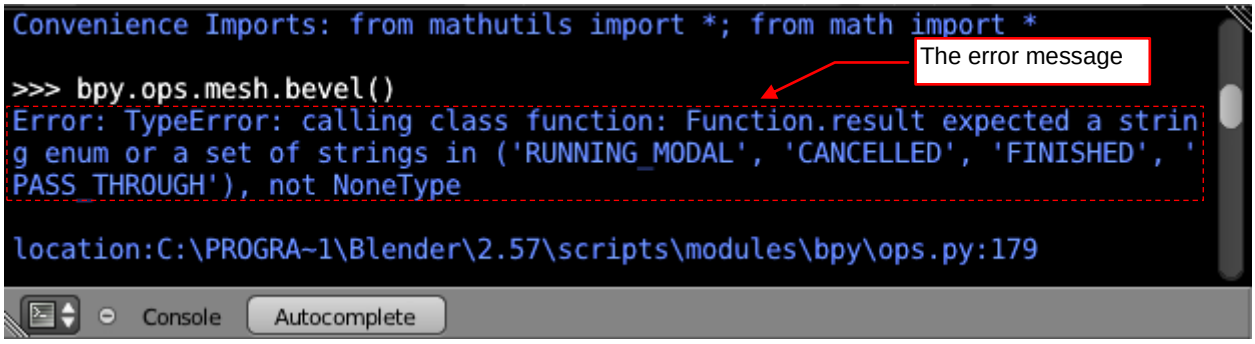


Figure 4.1.14 The full information about the runtime exception

Well, it's water over the dam. Now that we know what went wrong, we would like to examine the state of script variables. Unfortunately, it is impossible at this moment, because the code execution already has been terminated (see the stack shown in Figure 4.1.15). In practice, when an error occurs in the script for the first time, let it terminate, to be able to see its full description. Using it, set a breakpoint on the line, at which it occurs. Then run the script again to break its execution at this line. This time you will be able to analyze the script state, and to come to the cause of the problem.

- 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 — it will appear in the Blender *System Console* (see page 127, Figure 6.3.8).

In this particular case, such a complex analysis is not necessary. Blender has written clearly that it expects to receive from a function the strictly defined value (it may mean the return value of the ***Bevel.execute()*** method). Indeed, in a hurry while writing this code I have forgotten completely that the ***execute()*** function must return one of the enumeration values, listed in this message. Usually it returns **'FINISHED'**. Let's fix our script right away (Figure 4.1.15):

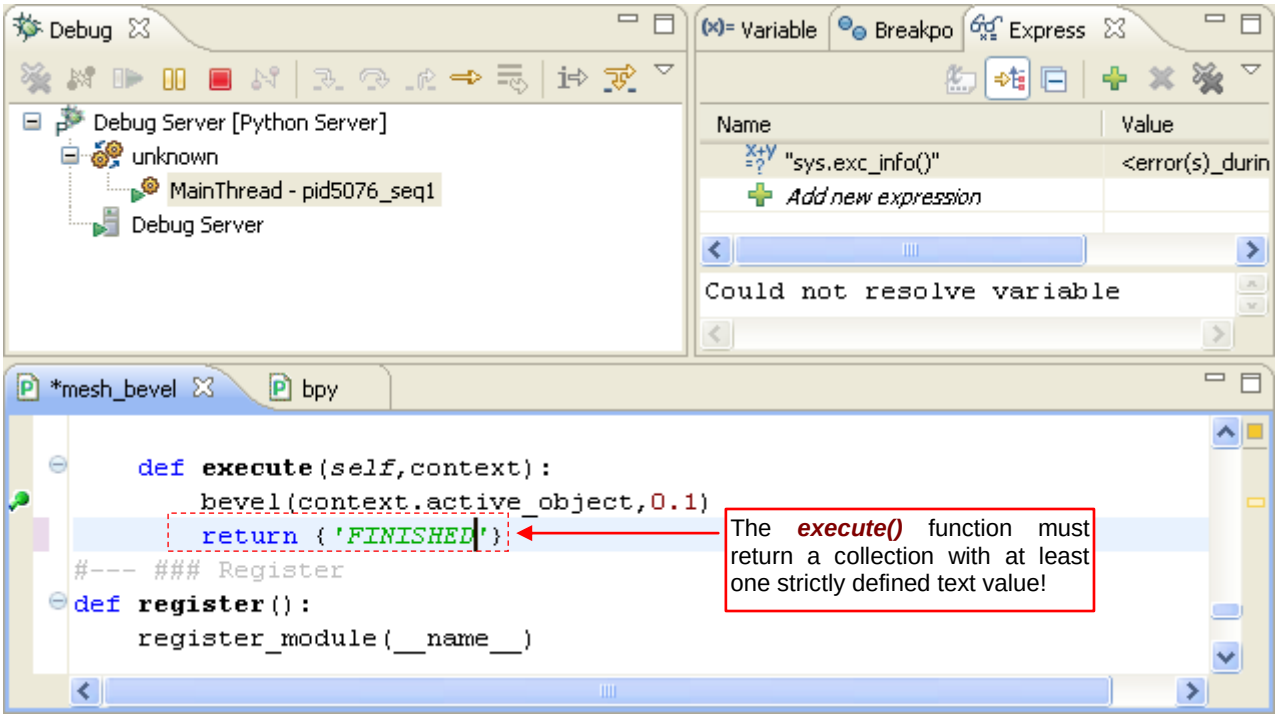


Figure 4.1.15 A quick fix of the code — directly in the *Debug* perspective

Just save the modified script on the disk. Then press the ***Run Script*** button, to reload the add-on code in Blender. Finally, invoke again this operator from the *Python Console* (Figure 4.1.16):

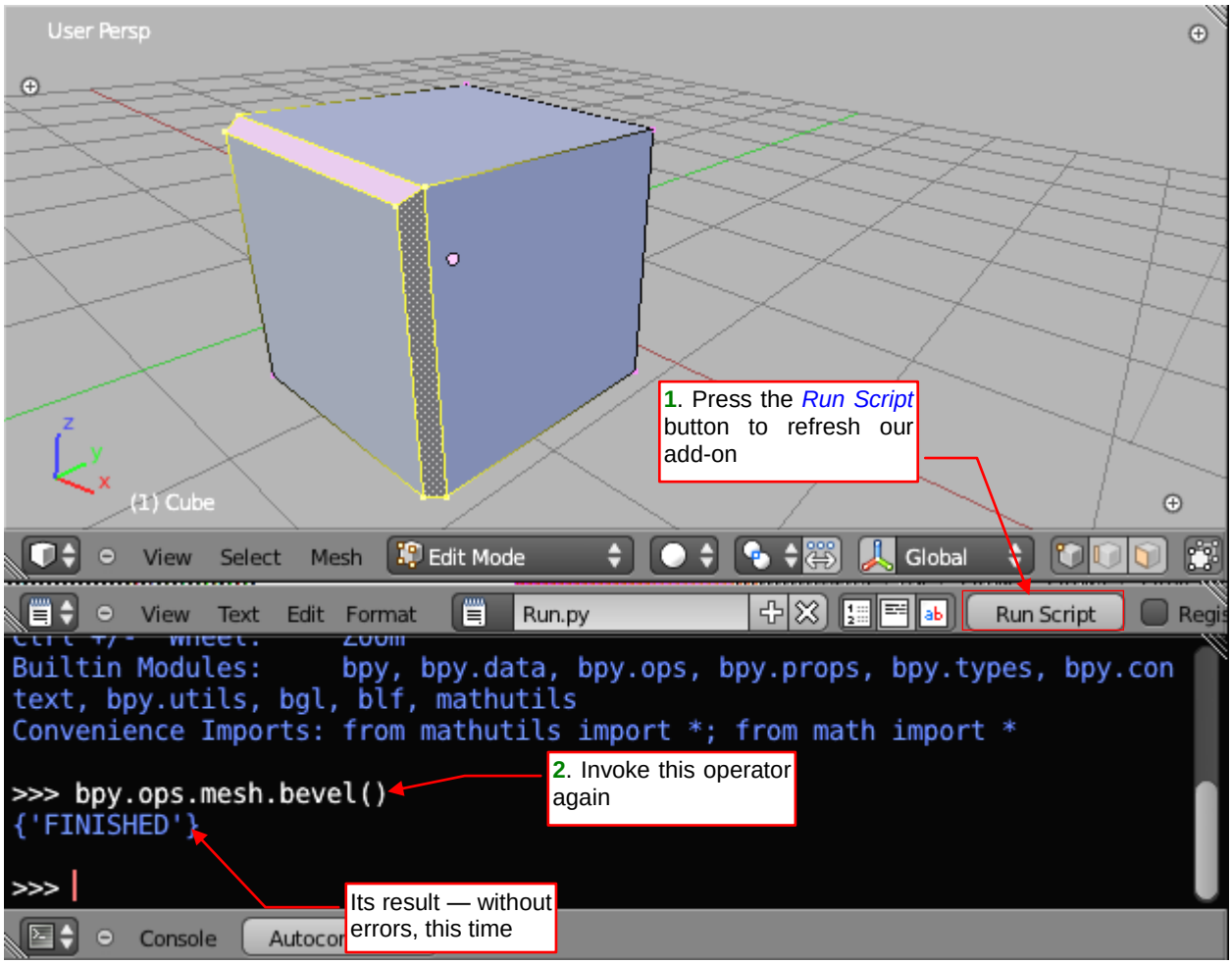


Figure 4.1.16 Another test of the fixed script

As you can see, after this correction our operator works properly. Now it can be added to the *Specials* menu (see also page 34). I will show how to do it in the next section.

Summary

- Each add-on must contain the ***bl_info*** structure (page 75). This is its „nameplate”, used by Blender to display the information in the *User Preferences:Add-Ons* tab;
- A procedure that changes something in the Blender data (like our ***bevel()***) must be converted into the Blender operator. It involves creation of a class that derives from ***bpy.types.Operator***. Place the call to the updating procedure inside the ***execute()*** method of this new class (page 75);
- Each add-on must implement the ***register()*** and ***unregister()*** script methods (page 76);
- The ***Run Script*** button reloads the current version of the add-on script, only. (It calls the ***unregister()*** method for the old version, and the ***register()*** method of the new one — see pages 82, 129);
- When you run the add-on script, it will just register its presence in Blender (page 77). You still have to invoke its operator — for example, using the *Python Console* (pages 77 - 78). In response to this call, Blender creates a new instance of the operator class, an invokes its ***execute()*** method;
- The information about the environment of this call — current selection, active object, etc. — is passed to the ***execute()*** function in the ***context*** argument (page 78);
- In case of script runtime error (when a runtime exception has been thrown), PyDev debugger breaks the execution (page 80). You can examine the state of the variables, at this moment. Unfortunately, I had never found a place where you would have seen the error message. This text will be displayed in the console when you let the script terminate (using the *Resume* command — page 81);

4.2 Adding the operator command to a Blender menu

Before we will add an operator to the menu, it is the final time to take care about the input data validation. To work properly, the ***bevel ()*** procedure requires two conditions:

1. the mesh is in the *Edit Mode*;
2. at least one of its edges is selected;

Let's begin the implementation of the first condition. In fact, we are going to add our operator to the mesh *Specials* menu, which is only available in the *Edit Mode*. Yet you never know whether someone in the future will add your operator to another menu or panel. Therefore the ***poll()*** method is always worth adding to your operator class (Figure 4.2.1):

```
#--- ### Operator
class Bevel(bpy.types.Operator):
    ''' Bevels selected edges of the mesh'''
    bl_idname = "mesh.bevel"
    bl_label = "Bevel"
    bl_description = "Bevels selected edges"

    #--- Blender interface methods
    @classmethod
    def poll(cls,context):
        return (context.mode == 'EDIT_MESH')

    def execute(self,context):
        bevel(context.active_object,0.1)
        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 have to declare ***poll()*** as the class method (not the typical instance method)

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

Figure 4.2.1 The basic „availability test" — implementation of the *poll()* function

Blender invokes this function to find out if "in the current situation" this command is available. The "current situation" is described by the ***context*** argument. It is an instance of the ***bpy.types.Context*** class (we have already met this class — see pages 78, 56, 54). The ***poll()*** function may examine the ***context*** object and returns *True*, when the operator is available. Otherwise, it returns *False*.
This is the place for „general" tests, such as the condition #1. In our implementation the ***poll()*** function returns *True* when the mesh is in the *Edit Mode*. (This is the meaning of the **'EDIT_MESH'** mode value. If we were in other edit mode — the armature, for example — the ***context.mode*** field would return a different value).

- 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 it here will cause a script runtime error.

Notice the *@classmethod* expression before the header of the ***poll()*** function. (In the programmer's jargon, it is called a "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.

By the way, have you noticed that the Blender API requires from your operator class to implement strictly defined methods? It is a kind of a "contract" between your script and the Blender core system. You agree to prepare a class with specific functions. Blender agrees to call them in the strictly defined circumstances.

[1] Probably it improves the performance of the Blender environment. The ***poll()*** methods are implemented by all GUI controls, and they are called every time the Blender screen is refreshed. (The ***poll()*** functions of appropriate controls are called when the user do anything — pulls down a menu, clicks a button, etc.). If ***poll()*** were an instance method, like ***execute()***, Blender would every time create the instances of control objects just to call their ***poll()*** methods, and then discard them immediately. I suppose that it would work more slowly, perhaps too slowly. To call the class method you not need to create its instance (an object), and therefore this operation requires less CPU time.

Such a list of contracted functions and properties is called "interface" in the object-oriented programming. To help you a little in its implementation, Blender API delivers the base for your operator: the ***bpy.types.Operator***[1] class. In the object-oriented programming jargon, ***Operator*** is the so-called "abstract class". It just provides the default, empty implementations of all methods required by the operator interface. Our operator class (***Bevel***) inherits this default content from its base (***bpy.types.Operator***). That's why it is possible to implement (override, in fact) in the ***Bevel*** just these ***Operator*** methods, which are specific for this derived class.

We will not check the condition #2 („at least one of the mesh edges is selected") in the ***poll()*** method. It is too specific. It would be a very strange command, available only when something was selected on the mesh! Half of the users would have no luck to see it in this state, and concluded that this add-on does not work. It is better to make the Bevel command available in the Specials menu all the time. If the user invokes it without marking any mesh edge before, it will display an appropriate message. In this way, she/he will know how to use it next time.

We could add such "advanced validation" to the ***Operator.execute()*** method. However, in certain situations, this method may be called repeatedly, for the same context and with different other input parameters. (You'll find this in the next section). Therefore, it is not good place for such a check, and certainly not to display the messages for the user. There is a better place, in another method of this interface: ***Operator***: ***invoke()*** (Figure 4.2.2):

```
#--- ### Operator
class Bevel(bpy.types.Operator):
    ''' Bevels selected edges of the mesh'''
    bl_idname = "mesh.bevel"
    bl_label = "Bevel"
    bl_description = "Bevels selected edges"
    #--- Blender interface methods
    @classmethod
    def poll(cls,context):
        return (context.mode == 'EDIT_MESH')

    def invoke(self, context, event):
        #input validation: are there any edges selected?
        selected = 0 #edge count
        for edge in context.object.data.edges:
            if edge.select:
                selected += 1

        if selected > 0:
            return self.execute(context)
        else:
            self.report(type = 'ERROR', message = "No edges selected")
            return {'CANCELLED'}

    def execute(self,context):
        bevel(context.object,0.1)
        return {'FINISHED'}
```

Blender calls the ***invoke()*** method, when operator has been invoked from a menu or a button.

It may also call the ***execute()*** method, later (when user will change an operator parameter in the *Tool Properties* pane)

We will not use the ***event*** in our script. However, it is useful for so-called "modal operators".

Counting the selected edges of the mesh

If nothing is selected — let the user to know about it!

I noticed, that Blender interface code uses ***context.object*** instead of ***active_object***. I will adhere to this convention, as a precaution.

Figure 4.2.2 Additional validation of the input data — in the *invoke()* method

Blender expects that the ***invoke()*** function will return similar codes like the ***execute()*** method. Our implementation of ***invoke()*** begins with the counting of the mesh selected edges. If there is none, it displays the warning message and returns the **'CANCELLED'** code. Otherwise, it calls the ***execute()*** method and returns its result (**'FINISHED'**).

[1] In addition to the ***Operator*** interface, the Blender API contains two other interfaces (abstract classes): ***Menu*** and ***Panel***. Obviously, they serve to implement the user interface controls. You can find all of them in the ***bpy.types*** module, as well as in the PyDev autocompletion suggestions. I wish the descriptions of these interfaces in the Blender documentation were better. Many of the details, which I am describing here, are based on the various examples and my own observations, only!

The ***invoke()*** method receives, except the ***context*** argument, another object: ***event***. This is the 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). To check the ***event*** fields in PyDev debugger, always use the *Expressions* window. Type there names of particular ***bpy.types.Event*** fields, for example „event.type", or „event.value". Any attempt to expand the fields of the ***event*** object in the *Variables* pane generates a Windows fault and terminates the Blender process!

I would like to draw your attention for a moment on the loops in Python. Writing the code shown in Figure 4.2.2 (page 85), I have tried to implement the loop that counts the selected edges in the most readable way. It was a piece of code in the „visual basic" programming style. Browsing the Python code examples on the Internet, you might encounter the other, "single line" solutions for such an operation (Figure 4.2.3):

```
    def invoke(self, context, event):
        #input validation: are there any edges selected?
        selected = list(filter(lambda e: e.select, context.object.data.edges))

        if len(selected) > 0:
            return self.execute(context)
        else:
            self.report(type = 'ERROR', message = "No edges selected")
            return {'CANCELLED'}
```

selected — the list of the selected mesh edges

Figure 4.2.3 Alternative way to count the selected edges in the *invoke()* procedure

This is an expression in the special „python" (or maybe even the „lisp") style. The ***filter()*** function returns so called *iterator*, which is converted by the ***list()*** function into a collection (list). Then you can check in the conditional expression the length of this list. In the ***filter()*** function I have used the unnamed, temporary *lambda function*. This *lambda* receives from the filter a single argument (***e***) — the element of the input collection. *Lambda function* returns the value of its last expression (here: the sole expression) — that is *True*, when the ***e*** edge is selected. (The detailed description of the standard ***filter()*** function you can find in the Python documentation). The code readability depends on the advancement of the reader. For the experienced Python programmer the ***filter()*** expression with *lambda function* is as much readable, as the loop shown in Figure 4.2.2.

All right, our enhanced operator is ready to use. Yet how to add it to the Blender ***Specials*** menu (Figure 4.2.4)?

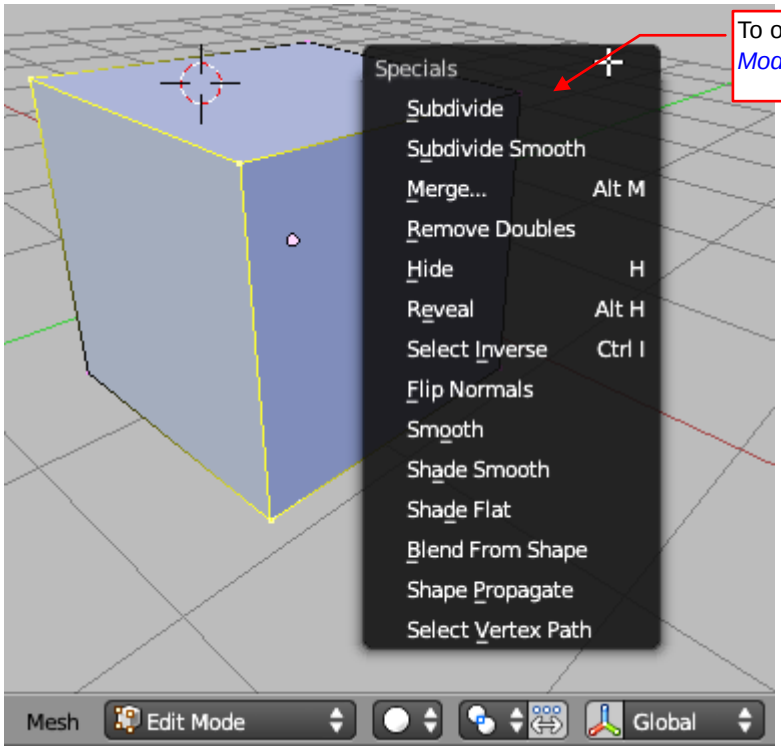


Figure 4.2.4 The *Specials* menu (in the mesh *Edit Mode*)

All Blender menus are created in the same way that is available for your add-on: using the Python API. You just need to discover the name of the class that implements the *Specials* menu. Let's begin with finding the file that should contain its code. Scripts that implement the entire Blender user interface can be found in *scripts\startup\bl_ui* directory (Figure 4.2.5):

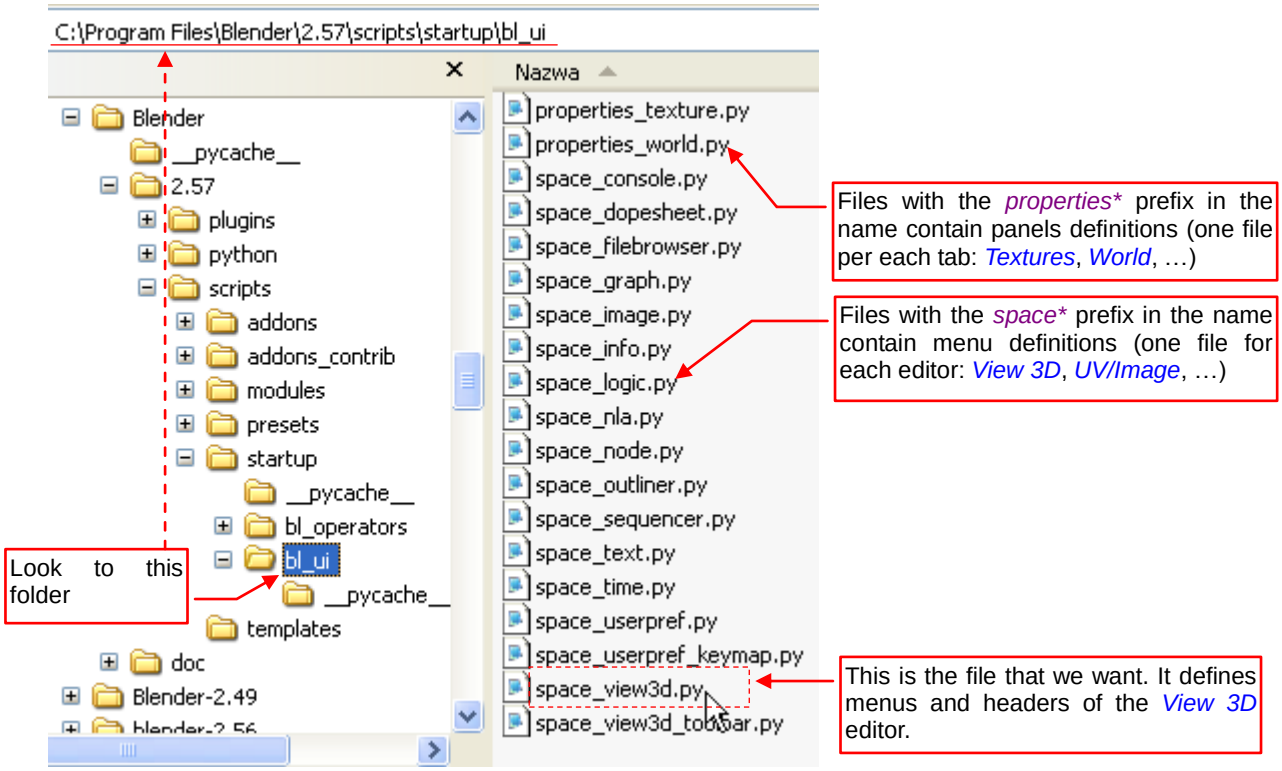


Figure 4.2.5 Searching for the file with the *View 3D* menu definitions

Files with names starting with *properties_** contain various panel classes for the *Properties* editor. Omit them, at this moment. There are also other files, which names have following structure: *space_<editor name>.py*. They contain definitions of the menus and headers for each Blender editor. The *Specials* menu belongs to the *View 3D* editor, so we should look for it inside the *space_view3d.py* file.

Open this file in your favorite “add hoc” text editor (it can be just standard Notepad, or popular Notepad++ — what you like). Search its content for the “human” name of this menu - the **"Specials"** text (Figure 4.2.6):

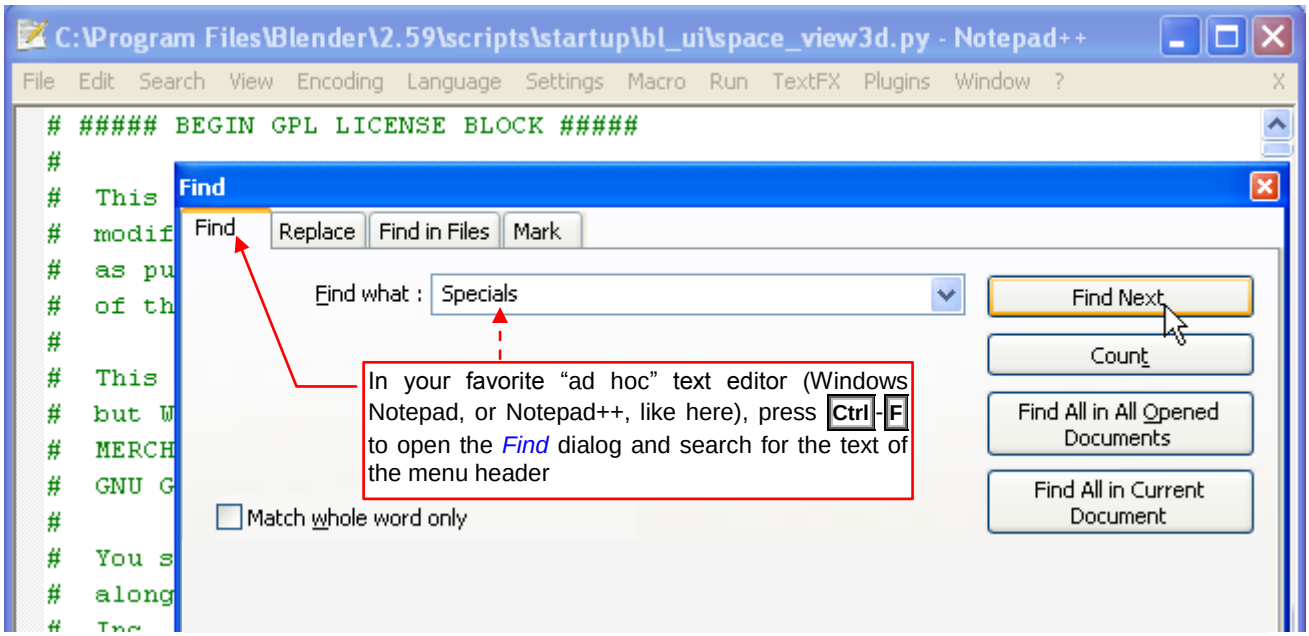


Figure 4.2.6 Searching for the “Specials” phrase in the *space_view3d.py* file

Just remember that the same text may appear in many different menus! So is also in this case. First, I found the menu class that implements *Specials* menu for the *Object Mode* (Figure 4.2.7):

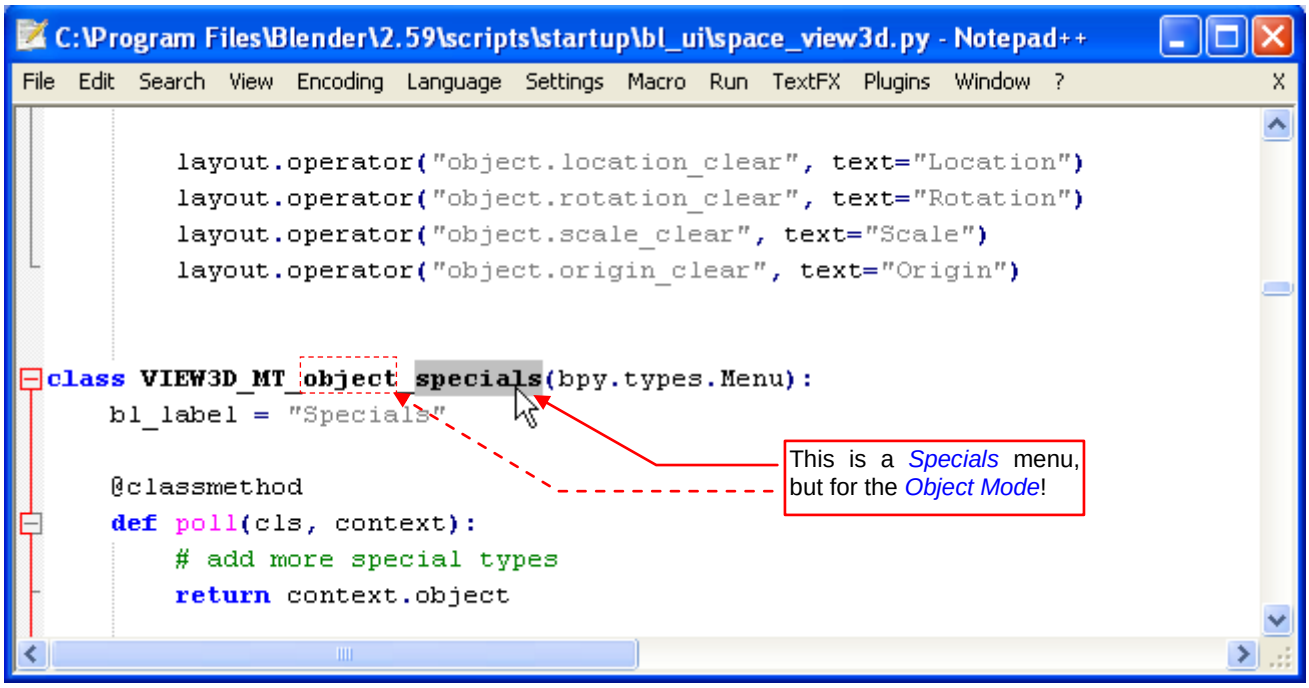


Figure 4.2.7 One of the incorrect hits: similar menu for the other mode

How did I know it was not the menu that I was looking for? Although it had the proper header (the value of the ***bl_label*** class field), it contained different items (the lines ***layout.operator(<operator name>, text = <display name>)***) than the menu shown in Figure 4.2.4!

After finding another definition of the *Specials* menu, I realized that part of their class name is the symbol of Blender mode, in which they are used: „object", „particle"… The third was the one I was looking for: „edit_mesh" (Figure 4.2.8):

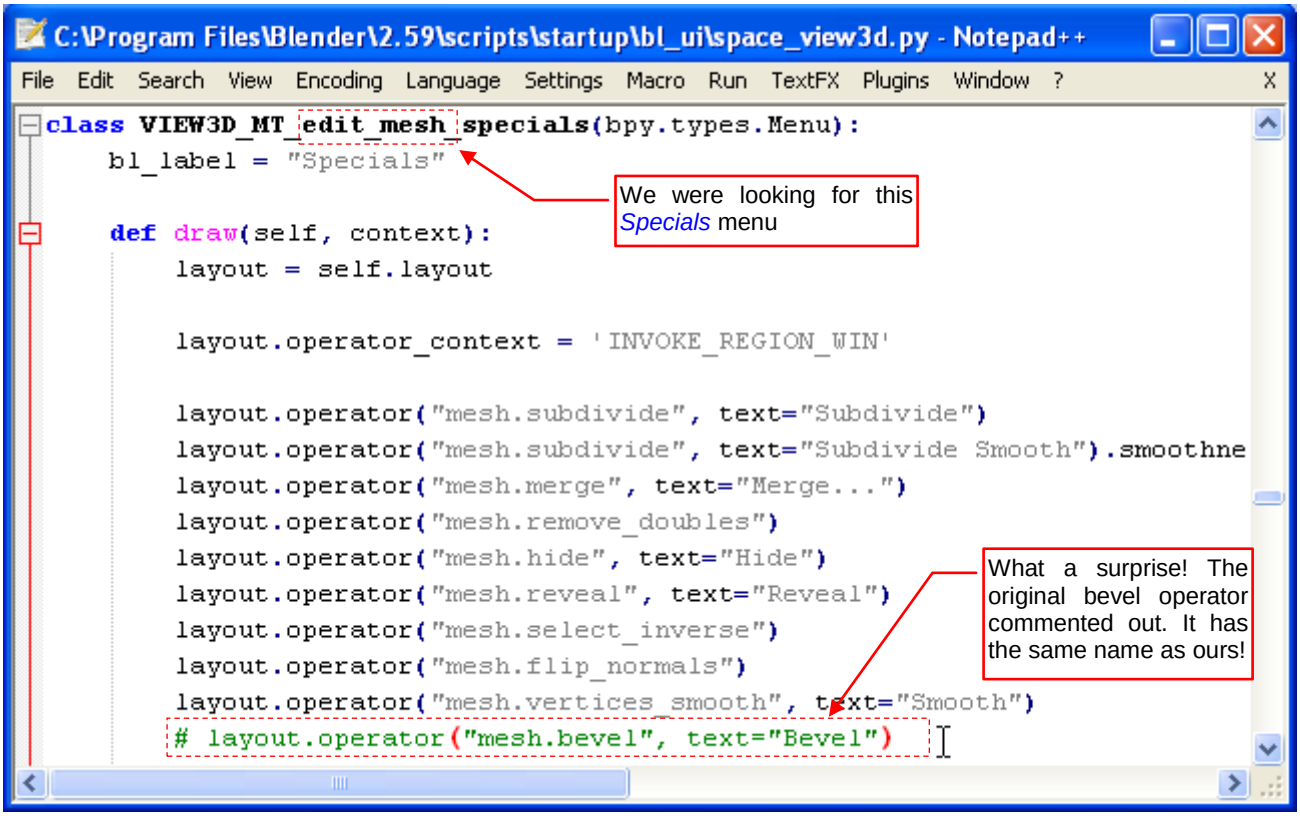


Figure 4.2.8 Class that implements the *Specials* menu for the *Edit Mode*

When I know the menu class name, I can write the code that will add our operator to this menu (Figure 4.2.9):

This is a helper function. Invokes the same expression that we have seen in the menu class code (see Figure 4.2.8)

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

#--- ### Register
def register():
    register_module(__name__)
    bpy.types.VIEW3D_MT_edit_mesh_specials.prepend(menu_draw)

def unregister():
    bpy.types.VIEW3D_MT_edit_mesh_specials.remove(menu_draw)
    unregister_module(__name__)
```

