Getting Started with PyCharm

What this tutorial is about

This tutorial aims to walk you step by step through creating, running and debugging a simple Python project, using PyCharm - the Python IDE with complete set of tools for productive development.

What this tutorial is not about

Python programming is out of scope of this tutorial. To learn more about the Python language, please refer to the official website.

Before you start

Make sure that:

- You are working with PyCharm version 2.7 or higher
- At least one Python interpreter, version from 2.4 to 3.3 is properly installed on your computer. You can download an interpreter from this page.

Downloading and installing PyCharm

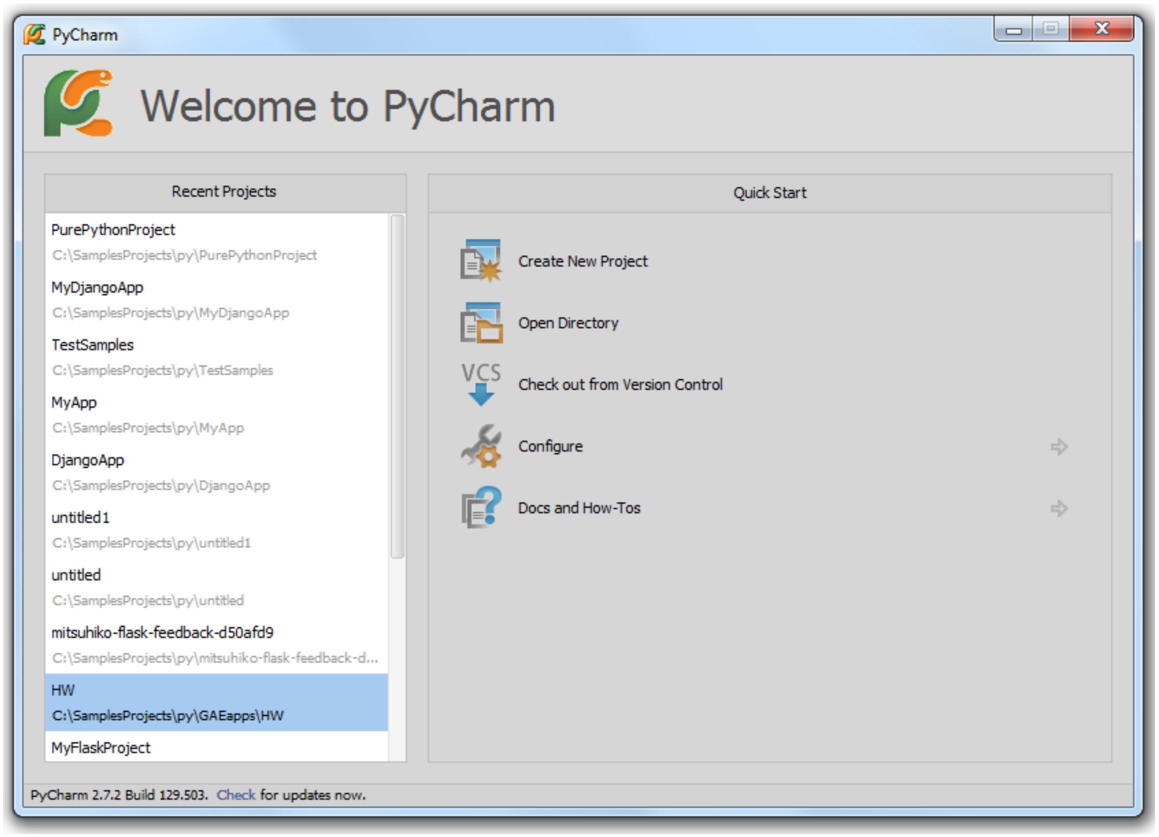
If you still do not have PyCharm, download it from this page. To install PyCharm, follow the instructions, depending on your platform.

Launching PyCharm

There are many ways to launch the IDE, depending on your OS.

- If you have a desktop shortcut icon , just double-click it.
- Under the bin directory of your PyCharm installation, double-click pycharm.exe or pycharm.bat (Windows), or pycharm.sh (MacOS and Linux).

PyCharm starts and shows the Welcome screen:


PyCharm

Welcome to PyCharm

Recent Projects

PurePythonProject
C:\SamplesProjects\py\PurePythonProject
MyDjangoApp
C:\SamplesProjects\py\MyDjangoApp
TestSamples
C:\SamplesProjects\py\TestSamples
MyApp
C:\SamplesProjects\py\MyApp
DjangoApp
C:\SamplesProjects\py\DjangoApp
untitled1
C:\SamplesProjects\py\untitled1
untitled
C:\SamplesProjects\py\untitled
mitsuhiko-flask-feedback-d50afd9
C:\SamplesProjects\py\mitsuhiko-flask-feedback-d...
HW
C:\SamplesProjects\py\GAEapps\HW
MyFlaskProject

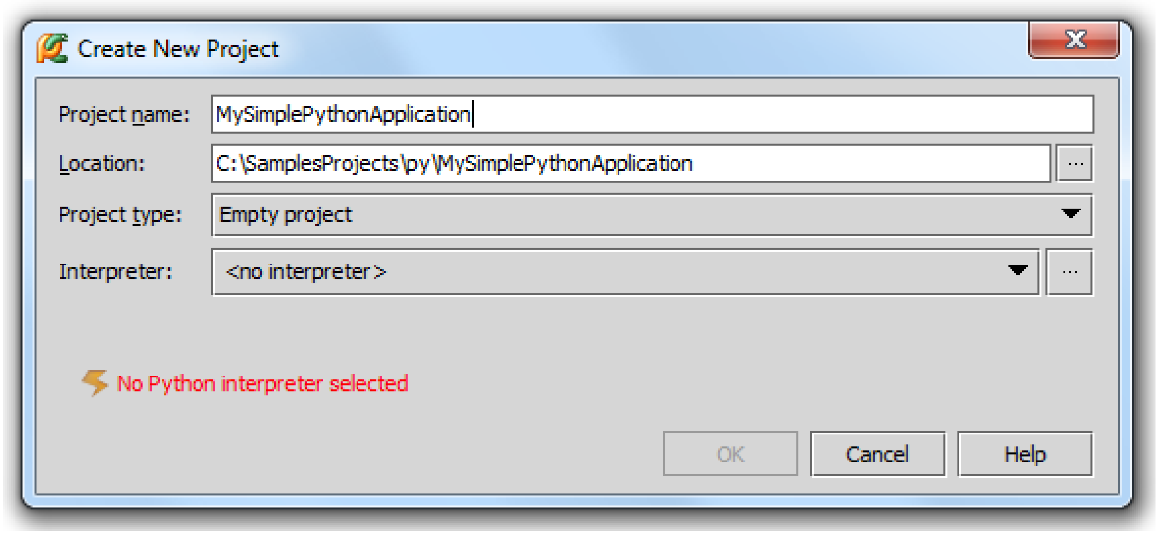
Quick Start

PyCharm 2.7.2 Build 129.503. Check for updates now.

Creating a simple Python project in PyCharm

To create a new project, click the link Create New Project. You see the Create New Project dialog box, where you have to define all the necessary settings for the new project.

Note that you can create a new project anytime... To do that, use File New Project on the main menu.


Create New Project

Project name: MySimplePythonApplication
Location: C:\SamplesProjects\py\MySimplePythonApplication
Project type: Empty project
Interpreter: <no interpreter>

No Python interpreter selected

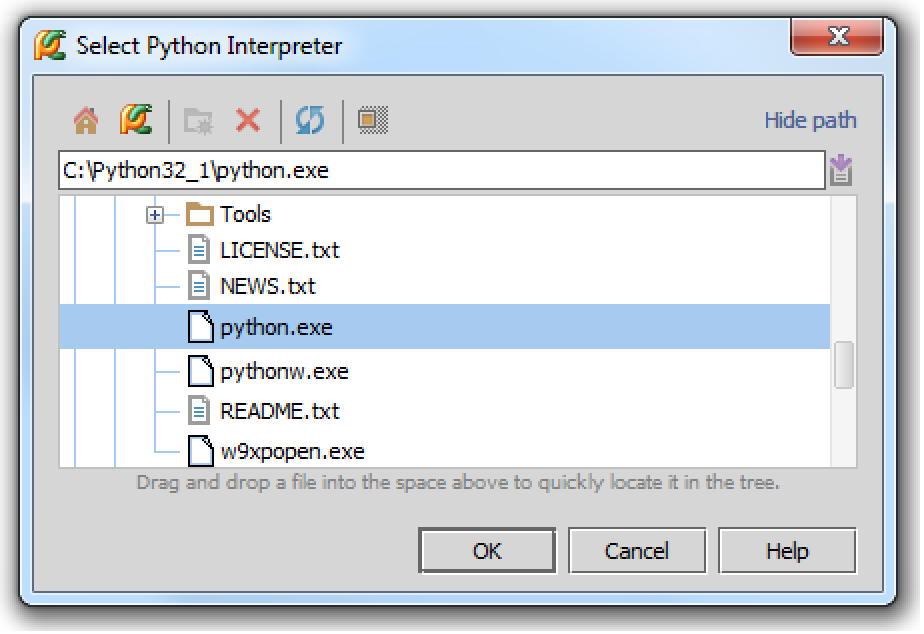
First, specify the project name - let it be MySimplePythonApplication. Note that PyCharm suggests project location by default. You can either accept the default location, or click the browse button and find some suitable place of your choice.

Next, select the project type. PyCharm suggests several project templates for the development of the various types of applications (Django, Google AppEngine, etc.). When PyCharm creates a new project from a project template, it produces the corresponding directory structure and specific files.

However, our task here is to create a project for Python. In this case let's select the type *Empty project* - it is most suitable for plain Python programming. In this case PyCharm will not produce any special files or directories.

And finally, let's choose a Python interpreter. As you see, PyCharm informs you that Python interpreter is not yet selected. Since you have at least one Python interpreter at your disposal, let's define it as the project interpreter.

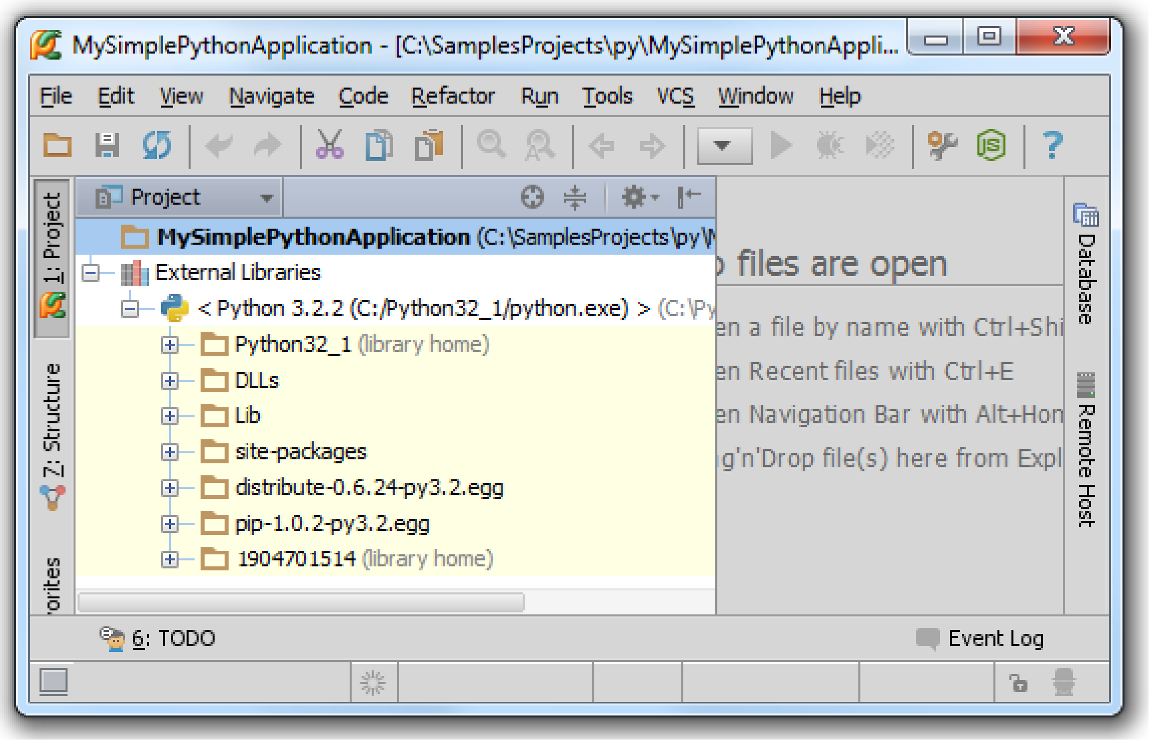
To do that, click the browse button next to the *Interpreter* field. In the Python Interpreters dialog box, click + , choose *Local...*, and then select the desired interpreter from your file system:



When all the necessary settings are done, OK button becomes enabled - so click it and get your project ready.

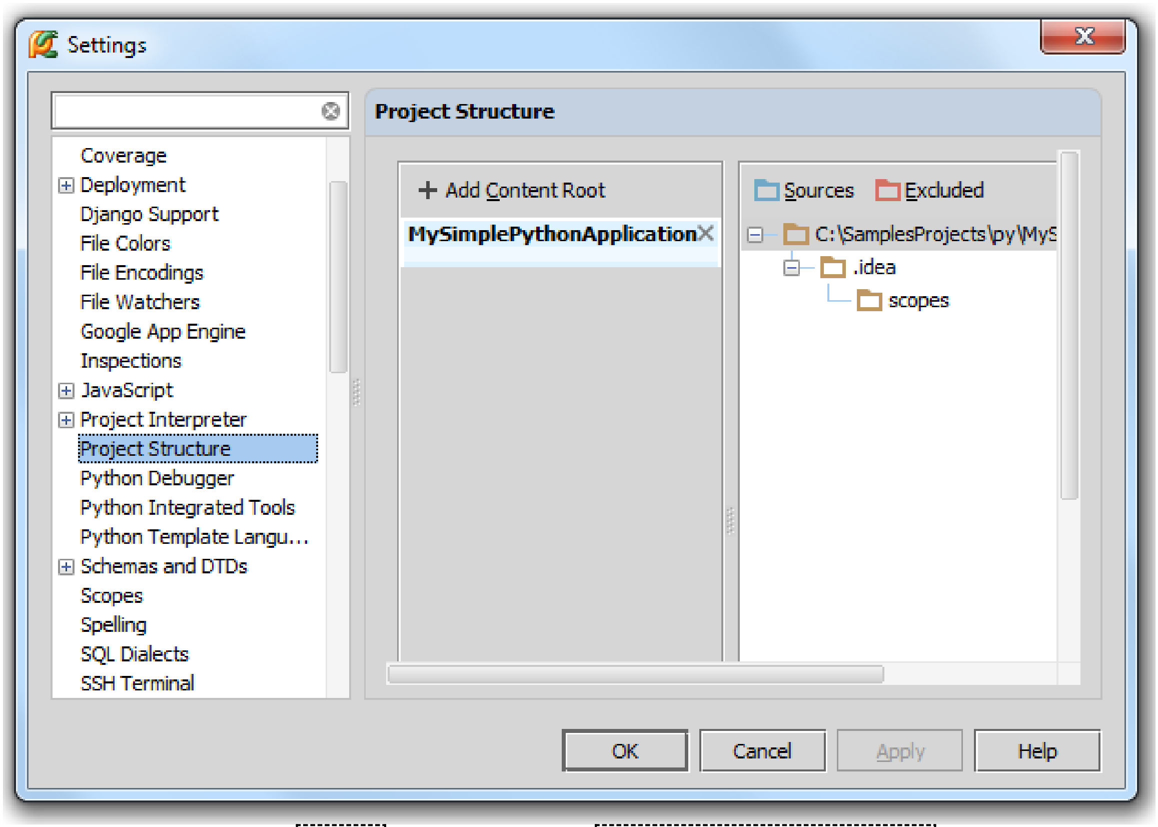
Exploring and configuring project structure

You can view the initial project structure in the Project tool window:


MySimplePythonApplication - [C:\SamplesProjects\py\MySimplePythonAppli...
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Project
MySimplePythonApplication (C:\SamplesProjects\py\M
External Libraries
< Python 3.2.2 (C:/Python32_1/python.exe) > (C:\Py
Python32_1 (library home)
DLLs
Lib
site-packages
distribute-0.6.24-py3.2.egg
pip-1.0.2-py3.2.egg
1904701514 (library home)
files are open
en a file by name with Ctrl+Shi
en Recent files with Ctrl+E
en Navigation Bar with Alt+Hon
g'n'Drop file(s) here from Expl
1: Project
7: Structure
orites
Database
Remote Host
6: TODO
Event Log

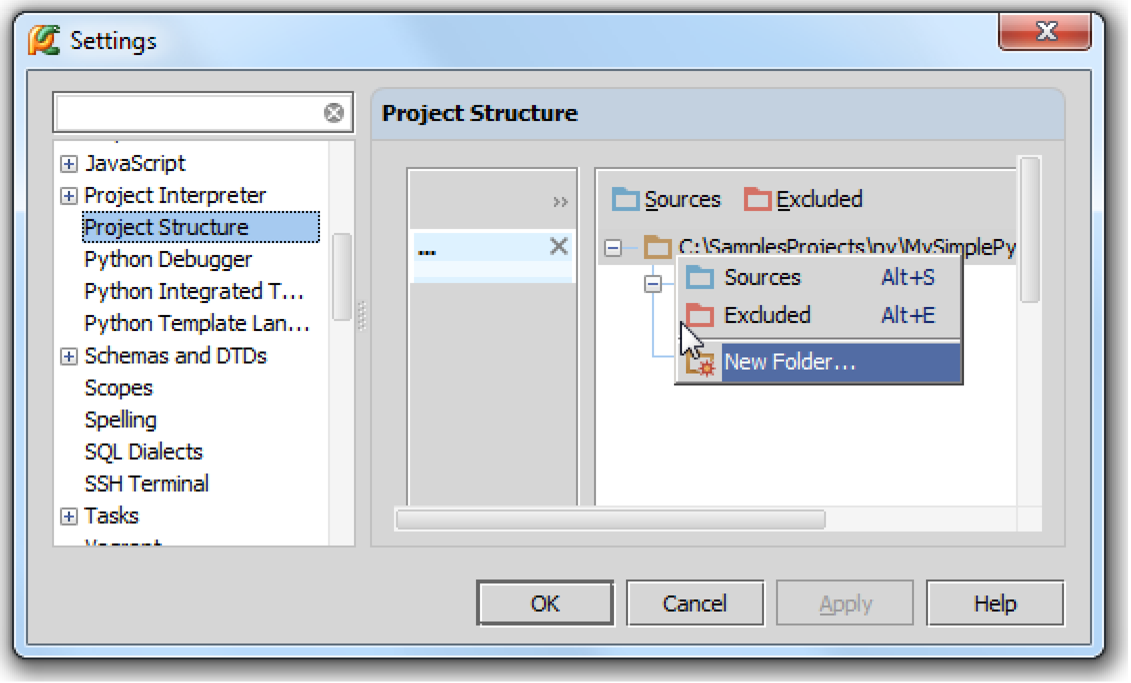
As you see, the project contains just the project root, and the Python interpreter you've specified under the External Libraries node.

Next, let's explore and configure the project structure in more detail: click on the main toolbar, and then select the Project Structure page:

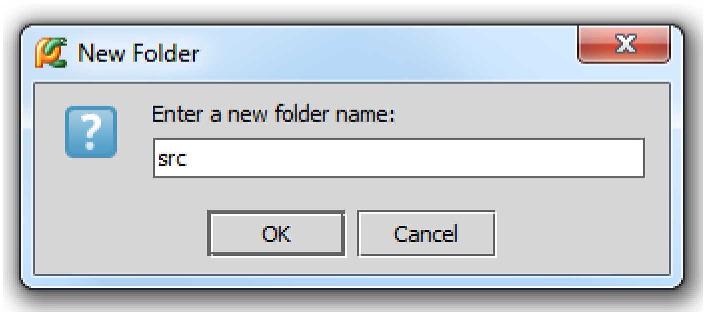


Under the project root you see .idea directory - it contains MySimplePythonApplication.iml file that reflects the project structure, and several XML files, each one responsible for its own set of settings, which can be recognized by their names: encodings.xml, vcs.xml etc. Note that .idea directory is not visible in the Project tool window.