This line forces Blender to use the ***invoke()*** method of the operator, instead of ***execute()***

Adding and removing the menu command

Figure 4.2.9 Adding the operator command to the *Specials* menu

The first tests of the modified ***register()*** and ***uregister()*** methods are successful (Figure 4.2.10):

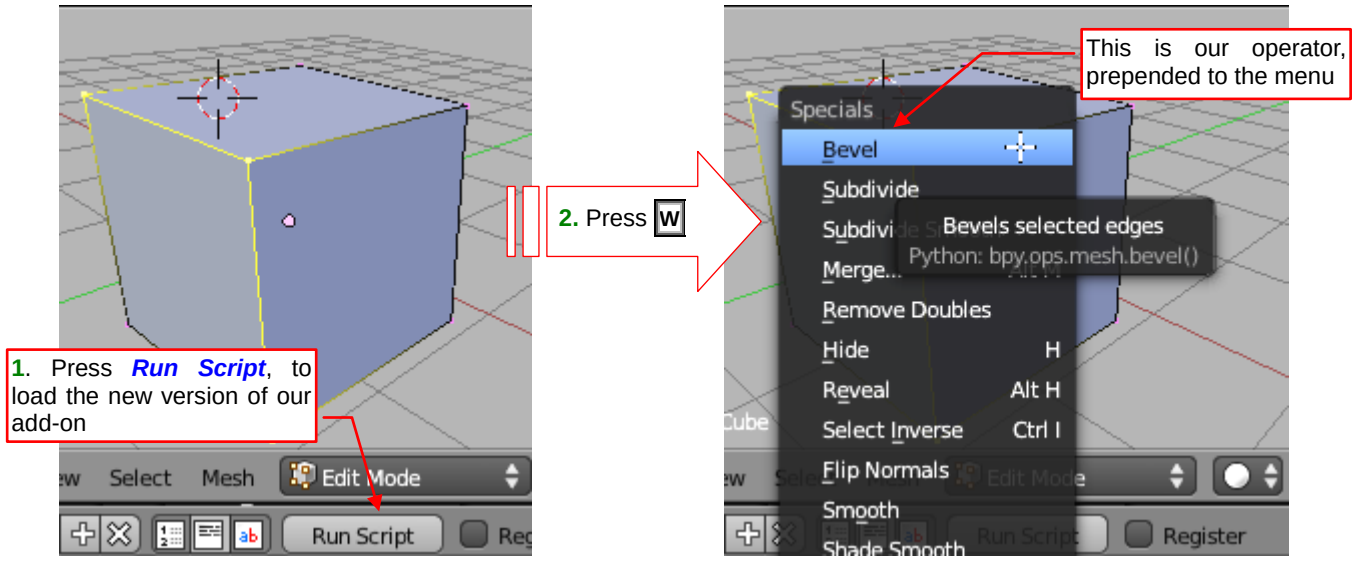


Figure 4.2.10 Checking the menu update

Another test — an “empty” invocation without any selected edges — gives the expected result (Figure 4.2.11):

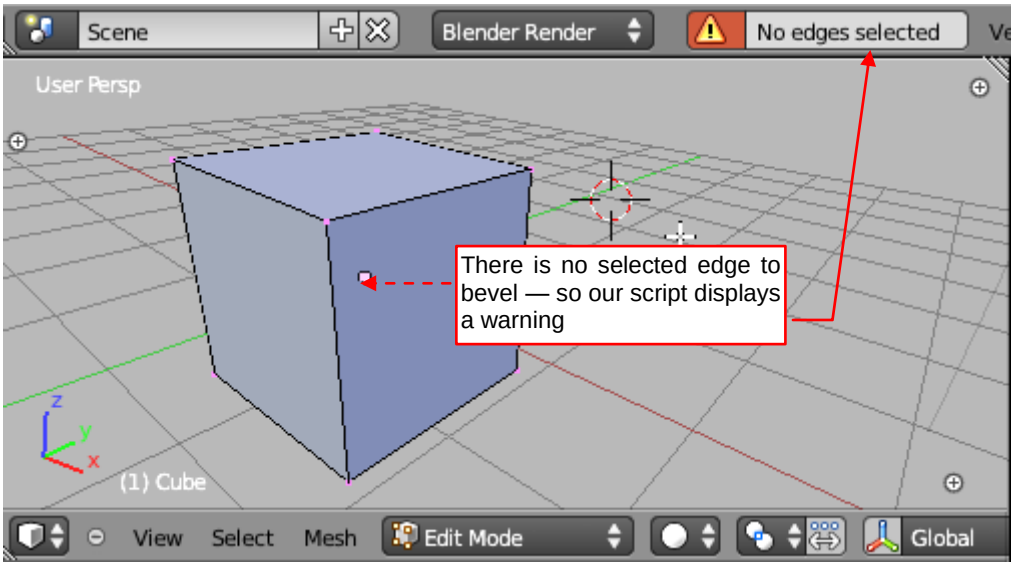


Figure 4.2.11 The result of invoking the *Bevel* command without any selected edges

However, when I selected some edges of the mesh and invoked again the *Bevel* command — I saw the same warning, again! (Figure 4.2.12):

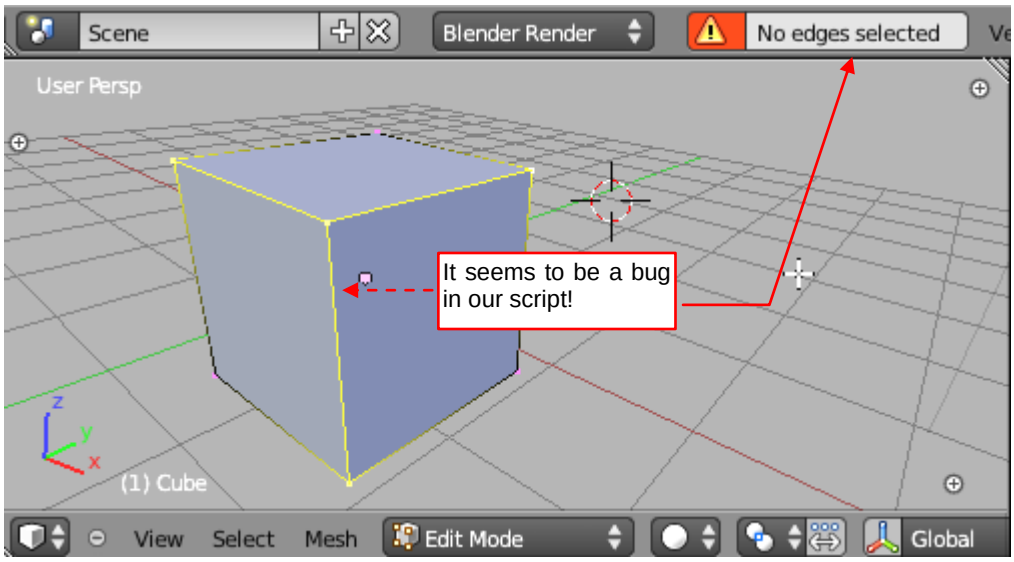


Figure 4.2.12 The result of invoking the *Bevel* command with some edges selected

I chattered so much on pages 84—86 that I made a stupid mistake. I forgot about the thing, which I described myself in the previous chapter (see pages 52, 53). Before you start counting the edges, switch Blender to the *Object Mode*, and when it is done - back to the *Edit Mode* (Figure 4.2.13):

```
    def invoke(self, context, event):
        #input validation: are there any edges selected?
        bpy.ops.object.editmode_toggle()
        selected = list(filter(lambda e: e.select, context.object.data.edges))
        bpy.ops.object.editmode_toggle()

        if len(selected) > 0:
            return self.execute(context)
        else:
            self.report(type='ERROR', message="No edges selected")
            return {'CANCELLED'}
```

Switch Blender into the *Object Mode* to count these edges properly

Figure 4.2.13 Fix in the program: any reference to the mesh data must be performed in the *Object Mode*!

After this fix, ***invoke()*** finds the selected edges, and the command works properly (Figure 4.2.14):

Figure 4.2.14 Successful test of the fixed script

We have already achieved the effect similar to the Bevel command from Blender 2.49 (see page 34). Our *Bevel* command lacks only the interactive ("dynamic") width changing. We will add this functionality in the next section.

Summary

- 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 of the current mode (page 84);
- The further, more detailed verification of input data should be implemented in another method: ***invoke()*** (page 85). Blender calls this method when user invokes your operator from a menu (or presses a button). This is controlled by appropriate field of the ***bpy.types.Menu*** class (see page 89). The same applies to the panels. (I have not described panels here — the custom controls that derive from the ***bpy.types.Panel*** class);
- Add your operator to a Blender menu in the ***register()*** method, and remove in ***unregister()*** (page 89). To write this fragment of code, we have to know the Python class name, that implements this particular Blender menu;
- You can find the class name of a Blender menu in the Python script files that define the Blender user interface (pages 87 - 88);

4.3 Implementation of dynamic interaction with the user

In Blender 2.5, it is very simple to implement a dynamic interaction between your operator code and the user — its certain scheme, at least. It allows the user to change continuously the operator parameters (using mouse, for example), while Blender is updating the result on the screen.

First, add to the operator class the ***width*** parameter (as a class field). Create it, using appropriate function form the ***bpy.props*** module (Figure 4.3.1):

```
#--- ### Imports
import bpy
from bpy.utils import register_module, unregister_module
from bpy.props import FloatProperty

                    ...
        The remaining code of the script
                    ...
#--- ### Operator
class Bevel(bpy.types.Operator):
    ''' Bevels selected edges of the mesh'''
    bl_idname = "mesh.bevel"
    bl_label = "Bevel"
    bl_description = "Bevels selected edges"
    bl_options = {'REGISTER', 'UNDO'} #Set this options, if you want to update
    #                                  parameters of this operator interactively
    #                                  (in the Tools pane)
    #--- parameters
    width = FloatProperty(name="Width", description="Bevel width",
                          subtype = 'DISTANCE', default = 0.1, min = 0.0,
                                                step = 1, precision = 2)

    #--- Blender interface methods
    @classmethod
    def poll(cls,context):
        return (context.mode == 'EDIT_MESH')

    def invoke(self, context, event):
        #input validation: are there any edges selected?
        bpy.ops.object.editmode_toggle()
        selected = list(filter(lambda e: e.select, context.object.data.edges))
        bpy.ops.object.editmode_toggle()

        if len(selected) > 0:
            return self.execute(context)
        else:
            self.report(type='ERROR', message="No edges selected")
            return {'CANCELLED'}

    def execute(self,context):
        bevel(context.object,self.width)
        return {'FINISHED'}
```

Import the class for the attribute of **Float** type.

Currently, there is still no description of these options in the official Blender API documentation. This combination I have copied from the code of *Twisted Torus* add-on, or something like that.

Creation of the *width* operator parameter

Using the *width* parameter value

Figure 4.3.1 Changes in the class definition

The field created in this way, Blender will display as a control on the screen. The ***bpy.props*** module contains classes to define the parameters (attributes) of four basic types: ***Bool****, ***Float****, ***Int****, ***String****. Additionally, there are also one-dimensional arrays (the ****Vector**** classes) of each of these types. The bevel width is a *Float* value, in our script. That's why I import from ***bpy.props*** just a single class — ***FloatProperty()***. In its constructor, you can set up all the properties of a GUI control: the label (***name***), tooltip ***description***, default value, and the range. The ***step*** parameter determines the value of increment/decrement, used when the user will click the arrows on the control ends. The ***precision*** parameter determines the number of digits displayed in the control text area, after the decimal dot.

After adding the parameters (properties) to the operator class, you should add to it a field called ***bl_options***. (We have not used it, so far. It is an optional element of the operator interface). Assign to it a list of two values: {**'REGISTER'**, **'UNDO'**} (Figure 4.3.1). You have to use exactly these values. If you assign it single value of **'REGISTER'**, or **'UNDO'**, you will not obtain the effect, which is shown in Figure 4.3.2:

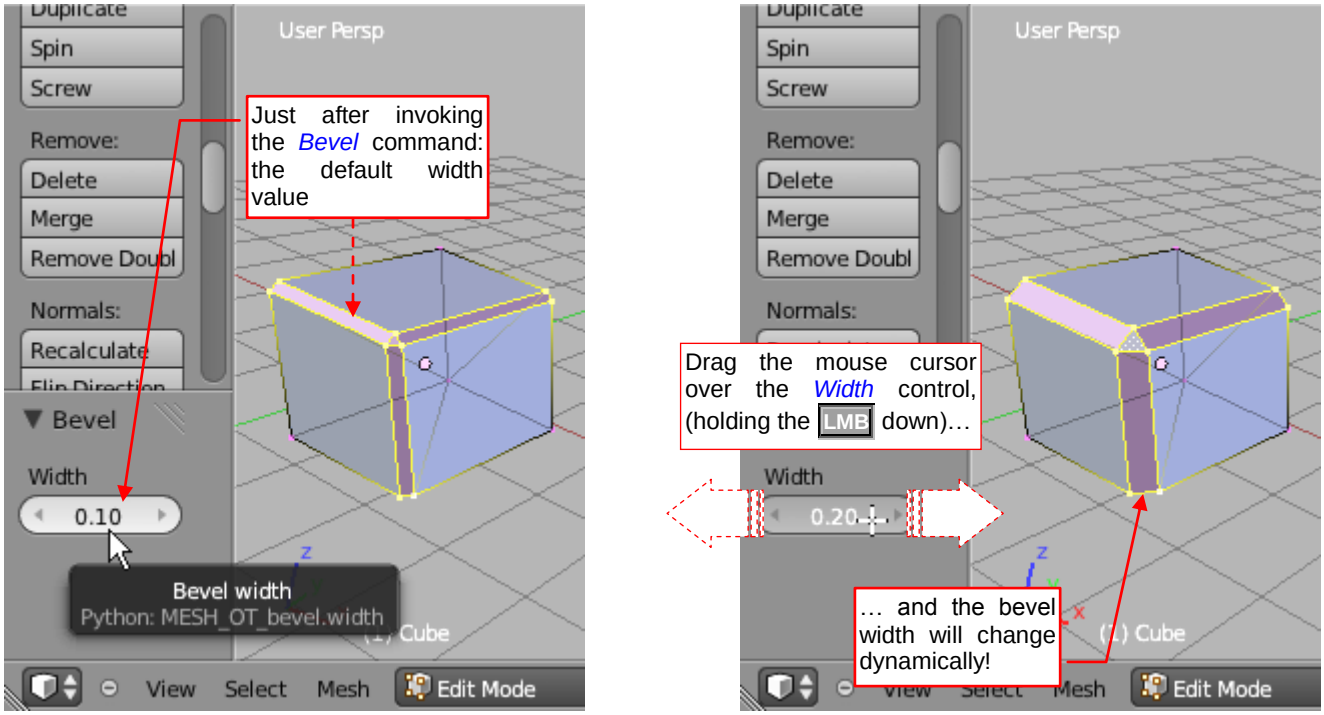


Figure 4.3.2 Dynamic change of the bevel width

Invoke our command (*Specials→Bevel*). It bevels the selected mesh edges as previously — using the default width. Now press the T key, to open the *Tool* shelf (on the left side of the screen). In the *Tool properties* area you can see a panel having the same name, as our operator (*Bevel*). Such panel contains the controls with the parameters of the last used command — in our case it is the bevel *Width*. When you change its value here — Blender will update immediately the result on the screen. When you drag the mouse cursor (holding the LMB down) over this control, the bevel width will change dynamically, like in Blender 2.49 (see page 34, Figure 3.1.2).

How does Blender get this effect from our srcript? To track down such interactive events, a simple printing of a diagnostic text in the console is better suited than the debugger. Put for a moment apropriate ***print()*** statements in both operator methods: ***invoke()*** and ***execute()*** (Figure 4.3.3):

```
def invoke(self, context, event):
    #input validation: are there any edges selected?
    print("in invoke()")
    bpy.ops.object.editmode_toggle()
    selected = list(filter(lambda e: e.select, context.object.data.edges))
    bpy.ops.object.editmode_toggle()

    if len(selected) > 0:
        return self.execute(context)
    else:
        self.report(type='ERROR', message="No edges selected")
        return {'CANCELLED'}

def execute(self,context):
    print("in execute(), width = %1.2f" % self.width)
    bevel(context.object,self.width)
    return {'FINISHED'}
```

Diagnostic messages (for the Blender *System Console*)

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

Reload this new add-on version, and invoke again the *Specials→Bevel* command (Figure 4.3.4):

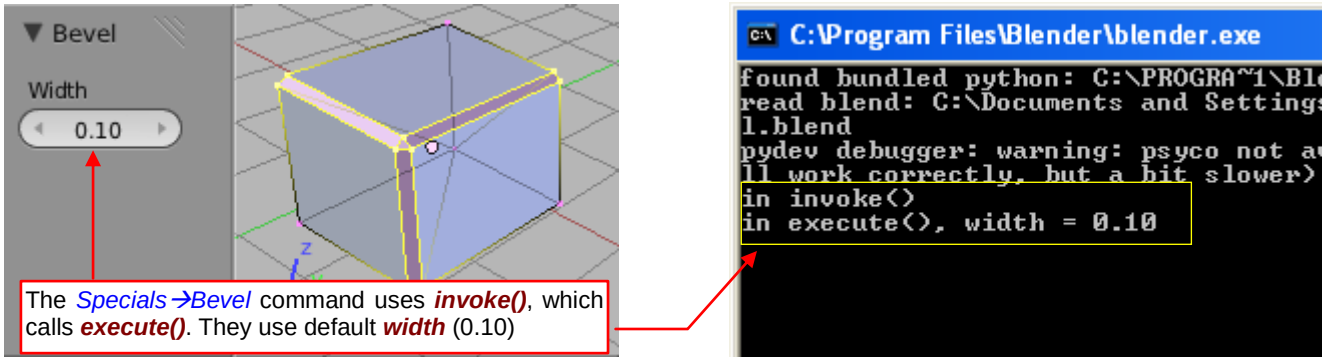


Figure 4.3.4 The state immediately after the *Specials→Bevel* command

Immediately after this invocation, two messages have appeared in the console (Figure 4.3.4). It seems that Blender has called the ***invoke()*** method, which in turn (see Figure 4.3.1) has called ***execute()*** with the default value of the ***width*** parameter.

Now let's change the value of *Bevel:Width* field in the *Tool Properties* pane. I have pressed ten times the "arrow" on its right side, increasing the bevel width from 0.1 to 0.2 (Figure 4.3.5):

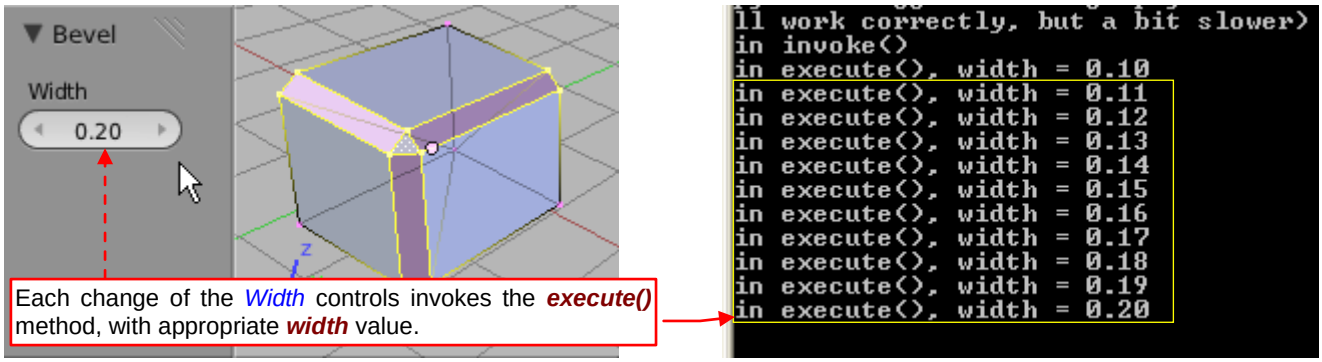


Figure 4.3.5 The state after increasing the *Width* value to 0.2 (in 10 steps)

Do you see? It seems that every time I have changed the value of the control, Blender has called *Undo* command, and then simply has invoked the operator again. It uses directly its ***execute()*** method, calling it with the ***width*** parameter set to the current value read from the *Width* control.

I suppose that Blender every time just invokes the operator method: ***bpy.ops.mesh.bevel(width = <current control value>)***. Since you added the ***width*** parameter to the ***Bevel*** class, its method received an optional argument with the same name (Figure 4.3.6):

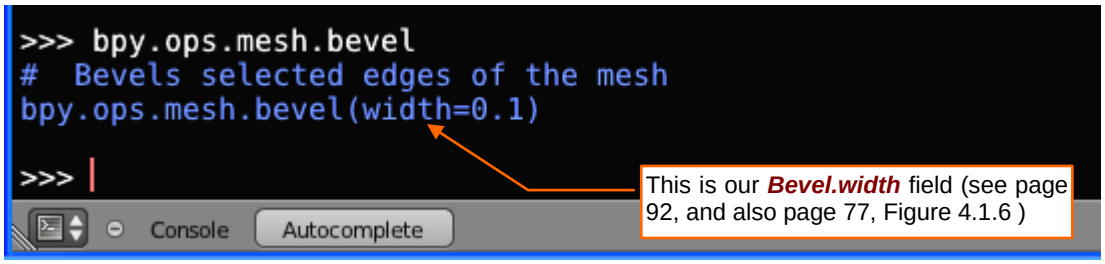


Figure 4.3.6 The named argument of the *bpy.ops.mesh.bevel()* method

I think that the roles of the ***invoke()*** and ***execute()*** procedures 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 a 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 89).