Next let's add a source root, where all the work will actually be performed. In the same Project Structure page, right-click the project root, and choose New Folder on the context menu:



Next, type the directory name:



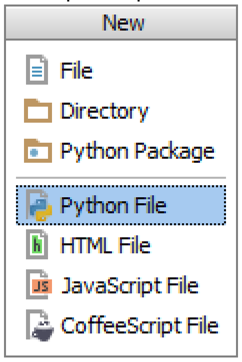
Finally, let's mark this directory as the source root: select src directory, and click - you see that src directory is now marked with icon.

Click OK to apply changes and close the Settings/Preferences dialog.

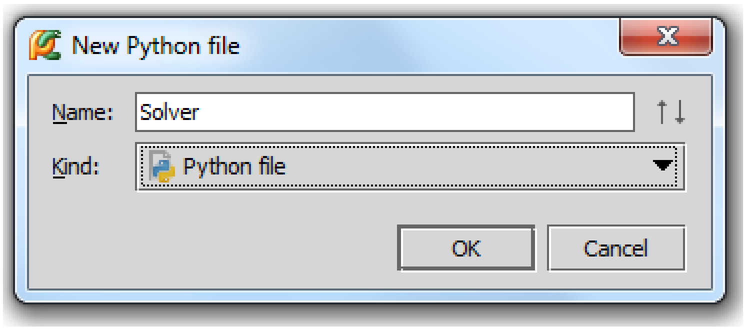
Note that actually this step is optional. You can just create a file under the project root, and it will be perceived as the source, since by default the project root is the source root.

Creating a Python class

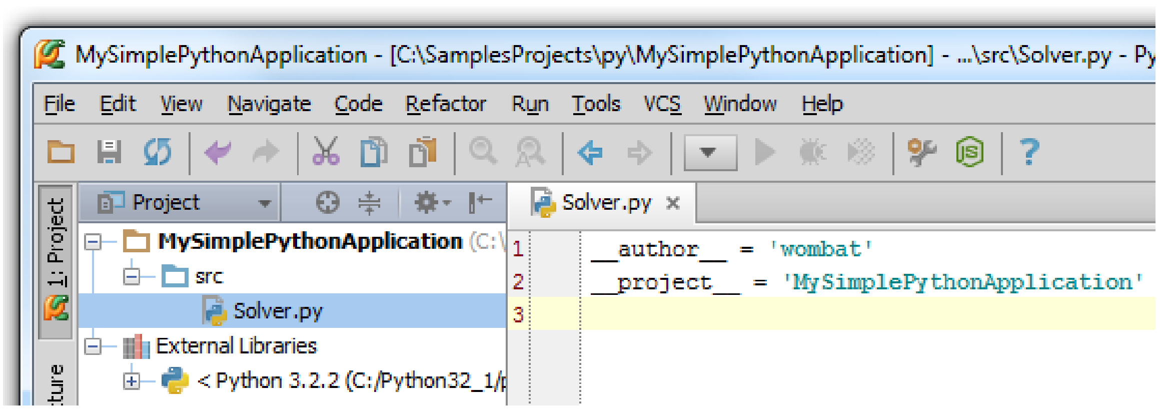
Select src directory in the project tool window, and press Alt+Insert:



Choose the option Python file from the pop-up window, and then type the new file name (Solver):



PyCharm creates a new Python file and opens it for editing:


MySimplePythonApplication - [C:\SamplesProjects\py\MySimplePythonApplication] - ...\src\Solver.py - Py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
1: Project
Project
MySimplePythonApplication (C:\
src
Solver.py
External Libraries
< Python 3.2.2 (C:/Python32_1/
Solver.py

```
__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'
```

Editing source code

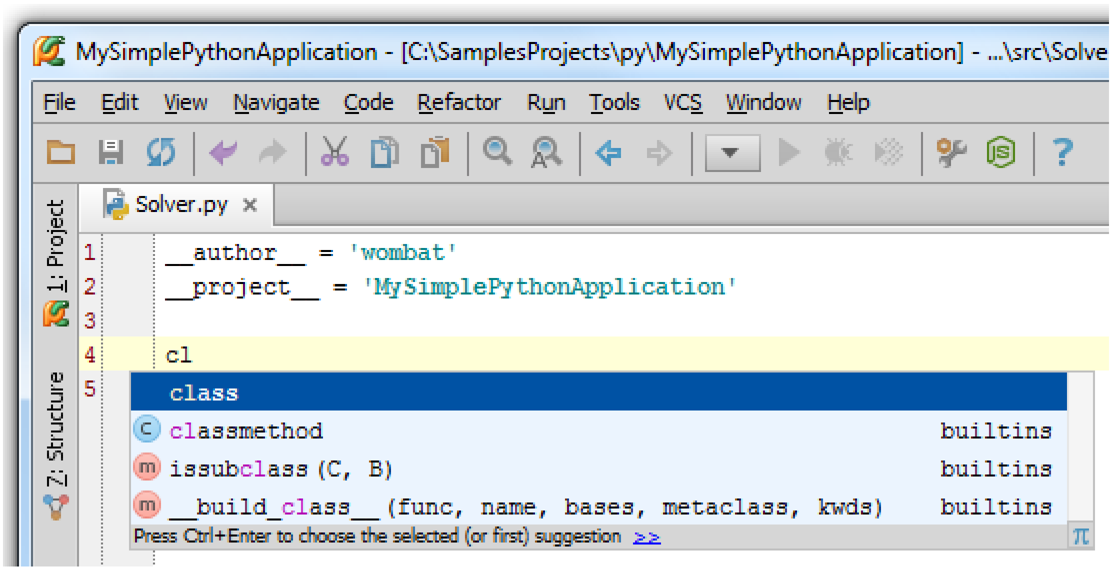
Let us first have a look at the Python file we've just generated. The stub contains just two lines:

```
_author_ = 'wombat'
_project_ = 'MySimplePythonApplication'
```

Since a Python file is produced by file template, PyCharm has substituted actual values instead of the pre-defined variables $PROJECT_NAME and $USER.

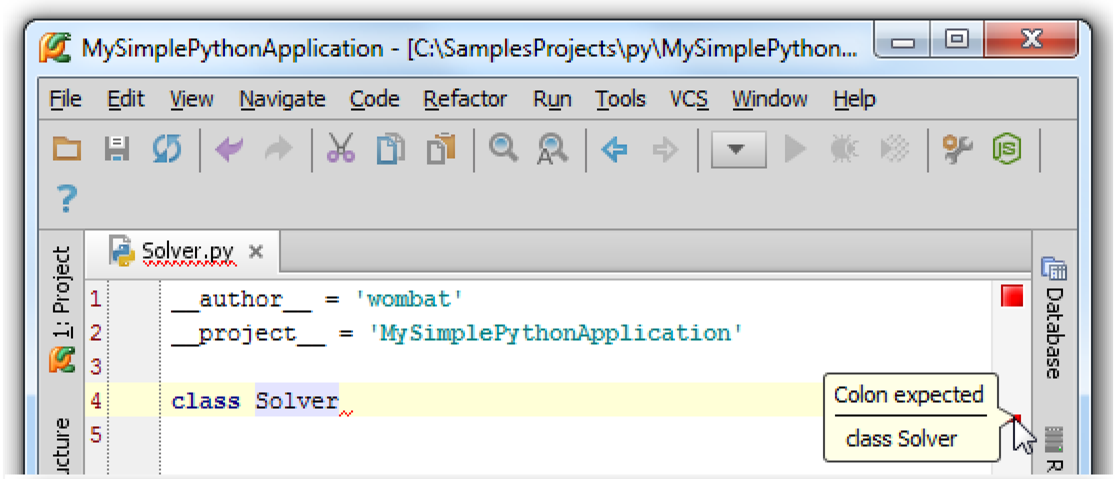
Now let us proceed with creating some meaningful contents - a simple application to solve a quadratic equation.

Immediately as you start typing, you understand that PyCharm, like a pair-programmer, looks over your shoulder and suggests proper choices. For example, you want to create a Python class. As you just start typing the keyword, a suggestion list appears:

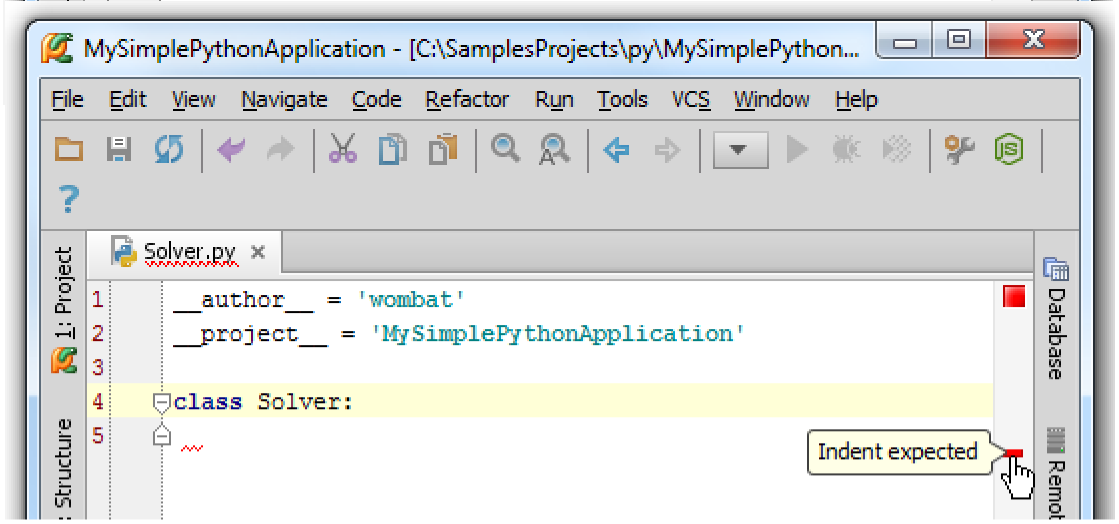

MySimplePythonApplication - [C:\SamplesProjects\py\MySimplePythonApplication] - ...\src\Solve
File Edit View Navigate Code Refactor Run Tools VCS Window Help
1: Project
7: Structure
Solver.py
class
classmethod builtins
issubclass (C, B) builtins
__build_class__ (func, name, bases, metaclass, kwds) builtins
Press Ctrl+Enter to choose the selected (or first) suggestion >>

```
__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

cl
```

Choose the keyword class and type the class name (Solver). PyCharm immediately informs you about the missing colon, then expected indentation:


MySimplePythonApplication - [C:\SamplesProjects\py\MySimplePython...
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Solver.py
Colon expected
class Solver

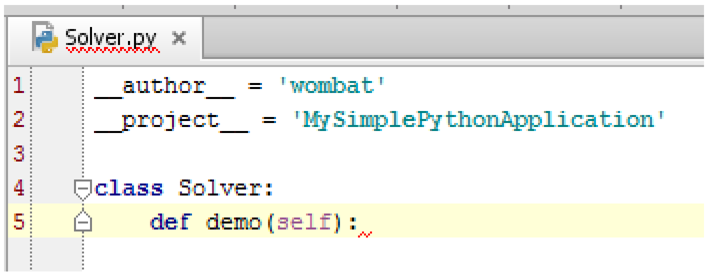
```
__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver
```


MySimplePythonApplication - [C:\SamplesProjects\py\MySimplePython...
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Solver.py
Indent expected

```
__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver:

```

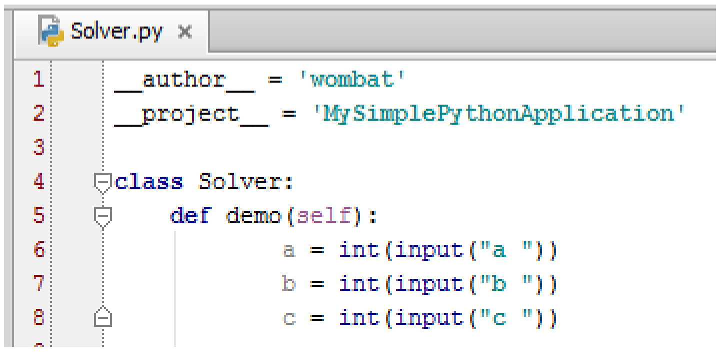
Note the error stripes in the right gutter. Hover your mouse pointer over an error stripe, and PyCharm shows a balloon with the detailed explanation. Since PyCharm analyses your code on-the-fly, the results are immediately shown in the inspection indicator on top of the right gutter. This inspection indication works like a traffic light: when it is green, everything is OK, and you can go on with your code; a yellow light means some minor problems that however will not affect compilation; but when the light is red, it means that you have some serious errors.

Let's continue creating the function 'demo': when you just type the opening brace, PyCharm creates the entire code construct (mandatory parameter 'self', closing brace and colon), and provides proper indentation:



```
__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver:
    def demo(self):
```

Note as you type, that unused symbols are shown greyed out:

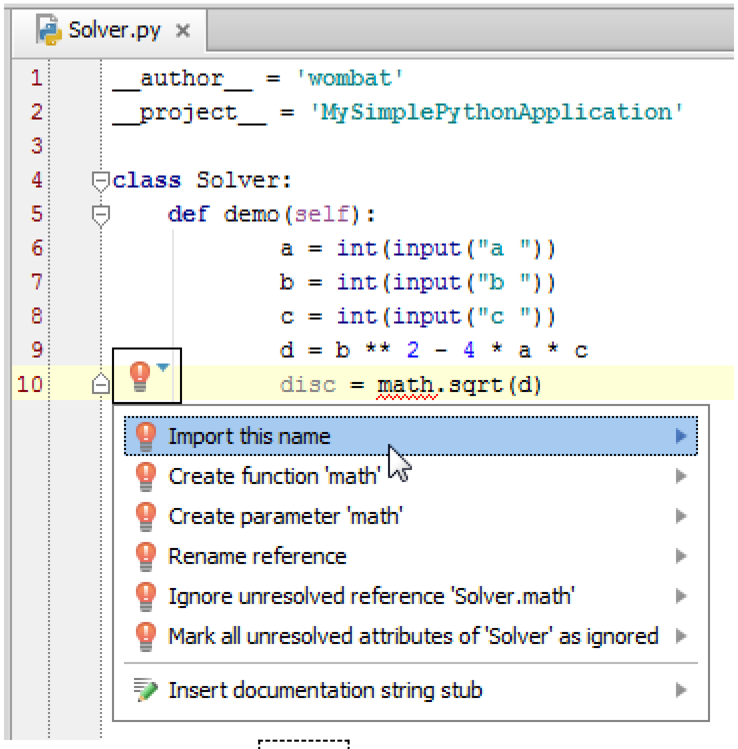


```python
__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver:
    def demo(self):
        a = int(input("a "))
        b = int(input("b "))
        c = int(input("c "))
```

As soon as you calculate a discriminant, they are rendered as usual. Next, pay attention to the unresolved reference 'math'. PyCharm underlines it with the red curvy line, and shows the red bulb.

Let's make a brief excursus into PyCharm's notion of intention actions and quick fixes. When you write your code, it is sometimes advisable to modify code constructs - in this case PyCharm shows a yellow light bulb. However, if PyCharm encounters an error, it shows the red light bulb.

In either case, to see what does PyCharm suggest you to do, press Alt+Enter - this will display the suggestion list, which in our case contains several possible solutions:


Solver.py
Import this name
Create function 'math'
Create parameter 'math'
Rename reference
Ignore unresolved reference 'Solver.math'
Mark all unresolved attributes of 'Solver' as ignored
Insert documentation string stub

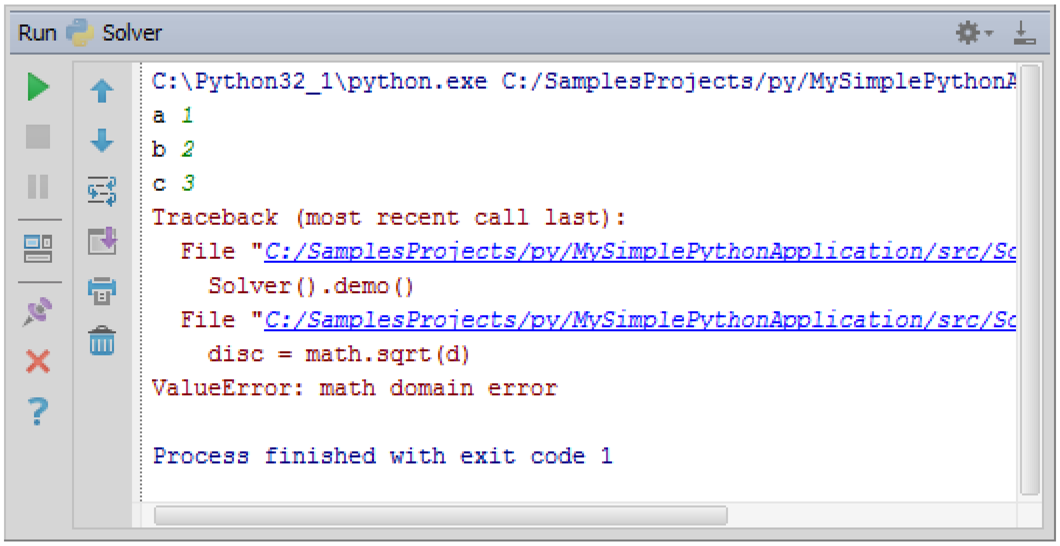
```python
__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver:
    def demo(self):
        a = int(input("a "))
        b = int(input("b "))
        c = int(input("c "))
        d = b ** 2 - 4 * a * c
        disc = math.sqrt(d)
```

Let's choose importing the math library. Im demo ent is added Solver er.py file. Next, calculate roots of the quadratic equation, and print them out, and finally, let's call the function demo of the class Solver:

```
import math

_author_ = 'wombat'
_project_ = 'MySimplePythonApplication'

class Solver:
 def demo(self):

   a = int(input("a "))
   b = int(input("b "))
   c = int(input("c "))
   d = b ** 2 - 4 * a * c
   disc = math.sqrt(d)
   root1 = (-b + disc) / (2 * a)
   root2 = (-b - disc) / (2 * a)
   print(root1, root2)
Solver().demo()
```

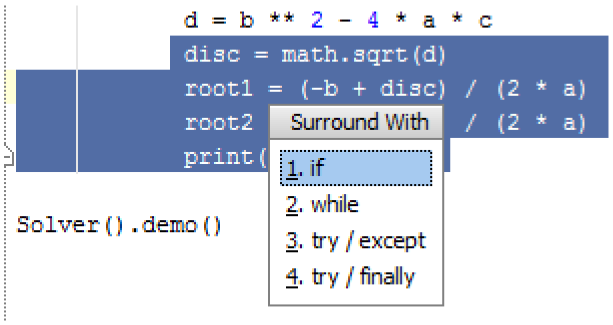
Then press Ctrl+Shift+F10 to run the script. A console appears in the Run tool window. In this console, you have to enter the a,b and c values, and expect to see a result.