There is still one detail that has to be added to this add-on. Our operator should "remember" the last bevel width. It will be used as the default value on the next use. This will greatly help the user.

How to implement it? Do not try to store anything in the ***Bevel*** object. It seems that Blender creates a new instance of this class on each response to the *Bevel* command. That is why our operator object appears on each call in its default state. The proper place to store such information between the subsequent calls is the Blender data file. Most elements of the scene can be used as if they were Python dictionaries (Figure 4.3.7):

```
context.object["our data"] = 1.0 #Store
stored_data = context.object["our data"] #Retrieve
```

Figure 4.3.7 Simplest example of storing information in the Blender data file

Be careful on retrieving such values. Usually you can not be sure if your data was previously placed there. It is better to read them using the standard ***get()*** method, which does not generate an exception when the desired element is missing. Here's modification, which allows our script to use the last bevel width as the default value on the next invocation (Figure 4.3.8):

```
#--- ### Operator
class Bevel(bpy.types.Operator):
    ''' Bevels selected edges of the mesh'''
    bl_idname = "mesh.bevel"
    bl_label = "Bevel"
    bl_description = "Bevels selected edges"
    bl_options = {'REGISTER', 'UNDO'} #Set this options, if you want to update
    #                                  parameters of this operator interactively
    #                                  (in the Tools pane)
    #--- parameters
    width = FloatProperty(name="Width", description="Bevel width",
                          subtype = 'DISTANCE', default = 0.1, min = 0.0,
                                                step = 1, precision = 2)
    #--- other fields
    LAST_WIDTH_NAME = "mesh.bevel.last_width" #name of the custom scene property

    #--- Blender interface methods
    @classmethod
    def poll(cls,context):
        return (context.mode == 'EDIT_MESH')

    def invoke(self, context, event):
        #input validation: are there any edges selected?
        bpy.ops.object.editmode_toggle()
        selected = list(filter(lambda e: e.select, context.object.data.edges))
        bpy.ops.object.editmode_toggle()

        if len(selected) > 0:
            last_width = context.scene.get(self.LAST_WIDTH_NAME,None)
            if last_width:
                self.width = last_width
            return self.execute(context)
        else:
            self.report(type='ERROR', message="No edges selected")
            return {'CANCELLED'}

    def execute(self,context):
        bevel(context.object,self.width)
        context.scene[self.LAST_WIDTH_NAME] = self.width
        return {'FINISHED'}
```

Just for the code clarity: the name of the dictionary item

An attempt to read the value (it may not exist, yet!)

If the last bevel width was stored: use it now!

Store the last used bevel width

Figure 4.3.8 Implementation of storing the last used width

Notice the script stores the last used bevel width in the current scene (***context.scene***), not in the modified object or its mesh (Figure 4.3.8). If it placed the width in the current object, then you would obtain different default values for various objects. I think it would be very confusing for the user. Thus, I prefer to store one width for all calls — and the best place to keep such single value is the current scene. Preserving the current values of the operator parameters in the Blender data has also another advantage that they are permanently stored when the user saves the file.

There is yet another problem with such data, which may occur later. The same dictionary keys may be used in the same scene by two different add-ons. In the result, one of them will overwrite the parameters of the other one, and probably first of these scripts will end up with an error. Therefore, you should use the most specific, long dictionary key names.

The storing of the last used width value was the final touch to this add-on. It is impossible to show this new functionality on the pictures, so I exceptionally skipped them (☺). Our *mesh_bevel.py* plugin is ready to use. When you put this file among the other Blender plugins (in the *scripts\addons* directory), our *Bevel* will appear in the *User Properties* window (Figure 4.3.9):

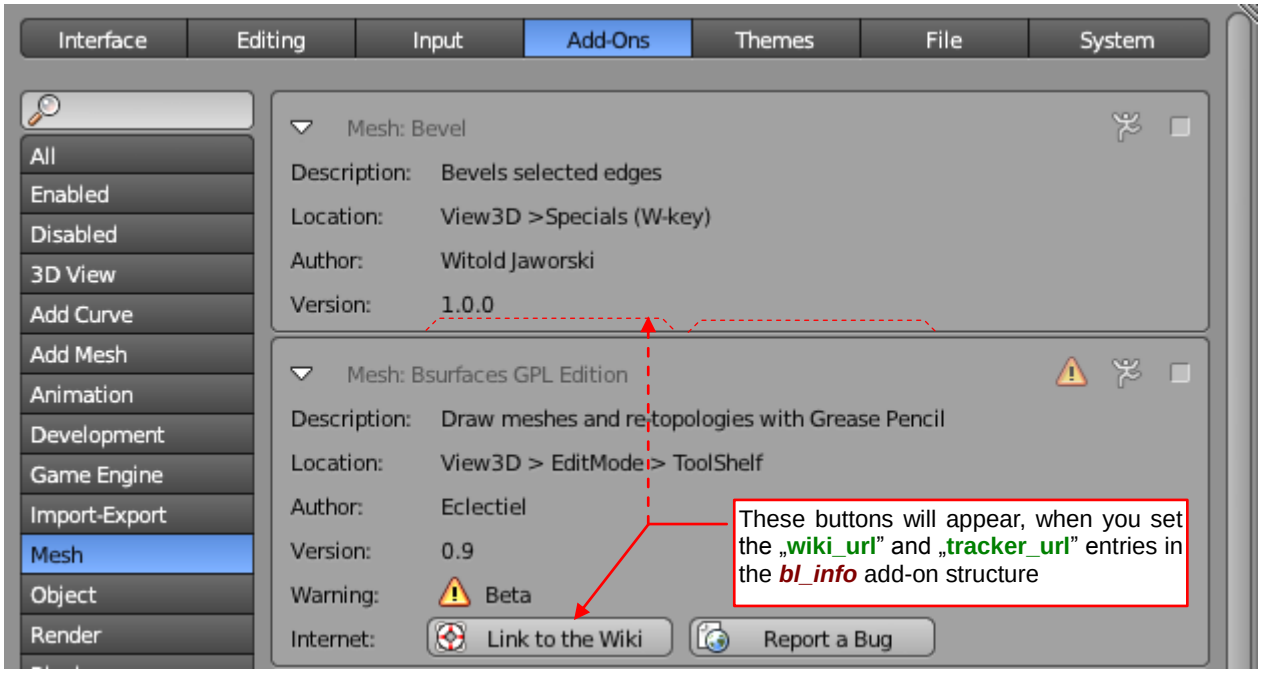


Figure 4.3.9 Our *Bevel* add-on, displayed in the *Add-Ons* tab

You still have to publish the description of this add-on in the *wiki.blender.org*, and to open a bug tracker for the eventual error notifications[1]. However, it is no longer the subject of this book. The full code of the script, we have written here, you will find on page 131.

[1] In the result of such user feedback, I added further modifications to this script. One of them is the dynamic adaptation of the bevel width. The last used value, as implemented in this book, did not fit well for large differences in the size of subsequent objects. To resolve this problem, I added to the ***invoke()*** procedure a code that estimates the object size. On this basis, the program decides whether to ignore the last used bevel value or to use the dynamically calculated default width. Such updates are natural to the development of each program. I think that the implementation of this additional functionality would complicate our script, obscuring the main ideas presented in this book.
If you want to analyze the full code of the current "real" version of this *mesh_bevel.py* add-on, you can get it from the http://airplanes3d.net/scripts-253_p.xml page.

Summary

- Create the operator parameters (*properties*) as a class fields, using appropriate function from the ***bpy.props*** module (page 92). The fields, created in this way, become automatically the named arguments of the operator method (from the ***bpy.ops*** namespace — see page 94);
- To make your command interactive, just add to its operator class following line: ***bl_options*** = {**'REGISTER'**,**'UNDO'**}. When you invoke it, you will see in the *Tool Properties* sidebar a panel with all command parameters, presented as the GUI controls. You can change them there using the keyboard or the mouse. The results of these changes are dynamically updated on the screen, in the *View 3D* editor (page 94);
- Save the current parameter values in the current scene. You can use them as the defaults on the next invocation of your command (page 95).

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 the details of the Python, Eclipse and PyDev installation procedures. Study them just in the case you stuck on some trifle.

5.1 Details of Python installation

The installation of the external Python interpreter has not changed for many years, so let me show it using the version 2.5.2, for which I already have prepared the illustrations.

Enter on the project page: www.python.org (Figure 5.1.1):

Figure 5.1.1 Main page of the Python project

Go to the *DOWNLOAD* section (Figure 5.1.2):

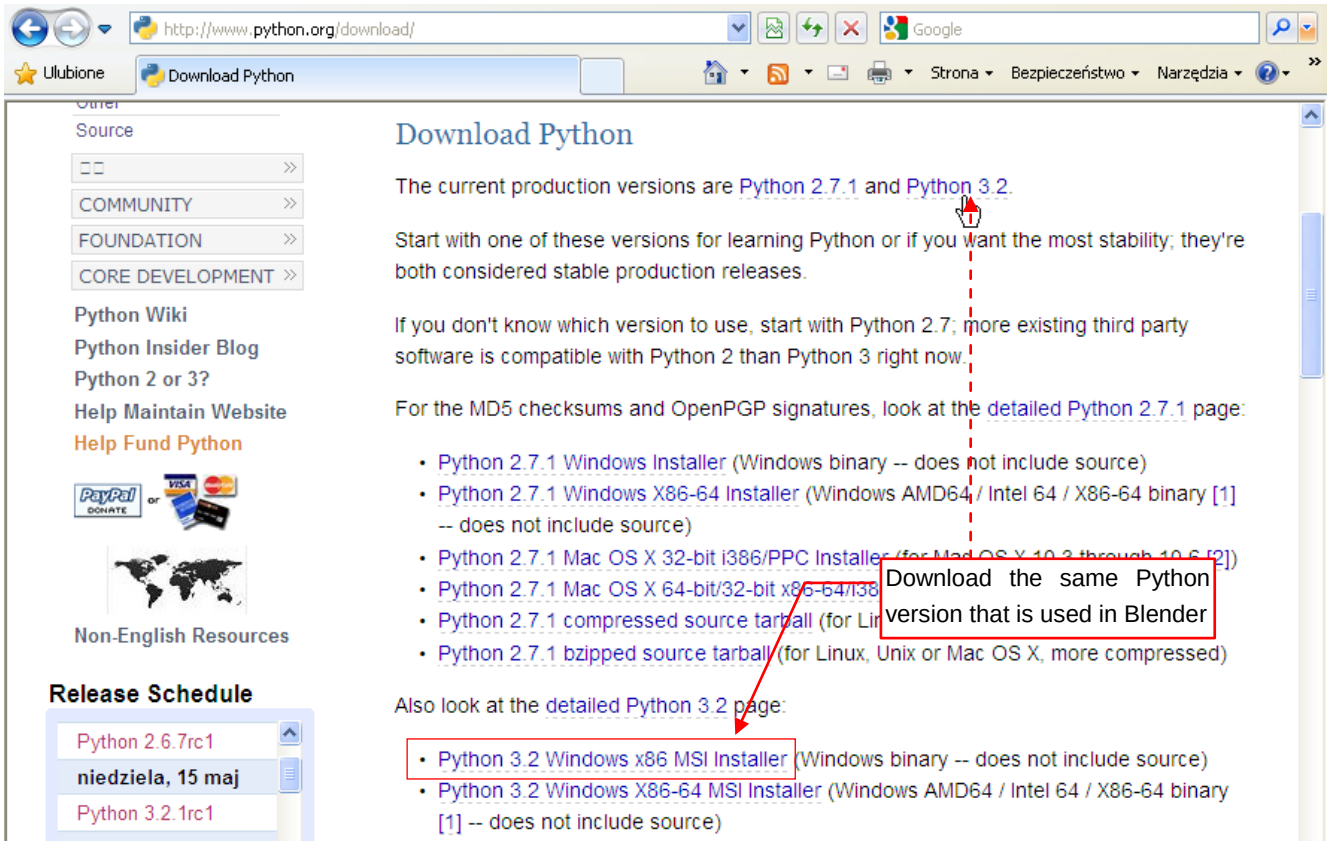


Figure 5.1.2 The download page, with various Python versions

Select the same Python version, which is used in your Blender. (If you cannot find the one with identical version — select the closest one).

Click in the selected link and choose the *Run* option (Figure 5.1.3):

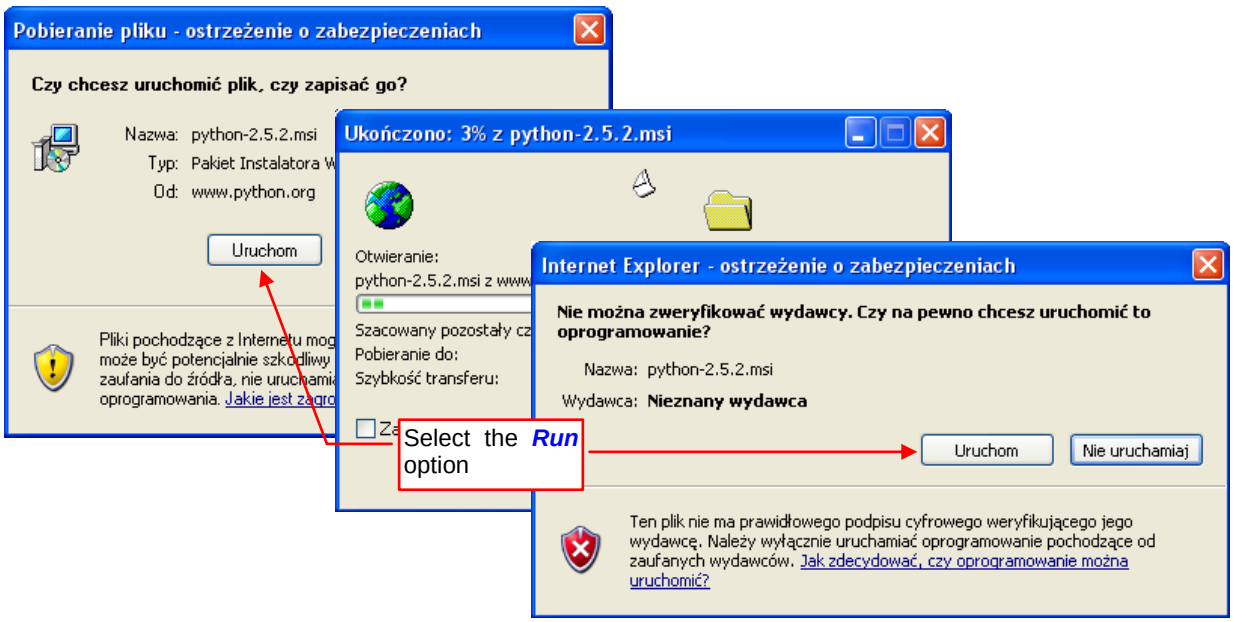


Figure 5.1.3 Downloading the installation program from the Python portal

(If you do not like to run the programs from the Internet directly, you can save this file on your disk, first).

Make sure, that you have the full (i.e. Administrator) privileges to your computer, and run the installation program. Go through the installer screens, just pressing the *Next* button (Figure 5.1.4):

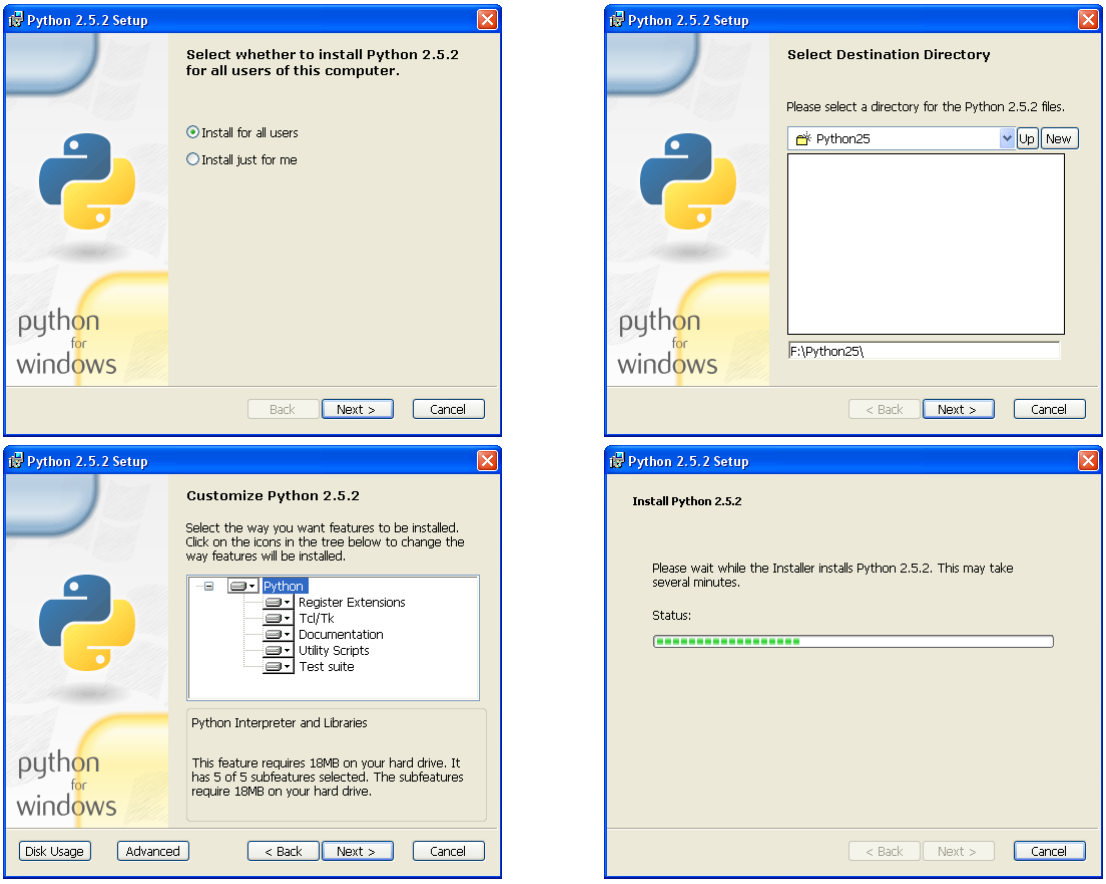


Figure 5.1.4 Subsequent screens of the Python installer

At the end, the program will display such a screen (Figure 5.1.5):

Figure 5.1.5 The last screen of the Python installation

Press the ***Finish*** button, to finish this process.

5.2 Details of the Eclipse and PyDev installation

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

Let's start by downloading Eclipse. Go to the http://www.eclipse.org/downloads page (Figure 5.2.1):

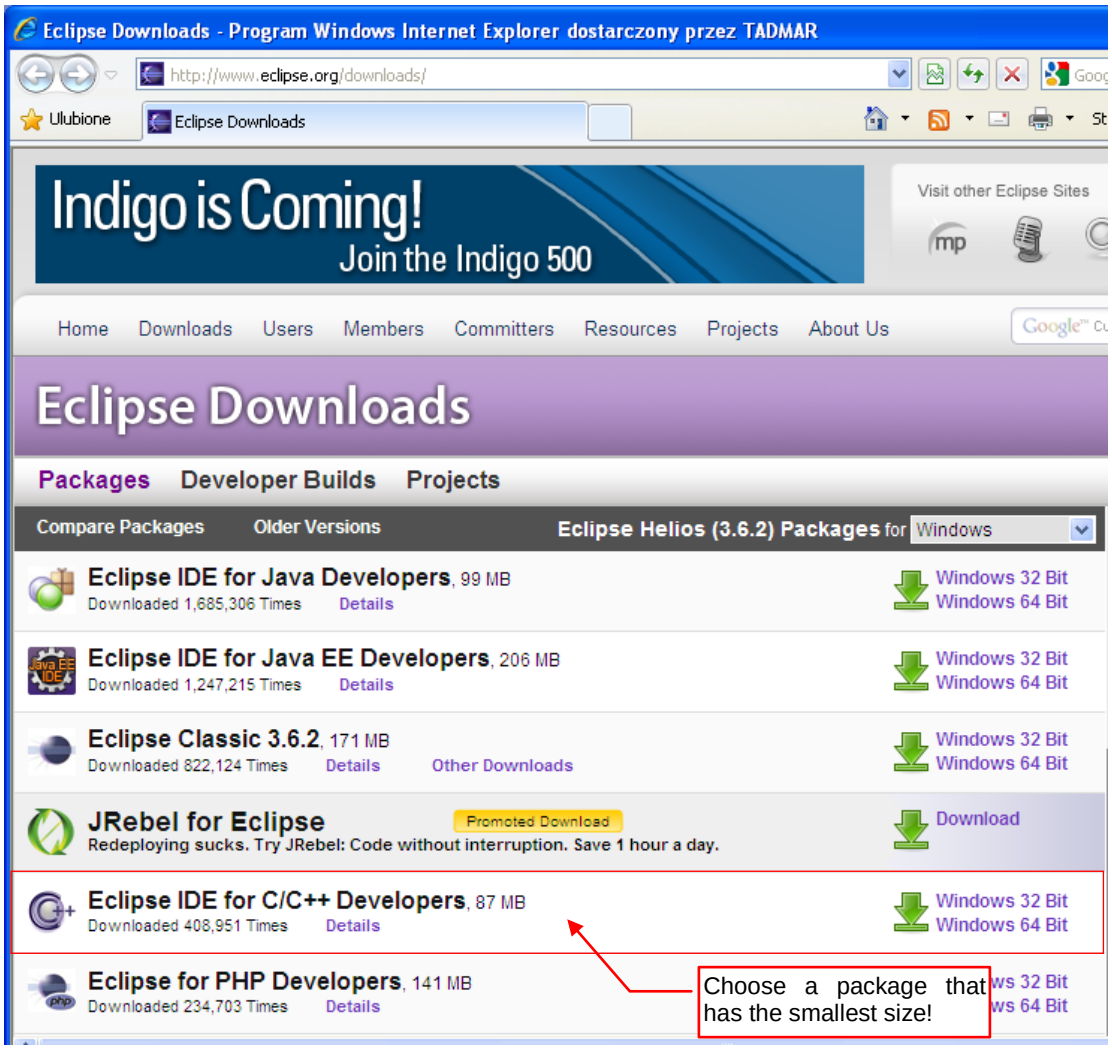


Figure 5.2.1 Selection of the Eclipse package

In fact, Eclipse is a kind of open programmers environment. It is just a framework, which can be adapted to work with any programming languages by appropriate plugins. On the Eclipse Internet site you can find some popular plugin packages for the most popular languages. There is no ready "Eclipse for Python" bundle among them, so we will make it ourselves. Just download any of these packages (I have chosen the one that has the smallest size). Since July 2011, it is ***Eclipse for Testers*** (87 MB), which uses the Eclipse version 3.7 („Indigo"). I wrote this book using the earlier version: 3.6 („Helios"). In that version, the smallest package was ***Eclipse IDE for C/C++ Developers*** (also 87 MB). *Eclipse for Testers* will install the PyDev plugin somewhat longer, but later its *eclipse.exe* will open the whole environment a little bit faster.

All the Eclipse packages are just plain **.zip* files. Save the downloaded one somewhere on your disk. (To write this book, I have downloaded file named *eclipse-cpp-helios-SR2-win32.zip*).

[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.

The downloaded file contains the *eclipse* folder, with the program ready to run (Figure 5.2.2):

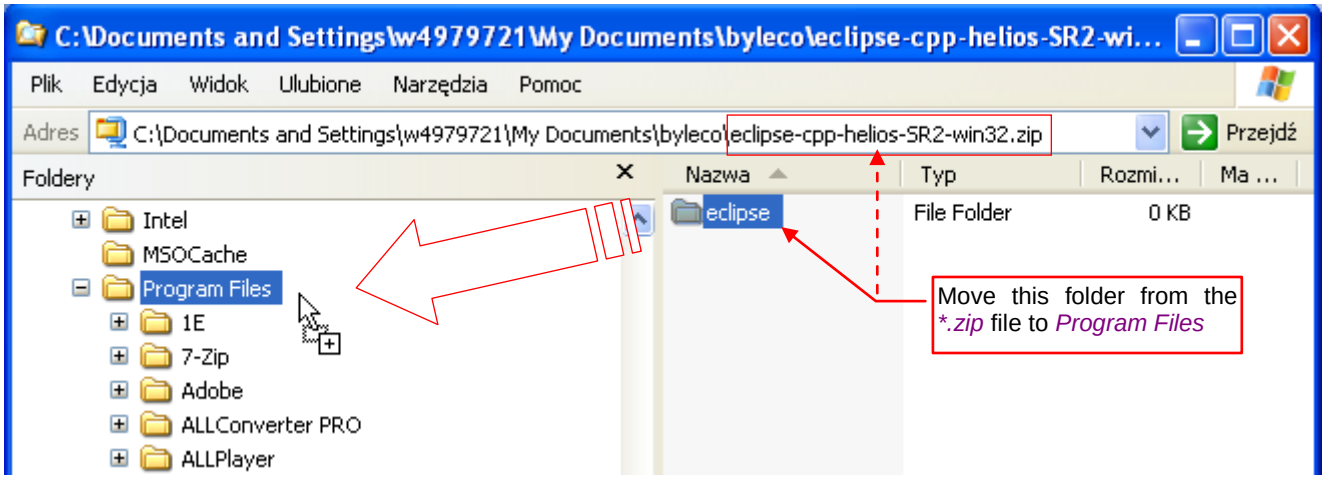


Figure 5.2.2 Unpacking the program folder

Just extract it to the *Program Files* folder. (Yes! There is no installer, which would do some unclear settings in your precious system! Eclipse has no external dependencies except the Java Virtual Machine, and does not change anything in the Windows registry. Thus, you can simultaneously use many alternate Eclipse versions, without any conflict).

To launch Eclipse, just run the *eclipse.exe* program (Figure 5.2.3):

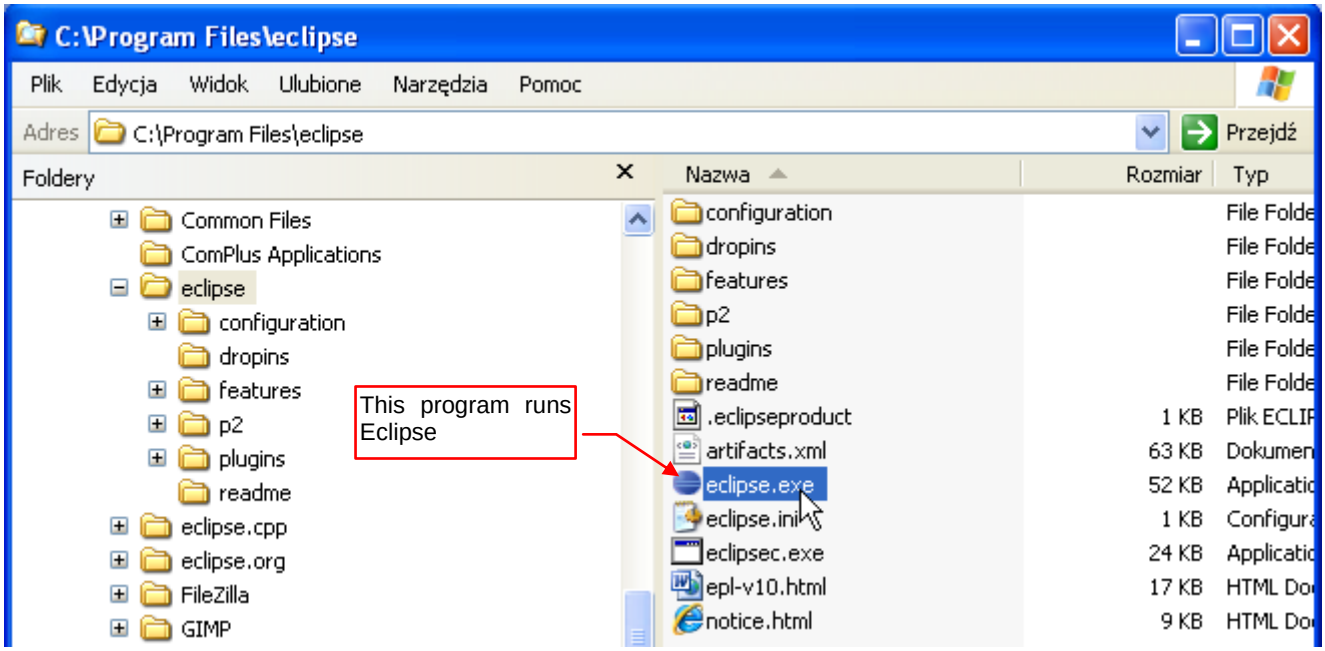


Figure 5.2.3 Launching Eclipse

You can insert a shortcut to this file in your favorite menu or place it on your desktop.