Oops... PyCharm reports a run-time error:



```
C:\Python32_1\python.exe C:/SamplesProjects/py/MySimplePythonA
a 1
b 2
c 3
Traceback (most recent call last):
  File "C:/SamplesProjects/py/MySimplePythonApplication/src/Sc
    Solver().demo()
  File "C:/SamplesProjects/py/MySimplePythonApplication/src/Sc
    disc = math.sqrt(d)
ValueError: math domain error

Process finished with exit code 1
```

It seems that some analysis is advisable, so let's make sure that the radicand 'd' is non-negative, and report an error, when it is negative. To do that, select the discriminant calculation statements, and then press Ctrl+Alt+T (CodeSurround with):



```
        d = b ** 2 - 4 * a * c
        disc = math.sqrt(d)
        root1 = (-b + disc) / (2 * a)
        root2 = (-b - disc) / (2 * a)
        print(

Solver().demo()
```

PyCharm creates a stub 'if' construct, leaving you with the task of filling it with the proper contents. Finally, it would be n the whole calc eated more than once, so let's use the 'Surround with' action again: select the entire body of the function demo and surround it with while. You'll end up with the code like the following:

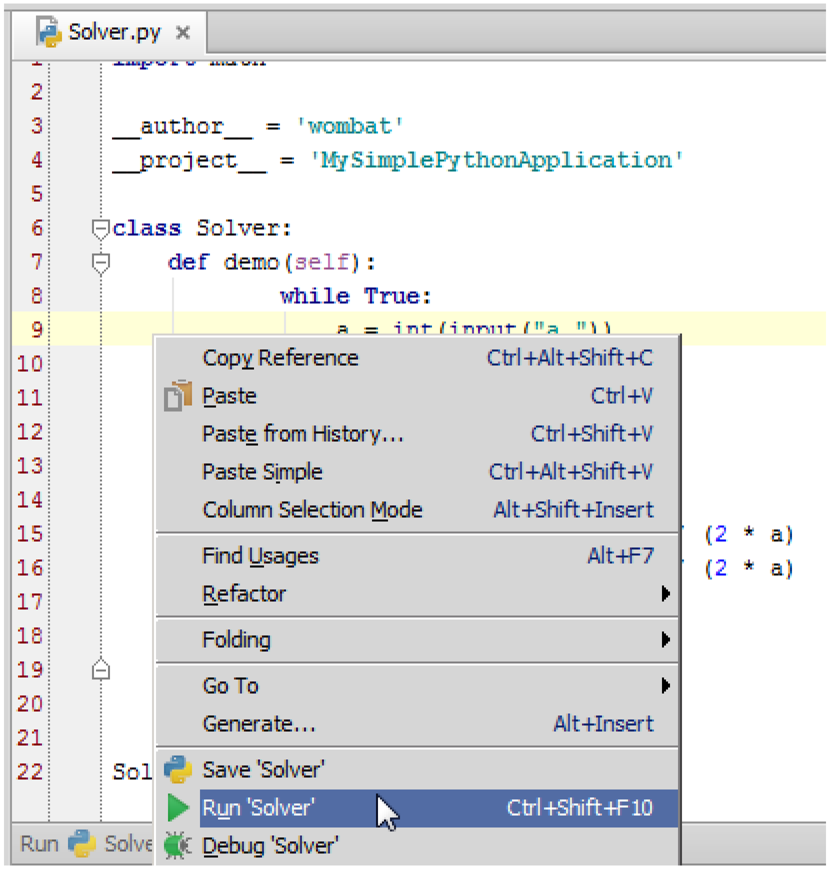
```
import math

_author_ = 'wombat'
_project_ = 'MySimplePythonApplication'

class Solver:
 def demo(self):
  while True:
   a = int(input("a "))
   b = int(input("b "))
   c = int(input("c "))
   d = b ** 2 - 4 * a * c
   if d>=0:
    disc = math.sqrt(d)
    root1 = (-b + disc) / (2 * a)
    root2 = (-b - disc) / (2 * a)
    print(root1, root2)
   else:
    print('error')

Solver().demo()
```

Next, let's run and debug this script.

Running application

You have already launched the Solver script with the keyboard shortcut, so let's just remind how it's done. PyCharm suggests several ways to run a script opened in the editor.

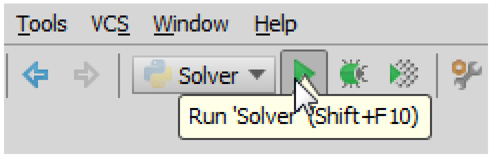
- First, you can use the keyboard shortcut Ctrl+Shift+F10
- Second, you can use the context menu command, invoked by right-clicking on the editor background:



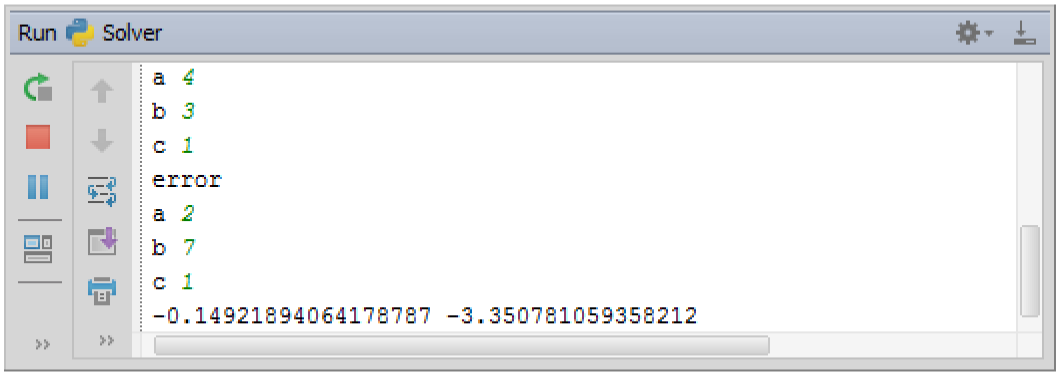
```
import math

__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver:
    def demo(self):
            while True:
                a = int(input("a "))
```

Solver.py
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Paste from History... Ctrl+Shift+V
Paste Simple Ctrl+Alt+Shift+V
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Go To
Generate... Alt+Insert
Save 'Solver'
Run 'Solver' Ctrl+Shift+F10
Debug 'Solver'
Run Solver

- Finally, it is possible to run a script from the main toolbar, using the temporary run/debug configuration 'Solver' (the notion of a run/debug configuration will be considered in more detail in the next section):



In either case, PyCharm opens Run tool window, and shows the application input and output:

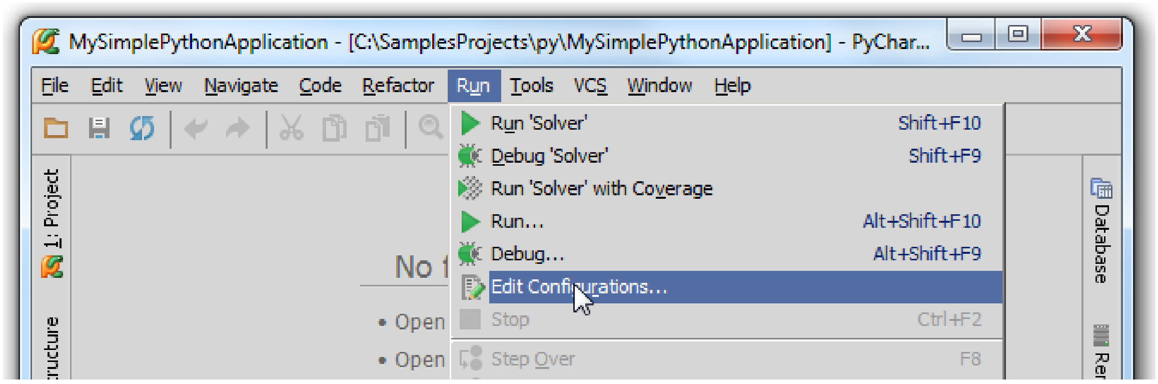


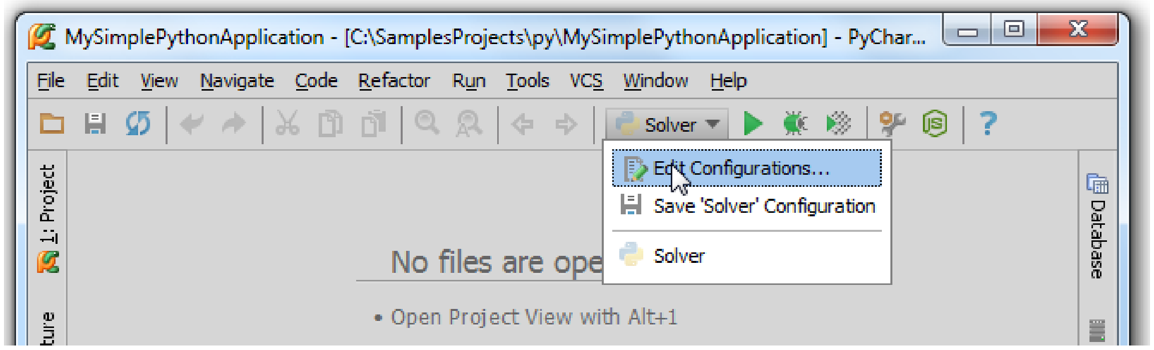
```
a 4
b 3
c 1
error
a 2
b 7
c 1
-0.14921894064178787 -3.350781059358212
```

Run/debug configuration

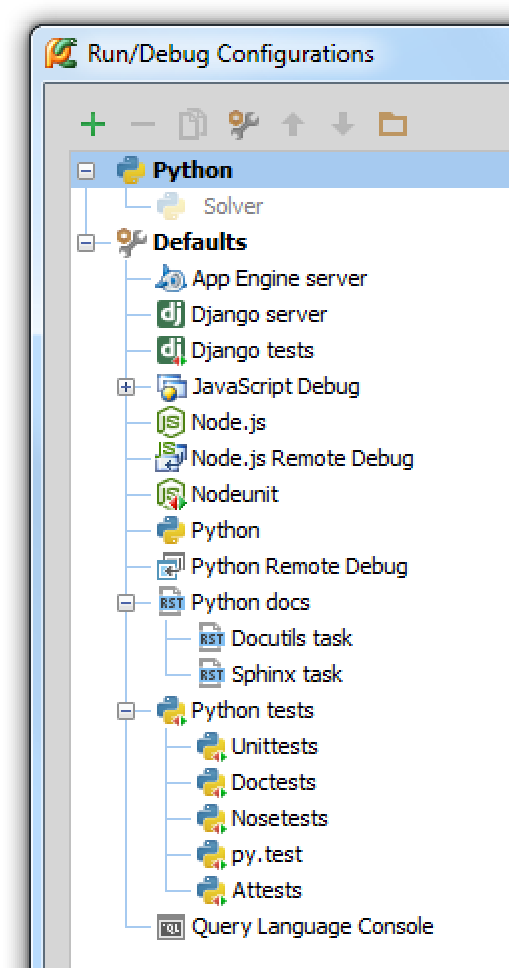
Each script is executed using a special profile, or a run/debug configuration. Such a profile is used for both running and debugging applications, and specifies the script name, working directory, actions to be performed before launch, etc.