When you launch *eclipse.exe* program, it always displays a dialog box where you can select the location of the Eclipse projects directory (it is called “workspace”). You may just confirm this default (Figure 5.2.4):

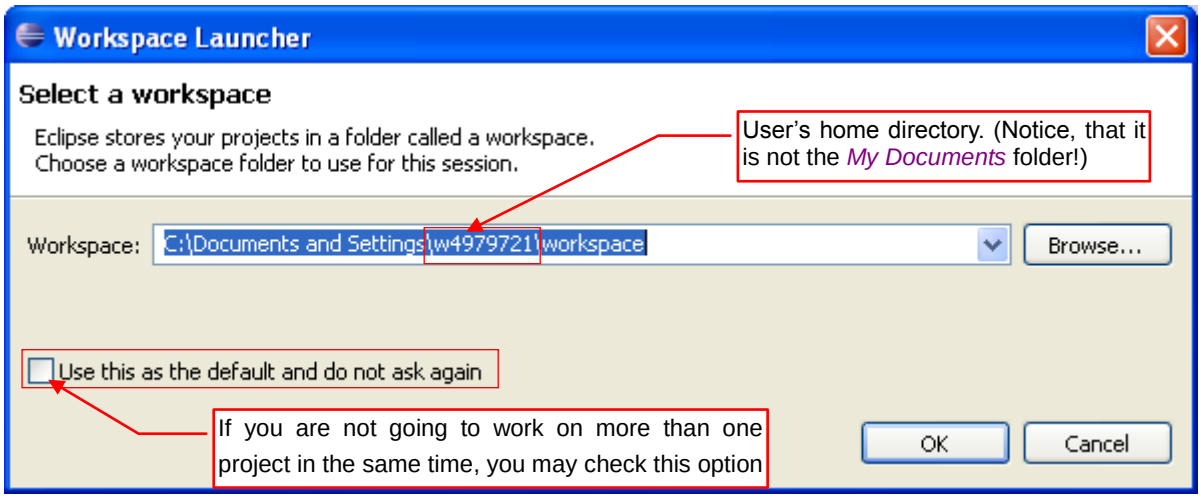


Figure 5.2.4 Selecting the current workspace

Each of Eclipse projects is a separate folder, containing a few configuration files, and the files with your code. (If your script is located elsewhere on the disk, you can put in the project just its shortcut). Notice that the default Eclipse workspace folder is located in the root directory of the user profile. (In this example, the username is W4979721). This is not *My Documents* folder — just one level up. (It is the *Unix/Linux* convention of the home directory). If you keep all your data in the *My Documents* folder — change the path displayed in this window. Eclipse will create the appropriate directory, if it does not exist.

Eclipse is always trying to open in the workspace the recently used project. On the first launch it is impossible, because the workspace folder is empty. Therefore, the version 3.6 displays following warning (Figure 5.2.5):

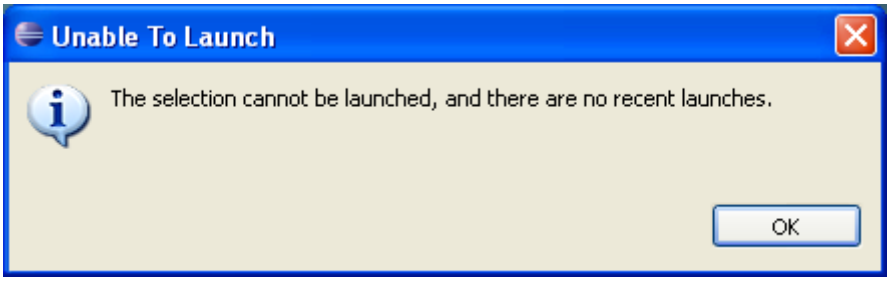


Figure 5.2.5 The warning, displayed on the first launch of an Eclipse workspace

They have fixed it in the version 3.7. In any case, there is nothing to worry about.

When there is no active project, in the current workspace, Eclipse displays the Welcome pane. (Figure 5.2.6):

Figure 5.2.6 Eclipse window on the first launch

The best way to install the PyDev plugin is to use Eclipse internal plugin management facilites. Invoke the ***Help→Install New Software*** command (Figure 5.2.7).

(The location of this command in the *Help* menu may be a little surprise for regular Windows users . They rather would expect it in the *Edit* or the *File* menu. It is just specific for Eclipse.)

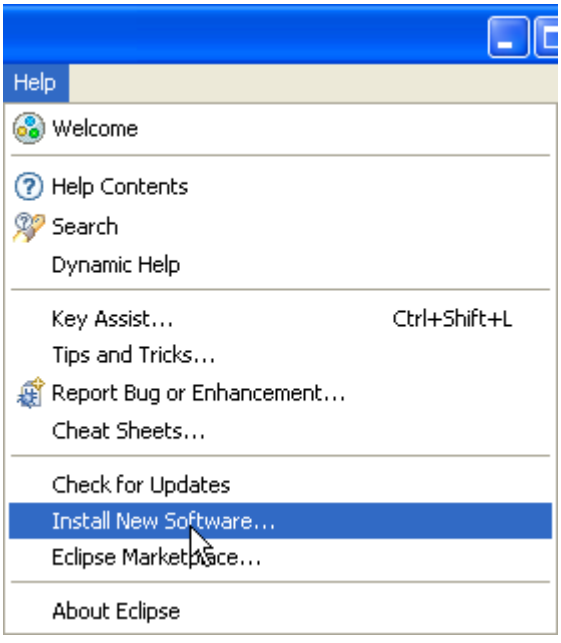


Figure 5.2.7 Installation of an Eclipse plugin

In the ***Install*** dialog, type the address of the PyDev project automatic updates page: http://pydev.org/updates (Figure 5.2.8):

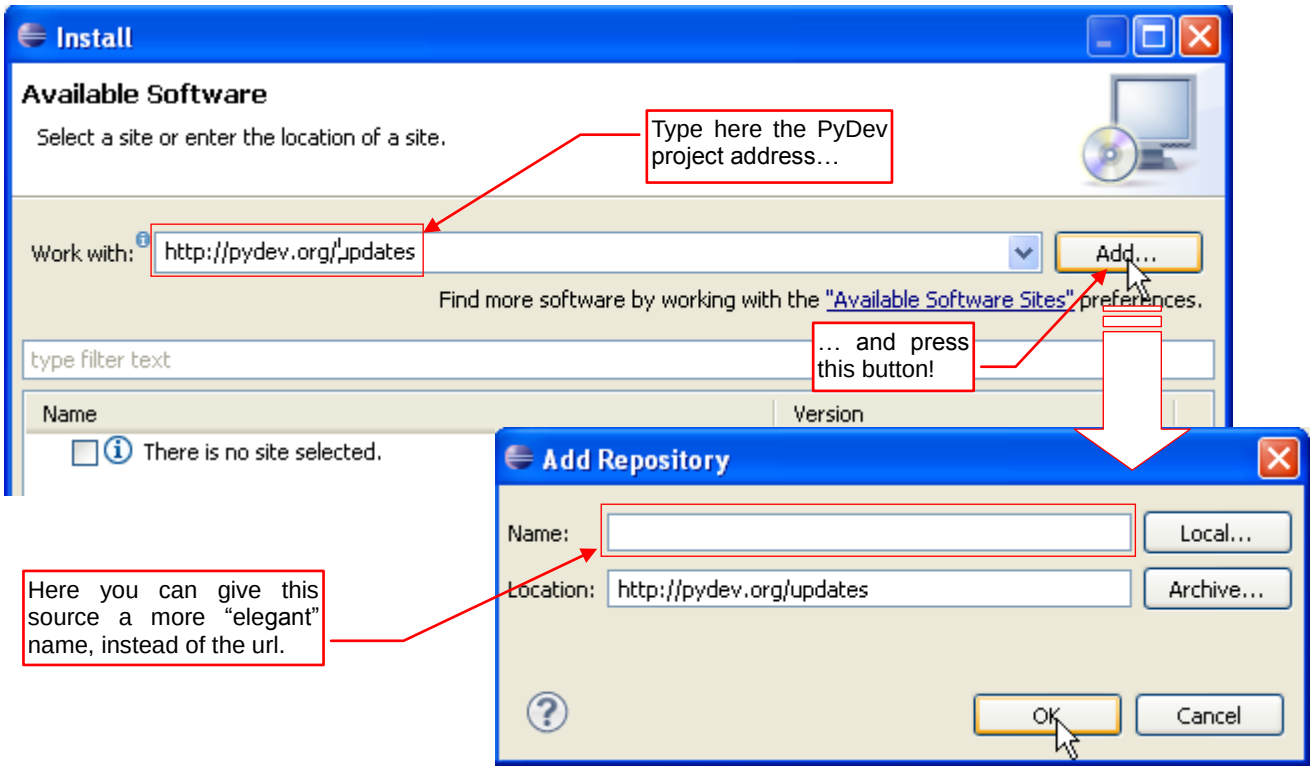


Figure 5.2.8 Adding to the software vendors list the PyDev entry

Then press the ***Add*** button. It opens the ***Add Repository*** dialog box. When you confirm it, Eclipse will read the components, exposed on this page (Figure 5.2.9):

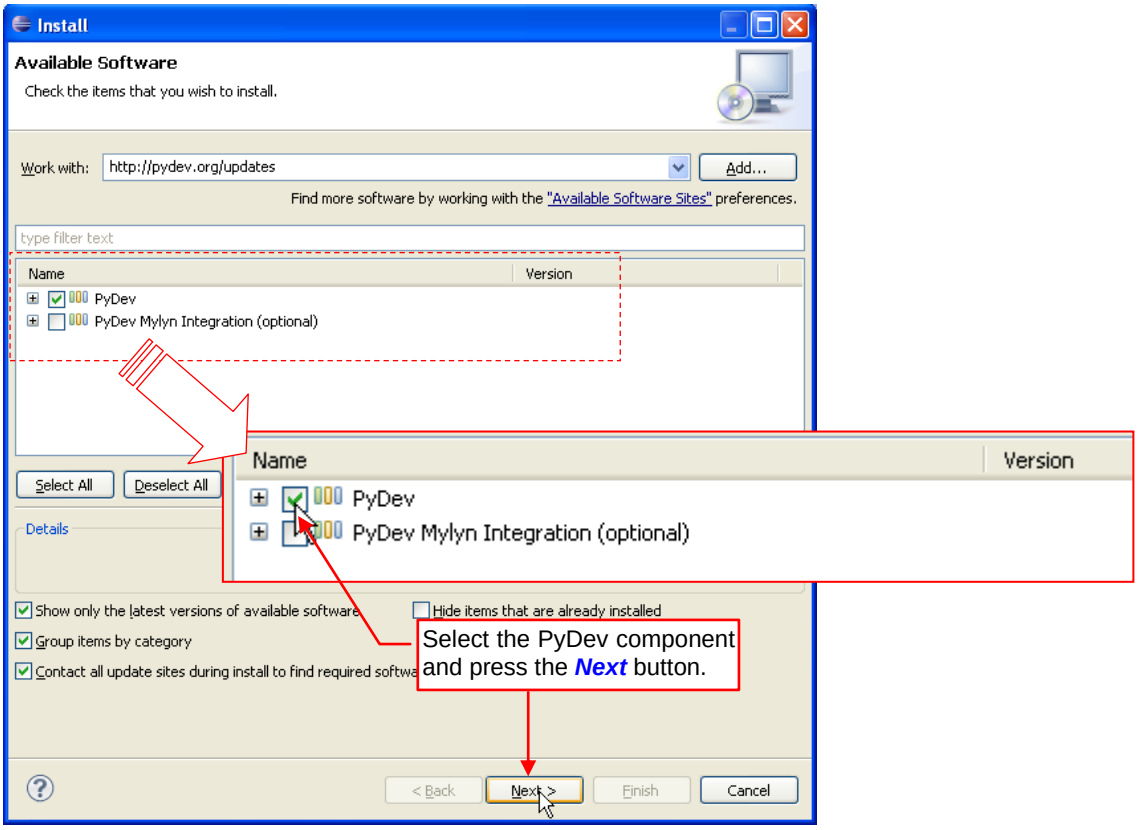


Figure 5.2.9 Selection of the PyDev plugin from its project site

Select on the list the PyDev component and press the ***Next*** button.

Eclipse will display an additional list containing the details of installed components (Figure 5.2.10):

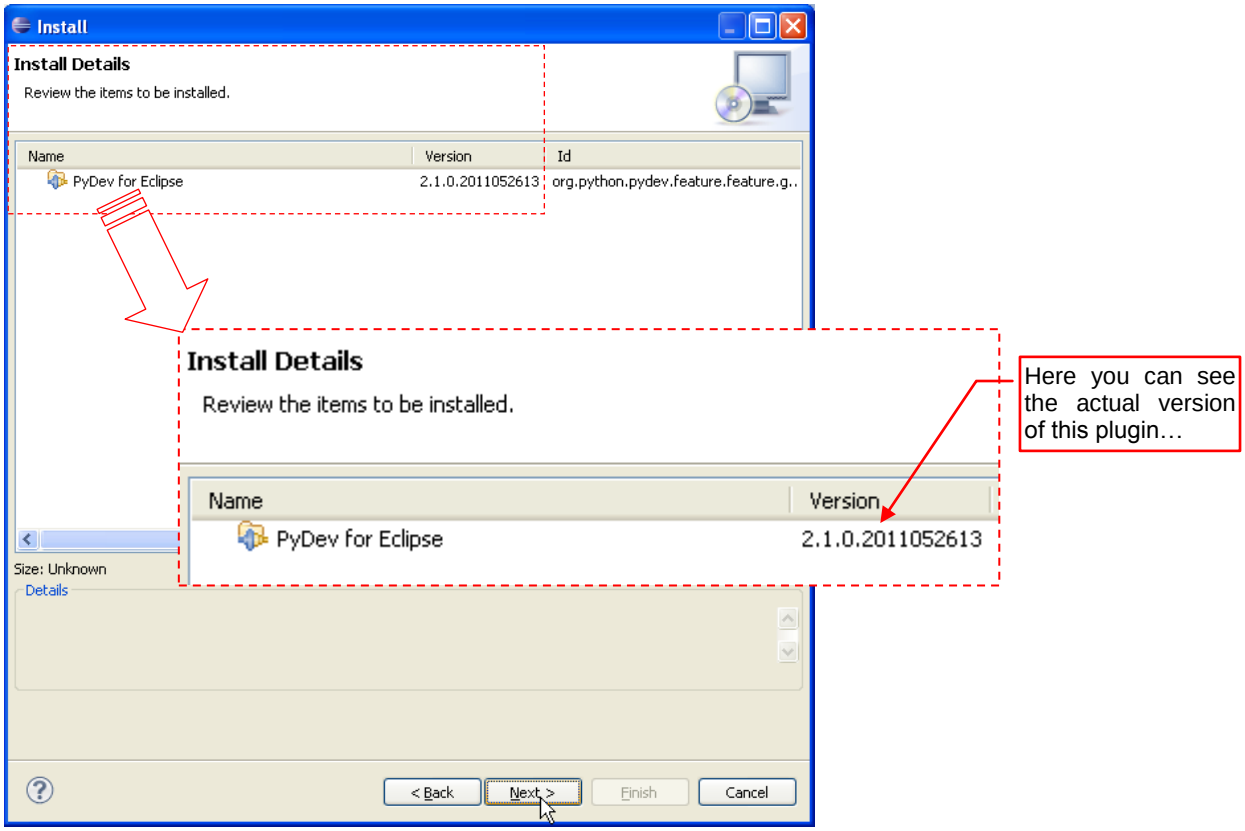


Figure 5.2.10 Confirmation of the installation details

After pressing another ***Next*** button, Eclipse will display the PyDev license agreement, for your acceptance (Figure 5.2.11):

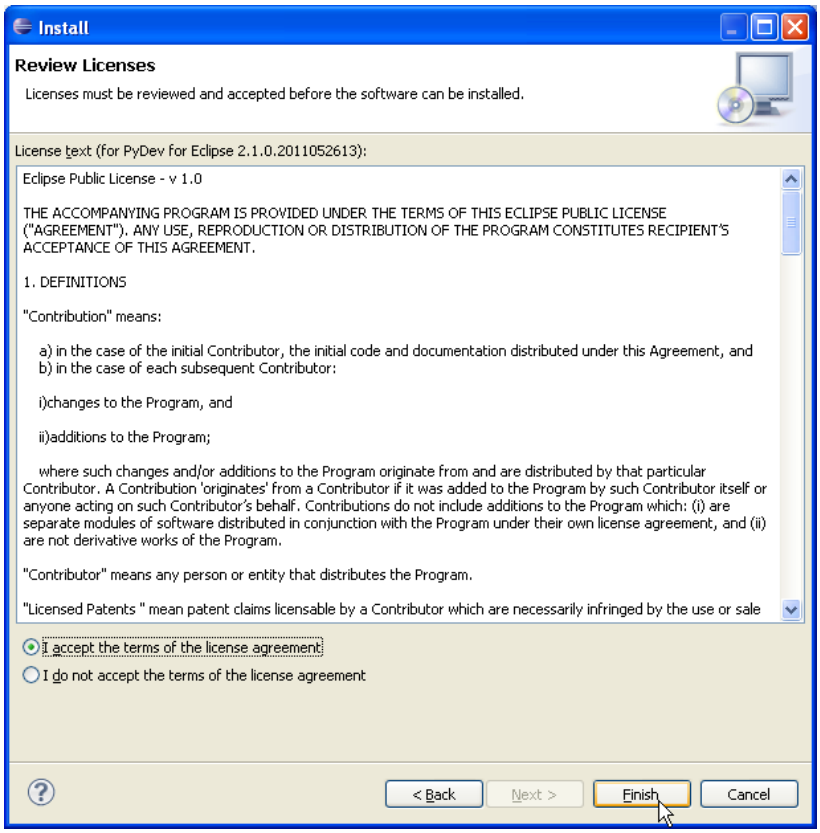


Figure 5.2.11 Acceptance of the PyDev license agreement

When you press the ***Finish*** button, it will launch the installation process.

During the installation, Eclipse downloads from the Internet the appropriate components. It shows the standard progress dialog (Figure 5.2.12):

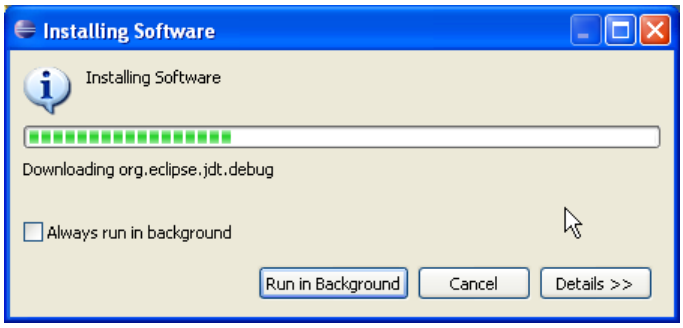


Figure 5.2.12 The progress of the installation

Once the download is complete, Eclipse will ask you for the confirmation of the plugin certificate, in the next window (Figure 5.2.13):

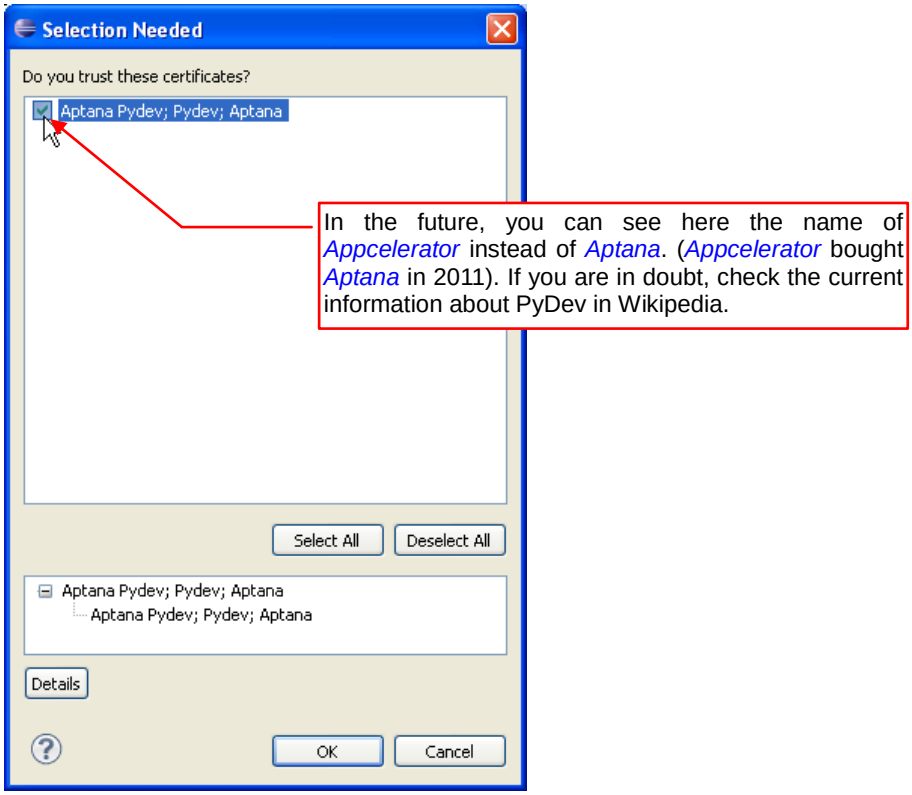


Figure 5.2.13 PyDev certificate confirmation

After confirmation of the certificate the last window will appear, finishing the installation process. (Figure 5.2.14):

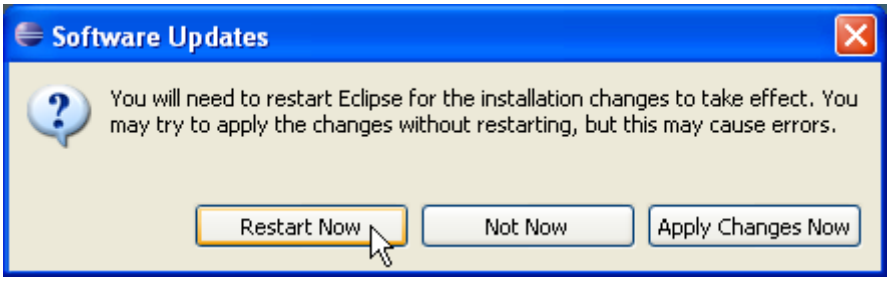


Figure 5.2.14 Final window of the PyDev plugin installation

I think it is always worth to agree on the proposed restart of the Eclipse.

5.3 Details of the PyDev configuration

Once installed, you have to configure the default PyDev Python interpreter. This information is stored in the current Eclipse *workspace* (ref. page 11, Figure 1.2.4). To set it, use the ***Window→Preferences*** command (Figure 5.3.1):

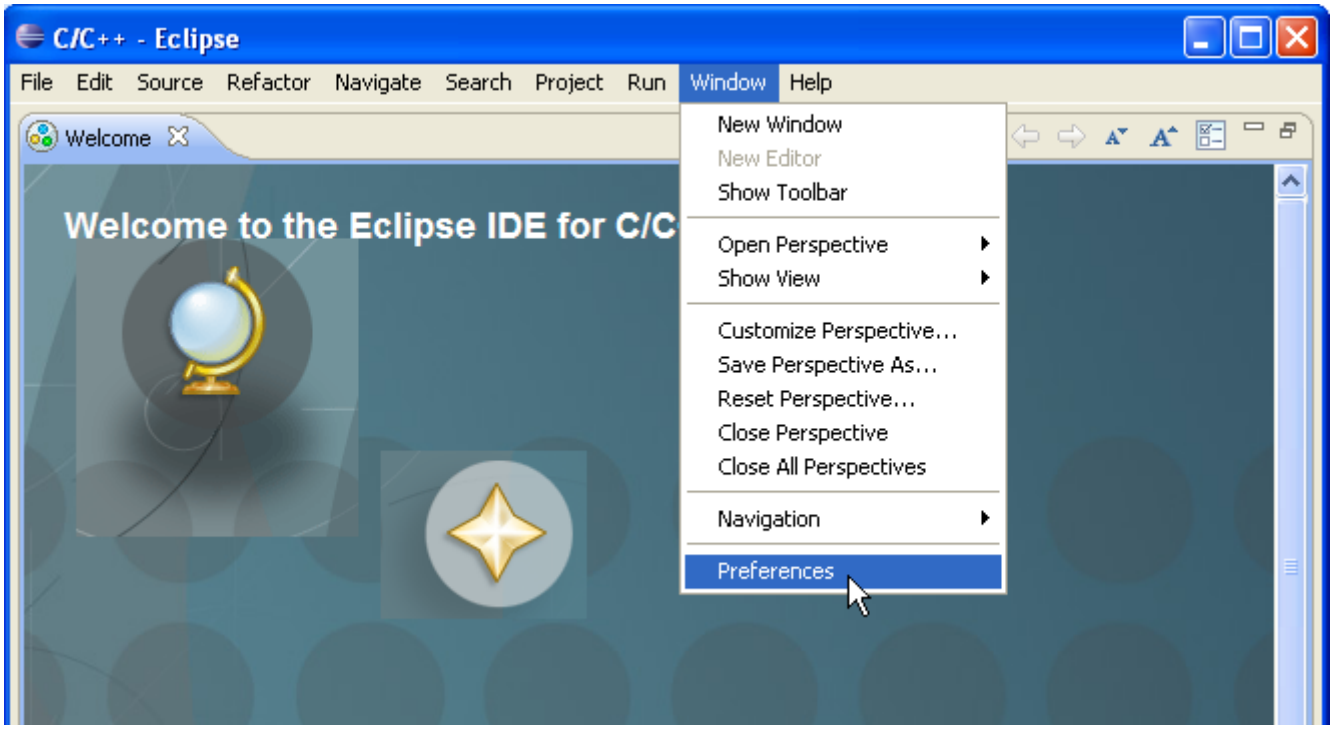


Figure 5.3.1 Opening the current *workspace* configuration

In the *Preferences* window expand the ***PyDev*** section and highlight ***Interpreter - Python*** (Figure 5.3.2):

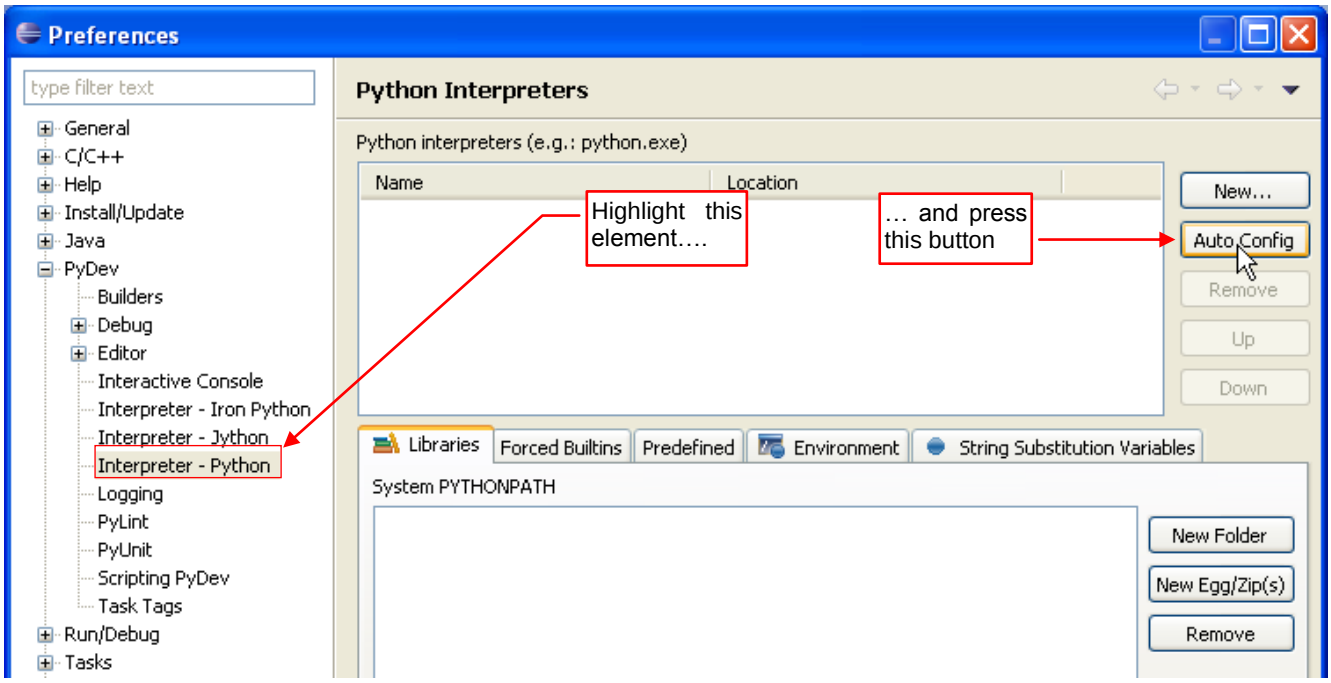


Figure 5.3.2 Automatic configuration of the Python interpreter

Then press the ***Auto Config*** button. If the path to your external Python interpreter is added to the **PATH** system variable, PyDev will find it. (In this case, the configuration wizard will open the window shown in Figure 5.3.5). It will also find alternative interpreters, if they are installed in their default directories. In such a case, PyDev will display their list, asking you to select one.

However, if the program could not find Python - it displays a message (Figure 5.3.3):

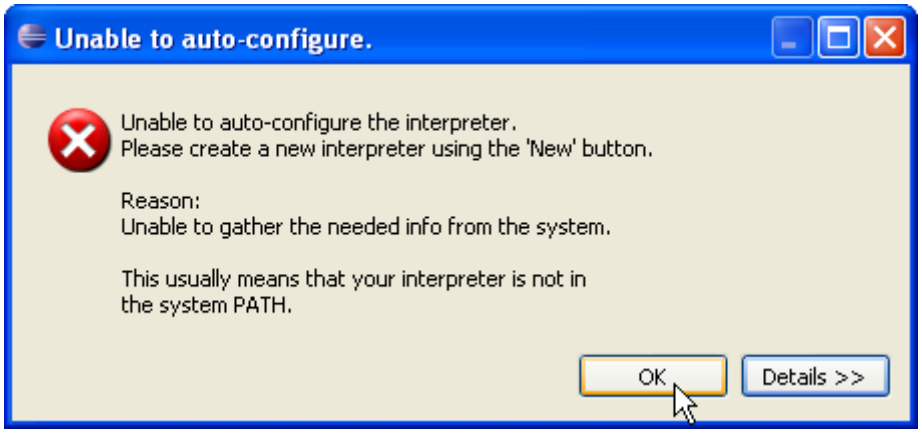


Figure 5.3.3 PyDev warning, when it cannot find the Python interpreter

In such a case, in the *Properties* window press the ***New*** button (see Figure 5.3.2). It will open the window of „manual" Python selection (Figure 5.3.4):

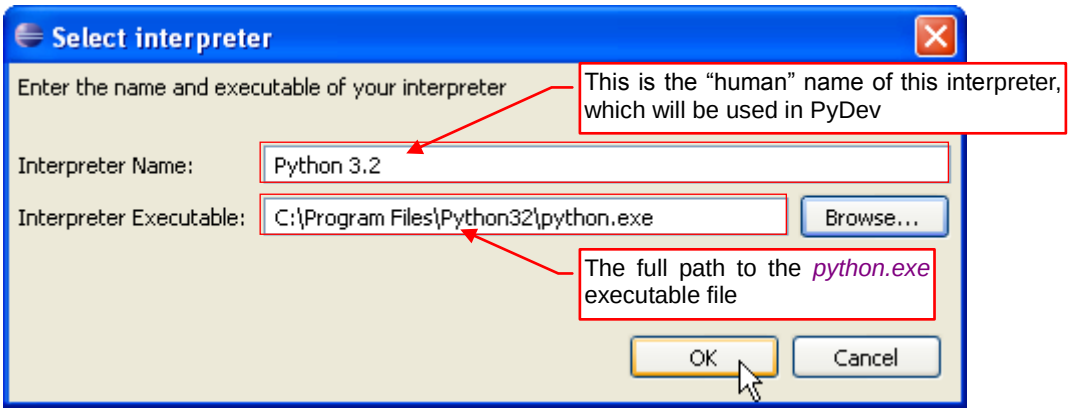


Figure 5.3.4 "Manual" Python configuration

If you do not make any mistake in the path, then after pressing the ***OK*** button PyDev will display another window with some Python directories. The selected ones will be added to the **PYTHONPATH** configuration variable (Figure 5.3.5). Just accept it without any changes:

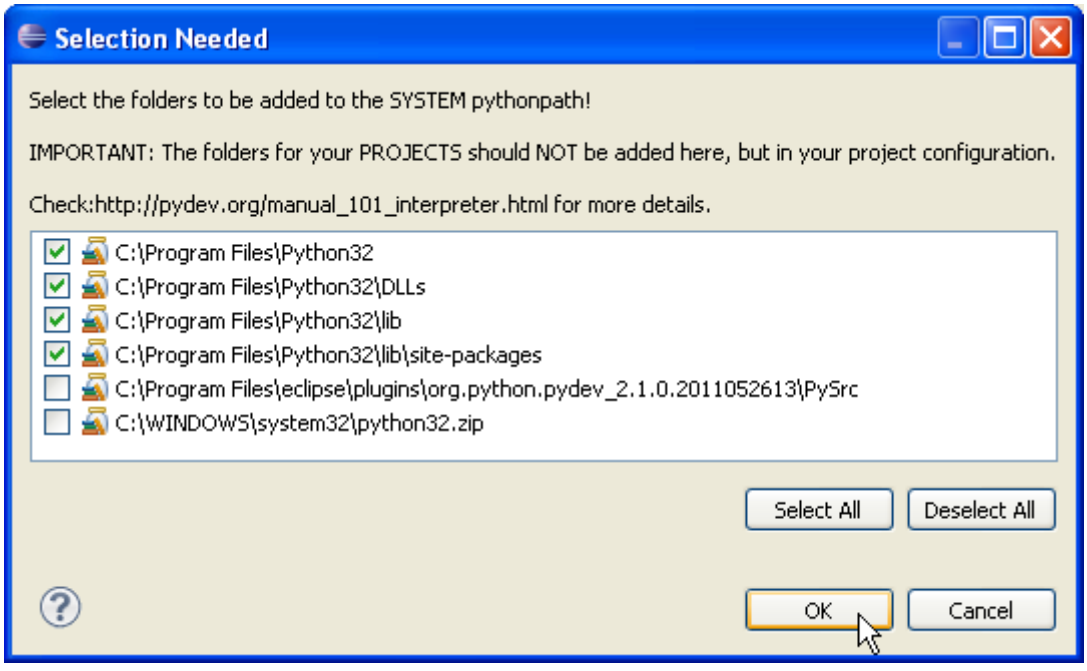


Figure 5.3.5 Selection of the directories that will be added to the PYTHONPATH system variable

In the result, the configured Python interpreter appears in the *Preferences* window (Figure 5.3.6):

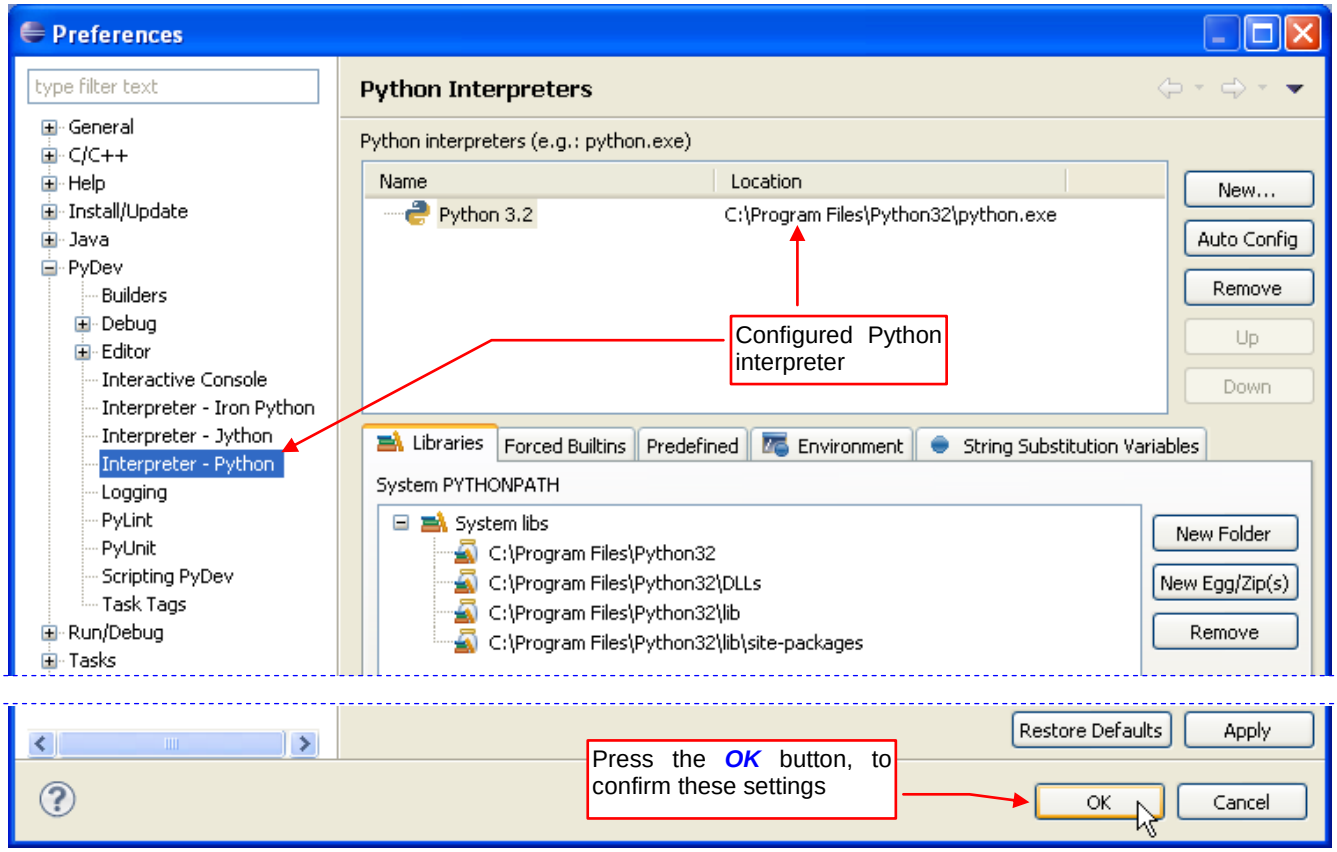


Figure 5.3.6 Configured Python interpreter

When you accept this, pressing the ***OK*** 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 5.3.7):

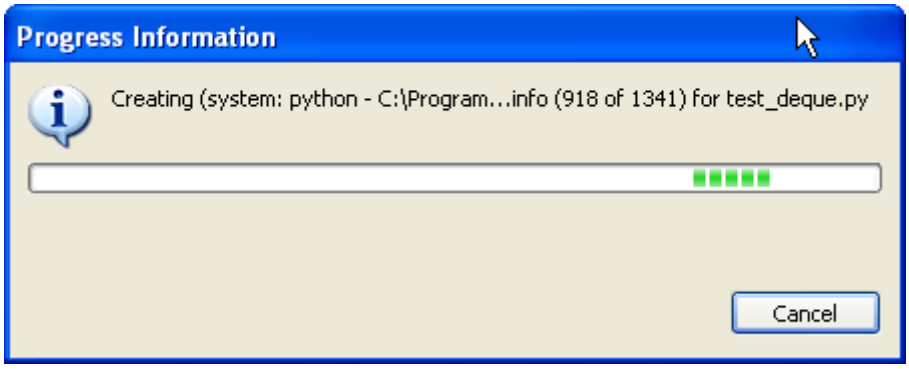


Figure 5.3.7 Processing the PYTHONPATH **files**

- Beware: During the installation of PyDev version 2.2.1 (more precisely: 2.2.1.2011071313) in the Eclipse 3.7 („Indigo") *Eclipse for Testers* package, I saw an error message of Java runtime exception, in one of the running programs. It had appeared on this last stage of the Python interpreter configuration. Nevertheless, the Python files were still processed (underneath the window with this message), and this processing was finished within a few seconds. This error did not cause any noticeable irregularities in the operation of the Eclipse + PyDev environment.

During the earlier installations — PyDev version 2.1.0 (2.1.0.2011052613) in the Eclipse 3.6 („Helios") *Eclipse for C/C++ Developers* package, there were no such errors.

Chapter 6. Others

In this chapter, you will find all the other additional materials of this book. (It is kind of the final "hodgepodge". It would be too difficult to divide this information into a well-organized structure).

6.1 Updating the Blender API predefinition files

The Blender Python API changes a little in each Blender release. In the *doc* folder (see page 39) you will find the shortcut, that will update your PyDev predefinition files for this latest version (Figure 6.1.1):

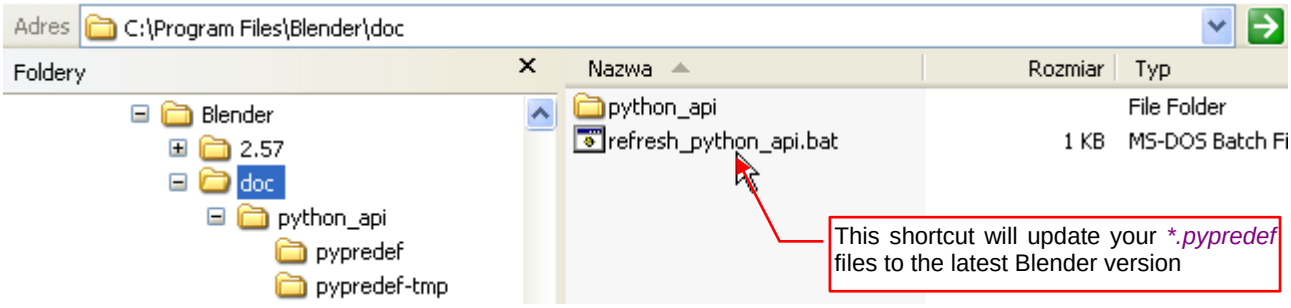


Figure 6.1.1 Contents of the *doc* folder

This Windows batch file calls the *pypredef_gen.py* script, from *doc\python_api* directory (Figure 6.1.2):

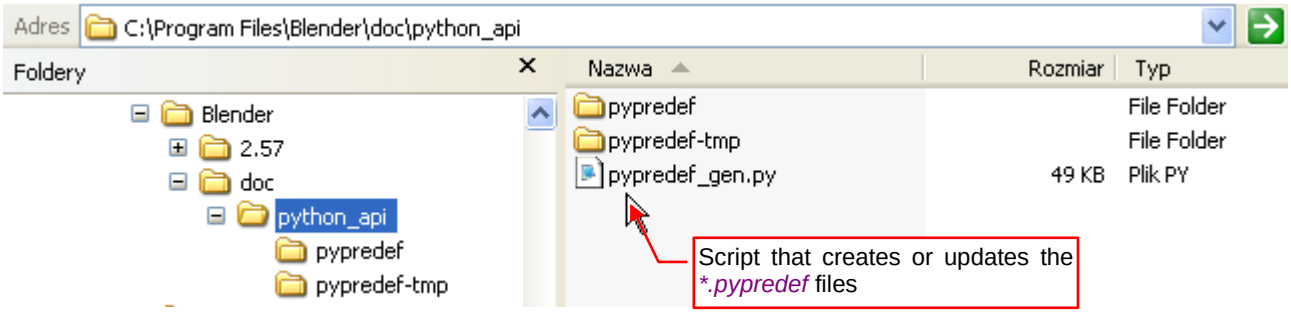


Figure 6.1.2 Contents of the *doc\python_api* folder

Theoretically *pypredef_gen.py* should run properly also in other operating systems, like Linux. I have not tried it. This script is a reworked version of the *sphinx_doc_gen.py*, developed by Campbell Barton for automatic generation of the Blender API documentation. (The same, which is published on the blender.org pages). Thanks to this code, descriptions of all functions and methods in the PyDev predefinition files are the same as in the official API documentation. Just like there, they contain even the descriptions of each procedure parameter. The only module that is not documented this way is ***bge***. In addition, the ***bpy.context*** has some gaps, because it has the variable structure that depends on the kind of the current Blender editor (*View 3D*, *Python Console*, etc).

The result of the *pypredef_gen.py script* — the **.pypredef* files for the corresponding Python API modules — are placed in the *doc\python_api\pypredef* folder (Figure 6.1.3):

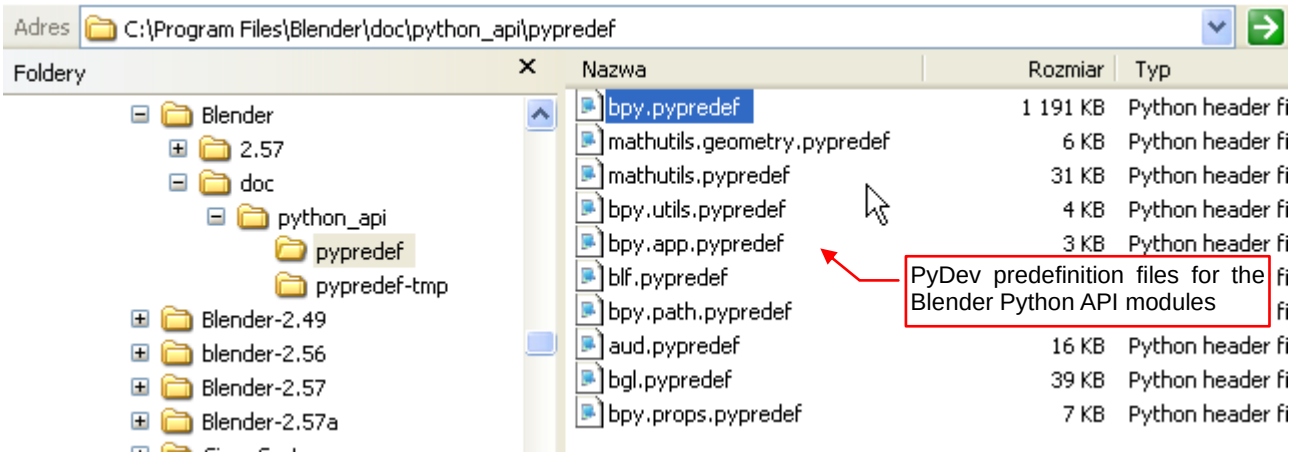


Figure 6.1.3 Contents of the *doc\python_api\pypredef* folder

This folder should be referenced in the PyDev project configuration as the external library (see page 40).

When you upload a new Blender version (or move the *doc* folder into the directory with another Blender release) run the *doc\refresh_python_api.bat* shortcut (Figure 6.1.4):

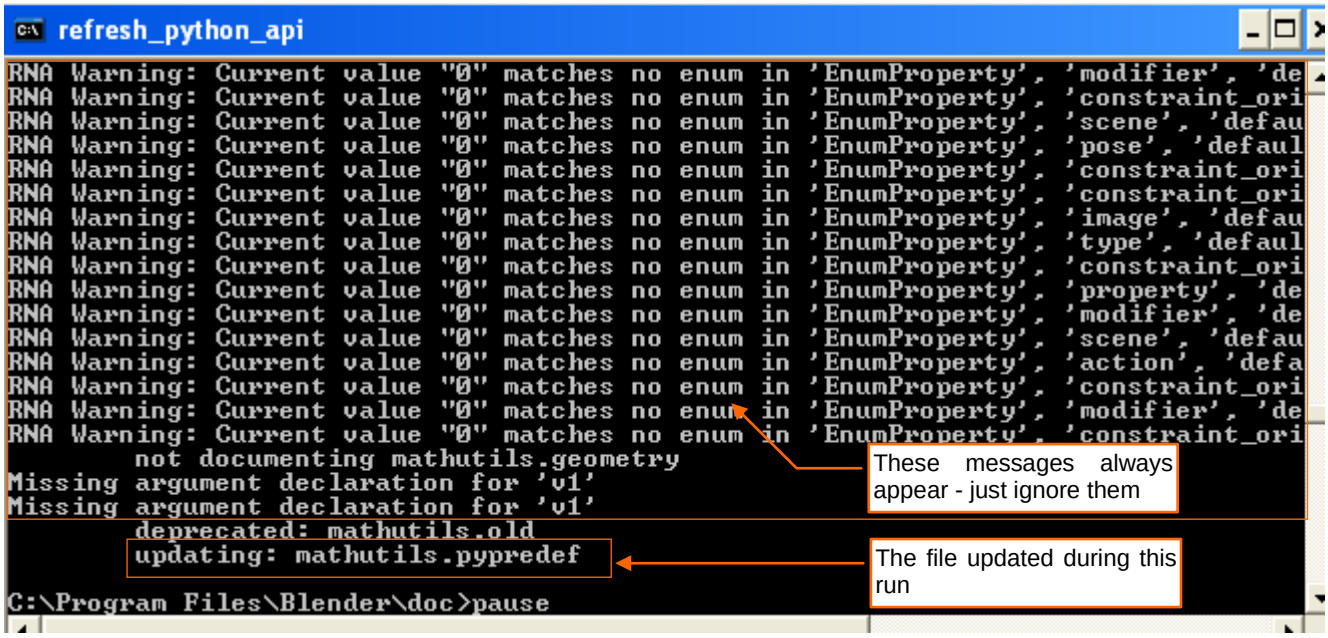


Figure 6.1.4 Updating of the Blender API predefinition files

That is all. Just remember to add the *doc\python_api\pypredef* path to the configuration of each PyDev project. To do it, go to the project properties (Figure 6.1.5):

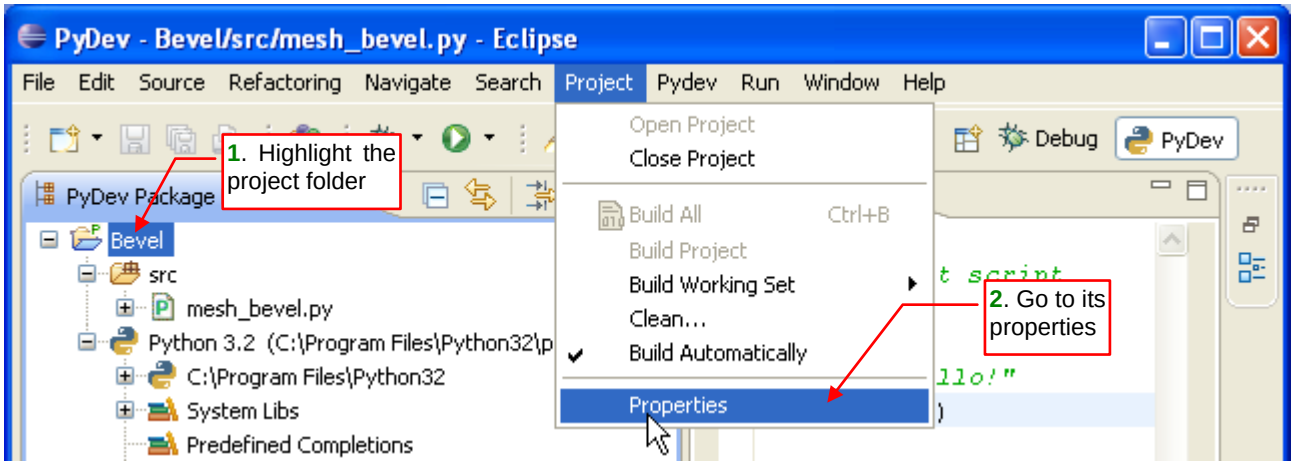


Figure 6.1.5 Opening the project properties

In the *Properties* window select the ***PyDev – PYTHONPATH*** section, and then, on the right pane — the ***External Libraries*** tab (Figure 6.1.6):

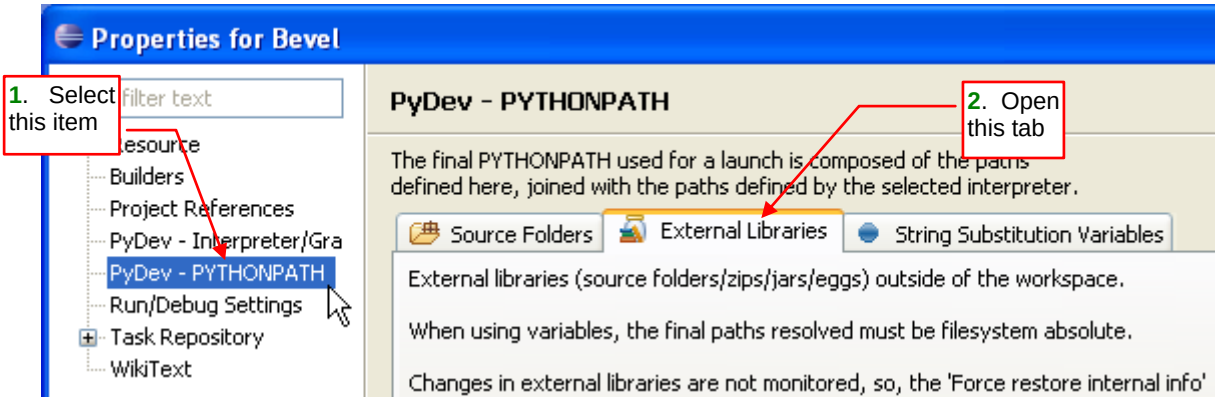


Figure 6.1.6 Opening the *PyDev - PYTHONPATH* pane

Initially, the project does not have any external libraries (the list in this tab is empty). Press the ***Add source folder*** button, and add the *doc\python_api\pypredef* directory (Figure 6.1.7):