PyCharm suggests a number of default run/debug configurations for the various types of applications (Python scripts, Django applications, tests, etc.) You can view the available defaults in the
Run/Debug Configurations dialog, which is invoked either by RunEdit Configurations... command on the main menu, or by clicking the drop-down list in the Run area of the main toolbar:


MySimplePythonApplication - [C:\SamplesProjects\py\MySimplePythonApplication] - PyChar...
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Run 'Solver' Shift+F10
Debug 'Solver' Shift+F9
Run 'Solver' with Coverage
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Edit Configurations...
Stop Ctrl+F2
Step Over F8
1: Project
Database


MySimplePythonApplication - [C:\SamplesProjects\py\MySimplePythonApplication] - PyChar...
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Solver
Edit Configurations...
Save 'Solver' Configuration
Solver
No files are ope
Open Project View with Alt+1
1: Project
Database

Let's look at the Edit Configurations dialog more attentively. Its left-hand part contains a tree view with two top-level nodes: Python and Default:



The lower node contains the list of default run/debug configurations. These default run/debug configurations are nameless, but each new run/debug configuration is created on the grounds of a default one, and gets the name of your choice.

The upper node is called Python and contains just one run/debug configuration *Solver*, which is shown grayed out. What does it mean?

Run/debug configuration *Solver* is a temporary profile, which PyCharm has produced, when you've just run the Solver script. It resides under the node Python, since this run/debug configuration is created on the base of the default configuration of the Python type.

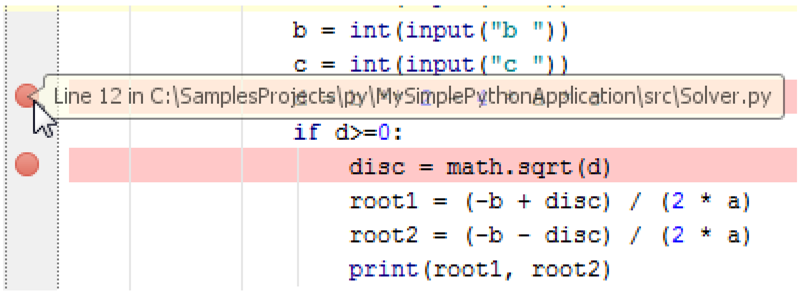
You can save this run/debug configuration and thus make it permanent. Permanent run/debug configurations are rendered in a normal font. Unlike temporary configurations, the number of permanent ones is unlimited.

Let's use the same temporary run/debug configuration *Solver* for debugging the Solver script.

Debugging application

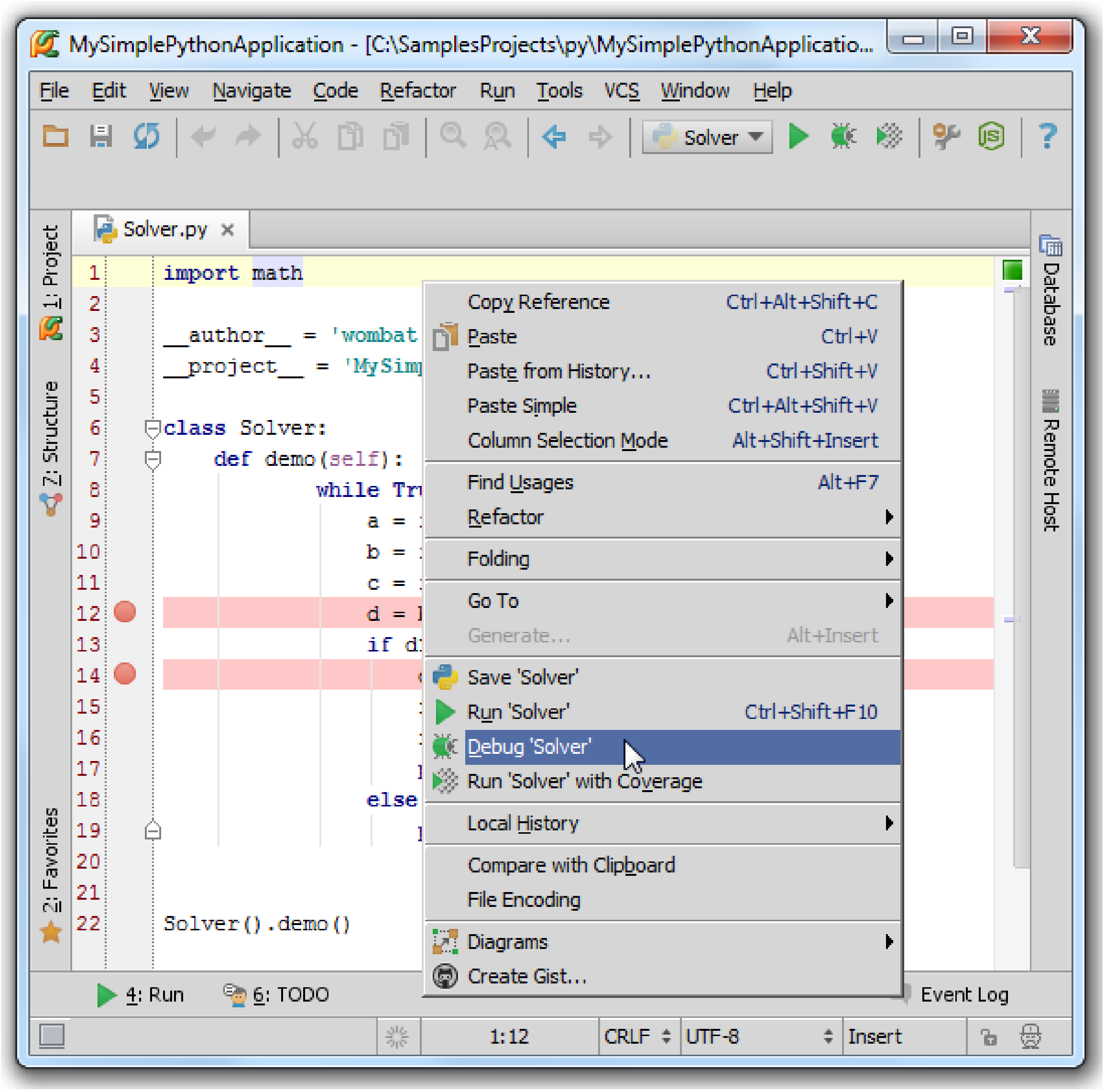
What will you do to execute your application step by step, examine program information related to variables, watches, or threads, find out the origin of exceptions? This is where the debugging process comes to help.

To start debugging, you have to set breakpoints first. To create a breakpoint, just click the left gutter:



```
        b = int(input("b "))
        c = int(input("c "))
Line 12 in C:\SamplesProjects\py\MySimplePythonApplication\src\Solver.py
        if d>=0:
            disc = math.sqrt(d)
            root1 = (-b + disc) / (2 * a)
            root2 = (-b - disc) / (2 * a)
            print(root1, root2)
```

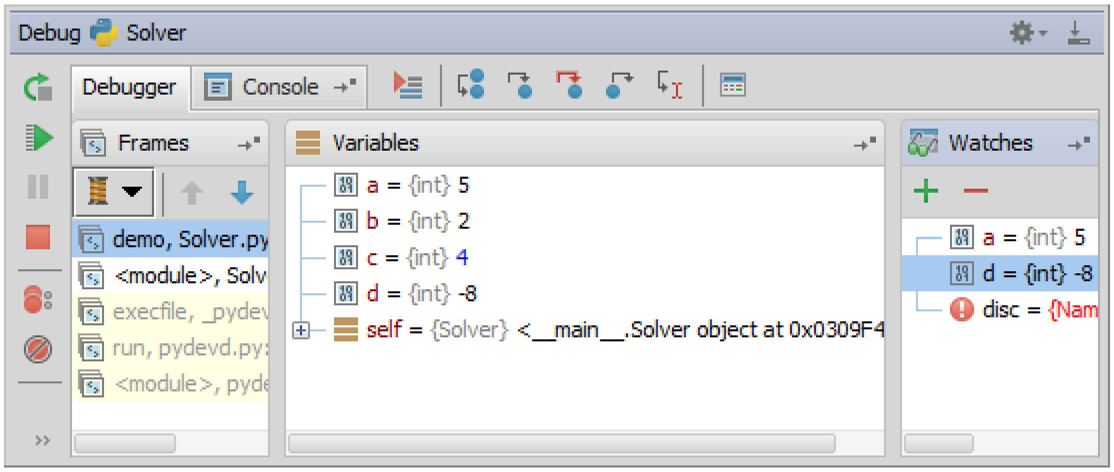
Next, right-click the editor background, and choose Debug 'Solver' on the context menu:



MySimplePythonApplication - [C:\SamplesProjects\py\MySimplePythonApplicatio...
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Solver
Solver.py
1: Project
7: Structure
2: Favorites
Database
Remote Host
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Paste from History... Ctrl+Shift+V
Paste Simple Ctrl+Alt+Shift+V
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Go To
Generate... Alt+Insert
Save 'Solver'
Run 'Solver' Ctrl+Shift+F10
Debug 'Solver'
Run 'Solver' with Coverage
Local History
Compare with Clipboard
File Encoding
Diagrams
Create Gist...
4: Run
6: TODO
Event Log
1:12 CRLF UTF-8 Insert

```
import math

__author__ = 'wombat
__project__ = 'MySim

class Solver:
    def demo(self):
        while Tr
            a = 
            b = 
            c = 
            d = 
            if d

            else

Solver().demo()
```