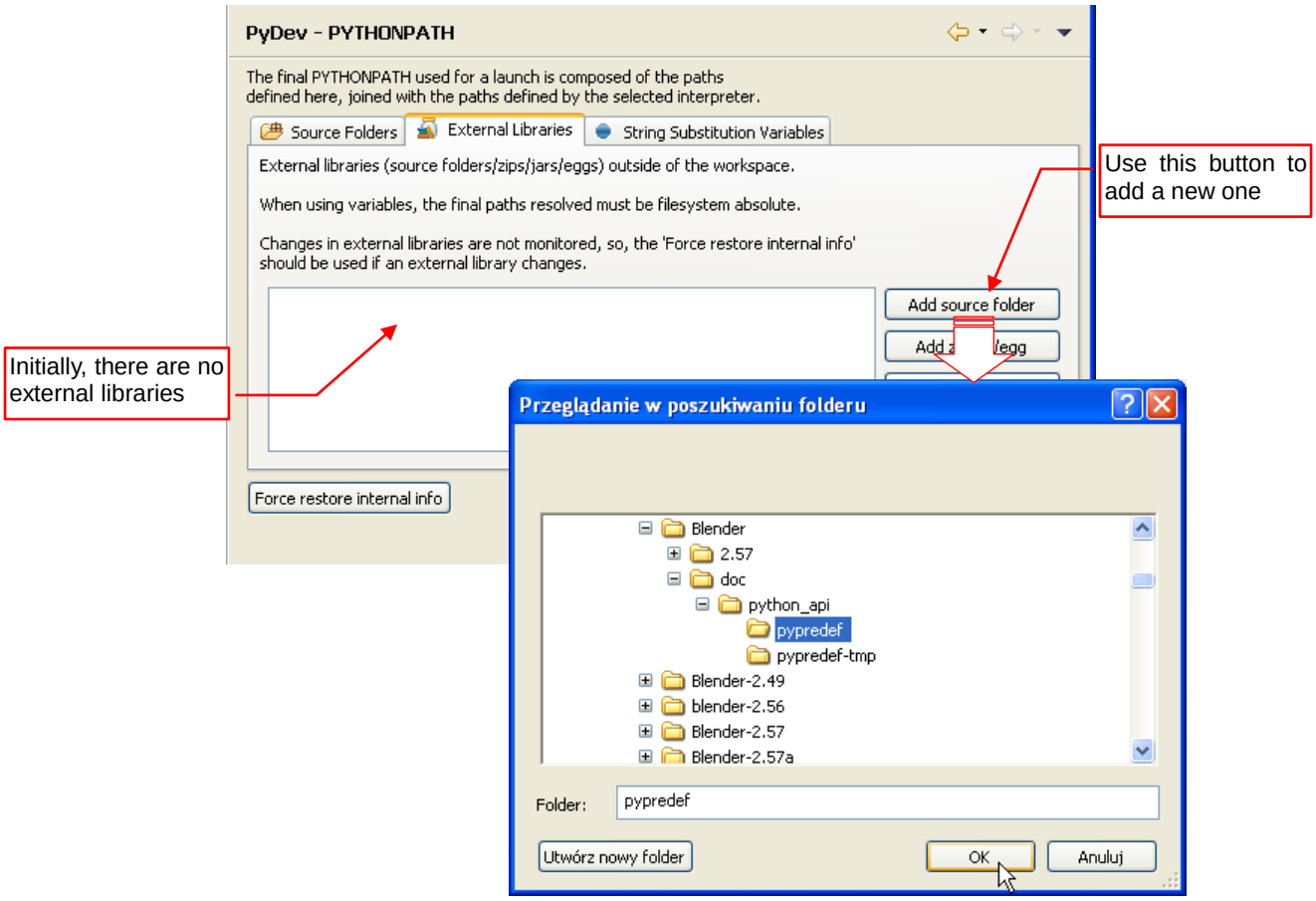


Figure 6.1.7 Adding the PyDev predefinition files folder as an "external library"

When the folder with Blender API files is already on the list, press the ***Force restore internal info*** button. From the description in the window, it seems that you have to do it after each change in this list (Figure 6.1.8):

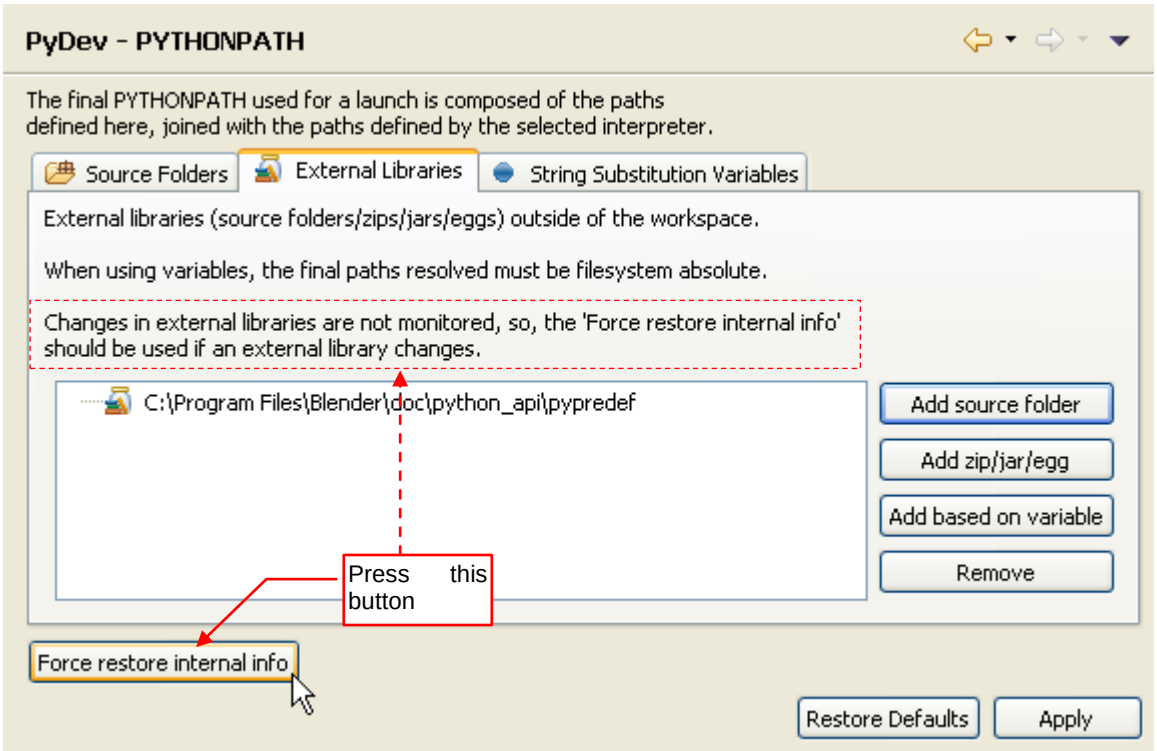


Figure 6.1.8 Forcing the refreshing of the project internal data

When you have approved the changes in the project configuration, add to the beginning of the script the „*import **bpy***" statement. It suggests PyDev that it should use the declarations from the ***bpy*** (Blender API) module. Then, at the time of writing the code, just type a dot after the name of the object. PyDev will display the list of its methods and fields (Figure 6.1.9):

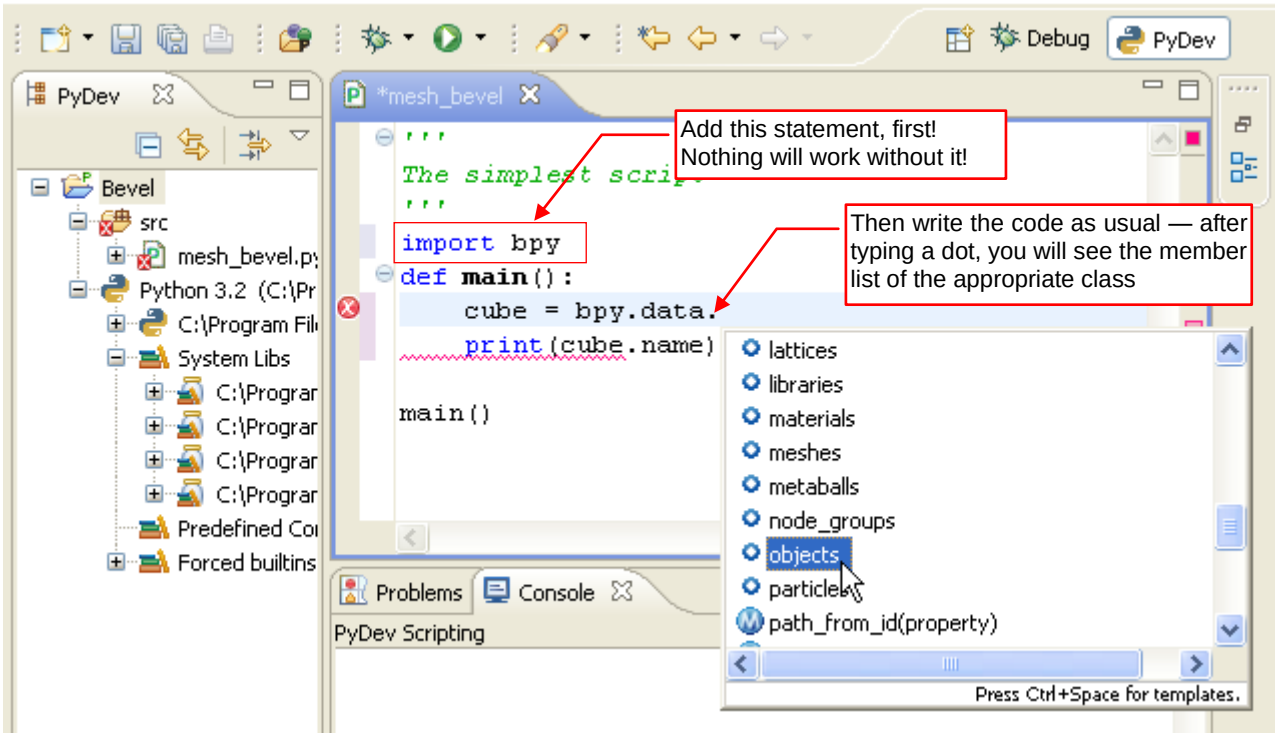


Figure 6.1.9 Code autocompletion — after typing a dot

More about the PyDev autocompletion functions you can find on page 41 and the next.

- Beware: On the official PyDev page (pydev.org) you can find different description how to use the predefinition (**.pypredef*) files[1]. The problem is that the addition to the *Predefined* section, described there, did not work on my computer. Thus, I introduced here the proven and effective, although somewhat "unorthodox", method.

Moreover I believe, that assigning such a Blender API reference to the particular project, not the whole workspace, is better. You can write in parallel yet another project in the external Python. In such a project, the hints on the Blender API would only disturb the user.

[1] It is described in this article: http://pydev.org/manual_101_interpreter.html

6.2 Importing an existing file to the PyDev project

You can add to the PyDev project the files that have already existed on your disk. Pull down the context menu from the folder, where you want to have them, and invoke the ***Import…*** command (Figure 6.2.1):

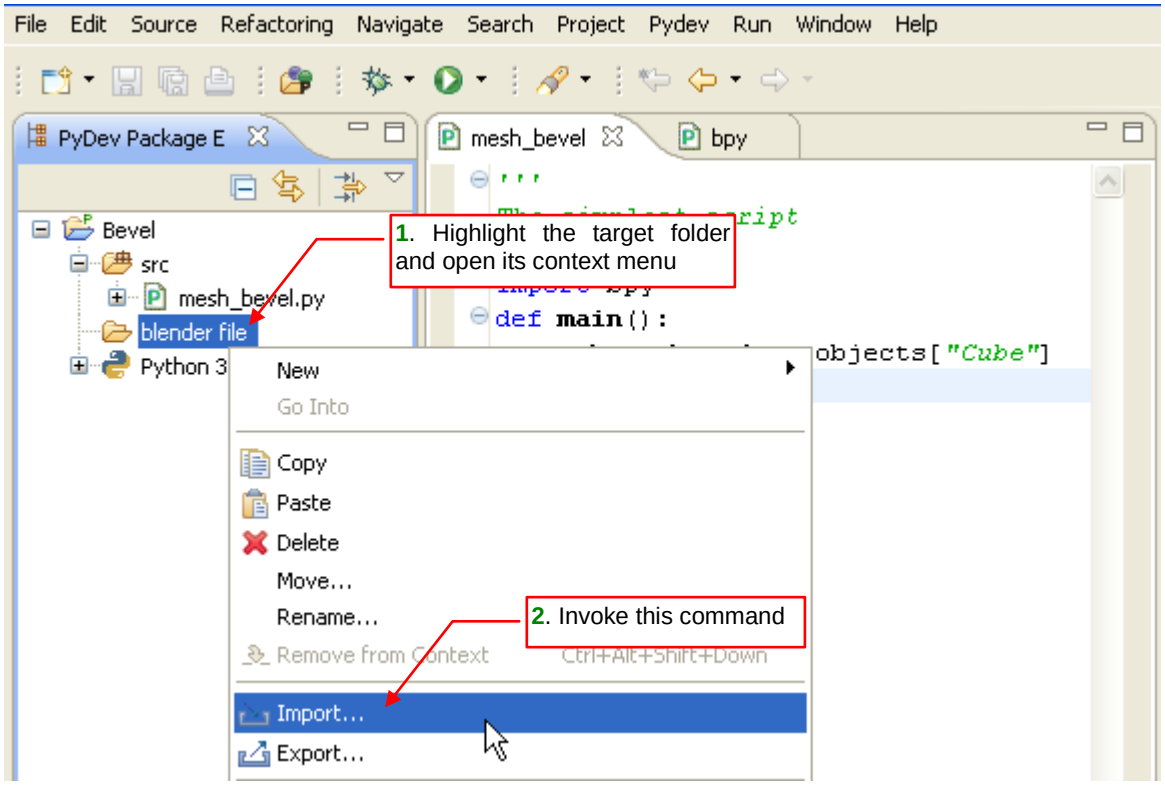


Figure 6.2.1 Importing an existing file to the project folder

It opens the *Import* wizard window. On the first pane, select the ***General→File System*** item as the source (Figure 6.2.2):

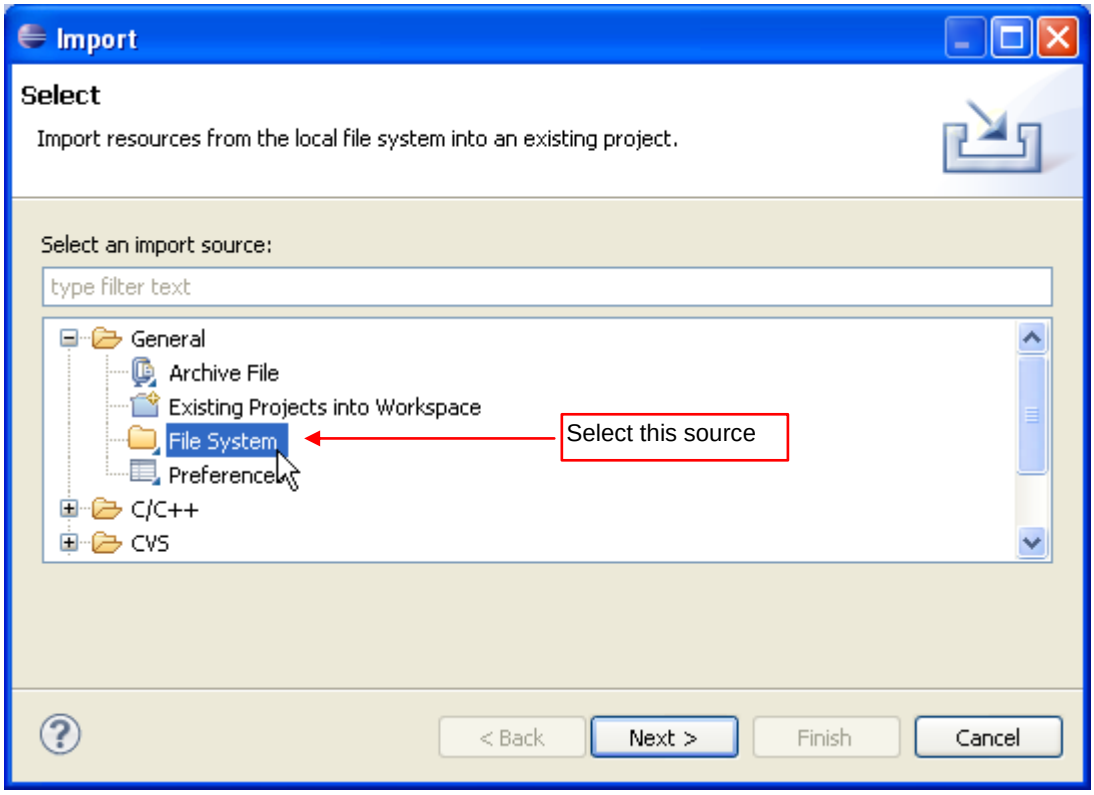


Figure 6.2.2 Selecting the import source

On the next pane, select the folder that contains the file/files to import (Figure 6.2.3):

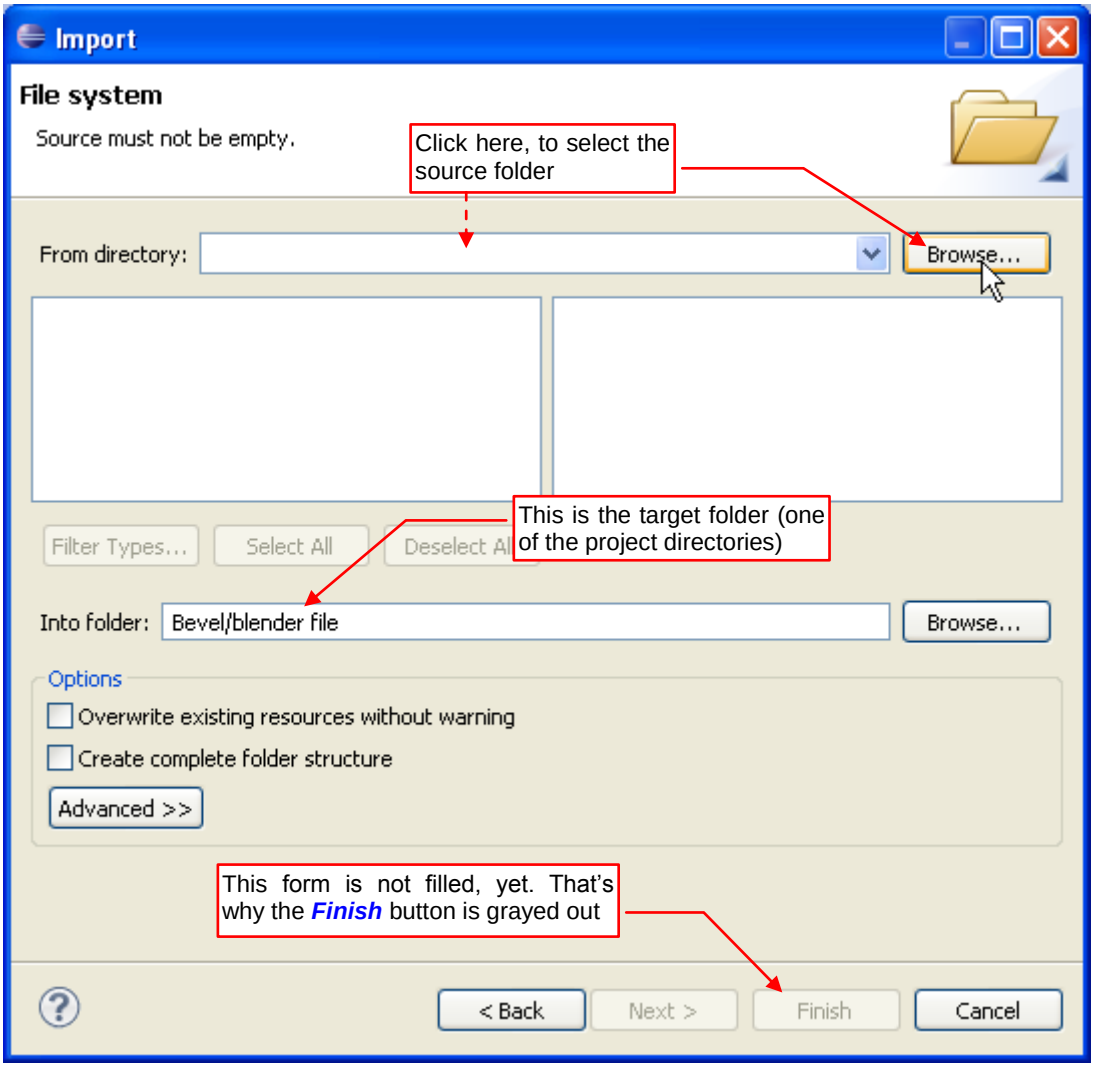


Figure 6.2.3 Empty import wizard pane

To select an existing source folder, or to create a new target one, use the ***Browse...*** buttons (Figure 6.2.4):

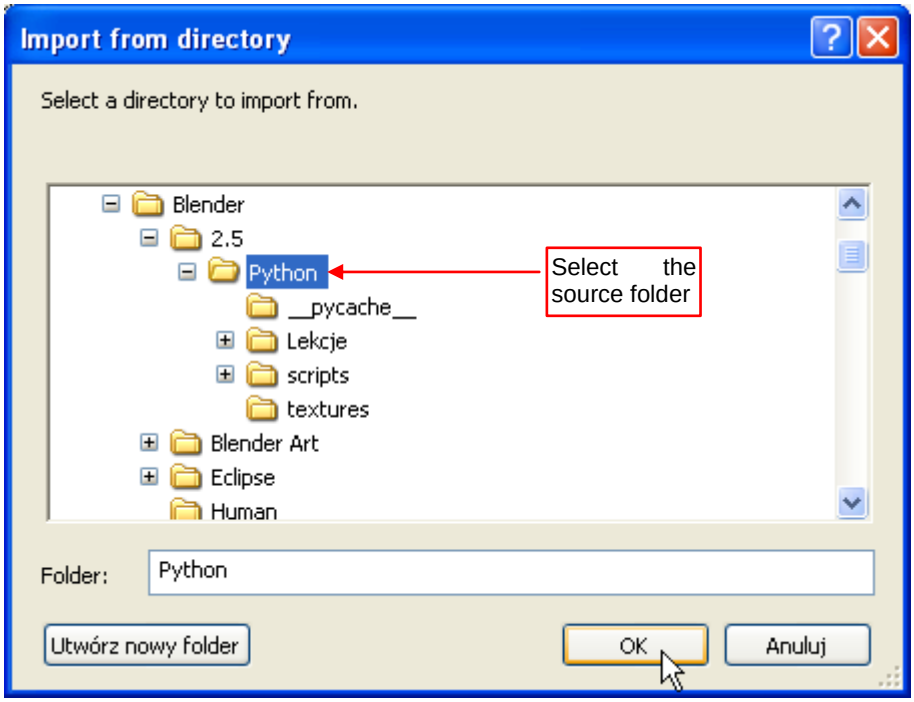


Figure 6.2.4 Selection of the folder

When the source directory is selected, you will see its content in the right pane (Figure 6.2.5):

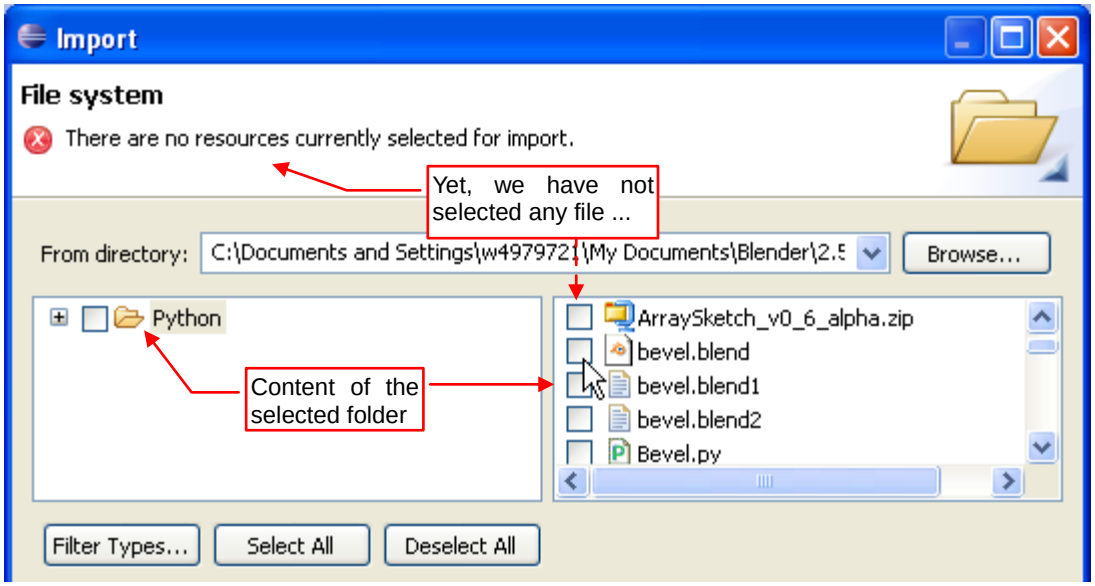


Figure 6.2.5 Displaying the source folder content

Select at least one file to import (Figure 6.2.6):

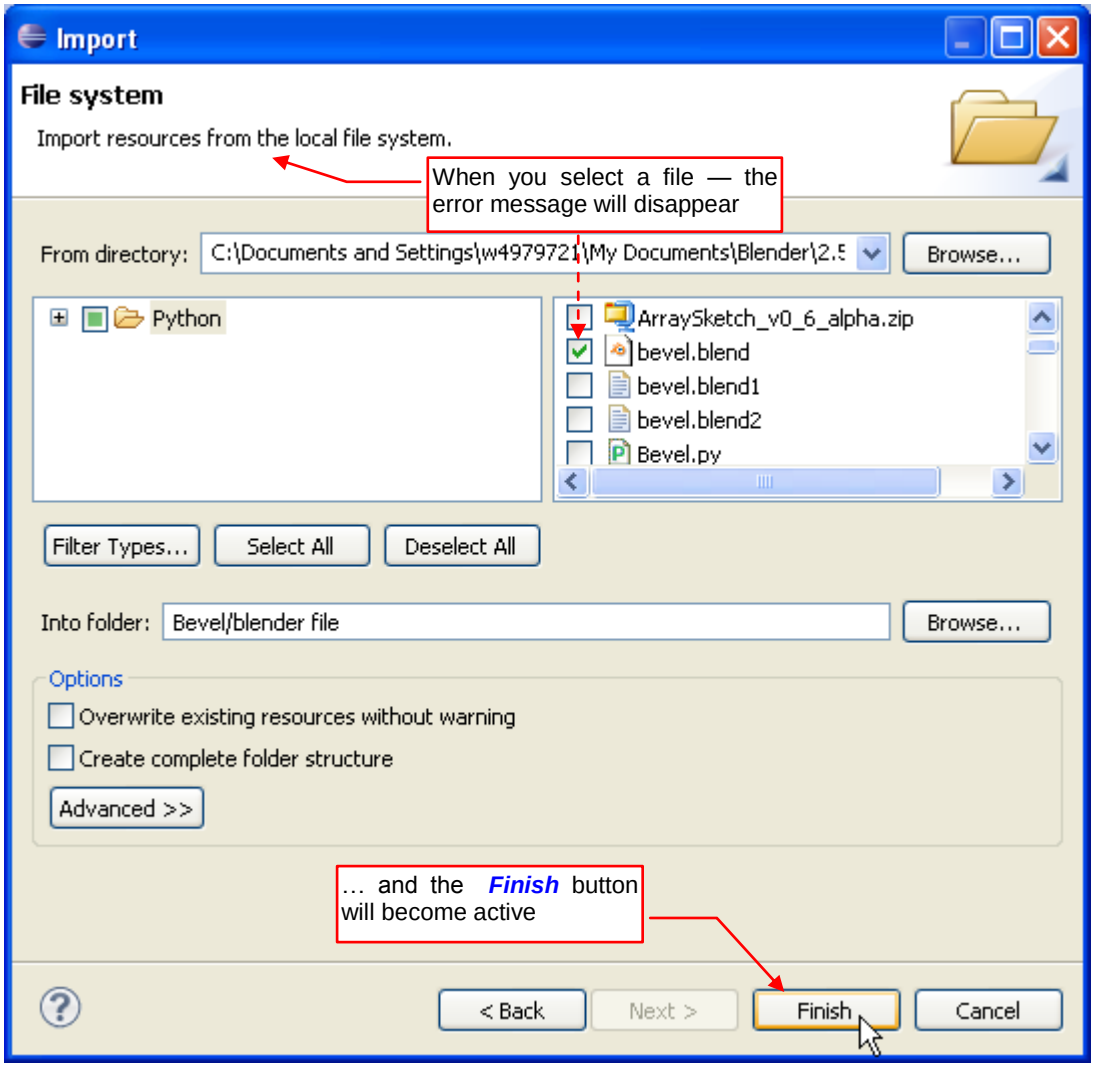


Figure 6.2.6 Selection of the imported file

When you do this, the wizard will conclude that it has all the data, now. It will activate the ***Finish*** button. Press it, to copy the selected files to the target project folder.