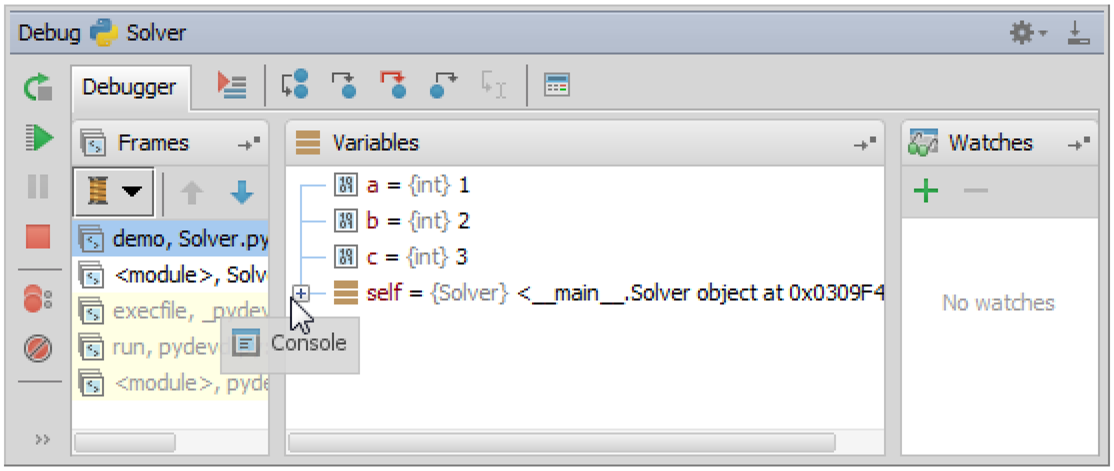
PyCharm starts the debugging session and shows the Debug tool window. The following image corresponds to the default layout of panes and tabs:



```
a = {int} 5
b = {int} 2
c = {int} 4
d = {int} -8
self = {Solver} <__main__.Solver object at 0x0309F4
```

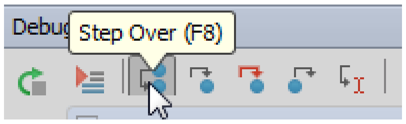
```
a = {int} 5
d = {int} -8
disc = {Nam
```

The Debug tool window shows dedicated panes for frames, variables, and watches, and the console, where all the input and output information is displayed. If you want the console to be always visible, just drag it to the desired place:

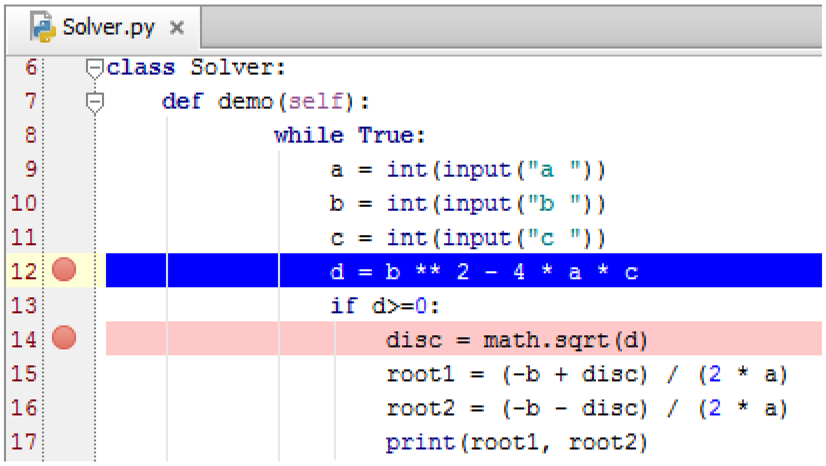


```
a = {int} 1
b = {int} 2
c = {int} 3
self = {Solver} <__main__.Solver object at 0x0309F4
```

Use the stepping toolbar buttons to step through your application:



As you step through the application, each reached breakpoint becomes blue:

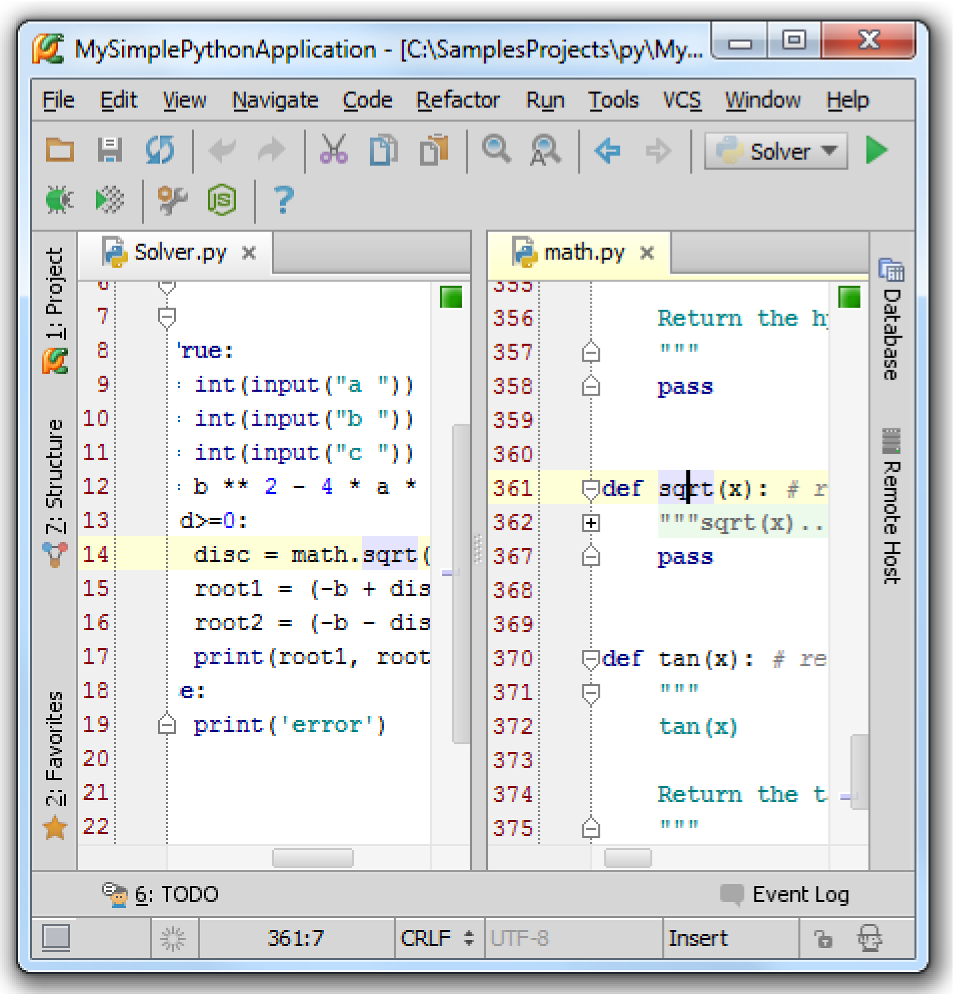


```
class Solver:
    def demo(self):
        while True:
            a = int(input("a "))
            b = int(input("b "))
            c = int(input("c "))
            d = b ** 2 - 4 * a * c
            if d>=0:
                disc = math.sqrt(d)
                root1 = (-b + disc) / (2 * a)
                root2 = (-b - disc) / (2 * a)
                print(root1, root2)
```

Exploring navigation

Navigation gives a special zest to PyCharm. Let us have a brief look over just some of the numerous PyCharm's navigation facilities.

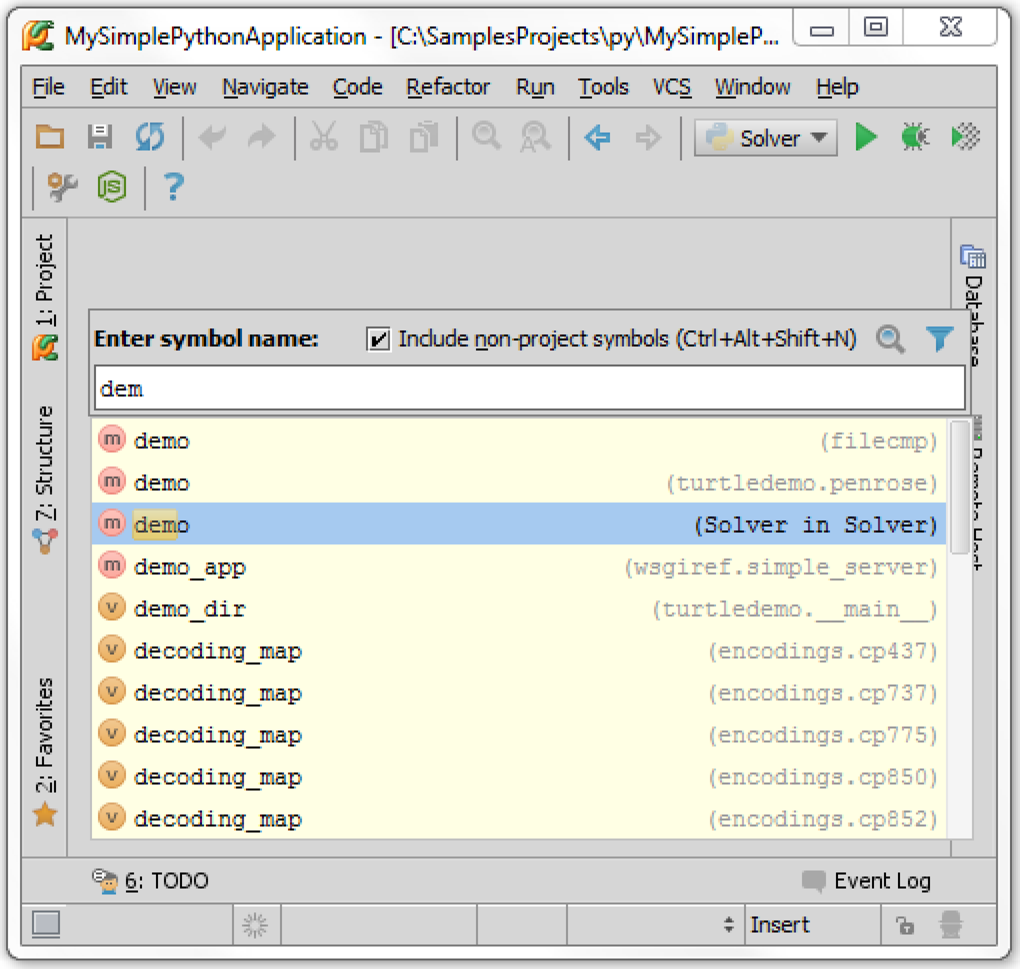
- Imagine that you have stopped working and went out for some coffee... and when you come back, you don't remember what exactly have you been doing and where have you stopped. In such a situation you can use one of the most needed features - navigation to last edit location. Press Ctrl+Shift+Backspace - and here you are!
- With PyCharm, it's quite easy to go to a declarati ol. For example, in our c e the caret at the call to sqrt function, and press Ctrl+B - PyCharm immediately opens math.py at the declaration of the sqrt function:



```
'rue:
: int(input("a "))
: int(input("b "))
: int(input("c "))
: b ** 2 - 4 * a *
d>=0:
 disc = math.sqrt(
 root1 = (-b + dis
 root2 = (-b - dis
 print(root1, root
e:
 print('error')
```

```
        Return the h
        """
        pass


def sqrt(x): # r
    """sqrt(x)..
    pass


def tan(x): # re
    """
    tan(x)

    Return the t
    """
```