You can import to your PyDev project various files, for example — the Blender file with the testing environment. Just double click, to open it (Figure 6.2.7). In this way, you have everything "in one place":

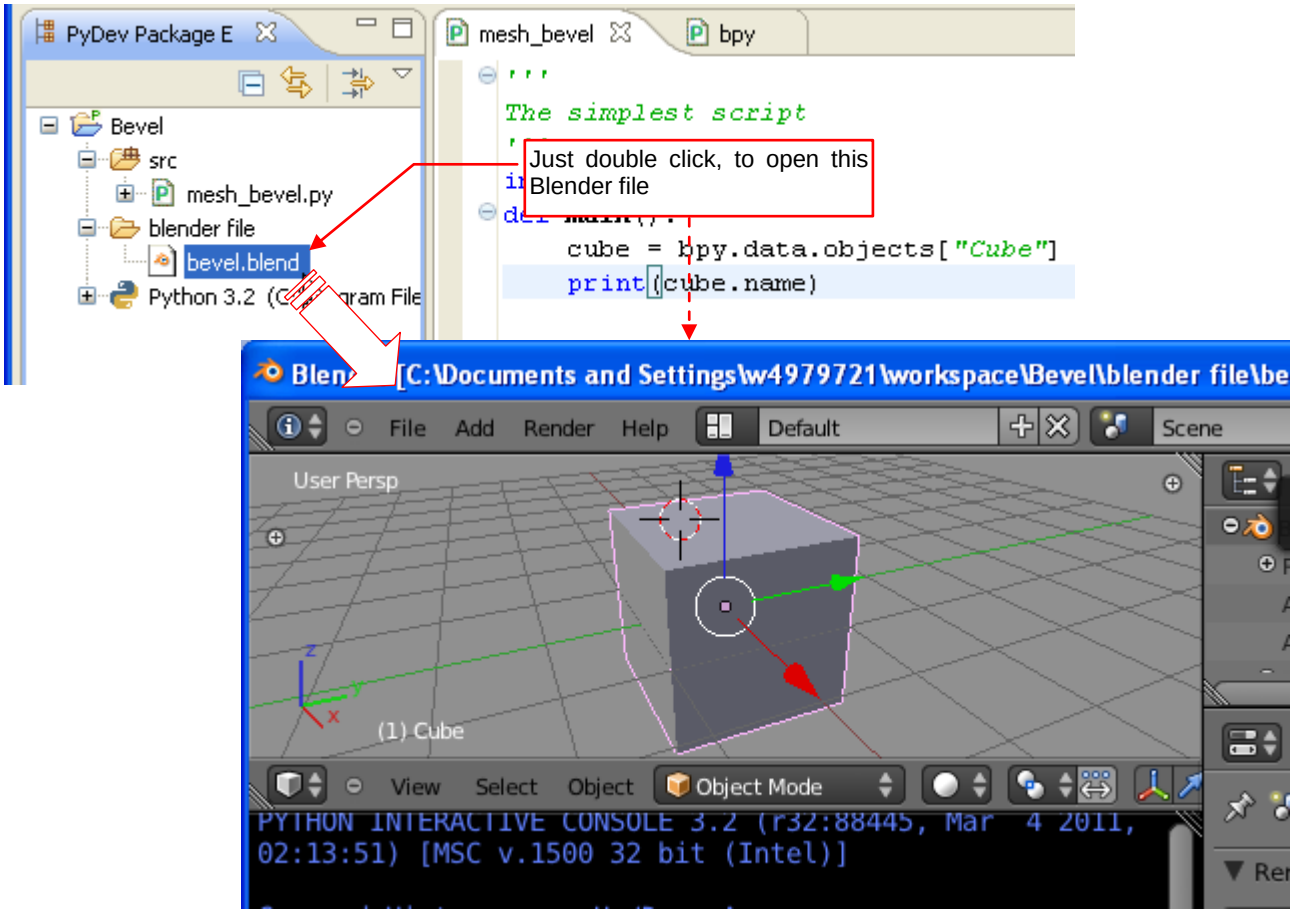


Figure 6.2.7 Opening the script "testbed"

- PyDev creates in the project folder copies of the imported files. This means that changes you will make to them will not affect their originals.

However, it is also possible to link to the project a file, which is located somewhere in another directory. That can happen when you want to work directly on an existing Blender add-on. They are placed in the Blender *scripts\addons* subfolder. To link it, start the *Import* operation in the same way as before (Figure 6.2.8):

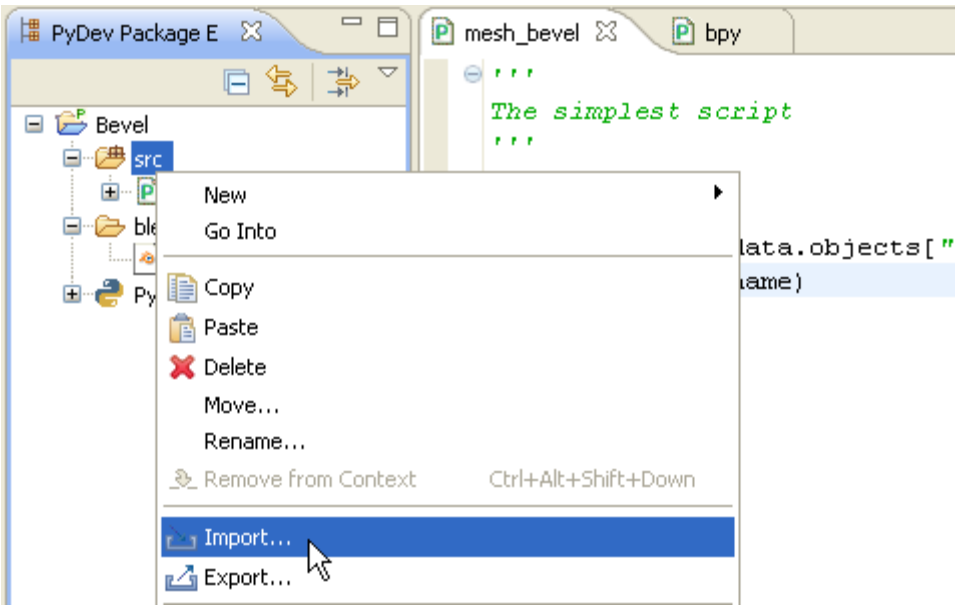


Figure 6.2.8 Linking an existing file — start with the *Import* command…

In the Import wizard window select the source folder and the file, you want to link (Figure 6.2.9):

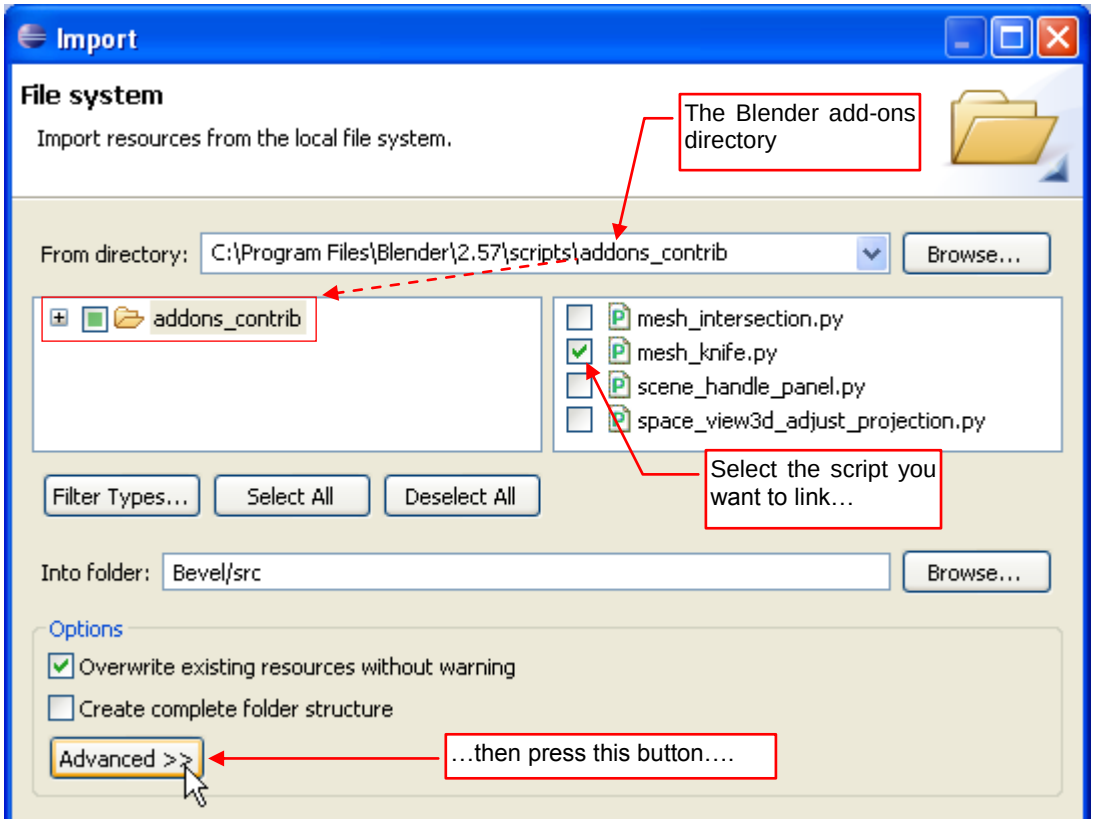


Figure 6.2.9 Linking another add-on script to the project

Then press the ***Advanced>>*** button, to display the additional wizard options (Figure 6.2.10):

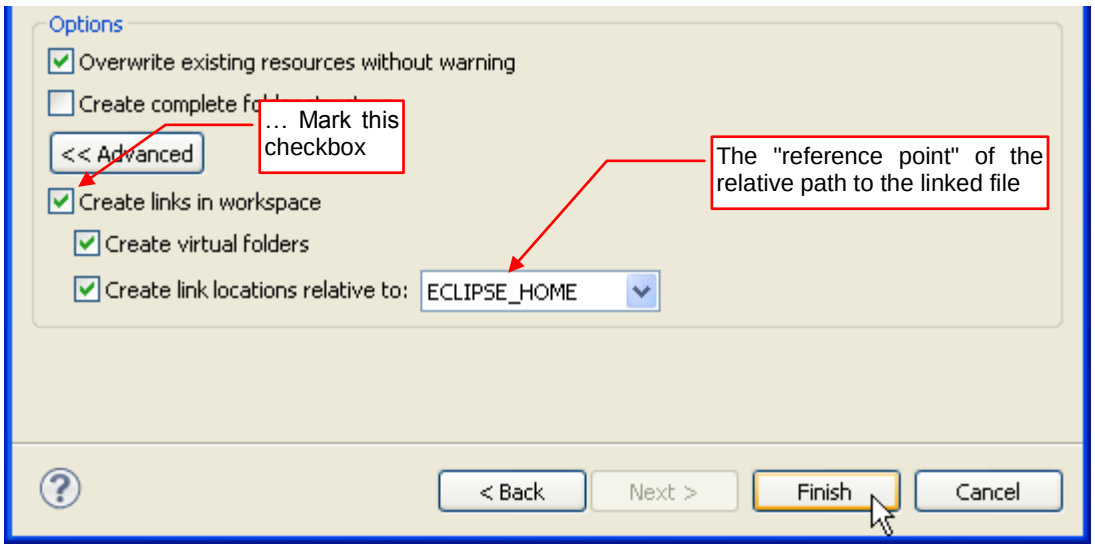


Figure 6.2.10 Selection of the linking options

Check the ***Create links in workspace*** option, first. This makes PyDev not create a local copy of the file in the project folder. Instead, it will create there a reference to the original script. This way you can easily change or reuse the code of plugins, located in the Blender directory.

PyDev also allows you to specify whether to store in the reference the full path to the specified file, or a relative path. It is controlled by the ***Create a link is relative locations*** option. I always use relative paths, because it is easier to move the project into another location with such a setting. PyDev can also determine the place in the directory structure, which will be the "reference point" of such a relative path. For the Blender plugins I would propose to use the *ECLIPSE_HOME* option. (Most likely, you have both Blender and Eclipse in the same *Program Files* directory).

When you press the ***Finish*** button, the PyDev will add the link to our project (Figure 6.2.11):

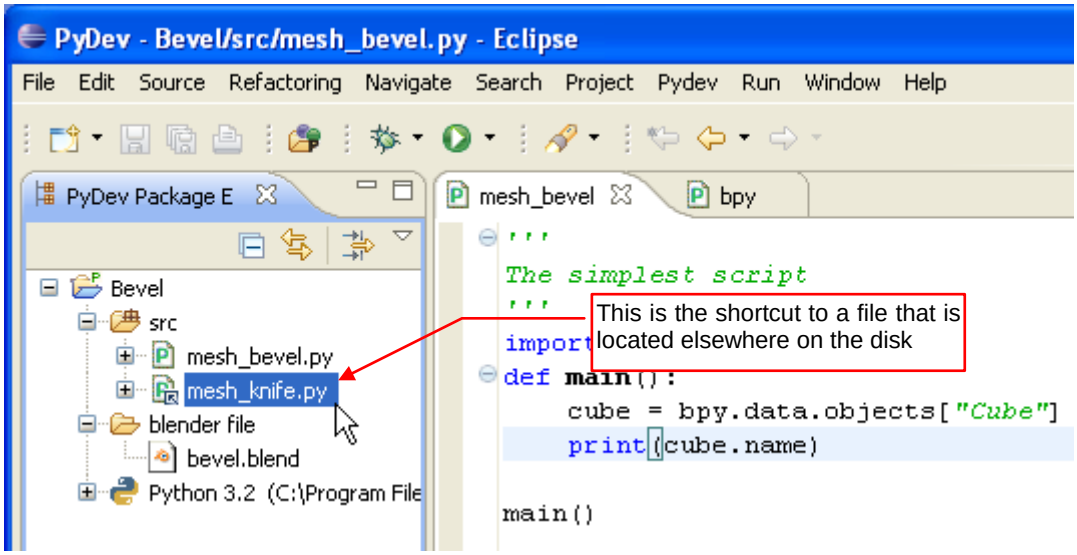


Figure 6.2.11 The source file, linked to this project

As you can see, the linked files are marked in Eclipse by an additional arrow in the lower right corner of their icon. When you look at the properties of this shortcut, you can read or change the position of the referenced file (Figure 6.2.12):

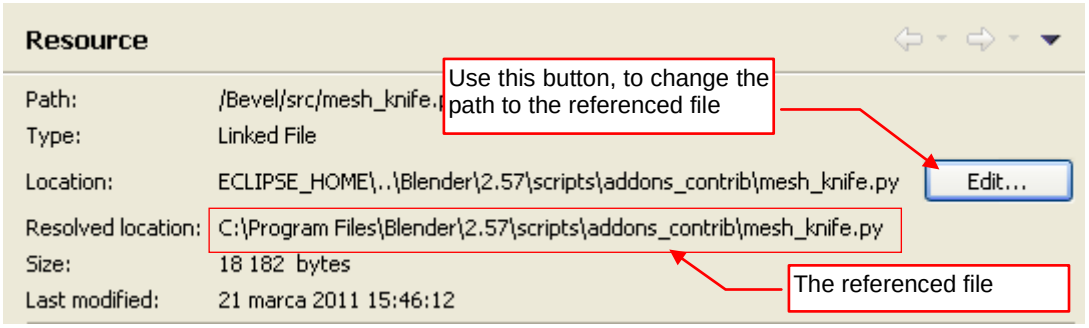


Figure 6.2.12 Properties of the linked file

6.3 Details of the Blender scripts debugging

Blender executes scripts using its own, embedded Python interpreter. You can debug them using the built-in, standard Python debugger. Unfortunately, this tool works in the "conversational" mode, in the console. Thus, it is not the "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 to keep track of the programs that are running on another computer (Figure 6.3.1):

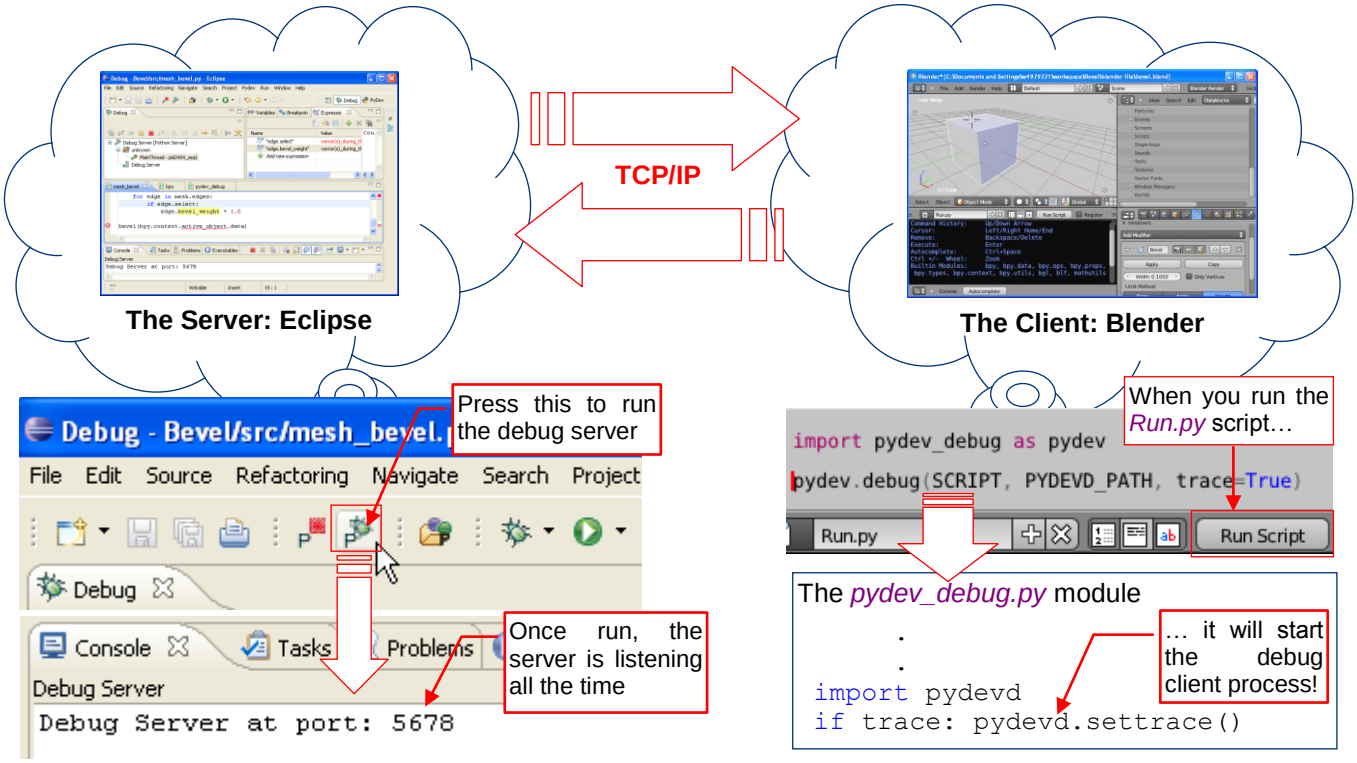


Figure 6.3.1 Tracing the Blender script execution: the use of the PyDev remote debugger

In the IDE (like Eclipse) you have to run the server process that starts "listening" to eventual requests from the debugged scripts. These requests will be sent from a remote debugger client, included in the code of the tracked script. In our case, this debugger client code is in the ***pydevd*** Python *package*. It is imported and initialized in the *pydev_debug.py* helper module (see page 129), which is used in the *Run.py* script template. (This is the code, which runs our script — see page 58). The communication between the remote debugger client and its server is realized through the network. Long ago, someone noticed that there are no obstacles to 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, when two persons are sitting in the same room and talking to each other via the phone. Fortunately, the programs are "stupid" and do not complain that they have to communicate in such a circuitous way. This solution works correctly, and it only counts.

Use the ***PyDev→Start Debug Server*** command, to start the server of its remote debugger (Figure 6.3.2):

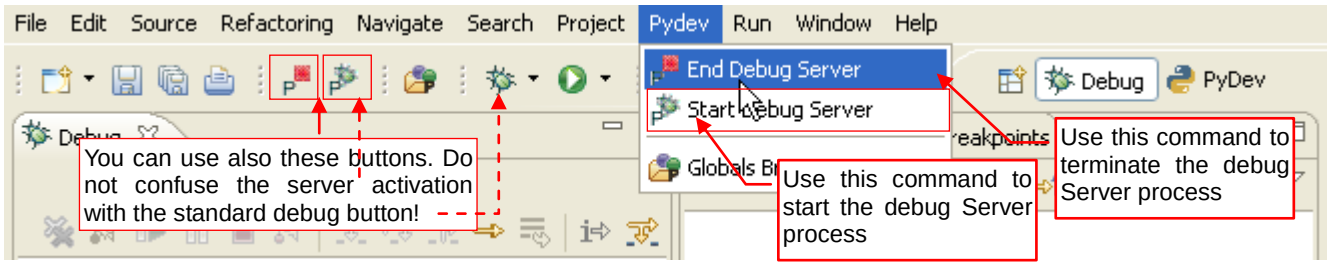


Figure 6.3.2 The commands that control the server of the PyDev remote debugger

What to do, when these PyDev commands do not appear[1] on the toolbar nor menu, as in Figure 6.3.2?

Sometimes the *Start/End Debug Server* commands can be just turned off in the *Debug* perspective! To enable them, use the ***Window→Customize Perspective*** command (Figure 6.3.3).

In the ***Customize Perspective*** window, open the ***Command Groups Availability*** tab (Figure 6.3.4). Find on the *Available command groups* list (on the left) a command group named ***PyDev Debug***. Just enable it, to let the *Start Debug Server* and *End Debug Server* commands appear on the toolbar and the menu!

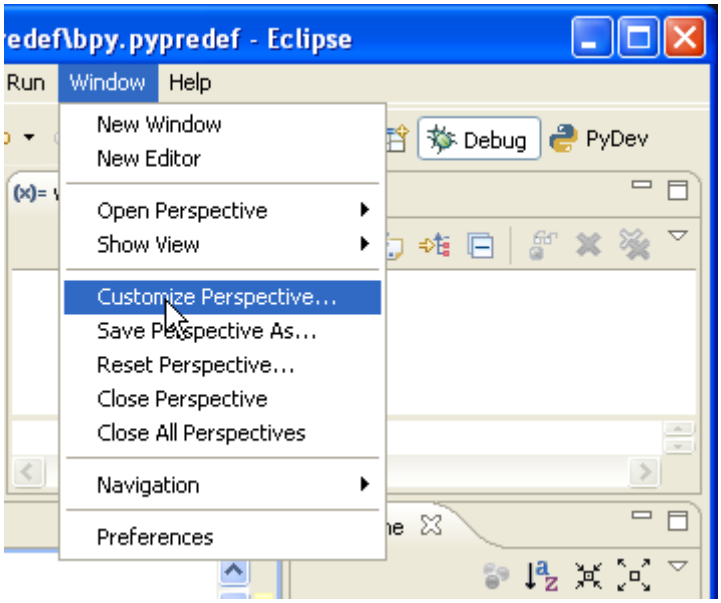


Figure 6.3.3 Opening the *Customize Perspective* window

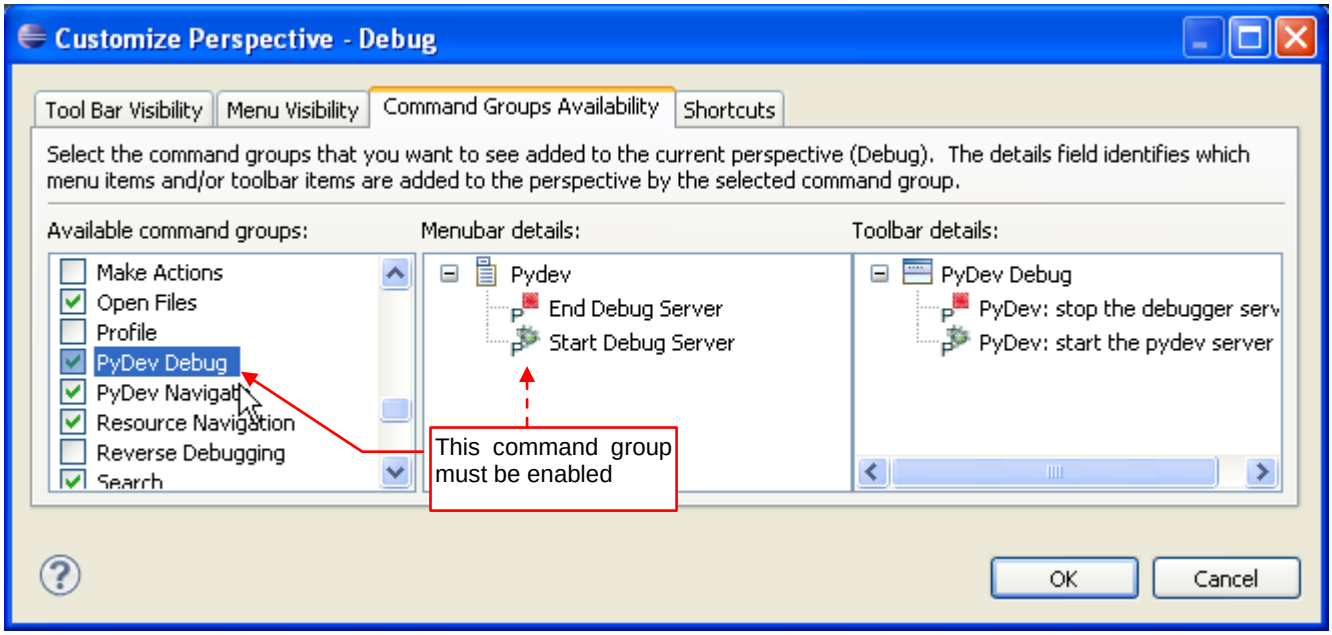


Figure 6.3.4 Enabling the PyDev remote debugger controls

When I made the Eclipse/PyDev installation for this book, the *Start* / *End Debug Server* commands were in the proper place. I did not have anything to fix in the configuration of the *Debug* perspective. I suppose that such a problem may be related to the way in which this perspective was added to the project.

- By the way, you have learned how to customize the 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.