- Very useful is the ability to quickly find any class, file, or symbol by name. For example, press Ctrl+Alt+Shift+N and type the name of a symbol you want to go to:


MySimplePythonApplication - [C:\SamplesProjects\py\MySimpleP...
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Solver
1: Project
7: Structure
2: Favorites
Database
6: TODO
Event Log
Insert

Enter symbol name: ☑ Include non-project symbols (Ctrl+Alt+Shift+N)

dem

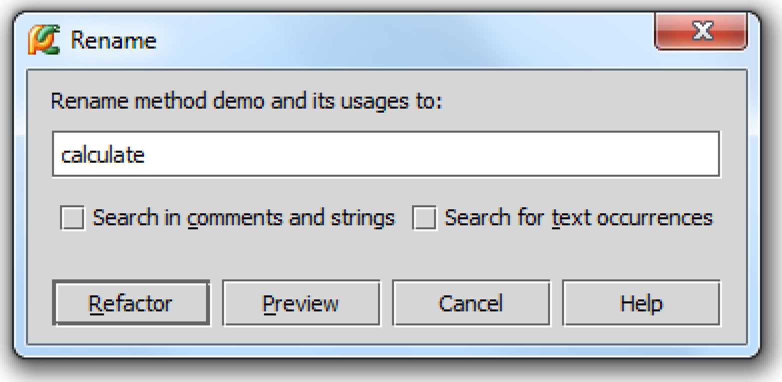
demo	(filecmp)
demo	(turtledemo.penrose)
demo	(Solver in Solver)
demo_app	(wsgiref.simple_server)
demo_dir	(turtledemo.__main__)
decoding_map	(encodings.cp437)
decoding_map	(encodings.cp737)
decoding_map	(encodings.cp775)
decoding_map	(encodings.cp850)
decoding_map	(encodings.cp852)

You can find all the available navigation commands under the Navigate menu. Described in this tutorial are just few examples... Learn more here.

Refactoring

Let's change the name of the function `demo`, and give it some more descriptive name, say, `calculate`. It is quite possible just to overtype an existing name with a new one. However, in this case, you will have to type new name twice: first time for the function declaration, and second time for the function call. In this small example it is not a problem, but consider working on a large project, with numerous function calls... It is much more advisable to use rename refactoring instead.

Place the caret at the function declaration, press Shift+F6, and then type the new name in the Rename dialog:

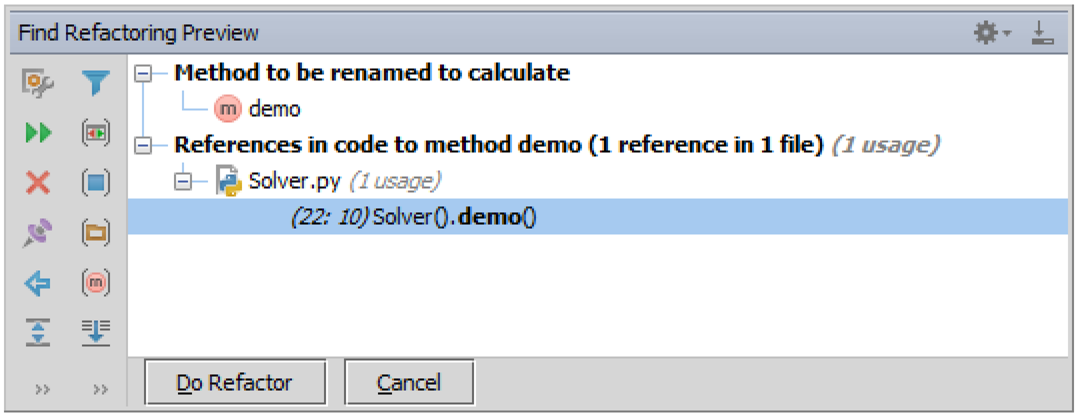


Rename method demo and its usages to:

calculate

☐ Search in comments and strings ☐ Search for text occurrences

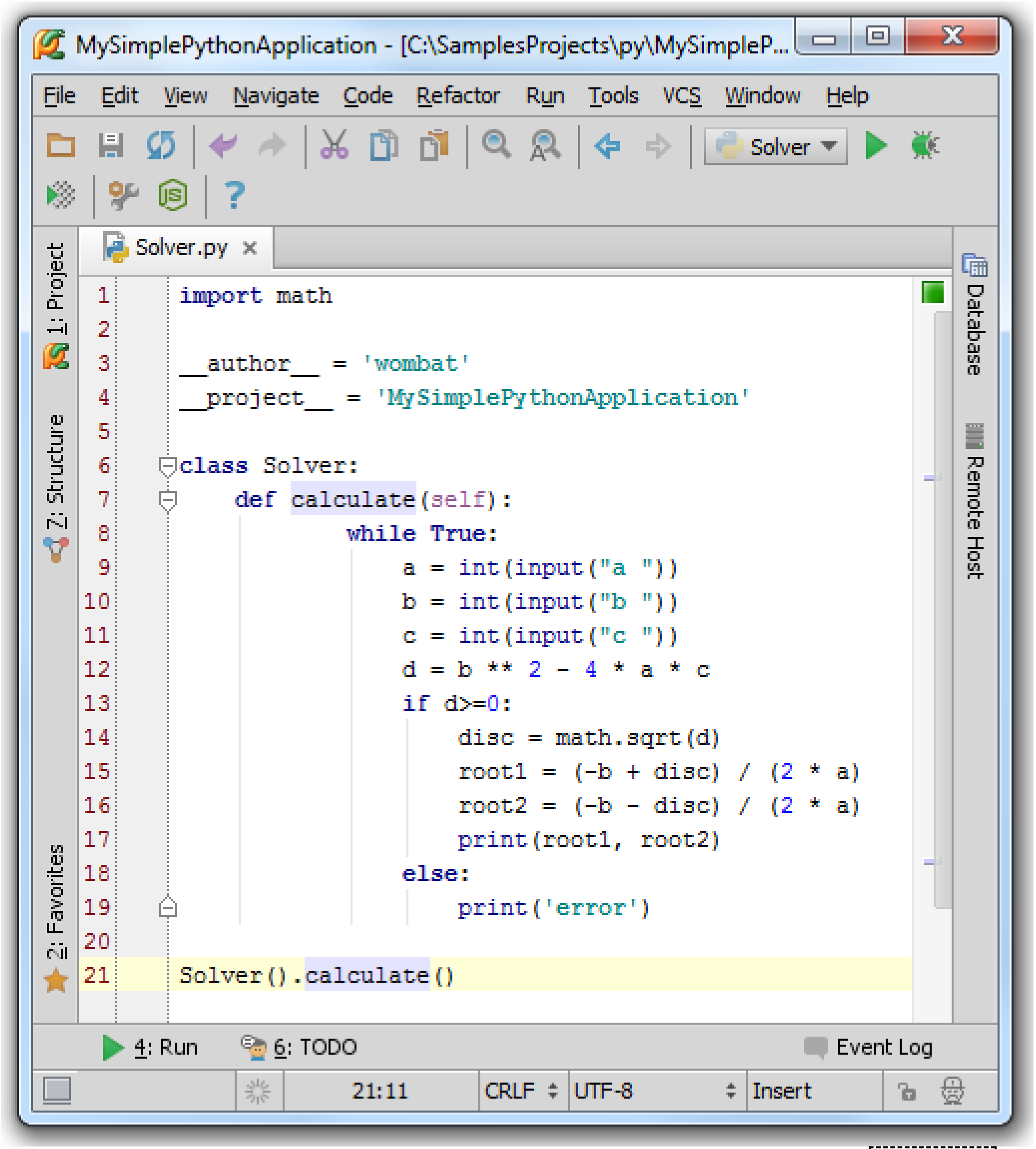
Click Refactor. All the found occurrences appear in the Find tool window:



Method to be renamed to calculate
- demo

References in code to method demo (1 reference in 1 file) *(1 usage)*
- Solver.py *(1 usage)*
  - *(22: 10)* Solver().**demo()**

Click Do Refactor button - you see that the function name has been changed for both the function declaration and the function call:



```
import math

__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver:
    def calculate(self):
        while True:
            a = int(input("a "))
            b = int(input("b "))
            c = int(input("c "))
            d = b ** 2 - 4 * a * c
            if d>=0:
                disc = math.sqrt(d)
                root1 = (-b + disc) / (2 * a)
                root2 = (-b - disc) / (2 * a)
                print(root1, root2)
            else:
                print('error')

Solver().calculate()
```

It's possible to modify this class further: move it to a different folder, change signature of the calculate function, extract variables etc. All these actions are performed by means of the various refactorings. We'll consider these refactorings in more detail in the dedicated tutorials.

Summary

So, this brief tutorial is over. Let's summarize our achievements:

- We have created a project
- We have populated this project with a directory and a class
- We have run and debugged our application
- We have explored shortly navigation and refactoring.

Congrats! Find more information in the PyCharm documentation, in blogs, and in tutorials.