While debugging the script, you will frequently check the current state of its variables. The PyDev provides the ***Variables*** pane for this purpose (Figure 6.3.5):

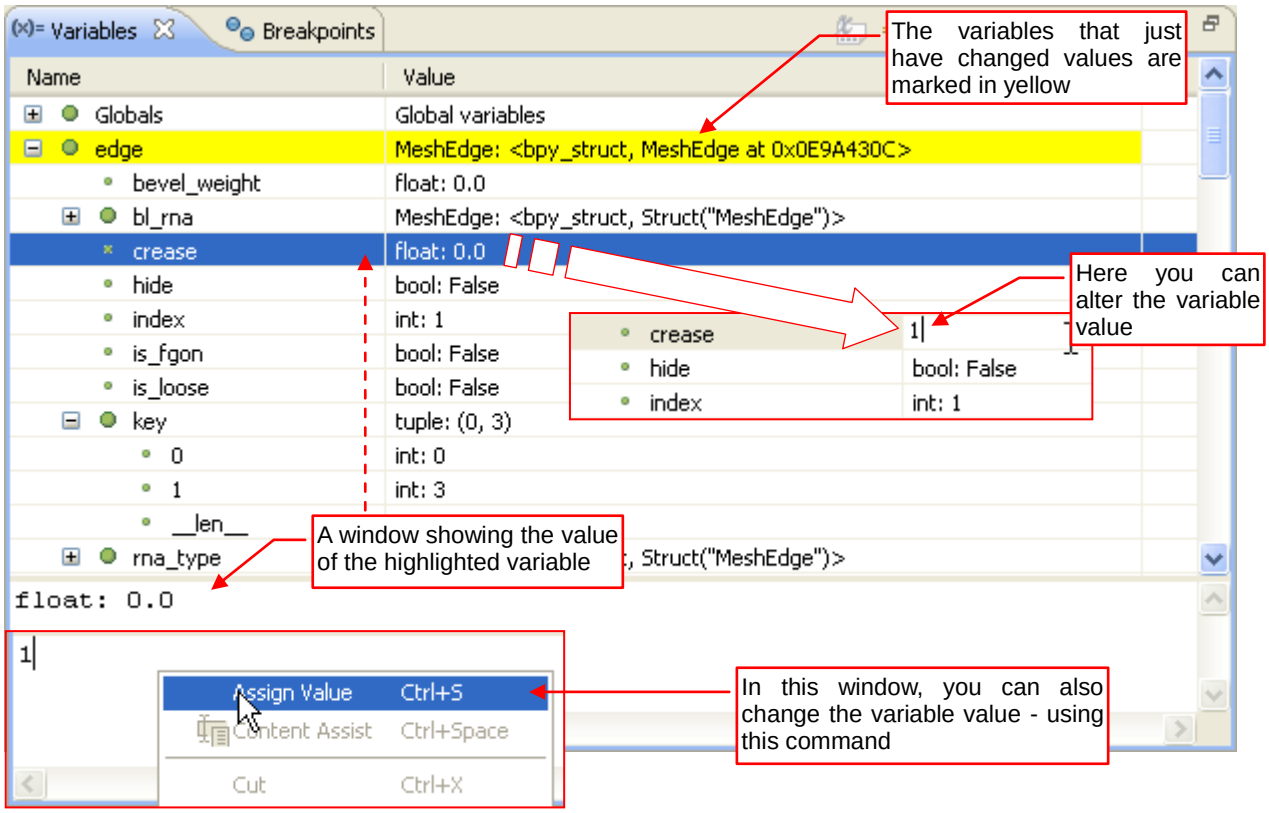


Figure 6.3.5 The *Variables* panel

The panel is divided into the list with names and values of global and local variables, and the detail area. In the detail area PyDev shows the value of the variable, which is highlighted on the list. I think that detail area is useful for checking longer string values. When the value of a variable is an object reference, Eclipse displays the **[+]** or **[-]** icon next to it. Click this icon to inspect the fields of this object.

In the *Variables* window you can also change the current variable values. Usually you will simply type them in the *Value* column (Figure 6.3.5). You can also change them in the detail area (using the ***Assign Value*** command from its context menu).

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 6.3.6):

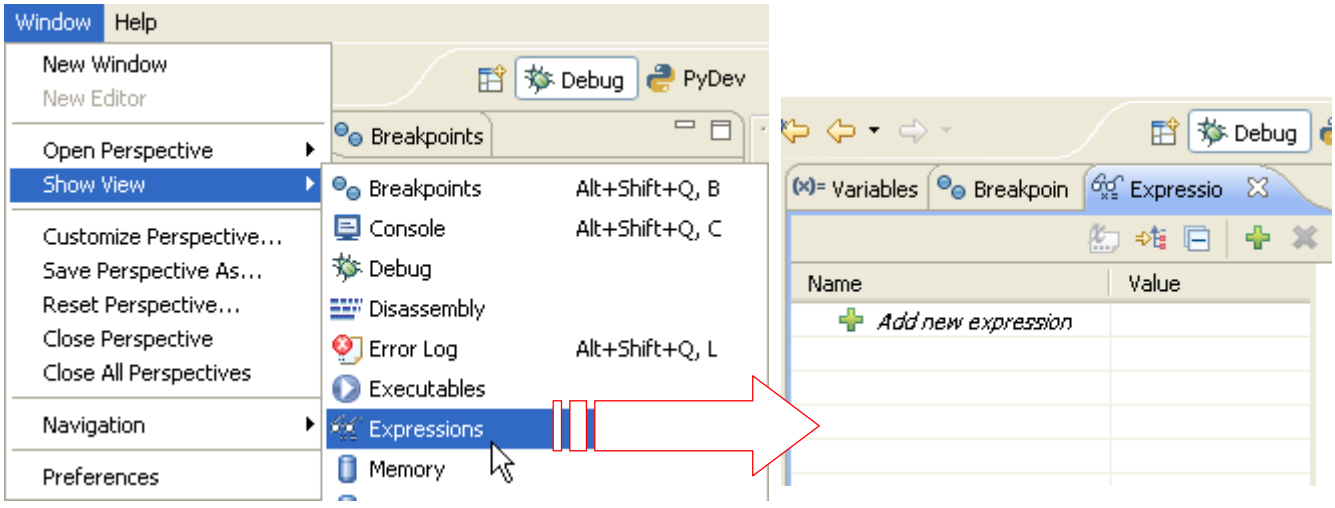


Figure 6.3.6 Adding the *Expressions* panel

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 an larger field the value of highlighted list item. Unlike in the *Variables* pane, here you can evaluate any Python expression, at every step of the script execution (Figure 6.3.7):

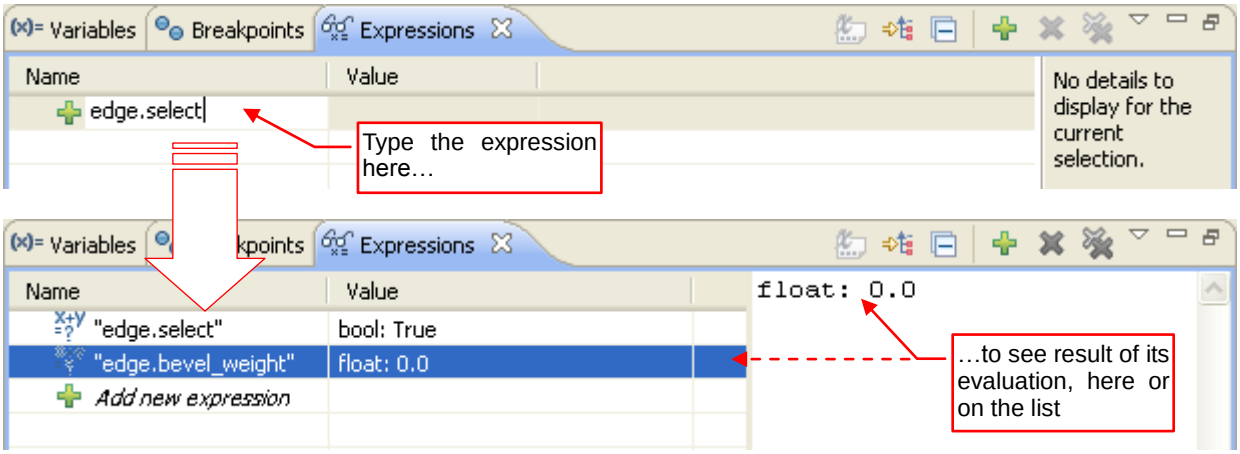


Figure 6.3.7 Adding new items to the *Expressions* list

In the *Expressions* pane you can simply enter the variable name. More often, however, it is used to track the selected fields of an object. You can also enter here a reference to a specific list item (eg, ***selected [0]***). In contrast to the *Variables* window, here you cannot change the expression result (the content of the *Value* column is read-only).

Another useful element for the script debugging is the Blender System Console. This is additional text window, beside the main window of the program. When you start Blender, the console appears first for a moment, and then the main window. Using the ***Help→Toggle System Console*** command you can control its visibility (Figure 6.3.8).

Blender *System Console* is the *standard output* of all scripts. (Do not confuse it with the Blender *Python Console*! There you see only the results of manually typed commands).
In the *System Console* you see all the texts printed by scripts (i.e. the results of the calls to the standard ***print()*** function). When an error occurs during the script execution, here you will find its detailed description.

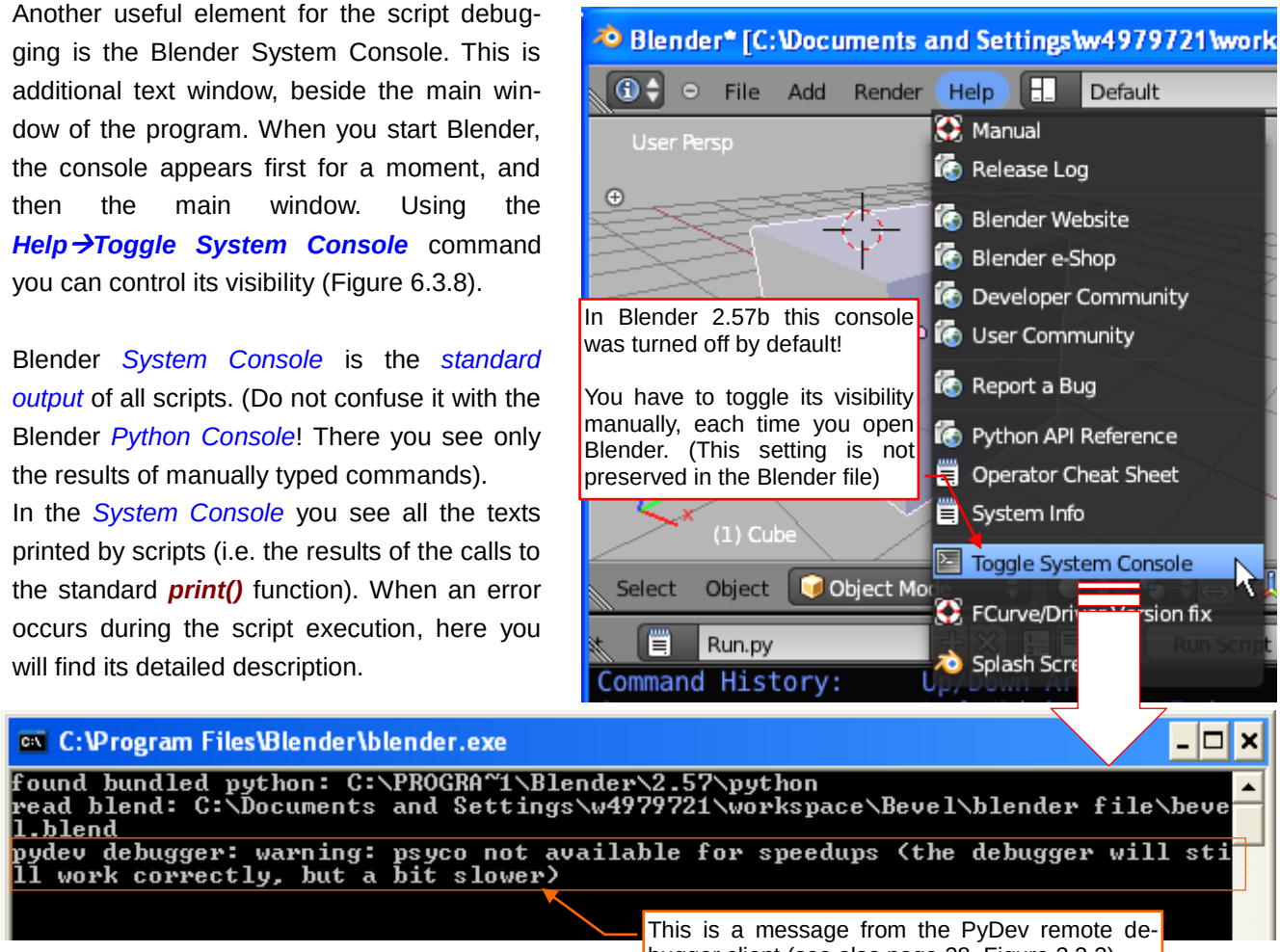


Figure 6.3.8 The Blender *System Console*

While debugging a script (i.e. when you trace its code in the Eclipse) Blender is "locked". In fact, it is patiently waiting for completion of the operation that you have started by pressing the ***Run Script*** button.

Still, if you enter an expression in the *Server Debug Console* in Eclipse, the Blender Python interpreter will evaluate it, and its result will be displayed in the Blender *System Console*. (Figure 6.3.9):

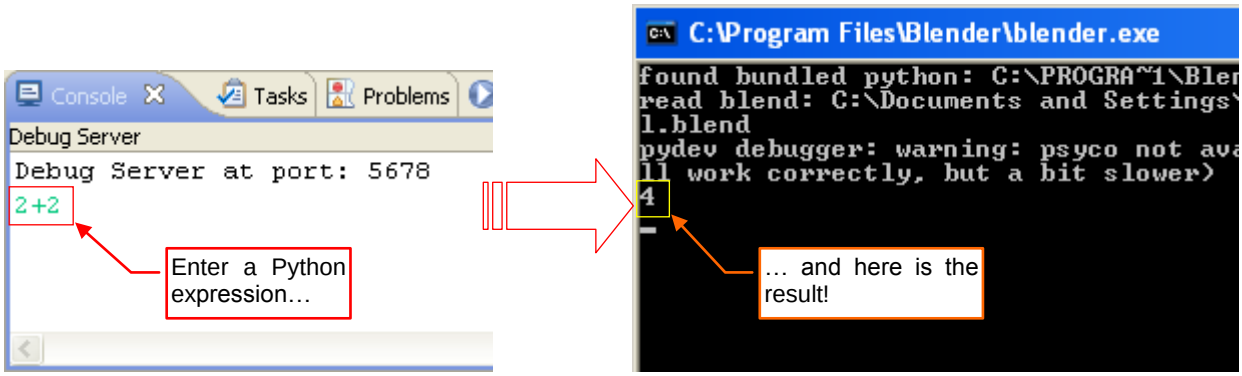


Figure 6.3.9 "Cooperation" of the Eclipse and Blender consoles during the debug session

You can treat it as an „ad hoc" method to check the values of various expressions - for example, a field of an object (Figure 6.3.10):

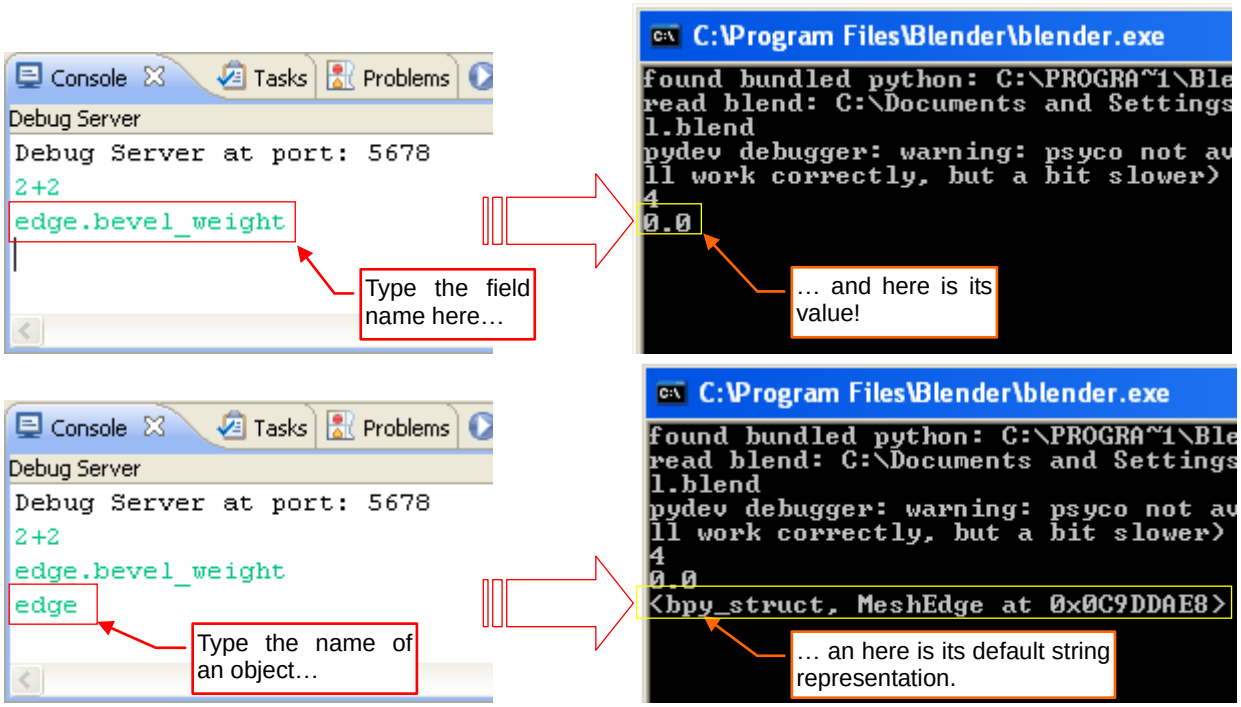


Figure 6.3.10 "Cooperation" of the Eclipse and Blender consoles — other examples

Of course, the same can be checked in the *Expressions* panel. On the other hand, in the server console you can more — for example, you can call a method of an object.

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

Actually, to track the script execution in the PyDev remote debugger you need just add two following lines to your code (Figure 6.4.1):

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

Figure 6.4.1 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 thing from a longer list of everything, which is needed or worth to do during such an initialization. Hence, these two lines were expanded to a procedure named ***debug()*** (Figure 6.4.2):

```
'''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()  #<-- debugger stops at the next statement

    #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***

Starting 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 6.4.2 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 only one procedure: ***debug()*** (Figure 6.4.2). This allowed for maximum simplification of the *Run.py* code — the script template, which has to be updated for each new project (see page 58).

- Place the *pydev.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 124);

Notice (Figure 6.4.2) that the ***debug()*** procedure loads the user's ***script*** module using the *import* statement. It allows for debugging Blender *add-ons*[1]. Before the import, my program attempts to handle the previously loaded module as the *add-on*, and 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, from the very beginning implement the required ***register()*** and ***unregister()*** methods. It will allow for properly handling of its Blender registration process, every time you will press the ***Run Script*** button (see page 60).

[1] Each Blender add-on implements at least one class that derives from the corresponding base classes: ***bpy.types.Operator***, ***bpy.types.Panel***, or ***bpy.types.Menu***. It also must contain two module methods: ***register()*** and ***unregister()***, that perform registration of these add-on classes for the use in Blender. When the plugin is loaded, Blender calls its ***register()*** method, and when it is turned off — it calls ***unregister()***.Then Blender itself creates, when it is needed, the instances of the registered add-on classes. (This is the typical application model for an event-driven environment: „don't call me, I will call you". For example, Windows handles its applications in the same way). That is why you have to put the breakpoints in your add-on code. When Blender creates an instance of the add-on class, and invokes one of the class methods, they will break execution of this script into the PyDev debugger.

6.5 The full code of the *mesh_bevel.py* add-on

In subsequent chapters of this book, I have created gradually the script code of *mesh_bevel.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 the "one piece". If you want to copy this text to the clipboard — beware of the tab spacing! They are all removed, when you copy this code 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 three parts. The first part is a header that contains the GPL information and the *import* statements (Figure 6.5.1):

```
# ##### 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 #####

'''
Bevel add-on
A substitute of the old, 'destructive' Bevel command from Blender 2.49
'''

#--- ### Header
bl_info = {
    "name": "Bevel",
    "author": "Witold Jaworski",
    "version": (1, 0, 0),
    "blender": (2, 5, 7),
    "api": 36147,
    "location": "View3D > Specials (W-key)",
    "category": "Mesh",
    "description": "Bevels selected edges",
    "warning": "",
    "wiki_url": "",
    "tracker_url": ""
    }

#--- ### Imports
import bpy
from bpy.utils import register_module, unregister_module
from bpy.props import FloatProperty
```

To be continued on the next page…

Figure 6.5.1 The *mesh_bevel.py* script, part 1 (the declaration)

The next part contains the ***bevel()*** procedure, which implements the core operation (Figure 6.5.2):

```
#--- ### Core operation
def bevel(obj, width):
    """Bevels selected edges of the mesh
       Arguments:
            @obj (Object): an object with a mesh.
                           It should have some edges selected
            @width (float):width of the bevel
       This function should be called in the Edit Mode, only!
    """
    #
    #edge = bpy.types.MeshEdge
    #obj = bpy.types.Object
    #bevel = bpy.types.BevelModifier

    bpy.ops.object.editmode_toggle() #switch into OBJECT mode
    #adding the Bevel modifier
    bpy.ops.object.modifier_add(type = 'BEVEL')
    bevel = obj.modifiers[-1] #the new modifier is always added at the end
    bevel.limit_method = 'WEIGHT'
    bevel.edge_weight_method = 'LARGEST'
    bevel.width = width
    #moving it up, to the first position on the modifier stack:
    while obj.modifiers[0] != bevel:
        bpy.ops.object.modifier_move_up(modifier = bevel.name)

    for edge in obj.data.edges:
        if edge.select:
            edge.bevel_weight = 1.0

    bpy.ops.object.modifier_apply(apply_as = 'DATA', modifier = bevel.name)

    #clean up after applying our modifier: remove bevel weights:
    for edge in obj.data.edges:
        if edge.select:
            edge.bevel_weight = 0.0

    bpy.ops.object.editmode_toggle() #switch back into EDIT_MESH mode
```

To be continued on the next page…

Figure 6.5.2 The *mesh_bevel.py* script, part 2 (main procedure)

The last part of this code contains the implementation of the ***Bevel*** operator and the add-on registration (Figure 6.5.3):

```
#--- ### Operator
class Bevel(bpy.types.Operator):
    ''' Bevels selected edges of the mesh'''
    bl_idname = "mesh.bevel"
    bl_label = "Bevel"
    bl_description = "Bevels selected edges"
    bl_options = {'REGISTER', 'UNDO'} #Set this options, if you want to update
    #                                  parameters of this operator interactively
    #                                  (in the Tools pane)
    #--- parameters
    width = FloatProperty(name="Width", description="Bevel width",
                          subtype = 'DISTANCE', default = 0.1, min = 0.0,
                                                step = 1, precision = 2)
    #--- other fields
    LAST_WIDTH_NAME = "mesh.bevel.last_width" #name of the custom scene property

    #--- Blender interface methods
    @classmethod
    def poll(cls,context):
        return (context.mode == 'EDIT_MESH')

    def invoke(self, context, event):
        #input validation: are there any edges selected?
        bpy.ops.object.editmode_toggle()
        selected = list(filter(lambda e: e.select, context.object.data.edges))
        bpy.ops.object.editmode_toggle()

        if len(selected) > 0:
            last_width = context.scene.get(self.LAST_WIDTH_NAME,None)
            if last_width:
                self.width = last_width
            return self.execute(context)
        else:
            self.report(type='ERROR', message="No edges selected")
            return {'CANCELLED'}

    def execute(self,context):
        bevel(context.object,self.width)
        context.scene[self.LAST_WIDTH_NAME] = self.width
        return {'FINISHED'}

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

#--- ### Register
def register():
    register_module(__name__)
    bpy.types.VIEW3D_MT_edit_mesh_specials.prepend(menu_draw)

def unregister():
    bpy.types.VIEW3D_MT_edit_mesh_specials.remove(menu_draw)
    unregister_module(__name__)

#--- ### Main code
if __name__ == '__main__':
    register()
```

Figure 6.5.3 the *mesh_bevel.py* script, part 3 (the add-on code)

Bibliography

Książki

[1] Thomas Larsson, *Code snippets.Introduction to Python scripting for Blender 2.5x*, free e-book, 2010.

[2] Guido van Rossum, *Python Tutorial*, (part of Python electronic documentation), 2011

Internet

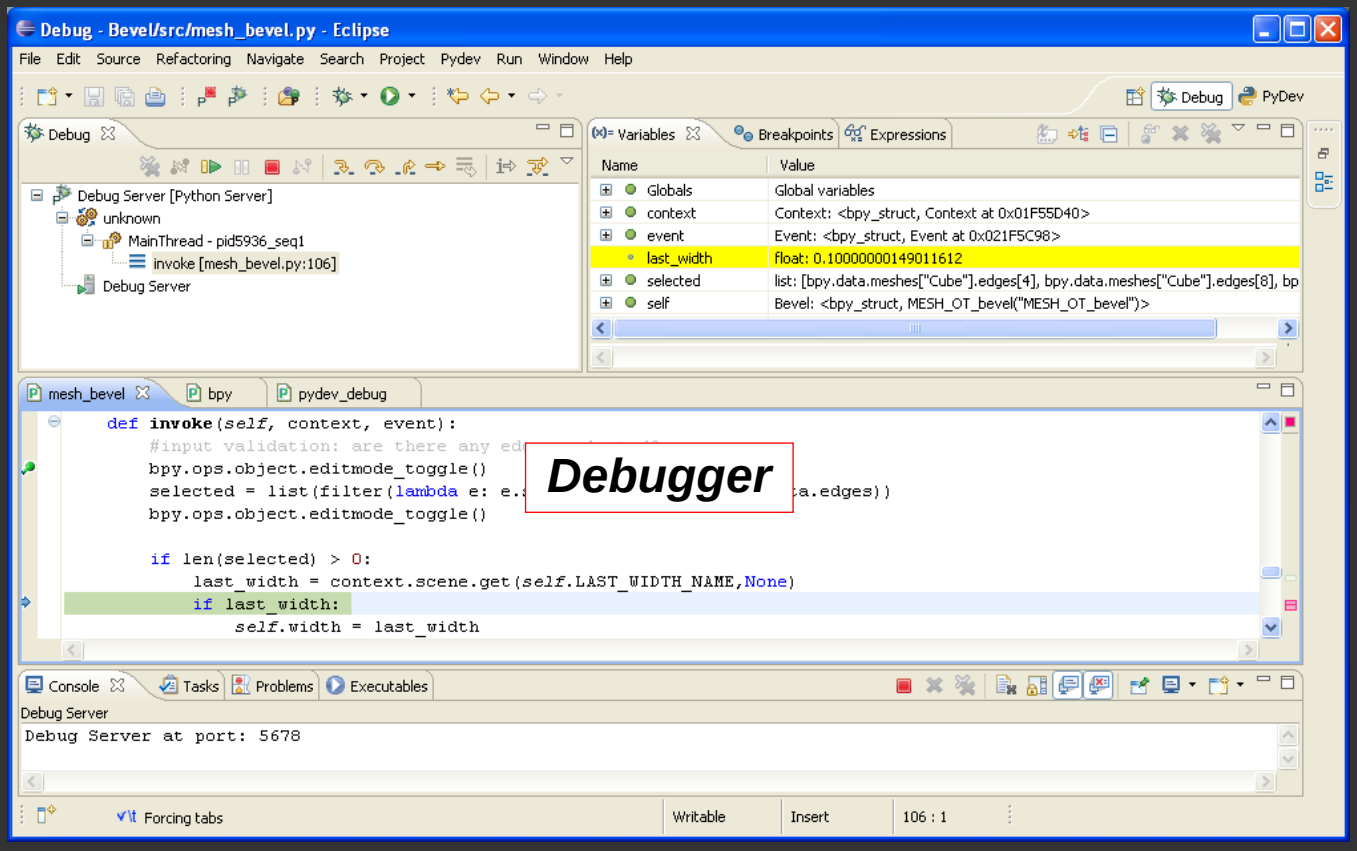
[1] http://www.blender.org

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

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

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

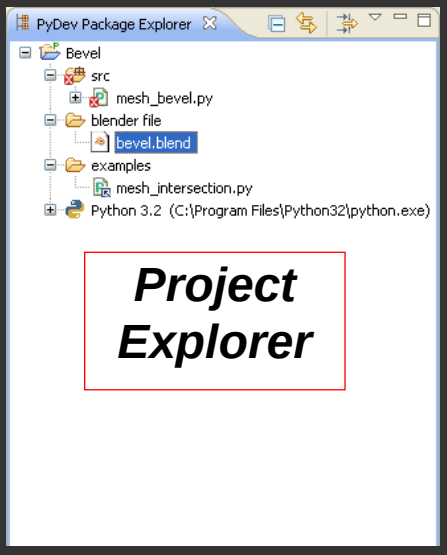
[5] http://wiki.blender.org, in particular http://wiki.blender.org/index.php/Extensions:Py/Scripts



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

I am showing in it, how to arrange a convenient development environment to write 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 in this way not only the tools, but also the methods that I use. This description will allow you to gain more skill needed to write your own scripts.



ISBN: 978-83-931754-2-0

Free electronic publication